Master's Thesis

Dynamic Fault Tolerance and Task Scheduling in Distributed Systems

Philip Ståhl
Jonatan Broberg

Department of Electrical and Information Technology,
Faculty of Engineering, LTH, Lund University, 2016.

Dynamic Fault Tolerance and Task Scheduling in Distributed Systems

Philip Ståhl
ada10pst@student.lu.se
Jonatan Broberg
elt11jbr@student.lu.se

Mobile and Pervasive Computing Institute (MAPCI)
Lund University

Advisor: Björn Landfeldt
bjorn.landfeldt@eit.lth.se

June 1, 2016

Printed in Sweden
E-huset, Lund, 2016

Abstract

Ensuring a predefined level of reliability for applications running in distributed environments is a complex task. Having multiple identical copies of a task increases redundancy and thereby the reliability. Due to varying properties of the often heterogeneous and vast number of components used in distributed environments, using static analysis of the environment to determine how many copies are needed to reach a certain level of reliability is insufficient. Instead, the system should dynamically adapt the number of copies as the properties of the system changes. In this thesis, we present a dynamic fault tolerant model and task scheduling, which ensures a predefined reliability by replicating tasks. Reliability is ensured over time by detecting failures, and dynamically creating new copies. Furthermore, the resources used are kept to a minimum by using the optimal number of task copies. Finally, the model was implemented using Ericsson's IoT-environment Calvin, thus providing a platform which can be used for further research and experiments.

Acknowledgements

We would like to thank our supervisor Björn Landfeldt and our examiner Christian Nyberg for their valuable input. We would also like to express our gratitude and appreciation to the people at MAPCI and Ericsson for their support during the work.

Table of Contents

1 Introduction **1**

1.1 Background and Motivation . . . 1
1.2 Related work . . . 2
1.3 Our contributions . . . 3
1.4 Goal . . . 4

2 Background **5**

2.1 Computational Environment . . . 5
2.1.1 Types of distributed computing 5
2.1.2 Dynamic versus static environments 6
2.2 Faults in distributed environments . . . 6
2.2.1 Types of Faults 7
2.2.2 Fault models 7
2.2.3 Failure distribution 8
2.3 Reliability . . . 9
2.3.1 Reliability definition 9
2.3.2 Modeling reliability 9
2.4 Fault tolerance techniques . . . 10
2.4.1 Checkpoint/Restart 11
2.4.2 Rollback recovery 11
2.4.3 Replication 11
2.4.4 Load balancing 13
2.5 Task scheduling . . . 14
2.6 Monitoring . . . 14
2.7 Virtualization and containers . . . 15

3 Design of a dynamic fault tolerant system **17**

3.1 Methodology . . . 17
3.2 Introduction . . . 17
3.3 Limitations . . . 18
3.4 System model . . . 19
3.4.1 Computational environment 19
3.4.2 Storage of data 20
3.4.3 Application model 20

3.4.4 Replication scheme 21
3.4.5 Fault model 22
3.5 Monitoring . . . 23
3.5.1 Heartbeats 23
3.5.2 Monitoring system reliability 24
3.6 Reliability model . . . 24
3.6.1 Definitions 24
3.6.2 Expressing reliability 24
3.6.3 Expressing time t 25
3.7 Scheduling algorithm . . . 28
3.7.1 Optimization 30
3.8 Handling node failure . . . 30
3.8.1 Selecting node 32
3.9 Self-adapting . . . 33
3.9.1 Adapting time t 34
3.9.2 Adapting nodes' *MTBF* 34
3.10 Implementation . . . 34
3.10.1 Calvin 34
3.10.2 Our contributions to Calvin 36

4 Evaluation 39
4.1 Values used in the experiments . . . 39
4.2 Computational environment . . . 39
4.2.1 Simulating node failure 40
4.2.2 Node specification 41
4.3 Storage used . . . 41
4.4 Application used in experiments . . . 41
4.5 Experiments . . . 42
4.5.1 Measurement of time t_R 42
4.5.2 Measurement of node failure detection time 44
4.5.3 Measurement of replication time t_r 45
4.5.4 Ensuring a certain reliability level 48
4.5.5 Optimal number of replicas 51
4.5.6 Self-adaptive reliability model 55
4.5.7 Energy efficient for many applications 56
4.5.8 Considering nodes' load 58

5 Discussion 63

6 Future Work 67

7 Conclusions 69

References 71

A Figures 77

List of Figures

3.1 Computational environment 20
3.2 Application model 21
3.3 Application model with replicas 22
3.4 Replication request 27
3.5 Number of replicas needed 28
3.6 Handling a node failure 33

4.1 Computational environment in experiments 42
4.2 Fitted distributions in experiment 4.5.1, $MTBF = 30$ s 44
4.3 Fitted distributions in experiment 4.5.1, $MTBF = 10$ s 44
4.4 Replication time in experiment 4.5.3, server-server 46
4.5 Replication time in experiment 4.5.3, laptop-laptop 47
4.6 Time parts in experiment 4.5.3, server-server, increasing variable size . 48
4.7 Time parts in experiment 4.5.3, server-server, increasing queue size . . 48
4.8 Time parts in experiment 4.5.3, laptop-laptop, increasing variable size 49
4.9 Time parts in experiment 4.5.3, laptop-laptop, increasing queue size . 49
4.10 Reliability over time in experiment 4.5.4 51
4.11 Total number of replicas in experiment 4.5.5.1 52
4.12 Number of replicas in experiment 4.5.5.1 on nodes with $MTBF = 40$ s 53
4.13 Number of replicas in experiment 4.5.5.1 on nodes with $MTBF = 15$ s 53
4.14 Number of replicas in experiment 4.5.5.1 on nodes with $MTBF = 7.5$ s 54
4.15 Total number of replicas in experiment 4.5.5.2 54
4.16 Number of replicas in experiment 4.5.5.2 on nodes with $MTBF = 25$ s 55
4.17 Number of replicas in experiment 4.5.5.2 on nodes with $MTBF = 10$ s 55
4.18 Total number of replicas in experiment 4.5.6 57
4.19 Nodes' reliabilities in experiment 4.5.6 57
4.20 Total number of nodes used in experiment 4.5.7 58
4.21 Number of replicas in experiment 4.5.7 on nodes with $MTBF = 40$ s 58
4.22 Number of replicas in experiment 4.5.7 on nodes with $MTBF = 15$ s 59
4.23 Number of replicas in experiment 4.5.7 on nodes with $MTBF = 7.5$ s 59
4.24 CPU usage for nodes in experiment 4.5.8 60
4.25 Number of replicas on node *Kevin* in experiment 4.5.8 61
4.26 Number of replicas in experiment 4.5.8, all nodes but *Kevin* 61

A.1 Data flow at replication 78
A.2 System before a node failure 78
A.3 System after a node failure 78
A.4 Communication after a node failure 79
A.5 Average replication time with varying state size 80
A.6 Time between failures in experiment 4.5.6 80
A.7 Node failure detecting time, best case 81
A.8 Node failure detecting time, worst case 81

List of Tables

4.1 Average t_{R} and the 95th percentile in experiment 4.5.1 43
4.2 Time, deviation and RSD in experiment 4.5.3, increased variable size . 50
4.3 Time, deviation and RSD in experiment 4.5.3, increased queue size . . 50
4.4 *MTBF* for nodes in experiment 4.5.5.1 51

List of Symbols

λ	Failure rate
σ	Standard deviation
μ	Mean value
$f(t)$	Probability of a replica failing within time t
$F(t)$	Probability of a node failing within time t
$P(k)$	Probability of k failures
$R(t)$	Probability of not experience a failure within a time t
$R_{\mathbf{req}}$	Required reliability
t	Time from a failure happening to the system is restored
$t_{\mathbf{create}}$	Time for creating a new replica
$t_{\mathbf{d}}$	Time to detect a failure
$t_{\mathbf{de\text{-}serialize\ state}}$	Time for de-serializing a task's state
$t_{\mathbf{get\ state}}$	Time to get a task's state
$t_{\mathbf{h}}$	Time between sending heartbeats
$t_{\mathbf{R}}$	Total time for having a new replica operational
$t_{\mathbf{r}}$	Time for replicating a task to another node
$t_{\mathbf{replication\ msg}}$	Time for sending all information needed for creating a replica
$t_{\mathbf{response}}$	Time for sending a response
$t_{\mathbf{send\ reply}}$	Time for sending a reply
$t_{\mathbf{serialize\ state}}$	Time for serializing a task's state
$t_{\mathbf{timeout}}$	Timeout time for listening for heartbeats
$t_{\mathbf{transmit\ state}}$	Time for transmitting a task's state
$t_{\mathbf{query\ storage}}$	Time for querying the storage and receiving a response

Abbreviations

CHDS	Centralized Heterogeneous Distributed Systems
DHT	Distributed Hash Table
HDCS	Heterogeneous Distributed Computing Systems
IoT	Internet of Things
MAS	Multi-Agent System
MTBF	Mean Time Between Failures
MTTF	Mean Time To Failure
RMS	Resource Management System
RSD	Relative Standard Deviation
TCP	Transmission Control Protocol
UDP	User Datagram Protocol
VM	Virtual Machine

Chapter 1

Introduction

1.1 Background and Motivation

Ensuring a predefined level of reliability is of major concern for many cloud service providers. Cloud computing, and distributed computing in general, is growing rapidly, and users demand more services and higher availability and reliability. Therefore, providing fault tolerant systems and improving reliability by more sophisticated task scheduling algorithms have become very important and gained plenty of research interest.

Distributed systems often consist of heterogeneous hardware and any of the, often vast number of, components may fail at any time. Consequently, as computational work is distributed across multiple resources, the overall reliability of the application decreases. To cope with this issue, a fault tolerant design or error handling mechanism needs to be in place. Ensuring reliability raises complexity to the resource allocation decisions and fault tolerance mechanisms in highly dynamic distributed systems. For cloud service providers, it is necessary that this increased complexity is taken care of without putting extra burden on the user. The system should therefore ensure these properties in a seamless way.

In some cases, vendors, such as carrier providers, are obliged by law to achieve a specific level of availability or reliability. In such cases, quality aspects such as latency and resource usage may need to be sacrificed to reach the required level. By using static and dynamic analysis of the infrastructure and the mean-time-between-failure for the available resources, a statistical model can be created to verify that the required level is reached. However, due to the dynamic properties of distributed environments, such a model must also be dynamic.

Furthermore, fault tolerance is of particular importance for long running applications or services where a predefined level of reliability is required. Despite fulfilling the required level at the time of deployment, as the state of the system changes, the required reliability may no longer be fulfilled and actions must be taken to still reach the required level.

One way of increasing the reliability is by replicating application tasks, i.e. creating identical copies. Having identical copies, also called replicas, results in increased redundancy and allows for continued execution of the application, despite the event of failure of a task replica. Seamlessly being able to continue the execution or computations performed, without losing any data, is of particular

interest for stream processing applications.

A drawback of replication is the extra resources needed. If replicating a task n times with every replica performing the same computation on the same input, n times as many resources are needed, hence a great deal of computational resources are wasted. Dynamic analysis of the system and an adaptive scheduling technique could help in determining on which resources replicas should be placed for optimal resource usage and load balancing.

In this master thesis, a model with dynamic fault tolerance is presented, which ensures the required reliability is met by replicating tasks. By dynamically adapting to changing properties of the system and available resources, it also ensures that the optimal number of replicas are used.

1.2 Related work

The interest in reliability for distributed systems has gained increased attention recently [57]. Due to that heterogeneous resources often compose distributed environments such as cloud and grid systems, ensuring reliability is a complex task [2, 17].

A large number of scheduling techniques have been designed, aiming at maximizing reliability of computational jobs in distributed environments under various constraints such as meeting task deadlines or minimizing execution time [1, 13, 14, 18, 27, 30, 32]. Maximizing reliability is for these algorithms a secondary concern, while meeting the constraints is the primary. In contrast, [33–36] have developed models which focus on increasing the reliability. Common for these scheduling techniques are that while they try to maximize reliability, they do not ensure that a certain level of reliability is met. Furthermore, the algorithms are usually static in the way that they do not account for the dynamic behavior of distributed systems. In addition, they make assumptions such as known execution times of tasks, which make them unsuitable for long running applications or services without a known execution time.

Plenty of work has been done in the area of designing fault tolerant systems by using checkpoint/restart techniques [40, 64, 65]. These techniques rely on the notion of a stable storage, such as a hard-drive, which is persistent even in the case of a system failure. Checkpoint techniques are usually employed for applications or jobs where the computations executed take a very long time. For such computations, a substantial amount of computational resources are wasted if the job has to redo all computations in case of a failure.

Some previous attempts at designing fault tolerant systems by the use of replication have been made [11, 23, 37, 38, 44]. In [38], a static number of replicas is used for every application being deployed. Furthermore, they do not guarantee that all replicas are deployed, instead they use a best-effort approach, where replicas are deployed only if resources are available. In contrast to [38], [11, 23, 37] dynamically determine the number of replicas based on the state of the system. However, they are all static in that failed replicas are not restarted, and do therefore not ensure the reliability is met over time. Furthermore, while [37] dynamically determines the number of replicas to use, the scheduling decisions

on which resources to put the replicas is done after the number of replicas has been determined. As the reliability vary between resources in a heterogeneous environment, the number of replicas needed depends on which resources that are used. Therefore, determining the number of replicas needed and where to place them should be a joint process.

A quite old but still relevant work is found in [12] in which a framework for dynamically replicating tasks in a Multi-Agent Systems (MAS) is presented. The authors introduce a software architecture which can act as support for building reliable MAS. Since the available resources often are limited, they say that it is not feasible to replicate all components. Therefore, they introduce the concept of criticality for each agent, which is allowed to evolve during runtime. An agent's criticality is calculated using the CPU usage and communication activity of the agent, and is used to determine the number of replicas, along with a predefined minimum number of replicas. The proposed solution also allows for dynamically adapting the number of replicas and the replication strategy itself (passive/active) in order to maximize the reliability of the agents based on available resources.

Other approaches to improve reliability in MAS by the use of replication are presented in [54–56]. While being adaptive to system state, the solution presented in [55] still faces the problem of having a single-point-of-failure due to a communication proxy. This problem is avoided in [54], where a decentralized solution is proposed, and where the number of replicas and their placement depends on the system state. However, instead of ensuring a given level of reliability is met, they aim at maximizing the reliability and availability based on available resources.

A dynamic and adaptive algorithm, which dynamically varies the number of replicas depending on system load is presented in [40]. The proposed algorithm does not ensure a certain reliability. Instead, it reduces the number of replicas during peak hours, in order to reduce system load. Since the reliability of a system decreases during higher load [20, 21], the number of replicas should be increased instead of decreased in order to stay above the required level of reliability.

A fault tolerant scheduling technique incorporating a replication scheme is presented in [39]. While being dynamic in that failed replicas are restarted, it is static in that the user defines the number of replicas to use, hence it does not ensure a specific level of reliability is met.

The techniques used in [12, 29, 45] are more dynamic and adaptive to the dynamic behavior of distributed systems. However, reliability is defined as producing the correct result, and is achieved by techniques like *majority voting* and *k-modular redundancy*. An adaptive approach, which adapts to changes in the execution environment is presented in [28]. They present an adaptive scheduling model based on reinforcement learning, aiming at increasing the reliability. However, they assume that a task's profile, including execution time, is available.

1.3 Our contributions

To our knowledge, no previous attempt has been made which in a fully dynamic manner ensures a predefined level of reliability for long running applications or

services. Some previous work dynamically calculates the number of replicas, but are static in that failed replicas are not restarted, while others use a static number of replicas, and dynamically restart failed ones.

We propose a framework which ensures a user determined level of reliability by the use of replication. Furthermore, the method ensures a minimized use of resources by not using more replicas than needed, and by minimizing the number of resources needed. This is achieved by placing replicas on the most reliable resources first and foremost. Finally, the system is periodically monitored in order to adapt to changing system properties.

The framework is not limited to a specific type of distributed environment, nor the reliability model used. The reliability model and the task placement decisions are easily replaced to consider more parameters, or to include load balancing.

Our model is implemented using the actor-based application environment *Calvin* [31], developed by Ericsson. While *Calvin* is mainly an environment for Internet of Things (IoT) applications, it suites our purpose well. The model is evaluated by running a set of experiments in a small cluster. Furthermore, the implementation provides a platform for further research and experiments.

1.4 Goal

The goal of this thesis was to devise a method for dynamically ensuring a predefined level of reliability for distributed applications or services by dynamically replicating tasks. The goal was further to implement the method and provide a flexible and extensible platform, which can be used for further research and experiments.

First, a reliability model was designed, describing the reliability of the available resources, and for applications using these resources.

Secondly, a framework was designed which automatically detects node failures and based on the reliability of the available resources creates enough replicas to reach the required reliability level. Furthermore, the system and its running applications are periodically monitored in order to adapt the resources' reliability and the number of replicas needed as the properties of the system vary over time.

Lastly, the model was implemented and tested using the IoT application framework *Calvin*.

The report is structured as follows: in chapter 2 all necessary background theory is provided, in chapter 3 we present our model and contribution in more detail, chapter 4 presents an evaluation of our solution and in chapter 5 the solution is discussed. Chapter 6 presents future work and chapter 7 concludes the report.

Chapter 2

Background

In this chapter we provide all the necessary background theory to fully understand the rest of the report.

2.1 Computational Environment

The computational environment used in this thesis is distributed computing, i.e. several resources working together towards a common goal.

2.1.1 Types of distributed computing

Distributed Computing Systems (DCS) are composed of a number of components or subsystems interconnected via an arbitrary communication network [15,47]. There are a number of different types of distributed environments, e.g. grids, clusters, clouds and heterogeneous distributed computing systems.

2.1.1.1 Grid computing

A grid is a collection of autonomous resources, that are distributed geographically and across several administrative domains, and work together to achieve a common goal, i.e. to solve a single task [6,7,48].

Each domain in a grid usually has a centralized scheduling service called Resource Management System (RMS) which accepts job execution requests and sends the jobs' tasks to the different resources for execution [48].

The reliability of the grid computing is very critical but hard to analyze due to its characteristics of massive-scale service sharing, wide-area network, heterogeneous software/hardware components and complicated interactions among them [17].

2.1.1.2 Cluster

A cluster system is usually a number of identical units managed by a central manager. It is similar to a grid, but differ in that resources are geographically located at the same place. The resources work in parallel under supervision of a

single administrative domain. From the outside it looks like a single computing resource [6].

2.1.1.3 Cloud

Cloud has been described as the next generation of grids and clusters. While it is similar to clusters, the main difference is that cloud consists of multiple domains [6]. The domains can be geographically distributed, and software and hardware components are often heterogeneous. Therefore, analyzing and predicting workload and reliability are usually very challenging [2].

2.1.1.4 Heterogeneous distributed computing systems

A Heterogeneous Distributed Computing System (HDCS), is a system of numerous high-performance machines connected in a high-speed network. Therefore, high-speed processing by computational heavy applications is possible [14].

The majority of distributed service systems can be viewed as a Centralized Heterogeneous Distributed System (CHDS). A CHDS consists of heterogeneous sub-systems which are managed by a centralized control center. The sub-systems have various operating platforms and are connected in diverse topological networks [46].

2.1.2 Dynamic versus static environments

A distributed computing environment can be either static or dynamic. In a static environment, only homogenous resources are usually installed [6]. For load-balancing and scheduling algorithms, prior knowledge of node capacity, processing power, memory, performance and statistics of user requirements are required. Changes in load during runtime are not taken into account which makes the environment easy to simulate but not well suited for heterogeneous resources. Once the system has been put into place, resources are neither added nor removed from the system.

In contrast to static environments, a dynamic environment usually consists of heterogeneous resources [6]. Once the system has been put into place, resources can be dynamically added or removed. In such environments, prior knowledge is therefore not enough for load-balancing and scheduling algorithms, since the requirements of the user and the available resources can change during runtime. Runtime statistics are therefore usually collected and taken into account. Dynamic environments are difficult to simulate, but algorithms exist which easily adopt to runtime changes.

2.2 Faults in distributed environments

In distributed environments, a large number of faults can occur due to its complexity. Therefore, when modeling faults and reliability in such environments, a

fault model is usually employed, describing which kinds of failures that are being considered.

2.2.1 Types of Faults

A fault is usually used to describe a defect at the lowest level of abstraction [9]. A fault may cause an error which in turn may lead to a failure, which is when a system has not behaved according to its specification.

In distributed environments, especially with heterogeneous commodity hardware, several types of failures can take place, which affect the running applications. Such failures include, but are not limited to, overflow, timeout, resource missing, network, hardware, software, and database failure [17]. Failures are usually considered to be either [48]:

- Job related
- System related
- Network related.

In [20], almost ten years of real-world failure data of 22 high performance computing systems was studied and concluded hardware failures to be the single most common type of failure, ranging from 30 to more than 70 percent depending on hardware type, while 10 to 20 percent of the failures were software failures.

2.2.2 Fault models

When studying the reliability of distributed systems or applications running in distributed computing environments, one usually starts with specifying which fault model that is used. A reliability model is then designed, and validated with respect to this fault model [9]. In the fault models described in this section, and in the rest of the report, a computational resource is referred to as a node.

2.2.2.1 Byzantine fault model

The Byzantine fault model allows nodes to continue interaction after failure. Correctly functioning nodes cannot automatically detect if a failure has occurred. Even if it was known that a failure occurred, nodes cannot detect which node that has failed.

Furthermore, the system's behavior can be inconsistent and arbitrary [8]. Nodes can fail (become Byzantine) at any point of time and stop being Byzantine at any time. A Byzantine node can send no response at all, or it can send an incorrect result. All Byzantine nodes might send the same incorrect result, thereby making it hard to identify malicious nodes [29]. The Byzantine fault model is very broad since it allows failed nodes to continue interacting, but it is therefore also very difficult to simulate and to analyze.

2.2.2.2 Fail-stop fault model

The fail-stop model, also called the crash-stop model, is in comparison to the Byzantine fault model much simpler. When a node fails it stops producing any output and stops interacting with the other nodes [9]. This allows for the rest of the system to automatically detect when a node has failed. Due to its simplicity, it does not handle subtle failures such as memory corruption but rather failures such as system crashes [8].

2.2.2.3 Crash-failure fault model

The crash failure model is quite alike the fail-stop model with the difference that nodes do not automatically detect the failure of a node [9, 56].

2.2.2.4 Fail-stutter fault model

Since the Byzantine model is very broad and complicated and the fail-stop model doesn't represent enough real-world failures, a third middle ground model has been developed. It is an extension of the fail-stop model but differ in that it also allows for performance fault, such as unexpectedly low performance of a node [8].

2.2.3 Failure distribution

Models describing the nature of failures in distributed computing environments are usually based on certain assumptions and are usually only valid under those assumptions. Commonly made assumptions about failures and the computational environment are [15–17, 29, 42, 46]:

- Each component in the system has only two states: *operational* or *failed*
- Failures of components are statistically independent
- Components have a constant failure rate, i.e. the failure of a component follows a Poisson process
- Fully reliable network

When modeling reliability for grid systems, it is common to also assume a fully reliable Resource Management System (RMS) [7, 44], which is a non-negligible assumption since the RMS is a single-point-of-failure.

Constant failure rates are not likely to model the actual failure scenario of a dynamic heterogeneous distributed system [14]. The failure rate for a hardware component often follows a bathtub shaped curve [2]. The failure rate is usually higher in the beginning since the probability that a manufacture failure would affect the system is higher in the beginning of the system's lifetime. After stabilizing, the failure rate drops and later increases again due to that the component gets worn out.

Statistically independent failures are also not very likely to reflect the real dynamic behavior of distributed systems [2, 17]. Faults may propagate throughout the system, thereby affecting other components as well [5]. In grid environments,

a sub-system may consist of resources using a common gateway to communicate with the rest of the system. In such a scenario, the resources do not use independent links [33]. As failures are likely to be correlated [19], the probability of failure increases with the number of resources a task uses.

Other factors also affect the likelihood of resource failures. Several studies have concluded a relationship between failure rate and the load of the system [20,21]. Furthermore, [20, 21] also show that failures are more likely to occur during daytime than at night, which may be a consequence of the system load being higher during daytime. In addition, components which have failed in the past are more likely to fail again [21].

While a Poisson process is commonly used to describe the probabilities of failures, [20] shows that failures are better modelled by a Weibull distribution with a shape parameter of 0.7 - 0.8. However, despite not always reflecting the true dynamic failure behavior of a resource, a Poisson process has been experimentally shown to be reasonably useful in mathematical models [58]. The Poisson distribution expresses the probability of k failures during a specific period of time, and is defined as follows:

$$P(k \text{ failures}) = \frac{\lambda^k \cdot e^{-\lambda}}{k!} \tag{2.1}$$

where λ is the failure rate, i.e. the average number of failures occurring in time t.

2.3 Reliability

2.3.1 Reliability definition

Reliability in the context of software applications can have several meanings, especially for applications running in distributed systems. Often, reliability is defined as the probability that the system can run an entire task successfully [1, 15,41,42,46,49,50]. A similar definition, for applications running in distributed environments, is that reliability is the probability of a software application to perform its intended functions for a specified period of time [2–4], and is commonly used for applications with time constraints. Finally, reliability can also be defined as the probability that a task produces the correct result [3,7,29,44,45]. The latter is usually used together with the Byzantine fault model.

2.3.2 Modeling reliability

The reliability of a system highly depends on how the system is used [3]. In order to determine the reliability of a system, one needs to take all factors affecting the reliability into account [2]. However, including all factors is not feasible. In [26], 32 factors affecting the reliability of software are listed, excluding environmental factors such as hardware and link failure. Other environmental conditions affecting reliability include the amount of data being transmitted, available bandwidth and operation time [17,42].

For distributed applications, the probability of failure increases since it is dependent on more resources [15]. Most reliability models are based on the Mean-Time-To-Failure (MTTF), or the Mean-Time-Between-Failures (MTBF), of resources [16]. Conventionally, MTTF refers to non-repairable resources, while MTBF refers to repairable objects [23].

Definition 2.1. *The Mean-Time-To-Failure for a component is the average time it takes for a component to fail, given that it was operational at time zero.*

Definition 2.2. *The Mean-Time-Between-Failure for a component is the average time between successive failures for that component.*

The $MTBF$ can be calculated as

$$MTBF = \frac{\text{total time}}{\text{number of failures}} \tag{2.2}$$

From t and the $MTBF$, the failure rate λ used in eq. (2.1) can be calculated as $\lambda = t/MTBF$. Equation (2.1) can therefore be re-written as

$$P(k \text{ failures during time } t) = \frac{\left(\frac{t}{MTBF}\right)^k \cdot e^{-\left(\frac{t}{MTBF}\right)}}{k!} \tag{2.3}$$

The probability of surviving corresponds to having zero failures, and can be expressed as

$$P(0 \text{ failures during time } t) = \frac{\left(\frac{t}{MTBF}\right)^0 \cdot e^{-\left(\frac{t}{MTBF}\right)}}{0!} = e^{-\left(\frac{t}{MTBF}\right)} \tag{2.4}$$

2.4 Fault tolerance techniques

Fault tolerance techniques are used to either predict failures and take appropriate actions before they occur [52], or to prepare the system for failures, and take appropriate actions first when they occur. Considering the whole life-span of a software application, fault tolerant techniques can be divided into four different categories [2]:

1. Fault prevention - elimination of errors before they happen, e.g. during the development phase
2. Fault removal - elimination of bugs or faults after repeated testing phases
3. Fault tolerance - provide service complying with the specification in spite of failure
4. Fault forecasting - predicting or estimating faults at architectural level during design phase or before actual deployment

Limited to already developed applications, fault tolerance techniques can be divided into reactive and proactive techniques [52]. A reactive fault tolerant technique prepares the system for failure, and reacts when a failure occurs and tries to reduce the effect of the failure. They therefore consists of preparing for, detecting, and recovering from failure in order to allow computations to continue [5]. The proactive technique on the other hand, tries to predict failures and proactively replace the erroneous components.

Common fault tolerance techniques include *checkpointing*, *rollback recovery* and *replication* [5].

2.4.1 Checkpoint/Restart

Fault tolerance by the use of periodic checkpointing and rollback recovery are the most basic form of reactive fault tolerance techniques [8].

Checkpointing is a fault tolerance technique which periodically saves the state of a computation to a persistent storage [5, 8]. In the case of failure, a new process can be restarted from the last saved state, thereby reducing the amount of computations needed to be redone.

2.4.2 Rollback recovery

Rollback recovery is a technique in which all actions taken during execution are written to a log. At the event of failure, the process is restarted, and the log is read and all actions replayed, thereby re-constructing the previous state [8]. In contrast to checkpointing, rollback recovery returns the process to the most recent state, not only the last saved one. Rollback recovery can be used in combination with checkpointing in order to reduce recovery time by not having to replay all actions, but only those from the latest checkpoint.

2.4.3 Replication

Task replication is a commonly used fault tolerant technique, and is based on the assumption that the probability of a single resource failing is greater than the probability of multiple resources failing simultaneously [51]. Replication can be used both as a reactive and proactive fault tolerance technique.

Using replication, several identical processes are scheduled on different resources and simultaneously perform the same computations [5]. With the increased redundancy, the probability of at least one replica finishing increases at the cost of more resources being used. Furthermore, the use of replication effectively protects against having a single-point-of-failure [51].

Replication also minimizes the risk of failures affecting the execution time of jobs, since it avoids the re-computation typically necessary when using checkpoint/restart techniques [37].

There are three different strategies for replicating a task: *active*, *semi-active* and *passive*. A replica, whether active, semi-active or passive replication is used, is defined as [23]:

Definition 2.3. *The term replica or task replica is used to denote an identical copy of the original task*

2.4.3.1 Active replication

In active replication, there are one or several replicas of a task running simultaneously. All replicas receive an exact copy of the task's input and they all perform the same computations.

Often, there is one primary task and several back-up replicas. The primary task is monitored by the back-up replicas for incorrect behavior and if the primary task fails or behaves in an unexpected way, one of the back-up replicas will promote itself as the primary task [8]. Since the back-up replicas already are in an identical state as the primary task, the transition will take negligible amount of time.

Depending on the set-up, the primary task could be the only one producing output, or all replicas could produce output. In the latter case, the receiver will receive multiple results. A consensus algorithm could then be deployed to determine which result is correct, and thereby protect against Byzantine faults. More on this in section 2.4.3.4.

Active replication is feasible only if all replicas receive exactly the same input. When replicating, the task replicas are likely not to be synchronized. Therefore, if they perform computations or make external requests which are time-dependent, they are likely to produce different results. To avoid this, this kind of replication is usually done for tasks performing deterministic[1] computations, which are not time-dependent.

A drawback with active replication is that having n identical copies of a task, all computations are performed n times, and thereby wasting computational capacity. But while replication increases the system load, it may help to improve performance by reducing task completion time [53].

2.4.3.2 Semi-active replication

Semi-active replication is very similar to active replication, but differs in that decisions common for all replicas are taken by one site. This solves the issue with time-dependent computations, since all such computations could be performed by one site. However, this introduces a single-point-of-failure.

2.4.3.3 Passive replication

Passive replication is the case when a second machine, typically in idle or power off state has a copy of all necessary system software as the primary machine. If the primary machine fails the "spare" machine takes over, which might incur some interrupt of service. This type of replication is only suitable for components that have a minimal internal state, unless additional checkpointing is employed.

[1] A deterministic function always produces the same output given a certain input.

2.4.3.4 Consensus

In active and semi-active replication, several results are being produced. If non-deterministic computations are done, some form of consensus algorithm is usually deployed to determine which result is correct. The consensus problem can be viewed as a form of agreement. A consensus algorithm lets several replicas execute in parallel, independent of each other and the receiver of their results determine which result is considered correct. Such algorithms are usually used with the Byzantine fault model section 2.2.2.1, where a resource can produce an incorrect result.

Based on achieving consensus, two different redundancy strategies can be identified, traditional and progressive redundancy [29]. In traditional redundancy an odd number of replicas run simultaneously and afterwards a voting takes place to determine which result is correct. The result with the highest number of votes is considered correct and consensus is reached.

In progressive redundancy the number of replicas needed is minimized. If with traditional redundancy, $k \in \{3, 5, 7...\}$ replicas are executed, progressive redundancy first executes $(k+1)/2$ replicas and reaches consensus if all replicas return the same result. If some replica return a deviant result, an additional number of replicas are executed until enough replicas have returned the same result, i.e. consensus is reached. In worst case k replicas are executed, the same as for traditional redundancy. A disadvantage with progressive consensus is that it might take longer time if consensus is not reached after the first iteration.

Finally, [29] present a third strategy, an iterative redundancy alternative which focuses more on reaching a required level of reliability in comparison to reaching a certain level of consensus.

2.4.4 Load balancing

The term load balancing is generally used for the process of transferring load from overloaded nodes to under-loaded nodes in order to improve the overall performance. Load balancing techniques for distributed environments consists of two parts, resource allocation and task scheduling.

Load balancing algorithms can be divided into three categories based on the initiation of the process:

- Sender initiated - an overloaded node sends requests until it finds a proper node which can accept its load.
- Receiver initiated - an under-loaded node sends requests for more work until it finds an overloaded node.
- Symmetric - a combination of sender initiated and receiver initiated.

Load balancing can also be divided into static and dynamic load balancing. In contrast to static load balancing algorithms, dynamic algorithms takes the nodes' previous states and performance into account. The static load balancing only consider properties such as processing power and available memory which might lead to the disadvantage that the selected node gets overloaded [10].

2.5 Task scheduling

Task scheduling is the process of mapping tasks to available resources. A scheduling algorithm can be divided into three simple steps [60]:

1. Collect the available resources
2. Based on task requirements and resource parameters, select resources to schedule the tasks on
3. Send the tasks to the selected resources

Based on which requirements and parameters are considered in step 2 above, a task scheduling algorithm can achieve different goals. They can for example aim at maximizing the total reliability, minimizing the overall system load, or meeting all the tasks' deadlines [59].

Task scheduling algorithms can be divided into static and dynamic algorithms depending on whether or not the scheduling mapping is based on pre-defined parameters or if the parameters might change during runtime [30].

Many studies have been done aiming at improving reliability by improved task allocation in distributed systems by various scheduling algorithms. However, they only consider some system constraints such as processing load, memory capacity, and communication rate [18]. Finding an optimal solution and maximizing the overall system reliability at the same time is a NP-hard problem [18, 32, 59].

Furthermore, taking many factors into account when scheduling tasks results in a big overhead, thus reducing performance.

2.6 Monitoring

Monitoring is common in distributed systems for detecting when resources fail. A commonly used technique is the use of heartbeats, where light-weight messages are sent, usually using UDP, between resources, informing each other that they are still operational. The heartbeats are periodically sent with a certain frequency. If a resource stops receiving heartbeats from another resource, it is assumed to have died. Another similar technique is to send a message and wait for a reply, and if no reply is received within a certain time, the resource is assumed to be dead [62].

Monitoring is also an important part of dynamic fault tolerant techniques which adapts to changing system behavior. Depending on which parameters are taken into account in the reliability model or scheduling algorithm used, the system resources must be monitored and various information must be shared. Resource information, such as current load of a resource, can be shared using either a pull or push strategy. If using a centralized storage, a pull strategy involves the storage requesting information from the resources, while a push strategy involves the resources sending the information to the storage without it being requested.

Furthermore, the monitoring can take place in various forms. Using pull/push strategy, the information is sent from the resources to some monitoring system. But the monitoring system can also collect the information needed, e.g. by scanning logs [62].

2.7 Virtualization and containers

Virtualization is a broad term of creating a virtual version of something, e.g. a server, a hard drive or a private network (VPN). The advantages of virtualization are savings in space and hardware cost. Furthermore, the system becomes more dynamic. For instance, it is possible to run several virtual servers on one physical server. With virtualization the software and the hardware gets separated which enables the possibility of moving a virtual machine, a VM, from one physical location to another [66]. This also allows for starting several identical servers, by starting several virtual machines using the same image.

A container is very similar to a virtual machine but more light-weight. Containers running on the same physical machine share the machines operating system and binaries while virtual machines running on a single physical machine have their own operating systems and dedicated resources, isolating them from each other [61].

Chapter 3

Design of a dynamic fault tolerant system

In this chapter, the key components of our fault tolerant framework are described. Of the various fault tolerant techniques described in section 2.4, our model focuses on fault tolerance, using reactive techniques.

3.1 Methodology

Having a fault tolerant framework or model includes being able to express the reliability of resources, and applications using those resources, as well as being able to detect failures. The methodology for devising our model is mainly literature study as well as meetings and discussions with our supervisor and colleagues at MAPCI.

We implemented the model using Ericsson's actor-based application environment Calvin [31], and conducted a set of experiments to show its usefulness. The implementation provides a platform which can be used for further research and experiments. The model is not dependent on the reliability model used, and it could easily be replaced.

Since implementing our model using Calvin, some understanding of the results may be lost. Calvin is designed for light-weight IoT applications, and not for data stream processing applications where tasks are replicated. However, the implementation provides a mean of testing other algorithms and reliability models than the ones used in this thesis, and we believe that the results can still give a good first interpretation of the possibilities, which are further discussed in chapter 5.

3.2 Introduction

In the case of data stream processing, reliability is of particular interest in order not to lose any valuable data. In this thesis, focus is on the case when a process of some kind produces data which needs to be transformed or processed by a task T, which sends the result to a consumer. The task T is running within a cluster, and the user of this service demands a certain level of reliability, and active replication is used to ensure this.

Choosing streaming services with deterministic processing allows us to avoid timing issues and cases when the consumer receives different results from the replicas. We therefore adapt the fail-stop fault model described in section 2.2.2.2. However, since active replication will be used where the receiver receive a result from each replica, our model could be extended with consensus algorithms such as those presented in section 2.4.3.4 to determine whether or not the correct result was received. One could then adapt the broader Byzantine fault model.

Furthermore, using long running applications or services, such as stream processing applications, allows us to focus on ensuring a required reliability despite event of failures. Therefore, in contrast to the various scheduling algorithms in [1,13,14,18,27,30,32] aiming at meeting task deadlines or minimizing execution time, a greedy scheduling algorithm will be presented which solely aims at reaching a required level of reliability with the minimum number of replicas. The scheduling algorithm presented only takes the reliability of the available nodes into account, but as shown in experiment 4.5.8, it can easily be replaced by another algorithm, which for example also takes the nodes' loads into account.

Unlike [11,23,37,38,44], we propose a fully dynamic model which monitors the system with its running applications and services, and dynamically creates new replicas in order to ensure that the required reliability is met. Determining the number of replicas needed only when deploying an application or service may suffer if the system is in a reliable state at the time of deployment, but later becomes less reliable. As mentioned in section 2.2.3, the reliability of the resources vary over time, the system and its running applications must therefore be periodically monitored and the number of replicas increased or decreased over time. In our model this is achieved by monitoring both how many replicas are operational, on which nodes they are executing, as well as detecting node failures, in order to dynamically adapt the reliability model as the properties of the system changes.

The rest of this chapter is composed as follows, in section 3.3 we go through the limitations we have done. Section 3.4 gives a description of the system model used, e.g. in which computational environment we have tested our model, how faults were modeled and their distribution and what kind of applications we used. In section 3.6 - 3.9 we present our main contributions consisting of a reliability model, a scheduling algorithm and how our model is self-adapting. Last in this chapter, in section 3.10, we briefly describe how Calvin works and present the main parts of our implementation of the model.

3.3 Limitations

As mentioned in section 2.2.3, a commonly used assumption is having fully reliable links between nodes. We also make this assumption, and thereby limit our model to only consider node failures. However, the reliability model used in this thesis could easily be replaced by a more sophisticated one, since the scheduling algorithm presented in section 3.7 is independent of the reliability model used. Also, we do not distinguish between different kinds of failures, hence the reason for why a node failed is not important in our model. The reason of failures could however be taken into account in a more sophisticated reliability model.

Furthermore, we assume the replicated tasks are deterministic and always produces the correct result for a given input. We can thereby define reliability as the probability of producing a result and not lose any data, instead of the probability of producing the correct result as done in [12, 29, 45]. However, as mentioned before, this can be solved by extending our model with consensus algorithms such as majority voting. Since we do not gain any extra useful information by doing so, we have chosen to leave this to future work.

Moreover, the time to process each input is assumed to be less than the MTBF of the used nodes. Otherwise all nodes could die before a result is produced, despite always having at least one replica operational. By further extending the model with checkpointing, described in section 2.4.1, one could allow longer processing times.

We also assume the case where all nodes are within the same cluster, connected with high-bandwidth and low-latency links, and all nodes are reachable from every other node. We therefore assume that the time it takes to send a message between two nodes is the same for all nodes. Therefore, it is assumed that the replication time described in section 3.6.3.2 does not depend on between which nodes the replication is done.

By the use of these limitations we avoid unnecessary complexity in form of checkpointing and consensus. The impact of this will be discussed later in chapter 5.

3.4 System model

In this section, we describe the system and application models employed in this work.

3.4.1 Computational environment

In this report we assume that all resources are within the same cluster, with low-latency connections between them. The resources consist of heterogeneous hardware with high-bandwidth, low-latency redundant links between them. Due to its heterogeneous properties, nodes cannot be assumed to have the same failure rate.

An example of four interconnected nodes is shown in fig. 3.1. We refer to a computational resource as a *node*. Due to the lack of access to a cluster consisting of heterogeneous hardware components, the cluster used for experiments consisted of homogeneous hardware components, and varying behavior in terms of failure is simulated in the experiments. This if further described in chapter 4.

Since all nodes are interconnected, we have redundant paths which results in that all nodes are directly reachable even if a node fails. In Appendix A.2 and Appendix A.3, a system of one replicated actor is shown before and after a node failure.

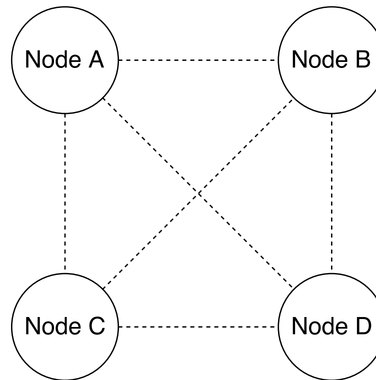

Figure 3.1: Computational environment, interconnected nodes

3.4.2 Storage of data

Information such as how many replicas are currently executing, and on which nodes they are running, must be globally available and accessible from every node in the system.

If information is only stored locally on a node, that information is lost in case that the node fails. In case the information is written to a persistent storage, it may not be lost, but that information will be unavailable until node has been restarted.

Instead of storing information locally, a remote database can be used. In this case, a single-point-of-failure is introduced, resulting in a system where the reliability is no more than the reliability of the database. If a remote database is used, this must be taken into account in the reliability model.

To achieve a fault tolerant redundant storage of information, a Distributed Hash Table (DHT) can be used. Using DHT efficiently avoids having a single-point-of-failure.

In Calvin, the framework in which the model was implemented and evaluated, there are two storage alternatives. A DHT implementation called Kademlia, see [63] for further information, and a proxy storage. Calvin and its storing alternatives are further described in section 3.10.1.

When storing data using DHT, the data is first stored locally and later flushed, i.e. sent to other nodes. When a task is replicated to another node, the replication must only be considered successful if the storage is properly updated and the information shared to other nodes. Otherwise, the replication may only be partially reflected in the storage and use of this data will be useless.

3.4.3 Application model

The fault tolerant framework presented in this paper is general and may be used in various contexts. However, it is particularly of value for long running applications and services running in dynamic environments where a predefined level of reliability must be met. Long running applications are particularly vulnerable to failure because they usually require many resources and usually must produce

precise results [5].

In this paper, the application used consists of a producer, a task T and a consumer, and can be modelled as shown in fig. 3.2. The task T shown in the figure could for example be a service for which we require a certain reliability. In contrast to [1,13,14,18,27,30,32], no assumptions are made about the execution time of the application. However, the time to process each input is assumed to be less than the MTBF of the nodes, as described in section 3.3. While our model is general, it is particularly beneficial for long running applications and services, as they in large-scale distributed environments are required to stay operational even in the case of unpredictable failures [28].

A typical example of a long running application is data stream processing, where a continuous stream of data is sent to a service which performs some computations on the data, and sends a result to a receiving process. The execution time is unknown, and in case of a failure, data may be lost. By sending the data to several replicas, redundancy is achieved and reliability thereby increased as the probability of at least one replica produces a result is increased.

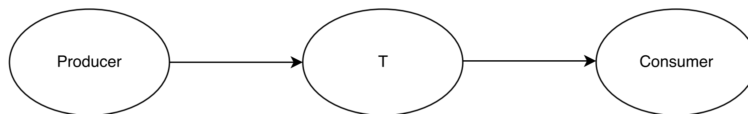

Figure 3.2: An application model where a producer transmits data to a task T, which transforms the data, and sends the result to a consumer. T may be seen as a task or service running in a cluster and requiring a certain level of reliability.

3.4.4 Replication scheme

As previously mentioned, reliability will be ensured by the use of replication. More specifically, active replication is used, where each replica receives the same input, performs the same computations, and produces the same output. Figure 3.3 shows how the application in fig. 3.2 looks after replicating task T four times. It is also possible to have several tasks replicated, this scenario is shown in Appendix A.1.

In contrast to the case with one *primary* task and several *back-up* replicas, as described in section 2.4.3, we adapt a fan-in fan-out model, where all replicas both receive the same input, but also all transmit its result to the consumer.

After replicating, the various replicas may not be synchronized. But, since assuming only deterministic calculations are done by the tasks replicated, given the same input they will all produce the same result, even if not synchronized. However, since the receiver receive a result from each replica, our model allows for easy extension to also include a majority decision at the consumer to determine whether or not the correct result was received.

The process of replicating a task in Calvin, the framework in which our model was implemented, is further described in section 3.10.1.4.

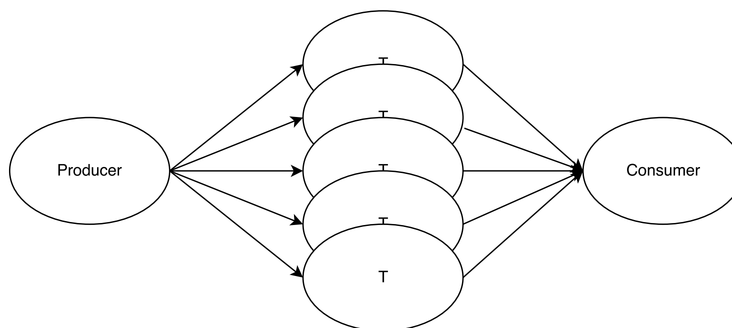

Figure 3.3: An application model where a task T has been replicated four times.

3.4.5 Fault model

In this paper, we adapt the *fail-stop* fault model, which is described in section 2.2.2.2 and commonly used when presenting fault tolerance techniques [8]. Nodes in the system have one of two states: *operational* or *failed*. If a node fails, all running tasks on that node are dead. Furthermore, after a node has died, it will be restarted. However, the tasks that were being executed before it died will not be restarted when the node is restarted. The reason for why a node died is irrelevant.

Like [29], only system related failures are considered, assuming failures depends on the nodes, not the jobs running on them and the computations they perform. Furthermore, unlike the reliability models presented in [11,32,43], the network is assumed fully reliable, and thus link failures are not accounted for.

3.4.5.1 Failure distribution

A commonly used assumption, also adapted in this paper, is that failures are statistically independent. Furthermore, failures are assumed to follow a Poisson process, as this seems to be widely accepted in the research community [58].

While most models using the Poisson process to model the probability of failure assume constant failure rates, we assume they are constant only for a given period of time. By monitoring the system resources and registering failure times, the *MTBF* of a node can by dynamically adapted. This is accomplished by using the time of the three latest failures to determine the *MTBF* as of definition 2.2. This is further described in section 3.9.2.

Since we assume the *MTBF* for a node is constant during a period of time, failures can be modeled using a Poisson process. Using a resource's *MTBF*, the reliability for that resource can using eq. (2.1) be expressed as

$$R(t) = e^{-t/MTBF} \tag{3.1}$$

Equation (3.1) expresses the probability that a given resource will work for a time t. Correspondingly, the probability that a resource will fail during a time interval of length t is

$$F(t) = 1 - e^{-t/MTBF} \tag{3.2}$$

In our case t represents the time it takes to detect a node failure and create a new replica, and is further discussed in section 3.6.3.

In order to use eq. (2.4), the *MTBF* of a node must be available, and to calculate a node's *MTBF* at least two failures must have occurred. However, the system still need to be able to calculate the reliability for a node, even if the node has not yet failed. Therefore, a default value for the *MTBF* will be used in the experiments when no failure data is available for a node, or if it has not yet failed twice. In a real situation, the default value could be based on past failure data for resources of similar type. The default value used in the experiments is further described in section 4.1.

As time goes and nodes fail, and the time of each failure is stored, the system will get more precise values for the *MTBF* for the nodes in the system. By using the latest three failure times, one can adapt the *MTBF* as nodes start failing more or less often.

3.5 Monitoring

Monitoring the system is crucial for achieving an entirely dynamic system, both in which the required reliability is met over time, but also where no more resources than necessary are being used.

3.5.1 Heartbeats

The ability to detect node failures is crucial for knowing when a new replica is needed in order to keep the required reliability. Furthermore, since the reliability model presented in section 3.6 is based on *MTBF* of the nodes in the system, it is based on the assumption that node failures are detectable.

To achieve this, a heartbeat system is used, where nodes periodically send UDP messages, called heartbeats, every t_{h} seconds. The heartbeats contain a node identifier and are sent to all the other nodes in the system. Nodes are considered operational as long as heartbeats are received from them, and if no heartbeat is received from a node for t_{timeout} seconds, the node is assumed dead.

In the experiments conducted, see chapter 4, t_{h} was set to 0.2 seconds, and t_{timeout} to 0.5 seconds. These values should be modified in a real setting. For the experiments, a high-bandwidth low-latency cluster was used, why we could use a high heartbeat frequency and a low timeout time.

As further described in section 3.6.3.1, these values affect the reliability model, and thereby the number of replicas needed. A higher frequency and lower timeout means a shorter time in which the system is in a vulnerable state, and therefore a lower number of replicas may be needed to reach the required reliability. On the other hand, higher frequency means increased network traffic in the system as more heartbeats are being sent, although the size of a heartbeat message is low, only 126 Bytes.

3.5.2 Monitoring system reliability

In order to ensure that the optimal number of replicas is used over time, as the properties of the system vary, the system and its running applications or services must be periodically monitored.

In our model, the reliability of the running applications are monitored periodically, and if the reliability is not met, appropriate actions are taken, which is further described in section 3.7.

On the other hand, if the required reliability is met, more replicas than necessary may be used to reach that reliability level. One must therefore minimize the number of replicas by moving the replicas to more reliable nodes, and deleting replicas on less reliable nodes as long as the required reliability is still met. The algorithm for optimizing is further described in section 3.7.1.

In the experiments, all applications were monitored and the greedy scheduling algorithm followed by the optimization algorithm, both presented in section 3.7, were run every five seconds.

3.6 Reliability model

In this section the reliability model used in our experiments is presented. Note that the scheduling algorithm presented in section 3.7 is not dependent on the reliability model used.

3.6.1 Definitions

In this paper, we use the following definitions of reliability:

Definition 3.1. *The reliability of a process is the probability that the resource on which the process is running is functioning during the time of execution.*

For long running applications or services, where a replication scheme is used and new replicas dynamically being created as old ones fail, the reliability can be defined as [23]:

Definition 3.2. *The reliability of a process, with n task replicas, is the probability that at least one replica is always operational. This can be expressed as the probability that not all replicas fail during the time from that a task replica dies, until a new replica is operational.*

3.6.2 Expressing reliability

Using replication and the reliability definition defined in definition 3.2, the reliability of a task T with n replicas, is the probability that at least one replica is successful during a time t, where t is the time from that a replica fails, until a new replica is operational. This corresponds to at least one replica surviving time t, i.e. not all replicas fail, and can be expressed as

$$R_T(t) = 1 - \prod_{k=1}^{n} f_k(t) \tag{3.3}$$

where $f_k(t)$ is the probability that replica k fails during time t. Since assuming tasks themselves do not fail unless the resources they use fail, the reliability of a task is dependent on the reliability of the resources it uses, not the number of replicas. Considering only node failures, the reliability of a task T with n replicas placed on m different nodes, can using eq. (3.2) be expressed as

$$R_T(t) = 1 - \prod_{k=1}^{m} F_k(t) = 1 - \prod_{k=1}^{m} (1 - e^{-t/MTBF_k}) \tag{3.4}$$

where $MTBF_k$ is the *mean-time-between-failure* for node k. Using this model, reliability is only increased through replication if the replicas are scheduled on separate nodes. Furthermore, eq. (3.4) is based on the assumption that failures of nodes are statistically independent, which is a commonly used assumption as described in section 2.2.3.

Given a time t, a required reliability level R_{req}, and assuming the replicas are running on m separate nodes, we get

$$R_T(t) = 1 - \prod_{k=1}^{m} F_k(t) \geq R_{\text{req}} \tag{3.5}$$

which must be fulfilled by the system. Assuming that failure rates differ among resources, fulfilling eq. (3.5) is a scheduling problem, since the number of replicas needed is dependent on which resources the replicas are running on.

The scheduling problem to fulfill a reliability R_{req} refers to selecting m nodes on which to place m replicas such as the reliability level of the task, expressed in eq. (3.4), exceeds R_{req}.

3.6.3 Expressing time t

The time t used in eq. (3.4) is the time it takes from that a failure happened, until a new replica is operational, and consists of the time it takes to detect the failure, and the time it takes to create a new replica. The time t can therefore be expressed as

$$t = t_{\text{d}} + t_{\text{R}} \tag{3.6}$$

where t_{d} is the time to detect that a node has failed, and t_{R} is the time it takes to create a new replica. A new replica is created by sending a replication request to a node currently holding a replica, including a third node on which the new replica is to be created. This process is further described in section 3.6.3.2.

3.6.3.1 Node failure detection time, t_{d}

As described in section 3.5.1, heartbeats are periodically sent to all nodes in the system every t_{h} seconds, and if no heartbeat is received from a node for t_{timeout} seconds, it is assumed dead. Note that t_{h} must be lower than t_{timeout}.

The timeout time, t_{timeout} is only an upper bound for how long it takes to detect a node failure. A node can fail just before sending a heartbeat, or directly

after, which will affect the actual time the node has been dead before it is detected. This corresponds to the best and worst case scenario.

If a node A dies precisely before sending heartbeat H_{k} to node B, the last received heartbeat from A was $H_{\mathrm{k-1}}$. t_{timeout} seconds after $H_{\mathrm{k-1}}$ was received, node B will assume A has failed. Node A will then have been dead for $t_{\mathrm{timeout}} - t_{\mathrm{h}}$ seconds.

In the latter case, node A dies directly after sending a heartbeat H_{k}, and t_{timeout} seconds later node B will assume node A has failed. Node A has in this case been dead for t_{timeout} seconds when node B assumes it has failed.

Assuming that the probability of a node dying just before and directly after sending a heartbeat is the same, we get a theoretical average detection time of $t_{\mathrm{timeout}} - \frac{t_{\mathrm{h}}}{2}$ seconds.

An example of the best and worst case scenario is shown in Appendix A.7 and Appendix A.8.

In eq. (3.6) we will use

$$t_{\mathrm{d}} = t_{\mathrm{timeout}} \tag{3.7}$$

which is the worst case scenario. This result in the calculated reliability always being equal to or lower than the actual reliability, since the time to detect the failure always is t_{timeout} or lower.

3.6.3.2 Replication request time, t_R

The time t_{R} in eq. (3.6) is the time from that a replication request is sent, until a new replica is operational and a response is received. It consists of the time it takes to send a replication request to a node, for that node to replicate its replica to another node and send a response.

To know which node to send the replication request to, one must first find out on which nodes current replicas are running. Thereafter, since the reliability is dependent on which nodes the replicas are running on, not solely on the number of replicas, one must also find a node which does not already have a replica and include this node in the request. The time t_{R} can be expressed as in eq. (3.8) where $t_{\mathrm{query\ storage}}$ is the time it takes to find which nodes currently holding a replica and which nodes do not, $t_{\mathrm{replicate\ msg}}$ is the time to send a replication message to another node, t_{r} is the time to replicate the task to another node, and t_{response} is the time to send a response to the requesting node. The process of sending a replication request is shown in fig. 3.4. The part of querying the storage is excluded from this figure.

$$t_R = t_{\mathrm{query\ storage}} + t_{\mathrm{replicate\ msg}} + t_{\mathrm{r}} + t_{\mathrm{response}} \tag{3.8}$$

As mentioned, the time t_{r} is the time it takes to replicate a task. Depending on the state of the task to replicate, it is likely to be a significant part of the total time t_{R}. The time to replicate a task consists of the time to get the state of the task, serialize it, transmit it to the other node, de-serialize it, and create a new identical replica using that state, and send the response.

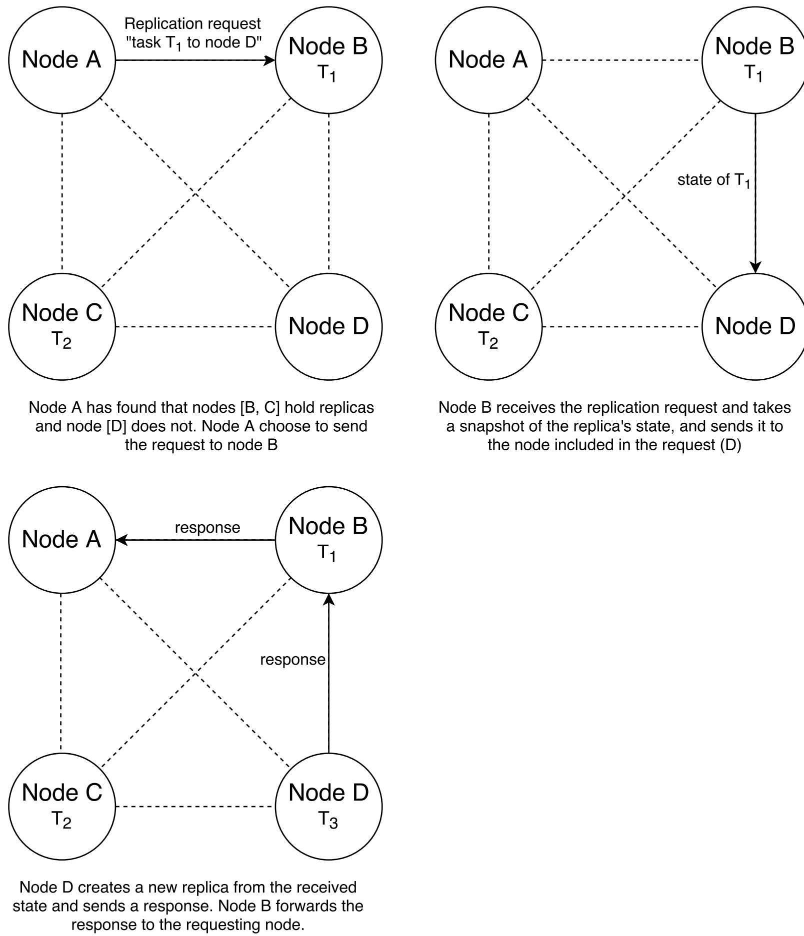

Figure 3.4: The process of node A asking node B to replicate its replica T_1 to node C, and receiving a response.

$$t_{\mathrm{r}} = t_{\text{get state}} + t_{\text{serialize state}} + t_{\text{transmit state}} + t_{\text{de-serialize state}} + t_{\text{create new}} + t_{\text{send reply}} \tag{3.9}$$

In our model, the time t_{R} is measured and stored every time a replication takes place. Since t_{r} depends on the size of the state, the time to replicate different tasks are likely to vary. Therefore, the time t_{R} is stored per task type. Consequently, different types of tasks may require different number of replicas to reach the same reliability, as the reliability is dependent on the time t_{R}. Figure 3.5 shows how the number of replicas needed to reach a reliability above 0.99, using nodes with a *MTBF* of 10 seconds, vary depending on the time t.

Other factors affect the time t_{R}. The node to which the replication request is sent may die before the replication is finished. The sender of the request assumes the receiving node has failed if no response is received within a timeout time.

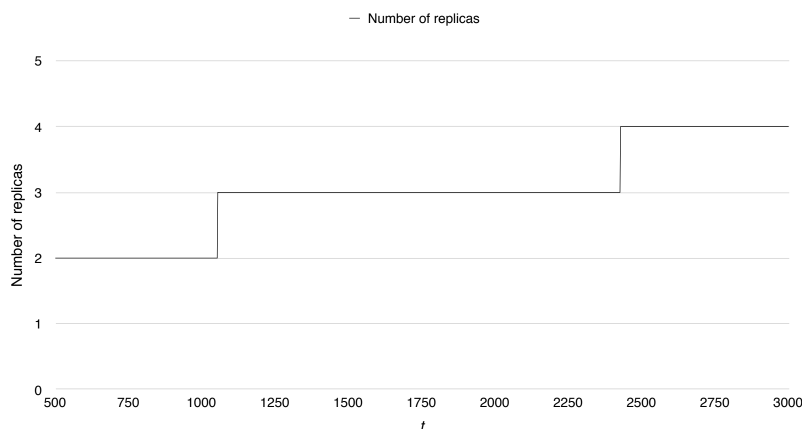

Figure 3.5: The number of replicas needed to reach a reliabilty above 0.99, using nodes with a *MTBF* of 10 seconds, depending on the time t.

In this case, a new node is selected and asked to replicate its replica. When this happens, the time t_R, will be relatively high, as time was wasted sending the request to the first node.

This situation can theoretically continue until there are no more nodes holding a replica. One could argue that this should be taken into account in the reliability model, as the time t_R affects the reliability. Also the node being selected to run algorithm 1, described in section 3.8, may die, in case a new node is selected. These situations are further described in section 3.8. The probability of these situations happening depend on how often nodes fail, and how long time the replication takes. If nodes fail very frequently, there is a higher risk of a node failing before finishing the replication.

Instead of including the probability of these situations in the reliability model, a log-logistic distribution is first fitted to the previously registered times t_R. From the fitted distribution, the 95th percentile value is used as the value for t_R in eq. (3.6). This means in 95 percent of the cases, the time t_R will actually be lower than the value used in eq. (3.6). The reason for choosing log-logistic distribution is mentioned in experiment 4.5.1.

Finally, since assuming that all nodes are within the same cluster and that the latency between all nodes are low, we assume the time t_R is not dependent on between which nodes the replication takes place.

3.7 Scheduling algorithm

To fulfill eq. (3.5) using the minimum number of replicas, one could simply place a new replica on the most reliable node available, until the required level is reached. A greedy scheduling algorithm doing this, similar to one presented in [23], and which fulfills eq. (3.5) is shown below:

Algorithm 1 Greedy scheduling algorithm to fulfill a given reliability

Precondition: R_{req} is the reliability to fulfill.

1: **procedure** GREEDY SCHEDULING ALGORITHM
2: *current nodes* $\leftarrow$ current nodes
3: *operational nodes* $\leftarrow$ operational nodes
4: *available nodes* $\leftarrow$ *operational nodes* $\setminus$ *current nodes*
5: SORT(*available nodes*, *descending*)
6: **while** RELIABILITY(*current nodes*) $\leq R_{\text{req}}$ **do**
7: *node* $\leftarrow$ *available nodes.pop* ▷ take the most reliable node
8: *replica* $\leftarrow$ *new replica*
9: PLACE REPLICA ON NODE(*replica*, *node*)
10: *current nodes.add*(*node*)
11: **end while**
12: **end procedure**

Function 1 Sorts the given nodes in order of preference

Returns the given nodes in sorted order, with the most or least (depending on *order*) preferred nodes first. We only consider the nodes' reliability when sorting, but it can easily be changed to consider other parameters as well.

Input: *nodes* is the list to sort, and *order* is the order to sort in (ascending or descending).

1: **function** SORT(*nodes*, *order*)
2: SORT AFTER RELIABILITY(*nodes*, *order*)
3: **return** *nodes*
4: **end function**

In our model, a new replica is placed on the most reliable node until the required reliability level is reached. However, we do not account for whether or not the selected node has enough available resources for the new replica, hence we do not consider system load. The scheduling algorithm is however independent of both the selection of nodes, function 1, and the calculation of the current reliability, function 2, why these two are two separate function calls.

In our model, the reliability function uses eq. (3.4) for determining the reliability, and the sort function simply sorts available nodes after reliability, but they could both easily be replaced by more sophisticated models, e.g. which also consider nodes' loads when sorting them.

The algorithm is run when deploying an application, when a failure is detected as further described in section 3.8, and also periodically in combination with an optimization algorithm, described next in section 3.7.1.

Function 2 Calculates the reliability of the given nodes

Calculates the reliability of the nodes currently holding a replica. This function could be changed to use a different reliability model.

Precondition: The time t used in this function depends on the task type and is described in section 3.6.3.

Input: *current nodes* is a list of nodes currently holding a replica.

1: **function** RELIABILITY(*current nodes*)
2: $n \leftarrow current\ nodes.length$
3: return $1 - \prod_{k=1}^{n} (1 - R_{\text{current nodes[k]}}(t))$
4: **end function**

3.7.1 Optimization

Algorithm 1 creates the minimal number of new replicas needed to reach the required reliability. However, it does not ensure that the minimum number of replicas is used over time. As more reliable nodes become available, fewer replicas could be used while still meeting the required reliability. Therefore, algorithm 2 is periodically run, as part of the application monitoring as described in section 3.5.2. By first running the greedy scheduling algorithm, and then the optimization algorithm, we ensure both that the required reliability is met, and that it is met while using the minimum number of replicas possible.

The first part of algorithm 2 makes sure the most reliable nodes are used, by moving the replicas from the least reliable nodes used, to the most reliable nodes available, as long as more reliable nodes are available. To avoid putting the system in a vulnerable state, moving a replica involves first creating a new replica, and later deleting the old one. If deleting before creating the new replica, one risk not having enough replicas until the new replica is operational. Replicating first avoids this problem.

The second part make sure no unnecessary replica is used, by deleting the replica on the least reliable node as long as the required reliability is still met. However, moving replicas to more reliable nodes may be unnecessary overhead if it does not affect the overall number of replicas needed. For example, moving from a node with a reliability of 0.999 to a node with reliability 0.9991 will perhaps not significantly increase the overall reliability, and if it does not result in another replica can be deleted, it may be unnecessary. Algorithm 2 does not account for this.

3.8 Handling node failure

In the case of a node failure, one must first determine whether or not any replicas were running on the failed node, and if so, determine whether or not new replicas are needed to still fulfill the required reliability level. This is accomplished by

Algorithm 2 Optimization algorithm

```
1: procedure MOVE TO MOST RELIABLE NODES
2:     current nodes ← current nodes
3:     operational nodes ← operational nodes
4:     available nodes ← operational nodes \ current nodes
5:
6:     SORT(available nodes, descending)
7:     most reliable ← available nodes.pop
8:     SORT(current nodes, ascending)
9:     least reliable ← current nodes.pop
10:
11:    while RELIABILITY(most_reliable) > RELIABILITY(least reliable) do
12:        replica ←REPLICA AT(least reliable)
13:        REPLICATE TO(replica, most reliable)
14:        DELETE REPLICA FROM NODE(replica, least reliable)
15:
16:        current nodes.add(most reliable)
17:        available nodes.add(least reliable)
18:        least reliable ← current nodes.pop
19:        most reliable ← available nodes.pop
20:    end while
21: end procedure
22:
23: procedure DELETE UNNECESSARY REPLICAS
24:    least reliable ← current nodes.pop
25:    while RELIABILITY(current nodes) ≥ R_req do
26:        replica ←REPLICA AT(least reliable)
27:        DELETE REPLICA FROM NODE(replica, least reliable)
28:        least reliable ← current nodes.pop
29:    end while
30: end procedure
```

running algorithm 1. However, when a node fails, all other nodes in the system will detect its failure, since no node will receive heartbeats from it. If all other nodes run algorithm 1, one could end up in a situation where every remaining node creates a new replica. While this will increase the reliability, it may result in an unnecessarily high number of new replicas being created, thus the optimal number of replicas will no longer be used.

To cope with this, a selection process must take place in which a single node is selected, which will be responsible for deciding what actions to take. Since assuming a dynamic system where nodes fail and new nodes are introduced, this selection must be done every time a node dies. Furthermore, the selected node may also die before it manages to create any new replica. Therefore, all nodes will send a *lost node* request to the selected node. When finished, the selected node sends a reply back to all nodes it received a *lost node* message from. If the sending nodes do not receive a reply within a specific time, they assume the selected node died, and a new selection process will begin.

The algorithm is shown below:

Algorithm 3 Handling a failed node

1: $operational\ nodes \leftarrow$ operational nodes
2: $lost\ node\ id \leftarrow$ lost node id
3: SORT AFTER ID($operational\ nodes$)
4: **do**
5: $\quad node \leftarrow operational\ nodes.pop$
6: $\quad$ LOST NODE REQUEST($node$, $lost\ node\ id$)
7: $\quad$ wait for reply
8: $\quad reply \leftarrow reply\ from\ node$
9: **while** $reply\ is\ not\ successful\ OR\ request\ timeout$

An example is presented in fig. 3.6. In the situation shown in the figure, there are four connected nodes, A, B, C and D, and the node C has just failed. When the other nodes (A, B and D) detect the failure they will select the node with the highest ID, in this case node A, and inform it that node C has failed. Node A will then determine which actions to take.

3.8.1 Selecting node

As described, the node responsible for handling the detected failure is the node with highest ID among the nodes still being operational.

If the selected node determines a new replica is needed, it will send a replication request to a node currently holding a replica. If the selected node is one of the nodes holding a replica, there is no need for a replication request, since the node could simply replicate its own replica. One could therefore avoid having to send a replication request if one selected a node currently holding a replica in the selection process. However, to find out which nodes currently hold a replica, one must query the storage used, as described in section 3.6.3.2. Whether or not

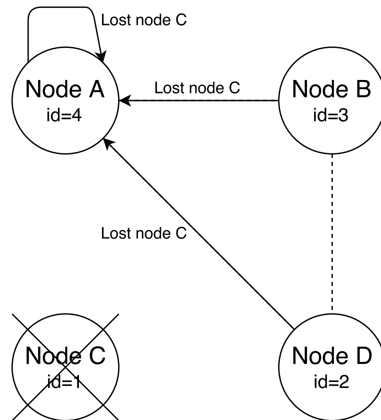

Figure 3.6: Four interconnected nodes after nodes A, B and D have detected the failure of node C.

a centralized database is used, or DHT as used in Calvin where our model was implemented, this would result in every node detecting the failure querying the storage, and this approach is not scalable. Therefore, the selected node is any of the nodes still operational, and solely the selected node will query the storage for the information needed to handle the failure.

3.9 Self-adapting

Adaption refers to changing the behavior as the state of the system changes. For a distributed system, resources must be continuously monitored in order to adapt to changing behavior [28].

To ensure a certain level of reliability, the framework must ensure that the current state of the system is taken into account when determining the number of replicas needed. Furthermore, the system and the running jobs must be continuously monitored. If the reliability of a running application task decreases below the required level, more replicas must be created. On the other hand, if the reliability increases, the number of replicas should be decreased in order not to waste resources, as long as doing so does not result in a reliability lower than the required one.

When monitoring the system, all parameters used in the reliability model must be monitored. In our case, we consider only the nodes' *MTBF*, the time to detect node failures, and the time it takes to create a new replica. Therefore, all nodes' failure times, and the time to create new replicas are registered and stored. The time to detect failure is as mentioned in section 3.6.3.1 static.

3.9.1 Adapting time t

Since using all registered times t_R to fit the log-logistic distribution, and selecting the 95th percentile value as described in section 3.6.3.2, the value of t_R used in eq. (3.6) will depend on all previous registered values. Therefore, if the replication time starts increasing, e.g. during higher system load, the value used in the reliability model will also increase, thereby adapting to changing behavior.

3.9.2 Adapting nodes' *MTBF*

As described in section 3.4.5.1, only the three latest failure times for a node are used to determine its *MTBF*. This allows for adapting a node's *MTBF* in case it starts failing more or less often.

However, if a node starts dying more often, the model needs three new failure times before the *MTBF* has been update to correctly reflect its new behavior. This is a clear limitation to our model. Instead, the past failure times could be used together with some machine-learning algorithm or regression analysis, to predict when a node is about to fail more or less often, and adapt its *MTBF* before it actually happens.

3.10 Implementation

For implementing our model, we used the actor based application environment *Calvin*. Below follows a brief description of the main parts of the Calvin framework, for further info we refer to [31].

3.10.1 Calvin

Calvin is an actor-based application environment for light-weight IoT applications, and is written in Python. It was developed by *Ericsson* and made open source in the summer of 2015. While Calvin is an application environment for IoT applications, it suites well for implementation of our model.

The main features of Calvin are the use of runtimes (containers) in which actors (application tasks) run. Since the runtimes separate the hardware from the software, it is possible to use Calvin on different platforms. In the developing phase of new applications the developer does not need to consider on which operating systems the application should run.

3.10.1.1 Storage

In Calvin there are two ways of storing data, either by the Kademlia implementation of a DHT or by the use of a proxy storage.

Using Kademlia, data is stored both locally and remotely, by distributing the data to other nodes in the system. The data is therefore available despite one of the storing nodes fails. One of the configurable parameters used in Kademlia is k, representing the maximum size of a bucket-list. The value of k should be chosen

such that the probability of k nodes failing within an hour is very low [63]. DHT will not be further explained in this thesis, instead we refer to [63] for further information about DHT and Kademlia especially.

When using the proxy storage, one runtime acts as a storage node, and all data is sent to that runtime. All requests for data are also sent to the runtime acting as proxy storage. However, the proxy storage introduces a single-point-of-failure, since all data is lost if that runtime fails.

3.10.1.2 Actor model

An application in Calvin consist of a set of connected *actors*. An actor usually represents part of a device, service or computation. Actors communicate by sending data, called tokens, on their out-ports to other actors in-ports.

Multiple actors may send data to another actor's in-port, and each of an actor's out-ports may have several receivers. For each receiver of outgoing messages, or sender of incoming messages, there is a separate queue of tokens to process or to send.

The runtime, see section 3.10.1.3, in which an actor is executing is responsible for the communication between runtimes and between actors.

The state of an actor is needed when replicating an actor. In Calvin, the state of an actor consists mainly of actor type, in-port and out-port connections, and for each port a queue with data to process or to send, and read and write positions for the incoming and outgoing queues. The queues are the major part of the message size.

The actor model corresponds well to the kind of applications and tasks described in section 3.4.3, as they perform deterministic calculations.

3.10.1.3 Runtimes

The Calvin framework use a concept of *runtimes*. A mesh of connected runtimes makes up the distributed execution environment on which one can deploy an application. A runtime is a self-managed container for application actors and provides data transport between actors both within the same runtime and between different runtimes.

Each runtime has a *storage*. If DHT is used, all the information in the storage is first stored locally and later flushed, i.e. distributed in a multicast approach. If a proxy storage is used, the information is also stored locally at first, but later flushed to the proxy storage only.

In our model a runtime corresponds to a node.

3.10.1.4 Replication

The process of replicating an actor in Calvin starts with getting the state of the actor to replicate, serializing it and sending it to the runtime in which the new replica is to be created. The receiving runtime then de-serializes the state and uses it to create the replica and setup its port connections.

Part of creating the new replica is decoding the in- and out-ports' queues, which are part of the state. The decoding is CPU bound and affects the total replication time, as shown in experiment 4.5.3.

3.10.2 Our contributions to Calvin

Here follows a description of the most important parts of our changes and new features in Calvin.

3.10.2.1 Fan-in connectivity

We extended the Calvin framework to allow a fan-out/fan-in connectivity model where actors can have multiple producers (fan-in) and multiple consumers (fan-out). Previously, only multiple receivers were allowed, but with our changes an actor's in-port can receive from multiple senders.

3.10.2.2 Actor replication

Previously, the framework allowed for dynamically migrating an actor from one runtime to another. We extended this functionality by allowing dynamic replication of actors.

3.10.2.3 Node Resource Reporter

To share various kinds of resources between runtimes, e.g. CPU usage as used in experiment 4.5.8, we created a resource reporter actor. When a runtime is created, it automatically creates a resource reporting actor. The actor periodically reports the node's CPU usage to all connected runtimes.

3.10.2.4 Heartbeat Actor

The heartbeat system described in section 3.5.1, was implemented using actors. Each runtime creates a heartbeat actor when it is started. The heartbeat actor listens for heartbeats as well as periodically sends heartbeats to other runtimes.

Unlike the resource manager, heartbeat messages are sent using UDP instead of TCP. With TCP, an initial handshake it used to setup the communication channel. For a heartbeat system, this is unnecessary overhead.

For every heartbeat received, a timeout of t_{timeout} seconds is set. At timeout, the runtime from which the heartbeat was sent is assumed dead. On the other hand, when a heartbeat is received, all timeouts related to that runtime are canceled. When a timeout happens, algorithm 3 is deployed and new replicas are created if needed to reach the required reliability, see section 3.8.

3.10.2.5 Application monitor

Each runtime also periodically monitors the applications deployed to it, to ensure the required reliability is fulfilled by running algorithm 1 and creating new replicas if needed. Also the optimization algorithm, is run as part of this monitoring.

Chapter 4

Evaluation

In order to validate our model and to show some of the possibilities, we carried out a set of experiments. The goal of the experiments was to show the three main properties of our model. Firstly, that it dynamically ensures the required level of reliability is met, despite the event of node failures, by dynamically creating new replicas when old ones are lost. Secondly, that it uses the optimal number of replicas by choosing the most reliable nodes, and also removing unnecessary ones. Thirdly, that it adapts to changing properties of the system.

We also measured the replication request time t_R to find the best fitting distribution for these values, and also the replication time t_r for varying state sizes. Finally, we measured the actual time for detecting failures to demonstrate that the values lie between the best and worst case described in section 3.5.1.

4.1 Values used in the experiments

As described previously, heartbeats were sent every 200 ms in the experiments, and nodes were considered dead if no heartbeat was received for 500 ms. Furthermore, the optimization algorithm was run every 5 seconds.

As described in section 3.6, the reliability model is based on nodes' *MTBF* and the time t_R. In order to determine the *MTBF* for a node, at least two failure times must have been registered. Before a node has failed twice, a default value is used. The default *MTBF* used in the experiments was 10 seconds.

In addition to the *MTBF*, a default value for t_R must be used before any such times have been registered. The default value used in the experiments was 2 seconds.

The state size of the actor replicated in the experiments was 1901 bytes.

4.2 Computational environment

In the experiments, a cluster consisting of 6 inter-connected servers with homogeneous hardware components was used. A laptop was also used when measuring the replication time t_r. The specification of the servers and the laptop are described in section 4.2.2.1 and section 4.2.2.2. The various servers will be referred to as *Gru*, *Dave*, *Kevin*, *Mark*, *Jerry* and *Tim*.

As mentioned in chapter 3, the model was implemented in the IoT application environment Calvin. In the experiments, one or two, depending on experiment, Calvin runtimes were started on five of the six servers. These runtimes were used to represent actual nodes, and were periodically killed and restarted to simulate node failure. These five servers are referred to as the unstable servers.

On the sixth server, *Gru*, two runtimes were started which were never killed. One of these stable runtimes was used as a proxy storage, further described in section 4.3, and the reason for having the other stable runtime is described in section 4.4. The proxy storage was started in every experiment and the other stable runtime was started in every experiment expect the one where we measured the actual time for detecting failures.

In the experiments, the runtimes are referred to as nodes, and when referring to creating nodes with a specific *MTBF*, we refer to the runtimes created on the 5 unstable servers.

4.2.1 Simulating node failure

As mentioned, the servers used in the experiments had homogeneous hardware components, but varying failure rates were simulated by killing and restarting the runtimes with varying rates.

The process of killing and restarting is shown in algorithm 4. If a node was to have a *MTBF* of 15 seconds, the time between failures followed a normal distribution with mean 15 and standard deviation of 1.

Since the experiments only ran for less than 30 minutes, the actual *MTBF* for a given node may during the duration of a experiment be either lower or higher than the given *MTBF*. The reason for using values from a normal distribution with a mean equal to the given *MTBF*, instead of a fixed time, was to simulate a more realistic situation. If all runtimes would have been started at the same time, with the same fixed time to sleep before killing them, they would all have been killed at the same time. Furthermore, if we would have waited some time before starting each runtime, two runtimes would perhaps never have died at the same time. By picking values from a normal distribution, two or more runtimes may fail at the same time during the time of the experiments.

Algorithm 4 Simulating node failures

1: **while** *true* **do**
2: START RUNTIME
3: $t_s \leftarrow \mathcal{N}(MTBF, 1)$
4: SLEEP t_s
5: KILL RUNTIME
6: **end while**

4.2.2 Node specification

4.2.2.1 Server specification

The servers used in the experiments all had a Intel(R) Xeon(R) CPU E5-2420 v2 of 2.20 GHz and 24 GB RAM. Furthermore, they were all connected with a 1000 Mb/s link with a latency of less than 0.2 ms. The OS installed on the servers was Ubuntu 14.04 LTS.

4.2.2.2 Laptop specification

The laptop used in the experiment where the replication time was measured, was a Dell Vostro v131 with a Intel i5, 2.3 GHz processor, 4 GB 1333 MHz RAM. It was equipped with a SSD with a read speed of 540 MB/s and a write speed of 520 MB/s. Furthermore, the installed OS was the same as for the servers, Ubuntu 14.04 LTS.

4.3 Storage used

As mentioned in section 3.10.1.1, Calvin supports use of both the Kademlia implementation of DHT, and a proxy storage. The Kademlia parameter k as described in section 3.10.1.1 should be chosen such that the probability of n nodes failing within an hour is very low. Since we wanted to be able run an experiment within an hour, and simulate multiple node failures during that time, all nodes would experience failure within an hour. Furthermore, Kademlia periodically re-distributes responsibility of storing keys, as nodes leave and join the network [63]. But due to killing and restarting runtimes as frequently as we did in the experiments, keys were found to not being re-distributed correctly.

Therefore, we experienced the DHT to be very unstable, and not suitable for our experiments, where nodes were frequently killed and restarted. Therefore, the proxy storage was used in the experiments. This introduced a single-point-of-failure, but the runtime acting as a proxy storage was never killed during the experiments. In a more realistic setting, nodes are unlikely to fail as often as they do in our experiments, and the use of DHT could in such a case be more suitable.

4.4 Application used in experiments

The Calvin application used in the experiments was of the simplest form, consisting only of a producing actor, a service actor and a consuming actor. The producing actor produced integer numbers and sent those to the service actor which simply forwarded them to the consuming actor, which printed the values to standard out.

A predefined level of reliability was required only for the service actor, and it was therefore the only one being replicated. As neither the producing nor the consuming actor were replicated, the whole application would have died in case the runtime on which they were deployed died. Therefore, the consumer

and producer were both placed on the stable runtime, mentioned in section 4.2. The service actor however, was only deployed to the unstable runtimes, which were periodically killed and restarted. When measuring the reliability in the experiments, we refer to the reliability of the service actor.

The computational environment, with an application consisting of a consumer, two service replicas, and a consumer, is shown in fig. 4.1.

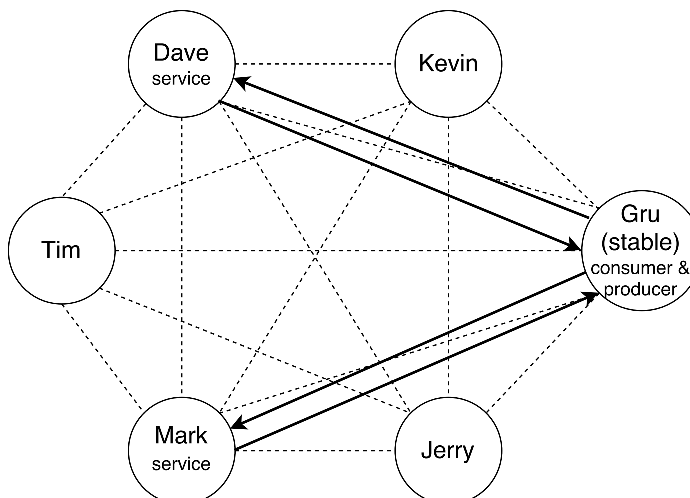

Figure 4.1: The computational environment used in the experiments. In this example, there is a producer and consumer on Gru, and two replicas of the service actor located on Dave and Mark.

4.5 Experiments

In experiment 4.5.1 and 4.5.4 to 4.5.8 the replication request time used by the model, t_R was logged. Recall that the model calculates and uses the 95th percentile value, see section 3.6.3.2.

4.5.1 Measurement of time t_R

As mentioned in section 3.6.3, t is the time it takes from that a node on which a replica is running dies, until a new replica is operational. It depends on the time it takes to detect that the node died, and the time it takes to create a new replica.

The time for detecting that a node has died, t_d, was as described in section 3.6.3.1 static and set to the upper bound $t_{timeout}$, which in the experiments were 0.5 seconds. While t_d is static, the time for the replication request, t_R, varies. In order to model the variation correctly, an experiment was conducted where times were registered and later used to find the best fitting distribution.

The experiment was conducted in two settings. In both settings, two runtimes were started on each of the non-stable servers. In the first experiment, each

runtime was given a *MTBF* of 30 seconds, and in the other a *MTBF* of 10 seconds. Furthermore, the required reliability was in the first experiment set to 0.999999, and 0.99995 in the second. Each time t_R was then registered for the duration of the experiment.

The experiment was conducted in two settings with varying failure rates since with higher failure rate, there is a higher risk of the node handling the lost node dies before it finishes, or a node dies before the replication is finished, as described in section 3.6.3.2. With more stable nodes, i.e. higher *MTBF*, there is a lower risk of this happening. Therefore, the replication times are likely to have less variation in this case. The goal was to find the distribution, among those tested, which best fitted the two data-sets.

Results

The first experiment ran for 230 minutes, and 2000 times were registered, and the second experiment ran for 80 minutes, and 2000 times were registered.

The distributions tested were *beta*, *birnbaumsaunders*, *exponential*, *extreme value*, *gamma generalized pareto*, *inversegaussian*, *logistic*, *logalogistic*, *lognormal*, *nakagami*, *normal*, *rayleigh*, *rician*, *tlocationscale*, *weibull*. Matlab was used to determine which distribution best fitted the data, and the Bayesian information criterion was used to determine how good of a fit a distribution was.

Figure 4.2 shows the four best fitted distributions for the first experiment, with nodes' *MTBF* set to 30 seconds, and fig. 4.3 shows the four best fitted distributions for the second experiment. The figures only show registered values less than 500 ms, for aesthetic reasons. The average values, and number of values above 100 and 500 ms in both experiments are shown in table table 4.1.

When nodes fail more often, there is a higher probability of nodes failing before it finishes to handle a *lost node* request, or to replicate one of its replicas. As described in section 3.6.3.2, this results in a higher time t_R. This is why the average is higher, and there are more times above 100 ms and 500 ms, and the 95th percentile value, in the experiment with *MTBF* of 10 seconds, than the experiment with *MTBF* of 30 seconds.

The log-logistic distribution was found to best fit both data-sets, and is therefore the one used in our model, as described in section 3.6.3.1.

MTBF	average t_R	number of values above 100 ms	number of values above 500 ms	95th percentile
30 s	86.6	311	15	145.1
10 s	107.3	597	34	194.5

Table 4.1: Average time t_R, number of times above 100 ms and 500 ms, and the 95th percentile value for *MTBF* 30 and 10 in experiment 4.5.1

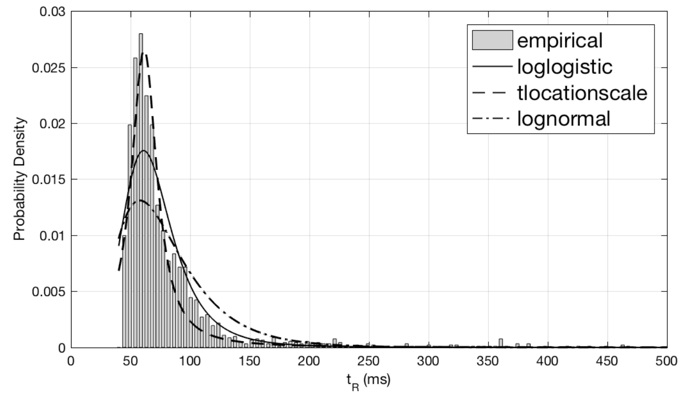

Figure 4.2: Result for experiment 4.5.1. The four best fitted distributions for the failure time data in the first experiment with *MTBF* of nodes set to 30 seconds.

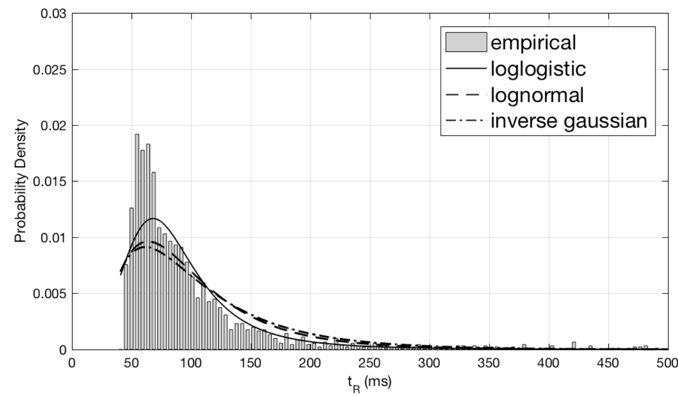

Figure 4.3: Result for experiment 4.5.1. The four best fitted distributions for the failure time data in the second experiment with *MTBF* of nodes set to 10 seconds.

4.5.2 Measurement of node failure detection time

As described in section 3.5, heartbeats are used to determine whether or not another node is operational. If no heartbeat is received from a node within t_{timeout} seconds, it is assumed dead.

In order to measure the time it takes to detect a failed node, an experiment was conducted in which two runtimes were started on the same server. One of the runtimes was periodically killed and restarted. Before killing the runtime, the current time was logged, and on the other node, the time was logged when the node was assumed dead, which in the experiments was 500 ms after the last received heartbeat. The failing node was killed and restarted 1800 times.

Results

The average of the 1800 measured times was 333.5 ms, the minimum was 302.1 ms and the maximum value was 499.8 ms.

All values lie between the theoretical minimum and maximum values of 300 and 500 ms, corresponding to the best and worse case scenarios described in section 3.6.3.1. The average however it is clearly below the theoretical average of 400 ms. The standard deviation of the times registered was 28.8.

4.5.3 Measurement of replication time t_r

The time it takes to replicate a task, depends on the size of the task state, which have to be sent to the node where the new replica should be created. In Calvin, the state of a task corresponds to the state of the actor representing the task, and consists mainly of the queues of incoming and outgoing data for the actor's in- and out-ports, and any local variables the actor has, as described in section 3.10.1.2.

In the following two experiments, two runtimes were used. The application was first deployed to one of these runtimes, and the service actor was replicated to the other runtime and the time measured. This was repeated for various sizes of the actor's state. The actor had only one out-port and no in-port, and in the first experiment the state of the actor was incrementally increased by increasing the size of a local variable, while in the second experiment the state was increased by increasing the size of its output queue. The state of the actor is only one part of the total replication message. The other part is however static, and its size during the experiment was 349 bytes.

Both experiments was conducted in two settings. In one, two servers were used with a runtime on each of them, while in the other, a laptop was used on which both the two runtimes were started. The specification of the servers and the laptop are found in section 4.2.2.1 and section 4.2.2.2. For each state size, the actor was replicated 10 times and the times measured.

While measuring the total time, t_r, several other times were measured along the way, namely the times to serialize the state, send the state to the other runtime, de-serialize it, create and start the new actor, and send the reply, corresponding to the times described in eq. (3.9).

When the replication was done between runtimes on the same laptop, and the queue size increased, the times were only measured for state sizes less than or equal to 400 MB. This was due to the replication taking too long. For a state size of 400 MB, each replication took nearly two hours, as shown below, and during the time of the experiment the laptop could not be used, to not affect the results.

For three state sizes, 1, 50 and 100 MB, we measured the replication time 100 times and calculated the standard deviation of these values.

The goal of the experiment was to see how the replication time vary depending on the state size, in which part of the replication process most time is spent, and finally how the replication times varied.

Results - total replication time

Figure 4.4 and 4.5 show how the total replication time vary depending on the state size when replicating between runtimes on different servers, and between runtimes on the same laptop respectively. As shown, the replication time is exponential, and the replication time was higher when increasing the queue size than when increasing the variable size. This is due to the extra time needed when creating the new actor, since all values in the queue have to be decoded.

As seen in fig. 4.5, when replicating using the laptop, there is a big increase in time when increasing the state size from 200 MB to 400 MB, by increasing the queue. This is due to running out of memory, and the process started using the laptop's SWAP memory, which significantly reduced the performance.

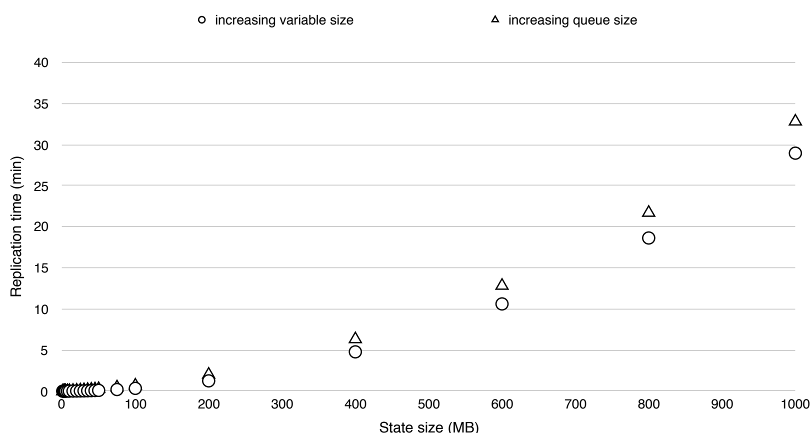

Figure 4.4: Result for experiment 4.5.3. The average replication time as a function of state size, where the state size was increased by increasing the size of one of the actor's variables, and the replication was done between two runtimes on different servers.

Results - replication time parts

Figure 4.6 and fig. 4.7 show how much of the total replication time was spent on the various parts of the replication process when increasing the variable size and queue size respectively, and when replicating between the two servers. As shown, the time to create the new replica takes a significantly larger part of the total time when increasing the queue size rather than the variable size. This is due to the time spent on decoding the queue's values as described previously.

Furthermore, significantly more time was spent on transmitting the state to the other node as the state size was increased. Hence for large state sizes, the total replication time depends mostly on the bandwidth of the links connecting the various nodes. For smaller state sizes, the replication time is more CPU bound, especially when increasing the queue size.

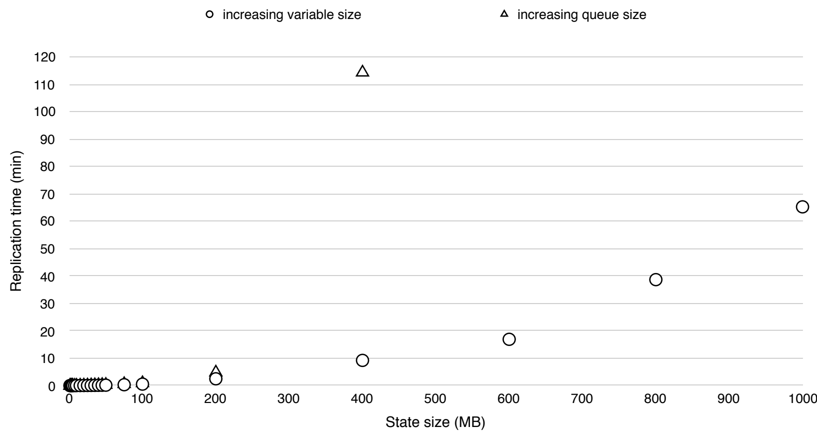

Figure 4.5: Result for experiment 4.5.3. The average replication time as a function of state size, where the state size was increased by increasing the size of the actor's port queue, and the replication was done between two runtimes on the same laptop.

Figure 4.8 and fig. 4.9 show how much of the total replication time was spent on the various parts of the replication process, when increasing the variable size and queue size respectively, and when the replication was done between two runtimes on the same laptop. Again, more time is spent on creating the new actor when increasing the queue size than when increasing the variable size.

Furthermore, the time spent on transmitting the state increases when increasing the state size. However, as shown in fig. 4.9, when replicating from laptop to laptop an actor with state size above 100 MB (when increasing the queue size), more time was spent on creating the actor. This, as described before, is due to running out of memory. In the experiment, this happened when the size was 200 MB or more, and when this happens, the process needs to use the laptop's swap memory when creating the new actor. The laptop's 1333 MHz ram memory has a speed of approximately 21 GB/s, which compared to the SSD speed of 540 MB/s explains the significantly drop in performance.

Results - replication time variance

For 1, 50 and 100 MB the replication time was measured 100 times. The variance of the replication times was very small. Table 4.2 and table 4.3 show the average replication time (μ), the standard deviation (σ), and the relative standard deviation (RSD), calculated as $\frac{\sigma}{\mu}$, for these three state sizes when increasing the variable's size, and the size of the queue respectively. The variance was as shown very low, except for the smaller state size when replication was done between runtimes on the same laptop.

However, in a real setting, the replication time will vary depending on the state of the system. When increasing the queue size, the replication time is more CPU bound than when increasing the size of a variable, and therefore the replication

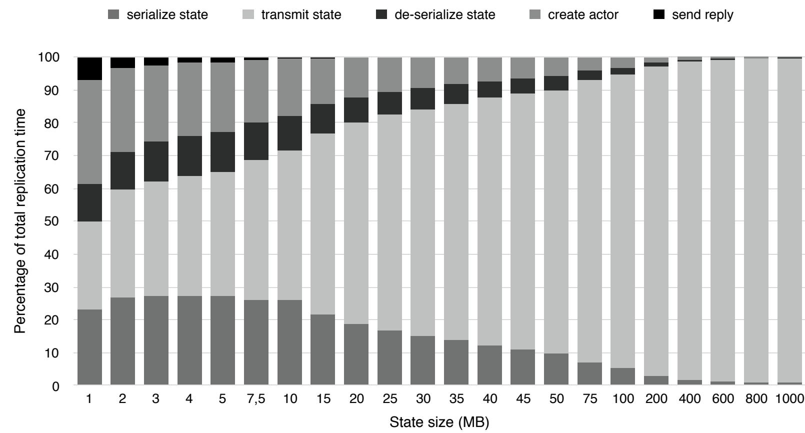

Figure 4.6: Result for experiment 4.5.3. Time, of the total replication time, spent on the different parts of the replication, when replicating from server to server, and increasing the size of one of the actor's variables.

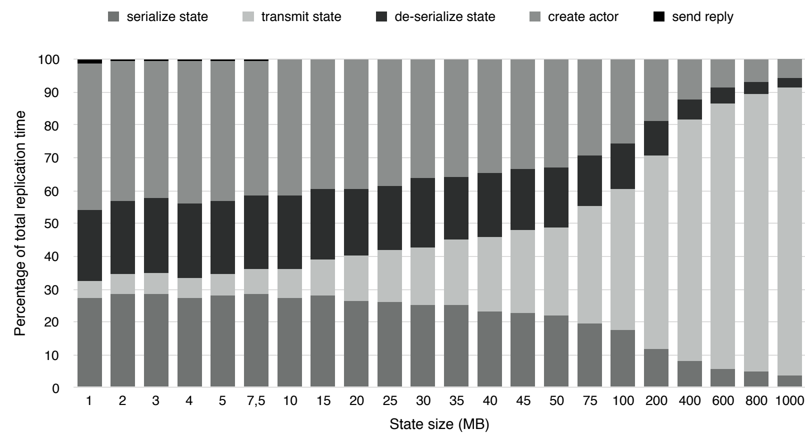

Figure 4.7: Result for experiment 4.5.3. Time, of the total replication time, spent on the different parts of the replication, when replicating from server to server, and increasing the size of the actor's port queue.

time is dependent on the current load of the two nodes. Furthermore, for larger state sizes when the transmission time is the largest part of the replication time, the total time will vary depending on the network load of the system.

4.5.4 Ensuring a certain reliability level

In this experiment, two runtimes were started on each of the unstable servers. Each runtime was given a *MTBF* of 20 seconds.

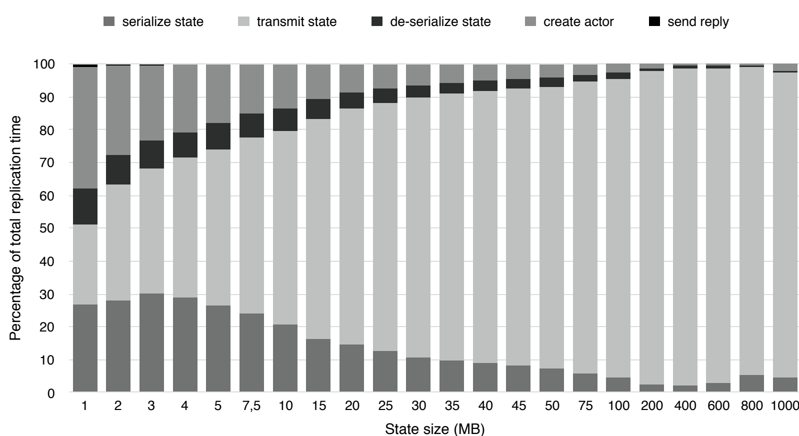

Figure 4.8: Result for experiment 4.5.3. Time, of the total replication time, spent on the different parts of the replication, when replicating from laptop to laptop, and increasing the size of one of the actor's variables.

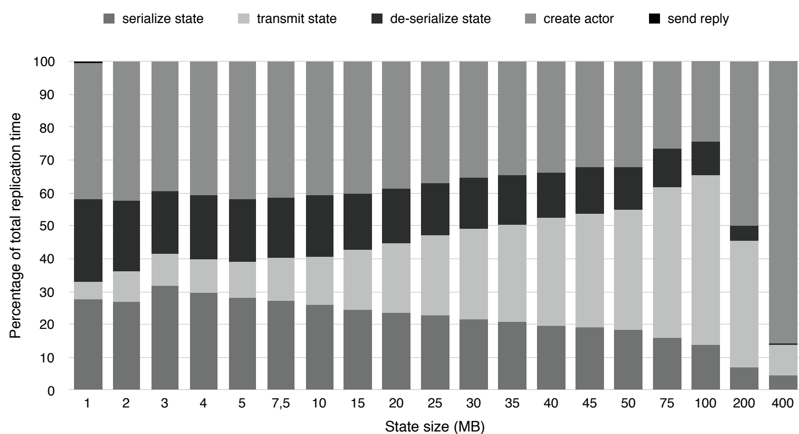

Figure 4.9: Result for experiment 4.5.3. Time, of the total replication time, spent on the different parts of the replication, when replicating from laptop to laptop, and increasing the size of the actor's port queue.

After the application was started, the reliability level of the nodes on which replicas were running was periodically measured. The required reliability was set to 0.98.

The experiment was considered successful if the momentary reliability, when dropped below the required, was restored to a level above the required.

replicating from/to	state size (MB)	μ (s)	σ (s)	RSD
server/server	1	0.041	0.001	0.024
server/server	50	5.523	0.056	0.010
server/server	100	20.177	0.385	0.019
laptop/laptop	1	0.087	0.020	0.229
laptop/laptop	50	10.617	1.210	0.114
laptop/laptop	100	34.019	1.412	0.041

Table 4.2: Result for experiment 4.5.3. Average replication time (μ), standard deviation (σ), and the relative standard deviation (RSD), for state sizes 1, 50 and 100 MB, where the state size was increased by increasing the size of one of the actor's variables.

replicating from/to	state size (MB)	μ (s)	σ (s)	RSD
server/server	1	0.251	0,008	0,032
server/server	50	16,547	0.256	0,015
server/server	100	42.555	0.414	0,010
laptop/laptop	1	0.348	0.043	0.124
laptop/laptop	50	19.075	0.299	0.016
laptop/laptop	100	50.893	1.066	0.021

Table 4.3: Result for experiment 4.5.3. Average replication time (μ), standard deviation (σ), and the relative standard deviation (RSD), for state sizes 1, 50 and 100 MB, where the state size was increased by increasing the size of the actor's port queue.

Results

The experiment ran for 5 minutes, and fig. 4.10 shows how the reliability vary during this period. The average reliability for the duration of the whole experiment was 0.996, clearly above the required level of 0.98. The average of the 95th percentile values for the replication time, t_R was during the experiment 83.9 ms.

As shown in fig. 4.10, the reliability was lower in the beginning of the experiment. This is due to the lack of failure data for nodes, therefore assuming the default value for the *MTBF* of 10 seconds. As the nodes starts failing, the system learns their actual *MTBF*, which in the experiment was 20 seconds. Every drop in reliability corresponds to a failure of one of the nodes on which a replica is executing. The experiment shows clearly that the momentary reliability level is restored to a level above the required level and the experiment was therefore considered successful.

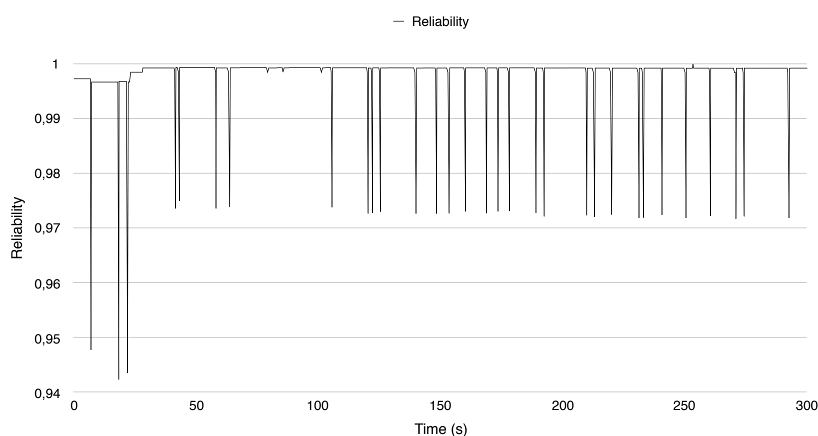

Figure 4.10: Result for experiment 4.5.4. Reliability over time in a system with failing nodes.

4.5.5 Optimal number of replicas

4.5.5.1 Choosing the most reliable

In this experiment, the two runtimes were started on each of the unstable servers. The *MTBF* for the various runtimes varied. The mean values given to the different nodes are presented in table 4.4.

node	*MTBF* (s)
$dave_1$	7.5
$dave_2$	7.5
tim_1	7.5
tim_2	7.5
$kevin_1$	15
$kevin_2$	15
$mark_1$	15
$mark_2$	40
$jerry_1$	40
$jerry_2$	40

Table 4.4: Mean-time-between-failures for the ten unstable runtimes in experiment 4.5.5.1.

After the application was started, the reliability level of the nodes on which replicas were running was periodically measured. The required level of reliability was set to 0.999.

The experiment was considered successful if, as the system learned the actual *MTBF* for the nodes, the system placed the replicas on the most reliable nodes.

Furthermore, for the experiment to be considered successful, the momentary reliability should lie above the required, and if below it should be restored to a level above the required.

Results

The experiment ran for 5 minutes, and fig. 4.11 shows the number of replicas over time, while figs. 4.12 to 4.14 show the number of replicas for each of the unstable nodes over time.

The average of the 95th percentile values for the replication time, t_R was during the experiment 99.1 ms.

As shown in the figures, the number of replicas decreased over time, from three replicas at the start of the experiment, to later only two. This is due to the system learning the nodes' *MTBF* and thereafter placing the replicas on the more reliable ones. This is shown in figs. 4.12 to 4.14, where the number of replicas per node is shown. It is clear that the more reliable nodes, $mark_2$, $jerry_1$, and $jerry_2$, were chosen more often than the less reliable ones. The experiment was therefore considered successful.

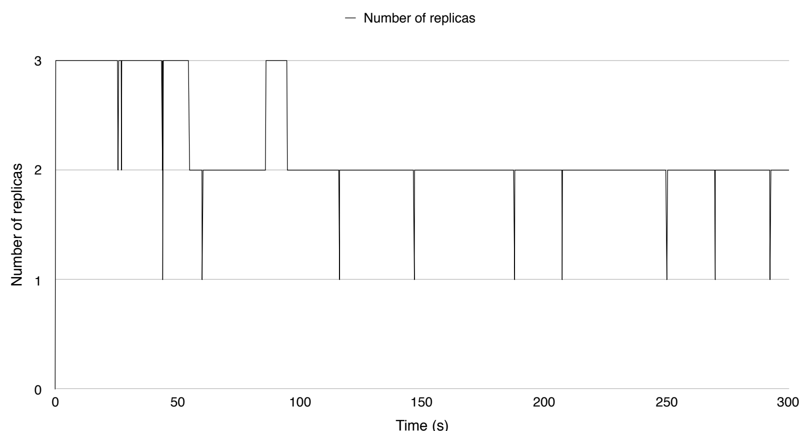

Figure 4.11: Result for experiment 4.5.5.1. Total number of replicas.

4.5.5.2 Optimal replicas

In order to show that the system automatically reduces the number of replicas in case more reliable nodes are available, we conducted an experiment in which five runtimes were started on the unstable servers. Two of these were given a *MTBF* of 25 seconds, while the other three were not killed. Since these three were not killed, they were not actually *unstable*, but the system used the default *MTBF* of 10 seconds for these nodes. Consequently, despite not failing, the system considered them less reliable than the two failing nodes with a *MTBF* of 25 seconds. The reason for not killing those three nodes were simply in order to show that the system automatically minimizes the number of replicas when the optimization algorithm

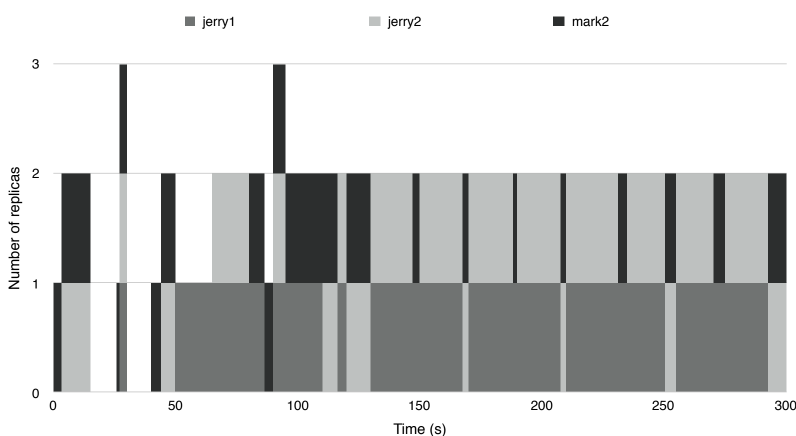

Figure 4.12: Result for experiment 4.5.5.1. Number of replicas per node, for the three nodes with a *MTBF* of 40 seconds.

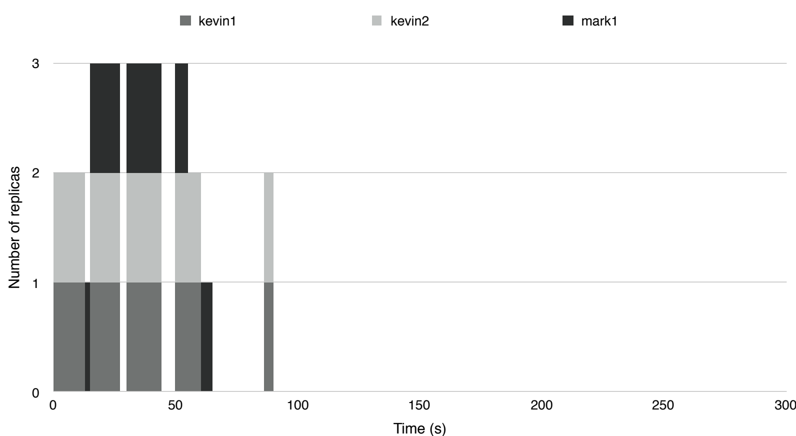

Figure 4.13: Result for experiment 4.5.5.1. Number of replicas per node, for the three nodes with a *MTBF* of 15 seconds.

runs, not only when a node fails. The required reliability in the experiment was set to 0.999. Finally, for aesthetic reasons, a delay of 5 seconds were used between killing and restarting nodes *kevin* and *jerry*.

Results

The experiment ran for 5 minutes. As mentioned, the default *MTBF* is 10 seconds when no failure data is known for a node. Therefore, before the reliable nodes, *kevin* and *jerry*, had failed twice, the system used the default *MTBF* of 10 seconds. After having failed twice, the system could learn their actual *MTBF* of 25 seconds. This is why there is a period during which none of the reliable nodes are given any replicas. After their second failure however, it is known that they are actually

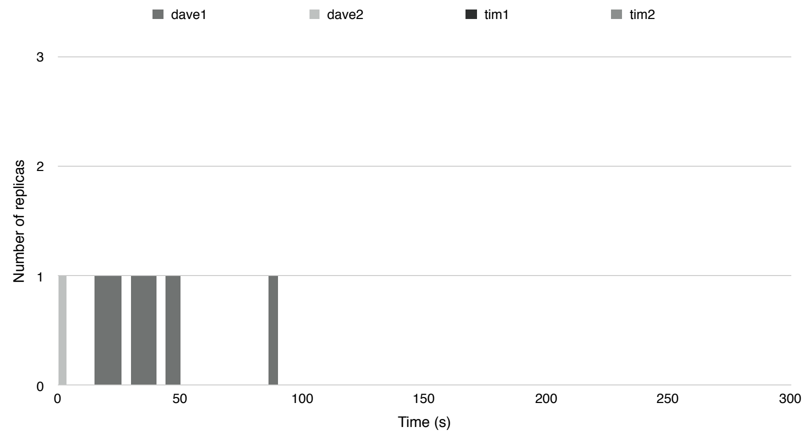

Figure 4.14: Result for experiment 4.5.5.1. Number of replicas per node, for the four nodes with a *MTBF* of 7.5 seconds.

more reliable than the other ones, why replicas are then moved to those nodes.

The average of the 95th percentile values for the replication time, t_R was during the experiment 67.7 ms. This resulted in that as long as the two reliable nodes were available, only two replicas were needed, but if one, or both of them died, three replicas were needed to reach the required reliability. This is shown in fig. 4.15, where the total number of replicas are shown.

Figures 4.16 and 4.17 shows the number of replicas per node and it seen that the system moves replicas from the less reliable nodes to the more reliable nodes as soon there is a more reliable node available. The experiment was therefore considered successful.

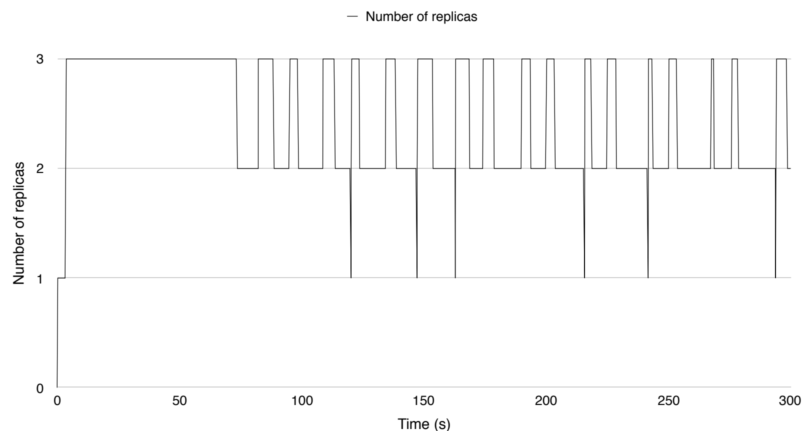

Figure 4.15: Result for experiment 4.5.5.2. Total number of replicas.

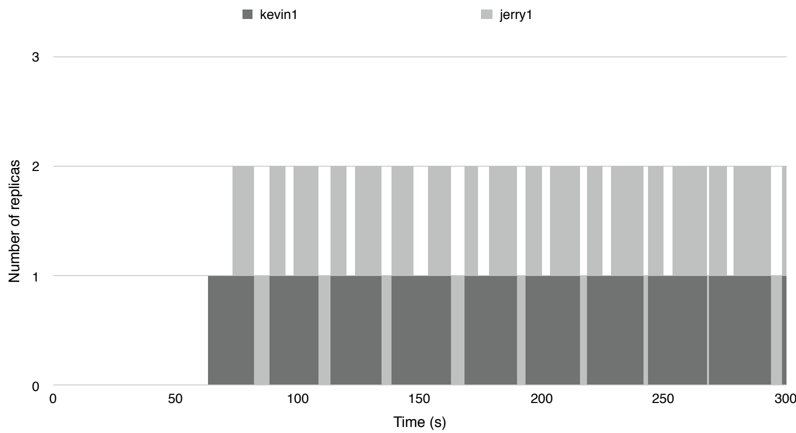

Figure 4.16: Result for experiment 4.5.5.2. Number of replicas per node, for the two nodes with a *MTBF* of 25 seconds.

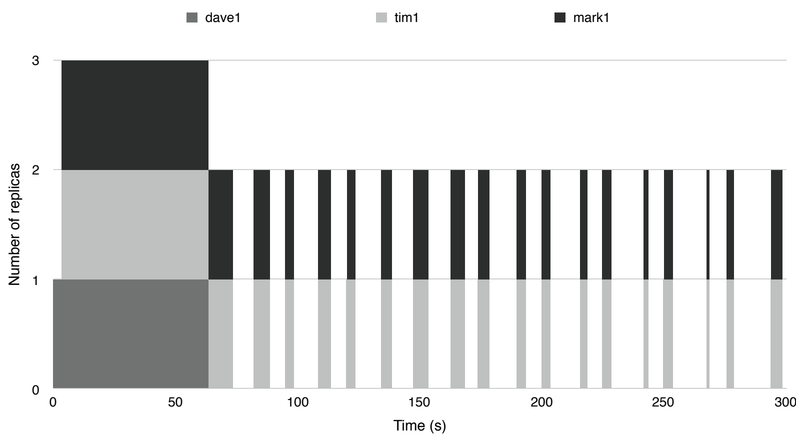

Figure 4.17: Result for experiment 4.5.5.2. Number of replicas per node, for the three nodes which were not killed, and therefore were assumed to have a *MTBF* of 10 seconds.

4.5.6 Self-adaptive reliability model

In this experiment, one runtime was started on each of the unstable servers. Each runtime was given a *MTBF* of 25 seconds. But, instead of picking times to sleep from a normal distribution as described in section 4.2.1, the time to sleep between failures were calculated by eq. (4.1). This means the time between failures varied between 5 and 45 seconds.

The required reliability was in this experiment set to 0.99999.

$$t_{\text{sleep}} = t_{\text{i}} + 20 * \sin\left(2\pi t_{\text{elapsed}}/300\right) \tag{4.1}$$

Where t_{i} is the numbers following a normal distribution with mean 25 and

standard deviation of 1, and t_{elapsed} is the total elapsed time for the experiment. Given $t_{\text{i}} = 25$, Appendix A.6 shows how the actual time to sleep varies over time.

The process of killing nodes therefore differed from algorithm 4. How nodes were killed and restarted in this experiment is shown in algorithm 5.

Algorithm 5 Simulating node failures

1: $t_{\text{start}} \leftarrow$ current time
2: **while** *true* **do**
3: START RUNTIME
4: $t_{\text{now}} \leftarrow$ current time
5: $t_{\text{elapsed}} \leftarrow t_{\text{start}} - t_{\text{now}}$
6: $t_{\text{s}} \leftarrow \mathcal{N}(MTBF, 1) + 20 * \sin(2\pi t_{\text{elapsed}}/300)$
7: SLEEP t_{S}
8: KILL RUNTIME
9: **end while**

The experiment was considered successful if the number of replicas used increased as the time between failures decreased for the various nodes, and if the number of replicas decreased as the time between failures increased.

Results

The experiment ran for 30 minutes. Figure 4.18 shows how the number of replicas varied over time, and fig. 4.19 shows how the calculated reliability for the various nodes varied over time. Since the number of replicas decreased as the *MTBF* for the various nodes increased, and the number of replicas increased as the *MTBF* for the nodes decreased the experiment was considered successful.

The average of the 95th percentile values for the replication time, t_{R} was during the experiment 209.2 ms. The reason for the higher value of t_{R}, compared to experiment 4.5.4, 4.5.5.1 and 4.5.5.2 is due to that when the *MTBF* for the nodes were at it lowest, approximately 5 seconds, there was a high probability of replication request failing, recall section 3.6.3.2.

4.5.7 Energy efficient for many applications

In this experiment, two runtimes were started on each of the unstable servers. The *MTBF* for the runtimes were the same as for experiment experiment 4.5.5.1, and shown in table 4.4.

Furthermore, in this experiment five applications were started instead of only one. The applications were identical, described in section 4.4, with a required reliability of 0.99999.

Since the replicas are placed on the most reliable nodes, each application will have its replicas on the same nodes, i.e. the most reliable ones. Therefore, theoretically, the total number of nodes having a replica will be no greater than the number of replicas for the application with the most number of replicas. The

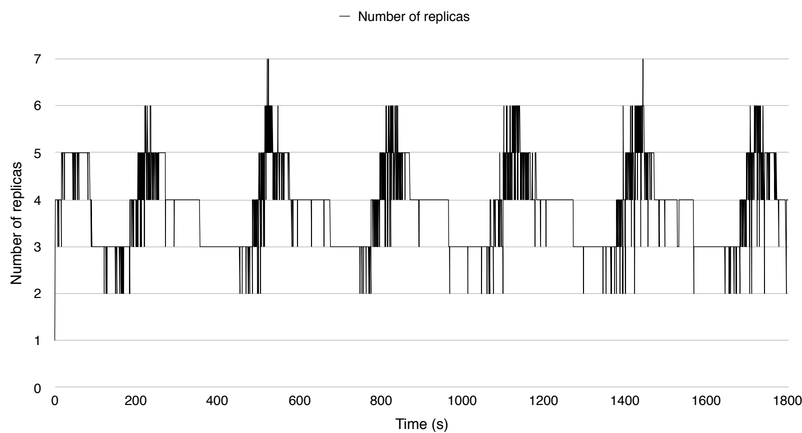

Figure 4.18: Result for experiment 4.5.6. Number of replicas over time.

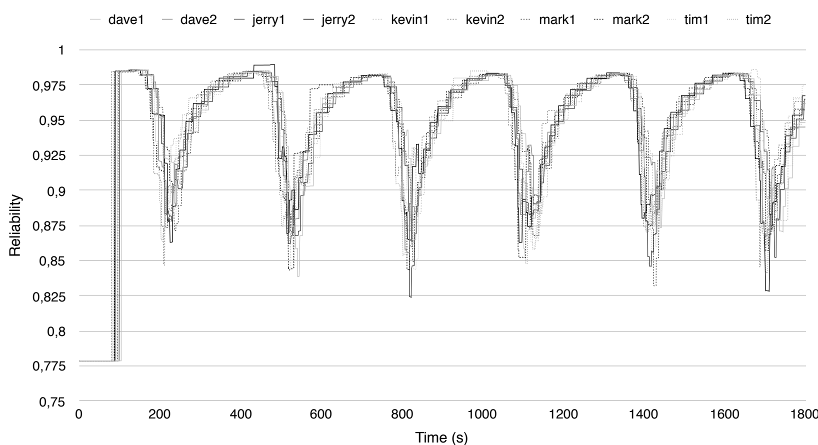

Figure 4.19: Result for experiment 4.5.6. The various nodes reliabilities over time.

experiment was therefore considered successful if the number of nodes used at any time were kept to a minimum.

Results

The experiment ran for 5 minutes, and fig. 4.20 shows the total number of nodes used over time. Figure 4.21, fig. 4.22, and fig. 4.23 show the number of replicas running at the various nodes. Since the model does not place two replicas of the same task on the same node, the nodes had at most 5 actors, one replica per application, at any time.

The average of the 95th percentile values for the replication time, t_R was during the experiment 263.4 ms. This resulted in that when the system had learned the

nodes' *MTBF*, only two replicas per application were needed to reach the required reliability, while three replicas were needed when the *MTBF* was unknown and the default value of 10 seconds was used. As shown in the figure, the number of nodes used varied between two to three most of the time, until after about two minutes, after which only two nodes were needed. Despite having 10 nodes, at most four were used to at any time. The experiment was therefore considered successfully.

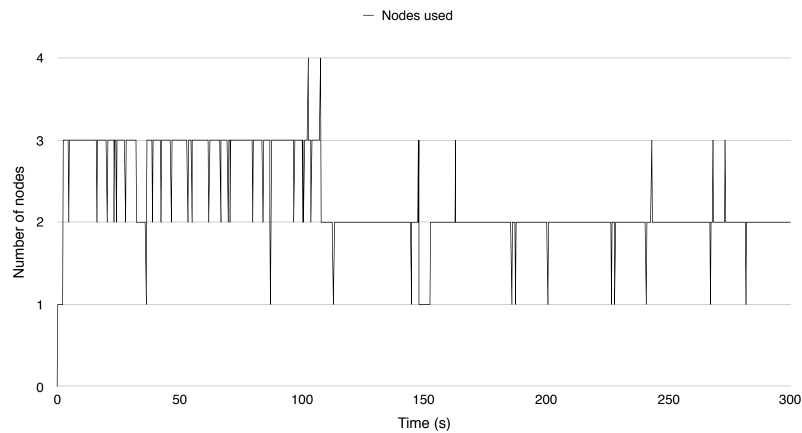

Figure 4.20: Result for experiment 4.5.7. Number of nodes used.

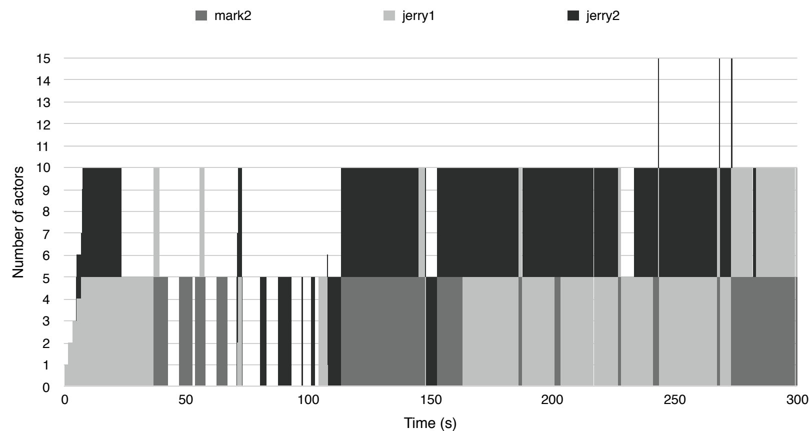

Figure 4.21: Result for experiment 4.5.7. Number of actors per node, for the three nodes with a *MTBF* of 40 seconds.

4.5.8 Considering nodes' load

Lastly, to show that the reliability model and selection of nodes to place replicas on are replaceable with more sophisticated algorithms, we conducted an experiment

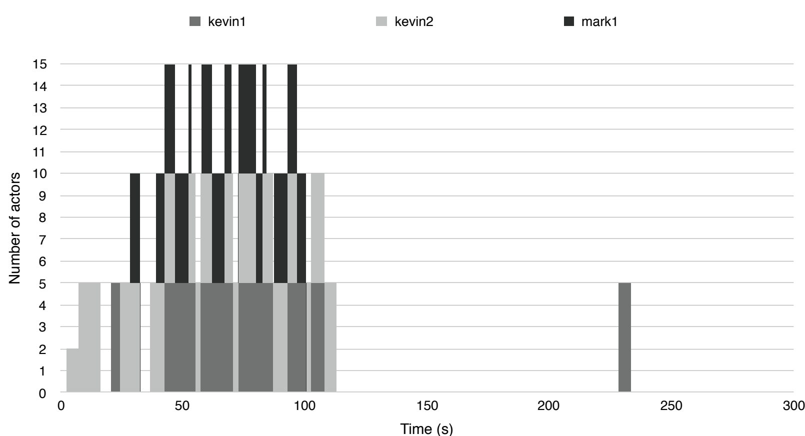

Figure 4.22: Result for experiment 4.5.7. Number of actors per node, for the three nodes with a *MTBF* of 15 seconds.

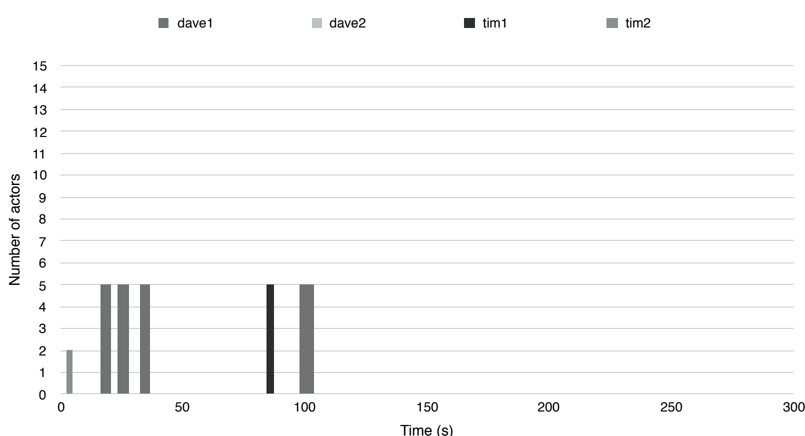

Figure 4.23: Result for experiment 4.5.7. Number of actors per node, for the three nodes with a *MTBF* of 7.5 seconds.

where the system avoids placing replicas on nodes with a CPU usage above 15 percent.

In this experiment, one runtime was started on each of the unstable servers. Three of the runtimes, *Tim*, *Mark*, and *Jerry*, had a *MTBF* of 10 seconds, and two, *Dave* and *Kevin*, had a *MTBF* of 40 seconds. Furthermore, the load on *Kevin* was increased after a while by starting dummy jobs simply to consume CPU resources, and was later decreased again. The required reliability in this experiment was set to 0.999.

The experiment was considered successful if, despite being one of the more reliable nodes, the system avoided to place replicas on *Kevin*, during the time its load was above the threshold of 15 percent.

Results

The experiment ran for 5 minutes. Figure 4.24 shows the CPU usage for the various nodes. Figure 4.25 and fig. 4.26 show the number of replicas for *Kevin* and the other nodes during the experiment.

The average of the 95th percentile values for the replication time, t_{R} was during the experiment 47.9 ms.

Figure 4.25 shows that no replica was placed on *Kevin* during the time its load exceeded the threshold of 15 percent and the experiment was therefore considered successful.

This experiment also shows that our model is general and key components such as calculating the reliability of a node as well as sorting the available nodes may be replaced.

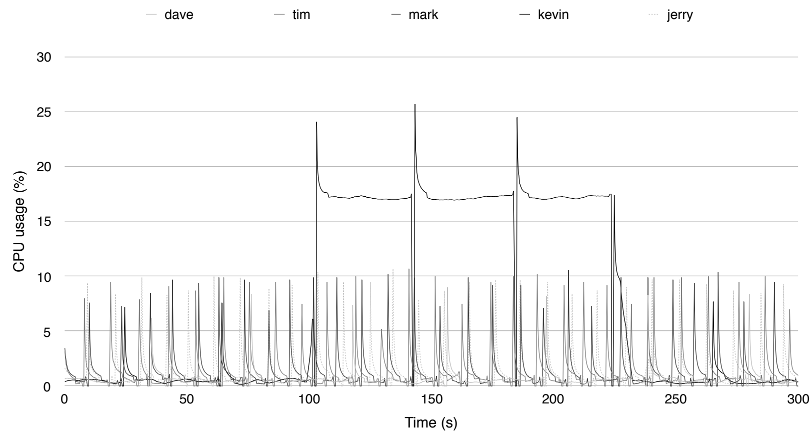

Figure 4.24: CPU usages in percent for the nodes used in experiment 4.5.8.

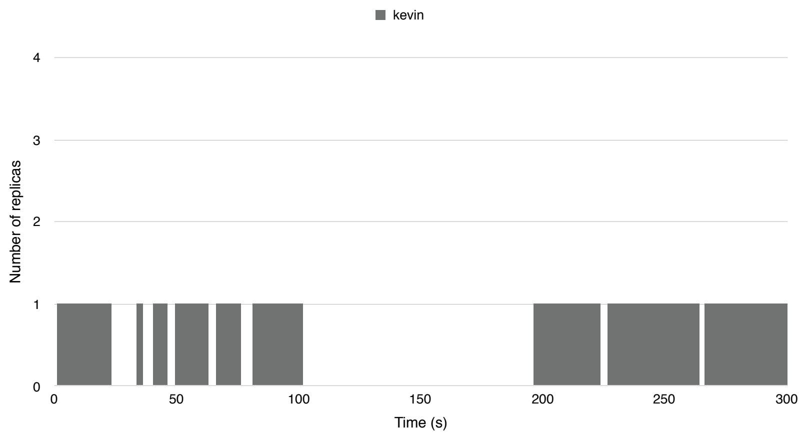

Figure 4.25: Result for experiment 4.5.8. The number of replicas over time for node *Kevin*.

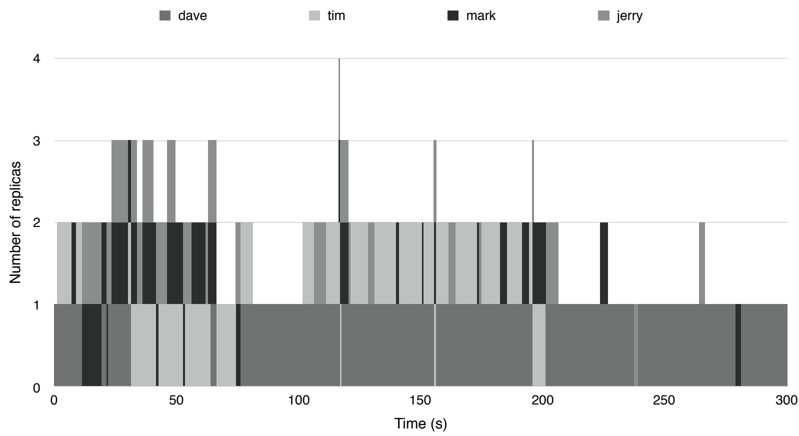

Figure 4.26: Result for experiment 4.5.8. The number of replicas per node for *Tim*, *Mark*, *Jerry*, and *Dave*.

Chapter 5
Discussion

The reliability model used in this thesis has several limitations. Firstly, it requires a node to have failed at least twice before knowing its *MTBF*, otherwise a default value is used. In case the default value is higher than the actual, it results in the calculated reliability of a node, using eq. (3.4), is higher than the actual, before the node has failed twice. Secondly, even though dynamically adapting a node's *MTBF* by using only the three latest failure times, it takes two new failures before the calculated *MTBF* correctly reflects the node's new behavior. Until then, the calculated *MTBF* will be higher, and thereby the calculated reliability will also be higher than the actual.

Since periodically monitoring the reliability, some regression analysis would be beneficial to try predict the reliability of nodes. This would also be beneficial if the system load varies in a periodic way, for instance with higher load during daytime and lower load during night. Since failures are more likely to occur when the system load is high, predicting system load would allow for taking preventive actions, e.g. creating new replicas.

Moreover, if a node fails frequently in the beginning of its lifetime, and then stabilizes, its *MTBF* will still be based on the registered failure times, resulting in it having a low calculated *MTBF*. Therefore, one could argue that the time since the last registered failure should be considered in the reliability model for a node.

Despite our reliability model having these limitations, algorithm 1 is designed to use the reliability model in a plugin fashion, namely by calling function 2. This allows for easily replacing the reliability model without changing the algorithm itself. Furthermore, determining on which node to place a new replica also involves an external function call. In our case, the most reliable node is selected, since the available nodes are sorted after reliability, see function 1. This function could also be replaced without changing the scheduling algorithm. One could for example exclude nodes where the load is above a given threshold, as shown in experiment 4.5.8.

Our model ensures that the reliability is higher than the required as long as we do not experience a node failure. After a node failure and before the system has recovered, by creating new replicas, the reliability is lower than the required. Our model does therefore not guarantee that the average reliability is higher than the required, it depends mainly on the relationship between the *MTBF* and the time it takes to replicate a task, but also on how much above and below the required reliability the momentary reliability is.

Furthermore, to recover from a failure, the failures must be detectable. This is achieved using the heartbeat system described in section 3.5.1. If the application consists of several tasks being replicated, as shown in Appendix A.1, the heartbeat system will no longer be necessary. Assume we have two tasks A and B, with replicas A_1, A_2 and B_1, B_2, where A_1 and A_2 both send their results to B_1 and B_2. Such a situation would allow to use a fault model which in addition to node failures also consider task and link failures. A task failure is detected if both replicas of B stop receiving results from A_1 or A_2. Furthermore, link failures are detected if for instance B_1 receives result from both A_1 and A_2 while B_2 only receives results from A_1. The link between A_2 and B_2 is then likely to have failed. This is however based on the assumption that the task A periodically sends results to B.

Moreover, the assumption of a fully reliable network with redundant paths is unlikely to reflect the real characteristics of distributed systems. In case of a link failure in a real setting, some nodes may not receive heartbeats from a node, while others will. The nodes not receiving heartbeats will then falsely assume the node has died. Such a situation may result in new replicas being created, perhaps more than necessary, and thereby putting an unnecessarily high demand on the system in terms of resource usage. Although, since the optimization algorithm in run periodically, the unnecessary replicas will eventually be removed.

We further assume that tasks always will produce the correct result. Since the receiving task will receive a result of each replica, some consensus algorithm such as *majority voting* or *k-modular redundancy* could be implemented at the receiver side. One could thereby adapt the broader Byzantine fault model instead of the fail-stop model. By doing so, the reliability model could also be updated from the probability of producing a result, to also consider the probability of producing the correct result.

With the primary objective being to ensure a certain level of reliability, and doing so using active replication, we require much more resources, which puts an extra burden on the system. The extra load on the system may affect tasks' execution times, thus decreasing tasks' performances. However, using active replication instead of checkpointing or rollback recovery, the extra time needed in case of failure is avoided, and in such cases the execution time may instead be improved by using replication. Furthermore, since we minimize the number of replicas used for each application, by placing replicas on the most reliable nodes, we consequently use the minimum number of nodes overall, as shown in experiment 4.5.7. Our model is therefore energy efficient, although the nodes not used are not shut down, nor put in idle mode.

In our experiments we replicated only one of the tasks in the used application. When handling a lost node, algorithm 1 must be run for every task which had a replica on the lost node. In a more realistic setting, there will likely be several different task replicas on the lost node. One must therefore decide in which order the tasks should be handled and new replicas created if needed. One possible solution is to prioritize all tasks, and start replicating the task with highest priority. However, the time t used in the reliability model is defined as the time it takes from that a failure occur until a new replica is operational. Therefore, when handling a lost node and start measuring the time, all tasks on the lost node will have the

same start time. This will result in that the task with the lowest priority will have the largest time t, since it will be handled last. Since this affects the reliability, the task with the lowest priority will thereby require the most number of replicas. Consequently, the task with lowest priority will require the most resources. However, in experiment 4.5.7 we had five applications, and consequently five tasks being replicated from the same node in case of node failures. This explains why the average replication time, mentioned in experiment 4.5.7 is higher than in most of the other experiments. In the experiment we simply replicated the tasks in random order.

As mentioned, the reliability is dependent on the replication time, which depends on the state of the task to replicate. The Calvin framework is only designed for light-weight IoT applications, thus not for applications where the actor state is several megabytes, and not optimized for replicating actors of such sizes. Despite this, our experiments show that it takes less than 40 minutes to replicate an actor with a state of 1 gigabyte between two servers in the same cluster. Assuming a *MTBF* of one year (31536000 s), and a time t of 1 hour (3600 s) for detecting and replicating, having only two replicas gives using eq. (3.4) a reliability of

$$
\begin{aligned}
R_T(3600) &= 1 - \prod_{k=1}^{2} F_k(3600) \\
&= 1 - \prod_{k=1}^{2} (1 - R(3600)) \\
&= 1 - (1 - e^{3600/31536000})^2 \\
&= 0.9999999863
\end{aligned}
$$

This means despite having a large time t, a high reliability can be achieved by only having two replicas.

Chapter 6

Future Work

To the best of our knowledge, our model is the first of its kind, due to its fully dynamic behavior. Furthermore, the implementation provides a means to carry out experiments. Thereby, there are many possibilities for future work.

Our model could be extended by considering more types of failures such as link failures, task failures, but also tasks not producing the correct result. In the latter case, some consensus algorithm could be implemented in order to hamper the risk of using incorrect results. Link failures could for instance be detected if the node responsible for handling the node failure awaits request from several nodes before running algorithm 1. If a node receives only a single *lost node* message, it is likely that the node is still alive but the link between the node and the sender of the *lost node* message has failed.

The process of selecting where to place new replicas could be extended to consider whether or not the selected node has capacity for the new replica. The application author could, using some metric, define the resources needed for a task replica, and the system administrator assign each node a metric of their capacities. These metrics could then be used to determine whether or not the available nodes have the required capacity for a replica.

Furthermore, as mentioned in chapter 5, our model is energy efficient as it will use the minimum number of nodes. However, this is based on the assumption that nodes always have enough capacity for the replicas placed on them. If the available capacity of nodes and the required capacity needed by the tasks are taken into account, the task placement decision is turned into a bin-packing problem for optimizing the number of nodes needed. By further extending the system with functionality for dynamically shutting down/starting nodes, unused nodes could be shut down to save energy and money.

Besides from considering additional types of failures and the capacity of nodes, our model would benefit greatly from applying statistical regression analysis on both nodes' failure behavior and system load, but also the times t_R. By predicting when the nodes are about to get overloaded, or the reliability of nodes are about to decrease, one could take preventative measures, e.g. move replicas to other nodes.

Our model is yet to be evaluated on highly unreliable system during extreme load. In such a case, due to the unreliability of the system, the number of replicas needed to ensure the required reliability level will increase. This will further increase the system load, thereby decreasing the reliability of the system even further, which may turn into a vicious circle.

Furthermore, since the replication time as mentioned depends on the number of other tasks being handled before, the reliability is affected by the number of other tasks running on the same node. This should either be considered in the reliability model, or the handling of failures must be changed. Instead of selecting a single node responsible for handling the failures, one node could be selected for each replica lost. The time t would thereby no longer be dependent on how many other tasks were running on the lost node.

If a single node is used to handle all lost replicas, one must consider in which order they are handled. One way would be to replicate the actors with the biggest states first in order to achieve a uniform distribution of the replication times, or to handle the highest priority tasks first as described in chapter 5. We leave the investigation of which algorithm that is optimal to future work.

Furthermore, our model does not guarantee that the average of the actual reliability is above the required, it only aims at keeping the momentary reliability higher than the required. This allows for the user specifying various reliability requirements. One such requirement could be that the time the system is in a vulnerable state should not exceed some threshold. Keeping the average reliability above a required level could be achieved by calculating the current average based on the quote between *MTBF* and t_r, replication time. In the simplest case, when all nodes have the same *MTBF* this could be achieved by using eq. (6.1).

$$\frac{(MTBF/n - t_R)}{(MTBF/n)} \cdot R^n + \frac{t_R}{(MTBF/n)} \cdot R^{n-1} \tag{6.1}$$

Our model is also to be tested using geographically distributed clusters. If such a case, one must take into account that the replication time will be higher when replicating to a node in another cluster than within the same cluster. Furthermore, whether or not the heartbeat system and handling of lost nodes should be done globally or only locally within a cluster is yet to be investigated.

Finally, assuming statistically independent failures, the reliability of a node depends only on the time t, and the node's *MTBF*. However, since failures are likely to not be statistically independent, they should be taken into account in the reliability model. For example, if the switch of a rack fails, all nodes in that rack are unavailable. Therefore, the reliability of when having two replicas placed on two nodes in the same rack, is likely less than the reliability of two replicas placed on nodes in different racks.

Chapter 7

Conclusions

The goal of this thesis was first to devise a method for dynamically ensuring a predefined required level of reliability for distributed applications or services. This was accomplished by first designing reliability model based on nodes' *mean-time-between-failures*, and secondly a framework which automatically detects node failures and ensures the predefined reliability is met.

Reliability is increased by replication of tasks, and the required reliability is ensured by creating enough replicas. In order to optimize the number of replicas over time, the system is periodically monitored in order to dynamically adapt to resources' changing behavior.

Finally, the goal was also to provide a platform to be used to further experiments, and was accomplished by implementing our model using Calvin. Both the reliability model used and the task placement function are replaceable, which gives rise to various experimental opportunities.

References

[1] S. M. Shatz, J. Wang and M. Goto, "Task Allocation for Maximizing Reliability of Distributed Computer Systems", *IEEE Transactions on Computers*, vol. 41, no. 9, pp. 1156-1168, Sep. 1992.

[2] W. Ahmed and Y. W. Wu, "A survey on reliability in distributed systems", *Journal of Computer and System Sciences*, vol. 78, no. 8, pp. 1243?1255, Dec. 2013.

[3] A. Immonen and E. Niemelä, "Survey of reliability and availability prediction methods from the viewpoint of software architecture", *Software & Systems Modeling*, vol. 7, no. 1, pp. 49-65, Feb. 2008.

[4] C. A. T?nasie, S. Vîntutis and A. Grigorivici, "Reliability in Distributed Software Applications", *Informatica Economic?*, vol. 15, no. 4, pp. 167-177, 2011,

[5] C. Dabrowski, "Reliability in grid computing systems", *Concurrency Computation: Practice Experience - A Special Issue from the Open Grid Forum*, vol. 21, no. 8, pp. 927-959, June 2009.

[6] M. Katyal and A. Mishra, "A Comparative Study of Load Balancing Algorithms in Cloud Computing Environment", *International Journal of Distributed and Cloud Computing*, vol. 1, no. 2, pp. 5-14, Dec. 2013.

[7] Y. Dai and G. Levitin, "Reliability and Performance of Tree-Structured Grid Services", *IEEE Transactions on Reliability*, vol. 55, no. 2, pp. 337-349, June 2006.

[8] M. Treaster, "A Survey of Fault-Tolerance and Fault-Recovery Techniques in Parallel Systems", *Cornell University Library*, Jan. 2005.

[9] F. C. Gärtner, "Fundamentals of Fault-Tolerant Distributed Computing in Asynchronous Environments", *ACM Computing Surveys*, vol. 31, no. 1, pp. 1-26, Mar. 1999.

[10] K. Garala, N. Goswami and P. Maheta, "A Performance Analysis of Load Balancing Algorithms in Cloud Environment", *2015 International Conference on Computer Communication and Informatics (ICCCI)*, pp.1-6, Jan. 2015.

[11] S. Wang, K. Li, J. Mei, K. Li and Y. Wang, "A Task Scheduling Algorithm Based on Replication for Maximizing Reliability on Heterogeneous Computing Systems", *28th IEEE International Parallel & Distributed Processing Symposium Workshops (IPDPSW)*, pp. 1562-1571, May 2014.

[12] Z. Guessoum, J. Briot, O. Marin, A. Hamel and P. Sens, "Dynamic and Adaptive Replication for Large-Scale Reliable Multi-agent Systems", *Lecture Notes in Computer Science*, vol. 2603, pp 182-198, Apr. 2003.

[13] F. Cao and M. Zhu, "Distributed workflow mapping algorithm for maximized reliability under end-to-end delay constraint", *The Journal of Supercomputing*, vol. 66, no. 3, pp. 1462-1488, Dec. 2013.

[14] A. Dogan and F. Özüner, "Matching and Scheduling Algorithms for Minimising Execution Time and Failure Probability of Applications in Heterogeneous Computing", *IEEE Transactions on Parallel and Distributed Systems*, vol. 13, no.3, pp. 308-323, Mar. 2002.

[15] H. Wan, H. Huang, J. Yang and Y. Chen, "Reliability model of distributed simulation system", *2011 International Conference on Quality, Reliability, Risk, Maintenance, and Safety Engineering (ICQR2MSE)*, pp. 99-104, June 2011.

[16] C. S. Raghavendra and S. V. Makam, "Reliability Modeling and Analysis of Computer Networks", *IEEE Transactions on Reliability*, vol. 35, no. 2, pp. 156-160, June 1986.

[17] Y. Dai, B. Yang, J. Dongarra and G. Zhang, "Cloud Service Reliability: Modeling and Analysis", *15th IEEE Pacific Rim International Symposium on Dependable Computing*, Nov. 2009.

[18] H. Faragardi, R. Shojaee, M. Keshtkar and H. Tabani, "Optimal task allocation for maximizing reliability in distributed real-time systems", *12th International Conference on Computer and Information Science (ICIS), 2013 IEEE/ACIS*, pp. 513-519, June 2013.

[19] A. J. Oliner, R. K. Sahoo, J. E. Moreira and M. Gupta, "Performance Implications of Periodic Checkpointing on Large-scale Cluster Systems", *19th IEEE International Parallel and Distributed Processing Symposium*, Apr. 2005.

[20] B. Schroeder and G. Gibson, "A large-scale study of failures in high-performance computing systems", *IEEE Transactions on Dependable and Secure Computing*, vol. 7, no. 4, pp.337-350, Nov. 2010.

[21] Y. Zhang, M. S. Squillante, A. Sivasubramaniam and R. K. Sahoo, "Performance Implications of Failures in Large-Scale Cluster Scheduling", *Lecture Notes in Computer Science* , vol. 3277, pp. 233-252, 2005.

[22] L. Fiondella, and L. Xing, "Discrete and continuous reliability models for systems with identically distributed correlated components", *Reliability Engineering & System Safety*, vol. 133, pp. 1-10, Jan. 2015.

[23] A. Litke, D. Skoutas, K. Tserpes and T. Varvarigou, "Efficient task replication and management for adaptive fault tolerance in Mobile Grid environments", *Future Generation Computer Systems*, vol. 23, no. 2, pp. 163-178, Feb. 2007.

[24] Y. Ling and Y. Ouyang, "Real-time fault-tolerant scheduling algorithm for distributed computing systems", *Journal of Digital Information Management*, vol. 10, no. 5, pp. 289-294, Oct. 2012.

[25] J.T. Daly, "A higher order estimate of the optimum checkpoint interval for restart dumps", *Future Generation Computer Systems*, vol. 22, no. 3, pp. 303-312, Feb. 2006.

[26] X. Zhang and H. Pham, "An analysis of factors affecting software reliability", *The Journal of Systems and Software*, vol. 50, no. 1, pp. 43-56, Jan. 2000.

[27] P. Saxena and K. Govil, "An Algorithm for Optimized Time, Cost, and Reliability in a Distributed Computing System", *International Journal of Advanced Networking and Applications*, vol. 4, no. 5, pp. 1710-1718, Apr. 2013.

[28] M. Hussin, N. A. W. A Hamid and K. A. Kasmiran, "Improving reliability in resource management through adaptive reinforcement learning for distributed systems", *Journal of Parallel and Distributed Computing*, vol. 75, pp. 93-100, Jan. 2015

[29] Y. Brun, J. Bang, G. Edwards and N. Medvidovic, "Self-Adapting Reliability in Distributed Software Systems", *IEEE Transactions on Software Engineering*, vol. 41, no. 8, pp. 764-780, Aug. 2015.

[30] P. Chevochot and I. Puaut, "Scheduling Fault-Tolerant Distributed Hard Real-Time Tasks Independently of the Replication Strategies", *Sixth International Conference on Real-Time Computing Systems and Applications, 1999. RTCSA '99*, pp. 356-363, Dec. 1999.

[31] P. Persson and O. Angelsmark, "Calvin - Merging Cloud and IoT", *Procedia Computer Science*, vol. 52, pp. 210?217, 2015

[32] P. Yin, S. Yu, P. Wang and Y. Wang, "Task allocation for maximizing reliability of a distributed system using hybrid particle swarm optimization", *The Journal of Systems and Software*, vol. 80, no. 5, pp. 724-735, May 2007.

[33] Y. Dai and G. Levitin, "Optimal Resource Allocation for Maximizing Performance and Reliability in Tree-Structured Grid Services", *IEEE Transactions on Reliability*, vol. 56, no. 3, pp. 444-453, Sep. 2007.

[34] A. Dogan and F. Özgüner, "Matching and Scheduling Algorithms for Minimizing Execution Time and Failure Probability of Applications in Heterogeneous Computing", *IEEE Transactions on Parallel and Distributed Systems*, vol. 13, no. 3, pp. 308-323, Mar. 2002.

[35] S. Srinivasan and N. K. Jha, "Safety and Reliability Driven Task Allocation in Distributed Systems", *IEEE Transactions on Parallel and Distributed Systems*, vol. 10, no. 3, pp. 238-251, Mar. 1999.

[36] S. Kartik, C. Siva Ram Murthy, "Improved Task-Allocation Algorithms to Maximize Reliability of Redundant Distributed Computing Systems", *IEEE Transactions on Reliability*, vol. 44, no. 4, pp. 575-586, Dec. 1995.

[37] M. Amoon, "Design of a Fault-Tolerant Scheduling System for Grid Computing", *2011 Second International Conference on Networking and Distributed Computing*, pp. 104-108, Sept. 2011.

[38] M. Chtepen B. Dhoedt, F. Turck, P. Demeester, F. Claeys and P. Vanrolleghem, "Evaluation of replication and rescheduling heuristics for grid systems with varying resource availability", *Proceedings of the 18th IASTED International Conference on Parallel and Distributed Computing and Systems*, pp.622-627, Nov. 2006.

[39] J. H. Abawajy, "Fault-Tolerant Scheduling Policy for Grid Computing Systems", *Proceedings of the 18th International Parallel and Distributed Processing Symposium*, Apr. 2004.

[40] M. Chtepen, F. Claeys, B. Dhoedt, F. Turck, P. Demeester and P. A. Vanrolleghem, "Adaptive Task Checkpointing and Replication: Toward Efficient Fault-Tolerant Grids", *IEEE Transactions on Parallel and Distributed Systems*, vol. 20, no. 2, Feb. 2009

[41] J. E. Pezoa and M. M. Hayat, "Performance and Reliability of Non-Markovian Heterogeneous Distributed Computing Systems", *IEEE Transactions on Parallel and Distributed Systems*, vol. 23, no. 7, pp. 1288-1301, July 2012.

[42] Y. Dai, Y. Pan and X. Zou, "A Hierarchical Modeling and Analysis for Grid Service Reliability", *IEEE Transactions on Computers*, vol. 56, no. 5, pp. 681-691, May 2007.

[43] D. Chen and T. Huang, "Reliability analysis of distributed systems based on a fast reliability algorithm", *IEEE Transactions on parallel and distributed systems*, vol. 3, no. 2, pp. 139-154, Mar. 1992.

[44] Gregory Levitin, Yuan-Shun Dai and H. Ben-Haim, "Reliability and Performance of Star Topology Grid Service with Precedence Constraints on Subtask Execution", *IEEE Transactions on Reliability*, vol. 55, no. 3, pp. 507-515, Sept. 2006.

[45] W. T. Tsai, D. Zhang, Y. Chen, H. Huang, R. Paul and N. Liao, "A Software Reliability Model for Web Services", *Proceedings of the IASTED Conference on Software Engineering and Applications*, Nov. 2004.

[46] Y.S. Dai, M. Xie, K.L. Poh and G.Q. Liu, "A study of service reliability and availability for distributed systems", *Reliability Engineering and System Safety*, vol. 79, no. 1, pp. 103-112, Jan. 2003.

[47] M. Lin, M. Chang and D. Chen, "Efficient algorithms for reliability analysis of distributed computing systems", *Information Sciences*, vol. 117, no. 1-2, pp. 89-106, July 1999.

[48] E. Huedo, R. S. Montero and I. M. Llorente, "Evaluating the reliability of computational grids from the end user?s point of view", *Journal of Systems Architecture*, vol. 52, no. 12, pp. 727-736, Dec. 2006.

[49] A. Kumar and D. P. Agrawal, "A Generalized Algorithm for Evaluating Distributed-Program Reliability", *IEEE Transactions on Reliability*, vol. 42, no. 3, pp. 416-426, Sept. 1993.

[50] D. Chen, M. Sheng and M. Homg, "Real-Time Distributed Program Reliability Analysis", *Proceedings of the Fifth IEEE Symposium on Parallel and Distributed Processing*, pp. 771-778 Dec. 1993.

[51] T. Altameem, "Fault Tolerance Techniques in Grid Computing Systems", *International Journal of Computer Science and Information Technologies*, vol. 4, no. 6, pp. 858-862, Dec. 2014.

[52] A. Bala and I. Chana, "Fault Tolerance Challenges, Techniques and Implementation in Cloud Computing", *International Journal of Computer Science Issues*, vol. 9, no. 1, pp. 288-293, Jan. 2012.

[53] Y. Li and, M. Mascagni, "Improving Performance via Computational Replication on a Large-Scale Computational Grid", *3rd IEEE/ACM International Symposium on Cluster Computing and the Grid*, pp. 442-448, May 2003.

[54] D. Sylvian, Z. Guessoum and M. Ziane, "Adaptive Replication in Fault-Tolerant Multi-Agent Systems", *IEEE/WIC/ACM International Conferences on Web Intelligence and Intelligent Agent Technology*, vol. 2, pp. 304-307, Aug. 2011.

[55] A. Fedoruk and R. Deters, "Improving Fault-Tolerance by Replicating Agents", *Proceedings of the first international joint conference on Autonomous agents and multiagent systems: part 2*, pp. 737-744, Jan. 2002.

[56] Z. Guessoum, J. Briot, N. Faci and O. Marin, "Towards Reliable Multi-Agent Systems: An Adaptive Replication Mechanism", *Multiagent and Grid Systems*, vol. 6, no. 1, pp. 1-24, Mar. 2010.

[57] R. Yadav and A. S. Sidhu, "Fault Tolerant Algorithm for Replication Management in Distributed Cloud System", *2015 IEEE 3rd International Conference on MOOCs, Innovation and Technology in Education (MITE)*, pp. 78-83, Oct. 2015.

[58] J. S. Plank and W. R. Elwasif, "Experimental Assessment of Workstation Failures and Their Impact on Checkpointing Systems", *Digest of Papers. Twenty-Eighth Annual International Symposium on Fault-Tolerant Computing, 1998*, pp. 48-57, June 1998.

[59] R. Nallakumar, N. Sengottaiyan and K. S. Sruthi Priya, "A Survey on Scheduling and the Attributes of Task Scheduling in the Cloud", *International Journal of Advanced Research in Computer and Communication Engineering*, vol. 3, no. 10, pp. 8167-8171, Oct. 2014.

[60] A. Jangra and T. Saini, "Scheduling Optimization in Cloud Computing", *International Journal of Advanced Research in Computer Science and Software Engineering*, vol. 3, no. 4, pp. 62-65, Apr. 2013.

[61] D. Strauss, "Containers?Not Virtual Machines?Are the Future Cloud", June 2013. [Online].
Available: http://www.linuxjournal.com/content/containers%E2%80%94not-virtual-machines%E2%80%94are-future-cloud. [Accessed: 25 May 2016].

[62] K. R. Joshi, M. A. Hiltunen, W. H. Sanders and R. D. Schlichting, "Probabilistic Model-Driven Recovery in Distributed Systems", *IEEE Transactions on Dependable and Secure Computing*, vol. 8, no. 6, pp. 913-928, Nov./Dec. 2011.

[63] P. Maymounkov and D. Mazières, "Kademlia: A Peer-to-peer Information System Based on the XOR Metric", *Lecture Notes in Computer Science*, vol. 2429, pp. 53-65, Oct. 2002.

[64] U. Malladi, "Notice of Violation of IEEE Publication Principles Design, analysis and performance evaluation of a new algorithm for developing a fault tolerant distributed system", *12th International Conference on Parallel and Distributed Systems*, vol. 1, July 2006.

[65] P. Li and B. McMillin, "Fault-tolerant distributed deadlock detection/resolution", *Proceedings., Seventeenth Annual International Computer Software and Applications Conference, 1993. COMPSAC 93.*, pp. 224-230, Nov. 1993.

[66] Y. Li, W. Li and C. Jiang, "A Survey of Virtual Machine System: Current Technology and Future Trends", *2010 Third International Symposium on Electronic Commerce and Security (ISECS)*, pp. 332-336, July 2010.

Appendix A
Figures

This Appendix consist of various figures.

- Appendix A.1 shows an extended application model with two replicated services.
- Appendix A.2 and Appendix A.3 shows the computational environment of four nodes and one service between a producer and a consumer before and after a node failure. The required reliability is 0.995 and for a certain replication time and failure rate, the reliability of the nodes are given under their name.
- Appendix A.4 shows the communication flow after a node failure.
- Appendix A.5 the measure replication times for state sizes less than 100 MB.
- Appendix A.6 shows the function from which the sleep-times in experiment 4.5.6 where picked from.
- Appendix A.7 and Appendix A.8 shows the worst and best case scenario for detecting a node failure.

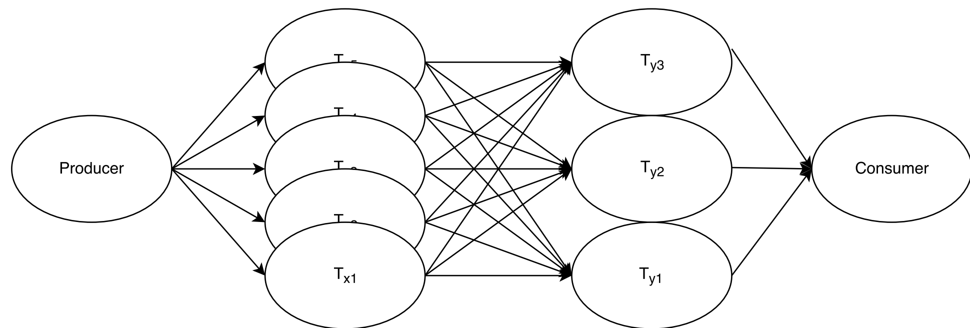

Figure A.1: The data flow when two tasks, *T1* and *T2* with different required reliability are replicated.

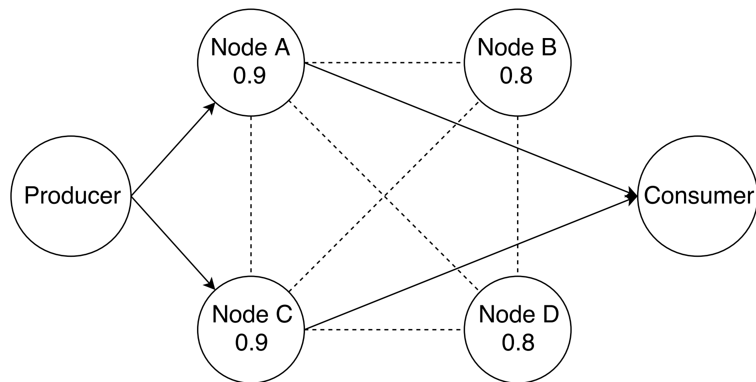

Figure A.2: Before a node failure. Here two replicas (placed on Node A and C) are required to achieve required reliability.

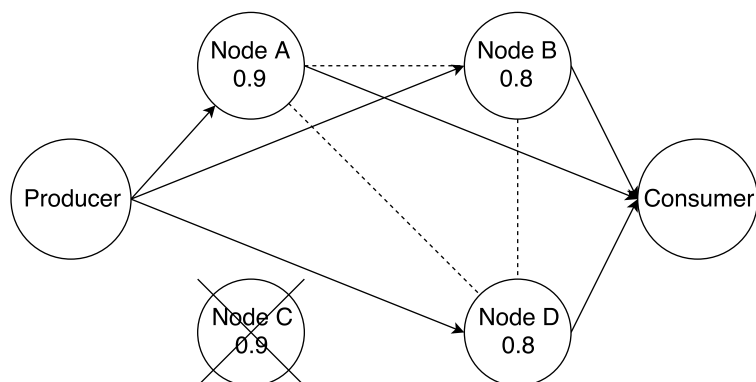

Figure A.3: After Node C has failed. Now three replicas (placed on Node A, B and D) are required to achieve required reliability.

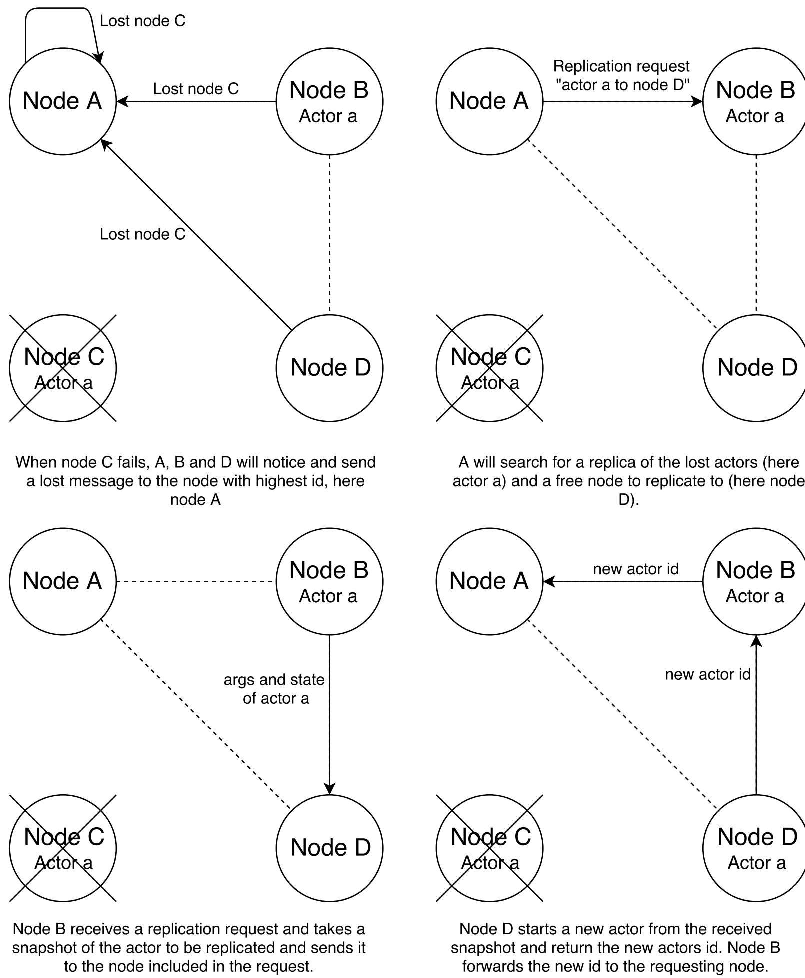

Figure A.4: The most important parts of the communication after a node has failed.

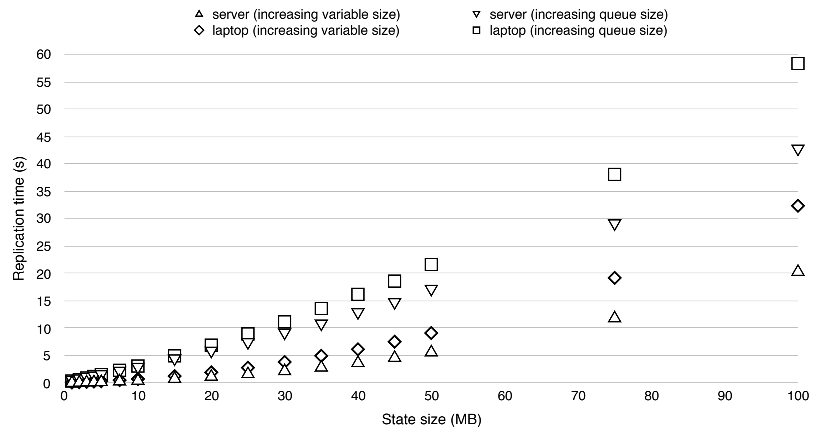

Figure A.5: The average replication time as a function of state size.

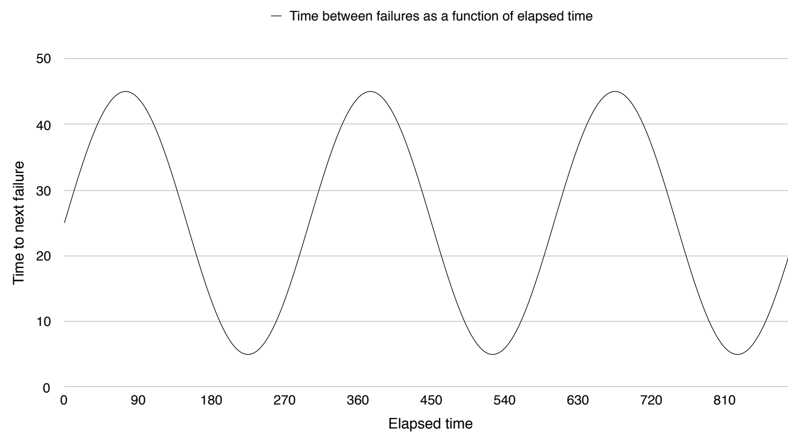

Figure A.6: Time between failures as of eq. (4.1) used in experiment 4.5.6.

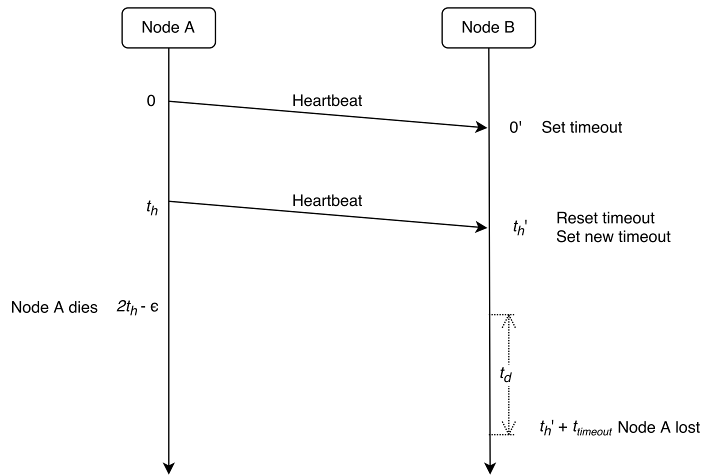

Figure A.7: Best case scenario for detecting a node failure. Node A fails just before sending a heartbeat. The actual detection time, t_d is therefore close to $t_\mathrm{timeout} - t_\mathrm{h}$.

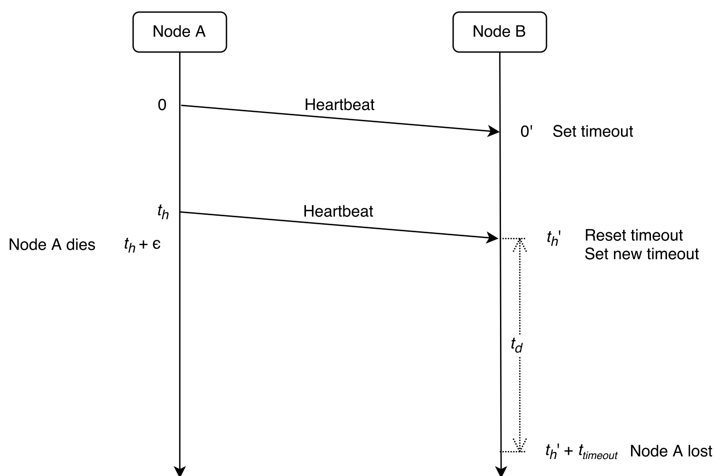

Figure A.8: Worst case scenario for detecting a node failure. Node A fails directly after sending a heartbeat. The actual detection time, t_d is therefore close to t_timeout.

LUND UNIVERSITY

Series of Master's theses
Department of Electrical and Information Technology
LU/LTH-EIT 2016-510

http://www.eit.lth.se