UNIVERSITY OF CALIFORNIA
SANTA CRUZ

RINGER: DISTRIBUTED NAMING ON A GLOBAL SCALE

A project submitted in partial satisfaction of the
requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER SCIENCE

by

Ian Gerald Pye

December 2008

The Project of Ian Gerald Pye
is approved:

Scott Brandt

Carlos Maltzan

Lisa C. Sloan
Vice Provost and Dean of Graduate Studies

Abstract

Ringer: Distributed Naming on a Global Scale

by

Ian Gerald Pye

This project describes an attempt to bridge the divide between immutable web-based content and traditional read-write distributed filesystems. The system we have developed (dubbed Ringer) provides a completely distributed service which comprises both a wide area read-write filesystem and a searchable content index.

Ringer fills the gap between distributed filesystems and the Internet. The Internet, except in specialized environments like wikis, is an extremely difficult medium for users to generate collaborative content. Almost all files are immutable except to their owners, and when multiple authors are allowed it is up to the application (Mediawiki, Google Docs for example) to support this. Current distributed filesystems either do not scale, or make trade-offs between distributed search and file mutability. Ringer facilitates mutability of files at a filesystem level, freeing applications running above Ringer to focus on their core tasks. Since there is no reliance on content referencing other content, Ringer is able to effectively index and rank data in more general formats than pure HTML.

Ringer works by separating nodes into two classes. Client nodes host files and transfer data directly between each other in a peer-to-peer manor. All metadata and search operations are sent to an overlay graph of specialized metadata nodes. Access and concurrency control is maintained via these metadata nodes. The metadata nodes are arranged into an arbitrary

graph, which can be constructed so that no single node, or small group of nodes, is a choke point. Metadata nodes support two types of search: a tagging-based distributed index and a completely automated similarity hash search.

Chapter 1

Introduction

There is currently no good way to organize important mutable information on a global scale. This paper presents a solution to this problem. The information which we consider in this paper is found in documents which lack the highly connected nature of hypertext, where pages routinely reference many other pages. These documents may be spread over diverse locations, without a centralized database organizing them all. Multiple writers and multiple readers all expect good consistency. Information may not be repeated across multiple documents: some data only exists in one particular document, and this one document may be constantly edited. We present a system which enables distributed finding of one correct document in a sea of false leads, and controlled editing of this document.

As a motivating example, consider the United Nations. The UN has offices all over the world, each of which is dedicated to producing information. This information is encoded in a wide variety of document formats, most of which do not reference other documents. These documents have to be available on demand to anyone who asks, but a hot-cold distribution of

interest is expected where most documents are seldom read while a few are frequently accessed. Furthermore, while documents should be stored redundantly, they are not immutable; writes as well as reads must be expected. And of course, all this has to be done on a budget. Financial constraints combined with a large volume of documents make storing all information centrally (with local cached copies for fast access) untenable.

The problem we have solved is twofold: first finding information and then accessing and updating that information. Ringer upholds concurrency, while supporting writes as well as reads. We have implemented security, privacy and user privilege systems. Ringer also provides reasonable performance over low bandwidth connections.

Our solution has three features:

1. **Filesystem Semantics**: Concurrency control and close-to-open consistency are essential to keep things manageable in a distributed system with many readers and writers.

2. **Database-Style Indexing**: We need the ability to search on arbitrary file attributes and get good results quickly. Additionally, Ringer's approach should work well even when information is found in only a few sources.

3. **Internet-Style Connectivity**: Files need to be available on demand, but only downloaded (a slow operation on a low bandwidth network) as needed. Most files are expected to be accessed rarely.

A naive approach to global document organization might be to simply rely on email to spread documents. This has the benefit of being easy to deploy, low cost and a process which (most) everyone understands. However, with this approach there is no way to discover what

documents are out there, only the local collection is visible to a user. As the number of files scales, this approach falls apart.

Another possible solution is to give each office a web-server and access to a VPN, and throw a search appliance on the resulting network. However, because of the many formats which documents can take, the expected lack of references and the lack of redundancy, this approach does not work well [6].

A third solution might be a decentralized peer to peer file-sharing network, in the manor of Frangipani [15]. This fails because of the search requirement. It is not enough to be able to read a document; users have to be able to find the document first.

The common thread here is that search is hard, especially when combined with concurrency and stability.

The Ringer system works by adding a metadata superstructure onto a decentralized P2P network, separating out the handling of data and metadata. Clients transfer data directly between each other in a traditional P2P manner. Clients find each other via a graph of metadata servers (MDS) organized in parent/child relationships. The MDS graph can be arbitrarily organized. In general, parents know what files their children have, but not more than that. To facilitate search, we have explored topologies where MDSs are grouped into sparsely connected clusters or rings (hence the name Ringer) connected by a few, or even one, top level MDS. Ringer differs from other P2P systems such as Ivy [10] since while there is no single MDS, there is also not a complete lack of centrality. Our hybrid approach simplifies implementation and search while avoiding the bottlenecks caused by a single central MDS.

Chapter 2

A Distributed Filesystem

Similar to the Ceph object based filesystem [16], Ringer separates out the handling of data and metadata through node specialization. This is in sharp contrast to most P2P networks which use uniform nodes. Ceph differentiates data and metadata to allow for scalable numbers of parallel metadata operations, and also to support a pseudo-random object placement scheme (CRUSH). Conversely, Ringer uses node specialization to facilitate search rather than performance. Virtual metadata nodes add structure to the Ringer topology while creating a secondary connectivity graph, allowing clients to find one another quickly.

We first focus on the client filesystem, followed by the metadata servers (MDS). We then show how these pieces interact to produce coherent reads and writes.

2.0.1 Client

The Ringer client is implemented via the FUSE library as a mountable local volume. Data is stored in a shadow directory which is unique to each user. Each client connects to

(registers with) only one MDS. This MDS is known as the "owning" MDS and is passed via a configuration file or command line parameter to the client at initialization time. All metadata lookups are sent to this single MDS. In practice, this involves translating from the native Unix "struct stat" structure to Ringer's metadata objects (dubbed "rnodes") and vice-versa. Data is transfered directly between clients in blocks. Currently a 4K block-size, the native block-size of FUSE, is used. Clients are identified by a client id, consisting of a host and port pair, which is unique to each user. Therefore, more than one Ringer volume may be simultaneously mounted on a single physical system.

Files are owned by only the one client which created that file. Ownership cannot be transferred. A client is responsible for registering newly created files with that client's MDS.

Ringer supports both a synchronous and an asynchronous communication protocol for data transfer. A client may contact another client and ask for a block of data from a file. The requesting client blocks until the data is sent back. Alternatively, a client *or MDS* may contact a client and request that a block of data be sent to a specified host. The requested block is sent to the noted recipient and handled by a separate callback function. The recipient may be the client making the request or a third party. For example, a MDS may realize that a client's cached data is invalid and send a request to update the data. In this form, the requested block is sent to the noted recipient and handled by a separate callback function.

2.0.2 Metadata Server

The MDS maintains a list of registered clients, and can connect to an unlimited number of parent MDSs as a client MDS via the same protocol clients use to connect to their owning

MDS. Ancestor MDSs know which files (via an in memory hash table of Ringer paths) their descendent MDSs own, but do not store more detailed file metadata. Since MDSs can have arbitrarily many parents, the MDS graph for a particular network can be as complete or sparse as desired.

When a client owns a file, the client's MDS owns the rnode for that file

2.0.3 Single Namespace

Ringer paths are globally unique for each file. They are formed by appending the owning client's client id to the file path. Some path mangling is required to translate ringer paths to locally understandable paths, but we feel that the simplicity of unique paths outweighs the additional translation logic.

This global namespace can be traversed as follows: In the root of each Ringer volume, a .hosts directory exists. The .hosts directory contains a set of directories corresponding to all of the clients connected to the local client's owning MDS, the local client's owning MDS's children and the local client's owning MDS's parents. In the case of a binary tree structured MDS graph, there would be clients from at most 4 MDSs in one .hosts file. Each client is mounted inside its directory. Note that a path of the form `.hosts/client_id1/.hosts/client_id2/foo` is the same to Ringer as `client_id2/foo`. Long path names are simplified automatically. Clients further away in the topology (for example, associated with a grandparent MDS) can be accessed via a chain of intermediate clients.

2.0.4 Reads

Close-to-open consistency is maintained by a system of leases, arbitrated by the MDS which owns the rnode for the file under consideration.

When a client wishes to read a file, it must request a read lease from the MDS which owns the rnode of that file. The client initially sends the request to the owning MDS of that client, which then finds the correct MDS by successively checking with parent MDSs until some ancestor recognizes the file. This ancestor MDS forwards the request to the MDS which actually owns the file. The owning MDS checks file permissions, decides whether or not to grant the lease and passes back permission (or denial) down the chain of intermediate MDSs. Read leases are always granted to clients with read permission, except when another client has a current write lease for the file. Note that leases operate on the file (not block) level. If a read lease is not granted, the client may try again after a few moments.

To eliminates the lookup overhead when a file is repeatedly accessed, each client maintains a cache of known filenames and their associated MDSs.

If granted a read, the requesting client first downloads the rnode for that file. Rnodes maintain a "last modified" date for the associated file; there is no caching of rnodes. The requesting client then checks to see if the last modified date matches the date for any locally cached relevant blocks. If the last modified dates differ or the file has not yet been read, the requesting client contacts the owning client directly and downloads the requested blocks. The owning client's IP address and port are listed in the rnode for the file.

Leases do not time out quickly; they should be explicitly terminated by the lease-

holder. Whenever the client's `read()` system call ends, the `endRead()` function is invoked. This function contacts the owning metadata server for the file, and requests lease termination. The call is asynchronous, returning without waiting for a response from the MDS.

One performance improvement would be maintaining read leases beyond one `read()` call. This optimization would not change the overall Ringer protocol however.

2.0.5 Writes

Writes follow the same protocol as reads, with one important difference. Only one client can have the lease for a file at any given time, if that lease is a write lease. When a write lease is requested, the MDS only grants the lease if there are no outstanding read leases for the file. Because leases are short lived, clients busy-wait until a write lease is granted. The duration of the write lease is the duration of the client's `write()` system call.

2.0.6 Failures

Because of the distributed nature of Ringer, network or system failures can be handled gracefully. A client being lost for any reason will only cause those files owned by the lost client to become unavailable to new readers, while local copies of these files will still be accessible to existing readers. If a MDS is lost, those files owned by the MDS are lost. However, clients connected to this MDS may re-connect to a different MDS, while multiple parents allow any children of the failed MDS to still access the rest of the network.

Chapter 3

Security and Authentication

The security and privacy of Ringer is built transparently on top of the GNU Transport Layer Security Library (GnuTLS)[5]. GnuTLS intercepts data streams at the TCP socket level; transmitted data is encrypted and compressed on its way out and decrypted and decompressed when it is received at the other end. A handshake protocol is used to negotiate a cypher and authenticate both the client and server. By default, Ringer uses a relaxed protocol which promotes interoperability. If a non-encrypted client (server) asks to connect to a encrypted server (client) the encryption will be dropped and the connection made in the clear. This behavior is customizable, and Ringer can enforce encryption if security is a priority over flexibility.

The GnuTLS library allows for both X.509 and OpenPGP based authentication; Ringer uses the PGP model.

Trust in Ringer flows uphill, in the sense that clients trust their MDS to be a trusted introducer and MDSs trust their parents to be trusted introducers. Trust is not recursive, and because a certificate is trusted (but not signed) by a parent MDS does not mean that the child

MDS will trust the certificate. Therefore, whenever a MDS receives a certificate to sign, it passes it up to all of its parents to sign as well.

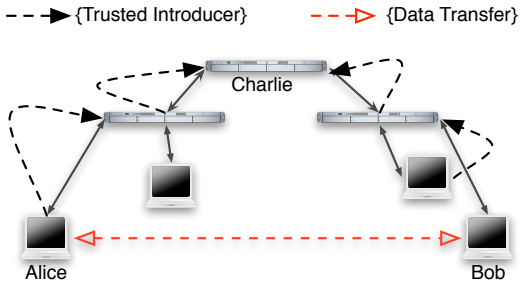

Figure 3.1: In Ringer, trust flows uphill. Parents are trusted introducers, children are not. In this example, Alice and Bob are introduced to each other by Charlie.

An example of how this all works is seen in Figure 3.1. Alice wants to communicate with Bob, but they do not share a common top level MDS. However, both Alice and Bob do trust Charlie to be an introducer and Charlie has singed both Alice and Bob's certificates. Therefore, when Alice and Bob send each other their certificates, they both see that they are being introduced by a trusted introducer.

Ringer supports `user/group/all` style read and write permissions. These are set using the Unix `chmod` command. Permissions can only be changed by the client which owns the file. All users on the same host machine are considered part of the same group; adding and managing new groups is left for future work. Each client is identified uniquely by their machine's host-name and local user id.

Chapter 4

Search

Search is the key component of Ringer. Any document, once found, can be read. It is the finding, in a distributed manor, which is the hard part. One major goal of Ringer is not to be tied to any one search program. Instead, we designed Ringer to be a search framework, allowing many different algorithms to be easily implementable on top of the basic Ringer architecture. Currently we explore two example approaches to search: human tagging and machine computed binary similarity.

4.0.7 Tag Search

The first search method we support is a tag based one. Files are tagged by human users, using the extensible attribute component of the file's local inode. The new tag is relayed to the client's MDS, which needs to update the index of tags. This MDS attempts to find a MDS which "owns" the tagged keyword, via a breadth-first search of its ancestors. A parent MDS knows all of the keywords which its children own, and can respond to a keyword lookup request

with the address of the owning child. If no result comes back after a given timeout period, or if the owning MDS is down, the client's MDS notifies its parents that it has assumed ownership of the keyword. If another MDS is found to be the owner, this MDS is sent the absolute Ringer path of the tagged file. Note that there may be multiple owners for each keyword.

This scheme is optimized for the assumption that clients are likely to re-use metadata tags, leading to fast lookup for commonly used tags. The number of owners is a function of how well connected the MDS parent relation graph is. Note that there is no need to prepare a mapping of keywords to MDSs in advance. There is also no reliance on any kind of central authority to arbitrate keywords. Associating keywords with specific owners allows us to prune the search space. There is no need to ask every MDS if they have a matching file: you only need to go up your list of ancestors high enough to identify which MDS owns the requested tags.

Human users (searchers) can query the system via a standalone program. Searchers input a list of search keywords and a MDS via a command line interface. The search program then contacts that MDS and relays the keyword list. This MDS then looks up keyword owners, in the manor described above. Again, if there is no owner, the MDS assumes ownership. After collecting a list of owners, the MDS contacts the owners directly, asking for all files matching each keyword. The responses are then collected into a single response and sent back to the issuing search program. The results are presented to the searcher as a list of Ringer file paths, grouped by the keywords satisfied.

An extension of tag based search would be a full text keyword search. In this form, all words in a file would be considered keywords, and indexed accordingly. Beyond the communication overhead, we consider this extension unsatisfactory because our desire is for Ringer

to work equally well with all data formats and languages, and not require specialized handling for any particular format.

4.0.8 Similarity Search

The other type of search implemented came about because we wanted a search which did not need any human input.

Hash functions allow for a compact representation of a file, so that identical files can be quickly identified. With a *similarity hash*, files which have similar content hash to similar values.

We integrated an existing similarity hashing function [1] as a library in Ringer to create the similarity search. This code works as follows: since the goal is binary similarity, the hashing function is initialized with a set of n byte-strings to look for. These byte-strings are static; the same strings are used on all files throughout the Ringer system. In each file processed, the similarity hash counts the number of occurrences of each string. These counts are turned into an n dimensional vector. Similarity is defined by the angle between vectors. As two files get closer to each other, we expect this angle to go to 0.

In Ringer, similarity search uses a basic broadcast pattern. When a file is added or edited, its similarity vector is computed and stored as part of its metadata. Because we use a bounded set of strings to look for, this does not grow too large. For search, a file and an epsilon are given to the search client. The target file is hashed, and the hash value and the epsilon sent to a MDS. This MDS sends the search to its parents and children, and so on. Any MDS which

[1] Written by Caitlin Sadowski and Greg Levin, both graduate students at UCSC.

owns files with similarity vectors within an angle of epsilon to the searched for file's similarity vector are returned. The results are recursively aggregated and returned, after a timeout period ends, as a list of Ringer paths to the original requesting search client.

In practice, we do not separate out tagging and similarity search, instead giving the search client a mixed list of tags and file paths to hash. If a directory is given, all files in that directory are hashed separately.

Chapter 5

Experimental Results

Testing is still ongoing. We have completed a few small scale evaluations, presented below, and feel that the filesystem is relatively stable, even in a wide area, low bandwidth environment. Further tests will evaluate Ringer's performance, including search, as a large distributed file system. Using the Amazon Elastic Compute Cloud, we are planning on measuring the scalability of Ringer as it grows from from a few client nodes and one MDS to hundreds of clients and MDSs.

5.0.9 Micro-Benchmarks

We first present several micro-benchmarks. We measure the time to stat a file and read the file's metadata, acquire a read lease, acquire a write lease, end the read lease, and finally end the write lease. All of these require communication with a separate MDS process. Each time is the average time in microseconds measured using the `gettimeofday()` system call of 1000 runs. Our results are presented in Table 5.0.9.

test	local	remote (DSL)	remote (GigE)	mixed (DSL)
`put_rnode()`	223	14,575	746	n/a
`get_rnode()`	472	35308	724	n/a
`get_read()`	413	31103	501	33000
`get_write()`	791	4403	524	786
`end_read()`	498	33471	530	57000
`end_write()`	874	5403	525	829

Table 5.1: Metadata micro-benchmarks (in microseconds).

local refers to a client communicating with a MDS over a loopback device. *remote (DSL)* refers to a client and a remote MDS accessed via a DSL connection. *remote (GigE)* is the same but using a gigabit ethernet connection for the client/MDS communication. Finally, *mixed (DLS)* refers to local client speaking to a local MDS which in turn communicates with a remote MDS via DSL.

The major differences within columns are caused by the synchronous nature of the three `get*()` calls, which all wait for a response from the server before returning. The other methods (`end_read()`, `end_write()` and `put_rnode()`) all return immediately after any client-side processing.

While the latency of DSL obviously adds to the time required to get leases, the overhead is cut almost in half because no response is needed to end a lease.

5.0.10 Integrated Performance

Next we compare Ringer and the SSHFS filesystem, chosen because it is also FUSE based, as they perform a sequential read over a gigabit ethernet connection.

The results depicted in Figure 5.1 show elapsed time in seconds to complete a series

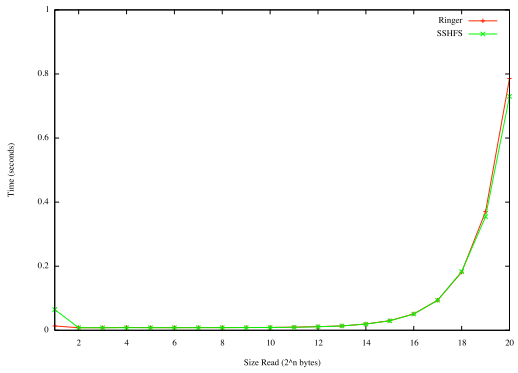

Figure 5.1: Time in seconds to read 2^n bytes of sequential data.

of reads, from 2 bytes to 2 megabytes. These are the mean times out of 50 runs.

For this test using Ringer, one client and MDS are hosted on the same machine, while the reading client is mounted on a separate computer. SSHFS uses a client-server model. The times for Ringer and SSHFS are very similar. We interpret this as showing that the added overhead Ringer incurs by finding files via a MDS is acceptably small and is largely eclipsed by data transfer.

Turning our attention to writes in Figure 5.2, we see that the situation is very similar.

Ringer writes data only to the local disk, and after this the writing client considers the operation to be complete.

It is only when another client reads the written data is data sent over the network. For testing purposes, we consider this to be a read. Instead of data transfer then, the write times shown here are dominated by the calls to the MDS, which lead to the mostly flat performance of Ringer. The uptick at the shows the effect of writing larger files to local disk. SSHFS however sends all writes to the server, and so write time grows with write size. Especially on a slow

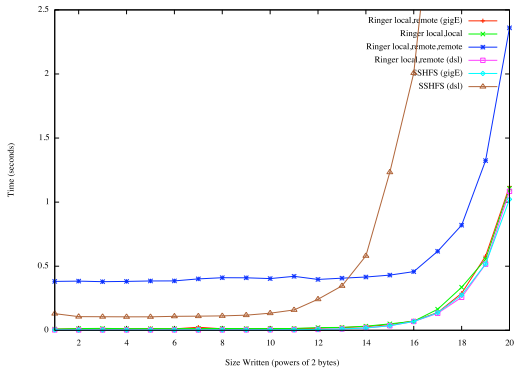

Figure 5.2: Time in seconds to write 2^n bytes of sequential data.

connection, network time comes to dominate disk access.

5.0.11 EC2

As another experiment to test scalability, we set up a larger Ringer network on a set of nodes using the Amazon Elastic Compute Cloud framework. Figure 5.3 shows the total elapsed time in seconds to setup and take down a network composed of one MDS and between one and ten clients. Each node is run on a separate small (1 core) instance, with all instances in the same availability zone.

5.0.12 Block Size

We also look at the effect of varying the block size has on Ringer.

Because Ringer is build on Fuse, we did not change the size of data which Ringer handles on a local level. Instead, we changed the amount of data which is requested whenever a block is not in the local file cache. As expected, with no network delay (see Figure 5.4), this

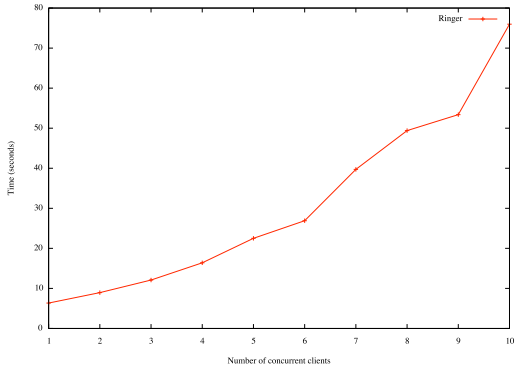

Figure 5.3: Time in seconds to launch and stop concurrent clients.

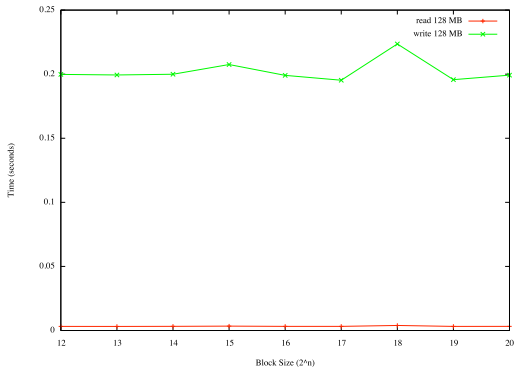

Figure 5.4: Block-Size: Time needed to read and write 128K of data as the block size increases, with no network delay.

does not effect things greatly.

Because leases last only the length of the system read() and write() calls, we believe that very long block sizes here would result in faster access times due to the reduction in the number of leases needed for a transaction. Unfortunately, this system block size is defined by Fuse and is not accessible to the application writer.

Chapter 6

Related Work

The history of distributed filesystems is almost as long as the history of computer networks. The original model and still the most commonly used systems in practice are client and server rigs such as NFS [11] and CIFS[7]. These systems are conceptually simple and, largely because of their ubiquity, easy to deploy and control. The existence of a single central server however is a major limitation on how far such systems can be scaled. Both NFS and CIFS focus exclusively on providing high speed connections between systems operating on a LAN.

One attempt to extend a client-server based network to low bandwidth WANs is the Low-bandwidth Network File System[9]. Here, a file is broken into chunks based on Rabin[12] fingerprints. A Rabin fingerprint is the polynomial representation of the data modulo a predetermined irreducible polynomial.

In the High Performance Computing world, where network interconnect is assumed to be fast and the limiting factor is largely disk bandwidth, the ability to handle petabytes of data

extremely quickly has become the greatest challenge to any distributed filesystem. Currently, this is accomplished by striping files across multiple disks, while separating out storage and compute nodes.

One implementation of this is via an object based interface. As reported by Mesnier[8], "Objects are storage containers with a file-like interface, effectively representing a convergence of the NAS and SAN architectures. Objects capture the benefits of both NAS (high-level abstraction that enables cross platform data sharing as well as policy-based security) and SAN (direct access and scalability of a switched fabric of devices)."

Some implementations of Object based file systems include xFS[13] and Ceph[16], discussed earlier.

There has been widespread interest in peer-to-peer and wide-area networks in the past 10 years. To briefly mention a few, the Gnutella and Freenet networks [4, 2] are both popular examples.

Closer to Ringer, Kosha [3], extends the basic client-server model of NFS with peer-to-peer data transfer. Kosha uses NFS to provide hierarchical file organization, directory listings and file permissions, very much like the MDS concept in Ringer. Kosha peers transfer data directly between each other, again in a similar manner to Ringer. A key difference is that Ringer supports a de-centralized MDS network, while Kosha relies on a single NFS server to handle the metadata requests of all its clients. Also, while Ringer has an emphasis on wide-area search, Kosha is targeted at providing local area performance.

Another system similar to Ringer is Anglano and Ferrino's work with N3FS. In [1], they evaluate using the Chord [14] distributed hash table like the MDS network in Ringer to

provide metadata storage and lookup. The difference again with Ringer is its emphasis on search.

Chapter 7

Conclusions and Future Work

In the near future, we are focused on testing Ringer's performance running on a large number of nodes connected only via a low bandwidth network. Additionally, in order to see how Ringer executes on a resource limited environment, we are porting both the client and MDS parts of Ringer to run on a 500 MHz ARM powered NAS box.

Another active area of research for us is determining the optimal connectedness for the MDS network. More parents for each MDS mean a better chance of finding queried rnodes in searches but makes each search take longer. We believe that two or three parents for each MDS will provide enough search paths for most queried rnodes to be found, and found relatively quickly. A further refinement would entail dynamically updating the parent/child relationships so that a MDS hosting a "hot" rnode has many paths leading to it, but "cold" rnodes are only found via a few paths.

In conclusion, information exists in forms where it is either difficult to find or, once found, difficult to update in a coherent manor. Our solution is to create a hybrid network where

content is found via a collection of metadata servers but transfered directly between clients. The resulting lack of any central authority ensures scalability, while unique file paths and a system of leases provide coherency. The specialized metadata servers in turn facilitate search. We hope that Ringer will prove a reliable, efficient and scalable way to facilitate globally distributed collaboration.

Bibliography

[1] C Anglano and A Ferrino. Using chord for meta-data management in the n3fs distributed file system. *Peer-to-Peer Systems*, Jan 2004.

[2] J Berkes. Decentralized peer-to-peer network architecture: Gnutella and freenet. *University of Manitoba Winnipeg*, Jan 2003.

[3] A Butt, T Johnson, Y Zheng, and Y Hu. Kosha: A peer-to-peer enhancement for the network file system. *Journal of Grid Computing*, Jan 2006.

[4] I Clarke, O Sandberg, B Wiley, and T Hong. Freenet: A distributed anonymous information storage and retrieval system. *Workshop on Design Issues in Anonymity and Unobservability*, Jan 2000.

[5] The gnu transport layer security library. http://www.gnu.org/software/gnutls/.

[6] David Hawking. Challenges in enterprise search. In *ADC '04: Proceedings of the 15th Australasian database conference*, pages 15–24, Darlinghurst, Australia, Australia, 2004. Australian Computer Society, Inc.

[7] P Leach. Cifs authentication protocols specification. *Microsoft*.

[8] M Mesnier, G Ganger, E Riedel, and C Mellon. Object-based storage. *Communications Magazine*, Jan 2003.

[9] A Muthitacharoen, B Chen, and D Mazières. A low-bandwidth network file system. *Proceedings of the eighteenth ACM symposium on Operating . . .*, Jan 2001.

[10] A Muthitacharoen, R Morris, T Gil, and B Chen. Ivy: A read/write peer-to-peer file system. *Proc. of OSDI*, Jan 2002.

[11] B Pawlowski, S Shepler, C Beame, and B Callaghan. The nfs version 4 protocol. *Proceedings of the 2nd international system administration . . .*, Jan 2000.

[12] M.O. Rabin. Fingerprinting by random polynomials. Technical report, Technical Report TR-15-81, Center for Research in Computing Technology, Harvard University, 1981.

[13] O Rodeh and A Teperman. zfs-a scalable distributed file system using object disks. *Mass Storage Systems and Technologies*, Jan 2003.

[14] I Stoica, R Morris, D Karger, and M Kaashoek. Chord: A scalable peer-to-peer lookup service for internet applications. *Proceedings of the 2001 SIGCOMM conference*, Jan 2001.

[15] C Thekkath, T Mann, and E Lee. Frangipani: a scalable distributed file system. *ACM SIGOPS Operating Systems Review*, Jan 1997.

[16] S Weil, S Brandt, E Miller, and D Long. Ceph: A scalable, high-performance distributed file system. *Proceedings of the 7th Symposium on Operating Systems Design . . .*, Jan 2006.