Semester Thesis on Chord/CFS:

Towards Compatibility with Firewalls and a Keyword Search

David Baer

Student of Computer Science
Dept. of Computer Science
Swiss Federal Institute of Technology (ETH)
ETH-Zentrum, CH-8092 Zurich, Switzerland
email: david@student.ethz.ch
Summer 2002

Prof. Dr. Roger Wattenhofer
Distributed Computing Group
Advised by Keno Albrecht

Contents

1 Introduction

This semester thesis discusses two selected problems of a concrete implementation of a completely distributed file system.

Distributed file systems (peer-to-peer file systems) is a rapidly expanding research topic. Accordingly, there is a vast amount of proposals on this subject. These works can be roughly divided into two main topics. A lot of research is done for peer-to-peer file sharing systems, in which a (possibly) huge number of (unknown) users share files. Examples for such file sharing systems are Napster [4], Gnutella [5] and P-Grid [6]. A slightly different problem are distribute file systems similar to a UNIX-file system. The two approaches mainly differ in the question where the files are stored. While for file sharing systems it would be unacceptable to copy files onto machines different from the one that offers it, this is exactly what happens in file systems. Examples for such file systems are CFS/Chord (that is treated in this thesis), Viceroy [7] or Small-World [8]. An additional distributed file system is Freenet [9] with focus on a different problem (reducing the risk of publishing/reading of critical material in a censored or oppressed society).

General problems for all distribute file systems include scalability in terms of the number of participants as well as different file sizes. Another problem that arises with all distributed file systems is a trade off of including centralized units versus completely distributed which adds to the individual complexity of each participant.

The focus of this paper is on CFS with the underlying component Chord. It is a fully distribute file system: a system where files can be saved and read again afterwards, which is distributed onto several different servers and which avoids any central components (unlike AFS in UNIX).

The two topics that I am treating here are:

- a possibility to include machines behind a firewall into Chord.
- an algorithm to enable a keyword search in CFS.

2 Chord/CFS: An Overview

2.1 Chord

Chord [1] is a distributed look up protocol developed at the MIT Laboratory for Computer Science. The Chord protocol constructs a logical ring within all N participating servers. It supports just one operation: given a key, it maps the key onto a node. Chord uses a cryptographic hash function to distribute the data items evenly over that ring. Each node on the ring keeps the addresses of several ($O(\log N)$) other nodes in the ring in a **finger table**. Look up is done with $O(\log N)$ messages, while join- and leave operations take no more than $O(\log^2 N)$ with high probability. For stability each Chord server keeps a table with the following r (r was chosen to be $2 \log_2 N$) servers (**successors**) . Every node in Chord knows its predecessor. For a more detailed introduction to Chord, please refer to [1].

2.2 CFS

The Cooperative File System (CFS) [2] is a peer-to-peer read-only file system that is built on top of Chord. It hashes the blocks and the servers (cryptographically) onto a value of specified length. The files are stored at the (according to Chord) corresponding server on the ring. All servers on the ring are therefore responsible for storing and providing files that are hashed onto a specific interval on the ring. The fact that the cryptographic hash function *SHA-1* is used guarantees with high probability that all files and servers are uniformly distributed over the given interval (which is 0 to 2^{160}-1).

Large files are divided into smaller blocks. Each block is hashed and stored independently according to the calculated hash value, which serves as identifier. A file is held together by an inode block which contains all identifiers of a file. An inode block is then hashed again and registered in the same manner in a directory block whose hash value is entered into a root-block. Root-blocks are identified by a public key which serves in the same time as an authentication for the whole root-block. The root-block has been signed before with the corresponding secret key.

CFS is divided into three layers. The Chord layer (which is at the *bottom*) maintains routing tables and is used to find blocks. On top of it there is the **DHash** (distributed hash) layer which is the core of CFS and whose task it is to store data blocks reliably. On top there is the FS (file system) layer which provides a file system interface to clients.

To increase availability CFS **replicates** files on the following k servers (in the ID ring) where $k \leq r$.

To avoid overload of servers that hold popular files DHash uses a **caching** mechanism that replicates files on their *look up path*, that is all the servers that have been contacted to find the server holding the file. Therefore popular files will be cached more often and can be found more quickly.

To take into account that different servers may have different bandwidths and storage amount to share, CFS introduces the notion (from [3]) of **virtual servers**: One real server may act as multiple virtual servers and therefore be *present* at several different positions in the ring.

2.3 Advantages

Chord/CFS seems to fulfill the requirements of a distributed peer-to-peer file system very well. It is completely distributed and does not need any central node. With look up operations being in $O(\log N)$ in space and messages it is also scalable in terms of number of participants. Concepts such as virtual servers and the fragmentation into blocks of large files handle in an uncomplicated manner different *strength* of servers or different file sizes. Replication guarantees (according to the authors of [2]) high availability. All these results are achieved by surprisingly simple means. The use of well known technologies such as SHA-1 keeps the complexity manageable and can - in the case of SHA-1 in combination with the public key/secret key naming - even guarantee some authenticity.

2.4 Problems

Of course Chord/CFS has not solved all problems yet. One remaining problem of Chord is that servers behind firewalls are not accessible by other servers in the ring or clients.

Another problem of CFS is a keyword search that would allow the use of keywords (maybe even with wild cards) to locate files rather than a file name. For these two problems I am going to provide some answers in the following two chapters.

Other problems include a possible usage of such a CFS. While the authors of [1] claim that it works stable in the case of “many” joins and leaves of servers, it is clear that in such a case a big problem will be the copying of files from one server to another as in such a case the responsibility of a server may change rapidly. As a lowest bound each server should remain in the ring at least as long as it takes to

copy all files it is responsible for onto its machine and possible files it is providing from its machine. A kind of file sharing such as Napster [4] or Gnutella [5] seems unrealistic because all servers in the ring would need to host other files. The most likely application would be some sort of a distributed Unix-file system. The fact that the files are replicated several times is only useful if the underlying network is very weak and/or a (very) high availability of the files must be guaranteed. Another problem for such a file system is that it is read only and updates (which are handled as replacements) can only be done by the original author.

3 A Proposal for the Firewall-Problem in Chord

3.1 The Problem

Servers behind firewalls in Chord may not be accessed by clients and other servers because administrators are very often unwilling (for security reasons) to open ports other than the HTTP-port for incoming connections. In addition, the identifier of a node is based on its IP address which is presumed to be unique [10].

3.2 Overview

The problem described above can be approached by using the already existing concept of virtual servers. Existing servers in the ring which have bandwidth disposable can serve as proxies for other servers behind firewalls. These proxy-servers internally add a virtual server for a firewall server, but just forward all messages they receive. On the other hand, a firewall server needs to open and maintain a standing connection to a proxy server once (see figure 1).

3.3 Details

3.3.1 Requirements for Proxy Servers

Each server has to decide whether or not it would like to offer some of its bandwidth capacity to other servers behind a firewall. Therefore such a server should have sufficient bandwidth-capacity and it should also remain in the ring for some time to preserve stability.

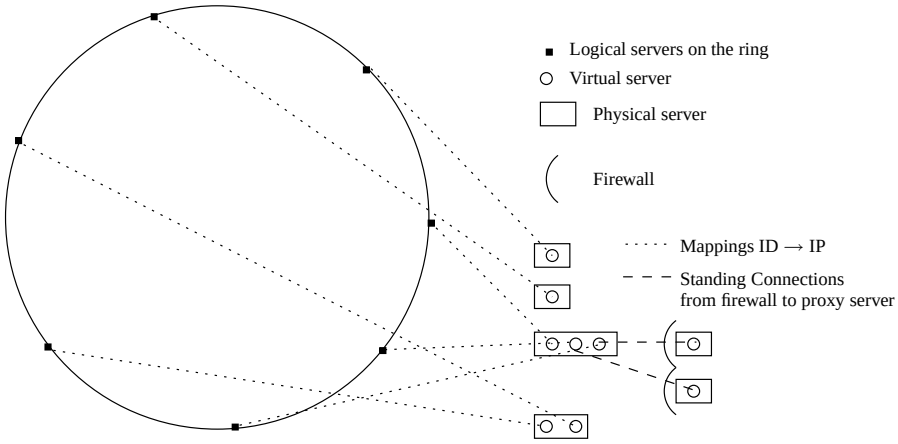

Figure 1: Firewall Servers

3.3.2 Finding a Proxy Server

When joining the ring every server creates some additional entries to the successor and finger table. These are an *id*, the *IP-address* and the *number of hops* of a server that can be reached over that link and that declared itself to be willing to act as a proxy server. This information is passed in a recursive way with always the closest link known to a server to its "logical neighbors". This list is updated at certain intervals. To be effective it is assumed that there will not be too many changes in such information (see 3.3.1).

A server behind a firewall but willing to join the ring and act as a server, will have to ask (as a client) its entry point into the ring for the list of proxy servers. It can then decide by "pinging" them and therefore optimizing the crucial latency, which server it would like to use. Then it will establish a standing connection to that proxy server.

3.3.3 Communication after Establishment of Connection

As soon as a connection between a firewall server and a proxy server has been established, a firewall server will get an id (built from the IP and the internal virtual server number of the proxy server as with regular virtual servers) and start acting as a server in the ring. The proxy server will simply tunnel all requests to and from the firewall server. While look up of files will always have to go through the proxy server, there are several possibilities of what can be done with replies:

- Straight forward all replies are tunneled through the proxy server. This is simple, requires the least changes to the existing protocol but would consume a lot of bandwidth of the proxy server, reducing its capability to act for other firewall servers as a proxy.
- A more sophisticated solution would send data items on a direct link between firewall server and client. This needs several changes to the existing protocol. The firewall server needs to know the IP-address of the client. In the reply message the firewall server will return a message with his IP, announcing that he will open a connection to the client directly. This solution relieves the proxy server from all the (large) data traffic downwards but can only work if the client is not itself behind a firewall.

3.4 Advantages

The proposed concept is relatively simple and uses many existing features of Chord. It solves the problem of servers behind firewalls that cannot be contacted from outside in reversing that act.

3.5 Disadvantages

This solution will of course increase latency of all communication with a firewall server and therefore reduce performance of the whole ring as there is one additional hop in each direction.

Other topics that are arisen by this concept are security issues. The proxy server controls all access to the firewall server and its contents. But as all data is replicated and signed, these facts do not lead to new problems.

4 A Proposal for the Keyword Search in CFS

4.1 The Problem

A keyword search could be addressed in the same manner as a file search: a keyword is hashed to a specific value and then a node responsible for that value keeps a table of the files that register that keyword. The problem is that very popular keywords will lead to load balancing problems [10]. The solution used with files does not work here as caching will lead to stale data (because keyword indices are very likely to change rapidly).

4.2 Possible "naive" solutions

Other solutions tend to have problems with scalability. This is true with Gnutella ([5], [11]), where a flooding algorithm is used as with Napster [4], where a centralized server takes care of a keyword directory.

Most obvious solutions are a trade-off of centralized nodes, huge tables, too many messages, only find part of the files containing a certain keyword, ...

4.3 Overview

```
//RPC handler on server n.
n.lookupStep(key)
    if key ∈ (n.predecessor ... n.endOfKwIntervall] then
        if key ∈ (myKwTable ∪ replicatedKw) then
            return [COMPLETE, key, key_nodes] // n returns the IDs of the files containing the keyword
        else //n should have the keyword, but doesn't ...
            return NONEXISTENT //... so it is non-existent
    else if (({key_kwRepZone} ∩ {x | x ∈ (stored ∪ cached ∪ replicated)}) ≠ ∅) then
        //Find the next direct link into the replication zone.
        return [CONTINUE, key, nextNode in {key_kwRepZone} ∩ {x | x ∈ (stored ∪ cached ∪
replicated)}) ≠ ∅ ]
    else if (⋃_{i=0}^{log2 ringSize} {x_i|(key−2^i modulo ringSize)_kwRepZone} ∩ {x| x ∈ (stored ∪ cached ∪
replicated)}) ≠ ∅) then
        //Find the next node, that has a direct link into the replication zone.
        return [CONTINUE, key, nextNode in ⋃_{i=0}^{log2 ringSize} {x_i|(key−2^i modulo
ringSize)_kwRepZone} ∩ {x| x ∈ (stored ∪ cached ∪ replicated)}) ≠ ∅]
    else
        //Find highest server < key in my finger table or successor list.
        next_hop = lookup_closest_pred(key)
        succ_list = {s|s∈ {fingers ∪ successors} > next_hop}
        return [CONTINUE, next_hop, succ_list]
```

Figure 2: The FIND-Keyword Algorithm.
endOfKwIntervall is the last ID, that is in *n*'s replicated keywords table (replicatedKw) if it exists.
$kw_{kwRepZone}$ is the zone on the ID ring where the keyword *kw* should be replicated.

The general idea is to use the same algorithm as for a file look up with some slight modifications. First, the keyword remains replicated in a certain well-defined area. Secondly, searching nodes will try to jump *somewhere* into that zone from a *node as far away as possible*.

4.4 Details

4.4.1 Registration of Keywords

A machine A that adds a file to CFS creates a list of keyword for particular file and adds it to file header(inode block). Server B hosting the file header then hashes each keyword onto a value and is responsible to register the keyword at the appropriate server C (in the same manner as for files) in the ring. That server adds the keyword to the keyword list (*myKwTable*) together with the server holding the corresponding file. It is responsible for replicating it to the next r ($= 2 \log_2 N$) servers (there being *replicatedKw*). The "official" **replication zone** is only $\log_2 N$, the remaining $\log_2 N$ replications are used to prevent overshooting. Every server C is also responsible for correct replication of that table.

4.4.2 Look up

The find algorithm of CFS needs most adaptations for keywords. Between the check, whether the node *n* is responsible for that keyword and the reply with the next server to continue the search several other checks have to be done (see figure 2):

- Find out whether *n* has a link directly into the zone where the keyword may be. This is done with the estimated total number of Chord nodes (*N*)in the system, based on the density of nodes nearby on the ID ring as it is already done in the *server selection* of [2]. If so, *n* returns that node.
- The node *n* calculates all IDs *i* which should have a direct link to the keyword or the replication zone of that keyword. It then checks with its tables to find and return the next node from his position on the ID ring in that set of *i*s.
- Only if these checks fail, *n* will return a node to continue the search as in the regular find algorithm.

The idea is to distribute the target where a certain keyword may be found over a certain zone (the $key_{kwRepZone}$) and to use big jumps to land there to relieve the servers in front of the target zone from look up procedures.

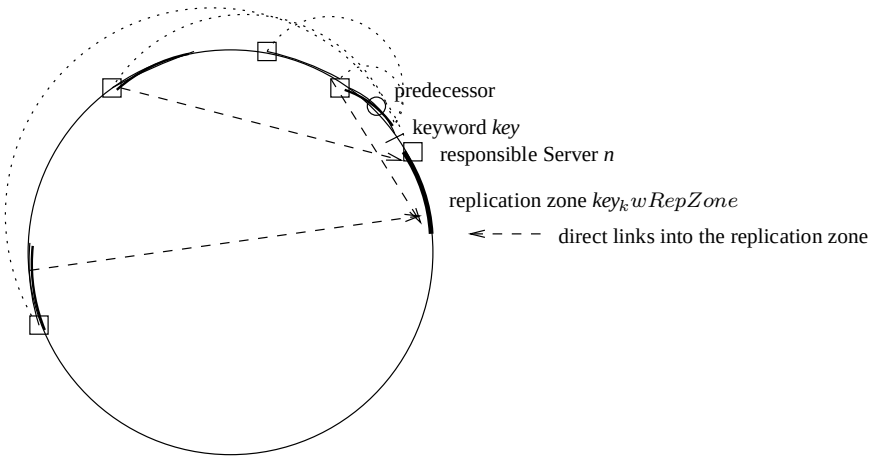

Figure 3: The replication Zone and the links into it.

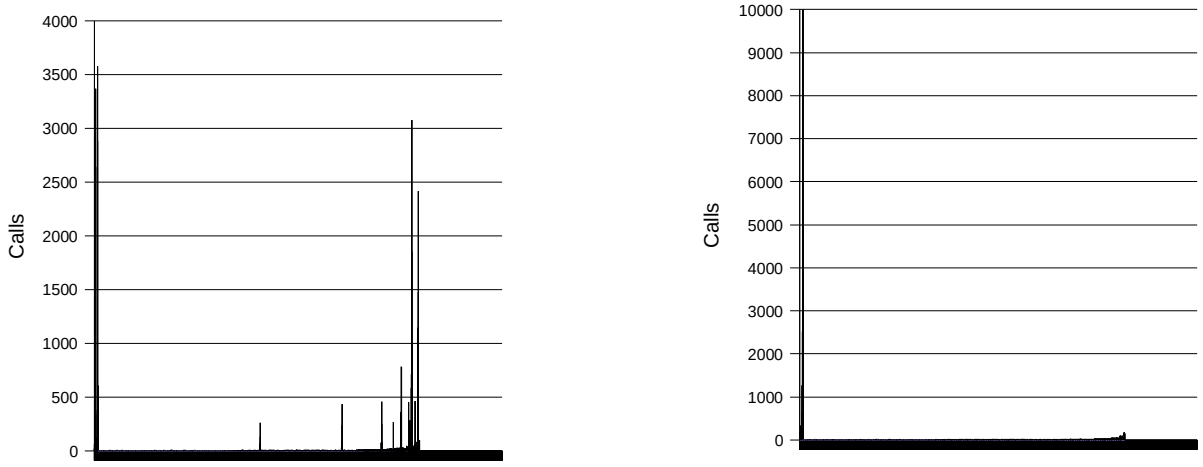

Figure 4: The results of a simulation as described in section 4.5. The x-axis displays the ID-ring. The keyword that was searched was *122919633196828977462 27*. On the left there is the newly adapted algorithm, while the diagram on the right hand side shows the regular look up algorithm. Note the different scales on the y-axis.

4.5 Simulation

A simulation has been written in Java that focus only on the look up algorithm for keywords. The results show that the heavy bursts that occur with the original CFS file look up algorithms are distributed much better and several smaller bursts appear. The simulation that is shown in figure 4 and in figure 5 are of an extreme case, where only one keyword has been searched 10000 times from 5000 servers randomly in a ring with an interval of [0..2^{80}-1] and with 5500 different keywords.

Simulations with different random keywords always resulted in a slightly better distribution of peaks and in about 10% to 20% less messages than had been needed before.

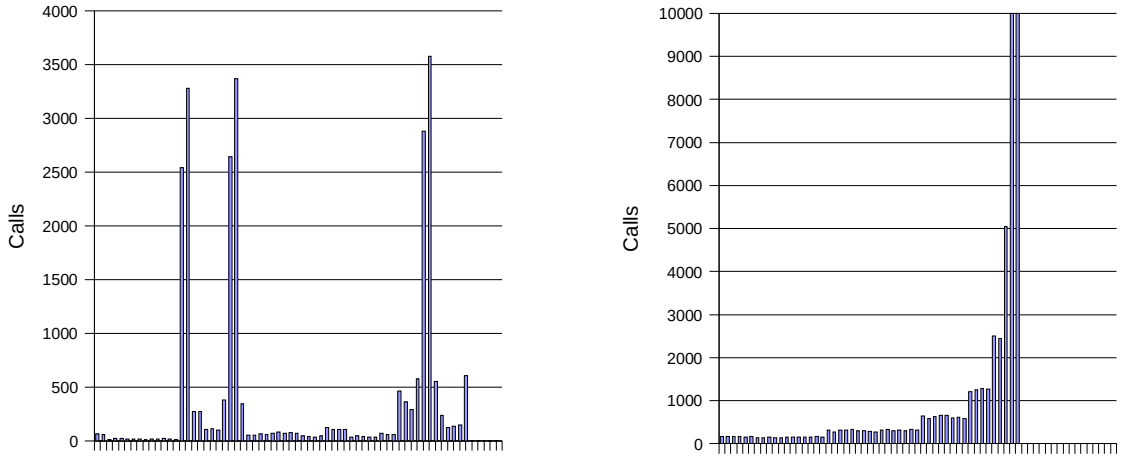

Figure 5: The same situation as in figure 4 with only the first (according to the ID) 67 servers (the keyword appears first on the 62^{nd}). The x-axis displays the ID-ring. On the left there is the adapted algorithm, while the diagram on the right hand side shows the regular look up algorithm. Watch the different scales on the y-axis!

4.6 Advantages

The modification proved to be successful under the tested conditions. It picks up many existing concepts and needs only slight modifications. It is an effective trade-off between the size of keyword tables, the number of messages needed to

find a keyword and the complexity of the algorithm and finds all existing keywords with guaranty.

4.7 Disadvantages

So far only the look up of keywords has been considered and no thoughts have been made for inserting keywords. This operation can also lead to load balancing problems and should therefore not be ignored. See 4.8 for an idea for that problem.

The solution presented here is very limited in scalability as been shown in the figures 4 and 5: The improvements are only of a factor two to three, and even decrease with the number of servers in the ring. It depends on the ratio of $\frac{n}{log_2 n}$, which is responsible for the decreasing success in quickly finding large jumps into the target zone with increasing *n*.

This solution for a keyword search does not include search with wild cards. For searching with wild cards, the concept of hashing would have to be dismissed. Neither does it provide an integrated search for combinations of keywords. This could be solved by searching individually every keyword and then handle the combinations.

4.8 Further Improvements

A possible way to solve the problem of inserting keywords would be to address just one node in the target zone. This node will then only notify the node responsible for that keyword, if that keyword is below some maximum number of servers providing files with that keyword. This results in a keyword search, where there is no guaranty, that all files will be found. This may not be very dramatic if search is done with only one keyword, but would certainly not be very welcome in the case of a combinatorial search.

References

[1] Ion Stoica, Robert Morris, David Karger, M. Frans Kaashoek, and Hari Balakrishnan, *Chord: A Scalable Peer-to-peer Lookup Service for Internet Applications*, ACM SIGCOMM 2001, San Diego, CA, August 2001, pp. 149-160.

[2] Frank Dabek, M. Frans Kaashoek, David Karger, Robert Morris, and Ion Stoica, *Wide-area cooperative storage with CFS*, ACM SOSP 2001, Banff, October 2001.

[3] Karge, D., Lehman, E., Leighton, T., Levine, M., Lewin,D., and Panigrahy, R. Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the world wide web. In Proceedings of the 29th Annual ACM Symposium on Theory of Computing (May 1997), pp. 654 663.

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

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

[6] P-Grid. http://www.p-grid.org

[7] Viceroy project homepage. http://www.cs.huji.ac.il/ anatt/viceroy/viceroy.html

[8] Small-World paper. http://www.cs.cornell.edu/home/kleinber/swn.pdf

[9] Freenet project. http://freenetproject.org/cgi-bin/twiki/view/Main/WebHome

[10] Chord faq. http://www.pdos.lcs.mit.edu/chord/faq.html

[11] Ritter J., Why Gnutella Can't Scale. No, Really.
http://www.darkridge.com/ jpr5/doc/gnutella.html