NTNU

Norwegian University of
Science and Technology

Dependability modelling of Jgroup/ARM

Ane Sæstad

Master of Science in Communication Technology
Submission date: June 2008
Supervisor: Bjarne Emil Helvik, ITEM

Norwegian University of Science and Technology
Department of Telematics

Problem Description

Distributed computing has the potential of providing highly dependable services by having replicas of the server processes on several nodes in the network. This technology is relevant for network internal services as well as external services. Keeping consistency between replicas is demanding and introduces performance penalties. Fault transparency, one of the virtues of distributed computing, also requires management functionality, which is critical for the QoS, both during fault handling and normal operation.

Jgroup/ARM is a java based prototype system for providing such services. It is of interest to perform an evaluation of the dependability attributes of the services provided by services built on the Jgroup/ARM middleware. The assignment has the following elements:

* A study of techniques for dependability modelling of distributed systems,
* Identification of the elements that should be included in a system/service delivery model,
* A state-diagram or Petri net dependability model shall be established for services provided by the Jgroup/ARM system.
* Evaluation of some simple scenarios is to be carried out by available tools.

Assignment given: 15. January 2008
Supervisor: Bjarne Emil Helvik, ITEM

NTNU

Innovation and Creativity

Dependability modelling of Jgroup/ARM

Ane Sæstad

Master thesis
Spring 2008
Supervisor: Bjarne E. Helvik

Norwegian University of Science and Technology
Faculty of Information Technology, Mathematics and Electrical Engineering
Department of Telematics

Preface

This master thesis is written by Ane Sæstad in the 10th semester of a Master of Science degree in Communication Technology at the Norwegian University of Science and Technology. I would like to thank my supervisor Bjarne E. Helvik for helpful advise during the semester.

Ane Sæstad

Trondheim, June 4, 2008

Contents

List of Figures

List of Tables

Acronym list

ARM Autonomous Replication Management

AS Additional Service

CPN Colored Petri Nets

DCS Distributed Computing System

DPS Distributed Processing System

DR Dependable Registry

EGMI External Group Method Invocation

FTA Fault Tree Analysis

GCS Group Communication System

GMIS Group Method Invocation Service

GMS Group Membership Service

GSPN Generalized Stochastic Petri Net

IGMI Internal Group method Invocation

JVM Java Virtual Machine

MC Management Client

MS Monitored Subsystem

MTBF Mean Time Between Failures

PGMS Partition-aware Group Membership Service

PN Petri Net

RM Replication Manager

RMI Remote Method Invocation

SAN Stochastic Activity Network

SM Service Monitor

SMS State Merging Service

SPN Stochastic Petri Net

QOS Quality of Service

Abstract

In the later years, fault tolerant distributed systems have been applied to a variety of network internal and external services. Keeping distributed systems consistent and fault tolerant require management functionality. Jgroup/ARM is a java based prototype system which automates this management functionality for fault tolerant distributed systems.

This thesis presents an evaluation of the dependability characteristics of the Jgroup/ARM system. Both static and dynamic modelling techniques are introduced, but the main focus is on the dynamic techniques; state-diagrams and Petri net models. Previous work is evaluated to find an approach suitable for dependability modelling of Jgroup/ARM.

A system delivery model for Jgroup/ARM is developed based on its functionality. The monitored subsystem is defined to include the hardware (processors), a given number of services in the system and the replication management functionality (ARM framework).

A Petri net model of Jgroup/ARM is developed in the Möbius modelling tool based on the system delivery model. A model of a single service replicated on a cluster of processors is developed, analyzed and expanded to include multiple services and the ARM framework functionality.

The dependability of Jgroup/ARM is evaluated through simulating example scenarios in Möbius. The results show that the system availability is very high, even with a relatively high failure rate.

Chapter 1

Introduction

This chapter gives an introduction to the thesis. It gives the motivation, objectives and methods used. Section 1.5 gives the structure of the report.

1.1 Motivation

Distributed computing has the potential of providing highly dependable services by having replicas of the server processes on several nodes in the network. This technology is relevant for network internal services (e.g. network management) as well as network external services (e.g. e-commerce). Keeping consistency between replicas is a demanding task which introduces performance penalties. Fault transparency, one of the advantages of distributed systems, also requires management functionality. The management functionality is important for the QoS both during fault handling and normal operation.

Jgroup/ARM is a system that provides management services for distributed systems. An evaluation of the dependability parameters of this system is of interest.

1.2 Objectives

The objective of this thesis is to develop a dependability model for the Jgroup/ARM middleware framework. This will be done by

- performing a theoretical study of techniques for dependability modelling of distributed systems.
- developing a system/service delivery model for the Jgroup/ARM system.
- establishing a state-diagram or a Petri net dependability model for the Jgroup/ARM system.
- evaluating simple scenarios with the Möbius modelling tool.

1.3 Scope and limitations

The dependability of a distributed system is an increasingly important issue. Keeping dependable systems consistent and fault tolerant requires management information. However, as the management functionality is important for the QoS, the Jgroup/ARM system was developed to automate it. The focus in this thesis is on the dependability modelling of distributed systems in general, and Jgroup/ARM in particular.

Basic knowledge about distributed systems and dependability are assumed, hence only a short introduction to these concepts are given in the report. Several modelling techniques are introduced in this thesis, but due to the scope only a Petri net diagram is completed. The modelling assumptions and simplifications made are described in Chapter 8.

1.4 Methodology

The project started with a literature study of both the Jgroup/ARM system and the different techniques for dependability modelling of distributed systems. A literature search was performed in the standard electronic databases[1] to find background information.

Examples of existing dependability models were studied before a dependability model of Jgroup/ARM was developed. The initial model was a state-diagram of a simplified system. As the state-diagrams are limited by state explosion, the focus was changed to stochastic Petri net models where the

[1] IEEE Xplore Journals, ACM Digital Library and SpringerLink(MetaPress)

complete system was considered. A study of the Petri net modelling technique and the use of the Möbius modelling tool followed.

The knowledge of modelling techniques, Jgroup/ARM and the Möbius modelling tool was applied to develop a Petri net diagram dependability model of the Jgroup/ARM system. Difficulties encountered when modelling in Möbius was solved by consulting the Möbius manual [21] and searching in on-line forums.

1.5 Structure of report

Chapter 1 gives the scope, motivation and methods for the project.
Chapter 2 defines terms used throughout the thesis.
Chapter 3 describes techniques for dependability modelling of distributed systems and introduces some existing designs.
Chapter 4 describes the Jgroup/ARM system.
Chapter 5 outlines the functionality of the Möbius modelling tool.
Chapter 6 contains a description of the methodology used while modelling Jgruop/ARM.
Chapter 7 gives the system delivery model based on knowledge from Chapters 3 and 4.
Chapter 8 presents the resulting dependability model based on the system delivery model from Chapter 7.
Chapter 9 includes validation and verification of the dependability model from Chapter 8.
Chapter 10 contains some example scenarios and their simulation results.
Chapter 11 describes the lessons learned and problems encountered during the modelling process.
Chapter 12 concludes the thesis and proposes topics for further work.
Appendix A contains the source code for the SAN models developed in Chapter 8.
Appendix B gives the results used for validation and verification in Chapter 9.
Appendix C contains the complete simulation results of the example scenarios simulated in Chapter 10.

Chapter 2

Definitions

The aim of this chapter is to introduce concepts that will be used throughout the thesis. Previous knowledge about dependability and distributed systems are assumed, for further details consult the textbooks [6, 3].

2.1 Distributed systems

A distributed system is a system where the processing servers are replicated on several nodes in the network. The network nodes available to the distributed system is termed the target environment. As stated in [24], one of the advantages of a distributed processing system is its ability to continue operation even when some of its components fail. This is called fault tolerance. The client using a distributed service may not be aware of failures in a system component, as long as a given number of components are still functioning.

Fault transparency is an important property of distributed systems. As the system functionality is replicated, replica failures can be hidden from the user of the service as long as a given number of replicas are working. The concept of fault transparency requires management functionality because the count of working and failed replicas needs to be kept, and decisions on when and where to create replicas must be made.

Keeping the different replicas consistent is also a consuming task. Numerous algorithms exist for this task; however, a description of these is beyond the scope of this thesis. Basically, keeping the consistency of system replicas requires an increase in the number of messages passed in the system. This

increase in complexity and capacity requirements is the main drawback of distributed system processing. Hence, features as fault tolerance and replica consistency increase the system complexity and introduce the need for management functionality.

Jgroup/ARM[1] is a middleware framework that will perform these management tasks automatically.

2.2 Dependability

Numerous definitions of dependability exist, but in [6] dependability is defined as *the trustworthiness of a system, such that reliance can justifiably be placed on the service it delivers.* The service delivered by the system is defined as the system behavior as perceived by its users. In the context of distributed systems, dependability is a collective term used to describe availability, reliability and performance predictability [7].

The dependability of a system describes how the system behaves by its operational characteristics. The dependability is defined by internal characteristics which can be measured. However, a system with given characteristics is not necessarily dependable. The dependability definition can be extended to also consider the threats to the system, means for improvement and attributes that describe the system dependability. A dependability tree was developed for illustration in [11] and is reproduced in Figure 2.1.

As shown in Figure 2.1 different *threats* are faults, errors and failures. These concepts are introduced in [6]. *Means* describes the efforts; fault avoidance and fault tolerance, that can be made to make a system more dependable. And finally *attributes* describes the dependability properties of the system.

The work in this thesis will be focused on modelling the dependability attributes of Jgroup/ARM.

2.2.1 Dependability parameters

The most common dependability parameters are introduced below, further details can be found in [6].

[1] introduced in Chapter 4

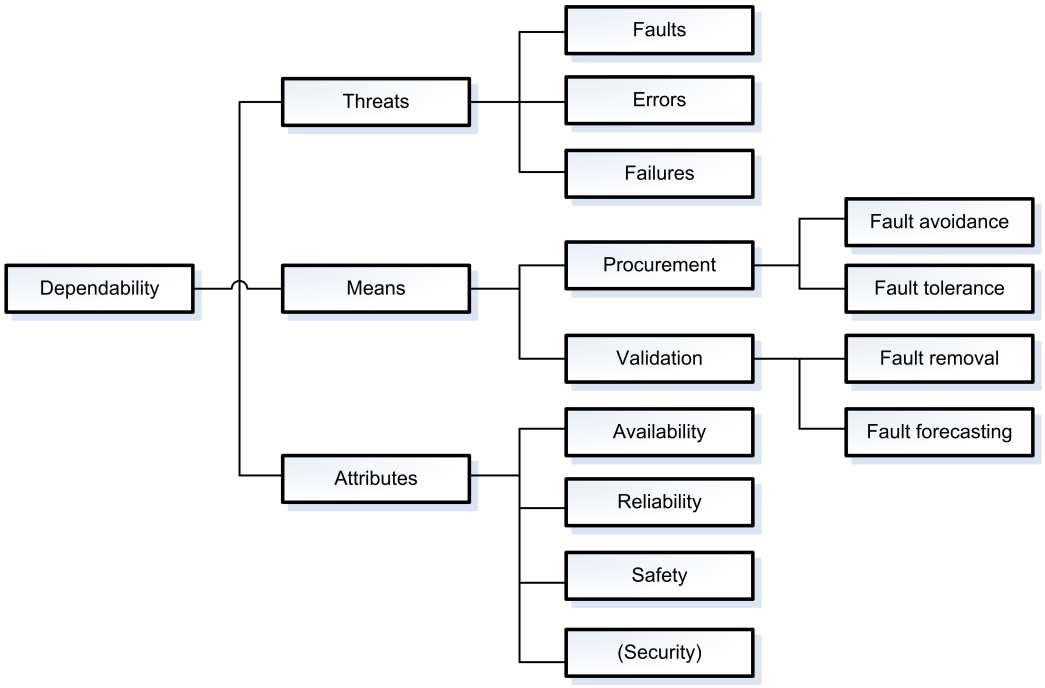


Figure 2.1: The different aspects of dependability linked in a dependability tree [11]

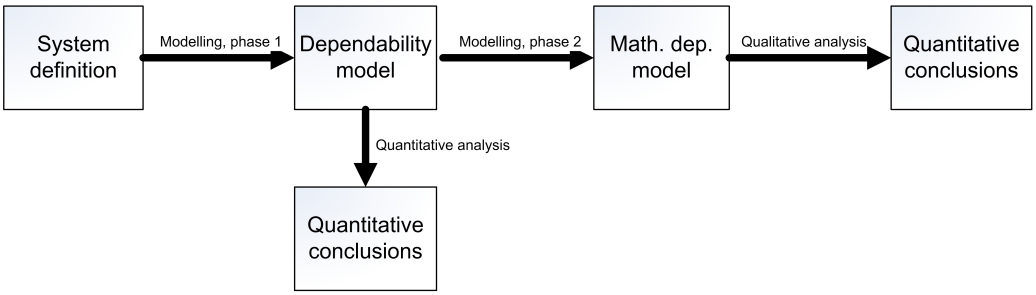


Figure 2.2: Modelling and analysis activities and results, from [6]

Dependability is defined as the availability and reliability offered by middleware platforms.

Reliability is the likelihood that a middleware platform provides services in a specified way, even when applications or middleware fail.

Availability gives the fraction of time the middleware is operational.

Performance predictability is strongly related to availability. Ensures that a client of a middleware platform can rely on a request being performed within a given time.

Safety is defined as a system's ability to provide service without a catastrophic failure occuring.

Performability quantifies the ability of a system to perform in presence of faults [2].

2.3 Dependability modelling

Dependability modelling is an alternative to testing. It can be of great help when evaluating important aspects of the behavior of a system, while the unnecessary details are abstracted. By modelling, effects of design changes can be evaluated without making any changes to the system [2].

Figure 2.2 presents a framework for dependability modelling. It can be seen that the dependability modelling process consists of several tasks, each of which produce different results. The tasks are modelling (phase 1), qualitative analysis, modelling (phase 2) and quantitative analysis, all described below [6].

Modelling, phase 1 transforms the system definition to a dependability model by focusing on failures, fault handling and repairs. The phase focuses on systemizing and concentration of system definition.

Qualitative analysis is a systematic deduction of the systems dependability characteristics, from the dependability model developed in the modelling phase.

Modelling, phase 2 describes the events in the system and their interrelation in terms of mathematical statistics. The process can be partly dictated by the model from phase 1.

Quantitative analysis consists of a derivation of the dependability measures of the system, by measures from the operations and maintenance. Also contains the dependability model and the dependability characteristics of its elements. The quantitative analysis can be done either by mathematical analysis or by simulation.

The main focus in this thesis is on the modelling, both phase 1 and phase 2. Each of the activities yield results in form of documents describing the work in its current state. The intermediate and end results are described below [6].

System definition is the description of the system at its current state in the development.

Dependability model is a description aimed at revealing the dependability characteristics of a system. The dependability model is the focus of this thesis. Different models exist and will be described in Chapter 3.

Mathematical reliability model is a quantitative description of the dependability characteristics and their interrelation.

Qualitative conclusions contains a catalog of e.g. critical components, dependability bottlenecks and minimal cut-sets.

Quantitative conclusions provides the dependability performance of the system in terms of the required measures (availability, reliability function, MTBF). From this document one can conclude whether the dependability requirements of a system are met or not.

As illustrated by Figure 2.2 the modelling of a system consists of several activities, all performed to get an idea of how the system will perform. A modell is a simplification of the real system. Hence, care must be taken not to simplify the system too much. This thesis will focus on the dependabil-

ity modelling with the quantitative analysis of the dependability model as subsequent task.

Chapter 3

Dependability modelling techniques

This chapter presents different techniques for dependability modelling of distributed systems and highlights their strengths and weaknesses.

In general, things to consider when modelling a system are [6]:

- the size and complexity of the system to be modelled.
- primary dependability measures to be determined.
- the type of the subsequent analysis.
- available modelling support and analysis tools.

Two main classes of dependability models are introduces and described in the following sections. Section 3.1 introduces the dynamic modelling techniques, while the static techniques are introduced in Section 3.2. Section 3.4 concludes the Chapter.

3.1 Dynamic models

Common to the dynamic modelling techniques are their strength in describing the dynamic behavior of systems. That is, to describe systems where dependencies exist between events and event system elements, and the time between events and the sequence of events are important.

The two models described in this section, state machines and Petri nets are essentially identical and a Petri net model can easily be converted to a state diagram. Converting a state diagram into a Petri net is also possible, but requires more work.

3.1.1 State-diagrams

This section will introduce the state machine as a tool for dependability modelling of distributed systems. The state machine is the oldest known formal model for sequential behavior, that is, behavior that cannot be defined by the inputs only, but depends on the history of the inputs [26]. In fact, any logic that determines the behavior of a system can be called a state machine.

A state machine can be represented by a state machine diagram or a state machine table. However, the state machine diagrams (henceforth termed state-diagrams) are easier to read and are the representation used in this thesis.

A state-diagram is built up by states (circles), transitions and actions. The state stores information about the input changes from system start to present. A transition indicates a state change and is described by conditions that would have to be fulfilled in order for the transition to be enabled. The action is a description of the activity performed at the given moment. This thesis will look at the event driven state-diagrams. In an event driven state-diagram the system will "'wait"' for a system event before it changes state.

Figure 3.1 illustrate the state-diagram of a system consisting of 2 servers, in which both the servers can fail and be repaired. The states are represented by circles and squares, where the circles represent up-states and the square represents the down-state. The actions are "'server fail"' and "'repair server"'. The current state and condition decide whether or not a transition can change the system from one state to another.

A state-diagram can be either a Mealy or a More model, or a mix of both. The Mealy type produces output as a function of the system state and the input. The More type produces output as a function of input only. Henceforth, the state-diagrams are assumed to be of the Mealy type.

In a state-diagram one can easily identify the operational states. The possible next states, and by which rate they can be reached, can be identified for each of the system states. However, state-diagrams may become cumbersome even for a relatively small system.

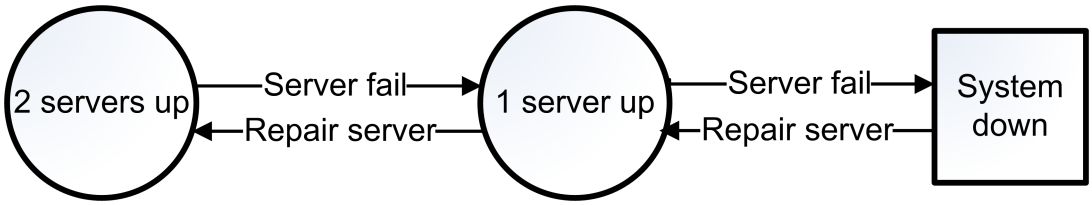


Figure 3.1: State-diagram showing a system with 2 servers. Servers can fail and be repaired.

The rapid growth of the state space is called state explosion. The size of the state space of a system tends to grow exponentially in the number of its processes and variables, where the base of the exponentiation depends on the number of local states a process has, the number of values a variable may store and on some kind of tightness of the connection between system components [25]. The state explosion limits the size of the system that can be modeled.

As an example, consider the simple system of n processors, each with k internal states. The state space of that simple system consists of n^k states! For a system of 5 processors each with 4 internal states the total state space generated is $5^4 = 625$ states. Increasing the number of processors to 10 will increase the state space to $10^4 = 10000$ states. Hence, state explosion occur even for relatively simple systems.

Because state-diagrams are such a simple and efficient tool, efforts are made to limit the state explosion. Several solutions are suggested, but are left out of the discussion due to the scope of this thesis. Some examples can be found in [25].

Markov models

To be able to analyze the dependability models numerically, the following assumptions about the system are made [3].

- The system has the Markov property; *next state depends only on current state, not the history of states visited*[1].
- Failures and repairs occur according to a Poisson process[2].

[1] A thorough introduction to Markov models can be found in [3, 19].

[2] Basic knowledge on Poison processes assumed known, introduction can be found in [3] amongst others

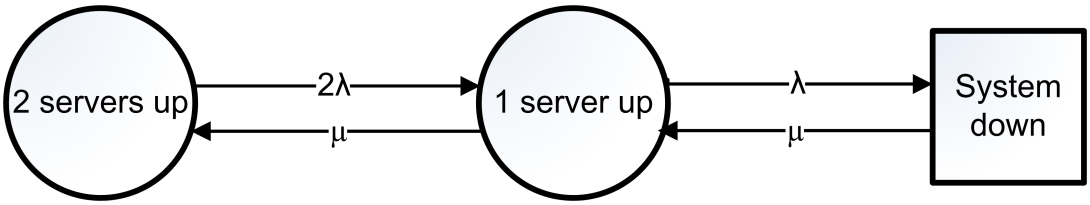


Figure 3.2: Markov model of a system with 2 servers. Servers can fail and be repaired.

- System components fail independent of each other.

Figure 3.2 illustrates the same system as Figure 3.1, the difference being that Figure 3.2 models the system as a Markov model. It is assumed that the two processors have independent failure and repair actions. All system times are assumed to occur according to a Poisson process.

λ is the failure rate
μ is the repair rate

A mathematical analysis of this model gives the availability, A, of the system. Where A = P_u = 1 - P_d, and P_u and P_d are the probability of being in a working and a failed state respectively.

The Markov model is one of the most important analytical methods for dependability modelling. However, as mentioned above, the system to be modeled must be described at the state level. The number of states required may be very large and the model quickly becomes incomprehensible [22]. State lumping techniques exist, but are left out of the discussion due to the scope of this thesis.

Previous work

Using state-diagrams to model distributed systems is a popular approach. It is fairly comprehensible and flexible; consequently it is used in many forms.

Chen and He use Markov models to model a distributed system under the occurrence of faults, reconfiguration and repair in [27]. The models are developed generically such that they can be used to evaluate the risk, reliability and availability of individual tasks in any homogeneous distributed system. Eventually the study is extended to analyze the dependability of any heterogeneous system consisting of numerous homogeneous distributed systems.

In [10] Lai et al. the availability of a typical distributed software/hardware system is modelled by a Markov model. In the system, identical copies of the distributed application software run on identical processors. The article considers both a two-host system and a general multi-host system. The steady-state availability is found by developing the Markov model and deriving its balance equations.

The Markov models are also used for modelling in [5]. However, here Hariri and Mutlu use several Markov models which is combined hierarchically in two levels to represent the complete system. At the higher level (user level) the availability of the tasks is analyzed using a graph-based approach. At the lower level (component level) detailed Markov models are developed to analyze the component availabilities. A systematic approach was developed to apply the two-level hierarchical model to the availability evaluation of processes and services provided by a distributed system.

The EURESCOM whitepaper [7] studies a dependability model of middleware platforms. Two aspects of dependability are discussed: the dependability of the generic functions provided by the middleware platform and how the middleware platform supports the dependability of the distributed applications residing on it. A discrete state continuous time dependability model is developed for a system replicated with active and passive replicas.

3.1.2 Petri nets

This section will first describe the general functionality of Petri nets and later apply this general knowledge to the modelling of distributed systems.

Petri nets are a graphical and mathematical modelling tool which can be used to model any area or system that can be described graphically, including distributed systems [17]. They give an abstract, but formal, method to model the information flow in a program/system.

The Petri net graph models the static properties of a system, whilst the execution of a Petri net model shows the dynamic system properties [9].

In a Petri net the conditions are represented by places (circles) and the events by transitions (bars). As an example, a place can be the number of working replicas in a system and transitions can be failure, view change or creation events. A place can contain tokens that are moved to another place when a transition fire. The places and transitions are connected by arcs. A place, p, is called an input place for a transition, t, if there is an arc from p to t. If

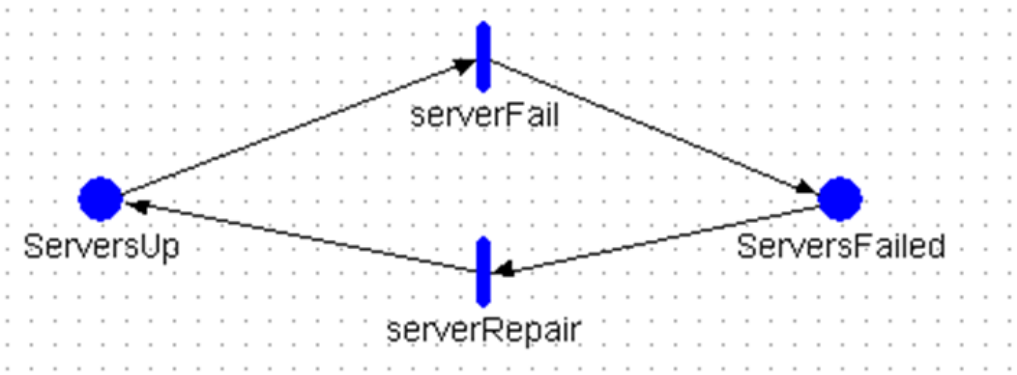


Figure 3.3: Petri net for a system with two servers, both of which can fail and be repaired independently.

the arc is from t to p, the place p is called the output place for the transition t. t can be enabled by each of its input places containing a token. Once t is enabled it can fire by moving a token from each input place and adding it to the output place. An arc can require more than one token to fire, how many are determined by the multiplicity of the arc. The number of tokens in a place is called the marking of the place. A new marking is generated each time a transition fires.

The Petri nets allows two different kinds of transitions; timed and immediate transitions. The timed transitions have firing times with given distributions and the immediate transitions fires as soon as they are enabled. By adding features like enabling functions to a Stochastic Petri net (SPN) flexibility can be added to its modelling power.

Stochastic Activity Networks (SANs) are a further generalization of the SPNs. In SANs the use of input and output gates are enabled, furthermore, an extended place which allows for places to contain user defined data structures can be used [21]. The input and output gates provide modelling flexibility. If no input gate is specified, the default input gate is represented by a line from a place to a transition, and holds if there is at least one token in the input place. A user identified input gate supersedes the default gate [20]. As before, a transition is enabled, and takes a token from the input places, when all input places hold. Correspondingly holds for the output gates. The default output gate is represented by a line from the transition to the output place and consists of a function that adds a token to its output place. Users may define output gates which supersede the default output gate as well.

Figure 3.3 shows the model from Figure 3.2 converted to a Petri net. The

two places *ServersUp* and *ServersFailed* contains a marking depending on the state of the servers. Initially, the marking of *ServersUp* is set to 2 tokens and the marking of *ServersFailed* to 0 tokens. Hence, only the transition *serverFail* can fire initially, as *ServersFailed* contains no tokens. As in Figure 3.2 all actions are assumed to be exponentially distributed. Transition *serverFail* fires with the rate λ, and *serverRepair* with the rate μ.

When *serverFail* fires, a token is moved from the input place (*ServersUp*) to the output place (*ServersFailed*), creating a new marking. Now both transitions are enabled and can fire. As with the system in 3.2, the system is defined as not working when both servers have failed, hence when place *ServersFailed* contains 2 tokens.

Petri nets are ideal to model distributed systems where multiple processes occur concurrently. The probability of all the required programs being operational defines the distributed system reliability [12]. The nature of Petri nets is asynchronous; it has no measure of time. The only important factor is the partial ordering of events. Hence, Petri nets are the best for systems with asynchronous and independent events.

The modelling generality of the Petri nets must be weighted against their analysis capability. A general model often grows too complex for analysis. Furthermore, the information in Petri nets can only be communicated to persons familiar with the technique, and for complex systems the Petri nets may be incomprehensible to all but its author.

Previous work

A Stochastic Petri Net (SPN) dependability model of a distributed system is suggested by Lopez-Benitez in [12]. SPNs are used to estimate both the reliability and availability of programs in distributed systems. The distributed system is considered available if a set of files/programs are available.

A basic reliability model including node failures and repairs is modeled by Petri net diagrams. Two possible models are introduced; the global repair model and the local repair model. The former assumes a centralized repair team and is used when analyzing the availability of programs subject to hardware faults that are repaired globally. The latter assumes that repairs are localized to the node where they occur, and can be used to evaluate program reliability where hardware can fail and be repaired locally. Thus, the normal operation of the system need not be interrupted.

The models suggested can also be used to study performability related issues and can easily be expanded to include faults related to communication and software as well.

The dependability of a LEO satellite network is modelled in [1], using stochastic activity networks (SANs) in Möbius. A model of an individual satellite is designed and used to explore the external elements' effect on the model dependability. The satellite model is used to develop a satellite network model. The network model is used to analyze the satellite network's dependability and performance.

3.2 Static models

This section introduces the two most commons techniques for static dependability modelling of distributed systems. Other techniques exist as well but are omitted due to the scope of this thesis.

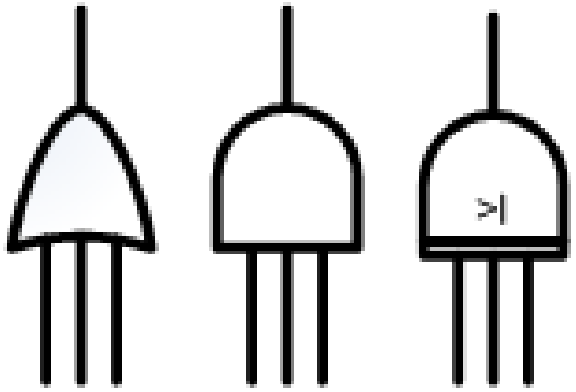
The main strength of the static models is their simple graphic representation. They are best suited for modelling and analyzing large systems. However, from the name it can be deduced that static models mainly are able to model the static dependability properties of a system and can not easily model a sequence of events.

3.2.1 Fault trees

The fault tree is a graphical model used to deduct the structure function of a system [3]. The function can be used to derive quantitative and qualitative dependability measures. The focus of the fault trees are faults and errors. Fault trees can be used to model system internal events and conditions and events in the environment.

The combinatorial gates used in fault trees are `or`, `and` and `more than` gates.

- The symbol representation of an `or` gate is illustrated in Figure 3.4(a). The output of an `or` gate is high (enabled) if one or more of the inputs are high (enabled).
- The symbol representation of an `and` gate is illustrated in Figure 3.4(b). The output of an `and` gate is high (enabled) if all of the inputs are high (enabled).



(a) The `or` gate (b) The `and` gate (c) The `more than` gate

Figure 3.4: The boolean operators in fault trees.

- The symbol representation of a `more than` gate is illustrated in Figure 3.4(c). The output of a `more than` gate is high (enabled) if more than a given number (l) of inputs are high (enabled).

The leaf nodes of the fault tree represent the lowest level faults. The definition of the lowest level has to be considered in the modelling context. In a distributed system the lowest level can be defined as a faulty PC, a faulty disk or a faulty CPU. These faults are called primary faults and are symbolized by a circle. Sometimes the primary event can not be determined due to lack of information; these leaf nodes are called undeveloped errors and are symbolized by a square.

Figure 3.5 illustrates the example from Section 3.1. The two servers are combined by an `and` gate, as both servers have to fail for the system to fail. An `or` gate combines the servers with the undeveloped error "'other faults"'. For simplicity the failure of a server is the primary fault in this example.

The stepwise approach described below can be used to analyze the fault tree (Fault Tree Analysis (FTA)) [23].

1. **Define the top event to study.**
   The top event is the failure the system should be analyzed with respect to. The top event may be obvious or hard to define. However, the difficulties mostly occur with lack of system knowledge. One top event are defined for each FTA. As mentioned above, no two top events exist in the same FTA.

2. **Obtain an understanding of the system.**

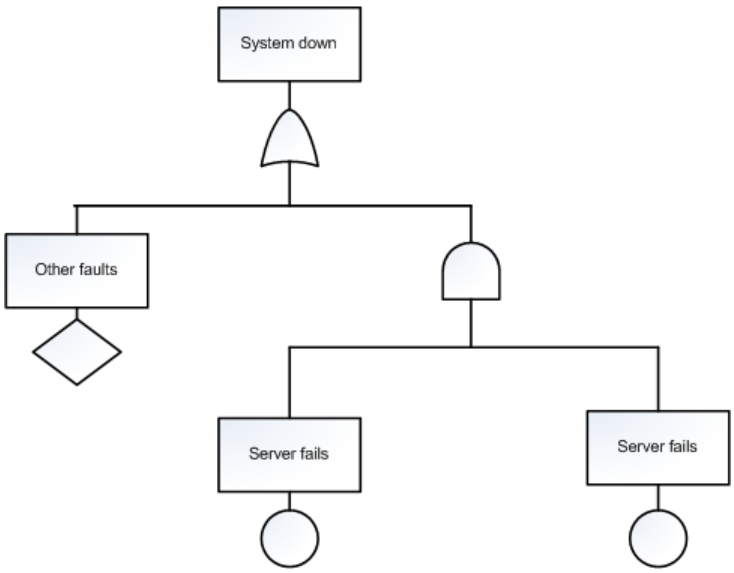


Figure 3.5: Fault tree for a system with two processors.

When the top event is defined, system knowledge is used to study all faults that may affect it. All the events are numbered and sequenced in the number of appearance.

3. **Construct the fault tree.**
   The event sequence built in step 2 is used to construct the fault tree. The top event is broken in lower level events that combined will lead to the top event. As illustrated in Figure 3.5 the tree is built up by events, `or` and `and` gates.

4. **Evaluate the fault tree.**
   The assembled fault tree is analyzed and evaluated for any possible improvements.

5. **Control the hazards identified**
   After the hazards are identified all possible methods are used to try to minimize the occurrence of the top event.

As with other static modelling techniques, the fault trees can model the static property of a system. However, it does not consider the sequence of events causing the error. The greatest advantage of the fault tree is its ability to model large systems. However, the fault tree is not capable of modelling complex behavior as dependencies amongst components and repairable systems which do not have a separate repair crew for each component [8]. The fault tree can be converted to a reliability block scheme.

3.2.2 Reliability block schemes

The reliability block schemes model the dependability structure of a system by considering the operational state of system elements. Models based on block schemes make it easier to deal with complex systems than what is the case of the dynamic models described in Section 3.1. However, the assumptions below might make the analysis less accurate [3, 6].

1. If the system is made up by more than one subsystem, each subsystem fails independent of all other subsystems and independent of the state of the system.

2. If a subsystem has failed, its services are restored independently of the state of all other subsystems.

3. The system behaves as intended. The fault tolerance handles failures until resources are exhausted, service restoration actions are always successful, failures do not propagate from one subsystem to others.

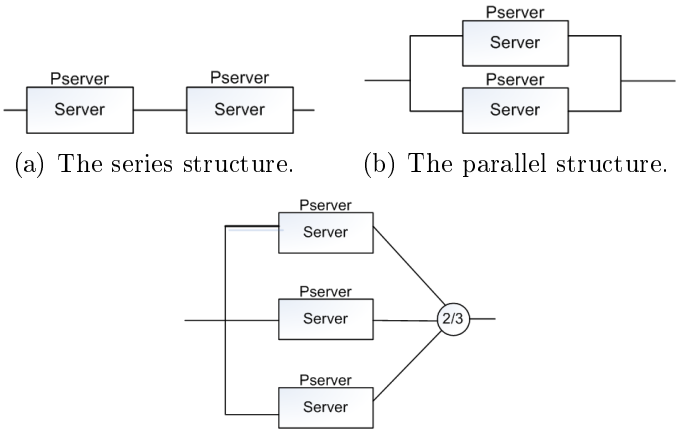
The reliability block approach is based on the assumption that a system structure can consist of a number of subsystems that may interact to provide a set of services. If the system is fault tolerant, the system might be able to deliver a service even if a subsystem fails.

The reliability block diagram can be used to analyze both the reliability and the availability of the system. The definition of working is different in the two cases. In the former, working should be interpreted as "'providing a service at a given instance of time"' and in the latter as "'the system has provided uninterrupted service in the period $(0,t)$"' [6].

The basic building block of the reliability block diagram is the reliability block. The reliability block is depicted as a square and represents a subsystem. The probability of the block working is denoted P_{block}.

Different structures exist for reliability block schemes, the simplest being a series structure, a parallel structure and a k-of-n structure. Figure 3.6 illustrates a system consisting of two (or in the case of 3.6(c) three) servers. In Figure 3.6(a) both servers need to be up for the system to be up. Hence, the probability of the series system being up is given by

$$P_{series} = \prod_{i=1}^{n} P_{server} = P_{server} \cdot P_{server} \tag{3.1}$$



(a) The series structure. (b) The parallel structure.

(c) The k-of-n structure

Figure 3.6: The reliability block structures

Figure 3.6(b) shows the same system, but here only one of the servers needs to be up to the system to be up. The probability of the parallel system being up is given by

$$P_{parallel} = 1 - \prod_{i=1}^{n}(1 - P_{server}) = 1 - (1 - P_{server}) \cdot (1 - P_{server}) \qquad (3.2)$$

The k-of-n structure is illustrated in Figure 3.6(c). The illustration represents the case where 2 of the 3 servers is required to work for the system to be up. The probability of a k-of-n system being up is given by (all subsystems assumed to be identical)

$$P_{k-of-n} = \sum_{j=2}^{3}(3\ nCr\ j)P_{server}^{j}(1-P_{server})^{3-j} = 3 \cdot P_{server}^{2} \cdot (1-P_{server}) \quad (3.3)$$

As can be seen from the illustrations, reliability block diagrams can be used to model the dependability of system structures. However, it can not consider dynamic behavior like a sequence of events.

3.3 How can dependability characteristics be obtained?

This section will give an introduction to how dependability parameters can be obtained from a dependability model.

3.3.1 Prototyping

The prototyping technique involves building a complete model of the system in question. Hence, the prototyping technique is very costly. Thus, prototyping should not be used early in a system development phase but be developed in the late stages.

3.3.2 Simulation

Using a simulator to determine dependability characteristics usually pose few problems and gives freedom in choosing the level of detail to be simulated. However, simulation has some drawbacks as well. The estimates can be inaccurate and simulation times may be excessively long to obtain a sufficient accuracy [6]. The latter problem increases with the introduction of fault tolerance to the system, because a large number of events must take place for every system failure observed. For some simulators this can be overcome by using stratified sampling, as described below.

Fault injection

Helvik et al. suggested an approach based on stratified sampling[3] with fault injection to obtain dependability characteristics of a service deployed in the Jgroup/ARM system in [16]. The strata is defined in terms of the number of near-coincident failures injected; single failures, double near-coincident failure and triple near-coincident failure. A near-coincident failure is a failure that occurs before the previous is handled. Experiments are assigned to strata after they are carried out.

[3]In stratified sampling each mutually exclusive subpopulation (stratum) is sampled separately.

The article performs an experimental evaluation which focuses on a service offered by a triplicated server. Predictions on the dependability characteristics of the service are obtained, consequently the work concludes that a service in a Jgroup/ARM system can obtain a very high availability and MTBF.

3.3.3 Mathematical analysis

To analyze dynamic dependability models mathematically, the theory of stochastic processes is used [6]. By mathematical analysis a closed form solution may be obtained. The closed form solution shows how the different system parameters, like failure rates and repair time, contribute to the system dependability. However, in a real size system the closed form solution might be difficult, or even impossible, to derive and requires high mathematical skills from the analyzer. Another drawback of mathematical analysis is the simplifications required to adapt the system to mathematically solvable models.

3.4 Concluding remarks

As discussed in this chapter, no modelling techniques exist that can model all aspects of the system dependability. However, the system tasks can be split into subtasks which can be modeled independently by the best suited technique. In [6] two different approaches are given.

1. Two or more models are made, each focusing on separate aspects of the system.
2. A hierarchical model is made. The model handles some complexity on each level to prevent each level from being too complex.

Chapter 4

The Jgroup/ARM system

This chapter gives a short introduction to the components and services of Jgroup/ARM. Further details can be found in [15]. This introduction to Jgroup/ARM is based on [16, 15, 13, 14].

4.1 Overview Jgroup/ARM

Jgroup/ARM is a middleware framework based on object groups, designed to simplify the development of dependable partition-aware applications.

Jgroup extends the Java Remote Method Invocations (RMI) through the group communication paradigm and has been designed specifically for application support in partitionable systems.

The Autonomous Replication Management (ARM) focuses on the deployment and operational aspects, where the gain in terms of improved dependability is assumed to be the greatest. The main objective of ARM is to localize failures and to reconfigure the system according to application-specific dependability requirements. Combining Jgroup/ARM can reduce the effort necessary for developing, deploying and managing dependable, partition-aware applications.

The Jgroup/ARM framework is automated to help with the management functions required when using distributed systems. It can handle both partitionings and failures of nodes and processes. Jgroup/ARM has functionality to merge the state of the partitions to a consistent state after short or long term partitioning. It also keeps track of the system state and creates and

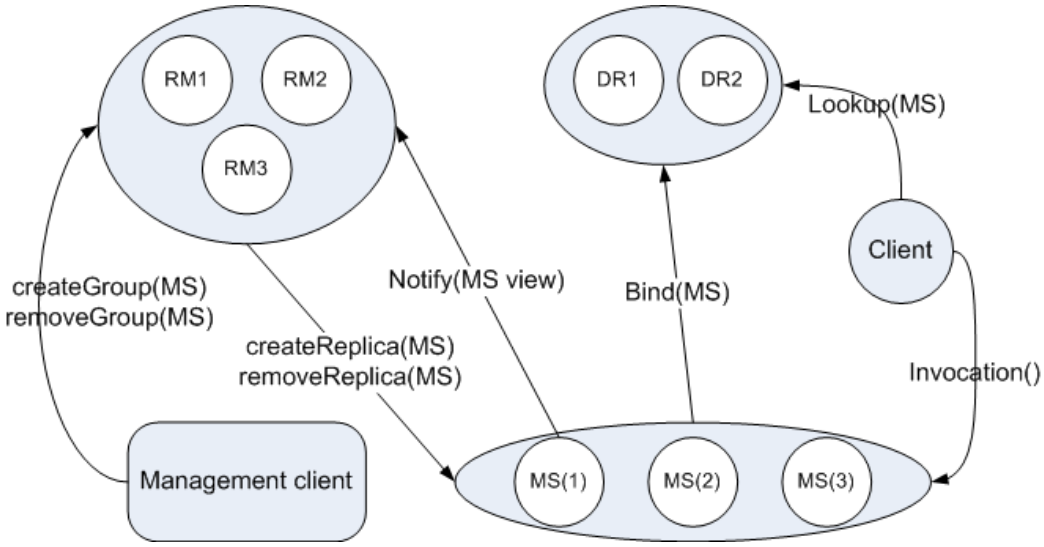


Figure 4.1: Jgroup/ARM architecture, modified from [15].

removes replicas when required. Figure 4.1 shows the architecture of the Jgroup/ARM framework.

4.2 Jgroup middleware

Jgroup aims to support dependable application development through the object group paradigm. In the object group paradigm a set of server objects (replicas) form a group [15]. Client objects interact with the server objects using External Group Method Invocation (EGMI). An EGMI corresponds to the standard RMI. The interface of the server group is obtained from a group proxy, and the group will to the client appear as a single server. However, replicating the server will render a more dependable system where servers can fail without the client being aware of the failure. The servers cooperate and keep the consistency by communicating by Internal Group Method Invocations (IGMIs).

Jgroup trades availability for consistency in that it allows all partitions to operate, much unlike the primary partition approach where only one partition is allowed to operate after a partitioning. Each object in a partition has a consistent view of the other objects in its partition. When partitions are reconciled, Jgroup has a State Merging Service (SMS) that merges the state of the different partitions.

The core facilities of Jgroup are described below [15, 14].

4.2.1 Partition-Aware Group Membership Service

A group is a collection of servers that cooperate in providing a distributed service [14]. The group size varies dynamically because servers can join and leave the group. The membership of a group is the servers that have joined the group and not yet left it. It is possible for different members in the group to have different perceptions of the group membership due to asynchrony and failures in the system.

The Partition-Aware Group Membership Service (PGMS) is responsible for tracking the membership and handle voluntary (join- and leave-messages) and involuntary (failures and repairs) variations. PGMS report the membership information to the members through installation of views. An installed view consists of a membership list and an identifier. A view represents the group membership as perceived by members included in the view.

PGMS must consider three issues;

1. The changes in group membership must be tracked accurately and timely for the installed views to represent current information.
2. A view should be installed only after the servers included have reached an agreement.
3. PGMS must guarantee that two views installed on different servers must be installed in the same order.

4.2.2 Group Method Invocation

Jgroup uses Group Method Invocation (GMI) for all types of communication, both within a group and between different groups. This extends the benefits of object orientation to internal communication as well. However, Jgroup distinguish between internal and external communication. Clients perform External GMI (EGMI) and servers perform Internal GMI (IGMI). The reasons for this separation are the following

Visibility
: Client should have access only to the service's public interface. The service's method invocations should be kept hidden.

Transparency
Clients should not be required to know whether they are sending invocations towards a single server or a group of servers.

Efficiency
EGMI has weaker semantics than IGMI which results in a more scalable system. If EGMI and IGMI were identical, a client would have to join a group in order to perform invocations towards it. This would result in a poor scalability, hence the separation.

All the internal methods of a service are collected to form the internal remote interface of the server object and the external remote interface consists of the external methods. An object performs a method invocation when it invokes a method on a group. A server completes an invocation when it terminates the execution of the associated method [14].

4.2.3 State Merging Service

Jgroup offers a service called State Merging Service (SMS). The SMS is the Jgroup version of an application-specific state reconciliation service. By state reconciliation is meant the process of constructing the state of merging partitions, where the new state reflects all non-conflicting concurrent updates to state.

SMS drives the state reconciliation by requesting "'getting"' and "'merging"' information about state from servers. The information collected is diffused to the remaining servers by a coordinator elected by the SMS.

The SMS elects a coordinator when multiple partitions merge. The coordinator is responsible for diffusing the partition state to servers outside its own partition. Servers receive the information and apply it to their own local copy of state. For a server to be part of a partition-aware system it must be able to act as a coordinator, which means that it must be able to maintain the entire state and diffuse this information to the other servers. The servers must also be able to apply incoming updates.

SMS satisfies three important properties [15]; *liveness*, *agreement* and *integrity*. The *liveness* property gives that if there is a time after which two servers install only views including each other, eventually each of them will become up to date with respect to the other. *Agreement* means that servers that install the same pair of views in the same order are guaranteed to receive the same state information through invocations of their merging methods in

the period occurring between two views. The *integrity* property states that SMS will not initiate a state reconciliation without reason.

4.3 Autonomous Replication Management

The Autonomous Replication Management (ARM) framework is an extension to Jgroup which decrease the manual interaction in management activities by automatically performing tasks such as distributing replicas on nodes and recovering from replica failures, amongst others. The following sections will describe the components of ARM [15].

4.3.1 Replication Manager

The Replication Manager (RM) has four tasks.

1. Provide an interface for installing, updating and removing a service.
2. Distribute replicas in the target environment to meet operational policies for all services.
3. Collect and analyze information about failures.
4. Recover from failures.

The RM is implemented as a central controller, which means that it can make consistent decisions on replica replacement and recovery actions. The RM itself is replicated by ARM to improve its dependability. For clients to communicate with the RM without joining the RM group, the RM has two EGMI interfaces. One Management interface which enables clients to request group creation, update and removal, and one Events interface which enables external components to provide RM with events that are relevant when performing its operations.

4.3.2 Management Client

The Management Client (MC) enables the system administrator to install, remove or update services on demand. The MC is also equipped with an interface, the Callback interface, which enables the MC to subscribe to system events and provide feedback to the system administrator.

4.3.3 Replication Management Policies

A policy is a way for administrators to specify how a system should autonomically react to changes in the target environment without human interaction. ARM has two policies implemented; the distribution policy and the replication policy. The distribution policy describes how the service replicas should be allocated onto the available sites and nodes. As input the policy needs the target environment and the number of replicas to be allocated. The target environment of an ARM deployment is the nodes available. The distribution policy is specific to each ARM deployment.

The replication policy is specific for each service deployed through ARM. It describes how the redundancy level of a service should be maintained. The input needed by the replication policy is the target environment and the initial and minimum redundancy level required.

4.3.4 Object Factories

In [15] it is defined that "'The purpose of the object factory is to facilitate installation and removal of service replicas on demand."' Thus, each node in the target environment must run a Java Virtual Machine (JVM) hosting an object factory. The object factories work as bootstrap agents and enable the RM to remove or install replicas and keep track of available nodes.

The object factories each keeps a list of available nodes which enable them to respond to queries about which replicas are hosted on the node. Normally each replica run in a separate JVM.

4.3.5 Failure recovery

The failure recovery is managed by the RM. As mentioned above, failure recovery consists of three tasks.

1. Determine need for recovery.
2. Determine the nature of failure.
3. Recovery action.

Each RM has a Service Monitor (SM) which keeps track of installed replicas. As a service is deployed, a timer is associated with it in the SM. Task 1 of

failure recovery is done by monitoring the timers. If a timer expires the recovery algorithm is invoked. Each service has a separate timer. This enables RM to handle multiple concurrent failures in separate services. This may include the RM itself, as long as at least one RM replica is operational.

The two last tasks are managed by abstractions of the replication and distribution policy for the service in question. The RM keeps a specific replication policy for each deployed service and maintains the service's state. The replication policy combined with the state information determines whether a recovery action is needed.

A recovery action has three abstractions; restart, relocation and group failure handling.

Restart is used when the node's factory is available.

Relocation is used when the node is unavailable.

Group failure handling is used only when all replicas of a service have failed.

4.4 Functionality of Jgroup/ARM

What distinguishes Jgroup/ARM from other middleware platforms is the policy based autonomous replication management facility, support for partition awareness and interactions based solely on RMI.

Chapter 5

The Möbius modelling tool

Möbius is a software modelling tool developed by the PERFORM group at University of Illinois at Urbana-Champaign [21]. It allows for modelling of different submodels by different modelling formalisms which can be combined into a composed model. Hence, it can be used to model complex systems.

Möbius allows for the use of Stochastic Area Networks (SANs), as an extension to Stochastic Petri Nets. SANs offers new functionality such as input and output gates, and an extended place which enables user defined structures to be associated with a place in a Petri net [4].

The modelling and simulation in this thesis will be performed by Möbius due to its modelling flexibility and strength in modelling complex systems. Möbius generates C++ code from the models designed.

The SAN models in Möbius are built up by the following elements illustrated in Figure 5.1:

- **Place**
  The place represents the state of the system. The marking of a place is decided by the number of tokens it contains.

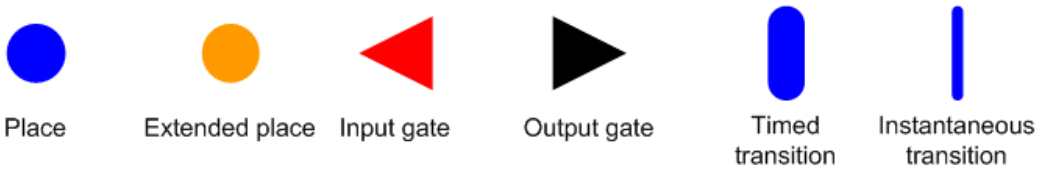


Figure 5.1: Symbols used in Möbius

- **Extended place**
  The extended place allows the model to handle representations such as structures and arrays of primitive data types [21].

- **Input gate**
  The input gate controls when a transition should be enabled and defines the marking changes that will occur when the transition completes. An input gate consists of an enabling predicate and an input function. The enabling predicate contains a Boolean expression that controls whether the connected transition is enabled. The input function defines the marking changes. The input gate is connected to an input place and a transition. A direct arc from an input place to a transition represents the gate where the transition is enabled as long as the input place contains one or more tokens.

- **Output gate**
  As the input function the output gate defines the marking changes that will occur when the transition completes. However, the output gate is associated with a single case. A transition directly connected to an output place corresponds to the output function that adds a token to each of the output places.

- **Timed transition**
  A timed transition is a transition which has a duration that affects the dependability of the system modelled. Each timed activity has an associated activity time distribution function associated with its duration. The distribution functions can be generally distributed random variables.

- **Instantaneous transition**
  Instantaneous transitions completes immediately when enabled in the system.

C++ code can be placed in the input and output gates to define more complex functionality than removing a token from or adding a token to the input place or output place respectively. When compiling the model designed, Möbius generates C++ source code which is run during simulation or numerical solution.

5.1 The model components

The dependability measures are defined in a *reward model*. The rewards are defined on the *SAN models* defined. A *study* is created for each reward model, to define the values of the global variables used in the SAN models. Each study can contain several *experiments* which can be activated and deactivated as desired. A *simulator* or *state space generator* is created based on a study.

5.2 Initialization

Global variables may be used to define rates, number of elements in the system and so on. The global variables are initialized in a study. However, if global variables are used to define the size of an array or structure all the elements are initialized to the same value. Möbius does not allow arrays and structures to be dynamic. However, global variables can be used to set the size of arrays and structures if the initialization is customized using the "'Custom Initialization"' box in the menu bar.

Below is an example that initializes a matrix, *ServiceParam*, to its initial values;

```
ServiceParam->Index(0)->Index(0)->Mark() = 3;
ServiceParam->Index(0)->Index(1)->Mark() = 3;
ServiceParam->Index(0)->Index(2)->Mark() = 1;

ServiceParam->Index(1)->Index(0)->Mark() = 2;
ServiceParam->Index(1)->Index(1)->Mark() = 2;
ServiceParam->Index(1)->Index(2)->Mark() = 1;

ServiceParam->Index(2)->Index(0)->Mark() = 3;
ServiceParam->Index(2)->Index(1)->Mark() = 3;
ServiceParam->Index(2)->Index(2)->Mark() = 2;
```

After initialization the matrix contains the following values:

$$ServiceParam = \begin{bmatrix} 3 & 3 & 1 \\ 2 & 2 & 1 \\ 3 & 3 & 2 \end{bmatrix}$$

Chapter 6

Methodology

This chapter describes the methods used to develop a dependability model for Jgroup/ARM. The main goal of this thesis is to create a state-diagram or a SPN model of the services provided by Jgroup/ARM.

As the Jgroup/ARM system is fairly complex, the first step in creating a dependability model was to simplify the system. A simplified system was modeled both as a state-diagram and a Petri net. When using the state-diagram modelling technique, the effects of the state explosion concept (introduced in Chapter 3.1) could be observed by modelling even a simplified system. Consequently, it was decided to use a SPN model to model the dependability of Jgroup/ARM.

An evaluation of available tools were made, and the choice landed on the Möbius modelling tool (introduced in Chapter 5). At first a simple system model was implemented. When the simple part was functioning correctly it was incrementally extended to include the complete Jgroup/ARM functionality.

Möbius enables the use of SANs such that a model can consist of several submodels. Hence, the Jgroup/ARM system was split in three; the hardware functionality, the services and the replication management. This split was made to simplify the modelling process and the resulting dependability model. The modelling process was simplified in that each part of the system model could be created separately even with the dependencies between the different submodels. As the functionality was distributed in three SAN submodels, each SAN became small and relatively comprehensible.

When the Petri net dependability model was finished, the Möbius simulator

was used to simulate the dependability measures of the Jgroup/ARM system. The thesis was concluded by an evaluation of the achieved simulation results.

Chapter 7

System delivery model

Section 7.1 describes the system assumptions made and the modelling technique chosen. Section 7.2 gives an example of a service delivery model for a simplified Jgroup/ARM system and Section 7.3 defines the elements that should be included in a system delivery model of the complete Jgroup/ARM system.

7.1 Modelling Jgroup/ARM

7.1.1 Suitable modelling techniques

Chapter 3 introduced the techniques for dependability modelling of distributed systems. When modelling Jgroup/ARM, it is desirable to include the dynamic behavior of a system. Hence, one of the dynamic modelling techniques should be chosen. In Section 7.2 a simplified Jgroup/ARM system is modeled using both a state-diagram and a Petri net model.

The state-diagram technique is used to perform dependability analysis of a distributed system in [10]. However, due to state explosion and the complexity of Jgroup/ARM the state-diagram grows rapidly. The hierarchical model suggested in [5] could be used to combine independent Markov models to a system.

A Petri net model for a distributed system is developed in [12]. The Petri net models have a higher entry level for understanding than the state-diagrams. However, it is a technique with strengths when it comes to describing the

No	Assumption
1	Identical processors
2	Uniform and independent failures and repairs of replicas and processors
3	Assume that all replicas have the same failure rate irrespective of which service fails
4	All times exponentially distributed
5	Assume Markov properties
7	Assume one repair unit for all services
8	Assume manual repair of processors and automatic repair of service and RM replicas
9	Both replicas and processors only have two states; working and failed (up and down)

Table 7.1: System assumptions

dynamic behavior of the Jgroup/ARM system.

7.1.2 Assumptions

Certain assumptions about the system and system times are made to make the Jgroup/ARM dependability model numerically solvable. Care must be taken when making simplifying assumptions, as a change in the system behavior due to assumptions made is undesirable. The assumptions made are summed up in Table 7.1 and discussed below.

First of all, it is assumed that all processors in the target environment are identical and have identical capacity. They fail with the same rate and their mean repair time is identical. The processors' failure and repair processes occur independently.

The system services can be initialized with different replication policies[1]. However, all service replicas[2] are assumed to have the same failure and repair rate. It is assumed that all system times are exponentially distributed.

It is assumed that the next state of the system is dependent only on the

[1]The replication policy gives the initial and required number of replicas
[2]Including RM replicas.

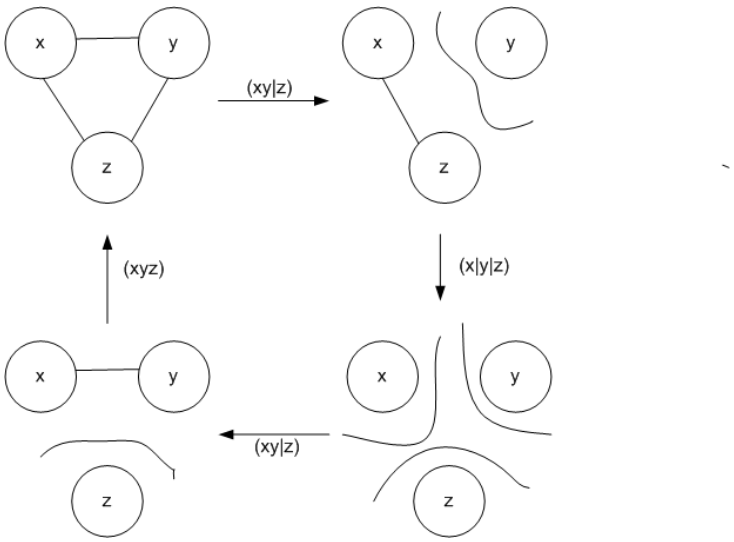


Figure 7.1: Example trajectory of partitioning, from [15]

current state of the system and the events that occur and not on the history of events in the system. Thus, it is assumed that the Markov property applies to the system. We assume the Markov properties to be able solve the problem numerically. However, both state-diagrams and Petri net models are considered. Using Petri nets and simulation, the Markov assumption is no longer necessary. Assuming Markov properties will still be possible, but the distributions may also be changed.

It is assumed that the system contains one repair unit, and upon replica creation only one service replica is created. Two different repairs exist, manual and automatic repair. The former is used for processor failures and the latter for restoring service and RM replicas. The mean duration of the former is generally longer than the latter.

7.1.3 Partitioning

By partitioning it is meant that a failure in the communication medium separates the target environment into two, or more, partitions. Jgroup/ARM offers functionality for all partitions to continue operation and merge their state when the partitioning is repaired. However, partitioning introduces an enormous complexity to the system model.

Figure 7.1 from [15] illustrates a sample sequence of partitioning events, in rapid succession, for a system of three processors; x, y and z. It can be seen

that a great number of possible sequences exist. Adding the partitioning functionality to a state-diagram of Jgroup/ARM will explode the number of states required.

A state-diagram showing parts of the state space of Jgroup/ARM considering partitioning and view cardinality is presented in Figure 7.2[15]. The target system has three processors; x, y and z. The replication policy initializes the system with three replicas of the Monitored Subsystem (MS), and requires two working replicas as a minimum for the system to deliver required services.

Each state is identified by the number of replicas in each partition and the number of members in the various views. A partition is illustrated by a | symbol. The number of members in each view is given by the number in parenthesis, concurrent views are possible and indicated by the + symbol. As illustrated in the figure, ARM will try fulfilling the replication policy of the system. When all processors are partitioned, ARM will order creation of a new replica in each of the partitions. The extra replicas are removed when the partitions merge.

The system states might be stable or unstable, the stable states are bold in Figure 7.2. A state is stable if ARM does not need to increase or decrease the redundancy level.

The state-diagram does not consider factors such as replica or processor failure/repair. Adding these dimensions to the diagram would explode the number of states needed. However, the greatest problem with partitioning is not its effects on dependability, but rather the complexity introduced by all the extra signaling and the management functionality needed. Thus, when designing a dependability model of Jgroup/ARM the partitioning is not considered.

7.2 Example system delivery model

An evaluation of the dependability parameters of a service deployed in the Jgroup/ARM system was presented in [16]. The system states entered while running simulations was illustrated in a state-diagram[3]. An idealized version of the state-diagram is depicted in Figure 7.3 and will be used as a basis for a state-diagram and a SPN dependability model for a service deployed in Jgroup/ARM.

[3]The state-diagram only included states and events relevant to the service monitored.

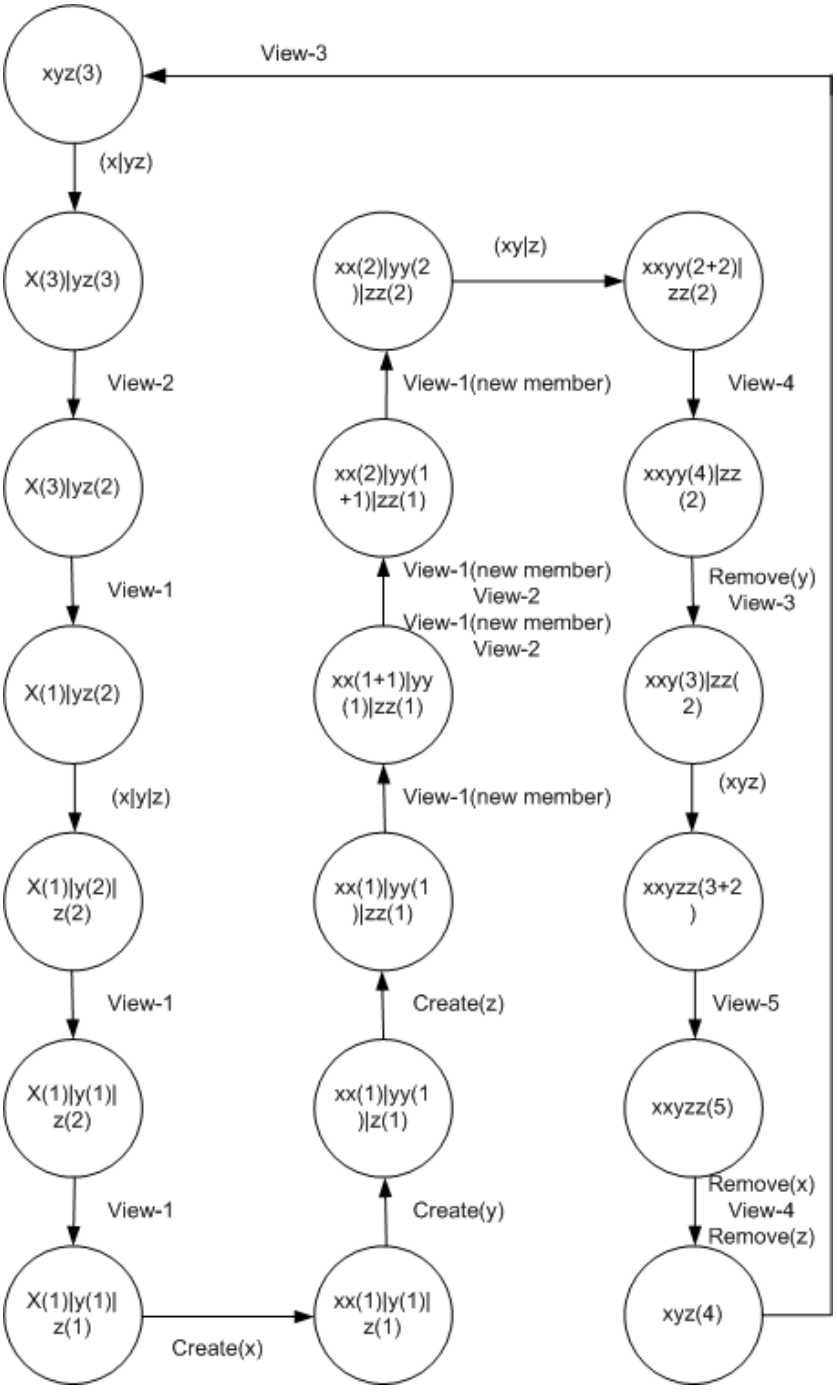


Figure 7.2: Example state-diagram considering partitioning and view cardinality, modified from [15]

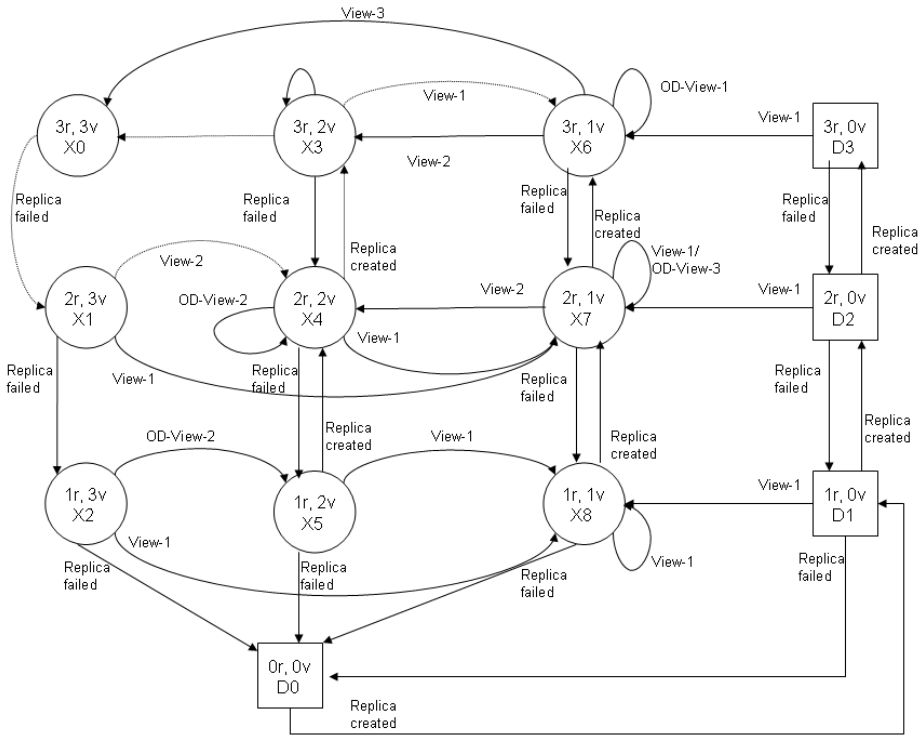


Figure 7.3: State-diagrams showing the view cardinality and number of working replicas of a service in Jgroup/ARM, modified from [16].

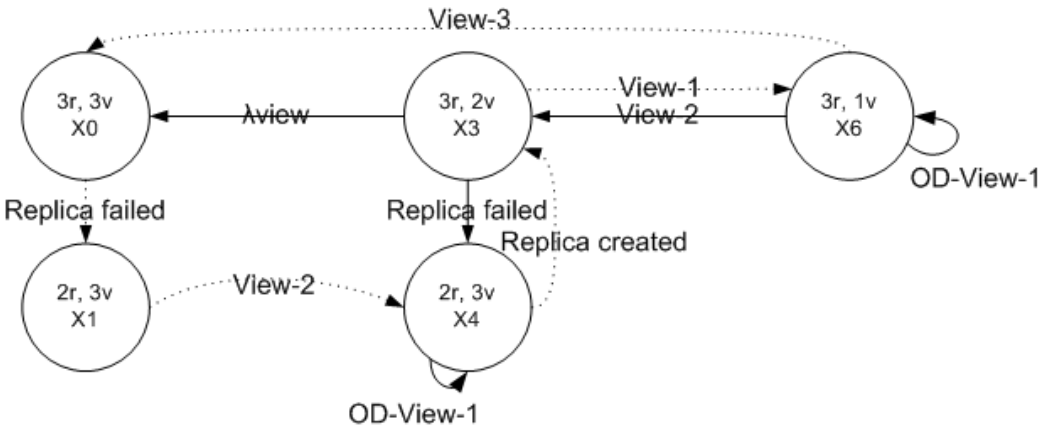


Figure 7.4: State diagram showing the normal trajectory of the MS, excerpt from Figure 7.3

The target system, as introduced in [16], consists of a cluster of 8 identical processors. However, the number of processors considered in the following models is, for simplicity, 4. The service is the MS and needs 2 replicas to deliver the required service. The initial number of replicas is 3.

Figure 7.3 illustrates two modelling dimensions; the number of working replicas and the view cardinality of the group. However, the processor capacity is assumed to be unbounded. The diagram illustrates only the state of the MS, independently of the states of the ARM and additional services. Circles and squares represent available and unavailable states respectively. Each state is identified by X# and a tuple (xr,yv), where x is the number of installed replicas and y is the number of members in the current view of the server group [16]. Only events that may affect the availability of the service are considered in the diagram, that is view changes (View-i, where i is the cardinality of the view), replica failures and replica creations.

The normal scenario is the sequence of events indicated by dashed arrows, starting and ending in X0 as illustrated in Figure 7.4. When a replica failed event occurs the MS state is changed from X0 to X1. A View-2 event changes the state to X4, a replica created brings the MS to state X3. The new replica has a singleton view and hence sends a View-1 event to the ARM and consequently enters state X6. A View-3 event brings the MS state back to X0. Further explanation of the model, from [16], exceeding the normal trajectory is left out of this section as only the normal trajectory is modeled in the following section.

In Figures 7.3 and 7.4 two important dimensions are left out of the model,

namely processor capacity and partitioning. With a limited number of processors and a distribution policy stating that no two replicas of the same service can reside on the same processor, the processor capacity is a limiting dimension.

Jgroup/ARM offers support for partitioning and merging, as described in 4. However, this option introduces complexity that explodes the number of system states. As discussed above partitioning is left out of the system model due to the scope of this thesis.

7.2.1 State-diagram model of the MS

Figure 7.5 shows the state-diagram for the normal trajectory (Figure 7.4) of a MS in a Jgroup/ARM system with a cluster of 4 processors. The initial number of MS replicas is 3, 2 of which is required to be up for the service to be up. The model has three dimensions; number of replicas vertically, view cardinality horizontally and processor capacity diagonally. In these models it is assumed that a working processor has indefinite capacity.

Each state is identified by a number, i, and (xr, yv, ze), where x is the number of replicas, y is the view cardinality and z is the number of available processors (i.e. processors that are up and currently not holding a replica of the MS). Both the replicas and the processors may fail. It is assumed that a replica residing on a failed processor is moved to one of the extra processors if such a processor exists.

The figure does not illustrate the complete state space for the system. As a simplification, different failed states are merged to one failed state. However, in reality the system will enter different failed states depending on the number of replicas, the view and the number of additional processors. The dotted lines illustrate that failures and repairs can occur even during a system failure.

To find the model's dependability characteristics one can either solve the model numerically (by assuming Markov qualities and creating balance equations) or solve it by simulation. However, the simplest tools for solving dependability models are based Petri nets. Thus, in Section 7.2.2 the state diagram is converted to a Petri net.

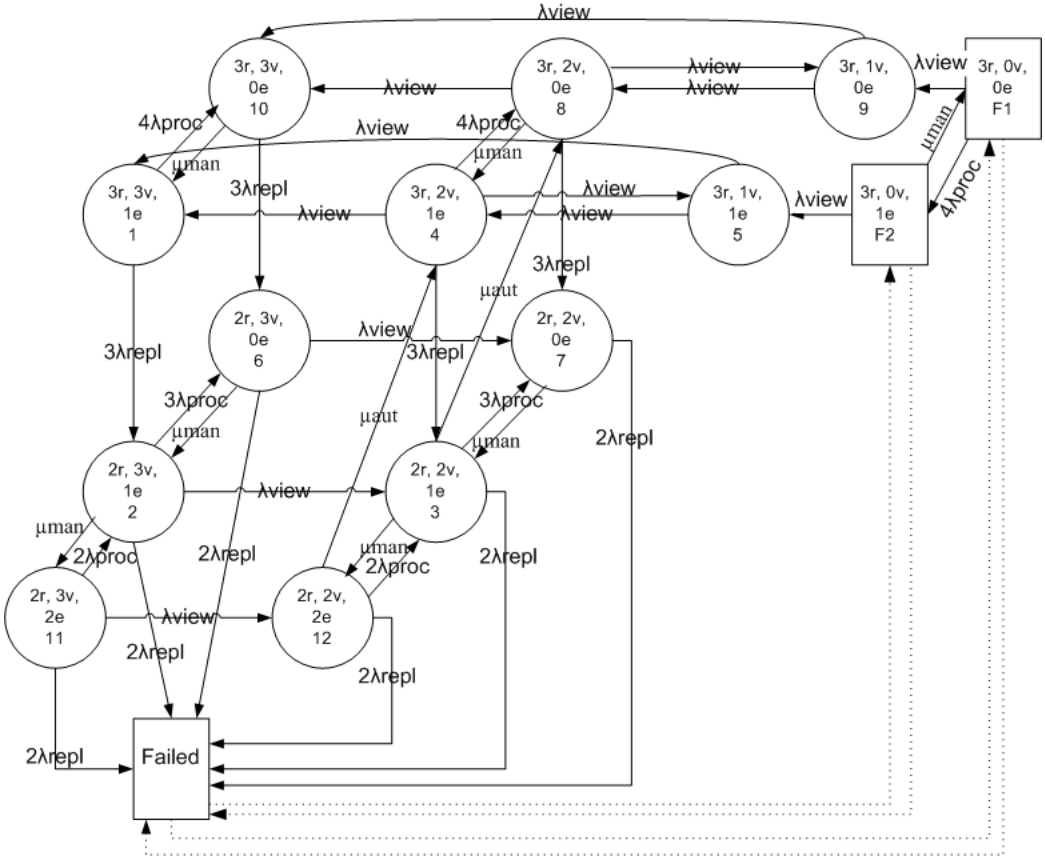
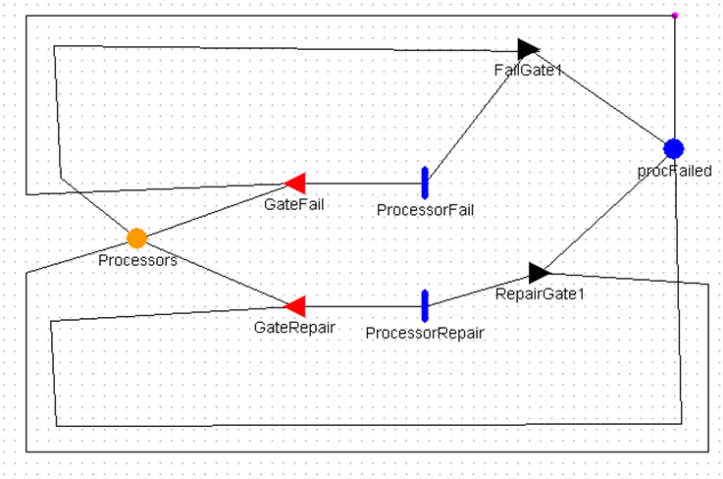
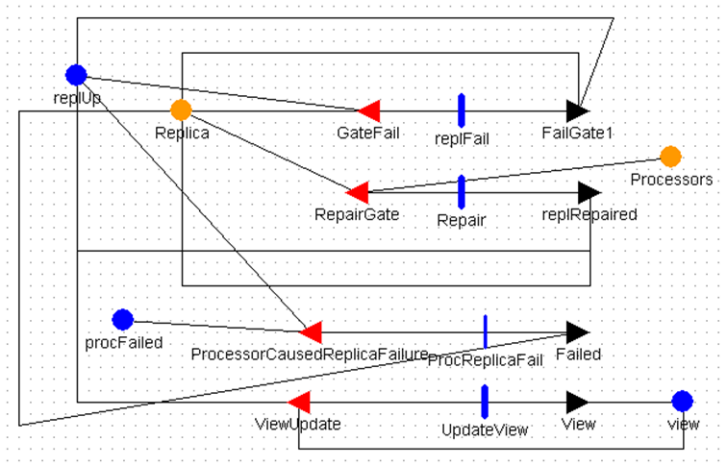


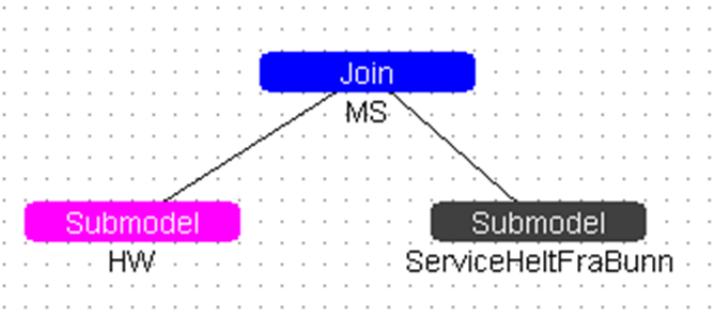
Figure 7.5: State diagram for a service in Jgroup/ARM



(a) The hardware submodel



(b) The service submodel



(c) The composed model

Figure 7.6: Petri net diagram for the MS

7.2.2 Petri net model of the MS

The Petri net in Figure 7.6 is a convertion of the state model in Figure 7.5. As can be seen from the figure, the conversion is not done directly. Using direct conversion, with one place for each state, the model quickly grows incomprehensible. Hence, the direct translation design in rejected. The use of Möbius enables use of more refined models.

The system is separated in two parts; the hardware submodel (Figure 7.6(a)) which models the failure and recovery of a processors and the service submodel (Figure 7.6(b)) which models the view updates, failure and recovery of service replicas. The two submodels are joined by a join-node in a composed model (Figure 7.6(c)).

As can be seen from both Figures 7.5 and 7.6 even a simple scenario quickly grown complex, even without considering the partitioning dimension.

7.3 System delivery model

When analyzing a system, the easiest way to define a system's dependability is to check whether or not the system delivers the services it is supposed to deliver. This can be measured by the availability and the mean time between failures (MTBF). The measurements could eventually be extended to include the performability as well as system down times, with rate rewards for states and pulse rewards for transitions.

Questions to be asked when evaluation the dependability of a system are stated below.

- Does the system deliver the promised services?
- Does the system generate replicas as required?
- Does replica failures cause service unavailability, either due to failure of all replicas or too many replicas?
- How long does it take from a replica fails to it is replaced by a new replica?
- Does the system have enough resources to provide sufficient amount of replicas to all the services? All services can not be replicated on all processors, the resources must be distributed. Certain services are needed for the system to be up and should be prioritized?

For a Jgroup/ARM system to be working several components have to be up. The first issue is the system resources. The processor cluster serving the Jgroup/ARM system is of finite size. Due to the defined distribution policy in Jgroup/ARM, no two replicas of the same service can reside on the same processor. A service replica can not be created without sufficient resources in the system. Hence, concurrent failure in too many processors may lead to system failure.

Certain defined services must also be up for the Jgroup/ARM system to be working correctly. Thus, the service may become unavailable due to rapid replica failures. If the system can not produce replicas fast enough, the service will go down. If the service is amongst the services required by the system, a service failure will result in system failure.

Coverage is the third issue. The system must be able to handle the failures in the system elements/processes properly. Insufficient failure handling may also cause system down time. In Jgroup/ARM this functionality is maintained by the ARM framework. The replication manager itself is replicated by Jgroup. However, too rapid failure in the RM replicas, or the processors they reside on, might lead to system failure as well.

Thus, the Monitored Subsystem (MS) of the Jgroup/ARM system includes;

- Replication Management (ARM framework)
- Group membership service (view updates)
- Given services
- Processors

The core components of Jgroup/ARM and their interactions is illustrated in Figure 4.1. The system requirements may vary from implementation to implementation. In this thesis it is assumed that more than one processor, more than one RM replica and one, or more, replicas of given services must be available for the system to be available.

As described in Chapter 3 the static models can not be used to model the dynamic behavior of the system. For this reason this thesis will focus on the dynamic modelling methods; state-diagrams and Petri-nets. However, as Section 7.2 showed, the state-diagrams rapidly grow complex when modelling in more than one dimension. As the Jgroup/ARM framework requires four dimensions, Petri net diagrams are the tool chosen for modelling. As mentioned above, the four necessary dimensions are *view cardinality*, *processor capacity*, *replication capacity* and *partitioning*. Partitioning is left out of the

model due to the complexity it introduces, as described in Chapter 7.1.3. The first three dimensions were illustrated in Figure 7.5.

A dependability model based on the factors defined in this chapter will be developed in the following chapters.

Chapter 8

The dependability model

This chapter will present the resulting dependability model, as modeled in Möbius. In Section 8.1 the modelling assumptions are introduced.The model used is described in Section 8.2 and the important places used are listed in Section 8.6.

8.1 Assumptions and simplifications

As for modelling any system, certain assumptions are needed when developing a dependability model of Jgroup/ARM. The true system functionality is to complex; hence assumptions are made to make the model solvable.

The number of replicas a processor can host is defined and assumed identical for all processors in the target environment.

When modelling, it is assumed that the components of the ARM framework[1] replicas are co-located on the same nodes. This assumption was also made in [15] and is a reasonable assumption as neither RM nor DR can function correctly without the other. Hence, by assuming co-location the ARM framework functionality is either up or down.

In [16] stratified sampling with fault injection is used to observe the states a Jgroup/ARM system enters during operation. It is observed that some outdated view change events occur during system operation. However, an ideal system is assumed when creating the dependability model of Jgroup/ARM.

[1] The Dependable Registry (DR) and the RM

No	Assumption
1	All processors have a capacity defined by a global variable.
2	DR and RM are co-located.
3	Assume an ideal system with no outdated view-change events.
4	Partitioning is not considered.
5	A replica can only be created on a processor that is up and currently not holding a replica of the service considered.
6	No repair can not occur before the first failure.
7	Upon service replica repair only one replica is created at a time.
8	Can only create replica for a service group with a correct view.
9	Never create more replicas than the initial number.
10	No communication delay.
11	Priority is not implemented.
12	Multicast and leadercast are not implemented.
13	The services are not associated with a given RM.
14	All replicas can send view update messages.
15	Both software and hardware have only two states, up and down. This means that only crash failures are considered.

Table 8.1: Modelling assumptions

Thus, no outdated events will occur. It is assumed that all service replicas can send view updates because no group leader selection is implemented.

In the model it is assumed that no repair of processor or service replica can occur before a failure has occurred. Consequently, the number of replicas of a given service will never exceed the initial number.

A service's group view cardinality needs to equal the actual number of currently active replicas for a service to be eligible for a new replica creation. A replica can be created only by being assigned to working processors, and due to the distribution policies of Jgroup/ARM no two replicas of the same service can reside on the same processor. Hence, for a replica to be assigned to a processor the processor must currently be working and contain no replicas of the service in question.

In Jgroup/ARM, priority is given to creation of RM and other services in the MS. However, this functionality has not been implemented in the dependability model. The only difference between services and the RM is that the services can not create a replica without "'approval"' from the RM. The RM on the other hand can create a replica when the number of working replicas is below a given threshold.

Jgroup/ARM has different algorithms for external group method invocations; anycast, multicast and leadercast. In the model developed only anycast is implemented.

In the model it is assumed that all the RM replicas have knowledge on all the service replicas. A given service is not associated with a RM replica, it is assumed that any RM replica can monitor its status.

The communication substrate is assumed to be free from errors and delay.

The assumptions made when modelling Jgroup/ARM are listed in Table 8.1.

8.2 The model

The dependability modelling started with a model with four processor and one single monitored service as in Section 7.2. When the elementary model worked it was expanded to also cover several monitored services and the ARM framework.

The system structure is broken down to three submodels. One SAN submodel for the hardware (the processor failure and repair), one SAN submodel for

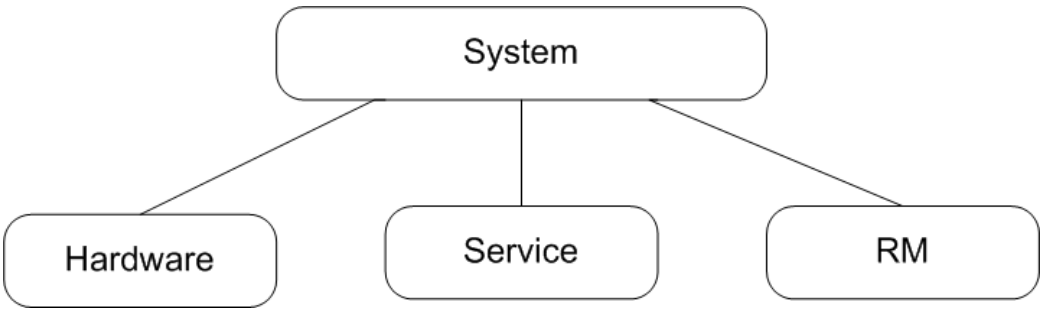


Figure 8.1: Structure of the model in Möbius. One Join-node and three SAN submodels.

the service monitored (replica failure, view change events) and one for the replication management.

This split was made to simplify the model, as the functionality easily can be classified as hardware behavior, service related behavior or management functionality.

The modelling tool used is the Möbius modelling tool which was described in Chapter 5. Figure 8.1 shows the final design of the dependability model. The three SAN models are composed to a complete system by a Join-node. The source code for the SAN models are attached in Appendix A. The functionality contained in the different SANs is described in the following sections. The main points are emphasized, followed by a detailed description.

8.2.1 The hardware submodel

Basically, the hardware SAN models the failure and repair of processors and notifies the other submodels of the state of the target environment. It contains four extended places; *Processors, ProcNrFailed, ProcNrFailedService and ProcFixed*. The places and transitions in the hardware SAN are described below.

***Processors* - used to keep track of failed and working processors.**

Processors is the main extended place of the submodel. It contains an array with size equal to the numbers of processors in the processor cluster. 1 in position i indicates that processor i is up. It is assumed that all processors are up initially.

The number of processors currently working could also have been modeled by

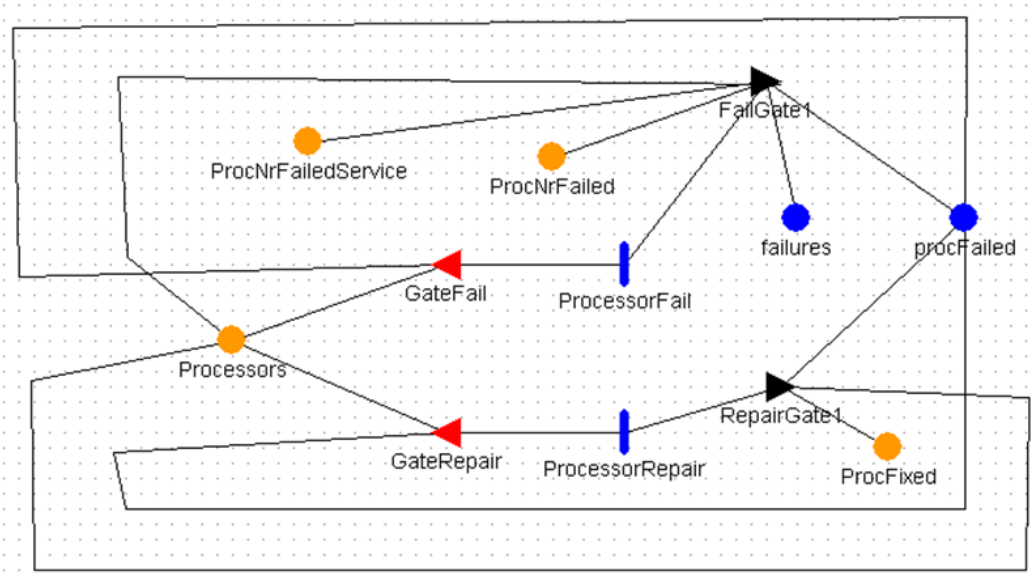


Figure 8.2: The hardware SAN.

a place with one token for each working processor. However, the extended place containing an array is chosen because knowledge on the number of failed and working processors is not sufficient when modelling Jgroup/ARM. Information about currently failed processors is important, which is why an array with one element for each processor is used.

***ProcNrFailed* and *ProcNrFailedService* - used to notify the RM and the Service SANs of processor failure.**

ProcNrFailed contains an array with two elements. The first element is either 0 or 1, where the former represents no new processor failure and the latter represents that a failure has occurred. The second element contains the number of the failed processor. Both elements are initialized to 0. *ProcNrFailed* is used to notify the replication manager submodel of a processor failure. Figure 8.3 illustrates how the HW submodel informs the RM and Service submodels of a processor failure. *ProcNrFailedService* is identical to *ProcNrFailed*, the only difference being that it is used to notify the service submodel of the processor failure.

Thus, two different places are used to notify the two submodels of processor failure. This might seem bothersome, but the need for two places is caused by the design of the model. When the places are given values they enable instantaneous transitions in the two other submodels. How the other submodels react to the transitions will be described in Sections 8.2.2 and 8.2.3,

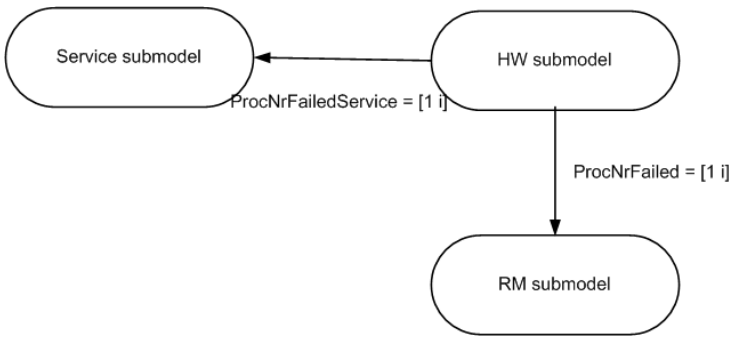


Figure 8.3: The hardware submodel signals the RM and Service submodels that a processor has failed

the main point here is that after some updates has occurred the output gates of the respective transitions "'reset"' the values of the places. By using only one place for both submodels only one of the submodels will react to the processor failure. Hence, two places are used to notify both submodels when a processor fails.

***ProcFixed* - used to notify the RM when a processor is repaired.**

ProcFixed contains an array of two elements. The first is 0 when no processors have been repaired and 1 when a processor has been repaired. The second element is the number of the currently fixed processor. This extended place is used to notify the RM that a processor is again available.

The submodel also contains a place called *procFailed*. This place is initialized to zero and contains a token for every processor that is failed at any time. It is incremented when a processor fails and decremented when it is repaired.

Failure and repair of processors are the two transitions in the SAN.

Two timed transitions are included in the hardware submodel; *ProcessorFail* and *ProcessorRepair*. *ProcessorFail* has an input gate stating that for the transition to be enabled one or more replicas need to be up. The firing of the transition has an exponential distribution with rate
(number of working processors + 1)· λ_{proc}. The output gate, FailGate1, defines the consequences of the transition. A processor, i, is chosen randomly amongst the currently working processors. Element i in Processors is set to 0, procFailed in incremented by 1 and *ProcNrFailed* is updated with the currently failed processor. The reason for the "'+ 1"' in the failure rate is that Möbius performs the defined functionality in both the input and output gates before the transition fires. Hence, by the time the failure rate is

calculated, the number of working processors has already been decremented.

The other timed transition, *ProcessorRepair*, is also exponentially distributed, with rate μ_{man} which symbolizes manual repair. The transition can only fire if one or more processors are currently failed. A processor, i, is randomly selected amongst the failed ones and position i in *Processors* are set to 1. Consequently, *procFailed* is decremented by one and *ProcFixed* is updated with 1 in element 0 and the currently fixed processor number in element 1.

8.2.2 The service submodel

The service submodel basically keeps track of the failure and repair processes of the services. The main functionality is to capture replica failures, inform the RM of changes in the group's view and to react to replica creation orders from the RM.

The choice of modeling all the services in one submodel, instead of modeling one service in a SAN, and then use the replicate functionality in Möbius, is done because it is desirable to be able to separate between the services. This is necessary when defining reward models. The Jgroup/ARM system depends on several services to be up for the system to be considered as working. The design of the submodel is shown in Figure 8.4 and described below.

***ServiceRepliceDist* - matrix that maps service replicas to processors.**

The extended place *ServiceRepliceDist* contains a matrix which maps service replicas to processors. The number of rows correspond to the number of services and the number of columns to the number of processors. The matrix consists of 1's and 0's. 1 in element i, j represents service i having a replica on processor j.

***LoadDist* - represents the load of each processor at any time.**

The extended place *LoadDist* represents the load (the number of service replicas) on the different processors. The service and RM submodels increment the marking of a given processor upon replica creation. The markings are used by the replication manager to decide where to create a new service replica, and hence it will be further described in Section 8.2.3.

***newReplUp* and *viewNew* - keeps the real replica count and the current view of each service group respectively.**

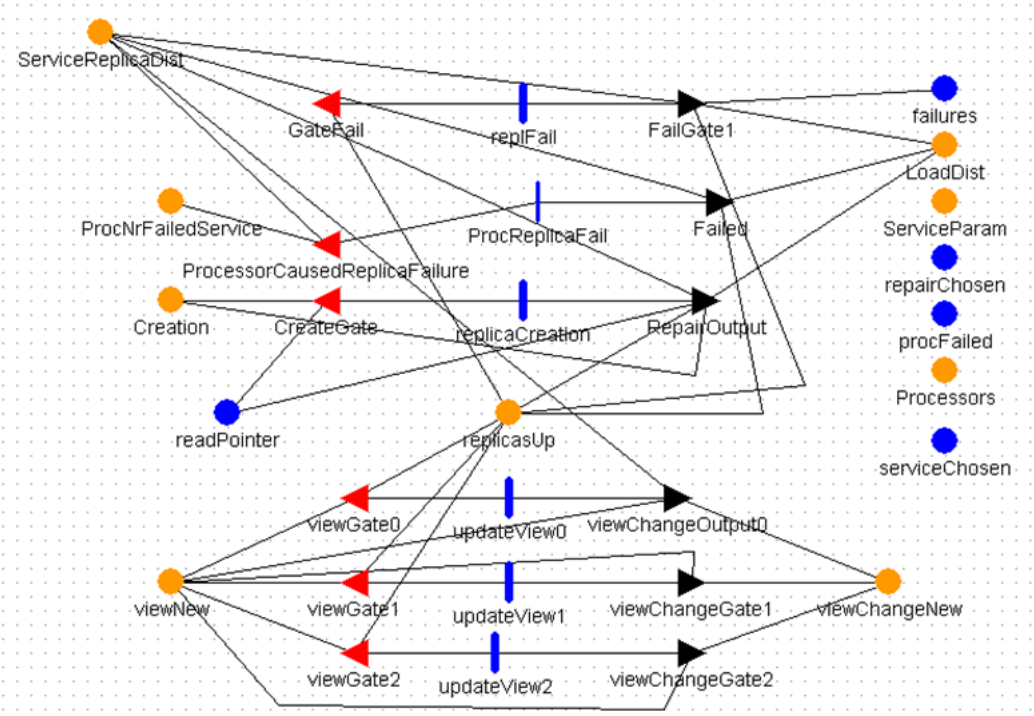


Figure 8.4: The service SAN.

newReplUp contains an array with the size of the number of services currently in the system. Each array element represents the number of replicas of the corresponding service. E.g if element i is 3, service i has 3 working replicas.

An extended place, *viewNew*, is used to store the cardinality each service has of its view. It contains an array with one element for each service in the system.

Replica failure can be caused by failure of the replica or by failure of the processor the replica resides on.

Two different events can cause failure of a service replica; software failure and failure of the processor the replica resides on. The failure of a replica due to software failure can occur as long as the given service has one or more working replicas.

A random service, s, with one (or more) replica(s) is chosen in the input gate of the transition *replFail* and its service number is stored in the place *serviceChosen*. The failure transition, *replFail*, is exponentially distributed with rate $(newReplUp[s] + 1) \cdot \lambda_{repl}$, because each of the working replicas fail with rate λ_{repl}.

In the output gate, a working replica, r, of the chosen service is randomly selected and stored in *replicaChosen*. Consequently the output gate updates the affected extended places. In *newReplUp* the element of the service chosen is decremented by one, in *LoadDist* the element of the processor that contained the chosen replica is decremented and finally the *ServiceReplicaDist* is updated by setting *ServiceReplicaDist*[s][r] = 0. This completes the replica failure transition.

As mentioned above, replicas can also fail due to processor failure. This is captured by the action *procReplicaFail* which is enabled by the first element of *ProcNrFailedService* equaling one. When the enabling condition is true, *ServiceReplicaDist* is scanned, and each service having a replica on the failed processor gets its element in *newReplUp* decremented and the corresponding element in *ServiceReplicaDist* set to zero, as the given processor is failed and consequently no longer contains any replicas.

***Creation* - a queue used by RM to signal that a service can create a replica. *replicaCreation* is the transition in the Service SAN which creates a new replica.**

To create a new replica, the service submodel needs to receive a "'signal"' from the RM, as the RM decides which service creates a replica when and where. The signal is sent by setting the elements of *Creation* to certain

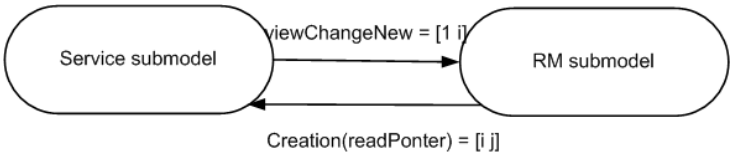


Figure 8.5: The service submodel signals the replication manager submodel that a view change has occured and the replication manager enables the service to create a replica.

values. The service submodel keeps a pointer, *readPointer*, which points to the first element of the queue. Each row in *Creation* contains two columns initialized to -1. The first column gives the number of the service that can create a replica, the second element gives the processor the replica can be created on. Figure 8.5 illustrates how the service submodel interacts with the RM submodel.

The input gate of the timed transition *replicaCreation* checks that the first element of the queue has a valid service value. The transition time has an exponential distribution with rate μ_{aut}. The average time of a automatic repair is normally shorter than the average time of a manual repair. It is assumed that no manual action is required to create a new replica. Manual repair is only required in processor repair as described in Section 8.2.1.

In the output gate of the transition, the chosen processor's element in *Load-Dist* is incremented by one as a new replica now resides on that processor. Consequently, *newReplUp* and *ServiceReplicaDist* are also updated. The former gets the element corresponding to the chosen service incremented by one, and the latter gets the element of the chosen service on the chosen processor set to one. To disable the instantaneous *replicaCreation* transition, both elements of *createReplica* are reset to -1, and the *readPointer* is incremented (modulo the number of rows) to point to the next element in the queue.

***updateView* - transition which updates a service's outdated view. The Service SAN uses extended place *viewChangeNew* to notify RM of a view change event.**

The last transition in the service submodel is the view change transition. For simplicity, a transition is made for each service in the system[2]. Hence, in Figure 8.4 there are three viewChange transitions which performs the same functionality. The input gate compares the view cardinality and the actual numbers of replicas up (from *viewNew* and *newReplUp* respectively) and

[2]Due to difficulties changing the design in Möbius, as described in 11.3 on page 93

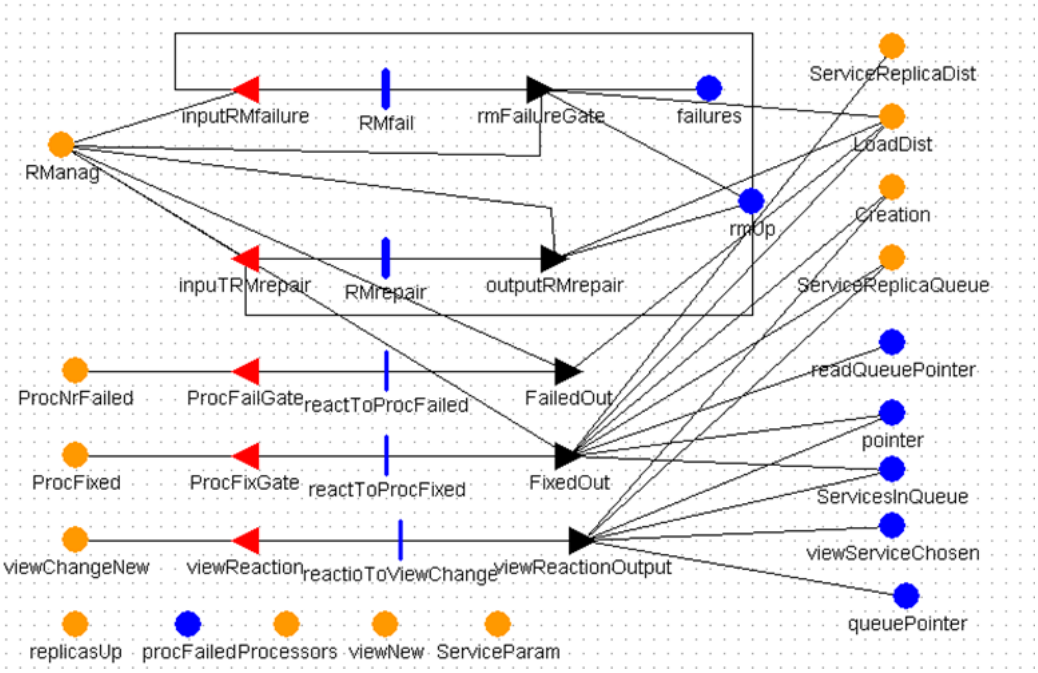


Figure 8.6: The replication manager SAN.

enables the timed transition *updateView* if the two cardinalities differ. The transition time for service i is assumed to be exponentially distributed with rate (replicasUp->Index(i)->Mark() + 1)· λ_{view}.

Next, the *viewNew* is updated for the chosen service to equal the current working replicas for that service. For a given service i that means

$$viewNew[i] = newReplUp[i].$$

Consequently, the extended place *viewChangeNew* is updated. *viewChangeNew* corresponds to the *ProcNrFailedService* place in the hardware model, and contains an array of two elements. The first elements indicates a view change by its first element equaling one. The second element the service in which a view change has occurred. The place is used to notify the RM of the occurred event, as the RM is the system component which reacts to view change events.

8.2.3 The replication manager submodel

The RM SAN contains the ARM framework functionality. The RM itself is replicated and its replicas can fail and be repaired as other service replicas. Additionally, the RM submodel reacts to view updates, processor failures and processor repairs. The last important task of the RM submodel is to signal to the services when and where a replica can be created. Figure 8.6 shows the design of the replication manager submodel.

***RManag* - maps RM replicas to processors.**

The extended place *RManag* contains an array with one element for each processor. The number of initial RM replicas is given by the global variable *initRM*, and stored in the place *rmUp*, and the initialization of the array is done in the custom initialization field. However, as with the service, 1 in element i represents that a RM replica resides on processor i.

RM replica failure and repair - the RM can create a RM replica as long as available processors exist.

As with a service replica, the RM replicas can fail and be repaired. The *RMfail* transition is identical to the replica failure in the service submodel, and thus will not be described in detail here. However, unlike in the service submodel, there is only one replicated RM, and the only decision made is which of the RM replicas fails. The places affected by this transition is *LoadDist* where the element of the processor the chosen replica resided on is decremented, *rmUp* which is decremented by one and *RManag* where the chosen replica element is set to 0.

Unlike the creation of a service replica which needs a "'go-signal"' from the RM to fire, a RM replica can be created as long as the number of working RM replicas is less than the initial number of RM replicas. The repair process is exponentially distributed with repair rate μ_{man}. A processor is chosen randomly amongst the processors not currently holding a RM replica. However, the processor chosen must have sufficient capacity to handle an additional service replica. After firing the transition, *rmUp* is incremented, *LoadDist* for the chosen processor is incremented and the chosen processor element in *RManag* is set to 1.

***viewChangeNew* - tells the RM submodel whether or not any new view changes have occurred in the Service submodel. The RM reacts to the view change if it is enabled.**

The extended place *viewChangeNew* was introduced in Section 8.2.2. It

contains an array of two elements, the first element is set to zero when there is no new view change, and to one when there is. The second element is set to the number of the service with a view change. The instantaneous transition *reactionToViewChange* fires whenever the first element of *viewChangeNew* equals one.

The output gate performs several actions. First of all, a new replica is only created if the replica count of the given service does not exceed the initial number of replicas. Secondly, an array of processors where the chosen service (given by the second element in *viewChangeNew*) can create a replica is updated. A processor is added to the array if it is currently working, its current load does not exceed its defined maximal load and it does not contain a replica of the chosen service. If there are several eligible processors in the cluster, the one with the least heavy load is chosen, and the extended place *Creation* is updated. The element pointed to by the pointer is set to the number of the service to get a replica created and the second element gives the number of the processor to create the replica on.

However, if there are no eligible processor, the service is added to a queue in *ServiceReplicaQueue* (if it is not already in it). Every time a service is added to the queue the place *ServicesInQueue* is incremented. This place will be used to determine whether the queue is empty or not.

The final action in the output gate is to reset the *viewChangeNew* to disable the instantaneous transition. Both the first and the second element of the array is set to zero.

***reactToProcFailed* and *ReactToProcFixed* - transitions that handle the failure and repair of processors in the target environment.**

The effect of a failed processor is captured in the *reactToProcFailed* instantaneous transition. The transition is enabled by the first element of *ProcNrFailed* equaling one. The second element of *ProcNrFailed* gives the number of the failed processor. This extended place was described in Section 8.2.1, and the hardware model is also where its values are set. The output gate of the transition immediately sets the *LoadDist* element of the failed processor to zero, as a failed processor can not hold any service replicas. *RManag* is scanned to see if the failed processor contained any RM replicas. If so, *RManag* is updated by setting the processor element in question to zero.

The last instantaneous transition in the RM submodel is *ReactToProcFixed*. This transition is enabled when the first element of *ProcFixed* is 1. This happens, as described in Section 8.2.1, when a processor is repaired. When *ReactToProcFixed* fires it checks whether or not there are any services in the

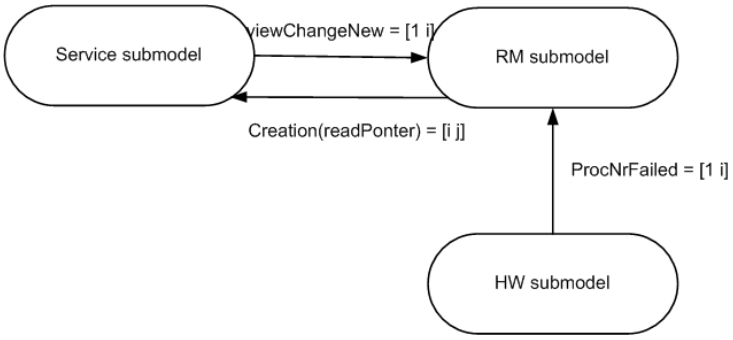


Figure 8.7: The replication manager interacts with the two other submodels.

queue waiting for a processor to place a replica on. If there is, that is if the mark of *ServicesInQueue* is greater than zero, *createReplica* is enabled for the service first in line (FCFS) on the processor that has been repaired. Reset *ProcFixed* to disable the instantaneous transition.

Adding a service to the model would require manually changing the initialization of places *ServiceReplicaDist* and *ServiceParam* in the custom initialization field, as Möbius does not support dynamic initialization of arrays with size defined by global variables.

8.3 The mapping problem

To make the mapping of the services to distinct processors clearer, this section presents an example mapping. All the time considering a system of four processors, the array [1 1 1 1] in the *Processor* place will represent the state where all four processors are up. This solves the problem of knowing which processors are up at any given time.

As explained above the extended place *serviceReplicaDist* contains the distribution of service replicas on processors. With all processors being operative an example matrix may look like the one below. In this example three services are considered. Service 0 has replicas on processors 0, 1 and 2, service 1 on processors 1, 2, and 3 and service 2 on processors 0, 2 and 3.

$$\begin{bmatrix} 1 & 1 & 1 & 0 \\ 0 & 1 & 1 & 1 \\ 1 & 0 & 1 & 1 \end{bmatrix}$$

Place	Initial value
Processors	[1 1 1 1]
Rmanag	[1 1 1 0]
LoadDist	[3 3 4 2]
viewChangeNew	[0 0]
ProcNrFailed	[0 0]
ProcNrFailedService	[0 0]
ProcNrFixed	[0 0]
newReplUp	[3 2 3]
viewNew	[3 2 3]
ServiceParam	$\begin{bmatrix} 3 & 3 & 1 \\ 2 & 2 & 1 \\ 3 & 3 & 2 \end{bmatrix}$
ServiceReplicaDist	$\begin{bmatrix} 1 & 1 & 1 & 0 \\ 0 & 1 & 1 & 0 \\ 1 & 0 & 1 & 1 \end{bmatrix}$
Creation	$\begin{bmatrix} 0 & 0 \\ 0 & 0 \\ 0 & 0 \end{bmatrix}$
rmUp	3
procFailed	0
pointer	0
readPointer	0

Table 8.2: Initial values of example Jgroup/ARM system

Correspondingly, the RM replicas are associated with processors through the extended place *RManag*. The place contains an array with one element for each processor. Each element value indicates whether or not a RM replicas resides on the respective processor. *RManag* = [1 1 1 0] represents a RM having replicas on processors 0, 1 and 2.

Hence, the mapping problem is solved and the effects of processor failure can be captured both in the service and the RM submodels.

8.4 System events

This section presents an example system and the main system events. The system has a target environment with 4 processors, 3 services and 3 RM

	Initiation submodel	**Affected submodel**	
	HW	*RM*	Service
Transition	ProcessorFail	ReactToProcFail	ProcReplicaFail
Enabling function	One or more processors up	Failure of a processor ProcNrFailed enabled by HW	Failure of a processor ProcNrFailedService enabled by HW
Places changed	Processors, procFailed, ProcNrFailed	LoadDist, rmUp, RManag, procNrFailed	ServiceReplicaDist, replicasUp

Table 8.3: Processor failure consequences

replicas. The initial marking is defined in Table 8.2. The following sections will use this initial marking and demonstrate marking changes in the different submodels when a processor fails, a service group updates its view and a processor is repaired.

Each section starts with a short summary of the changes caused by the system event and ends with a detailed example of the event.

8.4.1 Processor fail

RM reacts immediately when a processor fails, as long as at least one RM replica is up. It removes the RM replica residing on the failed processor and changes the processor's load. The service SAN reacts in a similar matter. As a processor has failed the service submodel removes every replica residing on the failed processor and changes the replica count for the services which had a replica on the failed processor.

Example

Now assume that processor 1 fails. As listed in Table 8.3 this will affect the places *Processors*, *ProcNrFailed*, *ProcNrFailedService* and *procFailed*. The latter is incremented and the two former are changed to reflect the failed processor;

$Processors = [1\ 0\ 1\ 1]$, $ProcNrFailedService = [1\ 1]$ and $ProcNrFailed = [1\ 1]$.

Table 8.3 illustrates that the transition ReactToProcFailure in the RM submodel is enabled when the first element of *ProcNrFailed* equals 1. The actions

	Initiation submodel	**Affected submodel**	
	Service	*RM*	*Service*
Transition	updateView	reactionToViewChange	replicaCreation
Enabling function	One, or more, services has an inconsistent view of its group	Service enables viewChangeNew	RM signals that a replica can be created by enabling Creation
Places changed	ViewNew, viewChangeNew	pointer, serviceChosen, servicesInQueue, Creation, ServiceReplicaQueue	readPointer, LoadDist, ServiceReplicaDist, Creation, replicasUp

Table 8.4: View change consequences

taken are described in Section 8.2.3, and the resulting extended places are;

$RManag = [1\ 0\ 1\ 0]$, $LoadDist = [3\ 0\ 4\ 2]$, rmUp $= 2$ and $ProcNrFailed = [0\ 0]$.

The service SAN also reacts to a processor failure. As illustrated in Table 8.3 the transition *ProcReplicaFail* is enabled by the first element of *ProcNrFailedService* equaling 1. The actions taken are described in Section 8.2.2. The resulting extended places are; $newReplUp = [2\ 1\ 3]$, $viewNew = [3\ 2\ 3]$ and

$$ServiceReplicaDist = \begin{bmatrix} 1 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 \\ 1 & 0 & 1 & 1 \end{bmatrix}$$

8.4.2 View update

A view change event can occur when a service has a view cardinality that differs from the actual number of working replicas. When a view change occurs, the RM reacts immediately. The RM checks whether or not the number of replicas of the given service has a sufficient number of replicas. If not, the RM tries to find an available processor to create a new replica on. If such a processor is found the RM tells the service to create a replica on the chosen processor. If no such processor exists, the service is added to a queue of services waiting for processors to become available.

Example continued

Continuing the example started above, a view change event can occur for service 1 as its elements in *viewNew* and *newReplUp* differ. Service 1's el-

ement in *viewNew* is set to equal its element in *newReplUp* and the RM is told about the view change by enabling *viewChangeNew*.

Thus, *viewNew* = [3 1 3] and *viewChangeNew* = [1 1].

The *reactionToViewChange* transition in RM is enabled by a 1 in the first element in *viewChangeNew* as illustrated in Table 8.4. RM first checks whether or not the service already has an entry in the *Creation* array.

In this example, the queue is empty. It can be seen that the processors eligible for hosting a replica of service 1 are processors 0 and 3. They are both currently working and neither contains a replica of service 1. As the load of processor 0 is 3 and the load of processor 3 is 2, processor 3 is chosen. The entry is added to the *Creation* queue, and *viewChangeNew* is reset to disable the instantaneous transition.

Pointer = 1, *viewChangeNew* = [0 0] and

$$Creation = \begin{bmatrix} 1 & 3 \\ 0 & 0 \\ 0 & 0 \end{bmatrix}$$

The reaction in the service submodel is not immediate. With a rate of μ_{aut} a service replica is created. The marking of the *readPointer* is used to find the first service in the queue. Here, a new replica of service 1 will be created on processor 3 and *LoadDist*, *ServiceReplicaDist*, *replicasUp* and *Creation* is updated.

readPointer = 1, *newReplUp* = [3 2 3], *LoadDist* = [3 0 4 3],

$$ServiceReplicaDist = \begin{bmatrix} 1 & 0 & 1 & 1 \\ 0 & 0 & 1 & 1 \\ 1 & 0 & 1 & 1 \end{bmatrix}$$

and

$$Creation = \begin{bmatrix} 0 & 0 \\ 0 & 0 \\ 0 & 0 \end{bmatrix}$$

.

8.4.3 Processor repair

The RM reacts immediately upon the repair of a processor. It check the queue to see whether any services are waiting for a replica to be created.

	Initiation submodel	**Affected submodel**	
	HW	*RM*	*Service*
Transition	ProcessorRepair	reactToProcFixed	replicaCreation
Enabling function	A processor is repaired	Repair of processor ProcNrFixed enabled by HW	RM signals that a replica can be created by enabling Creation
Places changed	Processors, procFailed and ProcNrFixed	LoadDist, ServiceReplicaDist, rmUp, RManag, Creation, ServicesInQueue, ServiceReplicaQueue, pointer, replicasUp, procnrFixed	readPointer, LoadDist, ServiceReplicaDist, Creation, replicasUp

Table 8.5: Processor repair consequences

If the queue is not empty, and the newly repaired processor has not yet reached its maximum capacity, the RM tells the services in the queue to create a replica on it. Consequently, the services in the queue create their replicas.

Example continued

The third system event, processor repair, is illustrated in Table 8.5. In this example processor 1 is the only failed processor and hence, the only processor that can be repaired. When processor 1 is fixed, the hardware submodel changes the markings of the places *Processors*, *procFailed* and *ProcNrFixed*. *procFailed* is decremented to reflect that one less processor is now failed.

Processors = [1 1 1 1] and *ProcNrFixed* = [1 1].

The RM submodel reacts to a repaired processor as soon as the first element of *ProcNrFixed* is set to 1. When it discovers that a processor is repaired it checks the *ServiceReplicaQueue* to see whether or not it contains any services. In this example it does not, and no changes are made other than resetting *ProcNrFixed* to disable the instantaneous transition,

ProcNrFixed = [0 0].

As no services were in the queue, no new services are added to the *Creation* queue and hence, the repair of the processor will not affect the service submodel.

8.5 Load sharing

Load sharing has not yet been implemented in the model, but an example solution will be outlined in this section. The load sharing in Jgroup/ARM should be implemented in the RM SAN.

In this solution load sharing is assumed to occur in two cases; when a processor fails and when a processor is repaired. When a processor fails, all replicas residing on that processor fail. However, when using load sharing these replicas can be created on other processors, if such eligible processors exist. The processor chosen is the available processor with the lightest load. For each service, an entry is made in the *Creation* queue if an eligible processor exist. The array elements contain the service chosen and the processor to create a replica on. If no eligible processor currently exists, the service is added to the *ServiceReplicaQueue*. The services in this queue wait for a processor to become available.

Using the functionality in the RM submodel eliminates the need for the place *ProcNrFailedService* in the HW and service submodels and the action *ProcReplicaFail* in the service submodel.

The other case where load sharing could be implemented is when a processor is repaired. As stated in [24], load sharing is advantageous if a processor is very heavily loaded and other processors are relatively free to process more jobs. When a new processor becomes available the RM will first create a replica of the services in the *ServiceReplicaQueue*. After removing the replicas from this queue, the load of the processors is inspected.

If the newly repaired processor has a significantly lighter load than the other processors, replicas are removed from their current processors and added to the *Creation* array with the new processor as target processor. However, this will only be done as long as the load of the newly fixed processor is below some threshold value and the service in question currently has more than the required number of working replicas.

8.6 Extended place definitions

The list below describes the extended places used in the dependability model.

Processors Contains an array initialized to [1 1 ... 1], where Processors[i] = 1

represents that processor i is currently up and Processors[i] = 0 represents processor i being failed.

RManag Contains an array with size set to the number of processors. RManag[i] = 1 gives that a RM replica resides on processor i.

replicasUp Contains an array with one element for each service in the system. The initialization depends on the initial number of replicas of a given service. An example initialization is [3 2 3]. Service 0 and 2 initially have 3 replicas, and service 1 has 2 replicas. At any time this array will reflect the current number of replicas that are up for each service.

viewNew Initially *replicasUp* and *viewNew* are identical. This extended place is used to keep track of the group view each service has. It is updated by the *updateView* transition.

ServiceParam This extended place gives the replication policy for each of the services, one row for each service. The first column contains the initial view cardinality, the second column the number of initial replicas and the last column the replicas required for the service to be available. An example initialization is given below.

$$\begin{bmatrix} 3 & 3 & 1 \\ 2 & 2 & 1 \\ 3 & 3 & 2 \end{bmatrix}$$

ServiceReplicaDist The *ServiceReplicaDist* extended place contains a matrix where each row represents a service and each column a processor. If a given element is 1 it means that a replica of the service resides on the processor. An example initialization is given below. It can be seen that service 0 has replicas on processors 0, 1 and 2, service 1 on processors 1 and 2 and service 2 on processors 0, 2 and 3.

$$\begin{bmatrix} 1 & 1 & 1 & 0 \\ 0 & 1 & 1 & 0 \\ 1 & 0 & 1 & 1 \end{bmatrix}$$

LoadDist Contains an array initialized according to the load on the different processors in the cluster. An example initialization is [2 2 3 2], the size is given by the number of processors in the cluster. This array represents a load of 2 service replicas on processors 0, 1 and 3, and 3

service replicas on processor 2. This extended place is used to determine which processor a service replica should be created on.

ProcNrFailed Contains an array initialized to [0 0]. 1 in the first element represents that a processor has failed, while the second element represents which processor has failed. The extended place is shared by HW and RM and is used to notify RM of a processor failure.

ProcNrFailedService Identical to *ProcNrFailed*, only difference being that it is shared between the service and RM submodels.

ProcFixed This extended place contains an array of two elements. It is initialized to [0 0]. The first element gives whether a processor is repaired or not by 1 or 0 respectively. The second element gives the number of the repaired processor. Used by HW to notify the RM submodel of a processor repair.

viewChangeNew Contains an array initialized to [0 0]. 1 in the first element represents that a view change event has taken place. The second position represents the number of the service which has changed view. The extended place is shared among Service and RM and is used to notify the RM when a group changes their view.

ServiceReplicaQueue The service queue is initially empty. As service replicas and processors fail, the queue will be filled with services waiting for a processor to create a replica on. [1 2] represents services 1 and 2 waiting for a processor to become available.

Creation This extended place contains a matrix with one row for each service and two elements in each row. The first element gives a service number and the second element a processor number. The place is used as a queue. Each row represents an element in the queue. Each service is only added to the queue once, hence the matrix only needs one row for each service. The first element gives the service to be replicated, and the second element the processor the replica can be created on.

Pointers are used to read from and write to the queue.

$$\begin{bmatrix} 1 & 2 \\ -1 & -1 \\ -1 & -1 \end{bmatrix}$$

The matrix illustrates the case where service 1 can create a replica on processor 2. The extended place is shared between RM and service,

and is used by RM to tell the given service that it can create a new replica and on which processor that replica shall be created.

Chapter 9

Validation and verificaion

Validation and verification is needed in order to prove the correctness of the dependability model developed. Validation is the process of ensuring that the model design is correct and builds the intended system. Verification is the process of ensuring that the output from a model simulation is feasible [18]. Both approaches are applied to the Jgroup/ARM dependability model as will be described in this chapter.

9.1 Validation

The development of the dependability model was done in an incremental order. The model started out representing only a simple system, which could easily be validated by checking the states generated by the state space generator and reading the trace file of a simulation. The trace files were used to debug and eventually validate that the system behaved as intended. As the simple system worked, the model was expanded.

For each piece of functionality added, simulations were run and the trajectory in the trace files validated. Additional places and variables were added to the model to validate its operation. The actual number of failures of each type were counted by the use of impulse rewards and compared to the failure rates used. The two rates corresponded in a probable way, and the model operation was assumed to be correct.

Parameter	Value	Description
Cap	4	Capacity of a processors
initRM	3	Initial number of RM replicas
λ_{Proc}	0.001	Processor failure rate
λ_{Repl}	0.002	Replica failure rate
λ_{View}	0.9	View update rate
μ_{Man}	0.2	Manual repair rate
μ_{Aut}	2	Automatic repair rate
numFailed	0	Initial number of processors failed
numProc	4	Number of processors
numServices	3	Number of services in the system
replReq	1	Service replicas required for service to work. Used in initial system.

Table 9.1: Parameter values used in this section

9.2 Verification

The verification of the dependability model was done by analyzing each of the submodel SANs separately and comparing the results to theoretical solutions of simplified models.

During verification steady state simulations were run in Möbius with an initial transient period of 500 hours, a batch size of 100 hours and minimum number of batches set to 20 and maximum number to 100000. The actual number of batches run was decided by Möbius. It will stop when the confidence interval is within 10% of the mean 95% of the time. All the result files for the simulations in this chapter are attached in attachment C.1.

The system assumed here is a system with 4 processors, 3 RM replicas and 3 services with 3 initial replicas each. It has been assumed that 1 processor and 1 replica of each service (the RM included) is required for the services and the system to be up.

The parameters used are given in Table 9.1.

9.2.1 The hardware SAN

Figure 9.1 shows the Markov chain for a system with 4 processors. Each processor can fail with rate λ and be repaired with rate μ. One repair unit is assumed.

In Möbius the solution to the model can be found either by simulation or by generating the state space and solve the model by a numerical solver. The HW submodel was expanded from the simple case in chapter 7.3 to the more complex case in Chapter 8.2.1. However, when modelling the availability of the processor cluster the simple model from Chapter 7.3 can be used. The extra functionality added to the hardware SAN is for notification purposes only. Hence, the model used in simulation and numerical solution is depicted in Figure 7.6(a).

Using the hardware submodel created in Möbius to simulate the system with 4 processors and parameters $\lambda = 0.01$ and $\mu = 0.5$ [1] yields a resulting mean availability of 1, with a variance of $5.058 \cdot 10^{-10}$.

Using the state space generator, 5 states is generated. This corresponds to the 5 states in the state diagram in figure 9.1. The state space generator generates a transition matrix, attached in Appendix B, which is displayed in (9.1).

$$\begin{bmatrix} 0 & \frac{1}{25} & 0 & 0 & 0 \\ \frac{1}{2} & 0 & \frac{3}{100} & 0 & 0 \\ 0 & \frac{1}{2} & 0 & \frac{1}{50} & 0 \\ 0 & 0 & \frac{1}{2} & 0 & \frac{1}{100} \\ 0 & 0 & 0 & 0 & \frac{1}{2} \end{bmatrix} \tag{9.1}$$

The numerical solver also gives the availability, A = 0.9999867 with confidence interval +/-$1.332395 \cdot 10^{-5}$.

The availability of the processor cluster can be found numerically by deriving the balance equations for the system in Figure 9.1. The steady state probability of being in each state is denoted P_i, where i is the number of working processors. Hence, the set of balance equations is

$$4\lambda \cdot P_4 = \mu \cdot P_3 \tag{9.2}$$

[1] Due to some issues in Möbius as described in Chapter 11.3

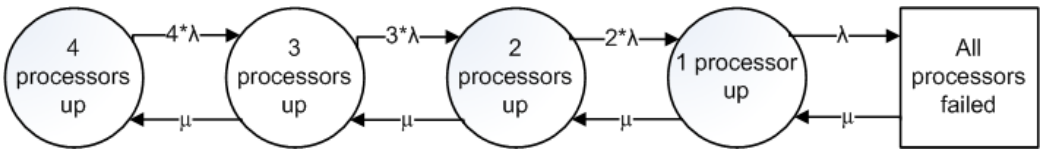


Figure 9.1: Markov model for a system with 4 processors, 1 of which is required to be up for the system to be up.

$$3\lambda \cdot P_3 + \mu \cdot P_3 = 4\lambda \cdot P_4 + \mu \cdot P_2 \tag{9.3}$$

$$2\lambda \cdot P_2 + \mu \cdot P_2 = 3\lambda \cdot P_3 + \mu \cdot P_1 \tag{9.4}$$

$$\lambda \cdot P_1 + \mu \cdot P_1 = 2\lambda \cdot P_2 + \mu \cdot P_0 \tag{9.5}$$

$$\mu \cdot P_0 = \lambda \cdot P_1 \tag{9.6}$$

$$P_4 + P_3 + P_2 + P_1 + P_0 = 1 \tag{9.7}$$

The availability of a system is the fraction of time the system is available. Hence, the availability, A, is the sum of all the up-states.

$$A = P_4 + P_3 + P_2 + P_1 \tag{9.8}$$

The unavailability of the system is the time the system spends in the down state, P_0. Denoting the unavailability U gives $U = P_0$. We also know that $A = 1 - U$. Hence, we can find the availability of the system by calculating $A = 1 - U = 1 - P_0$.

Solving the set of equations with the given values yields the results

$$A = 1 - P_0 = 1 - 3.539 \cdot 10^{-6} = 0.999996461 \tag{9.9}$$

Thus, the model for the hardware subsystem works correctly.

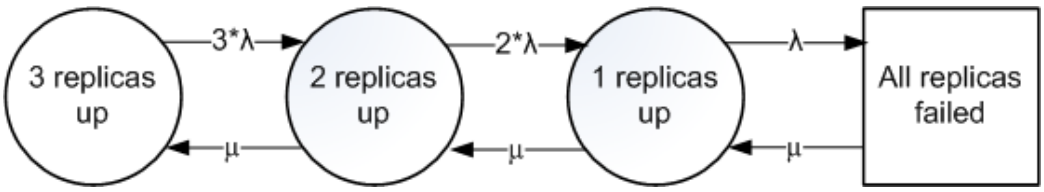


Figure 9.2: Markov model for a system with 3 service replicas, 1 of which is required to be up for the system to be up.

9.2.2 The service SAN

Figure 9.2 illustrates the Markov model of a service with three replicas. This is a simplified model of the dependability model created. Its focus is only on replica creation and failure. It only considers automatic repair, as the processor failure and repair are not included in this simple model. However, the numerical solution of this simple model will be compared to the simulation results of the Service SANs developed in Chapters 7.3 and 8.2.2.

The balance equations for Figure 9.2 is given by the balance equations 9.10 to 9.14.

$$3\lambda \cdot P_3 = \mu \cdot P_2 \tag{9.10}$$

$$2\lambda \cdot P_2 + \mu \cdot P_2 = 3\lambda \cdot P_3 + \mu \cdot P_1 \tag{9.11}$$

$$\lambda \cdot P_1 + \mu \cdot P_1 = 2\lambda \cdot P_2 + \mu \cdot P_0 \tag{9.12}$$

$$\mu \cdot P_0 = \lambda \cdot P_1 \tag{9.13}$$

$$P_3 + P_2 + P_1 + P_0 = 1 \tag{9.14}$$

The availability of a system is the fraction of time the system is available. Hence, the availablity, A, is the sum of all the up-states.

$$A = P_3 + P_2 + P_1 \tag{9.15}$$

The unavailability of the system is the time the system spends in the down state, P_0. Denoting the unavailability U gives $U = P_0$. We also know that

Model	Mean availability	Confidence interval
Numerical solution of simplified model	0.999444651	
System with one service and four processors	1.0000	+/- $2.3807178826 \cdot 10^{-9}$
Availability of one service in full system model	0.99997384464	+/- $5.1264511701 \cdot 10^{-5}$

Table 9.2: Numerical and simulation results for Service SAN.

A = 1 - U. Hence, we can find the availability of the system by calculating A = 1 - U = 1 - P_0.

Solving the set of equations with the given values yields the results

$$A = 1 - P_0 = 0.999444651 \tag{9.16}$$

Simulation of the model with one service and no RM using the parameters and batches as described in the introduction yields an availability of 1.0000 with a confidence interval of +/- $2.3807178826 \cdot 10^{-9}$.

Simulation of the final model from Chapter 8, where the reward model estimates the availability of one given service in the system, yields a availability of 0.99997384464 and a confidence interval of +/- $5.1264511701 \cdot 10^{-5}$.The results are listed in Table 9.2.

9.2.3 The RM SAN

The theoretical model of a RM is identical to that of the service SAN, as neither consider other events in the system and both have the same number of replicas and the same failure and repair rates.

The RM does not exist in the model from Chapter 7.3, so the only simulation result for the RM is the one from the final dependability model developed in Chapter 8. The mean RM availability when simulating the system is 1.0000. A reason for the higher availability of a RM than a service might be that the RM does not need a signal to create a replica, whilst the service needs the RM to assign the service replica to a processor before it can create a new replica.

Model	Mean availability	Confidence interval
Numerical solution of simplified model	0.999444651	
Availability of one service in full system model	1.0000	0.000

Table 9.3: Numerical and simulation results for RM SAN.

9.2.4 The composed system

Table 9.4 lists the simulation results for the composed system using both the simple model of Chapter 7.2 and the complete model from Chapter 8.2.

Model	Mean availability	Confidence interval
System with one service and four processors	0.99911384121	$+/- 1.2638266619 \cdot 10^{-3}$
Availability of one service in full system model	1.0000	$+/- 5.0586753828 \cdot 10^{-10}$

Table 9.4: Numerical and simulation results for the composed system.

9.3 States generated

Table 9.5 gives the number of states generated for the simplified system from Chapter 7.3. As described above, this model consists of four processors and one service. As there is only one service, the RM has not yet been introduced.

For the HW SAN, the availability results will be the same when expanding to the full system described in Chapter 8. This is because the HW submodel does not depend on the RM to function. However, the number of states generated by the state space generator is increased to 120 for the full model. This is caused by the additional places used to notify the other submodels.

SAN	States generated
Hardware	5
Service	50
Composed	230

Table 9.5: Number of states generated for system with four processors, 1 service and no RM.

In Chapter 3.1 the concept of state explosion was introduced. As described above, the state space of the hardware SAN model increase from 5 to 120 by adding the functionality to notify the Service and RM SANs on processor failure. The state space of one service with three initial replicas was 50 states. However, adding the complexity necessary to model more services in one SAN also increase the state space of the model.

Finding an upper bound on the number of possible system states can be done using combinatorics. Assuming that all processors are working, the extended place *ServiceReplicaDist* contains an array of 12 elements[2] all of which can be 1 or 0. The number of markings possible are $2^{12} = 4096$. Hence, the number of states possible considering only that place is 4096. As the Service SAN model consists of several other places as well a theoretical upper bound on the number of possible states is 301989888. Solving a state diagram of that size is bothersome. Considering that this number of states represents the state space of the Service SAN only, it is obvious that the state space of the composed model will be incomprehensible. Henceforth, state-diagrams are left out of this thesis due to the size of the state space.

[2]Assuming a system of 4 processors and 3 services.

Chapter 10

Example scenarios

This chapter presents four different modelling scenarios, simulated by the Möbius modelling tool. It describes how the different dependability parameters are obtained and finally it presents and discusses the simulation results.

10.1 Simulation environment

Möbius can be used to solve a model for several different scenarios.

The Möbius modelling tool needs some parameters to define the simulator. These are given in Table 10.1. As explained in Chapter 9.2, the simulator uses the batch means method and will run batches until the confidence interval is within 10% of the mean 95% of the time. This can be changed if a more/less accurate value is desired. The simulations used in the experiments in this thesis are all steady-state interval simulations. Hence, the duration of each batch has to be long enough to assume normal distribution.

Simulation parameter	**Value**
Initial period	500 hrs
Batch duration	10000 hrs
Minimum number of batches	20
Maximum number of batches	10000

Table 10.1: Steady state simulation parameters

Parameter	Experiment 1	Experiment 2	Experiment 3	Experiment 4
Cap	3	3	3	3
initRM	3	3	3	3
λ_{Proc}	0.0001	0.0001	0.001	0.001
λ_{Repl}	0.0002	0.0002	0.002	0.002
λ_{View}	1.0	1.0	1.0	1.0
μ_{Man}	0.5	0.1	0.5	0.1
μ_{Aut}	12.0	6.0	12.0	6.0
numFailed	0	0	0	0
numProc	4	4	4	4
numServices	3	3	3	3

Table 10.2: The 4 scenarios simulated

10.2 Scenarios

[16] experimentally obtains service dependability characteristics of a service in the Jgroup/ARM system. These values were originally used to form example scenarios in Möbius. However, difficulties were encountered when using these values for simulation[1]. Thus, a set of values, given in Table 10.2, was chosen for four different scenarios. The time unit used is hours and the different parameters were introduced in Table 9.1.

Most of the parameters common to all the experiments are pretty self-explanatory. However, λ_{View} might need an introduction. It is assumed that a view change update is sent once every hour. This rate can be optimized in future work, as sending updates too often will congest the system. However, using a too low rate might lead to stale state information being used. Hence, the view update rate should be optimized in the future.

Experiment 1 considers the case where processors and replicas fail once and twice every 10000 hours respectively. The mean repair time of a manual repair is 2 hours and the mean time automatic replica repair is 5 minutes. The repair times are changed to 10 hours and 10 minutes respectively in Experiment 2.

Experiments 3 and 4 consider a system where the failure of processors occurs once every 1000 hours and the replicas fail twice ever 1000 hours. In Experiment 3 the mean time for manual repair is assumed to be 2 hours and the mean time for replica repair 5 minutes. The repair times are changed to 10

[1]The problems are described in Chapter 11.3

hours and 10 minutes respectively in Experiment 4.

The simulation results are given in Table 10.3, and the complete result files are attached in Appendix C.2.

10.3 Dependability measures

The reward models in Möbius are used to measure the desired dependability measures. This section describes how the different measures are obtained.

10.3.1 Availability

The Jgroup/ARM framework is defined to be available (working) when at least two processors and two or more RM replicas are up. However, for clients certain services may be required for the system to be considered available. It is assumed that service 0 is a necessary service and needs at least one replica to perform its operations. Hence, the availability condition is that at least two processors, two RM replicas and one replica of service 0 is working.

The simulation type is steady-state simulation with simulation parameters as defined in Table 10.1. The availability can be found by defining a rate reward which returns 1 when the availability condition is true.

10.3.2 Mean time between system failures

Estimating the Mean Time Between system Failures (MTBF_{system}) can be done by adding functionality for capturing system failure to the dependability model and using impulse rewards to count the number of system failures. Figure 10.1 shows the additional transition added to the RM SAN. The transition is instantaneous and enabled every time the marking of the place *systemFailedBool* is 1. This marking is set by the three submodels. The system fails when too many processors fail (the marking of *procFailed* exceeds a given threshold, in the case simulated numProc - 1), when there are only one or less functioning RM replicas or when all replicas of service 0 fail (as in Section 10.3.1). This means that some additional functionality is added to the output gates of transitions *replicaFail*, *RMfail* and *processorFail*. The marking of *systemFailedBool* is set to 1 if either of the transitions causes

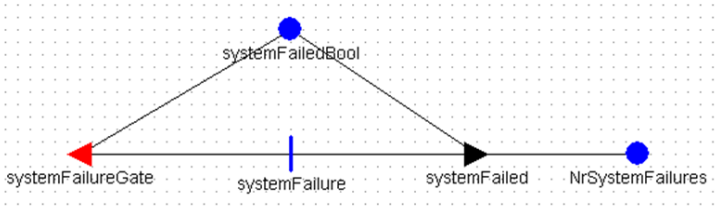


Figure 10.1: Additional functionality needed to measure system MTBF.

system failure. After the transition *systemFailure* has fired, the marking of *systemFailedBool* is reset to 0.

With this new functionality implemented an impulse reward can be defined on transition *systemFailure*. The impulse reward returns 1 each time the transition fires and uses this to find the arrival rate of system failures. The simulation type is steady-state and defined by the parameters given in Table 10.1. It is known that $\text{MTBF}_{system} = \frac{1}{\lambda_{system}}$, thus MTBF_{system} can be found.

10.3.3 System down times

From [3] we know that the mean down time (MDT) of a system can be found by the relationship U = MDT/MTBF. We also know that U = 1 - A, hence the mean down time of the system can be found by

$$MDT = U \cdot MTBF = (1 - A) \cdot MTBF$$

10.3.4 Performablity

The system performability can be measured by defining a service level and measuring the time the system performs at this level or better. However, due to problems with Möbius[2] this measure has not been implemented.

10.4 Results

All the experiments have been evaluated using simulation, because the number of states will make it infeasible, if not impossible, to solve the model numerically as described in Chapter 9.3.

[2]Chapter 11.3

Dependability measure	**Experiment 1**	**Experiment 2**	**Experiment 3**	**Experiment 4**
Availability (A)	0.999999157764	0.99999815674	0.99999805405	0.999990935
Availability confidence interval (+/-)	$1.95905665 \cdot 10^{-6}$	$4.287414964 \cdot 10^{-6}$	$4.919364903 \cdot 10^{-6}$	$2.463865820 \cdot 10^{-5}$
λ_{system}	$5.597014925 \cdot 10^{-4}$	$5.326370757 \cdot 10^{-4}$	$4.98500 \cdot 10^{-3}$	$5.654545455 \cdot 10^{-3}$
λ_{system} confidence interval (+/-)	$5.593950630 \cdot 10^{-5}$	$5.325941890 \cdot 10^{-5}$	$3.209132042 \cdot 10^{-4}$	$4.629330891 \cdot 10^{-4}$
MTBF_{system} (hrs)	1786.666347	1877.45098	200.6	176.848875
MDT_{system} (hrs)	$1.50472 \cdot 10^{-3}$	$3.46063 \cdot 10^{-3}$	$3.90358 \cdot 10^{-4}$	$1.603135 \cdot 10^{-3}$
Batches completed	670	766	200	110

Table 10.3: Simulation results for the four experiments

Table 10.3 shows the simulation results of the four example scenarios simulated. The MTBF_{system} and MDT_{system} are calculated from the values of the obtained rate and impulse rewards. The result files generated by the Möbius simulator is attached in Appendix C.2.

10.5 Discussion

As can be seen from Table 10.3, the availability of the Jgroup/ARM system is very high for all the experiments.

Comparing Experiment 1 and Experiment 2, the parameters changed are the repair rates. As the mean repair times of Experiment 1 are shorter than those of Experiment 2, it is expected that the availability of Experiment 1 is higher than that of Experiment 2. The results in Table 10.3 reflect this expected property. The same argument can be used when comparing Experiments 3 and 4.

Comparing Experiments 1 and 3, and 2 and 4, the repair times are identical for each experiment pair. However, as failures occur less frequently in Experiments 1 and 2 the availability of Experiment 1 is expected to be higher than that of Experiment 3 and the availability of Experiment 2 higher than that of Experiment 4. Table 10.3 confirms these expectations.

Thus, the availability of the Jgroup/ARM system with one necessary service is very high for all the experiments conducted. As more services are added to the MS the availability will decrease, as more replicas are required to be working for the system to be available.

The estimated MTBF_{system} is high for Experiments 1 and 2, and relatively low for Experiments 3 and 4. However, this is easily explained by the failure rates of the different experiments. Experiments 3 and 4 fail with rates 10 times those of Experiments 1 and 2.

For the pair of Experiment 1 and Experiment 2, MTBF_{system} is highest for Experiment 2, which has the highest average repair times. In the case of Experiment 3 and Experiment 4, MTBF_{system} is highest for Experiment 3, which has the lowest average repair times. Hence, it can be seen that there is no general monotonous connection between the repair times and the MTBF_{system}. The differences observed in MTBF_{system} can be caused by stochastic and systematic variations.

Chapter 11

Lessons learned

This chapter describes the difficulties encountered while modelling Jgroup/ARM, the design decisions made and the problems encountered while using Möbius.

11.1 Modelling issues and difficulties

Modelling such a complex system as the Jgroup/ARM framework introduces several issues. How the different issues are solved is given in the model description in Chapter 8, whilst the design decisions are discussed in Section 11.2. The main problems of making a dependability model of the Jgroup/ARM system are summarized in the list below.

- The need to keep track of which processors are up and which are down at any given time.
- How to map service replicas and RM replicas to processors?
- Functionality for handling processor failure in both service and RM submodels.
- The RM has to decide which service creates a replica when and where.
- Which components are necessary for a Jgroup/ARM system to be available (defined in Chapter 7).
- What happens to the load of a processor when it fails? Load sharing functionality needs to be implemented, and reallocation of service replicas must occur when processors fail and when a single node has heavier load than the other nodes.

- The processors do not have indefinite capacity, hence each service can not be replicated on all processors.
- How to model the ARM framework? Can it be assumed that the RM and the DR services are co-located?
- In Jgroup/ARM there are two types of failures; replica failure and processor failure. The model needs to capture the effect of a processor failure on all service replicas residing on it.
- What is the availability criterion of the system?
- Which values can be used for failure and repair rates?

11.2 Design decisions and considerations

When developing a dependability model for Jgroup/ARM there are several design issues to be considered. The first issue is to define the services and components that need to be included in the model. This was done in the system delivery model described in Chapter 7. The chapter concluded that the elements and services to be included in the monitored subsystem are the processor cluster (hardware), a number of services (both monitored and additional), the PGMS (view updates), DR and the RM (DR and RM functionality co-located) functionality.

Several possible model designs were considered. The first approach used was to convert the state-diagram in Figure 7.5 directly to a Petri net model. However, this model got unintelligible even for the MS with one service and no RM. Due to the complexity and dependencies in the system state, the decision made was to split the model in three submodels; service, replication manager and hardware. The model design with 3 SANs has been chosen to simplify the complexity of the Jgroup/ARM model. It makes the model more comprehensible when each system part is modeled independently, as Möbius contains functionality to create a composed model of several SANs.

The modelling started with a simple system with one replicated service and four processors. When this system worked, it was expanded to include multiple services and the replication management as well. To be able to associate RM and service replicas with distinct processors neither of the SANs uses replication. By not using replication the state space is limited as well. The design chosen uses extended places to map between RM replicas and processors and between service replicas and the processors they reside on. Parts

of the system complexity are moved from the model into code in input and output gates.

However, to be able to address distinct services the design chosen models all services in one SAN by a matrix, as described in Chapter 8. This is done to be able to distinguish between the services, both when it comes to the distribution on processes and when defining reward models, as different services may have different replication policies.

Because of dependabilities the different submodels need to be able to exchange information about their states. As described in Chapter 8 this is done by setting extended places to defined values when important events occur.

The state space size may be reduced [21] by sharing places between different SAN submodels. State lumping can be achieved i.e. when a processor fails. When the processor fails there is no longer need to keep the state of the service replicas residing on that particular computer. Their elements are set to 0 and there is no possibility for a replica failure of a replica on a failed computer. Hence, the state space is reduced. The same argument is valid for RM replicas.

11.3 Möbius difficulties

The Möbius modelling tool is a university developed tool used with no guarantees for support. Encountering problems when using Möbius, solutions can be found in the user manual [21], by emailing the development group or by searching the Möbius forum[1].

Technical problems such as how to use extended places, rate reward and arrays was easily be solved by reading postings on the forum or search the user guide. A couple of questions were also directed to the development group by email. For these simple questions it was relatively easy to find the solution. However, as the model grew more complex some problems occurred.

The first issue occurred when trying to add more than one experiment to one study. The simulator would finish the first experiment and start on the second. The results of the second experiment would converge and the simulation of the experiment stop, but the simulator would not start simulating the

[1] http://www.crhc.uiuc.edu/archive/html/mobius-users/

third experiment. Hence, each experiment had to be defined in a separate study.

Another, the main, difficulty that occurred was never really solved. When simulating or using the state space generator, for some values, an error message occurred and the simulation/state space generation was aborted. The error message reads:
"'<name of simulator/state space generator>_Windows_debug.exe has encountered a problem and needs to close. We are sorry for the inconvenience. If you were in the middle of something, the information you were working on might be lost. Please tell Microsoft about this problem."'

The cause of this error was never really discovered. It occurred sporadically, and the simulations failed in different states each time. The error did not occur during compilation, it only showed when running a simulation or using the state space generator. The error was experienced when

1. **Parameter values in the study used had been changed.** Sometimes changing the values in a study caused the error message to occur. However, this did not apply to all parameters. E.g. the first experiments, based on values from [16] had to be rejected, but the values in Table 10.2 gave valid results.

2. **The initialization of RManag had been changed.** Due to the capacity limitations set on the processors, it was desirable to change the initial distribution of the RM replicas. The initial distribution was [1 1 1 0]. The change desired was to move the RM replica on processor 2 to processor 3. However, making this change in the custom initialization led to the error.

3. **Functionality had been changed or added.** Adding to, or changing, the defined behavior of the model might provoke the error. Although it did not occur every time a line of code was added, some alterations were successful. However, in some cases even removing the newly added functionality would not remove the error. In those cases the project had to be deleted and resaved before it would again function correctly. It was also experienced that by adding and removing the lines of code several times, compiling the model between each insertion/deletion, the model eventually worked correctly.

4. **The simulation experiment had been changed.** As described above, difficulties were encountered while trying to simulate more than one experiment in one simulation. Hence, separate studies and simulations were created for each experiment.

It was also experienced that certain services and simulators would all of a sudden stop working. The file objects in Möbius could no longer be opened. Trying to do so would lock Möbius. The studies and simulators had to be deleted and created again.

However, the four experiments in Table 10.2 were simulated and the desired results eventually obtained. Lots of time and effort were put into finding the solution, but the cause of the Möbius problems still remains unknown.

Chapter 12

Conclusion and further work

This chapter concludes this thesis and suggests topics for further work.

12.1 Conclusion

The main objective of this thesis was to develop a dependability model for the Jgroup/ARM system. This task was divided into several subtasks in the introduction;

- **Perform a theoretical study of techniques for dependability modelling of distributed systems**

  Different techniques for dependability modelling of distributed systems were introduced in Chapter 3. A study of existing dependability models of distributing systems limits the modelling techniques feasible for modelling Jgroup/ARM to state-diagrams and Petri nets. The Petri nets will normally scale better for large systems, whilst the state-diagrams are easier to understand.

- **Develop a system/service delivery model**

  The system/service delivery model for Jgroup/ARM was defined in Chapter 7. In the system dependability model created, several components were required to be operational for the system to deliver the required services. First of all, more than one working processor is needed in the target environment. Second, the ARM framework is also required for the Jgroup/ARM system to deliver the required services. Hence, the DR and RM functionality is co-located and more than one

copy of the framework is required for the system to be up. The third factor included in the MS is that given services in the system may be necessary for system availability. In the model created one given service is required to have at least one working replica for the system to be considered up.

However, the MS could be changed, either by adding more necessary services, by adding/removing required service to/from the MS, or by changing the number of required processors/replicas.

- **Establish a state-diagram or a petri net dependability model for the Jgroup/ARM system**

  Due to the advantages of Petri nets, a Petri net dependability model is developed for the Jgroup/ARM system in chapter 8.

  The Möbius modelling tool developed by the PERFORM group at University of Illinois at Urbana-Champaign was used to create the dependability model. Using the tool, complex functionality can be hidden from the model by adding C++ code. The complete model consists of three SANs combined using the Möbius modelling tool. The model design is illustrated in Figures 8.1, 8.2, 8.4 and 8.6.

  As some problems occurred when using Möbius there are still some shortcomings in the dependability model developed. However, the model proposed in Chapter 8 reflects the basic functionality of the Jgroup/ARM system and suggested solutions for parts of the remaining functionality are given.

  Section 12.2 suggests topics for further development of the model developed.

- **Evaluate simple scenarios with the Möbius modelling tool**

  Some simple scenarios are evaluated by running simulations of the dependability model in Chapter 10. In the scenarios run, the failure rate of processors and service/RM replicas are varied. For all the scenarios the resulting system availability is very high.

  It is seen that the MTBF_{system} is high for the experiments with low component failure rates. As the failure rates are increased, the MTBF_{system} is decreased. The results also show that the MDT_{system} is shorter for the experiments with shorter repair times. Hence, the dependability parameters behaves as expected. Consequently it can be concluded that Jgroup/ARM is a very dependable system.

12.2 Further work

- **Expand the dependability model**

  The dependability model should be expanded to include the Jgroup/ARM functionality omitted from this model. Load sharing, partitioning and priority are examples of functionality not included in the current dependability model. An example solution as to how the load sharing could be implemented is given in Chapter 8.5. The view update rate should be optimized to prevent message flooding and use of stale information.

- **Simulate more complex scenarios**

  The scenarios simulated in this thesis are fairly simple. The system simulated provides three services and contains only four processors in the target environment. By increasing the system size more complex scenarios could be simulated. The number of services in the system, and in the monitored subsystem, could be changed to increase the load of the processors in the target environment. The number of processors in the target environment should also be varied.

References

[1] Eleftheria Athanasopoulou, Purvesh Thakker, and William Sanders. Evaluating the dependability of a leo satellite network for scientific applications. In *Second International Conference on the Quantitative Evaluation of Systems (QEST'05)*. IEEE Computer Society, 2005.

[2] J. A. Couvillion, R. Freire, R. Johnson, II Obal, W. D. I. I. A. Obal W. D., M. A. A. Qureshi M. A. Qureshi, M. A. Rai M. Rai, W. H. A. Sanders W. H. Sanders, and J. E. A. Tvedt. Performability modeling with ultrasan. *Software, IEEE*, 8(5):69–80, 1991.

[3] Peder J. Emstad, Poul E. Heegaard, and Bjarne E. Helvik. *Pålitelighet og ytelse med simulering.* tapir akademiske forlag, 2004.

[4] Q. Gan and B. E. Helvik. Dependability modelling and analysis of networks as taking routing and traffic into account. In B. E. Helvik, editor, *Next Generation Internet Design and Engineering, 2006. NGI '06. 2006 2nd Conference on*, page 8 pp., 2006.

[5] S. Hariri and H. Mutlu. Hierarchical modeling of availability in distributed systems. *Software Engineering, IEEE Transactions on*, 21(1):50–56, 1995.

[6] Bjarne E. Helvik. *Dependable Computing Systems and Communication Networks - Design and Evaluation.* 2006.

[7] Sune Jakobsson, Erik Berg, Bertrand Mathieu, Yvon Gourhant, Remi Kerboul, Marcin Solarski, and Christian Egelhaaf. White paper on middleware platforms scalability and dependability, 2001.

[8] Allen M. Johnson Jr. and Malek Miroslaw. Survey of software tools for evaluating reliability, availability, and serviceability. *ACM Comput. Surv.*, 20(4):227–269, 1988. 50062.

[9] Vinod Kumar and K. K. Aggarwal. Petri net modelling and reliability evaluation of distributed processing systems. *Reliability Engineering & System Safety*, 41(2):167–176, 1993.

[10] C. D. Lai, M. Xie, K. L. Poh, Y. S. Dai, and P. Yang. A model for availability analysis of distributed software/hardware systems. *Information and Software Technology*, 44(6):343–350, 2002.

[11] Jean-Claude Laprie. Dependability: Basic concepts and associated terminology. *Dependable Computing and Fault Tolerant Systems*, 5, 1992.

[12] N. Lopez-Benitez. Dependability modeling and analysis of distributed programs. *Transactions on Software Engineering*, 20(5):345–352, 1994.

[13] Hein Meling and Bjarne E. Helvik. Performance consequences of inconsistent client-side membership information in the open group model. In *Performance, Computing, and Communications, 2004 IEEE International Conference on*, 2004.

[14] Hein Meling and Alberto Montresor. Jgroup tutorial and programmer's manual, 2002.

[15] Hein Meling, Alberto Montresor, Bjarne E. Helvik, and Ozalp Babaoglu. Jgroup/ARM: A distributed object group platform with autonomous replication management. *Software: Practice and Experience*, page 39, 2007.

[16] Alberto Montresor, Bjarne E. Helvik, and Hein Meling. An approach to experimentally obtain service dependability characteristics of the Jgroup/ARM system. In Mario Dal Cin Pataricza, Mohamed Kaaniche, and Andras, editors, *European Dependable Computing Conference (EDCC-5)*, 2005.

[17] T. Murata. Petri nets: Properties, analysis and applications. *Proceedings of the IEEE*, 77(4):541–580, 1989.

[18] Balci Osman. Principles and techniques of simulation validation, verification, and testing, 1995. 224456 147-154.

[19] Sheldon M. Ross. *Introduction to probability models*. 2003.

[20] W. H. Sanders and II Obal, W. D. Dependability evaluation using ultrasan. In II Obal, W. D., editor, *Fault-Tolerant Computing, 1993. FTCS-23. Digest of Papers., The Twenty-Third International Symposium on*, pages 674–679, 1993.

[21] William H. Sanders. Möbius manual, 1994.

[22] William H. Sanders and Luai M. Malhis. Dependability evaluation using composed san-based reward models. *Journal of Parallel and Distributed Computing*, 15(3):238–254, 1992.

[23] Michael Stamatelatos, William Vesely, Joanne Dugan, Joseph Fragola, Josepd Minarick, and Jan Railsback. Fault tree handbook with aerospace applications. Technical report, 2002.

[24] Satisk K. Tripathi, David Finkel, and Erol Gelenbe. Load sharing in distributed systems with failures. *Acta Informatica*, 25:677–689, 1988.

[25] Antti Valmari. The state exposion problem. *Lecture notes in computer science*, 1996. Spredte bind utgjør samtidig bind i underserien Lecture notes in artificial intelligence og Lecture notes in bioinformatics.

[26] F. Wagner and P. Wolstenholme. Misunder standings about state machines. *Computing & Control Engineering Journal*, 15(4):40–45, 2004.

[27] Chen Yinong and He Zhongshi. Dependability modelling of homogeneous and heterogeneous distributed systems. In He Zhongshi, editor, *Autonomous Decentralized Systems, 2001. Proceedings. 5th International Symposium on*, pages 176–183, 2001.

Appendix A

Source code

The source code for the SAN models created in Möbius is attached in this Appendix. An archived version of the dependability model, containing executable models, exist and is attached electronically in DAIM. In order to view the model Möbius is needed. To open the project, it first has to be unarchived and resaved by Möbius commands.

A.1 The HW SAN

```
// This C++ file was created by SanEditor

#include "Atomic/HW/HWSAN.h"

#include <stdlib.h>
#include <iostream.h>

#include <math.h>


/*****************************************************************
                          HWSAN Constructor
*****************************************************************/


HWSAN::HWSAN(){


  Activity* InitialActionList[2]={
    &ProcessorFail, //0
    &ProcessorRepair  // 1
  };
```

```
  BaseGroupClass* InitialGroupList[2]={
    (BaseGroupClass*) &(ProcessorFail),
    (BaseGroupClass*) &(ProcessorRepair)
  };

  procFailed = new Place("procFailed" ,0);
  systemFailure = new Place("systemFailure" ,0);
  int temp_Processorsprocvalue = 1;
  Processors = new proc("Processors",temp_Processorsprocvalue);
  profFailedAr_state temp_ProcNrFailedprofFailedAr = {0,0};
  ProcNrFailed = new profFailedAr("ProcNrFailed",temp_ProcNrFailedprofFailedAr);
  toService_state temp_ProcFixedtoService = {0,0};
  ProcFixed = new toService("ProcFixed",temp_ProcFixedtoService);
  profFailedAr_state temp_ProcNrFailedServiceprofFailedAr = {0,0};
  ProcNrFailedService = new profFailedAr("ProcNrFailedService",
        temp_ProcNrFailedServiceprofFailedAr);
  BaseStateVariableClass* InitialPlaces[6]={
    procFailed,  // 0
    systemFailure,  // 1
    Processors,  // 2
    ProcNrFailed,  // 3
    ProcFixed,  // 4
    ProcNrFailedService   // 5
  };
  BaseStateVariableClass* InitialROPlaces[0]={
  };
  initializeSANModelNow("HW", 6, InitialPlaces,
                        0, InitialROPlaces,
                        2, InitialActionList, 2, InitialGroupList);


  assignPlacesToActivitiesInst();
  assignPlacesToActivitiesTimed();

  int AffectArcs[8][2]={
    {0,0}, {1,0}, {2,0}, {3,0}, {5,0}, {0,1}, {2,1}, {4,1}
  };
  for(int n=0;n<8;n++) {
    AddAffectArc(InitialPlaces[AffectArcs[n][0]],
                 InitialActionList[AffectArcs[n][1]]);
  }
  int EnableArcs[2][2]={
    {0,0}, {0,1}
  };
  for(int n=0;n<2;n++) {
    AddEnableArc(InitialPlaces[EnableArcs[n][0]],
                 InitialActionList[EnableArcs[n][1]]);
  }

  for(int n=0;n<2;n++) {
    InitialActionList[n]->LinkVariables();
  }
  CustomInitialization();

}

void HWSAN::CustomInitialization() {

}
HWSAN::~HWSAN(){
  for (int i = 0; i < NumStateVariables-NumReadOnlyPlaces; i++)
    delete LocalStateVariables[i];
```

```
};

void HWSAN::assignPlacesToActivitiesInst(){
}
void HWSAN::assignPlacesToActivitiesTimed(){
  ProcessorFail.procFailed = (Place*) LocalStateVariables[0];
  ProcessorFail.systemFailure = (Place*) LocalStateVariables[1];
  ProcessorFail.Processors = (proc*) LocalStateVariables[2];
  ProcessorFail.ProcNrFailed = (profFailedAr*) LocalStateVariables[3];
  ProcessorFail.ProcNrFailedService = (profFailedAr*) LocalStateVariables[5];
  ProcessorRepair.procFailed = (Place*) LocalStateVariables[0];
  ProcessorRepair.Processors = (proc*) LocalStateVariables[2];
  ProcessorRepair.ProcFixed = (toService*) LocalStateVariables[4];
}
/*****************************************************************/
/*                  Activity Method Definitions                  */
/*****************************************************************/

/*======================ProcessorFailActivity=======================*/

HWSAN::ProcessorFailActivity::ProcessorFailActivity(){
  TheDistributionParameters = new double[1];
  ActivityInitialize("ProcessorFail",0,Exponential, RaceEnabled, 5,1, false);
}

HWSAN::ProcessorFailActivity::~ProcessorFailActivity(){
  delete[] TheDistributionParameters;
}

void HWSAN::ProcessorFailActivity::LinkVariables(){
  procFailed->Register(&procFailed_Mobius_Mark);
  systemFailure->Register(&systemFailure_Mobius_Mark);



}

bool HWSAN::ProcessorFailActivity::Enabled(){
  OldEnabled=NewEnabled;
  NewEnabled=((procFailed->Mark()<4));
  return NewEnabled;
}

double HWSAN::ProcessorFailActivity::Rate(){
  return (numProc-(procFailed->Mark())+1)*lambdaProc;
  return 1.0;  // default rate if none is specified
}

double HWSAN::ProcessorFailActivity::Weight(){
  return 1;
}

bool HWSAN::ProcessorFailActivity::ReactivationPredicate(){
  return false;
}

bool HWSAN::ProcessorFailActivity::ReactivationFunction(){
  return false;
}

double HWSAN::ProcessorFailActivity::SampleDistribution(){
  return TheDistribution->Exponential((numProc-(procFailed->Mark())+1)*lambdaProc);
```

```
}

double* HWSAN::ProcessorFailActivity::ReturnDistributionParameters(){
  TheDistributionParameters[0] = Rate();
  return TheDistributionParameters;
}

int HWSAN::ProcessorFailActivity::Rank(){
  return 1;
}

BaseActionClass* HWSAN::ProcessorFailActivity::Fire(){
  ;
  #include <cstdlib>

//Create an array of all operative processors.

int a;
int b;
int i;
int n=0;
int o = numProc-procFailed->Mark();
int Up [o];

for(i=0; i<numProc; i++){

        if(Processors->Index(i)->Mark()==1){
                Up[n]=i;
                n++;
        }
}
//Pick one random processor that failes.
a = (rand() % o);
b = Up[a];
Processors -> Index(b) -> Mark() = 0;
//Notify RM and Service submodels of Processor failure
ProcNrFailed ->Index(0)->Mark() = 1;
ProcNrFailed ->Index(1)->Mark() = b;
ProcNrFailedService ->Index(0)->Mark() = 1;
ProcNrFailedService ->Index(1)->Mark() = b;
procFailed->Mark() ++;

//Capture system failures
if(procFailed->Mark() >= (numProc-1)){
        systemFailure->Mark() =1;
}




  return this;
}

/*======================ProcessorRepairActivity=======================*/

HWSAN::ProcessorRepairActivity::ProcessorRepairActivity(){
  TheDistributionParameters = new double[1];
  ActivityInitialize("ProcessorRepair",1,Exponential, RaceEnabled, 3,1, false);
}

HWSAN::ProcessorRepairActivity::~ProcessorRepairActivity(){
  delete[] TheDistributionParameters;
```

```
}

void HWSAN::ProcessorRepairActivity::LinkVariables(){
  procFailed->Register(&procFailed_Mobius_Mark);


}

bool HWSAN::ProcessorRepairActivity::Enabled(){
  OldEnabled=NewEnabled;
  NewEnabled=((procFailed->Mark()>0));
  return NewEnabled;
}

double HWSAN::ProcessorRepairActivity::Rate(){
  return muMan;
  return 1.0;  // default rate if none is specified
}

double HWSAN::ProcessorRepairActivity::Weight(){
  return 1;
}

bool HWSAN::ProcessorRepairActivity::ReactivationPredicate(){
  return false;
}

bool HWSAN::ProcessorRepairActivity::ReactivationFunction(){
  return false;
}

double HWSAN::ProcessorRepairActivity::SampleDistribution(){
  return TheDistribution->Exponential(muMan);
}

double* HWSAN::ProcessorRepairActivity::ReturnDistributionParameters(){
  TheDistributionParameters[0] = Rate();
  return TheDistributionParameters;
}

int HWSAN::ProcessorRepairActivity::Rank(){
  return 1;
}

BaseActionClass* HWSAN::ProcessorRepairActivity::Fire(){
  ;

#include <cstdlib>
int a;
int b;
int i;
int n=0;
int fail = procFailed->Mark();
int Down [fail];
//Create an array of failed processors
for(i=0; i<numProc; i++){

        if(Processors->Index(i)->Mark()==0){
                Down[n]=i;
                n++;
        }
}
```

```
//Pick random failed processor to be repaired
a = (rand() % procFailed->Mark());
b = Down[a];
Processors -> Index(b) -> Mark() =1;
//Notify RM
ProcFixed ->Index(0)->Mark() = 1;
ProcFixed ->Index(1)->Mark() = b;

procFailed->Mark() --;
  return this;
}
```

A.2 The Service SAN

```
// This C++ file was created by SanEditor

#include "Atomic/Service/ServiceSAN.h"

#include <stdlib.h>
#include <iostream.h>

#include <math.h>


/****************************************************************
                        ServiceSAN Constructor
****************************************************************/


ServiceSAN::ServiceSAN(){


  Activity* InitialActionList[6]={
    &ProcReplicaFail, //0
    &replFail, //1
    &updateView0, //2
    &updateView1, //3
    &updateView2, //4
    &replicaCreation  // 5
  };

  BaseGroupClass* InitialGroupList[6]={
    (BaseGroupClass*) &(replFail),
    (BaseGroupClass*) &(updateView0),
    (BaseGroupClass*) &(updateView1),
    (BaseGroupClass*) &(updateView2),
    (BaseGroupClass*) &(replicaCreation),
    (BaseGroupClass*) &(ProcReplicaFail)
  };

  procFailed = new Place("procFailed" ,0);
  serviceChosen = new Place("serviceChosen" ,0);
  viewServiceChosen = new Place("viewServiceChosen" ,0);
```

```
repairChosen = new Place("repairChosen" ,0);
readPointer = new Place("readPointer" ,0);
systemFailure = new Place("systemFailure" ,0);
int temp_Processorsprocvalue = 1;
Processors = new proc("Processors",temp_Processorsprocvalue);
hvor_state temp_ServiceReplicaDisthvemhvahvorvalue = 0;
ServiceReplicaDist = new hvemhvahvor("ServiceReplicaDist",
      temp_ServiceReplicaDisthvemhvahvorvalue);
int temp_viewNewviewArrayvalue = 0;
viewNew = new viewArray("viewNew",temp_viewNewviewArrayvalue);
int temp_LoadDistprocLoadvalue = 0;
LoadDist = new procLoad("LoadDist",temp_LoadDistprocLoadvalue);
profFailedAr_state temp_ProcNrFailedServiceprofFailedAr = {0,0};
ProcNrFailedService = new profFailedAr("ProcNrFailedService",
      temp_ProcNrFailedServiceprofFailedAr);
toService_state temp_viewChangeNewtoService = {0,0};
viewChangeNew = new toService("viewChangeNew",temp_viewChangeNewtoService);
int temp_replicasUpreplUpvalue = 0;
replicasUp = new replUp("replicasUp",temp_replicasUpreplUpvalue);
NY_state temp_CreationKOvalue = {-1,0};
Creation = new KO("Creation",temp_CreationKOvalue);
hvor_state temp_ServiceParamhvemhvahvorvalue = 0;
ServiceParam = new hvemhvahvor("ServiceParam",temp_ServiceParamhvemhvahvorvalue);
BaseStateVariableClass* InitialPlaces[15]={
  procFailed,  // 0
  serviceChosen,  // 1
  viewServiceChosen,  // 2
  repairChosen,  // 3
  readPointer,  // 4
  systemFailure,  // 5
  Processors,  // 6
  ServiceReplicaDist,  // 7
  viewNew,  // 8
  LoadDist,  // 9
  ProcNrFailedService,  // 10
  viewChangeNew,  // 11
  replicasUp,  // 12
  Creation,  // 13
  ServiceParam   // 14
};
BaseStateVariableClass* InitialROPlaces[0]={
};
initializeSANModelNow("Service", 15, InitialPlaces,
                      0, InitialROPlaces,
                      6, InitialActionList, 6, InitialGroupList);


assignPlacesToActivitiesInst();
assignPlacesToActivitiesTimed();

int AffectArcs[26][2]={
  {10,0}, {7,0}, {9,0}, {12,0}, {0,0}, {1,1}, {12,1}, {5,1},
  {7,1}, {9,1}, {14,1}, {8,2}, {12,2}, {11,2}, {8,3}, {12,3},
  {11,3}, {8,4}, {12,4}, {11,4}, {3,5}, {4,5}, {13,5}, {12,5},
  {9,5}, {7,5}
};
for(int n=0;n<26;n++) {
  AddAffectArc(InitialPlaces[AffectArcs[n][0]],
               InitialActionList[AffectArcs[n][1]]);
}
int EnableArcs[11][2]={
  {7,0}, {10,0}, {12,1}, {8,2}, {12,2}, {8,3}, {12,3}, {8,4},
```

```
    {12,4}, {4,5}, {13,5}
  };
  for(int n=0;n<11;n++) {
    AddEnableArc(InitialPlaces[EnableArcs[n][0]],
                 InitialActionList[EnableArcs[n][1]]);
  }

  for(int n=0;n<6;n++) {
    InitialActionList[n]->LinkVariables();
  }
  CustomInitialization();

}

void ServiceSAN::CustomInitialization() {


}
ServiceSAN::~ServiceSAN(){
  for (int i = 0; i < NumStateVariables-NumReadOnlyPlaces; i++)
    delete LocalStateVariables[i];
};

void ServiceSAN::assignPlacesToActivitiesInst(){
  ProcReplicaFail.ServiceReplicaDist = (hvemhvahvor*) LocalStateVariables[7];
  ProcReplicaFail.ProcNrFailedService = (profFailedAr*) LocalStateVariables[10];
  ProcReplicaFail.LoadDist = (procLoad*) LocalStateVariables[9];
  ProcReplicaFail.replicasUp = (replUp*) LocalStateVariables[12];
  ProcReplicaFail.procFailed = (Place*) LocalStateVariables[0];
}
void ServiceSAN::assignPlacesToActivitiesTimed(){
  replFail.replicasUp = (replUp*) LocalStateVariables[12];
  replFail.serviceChosen = (Place*) LocalStateVariables[1];
  replFail.systemFailure = (Place*) LocalStateVariables[5];
  replFail.ServiceReplicaDist = (hvemhvahvor*) LocalStateVariables[7];
  replFail.LoadDist = (procLoad*) LocalStateVariables[9];
  replFail.ServiceParam = (hvemhvahvor*) LocalStateVariables[14];
  updateView0.viewNew = (viewArray*) LocalStateVariables[8];
  updateView0.replicasUp = (replUp*) LocalStateVariables[12];
  updateView0.viewChangeNew = (toService*) LocalStateVariables[11];
  updateView1.viewNew = (viewArray*) LocalStateVariables[8];
  updateView1.replicasUp = (replUp*) LocalStateVariables[12];
  updateView1.viewChangeNew = (toService*) LocalStateVariables[11];
  updateView2.viewNew = (viewArray*) LocalStateVariables[8];
  updateView2.replicasUp = (replUp*) LocalStateVariables[12];
  updateView2.viewChangeNew = (toService*) LocalStateVariables[11];
  replicaCreation.readPointer = (Place*) LocalStateVariables[4];
  replicaCreation.Creation = (KO*) LocalStateVariables[13];
  replicaCreation.repairChosen = (Place*) LocalStateVariables[3];
  replicaCreation.replicasUp = (replUp*) LocalStateVariables[12];
  replicaCreation.LoadDist = (procLoad*) LocalStateVariables[9];
  replicaCreation.ServiceReplicaDist = (hvemhvahvor*) LocalStateVariables[7];
}
/*****************************************************************/
/*                  Activity Method Definitions                  */
/*****************************************************************/

/*======================ProcReplicaFailActivity=======================*/


ServiceSAN::ProcReplicaFailActivity::ProcReplicaFailActivity(){
  ActivityInitialize("ProcReplicaFail",5,Instantaneous , RaceEnabled, 5,2, false);
```

```
}

void ServiceSAN::ProcReplicaFailActivity::LinkVariables(){




  procFailed->Register(&procFailed_Mobius_Mark);
}

bool ServiceSAN::ProcReplicaFailActivity::Enabled(){
  OldEnabled=NewEnabled;
  NewEnabled=(((ServiceReplicaDist->Index(0)->Index(
        ProcNrFailedService->Index(1)->Mark())->Mark()== 1 ||
        ServiceReplicaDist->Index(1)->Index(
        ProcNrFailedService->Index(1)->Mark())->Mark()== 1 ||
        ServiceReplicaDist->Index(2)->Index(
        ProcNrFailedService->Index(1)->Mark())->Mark()== 1) &&
        ProcNrFailedService->Index(0)->Mark()==1));
  return NewEnabled;
}

double ServiceSAN::ProcReplicaFailActivity::Weight(){
  return 1;
}

bool ServiceSAN::ProcReplicaFailActivity::ReactivationPredicate(){
  return false;
}

bool ServiceSAN::ProcReplicaFailActivity::ReactivationFunction(){
  return false;
}

double ServiceSAN::ProcReplicaFailActivity::SampleDistribution(){
  return 0;
}

double* ServiceSAN::ProcReplicaFailActivity::ReturnDistributionParameters(){
    return NULL;
}

int ServiceSAN::ProcReplicaFailActivity::Rank(){
  return 1;
}

BaseActionClass* ServiceSAN::ProcReplicaFailActivity::Fire(){
  ;
  //React to processor failure
int procFailed = ProcNrFailedService->Index(1)->Mark();
int i;
//All services with replicas on failed processor must looses the replica
for(i=0; i<3; i++){
        if(ServiceReplicaDist->Index(i)->Index(procFailed)->Mark()==1){
                ServiceReplicaDist->Index(i)->Index(procFailed)->Mark()=0;
                LoadDist->Index(procFailed)->Mark() = 0;
                replicasUp->Index(i)->Mark()--;
        }
}

ProcNrFailedService->Index(0)->Mark() = 0;
ProcNrFailedService->Index(1)->Mark() = 0;
```

```
  return this;
}

/*======================replFailActivity=======================*/

ServiceSAN::replFailActivity::replFailActivity(){
  TheDistributionParameters = new double[1];
  ActivityInitialize("replFail",0,Exponential, RaceEnabled, 6,1, false);
}

ServiceSAN::replFailActivity::~replFailActivity(){
  delete[] TheDistributionParameters;
}

void ServiceSAN::replFailActivity::LinkVariables(){

  serviceChosen->Register(&serviceChosen_Mobius_Mark);
  systemFailure->Register(&systemFailure_Mobius_Mark);



}

bool ServiceSAN::replFailActivity::Enabled(){
  OldEnabled=NewEnabled;
  NewEnabled=((replicasUp->Index(0)->Mark() > 0 || replicasUp->Index(1)->Mark() > 0 ||
        replicasUp->Index(2)->Mark() > 0));
  return NewEnabled;
}

double ServiceSAN::replFailActivity::Rate(){
  return (replicasUp->Index(serviceChosen->Mark())->Mark()+1)*lambdaRepl;
  return 1.0;  // default rate if none is specified
}

double ServiceSAN::replFailActivity::Weight(){
  return 1;
}

bool ServiceSAN::replFailActivity::ReactivationPredicate(){
  return false;
}

bool ServiceSAN::replFailActivity::ReactivationFunction(){
  return false;
}

double ServiceSAN::replFailActivity::SampleDistribution(){
  return TheDistribution->Exponential((replicasUp->Index(
        serviceChosen->Mark())->Mark()+1)*lambdaRepl);
}

double* ServiceSAN::replFailActivity::ReturnDistributionParameters(){
  TheDistributionParameters[0] = Rate();
  return TheDistributionParameters;
}

int ServiceSAN::replFailActivity::Rank(){
  return 1;
}
```

```
BaseActionClass* ServiceSAN::replFailActivity::Fire(){
  int up[3];
int counter;
int n=0;
//Create array of services with working replicas
for(counter = 0; counter<3; counter ++){
        if(replicasUp->Index(counter)->Mark() > 0){
                up[n] = counter;
                n++;
        }
}

//Pick service to experience replica failure
int temp= (rand() % n);
serviceChosen->Mark() = up[temp];
  #include <cstdlib>

//Create an array of all operative replicas.

int a;
int b;
int i;
int o = replicasUp->Index(serviceChosen->Mark())->Mark();
n=0;
int Up [o];

for(i=0; i<numProc; i++){

        if(ServiceReplicaDist->Index(serviceChosen->Mark())->Index(i)->Mark()==1){
                Up[n]=i;
                n++;
        }
}
//Pick one random replica that failes.
a = (rand() % n);
b = Up[a];
ServiceReplicaDist->Index(serviceChosen->Mark()) -> Index(b) -> Mark() = 0;
LoadDist->Index(b)->Mark() --;




replicasUp->Index(serviceChosen->Mark())->Mark() --;
//Capture system failure
if(replicasUp->Index(0)->Mark()> ServiceParam->Index(0)->Index(2)->Mark()){
        systemFailure->Mark() = 1;
}


  return this;
}

/*======================updateView0Activity=======================*/

ServiceSAN::updateView0Activity::updateView0Activity(){
  TheDistributionParameters = new double[1];
  ActivityInitialize("updateView0",1,Exponential, RaceEnabled, 3,2, false);
}

ServiceSAN::updateView0Activity::~updateView0Activity(){
  delete[] TheDistributionParameters;
}
```

```
void ServiceSAN::updateView0Activity::LinkVariables(){



}

bool ServiceSAN::updateView0Activity::Enabled(){
  OldEnabled=NewEnabled;
  NewEnabled=((viewNew->Index(0)->Mark() != replicasUp->Index(0)->Mark() ));
  return NewEnabled;
}

double ServiceSAN::updateView0Activity::Rate(){
  return (replicasUp->Index(0)->Mark()+1)*lambdaView;
  return 1.0;  // default rate if none is specified
}

double ServiceSAN::updateView0Activity::Weight(){
  return 1;
}

bool ServiceSAN::updateView0Activity::ReactivationPredicate(){
  return false;
}

bool ServiceSAN::updateView0Activity::ReactivationFunction(){
  return false;
}

double ServiceSAN::updateView0Activity::SampleDistribution(){
  return TheDistribution->Exponential((replicasUp->Index(0)->Mark()+1)*lambdaView);
}

double* ServiceSAN::updateView0Activity::ReturnDistributionParameters(){
  TheDistributionParameters[0] = Rate();
  return TheDistributionParameters;
}

int ServiceSAN::updateView0Activity::Rank(){
  return 1;
}

BaseActionClass* ServiceSAN::updateView0Activity::Fire(){
  ;



viewNew->Index(0)->Mark() = replicasUp->Index(0)->Mark();
//Notify RM of view change
viewChangeNew->Index(0)->Mark() = 1;
viewChangeNew->Index(1)->Mark() = 0;
  return this;
}

/*======================updateView1Activity=======================*/

ServiceSAN::updateView1Activity::updateView1Activity(){
  TheDistributionParameters = new double[1];
  ActivityInitialize("updateView1",2,Exponential, RaceEnabled, 3,2, false);
}
```

```
ServiceSAN::updateView1Activity::~updateView1Activity(){
  delete[] TheDistributionParameters;
}

void ServiceSAN::updateView1Activity::LinkVariables(){



}

bool ServiceSAN::updateView1Activity::Enabled(){
  OldEnabled=NewEnabled;
  NewEnabled=((viewNew->Index(1)->Mark() != replicasUp->Index(1)->Mark() ));
  return NewEnabled;
}

double ServiceSAN::updateView1Activity::Rate(){
  return (replicasUp->Index(1)->Mark()+1)*lambdaView;
  return 1.0;  // default rate if none is specified
}

double ServiceSAN::updateView1Activity::Weight(){
  return 1;
}

bool ServiceSAN::updateView1Activity::ReactivationPredicate(){
  return false;
}

bool ServiceSAN::updateView1Activity::ReactivationFunction(){
  return false;
}

double ServiceSAN::updateView1Activity::SampleDistribution(){
  return TheDistribution->Exponential((replicasUp->Index(1)->Mark()+1)*lambdaView);
}

double* ServiceSAN::updateView1Activity::ReturnDistributionParameters(){
  TheDistributionParameters[0] = Rate();
  return TheDistributionParameters;
}

int ServiceSAN::updateView1Activity::Rank(){
  return 1;
}

BaseActionClass* ServiceSAN::updateView1Activity::Fire(){
  ;

viewNew->Index(1)->Mark() = replicasUp->Index(1)->Mark();
//Notify RM of view change
viewChangeNew->Index(0)->Mark() = 1;
viewChangeNew->Index(1)->Mark() = 1;
  return this;
}

/*======================updateView2Activity=======================*/

ServiceSAN::updateView2Activity::updateView2Activity(){
  TheDistributionParameters = new double[1];
  ActivityInitialize("updateView2",3,Exponential, RaceEnabled, 3,2, false);
}
```

```
ServiceSAN::updateView2Activity::~updateView2Activity(){
  delete[] TheDistributionParameters;
}

void ServiceSAN::updateView2Activity::LinkVariables(){



}

bool ServiceSAN::updateView2Activity::Enabled(){
  OldEnabled=NewEnabled;
  NewEnabled=((viewNew->Index(2)->Mark() != replicasUp->Index(2)->Mark() ));
  return NewEnabled;
}

double ServiceSAN::updateView2Activity::Rate(){
  return (replicasUp->Index(2)->Mark()+1)*lambdaView;
  return 1.0;  // default rate if none is specified
}

double ServiceSAN::updateView2Activity::Weight(){
  return 1;
}

bool ServiceSAN::updateView2Activity::ReactivationPredicate(){
  return false;
}

bool ServiceSAN::updateView2Activity::ReactivationFunction(){
  return false;
}

double ServiceSAN::updateView2Activity::SampleDistribution(){
  return TheDistribution->Exponential((replicasUp->Index(2)->Mark()+1)*lambdaView);
}

double* ServiceSAN::updateView2Activity::ReturnDistributionParameters(){
  TheDistributionParameters[0] = Rate();
  return TheDistributionParameters;
}

int ServiceSAN::updateView2Activity::Rank(){
  return 1;
}

BaseActionClass* ServiceSAN::updateView2Activity::Fire(){
  ;

viewNew->Index(2)->Mark() = replicasUp->Index(2)->Mark();
//Notify RM of view change
viewChangeNew->Index(0)->Mark() = 1;
viewChangeNew->Index(1)->Mark() = 2;
  return this;
}

/*======================replicaCreationActivity=======================*/

ServiceSAN::replicaCreationActivity::replicaCreationActivity(){
  TheDistributionParameters = new double[1];
  ActivityInitialize("replicaCreation",4,Exponential, RaceEnabled, 6,2, false);
```

```
}

ServiceSAN::replicaCreationActivity::~replicaCreationActivity(){
  delete[] TheDistributionParameters;
}

void ServiceSAN::replicaCreationActivity::LinkVariables(){
  readPointer->Register(&readPointer_Mobius_Mark);

  repairChosen->Register(&repairChosen_Mobius_Mark);



}

bool ServiceSAN::replicaCreationActivity::Enabled(){
  OldEnabled=NewEnabled;
  NewEnabled=((Creation->Index(readPointer->Mark())->Index(0)->Mark()>-1 &&
        Creation->Index(readPointer->Mark())->Index(1)->Mark()>-1));
  return NewEnabled;
}

double ServiceSAN::replicaCreationActivity::Rate(){
  return muAut;
  return 1.0;  // default rate if none is specified
}

double ServiceSAN::replicaCreationActivity::Weight(){
  return 1;
}

bool ServiceSAN::replicaCreationActivity::ReactivationPredicate(){
  return false;
}

bool ServiceSAN::replicaCreationActivity::ReactivationFunction(){
  return false;
}

double ServiceSAN::replicaCreationActivity::SampleDistribution(){
  return TheDistribution->Exponential(muAut);
}

double* ServiceSAN::replicaCreationActivity::ReturnDistributionParameters(){
  TheDistributionParameters[0] = Rate();
  return TheDistributionParameters;
}

int ServiceSAN::replicaCreationActivity::Rank(){
  return 1;
}

BaseActionClass* ServiceSAN::replicaCreationActivity::Fire(){
  ;


repairChosen->Mark() = Creation->Index(readPointer->Mark())->Index(0)->Mark();
replicasUp->Index(repairChosen->Mark())->Mark() ++;
int choise = Creation->Index(readPointer->Mark())->Index(1)->Mark();
LoadDist->Index(choise)->Mark()++;

//Create a replica of the chosen service on chosen processor.
```

```
ServiceReplicaDist->Index(repairChosen->Mark())->Index(choise)->Mark()=1;
Creation->Index(readPointer->Mark())->Index(0)->Mark() = -1;
Creation->Index(readPointer->Mark())->Index(1)->Mark() = -1;
readPointer->Mark() = ((readPointer->Mark()+1)%(numServices));




  return this;
}
```

A.3 The RM SAN

```
// This C++ file was created by SanEditor

#include "Atomic/RM/RMSAN.h"

#include <stdlib.h>
#include <iostream.h>

#include <math.h>


/*****************************************************************
                          RMSAN Constructor
*****************************************************************/


RMSAN::RMSAN(){


  Activity* InitialActionList[6]={
    &reactToProcFailed, //0
    &reactToProcFixed, //1
    &reactioToViewChange, //2
    &Instantaneous_Activity1, //3
    &RMfail, //4
    &RMrepair  // 5
  };

  BaseGroupClass* InitialGroupList[6]={
    (BaseGroupClass*) &(RMfail),
    (BaseGroupClass*) &(RMrepair),
    (BaseGroupClass*) &(reactToProcFailed),
    (BaseGroupClass*) &(reactToProcFixed),
    (BaseGroupClass*) &(reactioToViewChange),
    (BaseGroupClass*) &(Instantaneous_Activity1)
  };

  rmUp = new Place("rmUp" ,3);
  procFailed = new Place("procFailed" ,0);
```

```
viewServiceChosen = new Place("viewServiceChosen" ,0);
ServicesInQueue = new Place("ServicesInQueue" ,0);
pointer = new Place("pointer" ,0);
queuePointer = new Place("queuePointer" ,0);
readQueuePointer = new Place("readQueuePointer" ,0);
systemFailed = new Place("systemFailed" ,0);
systemFailure = new Place("systemFailure" ,0);
int temp_RManagRMvalue = 0;
RManag = new RM("RManag",temp_RManagRMvalue);
hvor_state temp_ServiceReplicaDisthvemhvahvorvalue = 0;
ServiceReplicaDist = new hvemhvahvor("ServiceReplicaDist",
      temp_ServiceReplicaDisthvemhvahvorvalue);
card_state temp_ServiceParamservvalue = {0,0,0};
ServiceParam = new serv("ServiceParam",temp_ServiceParamservvalue);
int temp_LoadDistprocLoadvalue = 0;
LoadDist = new procLoad("LoadDist",temp_LoadDistprocLoadvalue);
int temp_Processorsprocvalue = 1;
Processors = new proc("Processors",temp_Processorsprocvalue);
int temp_viewNewviewArrayvalue = 3;
viewNew = new viewArray("viewNew",temp_viewNewviewArrayvalue);
toService_state temp_viewChangeNewtoService = {0,0};
viewChangeNew = new toService("viewChangeNew",temp_viewChangeNewtoService);
profFailedAr_state temp_ProcNrFailedprofFailedAr = {0,0};
ProcNrFailed = new profFailedAr("ProcNrFailed",temp_ProcNrFailedprofFailedAr);
int temp_ServiceReplicaQueueSRQvalue = 10000;
ServiceReplicaQueue = new SRQ("ServiceReplicaQueue",temp_ServiceReplicaQueueSRQvalue);
toService_state temp_ProcFixedtoService = {0,0};
ProcFixed = new toService("ProcFixed",temp_ProcFixedtoService);
int temp_replicasUpreplUpvalue = 0;
replicasUp = new replUp("replicasUp",temp_replicasUpreplUpvalue);
NY_state temp_CreationKOvalue = {-1,0};
Creation = new KO("Creation",temp_CreationKOvalue);
BaseStateVariableClass* InitialPlaces[21]={
  rmUp,  // 0
  procFailed,  // 1
  viewServiceChosen,  // 2
  ServicesInQueue,  // 3
  pointer,  // 4
  queuePointer,  // 5
  readQueuePointer,  // 6
  systemFailed,  // 7
  systemFailure,  // 8
  RManag,  // 9
  ServiceReplicaDist,  // 10
  ServiceParam,  // 11
  LoadDist,  // 12
  Processors,  // 13
  viewNew,  // 14
  viewChangeNew,  // 15
  ProcNrFailed,  // 16
  ServiceReplicaQueue,  // 17
  ProcFixed,  // 18
  replicasUp,  // 19
  Creation   // 20
};
BaseStateVariableClass* InitialROPlaces[0]={
};
initializeSANModelNow("RM", 21, InitialPlaces,
                      0, InitialROPlaces,
                      6, InitialActionList, 6, InitialGroupList);
```

```
  assignPlacesToActivitiesInst();
  assignPlacesToActivitiesTimed();

  int AffectArcs[32][2]={
    {16,0}, {9,0}, {12,0}, {1,0}, {5,1}, {6,1}, {3,1}, {18,1},
    {17,1}, {20,1}, {4,2}, {3,2}, {5,2}, {15,2}, {20,2}, {13,2},
    {10,2}, {12,2}, {19,2}, {11,2}, {17,2}, {8,3}, {7,3}, {0,4},
    {8,4}, {9,4}, {12,4}, {1,5}, {0,5}, {13,5}, {9,5}, {12,5}
  };
  for(int n=0;n<32;n++) {
    AddAffectArc(InitialPlaces[AffectArcs[n][0]],
                 InitialActionList[AffectArcs[n][1]]);
  }
  int EnableArcs[8][2]={
    {9,0}, {16,0}, {18,1}, {15,2}, {0,2}, {8,3}, {0,4}, {0,5}
  };
  for(int n=0;n<8;n++) {
    AddEnableArc(InitialPlaces[EnableArcs[n][0]],
                 InitialActionList[EnableArcs[n][1]]);
  }

  for(int n=0;n<6;n++) {
    InitialActionList[n]->LinkVariables();
  }
  CustomInitialization();

}

void RMSAN::CustomInitialization() {
RManag->Index(0)->Mark() = 1;
RManag->Index(1)->Mark() = 1;
RManag->Index(2)->Mark() = 1;
RManag->Index(3)->Mark() = 0;

if (numServices == 3){
ServiceParam->Index(0)->Index(0)->Mark() = 3;
ServiceParam->Index(0)->Index(1)->Mark() = 3;
ServiceParam->Index(0)->Index(2)->Mark() = 1;

ServiceParam->Index(1)->Index(0)->Mark() = 2;
ServiceParam->Index(1)->Index(1)->Mark() = 2;
ServiceParam->Index(1)->Index(2)->Mark() = 1;

ServiceParam->Index(2)->Index(0)->Mark() = 3;
ServiceParam->Index(2)->Index(1)->Mark() = 3;
ServiceParam->Index(2)->Index(2)->Mark() = 1;

int i;
for(i=0; i<numServices; i++){
        replicasUp->Index(i)->Mark()= ServiceParam->Index(i)->Index(1)->Mark();
}
int j;
for(j=0; j<numServices; j++){
        viewNew->Index(j)->Mark() = ServiceParam->Index(j)->Index(0)->Mark();
}

ServiceReplicaDist->Index(0)->Index(0)->Mark() = 1;
ServiceReplicaDist->Index(0)->Index(1)->Mark() = 1;
ServiceReplicaDist->Index(0)->Index(2)->Mark() = 1;
ServiceReplicaDist->Index(0)->Index(3)->Mark() = 0;

ServiceReplicaDist->Index(1)->Index(0)->Mark() = 0;
```

```
ServiceReplicaDist->Index(1)->Index(1)->Mark() = 1;
ServiceReplicaDist->Index(1)->Index(2)->Mark() = 1;
ServiceReplicaDist->Index(1)->Index(3)->Mark() = 0;

ServiceReplicaDist->Index(2)->Index(0)->Mark() = 1;
ServiceReplicaDist->Index(2)->Index(1)->Mark() = 0;
ServiceReplicaDist->Index(2)->Index(2)->Mark() = 1;
ServiceReplicaDist->Index(2)->Index(3)->Mark() = 1;

int k;
for(k=0; k<numProc; k++){
        LoadDist->Index(k)->Mark() = (ServiceReplicaDist->Index(0)->Index(k)->Mark()
+ServiceReplicaDist->Index(1)->Index(k)->Mark()
+ServiceReplicaDist->Index(2)->Index(k)->Mark()
+RManag->Index(k)->Mark());
}
}else if (numServices == 4){
ServiceParam->Index(0)->Index(0)->Mark() = 3;
ServiceParam->Index(0)->Index(1)->Mark() = 3;
ServiceParam->Index(0)->Index(2)->Mark() = 1;

ServiceParam->Index(1)->Index(0)->Mark() = 2;
ServiceParam->Index(1)->Index(1)->Mark() = 2;
ServiceParam->Index(1)->Index(2)->Mark() = 1;

ServiceParam->Index(2)->Index(0)->Mark() = 3;
ServiceParam->Index(2)->Index(1)->Mark() = 3;
ServiceParam->Index(2)->Index(2)->Mark() = 1;

ServiceParam->Index(2)->Index(0)->Mark() = 4;
ServiceParam->Index(2)->Index(1)->Mark() = 4;
ServiceParam->Index(2)->Index(2)->Mark() = 2;

int i;
for(i=0; i<numServices; i++){
        replicasUp->Index(i)->Mark()= ServiceParam->Index(i)->Index(1)->Mark();
}
int j;
for(j=0; j<numServices; j++){
        viewNew->Index(j)->Mark() = ServiceParam->Index(j)->Index(0)->Mark();
}

ServiceReplicaDist->Index(0)->Index(0)->Mark() = 1;
ServiceReplicaDist->Index(0)->Index(1)->Mark() = 1;
ServiceReplicaDist->Index(0)->Index(2)->Mark() = 1;
ServiceReplicaDist->Index(0)->Index(3)->Mark() = 0;

ServiceReplicaDist->Index(1)->Index(0)->Mark() = 0;
ServiceReplicaDist->Index(1)->Index(1)->Mark() = 1;
ServiceReplicaDist->Index(1)->Index(2)->Mark() = 1;
ServiceReplicaDist->Index(1)->Index(3)->Mark() = 0;

ServiceReplicaDist->Index(2)->Index(0)->Mark() = 1;
ServiceReplicaDist->Index(2)->Index(1)->Mark() = 0;
ServiceReplicaDist->Index(2)->Index(2)->Mark() = 1;
ServiceReplicaDist->Index(2)->Index(3)->Mark() = 1;

ServiceReplicaDist->Index(2)->Index(0)->Mark() = 1;
ServiceReplicaDist->Index(2)->Index(1)->Mark() = 1;
ServiceReplicaDist->Index(2)->Index(2)->Mark() = 1;
ServiceReplicaDist->Index(2)->Index(3)->Mark() = 1;
```

```
int k;
for(k=0; k<numProc; k++){
        LoadDist->Index(k)->Mark() = (ServiceReplicaDist->Index(0)->Index(k)->Mark()
+ServiceReplicaDist->Index(1)->Index(k)->Mark()
+ServiceReplicaDist->Index(2)->Index(k)->Mark()
+RManag->Index(k)->Mark());
}

}



}
RMSAN::~RMSAN(){
  for (int i = 0; i < NumStateVariables-NumReadOnlyPlaces; i++)
    delete LocalStateVariables[i];
};

void RMSAN::assignPlacesToActivitiesInst(){
  reactToProcFailed.RManag = (RM*) LocalStateVariables[9];
  reactToProcFailed.ProcNrFailed = (profFailedAr*) LocalStateVariables[16];
  reactToProcFailed.LoadDist = (procLoad*) LocalStateVariables[12];
  reactToProcFailed.procFailed = (Place*) LocalStateVariables[1];
  reactToProcFixed.ProcFixed = (toService*) LocalStateVariables[18];
  reactToProcFixed.queuePointer = (Place*) LocalStateVariables[5];
  reactToProcFixed.readQueuePointer = (Place*) LocalStateVariables[6];
  reactToProcFixed.ServicesInQueue = (Place*) LocalStateVariables[3];
  reactToProcFixed.ServiceReplicaQueue = (SRQ*) LocalStateVariables[17];
  reactToProcFixed.Creation = (KO*) LocalStateVariables[20];
  reactioToViewChange.viewChangeNew = (toService*) LocalStateVariables[15];
  reactioToViewChange.rmUp = (Place*) LocalStateVariables[0];
  reactioToViewChange.pointer = (Place*) LocalStateVariables[4];
  reactioToViewChange.ServicesInQueue = (Place*) LocalStateVariables[3];
  reactioToViewChange.queuePointer = (Place*) LocalStateVariables[5];
  reactioToViewChange.Creation = (KO*) LocalStateVariables[20];
  reactioToViewChange.Processors = (proc*) LocalStateVariables[13];
  reactioToViewChange.ServiceReplicaDist = (hvemhvahvor*) LocalStateVariables[10];
  reactioToViewChange.LoadDist = (procLoad*) LocalStateVariables[12];
  reactioToViewChange.replicasUp = (replUp*) LocalStateVariables[19];
  reactioToViewChange.ServiceParam = (serv*) LocalStateVariables[11];
  reactioToViewChange.ServiceReplicaQueue = (SRQ*) LocalStateVariables[17];
  Instantaneous_Activity1.systemFailure = (Place*) LocalStateVariables[8];
  Instantaneous_Activity1.systemFailed = (Place*) LocalStateVariables[7];
}
void RMSAN::assignPlacesToActivitiesTimed(){
  RMfail.rmUp = (Place*) LocalStateVariables[0];
  RMfail.systemFailure = (Place*) LocalStateVariables[8];
  RMfail.RManag = (RM*) LocalStateVariables[9];
  RMfail.LoadDist = (procLoad*) LocalStateVariables[12];
  RMrepair.rmUp = (Place*) LocalStateVariables[0];
  RMrepair.procFailed = (Place*) LocalStateVariables[1];
  RMrepair.Processors = (proc*) LocalStateVariables[13];
  RMrepair.RManag = (RM*) LocalStateVariables[9];
  RMrepair.LoadDist = (procLoad*) LocalStateVariables[12];
}
/*****************************************************************/
/*                  Activity Method Definitions                  */
/*****************************************************************/

/*======================reactToProcFailedActivity=======================*/
```

```
RMSAN::reactToProcFailedActivity::reactToProcFailedActivity(){
  ActivityInitialize("reactToProcFailed",2,Instantaneous , RaceEnabled, 4,2, false);
}

void RMSAN::reactToProcFailedActivity::LinkVariables(){



  procFailed->Register(&procFailed_Mobius_Mark);
}

bool RMSAN::reactToProcFailedActivity::Enabled(){
  OldEnabled=NewEnabled;
  NewEnabled=((RManag->Index(ProcNrFailed->Index(1)->Mark())->Mark()== 1 &&
        ProcNrFailed->Index(0)->Mark()==1));
  return NewEnabled;
}

double RMSAN::reactToProcFailedActivity::Weight(){
  return 1;
}

bool RMSAN::reactToProcFailedActivity::ReactivationPredicate(){
  return false;
}

bool RMSAN::reactToProcFailedActivity::ReactivationFunction(){
  return false;
}

double RMSAN::reactToProcFailedActivity::SampleDistribution(){
  return 0;
}

double* RMSAN::reactToProcFailedActivity::ReturnDistributionParameters(){
    return NULL;
}

int RMSAN::reactToProcFailedActivity::Rank(){
  return 1;
}

BaseActionClass* RMSAN::reactToProcFailedActivity::Fire(){
  ;
  //React to processor failure
if(ProcNrFailed->Index(0)->Mark()==1){
        int procFailed = ProcNrFailed->Index(1)->Mark();
        int i;

        RManag->Index(procFailed)->Mark() = 0;
        LoadDist->Index(procFailed)->Mark() = 0;
        ProcNrFailed ->Index(0)->Mark() = 0;
        ProcNrFailed ->Index(1)->Mark() = 0;
}

  return this;
}

/*======================reactToProcFixedActivity=======================*/


RMSAN::reactToProcFixedActivity::reactToProcFixedActivity(){
```

```
  ActivityInitialize("reactToProcFixed",3,Instantaneous , RaceEnabled, 6,1, false);
}

void RMSAN::reactToProcFixedActivity::LinkVariables(){

  queuePointer->Register(&queuePointer_Mobius_Mark);
  readQueuePointer->Register(&readQueuePointer_Mobius_Mark);
  ServicesInQueue->Register(&ServicesInQueue_Mobius_Mark);


}

bool RMSAN::reactToProcFixedActivity::Enabled(){
  OldEnabled=NewEnabled;
  NewEnabled=((ProcFixed->Index(0)->Mark() == 1));
  return NewEnabled;
}

double RMSAN::reactToProcFixedActivity::Weight(){
  return 1;
}

bool RMSAN::reactToProcFixedActivity::ReactivationPredicate(){
  return false;
}

bool RMSAN::reactToProcFixedActivity::ReactivationFunction(){
  return false;
}

double RMSAN::reactToProcFixedActivity::SampleDistribution(){
  return 0;
}

double* RMSAN::reactToProcFixedActivity::ReturnDistributionParameters(){
    return NULL;
}

int RMSAN::reactToProcFixedActivity::Rank(){
  return 1;
}

BaseActionClass* RMSAN::reactToProcFixedActivity::Fire(){
  ;
  //check queue to see if it contains any services. FCFS.
int i;

int fixed = ProcFixed->Index(1)->Mark();
if(ServiceReplicaQueue->Index(queuePointer->Mark())->Mark()<numServices){

        int ServiceCreation = ServiceReplicaQueue->Index(0)->Mark();
    Creation->Index(ServiceCreation)->Index(0)->Mark() = 1;
    Creation->Index(ServiceCreation)->Index(1)->Mark() = fixed;

    for(i=0; i<numServices; i++){
        if(ServiceReplicaQueue->Index(i)->Mark()<numServices){
            ServiceReplicaQueue->Index(i)->Mark() = ServiceReplicaQueue->Index(i+1)->Mark();
            if(ServiceReplicaQueue->Index(i+2)->Mark()>numServices){
                ServiceReplicaQueue->Index(i+1)->Mark() = 11000;
            }
        }
    }
```

```
        readQueuePointer->Mark() = ((readQueuePointer->Mark()+1)%numServices);
        ServicesInQueue->Mark() --;
}



//Load sharing functionality; not yet implemented
//int j;

//for(j=0; j<numProc; j++){
//if(LoadDist->Index(fixed)->Mark() < numServices){
//        if(LoadDist->Index(j)->Mark() > ((numServices+1)/2)){
//tjenesten som velges må ha flere enn en replica!
//
//endre servicereplicaDist for valgt tjeneste!
//        int possible[numServices];
//        int n=0;
//        int k=0;
//                for(k=0; k<numServices; k++){
//                        if(ServiceReplicaDist->Index(k)->Index(j)->Mark()==1 &&
                                replicasUp->Index(k)->Mark()>1){
//                                possible[n] = k;
//                                n++;
//                        }
//                }
//                if(n>0){
//
//Velger en tilfeldig av de mulige tjenestene

//                        int choise = possible[(rand() % n)];
//                        LoadDist->Index(j)->Mark() --;
//                        ServiceReplicaDist->Index(choise)->Index(j)->Mark() = 0;
//                        replicasUp->Index(choise)->Mark() --;
//                        Creation->Index(pointer->Mark())->Index(0)->Mark() = fixed;
//                        Creation->Index(pointer->Mark())->Index(0)->Mark() = choise;
//                        pointer->Mark() = ((pointer->Mark()+1)%numServices);
//                }
//        }
//}
//}
ProcFixed->Index(0)->Mark() = 0;
ProcFixed->Index(1)->Mark() = 0;
  return this;
}

/*======================reactioToViewChangeActivity=======================*/


RMSAN::reactioToViewChangeActivity::reactioToViewChangeActivity(){
  ActivityInitialize("reactioToViewChange",4,Instantaneous , RaceEnabled, 11,2, false);
}

void RMSAN::reactioToViewChangeActivity::LinkVariables(){

  rmUp->Register(&rmUp_Mobius_Mark);
  pointer->Register(&pointer_Mobius_Mark);
  ServicesInQueue->Register(&ServicesInQueue_Mobius_Mark);
  queuePointer->Register(&queuePointer_Mobius_Mark);
```

```
}

bool RMSAN::reactioToViewChangeActivity::Enabled(){
  OldEnabled=NewEnabled;
  NewEnabled=((viewChangeNew->Index(0)->Mark()==1 && rmUp ->Mark() > 0));
  return NewEnabled;
}

double RMSAN::reactioToViewChangeActivity::Weight(){
  return 1;
}

bool RMSAN::reactioToViewChangeActivity::ReactivationPredicate(){
  return false;
}

bool RMSAN::reactioToViewChangeActivity::ReactivationFunction(){
  return false;
}

double RMSAN::reactioToViewChangeActivity::SampleDistribution(){
  return 0;
}

double* RMSAN::reactioToViewChangeActivity::ReturnDistributionParameters(){
    return NULL;
}

int RMSAN::reactioToViewChangeActivity::Rank(){
  return 1;
}

BaseActionClass* RMSAN::reactioToViewChangeActivity::Fire(){
  ;



int ytre;
int n= 0;

//finner hvilken tjeneste som har endret view
int servChosen = viewChangeNew->Index(1)->Mark();
int procChosen;
int load;

int possible[numProc];



int counter;

        //Add creation to creationQueue if not allready in it.

        int k;
        int inQueue=0;
        for(k=0; k<numServices; k++){
                if(servChosen == Creation->Index(k)->Index(0)->Mark()){
                        inQueue = 1;
```

```
                }
        }

//Add service to creation queue, if it is not already in it.
if(inQueue==0){

//Array of processors the replica can be created on
for(counter = 0; counter < numProc; counter++){
    if((Processors->Index(counter)->Mark() == 1) &&
(ServiceReplicaDist->Index(servChosen)->Index(counter)->Mark()==0 &&
LoadDist->Index(counter)->Mark() < Cap) ){
        if(replicasUp->Index(servChosen)->Mark() <
ServiceParam->Index(servChosen)->Index(1)->Mark()){
                possible[n]= counter;
                n++;
        }
        }
}

if(n>0){


                  //Find the processor with the lightest load
        int j;
        int temp = 100000;
        for(j=0; j<n; j++){
                    if(LoadDist->Index(possible[j])->Mark()<temp){
                        temp = LoadDist->Index(possible[j])->Mark();
                        procChosen = possible[j];
                    }
        }



        Creation->Index(pointer->Mark())->Index(0)->Mark() = servChosen;
        Creation->Index(pointer->Mark())->Index(1)->Mark() = procChosen;
        pointer->Mark()= ((pointer->Mark()+1)% numServices);
}else{
        //Add service to queue if it is not allready in it
        int x;
        int test;
        for(x=0; x<ServicesInQueue->Mark(); x++ ){
                if(servChosen == ServiceReplicaQueue->Index(
queuePointer->Mark())->Mark()){
                        test= 1;
                }
        }
        if(test==0){
                ServiceReplicaQueue->Index(queuePointer->Mark())->Mark() =
servChosen;
                ServicesInQueue->Mark()++;
                queuePointer->Mark() = ((queuePointer->Mark()+1)%numServices);
        }
}
}

viewChangeNew->Index(0)->Mark()=0;
viewChangeNew->Index(1)->Mark()=0;
  return this;
}

/*====================Instantaneous_Activity1Activity======================*/
```

```
RMSAN::Instantaneous_Activity1Activity::Instantaneous_Activity1Activity(){
  ActivityInitialize("Instantaneous_Activity1",5,Instantaneous , RaceEnabled, 2,1,
        false);
}

void RMSAN::Instantaneous_Activity1Activity::LinkVariables(){
  systemFailure->Register(&systemFailure_Mobius_Mark);
  systemFailed->Register(&systemFailed_Mobius_Mark);
}

bool RMSAN::Instantaneous_Activity1Activity::Enabled(){
  OldEnabled=NewEnabled;
  NewEnabled=((systemFailure->Mark() == 1));
  return NewEnabled;
}

double RMSAN::Instantaneous_Activity1Activity::Weight(){
  return 1;
}

bool RMSAN::Instantaneous_Activity1Activity::ReactivationPredicate(){
  return false;
}

bool RMSAN::Instantaneous_Activity1Activity::ReactivationFunction(){
  return false;
}

double RMSAN::Instantaneous_Activity1Activity::SampleDistribution(){
  return 0;
}

double* RMSAN::Instantaneous_Activity1Activity::ReturnDistributionParameters(){
    return NULL;
}

int RMSAN::Instantaneous_Activity1Activity::Rank(){
  return 1;
}

BaseActionClass* RMSAN::Instantaneous_Activity1Activity::Fire(){
  ;
  systemFailure->Mark() = 0;
systemFailed->Mark() ++;
  return this;
}

/*======================RMfailActivity=======================*/

RMSAN::RMfailActivity::RMfailActivity(){
  TheDistributionParameters = new double[1];
  ActivityInitialize("RMfail",0,Exponential, RaceEnabled, 4,1, false);
}

RMSAN::RMfailActivity::~RMfailActivity(){
  delete[] TheDistributionParameters;
}

void RMSAN::RMfailActivity::LinkVariables(){
  rmUp->Register(&rmUp_Mobius_Mark);
```

```
  systemFailure->Register(&systemFailure_Mobius_Mark);


}

bool RMSAN::RMfailActivity::Enabled(){
  OldEnabled=NewEnabled;
  NewEnabled=((rmUp->Mark()>0));
  return NewEnabled;
}

double RMSAN::RMfailActivity::Rate(){
  return (rmUp->Mark()+1)*lambdaProc;
  return 1.0;  // default rate if none is specified
}

double RMSAN::RMfailActivity::Weight(){
  return 1;
}

bool RMSAN::RMfailActivity::ReactivationPredicate(){
  return false;
}

bool RMSAN::RMfailActivity::ReactivationFunction(){
  return false;
}

double RMSAN::RMfailActivity::SampleDistribution(){
  return TheDistribution->Exponential((rmUp->Mark()+1)*lambdaProc);
}

double* RMSAN::RMfailActivity::ReturnDistributionParameters(){
  TheDistributionParameters[0] = Rate();
  return TheDistributionParameters;
}

int RMSAN::RMfailActivity::Rank(){
  return 1;
}

BaseActionClass* RMSAN::RMfailActivity::Fire(){
  ;
  #include <cstdlib>

//Create an array of all operative rmReplicas.

int a;
int b;
int i;
int n=0;
int o = rmUp->Mark();
int Up [o];

for(i=0; i<numProc; i++){

        if(RManag->Index(i)->Mark()==1){
                Up[n]=i;
                n++;
        }
}
if(n>o){
```

```
//Pick one random replica that failes.
a = (rand() % o);
b = Up[a];
RManag -> Index(b) -> Mark() = 0;
LoadDist->Index(b)->Mark() --;
}
rmUp->Mark() --;

//Capture system failure
if(rmUp->Mark() <= 1){
        systemFailure->Mark() = 1;
}


  return this;
}

/*======================RMrepairActivity=======================*/

RMSAN::RMrepairActivity::RMrepairActivity(){
  TheDistributionParameters = new double[1];
  ActivityInitialize("RMrepair",1,Exponential, RaceEnabled, 5,1, false);
}

RMSAN::RMrepairActivity::~RMrepairActivity(){
  delete[] TheDistributionParameters;
}

void RMSAN::RMrepairActivity::LinkVariables(){
  rmUp->Register(&rmUp_Mobius_Mark);
  procFailed->Register(&procFailed_Mobius_Mark);



}

bool RMSAN::RMrepairActivity::Enabled(){
  OldEnabled=NewEnabled;
  NewEnabled=((rmUp->Mark()<initRM));
  return NewEnabled;
}

double RMSAN::RMrepairActivity::Rate(){
  return muAut;
  return 1.0;  // default rate if none is specified
}

double RMSAN::RMrepairActivity::Weight(){
  return 1;
}

bool RMSAN::RMrepairActivity::ReactivationPredicate(){
  return false;
}

bool RMSAN::RMrepairActivity::ReactivationFunction(){
  return false;
}

double RMSAN::RMrepairActivity::SampleDistribution(){
  return TheDistribution->Exponential(muAut);
}
```

```
double* RMSAN::RMrepairActivity::ReturnDistributionParameters(){
  TheDistributionParameters[0] = Rate();
  return TheDistributionParameters;
}

int RMSAN::RMrepairActivity::Rank(){
  return 1;
}

BaseActionClass* RMSAN::RMrepairActivity::Fire(){
  ;
  //Create an array with the size of the number of processors a
new replica can be created on
//Add the matches (the processor) to the array (processor up, no replica)
//Take a random processor
//Create replica on that processor


int i;
int possibleMatch = (numProc-procFailed->Mark()) - (rmUp->Mark());
int Match [possibleMatch];
int counter=0;
for(i=0; i<numProc; i++){

        if(Processors->Index(i)->Mark()==1 && RManag->Index(i)->Mark()==0 &&
 LoadDist->Index(i)->Mark()<Cap){
                Match[counter]=i;
                counter++;
        }
}

int random;
random = (rand() % counter);
int rmToBeRepaired = Match[random];
RManag->Index(rmToBeRepaired)->Mark()= 1;
LoadDist->Index(rmToBeRepaired)->Mark() ++;


rmUp->Mark()++;
  return this;
}
```

A.4 The composed SAN

```
#include "Composed/compNew/compNewRJ.h"
char * compNewRJ__SharedNames[] = {"Creation", "LoadDist", "ProcFixed",
"ProcNrFailed", "ProcNrFailedService", "Processors", "ServiceReplicaDist",
"procFailed", "replicasUp", "systemFailure", "viewChangeNew", "viewNew", "viewServiceChosen"};

compNewRJ::compNewRJ():Join("System", 3, 13,compNewRJ__SharedNames) {
  HW = new HWSAN();
  ModelArray[0] = (BaseModelClass*) HW;
  ModelArray[0]->DefineName("HW");
  Service = new ServiceSAN();
```

```
ModelArray[1] = (BaseModelClass*) Service;
ModelArray[1]->DefineName("Service");
RM = new RMSAN();
ModelArray[2] = (BaseModelClass*) RM;
ModelArray[2]->DefineName("RM");

SetupActions();
if (AllChildrenEmpty())
  NumSharedStateVariables = 0;
else {
  //**************  State sharing info  **************
  //Shared variable 0
  Creation = new KO("Creation");
  addSharedPtr(Creation, "Creation" );
  if (Service->NumStateVariables > 0) {
    Creation->ShareWith(Service->Creation);
    addSharingInfo(Service->Creation, Creation);
  }
  if (RM->NumStateVariables > 0) {
    Creation->ShareWith(RM->Creation);
    addSharingInfo(RM->Creation, Creation);
  }

  //Shared variable 1
  LoadDist = new procLoad("LoadDist");
  addSharedPtr(LoadDist, "LoadDist" );
  if (Service->NumStateVariables > 0) {
    LoadDist->ShareWith(Service->LoadDist);
    addSharingInfo(Service->LoadDist, LoadDist);
  }
  if (RM->NumStateVariables > 0) {
    LoadDist->ShareWith(RM->LoadDist);
    addSharingInfo(RM->LoadDist, LoadDist);
  }

  //Shared variable 2
  ProcFixed = new toService("ProcFixed");
  addSharedPtr(ProcFixed, "ProcFixed" );
  if (HW->NumStateVariables > 0) {
    ProcFixed->ShareWith(HW->ProcFixed);
    addSharingInfo(HW->ProcFixed, ProcFixed);
  }
  if (RM->NumStateVariables > 0) {
    ProcFixed->ShareWith(RM->ProcFixed);
    addSharingInfo(RM->ProcFixed, ProcFixed);
  }

  //Shared variable 3
  ProcNrFailed = new profFailedAr("ProcNrFailed");
  addSharedPtr(ProcNrFailed, "ProcNrFailed" );
  if (HW->NumStateVariables > 0) {
    ProcNrFailed->ShareWith(HW->ProcNrFailed);
    addSharingInfo(HW->ProcNrFailed, ProcNrFailed);
  }
  if (RM->NumStateVariables > 0) {
    ProcNrFailed->ShareWith(RM->ProcNrFailed);
    addSharingInfo(RM->ProcNrFailed, ProcNrFailed);
  }

  //Shared variable 4
  ProcNrFailedService = new profFailedAr("ProcNrFailedService");
  addSharedPtr(ProcNrFailedService, "ProcNrFailedService" );
```

```
if (HW->NumStateVariables > 0) {
  ProcNrFailedService->ShareWith(HW->ProcNrFailedService);
  addSharingInfo(HW->ProcNrFailedService, ProcNrFailedService);
}
if (Service->NumStateVariables > 0) {
  ProcNrFailedService->ShareWith(Service->ProcNrFailedService);
  addSharingInfo(Service->ProcNrFailedService, ProcNrFailedService);
}

//Shared variable 5
Processors = new proc("Processors");
addSharedPtr(Processors, "Processors" );
if (HW->NumStateVariables > 0) {
  Processors->ShareWith(HW->Processors);
  addSharingInfo(HW->Processors, Processors);
}
if (Service->NumStateVariables > 0) {
  Processors->ShareWith(Service->Processors);
  addSharingInfo(Service->Processors, Processors);
}
if (RM->NumStateVariables > 0) {
  Processors->ShareWith(RM->Processors);
  addSharingInfo(RM->Processors, Processors);
}

//Shared variable 6
ServiceReplicaDist = new hvemhvahvor("ServiceReplicaDist");
addSharedPtr(ServiceReplicaDist, "ServiceReplicaDist" );
if (Service->NumStateVariables > 0) {
  ServiceReplicaDist->ShareWith(Service->ServiceReplicaDist);
  addSharingInfo(Service->ServiceReplicaDist, ServiceReplicaDist);
}
if (RM->NumStateVariables > 0) {
  ServiceReplicaDist->ShareWith(RM->ServiceReplicaDist);
  addSharingInfo(RM->ServiceReplicaDist, ServiceReplicaDist);
}

//Shared variable 7
procFailed = new Place("procFailed");
addSharedPtr(procFailed, "procFailed" );
if (HW->NumStateVariables > 0) {
  procFailed->ShareWith(HW->procFailed);
  addSharingInfo(HW->procFailed, procFailed);
}
if (Service->NumStateVariables > 0) {
  procFailed->ShareWith(Service->procFailed);
  addSharingInfo(Service->procFailed, procFailed);
}
if (RM->NumStateVariables > 0) {
  procFailed->ShareWith(RM->procFailed);
  addSharingInfo(RM->procFailed, procFailed);
}

//Shared variable 8
replicasUp = new replUp("replicasUp");
addSharedPtr(replicasUp, "replicasUp" );
if (Service->NumStateVariables > 0) {
  replicasUp->ShareWith(Service->replicasUp);
  addSharingInfo(Service->replicasUp, replicasUp);
}
if (RM->NumStateVariables > 0) {
  replicasUp->ShareWith(RM->replicasUp);
```

```
      addSharingInfo(RM->replicasUp, replicasUp);
    }

    //Shared variable 9
    systemFailure = new Place("systemFailure");
    addSharedPtr(systemFailure, "systemFailure" );
    if (HW->NumStateVariables > 0) {
      systemFailure->ShareWith(HW->systemFailure);
      addSharingInfo(HW->systemFailure, systemFailure);
    }
    if (Service->NumStateVariables > 0) {
      systemFailure->ShareWith(Service->systemFailure);
      addSharingInfo(Service->systemFailure, systemFailure);
    }
    if (RM->NumStateVariables > 0) {
      systemFailure->ShareWith(RM->systemFailure);
      addSharingInfo(RM->systemFailure, systemFailure);
    }

    //Shared variable 10
    viewChangeNew = new toService("viewChangeNew");
    addSharedPtr(viewChangeNew, "viewChangeNew" );
    if (Service->NumStateVariables > 0) {
      viewChangeNew->ShareWith(Service->viewChangeNew);
      addSharingInfo(Service->viewChangeNew, viewChangeNew);
    }
    if (RM->NumStateVariables > 0) {
      viewChangeNew->ShareWith(RM->viewChangeNew);
      addSharingInfo(RM->viewChangeNew, viewChangeNew);
    }

    //Shared variable 11
    viewNew = new viewArray("viewNew");
    addSharedPtr(viewNew, "viewNew" );
    if (Service->NumStateVariables > 0) {
      viewNew->ShareWith(Service->viewNew);
      addSharingInfo(Service->viewNew, viewNew);
    }
    if (RM->NumStateVariables > 0) {
      viewNew->ShareWith(RM->viewNew);
      addSharingInfo(RM->viewNew, viewNew);
    }

    //Shared variable 12
    viewServiceChosen = new Place("viewServiceChosen");
    addSharedPtr(viewServiceChosen, "viewServiceChosen" );
    if (Service->NumStateVariables > 0) {
      viewServiceChosen->ShareWith(Service->viewServiceChosen);
      addSharingInfo(Service->viewServiceChosen, viewServiceChosen);
    }
    if (RM->NumStateVariables > 0) {
      viewServiceChosen->ShareWith(RM->viewServiceChosen);
      addSharingInfo(RM->viewServiceChosen, viewServiceChosen);
    }

  }

  Setup();
}

compNewRJ::~compNewRJ() {
  if (!AllChildrenEmpty()) {
```

```
    delete Creation;
    delete LoadDist;
    delete ProcFixed;
    delete ProcNrFailed;
    delete ProcNrFailedService;
    delete Processors;
    delete ServiceReplicaDist;
    delete procFailed;
    delete replicasUp;
    delete systemFailure;
    delete viewChangeNew;
    delete viewNew;
    delete viewServiceChosen;
  }
  delete HW;
  delete Service;
  delete RM;
}
```

Appendix B

Transition matrix

```
1
5

1
2
4.0000000000e-002
0
2
1
5.0000000000e-001
3
3.0000000000e-002
0
3
2
5.0000000000e-001
4
2.0000000000e-002
0
4
3
5.0000000000e-001
5
1.0000000000e-002
0
5
4
```

```
5.0000000000e-001
0
```

B.1 Interpreting the transition matrix

The first two rows indicate the number of states. The matrix is given as a sparse matrix with a zero separating the outgoing states (rows of the matrix). The matrix represented above corresponds to the matrix in equation 9.1 on page 79.

For each state the successor states are listed, followed by the rate of the event.

Due to the size, the transition matrices for the other submodels and composed models are left out of the appendix.

Appendix C

Simulation results

The simulation results of the simulations run for verification in chapter 9 is attached in Appendix C.1 and Appendix C.2 contains the results of the simulations run in chapter 10.

C.1 Verification simulations

This section provides the complete result files generated during the simulations in Chapter 9.

Simplified system HW SAN results

```
_____
_________________________________Simulator Results_________________________________
Date:                           Fri May 09 11:12:42 CEST 2008

_____________________________Simulator Configuration_____________________________
Simulation Type:                Steady State
Project Name:                   ForsteVersjonVirker
Study Name:                     test
Random Number Generator:        Lagged Fibonacci
Random Number Seed:             31415
Maximum Batches:                100000
Minimum Batches:                10000
Data Reporting Frequency:       1000
Display Update Frequency:       1000
```

```
BuildType:                    Optimize
Runname:                      Results
Output File:                  Results_output.txt
Results File:                 Results_results
Jackknife Variance:           On
Processors Per Experiment:    1
Maximize Processor Usage:     false
Processors Selected for Simulation:
        labbetull03           x86

_____________________________________Experiment 1_____________________________________
Start Time:                   Fri May 09 11:12:41 CEST 2008
Finish Time:                  Fri May 09 11:12:42 CEST 2008
Elapsed Running Time:         0.671
Total CPU Time:               0.281
Batches Completed:            10000
Experiment Name:              Experiment 1
Global Variable Settings:
        double                lambdaProc              0.01
        double                muMan                   0.2
        int                   numFailed               0
        int                   numProc                 4


_____________________________________Mean Results_____________________________________
Name                    Time              Mean                       Confidence
                                                                     Interval
availability                           1,0000000000E00   +/-   5,0586753828E-10


_____________________________________Variance Results_____________________________________


_____________________________________Interval Results_____________________________________


_____________________________________Distribution Results_____________________________________

<END_MOBIUS_RESULTS>
```

Final HW SAN results

```
_____________________________________________________________________
--------------------------------Simulator Results--------------------------------
Date:                          Fri May 09 11:10:32 CEST 2008

-----------------------------Simulator Configuration-----------------------------
Simulation Type:               Steady State
Project Name:                  MS
Study Name:                    HWstudy
Random Number Generator:       Lagged Fibonacci
Random Number Seed:            31415
Maximum Batches:               10000
Minimum Batches:               20
Data Reporting Frequency:      1000
Display Update Frequency:      1000
BuildType:                     Optimize
Runname:                       Results
Output File:                   Results_output.txt
Results File:                  Results_results
Jackknife Variance:            On
Processors Per Experiment:     1
Maximize Processor Usage:      false
Processors Selected for Simulation:
        labbetull03            x86

---------------------------------Experiment 1---------------------------------
Start Time:                    Fri May 09 11:10:31 CEST 2008
Finish Time:                   Fri May 09 11:10:32 CEST 2008
Elapsed Running Time:          0.797
Total CPU Time:                0.031
Batches Completed:             1000
Experiment Name:               Experiment 1
Global Variable Settings:
        double                 lambdaProc              0.01
        double                 muMan                   0.2
        int                    numFailed               0
        int                    numProc                 4
        int                    numServices             3


---------------------------------Mean Results---------------------------------
Name                    Time              Mean                      Confidence
```

```
                                                                    Interval
availHW                                 9,9950366924E-01  +/-   3,7653408127E-04

_______________________________Variance Results_______________________________

_______________________________Interval Results_______________________________

_____________________________Distribution Results_____________________________

<END_MOBIUS_RESULTS>
```

Simplified system results

```
_____________________________Simulator Results_____________________________
Date:                          Fri May 09 11:14:49 CEST 2008

____________________________Simulator Configuration____________________________
Simulation Type:               Steady State
Project Name:                  ForsteVersjonVirker
Study Name:                    composedAvail
Random Number Generator:       Lagged Fibonacci
Random Number Seed:            31415
Maximum Batches:               10000
Minimum Batches:               20
Data Reporting Frequency:      1000
Display Update Frequency:      1000
BuildType:                     Normal
Runname:                       Results
Output File:                   Results_output.txt
Results File:                  Results_results
Jackknife Variance:            On
Processors Per Experiment:     1
Maximize Processor Usage:      false
Processors Selected for Simulation:
        labbetull03            x86

_______________________________Experiment 1_______________________________
Start Time:                    Fri May 09 11:14:40 CEST 2008
```

```
Finish Time:                 Fri May 09 11:14:49 CEST 2008
Elapsed Running Time:        8.547
Total CPU Time:              7.234
Batches Completed:           10000
Experiment Name:             Experiment 1
Global Variable Settings:
        int                  initRepl                3
        double               lambdaProc              0.0010
        double               lambdaRepl              0.0020
        double               lambdaView              0.9
        double               muAut                   2.0
        double               muMan                   0.2
        int                  numFailed               0
        int                  numProc                 4
        int                  replReq                 1


_____________________________________Mean Results_____________________________________
Name                    Time              Mean                       Confidence
                                                                     Interval
availab                                   1,0000000000E00   +/-   2,3807178826E-09
ServiceAvail                              1,0000000000E00   +/-   5,0586753828E-10


_____________________________________Variance Results_____________________________________


_____________________________________Interval Results_____________________________________


_____________________________Distribution Results_____________________________

<END_MOBIUS_RESULTS>
```

Final system results

```
_____________________________________Simulator Results_____________________________________
Date:                        Fri May 09 11:07:42 CEST 2008

_____________________________Simulator Configuration_____________________________
Simulation Type:             Steady State
```

```
Project Name:                MS
Study Name:                  System
Random Number Generator:     Lagged Fibonacci
Random Number Seed:          31415
Maximum Batches:             100000
Minimum Batches:             20
Data Reporting Frequency:    1000
Display Update Frequency:    1000
BuildType:                   Normal
Runname:                     Results
Output File:                 Results_output.txt
Results File:                Results_results
Jackknife Variance:          On
Processors Per Experiment:   1
Maximize Processor Usage:    false
Processors Selected for Simulation:
        labbetull03          x86

---------------------------------Experiment 1---------------------------------
Start Time:                  Fri May 09 11:07:35 CEST 2008
Finish Time:                 Fri May 09 11:07:42 CEST 2008
Elapsed Running Time:        6.531
Total CPU Time:              5.14
Batches Completed:           1000
Experiment Name:             Experiment 1
Global Variable Settings:
        short                Cap                    4
        int                  initRM                 3
        int                  initRepl               3
        double               lambdaProc             0.0010
        double               lambdaRepl             0.0020
        double               lambdaView             0.9
        double               muAut                  2.0
        double               muMan                  0.2
        int                  numFailed              0
        int                  numProc                4
        int                  numServices            3
        short                repChos                0
        int                  replReq                1


---------------------------------Mean Results---------------------------------
Name                    Time              Mean                      Confidence
```

```
                                                                  Interval
systemAvail                          9,9911384121E-01  +/-   1,2638266619E-03
serviceAvail                         9,9997384464E-01  +/-   5,1264511701E-05
HWavail                              1,0000000000E00   +/-   0,0000000000E00


_______________________________Variance Results_______________________________


_______________________________Interval Results________________________________


_____________________________Distribution Results_____________________________

<END_MOBIUS_RESULTS>
```

C.2 Example scenarios

This section provides the complete result files generated during the simulations in Chapter 10.

Experiment 1 results

```
_____________________________Simulator Results_______________________________
Date:                        Mon May 26 14:22:07 CEST 2008

____________________________Simulator Configuration____________________________
Simulation Type:             Steady State
Project Name:                MS
Study Name:                  FinalExp1
Random Number Generator:     Lagged Fibonacci
Random Number Seed:          395
Maximum Batches:             10000
Minimum Batches:             20
Data Reporting Frequency:    1
Display Update Frequency:    1
BuildType:                   Optimize
Runname:                     Results
Output File:                 Results_output.txt
```

```
Results File:                Results_results
Jackknife Variance:          On
Processors Per Experiment:   1
Maximize Processor Usage:    false
Processors Selected for Simulation:
        labbetull03          x86

_____________________________________Experiment 1_____________________________________
Start Time:                  Mon May 26 14:22:06 CEST 2008
Finish Time:                 Mon May 26 14:22:07 CEST 2008
Elapsed Running Time:        0.453
Total CPU Time:              0.359
Batches Completed:           670
Experiment Name:             Experiment 1
Global Variable Settings:
        short                Cap                    4
        int                  initRM                 3
        int                  initRepl               3
        double               lambdaProc             1.0E-4
        double               lambdaRepl             2.0E-4
        double               lambdaView             1.0
        double               muAut                  12.0
        double               muMan                  0.5
        int                  numFailed              0
        int                  numProc                4
        int                  numServices            3
        short                repChos                0
        int                  replReq                1


_____________________________________Mean Results_____________________________________
Name                   Time              Mean                      Confidence
                                                                   Interval
systemAvail                              9,9999915776E-01  +/-   1,9590566490E-06
MTBF                                     5,5970149254E-04  +/-   5,5939506295E-05


_____________________________________Variance Results_____________________________________


_____________________________________Interval Results_____________________________________
```

```
------------------------------Distribution Results------------------------------

<END_MOBIUS_RESULTS>
```

Experiment 2 results

```
--------------------------------Simulator Results-------------------------------
Date:                          Mon May 26 14:23:39 CEST 2008

------------------------------Simulator Configuration-----------------------------
Simulation Type:               Steady State
Project Name:                  MS
Study Name:                    FinalExp2
Random Number Generator:       Lagged Fibonacci
Random Number Seed:            395
Maximum Batches:               10000
Minimum Batches:               20
Data Reporting Frequency:      1
Display Update Frequency:      1
BuildType:                     Optimize
Runname:                       Results
Output File:                   Results_output.txt
Results File:                  Results_results
Jackknife Variance:            On
Processors Per Experiment:     1
Maximize Processor Usage:      false
Processors Selected for Simulation:
        labbetull03            x86

---------------------------------Experiment 2-----------------------------------
Start Time:                    Mon May 26 14:23:38 CEST 2008
Finish Time:                   Mon May 26 14:23:39 CEST 2008
Elapsed Running Time:          0.843
Total CPU Time:                0.437
Batches Completed:             766
Experiment Name:               Experiment 2
Global Variable Settings:
        short                  Cap                     4
        int                    initRM                  3
        int                    initRepl                3
        double                 lambdaProc              1.0E-4
```

```
        double              lambdaRepl              2.0E-4
        double              lambdaView              1.0
        double              muAut                   6.0
        double              muMan                   0.1
        int                 numFailed               0
        int                 numProc                 4
        int                 numServices             3
        short               repChos                 0
        int                 replReq                 1


----------------------------------Mean Results-----------------------------------
Name                  Time              Mean                       Confidence
                                                                   Interval
systemAvail                          9,9999815674E-01  +/-   4,2874149635E-06
MTBF                                 5,3263707572E-04  +/-   5,3259418897E-05


--------------------------------Variance Results---------------------------------


--------------------------------Interval Results---------------------------------


------------------------------Distribution Results-------------------------------

<END_MOBIUS_RESULTS>
```

Experiment 3 results

```
--------------------------------Simulator Results--------------------------------
Date:                         Mon May 26 14:31:41 CEST 2008

-----------------------------Simulator Configuration-----------------------------
Simulation Type:              Steady State
Project Name:                 MS
Study Name:                   FinalExp3
Random Number Generator:      Lagged Fibonacci
Random Number Seed:           395
Maximum Batches:              10000
Minimum Batches:              20
```

```
Data Reporting Frequency:    1
Display Update Frequency:    1
BuildType:                   Optimize
Runname:                     Results
Output File:                 Results_output.txt
Results File:                Results_results
Jackknife Variance:          On
Processors Per Experiment:   1
Maximize Processor Usage:    false
Processors Selected for Simulation:
        labbetull03          x86

_____________________________________Experiment 3_____________________________________
Start Time:                  Mon May 26 14:31:40 CEST 2008
Finish Time:                 Mon May 26 14:31:41 CEST 2008
Elapsed Running Time:        1.469
Total CPU Time:              0.141
Batches Completed:           200
Experiment Name:             Experiment 3
Global Variable Settings:
        short                Cap                     4
        int                  initRM                  3
        int                  initRepl                3
        double               lambdaProc              0.0010
        double               lambdaRepl              0.0020
        double               lambdaView              1.0
        double               muAut                   12.0
        double               muMan                   0.5
        int                  numFailed               0
        int                  numProc                 4
        int                  numServices             3
        short                repChos                 0
        int                  replReq                 1


_____________________________________Mean Results_____________________________________
Name                  Time              Mean                       Confidence Interval
systemAvail                          9,9999805405E-01  +/-   4,9193649033E-06
MTBF                                 4,9850000000E-03  +/-   3,2091320419E-04


_____________________________Variance Results_____________________________
```

```
_______________________________Interval Results________________________________

_____________________________Distribution Results______________________________

<END_MOBIUS_RESULTS>
```

Experiment 4 results

```
_______________________________Simulator Results_______________________________
Date:                        Mon May 26 14:33:08 CEST 2008

____________________________Simulator Configuration____________________________
Simulation Type:             Steady State
Project Name:                MS
Study Name:                  FinalExp4
Random Number Generator:     Lagged Fibonacci
Random Number Seed:          395
Maximum Batches:             10000
Minimum Batches:             20
Data Reporting Frequency:    1
Display Update Frequency:    1
BuildType:                   Optimize
Runname:                     Results
Output File:                 Results_output.txt
Results File:                Results_results
Jackknife Variance:          On
Processors Per Experiment:   1
Maximize Processor Usage:    false
Processors Selected for Simulation:
        labbetull03          x86

_________________________________Experiment 4__________________________________
Start Time:                  Mon May 26 14:33:06 CEST 2008
Finish Time:                 Mon May 26 14:33:06 CEST 2008
Elapsed Running Time:        0.469
Total CPU Time:              0.063
Batches Completed:           110
Experiment Name:             Experiment 4
Global Variable Settings:
```

```
        short               Cap                 4
        int                 initRM              3
        int                 initRepl            3
        double              lambdaProc          0.0010
        double              lambdaRepl          0.0020
        double              lambdaView          1.0
        double              muAut               6.0
        double              muMan               0.1
        int                 numFailed           0
        int                 numProc             4
        int                 numServices         3
        short               repChos             0
        int                 replReq             1


_____________________________________Mean Results_____________________________________
Name                    Time              Mean                       Confidence
                                                                     Interval
systemAvail                              9,9999093500E-01  +/-   2,4638658198E-05
MTBF                                     5,6545454545E-03  +/-   4,6293308908E-04


_____________________________________Variance Results_____________________________________


_____________________________________Interval Results_____________________________________


_____________________________________Distribution Results_____________________________________

<END_MOBIUS_RESULTS>
```