Scalability and Performance through Distribution
An approach to distribute a standalone Erlang implementation of Redis

Joachim Nilsson
Abstract

Scalability and Performance through Distribution

Joachim Nilsson

This report evaluates the possibilities of scalability and increased performance by distributing a system. Also, it will illustrate the relative ease of implementation using Erlang. The illustration is made by distributing the stand alone key-value datastore Edis. Edis is an Erlang implementation of Redis, it is a project under development at Inaka Systems. The distribution is achieved by using mechanisms provided by Erlang, supervisors and monitors. Furthermore, logic to enforce consistency and synchronization of the system is implemented. The future ambition of Edis is to combine the speed of Redis with the scalability of Riak. In this report it is shown how distribution is a key step for achieving this ambition.
Acknowledgements

    Chad DePue
    Danielle Gold
    Martina Freers
    Fernando Benavides
    Marcelo Gornstein
    Marcos Almonacid
    Lucho Appili

I would like to thank all of the above for the opportunity to learn about new exciting technologies and for getting a life experience in Buenos Aires that has come to define the current direction of my life, professionally and personally.

Last but not least, I’d like to thank Justin Pearson for having the greatest patience that I have ever come across.
# Table of Contents

**Introduction**
- Clouds .............................................. 4
- Distributed systems ........................... 5
- Hash Tables ...................................... 5
- Key-value datastores and NoSQL ..... 6
- Logical Time ...................................... 7
- Global Snapshots ................................. 8
- Developments in Functional Programming 9

**Theory**
- The context to motivate a Key-Value datastore 10
- Existing products .............................. 11
  - Redis ............................................ 11
  - Riak ............................................. 13
- Initial state of Edis .............................. 14
  - System Structure and Client Interaction 14
  - The Role of the Backend Database 15
- Erlang/OTP ........................................ 16
- Research on Consensus Algorithms .... 18
  - Consensus ..................................... 19
  - Paxos .......................................... 20
  - Gossip protocol ............................... 21
  - Vector clocks ................................ 22

**Problem and Goal**
- Consistency, Availability and Partition Tolerance 26
- A framework for multi-node structure 27
- Executing operations and synchronizing the cluster 28
- Classifying CAP combination for Edis 28

**Method**
- Tools .............................................. 28
- Implementation Plan .......................... 29
  - Design of a framework for multi-node structure 29
  - Executing operations and synchronizing the cluster 29

**Resulting System**
- Selection of consensus algorithm 30
- The framework supporting multiple nodes 31
- Executing operations and synchronizing the cluster 32
  - Operation requests: The multi-node interface layer 33
    - Cluster synchronization: Implementation of Vector Clocks & Timestamps 35

**Testing**
- The framework supporting multiple nodes 39
- Executing operations and synchronizing the cluster 39
- System limitation ............................. 40
1 Introduction

Trends are characterised by technological development and how they are adopted and used in society. Currently, there is a focus on the abstract concept of a cloud [21]. This project aims to solve a couple of main problems to make a standalone key-value datastore operate in the cloud. The challenge is to implement logic to distribute the datastore Edis, which is a fully protocol compatible standalone version of Redis 2.4 [1] implemented in Erlang, including the Open Telecom Platform (OTP) [8]. This project has been initiated by Inaka Networks, a US consulting group with development resources in Buenos Aires, Argentina. Their expertise is in bootstrapping; taking a startup idea and build it to a "minimum viable product", before the client team takes over. Inakas final goal of Edis is to develop a datastore for real time publishing platforms, second screen applications where a smartphone camera is used to photograph a screen and finally also for analytics. This project is limited to designing and implementing an infrastructure to make Edis distributed. In this report there is an approach to introduce a service into the cloud, to gain the benefits of low cost, software as a service, no maintenance, accessibility and large memory resources.

In order to define the context of the problem, this introduction contain brief information of what clouds are and the concept of distributed systems that they are built upon. Subsequently, hash tables are introduced to provide context by referring to a data structure that has similarities to key-value datastores, distributed hash tables could be seen as their precursor. The subsection of key-value datastores include the concept of NoSQL because this is how they are classified to distinguish them from traditional relational databases. Moving on, there is a subsection on logical time where it is presented why the notion of time is necessary but complicated in a distributed system without a centralised system clock in its’ nodes. Another useful concept to understand the state of a distributed system at any given time is “global snapshots”, which in the context of Edis is a depiction of what data is stored at a given time. Finally, this introduction contain background information on the developments of functional programming, to get some context of the paradigm that the implementation language Erlang belongs to.

After the introduction, there is the theory section, which has more substantial information about the problem domain so that the resulting system can be sufficiently motivated and represented. The next section is defining the problem and what goal can be achieved as a solution to the problem. The method section maps out what tools have been involved and a plan to implement the solution suggested in the previous section.
The testing section has a description of the introduced logic and the limitations to their performance. Finishing off this report, there is a discussion section with some considerations for the execution of this project, and finally a section on future directions that has potential to deliver value.

1.1 Clouds

Using cloud services can save time and money [21]. Setup and maintenance can be provided by a couple of clicks in a dashboard. A cloud service can be many computation related solutions, Amazon Web Services for example offer different types of computing and storage services.

Four main features of the cloud is that it has massive scale, on-demand access, data-intensive nature, programming paradigms for data intensive tasks such as NoSQL, MapReduce and MongoDB [20]. This enables the provision of Hardware as a Service (HaaS) solutions with access to hardware machines for a pay as you go service. At a higher abstraction there is the Infrastructure as a Service (IaaS), one can say it subsumes HaaS, that provides access to flexible computing and storage infrastructure by using virtualization, for example Amazon Web Services. Moving on to a higher abstraction level from physical hardware, we have Platform as a Service solutions (PaaS), where one can get access to flexible computing and storage infrastructure with the distinguishing factor that it is often tightly coupled with a software platform, for example Google’s AppEngine. Another service that is enabled by qualities of the cloud is the Software as a Service (SaaS) that provides access to software services, whenever they are required and to whatever capacity required, for example Google docs. Computation intensive tasks can be found in systems that have message passing interfaces, high performance computing, for example weather forecast modelling. For data-intensive computing, the focus shifts from computation to the data. Computing capacity will no longer be the bottleneck. Instead, I/O operations such as disk operations or network operations, become the most important resource metric [20].

1.2 Distributed systems

A cloud is a distributed system. The purpose of using the term “cloud”, is to provide a more specific context, because a distributed system is a very general term and can describe a vast amount of things. A cloud has hundreds to thousands of machines in a datacenter, server side, and
thousands to millions of machines accessing these services as clients. Servers communicating among each other constitute a cluster, which is also a distributed system. If only clients communicate with servers, that too is an example of a distributed system. Another distributed system is clients communicating with each other, such as a peer-to-peer system like BitTorrent [20]. The use of the name “cloud” is to denote the scale and nature of operations between client and server side actors, but it is a distributed system and the core concepts of distributed systems stay the same, the future might hold another name for distributed systems as the nature of how we use them evolve [20].

There are a lot of definitions out there for what makes up a distributed system. As a working definition in this report, a distributed system is a number of nodes, that are autonomous, asynchronous, failure prone and programmable. For the context of this report, the working definition is for a distributed system where the nodes communicate with each other for a common purpose through an unreliable communication medium, such as a cloud, wired or wireless network. The unreliable communication part of the definition is to motivate the aspiration of developing fault tolerant systems in error prone conditions [6]. The working definition of a distributed system has been inspired by Professor Indranil Gupta, who used the following working definition; “A distributed system is a collection of entities, each of which is autonomous, programmable, asynchronous and failure-prone, and which communicate through an unreliable communication medium” [20]. Nodes are logical entities that have computing capacity, for example an EC2 instance at Amazon Web Services. The asynchronous aspect of a node in a distributed system makes it distinguished from a parallel system with tightly coupled network and a synchronised clock [20]. The inability to rely on a synchronised clock in a distributed system makes their design more complex, having to cater for clock skew and clock drift. The unreliable communication makes the design of a distributed system having to cater for failures as a norm. The vast number of nodes also introduces design considerations for scalability and the concurrency of interactions as they access the same data.

1.3 Hash Tables

Hash tables are data structures in one process, in one node where you can insert, lookup and delete values referenced by a specific unique key. The key is associated through a hash function to an index in the hash table.
that contain a value [22]. If the hash table has a hash function using
perfect hashing, a one to one mapping between keys and values [22], it
can be seen as quite similar to a key-value datastore. This type of
operation has the complexity of $O(1)$ which means that the operation is
computed within constant time [22]. Once a hash table runs in more than
one process or peer nodes, it is referred to as a distributed hash table
(DHT) [20]. The distribution offers increased capacity and availability by
partitioning the set of keys across the cooperating peer nodes, this also
replicates the stored data [26]. The distribution introduces the problem of
routing, so that the key is stored on the responsible node and to propagate
this data to each following node along the DHT routing path [26]. Also
there is now a dependency for the distributed hash table that the nodes
are running, it has to be fault tolerant to failing nodes. Other
considerations in the performance of a distributed hash table is to
maintain the efficiency of operations. Another consideration is how to
efficiently communicate with other nodes, for instance by considering
locality, identifying nodes that are closer, easier to talk to, given the
context of how the nodes are connected. This type of context is based on
the network topology [20]. DHT’s can be seen as a precursor to a
decentralised distributed system that appears as a single system, as it
provides the ability to search for a value with the benefit of abstraction
when it comes to the distribution of the hash table [20]. Referring to
DHT’s could be useful when it comes to mapping out the considerations
and problems for distributing a key-value datastore like Edis.

1.4 Key-value datastores and NoSQL

This subsection has a brief background on the specific technology that
makes up Edis, key-value datastores, which is an example of a NoSQL
data model. A very direct way of storing data is to associate it with a
particular reference, a key, and the data stored as the value, or in other
words, a key-value datastore. Edis is an example of a NoSQL approach
of containing data, to distinguish it from the very conventional relational
database with a standard query language. A key-value datastore has a
reference and a piece of data, a key and a value. For example, an account
number as a key, and associated value containing data such as balance
and transactions. It is essentially a dictionary data structure where one
can execute operations such as insert, lookup and delete by specifying the
key, whereas the operation is subjected upon the associated value.

The purpose of introducing key-value datastores as opposed to
standard SQL databases is to cater for a new mode of operation that is
becoming more conventional within cloud computing. We have data that is large and unstructured, there is a lot of random reads and writes, there is a shifting focus to a lot of write operations as opposed to read operations [20]. Within the distributed architecture, that lacks strong relation between data points, there is no longer a strong requirement to perform join operations [20]. What is actually needed is speed, and a system that is resilient and is not subjected to a single point of failure [6]. On top of this, a great attribute of key-value datastores in the cloud is that there is a lower cost of operation, with few system administrators and incremental scalability. The most important focus in scalability is to not merely add more high performing expensive machines as the computational requirements increase, but to add machines that can compute operations more effectively together, than a single machine can compute on its own. Another aspect of the NoSQL key-value datastores is that there is no need to retrieve an entire table or database, simply find the key and the computation is done, it has no dependencies on other keys or values, as in both columns and rows. This leads to less overhead of coordination and I/O requests.

## 1.5 Logical Time

One approach to determine the order of events (message passing) across different nodes is to have synchronised clocks in all nodes and use time stamps. Given a distributed system where this type of synchronised clock is not available, then instead of using absolute time, one can use the ordering of events [9]. If event A happens before an event B, then timestamp(A) < timestamp(B). Causality is also used frequently in everyday conversations, “I hop on a bus after I tap my Oyster”, or “Your meal is served after placing an order”. However, if one event happens at the very same time as another event, causality cannot be used to distinguish them. This is referred to as concurrent events. Logical timestamps in a distributed system with concurrent events and no access to a synchronised clock all poses challenges that will have to be addressed.

## 1.6 Global Snapshots

Global snapshots represent a complete picture of a system [20]. Imagine a photograph of a summit at the United Nations, with all delegates from each country present and having a clear and formulated message prepared to represent. That could be seen as a global snapshot. However. If all delegates went back home, so that they are not in the same place
anymore, it would be difficult to take a picture of them all. Especially if
they are sending each other messages and forming new opinions,
developing their ideas, potentially they don't have a clear and formulated
message prepared to represent. This second scenario is the case for a
distributed system [20]. Putting this into the context of the cloud, there
could be a key-value datastore (a United Nations summit) that is running
on multiple EC2 instances (a delegates position) hosted by Amazon Web
Services, and each node of the key-value datastore (an individual
delegate) that runs on an EC2 instance contain a set of keys and values.
This set of keys and values at each node could in the previous example of
UN delegates be seen as the clear and formulated message that the
delegate is prepared to represent. The purpose of the global snapshot is to
get a picture of the global state of a distributed system, including the state
of each individual node, and the state of communication in the system.
This information of global snapshots can be useful to perceive what a
consistent state of a distributed version of Edis should behave like. More
precisely, it should to a user client appear as a global snapshot.

### 1.7 Developments in Functional Programming

Erlang is a functional language that was developed by Ericsson for a
specific purpose, telecom switches. In recent years there are companies
such as Erlang Solutions who have found an extended industrial use of
Erlang. They are operational in the following industries: Advertising &
New Media, Automotive, Financial Services, Gaming and the traditional
Telecom sector [5]. This section aims to provide some insight into the
developments in the use of functional languages, largely based on the
article "Functional thinking: Why functional programming is on the rise"
by Neal Ford [5].

The runtimes of functional languages have become capable of
handling more mechanical and structural work [5]. For example, on the
Java Virtual Machine platform there is Scala and Clojure [5], and on the
.NET platform there is F# [5]. This means that developers can be relieved
of trivial tasks. Functional programming is joining the mainstream and
according to Neal Ford [5] all mainstream languages will become more
functional. This trend can be identified in industry as several companies
start using functional programming and Erlang specifically, for example
Klarnas invoice system, Facebook chat, Riak by Basho Technologies
described in the section above and also within multiple companies in the
gaming industry.

Functional programming languages open up for replacing core
structures with higher-order abstractions. Ford argues [5] that in object-oriented languages, for example Java, reuse of code is common at the framework level, reuse often require so much effort that they are reserved for larger problems. Within functional languages, reuse is performed based on fundamental data structures such as lists and maps, which will provide customization through higher-order functions. This makes abstractions of code available at many different levels for functional languages, thus not confined to extensive ventures such as for abstractions within object-oriented programming. Java is not very flexible, as a consequence the problem is sometimes bent to fit the already rigid structure. Functional programming offers a more malleable approach where the language can be adapted to the problem instead of the other way around. For example the functional language Scala was designed to accommodate hosting internal DSLs, and all Lisps (including Clojure) have unparalleled flexibility for adapting the language to the problem.

To sum up, Ford states [5] that functional languages gain ground as runtimes gain power and languages gain more powerful abstractions, leaving the developer to focus more on the implications of results rather than how to generate them. Languages become customization machinery for highly optimized operations when utilizing abstractions such as higher-order functions.

2 Theory

This section will provide more in depth theory related to Edis and the problem that will be defined in the next section. To start with, there is some details on two systems that are closely related to Edis, namely, Redis and Riak. There is also a section on the chosen implementation language Erlang. Finally, there is a research section on consensus algorithms to investigate how it can be possible for one request at a given node, to give the same result, as this same request to any other node in the Edis cluster.

2.1 The context to motivate a Key-Value datastore

The motivation for using a key-value architecture for data storage is simply that all types of data to be stored do not require the organization and structure of a SQL database. For some systems it is enough with a
simple key and a corresponding value. On a website like amazon.com, the lists of best seller, shopping carts, customer preferences, session management, sales rank and product catalog only require a key-value structure, using a relational database would lead to inefficiencies and limit scale and availability [2]. The situation where reliability in the presence of massive requests is a priority, system downtime has significant financial consequences and affects customer trust [2]. In order to get an intuitive idea of the type of system that have this set of requirements, one can look at amazon.com, which is one of the largest e-commerce operations in the world [2]. The platform provides services for many web sites worldwide and is implemented on top of an infrastructure of tens of thousands of servers and network components in numerous data centers spread out globally. Operational requirements are strict and reliability is within the top priorities. These extreme performance requirements will utilize scalability, where it is possible to combine both memory capacity and processing power of numerous machines. Difficulties will also occur when operating at this level of precision [2]. These difficulties include failure of network components, although the relative amount might be small it is not allowed to affect the functionality of the system, and therefore it has to be treated as a normal characteristic of the system. This is one situation where a key-value in-memory data store is to prefer over a relational database. Any other case where access to data needs to be fast and where the organization of data is not relevant will benefit from this type of data storage. Dynamo [2] is a pioneer in this area and has inspired other systems such as Riak [3]. The most fundamental part of developing Edis from a standalone system into a system comparable to Riak is to implement a framework for supporting a distributed architecture, and that is what this project will aim to accomplish.

2.2 Existing products

This section introduces two existing products that can serve as alternatives to Edis but that also have inspired the creation of Edis. Namely, Redis and Riak.

2.2.1 Redis

Edis is protocol compatible with Redis version 2.4. Some of its main advantages listed below have inspired the initiative to develop Edis. Therefore, this section aims to introduce Redis, based on "The Little Redis Book" [1]. Redis is developed by VMware in ANSI C and is a fast in-memory, persistent key/value store.
Redis' main advantages are:

- Speed
- Expressivity of the command set
- Ease of Deployment

Which are traded for:

- "Weak" Persistence
  - Must write to a disk-backed DB (slower than writing to RAM)
  - Data must fit in-memory
- Lack of Expandability
- Single-master multi-node infrastructure

Redis solves specific problems but is still quite generic. It is single threaded which is how every command is guaranteed to be atomic. A thread is the smallest sequence of programmed instructions that can be organized independently by the schedule mechanism in the operating system for granting access to the central processing unit. To grasp the concept, a line of threads can be considered to be waiting for access to the CPU to get their tasks processed. The atomicity signifies that another client is unable to start executing an operation until the initial operation has terminated.

Redis exposes five different data structures, only one of these is a typical key-value structure. To understand Redis, it is essential to understand these five data structures, how they work, what methods they expose and what is possible to model with them. From a relational perspective, the Redis data structure could be compared to tables. Using tables as a data structure gives the possibility of storing almost anything. However, simplicity and speed can be gained if data is stored in a structure, which is tailored to that data itself, not including structures if they are not necessary. The philosophy is to use specific data structures for specific problems. If dealing with strings, lists, hashes or sets, they are stored as strings, lists, hashes and sets, respectively. This is rather intuitive as it is also the conventional approach when coding. Verifying the existence of a value does not have to be more complex than calling exists(key), or slower than a constant time lookup that stays the same regardless of how many items there are (complexity O(1)).

The five data structures of Redis all have at least a key and a value. This is such a fundamental aspect of the system that it is imperative to understand these concepts in order to understand the system. However, it is not too complicated but can they can easily be
confused. A key is how you identify data, it is a reference to access a specific piece of data. Values represent the actual data associated with its given key. From Redis perspective, keys are everything. Redis will have no awareness of the data value. Data querying is flexible and powerful these days but Redis will not support it. The strength with this approach is that values can be anything, Redis never needs to read or understand them. Redis will simply treat a value as a byte array. Byte arrays are used to represent data within an arbitrary binary file, as such, a Redis value could be any given digital object. A few very common examples of values are strings, integers or serialized objects such as JSON, XML or other formats. For code examples there are web based tutorials available at: http://try.redis.io/

That Redis is persistent, it preserves a previous version of itself as it has been changed. The working version will be stored in-memory, or namely in the RAM. And the previous version will be stored on disk. It is possible to specify how one wants this to be managed, according to how the system will be used. RAM is still the most expensive part of server hardware and is often quite limited in size. In order to address this issue, Inaka defined the goal of developing a version of Redis that has a multi-node architecture. This would allow the system to utilize RAM across several servers.

Query limitations, data structures and storing data in-memory results in very high-speed performance. Performance will depend on a lot of things, such as which commands are used and the type of data but it tends to be measured in tens of thousands, or hundreds of thousands operations per second. Another strength of Redis is that it has specialized data structures. This is important as it impact how a client interacts with Redis. A common approach for developers of an SQL background is to minimize the number of round trips to the database. That will not be a prioritized consideration when working with Redis as simple data structures will often require more requests to the Redis server compared to a SQL server. Promoting a higher rate of server requests can be perceived as quite unconventional but in reality it tends to be an insignificant cost compared to the raw performance that is gained.

2.2.2 Riak

Like Redis, Riak is also a key-value data store. And just like Edis, it is open source and written in Erlang. An interesting quality of Riak is that it scales predictably and easily. This is something that will be attempted to be implemented into Edis as well, combined with the speed of Redis. However, these two qualities are out of scope for this project.
and an ambition that will be attempted for a final version of Edis. As it is relevant for the future development of Edis, this section aims to briefly introduce Riak. Riak is based [3] on the CAP theorem [4] and Amazon's Dynamo paper [2]. These sources are also central when it comes to designing a solution for making Edis distributed. In Riak nodes can be added or removed from the system in a dynamic manner. This means that the data across the cluster will be redistributed according to the amount of available nodes. The distributed environment is key for Riak, with core operations such as reading and writing becoming faster when more nodes are added to the cluster [3]. In other words, there are substantial scalability gains.

Initial state of Edis

This section provides a system description of Edis in its initial state, prior to this project. The initial state was a single-node Erlang implementation of Redis, which was originally written in C. Essentially, prior to this project there has only been a translation of Redis from C to Erlang, and that is what we have as Edis.

2.2.3 System Structure and Client Interaction

In Figure 1 there is a representation of the structure of Edis. It is a tree structure based on supervisors, monitors and worker processes which is the conventional structure of applications in Erlang. The application is at the top of the hierarchy, which basically is a container for information that is required to deliver a particular application. The application requires a specific organization of files and directories. A basic requirement is the descriptor file with the extension .app that specifies all the resources that are to be utilized by the application. The application is followed by a supervisor that is supervising all other supervisors of respective process group. What defines a process group is that they are a set of processes that interact to provide a common role in the system. In Figure 1 below, one process group is denoted by one color. As an entry point to the system, one would start in the blue listener process group. Clients can communicate with Edis through a port. The port has to be within a range that can be specified in the configuration file. In Figure 1 the port range is set to \{6384, 6386\}, which makes connection to Edis possible through ports 6384, 6385 and 6386. As this is the initial standalone system, the ports all belong to the one and only node on which the system is running. Each port is on standby until a client connects to it.
The ports are supervised by the Listener Supervisor to ensure that they are restarted if execution is not as intended, i.e., the traditional role of a supervisor in Erlang. When a client - port connection has been established, the client is assigned a manager in Figure 1's pink process group. There are several managers and each manager can have several clients. When a client makes a request to execute a specific function, it is sent to the Edis Command Runner (ECR). The ECR will refer this request to the specified datastore in the yellow datastore process group or to the green publication process group, according to what type of function was requested. In the yellow datastore process group the request will finally end up in the datastore (circle marked DB in Figure 1). Edis is an in-memory datastore, so if the computer is shut down the entire dataset is lost. Therefore there is the option to connect a backend database where the Edis dataset gets replicated and safely persisted. Currently, the following backends are supported: Leveldb, Hanoidb or Process Dictionary. The green publication process group is a type of messaging system. It has two basic functions, one is to publish a message into a given channel and the other is to subscribe to a given channel, in order to read published messages. This is an existing feature of the system and is not within the confines of this project.

![Diagram showing the process groups and tasks involved in the system.](image)

**Figure 1**

### 2.2.4 The Role of the Backend Database

The backend database is a completely separate program that
performs basic storage functionality and it is abstracted from the end-user. The storage functionality provide value by creating persistent backups, whereas Edis is stored only in-memory and therefore is not persistent on its own. The backend is used as a library, and does not contain any kind of server or command-line interface. In this project it will be considered as a "black box", it is of no concern how it operates. Edis is protocol compatible with Redis version 2.4 [1]. In other words, the operations available to an end user through the Edis client are the same as for Redis 2.4. These end-user operations are however not the same as the ones that are sent to the backend database. Therefore, the only aspect that is considered when it comes to the backend is that the Edis end-user operations are translated into the limited set of operations that the backend can interpret for executing the basic manipulation of data in storage. This set of operations act as an interface between Edis and the backend. The translation process of operations is already implemented in Edis, prior to this project. However, it will be shown later that this translation process has to be extended for cluster synchronization to get a consistent set of data at all nodes, once the system has been distributed. The operations that are relevant for this project are: delete, get, put and write. The backend is "pluggable", which means that it can be exchanged. Currently it is possible to choose between a couple of common backend databases that are currently on the market, namely: "Leveldb" (open source, developed by Google) and "Hanoidb" which is open source and available on Github, developed by a group of contributors. It is also possible to use a Process Dictionary based backend.

2.3 Erlang/OTP

Edis is implemented in Erlang. The conventional structure of how to build programs in Erlang will provide strong support for availability and partition tolerance in the Edis cluster. In order to gain an understanding of this, the language itself and its structure is described in this section, based on Joe Armstrong’s dissertation "Making reliable distributed systems in the presence of software errors" [6], Learn You Some Erlang for Great Good [7] and the Erlang Manuals [8].

Erlang is a concurrent process-based language, where the processes are strongly isolated. Everything is a process and to create and destroy a process is a lightweight operation. This strong isolation means that interaction between processes is limited to message passing and not
for instance by shared memory. The programming model makes extensive use of fail-fast processes. Processes have a unique name (Process Identifier, PID), which is used to send messages.

Erlang as a language was originally designed for building telecom-switching systems. For this purpose, two central requirements are reliability and fault-tolerance. Telecom systems will be expected to be available at any given time, to operate "forever" including being operational while performing software maintenance. Furthermore, this type of system should have soft real-time behavior and should to some extent be able to handle software and hardware errors. One of the founders of Erlang, Joe Armstrong, coins [6] the term Concurrency Oriented Programming (as opposed to object oriented, functional or logic). Normally, concurrency support, if any, is something provided by the OS and not by a programming language.

From the language itself it is given that messages will be 100% delivered or they will not be delivered at all. Also, messages will be received in the order in which they were sent. In spirit of process isolation there are no pointers to data structures contained in processes within a message, only constants and process ID's. The strong process isolation limits the consequences for errors, for example by corrupt or failing processes to pass their errors on to other processes. In other words, process isolation limits error propagation. If a process start to execute in a way that was not intended, it is killed immediately by a separate supervisor process. If a system is truly to be fault tolerant, it has to be running on at least two physically separated machines. If one machine crashes, software (a supervisor process) running on another machine has to recover what the first machine was doing, error handling is thus not performed locally. The one major aspect of Erlangs message passing that has to be considered in the Edis project is that it is used for synchronization across the nodes in a cluster. This will give Edis causal ordering, which in a distributed system is when synchronization is based on passing a sync message that is delivered to a second process at some point later in time than when it was sent. The complicated part about this is that there are no guarantees for how long it will take or if it will actually be received at all. This delay has to be considered, as Edis will not be consistent during this gap if the messages contain information that a specific key shall adopt a new value.

Erlang programs can easily be taken from being run at a uni-processor to a set of processors. Testing for a distributed system is facilitated by the possibility of running, developing and testing several Erlang nodes on the same operating system and the same processor. In the distributed system all operations will work in the same way as they
work in a single-node test system. Except for timing, which will be synchronized for all nodes in a uni-processor test system but not if several processors are involved. Distributed computing is supported by two essential primitives, spawn(Fun,Node) which spawns a function Fun on the remote node Node and monitor(Node) which is used for monitoring the behavior of an entire node, as opposed to only a process. The monitor is a type of link between processes but the controlled object in this case is an entire node and not only a process. However, Erlang will treat it as a process.

A central part of programming in Erlang is programming patterns called behaviors that are provided from libraries in the Open Telecom Platform (OTP). The behaviors do not have anything to do with telecom specifically, they just happen to be useful within the field of telecom as was the intended purpose of the language. Behaviors are abstractions of common programming patterns that are generic and universal. They provide a generic framework for what is needed to implement, for example a server or a finite state machine. Utilizing these frameworks, programming efforts can be focused on designing a solution to the specific problem at hand. Focus can be set on programming what will be specific for the system at hand. It is naturally possible to program a server from scratch but these behaviors have been tailored to be as effective and error free as possible. A special case of behavior is the supervisor, a type of meta-behavior as it coordinates primitive behaviors into supervision hierarchies.

In summary, number crunching (for example image compression) could be implemented better in other languages (for instance C++) but if you want to coordinate a system through a distributed system then it should be done in Erlang. Concurrency and fault tolerance are principal qualities of Erlang. While Java might be able to run anywhere, Erlang could be said to run forever. Different languages are needed for different systems, and when it comes to distributing an in-memory data store like Edis, Erlang is a natural choice.

2.4 Research on Consensus Algorithms

When executing a request, for example to write a new key, on one node in the Edis cluster, the result of this function must be recognized and replicated in all the nodes across the cluster. Ultimately, the cluster will be acting like one single storage unit although it is distributed across multiple nodes, where each node comprise a storage unit. This leads to the issue of establishing consensus among the nodes in the cluster so that
a consistent set of data will be found at every node. To obtain consensus in Edis there were three algorithms chosen by Inaka for consideration in order to finally choose one for implementation. The scope of this research will be to make an outline that is comprehensive enough to make the evaluation of which algorithm is most suitable for Edis. The purpose is not to declare a complete specification of each algorithm. The three algorithms that will be considered for supplying eventual consistency are: Paxos, Vector Clocks and the Gossip Protocol. The scope of considered consensus algorithms has been narrowed down to these three as a result of a design decision at Inaka. The research on consensus algorithms will be divided into three parts. First, the consensus problem will be described in detail. Second, the three algorithms will be evaluated and compared to each other in regards to how capable they are to enforce consistency in Edis. Third, to select one of these algorithms to implement as a solution for the consensus problem in Edis.

2.4.1 Consensus

The intention of implementing a consensus algorithm in Edis is to improve the relatively weak consistency, but the effort to achieve this is limited by the resources available. This puts two dimensions to the concept of an effective algorithm, these being: technical and business. The chosen algorithm needs to consolidate a consensus in the system on a purely technical level, but it also needs to be a solution that can be implemented with the resources available to consider the business dimension. In other words, the implementation cannot be too complex, mainly due to the fact that this is the first attempt of making Edis distributed. Within further development of Edis, more complex consensus algorithms could potentially be considered.

The nature of the consistency issues that is faced by Edis is that certain failure modes can result in the system having conflicting versions of the same data. Because of node failures and network partitions, some operations might not be performed at every single node, leaving them outdated. Thus, there is a potential to end up with contrasting values for the same key. In order to make the system consistent with every node having the same value for a given key, a consensus has to be established among the nodes of which this value will be for a given key. This type of issue requires the system to be designed in a manner to acknowledge the possibility that keys can have different values. This provides the possibility to keep keys constantly available as they don't have to be put on hold as consistency is enforced. However, the system also has to enforce a consensus algorithm to improve consistency, although it is not instant, it can be designed in a way that eventually the system reaches
consistency. This is according to the CAP theorem [4] the tradeoff that has to be made. In our case, a priority for availability and partition tolerance has been made, compromising consistency but together with a consensus algorithm the system will have eventual consistency.

2.4.2 Paxos

In the Paxos algorithm [10], [11], [12], there is a set of processes in a network that shall agree on a single value in the presence of failures. All processes play the roles of proposer, acceptor and learner. Implementing this in Edis, one process could be represented by one node. The algorithm allows processes to crash and subsequently recover. This is the type of behavior that can also be found in Edis, where the node supervisor can restart a process (a node) if it crashes. Another two qualities of the algorithm are that the cluster may drop messages between processes and the processes have access to persistent storage that survive crashes. Relating this to Edis, dropping messages is something that is addressed effectively by Erlang so that the system as a whole and processes are prepared to handle dropped messages [6]. The persistent storage is something already provided if choosing to use one of the compatible backend databases. Basically, the algorithm requires the majority of the replicas to run sufficiently without crashing or failures until one proposed value has been chosen. Some replicas are granted ability to make such a proposal. With the qualities of the system described above, this agreement is guaranteed. This guarantee of consensus is the main quality that makes the Paxos algorithm very attractive.

In the article "Paxos Made Simple" [11], Leslie Lamport describes two phases of Paxos. The first phase begins with a proposer who selects a proposal number n. This proposer is one replica, or from the point of view of Edis, it is a node. Since several replicas can act as a proposer, a proposal number is used to distinguish the proposals and for the ability to derive which proposal is the most recent. Also, a proposer can make multiple proposals and abandon proposals in the middle of the protocol at any time. In order for two proposers not to choose the same proposal number, they can be allocated with a disjoint set of numbers to be used for proposal numbering. This proposal number is broadcasted out into the cluster in a message called the prepare request. The replicas that receive this prepare request are called acceptors. The acceptors will compare the number n in the received prepare request to verify that it has not received any previous prepare request with a higher number. If this verification is positive, the acceptor will respond to the request with a promise not to
accept any more proposals numbered less than \( n \) and which has been the highest numbered proposal so far, if any.

In the second phase the proposer will send out an accept request to all acceptors of the proposal number \( n \) with a value \( v \). The value \( v \) is the value that the proposer initially proposed, unless there was a value with a higher numbered proposal among the responses from the acceptors, in which case this number will be chosen. As a second part of this second phase, the acceptor will only accept a received accept request if it has not already responded to a prepare request with a number greater than \( n \).

In case two proposers keep issuing proposals with increasing numbers in a manner that no one ever gets an accepted accept request in phase 2, a distinguished proposer must step in. To guarantee that this distinguished proposer enables the algorithm to progress, it has to be selected as the only actor trying to issue proposals. It will succeed in making a proposal that is accepted if it can communicate successfully with a majority of acceptors and if the proposal number that is used is greater than any prior proposal number. Since working with majorities, liveness can be achieved by electing a single distinguished proposer, given that enough of the system (proposer, acceptors and communication) is working properly.

2.4.3 Gossip protocol

The Gossip protocol [13], [14], [15] require each node in the cluster to forward messages to a small set of "gossip partners", randomly chosen from the same cluster. As opposed to the Paxos algorithms, the Gossip protocol does not give any strong reliability guarantees. Instead, probabilistic guarantees of reaching consensus are obtained. Allavena, Demers and Hopcroft argue in their article "Correctness of a Gossip Based Membership Protocol" [13] that the gossip protocol delivers greater scalability and fault tolerance than other consensus algorithms that have these reliability guarantees. By randomly choosing the set of gossip partners, one can achieve a probabilistic guarantee to reach all nodes in the cluster at some point. However, because message loss is a fact of any cluster, the random selection might not represent the whole cluster. When designing an implementation, it is essential to minimize the risk of making a random selection of gossip partners from a subset of the cluster that is too local (small) for the protocol to maintain connectivity and performance guarantees. Another difficulty of randomly selecting gossip partners from the whole cluster could be to maintain current data at all nodes about which other nodes are in the cluster. However, on an Erlang based system like Edis this type of information is...
something that can be retrieved by the function call nodes(). Therefore, data for contacting other nodes is nothing that needs to be stored and maintained locally at every node, but something that is retrieved when it is to be used.

The way that the Gossip protocol select a small set of gossip partners makes it a multicast system, which is distinguished from broadcast because it deals only with sending messages to a group of nodes, not all nodes. When a node has received gossip (a message), it is said to have been infected by it. After a predefined period of time for the node itself, the node will pass on the gossip to a randomly selected set of other nodes in the system. This is “push” gossip, but there can also be periodic attempts by each node to “pull” gossip, essentially to inquire a random set of nodes if there are any new relevant messages for them. If both push and pull gossip is used, the algorithm is said to be a hybrid. Message passing happens asynchronously so that messages are not sent at the same time by different nodes. Ultimately, the gossip will replicate across the entire distributed system. This is also known as an epidemic protocol.

The random selection of receiving nodes results in the same messages being sent to the same nodes multiple times, which is a drawback to this approach. If choosing this algorithm, it has to be considered whether the complexity and overhead of sending duplicate messages causes a risk to the performance of the system, given the resources available to the system. Another risk to consider is if the random selection of gossip partners is not selected efficiently, the epidemic growth of the system could be too slow.

2.4.4 Vector clocks

Vector clocks [9], [3], is a technique to derive which value is the most recent. The consistency issue in Edis boils down to the same problem, any given key has to have the same value across the cluster, this value being the most recently set value. It is based on counting how many times that the value for a given key has been altered and by which node. The node that has the key with the most amount of alterations will have the key that refers to the current value to be used. However, if the cluster is partitioned into two separate parts for some time and the one and same key starts to get changed independently in the two isolated cluster partitions, whereby the cluster connection is restored, it will be useless to just verify how many times the value has been changed. The value for a given key can have been changed 100 times in cluster partition A, and after all of this it is changed only once in cluster partition 22 Joachim Nilsson
B. The value to be used is the most recent, which could be the value in cluster partition B although it only changed once, as opposed to 100 times in partition A. In other words, it is not enough to count the amount of times that a value is changed. To address this, the vector clock is also registering what node changed the value by having an individual counter registering which node that change the value. This way it is possible to verify if the vector clock for one given key is a subset of the vector clock stored for the same key at another node. If one vector clock is a subset of the other, it descends from the other, and is therefore an older version of that key.

The following example illustrates how vector clocks determine descendence between two values. Brujo, Marcos and Marcelo are going to decide on a day next week to play a ping-pong game. In Edis, a node can represent an individual actor to set a proposal date and to come to a final agreement. In this example, an actor would be represented by Brujo, Marcos, or Marcelo. The conflict to be resolved in Edis would be what value is the correct one for a given key. Correct being the most recently set value for a given key. In this example the key can be chosen as PingPongDate and the value could be one and only one out of the following set {Monday, Tuesday, Wednesday, Thursday, Friday, Saturday, Sunday}.

Marcos starts with suggesting Tuesday. In the vector clock, he tags his name and increases a counter (from 0) to register that he made a suggestion:

Suggestion 1
PingPongDate = Tuesday
vclock = Marcos:1

Marcelo receives the suggestion but he suggests Wednesday and adds his tag to register that he made a new suggestion. He leaves the tag for Marcos suggestion so that when another actor reads the vector clock, it is possible to derive that Marcos suggestion was taken into consideration when this suggestion was made. Which suggestions were made initially in suggestion 1 by Marcos is however not known, it has been discarded. This makes suggestion 2 the following:

Suggestion 2
PingPongDate = Wednesday
vclock = Marcos:1, Marcelo:1
Brujo receives suggestion 2 from Marcelo and decides to make a suggestion that suits him better. As is customary, he adds a tag to represent that he made a suggestion, with a suggestion from Marcos and another from Marcelo taken into account.

Suggestion 3
PingPongDate = Thursday
vclock = Marcos:1, Marcelo: 1, Brujo:1

Marcos collects all vector clocks, both Brujo’s and Marcelo’s. He can tell that they are not the same, which means there might be disagreement on which date to play ping-pong. Or from the perspective of Edis, there might be an inconsistency among the values in the cluster for the key PingPongDate. It is now possible to derive the descendancy of the vector clocks to verify which vector clock represents the correct value to be used. They can be descendant either if they are equal, or if one vector clock is an earlier version of the other. The correct value being the most recently set value. Marcos can tell that Marcelo made suggestion 2 taking his own suggestion 1 into account. This means that suggestion 1 is a descendant of suggestion 2, which in turn means that between these two suggestions, suggestion 2 should be used. In the same way, he can tell that both suggestion 1 and 2 are descendants of suggestion 3. One could suggest that it is merely necessary to register how many times the value has been changed and not who made the change. Consider then a suggestion where the vector clock has registered 10 tags only for Marcelo and this is received by Brujo who has the vector clock in suggestion 3. This would suggest that the 10 tag suggestion is more advanced since suggestion 3 only has 3 tags. But by registering the names it is possible to derive that this 10 tag suggestion has not been taking the tags from suggestion 3 into consideration and therefore it is not known if that value is the result of a chain of suggestions. In other words, it is not possible tell where the value comes from and the vector clocks are in conflict.

Conflicting vector clocks require another mechanism to be enforced in order to reach reconciliation on which value to use. One mechanism is to use timestamps, inspired by Leslie Lamport [9]. A timestamp is registered on a key for every time its' value is changed. When a node is confronted with this type of conflict, it is required to compare the timestamp of the suggested value to the timestamp of the value that it currently has registered for that same specific key. But due to clock drift and that it is impossible to get reliable communication one cannot synchronize the clocks in a distributed system without a central clock that all processes have a secure communication to. Such a
centralized clock is not available in Edis. Unreliable communication is
due to risks of network errors and message loss, it is not possible to
calculate how long it takes for a message to leave the sender node and
arrive at the recipient node, or even if it will at all. Clock drift and
unreliable communication make timestamps unsuitable in situations
when two timestamps that are very close together are compared, they
could be just microseconds away from each other. As mentioned before,
there will be a compromise on either consistency, availability or partition
tolerance according to the CAP theorem [4], consistency was chosen to
be compromised. This last effort with timestamps will increase
consistency if vector clocks fail to establish descendance and if the
difference between TimestampNode1 and TimestampNode2 is greater
than the difference between their respective clocks, ClockNode1, and
ClockNode2. So there is a further extension of consistency that can be
guaranteed in a cluster of N nodes if the following holds:

\[ |\text{TimestampNode2} - \text{TimestampNode1}| > |\text{ClockNode2} - \text{ClockNode1}| \]

A future direction for Edis could be to investigate if this holds within
reasonable bounds or if other measures should be employed to increase
the consistency of Edis.

### 3 Problem and Goal

Currently, in Edis’ standalone state, data is only stored once in one node.
There is only one replica of the datastore. This constitutes a risk as the
replica can be corrupted, lost due to hardware errors or human mistakes.
Therefore, this project aims at achieving redundancy, several copies of
the data within the same system. The data could simply be copied, but
this is of no use in case the whole machine breaks down. As a standalone
system, Edis cannot either scale whenever there is a need for it to do so.
Dynamic scaling is not possible. If it is possible to provision more nodes
on demand, it will provide a tailored amount of capacity at a given point
in time. This has led Inaka to take the decision to make Edis distributed,
so that it is operating over several independent nodes. Designing such a
distributed system leads to the following three problems:

1. **Availability:** The system shall be available even if one node fails.
   This means that each node need to be independent and self
   sufficient so that it will not be affected by the failure of another
   node.
2. Consistency: Data has to be the same on separate nodes.
3. Partition Tolerance: Network failures and communication issues shall not affect the system.

In the following section there will be a closer look at these three qualities of a distributed system by doing a CAP classification of Edis. This is conducted to determine the nature and requirements of the system. Further on, there is also a closer look at what is required specifically for Edis in order to make such an implementation. The implementation will be considered in two parts:

1. A framework for multi-node structure
2. Executing operations and synchronizing the cluster to be consistent

3.1 Consistency, Availability and Partition Tolerance

For distributed systems, it is essential to plan for failure of individual nodes and message loss so that the overall system will continue to operate and be available even in the presence of errors. This problem is addressed in the CAP theorem (Brewer's theorem) [4] on the correlation of consistency, availability and partition tolerance (CAP). Brewer argues that the hard part of distributed systems is persisting state, whereas distributing computations is relatively easy. Primarily, he introduces three properties that can be identified in all distributed system, namely consistency, availability and partition tolerance. Brewer's theorem states, "You can have at most two of these properties for any shared-data system." There will in other words be a trade off, which of these three desirable characteristics will be compromised.

Consistency

A consistent set of data on all nodes throughout the cluster. Operations have the atomic characteristic that changes are performed universally and simultaneously in all nodes.

Availability

The system has to answer every request, also in case of failures.

Partition Tolerance

An arbitrary split between nodes shall not infringe consistency
This leaves us with three possible CAP combinations. There is the combination of consistency and availability which is also known as high-availability consistency as it provides a consistent and available system. However, in the case of a partition, it may become inconsistent. Recovering from partitioning can be done with a consensus protocol. Until this consensus has been established, there will be a period when service requests will be denied. This would make the system into a CP combination at the sacrifice of availability.

A second contender is the combination of consistency and partition tolerance. It is a combination that leads to a system with strong consistency, also known as enforced consistency. Nodes can be partitioned from the system, therefore they will not be accessible, availability has been compromised.

Finally we have the third possible combination, availability and partition tolerance. This situation is when the system will continue to be available even in case of a network failure of some nodes. In effect, there is a possibility of having an individual partition that temporarily, while it is partitioned, holds an inconsistent state, consistency has been compromised.

To classify what CAP combination a multi-node Edis will require more implementation details. Thus, the classification is left to the problem and goal section. Below follows a specification of the implementation considerations that are required.

### 3.2 A framework for multi-node structure

Allowing for keys to be stored at several nodes require fundamental structural changes to the system. Several nodes have to be started and they need to be able to recognize each other and be able to communicate with each other. Also, the management of the nodes needs to be supervised for the occurrence of errors. In Erlang this means that a supervisor has to have the capability to kill entire nodes that do not execute as intended. Killing a node is followed by a restart so that the cluster does not lose a node. There also has to be a monitor to notify the cluster if a node joins the cluster or goes down and leaves the cluster.
3.3 Executing operations and synchronizing the cluster

The system does not only have to execute a client operation request in the datastore, but also it has to pass the request to the other nodes in the cluster. Each individual node does in effect require the ability to receive client requests not only from a client herself, but also from other nodes in the system. This will lead to a verification issue, to distinguish if a value proposed by another node is to be accepted or if the locally stored (if any) should be kept.

3.4 Classifying CAP combination for Edis

In 3.3 above, it is stated that all nodes shall be the subject of a common supervisor and have a monitor. The supervisor behavior in Erlang works as such that a faulty node will be killed and restarted, this gives us availability. The monitor is a type of link between the nodes that makes them able to reconnect if a node has been restarted or if a temporary network failure has been recovered. This means that the monitor gives Edis partition tolerance. Edis therefore compromises consistency of data in the cluster. Compromised consistency requires certain mechanisms to have it improved. Section 2.4, Research on Consensus Algorithms, investigates and evaluates different mechanisms to increase consistency over an Edis cluster. Finally, this section compares the algorithms and chooses one to be implemented.

4 Method

This section presents the tools that were used to design the system and also it provides an implementation plan for the system, illustrating the setup of different executive entities in the system.

4.1 Tools

These are the tools that were used to realize the project.

- github.com: a web-based version control system used as a open source code repository. It is possible to create a profile for social networking functionality, make projects and to contribute with solutions to existing projects. The
platform allows to work locally on some code before committing it to the official project.

- Sublime Text 2: the code editor that was used [23].
- Erlang Shell: an emulator that can be run in a terminal to run Erlang code, mainly for testing ideas [8].
- Lager: a logging framework for Erlang/OTP developed by Basho [24].
- Erlang manuals [8].
- Rebar: a build tool for Erlang/OTP that facilitates compiling and testing [25].

4.2 Implementation Plan

The implementation can be divided into two parts. First there has to be a framework to coordinate the multi-node structure. The structure will have to address situations such as node failure (no response or unexpected behavior) and enabling nodes to join and leave the cluster, detailed in section 4.2.1. The second major issue that has to be addressed is how to propagate changes in one node across to the others and to maintain consistency across the cluster, detailed in section 4.2.2.

4.2.1 Design of a framework for multi-node structure

The purpose of this framework is to obtain an ability to run several nodes in a manner where they can communicate, be aware of the status of other nodes and there also has to be an ability to kill and restart nodes in case a node does not execute as expected. As a node is restarted it has to automatically join the cluster again. The framework for multi node support will be implemented by adding a new process group to Figure 1. The new process group will be a monitor and a supervisor. This means that complete nodes would be supervised and monitored in the exact same way as a common process.

4.2.2 Executing operations and synchronizing the cluster

An Edis client will be connected to one node only. If a client decides to execute a function, this will be done in the corresponding node. Therefore, the result of this execution has to reach the other nodes in the cluster. The purpose here is to make sure that data across the cluster is consistent and that there is a communication system that can keep all nodes in the system updated. This will be done by introducing another layer of logic for each interaction with the datastore. In this layer, consistency mechanism will be implemented together with broadcasts to other nodes to keep the cluster updated about changes that have been
executed locally in a node. To begin with, the consistency mechanism has to be decided by researching some options in section 4 to evaluate which algorithm would be suitable for Edis.

5 Resulting System

The resulting system of this project can be broken down into two parts. The first one is a framework to support a multi-node structure, this is how this project makes Edis a distributed system. As distribution is one of the key features of Erlang [6], there is a lot of support available to design this type of implementation through generic behaviors and built in functions. The second part is logic that coordinate operation requests and synchronize the cluster. This part of the system executes the ultimate operation requests to the datastore, it deals with how to propagate values to the rest of the cluster and how to derive which value to be used in order to obtain a consistent set of data across the cluster.

5.1.1 Selection of consensus algorithm

Comparing these three consensus algorithms, Paxos is the most complex algorithm but it is attractive as Lamport proved [18] that consensus will be achieved. At Google, the purpose of the project "Paxos Made Live" [10] was to implement the Paxos algorithm to achieve consensus in a key-value data store. Gaps were identified between the description of the Paxos algorithm and the needs of a real world system. Lamport's proof of reaching consensus was established only in theory. The protocol extensions that are required for an actual implementation will result in a system based on an unproven protocol [10]. Although this algorithm seems suitable for Edis, to resolve these implementation issues require an extent of resources that are not available to the Edis project.

One of the key features of the Gossip protocol is that there is a low frequency of interaction. However, in Erlang, interaction is based on sending messages and this is a fundamental and inherently low cost operation in Erlang, as established in section 4.3. Because of this, minimizing interaction between processes is not a measure that would result in significant decrease of complexity in Edis. At this brief level of consideration an implementation does not have any direct problems, but since the key feature of the algorithm is not a priority, the Gossip Protocol will not be implemented.

Vector clocks have been successfully implemented in the distributed key-value data store "Dynamo" [2] by Amazon and in the distributed general-purpose key-value data store Riak [3] by Basho.
Technologies. This shows that it is a solution that has been proven to be effective in similar systems which suggests that it could be a good solution for Edis as well. Riak is open source and basic functions for handling vector clocks can be utilized in the spirit of the open source community that Edis and Inaka is part of. Naturally the Apache license version 2.0 [16] of Riak will be respected. The implementation effort "Executing operations and synchronizing the cluster" could use Riaks vector clock as they are easy to implement and have been promoted by utilization in similar systems. Therefore, the algorithm for reaching consensus across the nodes in distributed Edis shall be vector clocks.

5.2 The framework supporting multiple nodes

Erlang provide potent support for connecting nodes. Each machine will have an application called the EPMD (Erlang Port Mapper Daemon), which acts as a name server that registers nodes. From the EPMD a node can set up a connection to another node. If this node is part of a group of nodes that already are connected, all nodes will be connected. The connection means that the nodes will be monitoring each other. The monitoring means that the nodes will know if the connection is lost or if a node disappears. In Erlang a node is known by its Process Identification (PID) and only the PID is required to send a message to a node.

Connecting two nodes is done by setting two parameters when starting them from a terminal. First, one sets a name together with the ip of the physical machine on which the node is running, separated by an @ sign, for example:

bar@192.168.1.124

In this case, the name of the node is "bar" and the physical machine on which the node is running has IP 192.168.1.124. This IP address also has to be specified in a file under the Edis directory called ".hosts.erlang". A cookie also has to be specified. This cookie will tell the system that the node that is being started shall be connected with all other nodes that were started with that same cookie specified. In other words, all nodes in one cluster will have the same cookie specified.

Apart from these parameters specified at startup, the general architecture of Edis has also been changed in order to support multiple nodes. This is represented by the red process group in Figure 2. This new architecture provides the system with a new set of supervisor and monitor. The supervisor is the actor that coordinates running multiple nodes simultaneously that from the point of view of a client is one single system. This is done by making sure that they execute as intended, if not,
they are killed and restarted. This is the generic role of the behavior "supervisor", as it is implemented in OTP [6]. This supervision for entire nodes is the same as how individual processes are supervised. Referring to the CAP theorem, the node supervisor provides the system with availability as it takes on the responsibility of keeping every node constantly accessible.

The monitor is implemented as a generic server behavior (comes from the OTP platform) and starts monitoring all nodes by the network kernel, which is an Erlang system process that is required for distributed Erlang to work. From a CAP perspective, the monitor will provide partition tolerance so that if the network for some reason is divided in two parts, these two can operate on their own as if nothing happened. When a node is connected to this monitor there will be a message prompted to the terminal saying "[node name] is UP!" or if a node is disconnected, "[node name] is DOWN!".

![Multi Node Edis](image)

**Figure 2**

### 5.3 Executing operations and synchronizing the cluster

This subsection describes the implementation of how Edis coordinate operation requests and synchronize the cluster. When a value is about to be stored in the datastore, it is not enough to simply store it in the local
datastore. As Edis has become distributed, this value has to be stored in the respective datastore at every node. The value will be stored locally and then broadcasted out to the rest of the cluster. This constitutes executing the operation requests. Synchronizing the cluster also comes into this broadcasting operation. When a broadcasted message arrives at a node, this message has to be examined. The purpose of the examination is to derive whether the suggested value is to replace the locally stored value or not. The suggested value could possibly be a value that is not the most recently chosen value to be stored. Therefore, vector clocks and timestamps are used to derive whether the suggested value or the locally stored value is to be used. By determining which value shall be used each time a value is suggested, the system obtains eventual consistency. In other words, values across the cluster become synchronized.

5.3.1 Operation requests: The multi-node interface layer

Whenever a write, put, delete or update is performed, the system has to store the resulting value in the local datastore, increase vector clocks and set new timestamps where suitable. However, this value also has to be sent out to all other nodes in the cluster. This is done by a type of broadcasting function, which is an Erlang BIF (Built In Function), called abcast. The abcasted message will be received by all nodes in the cluster through a generic server from the OTP platform. The message is directed to the handle cast callback function that will match the type of request in the message, the types are: write, put, delete or update. The source code is located in the module edis_db.erl [17], the operations are implemented as the following functions:

\[
\text{db_write(LocalActions, IncomingState), figure 3}
\]
db_write(LocalActions, AbcastActions, IncomingState), figure 4

```
db_write(LocalActions, AbcastActions, IncomingState) ->
(IncomingState#state.backend_mod):write(IncomingState#state.backend_ref, LocalActions),
abcast = gen_server:abcast(nodes(), process(IncomingState#state.index), [db_write, AbcastActions]),
ok.
```

db_put(Destination, EdisItem, State), figure 5

```
db_put(Destination, EdisItem, State) ->
IncrementedVClock = EdisItem#edis_item{
vclock = edis_vclock:increment(node(), EdisItem#edis_item.vclock),
timestamp = edis_vclock:timestamp(),
}
(State#state.backend_mod):put(
    State#state.backend_ref, Destination,
    IncrementedVClock),
abcast = gen_server:abcast(nodes(), process(State#state.index), [db_put, Destination, Incremented, IncrementedVClock]),
ok.
```

db_delete(Destination, State), figure 6

```
db_delete(Destination, State) ->
EdisItem = (State#state.backend_mod):get(State#state.backend_ref, Destination),
db_delete(Destination, EdisItem, State).
```

db_delete(Destination, EdisItem, State), figure 7

```
db_delete(Destination, EdisItem, State) ->
case EdisItem of
    #edis_item{} ->
        NewEdisItem = EdisItem#edis_item{
            vclock = edis_vclock:increment(node(), EdisItem#edis_item.vclock),
            timestamp = edis_vclock:timestamp(),
        }
        (State#state.backend_mod):delete(State#state.backend_ref, Destination),
        abcast = gen_server:abcast(nodes(), process(State#state.index), [db_delete, Destination, NewEdisItem]);
    not_found ->
        %% Assumption: All nodes have agreed once that the item should be deleted.
        not_found;
        {error, Reason} ->
            lager:error("~p-n", [Reason])
        end,
    ok.
```
As a process receives a message that is the result of an abcast, the message will be directed by the generic server behavior to a corresponding callback function, the type of callback function in this case is a handle cast. It is basically a process being consulted to execute a function in case there is a matching one available. All of the write put delete and update operations have corresponding callback functions. In these functions there will be an incoming value that is suggested by another node in the cluster. This value is stored in an Erlang data structure called a record, this instance of a record implementation in Edis is named #edis_item{} and contain fields that have information on what the key is called, what the value is, the vector clock and the timestamp, along with a few other fields that are not relevant for this implementation. As there is no possibility to know for how long this message has taken to arrive or if the value was old to begin with, the receiving node needs to check the descendance of the vector clocks as described in section 5.2.2. According to the descendance, a value is chosen and stored in the datastore.

5.3.2 Cluster synchronization: Implementation of Vector Clocks & Timestamps

The vector clock implementation edis_vclock.erl [19] is used to increase the consistency of values across the nodes in the Edis cluster. Every node stores a complete set of keys with respective values. This gives us redundancy, if one node fails the value can be obtained from another node. Having a complete replica at every single node is going to be addressed in the future, it provides safety if there is a node failure, but it is excessive to store at every single node if the system operates with a large number of nodes, as in hundreds. It is essential to employ a measure in which there will be a guarantee to have enough copies of data so that data is available if there is a node failure, but not too many replicas so that memory is occupied for unnecessary replicas instead of the memory being available to the system. The consistency ambition is that for every key K at every node in the cluster, to have the same respective value. If vector clocks fail to derive which value should be used, timestamps are compared and the most recent value is chosen. According to the CAP
Theorem [4] it is not possible to reach consistency, availability and partition tolerance at once, only a combination of two of them, compromising the third. As Edis has availability and partition tolerance already, there will be an attempt to improve consistency with vector clocks.

Functionality for specifying vector clocks and basic operations for them has been taken from the key-value data store Riak that is also implemented in Erlang, by Basho Technologies. The implementation of this existing set of functions into Edis has been based on how to apply them in the specific Edis processes in order to achieve greater consistency. In Erlang there is a data structure called "record" where data can be stored by specifying the name of a field and a corresponding value of that specific field. Edis uses a record named edis_item that contain, among other things, fields for the actual key and another for the value. Consistency is to be achieved among the values across the nodes, therefore the vector clocks has to be implemented together with the values. This leads to the natural choice of extending the existing edis_item record to also contain fields for a vector clock and for a timestamp.

The resulting implementation of vector clocks can be found in the file edis_vclock.erl [17]. In order to illustrate how vector clocks interact in the system, the example from section 4.4.4 is extended. Brujo, Marcos, Martina and Marcelo are going to decide on a day next week to play a ping-pong game. To get some intuition on how this would be represented in Edis; every actor to set a proposal date and to come to a final agreement on a date is a node, and in this example the actor is represented by Brujo, Marcos, Martina or Marcelo. The conflict to be resolved in Edis would be what value is the correct one for a given key. Correct being the most recently set value for a given key. In this example the key is PingPongDate and the value could be one out of the following set of atoms (according to Erlang convention, the data type "atom" is written with a lowercase letter):

\{monday, tuesday, wednesday, thursday, friday, saturday, sunday\}

Marcos suggests Tuesday as he has classes on Monday. Later, Brujo and Marcelo decide Wednesday. Brujo has also been talking to Martina by the coffee machine at some point and they decide on Friday. When Marcos pings everyone to verify if they agree on his Tuesday suggestion, he gets mixed responses. Martina claims to have settled Friday with Brujo and Marcelo claims to have settled Wednesday with Brujo. Since no suggestion was the same as Marcos initial suggestion,
Tuesday, it means that it has been discarded by the others as they could not make it. Remaining are Wednesday or Friday but Brujo cannot be reached so none of the others can determine the order that these communications happened and which is the correct choice. This situation can now be resolved with vector clocks, by tagging the value with a vector clock which has a counter that is augmented with one for a respective person whenever that same person change the suggested date. Breaking down the situation above, starting with Marcos initial suggestion and him tagging that proposal with a vector clock indicating that this is the first version:

For the incoming EdisItem, the record edis_item is specified with the following fields:

**Suggestion 1**

<table>
<thead>
<tr>
<th>Field</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>key</td>
<td>PingPongDate</td>
</tr>
<tr>
<td>value</td>
<td>tuesday</td>
</tr>
<tr>
<td>vclock</td>
<td>[{Marcos, 1}]</td>
</tr>
<tr>
<td>timestamp</td>
<td>edis_vclock:timestamp()</td>
</tr>
</tbody>
</table>

Brujo and Marcelo start talking and Marcelo suggests Wednesday:

**Suggestion 2**

<table>
<thead>
<tr>
<th>Field</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>key</td>
<td>PingPongDate</td>
</tr>
<tr>
<td>value</td>
<td>wednesday</td>
</tr>
<tr>
<td>vclock</td>
<td>[{Marcos, 1}, {Marcelo, 1}]</td>
</tr>
<tr>
<td>timestamp</td>
<td>edis_vclock:timestamp()</td>
</tr>
</tbody>
</table>

Marcelo has left Marcos tag from suggestion 1 alone which allows us to draw the conclusion that Marcelo made the suggestion of Wednesday with an existing initial suggestion taken into consideration. Now Brujo also replies, confirming the Wednesday suggestion that he had been discussing together with Marcelo:

**Suggestion 3**

<table>
<thead>
<tr>
<th>Field</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>key</td>
<td>PingPongDate</td>
</tr>
<tr>
<td>value</td>
<td>wednesday</td>
</tr>
<tr>
<td>vclock</td>
<td>[{Marcos, 1}, {Marcelo, 1}, {Brujo, 1}]</td>
</tr>
<tr>
<td>timestamp</td>
<td>edis_vclock:timestamp()</td>
</tr>
</tbody>
</table>

In the step below Martina suggests friday to the first initial suggestion that Marcos made, because she has not received the most recent suggestion 3 made by Marcelo and Brujo. This means that the vclock below in suggestion 4 is in conflict with the vector clock above in
Brujo has a look at the two vector clocks in suggestion 3 and 4 and can tell that they are in conflict. If one vector clock is not the descendant of another, they are in conflict. They can be descendant either if they are equal, or if one vector clock is an earlier version of the other. For example suggestion 2 is a descendant of suggestion 3 as it is a predecessor, an earlier version. Fortunately Brujo is on top of everything and remembers that the most current value is Friday since the last person he decided a date with was Martina and thus the value shall be Friday. This recognition of what was the most recently set value is determined by Edis by comparing the timestamps. The following step is to make sure that both vector clocks in suggestion 3 and 4 that were part of the conflict should now be descendants of this new suggestion, which is the result of resolving these two conflicting vector clocks. This is done by merging their vector clocks. The merge will leave any overlapping tags as they are and tags that one vector clock has and not the other will be added to the result of the merged vector clock. This merge could be considered as the two vector clocks in conflict to become subsets to the smallest possible vector clock that can be constructed. On top of this, a new tag has to be added for Brujo as this signifies him making a new suggestion. The following is obtained:

**Suggestion 5**

```
EdisItem#edis_item {key = PingPongDate,
value = wednesday,
vclock = [{Marcos, 1}, {Marcelo, 1}, {Brujo, 2}, {Martina, 1}],
timestamp = edis_vclock:timestamp()}
```

Marcos now asks around for suggestions from Martina and Marcelo. What vector clock that a specific person/node has stored will be depending on the communication efficiency of the system. Martina has at this point already received suggestion 5 back from Brujo, and Marcelo currently has suggestion 3. Marcelo can see from this that suggestion 3 is a descendant of suggestion 5 and will thereby override suggestion 3. The correct value has been found, in suggestion 5: Wednesday.
6 Testing

The testing is aimed at achieving confidence that the cluster is coordinated and maintained with its’ Erlang OTP framework supporting multiple nodes. The second goal for testing, is that when executing operations against the cluster, which is done through a client to a specific node, this operation is also reflected in all other nodes of the system. This boils down to evaluating the consistency of the system. Testing is therefore divided into the following two parts:

✦ The framework supporting multiple nodes
✦ Executing operations and synchronizing the cluster

Testing a distributed system in Erlang is facilitated by the possibility of running, developing and testing several Erlang nodes on the same operating system and the same processor [6]. Therefore, testing Edis has been performed by running two nodes on the same physical machine. To finish off the testing there is an empirical evaluation of the system.

6.1 The framework supporting multiple nodes

When opening up two nodes to be public and specifying the same cookie, they can recognize each other. This means that Edis is running on two nodes, in other words, the deployed multi-node structure is operational. In case a node would fail for whatever reason, it will be killed and restarted by the node supervisor, this functionality is provided by the generic supervisor behavior [6]. Development in Erlang is made very powerful when using the OTP platform by making generic behaviors like this available. This very simple functional test validates the flow of message passing, that the logic implemented in the db_* functions in figure 3 to 8 is applied.

6.2 Executing operations and synchronizing the cluster

Setting a key in one node will be recognized by the other node. Changing the value for a given key in one node will also change the value of that same key in the other node. As this is repeated, the system manages to determine the correct value to be used. This means that the consistency measures that have been implemented are working.

If one node is shutdown and reconnected, it will get synchronized after the first request that is made. The fact that the synchronization does
not occur until a first request has been made is the manifestation of compromising consistency that is directly visible when interacting with the system. When this first request has been made, the data across the cluster is synchronized, the system has eventual consistency. The EUnit tests specified in edis_vclock.erl [19] will only test the module edis_vclock itself, and not give any indication of how the consistency performs overall for the system.

6.3 System limitation

Given the boundaries derived in section 4.5.4, there could be further tests to make empirical evaluations based on performance under load. We have that if the vector clocks fail to establish descendance, consistency still holds when:

\[ |\text{TimestampNode2} - \text{TimestampNode1}| > |\text{ClockNode2} - \text{ClockNode1}| \]

The Achilles heel for the performance of Edis’ consistency is clock drift. Given a situation where the timestamps in the nodes are just microseconds away from each other, the possibility of the difference in the clock at each node grows greater, to the point where the condition above does not hold. This can be the case if there is an extremely high intensity of operations, such as purely automated requests to the cluster of an application like Twitter that has a very large number of active users at any given time. But it has to be seen in relation to the precision of the clock at each node, so that the scope and probability of clock drift can be determined. If the clock drift grows to be of larger proportions than the intensity of executed operations with their associated timestamps, the system will fail to derive descendance. In a uni-processor test system where all nodes have access to the same clock, it is not possible to investigate the limitations of clock drift, because the one and only clock simply does not drift away from itself. A very specific usage of the system where the degree of clock drift can be managed, and it is known how many operations is made per second, could be used to investigate the sensitivity to clock drift. However, this is a risk that is only exposed when the vector clocks fail to establish descendance, so the system might show to eventually recuperate after this type of failure.
7 Discussion

What has been accomplished in this project is developing a standalone system into an operational distributed system. This has meant structural changes in the architecture of the system to support the management of multiple nodes, the systems’ constant availability and support to deal with network partitions. The structural changes are largely a result of utilizing the OTP platform with generic behaviors and built in functions from the network kernel. Using these existing tools is a powerful way to set up distribution.

Consistency has been chosen as the one CAP [4] quality to be compromised. A node that is connected to Edis will not be updated with the existing keys until after a first request. This could be changed by making an automatic update for a node every time it is connected or restarted in the system. This could be implemented in the node supervisor. The consistency has been reinforced with vector clocks and timestamps so that each individual node can determine if values suggested by other nodes, or the locally stored value shall be used.

The distribution meant a more complex interaction between client side operation requests and the datastore as values require a verification to determine whether they are old and to be replaced with a new one. Also when determining what values to be used and stored in the datastore, the system has to broadcast values out to the rest of the cluster. These broadcast messages has to be received and handled by the other nodes which compare these suggested values with the locally stored values by verifying descendance by vector clocks. All of this has been implemented successfully, although further testing would help to formally prove the system. This project leaves Edis at a distributed level where it is now ready for implementation of sharding algorithms in order to become a highly scalable system, which can utilize memory capacity effectively throughout the cluster.

Edis could run as a cloud service to be used by other applications as clients. It would be especially beneficial if the applications that use Edis has a varying need for storage capacity, as Edis can scale accordingly. Also it would be a good fit if it is an application that need to execute a high frequency of storage operations, sine Edis is in-memory an relatively fast compared to accessing storage on disk.
8 Future Directions

At this point, a complete set of keys is stored at every node. This is good for redundancy, if nodes are lost, the data is replicated at other nodes. But if the system is to be run with over a hundred nodes it is not reasonable to have over a hundred copies, it will be a waste of storage space. Edis store data in the RAM, which is limited and expensive, however if using the leveldb backend, it is possible to utilize persisted disk space at the cost of speed. Effective use of storage space is therefore a high priority. There needs to be an evaluation of how many replicas are required in order to ensure that the system always can provide data even in the presence of node failures. This amount of replicas can then be distributed evenly among the individual nodes. This type of data replication and distribution algorithm could give Edis memory scalability so that for each node added to the system, there will be higher storage capacity. This type of algorithm is known as sharding, where each node has a partition of the complete dataset. The whole cluster in its' entirety would have several replicas of the dataset At this point, the storage capacity of Edis is limited to the capacity of the node which has the smallest RAM.

To get some idea about the performance of this system, a script can be written to investigate the behavior under load. The script could try to provoke the system by setting a value for a specific key at one node and deleting the value for this same key at another node. The operation should be repeated thousands of times. After setting a value, the script verifies that the key has the same value at both nodes. If this test is successful, one can derive that the value for the investigated key gets propagated across the cluster which means that the consistency measures that have been implemented are able to enforce the system to have a consistent data set, and thus be able to act as one single storage unit to any other system or end user that will interact with it. As mentioned in 6.3, clock drift exposes a weakness in the consistency of the system and should be paid extra attention. If the script is executed multiple times, each time having a client requesting to execute the operations at a higher frequency, the outcome can be compared to the other runs with a global snapshot of the system. If the global snapshot starts to display deviations from the former one, it is an indication that the consistency measures have failed to synchronise the cluster and thus the system capacity is established.
9 References

[1] Redis
Developer: VMware
The project: https://github.com/antirez/redis
Website: http://redis.io
Theory: https://github.com/karlseguin/the-little-redis-book
Visited 11 08 2013

[2] "Dynamo: Amazon’s Highly Available Key-value Store"
Written by: Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Sivasubramanian, Peter Vosshall and Werner Vogels
Presented: Symposium on Operating Systems Principles, October 14-17 2007, Stevenson, Washington, USA
Visited: 11 08 2013

Developer: Basho Technologies
The project: https://github.com/basho/riak
Website: http://docs.basho.com/riak/latest/
Visited 11 08 2013

Author: Brewer, E. A.
Published: Proceedings of the 19th Annual Acm Symposium on Principles of Distributed Computing , Portland, Oregon, USA
Date: 16-19 07 2000

Author: Neal Ford
Published: IBM DeveloperWorks
Date: 29 01 2013
Website: https://www.ibm.com/developerworks/library/j-ft20/
Visited: 10 03 2018

Author: Joe Armstrong
Published: Department of Microelectronics and Information Technology, Kungliga Tekniska Högskolan, TRITA–IMIT–LECS AVH 03:09
ISSN: 1651–4076
ISRN: KTH/IMIT/LECS/AVH-03/09–SE

Author: Fred Hebert
ISBN: 978-1-59327-435-1
Website: http://learnyousomeerlang.com
Visited 11 08 2013
[8] Erlang Documentation
Website: http://www.erlang.org/doc/apps/stdlib/index.html
Visited 11 08 2013

[9] “Time, clocks, and the ordering of events in a distributed system”
Author: Leslie Lamport (1978).
Published: Communications of the ACM 21 (7): 558-565.
Web version: http://www.stanford.edu/class/cs240/readings/lamport.pdf
Visited 11 08 2013

[10] “Paxos Made Live - An Engineering Perspective”
Authors: Tushar Chandra, Robert Griesemer, and Joshua Redstone
Date: June 26, 2007
Published: PODC ’07: 26th ACM Symposium on Principles of Distributed Computing.
Web version: http://research.google.com/archive/paxos_made_live.html
Visited 11 08 2013

Author: Leslie Lamport
Date: 01 11 2001
Visited 11 08 2013

[12] “Revisiting the PAXOS algorithm”
Authors: Roberto De Priscoa, Butler Lampson, Nancy Lyncha
Published: Theoretical Computer Science 243 (2000) 35-91
Visited 11 08 2013

Authors: Andre Allavena, Alan Demers, John E. Hopcroft
Published: 24th, 2005: Proceedings of the Annual ACM Symposium on Principles of Distributed Computing

[14] “Epidemic algorithms for replicated database maintenance”
Authors: Alan Demers, Dan Greene, Carl Hauser, Wes Irish, John Larson, Scott Shenker, Howard Sturgis, Dan Swinehart, Doug Terry

Authors: João Leitão, José Pereira, Luís Rodrigues.
Published: Proc. 37th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN’07)

[16] The Apache license version 2
Website: http://www.apache.org/licenses/LICENSE-2.0
Visited: 11 08 2013

[17] Resulting Edis source code (including the implementation pieces of this report)
Project: https://github.com/cbd/edis
Visited 20 12 2017
[18] “The part-time parliament.”
Author: Leslie Lamport

[19] Appendix: Vector Clock Implementation
Author: Joachim Nilsson
Date: 13 05 2013

Professor: Indranil Gupta
Created by: University of Illinois at Urbana-Champaign
Website: https://www.coursera.org/learn/cloud-computing
Visited: 05 03 2018

Author: Abdulaziz Aljabre
Published: Vol. 3 No. 1; January 2012, International Journal of Business and Social Science
Website: http://ijbssnet.com/journals/Vol_3_No_1_January_2012/26.pdf
Visited: 05 03 2018

[22] “8.3 Hash Tables, Data Structures and Algorithms”
Author: John Morris
Published: The University of Auckland, 1998
Website: https://www.cs.auckland.ac.nz/software/AlgAnim/hash_tables.html
Visited: 10 03 2018

[23] Sublime Text 2
Website: https://www.sublimetext.com/2
Visited: 10 03 2018

[24] Lager - Logging for Erlang
Website: https://github.com/erlang-lager/lager
Visited: 10 03 2018

[25] Rebar - Erlang Build Tool
Website: https://github.com/rebar/rebar
Visited: 10 03 2018

[26] “OpenDHT: A Public DHT Service and Its Uses”
Authors: Sean Rhea, Brighten Godfrey, Brad Karp, John Kubiatowicz, Sylvia Ratnasamy, Scott Shenker, Ion Stoica, and Harlan Yu, UC Berkeley and Intel Research
Published: SIGCOMM’05, August 21–26, 2005, Philadelphia, Pennsylvania, USA
Website: http://conferences.sigcomm.org/sigcomm/2005/paper-RheGod.pdf
Visited: 11 03 2018