AFRL-RI-RS-TR-2008-289
Final Technical Report
November 2008

CASTOR: WIDELY DISTRIBUTED SCALABLE INFOSPACES

Cornell University

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.

STINFO COPY

AIR FORCE RESEARCH LABORATORY
INFORMATION DIRECTORATE
ROME RESEARCH SITE
ROME, NEW YORK

NOTICE AND SIGNATURE PAGE

Using Government drawings, specifications, or other data included in this document for any purpose other than Government procurement does not in any way obligate the U.S. Government. The fact that the Government formulated or supplied the drawings, specifications, or other data does not license the holder or any other person or corporation; or convey any rights or permission to manufacture, use, or sell any patented invention that may relate to them.

This report was cleared for public release by the 88th ABW, Wright-Patterson AFB Public Affairs Office and is available to the general public, including foreign nationals. Copies may be obtained from the Defense Technical Information Center (DTIC) (http://www.dtic.mil).

AFRL-RI-RS-TR-2008-289 HAS BEEN REVIEWED AND IS APPROVED FOR PUBLICATION IN ACCORDANCE WITH ASSIGNED DISTRIBUTION STATEMENT.

FOR THE DIRECTOR:

/s/	/s/
GENNADY STASKEVICH	JAMES W. CUSACK, Chief
Work Unit Manager	Information Systems Division
	Information Directorate

This report is published in the interest of scientific and technical information exchange, and its publication does not constitute the Government's approval or disapproval of its ideas or findings.

REPORT DOCUMENTATION PAGE	*Form Approved* **OMB No. 0704-0188**

Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden to Washington Headquarters Service, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188) Washington, DC 20503.
PLEASE DO NOT RETURN YOUR FORM TO THE ABOVE ADDRESS.

1. REPORT DATE *(DD-MM-YYYY)* NOV 2008	**2. REPORT TYPE** Final	**3. DATES COVERED** *(From - To)* Feb 06 – Sep 08

4. TITLE AND SUBTITLE CASTOR: WIDELY DISTRIBUTED SCALABLE INFOSPACES	**5a. CONTRACT NUMBER** FA8750-06-2-0060
	5b. GRANT NUMBER
	5c. PROGRAM ELEMENT NUMBER 62702F
6. AUTHOR(S) Ken Birman	**5d. PROJECT NUMBER** ICED
	5e. TASK NUMBER 06
	5f. WORK UNIT NUMBER 04
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Cornell University 373 Pine Tree Road Ithaca, NY 14853	**8. PERFORMING ORGANIZATION REPORT NUMBER** N/A
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES) AFRL/RISB 525 Brooks Rd. Rome NY 13441-4505	**10. SPONSOR/MONITOR'S ACRONYM(S)** N/A
	11. SPONSORING/MONITORING AGENCY REPORT NUMBER AFRL-RI-RS-TR-2008-289

12. DISTRIBUTION AVAILABILITY STATEMENT
APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED. PA# 88ABW-2008-0920

13. SUPPLEMENTARY NOTES

14. ABSTRACT
The Cornell University Castor project was funded over a two-year period to create new technical options aimed at demanding event-notification settings. Applications of this type are common in warfighting and other DoD settings but currently are limited by inadequate scalability, reliability and poor performance.
Castor accomplished all goals originally stated in the SOW and in fact went beyond expectations. The team delivered a wide range of cutting-edge solutions, some of which are finding rapid uptake by major AF technology vendors. Our work has received keen interest from the very highest levels of industry, including CTO-level staff at the Air Force itself as well as Intel, Microsoft, Amazon, Red Hat, Cisco and we are collaborating closely with these and other vendors, including IBM and Raytheon. The final status report summarizes accomplishments and includes copies of some of the major publications by our group. Much of the software we developed is available for download from Cornell, as is a video demonstration of the Live Objects technology, which was briefed to the AF CTO, Mr. Kent Werner, in Spring 2008.

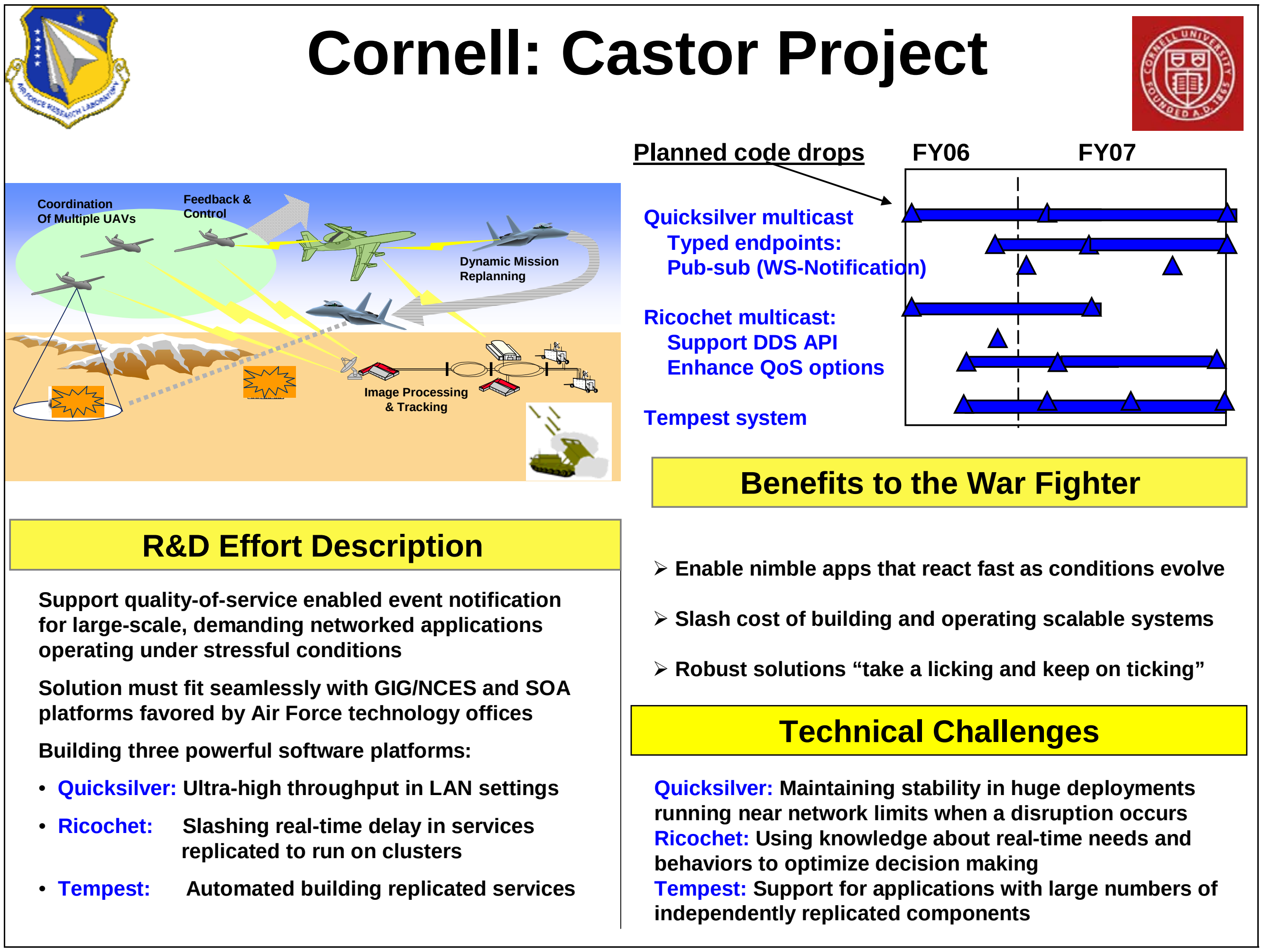
15. SUBJECT TERMS
Datacenters, Data Integrity, Distributed Systems, Event Notification, Multicast, Publish-Subscribe, Group Communication, Replication, Total Ordering.

16. SECURITY CLASSIFICATION OF:			**17. LIMITATION OF ABSTRACT**	**18. NUMBER OF PAGES**	**19a. NAME OF RESPONSIBLE PERSON** Gennady R. Staskevich
a. REPORT U	**b. ABSTRACT** U	**c. THIS PAGE** U	UU	227	**19b. TELEPHONE NUMBER (*Include area code)*** N/A

Standard Form 298 (Rev. 8-98)
Prescribed by ANSI Std. Z39.18

TABLE OF CONTENTS

Progress against Planned Objectives


Coordination Of Multiple UAVs
Feedback & Control
Dynamic Mission Replanning
Image Processing & Tracking
FY06
FY07

Cornell: Castor Project

Planned code drops

Quicksilver multicast
Typed endpoints:
Pub-sub (WS-Notification)

Ricochet multicast:
Support DDS API
Enhance QoS options

Tempest system

Benefits to the War Fighter

- Enable nimble apps that react fast as conditions evolve
- Slash cost of building and operating scalable systems
- Robust solutions "take a licking and keep on ticking"

R&D Effort Description

Support quality-of-service enabled event notification for large-scale, demanding networked applications operating under stressful conditions

Solution must fit seamlessly with GIG/NCES and SOA platforms favored by Air Force technology offices

Building three powerful software platforms:

- Quicksilver: Ultra-high throughput in LAN settings
- Ricochet: Slashing real-time delay in services replicated to run on clusters
- Tempest: Automated building replicated services

Technical Challenges

Quicksilver: Maintaining stability in huge deployments running near network limits when a disruption occurs
Ricochet: Using knowledge about real-time needs and behaviors to optimize decision making
Tempest: Support for applications with large numbers of independently replicated components

This final report overviews the Castor project, funded by AFRL and performed at Cornell University under the direction of Professor Kenneth P. Birman.

The original quad-chart associated with the SOW for the effort is reproduced above. Note that Castor was originally proposed as a 2 year effort running in FY 06 and FY 07. The ultimate effort was a bit longer and started later than was originally intended and finished only in September of 2008 (FY 08); the timeline is thus inaccurate. However, despite the shift in timeline, all objectives for the effort were fully achieved. Indeed, not only did we accomplish our SOW objectives, but we actually pushed beyond the originally stated goals, achieving major advances in several areas that relate directly to our planned effort, but go beyond what was initially expected.

As can be appreciated from the quad chart, the Castor effort had three primary sub-efforts: Quicksilver, Ricochet and Tempest. Each resulted in numerous publications, many in prestigious venues and winning best-paper awards or other recognition in several cases. Our work gained tremendous visibility in industry, attracting not just interest from such companies as IBM, Red Hat, Microsoft, Intel and Cisco, but also more measurable signs of impact. For example, Cornell is negotiating to contribute the Ricochet technology to the open-source Red Hat Linux community under a no-fee license. Microsoft, Cisco and Intel became so interested in our Quicksilver and Live Objects work that both companies briefed their CTOs, and we are now receiving funding from Cisco and Intel to encourage continued work on this topic. This work was also briefed to the CIO and CTO of the Air Force, and there has been talk of showing it to the Secretary of the Air Force in conjunction with a possible future "demos day" being discussed by the CIO's office. Thus, our work is gaining the kind of high-level attention that translates to real influence over time.

Castor also helped us strengthen our dialog with both AFRL, the AF CIO's office, AFOSR, and even other government agencies including OSD/DDS&T, NSF, DARPA and Treasury. The visibility Castor helped us achieve gave us credibility, and we used that to organize workshops and provide other forms of consulting expertise to these organizations. Castor even helped us shape the AF-TRUST "PRET", funded by AFOSR under the supervision of Bob Bonneau, in ways that focused on AFRL needs as we came to understand them through our studies and dialog with AFRL partners.

We've also enjoyed a vigorous and fruitful dialog with the standards communities in our areas, notably the Web Services community and the Autonomic Computing community. A proposal of ours for an enhanced Web Services eventing standard has received strong support from key players within the industry, and was published in the International Journal of Web Services and Systems.

The majority of this report will be fairly technical, although because we've included the actual papers published by the team, we won't drill down to the point of reproducing material that appears in the appendices. However, before doing so, we offer an executive summary of accomplishments under the project:

- **Our Ricochet event notification protocols** [NSDI 2007] achieve two to three orders of magnitude latency improvement in mission-critical event notification scenarios. This is the technology that the Linux community now hopes to standardize as the basis for a new generation of scalable, reliable, ultra-fast event notification in Linux data centers.
- **Maelstrom, a spin-off from Ricochet, offers a powerful new option for interconnecting data centers over WAN links**, including WAN links that have high delay-throughput products and experience some packet loss. Such links traditionally bedevil TCP users, who see performance collapse. Maelstrom [NSDI 2008] is totally transparent, requiring no application changes of any kind at all, and demonstrates dramatic performance improvements. Indeed, WAN links can often be completely hidden from applications.
- **The Smoke and Mirrors File System**, also a spin-off from our work on Ricochet and Maelstrom, shows that when these kinds of solutions are available, one can build high-performance scalable cluster file systems that can be mirrored transparently in real-time with no performance impact at all even when WAN latencies become very large.

- Our **Quicksilver Scalable Multicast** protocol, QSM, is setting performance and scalability records for supporting publish-subscribe applications in enterprise LAN settings that may include tens of thousands of applications. QSM won a best paper award at NCA 2008 and has drawn the attention of IBM. QSM technology now seems likely to play a big role in IBM's next generation Distributed Communications System for Web Services (DCS for WebSphere).
- We're very excited about our work on **Live Objects** [ECOOP 2008], a promising new "edge" computing concept that facilitates creation and exchange of rich, dynamic content in collaborative settings. Using popular web standards that extend the Microsoft and Linux web services platforms, live objects permit users to create live applications as mashups that combine content from various sources (such as images, videos, maps, weather forecasts, locations of vehicles or participants, etc). The overall goal is to support a range of applications that would include collaboration, medical consultations, gaming, and social networking. A tremendous number of military applications can be identified: the ability to quickly assemble applications by pulling information from anywhere, anytime, has long been a major goal for military information systems.
- **Tempest** [DSN 2008], our technology for automating the creation of scalable, robust Web Services, is inspiring work on a new generation of tools by vendors in which the difficult and error-prone steps of replicating data and control will be automated and standardized.

These highlights are just a glimpse of the research output of the group, which was extremely productive during the funding period. In the remainder of our report, we limit our focus to the work directly funded by AFRL under the Castor SOW. However, we do include copies of papers on some of the other work that occurred at Cornell during the same period and were able to achieve more by leveraging the Castor work, such as **Fireflies, SecureStream, Nightwatch, Antiquity, Dr. Multicast** and other systems.

AFRL encourages cross-cutting collaboration among funded organizations. In our case, we formed strong ties to the AFRL-sponsored team at Vanderbilt, headed by Doug Schmidt, the inventor who created the ACE/TAO Corba ORB and helped define the DDS standard for data dissemination in multicast settings. Schmidt's group is renowned for its emphasis on real-time applications, and by teaming with him and his effort, Cornell has leveraged their expertise without needing to create a duplicative and hence inefficient structure.

Other notable accomplishments include:

- Birman provided consulting help to the Air Force and to AFRL on many occasions. Most recently, these included organizing the Workshop on Managing Risks of Homogeneity, conducted at request of the AF-CIO, and running the RAPID study of DMO platforms and their communication needs.
- Birman also assisted the AF, AFOSR, DARPA, OSD/DDS&T, NSF, the White House OSTP, DHS and Treasury, from time to time, with policy workshops, targeted studies, and general advising on critical infrastructure protection challenges.
- Birman and Van Renesse delivered more than a dozen keynote talks, invited symposia, and similarly prestigious presentations. These occurred in diverse settings including major conferences, smaller workshops, commercial meetings organized by Microsoft,

Cisco, Intel and others, and even international University lectureships in settings such as Israel and India. This type of connection has all sorts of unexpected benefits. For example, though his visit to India (to present work on Live Objects), Birman entered into a dialog with the Infosys Corporation that ultimately led to sponsorship by that company of the new ACM-Infosys Prize, which rewards work by young researchers of a level that could eventually lead to Turing Awards or Nobel Prizes. Infosys also helped convince Amazon to underwrite a major increase in the payout of the Turing Award itself.

- Birman was program-committee chair for SOSP 2005, and Van Renesse was PC chair for OSDI 2008. Team members also participated on a great many other program committees.
- Team members took faculty positions at the Air Force Institute of Technology (Hopkinson) and the Navy Graduate School (Adina Crainiceanu). Hakim Weatherspoon, an African American graduate student, completed a post-doc with the team at Cornell and has now accepted an Assistant Professor position in that department. Dr. Dan Freedman, an Army Intelligence officer who sought a Cornell PhD in physical modeling is now following in Weatherspoon's footsteps, joining the group as a post-doc, with the goal of pursuing an eventual career in research on topics of importance to the Air Force and the US government.
- Several graduate students completed their PhDs and all took positions in the United States with companies or in teaching/research settings.

Technical Accomplishments: Details

We now offer a somewhat more technical summary. As noted earlier, we are including copies of the many published papers that document our effort in detail, giving protocols, proofs, experimental findings, and written to achieve a high professional communication standard. It is not our goal to simply duplicate that content here. Instead, for each of our major accomplishments and some of the more important spin-off activities, we include a page in a standardized form that describes:

1. The AF context in which the problem arose.
2. The approach we adopted.
3. The accomplishments documented in our papers and embodied in our software.
4. Potential impact for the warflighter.
5. Expected technology transition path.

As noted, we limit this section to work directly funded under the Castor program.

RICOCHET

The AF context in which the problem arose. Increasingly many computing systems are adopting an event-notification design paradigm, in which object oriented components (such as GIG-compatible web services, CORBA services, J2EE services, etc) interact over event streams (often using publish-subscribe APIs). Event streams can be slow and may not scale particularly well, especially in demanding settings such as large data centers operating under stress and experiencing failures. *An event notification protocol is needed for time-critical applications where low-latency and reliability are vital.*

The approach we adopted. Ricochet invents a new *lateral error correction* approach in which receivers help one-another out. The scheme employs an XOR-based code that operates across multiple event streams in a way that detects and repairs loss as much as three orders of magnitude faster than in more standard COTS-based products. Ricochet was tested by a red-team supplied by DARPA and "won" the test.

The accomplishments documented in our papers and embodied in our software. The Ricochet protocol is such a dramatic advance that it attracted the attention of the Red Hat CTO, Carl Trieloff, who proposed that the Cornell work become the centerpiece of a proposed new Linux standard for scalable event notification in enterprise data centers (cloud computing systems). Work is now underway to integrate Ricochet into Red Hat Linux. We note that Red Hat is a major vendor to the Air Force and military.

Potential impact for the warflighter. Many weapons systems operate by having one component detect threats, another computing incoming tracks, and a third target the response (for example, by firing on an incoming missile). In such systems, the delays associated with event notification are the key factor in determining response time and hence effectiveness of the defense. Similar needs exist when a warship or aircraft sustains damage and must reconfigure to maintain vital functionality. Ricochet slashes delays and hence enables a new generation of faster, more responsive, more robust solutions. The benefits extend to other settings too, including medical systems (both military ones and civilian systems), financial systems, intelligence applications, etc.

Expected technology transition path. We are very pleased by the dialog with Red Hat, but are also exploring other options. Vanderbilt has adopted Ricochet as a component of their DDS platform, and is exploring applications in settings of importance to the Air Force and Navy in partnership with Raytheon. Microsoft has hired one of the main Ricochet developers, Mahesh Balakrishnan, who will explore applications within Windows Vista and future platforms. The Cornell work itself is public domain, open source, with no license fees.

Main Publications:
Ricochet: Lateral Error Correction for Time-Critical Multicast. Mahesh Balakrishnan, Ken Birman, Amar Phanishayee, and Stefan Pleisch. Proc NSDI 2007 (Cambridge, MA; April 2007).

PLATO:Predictive Latency-Aware Total Ordering. Mahesh Balakrishnan, Ken Birman, and Amar Phanishayee. In Proceedings of the SRDS 2006: (Leeds, UK. October 2006.)

QUICKSILVER SCALABLE MULTICAST (QSM)

The AF context in which the problem arose. Like Ricochet, QSM aims at improved support for event-stream applications. Ricochet focuses on time-critical needs and works by incurring overhead (as much as 20%) to slash latency. Quicksilver focuses on situations where throughput and scalability are the key goals, especially at extremely high data rates.

The approach we adopted. QSM employs a unique new hierarchical protocol in which a system is organized into a tree of "domains", within which peers (data receivers) help one-another recover lost packets so that the sender can blast data rapidly using an unreliable UDP multicast.

The accomplishments documented in our papers and embodied in our software. The QSM protocols are setting performance and scalability records and represent a dramatic advance relative to prior work. Our NCA 2008 paper won a best-paper award, and IBM is now working with Cornell to understand how QSM concepts can be migrated into their next generation distributed communication system for Web Sphere, DCS. The Web Services (GIG) standards community has been working with us to flesh out potential enhancements to WS-EVENTING and WS-NOTIFICATION based on our work.

Potential impact for the warflighter. QSM achieves extremely fast and robust data delivery under conditions that cripple many of the more common COTS protocols. Under settings when other solutions would collapse, QSM remains stable and hence the applications depending upon it remain healthy.

Expected technology transition path. Our dialog with IBM is well advanced, but Microsoft has also shown interest in this work.

Main Publications:

QuickSilver Scalable Multicast (QSM). Krzysztof Ostrowski, Ken Birman, Danny Dolev. 7th IEEE International Symposium on Network Computing and Applications (IEEE NCA 2008). Cambridge, MA. July 2008. *Best paper award.*

Extensible Architecture for High-Performance, Scalable, Reliable Publish-Subscribe Eventing and Notification. Krzysztof Ostrowski, Ken Birman, and Danny Dolev. International Journal of Web Services Research. Volume 4, Number 4, Pgs 15-58. October-December 2007.

LIVE OBJECTS

The AF context in which the problem arose. It is extremely difficult to implement collaboration, planning, or rapid-response applications today. As a result, when a team in the field needs a new form of planning or collaboration tool, *years* can elapse between conception of the tool and delivery of a solution. Our Live Objects technology is a breakthrough that slashes these delays. The approach enables a simple drag-and-drop methodology for building applications much as one creates a powerpoint slide. Usually, no programming skills are needed – just drag-and-drop. The solution can then be shared by exchanging files. Live Objects build on and leverage GIG-based (web services) standards.

The approach we adopted. Live Objects are an exciting new "edge" computing concept that facilitates creation and exchange of rich, dynamic content in collaborative settings. Using web standards based on the same Microsoft and Linux web services platforms favored by the Air Force for its standard system deployments, live objects permit users to create live applications as mashups that combine content from various sources (such as images, videos, maps, weather forecasts, locations of vehicles or participants, etc). The experience is a bit like what Tom Cruise does in the movie Minority Report, when he assembles a crime-fighting “tool” by gesturing and pulling content from various sources. The applications can then be shared, and all users of any single live application share the identical content and updates in real-time.

The accomplishments documented in our papers and embodied in our software. In 2007 we completed an initial implementation of Live Objects [ECOOP 2008] and developed a demo that illustrates creation of a search-and-rescue application based on the technology. Microsoft, Cisco and Intel became so interested in our Quicksilver and Live Objects work that both companies briefed their CTOs, and we are now receiving funding from Cisco and Intel to encourage continued work on this topic. This work was also briefed to the CIO and CTO of the Air Force, and there has been talk of showing it to the Secretary of the Air Force in conjunction with a possible future “demos day” being discussed by the CIO’s office. Thus, our work is gaining the kind of high-level attention that translates to real influence over time.

Potential impact for the warflighter. Live Objects realize a dream of anytime, anywhere information access and seamless application development in which the military has made an enormous investment. By making it possible for non-experts to create applications without writing new code and to share them using email or web pages, the solution breaks a huge barrier to rapid application creation and deployment. The strongly typed component interfaces reduce the risk of errors: our platform detects compatibility and security issues at design time.

Expected technology transition path. There are many possible technology transition options, but at this early stage, we view the work as more of an R&D prototype.

Main Publications:
Programming with Live Distributed Objects. Krzysztof Ostrowski, Ken Birman, Danny Dolev, and Jong Hoon Ahnn. 22nd European Conference on Object-Oriented Programming (ECOOP 2008). Cyprus. July 2008. ***Video of our demo available at http://quicksilver.cs.cornell.edu***.

MAELSTROM

The AF context in which the problem arose. When we connect applications to data centers located far away, even over the fastest WAN links (40 Gbit optical), some data loss is inevitable and high latencies are a fact of life. TCP collapses in such settings, hence data moves at a crawl, causing applications to fail. The individual or application in the field is unable to download needed data, even though the bandwidth is available.

The approach we adopted. Maelstrom uses a variant of the Ricochet protocol to achieve completely transparent TCP acceleration on WAN links. In effect, we can place a kind of appliance on each end of the network and TCP will "magically" run at full link speeds. Maelstrom does this using a new forward-error correction scheme that runs at incredibly high speeds with low costs, and allows lost packets to be recovered on the destination side of the link. When TCP would normally need to request retransmission of data from the sender, Maelstrom instead recovers the lost data instantly, hence TCP never chokes back. No changes are needed to applications: the scheme is completely transparent.

The accomplishments documented in our papers and embodied in our software. The Maelstrom concept, as published in NSDI 2008, caught the attention of Cisco, which is helping us continue to develop our solution and to test it in realistic WAN settings.

Potential impact for the warflighter. The dream of anytime, anywhere information access has been incredibly hard to implement in military networks, which often suffer packet loss and long latencies. If Maelstrom can be deployed widely, it represents an inexpensive work-around that could dramatically change the end-user experience of a client trying to access remote services, a very frustrating and slow process today.

Expected technology transition path. Our hope is that Cisco might adopt Maelstrom into a product line. Cornell is making the technology available in a public-domain, no-fee, open source form and we already have many potential users, mostly in the financial sector.

Main Publications:

Maelstrom: Transparent Error Correction for Lambda Networks. Mahesh Balakrishnan, Tudor Marian, Ken Birman, Hakim Weatherspoon, Einar Vollset. USENIX Symposium on Networked System Design and Implementation (NSDI 08). April 2008.

SMOKE AND MIRRORS FILE SYSTEM

The AF context in which the problem arose. Maelstrom focuses on communication over WAN links, but of course many applications don't really use TCP directly. The other major model is file sharing: applications access files directly over a link, or share files. The same brick wall that makes communication over WAN networks so hard today also shows up in file based applications, which tend to perform very poorly under such conditions.

The approach we adopted. We implemented a new cluster-based file system that runs over Maelstrom. Called the "Smoke and Mirrors File System" (SMFS), this technology provides continuous active mirroring of file updates over Maelstrom, using a new concept that we call "network sync." One key idea is that because Maelstrom achieves data loss rates even lower than that of commercial disks, one can "sync" data to the optical link by waiting until the FEC packets are underway, without needing remote acknowledgement from the remote storage system. For most purposes, network sync is adequate, and the performance benefit is dramatic. A second idea was to combine network sync with a log-structured file system: by doing so, writes can be "concentrated" at the head of the log, creating a data stream nicely matched to the performance and reliability properties of Maelstrom.

The accomplishments documented in our papers and embodied in our software. Maelstrom exhibits no performance loss at all when operating on WAN links, irrespective of the link latency. Papers on this are just now being completed, but the financial community is already excited by the work and Birman was asked to give an invited talk at the annual Napa meeting of the Financial Services Technology Consortium, FSTC, a group of more than 100 major banks and financial companies. SMFS offers a potential way to reduce the risk of a catastrophic disruption of the financial system in the event of a WMD event in a major financial center like New York: with SMFS, data can be mirrored continuously at a safe distance.

Potential impact for the warflighter. Like Maelstrom, the warfighter benefits by having better access to vital data with dramatically reduced delays. SMFS eliminates a frustrating obstacle to information sharing and information-based decision making.

Expected technology transition path. It is probably too soon to speculate about transition paths here, but Cisco has shown very strong interest in this work and is providing some funding to encourage continued effort in the area.

Main Publications:

In preparation

TEMPEST

The AF context in which the problem arose. As the reader of this report will have noticed, Cornell's work often exploits highly complex, technically subtle mechanisms. Using the resulting tools in applications may not always be easy, and this is reflected in a theme of our work: we created Live Objects to make it easy to incorporate live data into applications, we created SMFS to show how Maelstrom could be used by a real file system. Tempest helps the developer create web services (GIG) applications that exploit Ricochet for data replication.

The approach we adopted. The developer starts by building a non-replicated web service in a standard manner, using a vendor-supplied tool such as the application builder technology provided by Microsoft in their Indigo platform for Windows Vista. Tempest then automatically introduces replication, fault-tolerance and self-management logic in a transparent, simple way.

The accomplishments documented in our papers and embodied in our software. Tempest was presented at DSN 2008 and we are making the software available for free download.

Potential impact for the warflighter. Reduced cost of development for new web services, and simpler, more automated ways of introducing robustness, could reduce expense for the Air Force and create a more nimble, responsive vendor community.

Expected technology transition path. Tempest was created to show the community how easy it can be to introduce replication and robustness within the GIG model. Our work is really intended to encourage imitation and in so doing, to move the "bar" so that vendors will no longer be able to insist that creating scalable applications is a complex, secret, and expensive art. Only time will confirm or refute this perspective.

Main Publications:
Tempest: Soft State Replication in the Service Tier. Tudor Marian, Mahesh Balakrishnan, Ken Birman, Robbert van Renesse. Accepted to DSN-DCSS 2008: 38th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, Anchorage, AL. June 24-27, 2008

DR MULTICAST

The AF context in which the problem arose. Many kinds of "legacy" applications have underlying communication patterns based on multicast. Yet the hardware multicast mechanism, IPMC, is viewed as a very risky technology in systems that run on large-scale, heavily loaded data centers. The issue is that IPMC, when scaled up, can cause performance problems in the data center routers and network interface cards. Worse, some applications work perfectly well in smaller configurations, but become destabilized in large ones and behave chaotically, disrupting the entire data center.

The approach we adopted. With Dr. Multicast (the MCMD), we introduce a completely transparent module that intercepts IPMC operations and, under control of an acceptable use policy set by the operator, maps them to the most appropriate action. An application can be blacklisted from using IPMC, in which case multicast is done via UDP on a point-to-point basis. The MCMD can also do an optimized assignment under which limited numbers of IPMC groups are made available, and are assigned to the uses that will benefit most. Work continues on this project, which will eventually explore other sorts of mappings, for example by replacing IPMC calls with Ricochet or Quicksilver Scalable Multicast.

The accomplishments documented in our papers and embodied in our software. We have a working prototype and will report on it at HotNets 2008. A more complete version is being finished now and we're targeting NSDI 2008 for a paper.

Potential impact for the warflighter. IPMC is key to scalability in systems such as the DoD's DMO applications, which support training and scenario evaluation by scalable simulation. Yet data center operators fear the technology. By taming IPMC and giving operators the ability to quickly fix problems if they do arise, the MCMD allows us to reintroduce IPMC in settings where applications were being forced to use costly alternatives such as point-to-point communication or even overlays. We gain better performance and scalability, simplicity, and better administrative control. Moreover, a situation in which IPMC could effectively disrupt entire data centers is eliminated.

Expected technology transition path. IBM is helping us with this work, and the Linux Red Hat community is extremely interested in this work. We are hoping to see the MCMD follow a path similar to the one used for Ricochet.

Main Publications:
Dr. Multicast: Rx for Datacenter Communication Scalability. Ymir Vigfusson, Hussam Abu-Libdeh, Mahesh Balakrishnan, Ken Birman, Yoav Tock. HotNets VII: Seventh ACM Workshop on Hot Topics in Networks. October 6-7, 2008. Calgary, Canada.

ANTIQUITY

Antiquity is the first of several activities that leveraged Castor funding, but was not funded “primarily” by the Castor effort (primary funding was from NSF). At the request of Dr. Hillman, PM for Castor, we include brief mention of the work because it benefitted from Castor.

The AF context in which the problem arose. In many military systems, data must be stored and shared among machines that are challenged by very difficult mobility and connectivity issues, causing the systems to experience frequent disconnections and failures. Antiquity is a new file system developed by Hakim Weatherspoon, who joined the team as a post-doc in 2006 and is now a Cornell faculty member. The goal was to demonstrate robust file storage under the most challenging conditions imaginable.

The approach we adopted. Hakim employs a sophisticated form of “Byzantine Agreement” in conjunction with a new kind of “log structured” file system of his own implementation to replicate file data in a robust way. If replicas are lost or damaged, Antiquity automatically and transparently repairs the lost data and regenerates missing replicas.

The accomplishments documented in our papers and embodied in our software. Hakim’s Eurosys paper [Eurosys 2006] reports on a remarkable experiment in which file availability was maintained in a PlanetLab experimental setup subject to extreme levels of churn: machines came and went every few minutes, and more than 1/3 failed outright during the experiment. File integrity was maintained continuously and availability was restored rapidly when file replication dropped below the minimum because of failures.

Potential impact for the warflighter. Hakim presented his work at AFRL in summer of 2007, at a workshop. There was great interest in the solution. Attendees indicated that a technology of this sort would be of huge value in AF settings involving mobile platforms with challenging connectivity and communications scenarios.

Expected technology transition path. Antiquity is available as an open-source solution, but we see the primary “story” here as one of educating vendors by showing them a way to solve this problem that really works well under conditions of extreme churn.

Main Publications:
Antiquity: exploiting a secure log for wide-area distributed storage. Hakim Weatherspoon, Patrick Eaton, Byung-Gon Chun, John Kubiatowicz EuroSys 2006, Leuven, Belgium, April 2006.

FIREFLIES

Fireflies is another project that leveraged Castor funding, but was not funded “primarily” by the Castor effort (primary funding was from NSF and Intel). We report on it for completeness.

The AF context in which the problem arose. Many distributed systems depend upon some form of monitoring infrastructure that serves as a “scaffold” over which the system itself runs. Yet the automation of this scaffold creation task is often weak, hence systems often require a great deal of manual configuration. Needed are more automated configuration tools.

The approach we adopted. Fireflies is an effort to automate the self-configuration of many kinds of distributed systems by creating a self-organizing “overlay” that not only structures itself, but also repairs itself if damaged and can even resist attacks using Byzantine Agreement. The approach is very general and can support all sorts of complex distributed applications, and indeed we used it as a tool in building our own NightWatch system, described below.

The accomplishments documented in our papers and embodied in our software. A paper on the Fireflies system was presented at Eurosys 2006. Fireflies was fully implemented and evaluated extensively under a wide range of fault patterns.

Potential impact for the warflighter. Solutions like Fireflies are needed to show vendors methods for automating system configuration. Lacking them, too many applications break when one tries to use them in an unexpected or challenging environment. While Fireflies itself is more of a proof of concept, it works and the approach is carefully documented and supported by source code that vendors can obtain from us on request. We see it as teaching tool, showing the industry that we don’t need to accept the current situation. Systems can be made self-configuring even under extremely difficult conditions.

Expected technology transition path. We are hoping that industry products will adopt some of the ideas demonstrated by the Fireflies effort.

Main Publications:

Fireflies: Scalable Support for Intrusion-Tolerant Overlay Networks. Robbert van Renesse and Havard Johansen. EuroSys 2006, Leuven, Belgium, April 2006.

NIGHTWATCH

NightWatch also leveraged Castor funding, but was not funded “primarily” by the Castor effort (primary funding was from Microsoft and NSF). Again, we report on it for completeness.

The AF context in which the problem arose. Delivery of video and audio streams is probably the single most common need in military networks. Yet such streams are often disrupted in tactical environments where connectivity can be challenging and participants are highly mobile. But there is also need for such a system to monitor its own behavior, so as to adapt itself if some participant begins to misbehave, for example by slowing down or exhibiting high error rates.

The approach we adopted. Given a robust scaffold such as the one implemented by Fireflies, one can use it to implement robust applications. We used Fireflies to implement a streaming media delivery system (Secure Stream [JCC 07]), and then built Nightwatch, a supervisory control system that monitors the underlying video delivery application and adaptively adjusts parameters so that if a participating node falls behind or begins to malfunction, the disruption is limited and very local. Nightwatch embodies a more general concept for self-monitoring and adaptation with potential applications in a wide range of distributed systems, including web services that provide other forms of data.

The accomplishments documented in our papers and embodied in our software. The system was implemented and evaluated, and we published on the work in several venues. Maya Haridisan, the PhD student who did this work, has now joined Microsoft Research in Silicon Valley and will continue to work on robust media delivery applications and related topics.

Potential impact for the warflighter. Nightwatch offer the promise of greatly enhanced delivery of streaming media to soldiers in the field and to others working with mobile platforms under challenging communications conditions.

Expected technology transition path. We are hoping that industry products will adopt some of the ideas demonstrated by the Nightwatch effort.

Main Publications:

SecureStream: An Intrusion-Tolerant Protocol for Live-Streaming Dissemination. Maya Haridasan, Robbert van Renesse. Journal of Computer Communications. Special issue on Foundation of Peer-to-Peer Computing. Elsevier. 2007.

Enforcing Fairness in a Live-Streaming System. Maya Haridasan, Ingrid Jansch-Porto, Robbert van Renesse. Multimedia Computing and Networking (MMCN 2008), San Jose, CA.

Full Publications List

2008

Dr. Multicast: Rx for Datacenter Communication Scalability. Ymir Vigfusson, Hussam Abu-Libdeh, Mahesh Balakrishnan, Ken Birman, Yoav Tock. HotNets VII: Seventh ACM Workshop on Hot Topics in Networks. October 6-7, 2008. Calgary, Canada.

Bosco: One-Step Byzantine Aysnchronous Consensus. Yee Jiun Song, Robbert van Renesse. The 22nd International Symposium on Distributed Computing (DISC 2008), Arcachon, France, September, 2008.

QuickSilver Scalable Multicast (QSM). Krzysztof Ostrowski, Ken Birman, Danny Dolev. 7th IEEE International Symposium on Network Computing and Applications (IEEE NCA 2008). Cambridge, MA. July 2008.

Programming with Live Distributed Objects. Krzysztof Ostrowski, Ken Birman, Danny Dolev, and Jong Hoon Ahnn. 22nd European Conference on Object-Oriented Programming (ECOOP 2008). Cyprus. July 2008.

Supporting Timeliness and Reliability via DDS and Ricochet. Joe Hoffert, Douglas Schmidt, Mahesh Balakrishnan, Ken Birman. OMG Workshop on Distributed Object Computing for Real-time and Embedded Systems. July 2008, Washington DC.

SENSTRAC: scalable querying of sensor networks from mobile platforms using tracking-style queries. Stefan Pleisch and Ken Birman. Int. Journal Sensor Networks, Vol. 3, No. 4, 2008. pp. 266 - 280

Nysiad: Practical Protocol Transformation to Tolerate Byzantine Failures. Chi Ho, Robbert van Renesse, Mark Bickford, and Danny Dolev. USENIX Symposium on Networked System Design and Implementation (NSDI 08). San Francisco, CA. April 2008.

Gossip-based Distribution Estimation in Peer-to-Peer Networks. Maya Haridasan, Robbert van Renesse To Appear in Proceedings of The 7th International Workshop on Peer-to-Peer Systems (IPTPS '08). Tampa Bay, FL. February 25-26, 2008.

Tempest: Soft State Replication in the Service Tier. Tudor Marian, Mahesh Balakrishnan, Ken Birman, Robbert van Renesse. Accepted to DSN-DCSS 2008: 38th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, Anchorage, AL. June 24-27, 2008

The Building Blocks of Consensus. Yee Jiun Song, Robbert van Renesse, Fred B. Schneider, Danny Dolev. 9th International Conference on Distributed Computing and Networking (ICDCN '08), Kolkata, India. January, 2008.

Enforcing Fairness in a Live-Streaming System. Maya Haridasan, Ingrid Jansch-Porto, Robbert van Renesse. Multimedia Computing and Networking (MMCN 2008), San Jose, CA.

Maelstrom: Transparent Error Correction for Lambda Networks. Mahesh Balakrishnan, Tudor Marian, Ken Birman, Hakim Weatherspoon, Einar Vollset. USENIX Symposium on Networked System Design and Implementation (NSDI 08). April 2008.

2007

Sliver: A Fast Distributed Slicing Algorithm. Vincent Gramoli, Ymir Vigfusson, Ken Birman, Anne-Marie Kermarrec, Robbert van Renesse. Technical Report. December 2007

SecureStream: An Intrusion-Tolerant Protocol for Live-Streaming Dissemination. Maya Haridasan, Robbert van Renesse. Journal of Computer Communications. Special issue on Foundation of Peer-to-Peer Computing. Elsevier.

Declarative Reliable Multi-Party Protocols Krzysztof Ostrowski, Ken Birman, Danny Dolev. Cornell University Technical Report (TR2007-2088). April, 2007.

Implementing High-Performance Multicast in a Managed Environment Krzysztof Ostrowski, Ken Birman, Danny Dolev. Cornell University Technical Report (TR2007-2088). April, 2007.

Exploiting Gossip for Self-Management in Scalable Event Notification Systems. Ken Birman, Anne-Marie

Kermarrec, Krzysztof Ostrowski, Marin Bertier, Danny Dolev, Robbert Van Renesse. Distributed Event Processing Systems and Architecture Workshop (DEPSA). June 2007.

Live Distributed Objects: Enabling the Active Web Krzysztof Ostrowski, Ken Birman, Danny Dolev. To Appear in IEEE Internet Computing. Nov/Dec 2007.

Optimizing Power Consumption in Large Scale Storage Systems. Lakshmi Ganesh, Hakim Weatherspoon, Mahesh Balakrishnan and Ken Birman. To Appear in Proceedings of the 11th Workshop on Hot Topics in Operating Systems (HotOS XI). San Diego, CA. May 7-9, 2007.

Ricochet: Lateral Error Correction for Time-Critical Multicast. Mahesh Balakrishnan, Ken Birman, Amar Phanishayee, and Stefan Pleisch. To Appear in Proceedings of the 4th USENIX Symposium on Networked Systems Design & Implementation (NSDI 07). Cambridge, MA. April 2007.

Active and Passive Techniques for Group Size Estimation in Large-Scale and Dynamic Distributed Systems. Dionysios Kostoulas, Dimitrios Psaltoulis, Indranil Gupta, Ken Birman, Al Demers. To Appear in the Journal of Systems and Software, 2007.

Scalable Multicast Platforms for a New Generation of Robust Distributed Applications. Ken Birman, Mahesh Balakrishnan, Danny Dolev, Tudor Marian, Krzysztof Ostrowski, Amar Phanishayee. To Appear in Proceedings of the Second IEEE/Create-Net/ICST International Conference on Communication System software and Middleware (COMSWARE). Bangalore, India. January 7-12, 2007.

Extensible Architecture for High-Performance, Scalable, Reliable Publish-Subscribe Eventing and Notification. Krzysztof Ostrowski, Ken Birman, and Danny Dolev. To Appear in the International Journal of Web Services Research. Volume 4, Number 4, Pgs 15-58. October-December 2007.

SENSTRAC: Scalable Querying of SENSor Networks from Mobile Platforms Using TRACking-Style Queries. Stefan Pleisch and Ken Birman. To Appear in International Journal of Sensor Networks (IJSNet). 2007

2006

Scalable Publish-Subscribe in a Managed Framework. Krzysztof Ostrowski, Ken Birman. Cornell Technical Report (TR2007-2086). October, 2006.

The QuickSilver Properties Framework. Krzysztof Ostrowski, Ken Birman, Danny Dolev. Abstract, presented at the OSDI’06 poster session, Seattle, WA, November 2006.

A Scalable Services Architecture. Tudor Marian, Ken Birman, and Robbert van Renesse. To appear in Proceedings of the IEEE Symposium on Reliable Distributed Systems (SRDS 2006). Leeds, UK. October 2006.

Defense Against Intrusion in a Live Streaming Multicast System. Maya Haridasan, Robbert van Renesse. In Proceedings of the 6th IEEE International Conference on Peer-to-Peer Computing (P2P2006), Cambridge, UK, September 2006.

Properties Framework and Typed Endpoints for Scalable Group Communication. Krzysztof Ostrowski, Ken Birman, Danny Dolev. Cornell University Technical Report TR2006-2062 (July, 2006).

Scalable Group Communication System for Scalable Trust. Krzysztof Ostrowski, Ken Birman. In Proceedings of The First ACM Workshop on Scalable Trusted Computing (ACM STC 2006). Fairfax, VA. November 3, 2006.

PLATO:Predictive Latency-Aware Total Ordering. Mahesh Balakrishnan, Ken Birman, and Amar Phanishayee. In Proceedings of the SRDS 2006: 25th IEEE Symposium on Reliable Distributed Systems, Leeds, UK. October 2006.

Cognitive Adaptive Radio Teams. Richard Lau, Stephanie Demers, Yibei Ling, Bruce Siegell, Einar Vollset, Ken Birman, Robbert vanRenesse, Howie Shrobe, Jonathan Bachrach, Lester Foster. To Appear in Proceedings of the 2006 International Workshop on Wireless Ad-hoc and Sensor Networks, (IWWAN 2006). New York, NY. June 2006.

Network-Aware Adaptation Techniques for Mobile File Systems. Benjamin Atkin, Ken Birman. In Proceedings of the The 5th IEEE International Symposium on Network Computing and Applications (IEEE NCA06). Cambridge, MA. June 2006.

QuickSilver Scalable Multicast. Krzysztof Ostrowski, Ken Birman, and Amar Phanishayee. Cornell University Technical Report TR2006-2063 (April, 2006).
Reliable Multicast for Time-Critical Systems. Mahesh Balakrishnan and Ken Birman. In Proceedings of the First IEEE Workshop on Applied Software Reliability (WASR 2006), Philadelphia, PA. June 2006.
How the Hidden Hand Shapes the Market for Software Reliability. Ken Birman, Coimbatore Chandersekaran, Danny Dolev, and Robbert van Renesse. In Proceedings of the First IEEE Workshop on Applied Software Reliability, Philadelphia, PA. June 2006.
Extensible Web Services Architecture for Notification in Large-Scale Systems. Krzysztof Ostrowski and Ken Birman. In Proceedings of the 2006 IEEE International Conference on Web Services (ICWS 2006). Chicago, IL, September 2006.
A general algebra and implementation for monitoring event streams. A. Demers, J. Gehrke, M. Hong, M. Riedewald, and W. White. Technical report, Cornell University, 2005
Towards Expressive Publish/Subscribe Systems. Alan Demers, Johannes Gehrke, Mingsheng Hong, Mirek Riedewald, and Walker White. In Proceedings of the 10th International Conference on Extending Database Technology (EDBT 2006), Munich, Germany, March 2006.
Chunkyspread: Multi-tree Unstructured End System Multicast. Vidhyashankar Venkataraman, Paul Francis. IPTPS 2006, February 2006
On Heterogeneous Overlay Construction and Random Node Selection in Unstructured P2P Networks. Vivek Vishnumurthy, Paul Francis. IEEE INFOCOM 2006, April 2006
The Untrustworthy Services Revolution. Ken Birman. IEEE Computer (ISSN 0018-9162). Vol.39 No.2, Pgs. 98-100. February 2006.
Navigating in the Storm: Using Astrolabe to Adaptively Configure Web Services and Their Clients. Ken Birman, Robbert van Renesse, and Werner Vogels. Cluster Computing Special Issue: Autonomic Computing. (ISSN 1386-7857 (Paper) 1573-7543 (Online)). Volume 9, Number 2. Pgs. 127-139. April 2006.
Mistral: Efficient Flooding in Mobile Ad-hoc Networks. S. Pleisch, M. Balakrishnan, K. Birman, and R. van Renesse. In Proceedings of the Seventh ACM International Symposium on Mobile Ad Hoc Networking and Computing (ACM MobiHoc 2006). Florence, Italy May 2006.
Autonomic Computing - A System-Wide Perspective. Robbert van Renesse and Kenneth P. Birman. "Autonomic Computing: Concepts, Infrastructure, and Applications". Pgs. 35-48. ed. Manish Parashar and Salim Hariri, CRC press, January 2006.
SENSTRAC: Scalable Querying of SENSor Networks from Mobile Platforms Using TRACking-Style Queries. Stefan Pleisch and Ken Birman. To Appear in Proceedings of The Third IEEE International Conference on Mobile Ad-hoc and Sensor Systems. Vancouver, Canada. October 9-12, 2006.
Fireflies: Scalable Support for Intrusion-Tolerant Overlay Networks. Robbert van Renesse and Havard Johansen. EuroSys 2006, Leuven, Belgium, April 2006.

Ricochet: Lateral Error Correction for Time-Critical Multicast

Mahesh Balakrishnan[†], Ken Birman[†], Amar Phanishayee[‡], Stefan Pleisch[†]
[†]*Cornell University and* [‡]*Carnegie Mellon University*
{*mahesh,ken,pleisch*}*@cs.cornell.edu, amarp+@cs.cmu.edu*

Abstract

Ricochet is a low-latency reliable multicast protocol designed for time-critical clustered applications. It uses IP Multicast to transmit data and recovers from packet loss in end-hosts using Lateral Error Correction (LEC), a novel repair mechanism in which XORs are exchanged between receivers and combined across overlapping groups. In datacenters and clusters, application needs frequently dictate large numbers of fine-grained overlapping multicast groups. Existing multicast reliability schemes scale poorly in such settings, providing latency of packet recovery that depends inversely on the data rate within a single group: the lower the data rate, the longer it takes to recover lost packets. LEC is insensitive to the rate of data in any one group and allows each node to split its bandwidth between hundreds to thousands of fine-grained multicast groups without sacrificing timely packet recovery. As a result, Ricochet provides developers with a scalable, reliable and fast multicast primitive to layer under high-level abstractions such as publish-subscribe, group communication and replicated service/object infrastructures. We evaluate Ricochet on a 64-node cluster with up to 1024 groups per node: under various loss rates, it recovers almost all packets using LEC in tens of milliseconds and the remainder with reactive traffic within 200 milliseconds.

1 Introduction

Clusters and datacenters play an increasingly important role in the contemporary computing spectrum, providing back-end computing and storage for a wide range of applications. The modern datacenter is typically composed of hundreds to thousands of inexpensive commodity blade-servers, networked via fast, dedicated interconnects. The software stack running on a single blade-server is a brew of off-the-shelf software: commercial operating systems, proprietary middleware, managed run-time environments and virtual machines, all standardized to reduce complexity and mitigate maintenance costs.

The last decade has seen the migration of time-critical applications to commodity clusters. Application domains ranging from computational finance to air-traffic control and military communication have been driven by scalability and cost concerns to abandon traditional real-time environments for COTS datacenters. In the process, they give up conservative - and arguably unnecessary - guarantees of real-time performance for the promise of massive scalability and multiple nines of timely availability, all at a fraction of the running cost. Delivering on this promise within expanding and increasingly complex datacenters is a non-trivial task, and a wealth of commercial technology has emerged to support clustered applications.

At the heart of commercial datacenter software is *reliable multicast* — used by publish-subscribe and data distribution layers [5, 7] to spread data through clusters at high speeds, by clustered application servers [1, 4, 3] to communicate state, updates and heartbeats between server instances, and by distributed caching infrastructures [2, 6] to rapidly update cached data. The multicast technology used in contemporary industrial products is derivative of protocols developed by academic researchers over the last two decades, aimed at scaling metrics like throughput or latency across dimensions as varied as group size [10, 17], numbers of senders [9], node and network heterogeneity [12], or geographical and routing distance [18, 21]. However, these protocols were primarily designed to extend the reach of multicast to massive networks; they are not optimized for the failure modes of datacenters and may be unstable, inefficient and ineffective when retrofitted to clustered settings. Crucially, they are not designed to cope with the unique scalability demands of time-critical fault-tolerant applications.

We posit that a vital dimension of scalability for clustered applications is the *number of groups* in the system. All the uses of multicast mentioned above induce large numbers of overlapping groups. For example, a computational finance calculator that uses a topic-based pub-sub system to subscribe to a fraction of the equities on the stock market will end up belonging in many multicast groups. Multiple such applications within a datacenter - each subscribing to different sets of equities - can result in arbitrary patterns of group overlap. Similarly, data caching or replication at fine granularity can result in a single node hosting many data items. Replication driven by high-level objectives such as locality, load-balancing or fault-tolerance can lead to distinct overlapping replica sets - and hence, multicast groups - for each item.

In this paper, we propose Ricochet, a time-critical reliable multicast protocol designed to perform well in the multicast patterns induced by clustered applications. Ricochet uses IP Multicast [15] to transmit data and recovers lost packets using *Lateral Error Correction* (LEC), a novel error correction mechanism in which XOR repair packets are probabilistically exchanged between receivers and combined across overlapping multicast groups. The latency of loss recovery in LEC depends inversely on the aggregate rate of data in the system, rather than the rate in any one group. It performs equally well in any arbitrary configuration and cardinality of group overlap, allowing Ricochet to scale to massive numbers of groups while retaining the best characteristics of state-of-the-art multicast technology: even distribution of responsibility among receivers, insensitivity to group size, stable proactive overhead and graceful degradation of performance in the face of increasing loss rates.

1.1 Contributions

- We argue that a critical dimension of scalability for multicast in clustered settings is the number of groups in the system.

- We show that existing reliable multicast protocols have recovery latency characteristics that are inversely dependent on the data rate in a group, and do not perform well when each node is in many low-rate multicast groups.

- We propose Lateral Error Correction, a new reliability mechanism that allows packet recovery latency to be independent of per-group data rate by intelligently combining the repair traffic of multiple groups. We describe the design and implementation of Ricochet, a reliable multicast protocol that uses LEC to achieve massive scalability in the number of groups in the system.

- We extensively evaluate the Ricochet implementation on a 64-node cluster, showing that it performs well with different loss rates, tolerates bursty loss patterns, and is relatively insensitive to grouping patterns and overlaps - providing recovery characteristics that degrade gracefully with the number of groups in the system, as well as other conventional dimensions of scalability.

2 System Model

We consider patterns of multicast usage where each node is in many different groups of small to medium size (10 to 50 nodes). Following the IP Multicast model, a group is defined as a set of receivers for multicast data, and senders do not have to belong to the group to send to it. We expect each node to receive data from a large set of distinct senders, across all the groups it belongs to.

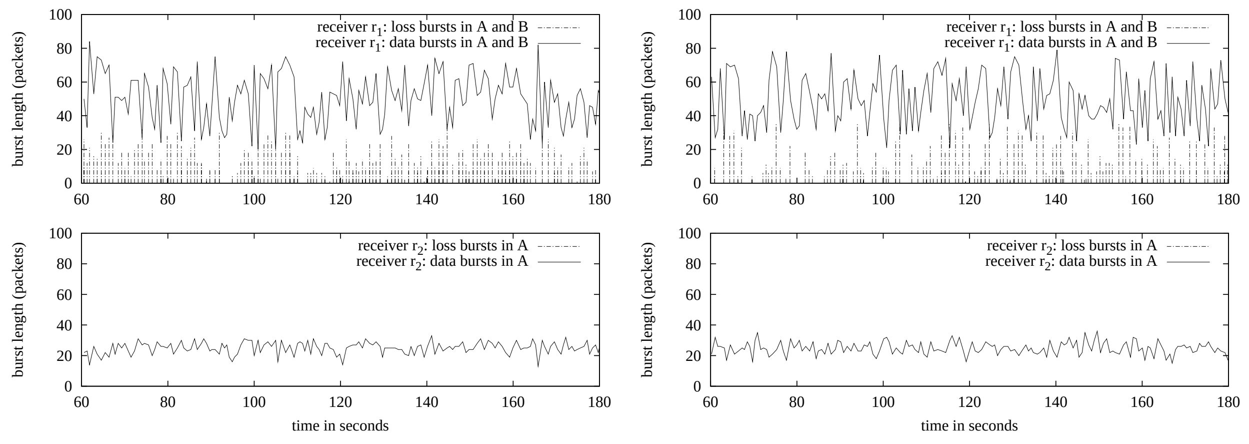
Where does Loss occur in a Datacenter? Datacenter networks have flat routing structures with no more than two or three hops on any end-to-end path. They are typically over-provisioned and of high quality, and packet loss in the network is almost non-existent. In contrast, datacenter end-hosts are inexpensive and easily overloaded; even with high-capacity network interfaces, the commodity OS often drops packets due to buffer overflows caused by traffic spikes or high-priority threads occupying the CPU. Hence, our loss model is one of short packet bursts dropped at the end-host receivers at varying loss rates.

Figure 1 strongly indicates that loss in a datacenter is (a) bursty and (b) independent across end-hosts. In this experiment, a receiver r_1 joins two multicast groups A and B, and another receiver r_2 in the same switching segment joins only group A. From a sender located multiple switches away on the network, we send per-second data bursts of around 25 1KB packets to group A and simultaneously send a burst of 0-50 packets to group B, and measure packet loss at both receivers. We ran this experiment on two networks: a 64-node cluster at Cornell with 1.3 Ghz receivers and the Emulab testbed at Utah with 2 Ghz receivers, all nodes running Linux 2.6.12.

The top graphs in Figure 1 show the traffic bursts and loss bursts at receiver r_1, and the bottom graphs show the same information for r_2. We can see that r_1 gets overloaded and drops packets in bursts of size 1-30 packets, whereas r_2 does not drop any packets — importantly, around 30% of the packets dropped by r_1 are in group A, which is common to both receivers. Hence, loss is both bursty and independent across nodes. Together, these graphs indicate strongly that loss occurs due to buffer overflows at receiver r_1.

The example in Figure 1 is simplistic - each incoming burst of traffic arrives at the receiver within a small number of milliseconds - but conveys a powerful message: it is very easy to trigger significant bursty loss at datacenter end-hosts. The receivers in these experiments were running empty and draining packets continuously out of the kernel, with zero contention for the CPU or the network, whereas the settings of interest to us involve time-critical, possibly CPU-intensive applications running on top of the communication stack.

Further, we expect multi-group settings to intrinsically exhibit bursty incoming traffic of the kind emulated in this experiment — each node in the system receives data from multiple senders in multiple groups and it is likely that the inter-arrival time of data packets at a node will vary widely, even if the traffic rate at one sender or group is steady. In some cases, burstiness of traffic could also occur due to time-critical application behavior - for example, imagine an update in the value of a stock quote triggering off activity in several system components, which then multicast information to a replicated central data-



(a) Cornell 64-node Cluster

(b) Utah Emulab Testbed

Figure 1: Datacenter Loss is bursty and uncorrelated across nodes: receiver r_1 (top) joins groups A and B and exhibits bursty loss, whereas receiver r_2 (bottom) joins only group A and experiences zero loss.

store. If we assume that each time-critical component processes the update within a few hundred microseconds, and that inter-node socket-to-socket latency is around fifty microseconds (an actual number from our experimental cluster), the central datastore could easily see a sub-millisecond burst of traffic. In this case, the componentized structure of the application resulted in bursty traffic; in other scenarios, the application domain could be intrinsically prone to bursty input. For example, a financial calculator tracking a set of hundred equities with correlated movements might expect to receive a burst of a hundred packets in multiple groups almost instantaneously.

3 The Design of a Time-Critical Multicast Primitive

In recent years, multicast research has focused almost exclusively on application-level routing mechanisms, or overlay networks ([13] is one example), designed to operate in the wide-area without any existing router support. The need for overlay multicast stems from the lack of IP Multicast coverage in the modern internet, which in turn reflects concerns of administration complexity, scalability, and the risk of multicast 'storms' caused by misbehaving nodes. However, the homogeneity and comparatively limited size of datacenter networks pose few scalability and administration challenges to IP Multicast, making it a viable and attractive option in such settings. In this paper, we restrict ourselves to a more traditional definition of 'reliable multicast', as a reliability layer over IP Multicast. Given that the selection of datacenter hardware is typically influenced by commercial constraints, we believe that any viable solution for this context must be able to run on any mix of existing commodity routers and OS software; hence, we focus exclusively on mechanisms that act at the application-level, ruling out schemes which require router modification, such as PGM [19].

3.1 The Timeliness of (Scalable) Reliable Multicast Protocols

Reliable multicast protocols typically consist of three logical phases: *transmission* of the packet, *discovery* of packet loss, and *recovery* from it. Recovery is a fairly fast operation; once a node knows it is missing a packet, recovering it involves retrieving the packet from some other node. However, in most existing scalable multicast protocols, the time taken to discover packet loss dominates recovery latency heavily in the kind of settings we are interested in. The key insight is that *the discovery latency of reliable multicast protocols is usually inversely dependent on data rate*: for existing protocols, the rate of outgoing data at a single sender in a single group. Existing schemes for reliability in multicast can be roughly divided into the following categories:

ACK/timeout: RMTP [21], RMTP-II [22]. In this approach, receivers send back ACKs (acknowledgements) to the sender of the multicast. This is the trivial extension of unicast reliability to multicast, and is intrinsically unscalable due to ACK implosion; for each sent message, the sender has to process an ACK from every receiver in the group [21]. One work-around is to use ACK aggregation, which allows such solutions to scale in the number of receivers but requires the construction of a tree for every sender to a group. Also, any aggregative mechanism introduces latency, leading to larger time-outs at the sender and delaying loss discovery; hence, ACK trees are unsuitable in time-critical settings.

Gossip-Based: Bimodal Multicast [10], lpbcast [17]. Receivers periodically gossip histories of received packets

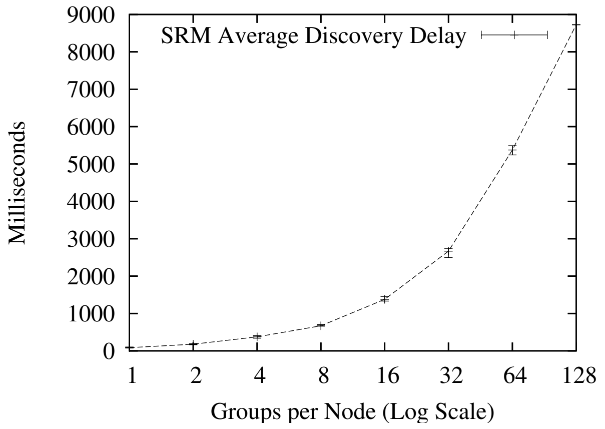


Figure 2: SRM's Discovery Latency vs. Groups per Node, on a 64-node cluster, with groups of 10 nodes each. Error bars are min and max over 10 runs.

with each other. Upon receiving a digest, a receiver compares the contents with its own packet history, sending any packets that are missing from the gossiped history and requesting transmission of any packets missing from its own history. Gossip-based schemes offer scalability in the number of receivers per group, and extreme resilience by diffusing the responsibility of ensuring reliability for each packet over the entire set of receivers. However, they are not designed for time-critical settings: discovery latency is equal to the time period between gossip exchanges (a significant number of milliseconds - 100ms in Bimodal Multicast [10]), and recovery involves a further one or two-phase interaction as the affected node obtains the packet from its gossip contact.

NAK/Sender-based Sequencing: SRM [18]. Senders number outgoing multicasts, and receivers discover packet loss when a subsequent message arrives. Loss discovery latency is thus proportional to the inter-send time at any single sender to a single group - a receiver can't discover a loss in a group until it receives the next packet from the same sender to that group - and consequently depends on the sender's data transmission rate to the group. To illustrate this point, we measured the performance of SRM as we increased the number of groups each node belonged in, keeping the throughput in the system constant by reducing the data rate within each group - as Figure 2 shows, discovery latency of lost packets degrades linearly as each node's bandwidth is increasingly fragmented and each group's rate goes down, increasing the time between two consecutive sends by a sender to the same group. Once discovery occurs in SRM, lost packet recovery is initiated by the receiver, which uses IP multicast (with a suitable TTL value); the sender (or some other receiver), responds with a retransmission, also using IP multicast.

Sender-based FEC [20, 23]: Forward Error Correction schemes involve multicasting redundant error correction information along with data packets, so that receivers can recover lost packets without contacting the sender or any other node. FEC mechanisms involve generating c repair packets for every r data packets, such that any r of the combined set of $r + c$ data and repair packets is sufficient to recover the original r data packets; we term this (r, c) parameter the *rate-of-fire*. FEC mechanisms have the benefit of *tunability*, providing a coherent relationship between overhead and timeliness - the more the number of repair packets generated, the higher the probability of recovering lost packets from the FEC data. Further, FEC based protocols are very stable under stress, since recovery does not induce large degrees of extra traffic. As in NAK protocols, the timeliness of FEC recovery depends on the data transmission rate of a single sender in a single group; the sender can send a repair packet to a group only after sending out r data packets to that group. Fast, efficient encodings such as Tornado codes [11] make sender-based FEC a very attractive option in multicast applications involving a single, dedicated sender; for example, software distribution or internet radio.

Receiver-based FEC [9]: Of the above schemes, ACK-based protocols are intrinsically unsuited for time-critical multi-sender settings, while sender-based sequencing and FEC limit discovery latency to inter-send time at a single sender within a single group. Ideally, we would like discovery latency to be independent of inter-send time, and combine the scalability of a gossip-based scheme with the tunability of FEC. Receiver-based FEC, first introduced in the Slingshot protocol [9], provides such a combination: receivers generate FEC packets from incoming data and exchange these with other randomly chosen receivers. Since FEC packets are generated from *incoming* data at a receiver, the timeliness of loss recovery depends on the rate of data multicast in *the entire group*, rather than the rate at any given sender, allowing scalability in the number of senders to the group.

Slingshot is aimed at single-group settings, recovering from packet loss in time proportional to that group's data rate. Our goal with Ricochet is to achieve recovery latency dependent on the rate of data incoming at a node *across all groups*. Essentially, we want recovery of packets to occur as quickly in a thousand 10 Kbps groups as in a single 10 Mbps group, allowing applications to divide node bandwidth among thousands of multicast groups while maintaining time-critical packet recovery. To achieve this, we introduce Lateral Error Correction, a new form of receiver-generated FEC that probabilistically combines receiver-generated repair traffic across multiple groups to drive down packet recovery latencies.

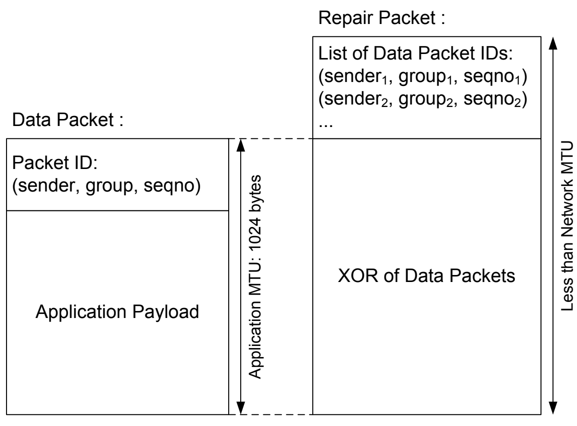


Figure 3: Ricochet Packet Structure

4 Lateral Error Correction and the Ricochet protocol

In Ricochet, each node belongs to a number of groups, and receives data multicast within any of them. The basic operation of the protocol involves generating XORs from incoming data and exchanging them with other randomly selected nodes. Ricochet operates using two different packet types: *data packets* - the actual data multicast within a group - and *repair packets*, which contain recovery information for multiple data packets. Figure 3 shows the structure of these two packet types. Each data packet header contains a packet identifier - a *(sender, group, sequence number)* tuple that identifies it uniquely. A repair packet contains an XOR of multiple data packets, along with a list of their identifiers - we say that the repair packet is *composed* from these data packets, and that the data packets are *included* in the repair packet. An XOR repair composed from r data packets allows recovery of one of them, if all the other $r-1$ data packets are available; the missing data packet is obtained by simply computing the XOR of the repair's payload with the other data packets.

At the core of Ricochet is the LEC engine running at each node that decides on the composition and destinations of repair packets, creating them from incoming data across multiple groups. The operating principle behind LEC is the notion that repair packets sent by a node to another node can be composed from data in any of the multicast groups that are common to them. This allows recovery of lost packets at the receiver of the repair packet to occur within time that's inversely proportional to the aggregate rate of data in all these groups. Figure 4 illustrates this idea: n_1 has groups A and B in common with n_2, and hence it can generate and dispatch repair packets that contain data from both these groups. n_1 needs to wait only until it receives 5 data packets in either A *or* B before it sends a repair packet, allowing faster recovery of lost packets at n_2.

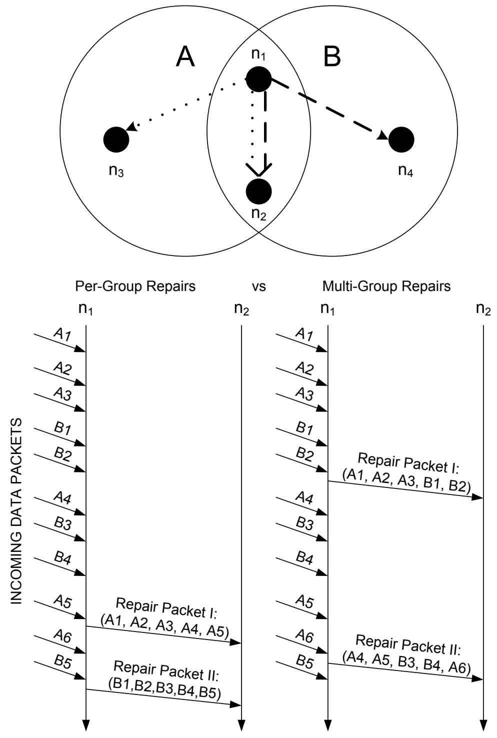


Figure 4: LEC in 2 Groups: Receiver n_1 can send repairs to n_2 that combine data from both groups A and B.

While combining data from different groups in outgoing repair packets drives down recovery time, it tampers with the coherent tunability that single group receiver-based FEC provides. The *rate-of-fire* parameter in receiver-based FEC provides a clear, coherent relationship between overhead and recovery percentage; for every r data packets, c repair packets are generated in the system, resulting in some computable probability of recovering from packet loss. The challenge for LEC is to combine repair traffic for multiple groups while retaining per-group overhead and recovery percentages, so that each individual group can maintain its own rate-of-fire. To do so, we abstract out the essential properties of receiver-based FEC that we wish to maintain:

1. Coherent, Tunable Per-Group Overhead: For every data packet that a node receives in a group with rate-of-fire (r, c), it sends out *an average of* c *repair packets* including that data packet to other nodes in the group.
2. Randomness: Destination nodes for repair packets are picked *randomly*, with no node receiving more or less repairs than any other node, on average.

LEC supports overlapping groups with the same r component and different c values in their rate-of-fire parameter. In LEC, the rate-of-fire parameter is translated into the following guarantee: For every data packet d that a node receives in a group with rate-of-fire (r, c), it selects an average of c nodes from the group randomly and sends each

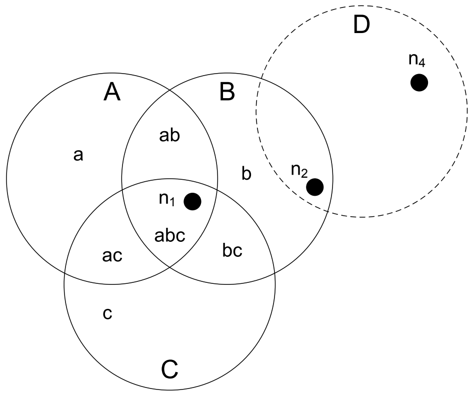


Figure 5: n_1 belongs to groups A, B, C: it divides them into disjoint regions abc, ab, ac, bc, a, b, c

of these nodes exactly one repair packet that includes d. In other words, the node sends an average of c repair packets containing d to the group. In the following section, we describe the algorithm that LEC uses to compose and dispatch repair packets while maintaining this guarantee.

4.1 Algorithm Overview

Ricochet is a symmetric protocol - exactly the same LEC algorithm and supporting code runs at every node - and hence, we can describe its operation from the vantage point of a single node, n_1.

4.1.1 Regions

The LEC engine running at n_1 divides n_1's neighborhood - the set of nodes it shares one or more multicast groups with - into *regions*, and uses this information to construct and disseminate repair packets. Regions are simply the disjoint intersections of all the groups that n_1 belongs to. Figure 5 shows the regions in a hypothetical system, where n_1 is in three groups, A, B and C. We denote groups by upper-case letters and regions by the concatenation of the group names in lowercase; i.e, abc is a region formed by the intersection of A, B and C. In our example, the neighborhood set of n_1 is carved into seven regions: abc, ac, ab, bc, a, b and c, essentially the power set of the set of groups involved. Readers may be alarmed that this transformation results in an exponential number of regions, but this is not the case; we are only concerned with non-empty intersections, the cardinality of which is bounded by the number of nodes in the system, as each node belongs to exactly one intersection (see Section 4.1.4). Note that n_1 does not belong to group D and is oblivious to it; it observes n_2 as belonging to region b, rather than bd, and is not aware of n_4's existence.

4.1.2 Selecting targets from regions, not groups

Instead of selecting targets for repairs randomly from the entire group, LEC selects targets randomly from *each region*. The number of targets selected from a region is set such that:

1. It is proportional to the size of the region
2. The total number of targets selected, across regions, is equal to the c value of the group

Hence, for a given group A with rate-of-fire (r, c_A), the number of targets selected by LEC in a particular region, say abc, is equal to $c_A * \frac{|abc|}{|A|}$, where $|x|$ is the number of nodes in the region or group x. We denote the number of targets selected by LEC in region abc for packets in group A as c_A^{abc}. Figure 6 shows n_1 selecting targets for repairs from the regions of A.

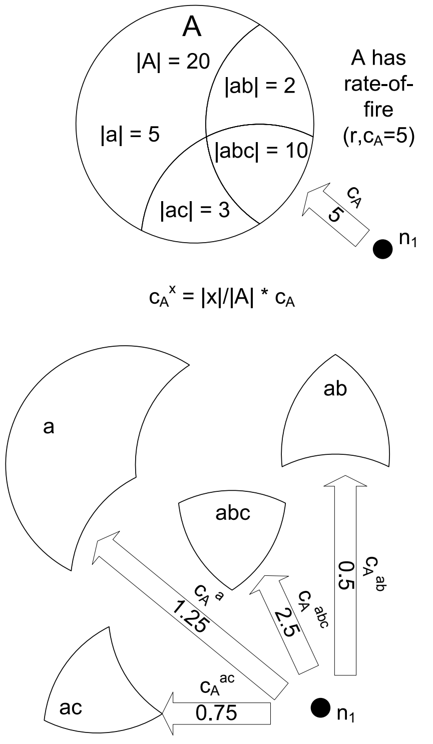


Figure 6: n_1 selects proportionally sized chunks of c_A from the regions of A

Note that LEC may pick a different number of targets from a region for packets in a different group; for example, c_A^{abc} differs from c_B^{abc}. Selecting targets in this manner also preserves randomness of selection; if we rephrase the task of target selection as a sampling problem, where a random sample of size c has to be selected from the group, selecting targets from regions corresponds to *stratified sampling* [14], a technique from statistical theory.

4.1.3 Why select targets from regions?

Selecting targets from regions instead of groups allows LEC to construct repair packets from multiple groups; since we know that all nodes in region ab are interested in data from groups A and B, we can create composite

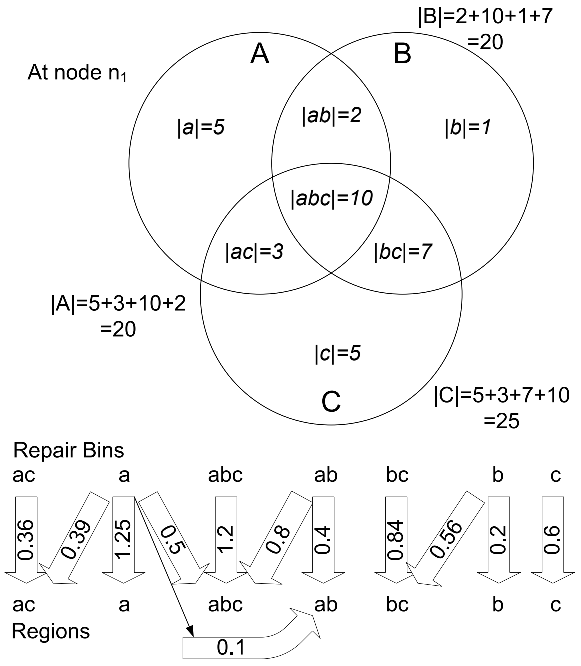


Figure 7: Mappings between repair bins and regions: the repair bin for ab selects 0.4 targets from region ab and 0.8 from abc for every repair packet. Here, $c_A = 5$, $c_B = 4$, and $c_C = 3$.

repair packets from incoming data packets in both groups and send them to nodes in that region.

Single-group receiver-based FEC [9] is implemented using a simple construct called a *repair bin*, which collects incoming data within the group. When a repair bin reaches a threshold size of r, a repair packet is generated from its contents and sent to c randomly selected nodes in the group, after which the bin is cleared. Extending the repair bin construct to regions seems simple; a bin can be maintained for each region, collecting data packets received in any of the groups composing that region. When the bin fills up to size r, it can generate a repair packet containing data from all these groups, and send it to targets selected from within the region.

Using per-region repair bins raises an interesting question: if we construct a composite repair packet from data in groups A, B, and C, how many targets should we select from region abc for this repair packet - c_A^{abc}, c_B^{abc}, or c_C^{abc}? One possible solution is to pick the maximum of these values. If $c_A^{abc} \geq c_B^{abc} \geq c_C^{abc}$, then we would select c_A^{abc}. However, a data packet in group B, when added to the repair bin for the region abc would be sent to an average of c_A^{abc} targets in the region; resulting in more repair packets containing that data packet sent to the region than required (c_B^{abc}), which results in more repair packets sent to the entire group. Hence, more overhead is expended per data packet in group B than required by its (r, c_B) value; a similar argument holds for data packets in group C as well.

Algorithm 1 Algorithm for Setting Up Repair Bins

```
1: Code at node $n_i$:
2: upon Change in Group Membership do
3:    while L not empty        {L is the list of regions}
      do
4:       Select and remove the region $R_i = abc...z$ from
         $L$ with highest number of groups involved (break
         ties in any order)
5:       Set $R_t = R_i$
6:       while $R_t \neq \epsilon$ do
7:          set $c_{min}$ to $min(c_A^{R_t}, c_B^{R_t}...)$, where {A,B,...}
            is the set of groups forming $R_t$
8:          Set number of targets selected by $R_i$'s repair
            bin from region $R_t$ to $c_{min}$
9:          Remove $G$ from $R_t$, for all groups $G$ where
            $c_G^{R_t} = c_{min}$
10:         For each remaining group $G'$ in $R_t$, set $c_{G'}^{R_t} =$
            $c_{G'}^{R_t} - c_{min}$
```

Instead, we choose the *minimum* of values; this, as expected, results in a lower level of overhead for groups A and B than required, resulting in a lower fraction of packets recovered from LEC. To rectify this we send the additional compensating repair packets to the region abc from the repair bins for regions a and b. The repair bin for region a would select $c_A^{abc} - c_C^{abc}$ destinations, on an average, for every repair packet it generates; this is in addition to the c_A^a destinations it selects from region a.

A more sophisticated version of the above strategy involves iteratively obtaining the required repair packets from regions involving the remaining groups; for instance, we would have the repair bin for ab select the minimum of c_A^{abc} and c_B^{abc} - which happens to be c_B^{abc} - from abc, and then have the repair bin for a select the remainder value, $c_A^{abc} - c_B^{abc}$, from abc. Algorithm 1 illustrates the final approach adopted by LEC, and Figure 7 shows the output of this algorithm for an example scenario. A repair bin selects a non-integral number of nodes from an intersection by alternating between its floor and ceiling probabilistically, in order to maintain the average at that number.

4.1.4 Complexity

The algorithm described above is run every time nodes join or leave any of the multicast groups that n_1 is part of. The algorithm has complexity $O(I \cdot d)$, where I is the number of populated regions (i.e, with one or more nodes in them), and d is the maximum number of groups that form a region. Note that I at n_1 is bounded from above by the cardinality of the set of nodes that share a multicast group with n_1, since regions are disjoint and each node exists in exactly one of them. d is bounded by the number of groups that n_1 belongs to.

4.2 Implementation Details

Our implementation of Ricochet is in Java. Below, we discuss the details of the implementation, along with the performance optimizations involved - some obvious and others subtle.

4.2.1 Repair Bins

A Ricochet repair bin is a lightweight structure holding an XOR and a list of data packets, and supporting an add operation that takes in a data packet and includes it in the internal state. The repair bin is associated with a particular region, receiving all data packets incoming in any of the groups forming that region. It has a list of regions from which it selects targets for repair packets; each of these regions is associated with a value, which is the average number of targets which must be selected from that region for an outgoing repair packet. In most cases, as shown in Figure 7, the value associated with a region is not an integer; as mentioned before, the repair bin alternates between the floor and the ceiling of the value to maintain the average at the value itself. For example, in Figure 7, the repair bin for abc has to select 1.2 targets from abc, on average; hence, it generates a random number between 0 and 1 for each outgoing repair packet, selecting 1 node if the random number is more than 0.2, and 2 nodes otherwise.

4.2.2 Staggering for Bursty Loss

A crucial algorithmic optimization in Ricochet is *staggering* - also known as interleaving [23] - which provides resilience to bursty loss. Given a sequence of data packets to encode, a stagger of 2 would entail constructing one repair packet from the 1st, 3rd, 5th... packets, and another repair packet from the 2nd, 4th, 6th... packets. The stagger value defines the number of repairs simultaneously being constructed, as well as the distance in the sequence between two data packets included in the same repair packet. Consequently, a stagger of i allows us to tolerate a loss burst of size i while resulting in a proportional slowdown in recovery latency, since we now have to wait for $O(i*r)$ data packets before despatching repair packets.

In conventional sender-based FEC, staggering is not a very attractive option, providing tolerance to very small bursts at the cost of multiplying the already prohibitive loss discovery latency. However, LEC recovers packets so quickly that we can tolerate a slowdown of a factor of ten without leaving the tens of milliseconds range; additionally, a small stagger at the sender allows us to tolerate very large bursts of lost packets at the receiver, especially since the burst is dissipated among multiple groups and senders. Ricochet implements a stagger of i by the simple expedient of duplicating each logical repair bin into i instances; when a data packet is added to the logical repair bin, it is actually added to a particular instance of the repair bin, chosen in round-robin fashion. Instances of a duplicated repair bin behave exactly as single repair bins do, generating repair packets and sending them to regions when they get filled up.

4.2.3 Multi-Group Views

Each Ricochet node has a *multi-group view*, which contains membership information about other nodes in the system that share one or more multicast groups with it. In traditional group communication literature, a *view* is simply a list of members in a single group [24]; in contrast, a Ricochet node's multi-group view divides the groups that it belongs to into a number of regions, and contains a list of members lying in each region. Ricochet uses the multi-group view at a node to determine the sizes of regions and groups, to set up repair bins using the LEC algorithm. Also, the per-region lists in the multi-view are used to select destinations for repair packets. The multi-group view at n_1 - and consequently the group and intersection sizes - does not include n_1 itself.

4.2.4 Membership and Failure Detection

Ricochet can plug into any existing membership and failure detection infrastructure, as long as it is provided with reasonably up-to-date views of per-group membership by some external service. In our implementation, we use simple versions of Group Membership (GMS) and Failure Detection (FD) services, which execute on high-end server machines. If the GMS receives a notification from the FD that a node has failed, or it receives a join/leave to a group from a node, it sends an update to all nodes in the affected group(s). The GMS is not aware of regions; it maintains conventional per-group lists of nodes, and sends per-group updates when membership changes. For example, if node n_{55} joins group A, the update sent by the GMS to every node in A would be a 3-tuple: *(Join, A, n_{55})*. Individual nodes process these updates to construct multi-group views relative to their own membership.

Since the GMS does not maintain region data, it has to scale only in the number of groups in the system; this can be easily done by partitioning the service on group id and running each partition on a different server. For instance, one machine is responsible for groups A and B, another for C and D, and so on. Similarly, the FD can be partitioned on a topological criterion; one machine on each rack is responsible for monitoring other nodes on the rack by pinging them periodically. For fault-tolerance, each partition of the GMS can be replicated on multiple machines using a strongly consistent protocol like Paxos. The FD can have a hierarchical structure to recover from failures; a smaller set of machines ping the per-rack failure detectors, and each other in a chain. We believe that such a semi-centralized solution is appropriate and sufficient in a datacenter setting, where connectivity and membership are typically stable. Crucially, the protocol itself does not need consistent membership, and degrades gracefully with the degree of inconsistency in the views; if a failed node is included in a view, performance will dip fractionally in all the groups it belongs to as the repairs sent to it by other nodes are wasted.

4.2.5 Performance

Since Ricochet creates LEC information from each incoming data packet, the critical communication path that a data packet follows within the protocol is vital in determining eventual recovery times and the maximum sustainable throughput. XORs are computed in each repair bin incrementally, as packets are added to the bin. A crucial optimization used is pre-computation of the number of destinations that the repair bin sends out a repair to, across all the regions that it sends repairs to: Instead of constructing a repair and deciding on the number of destinations once the bin fills up, the repair bin precomputes this number and constructs the repair only if the number is greater than 0. When the bin overflows and clears itself, the expected number of destinations for the next repair packet is generated. This restricts the average number of two-input XORs per data packet to c (from the rate-of-fire) in the worst case - which occurs when no single repair bin selects more than 1 destination, and hence each outgoing repair packet is a unique XOR.

4.2.6 Buffering and Loss Control

LEC - like any other form of FEC - works best when losses are not in concentrated bursts. Ricochet maintains an application-level buffer with the aim of minimizing in-kernel losses, serviced by a separate thread that continuously drains packets from the kernel. If memory at end-hosts is constrained and the application-level buffer is bounded, we use customized packet-drop policies to handle overflows: a randomly selected packet from the buffer is dropped and the new packet is accommodated instead. In practice, this results in a sequence of almost random losses from the buffer, which are easy to recover using FEC traffic. Whether the application-level buffer is bounded or not, it ensures that packet losses in the kernel are reduced to short bursts that occur only during periods of overload or CPU contention. We evaluate Ricochet against loss bursts of up to 100 packets, though in practice we expect the kind of loss pattern shown in 1, where few bursts are greater than 20-30 packets, even with highly concentrated traffic spikes.

4.2.7 NAK Layer for 100% Recovery

Ricochet recovers a high percentage of lost packets via the proactive LEC traffic; for certain applications, this probabilistic guarantee of packet recovery is sufficient and even desirable in cases where data ‘expires’ and there is no utility in recovering it after a certain number of milliseconds. However, the majority of applications require 100% recovery of lost data, and Ricochet uses a reactive NAK layer to provide this guarantee. If a receiver does not recover a packet through LEC traffic within a timeout period after discovery of loss, it sends an explicit NAK to the sender and requests a retransmission. While this NAK layer does result in extra reactive repair traffic, two factors separate it from traditional NAK mechanisms: firstly, recovery can potentially occur very quickly - within a few hundred milliseconds - since for almost all lost packets discovery of loss takes place within milliseconds through LEC traffic. Secondly, the NAK layer is meant solely as a backup mechanism for LEC and responsible for recovering a very small percentage of total loss, and hence the extra overhead is minimal.

4.2.8 Optimizations

Ricochet maintains a buffer of unusable repair packets that enable it to utilize incoming repair packets better. If one repair packet is missing exactly one more data packet than another repair packet, and both are missing at least one data packet, Ricochet obtains the extra data packet by XORing the two repair packets. Also, it maintains a list of unusable repair packets which is checked intermittently to see if recent data packet recoveries and receives have made any old repair packets usable.

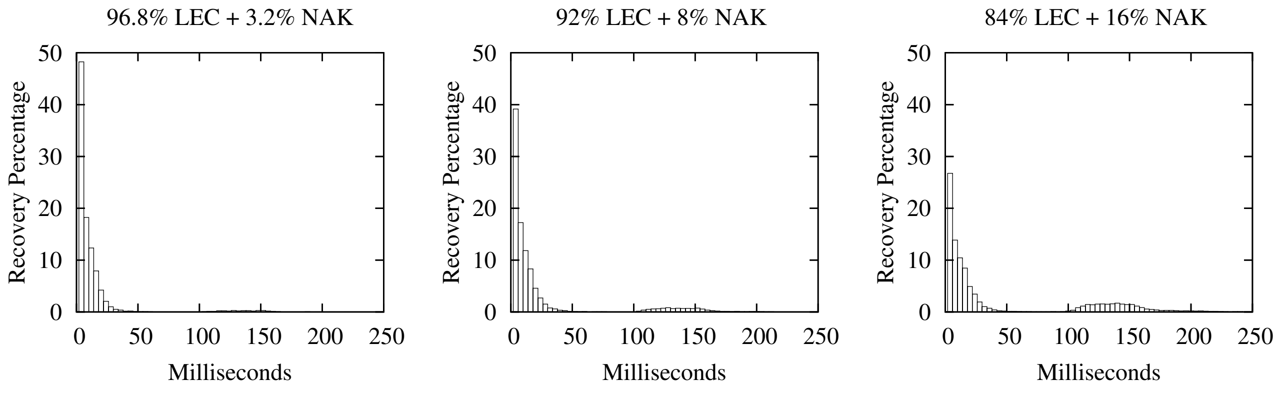
4.2.9 Message Ordering

As presented, Ricochet provides multicast reliability but does not deliver messages in the same order at all receivers. We are primarily concerned with building an extremely rapid multicast primitive that can be used by applications that require unordered reliable delivery as well as layered under ordering protocols with stronger delivery properties. For instance, Ricochet can be used as a reliable transport by any of the existing mechanisms for total ordering [16] — in separate work [8], we describe one such technique that predicts out-of-order delivery in datacenters to optimize ordering delays.

5 Evaluation

We evaluated our Java implementation of Ricochet on a 64-node cluster, comprising of four racks of 16 nodes each, interconnected via two levels of switches. Each node has a single 1.3 GHz CPU with 512 Mb RAM, runs Linux 2.6.12 and has two 100 Mbps network interfaces, one for control and the other for experimental traffic. Typical socket-to-socket latency within the cluster is around 50 microseconds. In the following experiments, for a given loss rate L, three different loss models are used:

· **uniform** - also known as the Bernoulli model [25] - refers to dropping packets with uniform probability equal to the loss rate L.



(a) 10% Loss Rate **(b) 15% Loss Rate** **(c) 20% Loss Rate**

Figure 8: Distribution of Recoveries: LEC + NAK for varying degrees of loss

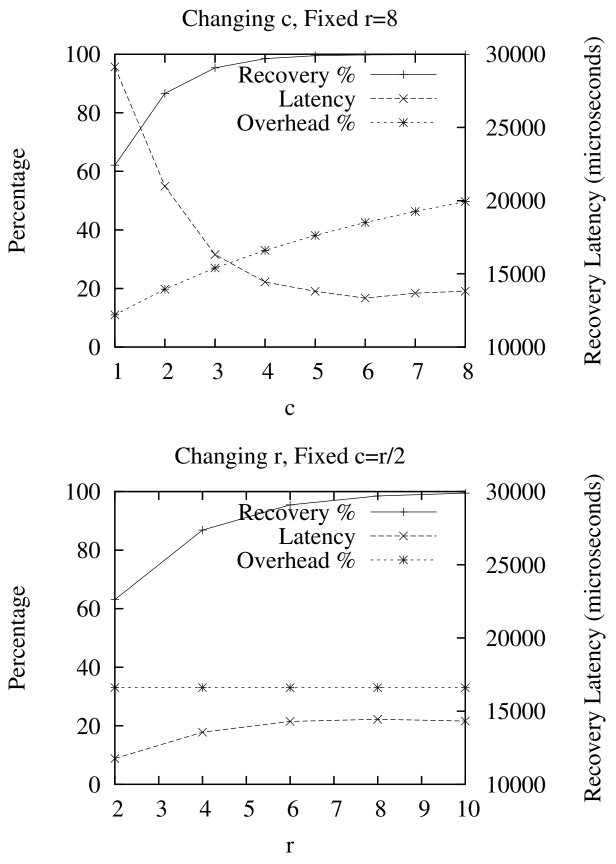


Figure 9: Tuning LEC : tradeoff points available between recovery %, overhead % (left y-axis) and avg recovery latency (right y-axis) by changing the rate-of-fire (r, c).

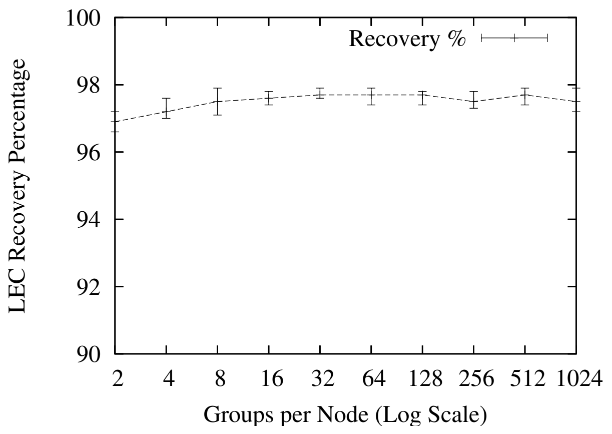
· **bursty** involves dropping packets in equal bursts of length b. The probability of starting a loss burst is set so that each burst is of exactly b packets and the loss rate is maintained at L. This is not a realistic model but allows us to precisely measure performance relative to specific burst lengths.
· **markov** drops packets using a simple 2-state markov chain, where each node alternates between a lossy and a lossless state, and the probabilities are set so that the average length of a loss burst is m and the loss rate is L, as described in [25].

In experiments with multiple groups, nodes are assigned to groups at random, and the following formula is used to relate the variables in the grouping pattern: $n * d = g * s$, where n is the number of nodes in the system (64 in most of the experiments), d is the degree of membership, i.e. the number of groups each node joins, g is the total number of groups in the system, and s is the average size of each group. For example, in a 16-node setting where each node joins 512 groups and each group is of size 8, g is set to $\frac{16*512}{8} \approx 1024$. Each node is then assigned to 512 randomly picked groups out of 1024. Hence, the grouping patterns for each experiment is completely represented by a (n, d, s) tuple.

For every run, we set the sending rate at a node such that the total system rate of incoming messages is 64000 packets per second, or 1000 packets per node per second. Data packets are 1K bytes in size. Each point in the following graphs - other than Figure 8, which shows distributions for single runs - is an average of 5 runs. A run lasts 30 seconds and produces $\approx$ 2 million receive events in the system.

5.1 Distribution of Recoveries in Ricochet

First, we provide a snapshot of what typical packet recovery timelines look like in Ricochet. Earlier, we made



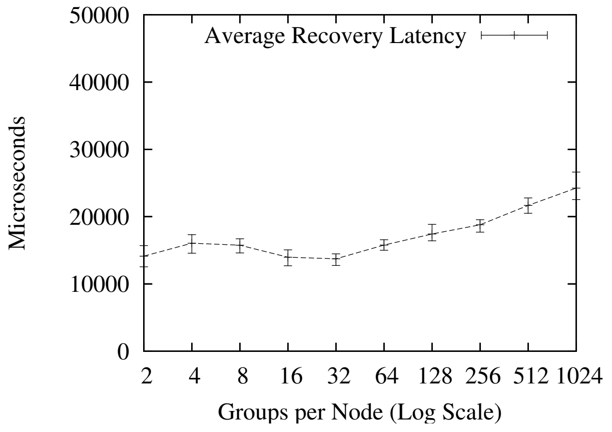
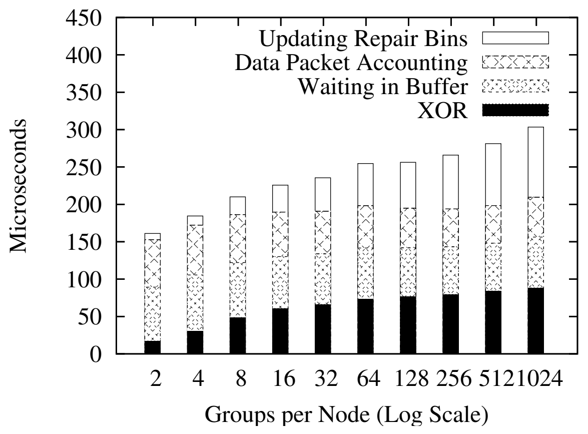


Figure 10: Scalability in Groups



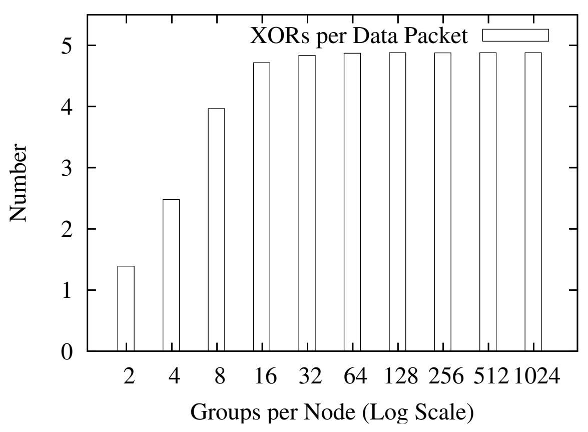
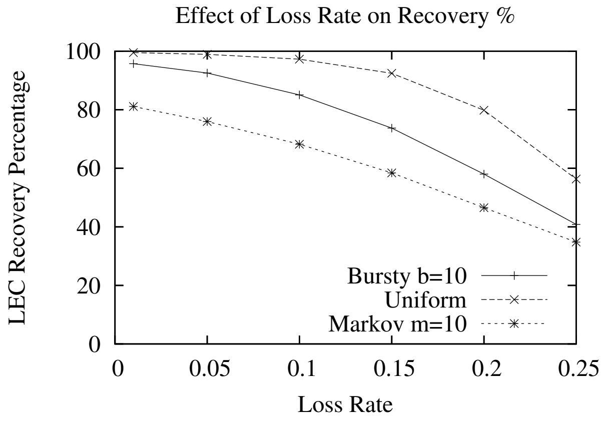


Figure 11: CPU time and XORs per data packet

the assertion that Ricochet discovers the loss of almost all packets very quickly through LEC traffic, recovers a majority of these instantly and recovers the remainder using an optional NAK layer. In Figure 8, we show the histogram of packet recovery latencies for a 16-node run with degree of membership $d = 128$ and group size $s = 10$. We use a simplistic NAK layer that starts unicasting NAKs to the original sender of the multicast 100 milliseconds after discovery of loss, and retries at 50 millisecond intervals. Figure 8 shows three scenarios: under uniform loss rates of 10%, 15%, and 20%, different fractions of packet loss are recovered through LEC and the remainder via reactive NAKs. These graphs illustrate the meaning of the LEC recovery percentage: if this number is high, more packets are recovered very quickly without extra traffic in the initial segment of the graphs, and less reactive overhead is induced by the NAK layer. Importantly, even with a recovery percentage as low as 84% in Figure 8(c), we are able to recover all packets within 250 milliseconds with a crude NAK layer due to early LEC-based discovery of loss. For the remaining experiments, we will switch the NAK layer off and focus solely on LEC performance; also, since we found this distribution of recovery latencies to be fairly representative, we present only the percentage of lost packets recovered using LEC and the average latency of these recoveries. Experiment Setup: $(n = 16, d = 128, s = 10)$, Loss Model: Uniform, [10%, 15%, 20%].

5.2 Tunability of LEC in multiple groups

The Slingshot protocol [9] illustrated the tunability of receiver-generated FEC for a single group; we include a similar graph for Ricochet in Figure 9, showing that the rate-of-fire parameter (r, c) provides a knob to tune LEC's recovery characteristics. In Figure 9.a, we can see that increasing the c value for constant $r = 8$ increases the recovery percentage and lowers recovery latency by expending more overhead - measured as the percentage of repair packets to all packets. In Figure 9.b, we see the impact of increasing r, keeping the ratio of c to r - and consequently, the overhead - constant. For the rest of the experiments, we set the rate-of-fire at $(r = 8, c = 5)$. Experiment Setup: $(n = 64, d = 128, s = 10)$, Loss Model: Uniform, 1%.



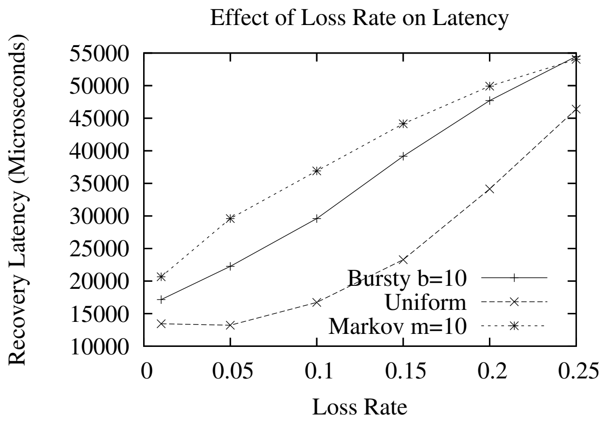
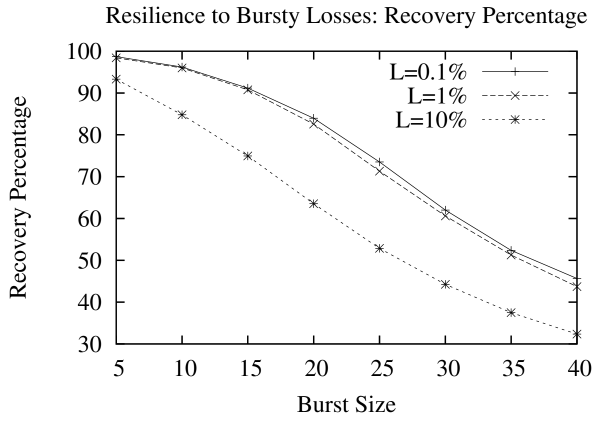


Figure 12: Impact of Loss Rate on LEC



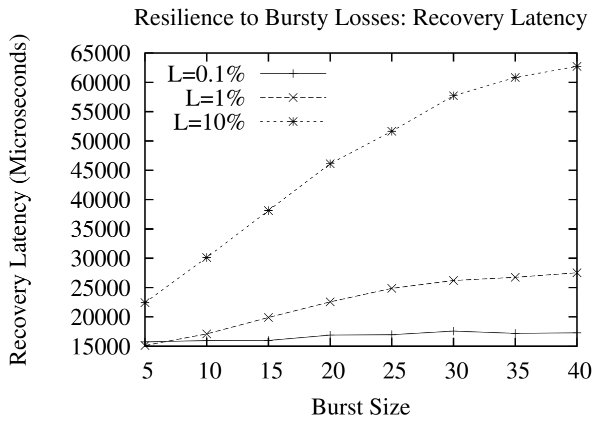


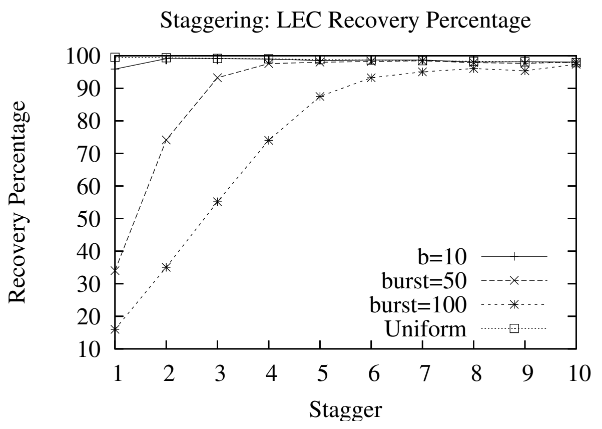
Figure 13: Resilience to Burstiness

5.3 Scalability

Next, we examine the scalability of Ricochet to large numbers of groups. Figure 10 shows that increasing the degree of membership for each node from 2 to 1024 has almost no effect on the percentage of packets recovered via LEC, and causes a slow increase in average recovery latency. The x-axis in these graphs is log-scale, and hence a straight line increase is actually logarithmic with respect to the number of groups and represents excellent scalability. The increase in recovery latency towards the right side of the graph is due to Ricochet having to deal internally with the representation of large numbers of groups; we examine this phenomenon later in this section.

For a comparison point, we refer readers back to SRM's discovery latency in Figure 2: in 128 groups, SRM discovery took place at 9 seconds. In our experiments, SRM recovery took place roughly 4 seconds after discovery in all cases. While fine-tuning the SRM implementation for clustered settings should eliminate that 4 second gap between discovery and recovery, at 128 groups Ricochet surpasses SRM's best possible recovery performance of 5 seconds by between 2 and 3 orders of magnitude.

Though Ricochet's recovery characteristics scale well in the number of groups, it is important that the computational overhead imposed by the protocol on nodes stays manageable, given that time-critical applications are expected to run over it. Figure 11 shows the scalability of an important metric: the time taken to process a single data packet. The straight line increase against a log x-axis shows that per-packet processing time increases logarithmically with the number of groups - doubling the number of groups results in a constant increase in processing time. The increase in processing time towards the latter half of the graph is due to the increase in the number of repair bins with the number of groups. While we considered 1024 groups adequate scalability, Ricochet can potentially scale to more groups with further optimization, such as creating bins only for occupied regions. In the current implementation, per-packet processing time goes from 160 microseconds for 2 groups to 300 microseconds for 1024, supporting throughput exceeding a thousand packets per second. Figure 11 also shows the average number of XORs per incoming data packet. As stated in section 4.2.2, the number of XORs stays under 5 - the value of c from the rate-of-fire (r, c). Experiment Setup: $(n = 64, d = *, s = 10)$, Loss Model: Uniform, 1%.



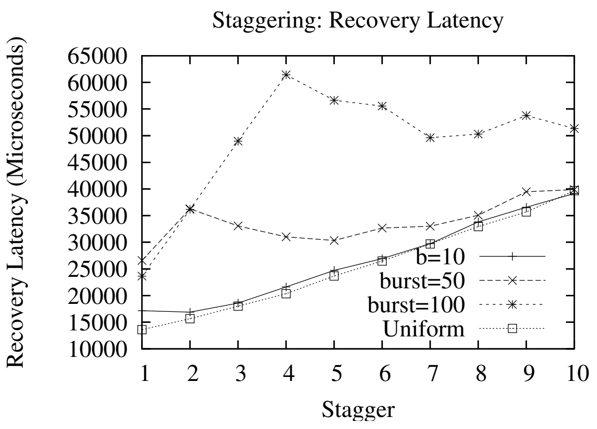


Figure 14: Staggering allows Ricochet to recover from long bursts of loss.

5.4 Loss Rate and LEC Effectiveness

Figure 12 shows the impact of the Loss Rate on LEC recovery characteristics, under the three loss models. Both LEC recovery percentages and latencies degrade gracefully: with an unrealistically high loss rate of 25%, Ricochet still recovers 40% of lost packets at an average of 60 milliseconds. For uniform and bursty loss models, recovery percentage stays above 90% with a 5% loss rate; markov does not fare as well, even at 1% loss rate, primarily because it induces bursts much longer than its average of 10 - the max burst in this setting averages at 50 packets. Experiment Setup: $(n = 64, d = 128, s = 10)$, Loss Model: *.

5.5 Resilience to Bursty Losses

As we noted before, a major criticism of FEC schemes is their fragility in the face of bursty packet loss. Figure 13 shows that Ricochet is naturally resilient to small loss bursts, without the stagger optimization - however, as the burst size increases, the percentage of packets recovered using LEC degrades substantially. Experiment Setup: $(n = 64, d = 128, s = 10)$, Loss Model: Bursty.

However, switching on the stagger optimization described in Section 4.2.2 increases Ricochet's resilience to burstiness tremendously, without impacting recovery latency much. Figure 14 shows that setting an appropriate stagger value allows Ricochet to handle large bursts of loss: for a burst size as large as 100, a stagger of 6 enables recovery of more than 90% lost packets at an average latency of around 50 milliseconds. Experiment Setup: $(n = 64, d = 128, s = 10)$, Loss Model: Bursty, 1%.

5.6 Effect of Group and System Size

What happens to LEC performance when the average group size in the cluster is large compared to the total number of nodes? Figure 15 shows that recovery percentages are almost unaffected, staying above 99% in this scenario, but recovery latency is impacted by more than a factor of 2 as we triple group size from 16 to 48 in a 64-node setting. Note that this measures the impact of the size of the group relative to the entire system; receiver-based FEC has been shown to scale well in a single isolated group to hundreds of nodes [9]. Experiment Setup: $(n = 64, d = 128, s = *)$, Loss Model: Uniform, 1%.

While we could not evaluate to system sizes beyond 64 nodes, Ricochet should be oblivious to the size of the entire system, since each node is only concerned with the groups it belongs to. We ran 4 instances of Ricochet on each node to obtain an emulated 256 node system with each instance in 128 groups, and the resulting recovery percentage of 98% - albeit with a degraded average recovery latency of nearly 200 milliseconds due to network and CPU contention - confirmed our intuition of the protocol's fundamental insensitivity to system size.

6 Future Work

One avenue of research involves embedding more complex error codes such as Tornado [11] in LEC; however, the use of XOR has significant implications for the design of the algorithm, and using a different encoding might require significant changes. LEC uses XOR for its simplicity and speed, and as our evaluation showed, we obtain properties on par with more sophisticated encodings, including tunability and burst resilience. We plan on replacing our simplistic NAK layer with a version optimized for bulk transfer, providing an efficient backup for LEC when sustained bursts occur of hundreds of packets or more. Another line of work involves making the parameters for LEC - such as rate-of-fire and stagger - adaptive, reacting to meet varying load and network characteristics. We are currently working with industry partners to layer Ricochet under data distribution, publish-subscribe and web-service interfaces, as well as building protocols with stronger ordering and atomicity properties over it.

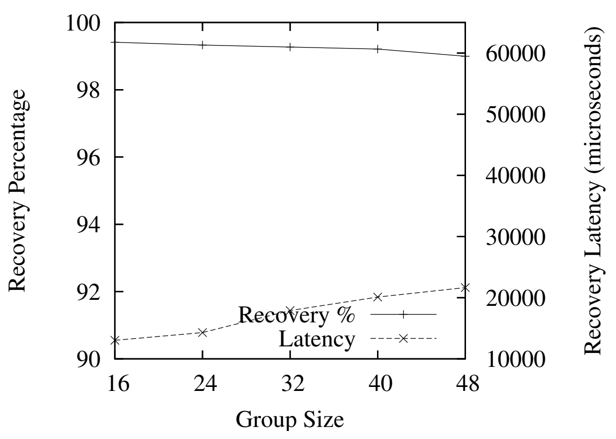


Figure 15: Effect of Group Size

7 Conclusion

We believe that the next generation of time-critical applications will execute on commodity clusters, using the techniques of massive redundancy, fault-tolerance and scalable communication currently available to distributed systems practitioners. Such applications will require a multicast primitive that delivers data at the speed of hardware multicast in failure-free operation and recovers from packet loss within milliseconds irrespective of the pattern of usage. Ricochet provides applications with massive scalability in multiple dimensions - crucially, it scales in the number of groups in the system, performing well under arbitrary grouping patterns and overlaps. A clustered communication primitive with good timing properties can ultimately be of use to applications in diverse domains not normally considered time-critical - e-tailers, online web-servers and enterprise applications, to name a few.

Acknowledgments

We received invaluable comments from Dave Andersen, Danny Dolev, Tudor Marian, Art Munson, Robbert van Renesse, Emin Gun Sirer, Niraj Tolia and Einar Vollset. We would like to thank our shepherd Mike Dahlin, as well as all the anonymous reviewers of the paper.

References

[1] Bea weblogic. `http://www.bea.com/framework.jsp?CNT=index.htm&FP=/content/products/weblogic`, 2006.

[2] Gemstone gemfire. `http://www.gemstone.com/products/gemfire/enterprise.php`, 2006.

[3] Ibm websphere. `www.ibm.com/software/webservers/appserv/was/`, 2006.

[4] Jboss. `http://labs.jboss.com/portal/`, 2006.

[5] Real-time innovations data distribution service. `http://www.rti.com/products/data_distribution/index.html`, 2006.

[6] Tangosol coherence. `http://www.tangosol.com/html/coherence-overview.shtml`, 2006.

[7] Tibco rendezvous. `http://www.tibco.com/software/messaging/rendezvous.jsp`, 2006.

[8] M. Balakrishnan, K. Birman, and A. Phanishayee. Plato: Predictive latency-aware total ordering. In *IEEE SRDS*, 2006.

[9] M. Balakrishnan, S. Pleisch, and K. Birman. Slingshot: Time-critical multicast for clustered applications. In *IEEE Network Computing and Applications*, 2005.

[10] K. P. Birman, M. Hayden, O. Ozkasap, Z. Xiao, M. Budiu, and Y. Minsky. Bimodal multicast. *ACM Trans. Comput. Syst.*, 17(2):41–88, 1999.

[11] J. W. Byers, M. Luby, M. Mitzenmacher, and A. Rege. A digital fountain approach to reliable distribution of bulk data. In *ACM SIGCOMM '98 Conference*, pages 56–67, New York, NY, USA, 1998. ACM Press.

[12] Y. Chawathe, S. McCanne, and E. A. Brewer. Rmx: Reliable multicast for heterogeneous networks. In *INFOCOM*, pages 795–804, 2000.

[13] Y. Chu, S. Rao, S. Seshan, and H. Zhang. Enabling conferencing applications on the internet using an overlay muilticast architecture. In *ACM SIGCOMM*, pages 55–67, New York, NY, USA, 2001. ACM Press.

[14] W. G. Cochran. *Sampling Techniques, 3rd Edition.* John Wiley, 1977.

[15] S. E. Deering and D. R. Cheriton. Multicast routing in datagram internetworks and extended lans. *ACM Trans. Comput. Syst.*, 8(2):85–110, 1990.

[16] X. Défago, A. Schiper, and P. Urbán. Total order broadcast and multicast algorithms: Taxonomy and survey. *ACM Computing Surveys*, 36(4):372–421, December 2004.

[17] P. T. Eugster, R. Guerraoui, S. B. Handurukande, P. Kouznetsov, and A.-M. Kermarrec. Lightweight probabilistic broadcast. *ACM Trans. Comput. Syst.*, 21(4):341–374, 2003.

[18] S. Floyd, V. Jacobson, C.-G. Liu, S. McCanne, and L. Zhang. A reliable multicast framework for light-weight sessions and application level framing. *IEEE/ACM Trans. Netw.*, 5(6):784–803, 1997.

[19] J. Gemmel, T. Montgomery, T. Speakman, N. Bhaskar, and J. Crowcroft. The pgm reliable multicast protocol. *IEEE Network*, 17(1):16–22, Jan 2003.

[20] C. Huitema. The case for packet level fec. In *PfHSN '96: Proceedings of the TC6 WG6.1/6.4 Fifth International Workshop on Protocols for High-Speed Networks V*, pages 109–120, London, UK, UK, 1997. Chapman & Hall, Ltd.

[21] J. C. Lin and S. Paul. RMTP: A reliable multicast transport protocol. In *INFOCOM*, pages 1414–1424, San Francisco, CA, Mar. 1996.

[22] T. Montgomery, B. Whetten, M. Basavaiah, S. Paul, N. Rastogi, J. Conlan, and T. Yeh. The RMTP-II protocol, Apr. 1998. IETF Internet Draft.

[23] J. Nonnenmacher, E. Biersack, and D. Towsley. Parity-based loss recovery for reliable multicast transmission. In *Proceedings of the ACM SIGCOMM '97 conference*, pages 289–300, New York, NY, USA, 1997. ACM Press.

[24] P. Verissimo and L. Rodrigues. *Distributed Systems for System Architects*. Kluwer Academic Publishers, Norwell, MA, USA, 2001.

[25] M. Yajnik, S. B. Moon, J. F. Kurose, and D. F. Towsley. Measurement and modeling of the temporal dependence in packet loss. In *INFOCOM*, pages 345–352, 1999.

PLATO: Predictive Latency-Aware Total Ordering

Mahesh Balakrishnan, Ken Birman and Amar Phanishayee

{mahesh, ken, amar}@cs.cornell.edu

Department of Computer Science

Cornell University

Keywords: Datacenters, Distributed Systems, Group Communication, Multicast, Replication, Total Ordering.

Abstract

PLATO is a predictive total ordering protocol designed for low-latency multicast in datacenters. It predicts out-of-order arrival of multicast packets by observing their inter-arrival times, and delays packets before passing them up to the application only if it believes the packets to have arrived in the wrong order. We show through experimentation on real datacenter-style networks that the inter-arrival time of consecutive packet pairs is an excellent predictor of out-of-order delivery. We evaluate an implementation of PLATO on the Emulab testbed, and show that it drives down delivery latencies by more than a factor of 2 compared to the fixed-sequencer protocol.

1 Introduction

Total ordering is a fundamental problem in distributed systems - in simple terms, it refers to the task of ensuring that a set of nodes deliver incoming multicast messages in the same order. The total ordering problem is defined within the context of the *group communication* paradigm, where processes communicate with each other using multicast groups providing different message delivery semantics.

Total ordering protocols occupy a critical slot in the communication stack of modern commercial datacenters, allowing applications to distribute and replicate data and functionality with strong consistency guarantees. Made popular by online e-commerce websites, datacenters are increasingly the computing platform of choice for a wider range of applications, ranging from computational finance to mission-critical applications. Application domains traditionally centered around expensive, specialized hardware and software - such as air-traffic control or military command-and-control - have begun to migrate towards commodity datacenters, lured as much by the cheap running costs and easy maintainability of COTS components as by the possibilities of the scale-out paradigm - massive scalability and very high availability.

A growing class of datacenter applications is time-sensitive and requires low-latency delivery of multicast messages. In some cases, timeliness is directly related to real-world metrics - for instance, the inventory service of an online bookseller has to reflect the latest item counts, to prevent losses caused by overselling. Other examples involve calculators running in financial datacenters, which require up-to-date information on stock quotes, and tracking applications on military datacenters dealing in location updates of targets. Such applications require the ability to spread data consistently and rapidly throughout a datacenter - mandating the need for a fast, low-latency total ordering protocol.

In this paper, we present PLATO, an optimistic total ordering protocol designed for time-critical datacenter applications. The key idea behind PLATO is to delay incoming data packets from the application only if there is a significant likelihood that they might be out of order - and to use a predictive scheme to determine that likelihood. PLATO allows applications to consume most incoming data packets within a few hundred microseconds of arrival, delaying packets to wait for ordering information only when it believes their arrival order to be inconsistent across the multicast group. In line with other optimistic ordering protocols [6, 7, 8], PLATO requires the application to have *rollback* capability, allowing delivered packets to be revoked when predictions are incorrect.

The predictor of out-of-order arrival used by PLATO is the inter-arrival time of consecutive packet pairs into user-space. Packet inter-arrival time is a simple yet powerful predictor of disorder in a datacenter setting - importantly, it is local information available at no extra cost or instrumentation at the receivers. To our knowledge, PLATO is the first predictive total ordering protocol - while there is at least one protocol that masks delay differences between receivers to achieve total order [8], we are not aware of any existing protocol that attempts to speculate on disorder on a per-packet basis.

The contributions of this paper are:

- We experimentally assess the causes and extent of out-of-order delivery on two datacenter-style switched networks - the Emulab testbed at Utah and a 252-node cluster at Cornell University.

- We propose the usage of inter-arrival times of consecutive packet pairs as a predictor of out-of-order delivery. We motivate this predictor by experimentally observing a high correlation between low inter-arrival times and out-of-order delivery.

- We design and implement PLATO, a predictive total-ordering protocol that uses the above predictor to decide whether arriving packets are in order or not - and waits for extra ordering information only in the latter case.

- PLATO is evaluated on the Emulab testbed, and performs significantly better than the existing fixed-sequencer protocol, slashing delivery latency by more than a factor of 2 while incurring less than 1% rollbacks.

In Section 2, we articulate the requirements of a time-critical datacenter ordering protocol. In Section 3, we assess the extent and causes of out-of-order delivery on datacenter-style networks, and show that the inter-arrival time of packets can be an excellent predictor of disorder. Section 4 provides the design and implementation details of PLATO, and Section 5 is the evaluation of the implementation.

2 The profile of a datacenter total ordering protocol

A time-critical total ordering protocol is likely to co-exist on nodes with other protocols, competing for bandwidth and CPU cycles. A typical design for a datacenter application is shown in 1.(a), where several nodes host a replicated service; they are queried by other nodes via TCP/IP or some other unicast protocol, and updated using totally ordered multicast. For example, the replicated inventory service mentioned earlier would receive updates from the service responsible for processing buy transactions and be queried by services requiring up-to-date information on the availability of items.

The existence of other competing protocols and a time-critical, possibly CPU-intensive application running on the node emphasizes the need for a light-weight, low-overhead ordering mechanism. Datacenter workloads are likely to vary due to external factors - Christmas season for online stores and high-activity periods for stock calculators come to mind - and the ordering protocol should be capable of working well at different data rates, providing timely delivery at low and intermediate throughputs while being able to sustain bursts of high-throughput

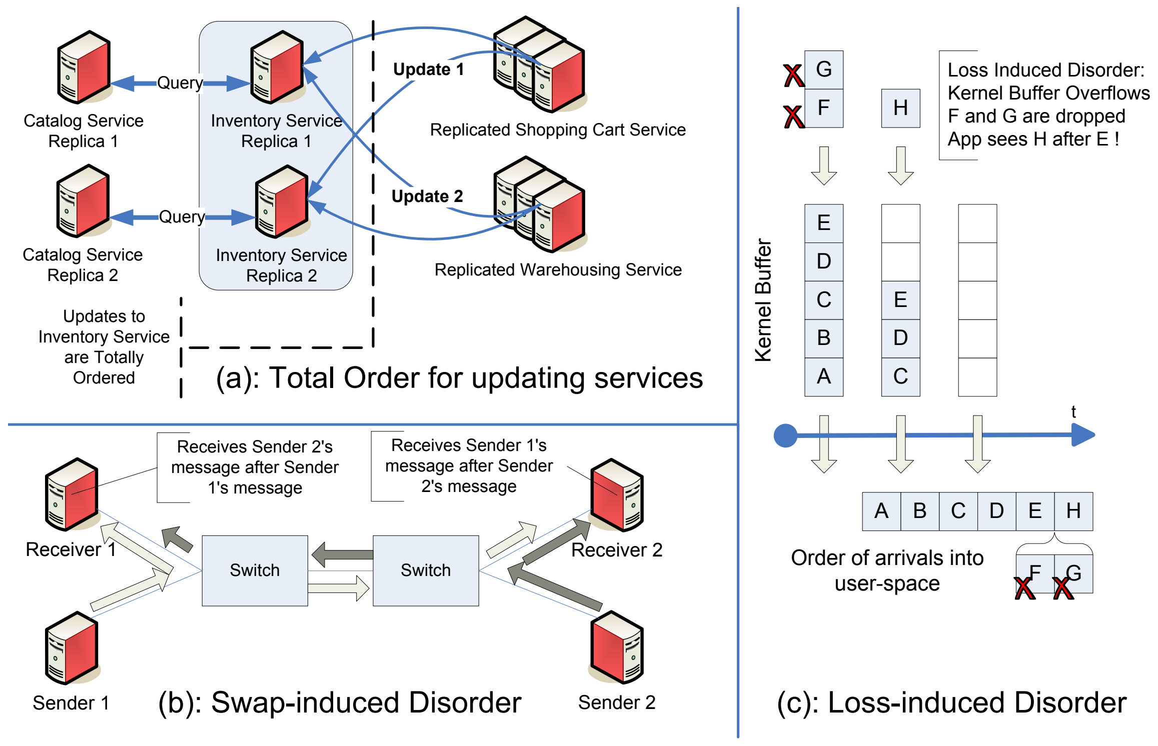

Query
Catalog Service Replica 1
Inventory Service Replica 1
Query
Catalog Service Replica 2
Inventory Service Replica 2
Update 1
Replicated Shopping Cart Service
Update 2
Replicated Warehousing Service
Updates to Inventory Service are Totally Ordered
(a): Total Order for updating services
Receives Sender 2's message after Sender 1's message
Receives Sender 1's message after Sender 2's message
Receiver 1
Switch
Switch
Receiver 2
Sender 1
(b): Swap-induced Disorder
Sender 2
Loss Induced Disorder:
Kernel Buffer Overflows
F and G are dropped
App sees H after E !
Kernel Buffer
G
F
H
E
D
C
B
A
E
D
C
t
A B C D E H
F G
Order of arrivals into user-space
(c): Loss-induced Disorder

Figure 1.

traffic. A related goal is throughput and performance stability, typically achieved by inducing exclusively proactive overheads and avoiding costly reactions to failures that further destabilize the system. Additionally, datacenters and large clusters exhibit specific failure modes and performance trade-offs, and the ideal ordering protocol for such settings should be able to exploit the natural properties of the underlying hardware - while retaining the ability to work well on many different kinds of commodity hardware.

Of this wishlist of properties, we would like to underscore the importance of performance at low and rapidly varying data rates. We have argued elsewhere that the natural use of multicast in a datacenter gives rise to large numbers of groups with low individual data rates [1]. Imagine a replicated data store where fine-grained objects are cloned and cached on different nodes with high-level objectives such as fault-tolerance and data locality; if multicast is used to update objects, each node has to belong to as many groups as the number of objects it caches or replicates, resulting in large numbers of groups that overlap in chaotic patterns. Another example involves financial calculators that use publish-subscribe libraries to subscribe to the latest prices for different equities, and hence belong to as many multicast groups as the equities they are interested in. The activity level within a single group can vary dramatically, even if the overall system throughput stays constant - the popularity of a single replicated object could experience sharp spikes, either independently or in correlation with other objects.

To summarize the properties mentioned thus far:

- The protocol should leverage the natural properties of the datacenter hardware,
- impose minimal and stable overheads,
- and crucially, work well at low per-group data rates that vary sharply over time and across groups.

3 Cluster Properties

It is a well-known fact that broadcasts on LANs arrive almost simultaneously at all receivers, and consequently the arrival of packets in different orders at different receivers is a very rare event. Multiple protocols have leveraged this property to provide optimistic delivery of broadcast messages to the application [6, 7]. In this paper, we extend this observation to IP Multicast [2] within datacenters.

Datacenters are typically heterogenous agglomerations of smaller homogenous clusters, interconnected by high-capacity switches. Intuitively, out-of-order delivery in switched networks occurs in two forms: *swaps* and *packet loss*. Swaps occur due to disparities in the distances between senders and receivers. Consider the simple case of two senders and two receivers illustrated in Figure 1.(b), where one sender is very close to one receiver and relatively far from the other one, and the other sender is placed close to the second receiver and far away from the first one. Nearly simultaneous multicasts from the two senders will arrive at different orders at the two receivers.

Packet loss in a datacenter almost never occurs within the networking fabric; more commonly, it is the end-host kernel that gets overwhelmed by the rate of incoming traffic and drops packets [1]. Figure 1.(c) illustrates how kernel buffer overflows trigger out-of-order delivery - the receiver delivers the packets immediately before and

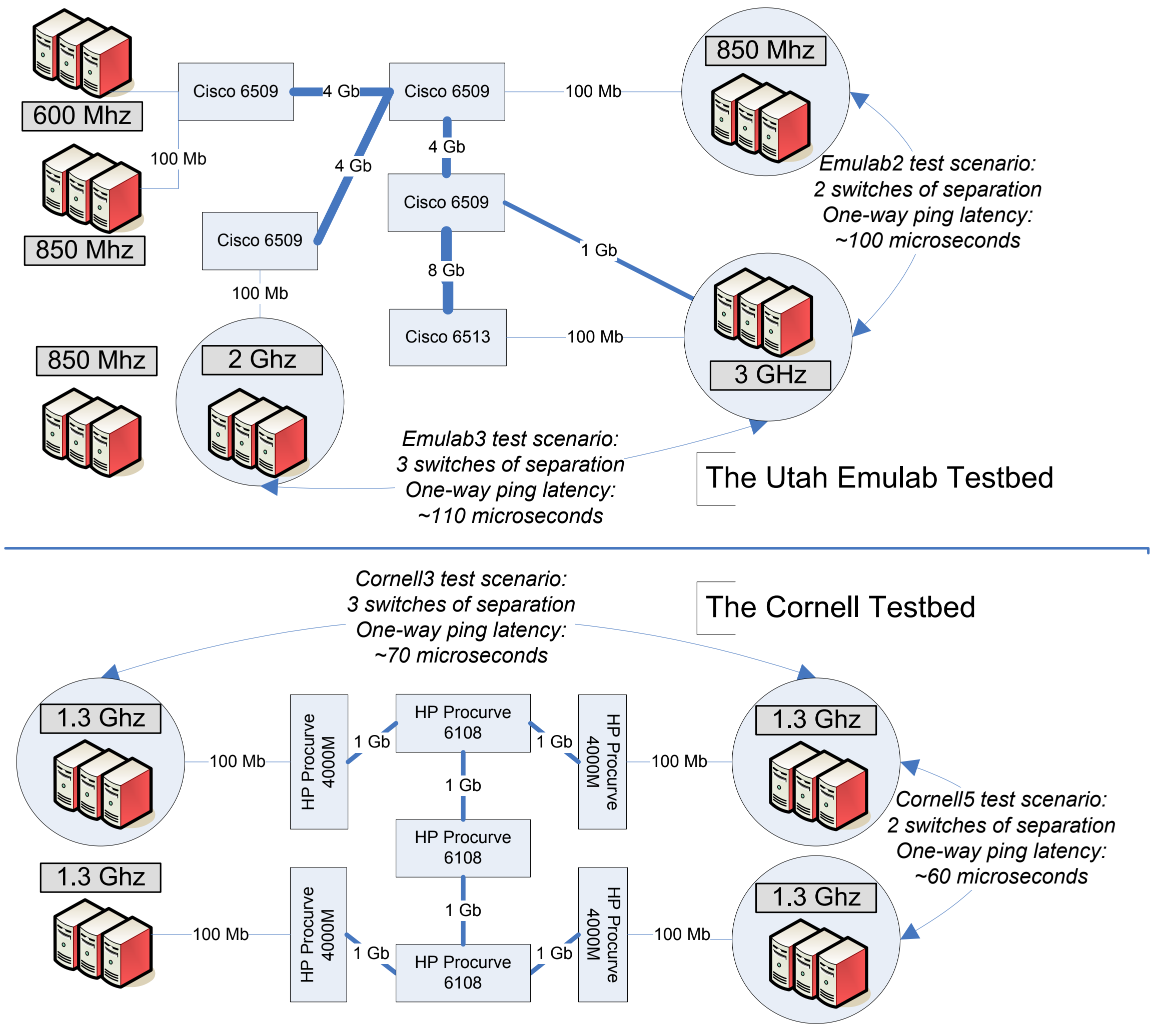


Figure 2. Clusters

after the loss burst in consecutive order.

3.1 Experiments

We ran simple experiments on two datacenter-style switched networks to evaluate the extent of out-of-order delivery of multicast messages. The first of these is the Emulab testbed at Utah [10], which is a heterogenous collection of several smaller clusters connected by Cisco 6500 series switches; Figure 2 shows the topology of the testbed (redrawn from information on www.emulab.net) - inter-node one-way ping latencies range from 100 to 300 microseconds, depending on the location of the nodes and how loaded the network is. The second network is a homogenous rack-style cluster of 252 1.3 Ghz nodes at Cornell University, comprising of 14 racks of 18 blade-servers each, interconnected via a 3-level hierarchy with HP Procurve 4000M switches at the leaves and HP Procurve 6108 switches in the interior - the network diameter is around 60-100 microseconds.

Our experiment involved placing a receiver and a sender each on two different parts of the network separated by multiple switches, and multicasting data at different rates to measure the frequency of out-of-order deliveries. We ran this experiment in four scenarios - $Cornell3$ and $Cornell5$: the Cornell cluster, with the 1.3 Ghz sender-receiver pairs separated by three switches and five switches, respectively, $Emulab3$: the Emulab testbed, where one sender-receiver pair consists of 3 Ghz nodes and the other sender-receiver pair is made up of 2 Ghz nodes three switches away, and $Emulab2$: the Emulab testbed, where one pair consists of 3 Ghz nodes and the other of 850 Mhz nodes two switches away. Figure 2 outlines the placement of these four scenarios.

Figure 3 shows the percentage of swaps and losses in these four scenarios, as we increase the multicast sending rate in the group. We measure simple swaps by comparing receiver logs after the experiment and locating consecutive packets which are delivered in inverted orders at the two receivers. As expected, the higher the rate of multicasts, the higher the probability of a swap occurring - for the Cornell cluster 5-switch scenario, the percentage of consecutive packet pairs which are swaps rises from 1% at 800 packets per second to 4% at 4000 packets per second. For the Emulab 3-switch scenario, the percentage of swaps rises from 0.7% at 800 packets a second to around 3.2% at 4000 packets per second. In these graphs, we do not show the frequency of more complex swap events, where a sequence of packets is swapped with another sequence - we observed a very small percentage $(< 0.0001\%)$ of these on the Cornell cluster, and none of them on the Emulab testbed.

Figure 3 also shows that packet loss increases with receive throughput, albeit less smoothly - the Cornell 5-switch scenario loses more packets at 2000 packets a second than at 2400 packets a second, and the Emulab 3-switch scenario exhibits more loss at 3200 packets per second than at 3600 packets per second. Our hypothesis for this uneven increase in packet loss is that the inter-arrival time of packets interacts with the OS thread scheduling mechanisms in complex ways - at intermediate rates, the receive thread is occasionally context-switched out and

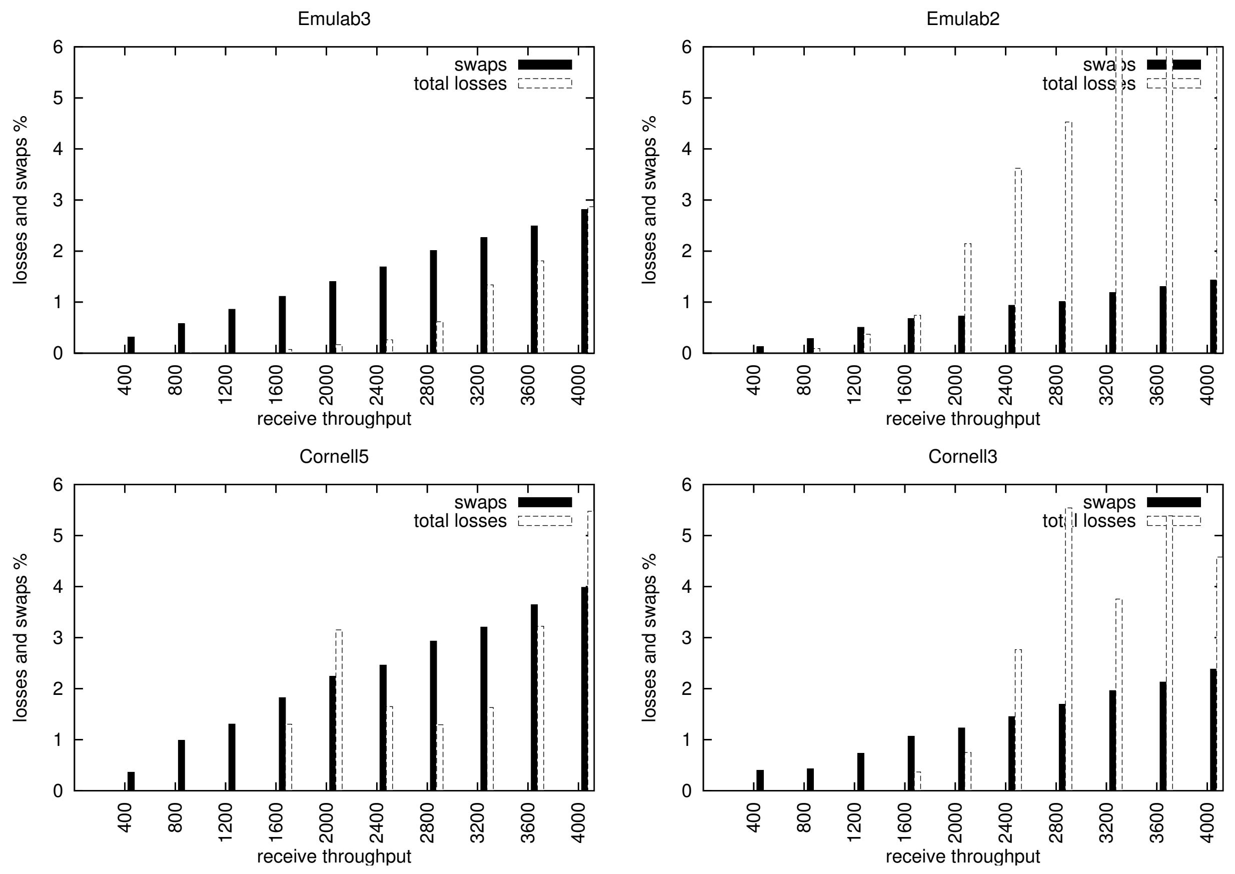


Figure 3. Disorder Characterization

loses packets while it's not running, whereas at very high rates the receive thread is continuously dequeueing packets off the socket and hence is rarely context-switched out.

With this experiment, we established that out-of-order delivery does occur in switched datacenter-style networks. Next, we explore the feasibility of using the inter-arrival time of consecutive packets into user-space as a predictor of both swaps and packet loss. Since swaps occur when multicasts are nearly simultaneous, it is natural to hypothesize that a swap would involve two packet arrivals that are very close to each other - in this case, we expect the arrival times of packets into user-space to reflect the actual timing of the multicasts. Since packet loss occurs when kernel buffers overflow, we'd expect to observe a sequence of very low user-space inter-arrival times immediately prior to the loss burst, as the receive thread rapidly empties packets from the full kernel buffer. Recall that we explained these scenarios in Figure 1.

To validate our hypotheses, we examined distributions of inter-arrival times for consecutive packet pairs. We are interesting in two metrics of the distributions:

- The time representing the 95th percentile of inter-arrival times of swapped packet pairs, and
- the percentage of all consecutive packet pairs - swapped or not - whose inter-arrival times fall within this limit.

Figure 4 shows this data in six settings: the top three graphs are for different throughputs, in the *Cornell3* scenario, and the bottom three are for a single throughput setting of 1200 packets per second, for the *Cornell5*, *Emulab2* and *Emulab3* scenarios. The top half of each graph shows the histogram for the inter-arrival time intervals for swapped packet pairs, and the vertical line in each graph is the 95th percentile of these intervals. The bottom half shows the inter-arrival time for all packet pairs, and as the vertical line continues down into this half, it indicates the percentage of inter-arrival times of all packet pairs that lie within it. The two metrics mentioned are stated on top of each distribution graph.

Why are we interested in knowing the percentage of all packet pairs that fall within the 95th percentile of swapped pair inter-arrival times? In the *Cornell3* 400 packet graph (top, left), 95% of all swaps and 14% of all packets have inter-arrival times of less than 128 microseconds. Hence, if we use an inter-arrival threshold of 128 microseconds to detect swaps - by raising a ‘red flag’ (we will elaborate later on what exactly this entails) whenever two packets arrive within that threshold time of each other - we would end up catching 95% of all swaps, and suspect 14% of all packet pairs of being swaps.

Figure 5 shows how the two metrics mentioned above vary with throughput, for the four different scenarios; this gives us a better understanding of how data rate affects the quality of prediction that can be obtained by observing the inter-arrival times. In the graph, the 95th percentile of inter-arrival times of swaps stays almost constant for all the scenarios - however, the percentage of all packet pairs that fall inside it goes up significantly

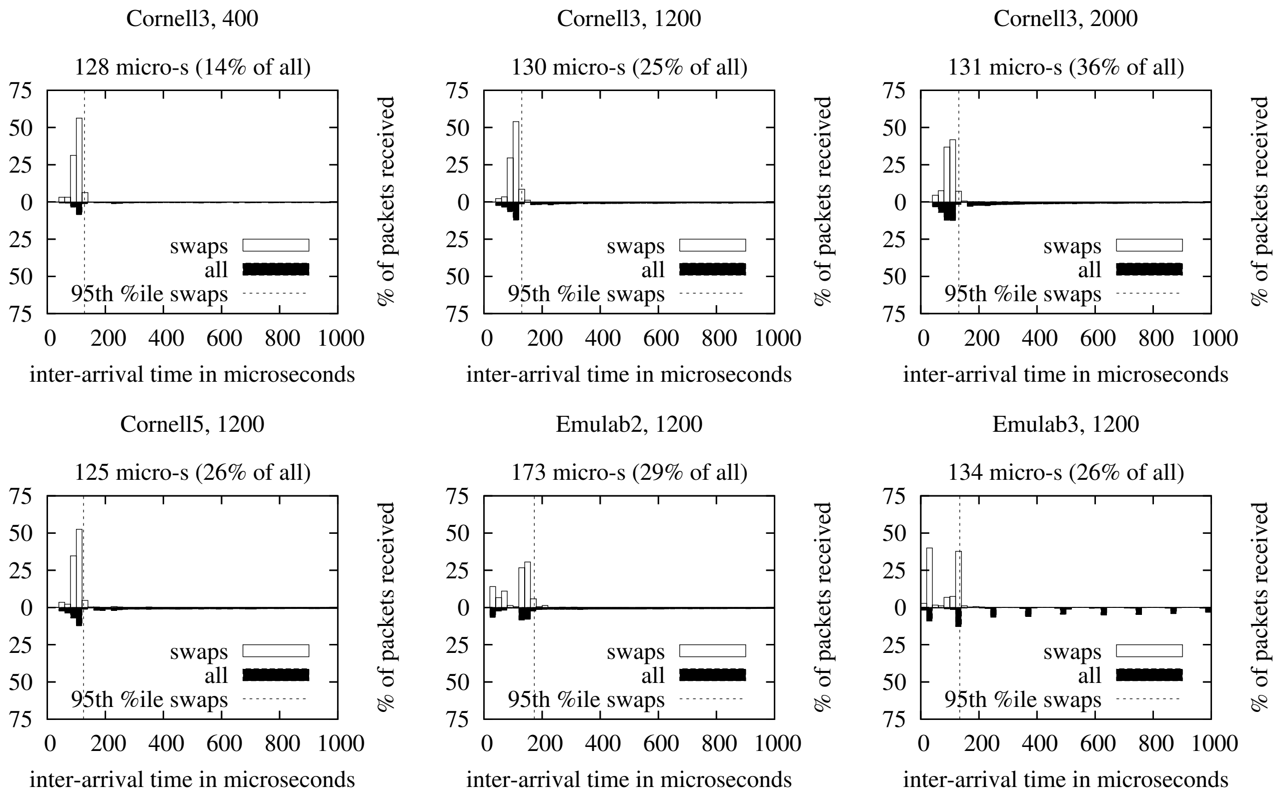


Figure 4. Histograms of Packet Inter-arrival Times

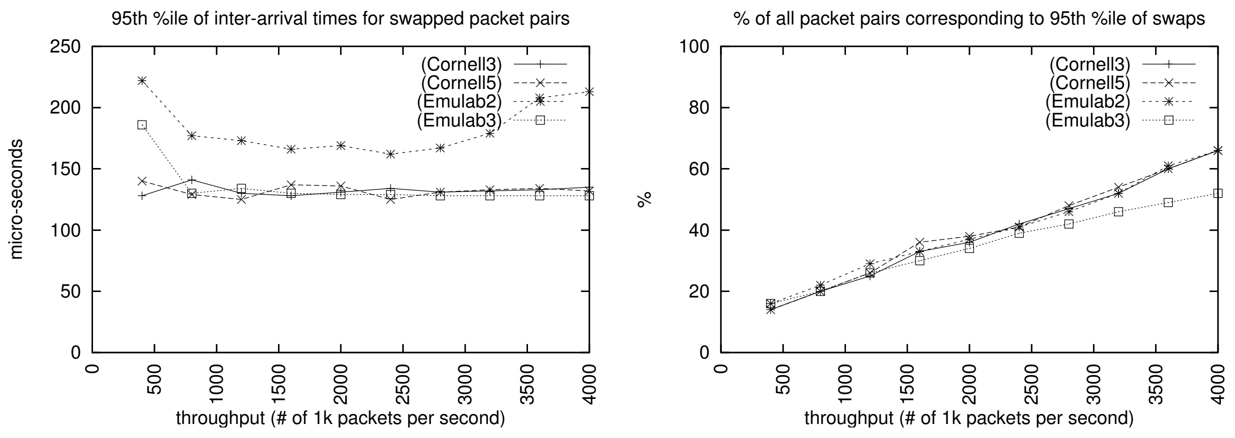


Figure 5. Variation of Swap metrics with throughput

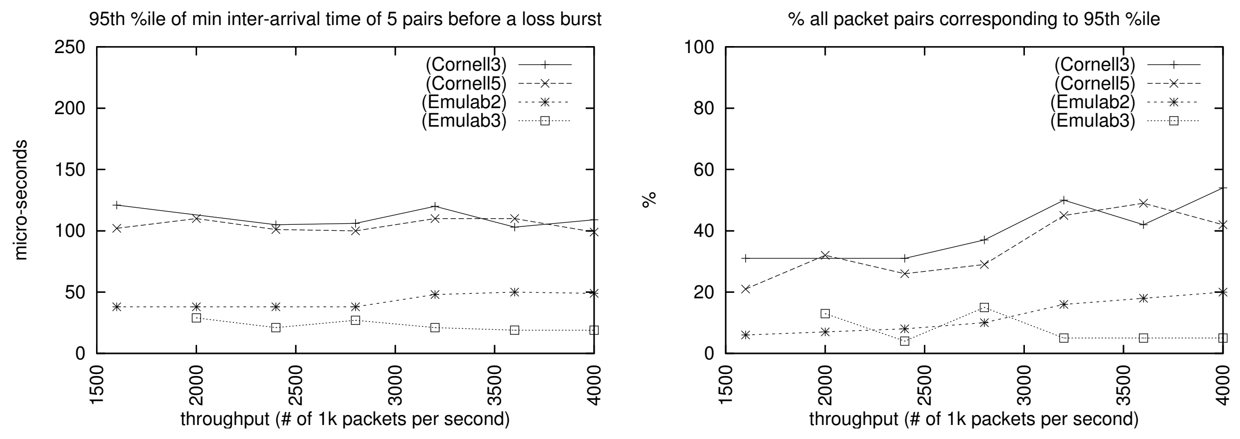


Figure 6. Variation of Loss metrics with throughput

as throughput rises. To continue using the metaphor of a 'red flag', a fixed threshold would catch all swaps, irrespective of throughput; however, it would also suspect a much higher percentage of all packet pairs of being swaps.

Next, we perform a very similar analysis of losses. Here, we measure the minimum inter-arrival time of the last five packet pairs immediately preceding a loss burst event. The rationale here is that the 'red flag' in this case gets raised when we observe a sequence of low inter-arrival times. Figure 6 shows this data; note that for certain throughput-scenario pairings we did not observe enough loss to compute a percentile, and we have not plotted these on the graph.

The data presented thus far leads us to formulate the following heuristic for detecting disorder, parameterized by a threshold Δ - *if the inter-arrival time between two packets at a receiver is less than Δ, it is reasonably probable that the packets have arrived in different order at other receivers; if the time is greater than Δ, it is highly improbable that the packets are delivered in different order at other receivers.*

This heuristic fits swap-induced disorders precisely - and, in our particular implementation of it, also suffices to catch loss-induced disorder; this will become clear once we describe our design.

4 The Design of a Predictive Ordering Protocol

To embed the heuristic presented above into a practical protocol, we need to examine the design space of total ordering algorithms. Defago, *et al.* [3] provide an analytical comparison of total ordering protocols, dividing them into five broad categories - *fixed-sequencer, moving-sequencer, privilege-based, communication history, and destinations agreement* - and conclude that for the non-uniform version of total ordering (where the delivery order at failed nodes does not matter), fixed-sequencer has the least latency and the second-highest throughput. The paper states that the moving-sequencer algorithm has slightly higher throughput than fixed-sequencer, but is more complicated to implement. Accordingly, we focus on the fixed-sequencer algorithm, both as a performance benchmark to compare against and as a starting point for our own design.

In the fixed-sequencer algorithm, a single receiver - the *sequencer* - periodically multicasts *sequencing messages*, establishing the correct order of delivery for the rest of the group. While most theoretical discussions of fixed-sequencer present the algorithm as sending out a single sequencing message for every received data message, in practice the sequencer can send out an ordered list of multiple received data messages in every sequencing message, allowing it to tune the overhead imposed by sequencing. If the sequencer sends one sequencing message for every k received data messages, one out of every $k+1$ messages in the group is pure sequencing overhead; we call k the *trade-off* parameter.

Armed with this basic understanding of the operation of the fixed-sequencer protocol, it becomes apparent that in most cases, messages are delayed unnecessarily while the receiver waits for ordering information from the sequencer. An ideal total ordering algorithm would delay only those packets which are received in different orders at different receivers, and deliver all other messages immediately to the application. This observation leads us towards a redesign of the fixed-sequencer protocol, where receivers wait for sequencing messages before delivering packets to the application only if they suspect them of arriving in the wrong order. What is needed is a decision mechanism that can be applied on a per-packet basis to selectively wait for sequencing information, and this is precisely the heuristic described in the previous section.

For a predictive mechanism to be a viable option, the application should provide some form of *rollback* capability to the protocol. We assume that the application provides the protocol with hooks that allow packets to be *revoked* once they are delivered, causing a rollback if the application has already consumed the packet. Since application rollbacks are likely to be very expensive, we aim at having a very low percentage of them - typically, one out of every thousand packets. Note that a packet revocation will only trigger an application rollback if the application has already consumed the packet.

4.1 PLATO: Design and Implementation

The basic idea behind PLATO is extremely simple: If two consecutive packets arrive within Δ of each other, they are suspected of arriving in incorrect order and further information is awaited from the sequencer node before they are delivered to the application.

A trivial implementation of this idea would involve delaying packets for Δ time and delivering them to the application if no other packet arrives within that time. However, Δ is likely to be in the tens or hundreds of microseconds, making an efficient implementation of this algorithm difficult if not impossible on commodity platforms - context switching granularity is typically in the milliseconds, and varies greatly over time and across hardware.

As a result, PLATO does not delay packets before passing them up to the application - instead, it tags each packet with a timestamp T_m before which it should not be used by the application, equal to Δ microseconds after its arrival time. Instead of sleeping for Δ microseconds and then waking up and delivering the packet to the application if no other packet arrives within that time-span, it just delivers the packet to the application and resumes listening on the socket. If another packet arrives within Δ microseconds, we revoke the last packet from the application instantly - since we are within the Δ envelope, we can be sure that the packet has not been consumed by the application, and hence the revocation does not trigger a potentially expensive application-level

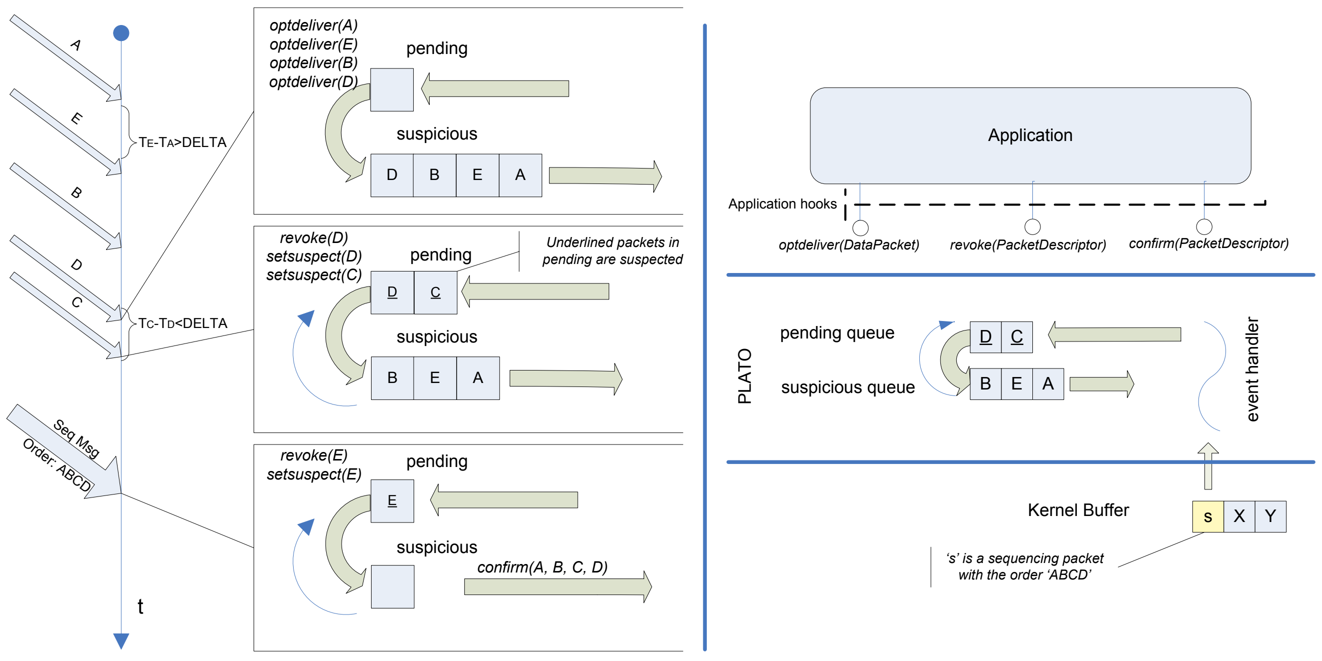


Figure 7. The PLATO Pipeline

rollback. An important metric for the protocol, then, is the frequency with which a revocation of a packet occurs after the corresponding time T_m has passed.

PLATO has three hooks into the application - *optdeliver*, which takes a data packet as a parameter and is called to optimistically deliver packets which may later be revoked; *revoke*, which takes a packet descriptor as a parameter and is called to revoke packets previously *optdeliver*-ed to the application, and *confirm*, which is called with the packet descriptor of a previously *optdeliver*-ed packet when the final ordering of that packet is known.

As shown in Figure 7, PLATO processes packets through a simple pipeline consisting of two queues - a *pending* queue, which consists of packets being conservatively withheld from the application, and a *suspicious* queue, which consists of packets already sent up to the application for which sequencing information has not yet arrived, and which can consequently be revoked from the application. Packets in the *pending* queue are marked *suspected* of being out-of-order and will not be delivered to application until sequencing information arrives for them; or, they are not *suspected* but are stuck in the queue behind one or more *suspected* packets. If no out-of-order arrivals occur, data packets travel through the *pending* queue to the *suspicious* queue, and then onto the application; if they do occur, the arrival of sequencing information can cause packets to be transplanted from the middle of one queue to the other, or to the application, violating queue FIFO order.

In addition, PLATO maintains a *sequencing* queue, which buffers sequencing information - this queue comes into play only if we receive sequencing information for a data packet which we have not yet received, and hence have to queue up all subsequent sequencing information until that data packet arrives and can be delivered to the application.

We now describe PLATO in terms of two simple events - the arrival of a data packet, and the arrival of a sequencing packet. When a data packet is received, PLATO tags it immediately with the arrival time. If ordering information for the data packet has already arrived from the sequencer, the packet is *optdeliver*-ed to the application and immediately *confirm*-ed, with no further processing. If not, the arrival time is compared with the previous data packet's arrival time, and the difference checked against Δ.

If the difference is less than Δ, the packet is tagged as *suspected*, and added to the *pending* list. Now we need to locate the previous data packet in the PLATO pipeline and prevent it from being used by the application. There are three possibilities -

1. It is in the *pending* queue and has not been *optdeliver*-ed to the application, in which case we can tag it as *suspected*.

2. It is the last packet in the *suspicious* queue and has been *optdeliver*-ed to the application, in which case we *revoke* it from the application and move it from the tail of the *suspicious* queue back to the head of the *pending* queue. Note that it is necessarily the last packet in the *suspicious* queue and cannot be in the middle, since it was the last packet to be received.

3. It is in neither the *suspicious* nor the *pending* queues, in which case it has already been sequenced and *confirm*-ed to the application. Nothing more has to be done in this case.

If the difference is more than Δ, the packet's fate depends on the contents of the *pending* queue. If the *pending* queue is non-empty - i.e, there are packets ahead of the current packet which are tagged *suspected* and are awaiting ordering information - then we need to add this packet to the end of the *pending* queue. If the *pending* queue is empty, then we can *optdeliver* the packet and add it to the end of the *suspicious* queue.

The second part of the protocol concerns its behavior when a sequencing packet is received. In its practical implementation, a sequencing packet contains a list of data packet descriptors - sorted by the order of arrival at the sequencer node. We iterate over this list of descriptors, and for each of them we locate the corresponding data packet within the PLATO pipeline (if we can't locate it in the pipeline, we buffer the descriptor - and all descriptors following it in this and subsequent sequencing packets - in the *sequencing* queue until we receive the data packet). Once we locate the data packet, we perform one of the following actions:

1. If the packet is in the *pending* queue, we remove it and *optdeliver* and *confirm* it to the application. We also dequeue all the packets from the *suspicious* queue, *revoke* them from the application, tag them as *suspected* and move them back to the head of the *pending* queue; these are packets incorrectly delivered to the application ahead of the currently sequenced packet.

2. If the packet is in the *suspicious* queue, we remove it and *confirm* it to the application. In this case, we have to dequeue all packets ahead of it in the *suspicious* queue, *revoke* them, tag them as *suspected* and move them back to the head of the *pending* queue.

4.2 Implementation and other Details

PLATO is implemented as an event-driven system with two threads, one running at high priority dequeueing packets off the socket and pushing them into the event queue, and the other servicing the queue and processing events. The implementation is written in Java - for our experiments, we use Java's System.nanoTime() method for determining the current system time at microsecond precision; this may not be universally portable, but is implemented on most modern platforms.

PLATO runs a link reliability layer that uses sender-based sequencing and negative acknowledgments to request unicast retransmissions of lost packets. Node failure is orthogonal to the protocol as it is presented, and any scheme that works to handle such faults in fixed-sequencer protocols should work equally well here. Also, while we have presented PLATO as a modification of fixed-sequencer, we could equally well have modified a moving-sequencer algorithm with similar results.

5 Performance Evaluation

To evaluate PLATO, we ran it in the Emulab 3-switch setting - recall that this involves a 3 Ghz sender-receiver pair and a 2 Ghz sender-receiver pair on the Utah Emulab testbed, with three switches separating them. Our first experiment was aimed at observing the impact of the Δ parameter on the performance of the protocol. Figure 8 plots delivery latency against the left y-axis as we run PLATO at increasing values of Δ. The left sub-graph shows performance at 400 packets per second, and the right sub-graph at 1600 packets per second, at two different values of k - the *trade-off* parameter (from Section 4). The horizontal fixed lines in these graphs show the performance of fixed-sequencer ordering for the same *trade-off* parameter, and consequently for the same overhead. The bars at the bottom of these graphs - plotted against the right y-axis - show the fraction of packets that are rolled back. We see from these graphs that the higher the value of Δ, the higher the delivery latency and lower the fraction of rollbacks - also, PLATO always out-performs fixed-sequencer for the corresponding value of k.

Next, we examine PLATO's performance in the presence of changing throughput levels. In Figure 9, we start out with 2 senders sending 200 packets/second each, and add 2 more senders 20 seconds into the experiment sending at 750 packets/second each for a total of 10 seconds - hence, the data rate in the group jumps from 400 packets per second to 1900 packets per second between time $t = 20$ and $t = 30$. On the left y-axis of these graphs, we plot 1-second moving averages every 10 milliseconds, of the delivery latencies achieved by PLATO and fixed-sequencer. As we can see from the graphs, PLATO's delivery latency remains constant throughout the experiment, whereas fixed-sequencer's delivery latency varies drastically with the data rate. The bars at the bottom of the

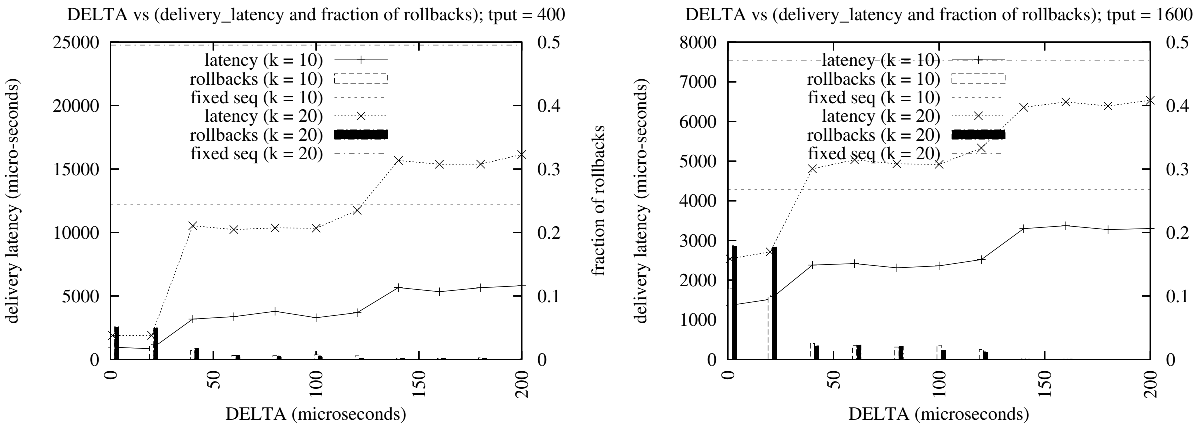


Figure 8. Δ vs Delivery Latency: 400 packets/second (left) and 1600 packets/second (right)

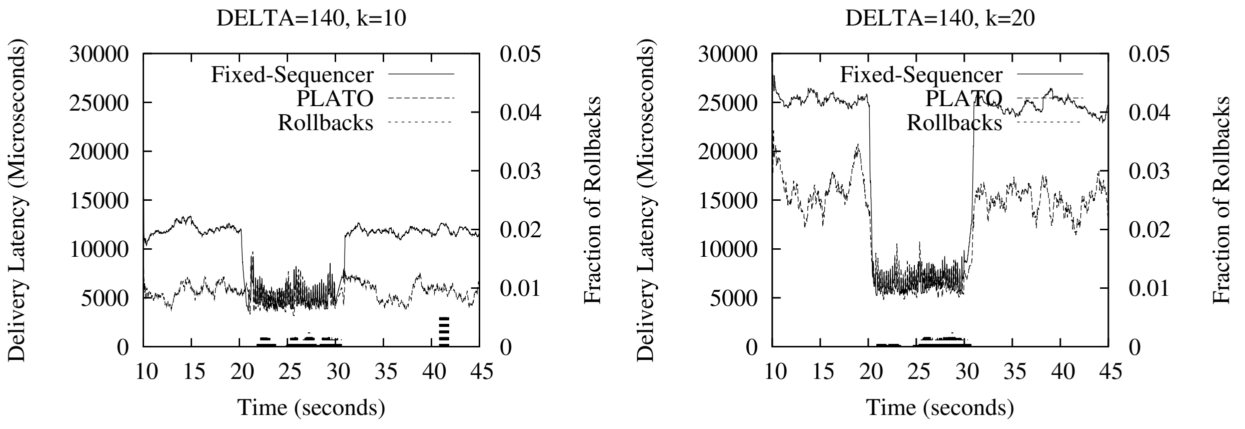


Figure 9. Traffic spikes up from 200 packets/sec to 1900 packets/sec between 20 and 30 seconds.

graphs is a 1-second moving fraction of rollbacks, plotted against the right y-axis. Note that for a higher value of k, delivery latencies are much higher for both protocols - since the latency to receiving ordering information from the sequencer node is higher.

Figure 10 shows how delivery latency and rollback fraction are affected by the data throughput, for a particular value of Δ. As the throughput goes up, the latency to receiving a sequencing packet goes down for a particular value of the *trade-off* parameter k, and consequently delivery latency drops. There is no real trend for rollbacks at this particular setting of Δ - all the values are within a tenth of a percent; however, for lower values of Δ we observe the fraction of rollbacks going up with throughput.

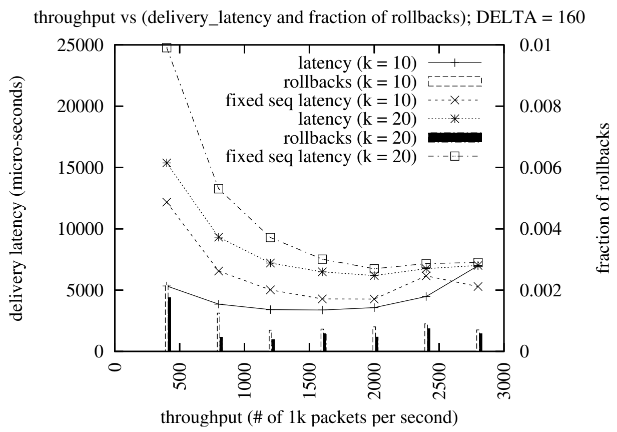


Figure 10. Throughput vs Delivery Latency

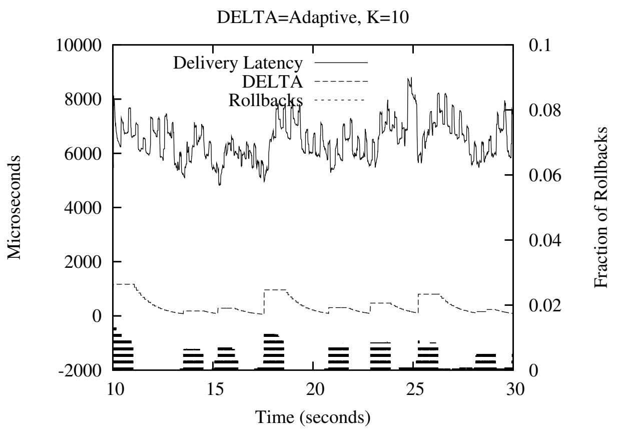


Figure 11. Setting Δ Adaptively

In Figure 11, we replace the static Δ parameterization of PLATO by a simple adaptive scheme. We multiply the current value of Δ by 1.5 when a rollback occurs and the current 1-second moving fraction of rollbacks is more than 0.01. Conversely, we multiply PLATO by $.9$ every 100 milliseconds or 1000 packets, whichever occurs first, if the moving fraction of rollbacks is less than 0.01. In Figure 11 we show that this simple mechanism gives good performance - for comparison, this setting is similar to the $k = 10$ scenario in Figure 9, during the traffic spike.

6 Related Work

A plethora of total ordering protocols exists in literature - we would like to point the reader to [4], which offers an excellent and thorough survey of this body of work, along with very useful categorizations. The particular subclass of ordering protocols that our work is closest to are the optimistic algorithms [6, 7, 9].

Sousa, et al. propose a solution for WANs where receivers observe network distances and delay packets appropriately to achieve a total ordering [8]. While this work is close in spirit to our own, it targets a completely different networking environment and uses a technique that works very well in the wide-area but may not be quite as useful in switched networks.

7 Conclusion

Low-latency data replication is a fundamental need for an emerging class of datacenter applications. PLATO is a total-ordering protocol designed for such settings - it targets the traffic patterns commonly observed in these applications and exploits the characteristics of the underlying hardware. We experimentally show that out-of-order delivery occurs to a reasonable degree on switched datacenter-style networks, and that the inter-arrival time of consecutive packet pairs is a powerful predictor of disorder.

8 Acknowledgments

We would like to thank Danny Dolev for his many valuable comments during the inception of this paper, Saikat Guha for a key discussion of protocol implementation, Mike Hibler for his quick responses on the Emulab testbed topology, and Art Munson and Einar Vollset for their feedback.

References

[1] M. Balakrishnan and K. Birman. Reliable multicast for time-critical systems. In *To Appear in Proceedings of the 1st Workshop on Applied Software Reliability (WASR 2006)*, 2006.

[2] S. E. Deering and D. R. Cheriton. Multicast routing in datagram internetworks and extended lans. *ACM Trans. Comput. Syst.*, 8(2):85–110, 1990.

[3] X. Défago, A. Schiper, and P. Urbán. Comparative performance analysis of ordering strategies in atomic broadcast algorithms. *IEICE Trans. on Information and Systems*, E86-D(12):2698–2709, December 2003.

[4] X. Défago, A. Schiper, and P. Urbán. Total order broadcast and multicast algorithms: Taxonomy and survey. *ACM Computing Surveys*, 36(4):372–421, December 2004.

[5] R. Guerraoui, R. Levy, B. Pochon, and V. Quma. High throughput uniform total order broadcast protocol for cluster environments. In *DSN*, June 2006.

[6] B. Kemme, G. Alonso, F. Pedone, and A. Schiper. Processing transactions over optimistic atomic broadcast protocols. In *19th IEEE International Conference on Distributed Computing Systems (ICDCS'99)*, 1999.

[7] F. Pedone and A. Schiper. Optimistic atomic broadcast. In *DISC*, pages 318–332, 1998.

[8] A. Sousa, J. Pereira, F. Moura, and R. Oliveira. Optimistic total order in wide area networks. In *SRDS '02: Proceedings of the 21st IEEE Symposium on Reliable Distributed Systems (SRDS'02)*, page 190, Washington, DC, USA, 2002. IEEE Computer Society.

[9] P. Vicente and L. Rodrigues. An indulgent uniform total order algorithm with optimistic delivery. In *SRDS*, pages 92–101, 2002.

[10] B. White, J. Lepreau, L. Stoller, R. Ricci, S. Guruprasad, M. Newbold, M. Hibler, C. Barb, and A. Joglekar. An integrated experimental environment for distributed systems and networks. In *Proc. of the Fifth Symposium on Operating Systems Design and Implementation*, pages 255–270, Boston, MA, Dec. 2002. USENIX Association.

Quicksilver Scalable Multicast (QSM)

Krzysztof Ostrowski[†], Ken Birman[†], and Danny Dolev[§]
[†]*Cornell University and* [§]*The Hebrew University of Jerusalem*
{krzys,ken}@cs.cornell.edu, dolev@cs.huji.ac.il

Abstract

QSM is a multicast engine designed to support a style of distributed programming in which application objects are replicated among clients and updated via multicast. The model requires platforms that scale in dimensions previously unexplored; in particular, to large numbers of multicast groups. Prior systems weren't optimized for such scenarios and can't take advantage of regular group overlap patterns, a key feature of our application domain. Furthermore, little is known about performance and scalability of such systems in modern managed environments. We shed light on these issues and offer architectural insights based on our experience building QSM.

1. Introduction

Web applications are becoming increasingly dynamic: users expect live content powered by real time data that can be modified, and demand a high-quality multimedia experience. To a degree these are opposing goals: peer-to-peer streaming systems are optimized to move data one-way; interactive client-server systems don't scale as well. Reliable multicast is a useful third option: instead of storing content on servers, we replicate it across clients and multicast updates between them in a peer-to-peer manner. Not every application can be built this way, but the model is broadly applicable; in particular, it is a good fit for interactive "mashups" of the sort shown on Figure 1. Each application object, such as a shared desktop or a text on the desktop, has its data replicated among all of its clients (the processes displaying it), which form a multicast group.

In [22], we described a general programming model based on this approach. Here, we focus on how to implement such systems. The key element of the platform is a multicast engine: most objects either have replicated state, or are driven by a stream of updates. Reliability is important, but weak guarantees are often sufficient: one rarely needs atomic transactions or consensus. Total ordering can be avoided: often updates originate at one or a few sources, and portions of objects can be locked prior to updating via separate locking facilities. More importantly, one needs high streaming bandwidth, low CPU footprint, and the ability to support large numbers of users, and large numbers of multicast instances. The last factor is important: each object could be accessed by a different group of clients, and so it needs its "own" separate instance of a multicast protocol.

Multicast groups corresponding to different objects often overlap in regular ways. To see this, consider airplane A, on a spatial desktop X of the sort shown on Figure 1. When the user opens X, the user can also see A. The group of X's clients is thus a subset of A's clients. Overlap occurs also if objects related to common topics are used by people with common interest. Furthermore, in [28] we show that even if objects are used at random, we can often partition them into a small number of subsets such that groups in each set overlap hierarchically. Regularities in group overlap occur naturally in our application domain. As we shall demonstrate, they could be leveraged to amortize overhead across protocol instances.

In the work reported here, we focus on enterprise computing environments: centers with thousands of commodity PCs running Microsoft Windows and communicating on a shared high-speed LAN. While our broader vision can be applied in WAN scenarios, the path to broad adoption of the paradigm leads through successful use in corporate settings and on campus networks, in context of applications such as collaborative editing, interactive gaming, online courses and videoconferencing, or distributed event processing. Accordingly, we assume an environment with system-wide support for IP multicast, in which packet loss is relatively uncommon and bursty (mostly triggered by overloaded recipients).

QSM is a system designed with precisely the objectives articulated above: it offers a simple, but useful reliability guarantee despite bursty loss. It streams at close to network speeds, at rates up to 9500 packets/s in a cluster of 1.3GHz/512MB nodes on a 100Mbps

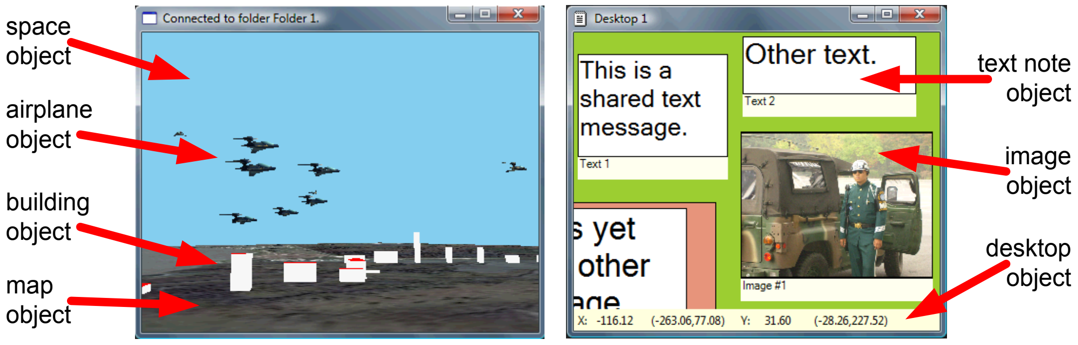


Figure 1. Example mashups enabled by our live objects platform and targeted by QSM. Each object shown here is replicated among its clients and backed by a multicast protocol. Other examples and videos of the platform in use can be found at http://liveobjects.cs.cornell.edu.

LAN. It has a low CPU footprint: 50% CPU usage on receivers at the highest data rates. Throughput falls by 5% as group size increases to 200 nodes and degrades gracefully with loss. QSM exploits regularities in group overlap, and amortizes protocol overheads across sets of nodes with a similar group membership.

In summary, this paper makes the following contributions:

- We propose a novel approach to scaling reliable multicast with the number of groups by leveraging regularity in the group overlap patterns to amortize overhead across protocol instances.
- We discover a previously unnoticed connection between memory usage and local scheduling policy in managed runtime environment (.NET), and multicast performance in a large system.
- We describe several techniques, such as priority I/O scheduling or pull protocol stack architecture, developed based upon these observations, which increase performance via reducing instabilities causing broadcast storms, unnecessary recovery and other disruptive phenomena.

2. Prior Work

Reliable multicast is a mature area [13] [16] [19] [23] [24], but existing systems were not optimized for our scenario. Most systems were designed for use with one group at a time. Some don't support multiple groups at all, others run separate protocol instances per group and incur overhead linear in the number of groups. Popular toolkits such as JGroups [2] perform best at small scale [3] and aren't optimized for network speeds. Systems that use IP multicast and run a protocol per group also suffer from *state explosion*: large numbers of IP multicast addresses in use require hardware to maintain a lot of state; this caused many data centers to abandon IP multicast based products. Also, the ability of NICs at client nodes to filter IP multicast is limited. With 1000s of addresses in active use local CPU is involved even on nodes that didn't subscribe to any of these addresses.

Systems like Isis and Spread can support lots of groups [1] [12] [26], but these groups are an application-level illusion; physically, there is just one group. In Isis, it consists of the members of all application groups. Spread uses a smaller set of servers to which client systems connect: each application-level message is sent to a server, multicast among servers and filtered depending on whether a server has clients in the destination group or not, and finally, relayed by servers to their clients. Support for large numbers of groups in such systems comes at a high cost: unwanted traffic and filtering in software, extra hops and higher latency, and bottlenecks created by the servers.

Application-level multicast systems such as OverCast, NARADA, NICE or SplitStream have been proposed [4] [6] [8] [14] that do not use IP multicast. These are very scalable, but messages follow circuitous routes, incurring high latency. In enterprise LANs, where IP multicast is available, such solutions simply don't fully utilize existing hardware support. Moreover, nodes are asked to forward messages that don't interest them at very high rates. This imposes additional overheads.

SpiderCast [7] and the techniques proposed in [27] have addressed the issue of scaling with the number of groups, but only in the context of unreliable multicast. Their overall approaches differ significantly from ours.

3. Protocol

QSM's approach to scalability is based on the concept of a *region*. A region is a set of nodes that are members of the same multicast groups. The system is partitioned into regions (Figure 2) by a *global*

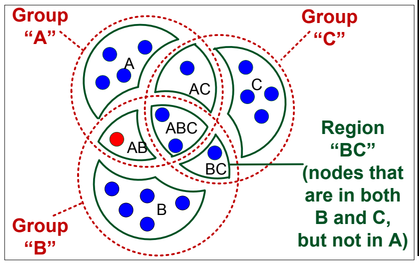


Figure 2. Groups overlap to form regions. Nodes belong to the same region if they joined the same groups.

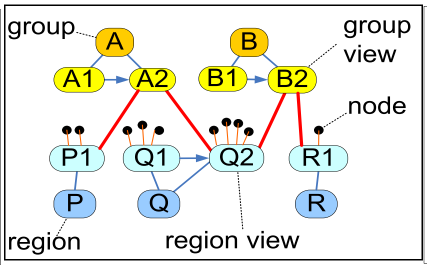


Figure 3. GMS keeps a mapping from groups to regions. Nodes use it to reliably construct distributed structures.

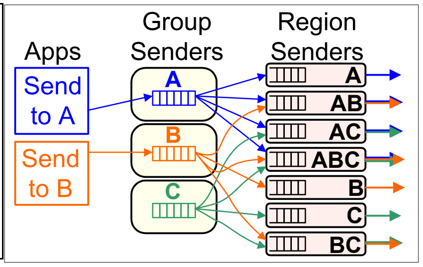


Figure 4. To multicast to a group, QSM sends the message to each of the regions that the group spans over.

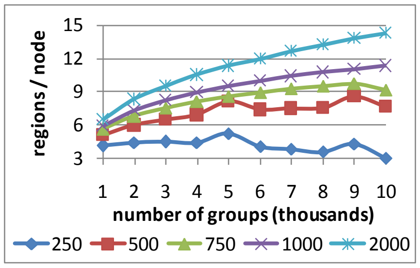


Figure 5. A set of irregularly overlapped groups is partitioned into a small number of regularly overlapped subsets.

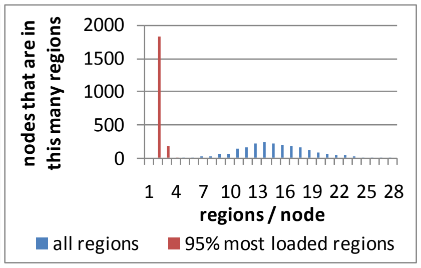


Figure 6. Most of the traffic a node sees is concentrated in just 2 of the regions of overlap to which it was assigned.

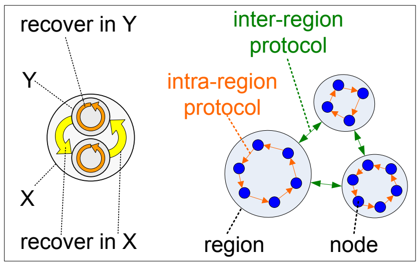


Figure 7. Recovery is done in a hierarchical manner, first locally, and then globally, via a hierarchy of token rings.

membership service (GMS [15]). Nodes contact the GMS to join groups, and it monitors their health. As the system changes, the GMS maintains a sequence of group and region *views* (sets of nodes that were members of given groups and regions at given points in time) and a mapping from one to the other (Figure 3). Each group view in this mapping is mapped to all region views that contain members of this group. The relevant parts of it are distributed to nodes, and are used to construct distributed structures, such as token rings, in a consistent manner. The GMS also assigns an individual IP address to each region. Multicasting to a group is then done by transmitting the message to each region the group spans over, using a single per-region IP multicast (Figure 4).

This scheme is less bandwidth-efficient than multicasting to a per-group IP multicast address, but it avoids the address explosion problem mentioned earlier: there are fewer regions than groups, and each node only needs to subscribe to a single IP multicast address at a time. The technique is most efficient when overlap is regular enough, so that regions consist of at least a few nodes, and each group maps to at most a few regions. As mentioned earlier, by using a technique described in [28], this can be achieved even in scenarios with irregular overlap, by partitioning groups into subsets. In a nutshell, we start by choosing the largest group, and then look for another group that is either contained in it, or overlaps with it in a way that doesn't produce small regions. We keep adding groups to the set as long as maintaining regular overlap is possible, and then simply start a new set of groups: pick the largest, and proceed as before. In the end, this greedy scheme yields a partitioning of groups such that in each partition, groups overlap regularly. The GMS can then simply run multiple instances of itself, each instance maintaining a mapping of the sort described before. Each node may now be a member of a few regions: at most one region for each subset of groups.

Only practice can tell how well this scheme works in real usage scenarios, but simulation results are promising. Figure 5 shows an experiment, in which a varying number of nodes (250 to 2000) subscribed to some 10% of a varying number of groups (1000 to 10000), using Zipf popularity with parameter α=1.5. Several studies suggest that this scenario is realistic [11] [17] [25]. After partitioning, an average node belongs to between 4 and 14 regions. Additionally, 95% of the packets a node will receive is concentrated in only 2 of these (Figure 6; here 2000 processes each join some

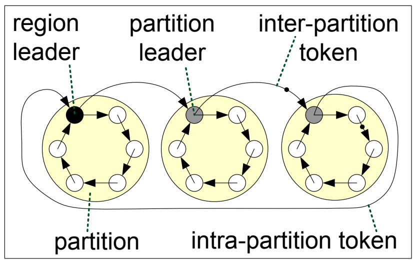


Figure 8. Token rings at the higher levels in the hierarchy are run by designated partition and regions leaders.

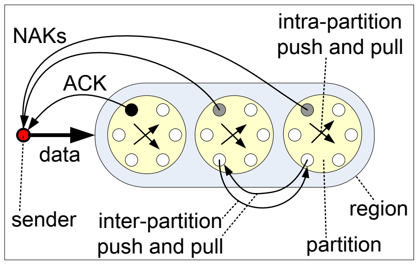


Figure 9. Per-partition rings enable recovery within partitions. Regional rings enable recovery across partitions.

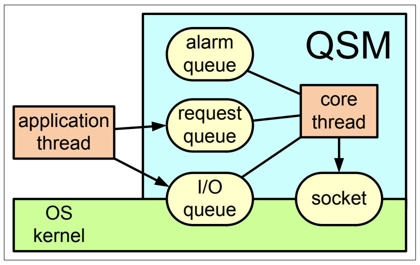


Figure 10. A single thread in QSM processes I/O, timer and application events based on its internal scheduling policy.

10% of a set of 10000 groups). In these experiments, no group is ever fragmented into more than ~5 regions. Regions generally contained 5-10 members, and the most heavily loaded ones often reflected the intersection of ~10 or more groups. This result suggests that in our design and analysis we can henceforth focus on just a single subset of regularly overlapping groups.

Recovery in QSM is hierarchical; the basic idea is to recover as locally as possible (Figure 7). To achieve this, groups are subdivided into smaller and smaller entities. First, a group is divided into regions it spans across. Each region is then subdivided into *partitions* of a fixed size. Each partition runs a local recovery protocol to ensure that its members received the same messages. Each region runs a higher-level protocol to ensure that all of its partitions received the same messages. Finally, a protocol run at an inter-regional level ensures that if the entire region lost a message, it is recovered from the source.

At each level, recovery is performed by a token ring (Figure 8, Figure 9). At the lowest level the ring is run by nodes in a partition. The token is carrying ACKs and NAKs, which neighbors on the ring compare, and use to initiate push or pull recovery from each other. The token is also used to calculate collective ACK and NAK data, describing the status of the entire partition (messages lost or received by all). The aggregate is intercepted by a designated *partition leader*, which participates in a higher-level ring. Again, neighbors compare ACKs and NAKs collected from partitions they represent, and use these to initiate recovery across partitions and calculate aggregate ACKs/NAKs for the entire region. These are collected by the *region leader*.

In QSM, this recursive scheme is only 3 levels deep, but it could be generalized [20] [21]. The overhead is extremely low: in our experiments, tokens circulate only 1 to 5 times/s. This is a key factor enabling high performance. Despite the low overhead, recovery in QSM is efficient, in part thanks to QSM's cooperative caching similar to [5], and massively parallel recovery. In each region only one partition, selected in a round-robin fashion, keeps a copy of each message. When a long burst of messages is lost by a node, often every partition is involved in recovery, and since nodes to recover from are picked at random among those in a partition, often the entire region is helping to repair the loss. The efficiency of this technique is especially high in very large regions.

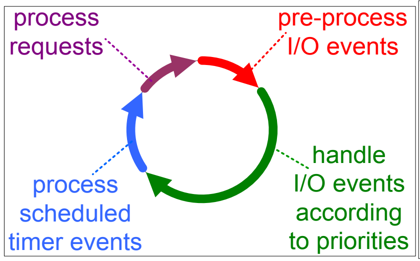
The key to QSM's scalability edge is the observation that, since nodes in the region are all members of the same groups, they receive exactly the same messages. Thanks to this, QSM can run a *single* token ring in each region and partition, and use it to perform recovery simultaneously for *all* groups.

The benefits are two-fold. First, each node in a region has at most four neighbors, and receives a fixed small number of control packets/s. This is in contrast to systems that run a separate protocol per group, where nodes may have unbounded in-degrees and experience arbitrary rates of control traffic. Furthermore, recovery overhead in each region depends *only* on the number of currently active senders, *not* on the number of groups or the total number of senders in the system. This is because when sending to a region, senders can index messages on a per-region basis, across groups.

The token carries a separate recovery record on a per-sender basis. When a sender ceases to actively multicast, in 2-3 token rounds QSM removes its record from the token, to reintroduce it again next time a message from this sender is seen in the region.

4. Architecture

We implemented QSM as a .NET library (mostly in C#). In the course of doing so, we found landmark features of managed environments, such as garbage collection and multithreading, to have surprisingly strong performance implications.



process requests
pre-process I/O events
handle I/O events according to priorities
process scheduled timer events

Figure 11. QSM uses custom time-sharing scheme, with a quantum per event type. I/O is handled like interrupts.

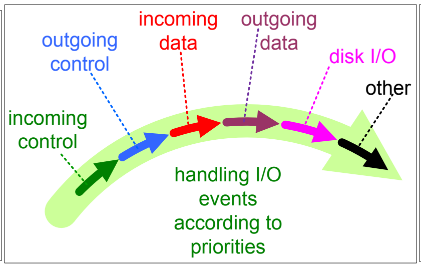
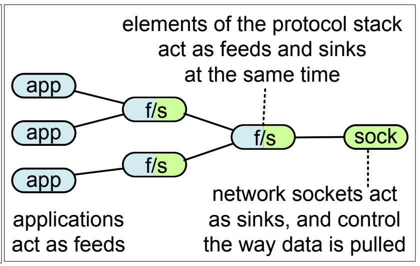


Figure 12. QSM prioritizes I/O events: control packets and inbound I/O are handled ahead of everything else.



elements of the protocol stack act as feeds and sinks at the same time
app
app
app
f/s
f/s
f/s
sock
applications act as feeds
network sockets act as sinks, and control the way data is pulled

Figure 13. Elements of QSM protocol stack form trees rooted at sockets. Sockets pull data from "their" trees.

QSM is single-threaded and event-driven: a dedicated core thread processes events from three queues. An *I/O queue*, based on a Windows I/O completion port, collects asynchronous I/O completion notifications for all network sockets or files used by QSM. An internal *alarm queue* based on a splay tree stores timer events. Finally, a lock-free *request queue* implemented on CAS-style operations allows the core thread to interact with other threads (Figure 10). The core thread polls its queues in a round-robin fashion and processes events of the same type in batches (Figure 11), up to the limit determined by its quantum (50ms for I/O, 5ms for timer events; no limit for application requests), except that if an incoming packet is found on a socket, the socket is drained of I/O to reduce the risk of packet loss. For I/O events, QSM further prioritizes their processing, in a manner reminiscent of interrupt handling. First, events are read off the I/O queue, and scattered across 6 priority queues. Then, they are handled in priority order. Inbound I/O is prioritized over outbound I/O to reduce packet loss and avoid contention. Control or recovery packets are prioritized over regular multicast, to reduce delays in reacting to packet loss (Figure 12). The pros and cons of using threads in event-oriented systems are hotly debated. In our case, multithreading was not only a source of overhead due to context switches, but more importantly, a cause of instabilities, oscillatory behaviors, and priority inversions due to the random processing order. Eliminating threads and introducing custom scheduling let us take control of this order, which greatly improved performance. In Section 5 we will see that the latency of control traffic is the key to minimizing memory overheads, and as a result, it has a serious impact on the overall system performance.

Control latencies and memory overheads motivate another design feature: a pull protocol stack architecture (Figure 13). QSM avoids buffering data, control, or recovery messages, and delays their creation until the moment they're about to be transmitted. The protocol stack is organized into a set of trees rooted at individual sockets, and consisting of *feeds* that can produce data and *sinks* that can accept it. Feeds register with sinks. Sinks pull data from registered feeds according to their local rate, concurrency, windows size, or other control policy. Using this scheme yields two advantages. First, bulky data doesn't linger in memory and stress a garbage collector. Second, information created just in time for transmission is fresher. ACKs and NAKs become stale rather quickly: if sent after a delay, they often trigger unnecessary recovery or fail to report that data was received. Likewise, recovery packets created upon the receipt of a NAK and stored in buffers are often redundant after a short period: meanwhile, the data may be recovered via other channels. Postponing their creation prevents QSM from doing useless work.

5. Evaluation

In our evaluation of QSM, we focus on scalability and on the interactions of the protocol with the runtime environment that have driven our architectural decisions. The experiments we report reveal a pattern: in each scenario, performance is limited by overheads related to memory management in .NET, which grow linearly with the amount of memory in use, causing the .NET CLR to steal CPU cycles from QSM. Managing the use of memory within QSM turned out to be the key to achieving stable, high performance.
In Section 5.1 and Section 5.2 we show that memory overhead at senders and receivers is linked to latency, and that latency is affected by the overhead it causes. In Sections 5.3 and Section 5.4, we show that this is also true also if the system is perturbed or if it is not saturated. In Section 5.5 we show how the number of groups can cause such effects.

Our results are abbreviated for lack of space (details can be found in our technical report). All results were obtained on a 200-node Pentium III 1.3 GHz, 512

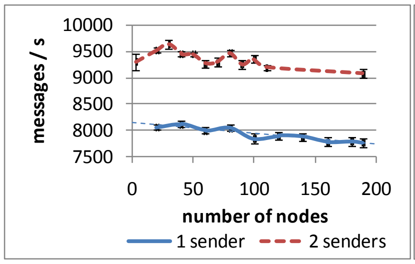


Figure 14. Max throughput in messages/s as a function of the number of receivers with 1 group and 1KB messages.

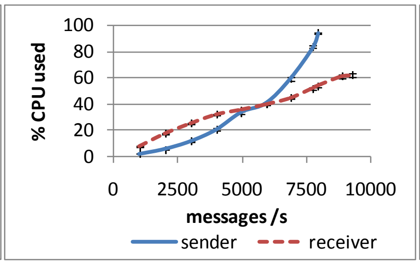


Figure 15. % CPU utilized as a function of multicast rate (single group, 100 receivers).

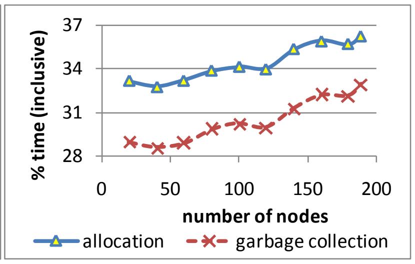


Figure 16. % of time spent on memory-related tasks on the sender: allocation and garbage collection in CLR code.

MB cluster, on a 100 Mbps LAN, running Windows Server 2003, .NET 2.0. Our benchmark is an ordinary .NET executable, using QSM as a library. We send 1000-byte arrays without pre-allocating them, with no batching, at the maximum rate. Our 95% confidence intervals were not always large enough to be visible.

5.1 Memory Overheads on the Sender

Figure 14 shows throughput in messages/s as a function of the number of receivers (all in a single group). Why does performance decrease with the number of receivers? Figure 15 shows that receivers are not CPU-bound, but the sender is. Profiling reveals that the CLR at the sender is taking over the CPU; specifically, memory allocation and garbage collection costs are growing by as much as 10-15% (Figure 16). Inspecting the managed heap shows that memory is used mostly by the multicast messages pending ACK on the sender: these have to be buffered for the purpose of loss recovery. Memory usage is actually 3 times larger than what the number of pending messages would suggest: at these high data rates, the CLR can't garbage collect fast enough, hence old data is still lagging in memory. Acknowledgement latency is caused by the increase in the time to circulate a token around the region for the purpose of state aggregation (token "roundtrip time"). Hence, our throughput degradation is ultimately caused by the latency to collect control information by the protocol. Just a 500ms increase in token RTT resulted in 10MB extra memory usage, inflated overhead by 10-15%, and degraded throughput by 5%. The need to reduce this latency to ensure a smooth token flow was among the key reasons for the architectural decisions outlined in the preceding section. Using a deeper hierarchy of token rings would also help to alleviate this problem (and indeed, this idea led us to the design proposed in [20]). On the other hand, simply increasing the token rates helps only up to a point (Figure 17), and causing tokens to carry larger amounts of feedback per round by making this amount proportional to the region size increases processing complexity, and despite memory saving, it is counterproductive (Figure 18).

5.2 Memory Overheads on the Receiver

The growth in cached data at the receivers repeats the pattern of performance linked to memory. The increase in the amount of such data slows us down, despite the fact that receiver CPUs are half-idle (Figure 15). How can memory overhead affect a half-idle node? Figure 19 shows results of an experiment where we varied *replication factor*, the number of receivers caching a copy of each message, causing a linear increase of memory usage. We see a super-linear increase of the token roundtrip time and a slow increase of the number of messages pending ACK on the sender, causing a sharp decrease in throughput (Figure 20). The underlying mechanism is as follows. The increased garbage collector activity and allocation overheads slow nodes down, and processing of the incoming packets and tokens takes more time. Although this is not significant on just a single node, it accumulates, since a token must visit all nodes to aggregate state. Increasing the number of caching replicas from 5 to all 200 nodes in the region, increases token RTT 3-fold!

5.3 Overheads in a Perturbed System

Another question to ask is whether our results would be different if the system experienced high loss rates or was otherwise perturbed. To find out, we performed two experiments. In the "sleep" scenario, one of the receivers experiences a periodic, programmed perturbation: every 5s, QSM instance on the receiver suspends all activity for 0.5s. This simulates the effect of an OS overloaded by disruptive applications. In the "loss" scenario, every 1s the node drops all incoming packets for 10ms, thus simulating 1% bursty packet loss. In practice, the resulting loss rate is up to 2-5%,

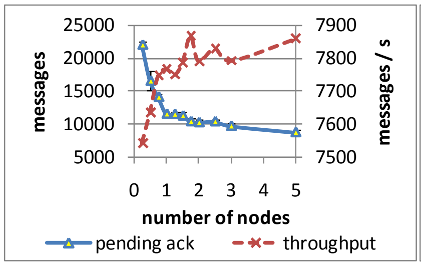


Figure 17. Throughput and the # of messages pending ACK as a function of token circulation rates.

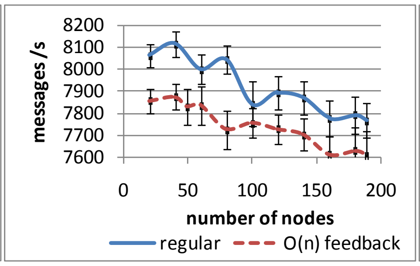


Figure 18. With O(n) feedback performance is worse due to higher overhead, despite the savings on memory usage.

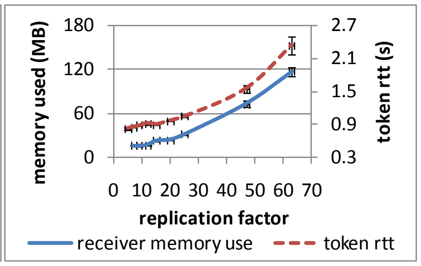


Figure 19. Results of varying the number of caching replicas per message in a 192-node region.

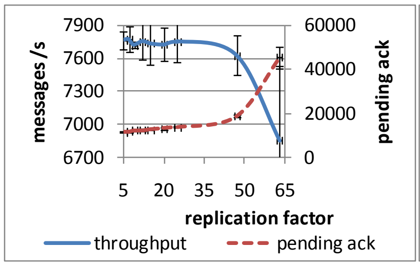


Figure 20. As a # of caching replicas increases, throughput decreases despite CPUs on receivers being 50% idle.

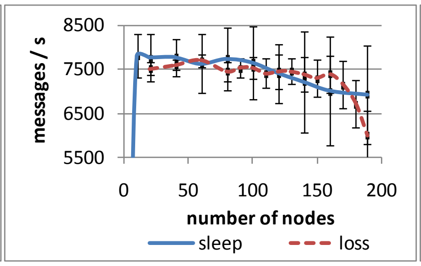


Figure 21. Throughput in the experiments with a perturbed node (1 sender, 1 group).

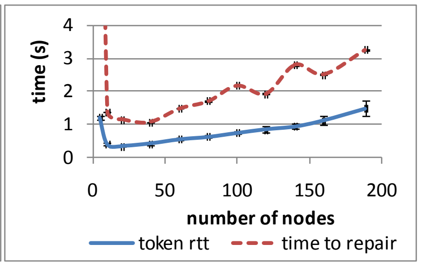


Figure 22. Token roundtrip time and the time to recover in the "sleep" scenario.

because recovery traffic interferes with regular multicast, causing further losses. In both scenarios, CPU usage at the receivers is in the 50-60% range and doesn't grow with system size, but the throughput decreases (Figure 21). In neither case does this decrease in throughput seem to be correlated to the loss rate. It does, however, correlate perfectly to the token RTT and memory utilization on the sender (Figure 22, Figure 23), repeating again the by now familiar pattern.

A closer look at this experiment reveals that while the increased ACK latency and resulting memory usage can be explained by the extra token rounds needed to perform recovery, the 2-fold overall increase in token RTT in these scenarios, as compared to undisturbed experiments, can't be as easily explained. The problem can be traced to a priority inversion. Because of repeated losses, the system maintains a high volume of forwarding traffic. Forwarded data tends to get ahead of tokens both on a sending and on a receiving path. As a result, tokens are slowed down.

5.4 Overheads in a Lightly-Loaded System

We've just discussed a perturbed system, now what if it's lightly loaded? We'll see that load has a super-linear impact on overheads. As we increase the multicast rate, the linear growth of traffic, combined with our fixed rate of state aggregation, linearly increases the amount of unacknowledged data and memory usage on the sender (Figure 24). This triggers higher overheads. For example, the time spent in the GC grows from 50% to 60%. Combined with the linearly growing demand for CPU due to the increasing volume of traffic, these effects together cause the super-linear growth of CPU overhead on the sender (Figure 15). The overhead skyrockets at the highest rates because the increasing amount of I/O slows down processing of control messages; much as in Section 5.2. We can confirm this by looking at the end-to-end latency (Figure 25), or at the delay in firing timer events (Figure 26), which at the highest rates get starved by the I/O.

5.5 Per-Group Memory Consumption

In this set of experiments, we explore scalability in the number of groups. One sender multicasts to a varying number of groups, in a round-robin fashion. Each receiver joins the same groups; the system contains just one region. QSM's regional recovery protocol is oblivious to the groups, but the system maintains a number of per-group data structures, which affects the memory footprint (Figure 27). Memory being involved, we expect the familiar pattern, where an increased memory usage triggers GC and decreases the throughput, and

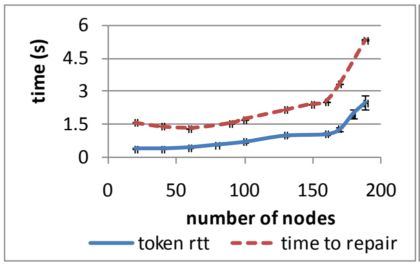


Figure 23. Token roundtrip time and the time to recover in the "loss" scenario.

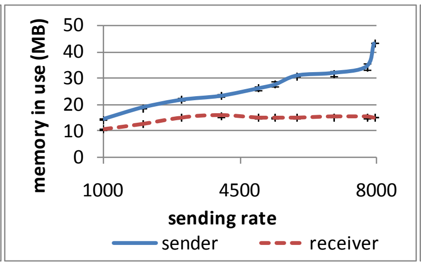


Figure 24. Linearly growing memory use on a sender and flat usage on receivers as a function of the sending rate.

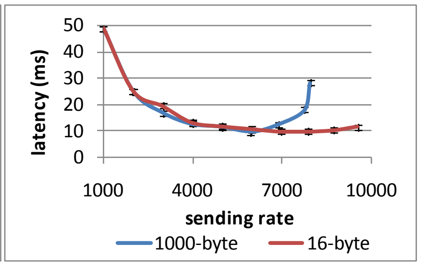


Figure 25. Latency measured from sending to receiving for varying sending rate and with various message sizes.

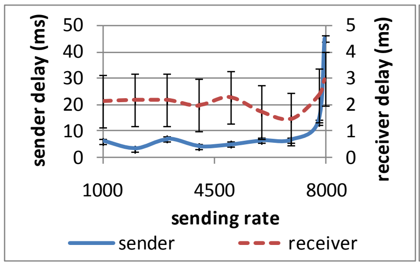


Figure 26. Delays in firing of timer events as a function of the sending rate, demonstrating “starvation through I/O”.

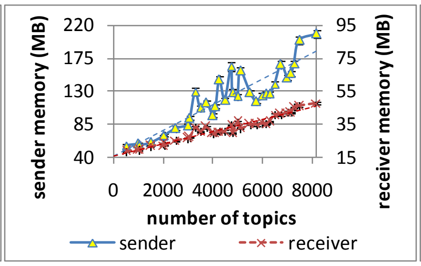


Figure 27. Memory use grows with the # of groups. Beyond a threshold, the system becomes increasingly unstable.

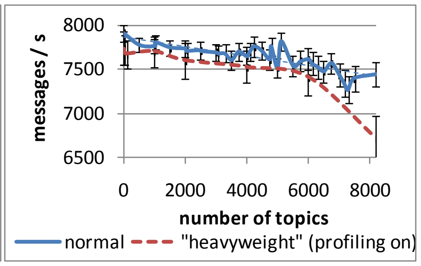


Figure 28. Throughput decreases with the # of groups (1 sender, 110 receivers, all groups perfectly overlap).

this is indeed the case (Figure 28). The effect becomes even clearer if we turn on extra tracing in per-group protocol stack components. This tracing is lightweight and has no effect on CPU, but increases memory usage, which burdens the GC. As expected, now throughput degrades even more (Figure 28, “profiling on”).

A closer look at this scenario provides an even deeper insight. Note how at 6000 groups throughput degrades sharply (Figure 28) due to the increased token RTT and control latency (Figure 29). The growth of overhead suddenly becomes super-linear, and event at 4000 groups we are actually starting to see spikes of occasional packet loss, clear signs of slight instability. Detailed analysis again points to sender overhead as the culprit. Most delays come from 40% of tokens (Figure 30), since it is caused by disruption in their flow, not system-wide increase of overhead. This disruption is caused by the sender, which is busy and delays about 10% of the tokens (Figure 31), causing irregularities in their flow. The magnitude of this delay increases with the number of groups.

6. Discussion

Our experiments clearly show that memory is a performance-limiting factor in QSM, and that its cost is tried to latency by a positive feedback loop. Our results aren’t specific to QSM and .NET; while managed environments do have overheads, we believe the phenomena we’re observing are universal. Application with large amounts of buffered data may incur high context switching and paging delays, and even minor tasks get costly as data structures get large. Memory-related overheads can be amplified in distributed protocols, manifesting as high latency when nodes interact. Traditional protocol suites buffer messages aggressively, so existing multicast systems certainly exhibit such problems no matter what language they’re coded in or what platform they use. The mechanisms QSM uses to reduce memory use, such as event prioritization, pull protocol stack or cooperative caching, should therefore be broadly useful. Below, we list our design insights.

1. ***Exploit structural regularity.*** We’ve recognized that even in irregular overlap scenarios one can restructure the problem to arrange for regularities, which can then be exploited by the protocol. This justified focus on optimizing performance in the scenario with a single heavily loaded set of regularly overlapping groups.
2. ***Minimize memory footprint.*** This applies especially to messages cached for recovery purposes.
   a. ***Pull data.*** Most protocols accept data whenever the application or a protocol layer pro-

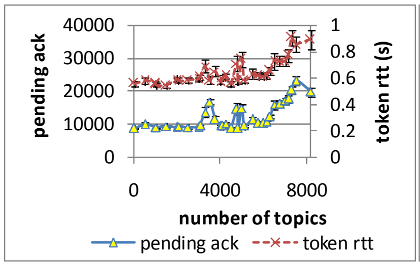


Figure 29. # messages pending ACK and a token RTT as a function of the # of perfectly overlapping groups.

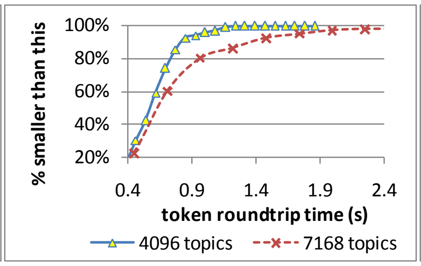


Figure 30. Cumulative distribution of the token RTT with 4096 and 7168 groups.

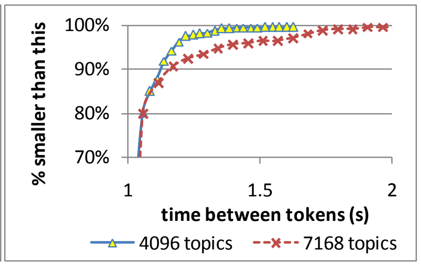


Figure 31. Cumulative distribution of the intervals between the subsequent tokens with 4096 and 7192 groups.

duces it. In contrast, by using an upcall driven "pull" architecture, QSM can delay generating messages until the very last moment and thus prevents data from piling up in the buffers.

b. ***Limit buffering and caching***. Most protocols buffer and cache data rather casually for recovery purposes. QSM avoids buffering and uses distributed, cooperative caching. Paradoxically, by reducing memory overheads, the reduction in cached data allows for a much higher performance.

c. ***Clear messages out of the system quickly***. Data paths should have rapid data movement as a key goal, to limit the amount of time packets spend in the send or receive buffers.

d. ***Message flow isn't the whole story***. Most protocols are optimized for steady low-latency data flow. To minimize memory usage, QSM sometimes tolerates an increased end-to-end latency for data, so as to allow for a faster flow of the control traffic; this allows faster cleanup and recovery.

3. ***Minimize delays***. Most situations in which we observed convoys and oscillatory throughputs can be traced to design decisions that permitted scheduling jitter or some form of priority inversion, delaying crucial messages behind less important ones. Implications included the following.

a. ***Event handlers should be short, predictable and terminating.*** Using the event-driven model consistently allowed us to eliminate the need for locking or preemption; we obtained a more predictable system, and got rid of multithreading, with its associated context switching overheads.

b. ***Drain input queues.*** From a memory footprint perspective, one might prefer not to pull in a message until QSM can process it. In data centers and clusters, though, most losses occur in the OS, not in the network, and loss rates soar if packets are left in the system buffers for too long.

c. ***Control the event processing order.*** In QSM, this involved single-threading, batched asynchronous I/O, and internal event prioritization. Small delays add up in large systems: tight control over event processing largely eliminated convoy effects and the oscillatory throughput problems.

d. ***Act upon fresh state.*** Our pull architecture has the added benefit of letting us delay the preparation of status packets until they are about to be transmitted, thus minimizing the risk that nodes act on stale information and trigger retransmissions that aren't longer needed, or other overheads.

4. ***Handle disruptions gracefully***. Broadcast storms are triggered when recovery itself becomes disruptive, causing convoy effects or triggering bursts of even more loss. In addition to the above, QSM employs the following techniques to keep balance.

a. ***Limit resources used for recovery***. QSM limits the maximum rate of the recovery traffic and delays the creation of recovery packets to prevent such traffic from overwhelming the system.

b. ***Act proactively on reconfiguration***. Reconfiguration after joins or failures can destabilize the system: changes reach different nodes at different times and structures such as trees and rings can take time to form. To address this, senders in QSM briefly suspend multicast on reconfiguration and receivers buffer unknown packets for a while in case a join is underway.

c. ***Balance recovery overhead***. In some protocols, bursty loss triggers a form of thrashing. QSM delays recovery until a message is stable on its caching replicas; then it coordinates a parallel recovery in which separate point-to-point retransmissions can be sent concurrently by tens of nodes.

7. Conclusions

The premise of our work is that new options are needed for performing multicast in modern platforms, specifically in support of a new drag-and-drop style of distributed programming inspired by web mash-ups, and for use in enterprise desktop computing environments, or in datacenters where multi-component applications may be heavily replicated. Using multicast in such settings requires a new flavor of scalability - to large numbers of multicast groups - largely ignored in previous work. QSM achieves this by exploiting regularities and commonality of interest.

Our performance evaluations led to a recognition that memory can be surprisingly costly. The techniques that QSM uses to reduce such costs and maintain high stable throughput despite perturbations should be useful even in systems that do not run in managed runtime environments.

8. Acknowledgements

Our work was funded by AFRL/IF, with additional funds from AFOSR, NSF, I3P, and Intel. We want to thank Jong Hoon Ahnn, Mahesh Balakrishnan, Lars Brenna, Lakshmi Ganesh, Maya Haridasan, Chi Ho, Ingrid Jansch-Porto, Tudor Marian, Amar Phanishayee, Stefan Pleisch, Robbert van Renesse, Yee Jiun Song, Einar Vollset, and Hakim Weatherspoon for feedback.

9. References

[1] Y. Amir, C. Danilov, M. Miskin-Amir, J. Schultz, and J. Stanton. The Spread toolkit: Architecture and Performance. 2004.

[2] B. Ban. Design and Implementation of a Reliable Group Communication Toolkit for Java. 1998.

[3] B. Ban. Performance Tests JGroups 2.5. http:// jgroups.org/javagroupsnew/perfnew/Report.html

[4] S. Banerjee, B. Bhattacharjee, and C. Kommareddy. Scalable Application Layer Multicast. SIGCOMM'02.

[5] K. Birman, M. Hayden, O. Ozkasap, Z. Xiao, M. Budiu, and Y. Minsky. Bimodal Multicast. TOCS 17(2), 1999.

[6] M. Castro, P. Druschel, A-M. Kermarrec, A. Nandi, A. Rowstron, and A. Singh. SplitStream: High-Bandwidth Multicast in a Cooperative Environment. SOSP'03.

[7] G. Chockler, E. Melamed, Y. Tock, and R. Vitenberg. SpiderCast: a Scalable Interest-Aware Overlay for Topic-Based Pub/Sub Communication. ACM DEBS 2007.

[8] Y. Chu, S. G. Rao, S. Seshan, and H. Zhang. A Case for End System Multicast. IEEE JSAC 20(8), 2002.

[9] E. Decker. http://researchweb.watson.ibm.com/compsci/project_spotlight/distributed/dsc/.

[10] D Dolev, and D Malki. The Transis Approach to High Availability Cluster Communication. CACM 39(4), 1996.

[11] X. Gabaix, P. Gopikrishnan, V. Plerou, H. E. Stanley. A Theory of Power-Law Distributions in Financial Market Fluctuations. Nature 423, p. 267-270, 2003.

[12] B. Glade, K. Birman, R. Cooper, and R. van Renesse. Light-Weight Process Groups in the ISIS System. Distributed Systems Engineering. Mar 1994. 1:29-36.

[13] M. Handley, S. Floyd, B. Whetten, R. Kermode, L. Vicisano, and M. Luby. The Reliable Multicast Design Space for Bulk Data Transfer, RFC 2887, August 2000.

[14] J. Jannotti, D. K. Gifford, K. L. Johnson, M. F. Kaashoek, and J. W. O'Toole. Overcast: Reliable Multicasting with an Overlay Network. OSDI'00.

[15] I. Keidar, J. Sussman, K. Marzullo, and D. Dolev. Moshe: A Group Membership Service for WANs. ACM TOCS 20(3), p. 191-238, August 2002.

[16] B. N. Levine, and J. J. Garcia-Luna-Aceves. A Comparison of Reliable Multicast Protocols. Multimedia Systems 6: 334-348, 1998.

[17] H. Liu, V. Ramasubramanian, and E.G. Sirer. Client Behavior and Feed Characteristics of RSS, A Publish-Subscribe System for Web Micronews. IMC 2005.

[18] S. Maffeis, and D. Schmidt. Constructing Reliable Distributed Communication Systems with CORBA. IEEE Communication Magazine, Vol. 14, No. 2, Feb. 1997.

[19] L. E. Moser, P. M. Melliar-Smith, D. A. Agarwal, R. K. Budhia, C. A. Lingley-Papadopoulos, and T. P. Archambault. The Totem System. FTCS 25 (1995).

[20] K. Ostrowski, K. Birman, and D. Dolev. Extensible Architecture for High-Performance, Scalable, Reliable Publish-Subscribe Eventing and Notification. JWSR 4(4), 2007.

[21] K. Ostrowski, K. Birman, and D. Dolev. Declarative Reliable Multi-Party Protocols. Cornell University Technical Report, TR2007-2088. March, 2007.

[22] K. Ostrowski, K. Birman, D. Dolev, and J. Ahnn. Programming with Live Distributed Objects. ECOOP'08.

[23] C. Papadopoulos, and G. Parulkar. Implosion Control For Multipoint Applications. 10th Annual IEEE Workshop on Computer Communications, Sept. 1995.

[24] S. Pingali, D. Towsley, and J. F. Kurose. A Comparison of Sender-Initiated and Receiver-Initiated Reliable Multicast Protocols. SIGMETRICS'94, pp. 221-230.

[25] B. M. Roehner. Patterns of Speculation: A Study in Observational Econophysics. Cambridge University Press (ISBN 0521802636). May 2002.

[26] L. Rodrigues, K. Guo, P. Verissimo, and K. Birman. A Dynamic Light-Weight Group Service. Journal of Parallel and Distributed Computing 60:12. 2000.

[27] Y. Tock, N. Naaman, A. Harpaz, and G. Gershinsky. Hierarchical Clustering of Message Flows in a Multicast Data Dissemination System. IASTED PDCS 2005.

[28] Y. Vifgusson, K. Ostrowski, K. Birman, and D. Dolev. Tiling a Distributed System for Efficient Multicast. Unpublished manuscript.

Extensible Architecture for High-Performance, Scalable, Reliable Publish-Subscribe Eventing and Notification

Krzysztof Ostrowski, Cornell University, USA

Ken Birman, Cornell University, USA

Danny Dolev, The Hebrew University of Jerusalem, Israel

ABSTRACT

Existing Web service notification and eventing standards are useful in many applications, but they have serious limitations that make them ill-suited for large-scale deployments, or as a middleware or a component-integration technology in today's data centers. For example, it is not possible to use IP multicast, or for recipients to forward messages to others, scalable notification trees must be setup manually, and no end-to-end security, reliability, or QoS guarantees can be provided. We propose an architecture that is free of such limitations and that may serve as a basis for extending or complementing the existing standards. The approach emerges from our work on QuickSilver, a new, extremely modular and extensible platform for high-performance, scalable, reliable eventing.

Keywords: architecture; eventing; extensible; multicast; notification; publish-subscribe; reliable; scalable

INTRODUCTION

Motivation

Notification is a valuable, widely used primitive for designing distributed systems. The growing popularity of RSS feeds and similar technologies shows that this is also true at the Internet scales. The WS-Notification (Graham et al., 2004) and WS-Eventing (Box et al., 2004) standards have been offered as a basis for interoperation of heterogeneous systems deployed across the Internet. Unlike RSS, they are subscription-based and, hence, free of the scalability problems of polling, and they support proxy nodes that could be used to build scalable notification trees. Nonetheless, they embody restrictions that make them unsuitable as a middleware technology in large-scale systems:

Copyright © 2007, IGI Global. Copying or distributing in print or electronic forms without written permission of IGI Global is prohibited.

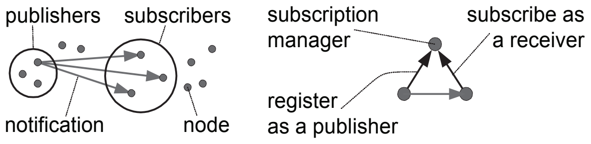
- **No forwarding among recipients:** Many content distribution schemes build overlays within which content recipients participate in message delivery. In current Web services notification standards, however, recipients are *passive* (limited to data reception). For example, given the tremendous success of BitTorrent for multicast file transfer, one could imagine a future event notification system that uses a BitTorrent-like protocol for data transfer. But BitTorrent depends on direct peer-to-peer interactions by recipients.
- **Not self-organizing:** While both standards permit the construction of notification trees, such trees must be manually configured and require the use of dedicated infrastructure nodes ("proxies"). Automated setup of dissemination trees by means of a protocol running directly between the recipients is often preferable, but the standards preclude this possibility.
- **Weak reliability:** Reliability in the existing schemes is limited to per-link guarantees resulting from the use of TCP. In many applications, end-to-end guarantees are required, and often of strong flavor, for example, to support virtually synchronous, transactional, or state-machine replication. Because receivers are assumed passive and cannot cache, forward messages, or participate in multiparty protocols, even weak guarantees of these sorts cannot be provided.
- **Difficult to manage:** It is hard to create and maintain an Internet-scale dissemination structure that would permit any node to serve as a publisher or as a subscriber, for this requires many parties to maintain a common infrastructure and agree on standards, topology, and other factors. Any such large-scale infrastructure should respect local autonomy, whereby the owner of a portion of a network can set up policies for local routing, availability of IP multicast, and so forth.
- **Inability to use external multicast frameworks:** The standards leave it entirely to the recipients to prepare their communication endpoints for message delivery. This makes it impossible for a group of recipients to dynamically agree upon a shared IP multicast address, or to construct an overlay multicast within a segment of the network. Yet such techniques are central to achieving high performance and scalability, and can also be used to provide QoS guarantees or to leverage emergent technologies.

In this article, we propose a principled approach to Web service notification in large-scale systems, free of the limitations listed above, which is modular and highly extensible. The design presented here is a basis for Quicksilver (Ostrowski & Birman, 2006c; Ostrowski, Birman & Dolev, 2006), a novel, reliable, and extremely scalable platform for publish-subscribe eventing and notification, under development at Cornell. While this architecture is inspired by our prior work on QuickSilver, it is designed to be generic, and it is compatible, in general, with a wide range of existing protocols.

Model

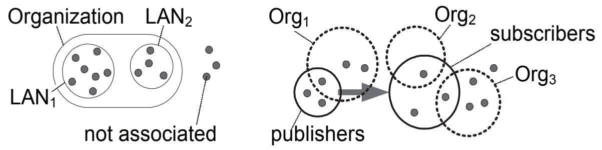
We employ the usual terminology, where events are associated with *topics*, produced

Figure 1. Publishers and subscribers register for a topic with the subscription manager



Copyright © 2007, IGI Global. Copying or distributing in print or electronic forms without written permission of IGI Global is prohibited.

Figure 2. Nodes can be scattered across administrative domains hierarchically divided into subdomains



by *publishers,* and delivered to *subscribers*. We use the term "*group X*" to refer to the set of nodes subscribed to topic "*X*." More than one node may publish to a given topic. The prospective publishers and subscribers register with a *subscription manager*. This entity can be independent of the publishers (Figure 1). The manager may be replicated to tolerate failures, or hierarchical, to scale. A single manager may track the publishers and subscribers for many topics, and many independent managers may coexist. Nodes may reside in many *administrative domains* (LANs, data centers, etc.). Nodes in the same domain may be *jointly managed*. It is often convenient to define policies, such as for message forwarding or resource allocation, in a way that respects domain boundaries, for example, for administrative reasons, or because communication within a domain is cheaper than across domains, as it is often related to network topology. Publishers and subscribers might be scattered across organizations. These need to cooperate in message dissemination, which often presents a logistic challenge (Figure 2).

Example

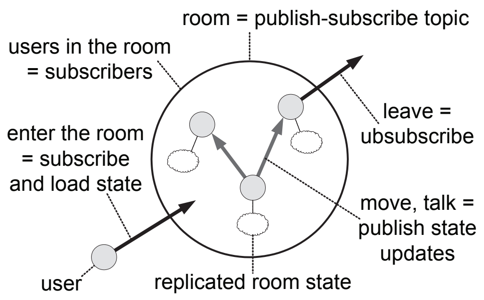
To facilitate discussion, the article uses a running example. Obviously, the architecture is not limited to any particular application; the architecture is intended to be flexible enough to serve as a general, multipurpose middleware component-integration technology, and to be used in settings such as large data centers, trading systems, military infrastructure, and so forth. However, the example helps us illustrate the architecture with specific scenarios that highlight the role of specific features.

The example involves a possible vision for the future of the Internet. Even today, the Web is moving towards very dynamic and interactive content. Massively multiplayer online gaming and virtual realities, such as the World of Warcraft and Second Life, are becoming increasingly popular. However, current techniques are insufficient for massive-scale deployments. As of this writing, the latter platform is reported to host online 17,000 of a total of 2 million of its users, on a server farm of nearly 2,600 dual-core Opteron machines. As the number of online users simultaneously browsing through such virtual realities will grow to tens and hundreds of millions, as users start to expect a smoother and more realistic experience, and they start transmitting high-resolution audio, video, and animation streams, it will become very difficult to host entire virtual worlds in a centralized manner. Centralized systems are invariably costly and suffer from bottlenecks and high latencies. We believe that the most cost-effective (and perhaps ultimately the only feasible) way to implement such scenarios is for the virtual worlds to be decentralized, and for the users to interact directly, without any expensive servers in the middle.

Apart from the scalability aspect mentioned above, we also believe that the Internet community is unlikely to accept a situation where all content is controlled by a handful of providers. Instead, we envision that just as today users create Web pages, they would eventually want to be able to create their own virtual rooms or landscapes, where participants could interact with one-another much as they do in online multiplayer games. A successful technology base of this sort could eventually transform

Copyright © 2007, IGI Global. Copying or distributing in print or electronic forms without written permission of IGI Global is prohibited.

Figure 3. A virtual room in a virtual world, modeled as a "publish-subscribe" topic


room = publish-subscribe topic
users in the room = subscribers
leave = ubsubscribe
enter the room = subscribe and load state
move, talk = publish state updates
user
replicated room state

the Web into a multi-verse, a federation of millions of interconnected virtual places or entire worlds, created and hosted by the Internet community collaboratively, with a mixture of infrastructure services, services hosted on data centers, and services introduced and hosted by individual users.

A high-performance, scalable, and reliable variant of the *publish-subscribe* paradigm could be the enabling technology for such applications. To see this, think of each location in a virtual world as a separate publish-subscribe topic, and of the users that currently reside inside the location as subscribers (Figure 3). The state of the room, for example, its interior, user positions, actions in progress, all objects inside the room, and so forth, is replicated among all the subscribers. The state is loaded by the users upon entering the room, and can be updated in a consistent manner by multicasting any updates (such as users speaking, moving, handing objects to each other, etc.) reliably to the set of all subscribers. The state can be retained while no visitors are in the room by a room *guardian*, a special entity that can either stay in the room at all times, or slip in only if the last "regular" user leaves, depending on how the room is setup. Unlike in the centralized approaches, here users interact directly with each other, with no server in between. Although this solution does require infrastructure components, for example, to track subscriptions, inform users of each other's existence, control the "guard," and so forth, none of the required infrastructure sits on the "critical path" and acts as a proxy or intermediary. If one user speaks or projects a video clip to others, the data can be transmitted directly to participants, without the involvement of any such infrastructure. Infrastructure that controls a virtual location could thus be hosted even on a home machine of the user who created it. The network of users' home machines, hosting their virtual rooms connected by virtual corridors, can thus form a background *backbone* structure that controls the virtual reality in a way similar to how DNS serves as a backbone of the Internet.

To realize the vision outlined above, we need a publish-subscribe platform that can provide very high performance, scalability in multiple dimensions, such as the size of the system, the number of publish-subscribe topics, or data rates, and so forth, end-to-end reliability guarantees, and a way to integrate with modern development platforms. In work reported elsewhere, we have created a system with these attributes. Initial results from experiments on the system (Quicksilver Scalable Multicast, or QSM) suggest that these goals can be achieved (Ostrowski & Birman, 2006b, 2006c). However, for truly massive adoption, we need publish-subscribe interoperability standards that allow a large number of independent users, residing in different administrative domains scattered across the Internet, to collaboratively form a single infrastructure for reliable publish-subscribe notification. This was our original reason to explore the architectural proposal that is the subject of this article.

Copyright © 2007, IGI Global. Copying or distributing in print or electronic forms without written permission of IGI Global is prohibited.

Let's now revisit the problems listed in the section entitled "Motivation" in the context of our example. Consider a user sitting in a cafeteria with his laptop that enters a virtual room and starts interacting with friends in our three-dimensional virtual reality. The user would act as one of several publishers and subscribers. Because such interactions would be ad-hoc, the technique that the users would use to disseminate data between them should be *self-organizing*, that is, it should not rely on proxies, or other dedicated infrastructure.

A simple way to meet this requirement could be for each publisher to send updates directly to all subscribers over TCP, but this is not acceptable for several reasons. First, to ensure that all users see a consistent history of events, we'd need *reliability guarantees*, such as a global ordering of all updates published by different users (so that all of them see events occurring in the same order, or that two users don't pick up the same object), atomicity (so that if one user can see something happening, then so do all the others) and so forth. While solutions to such problems are well known from the literature, they need more than a plain point-to-point TCP-based dissemination scheme. Thus, the users would need to run a suite of special reliability protocols.

Second, the wireless link of the user in the cafeteria, or his laptop, may not be fast enough to simultaneously send updates to ten other people who may be in the virtual room. If other users are connected, for example, directly to the campus network, over a wire, it may be desirable to arrange it so that the wireless user publishes updates to a user on a campus LAN, which is then responsible for *forwarding* it to the others. If the campus LAN is configured to enable IP multicast, it would be desirable to be able to exploit such *external mechanisms*. The users should thus be able to quickly form small overlays that can efficiently utilize whatever resources are available.

Finally, note that users will often reside in different administrative domains, for example, in a cafeteria wireless network, in different campus networks, on a cable LAN, and so forth. The administrators of these domains may need to impose acceptable use policies that specify how dissemination should be performed internally in the domain they own. For example, one domain might disallow IP multicast, while another might permit IP multicast provided that various rules are respected. Policies could govern sharing of connections, what data rates are acceptable, what multicast protocol to use, and so forth.

Different domains will often have distinct administrative policies. And yet, users residing in those domains would expect a smooth operation, as if the entire Internet formed a single, fully-connected administrative domain. Such *management* issues are a logistic headache, and require better *interoperability* standards. An update published by the user in the cafeteria should be disseminated in every campus network, or among wireless users, locally according to the local policies setup by the domain administrators, but all these domains need to cooperate with each other, ideally without reliance on costly *proxies*. Existing eventing standards have overlooked such issues, but they form the core of our proposal.

Design Principles

The limitations of the existing architectures, listed in the section on "Motivation," and our experience in designing scalable multicast systems, led us to the following design principles:

- **Programmable nodes:** Senders and recipients should not be limited to sending or receiving. They should be able to perform certain basic operations on data streams, such as forwarding or annotating data with information to be used by other peers, in support of local *forwarding policies*. The latter must be expressive enough to support protocols used in today's content delivery networks, such as overlay trees, rings, mesh structures, gossip, link multiplexing, or delivery along redundant paths.
- **External control:** Forwarding policies used by subscribers must be selected and

Copyright © 2007, IGI Global. Copying or distributing in print or electronic forms without written permission of IGI Global is prohibited.

updated in a consistent manner. A node cannot predict a-priori what policy to use, or which other nodes to peer with; it must thus permit an external trusted entity or an agreement protocol to control it: determine the protocol it follows, install rules for message forwarding or filtering, and so forth.

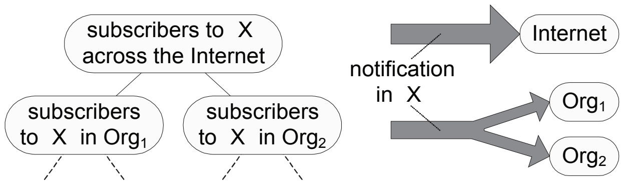
- **Channel negotiation:** The creation of communication channels should permit a handshake. A recipient might be requested to, for example, join an IP multicast address or subscribe to an external system. The recipient could also make configuration decisions on the basis of the information about the sender. For example, a LAN domain asked to create a communication endpoint for receiving could select a well-provisioned node as its entry point to handle the anticipated load.
- **Managed channels:** Communication channels should be modeled as *contracts* in which receivers have a degree of control over the way the senders are transmitting. In self-organizing systems, reconfiguration triggered by churn is common and communication channels often need to be reopened or updated to adapt to the changing topology, traffic patterns, or capacities. For example, a channel that previously requested that a given source transmits messages to one node may notify the source that messages should now be transmitted to some two other nodes.
- **Hierarchical structure:** The principles listed above should apply to not just individual nodes, but also to entire administrative domains, such as LANs, data centers, or corporate networks. This allows the definition and enforcement of Internet-scale forwarding policies, facilitating cooperation among organizations in maintaining the global infrastructure. The way messages are delivered to subscribers across the Internet thus reflects policies defined at various levels (for example, policies "internal" to data centers, and a global policy "across" all data centers).
- **Isolation and local autonomy:** A degree of a local autonomy of the individual administrative domains, such as how messages are forwarded internally, which nodes are used to receive incoming traffic or relay data to other domains, and so forth, should be preserved. In essence, the internal structure of an administrative domain should be hidden from other domains it is peering with and from the higher layers. Likewise, the details of the subcomponents of a domain should be as opaque as possible.
- **Reusability:** It should be possible to specify a policy for message forwarding or loss recovery in a standard way and post it into an online library of such policies as a contribution to the community. Administrators willing to deploy a given policy within their administrative domain should be able to do so in a simple way, for example, by drag-and-drop, within a suitable GUI.
- **Separation of concerns:** Motivated by the end-to-end principle, we separate implementation of loss recovery and other reliability properties from the unreliable dissemination of messages, as well as from message ordering, security, and subscription management. Accordingly, our design includes *reliability*, *dissemination*, *ordering*, *security*, and *management* frameworks, five independent, yet complementary structures. This decoupling gives our system an elegant structure and a degree of modularity and flexibility unseen in existing architectures.

Hierarchical View of the Network

A group X of subscribers for a given topic across the entire Internet can be divided into subsets Y_1, Y_2, ..., Y_N of subscribers in N top-level administrative domains (Figure 4). This may continue recursively, leading to a hierarchical perspective on the group X. Hierarchies of this sort have been previously exploited in scalable multicast protocols, for example, in RMTP (Paul, Sabnani, Lin, & Bhattacharyya, 1997), or in the context of content-based filtering

Copyright © 2007, IGI Global. Copying or distributing in print or electronic forms without written permission of IGI Global is prohibited.

Figure 4. A hierarchical decomposition of the set of subscribers along the domain boundaries



(Banavar, Chandra, Mukherjee, Nagarajarao, Strom, & Sturman, 1999). The underlying principle, implicit in many scalable protocols, is to exploit locality. Following this principle, sets of nodes, clustered based on proximity or interest, cooperate semi-autonomously in message routing and forwarding, loss recovery, managing membership and subscriptions, failure detection, and so forth. Each such set is treated as a single cell within a larger infrastructure. A protocol running at a global level connects all cells into a single structure. Scalability arises as in the *divide-and-conquer* principle. Additionally, the cells can locally share workload and amortize dissemination or control overheads, for example, buffer messages from different sources and locally disseminate such combined bundles, and so forth.

In the architecture described here we go one step further. Following our principle of *isolation* and *local autonomy*, each administrative domain should manage the registration of its own publishers and subscribers internally, and it should be able to decide how to distribute messages among them or how to perform loss recovery according to its local policy. Unlike in most hierarchical systems, where hierarchy and protocol are inseparable, and hence the "subprotocols" used at all levels of the hierarchy are identical, in our architecture we decouple the creation of the hierarchy from the specific "subprotocols" used at different levels of the hierarchy and we allow the "subprotocols" to differ. Thus for example, our architecture permits the creation of a single, global dissemination scheme for a topic that uses different mechanisms to distribute data in different organizations or data centers. Likewise, it permits the creation of a single Internet-scale loss recovery scheme that employs different recovery policies within different administrative domains. Previously, this has only been possible with proxies, which can be costly, and which introduce latency and bottleneck. In this article, we propose a way to do this efficiently, and in a very generic, flexible manner. This novel hierarchical protocol "composition" approach, motivated by the principles of locality and local autonomy, is central to our architecture.

ARCHITECTURE

The Hierarchy of Scopes

Our architecture is organized around the following key concepts: management scope, forwarding policy, channel, filter, session, recovery protocol, recovery domain, and agent.

A *management scope* (or simply *scope*) represents a set of jointly managed nodes. It may include a single node, span over a set of nodes residing within a certain administrative domain, or include nodes clustered based on other criteria, such as common interest. In the extreme, a scope may span across the entire Internet. We do not assume a 1-to-1 correspondence between administrative domains and the scopes defined based on such domains, but that will often be the case. A LAN scope (or just a LAN) will refer to a scope spanning all nodes residing within a LAN. The reader might find it easier to understand our design with such examples in mind.

A scope is not just *any* group of nodes; the assumption that they are *jointly managed* is essential. The existence of a scope is dependent upon the existence of an infrastructure that

Copyright © 2007, IGI Global. Copying or distributing in print or electronic forms without written permission of IGI Global is prohibited.

maintains its membership and administers it. For a scope that corresponds to a LAN, this could be a server managing all local nodes. In a domain that spans several data centers in an organization, it could be a management infrastructure, with a server in the company headquarters indirectly managing the network via subordinate servers residing in every data center. No such global infrastructure or administrative authority exists for the Internet, but organizations could provide servers to control the Internet scope in support of their own publishers or to manage the distribution of messages in topics of importance to them. Many global scopes, independently managed, could thus co-exist.

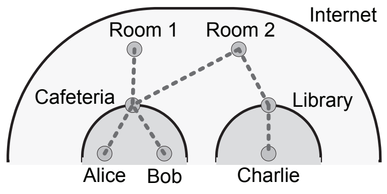
Like administrative domains, scopes form a hierarchy, defined by the relation of *membership*: one scope may declare itself to be a *member* (or a "subscope") of another. If X declares itself to be a member of Y, it means X is either physically or logically a part (subset) of Y. Typically, a scope defined for a subdomain X of an administrative domain Y will be a member of the scope defined for Y. For example, a node may be a member of a LAN. The LAN may be a member of a data center, which in turn may be a member of a corporate network. A node may also be a member of a scope of an overlay network. For a data center, two scopes may be defined, for example, *monitoring* and *control* scopes, both covering the entire data center, with some LANs being a part of one scope, or the other, or both. The corporate scope may be a member of several Internet-wide scopes, and so forth.

The generality in these definitions allows us to model various special cases, such as clustering of nodes based on interest or other factors. Such clusters, formed, for example, by a server managing a LAN and based on node subscription patterns, could also be treated as (virtual) scopes, all managed by the same server. Nodes would thus be members of clusters, and clusters (not nodes) would be members of the LAN. As we shall explain below, every such cluster, as a separate scope, could be locally and independently managed. For example, suppose that we are building an event notification system that needs to disseminate events reliably, and is implemented by an unreliable multicast mechanism coupled to a reliability layer that recovers lost packets. In the proposed architecture, different clusters could run different multicast or loss recovery protocols; this technique is used in the QSM platform (Ostrowski & Birman, 2006b, 2006c). Thus, if one cluster happens to reside within a LAN that permits the use of IP multicast, it could use that technology, while a different cluster on a network that prohibits IP multicast, or consisting of a large number of nodes across the Internet, could instead form an end-to-end multicast overlay.

The scope hierarchy need not necessarily be a tree (Figure 5). There may be many global scopes, or many superscopes for any given scope. However, a scope decomposes into a tree of subscopes, down to the level of nodes. The *span of a scope X* is the set of all nodes at the bottom of the hierarchy of scopes rooted at X. For a given topic X, there always exists a single global scope responsible for it, that is, such that all subscribers to X reside in the span of X. Publishing a message to a topic is thus always equivalent to delivering it to all subscribers in the span of some global scope, which may be further recursively decomposed into the sets of subscribers in the spans of its subscopes.

Suppose that Alice and Bob are sitting with their laptops in a cafeteria, while Charlie is in a library. Both the cafeteria's wireless network and the local network in the library are separately managed administrative domains, and they define their own *management scopes*. Alice's and Bob's laptops are also *scopes*, both of which

Figure 5. An example hierarchy of management scopes in a game



Copyright © 2007, IGI Global. Copying or distributing in print or electronic forms without written permission of IGI Global is prohibited.

become members of the cafeteria's scope. Now suppose that Alice opens her virtual realities browser and enters a virtual place "Room1" in the virtual reality, while Bob and Charlie enter "Room2." Each of the virtual rooms defines a global, Internet-wide scope that could be thought of as "the scope of all users in this room, wherever they are." If the networking support for Alice and Bob is still provided by the cafeteria and library wireless networks respectively, when the students enter the rooms, the cafeteria's and library's scopes become *members* of these global scopes (Figure 5).

The Anatomy of a Scope

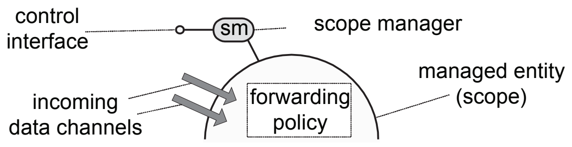
The infrastructure component administering a scope is referred to as a *scope manager* (SM). A single SM may control multiple scopes. It may be hosted on a single node, or distributed over a set of nodes, and it may reside outside of the scope it controls. It exposes a *control interface*, a Web service hosted at a well-known address, to dispatch control requests (e.g., "subscribe") directed to the scopes it controls (Figure 6).

A scope maintains communication *channels* for use by other scopes. A *channel* is a mechanism through which a message can be delivered to all those nodes in the span of this scope that subscribed to any of a certain set of topics. In a scope spanning a single node, a channel may be just an address/protocol pair; creating it would mean arranging for a local process to open a socket. In a distributed scope, a channel could be an IP multicast address; creating it would require all local nodes to listen at this address. It could also be a list of addresses, if messages are to be delivered to all of them, or if addresses are to be used in a random or a round-robin fashion. In an overlay network, for example, a channel could lead to a small set of nodes that forward messages across the entire overlay. In general, a scope that spans a set of nodes can thus be governed by a *forwarding policy* that determines how messages originating within the scope, or arriving through some communication channel, are disseminated internally, within the local scope.

Continuing our example, the cafeteria and the library would host the managers of their scopes on dedicated servers, and each of the student laptops would run a local service that serves as a scope manager for the local machine. The library's SM may be on a campus network with IP multicast enabled. When the cafeteria's SM requests a channel from the library's SM, the latter might, for example, dedicate some machine as the entry point for all messages coming from the cafeteria, instruct it to retransmit these messages to the IP multicast address, and instruct all other laptops to join the IP multicast address. Similarly, the cafeteria's SM might setup a local forwarding tree. In most settings, a scope manager would live on a dedicated server. It is also conceivable to offload, in certain scenarios, parts of the SM's functionality to nodes currently in the scope (e.g., to Alice's and Bob's laptops), but a single "point of contact" for the scope would still need to exist.

The control interfaces "exposed" by scopes (interfaces exposed by their scope managers) are "accessed" by other scopes (i.e., by their SMs). When interacting, scopes can play one of a few standard roles, corresponding to the three principal interaction patterns: *member-*

Figure 6. A scope is controlled by a scope manager, which exposes a standardized control interface, and may create a number of incoming data channels, to serve as "entry points" for the scope



Copyright © 2007, IGI Global. Copying or distributing in print or electronic forms without written permission of IGI Global is prohibited.

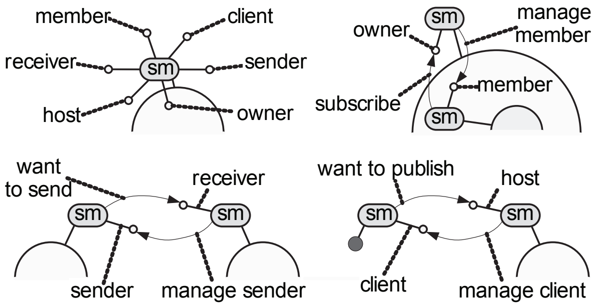
owner, *sender-receiver*, and *client-host* (see Figure 7). A *member* registers with an *owner* to establish a relation of *membership*, that is, to "declare" itself as a subscope of the owner. After such a relationship has been established, the member may then register with the owner as a publisher or subscriber in one or more topics. Depending on the topics for which the member has registered and its role in these topics, it may be requested by the owner to perform relevant actions, for example, by forwarding messages or participating in the construction of an overlay structure. At the same time, the owner is responsible for tracking the health of its members, and in particular for detecting failures and performing local reconfiguration.

The *client-host* relationship is similar to *member-owner*, in that a *client* can register with a *host* as a publisher or as a subscriber. However, unlike the *member-owner* relationship, which involves a mutual commitment, the *client-host* relationship is more casual, in the sense that the host cannot rely on the client to perform any tasks for the system, or even to notify the host when it leaves, and similarly, the client cannot rely on the host to provide it with the same quality of service and reliability properties as the regular members. Thus for example, while a member can form a part of a forwarding tree, a client will not; the latter will also typically get weaker reliability guarantees, because the protocols run by the scope members will not be prevented from making progress when transient clients are unreachable. The same applies to publishers. A long-term publisher that forms a part of a corporate infrastructure will register as a member, but handheld devices roaming on a wireless network will usually register as clients.

The *sender-receiver* relationship is similar to a publisher registering as a client or member in that the *sender* registers with the *receiver* to send data. However, whereas a *member* registers with its *owner* to publish to the set of all subscribers in a topic, including nodes inside as well as nodes outside of the *owner*, a *sender* will register with a *receiver* to establish a communication channel between the two to disseminate messages only within the scope of the receiver. When a publisher registers as a member with an owner scope that is not the global scope for the given topic, the owner may itself be forced to subscribe with its superscope. The sender-receiver relationship is horizontal; no cascading subscriptions take place. On the other hand, while a member scope never exchanges data with the owner (instead, it is "told" by the owner to form part of a dissemination structure that the owner controls), the sender-receiver relationship serves exactly this purpose; the two parties involved in the latter will negotiate protocols, addresses, transmission rates, and so forth.

The reader should recognize in our construction the design principles we articulated earlier. Scopes, whether individual nodes, LANs

Figure 7. When interacting, scopes follow one of a few standardized relationship patterns, in which they can play one of a few standard roles



Copyright © 2007, IGI Global. Copying or distributing in print or electronic forms without written permission of IGI Global is prohibited.

or overlays, are *externally controlled* using the control interfaces exposed by SMs, may be *programmed* with policies that govern the way messages are distributed internally, forwarded to other scopes, and so forth, and transmit messages via *managed* communication channels, established through a dialogue between a pair of SMs, and dynamically reconfigured.

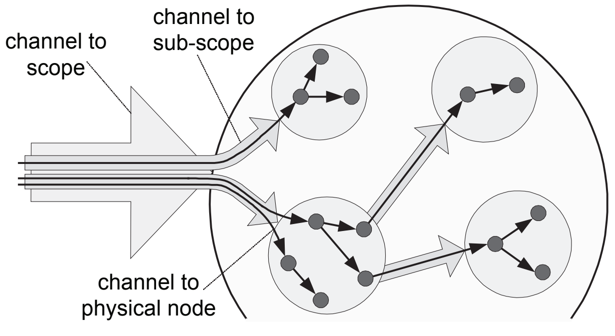
Hierarchical Composition of Policies

Following our design principles, we propose to solve the issue of a large-scale global cooperation in message delivery between independently managed administrative domains by introducing a hierarchical structure in which forwarding policies defined at various levels are merged into a single dissemination scheme. Each scope is configured with a policy dictating, on a per-topic (and perhaps a per-sender) basis, how messages are forwarded among its members. For example, a policy governing a global scope might determine how messages in topic T, originating in a corporate network X, are forwarded between the various organizations. A policy of a scope of the given organization's network might determine how to forward messages among its data centers, and so forth. A policy defined for a particular scope X is always defined at the granularity of X's members (not individual nodes). The way a given subscope Y of X delivers messages internally is a decision made autonomously by Y. Similarly, X's policy may specify that Y should forward messages to Z, but it is up to Y's policy to determine how to perform this task.

Accordingly, a global policy may request that organization X forward messages in topic T to organizations Y and Z. A policy governing X may then determine that to distribute messages in X, they must be sent to LAN_1, which will forward them to LAN_2. The same policy might also specify which LANs within X should forward to Y and Z, and finally, the policies of these LANs will delegate these forwarding tasks to individual nodes they own. When all the policies defined at all the involved scopes are combined together, they yield a global *forwarding structure* that completely determines the way messages are disseminated (Figure 8). In the examples given, the forwarding policies are simply graphs of connections: each message is always forwarded along every channel. In general, however, a channel could be constrained with a *filter* that decides, on a per-message basis, whether to forward or not, and may optionally tag the message with custom attributes (more details are in the section "Communication Channels"). This allows us to express many popular techniques, for example, using redundant paths, multiplexing between dissemination trees, and so forth.

Every scope manager maintains a mapping from topics to their forwarding policies. A forwarding policy is defined as an object that

Figure 8. Channels created in support of forwarding policies defined at different levels



Copyright © 2007, IGI Global. Copying or distributing in print or electronic forms without written permission of IGI Global is prohibited.

lives in an *abstract context*, and that exposes a fixed set of *events* to which members must be prepared to react, and the *operations* and *attributes*. A scope manager might be thought of as a *container* for objects representing *policies*. The interfaces that the container and the policies expose to each other and interact with are standardized, and may include, for example, an event informing the policy that a new member has been added to the set of members locally subscribed to the topic, or an operation exposed by the container that allows the policy to request a member to establish a channel to another member and instantiate a given type of filter (Figure 9). A scope manger thus provides a *runtime environment* in which policies can be hosted. This allows policies to be defined in a standard way, independent not only of the platform, but also of the type of the administrative domain. For example, one can imagine a policy that uses some sort of a novel mesh-like forwarding structure with a sophisticated adaptive flow and rate control algorithm. Our architecture would allow that policy to be deployed, without any modifications, in the context of a LAN, data center, corporate network, or a global scope. In effect, we've separated the policy from the details of how it should be implemented in a particular domain. Policies may be implemented in any language, expose and consume Web service APIs, stored online in *protocol libraries*, downloaded as needed, and executed in a secure manner.

Graphs of connections for different topics, generated by their respective policies, are superimposed (Figure 10). The SM of the scope maintains an aggregate structure of channels and filters, and issues requests to the SMs of its members to create channels, instantiate filters, and so forth. If multiple policies request channels between the same pair of nodes, the SM will not create multiple channels, but rather a single channel, for multiple topics. To avoid the situation where every member talks to every other member, the SM may use a single forwarding policy for multiple topics, to ensure that channels created for different topics overlap.

Communication Channels

Consider a node X, which is a member of a scope Y that, based on a forwarding policy at Y, has been requested to create a communication channel to scope Z to forward messages in topic T. Following the protocol, X asks the SM of Z for the specification of the channel to Z that should be used for messages in topic T. The SM of Z might respond with an address/protocol pair that X should use to send over this channel. Alternatively, a forwarding policy defined for T at scope Z may dictate that, in order to send to Z in topic T, scope X should establish channels to members A and B

Figure 9. A forwarding policy as a code snippet

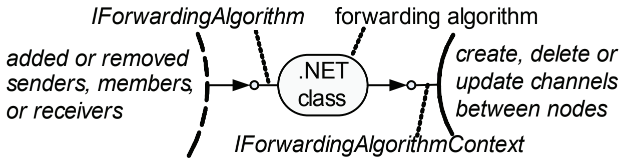
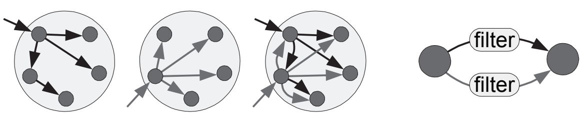

IForwardingAlgorithm
forwarding algorithm
added or removed senders, members, or receivers
.NET class
create, delete or update channels between nodes
IForwardingAlgorithmContext

Figure 10. Forwarding graphs for different topics are superimposed. Two members may be linked by multiple channels, each with a different filter



Copyright © 2007, IGI Global. Copying or distributing in print or electronic forms without written permission of IGI Global is prohibited.

of Z, constrained with filters α and β. After X learns this from the SM of Z, it contacts SMs of A and B for further details. Notice that the channel to Z decomposes into subchannels to A and B through a policy at a target scope Z. This procedure will continue hierarchically, until the point when X is left with a tree with filters in the internal nodes and address and protocol pairs at the leaves (Figure 11). Now, to send a message along a channel constructed in this way, X executes filters, starting from the root, to determine recursively which subchannels to use, proceeding until it is left with just a list of address/protocol pairs, and then transmits the message. Filters will usually be simple, such as modulo-**n**; hence X can perform this procedure very efficiently. Indeed, with network cards becoming increasingly powerful and easily programmable (Weinsberg, Dolev, Anker, & Wyckoff, 2006), such functionality might even be offloaded to hardware.

Accordingly, to support the hierarchical composition of policies described in the preceding section, we define a channel as one of the following: an address/protocol pair, a reference to an external multicast mechanism, or a set of subchannels accompanied by filters. In the latter case, the filters jointly implement a multiplexing scheme that determines which subchannels to use for sending, on a per-message basis (Figure 12, Figure 13).

Consider now the situation where scope X, spanning a set of nodes, has been requested to create a channel to scope Y. Through a dialogue with Y and its subscopes, X can obtain a detailed channel definition, but unlike in prior examples, X now spans a *set of nodes*, and as such, it cannot *execute* filters or *send* messages. To address this issue, we now propose two simple, generic techniques: *delegation* and *replication* (Figure 14). Both of them rely on the fact that if X receives messages in a topic T, then some of its members, Z, must also receive

Figure 11. A channel split into subchannels and a possible filter tree corresponding to it

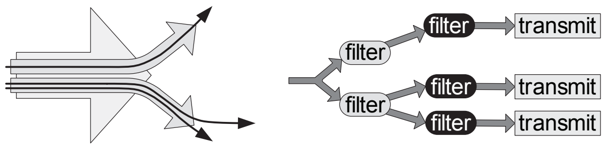


Figure 12. A channel may be an address/protocol pair (left), or it may consist of subchannels, with an algorithm deciding what goes where (right)

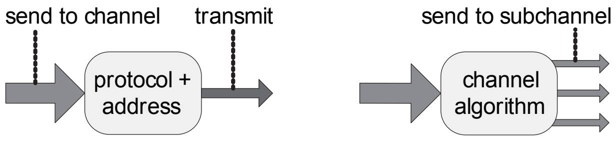
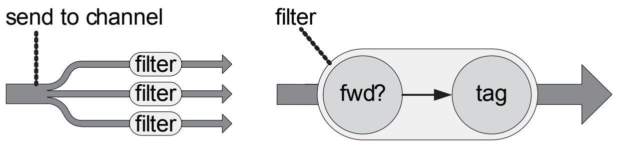
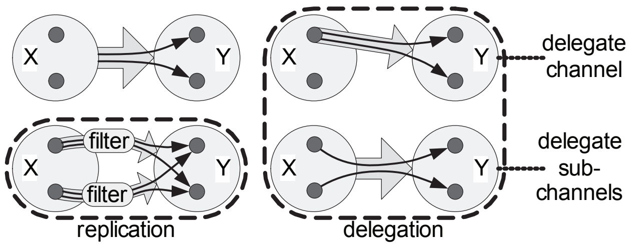


Figure 13. A distributed scope may delegate a channel or some of its subchannels to its members, or it may replicate the channel among members with filters that jointly implement a round-robin policy, and so forth



Copyright © 2007, IGI Global. Copying or distributing in print or electronic forms without written permission of IGI Global is prohibited.

Figure 14. Channel algorithms are realized as sets of filters, one per subchannel, deciding whether to forward, and optionally adding custom tags.



them (for otherwise X would not declare itself as a subscriber and it would not be made part of the forwarding structure for topic T by X's superscope). In case of *delegation*, X requests such a subscope Z to create the channel on behalf of X, essentially "delegating" the entire channel to its member. The problem can be recursively delegated, down to the level where a single physical node is requested to create the channel, and to forward messages it receives along the channel. A more sophisticated use of delegation would be for X to delegate subchannels. In such case, X would first contact Y to obtain the list of subchannels and the corresponding filters, and for each of these subchannels, delegate it to its subscopes. In any case, X delegates the responsibility for forwarding over a channel to its subscopes.

Our approach is flexible enough to support even more sophisticated policies. An example of one such policy is a *replication* strategy, outlined here. In this scheme, scope X could request that **n** of its subscopes create the needed channel, constraining each with a modulo-**n** filter based on a message sequence number. Hence, while each of the subscopes would create the same channel on behalf of its parent scope (hence the name "replication"), subscope **k** would only forward messages with numbers **m** such that **m** mod **n** equals **k**. By doing this, X effectively implements a round-robin policy of the sort proposed in protocols such as MIT's SplitStream. Although all subscopes would create the same channel, the round-robin filtering policy would ensure that every message is forwarded only by one of them. This technique could be useful, for example, in the cases where the volume of data in a topic is so high that delegation to a single subscope is simply not feasible.

Our point is not that this particular way of decomposing functionality should be required of all protocols, but rather that for an architecture to be powerful enough and flexible enough to be used with today's cutting-edge protocols and to support state-of-the-art technologies, it needs to be flexible enough to accommodate even these kinds of "fancy" behaviors. Our proposed architecture can do so. The existing standards proposals, in contrast, are incredibly constraining. Each only accommodates a single rather narrowly conceived style of event notification system.

Constructing the Dissemination Structure

A detailed discussion of how forwarding policies can be defined and translated to filter networks is beyond the scope of this article. We describe here just one simple, yet fairly expressive scheme.

Suppose that the forwarding policies in all scopes define forwarding trees on a per-topic basis and possibly also depending on the location at which the message locally originated. By saying that a message *locally originated from* a member X of scope Y, we mean that either the message was created by X (if X is itself a node) or a member of X, or that the message was created outside of Y, but X is (or contains) the first node in all of Y to which the message

Copyright © 2007, IGI Global. Copying or distributing in print or electronic forms without written permission of IGI Global is prohibited.

Figure 15. An example hierarchy of scopes with cascading subscriptions

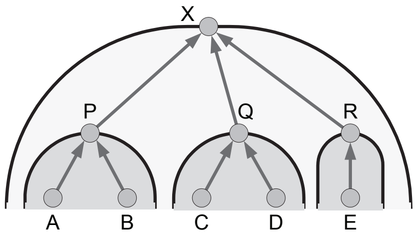
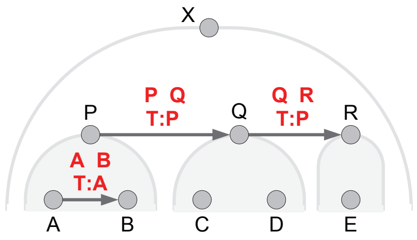


Figure 16. Channels created by the policies based on subscriptions



was forwarded. The technique described here implicitly assumes that messages do not arrive along redundant paths, that is, that the forwarding policy at each level is a tree, not a mesh. This scheme may be extended to cover the more general case with redundant paths and randomized policies, but we omit it here, for it would unnecessarily complicate our example. A comprehensive treatment of dissemination policies is beyond the scope of this article.

A scope manager thus maintains a graph, in which the scope members are linked by edges, labeled with constraints such as β:X, μ:Y, …, meaning that a message is forwarded along the edge if it was sent in topic β and locally originates at X, or if it was sent in topic μ and locally originates at Y, and so on. We shall now describe, using a simple example, how a structure of channels is established based on scope policies, and how filters are instantiated. We shall then explain how messages are routed.

Consider the structure of scopes depicted on Figure 15. Here A, B, C, D, and E are student laptops. P, Q, and R are three departments on a campus, with independent networks, hence they are separate scopes. X represents the entire campus. All students subscribe to topic T. Topic T is local to the campus, and X serves as its root. The scopes first register with each other. Then, the laptops send requests to subscribe for topics to P, Q, and R. Laptop A requests the publisher role, all others request to be subscribers. None of P, Q, or R are roots for the topic, hence they themselves subscribe for topic T with X, in a cascading manner. Now, all scopes involved create objects that represent local forwarding policies for topic T, and feed these objects with the "new member" events. The policy at P for messages in T originating at A creates a channel from A to B. Similarly, the policy at X for messages in T originating at P creates channels P to Q and Q to R (Figure 16). Each channel has a label of the form "**X-Y, T:Z**", meaning that it is a channel from X to Y, for messages in T originating at Z. Note that no channels have been created in scope Q. Until now, Q is not aware of any message sources because neither C nor D is a publisher, and because no other scope has so far requested a channel to Q, hence there is no need to forward anything. Channels are now delegated to individual nodes, as described in the section "Communication Channels." P delegates its channel to B, and Q delegates to D (Figure 17, delegated channels are in blue).

Figure 17. B and D contact Q and R to create channels. Q and R select C and E as entry points. Q now has a local message source and creates its own local channel, from C do D

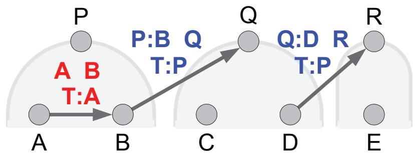
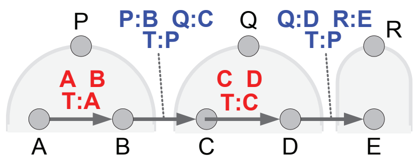


Figure 18. Channels are delegated



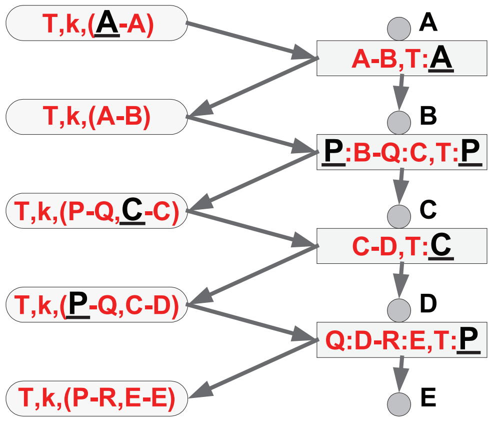
Copyright © 2007, IGI Global. Copying or distributing in print or electronic forms without written permission of IGI Global is prohibited.

Channel labels are extended to reflect the fact that they have been delegated, for example, **P-Q** becomes **P:B-Q**, which means that P has delegated its source endpoint to its member B, and so forth. At this point, B and D contact the destinations Q and R and request the channels be created. Q and R select C and E as their *entry points* for these channels. Now Q also has a local source of messages (node C, the entry point), so now it also creates an instance of a forwarding policy, which determines that messages in T, locally originating at C, are forwarded to D (Figure 18). The entire forwarding structure is complete.

A message in transit is a tuple of the form (**T**, **k**, **r**), where **T** is the topic, **k** is the identifier that may include the source name, one or more sequence numbers, and so forth, and **r** is a *routing record*. The routing record is an object of the form $(\mathbf{X}_K–\mathbf{Y}_K)$, $(\mathbf{X}_{(K-1)}–\mathbf{Y}_{(K-1)})$, …, $(\mathbf{X}_1–\mathbf{Y}_1)$, in which every pair of elements $(\mathbf{X}_i, \mathbf{Y}_i)$ represents the state of dissemination in one scope; $\mathbf{X}_i$ is the member of the scope that the message locally originated from and $\mathbf{Y}_i$ is the member of the scope that the message is moving inside of or that it is entering. Pair $(\mathbf{X}_1, \mathbf{Y}_1)$ represents individual nodes, and for each **i**, scopes $\mathbf{X}_i$ and $\mathbf{Y}_i$ are a level below $\mathbf{X}_{i+1}$ and $\mathbf{Y}_{i+1}$, respectively, and $\mathbf{Y}_i$ is always a member of $\mathbf{Y}_{i+1}$. This list of entries does not need to "extend" all the way up to the root. If entries at a certain level are missing, they will be filled up when the message jumps across scopes, as it is explained below.

When a message arrives at a node, the node iterates over all of its outgoing channels, and matches channel filters against the routing record. Message **T**, **k**, $((\mathbf{X}_K–\mathbf{Y}_K), (\mathbf{X}_{(K-1)}–\mathbf{Y}_{(K-1)}), …, (\mathbf{X}_1–\mathbf{Y}_1))$ matches channel $(\mathbf{P}_L{:}\mathbf{P}_{L-1}{:}…{:}\mathbf{P}_1 – \mathbf{Q}_L{:}\mathbf{Q}_{L-1}{:}…{:}\mathbf{Q}_1, \mathbf{T{:}R})$ when $(\mathbf{K \geq L}$ □ $\mathbf{X}_L = \mathbf{R}$ □ $\mathbf{K < L}$ □ $\mathbf{P}_L = \mathbf{R})$ holds. This condition has two parts. If $\mathbf{K \geq L}$, then in the scope in which the channel endpoints $\mathbf{P}_L$, $\mathbf{Q}_L$ and **R** are members, the message originated at $\mathbf{X}_L$ and is currently at $\mathbf{Y}_L$. According to our rules, the message should be forwarded iff $\mathbf{X}_L = \mathbf{R}$ (and $\mathbf{Y}_L = \mathbf{P}_L$, but this is always true). If $\mathbf{K < L}$, then the routing record does not carry any state for the scope at which $\mathbf{P}_L$, $\mathbf{Q}_L$ and **R** are members. This means the message must have originated in this scope (the recovery record is filled up when the message is forwarded, as explained below), hence the condition $\mathbf{P}_L = \mathbf{R}$. Note that there might be several channels originating at the node; the message is forwarded across each one it matches. Now, when the message

Figure 19. The flow of messages (rounded rectangles, left), and the channels (square rectangles, right) in the scenario of Figure 17. Elements compared against each other are shown as black, bold, and underlined



Copyright © 2007, IGI Global. Copying or distributing in print or electronic forms without written permission of IGI Global is prohibited.

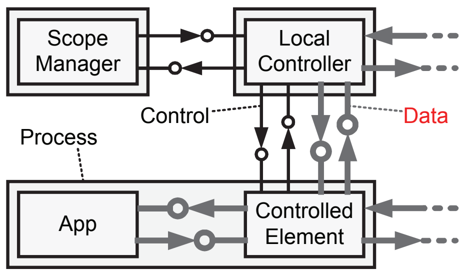
is forwarded, its routing record is modified as follows. First, the record is extended. If $\mathbf{K} < \mathbf{L}$, then for each $\mathbf{i}$ such that $\mathbf{i} > \mathbf{K}$ and $\mathbf{i} \leq \mathbf{L}$, we set $\mathbf{X}_i = \mathbf{P}_i$ and $\mathbf{Y}_i = \mathbf{P}_i$. Then, the details internal to the scope that the channel is leaving are replaced with the details internal to the scope the channel is entering, so for each $\mathbf{i}$ such that $\mathbf{i} < \mathbf{L}$, we set $\mathbf{X}_i = \mathbf{Q}_i$ and $\mathbf{Y}_i = \mathbf{Q}_i$. Finally, the current scope at the level at which the channel is defined is updated, we set $\mathbf{Y}_L = \mathbf{Q}_L$. The original value of $\mathbf{X}_L$, once set, is not changed until the message is forwarded out of the scope of which $\mathbf{X}_L$ is the member. Entries for $\mathbf{i} > \mathbf{L}$ also remain unchanged.

The flow of messages and matching in the example of Figure 18 is shown on Figure 19. When created on the publisher **A**, the message initially has a routing record **A**–**A**, since we know that it originated locally at **A** and that it is still at **A**. No entries are included at this point for routing in the scopes above **A**. There is no need for doing so since no routing has been done so far at those higher levels, so there is no state to maintain. Now the routing record is compared against channel **A-B, T:A**. The local origin **A** does match the channel's constraint **A**, so the message is forwarded along **A-B**, with a new record **A-B** to reflect the fact that it is now entering **B**. While matching this record to channel **P:B**–**Q:C** at **B**, we find that the record lacks routing information for the scope at which this channel is defined. This means that the message must have originated at the channel's source endpoint (which we know is **P** from the channel's description). We find that the channel's constraint **P** is the same as the channel's source endpoint **P**, so we again forward the message. While doing so, we extend its routing record with a new entry **P-Q** to record the fact that we made progress at this higher level. We also replace information local to P (the entry **A-B**) with new information local to **Q** (the new entry **C-C**). Forwarding at **C** and **D** works similarly to how it worked on **A** and **B**.

The Local Architecture of a Dissemination Scope

So far we focused in our description on peer-to-peer aspects of the dissemination framework, but we have not described how applications use this infrastructure, and how leaf scopes are internally organized. Of a number of possible architectures of the leaf scopes, the most general one involves the following components: *scope manager*, *local controller*, and *controlled element* (Figure 20). The *controlled element* is a very thin layer that resides in the application process. Its purpose is to serve as a hookup to the application process that allows for controlling the way the application communicates over the network. As such, it serves two principal purposes: (a) passing to the local controller requests to subscribe/unsubscribe, and (b) opening send and receive sockets in the process per request from the controller. The controlled element does not forward messages, nor does it include any other peer-to-peer functionality, and it is not a part of the management network. This simplicity allows it to be easily incorporated into legacy applications. It can be implemented in any way, for as long as it exposes a *control endpoint* (Figure 20, thin, black), a standardized Web service interface, and as long as it can

Figure 20. The architecture of a leaf scope in the most general scenario, with a local scope manager, a thin controlled element linked to the application, and with a local controller to handle peer-to-peer aspects, for example, forwarding



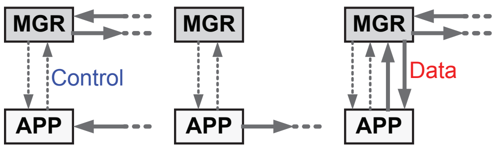
Copyright © 2007, IGI Global. Copying or distributing in print or electronic forms without written permission of IGI Global is prohibited.

create *communication endpoints* (thick, red) to receive or connect to remote endpoints to send. The local controller implements all the peer-to-peer functionality such as forwarding, and so forth, and hosts such elements as channels or filters. It may also create communication endpoints, and may send data to or receive it from the controlled element. The controller is not a part of the management network, and it does not interact with any scope mangers besides the local one. These interactions are the job of the local SM, which, on the other hand, never sends or receives any messages itself.

In general, the three components can live in separate processes, or even physically on different machines, and may communicate over local sockets or over the network. However, in the typical scenario, the scope manager and local controllers are located in a single process ("manager"). The manager runs as a system service and may control multiple applications, either local or running on subordinate devices managed by the local node, through the control elements embedded in these applications. Within this scenario, we can distinguish three basic subscenarios, or three patterns of usage (Figure 21), depending on whether applications participate in sending or receiving directly, or only through the manager.

In the first scenario, the applications only receive data, directly from the network (Figure 21, left). When the manager is requested to create an incoming channel to the scope, it may either arrange for all applications to open the same socket to receive messages directly from the network (if the applications are all hosted one the same machine), or it may make them all subscribe to the same IP multicast address (if they are running on multiple subordinate devices), or it may have them create multiple endpoints, in which case the definition of the local channel endpoint will include a set of addresses rather than just one. The manager does not sit on the critical path; hence we avoid bottleneck and latency. If the local scope is required to forward data to other scopes, the manager also creates an endpoint (opens the same socket, or extends the receive channel endpoint definition to include its own address), to receive the data that needs to be forwarded.

Figure 21. Three example architectures with the scope manager and the local controller merged into a single local system "daemon"



In our example, the library's server might act as a leaf scope, and students' laptops might act as subordinate devices that do not host their own local scope managers and do not forward messages. Applications on the laptops would have the embedded controlled elements that communicate with the local controller on the library server via a standard Web interface. When students subscribe to a local topic, such as an online lecture transmitted by another department, the server chooses an IP multicast address and has all the laptops subscribe to it. The data arriving on the local network is received by all devices without any intermediary. If the library needs to forward messages, the server also subscribes to the IP multicast address and creates all the required channels and filters.

In the second scenario, the applications act as publishers (Figure 21, center). There is no need to forward data, hence the manager does not create any send or receive channels. In order to support this scenario, the controlled element must allow transmitting data to multiple IP addresses; embed various headers provided by the local controller, and so forth. There can be different "classes" of controlled elements, depending on what functionality they provide; this scenario might be feasible only for some such classes. This scenario avoids proxies, thus it could be useful, for example, in streaming systems.

In the third scenario, the applications communicate only with the local controller, which acts as a proxy (Figure 21, right). Unlike in the first two scenarios, this introduces

Copyright © 2007, IGI Global. Copying or distributing in print or electronic forms without written permission of IGI Global is prohibited.

a bottleneck, but since the controlled elements do not need to support any external protocols or to be able to embed or strip any external headers, this scenario is always feasible. This scenario could also be used to interoperate with other eventing standards, such as WS-Eventing or WS-Notification. Here, the manager could act as the publisher or a proxy from the point of view of the application, and provide or consume messages in the format defined by all those other standards, while using our general infrastructure "under the hood." And similarly, for high-performance applications that reside on the server, an efficient implementation is possible using shared memory as a communication channel between the manager and the applications, and permitting applications to deserialize the received data directly from the manager's receive buffers, without requiring extra copy or marshaling across domains.

Sessions

We now shift focus to consider architectural implications of reliability protocols. Protocols that provide stronger reliability guarantees traditionally express them in terms of what we shall call *epochs*, corresponding to what in group communication systems are called *membership views*. In group communication systems, the lifetime of a topic (also referred to as a *group*) is divided into a sequence of *views*. Whenever the set of topic subscribers changes after a "subscribe" or an "unsubscribe" request, or a failure, a new view is created. In group communication systems, the corresponding event initiates a new epoch. Subscribers are notified of the beginnings or endings of epochs, and of the membership of the topic for each epoch. One then defines consistency in terms of which messages can be delivered to which subscribers and at what time relative to epoch boundaries. The set of subscribers during a given epoch is always fixed.

Whereas group communication views are often defined using fairly elaborate models (such as virtual synchrony, a model used in some of our past research, or consensus, the model used in Lamport's Paxos protocol suite), the architectural standard proposed here needs to be flexible enough to cover a range of reliability protocols, include many that have very weak notions of views. For example, simple protocols, such as SRM or RMTP, do not provide any guarantees of consistent membership views for topics.

In developing our architectural proposal, we found that even for protocols such as these two, in which properties are not defined in terms of epochs, epochs can still be a very useful, if not a universal, necessary concept. In a dynamic system, configuration changes, especially those resulting from crashes, usually require reconfiguration or cleanup, for example, to rebuild distributed structures, release resources, or cancel activities that are no longer necessary. Most simple protocols lack the notion of an epoch because they do not take such factors into account and do not support reconfiguration. Others do address some of these kinds of issues, but without treating them in a systematic manner. By reformulating such mechanisms in terms of epochs, we can standardize a whole family of behaviors, making it easier to talk about different protocols using common language, to compare protocols, and to design applications that can potentially run over any of a number of protocols, with the actual binding made on the basis of runtime information, policy, or other considerations.

Our design includes two epoch-like concepts: *sessions*, which are global to the entire topic, across the Internet, are shared by different frameworks (reliability, ordering, etc.), and which we discuss in this section, and *local views*, which are local to scopes, and which are discussed in the section on "Building the Hierarchy of Recovery Domains".

A *session* is a generalization of an epoch. In our system, sessions are used primarily as means of reconfiguring the topic to alter its security or reliability properties, or for other administrative changes that must be performed online, while the system is running.

The lifetime of any given topic is always divided into a sequence of sessions. However, session changes may be unrelated to member-

Copyright © 2007, IGI Global. Copying or distributing in print or electronic forms without written permission of IGI Global is prohibited.

ship changes of the topic. Since introducing a new session involves a global reconfiguration, as described further, it is infeasible for an Internet-scale system to introduce a new session, and disseminate membership views, every time a node joins or leaves. Instead, such events are handled locally, in the scopes in which they occur, and without introducing a new, global, Internet-wide epoch.

Session numbers are assigned globally, for consistency. As explained before, for a given topic, a single global scope ("root") always exists such that all subscribers to that topic reside within its span. Although, as we shall explain, dissemination, reliability, and other frameworks may use different scope hierarchies, the root scope is always the same and all the dissemination, reliability, and other aspects of it are normally managed by a single scope manager. This top-level scope manager maintains the topic's metadata; it is also responsible for assigning and updating session numbers. Note that local topics, for example, internal to an organization, may be rooted locally, for example, in the headquarters, and managed by a local SM, much in a way local newsgroups are managed locally. Accordingly, for such topics, sessions are managed by a local server (internal to the organization).

To conclude, we explain how sessions impact the behavior of publishers and subscribers. After registering, a publisher waits for the SM to notify it of the session number to use for a particular topic. A publisher is also notified of changes to the session number for topics it registered with. All published messages are tagged with the most recent session number, so that whenever a new session is started for a topic, within a short period of time no further messages will be sent in the previous session. Old sessions eventually quiesce as receivers deliver messages and the system completes flushing, cleanup, and other reliability mechanisms used by the particular protocol. Similarly, after subscribing to a topic, a node does not process messages tagged as committed to session **k** until it is explicitly notified that it should receive messages in that session. Later, after session **k+1** starts, all subscribers are notified that session **k** is entering a *flushing* phase (this term originates in *virtual synchrony* protocols, but similar mechanisms are common in many reliable protocols; a protocol lacking a flush mechanism simply ignores such notifications). Eventually, subscribers report that they have completed flushing and a global decision is made to cease any activity and *cleanup* all resources pertaining to session **k**, thus completing the transition.

Incorporating Reliability, Ordering, and Security

As mentioned earlier, we rooted our design in the principle of *separation of concerns*, and we implement tasks such as reliability, ordering, security, or scope management independently from dissemination. In the section entitled "The Local Architecture of a Dissemination Scope," we explained how the management and the dissemination infrastructures interact in our system. The remaining frameworks, reliability, security, and ordering, are decomposed in a similar manner, and they also include three base components: (a) the *controlled element* that lives in the application processes and implements only base functionality, related to sending or receiving from the applications, but none of the peer-to-peer or management aspects, (b) the *local controller* that may live outside of the application process, and where all the peer-to-peer aspects are implemented, and (c) the *scope manager* that implements the interactions with other scope managers, but that is not involved in any activities related to the data flows, such as forwarding, calculating recovery state, managing encryption keys, assigning message order, and so forth.

In general, each of the *dissemination*, *reliability*, *security,* and *ordering* frameworks has a separate hierarchy of scopes and a separate network of scope managers. For example, reliability scopes isolate and encapsulate the local aspects related to reliability, such as loss recovery, and so forth, and hide their internal details from other scopes, just like dissemination scopes manage local dissemination and

Copyright © 2007, IGI Global. Copying or distributing in print or electronic forms without written permission of IGI Global is prohibited.

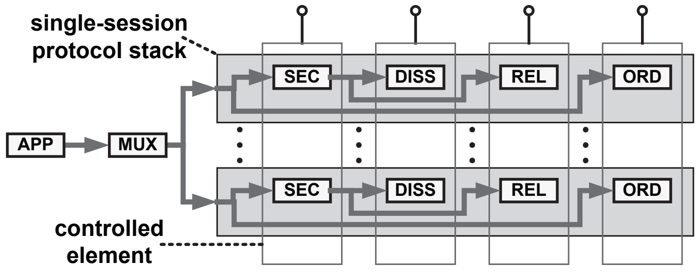
hide the local aspects of message delivery. In some cases, the different scopes would overlap. This will be normally the case, for example, with the four "flavors" of scopes (ordering, security, reliability, and dissemination) local to a node. In such cases, a single local service would act as a scope manager for all the scopes of all four flavors. The same would typically be the case for the servers that control administrative domains, such as a departmental LAN, a wireless network in a cafeteria, a data center, a corporate network, and so forth. Scopes of all flavors would again overlap, and they would be managed by a single server.

Irrespective of whether components of the four different frameworks overlap, or are physically hosted on the same machine or in the same process, the frameworks always logically converge in the application (Figure 22). The complete local architecture includes a *multiplexer* (MUX), which serves as the entry point for messages from the application, and assigns messages to sessions, and a separate protocol stack for each session (rows on Figure 22). Elements of the per-session stacks are subcomponents owned by the four "controlled elements" (columns on Figure 22): *security* (SEC), *dissemination* (DISS), *reliability* (REL), and *ordering* (ORD), each of which exposes the standard Web interface required for interaction with its corresponding local controller. Now, when the application sends a message, it is first assigned to a session by the multiplexer, and assigned a local sequence number within the session. It is then passed to the appropriate per-session protocol stack, simultaneously to the subcomponents that handle security and ordering. Each of these two subcomponents processes the message independently and concurrently. The security component may encrypt and sign it, if necessary, and then pass it further to the dissemination component for transmission, and independently, to the reliability component to place it in a local cache for the purpose of retransmission, forwarding, and so forth, and to update the local structures to record the fact that the message was created. At the same time, the ordering component, also working in parallel with the dissemination and reliability components, records the presence of the message in its own structures, which are used later by the ordering infrastructure to generate *ordering requests*, to be submitted to the orderer (for details, see the section entitled "Ordering").

On the receive path, the process would look similar (Figure 23). Messages may arrive either through the dissemination framework, in the normal case, or via the reliability framework if they were initially lost, and have been later recovered. Messages that arrive from the dissemination framework are routed via the reliability subcomponent so that they are registered, can be cached, or so that delivery can be suppressed. When ordering arrives from the ordering framework, messages can be decrypted, placed in a buffer (BUF), and delivered in the appropriate order to the application.

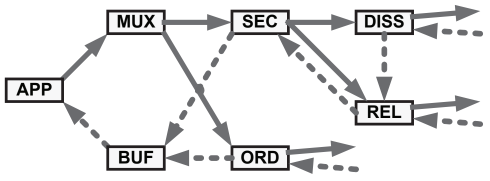
The exact manner in which the subtasks performed by the four subcomponents are syn-

Figure 22. Internal architecture of the application process



Copyright © 2007, IGI Global. Copying or distributing in print or electronic forms without written permission of IGI Global is prohibited.

Figure 23. Processing messages on the send path (red and solid lines) and on the receive path (blue and dotted lines)



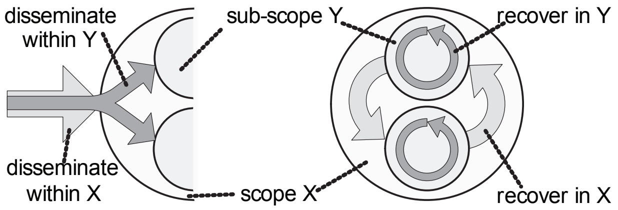
chronized may vary. On the send path, messages may not be transmitted until the entire protocol stack for a given session can be assembled, that is, if the dissemination framework learned of a new session, but the reliability framework has not, the transmission might be postponed until the information about the new session propagates across the reliability framework as well, to avoid problems stemming from such misalignments. On the receive path, the decryption of the message might be postponed until the ordering is known, to avoid maintaining two copies of the message in memory, one for the purpose of loss recovery (encrypted) and one for delivering to the application (decrypted), and so forth.

Hierarchical Approach to Reliability

Our approach to reliability resembles our hierarchical approach to dissemination. Just as channels are decomposed into subchannels, in the reliability framework we decompose the task of loss repair and providing other reliability goals. Recovering messages in a given scope is modeled as recovering within each of its subscopes, concurrently and independently, then recovering across all the subscopes (Figure 24). For example, suppose that scope **X** has members $\mathbf{Y_1}$, $\mathbf{Y_2}$, …, $\mathbf{Y_K}$. A simple reliability property **P**(**X**) requiring that "if some node **x** in the span of scope **X** receives a message **m**, then for as long as either **x** or some other node keeps a copy of it, every other node **y** in the span of **X** will also eventually get **m,**" can be decomposed as follows. First, we ensure **P**($\mathbf{Y_i}$) for every $\mathbf{Y_i}$, that is, we ensure that in each subscope $\mathbf{Y_i}$ of scope **X**, if one node has the message, then so eventually do the others. The protocols that lead to this goal can run in all these subscopes independently and concurrently. Then, we run a protocol across subscopes $\mathbf{Y_1}$, $\mathbf{Y_2}$, …, $\mathbf{Y_K}$, to ensure that if any of them has in its span a node **x** that received message **m**, then each of the other $\mathbf{Y_i}$ also eventually has a node that received **m**. When these tasks, that is, recovery in each $\mathbf{Y_i}$ plus the extra recovery across all $\mathbf{Y_i}$, are all performed for sufficiently long, **P**(**X**) is eventually established.

Coming back to our example, assume that students with their laptops sit in university departments, each of which is a scope. Suppose that some, but not all of the students received a message with a homework problem set from their professor sitting in a cafeteria. We would like to ensure that the problem set gets reliably delivered to all students. In our architecture, this would be achieved by a combination of protocols: a protocol running in each department would ensure that internally, for every

Figure 24. The similarities between a hierarchical dissemination (left) and hierarchical recovery (right)



Copyright © 2007, IGI Global. Copying or distributing in print or electronic forms without written permission of IGI Global is prohibited.

pair **x**, **y** of students, if **x** got the message then so eventually does **y**, and likewise a protocol running across the departments ensures that for each pair of departments **x**, **y**, if some students in **x** got the message, so eventually do some students in **y**. In the end, this yields the desired outcome.

Just like recovery among individual nodes, recovery among LANs might also involve comparing their "state" (such as aggregated ACK/NAK information for the entire LAN) or forwarding lost messages between them. We give an example of this in the section on "Recovery Agents." As mentioned earlier, in our architecture, different recovery schemes may be used in different scopes, to reflect differences in the network topologies, node or communication link capacities, the availability of IP multicast and other local infrastructure, the way subscribers are distributed (e.g., clustered or scattered) and so forth.

For example, in one department, the machines of the students subscribed to topic **T** could form a spanning tree. The property we mentioned above could be guaranteed by making neighbors in the tree compare their state, and upon discovering that one of them has a message **m** that the other is missing, forwarding **m** between the two of them. The same approach may also be used across the departments, that is, departments would form a tree, the departments "neighboring" on that tree could compare what their students got, and perhaps arrange for messages to be forwarded between them. For the latter to be possible, the departments need a way to calculate "what their students got," which is an example of an aggregated, "department-wide" state. Finally, some departments could use a different approach. For example, a department enamored of gossip protocols might require that student machines randomly gossip about messages they got; a department that has had bad experiences with IP multicast and with gossip might favor a reliability protocol that runs on a token ring instead of a tree, and a department with a site-license for a protocol such as SRM (which runs on IP multicast) might favors its use, where the option is available. In each department, a different protocol could be used locally. As long as each protocol produces the desired outcome (satisfies the reliability property inside of the department), and as long as the department has a way to calculate aggregate "department-wide" state needed for inter-department recovery, these very different policies can be simultaneously accommodated.

Just as messages are disseminated through channels, forming what might be termed *dissemination domains*, reliability is achieved via *recovery domains*. A recovery domain **D** in scope **X** may be thought of as a "distributed recovery protocol running among some nodes within **X** that performs recovery-related tasks for a certain set of topics."

For example, when some of the students sitting in a library subscribe to topic **T**, the library might create a "local recovery domain for topic **T.**" This domain could be "realized," for example, as a spanning tree connecting the laptops of the subscribed students and running a recovery protocol between them. The library could internally create many domains, for example, many such trees of student's laptops.

The concept of a recovery domain is dual to the notion of a channel; here we present the analogy:

- Just like a channel is created to disseminate messages for some topics $\mathbf{T_1}$, $\mathbf{T_2}$, …. $\mathbf{T_k}$ in scope **X**, a recovery domain is created to handle loss recovery and other reliability tasks, again for a specific set of topics, and in a specific scope. Just like there could exist multiple channels to a scope, for example, for different sets of topics, there could also exist multiple recovery domains within a single reliability scope, each ensuring reliability for different sets of topics.
- Just as channels may be composed of sub-channels, a recovery domain **D** defined at a scope **X** may be composed of *subdomains* $\mathbf{D_1}$, $\mathbf{D_2}$, …. $\mathbf{D_n}$ defined at subscopes of **X** (we will call them the *members* of **D**). Each such subdomain $\mathbf{D_i}$ handles recovery for a set of subscribers in the respective

Copyright © 2007, IGI Global. Copying or distributing in print or electronic forms without written permission of IGI Global is prohibited.

subscope, while **D** handles recovery across the subdomains. The hierarchy of recovery domains reflects the hierarchy of scopes that have created them, just as channels are decomposed in ways that reflect the hierarchy of scopes that have exposed those channels.

- Just as channels are composed of subchannels via applying filters assigned by forwarding policies, a recovery domain **D** performs its recovery tasks using a *recovery protocol*. Such a protocol, assigned to **D**, specifies how to combine recovery mechanisms in the subdomains of **D** into a mechanism for all of **D**. Recovery protocols are defined in terms of how the subdomains "interact" with each other. We explain how this is done in more detail in the section entitled "Hierarchical Approach to Reliability."
- Just like a single channel may be used to disseminate messages in multiple topics, a recovery domain may run a single protocol to perform recovery simultaneously for a set of topics. In both cases, reusing a single mechanism (a channel, a token ring, a tree, etc.) may significantly improve performance due to the reduction in the total number of control messages and other such optimizations. Indeed, we implemented and evaluated this idea in QSM (Ostrowski & Birman, 2006b, 2006c).

Each individual node is a recovery domain on its own. On the other hand, in a distributed scope such as a LAN, the library in our example, many cases are possible. In one extreme, a single domain may cover the entire LAN. All internal nodes could thus form a token ring, or gossip randomly to exchange ACKs for messages in all topics simultaneously, and use this to arrange for local repairs. In the other extreme, separate domains could be created for every individual topic; subscribers to the different topics could thus form separate structures, such as separate rings and trees, and run separate protocol instances in each of them, exchanging state and the lost messages. In our system, recovery domains actually handle recovery for specific *sessions*, not just specific topics. Each of the recovery domains created internally by a scope performs recovery for some set of sessions, and these sets are such that for each session in which this scope has subscribers, there is a recovery domain in this scope that performs recovery for this session.

A recovery domain **D** of a data center could have as its members recovery domains created by the LANs in that data center (by the SMs of these LANs). Note that in this case, members of **D** would themselves be distributed domains, that is, sets of nodes. A recovery protocol running in **D** would specify how all these different sets of nodes should exchange state and forward lost messages to one another. Note the similarity to a forwarding policy in a data center, which would also specify how messages are forwarded among sets of nodes. As explained in the section on "Recovery Agents" and the section on "Implementing Recovery Domains with Agents," recovery protocols are implemented through delegation, just like forwarding. A concept of a *recovery protocol* is, to some extent, dual, symmetric to the notion of a *forwarding policy*.

Building the Hierarchy of Recovery Domains

Before we show how the hierarchical recovery scheme can be implemented, we need to explain how domains created at different scopes are related to each other. As explained in the preceding section, domains are organized by the relation of membership: domains in superscopes can be thought of as containing domains in subscopes as members. Just as was the case for scopes, a given domain can have many parents, and there may be multiple global domains, but for a given topic, all domains involved in recovery for that topic always form a tree. Domains know their *members* (subdomains) and *owners* (superdomains), and through a mechanism described below, also their *peers* (other domains that have the same parent). This knowledge of *membership* allows the scopes that create those

Copyright © 2007, IGI Global. Copying or distributing in print or electronic forms without written permission of IGI Global is prohibited.

domains to establish distributed structures in an efficient way.

A consistent view of membership is the basis for many reliable protocols, and could benefit many others that don't assume it. Knowing the members of a topic helps to determine which nodes have crashed or disconnected. In existing group communication systems, this is usually achieved by a Global Membership Service (GMS) that monitors failures and membership changes for all nodes, decides when to "install" new membership views for topics, and notifies the affected members of these new views, including the lists of topic members. Nodes then use those membership views to determine, for example, what other nodes should be their neighbors in a tree, who should act as a leader, and so forth.

In our framework, the manager of the root scope for a given topic is responsible for creating the top-level recovery domain, announcing when sessions for that topic begin or end, and so forth. However, if the root SM, which in case of an Internet-wide scope would "manage" the entire Internet, had to process all subscriptions, and respond to every failure across the Internet, it would lead to a non-scalable design: beyond a certain point the system would be constantly in the state of reconfiguration, trying to change membership or install new sessions, and hence unable to make useful progress. It would also violate the principle of isolation: the higher-level scopes would process information that should be local, for example, a corporate network would have to know which nodes in data centers are subscribers, whereas according to our architectural principles, the administrator of a corporate network, and the policies defined at this level, should treat the entire data centers as black boxes.

To avoid the problem just mentioned, rather than collecting all information about membership in a topic **T** and processing it centrally, we distribute this information across all scope managers in the hierarchy of scopes for topic **T** (recall this hierarchy defined in the section entitled "The Hierarchy of Scopes"). Each SM thus has only a partial membership view for each topic and session. This scheme is outlined below.

In the reliability framework, if a scope **X** subscribes to a topic **T**, it first selects or creates a local recovery domain **D** that will handle the recovery for topic **T** locally in **X**, and then sends a request to one of its parent scopes, some **Y**, asking to subscribe this specific domain, to topic **T**. At this point, it is not significant which of its parent scopes **X** directs the request to. **X** may be manually setup by an administrator with the list of parent scopes, and to send requests in all topics that have names matching a certain pattern to a given parent, or it could use an automated, or a semi-automated scheme to discover the parent scopes that it should subscribe with. Exactly how such a discovery scheme can be most efficiently constructed is beyond the scope of this article, but we do hope to explore the issue in a future work.

The superscope **Y** processes the **X**'s subscription request jointly with requests from other of its subscopes, for example, batching them together for efficiency. It then either joins **X** and other subscopes to an existing recovery domain or creates a new one, some **D'.** When joining an existing recovery domain, **Y** follows a special protocol, some details of which are given in the section entitled "Reconfiguration". In any case, scope **Y** informs all scopes of the new membership of domain **D'**. So for each recovery domain **D"** that is a member of **D',** the subscope of **Y** that owns this domain will be notified by **Y** that domain **D'** changed membership, and will be given the list of all subdomains of **D'** together with the names of the scopes that own those subdomains. Finally, if domain **D'** has just been created, and scope **Y** is not the root scope for the topic, **Y** itself sends a request to one of its parent scopes, asking to subscribe domain **D'** to the topic. Topic subscriptions thus travel all the way to the root scope for the topic, in a cascading manner, creating a tree of recovery domains in a bottom-up fashion.

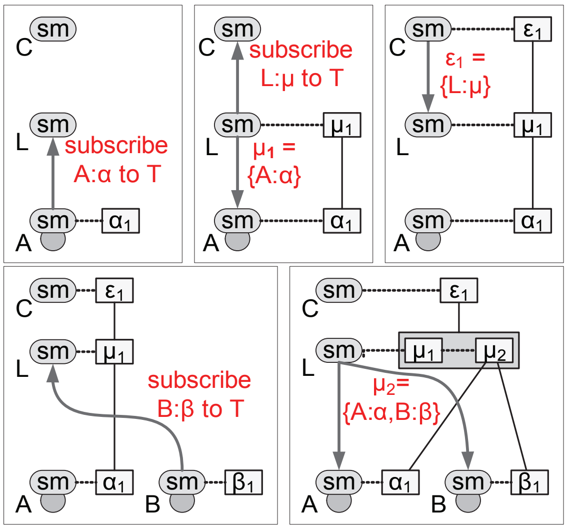
In our example, when a student **A** sitting in a library **L** enters a virtual room **T**, **A**'s laptop creates a local recovery domain $\mathbf{D_A}$, and sends a request to the library server. Suppose that there

Copyright © 2007, IGI Global. Copying or distributing in print or electronic forms without written permission of IGI Global is prohibited.

are other students in the library that are already subscribed to **T**, so the library server already has a recovery domain $\mathbf{D}_L$ that performs recovery in topic **T**. In this case all that is left to do for the library server is to update the membership of $\mathbf{D}_L$, and to inform student **A**'s laptop, as well as the laptops of the other students subscribed to **T**, of the new membership of $\mathbf{D}_L$, so that the laptops can update the distributed recovery structure, and build a new distributed structure. For example, if laptops used a spanning tree, or a token ring, to compare the sets of messages they have received and exchange messages between neighbors, the spanning tree or the ring may be updated to include the new laptop. On the other hand, suppose that student **A** is the first in the library to subscribe to **T**. In this case, the library server creates a new recovery domain $\mathbf{D}_L$, with $\mathbf{D}_A$ as its only member, and sends its own request, in a cascading manner, to a campus server **C**, asking to subscribe $\mathbf{D}_L$ to topic **T**, and so on.

The above procedure effectively constructs a hierarchy of subdomains, with the property that for each topic **T**, the recovery domains subscribed to **T** form a tree. At the same time, a membership hierarchy is built in a distributed manner. Specifically, for each domain $\boldsymbol{\mu}$ in some scope **S**, **S** will maintain a list of the form $\{\ \mathbf{X}_1{:}\boldsymbol{\beta}_1,\ \mathbf{X}_2{:}\boldsymbol{\beta}_2,\ \dots,\ \mathbf{X}_K{:}\boldsymbol{\beta}_K\ \}$, in which $\boldsymbol{\beta}_1$, $\boldsymbol{\beta}_2$, …, $\boldsymbol{\beta}_K$ are the *members*, that is, subdomains of domain $\boldsymbol{\mu}$, and $\mathbf{X}_1$, $\mathbf{X}_2$, …, $\mathbf{X}_K$ are the names of the scopes that own those subdomains (i.e., that created them). In the process of establishing this structure, each of the scopes $\mathbf{X}_i$ receives the list, along with any future updates to it. This information is not "pushed" all the way down to the leaf nodes. Instead, every scope maintains the membership of the domains it created (so, for example, scope **S** maintains the list mentioned above), plus a copy of the membership of all superdomains of the domains it created (so, for example, each $\mathbf{X}_i$ has a copy of the list above), but not the membership of any domains created below it (so, for example, **S** would not track the membership of any of the domains $\boldsymbol{\beta}_1$, $\boldsymbol{\beta}_2$, …, $\boldsymbol{\beta}_K$), more than one level above it (so, for example, while scope **S** would know what are the peers of its own domain $\boldsymbol{\mu}$, scopes $\mathbf{X}_1$, $\mathbf{X}_2$, …, $\mathbf{X}_K$ would not know those

Figure 25. Node A subscribes to topic T with the library L. Library L subscribes with campus C. Membership information and view numbers are passed one level down (never up) the hierarchy.



Copyright © 2007, IGI Global. Copying or distributing in print or electronic forms without written permission of IGI Global is prohibited.

peers, because this information is, logically, two levels "above"), or any internal details of its peers (so, for example, while all of $\mathbf{X_1}$, $\mathbf{X_2}$, …, $\mathbf{X_K}$ would know the membership of $\boldsymbol{\mu}$, that is, the entire list { $\mathbf{X_1}$:$\boldsymbol{\beta_1}$, $\mathbf{X_2}$:$\boldsymbol{\beta_2}$, …, $\mathbf{X_K}$: $\boldsymbol{\beta_K}$ }, they would not know the membership of any of the domains $\boldsymbol{\beta_1}$, $\boldsymbol{\beta_2}$, …, $\boldsymbol{\beta_K}$ besides their own; they only know the names of their peer domains).

Figure 25 shows an example of the structure the system would construct in a scenario similar to the one above. Laptop **A** creates a new domain $\boldsymbol{\alpha}$, which may have a version number, like any other domain, say $\boldsymbol{\alpha_1}$. Then, **A** directs a request subscribe **A**:$\boldsymbol{\alpha}$ to **T** to the library **L**. Note that the version number of $\boldsymbol{\alpha}$ was not included in the request to **L**. This is a detail internal to **A** that **L** does not need to know about. Now, the library creates a new domain $\boldsymbol{\mu}$, and gives it a version number, say $\boldsymbol{\mu_1}$, and directs subscribe **L**:$\boldsymbol{\mu}$ to **T** to the campus **C** (omitting the version number). Concurrently, **L** notifies **A** that **A:**$\boldsymbol{\alpha}$ is now a member of $\boldsymbol{\mu_1}$. This means that a domain $\boldsymbol{\mu}$ has been created at **L**, and version (view) number **1** of $\boldsymbol{\mu}$ has just a single member **A:**$\boldsymbol{\alpha}$. Similarly, **C** creates a new domain $\boldsymbol{\varepsilon}$ with initial version $\boldsymbol{\varepsilon_1}$ that includes a single member **L:**$\boldsymbol{\mu}$ and notifies **L**. Later, another laptop **B** in the library **L** also joins topic **T**. This time, no request is sent to campus **C**. The library handles the request internally. A new version (view) $\boldsymbol{\mu_2}$ of domain $\boldsymbol{\mu}$ is created with two members **A:**$\boldsymbol{\alpha}$ and **B:**$\boldsymbol{\beta}$, and both **A** and **B** are notified of this new view. **A** and **B** undergo a special protocol to "transition" from recovery domain $\boldsymbol{\mu_1}$ to recovery domain $\boldsymbol{\mu_2}$ in a reliable manner, and the protocol running for $\boldsymbol{\mu_1}$ eventually quiesces. The protocols that run at higher levels are unaffected. Domain $\boldsymbol{\varepsilon_1}$ still has only a single member **L:**$\boldsymbol{\mu}$, and the view change that occurred internally in domain $\boldsymbol{\mu}$ is transparent to **C**, and to the protocols that run at this level, and handled internally in the library **L**, between nodes **A** and **B**.

By keeping the information about the hierarchy of domains distributed, and by limiting the way in which this information is propagated to only one level below, we remain faithful to the principles of isolation and local autonomy laid out earlier. At the same time, this enables significant scalability and performance benefits. Because parent domains are oblivious to the membership of their subdomains, and reconfiguration can often be handled internally, as in the example above, churn and failures in lower layers of the hierarchy do not translate to churn and failures in higher layers. A failure of a node or a mobile user with a laptop joining or leaving the system does not need to cause the Internet-wide structure of recovery domains, potentially spanning across tens of thousands of nodes, to fluctuate and reconfigure.

As stated earlier, a single recovery domain may perform recovery for multiple topics (sessions), simultaneously. Additionally, recall that the domain hierarchy, with multiple topics, may not be a tree. We now present an example of how and why this could be the case. Suppose that nodes in a certain scope **L** are clustered based on their interest. Nodes **A** and **B** would be in the same cluster if **A** and **B** subscribed to the same topics. Clusters are thus defined by sets of topic names, for example, cluster $\mathbf{R_{XY}}$ would include nodes that have subscribed to topics **X** and **Y**, and that have not subscribed to any other topics besides these two. The set of nodes subscribed to each topic would thus include nodes in a certain set of clusters. We might even think of as topics "including" clusters, and the clusters including the individual nodes (Figure 26). Accordingly, a scope manager in **L** might create a separate recovery domain for each cluster, and then a separate recovery domain for each topic. If node **A** subscribes to topic **X**, and nodes **B** and **C** subscribe to topic **Y**, then **L** could create a recovery domain $\mathbf{R_X}$ for cluster $\mathbf{R_X}$ and a recovery domain $\mathbf{R_{XY}}$ for cluster $\mathbf{R_{XY}}$. Domain $\mathbf{R_X}$ would have a single member **A:**$\boldsymbol{\alpha}$, while $\mathbf{R_{XY}}$ would have two members, **B:**$\boldsymbol{\beta}$ and **C:**$\boldsymbol{\varphi}$. Scope **L** would also create a local domain **L:X** for topic **X** and **L:Y** for topic **Y**. Domain **L:X** would have members **L:**$\mathbf{R_X}$ and **L:**$\mathbf{R_{XY}}$ while domain **L:Y** would have a single member **L:** $\mathbf{R_{XY}}$. Domains **L:X** and **L:Y** defined at scope **L** could themselves be members of some higher-level domains, defined at a higher-level scope **C**, and so on. Now, the protocol running in

Copyright © 2007, IGI Global. Copying or distributing in print or electronic forms without written permission of IGI Global is prohibited.

domain $\mathbf{R_{XY}}$ at scope **L**, for example, would perform recovery simultaneously for topics **X** and **Y**. As said earlier, the protocol running in $\mathbf{R_{XY}}$ would also be used to calculate aggregate information about domain $\mathbf{R_{XY}}$, to be used in the higher-level protocols. In our example, the information collected by the protocol running in $\mathbf{L{:}R_{XY}}$ would be used by two such protocols, a protocol running in domain **L:X** and a protocol running in **L:Y**.

While the structure just described may seem complex, the ability to perform recovery in multiple topics simultaneously is important in systems like our virtual worlds, where the number of topics (virtual rooms) may be very large. QSM, mentioned previously, uses the architecture just presented, and is able to scale to thousands of publish-subscribe topics.

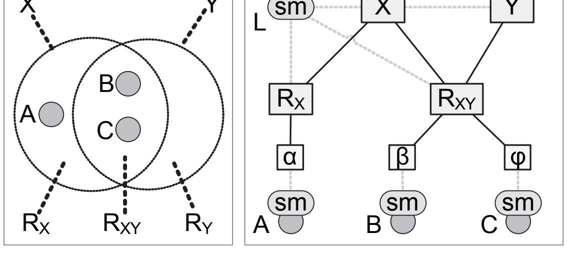
To complete the discussion of recovery hierarchy, we now turn to sessions. As explained earlier, in our architecture recovery is always performed in the context of individual sessions, not topics, because whenever a session changes, so can the reliability properties of the topic. The creation of the domain hierarchy, outlined above, is mostly independent of the creation of sessions. The only case when these two processes are synchronized arises when the last member, across the entire Internet, leaves the topic, or when the first member rejoins the topic after a period when no members existed, for in such cases, it is impossible to handle the event via a local reconfiguration between members (such as transferring the state from some existing member to the newly joining member, or "flushing" any changes from the departing member to some of the existing members). Such an event will force an existing session to be flushed or a new session to be created.

In any case, sessions for a topic **T** are created by the scope that serves as the root for **T**. The root maintains topic metadata and the information about sessions in persistent storage. It assigns new session number whenever a new session is created, and then *installs* the new session and *flushes* the existing session in the top-level recovery domain that it created to perform recovery in topic **T**. More on how the *installing* and *flushing* of a session are realized will be explained in the section "Implementing Recovery Domains with Agents." Now, concurrently with installing of a new session and flushing of the old session in the top-level recovery domain, the scope manager passes the session change event further, to the subdomains that are members of this global recovery domain, by communicating with the scope managers that created those subdomains. This notification travels down the hierarchy of recovery domains in a cascading manner, until the session change events are disseminated across the entire structure.

Recovery Agents

The reader will have noticed by now that the structure of recovery domains we've just described "exists" only virtually, inside the scope managers. These recovery domains are

Figure 26. A hierarchy of recovery domains in a system that clusters nodes based on interest



Copyright © 2007, IGI Global. Copying or distributing in print or electronic forms without written permission of IGI Global is prohibited.

"implemented" by physical protocols running directly between physical nodes, the publishers and subscribers, in a manner similar to how we implemented channels between scopes, by *delegating* the tasks that the recovery domains are responsible for to physical nodes. Just as channels between scopes are "implemented" by physical connections between nodes that can be constrained with filter chains and that are "installed" in the physical nodes by their superscopes, in the reliability framework recovery domains are "implemented" by *agents*. Similarly to filters, these agents are also small, "downloadable" components, which are installed on physical nodes by their superscopes. Before going into details of how precisely this is done, however, we first explain how existing recovery protocols can be modeled in a hierarchical manner that is compatible with our architecture.

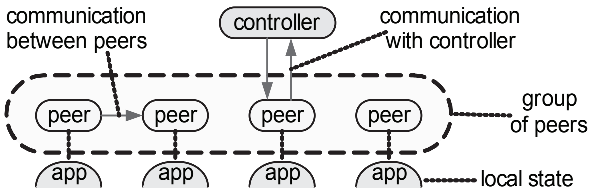
Modeling Recovery Protocols

The *reliability framework* is based on an abstract model of a scalable distributed protocol dealing with loss recovery and other reliability properties. In this model, a protocol such as SRM, RMTP, virtual synchrony, or atomic commit, is defined in terms of a group of cooperating *peers* that exchange control messages and can forward lost packets to each other, and that may perhaps interact with a distinguished node, such as a sender or some node higher in a hierarchy, which we will refer to as a *controller* (Figure 27). The controller does not have to be a separate node; this function could be served by one of the peers. The distinction between the peers and the controller may be purely functional. The point is that the group of peers, as a whole, may be asked to perform a certain action, or calculate a value, for some higher-level entity, such as a sender, a higher-level protocol, or a layer in a hierarchical structure. Examples of such actions include retransmitting or requesting a retransmission for all peers, reporting which messages were successfully delivered to all peers, which messages have been missed by all peers, and so forth. Irrespectively of how exactly the interaction with the controller is realized, it is present in this form or another in almost every protocol run by a set of receivers. We shall refer to the possible interactions between the peers and the controller as the *upper interface*. Notice that some reliability protocols aren't traditionally thought of as hierarchical; we would view them as supporting only a one-level hierarchy. The benefit of doing so is that those protocols can then be treated side by side with protocols such as SRM and RMTP, in which hierarchy plays a central role.

Each peer inspects and controls its *local state*. Such state could include, for example, a list of messages received, and perhaps copies of those that are cached (for loss recovery), the list and the order of messages delivered, and so forth. Operations that a peer may issue to change the local state could include retrieving or purging messages from cache, marking messages as deliverable, delivering some of the previously missed message to the application, and so forth. We refer to such operations, used to view or control the local state of a peer, as a *bottom interface*.

In protocols offering strong guarantees, peers are typically given the membership of

Figure 27. A group of peers in a reliable protocol



Copyright © 2007, IGI Global. Copying or distributing in print or electronic forms without written permission of IGI Global is prohibited.

their group, received as a part of the initialization process, and subsequently updated via *membership change* events. Peers send control messages to each other to share state or to request actions, such as forwarding messages. Sometimes, as in SRM, a multicast channel to the entire peer group exists.

To summarize, in most reliable protocols, a peer could be modeled as a component that runs in a simple environment that provides the following interface: a *membership view* of its peer group, *channels* to all other peers, and sometimes to the entire group, a *bottom interface* to inspect or control local state, and an *upper interface*, to interact with the sender or the higher levels in the hierarchy concerning the aggregate state of the peer group (Figure 28). In some protocols, certain parts of this interface might be unavailable, for example, in SRM peers might not know other peers. The bottom and upper interfaces also would vary.

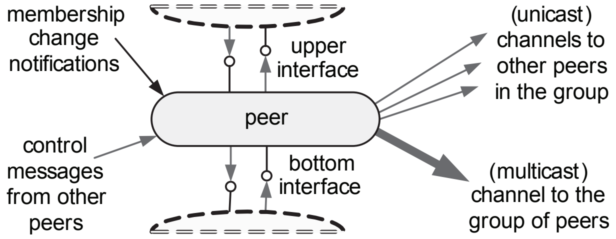
This model is flexible enough to capture the key ideas and features of a wide class of protocols, including virtual synchrony. However, because in our framework protocols must be reusable in different scopes, they may need to be expressed in a slightly different way, as explained below.

In RMTP, the sender and the receivers for a topic form a tree. Within this tree, every subset of nodes consisting of a parent and its child nodes represents a separate local recovery group. The child nodes in every such group send their local ACK/NAK information to the parent node, which arranges for a local recovery within the recovery group. The parent itself is either a child node in another recovery group, or it is a sender, at the root of the tree. Packet losses in this scheme are recovered on a hop-by-hop basis, either top-down or bottom-up, one level at a time. This scheme distributes the burden of processing the individual ACKs/NAKs, and of retransmissions, which is normally the responsibility of the sender. This improves scalability and prevents ACK implosion.

There are two ways to express RMTP in our model. One approach is to view each recovery group consisting of a parent node and its child nodes as a separate group of peers (Figure 29). Since internal nodes in the RMTP tree simultaneously play two roles, a "parent" node in one recovery group and a "child" node in another, we could think of each node as running two "agents," each representing a different "half" of the node, and serving as a peer in a separate peer group. In this perspective it would be not the nodes, but their "halves" that would represent peers. Every group of peers, in this perspective, would include the "bottom agent" of the parent node, and the "upper agents" of its child nodes. When a node sends messages to its child nodes as a result of receiving a message from its parent, of vice versa, we may think of those two "agents" as interacting with each other through a certain interface that one of them views as upper, and the other as bottom. These two types of agents play different roles in the protocol, as explained below.

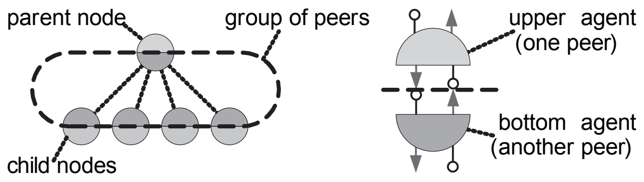
The bottom agent of each node interacts via its *bottom interface* with the local state of

Figure 28. A peer modeled as a component living in abstract environment (events, interfaces, and so forth)


membership change notifications
upper interface
(unicast) channels to other peers in the group
peer
control messages from other peers
bottom interface
(multicast) channel to the group of peers

Copyright © 2007, IGI Global. Copying or distributing in print or electronic forms without written permission of IGI Global is prohibited.

Figure 29. RMTP expressed in our model. A node hosts "agents" playing different roles

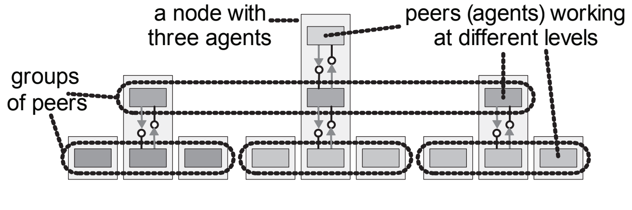


the node. It also serves as a distinguished peer in the peer group, composed of itself and the upper agents of the child nodes. A protocol running in this peer group is used to exchange ACKs between child nodes and the parent node and arrange for message forwarding between peers, but also to calculate collective ACKs for the peer group, that is, which messages were not recoverable in the group. This is communicated by the bottom agent, via its *upper interface*, to the upper agent. The upper agent of every node interacts via its *bottom interface* with the bottom agent. What the upper agent considers as its "local state" is not the local state of the node. Instead, it is the state of the entire recovery group, including the parent and child nodes, that is collected for the upper agent by the bottom agent though the protocol that the bottom agent runs with the upper agents in child nodes. Such interactions, between a component that is logically a part of a "higher layer" ("upper agent") with components that reside in a "lower layer" ("bottom agent"), both components co-located on the same physical node, and connected via their upper and bottom interfaces, are the key element in our architecture.

At the top of this hierarchy is the sender, the root of the tree. The bottom agent of the sender node collects for the upper agent the state of the top-level recovery group, which subsumes the state of the entire tree, and passes it to the upper agent through its upper interface. The upper agent of the sender can thus be thought of as "controlling" through its *bottom interface* the entire receiver tree.

The second way to model RMTP, which builds on the concepts we just introduced, captures the very essence of our approach to combining protocols. It is similar to the first model, but instead of the "upper" and "bottom" agents, each node can now host multiple agents, again connected to each other through their "bottom" and "upper" interfaces. Each of these agents works at a different level. We may think of every node as hosting a "stack" of interconnected agents (Figure 30). In this structure, the sender would not be the root of the hierarchy any more. Rather, it would be treated in the very same way as any of the receivers. The same structure could feature multiple senders, and a single recovery protocol would run for all of them simultaneously.

Figure 30. Another way to express RMTP. Each node hosts multiple "agents" that act as peers at different levels of the RMTP hierarchy

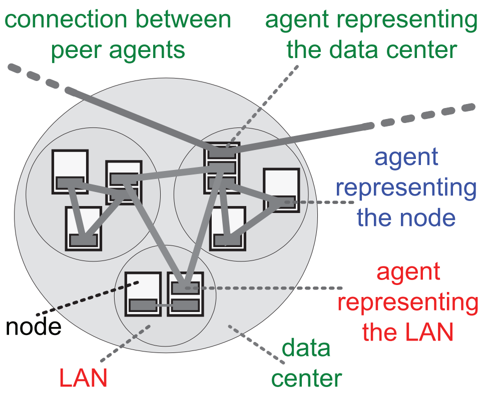


Copyright © 2007, IGI Global. Copying or distributing in print or electronic forms without written permission of IGI Global is prohibited.

Focusing for a moment on a concrete example, assume that nodes reside in three LANs, which are part of a single data center (Figure 30). Each of these administrative domains is a scope. Each node hosts, in its "agent stack," a "node agent" (bottom level, green, Figure 30). The "local state" of the node agent, accessed by the node agent through its bottom interface, is the state of the node, such as the messages that the node received or missed, and so forth. Now, in each LAN, the node agents of all nodes in that LAN form a peer group and communicate with each other to compare their local state, or to arrange for forwarding messages between them. One of these node agents in each LAN serves as a leader (or "parent"), and the others serve as subordinates (or "children"). The leader collects the aggregate ACK/NAK information about the LAN from the entire peer group. The node that hosts the leader also runs anther, higher-level component that we shall call a "LAN agent" (middle level, orange, Figure 30). The LAN agent accesses, through its bottom interface, the aggregated state of the LAN that the "leader" node agent, co-located with it on the same "leader" node, calculated. The LAN agent can therefore be thought of as controlling, through its bottom interface, the entire LAN, just like a node agent was controlling the local node. Now, all the LAN agents in the data center again form a peer group, compare their state (which are aggregate states of their LANs), arrange for forwarding (between the LANs), and calculate aggregate ACK/NAK information about the data center. Finally, one of the nodes that host the LAN agents hosts an even higher-level component, a "data center agent" (top level, blue, Figure 30). The aggregate state of the data center, collected by the peer group of LAN agents, is communicated through the upper interface of the "leader" LAN agent to the data center agent; the latter can now be thought as controlling, through its bottom interface, the entire data center. In a larger system, the hierarchy could be deeper, and the scheme could continue recursively.

Note the symmetry between the different categories of agents. In essence, for every entity, be it a single node, a LAN, or a data center, there exists exactly one agent that collects, via lower-level agents, the state of the entity it represents, and that acts on behalf of this entity in a protocol that runs in its peer group. The agent that represents a distributed scope is always hosted together with one of the agents that represent subscopes. By now, the reader should appreciate that this structure corresponds to the hierarchy of recovery domains we intro-

Figure 31. A node as a "container" for agents

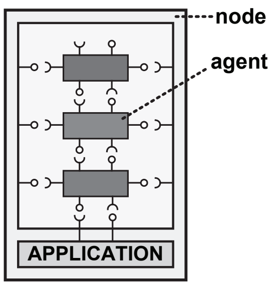

connection between peer agents
agent representing the data center
agent representing the node
agent representing the LAN
node
LAN
data center

Copyright © 2007, IGI Global. Copying or distributing in print or electronic forms without written permission of IGI Global is prohibited.

duced in the section "Building the Hierarchy of Recovery Domains." In our design, every recovery domain is represented, as a part of some higher-level domain, by an agent that collects the state of the domain, which it represents, and acts on behalf of it in a protocol that runs among the agents representing other recovery domains that have the same parent (Figure 31). In order to be able to do their job, these agents are updated whenever a relevant event occurs. For example, they receive a membership change notification, with the list of their peer agents, when a new recovery domain is created. To this end, each agent maintains a bi-directional channel from the scope that created the recovery domain represented by this agent, down to the node that hosts the agent, along what we call an agent "delegation chain."

Note also that as long as the interfaces used by agents to communicate with one-another are standardized, each group of agents could run an entirely different protocol, because the only way the different peer groups are connected with each other is through the bottom and upper interfaces of their agents. For example, in smaller peer groups with small interconnect latency agents could use a token ring protocol, whereas in very large groups over a wide area network, with frequent joins and leaves and frequent configuration changes agents might use randomized gossip protocol. This flexibility could be extremely useful in settings where local administrators control policies governing, for example, use of IP multicast, and hence where different groups may need to adhere to different rules. It also allows for local optimizations. Protocols used in different parts of the network could be adjusted so as to match the local network topology, node capacities, throughput or latency, the present of firewalls, security policies, and so forth. Indeed, we believe that the best approach to building high-performance systems on the Internet scale is not through a uniform approach that forces the use of the same protocol in every part of the network, but by the sorts of modularity our architecture enables, because it can leverage the creativity and specific domain expertise of a very large class of users, who can tune their local protocols to match their very specific needs.

Figure 32. A hierarchy of recovery domains and agents implementing them



The flexibility enabled by our architecture also brings a new perspective on a node in a publish-subscribe system. A node subscribing to the same topics in different portions of the Internet, joining an already established infrastructure (existing recovery domains and agents implementing them running among existing subscribers) may be forced to follow a different protocol, potentially not known in advance. Indeed, if we exploit the full power of modern runtime platforms, a node might be asked to use a protocol that must first be downloaded and installed, in plug-and-play fashion. Because in our architecture, elements "installed" in nodes by the forwarding framework (filters and channels) and by the recovery framework (agents) are very simple, and communicate with the node via a small, standardized API (recall the abstract model of a peer in Figure 28, which can also be interpreted as the abstract model of an agent in the recovery framework), these elements can be viewed as downloadable software components.

Thus, a filter or a recovery agent can be a piece of code, written in any popular language, such as Java or one of the family of .NET languages, that exposes (to the node hosting it, or to agents above and below in the agent stack) and consumes a standardized interface, for example, described in WSDL. Such components could be stored in online repositories, and downloaded

Copyright © 2007, IGI Global. Copying or distributing in print or electronic forms without written permission of IGI Global is prohibited.

as needed, much as a Windows XP user who tries to open a file in the new Vista XPS format will be prompted to download and install an XPS driver, or a visitor to a Web page that uses some special Active-X control will be given an opportunity to download and install that control. In this perspective, the subscribers and publishers that join a publish-subscribe infrastructure, rather than being applications compiled and linked with a specific library that implements a specific protocol, and thus very tightly coupled with the specific publish-subscribe engine, can now be thought of as "empty containers" that provide a standard set of hookups to host different sorts of agents. Nodes using a publish-subscribe system are thus runtime platforms, programmable "devices," elements of a large, flexible, programmable, dynamically reconfigurable runtime environment, offering the sort of flexibility and expressive power unseen in prior architectures.

We believe that in light of the huge success of extensible, component-oriented programming environments, standards for distributed eventing must incorporate the analogous forms of flexibility. To do otherwise is to resist the commercial, off-the-shelf (COTS) trends, and history teaches that COTS solutions almost always dominate in the end. It is curious to realize that although Web services standards were formulated by some of the same companies that are leaders in this componentized style of programming, they arrived at standards proposals that turn out to be both rigid and limited in this respect.

One part of our architecture, for which API standardization options may not be obvious, includes the upper and bottom interfaces. As the reader may have realized, the exact form of these interfaces would depend on the protocol. For example, while a simple protocol implementing the "last copy recall" semantics of the sort we used in some of our examples require agents to be able to exchange a simple ACK/NAK information, more complex protocols may need to determine if messages have been persisted to stable storage, to be able to temporarily suppress the delivery of messages to the application, control purging messages from cache, decide on whether to commit a message (or an operation represented by it) or abort it, and so forth. The "state" of recovery domain and the set of actions that can be "requested" from a recovery domain may vary significantly.

As it turns out, however, defining the upper and bottom interfaces in a standard way is possible for a wide range of protocols. In our technical report (Ostrowski et al., 2006), we have outlined elements of a novel architecture being developed by the three authors of this article, called the "QuickSilver Properties Framework," and based on the very architecture presented here, that achieves precisely this form of standardization. Moreover, the properties framework allows a large class of protocols, including such protocols as virtually synchronous multicast and multicast with transaction semantics, and so forth, to be implemented in a declarative manner, using a special, domain-specific rule-based language, without requiring that the developer worry about performance and scalability aspects.

The key idea behind this approach is based on the observation that the state of most distributed protocols can be accurately described by a set of "properties." Properties are essentially variables that can be associated with various distributed entities (the reader might think of these entities as recovery domains, which they would indeed be, were the system described there to be implemented within the architecture described here). An example of a property is **Received**(**x**), parameterized by an entity name **x**. The value of this variable would be the set of identifiers of all messages that have been received by at least one node that is still in **x**. Other examples, values of all of which would again be sets of message identifiers, include **Cached**(**x**) – the messages cached at some nodes in **x**, **Cleaned**(**x**) – the messages received, but no longer cached in **x**, **Stable**(**x**) – messages received by all nodes in **x**, and so on.

In Ostrowski et al. (2006), we argue that the logic of most protocols could be modeled as a set of rules that determine how the values of such properties are created and propagated.

Copyright © 2007, IGI Global. Copying or distributing in print or electronic forms without written permission of IGI Global is prohibited.

For example, some properties are aggregated. For a distributed entity **x**, **Received**(**x**) can be defined as the set sum of **Received**(**y**) for all **y** that are members of **x**. If this rule is applied recursively to a distributed entity, it will yield the set of messages that are received by any node in the span of that entity, as requested. Similarly, **Stable**(**x**) can be defined as the set intersection of **Stable**(**y**) for all **y** that are members of **x**. A rule that implements message cleanup could be modeled as **CanClean**(**root**) ← **Stable**(**root**), where root is the top-level entity (the top-level recovery domain), and **CanClean**(**x**) is a property, the value of which is disseminated rather than aggregated, that is, passed in a top-down fashion, from the root down to the individual nodes. As it turns out, such rules can be implemented on top of the architecture outlined in this article, using a special version of an agent that supports a few simple mechanisms, such as different flavors of property aggregation or dissemination, and the ability to produce a property based on a value of a certain expression, either periodically, or in response to events, such as receiving a message, and so forth.

We believe that even without using the properties framework, a *set of properties*, expressed in a standard manner, including their "types," is a good candidate for the upper or bottom interface. Agents could still be implemented in an imperative manner, explicitly use the messaging API and the membership notifications from the scopes controlling them, but using the "properties" API to interact with other agents in the agent stack. The details of how exactly such interface could be defined is, however, beyond the scope of this article. Moreover, other similarly expressive schemes may exist. Indeed, it is conceivable that agents co-located on the stack could be able to "negotiate" the manner in which they interact, and download appropriate "converter" components if necessary to ensure that their upper and bottom interfaces match against each other.

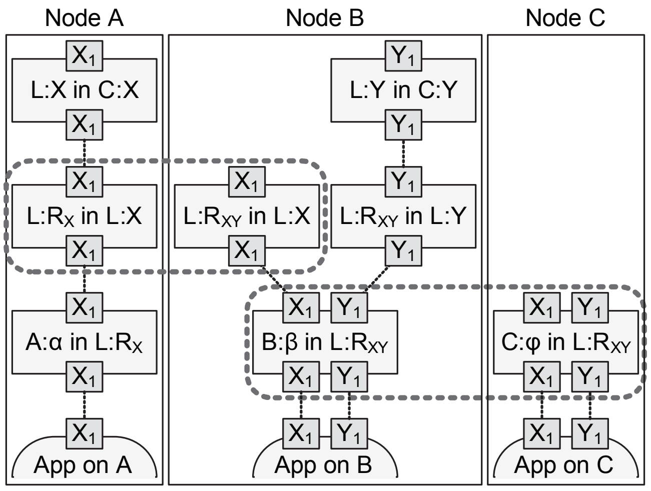
To conclude this section, we now turn to recovery in many topics at once. Throughout this section, the discussion focused on a single topic, or a single session, but as mentioned before, the recovery domains created by the reliability framework, and hence the sets of agents that are instantiated to "implement" those recovery domains, may be requested to perform recovery in multiple sessions at once, for reasons of scalability. As mentioned earlier, after recovery domains are established, and agents instantiated, the root scope may issue requests to install or flush a session, passed along the hierarchy of domains, in a top-down fashion. These notifications are a part of the standard agent API. Agents respond to the notifications by introducing or eliminating information related to a particular session in the state they maintain or control messages they exchange.

For example, agents in a peer group could use a token ring protocol and tokens circulating around that ring could carry a separate recovery record for each session. After a new session would get installed, the token would start to include the recovery record for that session. When a flushing request would arrive for a session, the session would eventually quiesce, and the recovery record related to that session would be eliminated from the tokens, and from the state kept by the agents. The exact manner in which introducing a new session and flushing are expressed would depend on how the agent implements it. The available agent implementation might not support parallel recovery in many sessions at once; in this case, the scope manager could simply create a separate recovery domain for each topic, or even for each session, so that separate agents are used for each.

If an agent performs recovery for many sessions simultaneously, its "upper" and "bottom" interfaces would be essentially arrays of interfaces, one for each session. Likewise, agent stacks might no longer be vertical. The stacks for a scenario of Figure 26 are shown on Figure 33. Here, agents on node B form a tree. Two parts of the upper interface of one agent that correspond to two different sessions that the agent is performing recovery for, are connected with two bottom interfaces of two independent higher-level agents. At this point, the structure depicted in this example may seem

Copyright © 2007, IGI Global. Copying or distributing in print or electronic forms without written permission of IGI Global is prohibited.

Figure 33. Agents stacks that are not simply vertical that may be created on nodes in the scenario from Figure 26. Agents are shown as gray boxes, with the parts of their upper or bottom interfaces corresponding to particular sessions as small yellow boxes. On node B, the bottom-level agent connects the two parts of its upper interface to two different higher-level agents, one to each of them. The peer groups are circled with thick dotted red lines



confusing. In the section entitled "Implementing Recovery Domains with Agents," we come back to this scenario, and we explain how such structures are built.

Implementing Recovery Domains with Agents

In the section entitled "Building the Hierarchy of Recovery Domains," we've explained how a hierarchy of recovery domains is built, such that for each session, there is a tree of domains performing recovery for that session. In the section on "Recovery Agents," we indicated that the recovery domains are "implemented" with agents, and in the section entitled "Modeling Recovery Protocols," we explained how recovery protocols can be expressed in a hierarchical manner, by a hierarchy of agents that represent recovery domains. We now explain how agents are created.

A distributed recovery domain **D** in our framework (i.e., a domain different than a node, not a leaf in the domain hierarchy) will correspond to a peer group. When **D** is created at some scope **X**, the latter selects a protocol to run in **D**, and then every subdomain $\mathbf{Y_k{:}D_k}$ of **D** is requested to create an agent that acts as a "peer $\mathbf{Y_k{:}D_k}$ within peer group **X:D**". We will refer to an agent defined in this manner as "$\mathbf{Y_k{:}D_k}$ in **X:D.**" Note how the membership algorithm provides membership view at one level "above," that is, the scope that owns a particular domain would learn about domains in all the sibling scopes. This is precisely what is required for each peer $\mathbf{Y_k{:}D_k}$ in a peer group **X:D** to learn the membership of its group. Hence, the scopes $\mathbf{Y_k}$ that own the different domains $\mathbf{D_k}$ will learn of the existence of domain **X:D**, each of them will realize that they need to create an agent "$\mathbf{Y_k{:}D_k}$ in **X:D,**" and each of them will receive from **X** all the membership change events it needs to keep its agent with an up to date list of its peers.

For example, on Figure 26, domains **B:β** and **C:φ** are shown as members of $\mathbf{L{:}R_{XY}}$, so according to our rules, agents "**B:β** in $\mathbf{L{:}R_{XY}}$" and "**C:φ** in $\mathbf{L{:}R_{XY}}$" should be created to implement $\mathbf{L{:}R_{XY}}$. Indeed, the reader will find those agents on Figure 33, in the protocol stacks on nodes **B** and **C**.

Copyright © 2007, IGI Global. Copying or distributing in print or electronic forms without written permission of IGI Global is prohibited.

When the manager of a scope **Y** discovers that an agent should be created for one of its recovery domains $\mathbf{D}_k$ that is a member of some **X:D**, two things may happen. If **X** manages a single node, the agent is created locally. Otherwise, **Y** delegates the task to one of its subscopes. As a result, the agents that serve as peers at the various levels of the hierarchy are eventually delegated to individual nodes, their definitions downloaded from an online repository if needed, placed on the agent stack and connected to other agents or to the applications. We thus arrive at a structure just like in Figure 30, Figure 31, Figure 32, and Figure 33, where every node has a stack of agents, linked to one another, with each of them operating at a different level.

While agents are delegated, the records of it are kept by the scopes that recursively delegated the agent, thus forming a *delegation chain*. This chain serves as a means of communication between the agent and the scope that originally requested it to be created. The scope and the agent can thus send messages to one another. This is the way membership changes or requests to install or flush sessions can be delivered to agents.

When the node hosting a delegated agent crashes, the node to which that agent is delegated changes. That is, some other node is assigned the role of running this agent, and will instantiate a new version of it to take over the failed agents responsibilities. Here, we again rely on the delegation chain. When the manager of the superscope of the crashed node (e.g., a LAN scope manager) detects the crash, it can determine that an agent was delegated to the crashed node, and it can request the agent to be redelegated elsewhere. Since our framework would transparently recreate channels between agents, it would look to other peers agents as if the agent lost its cached state (not permanently, for it can still query its bottom interface and talk to its peers). On the one part, this frees the agent developer from worrying about fault-tolerance. On the other part, this requires that agent protocols be defined in a way that allows peers to crash and "resume" with some of their state "erased." Based on our experience, for a wide class of protocols this is not hard to achieve.

Reconfiguration

Many large systems struggle with costs triggered when nodes join and leave. As a configuration scales up, the frequency of join and leave events increases, resulting in a phenomenon researchers refer to as "churn." A goal in our architecture was to support protocols that handle such events completely close to where they occur, but without precluding global reactions to a failure or join if the semantics of the protocol demand it. Accordingly, the architecture is designed so that management decisions and the responsibility for handling events can be isolated in the scope where they occurred. For example, in the section on "Building the Hierarchy of Recovery Domains," we saw a case in which membership changes resulting from failures or nodes joining or leaving were isolated in this manner. The broad principle is to enable solutions where the global infrastructure is able to ignore these kinds of events, leaving the local infrastructure to handle them, without precluding protocols in which certain events do trigger a global reconfiguration.

An example will illustrate some of the tradeoffs that arise. Consider a group of agents implementing some recovery domain **D** that has determined that a certain message **m** is locally cached, and reported it as such to a higher-level protocol. But now suppose that a node crashed and that it happens to have been the (only) one on which **m** was cached. To some extent, we can hide the consequences of the crash: **D** can reconfigure its peer group to drop the dead node and reconstruct associated data structures. Yet **m** is no longer cached in **D** and this may have consequences outside of **D**: so long as **D** cached **m**, higher level scopes could assume that **m** would eventually be delivered reliably in **D**; clearly, this is no longer the case. Thinking back to the properties framework, the example illustrates the risk that a property such as **Cached**(**x**), defined earlier, might not grow

Copyright © 2007, IGI Global. Copying or distributing in print or electronic forms without written permission of IGI Global is prohibited.

monotonically: here, **Cached(x)** has lost an item as a consequence of the crash

This is not the setting for an extended discussion of the ways that protocols handle failures. Instead, we limit ourselves to the observations already made: a typical protocol will want to conceal some aspects of failure handling and reconfiguration, by handling them locally. Other aspects (here, the fact that **m** is no longer available in scope **D**) may have global consequences and hence some failure events need to be visible in some ways outside the scope. Our architecture offers the developer precisely this flexibility: events that he wishes to hide are hidden, and aspects of events that he wishes to propagate to higher-level scopes can do so.

Joining presents a different set of challenges. In some protocols, there is little notion of state and a node can join without much fuss. But there are many protocols in which a joining node must be brought up to date and the associated synchronization is a traditional source of complexity. In our own experience, protocols implementing reconfiguration (especially joins) can be greatly facilitated if members of the recovery domains can be assigned certain "roles". In particular, we found it useful to distinguish between "regular" members, which are already "up to date" and are part of an ongoing run of a protocol, and "light" members, which have just been added, but are still being brought up to date.

When a new member joins a domain, its status is initially "light" (unless this is the first membership view ever created for this domain). The job of the "light" members, and their corresponding agents, is to get up-to-date with the rest of their peer group. At some point, presumably when the light members are more or less synchronized with the active ones, the "regular" agents may agree to briefly suspend the protocol and request some of these "light" peers to be promoted to the "regular" status.

The ability to mark members as "light" or "regular" is a fairly powerful tool. It provides agents with the ability to implement certain forms of agreement, or consensus protocols that would otherwise be hard to support. In particular, this feature turns out to be sufficient to allow our architecture to support virtually synchronous, consensus-like, or transactional semantics.

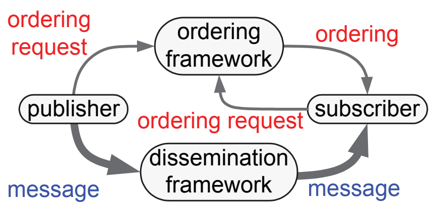
Ordering

As mentioned in the section on "Incorporating Reliability, Ordering, and Security," ordering is implemented independently of dissemination. When a publisher submits a message to the dissemination framework, it simultaneously submits an *ordering request* into the ordering framework. An ordering request is a small object that lists the sender identifier, message topic and a sequence number. In response to the ordering requests, the ordering framework produces *orderings*. Orderings are small objects assigning indexes to batches of messages, perhaps coming from different publishers. These orderings are then delivered to the interested subscribers (Figure 34).

Because the ordering framework is designed to process requests from different publishers together, it is possible to, for example, totally order messages in each topic, or even totally order messages across different topics.

Messages transmitted by publishers may be marked as *ordered*, as a hint for subscribers that these messages should not be delivered until

Figure 34. For all messages for which ordering across multiple senders is required, publishers create ordering requests and submit these in batches to the ordering framework. The latter produces orderings in response to the ordering requests, and delivers these orderings to the subscribers



Copyright © 2007, IGI Global. Copying or distributing in print or electronic forms without written permission of IGI Global is prohibited.

an ordering for these messages arrives through the ordering framework. The subscribers that do not care about ordering will simply not subscribe to the ordering framework, and will ignore such markings.

The ordering framework consists of two components: a component that collects ordering requests from multiple sources and produces orderings in response to these requests, and a component that delivers these orderings to subscribers. The first of these components is essentially an aggregation mechanism, and the second is similar to the dissemination framework.

Like every other element of our architecture, ordering is performed in a hierarchical manner that respects isolation and local autonomy of administrative domains. Thus, the ordering framework also relies on the concept of *management scopes*, in this case the *ordering scopes*, which may or may not overlap with the other flavors of scopes. Each scope can define its own policies that govern the way ordering is performed: how ordering requests are collected, which member of the scope should serve as the *orderer* (collect these requests and produce orderings), and how these orderings should be disseminated to subscribers. The resulting architecture is a product of policies defined at various levels, just as it was the case for dissemination and reliability.

To prevent the article from becoming excessively long, we shall omit the details of the ordering framework. The design shares common elements and ideas with the dissemination and reliability frameworks, and will be covered comprehensively in the first author's Ph.D. thesis.

Other Possible Frameworks

Up until now, we've focused on the dissemination, recovery, and ordering frameworks within our overall architecture. However, the same structure can also support additional frameworks, which exist as logical "siblings" to the ones already presented. These include:

- Flow and rate control. Architectural mechanisms in support of flow and congestion control handling, negotiating rates, managing leases on bandwidth, and so forth.
- Security. Architectural support for integrating security (managing keys, granting or revoking access, managing certificates, etc.) into scalable eventing systems.
- Auditing. Self-verification, detection of inconsistencies (e.g., partitions, invalid routing, and so forth).
- Failure detection and health monitoring. Scalable detection of faulty or underperforming machines.

CONCLUSION

We have argued that new and more flexible event notification standards are going to be needed if the Web services community is to gain the benefits of architectural standardization while also exploiting the full power of component architectures and integration platforms. The proposal presented here draws heavily from our experience building Quicksilver, a new and extremely scalable eventing infrastructure that scales in multiple dimensions, integrates seamlessly with modern component-style platforms, and has the flexibility to support a wide range of reliability models including best-effort, virtual synchrony, consensus, and transactional ones. In contrast, our experience trying to express Quicksilver within the existing Web services options was discouraging; we found them to be narrowly conceived and incapable of offering needed flexibility and scalability.

ACKNOWLEDGMENT

Our research was supported by AFRL/Cornell Information Assurance Institute. We acknowledge Mahesh Balakrishnan, Lars Brenna, Yejin Choi, Maya Haridasan, Tudor Marian, Ingrid Jansch-Porto, Robert van Renesse, Yee Jiun Song, Einar Vollset, and Hakim Weatherspoon for their feedback.

Copyright © 2007, IGI Global. Copying or distributing in print or electronic forms without written permission of IGI Global is prohibited.

REFERENCES

Banavar, G., Chandra, T., Mukherjee, B., Nagarajarao, J., Strom, R., & Sturman, D. (1999, May 31-June 4). An efficient multicast protocol for content-based publish-subscribe systems. In *Proceedings of the 19th IEEE International Conference on Distributed Computing Systems (ICDCS) (IEEE Computer Society),* Washington, D.C. (p. 262).

Box, D., Cabrera, L.F., Critchley, C., Curbera, D., Ferguson, D., Geller, A., et al. (2004). Web services eventing (WS-eventing). Retrieved August 30, 2007, from http://www.ibm.com/developerworks/Webservices/library/specification/ws-eventing/

Floyd, S., Jacobson, V., Liu, C., McCanne, S., & Zhang, L. (1996). A reliable multicast framework for light-weight sessions and application level framing. *IEEE/ACM Transactions on Networking, 5*(6), 784-803.

Graham, S., Niblett, P., Chappell, D., Lewis, A., Nagaratnam, N., Parikh, J., et al. (2004). Web services brokered notification (WS-brokered-notification). Retrieved August 30, 2007, from http://www.ibm.com/developerworks/library/specification/ws-notification/

Keidar, I., & Dolev, D. (1995, May 22-25). Increasing the resilience of atomic commit, at no additional cost. In *Proceedings of the Fourteenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS '95),* San Jose, CA (pp. 245-254). New York, NY: ACM Press.

Keidar, I., Sussman, J., Marzullo, K., & Dolev, D. (2002). Moshe: A group membership service for WANs. *ACM Transcations on Computer Systems*, *20*(3), 191-238.

Ostrowski, K., & Birman, K.P. (2006a, September 18-22). Extensible Web services architecture for notification in large-scale systems. In *Proceedings of the IEEE International Conference on Web Services (ICWS'06)* (Vol. 00, pp. 383-392). Washington, D.C.: IEEE Computer Society.

Ostrowski, K., & Birman, K.P. (2006b, November 3). Scalable group communication system for scalable trust. In *Proceedings of the First ACM Workshop on Scalable Trusted Computing (STC '06)* Alexandria, VA (pp. 3-6). New York, NY: ACM Press.

Ostrowski, K., & Birman, K.P. (2006c). *Scalable publish-subscribe in a managed framework* (Tech. Rep.). Cornell University.

Ostrowski, K., Birman K.P., & Dolev, D. (2006). *Properties framework and typed endpoints for scalable group communication* (Tech. Rep). Cornell University.

Paul, S., Sabnani, K.K., Lin, J. C.-H., & Bhattacharyya, S. (1997). Reliable multicast transport protocol. *IEEE Journal of Selected Areas in Communications, 15*(3), 407-421.

Weinsberg, Y., Dolev, D., Anker, T., & Wyckoff, P. (2006, November). Hydra: A novel framework for making high-performance computing offload capable. In *Proceedings of the 31st IEEE Conference on Local Computer Networks (LCN 2006)*. Tampa, FL.

Krzysztof Ostrowski is a PhD student in the Department of Computer Science at Cornell University. He is currently building QuickSilver, a new type of a development platform with an extremely fast and scalable group communication engine at the core, offering a range of strong reliability properties and a smooth integration with Windows, Visual Studio, and the web service technologies. This new platform is aimed at enabling a new style of programming, characterized by casual use of publish-subscribe topics to represent distributed, interactive content. Before joining Cornell, Ostrowski spent four years in the industry.

Ken Birman is a professor of computer science at Cornell University. He currently heads the QuickSilver project, which is developing the world's fastest and most scalable publish-subscribe system and a new, highly automated, platform aimed at making it dramatically easier to build scalable clustered applications. Previously he worked on fault-tolerance, security, and reliable multicast. In 1987 he founded a company,

Copyright © 2007, IGI Global. Copying or distributing in print or electronic forms without written permission of IGI Global is prohibited.

Isis Distributed Systems, which developed robust software solutions for stock exchanges, air traffic control, and factory automation. For example, Isis currently operates the New York and Swiss Stock Exchanges, the French air traffic control system, and the US Navy AEGIS warship. The technology permits these and other systems to automatically adapt themselves when failures or other disruptions occur, and to replicate critical services so that availability can be maintained even while some system components are down. In contrast to his past work, Birman's recent work has focused on issues of scale, self-management and self-repair mechanisms for complex distributed systems, such as large data centers and wide-area publish-subscribe. The very large scale of these kinds of applications poses completely new challenges. For example, while protocols for data replication on a small scale are closely tied to dahabase concepts such as two-phase commit, these large scale applications are best viewed as probabilistic systems, and the most appropriate technologies are similar to techniques seen in peer-to-peer file sharing applications. Birman is the author of several books. His most recent textbook, Reliable Distributed Computing: Technologies, Web Services, and Applications*, was published by Springer-Verlag in May of 2005. Previously he wrote two other books and more than 200 journal and conference papers, including one that appeared in Scientific American in May, 1996. Dr. Birman was also editor in chief of* ACM Transactions on Computer Systems *from 1993-1998 and is a fellow of the ACM.*

Danny Dolev received his BSc degree in mathematics and physics from the Hebrew University, Jerusalem in 1971. His MSc thesis in applied mathematics was completed in 1973, at the Weizmann Institute of Science, Israel. His PhD thesis was on Synchronization of Parallel Processors *(1979). He was a post-doctoral fellow at Stanford University, 1979-1981, and IBM Research Fellow 1981-1982. He joined the Hebrew University in 1982. From 1987 to 1993 he held a joint appointment as a professor at the Hebrew University and as a research staff member at the IBM Almaden Research Center. He is currently a professor at the Hebrew University of Jerusalem. His research interests are all aspects of distributed computing, fault tolerance, security and networking—theory and practice.*

Copyright © 2007, IGI Global. Copying or distributing in print or electronic forms without written permission of IGI Global is prohibited.

Programming with Live Distributed Objects

Krzysztof Ostrowski[1], Ken Birman[1], Danny Dolev[2], and Jong Hoon Ahnn[1]

[1] Cornell University, and
[2] The Hebrew University of Jerusalem
{krzys, ken, ja275}@cs.cornell.edu, dolev@cs.huji.ac.il

Abstract. A component revolution is underway, bringing developers improved productivity and opportunities for code reuse. However, whereas existing tools work well for builders of desktop applications and client-server structured systems, support for other styles of distributed computing has lagged. In this paper, we propose a new programming paradigm and a platform, in which instances of distributed protocols are modeled as "live distributed objects". Live objects can represent both protocols and higher-level components. They look and feel much like ordinary objects, but can maintain shared state and synchronization across multiple machines within a network. Live objects can be composed in a type-safe manner to build sophisticated distributed applications using a simple, intuitive drag and drop interface, very often without writing any code or having to understand the intricacies of the underlying distributed algorithms.

1 Motivation

It has become common to build applications in a component-oriented manner, composing reusable building blocks by binding strongly-typed interfaces. At runtime, an underlying object-oriented managed environment, such as Java/J2EE or .NET provides further checking and support. The paradigm has numerous benefits: it promotes clean, modular architectures, facilitates extensions, enables collaborative development and code reuse, and by making contracts between components explicit and their code more isolated, reduces the risk of bugs resulting from badly documented or implicit assumptions such as cross-component behavior or side effects.

Unfortunately, distributed systems developers are only able to exploit these tools in limited ways, typically wedded to client-server programming styles. Moreover, the most widely used technologies can be awkward and inflexible. For example, a developer uses different methods to access a system depending on whether it is hosted on a single remote server [6], cloned for load-balancing on a cluster [37], or using state machine replication [52]. Yet even as the available tools have standardized on these limited options, the research community is creating a wave of powerful new technologies that includes peer-to-peer and gossip protocols, multicast with various levels of consistency, ordering and timing, Byzantine state replication, distributed hash tables, credential management services, naming services, content distribution networks, etc.

Our goal is to break through this barrier by treating protocols as components in the same sense as in .NET or COM. We propose a technology in which application components and protocols are unified within a single object-oriented paradigm. Our "live

J. Vitek (Ed.): ECOOP 2008, LNCS 5142, pp. 463–489, 2008.
© Springer-Verlag Berlin Heidelberg 2008

distributed objects" represent running instances of distributed protocols, but they have types and support composition, much like "ordinary" objects. While ours is certainly not the first approach to unify distributed protocols with object-oriented environments, we innovate in ways that make the solution uniquely powerful:

• *We leverage the type system without being language-specific.* Our platform offers mechanisms such as reflection and dynamic type checking, previously seen only in systems closely tied to an underlying language, such as Smalltalk, Java, ML or IOA. In our interactive GUI, type-checking prevents users from dropping objects in inappropriately. Down the road, we'll use type checking to ensure that replicated application objects use a protocol with sufficiently strong properties.

• *It can be incrementally deployed, and supports legacy applications,* including Excel spreadsheets, Oracle databases, and web services. For example, we can import data from a database, multicast it, and export it back into a set of desktop spreadsheets.

• *Our object-oriented embedding can support any distributed protocol as a reusable component.* Existing systems are protocol-agnostic only in the limited sense that users can choose among several different protocols to implement communication. For us, protocols are objects; a small shift in perspective with broad implications.

• *The approach extends from the UI to the hardware level*, whereas prior systems focused on one class of application objects, e.g. shared data structures or UI components[1]. Jini has a vision similar to ours, but is tightly bound to the client-server paradigm, whereas our model is focused on distributed multi-party protocols.

• *We support composition of behavioral protocol types.* Prior composition toolkits either lacked types, or used a limited form of typing, where the protocol type was the type of the implementing class, and composition was achieved via inheritance.

• *Our model is replication-centric.* Although many live objects don't replicate state, the handling of replication and scalability sets our solution apart from prior ones. We're able to support various replication (multicast) models, and to express this in a type system.

• *Our system may be the first drag and drop tool for type-safe protocol composition.* Drag and drop mechanisms are easy to use and yet can support sophisticated applications. For many applications, no new code is needed at all. Prior systems (including some from which we took inspiration, such as Ensemble [33], BAST [20], x-Kernel [45], and I/O automata [36]) were programmer-intensive.

Although the current system is quite usable, live objects raise a number of questions, only some of which have been addressed. The technology requires a scalable multicast layer capable of supporting very large groups, and in which a single node can join large numbers of object-groups. In work reported elsewhere, we describe *Quicksilver*, a high-performance, scalable communication layer that achieves these goals [46,47,48]. We're also collaborating with a group at INRIA/IRISA on a gossip-based infrastructure compatible with live objects; we expect this to be useful for discovering and tracking system configuration information. Looking further out, we're extending Quicksilver to support a range of reliability models (expressed in a new protocol scripting language [47]), and are implementing a new security architecture based on reflection. We also have ideas for WAN and mobile applications, debugging, performance tuning, system management, and object state persistence. However, all of these questions lie beyond the scope of the present paper.

[1] Demos of this functionality and a prototype of our platform are available on our website [34].

2 Prior Work

While we believe our work to be innovative in the ways just described, we're not the first to integrate the object-oriented and distributed programming models.

There are many language abstractions for distributed protocols, including remote objects [17, 20], fault-tolerant objects [24], multicast objects [19], asynchronous collections [9], tuple spaces [6, 38], and replicated objects driven by multicast [25, 37] or two-phase commit [34]. None matches the requirements described above. First, these abstractions are all specialized to support specific protocols. For example, asynchronous collections cannot easily be used to express two-phase commit or leader election. Second, most lack the notion of a distributed type, and in those that do, this notion is shallow, e.g. the type of a multicast object [19] is determined by the type of transmitted events, and the type of an asynchronous collection [9] is the type of the implementing class. The former definition can't convey information about subtle behaviors of protocols such as virtual synchrony [5], while the latter severely restricts reusability. Finally, most lack support for composition.

The idea of defining object types in terms of their *behaviors* is not new [55]. CSP [24] and π-calculus [41] were some of the first protocol specifications, and these early process calculi serve as a basis for recent specification efforts, such as BPEL [3], SSDL [49], and WSCL [4]. As recently noted [19], the weakness of process calculi, and specifications based on them, is that they can't express the semantics of replication or the behavior of protocols such as consensus in a clean way. For example, while BPEL is clearly strong enough to express business processes, the language defines protocols in terms of sets of participants fixed at the outset, and can't model dynamic join or leave events. It would be very hard to express replication properties, such as "*once any group member does X, eventually all operational members do too*" [12].

On the other hand, while *state-based* approaches such as I/O automata [36], CFSM [7], interface automata [1], and others [18] are very expressive, they combine functional descriptions of protocol behaviors with the specifics of their implementations expressed through state transitions. This is useful in correctness proofs, but it may be a weakness in the context of a type system. Two protocols implemented using different data structures and states can exhibit the same external behavior, e.g. "*messages are totally ordered and delivered atomically with respect to failures*". We believe that protocols that behave equivalently should be considered to have the same distributed type; state transition representations can easily obscure such relationships [27].

Live objects support an extensible style of formal behavioral specifications for group and multicast protocols [2, 12, 22, 26]. As one composes protocols, a constructive distributed type system is obtained. The type checking mechanism is itself componentized, and can be extended by developers.

The idea of building protocols from simpler components dates to the x-Kernel [45] and to systems like Ensemble [33], which constructed replication protocols from microprotocols. Among such systems, BAST [20] is closest to ours in terms of the diversity of protocols it can express, but lacks a behavioral notion of a protocol type: protocol types in BAST are determined by the types of the implementing classes, and composition is achieved by inheritance. The creators of BAST observed that in retrospect, inheritance wasn't the right mechanism for this task. We've drawn lessons from these experiences and created a model in which inheritance isn't used at all: we treat protocols as black boxes and connect them with typed event channels in a visual designer. Our protocol objects interact via events, much as in Smalltalk [21].

Jini [57], the widely used Java-based platform in which clients access services by dynamically loading proxy code, is highly relevant prior work. The strongest contrast is that Jini has a pervasive client-server bias, making it very hard to express object replication, particularly in applications that use strong consistency or (at the other extreme) peer-to-peer protocols.

This client-server bias is visible in many ways. First, Jini lacks a rigorous notion of a group [43], and it is hard to implement consistency across a set of group members, state replication within the group, coordination, leader-election, etc. Jini's lookup, join, and discovery specifications lack membership views (needed to assign tasks to group members) and synchronized state transfer (used to initialize new group members). Moreover, Jini doesn't guarantee consistent failure detection. Thus, while services in Jini can be grouped, the mechanism lacks expressive power to facilitate building systems that use stronger forms of replication. Additionally, abstractions such as notification and transactional protocols can't be directly modeled as objects in Jini. Finally, Jini lacks distributed types and protocol composition mechanisms.

Live objects are replication-centric, with a strong notion of protocol types and composition. This makes live objects particularly appropriate for building applications in which users collaborate, share content, or engage in other kinds of peer-to-peer behaviors, (obviously we can also support traditional non-replicated and client-server behaviors). Complex protocols can be modeled as objects, in a manner that separates behavior of the protocol from its implementation.

Many of these same issues distinguish our work from WS-* standards. Elsewhere [48], we discuss issues that arise if one tries to use WS-Notification or WS-Eventing to implement live objects. We concluded that the relevant WS-* standards are tightly bound to specific protocol implementations; as written, they cannot accommodate commercially important protocols such as peer-to-peer video streaming, BitTorrent, or Byzantine replication. We've proposed an extended WS-based eventing standard matched to the work described here, and able to overcome this problem [48].

JXTA [57] is probably the most sophisticated existing collaboration technology for peer-to-peer systems, but it doesn't support stronger replication and consistency models. While JXTA does have notions such as a group and a membership view, members can have inconsistent views. Researchers have struggled to layer reliable multicast on these mechanisms [35]. Groupware toolkits, such as Croquet [53], Groove [39], and group communication [5] toolkits all support replication, and some support strong forms of consistency. However, unlike Jini, JXTA and our work, none of these is positioned as a general-purpose interoperability platform.

3 Model

3.1 Objects and Their Interactions

A *live distributed object* (or *live object*) is an instance of a distributed protocol: programming logic executed by a set of components that may reside on different nodes and communicate by sending messages over the network. For flexibility, we won't assume that the machines running the protocol "know" about one-another or that they share any common state. Thus, a live object could be a Byzantine fault-tolerant replicated state machine, but it could also be an entity with purely local state, one that uses gossip to share data, or an IP multicast channel.

Live objects have *behavioral types*. Suppose that object A logs messages on the nodes where it runs, using a reliable, totally ordered multicast to ensure consistency between replicas. Object B might offer the same functionality, but be implemented differently, perhaps using a gossip protocol. As long as A and B offer the same interfaces and equivalent properties (consistency, reliability, etc), we consider A and B to be implementations of the same type. The concept of behavioral equivalence is the key here; we define it more carefully in section 3.2.

When node Y executes live object X, we'll say that a *proxy* of live object X is running on Y. Thus, a live object is executed by the group of its proxies (Figure 1). A proxy is a functional part of the object running on a node. When two objects have proxies on overlapping sets of nodes, their respective proxies may interact. We can think of the live objects as interacting through their proxies.

A *reference to a live object X* is a complete set of instructions for constructing and configuring a proxy of X on a node. Thus, when node Y wants to access live object X, node Y uses a reference to X as a recipe with which it can create a new proxy for X that will run locally on Y. The proxy then executes the protocol associated with X. For example, it might seek out other proxies for X, transfer the current distributed state from them, and connect itself to a multicast channel to receive updates. Unlike proxies, which can have state, references are just passive, stateless, portable recipes.

The instructions in a reference must be complete, but need not be self-contained. Some of their parts can be stored inside online repositories, from which they need to be downloaded. These repositories are themselves live objects, referenced by the objects that use them. Thus, given a reference, a node can dereference it without prior "knowledge" of the protocol. An exception is thrown if dereferencing fails (for example, if a repository containing a part of the reference is unavailable).

We model proxies in a manner reminiscent of I/O automata. A proxy runs in a virtual context consisting of a set of *endpoints:* strongly-typed bidirectional event channels, through which the proxy can communicate with other software on the same node (Figure 1). Unlike in I/O automata, a proxy can use external resources, such as local network connections, clocks, or the CPU. These interactions are not expressed in our

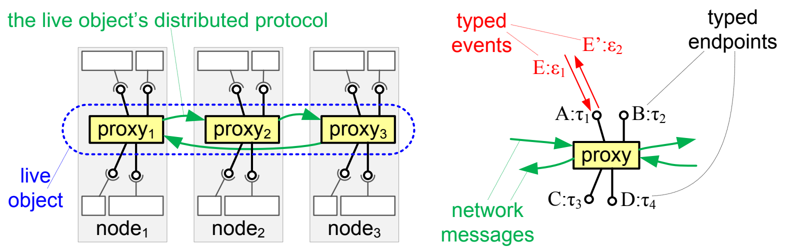


Fig. 1. To access a live object (protocol), a node starts a *proxy:* a software component that runs the protocol on the node, and may communicate with proxies on other nodes by sending messages over the network. On a given node, proxies for different objects communicate via *endpoints:* strongly-typed, bidirectional event channels.

model and they are not limited in any way. However, interactions of a live object's proxy with any other component of the distributed system must be channeled through the proxy's endpoints.

All proxies of the same live object run that live object's *code*. Unlike in state machines [37, 52], we need not assume that proxies run in synchrony, in a deterministic manner, or that their internal states are identical. We do assume that each proxy of a live object X interacts with other components of the distributed system using the same set of endpoints, which must be specified as part of X's type. To avoid ambiguity, we sometimes use the term *instance of endpoint E at proxy P* to explicitly refer to a running event channel *E*, physically connected to and used by *P*.

Because our model is designed to facilitate component integration, we shall adopt a somewhat radical perspective, in which the entire system, all applications and infrastructure are composed of live objects. Accordingly, endpoints of a live object's proxy will be connected to endpoints exposed by proxies of other live objects running on the same node (Figure 2). When proxies of two different objects X and Y are connected through their endpoints on a certain node Z, we'll say that *X* and *Y* are connected on *Z*.

Example (a). Consider a distributed collaboration tool that uses reliable multicast to propagate updates between users (Figure 2). Let **a** be an application object in this system that represents a collaboratively edited document. Proxies of **a** have a graphical user interface, through which users can see the document and submit updates. Updates are disseminated to other users over a reliable multicast protocol, so that everyone can see the same contents. The system is designed in a modular way, so instead of linking the UI code with a proprietary multicast library, the document object **a** defines a typed endpoint **reliable_channel_client**, with which its proxies can submit updates to a reliable multicast protocol (event **send**) and receive updates submitted by other proxies and propagated using multicast (event **receive**). Multicasting can then be implemented by a separate object **r**, which has a matching endpoint **reliable channel**. Proxies of **a** and **r** on all nodes are connected through their matching endpoints. ■

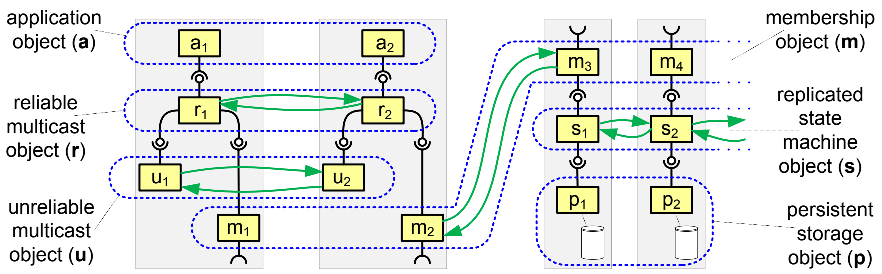

application object (a)
reliable multicast object (r)
unreliable multicast object (u)
a1
a2
r1
r2
u1
u2
m1
m2
m3
m4
s1
s2
p1
p2
membership object (m)
replicated state machine object (s)
persistent storage object (p)

Fig. 2. Applications in our model are composed of interconnected live objects. Objects are "connected" if endpoints of a pair of their proxies are connected. Connected objects can affect one-another by having their proxies exchange events through endpoints. A single object can be connected to multiple other objects. Here, a reliable multicast object **r** is simultaneously connected to an unreliable multicast object **u**, a membership object **m**, and an application object **a**. The same object can be accessed by different machines in different ways. For example, **m** is used in two contexts: by the multicast object **r**, and by replicas of a membership service. The latter employs a replicated state machine **s**, which persists its state through a storage object **p**.

Similarly, object **r** may be structured in a modular way: rather than being a single monolithic protocol, **r** could internally use object **u** for dissemination and object **m** for membership tracking [12]. Additional endpoints **unreliable channel** and **membership** would serve as contracts between **r** and its internal parts **u** and **m**.

Figure 2 illustrates several features of our model. First, a pair of endpoints can be connected multiple times: there are multiple connections between different instances of the **reliable channel** endpoint of object **r** and the **reliable_channel_client** endpoint of **a,** one connection on each node where **a** runs. Since objects are distributed, so are the control and data flows that connect them. If different proxies of **r** were to interact with proxies of **a** in an uncoordinated manner, this might be an issue. To prevent this, each endpoint has a type, which constrains the patterns of events that can pass through different instances of the endpoint. These types could specify ordering, security, fault-tolerance or other properties. The live objects runtime won't permit connections between **a** and **r**, unless their endpoint types declare the needed properties.

A single object could also define multiple endpoints. One case when this occurs is when the protocol involves different roles. For example, the membership object **m** has two endpoints, for clients and for service replicas. The role of the proxy in the protocol depends on which endpoint is connected. In this sense, endpoints are like interfaces in object-oriented languages, giving access to a subset of the object's functionality. Another similarity between endpoints and interfaces is that both serve as contracts and isolate the object's implementation details from the applications using it. We also use multiple endpoints in object **r**, proxies of which require two kinds of external functionality: an unreliable multicast, and a membership service. Both are obligatory: **r** cannot be activated on a platform unless both endpoints can be connected.

Earlier, we commented that not all live objects replicate their state. We see the latter in the case of the persistent store **p.** Its proxies present the same type of endpoint to the state machine **s**, but each uses a different log file and has its own state.

Our model promotes reusability by isolating objects from other parts of the system via endpoints that represent strongly typed contracts. If an object relies upon external functionality, it defines a separate endpoint by which it gains access to that functionality, and specifies any assumptions about the entity it may be connected to by encoding them in the endpoint type. This allows substantial flexibility. For example, object **u** in our example could use IP multicast, an overlay, or BitTorrent, and as long as the endpoint that **u** exposes to **r** is the same, **r** should work correctly with all these implementations. Of course this is conditional upon the fact that the endpoint type describes all the relevant assumptions **r** makes about **u**, and that **u** does implement all of the declared properties.

3.2 Defining Distributed Types

The preceding section introduced endpoint types, as a way to define contracts between objects. We now define them formally and give examples of how typing can be used to express reliability, security, fault-tolerance, and real time properties of objects.

Formally, the type $\mathbf{\Theta}$ of a live object is a tuple of the form $\mathbf{\Theta} = (\mathbf{E}, \mathbf{C}, \mathbf{C'})$. **E** in this definition is a set of named endpoints, $\mathbf{E} = \{(\mathbf{n_1}, \boldsymbol{\tau}_1), (\mathbf{n_2}, \boldsymbol{\tau}_2), \ldots, (\mathbf{n_k}, \boldsymbol{\tau}_k)\}$, where $\mathbf{n_i}$ is the name and $\boldsymbol{\tau}_i$ is the type of the i^th^ endpoint. **C** and **C′** represent sets of constraints describing security, reliability, and other characteristics of the object (**C**), and of its environment ($\mathbf{C}'$). $\mathbf{C}$ models constraints *provided* by the object, such as semantics of the protocol: guarantees that the object's code delivers to other objects connected to it. $\mathbf{C}'$ models constraints *required*, which are prerequisites for correct operation of the object's code. Constraints can be described in any formalism that captures aspects of object and environment behavior in terms of endpoints and event patterns. Rather than trying to invent a new, powerful formalism that subsumes all the existing ones, we build on the concepts of aspect-oriented programming [28], and we define $\mathbf{C}$ to be a finite function from some set $\mathbf{A}$ of aspects to predicates in the corresponding formalisms. For example, constraints $\mathbf{C} = \{(\mathbf{a}_1, \boldsymbol{\varphi}_1), (\mathbf{a}_2, \boldsymbol{\varphi}_2), \ldots, (\mathbf{a}_m, \boldsymbol{\varphi}_m)\}$ would state that in formalism $\mathbf{a}_1$ the object's behavior satisfies formula $\boldsymbol{\varphi}_1$, and so on. We'll give examples of various practically useful formalisms and constraints later in this section.

Type $\boldsymbol{\tau}$ of an endpoint is a tuple of the form $\boldsymbol{\tau} = (\mathbf{I}, \mathbf{O}, \mathbf{C}, \mathbf{C}')$. $\mathbf{I}$ is a set of *incoming events* that a proxy owning the endpoint can receive from some other proxy, $\mathbf{O}$ is a set of *outgoing events* that the proxy can send over this endpoint, and $\mathbf{C}$ and $\mathbf{C}'$ represent constraints provided and required by this endpoint, defined similarly to constraints of the object, but expressed in terms of event patterns, not in terms of endpoints (for example, an endpoint could have an event of type *time*, and with a constraint that time advances monotonically in successive events). Each of the sets $\mathbf{I}$ and $\mathbf{O}$ is a collection of named events of the form $\mathbf{E} = \{(\mathbf{n}_1, \boldsymbol{\varepsilon}_1), (\mathbf{n}_2, \boldsymbol{\varepsilon}_2), \ldots, (\mathbf{n}_k, \boldsymbol{\varepsilon}_k)\}$, where $\mathbf{n}_i$ is event name and $\boldsymbol{\varepsilon}_i$ is its type. Event types can be value types of the underlying type system, such as .NET or Java primitive types and structures, or types described by WSDL [13] etc., but not arbitrary object references or addresses in memory. We assume that events are serializable and can be transmitted across the network or process boundaries. References to live objects are also serializable, hence they can also be passed inside events. The subtyping relation on the event types is inherited from the underlying type system.

The purpose of creating endpoints is to connect them to other, matching endpoints, as described in Section 3.1 and illustrated on Figure 2. **Connect** is the only operation possible on endpoints. We say that endpoint types $\boldsymbol{\tau}_1$ and $\boldsymbol{\tau}_2$ *match*, denoted $\boldsymbol{\tau}_1 \propto \boldsymbol{\tau}_2$, when the following two conditions hold.

1. For each output event $\mathbf{n}$ of type $\boldsymbol{\varepsilon}$ of either endpoint, its counterpart must have an input event with the same name $\mathbf{n}$, and with either type $\boldsymbol{\varepsilon}$, or some supertype of $\boldsymbol{\varepsilon}$. This guarantees that all events can be delivered between the two connected proxies.
2. The provided constraints of each of the endpoints must imply (be no weaker than) the required constraints of the other. This ensures that the endpoints mutually satisfy each other's requirements.

Formally, for $\boldsymbol{\tau}_1 = (\mathbf{I}_1, \mathbf{O}_1, \mathbf{C}_1, \mathbf{C}_1')$ and $\boldsymbol{\tau}_2 = (\mathbf{I}_2, \mathbf{O}_2, \mathbf{C}_2, \mathbf{C}_2')$ we define:

$$\boldsymbol{\tau}_1 \propto \boldsymbol{\tau}_2 \Leftrightarrow \mathbf{O}_1 \to^* \mathbf{I}_2 \wedge \mathbf{O}_2 \to^* \mathbf{I}_1 \wedge \mathbf{C}_1 \Rightarrow^* \mathbf{C}_2' \wedge \mathbf{C}_2 \Rightarrow^* \mathbf{C}_1'. \tag{1}$$

Relation $\to^*$ between two sets of named events expresses the fact that events from the first can be understood as events from the second. Formally, we express it as follows:

$$\mathbf{E} \to^* \mathbf{E}' \Leftrightarrow \forall\, (\mathbf{n}, \boldsymbol{\varepsilon}) \in \mathbf{E}\ \exists\, (\mathbf{n}, \boldsymbol{\varepsilon}') \in \mathbf{E}' \text{ such that } \boldsymbol{\varepsilon} \leq \boldsymbol{\varepsilon}'. \tag{2}$$

Operator "$\leq$" on types always represents the relation of subtyping in this paper.

Relation $\Rightarrow^*$ between two sets of constraints expresses the fact that the constraints in the first set are no weaker than constraints in the second. Formally, we write this as:

$$\mathbf{C} \Rightarrow^* \mathbf{C}' \Leftrightarrow \forall\ (\mathbf{a}, \boldsymbol{\varphi}') \in \mathbf{C}'\ \exists\ (\mathbf{a}, \boldsymbol{\varphi}) \in \mathbf{C} \text{ such that } \boldsymbol{\varphi} \Rightarrow_{\mathbf{a}} \boldsymbol{\varphi}'. \tag{3}$$

Relation $\Rightarrow_{\mathbf{a}}$ is simply a logical consequence in formalism **a**. Intuitively, this definition states that if $\mathbf{C}'$ defines a constraint defined in some formalism, then **C** must define a constraint that is no weaker than that, in the same formalism. For example, if $\mathbf{C}'$ defines some reliability constraint expressed in temporal logic, then **C** must define an equivalent or stronger constraint, also in temporal logic, in order for $\mathbf{C} \Rightarrow^* \mathbf{C}'$ to hold.

For a pair of endpoint types $\boldsymbol{\tau}_1$ and $\boldsymbol{\tau}_2$, the former is a subtype of the latter if it can be used in any context in which the latter can be used. Since the only possible operation on an endpoint is connecting it to another, matching one, hence $\boldsymbol{\tau}_1 \leq \boldsymbol{\tau}_2$ holds iff $\boldsymbol{\tau}_1$ matches every endpoint that $\boldsymbol{\tau}_2$ matches, i.e. $\boldsymbol{\tau}_1 \leq \boldsymbol{\tau}_2$ iff $\forall_{\boldsymbol{\tau}'}\ (\boldsymbol{\tau}_2 \propto \boldsymbol{\tau}') \Rightarrow (\boldsymbol{\tau}_1 \propto \boldsymbol{\tau}')$, which after expanding the definition of "$\propto$" can be formally expressed as follows:

$$\boldsymbol{\tau}_1 \leq \boldsymbol{\tau}_2 \Leftrightarrow \mathbf{O}_1 \rightarrow^* \mathbf{O}_2 \wedge \mathbf{I}_2 \rightarrow^* \mathbf{I}_1 \wedge \mathbf{C}_1 \Rightarrow^* \mathbf{C}_2 \wedge \mathbf{C}_2' \Rightarrow^* \mathbf{C}_1'. \tag{4}$$

Intuitively, $\boldsymbol{\tau}_1 \leq \boldsymbol{\tau}_2$ if (a) $\boldsymbol{\tau}_1$ defines no more output events and no fewer input events than $\boldsymbol{\tau}_2$, (b) the types of output events of $\boldsymbol{\tau}_1$ are subtypes and the types of input events of $\boldsymbol{\tau}_1$ are supertypes of the corresponding events of $\boldsymbol{\tau}_2$, and (c) the provided constraints of $\boldsymbol{\tau}_1$ are no weaker and the required constraints of $\boldsymbol{\tau}_1$ are no stronger than those of $\boldsymbol{\tau}_2$.

Subtyping for live object types is defined in a similar manner. Type $\boldsymbol{\Theta}_1$ is a subtype of $\boldsymbol{\Theta}_2$, denoted $\boldsymbol{\Theta}_1 \leq \boldsymbol{\Theta}_2$, when $\boldsymbol{\Theta}_1$ can replace $\boldsymbol{\Theta}_2$. Since the only thing that one can do with a live object is connect it to another object through its endpoints, this boils down to whether $\boldsymbol{\Theta}_1$ defines all the endpoints that $\boldsymbol{\Theta}_2$ defines, and whether the types of these endpoints are no less specific, and whether $\boldsymbol{\Theta}_1$ guarantees no less and expects no more than $\boldsymbol{\Theta}_2$. Formally, for two types $\boldsymbol{\Theta}_1 = (\mathbf{E}_1, \mathbf{C}_1, \mathbf{C}_1')$ and $\boldsymbol{\Theta}_2 = (\mathbf{E}_2, \mathbf{C}_2, \mathbf{C}_2')$, we define:

$$\boldsymbol{\Theta}_1 \leq \boldsymbol{\Theta}_2 \Leftrightarrow \mathbf{E}_1 \leq^* \mathbf{E}_2 \wedge \mathbf{C}_1 \Rightarrow^* \mathbf{C}_2 \wedge \mathbf{C}_2' \Rightarrow^* \mathbf{C}_1'. \tag{5}$$

Relation $\leq^*$ between sets of named endpoints used above is defined as follows:

$$\mathbf{E} \leq^* \mathbf{E}' \Leftrightarrow \forall\ (\mathbf{n}, \boldsymbol{\tau}') \in \mathbf{E}'\ \exists\ (\mathbf{n}, \boldsymbol{\tau}) \in \mathbf{E} \text{ such that } \boldsymbol{\tau} \leq \boldsymbol{\tau}'. \tag{6}$$

The use of types in our platform is limited to checking whether the declared object contracts are compatible, to ensure that the use of objects corresponds to the developer's intentions. The live objects platform performs the following checks at runtime:

1. When a reference to an object of type $\boldsymbol{\Theta}$ is passed as a value of a parameter that is expected to be a reference to an object of type $\boldsymbol{\Theta}'$, the platform verifies that $\boldsymbol{\Theta} \leq \boldsymbol{\Theta}'$.
2. When an endpoint of type $\boldsymbol{\tau}$ is to be connected to an endpoint of type $\boldsymbol{\tau}'$, either programmatically or during the construction of composite objects described in Section 4.2, the platform verifies that the two endpoints are compatible i.e. that $\boldsymbol{\tau} \propto \boldsymbol{\tau}'$.

We believe that in practice, this limited form of type safety is sufficient for most uses. For provable security, the runtime could be made to verify that live object's code implements the declared type prior to execution. Techniques such as proof-carrying code [44] and domain-specific languages with limited expressive power could facilitate this.

3.3 Constraint Formalisms

We conclude this section with a discussion of different formalisms that can be used to express the constraints in the definition of objects and endpoints. The issue is subtle because on the one hand, a type system won't be very helpful if it has nothing to check, but on the other hand, there are a great variety of ways to specify protocol properties. It isn't much of an exaggeration to suggest that every protocol of interest brings its own descriptive formalism to the table! As noted earlier, many prior systems have effectively selected a single formalism, perhaps by defining types through inheritance. Yet when we consider protocols that might include time-critical multicast, IPTV, atomic broadcast, Byzantine agreement, transactions, secure key replication, and many others, it becomes clear that no existing formalism could possibly cover the full range of options.

A further issue is the incompleteness of many specifications, in a purely formal sense. For example, one popular formalism is temporal logic [22,12]. Here, we assume a global time and a set of locations, and a function that maps from time to events that occur at those locations. In the context of endpoint constraints, we can think of instances of the endpoint as locations, and the endpoint's incoming and outgoing events, and explicit connect/disconnect events, as the events of the temporal logic. Constraints would be expressed as formulas over these events, identifying the legal event sequences within the (infinite) set of possible system histories.

Example (b). Consider the **reliable channel** endpoint, exposed by the reliable channel **r** in the example in Section 3.1. The endpoint's type might define one incoming event **send**(**m**) and one outgoing event **receive**(**m**), parameterized by message body **m**. Constraints provided by the channel object **r** might include a temporal logic formula stating that if event **receive**(**m**) is delivered by **r** through some of the instances of the endpoint sooner than **receive**(**m′**), then for any other instance of the endpoint, if both events are delivered, they are delivered in the same sequence. ■

Example (b) illustrates a safety property of a type for which temporal logic is especially convenient. Chockler et. al. use temporal logic to specify a range of reliable multicast protocols in [12]. However, the FLP impossibility result establishes that these protocols cannot guarantee liveness in traditional networks. Thus, while we can express a liveness constraint in such a logic, no protocol could achieve it – in effect, such a protocol type would be useless in real systems!

Temporal logic is just one of many useful formalisms. In our work on a security architecture, still underway, we're looking into using a variant of the BAN logic [9] to define security properties provided by live objects or expected from their environment. Real-time and performance guarantees are conveniently expressed as probabilistic guarantees on event occurrences, e.g. in terms of predicates such as "*at least* **p** *% of the time*, **receive**(**m**) *occurs at all endpoint instances at most* **t** *seconds following* **send**(**m**)," or "*at least* **p** *% of the time*, **receive**(**m**) *occurs at all different endpoint instances in a time window of at most* **t** *seconds*".

Yet another useful formalism would be a version of temporal logic that talks about the number of instances of different endpoints in time. For example, constraints of the sort "*at most one instance of the* **publisher** *endpoint may be connected at any given time*" could describe single-writer semantics or similar assumptions made by the protocol designer. Constraints of this sort could also express fault-tolerance properties, e.g. define the minimum number of proxies to maintain a certain replication level etc.

In general, with formalisms like those listed above, type-checking might involve a theorem prover, and hence may not always be practical. In practice, however, the majority of object and endpoint types would choose from a relatively small set of standard constraints, such as best-effort, virtually-synchronous, transactional, or atomic dissemination, total ordering of events etc. Predicates that represent common constraints could be indexed, and stored as macros in a standard library of such predicates, and the object and endpoint types would simply list such macros. The runtime would perform type-checking by comparing such lists, and using cached known facts, such as that a virtually synchronous channel is also best-effort reliable etc. By taking advantage of late binding and reflection, features of .NET and of most Java platforms, it is easy to make these mechanisms extensible in a "plug and play" manner. This will allow developers to introduce additional formalisms down the road.

3.4 Group Types

Readers familiar with group communication [5,11] may be concerned that although our model is fundamentally about creating and working with groups of entities (live object proxies), the type system itself lacks a rigorous notion of a group. This actually makes our model simpler and more generic, without preventing us from expressing group properties. For example, to model a virtually synchronous group, we can define a pair of endpoints **channel** and **membership**, and specify constraints on the occurrences of events on the two endpoints, as in group communication specifications [12]. Within groups of endpoints, one can use temporal logic formulas with operators such as *everywhere* and *everywhere within a membership view*, much as in [2,12,22]. To bind to such a group an object would define two matching endpoints. This approach has the advantage of generality: we can potentially express a range of group semantics.

4 Language Embeddings and Support for Composition

4.1 Language Embeddings

Our model has a good fit with modern object-oriented programming languages. There are two aspects of this embedding. On one hand, live object code can be written in a language like Java and C# (we will demonstrate this in Section 4.2). On the other hand, live objects, proxies, endpoints, and connections between them are first-class entities that can be used within C# or Java code. Their distributed types build upon and extend the set of non-distributed types in the underlying managed environment. In this section, we'll discuss each of the new programming language entities we introduce: *references to live objects*, *references to proxies*, *references to endpoint instances*, and *references to connections between endpoints*. An example of their use is shown in Code 1. We will conclude this section with a discussion of two more advanced mechanisms, *template object references* and *casting operator extensions*.

Code 1. An example piece of code in a language similar to C#, but with a simplified syntax for legibility. Here, "ReceiveObject" is a handler of an incoming event of a live object proxy. The event is parameterized by a live object reference "ref_object". If the reference is to a shared folder, the code launches a new proxy to connect to the folder's protocol and attaches a handler to event "AddedElement" generated by this protocol, in order to monitor this folder's contents.

```
01  void ReceiveObject(ref<liveobject> ref_object) // code of an event handler
02  {
03    if (referenced_type(ref_object) is SharedFolder)
04    {
05      ref<SharedFolder> ref_folder := (ref<SharedFolder>) ref_object;
06      SharedFolder folder := dereference(ref_folder); // creates a proxy
07      external<FolderClient> folder_ep := endpoint(folder, "folder");
08      internal<FolderClient> my_ep := new_endpoint<FolderClient>();
09      my_ep.AddedElement += ...; // here's a code that registers an event handler
10      connection my_connection := connect(folder_ep, my_ep);
11      // some code to store the newly created proxy and endpoint connection references
12    }
13  }
```

A. References to Live Objects. Operations that can be performed on these references include reflection (inspecting the referenced object's type), casting, and dereferencing (the example uses are shown in Code 1, in lines 03, 05, and 06 accordingly). Dereferencing results in the local runtime launching a new proxy of the referenced object (recall from Section 3.1 that references include complete instructions for how to do this). The proxy starts executing immediately, but its endpoints are disconnected A reference to the new proxy is returned to the caller (in our example it is assigned to a local variable **folder**). This reference controls the proxy's lifetime. When it is discarded and garbage collected, the runtime disconnects all of the proxy's endpoints and terminates it. To prevent this from happening, in our example code we must store the proxy reference before exiting (we would do so in line 11).

Whereas a proxy must have a reference to it to remain active, a reference to a live object is just a pointer to a recipe for constructing a proxy for that object, and can be discarded at any time. An important property of object references is that they are serializable, and may be passed across the network or process boundaries between proxies of the same or even different live objects, as well as stored on in a file etc. The reference can be dereferenced anywhere in the network, always producing a functionally equivalent proxy – assuming, of course, that the node on which this occurs is capable of running the proxy. In an ideal world, the environmental constraints would permit us to determine whether a proxy actually can be instantiated in a given setting, but the world is obviously not ideal. Determining whether a live object can be dereferenced in a given setting, without actually doing so, is probably not possible.

The types of live object references are based on the types of live objects, which we will define formally below. To avoid ambiguity, if **Θ** is a live object type, and **x** is a reference to an object of type **Θ**, we will write **ref<Θ>** to refer to the type of entity **x**.

The semantics of casting live object references is similar to that for regular objects. Recall that if a regular reference of type **IFoo** points to an object that implement **IBar**, we can cast the reference to **IBar** even if **IFoo** is not a subtype of **IBar**, and while as a result the type of the reference will change, the actual referenced object will not. In a similar manner, casting a live object reference of type **ref<Θ>** to some **ref<Θ′>** produces a reference that has a different type, and yet dereferencing either of these references, the original one or the one obtained by casting, result in the local runtime creating the same proxy, running the same code, with the same endpoints. A reference can be cast to **ref<Θ>** for as long as the actual type of the live object is a subtype of **Θ**.

B. References to Proxies. The type of a proxy reference is simply the type of the object it runs, i.e. if the object is of type **Θ**, references to its proxies are of type **Θ**. Proxy references can be type cast just like live object references. One difference between the two constructs is that proxy references are local and can't be serialized, sent, or stored. Another difference is that they have the notion of a lifetime, and can be disposed or garbage collected. Discarding a proxy reference destroys the locally running proxy, as explained earlier, and is like assigning **null** to a regular object reference in a language like Java. The live object is not actually destroyed, since other proxies may still be running, but if all proxy references are discarded (and proxies destroyed), the protocol ceases to run, as if it were automatically garbage collected.

Besides disposing, the only operation that can be performed on a proxy reference is accessing the proxy endpoints for the purpose of connecting to the proxy. An example of this is seen in line 07, where we request the proxy of the shared folder object to return a reference to its local instance of the endpoint named "folder".

C. References to Endpoint Instances. There are two types of references to endpoint instances, *external* and *internal*. An external endpoint reference is obtained by enumerating endpoints of a proxy through the proxy reference, as shown in line 07. The only operation that can be performed with an external reference is to connect it to a single other, matching endpoint (line 10). After connecting successfully, the runtime returns a connection reference that controls the connection's lifetime. If this reference is disposed, the two connected endpoints are disconnected, and the proxies that own both endpoints are notified by sending explicit **disconnect** events.

An internal endpoint reference is returned when a new endpoint is programmatically created using operator **new** (line 08). This is typically done in the constructor code of a proxy. Each proxy must create an instance of each of the object's endpoints in order to be able to communicate with its environment. The proxy stores the internal references of each of its endpoints for private use, and provides external references to the external code per request, when its endpoints are being enumerated. Internal references are also created when a proxy needs to dynamically create a new endpoint, e.g. to interact with a proxy of some subordinate object that it has dynamically instantiated.

An internal reference is a subtype of an external reference. Besides connecting it to other endpoints, it also provides a "portal" through which a proxy that created it can send or receive events to other connected proxies. Sending is done simply by method calls, and receiving by registering event callbacks (line 09).

An important difference between external and internal endpoint references is that the former could be serialized, passed across the network and process boundaries, and then connected to a matching endpoint in the target location. The runtime can implement this e.g. by establishing a TCP connection to pass events back and forth between proxies communicating this way. This is possible because events are serializable.

Internal endpoint references are not serializable. This is crucial, for it provides isolation. Since any interaction between objects must pass through endpoints, and events exchanged over endpoints must be serializable, this ensures that an internal endpoint reference created by a proxy cannot be passed to other objects or even to other proxies of the same object. Only the proxy that created an endpoint has access to its portal functionality of an endpoint, and can send or receive events with it.

D. References to Connections. Connection references control the lifetime of connections. Besides disposing, the only functionality they offer is to register callbacks, to be invoked upon disconnection. These references are not strongly typed. They may be created either programmatically (as in line 10 in Code 1), or by the runtime during the construction of a composite proxy. The latter is discussed in detail in Section 4.2.

E. Template Object References. Template references are similar to generics in C# or templates in C++. Templates are parameterized descriptions of proxies; when dereferencing them, their parameters must be assigned values. Template types do not support subtyping, i.e. references of template types cannot be cast or assigned to references of other types. The only operation allowed on such references is conversion to non-template references by assigning their parameters, as described in Section 4.2.

Template object references can be parameterized by other types and by values. The types used as parameters can be object, endpoint, or event types. Values used as parameters must be of serializable types, just like events, but otherwise can be anything, including *string* and *int* values, live object references, external endpoint references etc.

Example (c). A channel object template can be parameterized by the type of messages that can be transmitted over the channel. Hence, one can e.g. define a template of a reliable multicast stream and instantiate it to a reliable multicast stream of video frames. Similarly, one can define a template dissemination protocol based on IP multicast, parameterized with the actual IP multicast address to use. A template shared folder containing live objects could be parameterized by the type of objects that can be stored in the folder and the reference to the replication object it uses internally. ■

F. Casting Operator Extensions. This is a programmable reflection mechanism. Recall that in C# and C++, one can often cast values to types they don't derive from. For example, one can assign an integer value to a floating-point type. Conversion code is then automatically generated by the runtime, and injected into this assignment. One can define custom casting operators for the runtime to use in such situations. Our model also supports this feature. If an external endpoint or an object reference is cast to a mismatching reference type, the runtime can try to generate a suitable wrapper.

Example (d). Consider an application designed to use encrypted communication. The application has a user interface object **u** exposing a **channel** endpoint, which it would like to connect to a matching endpoint of an encrypted channel object. But, suppose that the application has a reference to a channel object **c** that is not encrypted, and that exposes a **channel** endpoint of type lacking the required security constraints. When the application tries to connect the endpoints of **u** and **c**, normally the operation would fail with a type mismatch exception. However, if the **channel** endpoint of **c** can be made compatible with the endpoint of **u** by injecting encryption code into the connection, the compiler or the runtime might generate such wrapper code instead. Notice that proxies for this wrapper would run on all nodes where the channel proxy runs, and hence could implement fairly sophisticated functionality. In particular, they could implement an algorithm for secure group key replication. In effect, we are able to wrap the entire distributed object: an elegant example of the power of the model. ■

The same can be done for object references. While casting a reference, the runtime may return a description of a composite reference that consists of the old proxy code, plus the extra wrapper, to run side by side (we discuss composite references in Section 4.2). In addition to encryption or decryption, this technique could be used to automatically inject buffering code, code that translates between push and pull interface, code that persists or orders events, automatically converts event data types, and so on.

Currently, our platform uses casting only to address certain kinds of binary incompatibilities, as explained in Section 5.2. In future work, we plan to extend the platform to support more sophisticated uses of casting, e.g. as in the example above, and define rules for choosing casting operators when more that one is available.

4.2 Construction and Composition

As noted in Section 4.1, a live object *exists* if references to it exist, and it *runs* if any proxies constructed from these references are active. Creating new objects thus boils down to creating references, which are then passed around and dereferenced to create

Code 2. An example live object reference, based on a shared document template, parameterized by a reliable communication channel. The channel is composed of a dissemination object and a reliability object, connected to each other via their "UnreliableChannel" endpoints, much like **r** and **u** in Figure 2. The "ReliableChannel" endpoint of the reliability object is exposed by the channel. The dissemination object reference is to be found as an object named "MyChannel", of type "Channel", in an online repository ("Id" and "Channel" are predefined types). The reference to the repository is to be found, as an object named "QuickSilver", of type "Folder", i.e. containing channels, in another online repository, the "registry" object (see Section 0).

```
parameterized object // an object based on a parameterized template
  using template primitive object 3 // the id of a "shared document" template
{
  parameter "Channel" :
    composite object // a complex object built from multiple component objects
    {
      component "DisseminationObject" :
        external object "MyChannel" as Channel
          from external object "QuickSilver" as Folder<Id, Channel>
            from primitive object 2  // the id of a predefined "registry" object
      component "ReliabilityObject" :
        ... // specification of some loss recovery object, omitted for brevity
      connection // an internal connection between a pair of component endpoints
        endpoint "UnreliableChannel" of "DisseminationObject"
        endpoint "UnreliableChannel" of "ReliabilityObject"
      export // endpoints of the components to be exposed by the composite object
        endpoint "ReliableChannel"   of "ReliabilityObject"
    }
}
```

running applications. Object references are hierarchical: references to complex objects are constructed from references to simpler objects, plus logic to “glue” them together. The construction can use four patterns, for constructing *composite*, *external*, *parameterized*, and *primitive* objects. We shall now discuss these, illustrating them with an example object reference that uses each of these patterns, shown in Code 2.

A. Composite References. A composite object consists of multiple internal objects, running side by side. When such an object is instantiated, the proxies of the internal objects run on the same nodes (like objects **r** and **u** in Figure 2). A composite proxy thus consists of multiple embedded proxies, one for each of the internal objects. A composite reference contains embedded references for each of the internal proxies, plus the logic that glues them together. In the example reference shown in lines 05 to 18 in Code 2, there is a separate section “**component** *name* : *reference*” for each of the embedded objects, specifying its internal name and reference. This is followed by a section of the form “**connection** *endpoint1 endpoint2*”, for each internal connection. Finally, for every endpoint of some embedded internal object that is to be exposed by the composite object as its own, there is a separate section “**export** *endpoint*”.

When a proxy is constructed from a composite reference, the references to any internal proxies and connections are kept by the composite proxy, and discarded when the composite proxy is disposed of (Figure 3). The lifetimes of all internal proxies are thus connected to the lifetime of the composite. Embedded objects and their proxies thus play the role analogous to member fields of a regular object.

B. External References. An external reference is one that has not been embedded and must be downloaded from somewhere. It is of the form “**external object** *name* **as** *type* **from** *reference*”, where *reference* is a reference to the live object that represents some online repository containing live object references, and *name* is the name of the object, the reference to which is to be retrieved from this repository. The type **Θ** of the retrieved object is expected to be a subtype of *type*, and the type of the external reference is **ref**<*type*>. One example of such a reference is shown in lines 08 to 10, and another (embedded in the first one) in lines 09 to 10.

The repository could be any object of type **Θ** ≤ **folder**, where type **folder** is a built-in type of objects with a simple dictionary-like interface. Objects of this type have an endpoint with input event **get**(**n**) and with output events **item**(**n**, **r**) and **missing**(**n**). To retrieve an external reference, the runtime creates a repository object proxy from the embedded reference, runs it, connects to its folder endpoint, submits the **get** event, and awaits response. Once the response arrives, the repository proxy can be discarded.

The “**as** *type*” clause allows the runtime to statically determine the type of the reference without having to engage in any protocol. In case of composite, parameterized, or primitive references, the runtime can derive the type right from the description. The “**as** *type*” clause can still be used in the other categories of references as an explicit type cast, in case it is necessary e.g. to hide some of the object’s endpoints.

The types in the reference (such as **Channel** in line 08 or **Folder<Id**, **Channel>** in line 09) could either refer to the standard, built-in types, or they could be described explicitly using a language based on the formalisms in Section 3.2. To keep our example simple, we assume that all types are built-in, and we refer to them by names.

C. Parameterized References. These references are based on template objects introduced in Section 4.1. They include a section "**using template** *reference*", where *reference* is an embedded template object reference, and a list of assignments to parameter values, each in a separate section of the form "**parameter** *name* : *argument*", where the *argument* could be a type description or a primitive value, e.g. an embedded object reference. For example, the reference in Code 2 is parameterized with a single parameter, **Channel**. The type of the parameter needn't be explicitly specified, for it is determined by the template. In our example, the template expects a live object reference to a reliable communication channel. The specific reference used here to instantiate this template is the composite reference in lines 05 to 18.

D. Primitive References. The types of references mentioned so far provide a means for recursively constructing complex objects from simple ones, but the recursion needs to terminate somewhere. Hence, the runtime provides a certain number of built-in protocols that can be selected by a known 128-bit identifier (lines 02 and 10 in Code 2). Of course even a 128-bit namespace is finite, and many implementations of the live objects runtime could exist, each offering different built-in protocols. To avoid chaos, we reserve primitive references only for objects that either cannot be referenced using other methods, or where doing so would be too inefficient. We will discuss two such objects: the *library* template and the *registry* object.

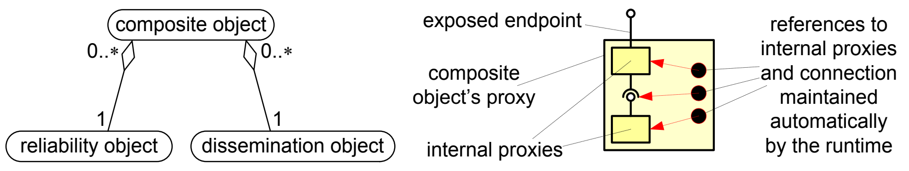


Fig. 3. A live object class diagram for the composite object in Code 2 (left) and the structure of the composite proxy (right). When constructing a composite proxy, the runtime automatically constructs all the internal proxies and the internal connections between them, and stores their references in the composite proxy. Embedded proxies and connections are destroyed together with the composite proxy. The latter can expose some of the internal endpoints as its own.

Code 3. An example live object reference for a custom protocol, implemented in a library that can be downloaded from http://www.mystuff.com/mylibrary.dll. Objects running this protocol are of type "MyType1", and can be found in the library under name "MyProtocol1". The library template provides the folder abstraction introduced in Section 0.

```
external object "MyProtocol1" as MyType1 // my own, custom implementation
  from parameterized object  // an instance of the library template
    using template primitive object 1  // an id of a built-in library template
  {
    parameter "URL" : http://www.mystuff.com/mylibrary.dll
  }
```

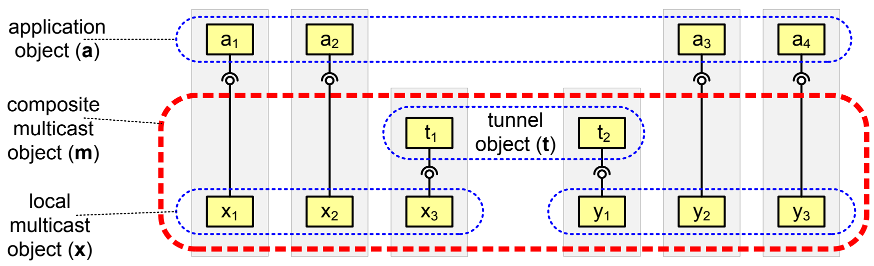


Fig. 4. An example of a hybrid multicast object m, constructed from two local protocols x, y that disseminate data in two different regions of the network, e.g. two LANs, combined using a tunnel object t that acts as a repeater and replicates messages between the two LANs. Different proxies of the composite object **m**, running on different nodes, are configured differently, e.g. some use an embedded proxy of object **x**, while others use an embedded proxy of object **y**.

Code 4. A portable reference to the "hybrid" object **m** from Figure 4 built using the registry.

```
external object "MyChannel" as Channel
  from external object "MyPlatform" as Folder<Id, Channel>
    from primitive object 2 // the registry
```

Code 5. An example of a "proper" use of the registry, to specify a locally configured multicast platform, which could then be used by external references like the one in Code 4. Here, the local instance of the communication platform is configured with the address of a node that controls a region of the Internet, from which other objects can be bootstrapped.

```
parameterized object
  using template external object "MyPlatform" as Folder<Id, Channel>
    from parameterized object // from a binary downloaded from the url below
      using template primitive object 1 // the library template
    { parameter "URL" : http://www.mystuff.com/mylibrary.dll }
{ parameter "LocalController" : tcp://192.168.0.100:60000 }
```

D.1 Library. A library is an object of type **folder**, representing a binary containing executable code, from which one can retrieve references to live objects implemented by the binary. The library template is parameterized by URL of the location where the binary is located (see Code 3, lines 02 to 06). The binary can be in any of the known formats that allow the runtime to locate proxy code, object and type definitions in it, either via reflection, or by using an attached manifest (we show one example of this in Section 5.2). After a proxy of a library is created, the proxy downloads the binary and loads it. When an object reference retrieved from a library is dereferenced, the library locates the corresponding constructor in the binary, and invokes it to create the proxy.

D.2 Registry. The registry object is again a live object of type **folder**, i.e. a mapping of names to object references. The registry references are stored locally on each node, can be edited by the user, and in general, the mapping on each node may be different. Proxies respond to requests by returning the locally stored references.

The registry enables construction of complex heterogeneous objects that can use different internal objects in different parts of the network, as follows

Example (e). Consider a multicast protocol constructed in the following manner: there are two LANs, each running a local IP multicast based protocol to locally disseminate messages: local multicast objects **x** and **y** (Figure 4). A pair of dedicated machines on these LANs also run proxies of a tunneling object **t**, connected to proxies of **x** and **y**. Object **t** acts as a "repeater", i.e. it copies messages between **x** and **y**, so that proxies running both of these protocols receive the same messages. Now, consider an application object **a**, deployed on nodes in both LANs, and having some of its proxies connected to **x**, and some to **y**. From the point of view of object a, the entire infrastructure consisting of **x**, **y**, and **t** could be thought of as a single, composite multicast object **m**. Object **m** is heterogeneous in the sense that its proxies on different machines have a different internal structure: some have an embedded object **x** and some are using **y**. Logically, however, **m** is a single protocol and we'd like to be able to fully express it in our model. The problem stems from the fact that on one hand, references to **m** must be complete descriptions of the protocol, so they should have references to **x** and **y** embedded, yet on the other hand, references containing local configuration details are not portable. The registry object solves this problem by introducing a level of indirection (Code 4). ■

The reader might be concerned that the portability of live objects is threatened by use of the registry. References that involve registry now rely on all nodes having properly configured registry entries. For this reason, we use the registry sparingly, just to bootstrap the basic infrastructure. Objects placed in the registry would represent the entire products, e.g. "the communication infrastructure developed by company XYZ", and would expose the **folder** abstraction introduce earlier, whereby specific infrastructure objects can be loaded. An example of such proper use is shown in Code 5.

5 System

5.1 Embedding Live Objects into the Operating System Via Drag and Drop

Our implementation of the live object runtime runs on Microsoft Windows[2] with .NET Framework 2.0. The system has two major components: an embedding of live objects into Windows drag and drop technologies, discussed here, and embedding of the new language constructs into .NET, discussed in Section 5.2.

Our drag and drop embedding is visually similar to Croquet [53] and Kansas [54], and mimics that employed in Windows Forms, tools such as Visual Studio (or similar ones for Java), and in the Object Linking and Embedding (OLE) [8], XAML [40], and ActiveX standards used in Microsoft Windows to support creation of compound documents with embedded images, spreadsheets, drawings etc. The primary goal is to enable non-programmers to create live collaborative applications, live documents, and business applications that have complex, hierarchical structures and non-trivial internal logic, just by dragging visual components and content created by others from toolbars, folders, and other documents, into new documents or design sheets.

[2] Porting our system from C#/.NET to Mono, to run under Linux, or building a Java/J2EE version of the runtime, shouldn't be a problem, but we haven't yet undertaken this task.

Our hope is that a developer who understands how to create a web page, and understands how to use databases and spreadsheets as part of their professional activities, would use live objects to glue together these kinds of components, sensors capturing real-world data, and other kinds of information to create content-rich applications, which can then be shared by emailing them to friends, placing them in a shared repository, or embedding them into standard productivity applications.

Live object references are much like other kinds of visual components that can be dragged and dropped. References are serialized into XML, and stored in files with a ".liveobject" extension. These ".liveobject" files can easily be moved about. Thus, when we talk about emailing a live application, one can understand this to involve embedding a serialized object reference into an HTML email. On arrival the object can be activated in place. This involves deserializing the reference (potentially running online repository protocols to retrieve some of its parts), followed by analysis of the object's type. Live objects can also be used directly from the desktop browser interface. We configured the Windows shell to interpret actions such as doubleclick on ".liveobject" files by passing the XML content of the file to our subsystem, which processes it as described above. Note that although our discussion has focused on GUI objects, the system also supports services that lack user interfaces.

We have created a number of live object templates based on reliable multicast protocols, including 2-dimensional and 3-dimensional desktops, text notes, video streams, live maps, and 3-dimensional objects such as airplanes and buildings. These can be mashed up to create live applications such as the ones on our web site (Figure 5).

Although the images in Figure 5 are evocative of multi-user role-playing systems such as Second Life, Live Objects differ in important ways. In particular, live objects can run directly on the user nodes, in a peer-to-peer fashion. In contrast, systems such as Second Life are tightly coupled to the data centers on which the content resides and is updated in a centralized manner. In Second Life, the state of the system lives in that data center. Live objects keep state replicated among users. When a new proxy joins, it must obtain some form of a checkpoint to initialize itself, or starts in a **null** state.

As noted earlier, live objects support drag and drop. The runtime initiates a drag by creating an XML to represent the dragged object's reference, and placing it in a clipboard. When a drop occurs, the reference is passed on to the application handling the drop. The application can store it as XML, or it can deserialized it, inspect the type of the dropped object, and take the corresponding action based on that. For example, the spatial desktop on Figure 5, only supports objects with a 3-dimensional user interface. Likewise, the only types of objects that can be dropped onto airplanes are those that represent textures or streams of 3-dimensional coordinates. The decision in each case is made by the application logic of the object handling the drop.

Live objects can also be dropped into OLE-compliant containers, such as Microsoft Word documents, emails, spreadsheets, or presentations. In this case, an OLE component is inserted with an embedded XML of the dragged object's reference. When the OLE component is activated (e.g. when the user opens the document), it invokes the live objects runtime to construct a proxy, and attaches to its user interface endpoint (if there is one). This way, one can create documents and presentations, in which instead of static drawings, the embedded figures can display content powered by any type of a distributed protocol. Integration with spreadsheets and databases is also possible, although a little trickier because these need to access the data in the object, and must trigger actions when a new event occurs.

As mentioned above, one can drag live objects into other live objects. In effect, the state of one object contains a reference to some other live object. This is visible in the desktop example on Figure 5. This example illustrates yet another important feature. When one object contains a reference to another (as is the case for a desktop containing references to objects dragged onto it), it can dynamically activate it: dereference, and connect to the proxy of the stored object, and interact with the proxy. For example, the desktop object automatically activates references to all visual objects placed on it, so that when the desktop is displayed, so are all objects, the references of which have been dragged onto the desktop.

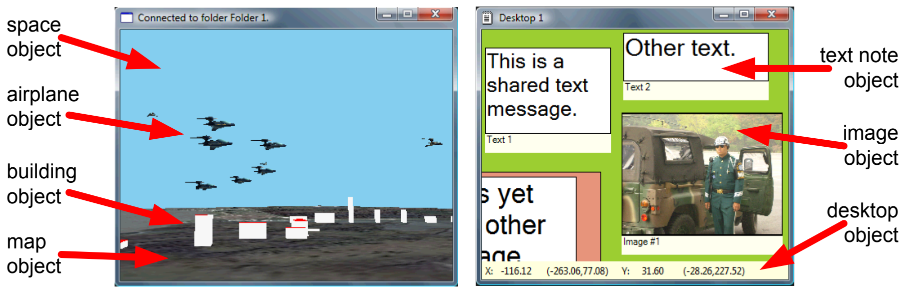


Fig. 5. Screenshots of our platform running live objects with an attached user interface logic. The 3-dimensional space, the area map embedded in this space, as well as each of the airplanes and buildings (left) is a separate live object, with its own embedded multicast channel. Similarly, the green desktop, and the text notes and images embedded in it are independent live objects. Each of these objects can be viewed and accessed from anywhere on the network, and separately embedded in other objects to create various web-style mash-ups, collaborative editors, online multiplayer games, and so on. Users create these by simply dragging objects into one another. Our download site includes a short video demonstrating the ease with which applications such as these can be created.

By now, the reader will realize that in the proposed model, individual nodes might end up participating in large numbers of distributed protocol instances. Opening a live document of the sort shown on Figure 5 can cause the user's machine to join hundreds of instances of a reliable, totally ordered multicast protocol with state transfer, which support the objects embedded in the document. This might lead to scalability concerns. In our prior work we demonstrated that this problem is not a showstopper. Our Quicksilver Scalable Multicast (QSM) system [46], can support thousands of overlapping multicast groups, communicating at network speeds with low overhead.

5.2 Embedding Live Object Language Constructs into .NET Via Reflection

Extending a platform such as .NET to support the new constructs discussed in Section 4.1 would require extending the underlying type system and runtime, thus precluding incremental deployment. Instead, we leverage the .NET reflection mechanism to implement dynamic type checking. This technique doesn't require modifications to the

.NET CLR, and it can be used for other managed environments, such as Java. The key idea is to use ordinary .NET types as "aliases" representing our distributed types. Whenever such an alias type is used in a .NET code, the live objects runtime "understands" that what is "meant" by the programmer is actually the distributed type. Aliases are defined by decorating.NET types with attributes, as in Code 6 and Code 7.

Example (f). Consider a template object type **channel** for multicast channels, parameterized by the .NET type of the messages that can be transmitted. One defines an alias type as a .NET interface annotated with **ObjectTypeAttribute** (Code 6, line 01). When a library object (of Section 4.2) loads a new binary, the runtime scans the binary for .NET types annotated this way and registers them on its internal list of aliases.

Code 6. A .NET interface can be associated with a live object type using an "ObjectType" attribute (line 01). The interface may then be used anywhere to represent the represented live object type. The live objects runtime uses reflection to parse such annotations in binaries it loads and build a library of built-in objects, object types and templates. Object and type templates are defined by specifying and annotating generic arguments (line 03).

```
01 [ObjectTypeAttribute] // annotates "IChannel" as an alias for a live object type
02 interface IChannel<
03   [ParameterAttribute(ParameterClass.ValueClass)] MessageType>
04 {
05   [EndpointAttribute("Channel")] EndpointTypes.IDual<
06     Interfaces.IChannel<MessageType>,
07     Interfaces.IChannelClient<MessageType>>
08   ChannelEndpoint { get; } // returns an external reference to endpoint "Channel"
09 }
```

Parameters of the represented live object type are modeled as generic parameters of the alias. Each generic parameter is annotated with **Parameter Attribute** (line 03), to specify the kind of parameter it represents. The classes of parameters supported by the runtime include *Value*, *ValueClass*, *ObjectClass*, *EndpointClass*, and others we won't discuss here. *Value* parameters are simply serializable values, including .NET types or references to live objects, The others represent the types of values, types of live objects and types of endpoints. For example, we could define a live object type template parameterized by the type of another live object. A practical use of this is a typed folder template, i.e. a folder that contains only references to live objects of a certain type. For example, an instance of this template could be a folder that contains reliable communication channels of a particular type. Another good example is a factory object that creates references of a particular type, e.g. an object that configures new reliable multicast channels in a multicast platform.

An alias interface for a live object type is expected to specify only .NET properties, each annotated with **EndpointAttribute** (line 05). Each property defines one named endpoint for all live objects of this type. The property can only have a getter (line 08), which must return a value of a .NET type that is an alias for some endpoint type. The example in Code 6 uses alias **EndpointTypes**.**IDual<Interface1**, **Interface2>**. This is an alias template built into the runtime, but parameterized by two .NET interfaces.

Code 7. A live object template is defined by decorating a generic class definition (line 01), its generic class parameters (line 03), and constructor parameters (line 08) with .NET attributes. To specify the template live object's type, the class must implement an interface that is annotated to represent a live object type (line 04 referencing the definition shown in Code 6). In the body of the class, we create endpoints to be exposed by the proxy (created in lines 11-12, exposed in lines 19-25), handle incoming events (line 27) and send events through its endpoints.

```
01 [ObjectAttribute("89BF6594F5884B6495F5CD78C5372FC6")]
02 sealed class MyChannel<
03   [ParameterAttribute(ParameterClass.ValueClass)] MessageType>
04 : ObjectTypes.IChannel<MessageType>, // specifies the live object type
05   Interfaces.IChannel // we implement handlers to all incoming events, see line 12
06 {
07   public MyChannel(
08     [Parameter(ParameterClass.Value)] // also a parameter of the template
09     ObjectReference<ObjectTypes.IMembership> membership_reference)
10   {
11     this.myendpoint = new Endpoints.Dual<
12       Interfaces.IChannel, Interfaces.IChannelClient>(this);
13     ... // the rest of the constructor would contain code very similar to that in Code 1
14   }
15   // this is our internal reference to the channel endpoint, the endpoint's "backdoor"
16   private Endpoints.Dual<
17     Interfaces.IChannel, Interfaces.IChannelClient> myendpoint;
18
19   EndpointTypes.IDual<
20     Interfaces.IChannel<MessageType>,
21     Interfaces.IChannelClient<MessageType>>
22   ObjectTypes.IChannel.ChannelEndpoint
23   {
24     get { return myendpoint; } // returns an external endpoint reference
25   }
26    // this is a handler for one of the incoming events of the channel endpoint
27   Interfaces.IChannel.Send(MessageType message) { ... } // details omitted
28   ... // the rest of the alias definition, containing all the other event handlers etc.
29 }
```

The methods defined by these interfaces, again accordingly annotated, are used by the runtime to compile the list of this endpoint's incoming and outgoing events, and similar annotations can be used to express its constraints. When the alias defined this way is used in some context with its generic parameters assigned (lines 05-07), the runtime treats it as an alias for the specific endpoint type, with the specific events defined by those interfaces.

Having defined the object's type, we can define the object itself. This is again done via annotations. An example definition of a live object template is shown in Code 7.

A live object template is defined as a .NET class, the instances of which represent the object's proxies. The class is annotated with **ObjectAttribute** (line 01) to instruct the runtime to build a live object definition from it. This template has two parameters: the type parameter representing the type of messages carried by the channel (line 03), and a "value" parameter - the reference to the membership object that this channel should use (lines 08-09). To specify the type of the live object, line 03 inherits from an alias. This forces our class to implement properties returning the appropriate endpoints (lines 19-25). The actual endpoints are created in the constructor (lines 11-12). While creating endpoints, we connect event handlers for incoming events (hooking itself up, in line 12, and implementing these handlers, as in line 27). ■

While the use of aliases is convenient as a way of specifying distributed types, alias types are, of course, not distributed, and the .NET runtime doesn't understand subtyping rules we defined in Section 3.2. The actual type checking is done dynamically. When the programmer invokes a method of a .NET alias to request a type cast, or to create a connection between endpoints, the runtime uses its internal list of aliases to identify the distributed types involved and performs type checking by itself. The physical .NET types of aliases are irrelevant. Indeed, if the runtime determines that two different .NET types are actually aliases for the same distributed type, it will inject a wrapper code, as demonstrated below.

Example (g). Suppose that binary **Foo.dll** defines an object type alias **IChannel** as in Code example 6, and an object template alias **MyChannel** as in Code example 7. Now, suppose that a different, unrelated binary **Bar.dll** also defines an alias **IChannel** in exactly the same way, as in Code 6, and then uses this alias, e.g. in the definition of an application object that could use channels of the corresponding distributed type. If both binaries are loaded by the live objects runtime, we will end up with two distinct, binary-incompatible .NET aliases **IChannel**, representing the same distributed type. Whenever the programmer makes an assignment between these two types, the runtime dynamically creates, compiles, and injects the appropriate wrapper to forward method calls between the incompatible interfaces, to make the assignment legal in .NET. ■

6 Conclusions

Our paper described the architecture and implementation of a system supporting live distributed objects, a strongly typed, object-oriented platform in which distributed protocols are treated as first-class objects. The platform is working and quite versatile, but is still a work in progress. Future challenges include implementing our security and WAN architectures (designed but not yet operational), providing runtime monitoring and debugging tools, and automated self-configuration and tuning.

Acknowledgements. Our work was funded by AFRL/IF, AFOSR, NSF, I3P and Intel. We'd like to thank Mahesh Balakrishnan, Kathleen Fisher, Paul Francis, Lakshmi Ganesh, Rachid Guerraoui, Chi Ho, Maya Haridasan, Annie Liu, Tudor Marian, Greg Morrisett, Andrew Myers, Anil Nerode, Robbert van Renesse, Yee Jiun Song, Einar Vollset, and Hakim Weatherspoon for the feedback they provided.

References

1. de Alfaro, L., Henzinger, T.: Interface automata. SIGSOFT Softw. Eng. Notes 26, 5 (2001)
2. Anceaume, E., Charron-Bost, B., Minet, P., Toueg, S.: On the Formal Specification of Group Membership Services. Cornell University Tech. Report TR95-1534 (August 1995)

3. Andrews, T., et al.: Business Process Execution Language for Web Services v1.1. May (2003), `http://download.boulder.ibm.com/ibmdl/pub/software/dw/specs/ws-bpel/ws-bpel.pdf`
4. Banerji, A., et al.: Web Services Conversation Language (WSCL), http://www.w3.org/TR/wsc110
5. Birman, K.: The Process Group Approach to Reliable Distributed Computing. Communications of the ACM 36(12), 37–53 (1993)
6. Birrell, A., Nelson, G., Owicki, S., Wobber, W.: Network Objects. In: SOSP 1993
7. Brand, D., Zafiropulo, P.: On communicating finite-state machines. JACM, 30(2) (1983)
8. Brockschmidt, K.: Inside OLE. Microsoft Press (1995)
9. Burrows, M., Abadi, M., Needham, R.: A Logic of Authentication. TOCS 8(1), 18–36 (1990)
10. Carriero, N., Gelernter, D.: Linda in Context. CACM 32(4), 444–458 (1989)
11. Cheriton, D., Zwaenepoel, W.: Distributed Process Groups in the V Kernel. ACM Transactions on Computer Systems 3(2), 77–107 (1985)
12. Chockler, G., Keidar, I., Vitenberg, W.: Group Communication Specifications: A Comprehensive Study. ACM Computer Surveys 33(4):1, 43 (2001)
13. Christensen, E., Curbera, F., Meredith, G., Weerawarana, S.: Web Services Description Lan-guage (WSDL). W3C Note 15 March (2001), `http://www.w3.org/TR/wsdl`
14. Eugster, P., Guerraoui, R.: On Objects and Events. In: OOPSLA 2001, pp. 254–269 (2001)
15. Eugster, P., Guerraoui, R.: Distributed Programming with Typed Events. IEEE Software 21(2), 56–64 (2004)
16. Eugster, P., Damm, H., Guerraoui, R.: Towards Safe Distributed Application Development. In: ICSE 2004, pp. 347–356 (2004)
17. Eugster, P., Guerraoui, R., Sventek, J.: Distributed Asynchronous Collections: Abstractions for Publish/Subscribe Interaction. In: Bertino, E. (ed.) ECOOP 2000. LNCS, vol. 1850, pp. 252–276. Springer, Heidelberg (2000)
18. Fu, X., Bultan, T., Su, J.: Conversation Specification: A New Approach to Design and Anal-ysis of E-Service Composition. In: WWW 2003, Budapest, Hungary, May 20-24 (2003)
19. Fuzzati, R., Nestmann, U.: Much Ado About Nothing. In: Algebraic Process Calculi, the First Twenty Five Years and Beyond. Process algebra, `http://www.brics.dk/NS/05/3/`
20. Garbinato, B., Guerraoui, R.: Using the Strategy Pattern to Compose Reliable Distributed Protocols. In: Proceedings of 3rd USENIX COOTS, Portland, Oregon (June 1997)
21. Goldberg, A., Robson, D.: Smalltalk-80: the language and its implementation. Addison-Wesley Longman Publishing Co., Inc., Boston (1983)
22. Halpern, J., Fagin, R., Moses, Y., Vardi, M.: Reasoning about Knowledge. MIT Press, Cambridge (1995)
23. Hickey, J., Lynch, N., van Renesse, R.: Specifications and proofs for Ensemble layers. In: Cleaveland, W.R. (ed.) ETAPS 1999 and TACAS 1999. LNCS, vol. 1579, Springer, Heidelberg (1999)
24. Hoare, C.: Communicating sequential processes. CACM 21(8), 666–677 (1978)
25. Jul, E., Levy, H., Hutchinson, N., Black, A.: Fine-Grained Mobility in the Emerald System. ACM TOCS 6(1), 109–133
26. Karr, D.: Specification, Composition, and Automated Verification of Layered Communication Protocols. Ph.D. Thesis. Cornell University (1997)

27. Keidar, I., Khazan, R., Lynch, N., Shvartsman, A.: An inheritance-based technique for building simulation proofs incrementally. ACM Trans. Soft. Eng. Methodol. 11(1), 63–91 (2002)
28. Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C., Loingtier, J.-M., Irwin, J.: Aspect-Oriented Programming. In: Aksit, M., Matsuoka, S. (eds.) ECOOP 1997. LNCS, vol. 1241, pp. 220–242. Springer, Heidelberg (1997)
29. Krumvieda, C.: Distributed ML: Abstractions for Efficient and Fault-Tolerant Prgramming. Technical Report, TR93-1376, Cornell University (1993)
30. Lamport, L.: The Temporal Logic of Actions. ACM Toplas 16(3), 872–923 (1994)
31. Liskov, B.: Distributed Programming in Argus. CACM 31(3), 300–312 (1988)
32. Liskov, B., Schieffler, R.: Guardians and Actions: Linguistic Support for Robust, Distributed Programs. ACM TOPLAS 5, 3 (1983)
33. Liu, X., Kreitz, C., van Renesse, R., Hickey, J., Hayden, M., Birman, K., Constable, R.: Building Reliable, High-Performance Communication Systems from Components. In: SOSP (1999)
34. Live Objects at Cornell, http://liveobjects.cs.cornell.edu/
35. Loesing, K., Wirtz, G.: An Implementation of Reliable Group Communication Based on the Peer-to-Peer Network JXTA. In: AICCSA 2005 (2005)
36. Lynch, N., Tuttle, M.: Hierarchical correctness proofs for dist.ributed algorithms. In: PODC 1987 (1987)
37. Maffeis, S., Schmidt, D.: Constructing Reliable Distributed Communication Systems with CORBA. IEEE Communications Magazine 14 (February 1997)
38. Makpangou, M., Gourhant, Y., Le Narzul, J.-P., Shapiro, M.: Fragmented Objects for Distri-buted Abstractions, pp. 170–186. IEEE Computer Society Press, Los Alamitos (1994)
39. Microsoft. Microsoft Office Groove, http://office.microsoft.com/en-us/groove/
40. Microsoft. XAML Overview, http://msdn2.microsoft.com/en-us/library/ms752059.aspx
41. Milner, R., Parrow, J., Walker, D.: A Calculus of Mobile Processes, parts I and II. LFCS Report 89-85. University of Edinburgh (June 1989)
42. Miranda, H., Pinto, A., Rodrigues, L.: Appia, a Flexible Protocol Kernel Supporting Multiple Coordinated Channels. In: Proc. of 21st ICDCS, Phoenix, Arizona, pp. 707–710 (2001)
43. Montresor, A., Davoli, R., Babaoglu, O.: Enhancing Jini with group communication. In: ICDCS Workshop, April 2001, pp. 69–74 (2001)
44. Necula, G.: Proof-Carrying Code. ACM SIGPLAN-SIGACT POPL 1997 (1997)
45. O'Malley, S., Peterson, L.: A Dynamic Network Architecture. TOCS 10(2), 110–143 (1992)
46. K. Ostrowski, K. Birman, D. Dolev. Quicksilver Scalable Multicast (in submission)
47. Ostrowski, K., Birman, K., Dolev, D.: Declarative Reliable Multi-Party Protocols. Cornell University Technical Report, TR2007-2088 (March 2007)
48. Ostrowski, K., Birman, K., Dolev, D.: Extensible Architecture for High-Performance, Scalable, Reliable Publish-Subscribe Eventing and Notification. JWSR v. 4, no 4 (October- December 2007)
49. Parastatidis, S., Webber, J., Woodman, S., Kuo, D., Greenfield, P.: SOAP Service Description Language (SSDL). Technical Report, University of Newcastle, CS-TR-899 (2005)
50. Reiter, M., Birman, K.: How to securely replicate services. In: TOPLAS, vol. 16(3), pp. 986–1009 (1994)
51. van Renesse, R., Birman, K., Hayden, M., Vaysburd, A., Karr, D.: Building Adaptive Systems Using Ensemble. Software Practice and Experience. 28(9), pp. 963-979 (August 1998)

52. Schneider, F.: Implementing Fault-Tolerant Services Using the State Machine Approach: a Tutorial. ACM Computng Surveys 22(4), 299–319 (1990)
53. Smith, D., Kay, A., Raab, A., Reed, D.: Croquet: a collaboration system architecture. Creating, Connecting and Collaborating Through Computing, C5 2003, p. 2–9 (2003)
54. Smith, R., Wolczko, M., Ungar, D.: From Kansas to Oz: Collaborative Debugging When a Shared World Breaks. CACM, 72–78 (1997)
55. Snyder, A.: Encapsulation and Inheritance in Object-Oriented Programming Languages. In: OOPLSA 1986
56. van Steen, M., Homburg, P., Tanenbaum, A.: Globe: A Wide Area Distributed System. IEEE Concurrency 7(1), 70–78 (1999)
57. Sun Microsystems, Inc. JXTA v2.0 Protocols Specification, `http://www.jxta.org`
58. Waldo, J.: The Jini architecture for network-centric computing. CACM 42(7), 76–82 (1999)

Maelstrom: Transparent Error Correction for Lambda Networks [*]

Mahesh Balakrishnan, Tudor Marian, Ken Birman, Hakim Weatherspoon, Einar Vollset
{mahesh, tudorm, ken, hweather, einar}@cs.cornell.edu
Cornell University, Ithaca, NY-14853

Abstract

The global network of datacenters is emerging as an important distributed systems paradigm — commodity clusters running high-performance applications, connected by high-speed 'lambda' networks across hundreds of milliseconds of network latency. Packet loss on long-haul networks can cripple application performance — a loss rate of 0.1% is sufficient to reduce TCP/IP throughput by an order of magnitude on a 1 Gbps link with 50ms latency. Maelstrom is an edge appliance that masks packet loss transparently and quickly from inter-cluster protocols, aggregating traffic for high-speed encoding and using a new Forward Error Correction scheme to handle bursty loss.

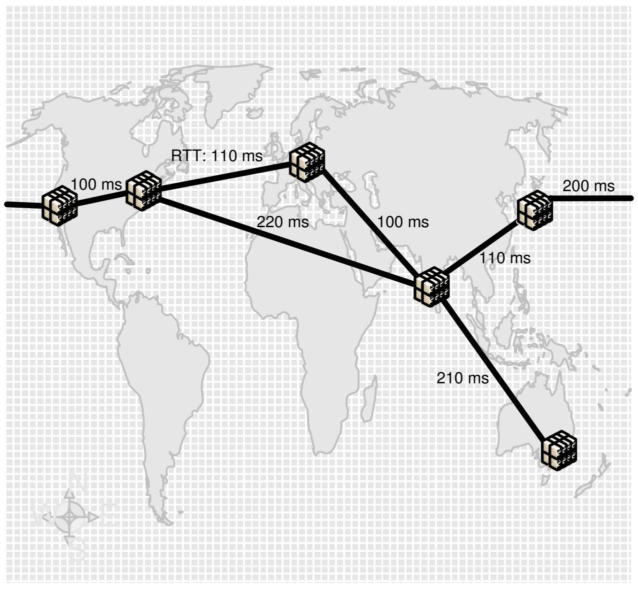


Figure 1: Example Lambda Network

1 Introduction

The emergence of commodity clusters and datacenters has enabled a new class of globally distributed high-performance applications that coordinate over vast geographical distances. For example, a financial firm's New York City datacenter may receive real-time updates from a stock exchange in Switzerland, conduct financial transactions with banks in Asia, cache data in London for locality and mirror it to Kansas for disaster-tolerance.

To interconnect these bandwidth-hungry datacenters across the globe, organizations are increasingly deploying private 'lambda' networks [35, 39]. Raw bandwidth is ubiquitous and cheaply available in the form of existing 'dark fiber'; however, running and maintaining high-quality *loss-free* networks over this fiber is difficult and expensive. Though high-capacity optical links are almost never congested, they drop packets for numerous reasons — dirty/degraded fiber [14], misconfigured/malfunctioning hardware [20,21] and switching contention [27], for example — and in different patterns, ranging from singleton drops to extended bursts [16,26].

Non-congestion loss has been observed on long-haul networks as well-maintained as Abilene/Internet2 and National LambdaRail [15, 16, 20, 21] — as has its crippling effect on commodity protocols, motivating research into loss-resistant data transfer protocols [13, 17, 25, 38, 43]. Conservative flow control mechanisms designed to deal with the systematic congestion of the commodity Internet react too sharply to ephemeral loss on over-provisioned links — a single packet loss in ten thousand is enough to reduce TCP/IP throughput to a third over a 50 ms gigabit link, and one in a thousand drops it by an order of magnitude. Real-time applications are impacted by the reliance of reliability mechanisms on acknowledgments and retransmissions, limiting the latency of packet recovery to at least the Round Trip Time (RTT) of the link; if delivery is sequenced, each lost packet acts as a virtual 'road-block' in the FIFO channel until it is recovered.

Deploying new loss-resistant protocols is not an alternative in corporate datacenters, where standardization is the key to low and predictable maintenance costs; neither is eliminating loss events on a network that could span thousands of miles. Accordingly, there is a need to

[*]This work was supported in part by grants from AFOSR, AFRL, NSF and Intel Corporation.

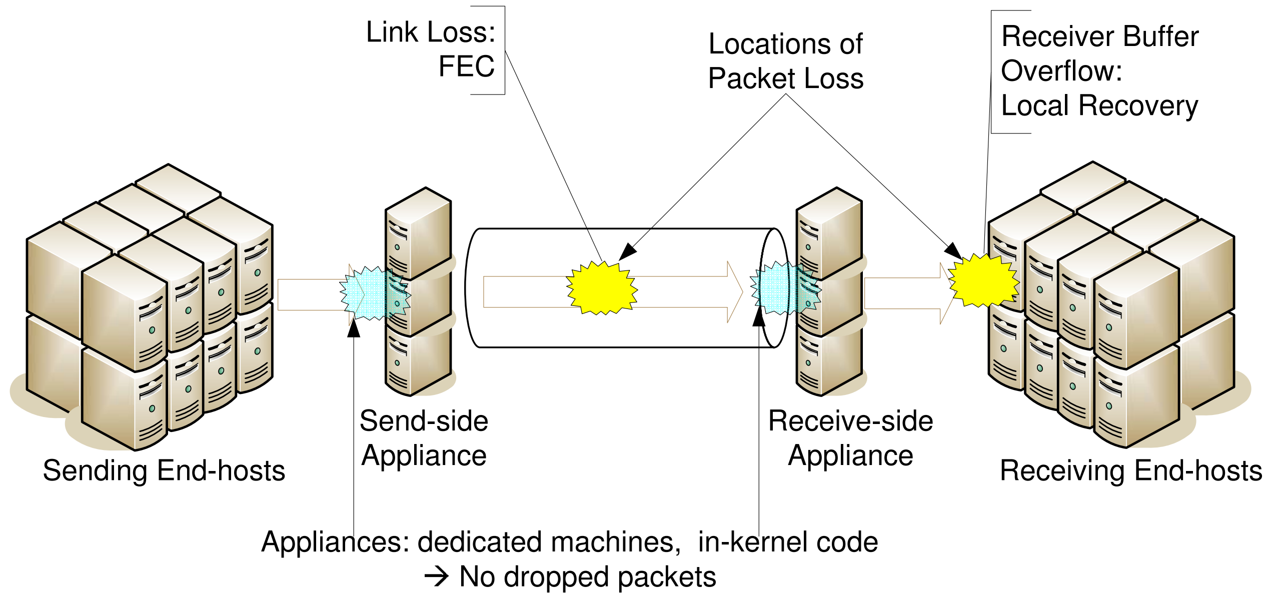


Figure 2: Maelstrom Communication Path

mask loss on the link, *rapidly* and *transparently*. Rapidly, because recovery delays for lost packets translate into dramatic reductions in application-level throughput; and transparently, because applications and OS networking stacks in commodity datacenters cannot be rewritten from scratch.

Forward Error Correction (FEC) is a promising solution for reliability over long-haul links [36] — packet recovery latency is independent of the RTT of the link. While FEC codes have been used for decades within link-level hardware solutions, faster commodity processors have enabled packet-level FEC at end-hosts [18, 37]. End-to-end FEC is very attractive for inter-datacenter communication: it's inexpensive, easy to deploy and customize, and does not require specialized equipment in the network linking the datacenters. However, end-host FEC has two major issues — First, it's not transparent, requiring modification of the end-host application/OS. Second, it's not necessarily rapid; FEC works best over high, stable traffic rates and performs poorly if the data rate in the channel is low and sporadic [6], as in a single end-to-end channel.

In this paper, we present the Maelstrom Error Correction appliance — a rack of proxies residing between a datacenter and its WAN link (see Figure 2). Maelstrom encodes FEC packets over traffic flowing through it and routes them to a corresponding appliance at the destination datacenter, which decodes them and recovers lost data. Maelstrom is completely transparent — it does not require modification of end-host software and is agnostic to the network connecting the datacenter. Also, it eliminates the dependence of FEC recovery latency on the data rate in any single node-to-node channel by encoding over the *aggregated* traffic leaving the datacenter. Finally, Maelstrom uses a new encoding scheme called *layered interleaving*, designed especially for time-sensitive packet recovery in the presence of bursty loss.

The contributions of this paper are as follows:

- We explore end-to-end FEC for long-distance communication between datacenters, and argue that the rate sensitivity of FEC codes and the opacity of their implementations present major obstacles to their usage.

- We propose Maelstrom, a gateway appliance that transparently aggregates traffic and encodes over the resulting high-rate stream.

- We describe *layered interleaving*, a new FEC scheme used by Maelstrom where for constant encoding overhead the latency of packet recovery degrades gracefully as losses get burstier.

- We discuss implementation considerations. We built two versions of Maelstrom; one runs in user mode, while the other runs within the Linux kernel.

- We evaluate Maelstrom on Emulab [45] and show that it provides near lossless TCP/IP throughput and latency over lossy links, and recovers packets with latency independent of the RTT of the link and the rate in any single channel.

2 Model

Our focus is on pairs of geographically distant datacenters that coordinate with each other in real-time. This has long been a critical distributed computing paradigm in application domains such as finance and aerospace; however, similar requirements are arising across the board as globalized enterprises rely on networks for high-speed communication and collaboration.

Traffic Model: The most general case of inter-cluster communication is one where any node in one cluster can communicate with any node in the other cluster. We make no assumptions about the type of traffic flowing through the link; mission-critical applications could send dynamically generated real-time data such as stock quotes, financial transactions and battleground location updates, while enterprise applications could send VoIP streams, ssh sessions and synchronous file updates between offices.

Loss Model: Packet loss typically occurs at two points in an end-to-end communication path between two datacenters, as shown in Figure 2 — in the wide-area network connecting them and at the receiving end-hosts. Loss in the lambda link can occur for many reasons, as stated previously: transient congestion, dirty or degraded fiber, malfunctioning or misconfigured equipment, low receiver power and burst switching contention are some reasons [14, 20, 21, 23, 27]. Loss can also occur at receiving end-hosts within the destination datacenter; these are usually cheap commodity machines prone to temporary overloads that cause packets to be dropped by the kernel in bursts [6] — this loss mode occurs with UDP-based traffic but not with TCP/IP, which advertises receiver windows to prevent end-host buffer overflows.

What are typical loss rates on long-distance optical networks? One source of information is TeraGrid [5], an optical network interconnecting major supercomputing sites in the US. TeraGrid has a monitoring framework within which ten sites periodically send each other 1 Gbps streams of UDP packets and measure the resulting loss rate [3]. Each site measures the loss rate to every other site once an hour, resulting in a total of 90 loss rate measurements collected across the network every hour. Between Nov 1, 2007 and Jan 25, 2007, 24% of all such measurements were over 0.01% and a surprising 14% of them were over 0.1%. After eliminating a single site (Indiana University) that dropped incoming packets steadily at a rate of 0.44%, 14% of the remainder were over 0.01% and 3% were over 0.1%.

These numbers reflect the loss rate experienced for UDP traffic on an end-to-end path and may not generalize to TCP packets. Also, we do not know if packets were dropped within the optical network or at intermediate devices within either datacenter, though it's unlikely that they were dropped at the end-hosts; many of the measurements lost just one or two packets whereas kernel/NIC losses are known to be bursty [6]. Further, loss occurred on paths where levels of optical link utilization (determined by 20-second moving averages) were consistently lower than 20%, ruling out congestion as a possible cause, a conclusion supported by dialogue with the network administrators [44].

Other data-points are provided by the back-bone networks of Tier-1 ISPs. Global Crossing reports average loss rates between 0.01% and 0.03% on four of its six inter-regional long-haul links for the month of December 2007 [1]. Qwest reports loss rates of 0.01% and 0.02% in either direction on its trans-pacific link for the same month [2]. We expect privately managed lambdas to exhibit higher loss rates due to the inherent trade-off between fiber/equipment quality and cost [10], as well as the difficulty of performing routine maintenance on long-distance links. Consequently, we model end-to-end paths as dropping packets at rates of 0.01% to 1%, to capture a wide range of deployed networks.

3 Existing Reliability Options

TCP/IP is the default reliable communication option for contemporary networked applications, with deep, exclusive embeddings in commodity operating systems and networking APIs. Consequently, most applications requiring reliable communication over any form of network use TCP/IP.

3.1 The problem with commodity TCP/IP

ACK/Retransmit + Sequencing: Conventional TCP/IP uses positive acknowledgments and retransmissions to ensure reliability — the sender buffers packets until their receipt is acknowledged by the receiver, and resends if an acknowledgment is not received within some time period. Hence, a lost packet is received in the form of a retransmission that arrives no earlier than 1.5 RTTs after the original send event. The sender has to buffer each packet until it's acknowledged, which takes 1 RTT in lossless operation, and it has to perform additional work to retransmit the packet if it does not receive the acknowledgment. Also, any packets that arrive with higher sequence numbers than that of a lost packet must be queued while the receiver waits for the lost packet to arrive.

Consider a high-throughput financial banking application running in a datacenter in New York City, sending updates to a sister site in Switzerland. The RTT value between these two centers is typically 100 milliseconds; i.e., in the case of a lost packet, all packets received within the 150 milliseconds between the original packet send and the

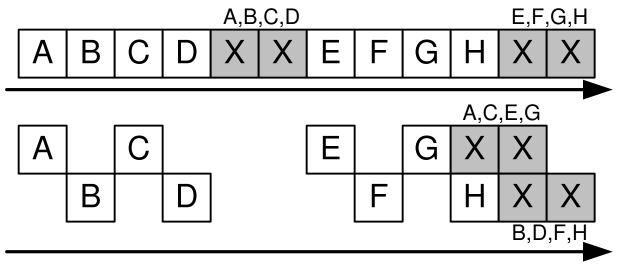


Figure 3: Interleaving with index 2: separate encoding for odd and even packets

receipt of its retransmission have to be buffered at the receiver.

Notice that for this commonplace scenario, the loss of a single packet stops all traffic in the channel to the application for a seventh of a second; a sequence of such blocks can have devastating effect on a high-throughput system where every spare cycle counts. Further, in applications with many fine-grained components, a lost packet can potentially trigger a butterfly effect of missed deadlines along a distributed workflow. During high-activity periods — market crashes at stock exchanges, Christmas sales at online stores, winter storms at air-traffic control centers — overloaded networks and end-hosts can exhibit continuous packet loss, with each lost packet driving the system further and further out of sync with respect to its real-world deadlines.

Sensitive Flow Control: TCP/IP is unable to distinguish between ephemeral loss modes — due to transient congestion, switching errors, or dirty fiber — and persistent congestion. The loss of one packet out of ten thousand is sufficient to reduce TCP/IP throughput to a third of its lossless maximum; if one packet is lost out of a thousand, throughput collapses to a thirtieth of the maximum.

3.2 The Case For (and Against) FEC

FEC encoders are typically parameterized with an (r, c) tuple — for each outgoing sequence of r data packets, a total of $r + c$ data and error correction packets are sent over the channel [1]. Significantly, redundancy information cannot be generated and sent until all r data packets are available for sending. Consequently, the latency of packet recovery is determined by the rate at which the sender transmits data. Generating error correction packets from less than r data packets at the sender is not a viable option — even though the data rate in this channel is low, the receiver and/or network could be operating at near full capacity with data from other senders.

FEC is also very susceptible to bursty losses [34]. *Interleaving* [32] is a standard encoding technique used to combat bursty loss, where error correction packets are generated from alternate disjoint sub-streams of data rather than from consecutive packets. For example, with an interleave index of 3, the encoder would create correction packets separately from three disjoint sub-streams: the first containing data packets numbered $(0, 3, 6...(r-1)*3)$, the second with data packets numbered $(1, 4, 7...(r-1)*3+1)$, and the third with data packets numbered $(2, 5, 8, ...(r-1)*3+2)$. Interleaving adds burst tolerance to FEC but exacerbates its sensitivity to sending rate — with an interleave index of i and an encoding rate of (r, c), the sender would have to wait for $i*(r-1)+1$ packets before sending any redundancy information.

These two obstacles to using FEC in time-sensitive settings — rate sensitivity and burst susceptibility — are interlinked through the tuning knobs: an interleave of i and a rate of (r, c) provides tolerance to a burst of up to $c*i$ consecutive packets. Consequently, the burst tolerance of an FEC code can be changed by modulating either the c or the i parameters. Increasing c enhances burst tolerance at the cost of network and encoding overhead, potentially worsening the packet loss experienced and reducing throughput. In contrast, increasing i trades off recovery latency for better burst tolerance without adding overhead — as mentioned, for higher values of i, the encoder has to wait for more data packets to be transmitted before it can send error correction packets.

Importantly, once the FEC encoding is parameterized with a rate and an interleave to tolerate a certain burst length B (for example, $r = 5$, $c = 2$ and $i = 10$ to tolerate a burst of length $2*10 = 20$), all losses occurring in bursts of size less than or equal to B are recovered with the same latency — and this latency depends on the i parameter. Ideally, we'd like to parameterize the encoding to tolerate a maximum burst length and then have recovery latency depend on the actual burstiness of the loss. At the same time, we would like the encoding to have a constant rate for network provisioning and stability. Accordingly, an FEC scheme is required where latency of recovery degrades gracefully as losses get burstier, even as the encoding overhead stays constant.

4 Maelstrom Design and Implementation

We describe the Maelstrom appliance as a single machine — later, we will show how more machines can be added to the appliance to balance encoding load and scale to multiple gigabits per second of traffic.

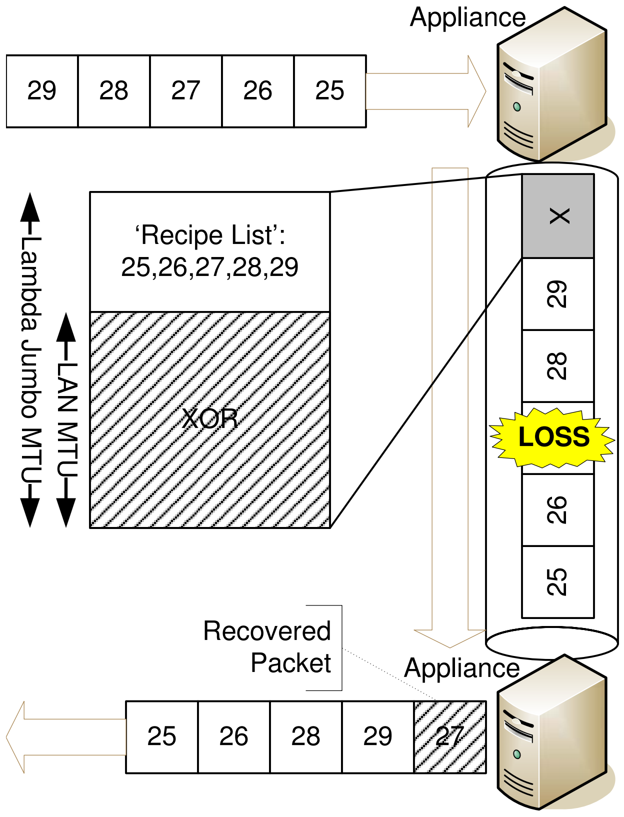


Figure 4: Basic Maelstrom mechanism: repair packets are injected into stream transparently

4.1 Basic Mechanism

The basic operation of Maelstrom is shown in Figure 4 — at the send-side datacenter, it intercepts outgoing data packets and routes them to the destination datacenter, generating and injecting FEC repair packets into the stream in their wake. A repair packet consists of a 'recipe' list of data packet identifiers and FEC information generated from these packets; in the example in Figure 4, this information is a simple XOR. The size of the XOR is equal to the MTU of the datacenter network, and to avoid fragmentation of repair packets we require that the MTU of the long-haul network be set to a slightly larger value. This requirement is usually satisfied in practical deployments, since gigabit links very often use 'Jumbo' frames of up to 9000 bytes [19] while LAN networks have standard MTUs of 1500 bytes.

At the receiving datacenter, the appliance examines incoming repair packets and uses them to recover missing data packets. On recovery, the data packet is injected transparently into the stream to the receiving end-host. Recovered data packets will typically arrive out-of-order, but this behavior is expected by communication stacks designed for the commodity Internet.

4.2 Flow Control

While relaying TCP/IP data, Maelstrom has two flow control modes: *end-to-end* and *split*. With end-to-end flow control, the appliance routes packets through without modification, allowing flow-control between the end-hosts. In *split* mode, the appliance acts as a TCP/IP endpoint, terminating connections and sending back ACKs immediately before relaying data on appliance-to-appliance flows; this is particularly useful for applications with short-lived flows that need to ramp up throughput quickly and avoid the slow-start effects of TCP/IP on a long link. The performance advantages of splitting long-distance connections into multiple hops are well known [7] and orthogonal to this work; we are primarily interested in isolating the impact of rapid and transparent recovery of lost packets by Maelstrom on TCP/IP, rather than the buffering and slow-start avoidance benefits of generic performance-enhancing proxies. In the remainder of the paper, we describe Maelstrom with end-to-end flow control.

Is Maelstrom TCP-Friendly? While Maelstrom respects end-to-end flow control connections (or splits them and implements its own proxy-to-proxy flow control as described above), it is not designed for routinely congested networks; the addition of FEC under TCP/IP flow control allows it to steal bandwidth from other competing flows running without FEC in the link, though maintaining fairness versus similarly FEC-enhanced flows [30]. However, friendliness with conventional TCP/IP flows is not a primary protocol design goal on over-provisioned multi-gigabit links, which are often dedicated to specific high-value applications. We see evidence for this assertion in the routine use of parallel flows [38] and UDP 'blast' protocols [17, 43] both in commercial deployments and by researchers seeking to transfer large amounts of data over high-capacity academic networks.

4.3 Layered Interleaving

In layered interleaving, an FEC protocol with rate (r, c) is produced by running c multiple instances of an $(r, 1)$ FEC protocol simultaneously with increasing interleave indices $I = (i_0, i_1, i_2...i_{c-1})$. For example, if $r = 8$, $c = 3$ and $I = (i_0 = 1, i_1 = 10, i_2 = 100)$, three instances of an $(8, 1)$ protocol are executed: the first instance with interleave $i_0 = 1$, the second with interleave $i_1 = 10$ and the third with interleave $i_2 = 100$. An $(r, 1)$ FEC encoding is simply an XOR of the r data packets — hence, in layered interleaving each data packet is included in c XORs, each of which is generated at different interleaves from the original data stream. Choosing interleaves appropriately (as we shall describe shortly) ensures that the c XORs containing a data packet do not have any other

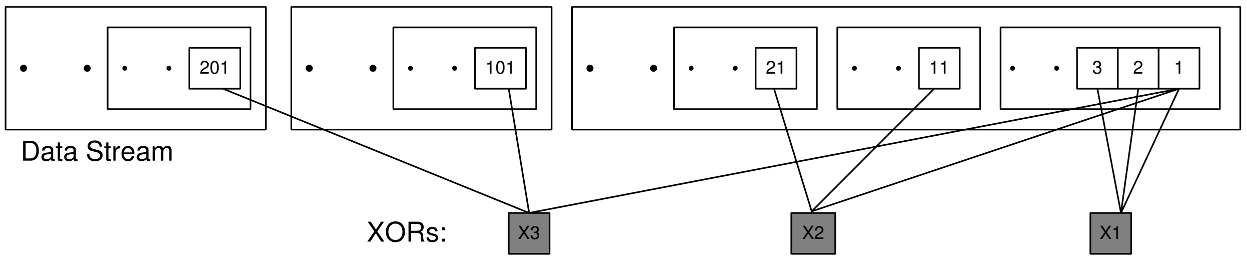


Figure 5: Layered Interleaving: $(r = 3, c = 3)$, $I = (1, 10, 100)$

data packet in common. The resulting protocol effectively has a rate of (r, c), with each XOR generated from r data packets and each data packet included in c XORs. Figure 5 illustrates layered interleaving for a $(r = 3, c = 3)$ encoding with $I = (1, 10, 100)$.

As mentioned previously, standard FEC schemes can be made resistant to a certain loss burst length at the cost of increased recovery latency for all lost packets, including smaller bursts and singleton drops. In contrast, layered interleaving provides graceful degradation in the face of bursty loss for constant encoding overhead — singleton random losses are recovered as quickly as possible, by XORs generated with an interleave of 1, and each successive layer of XORs generated at a higher interleave catches larger bursts missed by the previous layer.

The implementation of this algorithm is simple and shown in Figure 6 — at the send-side proxy, a set of repair bins is maintained for each layer, with i bins for a layer with interleave i. A repair bin consists of a partially constructed repair packet: an XOR and the 'recipe' list of identifiers of data packets that compose the XOR. Each intercepted data packet is added to each layer — where adding to a layer simply means choosing a repair bin from the layer's set, incrementally updating the XOR with the new data packet, and adding the data packet's header to the recipe list. A counter is incremented as each data packet arrives at the appliance, and choosing the repair bin from the layer's set is done by taking the modulo of the counter with the number of bins in each layer: for a layer with interleave 10, the xth intercepted packet is added to the $(x \bmod 10)$th bin. When a repair bin fills up — its recipe list contains r data packets — it 'fires': a repair packet is generated consisting of the XOR and the recipe list and is scheduled for sending, while the repair bin is re-initialized with an empty recipe list and blank XOR.

At the receive-side proxy, incoming repair packets are processed as follows: if all the data packets contained in the repair's recipe list have been received successfully, the repair packet is discarded. If the repair's recipe list contains a single missing data packet, recovery can occur immediately by combining the XOR in the repair with

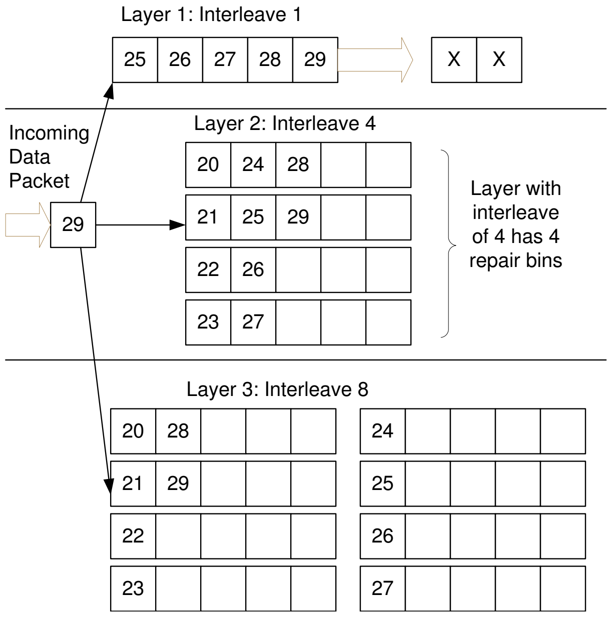


Figure 6: Layered Interleaving Implementation: $(r = 5, c = 3)$, $I = (1, 4, 8)$

the other successfully received data packets. If the repair contains multiple missing data packets, it cannot be used immediately for recovery — it is instead stored in a table that maps missing data packets to repair packets. Whenever a data packet is subsequently received or recovered, this table is checked to see if any XORs now have singleton losses due to the presence of the new packet and can be used for recovering other missing packets.

Importantly, XORs received from different layers interact to recover missing data packets, since an XOR received at a higher interleave can recover a packet that makes an earlier XOR at a lower interleave usable — hence, though layered interleaving is equivalent to c different $(r, 1)$ instances in terms of overhead and design, its recovery power is much higher and comes close to standard (r, c) algorithms.

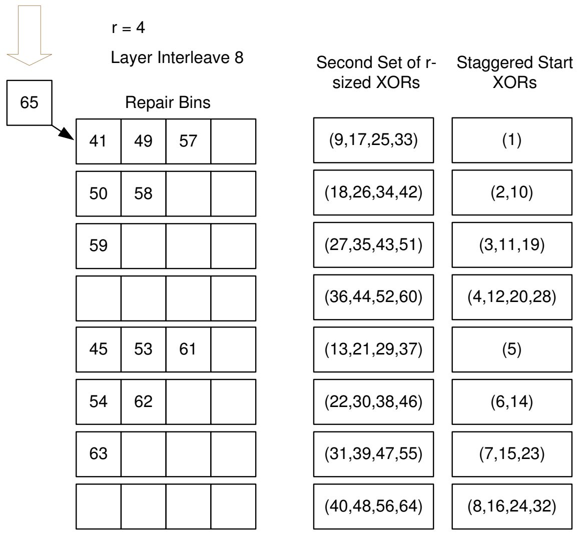


Figure 7: Staggered Start

4.3.1 Optimizations

Staggered Start for Rate-Limiting In the naive implementation of the layered interleaving algorithm, repair packets are transmitted as soon as repair bins fill and allow them to be constructed. Also, all the repair bins in a layer fill in quick succession; in Figure 6, the arrival of packets 36, 37, 38 and 39 will successively fill the four repair bins in layer 2. This behavior leads to a large number of repair packets being generated and sent within a short period of time, which results in undesirable overhead and traffic spikes; ideally, we would like to rate-limit transmissions of repair packets to one for every r data packets.

This problem is fixed by 'staggering' the starting sizes of the bins, analogous to the starting positions of runners in a sprint; the very first time bin number x in a layer of interleave i fires, it does so at size x mod r. For example, in Figure 6, the first repair bin in the second layer with interleave 4 would fire at size 1, the second would fire at size 2, and so on. Hence, for the first i data packets added to a layer with interleave i, exactly i/r fire immediately with just one packet in them; for the next i data packets added, exactly i/r fire immediately with two data packets in them, and so on until $r * i$ data packets have been added to the layer and all bins have fired exactly once. Subsequently, all bins fire at size r; however, now that they have been staggered at the start, only i/r fire for any i data packets. The outlined scheme works when i is greater than or equal to r, as is usually the case. If i is smaller than r, the bin with index x fires at $((x \bmod r) * r/i)$ — hence, for $r = 4$ and $i = 2$, the initial firing sizes would be 2 for the first bin and 4 for the second bin. If r and i are not integral multiples of each other, the rate-limiting still works but is slightly less effective due to rounding errors.

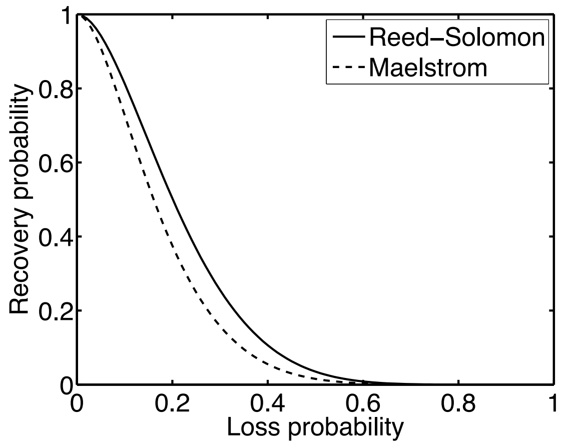


Figure 8: Comparison of Packet Recovery Probability: r=7, c=2

Delaying XORs In the naive implementation, repair packets are transmitted as soon as they are generated. This results in the repair packet leaving immediately after the last data packet that was added to it, which lowers burst tolerance — if the repair packet was generated at interleave i, the resulting protocol can tolerate a burst of i lost data packets excluding the repair, but the burst could swallow both the repair and the last data packet in it as they are not separated by the requisite interleave. The solution to this is simple — delay sending the repair packet generated by a repair bin until the next time a data packet is added to the now empty bin, which happens i packets later and introduces the required interleave between the repair packet and the last data packet included in it.

Notice that although transmitting the XOR immediately results in faster recovery, doing so also reduces the probability of a lost packet being recovered. This trade-off results in a minor control knob permitting us to balance speed against burst tolerance; our default configuration is to transmit the XOR immediately.

4.4 Back-of-the-Envelope Analysis

To start with, we note that no two repair packets generated at different interleaves i_1 and i_2 (such that $i_1 < i_2$) will have more than one data packet in common as long as the Least Common Multiple (LCM) of the interleaves is greater than $r * i_1$; pairings of repair bins in two different layers with interleaves i_1 and i_2 occur every $LCM(i_1, i_2)$ packets. Thus, a good rule of thumb is to select interleaves that are relatively prime to maximize their LCM, and also ensure that the larger interleave is greater than r.

Let us assume that packets are dropped with uniform, independent probability p. Given a lost data packet, what is the probability that we can recover it? We can recover a data packet if at least one of the c XORs containing it is received correctly and 'usable', i.e, all the other data packets in it have also been received correctly, the probability of which is simply $(1-p)^{r-1}$. The probability of a received XOR being unusable is the complement: $(1-(1-p)^{r-1})$.

Consequently, the probability x of a sent XOR being dropped or unusable is the sum of the probability that it was dropped and the probability that it was received and unusable: $x = p+(1-p)(1-(1-p)^{r-1}) = (1-(1-p)^r)$.

Since it is easy to ensure that no two XORs share more than one data packet, the usability probabilities of different XORs are independent. The probability of all the c XORs being dropped or unusable is x^c; hence, the probability of correctly receiving at least one usable XOR is $1 - x^c$. Consequently, the probability of recovering the lost data packet is $1 - x^c$, which expands to $1 - (1 - (1 - p)^r)^c$.

This closed-form formula only gives us a lower bound on the recovery probability, since the XOR usability formula does not factor in the probability of the other data packets in the XOR being dropped and *recovered*.

Now, we extend the analysis to bursty losses. If the lost data packet was part of a loss burst of size b, repair packets generated at interleaves less than b are dropped or useless with high probability, and we can discount them. The probability of recovering the data packet is then $1 - x^{c'}$, where c' is the number of XORs generated at interleaves greater than b. The formulae derived for XOR usability still hold, since packet losses with more than b intervening packets between them have independent probability; there is only correlation within the bursts, not between bursts.

How does this compare to traditional (r, c) codes such as Reed-Solomon [46]? In Reed-Solomon, c repair packets are generated and sent for every r data packets, and the correct delivery of any r of the $r + c$ packets transmitted is sufficient to reconstruct the original r data packets. Hence, given a lost data packet, we can recover it if at least r packets are received correctly in the encoding set of $r + c$ data and repair packets that the lost packet belongs to. Thus, the probability of recovering a lost packet is equivalent to the probability of losing $c - 1$ or less packets from the total $r + c$ packets. Since the number of other lost packets in the XOR is a random variable Y and has a binomial distribution with parameters $(r + c - 1)$ and p, the probability $P(Y \leq c - 1)$ is the summation $\sum_{z \leq c-1} P(Y = z)$. In Figure 8, we plot the recovery probability curves for Layered Interleaving and Reed-Solomon against uniformly random loss rate, for $(r = 7, c = 2)$ — note that the curves are very close to each other, especially in the loss range of interest between 0% and 10%.

4.5 Local Recovery for Receiver Loss

In the absence of intelligent flow control mechanisms like TCP/IP's receiver-window advertisements, inexpensive datacenter end-hosts can be easily overwhelmed and drop packets during traffic spikes or CPU-intensive maintenance tasks like garbage collection. Reliable application-level protocols layered over UDP — for reliable multicast [6] or high speed data transfer [17], for example — would ordinarily go back to the sender to retrieve the lost packet, even though it was dropped at the receiver after covering the entire geographical distance.

The Maelstrom proxy acts as a local packet cache, storing incoming packets for a short period of time and providing hooks that allow protocols to first query the cache to locate missing packets before sending retransmission requests back to the sender. Future versions of Maelstrom could potentially use knowledge of protocol internals to transparently intervene; for example, by intercepting and satisfying retransmission requests sent by the receiver in a NAK-based protocol, or by resending packets when acknowledgments are not observed within a certain time period in an ACK-based protocol.

4.6 Implementation Details

We initially implemented and evaluated Maelstrom as a user-space proxy. Performance turned out to be limited by copying and context-switching overheads, and we subsequently reimplemented the system as a module that runs within the Linux 2.6.20 kernel. At an encoding rate of $(8, 3)$, the experimental prototype of the kernel version reaches output speeds close to 1 gigabit per second of combined data and FEC traffic, limited only by the capacity of the outbound network card.

Of course, lambda networks are already reaching speeds of 40 gigabits, and higher speeds are a certainty down the road. To handle multi-gigabit loads, we envision Maelstrom as a small rack-style cluster of blade-servers, each acting as an individual proxy. Traffic would be distributed over such a rack by partitioning the address space of the remote datacenter and routing different segments of the space through distinct Maelstrom appliance pairs. In future work, we plan to experiment with such configurations, which would also permit us to explore fault-tolerance issues (if a Maelstrom blade fails, for example), and to support load-balancing schemes that might vary the IP address space partitioning dynamically to spread the encoding load over multiple machines. For this paper, however, we present the implementation and performance of a single-machine appliance.

The kernel implementation is a module for Linux 2.6.20 with hooks into the kernel packet filter [4]. Maelstrom proxies work in pairs, one on each side of the long haul link. Each proxy acts both as an ingress and egress router at the same time since they handle duplex traffic in the following manner:

- The egress router captures IP packets and creates redundant FEC packets. The original IP packets are routed through unaltered as they would have been originally; the redundant packets are then forwarded to the remote ingress router via a UDP channel.
- The ingress router captures and stores IP packets coming from the direction of the egress router. Upon receipt of a redundant packet, an IP packet is recovered if there is an opportunity to do so. Redundant packets that can be used at a later time are stored. If the redundant packet is useless it is immediately discarded. Upon recovery the IP packet is sent through a raw socket to its intended destination.

Using FEC requires that each data packet have a unique identifier that the receiver can use to keep track of received data packets and to identify missing data packets in a repair packet. If we had access to end-host stacks, we could have added a header to each packet with a unique sequence number [37]; however, we intercept traffic transparently and need to route it without modification or addition, for performance reasons. Consequently, we identify IP packets by a tuple consisting of the source and destination IP address, IP identification field, size of the IP header plus data, and a checksum over the IP data payload. The checksum over the payload is necessary since the IP identification field is only 16 bits long and a single pair of end-hosts communicating at high speeds will use the same identifier for different data packets within a fairly short interval unless the checksum is added to differentiate between them. Note that non-unique identifiers result in garbled recovery by Maelstrom, an event which will be caught by higher level checksums designed to deal with tranmission errors on commodity networks and hence does not have significant consequences unless it occurs frequently.

The kernel version of Maelstrom can generate up to a Gigabit per second of data and FEC traffic, with the input data rate depending on the encoding rate. In our experiments, we were able to saturate the outgoing card at rates as high as $(8, 4)$, with CPU overload occurring at $(8, 5)$ where each incoming data packet had to be XORed 5 times.

4.7 Buffering Requirements

At the receive-side proxy, incoming data packets are buffered so that they can be used in conjunction with XORs to recover missing data packets. Also, any received XOR that is missing more than one data packet is stored temporarily, in case all but one of the missing packets are received later or recovered through other XORs, allowing the recovery of the remaining missing packet from this XOR. In practice we stored data and XOR packets in double buffered red black trees — for 1500 byte packets and 1024 entries this occupies around 3 MB of memory.

At the send-side, the repair bins in the layered interleaving scheme store incrementally computed XORs and lists of data packet headers, without the data packet payloads, resulting in low storage overheads for each layer that rise linearly with the value of the interleave. The memory footprint for a long-running proxy was around 10 MB in our experiments.

4.8 Other Performance Enhancing Roles

Maelstrom appliances can optionally aggregate small subkilobyte packets from different flows into larger ones for better communication efficiency over the long-distance link. Additionally, in split flow control mode they can perform send-side buffering of in-flight data for multigigabyte flows that exceed the sending end-host's buffering capacity. Also, Maelstrom appliances can act as multicast forwarding nodes: appliances send multicast packets to each other across the long-distance link, and use IP Multicast [11] to spread them within their datacenters. Lastly, appliances can take on other existing roles in the datacenter, acting as security and VPN gateways and as conventional performance enhancing proxies (PEPs) [7].

5 Evaluation

We evaluated Maelstrom on the Emulab testbed at Utah [45]. For all the experiments, we used a 'dumbbell' topology of two clusters of nodes connected via routing nodes with a high-latency link in between them, designed to emulate the setup in Figure 2, and ran the proxy code on the routers. Figure 10 shows the performance of the kernel version at Gigabit speeds; the remainder of the graphs show the performance of the user-space version at slower speeds. To emulate the MTU difference between the long-haul link and the datacenter network (see Section 4.1) we set an MTU of 1200 bytes on the network connecting the end-hosts to the proxy and an MTU of 1500 bytes on the long-haul link between proxies; the only exception is Figure 10, where we maintained equal MTUs of 1500 bytes on both links.

5.1 Throughput Metrics

Figures 9 and 10 show that commodity TCP/IP throughput collapses in the presence of non-congestion loss, and that Maelstrom successfully masks loss and prevents this

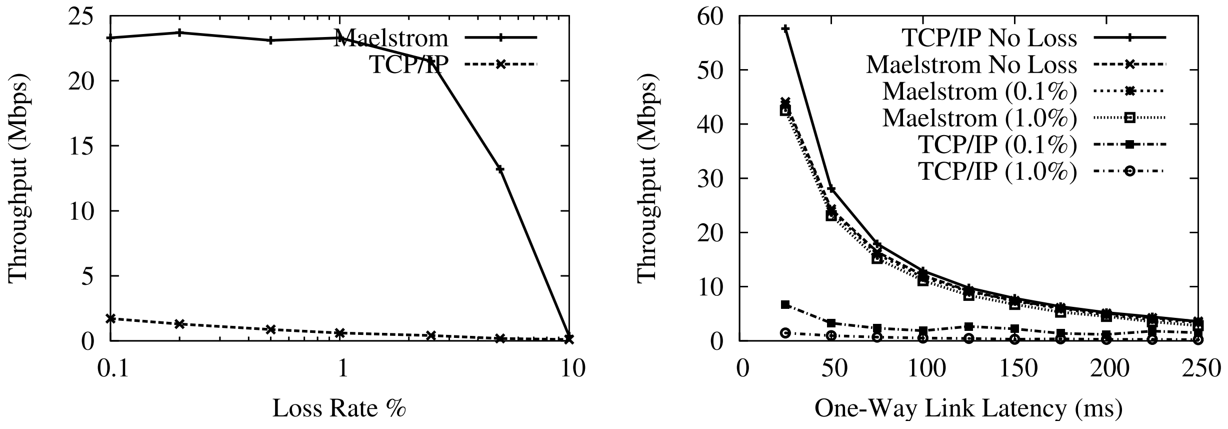


Figure 9: User-Space Throughput against (a) Loss Rate and (b) One-Way Latency

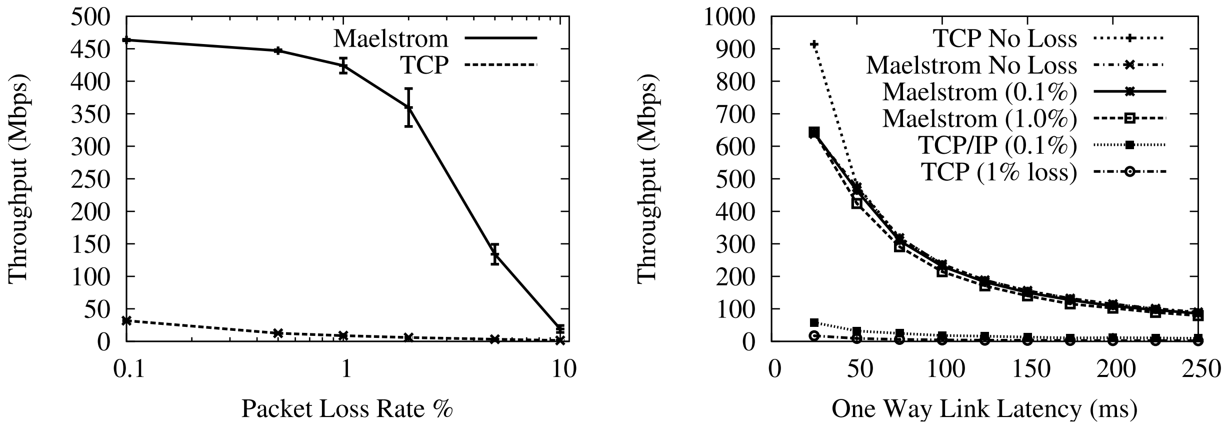


Figure 10: Kernel Throughput against (a) Loss Rate and (b) One-Way Latency

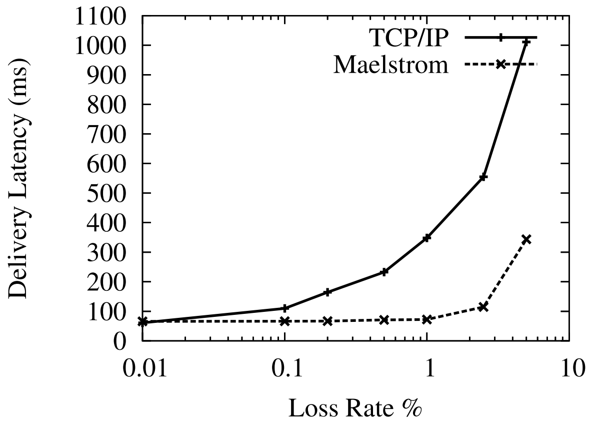
collapse from occurring. Figure 9 shows the performance of the user-space version on a 100 Mbps link and Figure 10 shows the kernel version on a 1 Gbps link. The experiment in each case involves running iperf [41] flows from one node to another across the long-distance link with and without intermediary Maelstrom proxies and measuring obtained throughput while varying loss rate (left graph on each figure) and one-way link latency (right graph). The error bars on the graphs to the left are standard errors of the throughput over ten runs; between each run, we flush TCP/IP's cache of tuning parameters to allow for repeatable results. The clients in the experiment are running TCP/IP Reno on a Linux 2.6.20 that performs autotuning. The Maelstrom parameters used are $r = 8$,$c = 3$, $I = (1, 20, 40)$.

The user-space version involved running a single 10 second iperf flow from one node to another with and without Maelstrom running on the routers and measuring throughput while varying the random loss rate on the link and the one-way latency. To test the kernel version at gigabit speeds, we ran eight parallel iperf flows from one node to another for 120 seconds. The curves obtained from the two versions are almost identical; we present both to show that the kernel version successfully scales up the performance of the user-space version to hundreds of megabits of traffic per second.

In Figures 9 (Left) and 10 (Left), we show how TCP/IP performance degrades on a 50ms link as the loss rate is increased from 0.01% to 10%. Maelstrom masks loss up to 2% without significant throughput degradation, with the kernel version achieving two orders of magnitude higher throughput that conventional TCP/IP at 1% loss.

The graphs on the right side of Figures 9 and 10 show TCP/IP throughput declining on a link of increasing length when subjected to uniform loss rates of 0.1% and 1%. The top line in the graphs is the performance of TCP/IP without loss and provides an upper bound for performance on the link. In both user-space and kernel versions, Maelstrom masks packet loss and tracks the lossless line closely, lagging only when the link latency is low and TCP/IP's throughput is very high.

To allow TCP/IP to attain very high speeds on the gi-



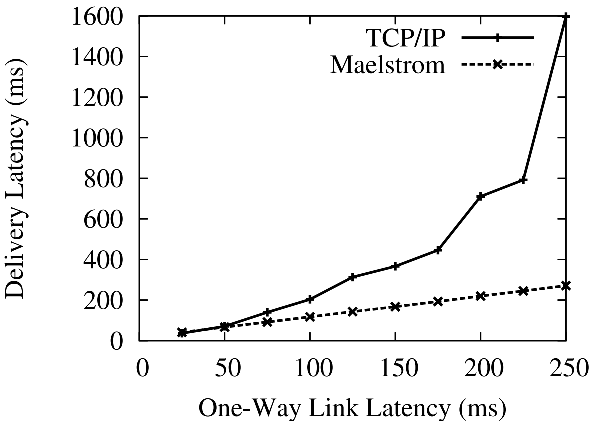
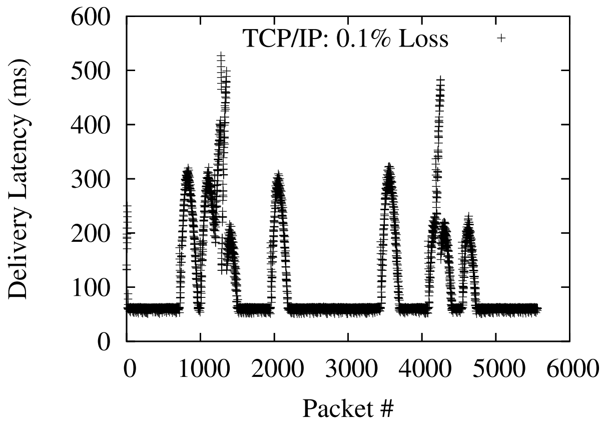


Figure 11: Per-Packet One-Way Delivery Latency against Loss Rate (Left) and Link Latency (Right)



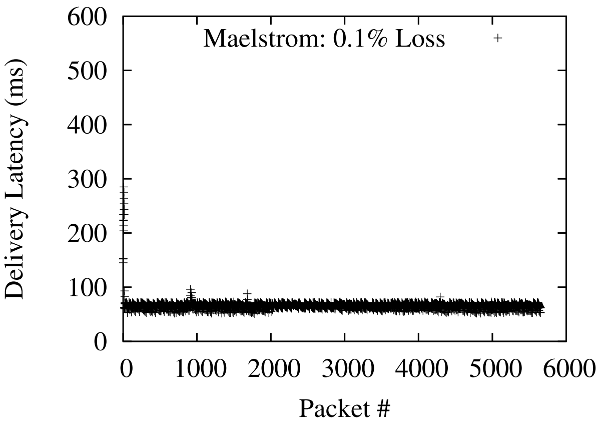


Figure 12: Packet delivery latencies

gabit link, we had to set the MTU of the entire path to be the maximum 1500 bytes, which meant that the long-haul link had the same MTU as the inter-cluster link. This resulted in the fragmentation of repair packets sent over UDP on the long-haul link into two IP packet fragments. Since the loss of a single fragment resulted in the loss of the repair, we observed a higher loss rate for repairs than for data packets. Consequently, we expect performance to be better on a network where the MTU of the long-haul link is truly larger than the MTU within each cluster.

5.2 Latency Metrics

To measure the latency effects of TCP/IP and Maelstrom, we ran a 0.1 Mbps stream between two nodes over a 100 Mbps link with 50 ms one-way latency, and simultaneously ran a 10 Mbps flow alongside on the same link to simulate a real-time stream combined with other inter-cluster traffic. Figure 11 (Left) shows the average delivery latency of 1KB application-level packets in the 0.1 Mbps stream, as loss rates go up.

Figure 11 (Right) shows the same scenario with a constant uniformly random loss rate of 0.1% and varying one-way latency. Maelstrom's delivery latency is almost exactly equal to the one-way latency on the link, whereas TCP/IP takes more than twice as long once one-way latencies go past 100 ms. Figure 12 plots delivery latency against message identifier; the spikes in latency are triggered by losses that lead to packets piling up at the receiver.

A key point is that we are plotting the delivery latency of all packets, not just lost ones. TCP/IP delays correctly received packets while waiting for missing packets sequenced earlier by the sender — the effect of this is shown in Figure 12, where single packet losses cause spikes in delivery latency that last for hundreds of packets. The low data rate in the flow of roughly 10 1KB packets per RTT makes TCP/IP flow control delays at the sender unlikely, given that the congestion control algorithm is Reno, which implements 'fast recovery' and halves the congestion window on packet loss rather than resetting it completely [22]. The Maelstrom configuration used is

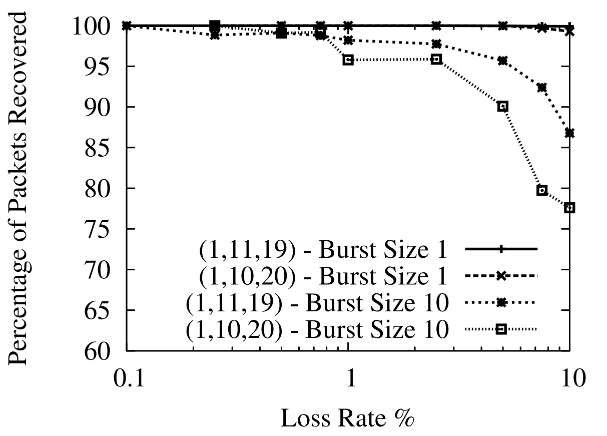


Figure 13: Relatively prime interleaves offer better performance

$r = 7, c = 2, I = (1, 10)$.

5.3 Layered Interleaving and Bursty Loss

Thus far we have shown how Maelstrom effectively hides loss from TCP/IP for packets dropped with uniform randomness. Now, we examine the performance of the layered interleaving algorithm, showing how different parameterizations handle bursty loss patterns. We use a loss model where packets are dropped in bursts of fixed length, allowing us to study the impact of burst length on performance. The link has a one-way latency of 50 ms and a loss rate of 0.1% (except in Figure 13, where it is varied), and a 10 Mbps flow of udp packets is sent over it.

In Figure 13 we show that our observation in Section 4.4 is correct for high loss rates — if the interleaves are relatively prime, performance improves substantially when loss rates are high and losses are bursty. The graph plots the percentage of lost packets successfully recovered on the y-axis against an x-axis of loss rates on a log scale. The Maelstrom configuration used is $r = 8, c = 3$ with interleaves of $(1, 10, 20)$ and $(1, 11, 19)$.

In Figure 14, we show the ability of layered interleaving to provide gracefully degrading performance in the face of bursty loss. On the top, we plot the percentage of lost packets successfully recovered against the length of loss bursts for two different sets of interleaves, and in the bottom graph we plot the average latency at which the packets were recovered. Recovery latency is defined as the difference between the eventual delivery time of the recovered packet and the one-way latency of the link (we confirmed that the Emulab link had almost no jitter on correctly delivered packets, making the one-way latency an accurate estimate of expected lossless delivery time). As expected, increasing the interleaves results in much higher recovery percentages at large burst sizes, but comes at the cost of higher recovery latency. For example, a $(1, 19, 41)$ set of interleaves catches almost all packets in an extended burst of 25 packets at an average latency of around 45 milliseconds, while repairing all random singleton losses within 2-3 milliseconds. The graphs also show recovery latency rising gracefully with the increase in loss burst length: the longer the burst, the longer it takes to recover the lost packets. The Maelstrom configuration used is $r = 8, c = 3$ with interleaves of $(1, 11, 19)$ and $(1, 19, 41)$.

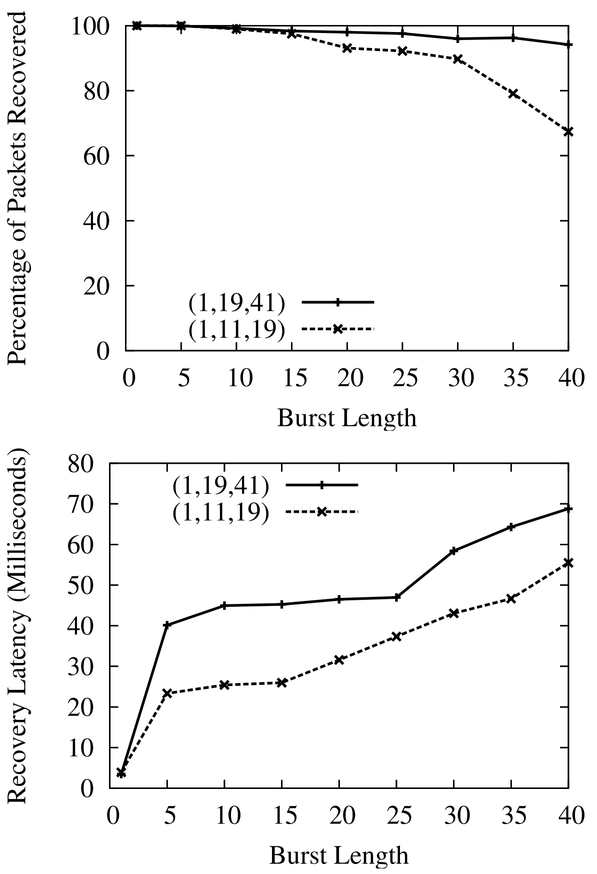


Figure 14: Layered Interleaving Recovery Percentage and Latency

In Figures 16 and 17 we show histograms of recovery latencies for the two interleave configurations under different burst lengths. The histograms confirm the trends described above: packet recoveries take longer from left to right as we increase loss burst length, and from top to bottom as we increase the interleave values.

Figure 15 illustrates the difference between a traditional FEC code and layered interleaving by plotting a 50-element moving average of recovery latencies for both codes. The channel is configured to lose singleton packets randomly at a loss rate of 0.1% and additionally lose long bursts of 30 packets at occasional intervals. Both codes

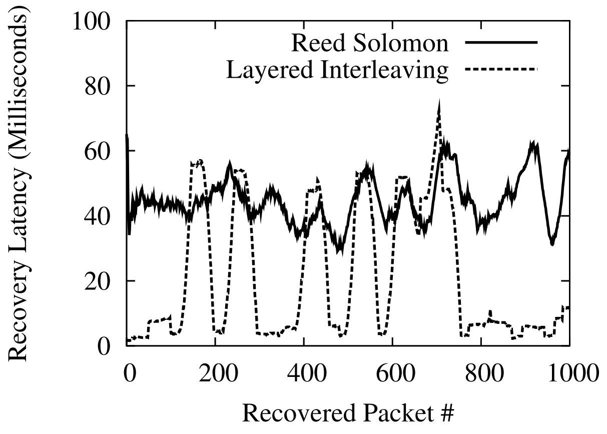


Figure 15: Reed-Solomon versus Layered Interleaving

are configured with $r = 8, c = 2$ and recover all lost packets — Reed-Solomon uses an interleave of 20 and layered interleaving uses interleaves of $(1, 40)$ and consequently both have a maximum tolerable burst length of 40 packets. We use a publicly available implementation of a Reed-Solomon code based on Vandermonde matrices, described in [36]; the code is plugged into Maelstrom instead of layered interleaving, showing that we can use new encodings within the same framework seamlessly. The Reed-Solomon code recovers all lost packets with roughly the same latency whereas layered interleaving recovers singleton losses almost immediately and exhibits latency spikes whenever the longer loss burst occurs.

6 Related Work

A significant body of work on application and TCP/IP performance over high-speed long-distance networks exists in the context of high-performance computing, grids and e-science. The use of parallel sockets for higher throughput in the face of non-congestion loss was proposed in PSockets [38]. A number of protocols have been suggested as replacements for TCP/IP in such settings — XCP [25], Tsunami [43], SABUL [13] and RBUDP [17] are a few — but all require modifications to end-hosts and/or the intervening network. Some approaches seek to differentiate between congestion and non-congestion losses [8].

Maelstrom is a transparent Performance Enhancing Proxy, as defined in RFC 3135 [7]; numerous implementations of PEPs exist for improving TCP performance on satellite [42] and wireless links [9], but we are not aware of any PEPs that use FEC to mask errors on long-haul optical links.

End-host software-based FEC for reliable communication was first explored by Rizzo [36, 37]. OverQOS [40] suggested the use of FEC for TCP/IP retransmissions over aggregated traffic within an overlay network in the commodity Internet. AFEC [34] uses FEC for real-time communication, modulating the rate of encoding adaptively. The use of end-host FEC under TCP/IP has been explored in [30].

A multitude of different FEC encodings exist in literature; they can broadly be categorized into optimal erasure codes and near-optimal erasure codes. The most well-known optimal code is Reed-Solomon, which we described previously as generating c repair packets from r source packets; any r of the resulting $r + c$ packets can be used to reconstruct the r source packets. Near-optimal codes such as Tornado and LT [29] trade-off encoding speed for large data sizes against a loss of optimality — the receiver needs to receive slightly more than r source or repair packets to regenerate the original r data packets. Near-optimal codes are extremely fast for encoding over large sets of data but not of significant importance for real-time settings, since optimal codes perform equally well with small data sizes. Of particular relevance are Growth Codes [24], which use multiple encoding rates for different overhead levels; in contrast, layered interleaving uses multiple interleaves for different burst resilience levels without modulating the encoding rate.

The effect of random losses on TCP/IP has been studied in depth by Lakshman [28]. Padhye's analytical model [33] provides a means to gauge the impact of packet loss on TCP/IP. While most published studies of packet loss are based on the commodity Internet rather than high-speed lambda links, Fraleigh et al. [12] study the Sprint backbone and make two observations that could be explained by non-congestion loss: a) links are rarely loaded at more than 50% of capacity and b) packet reordering events occur for some flows, possibly indicating packet loss followed by retransmissions.

7 Future Work

Scaling Maelstrom to multiple gigabits per second of traffic will require small rack-style clusters of tens of machines to distribute encoding load over; we need to design intelligent load-balancing and fail-over mechanisms for such a scheme. Additionally, we have described layered interleaving with fixed, pre-assigned parameters, and the next step in extending this protocol is to make it adaptive, changing interleaves and rate as loss patterns in the link change.

8 Conclusion

Modern distributed systems are compelled by real-world imperatives to coordinate across datacenters separated by

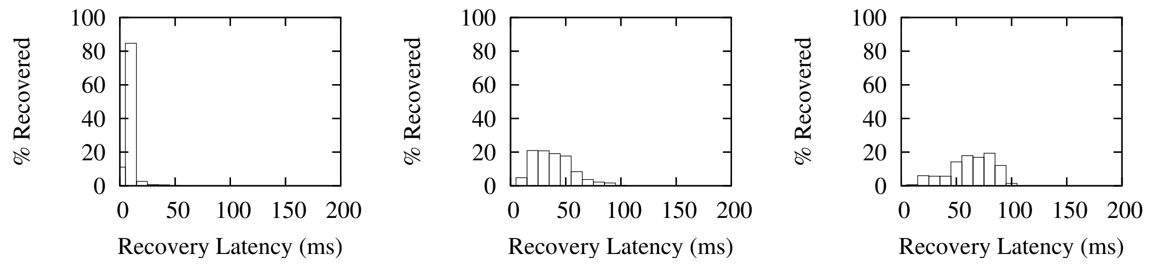


Figure 16: Latency Histograms for I=(1,11,19) — Burst Sizes 1 (Left), 20 (Middle) and 40 (Right)

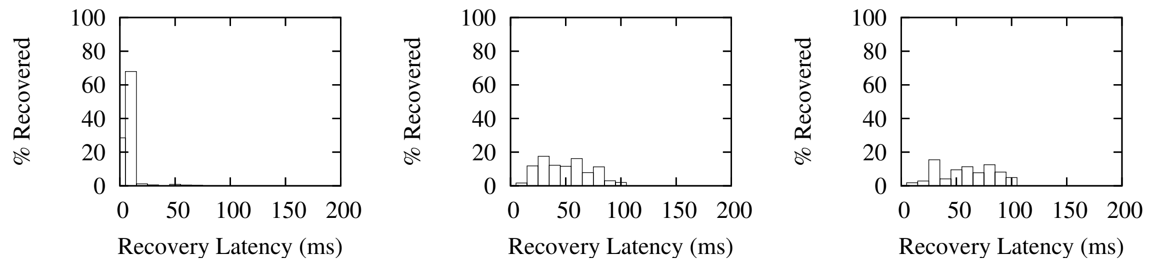


Figure 17: Latency Histograms for I=(1,19,41) — Burst Sizes 1 (Left), 20 (Middle) and 40 (Right)

thousands of miles. Packet loss cripples the performance of such systems, and reliability and flow-control protocols designed for LANs and/or the commodity Internet fail to achieve optimal performance on the high-speed long-haul 'lambda' networks linking datacenters. Deploying new protocols is not an option for commodity clusters where standardization is critical for cost mitigation. Maelstrom is an edge appliance that uses Forward Error Correction to mask packet loss from end-to-end protocols, improving TCP/IP throughput and latency by orders of magnitude when loss occurs. Maelstrom is easy to install and deploy, and is completely transparent to applications and protocols — literally providing reliability in an inexpensive box.

Acknowledgments

We would like to thank our shepherd Robert Morris and the other reviewers for extensive comments that significantly shaped the final version of the paper. Danny Dolev, Lakshmi Ganesh, T. V. Lakshman, Robbert van Renesse, Yee Jiun Song, Vidhyashankar Venkataraman and Vivek Vishnumurthy provided useful comments. Tom Boures provided valuable insight into the quality of existing fiber links, Stanislav Shalunov provided information on loss rates on Internet2, and Paul Wefel gave us access to TeraGrid loss measurements.

Notes

[1]Rateless codes (e.g, LT codes [29]) are increasingly popular and used for applications such as efficiently distributing bulk data [31] — however, it is not obvious that these have utility in real-time communication.

References

[1] Global crossing current network performance. `http://www.globalcrossing.com/network/network_performance_current.aspx`. Last Accessed Feb, 2008.

[2] Qwest ip network statistics. `http://stat.qwest.net/statqwest/statistics_tp.jsp`. Last Accessed Feb, 2008.

[3] Teragrid udp performance. `network.teragrid.org/tgperf/udp/`. Last Accessed Feb, 2008.

[4] Netfilter: firewalling, nat and packet mangling for linux. `http://www.netfilter.org/`, 1999.

[5] Teragrid. `www.teragrid.org`, 2008.

[6] M. Balakrishnan, K. Birman, A. Phanishayee, and S. Pleisch. Ricochet: Lateral error correction for time-critical multicast. In *NSDI 2007: Fourth Usenix Symposium on Networked Systems Design and Implementation*, 2007.

[7] J. Border, M. Kojo, J. Griner, G. Montenegro, and Z. Shelby. Performance Enhancing Proxies Intended to Mitigate Link-Related Degradations. *Internet RFC3135, June*, 2001.

[8] S. Bregni, D. Caratti, and F. Martignon. Enhanced loss differentiation algorithms for use in TCP sources over heterogeneous wireless networks. In *GLOBECOM 2003: IEEE Global Telecommunications Conference*, 2003.

[9] R. Chakravorty, S. Katti, I. Pratt, and J. Crowcroft. Flow aggregation for enhanced tcp over wide area wireless. In *INFOCOM*, 2003.

[10] D. Comer, Vice President of Research and T. Boures, Senior Engineer. Cisco systems, inc. *Private Communication.*, October 2007.

[11] S. E. Deering and D. R. Cheriton. Multicast routing in datagram internetworks and extended lans. *ACM Trans. Comput. Syst.*, 8(2):85–110, 1990.

[12] C. Fraleigh, S. Moon, B. Lyles, C. Cotton, M. Khan, D. Moll, R. Rockell, T. Seely, and S. Diot. Packet-level traffic measurements from the Sprint IP backbone. *IEEE Network*, 17(6):6–16, 2003.

[13] Y. Gu and R. Grossman. SABUL: A Transport Protocol for Grid Computing. *Journal of Grid Computing*, 1(4):377–386, 2003.

[14] R. Habel, K. Roberts, A. Solheim, and J. Harley. Optical domain performance monitoring. *Optical Fiber Communication Conference*, 2000.

[15] T. Hacker, B. Athey, and B. Noble. The end-to-end performance effects of parallel TCP sockets on a lossy wide-area network. In *IPDPS*, 2002.

[16] T. J. Hacker, B. D. Noble, and B. D. Athey. The effects of systemic packet loss on aggregate tcp flows. In *Supercomputing '02: Proceedings of the 2002 ACM/IEEE conference on Supercomputing*, 2002.

[17] E. He, J. Leigh, O. Yu, and T. Defanti. Reliable Blast UDP: predictable high performance bulk data transfer. *IEEE International Conference on Cluster Computing*, 2002.

[18] C. Huitema. The case for packet level fec. In *PfHSN '96: Proceedings of the TC6 WG6.1/6.4 Fifth International Workshop on Protocols for High-Speed Networks V*, pages 109–120, London, UK, UK, 1997. Chapman & Hall, Ltd.

[19] J. Hurwitz and W. Feng. End-to-end performance of 10-gigabit Ethernet on commodity systems. *Micro, IEEE*, 24(1):10–22, 2004.

[20] Internet2. End-to-end performance initiative: Hey! where did my performance go? - rate limiting rears its ugly head. `http://e2epi.internet2.edu/case-studies/UMich/index.html`.

[21] Internet2. End-to-end performance initiative: When 99% isn't quite enough - educause bad connection. `http://e2epi.internet2.edu/case-studies/EDUCAUSE/index.html`.

[22] V. Jacobson. Modified TCP Congestion Avoidance Algorithm. *Message to end2end-interest mailing list, April*, 1990.

[23] L. James, A. Moore, M. Glick, and J. Bulpin. Physical Layer Impact upon Packet Errors. *Passive and Active Measurement Workshop (PAM 2006)*, 2006.

[24] A. Kamra, J. Feldman, V. Misra, and D. Rubenstein. Growth codes: Maximizing sensor network data persistence. In *Proceedings of ACM Sigcomm*, Pisa, Italy, September 2006.

[25] D. Katabi, M. Handley, and C. Rohrs. Congestion control for high bandwidth-delay product networks. *Proceedings of the 2002 conference on Applications, technologies, architectures, and protocols for computer communications*, pages 89–102, 2002.

[26] D. Kilper, R. Bach, D. Blumenthal, D. Einstein, T. Landolsi, L. Ostar, M. Preiss, and A. Willner. Optical Performance Monitoring. *Journal of Lightwave Technology*, 22(1):294–304, 2004.

[27] A. Kimsas, H. Øverby, S. Bjornstad, and V. L. Tuft. A cross layer study of packet loss in all-optical networks. In *AICT/ICIW*, page 65, 2006.

[28] T. Lakshman and U. Madhow. The performance of TCP/IP for networks with high bandwidth-delay products and random loss. *IEEE/ACM Transactions on Networking (TON)*, 5(3):336–350, 1997.

[29] M. Luby. LT codes. *The 43rd Annual IEEE Symposium on Foundations of Computer Science*, 2002.

[30] H. Lundqvist and G. Karlsson. TCP with End-to-End Forward Error Correction. *International Zurich Seminar on Communications (IZS 2004)*, 2004.

[31] P. Maymounkov and D. Mazieres. Rateless codes and big downloads. *IPTPS03*, 2003.

[32] J. Nonnenmacher, E. Biersack, and D. Towsley. Parity-based loss recovery for reliable multicast transmission. In *Proceedings of the ACM SIGCOMM '97 conference*, pages 289–300, New York, NY, USA, 1997. ACM Press.

[33] J. Padhye, V. Firoiu, D. Towsley, and J. Kurose. Modeling tcp throughput: a simple model and its empirical validation. *SIGCOMM Comput. Commun. Rev.*, 28(4):303–314, 1998.

[34] K. Park and W. Wang. AFEC: an adaptive forward error correction protocol for end-to-endtransport of real-time traffic. *Computer Communications and Networks, 1998. Proceedings. 7th International Conference on*, pages 196–205, 1998.

[35] M. Reardon. Dark fiber: Businesses see the light. `http://www.news.com/Dark-fiber-Businesses-see-the-light/2100-1037_3-5557910.html?part=rss&tag=5557910&subj=news.1037.5`, 2005. Last Accessed Feb, 2008.

[36] L. Rizzo. Effective erasure codes for reliable computer communication protocols. *SIGCOMM Comput. Commun. Rev.*, 27(2):24–36, 1997.

[37] L. Rizzo. On the feasibility of software FEC. *Univ. di Pisa, Italy, January*, 1997.

[38] H. Sivakumar, S. Bailey, and R. L. Grossman. Psockets: the case for application-level network striping for data intensive applications using high speed wide area networks. In *Supercomputing '00: Proceedings of the 2000 ACM/IEEE conference on Supercomputing (CDROM)*, page 37, Washington, DC, USA, 2000. IEEE Computer Society.

[39] Slashdot.com. Google's secret plans for all that dark fiber. `http://slashdot.org/articles/05/11/20/1514244.shtml`, 2005.

[40] L. Subramanian, I. Stoica, H. Balakrishnan, and R. H. Katz. Overqos: An overlay based architecture for enhancing internet qos. In *NSDI 04: First Usenix Symposium on Networked Systems Design and Implementation*, 2004.

[41] A. Tirumala, F. Qin, J. Dugan, J. Ferguson, and K. Gibbs. Iperf-The TCP/UDP bandwidth measurement tool. *`http://dast.nlanr.net/Projects/Iperf`*, 2004.

[42] D. Velenis, D. Kalogeras, and B. Maglaris. SaTPEP: a TCP Performance Enhancing Proxy for Satellite Links. *Proceedings of the Second International IFIP-TC6 Networking Conference on Networking Technologies, Services, and Protocols; Performance of Computer and Communication Networks; and Mobile and Wireless Communications*, pages 1233–1238, 2002.

[43] S. Wallace et al. Tsunami File Transfer Protocol. In *PFLDNet 2003: First Int. Workshop on Protocols for Fast Long-Distance Networks*, 2003.

[44] P. Wefel, Network Engineer. The University of Illinois' National Center for Supercomputing Applications (NCSA). *Private Communication.*, Feb 2008.

[45] B. White, J. Lepreau, L. Stoller, R. Ricci, S. Guruprasad, M. Newbold, M. Hibler, C. Barb, and A. Joglekar. An integrated experimental environment for distributed systems and networks. In *Proc. of the Fifth Symposium on Operating Systems Design and Implementation*, Boston, MA, 2002.

[46] S. Wicker and V. Bhargava. *Reed-Solomon Codes and Their Applications*. John Wiley & Sons, Inc. New York, NY, USA, 1999.

Tempest: Soft State Replication in the Service Tier*

Tudor Marian, Mahesh Balakrishnan, Ken Birman, Robbert van Renesse
Department of Computer Science
Cornell University, Ithaca, NY 14853
{tudorm,mahesh,ken,rvr}@cs.cornell.edu

Abstract

Soft state in the middle tier is key to enabling scalable and responsive three tier service architectures. While soft-state can be reconstructed upon failure, replicating it across multiple service instances is critical for rapid fail-over and high availability. Current techniques for storing and managing replicated soft state require mapping data structures to different abstractions such as database records, which can be difficult and introduce inefficiencies. Tempest is a system that provides programmers with data structures that look very similar to conventional Java Collections but are automatically replicated. We evaluate Tempest against alternatives such as in-memory databases and we show that Tempest does scale well in real world service architectures.

1 Introduction

Service-Oriented Architectures (SOAs) have emerged as the paradigm of choice for structuring large datacenter-hosted systems. Most contemporary large-scale applications are built as SOAs: online stores, search engines, enterprise software and financial infrastructure are some examples. The canonical design for such systems is a three-tier architecture: a first tier load-balancing proxies sends requests to a second tier of state-less service logic which in turn accesses and updates a third tier of durable databases or filesystems.

Soft state in the service tier is key to building highly responsive and scalable SOAs. Soft state is characterized as data that does not have to be stored durably and can be reconstructed at some cost [33, 18, 14] — examples include short-lived user sessions, stored aggregates and transformations on large datasets, and general purpose write-through caches for files and database records. Third-tier constructs are extremely fault-tolerant but correspondingly slow and expensive, and soft state is typically used to limit their role in performance-critical data paths. For example, the developer of an online travel service might use the memory of the service instance to store intermediate choices made by a user during the booking process, so that only the final sale transaction — a small fraction of all user activity — hits the third-tier database.

In this paper, we consider the availability of soft state stored in the service tier. When soft state is lost or made unavailable due to service instance crashes and overloads, reconstructing it through user interaction or third-tier re-access can be expensive in time and resources. Replicating soft state provides applications with two critical capabilities: rapid fail-over to other instances during crashes and fine-grained load-balancing across instances to prevent overload [33]. For example, a user request can be transparently redirected during a crash or overload to a different service instance that has up-to-date session context, without requiring her to log in again.

Many options exist for adding high availability to programs that manipulate soft state and these can be broadly classified into three categories: clustered application servers [3], messaging toolkits, and collocated in-memory databases. However, all these options require the developer to write code in "state-aware" ways, mapping data structures to special replication-aware containers, replicated state-machine stores and database-style records, respectively. Such mapping needs to be done carefully to avoid performance issues — for example, storing fine-grained variables in a database could result in severe locking contention [1]. However the natural way for programmers to store and manage soft state in a service is to use conventional in-memory data structures such as hash tables or linked lists.

In this paper, we present Tempest, a Java runtime library designed for easy storage and replication of service-level soft state. Tempest provides developers with *TempestCollections*: custom data structures that look similar to conven-

*This work was supported by DARPA/IPTO under the SRS program and by the Rome Air Force Research Laboratory, AFRL/IF, under the Prometheus program. Additional support was provided by the NSF, AFOSR, and by Intel.

tional Java Collections [27]. Data stored in these structures is transparently replicated across multiple machines, providing fail-over and load-balancing for soft state with zero extra effort by the developer. Under the hood, Tempest uses a fast but unreliable IP multicast operation to spread/broadcast invocations to multiple service instances and then uses gossip-based reconciliation to maintain replica consistency in the face of faults and overloads. Additional adaptive mechanisms are used to maintain high responsiveness during failures.

High-performance in-memory databases are used extensively to store soft state in currently deployed systems [5, 22] and we show that Tempest outperforms them by more than an order of magnitude in large-scale SOA settings. Real-world SOAs often have many services interacting with each other to perform complex tasks — for example, a first-tier front-end could contact a hundred second-tier services to assemble a webpage [15]. Further, each service is potentially contacted in parallel by a large number of load-balancing first-tier front-ends. Tempest scales in both the number of front-ends querying a single service and the number of services being queried by a single front-end. In contrast, in-memory databases fail to scale in these dimensions due to contention, large latency variations and inefficiencies in cross-process interactions between the service and the database.

Accordingly, the contributions of this paper are as follows:

- We present a Java runtime library that exposes data structures to programmers that are transparently replicated across multiple nodes.
- We describe the gossip-based mechanisms used within the system for rapidly replicating data and speeding-up access to it.
- We evaluate Tempest on two datacenter-style testbeds — the Emulab testbed at Utah [30] and a 255 node cluster at Cornell. We show that Tempest maintains rapid responsiveness under heavy loads and outperforms in-memory and on-disk databases while scaling in two important dimensions — the number of front-ends accessing a single service and the number of services composing a single response.

The remainder of this paper is structured as follows: Section 2 describes the interface and semantics provided by TempestCollections to service developers. Section 3 describes the protocols and mechanisms used by Tempest to implement the TempestCollection abstraction, and Section 4 provides an evaluation of Tempest on datacenter testbeds.

2 The TempestCollection Abstraction

2.1 Service Model

Services are self-contained entities designed to support interoperable machine to machine interaction over a network [31]. Each service exposes an API through which a set of methods can be invoked by clients, and each service offers its own quality of service and availability guarantees. Take for example the interface of a shopping cart service as listed in Figure 1.

```
public interface ShoppingCartIF extends Iterable {
    update int add(String itemSymbol, int count);
    update int remove(String itemSymbol, int count);
    update int update(String itemSymbol, int count);
    read int check(String itemSymbol);
}
```

Figure 1. ‘Shopping Cart’ service interface.

Add, remove and update do the obvious things; these are classified as update operations because they change state. Check is a read operation; it retrieves the current number of items in the shopping cart for the symbol of interest. Clients issue `add/remove/update` and `check` requests against the service; the service processes each request and in return sends back a reply. This simple example can be trivially extended to services like item browsing history, product availability, product rating, or caching services.

In this work we assume that business logic is collocated with soft state stored in the memory of the service instance; as mentioned before, this is a natural design choice for applications requiring scalability and responsiveness. For example, storing shopping cart information in-memory allows the service to handle a large quantity of browsing traffic that otherwise would have reached the third tier. A developer implementing the shopping cart service in Java could use different data structures to store the state of the cart; a natural way would involve using a hash table to store mappings between item identifiers and corresponding counts.

Service state is modified by updates sent to it through its *interface* — in the conventional three-tier setup, this refers to database state hidden by the service, but in our case it includes soft state maintained by the service. In our shopping cart example, items are added to or subtracted from the cart.

The implementation of a service as a Java application running on a single node is obviously prone to crashes, overloads and slowdowns. Our goal is to transparently replicate a service on multiple nodes while retaining the programming ease and familiarity of Java's built-in Collection data structures. Accordingly, we provide developers with *TempestCollections* — data structures very similar to vanilla Collections but providing automatic replication of the data stored in them.

2.2 TempestCollection: Syntax and Semantics

TempestCollections are syntactically identical to standard Java Collections. For example, a TempestHashtable exposes `get` and `put` methods while a TempestSet has `add, remove` methods. Like most Java Collections, objects stored in a TempestCollection cannot be modified in place. For example, to change a field inside an Object stored in a TempestSet, the programmer would have to remove the Object, modify it and then re-insert it into the set.

This is a very common programming idiom within the Java Collections framework. For example, Java TreeSets provide ordered iteration over their elements, and changing the value of an item in-place can push the TreeSet into an inconsistent state by modifying the outcome of compare operations. Programmers are expected to instead change values by removal, modification and re-insertion if they want the TreeSet to remain consistent and ordered. In general, many Collections involve comparisons through `equals` and `compareTo` — such as HashMaps, TreeSets or HashSets — and do not allow safe in-place modification of objects stored within them. In this respect, TempestCollections offer identical semantics.

To prevent accidental modification of stored items, TempestCollections implement *by-value* parameter passing. Deep clones of added Objects are stored within the TempestCollection and clones of stored Objects are returned by accessor functions. For example, calling `put(`K`, `A`)` on a TempestHashMap will result in a clone A' being stored within the collection, and calling `get(`K`)` will return A'' to the programmer.

However, the Tempest runtime can alter the contents of TempestCollections by adding and / or removing items to keep collections consistent across replicas. TempestCollections provide *eventual consistency* — all replicas converge to the same set of objects [10, 8]. An implication of this model is that the programmer is not provided with ACID transactions; however, this is not a major limitation for soft state management [8]. In many soft state applications, data stored within structures is naturally immutable — for instance, a browsing history service that stores a list of item identifiers. For others, updates do not depend on current state — for example, a map from users identifiers to last viewed items. Even if the soft state is manipulated with arbitrary operations, it is expected by definition to not have strong semantics — the user is always asked to verify the contents of a shopping cart or the final itinerary of a travel plan before committing to it.

To summarize, TempestCollections are data structures exposing interfaces identical to those in the Java Collections framework and supporting similar semantics by not allowing in-place modifications of stored Objects. The sole deviation from the Java Collections framework – aside the weak consistency implications – is that Tempest enforces Object immutability by passing parameters by-value — a side effect of this is the possibility for services to operate on stale data.

3 Tempest Architecture

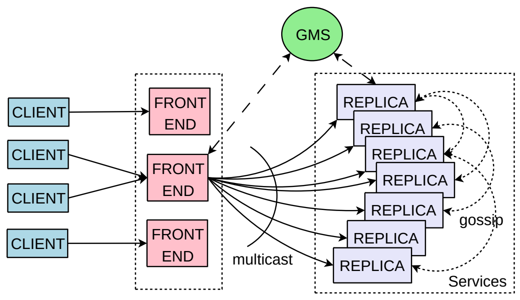


Figure 2. Tempest architecture.

In this section we describe the mechanisms used to implement replicated TempestCollections. Tempest services reside on second-tier *servers*; a single server represents the platform configuration on a single computer and might run several services. A service instance stores data in one or more TempestCollections. Multiple instances of a service execute across different servers, and invocations to this service are sent by first-tier front-ends to all the service instances — see Figure 2 (front end initiates a multicast to the servers that contain replicas of the same service instance).

The life-cycle of a Tempest invocation begins when a client sends a request over the Internet to the datacenter, which gets load balanced to a web-facing front-end node. The front-end is then responsible for contacting a set of services and aggregating individual service responses into a composite result that it returns to the client. Front-ends use IP multicast to perform web-service invocations on service instances, allowing very rapid communication in the general case; when multicast packets are dropped or delivered at different orders across instances, gossip-based reconciliation is used to repair gaps and errors in the TempestCollections maintained by them.

3.1 Client Invocations

When a client request enters the datacenter at a front-end, it's tagged with a web service invocation identifier (wsiid) consisting of a tuple containing the front-end node identifier and sequence number. Front-end node identifiers are obtained by applying the SHA1 consistent hash function over the front-end's IP address and port pair. Each Tempest request is thus uniquely identified by its wsiid.

As mentioned previously, Tempest differentiates between updates and queries or reads. For updates, Tempest uses IP multicast to send the operation directly to the full set of Tempest servers that hold replicas of the service for which the requests were intended. A hashing mechanism is employed to determine which server instance is responsible for replying. In the absence of message loss, which common, IP multicast within datacenters is reliable and ordered.

For read requests, front-ends use an adaptive querying mechanism. Each front-end periodically multicasts a beacon to each service and waits for unicast responses from each instance. It selects the k instances that respond first — where k is the *redundant querying* parameter — and subsequently directs service read invocations to these instances.

3.2 The Tempest Gossip Mechanism

Tempest is designed under the assumption that the multicast protocol used might not be fully reliable or might recover lost packets at high latencies. If some replicas miss an update, they can become inconsistent. Tempest uses a gossip protocol to repair these kinds of inconsistencies rapidly. Servers use a custom tailored gossip protocol to reconcile differences between the TempestCollection replicas.

Tempest keeps track of all the operations performed at the data structure boundary — this is possible due to our *by-value* semantics of altering the collections. When an object is added to a collection, it is annotated with the web service invocation identifier of the corresponding invocation; when an object is removed from a collection, a death certificate for it is created and annotated with the wsiid. A death certificate is simply a means of retaining the information necessary to identify which objects were removed from a collection. In particular each TempestCollection keeps a history of the removed objects in an internal private data structure not exposed via the standard interface.

The anti-entropy mechanism works by having each server "gossip about" the sets of web service invocation identifiers (wsiids) that annotated objects in TempestCollections. Suppose for example that during one gossip round we have two service replicas r_1 and r_2 respectively engaged in an exchange; let their *sets* of wsiids be denoted by $w(r_1)$ and $w(r_1)$. If $w(r_1) = w(r_2)$ no action is taken, otherwise some invocations were missed by one (or both) and a "reconciliation" phase is triggered:

- If $w(r_1) \subset w(r_2)$ then r_1 missed invocations and holds a stale version of the state – as a result r_1 retrieves from r_2 the objects and death certificates annotated with the wsiids from the set $w(r_2) \setminus w(r_1)$. Objects referred by the death certificates are removed, newly received objects are added; also r_1's set of wsiids is updated accordingly: $w(r_1) \leftarrow w(r_2)$.
- If $w(r_1) \not\subset w(r_2)$ and $|w(r_1)| \neq |w(r_2)|$ (the sets have different cardinality) both replicas have missed at least one update each, therefore to make progress it is safe for any of the replicas to assume the other replica's state – without violating the "eventual consistency" guarantees offered by the system. Choose the replica that has the smaller w set – let it be r_1 without loss of generality; r_1 performs the following steps:
  - For every identifier i in the set $w(r_1) \setminus w(r_2)$, if i annotates an object then the object is discarded, otherwise if i annotates a death certificate the object referred by the death certificate is "resurrected" (added back to the collection).
  - Fetch from r_2 all objects and death certificates annotated with identifiers from the set $w(r_2) \setminus w(r_1)$. Remove objects referred by the death certificates, add the new objects, and update $w(r_1) \leftarrow w(r_2)$. Here we used the heuristic of discarding the state of the replica that received less invocations, however one can imagine other criteria.
- If $w(r_1) \not\subset w(r_2)$ and $|w(r_1)| = |w(r_2)|$ then the initiator of the gossip round between r_1 and r_2 "plays the role" of the replica with the smaller w and performs the same operations as in the previous case.

An upcall is provided such that the service developer is notified when a gossip reconciliation was triggered.

If no new invocations are issued against the system, and if no permanent network partition that splits the servers into two or more disjoint communication parties occurs the TempestCollection replicas will eventually contain identical elements with probability 1.0 [9].

During a gossip round, there can never be more than 3 messages issued per process (by protocol design). Currently the sets of web service identifiers are monotonically increasing as new invocations are issued, therefore gossip messages size increases with time. We are working on a method for garbage collecting the stale wsiids by appending an epoch number at wsiid generation time — tempest servers will discard wsiids that are more than δ epochs old for some choice of parameter δ. Another option is to use efficient set reconciliation methods like the ones in [20, 4].

The strength of gossip protocols lies in their simplicity, the fact that they are robust (there are exponentially many paths information can travel in between two endpoints), and the ease with which they can be tuned to trade speed of delivery against resource consumption. The epidemic protocols implemented in Tempest evolved out of our previous work on simple primitive mechanisms that enable scalable services architectures in the context of large-scale datacenters. A more thorough description of the basic protocols and some of the optimizations can be found in [19].

3.3 Membership and Failure Detection

Membership in Tempest is handled by the Group Membership Service (GMS), which maintains the mapping between servers and service replicas. In addition, it also acts as a UDDI (Universal Description Discovery and Integration) registry providing appropriate WSDL (Web Services Description Language) descriptions for the services deployed on Tempest servers. The GMS also fills the administrator role for Tempest servers, monitoring the overall stress and spawning new servers to match the load imposed on the system. Finally, it monitors components to detect failures and adapt the configuration.

Tempest assumes that processes fail by crashing and can be reliably detected as faulty by timeout. Accordingly, Tempest processes monitor the peers with which they interact using a secondary gossip-based heartbeat mechanism. Processes that are thought to be deceased are reported to the GMS, which waits for f distinct suspicions before actually declaring it *deceased*. It then updates and disseminates group membership information to all interested parties. While in our experiments the GMS is hosted on a single high-end node, in a datacenter it could potentially be replicated and partitioned across multiple machines for scalability and fault-tolerance.

3.4 Node Recovery and Checkpointing

TempestCollections are automatically checkpointed. Periodically, each Tempest server batches the items in each TempestCollection and writes them atomically to disk. When a node crashes and reboots, upon starting the Tempest server, the services are brought up to date with the state that was last written to disk before the crash.

When a server is newly spawned, or when a server that has been unavailable for a period of time missed many updates, Tempest employs a bulk transfer mechanism to bring the server up to date. In such cases, a source server is selected and the contents of the relevant TempestCollections are transmitted over a TCP connection. When multiple services are collocated in a single server, the transfers are batched and sent over a single shared TCP stream.

Newly spawned services and services that rebooted after a crash will consequently "catch up" gracefully with the rest of the service replicas by means of the epidemic protocols.

4 Experimental Evaluation

Tempest was implemented in Java, enhancing the Apache Axis Soap [28] web services stack with a new transport protocol that uses a multicast primitive, i.e. SOAP over TempestTransport instead of SOAP over HTTP. The

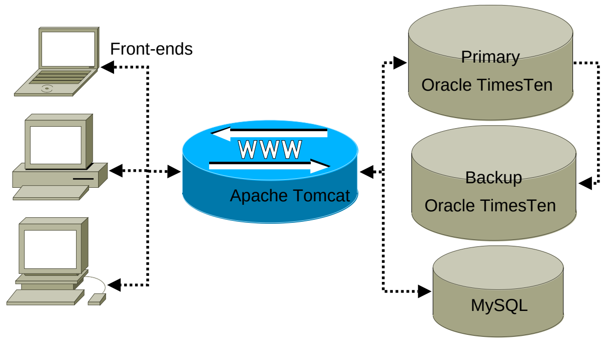


Figure 3. Baseline configurations.

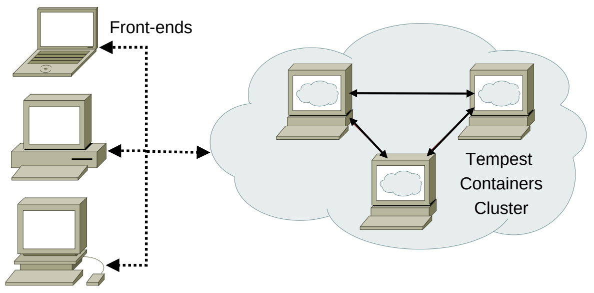


Figure 4. Tempest configuration.

deep cloning capability was implemented using the Java Reflection API. The system components are built with Java's non-blocking I/O primitives using a high performance event driven model similar to the SEDA [29] architecture.

The evaluation is structured as follows: in subsection 4.1 we show that a single replicated Tempest service can provide rapid response to large numbers of concurrent front-end requests. In subsection 4.2 we show that this is true even when services are heavily loaded. Finally, in subsection 4.3, we show that the two knobs provided by Tempest — number of replicas per service and number of redundant queries — enable rapid predictable response for "service-clouds" composed of many collaborating services with differing timing characteristics.

4.1 Scalability in the Number of Concurrent Connections – Micro bechmarks

We ran a set of micro benchmarks to compare Tempest against four multi-tier baseline scenarios. In all configurations we had the same set of front-ends interacting with the `ShoppingCart` web service. On one hand we deployed the service on top of the Apache Tomcat server. The service stores the data using a relational database repository as shown in figure 3. We stored the data using the Oracle TimesTen in memory database (configured in "high performance cache-mode" for in-memory operations only) first co-located with the Tomcat server, second on a remote third-tier machine and lastly deployed in a primary-backup configuration with the primary co-located with the Tomcat container and the backup on the third-tier machine. The primary-backup scheme provided by TimesTen that we used is called *return receipt*, and it ensures that upon submitting a request to the master the application is blocked until the replication scheme confirms that the update has been received by the backup. Since we configured TimesTen to work without committing durably to disk every transaction, the stronger *return twosafe* replication mode was not necessary. We also use an ubiquitous on-disk database engine, and for that purpose we relied on MySQL 5.0 with the InnoDB storage engine configured for ACID compliance — flushing the log after every transaction commit, and the underlying operating system (Linux 2.6.15) with the file system mounted in synchronous mode and with barriers enabled. On the other hand we have deployed the `ShoppingCart` service on 3 replicated Tempest servers gossiping at a rate of once every 100 milliseconds (see figure 4) – we did not replicate Tomcat for load balancing since all Tempest replicas were configured to receive every update. The Tempest `ShoppingCart` service stores the data inside a *TempestMap*.

The workload consists of multiple clients issuing 1024 byte requests at a rate of 100 requests per second against the `ShoppingCart` service. Requests are issued in a closed loop [25]. Every experiment had a startup phase in which we populated the data repository with 1024 distinct objects. Client requests were drawn from a Zipf distribution (with $s = 1$) over the space of object identifiers – reads and writes equally distributed. We report measurements of the Web Service Interaction Time, i.e. the request latency as observed by 1, 2, 4, 16, 32, 64, 128, 256, 512, 800 and 1024 concurrent clients. Results are averaged over 40000 runs per client.

Figure 5 shows that Tempest latency is significantly less – often by over an order of magnitude – than any of the baselines, thus confirming that fault-tolerant services with time-critical properties can be built on top of the Tempest platform. The graphs also indicate that Tempest scales well with the number of concurrent requests.

As can be seen from the breakdown of the latency, most of the overhead comes from the round trip time and the Tomcat container, which is to be expected since the workload consists of operations on small data records over the database — we hypothesize that database lock contention has not kicked in yet, the Tomcat container being the first one to experience severe overload.

Looking more carefully at the breakdown of the latency in figure 5 (the 1-to-32 concurrent clients spectrum) one can notice that the time spent by a Tempest service manipulating the data (i.e. performing object deep cloning, data structure lock contention, web service invocation identifier tagging and index maintenance) is small compared to the database interaction — as a matter of fact it grows remains around 1 millisecond no matter what the number of concurrent clients is — showing that fine grained data structures allow for better performance under contention.

4.2 Graceful Recovery under Heavy Load

Next, we ran a set of experiments to report on Tempest's behavior in the face of failures. Node crashes turned out not to be especially interesting since Tempest's gossip failure detection protocols quickly detect that the node has failed, expel it from the group and shift work to other nodes. More details on the timeliness of a variant of the gossip based failure detector we used can be found in our previous work [19]. We did however identify a class of overload scenarios that have a more visible impact on the Tempest replicated services. These scenarios degrade some service components without crashing them. The services become lossy and inconsistent, and queries return results based on stale data. Two questions are of interest here: behavior during the overload, and the time required to recover after it ends.

We replicated the `ShoppingCart` service on 6 Tempest servers running on the Cornell cluster – each machine is a 1.33Ghz Intel single CPU blade-server with 512MB of RAM. We inject a single source stream of updates at a particular rate of one update every 20 milliseconds. The same client perform query requests on 8 concurrent threads at the same time. The query stream is at a higher rate than the updates (in this case 4 times higher). Client requests were drawn from a Zipf distribution (with $s = 1$) over the space of object identifiers – reads and writes equally distributed.

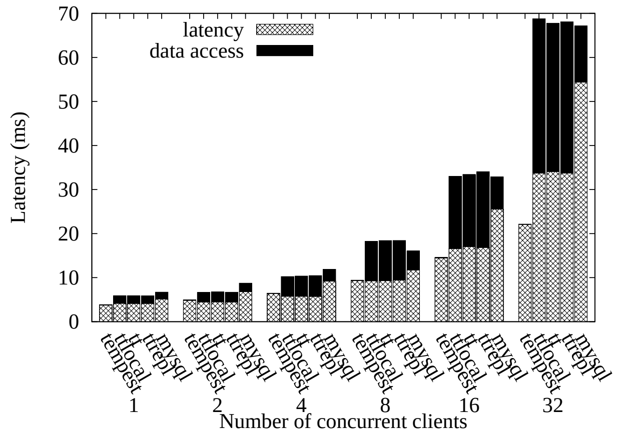
The overload unfolds in the following way:

- At time t from the start of the experiment 128 "rogue" clients bombard with requests 3 of the Tempest servers. Call the Tempest services *victims*.
- At time $t + \Delta$ the rogue clients terminate.

In the experiments that follow, t is 10, and Δ is 30 seconds.

The rogue clients bombard the victims with multiple streams of continuous IP multicast requests in the attempt to saturate their processing capacity. However, we found that this was not enough to perturb the normal behavior of the servers, hence at the same time we superimposed additional background load on the victim servers. These attacks do not actually cause the servers to crash, but they do cause them to become overloaded in processing incoming updates and hence return stale results.

Server overloads will not influence the performance of Tempest at non-attacked services, hence *we report only on the impact of the disruption at the affected replicas*. Figure 6 shows the number of "stale" query results on the y-



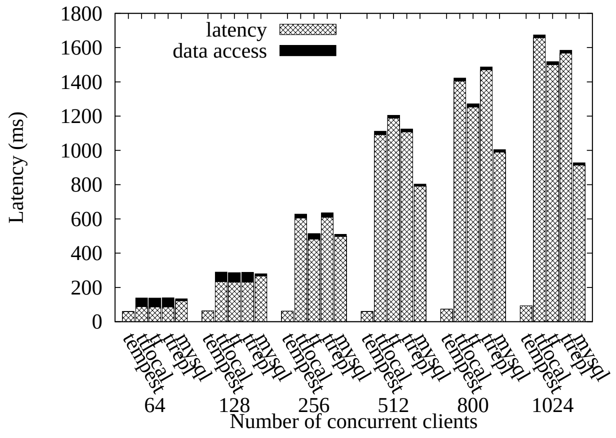


Figure 5. Request latency. Each group of bars represent tempest (*tempest*), times ten on the local machine with tomcat (*ttlocal*), times ten on a remote machine (*tt*), times ten in primary-backup mode with the primary on the same machine as tomcat and the backup on a remote machine (*ttrepl*), and mysql on a remote machine (*mysql*).

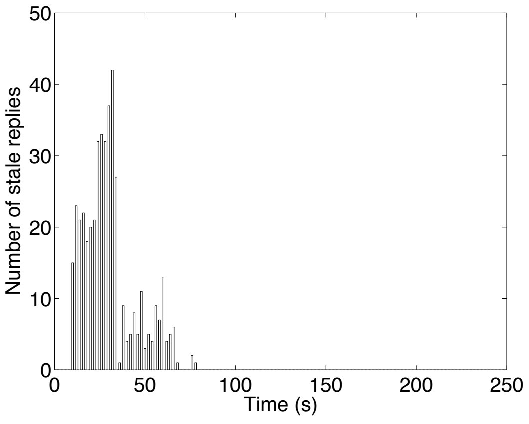


Figure 6. Number of stale results.

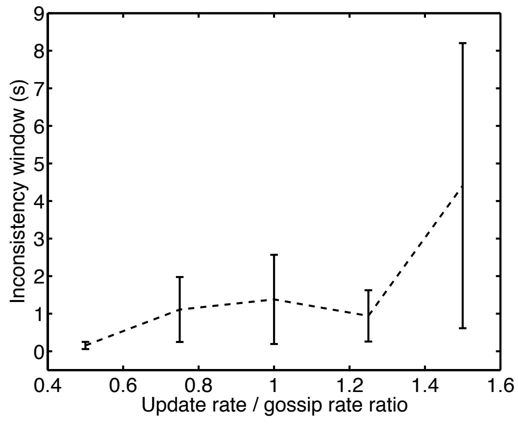


Figure 7. Inconsistency window.

axis against the time in seconds on the x-axis, binned in 2-second intervals. The client issues an update every 20 milliseconds and the Tempest gossip rate is set at once every 40 milliseconds. Throughout this period, the victim nodes are overloaded and drop packets, while the Tempest repair protocols labor to repair the resulting inconsistencies. Meanwhile, queries that manage to reach the overloaded nodes could glimpse stale data (not reflecting recent issued updates since the updates were lost). Once the attack ends, Tempest is able to gracefully recover.

The ratio of the gossip rate to the update rate will determine the robustness of Tempest to this sort of overload scenario. To quantify this effect, Figure 7 shows the inconsistency window as perceived by clients during the disruption. This is the period of time during which clients of a service see more than one stale query result within a 2-second interval. The inconsistency window is plotted against the ratio between the update rate and the Tempest gossip rate, with the update rate fixed at 1/20 milliseconds. The window is minimized when the gossip rate is at least as fast as the update rate.

4.3 Scalability in the Number of Services

To estimate how Tempest scales in different dimensions — in particular size of the collaborating services, number of front-ends and number of replicas — we built a synthetic PetStore on top of Tempest and evaluated it on the Emulab testbed. The application consists of a battery of front ends issuing requests to a “cloud” of services.

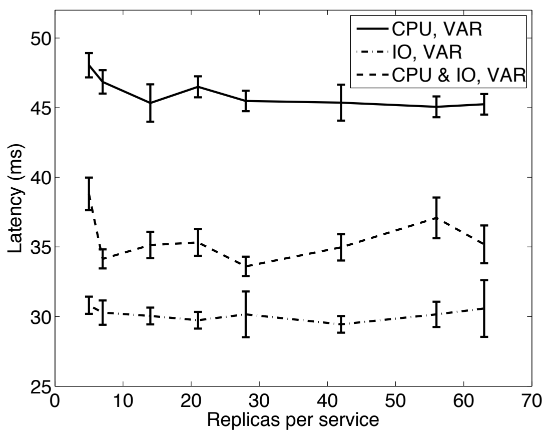


Figure 8. Large variance service latency.

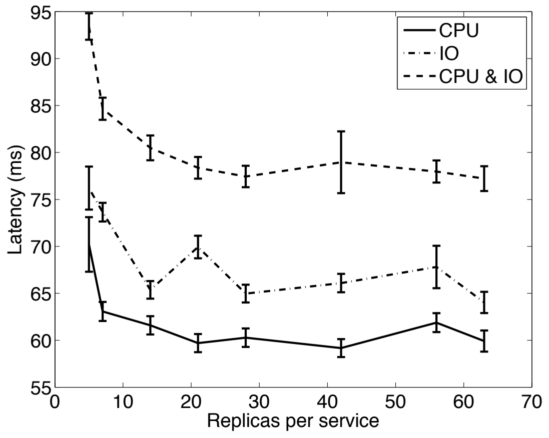


Figure 9. Small variance service latency.

The services have different response time characteristics: some are IO intensive – for example an indexing service may access disk much more often than the average service, others are CPU intensive – for example a recommendation service may require considerably more CPU cycles than the average service, while other services are both IO and CPU bound. We also consider the response time variances for these types of services, in particular the PetStore services have both small and large response time variance. We observed that services performing multiple IO operations are likely to suffer from scheduling delays. Lock contention within Tempest may be another cause for large response time variance.

We ran a set of baseline experiments to measure the behavior of each type of service individually, under normal load. The experiment consisted of two front ends issuing request streams (half updates half reads) of one query every 40 milliseconds in closed loop to a single replicated service. Services have the gossip rate set for once every 100 milliseconds. We repeated the experiment for various number of replicas and for each of the types of services mentioned above. Figure 8 shows the query latency for services with large response time variance, and small response time variance respectively (figure 9). The error bars represent standard error. Note that even for services that we instrumented to have small response time variance, if they are IO bound they do exhibit large variance — in particular note the CPU & IO bound service for 42 replicas and the IO bound service for 56 replicas. We should note that for this client request load, the service instances become overloaded if we drop below about 3 replicas, and we don't report those values (response times are meaningless when the service isn't able to keep up with the request rate).

Next we evaluated the PetStore as a "cloud" of seven services — the six with the characteristics presented in the previous experiments, along with another baseline service that shows the overhead caused by Tempest. Four front-ends perform multi-service requests (half queries half updates) against the PetStore in a closed loop, each at a rate of once every 50 milliseconds — we chose the rate so as to not completely overload the platform and observe queueing effects instead.

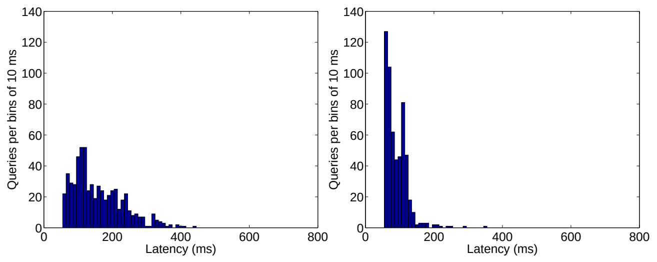


Figure 10. Pet-store response time histograms, left: no replicas, right: 8 replicas.

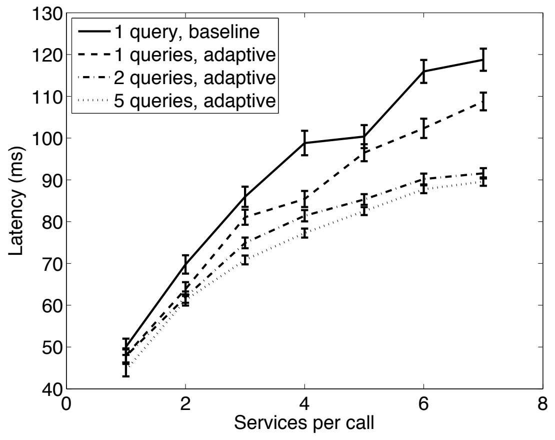


Figure 11. Pet-store latency, 5 replicas each.

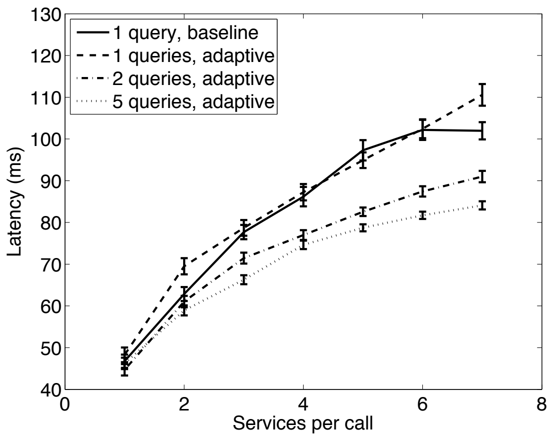


Figure 12. Pet-store latency, 8 replicas each.

Figure 10 show response time distributions for multi-service requests sent to all services — every request issued by a front-end is sent in parallel to each service, the front end returns when replies from every service is received. Requests have the redundant querying parameter $k = 2$. Each histogram shows the number of requests per bins 10 milliseconds wide. We show the scenarios: the one in which neither of the services is replicated, and the one in which services have 8 replicas each. The graphs show that replication provides more opportunities for queries to be absorbed by load balancing and that redundant querying pays off.

Figures 11 and 12 show response times for multi-service requests (with standard error denoting the error bars). Every multi-request issued by a front-end chooses at random n distinct services, where n is the number of services per query, presented on the x-axis. We used the adaptive query algorithm with the k parameter set to 1, 2 and 5. For baseline we used a simple query discovery algorithm by which the first query for a service is multicast, and all subsequent queries are sent to the one replica that replied the fastest to the multicast. In figure 11 every service is replicated 5 times, while in figure 12 every service is replicated 8 times. First we conclude that redundant querying does indeed improve performance, with the largest payoff for $k = 2$. Second, the adaptive querying algorithm pays off mostly in scenarios where the number of replicas is small.

5 Related Work

Soft state mechanisms have been used extensively in network protocols [32, 12], as well as in large cluster-based services like Porcupine [24] and others [14, 6, 26]. Proposals exist for extending the standard web-service model to include soft state — a prominent example is the Grid Computing standard [13]. Recovery-oriented computing [7] is an alternative approach to providing fast failover and availability in the face of failures — however, it does not replace replication as a mechanism for balancing heavy load across multiple machines. Distributed data structures have been proposed before [16] as building blocks for clustered services. The work in [33] is very similar in spirit to Tempest, but examines the orthogonal question of providing customizable durability levels through a single storage abstraction; one of these levels is meant for soft state that needs to be replicated for high availability. SSM [18] is a system for managing and storaging a particular category of soft state — user session information.

Clustered application servers like BEA WebLogic Application Server [3] and IBM WebSphere [17] allow storage of state in special containers that are typically stored within persistent databases. There has been a large amount of work in the field of fault-tolerant middleware, especially around CORBA [2, 21, 11], but most of this work does not consider interaction with a database third tier. DBFarm [23] is an architecture for scaling a core of multiple databases through the use of less reliable replicas.

6 Conclusion

Modern three-tier architectures achieve scalability and responsiveness through the extensive use of soft state techniques in the service tier. Availability and rapid fail-over requires data replication, and Tempest provides programmers with data structure abstractions for storing and managing replicated soft state. Tempest scales well in key dimensions — the number of front-ends contacting a service and the number of services contacted by a front-end — and outperforms in-memory databases in realistic settings. As a result, Tempest simplifies the construction of highly responsive systems that seamlessly mask load fluctuations and faults from end-users.

References

[1] M. K. Aguilera, A. Merchant, M. Shah, A. C. Veitch, and C. T. Karamanolis. Sinfonia: a new paradigm for building scalable distributed systems. In *SOSP*, pages 159–174, 2007.

[2] R. Baldoni and C. Marchetti. Three-tier replication for FT-CORBA infrastructures. *Software Practice and Experience, 2003*, 6 2003.

[3] BEA Systems, Inc. Clustering the BEA WebLogic Application Server, 2003. http://e-docs.bea.com/wls/docs81/cluster/overview.html.

[4] J. Byers, J. Considine, and M. Mitzenmacher. Fast approximate reconciliation of set differences. Boston University Computer Science Technical Report 2002-019., 2002.

[5] L. Camargos, F. Pedone, and M. Wieloch. Sprint: a middleware for high-performance transaction processing. In *EuroSys '07: Proceedings of the ACM SIGOPS/EuroSys Eu-*

ropean Conference on Computer Systems 2007, pages 385–398, New York, NY, USA, 2007. ACM.

[6] G. Candea, J. Cutler, and A. Fox. Improving availability with recursive microreboots: a soft-state system case study. *Perform. Eval.*, 56(1-4):213–248, 2004.

[7] G. Candea, S. Kawamoto, Y. Fujiki, G. Friedman, and A. Fox. Microreboot - a technique for cheap recovery. In *OSDI*, pages 31–44, 2004.

[8] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. Dynamo: amazon's highly available key-value store. In *SOSP '07: Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles*, pages 205–220, New York, NY, USA, 2007. ACM.

[9] A. Demers, D. Greene, C. Hauser, W. Irish, J. Larson, S. Shenker, H. Sturgis, D. Swinehart, and D. Terry. Epidemic algorithms for replicated database maintenance. In *Proceedings of the sixth annual ACM Symposium on Principles of Distributed Computing*, pages 1 – 12, Vancouver, British Columbia, Canada, 1987.

[10] A. Demers, K. Petersen, M. Spreitzer, D. Terry, M. Theimer, and B. Welch. The Bayou Architecture: Support for Data Sharing among Mobile Users. In *IEEE Workshop on Mobile Computing Systems & Applications*, 1994.

[11] P. Felber, X. Défago, P. Eugster, and A. Schiper. Replicating CORBA objects: a marriage between active and passive replication. In *Second IFIP International Working Conference on Distributed Applications and Interoperable Systems (DAIS'99)*, pages 375–387, Helsinki, Finland, 1999.

[12] S. Floyd, C. Liu, S. McCanne, and L. Zhang. A reliable multicast framework for light-weight sessions and application level framing. *IEEE/ACM Transactions on Networking (TON)*, 5(6):784–803, 1997.

[13] I. Foster, K. Czajkowski, D. Ferguson, J. Frey, S. Graham, T. Maguire, D. Snelling, and S. Tuecke. Modeling and Managing State in Distributed Systems: The Role of OGSI and WSRF. *Proceedings of the IEEE*, 93(3):604–612, March 2005.

[14] A. Fox, S. Gribble, Y. Chawathe, E. Brewer, and P. Gauthier. Cluster-based scalable network services. *Proceedings of the sixteenth ACM symposium on Operating systems principles*, pages 78–91, 1997.

[15] J. N. Gray. A Conversation with Werner Vogels: Learning from the Amazon technology platform. *ACM Queue*, 4(4), May 2006.

[16] S. D. Gribble, E. A. Brewer, J. M. Hellerstein, and D. E. Culler. Scalable, distributed data structures for internet service construction. In *OSDI*, pages 319–332, 2000.

[17] IBM. WebSphere Information Integrator Q replication, 2005. http://www-128.ibm.com/developerworks/db2/library/techarticle/dm-0503aschoff/.

[18] B. C. Ling, E. Kiciman, and A. Fox. Session state: beyond soft state. In *NSDI'04: Proceedings of the 1st conference on Symposium on Networked Systems Design and Implementation*, pages 22–22, Berkeley, CA, USA, 2004. USENIX Association.

[19] T. Marian, K. Birman, and R. van Renesse. A Scalable Services Architecture. In *Proceedings of the 25th IEEE Symposium on Reliable Distributed Systems (SRDS 2006)*. IEEE Computer Society, 2006.

[20] Y. Minsky, A. Trachtenberg, and R. Zippel. Set reconciliation with nearly optimal communication complexity. *IEEE Transactions on Information Theory*, 49(9):2213–2218, 2003.

[21] P. Narasimhan, L. E. Moser, and P. M. Melliar-Smith. Lessons Learned in Building a Fault-Tolerant CORBA System. In *DSN '02: Proceedings of the 2002 International Conference on Dependable Systems and Networks*, pages 39–44, Washington, DC, USA, 2002. IEEE Computer Society.

[22] M. Pezzini. The Evolution of Transaction Processing in Light of .NET and J2EE. *Business Integration Journal Online*, November 2005.

[23] C. Plattner, G. Alonso, and M. T. Özsu. Dbfarm: A scalable cluster for multiple databases. In *Middleware*, pages 180–200, 2006.

[24] Y. Saito, B. N. Bershad, and H. M. Levy. Manageability, availability and performance in Porcupine: a highly scalable, cluster-based mail service. In *SOSP '99: Proceedings of the seventeenth ACM symposium on Operating systems principles*, pages 1–15, New York, NY, USA, 1999. ACM Press.

[25] B. Schroeder, A. Wierman, and M. Harchol-Balter. Open vs closed: a cautionary tale. In *Proceedings of the 3rd Symposium on Networked System Design and Implementation (NSDI)*. Networked System Design and Implementation (NSDI), 2006.

[26] K. Shen, T. Yang, L. Chu, J. L. Holliday, D. A. Kuschner, and H. Zhu. Neptune: scalable replication management and programming support for cluster-based network services. In *USITS'01: Proceedings of the 3rd conference on USENIX Symposium on Internet Technologies and Systems*, pages 17–17, Berkeley, CA, USA, 2001. USENIX Association.

[27] Sun Microsystems. The Collections Framework, 1995. http://java.sun.com/docs/books/tutorial/collections/index.html.

[28] The Apache Software Foundation. Apache Axis, 2006. http://ws.apache.org/axis/.

[29] M. Welsh, D. E. Culler, and E. A. Brewer. SEDA: An Architecture for Well-Conditioned, Scalable Internet Services. In *Symposium on Operating Systems Principles*, pages 230–243, 2001.

[30] B. White, J. Lepreau, L. Stoller, R. Ricci, S. Guruprasad, M. Newbold, M. Hibler, C. Barb, and A. Joglekar. An integrated experimental environment for distributed systems and networks. In *Proc. of the Fifth Symposium on Operating Systems Design and Implementation*.

[31] World Wide Web Consortium. Web Services Architecture, 2002. http://www.w3.org.

[32] L. Zhang, S. Deering, D. Estrin, S. Shenker, and D. Zappala. RSVP: a new resource reservation protocol. *Communications Magazine, IEEE*, 40(5):116–127, 2002.

[33] X. Zhang, M. A. Hiltunen, K. Marzullo, and R. D. Schlichting. Customizable service state durability for service oriented architectures. *Sixth European Dependable Computing Conference*, 0:119–128, 2006.

Dr. Multicast: *Rx* for Data Center Communication Scalability[*]

Ymir Vigfusson, Cornell University
Hussam Abu-Libdeh, Cornell University
Mahesh Balakrishnan, Cornell University
Ken Birman, Cornell University
Yoav Tock, IBM Haifa Research Lab

Abstract

Data centers avoid IP Multicast because of a series of problems with the technology. We propose *Dr. Multicast* (the MCMD), a system that maps traditional IPMC operations to either use a new point-to-point UDP multisend operation, or to a traditional IPMC address. The MCMD is designed to optimize resource allocations, while simultaneously respecting an administrator-specified acceptable-use policy. We argue that with the resulting range of options, IPMC no longer represents a threat and could therefore be used much more widely.

1 Introduction

As data centers scale up, IP multicast (IPMC) [8] has an obvious appeal. Publish-subscribe and data distribution layers [6, 7] generate multicast distribution patterns; IPMC permits each message to be sent using a single I/O operation, reducing latency both for senders and receivers (especially, for the last receiver in a large group). Clustered application servers [1, 4, 3] need to replicate state updates and heartbeats between server instances. Distributed caching infrastructures [2, 5] need to update cached information. For these and other uses, IPMC seems like a natural match.

Unfortunately, IPMC has earned a reputation as a poor citizen. Routers must maintain routing state and perform a costly per-group translation[11, 9]. Many NICs can only handle a few IPMC addresses; costs soar if too many are used. Multicast flow control is also a black art. When things go awry, a multicast storm can occur, disrupting the whole data center. Perhaps most importantly, management of multicast use is practically unsupported.

Our paper introduces *Dr. Multicast* (the MCMD), a technology that permits data center operators to enable IPMC while maintaining tight control on its use. Applications are coded against the standard IPMC socket interface, but IPMC system calls are intercepted and mapped into one of two cases:

- A true IPMC address is allocated to the group.
- Communication to the group is performed using point-to-point UDP messages to individual receivers, using a new *multi-send* system call.

The MCMD tracks group membership, using a gossip protocol. It translates each send operation on a multicast group into one or more send operations, optimized for system objectives. Finally, to implement this optimization policy, it instantiates in a fault-tolerant fashion a service that computes the best allocation of IPMC addresses to groups (or to overlapping sets of groups), adapting as use changes over time.

Users benefit in several ways:

- Policy: Administrators can centrally impose traffic policies within the data center, such as limiting the use of IPMC to certain machines, placing a cap on the number of IPMC groups in the system or eliminating IPMC entirely.
- Performance: The MCMD approximates the performance of IPMC, using it directly where possible. When a multicast request must be translated into UDP sends, the multi-send system call reduces overheads.
- Transparency and Ease-of-Use: Applications express their intended communication pattern using standard IPMC interfaces, rather than using hand-coded implementations of what is really an administrative policy.
- Robustness: The MCMD is implemented as a distributed, fault-tolerant service.

We provide and evaluate effective heuristics for the optimization problem of allocating the limited number of IPMC addresses, although brevity limits us to a very terse review of the framework, the underlying (NP-complete) optimization question, and the models used in the evaluation.

[*]This work was supported by grants from AFRL, AFOSR, NSF, Cisco and Intel.

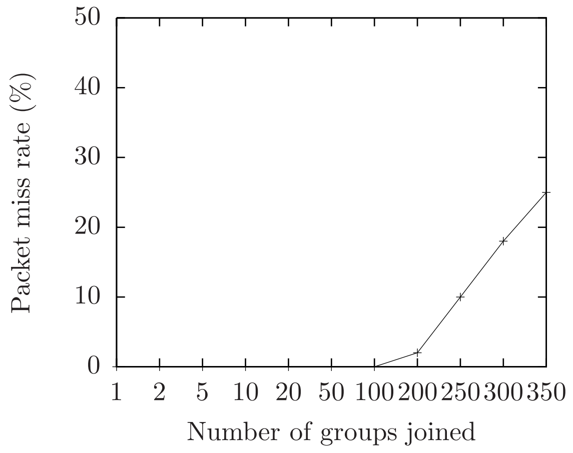


Figure 1: Receiver packet miss rate vs. number of IPMC groups joined

2 IPMC in the Data Center

Modern data centers often have policies legislating against the use of IPMC, despite the fact that multicast is a natural expression of a common data communication pattern seen in a wide range of applications. This reflects a number of pragmatic considerations. First, IPMC is perceived as a potentially costly technology in terms of performance impact on the routing and NIC hardware. Second, applications using IPMC are famously unstable, running smoothly in some settings and yet, as scale is increased, potentially collapsing into chaotic multicast storms that disrupt even non-IPMC users.

The hardware issue relates to imperfect filtering. A common scheme used to map IP group addresses to Ethernet group addresses involves placing the low-order 23 bits of the IP address into the low-order 23 bits of the Ethernet address [8]. Since there are 28 significant bits in the IP address, more than one IP address can map to an Ethernet address. The NIC maintains the set of Ethernet mappings for joined groups and forwards packets to the kernel only if the destination group maps to one of those Ethernet addresses. As a result, with large numbers of groups, the NIC may accept undesired packets, which the kernel must discard.

Figure 1 illustrates the issue. In this experiment, a multicast transmitter transmits on $2k$ multicast groups, whereas the receiver listens to k multicast groups. We varied the number of multicast groups k and measured the CPU consumption as well as the packet loss at the receiver. The transmitter transmits at a constant rate of 15,000 packets/sec, with a packet size of 8,000 bytes spread across all the groups. The receiver thus expects to receive half of that, i.e. 7,500 packets/sec. The receiver and transmitter have 1Gbps NICs and are connected by a switch with IP routing capabilities. The experiments were conducted on a pair of single core Intel® Xeon™ 2.6GHz machines. Figure 1 shows that the critical threshold that the particular NIC can handle is roughly 100 IPMC groups, after which throughput begins to fall off.

The issue isn't confined to the NIC. Performance of modern 10Gbps switches was evaluated in a recent review [10] which found that their IGMPv3 group capacity ranged between as little as 70 and 1,500. Less than half of the switches tested were able to support 500 multicast groups under stress without flooding receivers with all multicast traffic.

The MCMD addresses these problems in two ways. First, by letting the operator limit the number of IPMC addresses in use, the system ensures that whatever the limits in the data center may be, they will not be exceeded. Second, by optimizing to use IPMC addresses as efficiently as possible, the MCMD arranges that the IPMC addresses actually used will be valuable ones – large groups that receive high traffic. As seen below, this is done not just by optimizing across the groups as given, but also by discovering ways to aggregate overlapping groups into structures within which IPMC addresses are shared by multiple groups, permitting even greater efficiencies.

The perception that IPMC is an unstable technology is harder to demonstrate in simple experiments: as noted earlier, many applications are perfectly stable under most patterns of load and scale, yet capable of being extraordinarily disruptive. The story often runs something like this. An application uses IPMC to send to large numbers of receivers at a substantial data rate. Some phenomenon now triggers loss. The receivers detect the loss and solicit retransmissions, but this provokes a further load surge, exacerbating the original problem. A multicast storm ensues, saturating the network with redundant retransmission requests and duplicative multicasts. With MCMD the operator can safely deploy such an application: if it works well, it will be permitted to use IPMC; if it becomes problematic, it can be mapped to UDP merely by changing the acceptable use policy. More broadly, the MCMD encourages developers to express intent in a higher-level form, rather than hand-coding what is essentially an administrative policy.

3 Design

The basic operation of MCMD is simple. It translates an application-level multicast address used by

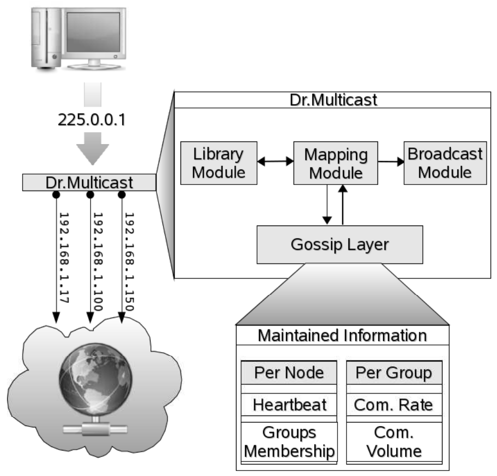


Figure 2: Overview of the MCMD architecture

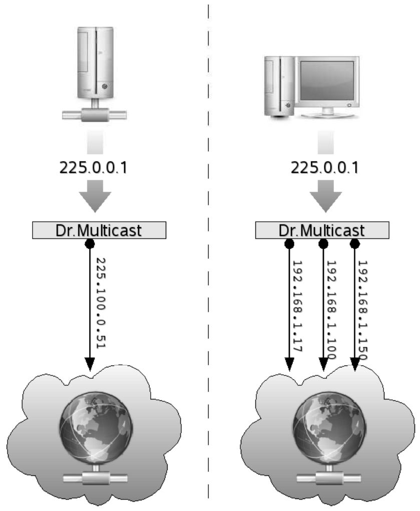


Figure 3: Two under-the-hood mappings in MCMD, a direct IPMC mapping on the left and point-to-point mapping on the right.

an application to a set of unicast addresses and network-level multicast addresses. MCMD has two components (see figure 2):

- A *library* module responsible for the *mechanism* of translation. It intercepts outgoing multicast messages and instead sends them to a set of unicast and multicast destinations.

- A *mapping* module responsible for the *policy* of translation. It determines the mapping from each application-level address and a set of unicast and network-level multicast addresses.

3.1 Library Module

The MCMD library module exports a `<sockets.h>` library to applications, with interfaces identical to the standard POSIX version. By overloading the relevant socket operations, MCMD can intercept join, leave and send operations. For example:

- `setsockopt()` is overloaded so that an invocation with the IP_ADD_MEMBERSHIP or IP_DROP_MEMBERSHIP option as a parameter results in a 'join' message being sent to the mapping module. In this case, the standard behavior of `setsockopt` – generating an IGMP message – is suppressed.

- `sendto()` is overloaded so that a send to a class D group address is intercepted and converted to multiple sends to a set of addresses from the kernel.

The library module interacts with the mapping module via a UNIX socket. It pulls the translations for each application-level group from the mapping module. Simultaneously, it pushes information and statistics about grouping and traffic patterns used by the application to the local mapping module.

3.2 Mapping Module

The mapping module plays two important roles:

- It acts as a Group Membership Service (GMS), maintaining the membership set of each application-level group in the system.

- It allocates a limited set of IPMC addresses to different sets of machines in the data center and orchestrates the IGMP joins and leaves required to maintain these IPMC groups within the network.

The mapping module uses a gossip-based control plane using techniques described in [13]. The gossip control plane is extremely resilient to failures and includes a decentralized failure detector that can be used to locate and eject faulty, i.e. irresponsive, machines. It imposes a stable and constant overhead on the system and has no central bottleneck, irrespective of the number of nodes.

The gossip-based control plane essentially replicates mapping and grouping information slowly and continuously throughout the system. As a result, the mapping module on any single node has a global view of the system and can immediately resolve an application-level address to a set of unicast and multicast addresses without any extra communication. The size of this global view is not prohibitive; for example, we can store membership and mapping information for a 1000-node data center within a few

MB of memory. For now, we're targetting systems with a low enough rate of joins, leaves and failures per second to allow for global replication of control information. In the future, we'll replace the global replication scheme with a more focused one to eliminate this restriction.

Knowledge of global group membership is sufficient for the mapping module at each node to translate application-level group addresses into network-level unicast addresses. To fulfill the second function of allocating IPMC addresses in the system, an instance of the specific mapping module running on a particular node in the system acts as a leader. It aggregates information from other mapping modules (via the gossip control plane) and calculates appropriate allocations of IPMC addresses to mandate within the data center. The leader can be chosen according to different strategies – one simple expedient is to query the gossip layer for the oldest node in the system. The failure of the leader is automatically detected by the gossip layer's inbuilt failure detector, which also naturally updates the pointer to the oldest node.

3.2.1 Gossip Control Plane

We first describe an implementation of the mapping module using only the gossip-based control plane. However, the Achilles heel of gossip at large system sizes is *latency* – the time it takes for an update to propagate to every node in the system. Consequently, we then describe approaches to add extra control traffic for certain kinds of critical updates – in particular, IPMC mappings and group joins – that need to be distributed through the system at low latencies.

Gossip-based Failure Detector: The MCMD control plane is a simple and powerful gossip-based failure detector identical to the one described by van Renesse [13]. Each node maintains its own version of a global table, mapping every node in the data center to a timestamp or heartbeat value. Every T milliseconds, a node updates its own heartbeat in the map to its current local time, randomly selects another node and reconciles maps with it. The reconciliation function is extremely simple – for each entry, the new map contains the highest timestamp from the entries in the two old maps. As a result, the heartbeat timestamps inserted by nodes into their own local maps propagate through the system via gossip exchanges between pairs of nodes.

When a node notices that the timestamp value for some other node in its map is older than T_1 seconds, it flags that node as 'dead'. It does not immediately delete the entry, but instead maintains it in a dead state for T_2 more seconds. This is to prevent the case where a deleted entry is reintroduced into its map by some other node. After T_2 seconds have elapsed, the entry is truly deleted.

The comparison of maps between two gossiping nodes is highly optimized. The initiating node sends the other node a set of hash values for different portions of the map, where portions are themselves determined by hashing entries into different buckets. If the receiving node notices that the hash for a portion differs, it sends back its own version of that portion. This simple interchange is sufficient to ensure that all maps across the system are kept loosely consistent with each other. An optional step to the exchange involves the initiating node transmitting its own version back to the receiving node, if it has entries in its map that are more recent than the latter's.

Gossip-based Communication: Thus far, we have described a decentralized gossip-based failure detector. Significantly, such a failure detector can be used as a general purpose gossip communication layer. Nodes can insert arbitrary state into their entries to gossip about, not just heartbeat timestamps. For example, a node could insert the average CPU load or the amount of disk space available; eventually this information propagates to all other nodes in the system. The reconciliation of entries during gossip exchanges is still done based on which entry has the highest heartbeat, since that determines the staleness of all the other information included in that entry.

Using a gossip-based failure detector as a control communication layer has many benefits. It provides extreme resilience and robustness for control traffic, eliminating any single points of failure. It provides extremely clean semantics for data consistency – a node can write only to its own entry, eliminating any chance of concurrent conflicting writes. In addition, a node's entry is deleted throughout the system if the node fails, allowing for fate sharing between a node and the information it inserts into the system.

Group Membership Service: The mapping module uses the gossip layer to maintain group membership information for different application-level groups in the system. Each node maintains in its gossip entry – along with its heartbeat timestamp – the set of groups it belongs to, updating this whenever the library module intercepts a join or a leave. A simple scan of the map is sufficient to generate an alternative representation of the membership information, mapping each group in the system to all the nodes that belong to it. If a node fails, its entry is removed from the gossip map; as a result, a subsequent scan of the map generates a groups-to-nodes table that excludes the node from all the groups it belonged to.

Mapping Module Leader: As mentioned previously, the gossip layer informs the mapping module of the identity of the oldest node in the system, which is then elected as a leader and allocates IPMC addresses. To distribute these allocations back into the system, the leader can just update its own entry in the gossip map with the extra IPMC information. When a receiver is informed of a relevant new mapping, it issues the appropriate IGMP messages required to join or leave the IPMC group as mandated by the mapping module.

A "pure" gossip protocol can have large propagation delays, resulting in undesirable effects such as senders transmitting to IPMC groups before receivers can join them. To mitigate these latency effects, the leader periodically broadcasts mappings at a fixed, low rate to the entire data center. The rate of these broadcasts is tunable; we expect typical values to be a few packets every second. The broadcast acts purely as a latency optimization over the gossip layer; if a broadcast message is lost at a node, the mapping is eventually delivered to it via gossip.

Latency Optimization of Joins: We are also interested in minimizing the latency of a join to an application-level multicast group; i.e., after a node issues a join request to a group, how much time elapses before it receives data from all the senders to that group? While the gossip layer will eventually update senders of the new membership of the group, its latency may be too high to support applications that need fast membership operations. The latency of leave operations is less critical, since a receiver that has left a group can filter out messages arriving in that group from senders who have stale membership information until the gossip layer propagates the change.

In MCMD, we explore two options to speed up joins. The first method is to have receivers broadcast joins to the entire data center. For most data center settings, this is a viable option since the rate of joins in the system is typically quite low. This approach is drawn on figure 2. The second method is meant for handling higher churn; it involves explicitly tracking the set of senders for each group via the gossip layer. Since each node in the system knows the set of senders for every group, a receiver joining a group can directly send the join using multiple unicasts to the senders of that group. The second option incurs more space and communication overhead in the gossip layer but is more scalable in terms of churn and system size.

Switching between these two options can be done by a human administrator or automatically by a designated node, such as the mapping module, simply by observing the rate of membership updates in the system via the gossip layer. Once again, the broadcasts or direct unicasts do not have to be reliable, since the gossip layer will eventually propagate joins throughout the system.

3.3 Kernel Multi-send System Call

Sending a single packet to a physical IPMC group is cheap since the one-to-many multiplexing is done on a lower level by routing or switching hardware in the network. However, when IPMC resources are exhausted, the group-address mapping in MCMD will map a logical IPMC group to a set of unicast addresses corresponding to its members. Thus a single `sendto()`-call at the interface would produce a series of sends at the library and kernel level of identical packets to a number of physical addresses. We modified the kernel to help alleviate the overhead caused by context-switching during the list of sends. We implemented a *multi-send* system call on the Linux 2.6.24 kernel with a `sendto()`-like interface that sends a message to multiple destinations.

4 Optimizing Resource Use

Beyond making IPMC controllable and hence safe, the MCMD incorporates a further innovation. We noted that our goal is to optimize the limited use of IPMC addresses. Such optimization problems are often hard, and indeed the optimization problem that arises here we have proven to be NP-complete (details omitted for brevity). Particularly difficult is the problem of mapping multiple application-level groups to the same IPMC address: doing so shares the address across a potentially large set of groups, which is a good thing, but finding the optimal pattern for sharing the addresses is hard.

A *topic* is a logical multicast group. Our algorithm can be summarized as follows.

- Find and merge all identically overlapping topics into *groups*, aggregating the traffic reports.
- Sort groups in descending order by the product of the reported traffic rate and topic size.
- For each group G, assign an IPMC address to topic G, unless the global or user address quota for ≥ 3 members have been exceeded.

- Enlist all remaining users in G for point-to-point communication over unicast.

Thus a large topic with high traffic is more likely to be allocated a dedicated IPMC address. Other groups might communicate over both IPMC and point-to-point unicast for members that have exceeded their NIC IPMC capacity, and yet others might perform multicast over point-to-point unicast entirely.

5 Related Work

Brevity prevents a detailed comparison of our work with previous work of [14, 15]; key differences stem from our narrow focus on data center settings. Our mathematical framework extends that of [12], but instead of inexact channelization we investigate zero filtering.

6 Conclusion

Many major data center operators legislate against the use of IP multicast: the technology is perceived as disruptive and insecure. Yet IPMC offers very attractive performance and scalability benefits. Our paper proposes Dr. Multicast (the MCMD), a remedy to this conundrum. By permitting operators to define an acceptable use policy (and to modify it at runtime if needed), the MCMD permits active management of multicast use. Moreover, by introducing a novel scheme for sharing scarce IPMC addresses among logical groups, the MCMD reduces the number of IPMC addresses needed sharply, and ensures that the technology is only used in situations where it offers significant benefits.

References

[1] BEA Weblogic. `http://www.bea.com/framework.jsp?CNT=features.htm&FP=/content/products/weblogic/server/`, 2008.

[2] GEMSTONE GemFire. `http://www.gemstone.com/products/gemfire/enterprise.php`, 2008.

[3] IBM WebSphere. `http://www-01.ibm.com/software/webservers/appserv/was/`, 2008.

[4] JBoss Application Server. `http://www.jboss.org/`, 2008.

[5] Oracle Coherence. `http://www.oracle.com/technology/products/coherence/index.html`, 2008.

[6] Real Time Innovations Data Distribution Service. `http://www.rti.com/products/data_distribution/`, 2008.

[7] TIBCO Rendezvous. `http://www.tibco.com/software/messaging/rendezvous/default.jsp`, 2008.

[8] Deering, S. Host Extensions for IP Multicasting. *Network Working Request for Comments 1112 (August 1989)* (1989).

[9] Fei, A., Cui, J., Gerla, M., and Faloutsos, M. Aggregated multicast: an approach to reduce multicast state. *Global Telecommunications Conference, 2001. GLOBECOM'01. IEEE 3* (2001).

[10] Newman, D. Multicast performance differentiates across switches. `http://www.networkworld.com/reviews/2008/032408-switch-test-performance.html`, 2008.

[11] Rosenzweig, P., Kadansky, M., and Hanna, S. The Java Reliable Multicast Service: A Reliable Multicast Library. *Sun Labs* (1997).

[12] Tock, Y., Naaman, N., Harpaz, A., and Gershinsky, G. Hierarchical clustering of message flows in a multicast data dissemination system. In *IASTED PDCS* (2005), S. Q. Zheng, Ed., IASTED/ACTA Press, pp. 320–326.

[13] van Renesse, R., Minsky, Y., and Hayden, M. A gossip-based failure detection service. In *Middleware'98, IFIP International Conference on Distributed Systems Platforms and Open Distributed Processing* (England, September 1998), pp. 55–70.

[14] Wong, T., and Katz, R. An analysis of multicast forwarding state scalability. In *ICNP '00: Proceedings of the 2000 International Conference on Network Protocols* (Washington, DC, USA, 2000), IEEE Computer Society, p. 105.

[15] Wong, T., Katz, R. H., and McCanne, S. An evaluation on using preference clustering in large-scale multicast applications. In *INFOCOM (2)* (2000), pp. 451–460.

Antiquity: Exploiting a Secure Log for Wide-Area Distributed Storage

Hakim Weatherspoon[*]
Cornell University
hweather@cs.cornell.edu

Patrick Eaton[*]
EMC Corporation
eaton_patrick@emc.com

Byung-Gon Chun and John Kubiatowicz
University of California, Berkeley
{bgchun,kubitron}@cs.berkeley.edu

ABSTRACT

Antiquity is a wide-area distributed storage system designed to provide a simple storage service for applications like file systems and back-up. The design assumes that all servers eventually fail and attempts to maintain data despite those failures. Antiquity uses a secure log to maintain data integrity, replicates each log on multiple servers for durability, and uses dynamic Byzantine fault-tolerant quorum protocols to ensure consistency among replicas. We present Antiquity's design and an experimental evaluation with global and local testbeds. Antiquity has been running for over two months on 400+ PlanetLab servers storing nearly 20,000 logs totaling more than 84 GB of data. Despite constant server churn, all logs remain durable.

Categories and Subject Descriptors

C.2.4 [**COMPUTER-COMMUNICATION NETWORKS**]: Distributed Systems—*Peer-to-peer applications*; D.4.3 [**OPERATING SYSTEMS**]: File Systems Management—*Distributed File Systems*; D.4.5 [**OPERATING SYSTEMS**]: Reliability—*Fault-tolerance*; D.0 [**SOFTWARE**]: General—Distributed wide-area storage systems

General Terms

Reliability, Performance, Design, Experimentation, Security

Keywords

Distributed Storage System, wide-area, archival storage systems, data integrity, data durability

[*]Work done while authors were graduate students at University of California, Berkeley.

This research was supported by the National Science Foundation under Cooperative Agreement No. ANI-0225660, `http://project-iris.net/`. Hakim Weatherspoon was supported in part by an Intel Foundation PhD Fellowship.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
EuroSys'07, March 21–23, 2007, Lisboa, Portugal.
Copyright 2007 ACM 978-1-59593-636-3/07/0003 ...$5.00.

1. INTRODUCTION

Many new distributed systems—like PlanetLab [5], the Global Information Grid (GIG) [3], and GRID—are composed of machines from multiple autonomous organizations that are geographically dispersed. In these systems, servers cooperate to provide services such as persistent storage. Systems designed in this manner exhibit good scalability and resilience to localized failures such as power failures or local disasters. Unfortunately, distributed systems involving multiple, independently-managed servers suffer from new challenges such as security (including malicious components), automatic management (reliable adaptation to failure in the presence of many individual components), and instability. In PlanetLab, for example, typically less than half of the active servers are stable (available for 30 days or more) [35].

Providing secure, consistent, and available storage in these systems that exhibit extremely high levels of churn, failure, and even deliberate disruption is a challenging problem. Existing wide-area distributed storage systems, however, are not well-suited for such environments. They often support only immutable (read-only) data, do not provide consistent access to mutable (modifiable) data, do not protect and secure access to data, or are not designed for the target environment (e.g. assume fail-stop failures).

Antiquity is a distributed storage system designed to maintain data securely, consistently, and with high availability in a dynamic wide-area environment. It uses a secure log structure to maintain the integrity of stored data. It replicates data on multiple servers so that data can be retrieved later even when some replicas fail. It integrates fault-tolerance protocols to handle faults ranging from server outages to Byzantine attacks.

To test our solutions, we deployed a prototype on PlanetLab, a surprisingly volatile environment [35]. Antiquity has been running in the wide-area for over two months on 400+ PlanetLab [5] servers maintaining nearly 20,000 logs containing more than 84 GB of data. Despite the volatility of the underlying system, all logs are durable; that is, no data is lost and all logs can be read. However, tests using periodic random reads reveal that, at any given time, 6% of the logs are not modifiable since they do not have a quorum (threshold) of replicas available temporarily due to server failure on PlanetLab. All eventually become modifiable again due to Antiquity's quorum repair protocol. Antiquity's quorum repair protocol replaces lost replicas while maintaining data consistency.

Antiquity was developed in the context of OceanStore [38]. In particular, a component of OceanStore was a primary replica implemented as a Byzantine agreement process. This primary replica serialized and cryptographically signed all updates. Given this total order of all updates, the question was how to durably store and maintain the order. Antiquity's implementation of the interface and structure of a secure log assisted in durably maintaining the order over time. When data is later read from Antiquity, the secure log and repair protocols ensure that data will be returned and that returned data is the same as stored.

The contributions of this paper are as follows.

- The design and analysis of a secure log interface that can be easily implemented in a distributed, fault-tolerant fashion.
- Design and implementation of a dynamic Byzantine fault-tolerant quorum repair protocol that maintains consistency and durability in the face of recurring server failure.
- Evaluation of an operational system that combines these features and is currently running in the wide-area.

This paper presents Antiquity's design and evaluates how effectively it can maintain data. In Section 2, we present an overview of Antiquity's goals, design, and assumptions. We describe the design in detail in Sections 3 and 4. In Sections 5 and 6, we evaluate Antiquity's ability to maintain data and discuss our experiences. Section 7 describes related work; Section 8 concludes.

2. OVERVIEW

Antiquity is a generic wide-area storage system that provides secure, durable storage. It is designed to serve as the storage layer for a variety of applications such as file systems [11, 34] and backup [36, 38]. Antiquity stores application data in a secure log to protect data integrity. It simultaneously supports many applications where application state is stored as separate logs. It provides to applications a limited interface by which they can create new logs, append data to the head of an existing log, and read data at any position in the log. It guarantees fault-tolerance through replication, consistency via dynamic Byzantine fault-tolerant quorum protocols, and efficiency by aggregation.

We describe how Antiquity integrates the above design points into one cohesive system in Sections 3 and 4, but first we discuss the goals, system model, and assumptions used to design Antiquity.

2.1 Storage System Goals

The design of Antiquity was guided by the following goals.

- Integrity: Only the owner can modify the log. Any unauthorized modifications to the log, as in substitution attacks, should be prevented.
- Incremental Secure Write and Random Read Access: A client can add data to a log securely as it is created, without local buffering. Further, the client can read arbitrary blocks without scanning the entire log.
- Durability and Consistency: The log should remain accessible despite temporary and permanent server failure. The system should ensure that logs are updated in a consistent manner.
- Efficiency/Low overhead: Protocols should limit the number of cryptographic operations and the amount of communication needed across the wide area. The infrastructure should amortize the cost of maintaining data and verifying certificates when possible.

2.2 System Model

The storage system stores logs on behalf of clients. The types of clients storing data in the system can vary widely as shown in Figure 1. The client may be the end-user machine, the server in a

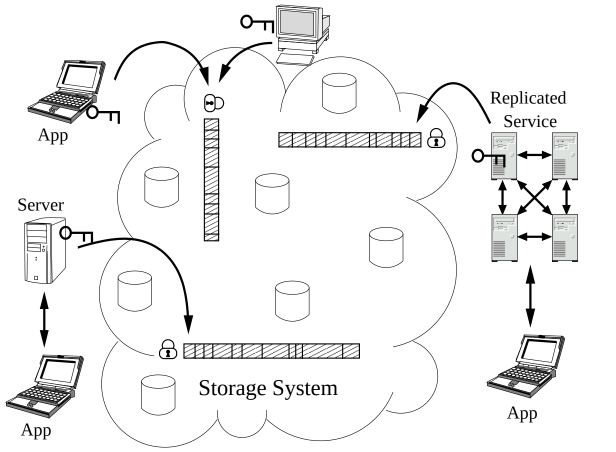


Figure 1: A log-structured storage infrastructure can provide storage for end-user clients, client-server systems, or replicated services.

client-server architecture, or a replicated service. In any case, the storage system identifies a client and its secure log by a cryptographic key pair; only principals that possess the private key can modify the log. Requests that modify the state of the log must include a certificate signed by the principal's private key. Although a log is non-repudiably bound to a single key pair, multiple instances of the principal may exist simultaneously. If multiple devices possess the same private key, then they can directly modify the same log.

Storage resources for maintaining the log are pre-allocated in chunks. When a new chunk, or *extent*, needs to be allocated, the system consults the *administrator*. The administrator authenticates the client needing to extend its log and selects a set of storage servers to host the extent. The newly-allocated portion of the log is replicated on the set of selected storage servers. To access or modify the extent, clients interact directly with the storage servers.

Applications interact with the log through a client library that exports a thin interface—`create()`, `append()`, and `read()`. To create a new log, a client obtains a new key pair and invokes the `create()` operation. The administrator authenticates the request and selects a set of servers to host the log.

After a log has been created, a client uses the `append()` operation to add data to the head of the log. The client library communicates directly with the log's storage servers to append data. The interface ensures that data is added to the log sequentially by predicating each write on the previous state of the log. If conflicting `append()` operations are submitted simultaneously, the predicate ensures at most one is applied to the log[1].

Data written to a log cannot be explicitly deleted. Instead, Antiquity supports implicit deletion based on an expiration time. The expiration time is set by the administrator when the extent is created. After the expiration time passes, the system can reclaim resources belonging to the extent. A client can prevent the expiration of an extent by extending the expiration time, though we do not discuss that feature further in this paper.

2.3 Assumptions

We assume that clients follow specified protocols, except for crashing and recovering. A malfeasant client, whether due to soft-

[1]We assume a storage server atomically handles each request. That is, a server processes requests one at a time, even though multiple requests may have been received at the same time.

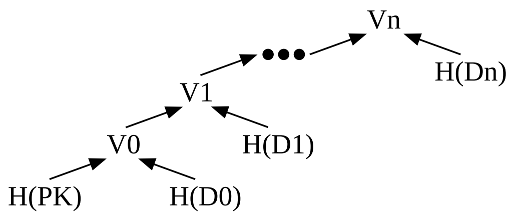


Figure 2: To compute the verifier for an extent, the system uses the recurrence relation $V_i = H(V_{i-1} + H(D_i))$. $V_{-1} = H(PK)$ where PK is a public key.

ware fault or compromised key, can prevent the system from appending data to a log. It cannot, however, affect data already stored in its log or logs belonging to other principals. If a principal's private key should be compromised, an attacker could append data to the log, but it cannot destroy data previously stored in the log. A principal can retrieve data from a log until the log's expiration time.

We assume that the administrator, tasked to select sets of storage servers to host logs [29], is trusted and non-faulty. The design, however, includes several mechanisms to mitigate the cost and consequences of this assumption. While each log uses a single administrator, different logs can use different administrators. By allowing multiple instances, the role of the administrator scales well. Second, the administrator's state can be stored as a secure log in the system. Thus, the durability of the state can be assured like any other log. Third, the state of the administrator can be cached to reduce the query load on an administrator. Finally, the administrator can be implemented as a replicated service to improve availability further.

Storage servers may exhibit Byzantine faults. We assume that, in the set of storage servers selected by the administrator to host a particular extent, a maximum threshold number of servers is faulty.

3. ANTIQUITY'S SECURE LOG

Antiquity supports a secure, append-only, log abstraction where a single log is owned by a single principal and identified by a cryptographic key pair. Only the owner of the log can `append()` to it. This narrow interface helps reduce the complexity of implementing Antiquity where consistency and durability need to be maintained efficiently. Though a single log is bound to a single cryptographic key pair, Antiquity maintains many logs associated with many separate key pairs.

Antiquity stores a log as an ordered sequence of container objects called extents. Similar to log segments in a log-structured file system [30], extents have a finite maximum size; however, each extent contains an ordered collection of variable-sized application-level blocks of data. Further, all data in an extent belongs to the same log owned by a single principal. To guard data integrity, individual log elements (blocks), whole extents, and the entire log itself are all self-verifying, which means the name of an object verifies its content.

In the following two subsections we describe the secure log structure and usage semantics. We conclude the section with a description an example file system application built on top of the secure log interface.

3.1 Secure Log Structure

A secure log is composed of two types of extents. The log head is a mutable, key-verified extent; all other extents are immutable hash-verified extents. The key-verified log head is named by a secure hash of the public key associated with the log. To verify the

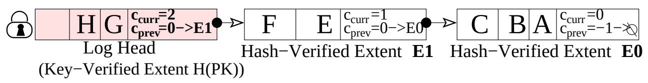


Figure 3: In two-level naming, each block is addressed by a tuple (*extent_name*, *block_name*). A block_name is simply the secure hash of the block. The extent_name for the log head is the secure hash of the public key H(PK). The extent_name for a hash-verified extent is the verifier.

contents of the log head, a server compares the data to the *verifier* included in the certificate (after confirming the signature on the certificate). A verifier is a cryptographically-secure hash that asserts the integrity of both the content and append-order of an extent.

We assume the mutable key-verified extent at the log head has a finite maximum size. When it becomes full, the system copies the content of the log head into an immutable hash-verified extent. A hash-verified extent is named by a function of the content of the extent. Specifically, the extent is named by the verifier in the extent's most recent certificate. A server can verify the integrity of a hash-verified extent by comparing an extent's contents to its name. These self-verifying techniques were made popular by the Self-certifying Read-only File System [16].

Making Extents Self-Verifying.

An extent verifier is computed from the names of the blocks in the extent using the chaining method [24] shown in Figure 2. Assume an extent contains a sequence of data blocks, D_i. Each data block is named with a secure, one-way hash function, $H(D_i)$. The verifier is computed using the recurrence relation $V_i = H(V_{i-1} + H(D_i))$, where $+$ is the concatenation operator. We bootstrap the process by defining V_{-1} to be a hash of the public key that signs the extent's certificate. This convention ensures that the names of extents owned by different principals do not conflict.

Creating verifiers in this manner has several advantages. When a block is appended to the log, the client can compute the verifier incrementally. This means it must hash only the new data, not all data in the log, to compute the running verifier. Additionally, a given verifier can be produced by only one particular sequence of `append()` operations. Thus, chaining creates a verifiable, time-ordered log recording data modifications. Furthermore, requiring the latest verifier as a predicate in subsequent `append()` operations assures servers maintain a consistent state of the log. Finally, when the log head is copied from a key-verified extent to a hash-verified extent, the verifier can be used as the new hash-verified name without modification.

Two-Level Naming.

To provide random access to any element in the log, Antiquity implements *two-level naming*. In two-level naming, each block is addressed not by a single name, but by a tuple. The first element of the tuple identifies the enclosing extent; the second element names the block within the extent. Retrieving data from the system is a two-step process. The system first locates the enclosing extent; then, it extracts individual application-level blocks from the extent. Both blocks and extents are self-verifying. Figure 3 illustrates two-level naming.

Two-level naming introduces added complexity in computing the address of a block of data. When an application writes a block to the log, the block is stored in the mutable extent at the head of the log. Because the log head is a mutable extent, the system cannot know the name of the hash-verified extent where the block will eventually and permanently reside.

Interface for Aggregation:

status	=	*create*(H(PK), cert);
status	=	*append*(H(PK), cert, predicate, data[ ]);
status	=	*snapshot*(H(PK), cert, predicate);
status	=	*truncate*(H(PK), cert, predicate);
status	=	*put*(cert, data[ ]);
status	=	*renew*(extent_name, cert);
cert	=	*get_cert*(extent_name);
data[]	=	*get_blocks*(extent_name, block_name[ ]);
extent	=	*get_extent*(extent_name);
data	=	*get_head*(extent_name);
mapping	=	*get_map*(extent_name);

Table 1: To support aggregation of log data, we use an extended API. A log is identified by the hash of a public key ($H(PK)$). Each mutating operation must include a certificate. The `snapshot()` and `truncate()` operations manage the extent chain; the `renew()` operation extends an extent's expiration time. The `get_blocks()` operation requires two arguments because the system implements two-level naming. The `get_*`(extent_name) **operations return either the entire extent or items stored within an extent such as the certificate, mapping, or various blocks (e.g. head). The** `extent_name` **is either H(PK) for the log head or verifier for hash-verified extents.**

To resolve this problem, each extent is assigned an integer corresponding to its position in the chain. When data is appended to the log, the address returned to the application identifies the enclosing extent by this counter. Each extent records the mapping between counter and permanent, hash-verified extent name for the previous extent. Both the mapping to the previous extent and position in the extent chain are stored in a metadata block (first block) within each extent; a call to `get_map()` returns the mapping (extent_counter, extent_name) of the previous extent (e.g. `get_map(E1)` in Figure 3 returns mapping (0, E0)).

Aggregating blocks into extents and extents into a log improves the system's efficiency in several ways. First, breaking a log into extents enables servers to intelligently allocate space for extents. Extents have a maximum size while the log itself can grow to be arbitrarily large. Second, extents decouple the infrastructure's unit of management from the client's unit of access. As a result, the storage infrastructure can amortize management costs over larger collections of data. Third, two-level naming reduces the query load on the system because clients need to query the infrastructure only once per extent, not once per block. Assuming data locality—that clients tend to access multiple blocks from an extent—systems can exploit the use of connections to manage congestion in the network better. Finally, clients writing multiple blocks to the log at the same time need only to create and sign a single certificate.

3.2 Using a Secure Log

To interact with the log, a client relies on a library that communicates with Antiquity using the interface shown in Table 1. The interface extends the `create()`/`append()` interface used by applications to support extents. All mutating operations require a certificate signed by the client for authorization. The certificate includes the verifier of the new version of the extent. The interface ensures that updates are applied sequentially by predicating each operation on the previous state of the extent. Upon completion of the operation, the certificate is stored with the extent. The `snapshot()` and `truncate()` operations help manage the chain of extents. The `renew()` operation extends the expiration time of an extent; we will not discuss `renew()` further.

Certificate contents:

verifier	token that verifies contents of log
num_blocks	the number of blocks in the container
size	the size of data stored in the container
seq_num	certificate sequence number
timestamp	creation time of certificate
ttl	time the certificate remains valid

Table 2: A certificate is contained within each operation and stored with each log. It includes fields to bind the log to its owner and other metadata fields.

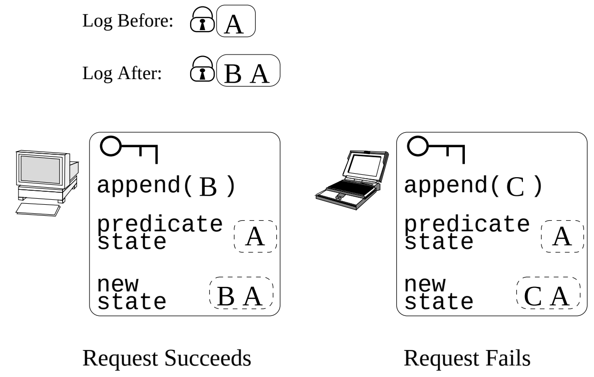


Figure 4: The interface ensures that data is added to the log in a sequential fashion by predicating each write on the previous state of the log. If conflicting `append()` operations are submitted simultaneously, the predicate ensures at most one is applied to the log, leaving the log in a consistent state.

Two of the operations enumerated in Table 1—`create()` and `snapshot()`—create new replicas. Each of these operations requires that the system contact the administrator for a configuration, set of servers, to host the new replicas. The most common operation, `append()`, does not require any interaction with the administrator.

Adding Data via `append()`.

To `append()` data to the log, a client creates a request and submits it to the storage servers. A request has three arguments: the predicate is a verifier that securely summarizes the current state of the log, the new data to append to the log, and a new certificate that includes a new verifier and new sequence number. The verifier in the certificate summarizes the next state of the log after appending data. The sequence number is a monotonically increasing number. Table 2 shows the contents of a certificate.

When a server receives an `append()` request, it determines if a request succeeds or not. It performs several checks using local knowledge. The certificate contained in the request must include a valid signature. Also, the predicate verifier contained in the request must match the current state of the log recorded by the storage server. Additionally, the verifier in the certificate must match the new verifier after appending new data to the log. Further, the sequence number in the certificate must be greater than the one currently stored. If these conditions are met, the server writes the new data to the log on its local store and returns success to the client. Otherwise, the request is rejected and failure is returned. A client receiving a failure response would need to update its local state (via `get_cert()`, `get_head()`, and `get_blocks()`) and submit a new `append()` request based on updated state.

If conflicting `append()` operations are submitted simultaneously, the predicate ensures at most one is applied to the log leaving the log in a consistent state. For example, in Figure 4, a storage server applies a workstation's request even though a laptop simultaneously submitted a (conflicting) request. As a result of the order the server handles the requests, the workstation's predicate matches the state of the log and the request succeeds. Success is returned to the workstation. However, the laptop's predicate does not match, the request fails, and failure is returned. Since the laptop's request failed, it would need to update its local state and submit a new `append()` request based on updated state.

Reading Data via `get_blocks()`.
To read data, the client library first accesses the mappings stored in the log to determine the name of the extent holding the data. It then uses the `get_blocks()` operation to retrieve the requested blocks from that extent. To accelerate the translation between counter and extent name, the client library caches the immutable mappings. Also, as an optimization, each extent contains not just the mapping for the previous extent, but a set of mappings that allow resolution in a logarithmic number of lookups.

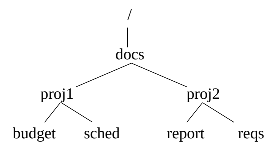
Managing the Extent Chain.
The chain of extents is managed via the `snapshot()` and `truncate()` operations. To prevent the extent at the log head from growing too large, the client library converts the log head to hash-verified form using the `snapshot()` operation. If the system must copy the extent to a new storage server, the transfer occurs directly between storage servers without client interaction. After data has been copied to a hash-verified extent, the library uses the `truncate()` to reset the log head and point to the previous extent created via `snapshot()`. While `snapshot()` and `truncate()` are typically used together, we have elected to make them separate operations for ease of implementation. Individually, each operation is idempotent, allowing the library to retry the operation until successful execution is assured. Further, each operation requires a predicate that prevents conflicting concurrent changes. The library uses the `append()`, `snapshot()`, and `truncate()` sequence to add more data to the log.

3.3 Example Application: A Versioning File System

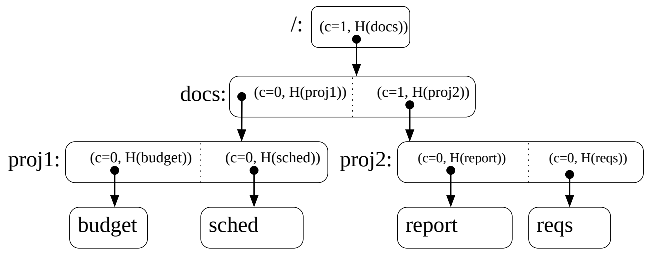
To demonstrate the use of the secure log interface, consider the implementation of a versioning file system application. Figure 5(a) shows a sample file system to be stored. We ignore inodes for this example and assume that files and directories are stored as a single block. The application translates the file system into a Merkle tree [31] where the secure pointer to a child file or directory is the (extent_counter, block_name) tuple. This file system structure is similar to others [11, 38, 36] and was implemented on top of a secure log interface in less than a week by one graduate student [15]. We illustrate the state of the log after initially archiving the file system (Figure 5) and after modifying two files (Figure 6).

To archive the file system, the application first creates a secure log using the `create()` operation. `create()` initializes the map block (first block of the log head) with the values $c_{\text{curr}} = 0$ and $c_{\text{prev}} = -1 \rightarrow \phi$ (current extent_counter and mapping to previous extent, respectively). Next, the application traverses the file system in a depth first manner calling `append()`, `snapshot()`, and `truncate()`. In particular, the application calls `append(budget, sched)` and records the verifiable pointers—($c = 0$, H(`budget`)) and ($c = 0$, H(`sched`)), respectively—in the `proj1` directory. Then, it calls `append(proj1)` and records the verifiable pointer in the `docs` directory ($c = 0$, H(`proj1`)). Similarly, the application calls `append(report, reqs)` and records the verifiable pointers in the `proj2` directory. However, before calling `append(proj2)`, the client library calls `snapshot()` which creates a hash-verified extent with the extent_name $E0$. The hash-verified extent mirrors the log head. The client library then calls `truncate()` which removes all the blocks from the log head and updates the mapping block by incrementing the current extent_counter to $c_{\text{curr}} = 1$ and setting the mapping of the previous extent to $c_{\text{prev}} = 0 \rightarrow E0$. The application, then, continues by calling `append(proj2)` and records the verifiable pointer ($c = 1$, H(`proj2`)) in the `docs` directory. Notice that the extent_counter is $c = 1$ instead of $c = 0$. Finally, the application calls `append(docs)`, records the verifiable pointer in the `/` directory, and calls `append(/)`. Figures 5(b) and 5(c) show the resulting Merkle tree and secure log, respectively. In a similar fashion, Figures 6(a) and 6(b) show the modified Merkle tree and secure log, respectively, after writing new versions of the `report` and `reqs` documents (`report'` and `reqs'`).

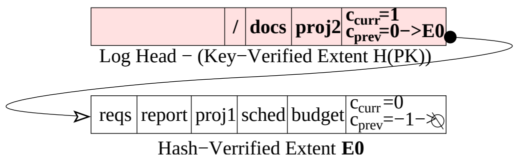
To read a particular file, the application reads the root of the file system stored at the head of the log and follows the pointers to the desired file. For example, assume the client wants to read the `sched` file. The application first calls `get_head(H(PK))` which returns the root of the file system `/'`. The root contains a verifiable pointer to the `docs'` directory ($c = 2$, H(`docs'`)). The application resolves the extent_counter $c = 2$ to the log head by calling `get_map(H(PK))`. The call returns the log head's current extent_counter value and map to the previous extent $c = 1 \rightarrow E1$ which can be cached for later use. Next, the application calls `get_blocks(H(PK), H(docs'))` which returns the `docs'`



(a) file system



(b) Merkle Tree



(c) Secure Log

Figure 5: (a) An example file system. (b) The application translates the file system into a Merkle tree. The verifiable pointers are of the form (*extent_counter*, *block_name*). (c) The Merkle tree is stored in two extents. The first extent, $E0$, is filled and has been converted to a hash-verified extent. The second extent, the log head identified by $H(PK)$, is a partially-filled key-verified extent.

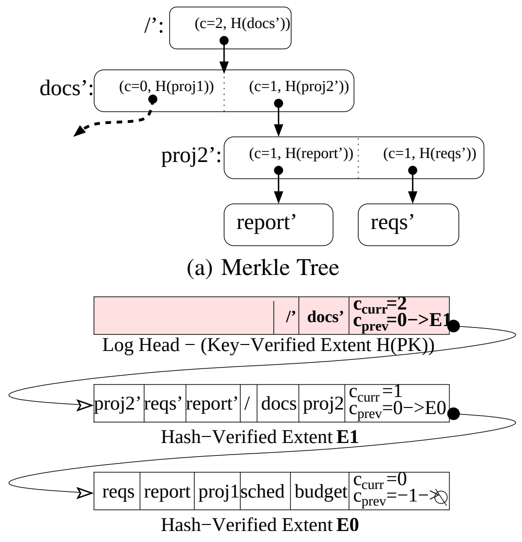


Figure 6: (a) The Merkle tree resulting from translating the updated file system. The dashed pointer indicates a reference to a block from the previous version. (b) The contents of the secure log after storing blocks of the updated file system.

directory. The `docs'` directory contains a verifiable pointer to the `proj1` directory ($c = 0$, H(`proj1`)). The application resolves the extent_counter $c = 0$ to extent_name $E0$ by calling `get_map(E1)`. The call returns the map to the previous extent $c = 0 \rightarrow E0$. Notice that the mapping from $c = 1 \rightarrow E1$ was cached from the first `get_map()` call on the log head. Finally, the application calls `get_blocks(E0,proj1)` to retrieve the `proj1` directory and `get_blocks(E0,sched)` to retrieve `sched`.

4. ANTIQUITY'S REPLICATION, CONSISTENCY, AND DURABILITY STRATEGIES

Antiquity replicates a secure log on multiple servers to provide durability. A log is durable if it persists over time. To maintain durability and ensure that progress can be made (that is, new data can be written to the log), the system must maintain consistency across replicas. The system must maintain consistency despite a variety of server and network failures and conflicting update requests. Server failures include transient failure such as reboot, permanent failure such as disk failure, and erroneous failure such as database corruption or machine compromise. Network failures include dropped connections, temporary partitions, and transmission failure such as message drop, reorder, delay, or corruption.

Antiquity employs a dynamic Byzantine fault-tolerant quorum protocol to satisfy the durability and consistency requirements. A quorum is a threshold, taking into account that some members may be faulty or malicious. For instance, Malkhi and Reiter [26] demonstrated that with self-verifying data, a configuration with $n > 3f$ servers and a quorum $q = n - f$ servers can make progress when up to f servers are faulty. In that work, configurations were static for the lifetime of the system. Martin and Alvisi [29] extended the protocol to maintain consistency in a dynamic environment. They utilize quorums to maintain two properties, *soundness* and *timeliness*, that guarantee consistency in a dynamic environment. Informally, soundness ensures data read by a client was previously written to a quorum of servers; timeliness ensures the data read is the most recent value written.

We have adapted the dynamic Byzantine quorum protocols to tolerate the failures and arbitrary behavior experienced on an environment such as PlanetLab. In particular, Antiquity creates a new configuration when a quorum in the old configuration is no longer available. We discuss the consistency via sound writes and durability via repair.

4.1 Consistency Semantics

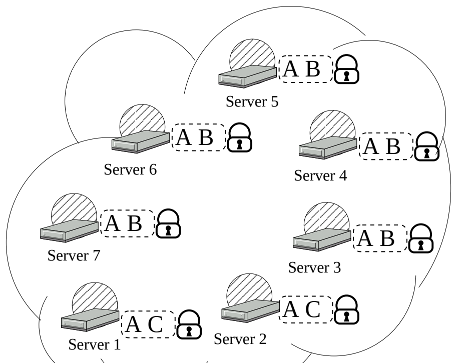
The client interacts with many replicas to complete a single operation. Operations that modify replicated state result in one of three states: sound, unsound, or undefined.

- The result of an operation is *sound* if the client receives a positive acknowledgment from a threshold of servers. A sound response means that the request succeeded and the data is durable.
- On the other hand, the result of an operation is *unsound* if the client receives a negative acknowledgment from enough servers such that positive acknowledgment from a threshold is no longer possible (e.g. sizeof(negative acks) $\geq$ sizeof(server set) $-$ threshold$+1$). A request fails if the result is unsound. The storage system does not maintain unsound results; thus, unsound writes are not durable.
- Finally, the result is *undefined* if it is neither sound or unsound. An undefined result means the client did not receive sufficient acknowledgment from servers perhaps due to network or server failure. In the case of an undefined result, a timeout occurs and the client does not know whether the request is sound or unsound.

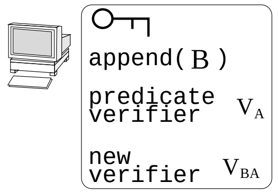
After a timeout, the client performs a `get_cert()` on all the servers and waits to receive acknowledgment from a threshold. If the state stored in the system has changed (another client updated the log), then the request is unsound. If the `get_cert()` fails to receive acknowledgment from a threshold of servers, then the client may trigger a repair audit that will determine the latest consistent state of the log (described in the next section). The client continually sends the request, reads the state of the system, then triggers a repair audit until the request is either sound or unsound.

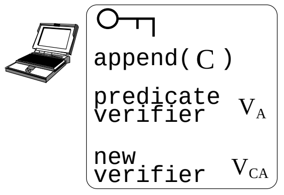
4.2 Consistency Example

Figure 7 illustrates the notions of sound, unsound, and undefined writes. Assume a log is replicated on seven servers. A threshold required for consistency and a sound response is five positive acknowledgments. The number required for an unsound response is three negative acknowledgments (total minus a threshold plus one, $7 - 5 + 1 = 3$). The initial value stored on all the log replicas is A. Further, assume two clients, a workstation and laptop, simultaneously submit conflicting operations. The workstation attempts to append the value B and receives five positive acknowledgments and two negative, thus the response is sound since a threshold acknowledged positively. The laptop, on the other hand, attempts to append the value C and receives five negative acknowledgments and two positive, thus the response is unsound. With this scenario, the storage system should maintain the workstation's appended value B over time. Furthermore, in the above example, if the workstation receives one less positive acknowledgment (four instead of five), possibly due to network transmission error, then the result would be undefined and timeout. The workstation could read the latest replicated state of the secure log, trigger a repair audit that will repair the distributed secure log if necessary, and resubmit the request until it receives sufficient server acknowledgment.



(a) Storage System





(b) Request Succeeds (c) Request Fails

Figure 7: Semantics of a Distributed Secure Log. (a) A secure log with the value *A* is initially replicated on to seven servers. In (b), a workstation attempts to `append()` the value *B*, predicated on *A* already being stored. The result of the request is sound since it reaches a threshold of servers (servers 3-7). In (c), a laptop, which possess the same private key as the workstation, simultaneously attempts to `append()` value *C*, predicated on *A* already being stored. The result of the request is unsound since the predicate fails on a threshold servers. Note that the two servers (server 1-2) apply *C* since the predicate matches local state. However, the system should return value *B* in any subsequent reads.

Alternatively, if both requests received unsound responses (e.g. both received three negative acknowledgments), then the log replicas would be in an inconsistent state since a threshold of the log replicas state do not agree. When the log replicas are in an inconsistent state, new data cannot be added to a threshold of the log replicas. When no progress can be made, the replicas need to be *repaired* to a consistent state. The quorum repair protocol is discussed next.

4.3 Quorum Repair

The repair protocol restores log replicas to a consistent state such that the latest sound write is the last write stored by a threshold of log replicas. It may be used when a client cannot make progress because replicas of the log are in an inconsistent state or a quorum is not available due to server failures. Figure 8 shows the repair process.

When a storage server receives a *repair audit*, it attempts to read the latest replicated state (latest sound write) from the other servers in the configuration. If a quorum responds and the data is in a consistent state, the storage server takes no action. If, however, a quorum does not respond or the replicas are in an inconsistent state (wedged), then the storage server will create a repair request, record it in local stable storage, and submit it to the administrator. If the server observes that it already stores a signed repair request

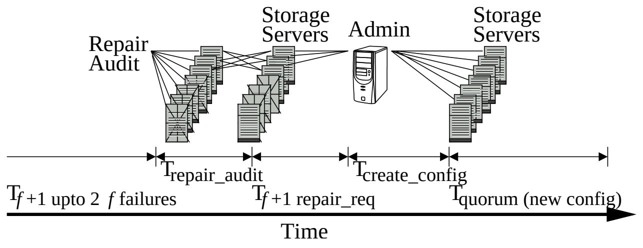


Figure 8: When a storage server believes that repair is needed, it sends a request to the administrator. After the administrator receives at least $f+1$ requests from servers in the current configuration, it creates a new configuration and sends message to servers in the set. The message describes the current state of the log; storage servers fetch the log from members of the previous configuration.

(a) **Soundness proof contents**

cert	certificate
config	configuration
ss_sigs[]	$2f+1$ or more signatures
	<H(cert+config)>$_{ss\_priv}$

(b) **Configuration contents**

object_id	cryptographically secure name of object
client_id	hash of client's public key: H(PK)
ss_set[]	set of storage servers: set of H(ss_PK)
f	fault servers tolerated
seq_num	configuration sequence number
timestamp	creation time of configuration
ttl	time the configuration remains valid

Table 3: (a) A soundness proof can be presented by any machine to any other machine in the network to prove that a write was sound. To provide this guarantee, the proof contains a set of q storage server signatures over an append's certificate (Table 2) and the storage configuration (Table 3(b)). (b) A configuration defines a set of storage servers that maintain a replicated log.

on its local disk, it will forward the same request to the administrator. Once a storage server is in the repair state, it does not accept updates until a new configuration is created.

Martin and Alvisi demonstrated that when the administrator receives $2f+1$ repair requests, it can create a new configuration to host the log while maintaining consistency (the latest sound write) between the old and new configurations [29]. Servers in the new configuration fetch the state from servers in the previous configuration. The administrator can reduce the amount of data that must be transferred during repair by retaining servers across configurations. After acquiring a copy of the log, a storage server in the new configuration responds to the administrator with a signature over the certificate (from the latest sound write) and the new configuration. The repair protocol is done after the administrator receives a quorum of responses from servers in the new configuration.

The Martin and Alvisi protocol can invoke a repair protocol when a quorum of servers is available. This, however, means that repair cannot be initiated when it is needed most—when less than a quorum of servers are available. Essentially, a quorum in the old configuration is required to agree to trigger a repair protocol that will create a new configuration. We adapted the protocol to allow repair to be triggered when less than a quorum is available while still maintaining consistency.

To ensure that no successful (sound) writes are lost during repair when less than a quorum is available, we base the repair protocol on a data structure called a *soundness proof*. Table 3(a) shows the contents of a soundness proof. A soundness proof includes a certificate (Table 2), a configuration (Table 3(b)), and a quorum q of server signatures over a hash of the certificate and configuration. It can be stored by and presented to any server as proof that a write was sound.

The new protocol is similar to the old, except the new protocol requires each server to store a soundness proof for successful operations and include the latest soundness proof in a repair request. We describe the base write protocol that creates the soundness proof below. The administrator then uses the latest soundness proof from the received repair requests to create a new configuration and initialize the configuration to the latest sound write.

To create soundness proofs required for repair, the base write protocol works as follows. There are two rounds; however, the second round is often sent with a subsequent operation. The client library does not report success to the application until the second round completes successfully. First, a client submits a request to the storage servers. When a storage server receives the message, it checks the request against its local state. If the request satisfies all conditions, the server stores the data to non-volatile storage and responds to the client with a signed positive acknowledgment. The client combines signed positive acknowledgments from a quorum of servers to create a soundness proof. Next, in the second round, the client sends the soundness proof to the servers, often as part of a subsequent operation. Each server stores the soundness proof to a stable storage and responds to the client. The client can be certain the log has been written successfully after sending the soundness proof to all servers and receiving responses from a quorum of servers [45].

4.4 Utilizing Distributed Hash Table (DHT) Technology for Data Maintenance

Antiquity uses distributed hashtable (DHT) technology to underly and connect the storage servers. It uses the DHT as a distributed directory; that is, the DHT does not store data, but rather it stores pointers that identify servers that store the data. A distributed directory provides a level of indirection that allows flexible data placement which can increase the durability and decrease the cost of repairing a given replica [8, 44]. The storage servers use the distributed directory to publish and locate extents and other storage servers. The storage servers also use the DHT in the traditional manner to cache soundness proofs to ensure they are available for all interested parties.

Antiquity also relies on the DHT to help monitor server liveness to determine when repair is necessary. Using a DHT to monitor the liveness of each extent separately is not efficient. Instead, Antiquity uses the DHT to monitor server availability and uses that metric as a proxy for extent availability. To monitor server availability, Antiquity periodically broadcasts a heartbeat message through a spanning tree defined by the DHT's routing tables [8]. A monitoring node receives liveness information from each node with a frequency depending on its distance in the spanning tree. If it fails to receive an expected heartbeat. it sends a repair audit.

Each server in Antiquity also serves as a *gateway*. A gateway accepts requests from a client and works on behalf of that client, determining the configuration to handle the request and multicasting the request to the appropriate storage servers. The use of a gateway lowers the bandwidth requirements of the client. Because all requests are signed and all data is self-verifying, inserting the gateway in the path between the client and the storage servers does not affect security. If the client believes a failure is due to a faulty gateway, it can resend the request through a different gateway. To make the soundness proof available to storage servers earlier, the gateway combines responses from the storage servers to create a proof and publishes that proof in the DHT.

Gateways perform other tasks to reduce load on the administrator. Gateways propose configurations; the administrator needs only to verify the configuration before signing it. Currently, Antiquity uses neighbor lists from the underlying DHT to determine configurations. To limit the number of configuration queries that an administrator must handle, other machines in the system can cache valid configurations. The administrator forwards its message to the gateway that handles the new configuration request, and the gateway multicasts the message to servers in the new configuration.

4.5 Discussion

Maintaining the consistency of data replicated across the wide-area is a challenging endeavor. Using a secure log structure and interface, however, significantly reduces the complexity of the design.

1. Most of the log is immutable and stored in hash-verified extents. The order and data integrity of those extent replicas are immediate—the extent name verifies both the order and content of a hash-verified extent. It is not possible for any server to corrupt a hash-verified extent in an undetectable manner.

2. The log head is the only extent in each log that is mutable and key-verified. The order and data integrity of the log head can be verified using the verifier contained in the certificate. This verifier ensures the order and data integrity of the entire log. There is only one sequence of appends that results in a particular verifier. The verifier provides a "natural" predicate that can be used to ensure the consistency of a log. Each storage server checks that the predicate verifier matches local state before applying any operation.

3. The secure log structure reduces the complexity of maintaining sound writes over time. Storage servers need to maintain only the latest soundness proof because all previous writes contribute to the verifier of the current state of the log. During a transient failure, a server needs only to retrieve soundness proofs from other servers. Once the server determines the latest sound write, it can then fetch any blocks that it is missing from an up-to-date server.

In summary, the secure log structure and its interface simplify the Antiquity design by reducing complexity of managing integrity and consistency.

5. EVALUATION

In this section, we evaluate the Antiquity design using a prototype running on PlanetLab and a local cluster. We focus our evaluation on the primitive operations provided by the storage system, but we also describe our experiences with a versioning archival back-up application.

5.1 Experimental Environment

The Antiquity prototype is written in Java using an event-driven programming style. It uses the Bamboo distributed hashtable [39] to locate storage servers and extents.

We are currently running two separate Antiquity deployments. Both deployments are configured to replicate each extent on a configuration of seven storage servers. Thus, each deployment can

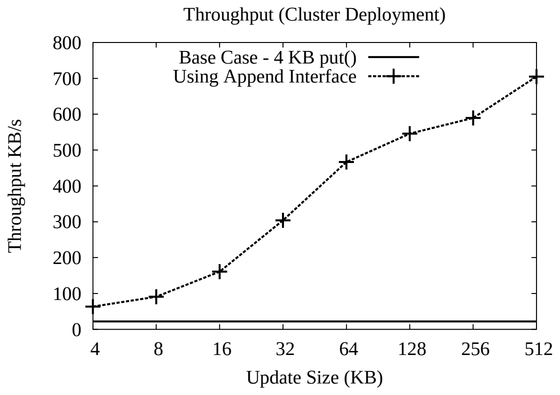


Figure 9: Aggregation increases system throughput by reducing computation at the client and in the infrastructure. The base case shows the throughput of a client that stores 4 KB blocks (and a certificate) using `put()` operation, as in a traditional DHT.

tolerate *two* faulty servers in each configuration. Both deployments are hosted on machines shared with other researchers, and, consequently, performance can vary widely over time.

The first deployment runs on 60 nodes of a local cluster. Each machine in the *storage cluster* has two 3.0 GHz Pentium 4 Xeon CPUs with 3.0 GB of memory and two 147 GB disks. Nodes are connected via a gigabit Ethernet switch. Signature creation and verification routines take an average of 3.2 and 0.6 ms, respectively. This cluster is a shared site resource; a load average of 10 on each machine is common.

The other deployment runs on the *PlanetLab* distributed research test-bed [5]. We use 400+ heterogeneous machines spread across most continents in the network. While the hardware configuration of the PlanetLab nodes varies, the minimum hardware requirements are 1.5 GHz Pentium III class CPUs with 1 GB of memory and a total disk size of 160 GB; bandwidth is limited to 10 Mbps bursts and 16 GB per day. Signature creation and verification take an average of 8.7 and 1.0 ms, respectively. PlanetLab is a heavily-contended resource; the average elapsed time of the cryptographic computations can be more than 210 and 10 ms.

We apply load to these deployments using 32 nodes of a different local cluster. Each machine in the *test cluster* has two 1.0 GHz Pentium III CPUs with 1.0 GB of memory and two 36 GB disks. Signature creation and verification takes an average of 6.0 and 0.6 ms, respectively. The cluster shares a 100 Mbps link to the external network. This cluster is also a shared site resource, but its utilization is lower than the storage cluster.

Parts of the evaluation have been presented in earlier work. In particular, the cluster deployment improves upon the preliminary performance presented by Eaton et al. [14]. Together, the performance and deployment evaluations demonstrate the efficacy of a system such as Antiquity.

5.2 Cluster Deployment

In addition to serving as a tool for testing and debugging, the Antiquity deployment on the cluster also allows us to observe the behavior of the system when bandwidth is plentiful and contention for the processor is relatively low.

Figure 9 shows how aggregation improves write performance. In this test, a single client submits synchronous updates of various sizes to Antiquity. The client library translates the requests into `append()`, `snapshot()`, and `truncate()` commands.

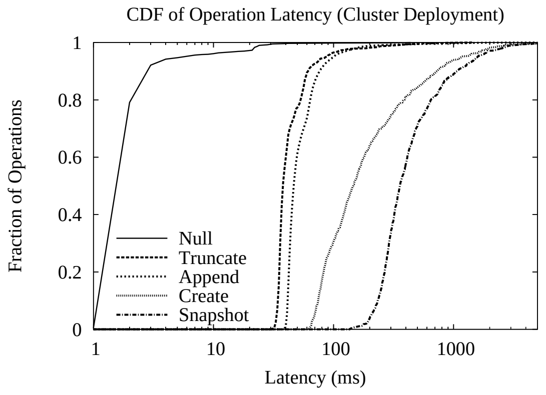


Figure 10: Different operations have widely varying latency. The latency is dependent on the amount of data that must be transferred across the network and the amount of communication with the administrator required. The latency CDF of all operations (even the `null()` RPC operation) exhibit a long tail due to load from other, unrelated jobs running on the shared cluster.

We record the write throughput observed by the client. The x-axis shows how much data the client writes with each request. The extent capacity is set to 1 MB. For comparison, we show the throughput of a client that stores data using synchronous `put()` operations with payload of 4 KB of application data, as in typical in a DHT.

Aggregation increases system throughput. At the client, aggregation reduces the cost of interacting with the system by amortizing the cost of creating and signing certificates and transmitting network messages over more data. In the storage system, aggregation reduces the number of quorum operations that must be performed to write a given amount of data to the system. The `put()` throughput is lower than `append()` operations of equivalent size because `put()` operations require the administrator to create a new configuration for each request.

Next, we measure the latency of individual operations. In this test, a single data source issues a variety of operations, including incremental writes of 32 KB using the `append()` interface. Extents are configured to have a maximum capacity of 1 MB. Figure 10 presents a Cumulative Distribution Function (CDF) of the latency of various operations. The latency of different operations varies significantly. The `append()` and `truncate()` operations are the fastest because they transfer little or no data and do not require any interaction with the administrator. The `create()` operation is slightly slower because, though it contains no application payload, it must contact the administrator to obtain a new configuration. Finally, the `snapshot()` operation is the slowest; it transfers large amounts of data and must contact the administrator to find a suitable configuration of storage servers. The latency distribution of all operations exhibit a long tail due to load from other unrelated processes running on the same machines; note, even the `null()` RPC call can take longer than one second, due to delay caused by load from unrelated jobs running on the cluster.

Table 4 decomposes the median latency into its component phases for different types of operations. Notice that interacting with the DHT consumes a significant fraction of time (`publish()` and `lookup()` are DHT operations). In particular, `append()` and `truncate()` interact with the DHT one time to `publish()` the soundness proof to support repair. However, operations that create extents (`create()` and `snapshot()`) interact with the DHT multiple times (locate/publish coordinating gateway, lookup

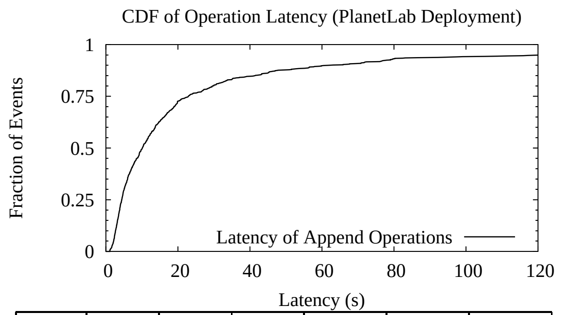
	Time (ms)			
	No Admin		Admin	
Phase	`trunc`	`append`	`cr-eate`	`snap-shot`
Phase				
T_{req}				
Signs Req	6.0	6.0	6.0	6.0
Send Req	1.8	4.2	1.8	1.8
Verify Req	0.6	1.0	0.6	0.6
`lookup()` Locations	(cached)	(cached)	13.2	13.2
`publish()` Gateway	(cached)	(cached)	7.2	7.2
subtotal	8.4	11.2	28.8	28.8
$T_{\text{create\_config}}$				
`lookup()` Neighbors	N/A	N/A	6.6	6.6
Send Config Req	N/A	N/A	1.6	1.6
Verify Config Req	N/A	N/A	0.6	0.6
Create New Config	N/A	N/A	8.2	8.2
Sign New Config	N/A	N/A	3.2	3.2
Reply w/ New Config	N/A	N/A	1.6	1.6
subtotal	0.0	0.0	21.8	21.8
T_{quorum}				
Send Req	1.8	6.6	1.8	1.8
Verify Req	0.6	1.0	1.2	1.2
Fetch Extent				98.4
Disk	4.1	5.9	4.1	61.9
`publish()` Location			7.2	62.3
Sign Result	3.2	3.2	3.2	3.2
Send Reply	1.6	1.6	1.6	1.6
Verify Replies	4.2	4.2	4.2	4.2
`publish()` Proof	7.2	7.2	63.3	63.3
subtotal	22.7	29.7	86.6	297.9
T_{resp}				
Reply w/ Proof	1.7	1.7	1.7	1.7
Verify Proof	4.2	4.2	4.2	4.2
subtotal	5.9	5.9	5.9	5.9
Total – Median	37.0	46.8	143.1	354.4
(Min)	(31.0)	(38.0)	(62.0)	(137.0)

Table 4: Measured breakdown of the median latency times for all operations. The average network latency and bandwidth between applications on the test cluster and storage cluster is 1.7 ms and 12.5 MB/s (100 Mbs), respectively. The average latency and bandwidth between applications within the storage cluster is 1.6 ms and 45.0 MB/s (360 Mbs). All data is stored to disk using BerkeleyDB which has an average latency and bandwidth of 4.1 ms and 17.3 MB/s, respectively. Signature creation/verification takes an average of 6.0/0.6 ms on the test cluster and 3.2/0.6 ms on the storage cluster. Bandwidth of the SHA-1 routine on the storage cluster is 80.0 MB/s. Finally, DHT `lookup()` /`publish()` take an average of 4.2/7.2 ms.

replica locations, publish replica location, and publish soundness proof). Furthermore, multiple `publish()` operations to the same identifier often take longer than expected since `publish()` sometimes competes with other BerkeleyDB operations for use of the disk (e.g. BerkeleyDB log cleaning).

5.3 PlanetLab Deployment

In this section, we present results from the Antiquity deployment on PlanetLab. For reasons illustrated in Figure 11, the focus of our evaluation of the PlanetLab deployment is not on its performance. That graph plots the CDF of the latency of more than 800 operations that append 32 KB of data to logs in the systems. The accompanying table reports several key points on the curve. Given the best of circumstances, the latency of an `append()` operation is one second. However, when configurations include distant or overloaded servers or bandwidth is restricted on some path, the latency increases considerably. Because of the characteristics of the PlanetLab testbed, many operations are very slow [37].



Min	25%	50%	90%	95%	99%	Max
1.08	4.41	10.2	63.1	124.5	302.1	615.9

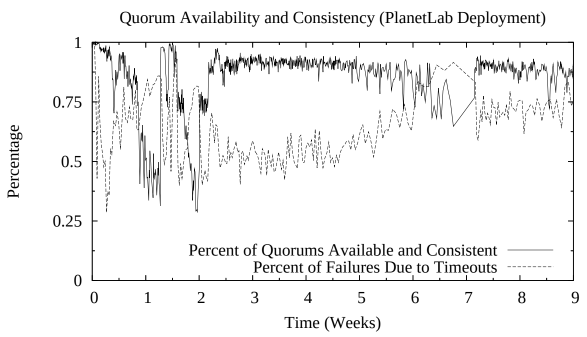
Figure 11: The latency of operations on PlanetLab varies widely depending on the membership and load of a configuration. As an example, this graphs illustrates the CDF of the latency for appending 32 KB to logs stored in the system. The table highlights key points in the curves.

Instead, with the PlanetLab deployment, we focus on how the design maintains data over time, especially as machines fail. We built a simple test application that writes logs to the system and periodically reads them to check that they are still available. Each log consists of one key-verified extent (the log head) and an average of four hash-verified extents (the number of hash-verified extents vary uniformly with an average of four). Key-verified extents vary in size uniformly up to 1 MB; all hash-verified extents are 1 MB. The average size of a log is 4.5 MB (0.5 MB log head and 4 x 1MB hash-verified extents). The test application stores 18,779 logs (18,779 log heads and 75,085 hash-verified extents) totaling 84 GB. We stopped writing new data to Antiquity because we reached the PlanetLab-enforced storage quota. After writing an extent to the system, this test application records a summary of the extent in a local database.

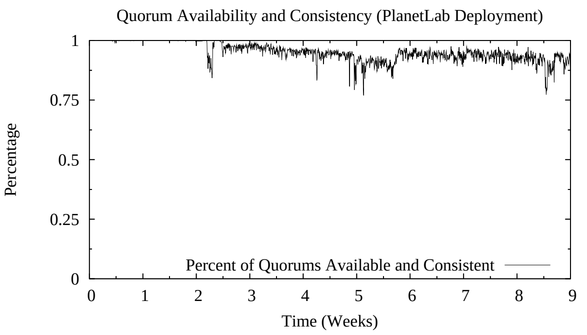
We perform various tests to measure the efficacy of the Antiquity deployment. First, we measure the percent of extents with at least a quorum of replicas available and in a consistent state in Section 5.3.1. This test measures the number of logs that can accept new writes from their owner. Next, in Section 5.3.2, we measure the cost of maintaining secure logs in terms of replicas created. In particular, we measure the average number of replicas created per unit time and the total number of replicas created. This test measures the systems ability to maintain sufficient replication levels in response to server failure.

5.3.1 Quorum Consistency and Availability

We compute quorum availability and consistency in two different ways. The first approach uses a test application that periodically reads a random extent. Every 10 seconds, the tester selects a random entry from the database and attempts to contact a quorum of the servers hosting that extent. It reports whether it was able to reach a quorum of servers. It also verifies that the replicas are in a consistent state and that state matches what was written. The second approach uses a server application availability trace, server database log, and extent configuration to compute the metrics. The first approach is an experimental method that includes intermittent effects such as server load and network performance. The sec-



(a) Periodic Application Read



(b) Server Application Availability Trace

Figure 12: Quorum Consistency and Availability. (a) Periodic reads show that 94% of quorums were reachable and in a consistent state. Up to 90% of failed checks are due to network errors and timeouts. (b) Server application availability trace shows that 97% of quorums were reachable and in a consistent state. This illustrates the increase in performance over (a) where timeouts reduced the percent of measured available quorums.

ond approach ignores such experimental effects and, instead, uses server availability to compute the metrics.

Figure 12(a) shows the percentage of quorums that were available and consistent over time, as measured by the first approach over a two-month period. The top curve shows the percentage of successful quorum checks. A software bug between week 1 through the middle of week 2 caused over half the servers not to respond to RPC requests. Periodic server application reboot temporarily masked the bug. But the performance continued to degrade until the problem was solved during the middle of week 2. A stale network file system handle prevented the test application from probing the system properly between weeks 6 and 7. Over the life of the test (including the period between week 1 through the middle of week 2 interruption), an average of 94% of checks reported that a quorum of servers was reachable and stored a consistent state. Even though, at any given time, 6% of the of the checks may not have a quorum of replicas available, later checks reveal that a quorum eventually becomes available and is consistent due to the repair protocol.

The observed availability matches computed estimates. Using a monitor on the remote hosts, we measured the average availability of machines in PlanetLab to be 90%. Note, this figure indicates that the node is up, not necessarily that the node can be reached over the network. Given that measurement, we would expect a quorum of servers to be available 94% of the time.

The lower curve on the plot shows the percentage of checks that failed due to RPC failures, network disruptions, and other timeouts. We attempt to reach a quorum through five different gateways before marking a check failed. Our measurements show that up to 90% of the failed checks may be caused by components outside of Antiquity. This percentage increases as the load on PlanetLab increases. Furthermore, the high load causes a number of Antiquity processes to be terminated due to resource exhaustion. Thus, the actual percentage of consistent quorums (shown next) is higher than the 94% measured from the application.

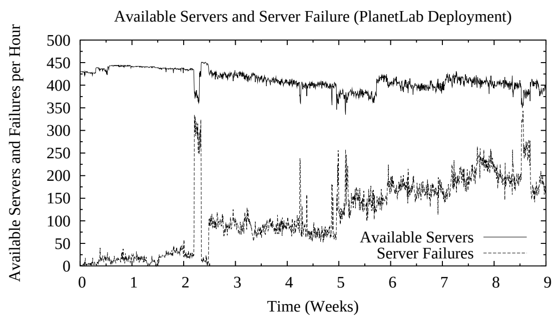


Figure 13: Number of servers with their Antiquity application available per hour. Additionally, number of servers with Antiquity application failures per hour. Most failures are due to restarting the unresponsive Antiquity instances. As a result, a single server may restart its Antiquity application multiple times per hour if the instance is unresponsive.

Figure 12(b) plots quorum availability and consistency computed by the second approach. The server application availability trace used in this approach ignores the software bug; as a result, until the middle of week 2, 100% of the extents had at least a quorum available and consistent. After the middle of week 2, however, server churn increased, tripling from 24 server failures per hour to 76. The cause for the increase in server churn is a watchdog timer that restarts a server's Antiquity application when it is unresponsive for over six minutes. Figure 13 shows the number of servers available and server failures during each hour of the test.

5.3.2 *Quorum Repair*

Antiquity's repair process maintains the availability of a quorum of servers for each extent. Figure 14 plots the cumulative number of replicas created in the PlanetLab deployment. During the period of observation, Antiquity initially created a total of 657,048 extent replicas (each of the 93,864 extents were initially created with 7 replicas). The replicas initially accounted for 577 GB of replicated storage (84 GB of unique storage).

In order to maintain the availability of a quorum of servers, Antiquity triggers the `repair()` protocol when less than a quorum of replicas are available. Each `repair()` replaces at least three replicas since that is the least number of unavailable servers required to trigger `repair()` with $f = 2$. The deployment experienced an average of 114 (Antiquity application) failures per hour. In response to failures, Antiquity triggered `repair()` 92 times per hour. As the number of unavailable servers accumulated, nearly every failure triggered a `repair()`. Each `repair()` replaced an average of four replicas. As a result, Antiquity created a total of 653,028 replicas due to `repair()` during the two month period of observation. Repair required less than 0.31 KB/s (320 Bps) per server. Coupled with maintaining the availability and consistency of up to 97% of the extents, this demonstrates that Antiquity is capable of maintaining sufficient replication levels in response to

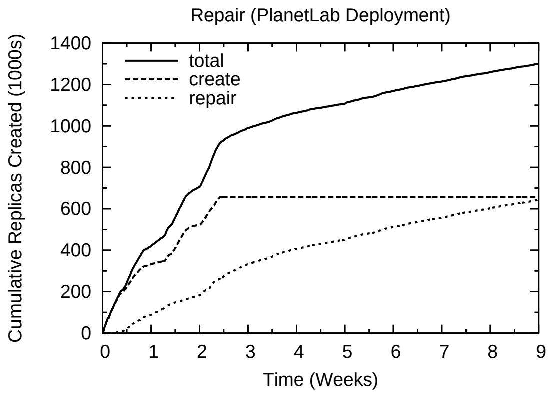


Figure 14: Number of replicas created over time due to storing new data and in response to failure.

server failure.

Another metric of concern is the time to repair an extent with less than a quorum of replicas available. On average, when a server failed, it took the system 30 minutes to detect and classify the server as failed (value of timeout) and three hours to replace replicas stored on the failed server with less than a quorum of remaining replicas available. Once repair completed for a particular extent, at least a quorum of servers were again available.

5.4 A Versioning Back-up Application

Finally, we have built a versioning back-up application that stores data in Antiquity. The application translates a local file system into a Merkle tree as shown in Figure 5 and used in similar previous systems [32, 11]. The application records in a local database when data was written to the infrastructure. It checks the local database before archiving any new data. This acts as a form of copy-on-write, reducing the amount of data transmitted. The file system we implemented is described fully in [15].

We stored the file system containing the Antiquity prototype (source code, object code, utility scripts, etc.) in PlanetLab. The file system is recorded in 15 1-MB extents. The system has repaired two of the 15 extents while ensuring both consistency and durability of the file system.

6. EXPERIENCE AND DISCUSSION

Putting it all together, Antiquity maintained 100% durability and 97% quorum availability of 18,779 logs broken into 93,864 extents. Reflecting on our experience, the structure of the secure log made this an easier task for three reasons.

First, maintaining the integrity of a secure log is easier than other structures since the verifier for the log (and each extent) defines the order of appends and cryptographically ensures the content. In particular, there is only one sequence of appends that results in a particular verifier. This verifier is used as a predicate to ensure that new writes are appended to the log in a consistent fashion. Furthermore, this verifier is used by the storage system to ensure that each replica stores the same state. In the deployment, this verifier was a critical component used to ensure the consistency and integrity of the log and all of its extents. Furthermore, it is cheap to compute, update, and compare.

Second, a storage system that implements a secure log is a layer or middleware in a larger system. The secure log abstraction bridges the storage system and higher level applications together. In fact, the secure log interface implemented by Antiquity is a result of breaking OceanStore into layers. In particular, a component of OceanStore was a primary replica implemented as a Byzantine Agreement process. This primary replica serialized and cryptographically signed all updates. Given this total order of all updates, the question was how to durably store and maintain the order? Furthermore, what should be the interface to this storage system? An append-only secure log answered both questions. The secure log structure assists the storage system in durably maintaining the order over time. The append-only interface allows a client to consistently add more data to the storage system over time. Finally, when data is read from the storage system at a later time, the interface and protocols ensure that data will be returned and that returned data is the same as stored.

Finally, self-verifying structures such as a secure log lend themselves well to distributed repair techniques. The integrity of a replica can be checked locally or in a distributed fashion. In particular, we implemented a quorum repair protocol where the storage server replicas used the self-verifying structure. The structure and protocol provided proof of the contents of the latest replicated state and ensured that the state was copied to a new configuration.

7. RELATED WORK

Antiquity builds on the experience of many prior systems.

7.1 Logs

The log-structured file system [30] used a log abstraction to improve the performance of local file systems. Zebra [20] uses a similar abstraction to improve the performance of a network file system. Schneier and Kelsey [41] and SUNDR [24] demonstrated how to use a secure log to store data on an untrusted remote machines. They do not address how to replicate the log.

7.2 Byzantine Fault-Tolerant Services

Byzantine fault-tolerant services have been proposed to help meet the challenges of unsecured, distributed environments. FarSite [2], OceanStore [38], and Rosebud [40] built distributed storage systems using Byzantine fault-tolerant agreement protocols [21, 7]. Abd-El-Malek et al. [1], Goodson et al. [18], the COCA project [46], Fleet [27], and Martin and Alvisi [29] built reliable services using Byzantine fault-tolerant quorum protocols [26]. Martin and Alvisi define a protocol that allows the configuration to be changed with the help of an administrator. HQ [10] has a hybrid structure that is similar to Antiquity's use of an administrator. During normal operation, clients interact using an efficient Byzantine quorum voting protocol. Under write contention or failures, a separate Byzantine agreement (or administrator for Antiquity) is invoked to resolve conflicts and possibly reconfiguring the set of servers. None of these systems trigger reconfiguration reactively.

7.3 Wide-area Distributed Storage Systems

Many researchers have used distributed hash table (DHT) technology to build wide-area distributed storage systems. Notable examples are Carbonite [8], CFS [11], Glacier [19], Ivy [34], PAST [13], Total Recall [6], and Venti [36]. Carbonite and Total Recall optimize for the wide-area by reducing the number of replicas created due to transient failures. Glacier uses aggregation to reduce storage overheads. Ivy uses a log structure similar to Antiquity; however, the log is block-based instead of extent-based. In particular, to grow the log, Ivy creates new blocks that incur high overheads since each block is individually maintained; whereas,extents reduce these overheads since Antiquity supports aggregation via a secure-append operation. None of these systems implement a Byzantine fault-tolerant consistency algorithm. Chain Replication [44] and Etna [33] both implement consistency protocols, but assume fail-stop failures.

7.4 Replicated Systems

Systems like GFS [17], Harp [25], Petal [22], Frangipani [43], and XFS [4] replicate data to reduce the risk of data loss. GFS and XFS also use aggregation. These systems target well-connected environments.

Distributed databases [12], the Amoeba distributed operating system [42], the Myriad online disaster recovery system [23], and EMC storage systems [9] use the wide-area replication to increase durability. Myriad and EMC replicate data between a primary and backup site. Wide-area recovery is initiated after site failure; single disk failure is repaired locally with RAID.

7.5 Digital Libraries

Digital libraries such as LOCKSS [28] preserve journals and other electronic documents. The documents are read-only and cannot be updated. They are replicated at many sites for durability. Many documents in a digital library do not have an "owner"; thus, the system uses voting to maintain the integrity. Antiquity, in contrast, assumes that all logs have an owner and only the owner can make changes to the log.

8. CONCLUSION

We described the design of the Antiquity wide-area distributed storage system. The design is tailored for dynamic environments where server failure is common. Antiquity combines a secure append-only log interface with dynamic Byzantine fault-tolerant quorums and quorum repair to maintain data integrity, consistency, and durability. Evaluation of a prototype running on PlanetLab demonstrates that the design is effective. The prototype stores 84 GB of data on 400+ servers under constant churn and all logs remain durable. At any given moment, however, 6% of the logs do not have a quorum (threshold) of replicas available temporarily due to server failure on PlanetLab. All eventually become available—Antiquity successfully repaired all quorums to an available and consistent state with its quorum repair protocol.

9. AVAILABILITY

The Antiquity source code is published under the BSD license and is freely available http://antiquity.sourceforge.net.

10. ACKNOWLEDGMENTS

We would like to thank Ken Birman whose comments and advice have greatly improved the presentation of this work. We are grateful to Anthony Joseph who has provided valuable input on the design and implementation of Antiquity. Also, Robbert van Renesse and Einar Vollset have provided valuable feedback on the presentation of Antiquity. Finally, we would like thank Mike Howard for maintaining the cluster at Berkeley and all of the groups that have contributed to making PlanetLab available. Without these two testbeds, this work would not have been possible.

11. REFERENCES

[1] M. Abd-El-Malek, G. Ganger, G. Goodson, M. Reiter, and J. Wylie. Fault-scalable byzantine fault-tolerant services. In *Proc. of ACM SOSP*, Oct. 2005.

[2] A. Adya, W. J. Bolosky, M. Castro, G. Cermak, R. Chaiken, J. R. Douceur, J. Howell, J. R. Lorch, M. Theimer, and R. P. Wattenhofer. Farsite: Federated, available, and reliable storage for an incompletely trusted environment. In *Proc. of USENIX OSDI*, Dec. 2002.

[3] N. S. Agency. Global information grid (gig). `http://www.nsa.gov/ia/industry/gig.cfm`. Last accessed September 2006.

[4] T. Anderson, M. Dahlin, J. Neefe, D. Patterson, D. Roselli, and R. Wang. Serverless Network File Systems. In *Proc. of ACM SOSP*, Dec. 1995.

[5] A. Bavier, M. Bowman, B. Chun, D. Culler, S. Karlin, S. Muir, L. Peterson, T. Roscoe, T. Spalink, , and M. Wawrzoniak. Operating system support for planetary-scale network services. In *Proc. of USENIX NSDI*, Mar. 2004.

[6] R. Bhagwan, K. Tati, Y. Cheng, S. Savage, and G. Voelker. Totalrecall: Systems support for automated availability management. In *Proc. of USENIX NSDI*, Mar. 2004.

[7] M. Castro and B. Liskov. Practical Byzantine fault tolerance. In *Proc. of USENIX OSDI*, 1999.

[8] B. Chun, F. Dabek, A. Haeberlen, E. Sit, H. Weatherspoon, M. F. Kaashoek, J. Kubiatowicz, and R. Morris. Efficient replica maintenance for distributed storage systems. In *Proc. of USENIX NSDI*, San Jose, CA, May 2006.

[9] E. Corp. Symmetrix remote data facility. `http://www.emc.com/products/networking/srdf.jsp`. Last accessed February 2007.

[10] J. Cowling, D. Myers, B. Liskov, R. Rodrigues, and L. Shrira. Hq replication: A hybrid quorum protocol for byzantine fault tolerance. In *Proc. of USENIX OSDI*, Nov. 2006.

[11] F. Dabek, M. F. Kaashoek, D. Karger, R. Morris, and I. Stoica. Wide-area cooperative storage with CFS. In *Proc. of ACM SOSP*, October 2001.

[12] A. Demers, D. Greene, C. Hauser, W. Irish, J. Larson, S. Shenker, H. Sturgis, D. Swindhart, and D. Terry. Epidemic algorithms for replicated database maintenance. In *Proc. of ACM PODC*, pages 1 – 12, Aug. 1987.

[13] P. Druschel and A. Rowstron. Storage management and caching in PAST, a large-scale, persistent peer-to-peer storage utility. In *Proc. of ACM SOSP*, 2001.

[14] P. Eaton, H. Weatherspoon, and J. Kubiatowicz. Efficiently binding data to owners in distributed content-addressable storage systems. In *3rd International Security in Storage Workshop*, Dec. 2005.

[15] P. R. Eaton. *Improving Access to Remote Storage for Weakly Connected Users*. PhD thesis, EECS Department, University of California, Berkeley, January 11 2007.

[16] K. Fu, M. F. Kaashoek, and D. Mazières. Fast and secure distributed read-only file system. In *Proc. of USENIX OSDI*, Oct. 2000.

[17] S. Ghemawat, H. Gobioff, and S. Leung. The google file system. In *Proc. of ACM SOSP*, pages 29–43, Oct. 2003.

[18] G. R. Goodson, J. J. Wylie, G. R. Ganger, and M. K. Reiter. Byzantine-tolerant erasure-coded storage. Technical Report CMU-CS-03-187, Carnegie Mellon University School for Computer Science, Sept. 2003.

[19] A. Haeberlen, A. Mislove, and P. Druschel. Glacier: Highly durable, decentralized storage despite massive correlated failures. In *Proc. of USENIX NSDI*, May 2005.

[20] J. H. Hartman and J. K. Ousterhout. The zebra striped network file system. In *Proc. of ACM SOSP*, 1993.

[21] L. Lamport, R. Shostak, and M. Pease. The Byzantine Generals Problem. *ACM TOPLAS*, 4(3):382–401, 1982.

[22] E. K. Lee and C. A. Thekkath. Petal: Distributed virtual disks. pages 84–92, 1996.

[23] S. A. Leung, J. MacCormick, S. E. Perl, and L. Zhang. Myriad: Cost-effective disaster tolerance. In *Proc. of USENIX FAST*, Jan. 2002.

[24] J. Li, M. Krohn, D. Mazières, and D. Shasha. Secure untrusted data repository SUNDR. In *Proc. of USENIX OSDI*, pages 121–136, Dec. 2004.

[25] B. Liskov, S. Ghemawat, R. Gruber, P. Johnson, L. Shrira, and M. Williams. Replication in the harp file system. In *Proc. of ACM SIGOPS*, 1991.

[26] D. Malkhi and M. Reiter. Byzantine quorum systems. In *Proc. of ACM STOC*, pages 569 – 578, May 1997.

[27] D. Malkhi, M. K. Reiter, D. Tulone, and E. Ziskind. Persistent objects in the fleet system. In *DISCEX II*, 2001.

[28] P. Maniatis, M. Roussopoulos, T. Giuli, D. S. H. Rosenthal, and M. Baker. The lockss peer-to-peer digital preservation system. *ACM Trans. Comput. Syst.*, 23(1):2–50, 2005.

[29] J.-P. Martin and L. Alvisi. A framework for dynamic byzantine storage. In *Proc. of the Intl. Conf. on Dependable Systems and Networks*, June 2004.

[30] J. Matthews, D. Roselli, A. Costello, R. Wang, and T. Anderson. Improving the performance of log-structured file systems with adaptive methods. In *Proc. of ACM SOSP*, Oct. 1997.

[31] R. Merkle. A digital signature based on a conventional encryption function. pages 369–378. Springer-Verlag, 1988.

[32] S. J. Mullender and A. S. Tanenbaum. A distributed file service based on optimistic concurrency control. In *Proc. of ACM SOSP*, pages 51–62, Dec. 1985.

[33] A. Muthitacharoen, S. Gilbert, and R. Morris. Etna: A fault-tolerant algorithm for atomic mutable dht data. Technical Report MIT-LCS-TR-993, MIT Laboratory for Computer Science, June 2004.

[34] A. Muthitacharoen, R. Morris, T. Gil, and B. Chen. Ivy: A read/write peer-to-peer file system. In *Proc. of USENIX OSDI*, 2002.

[35] L. Peterson, A. B. E. Fiuczynski, , and S. Muir. Experiences building planetlab. In *Proc. of USENIX OSDI*, Nov. 2006.

[36] S. Quinlan and S. Dorward. Venti: A new approach to archival data storage. In *Proc. of USENIX FAST*, Jan. 2002.

[37] S. Rhea, B. Chun, J. Kubiatowicz, and S. Shenker. Fixing the embarrassing slowness of opendht on planetlab. In *Proc. of USENIX Workshop on Real, Large Distributed Systems (WORLDS)*, Dec. 2005.

[38] S. Rhea, P. Eaton, D. Geels, H. Weatherspoon, B. Zhao, and J. Kubiatowicz. Pond: the OceanStore prototype. In *Proc. of USENIX FAST*, 2003.

[39] S. Rhea, D. Geels, T. Roscoe, and J. Kubiatowicz. Handling churn in a dht. In *Proc. of USENIX*, June 2004.

[40] R. Rodrigues and B. Liskov. Rosebud: A scalable byzantine-fault-tolerant storage architecture. Technical Report MIT-LCS-TR-932, MIT Laboratory for Computer Science, Dec. 2003.

[41] B. Schneier and J. Kelsey. Cryptographic support for secure logs on untrusted machines. Jan. 1998.

[42] A. S. Tanenbaum, R. van Renesse, H. van Staveren, G. J. Sharp, S. J. Mullender, J. Jansen, and G. van Rossum. Experiences with the Amoeba distributed operating system. *Communications of the ACM*, 33(12):46–63, 1990.

[43] C. Thekkath, T. Mann, and E. Lee. Frangipani: A scalable distributed file system. In *Proc. of ACM SOSP*, 1997.

[44] R. van Renesse and F. B. Schneider. Chain replication for supporting high throughput and availability. In *Proc. of USENIX OSDI*, May 2004.

[45] H. Weatherspoon. *Design and Evaluation of Distributed Wide-Area On-line Archival Storage Systems*. PhD thesis, EECS Department, University of California, Berkeley, October 13 2006.

[46] L. Zhou, F. Schneider, and R. van Renesse. Coca: A secure distributed on-line certification authority. *ACM Trans. Comput. Syst.*, pages 329–368, Nov. 2002.

Fireflies: Scalable Support for Intrusion-Tolerant Network Overlays

Håvard Johansen
University of Tromsø
Norway

André Allavena
University of Waterloo
Canada

Robbert van Renesse
Cornell University
USA

ABSTRACT

This paper describes and evaluates *Fireflies*, a scalable protocol for supporting intrusion-tolerant network overlays.[1] While such a protocol cannot distinguish Byzantine nodes from correct nodes in general, *Fireflies* provides correct nodes with a reasonably current view of which nodes are live, as well as a pseudo-random mesh for communication. The amount of data sent by correct nodes grows linearly with the aggregate rate of failures and recoveries, even if provoked by Byzantine nodes. The set of correct nodes form a connected submesh; correct nodes cannot be *eclipsed* by Byzantine nodes. *Fireflies* is deployed and evaluated on PlanetLab.

1. INTRODUCTION

Network overlays provide important routing functionality not supported directly by the Internet. Such functionality includes multicast routing, content-based routing, and resilient routing, as well as combinations thereof. In recent years, Distributed Hash Tables (DHTs) have been proposed to support network overlays. While it is often straightforward to support network overlays on DHTs, this choice can be questioned. DHTs dictate routes that are not optimal [25], and DHTs are hard to secure [30]. As network overlays are starting to be deployed for critical applications, efficiency and security are becoming important attributes.

In this paper we present an alternative support structure called *Fireflies*. *Fireflies* provides each of its members with a complete view of its live peers.[2] A small subset of these peers are marked as *neighbors*. With high probability, the mesh formed by the members and their neighbor links has a diameter logarithmic in the number of live members and connects all the reachable members that are not Byzantine.

An obvious disadvantage of providing members with a view of the full membership, compared to only a partial view as provided by a DHT, is decreased scalability. Memory requirements per member will grow linearly in the number of members. Given the availability of cheap memory this is not necessarily a problem. More alarmingly, the rate of member join and leave events will likely grow linearly with the number of members as well, possibly leading to an unmanageable amount of network bandwidth or latency. Many of these concerns have been addressed previously [2, 14, 15, 28]. We believe *Fireflies* can scale easily to thousands of members and that this is sufficient for many applications.

Providing each member with membership is a form of agreement. Previous works on Byzantine fault-tolerant agreement establish invariants that are impossible to invalidate. Even the most practical of these protocols (*e.g.*, [5, 19]) require several rounds of all members broadcasting state to all other members, and can consequently not scale up to more than perhaps a few dozen members. In order to scale to thousands or more members, we had to come up with a radically different approach. *Fireflies* makes use of epidemic techniques to form a probabilistic agreement, which can only establish invariants that hold with a certain probability. Because invariants never hold for certain, defense against adversaries trying to break agreement can never rest. The main contribution of this paper is this novel approach towards tolerating intrusions. A formal specification and correctness proof has been completed as well and is the subject of a future publication.

We distinguish three *states* of members: *correct*, *stopped*, and *Byzantine*. Correct members execute the protocols described in this paper faithfully. Stopped members are not executing any protocol steps. Byzantine members are not bound to the protocols. For convenience, we also refer to members that are either correct or Byzantine as *live members*. Members can switch between states at any time, which is commonly referred to as *churn*. Informally, *Fireflies* provides its correct members with a membership view that includes all members that have been correct for sufficiently long and excludes all members that have been stopped for sufficiently long.

A group membership protocol that tolerates Byzantine behavior of its members is said to be *intrusion-tolerant*.

[1]This work is supported by the DARPA/IPTO SRS program, the AFRL/Cornell Information Assurance Institute, NSF grant 0430161, and the Research Council of Norway IKT 2010 program.

[2]Fireflies, the beetles, model not only the on/off behavior of members, but like Byzantine members they are also known for their aggressive mimicry in order to dupe and devour related species.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
EuroSys'06, April 18–21, 2006, Leuven, Belgium.
Copyright 2006 ACM 1 59593 322 0/06/0004 ...$5.00.

There are limitations to group membership, particularly if intrusions are allowed. Correct members may be unreachable and appear stopped to other members. Recently stopped members may not yet have been detected and appear correct. A Byzantine member can disguise itself as a flaky correct member. Nonetheless, intrusion-tolerant network overlay routing protocols may be built using *Fireflies* as a building block. *Fireflies* currently supports a DHT and a multicast protocol, both of which are intrusion-tolerant as well.

We start with a description of related work in Section 2. Section 3 presents an informal specification of *Fireflies*. The membership protocol is the subject of Section 4. Section 5 describes the epidemic protocol that supports the membership protocol. An evaluation of *Fireflies* appears in Section 6. We discuss current applications of *Fireflies* in Section 7. Section 8 concludes.

2. RELATED WORK

The first paper that describes concrete defenses against Byzantine behavior in peer-to-peer (P2P) network overlays is [4]. While this paper addresses the problem of impersonation attacks, many of the problems discussed have to do exclusively with overlay routing table maintenance and message forwarding. In our protocol, the members do not need to route messages, while in [4], the problem of membership is not considered.

Some peer-to-peer storage services like OceanStore [18] and FARSITE [1] use traditional Byzantine consensus protocols among small groups of machines. These groups do not protect the integrity of the peer-to-peer network as a whole, but protect Byzantine failures among replicas of individual files or meta-files. Also, as the system scales up, the likelihood that one of these groups fail as a whole grows, potentially endangering the entire system. Recent work [10] on FARSITE attempts to isolate such groups from one another, at the cost of weakening the system's semantics.

In [12], the authors propose a mechanism called *Link Attestation Groups* to increase the robustness of overlay networks. A link attestation is a certificate stating that a particular participant x trusts that it has a good path to another participant y. Using a link-state protocol, groups of dozens of participants are formed. The authors argue that by providing access to the graph of attestations, applications will be able to build more reliable protocols, but so far none of these have been evaluated.

The problem of impersonation attacks was first considered in [9]. The paper proposes secure identifier generation as a solution, and both our protocol and [4] have embraced this solution.

The problem of intrusion-tolerant membership in P2P protocols is considered in [29]. The *Eclipse attack* is an attack where malicious members isolate correct members by filling the neighbor table of a correct member with addresses of malicious members. The paper suggest thwarting this attack by enforcing bounds on the in- and out-degrees of P2P members.

Many group membership protocols are based on providing members with consensus on membership views. Note that in such systems views may still be stale. Versions that tolerate Byzantine members have been designed and implemented (*e.g.*, [26, 27]), but the overhead of consensus renders such systems unscalable beyond a few dozen members.

In [28], Rodrigues and Blake argue that in many environments multi-hop routing is not cost-effective, and full membership maintenance is both possible and desirable. One-hop peer-to-peer routing protocols that maintain full membership for fail-stop environments are presented in [14, 15]. CONGRESS [2] is a non-P2P solution based on a scalable server hierarchy. Neither tolerates Byzantine behavior.

Most closely related to our work is the SWIM protocol [7]. As in *Fireflies*, SWIM has a separate failure detection protocol and an epidemic dissemination protocol. Unlike *Fireflies*, SWIM's failure detection protocol does not adapt to varying message loss, and SWIM is not tolerant of Byzantine behavior.

The SCAMP protocol [13] is another epidemic-style membership algorithm that, like *Fireflies*, uses a small number of gossip partners per member in order to increase scalability. SCAMP is not intrusion-tolerant, and does not have a failure detection component. Members have to leave the group explicitly by gossiping a message. SCAMP may converge to a non-random graph. CYCLON [32] presents an improvement over SCAMP, maintaining randomness even with high node churn.

There has been a variety of work on intrusion-tolerant epidemic protocols, apparently starting with [21]. These protocols consider the problem of correct members not accepting any malicious updates without using unforgeable signatures, and use a form of voting instead. Perhaps the earliest epidemic membership protocol is reported in [31], while epidemic dissemination was pioneered in the Clearinghouse system [8].

Drum [3] is a DoS-resistant multicast protocol. It uses a combination of gossip techniques, resource bounds for certain operations, and random UDP ports in order to fight DoS attacks, especially those directed against a small subset of the correct members. These techniques are orthogonal to the ones used by *Fireflies*, and can be used to make *Fireflies* less susceptible to DoS attacks.

3. MEMBERSHIP SPECIFICATION

Each member m has a unique identifier $m.id$. A member cannot choose nor modify its identifier. Each member m has a state that is either *correct*, *stopped*, or *Byzantine*.

Each correct member has a *view* $m.view$, which is a subset of all members. Informally, $m_2 \in m_1.view$ means that m_1 believes that m_2 is, at least until recently, not stopped. As well, $m_2 \notin m_1.view$ means that m_1 believes that m_2 is, at least until recently, not live.[3]

A correct member also has a set of *neighbors* $m.neighbors$, which is a subset of its view. The number of neighbors is logarithmic in the size of the view, *i.e.*, $|m.neighbors| = O(\log |m.view|)$. The mesh consisting of correct members and the links to their correct neighbors is connected and logarithmic in diameter, and thus forms a usable routing substrate.

We also want, again informally, that all correct members send a limited amount of data in that Byzantine members cannot cause correct members to send large amounts of data rendering the protocol unscalable. We cannot prevent a Byzantine member from sending large amounts of data.

[3]We do not provide Virtual Synchrony properties such as consensus on views among correct members.

Fireflies does not exactly provide these properties, as it is a probabilistic protocol. For example, it is possible that long time correct members are sometimes evicted from the views of other correct members, and it is possible that long time stopped members are included in the views of correct members. Also, correct members may be partitioned. The *Fireflies* protocol makes such inconsistent states infrequent, with probabilistic guarantees.

4. MEMBERSHIP PROTOCOL

In this section we describe the membership protocol that correct members[4] follow. For now we will assume that members have at their disposal a gossip channel that reliably broadcasts a message to all members within a time Δ. Section 5 will present a protocol that provides such a guarantee with high probability.

The basic idea of the membership protocol is that members monitor one another and use the gossip channel to disseminate *accusations* (failure notices). When a member m_1 receives an accusation for a member m_2, m_1 waits a time period of length 2Δ before removing m_2 from its view. Should m_2 receive, through gossip, an accusation about itself, then m_2 has the opportunity to gossip a *rebuttal* before 2Δ expires. There is an overhead associated with gossip, so we have to prevent Byzantine members from submitting frequent accusations about correct members. This is a complicated issue because correct members may accidentally accuse other correct members due, for example, to transient link failures. Thus not every false accusation is from a Byzantine member.

4.1 Assumptions

While we allow the network to lose messages, we do assume that all correct members can run a ping protocol effectively and apply a gossip-style broadcast protocol that can deliver messages to all correct members in a timely fashion. Details on the ping protocol appear in Section 4.6, while details on the implementation of the gossip protocol are presented in Section 5. While we do not require clocks to be synchronized, we do assume that clock rates among correct members are identical, although rates only need to be "similar" in practice.

Byzantine members have few restrictions. They know the exact state of every other member, and have zero-latency connections between each other. However, they do not have sufficient computational power to break cryptographic building blocks, and in particular they cannot forge public key certificates, or public key signatures of correct or stopped members.

We assume that there is a bounded probability P_{byz} that a live member is Byzantine. Note that this is a stronger condition than a bound on the probability that *any* member is Byzantine. Such a weaker condition would not suffice, as in the case that most non-Byzantine members are stopped, the few remaining correct members could be overwhelmed by Byzantine members. Nonetheless, the assumption that among all live members only a fraction is Byzantine is reasonable, particularly since we do not limit the fraction of stopped members among all members.

We assume that each member m has a unique identifier $m.id$ (*e.g.*, tax identifiers, passport numbers, etc.), and that a shared Certificate Authority (CA) does a thorough background check on each potential member before handing the member m a private key $m.priv$ and a corresponding public key certificate $m.cert$ that binds the member's identifier and network address to its public key.[5] Correct and stopped members never reveal their private key.

[4]We shall omit the adjective "correct" where obvious.

note	most recently known note of the peer
accusations	accusations, at most one per ring
nPings	#pings sent since last "pong" response
avgLoss	smoothed average of #pings lost + 1

Table 1: Fields in an *info* structure, one for each peer. The last two entries are used only for successor peers.

We also assume that trivial Denial-of-Service attacks by flooding can be detected and suppressed (see [3] for how this might be accomplished).

4.2 Data Structures

The members are organized on $2t + 1$ rings, which are circular address spaces (the value t is discussed below). Each member m attains a position on each ring by evaluating a secure collision-resistant hash function $\mathcal{H}$:

$$m.pos[ring] = \mathcal{H}(m.id \mid\mid ring)$$

(where '$||$' is the concatenation operation). The ordering of members on each ring is different (with high probability) as a result. Member m ranks members on each ring clockwise, with itself being 0, its first successor being rank 1, and so on. The basic idea of the protocol is that each member m_1 monitors, on each ring, the lowest ranked successor m_2 that m_1 believes to be live.

The members use two data structures that are gossiped: *notes* and *accusations*. A member creates notes in order to notify and convince the other members that it is live. A note is a tuple (*cert, epoch, enabled*), signed using the private key of the member. Here *cert* is the public key certificate of the member, *epoch* a number used to order its notes, and *enabled* a bitmap with $2t + 1$ bits, controlling which of the $2t + 1$ predecessors are allowed to issue accusations against the member.

If a member m_1 suspects a successor m_2 on a particular ring of having stopped, then m_1 accuses the note of m_2 last known to m_1 by creating an accusation. An accusation is a tuple (*note, ring, accuser*), signed by m_1, where *note* is the note of m_2, *ring* is the ring on which m_2 is a successor of m_1, and *accuser* is the identifier of m_1. A requirement on the accusation is that *note.enabled*[*ring*] is set. Thus a member can use the *enabled* bitmap in the notes it generates to restrict which predecessors can accuse the member, an ability that we will use to defeat repeated false accusations by Byzantine members.

The value t governs the number of Byzantine predecessors a member can tolerate. We choose t so that the probability of a member having more than t out of $2t+1$ live predecessors being Byzantine is small (see Section 4.5). Members set exactly $t + 1$ bits in their notes' *enabled* bitmaps. If a member m has at most t Byzantine predecessors, then m can disable all Byzantine predecessors that make repeated false

[5]Note that the CA can provide access control as well.

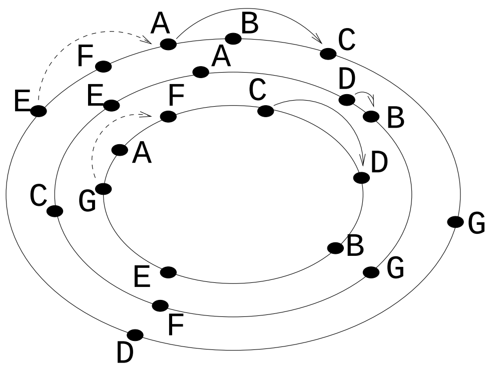


Figure 1: 3 rings with 7 members A through G. Valid accusations are shown with solid arrows, while invalid ones are shown in dashed arrows.

accusations while m is live. If, by mistake, m disables only correct predecessors and leaves t Byzantine predecessors enabled, then there is still at least one correct predecessor that can accuse m should m fail.

Each member m maintains a mapping $m.\mathit{info}$ of peer identifiers to information about these peers. ($m_1.\mathit{info}(m_2) = \perp$ means that m_1 does not have any information about m_2.) The fields in the *info* structure are listed in Table 1. The *Fireflies* protocol strives to ensure that the set of accusations is empty for a correct member and non-empty for a stopped member.

4.3 Valid Accusations

As we have not bounded the probability that a member is stopped, all predecessors of a member may be stopped with non-negligible probability. In order to allow such members to be accused in case they fail as well, a member must be able not only to accuse its immediate successor, but must also be able to make accusations skipping over stopped successors. Doing so may allow a Byzantine member to accuse any of its successors simply by claiming that it believes that the more immediate successors are all stopped.

In order to counter such attacks, we create rules that govern which accusations are considered *valid*. Informally, m_1 only allows the *highest ranked* live member to make *valid* accusations of m_2, and only on those rings that are enabled by m_2. Validity is defined recursively. Member m_1 considers an accusation for m_2 valid *iff*

- the accusation is correctly signed; *and*
- the note contained in the accusation corresponds to $m_1.\mathit{info}(m_2).\mathit{note}$; *and*
- the ring in the accusation is enabled in the note's *enabled* bitmap; *and*
- m_1 holds valid accusations for all members it ranks (on the given ring) between the accuser and m_2 itself, if any.

In Figure 1, we show a schematic depiction of how one of the members observes a group with 7 members, A through G. In this case, $t = 1$, and for simplicity we ignore ring deactivation. An accusation of B by D on the middle ring is a valid accusation (assuming the accusation refers to the note of B and is correctly signed by D) because there are no nodes in between D and B. This accusation is valid even though the accuser D is validly accused by C. The accusation by A of C on the outer ring is valid because there is a valid accusation against B, the node in between A and C. The accusation of A by E is invalid as there is no valid accusation of F.

A *Fireflies* member only maintains accusations that it considers valid, and associates a timer with each valid accusation that is set to 2Δ when the member first learns of the accusation. A valid accusation may depend on other valid accusations, and in case such a dependency changes, the timer needs to be reset. When the timer expires, the accusation leads to the removal of the accusee from the member's view.

4.4 Protocol Steps

Each correct member runs the same protocol. There are four events that trigger protocol transitions.

m_1 receives a note for a peer member m_2. If m_1 has a note for m_2 that is as recent as the one that arrived, then m_1 ignores it. Otherwise m_1 updates its note for m_2, removes any accusations that it has for m_2, cancels m_2's view removal timer if any, and includes m_2 in its view. In addition, the removal of accusations for m_2 may invalidate accusations that m_1 holds for other members. These accusations are removed as well.

m_1 suspects m_2. On each ring, m_1 monitors the lowest ranked successor m_2 for which m_1 does not hold valid accusations (unless m_2 has disabled the ring, in which case m_1 does not monitor anybody on that ring). Should m_1 suspect that m_2 has stopped, then it creates an accusation of m_2 that is subsequently gossiped to the other members.

m_1 receives an accusation for m_2. If m_1 does not consider the accusation valid, then m_1 ignores it. If $m_2 = m_1$, then m_1 replaces its note with a new one to act as a rebuttal, which is subsequently gossiped to the other members. If $m_2 \neq m_1$ and m_1 already has an accusation for m_2 on the same ring as the new accusation, then m_1 replaces its accusation only if the new one is from a higher ranked accuser.

At m_1, the timer of an accusation of m_2 expires. m_1 removes m_2 from its view.

4.5 Calculating t

We now turn to calculating a suitable value for t. If there are too few rings, correct members may not be able to fight Byzantine behavior. However, more rings result in higher overhead. Since we cannot give deterministic bounds on the number of Byzantine predecessors, we use a probabilistic approach. We want the minimum value of t so that the probability of a member having more than t out of $2t + 1$ live predecessors being Byzantine is smaller than ε:

$$\min_t : B(t; 2t + 1; 1 - P_{byz}) < \varepsilon$$

where $B(x; n; p)$ is a cumulative binomial distribution. We may want to make ε smaller for larger N so that the expected number of members for which this condition is violated does

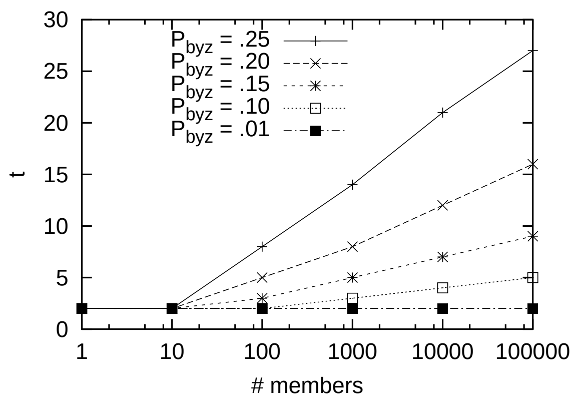


Figure 2: The magnitude of t for various N and P_{byz}.

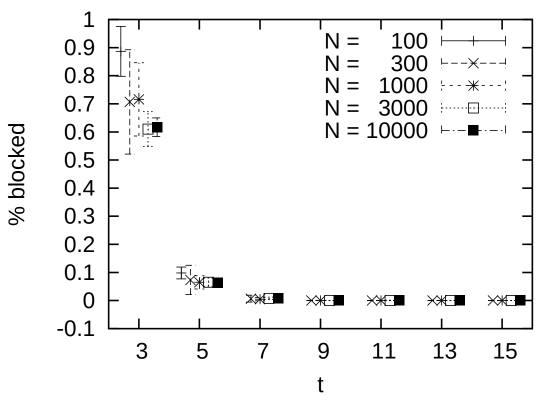


Figure 3: The likelihood of blocked accusations (99% confidence intervals) as a function of t for various N.

not grow linearly with N. For example, if we set $\varepsilon = 1/N$, then the expected number of such unfortunate members is 1, independent of N altogether. In Figure 2 we used the normal approximation to the binomial distribution to solve this equation for various N and P_{byz}, using $\varepsilon = 1/N$.

The protocol described above assumes a static t, implying that a maximum membership should be anticipated and enforced. It is, however, possible for a member to specify a number of rings that depends on the size of its view. Care should be taken to deal with Byzantine members specifying an *enabled* map in their note with a very large number of rings in order to try to consume all memory of correct members. The resulting extensions to the protocol are trivial and are not discussed in this paper.

Even if a stopped member m_1 has fewer than $t+1$ Byzantine live predecessors, it is possible that an enabled correct live predecessor m_2 cannot accuse m_1. Consider the stopped members between m_2 and m_1 on the corresponding ring and call them $s_1, \ldots, s_m$. If the ring is enabled for all s_i, then m_2 can accuse all s_i and thus m_1. Assume there is a member s_j that disabled the ring. Then m_2 cannot accuse m_1 until m_2 has received a valid accusation for s_j on a different ring. We say that the accusation of m_1 is blocked by s_j. Unfortunately, m_2 may never receive an accusation for s_j. Member s_j may have fewer than $t+1$ correct live predecessors, all of which being disabled. It may even be that the accusation of s_j is blocked by m_1, thus creating a loop preventing both m_1 and s_j of being accused.

Fortunately, it can be shown that the probability of blocked accusations is negligible. Formally, the probability of a stopped node not being accused is upper bounded by:

$$P \approx P_{\text{byz}} + (1 - P_{\text{byz}})P_{\text{stopped}}^{\ln n})^{t+1}$$

While the proof is outside the scope of this paper, a simulation bears this out as well. In Figure 3, we show the results of the following simulation. Initially, all members are correct and are present in all the views. At time $T = 0$, 75% of the members stop, 20% of the remaining members become Byzantine. In order to generate a worst-case scenario, the correct members disable as many correct predecessors as possible, and the Byzantine members do not emit any accusations. The graph shows that even for large N, a relatively small value for t makes blocked accusations highly unlikely.

4.6 Pinging

Members use pinging to detect failures. Essentially, a member m_1 monitors a member m_2 by sending "ping" messages to m_2 at regular intervals. Member m_2 returns a "pong" message for each ping that it receives. If m_1 does not receive pong messages from m_2 for more than some time period, m_1 considers m_2 stopped and issues an accusation.

A tricky detail is determining how long to wait before issuing an accusation. Using a static global timeout is not a good choice, as this will not scale well and can cause correct members to accuse other correct members more often than necessary. In particular, the timeout period has to adapt to the message loss characteristics between monitor and monitoree.

In Figure 4, we present a simple but effective protocol based on unreliable message passing. The members individually estimate the probability of message loss. We model pinging as a negative binomial experiment with parameters $r = 1$ and the probability of success p. Then the expected number of consecutively failed ping exchanges is $E(X) = (1-p)/p$. It follows that $p = 1/(E(X)+1)$. We estimate $E(X)+1$ by *avgLoss* using exponential smoothing. (The smoothing factor α is set to .999 in our current implementation.)

Having p, we can calculate the probability of making τ mistakes: $(1-p)^{\tau}$. We want this probability to be smaller than a configured constant $P_{mistake}$. It follows that $\tau > \log(P_{mistake})/\log(1-p)$.

If message loss is very low, τ would be set unrealistically low, and if $p = 1$, the expression would be undefined. We address both problems by having a minimum threshold τ_{min}. It follows that p should be set no higher than $1 - P_{mistake}^{1/\tau_{min}}$.

Byzantine members could potentially prevent detection of stopped members by forging pong messages. This is prevented by having each ping message contain a nonce that has to be signed by the monitoree and returned in the corresponding pong message. This strategy prevents both forging of pong messages and replay attacks, and this is why we chose pinging over a heartbeat protocol.

Byzantine members can, however, generate a modest amount of overhead on the system by not responding to ping messages from correct members, and rebutting the ensuing accusations. Such "nuisance attacks" are easily identifiable,

```
on time to ping $m$ on ring $r$:
    $p = \min(1/info(m).avgLoss, 1 - P_{mistake}^{1/\tau_{min}})$;    // est. probability of successful ping exchange
    $\tau = \log(P_{mistake})/\log(1-p)$;    // calculate threshold
    if $(info(m).nPing > \tau)$
        $info(m).accusation =$ new Accusation$(info(m).note, r, self.id)$;
    else
        send$(m,$ new Ping$(self.id))$;
        $info(m).nPing++$;

on receive Pong$(m)$:
    $info(m).avgLoss = (\alpha * info(m).avgLoss + (1-\alpha) \cdot (info(m).nPing))$;
    $info(m).nPing = 0$;
```

Figure 4: Pinging protocol. $P_{mistake}$ **(probability of making a mistake),** τ_{min} **(minimum threshold), and** α **(smoothing factor), are configuration constants.**

and such members can be manually removed by revoking their public key certificates.[6]

5. GOSSIP PROTOCOL

A gossip protocol is a simple group communication protocol whereby each member periodically picks a random member from its view and exchanges state information. Such protocols are known to be highly robust, as they are essentially flooding protocols. But unlike flooding protocols, they are efficient with probabilistic bounds on delivery latency [17]. In our particular situation, we have to concern ourselves with Byzantine members.

Say we have two members m_1 and m_2 exchanging notes and accusations. All notes and accusations are signed, and because we assume that Byzantine members cannot break the cryptographic building blocks, we do not have to worry about impersonation attacks [9]. We have also assumed that trivial Denial-of-Service attacks can be detected and suppressed.

Byzantine members can still attack the gossip protocol in the following two ways. In order to slow down dissemination, they can neglect to forward recent updates. This slow-down can be incorporated in the calculation of Δ. Byzantine members can also pretend that they have no information, causing correct members to transmit their entire state to them and thus causing unnecessary load on the correct members and on the network. In order to reduce the opportunities of Byzantine members to launch this attack, we will consider gossip protocols in which each member can only gossip with a small subset of the membership.

5.1 Partial Membership Gossip

Kermarrec et al. [17] shows that it is possible to build effective gossip protocols if each member only has a small set of uniformly chosen members it gossips with. Each member m selects k gossip *neighbors* from its view uniformly at random where k be large enough to create a connected graph of correct nodes. A classic result of Erdös and Rényi [11] shows that in a graph of n nodes, if the probability of two nodes being connected is $p_n = (\log n + c + o(1))/n$, then the probability of the graph being connected goes to $\exp(-\exp(-c))$.

The number of correct nodes in the view, n, is expected to be at least $(1 - P_{byz}) \cdot N$, where P_{byz} is a configured upper bound on the probability that a live node is Byzantine, and N is the total of the correct and the Byzantine nodes. Then the probability that one node is connected to another is $1 - (1 - 1/N)^k \approx k/N$. Thus $p_n \approx 2k/N$.[7]

In order for the correct nodes to be connected with probability φ, we obtain

$$k \geq \frac{N}{2n} \cdot \left(\log \frac{-n}{\log \varphi} + o(1) \right)$$

Next we determine the resulting Δ, the time to disseminate an update in this random graph. To better preserve resources, each member does not update all its k gossip neighbors in each round, but instead selects one neighbor for each round in a round-robin fashion. In order to simplify calculations, we will assume conservatively that it takes k rounds to update all gossip neighbors, and thus the dissemination runs a factor k slower than if all neighbors were updated in each round. If d_n is the diameter of the graph of correct nodes, then the expected amount of time to disseminate an update reliably among the correct nodes is therefore $\Delta = k * d_n$.

An asymptotic value for d_n can be determined. A recent result of Chung and Lu [5] shows that if $np_n \to \infty$ (which in our case it does), then the expected diameter of our graph is $(1 + o(1)) \frac{\log n}{\log np_n}$. Unfortunately, it does not provide the constants needed to tune the gossip protocol.

In order to find suitable constants, we ran simulation experiments with N ranging from 16 to 16,384 for varying P_{byz} and with k chosen as above (ignoring the $o(1)$ term), to determine if the resulting graphs of correct nodes are indeed connected and to obtain values for Δ. We ran each experiment 100 times. We encountered no disconnected graphs in any of our 3000 experiments. In Figure 5 we report the maximum Δ we observed for each N and P_{byz}.

5.2 Pseudo-Random Mesh

The analysis of the gossip protocol above tacitly ignores the possibility of a Byzantine member selecting more than k neighbors in order to increase the overall load on the correct

[6]Not discussed in this paper, a CRL can be reliable disseminated using the gossip protocol.

[7]We assume here that every correct node can connect to every other correct node. This assumption can be relaxed, but p_n has to be adjusted accordingly.

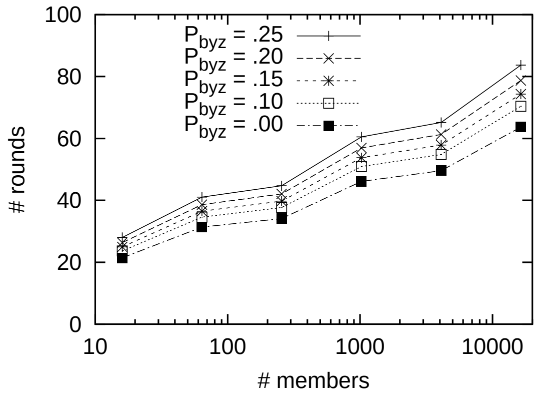


Figure 5: The maximum of 100 simulation experiments of the number of rounds required to disseminate an update to all correct members as a function of the total number of live members for various P_{byz}. In these experiments, $\varphi = .99999$.

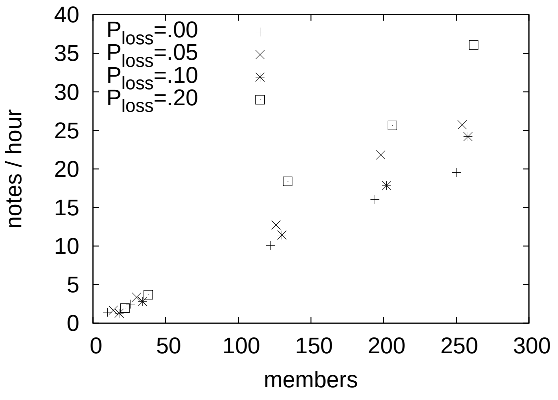


Figure 6: Number of notes sent per correct member per hour as a function of the number of members for various P_{loss}.

members. Also, Byzantine members could "gang up" on a small set of correct members, overwhelming them with gossip load [3]. In order to fight such attacks, we introduce a rule that determines who can gossip with whom. We use the same technique that we used in Section 4.2 to determine who monitors whom, except that we use k rings.

On each ring, a member initiates gossip only with the first successor in its view. For ring i, a member m sets up a secure mutually authenticated connection with the successor m_i using their private and public keys. Member m then sends $m.note$ and i to m_i, so that m_i can add m to its view if necessary and possible (existing accusations of $m.note$ may prevent this). Member m_i checks that $1 \leq i \leq k$ and that m_i is m's successor in m_i's view.

One complication is that even when m and m_i are both correct, they may have different views. In particular, m_i may know a "better" gossip neighbor n_i for m that is not in m's view. If such is the case, m_i sends n_i's note to m. Should m have plausible accusations for n_i, then it returns those to m_i and terminates the attempt to gossip. If no such accusations exist, then m was unaware of n_i. In that case m adds n_i to its view and tries to gossip with n_i instead.

If at any point in time m should determine a better gossip neighbor for ring i than m_i, then m terminates the existing connection. Note that newly joining and recovering members should gossip with at least $t + 1$ different members before they can be reasonably certain that they will be integrated into the "true" membership (as opposed to a fake membership created by Byzantine members [29]).

The neighbors thus chosen form a convenient low-diameter mesh that connects the correct members. *Fireflies* exposes the set of neighbors in its API, so that applications can gossip about information other than membership or use the mesh for multicast dissemination (as discussed in Section 7.2).

6. EVALUATION

With some moderate assumptions, such as having the graph of correct nodes be connected, we can formally prove that all correct nodes will be included in the view of correct members and that stopped nodes will almost certainly be removed from the views. In this section, we present the results of experiments that validate the performance of *Fireflies*.

Our prototype implementation is written in Python. The code can run both on a simulated network, or on a real network. In all experiments below, we used $t = 12$, resulting in 25 rings. We used $k = 8$ (corresponding to $\varphi = .999$), meaning that each member had about 16 neighbors for gossiping. Each member gossiped once every 3.75 seconds on average (resulting in one gossip per minute with every neighbor), although the *Fireflies* code randomizes the intervals at which a member gossips in order to prevent synchronized "waves" of gossip. The probabilistic upper bound on the time for gossip to spread, Δ, was chosen to be 5 minutes. Members pinged each of their monitorees every 30 seconds.

We will first describe some experiments performed on the simulated network, and then present experience gained from a deployment of the code on PlanetLab [24].

6.1 Simulation

In order to trigger frequent mistaken failure detections, we set $P_{mistake}$ to .001. Both the MTTF (Mean Time To Failure) and MTTR (Mean Time To Recovery) of the correct members was set to 6 hours. The intervals between stopping and going were exponentially distributed. The total number of members N ranged from 16 to 256. Each experiment ran for six simulation hours, and each experiment was run at least eight times. The graphs below show averages and 95% confidence intervals, except where the intervals were too small.

First we looked at the overhead in the absence of Byzantine members. Figure 6 shows the average number of notes sent (created or forwarded) per correct member per hour as a function of the number of members for various P_{loss}, the probability of message loss. In each case, we see a clear linear trend as a function of the number of members, as expected. Without loss, the expected number of notes is $N/12$, as on average there is one recovery every 12 hours. With loss, mistakes are made, leading to an increase in the number of notes generated. Due to adaptive pinging this is almost, but not completely, independent of P_{loss}. For example, for 5% loss the rate of notes sent is higher than for 10% loss. The

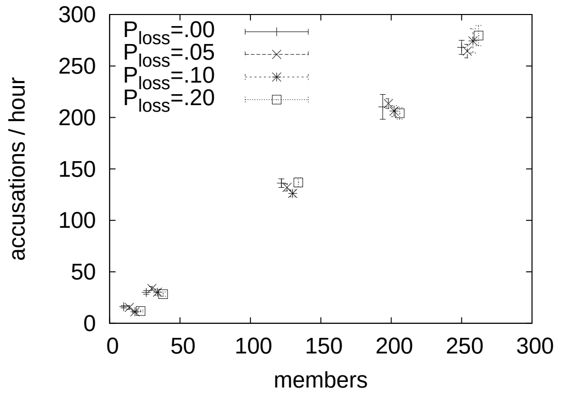


Figure 7: Number of accusations sent per correct member per hour as a function of the number of members for various P_{loss}.

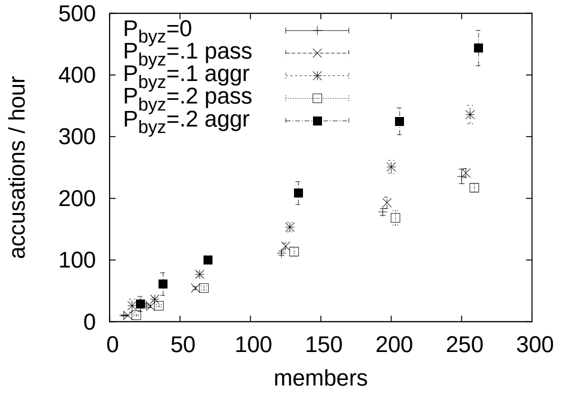


Figure 8: Number of accusations sent per correct member for various P_{byz} and styles of attack.

differences are due to rounding of τ, the pinging threshold above which a failure is assumed. The effect is that significantly fewer mistakes may be made than specified with $P_{mistake}$.

Figure 7 shows the average number of accusations sent per correct member per hour as a function of the number of members for various P_{loss}. Because accusations can be made on multiple rings, these rates are higher than for notes, but they do not depend much on the message loss probability. Partially this is due to the success of our adaptive pinging protocol, but it is also due to the dissemination of a mistaken accusation being squelched by the ensuing rebuttal.

Next we introduced Byzantine members. We varied P_{byz} from 0 to 0.2. We looked at two types of attacks. One is an "aggressive attack," where Byzantine members accused other members at any opportunity, and refrain from forwarding notes (rebuttals) from these members. The other is a "passive attack," where Byzantine members never accuse stopped members, and do not forward accusations of stopped members, in an effort to make stopped members appear correct. Neither style of attack was successful in any of our tests. Figure 8 shows the average number of accusations sent by correct members for various P_{byz} and styles of attack. The attacks had a moderate effect on traffic generated, but the amount stayed well within a factor of two of the case in which there were no Byzantine attacks.

6.2 Experience on PlanetLab

PlanetLab [24] is a world-wide collection of over 600 machines at over 275 sites connected to the Internet in 30 countries. PlanetLab can be used to test new scalable protocols and to deploy novel distributed services. We first deployed *Fireflies* on PlanetLab in early February 2005, and found the experience useful to find pragmatic problems and test solutions. However, the overheads we measured, some of which are presented below, are specific to PlanetLab only.

In this section we describe our experiments and indicate where further work on *Fireflies* is needed. Our prototype implementation uses TCP for gossip but UDP for pinging as our adaptive pinging protocol needs to determine when messages get lost.

While the majority of PlanetLab nodes tend to be fairly well-connected, some of the nodes are very heavily loaded, to the point of making some of these nodes effectively unreachable. Other nodes are only partially reachable, either due to configuration problems or due to heavy packet loss. For example, some nodes cannot send or receive UDP messages. This has two consequences for *Fireflies*. First, a node that cannot receive UDP packets will accuse its successors, even if they are correct. This is not a problem, as these successors will use their *enabled* bitmaps to disable the corresponding rings. Second, such a node will be accused by its predecessors. The accusations are effectively rebutted, and this accused member is not removed from the views as long as it is able to gossip new notes (using TCP). Unfortunately, the member cannot disable all rings (which would have its own problems), leading to a continuous background gossip of accusations and rebuttals. Besides a communication overhead on the network, these superfluous messages increase the load on machines.

6.3 Measurement Study

We now describe the results of one of our recent PlanetLab experiments, run February 24, 2006. The parameters were set the same as in the simulations above, except that $P_{mistake}$ was set to a more sensible .00001. For signatures we used SHA1 and RSA with 1024 bit keys. This resulted in a public key certificate of 163 bytes, a note of 49 bytes, and an accusation of 52 bytes.

Each time a member m_1 gossips with member m_2, they first exchange a collision-resistant hash of their sets of notes and accusations. If they are different, a full state reconciliation is done using the algorithm described in [22].

The experiment started with *Fireflies* agents running on 280 PlanetLab machines. During the experiment about 10 of those machines became unresponsive and fell out of the experiment. 10% of the *Fireflies* agents were configured to mount false accusations aggressively (chosen randomly), while another 10% where configured to mount a passive attack, not accusing and not forwarding accusations for failed members. At 7pm, we terminated 80 of the *Fireflies* agents, chosen randomly. At 7am, we restarted the agents.

Each agents writes a checkpoint to its log every 10 seconds. If an agent has not written a checkpoint to its log during a 2 minutes period of the experiment it is considered

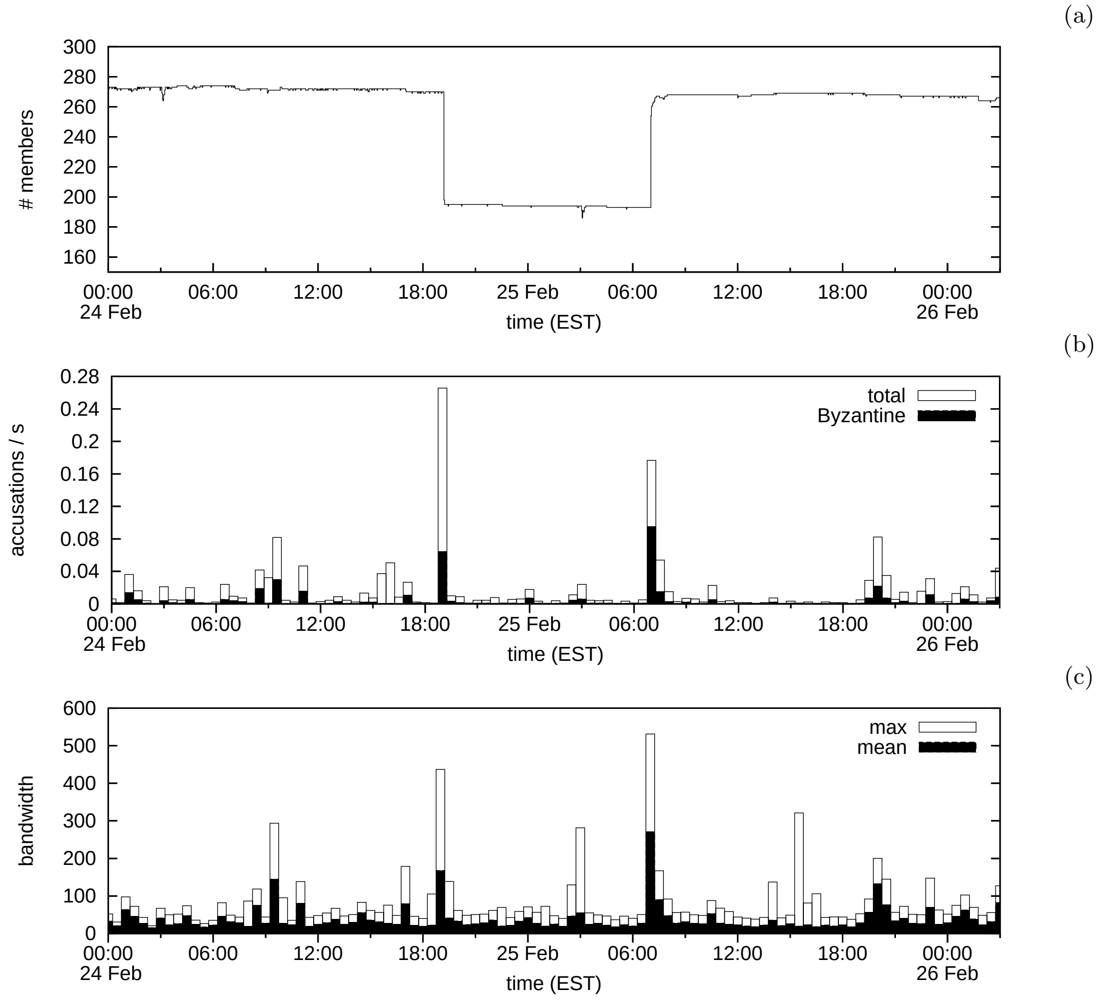


Figure 9: PlanetLab results: (a) size of the membership; (b) # accusations / second; (c) # bytes written per member per second.

failed in that period. The number of live agents in each time period is shown in Figure 9(a). We can observe that there is a fair bit of membership churn not under our control.

Figure 9(b) shows the aggregate rate of accusations created per second, divided into total and false accusations from Byzantine members. Two peaks can be clearly distinguished: when the agents are terminated, and when they are restarted. The first peak is obvious: the terminated agents are accused. The second peak is caused by several rejoining agents becoming unresponsive due to some heavily loaded PlanetLab machines's inability to accommodate the extra CPU and network overhead incurred when reintegrating the recovering agents into the membership. Accidental accusations from correct members gives aggressive Byzantine members opportunities to issue new false accusations, adding to the temporary flurry of communication.

Figure 9(c) shows the mean and maximum number of bytes sent per correct member per second. The bandwidth follows the rate of accusations, but is never above 500 bytes per member per second. The various peaks are caused by real failures. We have witnessed in our experiments various unexpected behavior on PlanetLab nodes. Sometimes the local file system on a node disappears or runs out of disk space, preventing logs to be written. Sometimes nodes are wrongly configured with an unroutable IP address. Sometimes nodes become unaccessible due to network outages, CPU overload, or, rarely, actual crashes. There have even been bugs in *Fireflies* agents causing them to crash or behave erratically. But the *Fireflies* infrastructure as a whole has survived all of these problems.

7. CURRENT APPLICATIONS

There are clear limitations to what *Fireflies* can offer. Byzantine members can disguise themselves as correct members by executing the protocol, or as stopped members by not executing at all, and so a correct member cannot determine which members are Byzantine unless they reveal themselves by sending messages that prove that they are not following the protocol. Also, views trail membership changes, and may be stale at any time. The question then is how *Fireflies* can be useful.

In this section, we provide examples of using *Fireflies* for building intrusion-tolerant network overlays. In particular, we show how *Fireflies* is used to support a Distributed Hash Table and a multicast protocol.

7.1 DHT

DHTs that are intrusion-tolerant are increasingly necessary. For example, the P6P overlay protocol is an IPv6 tunneling technology built over a DHT, and requires that links between correct members are fair [33]. An intrusion-tolerant DHT can be trivially implemented on *Fireflies*, simply by routing messages for an object identifier to the member in the view with the closest member identifier (assuming object and member identifiers are chosen from the same identifier space). Such an implementation is called a *One Hop DHT* (OHDHT),[8] as messages are not routed through intermediate members.

Other DHTs provide its members with only a partial view of the membership in order to increase scalability, and messages sent between members often take multiple hops. In a OHDHT, messages are less likely to get lost or intercepted along the way and encounter lower latency.

7.2 Multicast

A more interesting use of *Fireflies* is to build an intrusion-tolerant multicast protocol. For example, the Vigilante worm containment system [6] assumes a hypothetical intrusion-tolerant multicast primitive. Our protocol is heavily based on Chainsaw [23], which floods each message on the neighbor mesh. Flooding is done efficiently: when a member receives a large message, it notifies its neighbors only of the message identifier. Each member collects such notifications from its neighbors and requests the message from one. If no response is received within a short period of time, another neighbor is selected (see below). Measurements on Chainsaw have shown that this protocol is as efficient as the best multicast protocols based on DHTs [23], and the measurements of our version of the protocol support this as well.

Because the neighbor mesh connects all correct members, a message from a correct member is guaranteed to be delivered to all correct members. In order to prevent forging and prevent forwarding of illicit traffic, members sign messages and check signatures before accepting received messages. Correct members prefer uploading messages to neighbors from which they recently received a message. This strategy has two positive effects. First, it avoids using links to Byzantine members that are not forwarding messages. Second, it discourages freeloading. A paper that describes and evaluates our multicast protocol in the presence of Byzantine members and free-loaders is forthcoming.

As an alternative to using a pseudo-random mesh, *Fireflies*' complete membership information could be used to build Harary graphs [16], which can tolerate a fraction of Byzantine nodes and can thus form the basis of a secure broadcast protocol [20].

8. CONCLUSION

We presented *Fireflies*, a weakly-consistent, scalable protocol that supports network overlays and tolerates Byzantine members with high probability. *Fireflies* may be used to support an intrusion-tolerant Distributed Hash Table, or for building intrusion-tolerant overlay routing networks, or simply to organize computer resources in, say, a wide-area computational or storage grid.

Fireflies consists of three subprotocols. First, an adaptive pinging protocol makes the probability of a mistaken failure detection independent of message loss. Second, an intrusion-tolerant gossip protocol disseminates information between correct members within a probabilistic time bound. Third, the membership protocol uses accusations and rebuttals to implement the membership information that *Fireflies* provides, and which in turn induces a pseudo-random mesh that can be used for overlay routing.

Our evaluation shows that, at least for moderately sized memberships, *Fireflies* performs well. As the rate of failures and recoveries tends to grow linearly with the size of membership, the amount of information received per member has to grow at least linearly as well. We show that the amount sent (point-to-point) per correct member grows linear with the churn rate, almost independent of the behavior of Byzantine members (with a relative membership of up to 20%). The observed overheads for memberships up to about 280 members are low, and we believe that *Fireflies* will be able to scale up to another order of magnitude without difficulty.

Availability

The code (including the simulation code) is available on SourceForge (http://sourceforge.net/projects/fireflies).

Acknowledgements

The paper was significantly improved using suggestions from the anonymous Eurosys reviewers, from Dag Johansen, and from our shepherd, Christof Fetzer.

9. REFERENCES

[1] A. Adya, W.J. Bolosky, M. Castro, G. Cermak, R. Chaiken, J.R. Douceur, J. Howell, J.R. Lorch, M. Theimer, and R.P. Wattenhofer. FARSITE: Federated, Available, and Reliable Storage for an Incompletely Trusted Environment. In *Proc. of the 5th Symp. on Operating Systems Design and Implementation*, Boston, MA, December 2002. USENIX.

[2] T. Anker, G.V. Chockler, D. Dolev, and I. Keidar. Scalable group membership services for novel applications. In M. Mavronicolas, M. Merritt, and N. Shavit, editors, *Proc. of the Workshop on Networks in Distributed Computing*, pages 23–42. DIMACS, 1998.

[3] G. Badishi, I. Keidar, and A. Sasson. Exposing and eliminating vulnerabilities to Denial of Service attacks in secure gossip-based multicast. In *Proc. of the International Conference on Dependable Systems and Networks (DSN)*, pages 201–210, June – July 2004.

[4] M. Castro, P. Druschel, A. Ganesh, A. Rowstron, and D.S. Wallach. Secure routing for structured peer-to-peer overlay networks. In *Proc. of the 5th Usenix Symposium on Operation System Design and Implementation (OSDI)*, Boston, MA, December 2002.

[5] F. Chung and L. Lu. The diameter of random sparse graphs. *Advances in Applied Math*, 26(4):257–279, May 2001.

[6] M. Costa, J. Crowcroft, M. Castro, A. Rowstron, L. Zhou, L. Zhang, and P. Barham. Vigilante:

[8]Not to be confused with an $O(1)$ hop DHT, although OHDHTs are a member of that class.

End-to-end containment of Internet worms. In *Proc. of the 20th ACM Symp. on Operating Systems Principles*, Brighton, UK, October 2005.

[7] A. Das, I. Gupta, and A. Motivala. SWIM: Scalable Weakly-consistent Infection-style process group Membership. In *Proc. of the Int. Conf. on Dependable Systems and Networks DSN O2*, pages 303–312, Washington, DC, June 2002.

[8] A. Demers, D. Greene, C. Hauser, W. Irish, J. Larson, S. Shenker, H. Sturgis, D. Swinehart, and D. Terry. Epidemic algorithms for replicated database maintenance. In *Proc. of the 6th ACM Symp. on Principles of Distributed Computing*, pages 1–12, Vancouver, BC, August 1987.

[9] J. Douceur. The Sybil attack. In *Proc. of the 1st Int. Workshop on Peer-to-Peer Systems*, Cambridge, MA, March 2002.

[10] J.R. Douceur and J. Howell. Byzantine fault isolation in the Farsite distributed file system. In *Proc. of the 5th Int. Workshop on Peer-To-Peer Systems*, Santa Barbara, CA, February 2006.

[11] P. Erdös and A. Rényi. On the evolution of random graphs. *Magyar Tud. Akad. Mat. Kutato Int. Közl*, 5(17):17–61, 1960.

[12] M.J. Freedman, I. Stoica, D. Mazières, and S. Shenker. Group therapy for systems: Using link attestations to manage failures. In *Proc. of the 5th Int. Workshop on Peer-To-Peer Systems*, Santa Barbara, CA, February 2006.

[13] A.J. Ganesh, A.-M. Kermarrec, and L. Massoulié. SCAMP: Peer-to-peer lightweight membership service for large-scale group communication. In *Proc. of the 3rd International Workshop on Networked Group Communication*, London, UK, November 2001.

[14] A. Gupta, B. Liskov, and R. Rodrigues. One hop lookups for peer-to-peer overlays. In *Proc. of the 9th Workshop on Hot Topics in Operating Systems (HotOS-IX)*, pages 7–12, Lihue, HI, May 2003.

[15] A. Gupta, B. Liskov, and R. Rodrigues. Efficient routing for peer-to-peer overlays. In *Proc. of the 1st Symp. on Networked Systems Design and Implementation*, San Francisco, CA, March 2004.

[16] F. Harary. The maximum connectivity of a graph. In *Proc. of the National Academy of Sciences*, volume 48, pages 1142–1146, 1962.

[17] A.-M. Kermarrec, L. Massoulié, and A.J. Ganesh. Probabilistic reliable dissemination in large-scale systems. *IEEE Transactions on Parallel and Distributed Systems*, 14(3), March 2003.

[18] J. Kubiatowicz, D. Bindel, Y. Chen, P. Eaton, D. Geels, R. Gummadi, S. Rhea, H. Weatherspoon, W. Weimer, C. Wells, and B. Zhao. Oceanstore: An architecture for global-scale persistent storage. In *Proceedings of the 9th International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM, November 2000.

[19] C.S. Lewis and L. Saia. Scalable byzantine agreement. Technical report, CS Dept., University of New Mexico, 2004.

[20] M.-J. Lin, K. Marzullo, and S. Masini. Gossip versus deterministically constrained flooding on small networks. In *Proc. of the 14th Int. Conf. on Distributed Computing*, volume 1914 of *Lecture Notes in Computer Science*, pages 253–267. Springer, 2000.

[21] D. Malkhi, Y. Mansour, and M.K. Reiter. On diffusing updates in a Byzantine environment. In *Symposium on Reliable Distributed Systems*, pages 134–143, Lausanne, Switzerland, October 1999.

[22] Y. Minsky, A. Trachtenberg, and R. Zippel. Set reconciliation with nearly optimal communication complexity. *IEEE Transactions on Information Theory*, 49(9):2212–2218, September 2003.

[23] V. Pai, K. Kumar, K. Tamilmani, V. Sambamurthy, and A.E. Mohr. Chainsaw: Eliminating trees from overlay multicast. In *Proc. of the 4th Int. Workshop on Peer-To-Peer Systems*, Ithaca, NY, February 2005.

[24] L. Peterson, S. Shenker, and J. Turner. Overcoming the Internet impasse through virtualization. In *Third Workshop on Hot Topics in Networking (HotNets-III)*, San Diego, CA, November 2004.

[25] P. Pietzuch, J. Shneidman, J. Ledlie, M. Welsh, M. Seltzer, and M. Roussopoulos. Evaluating DHT-based service placement for stream-based overlays. In *Proc. of the 4th Int. Workshop on Peer-To-Peer Systems*, Ithaca, NY, February 2005.

[26] K. Potter Kihlstrom, L.E. Moser, and P.M. Melliar-Smith. The SecureRing protocols for securing group communication. In *Proc. of the the 31st Hawaii Int. Conf. on System Sciences*, volume 3, pages 317–326, Kona, HI, January 1998.

[27] M.K. Reiter. A secure group membership protocol. *IEEE Transactions on Software Engineering*, 22(1):31–42, January 1996.

[28] R. Rodrigues and C. Blake. When multi-hop peer-to-peer routing matters. In *Proc. of the 3rd Int. Workshop on Peer-to-Peer Systems*, La Jolla, CA, February 2004.

[29] A. Singh, M. Castro, P. Druschel, and A. Rowstron. Defending against Eclipse attacks on overlay networks. In *Proc. of the 11th European SIGOPS Workshop*, Leuven, Belgium, September 2004. ACM.

[30] E. Sit and R.T. Morris. Security considerations for peer-to-peer distributed hash tables. In *Proc. of the 1st Int. Workshop on Peer-To-Peer Systems*, Cambridge, MA, March 2002.

[31] R. van Renesse, Y. Minsky, and M. Hayden. A gossip-style failure detection service. In *Proc. of Middleware'98*, pages 55–70, The Lake District, UK, September 1998. IFIP.

[32] S. Voulgaris, D. Gavidia, and M. van Steen. CYCLON: Inexpensive membership management for unstructured P2P overlays. *Journal of Network and Systems Management*, 13(2), June 2005. Special issue on Self-Managing Systems and Networks.

[33] L. Zhou and R. van Renesse. P6P: A peer-to-peer approach to Internet infrastructure. In *Proc. of the 3rd Int. Workshop on Peer-To-Peer Systems*, San Diego, CA, February 2004.

SecureStream: An Intrusion-Tolerant Protocol for Live-Streaming Dissemination

Maya Haridasan

Department of Computer Science, Cornell University

Robbert van Renesse

Department of Computer Science, Cornell University

Abstract

Peer-to-peer (P2P) dissemination systems are vulnerable to attacks that may impede nodes from receiving data in which they are interested. The same properties that lead P2P systems to be scalable and efficient also lead to security problems and lack of guarantees. Within this context, live-streaming protocols deserve special attention since their time sensitive nature makes them more susceptible to the packet loss rates induced by malicious behavior. While protocols based on dissemination trees often present obvious points of attack, more recent protocols based on pulling packets from a number of different neighbors present a better chance of standing attacks. We explore this in SecureStream, a P2P live-streaming system built to tolerate malicious behavior at the end level. SecureStream is built upon *Fireflies*, an intrusion-tolerant membership protocol, and employs a pull-based approach for streaming data. We present the main components of SecureStream and present simulation and experimental results on the Emulab testbed that demonstrate the good resilience properties of pull-based streaming in the face of attacks. This and other techniques allow our system to be tolerant to a variety of intrusions, gracefully degrading even in the presence of a large percentage of malicious peers.

Key words: Content dissemination, multicast, live-streaming, intrusion-tolerance

1 Introduction

Access to multimedia contents over the network now accounts for a large fraction of Internet traffic. This has been possible in great part because of peer-to-peer (P2P) content distribution tools, which allow the distribution of popular data to a large number of interested users. One popular style of content distribution maps the problem to file sharing, where data is fully available prior to the dissemination. The main goal in file sharing is that all nodes receive the entire data within as little time as possible.

In this work, we are focused on a second scenario, P2P live-streaming, where data should be disseminated as it is generated. This style of distribution is useful to broadcast live events in close to real time and also to broadcast television over the web. In China, for example, live-streaming has become very popular, where participating peers' upload bandwidth is used to simultaneously propagate several channels to thousands of users [1].

Streaming to a large number of clients would be prohibitively expensive if the service provider should have enough bandwidth to satisfy all the clients. Several P2P multicast protocols which rely on users' upload resources have been proposed and widely studied as an appealing alternative to IP multicast, and previous work has shown that it can indeed be as efficient as IP multicast [2–5,1,6,7]. By having peers contribute to the streaming, anyone may start their own streaming session to any number of clients. Significant progress has been made, but little attention has been dedicated to the issue of security in such systems.

In this paper we target live-streaming, where malicious behavior can prevent nodes from receiving correct packets in time, and can therefore be severely disruptive. To illustrate the problem, we looked into the effects of one particular type of attack when using a single dissemination tree with varying branching factors and when using the more elegant SplitStream approach [4]. SplitStream is a robust and fair P2P system in which data is broken into several slices and each slice is propagated through a different dissemination tree.

As a measure of resilience, we compute the *continuity index* of a streaming session, which is the ratio of packets received by a peer within acceptable time. Through simulation we computed the minimum continuity index across participants for sessions with a thousand homogeneous nodes and varying ratios of malicious peers not forwarding packets.

In Figure 1, we present simulation results of the average and minimum continuity index across nodes for sessions with a thousand homogeneous nodes and various ratios of malicious peers not forwarding packets. In these experiments, attackers download data from their parents but do not forward it to their children. In the case of single trees, most certainly malicious behavior will prevent individual nodes from receiving any packet even with as low as 5% malicious members. SplitStream presents better resilience, but the damage incurred to individual nodes is still very visible.

We built SecureStream, which employs several techniques that reduce the

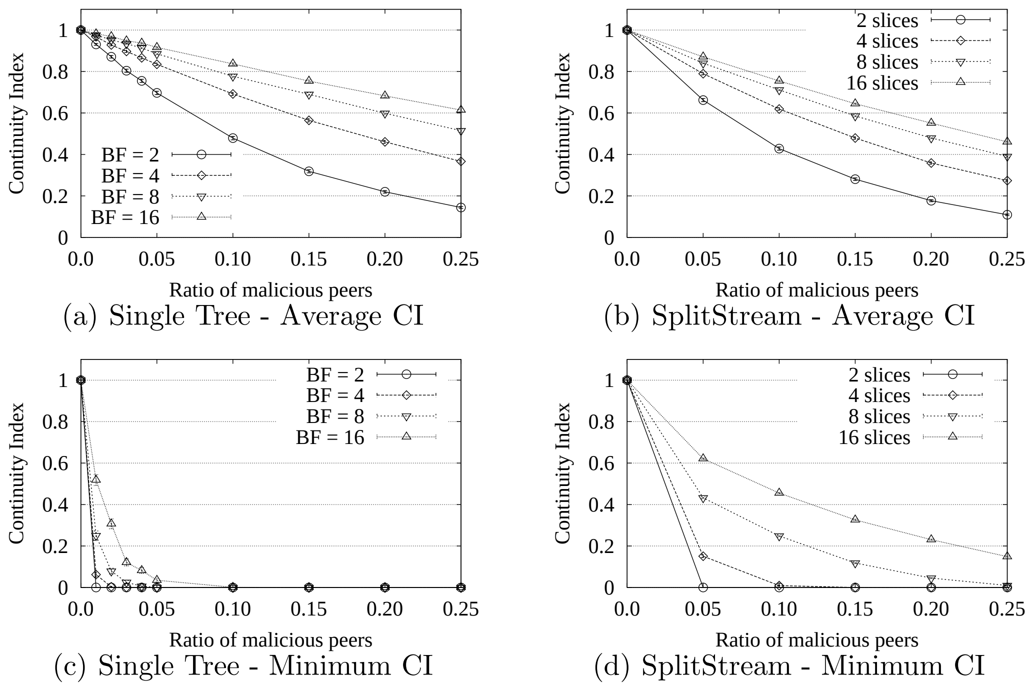


Fig. 1. Expected minimum and average continuity indices across all correct members under omission attacks.

opportunity for an attacker to compromise the quality of a streaming session, without incurring a high computational or network overhead. To repel forgery attacks, we employ an efficient packet authentication technique based on computing and distributing verification digests. To prevent attacks on the overlay structure (the membership protocol on top of which multicast systems operate), SecureStream is built upon *Fireflies*, a scalable one-hop Byzantine membership protocol [8]. *Fireflies* is a probabilistic protocol, in which members are presented with a reasonably current view of which members are live or not.

To achieve tolerance to denial-of-service attacks, SecureStream uses a pull-based packet dissemination approach, similar to the one used by the CoolStreaming [1] and Chainsaw protocols [6]. This approach is attractive because it offers participants a choice among multiple candidate packet sources. Because participants are not dependent on any particular peer and can immediately react to failures or attacks, attacks are less damaging.

Finally, we explore the use of auditing techniques that applied to SecureStream can help further alleviate the effects of malicious behavior, while incurring limited additional costs. We propose employing a variable threshold for node contribution, punishing peers who do not upload at least as much data as defined by the threshold.

This paper makes a few important contributions. We present a highly scalable live-streaming P2P protocol that can tolerate end system attacks. We leverage previous work and present a comparison of different authentication protocols for signing and verifying packets efficiently in the context of application level multicast. We also evaluate the effects of pull based protocols in the presence of internal malicious peers. Finally, we study the potential of auditing as a mechanism for encouraging node contribution.

The rest of the paper is organized as follows. In Section 2, the system model and assumptions are presented, including a list of possible attacks to P2P streaming systems. A description of the main techniques we employed in building SecureStream is presented in Section 3. In Section 4 results of the experimental evaluation of the resilience to malicious behavior are presented and analyzed. Section 5 presents related work, and Section 6 concludes.

2 System Model

Our model of the system assumes the existence of one source, assumed non-compromised, disseminating data at a fixed rate to a set of receivers with limited buffering capacity. All nodes have similar download and upload capacities, slightly larger than the download rate. The desired behavior is that the streamed data be received within a fixed latency relative to the source's original transmission.

The term security in the context of content dissemination protocols requires further definition. According to [9] multicast systems may have different requirements: secrecy means that only multicast group members (and all of them) should be able to decipher transmitted data; authenticity means that each group member can recognize whether a message was sent by a group member and make sure that the data was not modified in any way; anonymity implies that identity of group members should be kept secret from outsiders or from other group members; non-repudiation states that receivers of data should be able to prove to third parties that the data has been transmitted; access control means that it should be possible to control the group membership; and finally, service availability means that the system should be always up.

Not all applications require secrecy and anonymity of data, hence we are not concerned with these properties. On the other hand, we believe authenticity, non-repudiation and access-control are essential. We assume that the original data is non-compromised, and therefore implicitly achieve data integrity through authenticity. Our primary focus is on guaranteed availability, namely mechanisms that prevent nodes from being isolated or severely harmed during a streaming session. We expect the system to repel external attacks and tolerate a limited fraction of internal Byzantine nodes, and to degrade gracefully as the fraction of Byzantine nodes increases.

SecureStream is an application-level streaming system, and only attacks to the end system hosts are addressed in this work. Attacks on the underlying network infrastructure and low-level *denial-of-service* attacks are thus beyond the scope of this work.

2.1 Byzantine Behavior

We model all forms of deviation from the original protocol as byzantine behavior. These deviations may be due to node failures, node selfishness or purely malicious intents. Attacks may originate outside the system or be internal, and attackers may compromise nodes and then work in cooperation with these faulty internal peers. One important observation is that we opted for modeling selfishness as a byzantine behavior, and to assume that most nodes typically follow the given protocol.

To the best of our knowledge, there has been no evaluation on the percentage of selfish behavior in live-streaming systems, unlike with file sharing systems, and the properties of these systems are significantly different. Since nodes are only required to upload while the streaming session is occurring, it is our belief that few nodes would opt for deviating from the proposed protocol.

The simplest form of internal attacks are those in which a single node is compromised. The extent of harm that results depends on many factors, such as the multicast protocol being used and the location of the malicious node in the overlay. These effects can be localized and minimized if the protocol in use has no single points of failure. On the other hand, vulnerable systems like those based on a single dissemination tree can be crippled if a node high in the tree is compromised.

Collusion attacks pose much more complex problems; in these, an attacker compromises a set of nodes and exploits them to perform a coordinated attack to the system, and may orchestrate the attack to confound whatever defensive mechanisms are built into the dissemination infrastructure.

For the work presented here, we make several assumptions about compromised members. They do not have sufficient computational power to break cryptographic building blocks, and cannot forge public key certificates or signatures of correct or stopped members. A classification of the types of attacks that we address in our system is presented below.

Membership attacks: The system may be attacked by compromising the underlying overlay or membership protocol on which it runs. For example, systems that run on top of ring-based overlays are vulnerable to eclipse attacks [10], in which an attacker controls a large fraction of the neighbors of correct nodes, preventing correct overlay operation. Malicious nodes may also mimic flaky but correct members, or accuse other correct members of being down.

Forgery: In this category we include all attacks that involve fabrication and tampering of data being streamed in the system. Given time, these attacks can be easily avoided by use of a public key infrastructure. However, in the context of streaming the cost of signatures can become prohibitively high, forcing us to consider other kinds of data authentication protocols.

Denial-of-service (DoS) Attacks: Attacks in which malicious nodes overload peers with requests for packets or large amounts of duplicate packets, or other attacks that might compromise their ability to contribute to the streaming session.

Omission Attacks: Given our emphasis on low-latency data delivery, send-omission is an especially serious type of attack. By not forwarding all or part of the packets, a malicious node may disrupt overall system's availability. The main problem with this kind of attack is that a node's guilt cannot be easily proved.

3 Steps to Intrusion-Tolerant live-streaming

SecureStream employs a set of techniques to achieve resilience to the attacks previously mentioned. We use an intrusion-tolerant membership protocol to tolerate attacks to the membership layer. We also employ an efficient technique for avoiding forgery of packets by malicious peers. By employing a pull-based streaming protocol and imposing a structure to define what peers are allowed to communicate with one-another, we can avoid high-level DoS attacks and tolerate omission attacks. We also explore the potential of auditing as a tool for detecting malicious behavior. In this section, we describe these main components in further detail.

3.1 Presenting nodes with a correct view of live members

Peers in SecureStream use the membership knowledge provided by the *Fireflies* protocol to track the status of other peers. *Fireflies* is composed of three subprotocols: a pinging protocol is used to detect failures of nodes with an accuracy independent of message loss; an intrusion-tolerant gossip protocol is used for dissemination of information between correct members with probabilistic time bound Δ; and a membership protocol uses accusations and rebut-

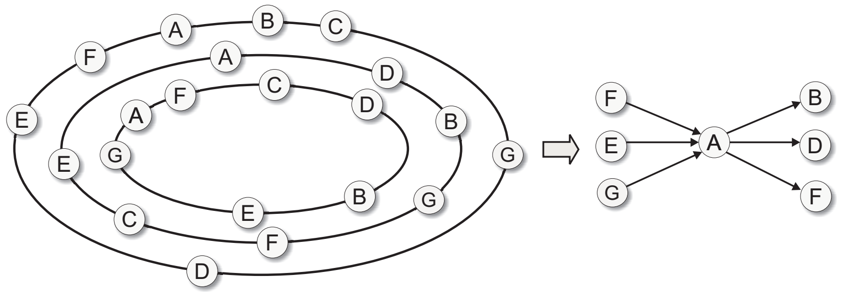


Fig. 2. In Fireflies multiple rings are used to define which peers monitor each other: A monitors B, D and F, and is monitored by E, F and G.

tals to implement the membership information that *Fireflies* provides. These components are briefly described below.

Members monitor each other for failure using an adaptive pinging protocol. Members do not use a static global timeout when waiting for the replies of ping messages, but rather estimate the probability of message loss and try to adapt to the message loss characteristics between monitor and monitoree.

Members are organized into rings, and their position on each ring depends on their identifier. These rings determine which nodes monitor, and are allowed to accuse, which other nodes (Figure 2). On each ring, each member m_i monitors the lowest ranked successor m_j that it believes to be live, and if it detects a failed node, it issues an *accusation* for that node.

When an accusation for a member m_i is received by a member m_j, m_j waits a time period of length 2Δ, and then removes m_i from its view if the accusation is valid. This time period is established so that an accused member may issue a new *note* (a *rebuttal*) to an *accusation* against itself. In order to avoid malicious nodes from abusively accusing its correct neighbors in the rings, nodes may invalidate up to t rings, implying that accusations issued by its neighbors on those rings will not be accepted as valid by any correct member. All notes and accusations are signed, and a certification authority is responsible for issuing private/public key pairs and public key certificates.

The dissemination of information such as accusations and rebuttals is performed using a robust gossip protocol. Each member periodically picks a random member from its view to exchange state information. The multiple ring structure induces a gossip mesh resilient to malicious attacks.

3.2 Ensuring Integrity of Data

One second important aspect which needs to be satisfied is that the data being distributed is correct. Several authentication protocols have been proposed for

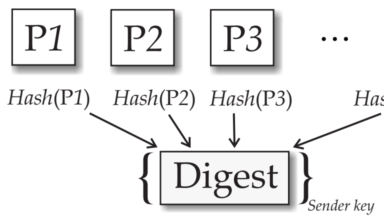


Fig. 3. In the linear digests' approach, packets' hashes are computed and combined into a single digest packet, which is then signed by the sender.

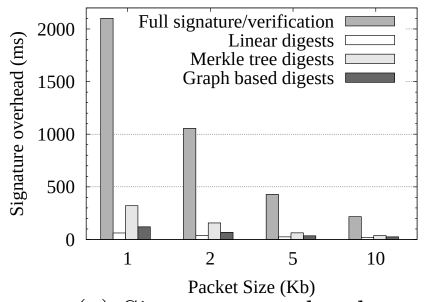
the general multicast paradigm, originally intended for IP Multicast. The standard point-to-point mechanism of appending a message authentication code (MAC) computed using a shared key does not meet the security requirements of a multicast session. If receivers and sender share the same key, any receiver would be able to forge messages. On the other hand, signing every packet using a traditional asymmetric cryptographic protocol induces high overhead, and is therefore not feasible.

Signing a packet consists of computing the hash of the contents of the packet using a secure hash function, and then signing the hashed value using the sender's private key. Variations in the packet size do not significantly contribute to the costs since computing the hash of packets is a cheap operation compared to the signature/verification operation. The choice of key size to be used is directly related to how crucial it is that the key be secret for a long time, and it is often recommended that keys of size 2048 or larger be used.

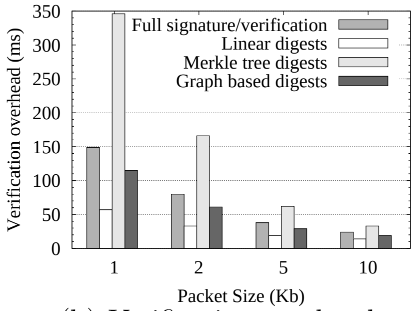
To avoid signing and verifying every packet, we group the hashes of n packets into a special message, and have it signed by the source (we call this approach *linear digests*)(Figure 3). The signed message needs to be sent to the receivers prior to the dissemination of data that it corresponds to. This implies in buffering of content on the source prior to the dissemination of data. The advantage is that this approach incurs the minimal network overhead of one hash per packet, while amortizing the cost of a single signature/verification operation over n packets.

Other approaches have been proposed to address the high costs of authenticating packets in a flow [11–15]. Wong and Lam [11] propose that the source compute the hashes of a limited number n of consecutive packets in the stream, and use them as leaves in a Merkle Tree where each internal node consists of the hash of its children. Each packet is verifiable upon receipt, since it is appended with the signed root node and the hashes of all needed interior nodes in the path from the root to itself in the Merkle Tree.

In graph-based authentication [13,16,17,14], the source only signs one packet, and the following packets in the stream are linked to it through hash chains that allow them to be verifiable. To tolerate packet loss, a graph is used instead of a single chain. Packets are represented by vertices in the graph, and a



(a) Signature overhead



(b) Verification overhead

Fig. 4. Computational overheads per second when transmitting 300 Kb/s using varying packet sizes.

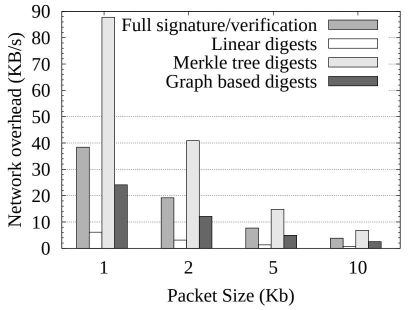


Fig. 5. Overheads per second when transmitting 300 Kb/s using varying packet sizes.

directed edge between nodes that represent packets P_i and P_j indicates that packet P_j contains the hash of packet P_i. A packet corresponding to a node can be authenticated if there is a path of already verified packets between the node and the source node of the graph.

The computational costs at the source and receivers are presented in Figures 4(a) and 4(b) and the network overheads for different authentication approaches when streaming 300 Kb/s are presented in Figure 5. We compared 4 techniques: signing and verifying every packet, linear digests, Merkle tree digests and a simple scheme of graph-based authentication. The code used for the evaluation was written in Python and executed on a Linux-based Pentium III 850 Mhz with 256 Mb RAM. Linear digests yield the lowest computational costs. Although the Merkle Tree approach is appealing due to its immediate verifiability, its network overhead is the highest, since one signature and a few hashes need to be appended to each packet in the flow.

When the packet sizes are large, which reduces the rate of packets per second, the computational costs of the three latter techniques are not significantly different. We therefore used linear digests since it minimizes network overhead and is the simplest technique. Our experience indicated that when using pull based streaming, keeping the rate of packets per second larger or equal to 30 yields good results, and reducing it further affects the quality of the streaming.

3.3 Allowing nodes to recover from malicious neighbors

We employ a pull-based approach to disseminate packets, following ideas used in the Chainsaw protocol [6]. The same rings used in *Fireflies* are used to determine a fixed set of neighbors with which each peer can exchange packets. This imposed mesh structure and the use of authenticated channels between neighbors allows the system to avoid high-level DoS attacks.

Initially, the source sends notifications to its neighbors as soon as it has available packets to disseminate. These notifications are small messages used only to inform neighbors of availability of packets. Each neighbor requests missing packets according to some pre-specified policy, to avoid overloading the source. As peers receive packets, they propagate notifications to their neighbors, and so packets get disseminated along the mesh. This pull-based approach to acquisition of packets yields a highly resilient multicast, since failure or misbehavior of one neighbor does not impede a peer from fetching packets from other neighbors. The predetermined set of neighbors for each peer also makes it hard for malicious peers to attack individual peers, since attackers lack a deterministic means of acquiring control of all of its neighbors.

Each member stores packets and forwards them to other peers while the packet is within its *availability window*. It also maintains an *interest window*, smaller than the availability window, which represents the set of packets in which the peer is currently interested. Different policies can be employed by peers about what packets to pick from each of its neighbors, and the choice of the appropriate policy is crucial to achieving best overall performance. Random selection of neighbors is usually a good candidate, leading to fair load balancing.

There is a predefined limit l on the number of outstanding requests to any neighbor. This policy not only improves the flow of packets in the absence of malicious behavior, but also makes it harder for malicious peers to overrequest packets from their neighbors. Peers maintain a queue of non-satisfied requests for packets, and if more than l requests by the same neighbor are present in the queue at any time, only the l most recent ones are maintained.

The protocol is simple and yet highly resilient to failures and attacks. The overhead incurred by notifications is not significant if large packets are used, and the protocol avoids receipt of duplicate packets. Since it is completely decentralized, the protocol does not present any single points of failure, another important consideration when building an intrusion-tolerant streaming protocol.

To ensure non-repudiation, peers may only forward packets once they have verified its authenticity. If peers are allowed to forward packets optimistically before ensuring that the packet has not been tampered with, it becomes infeasible to later identify the peer responsible for the tampering. This can be explored by malicious nodes, who may overload the system with incorrect packets without being accountable for them. In the linear digest approach, the packet that contains the digest is critical to the verifiability of packets, and therefore should be received by all nodes. Furthermore, it should ideally be received prior to other packets for which it contains hashes, so that they can be immediately verified.

In our streaming protocol, simply treating the digest packet as a regular packet would not yield the desired results. We proposed the following optimization to ensure immediate verifiability: each peer, when requesting a packet, uses a special bit in the request messages that indicates whether the digest packet for the current session has been received or not. Since digest packets are small, they can be appended to the packet sent in reply to the request. This would ensure that all packets are immediately verifiable, incurring a small overhead caused by duplicate digests.

3.4 Punishing Malicious Nodes

Despite the resilience of pull-based streaming to malicious behavior, we can provide further guarantees to correct peers by auditing their behavior. In this Section, we explore a simple auditing approach: to ensure that all nodes in the system contribute more than a particular specified threshold. Violations to this invariant may lead system nodes who contribute less than a particular threshold to suffer some type of punishment, such as being expelled from the system. Independent of the punishment, implementing an auditing component requires caution, and in this subsection we present some of the techniques we employ to achieve this goal.

As part of the auditing approach, each peer should group packets it receives from each neighbor every δ seconds. At the end of every interval, each peer generates and sends one signed receipt for all packets received from each of its neighbors during that interval, and collects receipts received from them. Peers are encouraged to forward receipts to their suppliers to guarantee that future requests for packets continue to be satisfied.

Auditing may be performed by dedicated external auditors, whose role is solely to identify misbehaving nodes. We propose a decentralized approach, which combines local auditors, executing at the participating peers, and global auditors, who react to violations reported by the local peers. A local auditor has two main roles. First, it acts as a representative of its local node, querying it for the set of packets it received and the set of receipts collected (packets it sent) over any particular time interval. The auditor publishes this information to an assigned subset of its neighboring nodes, from whom other auditors may obtain it. This level of indirection is used to guarantee that each node provides the same information to all auditors.

The second role consists of periodically auditing information about the nodes with whom their local node exchanges packets. For instance, if node A exchanges packets with nodes B, C and D in the live-streaming protocol, node A's auditor monitors information regarding these three nodes. This involves ensuring that: (1) the amount of data sent by these nodes satisfies the minimum threshold; and (2) the set of packets they claim to have received from node A corresponds to the set of packets A claims to have sent to them.

In our hybrid model of auditing, global auditors only respond and act upon violations flagged by the local auditors. In order to avoid delayed detection, local auditing works continuously within small groups of nodes. We argue that a variable threshold yields better results than a static one, leaving it to the global auditors to decide what this value should be. Therefore, besides acting on information provided by local auditors, global auditors also constantly sample the amount of packets sent and received by randomly chosen individual nodes, and use this information to decide what the threshold should be at any point in time.

4 Evaluation

We originally evaluated the resilience of pull-based streaming in the presence of attacks through simulation. We also implemented SecureStream using Python, and we validated the simulation results by running experiments with the real system on the Emulab testbed [18]. Emulab is a network testbed containing hundreds of nodes, in which real applications may be executed and evaluated. It allows arbitrary network topologies to be specified, leading to a controllable and repeatable environment.

4.1 Simulation

We built an event-driven simulator and simulated 200 node networks with 50ms inter-node latency. It would be possible to simulate and present results for networks with larger numbers of nodes, but a set of experiments on increasing numbers of nodes revealed that the behavior remains the same for networks as large as 5000 nodes. We opted for a smaller size but repeated each experiment 100 times to obtain better confidence in our results.

The target streaming rate in the experiments was fixed to 300 Kb/s, and packets of 10 Kb were used. Higher streaming rates yielded similar results as long as the packet size is accordingly increased to maintain a rate of 30 packets/s. Each streaming session lasted for 200 seconds. In the basic setting, the seed's upload capacity was fixed to twice the streaming rate while other peers had a fixed maximum upload capacity of 1.2 times the streaming rate. These values are used as our baseline since they are the lowest upload rates at the seed and non-seed nodes respectively that lead to good throughput when the system is not under attack.

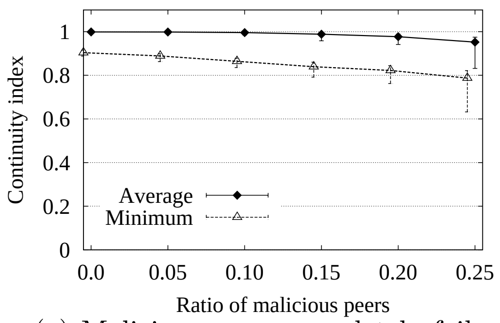
For each streaming session we computed the average and minimum download and upload rates across all correct members. We repeated each experiment 100 times, and we present the median and 95 percentile intervals across these repetitions.

We considered four types of malicious behavior. In the first type of attack malicious peers act as failed, neither requesting nor satisfying requests. In attack 2 they request packets but do not forward any packets. In attack 3 they overrequest packets from their neighbors, requesting as many distinct packets as possible from every neighbor. Finally, in attack 4 they overrequest packets and do not forward packets. The fourth type of attack is the most disruptive type and therefore the most likely, while the other three are considered mainly for comparison purposes.

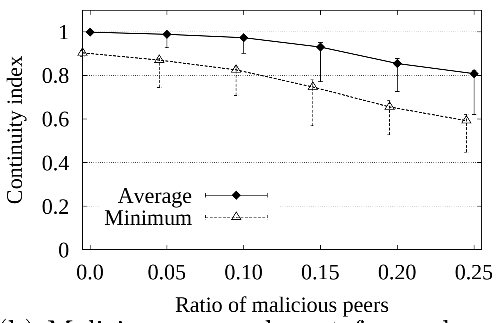
Figure 6 presents results for the basic setting under each of the attack types. We are interested in minimizing the overall damage to the streaming session. Damage is quantified by the impact on average download rates to healthy nodes, and the minimum download rate for any single healthy node.

As would be expected, the results show that peer failure does not significantly affect the download rates since peers can still request packets from other correct neighbors (Figure 6(a)). Since malicious peers do not request packets in this mode, they do not disrupt the total overall upload capacity. Even though upload rates are limited, overrequesting attacks are also not significantly disruptive, due to the random policy used by peers when satisfying neighbors' requests for packets and the upper limit on the number of outstanding requests by any neighbor (Figure 6(c)).

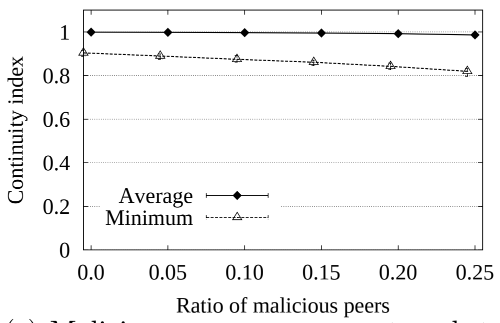
Figures 6(b) and 6(d) show that attacks in which peers consume packets from their neighbors, but do not forward packets, inflict the most harm. There are two main reasons for this vulnerability. First, since peers upload at a maximum rate of 1.2 times the streaming rate, the overall upload capacity of the system gets compromised from peers consuming and not contributing to the system. Second, malicious nodes neighboring the seed might impede some packets from ever being received by any other peer other than itself.



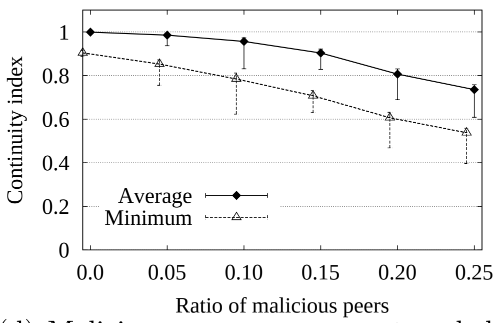
(a) Malicious peers completely fail



(b) Malicious peers do not forward any packets



(c) Malicious peers overrequest packets



(d) Malicious peers overrequest and do not forward packets

Fig. 6. Resilience under different types of Byzantine behavior and varying ratios of attackers

The latter effect causes the 95 percentile interval bars to be wide: there is a lot of variation depending on the number of compromised peers near the seed. To make this point clear, in Figure 7 we show the percentage of packets received by increasing numbers of peers during sample streaming sessions with varying ratios of Byzantine peers. The metric to focus on here is the fraction of packets only received by one peer, which is an indicator of malicious nodes neighboring the seed. Packets received only by malicious peers at the first hop will never be disseminated in the system. To confirm this hypothesis, we executed the same set of experiments and restricted the malicious attackers to being located at least 2 hops away from the seed. The obtained medians were very close to the medians obtained in the previous experiments. The main difference was that the percentile intervals were significantly reduced when the seed had no immediate malicious neighbor, which is an important result since the intervals are significant in the original experiments with attacks 3 and 4.

To improve the resilience, we can vary parameters to improve the overall upload capacity of the system, or to avoid situations in which malicious peers can isolate certain packets. First, we considered the upload capacity of the members. In Figure 8(a) we varied the value from 1.0 to 2.0 times the streaming rate and verified the improvements to resilience under attack 4. This graph presents the average and minimum download rates when the system has 25%

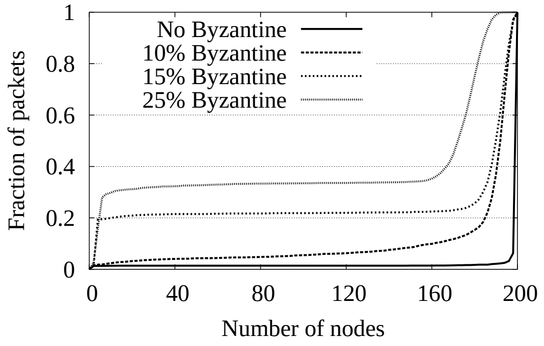


Fig. 7. CDF: Fraction of packets received by given number of nodes

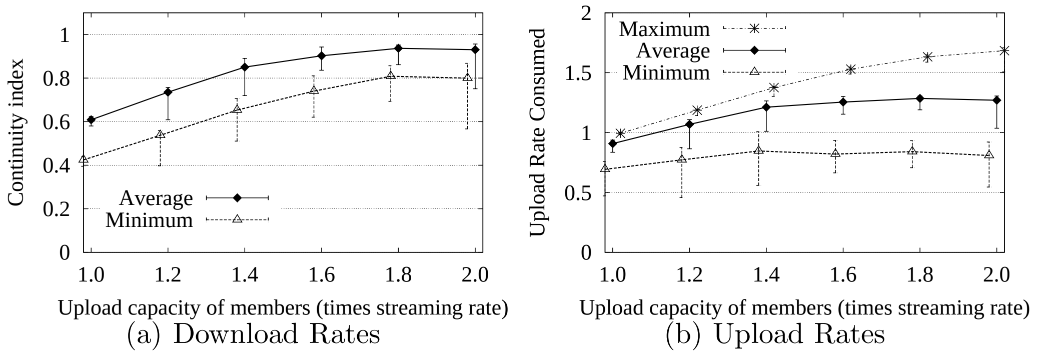
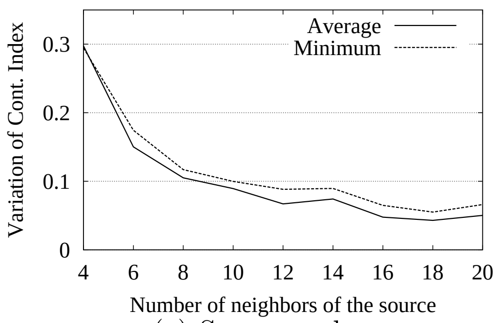


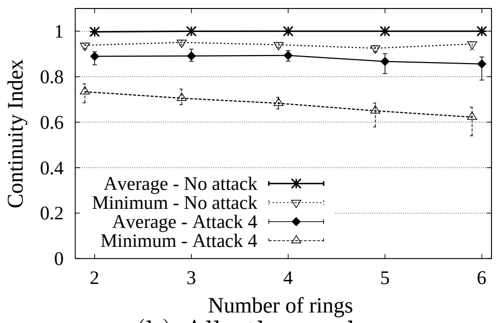
Fig. 8. Download and upload rates across nodes when maximum upload capacity is varied

of Byzantine members. The results show that the higher the upload capacity at non-seed peers the more resilient the system becomes. From Figure 8, which presents the minimum, average and maximum upload rates of members, we can see that as a consequence of increasing the upload capacity of peers the system becomes more unfair, with an increased difference between the maximum upload rate and minimum upload rate across peers. For the next few experiments we fixed the upload capacity of non-seed members to 1.4 times the streaming rate.

To improve the packet loss ratio at the first hop from the seed, we varied the upload capacity of the seed from 1.0 all the way to 6.0 times the streaming rate. Our results indicated that this naive approach to increasing the upload rate at the seed does not significantly affect the resilience of the system. We also observed that the number of neighbors of the seed is a more significant parameter than the upload capacity of the seed. We fixed the ratio of malicious nodes at 25%, the upload rate at non-seed nodes to 1.4 times the streaming rate and at the seed to 4.0 times the streaming rate, and varied the seed's number of neighbors from 4 to 20. The median slightly improves as the number of neighbors is increased, but more important, the percentile intervals are significantly reduced. In Figure 9(a) we present the absolute sizes of the 95 percentile intervals varying with the number of neighbors of the seed. The results show that a larger number of neighbors at the seed is desirable. This



(a) Source node



(b) All other nodes

Fig. 9. Sensitivity to number of neighbors

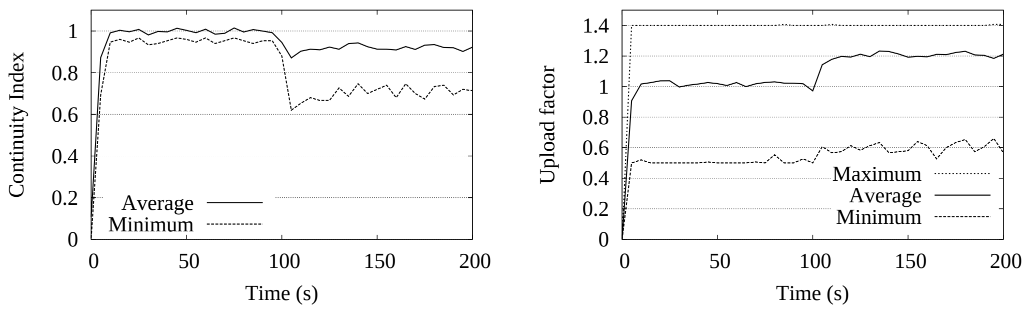
happens because with a higher number of neighbors the percentage of malicious neighbors of the seed tends to be closer to 25% across runs, and therefore there is less variation in the ratio of packets that are contained at the first hop from the seed.

Finally, to study the influence of the number of neighbors for each non-seed peer in the system, we evaluated the resilience with a varying number of rings used to define neighbors. The upload capacities at the seed and non-seed members were fixed to 4.0 and 1.4 times the streaming rate, respectively, and the seed had 16 neighbors. In Figure 9 we present the performance of the system using between 4 and 12 neighbors per node, both under no attacks and under attacks of type 4. The results surprisingly show that the use of larger numbers of neighbors does not improve resilience of the system, and even reduces when the system is under attack. Even though larger numbers of neighbors would lead to better connectivity between correct members, it also presents malicious members with more potential to overrequest packets and unbalance the system.

4.2 Emulab Testbed

In order to validate our simulation results, we ran experiments on a 200 node LAN on the Emulab testbed using our Python implementation of the Secure-Stream system. We performed extensive experiments under various parameter configurations, observing that the tendencies observed were similar to those verified through simulation.

To illustrate the behavior of the real system in execution, we present results of a sample streaming session in which 25% of the nodes are malicious, overrequesting packets and not forwarding them to neighbors. During the first 100 seconds all nodes act correctly, after which the malicious nodes start overrequesting and not forwarding packets. We fixed the upload capacity of the seed and non-seed members to 4.0 and 1.4 times the streaming rate respectively,



(a) Download Rates (b) Upload Rates

Fig. 10. Sample streaming session on Emulab

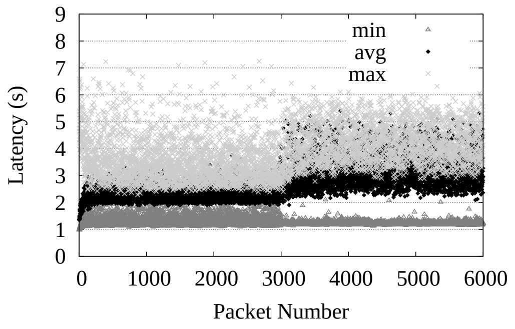
and the number of neighbors of the seed and non-seed members to 12 and 8 respectively.

Figure 10(a) presents the minimum and average continuity indices of correct peers throughout the sample session. Around the hundredth second, the average and minimum continuity indices decrease with the insertion of malicious peers. The minimum, average and maximum upload factors across all correct peers is presented in Figure 10(b). At the point when malicious nodes are inserted, the upload factors across correct peers increases to compensate for the malicious peers consuming the scarce resources from the system.

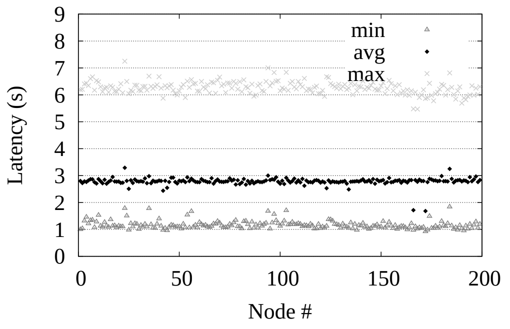
We also observed the effect of the system in the latency of packets. In Figure 11(a), a slight increase in the overall average and maximum delays per packet in the presence of attackers may be observed. Furthermore, an interesting behavior can be observed in Figure 11(b), which presents the minimum, average and maximum packet delays for each node in the system, relative to the time of origin of the packet at the source.

Unlike our initial suspicion, all nodes presented similar packet delays over the streaming session. This indicates that being close to the source does not imply in receiving packets faster than other nodes, since not all packets will be requested from the source, because of the limit in number of outstanding requests. To verify if this behavior might vary when the system scales to larger numbers of nodes, we simulated networks with up to five thousand nodes. The maximum latency used for bigger networks needs to be increased, but the average latency per node is still similar.

We also looked into the number of hops taken by packets before reaching all nodes. For the same streaming session, we registered the number of hops taken by each packet before reaching each node. In Figure 12 we present the CDF of the average and maximum number of hops taken by packets. This graph shows that for our sample session with 200 nodes, the average number of hops mostly varies between 3 and 5, while the maximum number of hops taken by each packet varies between 8 and 22. This large variation in the maximum



(a) Latency Per Packet



(b) Latency Per Node

Fig. 11. Latency of packets on Emulab

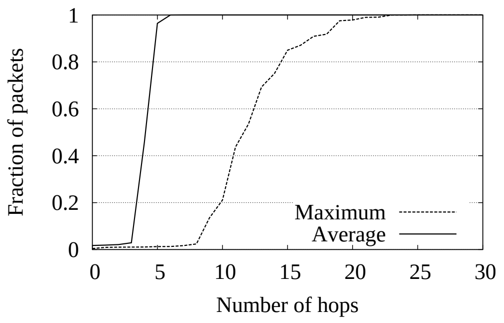


Fig. 12. CDF of average and maximum number of hops taken by packets

number of hops is also verified by the large variation in maximum latency observed in Figure 11(a).

4.3 Auditing

We also explored the potential of employing auditing in our simulations. Auditing ensures that nodes contribute more than a particular threshold factor t of upload capacity. A threshold of 0.5 during a 300 Kb/s streaming session, for instance, implies that nodes uploading less than 150 Kb/s will be removed from the system. To evaluate the effect of applying such thresholds both in the absence and presence of freeloading nodes (nodes that voluntarily contribute less than what is expected from them), we simulated audited sessions with 1000 nodes.

Figure 13 presents a detailed set of results on applying different thresholds to different freeloading profiles. The ratio of freeloading nodes was fixed to 30%, and their contribution factor (ratio relative to the streaming rate) is varied between 0, 0.25, 0.50, and 0.75 (0 meaning they do not contribute at all). We also consider a final profile named *mix*, where freeloading nodes have different contribution factors, uniformly distributed among 0, 0.25, 0.50 and 0.75.

Columns are clustered based on the threshold used to punish nodes (t). Within

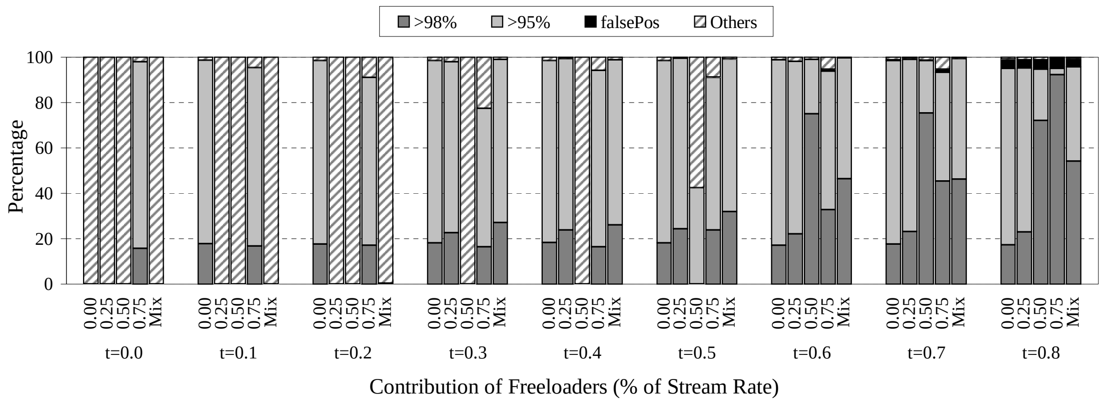


Fig. 13. Quality of streaming, measured by the percentage of correct nodes that receive over 95% and 98% packets, and percentage of nodes unfairly punished by the auditing system (falsePos). All experiments contain 30% freeloading nodes. Each cluster corresponds to a different threshold t being applied by the auditor. Within each cluster we considered 5 rates of contribution by freeloaders: 0, 0.25, 0.5, 0.75, and a mix of these values. Correct nodes contribute at a rate of 1.1.

each column we present the percentage of correct nodes that have an average upload factor of more than 98%, between 95% and 98%, false positives and others. False positives are correct nodes who get incorrectly punished by the auditing system. Notice that a threshold of 0 is equivalent to a system without auditing.

To help understand the graph, let us consider, for instance, the set of bars when $t = 0.4$. A threshold of 0.4 will detect and remove all freeloaders with an upload factor of 0 or 0.25. This may be confirmed in the first two bars, which indicate that the streaming quality is good, with almost all nodes receiving more than 95% of the data. The third bar, in which freeloaders have an upload factor of 0.5 presents unsatisfactory results, with no node receiving over 95% of the data. This was expected, since a threshold of 0.4 is not able to detect freeloaders that contribute with a factor of 0.5.

In the fourth bar, even though freeloaders do not get detected, they also do not disrupt the system significantly, since they contribute at a factor of 0.75, which is close to the ideal factor of 1.0. This is confirmed by the fact that even when there is no auditing (threshold is 0), the quality of the stream is satisfactory . The same observation holds for the *mix* configuration.

Two metrics are important when deciding the right threshold to apply: the quality of the streaming, captured by the percentage of nodes receiving over 95% of data; and the ratio of false accusations, which should be ideally null. From Figure 13 it is reasonable to assume that opting for a threshold such as $t = 0.6$ is the best approach, since it provides satisfactory streaming quality under the 5 different configurations. However, the goal of minimizing the ratio of false positives motivates the use of a dynamic threshold value, adjusted by global auditors based on sampling the current stream quality and upload factors of nodes across the system. One possibility consists in maintaining a null threshold (t = 0) while the actual download rates across the network are satisfactory, that is, not punishing nodes unless the performance of the session is compromised. As the system starts to degrade, global auditors may slowly increase the threshold until the performance improves again.

5 Related Work

Recent work on peer-to-peer streaming systems has focused on improving fairness among peers and resilience to churn, and have not addressed behavior in the presence of malicious peers. Splitstream [4] breaks the data into stripes and disseminates each stripe through a different dissemination tree. Ideally, each peer is an internal node in only one these trees, and therefore the system as a whole is fair. Figures 1(b) and 1(d) present SplitStream's resilience to omission attacks. Bullet [5] is another protocol which attempts to improve fairness by breaking the stream into packets and sending them to peers through different dissemination paths. Packets are pushed down a tree to certain peers and then exchanged between peers through random connections.

The pull-based style of streaming used in our system was previously used in CoolStreaming [1] and Chainsaw [6]. Coolstreaming breaks the data into packets and peers organized into a mesh request packets from their neighbors using a scheduling algorithm to identify the best sources of packets. Chainsaw uses a simpler policy for requesting packets from neighbors, randomly fetching packets from neighbors with available packets respecting only a limit on the number of outstanding requests. Chainsaw presents smaller delays for the receipt of packets compared to the Coolstreaming protocol.

Omission attacks are often characterized as rational behavior and there has been a lot of work regarding incentives for peer-to-peer systems. Most work on incentives has focused in file sharing systems such as Bittorrent [19], which present significantly different properties, and cannot be directly transferred to streaming protocols. Some more recent work has focused on rational behavior in live-streaming systems.

Ngan et al. [20] consider fairness issues in the context of tree-based peer-to-peer streaming protocols. The authors present mechanisms that can distinguish peers according to their level of cooperation to the system. One of their techniques involves the reconstruction of trees as a way of punishing freeloading nodes. Most of their mechanisms require peers to keep track of their parents' and children's behavior.

PULSE [21] is a P2P live-streaming system that tries to reward nodes that contribute resources and discourage peers from contributing an insufficient amount of resources. The main idea consists in using a pull-based dissemination protocol and moving nodes that contribute more closer to the source, therefore having a smaller lag for packets received. The system makes a few assumptions which could be compromised by malicious nodes present in the system, such as requiring that nodes have some knowledge of other nodes in the system. Also, the system is only evaluated in heterogeneous settings, showing that nodes with higher upload capacity have a smaller latency compared to less favored nodes. Results in a homogeneous setting are not presented.

In [22], the use of incentives is explored as a way of avoiding the presence of selfish nodes in the Chainsaw protocol [6]. This work argues why some naive approaches to enforcing incentives do not work, similar to the analysis presented in our work, and propose and evaluate the use of a technique that relies on preferential uploading to neighbors. Nodes that contribute more are more prone to receiving data back, but in a not so fixed manner as a tit-for-tat approach. Only preliminary results are presented, and malicious behavior is not the focus of the work.

Even though incentives encourage nodes to contribute and avoid nodes from acting selfishly, they do not extend the effect to nodes who are in the system with malicious intentions.

Drum[23] targets DoS attacks on gossip-based multicast protocols, eliminating vulnerabilities to such attacks. The main idea in Drum is to have half of the links of each peer be picked by the peer itself, and half be picked by other peers. That way, even if only malicious peers connect to a peer, the peer can still get correct data from the peers that it picks. The authors showed that the approach works well for multicast protocols which do not have time delays, but have not studied its performance for multicast systems where a high throughput of packets is desired and the upload capacities are limited.

BAR Gossip [24] is a live-streaming approach that tolerates the existence of selfish and malicious nodes. Time is divided into rounds, in which each peer communicates with another peer selected using a pseudo-random function. In each round, peers exchange their current history containing the identifiers of all the current data, as basis for the next exchanges. Nodes also perform a phase of *optimistic push*, forwarding useful updates to another pseudo-randomly picked peer with no guarantee of useful return. The approach requires that the broadcasting seed has full knowledge of all members in the system and always unicasts each update to 5% of the nodes, a limitation on scalability.

6 Conclusions

We presented the design and evaluation of SecureStream, a P2P live-streaming protocol tailored to handle byzantine attacks. We described the main components of SecureStream and the main techniques employed to resist against DoS, forgery, membership and omission attacks. Furthermore, we considered the benefits of employing an auditing system to avoid the damage incurred by freeloading behavior of nodes. We evaluated our system through simulation and emulation. Our results indicate that SecureStream tolerates a limited percentage of malicious nodes in the system, and that with the aid of an auditing component, it is able to provide satisfactory quality in the face of even larger attacks.

References

[1] X. Zhang, J. Liu, B. Li, T.-S. P. Yum, CoolStreaming/DONet: A Data-Driven Overlay Network for Efficient Live Media Streaming, in: Proceedings of the 2005 Conference on Computer Communications, Miami, FL, 2005.

[2] Y.-H. Chu, S. G. Rao, H. Zhang, A Case for End System Multicast, in: Proceedings of ACM Sigmetrics, Santa Clara, CA, 2000.

[3] J. Jannotti, D. K. Gifford, K. L. Johnson, M. F. Kaashoek, J. O. Jr., Overcast: Reliable Multicasting with an Overlay Network, in: Proceedings of the 4th Symposium on Operating Systems Design and Implementation, San Diego, CA, 2000.

[4] M. Castro, P. Druschel, A.-M. Kermarrec, A. Nandi, A. Rowstron, A. Singh, SplitStream: High-Bandwidth Multicast in Cooperative Environments, in: Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles, Bolton Landing, NY, 2003.

[5] D. Kostić, A. Rodriguez, J. Albrecht, A. Vahdat, Bullet: High Bandwidth Data Dissemination Using an Overlay Mesh, in: Proceedings of the 19th Symposium on Operating Systems Principles, Bolton Landing, NY, 2003.

[6] V. Pai, K. Kumar, K. Tamilmani, V. Sambamurthy, A. E. Mohr, Chainsaw: Eliminating Trees from Overlay Multicast, in: Proceedings of the 4th International Workshop on Peer-to-Peer Systems, Ithaca, NY, 2005.

[7] V. Venkataraman, P. Francis, J. Calandrino, Chunkyspread: Multitree Unstructured Peer to Peer Multicast, in: Proceedings of the 5th International Workshop on Peer-to-Peer Systems, Santa Barbara, CA, 2006.

[8] R. van Renesse, H. Johansen, A. Allavena, Fireflies: Scalable Support for Intrusion-Tolerant Network Overlays, in: Proceedings of the 1st ACM EuroSys, Leuven, Belgium, 2006.

[9] R. Canetti, J. Garay, G. Itkis, D. Micciancio, M. Naor, B. Pinkas, Multicast Security: A Taxonomy and Some Efficient Constructions, in: Proceedings of IEEE INFOCOMM, Orlando, FL, 1999.

[10] A. Singh, M. Castro, A. Rowstron, P. Druschel, Defending against Eclipse Attacks on Overlay Networks, in: Proceedings of the 11th ACM SIGOPS European Workshop, Leuven, Belgium, 2004.

[11] C. K. Wong, S. S. Lam, Digital Signatures for Flows and Multicasts, IEEE/ACM Transactions on Networking IEEE Communications Society 7.

[12] R. Gennaro, P. Rohatgi, How to sign digital streams, in: Proceedings of the 17th Annual International Cryptology Conference on Advances in Cryptology, London, UK, 1997.

[13] S. Miner, J. Staddon, Graph-Based Authentication of Digital Streams, in: Proceedings of the 2001 IEEE Symposium on Security and Privacy, Washington, DC, 2001.

[14] D. Song, J. D. Tygar, D. Zuckerman, Expander Graphs for Digital Stream Authentication and Robust Overlay Networks, in: Proceedings of the 2002 IEEE Symposium on Security and Privacy, Washington, DC, 2002.

[15] A. Perrig, R. Canetti, D. Tygar, D. Song, The TESLA Broadcast Authentication Protocol, Cryptobytes 5 (2).

[16] A. Perrig, R. Canetti, D. X. Song, J. D. Tygar, Efficient and Secure Source Authentication for Multicast, in: Proceedings of the Network and Distributed System Security Symposium, The Internet Society, San Diego, CA, 2001.

[17] A. Perrig, R. Canetti, J. D. Tygar, D. X. Song, Efficient Authentication and Signing of Multicast Streams over Lossy Channels, in: IEEE Symposium on Security and Privacy, Berkeley, CA, 2000.

[18] B. White, J. Lepreau, L. Stoller, R. Ricci, S. Guruprasad, M. Newbold, M. Hibler, C. Barb, A. Joglekar, An Integrated Experimental Environment for Distributed Systems and Networks, in: Proceedings of the Fifth Symposium on Operating Systems Design and Implementation, Boston, MA, 2002.

[19] B. Cohen, Incentives Build Robustness in BitTorrent, in: 1st Workshop on the Economics of Peer-to-Peer Systems, Berkeley, CA, 2003.

[20] T.-W. J. Ngan, D. S. Wallach, P. Druschel, Incentives-Compatible Peer-to-Peer Multicast, in: Second Workshop on the Economics of Peer-to-Peer Computing, Cambridge, MA, 2004.

[21] F. Pianese, J. Keller, E. W. Biersack, PULSE, a Flexible P2P Live Streaming System, in: Proceedings of the Ninth IEEE Global Internet Workshop, Barcelona, Spain, 2006.

[22] V. Pai, A. E. Mohr, Improving Robustness of Peer-to-Peer Streaming with Incentives, in: Proceedings of the 1st Workshop on the Economics of Networked Systems, Ann Arbor, MI, 2006.

[23] G. Badishi, I. Keidar, A. Sasson, Exposing and Eliminating Vulnerabilities to Denial of Service Attacks in Secure Gossip-Based Multicast, in: International Conference on Dependable Systems and Networks, Philadelphia, PA, 2004.

[24] H. C. Li, A. Clement, E. L. Wong, J. Napper, I. Roy, L. Alvisi, M. Dahlin, BAR Gossip, in: Proc. of the 7th Symposium on Operating Systems Design and Implementation (OSDI), Seattle, WA, 2006.

Enforcing Fairness in a Live-Streaming System*

Maya Haridasan[a], Ingrid Jansch-Porto[b] and Robbert van Renesse[a]

[a]Dept. of Computer Science, Cornell University
Ithaca, New York
[b]Institute of Informatics, Federal University of Rio Grande do Sul
Porto Alegre, Brazil
maya@cs.cornell.edu, ingrid@inf.ufrgs.br, rvr@cs.cornell.edu

ABSTRACT

We describe a practical auditing approach designed to encourage fairness in peer-to-peer streaming. Auditing is employed to ensure that correct nodes are able to receive streams even in the presence of nodes that do not upload enough data (opportunistic nodes), and scales well when compared to previous solutions that rely on tit-for-tat style of data exchange. Auditing involves two roles: local and global. Untrusted local auditors run on all nodes in the system, and are responsible for collecting and maintaining accountable information regarding data sent and received by each node. Meanwhile, one or more trusted global auditors periodically sample the state of participating nodes, estimate whether the streaming quality is satisfactory, and decide whether any actions are required. We demonstrate through simulation that our approach can successfully detect and react to the presence of opportunistic nodes in streaming sessions. Furthermore, it incurs low network and computational overheads, which remain fixed as the system scales.

1. INTRODUCTION

Video and audio streaming account for a large percentage of content accessed over the web. One popular style of streaming on the web is *on demand*, in which users access pre-stored content at will. Another style requires streams to be generated and disseminated in real-time. This may be the case with important social, political, or sporting events. An important property of *live-streaming* is that data is not available in advance, being generated just before transmission at the sender. Furthermore, interested users ideally want to receive the stream without much delay from its original transmission.

Several practical live-streaming systems now allow large numbers of interested users to receive streamed data in near real time, without requiring extensive amounts of resources. These systems are based on the peer-to-peer (P2P) paradigm, where nodes interested in receiving data also help disseminate it to each other, alleviating the bottleneck at the source. Initial protocols were based on building a tree-based overlay of nodes through which data would be pushed.[1–3]

More recent systems, such as Chainsaw and Coolstreaming, have shown that the use of a mesh of connected nodes and a pull-based data dissemination approach can provide similar results with better resilience to failures and churn (nodes joining and leaving the system).[4–7] In Chainsaw, for example, nodes notify each other of receipt of data packets, and request packets from their neighbors based on the received notifications. Practical systems based on pull-based streaming now exist in China, where they are used to disseminate television channels to thousands of users.[8]

Even though the P2P paradigm allows systems to scale with the number of users, it also leaves them vulnerable to opportunistic behavior. Opportunistic nodes attempt to receive a stream without uploading their fair share of data, reducing the overall upload capacity of the system. Despite the damage that they may cause, not much work has been done in studying mechanisms to avoid their presence in live-streaming systems. The goal of this

*The authors were supported by AFRL award FA8750-06-2-0060 (CASTOR), NSF award 0424422 (TRUST), AFOSR award FA9550-06-1-0244 (AF-TRUST), DHS award 2006-CS-001-000001 (I3P), and Intel Corporation. The views and conclusions herein are those of the authors.

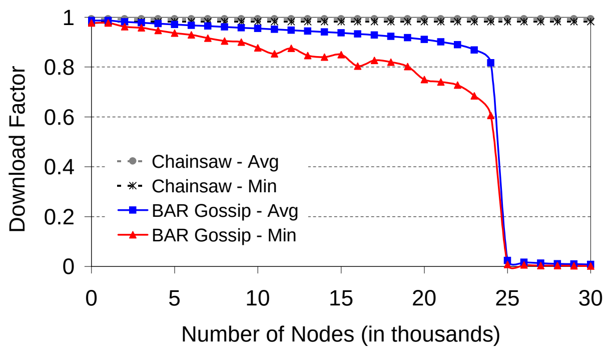


Figure 1. Minimum and average download rates across all nodes when using the BAR Gossip and Chainsaw protocols.

paper is to propose and evaluate a mechanism that can defend against this problem, whithout incurring large overheads.

The approach that most closely relates to our work is the BAR Gossip protocol,[9] which employs a *tit-for-tat* approach for encouraging nodes to contribute: a node only sends as much data to another node as it receives back. It provides an elegant solution shown to tolerate both opportunistic behavior and other malicious attacks. However, reliance on tit-for-tat does present a few undesirable requirements. To be efficient, the data source should ensure that packets are evenly spread across the system by sending data to a fixed proportion of nodes, and by sending different packets to different nodes. Furthermore, it requires the source and all nodes to have full membership knowledge. These restrictions affect scalability when the data source has bounded upload bandwidth.

To illustrate this problem, we fixed the upload capacity of a data source at 5 Mbps and simulated BAR Gossip when streaming 500 Kbps with increasing numbers of receivers, varied between one and thirty thousand nodes. We compare its scalability against the Chainsaw protocol,[4] for which we fixed the source's upload bandwidth to 2 Mbps. In Figure 1, we present the average and minimum download rates (as ratios of the stream rate) of both protocols when the number of nodes is increased. As observed, BAR Gossip is not able to sustain its performance without scaling the upload capacity of the source proportionally with the size of the system. Meanwhile, Chainsaw is able to scale well even with a fixed lower upload bandwidth at the source, but cannot handle the presence of opportunistic nodes.

We propose to use auditing to encourage data-sharing in live-streaming systems like Chainsaw. Our auditing approach establishes a minimum threshold for the amount of data sent by any node in the system, and removes nodes that upload less data than the threshold. Instead of relying on a tit-for-tat mechanism, we focus on encouraging nodes to respect the established protocol. Nodes are forced to provide accountable information regarding packets sent to and received from neighbors, and the auditing system is responsible for detecting and removing misbehaving nodes.

Notice that identifying the misbehaving nodes is not a trivial task, since there is no fixed minimum amount of data that nodes should contribute to the system. If we assume a model where misbehaving nodes simply did not upload any data, detecting them would be an easier task. However, once we assume that misbehaving nodes may adjust their contribution level based on the policy used by an auditing system, a more elaborate approach is required. This paper presents and evaluates an auditing model based on sampling the system and using the sampled information to build a global view of how the system is currently behaving. Based on it, auditors employ strategies to identify the misbehaving nodes that should be punished.

The paper is organized as follows. In section 2, we state the exact problem that we aim to solve and the assumptions considered in this work. In section 3, we review the pull-based streaming protocol employed in our system, followed by a description of our novel auditing approach in section 4. In section 5, we evaluate the proposed approach. We then discuss the costs of auditing, and briefly describe how to extend our model for heterogeneous systems, in section 6. Finally, we present related work in section 7, and conclude in section 8.

2. PROBLEM STATEMENT

Our approach focuses on a target streaming system consisting of one data *source* (assumed non-compromised), which disseminates data at a fixed rate to a dynamic set of receivers. The source has limited upload bandwidth, and hence can only send data directly to a small subset of interested receivers. Participating nodes are consequently required to forward packets to their neighbors, helping disseminate all packets across the system. The streamed data should be received by all nodes within a fixed latency from the source's original transmission, even in the presence of opportunistic nodes.

For simplicity, we first assume a system in which all nodes, except the source, have similar upload and download bandwidths; in Subsection 6.2, we briefly discuss how to extend our model to work in heterogeneous scenarios.

We assume that malicious nodes exhibit Byzantine behavior, while correct nodes follow the protocol as defined, requesting data as needed and sending data as requested from them. Altrustic nodes are a subgroup of correct nodes that are willing to upload more data than required from them. Finally, we employ the term *opportunistic* to refer to a subgroup of Byzantine nodes that attempt to give less data than they would if they behaved as correct nodes, with the intention of obtaining as much data as possible at least feasible cost. These may employ a simple strategy, such as refuse to contribute any upload resources, or a more elaborate strategy that allows them to cheat without being easily detected.

Notice that our model diverges from the one used in BAR Gossip,[9] in which nodes are classified as *Byzantine*, *Altruistic*, or *Rational*. In that model, rational nodes attempt to maximize their utility while still following the defined protocol. Our model is actually less lenient: nodes employing strategies to maximize their utility are classified as Byzantine, so that we can build a practical punishment-based system in which any node not contributing its fair share of data may be expelled from the system.

Throughout the paper we use the terms *upload factor* and *download factor* to refer to the ratio between an upload or download rate and the original stream rate. For example, given a stream rate of 500 Kbps, a download rate of 400 Kbps corresponds to a download factor of 0.8.

3. STREAMING SYSTEM MODEL

Our auditing approach is used over the Chainsaw protocol.[4] All nodes participating in the system are organized into a fully connected mesh overlay, where each node has the same number of neighbors. The source is randomly connected to a small subset of the nodes.

The streaming process starts at the source, which breaks the data stream into packets and sends notifications to its neighbors as soon as it has packets to disseminate. These notifications are small messages used only to inform neighbors of the availability of new packets. Based on the received notifications, each node requests missing packets, and the source satisfies as many requests as allowed by its upload capacity. Unlike BAR Gossip, with Chainsaw the upload capacity of the source does not need to increase with the size of the system; even an upload capacity of twice the stream rate is sufficient to ensure that the system performs and scales well.

As nodes receive packets, they mimic the role of the source, sending notifications to their own neighbors in the mesh, allowing packets to be propagated through the system. This pull-based approach to acquisition of packets (notify-request-send data) provides some resilience to failure or malicious behavior, since a participant will have multiple possible sources for each packet. The mesh overlay defines a predetermined set of neighbors for each peer, which also makes it hard for malicious peers to round up on individual peers since attackers lack a deterministic means of acquiring control of all of its neighbors. All nodes with exception of the source have a fixed upper limit on their upload contribution (e.g. 1.2 times the stream rate), defined by the protocol. Of course, this upper limit is not respected by opportunistic nodes, who attempt to reduce it with the goal of uploading less data.

On the course of a streaming session, each node stores packets and forwards them to other peers only while the packet is within its *availability window*, usually spanning a few seconds. Each node also maintains an *interest window*, which represents the set of packets in which the peer is currently interested. Nodes choose packets to request from each of its neighbors, respecting a maximum limit l on the number of outstanding requests to each

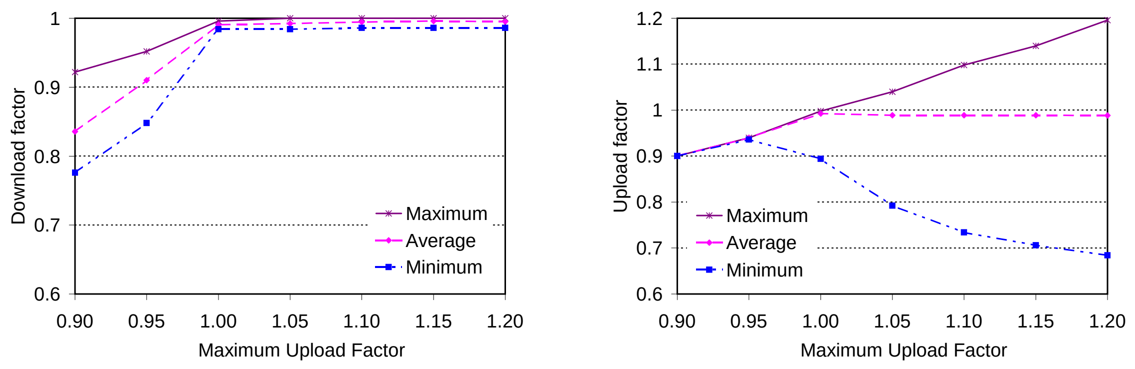


Figure 2. Download and upload factors of nodes in an ideal system where all nodes behave correctly.

neighbor. This limit not only improves the general flow of packets, but also makes it harder for malicious peers to overrequest packets from their neighbors: peers maintain a queue of non-satisfied requests from its neighbors, keeping only the l most recent ones.

3.1 Expected Behavior

Our first goal is to explore the typical signature of the system, since an understanding of the behavior of pull-based dissemination in the absence of opportunistic nodes will turn out to be important when we set out to introduce auditing. We conducted experiments using an event-based simulator, which is described in more detail in section 5.

In Figure 2, we evaluate the performance of 1000 nodes during an ideal execution of Chainsaw, where all the nodes behave correctly. We fixed the upload factor of the source at 4.0 (2 Mbps), and the stream rate to 500 Kbps. We varied the maximum upload factor of nodes to see how it affected both the download and upload factors of nodes across the system. The maximum upload factor is a fixed parameter which defines the maximum rate at which a node will upload data to all its neighbors. For fairness in nodes' bandwidth consumption, we would like all nodes to upload data at a factor as close as possible to 1.0. We varied the maximum upload factor of nodes from 0.9 to 1.2.

The left graph shows the minimum, average and maximum download factors across the nodes when the maximum upload factor of nodes is increased. As observed, by increasing the maximum upload factor of nodes, we increase the global upload capacity of the system, leading to a better flow of packets. However, the discrepancy among the upload factors of individual nodes also increases, as seen in the graph to the right. When the maximum upload factor is increased, some nodes participate more actively in dissemination while others end up contributing less, even though all of them are behaving correctly. This is an important consideration: when we introduce auditing, we do not want to punish nodes that are willing to contribute but cannot do so because of factors such as their physical positioning in the system. In all our future experiments we set the maximum upload factor to 1.1.

3.2 Effect of Opportunistic Behavior

Our next goal was to understand the expected behavior of correct nodes under different scenarios where opportunistic nodes compromise the system. We therefore studied how the download and contribution rates of correct nodes are affected under these conditions. Opportunistic nodes may contribute with some data in an attempt to disguise their opportunistic behavior. Therefore, we considered different rates of contribution for opportunistic nodes: 0 (pure freeloaders), 100, 200, 300 and 400 Kbps.

Figure 3 presents the average and minimum download factors among all correct nodes under different configurations. The stream rate was fixed at 500 Kbps, and all correct nodes had a maximum upload factor of 1.1 (550 Kbps). We ran experiments with 1000 nodes and increasing percentages of opportunistic nodes in the system (from 0 to 90%). On the x-axis, we vary the percentage of opportunistic nodes. As expected, we can observe that the download factors of correct nodes decreases since the aggregated upload capacity in the system becomes

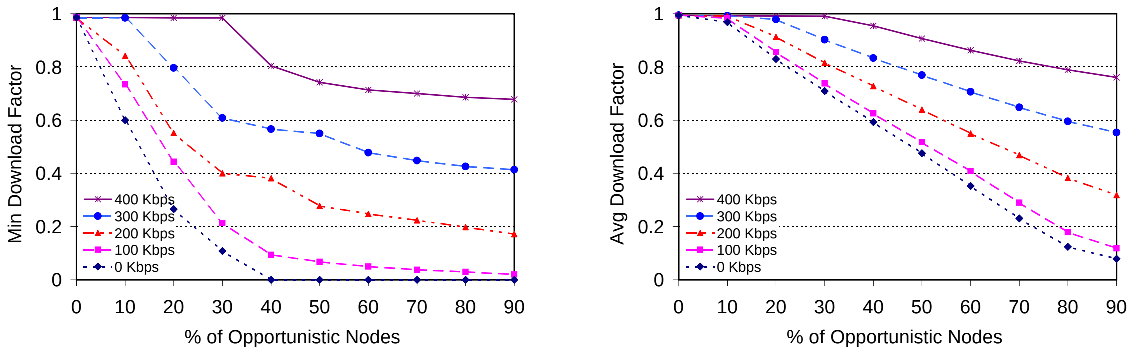


Figure 3. Minimum and average download factors across all correct nodes when opportunistic nodes are present. Each curve corresponds to a different contribution rate used by opportunistic nodes.

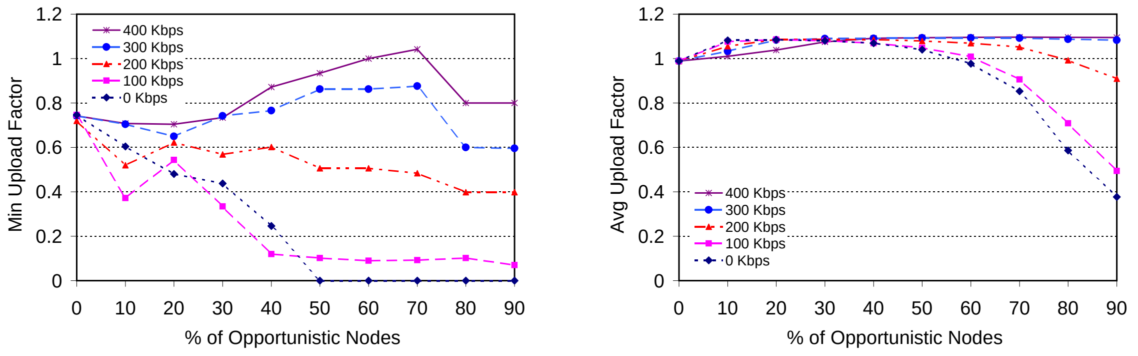


Figure 4. Minimum and average upload factors across all correct nodes when opportunistic nodes are present. Each curve corresponds to a different contribution rate used by opportunistic nodes.

insufficient to provide all nodes with all data. Nonetheless, the extent of the impact may be surprising: with just 10% opportunistic nodes, performance drops by as much as 40%.

Figure 4 presents the average and minimum upload factors among all correct nodes. Once again, on the x-axis we vary the percentage of opportunistic nodes, and on the y-axis we present the upload factors of nodes, which can vary up to 1.1. It is interesting to note that the average upload factor among correct nodes initially increases, and then starts falling when the percentage of opportunistic nodes increases significantly. This behavior can be explained by the fact that, initially, correct nodes start contributing more to compensate for the lack of data provided by a small percentage of opportunistic nodes; however, once the effect of opportunistic nodes becomes significant, the system collapses and correct nodes are not able to keep contributing.

Another important point to note is that the minimum upload factor does not follow a clearly defined pattern, making it hard to estimate the minimum contribution of correct nodes under compromised scenarios. Therefore, by applying thresholds to punish opportunistic nodes, correct nodes may also be unfairly penalized.

4. AUDITING PROTOCOL

Our idea for auditing the described live-streaming system against opportunistic behavior is motivated by the graphs presented in the previous section: we propose to employ auditing to ensure that all nodes in the system contribute more than a particular specified threshold. In Figure 5, we illustrate the potential benefit from using auditing in a system where 70% of the nodes are correct and 30% are opportunistic. The latter do not upload any data. During the first 100 seconds, no punishment was applied in an attempt to simulate a system with no auditing. At time t = 100s, auditing is enabled and opportunistic nodes start to be expelled from the system for low contribution. For this experiment, the minimum upload factor for nodes to stay in the system was set to 0.5.

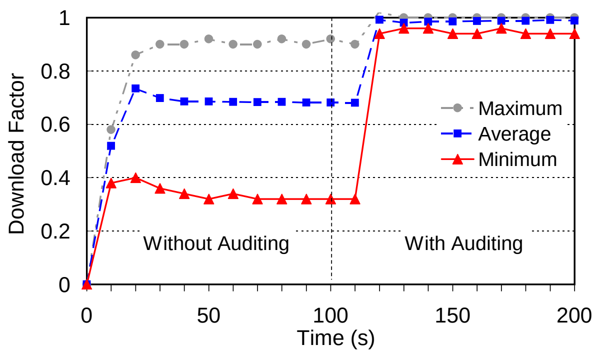


Figure 5. Download factor of correct nodes during a 200 second streaming session with 30% opportunistic nodes. Auditing is enabled in the last 100 seconds.

We present the minimum, average and maximum download factors across correct nodes varying along 200 seconds. As observed in this particular example, auditing has the potential to improve the quality of streamed sessions significantly, and at low cost. One important concern is that if the specified threshold is too high, more opportunistic nodes may be caught, but correct nodes may also be unfairly punished. In this experiment, no correct nodes were mistakenly expelled from the system.

4.1 Auditing components

We now give some additional details of the auditing architecture, focusing upon two aspects: (1) collecting accountable information about the download and upload factors of individual nodes in the system; and (2) establishing and applying the best threshold at any given time during execution. We employ two types of components to perform these two roles: local and global auditors. Local auditors are executed on the nodes participating in the system, and therefore cannot be trusted; if a node is malicious, it might report false data. Global auditors are trusted components that run on dedicated external nodes. There can be just one or a few global auditors. We describe their roles and interactions in detail below.

4.1.1 Local Auditors

Each node n runs a local auditor, which interacts with other local auditors and has two main roles:

Publish n's data exchange history: n's local auditor periodically compiles and distributes the history of packets exchanged by n. To acomplish this, every δ seconds, it queries the local streaming application running on n for the set of packets it sent and received using the streaming protocol in the most recent time interval (Figure 6). The local auditor signs and publishes the collected history to an assigned subset of its neighboring nodes, from whom other auditors may obtain it. This level of indirection is used to prevent nodes from masking their real upload and download factors by presenting different information to different auditors.

Audit n's neighbors' histories: n's local auditor periodically audits the published histories of the nodes with whom n exchanges packets. For instance, if node n exchanges packets with nodes p, q and r in the live-streaming protocol, n's local auditor compares these three nodes' histories with n's own history. This involves ensuring that: (1) the amount of data sent by these nodes satisfies the defined minimum threshold for the system; and (2) the set of packets they claim to have sent to and received from node n corresponds to the set of packets n claims to have respectively received from and sent to them. If the first check comparison fails, the local auditor issues an *accusation* against the node to a global auditor. In the second case, the local auditor is not able to prove the neighbor's misbehavior; instead, it instructs its local streaming application to not further exchange packets with the misbehaving neighbor. More complex types of checks may also be performed to address other types of Byzantine behavior.

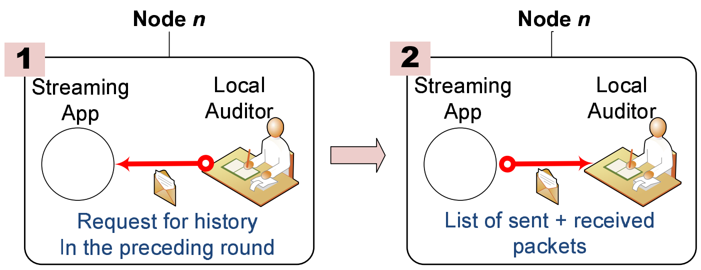


Figure 6. Local Auditing

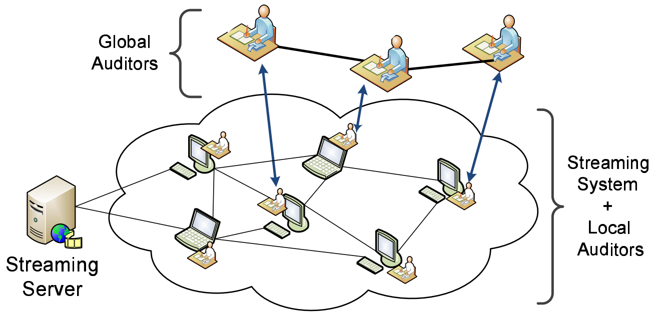


Figure 7. Global Auditing

There are two ways in which a node could pretend to be sending more or receiving less data than it actually does. It could send different histories to each neighbor, always lying about its interactions with other neighbors. For example, n could send a history to p pretending to send more data to q than it actually did, while it sends a different history to q where it pretends to send more data to p than it actually did. n's goal would be to send less data while not being caught by any of its neighbors. The process of publishing a node's history to a predefined set of neighbors ensures that the node cannot send conflicting histories to different neighbors undetected, therefore avoiding this problem.

A node could also lie about the set of packets sent to or received from a particular neighbor p. In this case, p will be able to identify that the node has lied and will therefore stop exchanging packets with n. Given that an opportunistic node's goal is to maximize its utility, it should have no interest in losing data exchange partners. Therefore, opportunistic nodes have no incentive to publish incorrect histories.

Summary: Local auditing ensures that correct information is available regarding the set of data sent and received by any node, and allows nodes to monitor each other's contribution rates.

4.1.2 Global Auditors

Global auditors are trusted components with global membership knowledge, who interact with one another and with the local auditors. As shown in Figure 7, global auditors execute on nodes external to the system. Their main roles are:

Define the minimum upload threshold: Global auditors periodically sample the state of the system by querying local auditors. They then cooperate to analyze the collected samples, and on this basis compute the minimum upload contribution threshold. Different strategies may be employed for choosing the best possible threshold, given different scenarios. Once thresholds are varied, they are gossiped to all local auditors, who then enforce the determined threshold.

Expurge nodes from the system: Global auditors are also responsible for verifying accusations issued by local auditors against particular nodes, and after validating the accusation, expurging misbehaving nodes from the system. Validation involves verifying that the accused node's history indeed indicates that the node is sending less data than the current threshold. Expurging a node involves informing the nodes' immediate neighbors of its status and forcing the removal of the node from the overlay mesh.

The number of global auditors may vary according to different parameters, such as the size of the system. The use of more global auditors distributes the load of sampling and improves efficiency in reacting to accusations against nodes. Global auditors are also perfect candidates to perform membership tasks such as acting as entry points to the P2P system, since they are required to have full membership knowledge of the system for performing their auditing roles.

Summary: Global auditing monitors the global health of the system to identify the best value for the minimum upload threshold at any time during a streaming session, and makes final decisions regarding punishment of nodes.

4.2 Adaptive Threshold Strategies

Choosing an upload threshold requires care: a low threshold may not be sufficient to identify opportunistic nodes, while high thresholds may incorrectly punish correct nodes. We considered different strategies for the choice of the minimum contribution t hreshold used for identifying misbehaving nodes.

The simplest strategy sets a fixed threshold (e.g., $t = 0.5$), independent of the current state of the system. In this case, any node contributing at a rate of less than 50% of the stream rate would be removed. One downside of using a fixed threshold is that opportunistic nodes that learn the threshold can simply contribute at the lowest possible upload factor, thus avoiding detection. From the graphs in section 3, it is clear that such a stretagy may disrupt the streaming session. Meanwhile, choosing a high threshold is not a practical option, since correct nodes would get unfairly punished.

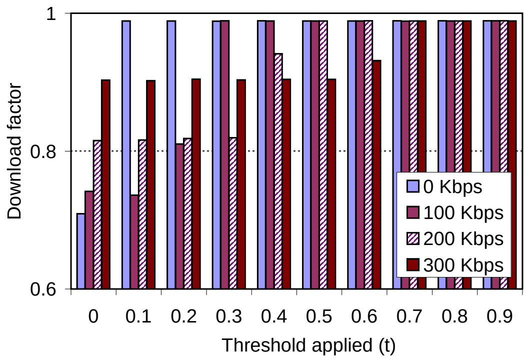
To avoid this problem, we have explored adaptive strategies. One simple strategy starts with a minimum threshold (e.g., $t = 0.5$), increasing it only if the system is compromised. Global auditors sample the system to identify the average download factor, and if this factor is lower than 0.98, increase the threshold. Once the download factor reaches a satisfactory level again, the threshold may be reduced back to its initial value. This *stepwise* approach allows the system to catch opportunistic nodes in case their presence starts affecting the performance of the system, while avoiding incorrect accusations of correct nodes.

We also considered a second adaptive strategy (*percentile-based*) for computing the threshold based on periodically sampled download and upload factors. The average download factors once again are used for detecting whether the threshold should be varied or not. In this strategy, our initial threshold is set to null, and the threshold is chosen from sampled upload factors. After each sampling, if the system seems to be in a compromised state, the collected upload factors are ordered and the value dividing the lowest 10 percent is used as the new threshold. This approach relies on efficiently sampling the system, and on fact that if the system's performance is not satisfactory, then at least 10 percent of the nodes are opportunistic.

5. EVALUATION

In this section, we evaluate the performance of our proposed auditing strategy over the original streaming protocol. We built an event-driven simulator and used it to simulate streaming sessions on networks with 1000 nodes and an average of 50ms inter-node latency. The target streaming rate in the experiments was fixed to 500 Kb/second, and all our experiments were repeated 10 times. Confidence intervals were small, and for simplicity are omitted from the graphs.

In all experiments, the source of the stream has an upload capacity of four times the stream rate (2 Mbps) and is connected to 20 arbitrarily selected nodes. Other nodes have enough download capacity to receive the stream, and upload factor of 1.1. We defined an availability window of 10 seconds and an interest window of 8 seconds. To evaluate the quality of each auditing strategy, we evaluate the average download factors of correct nodes during a 100 second time interval after auditing is first applied to the system. For the sample-based techniques, we considered that global auditors collected information from 100 nodes between each interval of 20 seconds. Notice that the sample size does not increase with the size of the system, which is a positive aspect of the auditing approach. In subsection 6.1 we discuss the costs involved in collecting these samples.



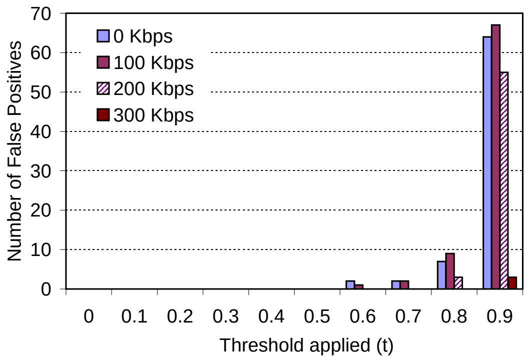


Figure 8. Quality of streaming when applying the fixed threshold strategy. Threshold is varied from 0 to 0.9 (x-axis), and the contribution rate of opportunistic nodes is varied from 0 to 400 Kbps. The first graph (left) presents the average download factors across all correct nodes. The second graph (right) presents the number of correct nodes incorrectly punished (false positives).

In Figure 8, we consider the use of fixed thresholds. We studied the effects of using different values for t, starting from 0 (no auditing) and increasing it until 0.9 (90% of the stream rate), and present a detailed set of results on applying different thresholds to different scenarios. In each scenario, the ratio of opportunistic nodes is fixed to 30%, but their contribution factor (profile) is varied among 0, 100, 200, and 300 Kbps. All other 70% nodes follow the protocol, with a maximum contribution rate set to 550 Kbps (upload factor = 1.1). We present the average download rates (left) and the number of correct nodes mistakenly removed from the system, termed false positives (right), for each of these configurations. The threshold applied is presented on the x-axis. In the left graph, as the threshold increases, higher download averages are observed, since more opportunistic nodes are detected and punished. However, the number of nodes incorrectly accused also increases with higher thresholds, as observed in the right graph.

Scenarios where opportunistic nodes contribute at higher rates (300 Kbps) are less disruptive to the system, but they also require higher thresholds to be applied. Different thresholds yield best results under different scenarios, but overall, from the results presented in Figure 8, we concluded that the best fixed threshold is $t = 0.6$, providing the best compromise in terms of performance and false positives across all scenarios.

In Figure 9, we compare all three strategies proposed in subsection 4.2 against each other and against a configuration with no auditing, under different scenarios. We set $t = 0.6$ for the fixed threshold strategy and as the initial threshold in the *stepwise adaptive* strategy. We summarize the three strategies in Table 1. We simulated sessions where 30% of the nodes were opportunistic and with varying ratios of contribution. In the x-axis, the contribution rate of opportunistic nodes is varied from 0 to 450 Kbps. All other nodes are correct, contributing at a maximum rate of 550 Kbps. We present both the average and the minimum download factors across all correct nodes in the system. As the contribution rate of opportunistic nodes increases, the download factors are expected to increase, which is clear from the curves presented.

Strategy	**Description**
No auditing	Fixed $t = 0.0$
Fixed threshold	Fixed $t = 0.6$
Stepwise adaptive	Minimum $t = 0.6$. If avg sampled download factor < 0.98, increase t by 0.1. Decrease t back to 0.6 when avg download is satisfactory again.
Percentile-based adaptive	Minimum $t = 0.0$. If avg sampled download factor < 0.98, t is chosen based on sampled upload factors ($t >$ lower 10% sampled values).

Table 1. Strategies used for defining the minimum upload threshold t

Figure 9 shows that all strategies yield significantly better results compared to an approach with no auditing. While both adaptive strategies yield excellent download rates to correct nodes, the fixed threshold strategy's performance is not as good when opportunistic nodes are contributing with 300 or slightly more Kbps (near 0.6

contribution factor). At those rates opportunistic nodes are harmful to the system, yet the fixed threshold of 0.6 is not able to detect them.

Finally, in Figure 10, we consider a scenario where opportunistic nodes contribute with different rates. We varied the percentage of opportunistic nodes in the system from 0 to 90%, and evenly assigned them different contribution rates. The graphs present the average and minimum download rates for these scenarios. Once again, no auditing performs significantly worse than any of the proposed strategies. Here, the stepwise adaptive approach yields the best results when large percentages of opportunistic nodes are present in the system. It is also simpler than the percentile-based approach, since it is based only on samples of the download rates of nodes. In both sets of experiments, the number of false positives was practically null under all three strategies considered (at most one in some cases).

6. DISCUSSION

6.1 Auditing Costs

The overheads imposed by auditing are an important consideration, which we address in this subsection. Most of the work of auditing is performed by local auditors, which are executed on the user nodes. The overhead is constant, independent of the size of the system, and is not significant, since nodes only exchange a small amount of accounting data at pre-defined intervals of time (for example, 10 seconds). If we consider a packet rate of 50 packets/s, in 10 seconds the maximum number of packets received and sent by each node is 1000. For each packet sent or received, the history needs to indicate which neighbor sent or received the packet. By using 4 bits to identify each neighbor, the history's size adds up to 4000 bits, or 500 bytes. This is not significant compared to the amount of regular data exchanged in a streaming session.

We also analyzed the costs of the global auditors. Since they are dedicated and external to the system, the overhead imposed by them is of higher concern. Global auditors' main tasks consist of sampling the system to collect download and upload rates of nodes, and of occasionally disseminating updates to the threshold value, through gossip. The sample size remains fixed independent of the size of the population. We ran simulations to estimate the worst-case standard deviation of the download rates across all nodes. Accordingly, we estimate that a sample size of 300 nodes is sufficient to provide 95% confidence, independent of the population size. For smaller systems, such as the ones simulated in this work, even a smaller number of samples was found to be sufficient to yield satisfactory results. Therefore, centralized costs are fixed, and provide a clear advantage for using auditing against tit-for-tat approaches in large-scale systems.

6.2 Heterogenous Systems

So far we considered the use of auditing to enforce node contribution in systems where all nodes are assumed to have homogeneous bandwidth resources, enough to upload and download at a rate close to the stream rate. Pull-based streaming may be extended to heterogenous systems by organizing nodes into multiple groups, according

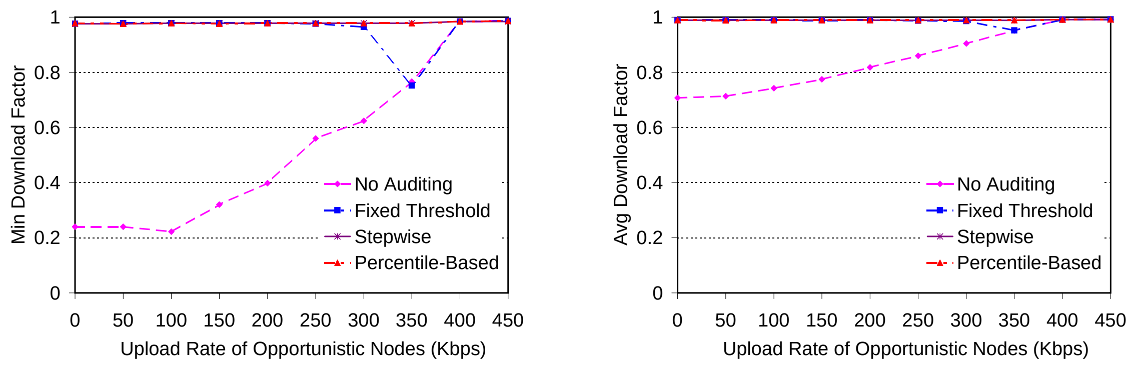


Figure 9. Minimum and average download factors across all correct nodes when using different strategies for choosing the threshold. The upload contribution rate of opportunistic nodes is varied in the x-axis, and the number of opportunistic nodes is fixed at 30%.

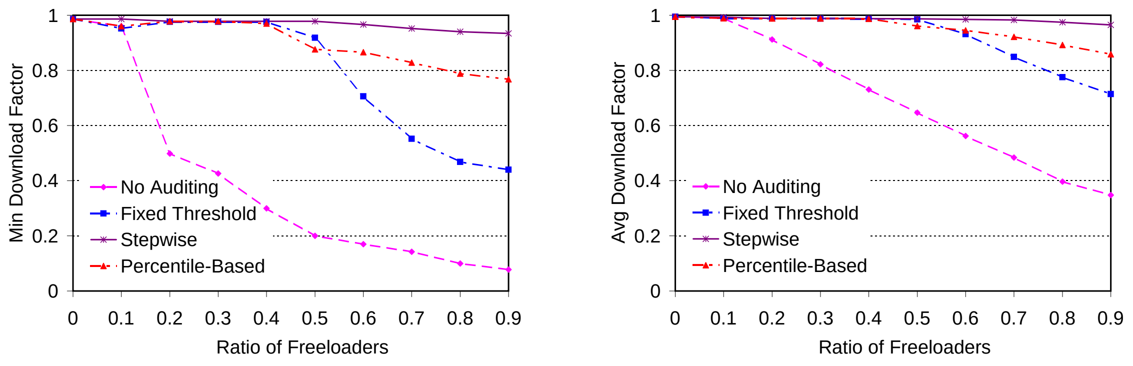


Figure 10. Minimum and average download factors across all correct nodes when using different strategies for choosing the threshold. Each session has mixed set of opportunistic nodes (contributing at different rates) and percentage of opportunistic nodes is varied on the x-axis.

to their upload bandwidths: nodes able to upload at a rate higher than the stream rate are placed in higher-lever groups, which are closer to the source. The source sends data to the highest level group only, who uses the basic protocol to disseminate data among each other. Nodes in lower levels may receive data at smaller rates, after some filtering is applied, and higher-level nodes may be used to act as sources to the lower-level nodes, alleviating the burden at the source.

Auditing can be used to avoid the presence of opportunistic and lower bandwidth nodes in the higher-level groups. It can ensure that the hierarchy of nodes is obeyed by all nodes, while allowing the system to leverage additional resources from privileged altruistic nodes to forward data to lower level groups. We intend to explore this further in future work.

7. RELATED WORK

Several P2P live-streaming protocols have been previously proposed. The first generation of systems (Overcast,[2] Narada[1]) relied on approaches based on pushing data through a single dissemination tree. Later approaches focused on improving fairness among peers and resilience to churn by breaking data into multiple substreams and sending them along disjoing paths (SplitStream,[3] Bullet[10]).

More recent systems like CoolStreaming[5] and Chainsaw[4] use a pull-based style of data dissemination. Coolstreaming breaks the data into packets, and peers organized into a mesh request packets from their neighbors using a scheduling algorithm. As we saw earlier, Chainsaw uses a simpler policy for requesting packets, randomly fetching them while respecting a maximum limit on the number of outstanding requests to each neighbor. Chainsaw presents smaller delays for the receipt of packets compared to the Coolstreaming protocol. In a more recent work,[11] mesh-based approaches are shown to present better performance over tree-based approaches.

Previous papers have considered a variety of possible mechanisms to encourage node contribution. Oversight[12] is a framework proposed to enforce download rate limitations on P2P media streaming systems. The protocol relies on a set of trusted nodes that store information on the data downloaded by each node receiving data. Nodes only send an object after consulting the trusted nodes to verify if the nodes requesting the stream are not overrequesting data. It is targeted to systems where nodes upload full media objects from each other, and not for live-streaming systems where all nodes are interested in receiving the exact same data in close to real time.

Ngan et al.[13] consider fairness issues in the context of tree-based peer-to-peer streaming protocols. The authors present mechanisms that rank peers according to their level of cooperation with the system. One of their techniques involves the reconstruction of trees as a way of punishing opportunistic nodes. Most of their mechanisms require peers to keep track of their parents' and children's behavior.

Pai et al. studied the effect of different types of incentives on the Chainsaw protocol.[14] After exploring tit-for-tat and some variations, the authors propose an algorithm that sets up local markets at every node, where neighbors compete for the node's upload capacity. Nodes favor neighbors who contribute more. Experiments were limited, with nodes classified as fast or slow nodes. The results indicate that the proposed algorithm improves the performance of the system when the total upload capacity is not enough to supply all the nodes. Pulse[15] is another live-streaming system where nodes choose their neighbors based on their history of interaction. Nodes are placed in the system according to their current trading performances, encouraging nodes to contribute more and therefore be closer to the source.

BAR Gossip[9] is a more recent live-streaming approach that tolerates the existence of opportunistic and malicious nodes. Time is divided into rounds, in which each peer communicates with another peer selected using a pseudo-random function. In each round, peers exchange their current history containing the identifiers of all the current data they hold, as basis for the next exchanges. Nodes also perform a phase of *optimistic push*, forwarding useful updates to pseudo-randomly picked peers with no guarantee of useful return.

8. CONCLUSION

We propose and evaluate a scalable auditing-based technique for enforcing fairness in a live-streaming system. Our approach employs local auditors that execute on all nodes in a streaming session. They are responsible for collecting auditable information about other neighbors' data exchanges, and for verifying that neighbors upload more data than a specified threshold. This threshold is defined by dedicated global auditors, which periodically sample the state of the system to determine if the overall download rate is compromised by the presence of opportunistic nodes. Global auditing determines the minimum threshold for uploads, and works with local auditing to punish nodes that do not upload enough data. We study the efficiency of our auditing approach through simulation, and show that it is able to maintain the throughput of the streaming system even in the presence of a large number of opportunistic nodes.

REFERENCES

1. Y.-H. Chu, S. G. Rao, and H. Zhang, "A Case for End System Multicast," in *Proc. of ACM Sigmetrics*, (Santa Clara, CA), 2000.
2. J. Jannotti, D. K. Gifford, K. L. Johnson, M. F. Kaashoek, and J. O. Jr., "Overcast: Reliable Multicasting with an Overlay Network," in *Proc. of the 4th Symposium on Operating Systems Design and Implementation (OSDI)*, (San Diego, CA), 2000.
3. M. Castro, P. Druschel, A.-M. Kermarrec, A. Nandi, A. Rowstron, and A. Singh, "SplitStream: High-bandwidth Content Distribution in Cooperative Environments," in *Proc. of the 19th ACM Symposium on Operating Systems Principles (SOSP)*, (Bolton Landing, NY), October 2003.
4. V. Pai, K. Kumar, K. Kamilmani, V. Sambamurthy, and A. E. Mohr, "Chainsaw: Eliminating Trees from Overlay Multicast," in *4th International Workshop on Peer-to-Peer Systems (IPTPS)*, (Ithaca, NY), February 2005.
5. X. Zhang, J. Liu, B. Li, and T.-S. P. Yum, "CoolStreaming/DONet: A Data-Driven Overlay Network for Efficient Live Media Streaming," in *Proc. of the 24th Conference on Computer Communications and Networking (INFOCOM)*, (Miami, FL), 2005.
6. M. Haridasan and R. van Renesse, "Defense Against Intrusion in a Live Streaming Multicast System," in *Proc. of the 6th IEEE International Conference on Peer-to-Peer Computing (P2P)*, (Cambridge, UK), September 2006.
7. N. Magharei and R. Rejaie, "PRIME: Peer-to-Peer Receiver-drIven MEsh-based Streaming," in *Proc. of the 26th Conference on Computer Communications (INFOCOM)*, (Anchorage, Alaska), April 2007.
8. "PPLive Homepage. Available: http://www.pplive.com."
9. H. C. Li, A. Clement, E. L. Wong, J. Napper, I. Roy, L. Alvisi, and M. Dahlin, "BAR Gossip," in *Proc. of the 7th Symposium on Operating Systems Design and Implementation (OSDI)*, (Seattle, WA), 2006.
10. D. Kostić, A. Rodriguez, J. Albrecht, and A. Vahdat, "Bullet: High Bandwidth Data Dissemination Using an Overlay Mesh," in *Proc. of the 19th ACM Symposium on Operating Systems Principles (SOSP)*, (Bolton Landing, NY), 2003.
11. N. Magharei, R. Rejaie, and Y. Guo, "Mesh or Multiple-Tree: A Comparative Study of Live P2P Streaming Approaches," in *Proc. of the 26th Conference on Computer Communications (INFOCOM)*, (Anchorage, Alaska), April 2007.

12. W. Conner, K. Nahrstedt, and I. Gupta, “Preventing DoS Attacks in Peer-to-Peer Media Streaming Systems,” in *Proc. of the 13th Annual Multimedia Computing and Networking Conference (MMCN)*, (San Jose, CA), 2006.
13. T.-W. Ngan, D. S. Wallach, and P. Druschel, “Incentives-Compatible Peer-to-Peer Multicast,” in *2nd Workshop on the Economics of Peer-to-Peer Systems*, (Cambridge, Massachussetts), June 2004.
14. V. Pai and A. E. Mohr, “Improving Robustness of Peer-to-Peer Streaming with Incentives,” in *Proc. of the 1st Workshop on the Economics of Networked Systems (NetEcon)*, (Ann Arbor, MI), 2006.
15. F. Pianese, J. Keller, and E. W. Biersack, “PULSE, a Flexible P2P Live Streaming System,” in *Proc. of the Ninth IEEE Global Internet Workshop*, (Barcelona, Spain), 2006.