ABSTRACT

Peer-to-peer technology is gaining grounds in many different application areas. This report describes the task of building a simple *distributed and replicated database system* on top of the distributed hash table *Chord* and the database application *Berkeley DB Java Edition* (JE). The prototype was implemented to support a limited subset of the commands available in JE.

These were the main challenges of realizing the prototype; *(1)* integration of the application level communication with the Chord level communication *(2)* design and implement a set of data maintenance protocols required to handle node joins and failures *(3)* run tests to verify correct operation *(4)* to quantify basic performance metrics for our local area test setup.

The performance of the prototype is acceptable, taken into consideration that network access is taking place and that it has not been optimized. There are challenges and features to support: *(a)* although Chord handles churn reasonably well, the application layer does not in the current implementation *(b)* operations that need to access all records in a specific database are not supported *(c)* an effective finger table is required in large networks

The current approach seems to be well suited for relatively stable networks, but the need to relocate and otherwise maintain data requires more complex protocols in a system with high churn.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Large distributed systems must be able to scale to handle increases of load and/or data. A solution is to add more nodes to the network. Traditional systems based on a consistent global view can stop working satisfactory with a few dozen participating nodes, or less with certain protocols. *Peer-to-peer technology* may be a mean to achieve a higher degree of scalability for distributed systems, as they are able to handle a much higher number of nodes. Node additions further result in highly isolated system changes, which make it easier to support a high degree of availability.

We have implemented a peer-to-peer based distributed database system built on top of the distributed hash table *Chord* and the database application *Berkeley DB Java Edition*. The primary goals were to implement a simple prototype that could be used for testing and to discover the main challenges raised by the switch to peer-to-peer technology.

A description of the problem and the goals set for this project are given in Chapter 2. Background information on Chord and Berkeley DB may be found in Chapter 3, followed by a brief survey on active areas of research on topics of relevance in Chapter 4. The design and implementation of the prototype are presented in Chapter 5 and 6. Tests of the prototype, and discussions on their results, are documented in Chapter 7, while Chapter 8 concludes the work and gives pointers to further work.

Chapter 2

Problem description and goals

The main goal of the project is to *develop a working prototype of a peer-to-peer database system, with focus on challenges related directly to peer-to-peer technology.* Originally, the existing codebase from a previous masters thesis project were to be used, but we soon realized it was not suitable. The switch from the implementation of the traditional client-server model to a peer-to-peer model was too dramatic. A majority of the old code was rendered useless, and so we decided to implement a new prototype from scratch. These subgoals, or demands, were set for the project:

- Source code to be written in Java.
- Achieve *scalability* through data distribution.
- Achieve high *availability* through data replication.
- Use a distributed hash table (DHT) for routing, and to minimize the need for global state knowledge.
- Determine the cost of a node join in our peer-to-peer system.

The original problem description for the project is included in Appendix A.

2.1 System limitations

As implementing a full-blown database system requires very much work, we had to set some limitations on our system. The listed problems are left out because we were able to reach our goals without taking them into consideration. However, if the system is to be of use in a situation similar to a production environment, they would have to be in place. Note that the cause for the limitations are the lack of developer resources and time. We have not assessed the complexity regarding implementing them in our prototype system. It might be straight-forward, or it might require the system to be radically extended in some way.

- **Security**
  Both security in the traditional sense and security related to malicious peers are ignored.
- **Transactions**
  Support for transactions is not implemented, as it poses little interest with regards to the goals we have set for the project.
- **Distributed cursors**
  Although important for system usage, we found no time for this, as it most certainly requires auxiliary functionality to be implemented.
- **Handling of large data volumes**
  The system will not claim to be able to handle large data volumes. All processing is done fully in memory. If the virtual machine is unable to handle the data volume automatically, so is our system.
- **Assumption of light-weight nodes**
  We assume that the nodes in the system each hold a relative small data volume. The reason is that we want to ignore situations were, for instance, data migration would take hours to complete due to limited network transfer capacity. Such cases require special synchronization protocols to provide acceptable response times and system liveness.

Chapter 3

Background on DHTs and Berkeley DB

This chapter briefly introduces distributed hash tables (DHTs) and Berkeley DB. The former is a general abstraction, the latter is an open-source database system. A basic degree of comprehension of them both is required to fully appreciate this report. The choice of using Berkeley DB as the underlying database system was dictated by the project task description.

3.1 Distributed Hash Tables

The goal of a distributed hash table (DHT) is to provide a distributive mean of looking up a value when given a key. The different parts of the DHT are stored on the nodes in a distributed system. A client requesting a lookup should need to know about only one system node. The system nodes must not be required to know about all other nodes in the system, as this greatly impedes the scalability of the system. A DHT helps solve this problem by functioning as a routing method, or making use of such a routing component. Each node only needs to have knowledge of a small subset of the other nodes, and is always capable of routing the request to a node that is nearer the target than itself. Eventually, the request will arrive at its destination node and can be fulfilled. Note that a DHT in itself is not an application the end user can make use of. An end-user application has to be built on top of the DHT, offering the services for the users. The DHT itself can either be integrated into the application, or function as a common service offered to several applications. The latter approach might be used when the different applications need the same data, but operate on it in different ways, or if several applications cooperate to maintain the same data.

3.1.1 Core DHT functionality

The core functionality of a DHT is to fetch data based on a key. Unlike a traditional hash table, a DHT is able to distribute its content over a set of nodes connected by network links. This way, the capacity is increased and the workload of each node is reduced. The DHT abstraction defines a simple store and fetch functionality.

A central component of a DHT is the hash algorithm. It is used to generate keys of *fixed length* of data items that are inserted into the DHT. In systems that require the DHT to be very fast, the hash algorithm must be very fast as well. It must also be applicable to the whole set of data items that might be inserted into the DHT. Furthermore, for (sub)optimal performance, it must strive to distribute the content evenly over the nodes in the system. The space of identifiers must be large compared to the number of data items to reduce the chance of two items getting the same hash. The hash algorithm also need to be able to handle changes in the underlying distribution structure. *Consistent hashing* [KLL$^+$97] has proven to be well suitable for use in DHTs. The main reason is that the clients relying on the hash for some reason, do not need to have the same view of the underlying distribution structure. In this context, this structure is the nodes making up the DHT. Further, when the structure changes, the major part of prior decisions based on the hash is still valid. Currently, the hash algorithms typically used are SHA-1 ([fip02] and [Sch05]) and MD5 ([Riv92]).

The key space in a DHT is flat, and the key has no meaning or context. This enables DHTs to be used in very different types of applications.

3.1.2 DHT interface and related abstractions

What makes a DHT even more appealing, is its simple interface. According to [BKK$^+$03], a DHT has only one operation: *lookup(key)*. This will return the identifier of the node handling the item with the specified key, along with the information needed to communicate with it. An application might use the lookup-operation to get hold of the node supposed to store an item, then send the data along with a store request to this node afterwards. The interface of a DHT can also consist of the two operations *get(key)* and *put(key, value)* (or *put(value)* depending on usage). Pastry[1] [RD01] exports *nodeID = pastryInit(Credentials, Application)* and *route(msg,key)*. Additionally it requires the application to implement the methods *deliver(msg,key)*, *forward(msg,key,nextId)* and *newLeafs(leafSet)*. It is evident that the DHT abstraction gives way for a simple interface for applications to work with.

The DHT abstraction has some related abstractions; *decentralized object location and routing* (DOLR) and *group anycast and multicast* (CAST). DOLR functions much as a DHT, but additionally incorporates the locality property and make extensive use

[1]Pastry is actually a DOLR, not a DHT. Consult the next paragraph.

of caching. It can be described as a decentralized dictionary service. CAST provides group communication and coordination. The groups are represented as trees, and the trees are distributed. Depending on the underlying overlay network[2], CAST can also utilize locality.

Work has been done in [DZDS03] to define a common API[3] for these abstractions and the underlying overlay networks. The result has been named *key-based routing* API (KBR), and represents common capabilities found in the different abstractions. According to the authors of the paper, work still has to be done to provide application programmers with a set of common services to use in their applications.

There seems to be some confusion in published papers regarding the terms DHT, DOLR and lookup service. For instance, Chord has been described as both a DHT and a lookup service. Further, Tapestry has been described as both a DHT and a DOLR. Throughout this report, the term *DHT* is used in a more general way, *incorporating the terms lookup service and DOLR*. CAST is not discussed further in this report.

3.1.3 The overlay network

The nodes in the system communicate through a logical overlay network created over the lower level communication links. There are several approaches to organize the network; random mesh, tiered structure or ordered lattices [Ver04]. The goal is to reduce the number of links to maintain and at the same time provide enough redundant links to enable all nodes in the network to communicate in cases where some nodes go down. Creating a link requires knowledge about the node to connect to, so routing information at the node is proportional to the number of (outgoing) links. Aspects as geographical proximity and link speed are important related to response times, transfer speed and stability. The more ordered and advanced the overlay network topology is, the more complex it is to maintain it in a dynamic system with high churn. It is important to remember that a link between two neighbors in the overlay network might actually go halfway around the world in the lower level physical network. Without knowledge about the organization of the overlay network, one can not draw any conclusions related to locality based on the overlay network structure.

Often the overlay network is to function over the Internet. This requires the system to take firewalls, translation systems and other network devices into account. If not, communication will be hindered. There are no generally applicable mean that solves all problems here, but tunneling and use of proxies overcome many of them.

One of the challenges that has not yet been completely solved, is that of *discovery*. How will new nodes obtain connectivity information about the existing nodes in the system? The approaches used so far are *static configuration*, using a *central directory* and using

[2]A logical network built on top of the lower level communication links, see chapter 3.1.3
[3]Application Programming Interface

the *domain name system service* (DNS). The first approach is quickly rendered useless as the number of nodes grows. The second turns out to be a single point of failure and may get capacity problems, while the third requires access and privileges to configure DNS servers. See [Ver04] for more details. The ideal discovery mechanism would enable the software to join the network automatically without user intervention, and at the same time utilize network resources optimally. In practice, one of the approaches mentioned is used in conjunction with an *external communication channel* for getting the information required to join the network. As soon as a node is connected to a DHT network, it will receive contact information about the other nodes it is required to know of. A consequence of this, is that a node need only to be configured with the address of a single existing node to connect to the network.

3.1.4 Routing in the overlay network

Several schemes have been developed to achieve fast and efficient routing in the overlay network. Common for as good as all of them, is that routing can be made faster by increasing the information about other nodes at each node. Then the routing path is shortened. However, by doing this, the space requirement increases and the work needed to maintain the routing information becomes more demanding. As a consequence, the schemes try to find a balance between amount of routing information and routing path length.

3.2 Berkeley DB

Berkeley DB (BDB) is "an open source embedded database library that provides scalable, high-performance, transaction-protected data management services to applications" [Sle04]. As one can see, BDB is an embedded database library. It was built to be embedded into an application and then run on the same computer as the application. No inter-process communication takes place, as the library and the application use the same address space. It is very important to note that BDB is not a database system comparable to well known systems like MySQL, Oracle or PostgreSQL. BDB is not a SQL[4] database.

BDB is able to store <*key, value*> pairs efficiently. The data is stored flat, and there is no notion of structure[5] like the relational database schema provides. The data itself can be stored in a number of *physical structures* however; *hash tables*, *Btrees*, *record-number-based storage* and *queues*. The data operations available are *get*, *put*

[4]Structured Query Language: a language to query and manipulate databases, capable of performing complex operations and on-the-fly data functions (aggregation, grouping, minimum, maximum and more).

[5]A number of data values can be stored under a single key in relational systems (*columns*).

and *del*. The keys can be searched/browsed by using *cursors*. BDB uses the concepts *environment* and *database*. These can be compared to the relational concepts database and table. Common resources and services for one or more databases are encapsulated by an environment, i.e. logs, transaction handling and shared memory areas. BDB databases are stored in regular files and one file can contain several databases.

Beside the *data access services* BDB also offer *data management services* like locking, concurrency, transactions and recovery. These services can be enabled or disabled as the application developer sees fit.

BDB is designed to provide application developers with industrial-strength database services. It gives the developer the choice to use the features he/she requires, it is even possible to use the subsystems separately. BDB is also designed to interact correctly with the operating system's native set of tools to ease administration and development. Last, many scripting language interfaces are available, allowing the developer to use the language he/she is most familiar with.

There are two approaches to use Berkeley DB with Java: Use the Java wrapping interface to the original Berkeley DB or use the pure Java version of Berkeley DB (JE). The original Berkeley DB and JE are two different products. They are very similar, but have a number of important and not-so-important differences. For instance, JE only supports the Btree physical structure. Both could have been used in our prototype. We chose to use the pure Java version, since the prototype itself is implemented in Java. The pure Java version is also faster than using the wrapping interface, as there is no type conversion between different programming languages involved. A drawback is that JE is much less mature than the original BDB.

Chapter 4

State of the art

Is peer-to-peer computing ready to take over for the traditional applications we use today? Probably not, but a lot of work is done to improve existing technology and to develop new and better solutions. Below we give a superficial survey of interesting topics currently being worked on. Although software applications do exist, they are mostly research systems. It is a long way to go before they can be used as commercial applications.

4.1 Distributed hash tables

During the depth study described in [Waa04], most of the available distributed hash table (DHT) designs were briefly surveyed. Now, over half a year later, no new major DHT has emerged. A lot of work has been done on the existing ones, especially the ability to handle churn. Many of the DHTs assume valid sequences of joins and leaves, but this assumption does not hold in real-world applications. Further, how much resources must be used to maintain the correctness of the system state. [BKLN02] is a theoretical analysis of this problem, with focus on the Chord DHT. Another research area is how to execute distributed queries in DHTs and related systems. This is very interesting, because a full-blown distributed database system based on peer-to-peer technology must support a rich query language (like SQL). If we extend our view to include peers with different data schemas, this has also been studied. *Piazza* ([pia]) utilizes *semantic mappings* to allow peers with different schemas to exchange data. The basic idea is that transitive relationships between schemas will enable any two peers to exchange data. The *Hyperion* project ([hyp]) takes a broader view and investigate the data management issues raised by peers needing to share data in the peer-to-peer paradigm. The project homepage state that the main goals of the project is to work out a precise definition of a peer-to-peer data management architecture and to develop efficient algorithms for search, retrieval and exchange of data.

The *P-tree* has been proposed as a solution to effective queries, as it supports both equality and range queries. It has been published as part of the *PEPPER* project at Cornell University. Papers regarding PEPPER can be found at [pep]. Another project studying queries in large networks of distributed data is *PIER*, which is described in [HCH$^+$05]. This software has been implemented in Java and the project itself has been work on for over three years. According to the paper, PIER is "an Internet-scale query engine" and "the first general-purpose relational query processor targeted at a peer-to-peer (p2p) architecture of thousands or millions of participating nodes on the Internet". More papers and information about PIER can be found on the project website ([pie]).

New overlay structures are being tested out, as the overlay can greatly affect the properties of a peer-to-peer system. Structures used in other contexts are also tried out, for instance butterfly graphs (basic idea described in [MNR02]) and de Bruijn networks (evaluated in [DGA04]).

It seems that Pastry ([RD01]) is still the most feature-rich and advanced core DHT implemented. The development activity is still quite high for this DHT. There is also activity on Chord, but it is less feature-rich than Pastry. Due to its simplicity, Chord is a good subject for testing out new things and doing analysis on. *Bamboo* ([bam] and [RGRK04]) is a DHT that is being actively worked on, and it is also continuously run on the PlanetLab ([PAR02], [BBC$^+$04], [pla]) system. PlanetLab is a large network of computers located around the world, and the Bamboo test system is open for public access. Bamboo is designed in such a way that components of the DHT can be replaced. It is capable of handling a higher churn rate then most other DHTs and has also been tested in real-world applications.

Other projects dealing with adjoining topics to our own (distributed database system using a DHT) include *P-Grid* ([pgr]), *The Stanford Peers* project ([sta]) and *BEST-PEER* ([bes]). All these project sites houses a number of papers on different aspects of peer-to-peer computing, many of which are published recently.

4.2 Distributed databases

One of the definitions of a distributed database is "a database that consists of two or more data files located at different sites on a computer network" ([web]). Most existing database systems are based on the traditional client-server model, and do not pose any interest with regards to peer-to-peer computing. These systems are very static with respect to nodes, and they often operate with periodically batch updates to maintain a degree of consistency. In addition to the batch updates, a controller is used to resolve/manage transactions that would result in inconsistent results. The main goal of a distributed database management system is to make the data distribution transparent to the users; present several databases as one. An example system supporting distributed databases is the database product range of *Oracle* (visit [ora] for more information).

When looking at peer-to-peer technology, there are lots of file sharing applications. However, these are made for sharing content that does not change after it has been published. Besides of the well known applications like Napster and Gnutella, a number of file systems based on peer-to-peer technology have been implemented. They are not within the scope of this project, but a list of systems is presented in Appendix C.

As it turns out, there are not many existing systems that resemble our prototype. One of the reasons is that a great deal of peer-to-peer technology is quite new. It is not thoroughly tested, and active research is still going on. Large scale peer-to-peer applications are also very expensive and difficult to test, due to both the fact that it is a distributed system and the high number of hosts involved. Systems exist for testing large peer-to-peer applications, for instance *PlanetLab* ([PAR02], [BBC^{+}04], [pla]). PlanetLab describes itself as ”an open platform for developing, deploying, and accessing planetary-scale services”. As mentioned in Chapter 4.1, the DHT Bamboo is run on PlanetLab. Another issue is that database systems must satisfy a set of strict requirements, for instance bounds on response times, near 100% successrate for requests and a high level of security. Many of the most important aspects revolving around database technology have not received much attention yet in the peer-to-peer world, as more fundamental problems must be solved first.

One existing system is *DHash*, developed by the Chord project ([cho]). It is a very simple storage system with only two data operations: `put(data)` and `get(key)`. As can be seen from the API, the system use immutable data items. This is not suitable for a database system. To obtain the key in DHash, you must first put the data and then distribute the key by external means. In many database applications, the typical case is that the key is somehow well-known, or at least well-defined, and that the value associated with the key changes over time. In DHash, the key for an item change every time the item itself is changed.

Chapter 5

Design

Most of the design choices made are related to the server part of the system. First the design process is described. In the following, a brief architectural overview is given, followed by a discussion of the client part design. Then we have a look at the Chord subsystem, as it relates to both the client and the server part. Lastly, the various design matters of the server part are documented. Each matter will be discussed in its own section. Note that although we talk about a client and a server part, the system is not a conventional client-server system. The server part in the system consists of a number of server nodes that self-organize in a logical peer-to-peer network, but is viewed as a single entity by the clients. Note also that a specific server node does not have knowledge about all other nodes in the system. In its most basic form, a node only knows about two other nodes; its predecessor and its successor.

5.1 The design process

This project has not been using a formal design process. There are three main reasons:

1. The resources available were very limited: one person and approximately 20 weeks of time.
2. The project was allowed to go on pretty much unrestrained (no formal deliveries etc.).
3. The implementation process was known to be heavily based on trial and error (exploration).

The design process could be said to be started with the depth study described in [Waa04]. Several important guidelines for a system based on peer-to-peer technology, Chord in special, were put forth there. With these as a starting point, we began building the prototype. The main focus was at all times to create a working prototype as soon as possible, and then extend and improve on it until the goals set for the project were met. Design and implementation went hand-in-hand, and the prototype evolved in short iterations. As only one person was involved in the system implementation, formal design documents were considered unnecessary. There was no need for coordination. It must be noted that a more formal development process would have been needed if the system had been bigger. The motto *embrace change* suited this project very well!

5.2 Architectural overview

This section documents the most important aspects of the system. It is necessary to get a grasp of these to understand the following discussions of more detailed parts of the system. The system as a whole is rather straight-forward, as decisions and correct operation can be determined by a node itself, or by consulting its immediate neighbors. There are no complicated distributed algorithms running. The goal of the system, besides offering database services, is to minimize the need for global knowledge. In this way we hope to achieve a higher degree of scalability than systems based on consistent "world views" can.

5.2.1 System overview

The distributed database system is built on top of Berkeley DB Java Edition (JE), and is split into a client and a server part. The client part is written to be as similar to the JE user interface as possible. The client part consists of the concepts *Environment*, *Database* and additional communication code to interact with the server part. The server part consists of a number of nodes organized in a peer-to-peer network. These nodes work together to serve requests from clients. Data is partitioned between the nodes in a way dictated by the routing protocol used by the distributed hash table (DHT) *Chord*. Nodes are organized in a logical ring, and support structures are used to make lookups effective (*finger tables*). Each node in the system maintains a number of local JE environments and databases. The number depends on how many of the put requests are routed to the given node, which in turn is dependent on the hash key (identifier) generated for the request. The hash key is a 160 bit long number generated by doing a SHA-1 digest of the name of the environment, the database name and the record key. Each node also has its own hash key, generated by doing a digest of the IP address and port number used by the node. The exact algorithm used to generate the identifier of a node is described in chapter 6. As nodes join and leave, data will migrate from one node to another. The migration always happen between two neighboring nodes. It is predetermined by the hash, in the sense that each database record has been issued a hash and this hash determine which node it belongs to. Note

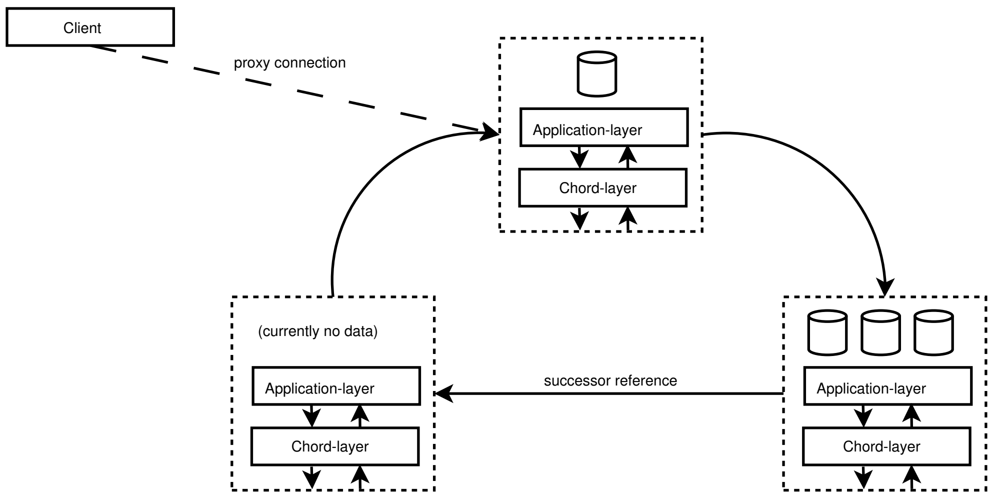


Figure 5.1: Brief system overview. The figure shows a system consisting of three server nodes organized in a Chord ring. Only successor references are shown. The client is using one of the nodes as a proxy to enter requests into the system. Note that a request is not in general processed at the proxy, but at the node to which the request hash correspond to. The data stores are Berkeley DB environments/databases.

that the "belongs-to" relation works in a dynamic system, and that the node it yields as result can change as nodes join or leave. If a specific sequence of node joins and failures is run several times, a record will be located at the same node at (roughly) any given space in time in each run. Consult Chapter 7.4 for results from test runs confirming this property. Figure 5.1 illustrates the main system components. Details are given in later chapters.

5.2.2 Client architecture

The client is used as the entry point to the system. It does not store any data, but simply forward requests to the server part of the system. The client part is very similar to the Berkeley DB Java Edition application. The user manipulates instances of `Environment` and `Database` as if they were local. All operations that change or access data are however forwarded to the servers and executed there. As opposed to the JE API, our client has a communication component used to communicate with the server part. More on the client design can be found in Chapter 5.4.

5.2.3 Chord subsystem architecture

At the start of this project, the Chord subsystem was already implemented as part of the depth study [Waa04]. During the course of the project, the code had to be extended and bugs had to be fixed. The main points of the design is stated here.

The class `Chord` contains the logic related to the Chord protocol. A thorough description of the Chord protocol/algorithm can be found in [SMLN$^+$02]. The goal is to maintain the data structures that describe the positions of the nodes in the Chord logical organization, which is a ring. After a node has joined the system, the node, and the rest of the system, is self-configuring and will adapt as new nodes join or existing ones leave. Two structures are vital for correct operation:

- **Predecessor**
  A reference to the node located immediately before the node in question.
- **Successor**
  A reference to the node located immediately after the node in question.

By using the two structures above, the system is capable of organizing itself into a Chord ring and route messages to the correct nodes based on hash values. However, they are not enough to provide effective operation and to give the system fault-tolerant properties. Two additional structures can be used to achieve these properties:

- **Successor list**
  This is an extension of the successor reference. The node obtains the addresses of its N immediate successors. N is the size of the successor list, and is a configuration value of the Chord algorithm. To gain anything at all compared to maintaining a single successor reference, it must be at least 2. Performance is not affected much by the value, neither gains in lookup times nor penalty related to maintenance. The systems ability to withstand node failures is however greatly affected. If the successor list size is set to 2, the system will fail if two neighboring nodes crash or leave at the same time. The effect will be that the Chord ring is broken, and routing of messages will fail if it somehow involves the node without a valid successor reference. As this situation can be hard to detect and fix manually in a large distributed system, counter-measures must be in place to make sure this does not happen. These are further discussed in Chapter 5.3.1.
- **Finger table**
  The finger table is a structure used to improve the performance of message routing in the system. According to the Chord technical report ([SMLN$^+$02]), it should have M entries, where M is the number of bits in the hash used in the system. In our case, this is 160 bits. The finger table would then list a subset of the nodes in the system at certain intervals. Let *baseid* be the hash of the node owning the table. Then the entries in the table would be $baseid + 2^I$, where I is in range $[0, M)$. Observe that the number of distinct server nodes in the finger table is dependent on the number of server nodes in the system and their hash values.

By using the structures described above, the Chord component is able to route the messages required for the servers to execute the database operations the clients issues.

It is also capable of routing around failed nodes, but temporary routing failures may occur. These will result in timeouts, and the operation should simply be retried. Descriptions of more detailed Chord issues can be found in Chapter 5.3.

It should be noted that the Chord implementation and logic is not bound to any specific communication protocol. Although UDP[1] is currently used, other protocols can be used as long as the required interface is implemented. As a matter of fact, the initial testing of the Chord implementation was done with a local interprocess communication system.

5.2.4 Server node architecture

A server node is responsible for storing a subset of the data in the system. It is split into two clearly separated layers: the Chord layer and the application layer. All communication goes through the Chord layer. Parts of the network traffic are related to Chord organizational maintenance. This is needed when nodes leave or join the ring. As a node failure can only be detected by another operative node, the state of a node must be checked by sending it messages. The Chord communication component also delivers application messages received from clients and other server nodes to the server component. Figure 5.2 is showing the different logical parts of a server node.

- **Chord**
  Handles the Chord protocol and delivers application messages from clients and other server nodes. To aid maintainability, a single class will be tying the two layers together. This implies that all communication to and from the application level must pass through at a well known location.

- **Database manager**
  Accepts incoming database operations and processes them in the local context. Also takes care of handles of different kinds; environments, databases and cursors.

- **Replication manager**
  Monitors replication status and makes sure the specified degree of replication is sustained. When nodes join or leave the network, the replication manager has to copy replication data to the new node or promote local replication data to primary data and ensure that the node accepts requests that were previously served by the node that left the system.

- **Data relocation manager**
  Checks whether the data located at the node actually belong there. When a new node is inserted, some data may have to be moved to the new node as a consequence of how the routing is performed by Chord. Due to the way the

[1]User Datagram Protocol: unreliable, connection-less network protocol.

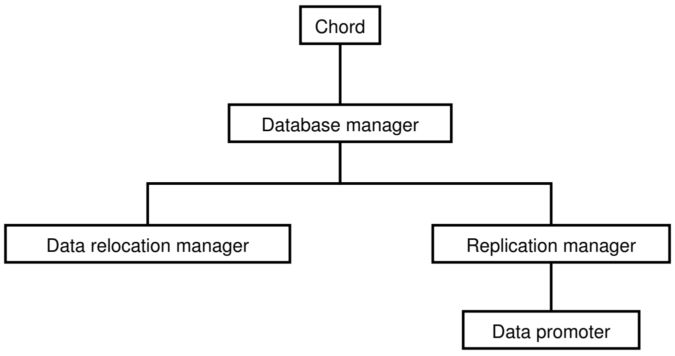


Figure 5.2: The main components of the server node.

Chord protocol is designed, the only event that can initiate the data relocation process is that of a *new predecessor*.
Say there are three nodes in a Chord ring; nodes 20, 40 and 60. Node 40 store items with ids 21, 25, 33 and 40. Then node 30 joins the ring. Items 21 and 25 must now be moved from node 40 to node 30. If node 50 joins the ring, node 40 does not need to care about this with regard to data relocation. The main challenge of data relocation, is the case of multiple new predecessors in a short period of time. This situation is investigated in Chapter 5.11.

- **Data promoter**
  Responsible for promoting replica data to primary data when this is necessary. Replica data must be promoted to primary data when a primary node fails. All nodes in the system are basically primary nodes. Upon node failure, the failed node's successor must take over the responsibility for the affected records. The process is elaborated in Chapter 5.10.

The following tasks are the main responsibilities of a server node:

1. Maintain Chord routing structures (required to perform other tasks).
2. Route/forward requests towards destination node.
3. Relocate data on node joins.
4. Process incoming database operations.
5. Make sure primary data is replicated at successor node.

5.3 Communication: Chord

Our database system has a very distinct separation between the application layer and the Chord layer. The communication at the Chord level is pure UDP. When it comes to the application level, we are free to choose the communication protocol that suits us best. However, no matter what we choose, we will be dependent on using the Chord layer. When ignoring the choice of the protocol to use, there are two alternatives:

- Use an independent communications module for the application layer.
- Integrate application layer communications with the Chord layer.

Due to the fact that the application layer does not know which servers it has to communicate with, the Chord layer must be utilized to retrieve the addresses of the correct peers. At a specific server node, the correct peer is always fetched from either the successor list or from the finger table. The process of finding the destination node stops when a server node is capable to establish that a specific node is the final destination node. This final node is always fetched from the successor list, because the finger table is sparse. The finger table is more sparse the higher the finger index is (remember the formula for the finger table entries stated in Chapter 5.2.3, consult [SMLN+02] for full details). If the final node was fetched from the finger table, we would risk picking the wrong node. The higher the number of nodes in the system is, the higher the risk would be.

It is evident that the former approach mentioned in the list above would be inefficient. Consider this typical work pattern:

1. Obtain hash for database operation.
2. Send lookup request to Chord.
3. Wait for and receive result from Chord.
4. Send application message.
5. Wait for acknowledgment/reply.

Observe that the system will know the address of the destination node at the end of step 2. By combining step 2 and 4, the time it takes to deliver an application level message can be greatly reduced, and a the typical work pattern would look like this:

1. Obtain hash for database operation.
2. Send application message via Chord.
3. Wait for acknowledgment/reply.

As can be seen, an integration of the application layer and Chord layer communications would be far more effective than using two independent subsystems. A technique that can be used to achieve this, is *piggy-backing*. We let the application message ride along on the Chord message, and then we ensure that the Chord subsystem delivers the application message when the destination node has been reached. As we stated in Chapter 2, we expect the majority of record keys and record data to be small, so that they will fit into a single datagram packet. This expectation will not hold in a production system. There are other aspects of UDP communication that must be handled as well, since UDP is an unreliable communication protocol:

- Transmission of large data structures (large as in size exceeding the capacity of a single UDP packet).
- Caching of messages to support resend-requests.
- A way to identify a message as a reply to another message.
- Mechanisms to support acknowledgments.
- Timeout mechanisms.

The three last requirements are already in place, or can be implemented by using existing functionality. The two first must be implemented from scratch.

By integrating our application layer communication with that of the Chord layer, we restrict ourself to use UDP communication for the application level as well. This is probably not a bad thing. Chord itself is very well suited for UDP, as was stated in [Waa04] (Chapter 5.2.1). Could the application layer benefit from using TCP[2] instead? As this protocol is reliable, we would not have to deal with out-of-order packages and missing packages. On the other hand, we would have to accept the connection setup time, which can result in a dramatic performance degrade in our application. A node may have to communicate with a number of other nodes, and these are not known before the Chord protocol gives an answer. Further, if finger tables are used, we could end up having to maintain connections to over 160 different nodes. How many connections should we pool? As each connection requires its own port, the computers running the software would have to open a range of ports, which is inconvenient. If the churn rate for the system is high, connections would also be invalidated frequently as the finger table or other Chord routing structures change. Last, by using TCP we would not be able to take advantage of piggy-backing application messages on the Chord messages, and the response time would then automatically increase by at least one message round-trip time.

[2]Transmission Control Protocol: reliable, in-order delivery, connection oriented protocol.

5.3.1 Avoiding ring breakage

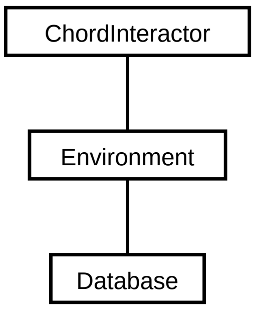
As we have decided to use the Chord subsystem for communication, we must make sure it will be capable to fulfill the application levels needs at all times. The main issue is to make sure a node always has a reference to its successor. If this reference is invalid, the Chord protocol cease to function properly. Invalidation can happen during transitional states when nodes join or leave the Chord ring. The most common cause is that the successor leave the the ring for some reason. The following structures can help us tackle this situation:

- The successor list (size of 2 or bigger).
- The predecessor reference.
- The finger table (if in use).
- The bootstrap node specified on startup.

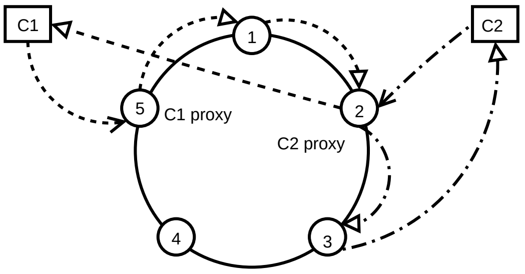
Using the successor list to obtain a new successor if the previous one fails is standard behavior in our system. If all nodes listed in the successor list have failed, the Chord node initiates a repair protocol. This consists of the node sending a `findSuccessor`-message with itself as argument. The only requirement for this action, is to have the address of another active Chord node which is already included in the ring. When the node receives an answer, it updates its successor reference, entry zero in the successor list, to the node specified in the answer. This is in effect the join procedure used during node startup (bootstrap process). The Chord protocol will now make sure the affected nodes update their primary routing structures in a short period of time. These structures are the predecessor reference and the successor list. The finger table is considered a secondary structure and will take longer to update. Note that the finger table structure has received rather little attention during this project, and it should be further tested and developed at the implementation level.

5.4 Client component

The design of the client component is highly dictated by the existing Berkeley DB Java Edition (JE) application programming interface (API). One of the project goals was to keep the API of the system as similar as possible to that of JE. As this project has its focus set at peer-to-peer technology, a lot of functionality from the JE API is left unimplemented. Parts of it is simple to implement, but is left out because it pose no interest to this project and due to the time constraints imposed on the project. Other parts are heavily influenced by the switch to a peer-to-peer based system, and it is even impossible to implement some of them without auxiliary structures and functionality



a) Client architecture



b) Proxy illustration

Figure 5.3: Overview of the client and proxies. As can be seen in a), it uses the same abstractions as the JE API. In addition it has a communication component, `ChordInteractor`, for connecting to the server part of the system. In b), a system with two clients and five servers is shown. Each client uses a proxy server node.

in the system. An example of this is methods that have to access all, or a subset of, the records in a specific database, without explicit knowledge of the record keys. The reason is simple; the record key is needed to determine which node the record is stored and processed on. A *distributed cursor* can be used as a concrete example. Proposals of solutions for these kind of situations can be found in [Waa04].

The client part is summarized in Figure 5.3a.

A comparison with the JE API reveals that almost all JE classes are missing in the figure (see Appendix B for a list of classes). The missing classes can be divided into two categories: do not need to be changed and need to be re-implemented. The classes that do not need to be changed are not affected by the switch to a peer-to-peer system. For instance, this holds for the classes `DatabaseConfig` and `JEVersion`. Examples of classes that would need to be re-implemented are `Transaction` and `Cursor`. These two classes are important concepts in database applications. Nevertheless, we decided to leave them out in this project. Both of them could be implemented in the system, but there are not enough time to do the work. Some of the JE classes are also possibly rendered useless, as they are abstractions for concepts that apply to a centralized database. For instance, what would you expect the class `BtreeStats` to tell you from a clients point of view? As the data of a database is distributed over several databases, and thus several independent btree-structures, there are no way to calculate the different statistical values. All basic JE classes (package `com.sleepycat.je`) are listed in Appendix B, along with their status regarding our prototype.

One new component has been added; the `ChordInteractor`. This component is responsible for all communication between the client and the server part of the system. As reflected by the name, all communication happen through the Chord subsystem. This way, we will not have to implement extra communication modules in neither the client nor the server part. Traffic from clients should however be discernible from inter-server communication. Communication is done through the same communication module as is used by the Chord subsystem in the servers, but the Chord operational logic itself is not present in the client. We want to keep the clients clearly separated from the server nodes. As the system is a peer-to-peer system, all nodes should offer the same set of services. As the clients are not able to store data and serve it on request, including them in the Chord ring would conflict with the demand of equality. Further, all algorithms dealing with data maintenance would be unnecessary complex. It would also cause severe problems related to request routing. All request that would happen to be routed to a "client node", would have to somehow be forwarded to the nearest server node. In a highly dynamic system it would be very hard and expensive to keep track of these artificially created routing anomalies. We decided to let the client use a *proxy*, which is any single server node in the Chord ring. All requests are sent to the proxy, which in turn use the Chord internal routing protocol to route the request to its target host. The reply could be sent backwards along the ring, or the target host could send it directly to the the client. We have settled for the latter approach, as this seems to be the most effective one. An illustration of the proxy concept is shown in Figure 5.3b. Two clients are shown, each sending a request. The path of the requests are drawn up (we assume that the nodes only keep track of their immediate successor).

Although we have decided to let each client use a single proxy, it might be a good idea to utilize a set of proxy nodes. This would provide a rudimentary way of load balancing, and reduce the chance of several clients swarming a single server node with requests. A set of proxies would also allow the client to continue operation in the case of a proxy failure. The server part would automatically adapt to the node failure, but without intervention the client would be cut off from the servers if it only had knowledge about a single proxy.

For a client to be able to connect to the server part, it must obtain the address of at least one server node. This must be done through external means. Note that it is enough to get the address of one server node. If the client has been able to connect, it can ask this server node to supply it the addresses of a number of other server nodes. The list of server nodes should be validated or updated at intervals to ensure that at least a subset of them are indeed still active. Several general methods for *discovery* are described in [Ver04]. Briefly, these are:

- Static configuration
- Centralized directory
- Use of the domain name system (DNS)

Although the methods are described in the context of peers discovering each other, they still apply to our case, where a client need to connect to a server system consisting of a number of peer nodes. Our client will require the the operator to supply the address of a server node to be able to connect (static configuration).

JE returns operation results through the class `OperationResult`. It has been designed for use with a centralized, embedded and single node database system. It is very simple, and is really nothing but a collection of defined strings. It poses two problems regarding distributed application:

- It does not implement `Serializable` or `Externalizable`, and cannot be sent over the network.
- It cannot convey information about errors that arise on remote hosts.

The second point above cause problems, as the standard JE way of giving feedback during database operations is either through an `OperationStatus` object or through a `DatabaseException` object. As an operation is executed on a remote host, the information the exception gives is lost. Our solution is to use a new result object. Depending on the result on the remote host, the result object is either "converted" into a `OperationResult` object, or a remote exception is rethrown locally at the client. As new code is introduced into the operation execution process, the client must be able to separate between JE database exceptions and other Java exceptions to aid debugging and troubleshooting. A new kind of errors are those related to *timeouts*. As the reason for a timeout can be complex, it is generally accounted for as a database exception. This suggests that the client should simply retry the operation that timed out.

5.5 Definition of node join and node leave

We will be talking a lot about node joins in the chapter. What is a node join? There are two possible definitions:

1. The new node is inserted into the Chord ring organization.
2. The new node is inserted into the Chord ring organization **and** has received all data is should from its neighboring nodes.

Later we will measure the cost of node joins. The value of the measurement will depend heavily on whether we base it on definition 1 or 2 of a node join.

Definition 1 is related to the Chord subsystem only. When the node has updated its predecessor and successor references, it has effectively joined the system. The time it takes to update these references is directly dependent of configuration parameters for Chord and the current state and environment of the system. First, a *findSuccessor* call is made. How fast this is resolved depends on the Chord routing structures and the network latency times between the involved nodes in the system. When the node receives an answer to the call, it updates its successor reference. Then, after a short time, specified by the Chord *stabilize interval*, the predecessor reference will be updated. Populating the successor list and the finger table is not considered part of the join, as they are not required for correct operation of the Chord subsystem. Note that when we speak of the successor reference, we mean the first entry in the successor list.

Definition 2 includes the process of transferring data to the new node. As described in more detail later, the predecessor must replicate its primary data to the new node, and the successor must relocate some of its data to fulfil the relocation requirement (based on the hashes). The join process is now made dependent on the amount of data at both the predecessor and the successor, and the network capacity between the three nodes involved.

Based on the goals and limitations we set for the project in Chapter 2, we will use definition 1 of the join process when measuring the costs of it.

A node leave is simply the event of a node stopping to reply to messages from other nodes. We will not separate between a planned node leave and an unplanned leave. The data maintenance protocol described later will handle both cases equally. For extra security regarding data availability, one could implement a planned leave protocol, where the node leaving does not do so until it has confirmed that its responsibilities have been successfully taken over by other nodes.

5.6 Handle management

JE operates with a concept called *handles*, which can be used to manipulate the underlying structure. In Java, a handle is essentially a reference to an object. In our application, the handles used are environment handles, database handles and cursor handles. When using JE, all handle management is local. But even there care must be taken to follow the proper routines for handles. For instance, all handles related to an environment should be explicitly closed before the environment handle itself is closed. If a handle has been closed, it can no longer be used. Attempts of using it will result in exceptions being thrown.

In the project described in [Uls03] and [Uls04], a distributed Berkeley DB was designed and implemented. If an environment entered the system, it was synchronously created at all servers and a unique identifier was created and maintained in a distributed dictionary service. Data were distributed based on a regular hash function. This approach is not suitable in our system, as the number of nodes can be very high. For a given operation, or batch of operations, only a small subset of the server nodes would be involved in the processing. Creating or opening an environment is an expensive operation, and further performance penalty would be introduced in the process of notifying all servers to perform this task. We choose to let environments be created and opened on request only, which is to say that an operation must arrive at the node before the environment is opened. To compensate for the performance penalty involved, each server node keeps a *handle pool*. Environments are only closed when the pool capacity has been reached. The factor limiting the pool size is the amount of main memory available to the Java virtual machine. If the environment cache size is set too low, database operations will be slower due to increased disk activity.

Environment and database handles are identified by name only in our application. Databases already have a name attribute. Environments do not have a name attribute, but they do have a *home* attribute. This is the path of the environment directory, and we use this to identify environments. To allow for different locations on different hosts, relative paths must be used. If an environment is located in `/var/datastore/dbRecords`, its name is 'dbRecords' and '/var/datastore/' is the base environment directory of this server node. The scheme allows using different base environment directories on the server nodes. We add the restriction that all environments must be stored in a single base environment directory to simplify handle creation and reduce performance penalties induced by disk access when searching multiple disks or base directories.

In a multi-threaded application, care must be taken to avoid that one thread closes a handle and another thread is working on it concurrently. For cursors, the only resolution of such a situation would be to open the cursor again and restart the process which was running. Note that a cursor handle would be invalidated if the underlying database or environment handle is closed.

5.7 Database operations

To keep operation execution code clearly separated from the rest of the server code, it was decided that all database operations should be implemented in their own objects. A requirement for this, is that all operations can be processed in the same way. As it turns out, this is the case. All operations are executed by running the method `execute(environment,database)` of the incoming operation object. In addition, all operation objects are required to implement methods to provide the names of the environment and the database the operation is to be executed on. To add another operation to the system, the following must be done:

- Write a new operation class, which boils down to implementing the `execute` method and the constructors required.
- Add code for creating instances of the new operation class in the client.
- Distribute/add operation class to all servers in the system.

The number of operations in our system will be low, as the database logic present in the system is very low level. The operations that will be implemented are *put* (several variants), *get* and *delete*. In addition to database operations, there are operations that execute on environments. These may be implemented in the same way.

5.8 Data distribution

Data distribution is necessary to achieve scalability, as it is a way of dividing the system work load to several different hosts. For the technique to be useful, the costs related to data distribution must not be too high. In our prototype, the critical factor is the cost of routing messages from an originating host to a destination host.

The most natural way to control distribution in our system is to make direct use of the Chord subsystem. It is based on *consistent hashing* [KLL+97], which confines the need for data migration on node joins and node failures to a very restricted subset of all the nodes in the system. For a single node join, only the successor of the newly joined node is affected when considering data distribution alone. In addition, the predecessor must replicate its own primary data to the new node.

It is important to realize that the properties of *consistent hashing* is a matter of necessity for our system. If we were to use regular hash techniques, we would end up with redistributing all data in the system on every change in the ring organization. This would make the current approach used in our system unusable, as it would drain all resources and leave the system in a state of request denial for too long after each node join or node crash.

How can the Chord hashing provide us with valid node selections for data distribution? As we know, each node in a Chord ring is assigned a unique[3] hash. In our system, all database operations are assigned a hash as well. The hash is generated by doing a SHA-1 digest of selected operation properties: the environment name, the database name and the record key. Note that the key must be specified for all record operations in JE, thus it is valid to assume it is known at hash generation time. We now have two sets of hashes; one for all nodes in the system and one for each record operation. Remember that no node in the system has knowledge of the whole set of node hashes. The system must now find the largest node hash that is smaller than the record operation hash. The successor of this node is the destination node for the record operation. This process is carried out by the Chord subsystem, and is actually one of the main tasks the Chord algorithm is designed to perform. The operation is named *findSuccessor* in the Chord documentation ([SMLN+02]). The *findSuccessor* operation is not directly compatible with out piggy-backing approach of delivering application messages. As one can see, the process of finding a destination node stops at the predecessor of the destination node. A mechanism of transporting the application message from the target node's predecessor to the target node itself must be implemented. Unlike most other

[3]There is actually a negligible chance of two nodes getting the same hash (one in 2^{80}). For a general discussion about SHA-1, see [Sch05].

operations in the system, we do not need to use the Chord routing algorithm. In fact, we cannot use it, as we would always end up at the node preceding the target node!

For the scheme outlined above to work, there is one critical requirement: Record data must be present on the node identified as destination node by the Chord subsystem. For a static system, this is not an issue at all. When nodes join and leave while requests are being served, one must take care to ensure that consistent results are given for the requests. The challenging situation is when data is migrating from one node to another to fulfill the location requirement. This situation can happen when a new node joins the system or an existing one leave. The actions taken differ for these two cases, and each one is described separately below.

For the case where a new node joins, a migration process is initiated. During this process, the new node cannot obtain the correct response to a request without consulting its successor. There are two approaches to handle the problem:

- Deny processing of incoming requests until data migration is finished. Note that this denial would only happen on requests to be handled at the node in question, not for the system as a whole. Routing will not be affected.
- Implement a synchronization protocol between the two nodes involved in the data migration process. The protocol must ensure that all existing data is seen at all times during the migration process. Without going into full detail, there are at least two approaches:
  - Forward all requests to be processed at the new node to its successor (the previous destination node). The successor must keep all data to be migrated intact until migration is completed, and then send a "back-log" for operations that modified records during migration. The modified records would only be reflected at the successor and the new node cannot take responsibility for the records until these modifications are made locally.
  - Check for data locally first. If found, generate response. If not found, consult successor. If a record was found at the successor, it must be fetched and deleted from the successor as part of the operation execution to avoid illegal overwriting. This approach is further complicated if the databases are allowed to keep duplicate record entries (several data entries for a single key).

A combination of the above approaches could also be used, for instance to allow all read-only requests and deny operations that modify records. Note that operation replication must be taken into consideration during migration as well.

Most of the observations stated above also apply to the case where an existing server node leaves the system. However, this process should be quite a bit faster, as the migration process is replaced with a *data promotion* process. What happens here, is that the replicated data from the failed node is promoted to primary data at the successor. The process is elaborated in Chapter 5.9.1.

As we have not set any specific goals regarding availability/uptime, but assume nodes are light-weight (see section 2.1), we choose to implement a simple request denial protocol to be used during data migration and promotion. As we are to end up with a working prototype, implementing a much more complex synchronization protocol seems too extensive for this project. Additionally, the complexity of such a protocol is unknown.

5.9 Data replication

To ensure that data is not lost in the database system, replication is performed. The *replication degree* of the system, that is how many copies of data are created, will be fixed to one. This choice is based on the fact that a higher degree costs too much compared to the benefits gained. The cost is expressed by higher storage requirements, higher request response times and more complex code for both replication and recovery. For instance, with a replication degree of one, we do not need to keep any metadata about the replication data, as we know it all "belong" to our predecessor. This greatly simplifies the promotion process of replicas to primaries during take-over. The *replication mode* has been set to *strict*. This means that all replicas are created and verified before an answer is sent to the client. Strict replication is a lot easier to handle than lazy replication, and it is further justified since the replication degree is fixed at one. Another advantage is that strict replication only affects updates.

Primary data on a node in the system is replicated to the *immediate Chord successor* of the node. This choice of replication location has the following advantages:

- The replication location is well defined. It can be derived from the hash of the record and a call to the Chord system. Note that a hash does not in general imply a specific node. If you ask the system to resolve hash H and the result is node N, you cannot later assume that H resolves to node N. New nodes may have joined the system, or existing ones left, that causes the record with hash H to migrate.
- The replica node is automatically promoted to primary node for the data it holds replicas for, as a consequence of the Chord routing algorithm.
- The replicated data is already located at the new primary node.

There are some challenges as well:

- Care must be taken to assure functional correctness of the system during node joins and failures/parts. The main challenge is related to situations where several nodes join and/or leave at the same time, causing already started protocols to be rendered invalid. **If replication data happens to be wrongly placed, it must be considered lost**.

The unit of replication is individual records. The primary node of a record is decided by the hash of the records environment, database name and the record key. The hash is generated by the SHA-1 digest algorithm. This scheme, which emerges from the way the Chord protocol is designed, can cause trouble for some types of operations. Without any support mechanisms it is impossible to retrieve all records for a specific environment and database without knowledge of all record keys. A few solutions to this problem were proposed in [Waa04]. Due to time constraints, these mechanisms will not be included in our system.

For consistency and control reasons, we do not want a node to serve replication data if the primary node is operational. More on this issue can be found in section 5.10. The main point of concern is what happens if records are updated at the replica node, but not on the primary.

There are two different types of replication used in the system:

1. Immediate individual record operation replication.
2. Batch based primary data replication.

Type 1 is initiated when a node receives an operation that must be replicated. This is generally all operations that modify a record.

Type 2 is used when all local primary data must be replicated to another node. This is necessary both when a new node joins or an existing node leaves the system. In the former situation, because the the new node does not have any replication data, and we want the primary node to populate it. We could have let the previous replica have copied its replica data to the new node, but then we have to deal with synchronization and consistency issues. In the latter situation, the replica data is lost and must be recreated. As with all data maintenance protocols, synchronization must be applied during database replication as well. We have the same choices here, and we still settle for a simple request denial approach.

5.9.1 Replication data

The system is to be capable of replicating environments and databases. Because the Chord lookup protocol is used, the entity of replication has to be that of an individual record. The replication data will be distributed in the system as the primary data, only offset by a number of nodes (depending on replication degree). In our current implementation, replication data is always stored at a node's successor.

There are several reasons to keep replication data separated from primary data within each node. First of all, it is necessary to keep the distinction to be able to assert correct operation with regards to synchronization protocols and lookup. We do not want a node to serve replication data without knowing it. A second concern is that of consistency. Replication data is not to be served directly to clients before it has been promoted to primary data. This promotion happens as a step in the take-over protocol. Take-over must be fast, so the mechanism used to separate replication data from primary data must not incur substantial delays during data promotion.

The following mechanisms could be used:

1. Maintain an in-memory structure to determine if a *database* is a primary or replica.
2. Maintain an in-memory structure to determine if a *record* is a primary or a replica.
3. Separate replicas from primaries by prefixing the environment home (directory).
4. Separate replicas from primaries by storing them on different places on disk.

Approach 1 is very fast, but information is lost if the node crashes (forced shutdown, power outage, virtual machine exception, machine restart, etc.). The information in the memory structure would have to be integrated with the handle manager, as so to refuse giving out handles for replica databases as if they were primary ones. To survive abnormal shutdowns, the memory structure could be written out to disk, but then care must be taken when restarting the node. The operator could have deleted, or even added environments while the node were down. The observant reader would now have realized that this approach is not possible! We cannot assume that we do not have replica and primary records for the same database. Therefore, if we are to keep track of which records are primary and which is replicas without separating them by holding them in different databases, we must go for approach 2.

As approach 1 is not possible due to the distribution scheme used, we must adapt. The solution is to keep track of every record stored at the node. This is not a good way to handle the situation. The structures holding the information would grow with the number of records, and we would also have to update them on put and delete operations. We could use the structures during data relocation, but we would then have to store the record key, the hash for the record and replica status. As JE, and our prototype, allows records to be of any size, the memory cost would be unacceptable. In addition, we cannot any longer let the handle holder restrict access to handles. The application would have to consult the information structure on every database operation to check if the record in question is a primary or a replica.

Approach 3 has the advantage that the operator is capable of separating primary and replica environments by simply looking at the directory names. To promote a replica to a primary, one of two actions must be performed: Rename directory or read each record from the replica database then write them into the primary database. The decision criteria would be whether the primary environment directory already exists or not.

Approach 4 may suffer from higher execution times, as data must be moved between two disk locations. The penalty is dependent on the file system used. Again, separating primaries from replicas is simple.

We think that approach 3 is good enough. The mechanism survives node crashes, is simple and does not impose unacceptable costs. If we could have used approach 1, we would have. The fact that the system has to operate with an individual record as the working unit is one of the major drawbacks of the current system design, but at the same time the basis for the data distribution scheme (hash is ”random”).

5.10 Take-over on node crash

Take-over is in its most simple form easy to support in the system. From the database system's point of view, there is one *passive* and one *active* part of the take-over protocol. The passive part is automatically performed by the Chord subsystem. As we know, operations are routed to a node for execution based on the hashes (identifiers) of the operation itself and those of the nodes in the system. When a node crashes, or otherwise leaves the Chord ring, the routing structures are updated so that the crashed node's successor becomes the target node for the operations that would have been routed to the crashed node. A consequence of this, is that the database system (application layer) does not have to take any actions related to request routing.

The active part of a take-over, is making sure that all data related to any operations that will be routed to a specific node is indeed present. This is very important. If the system somehow manages to misplace data records, valid requests for this data can be answered with incorrect results. Even worse, later events related to the maintenance of the Chord ring structure can potentially lead to stale data being presented to the clients.

A requirement for the active part of the take-over is that replication is enabled. If it is, the replica data present locally will be *promoted* to primary data. The node can now serve requests that the crashed node previously had the responsibility for. As long as the replica data is up to date, the results given by the new node will be consistent with those previously given by the crashed node. This requirement demand that the system use *strict replication.* To be able to promote data, the database system must somehow be notified about the change in the underlying Chord node organization.

This notification is best served by utilizing an *event publishing system*, which allow us to avoid active polling to detect events of interest. Details about the event publishing system can be found in Chapter 6 on implementation.

Take-over is performed by executing the following protocols:

1. Promotion of replica data on the failed node's successor.
2. Replication of all primary data stored on the failed node's predecessor.

Protocol 1 must be executed before protocol 2 to avoid mixing replication data from different sources. As explained earlier, mixing can cause stale data to be presented to clients, because we assume replication data on a server node belong to the predecessor alone. Note that the two protocols are initiated on different nodes; the failed node's predecessor and successor. Both are triggered by events from the Chord subsystem. As order is of importance, one must take actions to ensure that differences in network latency times between the nodes or other reasons cause the order to be invalidated. The events generated are those of a new predecessor and a new successor. If one of them is seen, the other can be excepted as well. This can be exploited for synchronization purposes.

5.11 Take-back on node join

Take-back is absolutely necessary to maintain correct operation of the system. If data is not reclaimed by joining nodes, the system will give invalid replies to requests. The reason is simple; data is not located where it is supposed to be. As each node only has knowledge about data that is local, it is not able to generate correct replies if data is located at another node. An exception to this is replicated data, which is by definition located at the nodes predecessor. However, replicated data is not considered local until it has been *promoted* to primary data. This happens during take-over, which is described in Chapter 5.10.

There are two distinct cases that cause complications for the take-back protocol. Let us say the nodes 100 and 200 are already inserted into the Chord ring. The other nodes in the system are ignored. Then let us say node 150 joins the system. Node 200 will start migrating all records with hashes smaller than or equal to 150 to node 150. Now, let us further say that another node joins the network, while data is still migrating. We have two possible scenarios:

a) A node with a hash placing it between node 150 and 200 joins, say node 170.

b) A node with a hash placing it between node 100 and 150 joins, say node 130.

In scenario a), data that should have been migrated to node 170, have been migrated to node 150 instead. The actions required to fix this error, is not implemented in any of the normal data maintenance protocols, so we must implement them in a dedicated module. Again, all records must be checked. The record hashes must be computed and compared against a relocation criteria: If record hash is bigger than node hash, then it must be relocated to the nodes successor. This protocol must be executed on the node to which data was migrated. In this concrete case, this is node 150. In addition, synchronization must take place between all three nodes involved. If they all were to go on with their data maintenance protocols as with a single node join, the risk of creating a deadlock situation would be high. This can be remedied in a number of ways, but the most simple one is to run all data maintenance protocols in a single thread. This way, only one maintenance protocol will get access to the databases at any time. The rest of the protocols will be queued (by another thread) and must wait for turn. Take notice of the fact that this approach is not suitable for huge data volumes and slow network connections, as it could cause smaller and more important maintenance tasks to be severely delayed. However, with the system limitations stated in section 2.1, we should be okay with it for our prototype.

Scenario b) can be tackled by using a simple synchronization protocol, which postpones the data migration from node 150 to 130 until the migration from 200 to 150 is completed. Node 200 will relocate all records correctly (viewed from its local context), whether node 150 gets a new predecessor during the process or not. One could save some time by adding more logic to the data relocation process on node 200, for instance we could let it operate with multiple relocation criteria, but this would be unreasonably complex for a system like ours. Beside the increased complexity of the algorithms themselves, one would have to distribute information about ring organization changes to other nodes then the ones they affect directly. In our example above, node 150 would have to inform node 200 that it has gotten a new predecessor (node 130).

Take-back is performed by executing the following protocols:

1. Can be executed in parallel:
   (a) Delete current replica data on the new node's successor.
   (b) Replicate primary data on the new node's predecessor.
2. Relocate data from the new node's successor.
3. Demote the primary data relocated in step 2 to replica data on the new node's successor.

As can be seen from the list above, most of the workload is placed on the new node's successor. Unlike in the case of a take-over, we do not need to synchronize the protocols executed by the new node's predecessor and successor.

5.12 Consequences of misplaced data

The consequences of misplaced data in the system depends on a few factors. If the only copy of a record is misplaced, the system will consider it non-existing. A client requesting the record will be told it does not exist, and the system has failed to fulfil its responsibilities.

If a replica of a record is misplaced, it might or might not cause problems. To open up for problems, it must be promoted to primary data first. We now have to possibilities:

- The record is misplaced such that the node it is on is not the successor to the correct location. The misplaced record will never be accessed by a client, because requests for it will be routed to another node.
- The record is misplaced such that the node it is on is the successor to the correct location. If the node at the correct location fails, the misplaced records is now correctly placed, but out of sync. If the primary data was properly replicated, the replica will correctly overwrite the copy of the record already present in the primary data and all is well again. However, if the misplaced record has been deleted at its correct location, we now have a stale record in the system that will be served upon request. This situation will not be corrected by the data maintenance protocols, and the system will remain in an inconsistent state until a client or application program detects the error and fix it.

Note that the former situation can be turned into the latter one over time, as nodes join and leave the system. Care must therefore be taken to clean up after aborted data maintenance protocols, as they are potential problems with severe effects for the clients of the system. To correct the error, one could let a background process access each record on a node and verify that it actually belong there. This process might be expensive in terms of resource usage, and a mean of restricting resource utilization should be implemented if such a process is realized.

5.13 System configuration

There are a number of configuration settings for the system. It is considered important to allow users to easily change these settings, as the system is a research and development prototype. Although not enforced, is it recommended to keep behavioral configuration settings equal on the servers. If servers are configured in many different ways, it can be extremely hard to debug and draw conclusions about system behavior. This warning applies especially to the Chord subsystem. For instance, all servers should agree on whether finger tables should be used or not, and on the size of the successor list. Certain setting combinations may cause a node to fail or behave strangely (see implementation of successor list reconciliation for a concrete example).

The system is configured with Java properties. These can be set in three different ways:

- Specified in a configuration file (following standard Java property syntax).
- Specified on the command line (-*D* arguments).
- Specified programatically.

The available configuration properties are all listed, and more or less documented, in the sample property file included in Appendix D. A small subset of the properties must be specified to be able to start the server and client. If this is not required, default values are used. For details of which properties are mandatory and which are optional, consult Chapter 6 on implementation.

Chapter 6

Implementation

This chapter describes different aspects of the source code of the prototype, and gives details of how the solutions proposed in Chapter 5 on design were implemented. Some practical information regarding how to use the prototype is also given at various points, most noticeably Chapter 6.1 and Chapter 6.10. The latter chapter is very specific and practical oriented. Some additional aspects that slipped our mind when writing the design chapter are also documented in the following chapters.

6.1 Code requirements and organization

The source code is organized in a package hierarchy. For the application level, the final source code weighs in at about 7100 lines (and 1500 lines of simple tests), including blank lines and comments (quite a bit of JavaDoc as well). This is not a very high number, but as there has been substantial exploration with trial and error, the actual number of code lines written is higher. In addition, the Chord subsystem consists of about 5200 lines (including a few unittests). The base package name is `no.ntnu.dbdb_p2p`. In the following, the base name is left out for brevity. The code organization is shown in Table 6.1, along with comments on the contents.

Subversion ([sub]) was used for source code versioning. In addition to being used for the obvious purposes, it was also used as a way to keep code in a single location. This was important during the project, as the developer worked at multiple locations. As a consequence of this, the revisions in the repository is not always usable. They might not even compile. Also, as this was a small project only involving one developer, there was not set any demand about that the code should compile or the system work for all revisions in the repository.

Apache Ant ([apa]) was used as the project build tool. These are the main targets:

(base)	Client code, substitute for package `com.sleepycat.je`. Note that classes from the Sleepycat package must be used in conjunction with the classes here.
common	Abstractions and tools used by both the client and the server.
devutils	Simple utility tools useful when developing and testing.
examples	Examples setting up and running simple database interactions.
server	The server internals. Most of the important code is present here.
server.datastructures	Classes representing datastructures (holder classes).
server.exceptions	Custom exceptions thrown by the system.
unittests.no.ntnu...	Some unittests testing the implementation (low level).

Table 6.1: Application level source code organization.

- 'compile' - compile all application sources.
- 'test' - compile tests, create test data and run tests.
- 'release' - same as targets 'compile' and 'test', and in addition create a JAR file.
- 'clean' - delete build directories, temporary files and class files in the source code tree.

The application level source code has the following dependencies:

- Java version 1.5 or higher.
- Chord subsystem library (`Chord.jar`).
- Berkeley DB Java Edition 1.7 or higher (`je.jar`).
- JUnit ([jun]) for unit tests (optional).
- Ant for using build file (optional).

6.2 Thread usage and synchronization

Our prototype uses a number of Java execution threads to carry out the required tasks. We must use threads to achieve an acceptable level of system responsiveness on behalf of other servers and the clients. A simple implementation where the system as a whole waits for a request, process it and then waits for the next request is not doable. There are two reasons:

- Utilization of the Chord subsystem.
- Need for synchronization within the application level and between different nodes.

The Chord subsystem is at all times sending messages to other Chord nodes as a part of the Chord protocol. This is necessary to maintain the Chord routing structures. If these messages are delayed due to request processing, the Chord protocol would fail to work. To allow for rather fast take-over and take-back, we have imposed strict timing requirements on the Chord protocol, compared to most other peer-to-peer system. The *stabilize* routine is run every 200 milliseconds as default.

When certain data maintenance protocols are started at a collection of nodes (normally three different server nodes), the tasks being performed must be synchronized to assure correct system behavior. While one thread is executing a task at a node, we need another thread to tell nodes requesting other tasks to be performed to wait until it is ready. This is done by sending a *request denial* message. If we had only one application level thread, these nodes would not be notified of the situation, and would have to assume the node they are contacting has failed or use high timeout values as a work-around.

The different threads in the system are synchronized on specific objects, using the `wait` and `notify`/`notifyAll` statements. The objects used for synchronization are mostly message or request queues (typically dynamic array structures). Processing threads sleep until a request arrives. If another task arrives while the previous one is being processed, it is processed immediately without the thread going to sleep again first. This is an important point, as without this mechanism one or more tasks can be left in the processing queue for a long time, until another incoming request triggers the notification of the thread again. The typical body of a `run` method for a thread looks like this (Java-/pseudocode):

```
while (doRun) {
    while (doRun && !tasks.isEmpty()) {
        // Process task.
        // If shutdown is initiated, ignore remaining tasks in queue.
    }
    if (doRun) { // Don't go to sleep if shutdown is initiated.
        synchronized (tasks) {
            try {
                // Wait for notify.
                tasks.wait();
            } catch (InterruptedException ie) {}
        }
    }
}
// Thread shutting down.
```

So far, this simple synchronization has been working on both Linux and Windows platforms without problems. The usage of queue objects allow us to change the thread of execution for a task. One thread insert the task into the queue, another one takes it out for processing. To achieve a higher degree of concurrency and throughput, the server should be made to utilize a pool of processing threads for requests.

Another problem with multiple threads in our prototype, is access to handles. As stated earlier, our prototype uses environment handles, database handles and cursor handles (JE concepts). There are two threads using handles; the `DatabaseManager`-thread and the `ServerRequestProcessor`-thread. Only certain actions actually cause problems, for instance if one thread closes the handle and another one tries to access it. All environment and database handles are fetched from the `HandleManager`-instance created by the `DatabaseManager`, which means that a synchronization mechanism can be implemented there. In our prototype we do not use any explicit synchronization mechanism, because we use the request denial protocol. This effectively stops the possibility for illegal handle management in the current implementation. Note that timeouts while waiting for lock acquisition may occur if there are data in the environment base directory on startup. The system is not implemented to handle this situation.

6.3 Communication

Network communication is a crucial and complex component in a distributed database system. In our prototype, all communication is done in or through the Chord communication system. There are three distinctive communication paths, and each of them will be described in the following sections, after a general explanation of how the communication is implemented. The structure of the datagram packets is explained subsequently.

6.3.1 Communication implementation overview

The nodes in the system communicate by sending datagram packets to each other. A node consists of the Chord layer and the application layer. The Chord layer handles all low-level communication. All incoming packets are separated into *requests* and *replies*. Requests are passed on unconditionally up through the system. Requests related only to Chord operation, never leave the Chord layer. If a request carries application level data, this payload is delivered to the application level entry point, which is the class *server.ChordCommunicator* or the class *ChordInteractor*. The former is used by servers, the latter by clients/users. When the payload has reached this far, the Chord layer forgets about it, and further actions are entirely up to the application layer.

Replies are not automatically passed up through the system. They must be explicitly asked for. This is done by matching of message sequence numbers. When a reply is fetched, it is deleted at the Chord layer. Replies that are not asked for, are eventually deleted as well. When a request is sent, a sequence number is generated for it, say `20`. To receive the reply to this request, the call `getReply(20)` must be issued. This call can be read as "get the reply to request with sequence number 20". Sequence numbers are only unique for a single node. To be able to uniquely identify a request, both the sender node and the sequence number must be used. Note that each reply holds both its own sequence number and the sequence number of the request it is a reply to. A reply can also serve as a new request, this depends on the application communication protocol (what meaning the messages have).

6.3.2 Chord

The Chord protocol requires a high volume of messages to be sent. However, when we started implementing the Chord protocol (during [Waa04]), raw performance was not in our mind. All Chord messages are implemented as subclasses of a base message class, `chord.communication.messages.Message`. These are converted to a byte array representation and included in a Java datagram packet (`java.net.Datagram`). The typical size of a message in byte array representation is about 600 bytes (no application label or data). We may have included some information in the messages that are not strictly necessary, for instance the address of the sender. This can be obtained through the Java API, but were added for convenience early in the development process.

Without going into full detail of the Chord protocol (consult [SMLN$^+$02] for that), the two most important messages are `NotifyMsg` and `GetSuccessorListMsg`. On a close second place comes `GetPredecessorMsg`. These three messages are used as a tool for the maintenance of the routing structures; the predecessor reference and the successor list (may be of size one). Resend is used to handle lost messages, but the design is based on the assumption that most messages arrive at their destinations (low loss rate). `NotifyMsg` does not generate a reply, but lost messages are not that important, as a new message will be sent shortly (current default delay is 200 milliseconds).

Each message is assigned a unique sequence number within the sending node. Note that the counter will eventually overflow, but it is not likely our prototype will run for such a long time! Replies are fetched by matching a given sequence number to the field *replyToSeqNr*. A timeout value is used to avoid the call to hang indefinitely if the reply message is lost during transfer (or not generated at all).

The Chord subsystem has two main threads running. One thread is running the `UDP-Communications` instance implementing the `Communications` interface required by the `Chord` instance. The other thread is running the `Chord` instance itself. Incoming messages are moved from the one thread to the other by using a message queue. Only complete messages are delivered from `UDPCommunications`, which imply that multidatagram messages are handled and assembled there. Large application messages should in general be embedded in a `ClientMsg`, as using an important message can cause the timeliness property to be broken for the Chord message (for instance a `find-Successor`).

Node space wrapping

As Chord nodes organize into a ring, node space wrapping must be handled. It is a special case, and should normally just occur for a single node in the ring; the one with the highest hash. To check if you need to perform wrapping, the hashes of the current node and the successor are compared. For all nodes except one, the hash of the current node should be smaller than the hash of the successor. If not, the current node is the node with the highest hash in the system, and the successor is the node with the lowest hash. On the implementation level, wrapping is handled in the following locations:

- `notify`-method in `chord.Chord`.
- `isInEE` and `isInEI` methods in `chord.Chord`.
- `findSuccessor`-method in `chord.Chord`.
- `getClosestPrecedingSuccessor`-method in `chord.SuccessorList`.

Note that failing to handle node wrapping properly in the routing code can cause infinite routing loops, which drain system resources. It can be hard to stop such loops as well, as taking down one of the nodes involved will be counter-measured by the system by replacing the node that left with another one. A time-to-live field in messages could be added to solve the problem. This value could be rather low, as the Chord algorithm provides bounds for the number of hops required to correctly route a message (consult [SMLN$^+$02]).

6.3.3 Client to server

Communication between a client and the peer-to-peer network takes place by using a *proxy* server node. All messages are sent to this server node, which in turn use the internal routing protocol to route the requests. When a request has arrived at its target and been carried out, the target node sends the reply directly back to the client. The address of the client is kept as field `requestor` in the messages exchanged.

All database related operations are located in *no.ntnu.dbdb_p2p.common*. When a client wants to issue a request, it first creates a operation object. This is sent to the `ChordInteractor`, which packages the operation object as application data inside a

Chord `ClientMsg`. An application data label is associated with the application, and the label triggers delivery within the application layer code. It is also in this way the application know what data to expect in the shipment.

6.3.4 Server to server

Besides from the Chord level traffic between server nodes, there is little traffic between server nodes in a system with low churn rate. In the current implementation, server to server communication at the application layer is concerning data maintenance protocols. These are described in Chapter 6.5 and 6.6 (and also in Chapter 5 on design). In addition, server node neighbors are exchanging messages related to database operation replication (explained in Chapter 6.7).

Although implemented at the Chord level, we must mention one important change we had to make. The Chord protocol is implemented in a way that make routing requests stop at the node immediately preceding the target node. This caused application messages to be delivered to the wrong node. To fix this, Chord forwards the application message to the target node in a `TransportMsg`. The problem described above only applies to situations where the application message is piggy-backed on a `FindSuccessor` message.

As most server to server communication occur between Chord neighbors, direct communication is used. There is no need to use the `findSuccessor` method when executing data maintenance protocols, as they are all targeted for a specific node. Direct communication between server nodes must be done by embedding the application messages in `TransportMsg` messages. `TransportMsg` messages are never routed, Chord simply accepts them and deliver the embedded application messages.

The server nodes operate with separate sequence numbers on messages they generate. The reason is that each server node has a message dispatcher thread, and the Chord sequence number is not known until a message is actually sent. Therefore, a mapping between server internal sequence numbers and Chord sequence numbers is used to make it possible to fetch replies at the application level. As some messages do not generate reply messages at the receiving nodes, the mapping must be cleaned up now and then to avoid old mapping entries to fill up the memory.

6.3.5 Datagram packet structure

One of the consequences of using Chord in a peer-to-peer network to implement a distributed database system, is that a message may have to flow through a number of hosts before it arrives at its destination. The time spent at intermediate hosts has to be reduced to a minimum to achieve acceptable response times. The Chord implementation is written to handle message objects, where application messages are possibly enclosed within. This forces the whole amount of data to be processed even when only a small part of it is needed to decide that the message must be forwarded to another node. The processing consists of recreating a number of objects through a `ByteArrayInputStream` and a `ObjectInputStream`. The recreated object is then cast to its correct class. The structure of the message object is so that no information can be retrieved without processing the whole message.

To avoid unnecessary work, the structure of the datagram packets should be changed. A proposal for a new message structure is described below. It was not implemented because the new structure was not thought of until late into the project. There was no time to allow for the required amount of testing and then run the necessary benchmarks afterwards.

The new structure consists of some important information stored separately from the message object. The application is capable of deciding what to do with the message object without processing the message object itself. A table representing a datagram packet payload is shown in Table 6.2.

The components of the packet are:

- **Message type:**
  A single byte representing the type of the message. This is used as a "sneak peek" of the message object.
- **Id/hash:**
  The hash of the node the message is sent to, or the hash of the operation the message carries. Based on this hash and the message type, the application node should be able to determine if the message has reached its destination or if it must be forwarded to another node. The hash is represented as a `java.math.BigInteger`-object. The maximum value is set to 2^{160}, which is represented with 21 bytes in the datagram packet. The byte representation is obtained with the method `toByteArray` of `BigInteger`.
- **Message object:**
  The Chord message object. If there is an application message, this will be enclosed within this object. The information carried in the object depends on the message the object represents. All Chord messages are defined in the package `chord.communications.messages`.

With the structure described above, an intermediate host should be able to take the correct action based on inspection of 22 bytes. In addition, a `BigInteger`-object must be recreated and used in the `findSuccessor`-method. The new message object might require some minor changes in other methods, for instance the findSuccessor-method. In addition one might add some error detection or correction mechanisms, but this

1	Message type
2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22	Hash
23 .. NN	Message object

Table 6.2: Proposed improved datagram packet structure. The first byte is labelled `1`, not `0`. *NN* varies with the Java virtual machine implementation and can be found with the method `getSendBufferSize` in `java.net.DatagramSocket`. *NN* was found for two different platforms; 8192 (Windows XP, Java 1.5, AMD64) and 55296 (Debian Linux, Java 1.5, AMD32).

depends on the usage of such mechanisms in the underlying communication layer (Java implementation).

6.4 Events and the event publishing system

An event publishing system is used to let the application level know about possibly interesting events happening in the underlying Chord subsystem. It can be considered a simple publish-subscribe system. All events are tunnelled from the Chord subsystem through the class `server.ChordCommunicator`. Interested parties must register themselves with this class.

As it turned out, the event publishing system was suited for delivering all kinds of incoming data, including database operation requests. Every request has an application data label attached, and the value of this label is used to deliver the application data itself to the correct processing entities. There can be any number of registered listeners. A listener must implement the interface `server.MessageHandler`, which consists solely of the method `deliverMessage`. Although not enforced at the implementation level, this method should be very fast, or else it will hog the `ChordCommunicator` thread and prevent this and other messages to be delivered timely. The typical behavior is to insert the message into a message queue and let another thread take it out and process it.

Below is a list of the labels used, along with a brief explanation. Note that instead of using the actual textual value of the label, we use the name of the constant used instead. The values of the constants can be found in `common.Constants`.

- **MSGHOOK_DBOP**
  A database operation. `DatabaseManager` is the only handler for these requests. Any server node can send such a request, either directly on behalf of a client or as a part of the message routing (forwarding).
- **MSGHOOK_PRED** (chord:predecessorchanged)
  A new predecessor has been detected by the Chord subsystem. This event is generated locally. It is handed over to `ServerRequestProcessor` for processing (see Chapter 6.5.1 and 6.6.1).
- **MSGHOOK_SUCC** (chord:successorchanged)
  A new successor has been detected by the Chord subsystem. This event is generated locally. It is handed over to `ServerRequestProcessor` for processing (see Chapter 6.5.2).
- **MSGHOOK_RELOC**
  An incoming data relocation batch. This event is generated by the succeeding node as part of the relocation protocol. `ServerRequestProcessor` is responsible for handling it.
- **MSGHOOK_RELOCDEL**
  A request for deletion of old replica data. This deletion is selective, so the key of each record to be deleted is included in the request, and a single request concerns only a single database.
- **MSGHOOK_DBREP**
  An incoming database replication batch. This event is generated by the preceding node as part of the database replication protocol. There are two registered handlers for this event: `ServerRequestProcessor` and `DatabaseManager`. The former is responsible for processing the request. The latter is required to perform synchronization, as a database replication request cannot be accepted when data promotion is going on (see Chapter 6.6.1 and 6.5.2).

- **MSGHOOK_DENIED**
  Message used to inform a client or a server that a request has been denied processing. This is due to other ongoing activities on the node, and the situation is temporary. Its occurrence simply means the request in question should be retried.

- **MSGHOOK_REP**
  An incoming replication request for a single database operation. `ServerRequestProcessor` is responsible for handling it, but methods in `DatabaseManager` is actually used to process it. The node's predecessor is generating these requests as it receives database operations. Not all operations require to be replicated.

- **MSGHOOK_ADMIN**
  Messages that instruct the node to perform administrative tasks. The following tasks are implemented:

  - **ADMIN_DONOTHINGSTATE**
    A message that tells a node to enter the "do-nothing" state, see Chapter 6.9 for details.
  - **ADMIN_CLOSEENVS**
    Instructs the node to close all environment handles to force all data to disk. Normal operation can be continued afterwards, but environments must be reopened (causes delay).
  - **ADMIN_NODECOUNT**
    This is used to obtain a count of the nodes in the system. It can also be used to obtain a list of all the nodes in the system. See Chapter 6.10.2.

  The labels listed above are contained in a message object of type `AdminMsg`.

Some of the requests above can be paired. For instance, a `MSGHOOK_SUCC` event on one node will result in one or more `MSGHOOK_DBREP` on another node (the node's successor). Also, some events are only generated locally. So far, this applies for Chord related events. These should in general be validated before they are acted upon, as an unstable Chord ring can result in data and ring maintenance protocols being initiated on false premises. This situation is hard to handle. On one side, using a "lengthy" validation process will cause the system to react slowly on changes in the Chord ring organization. For some events, like a node failing, this will result in temporary routing errors and loss of data (replica data not promoted yet). On the other side, reacting immediately can cause a higher rate of denied requests and incorrect relocation of data. This situation requires some sort of *roll-back* mechanism to regain a consistent system state. The main point to note, is that the application level is dependent on a stable and well-behaved Chord subsystem to operate correctly.

6.5 The take-back protocol

As described in the design chapter, data must be moved when a new node joins the Chord ring. This process is termed take-back. In conventional database systems, the term is used when a primary node goes down and then need to take back its responsibilities when it comes back online. In our system, take-back is performed for all nodes joining the system, but the amount of data ”taken back” is dependent on the hashes of the node and the data already present in the system. Unlike most non-peer-to-peer systems, take-back must be performed in our system, as the routing scheme will fail if data is located at incorrect nodes. The take-back protocol consists of two main steps, described in the following subsections:

1. Data relocation
2. Data replication

The steps are both initiated at other nodes than the one joining the system, but the new node is the target node for the operations. Data relocation is initiated by the new node's successor, while data replication is initiated by the new node's predecessor. The two steps can be performed in parallel, and should not require synchronization on the new node. However, both steps compete for the same resources (network capacity, CPU and disk access), so we chose to perform one at a time. This also allow us to use a single thread per server node for data maintenance protocols.

The take-back protocol is illustrated in Figure 6.5.

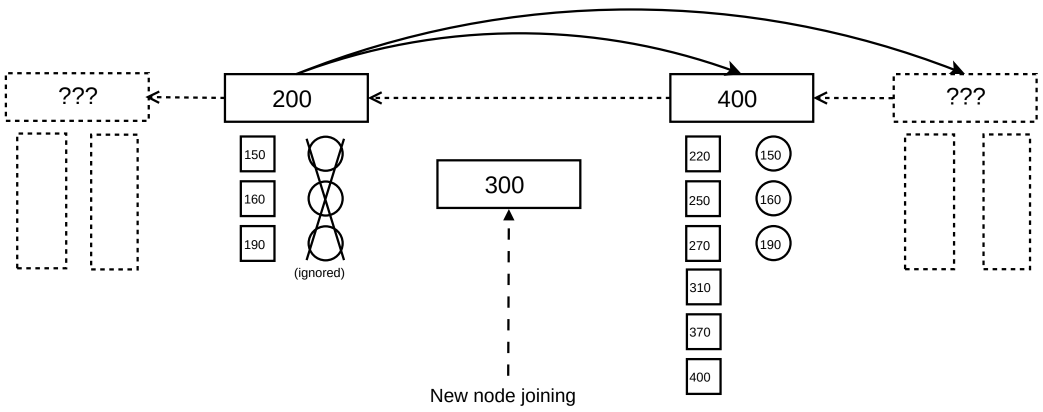
6.5.1 Data relocation

When new nodes join, existing data in the network may have to be relocated. This is vital to correct operation, as routing is based on the hashes (identifiers) of the server nodes and the records. The only Chord event that may cause relocation of data, is that of a node getting a new predecessor. When this happens, the node must transfer all its database records whose hash is smaller or equal to that of its new predecessor.

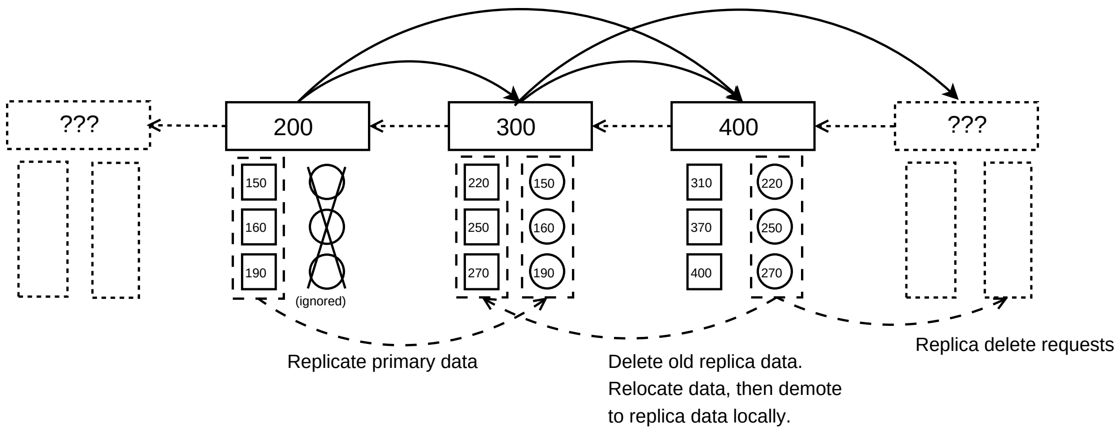
The following classes are relevant for data relocation:

- `server.ServerRequestProcesser`
- `server.DataRelocator`
- `server.BatchTransferUtils`

This is how data relocation is handled (events not related to data relocation are ignored):



a) Stable state.



b) A new node joins.

Figure 6.1: Figure illustrating the take-back protocol, which is started on node joins. Note that some successor references are left out in the figure.

1. A new node joins the system, generating the event *chord:predecessorchanged* at the succeeding node.
2. The event is delivered to the `ServerRequestProcessor` thread.
3. The event is processed and "validated".
4. The relocator object is used to start the actual relocation, with the new predecessor as argument. The following actions are performed until all data is processed:
   (a) Identify records to relocate and build a batch transfer unit.
   (b) Transfer batch transfer unit to predecessor.
   (c) Move relocated records to replica database.
   (d) Delete relocated records from primary database.

Step 3 is necessary as there are two situations generating a *chord:predecessorchanged* event: A new node joined and an existing node left. The preprocessing step determine which of the situations we are seeing. If the new (in sense of changed) predecessor has a hash that is bigger than our previous predecessor, a new node has joined the system. If the hash is smaller than that of our previous predecessor, the previous predecessor left for some reason. Both the address and hash of both the new and previous predecessor are delivered along with the *chord:predecessorchanged* event. The Chord subsystem is responsible for obtaining them.

As stated in the above list, the records to be relocated are picked out in step 4a. The relocation criteria is simple: If the hash of the record is smaller than or equal to the hash of the new predecessor, it is to be relocated. To increase the effectiveness of the relocation process, records are collected into a record batch before they are transferred over the network. When the batch has exceeded a certain size, it is transferred. The size threshold is specified by the constant *BATCH_SIZE_THRESHOLD* in `common.Constants`. A single batch only holds records for a specific database and environment. This allow us not to store the names of the environment and database for each record.

The final step in the relocation process is to demote the primary records that were relocated to replica records, as we are now the replica node for these records.

It should be noted that some extra logic is required to handle the special case of relocation in a system consisting of only two nodes. We felt it was important to make this possible, as testing and exploration of the prototype is likely to happen in a small system configuration.

Currently, there are no synchronization mechanism dealing with multiple, concurrent new predecessors. *The behavior of the system in a situation where a new predecessor is introduced before the previous relocation is completed, is not defined.*

6.5.2 Data replication

There are two types of data replication; individual database operation replication and primary database replication. The former is described in Chapter 6.7. In a stable system, all primary data will be replicated as data is manipulated through database operations. When a node joins, or an existing one fail, the replica data is considered lost. In a stable system (assume only two nodes for simplicity), the replica data for node 1 is correctly located at node 2. After node 3 joins and is inserted *between* node 1 and 2, the replica data is incorrectly located at node 2, as this node is not the successor of node 1 anymore. To correct the situation, replica data for node 1 must be generated at node 3. This is best done by letting the primary node (node 1) replicate its own data. Letting the replica node move the replica data from itself to the new node is not a good idea, as it can lead to synchronization issues (record overwrite, missed deletes) if the primary node receives request during the replication process. Also note that the replica data on node 2 must be deleted as a consequence of the node join.

Data replication is initiated by the event `chord:successorchanged`. If the reason for the event is that the previous successor failed, the replication protocol can be postponed. This happens when the new successor is not done promoting existing replica data (see Chapter 6.6.1), and replies to the replication request with a request denied message. When receiving a request denied message, the thread simply waits for a little while (currently 2 seconds, see constant `REQUEST_DENIED_BREAK` in `server.BatchTransferUtils`). For each delay period, the original request is resent. There is also a timeout value in case something goes wrong at the succeeding node.

Data replication is carried out in the following steps on the initiating node:

1. For each database, repeat until all records are replicated:
   (a) Create record batch (size dependent on configuration parameter *BATCH_SIZE_THRESHOLD*).
   (b) Transfer batch and check the number of records inserted at successor node.

During the process, incoming database operations are denied. Operation replication requests from the preceding node are accepted and carried out, but must compete with the primary database replication process for processing resources.

On the receiving node, each replication request, which is a record batch, is processed independently. Each record in the batch is inserted into the replica database. The name of the replica database is generated by the receiving node, which allows us to use different prefixes at different servers. This is no goal in itself, but a higher degree of node autonomy is important in a peer-to-peer system. The number of records successfully inserted is sent back to the initiating node as an acknowledgment for the record batch. If some records could not be inserted, they are ignored and a warning message is logged. If a node is operational, there is no reason why a record should not be inserted. Issues like deadlock is handled, but there are no handling of problems like running out of disk space or faulty hardware. Nodes experiencing such problems should actually be forced to shut down and leave the Chord ring.

6.6 The take-over protocol

The take-over protocol is started when Chord detects that the node has gotten a new predecessor (event *chord:predecessorchanged* and this predecessor has a hash that is smaller than that of the previous predecessor). The goal of the take-over protocol is to reestablish the primary data that were present on the node that failed, and to maintain the replication degree. Unlike the take-back protocol, the steps in the take-over protocol must be performed in a specific order to avoid mixing up replication data from two different nodes. As the two steps are initiated on two different nodes, synchronizing the steps is a little harder. This is elaborated in Chapter 6.6.1 on data promotion.

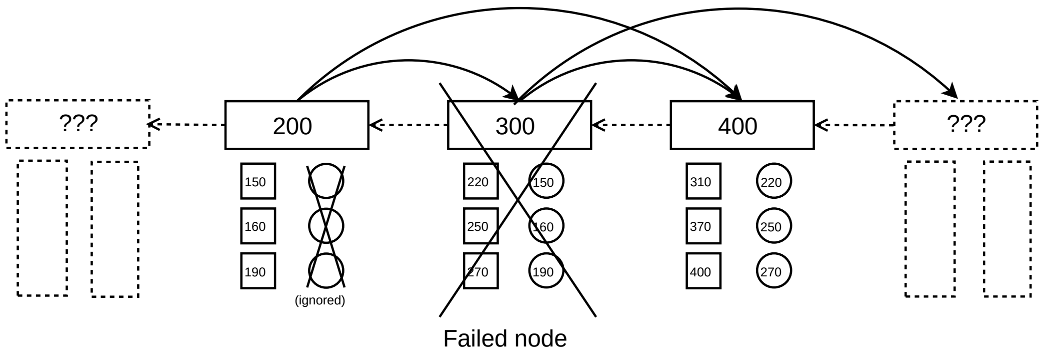
The take-over protocol is illustrated in Figure 6.6.

6.6.1 Data promotion

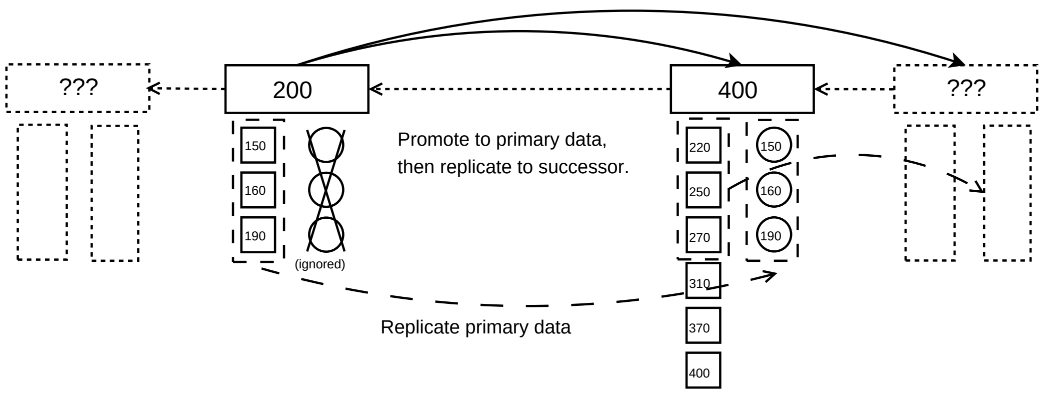
The process of promoting replica data to primary data is dependent on the way we separate replica and primary data. We chose to prefix environment directories (see Chapter 5.9.1 for a discussion on alternatives). To promote, we must do one of the following:

- Rename directory.
  The is by far the easiest and most effective approach, but we are often restrained from using it by the existence of the replica environment's corresponding primary environment. There is no way to merge two environments on the file system level.
- Read and write each record.
  This approach must be used if the node keeps both primary and replica data for the same environment. We read each record from the replica database, insert it into the primary database. The final step, after all records have been promoted is to delete all replica environment directories. Instead of deleting each record as we go, we delete them all at the file system level in the end. This is faster.

As mentioned, we must make sure data promotion is finished before new replication data is accepted. First of all, we do not want to mix replication data from two different nodes. Second, the deletion step above would remove data we still need. Data



a) A node fails.



b) State and actions after node failure.

Figure 6.2: Figure illustrating the take-over protocol, which is started on node failures. Note that some successor references are left out in the figure. Note that there are two replication processes taking place; one from node 200 to node 400, and one from node 400 to the rightmost ???-node.

promotion is initiated from `ServerRequestProcessor`, and the object keep track of whether a data promotion is ongoing or not. However, when this object is performing the promotion, it cannot at the same time take care of synchronizing the promotion and replication protocol. We let the `DatabaseManager` do this, as these two objects are run in separate execution threads.

The event *MSGHOOK_DBREP* is delivered to `DatabaseManager`, which checks if a data promotion is ongoing by calling a method of `ServerRequestProcessor`. If this is so, it sends a request denied message to the requesting node (the predecessor). If the other thread claims it is ready to accept the replication request, `DatabaseManager` waits for small amount of time and then checks if the condition still holds afterwards. If this is the case, it queues the replication request with the `ServerRequestProcessor` and it will be processed shortly. The small wait period is necessary to guard against timeout values, differences in node performance and network delay between the current node, its predecessor and the failed node. Remember that data promotion is initiated by the *chord:predecessorchanged* event on the current node, while replication is initiated by the *chord:successorchanged* event on our new predecessor (which was the predecessor of the failed node).

In addition to the data replication mentioned above (and in Chapter 6.6.2), there is another replication process that must be carried out. When a node promotes the local replica data to primary data, this primary data is no longer replicated anywhere at all. To correct this, the node must replicate the data at its successor. This replication is considered part of the promotion protocol, and must not be confused with the other replication protocol described in Chapter 6.6.2. It is the latter replication that must be synchronized between the predecessor and the successor of the failed node.

6.6.2 Data replication

This step has already been described in Chapter 6.5.2. The process is the same, but it is important that data promotion is finished before the replication starts. If not, replication data from different nodes will be mixed up. It is possible to solve this without delaying the replication process, simply by keeping the old and the new replica data separated until the old data has been promoted. The synchronization mechanisms used during take-over are explained in the previous chapter on data promotion (Chapter 6.6.1). Note that this step is the replication of primary data on the new predecessor, and that it is initiated at the predecessor. The replication included in the data promotion protocol is initiated by the local node and is targeted at the successor node. If we exclude the node that failed, three nodes are involved in the take-over protocol.

6.7 Database operation replication

Database operation replication must be performed to keep the primary and replica data consistent. The node executing the operation on the primary data has the responsibility of issuing the operation to be performed on the replica data as well. As stated earlier (see Chapter 5.9.1), replica data is to be held at the primary node's successor.

Not all operations are to be replicated, for instance operations that do not change records. To determine if an operation is to be replicated or not, the method `isReplicable` of `common.DbOperation` is used. The system configuration is also checked to see if replication is enabled or not. Last, a small detail: If there is only one node in they system, database operation replication is disabled until another node joins.

The replication is carried out by forwarding the operation request with the label MSGHOOK_DBREP attached. This request will be delivered to the `ServerRequestProcessor`, which derive the name of the replica database and then use methods in `DatabaseManager` to execute the operation. When it is done, a reply is sent back. This includes the status and any exceptions that may have been raised during execution. To save a little time, the request for replication is sent first, then the operation is executed locally and at last the results of the local and remote operation is compared. There is no support for roll-back in our prototype, but this can be solved either by using the transaction service in JE or by implementing a simple mechanism of your own (for example let the operations generate a roll-back operation for their own actions).

6.8 Debug logging

Logging for debug purposes is implemented by using the `java.util.logging` framework. The choice of logging framework was primarily made to avoid introducing more dependencies to external software. The logging done in the system is pretty straightforward. Most classes use parts of their classpath and the class name to name their loggers. Note that the classname is converted to lowercase. For instance, the class that start the server would use the logger `dbdb_p2p.server.server`. As can be seen, the elements `no.ntnu` have been removed for consistency. Note the following exceptions:

- `dbdb_p2p.client` - the classes in the `no.ntnu.dbdb_p2p` directory add a client tag to their loggers.
- `dbdb_p2p.threading` - all log statements related to threads use this logger.
- `chord` - the Chord subsystem.

The following properties can be used to control the log output:

- `dbdb_p2p.logconfigurationfile` - specify a file that contains log configuration items.
- `no.ntnu.dbdb_p2p.common.FileLogger.location` - specify where the log file should be created. If this property is set, the file logger is enabled.
- `no.ntnu.dbdb_p2p.common.FileLogger.level` - specify the log level for the file logger.
- `java.util.logging.ConsoleHandler.level` - specify the log level for the console logger.

The three last properties *must* be specified in the log configuration file specified by property `dbdb_p2p.logconfigurationfile`.

Note that the level specifications above does not control the logging of individual log statements. They only control the log statements that actually reach the file or console logger. Log levels for specific loggers can be specified in the log configuration file. This file must follow the standard Java properties file syntax. The loggers are configured to use the levels of their parents if no level is explicitly specified for them. The log file below would log all client related records, only records with level INFO or higher for the server and threads and only warnings for the Chord subsystem. Note that the level for the handle manager class is overridden and set to a different value then the rest of the server loggers. If further control over what is logged is needed, a filter can be applied to some or all of the loggers. A filter is a simple class that only need to implement a single method, `isLoggable(LogRecord record)`. To use it, make sure it is in the classpath and specify it for the loggers and/or handlers that is to use it with a property string in the log configuration file; `dbdb_p2p.server.handlemanager.filter=MyFilter.class`. As always, do not forget that post-processing is a powerful technique!

```
dbdb_p2p.client.level=ALL
dbdb_p2p.server.level=INFO
dbdb_p2p.server.handlemanager=FINER
dbdb_p2p.threading.level=INFO
chord.level=WARNING
```

The file logger will log to a specified file. Only the location of the file can be specified by the user, through the property `no.ntnu.dbdb_p2p.common.FileLogger.location`. The filename is composed by the date and time of creation of the file logger instance, the port number of the server/client and the extension *.log*.

Note that the log statements in the code can cause a slight decrease in performance, even when logging is disabled. It might be wise to write a script that removes log statements when doing accurate performance tests. All log statements are written in a consistent way; `this.logger.level(logtext);`. `level` is a valid log level (consult Java API documentation - `java.util.logging.Level`) and `logtext` is any valid text statement. Doing performance tests with all logging enabled is no use, it will yield invalid results!

The available loggers are listed in Appendix E.

6.9 The "do-nothing" state

Normally, taking down the system would result in that all data in the system will be located at a single node, as this is a consequence of the data maintenance protocols when node after node leave the system. This made it very difficult to inspect the local databases of the servers to check for consistency and errors in the data maintenance protocols. Therefore, a "do-nothing" state was introduced into the system. When the system enters this state, all requests and events related to data maintenance will be ignored. As a result, the local databases will be kept untouched as we take down node after node. All environment (and their databases) will be closed and flushed to disk as soon as the state change message is received.

The do-nothing state is implemented to make it terminal. If a node has entered do-nothing state, it cannot exit it without being restarted. The motivation for this, is that one of more nodes in a do-nothing state can cause invalid placement of data from which the system is unable to recover. From the clients' point of view, this is loss of data.

To place a system into the do-nothing state, use the tool `devutils.EnterDoNothing-State`. This will send a `DoNothingMsg` the specified start node, which will forward it to its successor. This continues until the whole ring has been reached. Finally, the last node, which is the predecessor of the start node, will send an acknowledgment to the tool. It is also possible to put just one or a subset of the nodes in a system into do-nothing state, but this is not recommended. The message being sent has a time-to-live field to avoid infinite looping, which can happen if the start node goes down after the message has been sent.

Note that inspection of databases on disk can be a little tricky. First of all, the environment directories must be copied to another location to allow for exclusive write access, which seems to be necessary to collect statistics for an environment. Further, there may have been database changes that are not yet flushed to disk. This process is controlled by a Berkeley DB maintenance background thread. To force all changes to disk, use the tool `devutils.ForceDatabasesToDisk`. There are some precautions to be taken when using this tool as well, these are described in Chapter 6.10.2.

6.10 Testing and exploring the prototype

This section will give some pointers for how to try out and explore the prototype system. We start by describing some tools for the Chord subsystem, then continue with the application level.

6.10.1 Chord subsystem

With the developer tools for the Chord subsystem, you can create a Chord ring and see how it evolves as nodes join and leave the network. The tools are very simple and only provide information on the state of the local node. The most useful tools are described below:

- `chord.devutils.GenerateHash`
  This simply generates the hash for a node with the specified IP address and port. Although convenient, the tool `no.ntnu.dbdb_p2p.devutils.CalculateNodeOrder` (see Chapter 6.10.2) is better, as it can handle many node specifications at once.
- `chord.devutils.UDPChordNode`
  Starts a Chord node and joins the network through the specified bootstrap node. Output is printed to the console, but it is not possible to interact with the node or network in any way.
- `chord.devutils.UDPChordInteractionNode`
  Starts a Chord node and joins the network through the specified bootstrap node. The user is taken to an interactive shell. It is possible to issue a number of commands to the Chord node. The commands available are listed in Appendix F. This tool can be used to verify that specific keys are routed correctly, and also to verify the routing structures found in Chord (predecessor reference, successor list and finger table).

6.10.2 Application level

The prototype is capable of accepting a set of database operations. To aid the development process, a number of simple applications were developed. First we describe how to start a server and a interaction client. Then we introduce some of the tools available.

A server can be started with the following UNIX command line:

```
java -cp .:../software/chord.jar:../software/je.jar
no.ntnu.dbdb_p2p.server.Server server.cfg
```

To start a server on a Windows platform, simply substitute colons with semicolons and slashes with backslashes. The command line above require all mandatory information to be specified in a file called `server.cfg`:

- `chord.udp.ip`
- `dbdb_p2p.server.env_home`.

You will also use `chord.udp.port` and `chord.bootstrapnode` most of the time. For a full list of available configuration properties, consult Appendix D. Note that configuration properties can also be specified on the command line directly. These will override the ones specified in the configuration file. To make a node join a network, specify the address of one of the existing nodes in the ring with `chord.bootstrapnode`, for instance `chord.bootstrapnode=10.0.0.10:20000`. Also note that the server environment home must exist before the server is started (an error message will be printed if it is not).

If an ordered list of the nodes, and their hashes, in the system is wanted, use the tool `devutils.CalculateNodeOrder`. All system nodes must be specified on the command line.

A description of other useful tools is given below:

- `examples.StringRecordInteractor`
  This tool lets the user issue operation requests in a simple interactive shell. It supports the operations `put`, `get` and `delete`. Both keys and values are expected to be `String` objects. Data of other types will cause exceptions. Each operation is timed, and the server that served the request is shown. This can be used to verify that operations are routed correctly, and that data is actually stored at the nodes. Behavior during ring changes can also be studied to a certain degree (manually).
- `examples.PutNumberSequenceRecords`
  Inserts a sequence of integers. Starts at 1 and continues until the specified upper limit is reached. The integer is represented as a `String` and used as both key and value.
- `examples.PutRandomStringRecords`
  Inserts a specified number of random string records into the system.
- `devutils.ForceDatabasesToDisk`
  A utility to send a message that requests a node to flush all its databases to stable storage. The message can be sent to all nodes in the system, or to a single node. Normal operation can continue after this utility has been used. Note that all handles will be closed to force all data to disk. As a consequence,

environments must be opened again on the next operation request. The operation is not designed to be executed while data maintenance protocols are active (at node joins and failures).

- `devutils.EnterDoNothingState`
  Tells the system (or a single node) to enter a state where data maintenance requests are ignored. This is necessary when nodes in the system are taken down and the data on disk is required to be kept in the state it was in before the system shutdown started. The do-nothing-state is terminal; it cannot be exited without restarting the nodes. Putting a single node, or a subset of the nodes, in this mode will cause the system to malfunction as the data maintenance protocols are not executed.

- `devutils.EnvironmentStats`
  Prints the names and number of records in all databases in a specified environment. If this is to be used on environments that are accessed by a running server instance, the directory must be copied to another location first. This is because the method collecting statistics require exclusive write access to the while environment. Also remember to force data to disk with `devutils.ForceDatabasesToDisk` before copying if the latest database state is to be reflected on disk.

- `devutils.ObtainNodeCount`
  Attempts to obtain the node count in the specified system. If the argument `--node-list` is specified, the tool will generate a list of the nodes in the system in addition to the node count. Note that the algorithm used will only work on stable networks.

- `devutils.OperationTimer`
  Inserts a sequence of integer records into the system and collect the response time and status for all operations. The results can be printed to the console or written to a file. The tool will also read back all records by default. This can be overridden with the switch `--put-only` (or `-p`). Further, the results can be analyzed with `OperationTimerOutputParser.py` (a Python script).

- `OperationTimerOutputParser.py`
  This Python script is located in `devutils`. It analyzes output generated by `devutils.OperationTimer` and prints a table showing simple statistics for the nodes in a system. An example analysis is included in Figure 6.10.2. The reason for hop 0 having the highest response time, is that the requests for this hop must travel all the way around the ring.

If the system is to be run on a cluster of computers, most tasks must be automated. Some scripts for the `BASH`[1] scripting language, available on Linux and Unix platforms

[1]GNU BASH - a Unix command language interpreter, called a shell.

```
>>> General info
System nodes: 10
Operations executed: 1000000
Operation types: put; 50.0%    get; 50.0%
Operation successrate: 100.0000%, (0 failed operations)
Operation denyrate: 0.0000% (0 attempts denied)

>>> Per hop statistics
                                                    Comparison
Hops    Count   Mean    Median  Min     Max         proxy mean
0:      53014   9.5     3.8     2.3     131.7          100%
1:      73564   5.0     4.7     3.1     154.4           52%
2:      26862   7.1     6.7     4.5     136.4           75%
3:      7554    7.4     7.0     5.2     88.8            77%
4:      105058  6.5     6.1     4.6     192.2           68%
5:      16690   6.7     6.3     4.2     60.6            70%
6:      210382  7.6     7.2     5.3     143.0           80%
7:      355976  6.8     6.4     4.6     299.5           71%
8:      106524  7.7     7.3     5.6     426.9           81%
9:      44376   8.3     7.7     6.3     120.4           87%
```

Figure 6.3: OperationTimer output analysis. The reason for hop 0 having the highest response time, is that the requests must travel around the whole ring. The size of the successor list is two in this example. Response times are in milliseconds.

(and some others), has been written during the projects. These can be found in the root `devutils` directory, in the subdirectory `endless_scripts`. Note that these were written on an ad-hoc basis, and is not aimed at general usage. They are also made specifically for the cluster for testing during this project. If they are to be used, they must be adjusted and reviewed.

6.11 Things to note

The list below gives a number of facts related to the behavior of the prototype. They are included to enlighten anyone that wishes to use or develop the application.

- If the first node is started with data in its base environment directory, replication data will not be promoted to primary data on startup.
- If several nodes are started with existing data in their base environment directory, the nodes will enter a deadlock state. This is because each node has only one thread handling inter-server requests. Both node will send data to the other, but none is able to process it and reply. This can be fixed by using more threads, but then a more elaborate scheme for handle management may have to be added as well.
- The data maintenance protocols are not written to handle unstable nodes. If a node fails while involved in execution of a data maintenance protocol, data may be lost. The problem is rooted at the application level.
- Occasionally a single put operation of a large batch is reported timed-out when using the development utilities that insert records. However, the operation is executed. This seems to be bug related to a missing reply/acknowledgment. It cannot be related to lost datagram packets, as the same records seem to time out. Also, a different record times out if the number of records inserted is changed (for example from 100 000 to 1 000 000).
- The data maintenance protocols are design and implemented based on the assumption that they will always be successful (eventually). If a protocol fails, the system state may be inconsistent. There are no background processes that cleans up after failed data maintenance protocols, for instance verifying that all records are correctly located.
- The JVM should be allowed to use more memory then the default value. This is because JE uses the JVM memory for caching and to hold data for the environments. The memory size can be set with the property `-XmxYYYm`, where YYY is the memory size in megabytes.

Chapter 7

Tests and measurements

To confirm that the prototype does work, we ran some simple tests. We also wanted to obtain some basic performance metrics to allow for comparison with other systems. The test setup is described in Chapter 7.1. The following chapters document the results of the tests, and Chapter 7.7 give pointers to further testing.

7.1 Basic test setup

The project was given access to use a small cluster of computers located at NTNU. There were six computers in the cluster, with the following specifications:

- Dual AMD Athlon XP 1600 processors, 1.4 GHz
- 1024 MB of RAM (main memory)
- Disk access through a SCSI system
- 100 Mbps local area network connection

As the computers have two processors each, we allowed each host to run two server instances (port 20000 and 21000). The different nodes were organized as indicated in Table 7.1 due to their hashes (order computed with `devutils.CalculateNodeOrder`). Each node is given an integer identifier based on their order, and a letter to identify the physical machine it is run on.

The node ordering is illustrated in Figure 7.1. Node positions on the ring with relation to the node identifier space is approximated.

1	`15ba51a25a6b6ac6c3b7b6a4ea63c13b9d9a810`	A
2	`ab1762f043620e621e1bb137902234d6e75e71a`	B
3	`1436270280b7576c561dd90481595e4fa83b633b`	C
4	`1b0d2b3715a060cd47c2f7a823f8406d79bcc9e7`	D
5	`1d00b3af8f75af91b87b5a957c55e4aaf036aee1`	A
6	`20686a0738ff2f67c77ba449fdc054a56e4d43ee`	C
7	`37b97be7d521ea886d1587c489334ada920b611b`	E
8	`4cc0388913d8c01b4211483dbb9ac3dfa09e126c`	E
9	`a0b3a9063588ac88f8c7cc48f9ff2d9c2226c0c9`	B
10	`cd397d8d50a6fa74facb318e017707702e03221a`	D
11	`e85dde6c18420e02d1f1cc6ad24c668dae680b98`	F
12	`f3b2717ffae4e8bcd4bc8f21498e40304445a49c`	F

Table 7.1: Test setup node ordering. Note that some hosts may have a hash with fewer digits than others.

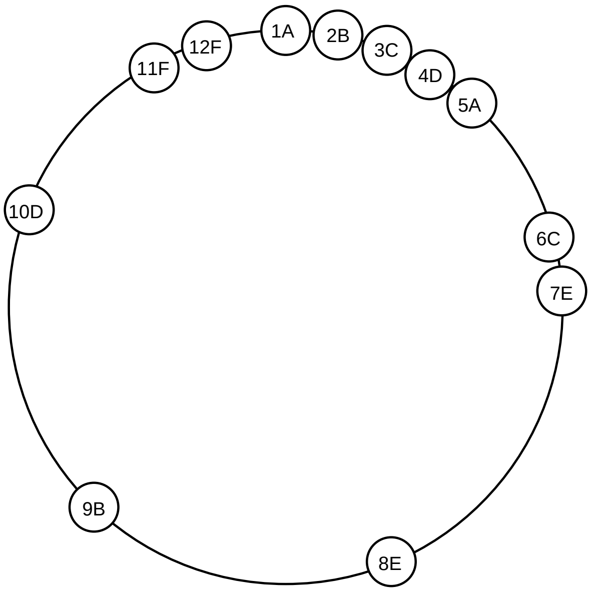


Figure 7.1: The Chord ring of our test setup. Node positions are roughly approximated with regards to the identifier space. The low number of nodes causes the identifier space to be divided in chunks of very uneven span. The letters indicate which physical computer the node is located at.

As can be seen, some of the nodes appear to be very close in the identifier space. However, the difference between these nodes are still numerically large. In a system with very few nodes, one can nevertheless get a situation where almost the whole identifier space will be the responsibility of a single node. As stated in various publications on DHTs, for instance [SMLN$^+$02], each node should run a number of *virtual nodes* to obtain a satisfactory degree of load balancing in most system configurations. This technique introduces more complexity to the software, and is not used in our prototype.

In our test setup, there are two occurrences where nodes running on the same physical machine are neighbors in the Chord ring. This is not good, but the chances of this happening in a large system are small. In general, one can assume that two neighboring nodes have independent failure modes. However, to assume that this property holds may not be good enough. It is possible to implement a mechanism in the application that checks if the predecessor or successor is on the same physical machine as itself, and change the node hash to be inserted at another place in the ring if necessary. If the successor list is large enough, Chord is able to handle cases where a number of neighboring nodes fail simultaneously, but the replication scheme at the application layer will be broken. The consequence is loss of data. The replication scheme is based on the assumption that neighboring nodes have independent failure modes.

Unless explicitly specified, the following steps are included in the response time measurements given in the following chapters:

1. Client sends the requests to the proxy.
2. The proxy routes the message. This might involve multiple node hops.
3. The target node processes the request.
4. The request is replicated if necessary.
5. The client receives a reply from the target node.

Unless stated otherwise, the test setup described in this chapter was used in the tests.

7.2 Operation response times and cost of node hopping

Node hopping is part of the routing protocol; as a node does not know the destination node for a request, it routes the message to a node it knows is closer to the destination than itself. Besides from the time it takes to send the message over the network, the message must be processed at each node. To get a picture of how node hopping affects the operation response times, we varied the size of the successor list in a system with 12 nodes.

Figure 7.2 shows the average response times for each hop for different test runs where the size of the successor list was varied. As can be seen, the size of the successor list greatly affects overall performance for the system. Setting the successor size equal to the number of nodes in the systems would take us back to where we started; a system using global views. According to [SMLN$^+$02], a lookup will with high probability require (at most) *O(log N)* hops. The average number of hops is given to be 1/2 $O(logN)$.

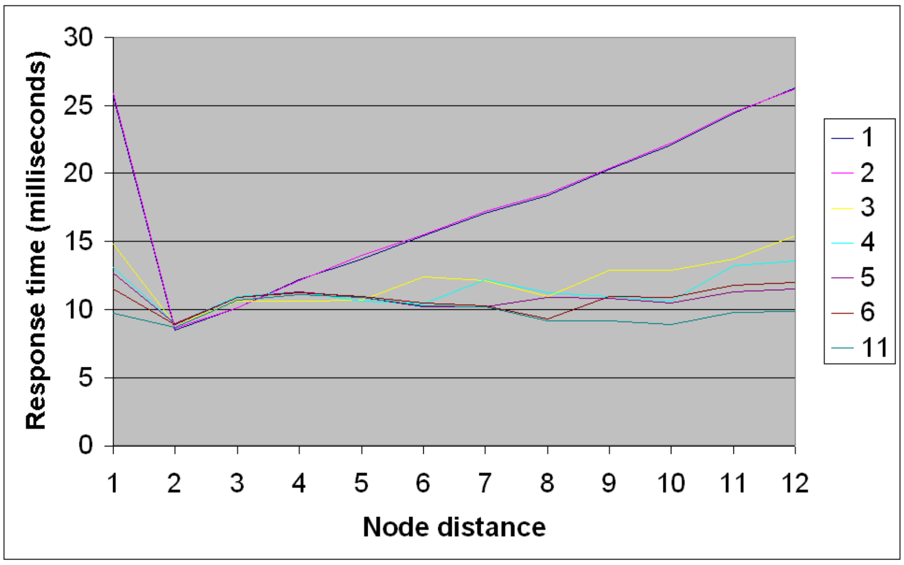


Figure 7.2: A graph of the average response times for each hops when varying size of successor list. Size of the successor list is given at the far right. The reason for the "spike" at distance 1, is that the request has to travel all the way around. This can be avoided by performing extra checks at the node.

7.3 Cost of node joins

To determine the cost of node joins, one must remember what definition of a node join the cost measurement is based on. We discussed two different definitions in Chapter 5.5.

When considering node joins at the Chord level, these are fast. It should take at most a few seconds before the basic routing structures are updated. The actual time it takes is dependent on how often the `stabilize` routing is run. Our prototype use every 200 milliseconds as default. Note that if a large number of nodes join and/or fail simultaneously in the same area of the ring, the time to stabilize the routing structures may be higher.

At the application level, the situation is quite different. As a node join requires data to be relocated, the time it takes for a node to join is dependent on the amount of data stored at the new node's successor and predecessor. The predecessor must replicate its primary data to the new node, and the successor must relocate (a subset of) its data to the new node.

If nodes fail, or new ones join, during an active data relocation protocol, the state of the current prototype implementation is not defined. To support a high rate of churn and unstable nodes, a new approach must be taken. Without going into detail, a possible solution would be to use the second node join definition and add an extra requirement: New nodes are not allowed to join the Chord network before they have all the data they need. All data relocation protocols must support transaction properties. If an abort is necessary, the node awaiting to join can delete all local data and start over. The nodes that are already included in the Chord ring must remain in a consistent state until the data maintenance protocol is committed. New nodes is forced to operate outside the Chord ring until they are ready to enter it. This approach may cause a higher delay before new nodes can join the system, but supports higher churn rates better. At extreme churn rates the approach will fail altogether, as the data maintenance protocols are never allowed to finish (commit).

There is reason to believe that reactive algorithms may be too resource consuming to work in systems with high churn. An alternative is to use *periodic* maintenance algorithms that are not triggered by overlay organizational events. A consequence may be that the system state is rendered temporary inconsistent. It is important to design the algorithms in such a way that a consistent state can be regained. Another variant is to let the Chord layer mask specific events for the application layer until they are declared to be stable, but this might be very hard to manage.

From the discussion above, we can conclude that the cost of node join is mainly expressed by increased complexity of the software. The choice of approach must be made on the need for availability and the expected churn rate for the application.

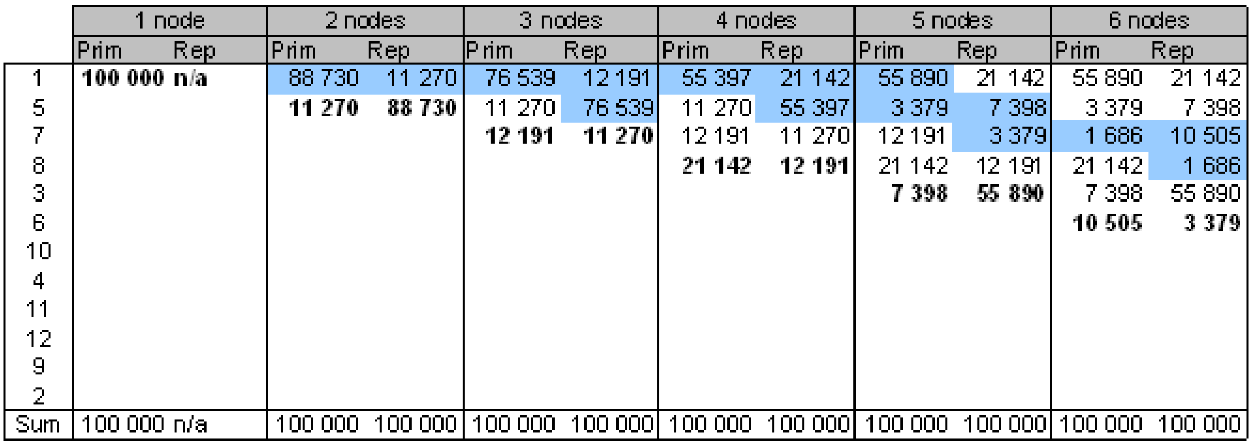
7.4 Verification of data location property

As explained in earlier chapters of the report, it is very important that data is located at the proper node for the system to function correctly. If the data is misplaced, the application will act as if the data does not exist (although the Chord routing protocol routes the message to the proper node). Correct operation of the data maintenance protocols is therefore a critical requirement for the application as a whole. The simple test routine described below was used to verify that no database records were lost during node joins and failures. Note that this test only demonstrate correct behavior for a sequence of "well behaved" node joins and failures. The test is only valid as long as the Chord routing protocol itself is valid.

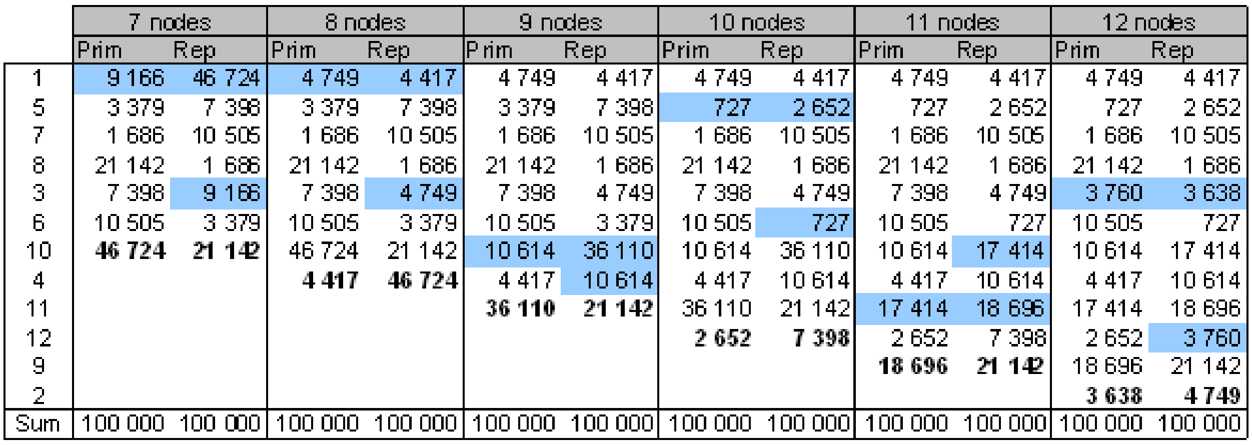
1. A single server is started.
2. 100000 records are inserted (*put* operations), then read back with `devutils.OperationTimer`.
3. 11 new servers are added one by one, where the system is allowed to stabilize for each node addition.
4. The number of records in both primary and replica databases on each node is found after each node addition. These should add up to 100000 records in both primary and replica databases.
5. All 100000 records are attempted read with the tool `devutils.GetNumber-SequenceRecords`.
6. 11 nodes are taken down one by one, where the system is allowed to stabilize after each failure. Nodes are taken down with the `kill` command of the Linux operating system.
7. The number of records in both primary and replica databases on each node is found after each node failure. These should add up to 100000 records in both primary and replica databases.
8. The number of records in the primary database is found. There is no replica database at this point, as there is only one node in the system.
9. All 100000 records are attempted read with the tool `devutils.GetNumber-SequenceRecords`. The record value is also validated by comparing the record value to the record key. They are expected to be equal for this specific case (inserted with `devutils.PutNumberSequenceRecords`).

The record count of step 4 is given in Table 7.2. The other verification properties were valid as well, and we can assert that the data maintenance protocols are correct for "well behaved" sequences of node joins and failures. Note that this test does not suggest anything at all about general sequences of joins and failures (concurrent events, failure during data maintenance protocol execution, etc.). The sum of all records in the system was reported as *5 000 050 000*, which is correct.

As shown in Table 7.2, the sum of records in the system add up to the total number of records originally inserted. This applies to both the primary and replica databases. We also see how data is moved around as new nodes are inserted.



	1 node		2 nodes		3 nodes		4 nodes		5 nodes		6 nodes	
	Prim	Rep	Prim	Rep	Prim	Rep	Prim	Rep	Prim	Rep	Prim	Rep
1	**100 000**	**n/a**	88 730	11 270	76 539	12 191	55 397	21 142	55 890	21 142	55 890	21 142
5			**11 270**	**88 730**	11 270	76 539	11 270	55 397	3 379	7 398	3 379	7 398
7					**12 191**	**11 270**	12 191	11 270	12 191	3 379	1 686	10 505
8							**21 142**	**12 191**	21 142	12 191	21 142	1 686
3									**7 398**	**55 890**	7 398	55 890
6											**10 505**	**3 379**
10												
4												
11												
12												
9												
2												
Sum	100 000	n/a	100 000	100 000	100 000	100 000	100 000	100 000	100 000	100 000	100 000	100 000



	7 nodes		8 nodes		9 nodes		10 nodes		11 nodes		12 nodes	
	Prim	Rep	Prim	Rep	Prim	Rep	Prim	Rep	Prim	Rep	Prim	Rep
1	9 166	46 724	4 749	4 417	4 749	4 417	4 749	4 417	4 749	4 417	4 749	4 417
5	3 379	7 398	3 379	7 398	3 379	7 398	727	2 652	727	2 652	727	2 652
7	1 686	10 505	1 686	10 505	1 686	10 505	1 686	10 505	1 686	10 505	1 686	10 505
8	21 142	1 686	21 142	1 686	21 142	1 686	21 142	1 686	21 142	1 686	21 142	1 686
3	7 398	9 166	7 398	4 749	7 398	4 749	7 398	4 749	7 398	4 749	3 760	3 638
6	10 505	3 379	10 505	3 379	10 505	3 379	10 505	727	10 505	727	10 505	727
10	**46 724**	**21 142**	46 724	21 142	10 614	36 110	10 614	36 110	10 614	17 414	10 614	17 414
4			**4 417**	**46 724**	4 417	10 614	4 417	10 614	4 417	10 614	4 417	10 614
11					**36 110**	**21 142**	36 110	21 142	17 414	18 696	17 414	18 696
12							**2 652**	**7 398**	2 652	7 398	2 652	3 760
9									**18 696**	**21 142**	18 696	21 142
2											**3 638**	**4 749**
Sum	100 000	100 000	100 000	100 000	100 000	100 000	100 000	100 000	100 000	100 000	100 000	100 000

Table 7.2: Record counts for data location property verification. Changed cells are marked blue, the new node is marked with bold font.

7.5 Request throughput

System throughput was not measured due to time constraints. It it not clear how to measure the throughput of the system. The rate is highly dependent on a number of factors:

- Application implementation (threading/concurrency).
- Number of nodes in the system.
- Network connectivity between nodes.
- Underlying database (Berkeley DB).
- Method of measurement.

We hope that the throughput of the system as a whole will increase with the number of nodes in the system, and it probably will. It is unclear what the relation between the number of system nodes and throughput is (i.e. how close to linear). The load on the different proxy nodes is also an important factor.

We performed a simple benchmark of a single node. The node was able to execute between 1000 - 1100 *put*-requests per second on our test machine (see Chapter 7.1, dual processors are not utilized in current implementation). 7 clients were used to saturate the server node. The number obtained is the definite upper bound of throughput per node. In a system consisting of several nodes, the average throughput of each node will be lower due to network latency, routing and traffic from other nodes.

7.6 Costs of operation replication

Replicating operations and their data in our system is necessary to support high availability and to avoid loss of data in case of node failures. Because the prototype uses a replication degree of one and strict replication, the cost of replication is expressed by the cost of transporting the operation to replicate to the succeeding node and the time it takes the succeeding node to execute the operation and reply. Note that there is no Chord lookup involved for a system using a replication degree of one and the succeeding node as the replica node. In addition, replication comes with the cost of using extra storage. In our prototype, there are two copies of each record; the primary and the replica. The actual cost of replication compared to the time it takes to execute on the primary node can be greatly affected by two factors:

- The work load on the succeeding node.
- The network latency and to some degree network bandwidth (depends on size of operation data).

As shown in Chapter 7.2, the time it takes from an operation is being sent to an answer is received is in the range 7 to 27 milliseconds for our test system. If the network latency between the primary and replica node is on the order of hundreds of milliseconds, or even seconds, the cost of replication is very high. An application deployed on the Internet would have to relax the response time bounds. Our test system is not a good system to use for illustration in this case, as the network latency between the nodes is in the range 0.1 - 0.3 milliseconds. This is very low compared to the typical Internet host. The fact that peer-to-peer systems often include hosts with different performance, suggests that fast nodes can be slowed down by the slower ones.

Test results for two equal test runs are show in Table 7.3 below, where replication was disabled in the first run and enabled in the second. 100000 *put*-operations were executed for each run. The size of the successor list was set to two.

Replication	**Average response time**	Comparison
Disabled	10.7 ms	100%
Enabled	21.2 ms	198%

Table 7.3: Costs of operation replication in response times.

As can be seen, we experienced nearly a twofold increase in response times when replication was enabled. This seems reasonable, as the request must be carried out at two nodes instead of one. The extra usage of CPU resources related to the replication should be low, as the prototype simply sends the request to the succeeding node, executes the operation locally and then fetches the result from the succeeding node.

7.7 Further testing

As this prototype was built to check if a high level of scalability could be achieved, more tests including a high number of nodes should be carried out. This is costly to do, due to the number of computers required and the cost of administering them all. One could perhaps do some more work on the prototype and try to utilize the Internet to perform some tests, or take advantage of the resources present at a university or other institution with large computer resources. The main problem in such a scenario is how to administer the hosts and how to collect the required measurements.

In addition to scalability tests, the finger table should be tested more thoroughly. It would be of great help if a large network was at disposal, but one might come far with simulations. An effective finger table is a necessity if the system is to perform in large networks. After a period of lesser-scale testing, it may be possible to test the application on a large network like PlanetLab ([PAR02], [BBC$^+$04], [pla]). Besides from correctness of the algorithms, it is also important to see how the application and overlay network evolve over time under churn.

The implementation should also be optimized. A profiler could be used to figure out which portions of the code is accessed the most, and to fine-tune these hot-spots for better performance. There are actually four different software components in the system: the Chord subsystem, the application layer code, Berkeley DB Java Edition and the Java virtual machine itself. The first two must be optimized by improving the source code, while a gain in performance may be obtained by optimal configuration of the last two.

Chapter 8

Conclusion

It has been shown that it is possible to use a distributed hash table (DHT) as a building block for a primitive distributed database system with dynamic participants (server nodes). Good scalability can be achieved with novel data maintenance algorithms in peer-to-peer networks with nodes that are so stable the the algorithms can finish before the node fails. To support scalability in systems with highly unstable nodes, more complex algorithms must be implemented. It still needs to be assessed if a DHT can be used to build scalable distributed database systems with a full-blown set of high-level database functionality, for example a query language like SQL[1]. This is a field of active research. A missing feature in the prototype is to be able to access all records in a specific database without explicit knowledge of the record keys.

It has become clear that an effective finger table is necessary to achieve acceptable response times in large networks. If the upper bound guarantees of Chord hold in practice, the system performs well enough too be used in many different application areas.

Isolated node joins and failures are handled automatically by the system, but the prototype does not handle concurrent node joins or failures at a specific point of the ring. Should be possible to handle with more complex data maintenance protocols.

8.1 Further work

A lot of work needs to be done before the current prototype can be classified as a release candidate for a database system. We would rather give a few pointers of further work of research interest. Implementing functionality already found in "conventional systems" in a peer-to-peer system raise a number of new challenges and problems to

[1]Structured Query Language

be addressed.

- Transactions.
- Design and implement more advanced synchronization protocols to allow for a higher degree of system availability (avoid block/deny state).
- Design and implement higher level database services.
- Find mechanisms to handle higher level operations that require access to all, or a subset of, records in a specific database without knowledge of the record keys. This is required to support features like distributed cursors and query languages. Query processing in peer-to-peer systems is a field of active research.

In addition, there are some important implementation tasks for the current prototype:

- Increase level of communication fault-tolerance (both at Chord and application level).
- Implement more effective message structure.
- Test and improve finger table related code.
- Tune implementation to achieve better performance.
- Improve threading to avoid node deadlocks that may occur in current implementation (due to message timeouts because of busy thread).

Appendix A

Problem description

Highly available database systems built on clusters of computers show problems when scaling beyond a dozen nodes. The task of this thesis is to design and prototype a simple distributed, replicated database using Berkeley DB and a distributed hash table – Chord. The research task of this thesis is to see if it is possible to apply the scalability of distributed hash tables to a simple record store, and make this a scalable, replicated record store. The main challenge is to integrate these two concepts in a clean way, and at the same time make the performance acceptable. If there is time for it, node joins and leaves should be handled by the system dynamically. We would like to see some performance measurements of the prototype.

Appendix B

Berkeley DB Java Edition classes

A list of the basic classes composing the Berkeley DB Java Edition API. The status of each class regarding their "usage-readiness" for this project is given in the second column. As can be seen from the table below, the JE API is far from fully implemented in our prototype!

Class	**Description**
BtreeStats	Rendered *useless*.
CheckpointConfig	*Semi-usable*. Can be used locally at the server nodes to force a checkpoint, but extra work is required if the client is to be able to checkpoint a specific environment. Must be able to identify all server nodes holding data for the specified environment. Note that checkpointing is handled by an environment background thread.
Cursor	*Not implemented*. A major task to implement in our system. Must have auxiliary functionality to identify all servers with data from a specific database.
CursorConfig	Ready for use, but new configuration parameters may emerge when reimplementing cursors.
Database	*Reimplemented*, but not fully. Had to add communication component. Note that the client use the reimplemented class, while the server nodes still use the original Berkeley JE class.
DatabaseConfig	Ready for use, can be extended to incorporate new configuration parameters.
DatabaseEntry	Ready for use.
DatabaseStats	Rendered *useless*.

Environment	*Reimplemented*, but not fully. As with the database class, a communication component were added.
EnvironmentConfig	Ready for use.
EnvironmentMutableConfig	Ready for use.
EnvironmentStats	*Semi-usable.* Can be used locally be the server, but not by clients. It is probably rendered *useless* in the system wide context.
ForeignKeyDeleteAction	Ready for use.
ForeignKeyNullifier	*Semi-usable.* Can still be applied locally on the servers, but it is unclear at the moment if it need to incorporate logic related to the peer-to-peer context (depends on implementation of secondary databases).
JEVersion	Ready for use.
JoinConfig	Depending on implementation of JoinCursor.
JoinCursor	*Not implemented.*
LockMode	Ready for use.
LockStats	*Semi-usable.* Can be used locally on servers, but a system wide abstraction for usage by the clients is probably useless.
OperationStatus	*Semi-usable.* Still used in prototype to mimic JE API, but it has also been replaced with another object capable of conveying more information (timeouts, exceptions on remote hosts).
SecondaryConfig	Ready for use.
SecondaryCursor	*Not implemented.* Must be reimplemented and require auxiliary functionality. Is also dependent on the secondary database implementation.
SecondaryDatabase	*Not implemented.* Must be reimplemented and require auxiliary functionality.
SecondaryKeyCreator	Ready for use (is an interface).
StatsConfig	Ready for use.
Transaction	*Not implemented.* Another major task. A transaction manager system component must be implemented to support transactions. Require a design process.
TransactionConfig	*Semi-usable.* Can be used as-is, but might have to be extended to suit the transaction implementation.
TransactionStats	Rendered *useless.*
VerifyConfig	Ready for use.

Appendix C

Peer-to-peer file systems

A list of peer-to-peer filesystems is given below, as they are one of the more mature and tested applications related to peer-to-peer systems. They might contain ideas and solutions that are well suited for peer-to-peer database systems as well. The list is not exhaustive.

- **OceanStore**
  The goal is to be a global persistent data store that can scale to handle extreme numbers of users.
  `http://oceanstore.cs.berkeley.edu`

- **FarSite**
  Federated, Available, and Reliable Storage for an Incompletely Trusted Environment
  `http://research.microsoft.com/sn/Farsite/`

- **Ivy**
  Ivy is a multi-user read/write peer-to-peer file system, based on logs (stored in DHash).
  `http://pdos.csail.mit.edu/ivy/`

- **CFS**
  The Cooperative File System (CFS) is a new peer-to-peer read-only storage system that provides provable guarantees for the efficiency, robustness, and load-balance of file storage and retrieval.
  `http://pdos.csail.mit.edu/papers/cfs:sosp01/`

- **Keso**
  A scalable, reliable and secure peer-to-peer file system.
  `http://web.it.kth.se/~mea/keso.html`

Appendix D

Configuration properties

Below the available configuration properties for the system are listed. Properties related to the Java virtual machine are not included, although some of them is important for the system (for instance the size of memory allocated to the JVM). The properties are sorted alfabetically.

- `chord.bootstrapnode`
  Required if the node is to join an existing network. Example: `10.0.0.10:20000`.

- `chord.fingers.enable`
  Enable or disable use of the finger table. Is disabled by default.
  `WARNING:` The finger table code is neither stable nor well tested!

- `chord.fingers.update_interval`
  Specify the update interval for the finger table. Default is `30000` milliseconds. **WARNING:** Setting this too low will cause excessive node load and network traffic!

- `chord.msgsend.attempts`
  Specify number of send attempts for messages before receiving node is declared failed.

- `chord.stabilize_inteval_ms`
  Specify how often the Chord stabilize routine is run. Default is every `200` milliseconds.

- `chord.successorlist.size`
  Specify the size of the successor list. Default is `2`.

- `chord.successorlist.retries`
  Specify number of send attempts for messages to successor nodes before they are declared failed/timed out.

- `chord.udp.ip`
  Required. Specify the IP address the server is to use (local IP).
- `chord.udp.port`
  Specify the port used by the server. Default is `20000`.
- `dbdb_p2p.logconfigurationfile`
  Specify the log configuration file.
- `dbdb_p2p.server.env_home`
  Required. Determine where the server stores the database files.
- `dbdb_p2p.server.replicate`
  Enables or disables replication. Replication is enabled by default. The allowed values are `true/false`, `on/off` and `yes/no`.
- `dbdb_p2p.shutdown_file`
  If the file specified by this property is present, the server shuts down. The default is `dbdb_p2p-shutdown`. If a relaitve path is specified, it is relative to the server start directory.

Appendix E

Available loggers

This is a list of the loggers available in the prototype.

```
chord

chord.chord
chord.successorlist
chord.utils

chord.communications.messagesequence
chord.communications.udpcommunications

dbdb_p2p

dbdb_p2p.client
dbdb_p2p.client.chordinteractor
dbdb_p2p.client.database
dbdb_p2p.client.environment

dbdb_p2p.server
dbdb_p2p.server.batchtransferutils
dbdb_p2p.server.chordcommunicator
dbdb_p2p.server.databasemanager
dbdb_p2p.server.databasereplicator
dbdb_p2p.server.datarelocator
dbdb_p2p.server.dbpool
dbdb_p2p.server.handlemanager
dbdb_p2p.server.operations
dbdb_p2p.server.request
dbdb_p2p.server.server
```

`dbdb_p2p.server.serverrequestprocesser`

`dbdb_p2p.threading`

Appendix F

UDPChordInteractionNode - commands

The output from the `help` command in `chord.devutils.UDPChordInteractionNode` is listed below, to give an impression of what can be done with this tool.

```
! msg_send    - Sends the specified message to the specified node.
! id10        - Displays the node's id in base 10.
! pred        - Displays this node's predecessor.
! lookup      - Looks up the specified key.
! help        - Displays this help text :)
! msg_view    - Executes the 'toString' method on the last message.
! hex_lookup  - Looks up the specified id.
! msg_timeout - Displays or sets current timeout value for messages.
! quit        - Exits the program.
! succ        - Displays this node's successor.
! ?           - Displays this help text :)
! fingers     - Displays this node's fingertable.
!               NB! May be a lot of output!
! log         - Set the log level: 'log LEVEL', LEVEL =
!               {ALL, SEVERE, WARNING, INFO, CONFIG, FINE, FINER, FINEST}
! msg_deliver - Delivers the last message received after executing the
!               'msg_send' command to the Chord node.
! adr         - Displays this node's address.
! id          - Displays the node's id in hexadecimal.
! exit        - Exits the program.
! succlist    - Displays this node's successor list.
```

Bibliography

[apa] Apache ant: a java-based build tool. http://ant.apache.org.

[bam] The bamboo distributed hash table a robust, open-source dht. http://bamboo-dht.org/.

[BBC$^+$04] Andy Bavier, Mic Bowman, Brent Chun, David Culler, Scott Karlin, Steve Muir, Larry Peterson, Timothy Roscoe, Tammo Spalink, and Mike Wawrzoniak. Operating system support for planetary-scale services. In *Proceedings of the First Symposium on Network Systems Design and Implementation (NSDI)*, March 2004.

[bes] BESTPEER: Adaptive peer-to-peer platform for object sharing. http://xena1.ddns.comp.nus.edu.sg/p2p/.

[BKK$^+$03] Hari Balakrishnan, M. Frans Kaashoek, David Karger, Robert Morris, and Ion Stocia. Looking up data in P2P systems. *Comunications of the ACM*, 46(2), February 2003.

[BKLN02] Hari Balakrishnan, David Karger, and David Liben-Nowell. Analysis of the evolution of peer-to-peer systems. In *Proceedings of the Twenty-First ACM Symposium on Principles of Distributed Computing*, 2002.

[cho] The chord project. http://pdos.csail.mit.edu/chord/.

[DGA04] Anwitaman Datta, Sarunas Girdzijauskas, and Karl Aberer. On de bruijn routing in distributed hash tables: There and back again. 2004.

[DZDS03] F. Dabek, B. Zhao, P. Druschel, and I. Stoica. Towards a common API for structured peer-to-peer overlays, 2003. http://citeseer.ist.psu.edu/562746.html.

[fip02] Secure hash standard. Technical report, Federal Information Processing Standards Publications, August 2002. http://csrc.nist.gov/publications/fips/fips180-2/fips180-2.pdf.

[HCH$^+$05] Ryan Huebsch, Brent Chun, Joseph Hellerstein, Boon Thau Loo, Petros Maniatis, Timothy Roscoe, Scott Shenker, Ion Stoica, and Aydan R.

Yumerefendi. The architecture of PIER: an internet-scale query processor. In *Proceedings of the 2005 CIDR Conference*, January 2005.

[hyp] The hyperion project. `http://www.cs.toronto.edu/db/hyperion/`.

[jun] Junit: a reggression testing framework. `http://www.junit.org`.

[KLL$^+$97] David Karger, Eric Lehman, Tom Leighton, Mathhew Levine, Daniel Lewin, and Rina Panigrahy. Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the world wide web. In *ACM Symposium on Theory of Computing*, pages 654–663, May 1997. `http://citeseer.ist.psu.edu/karger97consistent.html`.

[MNR02] Dahlia Malkhi, Moni Naor, and David Ratajczak. Viceroy: A scalable and dynamic emulation of the butterfly. In *Proceedings of the 21st ACM Symposium on Principles of Distributed Computing*, 2002.

[ora] Oracle corporation - database systems. `http://www.orcale.com/database/`.

[PAR02] Larry Peterson, Tom Anderson, and David Culler Timothy Roscoe. A blueprint for introducing disruptive technology into the internet. In *Proceedings of the First ACM Workshop on Hot Topics in Networking (HotNets)*, October 2002.

[pep] PEPPER: Peer-to-peer data management. `http://www.cs.cornell.edu/database/pepper/`.

[pgr] P-GRID: The grid of peers. `http://www.pgrid.org/`.

[pia] The piazza peer data management project. `http://data.cs.washington.edu/p2p/piazza/`.

[pie] PIER project homepage. `http://pier.cs.berkeley.edu/`.

[pla] Planetlab - an open platform for developing, deploying, and accessing planetary-scale services. `http://www.planet-lab.org/php/overview.php`.

[RD01] Anthony Rowstron and Peter Druschel. Pastry: Scalable, decentralized object location and routing for large-scale peer-to-peer systems. In *IFIP/ACM International Conference on Distributed Systems Platforms, Middleware 2001*, November 2001.

[RGRK04] Sean Rhea, Dennis Geels, Timothy Roscoe, and John Kubiatowicz. Handling churn in a DHT. In *Proceedings of the USENIX Annual Technical Conference*, June 2004.

[Riv92] R. Rivest. Rfc 1321 - the md5 message-digest algorithm. Technical report, MIT Laboratory for Computer Science and RSA Data Security, Inc., April 1992. `http://www.faqs.org/rfcs/rfc1321.html`.

[Sch05] Bruce Schneider. Schneier on security: Cryptanalysis of sha-1, February 2005. http://www.schneier.com/blog/archives/2005/02/cryptanalysis_o.html.

[Sle04] Sleepycat Software. *Berkeley DB Tutorial and Reference Guide, Version 4.2.52*, 2004. http://www.sleepycat.com/docs/ref/toc.html.

[SMLN^{+}02] Ion Stoica, Robert Morris, David Liben-Nowell, David Karger, M. Frans Kaashoek, Frank Dabek, and Hari Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. Technical report, Massachusetts Institute of Technology (MIT), Laboratory for Computer Science, January 2002. http://www.pdos.lcs.mit.edu/chord/papers/chord-tn.ps.

[sta] Stanford peers. http://www-db.stanford.edu/peers/.

[sub] Subversion: a compelling replacement for cvs. http://subversion.tigris.org.

[Uls03] Jostein Ulseth. Distributed and Replicated Berkeley DB. Technical report, The Norwegian University of Technology and Science, 2003. Depth Study.

[Uls04] Jostein Ulseth. Distributed and Replicated Berkeley DB. Master's thesis, The Norwegian University of Technology and Science, 2004.

[Ver04] Dinesh C. Verma. *LEGITIMATE APPLICATIONS OF PEER-TO-PEER NETWORKS*. John Wiley & Sons, Inc, 2004.

[Waa04] Kristian Waagan. Applying Distributed Hash Tables to Distributed Berkeley DB. Technical report, The Norwegian University of Technology and Science, 2004. Depth Study.

[web] Webopedia. http://www.webopedia.com/TERM/D/distributed_database.html.