

AFRL-RI-RS-TR-2008-325
Final Technical Report
December 2008



**DEFENSE ADVANCED RESEARCH PROJECTS
AGENCY (DARPA) NETWORK ARCHIVE (DNA)**

Net-Scale Technologies, Inc.

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-2008-325 HAS BEEN REVIEWED AND IS APPROVED FOR PUBLICATION IN ACCORDANCE WITH ASSIGNED DISTRIBUTION STATEMENT.

FOR THE DIRECTOR:

/s/
JAMES HANNA
Work Unit Manager

/s/
JAMES W. CUSACK, Chief
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) DEC 2008		2. REPORT TYPE Final		3. DATES COVERED (From - To) Sep 06 – Sep 08	
4. TITLE AND SUBTITLE DEFENSE ADVANCED RESEARCH PROJECTS AGENCY (DARPA) NETWORK ARCHIVE (DNA)				5a. CONTRACT NUMBER FA8650-06-C-7638	
				5b. GRANT NUMBER N/A	
				5c. PROGRAM ELEMENT NUMBER 62304E	
6. AUTHOR(S) B. Flepp				5d. PROJECT NUMBER DANA	
				5e. TASK NUMBER SC	
				5f. WORK UNIT NUMBER 01	
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Net-Scale Technologies, Inc. 281 State Hwy 79 Morganville, NJ 07751-1157				8. PERFORMING ORGANIZATION REPORT NUMBER N/A	
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES) AFRL/RISE 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-2008-325	
12. DISTRIBUTION AVAILABILITY STATEMENT <i>APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED. PA# 88ABW-2009-0322</i>					
13. SUPPLEMENTARY NOTES					
14. ABSTRACT The purpose of this project is to gain practical experience with designing and implementing a secure distributed storage system. In particular, this project investigated the various relevant topics. We sought to identify the hard problems in combining usability with security. Typically ease of use, convenience, any time anywhere access and security are contradicting requirements. We sought to understand how this is best approached. Further we endeavor to understand whether today's state-of-the-art is sufficient. A number of secure systems with secure sharing have been built, or are being built. However, no system is in practical widespread use. Does this mean there are unsolved problems, and if yes, what are they? For obvious security reasons, large organizations require review and certification of all software installed on laptops and PC's (Personal Computers). Is zero footprint web access a practical solution? Zero footprint web access makes this costly and time consuming process unnecessary because it runs entirely in an existing browser environment which has already been certified by most organizations.					
15. SUBJECT TERMS Network based data storage, Metadata aided intelligent retrieval, management metadata, automated metadata.					
16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT	b. ABSTRACT	c. THIS PAGE			James P. Hanna
U	U	U	UU	99	19b. TELEPHONE NUMBER (Include area code) N/A

Table of Contents

1.	Goal.....	1
1.1.	Project Goal	1
1.2.	Long Term Goal	1
2.	Overview	3
2.1.	Purpose of this Document	3
2.2.	Purpose of this Project.....	3
2.3.	Guiding Design Principles for Security	5
2.4.	Concepts to be Implemented and Tested	5
2.5.	Concepts Left Out in this Prototype	6
2.6.	Development Process.....	8
2.7.	Risk Factors	8
3.	Requirements	10
3.1.	Secure Distributed File System	10
3.2.	Storage Node.....	11
3.3.	Web Access Client.....	12
4.	Introduction to Cryptographic Technology Used.....	13
4.1.	Random Key Generation	13
4.2.	Symmetric Encryption	13
4.3.	Asymmetric Encryption.....	16
4.4.	User Certificates	17
4.5.	Cryptographic Hash Function (Message Digest).....	18
4.6.	Cryptographic Links.....	19
4.7.	Secure Sharing.....	20

4.8.	Inheritance of Access Rights	22
4.9.	Security Model	23
4.9.1.	Protection Against Unauthorized Data Read Access	23
4.9.2.	Protection Against Unauthorized Data Modification	23
4.9.3.	Protection Against Data Loss	24
4.9.4.	Private Key Protection.....	24
4.9.5.	Key Recovery.....	25
4.10.	Access Rights	25
5.	Architecture	26
5.1.	Overview	26
5.2.	Inheritance of Access Rights	31
5.3.	Encrypted File System.....	32
5.3.1.	Directories	34
5.3.2.	Root Directories	35
5.3.3.	Share Roots	36
5.3.4.	Back Links.....	37
5.3.5.	File System Efficiency	37
5.4.	Parallel Data Read with Automatic Load Balancing	37
5.5.	Data Write.....	38
5.5.1.	Transactional Nature of Write.....	39
5.6.	Network Data Synchronization	40
5.7.	End-to-End Security.....	41
5.8.	No Single Point of Failure.....	42
5.9.	Defense Against Denial of Service Attacks	43
6.	Secure Distributed File System Design	44

6.1.	Parallel Data Read Parameters	44
6.2.	Physical File Names	44
6.2.1.	File Name Spreading	45
6.3.	Root File Names	47
6.4.	Regular File Names	48
7.	Storage Node Design	49
7.1.	Storage Node Hardware	49
7.2.	Storage Node Software Configuration	50
7.3.	Transaction Queues for Data Integrity.....	51
7.3.1.	Transaction Queue Design.....	52
7.3.2.	What Gets Entered into the Transaction Queue?.....	53
7.3.3.	Some Design Consequences	53
7.3.4.	Headaches Caused by POSIX Lock Specification	54
7.3.5.	Problem with Combining rename() and File Locking.....	55
7.4.	Domain Name and Server List Management.....	56
8.	Access Client Design	59
8.1.	CSS.....	59
8.2.	JavaScript	59
8.2.1.	Graphical Interface to user	59
8.2.2.	Interface to Java	60
8.3.	Java	66
9.	Storage Node and Client Integration	67
9.1.	Overview	67
9.2.	Network File Server Functions	68
9.3.	Storage Node Access Protocol (WebDAV).....	69

9.4.	Design Decision: Apache mod_dav or Net-Scale Daemon	69
9.5.	Protocol Definition	71
9.5.1.	Overview	71
9.5.2.	GET Method	71
	GET Request Syntax and Header Fields	71
	GET Response Header Fields	72
	GET Response Status Codes	72
9.5.3.	PUT Method	72
	PUT Request Syntax and Header Fields	72
	PUT Response Header Fields	73
	PUT Response Status Codes	73
9.5.4.	COPY Method	74
	COPY Request Syntax and Header Fields	74
	COPY Response Header Fields	74
	COPY Response Status Codes	74
9.5.5.	DELETE Method	75
	DELETE Request Syntax and Header Fields	75
	DELETE Response Header Fields	75
	DELETE Response Status Codes	75
9.5.6.	IHLOAD Method	76
	IHLOAD Request Syntax and Header Fields	76

IHLOAD Response Header Fields	76
IHLOAD Response Status Codes.....	77
9.5.7. IHSAVE Method	77
IHSAVE Request Syntax and Header Fields	77
IHSAVE Response Header Fields	78
IHSAVE Response Status Codes	78
9.5.8. OPENTRANS Method	78
OPENTRANS Request Syntax and Header Fields	78
OPENTRANS Response Header Fields	78
OPENTRANS Response Status Codes	79
9.5.9. CLOSETRANS Method	79
CLOSETRANS Request Syntax and Header Fields	79
CLOSETRANS Response Header Fields	79
CLOSETRANS Response Status Codes	79
9.6. Design Compromises and Deferred Features.....	80
9.6.1. Changing the Locks (Access Revocation).....	80
9.6.2. File Date and Time Stamps	81
9.6.3. Directories in Memory	81
9.6.4. Root Directories Cannot be Shared	82
9.6.5. Cryptographic Link ID Rollovers	82
10. Decision Guidelines	83
10.1. Critical Items	83

10.2.	Important Items.....	84
10.3.	Less Important Items.....	85
	References.....	86
	List of Acronyms.....	87

List of Figures

Figure 1. Cryptographic links used for secure sharing of a document. The document is encrypted using random secret key k_d that is not shared with the users. Instead, each user gets a cryptographic link (CL1...CL3). Together with their private secret keys ($k_1...k_3$) users can derive the secret key k_d upon reading the document. Note that k_d is never stored anywhere.....	20
Figure 2. Secure inheritance of access rights over multiple directory levels.	22
Figure 3. An architecture overview, illustrated here with a single storage node (left) and a single user client (right). Orange components are built by Net-Scale.	27
Figure 4. Prototype system overview. The storage nodes are intentionally operated from places with (unreliable) consumer Internet connection and advertise their current IP address to a Dynamic DNS (DDNS) service provider. This allows the users to find the storage nodes even when their IP (Internet Protocol) addresses change frequently.....	29
Figure 5. Structure of a secure file, which can represent either a clear file or a clear directory.	33
Figure 6. An encrypted user directory contains pointers to its files and subdirectories in the encrypted data part.....	34
Figure 7. Example of a small physical directory tree that is used to store encrypted files which form the actual user directory trees. The user directory trees are not correlated to this physical directory tree at all. To conserve drawing space, one directory level is omitted and the file names are shortened.....	46
Figure 8. A possible storage node hardware configuration: Dell Inspiron 520s slim line design, which does not take up a lot of space. The software is installed and configured by Net-Scale and no keyboard or screen is necessary for operation.	49

Figure 9. Network diagram of the dynamic DNS (DDNS) setup. 57

Figure 10. Integration of a storage node and access client and integration of a storage node with the other storage nodes. HTTP is used only for starting the web applications. It sends the web content, JavaScript, and Java code to the browser. WebDAV is used for all storage data interactions between the access client and the storage node(s) and for the interactions between the storage nodes. 67

1. Goal

1.1. Project Goal

Demonstrate that a robust distributed storage network with end-to-end security and a good user experience is feasible by designing, implementing, and deploying a prototype system with multiple storage nodes and a zero footprint web access client.

1.2. Long Term Goal

Make network storage as vendor and technology independent, robust, and scalable as the Internet is today while maintaining end-to-end security and zero footprint web access.

This means an organization or group can operate such a storage network over decades mixing many storage nodes, which come from different vendors and from different technology generations. If a new node is added, the system simply gains more storage capacity, becomes more robust and provides higher access bandwidth, all without elaborate manual reconfiguration and optimization of the network.

Achieving this vision requires a reference design and implementation for the overlaying principles and interfaces such that technology from many vendors and many generations can work together. **Who will do it?** Commercial vendors have no incentive because that contradicts their most successful business model (lock customers into proprietary technology). On the other hand, the open source and research community has shown little commitment to this either. This community seems to be more focused on creating the best possible system within a single technology generation.

A global open system will have to make compromises and cannot be the best possible solution in all aspects. However, by spreading out globally its total value will be far greater than that of the sum of many superior "island" solutions. Not only is such a system vendor independent, but also its code diversity makes it much more secure against system-wide attacks by hackers.

This global solution is our main motivation. Success will require a good understanding of the work of all main players in this field including their goals and concerns. It will further require building robust easy to use reference systems quickly, and finally, it will require diplomatic skills for bringing people together.

How do we get there?

- ◆ Step 1 is to get a good understanding of the current state-of-the-art and who the players are. This involves studying the literature and actively participating in and organizing workshops and conferences. This activity is accompanied by developing the details of the interfaces and the requirements for participating systems. Finally, we need to build credibility for ourselves by building a real working prototype system, which solves at least a few recognized key challenges.

- ◆ Step 2 is gaining the trust and support of a championing organization with worldwide respect, which can successfully support and promote the concept. In our view, a prime candidate for this is DARPA (Defense Advanced Research Projects Agency).

- ◆ Step 3 consists of building alliances with key parties and implementing a plan to create and roll out the first real system with multiple organizations (representing industry, the open source community, and Government) involved from the very beginning.

- ◆ Step 4 involves setting up a robust governing structure of the new system for controlling the few things that need central control and for developing and maintaining guidelines and standards.

2. Overview

2.1. Purpose of this Document

This document contains the requirements, architecture and design of the secure distributed storage prototype system.

2.2. Purpose of this Project

The purpose of this project is to gain real life practical experience with designing and implementing a secure distributed storage system. In particular, we want to investigate:

- ◆ **What are the hard problems in combining usability with security?** Typically, ease of use, convenience, any time anywhere access and security are contradicting requirements.

We need to understand how this is best approached.

- ◆ **Is today's state-of-the-art sufficient?** A number of secure systems with secure sharing have been built, or are being built, including OceanStore [1], [2] (<http://oceanstore.cs.berkeley.edu>), Wuala [3] (<http://wua.la>), Allmydata (<http://allmydata.org>). However, no system is in practical widespread use. Does this mean there are unsolved problems, and if yes, what are they?

Is zero footprint web access a practical solution? For obvious security reasons, large organizations require review and certification of all software installed on laptops and PCs (Personal Computers). Zero footprint web access makes this costly and time consuming process unnecessary because it runs entirely in an existing browser environment which has already been certified by most organizations. Furthermore, zero footprint web access is device independent and can be extended relatively easily to work on hand-held mobile devices.

◆ **What are the hard problems in making a secure system vendor independent?** To our knowledge, nobody has addressed this issue yet. All existing systems and concepts assume that the entire system consists of a single coherent software layer. However, any long-term stable solution has to be vendor and technology independent.

We understand that this prototype will not address all critical technologies and key challenges. However, given the limited time and budget we feel that it is more important to complete an end-to-end system and be able to learn real life lessons from it, than working towards a large feature complete system but not gaining any real life feed-back at all.

This is also in line with the standard Net-Scale iterative development approach wherein this project can be viewed as the first iteration in a larger endeavor to set the standards and create reference implementations for a vendor and technology independent secure distributed storage mesh.

Conducting a formal trial is not budgeted within this project. However, such a trial could be realized in a follow-up project. Even without a formal trial, building this prototype system end-to-end and opening it to a few real users will provide invaluable lessons for the planning of future work.

2.3. Guiding Design Principles for Security

1. **Minimization of the security sensitive system parts.** In essence, the system will only rely on standard cryptographic algorithms and libraries for security. Any other part or code should not compromise security if it has bugs or malfunctions (it may bring down the system temporarily but no data is lost or compromised).
2. **Decentralization of break-in points.** Break-in to any part of the system will give an intruder at most access to a small part of the data. For example, there is no central user database which, if compromised would give an intruder access to the entire system. This is achieved in a large part by end-to-end security (see below).
3. **End-to-end security.** The access client through use of a private key can only decrypt data, which is stored on an external USB (Universal Serial Bus) dongle. Users do not have to trust the storage nodes nor the access network to keep their data secret.

2.4. Concepts to be Implemented and Tested

Following is the original list. Most concepts have been implemented and tested with the exception of distributed storage (Bullet 2 and its dependents, Bullets 3, 6, and 7).

- ◆ **No client side software installation (zero foot print web access).** Users are not required to install any client side software. The entire access client is loaded through the web and runs on the PC's existing web browser environment. Nevertheless, users can interact with the local file system, for example, for bulk upload or download.
- ◆ **No single point of failure.** Any individual component of the system can fail at any time with no prior warning and without affecting the system functionality.
- ◆ **Geographically distributed storage.** Data is replicated over multiple storage nodes in different parts of the country and if possible in different continents.

- ◆ **End-to-end security.** Security and access control are managed through data encryption and not through traditional gatekeepers. No single point of security vulnerability.
- ◆ **Secure sharing.** Users can give other users access to directory trees and therefore the files contained in these directories or folders. They can later revoke that access right without the need to re-encrypt all files and redistribute new keys to other users with access rights to those files.
- ◆ **Access bandwidth is higher than the fastest storage node.** Access clients will retrieve files from all available storage nodes in parallel, which makes the total access bandwidth approach to the sum of the available storage node bandwidths.
- ◆ **Data integrity.** When a user stores or modifies a document and not all storage nodes are on-line at that time, then the network will ensure data integrity behind the scenes. The user does not have to wait until all storage nodes are on-line for the write operation to complete.

2.5. Concepts Left Out in this Prototype

Concepts and technologies which are not implemented in this project include a) the creation of separate web access nodes for hand-held clients and cell phones, b) off-line capability of the access clients, and c) client features which limit document handling rights, such as allowing users to view a document but not to print or save it. We felt that these technologies are sufficiently well understood that they do not necessarily need to be tested as part of this project.

Additionally, there are a few concepts, which we do believe are not yet sufficiently understood, but which did not fit into this project. These include:

- ◆ **Secure search.** Data is not efficiently searchable in the network, as no index in the network is implemented. Advancements in cryptography may one day allow secure indexing on encrypted data (see, for example, [4]).
- ◆ **Spreading of files over multiple nodes.** In this prototype, all files are fully duplicated on all storage nodes. Access to 1 out of the n nodes is therefore sufficient for data reconstruction. However, the size of available storage space does not grow with the number of storage nodes brought on-line. This makes this prototype system wasteful with disk space but greatly reduces the complexity for managing the location of user data.
- ◆ **Self-healing.** While the system is robust against individual node failures, a node loss is not automatically detected and adding a new node involves manual intervention.
- ◆ **Synchronization.** No true data synchronization between a user client and the network storage. The *drag-and-drop update mode* should be an acceptable workaround in most cases.
- ◆ **Secure revision control** and the ability to browse and restore file and tree states of the past.
- ◆ **Key recovery** through use of shared secret methods: Users trust a cryptographically spread token to each of n friends, agencies, or other parties. Exactly k ($k < n$) tokens are needed to reconstruct the key. The parties don't know of each other. This makes key recovery both highly robust and highly secure.
- ◆ **Private key protection** through use of pass phrase and/or external dongle and/or the integration with existing infrastructure, such as a CAC (Common Access Card).

◆ **Encryption algorithm upgrades.** As the available compute power increases, existing encryption methods will become less secure and eventually obsolete. This requires a periodic upgrade of the methods in use. It can be achieved by re-encrypting all documents with new keys and new methods. Since the re-encryption with new keys is already built into the system, an automated method for upgrading the encryption algorithms can easily be added.

2.6. Development Process

This project includes work that is of research nature. Not all methods and technologies, which are intended to be used, are fully understood at project start. We therefore decided for an iterative development process even within such a small project. The first iteration consisted of conducting specific experiments to test the least understood and therefore most risky components. The outcome of this experimentation phase determined the details of the remainder of the system design and project plan.

2.7. Risk Factors

◆ **Feature limitations.** What features cannot be implemented or are inefficient using end-to-end security (e.g., business processes). The two main SDF (Secure Distributed File System) characteristics to watch out for are:

- Client needs to be on-line for processing to take place.
- Data needs to be transferred to client for processing.

One way to go around such limits is to give a process running in the network read and write access to the necessary data to do its job. This will enable almost any traditional feature but will also add a (controlled) component in the network that needs to be trusted.

◆ **Public key distribution.** According to some opinion, public key infrastructure failed in the past due to the overhead of public key distribution.

◆ **Long-term cryptographic strength.** We cannot make any guarantees how long the encryption methods used today will remain strong. We can continuously refresh data with new encryption methods. However, we cannot offer protection against somebody stealing the data disk and just keeping it until the time comes when the data can be cracked. This should not be a problem, except for very special applications.

◆ **Cryptographic links introduce cryptographic weaknesses.** Does adding cryptographic links reduce the cryptographic strength of the secret key they encrypt? According to Hilary Orman this is only of theoretical concern. In practice this reduces the search space only a tiny bit and does not significantly reduce security.

◆ **Clearing secret keys from Java memory.** Secret keys should only be kept in memory as short as possible and they must be overwritten with random data or zeros before freeing the memory. That's because otherwise that secret information could get into the hands of another application that claims that memory. For the same reason we also need to prevent such memory from even be written to swap space on disk. Unfortunately, the Java libraries we use for the prototype (add reference here) does not seem to provide such functionality and our application code cannot clear the keys manually because they are held in private parts of the classes which are not accessible by the application code. The solution will be to discover ways to clear secrets with the current library after all, find a better Java cryptographic library implementation, or worst, case, implement the necessary cryptographic algorithms ourselves. Java control over memory is, in principle, possible. See for example <http://java.sun.com/j2se/1.5.0/docs/api/java/nio/ByteBuffer.html>.

3. Requirements

Following is the original lists. Most requirements are fulfilled with the exception of those with dependencies to data distribution (see Section 2.4.). It concerns Section 3.1. , Bullet 4, Section 3.2. , Bullets 3, and 4, Section 3.3. Bullet 6).

3.1. Secure Distributed File System

- ◆ End-to-end security. The end user can only decrypt data with proper read privileges using that user's private key. The private key is stored on a USB dongle and is never stored on the client PC or a storage node.
- ◆ Data consistency through proper locking mechanism, including atomic operations over multiple files.
- ◆ Data integrity. Data is never left in an inconsistent state, even if network connection is lost in the worst possible moment.
- ◆ Synchronization, i.e., automatic propagation of modifications to other nodes.
- ◆ Avoidance of dead lock and infinite loop situations.

3.2. Storage Node

- ◆ Implementation of encrypted file system.
- ◆ Standard web server hosting the access client.
- ◆ Dynamic DNS (Domain Name System) client.
- ◆ Regular submission of health and statistic information to a central web server.
- ◆ System administrator data disk copy tool.
- ◆ System administrator configuration tools must be avoided or minimal.
- ◆ System administrator diagnostic tools: useful information in log files and regularly copied onto a central monitoring web server. Ability for remote access and remote software updates.

3.3. Web Access Client

- ◆ Web based Ajax (Asynchronous JavaScript and XML) client. No client side software installation or configuration required. Nice looking, easy to use, and quickly responding user interface.
- ◆ Bulk drag-and-drop with overwrite and update modes. Drag-and-drop between the local file system and network storage but also within the network storage or within the local file system. "Update mode" means that only files that are older on the target than the source are overwritten.
- ◆ Guaranteed error discovery. If a bulk upload completes without error, user must have guarantee that all data was transferred correctly.
- ◆ Robustness against network interruption and PC reboot. If a user's PC is shut down during a data transfer, the transfer must continue where it left off when the PC boots up again and the user reloads the web access client.
- ◆ Encryption and decryption of data is done by the Java applet on the fly and is hidden from the user.
- ◆ Parallel download with automatic load balancing to maximize the effective download bandwidth.

4. Introduction to Cryptographic Technology Used

4.1. Random Key Generation

The data of each document and directories is encrypted with a different random encryption key. Furthermore, these key changes each time the document or folder data changes. The random key therefore must be a true (not a pseudo) random number and the quality of the true randomness directly affects the encryption strength.

We are not deeply investigating the topic of cryptographic random number generation but assume the problem is solvable with acceptable quality and reasonable resources. On the C side, we use the random key generator of the OpenSSL (Open Source Secure Socket Library) library (`EVP_CIPHER_CTX_rand_key()`) for now. Of course, the C code is only used for testing and debugging and the random keys for real documents will be generated by the Java code of the access client.

4.2. Symmetric Encryption

- ◆ AES-128 (Advanced Encryption Standard, Federal Information Processing Standard U.S. FIPS PUB 197, November 26, 2001).

Symmetric encryption is used for encrypting the data in files and directories. We use AES, which is the Advanced Encryption Standard, standardized by NIST (National Institute of Standards and Technology) (FIPS-197). See also Wikipedia. AES seems to be widely used as secure and is generally regarded as the successor to DES (Digital Encryption Standard). AES was therefore an easy choice. However, we needed to make a few additional decisions, which are less obvious.

One is the encryption key length. AES supports key sizes of 128, 192, and 256 bit. 128 bit are viewed as providing strong encryption and we assume 128 bit requires less CPU (Central Processing Unit) time compared to longer key lengths. Furthermore, the Sun Java Runtime Environment comes standard with 128-bit support but requires extra installation steps for 192 and 256-bit support. The only reason for longer keys seems to be for applications where data needs to remain secret for several decades into the future. This is not the case for our prototype. Even a future system will likely be designed to support key refreshing to allow users to periodically re-encrypt stored files with stronger keys over time. Our conclusion is that for the purpose of proof of concept of this prototype, 128 bit encryption keys are adequate.

Next, we need to consider that AES is a block cipher, which encrypts exactly 16 bytes (no more and no less). We therefore need to decide on a method on how to use a block cipher algorithm for arbitrary file lengths. Just applying AES on each 16-byte block of a file individually is a bad idea. Because the same input always generates the same output, certain regularities of the input file are still visible in the output file. An good example is shown in Wikipedia. The same page also contains a good overview of better methods.

We decided for Cipher Feed-Back (CFB). The corresponding function in the OpenSSL library is called `EVP_aes_128_cfb128`. The first 128 stands for the encryption key length and the second 128 indicate the window size of the CFB algorithm (see the above Wikipedia page (http://en.wikipedia.org/wiki/Advanced_Encryption_Standard) for details). Smaller windows increase the computational overhead without apparent advantages for encryption strength.

Finally, we need to decide on the value used for the initialization vector (IV), which is needed for the CFB algorithm. This value is not critical for encryption strength. One should only avoid using the same IV/key pair to encrypt more than one set of data. Since we use different random keys for each file and directory, this is not a problem for our application. We therefore initialize the IV with all zeros.

About CPU time required for encryption/decryption: Encrypting or decrypting a 5-MB file takes less than about 100ms on a Dell Dimension E520 workstation (ws01). Even when executed on an older PC or laptop with a Java layer between we can expect that the bottleneck will be the communication bandwidth and not the encryption and decryption speed.

For the Java version of the library, we tested two different symmetric encryption providers: BouncyCastle and Sun JCE (Java Cryptographic Extension). Both implementations are using the same AES/CFB algorithms with no padding as the C version of the library, and are wrapped inside a Java JCE cipher object.

From a functional point of view, there is no difference in encoding and decoding results as for the C version, which means we can use the C version to encode and the Java version to decode and vice versa.

From a performance point of view, there are significant differences in both offset and throughput speed. The throughput of both Java providers are slower by a factor of two compared to the C version.

The offset to initialize the libraries show differences as follows:

1. C Version: < 20 ms
2. BouncyCastle Java Version: ~ 600 ms
3. Sun JCE Java Version: ~ 300 ms

4.3. Asymmetric Encryption

◆ RSA-2048

Asymmetric encryption is used for the cryptographic links to the root directories. This allows user A to grant access to user B without knowing user B's private key. Knowledge of user B's public key is sufficient to create the necessary cryptographic link.

The RSA algorithm requires a padding parameter. We use `RSA_PKCS1_PADDING`, which appears to be the most commonly used. The two additional parameters needed by the RSA algorithm are key length and the public exponent. The PKCS12 (Public Key Cryptography Standards Version 12) certificate issuer that creates the user's personal certificate determines both these values. For the prototype we will use 2048 bit key length (which is considered strong encryption) and uses a public exponent of 65537. Note that some people consider 1024 key length is considered sufficiently strong.

As AES, RSA is a block cipher algorithm which means it operates on a fixed block length. Unlike AES, the RSA block length is tied to the key length and the padding algorithm used. For `RSA_PKCS1_PADDING` the block length is $2048/8-11 = 245$ bytes (see http://openssl.org/docs/crypto/RSA_public_encrypt.html for more information). As we only use RSA for the encryption of random AES keys (of length 128 bit = 16 byte), no chaining is necessary, i.e., all data can be encrypted in a single block.

The encrypted output data size is always equal to the key length (256 bytes in case of a 2048-bit key). Note, that the RSA algorithm creates different output when called multiple times, even when the same input data, key, and parameters are used. However, all the different outputs generated will decrypt to the same (correct) clear text.

4.4. User Certificates

In order to use the SDF, users need a public and a private key. A convenient way to store both keys is the standard PKCS12 personal certificate format (see, for example, <http://en.wikipedia.org/wiki/PKCS12>). PKCS12 files, which typically have the file extension .p12, store the public key in clear and the private key encrypted with a pass phrase. Using this standard file format allows us to use an existing certificate authority to create user certificates (including the user's public and private keys). The OpenSSL library provides the functionality to decrypt and parse PKCS12 files. The user certificates will be used in a number of ways:

- ◆ A hash of the user's public key is used to create the physical file name of that user's root directory.
- ◆ A hash of two user's public keys is used for the physical file name of shared root directories.
- ◆ Other users to create cryptographic links for this user use a user's public key.
- ◆ User certificates are used to sign files that that user created.
- ◆ The storage nodes will check the digital signatures of files to enforce write policies (currently, only the owner of a file can modify or delete that file).

4.5. Cryptographic Hash Function (Message Digest)

We use cryptographic hash functions (also called message digests or md) in two places: 1) to derive a physical file name from one or two users' public keys, and 2) for digitally signing the content of files or directories. A cryptographic hash function has the following properties (from Wikipedia):

1. It is computationally infeasible to find a message that corresponds to a given message digest.
2. It is computationally infeasible to find two different messages that produce the same message digest.
3. Any change to a message (including single bit changes) will, with an exceedingly high probability, results in a completely different message digest.

For deriving a physical file name from public key(s) we don't need property 1 but we do need it for digitally signing a file. For simplicity, we use the same algorithm in both cases.

MD5 (Message Digest Algorithm 5) is a popular algorithm used widely on the Internet but today considered not secure. SHA1 (Secure Hash Algorithm) was designed to be the successor of MD5 and standardized by NIST. SHA-1 generates 128-bit output. Later, several new SHA versions were added to generate longer output: SHA-224, SHA-256, SHA-384, and SHA-512. Today, there are some concerns about the security of those too but apparently no successful attack has yet been reported. In 2007 NIST started an open competition for a new SHA-3 algorithm.

For now, we are using SHA-256, which in OpenSSL is called sha256.

Here are two examples, generated with SHA-256 (output in hex):

◆ "Net-Scale" generates:

972a7bbd7de162694142b35a298e111adf9429dbddb8ff311bb2467b78edf34b

◆ "Net Scale" generates:

66e27f6180abab57b328471f90c5123251a60c9849647c6fec6316fe5bb10a5

4.6. Cryptographic Links

A cryptographic link from a secret key k_1 to another secret key k_2 ($k_1 \bullet k_2$) lets anybody in possession of k_1 derive k_2 but not vice versa. In the present implementation we achieve this in one of two ways:

1. **Symmetric cryptographic link:** k_2 is encrypted using k_1 as the encryption key. This requires the party who creates the link to know secret k_1 .
2. **Asymmetric cryptographic link:** k_1 is the private key of a public/private key pair and the public key p_1 is used to encrypt k_2 . The party creating the cryptographic link therefore does not need to know secret k_1 .

We use cryptographic links for secure sharing of documents and for inheritance of access privileges in our secure directory tree.

4.7. Secure Sharing

The most basic case is shown in Figure 1. A document is encrypted using a random secret key k_d . That secret key is never stored nor is it directly shared with the users who have access to that document. Instead, each such user receives a cryptographic link CL_i from his or her private key k_i to the document key k_d . This allows users to calculate the document key k_d and therefore gives any user in possession of a cryptographic link to that key (k_d) access to the document itself.

Whenever the document is modified, it will be encrypted with a new random key and all

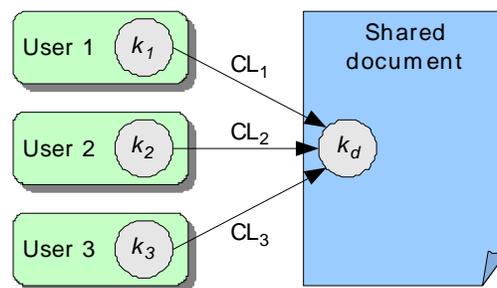


Figure 1. Cryptographic links used for secure sharing of a document. The document is encrypted using random secret key k_d that is not shared with the users. Instead, each user gets a cryptographic link ($CL_1 \dots CL_3$). Together with their private secret keys ($k_1 \dots k_3$) users can derive the secret key k_d upon reading the document. Note that k_d is never stored anywhere.

cryptographic links to that key are updated accordingly.

To revoke access rights of a user to that document that user's cryptographic link to the document key is simply removed. One can argue that the user could have stored the secret document key k_d locally and can therefore continue to read the document, even though his or her cryptographic link was removed. That is correct and could be prevented by automatically re-

encrypting the document with a new secret key whenever the access rights to it change. However, we argue that this does not add significantly to the security of the system. That's because if the user had bad intentions in the first place, as evidenced by him or her remembering the secret document key in the first place, that user might as well store the entire document at that time with no extra effort.

4.8. Inheritance of Access Rights

Figure 2 shows a case where access rights are automatically inherited over multiple directory levels. The content of each directory, i.e., its pointers to subdirectories and files, is encrypted using a random secret key k_{fi} . Each user gets a cryptographic link to the secret key of the high-

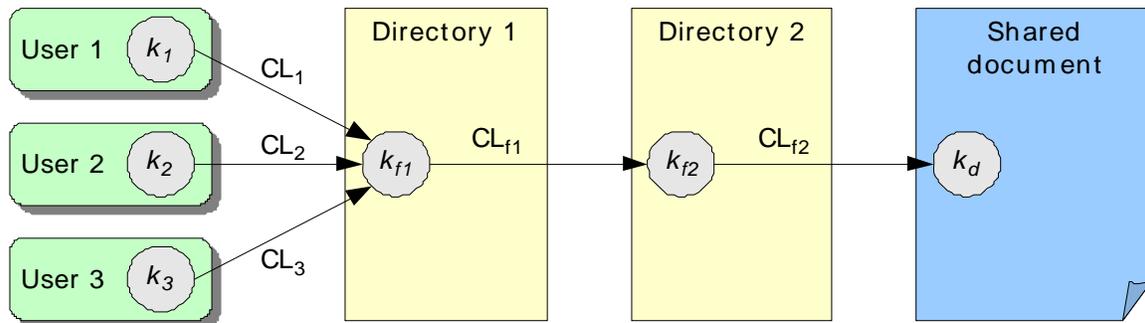


Figure 2. Secure inheritance of access rights over multiple directory levels.

est directory level they have access. Each directory has a cryptographic link from its secret key to the secret keys of all its subdirectories and files. In the case shown in Figure 2, user 1, for example, can use cryptographic link CL_1 to obtain secret key k_{f1} . That secret together with cryptographic link CL_{f1} is then used to obtain secret key k_{f2} , which in turn, together with CL_{f2} is used to calculate k_d , which is used to decrypt the document.

These examples show how document and directory information can be stored and shared securely, guaranteeing that only parties in possession of a legitimate private key can access that data. In the next section we discuss the management of access rights in general, including modification rights and the right to grant other users access.

The use of cryptographic links for inheritance of access rights and secure sharing of data is inspired by work described in [3] and [5].

4.9. Security Model

4.9.1. *Protection Against Unauthorized Data Read Access*

As shown in the previous section, this protection is tied through a series of cryptographic links back to a private secret key which only the user holds and which is never stored in the network.

4.9.2. *Protection Against Unauthorized Data Modification*

No cryptographic technology alone can protect against data loss, such as a hacker breaking into a system and erasing all data. However, cryptographic technology can be used for protection against unauthorized data modification without other users noticing, by signing documents that were written. This digital signature can be tied back to the same private keys of the users, which are used, for secure read access.

In addition, the owner of a directory or file can digitally sign the cryptographic links granting read and write access such that each user can detect unauthorized data modification by either a party without read nor write access or by a party with read but no write access to a file or directory.

In a secure system, such unauthorized data modifications must be treated equivalent to data loss (e.g., disc parity error due to physical data corruption).

4.9.3. *Protection Against Data Loss*

Protection against data loss, both physical data loss as well as unauthorized data modification by an adversary, is achieved through data redundancy in the network, i.e., by storing each file on multiple storage nodes.

Data loss on a single storage node then only leads to a reduction of the redundancy level but not to actual data loss. A system can detect this and take steps to restore the desired redundancy level automatically. However, in the interest of time, such steps will not be implemented in the present prototype system.

Finally, note the single point of failure if all storage nodes use the same or related server software. The only protection against this case is through use of diverse software written by independent parties, which is outside the scope of this project.

4.9.4. *Private Key Protection*

As so much of the system security depends on the users' private keys, we need to look at how to protect those keys. If, for example, a hacker succeeds in stealing a private key from a user, that hacker gains full access to the same privileges the user has.

In this prototype system, private keys will not be stored on any laptop or PC but on a read-only USB dongle. Commercial dongle products exist which store a pass phrase protected version of the private key and do all the encryption processing, which involves the private key on the dongle itself. If the user enters the wrong pass phrase n times the dongle self-destructs. It also self-destructs upon attempt to physically open it. This makes it virtually impossible to steal a user's private key, even if the user is negligent (short of writing down their pass phrase on the dongle and leave the dongle laying around).

Since this technology is commercially available there is not much value in testing it in the present prototype and in the interest of time we decided to store the user's private keys in clear on a read-only USB dongle. Each user will be handed three copies of that dongle, one to use and two to store in a safe place of the user's choice, preferably not in their house and not in their office.

4.9.5. *Key Recovery*

In a future system we anticipate users would backup their private keys differently: Threshold methods are well suited for this purpose. Users spread out their secret private key over n tokens, let's say 10. To reconstruct their secret key they need access to at least k of those tokens, let's say 3 but any k tokens will work. Users will trust friends or colleagues or companies that provide that service with safekeeping one of the n tokens. The ten parties do not know of each other. Unless at least 8 parties loose the token they were trusted with safekeeping, the user can still reconstruct their private key. On the other hand, no single party can break into the user's files. At least three of them would have to stick together. However, this scheme will not be implemented in this prototype.

4.10. Access Rights

In the general case, one can imagine a large number of fine-grained access rights. However, three access rights (per file and directory) are probably most important:

- ◆ Read access right.
- ◆ Write access right.
- ◆ Right to modify other user's access right.

For simplicity reasons only read access control was implemented in this prototype.

5. Architecture

The main architectural goals are:

- ◆ Provide ease of use, avoid client software certification and installation processes, and provide platform independence through zero foot print web access.
- ◆ Achieve high system availability through geographical storage node distribution providing redundancy for network outage, power outage, and hardware failure.
- ◆ Guarantee privacy through end-to-end security.
- ◆ Guarantee data integrity through end-to-end security.
- ◆ Protect against data loss through redundancy with geographical data distribution and write privilege enforcement by the storage node server software.

5.1. Overview

Figure 3 shows an architecture overview with a single storage node and a single user client. A user starts by accessing the web server URL (Uniform Resource Locator) that loads the Net-Scale web client to the browser and starts executing its code. The web client consists of static code including Cascading Style Sheets (CSS) and an initially empty HTML (Hyper Text Markup Language) page. All functionality is provided by the dynamic code that includes JavaScript and a Java applet. The JavaScript code creates HTML elements dynamically and places or moves them in the browser screen to create the desired user interface and effects. The CSS code defines the graphical attributes of the HTML objects. The JavaScript code further interacts with the Java applet.

The Java applet accesses the encrypted data on the storage node through the web server and the Net-Scale code connected to the web server. All data encryption and decryption is performed by the Java applet on the user PC or laptop. The Net-Scale code on the server node

provides only minimal functionality, as explained later. WebDAV (Web-based Distributed Authoring and Versioning) is used as the transport protocol. Only a small subset of WebDAV is needed. Not used from WebDAV, in particular, are encryption (as the data is already end-to-end encrypted) and authentication. Nevertheless, WebDAV has the advantage for our purpose of being a well-specified and documented protocol, running on top of HTTP, which minimizes firewall issues, and lets us reuse existing web server plug-ins.

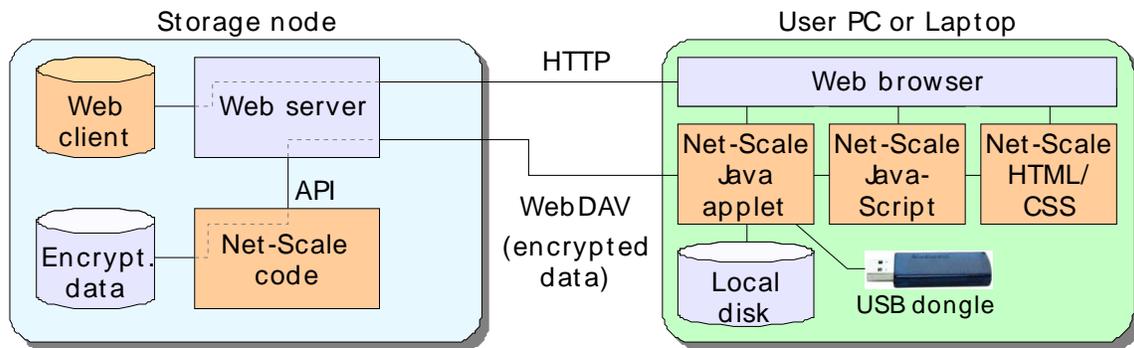


Figure 3. An architecture overview, illustrated here with a single storage node (left) and a single user client (right). Orange components are built by Net-Scale.

The Java applet further can access the local disk(s) of the user PC or laptop. Together with the user interface layer controlled by the JavaScript code, this provides the bulk upload and download capability (from and to the network storage). Finally, the Java applet reads the user's private key when needed which is stored on an external USB dongle. The private key is at the root of the end-to-end security infrastructure and is the only secret a user needs to keep. For this prototype system we will use regular USB drives as a dongle but much more secure USB devices for storing private keys exist. However, the private key is never stored on any disk. The USB dongles therefore are the only physical item users need to protect and without the dongle, users cannot access any data.

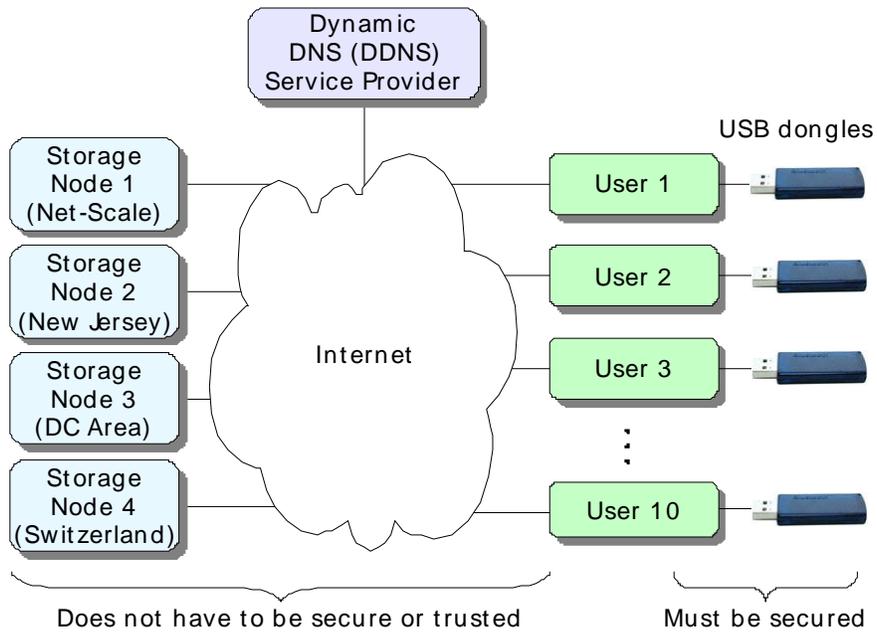


Figure 4. Prototype system overview. The storage nodes are intentionally operated from places with (unreliable) consumer Internet connection and advertise their current IP address to a Dynamic DNS (DDNS) service provider. This allows the users to find the storage nodes even when their IP (Internet Protocol) addresses change frequently.

Figure 4 shows a larger system with multiple storage nodes and users. Anticipated for this prototype system are four storage nodes and approximately ten users.

Storage nodes are intentionally operated from consumer Internet connections with limited bandwidth and network reliability. The prototype therefore has to prove that this architecture can provide high reliability and access bandwidth even in these conditions. Because most consumer Internet connections have no-static IP addresses, the system includes the use of a Dynamic DNS (DDNS) service provider.¹

A note about firewalls: no firewall required to make the system secure. However, a storage node will still work, even if installed behind a firewall, as long as the firewall is configured to pass HTTP (Hyper Text Transfer Protocol) requests to the storage node computer. This will require some manual firewall configuration by the user who hosts the storage node.

1 Note about access nodes: Readers who are familiar with the Net-scale Statement of Work for the Seedling project ("The Storage Network of the Future", Version 2.2) may miss the green access nodes shown in Figure 2 of that document. This is not because the access nodes are dropped from the general concept but they will not be implemented in this project. Their functionality is split between the storage nodes and the user clients as follows: all web content, including JavaScript and Java code is replicated on all storage nodes and all storage nodes include a web server to serve that content to the clients. The Java part of the web clients will directly communicate with the storage nodes instead of going through access nodes as intermediaries. Access nodes will be required for future hand-held web applications, which are not powerful enough to handle all client functionality on the device.

5.2. Inheritance of Access Rights

The general case of access control essentially requires an individual access control list for each file and directory. This involves a complex management overhead both for the system as well as for the users. Things become quickly complicated as soon as users start moving files within the directory tree. For example, what happens to the access control rights of document X if it is moved from folder A to folder B. Does it keep the original access control list or does it inherit any access control properties from folder B? How do users with access rights to document X know where it was moved?

For this prototype system we simplified the access control mechanism both to improve the user experience and also to limit the implementation cost. Any such simplification, of course, bears the danger that not all required cases can be satisfied. We believe, however, that the system we chose will satisfy the vast majority of all cases. This assumption will be put to the test by the prototype system usage.

Our simplified access control system assumes the presence of a directory tree. It has the following properties:

1. **Inheritance of access rights.** All its subdirectories and files inherit access rights assigned to a directory. For example, if John grants Fred access to directory `/john/projects/dna/`, then he has automatically access to `/john/projects/dna/doc/` and `/john/projects/dna/papers/` but not to `/john/projects/lagr/`. Furthermore, if John moves a file from the `dna/` to the `lagr/` directory, then Fred loses his access rights to that file. If, on the other hand, John adds a new file to the `dna/` directory, then Fred will automatically gain access to that new file.
2. **Confidentiality of access rights.** If John grants Fred access to `/john/projects/dna/`, then John cannot see who else has access rights to the same directory (or file).

5.3. Encrypted File System

The user data is stored in an encrypted file system that is overlaid on top of a regular file system. The content of user directories and user files are stored in regular files as shown in Figure 5.

In addition to the directory or file name and content which both are encrypted with secret key k_d , each file also stores the digital signature of the owner and a number of cryptographic links to the secret key k_d . The secret key k_d is not stored at all. Therefore, the cryptographic links provide the only means to access the user data. Secret keys and physical file names are chosen at random. The secret key k_d changes each time the user file or directory is modified. Somebody looking at the file system will see a collection of physical files with random file names. He or she can see who owns these items but cannot see if they represent files or directories and cannot recreate the directory tree nor access any user data.

SDF Item

SDF Version (current version: 1)	4 bytes, big-endian
Size of item header (in bytes) (file size minus encrypted data)	4 bytes, big-endian
Next free cryptographic link (CL) ID	4 bytes, big-endian
Size of CL 1 (in bytes)	2 bytes, big-endian
ID of CL 1	4 bytes, big-endian
Type of CL 1 (0= symmetric, 1= asymmetric)	4 bytes, big-endian
Size of CL 1 Field 1 (in bytes)	2 bytes, big-endian
CL 1 Field 1 (actual link)	Variable length
Size of CL 1 Field 2 (in bytes)	2 bytes, big-endian
CL 1 Field 2 (back link key)	Variable length
Size of CL 1 Field 3 (in bytes)	2 bytes, big-endian
CL 1 Field 3 (back link a)	Variable length
Size of CL 1 Field 4 (in bytes)	2 bytes, big-endian
CL 1 Field 4 (back link b)	Variable length
Repeat last 9 (blue) fields for CL 2, CL 3, ...	
Encrypted data	Variable length
Signature and public key of owner	To be added separately and later

Symmetric CL

CL Field 1 (actual link)
Item key (random, symmetric),
encrypted with CL parent's item key

CL Field 2 (back link key)
random symmetric back link key
encrypted with this owner's public key

CL Field 3 (back link a)
secure file name of CL parent,
encrypted with back link key

CL Field 4 (back link b)
Item key of CL parent,
encrypted with back link key

Asymmetric CL

CL Field 1 (actual link)
Item key (random, symmetric),
encrypted with CL parent's public key

CL Field 2 (back link key)
random symmetric back link key
encrypted with this owner's public key

CL Field 3 (back link a)
CL originating user's client certificate,
encrypted with back link key

CL Field 4 (back link b)
Not used

Figure 5. Structure of a secure file, which can represent either a clear file or a clear directory.

5.3.1. Directories

A user directory is just like a user file, except that the data part contains a list of pointers to the files and subdirectories of this directory. Such a pointer consists of the physical (random) file name of the user file or user subdirectory and the index of the cryptographic link to be used in order to access the data of that user file or user directory. This is illustrated in Figure 6.

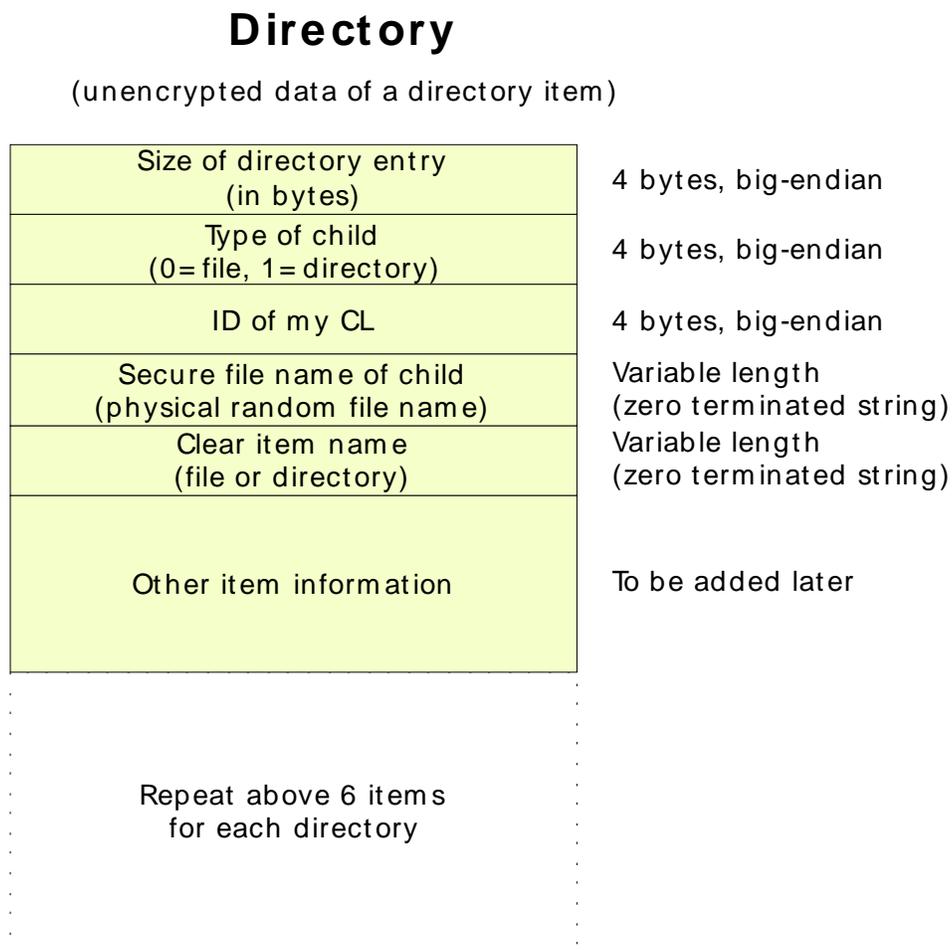


Figure 6. An encrypted user directory contains pointers to its files and subdirectories in the encrypted data part.

5.3.2. *Root Directories*

The above user directory structure allows anybody with access to a user directory to navigate its entire subdirectory tree by just following the pointers and resolving the cryptographic links along the way. However, how do we bootstrap this process or, in more practical terms, how does a user know the physical file name of his or her home directory and what is the secret to access the home directory's cryptographic link?

In this implementation we use a hash code of the user's public key as the physical file name of the home directory and the user's public key for the creation of the first cryptographic link in that home directory. That cryptographic link will therefore be an asymmetric cryptographic link while most other cryptographic links will be symmetric. Note, that in principle, all cryptographic links could be made asymmetrical but that would require unnecessary CPU overhead.

In order to read their home directory, all users need to know is their public and private key. Both are part of their user certificate, which is stored in the external USB dongle. User clients therefore do not need to store any configuration information at all. All information needed to bootstrap the process of navigating a user's home directory is available on the USB dongle. Any additional user settings can be stored in the user's directory tree in the network.

5.3.3. *Share Roots*

How is the process of sharing files or directories through cryptographic links bootstrapped? For example, how does Fred know what files and directories Mary allows him to access? In this implementation we use a hash code of the combination of Mary and Fred's public keys to create the physical file name of a shared root directory. Both Mary and Fred have cryptographic links to that user directory and Mary is the owner. Each time Mary wants to share a user file or directory with Fred, she will add a pointer to that user file or directory into the shared root directory. Fred therefore automatically sees all items Mary shares with him.

All Mary and Fred need to access that directory is their own private and public keys and the other person's public key. The necessity of a central user database can therefore be avoided. For example, Mary knows her own private and public key from her own USB dongle and she knows Fred's public key from Fred's certificate which also lets Mary verify that she uses Fred's correct public key and not that of a man in the middle. If and how Fred's certificate is made known to Mary is an infrastructure question. In this prototype we will use a publicly available database of all user certificates. In a more general case, user certificates may not be published and the decision left up to the users themselves with whom they would like to share their certificates.

Note, that different directories will be used for files Mary shares with Fred than vice versa. This is to distinguish read and write privileges.

Also note that not all possible share root directories from all possible user pairs need to be created up front. Share root directories can be created on the fly, as needed.

5.3.4. *Back Links*

The cryptographic links shown in Figures 5 and 6 contain an encrypted back link to their parent directory. The link consists of the physical file name of the parent directory, encrypted with the directory owner's public key. This lets the owner of the directory navigate the directory tree backwards but not anybody else. This capability is needed, for example, if we want to display to the owner with whom he or she shares a certain directory.

5.3.5. *File System Efficiency*

Long term, the system may be made more efficient by implementing the encrypted file system natively instead of overlaying it on top of a regular file system. This would also allow for making the smallest encrypted unit a block on the disk instead of an entire file. The latter will allow for secure efficient revision control by allowing a node to store incremental changes to a document. However, the technology behind this is well understood and does therefore not need to be tested as part of this prototype.

5.4. *Parallel Data Read with Automatic Load Balancing*

Parallel data retrieval is done as follows: The access client requests the first b bytes from the first server in the list. Then, without waiting for the request to complete, it requests the following b bytes from the second server in the list. This continues until all servers are busy processing requests for this client.

As soon as the first server completes its request the client will request the next b bytes (not yet requested) from that server. This continues until requests for all data bytes have been issued.

While the client waits for the last requests to complete, it monitors the time each open request has been pending. If one or more requests have been pending for more than t_p seconds and free storage nodes are available (i.e., storage nodes without pending requests by this client), then the client will issue a second request for those blocks to free servers until either all servers are busy again or all data is retrieved.

5.5. Data Write

For a data write to complete, the data must be successfully written to at least two storage nodes. The other storage nodes will subsequently automatically synchronize without the access client being involved.

The minimum of two storage node requirements is to avoid a single point of failure (even though it might only be very short in duration). On the down side, this will make write operation likely twice as slow because they will likely be limited by the upload speed of the user's Internet connection. The automatic synchronization in the network avoids this process from becoming even slower and it allows write operations to complete even if not all storage nodes are on-line.

This raises one question: what happens if a user wants to read a document which has recently been changed and whose changes are not propagated to all storage nodes yet. The access client requesting the modification time of that document from all storage nodes that are online handles this. It will then issue data requests only to those nodes with the newest modification time. To guarantee equal modification time for all copies of a document it is being created by the first server which receives a new document or document version and then reused by all other storage nodes.

This still leaves the question of what happens in the unlikely event of all storage nodes with the latest document revision being off line at the time a user wants to access that document. In that case the user will read the old document version. It will feel to the end user as if the document was not yet updated.

Finally, we need to look into what happens if a user modifies the document again while all storage nodes with the latest version are off-line. This case will be handled automatically by use of the modification times with the effect that this latest write operation will overwrite the previously (but currently inaccessible) latest document version.

5.5.1. Transactional Nature of Write

The write process described above is of transactional nature. This means if a write succeeds for one storage node but the access client cannot write the document successfully to a second storage node, then the overall write operation fails and the changes written to the first storage node must be undone. Each receiving storage node will therefore lock the file (to be modified), queue the received changes, wait for confirmation from the access client, and then commit the changes and unlock the file. If the connection closes before confirmation is received, then the changes are discarded.

This, of course, still leaves us with a Byzantine Generals' Problem: What happens if the access client successfully completes a write operation to two storage nodes and just informed the first node of that fact. However, the network connection to the second node is no longer available and the second node therefore never receives confirmation. The first node will act as if the write operation succeeded while the second node will discard it. In this case, the access client will inform the user that the write operation succeeded, possibly with a warning that this happened with temporarily reduced data redundancy. The changes will then propagate from the single node to all other nodes, except if the single node incurs a catastrophic failure before it could transfer the changes to the other nodes.

5.6. Network Data Synchronization

After the access client succeeded in writing a document (or directory) to two storage nodes, those will enter that document into an update queue. Each document in that queue will maintain a list of other storage nodes that have already confirmed receipt of the new document. Once all storage nodes confirmed receipt, the document will be removed from the queue. If a receiving server reports a newer modification time stamp for that file, then, of course, it will not be transferred but that node will still be entered into the list of that file.

Therefore, for each new or modified document (or directory), there will be two storage nodes trying to propagate the modifications to all other nodes.

Each node will process the update queue as fast as possible but will never issue more than one request to another storage node at a time. Since most likely the network bandwidth or possible CPU resources or disk speed on the receiving storage node are the bottleneck, issuing multiple parallel requests would not make the process any faster, just increase the overhead for connection management on both sides.

If the document is locked by the receiving storage node (e.g., because another node is already sending the update), then the queue processor will try again later.

More sophisticated methods are conceivable where document update spreads out in a waterfall principle. In order to focus resources on the main short-term goals, such methods will not be implemented in this project.

5.7. End-to-End Security

The storage nodes do not possess any secret keys at all, not even temporarily. Data decryption can only take place on the user side by the client software. The storage nodes do not have a traditional access control nor authentication mechanism. If a user asks for a particular file by its physical (random) file name, the storage node will return that file without any further authentication checks. Access and sharing is controlled by means of cryptographic links, not by sharing secret keys. This allows a user to later revoke or modify another user's access rights without re-encrypting all files and redistributing keys.

Therefore, most parts of the system do not need to be secure or trusted, as pointed out in Figure 4). In particular, this is true for all components in the network, including hardware, software, networks, and protocols. Security is mostly concentrated in the USB dongle that is in physical possession of the user.

Furthermore, users do not need to login. In fact, there is no concept of login or logout. Storage nodes will find a user's home directory based on the public key provided by the user client. The public key is also stored on the USB dongle. Therefore, to be able to access their data, users need the USB dongle to be plugged in to their laptops but will not need to login at all. We expect this to be a nice tradeoff for the inconvenience of being forced to carry the USB dongle to all locations from where the storage needs to be accessed.

Finally, since all encryption and decryption processing is done on the client side, and since no user authentication needs to take place, the storage nodes are reduced to simple data input and output, similar to a simple web server. This requires very low CPU and memory overhead and makes this design therefore highly scalable and efficient and low cost to operate. Most of the computing is done on the client where it is much cheaper than on the server side and highly distributed.

5.8. No Single Point of Failure

The system has no single point of failure with one exception. This is through the fact that we are using the same server software for all storage nodes. If a security flaw were to exist in the server software (Net-Scale software or web server software) which lets an intruder break into a storage node, then a hacker could create a script which could break into all storage nodes in parallel and erase all stored data within seconds. The only way to avoid this single point of failure is by using software diversity for the server software, which could most effectively be achieved, by using software from different origins for different storage nodes. This is beyond the scope of this project, however.

5.9. Defense against Denial of Service Attacks

Denial of service attacks attempt to flood a server with requests thereby exhausting its resources and making it unavailable for service to legitimate users. Since the physical file names on our storage nodes have random file names that are only known to legitimate users who could successfully read the user data in the parent directory, any intruder would have to randomly guess file names. This makes it fairly simple for storage nodes to detect illegitimate users and initiate defense mechanisms, e.g., by temporarily blocking access from intruder's IP addresses. However, such techniques will not be implemented in this prototype.

6. Secure Distributed File System Design

6.1. Parallel Data Read Parameters

The initial values of the above two parameters will be:

Block size b : 64 Kbytes (65,536 bytes)

Wait time t_p : 2 seconds

The above values have been determined under the assumption that the data retrieve bandwidth for a storage node (which could be the cable modem upload speed) is approximately 250 Kbytes/sec.

Note, that this request allocation algorithm of the last phase, i.e., the wait phase, could be greatly improved. However, that would add complexity to a feature that will not be very important and will therefore not be done in this project.

6.2. Physical File Names

All (encrypted) user data is stored on a separate disk which is mounted to /sdb. That directory (i.e., the root of the data disk) has two subdirectories:

- ◆ /sdb/dat/: Contains all encrypted user data, i.e., permanent data.
- ◆ /sdb/que/: Contains the transaction queues, i.e., temporary data.

All directory and file names below are random byte sequences. To ensure that we don't create any illegal file names (e.g., with a '/' character in them), the random bytes are encoded in their two-character lower case hexadecimal representation before used as a physical file name.

Hexadecimal is not the densest representation possible, i.e., we are using a bit more characters for file names than necessary. However, the loss is not huge, the method is very simple, and we can save on programming, testing, compliance between C and Java, and debugging overhead which more complicated encodings would bring with them.

The physical file names and their directory structure are chosen at random and have no correlation with the file and directory names or the directory tree that they represent. In principle, we don't need a physical directory tree at all as directories are also represented by files, and all files could be stored in a single directory. However, traditional file systems tend to become inefficient if the number of files per directory grows beyond a few thousand.

6.2.1. File Name Spreading

From the client's point of view, each encrypted file will have a random file name. However, behind the scenes, the storage nodes will spread these files over multiple directory levels to avoid creating too large directories that would make most file systems inefficient. The spreading is done by applying a non-cryptographic hash function to the directory name. For selecting the optimal point between directory tree depth and number of subdirectories per directory we would need to understand the overhead involved for finding an item in a directory (we assume the OS (Operating System) performs a linear search in memory) and the overhead involved in following directory pointers on the disk (which assumes disk seek operations to the right blocks).

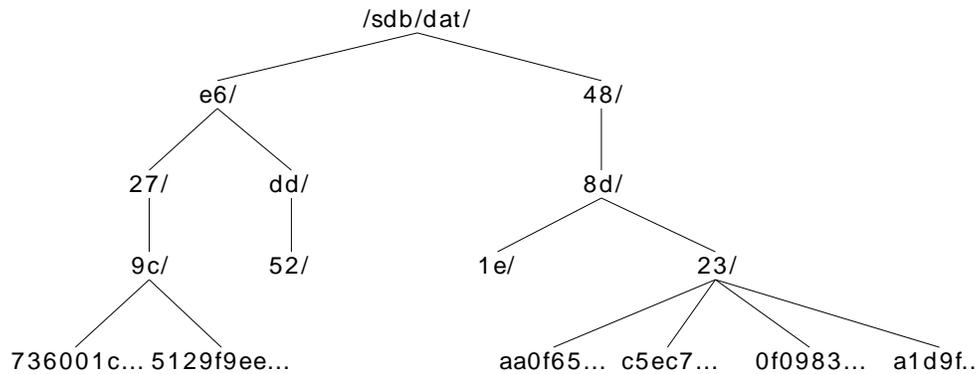


Figure 7. Example of a small physical directory tree that is used to store encrypted files which form the actual user directory trees. The user directory trees are not correlated to this physical directory tree at all. To conserve drawing space, one directory level is omitted and the file names are shortened.

For now, we selected 256 subdirectories and five subdirectory levels. This means each directory name consists of two hexadecimal characters and we can store a total of over 200 trillion (10¹²) files before the underlying file system becomes inefficient.

The function for file name spreading is called `nssdb_fname_to_path()`. Here is an example input and output:

- ◆ File name: a2c6e97f3d16231d
- ◆ Path: /sdb/dat/61b9/3c2f/c245/48ab/7e71/a2c6e97f3d16231d

The directory tree depth in this prototype is fix and consists of four (4) levels. We assume an average of 1,000 items per directory is efficient. This would then allow us to store 10¹² files before the system becomes inefficient. If we further assume an average of 100,000 files and directories per user, we could accommodate up to 10 million on a storage node. These settings

are more than sufficient for the present prototype system and can be changed at any time in the future to accommodate larger user communities. Note, that the actual capacity will be somewhat smaller as for practical reasons we chose 26, 676, and 1,000 instead of always 1,000 items per directory (see next section). This results in about 10^9 files or 10,000 users, which is still more than sufficient for the present prototype. Figure 7 shows an example tree.

6.3. Root File Names

Root file names are created through a cryptographic hash function from a user's public key. In case the root is shared between two users, then the hash function is applied to both users' public keys. The user who owns the node is hashed first, followed by the guest user.

The hash algorithm used is SHA-256, which generates an output of 64 bytes, which in turn will become a file name of 128 hexadecimal characters. Here's an example:

```
2c528bdd330c183656a944cab15b0d2f6d2a003a25736b51fdc74515f6c723b3
```

This may be more than we need. However, we need to ensure the chances of any collisions, i.e., two different users' public keys or key combinations generating the same hash code, is practically zero. Somebody with a deeper understanding of hash code collisions can probably guide us to a more optimal point but for now we stay on the side of caution.

6.4. Regular File Names

Regular file names are created with a cryptographic random function, i.e., the AES random key generator of OpenSSL. We are not sure yet if we need to take advantage of the cryptographic nature of the randomness yet or not. Cryptographic randomness allows us to assume that it will be nearly impossible for an adversary to guess a file name.

The length of the file name is always 40 characters. The first 8 are the lower case hexadecimal representation of the node ID. The other 32 characters are the lower case hexadecimal representation of 16 random bytes. The file names have no prefix or postfix (extender). The 16 random bytes create a name space of over 1038. Here is an example file name for node with ID 12:

0000000c745a1e30e24aec224325b2c57be6cb1a

7. Storage Node Design

7.1. Storage Node Hardware



Figure 8. A possible storage node hardware configuration: Dell Inspiron 520s slim line design, which does not take up a lot of space. The software is installed and configured by Net-Scale and no keyboard or screen is necessary for operation.

Each storage node consists of the exact same hardware configuration that is pre-in-stalled and preconfigured by Net-Scale. People who host a storage node will not have to install or configure any software.

The minimum technical specs are:

- ◆ State-of-the-art but not a high-end microprocessor, e.g., Intel Pentium dual-core, 1MB L2, 1.80GHz.
- ◆ 1GB memory.
- ◆ 100GB internal hard disk for OS and software.
- ◆ 1TB secondary internal hard disk for data storage.
- ◆ 100Mbit network card (no wireless LAN).
- ◆ 2 USB2 slots for optional future external storage extensions.
- ◆ Must be able to run Ubuntu 8.04.

The expected total hardware cost is \$1,000 or below. A possible configuration with Dell Inspiron 520s (slim line design) is shown in Figure 8.

7.2. Storage Node Software Configuration

- ◆ OS: Ubuntu 8.04 server
- ◆ srvnssdb (Net-Scale SDB daemon)
- ◆ DDNS client (not in use by the current prototype)
- ◆ Time synchronization (not in use by the current prototype). Time synchronization is important because time stamps are used for data synchronization between the storage nodes.

7.3. Transaction Queues for Data Integrity

Secure files are not modified directly in the data directory. Instead, modifications are first performed in a separate transaction queue directory and later applied to the data directory all at once. This is necessary for two reasons:

1. **Data Integrity.** A single client operation, such as creating a new file, typically translates into modifications of multiple secure files. This is particularly relevant when encryption keys are changed as those operations require all affected cryptographic links to be updated as well. If one secure file were to be modified but not the other affected ones (e.g., because the connection to the client was lost), then we could leave the secure tree in an unusable state. Its data could become inaccessible. In an encrypted file system that is the same as losing the data.
2. **Data Synchronization.** We need to be able to propagate modifications from one storage node to the others. Remember, that the server knows nothing about the relationships of the secure files nor does it have access to any secret keys. Therefore, what it has to do is keep track of the modifications applied to the secure files and faithfully repeat those in the other nodes.

7.3.1. *Transaction Queue Design*

◆ `nssdb_open_trans()` starts a new transaction and creates a subdirectory in `/sdb/que` which contains the process ID. We use the process ID such that later cleanup tools can later determine if a transaction queue is still alive or is a leftover of a killed process.

◆ A transaction queue contains three subdirectories to distinguish between adding a new file, modifying an existing file, and deleting a file:

- `add/`
- `modify/`
- `delete/`

◆ Once a transaction queue is created, all read and write operations first need to check if the file they want to open is in the transaction queue. In other words, the process that created the transaction will behave as if all modifications took effect immediately, while all other processes see the directory tree unchanged until the transaction is executed.

◆ Before entering a file into the transaction queue it is locked in the data directory. Other processes can still read the file but will not enter a file into their own transaction queue if it is locked.

◆ When executing a transaction queue, all files from the `/sdf/que/` directory are copied to their proper place in `/sdf/dat/` by using the `rename()` system call. This is the fastest way of executing the queue (to our knowledge) and the kernel guarantees that no single file is corrupted. However, if our own process dies during transaction queue execution, then the secure file system may still become corrupted. There is no way to prevent this. The best we can do is keep the queue execution as short as possible and as simple as possible (just a straight linear loop in our case) to eliminate (or minimize) chances of the process crashing or being killed by the system or due to power loss during transaction queue execution.

7.3.2. *What Gets Entered into the Transaction Queue?*

All files opened for writing or newly created files are entered into the transaction queue and locked in the data tree (unless they are new files). In addition, certain files opened for reading are also locked in the data tree. For example, if the application (libnssdf or jarnssdf) adds a new file they first load the parent directory, create the file, then update and write back the parent directory. The secure file corresponding to that parent directory needs to be locked between the read and write operations. Unfortunately, there is no way for libnssdb to know the intention of the caller. When opening a file for reading, the application therefore needs to tell libnssdb whether or not it needs the file locked.

7.3.3. *Some Design Consequences*

- ◆ Attempts to modify a section of the directory tree, which is already being modified, will fail. This avoids blocking users for lengthy time periods. Remember that a modification operation could be lengthy (e.g., several days when uploading a large file through a slow connection).
- ◆ When a process reads a section of the directory tree while another process modifies that section, it could be that the tree temporarily appears corrupted to the reading application. That can happen if the first application reads one node before the queue was executed and another node afterwards. The tree is not really corrupted, of course and preventing this would mean to block read operations during modifications and those can be lengthy as pointed out above.

7.3.4. *Headaches Caused by POSIX Lock Specification*

POSIX (Portable Operating System Interface) specifies that the kernel keep only one entry in the locking table per process and open file, even if that process opens the file multiple times. When closing a file, the lock gets removed, even if it was locked by a different file descriptor of the same process.

This caused some headaches in the following situation: Assume we need to keep a file locked across a read and a later write operation. The application will first read the file (open, read, close). Much later, it will write changes back (open, write, close). The solution sounds simple, open the file separately for the sole purpose of locking it. However, the close of the first read operation will remove that lock.

As a workaround, we use the file descriptor from opening the file for locking and will not close it until after execution of the transaction queue. This works fine. Just take care when working on the affected code as it needs to keep proper state across otherwise unrelated code pieces. Comments were added to all affected parts of the code, but of course, these dependencies are far from obvious or intuitive.

As another consequence of this POSIX lock specification, libnssdb becomes non re-entrant. This means we cannot have multiple sessions within the same process. For one, this is because one session locks a file and another session reads the same file, it would remove the first session's lock when closing the file. A second reason is that POSIX file locks don't work within the same process. If one session tries to lock a file, which is already locked by another session, then the lock should fail. However, if this is done within the same process, it does not fail and one session can therefore not detect what locks the other session has applied.

7.3.5. *Problem with Combining rename () and File Locking*

Our initial implementation was locking the original file then modifying a copy of that file in the `que/` directory. When done, it moved the file from the `que/` directory to the original place using the `rename ()` system call. This replaced the original file with the modified copy as an atomic operation. However, stress tests soon revealed file corruptions where one process could undo the modifications of another process.

The cause of the problem turned out to be the following: Opening a file and applying a lock to the opened file are two separate system calls and therefore not atomic. A process can open a file while it is still locked but before attempting to lock the file itself, the other process can call `rename()` which replaces the original file with a new one and then unlocks and closes the original (now removed) file. When the first process tries to apply a lock to the original file which is still open in that process, it will be successful and the process continues with the planned modifications but uses the old outdated file.

As a consequence of this, locks cannot be used for our purposes when applied to the same file we modify using `rename ()` calls. We therefore modified our implementation to use separate lock files that have the same name as the file they intend to protect but with an `".lck"` extension. The lock files are automatically created when used the first time and are automatically deleted when their corresponding data file is removed. Note, that we cannot remove lock files after each usage. This would create a similar race condition as the above: Two processes could both create their own lock files, one overwriting the others and both believing they have the file locked.

7.4. Domain Name and Server List Management

One problem that needs to be addressed stems from the fact that some storage nodes may be operated in environments without a static IP address, such as typical cable modem Internet connections at homes and in small businesses.

The prototype system will therefore use a Dynamic DNS (DDNS) service provider. DDNS services act like regular DNS as far as domain name lookup is concerned, except that they set the expiration dates of IP addresses to a very short interval, typically just a few minutes. This forces other DNS servers not to cache that information very long but instead query the root DNS (DDNS) server at almost every request. This ensures that clients do not use outdated IP addresses to connect to storage nodes.

In addition to this, DDNS servers allow the storage nodes to announce their current IP address on a regular basis, e.g., every few minutes, to the DDNS server. This is accomplished by installing a special piece of DDNS client code on each storage node.

Finally, this same setup can be used for the DDNS server to detect when a storage node is no longer online and automatically take it out of the list of available IP addresses (fail over functionality). An overview of this setup is shown in Figure 9

The same DDNS setup serves two additional purposes. First, it serves as a list of available storage nodes. While this prototype system will operate four storage nodes, that number is not hard coded in the clients or anywhere else in the system. Instead, the DDNS client of each storage node announces itself to the DDNS server, which keeps track of the available storage nodes at any given time. The DDNS server therefore makes a dynamic network database of available storage nodes unnecessary.

The second additional purpose of the DDNS setup is to allow the Java applet running on the users' browsers to connect to all available storage nodes in parallel. Normally, applets can only connect to the same server from where they were originally loaded. However, browsers enforce this by server name and domain, not by IP address. By using the same name and domain for each storage node (just different IP addresses), the Java applet can connect to all of them

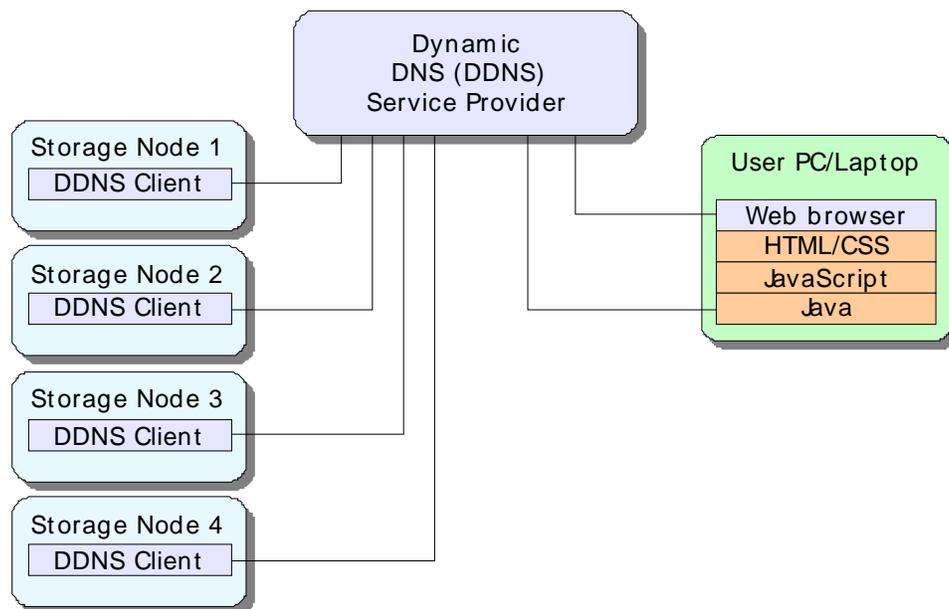


Figure 9. Network diagram of the dynamic DNS (DDNS) setup.

in parallel.

The steps involved by the user clients are as follows:

1. User enters the prototype system URL.
2. The browser does a DNS lookup.
3. The DDNS service provider returns in random order a list of available IP addresses, i.e., available storage nodes,

4. The browser selects the first IP address and requests the root document from there. Since all storage nodes run a web server with the same static content it does not matter to which node the user connects. Any one will return the same HTML, CSS, JavaScript, and Java code necessary to run the access client.
5. After receiving the web page (with HTML, CSS, JavaScript, and Java code), both the JavaScript code and the Java applet will start running automatically inside the browser (the Java applet will run in the background and not generate any visible output on the screen).
6. For each user interaction that requires communication with the storage nodes, the Java applet conducts a DNS lookup to receive the up to date list of IP addresses of the available storage nodes. Subsequently, the applet connects to all available nodes in parallel.

8. Access Client Design

The browsers listed in Figure 10 are in order of precedence. We will attempt to support as many browsers in the list as possible. Compromises may have to be made in the interest of time.

As is also pointed out in Figure 10, there are three main components on the client side to be developed: CSS, JavaScript and Java. Each of these components and their interfaces will be described next.

8.1. CSS

This module contains the cascade style sheets associated with the graphical interface. This includes modules positions and appearances in terms of colors, fonts, default graphics and layout.

8.2. JavaScript

This module takes care of HTML generation and of the entire user interface.

8.2.1. *Graphical Interface to user*

1. Generate panels with sliding bars
2. Progress bar
3. Accordion sections
4. Drag-and-drop
5. Long list handling

8.2.2. *Interface to Java*

1. function **getStorageNodesStatus**(/* No arguments */)

Description: Returns the number of Storage Nodes active and reachable.

Return value: Number of Storage Nodes that are reachable.

2. function **getUsersList**(/* No arguments */)

Description: Returns XML String with structure containing the list of all existing users with whom data can be shared.

Return value: XML String with users' information.

3. function **getFolderList**(

String fldLocation /* String indicating if the folder list is network, local or shared */

String parentFld /* String indicating the path to parent folder from where the first level of folders will be retrieved. "/" indicates root directory */

Int startFld /* Index indicating the first folder in list when a range of folders is retrieved */

Int length /* Number of folders in the list when a range of folders is retrieved */

)

Description: Returns XML string with structure containing folder list information at the specified level. It could be the network folder, the local folder or shared folder.

Return value: XML String with folder information.

4. function **getFileList**(

String fldLocation /* String indicating if the files are in network, local or shared */

String fldPath /* String indicating the folder path */

Int startFld /* Index indicating the first file in list when a range of files is retrieved */

Int length /* Number of files in list when a range of files is retrieved */

)

Description: Returns XML String with structure containing information about the list of files at the specified folder. It could be a network, local or shared folder.

Return value: XML String with file list information.

5. function **uploadFiles**(

Boolean updateFlag /* True (1) for update; False (0) for Overwrite */

String srcFldPath /* String indicating the folder path in local file system */

String destFldPath /* String indicating the folder path in the network */

String fileList /* List of filenames with ";" separated */

)

Description: This function operates initially synchronously, blocking the JavaScript. It generates a queue of files to upload. An estimate of the time required for upload will be provided to the user. In the background the Java module will proceed with the upload operation. A callback procedure from the Java process will unblock the JavaScript process when the operation is completed. Success Message will be displayed in the title of the logging section. In case of failure, an alert pop-up window will be displayed indicating the error and how many files were uploaded. User has the option to resume next time the network is re-established. During the waiting period, a progress bar will be displayed with the user having the option to cancel.

Return value: Estimated time for upload and then different messages depending on the state.

6. function **uploadFld**(

Boolean updateFlag /* True (1) for update; False (0) for Overwrite */

String srcFldPath /* String indicating the folder path in local file system */

String destFldPath /* String indicating the folder path in the network */

)

Description: Similar as before.

Return value: Estimated time for upload and queue list of files to be displayed by JavaScript, and then different messages depending on state.

7. function **downloadFiles**(

 Boolean updateFlag /* True (1) for update; False (0) for Overwrite */

 String srcFldPath /* String indicating the folder path in network file system, this could be a shared folder */

 String destFldPath /* String indicating the folder path in the local file system */

 String fileList /* List of filenames with ";" separated */

)

Description: This is a synchronous process. The list of files is given to the Java process, which acknowledges it and generates its own. When this is done, an estimate of the time to download is given to the user. The JavaScript process is blocked in this version, showing a progress bar with the option to cancel. In the background the download takes place. When the process is completed successfully, a Success message is displayed in the title of the logging area. If an error occurs, a pop-up window is displayed with the error message and the user can resume the next time the connection is re-established.

Return value: Estimated time for download operation to complete and then different messages depending on state.

8. function **downloadFld**(

 Boolean updateFlag /* True (1) for update; False (0) for Overwrite */

 String srcFldPath /* String indicating the folder path in network file system, this could be a shared folder */

 String destFldPath /* String indicating the folder path in the local file system */

)

Description: Similar as before.

Return value: Estimated time for download operation to complete and then different messages depending on state.

9. function deleteFiles(

String fldPath /* String indicating the folder path */

String fileList /* List of filenames with ";" separated */

)

Description: A list of files to be deleted in the network is passed to the Java process from the JavaScript. Once this list has been created by the Java process, an estimate of the time to delete is generated. A progress bar is displayed during this process execution with option to cancel. After successful completion, a success message is displayed in the title of the logging section. When an error occurs, a pop-up window is displayed and the process will resume depending on the error.

Return value: Estimated time to delete.

10. function deleteFld(

String fldPath /* String indicating the folder path */

)

Description: Similar as before.

Return value: Estimated time to delete.

11. function createFld(

String srcFldPath /* String indicating the path of the source folder */

String fldName /* String indicating the folder name of the new folder */

)

Description: This operation creates a new folder in the network underneath the source

folder with the folder name string.

Result value: Boolean of success or failure. If failure error message.

12. function `renameFld()`

String `fldPath` /* String indicating the whole path of the folder to rename */

String `newFldName` /* String indicating the folder name of the new folder */

)

Description: This operation renames an existing folder in the network.

Result value: Boolean of success or failure. If failure error message.

13. function `changeStatusShareFld()`

Boolean `addFlag` /* true for adding read permission; false for removing read permission */

String `fldPath` /* String indicating the whole path of the folder in the network which will be shared */

String `usrSharing` /* String indicating the user with whom this folder will be shared */

)

Description: This operation add or remove sharing permission to an existing folder in the network with a system user.

Result value: Boolean of success or failure. If failure error message.

In subsequent release, the following functions to operate in the network will be added:

14. function `copyFiles()`

String `srcFldPath` /* String indicating the source folder path */

String `destFldPath` /* String indicating the destination folder path */

String `fileList` /* List of filenames with ";" separated */

)

Description: A list of files will be copied from source folder to destination folder only in the network. This is not an operation that can be done on the shared folders. This list is passed to the Java process from the JavaScript. Once this list has been created by the Java process estimates of the time to copy is generated and sends to the user. This is a blocking operation. During that time a progress bar is displayed. After successful completion, a success message is displayed in the title of the logging section. When an error occurs, a pop-up window is displayed and the process will resume depending on the error.

Return value: Estimated time to copy.

15. function **copyFld**(

String srcFldPath /* String indicating the source folder path */

String destFldPath /* String indicating the destination folder path */

)

Description: Similar as before.

Return value: Estimated time to copy.

16. function **moveFiles**(

String srcFldPath /* String indicating the source folder path */

String destFldPath /* String indicating the destination folder path */

String fileList /* List of filenames with ";" separated */

)

Description: Similar as before.

Return value: Estimated time to move.

17. function **moveFld**(

String srcFldPath /* String indicating the source folder path */

String destFldPath /* String indicating the destination folder path */

)

Description: Similar as before.

Return value: Estimated time to move.

8.3. Java

This module is the heart of the client. It communicates with JavaScript to take care of user interactions and information display, as well as all the communication and data management from and to the Storage Nodes.

In addition, this module performs all the cryptographic operations to secure and decipher user's information.

9. Storage Node and Client Integration

9.1. Overview

Figure 10 shows an overview of storage node and access client integration and also the integration of a storage node with the other storage nodes.

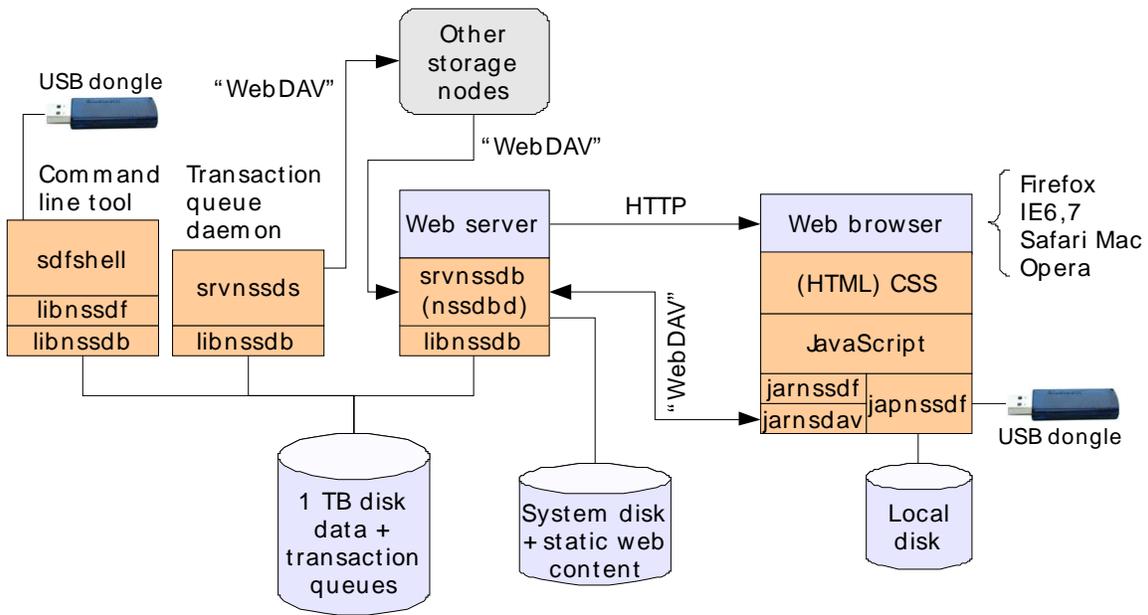


Figure 10. Integration of a storage node and access client and integration of a storage node with the other storage nodes. HTTP is used only for starting the web applications. It sends the web content, JavaScript, and Java code to the browser. WebDAV is used for all storage data interactions between the access client and the storage node(s) and for the interactions between the storage nodes.

9.2. Network File Server Functions

The network file server is almost like a regular file server because most of the functionality of implementing the encrypted file system is performed on the client (as is necessary for maintaining end-to-end security). What the file server is concerned, it stores regular files, they just happen to have random file names. There are a few particular tasks it needs to do beyond being a standard file server:

- ◆ **Generate a random file name.**
- ◆ **Check user certificate** to enforce the rule that items (files and directories) can only be modified by their owners, and to ensure only valid repository users can store data. This is not necessary for data integrity but it is necessary for protection against data loss and flooding of the storage node with junk data.
- ◆ **Receive a transaction queue.**
- ◆ **Execute transaction queue locally.** First, lock all affected files, then carry out all modifications on the queue and unlock the files.
- ◆ **Return a requested file.**
- ◆ **Synchronize transaction queue** with other storage nodes. Contact other nodes, and submit the transaction queue. Do this until all other storage nodes confirmed successful execution of the transaction queue on their side.

All items, except for the last, are carried out in reaction to a client request. The last one is carried out by an independently scheduled process.

9.3. Storage Node Access Protocol (WebDAV)

For communication between storage nodes and web clients (Java code) we use a proprietary protocol that is closely related to WebDAV/HTTP. This does not mean that our system works with other WebDAV clients. That would be impossible because WebDAV has no notion of our encrypted file system. However, whenever possible, we use the exact specifications provided by WebDAV and for concepts which are not known to WebDAV (e.g., transaction queues), we use proprietary WebDAV extensions. In summary, our proprietary protocol is a small subset of WebDAV with some proprietary extensions.

9.4. Design Decision: Apache mod_dav or Net-Scale Daemon

During early design discussions the mod_dav module would be used with Apache. This provides a simple interface for many of the functions required by the client. After some initial work apache2 was eliminated for the following reasons.

1. **Security.** The Apache module interface is complex, although it appears to be well understood by the Apache developer community. A module is inserted in a series of default handlers, which cannot be removed or bypassed. This is a potential security weakness, as rogue code could possibly be inserted via this chain - it becomes necessary to fully evaluate and understand Apache.
2. **Request vs. Transaction.** A connection is established with the web server for each request then the connection is closed. SDF requires transactions, which do have state; ideally the connection is not closed until the transaction is complete. Although there may be workarounds for this, they would require modification of the core apache code (never a good idea).

3. **Child Process vs. Threads.** The Net-Scale daemon spawns a child process for each connection request. The process terminates when the connection is closed. Any bugs within the code are isolated and do not propagate or affect the runtime environment. In Apache, connection requests are normally handled by threads; in the event of abnormal termination these threads may or may be cleaned up. The impact on the runtime environment, especially over time, is unknown.

4. **Subset of WebDAV Functionality.** Much of the WebDAV functionality is not required (and will not be used) by SDF. It is easier to add this functionality to the daemon than to worry about what is not being used.

9.5. Protocol Definition

9.5.1. Overview

- ◆ GET Method
- ◆ PUT Method
- ◆ COPY Method
- ◆ DELETE Method
- ◆ IHLOAD Method
- ◆ IHSAVE Method
- ◆ OPENTRANS Method
- ◆ CLOSETRANS Method

9.5.2. GET Method

The GET method is for reading a secure file. If <lock-value> is "yes" (see below), then it can only be called after an OPENTRANS request.

GET Request Syntax and Header Fields

- GET /<file-name>
- Lock: <lock-value>

The file name has to start with a '/' but it cannot contain any other '/' characters. That's because an SDB storage node has no concept of paths. It only knows file names (directory spreading and pre-pending the root paths is done by the node automatically and are transparent at this protocol level). Any text following "GET /<file-name>", e.g., "HTTP/1.1" is ignored.

If the <lock-value> header field is "yes", then the file is locked.

Any additional header fields and any content are ignored.

GET Response Header Fields

Upon success, GET returns the following HTTP header fields:

- Content-type: application/octet-stream
- Content-length: <length>
- Content-location: /<file-name>
- Last-modified: 0

<length> indicates the content length in bytes. <file-name> is the name of the file returned. It is the same as specified in the request. The Last-modified field is currently not supported and its value is always 0.

GET Response Status Codes

- 200: OK.
- 400: Bad request, e.g., file name (path) missing or not starting with a '/'.
- 404: File not found.
- 423: File is locked by another request.
- 451: Wrong request hierarchy state.
- 500: Internal server error.
- Socket closed by host: IP connection lost or i/o error while reading file.

9.5.3. *PUT Method*

The PUT method is for creating a new or modifying an existing secure file. It can only be called after an OPENTRANS request.

PUT Request Syntax and Header Fields

- PUT /<file-name>
- Content-length: <length>

If a file name is specified, it has to start with a '/'. If no file name is specified, a single '/' character must be sent instead. In that case, a new random file name will be created and returned. Any text following "PUT /<file-name>", e.g., "HTTP/1.1" is ignored.

The <length> header field must specify the data content in number of bytes. Any additional header fields are ignored.

PUT Response Header Fields

- Content-location: /<file-name>
- Last-modified: 0

<file-name> is the name of the file written. If the client specified a file name in the request, this is the same. If not, it is the new random file name that was created. The Last-modified field is currently not supported and its value is always 0.

PUT Response Status Codes

- 200: OK.
- 400: Bad request, e.g., file name (path) not starting with a '/'.
- 423: File is locked by another request.
- 451: Wrong request hierarchy state.
- 500: Internal server error.
- Socket closed by host: IP connection lost or i/o error while writing file.

9.5.4. *COPY Method*

The COPY method is for making a copy of a secure file in the network without sending the file content to the client and back. It can only be called after an OPENTRANS request.

COPY Request Syntax and Header Fields

- COPY /<src-file-name> /<dest-file-name>

If either file name is specified, it has to start with a '/'. If no <dest-file-name> is specified, a single '/' character must be sent instead. In that case, a new random file name will be created and returned. Any text following the destination file name is ignored. Also, any additional header fields and any content are ignored.

COPY Response Header Fields

- Content-location: /<dest-file-name>
- Last-modified: 0

<dest-file-name> is the name of the destination file written. If the client specified a file name in the request, this is the same. If not, it is the new random file name that was created. The Last-modified field is currently not supported and its value is always 0.

COPY Response Status Codes

- 200: OK.
- 400: Bad request, e.g., file name (path) not starting with a '/'.
- 404: Source file not found.
- 423: Destination file is locked by another request.
- 451: Wrong request hierarchy state.
- 500: Internal server error.

9.5.5. *DELETE Method*

The DELETE method is for removing a secure file. It can only be called after an OPENTRANS request.

DELETE Request Syntax and Header Fields

- DELETE /<file-name>

<file-name> must specify the secure file name. It must be prefixed with a '/'.

Any additional header fields and any content are ignored.

DELETE Response Header Fields

DELETE returns no additional header fields.

DELETE Response Status Codes

- 200: OK.
- 400: Bad request, e.g., file name (path) not starting with a '/'.
- 404: File not found.
- 423: File is locked by another request.
- 451: Wrong request hierarchy state.
- 500: Internal server error.

9.5.6. *IHLOAD Method*

The IHLOAD method is for reading the item header of a secure file. If <lock-value> is "yes" (see below), then it can only be called after an OPENTRANS request.

IHLOAD Request Syntax and Header Fields

- IHLOAD /<file-name>
- Lock: <lock-value>

The file name has to start with a '/' but it cannot contain any other '/'. Any text following "GET /<file-name>", e.g., "HTTP/1.1" is ignored.

If the <lock-value> header field is "yes", then the file is locked.

Any additional header fields and any content are ignored.

IHLOAD Response Header Fields

Upon success, IHLOAD returns the following HTTP header fields:

- Content-type: application/octet-stream
- Content-length: <length>
- Content-location: /<file-name>
- Last-modified: 0
- File-size: <file-size>

<length> indicates the content length (item header size) in bytes. <file-name> is the name of the file returned. It is the same as specified in the request. The Last-modified field is currently not supported and its value is always 0. <file-size> is the total size of the secure file in bytes.

IHLOAD Response Status Codes

- 200: OK.
- 400: Bad request, e.g., file name (path) missing or not starting with a '/'.
- 404: File not found.
- 423: File is locked by another request.
- 451: Wrong request hierarchy state.
- 500: Internal server error.
- 551: Wrong secure file format.
- Socket closed by host: IP connection lost or i/o error while writing data.

9.5.7. *IHSAVE Method*

The IHSAVE method is for modifying the item header of a secure file. It can only be called after an OPENTRANS request.

IHSAVE Request Syntax and Header Fields

- IHSAVE /<file-name>
- Content-length: <length>

Name of the secure file preceded by a '/'. Any text following "PUT /<file-name>", e.g., "HTTP/1.1" is ignored.

The <length> header field must specify the item header size in bytes. Any additional header fields are ignored.

IHSAVE Response Header Fields

- Content-location: /<file-name>
- Last-modified: 0

<file-name> is the name of the file written. It is the same name the client specified in the request. The Last-modified field is currently not supported and its value is always 0.

IHSAVE Response Status Codes

- 200: OK.
- 400: Bad request, e.g., file name (path) not starting with a '/'.
- 404: Source file not found.
- 423: File is locked by another request.
- 451: Wrong request hierarchy state.
- 500: Internal server error.
- 551: Wrong secure file format.
- Socket closed by host: IP connection lost or i/o error while writing file.

9.5.8. *OPENTRANS Method*

The OPENTRANS method is for starting a new transaction.

OPENTRANS Request Syntax and Header Fields

- OPENTRANS

Any text following "OPENTRANS ", e.g., "HTTP/1.1" is ignored. Any additional header fields and any content are ignored.

OPENTRANS Response Header Fields

OPENTRANS does not return any additional header fields.

OPENTRANS Response Status Codes

- 200: OK.
- 400: Bad request.
- 451: Wrong request hierarchy state.
- 500: Internal server error.

9.5.9. *CLOSETRANS Method*

The CLOSETRANS method is for completing a new transaction.

CLOSETRANS Request Syntax and Header Fields

- CLOSETRANS
- Transaction-queue: <execute-flag>

If a "Transaction-queue" header field is specified and its value is "execute", then the transaction queue is executed. Otherwise the queue is discarded. Any text following "CLOSETRANS ", e.g., "HTTP/1.1" is ignored. Any additional header fields and any content are ignored.

CLOSETRANS Response Header Fields

CLOSETRANS does not return any additional header fields.

CLOSETRANS Response Status Codes

- 200: OK.
- 400: Bad request.
- 451: Wrong request hierarchy state.
- 500: Internal server error.

9.6. Design Compromises and Deferred Features

We decided to make the following design compromises and defer the following features in order to minimize time to completion of the first SDF prototype.

9.6.1. *Changing the Locks (Access Revocation)*

The function `chlocks ()` changes the random item keys of the specified directory and all its sub-directories. A typical use of this function is to call it right after one or more read access privileges have been removed. This ensures that users whose read access was revoked can no longer read the directory tree, even if they secretly made copies of the random encryption keys.

While `chlocks ()` changes the keys of all directories it leaves the keys of files unchanged. The necessary re-encryption could be very lengthy as the entire file needs to be sent to the client and back. Instead, files are automatically re-encrypted each time the client modifies them. At that point there is no extra cost for re-encryption. Furthermore, immediate re-encryption would not really buy more security, as a user whose read access was revoked could easily have made a copy of the file beforehand. For regular users that's much easier to do than secretly keeping copies of the random keys (the latter requires code hacking).

Furthermore, the entire specified directory sub tree is updated in a single transaction. This is for code simplicity only. As a consequence, there is a limit on the directory sub tree size depending on the available client and server memory. The server's bottleneck is most likely the number of file descriptors that need to be kept open because of the locks applied to them. This limit should be in the hundreds of thousands of nodes, which is certainly sufficient for a proof of concept prototype and a limited pilot. Eventually, we will reach this limit and need to modify this function to break down the updating of larger directory sub trees into multiple transactions.

See the outdated [Access Revocation](#) page for more information.

9.6.2. *File Date and Time Stamps*

To make date and time stamps useful we need to use the client to create the time and store it in the SDF headers. We cannot use the time stamps of the storage node file systems for a number of reasons:

- ◆ They cannot be used for synchronization (update mode) because they would not let us decide whether or not a file provided by the client is newer or older than an existing one (time zone and clock settings may be different).
- ◆ The secure storage node files can be updated for housekeeping purposes (e.g., updating a cryptographic link) without changing the actual content. The storage node modify time is therefore different from what the user would consider the modify time.

As this is a low risk item we defer it to after our first prototype completion.

9.6.3. *Directories in Memory*

During encryption and decryption the client will keep at least two copies of an entire directory in memory (one encrypted the other in clear). This limits the maximum practical directory size (but not its content size) to a few hundred MB (or a few hundred thousand entries) depending on the available memory on the client computer.

Note, that the API of libnssdf and jarnssdf supports block wise processing and random block access for directories. Therefore, the client code itself does not have this limitation but the internals of libnssdf and jarnssdf do.

Because the encryption algorithms we use (stream ciphers) do not allow for random block access (if they did, it would be likely a security hole) the solution will be to store directories in multiple chunks on the storage node, e.g., of 1 MB each. Each chunk will have its own secret random key and cryptographic links.

9.6.4. Root Directories cannot be shared

Users cannot share their root directories with other users. They can share any file or directory underneath but not the root itself. The reasons for this are a) to reduce the development and testing effort a bit and b) we feel it probably does not make sense to share an entire root directory. However, this functionality can be added very easily if need be.

9.6.5. Cryptographic Link ID Rollovers

Cryptographic link IDs start at 1 and are incremented each time a new link is created. Once the variable, which holds the ID rolls over, uniqueness is no longer guaranteed. In case of Java, an out of bound error will be created in case of C the rollover will take place silently. This will happen after approximately 2 billion cryptographic links. In practice, this could only happen for items whose cryptographic links are constantly added and removed again. For the current demo and pilot prototype we therefore do not yet address this case.

10. Decision Guidelines

This section outlines the guiding principles we used for making decisions throughout the project. The number one guiding principle is the main project goal (Section 1.1.): “Demonstrate that a robust distributed storage network with end-to-end security and a good user experience is feasible by designing, implementing, and deploying a prototype system with multiple storage nodes and a zero footprint web access client. ”

Other guiding principles are organized in critical items, important items, and less important items.

10.1. Critical Items

These items are critical to the project and cannot be compromised.

- ◆ **Deadline.** The prototype system must be ready for usage by end users on September 1, 2008.
- ◆ **End-to-end security.** The only way to read unauthorized documents is by compromising the private key of a user with read access to that document. The only way to create or alter a document undetected is by compromising the private key of a user who has write access to that document or directory.
- ◆ **No single point of failure.** The only exception is the server software (see Section Error! Reference source not found.).
- ◆ **Zero footprint web access.** No client side software installation needed and no browser requirements other than HTML, CSS, JavaScript, and Java. For example, no Flash or other plug-in requirements.
- ◆ **Usability.** Any feature or functionality must be usable by an average user without special security education.

◆ **Secure sharing.** Users must be able to give other users secure read access to their documents without sharing any secret encryption keys.

◆ **Scalability.** The principles used in this prototype system must be scalable to consumer scale (millions of users). The software we implement should have the same scalability properties but if need be we can defer a fully scalable implementation until later.

◆ **Data integrity.** If a document is accepted by a storage node, then the user must be guaranteed that it was uploaded correctly and without any network errors. Likewise, if a document is downloaded without an error message, then users must be guaranteed that it was transferred and signed correctly.

10.2. Important Items

These items are important and should not be compromised if it can be avoided.

◆ **Intuitive user experience.** Wherever possible, the system should be intuitive for average end users without requiring them to study manuals or get training.

◆ **Convenience for storage node operators.** This includes equipment size, noise generated, power consumption, number of cables needed, and time to setup.

◆ **UI scalability.** The user interface should scale to millions of documents and should, in particular, not have the same limitations with long lists as the current Ajax access client on net-arc.net has.

◆ **Low maintenance.** The system should be essentially maintenance free. If a storage node has a problem, it should be easy to detect and replace the hardware. Also, system monitoring should be automatic and easy to observe for humans.

10.3. Less Important Items

These items are less important and can be compromised in support of critical and important items.

- ◆ **Server software portability.** The Net-Scale server software should be as portable as possible. However, in the interest of time we will avoid testing on multiple platforms.
- ◆ **Access control user interface.** While users must be able to control that has access to their documents, the user interface does not necessarily need to be able to deal with a large user base. For example, a simple linear list of all users would be acceptable for now, versus a more sophisticated user lookup implementation combined with customizable quick lists.

References

- [1] John Kubiawicz et al. OceanStore: An Architecture for Global-Scale Persistent Storage. 2000. <http://oceanstore.cs.berkeley.edu/publications/papers/pdf/asplos00.pdf>
- [2] Hakim Weatherspoon, Patrick Eaton, Byung-Gon Chun, and John Kubiawicz. Antiquity: Exploiting a Secure Log for Wide-Area Distributed Storage. 2007. <http://oceanstore.cs.berkeley.edu/publications/papers/pdf/antiquity06.pdf>
- [3] Dominik Grolimund, Luzius Meisser, Stefan Schmid, and Roger Wattenhofer. Cryptree: A Folder Tree Structure for Cryptographic File Systems. 2006. <http://dcg.ethz.ch/publications/srds06.pdf>
- [4] Eu-Jin Goh, Secure Indexes, Stanford University, CA. 2004. <http://eujingoh.com/papers/secureindex/index.html>.
- [5] Mikkhail J. Atallah, Keith B. Frikken, and Marina Blanton. Dynamic and Efficient Key management for Access Hierarchies. 2005. Department of Computer Science, Purdue University. <http://www.cse.nd.edu/~mblanton/papers/ccs05.pdf>

List of Acronyms

AES	Advanced Encryption Standard
Ajax	Asynchronous JavaScript and XML
CAC	Common Access Card
CFB	Cipher Feed-Back
CL	Cryptographic Link
CPU	Central Processing Unit
CSS	Cascading Style Sheets
DARPA	Defense Advanced Research Projects Agency
DES	Digital Encryption Standard
DNS	Domain Name Service
DDNS	Dynamic Domain Name Service
FIPS	Federal Information Processing Standard
HTML	Hypertext Markup Language
HTTP	Hypertext Transport Protocol
HTTPS	Hypertext Transport Protocol over transport layer security
IP	Internet Protocol
IV	Initialization Vector
JCE	Java Cryptography Extension
MD5	Message-Digest Algorithm 5
NIST	National Institute for Standards and Technology

OpenSSL	A popular open source library for writing SSL applications
OS	Operating System
PC	Personal Computer
PKCS	Public-Key Cryptography Standards
POSIX	Portable Operating System Interface
SDF	Secure Distributed File System
SHA	Secure Hash Algorithm
RSA	The Security Division of EMC, also the name of their public key encryption algorithm
SSL	Secure Socket Layer
URL	Uniform Resource Locator
USB	Universal Serial Bus
WebDAV	Web Distributed Authoring and Versioning