A Case Study of Server Selection

by

Tina Tyan

S.B., Computer Science and Engineering (2000)
Massachusetts Institute of Technology

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Engineering in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

September 2001

© Massachusetts Institute of Technology 2001. All rights reserved.

Author ....
Department of Electrical Engineering and Computer Science
August 8, 2001

Certified by ....
M. Frans Kaashoek
Professor of Computer Science and Engineering
Thesis Supervisor

Accepted by ....
Arthur C. Smith
Chairman, Department Committee on Graduate Theses

A Case Study of Server Selection

by

Tina Tyan

Submitted to the Department of Electrical Engineering and Computer Science
on August 8, 2001, in partial fulfillment of the
requirements for the degree of
Master of Engineering in Electrical Engineering and Computer Science

Abstract

Replication is a commonly used technique to improve availability and performance in a distributed system. Server selection takes advantage of the replicas to improve the end-to-end performance seen by users of the system. CFS is a distributed, cooperative file system that inherently replicates each piece of data and spreads it to machines dispersed around the network. It provides mechanisms for both locating and retrieving data. This thesis considers the application of server selection to improving performance in CFS, in both the data location and data retrieval steps. Various server selection metrics and methods were tested in an Internet testbed of 10-15 hosts to evaluate the relative benefits and costs of each method.

For the lookup step, we find that the triangle inequality holds with good enough correlation that past latency data stored on intermediary nodes can be used to select each server along the lookup path and reduce overall latency. For the data retrieval step, we find that selecting based on an initial ping probe significantly improves performance over random selection, and is only somewhat worse than issuing parallel requests to every replica and taking the first to respond. We also find that it may be possible to use past latency data for data retrieval.

Thesis Supervisor: M. Frans Kaashoek
Title: Professor of Computer Science and Engineering

Acknowledgments

Several of the figures and graphs in this thesis, as well as the pseudocode, came from the SIGCOMM[24] and SOSP[8] papers on Chord and CFS. There are a number of people who have helped me through the past year and who helped make this thesis possible. I'd like to thank Kevin Fu for his help getting me started, and Frank Dabek for his help on the later work. Thank you to Robert Morris for his invaluable feedback and assistance. I would also like to thank the other members of PDOS for their suggestions and ideas. I would especially like to thank my advisor, Frans Kaashoek, for his guidance, patience, and understanding. Thanks also to Roger Hu for always being available to answer my questions (and for all the free food), and Emily Chang for her support and assistance in getting my thesis in shape (particularly the Latex). Working on a thesis isn't just in the work, and so I'd like to extend my appreciation to the residents of the "downstairs apartment", past and present, for giving me a place to get away from it all, especially Danny Lai, Stefanie Chiou, Henry Wong, Roger, and Emily. A special thanks to Scott Smith, for always being there for me, whether with suggestions, help, or just an ear to listen. Lastly, I'd like to thank my family - Mom, Dad, Jeannie, and Karena - for all their support and love. I would never have gotten here without them.

Contents

1 Introduction **11**

2 Background **15**

2.1 Overview 15

2.1.1 Chord 16

2.1.2 CFS 20

3 Server Selection in CFS **23**

3.1 Server Selection at the Chord layer 23

3.1.1 Past Latency Data 25

3.1.2 Triangle Inequality 26

3.2 Selection at the dhash layer 26

3.2.1 Random Selection 27

3.2.2 Past Performance 27

3.2.3 Probing 28

3.2.4 Parallel Retrieval 29

4 Experiments **31**

4.1 Experimental Conditions 31

4.2 Chord Layer Tests 32

4.2.1 Evaluating the use of past latency data 32

4.2.2 Triangle Inequality 36

4.2.3 Chord Layer Selection Results 43

4.3 dhash Layer Tests 44

4.3.1 Server 44

4.3.2 Ping Correlation Test 45
4.3.3 Selection Tests 49
4.3.4 Parallel Downloading Tests - Methodology 50
4.3.5 Combined Tests 51

5 Related Work **57**

5.1 Internet Server Selection 57
5.1.1 Intermediate-level Selection 58
5.1.2 Client-side 61
5.2 Peer-to-peer Systems 65

6 Conclusion **69**

6.1 Conclusion 69
6.2 Future Work 70

List of Figures

2-1 Example Chord ring with 3 nodes 17

2-2 Example Chord Ring with Finger Tables 19

2-3 The pseudocode to find the successor node of an identifier *id*. Remote procedure calls and variable lookups are preceded by the remote node. 20

2-4 CFS Software Layers 21

3-1 Example of choice in lookup algorithm 25

4-1 RON hosts and their characteristics 32

4-2 Sightpath to Msanders Ping Times 34

4-3 Cornell to Sightpath Ping Times 35

4-4 Mazu to Nc Ping Times 35

4-5 nc to lulea Ping Times 36

4-6 Simple Triangle Inequality 37

4-7 Triangle Inequality: AC vs AB+BC 38

4-8 Relative Closeness 39

4-9 Chain of Length 3 41

4-10 Best and Worst Ping Ratios for Tests 1 and 2 47

4-11 Best Ping Ratio for Tests 1 and 2 (same as in figure 4-10) 47

4-12 Best and Worst Ping Ratios for Test 3 48

4-13 Best Ping Ratio for Test 3 (same as in figure 4-12) 48

4-14 Parallel Retrieval Results 53

4-15 Ping Probe Results 54

4-16 Aros Ping Probe Results 54

4-17 Random, Best Ping, Parallel Results 56

Chapter 1

Introduction

The Internet is becoming an ubiquitous part of life as increasingly large numbers of people are going online and the number of available resources grows. Users use the Internet to obtain all types of public data, from web pages to software distributions. However, along with increased popularity and usage come problems of scalability and performance degradation. When a popular resource is only available from a single source, that server can experience excessive load, to the point of being unable to accommodate all requests. Users located geographically or topologically far from the server are prone to excessive latency, which can also be caused by network bottlenecks or congestion. This sole source also provides a single point of failure; if this server goes down, then the resource becomes completely unavailable.

Server replication provides a viable solution to many of these problems, improving performance and availability. In replication, some subset of the server's data or a given resource is duplicated on multiple servers around the network. The existence of replicas allows requests for data to be distributed, lessening the load on each server and accommodating greater overall load. Replicas also add redundancy to the system; if one server goes down, there are still many others available to take up the slack. If the replicas are distributed across the network, then a given client has a greater chance of being located closer to a source of the data, and therefore experiencing lower latency than if it tried to contact one central server.

Along with replication comes the added problem of server selection. In order for replicated services to be useful, a replica must be selected and assigned to a client by some method. This selection can be done by the server, by the client, or by some intermediary

service. The criteria and method of selection will have varying impacts on performance and server load. It is in this step that the real benefits of replication are derived. The interesting thing about server selection is that there is no easy way to choose the best server; varying network conditions, server load, and other factors can change performance characteristics from client to client and request to request. Additionally, what is considered to be "best" varies from system to system.

This thesis considers the specific problem of server selection within CFS (Cooperative File System), a distributed, decentralized file system designed to allow users to cooperatively pool together available disk space and network bandwidth. CFS has many useful properties that make it an attractive system to use, and could benefit from improved performance. An important feature of CFS is its location protocol, which is used to locate data in the system in a decentralized fashion. CFS inherently takes care of replication, replica management, and replica location, three primary concerns in any replicated system. Although the basic motivation behind replication is the same as in other systems, CFS has specific requirements and characteristics that make it somewhat unique from other systems in which server selection has been studied.

CFS' decentralized, cooperative nature means that instead of having a dedicated set of servers and replicas with distinct clients that access them, every node in the system can act as both a client and a server. Also, the location protocol of CFS, to be described in greater detail in chapter 2, utilizes nodes to act as intermediaries in the lookup process. Therefore nodes acting as servers are constantly in use for purposes other than serving data, and do not have to be particularly high performance machines behind high bandwidth links. CFS also ensures that data is replicated on machines that are scattered throughout the Internet. The pool of servers for a particular piece of data is therefore quite diverse, and certain servers are likely to have better performance for a given client than others. Additionally, while Web-based systems must deal with files of all sizes, CFS uses a block-level store, dividing each file into fixed size blocks that are individually retrieved. These block sizes are known and can be taken advantage of in choosing a server selection mechanism, whereas the sizes of Web files are not known before retrieval, making any size-dependent mechanism hard to use. Storing data at the level of blocks also load balances the system, so that the selection method need not be particularly concerned with keeping the load balanced.

The primary concern for server selection in CFS is the improvement of the end-to-end

performance experienced by the client. For CFS, this does not only include the end-to-end performance of the data retrieval from a replica, but also includes the overall performance of the data location step. The requirements for selection for location are fairly distinct from those of data retrieval. The purpose of this thesis is to investigate and experimentally evaluate various server selection techniques for each stage of CFS operation, with the intention of improving overall end-to-end performance in the system.

Chapter 2 will give a brief overview of CFS and how it works. Chapter 3 goes into more detail on the requirements for selection in CFS, as well as those aspects of the system most directly relevant to server selection. Chapter 4 describes the experiments conducted to evaluate the selection techniques introduced in the previous selection, as well as the results so obtained. Previous work done in server selection, both for the World Wide Web and other peer-to-peer systems, is described in chapter 5. Lastly, chapter 6 wraps up the discussion and proposes future directions for work in this area.

Chapter 2

Background

This thesis studies server selection techniques and conditions in the context of the Chord/CFS system. To better understand the problems that server selection must address, we take a closer look at the system in question in this chapter. The next chapter will discuss server selection within Chord and CFS and highlight relevant aspects of the system.

2.1 Overview

In recent years, peer-to-peer systems have become an increasingly popular way for users to come together and share publicly-accessible data. Rather than being confined to the disk space and bandwidth capacity available from a single (or even several) dedicated server(s), the peer-to-peer architecture offers the ability to pull together and utilize the resources of a multitude of Internet hosts (aka nodes). As a simple example, if the system was comprised of 1000 nodes with only 1MB free disk space on each, the combined disk capacity of the entire system would total 1GB. Participating nodes are likely to contribute far more than 1MB to the system. Given more hosts and more available disk space, the potential available storage in such a system far exceeds that of a traditional client-server based system. Similarly, the potential diversity of the available data is increased by the fact that there can be as many publishers of data as there are users of the system. At the same time, since there is no single source of information, the bandwidth usage is distributed throughout the system.

Although the concept of the peer-to-peer architecture is attractive, implementing such a system offers some challenges. Recent systems like Napster and Gnutella have received widespread use, but each has its limitations. Napster is hampered by its use of centralized

servers to index its content[16]. Gnutella is not very scalable since it multicasts its queries [11]. Other peer-to-peer systems experience similar problems in performance, scalability, and availability [24]. One of the biggest challenges facing these systems is the problem of efficiently and reliably locating information in a decentralized way. CFS (Cooperative File System) is a cooperative file system designed to address this problem in a decentralized, distributed fashion. It is closely integrated with Chord, a scalable peer-to-peer lookup protocol that allows users to more easily find and distribute data. We give a brief overview of Chord and CFS in this chapter; details can be found in [7], [8], [24], and[25].

2.1.1 Chord

Chord provides a simple interface with one operation: given a key, it will map that key onto a node. Chord does not store keys or data itself, and instead acts as a framework on top of which higher layer software like CFS can be built. This simple interface has some useful properties that make it attractive for use in such systems. The primitive is efficient, both in the number of machines that need to be contacted to locate a given piece of data, and in the amount of state information each node needs to maintain. Furthermore, Chord adapts efficiently when nodes join or leave the system, and is scalable with respect to the total number of participating nodes in the system.

2.1.1.1 Chord Identifiers

Each node in the Chord system is associated with a unique m-bit Chord node identifier, obtained by hashing the node's IP address using a base hash function, such as SHA-1. Just as nodes are given node identifiers, keys are also hashed into m-bit key identifiers. These identifiers, both node and key, occupy a circular identifier space modulo 2^m. Chord uses a variant of consistent hashing to map keys onto nodes. In consistent hashing, the node responsible for a given key k is the first node whose identifier is equal to or follows k's identifier in the identifier space. When the identifiers are pictorially represented as a circle of numbers from 0 to $2^m - 1$, this corresponds to the first node clockwise from k. This node is known as the "successor" of k, or successor(k).

A simple example of a Chord network for which m=3 is seen in Figure 2-1. There are three nodes in this network, with identifiers 0, 1, and 3. The set of keys' identifiers to be stored in this network is $\{1, 2, 6\}$. Within this network, the successor of key 1 is node 1,

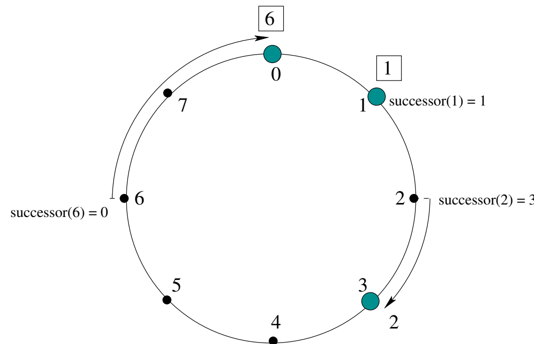

Figure 2-1: Example Chord ring with 3 nodes

since the identifiers match. The successor of key 2 is node 3, which can be found by moving clockwise along the circle to the next node after 2. Similarly, the successor for key 6 is the first node found clockwise after 6 on the identifier circle, which in this case wraps around to node 0.

2.1.1.2 Consistent Hashing

There are several benefits to using consistent hashing. Consistent hashing allows nodes to join and leave the network with minimal disruption. When a node joins the network, a certain set of keys that previously mapped to its successor are reassigned to that node. In the above example, if node 7 were to join the network, key 6 would be reassigned to it instead of node 0. Similarly, when a node leaves the network, its keys are reassigned to its successor. In this way, the consistent hashing mapping is preserved, and only a minimal number of keys — those for the node in question — need to be moved. Consistent hashing also provides load balancing, distributing keys with high probability in roughly equal proportions to each node in the network.

In a centralized environment where all machines are known, consistent hashing is straightforward to implement, requiring only constant time operations. However, a centralized system like that does not scale. Instead, Chord utilizes a distributed hash function, requiring each node to maintain a small amount of routing information.

2.1.1.3 Key Location

In order for consistent hashing to work in a distributed environment, the minimum amount of routing information needed is for each node to be aware of its successor node. A query for a given identifier k would therefore follow these successor pointers around the circle until the successor of k was found. This method is inefficient, however, and could require the query to traverse all N nodes before it finds the desired node. Instead, Chord has each node maintain a routing table of m entries (corresponding to the m-bit identifiers), called the finger table.

The finger table is set up so that each entry of the table is "responsible" for a particular segment of the identifier space. This is accomplished by populating each entry i of a given node n with the Chord ID and IP address of the node $s = successor(n + 2^{i-1} mod 2^m)$, where $1 \leq i \leq m$. This node, s, is also known as the ith finger of n, and is denoted as *n.finger[i].node*. Thus, the 1st finger of n is its immediate successor (successor$(n + 1)$), and each subsequent finger is farther and farther away in the identifier space. A given node therefore has more information about the nodes nearest itself in the identifier space, and less information about "far" nodes. Also stored in the ith entry of the finger table is the ith *finger interval*. This is fairly straightforward; since the ith finger is the successor of $(n + 2^{i-1})$ and the $i + 1^{th}$ finger is the successor of $(n + 2^i)$, then *n.finger[i].node* is this node's predecessor for any identifier that falls within the interval [*n.finger[i].node*, *n.finger[i+1].node*).

Figure 2-2 shows the finger tables for the simple Chord ring examined earlier. Since $m = 3$, there are 3 entries for each finger table. The finger table for node 1 is populated with the successor nodes of the identifiers $(1 + 2^{1-1})$ mod $2^3 = 2$, $(1 + 2^{2-1})$ mod $2^3 = 3$ and $(1 + 2^{3-1})$ mod $2^3 = 5$. In this particular network, successor$(2) = 3$, successor$(3) = 3$, and successor$(5) = 0$. The finger tables for the other nodes follow accordingly.

These finger tables allow Chord to systematically and efficiently search for the desired node by directing queries increasingly closer in the identifier space to that node. The closer a node is to a region of the identifier circle, the more it knows about that region. When a node n does not know the successor of a key k, it looks in its finger table for the set of nodes between itself and the destination in ID space. These nodes are all closer in ID space to the key than the origin, and are possible candidates to ask for its location. The simplest implementation chooses the node *n'* closest in ID space to k and sends it a *closest-preceding-*

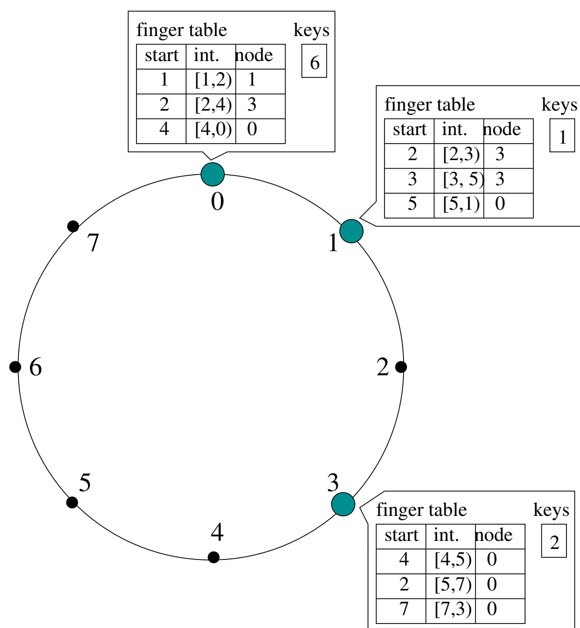

Figure 2-2: Example Chord Ring with Finger Tables

nodes RPC. Node n' then looks in its finger table and sends back the list of successors between itself and the destination, allowing n to select the next node to query. As this process repeats, n learns about nodes with IDs closer and closer to the target ID. The process completes when k's predecessor is found, since the successor of that node is also k's successor. The pseudocode for these procedures can be found in figure 2-3.

A simple Chord lookup query takes multiple hops around the identifier circle, with each hop eliminating at least half the distance to the target. Consequently, assuming node and key identifiers are random, the typical number of hops it takes to find the successor of a given key is O(log N), where N is the number of nodes in the network. In order to ensure proper routing, even in the event of failure, each node also maintains a "successor-list" of its r nearest successors on the Chord ring. In the event that a node notices its successor has failed, it can direct queries to the next successor in the "successor-list," thus updating its successor pointer.

The Chord protocol also has provisions for initializing new nodes, and for dealing with nodes concurrently joining or leaving the network. These aspects of the protocol are not relevant to server selection, and are not necessary to understand the system.

```
// ask node n to find id's successor
n.find_successor(id)
  n' = (id);
  return n'.;

// ask node n to find id's predecessor
n.find_predecessor(id)
  n' = n;
  while (id ∉ (n', n'.])
    l = n'.closest_preceding_nodes(id);
    n' = max n'' ∈ l s.t.n"isalive
  return n';

// ask node n for a list of nodes in its finger table or
// successor list that most closely precede id
n.closest_preceding_nodes(id)
  return {n' ∈ {fingers ∪ successors}
    s.t. n' ∈ (n, id)}
```

Figure 2-3: The pseudocode to find the successor node of an identifier *id*. Remote procedure calls and variable lookups are preceded by the remote node.

2.1.2 CFS

Chord was created as a general-purpose location protocol, with the intention that higher-level systems with more specific functionality would be built on top of it. One such system is CFS, a read-only file system designed to allow users to cooperatively pool disk space and network bandwidth in a decentralized, distributed fashion. CFS nodes cooperate to locate and load-balance the storage and serving of data in the system. CFS consists of three layers (Figure 2-4): the file system layer, which provides the user with an ordinary file system interface; the dhash layer, which takes care of data storage and retrieval issues; and the Chord layer, which provides mappings of keys and nodes, and acts as a location service. Data is not stored at the level of full files, but instead as uninterpreted blocks of data with unique identifiers. The client may interpret these blocks as file data or as file system metadata.

2.1.2.1 dhash layer

For the purposes of server selection, the relevant functions of CFS occur at the dhash layer. The way that blocks are assigned to servers is very closely tied to the way the underlying

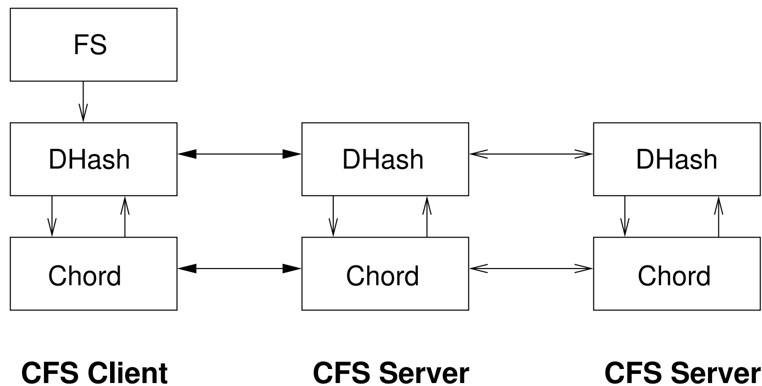

Figure 2-4: CFS Software Layers

Chord protocol functions. Each block of data has an associated unique identifier. This ID could be a cryptographic hash of the block's contents or a public key, and is equivalent to a key in Chord. A block is stored at the server that Chord identifies as the successor of the block's key.

There are some tradeoffs to keeping everything at the block level and storing the data in this way. Blocks are distributed roughly evenly around the ID space, since the hash function uniformly distributes the block IDs. Consequently, the data is not all retrieved through the same paths and bandwidth usage is spread around the network. This provides the system with a certain degree of load balancing. However, instead of doing one Chord lookup per file or file system accessed, this system requires there to be one Chord lookup per block retrieved, reducing location and retrieval efficiency.

2.1.2.2 Replication

CFS takes advantage of other properties of Chord to increase availability and potentially increase efficiency through replication. As part of its failure recovery mechanism, each Chord node maintains a "successor-list" of its nearest r successors on the Chord ring, which is automatically updated and maintained as nodes come and go. These successors are a natural place to replicate the data. There are two main concerns for replicated data: keeping track of the replicas so that the data can be found and updated, and making sure the replicas are independent of failure, so that if one machine goes down, the replica will still be available. The node is already keeping track of the r successors through its successor

list, so there is no need for an external mechanism to maintain the replicas. At the same time, these successors are unlikely to be near each other geographically or topologically, since their Chord IDs are hashes of their IP addresses. Numerically similar addresses are unlikely to hash to numerically similar identifiers, by the nature of the hash function. This means that the replicas are more likely to be failure independent and also that it is likely that at least one of the replicas will be close to a given client.

Aside from providing higher availability, this replication scheme paves the way for performance improvements through server selection, as will be described in greater detail in the following chapter.

Chapter 3

Server Selection in CFS

The current implementation of CFS does not do any sort of server selection, and instead just uses the first server found to contain the desired block. Performance tests run on a collection of 8 CFS servers spread over the Internet indicates that performance is strongly affected by the characteristics of the particular servers involved. When a server was added which happened to be far from the client, transfer times increased dramatically, while nearby servers helped to decrease transfer time. Although some of the effects of this can be alleviated through the use of pre-fetching (requesting several blocks simultaneously), these tests indicate a strong need for server selection to reduce variation in and improve the performance of the system.

There are two particular points in the CFS system where server selection could come into play. The first is at the Chord level, during the location protocol, while the second is at the dhash layer, in the actual data retrieval step. The nature of the selection required is somewhat different at these two layers, and leads to different requirements for a selection method.

3.1 Server Selection at the Chord layer

Server selection in the lookup layer strives to reduce the overall latency and variation in lookup times by eliminating as much as possible the need to make large geographical leaps in the lookup process. The CFS implementation of the Chord protocol facilitates the use of server selection by naturally providing a list of alternative nodes at each stage of the lookup process.

When a lookup is performed, the origin node n tries to find the predecessor of the desired key k by issuing a *closest-preceding-nodes* RPC to a node n' in its finger table that is closer in ID space to k than n is. Node n' subsequently looks in its own finger table and returns a group of successors that are all closer in ID space to k. This process is reiterated until the desired successor is found. In the current implementation, the node in this list of successors with the ID closest to k is the one selected for the next step of the implementation. While this method keeps the number of necessary hops around the Chord ring to a minimum since it makes the largest advances in ID space towards the target, the overall lookup time is strongly dependent on the performance of each intermediate node. If even one intermediate node thus chosen is particularly slow — in a different country, for instance — then the overall lookup time is significantly impacted. Potentially, server selection could be used at each stage of the lookup process to keep each query local, resulting in perhaps more hops but lower overall latency.

The developers of Chord and CFS are currently investigating an alternate lookup algorithm that would utilize server selection in a different way. Instead of simply picking the node with the closest ID, this algorithm examines all the possible servers in the finger table and selects the one that is most likely to keep the overall latency minimal. This is determined through a combination of factors. At each step of the lookup process, the algorithm strives to get closer to the destination ID through bit correction of the IDs - picking servers whose IDs match increasingly more bits of the destination ID. The number of future hops needed to arrive at the destination from that node can therefore be estimated by looking at the amount of bit correction needed on that node's ID. At the same time, CFS keeps track of the costs of all RPC's in the system, and can use this to estimate the average hop latency. By using a combination of these two estimates, and factoring in the latency between the selecting node and the potential intermediate node, the client can choose the intermediate node with the lowest potential cost/highest potential gain for the next step of the lookup.

Figure 3-1 gives a simple example to explain how this algorithm works. Nodes A and B are on the identifier ring between the client X and the target Y. B is closer to the target in ID space, and has a latency from the client of 100ms, while A is farther and has a latency of 10ms. As CFS runs, it keeps track of all previous transactions, and can determine the average latency of a hop based on this information. The client can also estimate the number of hops remaining between a given node and the target, based on the ID. Using these two

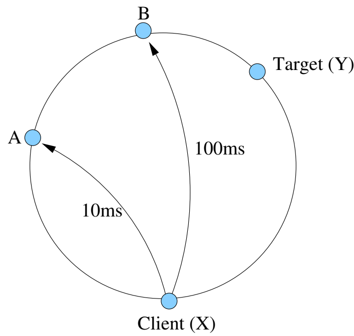

Figure 3-1: Example of choice in lookup algorithm

pieces of information, the client can then make a comparison between nodes A and B. If

$$HopLatency * NumHops_{AY} + Latency_{XA} < HopLatency * NumHops_{BY} + Latency_{XB}$$

then it may be better to pick node A rather than B. This is also another application of the triangle equality, with A and B acting as the intermediary nodes. If X-A-Y is not less than X-B-Y, then the whole thing falls apart.

The goals of server selection, and therefore the relevant metrics, can change depending on its overall purpose. In this case, the purpose of selection is not to find the server that will return data the fastest, but instead the server that responds fastest to a small lookup query. Therefore, round-trip latency should be a good indicator. However, it is not just a straightforward matter of always picking the successor with the lowest latency. There are a number of issues to be resolved in order to choose an appropriate selection method, and to determine if selection is even viable at all in reducing the overall latency of the lookup process.

3.1.1 Past Latency Data

Unlike with large downloads, where the time to probe the machines before selection is insignificant in comparison to the download time, each lookup hop only imposes a small overhead on top of the time to reach the machine. This overhead includes transmitting the request, searching the finger table, and returning the list of successors. Therefore, the time it would take to make just one round trip to the machine to probe it would impose a significant overhead on the overall time. Selection would be useful only if it could be done

solely on the basis of past latency data. If latency between machines varies widely over time, then past latency data may be insufficient to select the fastest machine.

Even assuming that the fastest machine can be found using past latency data, it is not clear that this will necessarily lead to better overall lookup latency. For every step of the lookup process except the initial request, the client chooses from a list of successors given to it by another node. The client does not maintain any past metrics for these machines and can rely only on the other node's information. Therefore, the available past performance data is not directly relevant to the client doing the selection, and it is impractical to probe the nodes before selecting one. If it could be shown that this indirect information is sufficient to improving the overall performance, then it will be possible to devise a good server selection method for the Chord layer.

3.1.2 Triangle Inequality

The specific question at hand is whether knowing which machine is closer to an intermediate machine is correlated to knowing which is closest to the client. A basic expression of this is the triangle inequality, which seeks to determine if machine A is close to machine B, and machine B is close to machine C, does it necessarily follow that machine A is close to machine C? More relevantly, if B is closer to X than Y, will A also be closer to X than Y? Since an actual lookup is likely to take several steps, it may also be useful to examine chains of such comparisons to see if utilizing the same methods through several levels of indirection still yields similar answers. If this is true, then always picking the fastest machine in each step of the lookup process will succeed in keeping the lookup process as topologically close to the client as possible, thus reducing the overall latency. If this isn't true, then latency won't help in deciding which machine to choose.

3.2 Selection at the dhash layer

After the lookup process is completed, and the key's successor is known, the client has the necessary information to download the desired block. It can choose to download from the actual successor, or to utilize the replicas on the following k successors in some sort of server selection scheme. The main goal of such a selection scheme would be to improve end-to-end performance.

3.2.1 Random Selection

The simplest selection method would be to randomly select a node from the list of replicas to retrieve the data from. This method incurs no additional overhead in selection, and has the effect of potentially distributing the load to these replicas if the block is popular. Although the random method has as good a chance of picking a slow server as a fast one, with each subsequent request it will likely pick a different server and not get stuck on the slow one. Without selection at all, if the successor of the key is slow, then all requests will be limited by the response time of this particular node.

3.2.2 Past Performance

Alternatively, selection could be done on the basis of past performance data. Clients can make an informed decision using past observed behavior to predict future performance. Previously, systems using past performance as their metric made use of a performance database, stored either at the client or at some intermediate resolver. This performance database included information on all known replicas for a previously accessed server. In a Web-based system with a few servers and many clients, this method is quite manageable. However, keeping such a performance database is less viable within CFS. The database is only useful if a client is likely to revisit a server. When a client has no knowledge of a particular set of replicas, it can not use past data to perform selection and must use some other form of selection for that initial retrieval. Even if it has accessed this data before, if the access did not take place recently, the performance data may no longer be relevant because of changes in the network. Thus the performance database must be kept up to date. With the potentially large number of nodes in a Chord network, and the way that blocks are distributed around the network, it does not seem likely that the same client will retrieve blocks from the same server with any great frequency. Therefore, it will constantly be encountering servers for which it has no information, while its stored performance data for other servers is rarely used and rapidly goes out of date.

However, past performance can still be used as the basis of server selection in CFS, without the need for a performance database. The method would in fact be very similar to that used in the Chord layer. Each node in the CFS system keeps in regular contact with its fingers and successors in order to keep its finger table and successor-list up to date.

The node therefore already stores fairly up-to-date latency information for each of these servers. Since the replicas for this node are located in the successors to the node, the node has past latency information for each replica. Using this information will require no extra trips across the network or additional overhead. Each lookup ends at the predecessor to the desired node, and its successor list will include all the replicas and their corresponding latency data. When the lookup returns the list of replicas to the client, the client can just use this information to select a server, much as it does for each stage of the lookup. A few crucial assumptions must be true for this method to work well. Since the nodes store past latencies, and not end-to-end performance data, latency must be a good indicator for end-to-end performance. The current block size in CFS is 8KB, which is adequately small enough that this is likely to be true. At the same time, the client is using latency data as measured from a different node to select a server that will provide it with good performance. In much the same way that the triangle inequality needs to hold for selection to work in the Chord layer, the triangle inequality also needs to hold for this method of selection to work at the dhash layer.

3.2.3 Probing

Unlike the location step, which only requires the round trip transfer of essentially one packet, the dhash layer involves the transmission of data blocks which can be of larger sizes. Therefore, the overall access time at the dhash layer is more likely to be dominated by the transfer of the actual data block. Thus, the additional overhead of probing the network before selection is more acceptable and is potentially countered by the increase in performance derived from selecting a good server. There are several different ways to probe the network. A fairly simple method would be to ping each replica, and request the block from the replica with the lowest ping time. This method works well if latency is a good indicator of the bandwidth of the links between the client and server. If servers tend to have low latency but also low bandwidth, then the performance will be worse than for machines with high latency and high bandwidth. In a system where very large files are transferred, latency may not be a very good indicator of actual bandwidth. However, since CFS transfers data at the level of blocks which are unlikely to be larger than tens of kilobytes, latency may be a good metric to use.

Since the issue at stake is the actual transfer time of a block, which is closely tied with

the bandwidth of the link, it may make sense to somehow probe the bandwidth rather than the latency of the system. A simple way to do this would be to send a probe requesting some size data from each server. Bandwidth can be calculated by dividing the size of the request by the time it takes for the request to complete. The server that has the highest bandwidth can than be selected to receive the actual request. To be useful, the probe should not be comparable in size to the actual request, or the time it takes to make the measurement will make a significant impact on the overall end-to-end performance. At the same time, probes that are too small will only take one round trip to fulfill the request, and are equivalent to just pinging the machine to find out the round trip time. Unfortunately, since CFS uses a block-level store, the size of the actual request is fairly small. Consequently, the probe must be correspondingly small, and therefore will be essentially equivalent to just measuring the latency.

3.2.4 Parallel Retrieval

The replicas can also be used to improve performance outside of just picking the fastest machine and making a request from it. Download times can potentially be improved by splitting the data over several servers and downloading different parts of the block from different servers in parallel. This effectively allows the client to aggregate the bandwidth from the different servers and receive the complete block in less time than it would take to get it from one server. This does not mean, however, that the best performance necessarily will come from downloading a fraction of the file from all the replicas. The end-to-end performance of these parallel downloads is constrained by the slowest of the servers. Even if 4 out of 5 of the servers are fast, as long as there is one slow server, the overall end-to-end performance could be worse than downloading the entire file from one fast server.

A better solution would then be to find the p fastest of the r replicas, and download $\frac{1}{p}$ of the block from each replica. This could be done in a few ways. The client could use one of the above two probing methods to find the p fastest servers and issue requests to just those machines. Alternatively, the client could simply issue a request for $\frac{blocksize}{p}$ to each of the replicas, and take the first p to respond with the data. Depending on how many replicas there are, how large p is, and the size of the request, this method could potentially use a lot of bandwidth.

3.2.4.1 Parallel Retrieval with Coding

This latter method bears closer examination. For $p = 1$, this is equivalent to issuing a request for the entire block to all the replicas and downloading from the fastest. In a way, it's an alternative to probing the network to find the fastest machine, only with higher bandwidth utilization. On the other hand, if $p = r$, then this is essentially the same as striping the file across the r servers. Performance is dependent on none of these servers being particularly slow. However, if $1 < p < r$, it is not just a simple matter of taking the data returned by the first p machines to respond. In order for this to be possible, it can not matter which blocks are returned, as long as enough data is returned. This is obviously not true of a normal block, since each segment will contain unique data. This situation is true, however, if the system utilizes erasure codes. The main idea behind erasure codes is to take a block or file composed of k packets and generate a n packet encoding, where $k < n$. The original file can then be recreated from any subset of k packets [3]. CFS currently does not use erasure codes, but it is worth studying whether there is a benefit to using them, at least in regards to end-to-end performance.

Regardless of which method is used to choose the parallel machines, the performance of the parallel retrievals depends a great deal on the characteristics of the replicas. If there are five replicas but only one fast one, it may be better to download only from one machine. On the other hand, if all five replicas are fast, it might prove optimal to download 1/5 of the block from all five servers. Unfortunately, it is impossible to know in advance which of these conditions is true; if it were, selection would not be necessary. While it is not possible to find *the* optimal values for p and r that will apply in any condition, it may be possible to find values that will lead to good performance most of the time. This thesis strives to experimentally evaluate each of the above techniques, and to resolve some of the issues associated with each method.

Chapter 4

Experiments

This thesis evaluates the different methods and issues of server selection described in the previous section by running experiments on an Internet test bed of 10-15 hosts. The objective was to use this testbed to generate conclusions that could be implemented and tested later on a larger deployed network. While the tests were designed to address specific issues that would arise in Chord and CFS, they were not run on Chord servers or within a Chord network. Therefore, some of the conclusions drawn from these studies could apply to other systems with similar parameters, and are not necessarily specific to Chord. Correspondingly, these conclusions require further verification and refinement within Chord.

4.1 Experimental Conditions

All tests were conducted on a testbed of 10-15 Internet hosts set up for the purposes of studying Resilient Overlay Networks(RON), described further in [2]. These machines were geographically distributed around the US and overseas at universities, research labs, and companies. They were connected to the Internet via a variety of links, ranging from a cable modem up to the five university computers on the Internet 2 backbone. Most of the hosts were Intel Celeron/733-based machines running FreeBSD with 256MB RAM and 9GB disk space. While some of the machines were more or less dedicated for use in this testbed, others were "in-use" machines. To a certain extent, this testbed attempts to reflect the diversity of machines that would be used in a real peer-to-peer network.

Name	Connection	Description
Aros	T3-	ISP in Salt Lake City, UT
CCI	T1?	.com in Salt Lake City, UT
CMU	T3 + Internet-2	Pittsburgh, PA
Cornell	T3 + Internet-2	Ithaca, NY
Kr	unknown	South Korea
Lulea	unknown	Lulea, Sweden
Mazu1	unknown	MA
MIT	T3 + Internet-2	Cambridge, MA
Msanders	T1	Foster City, CA
Nc	cable modem	Durham, NC
Nl	unknown	Netherlands
NYU	T3 + Internet-2	New York, NY
Pdi	unknown	Silicon Valley, CA
Sightpath	unknown	MA

Figure 4-1: RON hosts and their characteristics

4.2 Chord Layer Tests

4.2.1 Evaluating the use of past latency data

Server selection at the Chord layer is mainly focused on finding the machine with the best latency. Each step of the lookup process takes little more than one round trip time for the *closest-preceding-nodes* request to be relayed to the server and the information for the next node on the lookup path to be relayed back. Ping times are thus closely correlated to the performance of the lookups. At the same time, since the time for a given step in the lookup path is roughly equivalent to the ping time between the two nodes, pinging all the machines first to choose the fastest one adds a significant overhead to the overall performance. Instead, the selection mechanism must rely on past performance data to choose a node.

In order to test whether past ping data is an accurate predictor of future performance, tests were run to determine how much latency between machines varies over time. If latency varies widely, past data would be a fairly useless metric unless it was taken very recently. If latency holds fairly steady, then the past ping times stored in each node would be sufficient to select the next server on the lookup path.

4.2.1.1 Methodology

The necessary data was collected using perl scripts running on every machine in the testbed. Every five minutes, the script would ping each machine in a list that included every other server in the testbed, plus an additional ten Web hosts, and record the ping results in a file. The Web hosts included a random assortment of more popular sites, such as yahoo, and less popular sites in both the US and Canada. They were included in the test to increase the data set, and to test the variance in latency for in-use, commonly accessed servers. These scripts were run several times for several days on end to get an idea of how ping times varied over the course of a day as well as over the course of different segments of the week. The resulting ping times between each pair of machines were then graphed over time, and the average, median, and standard deviation for the ping times were calculated. Since each ping only consisted of one packet, sometimes packets were lost between machines. Packet losses are graphed with a ping time of -1, but were not included in calculating the statistics.

One point to note is that these experiments used ICMP ping, which may not be exactly representative of the actual behavior Chord will experience. Chord's lookup requests use a different communication protocol, which may therefore have different overheads and processing than the ICMP packets. Another experiment that may be interesting to conduct is to graph the round trip latencies for an actual request using the desired communication protocol.

4.2.1.2 Results

For the most part, results tended to look much like that in Figure 4-2. While individual ping packets sometimes experienced spikes in round trip time, overall the round trip latency remained steady over time. Some pairs of machines experienced greater variability in the size of the spikes, while others had far less variation. In general, machines with different performance tend to have fairly different ping times, and even with these variations, it is fairly obvious which are relatively better. Those that are closer in ping time tend to have more or less equivalent performance, so it does not matter as much if at a particular point in time, one has a better ping time than the other.

Several of the graphs depicted shifts in ping times such as those shown in Figure 4-3, and more drastically in Figure 4-4. Such shifts in latency are probably fairly common, and

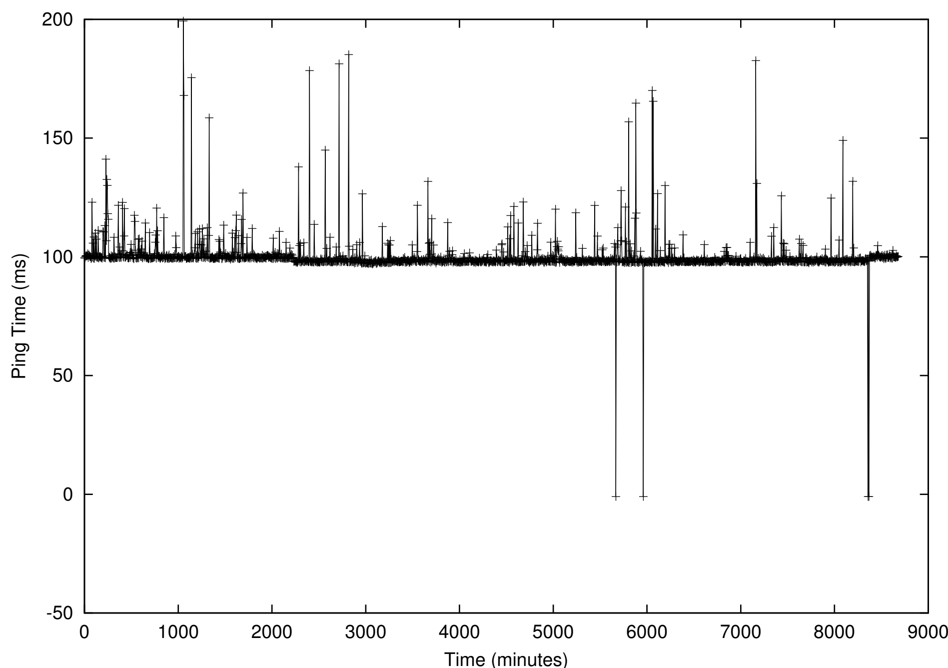

Figure 4-2: Sightpath to Msanders Ping Times

indicate rerouting somewhere in the Internet between the two machines. While it is evident that there will be a period of instability during which stored past latency data will not accurately predict future performance, for the most part, latency holds steady before and after the change. Therefore, while past latency data will not always be the best indicator, it is good enough the majority of the time.

However, on occasion results look like Figure 4-5. Each of the large spikes in the graph seems to correspond to a different day of the week, indicating a great deal of variability during the daytime hours. The machines in question are connected to a cable modem in the US and at a university in Sweden. The higher spikes on weekends might indicate greater usage of cable modems when people go home on weekends; however, none of the other results to or from the cable modem show similar spikes. In fact, none of the other results to or from the Swedish machine show similar spikes either. The route between these two particular machines may be unusually congested and be more significantly impacted by traffic over the Internet than the other routes. This graph is not very characteristic of latency trends in this set of experiments, including the ping times to the web hosts that were not part of the testbed.

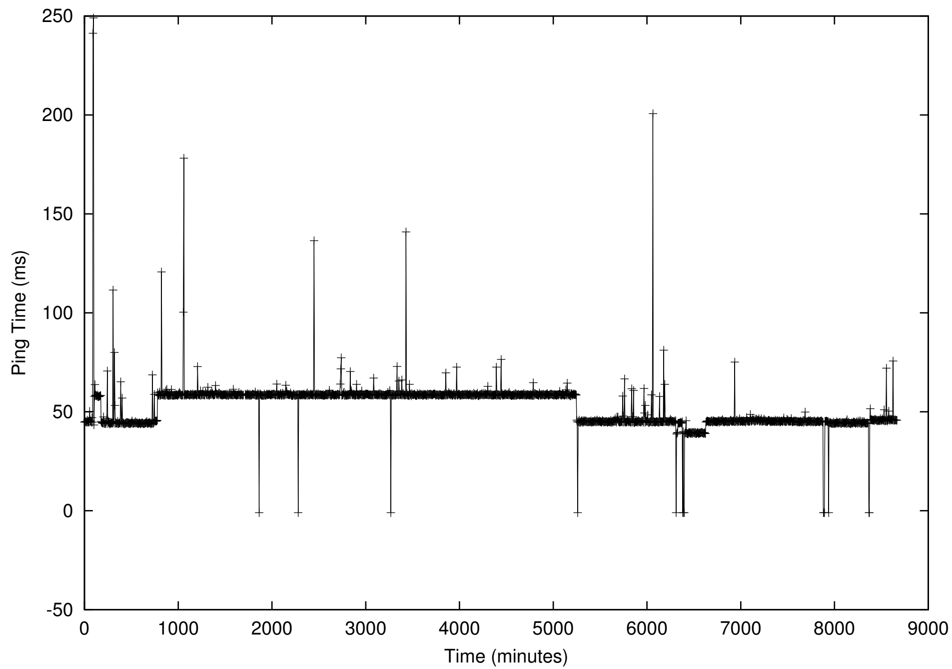

Figure 4-3: Cornell to Sightpath Ping Times

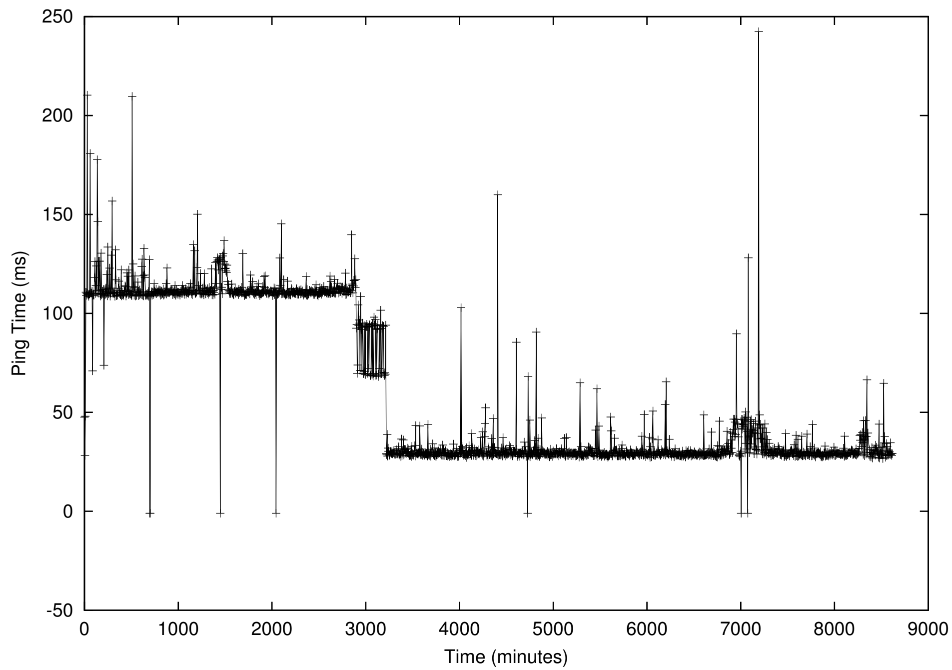

Figure 4-4: Mazu to Nc Ping Times

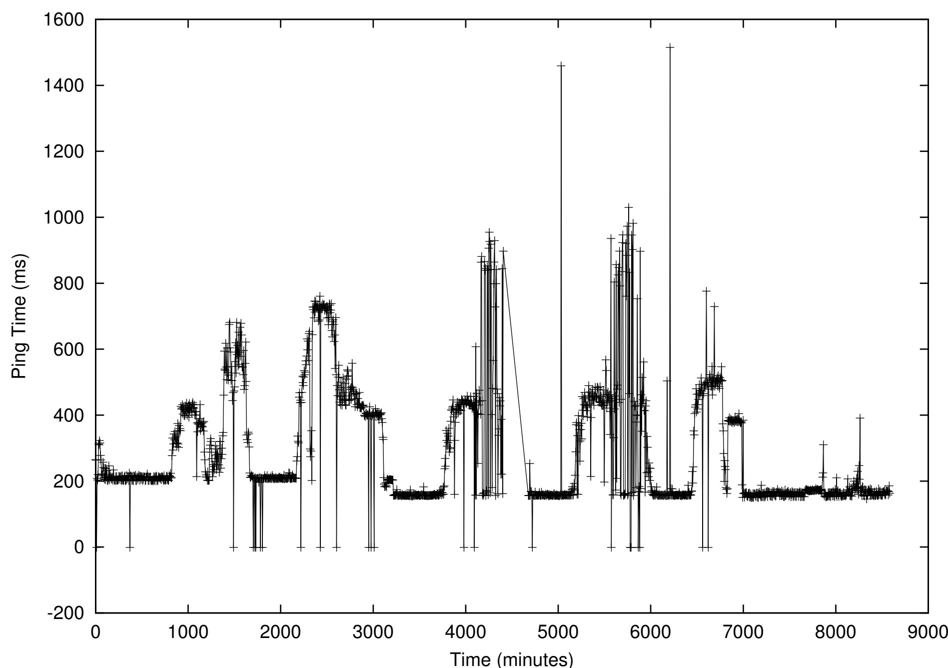

Figure 4-5: nc to lulea Ping Times

4.2.2 Triangle Inequality

The Chord protocol has the option at each step of the lookup process to select from a pool of servers, some further in ID space than others, some closer in network distance than others. The proposed server selection method is to select a server closer in network distance (using round trip latency as the distance metric) at each hop, in the hopes of minimizing the overall latency of the lookup process. Due to the fact that the data used to determine network distance is actually an indirect measurement made by another node, this approach is based on the assumption that the triangle inequality holds true most, if not all, of the time. The ping times collected in the previous experiment were used to test this assumption.

There are actually three variations of the triangle inequality that were tested, each increasingly specific to this problem.

4.2.2.1 Simple Triangle Inequality

The first examined is what is normally meant by the triangle inequality, and is the only variation that actually involves a triangle. It states that if machine A is close to machine B, and machine B is close to machine C, then machine A is also close to machine C. This

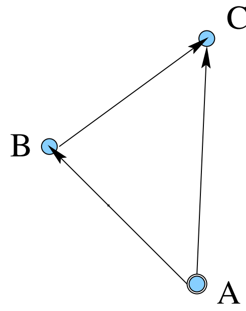

Figure 4-6: Simple Triangle Inequality

can be expressed as follows:

$$Ping_{AB} + Ping_{BC} \geq Ping_{AC}$$

This is generally useful since each step of the lookup algorithm essentially involves asking an intermediary node to find a third node.

In general, the triangle inequality is expected to hold. A direct path between two nodes should be shorter than an indirect path through a third node. However, network distances are a bit more complicated than physical distance. If the direct path is congested, lossy, or high latency, it may be ultimately slower than an indirect path made up of two fast links. This is somewhat akin to taking local roads to get around at traffic jam on a highway. Also, routers in the network may handle traffic in an unexpected manner, giving certain traffic higher priority than other traffic. Therefore, the triangle inequality should not hold all of the time.

4.2.2.2 Triangle Inequality Methodology

The simple triangle inequality was tested by collecting the ping data from the previous experiment in section 4.2.1, and finding all combinations of three machines where ping data existed between all three machines. Using this data, the components of the inequality were calculated, and the percentage of time the inequality held for all combinations of three machines was determined.

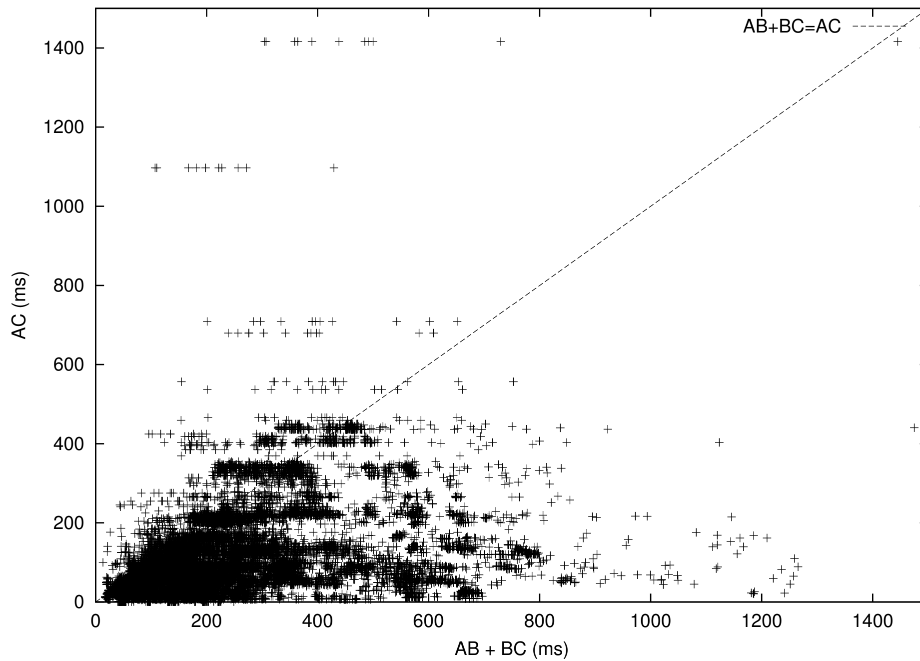

Figure 4-7: Triangle Inequality: AC vs AB+BC

4.2.2.3 Triangle Inequality Results

All told, 20 different samples of ping times taken at different times over the five day period were examined. Each of these samples consisted of a complete set of ping data taken from each machine at approximately the same time of day on the same day. Of the 30,660 combinations of 3 machines examined using these samples, the triangle inequality held for 90.4% of them.

Figure 4-7 pictorially illustrates how well these samples fit the triangle inequality. The diagonal line indicates the case when $Ping_{AB} + PingBC = Ping_{AC}$, and the triangle inequality holds for it and everything under it. This graph indicates that for the majority of cases, even those cases that do not fit the triangle inequality are not that far off. This is fairly consistent with the expected results and indicates that the triangle inequality can be assumed to hold in general.

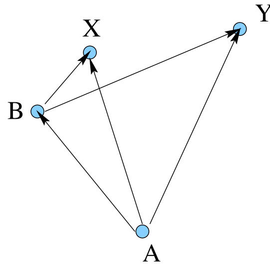

Figure 4-8: Relative Closeness

4.2.2.4 Relative Closeness

The second form of the triangle inequality is more restricted and specific. Instead of just determining if C is close to both A and B, this also tests relative closeness. Will asking an intermediate node for its closest machine also find the machine closest to the origin? In other words, if B says that it is closer to machine X than machine Y, will A also find that it is closer to X than Y? This is determined by calculating if the following is true:

$$(Ping_{AX} < Ping_{AY}) == ((Ping_{AB} + Ping_{BX}) < (Ping_{AB} + Ping_{BY}))$$

This is equivalent to:

$$(Ping_{AX} < Ping_{AY}) == (Ping_{BX} < Ping_{BY})$$

This expression is more relevant to Chord lookups, since the proposed selection method uses indirect data to select close machines.

From a purely logical standpoint, assuming latency is directly correlated to geographic distance, if A, B, X, and Y are randomly chosen out of a set of geographically dispersed servers, there should not be any real correlation in relative closeness between A and B. A has just as good a chance of being close to X as it is to Y, and B, quite independently, also has an equal chance of being close to one or the other. Therefore, the relative closeness expression above should be true about 50% of the time.

4.2.2.5 Relative Closeness Methodology

For this test, a list of all machines *to* which there was ping data was compiled, including all 15 machines in the testbed and the 10 additional Internet hosts. These machines were used to generate all possible combinations of X and Y. Then, a different list of machines for which ping data *from* those machines was available was also compiled and used to generate all possible combinations of A and B. Using this data, the percentage of trials for which the above expression held was calculated for all combinations of A, B, X, and Y.

4.2.2.6 Relative Closeness Results

In this case, some 783,840 combinations of A, B, X, and Y were generated using the same 20 samples of data from section 4.2.2.3. Relative closeness held for 67.2% of ABXY combinations. This correlation is better than expected, and suggests that indirect data from an intermediary node may be useful for determining relative proximity from the origin server. The question may be raised as to why the correlation is better than random, since A, B, X, and Y were essentially randomly chosen. The 50% correlation discussed previously assumes a network in which the servers are evenly dispersed, and where latency is directly correlated to geographical distance alone. In an actual network, this may not be the case. The network is not arranged homogenously, with each server behind identical links and at regular intervals from each other. Instead, some machines may be behind very high speed, high bandwidth links in a well-connected AS, while other machines may be attached to high latency links that are somewhat removed from the backbones and peering points. While geographic proximity does contribute to latency, it is not the only factor involved. Therefore, some servers may frequently be relatively faster, while others are relatively slower, regardless of where the client is located.

This can be seen within the particular testbed being used in these experiments. Three of the machines in the testbed are in countries outside of the US, one in Korea, one in the Netherlands, and one in Sweden. The latencies to these machines are significantly higher than to any other machine in the US. Therefore, if one of these machines was X, then A and B will always agree that Y is closer. The results are also biased, because four of the machines are connected to each other via the Internet-2 backbone, which has considerably higher bandwidth and lower latency than the regular Internet backbone. Most of these

servers also have good connections to the Internet, and therefore are relatively fast to the other servers in the testbed. A specific example would be NYU, which is never slower than the fifth fastest for any of the other machines. In fact, a quick examination of ranked ping times for each machine indicates that the relative rankings of the machines do not differ that much, although the results tend to be biased based on geographic region.

If these tests were run on a different testbed, the results may change. A less diverse testbed would not have as defined fast and slow machines, and would perhaps have a lower correlation. In this particular testbed, however, relative closeness holds with good correlation.

4.2.2.7 Chain of Machines

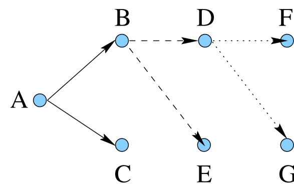

Figure 4-9: Chain of Length 3

The third test conducted investigated a more complicated scenario of a chain of the choices studied in the previous example. Since the actual lookup will involve several steps of selection decisions, we seek to determine whether the "right" decision can still be made with this degree of indirection.

Despite the indirection involved in these chains, what this experiment ultimately tests is essentially the same as the previous relative closeness tests. Once one of D or E is selected, then it should pick the "right" one of F or G with the approximately the same probability as it did in section 4.2.2.6. Since the intermediate node is not randomly chosen, the probability may not be exactly the same. However, since each indirect server was chosen to be close to the original, and since both relative closeness and the triangle inequality seem to hold with great probability, the probability of D or E selecting the right answer could potentially be even higher.

4.2.2.8 Chain of Machines Methodology

In order to test this, chains of machines were constructed such as the one in Figure 4-9. In this particular example, a chain of length 3, A determines which of B and C is closest to it, and asks that machine, say B, which of D and E is closest to it. B then asks that machine, say D, which of F and G is closest to it. This final choice is then compared to the relative closeness of AF and AG to find out if the "right" decision was made. Chains of length 2 were also studied, using combinations of A, B, C, D, and E.

4.2.2.9 Chain of Machines Results

The same 20 samples of data used in the previous tests were used to generate all possible combinations of five machines for chains of length 2 and all possible combinations of seven machines for chains of length 3. Of the 3,603,387 combinations examined for chains of length 2, 69% of them made the "correct" final selection of D or E. Of the 146,556,900 combinations examined for chains of length 3, 65% of them made the "correct" final selection of F or G. These results are consistent with the results of the relative closeness test. In fact, the results for chains of length 2 are actually better than those for relative closeness. Selecting the closer intermediate server may have increased the chances of selecting the correct final server. This is consistent with the triangle inequality. When the chain lengthens, however, the correlation drops somewhat. With the extra degree of indirection, the probability of selecting a "wrong" intermediate node (one that is slower for the origin) increases somewhat, and that intermediate node is more likely to pick the "wrong" final node, since what is close to it may not be close to the origin server. However, since relative closeness does hold 67% of the time, that intermediate node also has a good chance of selecting the "right" final node. Therefore, the drop in correlation is slight.

In fact, since the crucial step seems to be the final selection, it seems that it should not matter at all whether the closest machine is selected at each intermediate step. The "correct" final answer should be found with about the same probability regardless. This hypothesis was tested by conducting the same test, but selecting the worse of the two choices at each step until the final decision, at which point the best was selected. Sure enough, for chains of length 3, the correct final answer was still arrived at 61.7% of the time. This probability is lower than it is for selecting the best machine at each step for likely the same

reason the longer chains have a lower correlation. Selecting the slower server at each step increases the probability that that server will have a somewhat different view of the network than the origin, and will therefore pick the "wrong" server.

Since the probability of finding the "correct" server through indirect queries is about the same for selecting the best machine vs the worst machine, the next question to address is what the improvement is in overall latency. The overall latency was calculated as the sum of latencies between the original server and its chosen server at each level of the chain. The improvement in latency between selecting the worst server at each step and selecting the best server at each step was marked. Picking the best server had an overall latency that was one quarter to one fifth that of selecting the worst. Therefore, picking the fastest machine at each step does win considerably on latency, while still arriving at the correct answer with reasonably good accuracy. This test is of course a simplification of what will really happen in CFS. In CFS, it is unlikely that choosing different servers will lead to the same length chain to arrive at the destination. Therefore, additional considerations regarding the overall number of hops must also be taken into account. However, this test does provide a good indication of the benefits of this approach.

4.2.3 Chord Layer Selection Results

Overall, the results of these three tests, coupled with the study of variance in latency, indicate that the proposed selection method of picking the fastest server at each step of the lookup process based on past information should be successful at reducing lookup latency. This method requires a minimum of overhead, and should keep the variation of performance experienced by different machines in the network low. However, it is important to note that these tests only indicate that these assumptions are true in this particular testbed. While indicative, it is not necessarily representative of how often it holds for the Internet as a whole.

4.3 dhash Layer Tests

Selection at the dhash layer is directed towards improving the end-to-end performance experienced by the client during data retrieval. End-to-end performance is measured as the time from when the client issues its request to when it finishes receiving all of the expected data. To that end, although all of the tests below measure the time for the client to issue the request, the server(s) to read the data out of the file, and the server(s) to transmit the data back to the client, none of them include the time to write the data to disk on the client. In fact, none of the tests actually did write the data to disk after it was received. This decision was made with the assumption that local disk write time was the same for the various selection methods, and that the greatest variation in performance came from the actual retrieval of data across the network. The end-to-end performance measurement also did not take into account any additional processing time on the client that a particular method might actually require. Analysis and comparisons between the various methods of selection was thus purely based on network and server performance. Further analysis of additional overhead from processing, disk writes, and other costs should be done to fully compare these methods.

4.3.1 Server

For the purposes of the dhash layer related experiments, a simple TCP server was run on each host. Each host also stored a 3.5MB file for use by the TCP server. Upon startup, the server would open the file for reading and then wait for incoming connections. The protocol for contacting the server was very basic; it was designed to accept requests with a very simple syntax: "GET [size]". When it received such a request, the server would read [size] bytes of data from the open file into a buffer and send the buffer back over the connection to the client. The server never explicitly shut down a TCP connection until it detected that the client had closed down the other side of the connection. This was so the client had the option to send additional requests over the link without having to establish a new TCP connection and go through slow start again. The client could also just abort any ongoing connection it no longer needed, and the server would eventually detect the disconnection without the need for an explicit message from the client.

4.3.2 Ping Correlation Test

Although the size of the blocks used in CFS is relatively small, there is not necessarily a direct correlation between latency and bandwidth. A given server could be relatively close to the client, thus having low latency, but also be behind a small link, cutting down its bandwidth. If the block size is less than the size of one packet, then ping time may be a good indication of performance. However, the larger the block size, the less likely there is to be a good correlation. Thus, it is useful to determine the actual correlation between ping times and retrieval times within this testbed. How often does the fastest ping actually correlate with the fastest machine, and what is the penalty from downloading from the fastest pinging machine? How does this penalty vary with size? At what size does the correlation begin to weaken?

4.3.2.1 Ping Correlation Methodology

The basic test designed to answer these questions cycles through a list of servers, first pinging the machine and then actually retrieving data from that machine. Both the ping time and the retrieval time are recorded. This script uses a system call to the actual ping utility, specifying 1 ping packet and a max wait time of 3 seconds. When the test finishes running through all the machines, the machines are ranked by ping time and by retrieval time and the ranked lists are printed to a file for further analysis. This test was run for request sizes ranging from 1 byte up to 64KB, with ten iterations of the test at each size. There were also two other versions of this test. For one, all possible combinations of five machines were generated and the test was run once for each possible combination and size. The third version of the test used sizes of 1KB through 2MB, doubling each time, and requested from all machines once, but then subsequently generated all possible combinations of five machines to study. The reason behind studying different combinations of the machine was to vary the pool of ping and retrieval times, since some groups of machines may exhibit more variation than others. These tests were also run using each machine in the testbed as a client.

4.3.2.2 Ping Correlation Results

The data from all the tests run using the first two versions of the test was compiled together for analysis. The data for all the tests run using the third version of the test was compiled separately, since the range of sizes tested differed from the previous tests. The analysis for each set of data was the same, however. For each individual test with its rankings of machines, the ratio of the retrieve time from the fastest pinging machine to the fastest retrieve time was calculated. In order to determine if this ratio is actually significant, the ratio of the retrieve time from the slowest pinging machine to the fastest retrieve time was also calculated. In the final version of the test, a third ratio was computed of the retrieve time from a randomly selected machine to the fastest retrieve time. This was added to show the relative improvement in retrieval time derived from pinging as opposed to simply picking a machine. Ultimately, ratios from individual tests and different clients were averaged together to produce an overall ratio for each request size. This ratio essentially correlates to the penalty derived from selecting the machine with the fastest ping — the ideal is a ratio of one, where the fastest pinging machine is also the fastest retrieving machine. When the ratio goes above 1, then the retrieve time for the fastest pinging machine is worse than optimal. However, this machine could be the second fastest and still have reasonably good performance, keeping the ratio close to 1.

The overall average ratios for best ping time and worst ping time compiled from the first and second tests described above were graphed in Figure 4-10. The ratios for best ping, worst ping, and random from the third test were graphed in Figure 4-12. The best ping time ratios from Figure 4-10 and Figure 4-12 were graphed individually in Figure 4-11 and Figure 4-13, respectively. For both sets of data, results from thousands of combinations of servers from 13-14 clients were compiled. In both cases, the ratio for the fastest ping machine's retrieve time to the fastest retrieve time is quite close to 1, indicating that ping chose a machine that had the fastest, or close to the fastest retrieval time the majority of the time. Figure 4-11 and 4-13 indicate that for file sizes below 64KB, the ratio never gets higher than 1.1. Even for file sizes up to 2MB, the ratio stays under 1.5. Ping time therefore appears to have a good correlation with actual end-to-end performance and is a good metric to use for data of these sizes.

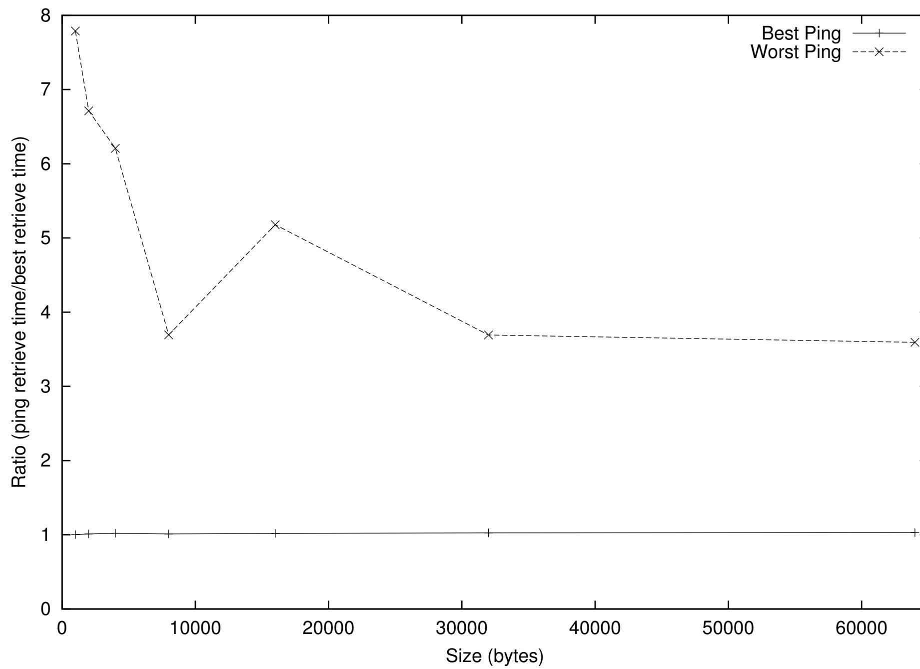

Figure 4-10: Best and Worst Ping Ratios for Tests 1 and 2

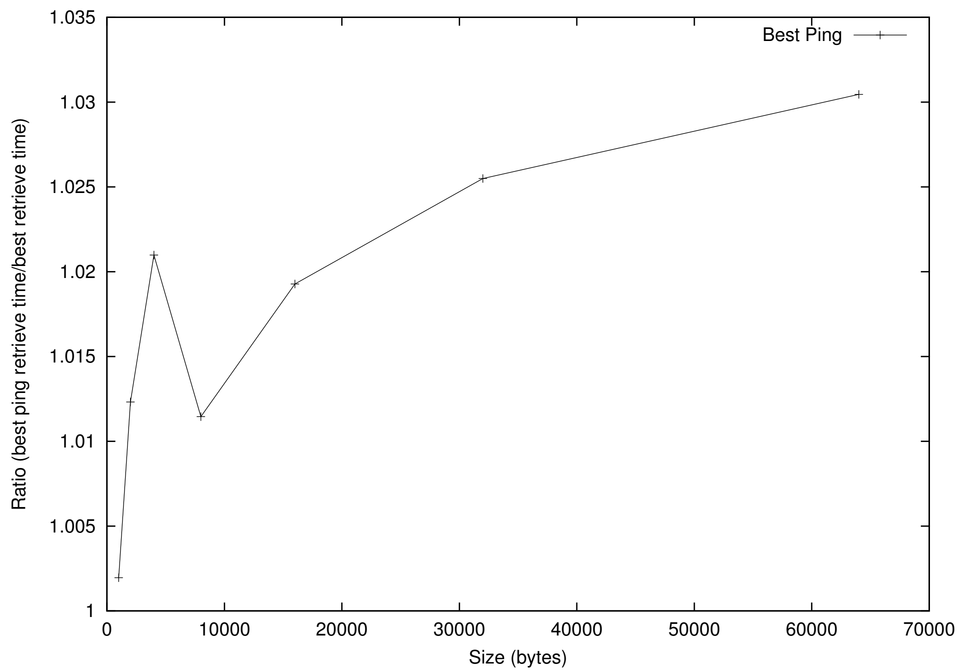

Figure 4-11: Best Ping Ratio for Tests 1 and 2 (same as in figure 4-10)

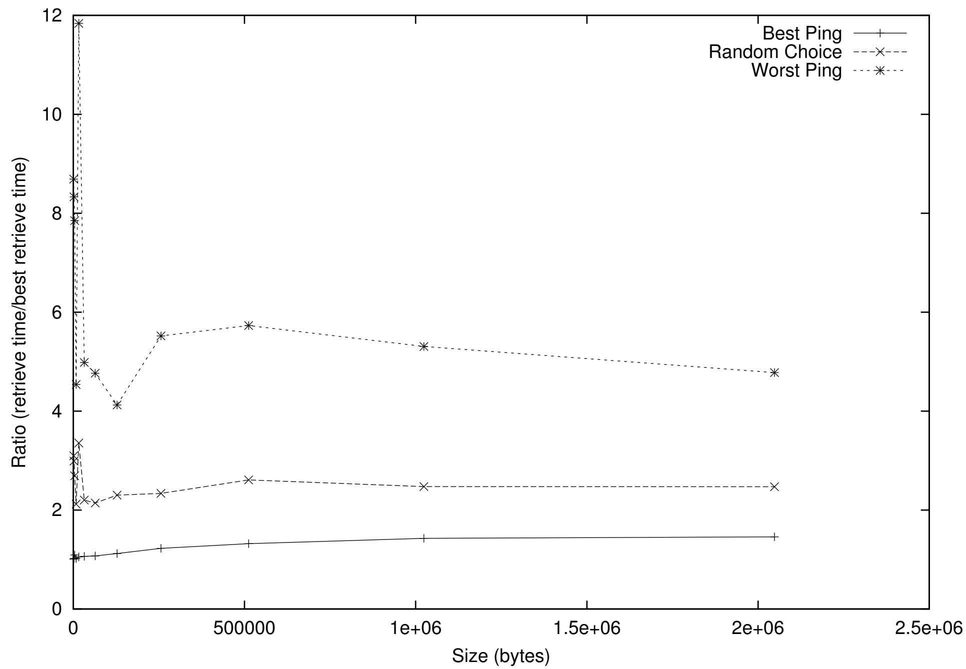

Figure 4-12: Best and Worst Ping Ratios for Test 3

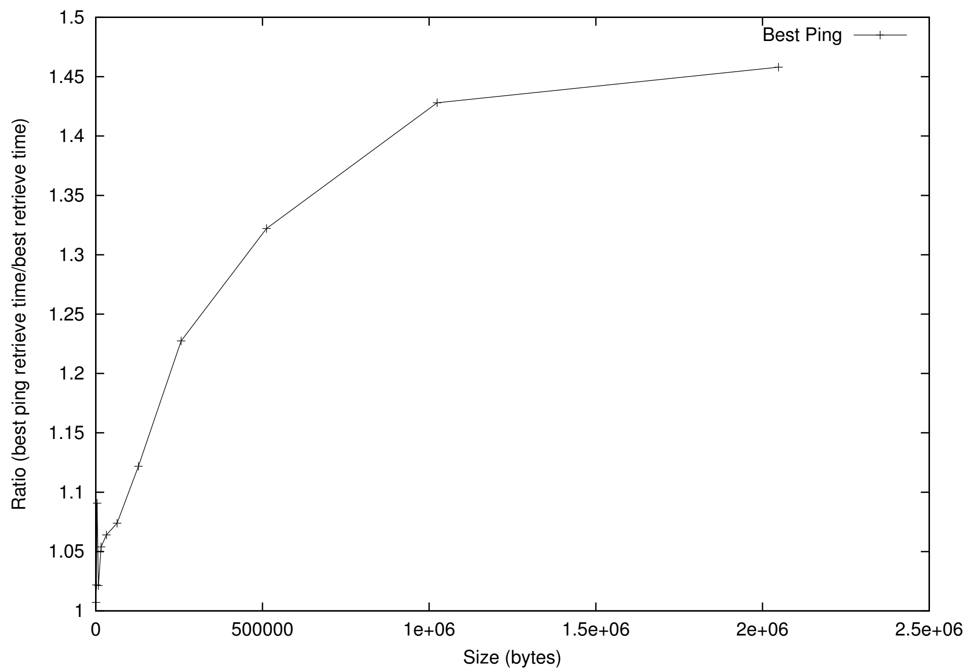

Figure 4-13: Best Ping Ratio for Test 3 (same as in figure 4-12)

4.3.3 Selection Tests

Although ping appears to select a high performance server for data retrieval with good probability, the overhead for the initial ping probe may cause another method to yield better overall performance. Since the primary concern of server selection is overall end-to-end performance, several different server selection techniques were studied, including random selection, ping, and parallel download. Bandwidth probing was not studied in any depth since for block sizes as small as those used in CFS, the bandwidth probes would be too small to yield useful information beyond a ping. There were also no specific experiments designed to test the past performance method. However, the results of the ping variance, triangle inequality and ping correlation experiments test the main points of concern for this method.

Each of the following programs were implemented separately to test different selection techniques, and then run together in a Perl script that will be described later. With the exception of the test to determine the correlation between the best ping time and the fastest server, which was implemented in Perl, all the programs were implemented in C. Most of the programs were based around asynchronous events and callbacks rather than threads. Each program was designed to make just one request per execution so that it could be utilized in a variety of different tests with different arguments.

4.3.3.1 Random Selection

The random selection program was designed to test the performance for requesting data from a randomly selected server. It takes in as arguments the size of the block to request and a list of machines to select from. A server is selected by generating a random number between 0 and *NumMachines - 1*, and finding the machine in the server list with that index. The output of the program is the time it takes for the client to generate the random number, the request to be made, and the data to be returned.

4.3.3.2 Latency Probing

The ping program was designed to test the performance for probing the servers and then requesting the data from the server with the lowest latency. The ping program does not use the ping utility, precisely, but instead issues an ICMP echo packet to each machine to probe

the latency between the client and the potential servers. This ping probe times out after a minute, in which case the program just aborts with no output printed. Otherwise, the program waits for the first ping packet to return, and issues a request for the appropriate size to that machine. The time printed in the output measures the period of time from before the program resolved the hostnames and sent out pings until after the data was fully retrieved. This program is actually a bit more complex than this, as will be more fully described in section 4.3.4.1.

4.3.4 Parallel Downloading Tests - Methodology

An alternative to picking one fast machine to download from is to download parts of the block in parallel from multiple machines. This allows the client to take advantage of the aggregate bandwidth of the combined servers while downloading smaller amounts of data from each machine, thus increasing overall performance. Although aggregating bandwidth is useful, the client must be careful not to request from so many servers at the same time that the bandwidth capacity of the link is exceeded. Also, if the client chooses to download a portion of the block from a slow machine, the overall retrieve time is limited to the speed of that machine. Therefore the degree of parallelism, as well as the machines to retrieve from, must be carefully chosen.

4.3.4.1 Ping

Assuming the block is replicated on n servers, one way to select the k to retrieve from is to ping all the machines and take the k fastest. The program described above in section 4.3.3.2 is not restricted to only retrieving from the machine with the fastest ping. It actually takes in two parameters additional to the list of repliacs: the number of machines to retrieve from and the size of the total request. Ping probes are issued to all n replicas in parallel, and the client requests a different $\frac{TotalRequest}{k}$ segment of the block from the first k machines to respond to the ping. The total time of this test is measured from before the ping packets are issued until the last portion of data has been retrieved. The test described in section 4.3.3.2 is simply this program run with $k = 1$.

4.3.4.2 Direct Parallel Download

If there aren't too many replicas, and the block size is reasonably small, another possible method is to issue a request to all n replicas and using the data from the first k to respond. As was described in section 3.2.4.1, this is not entirely equivalent to the parallel retrievals described in the previous section. When $k = 1$, this method is essentially another way to find the fastest machine without probing the network. When $k = n$, this is the same as striping, and in fact is a better test for striping than setting $k = n$ for the ping program, since there is no need to probe when requests are being issued to all the replicas. However, when $1 < k < n$, this method is more roughly equivalent to using erasure codes with parallel downloads than simple parallel downloading. With erasure codes, the client would request a different segment of the block from each of the n replicas and be able to reconstruct the block from the first k segments it receives. Using erasure codes is slightly different than the above described methods since the block will not be fully replicated on the n replicas, but instead be split into n encoded segments. If CFS were to use erasure codes, there would need to be further study into the overhead of encoding and reassembly, as well as management issues of how the segments are maintained as nodes leave and join the network. This thesis only seeks to discover if there are any performance benefits to using this type of retrieval, and therefore ignores any overhead associated with encoding. It also assumes that the client is not receiving any duplicate packets, and that the first k are unique and sufficient to reconstructing the block.

To this end, the program designed to test parallel downloads takes in the same parameters as the ping program - the number of machines to download from (k) and the size of the overall request (*TotalSize*). It then opens TCP connections to all n machines, and issues a request for $\frac{TotalSize}{k}$ to each machine. When it has finished receiving data from the first k machines to respond, it stops timing and prints out the total time of the test.

4.3.5 Combined Tests

Although it is interesting to study each of the above methods individually, the real question was the relative performance of the various methods under different conditions. These conditions include varying the size of the total request, changing the value of k, and seeing how well each method performed from clients with different characteristics to differing

groups of servers. Some of the methods might work better for clients behind slow links, and others with fast clients. Similarly, if the replicas were all behind high-bandwidth links, striping the block might have better performance than if only one or two replicas were fast. Since in the real network these conditions are unknown, the selection mechanism cannot be adapted to real-time conditions. Therefore, a selection method needs to be found that may not provide the best performance in every situation, but provides a reasonably good performance in most.

4.3.5.1 Combined Tests Methodology

In order to evaluate the different server selection methods, a perl script was written to run the previously described tests in various conditions. The script takes in a list of machines and generates all possible combinations of five servers from the list. The number five was chosen since it seemed to be a reasonable number of replicas. For each of these combinations of machines, the script iterated through request sizes ranging from 1 byte to 64KB (doubling in size each iteration). For each size, the random, ping, and parallel tests were run with k varying from 1 to 5 for the ping and parallel tests. All the results from these tests were written to file for later analysis. Each of the hosts in the testbed in turn acted as the client, with the rest of the hosts making up the pool of available servers. Results from early tests indicated that there was not much difference between 1 byte and 1KB, since they both less than a size of a packet, so later tests did not use a request size of 1 byte. For various clients, the number of available servers ranged from 9 to 13. All the individual tests were run serially so that the results were not corrupted by interference between the different tests. Overall, these tests reflect results for different sizes, clients, and combinations of servers.

4.3.5.2 Combined Tests Results

In order to get a sense of the overall performance of each of these methods, irrespective of external conditions, all the performance data from the various tests were compiled together and the median test times for each method at each size was determined. The resulting times are representative of "typical" performance for each metric, without taking into account the particular clients, servers, or time of day the test is done at. Since the objective is primarily to determine relative performance of each method, the specific times are less relevant to this study.

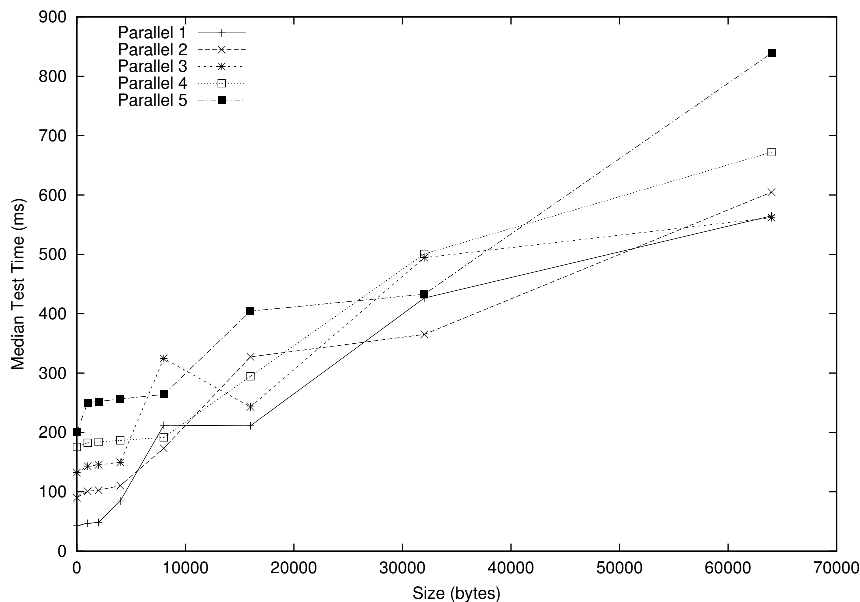

Figure 4-14: Parallel Retrieval Results

Figure 4-14 shows the results for direct parallel downloading without first probing the servers. Each line represents a different degree of parallelism. *Par 1* represents the case of requesting the whole block from all five servers, and using the first to finish returning the data. *Par 5* shows the results of striping the block across all five machines and retrieving 1/5 of the block from each server. The other lines *Par k* represent varying degrees of parallelism, where a fraction of the block $\frac{filesize}{k}$ is requested from all servers, and the process is complete when k servers return the data. From the graph, it is apparent that the best results were derived from downloading the block from only one or two machines. Whether one or two machines should be used is dependent on the size of the block in question.

The results for pinging the machines first to select the servers to download from are shown in figure 4-15. The overall trends and relative performance of the various degrees of parallelism closely matches that of directly downloading the block in parallel. In this case, pinging and downloading from just one machine is almost invariably better than other methods, although downloading from two machines has only somewhat worse performance. Striping from five machines is the worst for both methods.

In order to determine that the overall results were not skewed due to the results of one or two clients, the results from each individual client were also examined. For 11 out of the

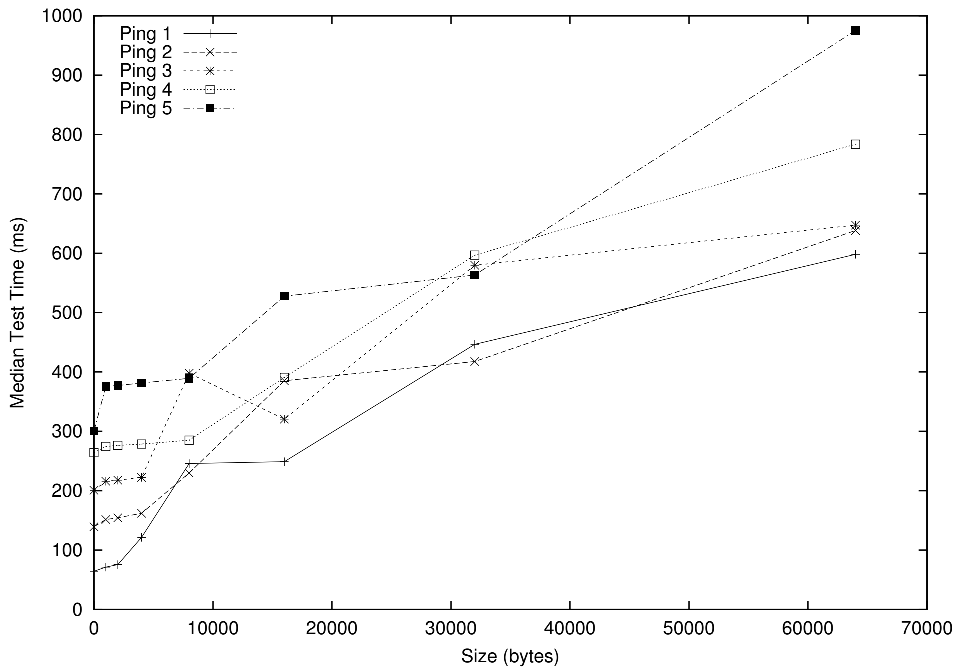

Figure 4-15: Ping Probe Results

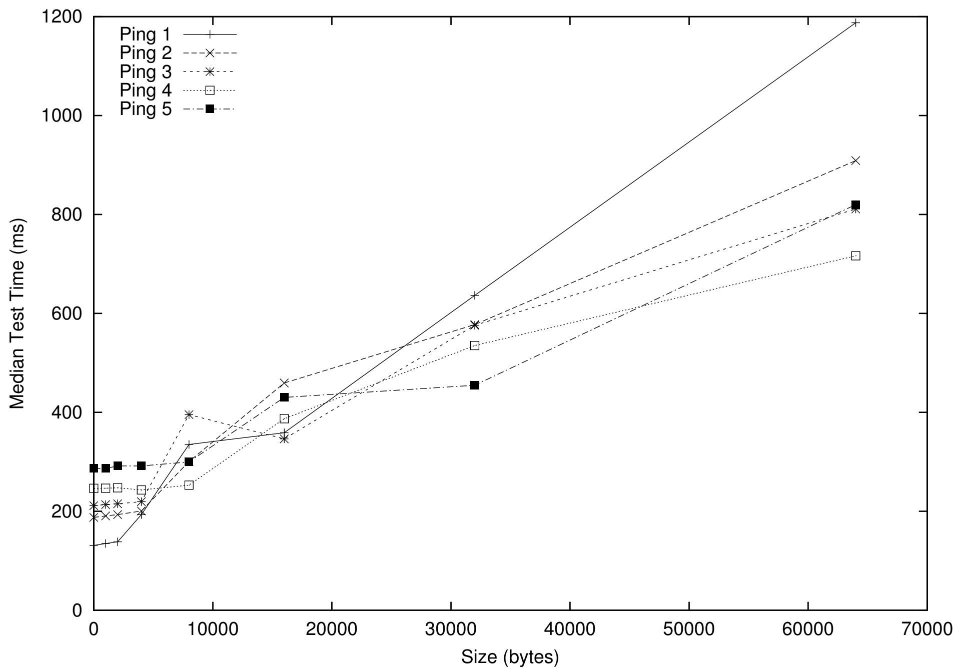

Figure 4-16: Aros Ping Probe Results

13 clients, the results paralleled the overall results, indicating that the best performance was to be derived from downloading from just one or two machines. For the other two, striping or downloading from four machines actually yielded better performance. However, as can be seen in figure 4-16, the performance derived from striping was not significantly better than that of downloading from just one machine, especially for smaller block sizes. Therefore, the policy of downloading the entire block from the fastest machine would yield the best performance in most cases, and only marginally worse performance for a minority of clients.

These results at first are surprising in light of the fact that previous work ([3],[19]) has shown significant improvements in download times through parallel access. However, that work was done with large files. For blocks of the size being examined in this test, performance of parallel downloading using TCP connections is strongly influenced by TCP-related overheads. When an already small block of data is split into even smaller fractions, the number of packets required to send that data is minimal. Therefore, TCP connection set-up time dominates the retrieval time and the window never opens enough to reach path capacity. This time is not reduced in any way by the striping of the data. Striping data only increases the number of TCP connections being established at the same time. In the event of a packet loss, TCP's congestion control mechanisms will cause the window size to decrease and the data throughput to be reduced. With five TCP connections instead of one, the chance that there is a packet loss on any of these links is increased. Therefore, the overall performance is reduced since any packet loss will require the client to have to wait longer for a response from one of the servers. CFS already provides parallelism in its block-level store and the way it distributes data around the network. Further parallelism in the server selection step does not add any optimizations to the overall end-to-end performance of the block retrieval.

While the above graphs illustrate the relative performance of different degrees of parallelism for each method, the different methods also need to be compared to each other. Figure 4-17 plots the best ping and parallel methods along with the random selection method. The results for the ping and direct parallel methods closely mirror each other, with ping being slightly worse due to the overhead of the ping probe. Both methods produce a significant improvement in overall performance compared to random selection. However, both methods also incur a certain degree of overhead that must be considered in the choice of a

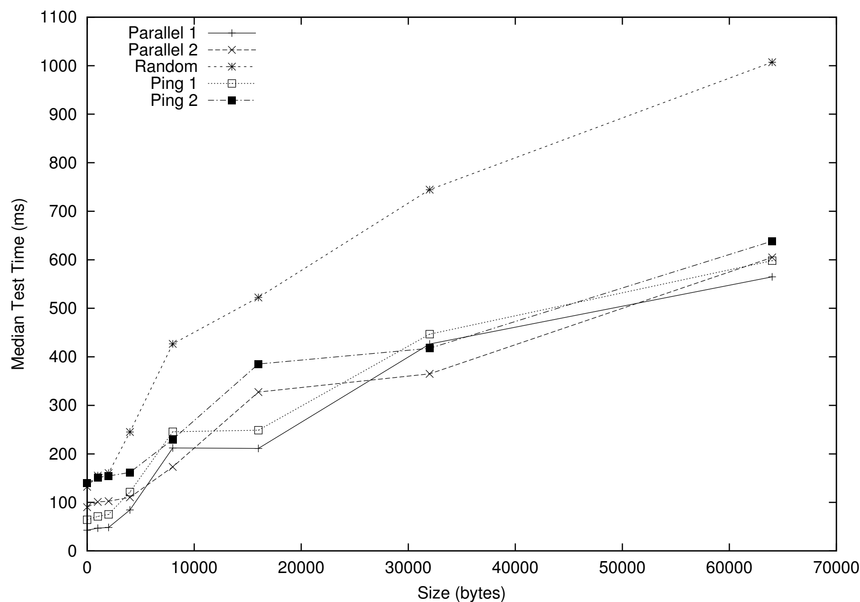

Figure 4-17: Random, Best Ping, Parallel Results

selection algorithm. Direct parallel downloads increases bandwidth usage, much of which is ultimately unproductive. Of five connections established for each block, only one is really used. In the meantime, unnecessary data is being transferred over the network through the other connections. Since the block sizes are so small, this bandwidth usage is minimal for the transfer of an individual block. However, real file accesses in CFS will require the retrieval of many blocks in order to reconstruct the file. If each of these retrievals uses its share of unnecessary bandwidth, the costs begin to increase. The probes used in the ping method add a latency cost to the end-to-end performance of a retrieval, and increase the number of packets being sent in the network at a given time. However, these additional packets occupy less bandwidth than parallel downloads do, and the overhead due to ping is relatively insignificant compared to the amount of performance gain derived from using this method. Factoring in the results of the ping correlation test, using ping probes to select the server from which to download a complete block is the best selection method of the three tested. However, further tests should be conducted to compare the performance of using ping probes to using past latency data. If the performance is comparable, then using past latency data may be a better overall method since it imposes no additional overhead and does not require additional mechanisms beyond those used at the lookup layer to implement.

Chapter 5

Related Work

The idea of improving performance of a distributed information service through replication has been a subject of interest in many past studies and systems. Although past studies have been primarily focused on improving performance in the World Wide Web, other studies apply to different kinds of distributed systems. While the specific system influences the direction and specific conclusions for each study, the concepts in question are relevant to any replicated system. This section will first examine relevant server selection work in the Internet in general, and then focus on selection within other peer-to-peer systems similar to Chord or CFS.

5.1 Internet Server Selection

In a replicated system, server selection can take place in one of three possible places: the server, the client, and the network (at some intermediary).

The server — most likely some central server that controls the replicas — can receive requests from clients and decide which replica to direct the request to. This is good from a load balancing perspective, since the server is very aware of what requests are being made at any given time and what servers are fulfilling those requests. However, this method requires a centralized server, which provides a central point of failure and a performance bottleneck, and is unlikely to be aware of the performance between the client and any one of the replicas. Server-side selection makes most sense in a centralized system where load balancing is the primary concern, perhaps in a cluster-based system. Since this is not relevant to CFS, server-side work will not be described here.

5.1.1 Intermediate-level Selection

The system can also have intermediate resolvers or proxies that reside between the clients and servers. These resolvers can keep track of server and network performance that is hopefully locally relevant, and select a server based on this information. These resolvers do not have to be aware of all the replicas in the network, nor do they have to handle all the client traffic. Therefore, there is no centralized point of failure, and there is some element of load distribution, if the resolvers are set up correctly.

DNS (Domain Name System) utilizes server selection and load balancing in different steps of its name resolution. When a DNS lookup request is made, for instance to an address such as www.lcs.mit.edu, the request will first go to the .edu server. From there, it will be redirected, to the mit.edu server, etc, until it finds the actual host. However, there can be more than one server for each level of the hostname. In the BIND implementation of DNS, BIND keeps track of the round-trip times for all of its queries to each server, and redirects subsequent queries to the server with the "best" round-trip time. While this method is effective for finding high performance servers, it is not very good at load balancing. BIND also addresses load balancing/sharing without considering performance. In DNS, a given hostname can map to a variety of IP addresses. When requests are made, the name server round-robins through the list of IP addresses, sharing the load equally among requests. This means, however, that on a given request, the client can be redirected to both far and close servers, and performance will vary widely [1].

In the SPAND (Shared Passive Network Performance Discovery) system, applications on clients utilize local repositories of shared network performance data to make predictions for future performance. This shared data is passively acquired; no probes are sent into the network. Instead, data from actual transactions are pooled together and shared between clients likely to have similar performance. This data focuses on application-level performance, and can include response time or Web page download time. The system does not specifically specify metrics, but instead provides a database-like interface to allow applications to specify their own metrics. The performance of this system depends on the amount of variation in network performance the clients experience due to network fluctuations, temporal fluctuations, and relevance of the data to individual clients. Stemm, et al, found that variation increased only slightly between receivers, and that performance could vary a

lot over large time frames but not as much for on a smaller time scale. However, network variation was enough that SPAND data could only effectively be utilized to differentiate among orders of magnitude change in performance [23].

A similar method with a different data-collection method, application-layer anycasting, was proposed by Fei, et al. Replicas were made part of an anycast group identified by an anycast domain name. The design centered around the use of anycast resolvers, which mapped the ADN to specific IP addresses on request. These resolvers had associated databases of server performance data that were periodically updated through server push and probing. For the former, the server would monitor its own performance and push information to the resolvers when interesting changes occurred. For the latter, probing agents, often co-located with resolvers, periodically send queries to servers to determine performance. The metrics used to evaluate performance were primarily concerned with response time and throughput. They tested this method on machines located around the US. This method, by their experimentation, had reasonable accuracy, and better response time performance than random selection or selection by number of hops. Anycasting does impose additional cost to the network; client queries to the resolver, server monitoring and push, and the probes sent by the probing agents. While anycast query resolution can be combined with DNS lookups, both probes and pushes add traffic to the network. However, they argue that the push messages are small, while probes are relatively infrequent and therefore to no impose a significant burden on the servers [26].

Heidemann and Visweswaraiah studied a variety of server selection algorithms that utilized an intermediate replica selector. The selector handled HTTP-requests from the client, selected a nearby replica, and returned a redirect message to the client. The algorithms used were random selection, domain-name-based approximation, geographic approximation, and measurement of propagation latency. Unlike this thesis, however, they did not measure overall performance of each of these methods, instead looking at the costs of the selection step itself. By their measurements, random selection and domain-name based selection had the least selection cost, while measuring latency or using geographic approximation had the greatest selection cost. Also, since the selector was not co-located with the client, its latency measurements were not as accurate unless they used source-routing, which is not always supported in the Internet [12].

Cache Resolver utilizes a combination of consistent hashing and DNS redirection to

coordinate caches and direct clients to an appropriate cache. Consistent hashing is used to map resources to a dynamically changing set of available caches. Since Cache Resolver is designed for use in the Web, it is limited by the use of largely unmodified browsers on the client side. Therefore, it uses modified DNS servers to support consistent hashing and perform the resource to cache mapping. An extension to the Cache Resolver system adds locality to the choice of caches by ensuring that users are always served by caches in their physically local regions. Users can specify their geographical region when downloading the resolver script, and the script in turn will generate geographically specific virtual names. When DNS resolves these virtual names, it will be directed towards DNS servers that are geographically local to the user [13].

Several commercial companies in recent years have based their business models around the problem of improving Internet performance through replication. The most well-known of these in recent years has been Akamai. Akamai's solution "FreeFlow" utilizes modified DNS resolvers to redirect requests to topologically closer replicas. There are several different aspects to their technology. The customer, a web service provider, will mark certain pieces of data on their web pages to be "Akamaized" (served by Akamai servers). The tags to these pieces of data will be automatically rewritten to point to Akamai servers, instead of the original server. When the web page containing this content is requested, the browser will request a lookup from the top level DNS server, which will redirect the query to one of Akamai's top level name servers. This server does some processing to determine the client's topological location and redirects the client to a lower level DNS resolver responsible for traffic near that user. Akamai also does some dynamic load balancing and real-time network monitoring to assist in their redirection decisions [22].

Another company, Cisco, has a different product to accomplish the same ends, the "Distributed Director". The DistributedDirector utilizes routing table information in the network to redirect users to the closest server, where closest is measured by client-server topological proximity and link latency. Cisco deploys agents throughout the network, both on border routers and within the network, that use the Director Response Protocol to coordinate with each other. After taking into account the BGP (Border Gateway Protocol) and IGP (Interior Gateway Protocol) routing table information about latencies, etc, a DNS or HTTP response is returned with the redirect to the "best" server for that client [9].

Most of these systems using intermediate resolvers require fairly complicated mecha-

nisms to keep the data relevant for use by the clients. Many base their server selection on geographical locality, rather than any performance metric. Most systems seem to depend on a fairly fixed set of replicas. They also require the use of additional components in the system, aside from the client and the server. If CFS were to use intermediate resolvers to do server selection, it would require either the deployment of additional, fixed machines or for some of the nodes to be designated as intermediaries. Both of these go against CFS' ability to adapt to a changing network, at least for the data retrieval step. Using intermediate nodes to do lookup, and picking the lowest latency lookup node in some ways reflects this use of intermediate resolvers.

5.1.2 Client-side

An alternative that is more relevant to this thesis is to place the server selection mechanism on the client. By locating it at the client, the selection mechanism can take advantage of locally relevant metrics to optimize end-to-end performance seen by the client.

Crovella and Carter studied the correlation between number of hops and round trip latency, finding that the two are not well correlated, and that the round trip latency is a much better indication of server performance than number of hops. They also determined that dynamic methods were more effective at reducing transfer times than static methods[6]. Additionally, they developed tools to measure both the uncongested and congested bandwidths of the bottleneck link in the path. While the predicted transfer time derived from the combination of these measurements performed well, they found that simply using round-trip latency measurements yielded very similar performance [5]. They continued to study server selection using the bandwidth probes devised in [5], with the objective to minimize cost due to the measurements. From their simulations, they determined that a predicted transfer time (PTT) derived from combining round trip time and bandwidth was more effective than other methods, but that simply taking the average of five pings and picking the fastest server had behavior that was fairly close to that of the PTT. Even sending only one ping and picking the fastest had good correlation, though correlation improved the more ping packets used. From this information, they derived the OnePercent Protocol, which sought to keep the number of bytes used to probe the network to one percent of the total request. In general, they found that this protocol yielded results that were fairly close to optimal [4].

Obracza and Silva revisited Crovella and Carter's study in 2000 to recorrelate hops to ping times. They studied both AS and network hops to determine the correlation between the two. They discovered that the correlation between AS and network hops was 70%, while the correlation between network hops and ping was found to be 50%. This was in contrast to Crovella and Carter's reported correlation of less than 10% between hops and ping. They surmised that the correlation had increased due to changes in the Internet since 1995, and that hops were potentially more useful than previously determined. However, their conclusion was that round trip time should be the metric of choice when trying to reduce the clients' perceived latency, and that it was also a lower cost metric to measure than hop count [17].

Both Sayal, et al, and Lewontin and Martin developed server selection methods that required the client to keep track of past performance data. In Lewontin's method, the client obtains a list of replica locations and keeps them in a vector, while a parallel vector is maintained containing performance information for each site. Each element of this vector is a running average of performance for each server, and is updated with each request. Server selection is thus essentially based on the best historical average [15]. Sayal's algorithm uses HTTP request latency as a measure of response time, and a list of replicas and latencies are kept. It then used one of two methods to select a server to send requests to. The first is probabilistic, selecting servers with probability inversely proportional to the most recently measured latency for the server. This allows the servers to be periodically refreshed since the requests won't always go to the same server or subset of servers. The other always sends requests to the server with the least latency, but also periodically refreshes. This requires extra overhead for refresh, but ensures that the request is always sent to the optimal server. Both these methods were found to yield better performance than most other methods [21].

Ogino, et al, of FastNet Inc, studied a server selection method that combined both network and server measurements to find the best server. They used AS (Autonomous System) path routing information to determine the logical distance of the network, and then measured the status of the network by round trip time, packet loss, throughput, and number of hops. They also measured the web server status based on load, as determined by the number of TCP connections, disk load, CPU idle status, and load averages. These factors, combined, were used to select the server. Their performance evaluations centered more on accuracy of selection (what percentage of the time was the "Best" server chosen?)

than on end-to-end performance. What they found, however, was that the round trip time was the most accurate metric for finding the correct server, while network information such as number of hops and length of path were far less useful [18]. Although they did not test it, this scheme is likely to impose a great deal of overhead, while not deriving a great deal of benefit from the extra measurements. This mechanism also involved the cooperation of both the client and the server.

Dykes, et al, empirically evaluated several different server selection algorithms in a very similar study to that described in this thesis. The algorithms of choice were two statistical algorithms, one using median latency and the other median bandwidth, a dynamic probe algorithm, two hybrid algorithms, and random selection. The two statistical algorithms used the median of stored data from past transfers, while the dynamic probe algorithm used tcping to probedall the servers and requested from the first to respond. The two hybrid algorithms used both statistical data and dynamic probes. The first used the median data to select the n machines with the fastest bandwidth and then sent probes to each of these servers. The second sent a probe to all the servers, and then waited a certain period of time for responses. It then picked the server with the fastest median bandwidth among those that replied within that length of time. Tests were performed on three clients in different geographical regions, accessing a pool of government web servers. Out of this pool of 42 web servers, ten were randomly chosen to be the group of replicated servers, and the six selection algorithms were iterated through. Number of bytes, algorithm overhead, connection time, read latency, and remaining read time were all measured and noted. They discovered that overall, the probing algorithms were better at reducing response times than the statistical algorithms, or latency, and that the simple probe was just as effective as the hybrid algorithms. Of the two statistical algorithms, latency was worse, with only Random having poorer performance. They also determined that overhead from the various selection algorithms was not very significant, and that connect, latency, and read times tend to follow the same performance order as total time. The overall distribution of times was found to have large skews, and long, low, flat tails that were attributed to dropped packets and the resulting TCP/IP retransmissions and window adjustments. They also studied time of day effects, and discovered that Internet traffic and server loads increase in the mornings, decrease in the late afternoons, and are much lower on weekends than during the week. The performance of the selection algorithms retained their relative order as load changed,

so one algorithm should be sufficient regardless of load. Overall, Dykes, et al's study [10] concluded that

1. dynamic probes were better than statistical estimators, especially in light of network changes
2. including nearby servers in the candidate set improve performance for all algorithms
3. performance should be evaluated by total elapsed time rather than latency or server load.
4. pathologically long delays can be reduced, but not eliminated by these algorithms
5. when there are a large number of servers, use bandwidth data to reduce the number of probes

Rodriguez, et al, studied the use of accessing multiple mirror sites in parallel as an alternative to server selection to load balance and speed up performance. Their general method was to open TCP connections to all the mirror sites and request pieces of the document from each site in an adaptive way. They studied two specific implementations of this method. One was to request sizes from each server proportional to its server rate, based on a database of periodically refreshed past server rates. The objective was to make requests such that all accesses completed at the same time, and the link was fully utilized. This method was found to work well when the network was stable, but wasn't as useful when network and server conditions changed rapidly. The other was a more dynamic method; the file was split into many small blocks, and one block was requested from each server. As one returned, then another block would be requested from that server. When all blocks were received, the file was reassembled and reconstructed. Performance could be improved by pipelining the blocks to the servers. This method was more adaptive to changing conditions in the network. From experimentation with a client in France and servers geographically dispersed in a number of different countries, they determined that these methods caused very high speedups when all the servers have similar performance, but less significant speedup if one server is much faster than the others. However, response time was found to be at least as low as that of the fastest server in all cases. Therefore, although parallel access worked best with no common bottleneck link, its performance was still as good as that of the fastest connection and was therefore equivalent or better than explicit server selection.

This study focused on transferring large documents, on the order of several hundreds of KB, and suggested that small documents should be grouped together into a larger one for parallel access. If CFS continues to use a block-level store with blocks on the order of 10s of KB, parallel access of the same block may not be nearly as useful as described in this study [19].

Byers, et al, proposed a different method of accessing mirror sites in parallel, utilizing erasure codes rather than renegotiating transfers with the servers. Erasure codes are designed so that a k-packet file can be encoded to generate n packets, any k of which are sufficient to reconstruct the entire file. These codes had been utilized in the past to provide fault tolerance. Byers' method still provides fault tolerance, but its main focus is on improving download time, and hence utilizes Tornado codes, which have extremely fast decoding and encoding algorithms, at the expense of some encoding efficiency. In other words, more than k packets may be needed to decode the file. Also, when downloading packets off of multiple servers, the recipient may receive duplicate packets. Performance was measured by the reception inefficiency, which is the combination of the previously mentioned decoding inefficiency and the distinctness inefficiency due to duplicate packets. This method was tested in simulation, and reception inefficiency and speedup were examined as the number of servers and the stretch factor (degree of encoding) were varied. They found that with moderate stretch factors, adding additional sources dramatically speeds up downloads. When one sender was significantly faster than another, the speedup was found to be equivalent to that of the fastest server. This is essentially the same result noticed by Rodriguez. Overall, the conclusion was that parallel access through tornado codes did provide improvements in performance over single downloads [3].

5.2 Peer-to-peer Systems

Several similar systems to CFS have been designed over the past few years, and many have had to tackle the question of server selection within their respective architectures. Below are a few examples of such systems and the mechanisms they use.

OceanStore is a utility infrastructure designed to provide persistent storage of information on a global scale. It makes use of untrusted machines around the Internet, and utilizes promiscuous caching, allowing data to be cached anywhere, anytime. Location is thus not

tied to information. Objects in the system are replicated and stored on multiple servers, but the replicas are not tied to those servers, acting as floating replicas. OceanStore utilizes two kinds of storage, active and archival. The active form of the data is the latest version and is located by use of a two-tier architecture. The first tier is a fast, probabilistic local algorithm that uses Bloom filters to route queries to likely neighbors at the smallest distance away. The metric of distance used is hop-count. The second tier is a global routing algorithm that utilizes a hierarchy of trees through which data is replicated. Queries are propagated up the tree until a copy is found. Due to the way the trees are formed, this is likely to be a copy on the closest replica to the requester. OceanStore also has deep archival storage, which utilizes erasure codes as described above to ensure the survival of data. At the same time, OceanStore protects itself against slow servers by requesting more fragments than needed and reconstructing the data as soon as enough fragments are available. Since search requests are propagated up the location tree, closer fragments tend to be discovered first. OceanStore provides for much stronger requirements and guarantees than CFS does, and has a correspondingly more complex structure. CFS' simpler routing and data location architecture lends itself to other forms of location and selection [14].

OceanStore is built around Tapestry, an overlay infrastructure for wide-area location and routing. Tapestry's node organization and routing inherently use distance information in constructing the topology. It also provides some flexibility in its routing architecture, storing the location of all nearby replicas at each node on the route to the root node. The root node is primarily responsible for knowing the location of a particular object and is deterministically assigned to that object. Systems using Tapestry can choose their own application-specific metrics in addition to the inherent distance metrics to select replicas. It is therefore similar to CFS/Chord in that it provides a mechanism to locate and manage replicas, but does not actually implement selection in its infrastructure. Unlike CFS, however, Tapestry takes into account distance and locality in its topology construction [27].

Pastry is a very similar location and routing system to Chord, using similar principles of node IDs and object IDs. Additionally to maintaining routing information about nodes that represent different parts of the ID space, it also maintains a "neighborhood set" at each node that keeps track of a given number of nodes that are closest to that node by some proximity metric. Pastry does not specify the metric, instead assuming that the overlying application will provide a function for determining the distance of a nearby node by its IP

address. At each step of routing, the query is forwarded to the node in the ID space most similar to that of the desired object that is also the closest by the proximity metric. In this way it keeps the overall routing fairly short and essentially guarantees routing to the closest replica [20].

Chapter 6

Conclusion

Replication is a common technique for improving availability and performance in a distributed system. Replicas dispersed throughout the network are failure-independent and increase the likelihood that any given client will be reasonably close to a source of data. In order to properly take advantage of a replicated system, server selection should be used to locate a replica with reasonably good performance for a particular client. "Good performance" can mean any number of things, such as good load-balancing, or good end-to-end performance. CFS is a decentralized, distributed cooperative system that inherently replicates each piece of data and spreads it to machines that are unlikely to be near each other in the network. It also provides load balancing and allows replicas to be located without any additional mechanisms beyond those needed to maintain the system. CFS therefore lends itself well to the use of server selection to improve performance, in this case end-to-end performance for the client.

6.1 Conclusion

Server selection can be used at two different levels of CFS: Chord, the lookup layer, and dhash, the data retrieval layer. The requirements of the two layers are fairly independent. The main concern for the lookup layer is to reduce the overall latency of a lookup path, made up of a series of indirect queries. At the dhash layer, the main concern is to decrease data retrieval time for a direct connection from the client to the chosen server. If the performance of both these layers was improved, the overall end-to-end performance experienced by a client requesting a block would correspondingly improve. Even if performance

is not absolutely optimal, server selection will help to reduce the variation in performance experienced by different clients.

Based on experimental evidence within a testbed of 12-15 Internet hosts, selection methods were found to improve the performance at each stage. In the lookup step, verification of the triangle inequality, combined with the relatively stable nature of ping times over time, indicate that the use of past performance data to select from a group of possible lookup servers can reduce the overall latency of the lookup. In the dhash layer, comparative performance of a number of different selection methods indicate that pinging the machines first and then downloading from the fastest provides reasonably good performance that is a significant improvement over random selection. However, it may be possible to use past data in a way very similar to that of the lookup step to select a machine instead, which has much lower overhead. Regardless of the method used, these experimental results prove the benefit of using server selection in CFS.

6.2 Future Work

The experiments described in this thesis provide a good basis for future work in the area of server selection in CFS. More work needs to be done to determine the actual value or improvement each technique offers CFS, rather than how it compares to the others. A comparative evaluation of the use of past latency data from an intermediate machine versus directly pinging all the replicas and selecting one should be made. Further investigation also needs to be made into the actual overhead and costs of each approach, in terms of latency, bandwidth usage, and server load, to determine if the method is actually worth using. More experiments should be conducted on an expanded testbed, or within simulation, so that the conclusions are not biased by the particular characteristics of the machines within this testbed. Recently, CFS has moved away from using TCP as its transfer protocol. Therefore, these experiments should be repeated using the new protocol to determine how well these methods perform using those protocols. Without the overhead of TCP connection set-up, the methods may yield different results. Similarly, a different latency probe should be used besides ICMP ping, to determine the relative overhead of using ICMP. Finally, the final candidate methods should be actually implemented within CFS and tested on a real CFS system to see the effects on overall file retrieval.

Bibliography

[1] Paul Albitz and Cricket Liu. *DNS and BIND.* O'Reilly and Associates, Inc., 1997.

[2] David Andersen, Hari Balakrishnan, Frans Kaashoek, and Robert M orris. Resilient overlay networks. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles*, October 2001.

[3] J. Byers, M. Luby, and M. Mitzenmacher. Accessing multiple mirror sites in parallel: Using tornado codes to speed up downloads. In *INFOCOM 99*, April 1999.

[4] R. Carter and M. Crovella. Server selection using dynamic path characterization in wide-area networks. In *IEEE Infocom '97*, 1997.

[5] Robert L. Carter and Mark E. Crovella. *Dynamic Server Selection using Bandwidth Probing in Wide-Area Networks.* Boston University Computer Science Department, March 1996.

[6] Mark E. Crovella and Robert L. Carter. Dynamic server selection in the internet. In *Third IEEE Workshop on the Architecture and Implementation of High Performance Communication Subsystems*, http://www.cs.bu.edu/faculty/crovella/paper-archive/hpcs95/paper.html, August 1995.

[7] Frank Dabek, Emma Brunskill, M. Frans Kaashoek, David Karger, Robert Morris, Ion Stoica, and Hari Balakrishnan. Building peer-to-peer systems with chord, a distributed lookup service. In *Proceedings of the 8th Workshop on Hot Topics in Operating Systems (HotOS-VIII)*, Schloss Elmau, Germany, May 2001. IEEE Computer Society.

[8] Frank Dabek, M. Frans Kaashoek, David Karger, Robert Morris, and Ion Stoica. Wide-area cooperative storage with CFS. In *Proceedings of the 18th ACM Symposium on Operating System s Principles (SOSP '01)*, To appear; Banff, Canada, October 2001.

[9] Kevin Delgadillo. *Cisco Distributed Director.* Cisco, http://www.cisco.com/warp/public/cc/pd/cxsr/dd/tech/dd_wp.htm, 1999.

[10] Sandra G. Dykes, Kay A. Robbins, and Clinton L. Jeffery. An empirical evaluation of client-side server selection algorithms. In *INFOCOM (3)*, pages 1361–1370, 2000.

[11] Gnutella website. http://gnutella.wego.com.

[12] J. Heidemann and V. Visweswaraiah. Automatic selection of nearby web servers. Technical Report 98688, USC/Information Science Institute, 1998.

[13] Karger and et al. Web caching with consistent hashing. In *8th International WWW Conference*, http://www8.org/w8papers/2awebserver/caching/paper2.html, May 1999.

[14] John Kubiatowicz, David Bindel, Yan Chen, Patrick Eaton, Dennis Geels, Ramakrishna Gummadi, Sean Rhea, Hakim Weatherspoon, Westly Weimer, Christopher Wells, and Ben Zhao. Oceanstore: An architecture for global-scale persistent storage. In *Proceedings of ACM ASPLOS*. ACM, November 2000.

[15] Steve Lewontin and Elizabeth Martin. Client side load balancing for the web. In *Sixth International World Wide Web Conference*, http://decweb.ethz.ch/WWW6/Posters/707/LoadBal.HTM, April 1997.

[16] Napster. http://www.napster.com.

[17] Katia Obraczka and Fabio Silva. Network latency metrics for server proximity, 2000.

[18] Ogino and et al. Study of an efficient server selection method for widely distributed web server networks. In *10th Annual Internet Society Conference*, http://www.isoc.org/inet2000/cdproceedings/1g/1g_1.htm, July 2000.

[19] Pablo Rodriguez, Andreas Kirpal, and Ernst Biersack. Parallel-access for mirror sites in the internet. In *INFOCOM (2)*, pages 864–873, 2000.

[20] Antony Rowstron and Peter Druschel. Pastry: Scalable, distributed object location and routing for large-s cale peer-to-peer systems. In *Proceedings of the 18th IFIP/ACM International Conference on Dist ributed Systems Platforms (Middleware 2001)*, November 2001.

[21] M. Sayal, Y. Brietbart, P. Scheuermann, and R. Vingralek. Selection algorithms for replicated web servers. In *Workshop on Internet Server Performance '98*, http://citeseer.nj.nec.com/sayal98selection.html, June 1998.

[22] SEC, http://www.sec.gov/Archives/edgar/data/1086222/0000950135-99-004176.txt. *Akamai Technologies S-1 Filing*, August 1999.

[23] Mark Stemm, Srinivasan Seshan, and Randy H. Katz. A network measurement architecture for adaptive applications. In *INFOCOM (1)*, pages 285–294, 2000.

[24] Ion Stoica, Robert Morris, David Karger, M. Frans Kaashoek, and Hari Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In *Proceedings of the ACM SIGCOMM '01 Conference*, San Diego, California, August 2001.

[25] Ion Stoica, Robert Morris, David Karger, M. Frans Kaa shoek, and Hari Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet app lications. Technical Report TR-819, MIT, Cambridge, MA, March 2001.

[26] Ellen W. Zegura, Mostafa H. Ammar, Zongming Fei, and Samrat Bhattacharjee. Application-layer anycasting: A server selection architecture and use in a replicated web service.

[27] Ben Y. Zhao, John Kubiatowicz, and Anthony D. Joseph. Tapestry: An infrastructure for fault-tolerant wide-area location and routing. Technical Report UCB/CSD-01-1141, April 2001.