Dynamic Fault-Tolerance and Task Scheduling in Distributed Systems

Philip Ståhl
ada10pst@student.lu.se

Jonatan Broberg
elt11jbr@student.lu.se

April 20, 2016

Master's thesis work carried out at Mobile and Pervasive Computing Institute (MAPCI), Lund University.

Supervisors: Björn Landfeldt, bjorn.landfeldt@eit.lth.se

Examiner: Christian Nyberg, christian.nyberg@eit.lth.se

Abstract

This document describes the Master's Thesis format for the theses carried out at the Department of Computer Science, Lund University.

Keywords: reliability, distributed computing, dynamic fault-tolerance, task scheduling, Poisson

Acknowledgements

We would like to thank our supervisor Björn Landfeldt and MAPCI for there input.

Maybe also: Shubhabatra, Ericsson, Jörn, Doan, examiner Christian Nyberg. (Or simply all the people at Mapci)

Contents

Chapter 1
Introduction

1.1 Background and Motivation

Ensuring a certain level of reliability is of major concern for many cloud system providers. As cloud computing is growing rapidly and users are demanding more services and higher performance, providing fault tolerant systems and improving reliability by more sophisticated task scheduling, has become a very important and interesting research area.

Cloud systems often consists of heterogeneous hardware and any of the often vast number of components may fail at any time. Therefore, 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.

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 need to be in place. This is of particular interest for cloud service providers, as users often desire a certain level of reliability. In some cases, vendors, for example carrier providers, are obliged by law to achieve a certain level of availability or reliability. In these cases, one may need to sacrifice other quality aspects such as latency and resource usage. By using static and dynamic analysis of the infrastructure and the mean-time-between-failure for the given resources, a statistical model can be used in order to verify that the desired level is reached. However, such a model should be dynamic, as failure rates are likely to vary over time, for example due to high or low load on the system. This is of particular importance for long running applications when requiring a certain level of reliability. Despite fulfilling the required level at the time of deployment, as the state of the system changes, the level may no longer be fulfilled.

Reliability can be increased by replicating application tasks, where all replicas perform the same computations on the same input. This allows for both increased redundancy and an easy way of detecting errors. This allows for continuing execution of the application even in case of a filing replica. Seamlessly being able to continue the execution, without losing any data is of particular interest in data stream processing.

One drawbacks of replicating a task n times, is that the resources needed increases. With *n* replicas, all performing the same computation on the same input, one need *n* times as much resources, hence a lot of computational resources is thus wasted. Dynamic analysis of the system and an adaptive scheduling technique could help in determining on which resources one should assign the tasks to for optimal resource usage and load balancing.

Throughout this master thesis, extensive literature studies has been made, first in order to find out what has already been done in the area and later in order to get valuable help in developing a reliability model.

1.2 Related work

The interest in reliability in distributed systems has gain increased knowledge [63]. Due to the uncertain heterogeneous environment of cloud and grid systems, increasing reliability is a complex task [TODO].

A lot of scheduling techniques has been designed, aiming at maximizing reliability of jobs in distributed environments, under various constraints such as meeting task deadlines or minimizing execution time [31] [20] [1] [36] [15] [16] [34]. Maximizing reliability for these algorithms are a secondary concern, while meeting the constraints are the primary. Other algorithms have been developed which put greater focus on increasing the reliability [37] [38], and some have increased reliability as the primary objective [39] [40]. Common for these scheduling techniques are that while they try to maximize reliability, they do not ensure a certain level of reliability to the user. Furthermore, the algorithms are usually static in the way that they do not account for the dynamic behavior of distributed system, and they make assumptions such as known execution times of tasks.

A lot of work has been done in the area of designing fault-tolerant systems by using checkpoint/restart techniques [TODO]. These techniques relies of the notion of a stable storage, such as a hard-drive, which is persistent even in the case of a system failure.

Some attempts at designing fault-tolerant systems by the use of replication has been made [41] [42] [12] [25] [49]. [42] assumes a static number of replicas, which 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 when resources are available. While both [25], [12] and [41] all dynamically determines the number of replicas based on the state of the system, it is static in the way that failed replicas are not restarted. The reliability level for long running applications are therefore decreased if replicas fail. Furthermore, while [41] dynamically determines the number of replicas to use, the selection of resources is done after the number of replicas has been determined. In order to ensure a certain level of reliability, these two parts need to be combined into one, because the resources selected affects the number of replicas needed in order to achieve the desired reliability.

A quite old but still relevant work is found in [14] where they present a framework for dynamic replication with an adaptive control in an multi-agent system. They introduce the software architecture which can act as support for building reliable multi-agent systems.

Since the available resources often are limited they say that it isn't feasible to replicate all components. Therefore they use a criticality for each agent which is allowed to evolve during runtime. The proposed solution allows for dynamically adapt the number of replicas and the replication strategy itself (passive/active). The number of replicas is partly based on the agents critically and a predefined minimum number of replicas. From CPU usage time and communication activity an agent activity is calculated which is used in calculating the agents critically. One restriction they do in their fault model is that processes can only fail by permanent crashes.

Other approaches to improve reliability in Multi-Agent Systems (MAS) by the use of replication is presented in [61] [60] [62]. While being adaptive to system state, the solution presented in [61] still faces the problem of having a single point of failure due to a communication proxy. This problem is avoided in [60], where a decentralized solution is proposed, where the number of replicas and their placement depends on the system state. The solution proposed in [62] involves distributed monitoring system...

[44] proposes an algorithm based on replication which dynamically varies the number of replicas depending on system load. However, the algorithm reduces the number of replicas during peak hours, in order to reduce system load. Since the reliability of system decreases during higher load [22] [22] [23], one should increase the number of replicas to keep the desired level of reliability.

A fault-tolerant scheduling technique incorporating a replication scheme is presented in [43]. While being dynamic in that failed replicas are restarted, it is static in that the user defines the number of replicas to use.

The techniques used in [33] [14] [50] are more dynamic and adaptive to the dynamic behavior of distributed systems. However, reliability is defined as producing the correct result, and achieved by techniques like voting and *k-modular redundancy*. An adaptive approach, which adapts to changes in the execution environment is presented in [32]. In it, they present an adaptive scheduling model based on reinforcement learning, aiming at increasing the reliability. However, they assume a task's profile is available.

1.3 Our contributions

To our knowledge, no previous attempt has been made which in a fully dynamic manner ensures a certain level of reliability for long running applications. 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 an user determined level of reliability for long running applications by the use of replication. Furthermore, the method ensures a minimized use of resource by not using more replicas than needed. This is achieved by scheduling replicas to the most reliable resources first and foremost. Furthermore, the system is continuously monitored in order to adapt the number of replicas as the state of the system changes.

The framework is not limited to a specific type of distributed environment, and its key concepts may be used in both grid and cloud systems.

Our model is implemented by extending the actor-based application environment *Calvin* [35], developed by Ericsson. While *Calvin* is mainly an environment for IoT applications, it suites our purpose well. The model is evaluated by running a set of experiments in a small cluster.

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 aims at evaluating our solution while chapter 6 presents future work. Chapter 7 concludes the report.

1.4 Goal

The goal of this thesis is to devise a method for dynamically ensuring a certain level of reliability for distributed applications or services. Reliability will be achieved through replication of tasks.

First, a reliability model will be designed, describing the reliability for an application running in a distributed environment.

Secondly, a framework will be designed which will automatically detect node failures and based on the reliability model spawns enough replicas to reach the desired reliability level above.

Lastly, the system will by periodically monitored in order to adapt the reliability model and the replicas needed as the properties of the system varies over time.

The model will be implemented and tested using the IoT application framework *Calvin*.

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 [17] [52]. There are a number of different types of distributed environments, e.g. grid, clusters, cloud and HDCS.

2.1.1.1 Heterogeneous distributed computing systems

Heterogeneous Distributed Computing Systems, HDCS is a system of numerous high-performance machines connected in a high-speed network. Therefore high-speed processing of heavy applications is possible [16].

The majority of distributed service systems can be viewed as CHDS, (Centralized Heterogeneous Distributed System). A CHDS consists of heterogeneous sub-systems which a managed by a centralized control center. The sub-systems have various operating platforms and are connected in diverse topological networks [51].

2.1.1.2 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] [53].

Each domain in a grid usually has a centralized scheduling service called Resource Management System (RMS) which accepts job execution requests and sends the job's tasks to the different resources for execution [53].

The reliability of the cloud 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 [19].

2.1.1.3 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].

2.1.1.4 Cloud

A cloud has been described as the next generation of grids and clusters. While it is similar to grid and clusters, for example parallel and distributed, the important difference is that cloud has multiple domains [6]. Machines can be geographically distributed, and software and hardware components are often heterogeneous, and therfore analyzing and predicting workload and reliability is usually very challenging [2].

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 installed [6]. Prior knowledge of node capacity, processing power, memory, performance and statistics of user requirements are required. Changes in load during run time is not taken into account which makes environment easy to simulate but not well suited for heterogeneous resources.

In a dynamic environment heterogeneous resources are installed [6]. In this scenario prior knowledge isn't enough since the requirements of the user can change during run time. Therefore run time statistics is collected and taken into account. The dynamic environment is difficult to simulate but there are algorithms which easily adopt to run time changes in load.

2.2 Faults in distributed environments

In a distributed environment a larger number of faults can occur due to the higher complexity. Therefore a fault model is usually employed, describing what kind of faults which is 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 may affect the running applications in the environment. These failure include, but are not limited to, overflow failure, timeout failure, resource missing failure, network failure, hardware failure, software failure, and database failure [19]. Failures are usually considered to be either [53]:

- Job related, or
- System related, or
- Network related.

The possible errors in a Grid environment can be divided into the following three categorize [25]:

- Crash failure - When a correctly working server comes to a halt.
- Omission failure - When a server fails to respond to incoming requests and to send messages
- Timing failure - When a server responds correctly but beyond the specified time interval

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

2.2.2 Fault models

When studying the reliability of distributed systems or reliability of applications running in distributed computing environments, one usually starts with specifying which fault model is used, and the model developed is then proved with respect to this fault model [9].

2.2.2.1 Byzantine Faults

The Byzantine fault model allows nodes to continue interaction after failure. Correctly working nodes cannot automatically detect if a failure has occurred and even if it was known that a failure has occurred they cannot detect which nodes has failed. The systems behaviour 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 try to send an incorrect result. All Byzantine nodes might send the same incorrect result making it hard to identify malicious nodes [33]. The Byzantine fault model is very broad since it allows failed nodes to continue interacting, therefore it is very difficult to analyze.

2.2.2.2 Fail-stop faults

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 a system crash or if the system hangs [8]. The fail-stop model has been criticized for not representing enough real-world failures.

2.2.2.3 Fail-stutter faults

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 it also allows for performance fault, such as unexpectedly low performance of a node [8].

2.2.2.4 Crash-failure model

The crash failure model is quite alike the fail-stop model with the difference that the other nodes do not automatically detect that a node has failed [9] [62].

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. Common assumptions are stated in definition 2.1 [17] [18] [19] [51] [47] [33]:

Definition 2.1. *Common assumptions when modeling failures are:*

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

For reliability modelling for the grid, it is common to also assume a fully reliable Resource Management System (RMS) [7] [49], which is an non-neglect-able assumption since the RMS in this case in fact is a single point of failure.

Constant failure rates are not likely to model the actual failure scenario of a dynamic heterogeneous distributed system [16]. The failure rate for a hardware component usually follows a bathtub shaped curve [2]. The failure rate is usually higher in the beginning due to that the probability that a manufacture failure would affect the system is higher in the beginning of the systems lifetime. After a certain time the failure rate drops and later increases again due to that the component gets worn out.

Statistically independent failures is also not very likely to reflect the real dynamic behaviour of distributed systems [2] [19]. 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 [37]. As failures are likely to be correlated [21], the probability of failure increases with the number of components on which the job is running.

Other factors affect the likelihood of resource failures as well. Several work has concluded a relationship between failure rate and the load of the system [22] [23]. Furthermore, studies show that failures are more likely to occur during daytime than at night [23] [22]. Finally, components which have failed in the past are more likely to fail again [23].

While a Poisson process is commonly used to describe failures, [22] showed 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 behaviour of a resource, a Poisson process has been experimentally shown to be reasonable useful in mathematical models [64].

2.3 Reliability

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] [17] [51] [47] [54] [55] [33] [46]. A similar definition of reliability, usually used for applications in distributed environments, is the probability of a software application, running in a certain environment, to perform its intended functions for a specified period of time [2] [3] [4], and is common for application with time constraints. Finally, reliability can also be defined as the probability that a task produces the correct result [3] [7] [49] [50] [33]. This definition is usually used together with the Byzantine fault model.

[2] defines a software application reliable if the following is achieved:

- Perform well in specified time t without undergoing halting state
- Perform exactly the way it is designed
- Resist various failures and recover in case of any failure that occurs during system execution without proceeding any incorrect result.
- Successfully run software operation or its intended functions for a specified period of time in a specified environment.
- Have probability that a functional unit will perform its required function for a specified interval under stated conditions.
- Have the ability to run correctly even after scaling is done with reference to some aspects.

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

Definition 2.2. *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 multi-task applications, where the tasks use more than one resource, reliability is defined as

Definition 2.3. *The reliability of a multi-task process is the probability that the tasks being executed within a given time without experiencing any type of failure (internal or external) during the time of execution.*

Finally, for long running applications, where a replication scheme is used, the reliability can be defined as [25]

Definition 2.4. *The reliability of a process, with n task replicas, is the probability that at least one replica is always running. This can be expressed as the probability that not all replicas fail during the time from that an actor dies until a new replica is up and running.*

2.3.1 Modelling reliability

Reliability of a system highly depends on how the system is used [3]. In order to determine the reliability of a system, one need to take all factors affecting the reliability into account [2]. However, including all factors if unfeasible. [29] lists 32 factors affecting the reliability of software, 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 [19] [47].

For distributed applications, the probability of failure increases since it is dependent on more components [17]. Most models used to model the reliability of a system is based on the mean time to failure, or the mean time between failures, of components [18]. Conventionally, Mean-Time-To-Failure (MTTF) refers to non-repairable resources, while Mean-Time-Between-Failures (MTBF) refers to repairable objects [25].

Definition 2.5. *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 2.6. *The Mean-Time-Between-Failure for a component is the average time between successive failures for that component.*

The MTBF can be calculated as

$$MTBF = (total\ time)/(number\ of\ failures) \tag{2.1}$$

From eq. (2.1), a reliability function for a component can be expressed as

$$R(t) = e^{-t/MTBF} \tag{2.2}$$

Equation (2.2) expresses the probability that a given resource will work for a time t. Correspondingly, the probability that a resource will fail during a time t is

$$F(t) = 1 - e^{-t/MTBF} \tag{2.3}$$

2.4 Fault tolerance techniques

Fault tolerance techniques are used to either predict failures and take an appropriate action before they actually occur [58] or to prepare the system that failures might occur and take an appropriate action 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 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 faults

4. Fault forecasting - Predicting or estimating faults at architectural level during design phase or before actual deployment

Limited to already developed application, fault tolerance techniques can be divided into reactive and proactive techniques [58]. A reactive fault tolerant technique reacts when a failure occur and tries to reduce the effect of the failure. They therefore consists of detecting fault and failures, and recovering from them 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 is the most basic form of fault tolerance [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 the actions replayed, which will reconstruct the previous state [8]. In contrast to check pointing, rollback recovery returns the state to the most recent state, not only the last saved one. Rollback recovery can however be used in combination with check pointing in order to decrease recovery time, not needing to replay all actions, but only those from the latest checkpoint.

2.4.3 Replication

Job replication is a commonly used fault tolerance 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 [57].

Using replication, several identification 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 [57].

Replication also minimizes the risk of failures affecting the execution time of jobs, since it avoids recomputation typically necessary when using checkpoint/restart techniques [41].

There are three different strategies for replicating a task: *Active*, *Semi-active* and *Passive*.

- In active replication, one or several replicas are run on a other machine and receive an exact copy of the primary nodes input. From the input they perform the same calculations as if they were the primary node. The primary node is monitored for incorrect behaviour and in event that the primary node fails or behaves in an unexpected way, one of the replicas will promote itself as the primary node [8]. This type of replication is feasible only if by assumption that the two nodes receives exactly the same input. Since the replica already is in an identical state as the primary node the transition will take negligible amount of time. A drawback with active replication is that all calculations are ran twice, thus a waste of computational capacity.

  Active replication can also be used in consensus algorithms such as majority voting or k-modular redundancy, where one need to determine the correct output [8]. In this case, there is no primary and backup replicas, instead every replica acts as a primary.

- Semi-Active replication is very similar to active replication but with the difference that decisions common for all replicas are taken by one site.

- 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 has a minimal internal state, unless additional check pointing is employed.

A replica, whether active, semi-active or passive replication is used, is defined as [25]:

Definition 2.7. *The term replica or task replica is used to denote an identical copy of the original task*

While replication increases the system load, is may help improving performance by reducing task completion time [59].

Consensus

Replication is usually used with some form of consensus algorithm. 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 afterwards they agree on which result is considered correct. These 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 [33]. In traditional redundancy an odd number of replicas are executed simultaneously and afterwards they vote for which result is correct. The result with the highest number of votes are considered correct and consensus is reached.

In progressive redundancy on the other hand the number of replicas needed is minimized. Assume that with traditional redundancy $k \in 3, 5, 7...$ replicas is executed, progressive redundancy only 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 is executed until enough replicas has return 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.

Furthermore [33] present a third strategy, an iterative redundancy alternative which focus is more on reaching a required level of reliability in comparison to reaching a certain level of consensus.

2.5 Task scheduling

Task scheduling is the process of mapping tasks to available resource. A scheduling algorithm can be divided into three simple steps [66]:

1. Collect the available resources
2. Based on task requirements and resource parameters choose resources to schedule the task on
3. Send the task 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 aim at maximizing the total reliability, minimizing the overall system load or meeting all the tasks deadlines [65].

Task scheduling algorithms can be divided into static and dynamic algorithms depending on if the scheduling mapping is based on pre-defined parameters or if the parameters might change during runtime [34].

Many studies have been recently done to improve reliability by proper task allocation in distributed systems, but they have only considered some system constraints such as processing load, memory capacity, and communication rate [20]. In fact finding an optimal solution and maximizing the overall system reliability at the same time is a NP-hard problem [20] [36] [65].

Taking many factors in account when scheduling tasks results in a big overhead in the cloud, thus decreasing performance.

2.6 Monitoring

Monitoring is common in distributed system for detecting when resources fail. The most common technique is based on heartbeats, i.e. the resources send "I'm still alive"-messages with a certain pulse to each other and if a resource stops receiving heartbeats from a resource it is assumed to have died. Another technique is to send a test message and wait for a reply and after a certain timeout assume that the resource has died [68].

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 hardware cost and space and in that the system becomes more dynamic. For instance, since hardware servers usually has a lot of free capacity it is possible to run two virtual servers on each physical server. Another advantage with virtualization is that it separates the software from the hardware which enables the possibility of moving a virtual machine from one physical location to another.

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 [67].

Calvin, an actor-based application environment developed by Ericsson, use a concept of runtimes. A runtime contains all necessary components to run various kinds of actors, by taking care of things such as scheduling and communication between runtimes. The runtimes work as containers and separate the hardware from the software, resulting in that it is possible to use Calvin on different platforms. In developing phase of new applications the developer does not need to consider on which operating systems the application should run.

Chapter 3

Design of a dynamic fault-tolerant system

3.1 Methodology

The methodology for devising our model is mainly literature study as well as meetings and discussions with both our supervisor but also with colleges at MAPCI.

To validate our model and show its usefulness, it was implemented in Ericsson's actor-based application environment Calvin [35]. A somewhat simplistic reliability model was made, and more time spent on implementation, in order to be able to conduct various kinds of experiments.

An alternative approach would have been to make a more sophisticated reliability model, which maps better to real-world characteristics of distributed environments. However, while a more sophisticate theoretical model may map better to the real-world, it is impossible to include all parameters affecting the reliability of an application or service running in an distributed environment. Furthermore, with no actual implementation, only simulations could be conducted to verify the model, and even though simulations can take a lot of parameters into account, it would have to be based on failure data, which is usually collected from in-house testing, and thus cannot be compared with failures that can occur in an actual operational environment [2].

Since we limit ourselves to testing only in Calvin we lose some of the understanding of the results but we still believe that it can give a good first interpretation of the possibilities, see chapter 5.

Since a lot of test were already written to the Calvin system a test-driven development of our code came natural.

3.2 Introduction

In the case of 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 producer of some kind produces data which needs to be transformed by a service *S* before being sent to a consumer. The service *S* is running within a cluster of some service provider, 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[1] allows us to avoid timing issues and cases when the consumer receives different results. However, since using active replication the consumer will receive a result from each replica and our model could therefore easily be extended with consensus algorithms such as those presented in section 2.4.3 to determine whether or not the correct result was received.

Furthermore, long running applications or services, such as streaming applications, allows us to focus on ensuring a desired reliabity even in the case of failures. Therefore, in contrast to the various scheduling algorithms in [31] [20] [1] [36] [15] [16] [34] aiming at meeting task deadlines or minimizing execution time, a greedy scheduling algorithm will be presented which solely aims at reaching a desired level of reliability with the minimum number of replicas.

Unlike [41] [42] [12] [25] [49], we propose a fully dynamic model which monitors the system with its running applications and services, and create new replicas in order to ensure that the desired reliability is met. This is achieved by monitoring both how many replicas are operational, on which nodes they are executing, as well as node failures, in order to dynamically adapt our reliability model as the properties of the system varies.

The rest of this chapter is composed as follows, in section 3.3 we go through what limitations we have done. Section 3.4 gives a description of the system model we have used, e.g. in which computational environment we test our model, how we model the faults and their distribution and what kind of applications we use. In section 3.5 we present our reliability model, which is our main contribution with this paper. We start by giving diverse definitions and a scheduling algorithm and then go through our self-adapting model. Last in this chapter, section 3.6 we briefly describe how Calvin works and present the main parts of the implementation of our model.

3.3 Limitations

By assuming fully reliable links between nodes, we limit our model to only consider node failures. However, the reliability model used in this thesis could easily be extended to consider more parameters, or be replaced by another reliability model, and the scheduling algorithm presented in algorithm 1 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, but could be taken into account in a more sophisticated reliability model.

Furthermore, we assume the replicated task/service is deterministic and always produces the correct result for a given input. This allows us to focus on the reliability of producing a result and not losing any data, instead of the reliability of producing the correct

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

result. As mentioned before this can quite easily be solved by extending the model with consensus such as majority voting. Due to that we don't gain any extra useful information we have chosen to leave this a future work.

By the use of these limitations we avoid unnecessary complexity in form of check pointing and consensus. The impact of this will be discussed later in report, 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 paper 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.

A small 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 consists 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 chapter A a system of one replicated actor is shown before and after a node failure.

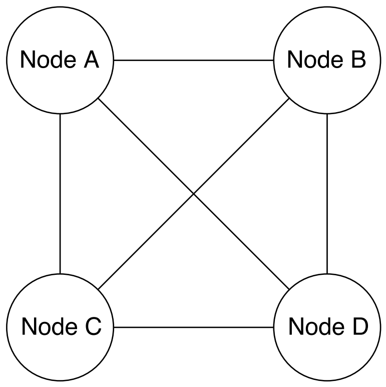

Figure 3.1: Computational environment, interconnected nodes

3.4.2 Monitoring

Monitoring the system is crucial to achieve an entirely dynamic system, in which certain reliability is dynamically ensured. One must know both where replicas are executing and be able to detect node failures.

3.4.2.1 Heartbeats

The ability to detect node failures is crucial for knowing when we need to create a new replica in order to keep the desired reliability. Furthermore, since the reliability model presented in section 3.5 is based on a *mean-time-between-failures* for the nodes in the system, it is based on the assumption that node failures is detectable.

To achieve this, a heartbeat system is used, where nodes periodically sends UDP messages, called heartbeats, containing a node identifier to the other nodes in the system. If no heartbeat is received from a node for a certain period of time, the node is assumed dead. The heartbeats are further described in section 2.6.

From that a node on which a replica is running fails, until a new replica is created, the reliability level may be lower than the desired level, hence the system in a vulnerable state. As further described in section 3.5, the time it takes to detect node failures is therefore an important part of the reliability model.

In our model, heartbeats are sent to all nodes in the system every 200 ms. A node is considered dead when no heartbeat is received for 500 ms, which is an upper bound to how long it takes to detect a node failure.

The frequency of the heartbeats, and the timeout time, is configurable and should be set depending on the desired behavior. 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 desired reliability.

3.4.3 Storage of data

All 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 stored on a single node, e.g. the executing node, that information is lost in the case that node fails. Also, if a remote database is used, we introduce a single-point-of-failure. Instead, to achieve a redundant storage in our model, Distributed Hash Table (DHT) is used, which efficiently avoids having a single-point-of-failure. More specifially, the DHT implementation used is Kademlia, and we refer to [?] for further info.

3.4.3.1 Updating storage

In order to know where replicas are executing, this information must be stored and globally available, see section 3.4.3 for further info.

When a task is replicated to another node, it must not be considered successful before the storage is properly updated. Otherwise, the replication must be partially reflected in the storage and will become useless.

3.4.4 Application model

The fault-tolerant framework presented in this paper is general and may be used in various contexts. However, it is of special value for long running applications and services, running dynamic environments, and where a certain level of reliability must be met, but where the reliability of the resources vary over time. Long running applications are particularly vulnerable to failure because they usually require many resources and usually must produce precise results [5].

In this paper we use a simple example application in our experiments. The application can be modelled as shown in fig. 3.2. The node *A* shown in the figure could for example be a service for which we require a certain reliability. No assumptions are made about the execution time of the application. However, our model 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 [32].

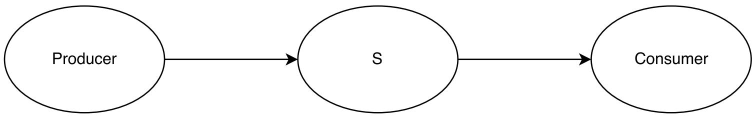

Figure 3.2: An application model where a producer transmits data to a task *S*, which transforms the data, and sends the result to a consumer. *S* may be seen as a task or service running in a cluster and requiring a certain level of reliability.

COMMENT: we should have some real examples of applications which would benefit from using our framework. ?

- telephone system?
- video trans-coding when streaming video to mobile phone? In this case one could imagine a case where a video file is located on a server with limited processing power. The video could therefore be streamed to another server in the cluster, with more processing power, which encodes it on the fly and stream the result to the user. To keep a continuous stream of data to the user, even in the case of failure of the encoding server, one could replicate the task and stream the video to several servers which encode and transmit the results to the user. While this would increase the amount of transmitted data to the user's mobile, it would also increase the user experience.
- Long running simulations? [17]

3.4.5 Replication scheme

As previously mentioned, reliability will be ensured by the use of replication. More specifically, active replication will be used where each replica receives the same input, and performs the same calculations. Figure 3.3 shows how the application in fig. 3.2 looks after replicating task *S* 4 times.

In contrast to the case with one *primary* replica and several *secondary* replicas, as described in section 2.4.3, we will adapt a fan-in fan-out model, where all replicas both receive the same input, but also all transmit its result to the consumer. This allows for easy extension to also include a majority decision at the consumer to determine whether or not the correct result was received.

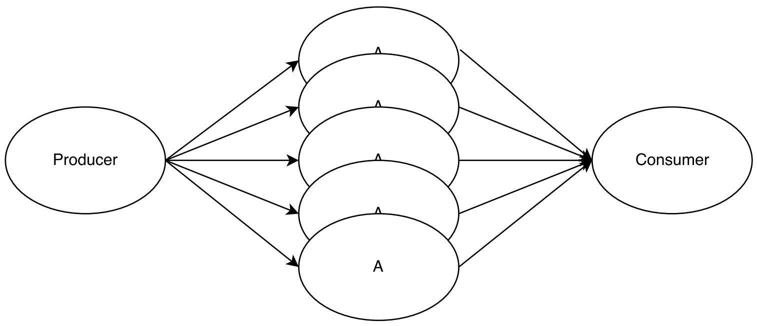

Figure 3.3: An application model where a task or service S has been replicated 4 times.

3.4.6 Fault model

In this paper, we adapt the *fail-stop* fault model, which is 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. Our model do not care whether a link died, or it was a hardware failure.

Like [33], we assume failures depends on the nodes, not the jobs running on them and the computations they perform.

Unlike the reliability models presented in XXX, we do not consider link failures in our model. However, in our experiments, link failures are indeed possible, but in the case of a link failure a node will be assumed dead, and a new replica created. Therefore, the desired reliability will still be met.

3.4.6.1 Failure distribution

We assume failures follow a Poisson process, as this seems to be widely accepted in the research community. However, while most assumes constant failure rates, our failure distributions are dynamic and monitoring of the system resources and events allows for the framework to be self-adaptive and dynamic in terms of resource failure rates.

The Poisson distribution is defined as follows:

$$P(kfailures) = \frac{\lambda^k \cdot e^{-\lambda}}{k!} \tag{3.1}$$

where λ is the failure rate, i.e. the average number of failures occurring in time t. In our case t represents the time it take to detect a node failure and spawn a new replica. From t and $MTBF$ we can calculate the failure rate λ as $\lambda = t/MTBF$. In our case k will always equal zero since we are interesting in knowing the probability that no failure occur during time t.

However, when a system first is set up and started, no knowledge about the failure rates of the system is known. Therefore, when the system is set up, we do start with assuming constant failure rates. As time goes and the time of failure for nodes are stored, one will get a more precise value for the MTBF for the nodes in the system. The time it takes to detect node failure and spawn a new replica is further discussed in section 3.5.2.

3.5 Reliability model

Here we present our main contribution, the self-adapting reliability model. But first we state some definitions of reliability.

3.5.1 Definitions

Using replication of a task, and reliability defined as in definition 2.4, the probability that a task T with n replicas is successful during a time t, corresponding to at least one replica survives, i.e. not all fail, can be expressed as

$$R_T(t) = 1 - \prod_{k=1}^{n} F_{Rep_k}(t) \tag{3.2}$$

where $F_{Rep_k}(t)$ is the probability that replica k fails during time t. Equation (3.2) corresponds to definition 2.4, i.e. at least one replica is always running.

Since we assume tasks themselves do not fail unless the resources they use fail, the reliability of a task replica is dependent on the reliability of the resources it uses. Limiting the model to node failures, either in terms of hardware failure or lost connectivity, eq. (3.2) can with eq. (2.2) be re-written as

$$R_T(t) = 1 - \prod_{k=1}^{m} F_{Res_k}(t) \tag{3.3}$$

where $F_{Res_1}(t) \cdots F_{Res_m}(t)$ are the failure probability functions of the m resources on which the n replicas are running. Using this model, reliability is only increased through replication if the replicas are scheduled on separate nodes.

Given a time t, a desired reliability level λ, and assuming the replicas are running on separate nodes, we get

$$\lambda \leq 1 - \prod_{k=1}^{n} F_{Res_k}(t) \tag{3.4}$$

which must be fulfilled by the system. Assuming that failure rates differ among resources, fulfilling eq. (3.4) is a scheduling problem, since the number of replicas needed is dependent on which resources the replicas are executed on.

The scheduling problem to fulfill a reliability λ refers to selecting n nodes on which to place n replicas such as the reliability level of the task, expressed in eq. (3.3), exceeds λ.

3.5.2 Expressing time t

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

$$t = Tf + Tr \tag{3.5}$$

where Tf is the time to detect that a node has failed, and Tr is the time it takes to spawn a new replica.

3.5.2.1 Node failure detection time

The time to detect a failed node depends on the frequency of the heartbeats. In our model, heartbeats are sent to all nodes in the system every 200 ms. A node is considered dead when no heartbeat is received for 500 ms, which is an upper bound to how long it takes to detect a node failure.

500 ms is only an upper bound since a node may fail just before sending a heartbeat, or directly after, which will affect the actual time it took to detect the failure. This corresponds to the best and worst case scenario.

If a node A dies before sending a heartbeat to B, the last received heartbeat from A was for B received 200 ms ago. The timeout on that heartbeat will timeout after 500 ms, resulting in the node A has been dead for close to 300 ms before detecting it.

In the latter case, node A dies directly after sending a heartbeat, since it will no longer send heartbeats, that one will timeout, and the time to detect it will therefore be 500 ms, which is thus an upper bound of detecting node failures.

"""Should we use the upper bound in our model, or the 'average value'???"""

3.5.2.2 Replication time

The time it takes to replicate a task depends on the time to find out on which nodes current replicas are running, and sending a replicate request to one of these nodes. Since the reliability is dependent on which nodes the replicas are running on, not solely on the number of replicas, we must also find a node which has no replica already and include this in the request. The receiving node will then send a request to that node including the current state of the task to replicate. See chapter A.

Definition 3.1. *Latency = propagation time + transmission time + queuing time + processing delay*

Definition 3.2. *Propagation time = distance / propagation speed*

Definition 3.3. *Handshake time = (ACK size / bandwidth) + (SYN+ACK size / bandwidth) + (ACK size / bandwidth)*

Definition 3.4. *Transmission time = package size / bandwidth*

Definition 3.5. *Package size = handshake + message size Handshake size = ACK + (SYN+ACK) + ACK*

Definition 3.6. *Replication time = "node dead discovery time" + "time to retrieve data from storage" + "time to send data to node" + "time to replicate"*

3.5.3 Scheduling algorithm

A simple greedy scheduling, similar to one presented in [25], which fulfills eq. (3.4) is shown below:

Algorithm 1 Greedy scheduling algorithm

```
1: $n \leftarrow 0$
2: $nodes \leftarrow available\ nodes$                    ▷ sorted by reliability, highest first
3: while $\lambda \geq \prod_{k=1}^{n} R_{Res_k}(t)$ do
4:     $node \leftarrow nodes.pop$                       ▷ take the node with highest reliability
5:     $replica \leftarrow new\ replica$
6:     PLACE REPLICA ON NODE($replica$, $node$)
7:     $n \leftarrow n + 1$
8: end while
```

This algorithm is run by a master node, selected when failure is detected. The selection process is further described in section 3.5.4.

3.5.4 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 any new replicas are needed by running the algorithm in section 3.5.3. However, when a node fails and thereby stop sending heartbeats to the other nodes in the system, the other nodes will all be aware of the failure. If all other nodes start running the algorithm, one could end up in a situation where every remaining node creates a new replica. This will of course only increase the reliability, but we will no longer ensure the optimal number of replicas being used.

To cope with this, selection process must take place to select a single node 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 managed to create any new replica. Therefore, all other nodes will send a *lost node* request to the selected node. When finished, the selected node sends a reply back whether or not the it managed to create a new replica if this was required to fulfill the required reliability. If the sending nodes does not receive a reply for some time, they assume the selected node died, and a new selection process will begin.

The algorithm is shown below:

A small example is presented in fig. 3.4. In the situation shown in the figure, there are 4 connected nodes, 1 to 4, and the other nodes have just detected the failure of node 4. Nodes 1 to 4 will select the node with the highest ID, in this case node 3, which will be responsible for determining which actions to take.

3.5.5 Self-adapting

Adaption refers to changing the behaviour depending on the changing state of the system. For a distributed system, resources most be continously monitored in order to adapt to

Algorithm 2 Handling a failed node

```
1: $nodes \leftarrow live\ nodes$                          ▷ Alive nodes sorted by ID
2: do
3:     $node \leftarrow nodes.pop$                        ▷ take the node with highest id
4:     LOST NODE REQUEST($node$)
5:     wait for reply
6:     $reply \leftarrow reply\ from\ node$
7: while $reply\ is\ not\ successful\ OR\ request\ timeout$
```

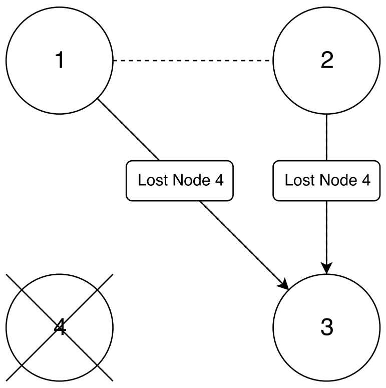

Figure 3.4: 4 interconnected nodes after nodes 1 to 3 have detected the failure of node 4.

changing behaviour [32].

To ensure a certain level of reliability, the framework must ensure both that the current state of the system is taken into account when calculating the number of replicas needed. Furthermore, the system and the running jobs must be continuously monitored, and the number of replicas must be increased is the reliability of a running job decreases below the desired level, or, the number of replicas must be decreased in order not to waste resources.

When monitoring the system, one must monitor all parameters affecting the reliability. In our case, we consider only the nodes' mean-time-between-failures, the time to detect node failures and the time it takes to spawn a new replica.

The replication time is stored per actor type. Since the state of different types of actors vary, and thereby the size of the states, the time to send the state from one node to another will also vary.

TODO: If we have time: Furthermore, we will take node CPU usage, and the time of day of the event into account in our fault model. This is based on that resource failure depends on both system load and time of day [23] [22].

3.6 Implementation

For implementing our model, we use an actor based application environment called *Calvin* [35].

3.6.1 Calvin

Calvin is a light-weight actor-based application environment for 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, is suites well for implementation of our model.

3.6.1.1 Storage

Calvin uses the Kademlia implementation of a DHT. This enables that any data stored by a node is distributed and available for all nodes in the network. DHT will not be further explained in this thesis, instead we refer to [**?**] for further information about DHT and Kademlia especially.

3.6.1.2 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 on their out-ports to other actors in-ports. The runtime, see section 3.6.1.3, 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 is the major part of the message size.

3.6.1.3 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 application. A runtime is a self-managed container for application actors and provides data transport between runtimes and actors. Each runtime has a *(*storage), all the information stored in storage is first stored locally and later flushed, i.e. distributed in a multicast approach.

3.6.2 Our contributions

Here follows a small description of the most important parts of our implementation.

3.6.2.1 Actor replication

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).

Furthermore, we implemented functionality for dynamically connecting and disconnecting actors, which allowed for dynamically adding and removing replicas of an actor.

3.6.2.2 Node Resource Reporter

We have added an actor which automatically deploys on a runtime at runtime started-up. It reports the node's CPU usage once every second to the runtime which distributes it to all connected nodes. Each node has a resource manager which stores the other nodes' usages.

3.6.2.3 Resource Manager

Besides storing the connected nodes usages the resource manager stores information about node failures. If a node fails a time stamp and the failed node's usage is stored, among with the time it took to replicate a new replica for each of the actors which were running on the failed node.

3.6.2.4 Heartbeat Actor

As each node periodically sends updates to the other nodes, it could be used as a heartbeat system. However, the messages and sent using TCP, which involved a initial handshake to setup the communication channel. For a heartbeat system, this is unnecessary overhead. Fort this reason, every runtime also starts a heartbeat actor, which periodically sends heartbeats in form of UDP messages to the other nodes. It also listens on a port and registers incoming heartbeats.

For every heartbeat sent, a timeout of 500 ms it set. When it timeouts, the node to which the heartbeat was sent is assumed dead, and when a heartbeat is received, all timeouts related to that node is canceled.

3.6.2.5 Application monitor

Each runtime also periodically monitors the applications deployed to it. We have also added an application monitor on each node that once every five seconds checks the current reliability for all actors in the applications started on the node.

If the current reliability is not achieved more replicas is created and if the reliability is unnecessary high replicas are deleted.

3.6.2.6 Failed node mechanism

When the monitor reports that a node has failed, a ... is executed. First information about all actors that were running on the failed node are fetched from storage, the actors marked for achieving a certain reliability is replicated to new nodes until the required level of reliability is reached.

Chapter 4
Evaluation

In order to validate our model, we conducted a set of experiments. The goal of the experiments were to show that our model dynamically ensures the desired level of reliability is met, despite the event of node failures and changing properties of the system, and doing so in an optimal way by choosing the most reliable nodes and thereby the minimum number of replicas.

4.1 Computational environment

As mentioned in chapter 3, the model was implemented in the IoT application environment Calvin. In order to evaluate our model, we set up an execution environment consisting of 6 servers with homogeneous hardware components. On five of these servers, two Calvin runtimes were started. These runtimes were periodically killed and restarted, simulating node failures. On the sixth server a single runtime was started, which was not killed.

The reason for having a stable runtime was to have the consumer deployed to this runtime, while the producers were deployed to the failing runtimes. The application if further described in section 4.2.

4.2 Application

The application used in the experiments is of simplest form, consisting only of one consuming and one producing actor. The producing actors produced integer numbers and sent those to the consuming actor which printed the values to standard out. Since the consuming actor is on a stable server, only the producing ones were to be replicated by the system.

The computational environment and the application is shown in fig. 4.1.

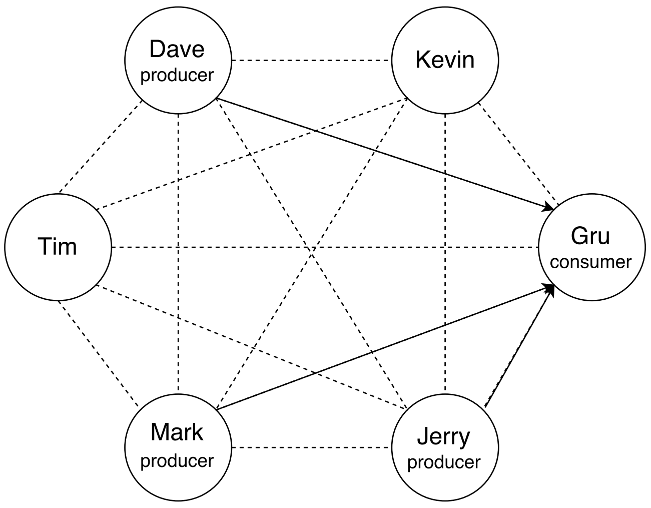

Figure 4.1: The computational environment used in the experiments. The servers in the cluster has gotten names and in the figure there is one producer replica on Dave, Mark och Jerry. Gru is the stable server which holds the consuming actor.

4.3 Experiments

4.3.1 Measurement of time t

As mentioned in section 3.5.2, the time *t* is the time it takes from that a node on which a replica dies, until a new replica is up and running, and depends on the time it takes to detect that the node died, and the time it takes to create a new replica.

4.3.2 Measurement of node failure detection time

As described in section 3.4.2, heartbeats are used to decide whether or not another node is healthy. If no heartbeat is received from a node within 500 ms, it is assumed dead.

In order to measure the time it takes to detect a failed node, an experiment was conducted in which two servers were used, each with a Calvin runtime. 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 it detected the loss of connectivity to the killed node. The failing node was killed and restarted 50 times.

Result

The average of the 50 measured times was 483.8 ms. This above the theoretical average of 400 ms, but lower than the upper bound of 500 ms.

4.3.3 Measurement of replication time

The time it takes to replicate a task, depends on the size of the task state, which have to be sent to the new node. Using Calvin, the state of a task corresponds to the state of the actor representing the task. In Calvin, the state consists mainly of the in- and output queue of values. In this experiment, we incrementally increase the state of a task, by increasing the size of its input and output queue, and measured the time it took to replicate the actor.

Each experiment is conducted using two servers, connected with a 1000 Mb/s link with a latency of less than 0.2 ms. A runtime was started on both servers, and an actor started. The actor was then replicated 10 times and the time each replication took was measured.

Result

Figure 4.2 shows how the replication time vary depending on the state size. The replication time is clearly linear in relation to the size of the state.

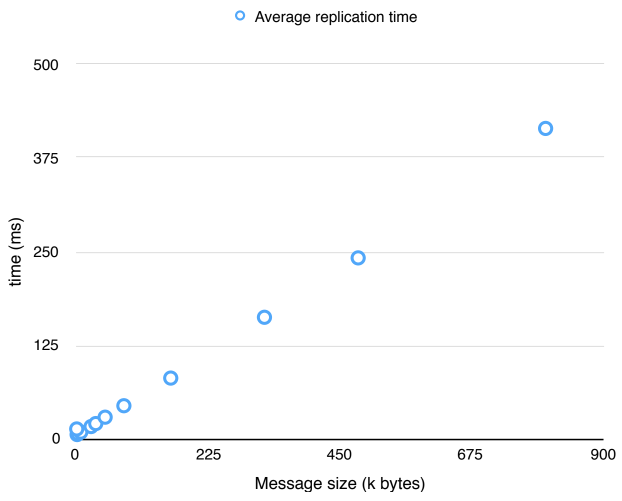

Figure 4.2: The average replication time as a function of state size.

4.3.4 Ensuring a certain reliability level

In this experiment, each runtime was given pseudo-random numbers according to a normal distribution with mean of 20 and a standard deviation of 1, i.e. the 6 runtimes all had the same MTBF, 20 seconds. Since nothing is known about failure times when the system first was started, it was assumed the MTBF for a node was 10 seconds.

After the application was started, the the reliability level of the nodes on which replicas were running was periodically measured. The desired level of reliability for the producing actors were 0.98.

The test was considered successful if the average reliability level during the duration of the test exceeded 0.98.

Result

The experiment ran for 5 minutes, and fig. 4.3 shows how the reliability vary during this period. The average reliability for the duration of the whole test was 0.997, clearly above the desired level of 0.98.

Every drop in reliability corresponds to a failure of one of the nodes on which a replica is executing.

As shown in the figure, the reliability was lower in the beginning of the test. This is due the lack of failure data for nodes, therefore assuming a MTBF of 10 seconds. As the nodes starts failing, the system learns their actual MTBF, which in the test was 20 seconds.

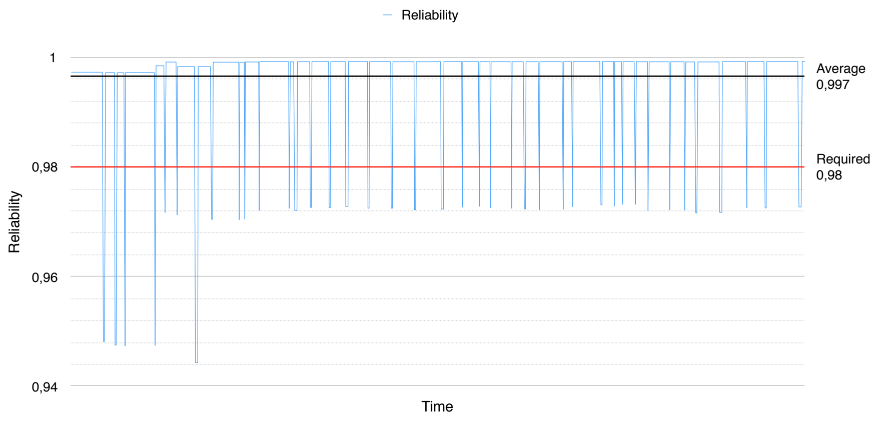

Figure 4.3: Reliability over time in a system with failing nodes.

4.3.5 Optimal number of replicas

Similar to the previous experiment, each runtime was given pseudo-random numbers according to a normal distribution. However, in this experiment, the mean value varied among the runtimes. The mean values given to the different nodes are presented in table 4.1, and all had a standard deviation of 1. Since nothing is known about the nodes when the test starts, it was assumed the mean-time-between-failure were 10.

After the application was started, the reliability level of the nodes on which replicas were running was periodically measured. The desired level of reliability for the producing actors were 0.999.

The test was considered successful if the number of replicas used decreased over time, as the system learned the actual MTBF for the nodes, and consequently more reliable nodes could be chosen. Furthermore, for the test to be successful, the average reliability for the duration of the test should exceed 0.999.

Result

The experiment ran for 5 minutes, and fig. 4.4 shows the number of replicas over time, while fig. 4.5, fig. 4.6, and fig. 4.7 show the number of replicas per node over time.

As shown in the figures, the number of replicas decreased over time, from three replicas at the start of the test, to later only needing two. Furthermore, fig. 4.5 shows that the more

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
$mark_2$	30
$jerry_1$	30
$jerry_2$	30

Table 4.1: Mean-time-between-failures for the eight non-stable runtimes in the experiment in section 4.3.5.

reliable nodes, $mark_2$, $jerry_1$, and $jerry_2$, were chosen more often than the less reliable ones, shown in fig. 4.5 and fig. 4.7.

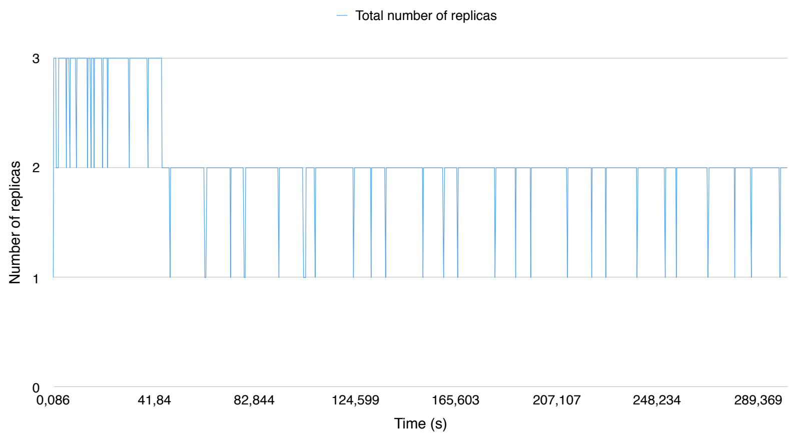

Figure 4.4: Total number of replicas.

4.3.6 Self-adaptive reliability model

In this experiment, the MTBF for the various nodes varied over time, following a $sinus - curve$. The pseudo-random numbers for the nodes followed a normal distribution with mean 25 and standard deviation of 1. However, the time to sleep between failures were calculated by 4.1.

$$T_{sleep} = t_i + 20 * \sin(2\pi T_{elapsed}/300) \tag{4.1}$$

Where t_i are the numbers following a normal distribution with mean 25 and standard deviation of 1, and $T_{elapsed}$ is the total elapsed time for the test. Given $t_i = 25$, the fig. 4.8 shows how the actual time to sleep varies over time.

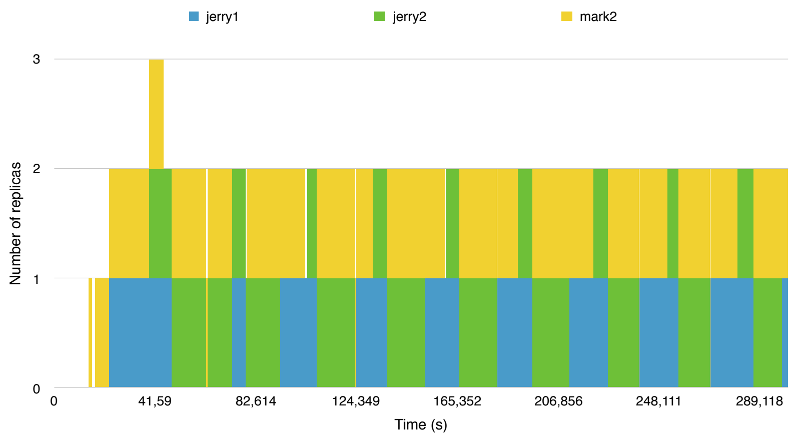

Figure 4.5: Number of replicas per node, for the three nodes with a MTBF of 30 seconds.

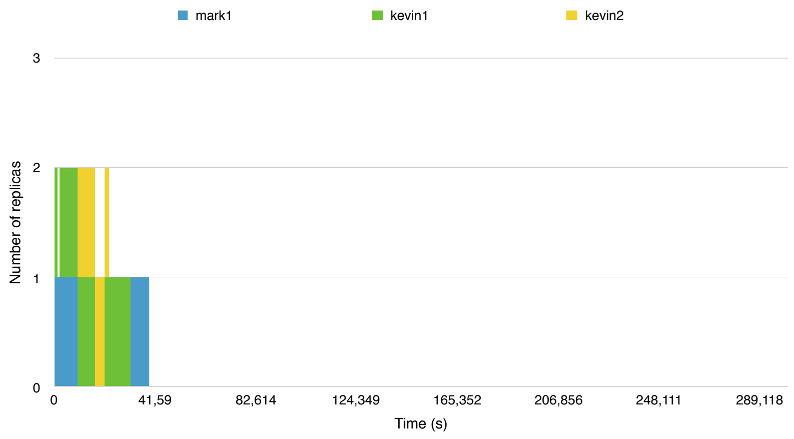

Figure 4.6: Number of replicas per node, for the three nodes with a MTBF of 15 seconds.

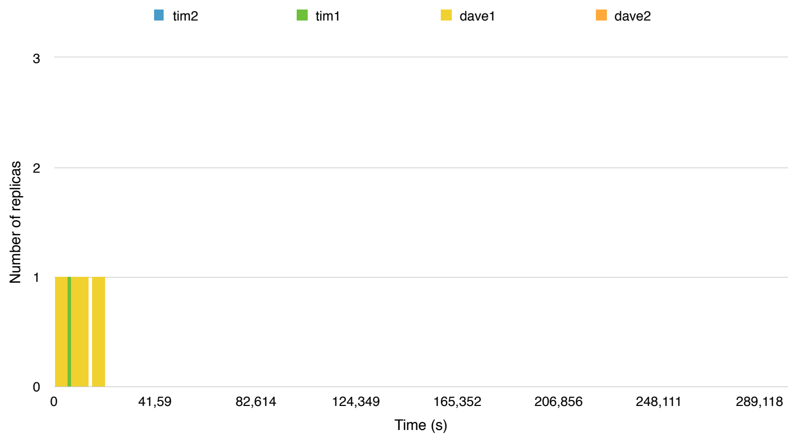

Figure 4.7: Number of replicas per node, for the four nodes with a MTBF of 7.5 seconds.

The test was considered successful if the number of replicas used increased as the time between failures decreased for the various nodes, after which it increased as the time

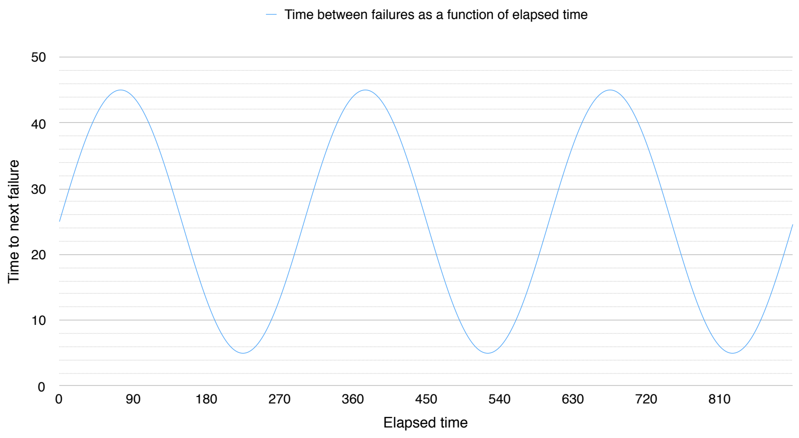

Figure 4.8: Time between failures as of eq. (4.1).

between failures increased.

Result

The test ran for 30 minutes. fig. 4.9 shows how the number of replicas varied over time, and fig. 4.10 shows how the calculated reliability for the various nodes vary over time.

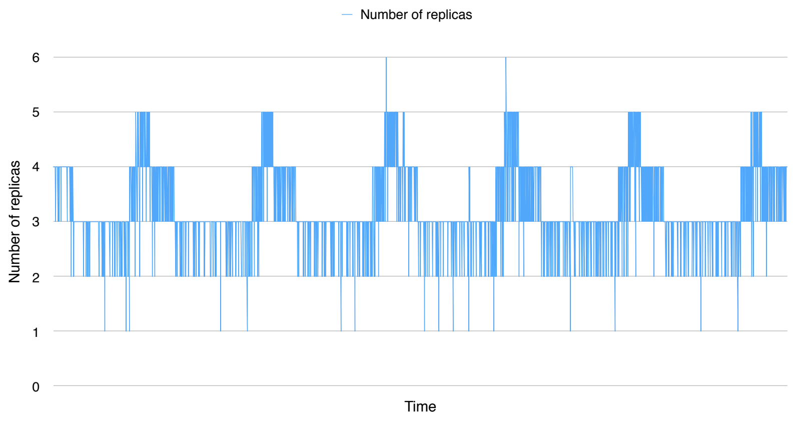

Figure 4.9: Reliability over time with failing nodes and varying time between failures.

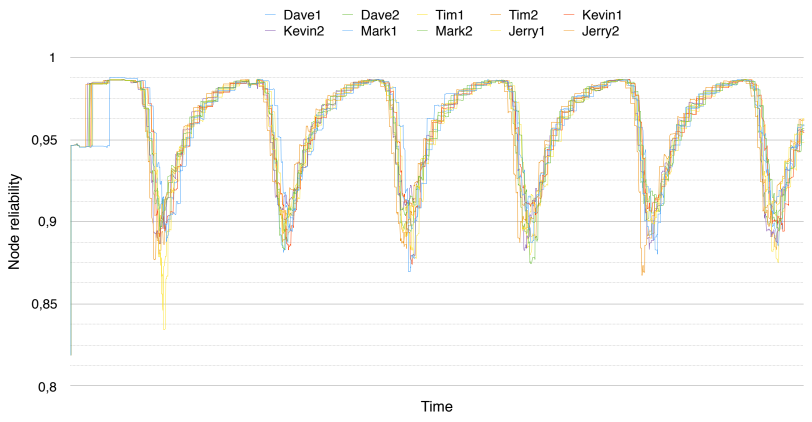

Figure 4.10: Reliability over time with failing nodes and varying time between failures.

Chapter 5
Discussion

When using a fail-stop fault model (see section 2.2.2.2), we assume that when a node dies all other nodes are aware of this. This is clearly limiting, as in the case of a link failure. Assume we have three nodes *A*, *B* and *C* and there is one replica on *A* and one on *B*. In case of a link failure between *A* and *B*, *A* will assume that *B* has failed and vice versa and both will ask the node with highest id to replicate the lost actors. If *C* has the highest id of all three then nothing will happen since *C* is aware of the replicas on both *A* and *B*. But if *A* has the highest id it will send the replication request to itself and since it's not aware of that there is a running replica on *B*, it will replicate its actor to *C*. Thereafter we have an unnecessary high reliability and, from a system perspective, there is a unnecessary high impact on the network load.

Our definition of reliability excludes the possibility of tasks calculating incorrect results, which is the case when using a Byzantine fault model. Since already using active replication to ensure reliability, it would be trivial to extend it to use consensus techniques such as majority voting or *k-modular redundancy* in order to extend the reliability definition to also include that the job calculates the correct result.

Our primary objective is to ensure a certain level of reliability. By using active replication, we require a lot more resources, which puts an extra burden on the system. In addition, the extra load on the system may affect the execution time, thus decreasing task performance.

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

Our model ensures that the reliability is higher than the required as long as we don't experience a node failure. After a node failure and before the system has recovered (replicated the lost actors) the reliability is lower than the required. Our model does not guarantee that the average reliability is higher than the required, it depends on the relationship between the MTBF and the time it takes to replicate an actor.

Finally, since using a *default MTBF* of nodes for which we have not yet experienced failure, our model is obviously limited, as this default value may be incorrect. To use in a real-world situation by a service provider, the reliability model must be extended. To cope with the MTBF, one could use past failure data from their cluster. Furthermore, the model should be extended to also take into account parameters like system load, link failures, etc.

Chapter 6
Future Work

To the best of our knowledge our model is the first of it's kind, due to its fully dynamic behavior. Therefore we had to limit our scope a lot and considering these limitations there is a lot of possible future work. Our model could be extended by considering more types of failures, e.g. link failure and soft failures (e.g. an actor dies but not the node or the actor produce an incorrect result). A consensus algorithm should be implemented in order to hamper the risk of using incorrect results. Link failures could, for instance be detected if the node responsible for replicating actors after a node failure awaits request from several nodes. If it only receives one node fail request among ten nodes its safe to assume that the node is still alive and the link to the requesting node has failed.

There are a lot of different kinds of failures which could be taken into account, see section 6.1 for an extended reliability model.

Our model could also be extended with some sort of machine learning in order to predict future failures. In the case of a video service its likely that the system load is higher in the evening and therefore also the probability of a failure. By predicting these failures replicas an be created in advance, i.e. when the probability of failure reaches above a certain threshold.

In our experiments we have had only one actor to replicate on the lost node, in bigger systems with many actors on each node we get the complexity of choosing which actor to replicate first. A solution could be to have a criticality of each actor type and start replicating the most critical actor on the lost node. However, replication time is defined as the time it takes from that we start replicating actors at all after a node failure to that the actor is up and running, i.e. all actors replication times have the same start time. This will result in that the least critical actor will have the largest replication time and thereby being having the most amount of replicas. Another, perhaps better solution would be to replicate the actors with smallest states first in order to minimize each actor's replication time, kind of an earliest deadline first algorithm. We leave the investigation of which algorithm is the optimal to future work.

6.1 Extended model

Reliability of server's availability and scalability (such as File Server, DB servers, Web servers, and email servers, etc.), communication infrastructure, and connecting devices [2].

For measuring reliability more accurate, more factors must be included (not only node failures). The overall reliability can then be expressed as:

$$R(t) = R_1(t) \cdot R_2(t) \cdots R_n(t) \tag{6.1}$$

where $R_k(t)$ is the probability that factor k is free from failures during time t. Some factors to consider are software (the program itself), OS (the device executing the program), hardware, network, electrical supply and load. The factors can be divided into static and dynamic factors. The static factors are those which does not change that frequently, such as electrical supply or hardware/software while the dynamic factors are those changing more frequently, for instance the current load (or load average for last 5 minutes etc).

Chapter 7
Conclusions

Bibliography

[1] Sol M. Shatz, Jia-Ping Wang and Masanori Goto, *Task Allocation for Maximizing Reliability of Distributed Computer Systems*, Computers, IEEE Transactions on, Volume 41 Issue 9, Sep 1992

[2] Waseem Ahmed and Yong Wei Wu, *A survey on reliability in distributed systems*, Journal of Computer and System Sciences Volume 78 Issue 8, December 2013, Pages 1243–1255

[3] A. Immonen, E. Niemelä, *Survey of reliability and availability prediction methods from the viewpoint of software architecture*, Software & Systems Modeling, p.49-65, February 2008

[4] C. A. Tănasie, S. Vîntutis, A. Grigorivici, *Reliability in Distributed Software Applications*, Informatica Economică vol. 15, no. 4/2011,

[5] Christopher Dabrowski, *Reliability in grid computing systems*, Concurrency Computation: Practice Experience, 2009

[6] Mayanka Katyal and Atul Mishra, *A Comparative Study of Load Balancing Algorithms in Cloud Computing Environment*, International Journal of Distributed and Cloud Computing, Volume 1 Issue 2, 2013

[7] Y. Dai, G. Levitin, *Reliability and Performance of Tree-Structured Grid Services*, IEEE transactions on reliability, Vol. 55, No. 2, June 2006

[8] Michael 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 Issue 1 p. 1-26, March 1999

[10] Soonwook Hwang and Carl Kesselman, *Grid Workflow: A Flexible Failure Handling Framework for the Grid*, High Performance Distributed Computing, 2003. Proceedings. 12th IEEE International Symposium on, p. 126-137, June 2003

[11] Prashant D. Maheta , Kunjal Garala and Namrata Goswami, *A Performance Analysis of Load Balancing Algorithms in Cloud Environment*, Computer Communication and Informatics (ICCCI), 2015 International Conference on, Jan 2015

[12] Shuli Wang et. al. *A Task Scheduling Algorithm Based on Replication for Maximizing Reliability on Heterogeneous Computing Systems*, Parallel & Distributed Processing Symposium Workshops (IPDPSW), 2014 IEEE International, p. 1562 1571, May 2014

[13] Michael R. Lyu *Reliability Engineering: A Roadmap*, Future of Software Engineering, FOSE '07, p. 153-170, 23–25 May 2007

[14] Guessoum Z. et. al. *Dynamic and Adaptive Replication for Large-Scale Reliable Multi-agent Systems*, Lecture Notes in Computer Science pp 182-198, April 2003

[15] F. Cao , M. M. Zhu, *Distributed workflow mapping algorithm for maximized reliability under end-to-end delay constraint*, The Journal of Supercomputing, Vol. 66, Issue 3, p. 1462-1488, December 2013

[16] A. Dogan, 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, March 2002

[17] H. Wan, H. Z. Huang, J. Yang, Y. Chen, *Reliability model of distributed simulation system*, Quality, Reliability, Risk, Maintenance, and Safety Engineering (ICQR2MSE), 2011 International Conference, June 2011

[18] C. S. Raghavendra, S. V. Makam, *Reliability Modeling and Analysis of Computer Networks*, IEEE Transactions on Reliability Vol. 35, Issue. 2, June 1986

[19] Y. Dai, B. Yang, J. Dongarra, G. Zhang *Cloud Service Reliability: Modeling and Analysis*, in PRDC, 2009

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

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

[22] B. Schroeder, G. Gibson, *A large-scale study of failures in high-performance computing systems*, IEEE Transactions on Dependable and Secure Computing (Volume:7, Issue: 4), November 2010

[23] Y. Zhang, M. S. Squillante, A. Sivasubramaniam, R. K. Sahoo, *Performance Implications of Failures in Large-Scale Cluster Scheduling*, Job Scheduling Strategies for Parallel Processing, Volume 3277 of the series Lecture Notes in Computer Science pp 233-252, 2005

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

[25] Antonios Litke et. al., *Efficient task replication and management for adaptive fault tolerance in Mobile Grid environments*, Future Generation Computer Systems Vol 23 Issue 2 p. 163-178, February 2007

[26] Real-time fault-tolerant scheduling algorithm for distributed computing systems

[27] Distributed Diagnosis in Dynamic Fault Environments

[28] A higher order estimate of the optimum checkpoint interval for restart dumps

[29] An analysis of factors affecting software reliability

[30] SLA-aware Resource Scheduling for Cloud Storage

[31] An Algorithm for Optimized Time, Cost, and Reliability in a Distributed Computing System

[32] Improving reliability in resource management through adaptive reinforcement learning for distributed systems

[33] Self-Adapting Reliability in Distributed Software Systems

[34] Scheduling Fault-Tolerant Distributed Hard Real-Time Tasks Independently of the Replication Strategies

[35] Per Persson, Ola Angelsmark, *Calvin - Merging Cloud and IoT*, Procedia Computer Science, Volume 52, p. 210–217, 2015

[36] Task allocation for maximizing reliability of a distributed system using hybrid particle swarm optimization

[37] Optimal Resource Allocation for Maximizing Performance and Reliability in Tree-Structured Grid Services

[38] Matching and Scheduling Algorithms for Minimizing Execution Time and Failure Probability of Applications in Heterogeneous Computing

[39] Safety and Reliability Driven Task Allocation in Distributed Systems

[40] Improved Task-Allocation Algorithms to Maximize Reliability of Redundant Distributed Computing Systems

[41] Design of a Fault-Tolerant Scheduling System for Grid Computing

[42] Evaluation of replication and rescheduling heuristics for grid systems with varying resource availability

[43] Fault-Tolerant Scheduling Policy for Grid Computing Systems

[44] Adaptive Task Checkpointing and Replication: Toward Efficient Fault-Tolerant Grids

[45] The decision model of task allocation for constrained stochastic distributed systems

[46] Performance and Reliability of Non-Markovian Heterogeneous Distributed Computing Systems

[47] A Hierarchical Modeling and Analysis for Grid Service Reliability

[48] Reliability analysis of distributed systems based on a fast reliability algorithm

[49] Reliability and Performance of Star Topology Grid Service with Precedence Constraints on Subtask Execution

[50] A Software Reliability Model for Web Services

[51] A study of service reliability and availability for distributed systems

[52] Efficient algorithms for reliability analysis of distributed computing systems

[53] Evaluating the reliability of computational grids from the end user's point of view

[54] A Generalized Algorithm for Evaluating Distributed-Program Reliability

[55] Real-Time Distributed Program Reliability Analysis

[56] Collaborative Reliability Prediction of Service-Oriented Systems

[57] Fault Tolerance Techniques in Grid Computing Systems

[58] Fault Tolerance Challenges, Techniques and Implementation in Cloud Computing

[59] Improving Performance via Computational Replication on a Large-Scale Computational Grid

[60] Adaptive Replication in Fault-Tolerant Multi-Agent Systems

[61] Improving Fault-Tolerance by Replicating Agents

[62] Towards Reliable Multi-Agent Systems: An Adaptive Replication Mechanism

[63] Fault Tolerant Algorithm for Replication Management in Distributed Cloud System

[64] Experimental Assessment of Workstation Failures and Their Impact on Checkpointing Systems

[65] A Survey on Scheduling and the Attributes of Task Scheduling in the Cloud

[66] Scheduling Optimization in Cloud Computing

[67] David Strauss, *Containers—Not Virtual Machines—Are the Future Cloud*, http://www.linuxjournal.com/content/containers%E2%80%94not-virtual-machines%E2%80%94are-future-cloud, Jun 2013

[68] *Probabilistic Model-Driven Recovery in Distributed Systems*

Appendices

Appendix A
Figures

Figure A.1 and fig. A.2 shows the computational environment of four nodes and one actor 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 nodes reliability is given under their name. fig. A.3 shows the communication flow after a node failure.

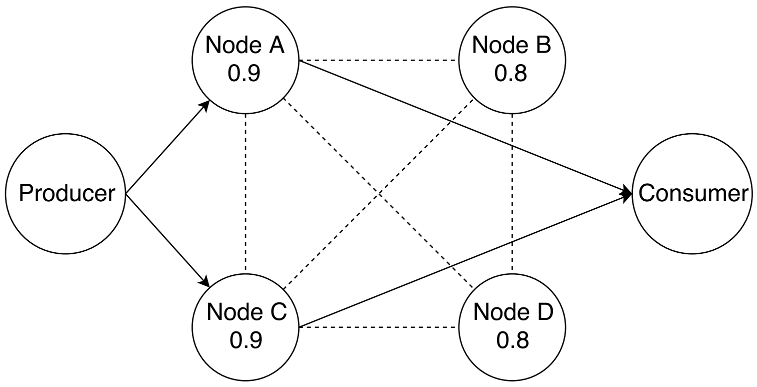

Figure A.1: Before a node failure. Here two replicas (placed on Node A and C) is required to achieve required reliability.

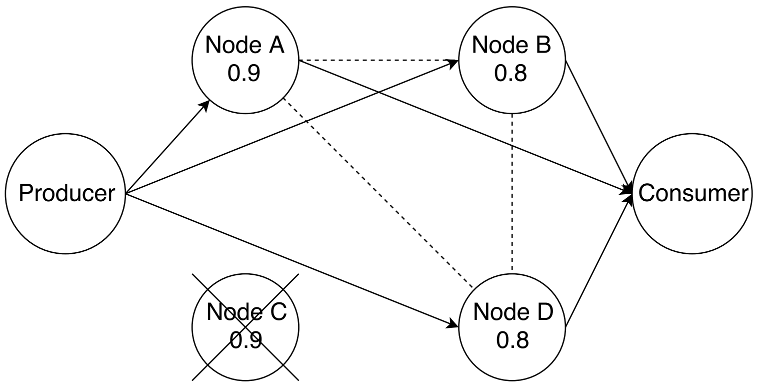

Figure A.2: After Node C has failed. Now three replicacs (placed on Node A, B and D) is reuiqred to achieve reuiqred reliability.

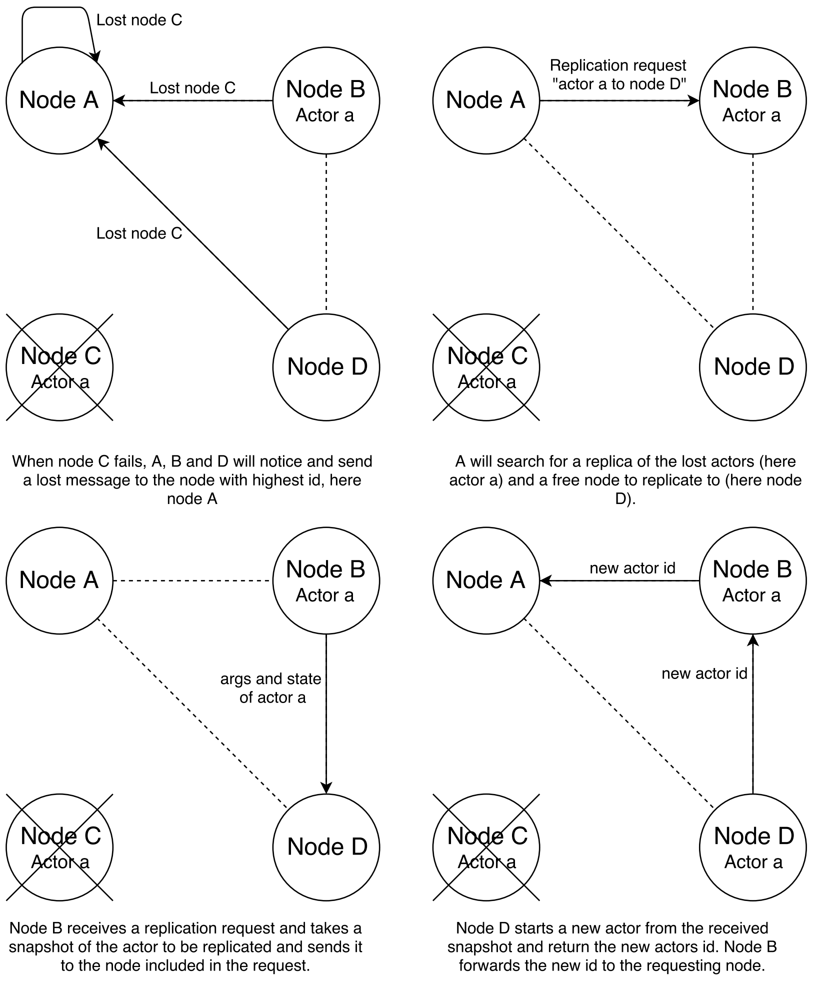
Lost node C
Node A
Lost node C
Node B
Actor a
Lost node C
Node C
Actor a
Node D
When node C fails, A, B and D will notice and send a lost message to the node with highest id, here node A
Node A
Replication request "actor a to node D"
Node B
Actor a
Node C
Actor a
Node D
A will search for a replica of the lost actors (here actor a) and a free node to replicate to (here node D).
Node A
Node B
Actor a
args and state of actor a
Node C
Actor a
Node D
Node B receives a replication request and takes a snapshot of the actor to be replicated and sends it to the node included in the request.
Node A
new actor id
Node B
Actor a
new actor id
Node C
Actor a
Node D
Actor a
Node D starts a new actor from the received snapshot and return the new actors id. Node B forwards the new id to the requesting node.

Figure A.3: The most important parts of the communication after a node has failed.

Appendix B
Code