AFRL-IF-RS-TR-2003-108
Final Technical Report
May 2003

YALLCAST: A NEW PARADIGM FOR CONTENT DISTRIBUTION

USC Information Sciences Institute

Sponsored by
Defense Advanced Research Projects Agency
DARPA Order No. J956

The views and conclusions contained in this document are those of the authors and should not be interpreted as necessarily representing the official policies, either expressed or implied, of the Defense Advanced Research Projects Agency or the U.S. Government.

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

This report has been reviewed by the Air Force Research Laboratory, Information Directorate, Public Affairs Office (IFOIPA) and is releasable to the National Technical Information Service (NTIS). At NTIS it will be releasable to the general public, including foreign nations.

AFRL-IF-RS-TR-2003-108 has been reviewed and is approved for publication.

APPROVED:

SIAMAK TABRIZI
Project Engineer

FOR THE DIRECTOR:

WARREN H. DEBANY, Jr.
Technical Advisor, Information Grid Division
Information Directorate

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 074-0188

1. AGENCY USE ONLY (Leave blank)	**2. REPORT DATE** May 2003	**3. REPORT TYPE AND DATES COVERED** Final Jun 00 – Dec 02

4. TITLE AND SUBTITLE

YALLCAST: A NEW PARADIGM FOR CONTENT DISTRIBUTION

5. FUNDING NUMBERS

C - F30602-00-2-0559
PE - 62301E
PR - J956
TA - 21
WU - C1

6. AUTHOR(S)

Robert Lindell

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

USC Information Sciences Institute
4676 Admiralty Way
Marina Del Rey, CA 90292-6695

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES)

Defense Advanced Research Projects Agency
3701 North Fairfax Drive
Arlington, VA 22203-1714

AFRL/IFGC
525 Brooks Rd
Rome NY 13441-4505

10. SPONSORING / MONITORING AGENCY REPORT NUMBER

AFRL-IF-RS-TR-2003-108

11. SUPPLEMENTARY NOTES

AFRL Project Engineer: Siamak Tabrizi, IFGC, 315-330-4823, tabrizis@rl.af.mil

12a. DISTRIBUTION / AVAILABILITY STATEMENT Approved for public release; distribution unlimited.	**12b. DISTRIBUTION CODE**

13. ABSTRACT ***(Maximum 200 Words)***

Disappointing delays in deploying network-layer multicast have triggered recent interest in application-level multicast. Application-level multicast is an attractive and deployable alternative, because it requires no support from the underlying network. Instead, application software automatically creates an overlay distribution tree spanning all the participants of a multicast group. The efficacy of the overlay tree is essential for the usability and performance of an application-level multicast architecture. The Yallcast project has designed and implemented algorithms for overlay tree construction and maintenance. The algorithms are designed for rapid tree repair, and for continual refinement of the distribution tree over time, based on observed loss or latency performance. These algorithms have been evaluated through simulations and experiments within the framework of an application-level multicast architecture called Yoid.

14. SUBJECT TERMS Application-level multicast, shared tree, topology adaptation	**15. NUMBER OF PAGES** 20 **16. PRICE CODE**

17. SECURITY CLASSIFICATION OF REPORT	**18. SECURITY CLASSIFICATION OF THIS PAGE**	**19. SECURITY CLASSIFICATION OF ABSTRACT**	**20. LIMITATION OF ABSTRACT**
UNCLASSIFIED	UNCLASSIFIED	UNCLASSIFIED	UL

TABLE OF CONTENTS

Executive Summary

Disappointing delays in deploying network-layer multicast have triggered recent interest in application-level multicast. Application-level multicast is an attractive and deployable alternative, because it requires no support from the underlying network. Instead, application software automatically creates an overlay distribution tree spanning all the participants of a multicast group. The efficacy of the overlay tree is essential for the usability and performance of an application-level multicast architecture.

The Yallcast project has designed and implemented algorithms for overlay tree construction and maintenance. The algorithms are designed for rapid tree repair, and for continual refinement of the distribution tree over time, based on observed loss or latency performance. These algorithms have been evaluated through simulations and experiments within the framework of an application-level multicast architecture called Yoid.

Yoid decided to implement a novel tree-first approach. Experience over the course of the project has shown that this technique is well suited for application layer multicast. Trees are built quickly and new members can join in a minimal amount of time. To support this approach, new techniques were developed to address fast loop detection and repair. These algorithms have performed well in both simulations and real world experimentation and usage using a variety of applications. To demonstrate the utility of Yoid, a number of multiparty audio, video, and whiteboard conferencing tools were ported to use the Yoid multicast substrate. Scripts to execute these applications over Yoid are provided with the Yoid software distributions, which are available for Windows, FreeBSD, Linux, and Solaris.

YALLCAST

A New Paradigm for Content Distribution

Robert Lindell, Project Leader

Ramesh Govindan, Paul Francis, Cengiz Alaettinoglu, Yuri Pryadkin

Graduate Students: Pavlin Radoslavov

ISI Computer Networks Division

1 Introduction

Motivated by the difficulties encountered in deploying network-layer multicast on a large operational Internet, several research groups have proposed *application-level multicast*. Application-level multicast requires no support from the underlying network. Rather, application software automatically creates an overlay distribution tree spanning all the participants of a multicast group and forwards application data over this topology. The application software also adapts the distribution tree to participant dynamics (joins and leaves) and network failures. Clearly, such a system can alleviate the deployment woes of network-layer multicast, and bring interesting collaborative applications to widespread use in the Internet.

More generally, this kind of distribution applies not just to traditional multicast applications (like shared whiteboards, and audio and video conferencing), but also to most forms of content distribution. For example, one can imagine a content distribution network's replication infrastructure to be built upon such an overlay. It might also be feasible to implement peer-to-peer file sharing systems on top of such overlays.

The Yallcast project has researched, developed, and released an implementation of application-level multicast that includes novel techniques for tree construction and adaptation. This system has been named *Your Own Internet Distribution* (YOID).

2 The Yoid Architecture

The Yoid application-level multicast architecture allows a collection of Internet hosts to self-organize and maintains a distribution tree topology for content dissemination. This functionality is implemented at the application layer and requires no network layer multicast support to operate, although it can take advantage of network multicast to improve the efficiency and performance of data delivery. The Yoid architecture can, in principle, be implemented either as an application shared service on each host or as a reusable library of common functionality included directly into the code base of particular applications.

Similar to IP multicast, a rendezvous mechanism is central to Yoid. Participants in a Yoid group rendezvous through a shared logical label for the group. In IP multicast, the logical label is a globally unique IP address. In Yoid, however, the logical label for a group syntactically resembles a URL and encodes the name of a rendezvous host, a port number that the rendezvous host is listening on, and the name of the group that is unique to the rendezvous host. Thus, a Yoid label of the form *yoid://foo.bar.org/partners:5555* denotes a Yoid group named *partners*, whose rendezvous host name is *foo.bar.org* and that host is listening on port 5555. Because group names need not be globally unique, Yoid does not require global coordination of group label assignment.

When a participating host (henceforth, a member) wishes to join a Yoid group, it contacts the rendezvous host (or simply, rendezvous). From the rendezvous, it obtains its own node ID that is unique within that group, as well as a list of some (not necessarily all) current members. We call this list the candidate-parents list. The host then uses this list to graft itself, in a manner described later, to the tree topology that is used to distribute application content. The rendezvous does not participate in application content forwarding but it is responsible for keeping it's list of members current, by explicitly checking for their liveliness. It can also plays a key role in other functions, such as partition healing and group security.

Using information provided by the rendezvous, Yoid hosts conspire to construct a shared-tree overlay. Each node on this shared tree is an end host on which a group member resides, and each link is a tunnel between two members. Such a shared-tree overlay construction is not the only way to achieve application-layer multicast. Another approach, which may be called mesh-first, has new members establish mesh links to several current members. Over this mesh, all members run a link-state or distance-vector routing protocol. Using this, members can conspire to construct distribution trees rooted at each source of data (i.e. source-specific trees).

The design of a centralized rendezvous host is an obvious impediment to scaling and robustness. The scaling drawbacks can be alleviated by carefully minimizing interactions between hosts and the rendezvous. A host normally communicates with the rendezvous only when joining or leaving the tree. When a host joins the tree, the rendezvous point provides a known identity that can be queried to learn the identity of other participants in the group. This is the bootstrapping operation that is needed to allow new members to discover the other members of the group. During the lifetime of the tree, the rendezvous is additionally responsible for electing and maintaining the identity of the root of the distribution tree. An election algorithm, combined with a keep-alive protocol, serves to maintain the identity of a root node. Additionally, a keep-alive protocol is used to maintain a fresh list of potential parent nodes of the tree for newly joining members to contact. For scaling reasons, this list of candidate parents is only a subset of the members of the group.

Since interaction with the rendezvous is limited, rendezvous crashes and restarts have a minimal effect on the operation of an existing tree. Data distribution on a tree continues undisturbed when a rendezvous crashes; however new nodes may not be able to join an existing tree. Nevertheless, for truly robust tree operation, a solution that allows multiple rendezvous nodes, with some transparent failover capability, is necessary.

3 The Yoid Protocol Stack

Paul Francis of ACIRI created an early prototype of Yoid. Yoid implements the protocol design shown in Figure 1. Each of these protocol layers is described below.

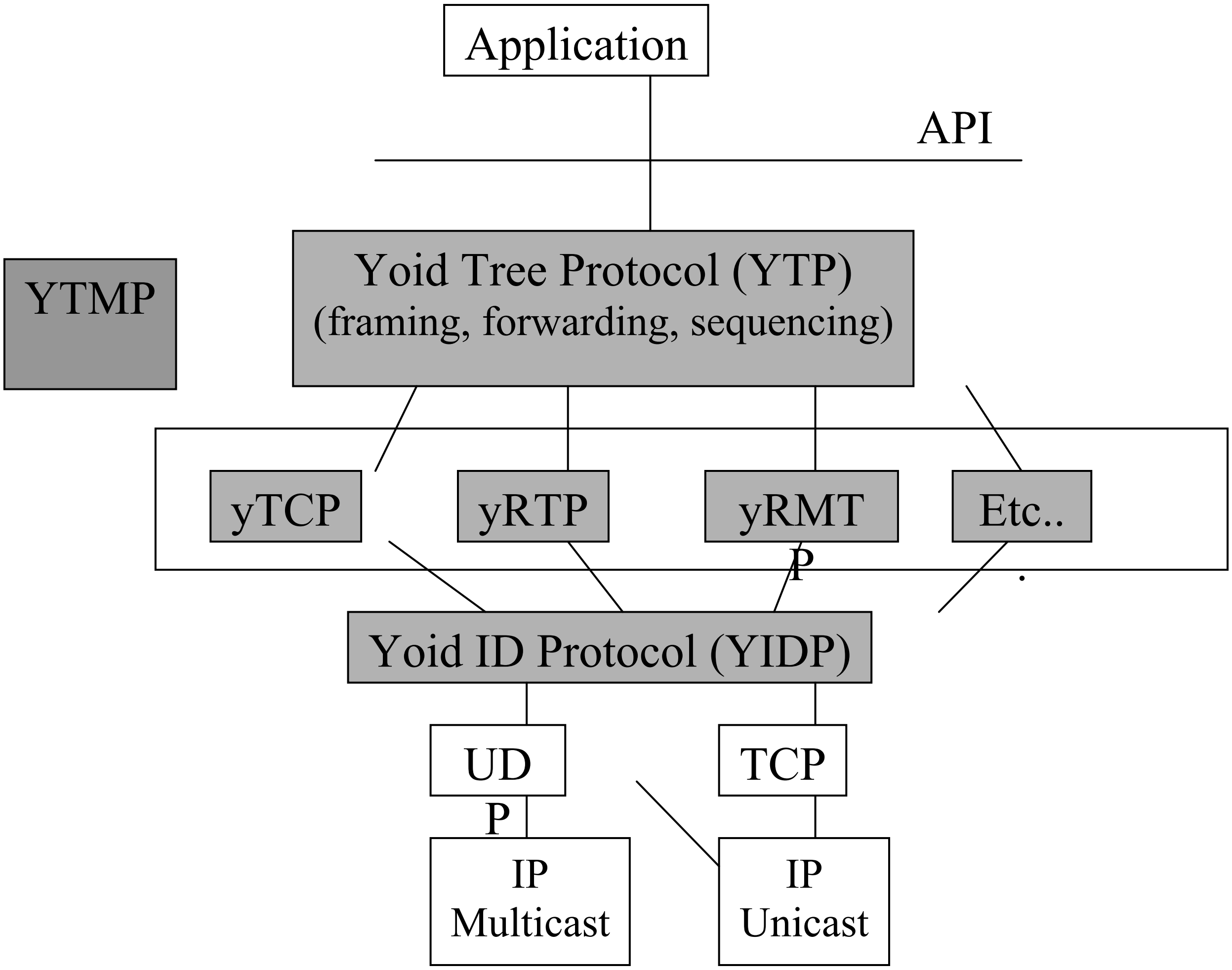

Figure 1 - The Yoid Protocol Stack

3.1 Yoid Identification Protocol (YIDP)

The Yoid Identification Protocol (YIDP) serves two primary purposes. First, it identifies the yoid group of a given frame. Second, it identifies the immediate source and destination group members of a given frame. By immediate we mean the member that last transmitted the frame and the member that will next receive the frame (as opposed to the original source and ultimate destination). These members are also referred to as the hop-by-hop source and destination members.

YIDP makes some attempt at being able to identify members without the use of IP addresses. The reason is that IP addresses may change over time (for instance a dial-up host that loses a connection and redials), or may undergo translation. The ability to cope with NAT boxes remains problematic because of the general inability to establish arbitrary incoming connections through NAT boxes.

3.2 Yoid Transport Protocol

The Yoid Transport Protocols (YTP) are a number of protocols used between two neighbors or within a cluster to do such things as flow control, congestion control, sequencing, lost packet recovery (retransmissions) and the like. Examples include the equivalents of TCP and RTP (yTCP and yRTP), and multicast equivalents of TCP and RTP (yMTCP and yMRTP). Note that the multicast equivalents can be much simpler than the reliable multicast protocols being developed for the wider Internet today, because of the strict scoping.

It is reasonable to consider running Yoid over TCP rather than UDP, in which case there would not need to be any protocol running at the Yoid transport layer. It is very convenient, however, to be able to run all of Yoid over a single UDP port, primarily for the purpose of dealing with firewalls and other boxes that may filter on port number. For this reason, the preferred stack is yTCP (or other Yoid transport) over YIDP over UDP.

3.3 Yoid Distribution Protocol

The Yoid Distribution Protocol (YDP) handles everything needed to move an application frame end to end over the tree-mesh with near-zero probability of loss. Reflecting the broad functionality of Yoid, YDP is itself rich in functionality. It handles framing, determines the type of forwarding (multicast, broadcast, etc.), has a hop-by-hop pushback flow control mechanism, and has a hop count. YDP can also identify the final destination(s) of the frame and the original source of a frame. In essence, YDP is to a Yoid topology what IP is to a router/host topology.

In addition to all this, YDP has a sequence number space that can sequence and uniquely identify every frame originated by any source. This sequence number is used not only to insure end-to-end reliability and ordering of frame delivery, but also to prevent looping of frames and duplicate delivery of frames to members acting as transits. This allows frames to be transmitted over the tree, the mesh, or a combination of both (for instance, in response to a temporary partition of the tree).

3.4 Yoid Tree Management Protocol

The Yoid Tree Management Protocol (YTMP) is the most complex and interesting research topic of this project. The YTMP layer is responsible for the construction and maintenance of a group distribution tree. The algorithms developed for YTMP will be described in detail in the next section.

4 Tree Management Algorithms

How does the Yoid shared tree actually get built? The first member to join the group is designated by the rendezvous to be the root of the tree. Each subsequent joining member contacts the rendezvous and obtains a list of current members. The joining member then selects one of the current members to be its parent. The choice of parent can be influenced by a number of criteria, including closeness and performance.

For example, a member might choose the topologically closest current member as its parent, if this can be determined (e.g. based on heuristics such as IP address prefix). Clearly, the choice of parent crucially determines overall perceived performance; we will return to this subject later. A member is not responsible for finding children, although it may reject members that request to be its children. When a member loses connectivity to its parent, it attempts to contact other members in order to select a new parent. When the member switches parents, its relationship with its offspring is unaffected. While this tree construction protocol is simple and requires little inter-member coordination, it presents two challenges.

First, this distributed tree construction is susceptible to loop formation. To deal with the possibility of loops caused by our simple, localized tree-grafting algorithm, we do not use loop avoidance techniques. Intuitively, loop avoidance may require a priori knowledge of the overlay topology, or dissemination of routing information. Yoid, however, starts off with a very simple method to graft nodes onto the overlay tree that doesn't require running a routing protocol. In keeping with this design, we use a novel loop detection mechanism, together with a technique for fast loop termination. Yoid augments a path-vector like approach with a novel switch-stamp mechanism to detect and rapidly terminate loops.

Second, the tree construction algorithm described above largely ignores issues of performance. In particular, because Yoid builds overlays using hosts, it can be susceptible to performance pathologies that arise from poor choices of topology.

To take a concrete example, consider a host that is behind a limited capacity broadband connection (such as a DSL line). If this host's fanout in a Yoid tree is N (one parent and N-1 children), it requires a bandwidth of N*R where R is the application content data rate (e.g. an audio stream). If this bandwidth exceeds the capacity of the host's connection to the network, it adversely affects the perceived performance at all hosts whose on-tree path to the source traverses this host.

One obvious way to avoid this problem is to have a static fanout limit at each Yoid host. This approach is undesirable because it may require manually configuring the fanout based on a host's network connection speed, and also because the available bandwidth can vary dynamically. Accordingly, Yoid dynamically refines the tree based on observed data losses. Each host compares its loss fingerprints (the specific Yoid frames lost within a fixed window, called losssprints), with those of its neighbors and, if the losssprints differ significantly, the host decides whether it should switch parents (for example, because its current parent has a high fanout). This kind of topology adaptation is unique to application-level infrastructures. It is also complementary to congestion control mechanisms that adapt the sending rate of the audio streams to the capacity of the tree.

To take another concrete example, consider a host behind a high latency network connection, such as a phone-line modem. If the tree construction algorithm results in this host being near the root of the tree, all hosts downstream of this host with respect to a given source will observe high latency data delivery. For real-time audio and video conferencing applications, clearly, this is a problem. Yoid deals with this kind of performance pathology by dynamic refinement as well. Specifically, Yoid hosts occasionally test new parents to see if they can consistently deliver Yoid frames at significantly lower latency, then switch to these parents.

Although we have described the latency and loss-rate refinement algorithms separately, they are intended to work together to balance reasonable latency performance with loss-rate. Simulation was used to understand how these algorithms work together and refine the techniques.

5 Yoid Simulations

For analysis, the Yoid project developed a packet-level simulator. The inputs to the simulator include the underlying network topology and the placement of the Yoid nodes. Each of the selected Yoid nodes is running the Yoid protocol stack. The topology routing engine distributes the packets throughout the tree. The engine forwards packets hop-by-hop, and at each hop it considers the latency and the bandwidth of the topology link to propagate or queue the packets. There is also a simple queue mechanism (FIFO) that tail-drops the packets if the queue is full. In all simulations, the queue size is 1000 packets.

In these simulations, an attempt was made to use a real-world router-level topology. The topology information was collected using a large number of traceroute requests sent over the Internet. The resulting topology had 102639 nodes and 142303 links. Nodes were recursively removed that had a fanout of one to obtain a topology resembling an Internet core of size 27646 nodes and 67310 links. The motivation for truncating the original topology was to remove the long, skinny branches that do not represent well the network connectivity at the edges, but are an artifact from the particular methodology used to obtain the topology information. Finally, after selecting the location of Yoid nodes in the topology, an extra leaf node was added to each location, and was used to simulate the Yoid members. This approach helped to simulate bottleneck edge links.

In all simulations a constant-rate 10kbps flow of packet size of 50 octets was chosen as an approximate model of the audio traffic generated by conferencing application Rat with low-end GSM encoding. In most cases the simulation used 200 receivers placed at random.

These simulations considered two metrics. The average end-to-end latency was computed as the sum of the pre-configured latency on each link from all sender nodes to all receivers, and then averaged across all senders and receivers. The worst-case data loss-rate represented the largest loss seen at a receiver in a window of 100 packets.

In most simulations, there were no observed loops throughout the entire run. In some cases however, 1 or 2 loops formed per simulation run, and these were quickly resolved within a few hundred milliseconds.

To stress the loss-rate refinement algorithm, the setup was similar to the first simulation, except that the selected bottleneck links have capacity of only 15kbps, but their latency was reduced to 10ms. To stress the latency refinement algorithm, the latency of the same edge links was increased, as in the previous simulation to 80ms, but at the same time the link bandwidth was increased to a default of 1Mbps. In both setups, the results are similar to those in the first simulation, though overall the losses for the loss-rate refinement algorithm simulation are slightly higher during the transient period due to the more restricted bottleneck links.

To test the sensitivity of the latency refinement algorithm to the latency improvement threshold, the same setup as in the first simulation was used, except that the latency improvement threshold was 5ms instead of 50ms. Because 5ms is smaller than the latency of any single link, the expectation is that the average latency will become much closer to the unicast average latency. Surprisingly, the initial drop of the latency soon after the sender was activated was not as sharp as in the other simulations. The reason for this was that with a smaller threshold a node would switch to parents that offer little improvement, therefore overall it took a larger number of iterations to achieve the improvement of the algorithm with a larger threshold. Indeed, the latency gradually continued to improve, but even after 3600 seconds of simulation time, it was similar to the results with 50ms threshold.

On smaller groups, the difference in latency between the Yoid tree and the unicast latency was only 10% and there were significantly fewer losses due to the much smaller number of parent switches.

6 Wide Area Experiments

To perform more extensive tests, the project needed access to a larger number of hosts spread across the Internet. The CAIRN testbed was the choice for our experiments, because it has tens of hosts spread across the United States, and included hosts in Europe as well. However, using GUI based applications such as Wb and Rat for debugging purpose was not very practical for execution on more than 2-3 hosts, and especially if a single person must control them.

To simplify debugging, the project developed from scratch a command-line traffic generator program named *msend* that can be used to send and receive rate-controlled data packets (either by unicast or multicast), and at the same time it can be used to collect and report statistics for the received data. In combination with a simple shell script, *msend* can be used to collect and process long-term statistics for all receivers, and email a periodic summary of the results to the person performing the experiment.

In the first set of tests using *msend* with Yoid over CAIRN, the project investigated how Yoid would perform over a long period of time across a wide area network. Yoid was started on 8 CAIRN hosts spread across US and Europe. On each of those hosts the *msend* application was started in a dual sender/receiver mode and with fixed transmission rate. After running the experiment for more than 24 hours, all hosts were still operating normally, and the loss rate was negligible.

In the second set of tests, the project evaluated how membership dynamics may have an impact on robustness. In particular, *msend* was started on the same 8 hosts as in the first experiment. Instances of the Yoid daemon were gracefully exited and restarted on some of the members, or in some cases, forcefully killed on particular Yoid hosts. Only the Yoid daemon was started or stopped, while *msend* was running all of the time. Early experiments encountered some implementation problems that resulted in crashes. After those problems were fixed, the rest of the experiment went much smoother. In most cases, after a member rejoins the group, it would start quickly receiving again the traffic from all other members. Even if the Rendezvous Point (RP) were killed, the rest of the members would continue to receive the data traffic, although no new members could join. However, unlike other members, if the RP rejoined, it had to wait of the order of one minute before receiving again the traffic. The reason for this longer interval was that now the RP would have to wait until some of the members already on the tree try to establish connection again with the RP.

7 Mbone Application Integration

Having the Yoid substrate was not sufficient for the purposes of this project. It was important to have applications built on top of the Yoid substrate that the project members and others could use. Integration of these applications helped to drive the research contributions of the Yoid project.

The first step was to integrate a very popular multicast shared white board application, Wb. This application uses IP multicast to distribute updates to the white board to all participants. The key challenge was to port this to Yoid *without any application level changes*. If possible, this would make it easier to distribute the application and let the application software evolve independently without requiring maintenance from the Yoid team. Furthermore, in the case of Wb, this was essential since sources for the application are no longer available. To accomplish this task, the Yoid project was required to develop a system call interception mechanism. This mechanism is described in a subsequent section of this report.

The next set of tools that was integrated was the Mbone audio teleconferencing applications Vat and Rat. These applications operated over Yoid without any changes or system call interception. After getting shared whiteboard and audio to work, the project added support to run the Mbone video conferencing tool Vic. The Yoid project released an initial version of Yoid with this complete complement of applications for conferencing. The release supported most popular platforms including Windows, Linux, FreeBSD, and Solaris. A substantial effort was made in the project to address portability and offer the Yoid functionality on the most popular platforms. It is the belief of this project that multi-platform support is essential for wide scale adoption of these types of technologies.

To simplify usage, the Yoid project created a number of helper scripts for both the Unix and Windows environment so that the *Yoidized* applications could be invoked with a single command with the group Yoid URL supplied as a command line argument.

To test the Yoid substrate, and integration with applications, the Yoid group regularly used the multicast conferencing tools to hold group meetings. The group tested both distribution trees constructed on a single LAN and other configurations that include members on cable modem connections or at other institutions connected over the WAN.

8 H.323 Conferencing Support using Yoid

H.323 is the ITU architecture for IP based audiovisual services and is supported by popular applications such as Microsoft's NetMeeting. The H.323 specification provides for multiparty conferencing through two mechanisms: IP multicast and server based solutions using a multipoint control unit (MCU). The use of an MCU is the predominant deployment model of multi-party conferencing using ITU H.323 based products, since most H.323 applications, such as NetMeeting, do not support IP multicast. The typical configuration is a hub and spoke client-server topology with many H.323 clients connected to a single Multipoint Control Unit (MCU). Multimedia network traffic is unicasted betweens clients and the MCU. The MCU provides all audio and video mixing capabilities for the conference.

The Yallcast project decided it would be useful and interesting to offer the ability to perform multi-party H.323 without using servers and without requiring native multicast support from either the network or the client application. This would allow individuals to conference over the Internet using applications such as NetMeeting, but with the ability to conference with more than two individuals at a time.

To incorporate Yoid technology into the H.323 architecture, a *Yoidized* MCU implementation was designed and developed to run locally at each H.323 client host. This MCU would interface directly with existing, unmodified, H.323 clients and distribute the multimedia data traffic over a Yoid tree rather than centrally from a single MCU server. For this implementation, the local Yoid daemon that runs on client hosts would appear to be an H.323 compliant MCU, but internally, operates quite differently. Each of the Yoid based MCUs join a Yoid group and multicast their audio and video data to all participants of the group. Each MCU then mixing the resulting audio signals and transmits this data locally to the H.323 client. This integration between H.323 clients and Yoid is shown in Figure 2.

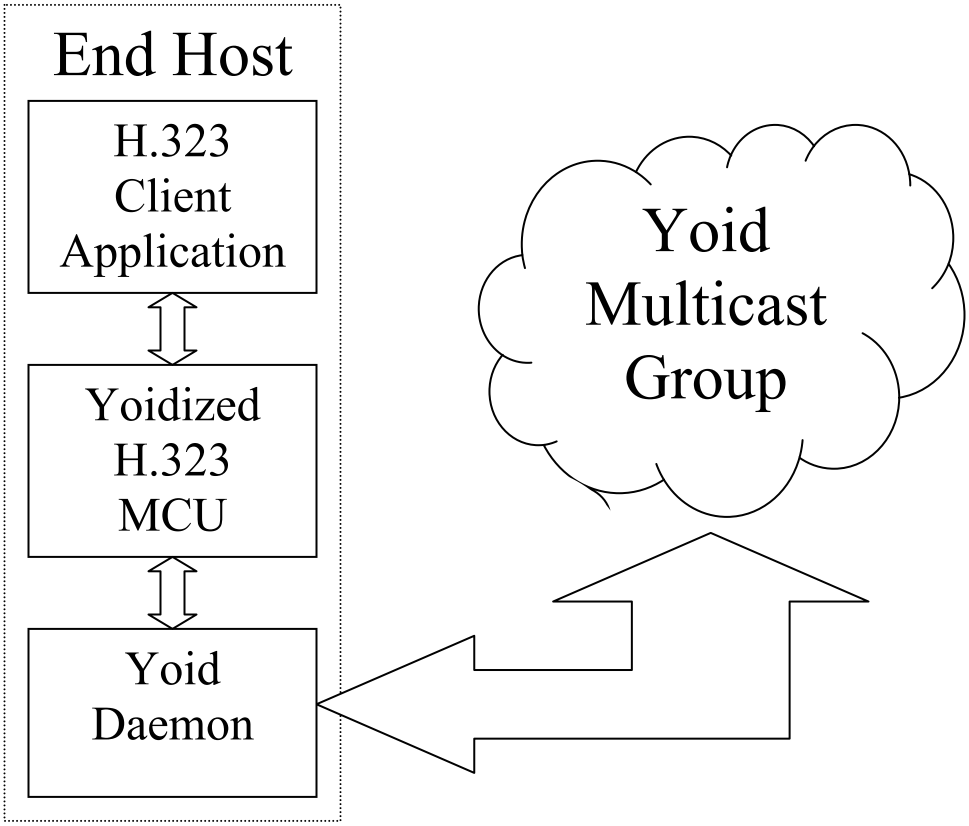

Figure 2 - Yoid and H.323 Host Integration

Rather than starting from scratch, the project modified an existing open source implementation of the H.323 protocol stack using OpenH323 (http://www.openh323.org). This modified MCU was tested with existing applications such as NetMeeting and the open source applications GnomeMeeting and Ohphone. It is possible to hold multiparty conferences using these H.323 clients on both Microsoft Windows and Linux with no interoperability issues.

Normally, an H.323 client and MCU cannot run on the same host, at the same time. This is because both applications expect to accept incoming call requests on a well known TCP port number and both applications cannot be bound to the same port simultaneously. We enhanced our system call interception functionality to address this issue so that we would not need to modify existing H.323 client applications. Such modification would be impossible with NetMeeting since this project does not have access to the source code of the application. Details of the Windows system call interception are presented separately below.

The *Yoidized* H.323 multipoint control unit (MCU) currently supports both audio and video traffic. In the future, it would be useful to incorporate support for the ITU shared whiteboard protocol.

9 System Call Interception

Certain applications require simple system call interception support to use them unmodified under Yoid. The basic problem is that some applications expect to only communicate with remote applications and are unable to communicate locally on the same host with the Yoid daemon. An example is the shared whiteboard "wb" under Unix in which only an execution binary is available. The Yallcast project did not have access to the source code to make any modifications. In order for "wb" to communicate locally with the Yoid daemon, it is necessary to intercept certain system calls related to UDP socket initialization.

One possibility for integrating Wb into Yoid would have been to use a pre-loaded YOID dynamic library, but unfortunately this technique was applicable for Wb given that Wb is statically compiled. The projected decided to use system call interception (a standard technique employed by many debuggers) for intercepting certain system calls made by Wb and re-directing packets to a Yoid daemon running on the same machine. For each intercepted system call, the Yoid library modifies some of its parameters such as network address or port. By carefully modifying the local and the destination addresses of the sent or received packets, the library can redirect (on the fly) all packets through the Yoid process. Then, the Yoid infrastructure delivers them to the other participants using the multicast tree overlay. All of this translation is completely transparent to the application, and therefore it allows the integration of any traditional multicast application without application modification. The project has prototyped such a library, and using it, was able to integrate YWb - a shared white board application. Wb can now be used without requiring any network multicast infrastructure. Initially, integration of YWb encompassed 3 processes: (1) a normal Wb process, (2) a process monitoring and intercepting system calls made by Wb, and (3) a Yoid daemon that does encapsulation/decapsulation as well as tree management and rendezvous. Eventually, the project was able to combine (2) and (3) into a single universal helper program, called Yudp, that is described later in this report.

9.1 Unix System Call Interception based on Strace

The Yoid library initially had a simple form of system call interception code that was developed in-house and worked properly on our development platforms. After more experience with Yoid on other platforms and especially older versions of operating systems, some users found that our system call intercept mechanisms would generate intermittent errors.

Instead of attempting to repair this simple system call interception code, it was decided to replace it with a code from a very popular and portable application known as strace. Strace is a system call interception diagnostic application that is widely used on numerous operating systems. The Yallcast project took the code base to strace and made simple changes to repackage it in the form of a library. This library, which is named stracelib, is now used to perform all system call interception support in Yoid for our Unix ports. A separate mechanism, not based on strace, is still required for the Microsoft Windows platform and is described below.

9.2 Microsoft Windows System Call Interception

System call interception was necessary to enable inter-operation of Microsoft NetMeeting and a *Yoidized* version of OpenMCU, running on the same machine. In accordance with the H323 specifications, both applications at startup begin listening on the same port (H.323 HostCall) and therefore cannot run on the same machine without modifications, unless system call interception is used.

There are a number of system call interception techniques (also known as API spying) under Windows-based operating systems. They include:

- Proxy DLL: a custom DLL that provides the same API as the standard DLL with the same name and forwards calls to the standard DLL after appropriate modifications of arguments are made. This technique is applicable only if the system calls that must be intercepted are exported by a standard system DLL, as is the case with WINSOCK.DLL that exports socket calls that Yoid needs to intercept.

- IAT patching: patching the Import Address Table for the executable. This method is more involved, as it requires modifications of the binary image of the loaded executable in memory. Its advantages are speed and ability to intercept only particular system calls without imposing performance penalty on system calls that are not intercepted.

- Winsock hooking using Layered Service Provider: a winsock-specific method of system call interception that is capable of intercepting all socket calls system-wide. This technique is often used by anti-virus software.

Yoid did not require system-wide socket call interception, so it narrowed the options to the first two techniques, and while IAT patching is a good way to intercept system calls selectively, it is too complex in implementation detail. Thus, it was decided to implement a WINSOCK.DLL proxy library. To make use of it, NetMeeting must be started from the same working directory where our proxy library is located. Then, it intercepts all *bind* system calls, for example when NetMeeting tries to bind locally to the HostCall port. It then modifies the port number so there will not be a conflict between NetMeeting and the OpenMCU process. This mechanism allows the unmodified NetMeeting application to locally call the OpenMCU over the loopback interface of the host.

10 Network Management

When users are running Yoid, it is important that tools exist to view what is happening in the constructed overlay when things go wrong. The Yoid project has developed capabilities within the Yoid implementation to aid in the diagnosis and monitoring of the health of the overlay network.

Yoid contains a network management interface adapted from the Zebra project. This interface supports access lists that selectively allow other hosts to log in and monitor the internal state of the Yoid daemon. This information can provide a wealth of information about the state of the overlay and the observed losses between neighbors.

The project has also developed Perl scripts that can walk the nodes of a Yoid tree and use the network management interface to collect statistics from each host. This can be used to construct a pictorial representation of the current topology of the overlay and provide information about packets losses and other relevant information.

Recently, support has been added to Yoid to multicast parent-child relationship information at a low rate to all group members. This information allows each host to reconstruct a view of the entire topology by correlating the information collected by observing the reports over a period of time.

11 YUDP – The Universal Yoid Interface Application

The project analyzed the interfaces between various applications and the Yoid daemon and attempted to design a single universal application that could be used to host the entire range of needs. Before the creation of *yudp*, the project had a collection of applications that were similar but had custom functionality to interface with each particular application.

yudp now serves as the universal application to interface with the full range of ported Yoid applications. It supports all audio, video, shared whiteboard, and an IP multicast gateway. It is also used as a daemon to support the DARPA FTN Cossack project.

yudp has range of options to control it's customization. It can communicate with the applications using unicast or multicast. It will allocate a block of unused ports on even or odd boundaries. It can perform system call interception, and gateway traffic to an IP multicast group.

In the process of creating this universal application, a number of rough edges in the underlying Yoid API were uncovered. An attempt has been made to improve this API based on the continuing experience of porting new applications. For example, the API has simplified how an application can associate UDP sockets with individual Yoid group demultiplexing channels. A single Yoid group now supports a large set of independent demultiplexing channels. This feature simplifies carrying multiple separate data flows for a given application.

As mentioned above, this project has developed and tested a gateway which can connect a Yoid multicast group with a native IP multicast group. This is extremely useful for allowing users who have native IP multicast support to interoperate their conferencing applications with users who must use application layer multicast. We tested this functionality during our group meetings by having part of our group conferencing over Yoid directly and others using IP multicast with a gateway connecting the two groups.

The gateway was originally developed as a separate application but was eventually folded into our universal application, *yudp*. It can now be activated using a command line option to *yudp*. This incorporation in *yudp* allows a single Yoid daemon to serve local applications and gateway to an IP multicast group at the same time.

12 Major Accomplishments

The following list summaries the major accomplishment of the Yallcast Project

- Development of novel tree management algorithms for application-level multicast that emphasize a tree-first approach
- Extensive simulation and testing of the correctness of the Yoid tree management algorithms
- Porting the full suite of Mbone teleconferencing applications to Yoid
- Developing support for multi-party H.323 conferencing over Yoid
- Implementation and public release of the *Yoid II* protocol
- Porting of Yoid to Windows, FreeBSD, Linux, Mac OS X, and Solaris

13 Publications and Presentations

Lindell, R., *Your Own Distribution Protocol (YOID)*, CENIC 2001, San Diego, CA

Lindell, R., *Your Own Distribution Protocol (YOID)*, Internet 2 Peer-to-Peer Workshop, Tempe AZ, January 2002

All publications are listed on the project website at:
http://www.isi.edu/div7/yoid/docs/index.html

14 Staffing

- Year 1
    - Co-PI: Ramesh Govindan and Paul Francis – Paul Francis left the project shortly after inception to join the startup Fast Forward Networks.
    - Computer Scientist: Cengiz Alaettinoglu – Cengiz Alaettinoglu left the project later in the year to join the startup Packet Design.
    - Programmer: Yuri Pryadkin
    - Graduate Student: Pavlin Radoslavov
- Year 2
    - PI: Ramesh Govindan – Ramesh Govindan left at the end of Year 2 to join ICSI.
    - Computer Scientist: Robert Lindell
    - Programmer: Yuri Pryadkin
    - Graduate Student: Pavlin Radoslavov
- Year 3
    - PI: Robert Lindell
    - Programmer: Yuri Pryadkin
    - Graduate Student: Pavlin Radoslavov – Pavlin Radoslavov graduated with his Ph.D.

15 Conclusion

The Yallcast project has successfully designed and implemented a tree-first approach to application layer multicast. Novel algorithms have been developed address the particular needs of this tree-first approach. These include fast loop detection, repair, and quality of service adaptation. These algorithms were extensively tested using simulation, experimentation in the wide area, and real world usage by hosting a variety of applications.

In addition to developing Yoid, a significant effort was made to port a variety of existing conferencing tools to run over Yoid. A mixture of multiparty audio, video, and whiteboard conferencing tools were ported to use the Yoid multicast substrate. These conferencing tools, hosted over Yoid, where used regularly to conduct weekly Yallcast meetings. Using the Yoid implementation on a regular basis helped to debug and improve the quality of the resultant Yoid software distributions that we have made publicly available. Scripts to execute these applications over Yoid are provided with the Yoid software distributions, which are available for Windows, FreeBSD, Linux, and Solaris.