Dynamic Fault-Tolerance and Task Scheduling in Distributed Systems

Philip Ståhl
ada10pst@student.lu.se

Jonatan Broberg
elt11jbr@student.lu.se

March 3, 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.

Your abstract should capture, in English, the whole thesis with focus on the problem and solution in 150 words. It should be placed on a separate right-hand page, with an additional *1cm* margin on both left and right. Avoid acronyms, footnotes, and references in the abstract if possible.

Leave a *2cm* vertical space after the abstract and provide a few keywords relevant for your report. Use five to six words, of which at most two should be from the title.

Keywords: MSc, template, report, style, structure

Acknowledgements

If you want to thank people, do it here, on a separate right-hand page. Both the U.S. *acknowledgments* and the British *acknowledgements* spellings are acceptable.

We would like to thank MAPCI and our supervisor Björn Landfeldt for there input. Shub...something for input. We also would like to thank our 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, load balancing and task scheduling in the cloud has become a very important and interesting research area. Cloud systems often consists of heterogeneous hardware and ensuring a certain level of reliability is therefore a complex task as any of the components may fail at any time.

Guaranteeing properties such as high 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. Dispite fulfilling the required level at the time of deployment, as the state of the system changes, the level may no longer be fulfilled. This speaks for a dyanmic model.

Reliability can be increased by cloning application tasks, where all clones produce the same output. This allows for both increased redundancy and an easy way of detecting errors. If the execution of one clone stops, e.g. due to hardware or link failure, it can easily be detected at the receiver side which allows for a scheduling mechanism to find a proper location for a new clone. This allows for continuing execution of the application even in case failure. Seamlessly being able to continue the execution, without losing any data is of particular interest in data stream processing.

Some of the 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, and since only the work of one task is needed, a lot of computational resources is thus wasted. Dynamic analysis of the system could help in determining on which resources one should assign the tasks to for optimal resource usage and load balancing.

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].

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. 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 [TODO reference], 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 a 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. The system is continously 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.

Finally, our solution is fully distributed and thereby avoids having a single point of failure, which may be the case in grid environments with a single Resource Management System. (THIS CONTRADICTS NOT BEING LIMITED TO A SPECIFIC DISTRIBUTED ENVIRONMENT)

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].

COPIED: Cloud computing is different from but related with grid computing, utility computing and transparent computing. Grid computing [1] is a form of distributed computing whereby a "super and virtual computer" composed of a cluster of networked, loosely-coupled computers acts in concert to perform very large tasks." 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]

COPIED: "Cluster systems are typically composed of a large number of identical, centrally managed computation nodes constructed from commodity components and linked by one or more network infrastructures such as Ethernet or Myrinet" [8].

There are a number of different types of distributed environments, e.g. grid, mobile grid, clusters, cloud and HDCS.

- Grid computing - A grid is the collection of autonomous resources that are distributed geographically and by a single domain work together to achieve a common goal, i.e. to solve a single task [6] [7].

- Mobile grid? -

- Cluster - A cluster is similar to a grid but with the difference that the 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 - Can be 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 analysing and predicting workload and reliability is usually very challenging [2].

- HDCS - Heterogeneous Distributed Computing Systems is a system of numerous high-performance machines connected in a high-speed network and therefore high-speed processing of heavy applications is achieved.

COPIED: "Cloud computing is a style of computing where service is provided across the Internet using different models and layers of abstraction [4]. It refers to the applications delivered as services [5] to the mass, " [58]

COPIED: Most of the distributed service systems can be modeled as a CHDS. This type of distributed systems consists of heterogeneous sub-systems with various operating platforms on different computers in diverse topological networks, which are managed by a control center. The heterogeneous sub-distributed systems are composed of different types of computers with various operating systems connected by diverse topologies of networks. These sub-systems exchange data with virtual machine through system service provider interface. They are connected with virtual nodes by routers [51].

COPIED: The grid computing system has emerged as an important new field, distinguished from conventional distributed computing systems by its focus on large-scale resource sharing, innovative applications, and, in some cases, high performance orientation. [47].

2.1.2 Dynamic versus static environments

Based on how the environment is configured it can be either static or dynamic.

In a static environment only homogeneous resources are installed. 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 [6].

In a dynamic environment heterogeneous resources are installed. 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 [6].

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]. The possible kind of errors in a Grid environment can be divided into the following three kinds [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 servers responds correctly but beyond the specified time interval

Failures are usually considered to be either

1. Job related, or
2. System related, or
3. Network related [53].

In [22], almost ten years of real-world failure data of 22 high performance computing systems is studied. The failures root-causes is divided into five categorize, Human, Environment, Network, Software and Hardware failure. The computing system can be divided into different hardware types, based on processor/memory chip model. From the data it is seen that hardware failures was the single most common type of failure, ranging from 30 to more than 70 % depending on hardware type. The second most common failure was software failure where ranging from approximately 10 - 20 %. But the root cause of 20 - 30 % of the failures are unknown, which for a majority of the hardware types leaves the possibility that any of the three other types of failure could be the second most common failure.

When talking about incorrectly behaviour of software three terms are widely used: failure, fault and error. For better understanding we give a definition for each of them.

- Failure: Is defined as the event, e.g. when the software produces the wrong result or when a server crashes. Or simply the inability to perform as required.
- Fault: A fault is the state of software. It is also called defect/bug.
- Error: An error is a human mistake.

To give a clear picture of the definitions above you can say that when a human writes code and does a mistake an error is committed and a fault is introduced in the code. Later, when the fault effects the execution of the software in an undesired way we have experienced a failure.

2.2.2 Fault life-cycle techniques

According to [13] the following four technical areas can be regarded as fault life-cycle techniques.

- Fault prevention - The initial mechanism for a reliable system, trying to avoid faults already in the development phase. A fault which isn't committed cost nothing to fix.
- Fault removal - The next line of defence, trying to detect and eliminate faults by validation and validation. For instance by testing and inspection.
- Fault tolerance - The last line of defence against faults which are undetected by fault removal. Can be obtained by catching faults as exceptions and/or by redundancy.
- Fault forecasting - Estimate the presence of faults during design phase or before actual deployment.

Fault tolerance and fault forecasting is considered the most important fault life-cycle techniques since they are the last line of defence against failures that haven't been prevented or detected.

2.2.3 Fault models

Byzantine Faults

COPIED: The Byzantine fault model is the most adversarial model. It 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 making it very difficult to analyse.

Fail-stop Faults

In comparison to the Byzantine fault model the fail-stop model is much simpler. When a node fails it stops producing output and its interaction with the other nodes [9]. This results in that the rest of the system automatically gets aware of that the node has failed. It is a simple model and does not handle subtle failures such as memory corruption but rather failures such as system crash or 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 but with the difference that the other nodes does not get aware of that the node has failed.

2.2.4 Failure Distribution

The failure rate for a hardware component is usually modelled with a bathtub shaped curve. The failure rate is 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. Failure rate for software on the other hand is often assumed to only drop over time since bugs are fixed and removed by time. If updates etc. are installed the failure rate might increase if new bugs was inserted, however software does not get worn out so an overall drop is expected. But when considering resource failure it is usually assumed to follow a Poisson process with a constant failure rate [15] [16] [17] [20] [21] [28]. For this model to be valid, it is assumed that failures between resources are statistically independent [15].

Assumptions such as constant resource failure rate and statistically independent failures, are not likely to model the actual failure scenario of a dynamic heteregenous distributed system [16]. [2] show that the probability of hardware failure are greater in the beginning and end of lifetime. Furthermore, by studying failures in high-performance computing systems, [22] found a Poisson distribution to be a poor fit, while a normal or lognofmrla distribution fitted the data much better. Furthermore, as failures are likely to be correlated [21], the probability of failure increases with the number of components on which the job is running.

Hardware failures do not usually have a constant rate of failure. Instead, hardware is less reliable and more likely to fail in the beginning and of its life-cycle [2].

Fault may propagate throughout the system, thereby affecting other components as well [5].

COPIED: failures is modeled well by a Weibull distribution (Weibull shape parameter of 0.7–0.8) with decreasing hazard rate. [22].

COPIED: "The model that is used is based on the Weibull distribution, since this model effectively represents the machine availability in large-scale computing environments [35] such as wide-area enterprises, the Internet and Grids." [25].

COPIED: correlation between the failure rate of a machine and the type and intensity of the workload running on it. [22].

COPIED: “where failure rates vary significantly depending on a node’s workload… the front-end nodes, which run a more varied, interactive workload, exhibit a much higher failure rate than the other nodes “ [22]

COPIED: “Failure rates are high initially, and then drop significantly during the first months. “ “Number of failures by hour of the day (left) vary” [22]

COPIED: "During peak hours of the day the failure rate is two times higher than at its lowest during the night. Similarly the failure rate during weekdays is nearly two times as high as during the weekend. We interpret this as a correlation between a system’s failure rate and its workload" [22]

COPIED: “In fact, it has been shown in [28] that there are strong spatial correlations between failures and nodes where a small fraction of the nodes incurs most of the failures. Possible reasons include: (1) some components (both hardware and software) are more vulnerable than others [37]; and (2) a component that just failed is more likely to fail again in the near future [13]. “ [23]

COPIED: “studies have pointed out that the likelihood of failures increases with the load on the nodes [2]. At the same time, there have also been other studies [3] showing that load on clusters exhibit some amount of periodicity, e.g. higher in the day/evenings, and lower at nights." [23]

COPIED: "“nodes that have failed in the past are more likely to fail again. We propose a scheduling strategy (rather a node assignment strategy for jobs), called Least Failure First (LFF), to take advantage of this observation. " [23]

2.2.5 Failure assumptions

Common assumptions about failures include [2].

- Not all factors are considered
- Hardware failure is considered as constant

Furthermore, another common assumption is statistically independent failures. However, this is very unlikely to reflect the real dynamic behaviour of distributed systems [2] [19].

Models which describes the reliability of distributed systems, or distributed applications, usually make assumptions such as [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

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-neglected assumptions since the RMS in this case in fast is a single point of failure.

2.3 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 cal be divided into four different categories [2]:

1. Fault prevention - elimination of errors before thay 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

Limiting the span to already developed and deployed applications, fault-tolerance consists of [5]:

- detecting fault and failures
- recovery to allow computations to continue

Fault tolerance usually involves one of the following techniques [5]:

1. checkpointing
2. replication
3. rescheduling

Fault-tolerance techniques can be divided into two categorizes: Task-level techniques and work-flow level techniques. Task-level techniques refer to those techniques which are applied in task-level to mask the effect of task crash failures. Work-flow techniques are those techniques that basically allow alternative task to launch to deal with user-defined exceptions and the failures the task-level techniques fail to cover [10].

Fault tolerance techniques can also be divided into reactive and proactive techniques. A reactive fault tolerant technique reacts when a failure occur and tries to reduce the effect of the failure. The proactive technique on the other hand tries to predict failures and proactively replace the erroneous components. COPIED: Software reliability prediction is a task to determine future reliability of software systems based on the past failure data [17]. “ [56].

One way of taking failures into account is to employ a reliable matching and scheduling algorithm in which tasks of an application are assigned to machines to minimize the probability of failure of that application. This type of approach was followed for allocating undirected task graphs to nonredundant and redundant real-time distributed systems [8], [9], [10], [11].

The basics, however, are checkpoint recovery and task replication The former is a common method for ensuring the progress of a long-running application by taking a checkpoint, i.e. saving its state on permanent storage periodically. A checkpoint recovery is an insurance policy against failures. In the event of a failure, the application can be rolled back and restarted from its last checkpoint—thereby bounding the amount of lost work to be re-computed. Task replication is another common method that aims to provide fault tolerance in distributed environments by scheduling redundant copies of the tasks, so as to increase the probability of having at least a simple task executed [25].

In case of a long running task, checkpoint periodically and restart from the last good state; In case of a task running on unreliable execution environments, have multiple replicas of the task run on different machines, so that as long as not all replicated tasks fail, the task will succeed to execute; [10].

2.3.1 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.

The basics, however, are checkpoint recovery and task replication. The former is a common method for ensuring the progress of a long-running application by taking a checkpoint, i.e. saving its state on permanent storage periodically. A checkpoint recovery is an insurance policy against failures. In the event of a failure, the application can be rolled back and restarted from its last checkpoint—thereby bounding the amount of lost work to be re-computed [25]

2.3.2 Replication

Job replication is a commonly used fault tolerant techinique, 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 identifical 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 minimzes the risk of failurs 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 determien 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.

COPIED: “Active replication removes the centralized control of primary backup and minimizes losses that occur when some replicas fail. Active replication incurs a high cost associated with keeping all replicas synchronized [50]. iterative redundancy complements active replication by specifying, at runtime, how many replicas should exist to guarantee a particular level or reliability“ [33].

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

Definition 2.1. *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].

2.3.3 Rollback recovery

COPIED: "Log-based rollback recovery protocols, or message logging protocols, supplement normal checkpointing with a record of messages sent by and received by each process. If the process fails, the log can be used to replay the progress of the process after the most recent checkpoint in order to reconstruct its previous state. This has the advantage that process recovery results in a more recent snapshot of the process state than checkpointing alone can provide" [8].

Consensus

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 vote for which result is correct. This allows for detecting of Byzantine faults, i.e. when a node crashes 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.

2.4 Reliability

Reliability in the context of software applications can have several meanings, especially for applications running in distributed systems.

Generally, reliability is defined as the probability that the system can run an entire task successfully [1] [17] [51] [47] [54] [55] [33]. A similar and common definition of reliability in distributed environments is that 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] [2], 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].

[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 used, the reliability can be defined as [25]

Definition 2.4. *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 it takes start a new replica of the first dying one.*

2.4.1 Reliability Model

In order to determine the reliability of a system, one need to take all factors affecting the reliability into account [2].

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] [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.*

2.4.2 Factors affecting reliability

Reliability of a system highly depends on how the system is used [3].

COPIED: overloaded resources leaving no reserve capacity for variations in demand · lack of flexibility in design that cause serious consequences for schedules and costs [4]

COPIED: in fact, the reliability of individual element is affected by various conditions such as failure rate, amount of data, bandwidth, operation time, etc [19].

COPIED: the operational probabilities are affected by various conditions such as failure rate, transmitted data, available bandwidth, and operation time [47]

2.4.3 Providing Reliability

Two techniques can be used in order to ensure reliability [2]:

- Predicting reliability early at architectural level to better analyze the system before actual application development.
- Providing a fault tolerant system to ensure seamless environment for end users in case if any unanticipated or unpredicted scenario has occurred during execution in real environment.

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].

We will briefly mention some dynamic load balancing algorithms:

- Central Queue Algorithm A non-distributed algorithm where the central manager maintains a cyclic FIFO-queue. Whenever a new activity arrives the manager inserts it into the queue and whenever a new request arrives it simply picks the first activity in the queue.
- Local Queue Algorithm When a new task is created on the main host it will be allocated on under-loaded nodes. Afterwards all new tasks are allocated locally since ... [11]. The algorithm is receiver initiated since when a node is under-loaded it randomly sends requests to remote load managers
- Ant Colony Optimization Algorithm As the name implies this algorithm is inspired by the behavior of real ants to find a optimal solution. Whenever an ant find food it moves back to the colony while leaving "markers", i.e. laying pheromone on the way. When more ants find the same place the path the pheromone will become denser.
- Honey Bee Foraging Algorithm This algorithm is quite similar to the Ant Colony Optimization algorithm. When bees find food they return to the bee's colony and use special dance movements for informing the other bees of how much food there is and where it's located. When the forager bees find more food a more energetic dance takes place. This phenomenon can be applied to servers, when an overloaded server receives a request it redirects it to other under loaded servers.
- Bidding An overloaded node request bids from other nodes. The node with the best bid (i.e. lowest load) wins the job.
- Max-Min
- Min-Min

2.6 Task scheduling

[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]

The problem of finding an optimal task allocation with maximum system reliability has been shown to be NP-hard; thus, existing approaches to finding exact solutions are limited to the use in problems of small size. This paper presents a hybrid particle swarm optimization (HPSO) algorithm for finding the near-optimal task allocation within reasonable time. [Task allocation for maximizing reliability of a distributed system using hybrid particle swarm optimization]

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 modeled 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]

Globus [9] provides a heartbeat service to monitor running processes to detect faults [25]

Nodes can alternate between working correctly and being crashed in our model.3 Hence, the status of a node is modeled by a state machine with two states, failed and working. Failed nodes do not send messages nor do they perform any computation. Working nodes execute faithfully the diagnosis procedure. In this section, we derive lower bounds on the diagnostic latency, start-up time, and state holding time achievable by any heartbeat-based diagnosis algorithm in completelyconnected networks. The maximum time between two consecutive heartbeats arriving from a continuously working node at any other node in the system sets a limit on how early failed nodes can be identified by the absence of a heartbeat. [27]

Chapter 3
Approach

3.1 Method

We have decided that use an action research oriented methodology. First we studied today's situation from literature and with input from our supervisor Björn Landfeldt. Therefrom we could state the background and motivation to the problem to be solved. The second step was to suggest a proper solution and implement it. Here we got input from literature and our supervisor among with Jörn och Shub. The third and most important step was to evaluate the solution in an objective way.

First of all we put together the necessary master thesis documents, a goal document and a project plan.

Thereafter we began our literature study of today's situation, as seen in 1.2 there is an certain interest in this subject...etc

During the first weeks Philip implemented a replication functionality in Calvin while Jonatan started writing the report and investigated how reliability can be defined. When the replication functionality was up and running Jonatan started writing an "actor lost" command which and deleted all the information about the lost actor used the replication functionality to start a new replica.

3.2 System model

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

3.2.1 Computational environment

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

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

3.2.2 Monitoring

Heart beat monitoring detects when a connectivity to a node is lost. In the case of lost connectivity, the node is assumed dead.

COPIED: "“The most basic form of monitoring is a simple heartbeat system. A monitor process listens for periodic messages from the monitored components. The message simply indicates that the component continues to function correctly enough to send messages. " [8].

3.2.3 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 simpel example application in our experiments. The application can be modeled as shown in figure 3.1.

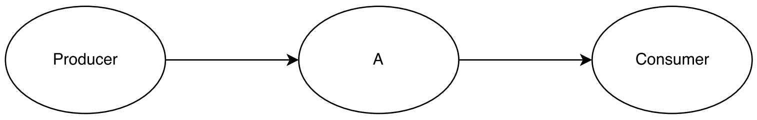

Figure 3.1: An application model where a producer transmits data to a task A, which transforms the data, and sends the result to a consumer.

"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 continous 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 inrease the amount of transmitted data to the user’s mobile, it would also increase the user experience.
- Long running simulations? [17]

COPIED: “Services in large-scale distributed environments (e.g., The Internet of Things [15]) are required to stay and continue operating even in the presence of malicious and unpredictable circumstances; that is, their processing capacity must not be significantly affected by the user requirements [11,8] “ [32].

COPIED: “Most workloads are large-scale long-running 3D scientific simulations, e.g. for nuclear stockpile stewardship. These applications perform long periods (often months) of CPU computation, interrupted every few hours by a few minutes of I/O for checkpointing. “ [22]

COPIED: “given the long execution times of many of the parallel applications that we are targeting – those in the scientific domain at national laboratories and supercomputing centers. “ [23].

COPIED: "The end result is that long-running, distributed applications are interrupted by hardware failures with increasing frequency" [8].

3.2.4 Replication scheme

In our framework, we ensure reliability based on task replication. We use active replication, where each replica receives the same input, and performs the same calculations. Figure 3.2 shows how the application looks like after replication of task *A*.

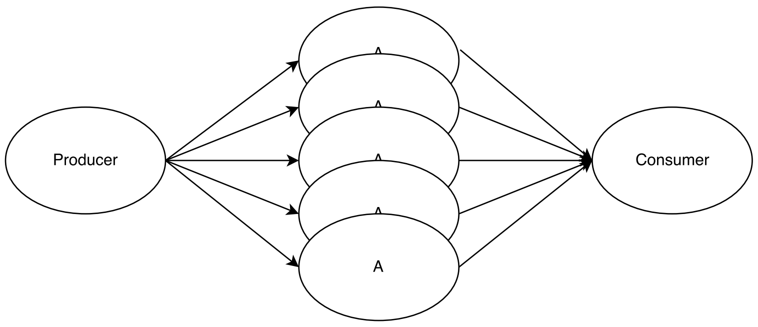

Figure 3.2: An application model where a producer transmits data to all replicas of task A, which transforms the data, and sends the result to a consumer.

3.2.5 Failure model

In our framework, we assume the *fail-stop* fault model, which is commonly used when presenting fault-tolerance techniques [8]. Nodes in the system have one of two states: *running* or *dead*. If a node dies, 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.

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 put into action (use other words), no knowledge about the failure rates of the system is known. Therefore, when the system is set up, we to start with assume 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 replica 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.

COPIED: "we assume nodes' failures depend on the nodes, and not on the computation they perform "[33]

3.2.6 Reliability model

Since resources, e.g. computational nodes in the system, will be restarted upon failure, we use MTBF as defined in 2.6, which can be calculated as

$$MTBF = (number\ of\ failures)/(total\ time) \tag{3.1}$$

From equation 3.1 a reliability function can be expressed as

$$R_r(t) = e^{-t/MTBF_r} \tag{3.2}$$

Equation 3.2 expresses the probability that a given resource will work for a time t.

The probability that a resource will fail during a time t is therefore

$$Rf_r(t) = 1 - e^{-t/MTBF_r} \tag{3.3}$$

For a complete analysis of system reliability, one must take all factors affecting reliability into account. However, including all factors if unfeasible. [29] lists 32 factors affecting the reliability of software, excluding environmental factors such as hardware and link failure. In our framework, we only care whether or not a node is available or has died.

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

$$R_T(t) = 1 - \prod_{k=1}^{n} Rf_{Rep_k}(t) \tag{3.4}$$

where $Rf_{Rep_k}(t)$ is the probability that replica k fails during time t, which it takes to replicate an actor. Equation 3.4 corresponds to definition 2.4, i.e. at least one replica is always running.

Assuming 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 to model to node failure, either in terms of hardware failure or lost connectivity, equation 3.4 can with equation 3.2 be re-written as

$$R_T(t) = 1 - \prod_{k=1}^{m} Rf_{Res_k}(t) \tag{3.5}$$

Where $R_{Res_1}(t) \cdots R_{Res_m}(t)$ are the reliabilities 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

$$\lambda \leq \prod_{k=1}^{n} R_{Res_k}(t) \tag{3.6}$$

which must be fulfilled by the system. Since assuming hetereogenous and non-constant failure rates for the various nodes, fulfilling equation 3.6 is a scheduling problem.

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 3.5, exceeds λ.

3.2.7 Self-adapting model

Adaption refers to changing the behaviour depending on the changing state of the system.

COPIED: "adaptation to changing conditions is achieved by both adaptive scheduling and adaptive execution" [53].

COPIED: "the distributed systems require consistent and iterative monitoring for valuation resources' behaviors and processing requirements. Therefore, an autonomous, scalable and highly dynamic learning approach is deserved [32]

COPIED: "The individual component can be monitored in real time and updates the parameters dynamically for the exponential distribution. The monitored information is simple: just the number of failures over the total running time of this component which has actually been recorded by log files in today's grids. Such a dynamic updating scheme can further validate the exponential assumption, though we may relax the above assumption somewhat to allow reasonable or gradual change in the failure rate (such as wear out) because, during a short enough period of service time, the parameter cannot change too much and using the latest value should be a good approximation " [47].

COPIED: "in the proposed system, we need to determine the degree of over-provisioning or job replicas as small as possible in order to minimize the system overhead " [41].

3.3 Implementation

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

The Calvin framework consists of *runtimes* on which one deploy applications consisting of connected *actors*. We will use this model to reflect our system of nodes and application tasks. We will deploy a single runtime per node, and have an actor representing the task we will replicate.

3.3.1 Scheduling algorithm

A simple greedy scheduling, similar to one presented in [25], which fulfills equation 3.6 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
```

Chapter 4
Evaluation

4.1 Performance metrics

performance metrics

COPIED: performance metrics: The performance of fault tolerance techniques is measured by standards metrics turnaround time, throughput, fail tendency and grid load. The parameters are defined as: Turnaround time, Throughput, Fail tendency, Grid load“ [57].

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 algorihtm 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 availble for another.

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

Many engineers and researchers base their reliability models on the assumption that components of a system fail in a statistically independent manner. This assumption is often violated in practice because environmental and system specific factors contribute to correlated failures, which can lower the reliability of a fault tolerant system. A simple method to quantify the impact of correlation on system reliability is needed to encourage models explicitly incorporating correlated failures. Previous approaches to model correlation are limited to systems consisting of two or three components or assume that the majority of the subsets of component failures are statistically independent. [24]

the interrelationship constraint between task modules has not been taken into account in calculation of distributed system reliability (DSR). In distributed simulation, the LPs simulation advances are bound by synchronization constraint, which will affect the executive time of system components. [17]

We do not take into account parameters such as X and Y [TODO].

Furthermore failure data which they have collected for their experiments through in-house testing cannot be compared with failures that can occur under actual operational environment [2]

A lot of researchers are of the view that service providers must provide some other detail to compute reliability of both type of services i.e. atomic service and composite service. Such as, external services it uses, how service are glued together in composite service, how frequently they call each other, and flow graph describing behavior of service [2]. Service Oriented Reliability Model (SORM) [18] computed reliability of atomic and composite services exploiting distinct technique [2]

Given the failure probability Pfik of each one of the mi replicas Ti k of task Ti , the new failure probability P fi0 for task Ti is: 0 mi Y Pfi =Pfi· Pfik. (2) k=1 The above corresponds to the probability of the event “all the replicas and the original task fail”. Respectively, the success probability is equal to the probability of the event “the original task or at least one of its replicas executes successfully”. The number of replicas issued depends on the failure probabilities of the original tasks and on the desired fault tolerance level in the Grid infrastructure. [25]

Chapter 6
Future Work

Replication impose extra buden 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

6.0.1 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 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). For simplicity we will use one reliability for all static factors and chose a number of dynamic factors to consider.

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] Survey of reliability and availability prediction methods from the viewpoint of software architecture

[4] Reliability in Distributed Software Applications

[5] Reliability in grid computing systems

[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] Reliability and Performance of Tree-Structured Grid Services

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

[9] Fundamentals of Fault-Tolerant Distributed Computing in Asynchronous Environments

[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] Distributed workflow mapping algorithm for maximized reliability under end-to-end delay constraint

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

[17] Reliability model of distributed simulation system

[18] Reliability Modeling and Analysis of Computer Networks

[19] Cloud Service Reliability: Modeling and Analysis

[20] Optimal task allocation for maximizing reliability in distributed real-time systems

[21] Performance Implications of Periodic Checkpointing on Large-scale Cluster Systems

[22] A large-scale study of failures in high-performance computing systems

[23] Performance Implications of Failures in Large-Scale Cluster Scheduling

[24] Discrete and continuous reliability models for systems with identically distributed correlated components

[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

Appendices

Appendix A
A

Chapter 8

Formatting

Avoid empty spaces between *chapter-section*, *section-sub-section*. For instance, a very brief summary of the chapter would be one way of bridging the chapter heading and the first section of that chapter.

8.1 Page Size and Margins

Use A4 paper, with the text margins given in Table 8.1.

Table 8.1: Text margins for A4.

margin	space
top	3.0cm
bottom	3.0cm
left (inside)	2.5cm
right (outside)	2.5cm
binding offset	1.0cm

8.2 Typeface and Font Sizes

The fonts to use for the reports are **TeX Gyre Termes** (a **Times New Roman** clone) for serif fonts, **TeX Gyre Heros** (a **Helvetica** clone) for sans-serif fonts, and finally **`TeX Gyre Cursor`** (a **`Courier`** clone) as mono-space font. All these fonts are included with the TeXLive 2013 installation. Table 8.2 lists the most important text elements and the associated fonts.

Table 8.2: Font types, faces and sizes to be used.

Element	**Face**	**Size**	**LaTeXsize**
Ch. label	**serif, bold**	24.88pt	`\huge`
Chapter	**serif, bold**	24.88pt	`\Huge`
Section	**sans-serif, bold**	20.74pt	`\LARGE`
Subsection	**sans-serif, bold**	17.28pt	`\Large`
Subsubsection	**sans-serif, bold**	14.4pt	`\large`
Body	serif	12pt	`\normalsize`
HEADER	SERIF, SMALLCAPS	10pt	
Footer (page numbers)	serif, regular	12pt	
Figure label	**serif, bold**	12pt	
Figure caption	serif, regular	12pt	
In figure	sans-serif	*any*	
Table label	**serif, bold**	12pt	
Table caption and text	serif, regular	12pt	
`Listings`	`mono-space`	$\leq$ 12pt	

8.2.1 Headers and Footers

Note that the page headers are aligned towards the outside of the page (right on the right-hand page, left on the left-hand page) and they contain the section title on the right and the chapter title on the left respectively, in SMALLCAPS. The footers contain only page numbers on the exterior of the page, aligned right or left depending on the page. The lines used to delimit the headers and footers from the rest of the page are $0.4pt$ thick, and are as long as the text.

8.2.2 Chapters, Sections, Paragraphs

Chapter, section, subsection, etc. names are all left aligned, and numbered as in this document.

Chapters always start on the right-hand page, with the label and title separated from the rest of the text by a $0.4pt$ thick line.

Paragraphs are justified (left and right), using single line spacing. Note that the first paragraph of a chapter, section, etc. is not indented, while the following are indented.

8.2.3 Tables

Table captions should be located above the table, justified, and spaced 2.0cm from left and right (important for very long captions). Tables should be numbered, but the numbering is up to you, and could be, for instance:

- **Table X.Y** where X is the chapter number and Y is the table number within that chapter. (This is the default in LaTeX. More on LaTeX can be found on-line, including whole books, such as [**?**].) or

- **Table Y** where Y is the table number within the whole report

As a recommendation, use regular paragraph text in the tables, bold headings and avoid vertical lines (see Table 8.2).

8.2.4 Figures

Figure labels, numbering, and captions should be formed similarly to tables. As a recommendation, use vector graphics in figures (Figure 8.1), rather than bitmaps (Figure 8.2). Text within figures usually looks better with sans-serif fonts.

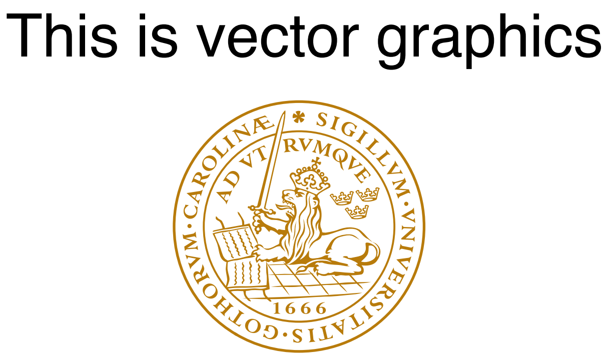

Figure 8.1: A PDF vector graphics figure. Notice the numbering and placement of the caption. The caption text is indented 2.0cm from both left and right text margin.

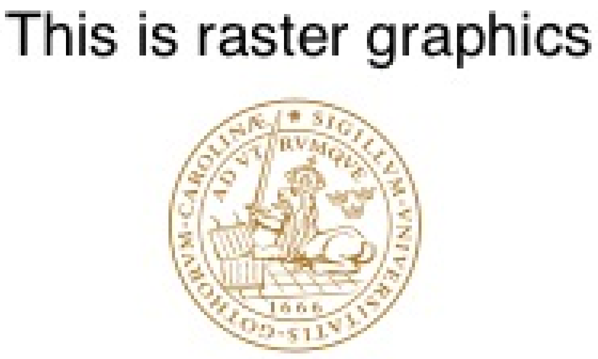

Figure 8.2: A JPEG bitmap figure. Notice the bad quality of such an image when scaling it. Sometimes bitmap images are unavoidable, such as for screen dumps.

For those interested in delving deeper into the design of graphical information display, please refer to books such as [**?**, **?**].

8.3 Mathematical Formulae and Equations

You are free to use in-text equations and formulae, usually in *italic serif* font. For instance: $S = \sum_i a_i$. We recommend using numbered equations when you do need to refer to the specific equations:

$$E = \int_0^\delta P(t)dt \quad \longleftrightarrow \quad E = mc^2 \tag{8.1}$$

The numbering system for equations should be similar to that used for tables and figures.

8.4 References

Your references should be gathered in a **References** section, located at the end of the document (before **Appendices**). We recommend using number style references, ordered as appearing in the document or alphabetically. Have a look at the references in this template in order to figure out the style, fonts and fields. Web references are acceptable (with restraint) as long as you specify the date you accessed the given link [**?**, **?**]. You may of course use URLs directly in the document, using mono-space font, i.e. `http://cs.lth.se/`.

8.5 Colours

As a general rule, all theses are printed in black-and-white, with the exception of selected parts in selected theses that need to display colour images essential to describing the thesis outcome (*computer graphics*, for instance).

A strong requirement is for using **black text on white background** in your document's main text. Otherwise we do encourage using colours in your figures, or other elements (i.e. the colour marking internal and external references) that would make the document more readable on screen. You may also emphasize table rows, columns, cells, or headers using white text on black background, or black text on light grey background.

Finally, note that the document should look good in black-and-white print. Colours are often rendered using monochrome textures in print, which makes them look different from on screen versions. This means that you should choose your colours wisely, and even opt for black-and-white textures when the distinction between colours is hard to make in print. The best way to check how your document looks, is to print out a copy yourself.

Chapter 9
Language

You are strongly encouraged to write your report in English, for two reasons. First, it will improve your use of English language. Second, it will increase visibility for you, the author, as well as for the Department of Computer Science, and for your host company (if any).

However, note that your examiner (and supervisors) are not there to provide you with extensive language feedback. We recommend that you check the language used in your report in several ways:

Reference books dedicated to language issues can be very useful. [**?**]

Spelling and grammar checkers which are usually available in the commonly used text editing environments.

Colleagues and friends willing to provide feedback your writing.

Studieverkstaden is a university level workshop, that can help you with language related problems (see Studieverkstaden's web page).

Websites useful for detecting language errors or strange expressions, such as

- `http://translate.google.com`
- `http://www.gingersoftware.com/grammarcheck/`

9.1 Style Elements

Next, we will just give some rough guidelines for good style in a report written in English. Your supervisor and examiner as well as the aforementioned **Studieverkstad** might have a different take on these, so we recommend you follow their advice whenever in doubt. If you want a reference to a short style guide, have a look at [**?**].

Widows and Orphans

Avoid *widows* and *orphans*, namely words or short lines at the beginning or end of a paragraph, which are left dangling at the top or bottom of a column, separated from the rest of the paragraph.

Footnotes

We strongly recommend you avoid footnotes. To quote from [**?**], *Footnotes are frequently misused by containing information which should either be placed in the text or excluded altogether. They should be avoided as a general rule and are acceptable only in exceptional cases when incorporation of their content in the text [is] not possible.*

Active vs. Passive Voice

Generally active voice (*I ate this apple.*) is easier to understand than passive voice (*This apple has been eaten (by me).*) In passive voice sentences the actor carrying out the action is often forgotten, which makes the reader wonder who actually performed the action. In a report is important to be clear about who carried out the work. Therefore we recommend to use active voice, and preferably the plural form *we* instead of *I* (even in single author reports).

Long and Short Sentences

A nice brief list of sentence problems and solutions is given in [**?**]. Using choppy sentences (too short) is a common problem of many students. The opposite, using too long sentences, occurs less often, in our experience.

Subject-Predicate Agreement

A common problem of native Swedish speakers is getting the subject-predicate (verb) agreement right in sentences. Note that a verb must agree in person and number with its subject. As a rough tip, if you have subject ending in *s* (plural), the predicate should not, and the other way around. Hence, *only one s*. Examples follow:

incorrect He have to take this road.

correct He has to take this road.

incorrect These words forms a sentence.

correct These words form a sentence.

In more complex sentences, getting the agreement right is trickier. A brief guide is given in the *20 Rules of Subject Verb Agreement* [**?**].

Chapter 10
Structure

It is a good idea to discuss the structure of the report with your supervisor rather early in your writing. Given next is a generic structure that is a starting point, but by no means the absolute standard. Your supervisor should provide a better structure for the specific field you are writing your thesis in. Note also that the naming of the chapters is not compulsory, but may be a helpful guideline.

Introduction should give the background of your work. Important parts to cover:

- Give the context of your work, have a short introduction to the area.
- Define the problem you are solving (or trying to solve).
- Specify your contributions. What does this particular work/report bring to the research are or to the body of knowledge? How is the work divided between the co-authors? (This part is essential to pinpoint individual work. For theses with two authors, it is compulsory to identify which author has contributed with which part, both with respect to the work and the report.)
- Describe related work (literature study). Besides listing other work in the area, mention how is it related or relevant to your work. The tradition in some research area is to place this part at the end of the report (check with your supervisor).

Approach should contain a description of your solution(s), with all the theoretical background needed. On occasion this is replaced by a subset or all of the following:

- **Method**: describe how you go about solving the problem you defined. Also how do you show/prove that your solution actually works, and how well does it work.
- **Theory**: should contain the theoretical background needed to understand your work, if necessary.

- **Implementation**: if your work involved building an artefact/implementation, give the details here. Note, that this should not, as a rule, be a chronological description of your efforts, but a view of the result. There is a place for insights and lamentation later on in the report, in the Discussion section.

Evaluation is the part where you present the finds. Depending on the area this part contains a subset or all of the following:

- **Experimental Setup** should describe the details of the method used to evaluate your solution(s)/approach. Sometimes this is already addressed in the **Method**, sometimes this part replaces **Method**.
- **Results** contains the data (as tables, graphs) obtained via experiments (benchmarking, polls, interviews).
- **Discussion** allows for a longer discussion and interpretation of the results from the evaluation, including extrapolations and/or expected impact. This might also be a good place to describe your positive and negative experiences related to the work you carried out.

Occasionally these sections are intermingled, if this allows for a better presentation of your work. However, try to distinguish between measurements or hard data (results) and extrapolations, interpretations, or speculations (discussion).

Conclusions should summarize your findings and possible improvements or recommendations.

Bibliography is a must in a scientific report. LaTeX and `bibtex` offer great support for handling references and automatically generating bibliographies.

Appendices should contain lengthy details of the experimental setup, mathematical proofs, code download information, and shorter code snippets. Avoid longer code listings. Source code should rather be made available for download on a website or on-line repository of your choosing.

Appendices

Appendix B
About This Document

The following environments and tools were used to create this document:

- operating system: Mac OS X 10.10.1
- tex distribution: MacTeX-2014, `http://www.tug.org/mactex/`
- tex editor: Texmaker 4.4.1 for Mac, `http://www.xm1math.net/texmaker/` for its XeLaTeX flow (recommended) or pdfLaTeX flow
- bibtex editor: BibDesk 1.6.3 for Mac, `http://bibdesk.sourceforge.net/`
- fonts `cslthse-msc.cls` document class):

  for XeLaTeX: TeX Gyre Termes, TeX Gyre Heros, `TeX Gyre Cursor` (installed from the TeXLive 2013)

  for pdfLaTeX: TeX Gyre font packages: tgtermes.sty, tgheros.sty, tgcursor.sty, gtxmath.sty (available through TeXLive 2013)

- picture editor: OmniGraffle Professional 5.4.2

A list of the essential LaTeXpackages needed to compile this document follows (all except `hyperref` are included in the document class):

- `fontspec`, to access local fonts, needs the XeLaTeX flow
- `geometry`, for page layout
- `titling`, for formatting the title page
- `fancyhdr`, for custom headers and footers
- `abstract`, for customizing the abstract

- `titlesec`, for custom chapters, sections, etc.
- `caption`, for custom tables and figure captions
- `hyperref`, for producing PDF with hyperlinks
- `appendix`, for appendices
- `printlen`, for printing text sizes
- `textcomp`, for text companion fonts (e.g. bullet)

Other useful packages:

- `listings`, for producing code listings with syntax colouring and line numbers

Appendix C
List of Changes

Since 2015/04/27

- Improved the **Structure** chapter and added more detailed comments for each part.

Since 2014/02/18

- Added the possibility to specify two supervisors. Use either of the `\supervisor{}` or `\supervisors{}{}` commands to set the names and contacts on the first page.

Since 2013/09/23

- Added missing colon ":" after *Examiner* on the front page.

Since 2013/08/30

- Changed fonts from Garamond (Times New Roman), Helvetica (Arial), Courier (Source Code Pro) to Tex Gyre fonts, namely Termes, Heros, Cursor, which are freely available with TexLive 2013 installation. These are all clones of Times New Roman, Helvetica and Courier, respectively. Garamond is problematic on some systems, being a non-freely available font.

- Corrected the *Face* column in Table 8.2 to correctly depict the font face.

Since 2013/02/22

- Number of words required in the abstract changed to 150 (from 300).

Since 2013/02/15

- Made a separate document class, for clarity.
- made it work with pdfLaTeX and garamond.sty, in addition to XeLaTeX and true type fonts. It is up to the user to get the hold of the garamond.zip from `http://gael-varoquaux.info/computers/garamond/index.html`.