Final report

A Java Implementation of a Highly Available, Scalable and Distributed Data Structure, LH*g

Ronny Lindberg

LiTH-IDA-Ex-97/65

1997-11-09

Abstract

Distributed systems use parallelism and can offer improved performance and scalability. New data structures for distributed systems, that can take advantage of this, are needed. Some applications also require high availability. LH*g is a high availability variant of the hash-based LH* Scalable Distributed Data Structure. It scales up with constant key search and insert performance, while surviving any single-site unavailability (failure). The results of building a thread-based prototype of LH*g in Java is documented in this report. This includes architecture, design and the protocol for message passing. A number of problems were identified and alternative solutions were in most cases proposed, implemented and tested. Measurements on the prototype are reported and discussed.

Acknowledgements

I would like to thank my advisor Professor Tore Risch for his enthusiasm and support during the work on this final report. It has been very interesting.

The help I got from Christer Elvin with the English language was very appreciated. Thank you for all the corrections and for making it through the report.

Contents

1 Introduction. 9

1.1 Background 9
1.2 The task 10
1.3 Delimitations 11
1.4 Overview. 11

2 Theoretical Background 12

2.1 Data structures 12
2.2 Distributed systems. 12
2.3 Scalable Distributed Data Structure (SDDS). . . . 14
2.4 High availability 14

3 The LH* Data Structure 16

3.1 LH* properties 16
3.2 LH* file structure. 16
3.3 Splitting 16
3.4 Addressing 18
3.5 Scan search. 19

4 The LH*g Data Structure 21

4.1 LH*g properties 21
4.2 LH*g file structure. 21
4.3 Primary bucket recovery 22
4.4 Parity bucket recovery. 23
4.5 Coordinator recovery 23
4.6 Record recovery 23
4.7 Example 24

5 Design Approaches 26

6 Design 27

6.1 Architecture 27
6.2 Clients and servers 28
6.3 The Coordinator 29

6.4 Prototype components. 29

7 The Protocol 30

7.1 Packets 30
7.2 How the protocol is described. 32
7.3 Insert/Find. 33
7.4 Update of physical allocation table (PAT) 34
7.5 Scan 35
7.6 Split 36
7.7 Primary bucket recovery 40
7.8 Parity bucket recovery 45
7.9 Coordinator recovery 46
7.10 Forwarding caused by recovery 49

8 Implementation. 51

8.1 The implementation language – Java 51
8.2 Sites and communication 51
8.3 Implementation problems 52
8.4 Not implemented. 54

9 Performance Measures 55

9.1 Simulation setup. 55
9.2 The first measurement 55
9.3 The second measurement 56
9.4 The third measurement 58

10 Conclusion and Future Work 60

11 References 61

1 Introduction

This first chapter gives the background of the report, the task and delimitations are then discussed, and finally there is an overview of the report.

1.1 Background

The traditional architecture is to deal with data through a single processor and its main memory (RAM), with disk as secondary storage. However, more and more applications need fast analysis of a very big and often unpredictable amount of data. The solution is parallelism, i.e. the process of performing tasks concurrently (Lewis et al., 1992). A distributed system uses parallelism and can, according to (Özsu et al., 1991), offer improved performance and scalability.

One target for a distributed system is multicomputers, i.e. collections of workstations or PCs over a network, or of share-nothing processors with a local storage linked through a high-speed network or bus (Tanenbaum, 1995). Almost all computers nowadays are connected to a network, and therefore part of such a collection. The distributed RAM in these networks can reach many gigabytes. Multicomputers offer best price-performance ratio (Tanenbaum, 1995) and are potentially superior supercomputers in computational power. The problem is to efficiently use the potential of multicomputers.

A frequent problem when dealing with distributed systems is that if too many or too few computers (sites) are used, performance deteriorates (Litwin et al., 1997). The optimal number of sites is either unknown or may vary during processing. *Scalability*, as defined in (Karlsson, 1997), is characterized by graceful growing when the amount of data increases. Further, a scalable data structure has (more or less) constant insert and retrieval time, and can (theoretically) handle any amount of data.

Another problem is that when more and more computers are used for distribution, the probability that all of them are working correctly at the same time is dramatically decreased. Therefore, *fault-tolerance* is necessary to increase the availability, i.e. the percentage of time a system is working correctly (Torbjørnsen, 1995).

The LH* data structure (Litwin et al., 1993) and (Litwin et al., 1996) is a hash-based distributed data structure that is scalable; it grows gracefully when the amount of data increases. The extension of LH* called LH*g (Litwin et al., 1997) is fault-tolerant – it survives any single site unavailability (failure) – and is therefore very suited for use in multicomputer-based systems. The high availability is required by many applications that need reliable data storage.

1.2 The task

The main purpose of the task is to verify the theory of LH*g (Litwin et al., 1997) and gain experience from building a prototype, and also to be able to draw some conclusions on the performance and usability of the data structure.

This final report from Computer Science and Engineering, University of Linköping, describes a prototype implementation in Java of the LH*g data structure. Simulations on the prototype were made to investigate the properties of the data structure.

The properties of LH*g is described in (Litwin et al., 1997). However, the exact protocol for message passing to use in the prototype had to be developed and tested. Problems with the protocol that had to be solved along the way are discussed.

The prototype runs in a single multi-threaded computer process, instead of being distributed over several sites, to ease debugging, testing and analysis. The availability of LH*g was studied with help of a “bomber”, that, with variable frequency, stopped a thread to simulate an unavailability of a server. The prototype uses integers as data.

The following basic LH*g operations were implemented in the prototype:

- insert – add/overwrite a record.
- find – get a record given the key (if the record exists).
- scan – get all records that match a given query.

Finally, measurements were made to investigate how different parameters affect the performance.

1.3 Delimitations

To simplify the implementation it is assumed that a message cannot disappear, show up or get to the wrong address when sent between two parts of the system. The message passing is thus considered reliable like for example TCP, and could actually be reimplemented in a non-prototype. One problem arises when a server is brought down to simulate a crashing server. The server sometimes needs to queue packets for later transmission, and none of these packets may get lost according to the assumption. A server is therefore only brought down when no messages can disappear, which is not very realistic.

To delimit the work on this final report, the number of measurements is delimited to only a few.

1.4 Overview

In the next chapter the theoretical background of the area is given, followed by a detailed description of LH* in chapter 3. Finally, LH*g is described in chapter 4.

The report continues with a chapter (5 “Design Approaches”) discussing the methods and ideas used during design and implementation. The design is presented in chapter 6 “Design”, followed by the protocol and detected problems in chapter 7. Some implementation details are discussed in chapter 8, and then the measurements done are analysed in chapter 9. Chapter 10 “Conclusion and Future Work” concludes the report.

2 Theoretical Background

The theoretical background gives an introduction and background to the area of interest, i.e. the highly available, scalable and distributed data structure LH*g. First, data structures and distributed systems in general are discussed. Then scalable distributed data structures are defined along with the concept of high availability.

2.1 Data structures

To be able to store high volumes of data, as is often required in modern data intensive systems, efficient data structures are needed. The traditional ones are e.g. Linear Hashing (Litwin, 1980) and B(+)-Trees. They are mainly designed for use on a single computer, the server. If the amount of data is larger than a single server can process efficiently, (Litwin et al., 1993) suggests the use of a distributed data structure that distributes the data over a number of computers (sites).

2.2 Distributed systems

There is a trend towards distribution and parallelism, and it is due to many factors (Litwin et al., 1996):

- Whatever capabilities a single processor or site has, a pool of sites can provide more resources and power.
- The evolution of high speed networks makes it more efficient to use the RAM of another computer than to use a local disk.
- It is common to find organizations with hundreds of interconnected sites, with dozens of megabytes per site, allowing a distributed RAM file to reach gigabytes.

There are several design issues connected to distributed systems (Tanenbaum 1995). *Transparency* is the problem of making the distributed system look like an ordinary time-sharing system, and can be achieved at two different levels. Easiest to do is to hide the distribution from the user, so that from the commands issued and the results displayed the user can not tell whether the system is distributed or not. At a lower level, the system can be made to look transparent to programs, i.e. the system call interface can be designed so that the existence of multiple processors is not visible.

One of the original goals of building distributed systems was to make them more reliable than single-processor systems. We need to distinguish various aspects of reliability. *Availability* can be defined as the fraction of time the system is usable, and can be enhanced by a design that does not require simultaneous functioning of critical components. Redundancy can further improve the availability by replicating pieces of hardware or software. *Security* is another aspect of reliability. Resources must be protected from unauthorized usage. Still another issue is *fault-tolerance*. The system should be able to continue working correctly even if a server crashes.

Performance is of course an issue. The distributed system must at least run as fast as would a single-processor system. To optimize performance, one often has to minimize the number of messages, because the network communication is the bottleneck.

Scalability is the issue of maintaining performance even though the distributed system grows. Although little is known about huge distributed systems, one clear guideline exists: one should avoid centralized components, tables and algorithms. A centralized component easily becomes a hot spot, if not depending on the CPU, the network capacity into and out of it would surely become a problem. Centralized tables are almost as bad, because even if the storage space suffices, again the network capacity becomes the problem. A centralized algorithm will cause the same problem as above, when collecting information from all sites for processing and then distributing the results. Decentralized algorithms, with the following characteristics, should instead be used:

1. No machine has complete information about the system.
2. Machines make decisions based on local information only.
3. Failure of one machine does not ruin the algorithm.
4. There is no implicit assumption that a global clock exists.

The last is perhaps not so obvious. An algorithm may not state that at exactly a specified time all machines should do something, because it is very difficult to get all the clocks exactly synchronized in a distributed system.

2.3 Scalable Distributed Data Structure (SDDS)

When dealing with distributed data structures, there are certain design issues that are specially interesting. A *Scalable Distributed Data Structure* (SDDS) is defined in (Litwin et al., 1993) and must meet the following constraints:

1. A file expands to new servers gracefully, and only when servers already used are efficiently loaded.
2. There is no centralized directory.
3. The file access and maintenance primitives, e.g. search, insert or split, never require atomic updates to multiple clients.

The first two of these constraints has to do with scalability, while the third more corresponds to the property of a distributed system – multiple, autonomous clients may never be simultaneous available. It is also stated in (Litwin et al., 1993) that to make an SDDS efficient, the message exchange should be minimized while the load factor should be maximized.

Today, several SDDSs exist, e.g. DDH (Devine, 1993),
RP* (Litwin et al., 1994) and different variants of
LH* (Litwin et al., 1996) – among them
LH*g (Litwin et al., 1997).

The advantages with an SDDS are those of a distributed system in general (see 2.2 “Distributed systems” on page 12) and the constraints make sure that the data structure is scalable.

2.4 High availability

It is well known that some applications require *high availability*, i.e. an unavailability of not more than five minutes per year. As noted in part 2.2 “Distributed systems” on page 12 availability is a design issue. There are several ways to achieve high availability (Torbjørnsen, 1995), and the most important ones are discussed below.

“It is better to avoid faults at all instead of having to handle them” (Torbjørnsen, 1995). This is called *fault avoidance*, and the point is that after a fault has occurred it is difficult both to detect and correct it. Often a system is single fault-tolerant, i.e. only one faulty component can be handled. To avoid a second

fault causing service interruption, faulty components should be repaired or replaced, which is called *maintenance*.

Fault-tolerance is based on the fact that faults exist, and means that the system is able to handle faults without going down. Redundancy is the key word, and can be used both to detect, analyse and recover from faults. The redundancy needs not to have the exact functionality as the module to be checked, for example a single parity bit can detect single bit errors in a 32 bit word.

When a fault has been detected the module has to be recovered into a consistent state. However, there are some negative consequences with replication that has to be taken into account:

- *Reduced fault-tolerance* – before the recovery is finished, the system is more vulnerable to new faults.
- *Performance degradation* – the recovery might use resources and there are less resources to serve the clients during a failure. This is causing an overall performance degradation.
- *Temporary unavailability* – there is a latency from when a fault occurs until it is detected, diagnosed, recovered and the service again is available. In this time period the service will appear unavailable.

There are three main strategies for recovery from a hardware error (Torbjørnsen, 1995):

- *Await replacement* – wait until the failed node has been repaired, and then recover the node.
- *Distributed spare* – the lost information is recovered at spare capacity in the rest of the nodes. When the node has been repaired, it is reconstructed by moving the recovered information back to the node.
- *Dedicated spares* – instead of spreading capacity over all nodes, a few nodes are kept passive and serve as spare nodes. When a failure occurs the information is reconstructed at a spare node. The failed node becomes a spare node when repaired.

3 The LH* Data Structure

To help explain how the subject of this report, the LH*g structure, works, the basis, i.e. LH*, is described in detail in this chapter.

3.1 LH* properties

LH* is an SDDS (recall 2.3 “Scalable Distributed Data Structure (SDDS)” on page 14) based on distributed hashing that can grow to practically any size. An insertion usually requires one message (three in the worst case) and a retrieval of an object, given its key, usually requires two messages (four in the worst case). Parallel operations (scan searches) on a file are also supported.

3.2 LH* file structure

An LH* file (Litwin et al., 1996) is stored at *server* computers (nodes), and is used by applications at *client* nodes. The file consists of *records* (tuples) identified by (primary) *keys* denoted c in what follows (see Figure 1). There can be a *non-key* part of the record, often structured into attributes (fields).

Startno.	Name	Country
101	Michael Johnson	USA
key	*non-key part*	

Figure 1. An example of a record.

A client can search, insert or delete a record. Records are stored in *buckets* with *capacity* of b records; $b >> 1$. Buckets are numbered 0, 1, 2...(M-1) where M denotes the current number of buckets in the file. There is one bucket of a file per server (different files may share servers though) and therefore buckets can be identified with their servers.

3.3 Splitting

The file starts with N buckets ($N \geq 1$), and scales up with inserts, through bucket splits. The splitting rules are based on those of *linear hashing* (LH). Every split moves about half of the records in a bucket into a bucket at a new server. The splits are done in the order 0...(N-1); 0, 1...(2N-1); 0, 1...(2^jN-1); 0... ;

j = 0, 1... The next bucket to split is denoted bucket n, where n is called the *split pointer* (see Figure 2).

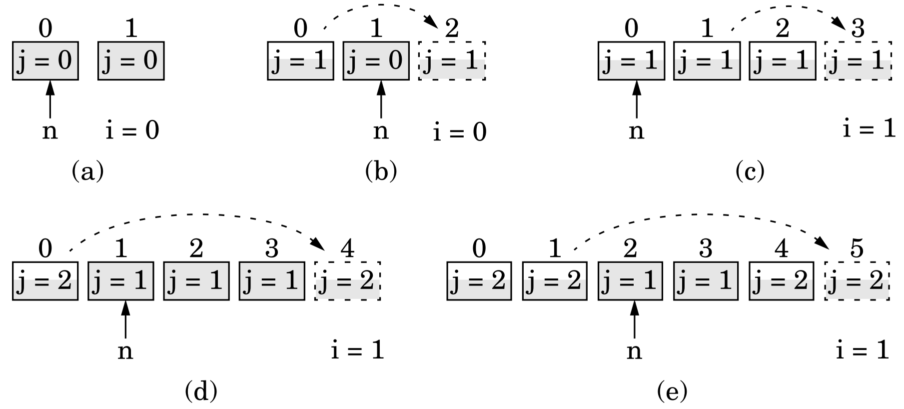


Figure 2. LH file evolving due to splits (N = 2). Shading shows the ideal "water level" in the buckets and dotted arrows mean "half the water was moved".*
- *(a) Two buckets created (N = 2). Bucket level j = 0 for both buckets, split pointer n = 0 and file level i = 0.*
- *(b) Overflow triggered split of bucket 0. Bucket 2 was created and rehashing of bucket 0 with h_1 moved some records to bucket 2. Bucket level j was set to j := 1 for both buckets and split pointer n was moved forward.*
- *(c) Split of bucket 1. Split pointer n reset to 0 and file level increased, i = 1.*
- *(d) Split of bucket 0.*
- *(e) Split of bucket 1. After the split of bucket 2 and 3, the split pointer n will again be reset and the file level i increased by one.*

A family of hash functions h_l, l = 0, 1... are used to perform the splits. Each h hashes a key c into bucket address $h_l(c) = c \bmod (2^l N)$. A split results in the replacement of function h_l currently used for bucket n with function h_{l+1}. This usually re-maps about half of the records into the new bucket $n + 2^l N$.

The splits are triggered by bucket overflows, which are reported to a dedicated node, e.g. bucket 0, called the *coordinator*. It then applies the *load control policy* to find out whether the overflow should trigger a split. If so, the coordinator initiates the split of bucket n.

Every bucket keeps track of its *bucket level*, denoted j. When the file is created the bucket level of all N buckets are initialized to $j = 0$. Then, whenever any bucket m splits, j is increased by one to $j + 1$. This corresponds to the index $l = j + 1$ of the hash function used to split the bucket, and it is also the level of the new bucket created (see Figure 2). The result of this is that at any time one only has $j = i$ or perhaps $j = i + 1$, for some $i = 0, 1...$ called the *file level*. The coordinator is the only node that knows the current values of n and i, collectively called the *file state*.

3.4 Addressing

To avoid a hot spot, the clients do not access the coordinator for the address computation. Instead each client caches an *image* of the file consisting of two parameters, n' and i'; $n' = i' = 0$ for a new client. The client uses these parameters to calculate the address (bucket number), a', where the request for insert, key search or deletion should be sent. These requests are sent using point-to-point messages.

It might happen that the address is not the correct address, a, since n' and i' may differ from the actual n and i (known only by the coordinator). Hence, any bucket m receiving a request first tests whether $m = a$. If the test fails, the server can calculate a better address and forwards the request there. A major property of LH* is that every request is delivered to the correct address in at most two forwardings. For an example of forwarding see Figure 3 below.

The correct server that got a forwarded request sends an *Image Adjustment Message* (IAM) to the client, making it possible for the client to update its image. The update algorithm guarantees that as long as there is no new split, an addressing error is not repeated.

The addresses a above are logical addresses (bucket numbers) and have to be translated to actual (physical network) addresses at both clients and servers. The mapping is in a *physical allocation table*, which can be static or may dynamically expand with the file.

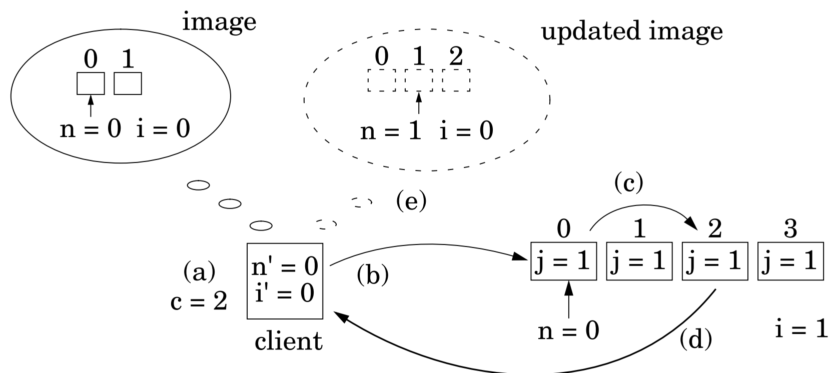


Figure 3. Example of forwarding (N = 2).

(a) The client wants to insert a record with key c = 2.

(b) According to the image there are two buckets in the file. The key hashes to bucket 0, so an insert packet is sent there.

(c) Bucket 0 tests if it is the correct receiver. The test fails and the address to forward to is calculated to bucket 2. The request is forwarded to bucket 2.

(d) Bucket 2 is the correct receiver, so the insert is done there. Because of the forwarding, an IAM is sent back to the client.

(e) The client receives the IAM and updates the image so that it better matches the actual file.

3.5 Scan search

An LH* file also supports the *scan search*, or *scan* in short, where every record is searched for some non-key values. The client ships a scan in parallel to every bucket, preferably using multicast.

The client must then determine when all replies have arrived. If *probabilistic termination* is used, only servers with selected records reply, and a time-out is set after the reception of every record.

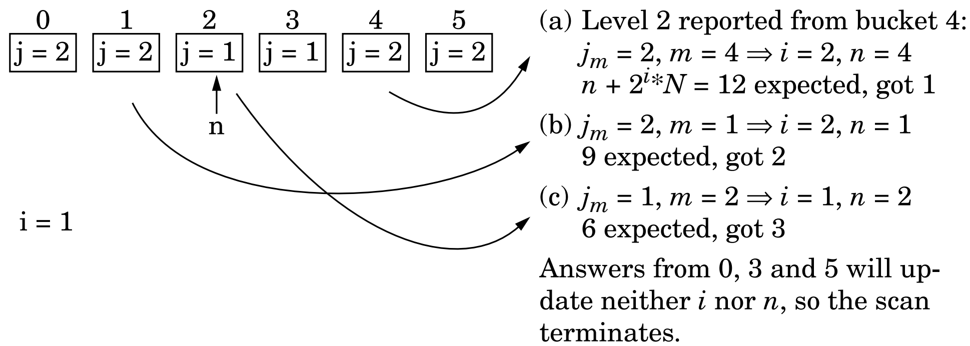


Figure 4. Example of deterministic termination of a scan search (N = 2).

Alternatively, the client uses *deterministic termination* to make sure that all selected records arrive for sure (see Figure 4 for an example). Every bucket replies with at least its address m and its level j_m. The client then terminates when it has received, in any order, $m = 0, 1...(2^i + n - 1)$ where $i = \min(j_m)$ and $n = \min(m)$ with $j_m = i$. Simplified – figure out the file state (n and i) from which the number of buckets can be calculated ($= n + 2^i*N$), and then make sure that all buckets replied.

4 The LH*g Data Structure

Now, enough theory has been described to finally take a closer look at the LH*g data structure.

LH*g (Litwin et al., 1997) is an extension of the LH* data structure described previously in chapter 3 “The LH* Data Structure” on page 16. It is a *high-availability* (fault-tolerant) schema obtained through *record grouping*. A record group of up to *k* members is a logical construct over an LH* file, called the *primary file*. The records in a record group are all stored at different buckets, and a combination of the records are stored in a second LH* file, called the *parity file*. The combination is calculated so that if any single record, including the parity record, becomes unavailable, one may recreate its contents. The fault-tolerance is thus achieved with redundancy (the parity file) that does not have the exact same functionality as the primary file (as described by Torbjørnsen, 1995 – see 2.4 “High availability” on page 14).

The main difference between LH* and LH*g is the high availability added for LH*g. This is needed by many applications that require high availability of data to work properly. The LH* structure can not guarantee this, because if there are many buckets in a file, the probability for unavailability (failures) increases. A failure of a bucket causes permanent loss of the data in that bucket, and this might not be accepted.

4.1 LH*g properties

The LH*g schema allows the file to survive any single bucket (server) unavailability without any loss of data. The failure-free search cost is that of LH*, i.e. 2 messages in general, and the general insert cost is twice as big as for LH* – 2 messages. The storage for an LH*g file is about $1/k$ times larger than that for the LH* file.

4.2 LH*g file structure

An LH*g file is constructed out of two LH* files, called *primary* and *parity* files. A single LH*g coordinator manages the file state for both files, and is maintained by primary bucket 0. The primary file, F1, has initially *k* buckets (= *N* for LH*) and contains *primary records*. Every bucket *m* in F1 belongs to some

bucket group $g1$ = Int(m/k) of size k, and is also provided with an *insert counter*, r = 0, 1, ... When a record c comes for storage to a bucket in F1 for the first time, it gets associated with successive values of r – the record was *inserted*. If a record, on the other hand, is *moved* to the bucket due to a split, the record keeps the r value it originally got.

A *record group* is a logical construct of up to k records, identified by the *group key* $g = (g1, r)$. The group key is added to the original record (see Figure 5), and is then never changed.

A *parity record* (see Figure 5) with g as key exists in the parity file, F2, for every record group in F1. The parity record also contains all keys, $c_1, ..., c_l$; $l \leq k$, of the records in group g. The non-key part of a primary record is then considered as a bit string, so that *parity bits* can be calculated using (even) parity, and also stored in the parity record. The content of a parity record suffices to reconstruct any primary record, if all the other records in the record group are available.

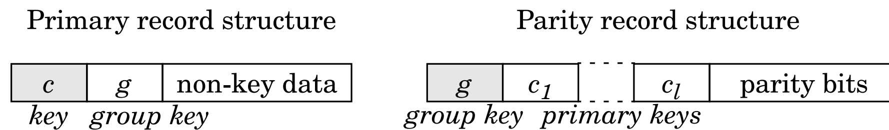


Figure 5. Primary and parity record structures.

4.3 Primary bucket recovery

If a primary bucket, m, is unavailable, the coordinator needs to recover all the records in the unavailable bucket and the record counter r. In short the algorithm for doing this is as follows – for the complete algorithm, see (Litwin et al., 1997):

1. The coordinator issues a scan in F2 requesting the number of records ever inserted into m. The sum of the answers makes r.
2. Every bucket in F2 should for each record containing a key that was stored in bucket m at the time of the failure, recreate the corresponding non-key data. The recreation is done by finding (in F1) all other records in the same record group, i.e. the records matching all other keys in the parity record. Then the missing non-key data can be reconstructed by combining the found records with the parity bits.

3. The primary key and the reconstructed non-key data make a reconstructed record, which is inserted into a new bucket with the same number, m, as the unavailable one.

4.4 Parity bucket recovery

A parity bucket can of course also become unavailable, and this is handled as follows (again refer to (Litwin et al., 1997) for the complete description):

1. A scan is issued in F1, requesting all records that were in the failed bucket. This is determined using the current file level, i, and split pointer, n, of F2.
2. These records are inserted in the new bucket of F2 (as a new record normally would).

4.5 Coordinator recovery

If bucket 0 of F1 fails (where the coordinator is located), then the file state (n and i) for both F1 and F2 has to be recovered, followed by a “normal” primary bucket recovery. The algorithm is quoted from (Litwin et al., 1997), because it turned out not to work deterministically without some modification (see 7.9 “Coordinator recovery” on page 46 for discussion):

1. A scan Q_3 with deterministic termination is sent to all available buckets of F1 and F2. Q_3 requests from each bucket its address m and its bucket level $j(m)$.
2. If there is some m received such that $j(m - 1) = j(m) + 1$, then assign n to $n := m$ and $i := j(m)$.
3. If no such m is found assign M to the largest m retrieved and i to $i := j(1)$. If $M = k*2^{j(1)} - 1$, then assign $n := 0$; otherwise assign $n := 1$.

In short, request from each bucket its bucket number and level. Then the file state for both files can be figured out.

4.6 Record recovery

To speed up the key search in F1 for record c in case of unavailability, the following algorithm can be used concurrently with the primary bucket recovery:

1. The coordinator sends a deterministic scan to F2, requesting the parity record g containing c.

2. If such a record is not found, the record does not exist.
3. Otherwise the coordinator issues for every other key in g a key search in F1. Then record c can be reconstructed using all the found records and the parity bits.

4.7 Example

An example of how the scaling and replication work in practice will now be presented. In Figure 6 below an example is shown with bucket capacity $b = 3$ for F1, $b = 5$ for F2 and bucket group size $k = 2$. The record structure is as in Figure 5 on page 22. For a primary record the key c is first, then the group key ($g1$, r) is in parentheses, and last the non-key part is in bit format or left out (...). Parity records start with the group key, followed by two primary keys ('__' means empty) because $k = 2$. Last are the parity bits, which are calculated using *exclusive-or* (xor).

Figure 6 (a) shows the content of the file just before the first split. The split is triggered by the insert of the record with $c = 36$.

It can be noticed how parity records are constructed, by looking at the first records inserted ($r = 0$) in primary bucket 0 and 1 ($c = 16$ and $c = 13$). The records get the same group key $g = (0, 0)$, because they are in the same bucket group ($g1 = \text{Int}(m/k) = \text{Int}(m/2) = 0$) and they both were inserted as the first record of each bucket. This means that they will be stored in the same record in the parity file, and that the combination of their non-key parts forms the parity bits (101 xor 110 = 011).

It can also be noticed that if a record is alone in its record group, as is for example record $c = 46$, the record is in principle mirrored in F2 (both records are the same – just change the order of the fields).

The split creates bucket 2 (Figure 6 (b)), which will belong to bucket group $g1 = 1$, and sets the insert counter to $r := 0$. Two records are moved from bucket 0 to bucket 2, but neither the group keys of the moved records nor the insert counter of bucket 2 are changed. When the new record $c = 14$ is inserted into bucket 2, the group key is $g = (1, 0)$, because it is the first record ($r = 0$) inserted into the bucket. The insertion of record $c = 24$ triggers the split of parity bucket 0.

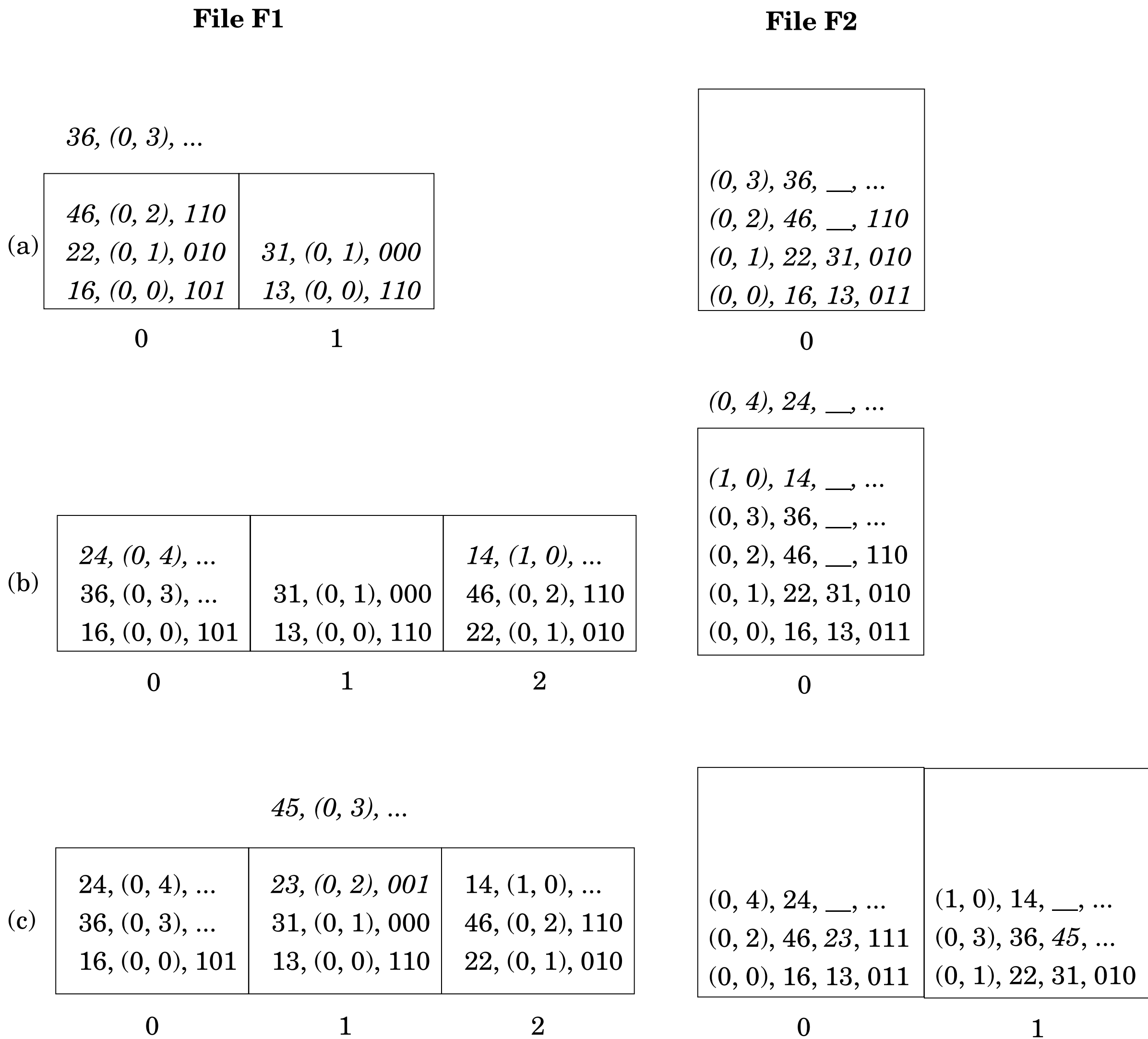


*Figure 6. The evolution of an LH*g file before the first split (a), before the second split (b), and after the second split (c). New and changed records are in italics.*

Figure 6 (c) shows the situation after the split of parity bucket 0. Three records were moved to bucket 1. Two records are then inserted ($c = 23$ and $c = 45$) into bucket 1 of F1. The corresponding records of F2 are updated – the keys are added to each parity record, and the parity bits are updated. Bucket 1 now overflows, which will trigger yet another split (not shown).

5 Design Approaches

The philosophy used to implement the prototype was to design the system as if it was the real distributed system. Threads were used to simulate processes on computers, and the message passing functionality was built of a few primitives (send/receive/broadcast) that easily can be implemented in a real environment. No global state was assumed, so all communication between the nodes used messages.

Java was selected as implementation language because the straightforward use of threads, and because it is a modern object-oriented language. Another plus is that there is high-level support for network communication, which enables the interested to make only some minor changes to the communication primitives (and some more) to get a full-scale system. More discussion of Java follows in 8.1 “The implementation language – Java” on page 51.

The LH*g structure consists of two LH* files (see 4.2 “LH*g file structure” on page 21 for details), and it was then natural first to try to implement a working LH* file, and then to put the two (almost identical) files together to form the LH*g file.

An object-oriented analysis and design were outlined mainly for the LH* file, but the modifications needed to get the LH*g file were kept in mind to get a general and intuitive design. The main idea during the design and implementation was to try the most straightforward solution first, just to make things work. Later, modifications could be necessary to improve performance or functionality.

6 Design

In this chapter the architecture of the system is described, followed by a high-level design description of the three main parts of the system – client, server and coordinator. Finally some special prototype components are briefly discussed.

6.1 Architecture

The main architecture is simple, because the primary file (F1) works like in the LH* case, and the parity file (F2) is an LH* file that stores the parity data for recovery purpose (see Figure 7). From the (application) clients' point of view, the interface to LH*g looks exactly like that of LH*, i.e. the client only "sees" the primary file (F1) – the high availability is transparent.

However, for each insert in F1, an insert has also to be done in F2 in order to keep the parity data up to date. The primary server now acts like a client that wants to insert data into F2, and therefore uses a client that handles the interface between F1 and F2 (Parity Client). The server and the client together reside at a site (numbered from 0 to x for F1, and from 0 to y for F2).

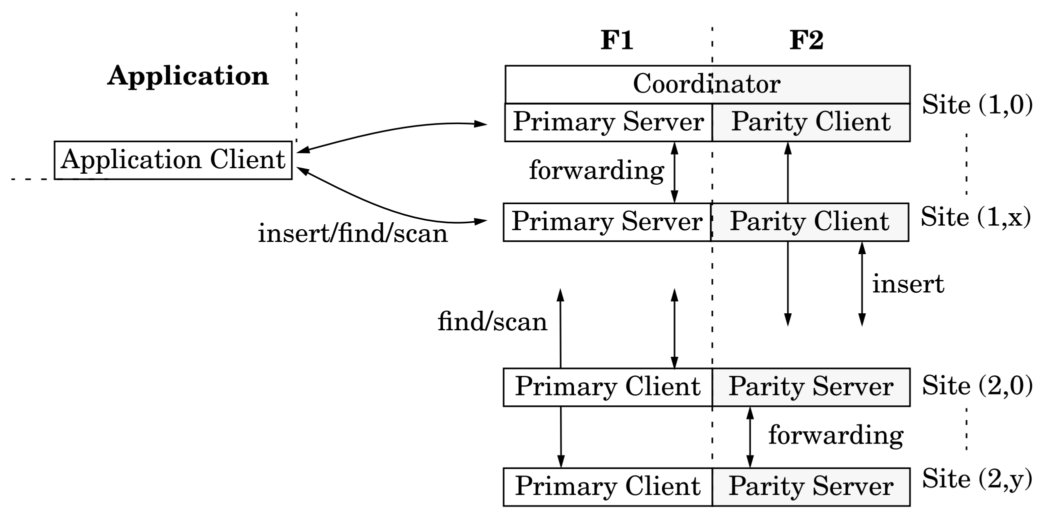


Figure 7. The architecture with important communication paths indicated.

On the other hand, when recovery has to be done, a parity server needs to do find and scan searches in F1, and the interface to the opposite file is again a client (Primary Client). Notice that the client behaves exactly the same in the three cases (though

named differently in Figure 7) – the client only executes basic operations on an LH* file. It is therefore implemented as one component.

6.2 Clients and servers

Both servers and clients are composed of three layers (see Figure 8). The bottom layer (Communicator) implements the primitive platform-dependent communication primitives like send, receive and broadcast.

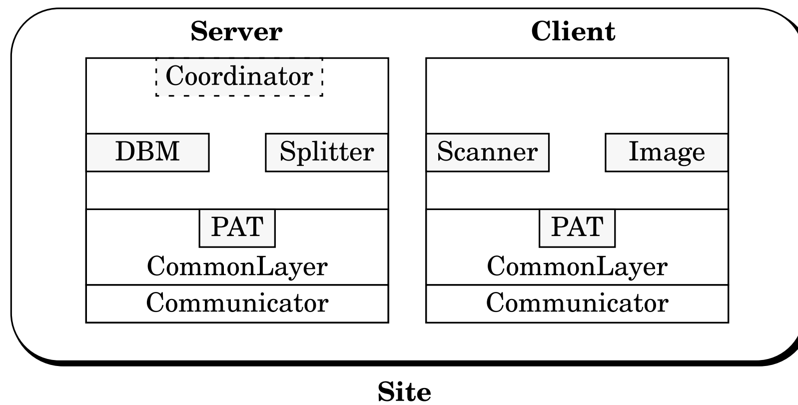


Figure 8. A primary or parity site, with the three layers of the server and the client indicated.

The CommonLayer is also the same in both the client and the server. It implements higher-level send primitives that can detect crashed servers and start recovery. The physical allocation table (PAT) is also kept in this layer and it contains the mappings between logical addresses (bucket numbers) and physical addresses (e.g. a port on a computer).

The top layer contains the specific features of the component. A server contains for example a database manager (DBM) that handles the local bucket, a Splitter that handles the splits, and finally a Coordinator (described in the next part). The Coordinator is only activated for primary bucket 0, but must exist in all servers, because any server can after a failure become the new coordinator, as we will see later in section 7.7 “Primary bucket recovery” on page 40.

A client has in its top layer, among other functionality, a Scanner that handles scan searches and can detect and report possible failures. Each client also has an Image used for addressing

of the LH* file it is working on (recall 3.4 “Addressing” on page 18).

On each site where there is a server, there is also a client that handles the communication with the opposite file (see both Figure 7 and Figure 8). This is important, because it makes the interface between F1 and F2 very easy to handle. In an application, of course, there is only a client (no server).

6.3 The Coordinator

The Coordinator coordinates the entire LH*g-file, i.e. both F1 and F2. It decides when a bucket may split, and when recovery should be started. It is the only place where a guaranteed correct PAT is available, so clients and servers ask the Coordinator when they need to know (details in 7.4 “Update of physical allocation table (PAT)” on page 34).

6.4 Prototype components

Some special arrangements have been made in order to get a more realistic prototype. To simulate the processes distributed in the network, each process got its own thread in the prototype, so that the prototype could run on one computer.

Physical addresses were introduced (though not needed) so that the physical allocation table (PAT) could be simulated. This also caused another table to be introduced to keep the mappings between the physical addresses and the actual threads.

The Bomber brings down one server (thread) at a time at random to simulate a crashing server. The bombing frequency is variable, but more than one server can never be unavailable at the same time.

7 The Protocol

In this chapter the protocol that was used to implement LH*g will be presented as use cases, but first the different packets will be given a short explanation.

7.1 Packets

Messages between clients and servers are included in packets, which the Communicator then can transfer. A short description of all the packets used are now given, and should only be seen as an introduction – the packets are described more in detail in the use cases following. They are grouped according to usage.

7.1.1 Packets used by clients

The client only needs to handle (send and receive) the following packets:

InsertPacket – is sent by a client to insert a record.

InsertAnswerPacket – a reply from a server to a client, whose image has to be updated.

FindPacket – is sent by a client to search for a record, given the key.

FindAnswerPacket – the answer from a server, possibly with image adjustment information included.

ScanRecordPacket – is sent by a client in parallel to all servers, asking a general query.

ScanRecordAnswerPacket – the answer from servers which has matching records. If it is a deterministic scan, all servers reply.

7.1.2 Packets used during split

Some packets are needed to handle the split of a bucket:

WannaSplitPacket – is sent by a server to the Coordinator on bucket overflows.

InitNewBucketPacket – is sent by the Coordinator during a split to setup the new server.

YouSplitPacket – reply from the new server. Indicates that a split is allowed.

SplitTransPacket – used to transfer records to the new server when splitting.

CommitSplitPacket – is sent to the Coordinator when the split has finished.

7.1.3 Packets used for PAT update

To update the physical allocation table, two packets are needed:

PATUpdatePacket – a request for a physical address of a server that is sent to the Coordinator.

PATUpdateAnswerPacket – the answer from the Coordinator.

7.1.4 General recovery packets

Two general packets are always used during recovery:

RecPacket – is sent to the Coordinator when a server can not be reached. The packet that could not be delivered is included.

CommitRecPacket – is sent to the Coordinator when a recovery process has succeeded.

7.1.5 Packets used for primary bucket recovery

The most complex recovery process, primary bucket recovery, needs four special packets:

ScanPrimRecPacket – is sent by the Coordinator and requests from each parity server the number of records that have ever been inserted in the failed bucket. The number of records that will be recovered at each server is also requested.

ScanPrimRecAnswerPacket – the answer to the ScanPrimRecPacket.

PrimRecSetupPacket – is sent from the Coordinator to the new server taking the place of the failed one. Sets up the record counter and tells the new server how many recovered records are expected.

PrimRecInsertPacket – transfers a recovered record from a parity buckets to the new location.

7.1.6 Packet used for parity bucket recovery

Parity recovery only requires one extra packet:

ParityRecSetupPacket – sets up for parity recovery.

7.1.7 Packets used for recovery of file state information

If the Coordinator is unavailable, the file state of both files need to be recovered:

CoordRecSetupPacket – initializes the new Coordinator.

ScanFileStateRecPacket – requests the bucket number and level from all buckets (in both F1 and F2).

ScanFileStateRecAnswerPacket – the answer to the ScanFile-StateRecPacket.

LevelUpdatePacket – is used to replicate the level of bucket 0 to bucket 1 in F1, and is needed for the recovery to work.

7.1.8 Packets used for record recovery

A find request to an unavailable bucket can be speeded up. Then these packets are needed:

ScanFindRecPacket – searches the parity file for the key of the requested record.

ScanFindRecAnswerPacket – the answer containing the parity record (or nothing).

7.1.9 Packet used when server receives wrong-addressed packet

Because none of the PATs are updated after a recovery, outdated addresses to servers exist, causing packets to be sent to the wrong physical address:

ForwardPacket – if the server is not the expected recipient, the packet is sent to the Coordinator which sorts things out.

7.2 How the protocol is described

The protocol will be described by use cases, i.e. different cases (examples) that illustrate what packets are used in different situations in a dynamic way. Each use case will have (at least) one figure showing what messages are sent, and in which order.

The figures in this chapter all have the same structure. At the top the Coordinator (Coord.) is placed between the primary file (F1) to the left and the parity file (F2) to the right (divided by a dotted line). Sites (server and client) are illustrated with ovals (parity sites are shaded ovals), and packets are drawn as numbered arrows. A broadcast is three arrows tightly together, pointing at the file the broadcast applies to. A square is an application client.

To the right, the names of the packets are written out, and the running text also refers to them.

7.3 Insert/Find

Insert and find are issued by a client to insert/replace a record or search for a record given the key.

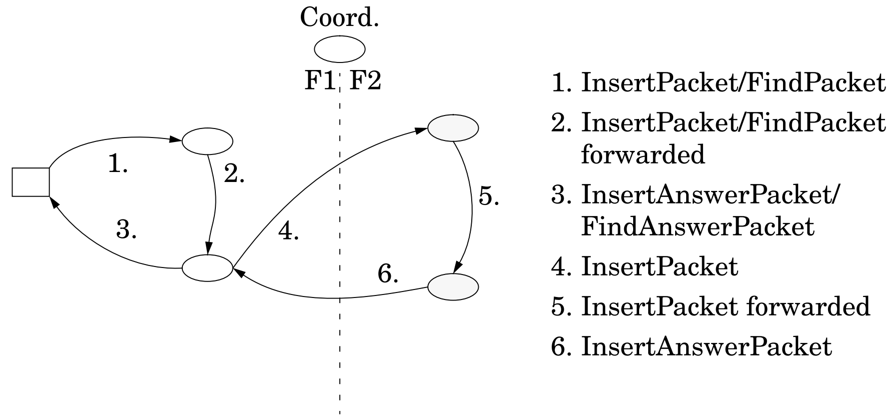


Figure 9. A client issuing an insert (1.-6.) or a find (1.- 3.) in the primary file.

The client (see Figure 9) is the initiator and uses the key to calculate the bucket number according to its image. The client then sends an InsertPacket/FindPacket to that server (1.). The server checks if it is the correct receiver, and if it is not, the packet is forwarded (2.) to another server. When reaching the correct server the insert is done or the bucket is searched for the key. In the insert case an answer (3.) is only sent if there has been a forwarding (as in this example), and it then contains image adjustment information (IAM). In the find case however, an answer is always sent back to the client. It contains the record, if found, or only the key if not (and if forwarded, IA information too).

In the find case we are now finished, but in the insert case, the parity file (F2) also has to be updated. The correct server tells its client (recall Figure 8 on page 28) to insert the record in F2. The next steps (4. - 6.) are then the same as for (1. - 3.).

An interesting note on insert can be done if "replacing insert" is allowed, i.e. if insertion of a new record, with a key that already exists in the file, replaces the old record. Then the group key of the old record must be reused, and the insert counter must not be updated. The old record must also be passed on in the Insert-Packet for F2, so that the parity bits can be updated correctly. If this is not done, F1 and F2 will not be consistently updated, causing recovery not to work correctly. This is the way insert was implemented in the prototype.

An oversight in all available theory descriptions, i.e. (Litwin et al., 1993, 1996 and 1997), caused some confusion. The initial number of buckets were not considered in the image adjustment algorithm, called (A3) in all three papers. The correct algorithm is given in Figure 10 below:

(A3) if $j > i'$ then $i' \leftarrow j - 1, n' \leftarrow a + 1;$
if $n' \geq 2^{i'} * \boldsymbol{N}$ then $n' \leftarrow 0, i' \leftarrow i' + 1;$

Figure 10. The image adjustment algorithm, when the initial number of buckets are considered (a, or a', is the address to which the packet first was sent).

7.4 Update of physical allocation table (PAT)

As described in section 6.2 “Clients and servers” on page 28 a physical allocation table (PAT) at each client and server keeps the mappings between bucket numbers and physical addresses. The only complete PAT that exists in the system is located at the Coordinator. All other PATs are cached copies of the PAT at the Coordinator, updated only when a mapping is not in the table, or when a mapping is incorrect. For this to work, all PATs must know the correct location of the Coordinator (and bucket 1, as will be motivated in section 7.9 “Coordinator recovery” on page 46).

When a packet needs to be sent to a bucket, the local PAT is searched for the matching physical address. If it is not found, the bucket number is sent in a PATUpdatePacket to the Coordinator, requesting the corresponding physical address. The original packet has to wait for a PATUpdateAnswerPacket to get back, and is queued while waiting, to allow other packets to be processed.

If another packet needs to be sent to the same bucket while waiting, a new PATUpdatePacket is not sent, because the reliable sending primitives guarantee that an answer will get back, and is therefore queued directly.

If the mapping is incorrect (due to server crashes and recovery), the receiver detects that and forwards the packet to the Coordinator. Then the Coordinator delivers the packet and sends a PATUpdateAnswerPacket to the sender (see 7.10 “Forwarding caused by recovery” on page 49 for more information on this kind of forwarding).

This way of distributing new and changed physical addresses is the simplest considered. Other, more efficient ways of doing this is proposed in (Litwin et al., 1996), and these should be addressed in future work.

7.5 Scan

A scan search is issued to ask a general (non-key) query. Different specialized scans are also used during recovery (see the sections on recovery, starting at 7.7 “Primary bucket recovery” on page 40), but the main principle is the same.

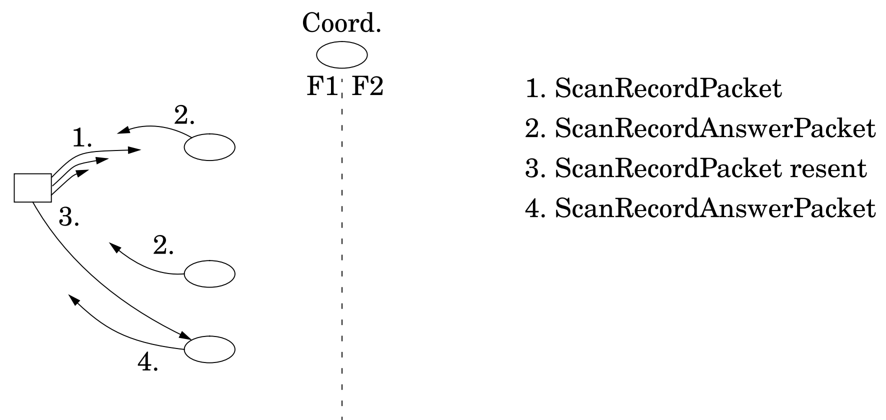


Figure 11. A client doing a scan search of F1, but one server did not answer.

A client sends a ScanRecordPacket (see Figure 11), with the query (in this case requesting the records matching the query) it wants answered in parallel using broadcasting (1.). A Scanner is then used to collect the answers. Only deterministic termination of scan search is implemented, so the Scanner waits for answers (2.) from all the servers (in F1), as described in 3.5 “Scan search” on page 19. If not all servers have replied after a while (due to failure etc.), a time-out causes the ScanRecordPacket to be resent point-to-point to the server not answering (3.). When all the answers have arrived (4.), the scan terminates and the received records can be processed by the client.

The main purpose of the resending is that when sending point-to-point, it can be detected whether the server is up or not. If not, the failure can be reported to the Coordinator, which (possibly) initiates recovery.

The time-out and resending might cause multiple replies to arrive at the client. Consider for example that the broadcast message is processed at the server, but that the answer (call it answer1) has been delayed on the way to the client. Before answer1 arrives, the time-out causes a resending. Now answer1 arrives, and later the answer from the resending (answer2) also arrives. The Scanner now has two answers from the same server, but only one should be included in the answer to the query. The Scanner therefore has to ignore one of the answers (in the implementation answer1 is ignored).

Another problem with multiple replies arises if the scan terminates when answer1 arrives (it was the last answer needed), and then a new scan search is started by the same client. When answer2 arrives, it will be interpreted as an answer from the second scan (which it is not) and this might cause an incorrect answer.

This problem was not dealt with in the implementation, because it rarely appeared and caused no problems (except for corrupt scan results). The solution however, is either to test all incoming records against the query to see if they match, or to mark the answers so that answers to different queries can be distinguished. The first solution is not safe, because the first query might have been requesting a subset of the second query, making the test work, but causing an incorrect answer. The second solution, to mark the answers, can be done in several ways. The safest one is to include the query itself in the answer, but a query can be big, so this might be costly. Another way is to give each scan a unique number, so that subsequent scans can be distinguished.

7.6 Split

When the file grows, buckets need to be split (see 3.3 “Splitting” on page 16 for theory). The first implementation of splits followed the protocol described in (Litwin et al., 1996), but it turned out to cause some problems. The protocol then had to be redesigned to work properly. The first implementation and the problems it caused will now be described, followed by the new protocol and potential problems with it.

The protocol first implemented worked like this (see Figure 12 below).

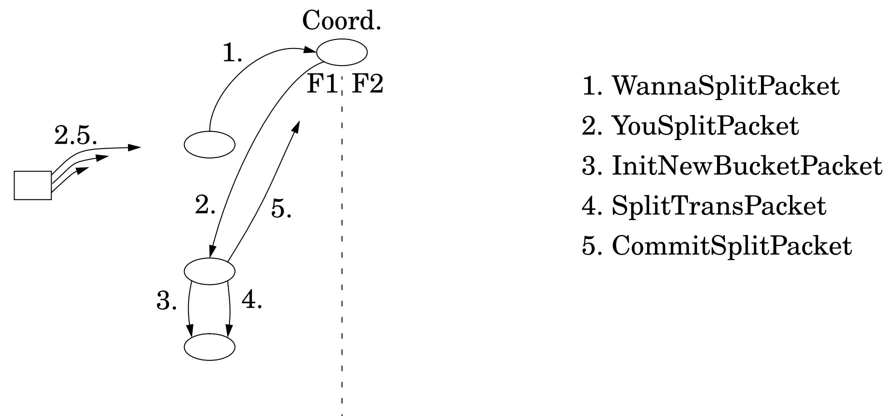


Figure 12. The protocol for split first implemented.

An overflowing bucket reports to the Coordinator (1.). It then applies the load control policy to find out if a split should be allowed. If it is, a YouSplitPacket containing the address of a spare server is sent to the bucket in turn to split (2.). The split is started and a packet is sent to initialize the new server (3.). Then the records to be moved are sent in SplitTransPackets (as implemented, just one) to the new server (4.). Finally a commit is sent back to the Coordinator (5.).

The load control policy used in the implementation was the simplest possible, i.e. uncontrolled splitting. This means that whenever the Coordinator receives a WannaSplitPacket, the next bucket in turn is allowed to split. Other load control policies have been suggested and evaluated in (Karlsson, 1997), (Karlsson et al., 1995) and (Litwin et al., 1996), and are called controlled splitting. What is best for LH*g is however left for future work to investigate.

The records to move from the splitting bucket were all included in one packet, the SplitTransPacket, because it was easiest to implement. However, the results in (Karlsson, 1997) probably apply to LH*g, and this would mean that sending more than one packet with more than one record in each is the most efficient way to do splits. The use of an LH file locally (Karlsson, 1997) would also speed up splits and offer possibilities for concurrency. This must also be addressed in future work.

A small issue to note is the possible race that can occur between the InitNewBucketPacket and the SplitTransPacket if the send primitives can not guarantee that the message first sent arrives first. Then the SplitTransPacket could arrive before the new server is initialized, causing problems. This was not, however, a problem in the implemented system.

The main problem arises when another client is sending a scan when the split is in progress. If the scan is sent after the YouSplitPacket has arrived, but before the new server has been activated (i.e. 2.5. in Figure 12), the answer to the scan from the splitting bucket will be constructed after the split has finished (no concurrency). The answer to the query will therefore only be based on about half the records previously in the bucket (the other half has been moved to the new bucket). The server also reports one more server than before the split, for the Scanner to wait for (because the bucket level j is increased). But the new server was not active when the scan was sent, and could therefore neither receive nor answer the query. This results in a nonterminating scan.

Two different solutions were considered. The first one was that the splitting server would pass scan messages along to the new server – but which packets, starting from when and so on. This would be very complicated to get to work. Instead, why not use the resending functionality already implemented for scan (actually it was not at this stage). This would work, but then we would have to wait until time-out occurs, which might take a while. Then we also allow a failure in the protocol for splits to cause actions designed for fault-detection, and as noted in part 2.4 “High availability” on page 14 – it is better to avoid faults than having to handle them. If we can avoid this, the protocol would be much robuster.

So a small change was made to the protocol. Instead of the Coordinator sending a YouSplitPacket to the splitting bucket, which then sent an InitNewBucketPacket to the new server, the reverse order would guarantee that the new server is initialized before the splitting begins. This is achieved by letting the Coor-

dinator initialize the new server first, which then sends the YouSplitPacket to the bucket to split (see Figure 13 below).

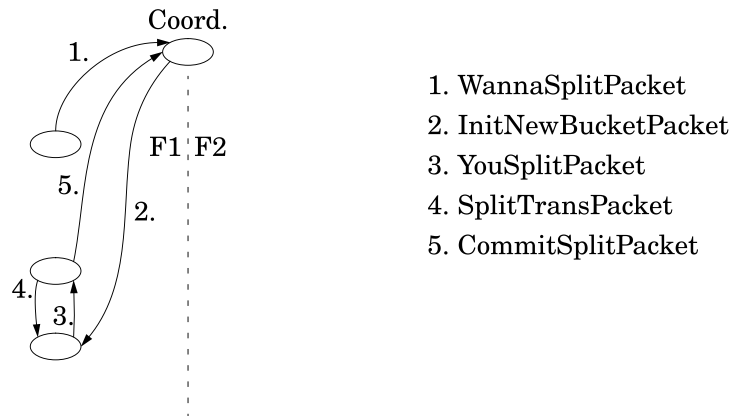


Figure 13. The enhanced protocol for split.

Now a scan is processed either before the YouSplitPacket, reporting a result based on the whole bucket (not knowing of any split), or after the YouSplitPacket, the result based on half the bucket. But in the last case the new bucket also must have got the scan (because it sent the YouSplitPacket). This makes scanning during a split work without resending (almost, see below).

There is however a very rare (theoretical?) case when this protocol also fails (Figure 14).

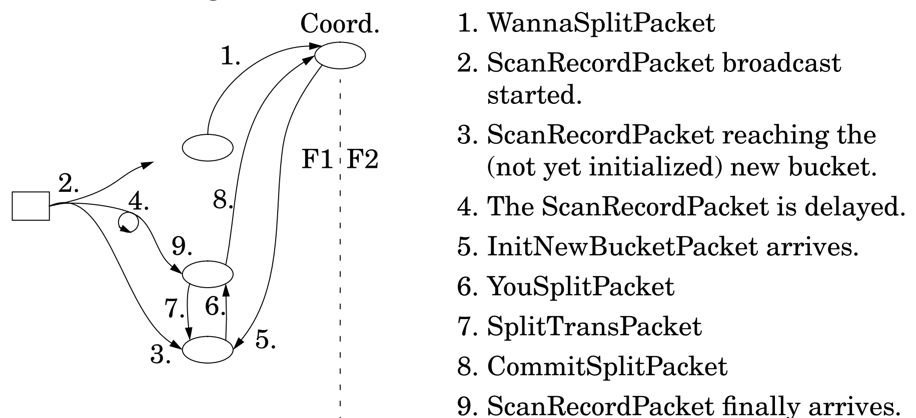


Figure 14. The case when the modified protocol fails.

Consider the scan not reaching the splitting bucket and the new bucket at the same time. Instead the scan (2.) reaches the new bucket first (3.), just before the InitNewBucketPacket. The scan is ignored by the new bucket. The scan packet for the bucket to split is delayed (4.), and in the mean time the InitNewBucket-

Packet arrives at the new server (5.), gets processed, and a YouSplitPacket is sent to the bucket to split (6.). This packet makes it to the splitting bucket before the scan, so the split is started before any scan answer has been constructed. When the scan finally arrives to the (splitted) bucket (9.), the answer will be based on the bucket content after the split. The new bucket ignored the scan (because it was not active) and therefore reports nothing. The scan does not terminate, because the answer from the new bucket is missing. This problem is however so rare that it can be accepted to solve it by using the built-in resending of the Scanner.

On the other hand, another case is now more likely to appear than in the first implementation. This has to do with too many answers. If a scan reaches both the new and the splitting bucket before the YouSplitPacket has arrived, the splitting bucket will not report that the new bucket exists (it does not know yet), but the new bucket will send an answer (it has been initialized; the packet is queued until the split is completed). This is simply solved by the Scanner by just ignoring the answer from the new bucket.

It can be concluded from the above discussion about scans and splits that the Scanner has to keep track of all the received answers – just counting them until the correct number is reached is not enough. The Scanner has to detect when a server is not responding (and which server) and must also handle duplicate and redundant packets.

7.7 Primary bucket recovery

The case when a primary bucket goes down will now be discussed. A possible failure is always reported to the Coordinator by sending a RecPacket, in which the packet that could not be delivered is included (see Figure 15, (1.)). The Coordinator first checks if the target bucket exists at a different physical address than reported in the RecPacket (by looking in its PAT). If the bucket exists at another address, it has already been recovered. Then the packet included is delivered to the new address and a PATUpdateAnswerPacket is sent back to the sender (to update its outdated PAT-entry). If the addresses are the same, the recovery process is started.

7.7.1 Primary bucket recovery – base case

Depending on which packet detected the failure, different ways to recover apply. The base case (Figure 15 below) is first described, and it applies if the reporting packet is an InsertPacket, ScanRecordPacket or PATUpdatePacket.

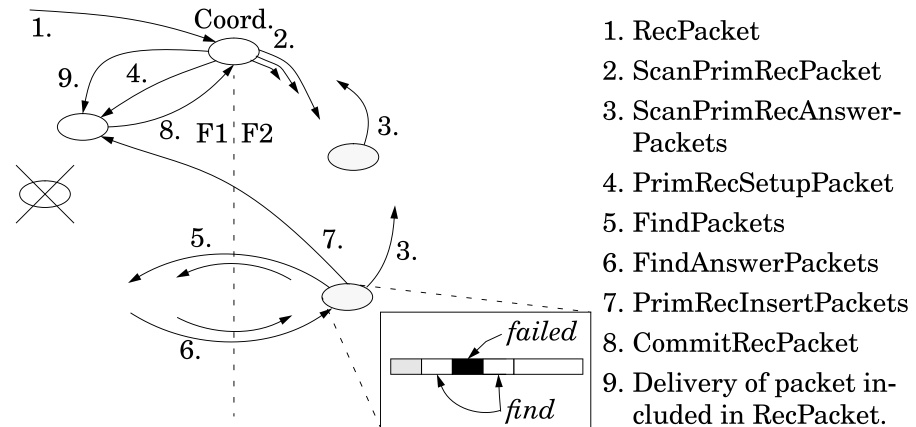


Figure 15. The protocol for recovery of a primary bucket (base case).

The base case implements the algorithm described in part 4.3 “Primary bucket recovery” on page 22. First the Coordinator starts a deterministic scan in F2 to recover the record counter *r* of the failed bucket (2.). Each parity server answers the scan (3.) with its contribution to the record counter and the number of records it will recover (for the new server to know how many records to expect). Then each parity server starts to recreate the records that were on the failed server. In parallel the Coordinator sums the answers and gets the record counter and the number of records that were on the failed server. This information is sent to the new location (4.).

The recovery of a record is done as follows. We know the key of the record to recover. The square in Figure 15 illustrates the storage structure of a parity record (recall Figure 5 on page 22) and the key is marked with “failed”. First find the non-key part of the other keys (marked with “find”) in F1 (5. and 6.). Then combine them with the parity bits to get the non-key part of the failed record. The record has been recovered and is sent in a PrimRecInsertPacket to the new location (7.) and inserted there.

It is not sure that the new location has been setup before the first PrimRecInsertPacket arrives there, i.e. (7.) might arrive before (4.). The protocol accepts the arrival of the packets in any

order, and, when all records have been recovered and the reconstructed record counter has arrived, the recovery process is finished, so a CommitRecPacket is sent to the Coordinator (8.). Now the original processing can continue, so the packet that detected the failure is sent to the new location (9.).

To decrease the number of packets, the last packet (9.) could be included in the previous packet to the new location (4.). This is not done in the implementation, simply because it seemed easier first to focus on the recovery, and then to continue with the processing as normal.

7.7.2 Record recovery

A special recovery case is when a find request can not reach a server. The find can then be answered faster than if the whole bucket first has to be recovered. How this is done is now described.

The Coordinator first checks that the requested record could have been on the failed server, i.e. calculates the correct bucket number and compares it with the number of the failed bucket (actually not done in the implementation, but should have been). If not the same, the record was not on the failed server (and must not be recovered), so the FindPacket can be sent to the correct bucket and processed there. Otherwise the record is reconstructed separately (if it exists) to speed up the find. In both cases the whole server is recovered (as in the base case, but last step left out) as a last step.

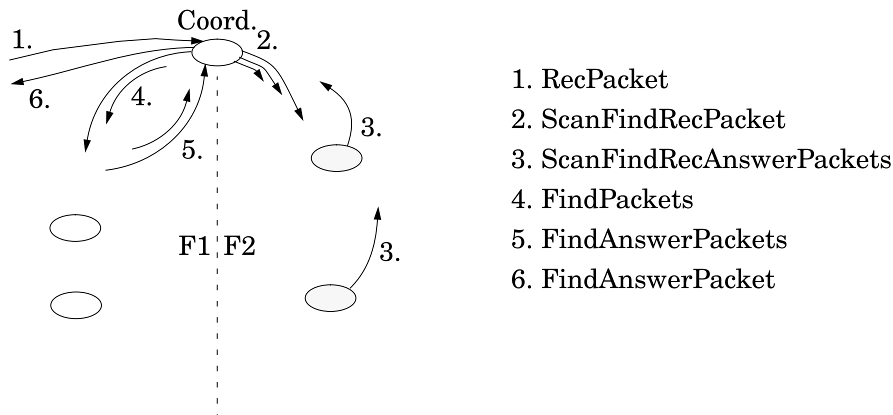


Figure 16. The record recovery process when the record searched for existed.

The implementation of record recovery (described in 4.6 “Record recovery” on page 23) works as shown in Figure 16 and follows the algorithm in (Litwin et al., 1997).

A failure is detected by a FindPacket (1.). The failed bucket is the location where the searched record was stored (if it existed). If the record existed, there exists a parity record containing the key. Therefore a deterministic scan is sent to F2 requesting the parity record containing the searched key. If such a record is not found, step (4.) and (5.) are excluded, and a negative answer can be sent (6). Otherwise the primary record is reconstructed (like in the base case) by finding the non-key part of all the other keys in the parity record and combining them. Then the record is returned in the answer (6.), and, as already mentioned, recovery of the server is started.

7.7.3 Primary bucket recovery – other special cases

As mentioned earlier, recovery has to be done in different ways depending on which packet detected the failure. Now the other (special) cases will be discussed.

During a split (recall Figure 13 on page 39) the splitting bucket or the new bucket might fail. The actions to take both apply to primary and parity recovery.

If the new bucket can not be reached when the Coordinator tries to send the InitNewBucketPacket, the Coordinator just picks another spare server from its (assumed) list of spares.

The bucket to split might get unavailable before the YouSplitPacket arrives. In this case the split is aborted and the bucket is reconstructed as in the base case. The reason to abort the split is that the splitting bucket might have an internal state (depending on the load control policy) which can not be reconstructed.

The last case that has to do with splitting is when the SplitTransPacket can not get through to the new bucket. The Coordinator now picks a new spare server and piggybacks (includes) the InitNewBucketPacket (i.e. only the level of the new bucket) with the SplitTransPacket, so that only one packet (the SplitTransPacket) has to be sent (avoiding a possible race between the two). The extra requirement on an inactive server is that it must be able to receive this kind of SplitTransPackets before initialized.

One case is not yet discussed, and it only applies to primary recovery. The case is when the LevelUpdatePacket (discussed later in 7.9 “Coordinator recovery” on page 46) detects the failure of bucket 1. Then the bucket is recovered as in the base case, but with the last step left out, because in the reconstruction process the level has already been set.

7.7.4 Problems connected with primary bucket recovery

Two problems were discovered with this implementation of primary bucket recovery. The first problem also applies to parity recovery, and arises if recovery is done of the same bucket number on the same physical address within the lifetime of a packet (i.e. the maximum delay a packet can have – might not be known). The scenario is as follows. A bucket at physical address A crashes, gets recovered at B, crashes again, and gets recovered at A. If somebody detected the first crash at A, but the RecPacket reporting it to the Coordinator was (very) delayed, it might not reach the Coordinator until after the bucket has been recovered the second time. The Coordinator thinks that the bucket (at A) has crashed again (according to the delayed RecPacket), but A has not crashed. The result is that recovery is incorrectly started, and we end up with two buckets that are the same.

The easiest solution to this problem is to never recover a bucket on the same physical address within a big enough time limit. The prototype uses this kind of solution. However, this might be a serious limitation, especially if delays can be big.

Another suggestion is to keep a *recovery-counter* at the Coordinator that is increased by one for each failure. If then all physical addresses and all servers are marked with a recovery-number, the Coordinator could discover that a RecPacket is delayed, and therefore not start recovery. The recovery-counter must be recovered in case of Coordinator failure. No extra messages are needed for this, because the recovery-counter can be recovered using the same information as for the PAT recovery (see 7.9 “Coordinator recovery” below). The recovery-number might also be used in a more effective PAT-update algorithm. By leaving out the largest recovery-number in a PAT, it can be determined which entries need to be updated, and thus get a more efficient algorithm. This last suggestion might perhaps be something to look at in future work.

The second problem arose if a split started in the parity file, while primary bucket recovery was in progress. This could happen, because recovery of primary records and a split could proceed concurrently. The problem is that each parity bucket needs to go through all its records (one at a time) and use some for recovery. If a split proceeds concurrently, records not yet handled in the recovery process might be moved away. The solution implemented was that the Coordinator simply did not allow any splits in the parity file during primary bucket recovery. A more elegant way to solve this, would have been to wait with the move until the recovery process is finished, or better, to concurrently move a record when the recovery process does not need it for recovery any more.

7.8 Parity bucket recovery

A failure of a parity bucket is detected during a split or by a Primary Client (recall Figure 7 on page 27) doing an insert. The recovery process is not as complex as in the primary case.

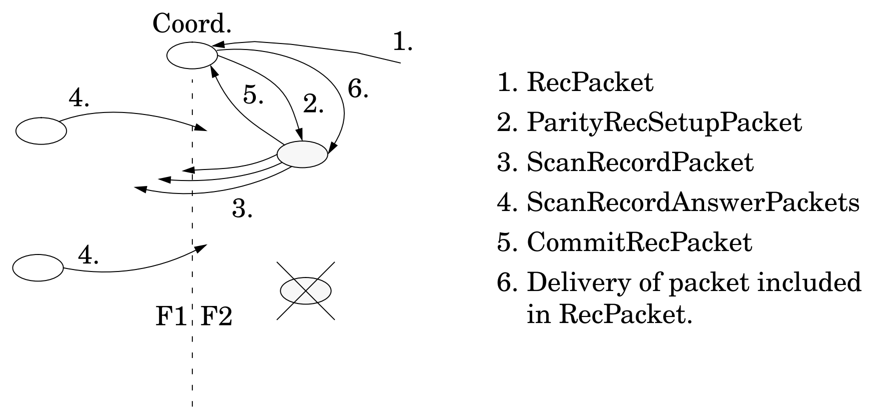


Figure 17. The protocol for recovery of a parity bucket.

In Figure 17 above the protocol for parity bucket recovery is described (see 4.4 “Parity bucket recovery” on page 23 for theory). First a RecPacket (1.) reports the failure (as in the primary case). The Coordinator picks a new location for the bucket, and sends a ParityRecSetupPacket there to setup the server (2.). When the server is initialized, it starts to recover itself by sending a deterministic scan to F1 requiring all the records that were replicated to the failed bucket at the time of the failure (3.). All primary buckets reply (4.) and the bucket can be reconstructed at the new location (by just inserting all the records in the replies). The recovery is now done, so a CommitRecPacket is

sent to the Coordinator (5.), which then delivers the packet that could not get through from the beginning (6.).

As for primary bucket recovery, the last message (6.) could be sent together with the setup packet (2.), which would save one message (not implemented).

The special cases that appear if recovery is needed during split, are handled exactly as for primary recovery (described in section 7.7.3 “Primary bucket recovery – other special cases” on page 43).

Insert of a record *c* into an LH*g file is done as described in 7.3 “Insert/Find” on page 33 – first the record is inserted in F1, then replicated to F2. If the insert into F2 fails and the bucket is recovered, the last step of the algorithm (6.) can be omitted if the failed bucket is the correct target for the insert. This is because the recovery is based on the records in F1, including record *c*, and the record will thus implicitly be inserted into the correct parity bucket during recovery.

7.9 Coordinator recovery

The failure of the Coordinator is of course causing most trouble. Both the file state information and the records in the bucket have to be recovered. All other recovery processes have been initiated by the Coordinator, but now the Coordinator is not available. Therefore bucket 1 of F1 takes the place of the Coordinator during file state recovery. This means that all clients and servers in both F1 and F2 must not only know the correct physical address for the Coordinator, but also for bucket 1 of F1. How this is solved in the implementation will be discussed later in this section, but first the protocol is presented (see Figure 18).

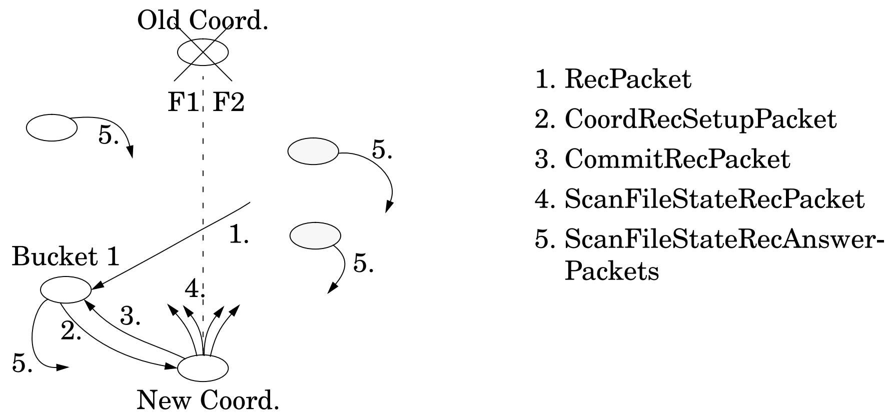


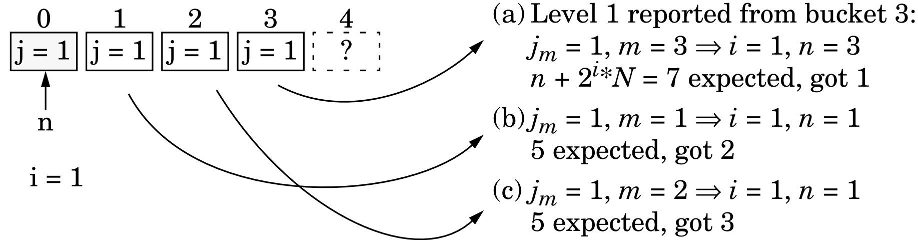
Figure 18. Recovery of the Coordinator.

A (possible) failure is reported to bucket 1 of F1 (1.). A new location is picked for the Coordinator, and a CoordRecSetupPacket is sent there for initialization (2.). The Coordinator sends back a CommitRecPacket (3.), because it is now ready to receive packets, though they are queued until the rest of the recovery has been done. Bucket 1 can now start to forward other packets it receives aimed for the Coordinator to the new location (not shown in the figure).

The Coordinator starts to recover the file state by sending a deterministic scan to all buckets in both F1 and F2. The scan contains the new physical address of the Coordinator, so that everyone gets to know the new address. The scan requests the bucket number and the level, so that the file state can be recovered. The physical address of each bucket is also requested, because the Coordinator needs to recover the PATs. When all answers from F1 and F2 have arrived (5.), the file state and the PATs for both F1 and F2 have been recovered. Now, only recovery of the records in the failed bucket (bucket 0) remains, and this follows the procedure previously described in section 7.7 “Primary bucket recovery” on page 40.

This way of doing file state recovery follows the algorithm described in section 4.5 “Coordinator recovery” on page 23. However, as mentioned in that section, this algorithm does not terminate deterministically. The problem has to do with the deterministic termination of the scan in F1 (recall the example in Figure 4 on page 20). What has not been considered is that it can not be determined definitely that all buckets have replied

until all buckets have, and bucket 0 can not answer (because it is unavailable!). An example of a case when the termination fails is shown in Figure 19 below.



The answer from bucket 0 is needed to determine if there is yet another bucket or not (would decrease expected to 4 and terminate), but bucket 0 is down. In this example there are no more buckets, so the scan does not terminate!

Figure 19. Example of when the scan for file state information in F1 does not terminate.

The problem arises when the level j of all the buckets are the same. Then the algorithm assumes that the file pointer n is $n = 1$; see (c) in Figure 19. The answer from bucket 0 changes this assumption to $n = 0$, which causes the algorithm to terminate. The problem is that bucket 0 can not answer – it is unavailable! Therefore the algorithm waits forever for yet an answer. This case can not be detected and solved easily (by stopping the waiting after a while and setting $n := 0$), because if yet another bucket existed (illustrated with bucket 4 in the example) n should be set to $n := 1$. The conclusion is that the file state can not be determined deterministically in this case.

The solution implemented to this problem is to simulate an answer from bucket 0. If bucket 0 had been available it would have reported in the answer the bucket number, i.e. 0, and the level. Therefore, the level of bucket 0 is replicated in bucket 1, so that whenever bucket 1 gets a ScanFileStateRecPacket, it sends two answers – one for itself and one simulated for bucket 0. In this way the scan for file state recovery always terminates.

The replication requires a LevelUpdatePacket to be sent each time the level of bucket 0 changes, i.e. after each split of bucket 0. For a file with 1 000 000 records, a bucket group size of 20, 75% load and a bucket size of 100, only 10 extra messages are needed, so this is not expensive.

As described, the physical address of the new Coordinator is distributed in the ScanFileStateRecPacket at no extra cost. The usage of bucket 1 as a temporary coordinator when the Coordinator is recovered makes it necessary, as mentioned earlier, for all buckets also to know the correct physical address of bucket 1. The address of bucket 1 only changes after it has been unavailable. When the bucket has been recovered, the new location has to be distributed to all clients and servers. The implementation does this at no additional cost by having all clients and servers listening for broadcast ScanPrimRecPackets. If it is bucket 1 that is being recovered, the physical address included makes all clients and servers aware of the location change.

Using scans to distribute new physical addresses of both the Coordinator and bucket 1 has one problem, though. It might happen that a server or a client for a moment is unaware of the correct location of any of the two. Consider this example. Bucket 1 fails, and is recovered. The ScanPrimRecPacket gets delayed on its way to a parity server. Before it gets there, the Coordinator fails. The parity server can now neither reach bucket 1 (has not got the new physical address yet) nor the Coordinator (unavailable). The parity server therefore has to wait until it gets the ScanPrimRecPacket or the ScanFileStateRecPacket, before the packet can be delivered.

7.10 Forwarding caused by recovery

Because only the PAT of the Coordinator gets updated after the move of a bucket, outdated entries at other servers and clients cause packets to be sent to old addresses. The status of the site at such an address can be:

1. Still unavailable – then a RecPacket is sent to the Coordinator.
2. Available, but spare – assuming that all sites become spares when available again.
3. Available, but another bucket – after a while a new bucket might be located at the site.

In case 2 and 3 a misdirected packet needs to be delivered to the correct address. Case 3 also requires that the server can tell to whom a packet is addressed. Therefore the bucket number of the intended receiver and the physical address of the sender are included in all packets aimed for servers.

The site forwards in these cases the packet to the Coordinator in a ForwardPacket. The Coordinator then decides what actions to take to deliver the packet. The ForwardPacket is processed as follows:

1. The Coordinator checks that the forwarded packet was not aimed for itself. If so, the packet is processed as normal.
2. If the forwarded packet has with splits to do, i.e. InitNew-BucketPacket, YouSplitPacket or SplitTransPacket, the same actions as if a failure had been reported are taken (see 7.7.3 "Primary bucket recovery – other special cases" on page 43).
3. Otherwise the packet is delivered to the correct address, and a PATUpdateAnswerPacket is sent back to the sender, updating its PAT.

A special case has to be handled, and it occurs if a server goes down, and then quickly comes up again (now a spare) before the failure has been detected (and the bucket recovered). Then a packet might arrive aimed for the failed bucket, but according to the rules above, the packet is forwarded to the Coordinator. It delivers the packet to the correct address, i.e. the same address as the packet just came from. The spare server gets the packet again, and there is a loop. Such a loop is detected by the Coordinator checking that the address the packet came from is not the same as the address to deliver to. When a loop is detected, recovery must be started, because there has been a failure that nobody has detected.

8 Implementation

In this chapter some implementation specific details will be discussed, starting with the implementation language and how sites and communication are implemented. Problems and some not implemented parts are then analysed.

8.1 The implementation language – Java

The implementation was made in the new programming language Java from Sun Microsystems. It is an object-oriented, distributed, interpreted, robust, secure, architecture neutral, portable, high-performance, multithreaded and dynamic language, Sun claims (Gosling et al., 1996). The most important properties considered are the object-orientation, distribution and the multithreaded possibilities. The object-orientation is a paradigm that suites the application area very well, because there are natural objects that need to communicate. The distribution consists of the high-level language support for network communication, which in future work is preferable for extension of the prototype to a real distributed system. The last and most important property is the language support for threads, which is very easy and straightforward to use.

A significant limitation is of course the interpretation. This makes the prototype run a bit slow. Another limitation is that threads with the same priority are not scheduled in a pre-emptive way on the Solaris platform (where the development was made). True pre-emption would have given a more realistic prototype, but on the other hand caused more synchronization problems. The fact that the scheduling is deterministic on Solaris is good for simulations, because it makes it possible to rerun the program and always get the same result. Tests on the Windows 95 platform (where Java is pre-emptive) showed at least no catastrophic behaviour of the prototype, but the prototype must be enhanced to work properly in such an environment. The advantages outweighs these limitations by far, though.

8.2 Sites and communication

In the implementation each client and server runs in its own thread. The principle is that each thread reads a packet from the input buffer, processes it and then continues with the next

packet. If no packet is available, the thread pauses until there is.

To communicate with other sites (threads), a packet is created. If the physical address is not known (only the bucket number is), it is looked up in the PAT. This is done in the CommonLayer (see 6.2 “Clients and servers” on page 28). The Communicator then uses the Global Address Table to find the mapping between the physical address (used to simulate reality) and the corresponding thread. The packet is then encoded to bytes and transferred via a pipe to the buffer of the target thread. The packet is recreated by the Communicator of the target thread, and then finally delivered to the upper layers for processing.

Failure of a site is detected by the Communicator when sending a packet, and is reported to the upper layer, which then takes the appropriate actions.

The limitation mentioned in 1.3 “Delimitations” on page 11 that packets may not disappear is solved by checking that the thread to stop (for unavailability simulation) does not have any packets queued that can disappear. This is not very realistic, but was needed to limit the work on the prototype.

The way the implementation was made makes it fairly easy to enhance the single-computer prototype to a full scale prototype. The Communicator has then to be rewritten (using the high-level support that Java has built in), and handling of spare servers has to be implemented. These are the only bigger changes that need to be done.

8.3 Implementation problems

A difficulty with threads is that the ability of debugging is limited. Big trace files had to be analysed to find out why the protocol did not always work. This is both time-consuming and in many cases difficult. Some tool for this had been very useful.

The implementation of the communication caused some problems with deadlocks. If two threads tried to send each other packets while both their buffers were full, a deadlock occurred. The problem is that sending a packet blocks the thread until the whole packet is transferred. The solution in some cases was to start a new thread for some recovery processes, but in some cases no real solution was implemented. Instead the buffer size was

increased, but this only decreases the probability for deadlock. This problem should be further analysed.

For the parity file the (group) key g is composed of the bucket group $g1$, and the record counter r. To calculate the bucket address in F2 where the record should be sent, a hash function $h_l(c_g) = c_g \bmod (2^l N)$ is used. But the group key g has got two components, $g1$ and r. This turned out to be an issue.

At first the sum of the two, i.e. $g1 + r$ was used as the number c_g to hash. This turned out to create a very bad distribution of records in the parity file. It was then noticed that if the distribution of primary keys in the primary file is perfect, the group keys g of the parity file are deterministically determined (see Figure 20 below).

	(0, 0)							
	(0, 1)							
	(0, 2)							
split	(0, 3)							
	(0, 4)	(1, 0)						
	(0, 5)	(1, 1)						
	(0, 6)	(1, 2)						
split	(0, 7)	(1, 3)						
	(0, 8)	(1, 4)	(2, 0)	(3, 0)				
	(0, 9)	(1, 5)	(2, 1)	(3, 1)				
	(0, 10)	(1, 6)	(2, 2)	(3, 2)				
split	(0, 11)	(1, 7)	(2, 3)	(3, 3)				
	(0, 12)	(1, 8)	(2, 4)	(3, 4)	(4, 0)	(5, 0)	(6, 0)	(7, 0)
	(0, 13)	(1, 9)	(2, 5)	(3, 5)	(4, 1)	(5, 1)	(6, 1)	(7, 1)
	(0, 14)	(1, 10)	(2, 6)	(3, 6)	(4, 2)	(5, 2)	(6, 2)	(7, 2)
	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮

Figure 20. The deterministic distribution of the group keys if perfect distribution in the primary file; bucket size $b = 4$. Reading the keys from top, left to right, gives the order in which the keys are inserted in the parity file.

Another way to calculate c_g, that performed well on random strings, was picked (called X33; $c_g := g1 + g1 * 2^5 + r$) from (Pettersson, 1993), and the distribution properties was tested and considered fair. The choice of how to combine $g1$ and r in the

best way was not further investigated, and should perhaps be addressed in future work.

8.4 Not implemented

There are some issues that were not considered or implemented in the prototype, and that have not already been mentioned.

The behaviour of a server when it restarts has not been considered. Perhaps the server should tell the Coordinator that it is back, so that the server can keep track of spares and possibly start recovery. The restarted server might not have the address to the Coordinator – how should this be handled? The prototype just creates a new thread if it needs another server. In reality the Coordinator must perhaps have a list of known spare servers, and also be able to reconstruct the list, or is there another solution?

What happens if more than one server goes down? Some of the records can be reconstructed, but how many? These are also unanswered questions.

Probabilistic termination of scans was not implemented, and not deletion of records either.

Future work should also address the possibilities for transaction handling. This is necessary for use in many applications.

9 Performance Measures

Measurements have been made on the thread-based prototype, which will be presented in this chapter. First the setup for the simulation is described, then the measurements are presented in diagrams.

9.1 Simulation setup

The tests have been focused on how the number of messages needed for one insert (efficiency) depend on the failure frequency. The tests have been made by letting four (application) clients insert 2,500 random records each. Meanwhile the Bomber is activated at variable frequency (failure frequency). Each time the Bomber is called, there is a constant probability for a server crash. This construction makes a file with many buckets crash more often than a file with few buckets, which seems realistic. A failure frequency of zero means zero failures, while a frequency of 0.2 corresponds to 45-80 failures during 10,000 inserts.

Each setup was run five times for each variation in failure frequency, to achieve a statistically stable result.

9.2 The first measurement

The first diagram (Figure 21) shows the efficiency for different bucket sizes. The bucket group size is kept constant at 10, and F2 has got 10 initial buckets. For low failure frequency (0.00625) a big bucket size is more efficient, because there are fewer splits, hence fewer buckets. Because of this, the probability for a failure in the file also decreases. As the failure frequency grows bigger, the efficiency of large buckets gets worse than that of smaller ones. This is because it costs more to recover a big bucket (more records to recover) than a small, and this finally outweighs the advantage of fewer buckets.

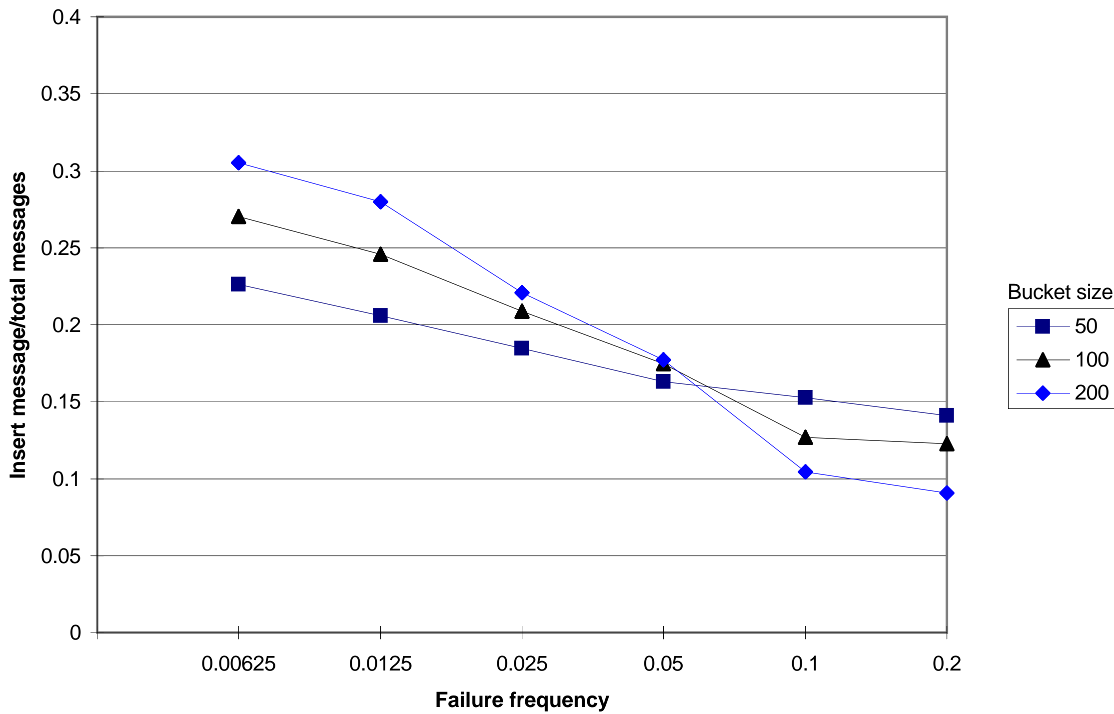


Figure 21. The first measurement. Efficiency with 10 initial buckets in F1 (i.e. bucket group size k = 10) and also 10 initial buckets in F2 for different bucket sizes.

9.3 The second measurement

It was then decided to investigate how the efficiency varies for different bucket group sizes. So, the bucket size was held constant at 100 and the initial amount of buckets for F1 and F2 was varied. The result is shown in Figure 22.

When no failures occur, it is more efficient with bigger bucket group sizes, because there are fewer records to store in the parity file, hence fewer buckets. For failure frequency = 0 it can be noticed that the efficiency for bucket group size k = 20 is smaller than for k = 10. This is due to that there is unnecessary storage space in F2 that costs messages. A small calculation on the average records per bucket in F2 shows that the buckets are only filled with 25 records (10,000 / 20 / 20 = 25). It can be concluded that about 5 buckets would be the best choice, and this would

decrease the number of buckets the parity clients need to keep track of, hence increase efficiency.

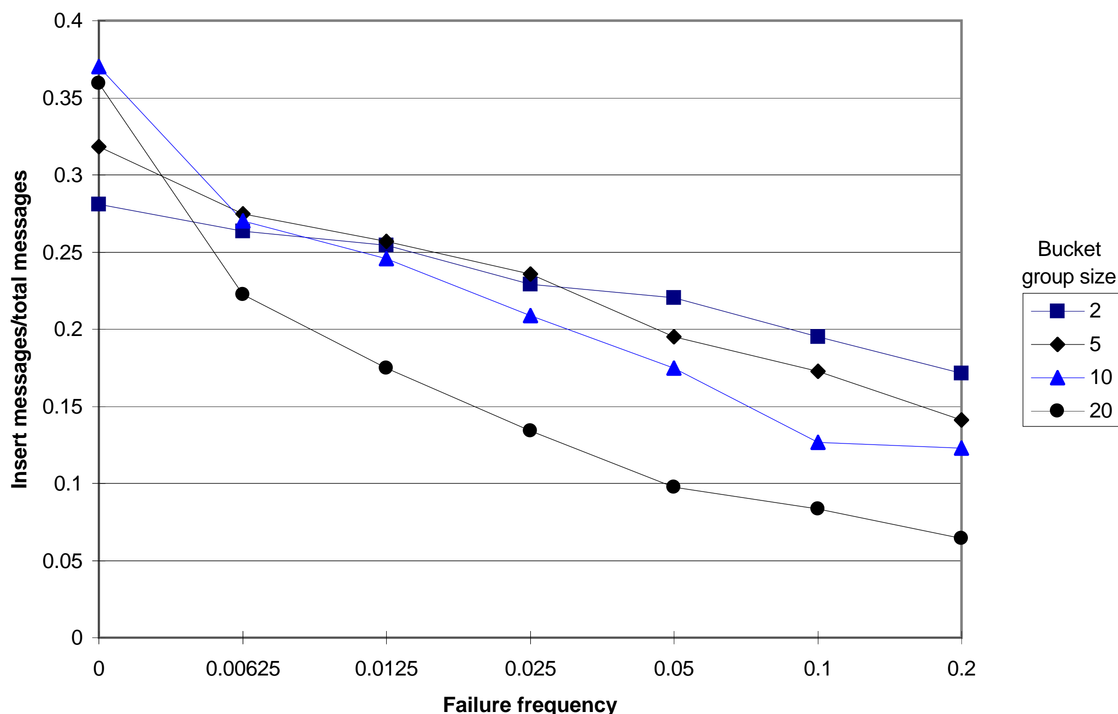


Figure 22. The second measurement. Efficiency with variable number of initial buckets and constant bucket size of 100.

The result of this analysis is that the number of initial buckets for F2 always should be one – to avoid unnecessary overhead. This makes sense, because there are always only $1/k$ as many records in F2 as in F1, so k times fewer buckets are needed, which always means one. This should cause the file level of F1 and F2 to be about the same. The possible disadvantage of this could be that both F1 and F2 splits at about the same time, and must be further investigated.

If we return to Figure 22, the increased recovery cost for bigger bucket group sizes can be observed as the difference between the first and the second measure point. Bucket group size $k = 2$ hardly changes, while efficiency for $k = 20$ is reduced with (more than) 40%. When the failure frequency increases, $k = 2$ (almost mirroring) causes the least messages to be sent, due to cheaper recovery, even though the probability for failure increases (because there are more buckets that can fail in F2). Bucket group

size 20 costs most messages, but on the other hand 10 times less storage space is needed.

It should however be considered that the impact of splits on these measurements are very low. This is because all records are moved in one packet (the SplitTransPacket), making a split cost only five messages (recall 7.6 “Split” on page 36). To use only one message to move the records might not be possible or optimal, as mentioned earlier, so this should be further investigated.

9.4 The third measurement

The third and final measurement was made with one initial bucket for F2, bucket size 50 for both files, and then the bucket group size for F1 was varied.

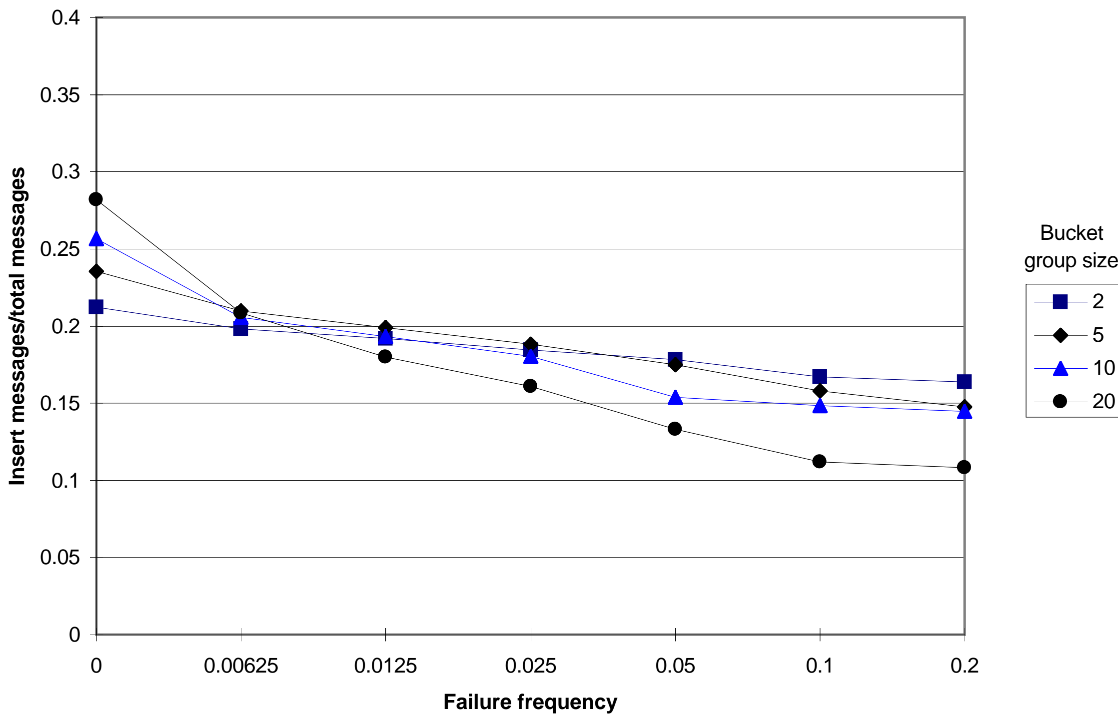


Figure 23. The third measurement. Efficiency with variable bucket group size and constant bucket size of 50.

The principal look of this diagram (Figure 23) is the same as for Figure 22. Bigger bucket group size is more efficient for low failure frequency, while bucket group size 2 (almost mirroring) is better when failures are frequent. This is the general conclusion that can be drawn. The breakpoint between $k = 2$ and $k = 20$, is somewhere between failure frequency 0.00625 and 0.0125. This

corresponds to about 15 failures (of about 350 servers – 4%) during the 10,000 inserts, and must in reality be seen as fairly high. It can be concluded from this that for less faulty environments a bucket group size of k = 20 works fine.

The difference between the curves are smaller than in Figure 22, and this depends on that recovery in general becomes twice as expensive when the bucket size doubles.

10 Conclusion and Future Work

The theory for LH*g has proven to be correct and implementable in all cases but two – the image adjustment algorithm was not considering the initial number of buckets, and the file state recovery algorithm did not always terminate. Solutions to both cases were presented.

An attractive architecture has been defined for LH*g. Further, the design of the system has been defined, including the protocol for message passing. The implementation of a fairly realistic prototype revealed a number of problems. Some solutions have been implemented, e.g. the enhanced split handling protocol, the adjustments needed to get file state recovery to work, and the special handling needed if primary or parity bucket unavailability was detected during a split. Other possible solutions to detected problems have been discussed, e.g. what is needed to be able to reuse a physical address (server), and how to handle multiple scan answers. These are things to be handled in future work.

Some limitations in the implementation language Java was found in the scheduling of threads. The limited debugging possibilities also caused some difficulties. On the other hand, Java has high-level language support for network communication, making the prototype very attractive for extension to a real distributed system.

Measurements on the prototype showed to be in accordance with the theory, so the prototype can be considered fairly realistic. The conclusion drawn was that lower degrees of redundancy, i.e. a group size k of 20, work well if the failure frequency is low.

Some problems and thoughts remain to be addressed in future work. For example how to distribute the physical addresses in a more efficient way, how to split more efficiently, and how the hash-value for the parity key should be calculated. The behaviour of LH*g when more than one bucket fails at the same time should also be investigated. For use in real applications the possibilities for transaction handling must be examined. Finally, a real implementation must be made, e.g. by extending the prototype and use TCP/IP.

11 References

Devine, R. (1993), *Design and implementation of DDH: A distributed dynamic hashing algorithm*, Proceedings of the 4th International Conference on Foundations of Data Organization and Algorithms (FODO).

Gosling, J. and McGilton, H. (1996), *The Java Language Environment – A White Paper*, http://java.sun.com/docs/white/langenv/.

Karlsson, J. S. (1997), *A Scalable Data Structure for a Parallel Data Server*, Licentiate Thesis No 609, LiU-Tek-Lic 1997:10, University of Linköping.

Karlsson, J. S., Litwin, W. and Risch, T. (1995), *LH*LH: A Scalable High Performance Data Structure for Switched Multicomputers*, Research report, University of Linköping.

Lewis, T. G. and El-Rewini, H. (1992), *Introduction to Parallel Computing*, Prentice Hall, New Jersey.

Litwin, W. (1980), *Linear hashing: A new tool for file and table addressing*, In Proceedings of VLDB, Montreal.

Litwin, W., Neimat, M.-A. and Schneider, D. A. (1993), *LH* – Linear Hashing for Distributed Files*, ACM-SIGMOD Int. Conf. On Management of Data.

Litwin, W., Neimat, M.-A. and Schneider, D. A. (1994), *RP* : A Family of Order-Preserving Scalable Distributed Data Structures*, 20th Intl. Conf. on Very Large Data Bases (VLDB).

Litwin, W., Neimat, M.-A. and Schneider, D. A. (1996), *LH* – A Scalable, Distributed Data Structure*, ACM Transactions on Database Systems, Vol. 21, No. 4, pp 480-525.

Litwin, W. and Risch, T. (1997), *LH*g: a High-availability Scalable Distributed Data Structure by Record Grouping*, Research report, University of Linköping and University of Dauphine.

Özsu, M. T. and Valduriez, P. (1991), *Principles of Distributed Database Systems*, Prentice Hall, New Jersey.

Pettersson, M. (1993), *Main-Memory Linear Hashing – Some Enhancements of Larson's Algorithm*, Research report, University of Linköping.

Tanenbaum, A. S. (1995), *Distributed Operating Systems*, Prentice Hall, New Jersey.

Torbjørnsen, O. (1995), *Multi-site Declustering Strategies for Very High Database Service Availability*, Thesis Norges Techn. Hogskoule, IDT Report 1995.2, 176.

Avdelning, Institution
Division, department

Institutionen för datavetenskap

Department of Computer and
Information Science

Datum
Date

1997-11-09

Språk
Language

- [ ] Svenska/Swedish
- [x] Engelska/English
- [ ] ____________

Rapporttyp
Report category

- [ ] Licentiatavhandling
- [x] Examensarbete
- [ ] C-uppsats
- [ ] D-uppsats
- [ ] Övrig rapport
- [ ] ____________

URL för elektronisk version

ISBN

ISRN

Serietitel och serienummer
Title of series, numbering

ISSN

LiTH-IDA-Ex- 97/65

Titel
Title

A Java Implementation of a Highly Available, Scalable and Distributed Data Structure, LH*g

Författare
Author

Ronny Lindberg

Sammanfattning
Abstract

Distributed systems use parallelism and can offer improved performance and scalability. New data structures for distributed systems, that can take advantage of this, are needed. Some applications also require high availability. LH*g is a high availability variant of the hash-based LH* Scalable Distributed Data Structure. It scales up with constant key search and insert performance, while surviving any single-site unavailability (failure). The results of building a thread-based prototype of LH*g in Java is documented in this report. This includes architecture, design and the protocol for message passing. A number of problems were identified and alternative solutions were in most cases proposed, implemented and tested. Measurements on the prototype are reported and discussed.

Nyckelord
Keywords

Distributed Storage Structure, Scalability, Linear Hashing, Java, High Availability, Distributed Systems

95-10-25/lisli
IdaMallar