HEURIKA

A DECENTRALISED SHARED FILE SYSTEM FOR LOCAL AREA NETWORKS

December 2003

AALBORG UNIVERSITY
Group E1-202

Aalborg University
Department of Computer Science, Frederik Bajers Vej 7E, DK 9220 Aalborg Øst

Title:

Heurika – a decentralised shared file system for local area networks

Project period:
DAT3, Sep. 12th – Dec. 18th, 2003

Project group:
E1-202

Members of the group:
Anders Rune Jensen
Jasper Kjersgaard Juhl
Lau Bech Lauritzen
Michael Gade Nielsen
Ole Laursen

Supervisor:
Marius Mikucionis

Number of copies: 7

Report – number of pages: 74

Appendix – number of pages: 5

Total amount of pages: 79

Abstract

This report describes the development of Heurika, a distributed shared file system for local area networks. With Heurika, every client in a network contributes to the shared file system in a peer-to-peer manner to avoid the bottleneck and single point of failure that plagues traditional centralised shared file systems.

A design for a decentralised peer-to-peer file system is presented and its characteristics theoretically analysed. Then a prototype of the design implemented in C++ is described and experiments in a simulated network with 96 peers presented.

The theoretical and the empirical results indicate that Heurika can provide better availability and scale better than a centralised shared file system.

Preface

This report documents the development of a distributed shared file system for local area networks. The system has been developed as part of the DAT3 semester at Department of Computer Science, Aalborg University.

Source code for the prototype of the system is available at `http://www.cs.auc.dk/~jasper/dat3/` and on the attached CD-ROM. The log files from our tests are only available on the CD-ROM.

We would like to thank Martin Qvist from the Department of Mathematical Sciences, Aalborg University for his kind help.

Aalborg, December 2003,

Lau Bech Lauritzen	*Jasper Kjersgaard Juhl*
Michael Gade Nielsen	*Anders Rune Jensen*
Ole Laursen	

Contents

Chapter 1

Introduction

Organisations usually deploy a wide range of different facilities with the single purpose of improving the work flow within the organisation. Most often a file storage facility is among these facilities, and the local area network (LAN) of the organisation plays an important role in providing access to it. This chapter presents various approaches for providing a file storage facility, and defines the scope of the project. It concludes with a presentation of useful ideas from related systems.

1.1 The centralised file system

A simple approach for providing a file storage facility is to run a central file server on a single machine as shown in Figure 1.1. This provides users with concurrent access to a common place for storing private and shared files. The file server may run the Network File System (NFS) [2] protocol from Sun Microsystems, the Server Message Block (SMB) protocol from Microsoft or other file systems with similar features.

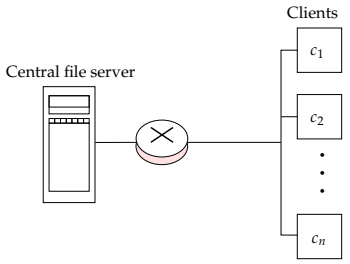

Figure 1.1: Clients accessing a central file server, running on a single machine.

This centralised approach ensures that clients can easily locate the file service, and that administrators can easily perform backup of important data. However, organisations whose work flow is highly dependent on the storage

facility are in a critical situation if the file server fails. We do not consider the problem of losing files since it, in most cases, can be alleviated by backup. But the failure of the file server may paralyse the organisation until the server is repaired or replaced. In terms of lost work hours, this may be very costly.

Another problem is that a single file server does not scale well if the number of requests grow significantly. Bottlenecks are likely to occur and the cost of adding more and more advanced components raises.

1.2 The replicated centralised file system

One way of achieving higher availability of the file sharing service is to have another server that can answer clients when the first server is down. This is known as *fail over*, and can transparently provide higher availability as long as the network is still running. For the extra servers, there are two possibilities: either they can be *passive*, meaning that they are unused as long as the primary server is running, or they can be *active*, meaning that they participate in serving client file requests with the first server.

1.2.1 Passive servers

The passive server approach is illustrated in Figure 1.2. An active server is the primary server that serves client requests, and the other secondary servers are passive as long as the primary server has not failed. Each passive server must have the following properties:

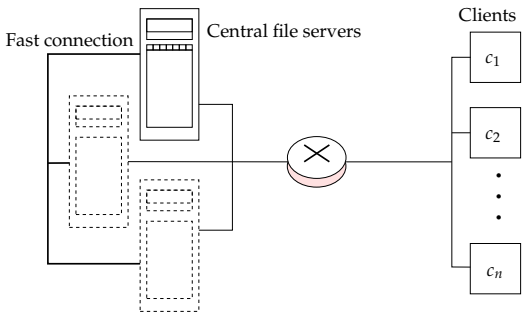

Figure 1.2: Clients accessing a central file server, running on a single machine. Several passive servers are ready to take over if the active server fails.

- It must hold exactly the same versions of the files in the active server, hence an entire replica of the files is needed. This can be achieved by using a fully replicated file system.
- It must be aware of the current states of the clients using the active server, e.g. current work directory and open files. This is easily achieved with a stateless file server protocols such as NFS [2].

The passive servers must be able to detect when the active server has failed. This can be done in two ways: either the passive servers can continually ask the active server whether it is operating, or the clients can themselves alert a passive server if the active server is not responding.

When a passive server has become active, the technical staff can disconnect the failed server and repair it without disturbing client activities. When the failed server is operating correctly again, it can be reconnected as a passive server and thereby be ready to take over if the new active server fails.

1.2.2 Active servers

Deployment of passive servers improves the availability of the file service, but having passive servers that hardly ever serves the clients seems wasteful. Instead all servers can be allowed to answer requests as illustrated in Figure 1.3. As with the passive approach the files must be fully replicated among the servers.

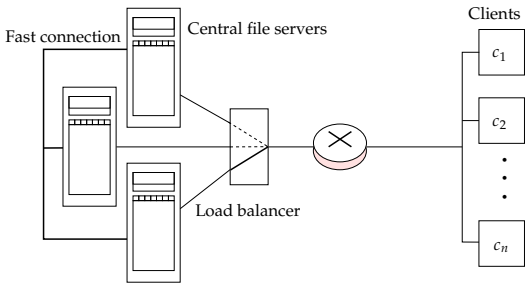

Figure 1.3: Clients accessing a central load balancer. Several active servers are ready to answer the clients.

A distribution mechanism of requests is needed to ensure that one server does not carry all of the work load – the load should be balanced among the servers. There are several approaches for load balancing [1] but the one shown in Figure 1.3 uses a special dedicated machine to select what servers to direct the incoming requests to. The most sophisticated load-balancing devices can detect failed and overloaded servers, and use this information to direct clients to one of the other servers. If a server fails, it can be disconnected, repaired and subsequently reconnected without disturbing clients as with the passive server approach.

1.2.3 Advantages and disadvantages

Fail over techniques solve the availability problem of the single centralised approach. If a load balancer is added, the scalability increases because the extra servers can serve clients in parallel. Compared to the simple approach, the per-

formance may however be worse since files need to be replicated when they are changed.

The primary disadvantage is cost. The extra file servers and the maintenance of them add to the cost of the file service and requires setting up a replicated file system among them. This added cost may even turn out to be too high compared with the cost of a disrupted work flow when a single server fails. And if additional storage is needed, all of the replicated servers will need to be upgraded, which causes additional expenses compared with the simple approach.

1.3 A peer-to-peer decentralised file system

An alternative approach, and the approach we will pursue in this project, for providing a file storage facility without a single point of failure, with better scalability and without resorting to a traditional expensive fail over solution, is to avoid any central servers and instead use the clients themselves for storing the files. Then the clients become peers and form a decentralised peer-to-peer network. To avoid losing files when peers crash or are shut down, the files must replicated throughout the network.

This approach could achieve:

- High availability because no single point is responsible for the file service.
- Much lower cost since there are no dedicated file servers to buy and maintain.
- Built-in scaling. When the number of peers and thus requests increase, so do the resources the file service can utilise.
- Faster serving of requests since there are many peers to use, e.g. when retrieving a file.
- Better utilisation of the computational resources in an organisation, e.g. better utilisation of free disk space, unused network bandwidth and unused processor time in each peer.

There are, however, also some disadvantages:

- The behaviour of the file service becomes much more unpredictable since it is spread out over the whole network.
- If the performance is not good enough, improving it is much more difficult than upgrading a few central servers.
- The burden the service puts on the peers may be too high. For instance, it may use too much network bandwidth or too much disk space. Due to

the replication and decentralised nature of a peer-to-peer system, there will be a certain amount of overhead in terms of duplicate data and extra coordination messages.

Hence, choosing between a distributed system approach and a traditional centralised approach will be a trade-off that depends on the particular circumstances in an organisation. In order to keep the discussions focused, we will make the following assumptions about the organisations that our system targets:

- The machines are equipped with hard drives with much more free space disk space compared to the amount of data that it should be stored in the file service (we will later quantify this, but e.g. 10 Gb free for storing 1 Gb of data per machine). Also, the rest of the hardware is powerful enough to handle the requests and responses. Hence, thin clients are not useful for our system.
- The internal network is a LAN with the following characteristics: high bandwidth (10/100 Mbit), low latency (round trip time on the order of 1 ms) and a limited number of clients (less than, say, 20,000).
- Malicious peers in the network can be tracked down (and the responsible person sacked), thus the system does not need to protect itself from denial-of-service attacks.

We believe these assumptions are valid for large number of organisations. New machines are sold with 100 or even 1000 Mbit Ethernet and several gigabytes of free disk space.

1.4 Aims of this project

The decentralisation of the file service introduces many problems concerning the peer-to-peer network that needs to be solved for it to be a viable solution. The aim of this project is to address these problems by *designing a decentralised peer-to-peer file system* with the following requirements (in order of priority):

High availability and fault-tolerance The failure of a peer must not bring down the system and must not cause files to be unavailable. Failure of multiple peers must not cause the system to end in an unresolvable inconsistent state.

Self-organising and dynamic The system must continue to function reliable when peers join and leave the network dynamically, and must itself organise the peers without human intervention.

Reasonable performance To be usable at all, the system must not be too slow or use too many resources (such as disk space, network bandwidth and CPU time).

Transparency The system should not look any different from the users' perspective than other networked file systems.

Scalability The system must be able to scale up to the number of machines connected to the LAN.

It is also an aim to *implement a working prototype* of the file system that demonstrates the features of the network. Finally, the last aim is to *analyse the design theoretically and in practise through the prototype*.

1.5 Related work

Several related projects already exist in the problem domain although most of the recent work seems to be intended for a wide-area network, the Internet. We will in this section present some of the ideas from these related systems that will be reused in our system.

For a decentralised distributed file system, the most basic problem that must be solved is that of locating the nodes to contact for a given file name. With traditional centralised distributed file systems such as NFS, the Andrew File System and Coda [2] the answer is simple: contact the central servers. For decentralised systems, there are various approaches:

1. One can maintain a separate dedicated directory service. To avoid a single point of failure, this service must itself be replicated. For scalability reasons, it may be necessary to decentralise the service to some degree; for instance, by organising it in a hierarchy such as the Internet DNS.

2. A completely decentralised, peer-to-peer approach is that of Gnutella [5] which floods the neighbourhood of nodes each time a lookup is performed. The underlying idea is that the active nodes are organised in a *logical* or *overlay* network where each node keeps routing information about a subset of the other nodes.

3. A recent advance is the application of hash-based lookup in overlay networks. Instead of using the routing information to flood the network, a hash function is used to determine where in the network the lookup key is located and the routing information is then used to guide the lookup to the right node.

   The usual hash designs, used for instance in general-purpose data structures such as hash tables, does not work in the face of nodes dynamically joining and leaving, and nodes having different incomplete views of

what constitutes the overlay network. Consequently, hash-based lookup rests on the concept of *consistent hashing* [6] where small changes do not require a global rehashing of the data.

In general, a dedicated directory service has the problem of much added complexity since it requires design, implementation and test of a separate distributed system.

The flooding approach is, at least conceptually, very simple but does not scale well [12] and cannot guarantee that data saved in one end of the overlay network is at all available at the other end unless the flood queries are allowed to travel an unlimited number of hops.

A number of systems have been devised for the Internet using the hash-based approach [9; 11; 15; 17; 22], collectively known as distributed hash tables. They differ in how the peers (known as nodes) are organised in the overlay network and how the routing tables are used and maintained. Common to all of them is that node identifiers and data keys are mapped into the same hash space. The data associated with a key is then placed on the nodes whose identifiers are closest to the key in this hash space given some distance metric. In general, r nodes are selected for some replication constant r which should be large enough to render data loss unlikely (with r replicas, $r - 1$ nodes may leave without causing data loss).

One system, Chord [17], deploys a ring topology where each node knows about the next r nodes in the ring to ensure that nodes can disappear without breaking the ring. Each node also maintains a table of the node halfway, three-quarter way, seven-eighth way and so on through the ring to be able to guarantee logarithmic time complexity for finding any node. The distance metric is the forward numeric distance through the ring with appropriate wrapping. The Cooperative File System (CFS) [3] is a distributed decentralised file system built on top of Chord.

The distance metric in another system, Kademlia [9], is based on XOR'ing the hash values and interpreting the result as an integer. The routing table in each node is split into a number of buckets of fixed size where each bucket in turn covers more distant and logarithmically larger areas of the hash space. Hence a given node knows many nodes in its neighbourhood, but only few nodes far away. For a file system on the Internet with potentially millions of nodes, this property is important to limit the size of the routing information in each node.

As illustrated in Figure 1.4, the nodes closest to a data key can be found by iteratively asking the known nodes for which nodes they think are closest to the data. A user on the originating node 001 has asked for a piece of data d with the key 110. The originating node has three buckets (denoted with dotted lines) and initially knows one node in each bucket, the nodes 000, 011 and 101. There is one more node, 111, in the network.

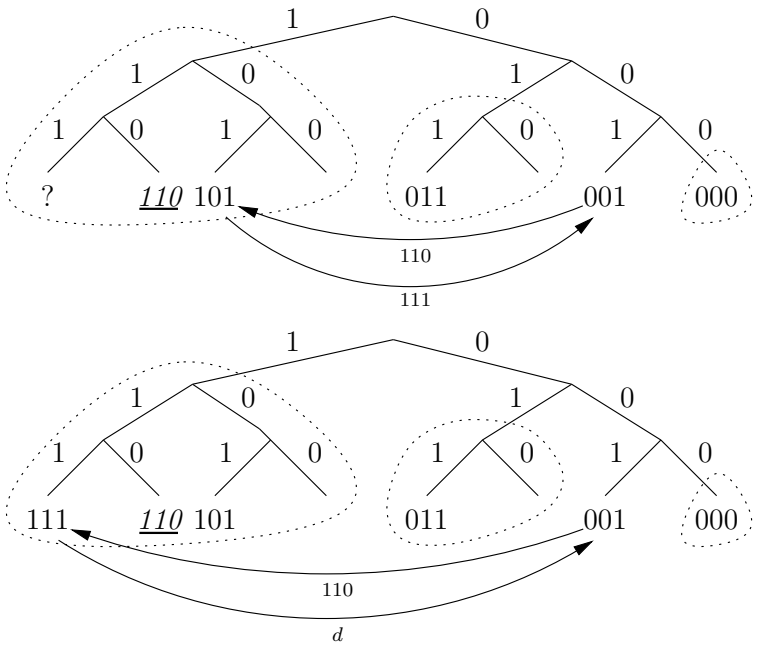

Figure 1.4: Example of Kademlia data retrieval with 3-bit keys and buckets of size 1 from the view point of the node 001.

At the first step, the node 101 is found to be the closest node to 110 since

$$101 \oplus 110 = 011 = 3$$
$$011 \oplus 110 = 101 = 5$$
$$000 \oplus 110 = 110 = 6$$

Node 101 is then asked for what it believes to be the closest nodes to 110. It only knows of the node 111 so it responds with 111. The originating node then asks node 111 for what it believes to be the closest nodes to 110, but since node 111 is storing d itself as there is no node at 110, it simply responds with d.

In Kademlia, the routing tables are mostly maintained by learning about other nodes from the messages that a node receives – in the above example, the originating node could have inserted node 111 in its bucket of most distant nodes if the bucket was not already full. To handle new nodes joining and nodes being inactive, extra random lookups are simulated from time to time.

Both Chord and Kademlia are attractive because they are (at least conceptually) quite simple. The main advantages of using XOR instead of a ring are that it is simpler to guarantee the logarithmic lookup, that XOR make it more feasible to use all messages for ensuring the connectivity of the overlay network (this will be explained in Section 2.2), and that there is more freedom in maintaining the routing information since Kademlia can choose to remember any node in a bucket instead of having to find the node halfway, three-quarter

way, etc. through a ring.

Thus the design of our system is based on the architecture of Kademlia. Other distributed hash table systems that will not be further considered in this project include Content-Addressable Network (CAN) [11] which uses a d-dimensional coordinate space, Pastry [15] which uses a technique known as prefix routing, and Tapestry [22] which uses a topology called Plaxton trees, and is used in the Oceanstore project [7] for a global-scale file system. A more detailed summary for Chord is given in [17].

Chapter 2

Design

This chapter presents the design of the system. First an overview of the component architecture is given, followed by a more detailed description of each component to define the underlying protocol, algorithms and data organisation in the system.

2.1 System overview

The system is based on a distributed hash table and consists of three components, the distributed hash table, the actual file system and the garbage collector, see Figure 2.1.

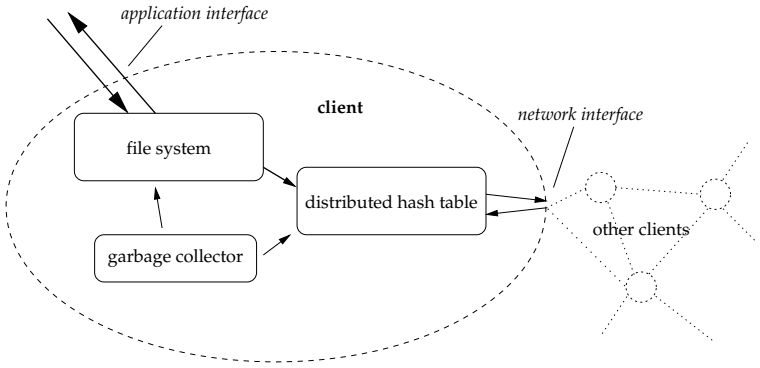

Figure 2.1: The architecture of a client of the system.

The distributed hash table component, described in Section 2.2, is concerned with the distributed storage and retrieval of data blocks which are at most B bytes large. To achieve high availability and ensure that blocks can be retrieved when nodes crash, it must replicate the data on different nodes.

The file system component, described in Section 2.3, is concerned with using the distributed hash table component to store files and directories by split-

ting them into blocks, and presents an ordinary file system interface to applications, albeit with looser guarantees than a local file system or a centralised file system such as NFS (Section 3.2 will examine this).

The garbage collector component, described in Section 2.4, is necessary because the two other components may leave blocks that are not needed or usable, i.e. if we visit all blocks in all files in all directories, some blocks will not be visited.

2.2 Distributed hash table

The interface to the distributed hash table component is *put(key, block)* that stores a block and *get(key)* that returns a block. Underneath this interface, each node participating in the overlay network keeps track of the addresses of a subset of other nodes in the network and maintains a list of the data blocks it is storing itself. This makes it possible to find and distribute the blocks.

The design of the distributed hash table is almost completely reused from the design of Kademlia [9] and uses the XOR metric to measure distance between keys.

Each node and each different data block is identified by a unique m-bit key (for some fixed value of m). To find out where to place a given block, its key k_b is XOR'ed with the key of a node k_n and the result $d = k_b \oplus k_n$, an m-bit integer, is interpreted as a distance. The block is then distributed to the r nodes in the network for which the distance d is shortest. Similarly, a block is retrieved by contacting anyone of the r nodes that the block is closest to.

Since we do not require each node to know about all the other nodes, the storage and retrieval process must be preceded by a lookup procedure that finds the closest nodes. Figure 2.2 illustrates the problem. As explained in Section 1.5 and in Figure 1.4, this can be done by iteratively asking the known nodes for what nodes they think are closest to the block key.

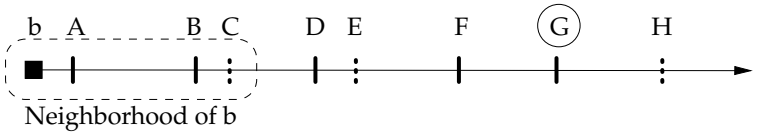

Figure 2.2: Nodes ordered on XOR distances to the block b. The nodes within the stippled box are responsible for storing b. The dotted nodes are the nodes that G does not know about. To store b, G must know about C.

2.2.1 Obtaining the keys

To obtain the keys needed to communicate with the component, a hash function H is required to produce an m-bit number. Examples of such hash func-

tions are SHA-1 [16], RIPEMD-160 [4] and MD5 [14]. To balance the work load among the nodes, it is important that the hash function distributes the values evenly. The pseudo-randomness property of the hash functions above assures this.

2.2.2 XOR and the topology of the space

Using XOR as the distance metric gives a topology that perhaps best can be explained by a tree, see Figure 2.3. The leaves correspond to nodes and the paths from the root to the leaves enumerate the node hash values. Then given a node n and a subtree that does not contain n, the distance from any node in that subtree to n is on the same order of magnitude (in base 2) [9].

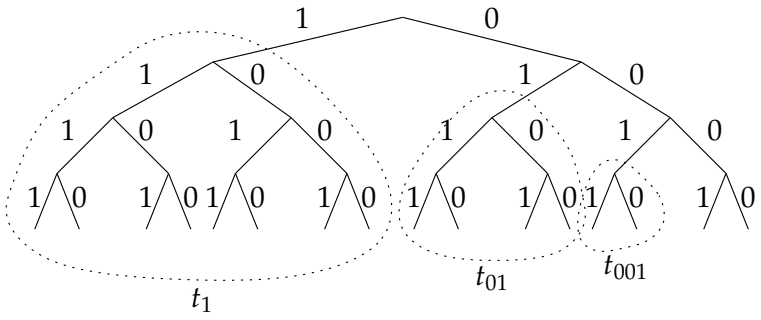

Figure 2.3: Illustration of the topology of the hash space. The XOR distance from the nodes 0000 and 0001 to a node in the subtree t_{001} is 0010 or 0011 (2 or 3). To a node in the subtree t_{01}, the distance is 0100, 0101, 0110 or 0111 (4, 5, 6, 7). To the subtree t_1, the distances are 1000, 1001, ..., 1111 (8, 9, ..., 15). Hence, for any nodes $a \in t_{001}, b \in t_{01}, c \in t_1$ we have the ordering $d(a) < d(b) < d(c)$ on the distances from 0000 and 0001 to a, b and c.

Apart from the rule that nodes in subtrees are on the same order of magnitude away, the tree does not give any visual clue precisely how far away a given node is, though. As illustrated in Figure 2.4, when the point of view is switched from one node to another, the distances inside a subtree that does not contain the two nodes are exchanged too!

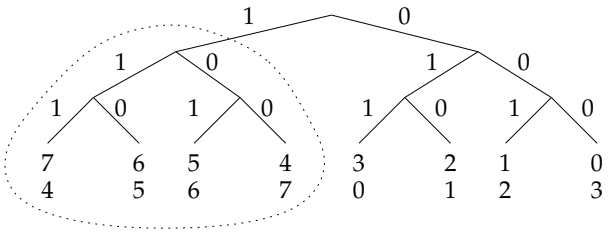

Figure 2.4: The distances from the node 000 and the node 011, respectively, to all the other nodes. Note that of the nodes in the left subtree (marked with a dotted line), 000 is closest to 100, whereas 011 is closest to 111.

In spite of this, XOR is in fact a valid metric since $x \oplus x = 0$, $x \oplus y > 0$ if $x \neq y$, and $\forall x, y : x \oplus y = y \oplus x$. Moreover, XOR has the triangle property, i.e., that $\forall x, y, z : x \oplus y + y \oplus z \geq x \oplus z$, which follows from the combination of the facts that $(x \oplus y) \oplus (y \oplus z) = x \oplus z$ and $\forall a \geq 0, b \geq 0 : a + b \geq a \oplus b$ [9].

Furthermore, XOR as distance metric is *unidirectional* [9] which means that for any point x and distance d, there is exactly one point y such that $x \oplus y = d$. To verify this, consider XOR'ing 1010 with some unknown number y and obtaining the distance 1100. The four digits of y will then cover all possible cases. In tabulated form:

$$\begin{array}{cccccc} & 1 & 0 & 1 & 0 \\ \oplus & y_4 & y_3 & y_2 & y_1 \\ \hline & 1 & 1 & 0 & 0 \end{array}$$

If $0 \oplus y_1 = 0$, then y_1 must be 0 since $0 \oplus 1 = 1$. Similar reasoning leads to the conclusion that $y_2 = 1$, $y_3 = 1$ and $y_4 = 0$. Thus, y is uniquely determined. The unidirectionality of XOR is contrary to the Euclidean distance metric where two points may be equidistant to another point, e.g. -5, 5 and the point 0, and ensures that there will not be two different groups of node to contact when looking up a hash value, see Figure 2.5. This means that only a single group needs to keep track of the what is near the hash value.

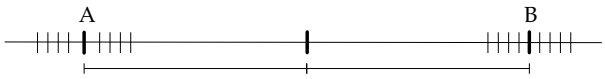

Figure 2.5: The group of nodes in the vicinity of A and of B are equally close to hash value in the middle. This cannot happen with the XOR *metric.*

Another important property of XOR is that it is symmetric [9] (i.e. $\forall x, y : x \oplus y = y \oplus x$), unlike the distance metric in a ring-based system such as Chord. With an Internet-wide system where the routing tables are always incomplete, this is important because it means that nodes that are close to each other and hence are more likely to know about each other, are also more likely to communicate. This communication helps keeping the most important part of the routing tables, the part that covers the nearest nodes, up to date. The symmetry is less important for a local area network where the routing tables can be allowed to grow to the size of the network.

2.2.3 Ensuring connectivity in the overlay network

To be able to scale to millions of hosts, Kademlia can not afford to store routing information about all nodes in the network at every node. But this is not a problem for a local area network with less than 20,000 nodes. Hence, a node in our system should never throw away information about other nodes.

It would, however, be very expensive to update all the available routing information regularly. Since an incoming message from a node A to a node B

has the side effect of telling node B that node A is alive, the total number N of necessary ping messages for $n \to \infty$ nodes would be (assuming nodes do not otherwise communicate):

$$N = n - 1 + n - 2 + \ldots + n - n = \sum_{i=1}^{n} (i - 1) = \Theta(n^2)$$

Since it *is* necessary to update the routing tables from time to time to ensure that the connectivity in the overlay network is not lost, we instead simulate the bucket approach in Kademlia. The routing table for a node is iteratively divided into $[2^i, 2^{i-1})$ ranges with $0 < i \leq m$ and i starting at m, as long as there are more than r nodes in the range that contains the node itself as shown in Figure 2.6.

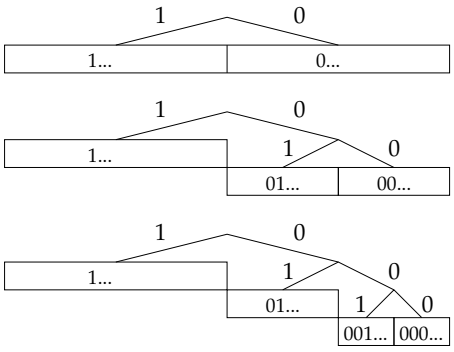

Figure 2.6: Division of the routing table for the node 000.... First step is to divide the routing table into two equal-sized ranges. All nodes with prefix 1 are in the leftmost range and nodes with prefix 0 are in the rightmost. When there are more than r elements in the range containing the node itself, that range is divided again. The division process stops when there are less than r nodes in the range that contains the node.

After the ranges have been constructed, a lookup is performed on a random hash value in each range. Each lookup returns information about r nodes, which ensures that the node has updated routing information for all ranges and that it knows its neighbourhood particularly well.

When a node joins the overlay network, it must perform a lookup for its own identifier followed by the above procedure. This ensures that it gets to know its neighbourhood and that the neighbourhood get to know it.

With this approach for updating routing tables, one can prove that the network will remain connected with very high probability, and that a lookup finds the closest nodes to a key in $O(\log n)$ time [9]. The probability of loosing connectivity depends on r, how often nodes crashes, joins and the time interval T_r between updates.

2.2.4 Ensuring data persistency

This section describes how data persistency is ensured by maintaining that the r nodes closest to a block store it. This can be violated by nodes crashing or joining.

Crashing nodes

A node A that crashes or otherwise leaves the network will result in some blocks being stored by only $r - 1$ live nodes. Another node B will then become responsible for storing these blocks and must somehow receive them. One way of doing this is to let all nodes send all the blocks they are responsible for at regular intervals, to the other $r - 1$ nodes that are responsible for the blocks. Eventually, B will then receive the blocks it is missing.

The time interval T_b at which the above procedure, which we refer to as *republishing* [9], is performed must be short enough to render it very unlikely that r nodes leave within it. Republishing is the corner stone in securing a high availability for the system. Unfortunately, with a straightforward implementation it is also very expensive when the system is storing many blocks. Thus we introduce two optimisations.

Instead of sending the entire blocks, we let each node send a *replication offer* message to the other responsible nodes. The message contains a list of block keys, and the receiver may then obtain any blocks it is missing either by the usual *get* procedure or directly from the sender. As long as crashes happen infrequently, this reduces the amount of data sent by orders of magnitude. For instance if a key is 4 *bytes* and a block is 4000 *bytes* the order of magnitude is 3.

Furthermore, from Kademlia [9] we get the optimisation that a node does not have to send a replication offer for a block if it has recently received a replication offer for that block. This is not necessary because the $r - 1$ other nodes responsible for the block must also have received the replication offer. With this scheme, only one node will have to send a replication offer for a given block and the other $r - 1$ offers may be saved.

Joining nodes

A node B that joins the network may end up being closer to a block than some of the r nodes that previously has been storing the block. The above republishing procedure will ensure that eventually B will receive a replication offer for the block.

The republishing procedure can easily be improved upon by having B ask some nodes to send a replication offer immediately, since B knows that it has just joined the network. It is enough to ask the nodes in the smallest subtree which consists of B and at least one other node. To see this, consider Figure 2.7 which shows the situation before and after B joins. Initially the node A

is the only node in the leftmost subtree and must thus store all blocks (since $r = 2$ and there is only one node in the rightmost subtree). When B enters, it will share the responsibility of storing the nodes in the leftmost subtree with A (since all nodes within this subtree are within the same order of magnitude apart from any other node outside the subtree). B will further take over from A the responsibility of half of the rightmost subtree (due to the way XOR distance works). But since A must have been storing the blocks from this subtree, it suffers for B to ask A to send a replication offer for the blocks that B should store.

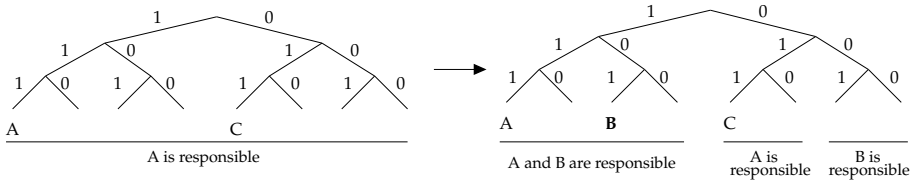

Figure 2.7: To the left is the hash space before B joins the network and to the right the hash space after B has joined (with $r = 2$). B takes over from A the responsibility of a subtree.

If there is more than one other node in the smallest subtree that contains B and another node, each of these other nodes will have been responsible for a different part of the subtree that B becomes responsible. Hence, B must ask all of them for replication offers.

2.2.5 Protocol

Having given a detailed description of the distributed hash table, we continue with a definition of what messages a client process participating in our system must support for the communication between the distributed hash tables.

Supported protocol primitives

The distributed hash table must be able to query for other nodes, request reading and writing of a block, offer blocks that should be replicated and ping nodes to see if they are alive. Besides these, there must be messages for the replies to these requests.

FIND CLOSEST (KEY): Sent when a node needs to discover the neighbourhood around a key. Upon receipt, a node should reply with a FIND CLOSEST REPLY message that contains the identifier keys for the r nodes it knows of closest to the key. Since no nodes are assured to know all available nodes, a node that wants to find the r nearest nodes may have to issue multiple iterative FIND CLOSEST messages to different nodes.

RETRIEVE BLOCK (KEY): Sent to a node to retrieve the block with the given key. If the node does not have the block, it returns a negative answer

NEGACK, otherwise it replies with a RETRIEVE BLOCK REPLY message that contains the block.

STORE BLOCK (KEY, BLOCK): Sent to a node to request that the node stores the block. The receiving node must henceforth from time to time ask the other neighbours around the key whether they maintain their replicate of the block; at first, the sending node will itself send out STORE BLOCK messages to the *r* closest neighbours. The receiving node must reply with an acknowledgement ACK, or a NEGACK if it was unable to store the block.

REPLICATION OFFER (KEY LIST): Sent to a node to ask it to check if it maintains the offered blocks. The receiving node should request any missing blocks with RETRIEVE BLOCK and reply to the initial request with an ACK.

SEND REPLICATION OFFER: Sent to a node to ask it to generate and send a replication offer to the sender. The replication offer should contain all keys of the blocks that the sender should store.

PING: Sent to a node to see if it is still alive. A live node should reply with an ACK message. Any message received counts as a ping reply, so it is only necessary to ping nodes that have not been heard from in some time.

The replies to the above are:

RETRIEVE BLOCK REPLY (BLOCK)): Sent to a node that issued a RETRIEVE BLOCK request.

FIND CLOSEST REPLY (HOST LIST): Sent to a node that issued a FIND CLOSEST request.

ACK: Sent to a node that either issued a STORE BLOCK, REPLICATION OFFER or a PING request indicating that the message has been received and accepted.

NEGACK: Sent instead of an ACK in case a request was received but could not be fulfilled.

Transport protocol

The User Datagram Protocol (UDP) is used for transmitting the protocol primitives over the network. UDP is chosen mainly because all messages easily can be made to fit within the limit on the size of the UDP datagrams (which is about 64 kb), and using UDP saves the connection-setup latency of a connection-oriented protocol such as Transmission Control Protocol.

UDP does guarantee the integrity of the received message so that either the message is received intact or it is not received at all. But our application level protocol must itself ensure that messages are resent if they are lost, and that

duplicates are suppressed. The first can be taken care of easily by means of timeouts, and the latter is ensured via sequence numbers. The sequence numbers are needed anyway for matching an incoming reply with a previously sent request.

Message layout

All messages are byte sequences and consist of at least the following:

- An identifier for determining which protocol primitive the message is (1 byte).
- A sequence number that is used for matching replies to requests (4 bytes).

A message request and a reply may furthermore contain a payload that corresponds to the input and output of the protocol primitive, respectively. The size of the payload may vary and is indicated via length fields in the messages; it must, however, not exceed the UDP maximum datagram size.

2.2.6 Algorithms

This section provides a high-level description of the various algorithms in the distributed hash table. The algorithm uses the global variables r and t that describes the number of replications and the time between republishing, respectively.

Finding the closest nodes to a key

When finding the r closest nodes to a given key k, the requesting node B first looks up the r closest nodes currently known to B. These nodes are bound to a list of *currently closest nodes*.

B iteratively asks each of nodes in *currently closest nodes* which r nodes they believe is the closest node. If one of the nodes received in the answer is closer than one of the nodes in *currently closest nodes* this new node is included instead of the node the furthest away. This process repeats until no closer nodes is received.

<the r closest nodes to k> **Closest (**key k**)** = "

1. Build a list *currently closest nodes* with r nodes closest to k
2. For each node A in *currently closest nodes*
   (a) Send the protocol primitive FIND CLOSEST (k) to A
   (b) If one of the nodes received in the answer is closer than a node in *currently closest nodes* replace the node with the one the furthest away and go to step 2.
3. return *currently closest nodes*"

Joining the network

The requirement for joining the network is a known node A in the network, and the following steps:

Join (a known node A**)** = "

1. The joining node B uses Algorithm Closest(k_B) at A to find the closest nodes to itself.
2. Refresh the routing table according to Section 2.2.3."

After each query the view of B expands, and can be summarised as:

- 1st view is the know node B.
- 2nd view is the closest nodes to B
- 3rd and final view is the view of the complete network

Republishing

A file block is republished every t minutes to compensate for nodes failing. This aids in ensuring that at least r closest nodes store the block. Republishing a block with key k_b amounts to performing the following steps:

Republish () = "

1. Find the r closest nodes by using Algorithm Closest(k_b).
   (a) Send a REPLICATION OFFER (k_b) to the r nodes.
2. If a node A receives a replication offer of block key k_b and it does not store that block, it sends a RETRIEVE BLOCK (k_b) to whom it received the offer from."

Hence A now stores the block k_b thus fulfilling the replication offer.

To avoid sending too many replication offer messages, when a block republish is due the node checks if any other blocks are also due and creates a list l_k containing the block keys. It then sends a replication offer using REPLICATION OFFER (l_b) to its r closest nodes.

The worst case scenario is that the r nodes are very closely located in the hash space and fail simultaneously, which means that they are the only ones storing the blocks. If this happens before any of the nodes have republished the blocks, the blocks are unavailable until any of the nodes re-enters the network.

2.3 File system

The interface of the file system component has means of opening and closing the file and the operations *seek(bytes, relative, forward)*, *read(bytes)*, *write(buffer)*, *symlink(name)*, *remove()* and *makedir(name)*, that work with directories and sequential files named by unique paths. The task of the file system component is to map this into keyed blocks that can be stored in the distributed hash table.

Files are keyed on their complete paths, including the file name itself. To lookup the file with the file path p, the hash function H is applied to yield the key $H(p)$. This key is used to retrieve a *file identifier block* that contains some house-keeping information and a file identifier that must be unique for each file and for each version of the file.

Keys for the blocks with the actual contents of a file are obtained by concatenating the block number with the file identifier and hashing the result, so the key for the i'th contents block of the file with identifier f is $k_{f,i} = H(fi)$. A file f_S smaller than B bytes simply consists of a single partial block, keyed $k_{f_S,1}$, whereas a large file f_L is split up into a number of full-sized blocks, keyed $k_{f_L,1}$, $k_{f_L,2}, \ldots, k_{f_L,n-1}$ and a partial block with the remainder, $k_{f_L,n}$.

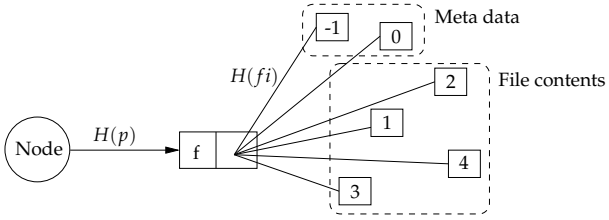

Figure 2.8: The structure of a file. First the hash of the file path, $H(p)$, is used to retrieve the file identifier f which is then used to to retrieve the file meta data and contents.

File meta data such as size, time stamps and access control lists are placed in the blocks $k_{f,0}$, $k_{f,-1}$, $k_{f,-2}$, ... for the file f. With this scheme, user-defined extended meta data can be added easily. Figure 2.8 illustrates the structure.

An immediate consequence of the indirect approach is that when a file is renamed or moved from p to p' resulting in a different hash, $H(p') \neq H(p)$, only the file identifier block must be moved to other nodes, and not the entire file. From a robustness point of view, the most useful property of the indirection is that two nodes can start distributing the blocks of a file and then only replace the file identifier block when they are finished. As will be explained in Section 3.2, this ensures that the contents of one node's version of a file is not interleaved with the contents of another node's version.

Directories have a special bit set in the house-keeping information in their file identifier block but are otherwise just ordinary files that contain a list of directory and file names. No other meta data about the files are stored in the directory file lists. The name listings are only needed for listing the contents of

directories or for building a tree of the file system (for searching or displaying purposes) and not for ordinary file reading since files are looked up directly by hashing their paths.

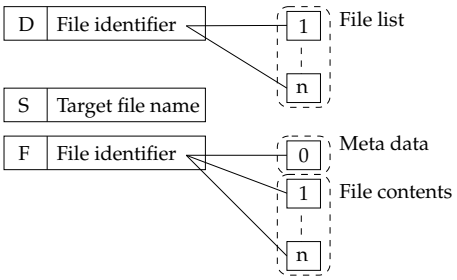

Figure 2.9: Illustration of file identifier blocks for directories, symbolic links and ordinary files.

Symbolic links can be implemented by setting a special bit in the file identifier block and letting the contents of it be the path of the file that the link refers to. Figure 2.9 summarises the three possible types of files.

2.3.1 Algorithms

This sections gives a description of the high-level algorithms associated with the file system's operation. Each algorithm works on an input file path, denoted p.

Creating a file

Creating and saving a new file f with full path p involves the following sequence of steps:

Create (file path p**)** = "

1. Generate the file identifier block b_f.
2. b_f consists of a file type flag, file size, meta data size, a random bit sequence, modified date and creation date. k_f is the identifier key which consists of a hash of the random number, version and date.
3. Split the meta data file into n_m and for each meta data block $m_i = m_0..m_{n-1}$:
   (a) Find the r closest nodes by using Algorithm Closest($k_{f,-i}$).
   (b) Store the block in each of the r nodes using STORE BLOCK ($k_{f,-i}$,m_i)
4. Split the file into n_f and for each file block $b_i = b_1..b_n$:

(a) Find the r closest nodes by using Algorithm Closest($k_{f,i}$).

(b) Store the block in each of the r nodes using STORE BLOCK ($k_{f,i}$,b_i).

5. Store the file identifier block b_f using STORE BLOCK ($H(p)$,b_f)
6. Add a line with the file name without the path to the appropriate directory file."

Looking up and retrieving a file

Looking up and retrieving a file f with full path p involves the following sequence of steps:

<the file of p> **Retrieve (**path p**)** = "

1. Retrieve the file identifier block b_f by using GET ($H(p)$).
2. Retrieve the file meta data m_f by requesting blocks $k_{f,0}$, $k_{f,-1}$, $k_{f,-2}$, ... using GET ($k_{f,i}$), see Figure 2.10.
3. Retrieve the remaining blocks of the file sequentially by incrementing the block number – starting with $k_{f,1}$.
   (a) Use $k_{f,i}$ to find the closest node and query it for the block assigned to that key.
4. The file is completed once the block number reaches the filesize divided by the block size."

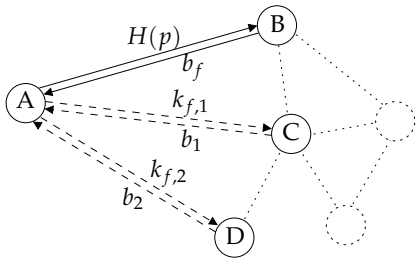

Figure 2.10: Example of a file retrieval where A fetches the first block b_1 from C and the second block b_2 from D.

Updating a file

Updating an existing file f with a file identifier k_{old} involves the following sequence of steps:

Update (file path p**)** = "

1. Generate a new file identifier k_f and its block b_f.

2. b_f consists of a file type flag, file size, meta data size, a random bit sequence, modified date and creation date. k_f is the identifier key which consists of a hash of the random number, version and date.
3. Split the meta data file into n_m and for each meta data block $m_i = m_0..m_{n-1}$:
   (a) Find the r closest nodes by using Algorithm Closest($k_{f,-i}$).
   (b) Store the block in each of the r nodes using STORE BLOCK ($k_{f,-i}$,m_i)
4. Split the file into n_f and for each file block $b_i = b_1..b_n$:
   (a) Find the r closest nodes by using Algorithm Closest($k_{f,i}$).
   (b) Store the block in each of the r nodes using STORE BLOCK ($k_{f,i}$,b_i)
5. Store the new file identifier block b_f using STORE BLOCK ($H(p)$,b_f)."

Using this algorithm, it is very unlikely for two or more clients updating a file at the same time to leave the system in an inconsistent state, since for that to happen the random IDs would have to be the same number. In such a case, the data of the node finishing last will be the new data.

The data blocks of the previous version of the file are left as garbage for the garbage collector to clean up.

Deleting a file

Deleting a file or directory with full path p is done using the following algorithm.

Remove (path p**)** = "

1. If p is a sequential file or symbolic link:
   (a) Get the directory list containing the file.
   (b) Remove p from the directory list.
2. Else p is a directory:
   (a) Get the directory list containing the directory.
   (b) Remove p from the directory list.
   (c) Recursively run down the directory structure from the position of p.
   (d) While doing this, delete all directories and files from their associated directory."

This process will leave the blocks of each file in the system for the garbage collector to deal with. The reason for the recursive deletion of entries in the deleted directory is due to the structure of file lookups which reply on the entries of a directory to determine if the file exists.

Renaming and moving a file

Renaming and moving files in the system is very convenient due to the structure of the distributed hash table. The file identifier holds the necessary information to find the file blocks and implicitly holds the file name in form of its hashed block key.

Since these two characteristics are not correlated, it is possible to move or rename the file simply by generating a new block key using the new file path.

2.3.2 Data organisation

Having given a detailed description of the file system, we continue with how files, directories, symbolic links are represented in the file system.

The uniqueness of a file is guaranteed by letting the file identifier consist of the creation date of the file, a random bit sequence and a time stamp when last modified.

The file system utilises three different file types, namely directories, sequential files and symbolic links. Directories and sequential files share the same identifier block structure illustrated in Figure 2.11. The random number, creation time stamp and modification time stamp, used to generate the identifier key, ensures that the system will not express name clashes regarding keys. Symbolic links vary from directories and files by not having associated file blocks and having to point to other identifier blocks. An in depth description of the files is found below.

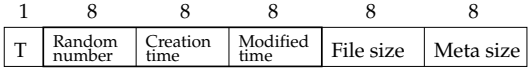

Figure 2.11: The structure of an identifier block with size in bytes. Its components are file type, a random number, creation time, modification time, file size and meta data size. The bold box represents the components used to generate the identifier key.

Directory

A directory consists of an identifier block which references to a number of blocks containing the directories, files and symbolic links within directory. The number of blocks is determined by the file size and the maximum block size. The same is true for meta data blocks.

The directory listings consist of a number of entries delimited by a newline character. The entries are file names stripped from the path. All housekeeping information, such as security and time stamps, is located in the meta data blocks.

Sequential file

Sequential blocks consist of an identifier block which points to the blocks containing the file contents. The number of blocks is determined by the file size and the maximum block size, as with the meta data blocks.

The data blocks contain the user data.

Symbolic link

A symbolic link is an identifier block that basically reference other identifier blocks which may either be other symbolic links or plain sequential files. The identifier block contains a file type flag, as in directories and sequential files, and the target's file path. The targeted file path contained in the symbolic link's identifier block can obtain the maximum length of the maximum block size minus the file type flag's size.

2.4 Garbage collector

The distributed hash table and file system components leaves *garbage* in terms of blocks that are not needed anymore. These blocks must be removed from the system because they waste resource, not only disk space but also network bandwidth due to republishing.

The following sections explain why the garbage occurs and how it can be safely removed.

2.4.1 The garbage

The garbage that may occur is:

- Blocks of a file that is not in any directory. This could be because the file has been deleted, or it could be because the node writing the file never completed the write so that the file name was not inserted into the relevant directory.
- Blocks of an old version of a file. Because of the file update algorithm in Section 2.3.1, the blocks of old file versions are never overwritten but simply ignored.
- Blocks from a version of a file that is in conflict with another version of the same file. These conflicts may arise when multiple nodes are writing the same file simultaneously, or during a *network split* where the network is temporarily partitioned into two or more subnetworks. One of the versions must be chosen and the other removed.
- Blocks stored by nodes that are no longer among the r closest nodes to the blocks. This happens when new nodes join the system.

Some of this garbage could be alleviated by letting the distributed hash table and file system algorithms clean up after themselves; for instance, the file update algorithm could actively remove the old blocks. But pushing the complexity to the garbage collector makes the algorithms much simpler: without a garbage collector, a transaction protocol with explicit commit operations would be needed to ensure that the system cleans up after algorithms that terminate without finishing (e.g. due to a crash).

2.4.2 Collecting garbage

We distribute the responsibility of collecting the garbage and let each node examine its own blocks to avoid a central bottleneck. For each block b_i, a node should do the following:

1. Check that b_i has not been received within some time interval T_g. If it has, the block should not be garbage collected since it may be part of a larger file write that has not been completed yet.
2. Else remove b_i if any of the following checks fail:
   (a) If b_i is a file identifier block then:
      i. Check that the file name for b_i appears in any directory.
      ii. Check that there is not a newer version b_i' of b_i that b_i is in conflict with.
   (b) Else if b_i is not a file identifier block:
      i. Check that the file identifier for b_i exists.
      ii. Check that b_i is part of the latest version of the file it belongs to.
   (c) Check that the node is among the r closest to b_i.

With this procedure, file identifier blocks, e.g. of deleted files, are removed before the blocks with the file data.

Note that the above formulation assumes that it is possible to get from a data block to its file identifier block and from a file identifier block to the path of the file. To realise this, we need to extend a data block to contain the key for the file identifier (m bits) along with the actual file data. And a file identifier block must contain the path of the file in addition to the other data defined in Section 2.3.

It would be possible to avoid this overhead by means of a mark-and-sweep garbage collection algorithm that marked live blocks by visiting the file identifier for each file in each directory and afterwards removed the unmarked blocks. But then each node would have to retrieve all file identifiers, even if they did not store blocks from some files.

The next sections will discuss how the checks in the garbage collection procedure can be designed.

Detecting whether a file appears in any directory

This is easily done when the file identifier block contains the entire path name of the file. Simply look up the directory and see if the file name is in it. To avoid removing blocks because of a temporary failure in the network, the block should be timestamped with the time that the condition occurred and then checked again at some later run before it can be removed.

It is unfortunately possible that a live block is removed during a network split even with the time stamp protection if the split lasts longer than the time stamp delay. It is unlikely that data will be completely lost, though, since it requires:

- That all r replicas of a the directory is stored in one part of the network.
- That all r replicas of a block is stored in the other part of the network.
- That the garbage collector for all of the r nodes storing the block decide to remove it.

Detecting conflicting versions

When multiple nodes are writing to the same file, the file data blocks will not interfere with each other (since they have different keys), but the file identifiers will. It is possible that the system will end up with multiple different file identifier blocks unless one of the nodes overwrites all of the file identifier blocks of the other nodes.

The problem is even worse during a network split since the conflict will emerge even if the nodes do not write to the file at the same time. To see this, consider the example in Table 2.1 where a network consisting of the nodes A, B, C, D, E, F, G is partitioned into the two subnetworks A, B, C, D and E, F, G. After the network is restored, there are multiple versions of the file identifier block.

The conflict can be detected by retrieving the replicas of the file identifier block from the r closest nodes and comparing them to the locally stored identifier block.

The question is how to resolve the conflict. A reasonable strategy is to be to remove the local block if its modification time stamp is older than the time stamp of one of the retrieved blocks. If the timestamps happen to be the same, choose the block where the random bit sequence interpreted as an integer is highest.

For this to work, the clocks of the nodes participating in the network must of course be synchronised from time to time.

A	B	C	D	E	F	G
		b	b	b		
			b_f	b_f	b_f	
b is written						
A	B	C	D	E	F	G
		b	b	b		
			b_f	b_f	b_f	
a network split occurs						
A	B	C	D	E	F	G
	b'	b'	b'	b''	b''	b''
	b'_f	b'_f	b'_f	b''_f	b''_f	b''_f
f is updated						
A	B	C	D	E	F	G
	b'	b'	b'	b''	b''	b''
	b'_f	b'_f	$\mathbf{b'_f}$	$\mathbf{b''_f}$	$\mathbf{b''_f}$	b''_f
the two networks merge						

Table 2.1: The file f, consisting of the data block b and the file identifier b_f, is written before the network is partitioned. During the network split, f is updated in both subnetworks (but not necessarily at the same time). When the network is restored, it is possible to retrieve different versions of b_f.

Detecting whether the file identifier exists

Given a data block b_i with the key for the file identifier, the r closest nodes should simply be asked for the file identifier. But since failure to retrieve the file identifier may stem from a failure in the network, the block b_i should be timestamped and first be allowed to be removed at a later run of the garbage collector.

Detecting whether the block is outdated

Even if the file identifier for the block b_i exists, it may be a file identifier for a new version of the file. Hence, if the retrieval procedure finds a file identifier block, further check whether the key for b_i is in the set of hash values that can be obtained from the file identifier. If it is not, the block should be removed.

Detecting a block stored by a node not among the r closest

This can be detected by keeping track of when the block was lasted offered in a replication offer by a node. If it is more than T_b time ago then either another

node or the node itself would have offered it for replication if the node were still among the r closest. To appreciate this, consider the following example:

1. The network consists of nodes A, B, C with $r = 3$ and each node store block b_i.
2. A, B and C send out replication offers for b_i to A, B and C at regular intervals.
3. D joins the network and is closer to b_i than C. Eventually, D will also get b_i after having received a replication offer.
4. A, B, C and D now send out replication offers to A, B and D.

Thus the replication offer time stamp for b_i in C is henceforth not updated, and C can safely remove b_i after some time.

2.4.3 When to run the collector

One last problem that must be solved is when to start the garbage collector.

A memory garbage collector can keep track of the free memory and only start when the memory is running out. In our system, all blocks must be republished as long they have not been removed. Hence, waiting for the disk space to run out before starting the garbage collector may be a problem since it will waste network bandwidth.

Another possibility is to monitor the network activity and start the garbage collector whenever the activity is low. A convenient property of the algorithms described in the preceding sections is that they allow the garbage collector to be incremental; the garbage collector can check one block, stop and then be restarted with the next block.

Whatever the approach, it should be combined with randomisation so that all nodes do not start collecting garbage at the same time – this could use all available network bandwidth for some time.

Chapter 3

Characteristics

This chapter will analytically characterise the design described in Chapter 2 and try to highlight the strengths and weaknesses. Even if blocks are not completely lost, they may still be temporarily unavailable for reasons we explore in Section 3.1. Section 3.2 explains the consistency guarantees that the system provides for entire files. Section 3.3 quantifies the probability of data loss (under some assumptions), and finally Section 3.4 examines the performance characteristics.

3.1 Availability of blocks

Under normal operation, all blocks are always available because of the replication in the distributed hash table. There are, however, some situations where abnormal conditions in the network might affect the availability of certain blocks.

3.1.1 Availability during a network split

Consider a network M which is partitioned into the two separate subnetworks M_1 and M_2 by a network failure. Under these circumstances, it may not be possible to retrieve a certain block in one of the subnetworks. Let $|M_1|$ and $|M_2|$ be the number of nodes in M_1 and M_2, respectively. A block is then available from a node in M_1 if one of the r replicating nodes is in M_1. We will proceed with deducing the probability P_{M_1} of this with the assumption that the hash function distributes the block evenly over the nodes.

In the case where $|M_2| < r$, at least one of the r replicating nodes will be in M_1 so the probability is $P_{M_1} = 1$.

When $|M_2| \geq r$, we instead consider the probability of a block not being available. A block is unavailable from a node in M_1 if and only if all r replicating nodes are in M_2. The number of combinations where all replicating nodes

are in M_2 is:

$$\binom{|M_2|}{r} = \frac{|M_2|!}{r!(|M_2| - r)!}$$

The number of combinations for the entire network M is:

$$\binom{|M|}{r} = \frac{(|M_1| + |M_2|)!}{r!(|M_1| + |M_2| - r)!}$$

The probability for a block to be unavailable from any node in M_1 must then be:

$$P_{M_1,\text{un}} = \frac{\binom{|M_2|}{r}}{\binom{|M|}{r}}$$

Hence, the probability for a block to be available from some node in M_1 is:

$$P_{M_1} = 1 - \frac{\binom{|M_2|}{r}}{\binom{|M|}{r}} = 1 - \frac{|M_2|!(|M_2| + |M_1| - r)!}{(|M_2| - r)!(|M_2| + |M_1|)!}$$

Note that P_{M_1} is the probability of a single block being available. The probability of b blocks being available is simply $(P_{M_1})^b$. Examples of how the probability changes as a function of b and with different values of r are shown in Figure 3.1 for a symmetric network split and in Figure 3.2 for an asymmetric network split.

As Figure 3.1 shows it is unlikely that it is possible to retrieve an entire file during a symmetric network split if the file consists of many blocks and moderate values of r are chosen, e.g. $3 \leq r \leq 5$. Small files that only consist of two blocks (the file identifier block plus one data block) are still retrievable with a probability above 90%, though.

During an asymmetric network split where the requesting node is part of the larger subnetwork, the probability of availability is much better as Figure 3.2 shows. Only very large files will not be retrievable. For a requesting node in the smaller subnetwork the probability is unfortunately so low that the system is completely useless. For instance, the probability of two blocks being available with $r = 7$ is only:

$$\left(1 - \frac{90!(90 + 10 - 7)!}{(90 - 7)!(90 + 10)!}\right)^2 = 28.4\%$$

Hence, symmetric network splits will severely hamper the availability of large files requested from any node, whereas asymmetric network splits mostly affect the nodes in the smallest subnetwork.

3.1.2 Availability without a network split

Even when the network is not partitioned, it is possible that a block temporarily cannot be retrieved:

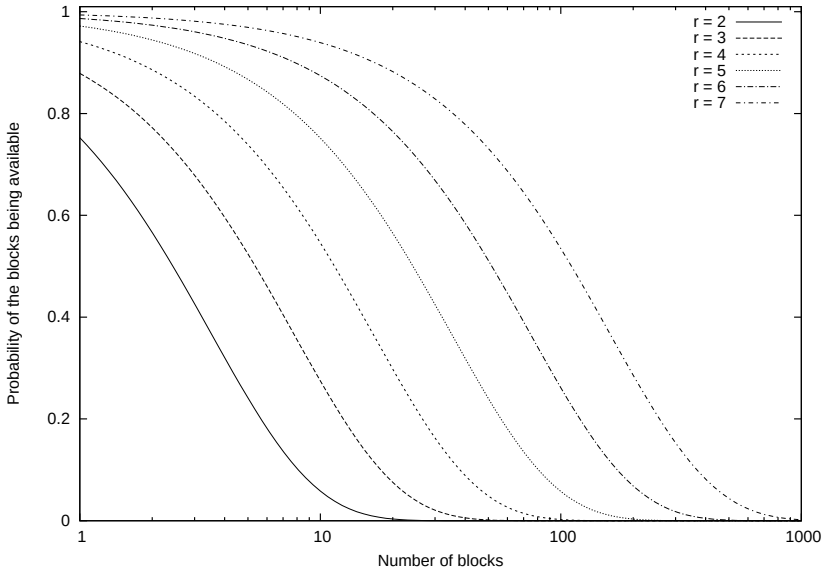

Figure 3.1: The probability of a given number of blocks being available in the subnetwork M_1 during a network split for different values of r with $|M_1| = |M_2| = 50$.

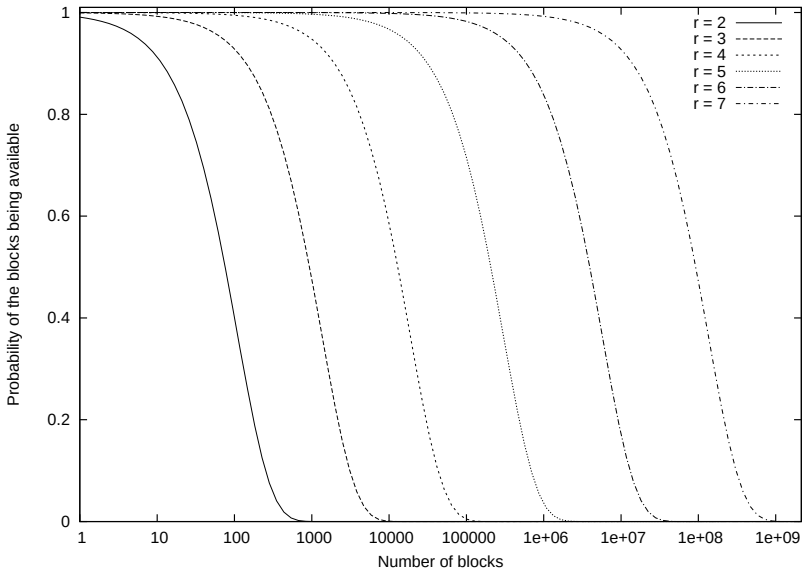

Figure 3.2: The probability of a given number of blocks being available in the subnetwork M_1 during a network split for different values of r with $|M_1| = 90$ and $|M_2| = 10$.

- If many nodes join the network within a short period of time, r of them may end up closest to a block although it is not possible to retrieve the block from them, yet.
- If a block is created in a split network that is subsequently reconnected, the block may not reside on the r nodes closest to it.

Since the hash function distributes keys uniformly in the hash space, it is unlikely that just r nodes joining could end up closest to a block – their identifiers are expected to be mapped to different places in the hash space. This is also the case if the network was split into two equal sized subnetworks; the hash function is expected to map to nodes on both sides of the network after the connection has been reestablished.

Note that after some time the unavailable blocks will be republished to the appropriate nodes as described in Section 2.2.6, and the blocks will then be available again.

3.2 File consistency guarantees

On a local centralised file system all file references are kept on the same machine and all file operations are processed on the same caches with local file system. When multiple users access the same file this ensures that all users operate on the same copy of the file and see each other's changes.

With a distributed file system, propagation delays and the lack of central storage space results in concurrency problems. Weaker consistency guarantees are then usually given to avoid abysmal performance. For instance, NFS operates with a cache at both the client and server side [2]. Sometimes the cache on the client slide will not be consistent with the actual file on the server, and although NFS tries to minimise this problem, e.g. by adaptively lowering the cache refresh intervals, it is only an approximation of the guarantee from a local file system.

Our system provides a guarantee which is actually stronger: When a file is requested we can guarantee that all received blocks belong to the same version of the file. The data are not intermingled even if multiple nodes write concurrently to the same file.

To see this, recall from Section 2.3.1 that the file identifier is looked up as the first step when a file is requested. This ensures that only blocks from that file identifier version are requested. A file modification does not modify or delete existing blocks except for the file identifier block which will always contain a new file identifier hash. Hence, concurrent write and read operations do not interfere.

However, our system is worse in guaranteeing that all nodes have the same view of what is the latest version of a file. If multiple nodes write to the same file, it is possible that some of the r nodes storing the file identifier block will

end up with one version and the rest of the r nodes with another. Even without concurrent writers, the same may happen during a network split – when the network is reconnected, different versions of the same file identifier exist.

The latter situation is theoretically unavoidable if normal operation is permitted during a network split since the disconnected nodes cannot communicate.

The consequence is that different nodes may retrieve different versions of the same file. We can only guarantee that eventually the problem will disappear since the garbage collector will detect the two conflicting versions and choose one of them, as described in Section 2.4.2.

3.3 Loss of data

Since nodes may crash or otherwise leave the system, all of the r nodes that are replicating a number of blocks may be unavailable. In the worst case, none of them ever rejoins and the blocks are completely lost.

In the following, we investigate the probability of this event to happen as a function of the replication constant r and the republishing time interval T_b. With the assumptions presented in Section 3.3.1 it turns out the probability can be quantified, as shown in Section 3.3.2. Section 3.3.3 discusses the implications of the estimate of the probability.

3.3.1 Analysis assumptions

First, we are considering the worst case so define a node failure to be a complete node breakdown where all data on the node is lost. If the node afterwards rejoins, it is as if it were a completely new node. This point of view makes it possible to consider node failures as binary events – either a node is up and functioning correctly, or it has failed and left the system.

Furthermore, we assume that the network under consideration will have more than r nodes running at *any* given time so that it is possible to maintain r replicas even if $r - 1$ nodes fail. We also assume that the republishing algorithm given in Section 2.2.6 is correct in maintaining the r replicas. For convenience, we assume it republishes the data instantly but the model we present is essentially the same as long as the algorithm completes much faster than T_b.

The last assumptions we make concern the nature of node failures:

1. The probability of a node failing is independent of the time of day and the day of the year.

2. Different nodes fail independently of each other.

The first assumption is obviously not true for any real system since the probabilities vary over time. But a fixed overestimate of the probability can be obtained by choosing the greatest probability for a node over time. The second

assumption can be claimed to be true for such events as hardware failures (e.g. disk crashes), but does not hold with failures that affect multiple machines, such as lightnings or computer viruses.

The two last assumptions taken together means that the nodes fail with a constant rate λ (e.g. one node per day).

3.3.2 The probability of loss of blocks

Given the assumptions in the preceding section, we can model the probability of losing blocks. We proceed by doing that in three steps.

If all blocks are replicated r times, obviously at least r nodes must fail for any blocks to be lost. Hence, we must obtain the probability of r or more nodes failing, or more general, precisely k nodes failing. This is the first step.

But r or more nodes failing is in fact not a sufficient condition for data loss since the failed nodes may not have any blocks in common. If the failed nodes do not have any blocks in common, there will still be at least one replica of each block somewhere in the system and no data is lost. Thus we must also obtain the probability that given that k nodes have failed (where $k \geq r$), at least r of them have blocks in common. This is our second step.

The third step is to combine the results. We start by computing the probability of k nodes failing within some time interval.

The probability of k nodes failing

The probability of k nodes failing within some time interval is easily found if we model the system as a *Poisson process* [18]. The model is valid because our system with the assumptions from Section 3.3.1 satisfies the following properties [21]:

- The number of node failures in non-overlapping time intervals of a given length are independent for all intervals. This is true because the failure rate is constant λ.
- For a sufficiently small time interval the probability of a single node failure is proportional to time. This is also a consequence of the constant failure rate.
- The probability for two or more node failures in a sufficiently small time interval is essentially zero. This follows from the assumption of the independence of the node failures. The probability of two nodes to fail in a small interval is the product of the two probabilities of each node to fail in the interval – this is much smaller than the probability of one of the nodes to fail.

For a Poisson process, the probability of k events happening within the time interval t is distributed as [18]

$$P_f(t,k) = \frac{(\lambda t)^k e^{-\lambda t}}{k!} \tag{3.1}$$

If we define the event to be a node failure, $P_f(t,k)$ gives the probability that k nodes have failed within the time interval t. Now observe that the k nodes must fail within T_b time – else the blocks would be republished so that at least r nodes were storing the block again. The probability we are looking for is thus $P_f(T_b,k)$ so we can proceed with the next step.

The probability of failed nodes having blocks in common

We now assume that k nodes have failed with $k \geq r$ and proceed with deducing the probability that at least r of the failed nodes have some blocks in common, or in other words that data is lost. The deductions are a little tricky because the blocks are distributed to the nodes in accordance with the XOR topology examined in Section 2.2.2.

Given a hash space with n nodes, divide the nodes into subtrees that are as small as possible but still contains at least r nodes each. An example is shown in Figure 3.3. With the assumption that the hash function distributes the nodes evenly in the hash space, each subtree will then contain between r and $2r-1$ nodes.

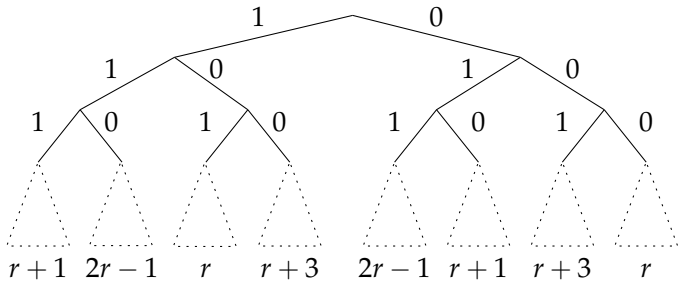

Figure 3.3: Example hash space where the nodes have been divided into eight subtrees. In this example, $r+1$ nodes are in the subtree where the node keys begin with 111, $2r-1$ nodes are in the subtree where the node keys begin with 110, etc. It was possible to divide the hash space into subtrees that each contains between r and $2r-1$ nodes because the hash function is assumed to distribute the nodes evenly.

Because of the XOR metric the nodes in each subtree are responsible for all blocks that have keys in their subtree, but not for any other blocks. This simplifies the analysis much because we only need to consider what happens when nodes fail inside a subtree.

Now observe that r nodes failing in any of the subtrees may cause data loss: the blocks in a subtree will certainly be lost if the subtree contains r nodes; if

the subtree contains $2r - 1$ nodes, the blocks may or may not be lost (see Figure 3.4).

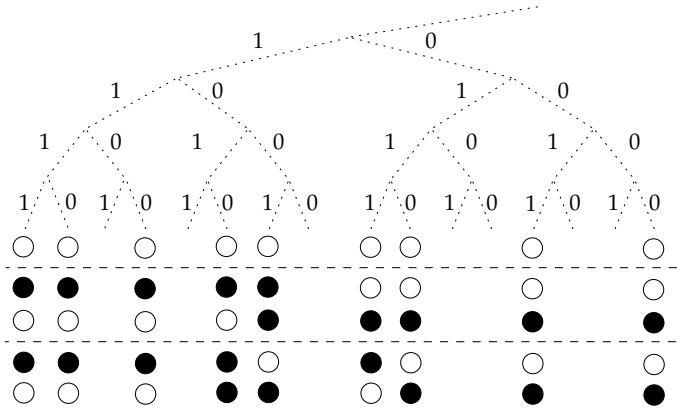

Figure 3.4: An example of a subtree with $2r - 1 = 9$ nodes ($r = 5$). The second and third row of nodes show possible failure outcomes that cause data loss (the black nodes are failed nodes). The fourth and fifth row show outcomes that do not cause data loss.

We assume the worst and pretend that r failed nodes in any subtree leads to lost blocks. Thus the probability of at least r failed nodes having blocks in common given that k nodes have failed can be reformulated as being the probability of at least r of the k nodes being in the same subtree.

The probability of at least r out of k nodes being in the same subtree can be found by counting. First choose one node arbitrarily to be a failed node. The failed node will be in one of the subtrees. Then the probability of $r - 1$ nodes failing in the same subtree out of the $n - 1$ nodes left must be calculated. We do this by counting the total number of ways the failed nodes may occur in in the system and counting the number of ways that leads to $r - 1$ failed nodes in the same subtree:

- There are $k - 1$ failed nodes left to choose from so the failed nodes may in total occur in $\binom{n-1}{k-1}$ different ways over the network.
- There will be $r - 1$ more failed nodes in the same subtree if and only if $r - 1$ nodes from the $k - 1$ failed nodes are in the subtree. This leaves $k - 1 - (r - 1) = k - r$ node failures to be distributed arbitrarily over any of the remaining $n - r$ nodes in total, which can be done in $\binom{n-r}{k-r}$ ways.

The counting of node failure outcomes leading to data loss is illustrated in Figure 3.5.

Hence, the probability P_s of losing data in a network of size n given that k nodes have failed must be at most (since we assumed the worst during the analysis):

$$P_s(k) = \frac{\binom{n-r}{k-r}}{\binom{n-1}{k-1}} = \frac{(k-1)!(n-r)!}{(k-r)!(n-1)!} \tag{3.2}$$

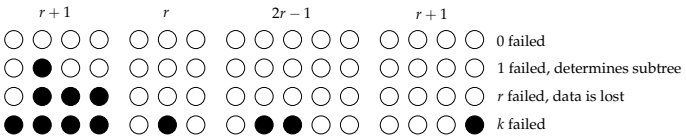

Figure 3.5: Example of node failure outcomes where $n = 16$, $k = 8$ and $r = 3$ that lead to data loss (the black nodes are failed nodes). The nodes have been divided into subtrees that contain r to $2r - 1$ nodes. First one node fails which gives a particular subtree. Then $r - 1$ nodes must fail in the same subtree to cause data loss. Finally, the remaining $k - r$ node failures may happen on arbitrary nodes.

We are now ready to proceed to step three.

Combining the results

The probability that both k nodes have failed and that the data is lost given that k nodes have failed is the product of the probabilities for these two events, $P_f(T_b, k)P_s(k)$. Since data loss may occur for any $k \geq r$, the probability P_l of losing data is

$$P_l = \sum_{k=r}^{n} P_f(T_b, k)P_s(k) \tag{3.3}$$

P_l is plotted as a function of T_b for various values of r in Figure 3.6. Larger values of r and smaller values of T_b give a lower probability of data loss.

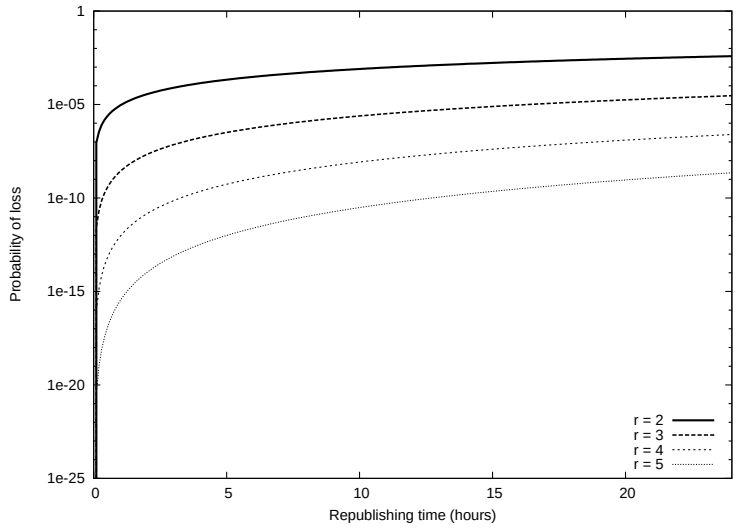

Figure 3.6: The probability of losing blocks (Eq. 3.3) with $n = 96$ and $\lambda = 1/24$ so that one node crashes every 24 hours.

Equation 3.3 gives an estimate of the probability of losing blocks in one time period T_b. There will be many such time periods when a system is put

into operation. Observe that for the system not to lose data at all, it must not lose data in any of these time periods. Thus if the network conditions do not change, the probability of *not* losing data over q periods is $(1 - P_l)^q$. Then the probability $P_l(q)$ of losing data over q periods is:

$$P_l(q) = 1 - (1 - P_l)^q \tag{3.4}$$

$P_l(q)$ is plotted as a function of q in Figure 3.7. When $r \geq 3$ it takes more than 10,000 years before the probability of data loss is 50% with $n = 96$, $\lambda = 1/24$ and $T_b = 1$ hour. This shows that the design of the distributed hash table is, at least theoretically, extremely robust.

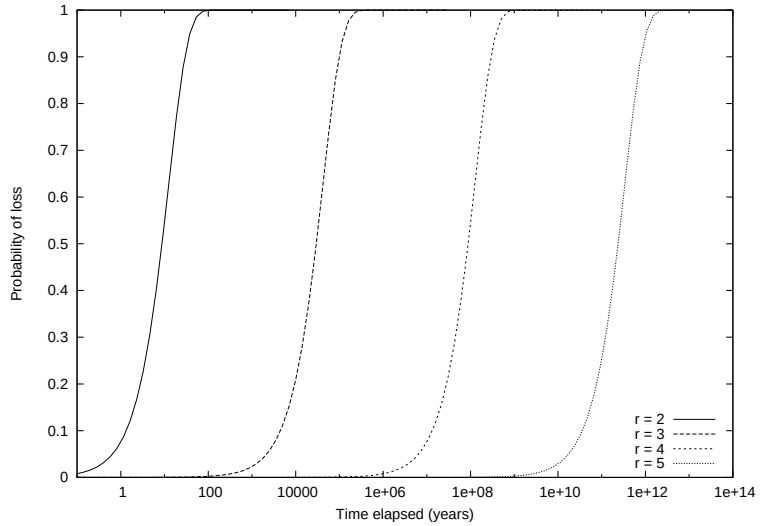

Figure 3.7: The probability of losing blocks (Eq. 3.4) with $n = 96$, $\lambda = 1/24$ and $T_b = 1$ hour.

3.3.3 Implications of the probability

As Figure 3.7 shows, the preceding analysis indicates that data loss is very unlikely with appropriate values of r and T_b. It should be taken into account that the analysis is for a system in normal operation (with no virus attacks, no lightnings etc.). But it should be stressed that the assumption is that the blocks on the failed nodes are completely lost. In a realistic environment, it seems more likely that the data will in fact be available as soon as the nodes have been rebooted.

In spite of this, the analysis is still useful as an estimate of the availability of all blocks at all points in time. Hence, the analysis indicates that our system will be highly available. It should also be kept in mind that some users may be served even if some blocks are missing temporarily since only a fraction of all files will be accessed at any given moment.

Apart from the insights that Eq. 3.4 provides into how the system reacts over time with failures, the equation can also be used to estimate the effect of choosing various values of T_b and r.

The trade-off is between resource usage and availability. Decreasing T_b increases the number of REPLICATION OFFER messages the system needs to send (thus using extra bandwidth), but prolongs the time the system is expected to run before data is lost. Increasing r makes write operations slower and increases the disk space and bandwidth used, but similarly prolongs the expected time. The performance considerations are discussed in more detail in Section 3.4.

3.4 Performance characteristics

It is difficult to analyse the performance of the design in general since it depends on the particular network and the usage patterns. In spite of this, Section 3.4.2 tries to deduce some general properties of the scalability of the read and write operations when many nodes are active. Section 3.4.1 analyses the communication overhead in the design, and Section 3.4.3 discusses how well the design supports various file access patterns.

3.4.1 Communication overhead of the system

The system sends more data than just the user data when reading and writing files, and similarly sends extra information to ensure the r replicas are kept. We define the *overhead* o to be the ratio between the size of this extra data s_e and the size of the user data s_u, $o = s_e/s_u$.

For the following examples, we assume that the maximum size of a block is $B = 60,000$ bytes and the size of a key is $K = 20$ bytes, which is the case for SHA-1 hash keys. The size of a file identifier block is $I = 41$ bytes and the size of a file is denoted by $|f|$.

For reading a file f, the extra data sent is the file identifier and the RETRIEVE BLOCK messages (each containing one key) for requesting blocks:

$$o_r = \frac{(\lceil |f|/B \rceil + 1)\, K + I}{|f|}$$

The overhead when reading is shown in the second and third column of Table 3.1.

The overhead for writing a file is much larger because $r - 1$ extra copies of the file must be stored. Also, each STORE BLOCK message contains the key of the block in addition to the block itself, and r file identifiers must be sent. Hence, the overhead is:

$$o_w = \frac{(r-1)|f| + r(\lceil |f|/B \rceil + 1)K + rI}{|f|} \tag{3.5}$$

|f|	o_r	o_r	o_w	o_w	o_b	o_b
30 kb	0.081 kb	0.27%	90.3 kb	301.1%	0.16 kb	0.53%
2 Mb	0.74 kb	0.04%	6.002 Mb	300.1%	2.8 kb	0.14%
100 Mb	33.4 kb	0.03%	300.1 Mb	300.1%	133 kb	0.13%

Table 3.1: The overhead of reading, writing and for sending republication offers for files of various size. A network with $r = 4$ and $T_b = 1$ is used in the calculations. The overhead is both represented as the number of bytes and as the percentage of the file size.

The overhead of writing is shown in the fourth and fifth column of Table 3.1. Clearly, when r is varied the above expression is dominated by the first term so $o_w \approx ((r-1)|f|)/|f| = r - 1$.

Some extra data is also sent from time to time to republish the data. If it is not necessary to actually send any blocks, then the only data sent is the replication offers. At least one replication offer must be sent to each of the $r - 1$ other replicating nodes for each block. The offer itself is just the block key of size K and must be sent each T_b; hence the republishing overhead for a file is:

$$o_b = \frac{r(\lceil |f|/B \rceil + 1)K}{|f| \times T_b}$$

The overhead of republishing per hour is shown in the sixth and seventh column of Table 3.1.

Finally, some extra data is sent to maintain the routing tables. Every node updates its routing table every T_r hour and this is done, as described in Section 2.2.3, by splitting the nodes into groups and sending a lookup for each group. Since the nodes are split into $\log_2(n/r)$ groups, this will result in $\log_2(n/r)$ lookups. Each lookup message contains a key and each response contains r IP addresses and port numbers, i.e. 6 bytes per host. Hence, the extra data for maintaining the routing table are:

$$o_m = \frac{\log_2(n/r)(K + 6r)}{T_r}$$

The extra data used to maintain a routing table in a network with $n = 500$, $r = 4$ and $T_r = 1$ hour is 306.5 bytes per hour or 0.09 bytes per second.

The various communication overheads of our system are relatively inexpensive, except the overhead of writing which is proportional to r because the $r - 1$ replicas are overhead. The overhead for the periodic replication and routing table updates is small enough to let the system scale to large amounts of data and large networks. In the next section we examine if this is also the case for reading and writing.

3.4.2 Scalability of reading and writing

For a distributed shared file system, the network is likely to become the bottleneck. The load on the network increases as the number of simultaneous reading or writing nodes increases. This will lead to much increased response times at some point, and eventually packet loss. The point where the network becomes congested depends on how the shared file system is designed.

We define the *total throughput* in the network to be the total amount of user data read or written in all nodes within some period of time. The total throughput in a centralised file system such as NFS is bound to the bandwidth of the network link between the file server and the network. Theoretically it can scale linearly up to the bandwidth limit and no further.

Our system does not possess this limit and can for reading in principle scale up to the sum of the bandwidths from each node to the network. In the best case, when multiple nodes are reading they will not interfere with each other, and the system will not penalise them for being active at the same time. The situation is the same for writing, but since the overhead for a write operation is $r - 1$, the total throughput is degraded by a factor.

Hence, it is to be expected that Heurika can outperform a centralised file system when many nodes are active at the same time. Whether this is in true in practise will be explored in the tests in Chapter 5.

3.4.3 Access characteristics

The system does not differ between sequential and random access to files. When the file identifier has been retrieved, the key for any block in a file can be computed. Hence reading files can be supported in the same way as a local filesystem.

Unfortunately, the same is not true for writing files since the system does not support partial modification of files. If a single file data block is modified and stored in the network, all other blocks in the file must be rewritten since they get a new key because the file identifier has changed. This problem cannot be solved easily because the changed file identifier is necessary for ensuring that files do not end in an inconsistent state.

Log files is one prominent application that needs efficient support for partial modifications since log files are usually built by appending one line at a time.

Chapter 4

Implementation

This chapter describes the implementation of a prototype of our system. The purpose of the prototype was to enable us to explore how our system performs in various situations, as described in Chapter 5.

The distributed hash table in Section 2.2 and the file system in Section 2.3 have been implemented with 4000 lines of C++ code. The garbage collector described in Section 2.4 has not been implemented due to the lack of time.

Section 4.1 presents important concepts regarding the protocol, routing table issues and division of labour in the implementation of the distributed hash table. Section 4.2 gives an overview of the implementation of the file system.

4.1 Distributed hash table

The main concept of the implementation of the distributed hash table are *work items* which represent the major operations that the distributed hash table may perform. When an operation is requested, a work item is created and placed in the *ready queue*. A worker from the *thread pool* will then eventually get to the work item and process it. Work items that have been partly processed and need a reply from the network to continue are placed in a *waiting queue*. They are later moved to the ready queue if a reply comes in or a time out occurs.

The following sections give an overview of the classes and discuss selected topics in more detail.

4.1.1 Class overview

The classes of the distributed hash table are illustrated in Figure 4.1. The following descriptions will briefly explain each class; thread classes are denoted by the index $_T$.

DHT: Represents the external interface of the distributed hash table (the *put* and *get* methods) and glues the other major classes together by holding a

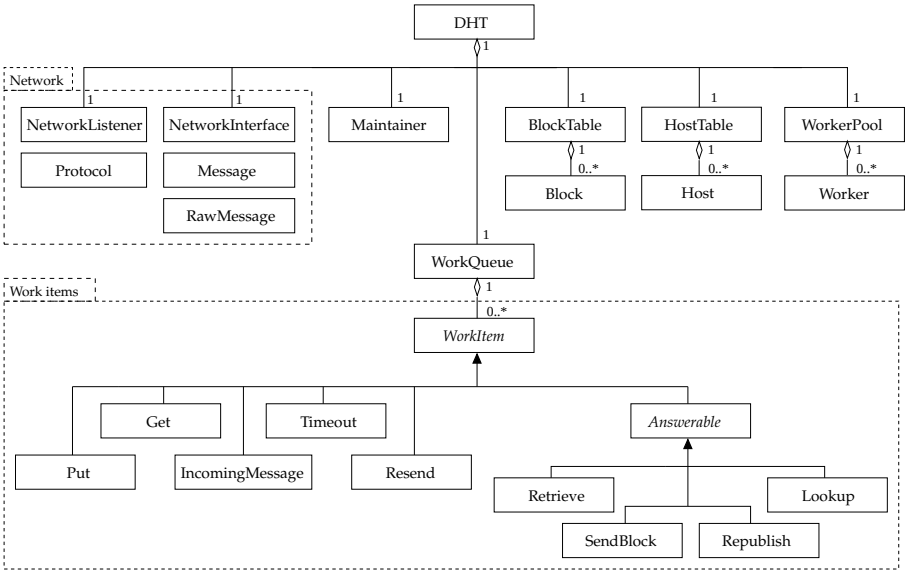

*Figure 4.1: The classes in the distributed hash table component. Note that the **WorkItem** suffix for the subclasses of **WorkItem** has been left out.*

WorkerPool, **WorkQueue**, **Maintainer**, **NetworkInterface**, **NetworkListener**, **BlockTable** and a **HostTable**.

WorkerPool: Represents a pool of idle workers that are ready to be assigned a task. Multiple workers are needed because some tasks are blocking (e.g. saving a block to disk). The number of workers is set as a fixed parameter in our prototype but should ideally be chosen dynamically.

Worker$_T$: Represents a worker that processes work items, by invoking the *do process* method on the various work items.

WorkQueue: Represents a ready and a waiting queue with methods for adding the appropriate work item and grabbing the next ready item. It also facilitates getting a waiting work item from a sequence number once a reply message is received.

WorkItem: The abstract base class for queued work items. It has the following subclasses:

PutWorkItem: Represents a put block request to the hash table that distributes the block to the r closest nodes (by using **LookupWorkItem** and **SendBlockWorkItem**).

GetWorkItem: Represents a get block request on the hash table. Uses **LookupWorkItem** and **RetrieveWorkItem**.

AnswerableWorkItem: Represents an operation that is awaiting a network message and needs to accept a message before being able to continue. **LookupWorkItem**, **SendBlockWorkItem** and **LookupWorkItem** are subclasses that are awaiting a reply.

LookupWorkItem Represents a look up operation for finding the r closest nodes.

SendBlockWorkItem: Sends a block to a single node for storage.

RetrieveWorkItem: Retrieves a block from a single node.

IncomingMessageWorkItem: Represents an incoming message from another host. The message may be a reply in which case the appropriate work item is taken out of the waiting queue, or it may be a request.

RepublishWorkItem: Represents the republishing operation for sending out replication offers to $r - 1$ other nodes.

ResendWorkItem: For resending a message when a work item in the waiting queue has had a timeout.

TimeoutWorkItem: Represents a fatal timeout after too many consecutive failed resend operations. The work item that experiences the timeout is taken out of the waiting queue and should continue under the assumption that the recipient node is unreachable.

Each subclass implements the abstract *do_process* method of **WorkItem**.

Maintainer$_T$**:** Represents a maintainer that assures that work items in the waiting queue do not wait forever and that the block table and the host table are kept up-to-date periodically. It creates **ResendWorkItem**s and **TimeoutWorkItem**s for items that have been too long in the waiting queue, and creates **RepublishWorkItem**s and **LookupWorkItem**s for the maintenance of the block table and host table. The periods between maintaining the block table and host table are randomised to avoid that all nodes perform the maintainence at the same time.

Message: Represents the base class for the protocol primitives described in Section 2.2.5. Each protocol message, both request and reply, has a corresponding class that contains the data structure of the protocol primitives, i.e. the data that is transmitted and received over the network. For instance, RETRIEVE BLOCK corresponds to **RetrieveBlockMessage**. Note that the subclasses of **Message** are not shown in the class diagram (to avoid cluttering it).

RawMessage: Represents a byte sequence of a **Message** that is ready for transmission.

Protocol: Represents the protocol specifications, see Section 2.2.5, that converts the various messages to a **RawMessage**. It also supports converting a **RawMessage** back to its corresponding **Message**.

NetworkInterface: Represents the network interface for host-to-host communication. It has a method for sending and a method for receiving raw messages.

NetworkListener$_T$: Listens on the network for incoming messages and creates a **IncomingMessageWorkItem** on the ready queue for each received message.

BlockTable: Holds the blocks of a node in a hash table keyed on the block key and facilitates getting and saving blocks.

Block: Represents a block.

Key: Represents an m-bit hash key. It supports setting and getting the n'th bit (used for the host table maintenance) and calculating the XOR distance between two keys.

HostTable: Holds the list of known hosts with methods for getting the nodes between two keys (used for the host table maintenance) and looking up nodes closest to a key.

Host: Represents the contact information for a host with an IP address and a port.

The following sections dive into the key aspects of the distributed hash tables and present the associated code snippets.

4.1.2 Work items and queues

Work items are at the heart of the system since all requests in the distributed hash table are processed through them. As shown in the class diagram, Figure 4.1, all work items derive from the abstract class **WorkItem** that provides a common interface consisting only of the method *process* (which in turn calls *do_process*).

The work items are processed by the worker threads. All work items, when not being processed, are situated in one of the queues in **WorkQueue**. When work items are first created they are queued in the ready queue, waiting for a worker to grab it. The workers ask the ready queue for work to do by calling the *grab_ready_item* method on **WorkQueue**. The method *grab_ready_item* is shown in Listing 4.1 and explained below.

Listing 4.1: ***WorkQueue****'s grab_ready_item*

```
1 WorkItemPtr WorkQueue::grab_ready_item()
2 {
```

```
3     ready_items.wait();
4
5     ready_lock.enterMutex();
6     WorkItemPtr item = ready_queue.front();
7     ready_queue.pop_front();
8     ready_lock.leaveMutex();
9
10    return item;
11  }
```

If the ready queue is empty, the caller will wait on a semaphore (Line 3) until an item is added to the queue and the semaphore is incremented. When no items are present in the queue, all unemployed workers will be blocked until more work arrives.

When there is work to do, the calling worker decrements semaphore and grab an item from the queue. Since the distributed hash table is multi-threaded the shared data structures must be protected by mutexes. The mutex *ready_lock* protects the critical region in Line 5-8. When the work item has been extracted from the queue, it is returned to the caller.

The two main work items are **PutWorkItem** and **GetWorkItem**. Both need to perform subtasks, e.g. initiate a lookup. This is done by creating the appropriate work item, e.g. a **LookupWorkItem**, setting it up to pass back control when it is done by registering two callback functions, one for success and one for failure, and then calling its *process* method. A typical example is shown in Listing 4.2 (slightly edited for readability).

*Listing 4.2: **PutWorkItem**'s do_process*

```
1   void PutWorkItem::do_process()
2   {
3     // first do a lookup
4     shared_ptr<LookupWorkItem> lookup(new LookupWorkItem(key));
5     lookup->add_dependent(shared_from_this());
6     lookup->on_completed = bind(&on_lookup_completed, this);
7     lookup->on_failed = bind(&on_lookup_failed, this);
8     lookup->process();
9   }
```

The **LookupWorkItem** first asks the host table for the *r* closest hosts which are then sent a lookup message through the network interface and afterwards awaits a reply. Since this may take some time and thus block the worker if it were waiting for it, the work item is transfered to the waiting queue and the associated worker will then immediately be ready for a new task.

Because of this arrangement, the lookup work item derives from the abstract subclass of **WorkItem** called **AnswerableWorkItem**. Classes that derive from **AnswerableWorkItem** must implement the two methods *do_accept* and *timeout*. *do_accept* will be called when a reply is received, and *timeout* is called when it has been detected that it was not possible to get through to the intended receiver, i.e. the outgoing message has been resent the maximum allowed number of times without an answer.

When the work item has finished its task, it passes its results back to the creator through the provided callback functions.

4.1.3 Organising hosts

When sending and retrieving blocks the host table is consulted to find the hosts that are closest to the block key. The method *get_closest* in **HostTable** takes as input the block key and the number of wanted nodes to return (usually simply *r*).

Listing 4.3 presents the part of the method that iteratively divides the hash space into smaller subranges to find a range that contains just enough hosts to satisfy the request.

*Listing 4.3: **HostTable**'s get_closest*

```
1  Key host_start, host_end;
2  for (int i = 0; i < Key::key_size; ++i) {
3    host_start.value[i] = 0;
4    host_end.value[i] = 255;
5  }
6
7  int n = Key::key_size*8-1;
8
9  for (; n >= 0; --n) {
10   // half the search space
11   bool nth_bit = block_key.get_nth_bit(n);
12   host_start.set_nth_bit(n, nth_bit);
13   host_end.set_nth_bit(n, nth_bit);
14
15   int size = nodes_in_range(host_start, host_end, wanted_nodes);
16
17   if (size <= wanted_nodes)
18     break;
19 }
```

The keys *host_start* and *host_end*, defined at Line 1, define the bounds of the current interval and are first set to all 0s and all 1s, Line 2-5. The loop index *n* starts at the most significant end of the keys (number of bits in a key minus 1), at Line 7. Then the loop in Lines 9-19 simply consider one bit of the block key at a time, each time cutting the current interval in half. At some point, the current interval will contain fewer than the wanted number of nodes and the loop stops, Line 17.

The rest of the method (not shown) takes one step back, sorts the found hosts by their XOR distance to the block key, removes the most distant nodes from consideration until there is at most the wanted number of nodes back, and finally returns the result.

4.1.4 Composition and decomposition of messages

Making a protocol message ready for transmission over the network involves composing a **RawMessage** of the message. This is handled by using the *compose* functions in **Protocol** that takes a message as input and returns a **RawMes-**

sage. A **RawMessage** is converted back into its equivalent message by using the *decompose* function, see Listing 4.4.

*Listing 4.4: The interface of **Protocol***

```
1  namespace Protocol
2  {
3    RawMessagePtr compose(const AckMessage &m);
4    RawMessagePtr compose(const NegAckMessage &m);
5    RawMessagePtr compose(const FindClosestMessage &m);
6    RawMessagePtr compose(const FindClosestReplyMessage &m);
7    RawMessagePtr compose(const RetrieveBlockMessage &m);
8    RawMessagePtr compose(const RetrieveReplyMessage &m);
9    RawMessagePtr compose(const PingMessage &m);
10   RawMessagePtr compose(const ReplicationOfferMessage &m);
11   RawMessagePtr compose(const StoreBlockMessage &m);
12
13   boost::shared_ptr<Message> decompose(const RawMessage &composed);
14
15   ...
16 }
```

A **RawMessage** consist of a byte sequence representation of a **Message**. Listing 4.5 shows one example of how this is done.

*Listing 4.5: compose function for **RetrieveBlockMessage** that uses fill_header*

```
1  void fill_header(RawMessage &raw, const Message &m,
2                   MessageType m_type)
3  {
4    raw.buffer[0] = m_type;
5
6    *reinterpret_cast<uint32_t *>(&raw.buffer[0] + 1) = htonl(m.seqno);
7    raw.host = m.host;
8  }
9
10 RawMessagePtr Protocol::compose(const RetrieveBlockMessage &m)
11 {
12   int size = header_size + Key::key_size;
13
14   RawMessagePtr raw(new RawMessage(size));
15
16   fill_header(*raw.get(), m, retrieve_block_type);
17
18   m.key.to_raw(reinterpret_cast<unsigned char*>(raw->buffer+header_size));
19
20   return raw;
21 }
```

In Line 12 the size of the byte sequence is calculated. Next step is to fill in the data. First the header is filled in, at Line 16, by calling *fill_header* which simply sets the first byte to the type of the message and fills in the message sequence number. Platform byte ordering is taken into account, Line 6, by converting the sequence number into the network byte ordering before copying it into the byte sequence.

When the header has been filled in, the rest of the message is copied into the byte sequence. For **RetrieveBlockMessage** this is simply done by copying the contents of the key to be received.

4.2 File system

The file system component lets an application work with the high-level abstraction of a file by taking care of the lower level calls to the distributed hash table and the organisation of blocks.

As with an ordinary UNIX file system, all calls to the file system are blocking, meaning the the caller will be blocked until the request has been fully processed. Given the architecture of the distributed hash table component, it is actually very easy to also provide an asynchronous interface if one is needed.

We proceed with a brief description of the prototype of the file system. It consists of one class only:

NetFile: Represents a network file and providing methods for opening, reading, writing, seeking, deleting and closing a file.

The constructor opens the file by requesting the file identifier block from the distributed hash table. If the file has been initialised in reading state and the identifier block is not found, an exception is thrown. On the other hand, if the file is opened for writing the file will be created if no identifier block is found.

The read operation is implemented by retrieving all blocks from the current position in the file and up to the specified number of bytes, in parallel for better performance. The relevant bytes can then be extracted.

The write operation is implemented by first retrieving all blocks of the file before the actual data are written to it. The blocks are then distributed under a new key, also in parallel. The new file identifier block is not written until the file is closed.

Seeking is achieved by shifting the current position in the file by a given offset, which may be defined as relative in both directions or static. Deleting is done simply by retrieving the directory file where the file is situated, removing its entry in it and writing the directory again, letting the garbage collector take care of the rest.

Closing the file consists of writing the file identifier block. The following code snippet from the *close* method illustrates how the file system communicates with the distributed hash table and how the file system ensures that its operations are blocking.

Listing 4.6: ***NetFile****'s close and on_close_file*

```
1  void NetFile::close()
2  {
3    if (state == writing) {
4      DHT::instance().put(ident_block_key, compose_ident(ident_block),
5                          bind(&NetFile::on_close_file, this, _1));
6      block.wait();
7
8      if (error_occurred) {
9        error_occurred = false;
10       throw CloseError();
```

```
11      }
12    }
13  }
14
15  void NetFile::on_close_file(bool complete)
16  {
17    if (!complete)
18      error_occurred = true;
19
20    block.post();
21  }
```

On Line 4 the identifier block is passed to the distributed hash table. Besides the key of the identifier block and the block itself, a callback function, *on_close_file*, is given to the *put* method. The callback is used when the distributed hash table has completed the task and needs to notify the file system of this. The *put* method will return immediately after having placed a work item in the work queue, and the file system therefore waits on the semaphore *block*, Line 6.

When the file indentifier has been distributed, *on_close_file* is called by a thread from the distributed hash table. A check for errors is performed and then the *block* semaphore is signalled at Line 20. This means that *close* can continue execution. It does so by checking if an error occurred, Line 8, throwing an exception if that is the case, or else just returns.

Chapter 5

Tests

This chapter describes an empirical evaluation of the prototype described in Chapter 4. The purpose of testing our prototype is to gain sufficient evidence that our design fulfills the requirements in Section 1.4, and further to show that some of the theoretical characteristics in Chapter 3 hold in practise.

Section 5.1 describes the environment used to simulate the network, Section 5.2 focus on how efficient the use of replication is in keeping all blocks available, and finally Section 5.3 examines the performance of the system. Fulfillment of the design requirements is discussed in the conclusion of the report in Chapter 6.

The implementation has been tested for correctness through programmed test cases. The test cases cover the algorithms necessary for conducting the experiments described in the following.

5.1 Test environment

The tests have been conducted in a simulated network environment maintained by the `dummynet` tool [13]. `dummynet` is included in the `ipfw` firewall on FreeBSD and supports sending messages through special pipes with associated queues and restrictions on bandwidth. Combined with aliasing of IP addresses to the local host, this enables one to setup a complete virtual network on a single machine with bandwidth restrictions, switches and realistic packet delay and loss when the network becomes congested. The programs under test must simply bind themselves to certain IP addresses.

Unfortunately, `dummynet` does not come with any tools to setup a network so instead we have written a small program that outputs the rules for a local area network connected with switches. The virtual network used for the tests described in the following is shown in Figure 5.1.

The benefit of using `dummynet` is that it has very minimal overhead compared to virtual computers such as VMWare [19], allowing us to run larger

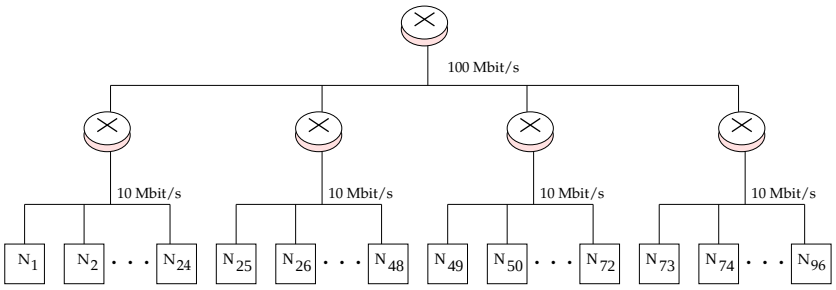

Figure 5.1: The virtual network consists of four subnetworks. Each subnetwork contains 24 nodes and each node is directly connected to a 10 Mbit/s switch. The four subnetworks are connected to each other with 100 Mbit/s links to a backbone switch.

tests on the same hardware. Furthermore, the program does not have to be specifically written for a simulation environment as required in other virtual networks such as ns-2 [10].

Since our tests are run in the simulated environment shown in Figure 5.1, the following should be kept in mind:

1. There is no other traffic on the network.
2. We test only one size and one particular simple type of network, namely the star topology shown in Figure 5.1.
3. A large number of instances of the test programs are running on a single machine – there is one process for each node so 96 instances in total. This means that during periods with peak load, e.g. in the performance test, many processes are competing for processor time.

The third problem was actually the reason that we let the test system use 10 and 100 Mbit/s links instead of 100 and 1000 Mbit/s links. The test hardware (an AMD Athlon XP 2700+) was not fast enough to support the higher bandwidths.

5.2 Robustness of the system during node crashes

This test measures how many replicas are available over time when nodes are killed with a constant rate.

The purpose of the test is to evaluate the robustness of the use of replication in ensuring that blocks are not lost. Ideally, we would have preferred to verify the data loss probability predicted in Section 3.3, but unfortunately the predicted probabilities are so small that they are difficult to test with the `dummynet` framework which runs in real time.

Instead we stress the system by taking nodes down at rate greater than the republishing interval T_b. This enables us to see how well the replicas ensure availability and how well the republishing algorithm performs.

5.2.1 Test setup

The 96 nodes are started with $r = 3$ and after the network has been setup, each writes a 550 kbyte file. This ensures that there are approximately 33 blocks per node. $T_b = 15$ minutes instead of one hour to speed up the test. After a short pause to ensure that all write operations are finished, a random node is killed every T minutes until all nodes except one have been killed.

We repeated the test five times with $T = 5$ minutes and five times with $T = 1$ minute. This means that three and fifteen nodes are killed per republish interval, respectively.

5.2.2 Test results

Figure 5.2 and 5.3 show the fraction of lost blocks as a function of the number of nodes killed, with 5 and 1 minute between nodes were killed, respectively.

To get a better picture of what is happening in the system during the tests, Figure 5.4 and 5.5 show the fraction of blocks that have 3, 2, 1 and 0 replicas during a typical test run with 5 and 1 minute between nodes were killed, respectively.

5.2.3 Evaluation of the results

The results show that the use of replication is a very powerful technique. With $r = 3$, one would perhaps expect that blocks would be lost quickly when three nodes are killed per republishing interval. Figure 5.2 illustrates that this is not the case, and Figure 5.4 shows that the republishing algorithm keeps up pace with a good margin until the network has been reduced to about $96 - 80 = 16$ nodes.

Even when fifteen nodes are killed per republishing interval, as shown in Figure 5.3, the system keeps 90% of the blocks until the network is reduced to $96 - 70 = 26$ nodes.

An explanation for these surprising results can be found in the theoretic analysis in Section 3.3.2. The probability P_s of actually losing data in a network of size $n = 96$ given that $r = 3$ nodes have failed is less than

$$P_s = \frac{(3-1)!(96-3)!}{(3-3)!(96-1)!} = \frac{2}{95 \times 94} = 0.0226\%.$$

5.3 Maximum read and write throughput

This test measures the total throughput of reading and writing with different values of r and different numbers of nodes that read and write simultaneously.

The motivation for the test is to explore how the system scales when multiple nodes are reading and writing at the same time, and to see if the write overhead predicted in Section 3.4 to be $o_w \approx r - 1$ holds in practise. As described

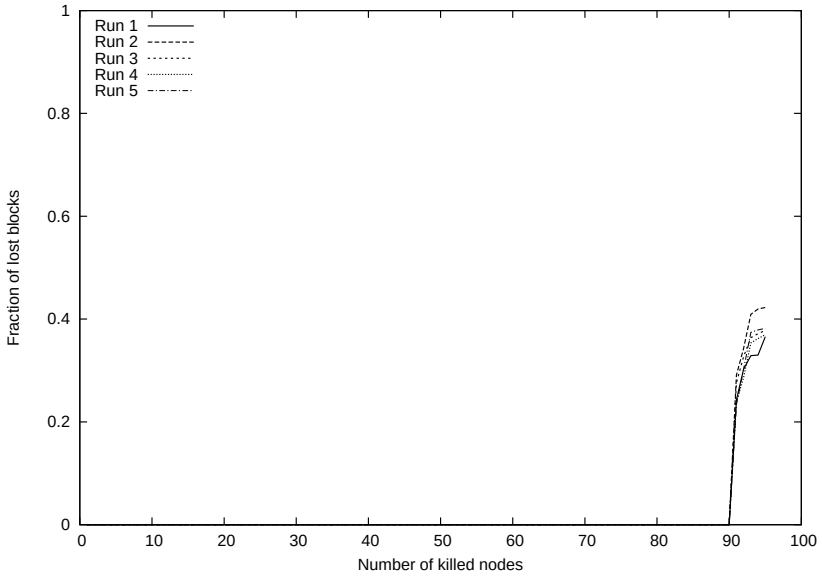

Figure 5.2: 96 nodes are started and the test kills one node every fifth minute. No blocks are lost until 90 nodes have been killed.

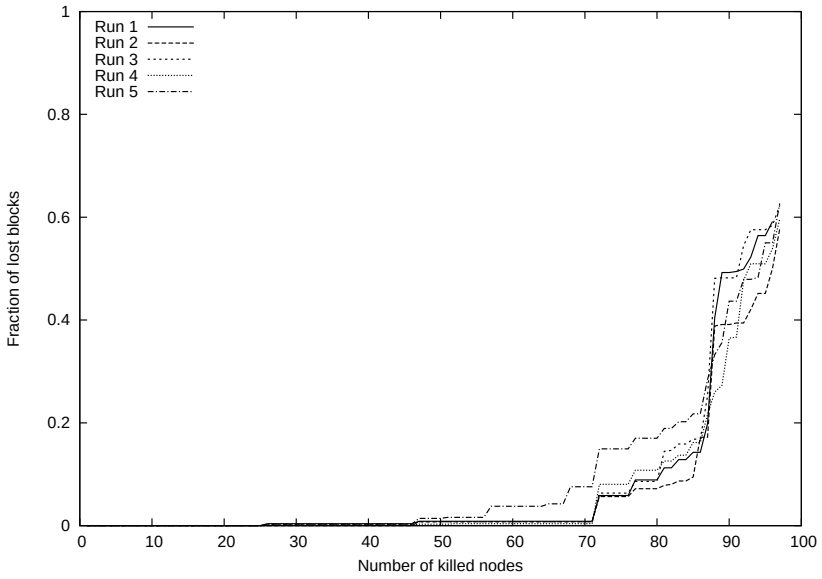

Figure 5.3: 96 nodes are started and the test kills a node every minute. No blocks are lost until between 25 and 45 nodes have been killed. When 70 nodes have been killed, the number of lost blocks increases dramatically.

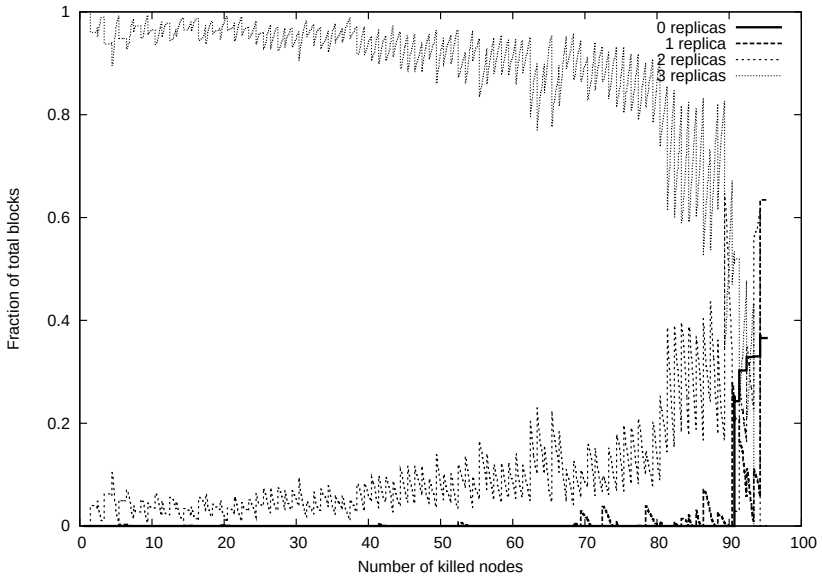

Figure 5.4: A closer view of a test run where a node is killed every fifth minute. The figure shows the fraction of blocks that have 3, 2, 1 and 0 replicas. 90% of the blocks have 3 replicas until about 50 nodes have been killed.

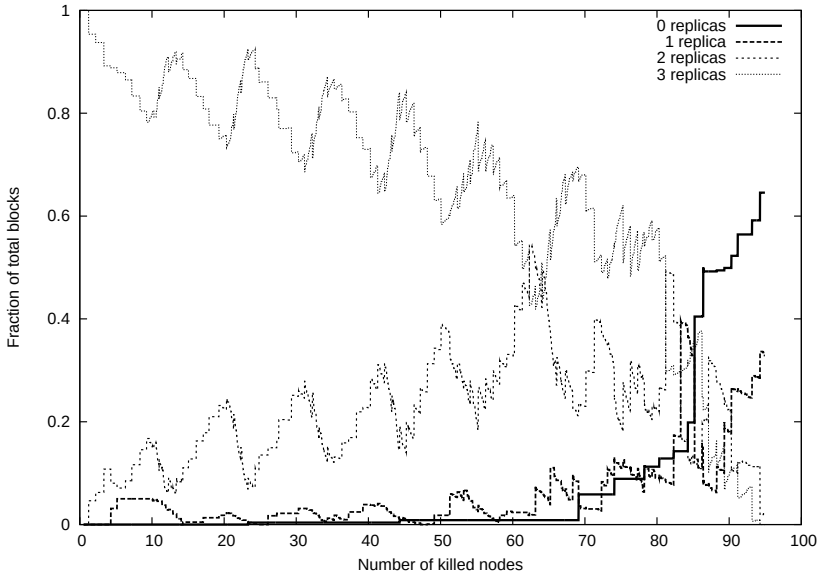

Figure 5.5: A closer view of a test run where a node is killed every minute. The number of replicas is much more unstable compared to Figure 5.4. Notice the oscillations; their period is about 12-15 minutes. Compare this with $T_b = 15$ minutes. The imprecision can be attributed to the fact that T_b is randomised a little to avoid having all nodes republish at the same time.

in Section 3.4.2, one may expect the total throughput of the system to be able to scale beyond what a centralised shared file system can deliver because all communication does not have to go through the backbone The maximum total throughput for a centralised shared file system for the 100 Mbit/s backbone used in this test is about 10-11 Mbyte/s.

5.3.1 Test setup

The 96 nodes are started at the same time with n nodes as active nodes and the other $96 - n$ nodes as passive nodes. The passive nodes simply participates in the network without reading or writing anything, while the active nodes run the following program once:

1. Wait until a predefined point in time common to all active nodes to ensure that the network has been setup.
2. Start a timer and start writing a 2 Mbyte file.
3. When the writing is finished, record the value of the timer and log the throughput (2 Mbyte divided by the recorded value). Then wait until another predefined point in time to ensure that all n nodes have finished writing.
4. Start a timer and read one of the 2 Mbyte files that have just been written but not the file that the node has written itself and not a file that one of the other nodes is reading.
5. When the reading is finished, record the value of the timer and log the throughput (again 2 Mbyte divided by the recorded value). Then wait a little longer to ensure that all nodes have finished reading.

Note that the active nodes start writing at the same time and start reading at the same time, and that the two activities do not overlap. Since the tests are short, routing table maintenance and republishing do not interfere with the results. During the tests, the blocks are never actually saved to the hard disk but simply kept in memory to avoid the single disk on the test machine becoming a bottleneck.

The test is repeated by killing all Heurika processes and starting them over again to ensure that the data from a previous test is not still being stored by the passive nodes.

5.3.2 Test results

The results of the tests were three test run sets of throughput values from each node for read and for write with the replication constant $r = 1, 2, \ldots, 6$ and with the number of active nodes $n = 1, 2, \ldots, 45$ (i.e. $3 \times 2 \times 6 \times 45$ sets of values).

Appendix A contains plots of the throughputs. Each figure in the appendix shows the throughput of every node in each test run as a function of n. The average throughput is shown as a bold line.

The average of the *total* throughputs for the three test runs is shown for read in Figure 5.6 and for write in Figure 5.7, both with total throughput as a function of n.

5.3.3 Evaluation of the results

Figure 5.6 and 5.7 show that the system does indeed scale well, in fact close to linearly in n when reading! The throughputs for reading do start growing slower at about 35 active nodes. This could be because the network becomes congested, but it could unfortunately also be because our test machine was too slow to keep pace. Anyway, the system is clearly able to handle more than $1/3$ of the nodes being active at the same time without penalising their throughput.

One interesting aspect of the read throughput is that higher values of r seems to give slightly better performance as shown in Figure 5.6. We attribute this to the fact that a reading node will have more nodes to choose from when retrieving a block with higher values of r. This helps balancing the load.

Even though the total write throughputs also scale well, albeit apparently sub-linearly in n as shown in Figure 5.7, the test does highlight that the write speed is slowed down much when the data are replicated. Since setting $r = 1$ or $r = 2$ is dangerous from an availability and data loss perspective unless the nodes are very stable, this slowdown is a necessary consequence of the design. Of course, a real implementation can hide the problem from the user by distributing the blocks in the background. We further discuss optimising some of the replicas away for short-lived files as a suggestion for future work in Section 7.5.

The graphs in Appendix A show that the individual read and write throughputs are quite close to the averages so that the graphs in Figure 5.6 and 5.7 are meaningful (at least for $n \leq 35$).

One thing that must be kept in mind is that the tests used a constant and relatively large file size. Hence, the results do not reflect how the system would handle a realistic usage pattern with a mixture of small and large files. The throughput is expected to be lower when processing many small files since more file identifiers must be retrieved. On the other hand, the file identifiers are quite small so more nodes can be expected to be active at the same time without disturbing each other.

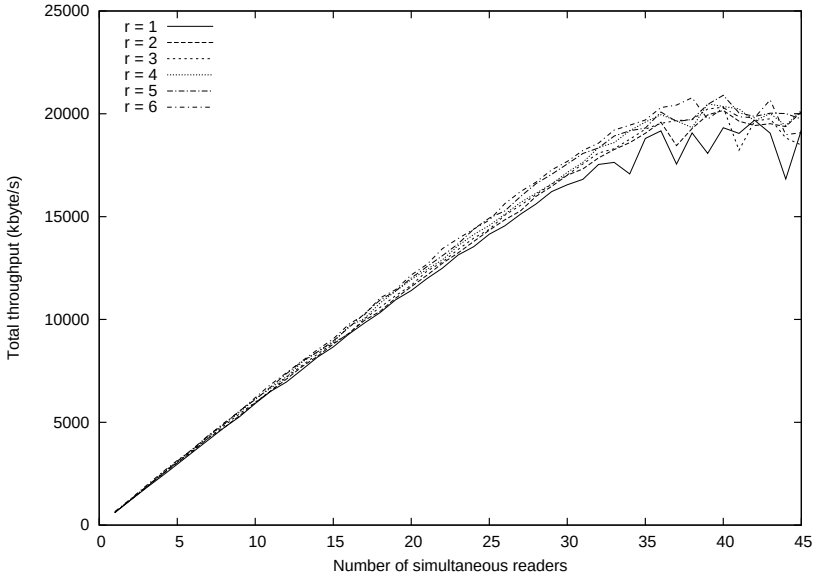

Figure 5.6: Total read throughput averaged over the three test runs. The throughputs scale linearly up to about 35 active nodes. Note that the throughput seems to be slightly better for higher values of r.

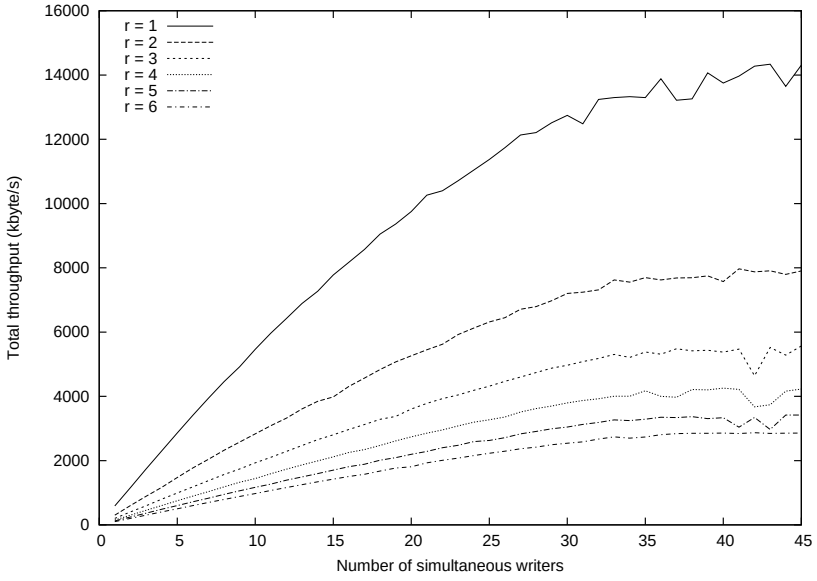

Figure 5.7: Total write throughput averaged over the three test runs. The throughputs scale sub-linearly. Higher values of r result in lower throughput.

Chapter 6

Conclusion

This chapter presents the conclusions of the project. We start by summarising the findings of the project in Section 6.1 and conclude in Section 6.2 by examines the achievements of the project in comparison with the project aims as listed in Section 1.4.

6.1 Summary

In Chapter 1 we reasoned about why there may exist a need for another approach for a shared file system than what is traditionally deployed in an organisation. We examined existing centralised solutions that, we believe, are inadequate in terms of ensuring availability when the overall cost of the solution is kept in mind. We concluded that a peer-to-peer decentralised file system designed for high availability and fault-tolerance could be a viable alternative.

Chapter 2 presented the design of our system, with three components: the distributed hash table, file system and the garbage collector. The distributed hash table uses the XOR metric to replicate blocks among nodes to ensure high block availability. The file system component organises the blocks of the distributed hash table into files and directories that user applications can manipulate. The garbage collector makes sure that leftover blocks are cleaned up from time to time.

In Chapter 3 we characterised our design by theoretical analyses, and concluded that:

Availability of blocks Symmetric network splits severely hamper the availability of large files requested from any node, while asymmetric network splits mostly affect the nodes in the smallest subnetwork.

File consistency guarantees It is possible to guarantee that retrieved files are consistent, although it is not possible to guarantee that all nodes will observe the same version of a file if there are multiple versions.

Loss of data Data loss is very unlikely with appropriate values of r and T_b. Decreasing T_b prolongs the time the system is expected to run before data is lost. Increasing r similarly prolongs the expected time.

Performance characteristics The overhead of writing a file is proportional to r because the $r - 1$ replicas are overhead and the other overheads in the system are relatively inexpensive.

The system is expected to scale better than a centralised file system, because reading in principle scale up to the sum of the bandwidths of the links between nodes; writing is degraded by a factor because of the replication overhead.

Random file access is supported efficiently, but small file changes are expected to be very inefficient.

Chapter 4 described the implementation of a prototype that consists of the distributed hash table and the file system. The purpose of the prototype was to enable us to explore how our system performs in various situations. The implementation turned out to be quite simple for a distributed file system and only required 4000 lines of C++ code.

Two types of tests were setup, in Chapter 5, to empirically evaluate the prototype in a star topology network. The first test examined the robustness of the use of replication by killing nodes at a large rate, and showed that the system can largely keep up even under extreme conditions. The second test examined maximum read and write throughput and showed that the system scales linearly when reading and slightly sub-linearly when writing; the write speed was also found to be degraded by a factor as the number of replicas was varied, as predicted in Chapter 3.

We conclude by following up on the introduction in Chapter 1 with a discussion of whether the project aims described in Section 1.4 have been fulfilled.

6.2 Fulfillment of the project aims

The project aims can be summarised as to:

- Design a decentralised peer-to-peer file system.
- Implement a working prototype.
- Analyse the design theoretically and in practise through the prototype.

We believe these have been fulfilled overall. The requirements for the design were:

High availability and fault-tolerance This has been achieved through the use of replicas that are periodically kept up to date to compensate for crashed

nodes. The theoretical analysis shows that this technique should be very efficient under normal conditions. The robustness test shows that the system can withstand even worse conditions.

Hence, we conclude that the design makes for a highly available system capable of tolerating node failures.

Self-organising and dynamic A hash function is used to map hosts and data into a logical hash space independent of any physical layout, and hosts may leave (or crash) and join this space as they wish. The procedure for joining and the periodic updates of replicas and routing tables ensure that this is taken care of automatically.

Hence, we conclude that the design requires minimal human intervention and that it is thus self-organising and dynamic.

Reasonable performance The theoretical considerations showed that the overhead for reading and the periodic updates were minimal under normal operation. The most serious overhead is for write operations, but it seems likely that it would be possible to mostly hide this by distributing the blocks in the background. The use of replication does increase the disk space usage quite dramatically because each file must be stored r times. But on the other hand, the design makes it possible to exploit the free disk space on all nodes

Hence, we conclude that the performance of the design is reasonable, although it in its current form is unsuited for some purposes due to the write overhead.

Transparency Although transparency is not actually implemented by the prototype as a UNIX mountable device or as a Windows drive service, the interface of the file system makes this achievable without modifying Heurika. Of course, the performance is vastly different in some situations – for example, appending to a log file is an $\Theta(n^2)$ operation in the number of lines.

Hence, we conclude that the design is implementable in a way that makes it transparent for the users, with the exception of some more specialised usage patterns.

Scalability The tests showed that the prototype scales very well with respect to read and write performance and much better than what a centralised file system is capable of. Furthermore, the analysis of the overhead for the periodic replication and routing table updates is small enough to let the system scale to large amounts of data and large networks.

Hence, we conclude that the design scales well for its intended purpose.

Of course, much is still unknown regarding the behaviour of the system in different situations. The garbage collector was not implemented and we still

do not know how it interacts with the rest of the system. Further analyses and tests would be necessary to fully understand the implications of the design in a real-world setting.

There is also still plenty of room left for improvements in the design; we address some of the most obvious issues as suggestions for future work in Chapter 7.

Chapter 7

Future work

Although the design of Heurika works, as demonstrated by the prototype, it is far from perfect. This chapter presents current problems with the system and discusses how they can be solved.

The following problems are investigated:

1. There is no support for access control. Section 7.1 describes the problems that arise when file permissions are introduced and suggests a solution.
2. It is not possible to lock files to gain exclusive access. Section 7.2 discusses how Heurika can take advantage of having locks.
3. The topology of the physical network is not taken into account, and locality principles regarding who is using which files are not used to decide where to save a block. Section 7.3 suggests an approach for using the response time to decide which node to request a block from, and Section 7.4 describes how a reasonable locality assumption can be used to achieve better performance.
4. The overhead for writing is high. One way to reduce it is the observation that it is a waste of work to replicate a block to r nodes if the blocks are modified shortly after. Section 7.5 describes such an optimisation.

7.1 File permissions

Ordinary file systems support the notion of file permissions as a means of protecting access to files. This feature is very important for some applications but it is non-trivial to implement for a decentralised distributed system. We proceed with a suggestion for a solution based on the assumption that the other nodes in the system cannot be trusted.

The file permissions must assure that:

- Only a closed group of users, the group of readers, are allowed to read a given file or directory.
- Only a closed group of users, the group of writers, are allowed to write to a given file or directory.

If the other nodes in the system cannot be trusted, the files must be encrypted so that only the group of readers can read the contents. Otherwise, the nodes that replicate the file would be able to read the parts they are replicating. The simplest solution for encryption is to give each group of readers a unique secret symmetric key K_R which is used to encrypt all files that should be readable by that group. Then everyone in the group can decrypt the file.

The problem of write protection is more difficult because every node must be able to verify that a block has not been modified by someone without write access before accepting the block.

One solution is to give each group of writers a private-public key pair and add a signature to each block [2]. When the block is written, it is first encrypted with the secret read key K_R. Then a digest of the block is generated and encrypted with the private key K_W^- for the group. The encrypted digest is attached to the block and sent to the nodes that are responsible for replicating the block. The receiving nodes can then decrypt the digest with the public key K_W^+ and check that it corresponds to the block contents before accepting to store the block. The check must also be performed when blocks are republished.

This procedure is shown in Figure 7.1 and 7.2.

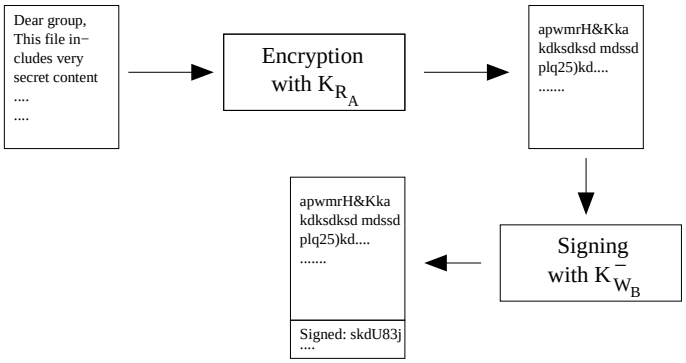

Figure 7.1: A block is first encrypted with symmetric encryption to assure that only group A has read access. Afterwards the encrypted block is signed with asymmetric encryption to ensure that only group B has write access (group B will usually be a subgroup of group A).

The described solution raises two immediate problems:

- How are the keys managed and accessed? A central key distribution service can be deployed [8], but at the risk of becoming a single point of failure and a bottleneck.

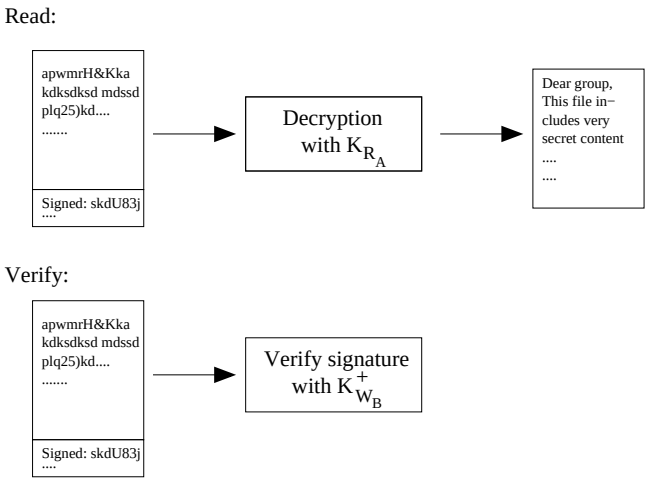

Figure 7.2: When a block is to be read, it is first decrypted with the symmetric read key as shown at the top. As shown below, all nodes that receive a block should verify the signature using the public write key from the group that has write access to the block. If the signature is invalid, the block may come from an intruder and should be rejected.

- How does a node find out who has write access to a particular file? Perhaps the directories in Heurika could be extended to support storing this information.

Future work is needed to resolve these problems and investigate how the implementation of file permissions affects the robustness and the performance of the design.

7.2 Locking files

The lack of locks means that nodes that wish to write to a file at same time cannot prevent each other from overwriting their changes. The situation is not as bad as for an ordinary file system where the different versions may be interleaved, and due to the way file updating is designed in Heurika, read locks are superfluous since a file can be read and written concurrently without problems.

Still, it would be useful if some sort of mutual exclusion for write operations was supported. Fortunately, the field of distributed mutual exclusion is well-studied [2], and the task is mostly to pick an algorithm that fits well with the architecture of our system. Moreover, locks can be used to avoid the problem that appending to a file is very inefficient.

The use of locks could also be exploited to make the file update algorithm behave differently in some circumstances. If a process holds a lock on a file, it is perhaps not necessary to enforce the strict file update algorithm we have

designed to ensure files are consistent.

Future work is needed to integrate lock support into the design and evaluate the result.

7.3 Exploiting response times for nodes

Whenever a node needs a block, it has r different nodes it could request the block from. Our prototype simply selects one randomly. This way, the load on popular blocks is divided evenly between the replicating nodes.

This seems reasonable for small networks where the response time between each node is more or less the same, but in a more complex network with a backbone, it is probably a better idea to choose the closest node in terms of network hops. Unfortunately, on a local area network with switches that do not touch the IP layer of the packets it is difficult to count the number of hops.

Instead, each node can monitor the response times from other nodes and choose the node with the lowest response time. This would also help relieve overloaded nodes.

The question is whether the conditions in the network change too quickly for the collected response times to be useful. As future work it would be interesting to implement response time monitoring and experiment with its usefulness.

7.4 Exploiting data locality principles

The distributed hash table in our system has the property that it distributes the block evenly among the nodes. This has the advantage of balancing the load and the data. But it also has the disadvantage of not taking data locality principles into account.

For example, the users working on nodes in a physical subnetwork are likely to be placed in the same department and thus need the same data. At the very least the user on a single node is likely to use the files he has created himself more than the other files in the shared file system.

Hence, it would make sense to store at least some block replicas in the same subnetwork. This would make it possible to access the blocks very efficiently without going through the backbone, and would also help make the most important files accessible if a network split occurs.

A simple solution is to place one of the replicas on the node that has created a given file. The hash key for the creating node could be stored in the file identifier so that other nodes would know where to look.

Future work is needed to understand what the consequences would be of such a modification of the design. It is conceivable that the data locality principles can be exploited in other ways, too. One obvious idea is to introduce

caching, although we have not discussed it because it interferes with what the operating system provides.

7.5 Life time of files

The prototype is designed as a blocking operation so that an application must wait until all blocks have been sent out in r replicas when a file is being written. But the life time of different files vary much and there is evidence that a considerable proportion of the files live only for a very short time before they are overwritten again [20].

For a short-lived file it is clearly wasteful to distribute it in r replicas that immediately become garbage. In other words, files that are going to be stored for a long period of time needs a stronger life time guarantee than files that are updated within a short period of time.

Hence, it might be beneficial to introduce the constant $r_{\min}$ and a time interval T_l. When a file is written, a replica of each block is sent to $r_{\min}$ different nodes with a special flag set. The nodes notice the flag, and after T_l time the republishing procedure is started and the block will end up being replicated on r nodes.

As future work, it would be interesting to see what implications this optimisation has on the performance.

Bibliography

[1] Tony Bourke. *Server load balancing*. O'Reilly, 2001.

[2] George Coulouris, Jen Dollimore, and Tim Kindberg. *Distributed Systems - Concepts and Design, 3rd edition*. Addison Wesley, 2001.

[3] 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 Systems Principles (SOSP '01)*, Chateau Lake Louise, Banff, Canada, October 2001.

[4] Hans Dobbertin, Antoon Bosselaers, and Bart Preneel. RIPEMD-160: A strengthened version of RIPEMD. In *Fast Software Encryption*, pages 71–82, 1996.

[5] Knowbuddy's Gnutella FAQ. `http://www.rixsoft.com/Knowbuddy/gnutellafaq.html`.

[6] David Karger, Eric Lehman, Tom Leighton, Mathhew Levine, Daniel Lewin, and Rina Panigrahy. Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the world wide web. In *ACM Symposium on Theory of Computing*, pages 654–663, May 1997.

[7] 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.

[8] James F. Kurose and Keith W. Ross. *Computer Networking - A Top-Down Approach Featuring the Internet*. Addison Wesley, 2. edition, 2003.

[9] P. Maymounkov and D. Mazieres. Kademlia: A peer-to-peer information system based on the XOR metric. In *Proceedings of IPTPS02*, March 2002.

[10] The network simulator, 2003. `http://www.isi.edu/nsnam/ns/`.

[11] Sylvia Ratnasamy, Paul Francis, Mark Handley, Richard Karp, and Scott Shenker. A scalable content addressable network. In *Proceedings of ACM SIGCOMM*, 2001.

[12] Jordan Ripper. Why Gnutella can't scale. No, really., February 2001. `http://www.darkridge.com/~jpr5/doc/gnutella.html`.

[13] Luigi Rizzo. Dummynet: a simple approach to the evaluation of network protocols. *ACM Computer Communication Review*, 27(1):31–41, 1997.

[14] Matt Robshaw. On recent results for MD2, MD4 and MD5. *RSA Laboratories' Bulletin*, 1996.

[15] Antony Rowstron and Peter Druschel. Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems. *Lecture Notes in Computer Science*, 2218:329–350, 2001.

[16] SHA-1, 2003. `http://csrc.nist.gov/cryptval/shs.html`.

[17] Ion Stoica, Robert Morris, David Karger, M. Francs Kaashoek, and Hari Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In *Proceedings of the 2001 conference on applications, technologies, architectures, and protocols for computer communications*, pages 149–160. ACM Press, 2001.

[18] Michael Sørensen. *En introduktion til sandsynlighedsregning*. Afdeling for Teoretisk Statistik, 3. edition, 2002.

[19] VMware is virtual infrastructure, 2003. `http://www.vmware.com/`.

[20] Werner Vogels. File system usage in windows NT 4.0. In *Symposium on Operating Systems Principles*, pages 93–109, 1999.

[21] Eric W. Weisstein. Poisson process – from mathworld, 2003. `http://mathworld.wolfram.com/PoissonProcess.html`.

[22] B. Y. Zhao, J. D. Kubiatowicz, and A. D. Joseph. Tapestry: An infrastructure for fault-tolerant wide-area location and routing. Technical Report UCB/CSD-01-1141, UC Berkeley, April 2001.

Appendix A

Throughput test results

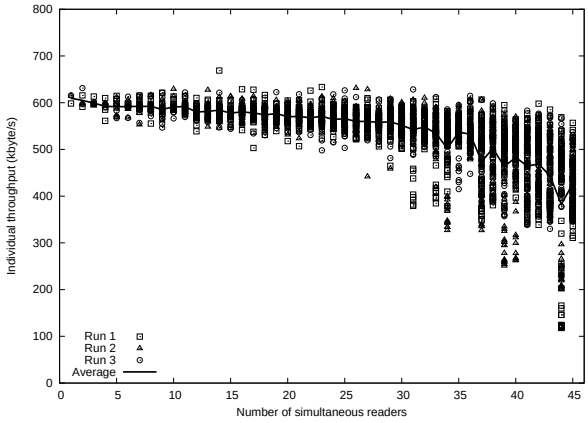

Figure A.1: Read throughput for $r = 1$.

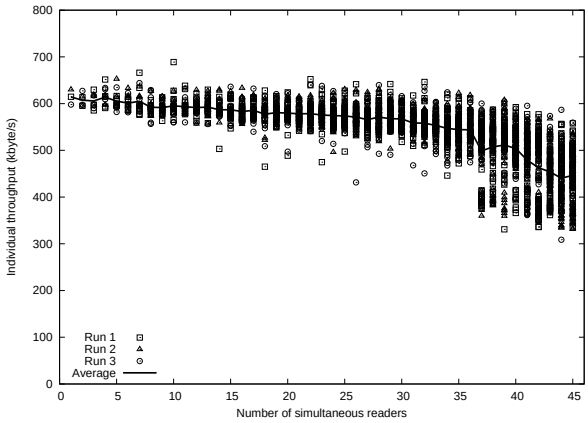

Figure A.2: Read throughput for $r = 2$.

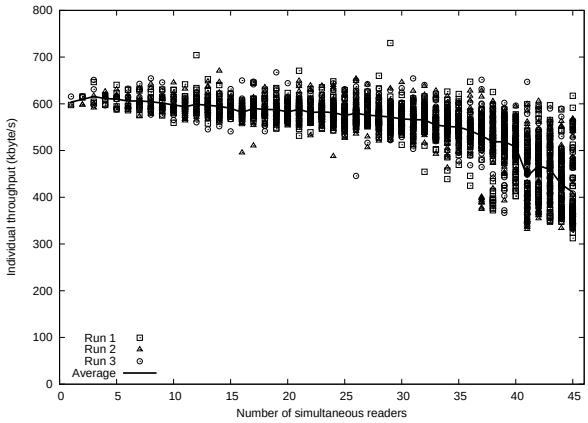

Figure A.3: Read throughput for $r = 3$.

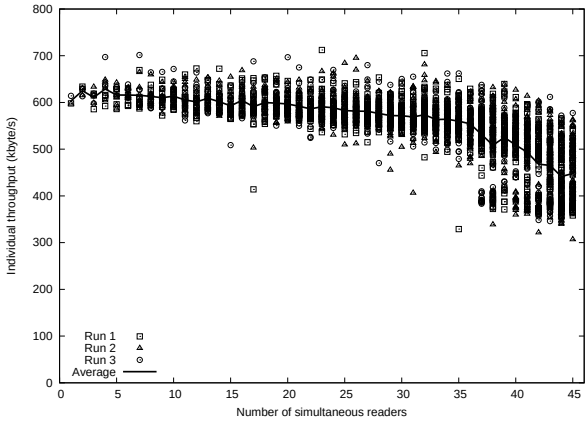

Figure A.4: Read throughput for $r = 4$.

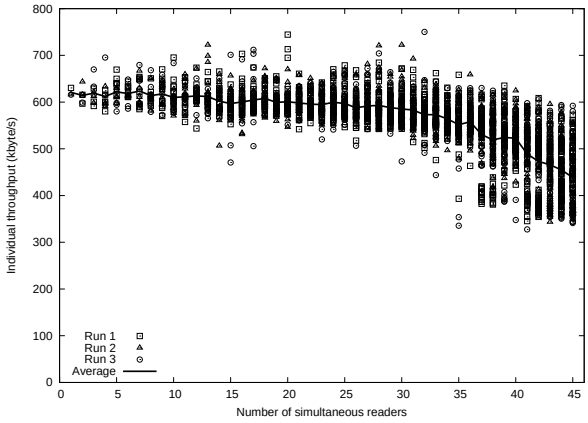

Figure A.5: Read throughput for $r = 5$.

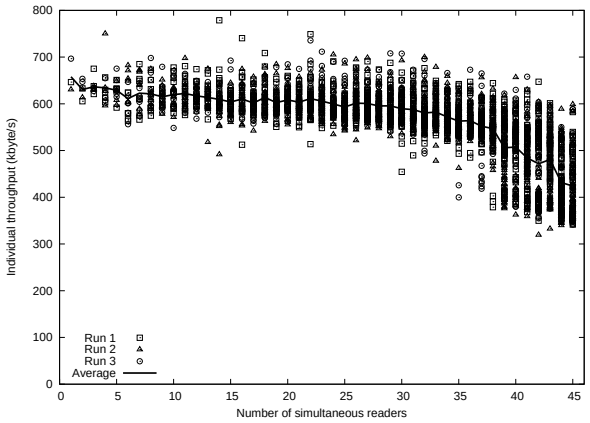

Figure A.6: Read throughput for $r = 6$.

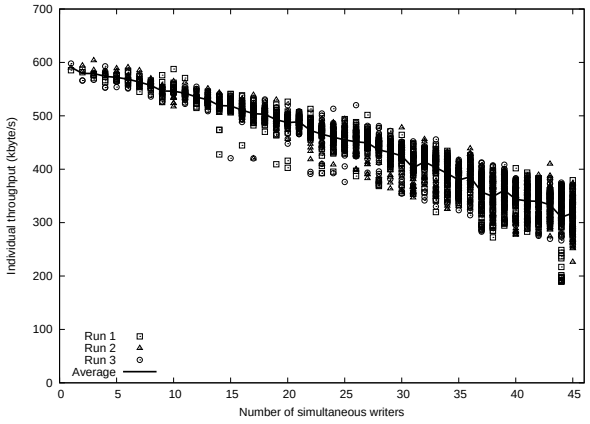

Figure A.7: Write throughput for $r = 1$.

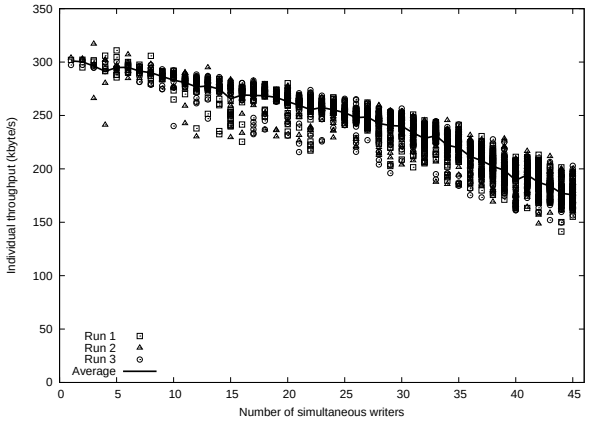

Figure A.8: Write throughput for $r = 2$.

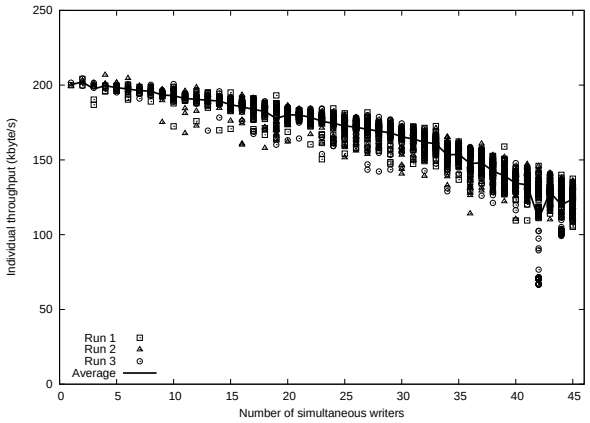

Figure A.9: Write throughput for $r = 3$.

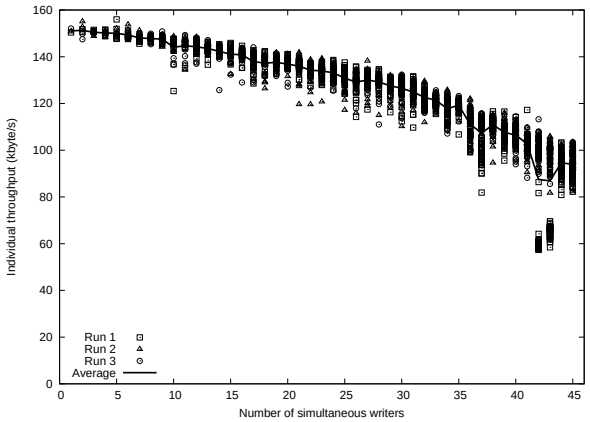

Figure A.10: Write throughput for $r = 4$.

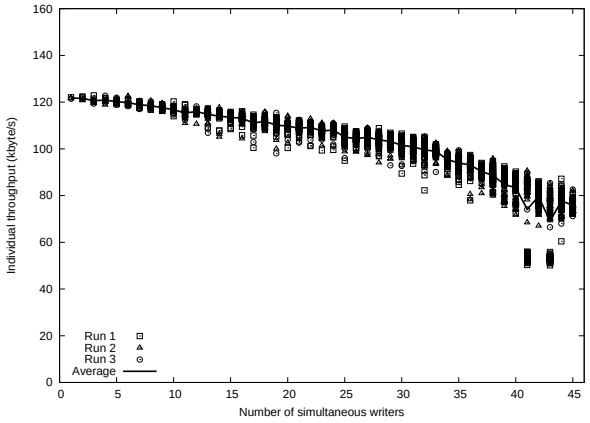

Figure A.11: Write throughput for $r = 5$.

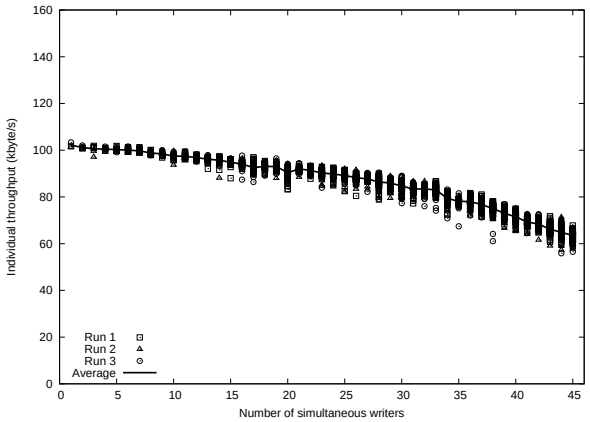

Figure A.12: Write throughput for $r = 6$.