Dynamic Fault-Tolerance and Task Scheduling in Distributed Systems

Philip Ståhl
ada10pst@student.lu.se

Jonatan Broberg
elt11jbr@student.lu.se

March 11, 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, Duan, examiner Christian Nyberg.

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 the whole master thesis work 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 behaviour 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 incorporates 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 behaviour 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 one.

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* **??**, developed by Ericsson. While *Calvin* is mainly an environment for IoT applications,

it suites our purpose well. The model is evaluated by...

The report is structured as follows: in Section 2 all necessary background theory is provided, in Section 3 we present our model and contribution in more detail, Section 4 aims at evaluating our solution while Section 5 and 6 presents limitations in our model and future work. Section 7 concludes the report.

Chapter 2
Background Theory

2.1 Computational Environment

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.

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].

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].

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].

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 homogeneous 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

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]:

1. Job related, or

2. System related, or

3. 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].

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 analyse.

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.

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].

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

When modeling system reliability, failures are usually assumed to follow a Poisson process with a constant failure rate [15] [16] [17] [20] [21] [28] [26]. For such a model to be valid, it is assumed that failures between resources are statistically independent with a constant failure rate [15].

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].

2.2.4 Failure assumptions

Models describing the nature of failures in distirbuted computing environments, are usually based on certain assumptions, and are usually only valid under those assumptions. Common assumptions include [17] [18] [19] [51] [47] [33]:

- 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-neglectable assumption since the RMS in this case in fact is a single point of failure.

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

Definition 2.2. *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 used, the reliability can be defined as [25]

Definition 2.3. *The reliability of a process, with n task replicas, is the probability 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 Reliability Model

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.

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 2.4. *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 2.5. *The Mean-Time-Between-Failure for a component is the average time between successive failures for that component.*

The MTBF can be calculated as

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

From equation 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

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

2.3.2 Factors affecting reliability

Reliability of a system highly depends on how the system is used [3]. When modelling a system's reliability, one must take all factors into account in order to create a proper model. However, with the vast number of factors affecting a system's reliability, it is practically 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].

2.4 Fault tolerance techniques

Fault tolerance techniques are used to predict failures and take an appropriate action before failures actually occur [58]. 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 checkpointing, 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 checkpointing 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 tolerant technique, and is based on the assumptions 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 a number of different strategies for replicating a task:

- Active
- Semi-active
- 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 checkpointing is employed.

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

Definition 2.6. *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 to implement some form of consensus algorithm. 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 vote for which result is correct. These algorithms are usually use the Byzantine fault model, where a resource can crash or produce an incorrect result.

Based on achieving consensus two different redundancy (replication) strategies can be identified, traditional and progressive consensus [33]. In traditional redundancy an odd number of replicas are executes 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.

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 Load balancing

The term load balancing is generally used for the process of transferring load from overloaded nodes to under loaded nodes and thus improving the overall performance. Load balancing techniques for a distributed environment must take two tasks into account, one part is the resource allocation and the other is the task scheduling.

The load balancing algorithms can be divided into three categorize based on the initiation of the process:

- Sender Initiated - An overloaded node send requests until it find a proper node which can except its load.
- Receiver Initiated - An under loaded node sends a message for requests until it finds an overloaded node.
- Symmetric - A combination of sender initiated and receiver initiated.

Load balancing is often divided into two categorize, namely static and dynamic algorithms. The difference is that the dynamic algorithm takes into account the nodes previous states and performance whilst the static doesn't. The static load balancing simply looks at things like processing power and available memory which might lead to the disadvantage that the selected node gets overloaded [11]

A dynamic load balancing can work in two ways, either distributed or non-distributed. In the distributed case the load balancing algorithm is run on all the nodes and the task of load balancing is shared among them. This implies that each node has to communicate with all the others, effecting the overall performance of the network.

In the non-distributed case the load balancing algorithms is done by only a single node or a group-node. Non-distributed load balancing can be run in semi-distributed form, where the nodes are grouped into clusters and each such cluster has a central node performing the load balancing. Since there is only one load balancing node the number of messages between the nodes are decreased drastically but instead we get the disadvantage of the central node becoming a single-point of failure and a bottleneck in the system. Therefore this centralized form of load balancing is only useful for small networks [11].

2.6 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 and 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].

2.7 Monitoring

Heartbeat-based monitors in which lack of a timely heartbeat message from the target indicates failure. Test-based monitors which send a test message to the target and wait for a reply. Messages may range from OSlevel pings and SNMP queries to application level testing, e.g., a test query to a database object. End-to-end monitors which emulate actual user requests. Such monitors can identify that a problem is somewhere along the request path but not its precise location. Error logs of different types that are often produced by software components. Some error messages can be modelled as monitors which alert when the error message is produced. Statistical monitors which track auxiliary system attributes such as load, throughput, and resource utilization at various measurement points, and alarm if the values fall outside historical norms. Diagnostic tools such as filesystem and memory checkers (e.g., fsck) that are expensive to run continuously, but can be invoked on demand. [Probabilistic Model-Driven Recovery in Distributed Systems]

Chapter 3

Approach

3.1 System model

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

3.1.1 Computational environment

In this paper we assume a distributed cloud system, with heterogeneous hardware and interconnected nodes as shown in figure 3.1. We refer to a computational resource as a *node*.

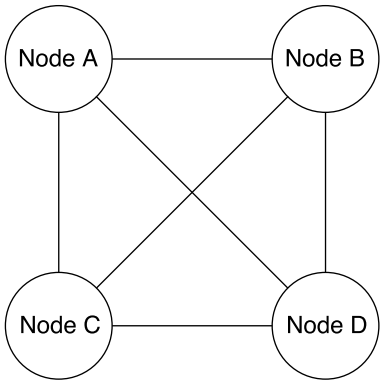

Figure 3.1: Computational environment, interconnected nodes

All information is globally accessed, hence we do not assume a single stable storage, but instead, information is shared in a multicast approach.

3.1.2 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 in a dynamic where a certain level of reliability must be met, but the reliability of the resources vary over time. Long running applications are particularly vulnerable to failure because they 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 figure 3.2.

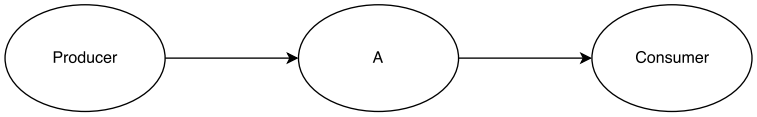

Figure 3.2: An application model where a producer transmits data to a task A, which transforms the data, and sends the result to a consumer.

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

- telephone system?
- video transcoding 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]

3.1.3 Replication scheme

We will ensure reliability based on task replication, using active replication, where each replica receives the same input, and performs the same calculations. Figure 3.3 shows how the application looks like after replication of task *A*.

3.1.4 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

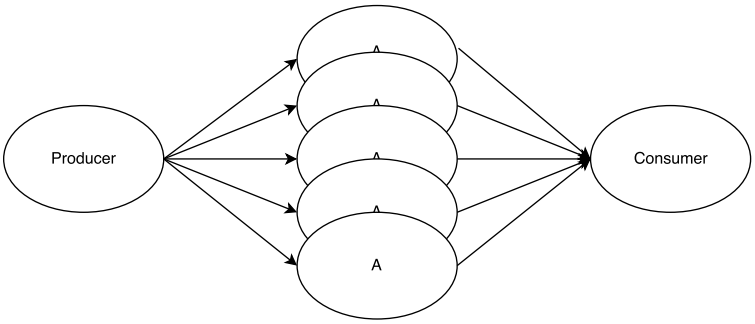

Figure 3.3: An application model where a producer transmits data to all replicas of task A, which transforms the data, and sends the result to a consumer.

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.

Failure distribution

We assume failures follow a Poission 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.

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.

Furthermore, the time it takes to replicate a task is not known at deployment time. First when several applications has been deployed, and the time to replicate tasks been measured, one can get an average replication time. The replication time will also be dynamic, and application dependent. Some tasks may take longer time to replicate than others, why a system wide average may be far from correct for a given application.

3.1.5 Reliability model

Using replication of a task, and reliability defined as in definition 2.3, 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

$$R_T(t) = 1 - \prod_{k=1}^{n} F_{Rep_k}(t) \tag{3.1}$$

where $F_{Rep_k}(t)$ is the probability that replica k fails during time t, which it takes to replicate an actor. Equation 3.1 corresponds to definition 2.3, 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, equation 3.1 can with equation 2.2 be re-written as

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

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 it takes to replicate a task, a desired reliability level λ, and assuming the replicas are running on separate nodes, we get

$$\lambda \leq 1 - \prod_{k=1}^{n} F_{Res_k}(t) \tag{3.3}$$

which must be fulfilled by the system. Assuming that failure rates differ among resources, fulfilling equation 3.3 is a scheduling problem, since the number of replicas needed is dependent on which resources the replicas are executed on.

The scheduling problem to fulfil a reliability λ refers to selecting n nodes on which to place n replicas such as the reliability level of the task, expressed in equation 3.2, exceeds λ.

3.1.6 Scheduling algorithm

A simple greedy scheduling, similar to one presented in [25], which fulfills equation 3.3 is shown below:

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

3.1.7 Self-adapting model

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 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 continously 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, once must monitor all parameters affecting the reliability. In our case, it is the node reliabilities and the time it takes to start a new replica. Node reliability is likely to vary over time, and the time it takes to start a new replica is likely to depend on the type of job.

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].

3.2 Implementation

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

3.2.1 Calvin

Calvin is a light-weight actor-based application environment for IoT applications and 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.

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 outports to other actors inports.

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.

Monitoring

The runtimes in the Calvin framework use a heart-beat technique to detect runtime connectivity.

Actor replication

We extended the Calvin framework to allow a fanout/fanin connectivity model where actors can have multiple producers (fanin) and multiple consumers (fanout). Furthermore, we implemented functionality for dynamically connecting and disconnecting actors, which allowed for dynamically adding and removing replicas of an actor.

Resource Manager

We added a resource manager actor which automatically deploys on a runtime at runtime started-up. It keeps track of the other nodes CPU usage and distributes its own usage once every second to the other nodes. It also keeps track of the connected nodes reliabilities. Due to its continuously communication with other nodes it fits great as a monitoring system. If a node stops receiving updates from a node we can assume that it has failed. But since there already is a built-in heartbeat system we will use that to detect node failures.

Chapter 4
Evaluation

4.1 Computational environment

In order to evaluate our model using Calvin, we set up an execution environment consisting of 8 servers, and started a runtime on each server. The servers were given a random time-between-failures t_f. After t_f seconds, the runtime was killed and restarted. This also maps well to the fail-stop fault model. Through the heart-beat system, the other runtimes are aware of a killed runtime.

4.2 Reliability level

The reliability level over time was measured by continuously and monitoring the reliability of the nodes on which replicas where running.

4.3 Performance metrics

Chapter 5
Limitations

Our model is obviously limited in that we do not account for link failures. However, the reliability model could easily be extended to include link failure probabilities, which then need to be taken into account in the scheduling algorithm as well.

Furthermore, using a fail-stop fault model, 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, a node may become unavailable for one node while being available for another.

In case of link failure two nodes might get unavailable of each other. Each node will assume that the other has failed and try to restart it while replicating new replicas until required reliability is achieved. Therefore, from a system perspective, there is a unnecessary high reliability which result in an 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 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.

5.0.1 Reliability model

Chapter 6
Future Work

Replication impose extra burden on the system as additional resources are needed and computational power is wasted. By combining checkpointing techniques with active replication, the number of replicas needed could possible be decreased. ([44] combines checkpointing and replication)

Predict future failures - machine learning (many parameters could be taken into account), and when the probability of failure reaches above a certain threshold, the running tasks on that node could be migrated. A high reliability in failure prediction allows for fewer replicas to be needed.

Implement consensus in Calvin system

6.0.2 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 measure reliability according to the dynamic definition (Definition **??**) we will use the following formula:

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

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 Environmen*, 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*,

[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] Calvin - Merging Cloud and IoT

[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

Appendices

Appendix A

A