Diplomarbeit

Entwicklung eines generischen Frameworks zur vereinfachten Umsetzung verteilter, daten-orientierter Netzwerkfunktionen

Ole Rixmann
Matrikelnummer: 3266061
11.9.2013

Betreut durch
Prof. Dr. rer. nat. habil. Dr. h. c. Alexander Schill,
Dr.-Ing. Thomas Springer &
Dipl. Inf. Tino Breddin
# Table of Contents

1 Introduction ................................. 1

2 Related Work ................................. 3
   2.1 Basic Concepts .................................. 3
      2.1.1 CAP-Theorem ................................ 3
      2.1.2 Leader Election ................................ 4
      2.1.3 Two-Phase-Commit-Protocol ..................... 5
      2.1.4 Paxos ........................................ 5
      2.1.5 Chain Replication ............................ 7
      2.1.6 Skiplists ..................................... 9
      2.1.7 Consistent Hashing ......................... 10
      2.1.8 Chord ....................................... 11
   2.2 Complex Systems .............................. 12
      2.2.1 Amazon Dynamo .............................. 12
      2.2.2 Hyperdex .................................. 14
      2.2.3 Chubby ..................................... 14
      2.2.4 BigTable .................................. 15
      2.2.5 Megastore .................................. 16
      2.2.6 Spanner .................................... 17
   2.3 Summary ..................................... 18

3 Analysis ...................................... 21
   3.1 Environment ................................. 21
   3.2 Requirements .................................. 24

4 System design ................................ 27
   4.1 Finding an approach to consistency .............. 27
   4.2 Basic design .................................. 30
   4.3 Paxos statemachine ............................. 32
      4.3.1 Modes of operation .......................... 34
   4.4 Paxos log ..................................... 35
      4.4.1 Log cutting ................................ 35
      4.4.2 Dealing with IO ............................ 37
      4.4.3 Membership change ......................... 38
      4.4.4 Summary .................................. 39
Selbstständigkeitserklärung

Hiermit erkläre ich, dass ich die von mir am heutigen Tag dem Prüfungsausschuss der Fakultät Informatik eingereichte Diplomarbeit zum Thema:

Entwicklung eines generischen Frameworks zur vereinfachten Umsetzung verteilter, daten-orientierter Netzwerkfunktionen

vollkommen selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt sowie Zitate kenntlich gemacht habe.

Dresden, den 11.9.2013

Ole Rixmann
1 Introduction

Distributed data storage and access are a problem for any IT-Service. Especially websites that grant many users access to the same data, like social networks or collaborative working platforms run into the problem that one cannot achieve arbitrary properties in a distributed system. According to the CAP-Theorem, which is described in chapter 2.1.1, the three aimed properties are (Consistency, Availability and Partition tolerance) and they cannot all be achieved by the same system at the same time.

Since many companies ran into these difficulties, they developed their own solutions. Google, for example, has many SQL-like and NO-SQL databases with different characteristics, suitable for different usage scenarios.

This thesis is done at Travelping, the technical context of this work is the infrastructure that Travelping produces and the software-ecosystem the company uses. Travelping provides Authentication, Authorization and Accounting (AAA) services for network-providers. Parallelism and redundancy lie in the nature of this AAA domain. The used software is already designed with respect to these two aspects. Most of the systems were written in Erlang, which has a trackrecord in- and was designed for- the development of telecommunication systems.

The components of the (Travelping) system have different requirements for data storage and distribution. The aim of this thesis is to develop a framework for distributed data storage and computation that is flexible enough to be integrated into some of Travelping’s components. Furthermore, it should allow to be adapted to the needs of the client-application in a way that improves scalability and failure tolerance of the specific component.

The system should be easily usable from Erlang in an idiomatic way, extending Erlang’s capabilities to build distributed systems. If the core of the system can be implemented independently of software that Travelping has licensed under non-free licenses, it may be used by the open source community.

- To achieve this, first there has to be an analysis of the needs that the respective components have.
- Further the current state of the art has to be researched with respect to these needs.
- The last two points give a good overview about what can be adapted or developed to fulfill the needs at Travelping, so this leads to developing or adapting a technology.
- Implementing this technology so it can be tested and used.
• Testing the performance and functionality of the developed system.

There are systems, for example databases, that are very performant in environments which have shared memory, limiting these systems to scale inside a single computer. The technology developed in this work should abstract different storage-backends on the layer of network communication. In the terminology of databases this role is performed by a transaction manager that coordinates requests to storage backends running on multiple computers.

The framework developed here should offer a generic interface through which a broad range of client applications can make use of it. Other systems, like databases, are limited in this aspect.

Depending on the underlying storage backend, the framework may be used for transient or persistent data which is required if the system should be used in discless environments. Further, it allows high-efficiency/high fault tolerance tradeoffs.

This document is structured into the following chapters. First, the research chapter analyzes the current state of the art in form of basic concepts and more complex systems that are composed of basic concepts. The second part covers the analysis of the requirements that the system has to fulfill. Next, design and development of the software are described in detail, which may serve as documentation for future users of the developed system. Finally, the softwares performance is tested as a part of a Travelping product and the thesis finishes with a conclusion.
2 Related Work

This chapter contains an analysis of the state of the art in distributed software architecture. The first part is about basic concepts of distributed system architecture, while the second part describes complex real-world systems that use concepts from the first part to perform domain-specific tasks.

2.1 Basic Concepts

This chapter starts with the CAP-Theorem that introduces problems and tradeoffs that every distributed system must face. A second concept, Leader Election, is explained while the rest of the chapters (Two-Phase-Commit-Protocols, Paxos, Consistent Hashing etc.) discuss technologies that are used to implement systems that achieve attributes discussed in the CAP-Theorem.

2.1.1 CAP-Theorem

The CAP-Theorem or Brewer’s Conjecture as presented in [Brewer2000] analyzes the characteristics one can hope to achieve in a distributed system under the assumption of an unstable network and failing computer hardware.

It takes a data-centric view because the state of any application is its data and most requests/tasks require access to the up-to-date application state. Distribution of computation (mutation on the application’s state) can be done easily [Brewer2000]. Every operation in a computer system will consist of both computation and data access, computation without a data basis is useless as well as data that cannot be interpreted (through computation). If an operation is highly data dependent, data will have to be accessed which may cause problems in a distributed system. Operations that focus on computation can be executed on a computer (if an efficient algorithm exists) after accessing the needed data.

The CAP-Theorem states that three goals can be achieved by a distributed system:

- Consistency: if data is read at the same point in time by two processes, they always get the same result.
- Availability: any data can be read or written at any point in time by any process that has a connection to at least one machine of the system.
• Partition Tolerance: if (due to link failure) a net-split occurs, the disconnected parts still work properly on their own.

In [Gilbert2002] it is proven that any system can achieve two of these goals at most under the assumption that nodes have a local timer but no absolute time is available.

In general, partition tolerance is impossible to achieve if less than a majority of the participating machines in a system is left in a partition (if these non-majority-partitions keep on working). The reason for this is that the same object can be changed at the same point in time in different working partitions, which can lead to inconsistencies [Gilbert2002]. If the application’s servers have distinct tasks and are able to perform them without interaction with the rest of the system, total partition tolerance is possible (but the system is then similar to several independent systems).

Specialized systems like a database tend to have either ACID [Haerder1983] (Atomicity, Consistency, Isolation, Durability)-like behaviour that uses strong consistency but is limited in terms of availability, or BASE (Basically Available, Soft-State and Eventual Consistency)-like attributes that guarantee availability but not consistency.

Real world applications use a mixture of both technologies to achieve the availability and consistency capabilities that the semantic of the specific application requires. An example for a system that requires ACID semantics is an accounting system that must make sure a transaction happens as expected. For a system that can be implemented using a BASE providing technology a search engine is an example. Here, availability is more important than consistency, if someone else gets a search result for the same request that is different, no problem occurs.

2.1.2 Leader Election

Leader election describes the problem to have a single node that has an outstanding position amongst a group of peers that is known to any node in the group at any point in time. This is described in [Stoller2000] for asynchronous distributed systems, an algorithm that fully solves this problem is not yet known. The problem is important because most distributed systems require a point of central coordination, i.e. a leader.

Depending on the structure of failures that need to be tolerated, leader election can be impossible. If, for example, a node is able to communicate with some nodes of a group but not its actual leader, leader election may be only possible after separating the group or by introducing an overlay-network (which can be used for communication with the master). Thus, a perfect algorithm for leader election would have to produce a “minimum-sized clique cover” first to determine which groups of nodes should have a common leader.

In Paxos [Lamport1998] leader election is explained further as distributed consensus about a leader. For this to work (with the help of Paxos) at least the size of a quorum must be known beforehand by the voters to decide if an election is rightful or not. With Paxos, only a majority-vote can elect a leader. If the group is split into several parts, one
single part can be able to perform the Paxos protocol and elect a leader. Therefore, the problem is not fully solved by Paxos.

### 2.1.3 Two-Phase-Commit-Protocol

Commit Protocols are used for a special kind of distributed consensus, an atomic commitment protocol (ACP). ACPs are used to implement transaction systems which can provide the transactional behaviour of distributed databases [wiki:2PC].

Every participating node has a local state and two logs, an undo- and a redo-log. A master serializes requests and sends them in a first phase to the participating nodes.

In this commit-request-phase every node performs the steps that are locally necessary for the request while keeping its logs. Afterwards, an acknowledgement is sent to the master (the vote).

If all acknowledgements have arrived, the master initializes the completion-phase by sending a commit-message to every participating node. In case the request fails on any node locally, the completion-phase is completed with a rollback-message that makes all state changes from the commit-request-phase undone. Correct handling of the logs assumed, transactions can be repeated until they eventually succeed in such a system.

By prepending a third phase to collect the current knowledge of the system (relating to the request) before issuing a request, probability of the request’s success can be increased. It is more unlikely that a participating node has different knowledge when handling the request compared to the point where the information was collected. If the requests have a strictly ordered ID (which needs to be handled as in Paxos 2.1.4), the common master is not required but may be used for performance issues. This will be explained in the following chapter about Paxos.

In terms of the CAP-Theorem, if the database is not replicated among the participating nodes, Two-Phase-Commit-Protocols increase only availability (for write-requests) and are aware of Partitions but not able to act on a network split; another protocol is needed for leader election to replace a failing master.

### 2.1.4 Paxos

Paxos is a protocol for distributed consensus, it was invented by Leslie Lamport in 1989 [Lamport1998]. Paxos is the foundation of complex systems that are mentioned later in this document.

Distributed consensus means that, among a group of participating computers, a value can be found for a subject. In the future (after it has once been decided) either no value will be found for the subject or the value decided.

Paxos is a three-phase-commit protocol, that is often split into three roles: Proposers, Acceptors and Learners that correspond to the three message rounds required in the protocol. In figure 2.1 a sequence diagram of the basic Paxos protocol is shown.
In the first round (prepare), values that may have passed through past first rounds are collected. The process which started the first round will, after a quorum of the participating processes has answered, choose the most recent value and start the second round with it (propose). When having received positive responses to the second round from a quorum of the participating processes, the proposing process knows, that every first round started in the future will find the value it proposed and choose it, because it must be the most recent. The chosen value must be the most recent if a quorum of the participating nodes still knows to what value it has answered in the propose round.

It is possible to implement a distributed state machine with the help of Paxos, using which, the core of a distributed system can be implemented.

A Paxos based system is fully operational as long as a majority of the participating Processes is working properly. If the system reconfigures itself on failure to the count of available Processes, it will survive any amount of failing Processes as long as no more than a minority of Processes fails at a single point in time. The state is fully replicated among the participating nodes. Thus, Paxos is a protocol to achieve consistency in a distributed system which is partition tolerant (it will not work if no quorum can be found). This is proven in Lamport1998.

What is not proven is that progress will actually happen, it is only shown that if progress happens it is consistent. For increasing availability, optimizations can be introduced into the basic protocol.

“Multi-Paxos” is a variant that is well suited to be used in Key-Value-Stores, in figure 2.2 and 2.3 the message flow is shown, in 2.3 the messages that are only exchanged between the components of an agent are omitted. For multi Paxos, a single proposer is elected
Figure 2.2: *Multi Paxos* (single proposer)

(with a round of regular Paxos), it will then be the only agent to issue new requests and the other agents will not accept requests from anyone but the single proposer. The promise for the next subject is given with the vote from the former subject, this saves one round of messages, and as the single proposer must have complete information, he can issue only valid proposals. If the underlying data-structure, which the statemachine exposes (for example a Map in a Key-Value-Store) allows for commuting operations, the master may even issue several commuting operations in parallel.

For multi Paxos a leader/single proposer is required. It can be elected using a full round of Paxos. Leader election can be done in Paxos but will not be perfect because in a split group only a quorum containing partition will be able to elect a master.

Regarding to the CAP-Theorem, Paxos fulfills consistency and partition tolerance perfectly (or as good as this is possible for partition tolerance) but lacks in availability. That is because the protocol gets slower as more machines participate (communication overhead) and the limited speedup (no horizontal scalability). But as read-requests can be served from every participating node and/or can be cached on other machines, availability for reading can be much better than for writing (this optimization can lead to read-inconsistencies). Thus, Paxos is well suited for infrequently changing data that may be read inconsistent if high availability is needed.

As a distributed master or a configuration database (both require consistency), Paxos is the heart of many other technologies.

### 2.1.5 Chain Replication

Chain replication, as described in the original chain replication paper [Renesse2004], is an instance of the primary/backup approach described in [Alsberg1976]. Primary/backup means that there is a single node accepting requests which writes backups to other nodes that can take the role of the primary if the primary should fail. Chain replication is a
Figure 2.3: *Multi Paxos*, single proposer is issuing requests, proposer, acceptor and learner collapsed into a single agent

A technique to achieve availability and consistency of objects in a distributed system, these goals can be compared to those of Paxos.

Figure 2.4: Illustration of a chain from the original *chain-replication-paper* [Renesse2004]

Every object has an associated chain that consists of (at least two) nodes that are arranged in the shape of a chain. Availability can be improved by having a chain for each object (the object may be a hashrange) that handles the object independent of the other chains. A coordinator that knows the topology of all chains is required to find the node to send a request to. The coordinator itself can be distributed through the usage of, for example, consistent hashing but this requires knowledge about the way that requests are mapped to come nearer to the client.
Participating nodes follow the chain replication protocol, which means that updates are accepted by the first node in a chain and then propagated through the chain to its tail. An update is finished if it reaches the tail, which is also the place where queries are issued. A single chain is illustrated in figure 2.4. Every node in a chain has a full copy of the chain’s object, so the chain will work with all but one node failing.

The task to serialize requests is distributed between the head (update) and the tail (queries) of a chain. This states a contrast to quorum-techniques like Paxos 2.1.4 where serialization is done in a distributed manner by a quorum of the participating nodes.

If strong consistency is needed, queries are only handled by the tail but can otherwise be dealt with by any node in the chain (which increases availability). As every predecessor to the tail knows any updates earlier than the tail, not yet fully processed updates may be read on non-tail-nodes.

Chain replication by itself is not partition tolerant, if a chain of N elements is split into N partitions, each node will think it is the only survivor and keep accepting requests which will lead to inconsistency. Leader election is not possible under these circumstances.

In terms of the CAP-Theorem, chain replication can achieve Consistency, Availability is at most as good as a single machine.

### 2.1.6 Skiplists

In general, trees are datastructures that have good performance characteristics (O(\(\log n\))) as long as they do not need to be rebalanced. Rebalancing a tree is very costly, even if done in small steps with every write and delete operation. Skiplists offer a probabilistic approach to trees, as they can be seen as a tree-like structure without a fixed branching factor, but with a fixed depth and some redundancy (redundant links).

![Figure 2.5: Illustration of a skiplist, from "Pugh1990"](image)

Figure 2.5: Illustration of a skiplist, from Pugh1990

Illustration 2.5 shows a skiplist that has a maximum-depth of four, containig ten elements without the head (where operations start) and the tail (where operations are terminated). Every element has pointers to the next element down the skiplist that has at least the height of the pointer’s index and that is not in the “shadow” of an object inbetween. A skiplist is balanced through the insertion of new elements where the height (amount of fingers) is determined by a random distribution that makes sure that the possibility to find a higher node (when moving at a “index-level”) is big. Inserting new elements in this manner leads by probability to a skiplist that achieves O(\(\log n\)) performance characteristics because the jumps forward on level I in the skiplist move across roughly \(2^I\) elements Pugh1990.
Stepping around explicit rebalancing is the huge benefit that this technique offers compared to regular trees but the insertion-cost is also higher than that of an unbalanced tree (where inserting is always only adding the new leaf to one old leave). Skiplist algorithms are also more easy to implement (especially in the distributed case) than balanced-tree-algorithms \cite{Pugh1990} and, for this reason, allow for easy adaptation and performance optimizations.

In Terms of the CAP-Theorem, skiplists offer good availability but no robustness to losing data. Even for parallel write operations the performance is very good, as can be seen in \cite{Pugh1990}. This is achieved through micro-locking and lazy deletion strategies.

As no replication is happening, consistency must be perfect (the same node is always used for access to a distinct object) but if netsplits occur, the unavailable nodes make the data which they keep unavailable for the rest of the system as well.

\section{2.1.7 Consistent Hashing}

In an application where some resource undergoes distributed access, it is often necessary to increase the availability of this resource to more than a single server handling the access. A solution to this problem is described in \cite{Karger1997} for a distributed (webserver/http) cache.

The idea used in this paper, and in Hash Rings in general, is based on the possibility (in these applications) to identify the server that is used to answer a request based on the data that the request and its attributes provide. This information from the request is mapped to an element of a finite and monotone keyspace (in practice this does a hash-function) which is then used to identify the server responsible for this request. How this is done depends on the architecture of the specific system and the data it uses. It can be as simple as a table that maps distinct key-ranges to servers and as complicated as the overlay-network described in \ref{2.1.8}. A consistent hash-function as described in \cite{Karger1997} is required to induce only minimal overhead when adding or removing nodes and assertions on the location of nodes in the presence of fault (which requires redundancy).

Most systems use a lighter scheme which is shown in figure \ref{2.6}. Here, the keyspace is split into sixteen equally sized parts that are each handled by a virtual node (vnode). The vnodes are implemented by a set of one to sixteen servers that can each serve any (reasonable) number of vnodes. A mapping of vnode to server is used to find the server responsible for a request.

Locating the server that is responsible for a specific request can either be done by the client itself or by a layer in the system between the client and the resource (which imposes an extra network-hop).

It is also possible to omit hashing the information provided by the request, instead mapping some of the information directly to an element of the keyspace. This can be done to locate similar parts of the resource on the same server, which can increase performance but creates a hot-spot that undermines the aim described here.
If the hash-funktion is equally distributed, only $1/\text{Nth}$ (if the whole keyspace is split among N Servers) of the work falls into the responsibility of each server (by probability). This leads to a perfect linear scaleup if the servers can answer the requests without further coordination among another.

In Terms of the CAP-Theorem, availability can be achieved through hashing and sharding. If the hash-function is consistent and a protocol keeps the replicas consistent, consistency may be achieved (but consistent hashing can only choose the machines that should have replicas of an object).

### 2.1.8 Chord

When using Consistent Hashrings, a bottleneck can emerge at the point where the topology is maintained if there are a lot of real machines participating (and many vnodes, because every machine handles at least one).

Chord [Stoica2001] offers a technique that is able to remove this bottleneck even if the underlying network and hardware are very unreliable (for example in a peer-to-peer-network). It does so by distributing the routing table among the participating machines in a manner that the amount of network-hops required to execute a request is in the magnitude of $O(\log n)$.

Chord does not map keys to values like a key-value-store does, instead it maps keys...
to machines that are responsible for a key’s value, this is comparable to what a naming-
service does. What objects are managed and how they are managed by the nodes is of no
cconcern for the chord protocol (no replication).

The protocol is split into two separate parts. The first part is a very simple protocol that
establishes a double-linked list between the nodes that make up the hashring. Only the
information about a node’s successor and predecessor on the ring are obtained through
the simple protocol, repeated polling keeps the ring up-to-date.

Another protocol that assumes a mostly correct underlying ring uses it to create/update
“fingers” (redundant pointers) to other nodes on the ring. Every node needs \( \log(N) \) finger-
pointers that each point to the node which \( 2^I \) nodes ahead on the ring, where \( I \) is the
finger’s index.

Requests are forwarded by every node to the node (from the local fingertable) which has
the lowest distance from the request’s key. By doing so, every network hop comes closer
to the node that is responsible for the key, which will always be reached if at least the
underlying ring is correct and the node online. Performance degrades when finger-tables
are outdated (up to \( O(N) \) if only the pointer to a node’s immediate successor works),
but if they are correct, requests take only \( O(\log N) \) network-hops to find the responsible
machine, in \[^{Stoica2001}\] this was shown by a test-setup.

In terms of the CAP-Theorem, chord has the same attributes that Hashrings have
(because chord only distributes the routing information that is required to find the right
machine in a hashring). Chord introduces extra complexity in the routing of requests
to achieve better robustness (compared to hashrings where the routing-information is a
single point of failure).

### 2.2 Complex Systems

Here, distributed systems are discussed that perform their tasks in the real world. They
are composed from technologies which have been discussed in the former chapter. Most
systems were created at Google and are examined here because they successfully handle
a huge load and Google publishes their experiences with these systems.

**2.2.1 Amazon Dynamo**

Dynamo, as described in \[^{DeCandia2007}\], is a Key-Value-Store that was developed and
is used at Amazon. Many applications at Amazon require only primary key access to the
request-specific objects. It was found that the time to answer a request in an e-commerce-
platform directly influences the revenue generated by the platform \[^{DeCandia2007}\]. For
this reason, Amazon has \[^{Service Level Agreements (SLAs)}\] for the different components
that make up a service that is exposed to the internet. To fulfill the \[^{SLAs}\] Amazon
Dynamo is optimized for availability.
Dynamo’s architecture is that of a consistent-hash-ring (no k-consistent-hash-function but the ring consisting of vnodes), many vnodes may be associated to a single physical machine. Each object is replicated on the N nodes that follow the node responsible for the object’s key on the hashring.

A quorum- like technology (weak-quorum) that has configurable consistency attributes, is used to store data on or read data from the responsible nodes. The consistency-parameters of Dynamo are depending on the client application. A tupel (N,R,W) defines at how many distinct locations (vnodes) an object is stored (N), how many of these locations must be read successfully to answer a read-request (R) and how many of these locations an object must be written to for a successfull write-operation (W). As no roll-back as in two-phase-commit (2.1.3) is possible when errors are detected, the application logic must handle occuring inconsistencies (weak consistency).

To meet the SLA’s requirements, Dynamo trades consistency for availability. Write- and read-requests will nearly never fail and always return after a very short period of time (depending on the configured parameters). On the other hand, read requests may return several versions of the requested object, which will then have to be reconciliated by the client-application logic.

Depending on the client application’s semantics this may be easy (e-shop cart where inconsistencies may be resolved by the customer) or impossible (bankaccount’s balance that must not become negative). Dynamo can always reconcile itself (force last update or similar strategies), but is is likely that this will break the client application’s semantic. To make client-side reconciliation possible, Dynamo provides (regular) time stamps and vector clocks which make it possible to determine which operations went on in-serial and which in-parallel. The client application must define the consistency-parameters (N,R,W) and make sure that it can handle the inconsistencies emerging from Dynamo operating on these parameters.

The information required to find the corresponding node for a key is exchanged through a whisper protocol through which the nodes exchange topology information. Every second each node sends a request to a random node, updating routing information on both sides. Unlike Chord (see chapter 2.1.8) or other peer-to-peer networks, the routing information to map a key to its responsible machine is available at any node.

Routing can either be done by any machine that is participating in a Dynamo instance or a client can ask any node for its routing information. To perform a request, besides the routing information a statemachine must be run on the client, for this reason a library in the client application is used. When issuing requests directly from the client, the time required for a request can be reduced by a large margin.

Riak Core

Riak Core is a framework which was extracted by the company Basho from the Key-Value-Store Riak which is build from the same design as Amazon Dynamo. It is no Key-
Value-Store by itself, instead it is a framework which allows an arbitrary application logic to be run in a Dynamo like system.

Request from clients are dispatched by the client application itself which forwards requests to the right vnode and node using Dynamo. In the destined vnode a client application callback is used to handle the request, allowing for client application specific behaviour.

At Basho Riak (a Key-Value-Store) and Riak Search (a text searchengine for Riak) are implemented using Riak Core.

### 2.2.2 Hyperdex

Hyperdex as described in [Escriva2012] is a Key-Value-Store with a unique approach to horizontal scalability.

It uses a hyperspace spanned from the primary key and all indices a database scheme uses. Subspaces are distributed among the participating nodes in a way that supports efficient searching if the indexed attributes can be limited in a query.

For achieving failure tolerance, each subspace is replicated to a configurable count of nodes. Through this, the indices are also replicated as they are part of the subspaces.

When updating or initially writing an object, a chain (see chapter [2.1.5](#)), is spanned through all related subspaces and their replicas. This leads to very long chains in worst case scenarios and inherits the problems chain replication has with partition tolerance.

### 2.2.3 Chubby

Chubby is a distributed, coarse-grained lock-server that is implemented as a Paxos statemachine. The incentive at Google, for building Chubby, was to create an easy way to use distributed coordination, but a Paxos statemachine library was found to be too difficult to use for many programmers. So Chubby is a substitute for such a library [Burrows2006].

Using Chubby, resource ownership can be made dependent on ownership of the lock corresponding to the resource. Resources in Chubby are files and they are accessed like a file system, it even fulfills some POSIX-requirements [Burrows2006]. Taking and loosing locks can be observed (through a subscription to the lock) by other clients. These features make Chubby very useful for the configuration and coordination of distributed systems that have a master/slave structure because the masters (but also the slaves) can be identified using Chubby resources.

A Chubby deployment consists of five nodes, which allows (because of Paxos) two nodes to fail while preserving full operation. Read-operations are resolved on the local Chubby node. Write operations can be coordinated in two ways (different Paxos modes), one is to pass the master (regular Paxos, Generalized Paxos), the other is to get a promise from the master for a future round which is then issued by the client (fast-Paxos).

Chubby achieves good performance by:
• Using time-leases for locks that are automatically extended by the master (asking the client if he still needs the lock)

• A system of timeouts that let a client keep a lock while the master is on failover.

• Sending read-requests to the nearest (fastest reachable) Paxos Server which will always be able to provide the newest data.

It is able to serve 10,000 clients with each Chubby node.

An open source implementation of Chubby exists as an apache project, “Apache Zookeeper”. Zookeeper developed ZAB, the Zookeeper Access Broadcast, to achieve better performance than the Paxos used in Chubby [Ladis2008]. ZAB is very similar to Paxos, but it relies on message ordering (using TCP), which allows optimizations, compared to Paxos which knows no ordering of messages.

In terms of the CAP-Theorem, Chubby inherits Paxos attributes, this is why Chubby is coarse-grained (availability for write-operations is limited).

2.2.4 BigTable

Google’s BigTable is a database-layer on top of Google File System (GFS), which provides unlimited (in rowcount and size) tables [Chang2006]. It uses a strong data-model, this means that BigTable needs to know about the data types which are stored under a column-key. BigTable makes use of data-locality and it does not have any query-capabilities besides row-ranges. The user (developer of a client system) is responsible to structure its application data in a way that he can find the things he needs by row-names and grouped. Thus, data-locality can be exploited.

Because BigTable does not support secondary indices, there are no queries possible besides (bruteforce) looking up all objects yourself. However, the later discussed technologies from Google provide transaction- and query-layers on top of BigTable. Rows are stored with a timestamp and the last three versions of a row are available in BigTable (this helps building transactions).

Column-families may be defined by the client to configure BigTables behaviour (for example in-memory vs disk columns). Further configuration may be done on locality groups, which span several column-families.

Row ranges are handled by tablets which are the unit of data organization and location, tablets (like vnodes in other systems) are spread over the participating nodes.

BigTable consists of one master and several tablet-servers (and the external parts like Chubby and GFS on which BigTable depends). The master assigns tablet-servers to tablets (this includes failover, growing/shrinking, adding of column-definitions etc.) and triggers cleanup in GFS. The tablet-servers process read- and write- requests with the help of a transaction log which is persisted to GFS. These logs are held in memory, and persisted to GFS in several stages that involve compaction.
GFS provides consistency and replication, so BigTable must not be concerned with these problems [Ghemawat2003]. There are several techniques used to increase performance, many tablet-servers log into the same GFS-file, bloom-filters are used to identify the location of data without having to read all the logs (from GFS) that may have related data.

In Terms of the CAP-Theorem BigTable is consistent (using logs on a consistent filesystem) and available (as long as GFS and Chubby are available and there are enough resources to serve the desired table). As BigTable will stop working if it is in another partition than Chubby, it is not partition tolerant [Chang2006].

### 2.2.5 Megastore

Key-Value-Stores like the former described BigTable have the desired performance characteristics of a system that is satisfying for web-scale applications. But from the developer’s point of view, the rich datamodel, transaction and query-capabilities of traditional RDBMS are more easy to use and lead to a faster “time to market” [Baker2011]. This discrepancy should be solved by Megastore, providing query and transaction capabilities on top of a Key-Value-Store (BigTable). It does so by providing secondary indexes, that can be used by client applications to perform joins. Further, it offers the possibility to use transactions (which may be slow if the applications data is not structured well).

The user (developer of a client-application) needs to structure his data into entity-groups, where each group will be handled by a set of machines running a Paxos algorithm (sharding and data-locality are at the user’s hands). Inside of an entity group all writes will be serialized and transactions (limited to data inside this entity group) can run naturally. If transactions need to run on data that is not in a single entity group, Megastore uses a 2-phase-commit protocol (above the Paxos that handles the locality groups). Transactions use the three last versions of each row that BigTable keeps as Multi Version Concurrency Control (MVCC) to enable rollbacks. For performance reasons, a developer can alter the attributes that column-families of Megastore have in BigTable.

Ideally, there is a Chubby node and a BigTable cluster inside each data center that runs Megastore. Each Paxos node (serving an entity group) writes a log to the local BigTable, the other Paxos nodes (for the same entity group) should be located in different data centers (writing to the local BigTables) so power outage and even natural-disasters should not be able to get an entity-group to an inconsistent state or unavailability.

Indexes are updated after a write operation (outside the entity group) and, for this reason, may not be consistent at the point a write is successful. There are several kinds of indices that Megastore can maintain, for example the inline index that is maintained within a (list-) attribute of another entity, which is very comfortable for the user.

In each datacenter, there is one coordinator (that has no persistent storage and only Chubby as a dependency) which is registered in Chubby and responsible for the local topology (detecting network partitions etc.).
Megastore achieves good performance (35,000 writes/second, 230,000 reads/second) \cite{Baker2011} and is being used by more than 300 applications in 2012 \cite{Corbett2012}. The latency of Megastore is shorter than the latency of the underlying systems (through caching of the logs it is writing to BigTable).

In terms of the CAP-Theorem Megastore is consistent and has limited availability which can be better than the availability of other Paxos-based Key-Value-Stores (due to entity groups). Without contact to Chubby or the local coordinator it will not work.

### 2.2.6 Spanner

As many applications at Google used Megastore instead of BigTable, it became obvious that Megastore has some weaknesses. First and foremost, Megastore’s transaction capabilities are limited (if transactions touch more than one entity group).

Spanner provides a similar Application programming interface (API) to the application developer as Megastore does, but achieves much better performance. This is done by using better timing information which is available due to a Time-Api that was invented by Google. A similar approach (to timing) is described by Leslie Lamport in \cite{lamport1978}, but was first successfully build into a productive system by Google.

Spanner is build around this Time-Api which provides a total ordering on events in a distributed System that does not require coordination on every request (the Time-information is gathered before the request arrives). Timestamps in Spanner are a point in time (the local clock) plus information about the uncertainty that this clock may have at this point in time. To keep this uncertainty low, there are special time-servers, called “GPS-Nodes” and “Amageddon-Nodes”, where Amageddon-Nodes have an atomic clock instead of a GPS. The Spanner-Servers get their time- and uncertainty-information by polling these time servers every 30 seconds. With this technique Google achieves a worldwide clock uncertainty of under 1 ms \cite{Corbett2012}.

Spanner is a multi-version-concurrency-control database, read-only transactions can happen at a defined point in the past on a snapshot of the whole database and readers must never be blocked by writers (if not intended).

On bottom of the logical architecture of Spanner is a Key-Value-Store that uses Paxos for replication and consistency. For persistence, Google’s successor to GFS, Colossus, is used in a similar fashion like GFS in Megastore.

The abstraction on which a Paxos group operates is called a tablet, which consists of several parts of rowspaces. Every Paxos group’s leader provides services for transactions using a lock table (and a two-phase-commit-protocol in cooperation with the other groups leaders involved in a transaction). The lock-table is not used for tablet-local transactions and bypassed completely for single reads or writes.

The group leader’s state is stored into the respective Paxos group (tablet) itself, so it can be used for seamless failover if the group-leader dies. A group-leader has a time-lease, if he does not extend his lease (by starting a new instance of the Paxos-protocol), another
master may be elected.

The client-application defines directories, which are a contiguous keyspace with a common prefix. These directories (or parts of their key-spaces) are then placed by the placement-logic into tablets that hold data which is likely to be used in the same transactions or accessed at the same time. Location requirements associated with directories are provided by the client-application and used in the placement-logic. If a table experiences high loads, the contained data can be sharded either through separating the contained directories into different tablets or by sharding the contained directories themselves.

![Physical layout of Spanner](Corbett2012)

Figure 2.7: Physical layout of Spanner, from \cite{Corbett2012}

The physical layout of Spanner as shown in figure 2.7 is built around the spanserver which serves 100-1000 tablets. Each spanserver has to participate in the corresponding Paxos-group of every tablet it serves and be the group’s leader if elected. The mapping of directory to tablet and tablet to spanserver is provided by the location proxy.

The universe-master is primarily used to display status information. The placement driver implements the placement-logic and executes it with the help of the zonemasters. Zonemasters assign data to their spanservers. However, this is not covered in detail by \cite{Corbett2012} but it is stated that this structure allows atomic non-blocking database-scheme changes at a defined timestamp in the future.

Spanner achieves outstanding performance on a global scale, this is mostly due to the automated data-locality-aware placement (that minimizes the likelihood of transactions that involve more than one Paxos-group) and heavy use of an unusual time-model.

### 2.3 Summary

In this chapter we looked into different basic technologies and complex systems. We found basic concepts for achieving hard and weak consistency in different ways and under different assumptions. For increasing availability all concepts we found fall back to sharding
keyspaces, either in a hashring, a hyperspace or in the hands of the client application's developer.

When trying to achieve consistency as well as availability while being partition tolerant, the time model used in the system is very important. This is the reason why Spanner seems to be the most performant system, which should be reconsidered.

The commercially used systems either provide strong consistency with the help of Paxos or they provide weak consistency. Weak consistency involves conflict resolving in the hands of the client application with the help of another technology like MVCC.
3 Analysis

This chapter starts with an analysis of the environment, trying to find components in which the system to be developed could be used. From the functionality that these components provide, a list of requirements for the system to be developed is created.

Based on the findings in the research chapter \[2\] and the requirements, a design for the system to be implemented is developed.

3.1 Environment

Travelping sells systems in the AAA domain. AAA equipment is used by telecommunication companies to provide the services they offer to their customers. The sold service can be access to the telephone network, to the internet or in the recent past to a television network.

The customers from Travelping provide internet access via different kinds of wireless networks, such as WLAN, GSM, 3G and LTE networks. Sold products are hard- and software appliances (appliance means a computer or a virtual machine containing the software necessary to fulfill a specific task), reaching from access points to server equipment. Each product needs to be highly customized for the specific customer because of the differing topology inherent to the customer’s network structure and the varying kinds of wireless networks which are used between Travelping’s customer and its own customers.

The appliances are all based on a self-maintained linux derivate called TPLINO. Many user space programs that provide functionality are written in Erlang. Each appliance fulfills one or more roles in the component diagram which can be seen in figure 3.1.

The Travelping system has a centralized part which is either hosted by Travelping or by its customer in its own datacenter. The Rest of the system is distributed as it makes up the radio equipment which the user must directly interact with. The centralized part of the system is called Travelping Open Subscriber Server.

A client trying to access the internet will first contact a local WTP (Wireless Termination Point) which is a relatively simple device. The WTP forwards the client’s traffic to the SCG (Session Control Gateway) which asks the PCS (policy control server) for the services to be performed in this case. If the request is legitimate and should get forwarded to the internet, this will be done by the SCG, relieving the rest of the system from the user traffic.

The PCS asks other components (UDR, LDR and Charging) for their knowledge to find its decision. This knowledge includes information about related accounting balance...
Various kinds of session representations reside in the PCS which interact with each other to provide the behaviour, concerning access modes and charging options that the customer needs. The customer is associated with at least two kinds of sessions, network sessions that represent a connected device and service sessions that represent services (internet access) that the customer is authorized for. The WTP and the SCG are also represented by a session in the PCS which is used to collect statistics and to correctly register and handle the case where a WTP or SCG loses connectivity or produces failures.

The decentralized part of the system may also include the SCG (often several SCGs). For some customers which use mobile-phone-networks, state of the art network termination equipment may be used as SCG but will still be controlled by the PCS.

A web interface is always accessible for the client for registration and management purposes. For example authorizing a new device or putting money on an empty account are tasks that can be done via the web interface. Another web interface and a JSON API are used by the customer to perform operative tasks to his customers and to see statistic information.

Internal communication between the different components in Travelping Open Subscriber Server (TPOSS) is done in different ways. Most components use Erlang message distribution. Other protocols are used where standardized functionality is provided and third party components may be usable as replacements. For all used protocols there are Erlang libraries which are used by the respective components.

As a good member in this ecosystem, the software to be developed should use Erlang message distribution and offer an easy to use Erlang API.

Several parts of the Travelping system do not achieve the wanted scalability or redundancy requirements. The following is a list of components that qualify for a reimplementation with the help of the system to be developed here.

**PCS** (Policy Control Server) The PCS operates on session data implementing the logic needed for different kinds of sessions. If the system needs to be restarted, the sessions are supposed to have timed out inbetween, so that data is stored in-memory only.

In its current implementation a master/slave replication provides vertical scalability. The implementation has similar performance characteristics as chain replication 2.1.5. At the end of a chain it sends a message from the tail of the chain to the head which then replies. The PCS has no horizontal scalability (increasing the throughput of the whole system), which should be improved by the system to be developed in this thesis.

A single (non-replicated) loadbalancer is used which is a single point of failure for the whole PCS; this point could be improved by sharing the routing table with more nodes.

When building the actual implementation of the PCS, it was found that fast master
failover is essential to provide responses fast enough for the domain. As users may use a third party protocol stack on their computers/devices, a master failover cannot be handled in cooperation with the client. The silent failover must be fast enough not to break the protocol spoken with the client (which may be a cellphone with a predefined network stack) to provide for a good user experience. Technologies that rely on the Erlang cluster manager to find and resolve failures (like the gen_leader module) can take minutes to do so under bad circumstances. The reason for this is that if a net-split occurs, the messages for error-handling with which Erlang signals and detects problems are not arriving, and prior to removing a machine from the cluster, a long running timeout must occur otherwise.

**UDR (User Data Repository)** User related data is stored on-disk in the UDR. To identify a former user, secondary indexes are used to find the device-ID (for example an ethernet address). This functionality is implemented by a thin layer of Erlang which uses a BigCouch cluster.

The weak consistency model, that CouchDB (the underlying technology of BigCouch) uses, led to inconsistencies in the past. CouchDB allows conflicts to emerge in a cluster, but needs the user to provide a function which can resolve conflicts after their occurrence. Depending on the domain of the application using the BigCouch, this model may be applicable, increasing overall availability of the system. Under other circumstances the consistency model may work against the domain requirements, which is for example the case in accounting.

If an account, that is not allowed to drop below zero, is accessed from different nodes of the BigCouch cluster at the same time, it is allowed to drop to zero in both parts. The conflict resolving function can then not calculate the new balance because it would be below zero. Saving the effort spent on synchronizing preceeding a request can not in all cases be recovered in conflict resolution trailing the request.

As several items in a user’s record have the semantics of an account, the described problem occurred in the past. For this reason the system to be developed should provide strong consistency and provide scalability through other technologies (other than allowing inconsistencies to occur because solving them afterwards may be impossible).

**LDR (Location Data Repository)** The LDR is very similar to the UDR but works on location instead of user-data. It is also a small layer of Erlang using a BigCouch that has the same problems with inconsistencies as the UDR.

The system to be developed in this thesis should be usable to replace both systems, providing strong consistency and allowing for horizontal scalability.

**SCG (Session Control Gateway)** The SCG routes user traffic to the internet and controls traffic to the PCS. Most work is spent in forwarding traffic here to work more
efficient than a pure software solution a switch may be integrated into the SCG. For this purpose OpenFlow is used.

OpenFlow is a specification of a switch (hard- or software) that sends the first packet of every connection to a software controller to determine where the connection should be routed. If no hardware switch is required, a software switch is used. Several OpenFlow switches may use the same controller, but the controller has no inherent scalability concept which makes the Controller a possible bottleneck. On the other hand, it is possible to use several SCGs instead of a single one in order avoid the bottleneck.

A framework like the one to be developed here, which is able to be used in the PCS, is most likely usable in the OpenFlow Controller as well. This is due to the similarities (in-memory-system, failure tolerance and a scheme to achieve availability needed) between the two components. If the SCG is multiplied to reach horizontal scalability like it is done now, failure tolerance may be worse than with the system to be developed here.

Deployment-scenarios reach from one to three nodes in the UDR, LDR and PCS right now, but in the future should be able to scale to more than three nodes which should partially be done by reimplementing components with the help of the software developed here. The systems built by Traveling have an intrinsic distributed architecture where location specific tasks are done by components in that location. So the centralized part of the products is not expected to grow over the boundaries of a single datacenter.

In the Traveling system (especially in the centralized part) there are several components that could benefit from the framework to be developed here. The not (horizontally) scalable PCS could benefit from the framework. Consequently, this will be the focus for the requirements. Where possible the framework should also support other components.

### 3.2 Requirements

In this chapter we derive requirements for the framework developed here, based on the observations we made in the last chapter.

**R1 Programming language** As the framework will be used to reimplement components of the TPOSS system, it should use the same programming language as the implementations of the respective components do right now (making reimplementations more simple). All possible candidates are implemented in Erlang, so this is the programming language of choice. Traveling’s policy is to have internal communication using the most performant solution for messaging. In the future ZeroMQ may be used for messaging. But as this has not been tested yet in Erlang message distribution will be used in the framework.
R2 **Generic Api** As a framework, which is not bound tight to a single usecase, a generic Api must be provided using which the behaviour that the specific component (PCS, UDR, LDR, SCG) provides can be implemented (in Erlang, see R1).

R3 **Storage modes** The PCS stores its data in memory while the UDR and LDR store their data on-disk. Thus, the system to be developed must include a flexible data storage mechanism. As the focus for a first reimplementation is on the PCS, the option to store data on-disk is not central.

R4 **Horizontal scalability** A central problem for the PCS is a strong demand on horizontal scalability. So the framework should provide a concept for horizontal scalability which can be used in the case of the PCS. But it should also be usable in the other components which already use such a concept from the framework that is supporting the current implementation.

R5 **Strong consistency** Due to requirements from the domain (being able to handle account-like data) and due to bad experiences at Travelping with a weak consistency model, the consistency model must be strong (no eventual consistency model).

R6 **Fast failover** Experiments with other technologies have shown that fast master failover (within a regular response time, not relying on client cooperation) is essential for the usecase in the PCS.

R7 **Single node deployment** Deployment scenarios for TPOSS start from a single node, so that the framework must be able to run on a single node. The framework must not work efficiently in this situation but it should be there as an option for very small or test setups.

R8 **PCS implementation** As a proof of concept the PCS should be reimplemented using the framework. As the PCS is very complex and there are many customer specific implementations, a partial reimplementation, that can be used for testing purposes, should be satisfying.

The priority is on requirement R8 (PCS implementation) because the framework should first be used in this component. Other requirements like R4 (Horizontal scalability) and R6 (Fast failover) have a high priority because they are required especially by the PCS.
Figure 3.1: A simplified component diagram of the Travelping architecture: boxes are components, clouds define locations, lines represent the main message flow while broken lines are used for administrative messages and communication with an optional database backend.
4 System design

In this chapter a system design is developed with the aim to fulfill the requirements defined in chapter 3.2. In order to find technologies that can be used in our design, the technologies which were discussed in the related work chapter 2 are examined in order to how far they fit the requirements.

A technology which fits the consistency requirements and a technology to provide for horizontal scalability are chosen to be the basis of the system design. The design is then developed by discussing the most important components and their functionality.

4.1 Finding an approach to consistency

Some of the systems examined in chapter 2 use weak consistency models and systems built on these technologies will thus fail to support requirement R5 (strong consistency). Amazon Dynamo and the BigCouch respectively the CouchDB, that is used in the current implementations of UDR and LDR, are not considered further as they do not fulfill the requirements of the Travelping domain.

The only technologies, that we looked into and which can provide for consistency in the domain of Travelping, are the family of commit-protocols (2-phase-commit, 3-phase-commit and Paxos), chain replication and hyperdex.

Chain replication is, by itself, not resistant to net-splits unless the tail of the chain knows about the chain length and the members of the chain that have already processed the current request. Even if the tail has this knowledge (which could be introduced into the protocol), the preceding members do not know if the tail will be successful at processing the request which will lead to an inconsistent state if the tail produces a failure before having completely processed the request. This is one of the problems that is solved by the Paxos algorithm. It makes sure that at least a majority knows about a decision before it is made public. Lengthening of the chain makes chain replication respond slower (scaling vertically) but this is expected as no scheme for horizontal scalability is included. On the other hand, a system based on chain replication is able to operate with only a single node from a chain being available.

Hyperdex builds a chain through different indices and replicas of these indices, which makes Hyperdex a concept for scalability that builds on chain replication. Consequently it is clear that Hyperdex inherits the problems that chain replication has with keeping consistency in the face of netsplits.

Hyperdex has an inherent concept for horizontal scalability. Through hashing of values
the position of an object in a hyperspace is calculated and then mapped to the corresponding machine for the subspace. However updates of objects, that change many values and lead to a relocation of the object in the hyperspace, are very expensive because they can build a very long chain through all subspaces and all replicas.

Correctness is proven for Paxos and only progress is implementation specific. However, resistance to net-splits means to actually have a majority of the participating entities participate in each decision, which is the example that was first used to introduce the CAP theorem. The second phase of the Paxos algorithm is proven to be optimal considering the possibility of netsplits and arbitrary but non byzantine failures \cite{Lamport2002}. The possibility of net-splits and the inability of a chain replication based system to stay consistent in a splitted situation is the reason why chain replication does not support requirement R5 (strong consistency) as well as Paxos does.

For these reasons most complex systems working in real environments, which we looked into in chapter 2 use Paxos to achieve consensus on a basic level. On top of the consensus providing layer semantics of a database system are then implemented. In the case of Chubby a mixture of a database and a locking-server is built on Paxos.

The topology is managed by a cluster manager that is itself distributed with the help of the Paxos algorithm (many Google systems use Chubby for this). In the case of the big Google systems the topology is managed by a tree-like system of cluster managers which provide as a whole the routing table of the system.

Doing so, the tree resembles the physical topology having a manager per datacenter, then one per country/continent etc.. In the environment we aim for the system should be used in a single datacenter only, so a simple cluster manager, that is replicated but not distributed for increased availability, can be used. The basic design of the mentioned complex systems, having Paxos as basic technology to provide consistency, seems usable given the requirements.

A system, that has a similar aim as followed in this thesis, is gen_leader. It is not usable here because it relies on the Erlang cluster manager to detect failure (which leads to long failover times in some cases). gen_leader exposes the \texttt{Api} of a gen_server, which is an Erlang behaviour commonly used. The actual logic running in the gen_server and gen_leader processes is provided through the event handling methods and the functions that wrap event sending. Both gen_server and gen_leader provide methods for sending an event to the running instance which will trigger the respective event on the right node. The gen_leader module tries to elect a leader, has its state replicated and will failover to a replica on failure, offering a generic and easy to use way to replicating application logic.

The same kind of \texttt{Api} as used by gen_server and gen_leader will be used in the system developed here, so the \texttt{Api} should be very easy to understand and use for Erlang programmers.

Most Paxos based systems have the application logic executed on the node that sees a request first. Afterwards the the group members will find consensus about the delta for the
state-change that was triggered by the request. After finding consensus the node to receive
the request can respond. In this model the application logic needs to use data structures
that are known to the Paxos implementation, allowing to compute the necessary deltas.
In most cases this means that the Paxos system is a database which is used for storing
data from an application that has no inherent consistency features.

Another way to use Paxos is to find consensus only about the order of requests that a
Paxos group sees. If the application logic is implemented as a pure function, that does a
state transistion for a given request only depending on its state and the request itself, all
states in a Paxos group must be consistent if they started from an identical state. Sideeffects should be triggered by the system on a single node after processing the request or on
all nodes to make sure the client receives the response. The advantage of this approach is
that the user (developer of the client application) is free to use any datastructure (including databases) to store its state while the application logic itself gets replicated. Further
more the system providing consensus does not need to provide the respective data struc-
tures and their access functions, making it more easy to be implemented. Offering an \texttt{Api}
like \texttt{gen_server} and \texttt{gen_leader} is possible using this approach. It offers the possibility to
implement more specialized systems than using a database built on Paxos would offer. As
requirement R2 (generic \texttt{Api}) states the approach, which restricts the client application
less, is to be favored, which is in this case the ability to use an arbitrary state instead of
a tight system of datastructures which the framework would have to provide.

Simple design seems to be very important when implementing a system based on Paxos.
In \cite{Chandra2007} the problems are described that Google encountered when building
Paxos based systems. Many optimizations were built into Paxos, making it much more
complicated than the simple pseudocode that the proofs operate on. So Google didn’t im-
plement a generic Paxos library but instead developed Chubby as a tool to leverage Paxos
features while keeping the complexity of the Paxos algorithm hidden from developers of
client applications.

They also use a huge testsuite and a test environment for Chubby where every occurring
error can be traced back to its origin; randomly generated input is used to trigger the
errors. Backtracing and monitoring of a failure is done by restoring the random number
generators to the startvalue that produced a failure. As Travelping’s resources are limited
compared to those of Google, a managable (less optimized) Paxos implementation, that
does not need such sophisticated testing environments, should be used. The basic Paxos
approach is also preferred because it has failover included instead of needing an external
coordinator which would have to be optimized to achieve requirement R6 (fast failover).

Horizontal scalability is not addressed directly by Paxos but can be achieved, for ex-
ample, by using hashing to find the Paxos group responsible for a request.

In this chapter we found Paxos (and other equal commit protocols) to be the best
fitting technology to fulfill all the requirements regarding to consistency that the domain
implicates.
4.2 Basic design

The system developed in this thesis will be called Dike after a figure from Greek mythology. In figure 4.1 the basic idea behind Dike is illustrated. Requests get serialized and replicated through Paxos and the application logic provides responses for requests and makes the state transitions necessary.

A cluster manager manages the routing table of a Dike system (adding/replacing nodes and adding Paxos groups). The manager itself can be implemented as application logic running in a Paxos group (to make it resistant to errors). The basic design of a single Paxos group can be seen in Figure 4.1. If the system grows bigger than a single datacenter, the cluster manager must be implemented similarly to the cluster managers from BigTable [2.2.4] and Megastore [2.2.5] (in a tree-like structure) to not become a bottleneck.

Regarding the requirements, only for R3 (Storage modes), R4 (Horizontal scalability), R6 (Fast failover) and R7 (Single node deployment), it is not immediately clear as to why they are fulfilled by this design.

Paxos can equally be used on-disk or in-memory; only the assumptions under which the system will be guaranteed to work without inconsistencies do change by using in-memory storage. To keep the data consistent for an in-memory Paxos system, it is required that at no point in time less than a majority of the participating nodes of a group must be available. In the terminology of the original Paxos paper this means that a majority of the legislators must be working at all times. If too few participating entities are available, the system must be restarted to be guaranteed to be free from inconsistencies again. The
only components using in-memory storage, that are candidates for reimplementation with the help of Dike are the PCS and an OpenFlow controller. In both cases this behaviour is acceptable, but the failure must be observed so that an operator can restart the component. Observing this error is easy as a Paxos based system does not work at all if less than a majority of the participating entities are available, which leads to immediate timeout errors at clients that try to use the component.

The UDR and LDR require on-disk storage, which is the default usage scenario for Paxos. Because of the different requirements of the different components at Travelping, different database backends will be needed to fulfill requirement R3 (storage modes).

Horizontal scalability is not directly supported by the Dike. The reason for this is that Dike should be used as a hashring but the measure which is used to split the keyspace (and also the amount of groups into which the keyspace must be split) is depending on the client application.

If used in an idiomatic way, every part of the seperated keyspace will be operated by a distinct Paxos group. Horizontal scalability can be achieved by adding nodes to the cluster. The cluster manager will distribute the groups among all nodes which leads to a distributed load between the participating nodes. In the beginning, when there are many Paxos groups and only few nodes, this method is not efficient (because it adds overhead for the big group count) but it fulfills requirement R4 (horizontal scalability).

Fast failover is difficult to achieve when using multi-Paxos because the failing master needs to be recognized and the Paxos group needs to be reconfigured before continuing. When using regular Paxos (which implies an extra message round for each log-entry compared to multi-Paxos), a message will either be appended to the log and the response will be send out (may be done by all groupmembers), or the issuing group member fails before adding the message to the log. In this case the client will timeout and reissue his request to another group member. As the used Paxos implementation should be as simple as possible (because of the difficulties to implement a highly optimized variant), we use regular Paxos which also provides the fastest failover Paxos can achieve (fulfilling requirement R6 (fast failover)).

To achieve requirement R7 (single node deployment) at least a majority of a Paxos group’s members in Dike instances needs to run on the single node. This is not efficient but possible and only limited consistency features will be achieved. Should the single machine collapse, all data is lost (in the in-memory case). If the harddrive becomes corrupted in the single machine, all data may be lost in an on-disk deployment. Another drawback is that an on-disk deployment on a single node requires more writes to a single disk before reaching consensus for a logposition than a deployment writing to multiple disks on multiple nodes.

If the installation is not required to be distributed to more than one node, the business-logic may be triggered by a simple stub-logic because requests on a single machine are always easily serializable (which is all that Paxos does in this design). Doing so, the
overhead induced by Paxos can be circumvented.

### 4.3 Paxos statemachine

The underlying technology that Dike uses is Paxos; the statemachine which provides the Paxos protocol for Dike is explained here in detail. In figure 4.2 the states, state transitions and messages, that are sent out when transitioning, are illustrated. Some default transitions are missing for better overview but will be mentioned in the description.

Every Paxos group consists of five members that keep the distributed log associated with the name of the group. The reason to use five entities (which is also the size of Paxos groups in other systems, for example Chubby) is that, when taking one node out for maintenance, one node can still fail without making the group dysfunctional (the group consists still of three members which is a quorum).

In the original paper [Lamport1998] Paxos is described as a protocol that is spoken between legislators that represent a participating entity in the protocol. Mapped to the computer-world a legislator is represented as a process. The behaviour of this process is described as a statemachine here. For communication messengers are used which have properties that match those of network messages in that they may get lost or arrive at any point in time after sending.

Most Paxos implementations (like for example the one used in Scalaris which we considered to use) split the legislator into three simple roles where each one participates in one round of the three round protocol. The roles are called Proposer, Acceptor and Learner. Each of these three roles can be represented as a statemachine, sending messages to the statemachines that represent the other roles. Together they provide the behaviour of a legislator.

To make the legislator as a whole more simple we use a different concept, uniting the three statemachines which make up the legislator behaviour within a single statemachine. This has the benefit that only a single process needs to be running, making liveness checks and failure recovery more easy. Drawbacks of this approach have not been found so far. The idea to use a single statemachine as the legislator is taken from Libpaxos, a general purpose Paxos library written in C++.

In Dike Paxos is used to find consensus about the value at an index of a log. This relation is illustrated in figure 4.1. The values, which are stored in the log, are the requests sent to the Paxos group.

For every log position on each member of the Paxos group which keeps the log an instance of the statemachine in 4.2 is started. Internal Paxos uses round numbers to distinguish between different trials (different trial from the same member and different members trying need to be marked by different round numbers). In the “final” round one member is first in preparing and then in proposing state while all other group members (or at least one less than a quorum) become first acceptor and then learner.
All rounds after the “final” one will immediately skip to the final state “decided” when getting to know about the decided value from the “final” round. To achieve this there is a transition missing in figure 4.2 that goes from every state to the “decided” state when getting a “decided” message. If no message is seen for some time, every participant will go to the “NIL” state (if it is not in the “decided” state) from where the participant will start a new round on its own after another timeout. Thanks to this procedure a value can be found in the future, even if messages do not arrive, or if the proposer for the Paxos round fails. In other systems this would be called master failover but as we use the whole Paxos protocol, which is masterless, this is just standard procedure that will also occur if no error happened (due to timing issues with messages arriving at an arbitrary point in the future).

The exact messages we use in statemachine 4.2 which need to be sent in the “final” Paxos round of the statemachine, are illustrated in figure 4.3. It can be seen that at least two whole message rounds (that can each be executed in parallel) are needed before a
result is found and can be sent to the client. Afterwards another message round (the third) is required to spread the final value (which will also be found by the non-proposers if they try to propose a higher round than the “final” one).

### 4.3.1 Modes of operation

When using the full Paxos protocol one participant will be started in the preparing state. The prepare messages, it sends out, will start the other participants for this Paxos’s subject (log position) in the acceptor state.

To use multi Paxos with the same statemachine definition, the current master needs to start its statemachine in the proposing state and the propose messages, it sends out, will start the other participant’s statemachines in the learner state. If the master fails, the statemachines will fall back to the complete protocol and execute rounds with higher Paxos numbers that will come to a decided value (which could be “undecided”).

The mode, in which the Paxos statemachine operates (regular Paxos, multi Paxos or fast Paxos), depends on the component that starts the Paxos statemachines; in the case of Dike this is the Paxos log.
4.4 Paxos log

As Paxos only provides consensus on a subject, there needs to be a second component in each Paxos system that uses the found values and starts instances of the Paxos statemachine for subjects. In the case of Dike this component creates a log that must be (due to Paxos and its own semantics) fully concurrent between the members of the Paxos group. The component providing the log will be called Paxos log and the subjects are indices in the log. A subject must include the Paxos group’s name and the log position to be distinguishable from other Paxos statemachine’s subjects (so that mislead messages cannot interfere).

The Paxos log relays messages to Paxos statemachines or starts them if needed. The behaviour of the Paxos log must resemble the behaviour of a running Paxos statemachine towards the other statemachines. For example, if a Paxos statemachine is started (on a different node) for a log-position whose value has already been decided, the log must not start the according Paxos instance locally (again) but instead answer with a “decide” message.

Adding requests is done by the Paxos group members themselves. If a client wants to make a request, he needs to contact one member of the respective Paxos group. The member starts a Paxos round for the newest log-position, he knows to have no Paxos instance running, locally. If he won the round, the request got through and the response can then be generated by the group member that added the request. In case another group member wins the Paxos round for a log position, that we had a request issued for, we need to reissue the request for a higher log position.

To keep the state of the client application consistent between the Paxos group’s members, it is important to start with a state that is valid at some log-position (for example the initial state for index zero) in every node, and then to add one request after the other without ever letting a gap occure in the log.

4.4.1 Log cutting

As requests are only added to and never removed from the log, it will grow indefinite. In order to recover and restart nodes the whole group history has to be loaded (either from disk or from another member) to reach the current state. Consequently it must be possible to cut old requests from the log. The idea used here is to have the client application’s state stored at some log position and then to cut the log underneath that position, this way shortening the log and stopping unbounded growth (if the client applications state does not grow indefinite).

Cutting the log is not trivial because of the situation illustrated in figure 4.4. The requests in figure 4.4 are arithmetic operations being processed by an application logic that implements simple arithmetics. At this point one member of the Paxos group cannot progress any further because the requests required for its application logic to progress while
Figure 4.4: One group member has a state that cannot progress (arrows mark the index upto where the member’s state has progressed and crosses mark deleted log positions)

keeping consistency are not available anymore. Even if it (re)starts the Paxos statemachine instances for the respective subjects, the information (the missing requests) cannot be recovered. To solve this problem every member of the Paxos group has to have a good guess about the indices that the application states of the other group members have. If it is known in the group that every member has progressed beyond a log position, it may be deleted without the possibility to cause inconsistencies.

In Dike this is solved by introducing a special request which is added to the log and must succeed in a Paxos round. It tells the group that the issuing member’s application state has processed a certain log position. Requests may be removed from the log up to the minimum of all log positions in the group. As these special requests could be sent after every processed request (which would lead to five special requests for one real request), they need to be sent in intervals to not congest the log. This is done in Dike by each Paxos log within a variance of an interval. Thus not all group members send these messages at the exact same log position, possibly causing longer response times because of other requests being reissued.

In figure 4.5 this special message is included. The special message has the name “persisted_at” and takes two arguments; first is the position in the group member list of the issuing member, second is the processed log position. Group member’s positions reach from 0 for the leftmost member to 4 for the rightmost member. Once a member learns which request was appended to the log next, it will be able to proceed with the application state. Requests must not be learned in the exact order of the log but the application logic can only progress if there are no gaps in the log.
4.4.2 Dealing with IO

Dike should be capable to store data on disk (the log and the client application’s state for every group). Furthermore, it is required that the client application’s state can be imported from another group member (in case of a new node joining or in recovery) because the old log positions get cut which makes the initial state useless. In both situations, depending on the client applications state, massive IO may be done by the member before returning to regular operations mode. This would lead to requests running into timeouts while the group member that issued the requests is performing the IO operations if no countermeasure would be taken.

For this purpose, it is possible to lock a Paxos log, which means that the client application logic on the member that is locked will not receive any requests and no requests that are appended to the group by other members will be processed by the locked member’s application logic. Meanwhile, the Paxos log will participate (through starting Paxos statemachines and appending results) in future rounds and thereby stay responsive towards the other group members. This way up to four members of a group may have a locked log at the same time and the unlocked member will still be able to process requests. New values are learned while the application logic is locked; so the application logic can process requests, that were added to the log, while being locked quickly after unlocking.

If a Paxos log is getting locked, it will first have to process any remaining requests that have not yet won a Paxos instance but were issued by the locking member. Otherwise the remaining requests may timeout while the log is locked and could be processed afterwards, sending the response to the client that is not expecting it anymore. This problem only exists if request aftereffects are only executed on one node (they could be executed by every member). Afterwards the lock is granted by sending a response message when the conditions are met.
The request that locks a group member is not appended to the log, so only the locking Paxos log instance will get to know about this event. If added to the log it could prevent a group from starting if more than a majority of its members lost their state (because the starting nodes need to copy a valid state).

4.4.3 Membership change

Importing a valid state needs to be done very carefully because it may cause inconsistencies if members with a further progressed log exist but are not online. Should the group with the copied state add to the log without the member with the further progressed log being online, they may append inconsistent values. But if this was not allowed, a group could run into an unrecoverable situation although a valid state is available (here we are breaking Paxos for practical reasons). Adding members or changing group membership is coordinated by the master and will be described in chapter 4.5.

Starting a group requires a list of all group members to be known to all group members. The reason for this is that the Paxos statemachine needs to generate unique round numbers to work properly depending on the member’s position in the group’s member list. As, in all cases, a Paxos group consists of five members, only the operation to replace a group member by a node that is no member yet, is required to manage a group’s topology. This second special message is appended to the log if it succeeds the Paxos log will, for all log positions after the member change, accept only values that were issued by a member of the altered member list. The one member, that got excluded, must not process any further requests (for the ones it has issued it would execute the aftereffects).

To replace a member it will for these reasons be necessary to first lock the log of the member that should leave the group. This makes sure that no requests having been issued by the leaving member are still being decided on. Then, the joining member will have to import the leaving member’s state (other member’s states would be possible too, if the information about the processed log position, that has been shared with the group by the member we are getting the state from, is not smaller than the processed log position of the member we are replacing). Afterwards the leaving member must issue exactly one request which is the group membership change. If successful, the joining member is informed and starts participating in the group while the leaving member stops itself.

In case of a persisted Dike node that is restarting or when importing another member’s state, only the actual processed log entries, the client application’s state (plus the actual position in the log) and the list of the group members are necessary to prevail consistency. Information about the progress of the other member’s application logic can be gathered as the group is working. Until all information has arrived the log may simply not be cut.
4.4.4 Summary

In this chapter we described a component which can be used as a universal distributed log and whose consistency relays on Paxos. Two operations are supported (besides adding values); these are member changes in the Paxos group and sharing the local log position with the other members so the log can be cut under the point which all members have processed.

4.5 Master

The master is a component of Dike that manages the system’s topology. The role it plays is called a cluster manager in other systems. Through the topology that the master creates it is possible to implement a hashring or shard the client application’s data by another measure that is depending on the client application’s semantics.

The master itself is a client application in Dike that consumes a request log as described in the last chapter 4.4. This way the master has the same concurrency attributes as the client applications that he manages.

In order to be able to communicate with the master or to participate in the running Dike system, it is required that every node knows the five nodes which participate in the Paxos log associated with the name “master”. These five nodes (and their correct order) are the information that must be spread through an external channel (for example by the configuration of participating nodes). All other information that is required to participate in the Dike is produced at runtime by the master.

Topology information is kept by the master as a routing table, a list of nodes and a list of groups that were added. The list of nodes and the list of groups are kept for performance reasons so the routing table must not be scanned to find which nodes and groups Dike currently consists of. The routing table is a mapping from Paxos group name to the list of participating nodes. Using the routing table it is possible to address any Paxos group (and any member within the group) that is participating in the Dike.

Dike starts with an empty system that knows no nodes besides the master nodes and no groups besides the master group. Nodes and groups are added at runtime to the Dike. This may be done by manually sending the corresponding requests to the master when starting (for an in-memory system) or when starting for the first time (for an on-disk system).

In figure 4.6 an example of the topology, in which Dike operates and that is partially (the distribution of the Paxos groups) arranged by the master, is illustrated. The loadbalancer must be provided by the client application and can either dispatch requests itself using the dispatcher, or it can forward requests to any clusternode. If the client application can run a dispatcher itself, loadbalancers can be omitted because the client can then directly access the right nodes needed for a request with the information in the routing table.

Adding a group or a node, that has already been added, has no effect on the topology.
Figure 4.6: Example of a Dike deployment: six nodes and three Paxos groups (A, B, C) in the cluster, six clients accessing the system through two loadbalancers

the master keeps. So every participating node can add all groups and itself to the master without causing any trouble. This way the client application can use the same mechanism on every node when starting the Dike. The master nodes must automatically start the master Paxos log and the master application logic. However, as the information about being a master or not being a master must be known to each node before starting, this is no problem. A message for bulk adding of groups can be added to the master to shrink the amount of messages that are being sent on startup if the client application is built as suggested here.

4.5.1 Adding nodes

The algorithms which are used by the master to add groups and nodes, are very simple and there might be more sophisticated ones. When adding a node first the master selects the node which is initially participating in most Paxos groups. From this node Paxos groups are then selected for migration until the node participates in less Paxos groups than the average of all nodes in the Dike. Afterwards, the algorithm recurses, selecting the node that is now participating in the most Paxos groups etc., until the joining node has reached the average number of Paxos groups.

Actual sending of requests to start nodes is done in the aftereffects of the request that triggered the joining, so the algorithm described does not make any actual migration until all changes, the joining node implicates, are calculated.
4.5.2 Adding groups

When adding a new group to the Dike, the initial members are selected as the five nodes which participated in the least Paxos groups when the request for adding arrived. Removing groups could be done very easily, although it would require another special message in the Paxos log that stops the group.

4.5.3 Removing nodes

Removing nodes works similar to adding groups - the member with the lowest load (defined as participation in Paxos groups) is selected and will get group memberships migrated from the stopping member until it is above average load, then the node with the next lowest initial load is selected etc.. When doing so, it must be made sure that no node participates more than once in a Paxos group. Nodes cannot simply be removed but must be replaced. If no nodes besides the masters are in the Dike, all Paxos groups will have exactly the masters as their members (the order of the group membership list must not be the same in each group).

4.5.4 Possible improvements

So far the master only knows about the Paxos groups that each node participates in. It would be possible to have periodic status updates from each participating node appended to the masters log that includes more sophisticated information about the nodes status. For example, disk-, ram- and processor-usage are information that could be used by the master to select nodes. Furthermore, there could be another Paxos group, for example “statistics-server” that collects these information, so that the master is not strained with this task. The client application logic could also provide information about expected resource usage to improve the master’s decision basis.

If a system consists of ten or more nodes, the master could enforce Paxos groups to being always on an aggregated set of five nodes. This way it would not be guaranteed that, at most, two nodes at all can fail but instead for each aggregated group two nodes could fail.

The master manages the topology of the Dike system at runtime. As the routing table the master keeps is required to send a request to a Paxos group, the master would become a bottleneck if he was involved in every sending of a request to a Paxos group. To prevent this bottleneck another component, the dispatcher, is used to cache the routing table.

4.6 Dispatcher/Client

The dispatcher has two duties in the Dike; it is used to find the members of a Paxos group when sending a request, and it is used on nodes that participate as members in Paxos
groups to verify that the locally running Paxos logs are the same the master expects to run. To fulfill these tasks the dispatcher needs to have access to the master’s routing table. Accessing the routing table is done on a locally cached copy inside the dispatcher.

When routing table entries are changed on the master, the corresponding nodes are actively informed about these changes by the master (the information is pushed), so that the actual group membership can follow the topology change on the master as fast as possible.

### 4.6.1 Access from the client

Every client wanting to append requests to a Paxos log (thus sending a request to the Dike) needs to know the information about group membership that is residing in the routing table. So the dispatcher must be part of every client.

Not all entries in a Paxos group’s member list, that is available locally on the dispatcher, need to be proper for sending requests. As long as one entry identifies a participating node correctly, a request can be added to the Paxos group. This is because every distributed Paxos log keeps group membership itself. Thus from the point of view of the Paxos log the application logic’s group membership is always correct.

As routing information on the client does not need to be completely correct, it is enough to pull the routing table in intervals from the master. This relieves the master from pushing routing table changes to all clients on all changes (which could block the master depending on the amount of clients).

### 4.6.2 Starting and stopping Paxos logs

When group membership changes, the master informs the dispatchers on the nodes, that are selected to replace other nodes, through a message which contains the new group member list. Each affected dispatcher then alters his routing table (so future checks do not find Paxos logs that are not legitimately running on this node) and starts the corresponding Paxos log instances which will start the corresponding Paxos statemachines. The started Paxos log imports the state from the node to be replaced after locking the replacee’s log, see chapter 4.4. Through the communication inside the Paxos log the replaced instance is informed which will update the local dispatcher after learning about the new group member list through the log itself.

To start a fresh Paxos group the master sends another message, so that the (empty) state can be initialized by the application logic, and no state transferring or replacing needs to happen.

If a Paxos log is started because of differences between the local routing table in the dispatcher and the running Paxos log instances, the state cannot be initialized by the application logic. Instead, it needs to be imported from another member of the Paxos group (because the log may be cut which makes progress from the initial state impossible).
The starting Paxos log instance selects a node with a fitting state, locks the Paxos groups log on this node and imports the state.

As nodes with fitting state all participating nodes can be used. But if a far-progressed state is chosen, other group members may ask for requests at log positions that are unknown to the then participating node. However, as there are other nodes that have this knowledge, it is possible to do so, if at most four nodes join at the same time. So ideally the group member with the least progressed application logic state is selected. This can be done by asking all members simultaneously for their progression which is no guarantee because progress may change between asking and importing state.

After unlocking the node can simply start working because inside the group the (re)joining node is already known. The case of differences between the local routing table and the locally running Paxos logs can occur if the dispatcher already pulled the fresh routing table before the message arrives, telling the dispatcher to join a group that is in the new routing table but not in the old. If this happens, it is possible to stop the starting Paxos log instance (which is trying to import the state from another member now) and instead start it with an empty state to be initialized by the client application logic.

In case a message to replace a member arrives after the dispatcher pulled the already updated routing table, the Paxos log will be started without the information to replace another node. In this case the actual group members will ignore it because he is not part of the current group member list. After having received the replace message the Paxos log instance may be restarted the right way.

A stopping Paxos log instance stops the application logic and sends a request to the dispatcher which updates the routing table before stopping itself. If the dispatcher was not informed it will try to restart the group when checking for running groups next time.

To check for the correctness of the locally running Paxos log instances, a timer, that triggers the check in an intervall, is used.

### 4.7 Application Logic

The client application consists of a part which is independent of Dike, it sends request to the Dike and expects responses. Another part of the client application logic is executed directly before answering requests by the Dike, this way providing the custom behaviour the client application needs (for example a database).

This part uses an interface through which it is addressed by the Dike. Design of this interface is strongly influenced by the interfaces that standard Erlang components use (for example gen_server). The idea is to have a component, similar to a statemachine, that can be implemented through callback functions in the interface and that can be addressed through a send message function, the interface provides for sending messages to implementations of the interface (which is used by the external part of the client application).
Being able to cut the Paxos group’s logs (see 4.4) requires the application logic’s state to be importable and exportable. As it may be sent over a network connection or written to disk, the state needs to be serialized. A serialized state can be independent from the programming language and the structures a more restrictive system than Dike could support. This is because a byte or string datatype (which are often used for serializing) is supported by most programming languages and databases (and these state representations can even be converted into the other, using for example Base64 encoding).

Regarding the client application’s state the interface exports three functions: initializing the application state if the log is empty (this is done when the Paxos log first starts); initializing the application state from a former serialized state, and serializing the current application state (used by the Paxos log when cutting).

Similar to gen_server the interface offers a function for asynchronous replying. An action in the application logic implementation may either provide an answer as a return value from the function that gets called by the Paxos log, or it may omit a reply value and instead send the reply asynchronously through the reply function as part of another request.

Other functions, that a gen_server supports, could also be included in the Dike, for example changing the source code of the application logic implementations. For this purpose the Paxos log would need to handle a source code change similarly to a group member change. A message would get appended to the log that indicates the source code-version needed to progress requests after this message. After loading the new source code from a source and converting the client applications state to the new version, each member of the Paxos group could continue progressing with the client application’s state.

The following list gives an overview of all operations that must be implemented by a client application:

- **init** For starting at logposition zero the client applications state is initialized here.
- **init(SerializedState)** The serialized client application state is deserialized here when joining an already working group or when rejoining after losing the local state.
- **export** Here the state is serialized for storing or for being sent to a joining node.
- **handle_call** This function is used to make state transitions, to send responses and to trigger sideeffects.

### 4.8 Database Backend

Each Paxos log instance and the corresponding client application need to store some data: the client applications state, the log entries (future entries the application logic has not yet processed and old entries that may be needed by other nodes, down to the point where the log can be cut) and the group member list, see 4.4.
For storing this data, the Dike offers an interface which abstracts from a database. The
database is expected to be a separate local database that is not distributed itself, so it
is not required (and should not have these features for performance reasons) to have any
horizontal or vertical scalability features. As a Paxos log knows which data it needs (the
linear numbers that describe log positions are the keys), the databases are not required
to have any search or query capabilities which make fast no-sql databases a good fit for
usage in Dike.

Different databases have different performance characteristics regarding the access modes
that can be used. A database may provide the possibility to be accessed by multiple han-
dlers from one (operating system) process, or only from one handler per process. Different
processes may have handlers on the same database, or the database may only be accessible
from a single process. Depending on the usage of handlers, the database may work more
or less efficiently.

To accompany for this, Dike offers the possibility to access databases in different ways.
Either each Paxos log instance, running on a node, uses its own database adapter, or all
Paxos log instances on a node share the same database adapter. If the database supports
this, all Dike nodes (each containing multiple Paxos log instances) on a machine may
use the same database with each node having a handler. This option should be statically
configured together with the rest of the database backend’s configuration. Other options
are a folder on the harddrive for on-disk adapters and special adapter related options that
are simply passed through by the Dike, main usage should be performance optimizations.

Depending on the usage scenario the keys, each Paxos log uses to store the values for
index positions, have to be of differing form to be distinguishable from other Paxos log
entries. If each Paxos log has a distinguished database, it is possible to store values simply
under the log position numbers they have. If all Paxos log instances (in one node) use the
same database, the keys have to include, besides the log position, the name of the Paxos
log group. When multiple nodes are started on the same physical machine, it may be more
efficient to have all nodes use the same database which can then coordinate disk access
more efficiently (this options is not useful for in-memory databases). In this case Paxos
log instances belonging to the same group may use the same database, so the node name
as well as the group name must be included in the entries keys to be distinguishable.

Functions needed in the interface to the database backend to be usable from Dike are:
A function for opening a handler, with the filename as argument (open), another function
is exported for updating (storing if no value available beforehand) values (update) and a
third one for reading a value (get) and one for deleting a bunch of entries (bulk_delete).

Bulk deleting is used when cutting the log and when deleting old persisted states, see

All logpositions to be deleted (and with that the Keys in the database) are known
to the Paxos log, so this can be used without querying. The Paxos statemachine and the
Paxos log when not cutting, only need the update and read functions to access single log
positions. Calls to the bulk_delete functions should be (depending on the settings for
Another function is used by the database adapter to signal if the adapter is persisting
to disk or not; this is used by the Paxos log when rejoining a group. An in-memory
database-adapter will not provide anything useful when the node was shutdown, so the
client application’s state needs to be imported from another node. On the other hand,
a persisting adapter will always (when not corrupted on disk) provide client application
state and log entries sufficient to directly start participating in the Paxos group (because
the other members were not able to cut their logs when the node was unavailable because
they did not see any “persisted_state” messages in the log issued by the missing node).
This has the disadvantage that if a node is missing, the log on the other nodes in Paxos
groups the missing nodes is member in, will grow indefinite.

Depending on the database, that is used within Dike, there are different requirements
for the running system to stay consistent. Paxos itself has no special requirements but
expects the legislators not to lose their ledgers when leaving the parliament (at least not
more than a minority not to lose their ledgers). This is equivalent to a Paxos log member
not losing its database state.

When using an in-memory database, this means that at no point more than two partic-
ipating nodes of a group may get shut down. The same rule must be adhered when using
a database that does updates to disk asynchronously (which may be done to keep bigger
application logic states than the RAM could store in a Dike that has in-memory performance characteristics through caching). In an on-disk system where the database adapter
writes synchronously to disk, it is only required that at no point more than a minority
(at most two nodes) suffer from disk-crashes or other failures that corrupt on-disk data.

The following list which gives an overview of all operations that must be implemented
by a database adapter:

- **open** For opening a database handler.
- **get** For retrieving the value that was stored at a log position.
- **update** To create new/update old log position entries.
- **bulk_delete** For deleting one or multiple log position entries.
- **close** Used to shutdown the database adapter.
- **persisting** Signals if the database adapter stores to disk or to memory.

### 4.9 Summary

The design of Dike as described here can be used to implement a generic application
logic. The application logic is then setup in a cluster and used to answer requests in a
manner that guarantees the application logics state to be free from inconsistencies on its five replicas.

Client application logic must be implemented in the form of a statemachine, as every computer system can be seen as a big statemachine, this should fit many usecases. When implementing the client application’s logic, it must be noted that all data required for a decision must either be already in the client application’s state or in the request because the states of the group members, which should be completely redundant, may become diverse otherwise (data that is concerned with this may be random numbers or the time when a request is answered for example).

When using Dike as an in-memory system, it will only be able to operate efficiently in an environment where fast network communication between the participating machines is provided. This will be provided in a single datacenter. When exceeding these boundaries either the Paxos mode needs to be changed as done by Chubby (using multi Paxos). Further the master needs to be changed to become location aware and be able to distribute the Paxos group memberships accordingly so failure of a datacenter doesn’t affect the majority of a Paxos group.

When using Dike as an on-disk system, writing to disk (at least to harddrive) takes so long that network communication speed may not be as important. In this case the participating machines may be located in different datacenters without limiting performance as bad as expected in the in-memory usecase.
5 Implementation

This chapter describes the implementation of the system which has been designed in the last chapter 4 to fit the requirements developed in chapter 3. The Design as well as the implementation are called Dike.

It is Paxos based and written in the programming language Erlang. It would be possible to be implemented in another programming language but since many components are designed to be independent acting entities that communicate through asynchronous messages, support for such structures in the programming language would be very helpful. In Erlang light weight processes are first class citizens (meaning that processes are handled in Erlang similar to variables or functions in other languages and for messaging between processes there is even syntax support).

An overview of the system and how the components interact with each other can be seen in figure 5.1 which will be explained in detail in the following chapters. Figure 5.1 has a focus on the message flow of a request and its response through the Dike system and does not picture all possible communication relationships.

The components that were designed in chapter 4 have slightly different names in the implementation due to conventions for naming Erlang modules. In table 5 the names are listed.

<table>
<thead>
<tr>
<th>Component name in Design</th>
<th>Component name in Implementation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Paxos statemachine</td>
<td>paxos_fsm</td>
</tr>
<tr>
<td>Paxos log</td>
<td>gen_paxos</td>
</tr>
<tr>
<td>Master</td>
<td>dike_master</td>
</tr>
<tr>
<td>Dispatcher</td>
<td>dike_dispatcher</td>
</tr>
<tr>
<td>Application logic</td>
<td>paxos_server</td>
</tr>
<tr>
<td>Database backend</td>
<td>dike_db_adapter</td>
</tr>
</tbody>
</table>

5.1 paxos_fsm

A first draft of the implementation in Erlang was done by Kota Uenishi, inspired by libpaxos, but it had conceptual failures (there was no real distinction between the first and the second phase of Paxos) that led to a redesign according to the concept discussed in 4.3.

In Erlang a gen_fsm is used to implement exactly the statemachine described in figure 4.2 Erlang uses behaviours to offer the ability to reuse standard components in projects
Figure 5.1: Sequence diagram of a request passing through the Dike system. The perspective of the issuing group member and the Paxos round in which the request gets decided are shown here. Dashed lines represent messages which are not send in every request. The paxos_fsm is “activated” for the lifetime of the instance for the log-position in which the request is being decided.

written in Erlang. The concept of a behaviour is comparable to an abstract class in Java; it offers some functionality but also expects defined hooks to be filled by the implementation. Erlang components that are written using these gen_ behaviours can be integrated into the OPEN TELECOM PLATFORM (OTP) which is a framework that offers many options for programs that transcend a single computer.

gen_fsm stands for generic final state machine. Our implementation has three hooks to be filled by the software which uses the paxos_fsm. The first is used for sending a message to another paxos_fsm, the second is used for broadcasting to all other paxos_fsms that participate in this Paxos instance and the third is used to signal the using software a decided value when the Paxos instance has found consensus.

These hooks are needed in case this statemachine should ever be used by another coordinator (which is gen_paxos in case of Dike). Another coordinator could use the
paxos_fsm in another mode, for example fast- or multi-Paxos.

5.2 gen_paxos

gen_paxos provides the distributed log. It coordinates the locally running paxos_fsm instances belonging to a Paxos group. It is implemented as a gen_server.

To register paxos_fsm instances a process registry called Regine is used. The paxos_fsms send all messages dedicated to paxos_fsms participating in the same group and log position to the gen_paxos instances on the corresponding nodes. Whether the requests are valid is checked in gen_paxos regarding the current state (group membership, already finished paxos_fsm instances etc.) of the Paxos log; if that is the case, they are being relayed (using the Regine registry) towards the corresponding paxos_fsm instance.

When a decision is found by a paxos_fsm instance, it signals this event to its gen_paxos by sending a message. On receiving this message the gen_paxos tries to progress with the application state. For this purpose a function exists that is called update_log_complete; it is not only called when a log entry got decided, but also in intervals.

The update_log_complete function only progresses with the client application’s state when the next log position (after the last log position the client application has seen) is decided. There may be later log positions decided but the order must be preserved to keep the client application’s state consistent between members of the Paxos group. Requests that are used to manage the group’s membership list or to signal the application states progress of other group members are processed directly, changing the gen_server state of the gen_paxos instance.

Other requests are being send to the paxos_server and there handled by the application logic (the paxos_server implementation).

If update_log_complete is called (for example through the timer that calls this in intervals) and in the log there is a gap in front of the position that the paxos_server has progressed (that means a newer round has been observed but not the next one), update_log_complete actively starts the paxos_fsm for the next log position to learn the already decided value.

5.3 dike_master

The dike_master is responsible for the topology of the system at runtime. It is implemented as a paxos_server which makes it as failure proof as Paxos and the underlying database allow.

He does not support all functionality discussed in the chapter 4.5 about the master. Removing nodes from the Dike is not yet supported and the member list of the master cannot be changed at runtime.
The master’s group membership list is implemented as an environment value which must be configured before starting Dike. This is done in the production systems at Travelping through a program called Enit that is also used to configure all Erlang applications in TPOSS.

The master’s member list could be changed at runtime to start a system that runs on a single node (all five master instances running here) where nodes are added when needed. When adding nodes, the master instances should be migrated to enhance robustness of the system (if the single node with all master instances on it fails, the topology information is not accessible). Changing the master’s group member list at runtime should be implemented or the system should be started on a minimum of five nodes. For the PCS this is not very important because it will lose all state on a restart, so it can be reconfigured and rebooted when scaling up.

Node removal is not implemented because it is not needed for this thesis and the use case at Travelping is also focused on scaling up, not down. In the future, node removal should be implemented - how this could be done is discussed in chapter 4.5.

The collecting of statistic information mentioned in 4.5 is not supported by the implementation, and research about good algorithms in this area would be needed for a useful implementation. As no sophisticated information about the different resources load factor on a node is available, information about the client application’s resource consumption in this regard is also of no use.

5.4 dike_dispatcher

The dispatcher is started on every node that is participating in or sending requests to the Dike. When starting Dike on nodes that are not part of the master’s group, only the dispatcher is started.

It is implemented as a gen_server which is inserted into a supervisor tree. As such, it will automatically be restarted by the Dike’s top level supervisor (which is himself supervised by the OTP) if it should fail.

A timer is used to signal the dispatcher in regular intervals that he should refresh the routing table from the master and check that the locally running Paxos log instances are the same which are found in the routing table. Requests are performed using the dispatcher; if it is found that an entry in the routing table is not correct when trying to issue a request, the dispatcher is forced to refresh the routing table. To find Paxos log instances that are not in the routing table is very unlikely as the Paxos log ends itself (and the client application logic) if it finds the appropriate message appended to the log. If it finds a Paxos log instance not to be running, it starts the instance, telling it to find a valid state from the other group members.

When issuing a request to a Paxos group that is not “master”, a function from the dispatcher called “request” is used. The client application logic provides the name of the
Paxos group for which it may have to hash some attribute of the request or determine
the right Paxos group on another way. The Paxos group’s member list is read from the
routing table; this can be done from outside the gen_server process because the table is
an ets (Erlang term storage) table which has read-concurrency enabled. So all members
one after the other are tried to append the request. If all members should fail, the routing
table is refreshed and the same procedure is repeated.

5.5 paxos_server

The paxos_server is the Erlang behaviour which has to be implemented by the client
application to specify the logic which a Paxos group should follow. The name is a remi-

niscence to gen_server.

After successfully appending a request to the distributed log of a Paxos group, the
“handle_call” function which the client application must implement is called by all mem-
bers of the Paxos group. The state, which is initialized by the client application through
the “paxos_server:init()” callback is passed to the “paxos_server:handle_call(Req, From,
State)” callback as well as the actual request and a reference to the calling process.

The return values can either be an Erlang tuple consisting of the atom “noreply” and the
new state calculated. Or it can be a tuple consisting of the atom “reply”, an anonymous
function which has no arguments, and the new state calculated. In the anonymous func-
tion are all sideeffects the request needs to trigger wrapped together. Arguments for the
anonymous function are not required because it catches all values needed for the function
calls that are the sideeffects from the “handle_call” function’s scope.

One sideeffect included in the anonymous function may be a call to “paxos_server:reply-
(From, ReplyMessage)” that is used to send a reply message to the process that sends
the request which was initially stored in the Paxos log. The anonymous function is only
executed on the Paxos group member which issued this request and won the Paxos instance
for this log position. The anonymous function representing the sideeffects must not include
the reply; the reply can also be sent asynchronously in the future as part of another
sideeffects function.

Two more callback functions must be implemented for paxos_server to suite the Dike’s
needs: functions to serialize/deserialize the paxos_server state. The “paxos_server:export-
_state(State)” function is used for serialization and the “paxos_server:init(SerializedState)”
function is used for deserialization. Serializing does not mean to export a string or a bi-
nary, but to return an Erlang term which has the paxos_server state fully contained
without any references to sources outside this term. However, strings and byte sequences
can fulfill this requirement if they have no references to outside sources encoded.

All callback functions described here, except the ones for serializing and deserializing
state, are very similar to the callback functions that need to be implemented in order to
use a gen_server, making it easy for the experienced Erlang developer to use the Dike.
5.6 dike_db_adapter

The database adapter is used to store the entries for the log positions from the Paxos statemachine instances to read these positions from the Paxos log and to store the client application’s state and the Paxos log’s state in order to be able to cut the log.

A database adapter is created by implementing the behaviour “dike_db_adapter”. Opening a database is done by the function “dike_db_adapter:open(Filename)” that gets the value of a filename as parameter. The filename is taken from the config of the Dike application and is stored there under the keyword “db_dir”. An in-memory database will also get this value (or “undefined” if there is no such value in the config) but will ignore it either way.

Return value of the open function is a handler that gets passed to the other functions in the “dike_db_adapter” behaviour. The handler maybe an arbitrary Erlang term; in the following two chapters different handlers will be explained.

For reading values the behaviours “dike_db_adapter:get(Handler, Key)” is used and for writing values “dike_db_adapter:update(Handler, Key, Value)”. The update function is expected to insert new values and overwrite old entries at the key’s position.

When deleting the outdated log-tail the function “dike_db_adapter:bulk_delete(Handler, KeyList)” is used; it is expected to delete all entries that are matched by the items in the KeyList.

Another function, “dike_db_adapter:close(Handle)” for closing the database handle, must be implemented as well in order to stop the database adapter in a clean way.

Depending on the mode in which the database was configured to work in the Dike config (option “db_mode”), see 4.8, there is a distinct database adapter per Paxos group in a node or one for the whole node. Disk adapters may use the option to have several handlers to a single database used from all nodes on a physical machine.

The described database adapter interface is very simple as it does not require any query-capacities from the underlying database. The following chapters will describe an in-memory and an on-disk adapter that were implemented in the course of this Dike implementation.

5.6.1 In-memory Adapter

The in-memory adapter is a wrapper around an Erlang Term Storage (ETS) table which is included in Erlang OTP. An ETS is a simple table which is developed to be aware and supportive of a concurrent system.

The table used here can be accessed by any process in a node for reading and writing, so the handler, returned by the open function, is simply a reference to this table. All other functions required for the behaviour are one-line implementations that call ETS functions which fulfill the role expected from the adapter.
5.6.2 On-disk Adapter

A database driver for the MDB (Memory Mapped Database) was implemented for the on-disk adapter in the course of this Dike implementation. MDB was written to avoid a caching layer used in most databases for performance reasons. By a mapping from a RAM area and a file on disk that is provided by the operating system, it is omitted to write a caching layer in the database itself. This should especially perform well when using SSDs which are getting cheaper and will be used by Travelping in future products.

The database driver was implemented using the ENIF (Erlang Non Internal Function) interface which offers a direct interface to C code, because MDB is implemented in C. Unfortunately, the C functions, which are called this way, are not allowed to block the thread they are running in (because Erlang starts these functions in a thread pool that is not supposed to block), so an extra thread must be created for each adapter known to the Erlang runtime system. Inter-Thread communication decreases communication performance with the database adapter by magnitudes as all communication must go through the operating system, requiring context switches of the CPU, but is required as otherwise (when blocking these threads to long) the whole Erlang runtime system crashes. To absorb part of this performance decrease, bulk operations are used between the dike_db_adapter and the thread in which the actual access to the MDB happens.

The MDB driver, called EMDB (Erlang MDB), returns a so called “source object”, which is an object that is handled by the Erlang runtime system but where the contents are provided by the ENIF. A gen_server is used to keep the source object in its state. This gen_servers process ID is returned by the “dike_db_adapter:open(Filename)” callback.

The gen_server (which is the database handle inside Dike) is used to cache database operations and make bulk transactions to have as few inter thread communication as possible. In future Erlang releases it may be allowed to block the ENIF calling thread and with that abandon the bulk transactions.

Aside from this, the EMDB dike_db_adapter is rather simple as the MDB has appropriate functions to match the expected behaviour.

MDB can be started in different modi; either with a forced write through of every transaction or with faster asynchronous writing to disk. These modes have impact on the assumptions under which a Dike has Paxos attributes, which has been dicussed in 4.8.

5.7 PCS Implementation

The PCS is a component in TPOSS. Its role in TPOSS has been described in chapter 3.1. Its reimplementation using Dike is called “tposs_pcrf” which implements the paxos_server behaviour.

Every node on which the PCS is started initially adds Paxos groups with the names of “pcrf_vnode-1” to “pcrf_vnode-N” where N can be configured through the Erlang environment for the application. Thus, a hashring is being setup inside Dike.
The module which implements the paxos_server callback is called tposs_pcrf_vnode. This module itself exports two behaviours; tposs_pcrf_network_session and tposs_pcrf_service_session which are implemented according to the needs of the Travelping product. tposs_pcrf_network_session represents a session which is associated with a device while the tposs_pcrf_service_session represents a service which a device may use. One network session may have several service sessions associated, for example a calling and an internet service. Inside the paxos_server implementation (tposs_pcrf_vnode) two ETS tables (for service and network sessions) are managed. Service and network sessions related to the same device are kept on the same vnode so they can interact without locking other vnodes.

From the SCG radius is spoken with the PCS, as radius server a library called “eradius” is used in the PCS. Eradius is split into two parts; a loadbalancer and a handler function which is implemented by the tposs_pcrf. The loadbalancer is configured which SCG on which port is associated with a certain session implementation.

On the nodes that receive requests from the loadbalancer, the requests are being parsed into an Erlang record which is passed on to a radius handler callback in the session implementation. The vnode (Paxos group) corresponding to a request is identified by the PCS implementation basing on IP or MAC addresses associated with a session in the session implementation specific radius handler callback.

From the session implementation calls to the hashring are forwarded by using functions in “tposs_pcrf_vnode”. At this point, data needed from other components to be used.
inside the Paxos group may be fetched and passed on as a request's arguments.

On the hashring (in the paxos_server implementation) the actual state of the sessions is kept and can be influenced or read with guaranteed Paxos attributes. Callbacks in the session implementation may be used to provide the needed behaviour of the individual session implementation.

The PCS is under development so the interface to tposs_pcrf_vnode is changing while products are being developed further.

5.8 Summary

The implementation of Dike described in this chapter is successfully used at Travelping in the implementation of the PCS which has been described in 5.7.

It depends on an environment in which network communication is efficient and where Erlang is an available technology. As for now the client application must be implemented in Erlang, if another programming language must be used an interface to the programming language for implementing the paxos_server callbacks would be needed.

The on-disk adapter could be improved to using a truely generic one which is optimized for usage in the Dike (writing several logs into the same file while caching in memory for fast read operations).

To include Dike in a system it is required that the system's domain and its requirements allow for sharding the data handled by the Dike as horizontal scalability is achieved through this. This could be circumvented by implementing locking in the client application logic as required when requests need to access several vnodes. Locking, on the other hand, will involve several Paxos groups which will increase the time needed for a request and increase load on the system.
6 Testing

The Dike is tested in two different ways in the course of this thesis. Unit tests assured the functionality of the system were being used during the development. This is common practice when implementing in an interpreted language without typing (Erlang is compiled to bytecode which is interpreted by the beamvm) because the compiler can find less bugs then in a language that is compiled to machine code or uses strong typing. We are using the unit tests which are described in chapter 6.1 to check if the system can fulfill the guarantees to failure tolerance which should be inherited from Paxos.

In a separate test, performance of the Dike is tested in the environment in which it is used at Travelping. This is done using the PCS implementation developed with help of the Dike and described in chapter 5.7. One aim of the performance test is to evaluate the horizontal scalability of the system.

The client application (PCS implementation) and the test setup are being described and afterwards the test results and an evaluation of them can be found.

6.1 Unit Tests

For unit testing a very simple client application is used which is an arithmetic server. It starts with a state of 0 and can add, substract, multiply and divide the state with another (integer) number, creating a new state with every request.

The test cluster is setup, using the Erlang common test module for the Dike application, on the local machine. The size of the test cluster varied between five and twelve nodes and all tests have been executed several times.

The unit tests include the following test cases which are concerned with Paxos attributes:

- In the first test case, the consistency of the Dike is verified after some requests have been processed. The requests may be generated by other test cases or they are simply generated to check the consistency of the system after the client application’s state has changed in the vnodes. This is done by comparing the client application state from all members of a Paxos group amongst another. Doing so for all Paxos groups the consistency of the whole system is verified.

- In the second test case it is verified that gen_paxos instances and their related paxos_servers are able to rejoin the system after an error forced them to stop. First requests are being added to the system, then one or two gen_paxos instances and
their related paxos_servers are being stopped in a Paxos group. After adding some more requests in their absence they are restarted by the dike_dispatcher on the involved nodes. At the end, consistency of the system is verified.

- In the third test case, the recovering mechanism of nodes is verified. While the last test case verified that a single group member can recover, here recovery of the whole Dike application is verified. If one or two nodes are not started immediately, the not initial application state needs to be recovered by the late arriving nodes. To provoke this situation, a cluster with three or four nodes is started. Requests are being send to the system which alter the client application state in the Paxos groups. The nodes that were started after the requests have been progressed need to import the client application’s state from another group member or to replay the log from the initial state. To verify that this is done correctly, the consistency is checked afterwards.

These test cases make sure the system is consistent after having up to two failing nodes and that the recovery mechanism for failed nodes is functional. Other test cases which, for example, check the behaviour of the database adapters are existing but they are not related to the goals of this thesis and are, for this reason, not mentioned here.

6.2 Client Application

The reimplementation of the PCS is used as client application.

The session implementations used for the series of tests done here are “tposs_pcrf_demo_session” and “tposs_pcrf_demo_service_session”. With in the distributed vnodes, a new session is stored or a previously stored session is looked up.

The test tries to examine the raw performance of the Dike system without complex client application logic that could take a long time to execute. On the other hand, the eradius server used introduces extra complexity not required for a raw Dike test, but for the PCS in general. Thus it cannot be omitted without stepping back to a purely artificial test client application implementation.

6.3 Benchmarking Software

For benchmarking a framework for distributed load testing called “Tsung” is used. Tsung is written in Erlang and generates radius requests using a module that was implemented in the course of this thesis as there existed no such a module beforehand.

Tsung is distributed itself which means that a single test run may send requests from several machines that are controlled by tsung and the results of the participating machines are collected on the machine that started the test.

Configuration of the Tsung setup is done through an XML-file. Here, the message sending nodes, the servers/ports to send messages to, and the behaviour of simulated
users are configured. A single user is configured in a way that resembles the behaviour of a user in the TPOSS system. First, the user sends an authentication message and then, in regular intervals, accounting messages. When stopping he sends an accounting stop message. The interval in which accounting messages are being send is set to 30 seconds, real users in TPOSS send these messages approximately every five minutes, this way the usercount in the real system may be ten times higher than in the test.

For each user participating in a test Tsung uses an individual port which limits a testserver to have at most 65535 users (a little less because other ports are needed for communication with the other testservers, DNS lookups etc.) because a network interface cannot have more open ports. Users are added in arrival phases, in this test we use one arrival phase that starts users in a linear manner (N users per second) until the system is overloaded at which point the test is aborted.

The amount of users starting per second depends on the test-run, a Dike system consisting of more nodes will have users added faster so all test-runs take approximately the same time. The Tsung setup is adapted in a manner that each testrun overloads the tested cluster within 20 to 30 minutes.

Overloading is defined by the needs at Travelping, products must fulfill SLAs that specify the time which a request may take to be processed by the Dike. So the tested cluster is defined to be overloaded here when the mean request duration exceeds 20 milliseconds.

### 6.4 Test Setup

In this chapter the actual test setups are described from the used hardware to the distribution of the required (software) components.

For the actual testing we have six Lanner FW-7568C at our disposal, these are rather small servers with a dual core Atom D525 CPU and 2 Gigabytes of RAM. They have no harddiscs but use a Compact Flash card to store the operating system and other applications. Each server has six Ethernet ports but only one is required through which they are connected to a Gigabit Ethernet switch.

This setup with six times the same hardware for the Dike cluster is more precise concerning the behaviour of the PCS in an actual productive system than a setup with virtual machines because of network communication that may not behave as unreliable in a big host system as in a real network of machines.

For generating requests with Tsung two office computers at Travelping are used, they have Intel i5 CPUs with four cores and eight threads. It is only required to use two machines because of the port cap which only allows about 60,000 users to be simulated by Tsung per machine, the processors are not working to their capacity in the testruns.

For testing the performance and scalability of the PCS we install a PCS on a single machine and test it with Tsung. Afterwards, the other machines are added to the cluster and tests will be executed until we have all six Lanner servers participating; this will
allow to measure the scaleup in the amount of machines that are used in the products at Travelping.

Placement of the loadbalancer(s) is not directly clear because the loadbalancer adds some load to the nodes which are running it. Experience shows that the loadbalancer produces (at least in the test cases up to three nodes) less load than a clusternode running on the same machine. So we derived a layout in which the loadbalancer does not interfere with the whole system performance which is shown in the following tables. Another possibility would be to have the loadbalancer on external machines but in products at Travelping the loadbalancer will be running on the same machines as the cluster nodes. Ideally five or more machines are available for the PCS which have each an instance of the loadbalancer, distributing the load introduced by running the loadbalancer uniformly.

In the following table an X represents an instance of the component which is described through the row running on the machine that is described through the column. Master nodes are also cluster nodes but in the test case with six nodes one machine is no masternode, in the other cases, all machines run at least one node that is in the master node list.

Here 6 test setups are described that will each be tested with an identical (except from the user arrival rate) Tsung setup.

<table>
<thead>
<tr>
<th>Machine</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Loadbalancer</td>
<td>X</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Masternodes</td>
<td>X X X X X</td>
<td>X X X</td>
<td>X X</td>
</tr>
</tbody>
</table>

Table 6.1: Machine and component layout for a one-node cluster

<table>
<thead>
<tr>
<th>Machine</th>
<th>1</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Loadbalancer</td>
<td>X</td>
<td>-</td>
</tr>
<tr>
<td>Masternodes</td>
<td>X X X X</td>
<td>X X</td>
</tr>
</tbody>
</table>

Table 6.2: Machine and component layout for a two-node cluster

<table>
<thead>
<tr>
<th>Machine</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Loadbalancer</td>
<td>-</td>
<td>X</td>
<td>-</td>
</tr>
<tr>
<td>Masternodes</td>
<td>X X X</td>
<td>X X</td>
<td>X X</td>
</tr>
</tbody>
</table>

Table 6.3: Machine and component layout for a three-node cluster
### Table 6.4: Machine and component layout for a four-node cluster

<table>
<thead>
<tr>
<th>Machine</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>Loadbalancer</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>-</td>
</tr>
<tr>
<td>Masternodes</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
</tbody>
</table>

### Table 6.5: Machine and component layout for a five-node cluster

<table>
<thead>
<tr>
<th>Machine</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>Loadbalancer</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>Masternodes</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
</tbody>
</table>

### Table 6.6: Machine and component layout for a six-node cluster

<table>
<thead>
<tr>
<th>Machine</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>Loadbalancer</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>Masternodes</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>-</td>
</tr>
<tr>
<td>Clusternodes</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>X</td>
</tr>
</tbody>
</table>
6.5 Results

This chapter shows the graphs that were generated by Tsung from the testruns described in the last chapter.

The rate of requests per second and the mean request duration are shown, so the point where each cluster configuration exceeds the SLA of 20 milliseconds and with that the performance of the cluster at that point can be identified.

![Figure 6.1: Results of the Tsung test on a one-node cluster](image1)

![Figure 6.2: Results of the Tsung test on a two-node cluster](image2)

There is a clear progression recognizable up to five nodes. With six nodes the result is not completely as expected. In table 6.7 the estimated requests per second, where the average response time took longer than 20 milliseconds, are listed for each testrun.

6.6 Evaluation

In this chapter the requirements from chapter 3.2 are compared with the implementation of the Dike. Some requirements are evaluated quantitatively on basis of the testresults from the last chapter, while others are discussed on a theoretical basis or by reference to unit tests that were used during development.
Programming language As programming language Erlang has been chosen and used as stated by the requirements. For communication Erlang message distribution is used, but all communication in the Dike is internal; external components are connected through the application logic where required in the PCS implementation. After all, it would still be possible to use other messaging systems than Erlang message distribution. For the future it may be an option to use nano-MQ (Message Queue) in the Dike because of its promised performance characteristics.

Generic Api The used Api resembles the behaviour of gen_server, gen_fsm etc. where possible. Some callbacks for serialization and deserialization are not used by the mentioned concepts but their purpose and the way to use them is easily understood. The PCS reimplementation uses this Api and so far the Api was sufficient for supporting the semantics of the PCS. On the other hand, only few programmers have used this, and all had a first hand introduction, so usability of the Api cannot really be evaluated here.

Storage modes The Dike implementation supports both storage modes through different database adapters. For the future this means that database adapters for different storage backends could be added.
One possibility, that is used for example at Google, would be a distributed storage system that is itself based on Paxos (here it could be based on Dike). Of course, the database layer would have to have different access modes to the storage medium (for example asynchronous write through) which could lead to different reliability requirements, see 4.8.

Both included database adapters are tested by the unit tests used during development. The unit tests showed that prior to productive use the on-disk adapter (and code that is executed only if a on-disk adapter is found) needs some further development because in rare cases recovery does not work as expected.

**Horizontal scalability** The test results of the test for horizontal scalability are illustrated in chapter 6.5. In table 6.7 the point, where each cluster configuration broke the SLA is extracted from the graphs.

In regards to breaking the SLA, a clean progression from each test case to the next is observed. Two testruns did not execute as expected though.

First, the test case with a two node cluster shows alternating spikes in the amount of requests per seconds and punctual spikes in the mean time a request took. In table 6.2 the cluster configuration for this testrun is described. In this test case
Table 6.7: Requests per second the cluster progressed when breaking the SLA.

<table>
<thead>
<tr>
<th>Cluster size</th>
<th>requests per second</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>115</td>
</tr>
<tr>
<td>2</td>
<td>200</td>
</tr>
<tr>
<td>3</td>
<td>430</td>
</tr>
<tr>
<td>4</td>
<td>600</td>
</tr>
<tr>
<td>5</td>
<td>780</td>
</tr>
<tr>
<td>6</td>
<td>880</td>
</tr>
</tbody>
</table>

one machine runs three masternodes while the other runs two. As three nodes are sufficient for a quorum, the machine running three nodes can progress without cooperation of the other machine. This leads to the second machine starting Paxos instances for log positions that the first machine’s nodes have already decided on. The created imbalance comes to the point where requests being progressed by the second machine (two nodes containing) are not added to the log, which means that they cannot be progressed any further, and especially that no response is ever send.

The described effect only appeared with real machines in this configuration, not when testing in a virtualized environment. We suspect the differences in network communication inside a machine, compared to communication over an ethernet network, to be the reason for it.

A solution for this problem could be to capture the current rate at which requests are added to the log and then to choose a log position in the future to which the application logic will have caught up fast instead of choosing the next unknown position which may have already been decided by a majority. If the application logic then does not progress as fast as expected, the log could be filled with NOOPs (no operation) by the issuing member.

Second, the test case with a six node cluster did not go through as expected. While in terms of the SLA the capacity of the cluster, when adding node six, increased by 100 requests per second, the rest of the test did not progress as expected. When comparing the point where the cluster failed, not accepting any requests anymore, the five node cluster reached roughly 1000 requests per second while the six node cluster already collapsed at about 900 requests per second. When inspecting the logs generated in this test case, it was found that communication between nodes seems hindered in the described setup. Log entries from the Erlang/OTP runtime system were found stating that other nodes cannot be reached. These kind of error messages are only generated when a node is failing to contact another node for over a minute, so something with the setup is expected to go wrong in the 6 node cluster case (normally single messages are sent within nanoseconds).
**Strong consistency** This requirement is fulfilled by choosing Paxos over other (weak) consistency providing algorithms. So far inconsistencies only occurred induced by bugs during development or by usage of the on-disk adapter which needs further investigation and debugging before it can be used in a production environment.

Included in the unit tests are multiple cases of Paxos groups that lose one or two members who recover and are checked for consistency afterwards.

**Fast failover** By choosing to use the full Paxos protocol instead of a further optimized variant (which would also have been more difficult to implement) the failover may be part of every appending operation to the log.

However, this does not guarantee that the client gets the right answer after a specified amount of time. If response messages are lost or requests don’t reach the cluster node selected for appending the request, a client will still get no response. The cluster node issuing a request could also fail after issuing but before answering; this would lead to a response not being send out. Inside a Paxos group consistency is guaranteed not on a single node but instead when asking a quorum of the group.

So the client still may have to resend requests or ask about the whereabouts of earlier send requests which must be included into the protocol spoken between the client application and the outside world.

**Single node deployment** This requirement has been tested in the unit tests and also with the cluster configuration in table 6.1 in the corresponding test case. Dike can be used in a single machine scenario. For more efficient usage of the given resources while not giving up redundancy (as only one real machine is involved), a stub logic could be used to trigger the application logic on requests that are naturally serialized on a single machine.

**PCS implementation** In a current project at Travelping the reimplementation of the PCS with the help of Dike is being used. Not all behaviour that is needed for the final product is already implemented but so far the required behaviour could be implemented using the API Dike provides.

Right now it is deployed in a setup similar to table 6.1 (in a one-node cluster) but for the future bigger installations and use of the already mentioned stub-logic is planned.

Summing up, it can be said that Dike fulfills all requirements, although there is much room for improvement, for example some flaws in the current implementation regarding on-disk persistence. Further development and testing is required for the on-disk adapter before it can be used in production.

Furthermore, the reasons that lead to the unexpected behaviour of the six node cluster described in table 6.6 need to be inspected.
The concept of Dike, to add requests to a distributed log which are consumed by the application logic running on the log-members has proven to work and support an application logic with a freely chosen structure as its state.
7 Conclusion

In this thesis a framework was developed that uses a generic application logic which can be setup in a cluster where it is guaranteed to tolerate a defined amount of errors. Depending on the requirements derived from domain of the application logic the availability of the system which is called Dike can be increased through sharding or hashing. In our research we did not find a framework which allows for a generic application logic as well as guaranteeing strong consistency and a concept for horizontal scalability. The contribution of this work is such a framework which can be used in an Erlang environment.

In the research chapter we found that only very few technologies can actually guarantee for a consistent state in a distributed system. Using the Paxos protocol consistency is guaranteed to prevail while tolerating net-splits and other failures. Dike was developed to tolerate non-byzantine failures (a failed machine is expected to not send out malformed messages) from up to two participating machines. The ability to tolerate two failing machines was chosen because this makes it possible to disable one node for management reasons while the system can still tolerate one machine failing due to unplanned errors. For tolerating more failing nodes, the Paxos group’s size would have to be increased which would lead to more communication overhead and thus decrease performance.

The CAP-Theorem implies that it is not possible to achieve consistency as well as availability while being partition tolerant. Because of this we have developed a system that is consistent and partition tolerant but the availability must be created by the design of the client application. To do this the client application can start any Paxos groups (each Paxos group can be seen as a seperate server) in Dike and spread requests among these groups. The same principle is used in other systems, for example in no-sql databases (known as hashing or sharding there).

In our research we found one system (which also uses sharding) that has an approach that may exceed the boundaries that are expressed by the CAP-Theorem. Spanner 2.2.6 uses a time model which bases on a subsystem for achieving a very accurate time in the whole system which was not considered when creating the CAP-Theorem. If the boundaries of the CAP-Theorem apply to spanner should be reconsidered.

The basic idea behind Dike is to have a generic application logic which behaves like a server or a statemachine whose state is replicated among five machines. Requests to the application logic are being serialized by using the Paxos protocol. As all replicas of such a server (which are called a Paxos group) are started with the same initial state and see the same requests in the same order, the state in all Paxos group members must be the same when examined at the same position of the distributed log if the application logic
is processing requests accordingly.

The implementation of Dike is focused on supporting the reimplementation of the PCS at Travelping. The PCS is a component that manages sessions from other parts of the TPOSS which interact inside the PCS. Sessions are being kept in-memory by the PCS, so the implementation of Dike is dedicated for supporting an application logic which can lose its state when the system needs to reboot efficiently. The assumptions under which the system is guaranteed to be consistent are depending on the storage backend used. Where a system using a database adapter that persists data to disk may lose data on up to two harddisks in a Paxos group, in an in-memory system at no point in time more than two nodes in a Paxos group may lose their state (for example through rebooting).

Testing the PCS implementation which uses Dike for functionality resulted in showing that the system fulfills the consistency guarantees inherited from Paxos for whom the system was designed. Two machines of the Dike cluster may stop working or a net-split may occur that separates up to two machines from the rest of the system without a decrease of availability or consistency.

Horizontal scalability is tested in a setup similar to the way the PCS is being used in Travelping products. The test results showed that horizontal scalability is achieved in terms of the SLAs which are used at Travelping but some test runs showed unexpected behaviour that needs further investigation when using the PCS in bigger setups than tested here.

The persisting database backend will also need more testing and performance optimizing before it can be used efficiently in other components of TPOSS. Important for the efficiency of a persisting database backend is the used kind of storage. Different database adapters will have differing performance characteristics depending on the kind of harddrive, RAID or SSD, onto which it is storing its data.

The Paxos implementation used in Dike can be improved to make the system overall more performant by saving a communication round between the Paxos group’s members for every request. To do so, (using multi Paxos) the coordinator will have to be changed to using epoches in which single proposers for each Paxos group are elected which can start the Paxos protocol with the second communication round. This and other possible improvements to Paxos have not been done in the course of this thesis because it is considered to be rather difficult. But if the effort is taken to do this, it will improve the system’s performance with the drawbacks of the more complex implementation and slower master failover.

Another feature that Dike is missing which is used in Erlang is the ability to change the implementation of the client application in a running Dike system (hot code swapping). How this could be done is discussed in the system design chapter 4.7. More complicated but also possible to implement would be the ability to exchange the implementation of the Dike itself in a running system. This could be done using the mechanism which Erlang offers for the used components like gen_server and gen_fsm.
So far, the used model is sufficient although requirements from client applications which are not yet known may lead to a more complex client application API (which would also be required for hot code swapping).

Finally, other more specialized systems to help implementing client applications may be build on top of Dike. Such systems may be domain specific databases or lock servers that would allow to use a different approach than used in the Dike itself for building distributed systems.
Glossary

**AAA** Authentication, Authorization and Accounting.

**Api** Application programming interface.

**Enit** a program used for administrating Erlang applications.

**ETS** Erlang Term Storage.

**horizontal scalability** means to increase the amount of requests which can be handled by a distributed system through adding more computers to it.

**MVCC** Multi Version Concurrency Control.

**OTP** Open Telecom Platform.

**Regine** An Erlang process registry used by Dike.

**Scalaris** an experimental Key-Value-Store using Paxos.

**SLA** Service Level Agreement.

**TPOSS** Travelping Open Subscriber Server.

**vertical scalability** means to increase the amount of errors (failing nodes) which can be tolerated in a distributed system by adding more computers to it.
Bibliography


