Åsmund Tjora

Modeling Run-Time Distributions in Passively Replicated Fault Tolerant Systems

Thesis for the degree of doktor ingeniør

Trondheim, December 2007

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

NTNU
Norwegian University of
Science and Technology

NTNU
Norwegian University of Science and Technology

Thesis for the degree of doktor ingeniør

Faculty of Information Technology, Mathematics and Electrical Engineering
Department of Engineering Cybernetics

ISBN 978-82-471-5787-9 (printed ver.)
ISBN 978-82-471-5790-9 (electronic ver.)
ISSN 1503-8181
ITK Report 2007-9-W

Theses at NTNU, 2007:259

Printed by Tapir Uttrykk

Summary

Many real-time applications will have strict reliability requirements in addition to the timing requirements. To fulfill these reliability requirements, it may be necessary to use a fault-tolerance strategy.

An active replication strategy, where several instances of the task is run in parallel, is the preferred choice for many real-time systems, as the parallel execution of the task instances gives a high probability that at least some of the instances finish successfully before the deadlines, even if others should fail. However, enabling several parallel executions of single tasks increase the need for processing power, which is costly and increases the requirements to space and energy consumption.

In a passive replication strategy, only one instance of a task is run at a time. If the task fails, a backup is readied, and the task is rerun on the backup. This requires fewer resources than active replication strategies, but the extra time needed for the rerun of the task can increase the probability of deadline misses. Thus, analyzing the timing of these systems is necessary.

Analysis using the worst-case execution times for the tasks in the fault tolerant system can often give very conservative results, especially if the tasks' normal execution times rarely approaches the worst case times.

The analysis of the *run-time distributions* of the tasks in passively replicated fault tolerant systems can be a useful tool for deciding whether a passive replication strategy is suitable for the system or not. Unlike worst-case execution time analysis, the distributions can also show the improvement in reliability for systems where the passive replication strategy does not work in the worst case scenario. This improvement may be so good that it justifies the use of the replication strategy.

In this work, mathematical models for run-time distributions of tasks in several classes of passive replication systems are developed. These models give the run-time distributions as functions of parameter distributions of the modeled system, like the fault-free runtime of a task, the fault detection time distribution, and the distribution of the time between fault detection and the start of the rerun of the task.

The different fault detection mechanisms used in passive replication systems lead to different structure of the mathematical models. Also, whether the replicas are homogeneous or inhomogeneous affect the model structure. Many other differences in the modeled systems' structure will lead to differences in the parameter distributions, but not in the structure of the mathematical models.

Models for systems using homogeneous and inhomogeneous replicas, with

watchdogs, timeouts, and acceptance tests as fault detectors are developed. One of the goals of the work has been to show the steps used to develop the models in a way so the same steps can be used to develop run-time models for systems that are not presented in the work.

The use of the models is shown with several examples, and the example results are compared to results obtained from discrete event system simulation.

Preface

This thesis presents the results of my research as a doctoral student at the Department of Engineering Cybernetics (ITK) at the Norwegian University of Science and Technology (NTNU). The Department of Engineering Cybernetics has also been funding this project.

I would like to thank my supervisor, Associate Professor Amund Skavhaug at the Department of Engineering Cybernetics, who encouraged me to start on the doctoral studies, and who has always helped me look forward, even when the direction of my project changed from what was originally planned. I would also like to thank my co-supervisor, Associate Professor Poul Heegaard at the Department of Telematics. My supervisors' advice, suggestions and questions have been important feedback for my work.

I would like to thank the dissertation committee: Dr. Karl-Erwin Grosspietsch, Dr. Kai Hansen, Professor Ivica Crnkovic and Professor Bjarne Helvik. I would also like to thank Associate Professor Sverre Hendseth who has been the administrator for the committee.

The road to finishing this work has been long and sometimes very difficult. I have several times "lost my sense of direction" and been unsure of how I should proceed, especially at the beginning of the study – something that is reflected in the fact that this work ended up being very different from what was the original plan. Thanks to all the people who have given me good ideas, comments and advice and helped me towards the goal. I would especially like to thank Professor Peder Emstad, whose course on traffic analysis introduced me to the use of moment-generating functions, and Professor Gerhard Chroust, whose comments to one of my workshop presentations gave me many of the ideas I've been working with.

I would like to thank to all the people at "D-blokka" for all their help and encouragement. Special thanks goes to Eva, Tove, and Unni who have always been helpful, to the regulars at the lunchroom, and to my office-mates Trygve, Ekrem, and Kenneth. Good friends at work and an inspiring working environment have been very important for me.

A warm thanks goes to my parents for telling me that I should never give up, even when the challenges seem to be too great. Without their support and encouragement, I would never have been able to finish this work.

Trondheim, November 2007,
Åsmund Tjora

Contents

Chapter 1

Introduction

When using a computer system to control another (usually non-computer) system, like an engine or a processing plant, the computer system is often not only required to produce the results needed for the control task, but also to produce these results at the correct time. If the state of the controlled system changes quickly, the control signals from the controlling computer system may be incorrect if they are a few milliseconds delayed, which means the computer system used for control must be able to handle strict timing requirements.

A *real-time computer system* is a computer system that is designed so that tasks running in the system can meet the strict timing requirements of the "real world", that is, processes that are outside of the computer system. The processing of real-time tasks running in the system must produce results within *deadlines* decided by the timing of the "real world" processes, and failure to meet such a deadline may result in a degradation of the overall system's performance, or even the failure of the system.

While high processing speed is sought after in most computer systems, in real-time system, *temporal determinism* is of higher importance than speed [43], as the system may fail if a single deadline is missed, even if all other deadlines are met by a large margin. For example, a speed-up strategy that increases the average processing speed by a large amount will be preferred for batch-processing systems, but may be unsuitable for real-time system if the speed-up is at the cost of occasional long delays that may cause deadline misses, as shown in figure 1.2.

Many real-time applications will, in addition to the timing requirements, also have requirements to the system's reliability [18]. The halt or the incorrect

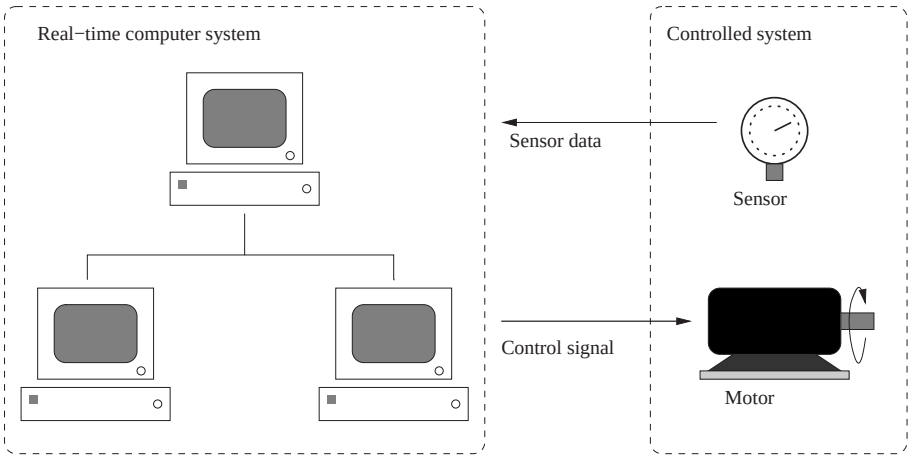


Figure 1.1: A real-time computer system used for control

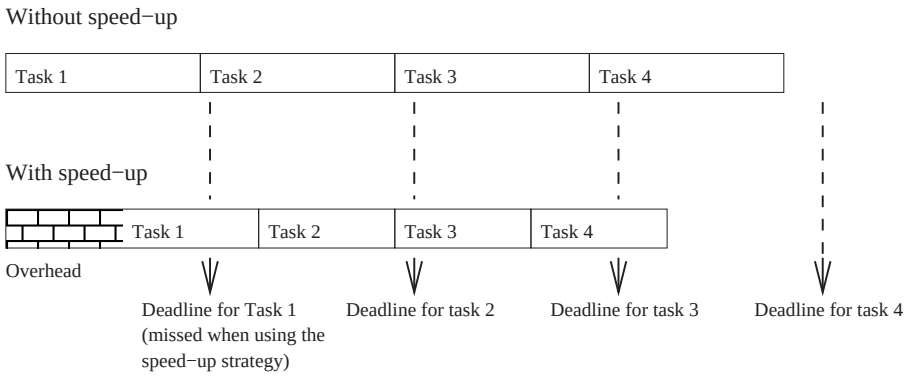


Figure 1.2: Timing when using a speed-up strategy that increases the average speed of the system. Because of the initial overhead, Task 1 misses its deadline.

operation of the control system due to a failure may be very costly, or it may even put human lives at risk. To fulfill the reliability requirements, it may be necessary to use a fault tolerance mechanism. The combination of real-time and fault tolerance is therefore an important study. If a fault tolerance mechanism enables the system to tolerate faults, but in doing so, causes a task to miss its deadline, it has introduced a new fault into the system. This way, some fault tolerance mechanisms may be unsuitable for the system, depending on the system's characteristics.

For real-time systems, it has been common to use active replication, running several instances of a task on several hardware nodes in parallel, as a means to achieve fault tolerance. If one of the replicas should fail, the affected replica is masked out, while the others continue to run. This will give a deterministic temporal behavior even if some of the replicas should fail, as the non-failed replicas will finish their execution normally. The possibility to compare results from the replicas can also function as a fault detection mechanism, increasing the range of faults that the mechanism may react to.

For general-purpose systems, fault tolerance mechanisms based on passive replication strategies have been common. In these systems, only one instance of the task, the primary, is active and running, while other instances are passive backups. If the active replica fails, one of the backups is activated, and the task, or unfinished parts of the task, is rerun using this backup. Because the detection of the failure, the activation and updating of the backup, and the rerun of the task is time-consuming, this kind of fault tolerance mechanism may cause problems if used in real-time systems.

1.1 Passive replication in real-time systems

While replication strategies based on active replication in many ways work better than passive replication strategies, active replication will also require more processing power. The extra hardware needed to run the instances of the task in parallel may cause the system to exceed limits to physical space, energy consumption, or cost. For some systems, a passive replication strategy will give "good enough" fault tolerance, and using an active fault tolerance strategy for these systems might be considered as "overengineering".

The requirements to timeliness and dependability for a system may be so high that a non-tolerant system becomes very costly or even impossible to make. Still, the same requirements may be on a level that does not justify using an active replication strategy, making the passive replication strategy the best choice.

A simple illustration of this is shown in figure 1.3.

To be able to decide if a passive replication strategy is a good solution for meeting the real-time system's reliability requirements, the timing of the system, especially when it is affected by a fault, must be analyzed.

Often, the system's tasks' worst-case execution times are used for the analysis. A simple approach is to see if a task's operational window is larger than the WCET for running both the primary and backup of the task as well as any overhead that may occur due to the fault handling.

Scheduling analysis in fault tolerant systems where failed tasks must be rerun, can give a probabilistic guarantee that all deadlines are met based on calculating a minimum tolerable distance between faults and the probability that faults do not occur closer than this distance [7, 12].

For some systems, the use of worst-case execution times in the analysis may be a bit too conservative. While the analysis may be used to decide whether or not the system will work if a fault occurs at the worst possible time, faults occurring at the worst possible time may be a very rare event, a fact that is not visible in the analysis results.

If a task rarely has a runtime near the worst case execution time, or if it is possible to detect failures and errors *during* execution, e.g. by the use of a watchdog mechanism for fault detection or by using checkpoints, a passive fault tolerance mechanism may improve the dependability of a system by a large degree, even if it does not tolerate faults in a worst case scenario. In figure 1.4, the timing of a task that needs to be rerun due to a fault is shown. While the task will miss its deadline in the worst case, it is also possible that the task is able to complete the rerun within the deadline for most fault occurrences. Depending on how reliable the system needs to be, the passive replication strategy may be "good enough" for the system.

Further, many methods of analysis assume that the system is only able to tolerate single faults. While some systems cannot tolerate several faults that occur close to each other, other systems may provide this tolerance, even if there are deadlines involved.

1.2 Using run-time distributions

Some, but not all, real-time systems may get a significant increase in reliability from using a passive replication strategy, even if the worst-case requirements are not fulfilled.

The runtimes of a system using a passive replication strategy may vary

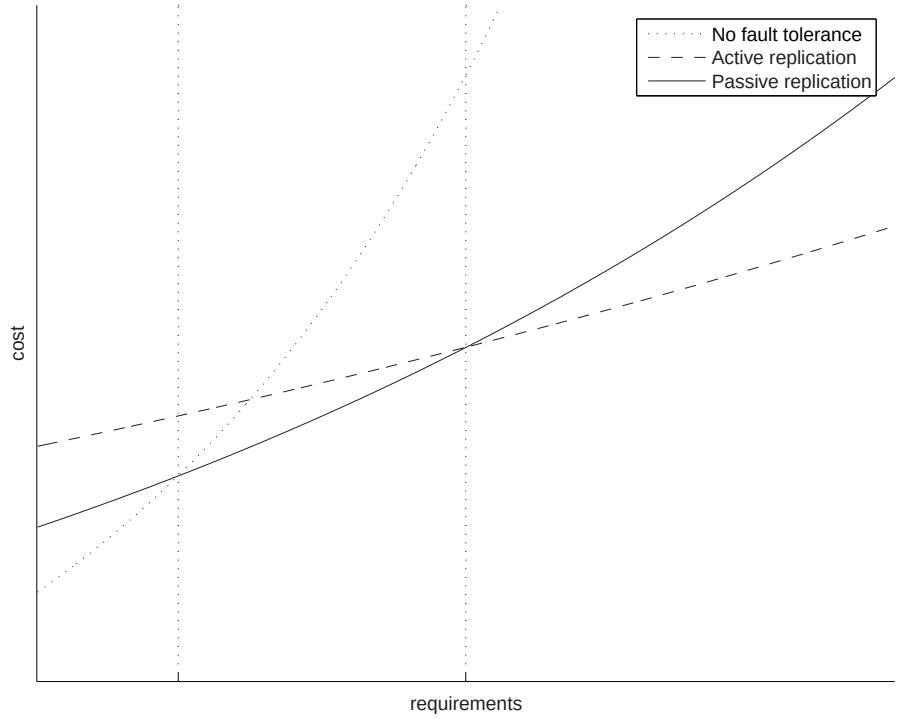


Figure 1.3: Cost vs. requirements for different replication strategies. For a system with very low requirements, a non-tolerant system may be the best solution, for a system with strict requirements, the active replication may be the best. Inbetween, the passive replication strategy may be the best solution.

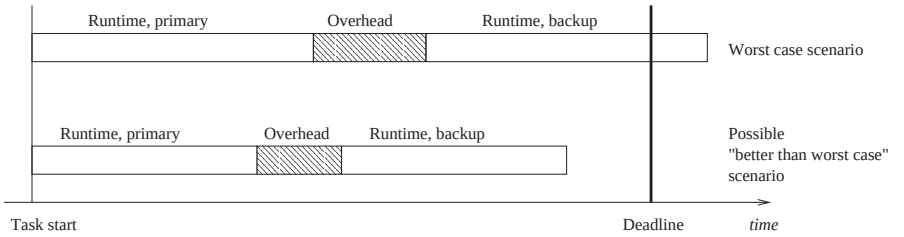


Figure 1.4: A passive replication strategy that will not work in a worst case scenario, might still improve the reliability of the system.

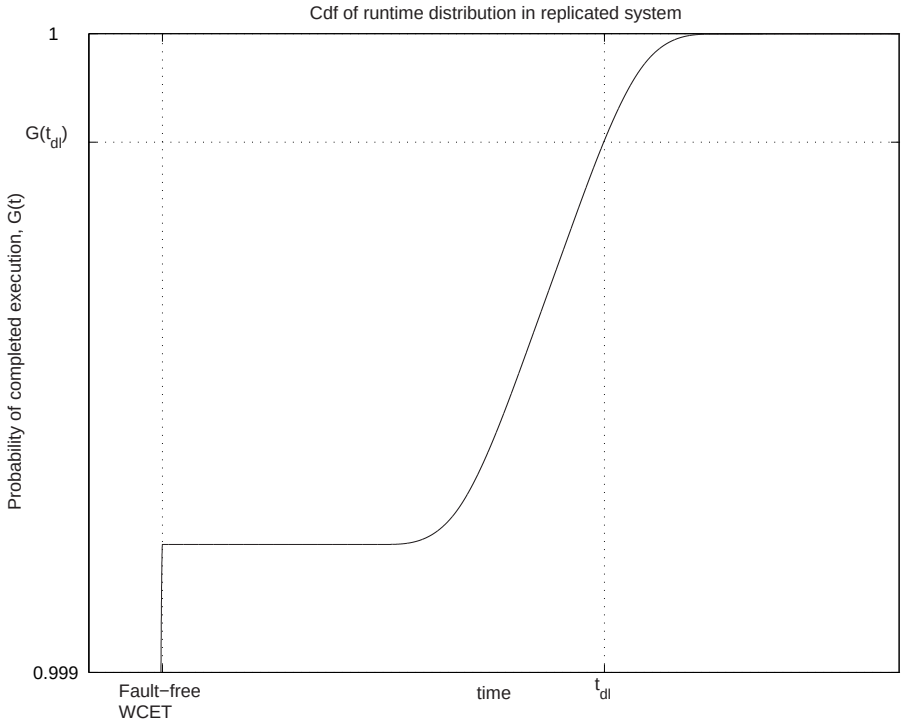


Figure 1.5: Using the run-time distribution to find the probability of a deadline miss.

greatly due to the extra time used during fault handling. By studying the *run-time distributions* of the tasks when they may be affected by faults, information on how suitable the replication strategies are for a given system can be found.

For a system with hard deadlines (i.e. the system does not tolerate that a task miss its deadlines), the probabilities of deadline misses can be found by using the task's run-time's cumulative distribution function, as shown in figure 1.5. If the deadline is at t_{dl} and the task's run-time distribution has a cdf of $G(t)$, the probability of a deadline miss, i.e. the probability that the task is not finished at t_{dl} will be $1 - G(t_{dl})$.

For a system with soft deadlines (i.e. the system tolerates deadline misses, but late results degrade the system's performance), lateness can be given a cost

function, which can be used together with the run-time distribution to find a mean cost for each execution [25].

Thus, by developing run-time distribution models, the background for reasoning about a replication strategy's suitability for a real-time system and for choosing the most optimal of several strategies, becomes more complete than when using only a worst-case execution time analysis.

1.3 Goals and contributions of the work

The goal of this work has been to develop *generic runtime distribution models for some common classes of passive replication systems*. This in order to have a tool to analyze the probabilities that a system may be able to tolerate faults and still have the time-critical tasks meet their deadlines.

There exist many different types of passive fault tolerant systems, fault detectors, fault models etc. Part of the work has been to investigate how different choices of fault detectors affect the run-time models. The effect the use of inhomogeneous replicas, i.e. systems where the different replicas perform the same task, but are not mere copies of each other, has on the model is also investigated. The choice of which models to derive is based on these investigations.

The runtime distribution models are derived using methods known from traffic theory, and are functions of distributions of the modeled system, like the fault-free run-time of the replicas, and the fault probabilities. A goal has been to present the derivation of the runtime models in a way that makes the same method easy to use when deriving runtime models for system types not presented in the work.

Some simple examples are given on the use of the runtime distribution models, and results from the derived models are compared to the results from discrete event system simulation.

Summary of the main contributions

- A developed argument for why some real-time systems may benefit from a passive replication fault tolerance strategy if a runtime distribution analysis is done in order to determine the suitability of this strategy.

- It has been shown how methods used for analysis in queuing and traffic theory can be adapted for the purpose of developing runtime distribution models for tasks in a passively replicated system.

- A developed argumentation for how the main differences between the mathematical model structures stem from whether homogeneous or inhomogeneous replication is used, and the fault detection methods that are used in the system. Furthermore, it is shown how many other differences between systems can be described using only different parameter distributions while keeping the structure of the mathematical models unchanged.
- Runtime distribution models have been developed for
  - Homogeneous and inhomogeneous systems using a watchdog mechanism for fault detection.
  - Homogeneous and inhomogeneous systems using a timeout mechanism for fault detection.
  - Homogeneous and inhomogeneous systems using acceptance tests for fault detection.
- Runtime models for some other system variants have also been developed together with an argumentation that the method used for developing the models can be used to develop runtime models for other systems that are not covered by this work directly.
- The application of the models has been demonstrated through examples, and the results have been compared to results from simulation.

Organization of the thesis

The thesis is organized as follows:

Chapter 2 gives a review of the theoretical background for the work.

Chapter 3 gives a textual description of the systems that are modelled.

Chapter 4 contains the derivation of the mathematical run-time models for the systems.

Chapter 5 contains examples on how the mathematical models can be used, and a description of the simulator that was used for comparison.

Chapter 6 contains discussions on this work as well as suggestions for future work.

Chapter 7 contains the conclusions to the thesis.

Appendices A– E provide the contents of papers [45, 46, 47, 49, 48] for easy reference. The layout of the papers and the section numbering have been changed to fit the format of the rest of the thesis, and reference numbers have been changed to refer to the bibliography at the end of the thesis. Apart from this content is the same as in the original papers.

Chapter 2

Dependable real-time systems

Because real-time computer systems are used for controlling other processes, the failure of the computer system may lead to the loss of control. This will, depending on the controlled system, often be very costly, and it may even lead to catastrophic results. Systems used for tasks where the consequences of failure can be severe need to be highly dependable: The probability of failure occurrences must very low, and if a failure occurs, the failure consequences should be the least malign possible. If it is feasible to get the system running again, it should be in the failed state only for a short period of time, and so on.

This chapter gives a brief overview of some dependability terminology and different fault tolerant mechanisms that are used later in this work. The chapter is not meant as an in-depth explanation of dependability theory.

2.1 System reliability

This section contains some brief background on dependability and definitions used in this work. Definitions of dependability attributes are given, as well as the definition of failures, errors and faults. Some common ways of classifying faults and failures are discussed, and, as this work focuses on real-time systems, a discussion on deadline failures and the hardness of deadlines is given.

2.1.1 Dependability attributes

Laprie and Kanoun [27] give a list of dependability attributes:

Availability: The readiness for usage.

Reliablilty: The continuity of service.

Safety: The nonoccurrence of catastrophic consequences on the environment.

Confidentiality: The nonoccurrence of the unauthorized disclosure of information.

Integrity: The nonoccurrence of improper alterations of information.

Maintainability: The ability to undergo repairs and evolutions.

This work focuses mainly on the system *reliability*, the probability of system failure and how passive redundancy changes the failure probability when there also are deadlines that the system's tasks must meet.

Some definitions that often are used in dependability theory [15, 39] are given below.

Time to failure

The time from when a component is put into operation until it fails is called the *time to failure*. This time is modelled as a random variable T, that has a cumulative distribution function

$$F(t) = \Pr[T \leq t] \tag{2.1}$$

and a probability density function

$$f(t) = \frac{d}{dt}F(t) = \lim_{\Delta t \to 0} \frac{\Pr[t < T \leq t + \Delta t]}{\Delta t} \tag{2.2}$$

For some systems, other variables may be more suitable than time, e.g. for a vehicle, it may be better to use the distance the vehicle has driven. For other systems, a discrete value may be the most suitable, like the number of times a component is used or the number of physical shocks it has endured.

Reliability

The reliability function (also called survivor function) of a component is defined as the probability that the component does not fail in the interval $\langle 0\ ,\ t]$.

The reliability function of a component is given by

$$R(t) = 1 - F(t) = \Pr[T > t] \tag{2.3}$$

As with the time to failure, other variables may be more suitable than time. For instance, a component that has a probability S of not failing each time it is used, reliability can be given as a function of the number of times n the component is used.

$$R(n) = S^n \tag{2.4}$$

Hazard function

The hazard function is defined as the probability that a component that is working at time t will fail in the time interval $\langle t, t + \Delta t]$ when $\Delta t \to 0$.

It can be shown that the hazard function is given by

$$z(t) = \frac{f(t)}{1 - F(t)} = \frac{f(t)}{R(t)} \tag{2.5}$$

Mean time to failure

The mean time to failure is the expected value of the time to failure T, defined by

$$\text{MTTF} = \text{E}[T] = \int_0^\infty t f(t) dt \tag{2.6}$$

Availability

If a component can be repaired or exchanged with a working component if it should fail, availability is a useful measure, given by the probability that the component is working at a time t. If the state of a component is given by the state variable $X(t)$ that is 1 if the component is working, and 0 if the component has failed, the availability function of the component is

$$A(t) = \Pr[X(t) = 1] \tag{2.7}$$

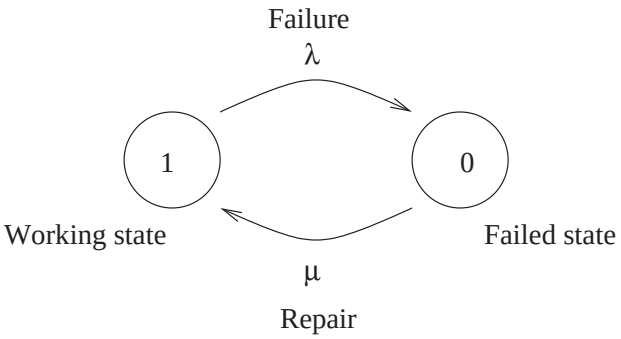


Figure 2.1: A simple model of a repairable component

If the uptimes of a repairable component is exponentially distributed, with a failure rate of λ, and the downtimes is exponentially distributed with a repair rate of μ, as shown in the model in figure 2.1, the availability of the system is given by

$$A(t) = \frac{\mu}{\lambda + \mu} + \frac{\lambda}{\lambda + \mu} e^{-(\lambda+\mu)t} \tag{2.8}$$

For many systems, the time-variant parts of the availability function will often "die out" over time. The availability function will often rapidly converge toward the *limiting availability*, which becomes a useful measure:

$$A = \lim_{t \to \infty} A(t) \tag{2.9}$$

Another useful measure is the average availability, the mean proportion of a time interval where the component has been available.

$$A_{av}(0, \tau) = \frac{1}{\tau} \int_0^\tau A(t) dt \tag{2.10}$$

When a system has been in service for a long time, the average availability can often be expressed as a function of the mean time to failure and the mean time to repair (MTTR).

$$A_{av} = \frac{\text{MTTF}}{\text{MTTF} + \text{MTTR}} \tag{2.11}$$

For many components, the average availability when τ approaches infinity, is the same as the limiting availability.

$$\lim_{\tau \to \infty} A_{av}(0, \tau) = A \tag{2.12}$$

For systems that can not be repaired, or systems where a failure will be catastrophic, reliability is often a better measurement than availability.

2.1.2 Faults, errors, and failures

If a system *fails*, i.e. stops working correctly, there must be a cause to the failure. The system may have been in an *erroneous* state, which itself has been caused by a *fault* in the system.

Definitions of failures, errors, and faults are given below [3].

Failure: Deviation of the delivered service from compliance with the specification. Transition from correct service delivery to incorrect service delivery.

Error: Part of a system state which is liable to lead to failure. Manifestation of a fault in a system.

Fault: Adjudged or hypothesized cause of an error. Error cause which is intended to be avoided or tolerated.

As listed in the definitions, faults may cause errors, and errors may lead to failure. A failure in one system may also be considered a fault in a system that depends on the system that failed.

For a system that consists of components, which themselves may be systems consisting of components, a failure in one of the components may cause an error in the system, and can thus be considered a fault in the system. This component failure can itself be caused by a failure in one of the component's subcomponents. In a fault-tolerant system that consists of replicated components, the fault tolerant mechanism's task is to hinder that the failure of one of the components propagates further, so the failure of the component does not cause the failure of the system as a whole.

As an example of faults, errors and failures, consider a server that runs some tasks for a client. Figure 2.2 shows what may happen if a bit in the server's memory is flipped due to radiation. The faults, errors and failures in this scenario are:

Fault – server: Radiation causes a bit to be inverted in the server's memory.

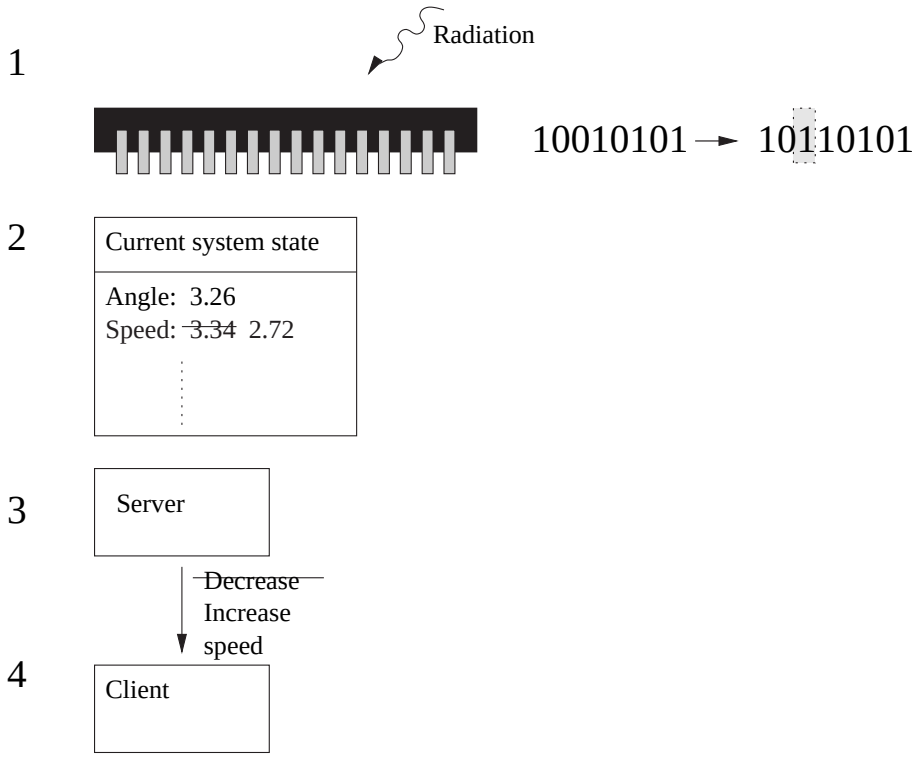


Figure 2.2: Fault (1), error (2) and failure (3) of a server. The server failure causes a fault in the server's client (4)

Error – server: The altered state of the server caused by the inverted bit.

Failure – server: Because of the error, the server delivers an incorrect result to the client.

Fault – client: The client gets an incorrect value from the server.

Further propagation: The fault causes an erroneous state in the client, which again may lead to the failure of the client. If there is a system depending on the client, the failure of the client can be viewed as a fault for this system, and so on.

2.1.3 Classification of faults and failures

To get a good overview, and thus a better understanding, of how different faults and failures affect the system, it is useful to group faults and failures in different classes.

Faults are often classified after the cause of the faults, whether the fault is introduced during the design or manufacturing of the component or after the component is put into operation, or whether the faults are permanent or disappear by themselves. Failures are often classified after the way that the component fails, and the severity of the failure. If the faults or failures occur independently of each other or if they are correlated is also an important way of classifying the faults and failures when modeling.

Cause of faults

A way of classifying faults is after the cause of the fault. For a physical system, the wear of a structure or physical damage to a component may be the cause of a failure. Errors made by programmers of software system are not uncommon, and some of these can be difficult to detect before the software is put into operation. A computer system consists of both hardware and software, and is subject to both physical and logical faults.

Some common classes of failure causes in a computer system are:

Design faults: Mistakes during the design of the system and/or components that the system consists of, so that the system does not work as intended. Design faults may occur in both hardware and software. Because of the faults' nature, all components based on the faulty design are affected.

Manufacturing faults: Faults may be introduced during the manufacturing of a component. This can be caused by both human errors and the failure of the tools and manufacturing equipment that is used. The same faults may be present in several components of the same manufacture.

Wear and aging: The physical, i.e. hardware, part of a computer system is, as other physical systems, subject to wear and aging of components, which can cause faults. The faults are usually permanent, i.e., present until repaired, or intermittent, i.e. the effect of the faults "come and go".

User faults: The "wrong use" of the system is a source of many faults, both in software and hardware. Deleting or overwriting the wrong files in a computer system, or manually turning off the power of equipment that needs to be on, are examples of user faults.

Environment: Radiation or electromagnetic noise may cause contents of memory or transmissions to change, and overheating of a system may cause unwanted behaviour. More severe occurrences like fires or floods may cause permanent damage to components, and several components may be damaged by the same occurrence.

Temporal behavior of faults

Another important classification of faults is the temporal behaviour of the faults, i.e. whether the faults are present until the affected component is repaired or if they disappear by themselves. Usually, three classes that describe the temporal behaviour of faults are used, as shown in figure 2.3.

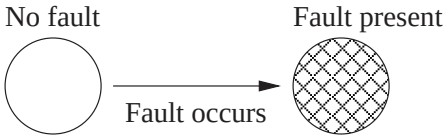
Permanent fault: When a permanent fault occurs, the fault will stay until the affected component is repaired or replaced.

Transient fault: A transient fault will either occur only once or for a short period of time, then it will disappear.

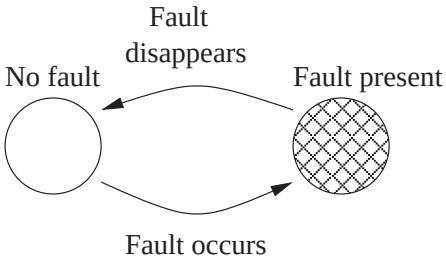
Intermittent fault: An intermittent fault is a fault that, when it occurs, will change between an *active* state where it may affect the system (i.e. cause errors) and a *passive* state where it does not affect the system.

An example of a permanent fault is the physical damage of a component because of an environmental effect or due to wear. As long as the damaged component is in the system, the fault is present, and the fault will not disappear

Permanent fault



Transient fault



Intermittent fault

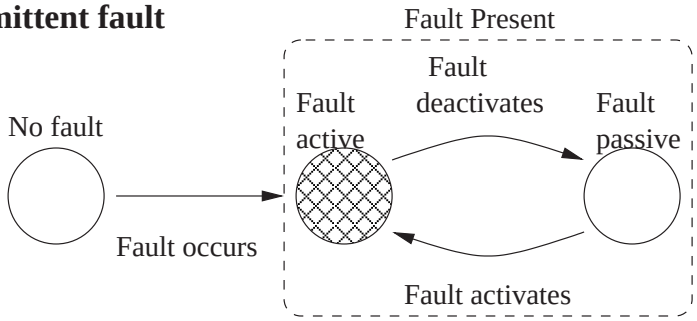


Figure 2.3: Temporal behavior of faults.

by itself. Design or programming errors may also result in permanent faults in a software system.

Radiation causing bits in a component to be flipped, or temporary electromagnetic noise[1] disrupting a transmission, will result in transient faults. These faults will disappear after occurring, and can usually not be repaired. User errors and short-term environmental disturbances may also lead to transient faults.

A loose wire that changes between being "connected" and "disconnected", or components that change properties as temperature changes, are examples of intermittent faults. These faults may seem to disappear by themselves (as transient faults), but the same faults recur at a later time. Poor synchronization leading to race conditions in software components may also appear as intermittent faults.

Correlation between faults

When modeling faults, the correlation between fault occurrences may be an important consideration. Several faults often occur as the result of a common cause, a failure of a component may propagate as faults in several other parts of the system, and environmental effects, like electromagnetic noise, may cause several faults that occur nearly simultaneously.

Software faults are seldom independent. If copies of the same component are used in a redundant system, a fault in one component is present in the other components. Thus, in a redundant system using copies of the same component, a failure due to a software fault may affect all the components. Even when using multiversion redundancy, assumptions of independent failure may be wrong [9], as faults present in the different versions are more likely to coincide than in a purely independent model.

Often, faults are modelled as independent, with no correlation between the fault occurrences. Using a constant hazard function and a negative exponential distributed time between faults makes each fault occurrence in the model independent on the previous, and this fault behavior model often makes the mathematical modeling easier. This model is often considered a "good enough" approximation of fault behaviour, even if the real fault behavior may differ from this.

[1] For the electromagnetic noise scenario, the fault may also be considered a permanent or intermittent fault, e.g. if a signal transmission line is placed in the vicinity permanent source of electromagnetic noise, the fault will likely recur, and it can be "repaired" by shielding or moving either the affected equipment or the noise source.

In most of this work, an independent fault model, where faults occur as if generated by a poisson process, is used. As mentioned earlier, this makes the mathematical modeling easier and the models "cleaner", while the models still can be considered a "good enough" approximation of the actual timing behavior of a redundant component group.

Failure modes

There are several ways that a component can fail, ranging from crashes where the component simply stops working to situations where the component delivers incorrect results. Different failure modes may have different effect on the system that the component is a part of, and the methods used to detect that a component has failed will also only cover certain failures. It is therefore useful to classify failure modes. Some of the most common classes used are [39, 32]:

Notified failure: The failure causes the component to stop all execution, and the other parts of the system are immediately notified of the failure. This failure mode is in many ways the "ideal" way a component may fail, as the detection of the failure is already given.

Stop failure: The failure causes the component to stop all execution, but the other components are not notified of the failure.

Omission failure: A scheduled message is not sent, not transmitted, or not received.

Timing failure: Messages arrive too late, too early, or out of order. This class of failure modes is important to consider in real-time systems. Deadline failures, i.e. the situation where results from a computation arrive after a given deadline, will be discussed further in section 2.1.4.

Value failure: The value of a response from a component does not comply with the system specification. Value failures are often subdivided into consistent failures, where several users get the same incorrect results, and inconsistent failures where different users do not get the same results. Inconsistent failures are also called *Byzantine failures*, after the "Byzantine generals problem"[26, 36]

The *Byzantine failures* is often considered the worst case. The inconsistent behaviour can make it difficult to determine if the failure is present, as the failure detector may see an acceptable result while the system that is using the result

gets an unacceptable result, and if there are several redundant components, it can be difficult to determine which of the components that has failed. It has been argued [19] that these failures are a very rare occurrence compared to other failure modes.

Criticality of failures

It is also common to classify failures after the consequences a failure has. Some failure modes may be considered less malign than others. In a typical database system, a stop failure would be better than a failure causing the corruption of data. In a system controlling an industrial process, a failure causing a shutdown of the process may be better than a failure causing the process to run out of control.

One of the goals in making dependable systems is to make the system so that if a failure occurs, the failure mode should be the least malign possible.

The severity of failures is important when deciding the reliability requirements of a system. If the failure of a system leads to "minor annoyances", the failures should not occur "too often", and the required reliability may be achieved without using any fault tolerance. If the failure may lead to catastrophic results, reliability requirements are very strict, and much effort must be put into ensuring that the system meets these requirements.

2.1.4 Deadline faults and failures

In a real-time system, an important class of failures is the failure of a task to meet its *deadline*, i.e. for a task to work correctly, it does not only have to produce correct results, but also to produce these results at the correct time.

In this work, the term **deadline failure** is used to describe the failure of a task to meet its deadline. As explained in 2.1.2 this failure will be considered as a fault in the system relying on the results of the task, and may cause errors and failures in this system.

In literature on real-time systems [24, 8, 41], it is normal to classify the deadlines in **hard** and **soft** after the system's ability to tolerate the faults caused by deadline misses. Systems with soft deadlines are able to tolerate missed deadlines with some degradation in the provided service, while in systems with hard deadlines, missed deadlines will lead to a system failure. Furthermore, the class of soft deadlines is sometimes divided further into **soft** and **firm**, where the results from a task that misses a soft deadline may still be of some value

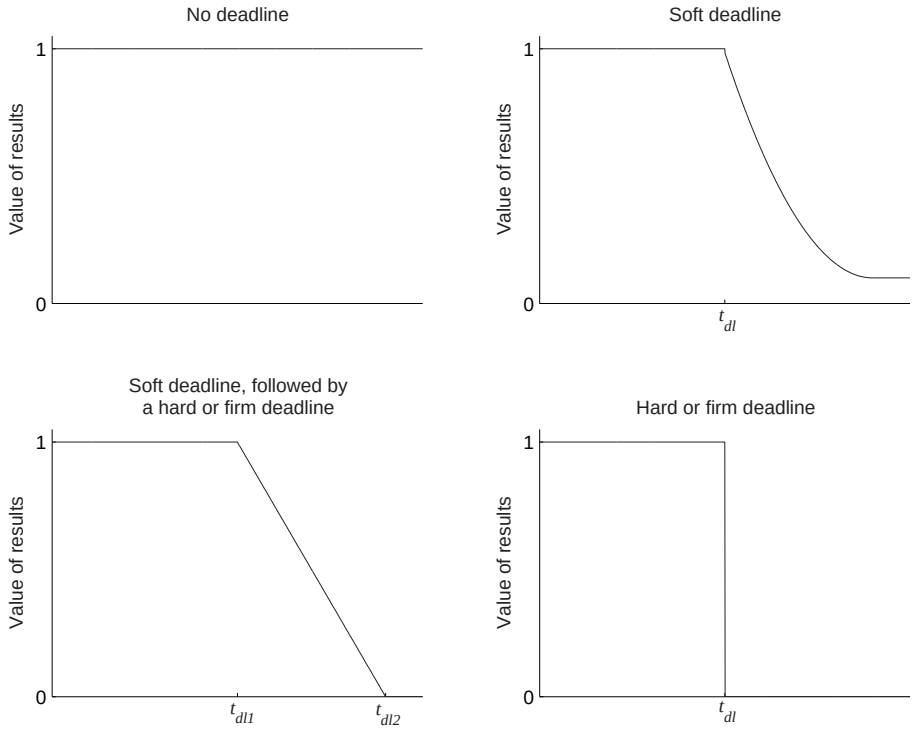


Figure 2.4: The value of a task's results arriving before and after the deadline for different kinds of deadlines.

to the system, while the results from a task that misses a firm deadline are valueless.

In this work, the following definitions are used for different deadlines:

Hard: The results from a task are valueless if they arrive after the deadline. Furthermore, deadline misses will cause failure of the system, and can not be tolerated.

Soft: The value of the results from a task decreases if they arrive after the deadline, but they do not necessarily become valueless. The system will, to a certain extent, tolerate missed deadlines.

Firm: The results from a task are valueless if they arrive after the deadline, however, the system is able to tolerate the missed deadline, i.e., late results can be discarded without causing the system to fail.

The value of the results as a function of time for different kinds of systems is shown in figure 2.4.

Note that it is not unusual that a soft deadline is followed by a firm deadline (i.e. while the results may have some value after the soft deadline, they may eventually become valueless) or hard deadline (i.e. while some lateness may be tolerated, very late results may cause the system to fail). This is shown in the bottom left graph, where the time t_{dl1} is the soft deadline where the results begins to lose their value, and the time t_{dl2} is the firm or hard deadline, where the results are valueless.

While systems tolerate that soft and firm deadlines are missed "once in a while", several deadline misses in a short timeframe may cause system failure.

2.2 Software fault-tolerant methods

To be able to create systems with high reliability requirements, *fault tolerance* is necessary. While a first approach to high reliability is *fault avoidance*, i.e. the use of methods that lessens the probability of introducing the faults in the first place, creating completely fault-free systems is very costly, and may be infeasible.

Fault tolerance are methods used to allow the system to operate in the presence of faults, i.e. hinder that faults in the system don't propagate to a situation where the system itself fails. This means that the system itself must be able to detect errors in and failures of its components, and do some corrective action to hinder propagation.

2.2.1 Fault tolerant structures

For a system to be able to be fault tolerant, some redundancy in the system is needed. Redundancy can be classified using three dimensions, *time*, *resource*, and *information* redundancy. A specific fault tolerant strategy may have redundancy in one, two or all three dimensions.

Resource redundancy

In fault tolerant structures using resource redundancy, extra resources are added to the system, so that the system may continue its service even if one part of the system fails.

In a computer system, this is typically done by adding more hardware. For example, a task can be executed on several nodes, usually concurrently. If one instance of a task or a node should fail, the others could still be able to execute properly. If the fault detection coverage is perfect, the system should be able to tolerate the failure of all but one of the executions.

Another example is the use of several redundant disks. If one of the disks should fail so it becomes unavailable or the data it stores is corrupted, the other disks may still provided the needed service. A third example is to provide several possible routes through a network to ensure connection if one link should fail.

In a real-time system, resource redundancy is usually preferred. If several instances of a task is computed concurrently (usually called *active replication*), all fault-free instances of the task should be able to finish at their normal execution times, thus the risk of the task missing the deadline is lessened. A timeout mechanism can be implemented to hinder that the results from the group of replicas is delayed beyond the deadline even if some of the replicas finish late.

Concurrent execution of several replicas also gives an advantage in fault detection. Because several results from the task are present, the results can be voted upon, and replicas delivering results that differ too much from the others can be considered faulty and masked out of the group. Voting is discussed further in section 2.2.2.

Even if the execution of the replicas is not concurrent, the time to correct a fault could be reduced if a hot backup is already prescheduled on another node [12], thus combining resource and time redundancy. If the primary replica succeeds its task, the scheduled slot for the backup is freed for other purposes.

Executing several instances of a task on several nodes also helps the system to tolerate permanent hardware faults. For larger and safety critical systems, the different nodes may be spread over several rooms or compartments, thus minimizing the risk of a common environmental cause (like a fire or other accident) to cause failures in all nodes at the same time.

The main drawback of a system using resource redundancy is the extra resources required. Extra processing power is needed to execute the different replicas, which will increase cost, power consumption and space needed for the system. For many large systems, these requirements are easily met, but in smaller systems, like embedded systems, the feasibility of using more resources

may be limited.

Time redundancy

If a failure is detected in a component, the *time redundancy* approach is to redo the failed operation, thus using extra time to correct the failures. A typical example of this approach is the retransmission of erroneous or lost packets in a networking system.

In a computer system, one use of time redundancy is to replicate components in a way that only one of the components is active, while the rest are passive. If the active component (or primary) should fail, the failed task is rerun on one of the backups, and this backup continues to provide service to the system as the new primary. These systems are called *passive replication* systems or *primary – backup* systems.

Unlike the active replication systems, where all replicas run the task and consequently get their states updated, the backups in a passive replication system are not automatically up to date. Because of this, a mechanism that keeps consistency between the replicas is needed. Different strategies for updating the backups exist. In systems where the backups are *cold*, backups are not updated during normal execution, i.e. backups are updated only in failure situations, when a backup becomes a primary. With *warm* backups, the state of the replicas are updated regularly, causing more overhead than with the cold strategy during normal execution but less overhead during failure situations, and with *hot* backups, the backups are updated as often as necessary to make them ready to take over as the primary at any instant. For a *strong consistency* system, a backup that takes over as the primary must be updated to the exact same state as the former primary had before it failed, while in a *weak consistency* system, the backup must be updated to a state sufficiently near the state of the former primary, which can lessen the time needed for updating.

The system state, or parts of the state, can be saved at *checkpoints* during the execution of a task. If an error is detected, the execution rolls back to a previous checkpoint, retrieves the stored state, and retries the execution from the checkpoint.

Depending on the system, the criteria for optimization of checkpoint placement may vary[50, 11, 25]. Using many checkpoints results in higher overhead during normal operation, but shortens the time needed to recover from a failure situation. For a typical general-purpose system, checkpoints are typically placed in ways that optimize average performance, while in real-time system, a minimization of lateness due to failure recovery is often the criteria, while the

average performance is less important.

If a checkpointed process communicates with other processes, a rollback may cause rollbacks in other parts of the system, and checkpoints ought to be placed in a manner that minimizes the probability and size of rollback chain reactions.

As systems using time redundancy are the main topic of this work, a more in-depth view of time-redundant systems is presented in chapter 3.

Information redundancy

Information redundancy is based on providing more information than is strictly needed, so that the needed information can be extracted even if part of the total provided information is erroneous or missing.

In its simplest form, this can be done by providing the same information several times, i.e. transmitting messages several times or creating several backups of data, so that the data is available even if one copy is destroyed or corrupted. However, to detect and correct minor errors that occur during transmission or storage, using several copies is too time and resource consuming for most systems.

The use of error correcting codes [13, 38, 6] is an effective way to use information redundancy to detect and correct errors in transmission and storage of data. By adding extra information to the data when transmitting or storing it, many of the errors that occur during transmission or storage can be corrected.

Another way of using information redundancy is to provide several versions of a program in a replication system. This is discussed further in 2.2.3.

2.2.2 Fault detection methods

If a fault tolerant system shall be able to correct faults, it must also be able to detect that a fault has occurred, i.e. it must have mechanisms for detecting that components or data the system consists of have either failed or are erroneous. There are several ways this can be done [20].

Notified Failures

As noted in section 2.1.3, the notified failure mode causes the rest of the system to be aware of the failure when it occurs. These failures will thus be detected regardless of the use of fault detectors. Typical examples are failures causing exceptions caught by the operating system, or exceptions raised by the failed component itself.

Watchdogs

A way of detecting stop failures is to make the components create "I'm alive" signals while it is operating, and monitoring these signals.

There are typically two different mechanisms for when these signals are created, they can be created as "heartbeats", i.e. the component will generate signals automatically as long as it is in operation, or they can be generated by request from the fault monitor, i.e. the fault monitor asks the component if it is still alive. If no there is no "I'm alive" signal registered for a reasonable time, the component is suspected of having suffered from a stop failure.

The fault monitoring described in the passive replication schemes of Fault Tolerant CORBA [35] is an example of a watchdog fault detection mechanism.

Timeout

Another time-based fault detection mechanism is to set a timeout for the results of computations, i.e. if a component does not deliver its results in what is considered a reasonable time, the component is considered failed. This detection mechanism will detect omission and some timing failures in addition to stop failures.

In a system with active replication, this method can be used in addition to voting, so omissions or timing failures in one of the components do not delay the results from the replicated group as a whole.

In systems where extra time is used to improve already acceptable results (e.g. imprecise computation systems), timeouts can be used to break off the result improvement so that the task will meet its deadline. For systems using an active imprecise replica in addition to a main precise replica, a timeout may be used to trigger the use of the results from the imprecise replica [8, 22].

The use of timeouts will detect stop failures, omission failures and some timing failures. To make sure that only "real" failures are detected, the timeout value has to be set to the same or later point of time than the task's worst-case execution time. This means that failures will be detected after the failed component should have sent its results. This method may therefore give less time for corrective action than the watchdog method.

Voting

In a system where several active replicas are used to produce the same results, comparison of the results, or *voting*, can be used to detect value failures in components.

Several voting strategies exist [29]. One of the most common voting strategies is *majority voting*, where the majority of the results must be sufficiently equal[2] to be used. Components producing results that differ from the majority are usually considered as failed and are masked out. This strategy will fail if there is no majority or if the majority of the results is incorrect.

The requirements of the majority voter can be relaxed, so that the group with the highest number of sufficiently equal results is chosen, whether this group forms the majority or not. This voting strategy is called *plurality voting*. If there is no distinct plurality, or if the plurality is incorrect, this voting strategy will fail.

In some systems, results cannot be expected to be equal, even when they are correct. In a system where several redundant sensors are used, the measured values can often be expected to differ. The sensor types may be based on different measurement principles, or they may be distributed physically, thus measuring local variations in the process. Also, in many systems, it can be expected that noise will affect the results. A *median voter* uses the median result from the components, and will thus be correct if the majority is correct, even if the results are not equal. A *weighted average voter* produces a new result from the weighted average of the results. Components that produce results that differ too much from the rest can be considered as failed and are masked out.

It should be noted that some faults are introduced during the design of the components, and may thus be present in several, or all, of the replicas used to produce the results. If the replicas all produce equal, but incorrect results, none of the voting strategies will detect this.

Error-detecting code

In data storage and transmission, extra information can be stored with the data to detect if the data has been corrupted.

A very common and simple method to detect errors in this way is using an extra parity bit. For each group of bits an extra bit is added, with a value that makes the total number of "ones" in the group even (for even parity) or odd (for odd parity). Thus, single bit errors in the group will be detected.

Cyclic redundancy code (CRC) is used for detecting errors in larger amounts of data, and is typically able to detect error bursts of a given length (typically a function of the length of the code used). Many different cyclic redundancy

[2]In many systems, results from the different replicas cannot be expected to be exactly equal, thus the term *sufficiently equal* is used.

codes exist. The parity bit, described above, can be considered a very simple form of the CRC.

In some applications, error detection is not enough, and as discussed in section in 2.2.1, it is also possible to use extra information to be able to correct errors in the data. This is useful for real-time systems, as retransmission may be too time consuming, for systems that have no feedback from the receiver to the transmitter, or in systems where it is impractical for the transmitter to retransmit the data.

Error detection codes is used to detect value failures if these failures are caused by errors in transmission or corruption of storage, but can only be used to detect and correct errors that occurred between the coding and decoding of the data. Thus errors occurring before coding (or after decoding) will not be detected by these methods.

Acceptance test

If there is only a single active replica that is used to produce results in the system, ordinary voting is not possible. It is, however, often possible to check if the results are reasonable by using an *acceptance test* or reasonability check [37].

An acceptance check may control if the results from the computation conforms to some of its specifications, but it is often not able to verify the correctness of the results.

To create an acceptance test, there must exist a way to differ between "acceptable" and "unacceptable" results. A test that gives "false positives", i.e. marks a result as faulty even if it is correct, may create more problems than it solves.

A typical acceptance test may be to compute the inverse of the task's computation and compare this to the task's input parameters, an approach that can be useful if the inverse exists and is relatively fast to compute. Other tests may be to check if the results have some necessary properties based on the input parameters (e.g. if the task sorts a list, check if the sorted output has the same number of elements as the unsorted input), or that they fulfill an easy to verify part of the specification (e.g. check if the output of the list sorter is sorted).

An example of using tests can be found in [33], where several programs creating the shortest possible palindromes from the input strings were tested. Various tests checked if the output formats where as specified, if the results actually were palindromes and that they were based on the input string, etc.

The acceptance test detects some value failures, as long as the failure causes the results to be "unacceptable" so they do not pass the test. Erroneous results

may still "pass" the test if the results seem acceptable.

Acceptance test is used to detect value failures in a way that uses less resources than the typical voting mechanism, as there is no need for several active replicas.

2.2.3 Homogeneity of replicas

Many replication systems are based on using copies of the same component as replicas, which for most systems gives a reasonable protection against failures. Most faults that affect one of the components will usually be tolerated even if all the replicas are exact copies of the affected component.

If the faults are introduced in the design phase, however, using copies of the same component in all replicas means that the same faults are present in all replicas. If a failure in one replica is caused by a design fault, it is a high probability that all the replicas will fail due to the same fault. Using components that have different design, but perform the same tasks, can thus improve the system's dependability further.

In this work, the term *homogeneous system* is used for replication systems where the components that the system consist of are of the same design, while an *inhomogeneous system* is a system where the replicas[3] differ in the internal design. The components of a heterogeneous system are usually the same from a "black-box" perspective.

Homogeneous fault tolerant systems

Using homogeneous replicas is, of course, the simplest approach for creating a replicated system. A common design for all the replicas makes both development and maintenance easier and less costly, while still providing tolerance for a wide range of faults.

A major problem with using homogeneous replicas is that faults that are introduced in the design of a component are common for all replicas of the component. If there is a fault introduced during the implementation of a software component, all copies of the component will have the same fault, and if one copy fails, the others are likely to fail in the same way.

[3]For an inhomogeneous system, "alternate" would be a better term than "replica". In this work, "replica" is used for components in both homogeneous and inhomogeneous systems.

Inhomogeneous fault tolerant systems

As some faults may be introduced in the design or implementation process of the components, it is sometimes useful to have a system where the redundant components are of different design, so that if one of the components fail due to a design fault, the others may still work properly as it is unlikely that the different designs have the same faults.

A well known approach in software replication is the N-version programming [2], using several active replicas of different design. The results from the replicas are voted upon, and replicas that produce results that differ from the others are suspected of failure and can be masked out.

In the description of recovery blocks [37], a block consists of several alternate ways of solving a problem. If the results from the first alternate fail an acceptance test, the next alternate in the block is tried out.

In real-time systems, a backup component that is able to provide acceptable, but not necessarily very good, results fast can be useful. If the slower, but more precise, components should fail, using a fast, imprecise backup may result in a higher probability that the deadlines are met. The use of imprecise backups is discussed below.

Imprecise backups

In some real-time systems, the timeliness of the results may be of greater importance than the precision of the results. In some systems, getting occasional results that are suboptimal may be acceptable. *Imprecise computation* [28] is a computing technique that aim for increased timeliness at the cost of precision of the results.

Typically, a task consists of a mandatory part that gives an acceptable (but still improvable) result and one or more optional parts that refine this result. As long as the mandatory part is completed, the optional parts can be omitted if there is insufficient time to complete them.

Another way to utilize imprecise results is by using imprecise backup replicas. If a task can be solved using either a slow method giving precise results, or a fast method giving imprecise results, the imprecise method can be used as a backup solution.

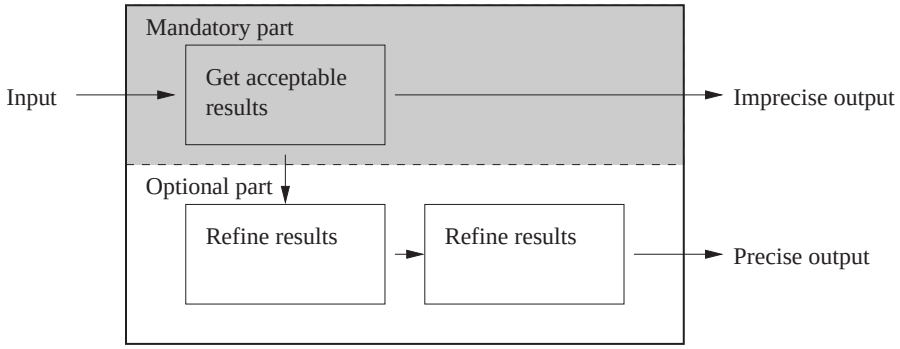
In an active replication system, the precise and imprecise results are computed simultaneously. If the precise results are ready within deadline, these results are used, otherwise, the imprecise results are used [8, 22]. This way, the deadline of the precise computation can be changed from a hard to a firm

deadline; as long as the computation of the imprecise results always meet the deadline, the results from the precise computation can be discarded if the deadline is missed.

For a passive replication system, using the imprecise method as a backup can decrease the time used for the rerun and thus increase the probability for getting results within the deadline. In some systems, it may also be possible to activate the imprecise backup (i.e. changing from a passive to an active replication system), if the precise replica has not finished its computation when the deadline draws near [14].

Figure 2.5 shows two ways of using imprecise computation. In the mandatory and optional parts system, the task is divided into two or more parts: A mandatory part that will give acceptable (i.e. good enough to not cause a failure, but still improvable) results, and one or more optional part that will improve the precision of the results. If there is no time to complete the optional parts (or if the optional part fails), the results from the mandatory part are used. In the precise primary and imprecise backup system, there are two methods of solving the task, the precise method is normally used, but in extraordinary situations (e.g. failure of the primary), the imprecise backup is used.

a. Method consisting of mandatory and optional parts



b. Precise primary method and imprecise backup method

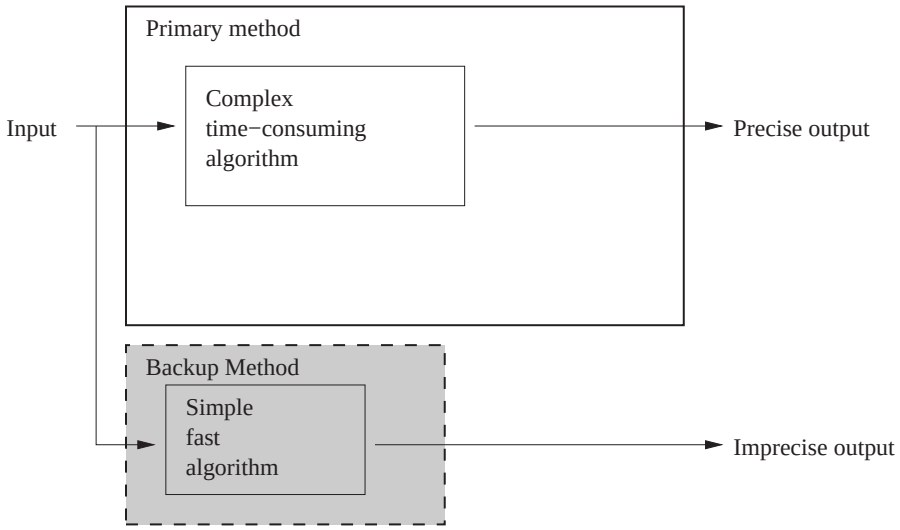


Figure 2.5: Two ways of using imprecise computations

Chapter 3

Fault tolerant mechanisms using time redundancy

As described in chapter 2, systems using time redundancy strategies for fault tolerance are the kind of systems that are most likely to experience problems when there are deadlines involved, and for that reason, these systems are the main focus of this work.

In this chapter, some basic fault tolerance mechanisms using passive replication are described further, with focus on how different structures and strategies will affect the mathematical models derived later in this work.

3.1 Description of the modelled systems

The classes of systems that are modelled can be described as servers using passive replication mechanisms to achieve fault tolerance. In these systems, there is one active primary object and several passive backup objects that either are pure replicas of the primary, or alternates that perform the same tasks as the primary.

In this section, the basic system structures of passive replication systems are described as well as a basic description of the timing of the systems when a fault occurs.

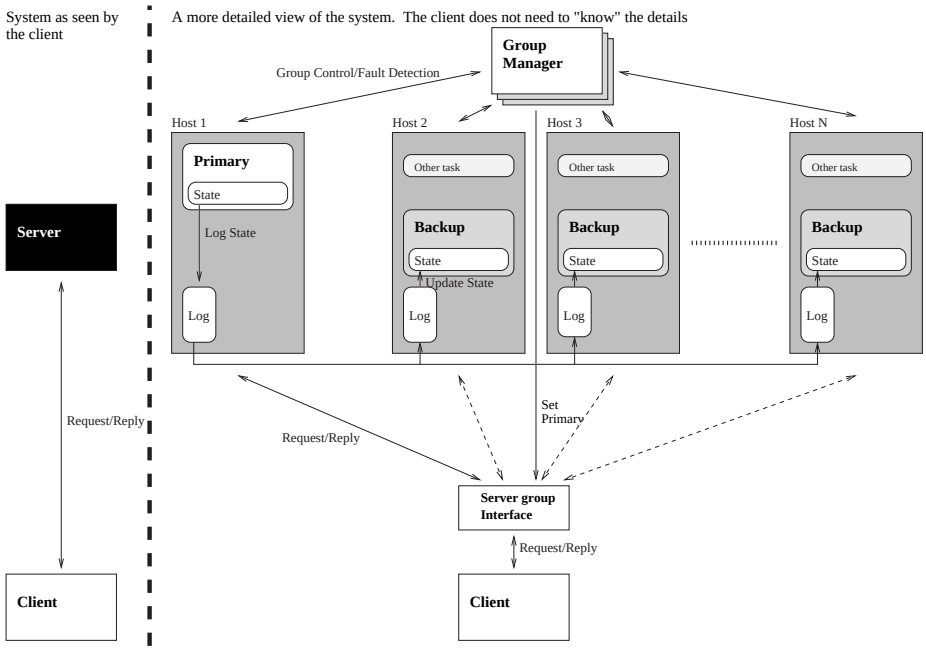


Figure 3.1: The mechanisms needed for fault tolerance are usually hidden from the client

3.1.1 System structure

In this section, some different strategies in the physical distribution of the replicas, keeping the consistency between the replicas and detecting faults are discussed, as well as if the choice of different strategies will change the mathematical equations or the distributions used in the equation.

The system as seen from the client

The client does not necessarily need to "know" the structure, physical distribution or the fault tolerance mechanisms of the server, other than the interface. Also, the occurrence of faults, as long as they are tolerated, may also be transparent to the client, i.e. the server, including its fault tolerance mechanism can be seen as a black box from the client, as shown in figure 3.1.

The client will thus not experience the faults that occur to the server, unless

these faults lead to a server failure. This happens when the fault tolerance mechanism is not able to detect or correct the fault, or when the time used for fault tolerance exceeds what is acceptable in a real-time system.

The server

In addition to the replicas of the server objects, a passively replicated server will typically have some functions for managing the replicas, like selecting the replica that shall function as the primary, instantiate new replicas if needed, keep track of which nodes that are non-faulty and reachable and keep consistency between the replicas. Also, mechanisms for fault detection are necessary. Some of, or all of these management functions may be replicated themselves, e.g. as a part of each of the server replicas.

Physical distribution of the server

The different replicas, as well as the components for managing the fault tolerance mechanisms are typically distributed over several nodes, using the nodes hosting the backups primarily for other tasks. For some systems, it may be possible or desirable to have several parts of the redundant system on a single node.

The physical distribution of the system will affect the system's ability to tolerate faults, as permanent faults will typically not be tolerated in a single-node system. The timing distributions for detection and correction of faults, as well as fault rates, will also be affected by the physical structure of the server.

While the range of tolerated faults and many of the parameters used in the mathematical model are affected by the physical structure of the server, the mathematical model itself is unchanged, as the activities of detecting a fault, reacting to the fault and making one of the backups the new primary, and rerunning the task that failed on the new primary are done in the same order regardless of the system's physical structure and the time distribution for these activities.

Consistency between replicas

To ensure consistency, the state of the primary is logged. If a failure is detected in the primary object, one of the backup objects is updated, if necessary, to the primary's state before the failure. For some systems, it is not necessary to update the new primary to the exact state the failed primary had. Instead, a state sufficiently near the primary's state (i.e. weak consistency) may be "good

enough", thus making it possible to save some time in the updating process [52].

If inhomogeneous backups are used, the backup is updated to the state corresponding to the former primary's state. Conversion of the states between the different kinds of replicas is either done as a part of the logging process, creating extra overhead during normal operation, or as a part of the updating process, making fault correction take more time.

The strategies used for logging the state of the primary and updating the backups may vary greatly, resulting in varying degrees of overhead during normal operation and during fault correction. Logging may be incremental, i.e. only changes to the state are logged, or by taking full snapshots of the primary's state, i.e. the complete state of the primary is logged, or a combination where a snapshot is taken, then a limited number of changes are logged, before a new snapshot is taken. In the same manner, the updating of the backups may be only in fault situations, each time the primary completes a transaction successfully, or at regular intervals, typically as full snapshots of the primary's state is taken.

For instance, logging only incremental changes in the primary, combined with only updating the backups when needed, will create little overhead during normal operation, while fault correction will take long time. On the other hand, mirroring the primary's state directly in the backups after each transaction will create much overhead during normal operation, but fault correction will be fast.

While the strategies chosen to log and update the backups will greatly affect some of the distributions used in the models, like the fault correction distribution, they have no direct impact on the structure of the models that are derived in this work.

Fault detector

As the system needs to know when to start the fault correction, it needs to be aware of when the active replica has suffered from a failure. The fault detector is therefore an important part of the passive replication system.

Of the fault detection methods discussed in section 2.2.2, the following are suitable for a passive replication system:

Watchdog: There is regular communication between the active object and the fault detector, so the fault detector gets messages indicating that the active object "is alive". If the fault detector does not get a message in what is considered a reasonable time, it is assumed that the active object

has failed. This fault detection mechanism will thus be able to detect stop failures.

Timeout: A maximum time is set for the task to finish its execution. If the task is not finished within this time limit, it is assumed that the active object has failed. In addition to stop failures, omission failures will be detected by this mechanism.

Acceptance test: The results (or partial results) from the task are checked to see if they meet a set of criteria that all acceptable results must meet. Many value failures can be detected with this mechanism.

The time between a fault occurrence and the detection of the fault is greatly dependent on the fault detection mechanism, and the different mechanisms will change the structure of the mathematical models. A further discussion on how the different fault detection mechanisms affect the mathematics is discussed in section 3.2.

Prescheduled backup

In [12], it has been proposed to use a passive replication system where the backups are prescheduled, i.e. a "hot" (ready to run) backup is scheduled to run some time after the latest expected finishing time for the primary. If the primary fails, the backup runs as scheduled, if there is no failure, the scheduled slot for the backup is freed.

As the backup will start to run at a fixed time regardless of when a fault is detected, the runtimes of tasks in this kind of systems can mathematically be treated as the runtimes in systems using timeout as a fault detection mechanism, with the timeout set to the start of the backup schedule and no time used for correction.

Checkpointed systems

In a checkpointed system, a task is divided into several parts, with checkpoints between these parts. The state of the server is recorded when the task reaches a checkpoint, so if a fault is detected, the task can be restarted from the previous checkpoint, i.e. it is not necessary to rerun the whole task from the beginning.

If fault occurrences and the correction of these occurrences can be handled within each part of the task, without affecting earlier or later parts, and if the

runtime of part i, including any fault handling that is necessary, is t_i, the total runtime of an N-part task is given by

$$t_{tot} = \sum_{i=1}^{N} t_i \tag{3.1}$$

If t_i is distributed by the probability density function $g_i(t)$, the whole task has a distribution that can be described as the convolution of the distributions of each part

$$g_{tot}(t) = g_1(t) * g_2(t) * \cdots * g_N(t) \tag{3.2}$$

It should be noted that the timing of systems rolling back more than one checkpoint at a time becomes much more complex to model mathematically.

3.1.2 Timing

Figure 3.2 illustrates how the system behaves during normal operation and during fault situations.

Fault-free runtime

During normal, non-faulty operation, the timing of the task execution is straightforward, and similar to the timing of a task in a non-fault-tolerant system, though there may be some overhead due to the fault tolerance mechanism, e.g the "I'm alive" signaling in systems using watchdogs and time used to state logging. In this work, this time is called the "fault-free runtime".

Fault detection time

When a fault occurs, there will be some time from the fault occurrence to the fault is detected, which in this work is called the "fault detection time". For watchdog fault detectors, this can be modelled as a stochastic time after fault occurrence, with a distribution that is a function of the "I'm alive" signaling frequency, the number of signals that can be missed before the detector reacts and so on. In systems using timeouts or acceptance tests as fault detectors, the detection time will be a part of other parameters, the timeout value or the normal runtime and test time, and a separate detection time is therefore not modeled.

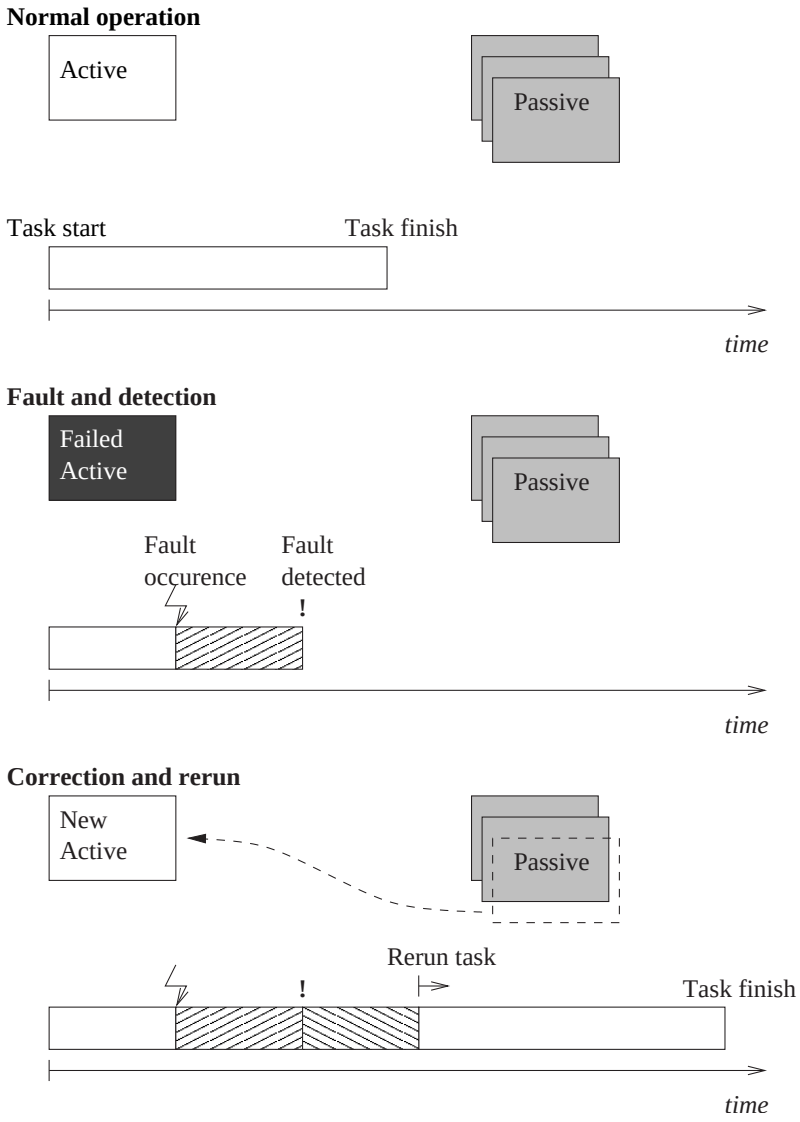


Figure 3.2: The Fault occurrence – Detection – Correction – Rerun cycle of a passive replication system

Fault correction time

When a fault is detected, it usually takes some time before the system is able to rerun the task. The backup that shall function as the new active replica must be brought up-to-date, it may be needed to alert other parts of the system, and other overhead may occur [17]. The time from a fault is detected to system is ready to rerun the task on a backup is called the "fault correction time" in this work.

Test time

In systems using acceptance tests, some time is needed to run the tests. This is called the "test time" in this work.

3.1.3 Fault models

In most of the models, we model faults as independent, generated by a poisson process with a constant intensity. This is a common assumption in reliability modelling, and enables us to model *when* the fault occurs, which is necessary in the models where the detection mechanism is based on watchdogs.

For models where the exact time the fault occurs is not necessary, like in systems using timeout or acceptance test for fault tolerance, it is also possible to use a constant probability that a replica fails.

There is a possibility that several faults will occur during the execution of a single task. If a new fault occurs during the correction of the previous fault, or during the rerun of a task, the new active replica is affected, and the whole detection – correction – rerun cycle will recur. If the new fault occurs before the previous fault is detected, it affects only the already failed replica, and in most of the models there is no need to model any further effect of the fault. For systems where several effects of faults are modeled, new faults leading to a different failure mode of the running replica may affect the system. If, for instance, the running replica has suffered from faults leading both to value and omission failures, the omitted results cannot be tested for the value failures.

3.2 Fault detection models

The fault detection mechanism used in the system will affect the mathematics used in the models. In this work, three classes of fault detectors are modeled. In the **watchdog** model, the time of the fault is detected is dependent on the time

of the fault occurrence, in the **timeout** model, the time of fault detection is a constant time after the start of the execution of the task, and in the **acceptance test** model, faults are detected by a test that is run after a replica has finished its execution.

This section gives a brief description of the different fault detection models, as well as a notified failure model, and explains how the timing differ between the models.

Notified failures

If the component that fails has a notified failure mode, the fault detection mechanisms are instantly aware of the failure, and the fault correction mechanism can be activated right away.

In a mathematical model, this means that the time when the component fails has to be modeled. As the detection is instant, no detection time needs to be modeled.

If the primary fails, and the time of the failure is t_f, the time from a task is started to the fault is detected is given by.

$$t_{det} = t_f \tag{3.3}$$

If there is one failure, and the time to prepare and run the backup is t_b, the total runtime of the task is

$$t_{run} = t_f + t_b \tag{3.4}$$

Watchdogs

If the component that fails has a silent failure mode, and the detection is based on watchdogs, there will be some time from the failure occurs to the fault detection mechanisms are aware of the failure.

As with the notified failure model, to model a watchdog mechanism, the time when the failure occurs has to be modelled. In addition, the time from the failure occurrence to the detection needs to be added. This fault detection time has a distribution that is a function of the signaling strategy, like time between the "I'm alive" signals and the maximum time the system will wait for such a signal before the component is considered failed.

Notified failures can also be modeled into this detection time distribution as a pulse, scaled after the relative frequency of notified to silent failures, at $t = 0$ in the detection time distribution's pdf.

If the primary fails, the time to the failure is t_f, and the from the failure to the detection of the failure is t_d, the total time from a task is started to the failure is detected is given by

$$t_{det} = t_f + t_d \tag{3.5}$$

If there is only one failure, and the time to prepare and run the backup is t_b, the total runtime of the task is

$$t_{run} = t_f + t_d + t_b \tag{3.6}$$

Timeout

To model the runtime distribution of a system using detection based on timeout of final or temporary results, the time when the failure occurs is not needed for the model, as the time when the fault is detected is given independently of the failure time. For a system that has failed, the time from execution start to failure plus the detection time will always be a constant.

If the primary fails, and the timeout is set to t_{to}, the time from a task is started to the failure is detected is the same as the timeout

$$t_{det} = t_{to} \tag{3.7}$$

The runtime of the task if there is one failure is

$$t_{run} = t_{to} + t_b \tag{3.8}$$

where t_b is the time used to prepare and run the backup.

Acceptance test

As with the timeout detection mechanism, the time when the failure occurs is not needed for the model. We assume that the failures that are detected with this mechanism do not affect the normal runtime of the component. We must, however add some extra time for the testing of the results. This time will be added to the normal runtime both for faulty and non-faulty runs.

If the primary fails, the time to test is t_{test}, and the runtime for the primary is t_p, the time from a task is started to the detection of the failure is

$$t_{det} = t_p + t_{test} \tag{3.9}$$

If the time used to prepare and run the backup is t_b, the total runtime of the task in case of one failure is

$$t_{run} = t_p + t_b + 2t_{test} \tag{3.10}$$

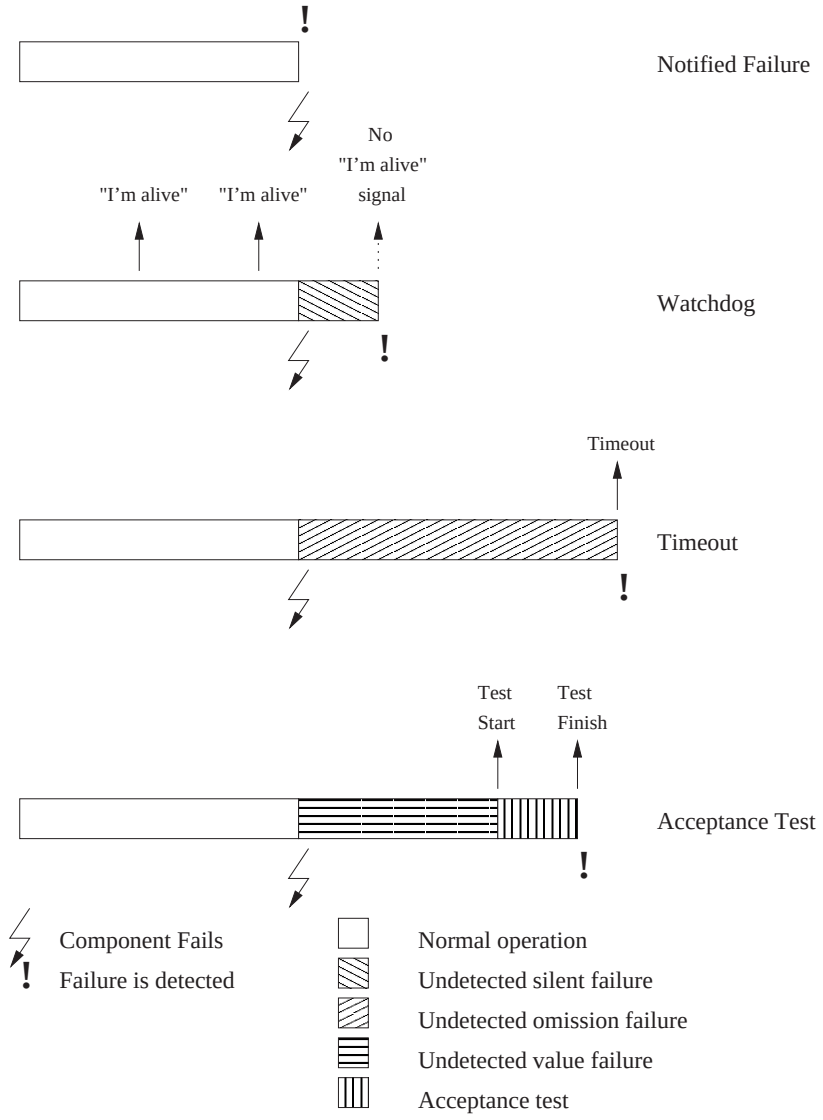


Figure 3.3: Time to detect failure of a component for different detection mechanisms

3.3 Homogeneous and inhomogeneous replication

In many fault tolerant systems, all the replicas in the server are instances of the same component. This makes design and implementation of the system easier, as diverse components for the same tasks are not needed, and updating the state space of a backup means to make it identical as the state space of the primary. This is often considered a "good enough" design, and enables the system to tolerate many of the faults that may occur.

As mentioned in 2.2.3, it may be desirable to use diverse components for replication. The variation in components can give the system protection against a wider range of faults, as some of the faults may be introduced during the design, implementation, or manufacture of the component, and it is probable that these faults recur when run on an identical component.

In real-time systems, the use of imprecise backups can increase the probability of a task that has suffered from a fault occurrence to reach its deadline, at the cost of the accuracy of the results when the backups are used. During normal operation the system will use a precise primary object, but if this should fail, a faster, less accurate backup object is used.

Because there are scenarios where inhomogeneous replication is used, it is reasonable to develop models for both homogeneous and inhomogeneous replication systems.

Mathematically, the difference between homogeneous and inhomogeneous replication systems is large enough that different models have been developed for the two.

For the homogeneous replicas, the same execution time is expected for all replicas given the same input parameters. This means that for a given task, the time use is constant for all replicas. For non-homogeneous replicas, the same execution time is not expected for all replicas, even if the distributions are the same. In this work, the runtime for two inhomogeneous replicas are modelled as independent. While this is not true for many systems, dependence between the runtimes of two variants of a task makes the models much more complex.

As an example of the mathematical difference between runtimes in a homogeneous and an inhomogeneous replicated system, consider a replicated task in which all methods implementing the task have a runtime that is uniformly distributed between t_a and t_b, as shown in figure 3.4a.

If a task has to be run twice on a homogenous replicated system, it will use the same runtime for both runs. This results in a new uniform distribution

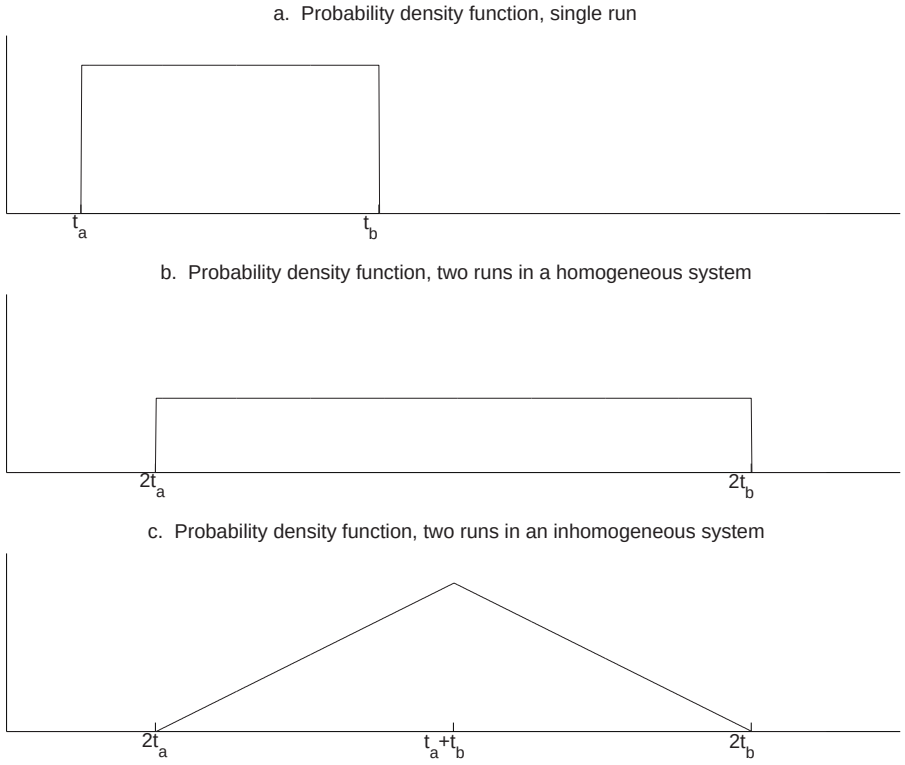


Figure 3.4: Difference in resulting distributions for homogeneous and inhomogeneous systems.

between $2t_a$ and $2t_b$, as shown in figure 3.4b.

If the task has to be run twice on a non-homogeneous replicated system, the runtime will be different for the two runs, resulting in a triangular distribution with minimum $2t_a$, mode $t_a + t_b$ and maximum $2t_b$, as shown in figure 3.4c.

Generalized, if the run-time distribution for the methods implementing a task has a moment generating function $\mathbf{M}(s)$, and the task is run i times, the resulting moment generating function for the total runtime will be $\mathbf{M}(is)$ if the methods are homogeneous and $\mathbf{M}(s)^i$ if the methods are inhomogeneous. Thus, the resulting distributions will be the same for the homogeneous and

inhomogeneous system only if the task is run only once, i.e. $i = 1$, or if the runtime is deterministic, i.e. $\mathbf{M}(s) = e^{-\tau s}$, where τ is the runtime.

3.4 The timing models

As shown in section 3.2, separate models for the three main fault detection strategies, watchdogs, timeout, and acceptance test, must be developed. Section 3.3 shows that the difference between homogeneous and inhomogeneous replication is large enough that separate models are needed for the two. As discussed in section 3.1, many other important differences between passive replication systems do change the timing distributions that are used in the mathematical models, but not the model structures themselves.

This results in six basic models: A model for a homogeneous and an inhomogeneous system using each of the fault detection strategies.

In addition, it can be useful to develop mathematical models for other fault processes, and for systems using combined fault detection.

Chapter 4

Derivation of the mathematical models

In this chapter, mathematical models for the runtime distributions for some of the passive fault tolerant structures described in chapter 3 is derived.

It should be noted that while the mathematical models presented in this chapter cover many different systems, there will always be system structures that are not covered by any of the models presented. It is, however, believed that the methods used to derive the runtime distribution models in this chapter also may be useful for deriving runtime distributions for many systems with structures not described in this work.

4.1 Introduction to the mathematics

The runtime distributions derived in this chapter are functions of distributions that are characteristic for the modeled systems, like the fault-free runtime distribution and the distribution of the time used for fault correction. The functions derived are tied to the structure of the modeled system, e.g. if the fault-free runtime distribution of the system changes, but the structure of the system remains unchanged, the structure of the function used to model the system remains unchanged while the parameters, i.e. the distribution, change.

4.1.1 Moment-generating functions

In the mathematical expression used in this chapter, the distributions will be represented in the form of their moment-generating functions, or mgf. While the use of moment-generating functions in many ways may seem more complex and less intuitive than the probability density function or the cumulative distribution function, mgfs are usually easier to use when expressing distributions as functions of other distributions. Many useful properties of the mgf can be found in common textbooks on the laplace transform, traffic theory, and control engineering [4, 16, 23].

Definition

Consider a random variable X which is distributed with probability density function $f(t)$ and cumulative distribution function $F(t)$. The moment generating function $\mathbf{F}(s)$ is defined as the expectation value of e^{-sX}.

$$\mathbf{F}(s) = \mathrm{E}[e^{-sX}] \tag{4.1}$$

This can be viewed as the laplace transform of the probability density function:

$$\mathbf{F}(s) = \mathcal{L}(f(t)) = \int_0^\infty e^{-st} f(t)dt = \int_0^\infty e^{-st} dF(t) \tag{4.2}$$

It should be noted that the moment-generating function, unlike the probability density function and the cumulative distribution function, is not a function of time (hence the use of the variable s instead of t).

4.1.2 The method

The method used to develop the models is basically the same as used by Kleinrock [21] to derive the expression of the busy period of queuing systems.

The basic steps in developing the models are

1. Set up the expression for the probability of a given number of discrete events, $\Pr[\phi = k]$. In the expressions in this chapter, the discrete events are faults affecting the system.

2. Set up the expression for the total runtime $Y = X_0 + X_1 + \cdots + X_n$ where the time used for each part $X_1 \cdots X_n$ is given, and the number of discrete events k is given. This usually means that Y can be set up as sums of

constant times, where the number of and type of the elements in the sum are dependent of k.

3. Set up the expectation function for e^{-sY}, with the same conditions to time use and number of discrete events as in the previous steps, i.e. the expression for $\mathrm{E}[e^{-sY}|X_i = x_i, \phi = k]$.

4. Remove the condition to the number of discrete events from the expectation function that was set up in the previous step. This is done by calculating the sum $\mathrm{E}[e^{-sY}|X_i = x_i] = \sum_{k=0}^{\infty} \mathrm{E}[e^{-sY}|X_i = x_i, \phi = k] \Pr[\phi = k]$.

5. Remove the condition to the time use. If the part X_i is distributed with the cumulative distribution function $F_i(t)$, the condition on time use is removed by solving the integral $E[e^{-sY}] = \int_0^{\infty} E[e^{-sY}|X_i = x_i]dF_i(x_i)$. By using equation 4.2, the results of this integral becomes an expression of the moment generating function $\mathbf{F}_i(s)$.

4.1.3 Limiting the number of replicas

When the description of the modelled systems was presented in chapter 3, limits to the number of replicas were not discussed. As a base for the models, a system where there always is a spare replica is used. As a result, there is a theoretical possibility that there can be a limitless number of faults occurring to the task, and thus, a limitless number of new replicas used for running the task.

In the mathematical models, the possibility of infinite faults and reruns will appear as an infinite sum. This can be a problem to handle mathematically, and a practical problem when the models are to be used for actual systems.

In a physical system, there is a limit to how many replicas can be used for a task, and because the execution of each replica takes some time, there will also be a limit to how many faults and reruns that can occur in a real-time system before a deadline miss is certain.

In this work, two different ways of solving this problem are presented, both resulting in models with a limit N to the number of faults that can occur to a single task. The first sets N as the limit to the number of faults a task is able to tolerate before the system fails. The second ignores any faults that occur to a task after the Nth fault has occurred. Figure 4.1 illustrates how the two models handles more than N faults.

As the fault probability is usually very low in the first place, N does not have to be a very large number before both models are a good approximate of the theoretical system with infinite replicas.

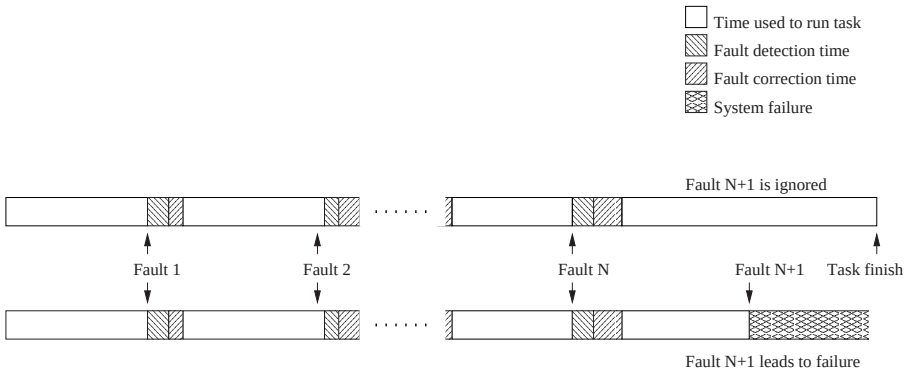


Figure 4.1: Two ways of handling more than N faults

$N+1$ faults cause a certain failure

There is an upper limit to the number of faults that can occur to a single task before the system fails. This will give the simplest solution, as the fault occurrence equations for the Nth fault will be the same as for earlier faults. This model will also be a more accurate description of a physical system where the number of replicas that can be used actually are limited and where the system will fail if all replicas have failed.

A problem with creating the models using this solution, is that the probability of failure gives a runtime distribution with a possibility of the task never finishing its execution, i.e. $\lim_{t\to\infty} G(t) < 1$.

The resulting distribution can therefore not be used in analysis where a "complete" distribution is needed, e.g. when using this distribution as the service time in a steady-state analysis of a queuing system, the results will be an infinite queue. It will, however, be reasonable to use this kind of distribution when analyzing the possibility of system failures.

The distributions derived using this method will not be marked any special way, i.e. $g(t)$, $G(t)$ and $\mathbf{G}(s)$ is used to describe these distributions.

No more than N faults

There is an upper limit to the number of faults that can occur to a single task, and any further faults will be ignored. This will give more complex equations, as a variant fault occurrence equation has to be used for the Nth part of the

system (i.e., the Nth replica that cannot fail). However, as the tasks in this model always finish at some time, i.e. $lim_{t\to\infty}G(t) = 1$, the results can be used for analysis where this is an important property. The solution derived from this method can be viewed as a system that has run for a long time but never has failed.

The distributions derived using this method will be marked with a tilde, i.e. $\tilde{g}(t)$, $\tilde{G}(t)$ and $\tilde{\mathbf{G}}(s)$ is used to describe these distributions.

4.1.4 Naming of distributions used in the models

In this work, a distribution's probability density function is given a function name with a lower-case letter (e.g. $f(t)$), the cumulative distribution function is given a function name with an upper-case letter (e.g. $F(t)$), and the moment-generating function is given a function name with a bold-face letter (e.g. $\mathbf{F}(s)$). Function names with the same letter and index indicate the same distribution, e.g. $f_\alpha(t)$, $F_\alpha(t)$, and $\mathbf{F}_\alpha(s)$ all describe the same distribution.

In the models and the equations used to derive them, the following distributions are used. It should be noted that some of the function names have meanings that vary between the models:

$g(t)$, $G(t)$, $\mathbf{G}(s)$ The distribution of the runtime of the task in a system where faults may occur, i.e. the runtime distribution models that are derived. No special marking indicates that the system fails if more than N faults occur.

$\tilde{g}(t)$, $\tilde{G}(t)$, $\tilde{\mathbf{G}}(s)$ The distribution of the runtime of the task in a system where faults may occur, and where the number of faults to a single task are limited to N.

$m(t)$, $M(t)$, $\mathbf{M}(s)$ The distribution of the fault-free runtime of a method performing the task. If indexed (e.g. $\mathbf{M}_i(s)$), $i = 0$ indicates the runtime distribution of the primary method.

$d(t)$, $D(t)$, $\mathbf{D}(s)$ In a system using watchdogs as fault detection, this is the distribution of the fault detection time. This is the interval between the time when a fault occurs and the time when the fault tolerance mechanism begins to act on that fault occurrence. If indexed, $i = 0$ indicates the fault detection time distribution for a fault in the primary method.

$d(t)$, $D(t)$, $\mathbf{D}(s)$ In an acceptance test system, this is the distribution of the test time.

$c(t)$, $C(t)$, $\mathbf{C}(s)$ The distribution of the fault correction time. This is the time the fault tolerance mechanism use from the detection of a fault to a backup object is ready to rerun the task. If indexed, $i = 1$ indicates the time to update the first backup.

$a_\tau(t)$, $A_\tau(t)$, $\mathbf{A}_\tau(s)$ For a watchdog system, this is the time to a replica fails in a failed run where the fault-free runtime would have been τ. If two fault-free runtimes are indicated (e.g. $\mathbf{A}_{\tau_1,\tau_2}(s)$), it implies that the fault rate changes after the time indicated first, and that the total fault free run-time is the sum of the two times.

$a_{\mathrm{to}i}(t)$, $A_{\mathrm{to}i}(t)$, $\mathbf{A}_{\mathrm{to}i}(s)$ For a timeout system, this is the distribution of time from a failed run of replica i starts to timeout.

$a_i(t)$, $A_i(t)$, $\mathbf{A}_i(s)$ In a system that combines timeout and acceptance test, this is the distribution of the time from a failed run of replica i starts to either a timeout or the beginning of the acceptance test, weighted on the probabilities of different kinds of failures.

4.2 Systems using watchdogs for fault detection

In this section, the expressions for systems with watchdogs as a fault detection mechanism are derived. In these systems, faults are modelled as causing a silent stop failed condition, which is detected by an external fault detector. This model can also be used on other mechanisms that detect faults in a component while it is, or should be, running, like notified failures and errors detected by a component's own exception handler.

4.2.1 The fault model

The faults are modelled as generated from a poisson process with intensity λ.

Many reliability models deal with changing fault rates, due to "infant mortality", maturity and wear of components. If a reliability model deals with a changing fault rate, $\lambda(t)$ due to these effects, the changes in fault rates are usually extremely slow compared to the single run of a task, which is the time frame this work is operating with. Using a fixed fault-rate is therefore a reasonable approximation.

If the execution of a method takes the time τ, the probability that at least one fault happens before the execution ends is given by

$$\Pr[\phi \geq 1] = 1 - e^{-\lambda\tau} \tag{4.3}$$

Given that a fault occurs during the execution, the cumulative density function (cdf) for the time from the start of the execution to the fault occurs is given by

$$A_\tau(t) = \begin{cases} \frac{1}{1-e^{-\lambda\tau}}(1-e^{-\lambda t}) & , \quad 0 \leq t \leq \tau \\ 1 & , \quad t > \tau \end{cases} \tag{4.4}$$

This distribution has the probability density function

$$a_\tau(t) = \begin{cases} \frac{\lambda e^{-\lambda t}}{1-e^{\lambda\tau}} & , \quad 0 \leq t \leq \tau \\ 0 & , \quad t > \tau \end{cases} \tag{4.5}$$

and the moment generating function

$$\mathbf{A}_\tau(s) = \frac{1-e^{(\lambda+s)\tau}}{1-e^{-\lambda\tau}} \frac{\lambda}{\lambda+s} \tag{4.6}$$

Fault model with varied fault rates for different parts of the system

It may also be useful to be able to vary the fault rate between different components. This can be done to model different effects, e.g. using components for backup that are more reliable than the primary component, lower fault probability for the correction phase than for running a method, fault bursts, where the first fault marks the start of a higher fault rate, or the possibility that several components are affected by the same faults.

In the models with changing fault rates, $\lambda_{\mathrm{m}i}$ is used for the fault rate during execution of the ith backup, and $\lambda_{\mathrm{c}i}$ is used for the fault rate during the updating of the ith backup.

The time τ used to prepare a backup and run a method is divided into the time τ_c used to prepare a backup and τ_m used to run the method on the backup. During the correction time (the time used to prepare a backup), the fault rate is λ_c, while during the time used to run a method, the fault rate is λ_m.

The probability that at least one fault happens during the time $\tau_\mathrm{c} + \tau_\mathrm{m}$ is given by

$$\Pr[\phi \geq 1] = 1 - e^{-\lambda_\mathrm{c}\tau_\mathrm{c}} e^{-\lambda_\mathrm{m}\tau_\mathrm{m}} = 1 - e^{-(\lambda_\mathrm{c}\tau_\mathrm{c} + \lambda_\mathrm{m}\tau_\mathrm{m})} \tag{4.7}$$

Given that a fault occurs during the execution, the cumulative distribution function for the time between the start of the interval and the fault occurrence is given by

$$A_{\tau_c,\tau_m}(t) = \begin{cases} \frac{1}{1-e^{-(\lambda_c\tau_c+\lambda_m\tau_m)}}(1-e^{-\lambda_c t}) & , \quad 0 \le t < \tau_c \\ \frac{1}{1-e^{-(\lambda_c\tau_c+\lambda_m\tau_m)}}(1-e^{-(\lambda_c-\lambda_m)\tau_c}e^{\lambda_m t} & , \quad \tau_c \le t < \tau_c+\tau_m \\ 1 & , \quad t \ge \tau_c+\tau_m \end{cases} \tag{4.8}$$

This corresponds to the probability density function

$$a_{\tau_c,\tau_m}(t) = \begin{cases} \frac{1}{1-e^{-(\lambda_c\tau_c+\lambda_m\tau_m)}}\lambda_c e^{-\lambda_c t} & , \quad 0 \le t < \tau_c \\ \frac{1}{1-e^{-(\lambda_c\tau_c+\lambda_m\tau_m)}}\lambda_m e^{-(\lambda_c-\lambda_m)\tau_c}e^{\lambda_m t} & , \quad \tau_c \le t < \tau_c+\tau_m \\ 0 & , \quad t \ge \tau_c+\tau_m \end{cases} \tag{4.9}$$

and the moment-generating function

$$\begin{aligned} \mathbf{A}_{\tau_c,\tau_m}(t) &= \frac{1}{1-e^{-(\lambda_c\tau_c+\lambda_m\tau_m)}}\left((1-e^{-\tau_c(\lambda_c+s)})\frac{\lambda_c}{\lambda_c+s}\right. \\ &\quad \left. + (e^{-\tau_c(\lambda_c+s)} - e^{-(\tau_c(\lambda_c+s)+\tau_m(\lambda_m+s))})\frac{\lambda_m}{\lambda_m+s}\right) \end{aligned} \tag{4.10}$$

4.2.2 Inhomogeneous systems

As described in 3.3, the systems are divided into *inhomogeneous* and *homogeneous* replication systems. For an inhomogeneous system, there is independency between the fault-free runtimes of the primary and the backups. The distribution of these runtimes may be the same, however.

First, a model where the time used for detection and correction of the fault is 0 is derived, i.e., in this model, the task will start running on a backup replica as soon as a fault occurs.

For a system where the runtimes for the replicas are inhomogeneous x_i, where x_0 indicate the runtime for the primary replica, the number of faults ϕ before one of the replicas completes a fault-free execution is given by the probability function

$$\Pr[\phi = k] = \begin{cases} e^{-\lambda x_0} & , \quad k = 0 \\ e^{-\lambda x_k}\prod_{i=0}^{k-1}(1-e^{-\lambda x_i}) & , \quad k > 0 \end{cases} \tag{4.11}$$

The total runtime of a system with ϕ faults is given by

$$Y = r_\phi + X_0 + X_1 + \cdots + X_{\phi-1} \tag{4.12}$$

where X_i is the time to fault for each faulty run, and r_ϕ is the runtime for the fault-free run.

What is sought after is the distribution of the total time use,

$$G(t) = \Pr(Y \leq t) \tag{4.13}$$

expressed by the mgf

$$\mathbf{G}(s) = \int_0^\infty e^{-st} dG(t) = \mathrm{E}[e^{-sY}] = \mathrm{E}[e^{-s(r+X_0+X_1+\cdots+X_{\phi-1})}] \tag{4.14}$$

We start by introducing conditions on run-times and number of faults

$$\begin{aligned} \mathrm{E}[e^{-sY}|r_\phi = x_\phi, \phi = k] &= \mathrm{E}[e^{-s(x_k+X_0+X_1+\cdots+X_{k-1})}] \\ &= \mathrm{E}[e^{-sx_k} e^{-sX_0} e^{-sX_1} \cdots e^{-sX_{k-1}}] \end{aligned} \tag{4.15}$$

Because the runtimes x_k and $X_0 \ \cdots \ X_{k-1}$ are independent, this can be rewritten as

$$\mathrm{E}[e^{-sY}|r_\phi = x_\phi, \phi = k] = \mathrm{E}[e^{-sx_k}]\mathrm{E}[e^{-sX_0}]\mathrm{E}[e^{-sX_1}] \cdots \mathrm{E}[e^{X_{k-1}}] \tag{4.16}$$

As the runtime in a faulty run would have been x_i if there had been no fault, the time to fault X_i is distributed by $A_{x_i}(t)$ described in equation 4.4. This means that $\mathrm{E}[e^{-sX_i}] = \mathbf{A}_{x_i}(s)$, and equation 4.16 can be rewritten as

$$\mathrm{E}[e^{-sY}|r_n = x_n, \phi = k] = \mathrm{E}[e^{-sx_k}] \prod_{i=0}^{k-1} \mathbf{A}_{x_i}(s) \tag{4.17}$$

The next step is to remove the condition on the number of faults. This is done by taking the sum of the products of the expectation function where the number of faults is k and probability of k faults, for each k.

$$\mathrm{E}[e^{-sY}|r_n = x_n] = \sum_{k=0}^{\infty} E[e^{-sY}|r_n = x_n, \phi = k] \Pr(\phi = k) \tag{4.18}$$

By doing this with the fault probability function 4.11 and the expectation function 4.17, the resulting expectation function is

$$\begin{aligned} &\mathrm{E}[e^{-sY}|r_n = x_n] \\ &= e^{-sx_0} e^{-\lambda x_0} + \sum_{k=1}^{\infty} e^{-sx_k} e^{-\lambda x_k} \prod_{i=0}^{k-1} \mathbf{A}_{x_i}(s)(1 - e^{-\lambda x_i}) \\ &= e^{-(\lambda+s)x_0} + \sum_{k=1}^{\infty} e^{-(\lambda+s)x_k} \prod_{i=0}^{k-1} (1 - e^{-(\lambda+s)x_i}) \frac{\lambda}{\lambda + s} \end{aligned} \tag{4.19}$$

As discussed in 4.1.3, there are some problems working further with the equation as it has an infinite sum. As described, two different "workarounds" are used, where $\mathbf{G}(s)$ describes a system that fails if there are more than N faults, and $\tilde{\mathbf{G}}(s)$ describes a system where more than N faults cannot occur.

The equation with a maximum of N faults before a failure is given by:

$$\begin{aligned}&\mathrm{E}[e^{-sY}|r_n = x_n] \\ &= e^{-(\lambda+s)x_0} + \sum_{k=1}^{N} e^{-(\lambda+s)x_k} \prod_{i=0}^{k-1}(1 - e^{-(\lambda+s)x_i})\frac{\lambda}{\lambda+s}\end{aligned} \tag{4.20}$$

The condition on time use is removed by integrating the expression with respect to the runtime distributions

$$\begin{aligned}&\mathbf{G}(s) = \mathrm{E}[e^{-sY}] \\ &= \int_0^\infty \cdots \int_0^\infty \mathrm{E}[e^{-sY}|r_n = x_n] dM_0(x_0) \cdots dM_N(x_N)\end{aligned} \tag{4.21}$$

By using the properties of the moment generating function (eq. 4.2), we get

$$\begin{aligned}&\mathbf{G}(s) \\ &= \mathbf{M}_0(\lambda+s) + \sum_{k=1}^{N} \left(\frac{\lambda}{\lambda+s}\right)^k \mathbf{M}_k(\lambda+s) \prod_{i=0}^{k-1}(1 - \mathbf{M}_i(\lambda+s))\end{aligned} \tag{4.22}$$

Equation 4.22 shows the mgf for the distribution of the runtime in a passive replicated fault tolerant system without the fault detection and fault correction times, where more than N faults will lead to the failure of the system.

If we use the model where no more than N faults may happen during a single task, we have to change the fault probability function to

$$\Pr[\phi = k] = \begin{cases} e^{-\lambda x_0} & , \quad k = 0 \\ e^{-\lambda x_k} \prod_{i=0}^{k-1}(1 - e^{-\lambda x_i}) & , \quad 0 < k < N \\ \prod_{i=0}^{N-1}(1 - e^{-\lambda x_i}) & , \quad k = N \end{cases} \tag{4.23}$$

By using this fault probability function to remove the condition of a given

number of faults, we get the expectation function

$$\begin{aligned}
&\mathrm{E}[e^{-sY}|r_n = x_n] \\
&= e^{-(\lambda+s)x_0} \\
&\quad + \left(\sum_{k=1}^{N-1} e^{-(\lambda+s)x_k} \prod_{i=0}^{k-1} (1 - e^{-(\lambda+s)x_i}) \frac{\lambda}{\lambda+s} \right) \\
&\quad + e^{-sx_N} \prod_{i=0}^{N-1} (1 - e^{-(\lambda+s)x_i}) \frac{\lambda}{\lambda+s}
\end{aligned} \tag{4.24}$$

When the conditions on time use is removed, we end up with

$$\begin{aligned}
&\tilde{\mathbf{G}}(s) \\
&= \mathbf{M}_0(\lambda+s) \\
&\quad + \left(\sum_{k=0}^{N-1} \left(\frac{\lambda}{\lambda+s} \right)^k \mathbf{M}_k(\lambda+s) \prod_{i=0}^{k-1} (1 - \mathbf{M}_i(\lambda+s)) \right) \\
&\quad + \left(\frac{\lambda}{\lambda+s} \right)^N \mathbf{M}_N(s) \prod_{i=0}^{N-1} (1 - \mathbf{M}_i(\lambda+s))
\end{aligned} \tag{4.25}$$

Equation 4.25 shows the mgf for the distribution of the runtime in a passively replicated fault tolerant system without fault detection and fault correction delays, where faults after the Nth fault is ignored.

Adding the fault correction delay

We will now expand the expression by adding the delay for fault correction. The correction process consists of updating and readying a backup object, so that the state of this object reflects the state of the failed primary object, and the backup is ready to be used as the new primary. We assume that faults may happen during the correction process.

This can be done by changing the equations 4.11–4.25 so that the correction time is included. Equation 4.11 is changed to

$$\begin{aligned}
&\Pr[\phi = k] = \\
&\begin{cases}
e^{-\lambda x_0} & , k = 0 \\
e^{-\lambda(x_k+y_k)}(1 - e^{\lambda x_0}) \prod_{i=1}^{k-1} (1 - e^{-\lambda(x_i+y_i)}) & , k > 0
\end{cases}
\end{aligned} \tag{4.26}$$

and equation 4.17 is changed to

$$\begin{aligned}&\mathrm{E}[e^{-sY}|r_n = x_n, c_n = y_n, \phi = k] \\ &= \begin{cases} e^{-sx_0} & , \quad k = 0 \\ e^{-s(x_k+y_k)}\mathbf{A}_{x_0}(s)\prod_{i=1}^{k-1}\mathbf{A}_{x_i+y_i}(s) & , \quad k > 0 \end{cases}\end{aligned} \tag{4.27}$$

where the c_n is the correction time for replica n.

The expression is otherwise derived as for the system without correction time. Using the fault probability function 4.26 to remove the condition on number of faults in the expectation function 4.27 yields

$$\begin{aligned}&\mathrm{E}[e^{-sY}|r_n = x_n, c_n = y_n] \\ &= e^{-(\lambda+s)x_0} \\ &\quad + \sum_{k=1}^{N}\left(\frac{\lambda}{\lambda+s}\right)^k e^{-(\lambda+s)(x_k+y_k)} \\ &\quad (1 - e^{-(\lambda+s)x_0})\prod_{i=1}^{k-1}(1 - e^{-(\lambda+s)(x_i+y_i)})\end{aligned} \tag{4.28}$$

Removing the conditions on runtimes and correction times gives the moment-generating function for the system with fault correction time included:

$$\begin{aligned}&\mathbf{G}(s) \\ &= \mathbf{M}_0(\lambda+s) \\ &\quad + \sum_{k=1}^{N}\left(\frac{\lambda}{\lambda+s}\right)^k \mathbf{M}_k(\lambda+s)\mathbf{C}_k(\lambda+s) \\ &\quad (1 - \mathbf{M}_0(\lambda+s))\prod_{i=1}^{k-1}(1 - \mathbf{M}_i(\lambda+s)\mathbf{C}_i(\lambda+s))\end{aligned} \tag{4.29}$$

Equation 4.29 can also be derived by combining the correction time with the run time for the replicas, i.e. let the run time for replica n be distributed with pdf $c_n(t) * m_n(t)$ where $*$ is the convolution operator. In the mgf domain, this corresponds to a normal multiplication, i.e. $\mathbf{C}(s)\mathbf{M}(s)$.

For the model where faults after the Nth are ignored, the fault probability

function is given by

$$\Pr[\phi = k] = \begin{cases} e^{-\lambda x_0} & , \quad k = 0 \\ e^{-\lambda(x_k+y_k)}(1-e^{\lambda x_0})\prod_{i=1}^{k-1}(1-e^{-\lambda(x_i+y_i)}) & , \ 0 < k < N \\ (1-e^{\lambda x_0})\prod_{i=1}^{N-1}(1-e^{-\lambda(x_i+y_i)}) & , \quad k = N \end{cases} \tag{4.30}$$

Using this probability function to remove the condition on the number of faults from the expectation function 4.27 gives the expectation function

$$\begin{aligned} &\mathrm{E}[e^{-sY}|r_n = x_n, c_n = y_n] \\ &= e^{-(\lambda+s)x_0} \\ &\quad + \left(\sum_{k=1}^{N-1} \left(\frac{\lambda}{\lambda+s} \right)^k e^{-(\lambda+s)(x_k+y_k)} \right. \\ &\qquad \left. (1-e^{-(\lambda+s)x_0}) \prod_{i=1}^{k-1}(1-e^{-(\lambda+s)(x_i+y_i)}) \right) \\ &\quad + \left(\frac{\lambda}{\lambda+s} \right)^N e^{-s(x_N+y_N)} \\ &\qquad (1-e^{-(\lambda+s)x_0}) \prod_{i=1}^{N-1}(1-e^{-(\lambda+s)(x_i+y_i)}) \end{aligned} \tag{4.31}$$

Removal of the timing conditions yields

$$\begin{aligned} &\tilde{\mathbf{G}}(s) \\ &= \mathbf{M}_0(\lambda+s) \\ &\quad + \left(\sum_{k=1}^{N-1} \left(\frac{\lambda}{\lambda+s} \right)^k \mathbf{M}_k(\lambda+s)\mathbf{C}_k(\lambda+s) \right. \\ &\qquad \left. (1-\mathbf{M}_0(\lambda+s)) \prod_{i=1}^{k-1}(1-\mathbf{M}_i(\lambda+s)\mathbf{C}_i(\lambda+s)) \right) \\ &\quad + \left(\frac{\lambda}{\lambda+s} \right)^N \mathbf{M}_N(s)\mathbf{C}_N(s) \\ &\qquad (1-\mathbf{M}_0(\lambda+s)) \prod_{i=1}^{N-1}(1-\mathbf{M}_i(\lambda+s)\mathbf{C}_i(\lambda+s)) \end{aligned} \tag{4.32}$$

Equation 4.32 shows the mgf for the distribution of the runtime in a passively replicated fault tolerant system with a watchdog fault detector, where faults after the Nth are ignored, and where there is no fault detection delay

Adding the fault detection delay

The equation is expanded further by adding the delay used for fault detection. During this time, the replica that is running is already in an erroneous state, so new faults happening during this time can be ignored.

The probability on the number of faults will thus be the same as for a system without fault detection, i.e. equation 4.26.

The expected time use for a system where k faults occur will be

$$\begin{aligned}&\mathrm{E}[e^{-sY}|r_n = x_n, c_n = y_n, d_n = z_n, \phi = k] \\ &= \begin{cases} e^{-sx_0} & , k = 0 \\ e^{-s(x_k+y_k)}\mathbf{A}_{x_0}(s)\prod_{i=1}^{k-1}\mathbf{A}_{x_i+y_i}(s)e^{-sz_i} & , k > 0 \end{cases}\end{aligned} \tag{4.33}$$

The mgf for a system with fault detection and fault correction times included is otherwise derived the same way as for the systems without detection, resulting in the moment-generating system for the inhomogeneous watchdog system

$$\begin{aligned}&\mathbf{G}(s) \\ &= \mathbf{M}_0(\lambda+s) \\ &\quad + \sum_{k=1}^{N}\left(\frac{\lambda}{\lambda+s}\right)^k \mathbf{M}_k(\lambda+s)\mathbf{C}_k(\lambda+s) \\ &\quad (1-\mathbf{M}_0(\lambda+s))\mathbf{D}_0(s)\prod_{i=1}^{k-1}(1-\mathbf{M}_i(\lambda+s)\mathbf{C}_i(\lambda+s))\mathbf{D}_i(s)\end{aligned} \tag{4.34}$$

Equation 4.34 gives an expression for the total runtime of a task in passive fault tolerant systems with inhomogeneous replicas using watchdogs as fault detection, where more than N faults lead to the failure of the system, as a function of the distributions of the fault free runtimes, the correction times, and the fault detection times of the systems.

For the model where more than N faults are ignored, the fault probability

equation 4.30 is used, giving the moment-generating function

$$\begin{aligned}
&\tilde{\mathbf{G}}(s) \\
&= \mathbf{M}_0(\lambda+s) \\
&\quad + \left(\sum_{k=1}^{N-1} \left(\frac{\lambda}{\lambda+s} \right)^k \mathbf{M}_k(\lambda+s)\mathbf{C}_k(\lambda+s) \right. \\
&\quad \left. (1-\mathbf{M}_0(\lambda+s))\mathbf{D}_0(s) \prod_{i=1}^{k-1} (1-\mathbf{M}_i(\lambda+s)\mathbf{C}_i(\lambda+s))\mathbf{D}_i(s) \right) \quad (4.35) \\
&\quad + \left(\frac{\lambda}{\lambda+s} \right)^N \mathbf{M}_N(s)\mathbf{C}_N(s) \\
&\quad (1-\mathbf{M}_0(\lambda+s))\mathbf{D}_0(s) \prod_{i=1}^{N-1} (1-\mathbf{M}_i(\lambda+s)\mathbf{C}_i(\lambda+s))\mathbf{D}_i(s)
\end{aligned}$$

Equation 4.35 gives an expression for the total runtime of a task in passive fault tolerant systems with inhomogeneous replication using watchdogs for fault detection, where there can be no more than N faults occurring to a single task.

4.2.3 Homogeneous systems

In the homogeneous system, the same method is tried as a backup as in the primary. Because the same methods with the same input parameters are used for both the primary and the backups, the time used to complete a given task will be the same for all replicas. We also assume that all backups are in the same state, so that for multiple failures, the times used to update the backups are the same for all backups. As described in 3.3, using the inhomogeneous model with the same distribution for all replicas will not be a solution for this problem, as the same runtime distribution for all replicas does not ensure the same runtime for all backups.

The steps used in the derivation of this model are the same as used for the heterogeneous system.

The work described here is an updated version of the work presented in [46].

As for the heterogeneous system, the runtime distribution in a system where the detection and correction time is negligible is derived first.

For a system where the runtime for a task on a replica is x, the probability function for the number of faults that occurs is

$$\Pr(\phi = k) = e^{-\lambda x}(1-e^{-\lambda x})^k \quad (4.36)$$

When there is a possibility of faults occurring, the total running time of the task will be

$$Y = r + X_0 + X_1 + \cdots + X_{\phi-1} \tag{4.37}$$

where r is the runtime for the fault free run, X_i is the time until abortion of a faulty run, and ϕ is the number of faulty runs.

We wish to find the distribution of the total time use for the task, expressed by this distribution's moment generating function

$$\mathbf{G}(s) = \int_0^\infty e^{-st} dG(t) = \mathrm{E}[e^{-sY}] = \mathrm{E}[e^{-s(r+X_0+X_1+\cdots+X_\phi-1)}] \tag{4.38}$$

By introducing conditions on the runtime and the number of faults, we get the expectation function

$$\begin{aligned} \mathrm{E}[e^{-sY}|r = x, \phi = k] &= \mathrm{E}[e^{-s(x+X_0+\cdots+X_{k-1})}] \\ &= \mathrm{E}[e^{-sx}e^{-sX_0}\cdots e^{-sX_{k-1}}] \end{aligned} \tag{4.39}$$

As the times x and $X_0 \cdots X_{k-1}$ are independent of each other, and since $\mathrm{E}[e^{-sX_i}] = \mathbf{A}_x(s)$, the expectation function can be rewritten as

$$\begin{aligned} \mathrm{E}[e^{-sY}|r = x, \phi = k] &= \mathrm{E}[e^{-sx}]\mathrm{E}[e^{-sX_{k-1}}]\cdots\mathrm{E}[e^{-sX_0}] \\ &= \mathrm{E}[e^{-sx}](\mathbf{A}_x(s))^k \end{aligned} \tag{4.40}$$

The condition on the number of faults is removed by multiplying the probability for a given number of faults k with the expectation function where k faults are given, and summing the products for all possible number of faults.

$$\mathrm{E}[e^{-sY}|r = x] = \sum_{k=0}^{\infty} \mathrm{E}[e^{-sY}|r = x, \phi = k] \Pr(\phi = k) \tag{4.41}$$

This results in the expectation function

$$\begin{aligned} \mathrm{E}[e^{-sY}|r = x] &= \sum_{k=0}^{\infty} e^{-xs}(\mathbf{A}_x(s))^k e^{-\lambda x}(1 - e^{-\lambda x})^k \\ &= e^{-x(\lambda+s)} \sum_{k=0}^{\infty} ((1 - e^{-x(\lambda+s)})\frac{\lambda}{\lambda+s})^k \end{aligned} \tag{4.42}$$

Again, we use this as a base to get two results, where $\mathbf{G}(s)$ describes a system that will fail if more than N faults occur to the same task and $\tilde{\mathbf{G}}(s)$ describes a system that no more than N faults may occur to the task.

By setting a maximum number of faults N before the system fails, the expectation function becomes

$$\mathrm{E}[e^{-sY}|r=x] = e^{-x(\lambda+s)} \sum_{k=0}^{N} ((1-e^{-x(\lambda+s)}) \frac{\lambda}{\lambda+s})^k \tag{4.43}$$

The condition on time use is removed by integrating this function with respect to the runtime distribution

$$\mathbf{G}(s) = \mathrm{E}[e^{-sY}] = \int_0^\infty \mathrm{E}[e^{-sY}|r=x] dM(x) \tag{4.44}$$

This gives us the distribution with the moment-generating function

$$\mathbf{G}(s) = \sum_{k=0}^{N} \frac{\lambda}{\lambda+s} \left(\sum_{i=0}^{k} (-1)^i \binom{k}{i} \mathbf{M}((i+1)(\lambda+s)) \right) \tag{4.45}$$

Equation 4.45 shows the mgf for the distribution of the runtime of a task in a homogeneous passive replication where the time used to detect and correct a fault is negligible and where more than N faults will lead to the failure of the system.

If the model where no more than N faults may occur to a single task, the probability function for the number of faults has to be rewritten to

$$\Pr[\phi = k | \phi \leq N] = \begin{cases} e^{-\lambda x}(1-e^{-\lambda x})^k & , k \leq N-1 \\ (1-e^{-\lambda x})^N & , k = N \end{cases} \tag{4.46}$$

Using this fault probability function to remove the condition on the number of faults yields the expectation function

$$\begin{aligned} \mathrm{E}[e^{-sY}|r=x] = e^{-x(\lambda+s)} & \left(\sum_{k=0}^{N-1} ((1-e^{-x(\lambda+s)}) \frac{\lambda}{\lambda+s})^k \right) \\ & + e^{-sx} ((1-e^{-x(\lambda+s)}) \frac{\lambda}{\lambda+s})^N \end{aligned} \tag{4.47}$$

By removing the condition on time use, we get the moment-generating function

$$\begin{aligned} \tilde{\mathbf{G}}(s) = & \left(\sum_{k=0}^{N-1} \frac{\lambda}{\lambda+s}^k \left(\sum_{i=0}^{k} (-1)^i \binom{k}{i} \mathbf{M}((i+1)(\lambda+s)) \right) \right) \\ & + \frac{\lambda}{\lambda+s}^N \left(\sum_{i=0}^{N} (-1)^i \binom{N}{i} \mathbf{M}(i\lambda + (i+1)s) \right) \end{aligned} \tag{4.48}$$

Equation 4.48 shows the mgf for the distribution of the runtime of a task in a passively replicated fault tolerant system, where the time used for fault detection and correction is negligible and faults after the Nth are ignored.

Adding time for fault correction

The expression is expanded by adding the time used for fault correction. We assume that a fault may happen during the fault correction process. As there is no correction during the initial run of the method, we have to distinguish between the initial run and the following runs. The modification is done by changing the probability function for the number of faults (equation 4.36) and the conditional expectation function (equation 4.40).

The new probability function for the number of faults is

$$\Pr[\phi = k] = \begin{cases} e^{-\lambda x} & , k = 0 \\ e^{-\lambda(x+y)}(1 - e^{-\lambda x})(1 - e^{-\lambda(x+y)})^{k-1} & , k \geq 1 \end{cases} \tag{4.49}$$

where c is the time used for fault correction.

The conditional expectation function is changed to

$$\begin{aligned} &\mathrm{E}[e^{-sY} | r = x, c = y, \phi = k] \\ &= \begin{cases} e^{-sx} & , k = 0 \\ e^{-s(x+y)}\mathbf{A}_x(s)\mathbf{A}_{x+y}(s)^{k-1} & , k > 0 \end{cases} \end{aligned} \tag{4.50}$$

Using the probability function 4.49 to remove the condition on the number of faults in expectation function 4.50 yields the expectation function

$$\begin{aligned} &\mathrm{E}[e^{-sY} | r = x, c = y] \\ &= e^{-(\lambda+s)x} \\ &\quad + \sum_{k=1}^{N} \left(\frac{\lambda}{\lambda + s} \right)^k e^{-(\lambda+s)(x+y)} \\ &\quad (1 - e^{-(\lambda+s)x})(1 - e^{-(\lambda+s)(x+y)})^{k-1} \end{aligned} \tag{4.51}$$

Removing the conditions on time use gives the moment generating function

$$\begin{aligned} &\mathbf{G}(s) \\ &= \mathbf{M}(\lambda + s) \\ &\quad + \sum_{k=1}^{N} \left(\frac{\lambda}{\lambda + s} \right)^k \left(\sum_{i=0}^{k-1} (-1)^i \binom{k}{i} (\mathbf{M}((i+1)(\lambda + s)) \right. \\ &\quad \left. - \mathbf{M}((i+2)(\lambda + s))\mathbf{C}((i+1)(\lambda + s)) \right) \end{aligned} \tag{4.52}$$

Equation 4.52 shows the moment generating function of the runtime distribution of a task in a homogeneous passive replicated system where correction starts as soon as a fault occurs, and where more than N faults to a single task lead to a system failure.

For a system where there can be no more than N faults occurring to the same task, the probability function for the number of faults is given by

$$\Pr[\phi = k] = \begin{cases} e^{-\lambda x} & ,k = 0 \\ e^{-\lambda(x+y)}(1 - e^{-\lambda x})(1 - e^{-\lambda(x-y)})^{k-1} & ,1 \leq k < N \\ (1 - e^{-\lambda x})(1 - e^{-\lambda(x+y)})^{N-1} & ,k = N \end{cases} \quad (4.53)$$

Using this fault probability function, we get the following expectation function when removing the condition on the number of faults

$$\begin{aligned} &\mathrm{E}[e^{-sY}|r = x, c = y] \\ &= e^{-(\lambda+s)x} \\ &\quad + \sum_{k=1}^{N-1} \left(\frac{\lambda}{\lambda+s}\right)^k e^{-(\lambda+s)(x+y)} \\ &\quad (1 - e^{-(\lambda+s)x})(1 - e^{-(\lambda+s)(x+y)})^{k-1} \\ &\quad + \left(\frac{\lambda}{\lambda+s}\right)^N e^{-s(x+y)}(1 - e^{-(\lambda+s)x})(1 - e^{-(\lambda+s)(x+y)})^{N-1} \end{aligned} \quad (4.54)$$

The moment generating function for the total runtime will then be

$$\begin{aligned} &\tilde{\mathbf{G}}(s) \\ &= \mathbf{M}(\lambda + s) \\ &\quad + \left(\sum_{k=1}^{N-1} \left(\frac{\lambda}{\lambda+s}\right)^k \left(\sum_{i=0}^{k-1} (-1)^i \binom{k}{i} (\mathbf{M}((i+1)(\lambda+s)) \right.\right. \\ &\quad \left.\left. - \mathbf{M}((i+2)(\lambda+s)))\mathbf{C}((i+1)(\lambda+s)) \right) \right) \\ &\quad + \left(\frac{\lambda}{\lambda+s}\right)^N \left(\sum_{i=0}^{N-1} (-1)^i \binom{N-1}{i} (\mathbf{M}(i\lambda + (i+1)s) \right. \\ &\quad \left. - \mathbf{M}((i+1)\lambda + (i+2)s))\mathbf{C}(i\lambda + (i+1)s) \right) \end{aligned} \quad (4.55)$$

Equation 4.55 describes the moment-generating function for the runtime distribution of a homogeneous watchdog system where there is no fault detection delay, and where any faults occurring to a single task after the Nth are ignored.

Adding the fault detection delay

The expression is expanded further by adding the fault detection time.

As the running object has already failed, and no new objects are running, we assume that new faults happening during fault detection time can be ignored.

Unlike the task execution time and the correction time distributions, we cannot assume that the fault detection times are homogeneous, and we have to use the same methods as for the inhomogeneous system to model this. We can, however, assume that all fault detection times have the same distribution.

While we still have the same fault number probability as in equation 4.49, we have to modify equation 4.50 to

$$\begin{aligned}&\mathrm{E}[e^{-sY}|r=x,c=y,d_i=z_i,\phi=k]\\&=\begin{cases}e^{-sx} & ,k=0\\ e^{-s(x+y)}\mathbf{A}_x(s)\mathbf{A}_{x+y}(s)^{k-1}\prod_{i=0}^{k-1}e^{-sz_i} & ,k>0\end{cases}\end{aligned} \tag{4.56}$$

where d_i is the time used for detecting the fault in the ith replica

The steps for deriving the expression are the same as for the systems without fault detection.

For a system that fails if more than N faults occur to the same task, the runtime distribution is given by

$$\begin{aligned}&\mathbf{G}(s)\\&=\mathbf{M}(\lambda+s)\\&\quad+\left(\sum_{k=1}^{N}\left(\frac{\lambda}{\lambda+s}\mathbf{D}(s)\right)^k\right.\\&\quad\left(\sum_{i=0}^{k-1}(-1)^i\binom{k}{i}(\mathbf{M}((i+1)(\lambda+s))\right.\\&\quad\left.\left.-\mathbf{M}((i+2)(\lambda+s)))\mathbf{C}((i+1)(\lambda+s))\right)\right)\end{aligned} \tag{4.57}$$

Equation 4.57 shows the moment-generating function for the runtime distribution for a task in a passive replication system where the replicas are homogeneous, the fault detection is based on a watchdog mechanism, and where more than N faults happening to a single task will lead to the failure of the system.

For the system model where no more than N faults may occur to a single

task, the runtime distribution is given by

$$
\begin{aligned}
&\tilde{\mathbf{G}}(s) \\
&= \mathbf{M}(\lambda+s) \\
&\quad + \left(\sum_{k=1}^{N-1} \left(\frac{\lambda}{\lambda+s} \mathbf{D}(s) \right)^k \right. \\
&\quad \left(\sum_{i=0}^{k-1} (-1)^i \binom{k}{i} (\mathbf{M}((i+1)(\lambda+s)) \right. \\
&\quad \left. \left. - \mathbf{M}((i+2)(\lambda+s))) \mathbf{C}((i+1)(\lambda+s)) \right) \right) \\
&\quad + \left(\frac{\lambda}{\lambda+s} \mathbf{D}(s) \right)^N \\
&\quad \left(\sum_{i=0}^{N-1} (-1)^i \binom{N-1}{i} (\mathbf{M}(i\lambda + (i+1)s) \right. \\
&\quad \left. - \mathbf{M}((i+1)\lambda + (i+2)s)) \mathbf{C}(i\lambda + (i+1)s) \right)
\end{aligned}
\tag{4.58}
$$

Equation 4.58 shows the moment-generating function for the runtime distribution of a task in a passive replication system where the replicas are homogeneous, the fault detection is based on a watchdog mechanism, and where any faults to a single task after the Nth are ignored.

4.2.4 Systems with multiple fault rates

It can sometimes be useful to use a model of a system where the fault rates are not the same for all the replicas. Examples are when we can assume that there is a possibility that the same fault will recur when the task is rerun on the backup, when faults arrive in bursts, or when it is believed that some replicas are considered more reliable than others.

Here, an inhomogeneous replication system with changing fault rates is modeled. Each replica i has a separate fault rate λ_{ci} during backup preparation and λ_{mi} during method execution.

For this system, the probability of a given number of faults is given by

$$
\begin{aligned}
&\Pr[\phi = k] \\
&= \begin{cases}
e^{-\lambda_{m0} x_0} & , k = 0 \\
e^{-(\lambda_{ck} y_k + \lambda_{mk} x_k)} & \\
\quad (1 - e^{-\lambda_{m0} x_0}) \prod_{i=1}^{k-1} (1 - e^{-(\lambda_{ci} y_i + \lambda_{mi} x_i)}) & , k > 0
\end{cases}
\end{aligned}
\tag{4.59}
$$

As for the systems with a constant fault rate, an expectation function for e^{-sY}, where Y is the runtime, is created with conditions to the number of faults and the time use.

$$\begin{aligned}
&\mathrm{E}[e^{-sY}|r_n = x_n, c_n = y_n, d_n = z_n, \phi = k] \\
&= \begin{cases} e^{-sx_0} & , k = 0 \\ e^{-sx_k + y_k} \mathbf{A}_{x_0}(s) \prod_{i=1}^{k-1} \mathbf{A}_{y_i,x_i}(s) e^{-sz_i} & , k > 0 \end{cases}
\end{aligned} \tag{4.60}$$

For a system that will fail if more than N faults occur to the same task, the removal of the condition on the number of faults yields

$$\begin{aligned}
&E[e^{-sY}|r_n = x_n, c_n = y_n, d_n = z_n] \\
&= e^{-(\lambda_{\mathrm{m}0}+s)x_0} \\
&\quad + \sum_{k=1}^{N} e^{-(\lambda_{\mathrm{c}k}+s)y_k} e^{-(\lambda_{\mathrm{m}k}+s)x_k} \\
&\quad (1 - e^{-(\lambda_{\mathrm{m}0}+s)x_0}) \frac{\lambda_{\mathrm{m}0}}{\lambda_{\mathrm{m}0}+s} e^{-sz_0} \\
&\quad \prod_{i=1}^{k-1} \Bigg((1 - e^{-y_i(\lambda_{\mathrm{c}i}+s)}) \frac{\lambda_{\mathrm{c}i}}{\lambda_{\mathrm{c}i}+s} \\
&\quad + (e^{-y_i(\lambda_{\mathrm{c}i}+s)} - e^{-(y_i(\lambda_{\mathrm{c}i}+s)+x_i(\lambda_{\mathrm{m}i}+s))}) \frac{\lambda_{\mathrm{m}i}}{\lambda_{\mathrm{m}i}+s} \Bigg) e^{-sz_i}
\end{aligned} \tag{4.61}$$

Removing the time condition yields

$$\begin{aligned}
&\mathbf{G}(s) \\
&= \mathbf{M}_0(\lambda_{\mathrm{m}0} + s) \\
&\quad + \sum_{k=1}^{N} \mathbf{C}_k(\lambda_{\mathrm{c}k} + s) \mathbf{M}_k(\lambda_{\mathrm{m}k} + s) \\
&\quad (1 - \mathbf{M}_0(\lambda_{\mathrm{m}0} + s)) \frac{\lambda_{\mathrm{m}0}}{\lambda_{\mathrm{m}0}+s} \mathbf{D}_0(s) \\
&\quad \prod_{i=1}^{k-1} \Bigg((1 - \mathbf{C}_i(\lambda_{\mathrm{c}i} + s)) \frac{\lambda_{\mathrm{c}i}}{\lambda_{\mathrm{c}i}+s} \\
&\quad + (\mathbf{C}_i(\lambda_{\mathrm{c}i} + s) - \mathbf{C}_i(\lambda_{\mathrm{c}i} + s) \mathbf{M}_i(\lambda_{\mathrm{m}i} + s)) \frac{\lambda_{\mathrm{m}i}}{\lambda_{\mathrm{m}i}+s} \Bigg) \mathbf{D}_i(s)
\end{aligned} \tag{4.62}$$

Equation 4.62 shows the moment generating function of the runtime of a task in an inhomogeneous, passive replication system with a watchdog fault detection mechanism, where the fault rates may differ among the parts of the

system, and where more than N faults occurring to a single task leads to the failure of the system.

If no more than N faults can occur to the same task, the probability function for the number of faults changes to

$$\begin{aligned}
&\Pr[\phi = k] \\
&= \begin{cases}
e^{-\lambda_{\mathrm{m}0} x_0} & , k = 0 \\
e^{-(\lambda_{\mathrm{c}k} y_k + \lambda_{\mathrm{m}k} x_k)} & \\
\qquad (1 - e^{-\lambda_{\mathrm{m}0} x_0}) \prod_{i=1}^{k-1} (1 - e^{-(\lambda_{\mathrm{c}i} y_i + \lambda_{\mathrm{m}i} x_i)}) & , 0 < k < N \\
(1 - e^{-\lambda_{\mathrm{m}0} x_0}) \prod_{i=1}^{N-1} (1 - e^{-(\lambda_{\mathrm{c}i} y_i + \lambda_{\mathrm{m}i} x_i)}) & , k = N
\end{cases}
\end{aligned} \tag{4.63}$$

Using this fault probability function to remove the condition on the number of faults from the expectation function 4.60 yields

$$\begin{aligned}
&E[e^{-sY} | r_n = x_n, c_n = y_n, d_n = z_n] \\
&= e^{-(\lambda_{\mathrm{m}0} + s) x_0} \\
&\quad + \Bigg(\sum_{k=1}^{N-1} e^{-(\lambda_{\mathrm{c}k} + s) y_k} e^{-(\lambda_{\mathrm{m}k} + s) x_k} \\
&\qquad (1 - e^{-(\lambda_{\mathrm{m}0} + s) x_0}) \frac{\lambda_{\mathrm{m}0}}{\lambda_{\mathrm{m}0} + s} e^{-s z_0} \\
&\qquad \prod_{i=1}^{k-1} \Bigg((1 - e^{-y_i (\lambda_{\mathrm{c}i} + s)}) \frac{\lambda_{\mathrm{c}i}}{\lambda_{\mathrm{c}i} + s} \\
&\qquad\quad + (e^{-y_i (\lambda_{\mathrm{c}i} + s)} - e^{-(y_i (\lambda_{\mathrm{c}i} + s) + x_i (\lambda_{\mathrm{m}i} + s))}) \frac{\lambda_{\mathrm{m}i}}{\lambda_{\mathrm{m}i} + s} \Bigg) e^{-s z_i} \Bigg) \\
&\quad + e^{-s(x_N + y_N)} \\
&\qquad (1 - e^{-(\lambda_{\mathrm{m}0} + s) x_0}) \frac{\lambda_{\mathrm{m}0}}{\lambda_{\mathrm{m}0} + s} e^{-s z_0} \\
&\qquad \prod_{i=1}^{N-1} \Bigg((1 - e^{-y_i (\lambda_{\mathrm{c}i} + s)}) \frac{\lambda_{\mathrm{c}i}}{\lambda_{\mathrm{c}i} + s} \\
&\qquad\quad + (e^{-y_i (\lambda_{\mathrm{c}i} + s)} - e^{-(y_i (\lambda_{\mathrm{c}i} + s) + x_i (\lambda_{\mathrm{m}i} + s))}) \frac{\lambda_{\mathrm{m}i}}{\lambda_{\mathrm{m}i} + s} \Bigg) e^{-s z_i}
\end{aligned} \tag{4.64}$$

Finally, removing the conditions on time yields the mgf

$$
\begin{aligned}
&\tilde{\mathbf{G}}(s) \\
&= \mathbf{M}_0(\lambda_{\mathrm{m}0}+s) \\
&\quad + \Bigg(\sum_{k=1}^{N} \mathbf{C}_k(\lambda_{\mathrm{c}k}+s)\mathbf{M}_k(\lambda_{\mathrm{m}k}+s) \\
&\qquad (1-\mathbf{M}_0(\lambda_{\mathrm{m}0}+s))\frac{\lambda_{\mathrm{m}0}}{\lambda_{\mathrm{m}0}+s}\mathbf{D}_0(s) \\
&\qquad \prod_{i=1}^{k-1} \Bigg((1-\mathbf{C}_i(\lambda_{\mathrm{c}i}+s))\frac{\lambda_{\mathrm{c}i}}{\lambda_{\mathrm{c}i}+s} \\
&\qquad + (\mathbf{C}_i(\lambda_{\mathrm{c}i}+s) - \mathbf{C}_i(\lambda_{\mathrm{c}i}+s)\mathbf{M}_i(\lambda_{\mathrm{m}i}+s))\frac{\lambda_{\mathrm{m}i}}{\lambda_{\mathrm{m}i}+s} \Bigg) \mathbf{D}_i(s) \Bigg) \\
&\quad + \mathbf{C}_N(s)\mathbf{M}_N(s) \\
&\qquad (1-\mathbf{M}_0(\lambda_{\mathrm{m}0}+s))\frac{\lambda_{\mathrm{m}0}}{\lambda_{\mathrm{m}0}+s}\mathbf{D}_0(s) \\
&\qquad \prod_{i=1}^{k-1} \Bigg((1-\mathbf{C}_i(\lambda_{\mathrm{c}i}+s))\frac{\lambda_{\mathrm{c}i}}{\lambda_{\mathrm{c}i}+s} \\
&\qquad + (\mathbf{C}_i(\lambda_{\mathrm{c}i}+s) - \mathbf{C}_i(\lambda_{\mathrm{c}i}+s)\mathbf{M}_i(\lambda_{\mathrm{m}i}+s))\frac{\lambda_{\mathrm{m}i}}{\lambda_{\mathrm{m}i}+s} \Bigg) \mathbf{D}_i(s)
\end{aligned}
\tag{4.65}
$$

Equation 4.65 shows the moment generating function of the runtime distribution for a task in an inhomogeneous passive replication system with a watchdog-based fault detection mechanism, where the fault rates differ between the parts of the system, and where no more than N faults may occur to the task.

4.3 Systems with timeout as a fault detection method

In this section, expressions for the timing in systems where timeout of the method return is used as a fault detection method are derived. Unlike the watchdog fault detection, the faults are not detected by checking the components while they run. Instead, a component is considered as failed if the results from running a task on the component do not arrive before a timeout. Normally, this causes the faults to be detected at a later time than for the watchdog mechanism, thus the execution of a task is slower in the case of a fault, however,

the implementation of this fault detection method is fairly simple, and the mechanism covers both silent and omission failures.

The naming of the distributions used in the derivation of the expressions in this section is the same as for the systems using watchdogs as fault detection, with the exception of the time to fault and fault detection times that are added together to form the distribution $a_{\text{to}i}(t)$, which in these equations is a dirac pulse at the timeout for replica i.

It should be noted that the expressions derived in this section are only valid if the worst case execution time for running a task on a replica is less than or equal to the time set as the timeout, i.e., it is assumed that the only way the timeout can be triggered is in the case of a failed replica.

4.3.1 The fault models

For the timeout system, two different fault models are used. The first is the poisson process model, the same model as used in the watchdog system.

The other fault model is a fixed fault probability model, where a replica (or correction process) has a fixed probability of failing. Unlike the models where a watchdog fault detector is assumed, it is not necessary to know the exact time when a fault occurs in the timeout based models, so it is possible to use the fixed fault probability model.

Faults generated by a poisson process

As in the previous section, faults are modelled as generated from a poisson process with intensity λ. If the execution of a method takes the time τ the probability that at least one fault happens before the execution time ends is given by equation 4.3

If a fault occurs to the running task, causing the active component to fail, this will be detected at the timeout, τ_i for component i. The distribution of this time can therefore be expressed with a dirac pulse as the pdf.

$$a_{\text{to}i}(t) = \delta(t - \tau_i) \tag{4.66}$$

This distribution has the moment generating function

$$\mathbf{A}_{\text{to}i}(s) = e^{-\tau_i s} \tag{4.67}$$

If faults may occur during the correction of a task, and there is a timeout function on the correction, the model must reflect the possibility of two timeouts,

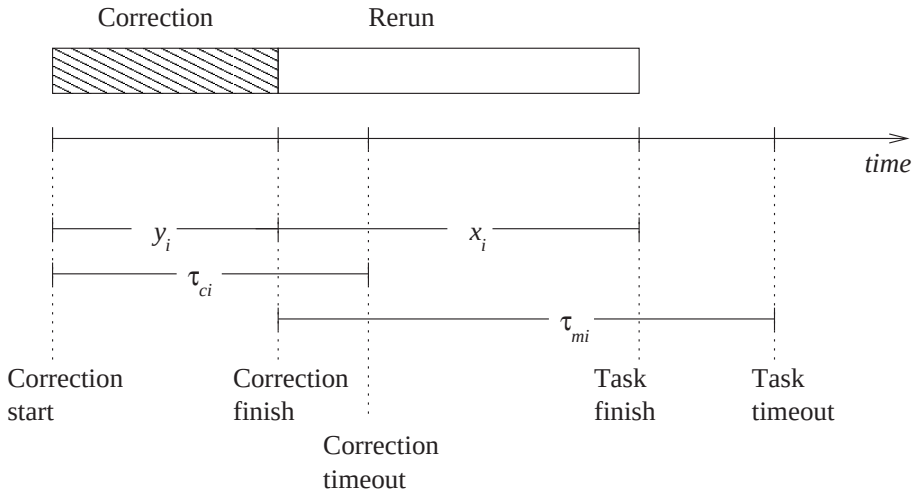


Figure 4.2: Correction and rerun of a task in a system with timeout detection. Faults during correction are detected at the correction timeout, faults during the rerun of the task are detected at the task timeout.

one for correction and one for the actual run. Figure 4.2 shows the correction and rerun timing with the normal, fault-free times and the timeouts.

If the timeout for correction of component i is set to τ_{ci}, the normal, non-faulty correction time is y_i, the timeout for the actual runtime of a component is τ_{mi} after the start of the run, and the normal, non-faulty runtime of the component is x_i, given that at least one fault occurs, the distribution for the detection time is given by the pdf

$$a_{toi}(t) = \frac{1-e^{-\lambda y_i}}{1-e^{-\lambda(y_i+x_i)}}\delta(t-\tau_{ci}) + \frac{1-e^{-\lambda x_i}}{1-e^{-\lambda(y_i+x_i)}}e^{-\lambda y_i}\delta(t-(y_i+\tau_{mi})) \tag{4.68}$$

This distribution has the mgf

$$\mathbf{A}_{toi}(t) = \frac{1-e^{-\lambda y_i}}{1-e^{-\lambda(y_i+x_i)}}e^{-s\tau_{ci}} + \frac{1-e^{-\lambda x_i}}{1-e^{-\lambda(y_i+x_i)}}e^{-(\lambda+s)y_i}e^{-s\tau_{mi}} \tag{4.69}$$

A fixed fault probability for each replica

As the exact time when the fault occurs is not needed in this model, it is also possible to use a fault model that only specifies the probability that a replica

has failed.

If $\kappa_{\mathrm{m}i}$ is the probability that the task running on replica i is correct, and $\kappa_{\mathrm{c}i}$ is the probability that updating and preparing replica i is correct, the probability that at least one fault occurs in replica i is given by

$$\Pr(\phi \geq 1) = 1 - \kappa_{\mathrm{c}i}\kappa_{\mathrm{m}i} \tag{4.70}$$

For a system where there can be no faults in (or where there is no timeout detector for) the correction of a task, equations 4.66–4.67 can be used as they are for the detection time.

If faults may occur during correction, and there is a timeout function on the correction, the distribution for the detection time is given by the pdf

$$a_{\mathrm{to}i}(t) = \frac{1-\kappa_{\mathrm{c}i}}{1-\kappa_{\mathrm{c}i}\kappa_{\mathrm{m}i}}\delta(t-\tau_{\mathrm{c}i}) + \frac{(1-\kappa_{\mathrm{m}i})\kappa_{\mathrm{c}i}}{1-\kappa_{\mathrm{c}i}\kappa_{\mathrm{m}i}}\delta(t-(y_i+\tau_{\mathrm{m}i})) \tag{4.71}$$

and the corresponding mgf

$$\mathbf{A}_{\mathrm{to}i}(s) = \frac{1-\kappa_{\mathrm{c}i}}{1-\kappa_{\mathrm{c}i}\kappa_{\mathrm{m}i}}e^{-s\tau_{\mathrm{c}i}} + \frac{(1-\kappa_{\mathrm{m}i})\kappa_{\mathrm{c}i}}{1-\kappa_{\mathrm{c}i}\kappa_{\mathrm{m}i}}e^{-sy_i}e^{-s\tau_{\mathrm{m}i}} \tag{4.72}$$

4.3.2 Inhomogeneous systems

The runtime distribution for the systems with timeout as a fault detection method is derived the same way as for the systems with watchdog fault detection. Instead of using a fault detection delay after a fault occurs, a fault will always be detected at the timeout, regardless of when the fault occurs. Thus, equations 4.11–4.16 can be used as they are for a system where there is no fault correction delay.

Using equation 4.16 and $\mathrm{E}[e^{-sX_i}] = \mathbf{A}_{\mathrm{to}i}(s)$, where $\mathbf{A}_{\mathrm{to}i}(s)$ is defined in equation 4.67, we get

$$\mathrm{E}[e^{-sY}|r_\phi = x_\phi, \phi = k] = \mathrm{E}[e^{-sx_k}]\prod_{i=0}^{k-1}\mathbf{A}_{\mathrm{to}i}(s) \tag{4.73}$$

Removing the condition on the number of faults from this expectation function, using the fault probability function 4.11, yields the new expectation function

$$\mathrm{E}[e^{-sY}|r_n = x_n] = e^{-(s+\lambda)x_0} + \sum_{k=1}^{\infty} e^{-(s+\lambda)x_k}\prod_{i=0}^{k-1} e^{-\tau_i s}(1-e^{-\lambda x_i}) \tag{4.74}$$

For a system that fails if the number of faults occurring to a single task is larger than N, removing the time conditions gives a runtime distribution with the moment-generating function

$$\mathbf{G}(s) = \mathbf{M}_0(\lambda + s) + \sum_{k-1}^{N} \mathbf{M}_k(\lambda + s) \prod_{i=0}^{k-1} e^{-\tau_i s}(1 - \mathbf{M}_i(\lambda)) \tag{4.75}$$

Equation 4.75 shows the moment-generating function for the runtime distribution in a passive replication system with inhomogeneous replicas where the fault detection is based on timeouts, there is no extra time used for fault correction, and where more than N faults to a single task lead to the failure of the system.

If there can be no more than N faults, the fault probability function 4.23 is used to remove the condition on the number of faults, resulting in the expectation function

$$\begin{aligned} &\mathrm{E}[e^{-sY} | r_n = x_n] = \\ &e^{-(s+\lambda)x_0} + \left(\sum_{k=1}^{N-1} e^{-(s+\lambda)x_k} \prod_{i=0}^{k-1} e^{-\tau_i s}(1 - e^{-\lambda x_i}) \right) \\ &+ e^{-sx_N} \prod_{i=0}^{N-1} e^{-\tau_i s}(1 - e^{-\lambda x_i}) \end{aligned} \tag{4.76}$$

Removing the conditions on time use gives the moment-generating function

$$\begin{aligned} \tilde{\mathbf{G}}(s) = \mathbf{M}_0(\lambda + s) &+ \left(\sum_{k=1}^{N-1} \mathbf{M}_k(\lambda + s) \prod_{i=0}^{k-1} e^{-\tau_i s}(1 - \mathbf{M}_i(\lambda)) \right) \\ &+ \mathbf{M}_N(s) \prod_{i=0}^{N-1} e^{-\tau_i s}(1 - \mathbf{M}_i(\lambda)) \end{aligned} \tag{4.77}$$

Equation 4.77 shows the moment-generating function for the runtime distribution in an inhomogeneous passive replication system where the fault detection is based on timeouts, there is no extra time used for fault correction, and where there can be no more than N faults occurring to a single task.

Adding the fault correction delay

As explained in 4.3.1, the expression for time to fault detection gets somewhat more complicated when fault correction is added, faults may occur in both the correction and rerun part of the task execution, and there is a timeout function

both on the correction part and the rerun part. If there is no separate timeout function for the correction part, equation 4.75 or 4.77 can be used by substituting $\mathbf{M}_i(s)$ with $\mathbf{M}_i(s)\mathbf{C}_i(s)$.

As there is no correction of the primary, equation 4.67 is used for the distribution of the time to fault detection for the primary, $\mathbf{A}_{\mathrm{to}0}$. The equation 4.69 is used for the distribution of the time to fault detection for the backups.

The expectation function with condition to number of faults and the time use is similar to the equation 4.27:

$$\mathrm{E}[e^{-sY}|r_n = x_n, c_n = y_n, \phi = k] = \begin{cases} e^{-sx_0} & , k = 0 \\ e^{-s(x_k+y_k)}\mathbf{A}_{\mathrm{to}0}(s)\prod_{i=1}^{k-1}\mathbf{A}_{\mathrm{to}i}(s) & , k > 0 \end{cases} \tag{4.78}$$

Using the fault probability function 4.26 to remove the condition on the number of faults in 4.78 yields the expectation function

$$\begin{aligned} &\mathrm{E}[e^{-sY}|r_n = x_n, c_n = y_n] = \\ &e^{-(\lambda+s)x_0} \\ &+\sum_{k=1}^{\infty} e^{-(\lambda+s)(x_k+y_k)} \\ &e^{-s\tau_{\mathrm{m}0}}(1-e^{-\lambda x_0}) \\ &\prod_{i=1}^{k-1} e^{-s\tau_{ci}}(1-e^{-\lambda y_i}) + e^{-s\tau_{\mathrm{m}i}}e^{-(\lambda+s)y_i}(1-e^{-\lambda x_i}) \end{aligned} \tag{4.79}$$

Limiting the number of faults before failure to N, and removing the time use conditions yields the moment-generating function

$$\begin{aligned} &\mathbf{G}(s) = \\ &\mathbf{M}_0(\lambda+s) \\ &+\sum_{k=1}^{N} \mathbf{M}_k(\lambda+s)\mathbf{C}_k(\lambda+s) \\ &e^{-s\tau_{\mathrm{m}0}}(1-\mathbf{M}_0(\lambda)) \\ &\prod_{i=1}^{k-1} e^{-s\tau_{ci}}(1-\mathbf{C}_i(\lambda)) + e^{-s\tau_{\mathrm{m}i}}\mathbf{C}_i(\lambda+s)(1-\mathbf{M}_i(\lambda)) \end{aligned} \tag{4.80}$$

Equation 4.80 shows the moment-generating function for the runtime distribution in a passively replicated system with timeout fault detection and inhomogeneous replicas, where more than N faults to a single task lead to the failure of the system.

For the model where no more than N faults may occur to a single task, the fault probability function 4.30 is used to remove the condition on number of faults from the expectation function 4.78. The resulting expectation function becomes

$$\begin{aligned}
&\mathrm{E}[e^{-sY}|r_n = x_n, c_n = y_n] = \\
&e^{-(\lambda+s)x_0} \\
&+ \Bigg(\sum_{k=1}^{N-1} e^{-(\lambda+s)(x_k+y_k)} \\
&e^{-s\tau_{\mathrm{m}0}}(1-e^{-\lambda x_0}) \\
&\prod_{i=1}^{k-1} e^{-s\tau_{\mathrm{c}i}}(1-e^{-\lambda y_i}) + e^{-s\tau_{\mathrm{m}i}} e^{-(\lambda+s)y_i}(1-e^{-\lambda x_i}) \Bigg) \\
&+ e^{-s(x_N+y_N)} \\
&e^{-s\tau_{\mathrm{m}0}}(1-e^{-\lambda x_0}) \\
&\prod_{i=1}^{N-1} e^{-s\tau_{\mathrm{c}i}}(1-e^{-\lambda y_i}) + e^{-s\tau_{\mathrm{m}i}} e^{-(\lambda+s)y_i}(1-e^{-\lambda x_i})
\end{aligned} \tag{4.81}$$

The condition on time use can then be removed, giving the moment generating function

$$\begin{aligned}
&\tilde{\mathbf{G}}(s) = \\
&\mathbf{M}_0(\lambda+s) \\
&+ \Bigg(\sum_{k=1}^{N-1} \mathbf{M}_k(\lambda+s)\mathbf{C}_k(\lambda+s) \\
&e^{-s\tau_{\mathrm{m}0}}(1-\mathbf{M}_0(\lambda)) \\
&\prod_{i=1}^{k-1} e^{-s\tau_{\mathrm{c}i}}(1-\mathbf{C}_i(\lambda)) + e^{-s\tau_{\mathrm{m}i}}\mathbf{C}_i(\lambda+s)(1-\mathbf{M}_i(\lambda)) \Bigg) \\
&+ \mathbf{M}_N(s)\mathbf{C}_N(s) \\
&e^{-s\tau_{\mathrm{m}0}}(1-\mathbf{M}_0(\lambda)) \\
&\prod_{i=1}^{k-1} e^{-s\tau_{\mathrm{c}i}}(1-\mathbf{C}_i(\lambda)) + e^{-s\tau_{\mathrm{m}i}}\mathbf{C}_i(\lambda+s)(1-\mathbf{M}_i(\lambda))
\end{aligned} \tag{4.82}$$

Equation 4.82 shows the moment-generating function for an inhomogeneous passive replication system with timeout as a fault detection method, where there can be no more than N faults occurring to a single task.

There is no extra detection time that needs to be added.

4.3.3 Homogeneous systems

The runtime distribution of a homogeneous system using timeout as a means for fault correction is derived in a similar way as the homogeneous watchdog-based system.

For a system with no extra time used for fault correction, equations 4.36–4.39 from the homogeneous watchdog-based system can be used as they are, but with X_i symbolizing the time to fault detection instead of the time to fault.

Using the time to fault detection distribution from equation 4.67, the expectation function 4.39 can be written as

$$\mathrm{E}[e^{-sY}|r = x, \phi = k] = \mathrm{E}[e^{-sx}](\mathbf{A_{to}}_i(s))^k \tag{4.83}$$

for the timeout system.

Removing the conditions on the number of faults from this expectation function, using the fault probability function 4.36, yields the new expectation function

$$\mathrm{E}[e^{-sY}|r = x] = \sum_{k=0}^{\infty} e^{-(s+\lambda)x} e^{-k\tau_{\mathrm{m}}s}(1 - e^{\lambda x})^k \tag{4.84}$$

By limiting the number of faults before failure to N and removing the timing condition, the moment-generating function for the runtime distribution is found

$$\mathbf{G}(s) = \sum_{k=0}^{N} e^{-k\tau_{\mathrm{m}}s} \left(\sum_{i=0}^{k-1} (-1)^i \binom{k}{i} \mathbf{M}((i+1)\lambda + s) \right) \tag{4.85}$$

Equation 4.85 shows the moment-generating function for the runtime distribution of a task in a passive replication system with homogeneous replicas and timeout-based fault detection, where there is no time used for correction, and where the system will fail if more than N faults occur to a single task.

If there can be no more than N faults occurring to a single task, the fault probability function 4.46 is used to remove the condition on the number of faults, yielding the expectation function

$$\begin{aligned} &\mathrm{E}[e^{-sY}|r = x] = \\ &\left(\sum_{k=0}^{N-1} e^{-(s+\lambda)x} e^{-k\tau_{\mathrm{m}}s}(1 - e^{\lambda x})^k \right) + e^{-sx} e^{-N\tau_{\mathrm{m}}s}(1 - e^{\lambda x})^N \end{aligned} \tag{4.86}$$

The moment generating function is found by removing the condition on the number of faults:

$$\begin{aligned}&\tilde{\mathbf{G}}(s) = \\ &\left(\sum_{k=0}^{N-1} e^{-k\tau_{\mathrm{m}}s}\left(\sum_{i=0}^{k-1}(-1)^i\binom{k}{i}\mathbf{M}((i+1)\lambda+s)\right)\right) \\ &\quad + e^{-N\tau_{\mathrm{m}}s}\left(\sum_{i=0}^{N-1}(-1)^i\binom{k}{i}\mathbf{M}(i\lambda+s)\right)\end{aligned} \tag{4.87}$$

Equation 4.87 shows the moment-generating function for the runtime distribution in a passive replication system with homogeneous replicas and timeout-based fault detection, where there is no extra time used for correction, and where no more than N faults may occur to a single task.

Adding the fault correction delay

As with the model for heterogeneous systems, a separate timeout for the correction process is assumed. Equation 4.67 is used for the timeout function for the primary, and is given the name $\mathbf{A}_{\mathrm{to0}}(s)$ here. Equation 4.69 is used for the timeout function for the backups, and is given the name $\mathbf{A}_{\mathrm{to}}(s)$ here.

When the time for fault correction is added, the expectation function with conditions to the times and number of faults becomes

$$\begin{aligned}&\mathrm{E}[e^{-sY}|r=x,c=y,\phi=k] = \\ &\begin{cases} e^{-sx} & ,k=0 \\ e^{-s(x+y)}\mathbf{A}_{\mathrm{to0}}(s)(\mathbf{A}_{\mathrm{to}}(s))^{k-1} & ,k>0 \end{cases}\end{aligned} \tag{4.88}$$

Removing the condition on the number of faults, using the fault probability function 4.49, yields the expectation function

$$\begin{aligned}&\mathrm{E}[e^{-sY}|r=x,c=y] = \\ &e^{-(\lambda+s)x} \\ &+\sum_{k=1}^{\infty} e^{-(\lambda+s)(x+y)}(1-e^{-\lambda x})e^{-\tau_{\mathrm{m}}s} \\ &((1-e^{-\lambda y})e^{-\tau_{\mathrm{c}}s}+(1-e^{-\lambda x})e^{-(\lambda+s)y}e^{-\tau_{\mathrm{m}}s})^{k-1}\end{aligned} \tag{4.89}$$

Limiting the number of faults before failure to N and removing the condition on the runtime and correction time gives the final moment generating function

for the total runtime distribution in this system:

$$\begin{aligned}
&\mathbf{G}(s) = \\
&\quad \mathbf{M}(\lambda + s) \\
&\quad + \sum_{k=1}^{N} \sum_{j=0}^{k-1} \binom{k-1}{j} e^{-(k-j)\tau_{\mathrm{m}} s} e^{-j\tau_{\mathrm{c}} s} \\
&\quad \left(\sum_{i=0}^{k-j} (-1)^i \binom{k-j}{i} \mathbf{M}((i+1)\lambda + s) \right) \\
&\quad \left(\sum_{i=0}^{j} (-1)^i \binom{j}{i} \mathbf{C}((i+k-j)\lambda + (k-j)s) \right)
\end{aligned} \tag{4.90}$$

Equation 4.90 shows the mgf for the distribution of the runtime of a task in a homogeneous passive replication system based on a timeout fault detection method, and where more than N faults in a single task lead to the failure of the system.

For a model where any faults after the Nth are ignored, the fault probability function 4.53 is used to remove the condition on the number of faults, resulting in the expectation function

$$\begin{aligned}
&\mathrm{E}[e^{-sY} | r = x, c = y] = \\
&\quad e^{-(\lambda+s)x} \\
&\quad + \left(\sum_{k=1}^{N-1} e^{-(\lambda+s)(x+y)} (1 - e^{-\lambda x}) e^{-\tau_{\mathrm{m}} s} \right. \\
&\quad \left. ((1 - e^{-\lambda y}) e^{-\tau_{\mathrm{c}} s} + (1 - e^{-\lambda x}) e^{-(\lambda+s)y} e^{-\tau_{\mathrm{m}} s})^{k-1} \right) \\
&\quad + e^{-s(x+y)} (1 - e^{-\lambda x}) e^{-\tau_{\mathrm{m}} s} \\
&\quad ((1 - e^{-\lambda y}) e^{-\tau_{\mathrm{c}} s} + (1 - e^{-\lambda x}) e^{-(\lambda+s)y} e^{-\tau_{\mathrm{m}} s})^{N-1}
\end{aligned} \tag{4.91}$$

Finally, removing the condition on time use from the expectation function

yields

$$
\begin{aligned}
&\tilde{\mathbf{G}}(s) = \\
&\quad \mathbf{M}(\lambda + s) \\
&\quad + \left(\sum_{k=1}^{N-1} \sum_{j=0}^{k-1} \binom{k-1}{j} e^{-(k-j)\tau_{\mathrm{m}} s} e^{-j\tau_{\mathrm{c}} s} \right. \\
&\quad \left(\sum_{i=0}^{k-j} (-1)^i \binom{k-j}{i} \mathbf{M}((i+1)\lambda + s) \right) \\
&\quad \left. \left(\sum_{i=0}^{j} (-1)^i \binom{j}{i} \mathbf{C}((i+k-j)\lambda + (k-j)s) \right) \right) \\
&\quad + \sum_{j=0}^{N-1} \binom{N-1}{j} e^{-(N-j)\tau_{\mathrm{m}} s} e^{-j\tau_{\mathrm{c}} s} \\
&\quad \left(\sum_{i=0}^{N-j} (-1)^i \binom{N-j}{i} \mathbf{M}(i\lambda + s) \right) \\
&\quad \left(\sum_{i=0}^{j} (-1)^i \binom{j}{i} \mathbf{C}((i+N-j-1)\lambda + (N-j)s) \right)
\end{aligned}
\tag{4.92}
$$

Equation 4.92 shows the mgf for the distribution of the runtime of a task in a homogeneous passive replication system where fault detection is based on timeouts, and there can be no more than N faults to a single task.

4.3.4 Using a fixed fault probability

The derivation for the system with a fixed fault probability for each replica is derived as in 4.3.2, however, the following probability function for the number of faults is used instead of 4.11.

$$
\Pr[\phi = k] = \begin{cases} \kappa_{\mathrm{m}0} & , k = 0 \\ \kappa_{\mathrm{m}k} \prod_{i=0}^{k-1} (1 - \kappa_{\mathrm{m}i}) & , k > 0 \end{cases} \tag{4.93}
$$

Using this probability function to remove the condition on the number of faults in equation 4.73, yields the expectation function

$$
\mathrm{E}[e^{-sY} | r_n = x_n] = \kappa_{\mathrm{m}0} e^{-sx_0} + \sum_{k=1}^{\infty} \kappa_{\mathrm{m}k} e^{-sx_k} \prod_{i=0}^{k-1} e^{-\tau_i s} (1 - \kappa_{\mathrm{m}i}) \tag{4.94}
$$

If more than N faults lead to failure, removing the timing conditions gives a runtime distribution with the moment generating function

$$\mathbf{G}(s) = \kappa_{\mathrm{m}0}\mathbf{M}_0(s) + \sum_{k=1}^{N} \kappa_{\mathrm{m}k}\mathbf{M}_k(s) \prod_{i=0}^{k-1} e^{-\tau_i s}(1-\kappa_{\mathrm{m}i}) \tag{4.95}$$

Equation 4.95 shows the moment generating function for an inhomogeneous passive replication system based on a timeout fault detection method, where a fixed fault probability for each replica is used instead of a poisson fault process, and where there is no extra time used for fault correction.

If there can be no more than N faults, the moment generating function can be found by letting $\kappa_{\mathrm{m}N} = 1$ in equation 4.95.

Adding the fault correction delay

In systems where there is a possibility of faults during correction and a timeout function on the correction part of the system, the expectation function 4.78 is used, but with the distribution for the time to detection, $\mathbf{A}_{\mathrm{to}i}(s)$ taken from equation 4.72 for $i > 0$.

The fault probability function for this system is

$$\Pr[\phi = k] = \begin{cases} \kappa_{\mathrm{m}0} & , k = 0 \\ \kappa_{\mathrm{m}k}\kappa_{\mathrm{c}k}(1-\kappa_{\mathrm{m}0})\prod_{i=1}^{k-1}(1-\kappa_{\mathrm{m}i}\kappa_{\mathrm{c}i}) & , k > 0 \end{cases} \tag{4.96}$$

Removing the condition on the number of fault yields the expectation function

$$\begin{aligned} &\mathrm{E}[e^{-sY} | r_n = x_n, c_n = y_n] = \\ &\kappa_{\mathrm{m}0}e^{-sx_0} \\ &+ \sum_{k=1}^{\infty} \kappa_{\mathrm{m}k}\kappa_{\mathrm{c}k}e^{-s(x_k+y_k)}(1-\kappa_{\mathrm{m}0})e^{-s\tau_{\mathrm{m}0}} \\ &\prod_{i=1}^{k-1}(1-\kappa_{\mathrm{c}i})e^{-s\tau_{\mathrm{c}i}} + (1-\kappa_{\mathrm{m}i})\kappa_{\mathrm{c}i}e^{-sy_i}e^{-s\tau_{\mathrm{m}i}} \end{aligned} \tag{4.97}$$

Limiting the number of faults before failure to N and removing the condition

on time use yields the moment generation function

$$\begin{aligned}&\mathbf{G}(s) = \\ &\kappa_{\mathrm{m}0}\mathbf{M}_0(s) \\ &+\sum_{k=1}^{N}\kappa_{\mathrm{m}k}\kappa_{\mathrm{c}k}\mathbf{M}_k(s)\mathbf{C}_k(s)(1-\kappa_{\mathrm{m}0})e^{-s\tau_{\mathrm{m}0}} \\ &\prod_{i=1}^{k-1}(1-\kappa_{\mathrm{c}i})e^{-s\tau_{\mathrm{c}i}} + (1-\kappa_{\mathrm{m}i})\kappa_{\mathrm{c}i}\mathbf{C}_i(s)e^{-s\tau_{\mathrm{m}i}}\end{aligned} \tag{4.98}$$

Equation 4.98 shows the moment generating function of the runtime of a task in a fault tolerant system where the fault detection is based on timeouts, where the fault probability for running and correction of the task on each replica is fixed, and where more than N faults occurring to a single task will lead to the failure of the system.

If there can be no more than N faults in the system, the moment-generating function can be found by setting $\kappa_{\mathrm{m}N} = \kappa_{\mathrm{c}N} = 1$ in equation 4.98.

4.4 Systems with acceptance failure detection

In this section, expressions for the runtime distributions in systems where acceptance tests are used as a fault detection method are derived.

In these systems, a method implementing a task runs to completion, and the results are then tested. If the results pass the test, the task is finished, if they fail the test, a value failure is assumed, and the task is rerun.

As the methods runs to completion regardless of the faults, no separate time to failure or time to detection distributions is needed. However, the time used to test the results needs to be modelled. The distribution of the test time is given the moment-generating function $\mathbf{D}(s)$. As with the timeout systems, the exact time of when a fault occurs is not modelled, so both the poisson process fault model and the fixed fault probability model can be used.

4.4.1 Inhomogeneous systems

Using the poisson process fault model, the probability that faults occur to k replicas is given by the probability function 4.26

A task where faults are detected in ϕ replicas will have the total runtime

$$Y = r_0 + d + \sum_{i=1}^{\phi} c_i + r_i + d \tag{4.99}$$

where r_i is the runtime for replica i, d is the test time and c_i is the time used to get replica i ready to run.

As before, we set up an expectation function for e^{-sY} with conditions to time use and number of faults

$$\begin{aligned}&\mathrm{E}[e^{-sY}|r_n = x_n, c_n = y_n, d = z, \phi = k] = \\ &\begin{cases} e^{-sx_0}e^{-sz} & , k = 0 \\ e^{-sx_0}e^{-sz}\prod_{i=1}^{k} e^{-s(y_i+x_i)}e^{-sz} & , k > 0 \end{cases}\end{aligned} \tag{4.100}$$

Removing the condition on the number of faults, using the fault probability function 4.26 yields the expectation function

$$\begin{aligned}&\mathrm{E}[e^{-sY}|r_n = x_n, c_n = y_n, d = z] = \\ &\quad e^{-(\lambda+s)x_0}e^{-sz} \\ &\quad + \sum_{k=1}^{\infty} e^{-s(k+1)z}e^{-(\lambda+s)(x_k+y_k)} \\ &\quad (e^{-sx_0} - e^{-(\lambda+s)x_0})\prod_{i=1}^{k-1}(e^{-s(x_i+y_i)} - e^{-(\lambda+s)(x_i+y_i)})\end{aligned} \tag{4.101}$$

Limiting the number of faults before failure to N and removing the conditions on time use gives the moment generating function

$$\begin{aligned}&\mathbf{G}(s) = \\ &\quad \mathbf{M}_0(\lambda+s)\mathbf{D}(s) \\ &\quad + \sum_{k=1}^{N} \mathbf{M}_k(\lambda+s)\mathbf{C}_k(\lambda+s)\mathbf{D}((k+1)s) \\ &\quad (\mathbf{M}_0(s) - \mathbf{M}_0(\lambda+s))\prod_{i=1}^{k-1}(\mathbf{M}_i(s)\mathbf{C}_i(s) - \mathbf{M}_i(\lambda+s)\mathbf{C}_i(\lambda+s))\end{aligned} \tag{4.102}$$

Equation 4.102 shows the moment generating function of the runtime distribution in an acceptance test based passive replication system, where more than N faults to a single task lead to the failure of the system.

If the number of faults that can occur to a single task is limited to N, the fault probability function 4.30 is used to remove the condition on number of

faults from the expectation function 4.100, resulting in the expectation function

$$\begin{aligned}
&\mathrm{E}[e^{-sY}|r_n = x_n, c_n = y_n, d = z] = \\
&\quad e^{-(\lambda+s)x_0}e^{-sz} \\
&\quad + \left(\sum_{k=1}^{N-1} e^{-s(k+1)z}e^{-(\lambda+s)(x_k+y_k)}\right. \\
&\quad \left.(e^{-sx_0} - e^{-(\lambda+s)x_0})\prod_{i=1}^{k-1}(e^{-s(x_i+y_i)} - e^{-(\lambda+s)(x_i+y_i)})\right) \\
&\quad + e^{-s(N+1)z}e^{-s(x_N+y_N)} \\
&\quad (e^{-sx_0} - e^{-(\lambda+s)x_0})\prod_{i=1}^{N-1}(e^{-s(x_i+y_i)} - e^{-(\lambda+s)(x_i+y_i)})
\end{aligned} \tag{4.103}$$

Removing the conditions on time use yields the moment-generating function

$$\begin{aligned}
&\tilde{\mathbf{G}}(s) = \\
&\quad \mathbf{M}_0(\lambda+s)\mathbf{D}(s) \\
&\quad + \left(\sum_{k=1}^{N-1} \mathbf{M}_k(\lambda+s)\mathbf{C}_k(\lambda+s)\mathbf{D}((k+1)s)\right. \\
&\quad (\mathbf{M}_0(s) - \mathbf{M}_0(\lambda+s)) \\
&\quad \left.\prod_{i=1}^{k-1}(\mathbf{M}_i(s)\mathbf{C}_i(s) - \mathbf{M}_i(\lambda+s)\mathbf{C}_i(\lambda+s))\right) \\
&\quad + \mathbf{M}_N(s)\mathbf{C}_N(s)\mathbf{D}((N+1)s) \\
&\quad (\mathbf{M}_0(s) - \mathbf{M}_0(\lambda+s))\prod_{i=1}^{k-1}(\mathbf{M}_i(s)\mathbf{C}_i(s) - \mathbf{M}_i(\lambda+s)\mathbf{C}_i(\lambda+s))
\end{aligned} \tag{4.104}$$

Equation 4.104 shows the moment generating function for the runtime of a task in an inhomogeneous passive replication system with acceptance test fault detection, where there can be no more than N faults occurring to a single task.

4.4.2 Homogeneous systems

As with the watchdog and timeout systems, a model for a system with homogeneous replicas with acceptance test as fault detection is derived.

A task where faults are detected in ϕ replicas will have a total runtime given by

$$Y = (\phi+1)r + (\phi+1)d + \phi c \tag{4.105}$$

where r is the time used to run the task on one replica, d is the test time, and c is the fault correction time.

The expectation function of e^{-sY} with conditions to the number of faults, the runtime, test time and correction time becomes

$$\mathrm{E}[e^{-sY}|r = x, c = y, d = z, \phi = k] = e^{-sx}e^{-sz}(e^{-s(x+y)}e^{-sz})^k \tag{4.106}$$

Using the number of faults probability function from 4.49 to remove the condition on the number of faults yields

$$\begin{aligned}&\mathrm{E}[e^{-sY}|r = x, c = y, d = z] = \\ &\quad e^{-(\lambda+s)x}e^{-sz} \\ &\quad + \sum_{k=1}^{\infty} e^{-(k+1)sz}e^{-(\lambda+(k+1)s)x}e^{-(\lambda+ks)y} \\ &\quad (1 - e^{-\lambda x})(1 - e^{-\lambda(x+y)}))^{k-1}\end{aligned} \tag{4.107}$$

Setting the maximum number of faults before failure to N and removing the timing conditions gives the moment-generating function

$$\begin{aligned}&\mathbf{G}(s) = \\ &\quad \mathbf{M}(\lambda + s)\mathbf{D}(s) \\ &\quad + \sum_{k=1}^{N} \mathbf{D}((k+1)s) \sum_{i=0}^{k-1} (-1)^i \binom{k-1}{i} \\ &\quad (\mathbf{M}((i+1)\lambda + (k+1)s) - \mathbf{M}((i+2)\lambda + (k+1)s)) \\ &\quad \mathbf{C}((i+1)\lambda + ks)\end{aligned} \tag{4.108}$$

Equation 4.108 shows the moment-generating function for a task in an acceptance test based passive replication system where the replicas are homogeneous, and where more than N faults occurring to a single task lead to the failure of the system.

If there can be no more than N faults occurring to the task, the fault probability function 4.53 is used to remove the condition on the number of faults.

This results in the expectation function

$$\begin{aligned}
&\mathrm{E}[e^{-sY}|r=x,c=y,d=z] = \\
&\quad e^{-(\lambda+s)x}e^{-sz} \\
&\quad + \left(\sum_{k=1}^{N-1} e^{-(k+1)sz}e^{-(\lambda+(k+1)s)x}e^{-(\lambda+ks)y} \right. \\
&\quad \left. (1-e^{-\lambda x})(1-e^{-\lambda(x+y)}))^{k-1} \right) \\
&\quad + e^{-(N+1)sz}e^{-(N+1)sx}e^{-Nsy} \\
&\quad (1-e^{-\lambda x})(1-e^{-\lambda(x+y)}))^{N-1}
\end{aligned} \tag{4.109}$$

Removing the timing conditions gives the moment-generating function

$$\begin{aligned}
&\tilde{\mathbf{G}}(s) = \\
&\quad \mathbf{M}(\lambda+s)\mathbf{D}(s) \\
&\quad + \left(\sum_{k=1}^{N-1} \mathbf{D}((k+1)s) \sum_{i=0}^{k-1} (-1)^i \binom{k-1}{i} \right. \\
&\quad (\mathbf{M}((i+1)\lambda+(k+1)s) - \mathbf{M}((i+2)\lambda+(k+1)s)) \\
&\quad \left. \mathbf{C}((i+1)\lambda+ks) \right) \\
&\quad + \mathbf{D}((N+1)s) \sum_{i=0}^{N-1} (-1)^i \binom{N-1}{i} \\
&\quad (\mathbf{M}(i\lambda+(N+1)s) - \mathbf{M}((i+1)\lambda+(N+1)s)) \\
&\quad \mathbf{C}(i\lambda+Ns)
\end{aligned} \tag{4.110}$$

Equation 4.110 shows the moment-generating function for the runtime distribution of a task in a homogeneous passive replication system with acceptance test-based fault detection, where there can be no more than N faults occurring to a single task.

4.4.3 Using a fixed fault probability

For a system with a fixed fault probability $1-\kappa_i$ for each replica i, the number of faults probability function becomes

$$\Pr[\phi = k] = \begin{cases} \kappa_0 & , k = 0 \\ \kappa_k \prod_{i=0}^{k-1}(1-\kappa_i) & , k > 0 \end{cases} \tag{4.111}$$

Using this fault probability function to remove the conditions on the number of faults from the expectation function 4.100 results in the following expectation function

$$\begin{aligned}&\mathrm{E}[e^{-sY}|r_n = x_n, c_n = y_n, d = z] = \\ &\quad \kappa_0 e^{-sx_0} e^{-sz} \\ &\quad + \sum_{k=1}^{\infty} \kappa_k e^{-s(x_k+y_k)} e^{-s(k+1)z} \\ &\quad (1-\kappa_0) e^{-sx_0} \prod_{i=1}^{k-1} (1-\kappa_i) e^{-s(x_i+y_i)}\end{aligned} \tag{4.112}$$

Limiting the number of faults before failure to N, and removing the timing conditions gives the moment-generating function

$$\begin{aligned}&\mathbf{G}(s) = \\ &\quad \kappa_0 \mathbf{M}_0(s) \mathbf{D}(s) \\ &\quad + \sum_{k=1}^{N} \kappa_k \mathbf{M}_k(s) \mathbf{C}_k(s) \mathbf{D}((k+1)s) \\ &\quad (1-\kappa_0) \mathbf{M}_0(s) \prod_{i=1}^{k-1} (1-\kappa_i) \mathbf{M}_i(s) \mathbf{C}_i(s)\end{aligned} \tag{4.113}$$

Equation 4.113 shows the moment generating function for the runtime distribution of a task in an acceptance test based passive replication system where there is a fixed fault probability for each replica, and where more than N faults to a single task lead to a system failure.

If there can be no more than N faults to a single task, 4.113 can be used with $\kappa_N = 1$.

4.5 Other systems

The previous sections in this chapter do cover run-time distribution models for many systems, but there are of course many similar systems that can not be modelled using the derived equations as they are, and where modification of the models has to be used.

In this system, an example of such a modification, a system class using both timeout and acceptance test as fault detectors, is derived.

Also, a short description of how the derived equations can be used in a simple checkpointed system is presented.

4.5.1 Combination of timeout and acceptance tests

Combining the timeout detection with acceptance tests can be useful, as they together can detect a wide range of failures. The timeout can detect crash and omission failures, while the acceptance test can detect some value failures, and combined, failures in all three categories are detected.

In this section, a runtime model of a system that combines the two is developed. In the modelled system, crash or omission failures are caught when the system reaches timeout. If there is no timeout, the acceptance test will check the results for value failures. If a component failure is detected, correction and rerun is performed as in the other modelled systems.

The fault model

For simplicity, the fault model with fixed fault probability for each part of the system is used. For each replica i, there is a probability $\kappa_{\mathrm{co}i}$ that readying the replica (i.e. the correction process) does not cause a crash or omission failure, there is a probability $\kappa_{\mathrm{mo}i}$ that running the replica does not cause a crash or omission failure, and there is a probability $\kappa_{\mathrm{v}i}$ that there is no value failure detected when testing the results.

If a replica fails, there are three possibilities of where the failure is detected:

- Crash or omission failure during correction, detection is at the timeout for the correction, τ_{ci}
- No crash or omission failure during correction, but crash or omission failure during the execution of a replica. Detection is at the timeout for the execution, $\tau_{\mathrm{m}i}$ after the execution started.
- No crash or omission failure, but a value failure detected by the acceptance test.

For a system where there is no correction time (e.g. the primary replica when the task execution starts), the time to failure detection is distributed with the pdf

$$a_{\mathrm{nc}i}(t) = \frac{1-\kappa_{\mathrm{mo}i}}{1-\kappa_{\mathrm{mo}i}\kappa_{\mathrm{v}i}}\delta(t-\tau_{\mathrm{m}i}) + \frac{1-\kappa_{\mathrm{v}i}}{1-\kappa_{\mathrm{mo}i}\kappa_{\mathrm{v}i}}\kappa_{\mathrm{mo}i}\delta(t-(x_i+z)) \quad (4.114)$$

which has the mgf

$$\mathbf{A}_{\mathrm{nc}i}(s) = \frac{1-\kappa_{\mathrm{mo}i}}{1-\kappa_{\mathrm{mo}i}\kappa_{\mathrm{v}i}}e^{-s\tau_{\mathrm{m}i}} + \frac{1-\kappa_{\mathrm{v}i}}{1-\kappa_{\mathrm{mo}i}\kappa_{\mathrm{v}i}}\kappa_{\mathrm{mo}i}e^{-s(x_i+z)} \quad (4.115)$$

If there is a correction time, and a separate timeout function for this, the time to failure detection is distributed with the pdf

$$\begin{array}{l} a_i(t) = \\ \frac{1-\kappa_{\mathrm{co}i}}{1-\kappa_{\mathrm{co}i}\kappa_{\mathrm{mo}i}\kappa_{\mathrm{v}i}}\delta(t-\tau_{\mathrm{c}i}) + \frac{1-\kappa_{\mathrm{mo}i}}{1-\kappa_{\mathrm{co}i}\kappa_{\mathrm{mo}i}\kappa_{\mathrm{v}i}}\kappa_{\mathrm{co}i}\delta(t-(y_i+\tau_{\mathrm{m}i})) \\ + \frac{1-\kappa_{\mathrm{v}i}}{1-\kappa_{\mathrm{co}i}\kappa_{\mathrm{mo}i}\kappa_{\mathrm{v}i}}\kappa_{\mathrm{co}i}\kappa_{\mathrm{mo}i}\delta(t-(y_i+x_i+z)) \end{array} \quad (4.116)$$

which has the mgf

$$\begin{array}{l} \mathbf{A}_i(s) = \\ \frac{1-\kappa_{\mathrm{co}i}}{1-\kappa_{\mathrm{cq}i}\kappa_{\mathrm{mo}i}\kappa_{\mathrm{v}i}}e^{-s\tau_{\mathrm{c}i}} + \frac{1-\kappa_{\mathrm{mo}i}}{1-\kappa_{\mathrm{co}i}\kappa_{\mathrm{mo}i}\kappa_{\mathrm{v}i}}\kappa_{\mathrm{co}i}e^{-sy_i}e^{-s\tau_{\mathrm{m}i}} \\ + \frac{1-\kappa_{\mathrm{mo}i}}{1-\kappa_{\mathrm{co}i}\kappa_{\mathrm{mo}i}\kappa_{\mathrm{v}i}}\kappa_{\mathrm{co}i}\kappa_{\mathrm{mo}i}e^{-s(y_i+x_i+z)} \end{array} \quad (4.117)$$

Deriving the equation

As in the previous systems, we start with the number of faults probability model

$$\begin{array}{l} \Pr[\phi = k] = \\ \begin{cases} \kappa_{\mathrm{mo}0}\kappa_{\mathrm{v}0} & ,k=0 \\ \kappa_{\mathrm{co}k}\kappa_{\mathrm{mo}k}\kappa_{\mathrm{v}k}(1-\kappa_{\mathrm{mo}0}\kappa_{\mathrm{v}0})\prod_{i=1}^{k-1}(1-\kappa_{\mathrm{co}i}\kappa_{\mathrm{mo}i}\kappa_{\mathrm{v}i}) & ,k>0 \end{cases} \end{array} \quad (4.118)$$

and an expectation function with condition to the number of faults and the times used on each part of the system

$$\begin{array}{l} \mathrm{E}[e^{-sY}|r_n = x_n, c_n = y_n, d = z, \phi = k] = \\ \begin{cases} e^{-s(x_0+z)} & ,k=0 \\ e^{-s(x_i+y_i+z)}\mathbf{A}_{\mathrm{nc}0}(s)\prod_{i=1}^{k-1}\mathbf{A}_i(s) & ,k>0 \end{cases} \end{array} \quad (4.119)$$

Removing the condition on the number of faults yields the expectation function

$$\begin{array}{l} \mathrm{E}[e^{-sY}|r_n = x_n, c_n = y_n, d = z] = \\ \kappa_{\mathrm{mo}0}\kappa_{\mathrm{v}0}e^{-s(x_0+z)} \\ + \sum_{k=1}^{\infty}\kappa_{\mathrm{co}k}\kappa_{\mathrm{mo}k}\kappa_{\mathrm{v}k}e^{-s(x_k+y_k+z)} \\ ((1-\kappa_{\mathrm{mo}0})e^{-s\tau_{\mathrm{m}0}} + (1-\kappa_{\mathrm{v}0})\kappa_{\mathrm{mo}0}e^{-s(x_0+z)}) \\ \prod_{i=1}^{k-1}(1-\kappa_{\mathrm{co}i})e^{-s\tau_{\mathrm{c}i}} + (1-\kappa_{\mathrm{mo}i})\kappa_{\mathrm{co}i}e^{-sy_i}e^{-s\tau_{\mathrm{m}i}} \\ + (1-\kappa_{\mathrm{v}i})\kappa_{\mathrm{co}i}\kappa_{\mathrm{mo}i}e^{-s(x_i+y_i+z)} \end{array} \quad (4.120)$$

Limiting the number of faults before failure to N and removing the conditions on time use, gives the moment generating function

$$\begin{aligned}\mathbf{G}(s) = \\ &\kappa_{\mathrm{mo}0}\kappa_{\mathrm{v}0}\mathbf{M}_0(s)\mathbf{D}(s) \ + \sum_{k=1}^{N} \kappa_{\mathrm{co}k}\kappa_{\mathrm{mo}k}\kappa_{\mathrm{v}k}\mathbf{M}_k(s)\mathbf{C}_k(s)\mathbf{D}(s) \\ &((1-\kappa_{\mathrm{mo}0})e^{-s\tau_{\mathrm{m}0}} + (1-\kappa_{\mathrm{v}0})\kappa_{\mathrm{mo}0}\mathbf{M}_0(s)\mathbf{D}(s)) \\ &\prod_{i=1}^{k-1}(1-\kappa_{\mathrm{co}i})e^{-s\tau_{\mathrm{c}i}} + (1-\kappa_{\mathrm{mo}i})\kappa_{\mathrm{co}i}\mathbf{C}_i(s)e^{-s\tau_{\mathrm{m}i}} \\ &\quad + (1-\kappa_{\mathrm{v}i})\kappa_{\mathrm{co}i}\kappa_{\mathrm{mo}i}\mathbf{M}_i(s)\mathbf{C}_i(s)\mathbf{D}(s)\end{aligned} \tag{4.121}$$

Equation 4.121 describes the moment generating function of the runtime of a combined timeout and acceptance test system, as a function of the fault-free runtimes, the correction times, the timeouts and the probabilities that failures does not occur.

4.5.2 Simple checkpointed systems

In the simple checkpointed system, the task is partitioned into N parts, with checkpoints between the parts. Before entering a new part, the state is logged. Each part i has a run-time distribution with mgf $\mathbf{G}_i(s)$. When an error is detected, the system goes back to the latest checkpoint, restarting the part in which the error was detected. This can be viewed as running a series of fault tolerant tasks, where the total runtime can be viewed as a convolution of the runtime for all parts of the system.

Given a checkpointed method consists of N parts where a fault detected in a part is corrected within the same part. Each part i have a run-time, including any fault fault handling, distributed by $G_i(t)$. The total runtime distribution of the system will then be equal to the convolution of the runtime distribution for all the parts, which, when written as a moment-generating function, becomes the product of the mgfs of the runtimes for the parts:

$$\mathbf{G}_{tot}(s) = \prod_{i=1}^{N} \mathbf{G}_i(s) \tag{4.122}$$

4.6 System structures with models not derived

As mentioned in the introduction to the chapter, there are of course systems with structures that do not fit with any of the models in this work. Many of

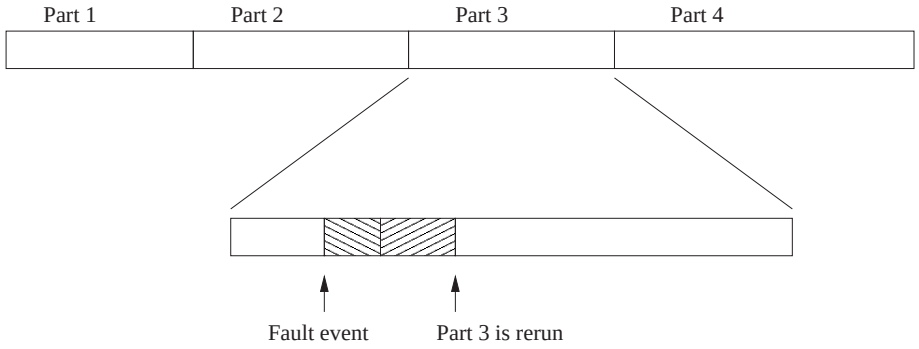


Figure 4.3: Program consisting of several fault-tolerant parts. If an error is detected in one of the parts, the affected part, not the whole program, is rerun.

the models not presented here can, however, be derived in the same way as the models that are derived in this work.

This section presents some thought about some system structures that are not modeled in this work, where the derivation of the models are less straightforward than the ones given in the work. It can be viewed as suggestions for future work, but also as a caution of some of the "pitfalls" that must be avoided when developing new models.

4.6.1 Combination of timeout and acceptance tests, with failure probabilities dependent on previous failure modes

In 4.5.1, a system where timeout and acceptance tests were combined for fault detection was described, and the runtime model for the system was derived. In the model, the fault model used different probabilities for value and omission failures, and the probabilities could also be varied between the different replicas.

A use for varying failure probabilities between the replicas is to model faults like design faults, that are inherent in the system, so that for a system where the primary fails, the backups have a higher probability of failure. A weakness with the model as it is, is that the failure *mode* of the primary is disregarded when it comes to the failure probabilities of the backups.

If, for instance, we want a model where a failure of the primary leads to a higher probability for the *same mode* of failure of the backup, but no change in

the probabilities for other failure modes, the model presented in 4.5.1 cannot be used.

To create a mathematical model for this system, the number of detected faults probability function must be changed so that not only the number of faults, but also the types of faults are a part of the function, e.g.

$$\Pr(\phi_{\mathrm{v}} = k_{\mathrm{v}}, \phi_{\mathrm{mo}} = k_{\mathrm{mo}}, \phi_{\mathrm{co}} = k_{\mathrm{co}}) = \cdots \tag{4.123}$$

As shown in figure 4.4, there can be several paths to a given number of failures, which must be reflected in the fault probability functions. For a system without correction failures, and the failure probabilities as shown in figure 4.4, the detected fault probability function becomes

$$\begin{aligned}&\Pr(\phi_{\mathrm{v}} = k_{\mathrm{v}}, \phi_{\mathrm{mo}} = k_{\mathrm{mo}}) = \\ &\left(\prod_{i=0}^{k_{\mathrm{v}}} 1 - \kappa_{\mathrm{v}i}\right)\left(\prod_{i=0}^{k_{\mathrm{mo}}} 1 - \kappa_{\mathrm{mo}i}\right)\left(\sum_{i=0}^{k_{\mathrm{mo}}} \kappa_{\mathrm{mo}i}\right)^{k_v}\end{aligned} \tag{4.124}$$

When adding the probabilities for correction failures, the probability function becomes more complex.

The time to detection function must also be changed. As the distributions and timeout values used in the model may vary between the replicas, the sequence of failures can be of importance also here, e.g. the timing distribution of a run that reaches timeout in the primary followed by a non-acceptance in the first backup can be different from a run that has a non-acceptance in the primary followed by a timeout of the backup.

The factors discussed here show that the full mathematical model for the runtime in this system can be very complex, but it should still be possible to derive using the methods described in this chapter.

The difference between the resulting distributions from this model and an approximation using the model derived in 4.5.1 will be very small for most systems, as the models will differ on the probabilities of the backups' failures, while the primary is unaffected.

4.6.2 Nested fault tolerant structures

It can often be desirable to design a fault tolerant structure out of parts that also are fault tolerant. For instance, a task that is replicated may itself be built from parts that also are replicated, as shown in figure 4.5.

At first, it may seem logical to find the runtime distributions in a way similar to the simple checkpointed system described in 4.5.2, e.g. using one of the

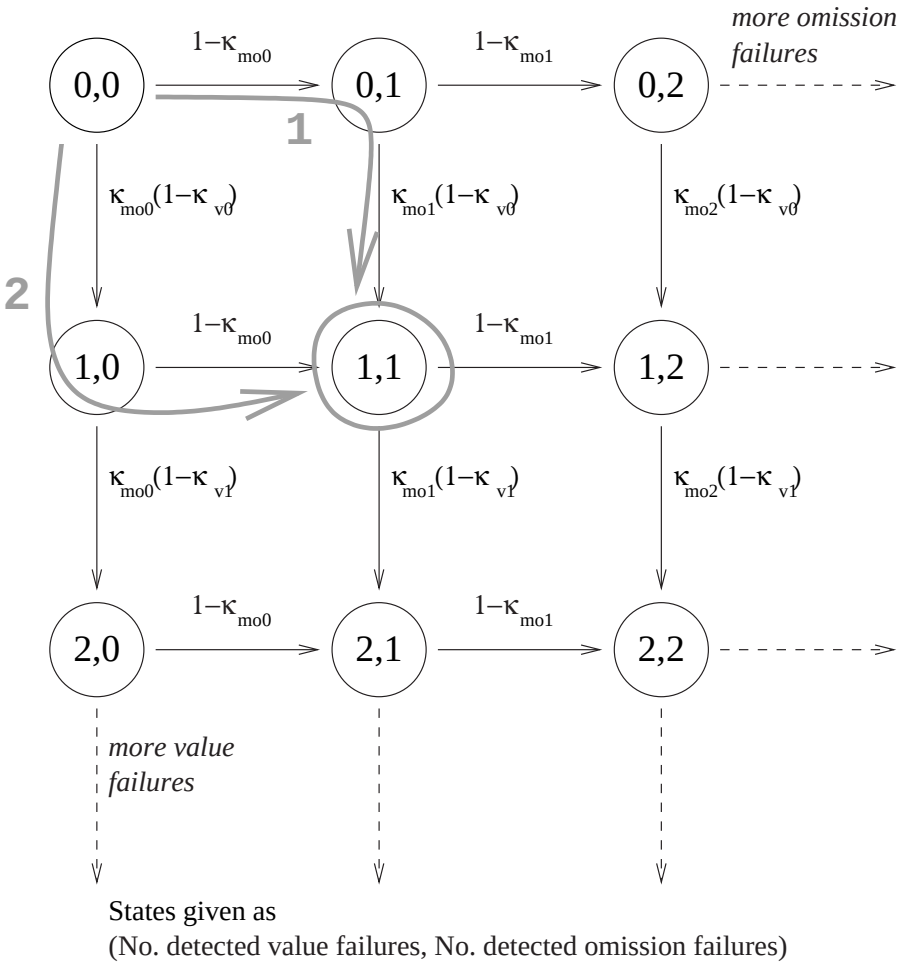


Figure 4.4: If one omission and one value failure occurs during the running of a task, there are two possible ways this can happen. In path 1, the omission failure occurs first, in path 2, the value failure occurs first. If there are more failures, more paths are possible

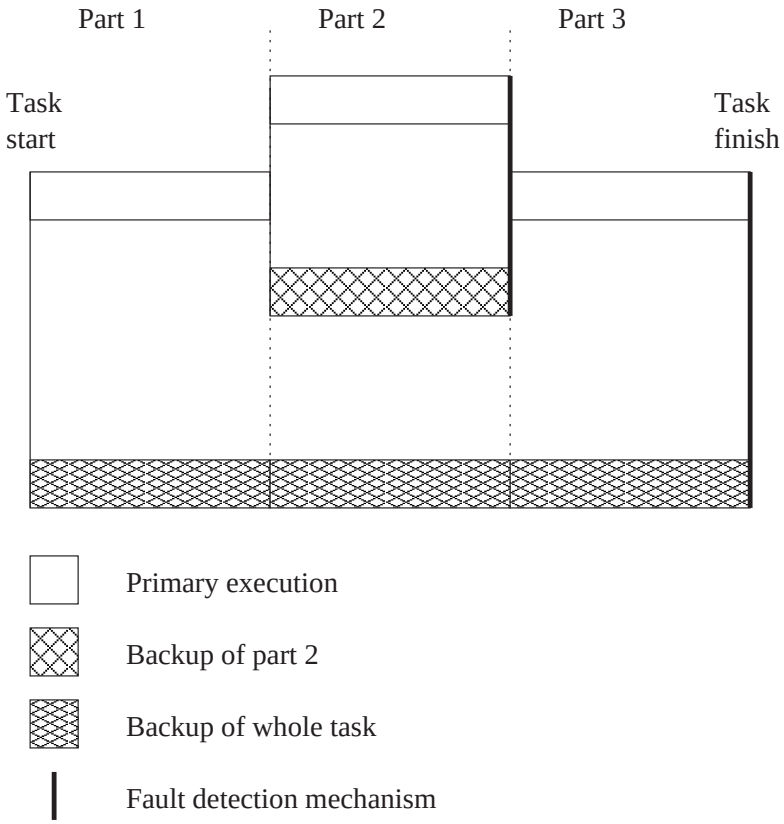


Figure 4.5: Nested fault tolerant structure. If a fault is detected after part 2, only part 2 is rerun. If a fault is detected at task finish, the whole task is rerun.

previously derived models to find the runtime distribution of part 2, and then use this runtime distribution as a part of the "fault-free" runtime distribution for the whole task in figure 4.5. For a system where *different and independent* faults are detected at the end of part 2 and the end of the task, this may also be a correct solution.

If, however, the same faults or faults that are correlated can be detected at the end of part 2 and the end of the task, using the solution described above is not correct. Seen from the last fault detector, the task itself has the possibility to correct some of the occurring faults before reaching the detection mechanism, so the "inner" fault tolerance structure of part 2 will affect the fault probabilities of the "outer" fault tolerant structure of the whole task, which must be reflected in the model.

4.6.3 Checkpointed systems that roll back further than the last checkpoint

A problem with the checkpointed system described in 4.5.2 is that there sometimes is a need for rolling back to an earlier checkpoint than the last, as a detected error may have a cause that occurred prior to the last checkpoint. Some checkpointed fault tolerant systems will therefore roll back more than one checkpoint, or even restart the task if certain criteria (e.g. recurrence of faults detected at the same checkpoint) is fulfilled.

A fault tolerant structure with a checkpoint strategy that allows rolling back more than one checkpoint will require a more complex mathematical runtime distribution model. The number of checkpoints that are rolled back, how far back it is possible to roll back before the whole task must be restarted, and other rollback strategy decisions may vary. This may lead to several different models, each customized to a given strategy.

4.7 Summary of main results

This section contains a summary of the results found earlier in the chapter.

The inhomogeneous system with watchdog fault detection

For a fault tolerant system based on passive, inhomogeneous replicas and a watchdog fault detection system, where more than N faults lead to the failure

of the system, the run-time distribution of a single task is given by equation 4.34:

$$\begin{aligned}
&\mathbf{G}(s)\\
&= \mathbf{M}_0(\lambda+s)\\
&\quad + \sum_{k=1}^{N} \left(\frac{\lambda}{\lambda+s}\right)^k \mathbf{M}_k(\lambda+s)\mathbf{C}_k(\lambda+s)\\
&\qquad (1-\mathbf{M}_0(\lambda+s))\mathbf{D}_0(s) \prod_{i=1}^{k-1}(1-\mathbf{M}_i(\lambda+s)\mathbf{C}_i(\lambda+s))\mathbf{D}_i(s)
\end{aligned}$$

If there can be no more than N faults occurring to the system, the runtime is given by equation 4.35:

$$\begin{aligned}
&\tilde{\mathbf{G}}(s)\\
&= \mathbf{M}_0(\lambda+s)\\
&\quad + \left(\sum_{k=1}^{N} \left(\frac{\lambda}{\lambda+s}\right)^k \mathbf{M}_k(\lambda+s)\mathbf{C}_k(\lambda+s)\right.\\
&\qquad \left.(1-\mathbf{M}_0(\lambda+s))\mathbf{D}_0(s) \prod_{i=1}^{k-1}(1-\mathbf{M}_i(\lambda+s)\mathbf{C}_i(\lambda+s))\mathbf{D}_i(s)\right)\\
&\quad + \left(\frac{\lambda}{\lambda+s}\right)^N \mathbf{M}_N(s)\mathbf{C}_N(s)\\
&\qquad (1-\mathbf{M}_0(\lambda+s))\mathbf{D}_0(s) \prod_{i=1}^{N-1}(1-\mathbf{M}_i(\lambda+s)\mathbf{C}_i(\lambda+s))\mathbf{D}_i(s)
\end{aligned}$$

The homogeneous system with watchdog fault detection

For a passive replication system based on homogeneous replicas and a watchdog based fault detector, where more than N faults to a single task lead to a system failure, the mgf of the runtime of a task is given by equation 4.57:

$$\begin{aligned}
&\mathbf{G}(s)\\
&= \mathbf{M}(\lambda+s)\\
&\quad + \left(\sum_{k=1}^{N} \left(\frac{\lambda}{\lambda+s}\mathbf{D}(s)\right)^k\right.\\
&\qquad \left(\sum_{i=0}^{k-1}(-1)^i \binom{k}{i} (\mathbf{M}((i+1)(\lambda+s))\right.\\
&\qquad \left.\left.- \mathbf{M}((i+2)(\lambda+s)))\mathbf{C}((i+1)(\lambda+s))\right)\right)
\end{aligned}$$

If there can be no more than N faults occurring to a single task, the runtime distribution is given by equation 4.58:

$$
\begin{aligned}
&\tilde{\mathbf{G}}(s) \\
&= \mathbf{M}(\lambda+s) \\
&\quad + \left(\sum_{k=1}^{k_{max}-1} \left(\frac{\lambda}{\lambda+s}\mathbf{D}(s) \right)^k \right. \\
&\quad \left(\sum_{i=0}^{k-1} (-1)^i \binom{k}{i} (\mathbf{M}((i+1)(\lambda+s)) \right. \\
&\quad \left. \left. - \mathbf{M}((i+2)(\lambda+s)))\mathbf{C}((i+1)(\lambda+s)) \right) \right) \\
&\quad + \left(\frac{\lambda}{\lambda+s}\mathbf{D}(s) \right)^N \\
&\quad \left(\sum_{i=0}^{N-1} (-1)^i \binom{N-1}{i} (\mathbf{M}(i\lambda+(i+1)s) \right. \\
&\quad \left. - \mathbf{M}((i+1)\lambda+(i+2)s))\mathbf{C}(i\lambda+(i+1)s) \right)
\end{aligned}
$$

System with changing fault rates and watchdog based fault detection

For a passive replication system with watchdog-based fault detection, where the fault rate may change between different parts of the system, and where more than N faults lead to the failure of the system, the runtime distribution is given by the moment-generating function in equation 4.62:

$$
\begin{aligned}
&\mathbf{G}(s) \\
&= \mathbf{M}_0(\lambda_{\mathrm{m}0}+s) \\
&\quad + \sum_{k=1}^{N} \mathbf{C}_k(\lambda_{\mathrm{c}k}+s)\mathbf{M}_k(\lambda_{\mathrm{m}k}+s) \\
&\quad (1-\mathbf{M}_0(\lambda_{\mathrm{m}0}+s))\frac{\lambda_{\mathrm{m}0}}{\lambda_{\mathrm{m}0}+s}\mathbf{D}_0(s) \\
&\quad \prod_{i=1}^{k-1} \left((1-\mathbf{C}_i(\lambda_{\mathrm{c}i}+s))\frac{\lambda_{\mathrm{c}i}}{\lambda_{\mathrm{c}i}+s} \right. \\
&\quad \left. + (\mathbf{C}_i(\lambda_{\mathrm{c}i}+s) - \mathbf{C}_i(\lambda_{\mathrm{c}i}+s)\mathbf{M}_i(\lambda_{\mathrm{m}i}+s))\frac{\lambda_{\mathrm{m}i}}{\lambda_{\mathrm{m}i}+s} \right) \mathbf{D}_i(s)
\end{aligned}
$$

If there can be no more than N faults occurring to a single task, the runtime distribution is given by equation 4.65:

$$
\begin{aligned}
&\tilde{\mathbf{G}}(s) \\
&= \mathbf{M}_0(\lambda_{\mathrm{m}0} + s) \\
&\quad + \Bigg(\sum_{k=1}^{N} \mathbf{C}_k(\lambda_{\mathrm{c}k} + s)\mathbf{M}_k(\lambda_{\mathrm{m}k} + s) \\
&\qquad (1 - \mathbf{M}_0(\lambda_{\mathrm{m}0} + s))\frac{\lambda_{\mathrm{m}0}}{\lambda_{\mathrm{m}0} + s}\mathbf{D}_0(s) \\
&\qquad \prod_{i=1}^{k-1} \Bigg((1 - \mathbf{C}_i(\lambda_{\mathrm{c}i} + s))\frac{\lambda_{\mathrm{c}i}}{\lambda_{\mathrm{c}i} + s} \\
&\qquad\quad + (\mathbf{C}_i(\lambda_{\mathrm{c}i} + s) - \mathbf{C}_i(\lambda_{\mathrm{c}i} + s)\mathbf{M}_i(\lambda_{\mathrm{m}i} + s))\frac{\lambda_{\mathrm{m}i}}{\lambda_{\mathrm{m}i} + s} \Bigg) \mathbf{D}_i(s) \Bigg) \\
&\quad + \mathbf{C}_N(s)\mathbf{M}_N(s) \\
&\qquad (1 - \mathbf{M}_0(\lambda_{\mathrm{m}0} + s))\frac{\lambda_{\mathrm{m}0}}{\lambda_{\mathrm{m}0} + s}\mathbf{D}_0(s) \\
&\qquad \prod_{i=1}^{k-1} \Bigg((1 - \mathbf{C}_i(\lambda_{\mathrm{c}i} + s))\frac{\lambda_{\mathrm{c}i}}{\lambda_{\mathrm{c}i} + s} \\
&\qquad\quad + (\mathbf{C}_i(\lambda_{\mathrm{c}i} + s) - \mathbf{C}_i(\lambda_{\mathrm{c}i} + s)\mathbf{M}_i(\lambda_{\mathrm{m}i} + s))\frac{\lambda_{\mathrm{m}i}}{\lambda_{\mathrm{m}i} + s} \Bigg) \mathbf{D}_i(s)
\end{aligned}
$$

Inhomogeneous system with timeout fault detection

For an inhomogeneous passively replicated fault tolerant system with timeout as a fault detection method, and where more than N faults occurring to a single task lead to the failure of the system, the runtime is distributed with a moment-generating function given by equation 4.80:

$$
\begin{aligned}
&\mathbf{G}(s) = \\
&\mathbf{M}_0(\lambda + s) \\
&\quad + \sum_{k=1}^{N} \mathbf{M}_k(\lambda + s)\mathbf{C}_k(\lambda + s) \\
&\qquad e^{-s\tau_{\mathrm{m}0}}(1 - \mathbf{M}_0(\lambda)) \\
&\qquad \prod_{i=1}^{k-1} e^{-s\tau_{\mathrm{c}i}}(1 - \mathbf{C}_i(\lambda)) + e^{-s\tau_{\mathrm{m}i}}\mathbf{C}_i(\lambda + s)(1 - \mathbf{M}_i(\lambda))
\end{aligned}
$$

If faults after the Nth are ignored, the runtime is distributed with the moment-generating function given by equation 4.82:

$$
\begin{aligned}
&\tilde{\mathbf{G}}(s) = \\
&\mathbf{M}_0(\lambda+s) \\
&+\Bigg(\sum_{k=1}^{N-1} \mathbf{M}_k(\lambda+s)\mathbf{C}_k(\lambda+s) \\
&\quad e^{-s\tau_{\mathrm{m}0}}(1-\mathbf{M}_0(\lambda)) \\
&\quad \prod_{i=1}^{k-1} e^{-s\tau_{\mathrm{c}i}}(1-\mathbf{C}_i(\lambda)) + e^{-s\tau_{\mathrm{m}i}}\mathbf{C}_i(\lambda+s)(1-\mathbf{M}_i(\lambda))\Bigg) \\
&+\mathbf{M}_N(s)\mathbf{C}_N(s) \\
&\quad e^{-s\tau_{\mathrm{m}0}}(1-\mathbf{M}_0(\lambda)) \\
&\quad \prod_{i=1}^{k-1} e^{-s\tau_{\mathrm{c}i}}(1-\mathbf{C}_i(\lambda)) + e^{-s\tau_{\mathrm{m}i}}\mathbf{C}_i(\lambda+s)(1-\mathbf{M}_i(\lambda))
\end{aligned}
$$

Homogeneous system with timeout fault detection

For a homogeneous passive replication system with timeout-based fault detection where more than N faults to a single task lead to the failure of the system, the moment-generating function for the runtime distribution of a task is given by equation 4.90:

$$
\begin{aligned}
&\mathbf{G}(s) = \\
&\quad \mathbf{M}(\lambda+s) \\
&\quad +\sum_{k=1}^{N}\sum_{j=0}^{k-1} \binom{k-1}{j} e^{-(k-j)\tau_{\mathrm{m}}s} e^{-j\tau_{\mathrm{c}}s} \\
&\quad \left(\sum_{i=0}^{k-j}(-1)^i \binom{k-j}{i} \mathbf{M}((i+1)\lambda+s)\right) \\
&\quad \left(\sum_{i=0}^{j}(-1)^i \binom{j}{i} \mathbf{C}((i+k-j)\lambda+(k-j)s)\right)
\end{aligned}
$$

If there can be no more than N faults occurring to a single task, the runtime distribution has a moment-generating function given by equation 4.92:

$$\begin{aligned}
&\tilde{\mathbf{G}}(s) = \\
&\quad \mathbf{M}(\lambda + s) \\
&\quad + \left(\sum_{k=1}^{N-1} \sum_{j=0}^{k-1} \binom{k-1}{j} e^{-(k-j)\tau_{\mathrm{m}} s} e^{-j\tau_{\mathrm{c}} s} \right. \\
&\quad \left(\sum_{i=0}^{k-j} (-1)^i \binom{k-j}{i} \mathbf{M}((i+1)\lambda + s) \right) \\
&\quad \left. \left(\sum_{i=0}^{j} (-1)^i \binom{j}{i} \mathbf{C}((i+k-j)\lambda + (k-j)s) \right) \right) \\
&\quad + \sum_{j=0}^{N-1} \binom{N-1}{j} e^{-(N-j)\tau_{\mathrm{m}} s} e^{-j\tau_{\mathrm{c}} s} \\
&\quad \left(\sum_{i=0}^{N-j} (-1)^i \binom{N-j}{i} \mathbf{M}(i\lambda + s) \right) \\
&\quad \left(\sum_{i=0}^{j} (-1)^i \binom{j}{i} \mathbf{C}((i+N-j-1)\lambda + (N-j)s) \right)
\end{aligned}$$

System with fixed fault probability and timeout fault detection

For an inhomogeneous timeout system where the fault probability model is based on a fixed fault probability for each part of the system instead of the poisson fault process, the run-time is given by the moment generating function in equation 4.98:

$$\begin{aligned}
&\mathbf{G}(s) = \\
&\kappa_{\mathrm{m}0} \mathbf{M}_0(s) \\
&+ \sum_{k=1}^{N} \kappa_{\mathrm{m}k} \kappa_{\mathrm{c}k} \mathbf{M}_k(s) \mathbf{C}_k(s) (1 - \kappa_{\mathrm{m}0}) e^{-s\tau_{\mathrm{m}0}} \\
&\prod_{i=1}^{k-1} (1 - \kappa_{\mathrm{c}i}) e^{-s\tau_{\mathrm{c}i}} + (1 - \kappa_{\mathrm{m}i}) \kappa_{\mathrm{c}i} \mathbf{C}_i(s) e^{-s\tau_{\mathrm{m}i}}
\end{aligned}$$

If there can be no more than N faults occurring to a single task, equation 4.98 is used with $\kappa_{\mathrm{m}N} = \kappa_{\mathrm{c}N} = 1$.

Inhomogeneous system with acceptance test

For an inhomogeneous passive replication system where the fault detection is based on acceptance tests, and where more than N faults to a single task leads to the failure of the system, the runtime distribution of a task is given by the moment-generating function in equation 4.102:

$$\begin{aligned}&\mathbf{G}(s) = \\ &\quad \mathbf{M}_0(\lambda+s)\mathbf{D}(s) \\ &\quad + \sum_{k=1}^{N} \mathbf{M}_k(\lambda+s)\mathbf{C}_k(\lambda+s)\mathbf{D}((k+1)s) \\ &\quad (\mathbf{M}_0(s) - \mathbf{M}_0(\lambda+s)) \prod_{i=1}^{k-1} (\mathbf{M}_i(s)\mathbf{C}_i(s) - \mathbf{M}_i(\lambda+s)\mathbf{C}_i(\lambda+s))\end{aligned}$$

If there can be no more than N faults occurring to the system, the runtime distribution is given by the moment-generating function in equation 4.104:

$$\begin{aligned}&\tilde{\mathbf{G}}(s) = \\ &\quad \mathbf{M}_0(\lambda+s)\mathbf{D}(s) \\ &\quad + \left(\sum_{k=1}^{N-1} \mathbf{M}_k(\lambda+s)\mathbf{C}_k(\lambda+s)\mathbf{D}((k+1)s) \right. \\ &\quad (\mathbf{M}_0(s) - \mathbf{M}_0(\lambda+s)) \\ &\quad \left. \prod_{i=1}^{k-1} (\mathbf{M}_i(s)\mathbf{C}_i(s) - \mathbf{M}_i(\lambda+s)\mathbf{C}_i(\lambda+s)) \right) \\ &\quad + \mathbf{M}_N(s)\mathbf{C}_N(s)\mathbf{D}((N+1)s) \\ &\quad (\mathbf{M}_0(s) - \mathbf{M}_0(\lambda+s)) \prod_{i=1}^{k-1} (\mathbf{M}_i(s)\mathbf{C}_i(s) - \mathbf{M}_i(\lambda+s)\mathbf{C}_i(\lambda+s))\end{aligned}$$

Homogeneous system with acceptance test

For a passive replication system with homogeneous replicas where the fault detection is based on acceptance tests, and where more than N faults to a single task leads to the failure of the system, the runtime of a task in the system is

distributed with a moment-generating function given by equation 4.108:

$$\begin{aligned}&\mathbf{G}(s) = \\ &\quad \mathbf{M}(\lambda + s)\mathbf{D}(s) \\ &\quad + \sum_{k=1}^{N} \mathbf{D}((k+1)s) \sum_{i=0}^{k-1} (-1)^i \binom{k-1}{i} \\ &\quad (\mathbf{M}((i+1)\lambda + (k+1)s) - \mathbf{M}((i+2)\lambda + (k+1)s)) \\ &\quad \mathbf{C}((i+1)\lambda + ks)\end{aligned}$$

If there can be no more than N faults occurring to a single task, the run-time distribution of a task is given by the moment-generating function in equation 4.110:

$$\begin{aligned}&\tilde{\mathbf{G}}(s) = \\ &\quad \mathbf{M}(\lambda + s)\mathbf{D}(s) \\ &\quad + \Bigg(\sum_{k=1}^{N-1} \mathbf{D}((k+1)s) \sum_{i=0}^{k-1} (-1)^i \binom{k-1}{i} \\ &\quad (\mathbf{M}((i+1)\lambda + (k+1)s) - \mathbf{M}((i+2)\lambda + (k+1)s)) \\ &\quad \mathbf{C}((i+1)\lambda + ks) \Bigg) \\ &\quad + \mathbf{D}((N+1)s) \sum_{i=0}^{N-1} (-1)^i \binom{N-1}{i} \\ &\quad (\mathbf{M}(i\lambda + (N+1)s) - \mathbf{M}((i+1)\lambda + (N+1)s)) \\ &\quad \mathbf{C}(i\lambda + Ns)\end{aligned}$$

System with fixed fault probability and acceptance test

For an inhomogeneous acceptance test system where there is a fixed fault probability for each replica, the run-time of the system is given by the moment-generating function in equation 4.113:

$$\begin{aligned}&\mathbf{G}(s) = \\ &\quad \kappa_0 \mathbf{M}_0(s)\mathbf{D}(s) \\ &\quad + \sum_{k=1}^{N} \kappa_k \mathbf{M}_k(s)\mathbf{C}_k(s)\mathbf{D}((k+1)s) \\ &\quad (1-\kappa_0)\mathbf{M}_0(s) \prod_{i=1}^{k-1} (1-\kappa_i)\mathbf{M}_i(s)\mathbf{C}_i(s)\end{aligned}$$

If there can be no more than N faults occurring to a single task, equation 4.113 is used with $\kappa_N = 1$.

System combining timeout and acceptance test

For an inhomogeneous system with both timeout and acceptance test as fault detection and fixed fault probabilities for the different fault types in each part of the system, the run-time of the system is given by the moment-generating function in equation 4.121:

$$\begin{aligned}
&\mathbf{G}(s) = \\
&\kappa_{\mathrm{mo}0}\kappa_{\mathrm{v}0}\mathbf{M}_0(s)\mathbf{D}(s) \; + \sum_{k=1}^{N} \kappa_{\mathrm{co}k}\kappa_{\mathrm{mo}k}\kappa_{\mathrm{v}k}\mathbf{M}_k(s)\mathbf{C}_k(s)\mathbf{D}(s) \\
&\quad ((1-\kappa_{\mathrm{mo}0})e^{-s\tau_{\mathrm{m}0}} + (1-\kappa_{\mathrm{v}0})\kappa_{\mathrm{mo}0}\mathbf{M}_0(s)\mathbf{D}(s)) \\
&\quad \prod_{i=1}^{k-1}(1-\kappa_{\mathrm{co}i})e^{-s\tau_{\mathrm{c}i}} + (1-\kappa_{\mathrm{mo}i})\kappa_{\mathrm{co}i}\mathbf{C}_i(s)e^{-s\tau_{\mathrm{m}i}} \\
&\qquad + (1-\kappa_{\mathrm{v}i})\kappa_{\mathrm{co}i}\kappa_{\mathrm{mo}i}\mathbf{M}_i(s)\mathbf{C}_i(s)\mathbf{D}(s)
\end{aligned}$$

If there can be no more than N faults to a single task, equation 4.121 is used with $\kappa_{\mathrm{mo}N} = \kappa_{\mathrm{co}N} = \kappa_{\mathrm{v}N} = 1$.

Chapter 5

Use of the models

In chapter 4, mathematical models for run-time distributions in various passive replication systems were developed. In this chapter, it is shown through examples how the models and the results from the models can be used.

To be able to "see" the run-time distributions, the distributions have to be transformed from the moment-generating function domain to the time domain, so this chapter also contains a short description on how MATLAB and Simulink can be used for a numerical transform from moment-generating functions to cumulative distribution functions.

To be able to compare the results from the mathematical models with results generated in another way, a discrete-event simulator that simulated the timing behavior of passive fault tolerant systems was developed. This chapter also contains a description of the simulator used.

5.1 The simulator

A discrete event simulator was developed to simulate the behavior of different passive replication systems. Simulating the systems is in many ways a more intuitive approach than using the mathematical models developed in chapter 4, however, the results from the simulator are less accurate. The main goal with the simulator was to enable comparison of the results from the mathematical models with results generated in a different way, thus having an "acceptance test" for the results.

The simulator used in this work was first presented in [47].

5.1.1 Simulator structure

Because of the various fault tolerance mechanisms that were to be simulated, the simulator was built in a way that the parts of it could be changed easily to match the different simulated systems. The main parts of the simulator and the communication between them is shown in figure 5.1.

Client

The client part of the simulator creates *process* objects, which are sent to the server.

Server

The main part of the simulator is the server, which implements both the normal service and the fault handling of the system. The operation of the server depends on the fault detection mechanism that is simulated.

Server operation in watchdog and timeout systems

During normal, fault-free operation, the server with watchdog or timeout fault detection operates as a two-state (**Idle** and **Busy**) machine. The fault-free behaviour is the same for both versions of the server.

When a *process* object arrives from the client, the time of the arrival is logged.

If the server is in the **Idle** state during *process* arrival, a processing time is drawn from the runtime distribution, and the server changes state to **Busy**, and the time for start of processing is logged.

If server is in not the **Idle** state, the *process* is enqueued. If the queue is not empty when the processing of a *process* object is finished, the processing of the first object in the queue is begun immediately, and the start of the processing time is logged.

When the processing of a *process* object is finished and the queue is empty, the server reenters the **Idle** state.

During fault handling, the server will operate according to the chosen fault tolerance behaviour.

If the server receives a *fault* object while it is in the **Busy** or **Correction** state, it will enter the **Error** state. The watchdog server will draw a fault detection time from the fault detection time distribution, while the timeout server will calculate the time remaining to timeout and use this as the fault

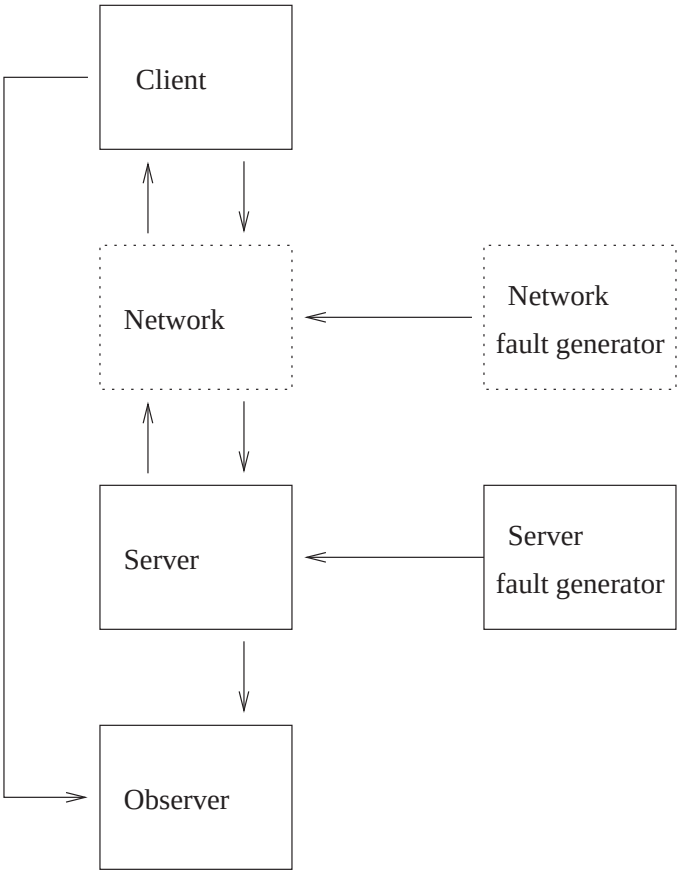


Figure 5.1: Main structure of the simulator

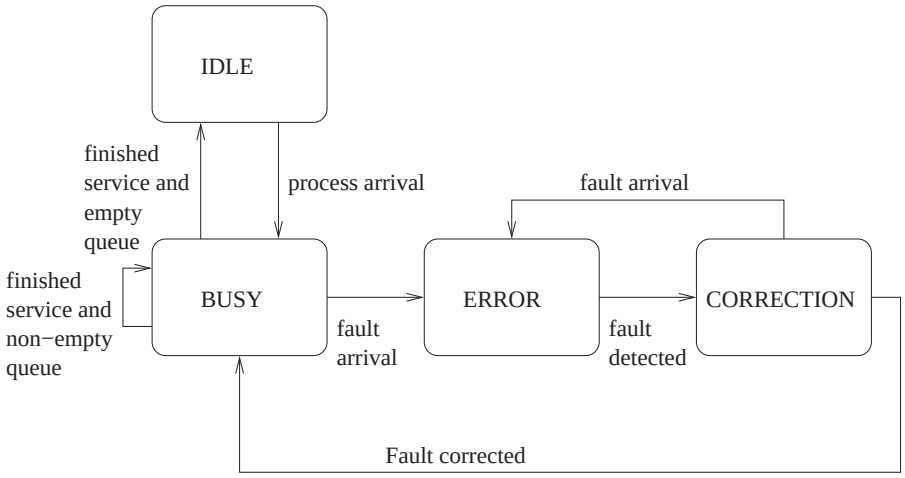


Figure 5.2: State machine diagram for the server part of the simulator in a watchdog or a timeout system

correction time. If the server is in the **Idle** or **Error** state, the arrival of the *fault* object is ignored as there is either no process that the fault affects, or the fault affects an already erroneous process.

The server will remain in the **Error** state for the duration of the fault detection time, then it will enter the **Correction** state. A correction time is drawn from the fault correction time distribution, when this time is up, the server will return to the **Busy** state (unless a new fault arrived while it was in the **Correction** state). For inhomogeneous systems, a new time is drawn from a runtime distribution, for homogeneous system, the runtime that was drawn when the processing of the *process* object started is used.

A state machine diagram for the server operation in a watchdog or timeout system is shown in figure 5.2.

Server operation in acceptance test systems

During normal operation, the acceptance test system has three states, **Idle**, **Busy** and **Testing**.

When a *process* object arrives from the client, the time of the arrival is logged.

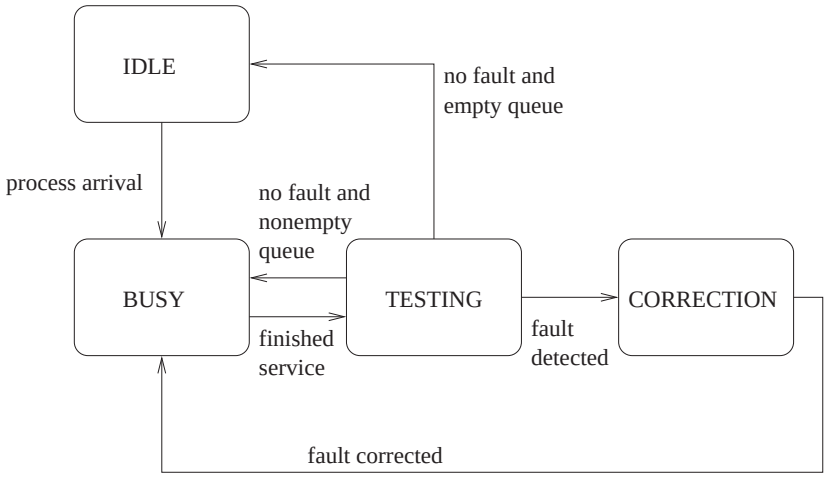


Figure 5.3: State machine diagram for the server operation in an acceptance test system

If the server is in the **Idle** state when the *process* arrives, a service time is drawn from the service time distribution, the server changes state to **Busy**, and the start of processing time is logged.

If the server is not in the **Idle** state when the *process* arrives, the *process* is enqueued.

When the processing time is up, the server enters the **Testing** state, and a test time is drawn from the test time distribution. The server will remain in the **Testing** state for the duration of the test time.

If there has been no *fault* arrival during the service, the server will enter the **Idle** state when the test time is finished if there is no enqueued *process* objects. If there is a *process* in the queue, the server will enter the **Busy** state and start service on the first *process* in the queue.

If there has been a *fault* arrival, the server will enter the **Correction** state after the test time is finished. A correction time is drawn from the correction time distribution, and the server will restart the service on the *process* when the correction time is finished, entering the **Busy** state.

A state machine diagram for the server operation in an acceptance test system is shown in figure 5.3.

Fault generator

The fault generator part of the simulator sends out *fault* objects that will trigger a fault event in the server. When a *fault* is created, a random time is drawn from a negative exponential distribution to decide when the next *fault* is to be created.

Observer

The observer is an object that collects data from the simulation and presents these data on a form that can be read by the data processing program. This was done by writing the results as text in a `.m` file, using a format that Matlab could read.

Network and Network fault generator

A simulator part for the *network* and a *network fault generator* was planned, but never implemented. As the main focus of this work has been on faults that occur when a task is processed on the server, this part was not prioritized.

5.1.2 Implementation

The simulator was implemented in C++ using the ADEVS library. The description provided here is meant as a brief overview of how the simulator was implemented.

ADEVS

ADEVS [1] (**A** **D**iscrete **EV**ent **S**ystem simulator) is a C++ library for constructing discrete event simulators, developed by James Nutaro and released under the GNU Lesser General Public Licence. It provides a simulator engine, a random generator with the possibility to draw from several common distributions, and basic classes that can be used as the simulator's building blocks.

The library was suitable for building the simulator described here. The documentation of the library was a bit lacking, and there were some minor problems with the version used (adevs-1.2)[1]. Most notably did the `triangular` function of the *rv* (random variable) class *not* draw values from a triangular

[1] There are later versions of the library available, but as the simulator worked after some workarounds were implemented, it has not been investigated whether the problems have been fixed in these versions.

distribution. It was, however, possible to create workarounds once the problems was discovered.

Program structure

The simulator was created using the ADEVS *staticDigraph* class, and using the functions `staticDigraph::add` and `staticDigraph::couple` to build the structure described in 5.1.1.

The main parts described in 5.1.1 were implemented as specializations of the ADEVS *atomic* class. The *atomic* class is the basic building block of the simulator which provides handling of communication between the parts of the simulator and handling of internal and external events.

The Client, Fault generator, and Observer were implemented as classes that were direct specializations of the *atomic* class, while the different variants of servers all are specializations of a *BaseServer* class, which again is a specialization of the *atomic* class.

Faults and Processes are classes that are specializations of the ADEVS *object* class. Objects of these classes are passed as messages between the parts of the simulator, and the process objects also function as containers for simulator results.

A *Random* class, specialized from the ADEVS *rv* class was also created, providing an interface for drawing random values that was specialized for this simulator.

5.1.3 Handling and presentation of results

The results from the simulation were written to a text file in MATLAB's `.m`-file format, and read by MATLAB.

To present the results in a form that could be compared to the results from the numerical calculations using the equations from chapter 4, a cumulative distribution function was created from the simulation results using MATLAB's `ecdf` (empirical cumulative distributed function) function.

5.2 Calculating the results numerically

As the mathematical models are written in the moment-generating function domain, and most of the results that are used are in the time domain, the equations have to be inverse-transformed in some way.

While a symbolic inverse-transform is possible for many systems, this approach is both time-consuming and complicated. Also, as the equations describe functions of distributions, new inverse transforms have to be calculated when distributions are changed. Because of this, the time-domain results used in the examples were calculated numerically.

This was done by building a model of the moment-generating function using Simulink. The moment-generating function of a distribution can be seen as the laplace transform of the distribution's probability density function. Thus, describing the moment-generating function of a distribution as a control system's transfer function and calculating this "control system's" step response, will generate the distribution's cumulative distribution function.

An example of how the structure of a Simulink model can be built for generating the cumulative distribution function from the moment generating function of a watchdog-based system is shown in figure 5.4.

It should be noted that as this method of calculating the cumulative distribution function may suffer from the same problems as other continuous system simulations, like numerical instability.

5.3 Examples

This section presents some examples of the use of the models derived in chapter 4. For some of the examples, results are compared to results from simulation, using the simulator described in 5.1. The examples in this section are:

Example 5.1 shows the use of and comparison of the runtime models for watchdog systems, using both the homogeneous and inhomogeneous system model and both strategies for handling more than N faults. Results are compared with simulator results, and a short description on how the results can be used for finding reliability as a function of the number of runs is given.

Example 5.2 shows how the parameters from the previous system can be changed and the resulting runtime models. In the example, this is used for optimizing the interval between "I'm alive" signals with respect to hard deadlines.

Example 5.3 gives a short example on how the model of the watchdog system with inhomogeneous replicas can be used in a system where the distributions vary between the different replicas. Results are compared to simulator results.

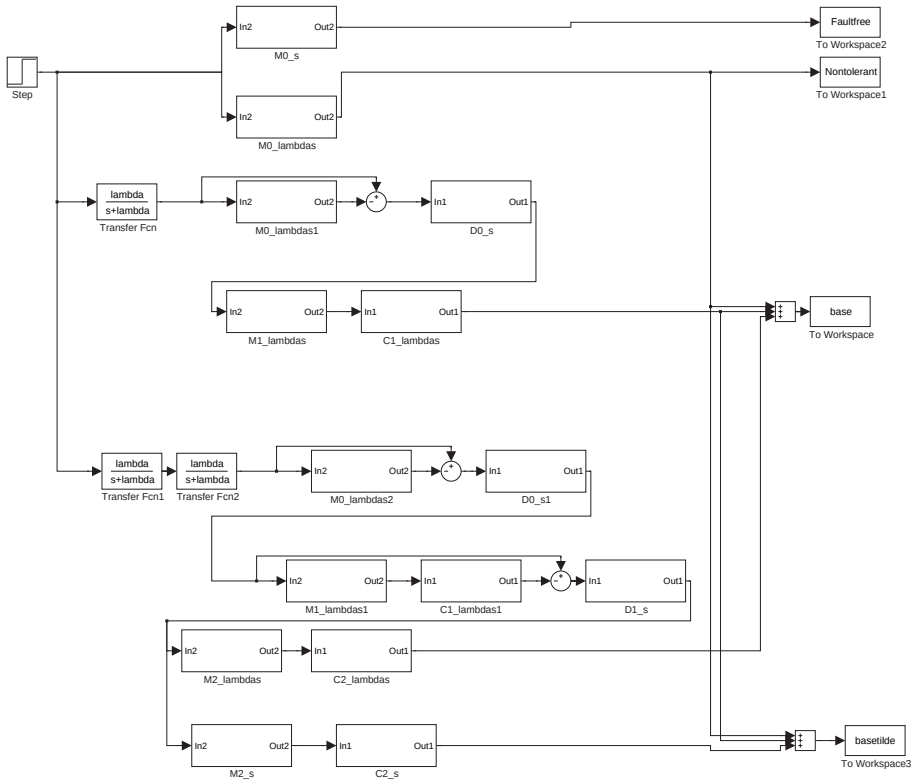


Figure 5.4: Model for generating the cumulative density function from the moment generating function

Example 5.4 shows how the model can be used to analyze the effect of using imprecise replicas in a watchdog system. The results are compared to simulator results.

Example 5.5 shows the use of the model for a timeout system. Results are compared to simulator results.

Example 5.6 shows how the model can be used to analyze the effect of using imprecise replicas in a timeout based system. Results are compared to simulator results.

Example 5.7 shows the use of the model for acceptance test systems. Results are compared to simulator results.

Example 5.8 shows the use of the model for the system that combines acceptance test and timeout as fault detection. Results are compared to similar systems that are affected by the same failures, but where one of the detection mechanisms is missing, and undetected faults lead to system failure.

Example 5.9 shows the use of the model in a system with checkpoints, where acceptance test and timeout fault detection are performed at each checkpoint. In the example, the model is used to optimize the number of checkpoints with respect to hard deadlines.

5.3.1 Some common distributions used in the examples

In most of the examples, a set of example distributions will be used. As the goal of this chapter is to show how the mathematical models are used, the distributions are chosen with the intent of being easy to use, i.e. it should be easy to draw random values from them in simulator programming, and it should be easy to derive their moment generating function and model these in Simulink.

For the fault-free runtimes, the following distributions were used:

m_α A triangular distribution with minimum time 6, maximum time 10, and mode 8

m_β A deterministic runtime of 9

m_γ A deterministic runtime of 3, used to model a fast, imprecise method

The following correction time distributions were used:

c_α A uniform distribution with minimum time 0 and maximum time 5

c_γ A uniform distribution with minimum time 0 and maximum time 1, used to model the update time for an object with a small statespace used as an imprecise backup

For detection time, the following distribution was used:

d_α A uniform distribution with minimum time 1 and maximum time 3

The timeout values used for the different distributions are set to the maximum value for each of the distributions.

For acceptance test time, the following distribution was used:

d_β A deterministic test time of 1

The cumulative distribution functions for these distributions are shown in figure 5.5

These distributions have the following probability density functions and the corresponding moment generating functions:

m_α, the triangular runtime distribution

$$m_\alpha(t) = \begin{cases} 0 & ,0 \leq t < 6 \\ \dfrac{t-6}{4} & ,6 \leq t < 8 \\ \dfrac{10-t}{4} & ,8 \leq t < 10 \\ 0 & ,t \geq 10 \end{cases} \tag{5.1}$$

$$\mathbf{M}_\alpha(s) = \frac{e^{-6s} - 2e^{-8s} + e^{-10s}}{4s^2} \tag{5.2}$$

m_β, the deterministic runtime distribution

$$m_\beta(t) = \delta(t-9) \tag{5.3}$$

$$\mathbf{M}_\beta(s) = e^{-9s} \tag{5.4}$$

m_γ, the runtime distribution for imprecise backups

$$m_\gamma(t) = \delta(t-3) \tag{5.5}$$

$$\mathbf{M}_\gamma(s) = e^{-3s} \tag{5.6}$$

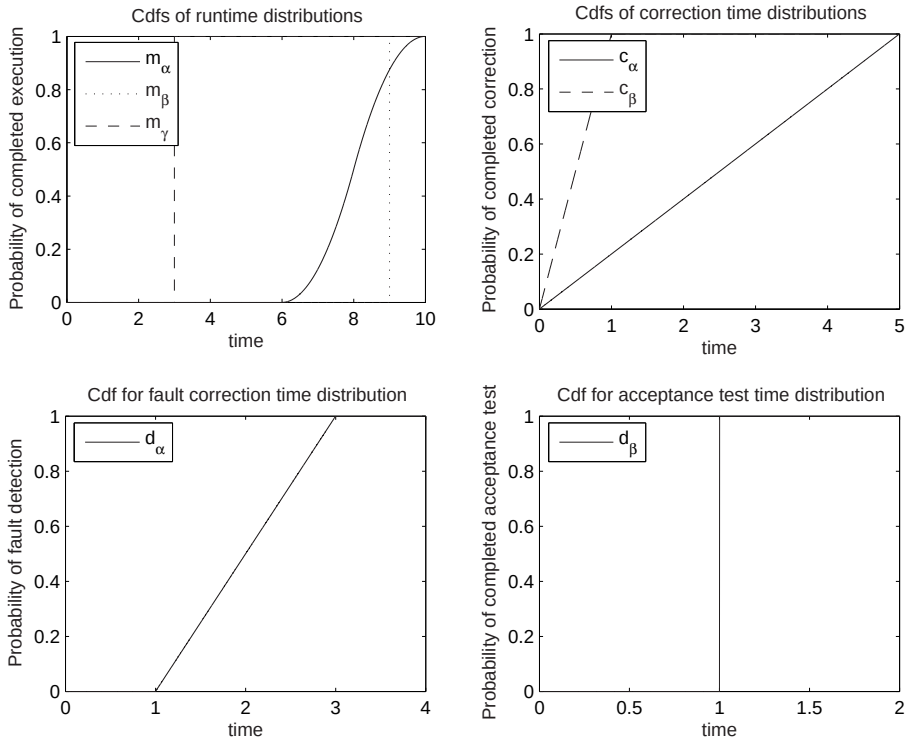


Figure 5.5: Cumulative distribution functions for the distributions listed in 5.3.1

c_α, the correction time distribution

$$c_\alpha(t) = \begin{cases} \frac{1}{5} & ,0 \leq t < 5 \\ 0 & ,t \geq 5 \end{cases} \tag{5.7}$$

$$\mathbf{C}_\alpha(s) = \frac{1 - e^{-5s}}{5s} \tag{5.8}$$

c_γ, the correction time distribution for imprecise backups

$$c_\gamma(t) = \begin{cases} 1 & ,0 \leq t < 1 \\ 0 & ,t \geq 5 \end{cases} \tag{5.9}$$

$$\mathbf{C}_\gamma(s) = \frac{1 - e^{-s}}{s} \tag{5.10}$$

d_α, the fault detection time distribution

$$d_\alpha(t) = \begin{cases} 0 & ,0 \leq t < 1 \\ \frac{1}{2} & ,1 \leq t < 3 \\ 0 & ,t \geq 3 \end{cases} \tag{5.11}$$

$$\mathbf{D}_\alpha(s) = \frac{e^{-s} - e^{-3s}}{2s} \tag{5.12}$$

d_β, the acceptance test time distribution

$$d_\beta(t) = \delta(t-1) \tag{5.13}$$

$$\mathbf{D}_\beta(s) = e^{-s} \tag{5.14}$$

For some of the examples, other distributions may be used. Distributions other than those listed here will be presented in the actual examples.

5.3.2 Systems using watchdogs as a fault detection mechanism

First, examples of systems using fault detection based on watchdogs will be presented. All the systems have the same backup structure: One primary and two backups. This makes it possible to use the same equations for all the systems, and change the distributions and fault rate to match each system. By

using the equations for watchdog systems derived the in previous chapter, 4.34, 4.35, 4.57, and 4.58 with $N = 2$, we get the following equations to use in the examples:

Equation for a system with inhomogeneous backups, where more than N faults leads to failure of the system:

$$\begin{aligned}
&\mathbf{G}_{inhom}(s) \\
&= \mathbf{M}_0(\lambda+s) \\
&\quad + \frac{\lambda}{\lambda+s}\mathbf{M}_1(\lambda+s)\mathbf{C}_1(\lambda+s)(1-\mathbf{M}_0(\lambda+s))\mathbf{D}_0(s) \\
&\quad + \left(\frac{\lambda}{\lambda+s}\right)^2 \mathbf{M}_2(\lambda+s)\mathbf{C}_2(\lambda+s)(1-\mathbf{M}_0(\lambda+s))\mathbf{D}_0(s) \\
&\quad (1-\mathbf{M}_1(\lambda+s)\mathbf{C}_1(\lambda+s))\mathbf{D}_1(s)
\end{aligned} \tag{5.15}$$

Equation for a system with inhomogeneous backups, where there can be no more than N faults:

$$\begin{aligned}
&\tilde{\mathbf{G}}_{inhom}(s) \\
&= \mathbf{M}_0(\lambda+s) \\
&\quad + \frac{\lambda}{\lambda+s}\mathbf{M}_1(\lambda+s)\mathbf{C}_1(\lambda+s)(1-\mathbf{M}_0(\lambda+s))\mathbf{D}_0(s) \\
&\quad + \left(\frac{\lambda}{\lambda+s}\right)^2 \mathbf{M}_2(s)\mathbf{C}_2(s)(1-\mathbf{M}_0(\lambda+s))\mathbf{D}_0(s) \\
&\quad (1-\mathbf{M}_1(\lambda+s)\mathbf{C}_1(\lambda+s))\mathbf{D}_1(s)
\end{aligned} \tag{5.16}$$

Equation for a system with homogeneous backups, where more than N faults leads to the failure of the system:

$$\begin{aligned}
&\mathbf{G}_{hom}(s) \\
&= \mathbf{M}(\lambda+s) \\
&\quad + \frac{\lambda}{\lambda+s}(\mathbf{M}(\lambda+s)-\mathbf{M}(2(\lambda+s)))\mathbf{C}(\lambda+s)\mathbf{D}(s) \\
&\quad + \left(\frac{\lambda}{\lambda+s}\right)^2 ((\mathbf{M}(\lambda+s)-\mathbf{M}(2(\lambda+s)))\mathbf{C}(\lambda+s) \\
&\quad - (\mathbf{M}(2(\lambda+s))-\mathbf{M}(3(\lambda+s)))\mathbf{C}(2(\lambda+s)))\mathbf{D}(s)^2
\end{aligned} \tag{5.17}$$

Equation for a system with homogeneous backups, where there can be no

more than N faults:

$$\begin{aligned}
&\tilde{\mathbf{G}}_{hom}(s) \\
&= \mathbf{M}(\lambda+s) \\
&\quad + \frac{\lambda}{\lambda+s}(\mathbf{M}(\lambda+s) - \mathbf{M}(2(\lambda+s)))\mathbf{C}(\lambda+s)\mathbf{D}(s) \\
&\quad + \left(\frac{\lambda}{\lambda+s}\right)^2 ((\mathbf{M}(s) - \mathbf{M}(\lambda+2s))\mathbf{C}(s) \\
&\quad\quad - (\mathbf{M}(\lambda+2s) - \mathbf{M}(2\lambda+3s))\mathbf{C}(\lambda+2s))\mathbf{D}(s)^2
\end{aligned} \tag{5.18}$$

Example 5.1 A basic system

In the first example, the triangular distribution m_α is used for both primary and backup methods, with the correction time distribution c_α and detection time distribution d_α. The mean time between faults, $\frac{1}{\lambda}$ is set to 10 000.

The goal with this example is to show how the mathematical models developed in chapter 4 are used, and how the results from these models compare to simulated results. Further, the results from the models of the homogeneous and inhomogeneous systems are compared, as well as results from the different ways of handling more than N faults described in 4.1.3.

As both the homogeneous and the inhomogeneous system models are to be used in the example, a distribution that shows the difference between the two models is used. As the inhomogeneous and homogeneous systems will behave the same if deterministic times are used, the triangular runtime distribution m_α is chosen for this example. For the inhomogeneous system, the same distribution is used for both the primary and the backups.

For both the homogeneous and inhomogeneous system, the simulator was set to 500 000 runs.

Figure 5.6 shows the calculated (from eq. 5.15) and simulated cumulative density functions for the execution time in the inhomogeneous system, while figure 5.7 shows the calculated (from eq. 5.17) and simulated cumulative density for the execution time in the homogeneous system.

Calculated and simulated results

While the curves of the simulated and calculated results are quite similar, the simulated results seem to have less faults than the calculated results for the inhomogeneous system, and more faults than the calculated for the homogeneous system.

For both the homogeneous and the inhomogeneous system, 400 of the 500 000 task runs in the simulation should have experienced faults for a perfect match between the simulated and calculated results. In the simulation of the inhomogeneous system, 378 tasks (5.5% fewer than expected) experienced faults,

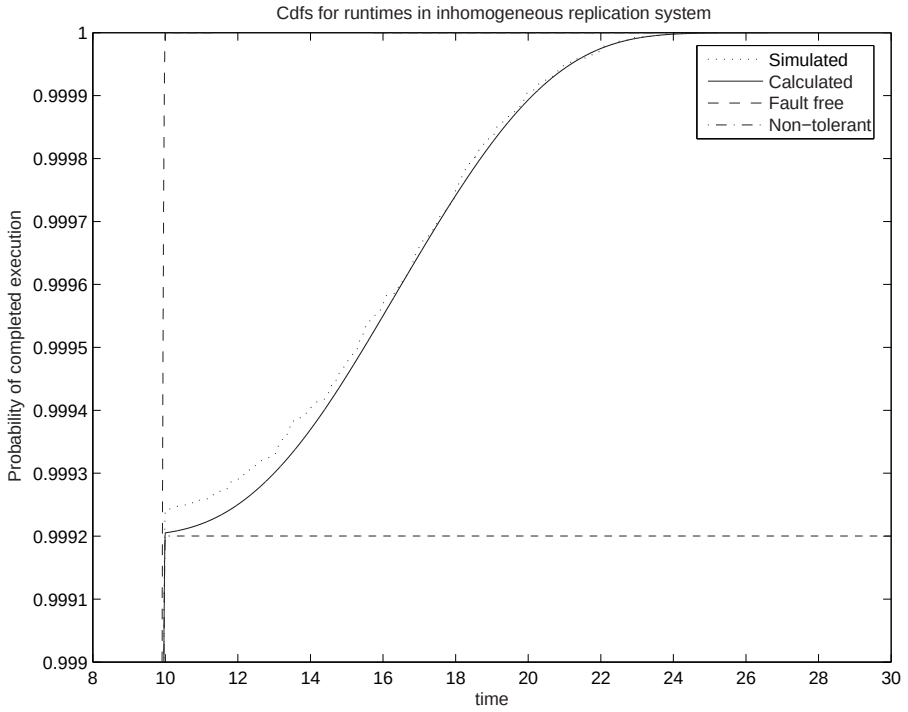


Figure 5.6: Cumulative density functions for the inhomogeneous watchdog system described in example 5.1.

while for the simulation of the homogeneous system, 412 tasks (3.0% more than expected) experienced faults.

Simulator results closer to the calculated results could probably have been achieved by increasing the number of simulator runs. To achieve this, the *observer* part of the simulator must be changed so it presents the results from the simulation in a format that is easier for Matlab to handle, as Matlab seemed to have problems handling more than approximately half a million results in the format currently produced by the simulator.

When comparing the two methods of getting results, the simulator is more intuitive, as the structure of the simulator in many ways matches the structure of the simulated system. The downside is the large number of simulator runs

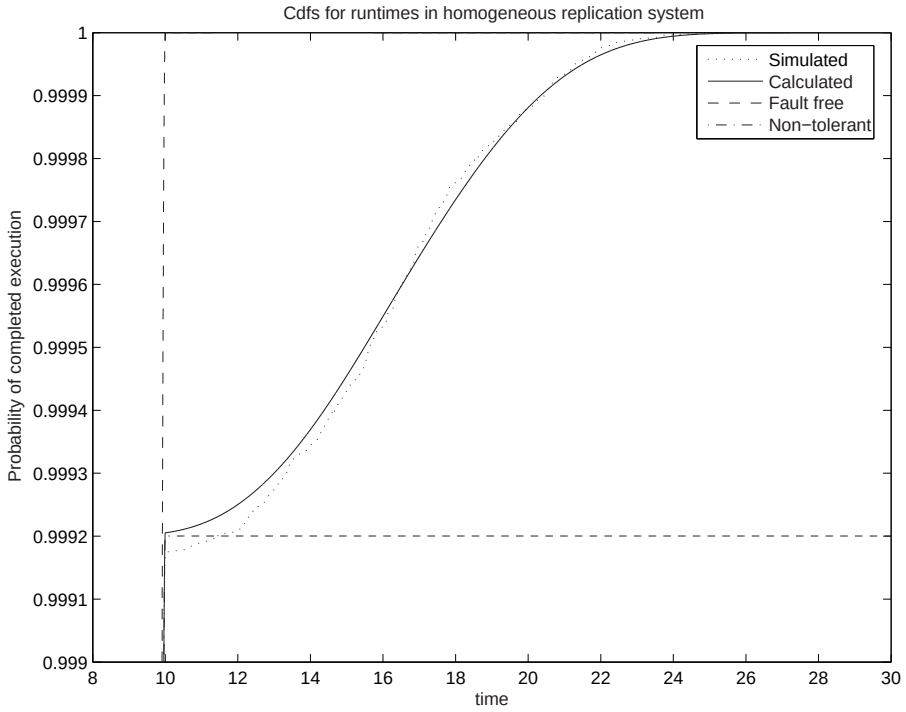


Figure 5.7: Cumulative density functions for the homogeneous watchdog system described in example 5.1.

needed to get "good" results. While using the mathematical models developed in chapter 4 is not as intuitive as the simulator, the computation is faster and easier to implement once the equations are in place. It should be noted that for the numerical generation of results that were used in the examples, numerical errors may occur.

Inhomogeneous and homogeneous systems

For the system parameters chosen in this example, the difference in the runtime distributions of a homogeneous and an inhomogeneous system is quite small, as shown in figure 5.8. The results seem slightly better for the inhomogeneous system. The difference between the two is highest around $t = 20.5$, where the probability of completion is $1 - 8.0 \times 10^{-5}$ for the inhomogeneous system

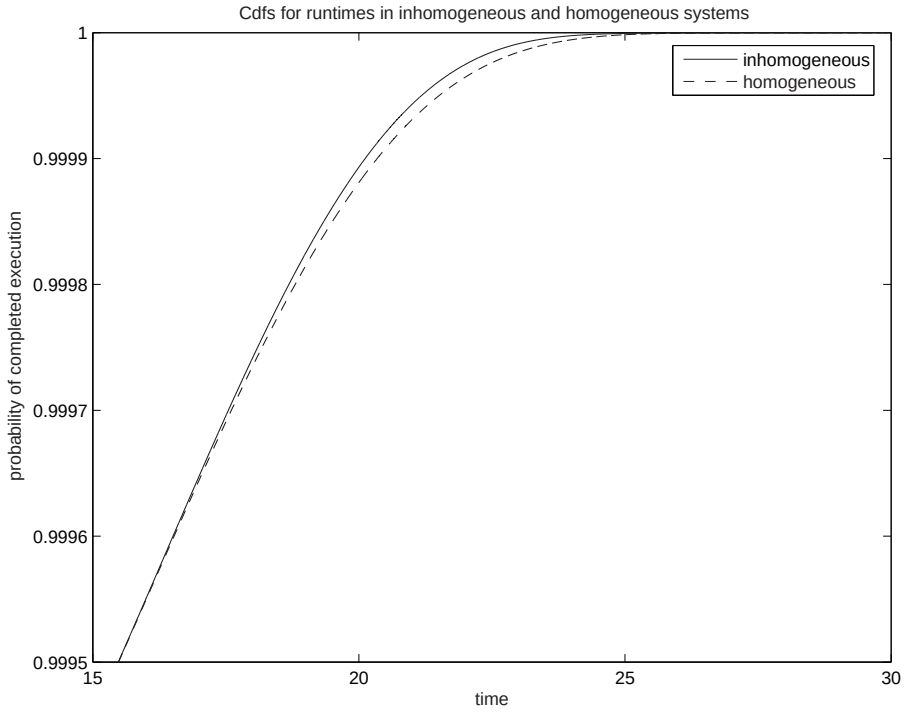


Figure 5.8: Comparison of the cdfs of the inhomogeneous and homogeneous systems in examplé 5.1

and $1 - 9.2 \times 10^{-5}$ for the homogeneous system.

The same runtimes are used for both the primary and the backups in the homogeneous systems, and this causes the observed run-time distributions for the backups to be somewhat skewed toward longer runtimes, as there is a higher probability that tasks with long runtimes will fail with the fault model used. In the inhomogeneous systems, the runtimes of the backups are unaffected by the runtimes of the primary.

More than N faults

The difference between $G(t)$ and $\tilde{G}(t)$ is, for this example, quite small. The difference between the two is at maximum after all backups have had a chance to run. The system where more than N faults are ignored will of course reach

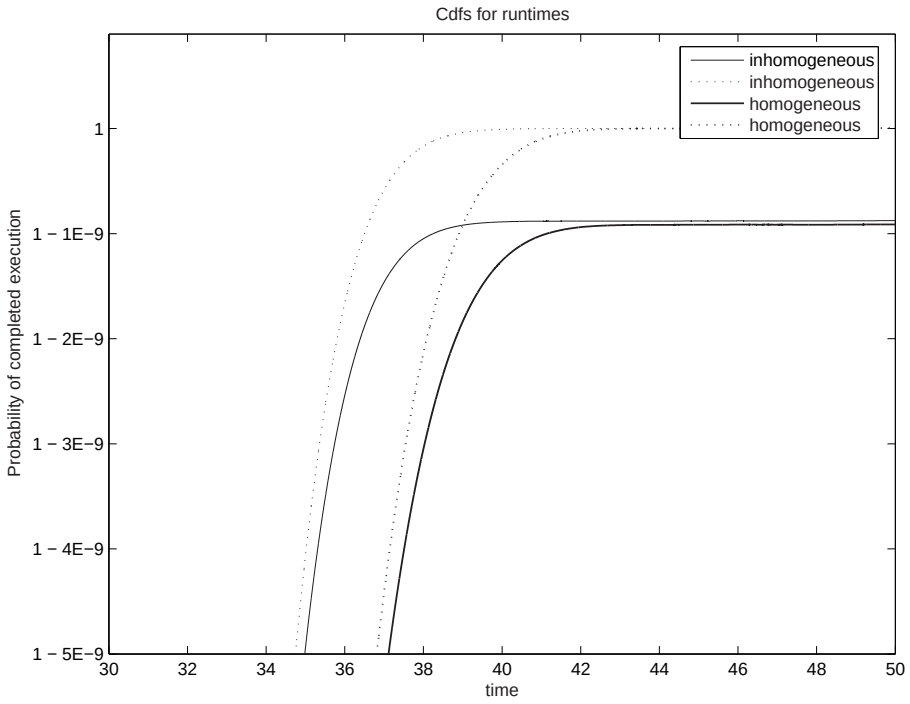


Figure 5.9: Comparison of the cdfs of the distributions for g (solid) and $\tilde{g}$ (dotted) for the inhomogeneous and homogeneous systems in example 5.1

a failure probability of 0, while the failure probability for a system where more than N faults lead to failure is around 9×10^{-10}, as shown in figure 5.9.

Reliability as a function of the number of runs

Assuming that the fault tolerance mechanism makes the system able to tolerate any faults except timing faults, the cumulative distribution functions can be used to make models of the system's reliability as a function of the number of runs. If the system fails when a task's runtime exceeds a deadline of t_{dl}, the probability that the system works after n task runs is

$$R_{t_{dl}}(n) = G(t_{dl})^n \tag{5.19}$$

Figure 5.10 shows the probability of system survival as a function of number of runs for different deadlines. As the curves show, the improvement is not very

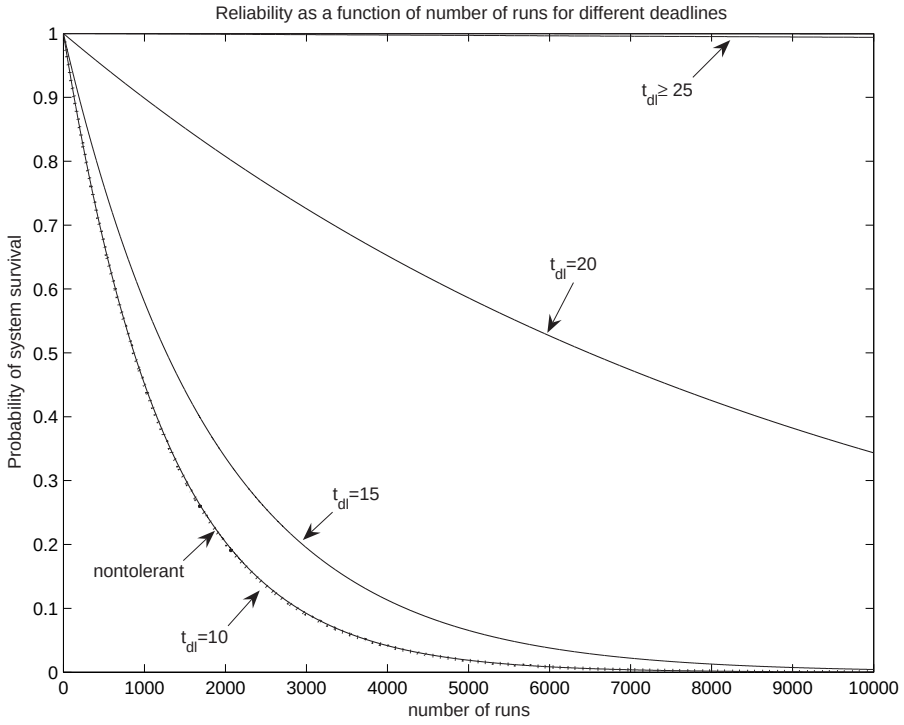


Figure 5.10: Reliability functions for different deadlines for the inhomogeneous system in example 5.1

high for a system where the deadline is below 15. If a fault occurs in such a system, the probability that the fault is detected and the backup has finished its rerun before the deadline is not very high. For deadlines around 20, the improvement in the systems reliability is significant. For deadlines above 25, the improvement is very good, even if the task still will fail to meet its deadline in the worst-case single fault scenario.

Example 5.2 Changing the parameters

In this example, the inhomogeneous system from example 5.1 is used as a base, but with some parameter changes. The goal with the example is to show how the results change when some parameter changes are applied, and how the

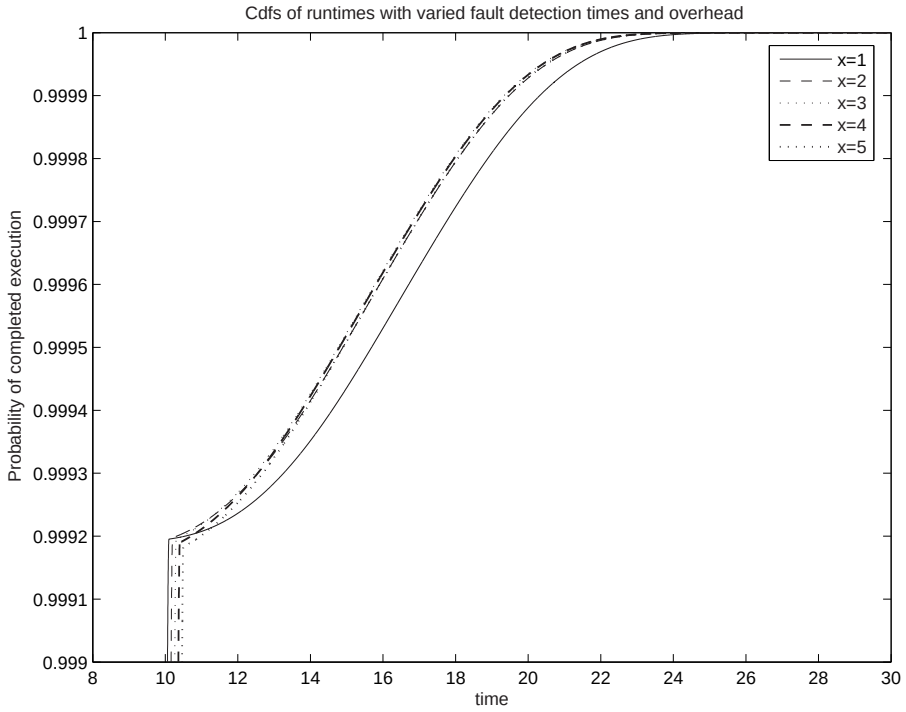


Figure 5.11: Cumulative density functions for the systems in example 5.2.

models can be used in an optimization problem.

In this example, it is assumed that the signaling used by the fault detection in the system cause some overhead, and that the overhead is larger for higher signaling frequencies.

This is modelled by letting the execution time of a method be increased with an overhead when the detection time is shortened.

A variable x is used to represent the changes in signaling frequency, higher x means shorter time between the "I'm alive" signals. The detection time is set to be uniformly distributed between $\frac{1}{x}$ and $\frac{3}{x}$, so that for $x = 1$, the distribution d_α is used.

$$\mathbf{D}_x(s) = \mathbf{D}_\alpha(\frac{s}{x}) \tag{5.20}$$

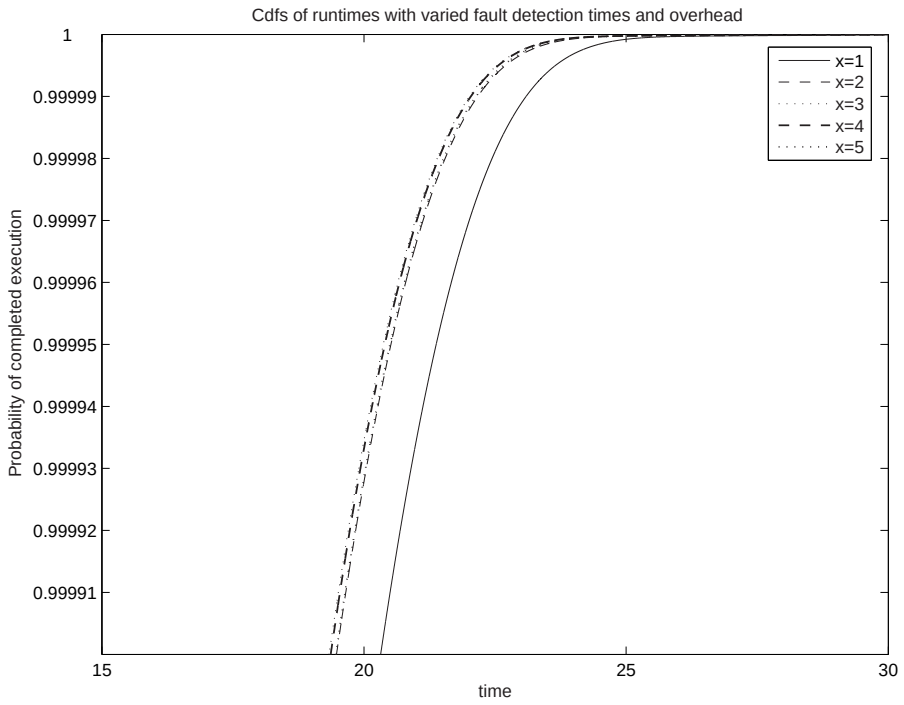


Figure 5.12: Cumulative density functions for the systems in example 5.2, details

The triangular distribution m_α is used for the fault free runtime for primary and backups, with an added constant overhead that increases the execution time by $0.1x$

$$\mathbf{M}_x(s) = e^{-0.1xs}\mathbf{M}_\alpha(s) \tag{5.21}$$

The results for the system with x equal to 1, 2, 3, 4 and 5 is shown in figures 5.11–5.12.

The results show that for the systems in this example, the system with $x = 3$ has the highest probability of completion for $11.3 < t < 24.6$, while $x = 4$ has the highest probability of completion for $24.6 < t < 38.7$. For $t > 38.7$, system with $x = 1$ performs best, as the extra time used for overhead in the other systems also gives a slightly higher fault probability. A summary of the probabilities of

deadline miss for different deadlines is given in table 5.1, with the values for the systems with least probability of deadline misses in boldface.

	Non-	x				
t_{dl}	tolerant	1	2	3	4	5
10	$\mathbf{8.00\times10^{-4}}$	2.07×10^{-3}	5.83×10^{-3}	1.21×10^{-2}	2.09×10^{-2}	3.21×10^{-2}
15	8.00×10^{-4}	5.64×10^{-4}	4.89×10^{-4}	$\mathbf{4.77\times10^{-4}}$	4.80×10^{-4}	4.90×10^{-4}
20	8.00×10^{-4}	1.19×10^{-4}	7.21×10^{-5}	$\mathbf{6.55\times10^{-5}}$	6.68×10^{-5}	7.15×10^{-4}
25	8.00×10^{-4}	7.83×10^{-7}	2.68×10^{-7}	2.33×10^{-7}	$\mathbf{2.31\times10^{-7}}$	2.42×10^{-7}
30	8.00×10^{-4}	8.79×10^{-8}	4.03×10^{-8}	3.32×10^{-8}	$\mathbf{3.25\times10^{-8}}$	3.42×10^{-8}
35	8.00×10^{-4}	6.09×10^{-9}	1.81×10^{-9}	1.51×10^{-9}	$\mathbf{1.51\times10^{-9}}$	1.60×10^{-9}
40	8.00×10^{-4}	$\mathbf{9.18\times10^{-10}}$	9.37×10^{-10}	9.66×10^{-10}	9.97×10^{-10}	1.02×10^{-9}

Table 5.1: Probability of deadline miss for the systems described in example 5.2 at different deadlines

Example 5.3 A simple inhomogeneous system

In this example, the model of a system where the primary and backup distributions vary is presented. The primary uses the triangular distribution m_α, the backups use the deterministic m_β. The detection and correction times are distributed as in example 5.1, d_α and c_α respectively. The mean time between faults is set to 10 000. Simulator results are used for comparison, and the simulator is set to run the task 500 000 times.

The goal with this example is to show that the mathematical models for inhomogeneous watchdog-based systems can be used when the runtime distribution for the primary and the backups vary.

The resulting cumulative distributions are shown in figure 5.13.

From the figure, it can be seen that the results from the simulated system seems "worse" than the results from the numerical calculation, with a slightly higher fault rate. As with the systems in example 5.1, the number of simulator runs that are affected by faults should have been 400 for an exact match, while in the simulation, the number of affected runs were 414, 3.5% more than expected. Apart from the higher than expected number of faults, the curve for the simulated results follows the curve for the calculated results closely.

Example 5.4 A system utilizing imprecise backups

In this example, a system using fast, but imprecise backups is modeled. For the primary's runtime distribution, the triangular distribution m_α is used, for

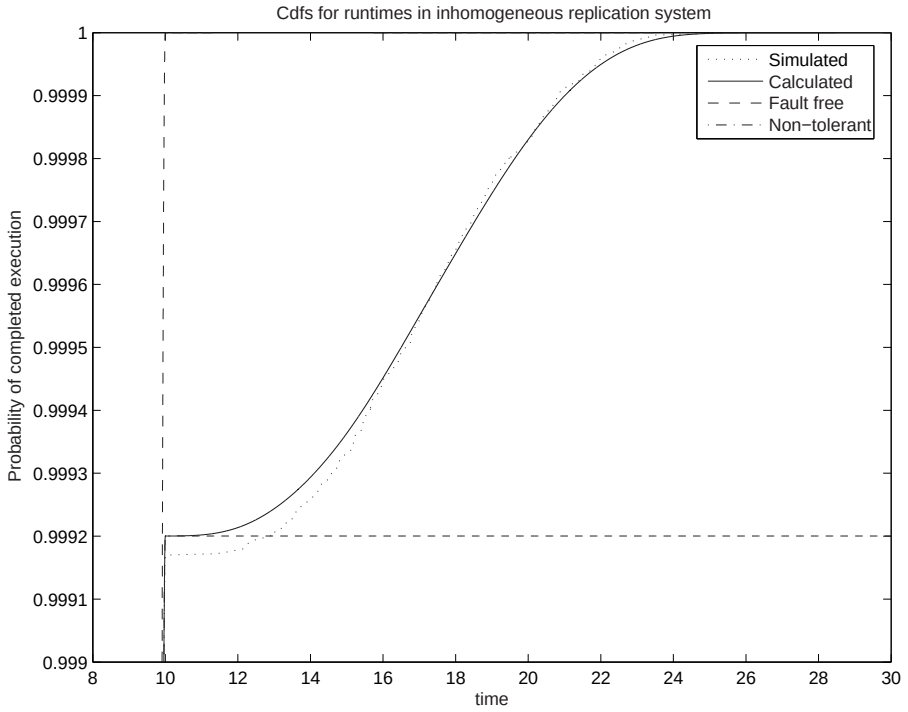


Figure 5.13: Cumulative density functions for the inhomogeneous watchdog system described in example 5.3.

the backup runtimes, the fast deterministic distribution m_γ is used. The fault detection time is the same for all replicas, d_α. It is assumed that the time used to bring the backups up to date is shorter than for the other examples, so the correction time distribution c_γ is used. The mean time between faults, $\frac{1}{\lambda}$ is set to 10 000. For the simulated results, the simulator was set to 500 000 runs.

The resulting cumulative distributions, compared to the results of the "precise" inhomogeneous system described in example 5.1, is shown in figure 5.14. The system with imprecise backups has a better temporal behavior than the precise system, which is quite notable for times less than 22. If the execution of the task in this example has a hard deadline $t_{dl} < 22$ and the system is able to tolerate impreciseness in some of the results, the use of imprecise backups

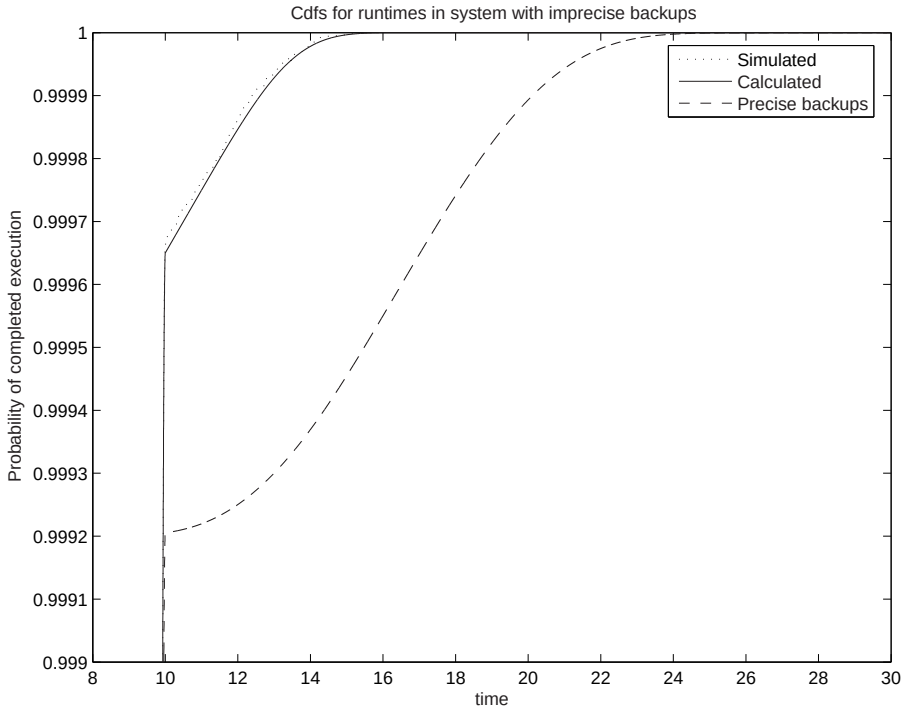


Figure 5.14: Cumulative density functions for the system with imprecise backups described in example 5.4.

would greatly improve the performance of the system.

Note that the primary runs do *not* experience fewer faults in this system than in the system with precise backups, even if the curves in figure 5.14 may give this impression. The reason of the seemingly lower fault rate is because early detection of faults in the primary may cause the backup to finish its run before the normal worst-case execution time of the primary.

The simulator results seem to follow the calculated results. As with the other systems using the m_α distribution as primary and a mean time between faults of 10 000, the expected number of fault-affected simulator runs are 400 out of the 500 000. The results shows 415 affected runs, 3.75% more than expected. For the timing region shown in figure 5.14, the simulation shows a slightly higher

probability of task completion than the calculated results.

5.3.3 Systems using timeout as a fault detection method

In the next examples, systems using timeout as a fault detection method are presented.

As with the watchdog systems, a structure with one primary and two backups is used for both the examples. For these examples, only the inhomogeneous system where more than N faults leads to the failure of the system is used. Using the equation 4.80 with $N = 2$, we get the following equation to use in the examples:

$$\begin{aligned}&\mathbf{G}(s) = \\ &\quad \mathbf{M}_0(\lambda+s) \\ &\quad + \mathbf{M}_1(\lambda+s)\mathbf{C}_1(\lambda+s)e^{-s\tau_{m0}}(1-\mathbf{M}_0(\lambda)) \\ &\quad + \mathbf{M}_2(\lambda+s)\mathbf{C}_2(\lambda+s)e^{-s\tau_{m0}}(1-\mathbf{M}_0(\lambda)) \\ &\quad\ (e^{-s\tau_{c1}}(1-\mathbf{C}_1(\lambda)) + e^{-s\tau_{m1}}\mathbf{C}_1(\lambda+s)(1-\mathbf{M}_1(\lambda)))\end{aligned} \tag{5.22}$$

Example 5.5 A basic system

As with the first example for watchdog systems, the triangular distribution m_α is used as runtime distribution for both primary and backup methods, and c_α is used for the fault correction distribution. The timeout for the task execution, τ_{mi}, is set to 10 for both primary and backups, and the timeout for fault correction, τ_{ci}, is set to 5. The mean time between faults, $\frac{1}{\lambda}$, is set to 10 000. For the simulated results, the simulator was set to run the task 500 000 times.

The calculated and simulated results are shown in figure 5.15. As is to be expected, there is no improvement in the system's fault tolerance if there is a hard deadline at $t < 16$, as the completion of a rerun of the task is not possible before this time. With deadlines at $t < 24$, the improvement in the system's fault tolerance is quite good.

Compared to the watchdog system in example 5.1, the timeout system performs worse, as faults are detected later. It should be noted that this example and example 5.1 both use the same distributions, without modeling any extra overhead due to "I'm alive" signaling, which is needed for the watchdog system, but not for the timeout system. Implementing a timeout mechanism is fairly simple, and it can also detect omissions from tasks that suffer from non-silent failures.

The simulator results follow the calculated results quite closely for this example, with faults in 395 of the 500 000 runs, the expected number was 400.

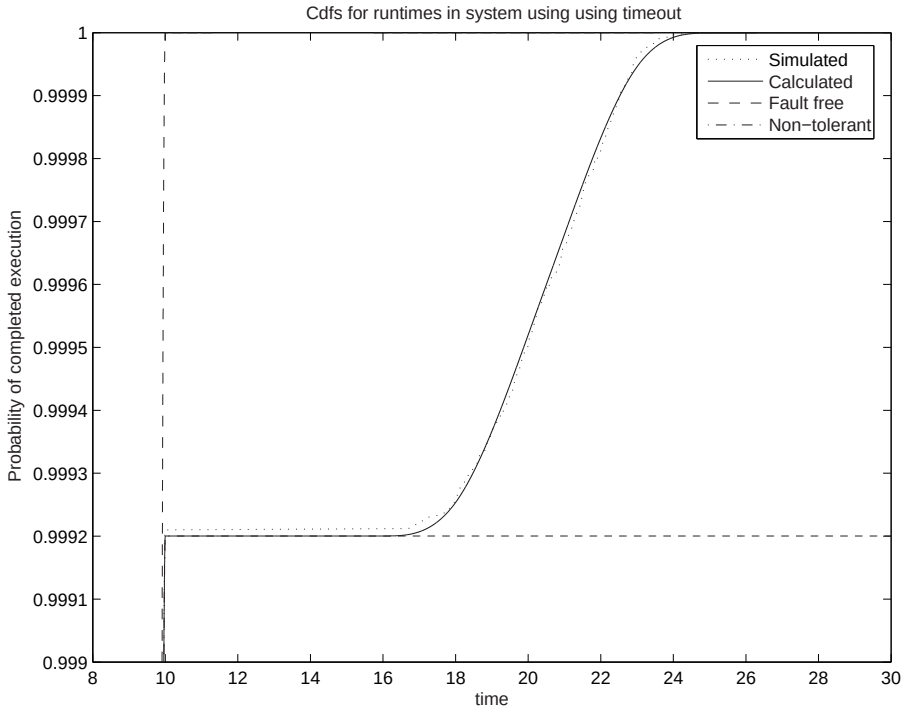


Figure 5.15: Cumulative distribution function for the system with timeout fault detection described in example 5.5.

Example 5.6 Using imprecise backups

In this example, a system with timeout fault detection and fast, imprecise backups is modeled.

In this example, the triangular distribution m_α is used as the runtime distribution for the primary method, while the fast deterministic m_γ is used as the runtime distribution for the backups. For the correction time distribution, c_γ is used. Timeouts are set to the worst case time use for each distribution, i.e. $\tau_{m0} = 10$, $\tau_{m1} = 3$, and $\tau_{c1} = 1$. As with earlier examples, the mean time between faults, $\frac{1}{\lambda}$, is set to 10 000, and the simulator is set to run the system 500 000 times.

The calculated and simulated results, compared to the calculated results

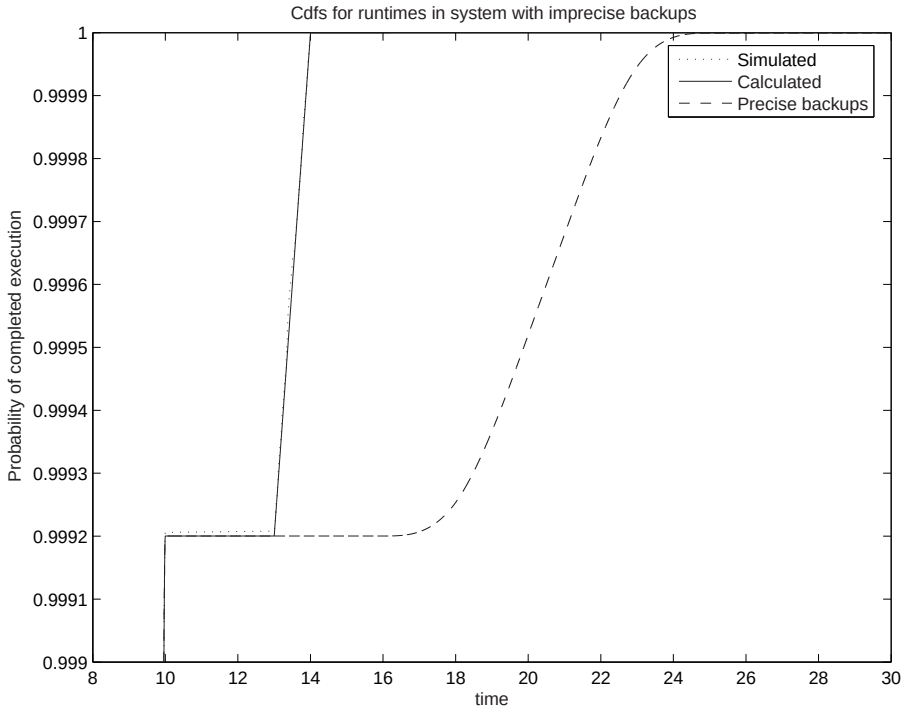


Figure 5.16: Cumulative distribution function for the system with timeout fault detection and imprecise backups described in example 5.6.

of the precise system from example 5.5 are shown in figure 5.16. As can be expected, the use of the imprecise backups results in a great temporal improvement, which can be useful if the system has deadlines between 14 and 25.

In this example, there is little difference between the simulated and calculated results. Of the 500 000 simulator runs, 397 runs suffered from faults, the expected number was 400.

5.3.4 Systems using acceptance test as a fault detection method

In the next example, a system using acceptance test as a fault detection method is presented. For the example, equation 4.102 with $N = 2$ is used:

$$\begin{aligned}&\mathbf{G}(s) = \\ &\quad \mathbf{M}_0(\lambda+s)\mathbf{D}(s) \\ &\quad + \mathbf{M}_1(\lambda+s)\mathbf{C}_1(\lambda+s)\mathbf{D}(2s)(\mathbf{M}_0(s) - \mathbf{M}_0(\lambda+s)) \\ &\quad + \mathbf{M}_2(\lambda+s)\mathbf{C}_2(\lambda+s)\mathbf{D}(3s)(\mathbf{M}_0(s) - \mathbf{M}_0(\lambda+s)) \\ &\quad (\mathbf{M}_1(s)\mathbf{C}_1(s) - \mathbf{M}_1(\lambda+s)\mathbf{C}_1(\lambda+s)\end{aligned} \tag{5.23}$$

Example 5.7 A basic acceptance test system

As with the first examples using watchdog and timeout fault detection, the triangular distribution m_α is used as the runtime distribution for the primary and backups for this system, and the distribution c_α is used for correction time. The time used for running the test is distributed with d_β. The mean time between faults, $\frac{1}{\lambda}$ is 10 000. As with the other systems, the simulator is set to run the task 500 000 times.

Simulated and calculated results are shown together with calculated results for a fault-free and non-fault tolerant system in figure 5.17. Note that for the fault-free and non-fault tolerant systems, there is no test time.

As with the timeout systems, systems with deadlines near the primary's WCET gain no improvement in reliability from using acceptance test as a fault detection method for this system. It is, however, the only of the modeled fault detection method that can detect if a value failure has occurred.

Simulation shows a higher fault rate than expected, with 418 fault-affected runs in of the 500 000. As with the other systems, the expected number of affected runs were 400. The curves of the calculated and simulated results otherwise seems to follow each other.

5.3.5 Systems combining acceptance test and timeout

As with the other examples, a system consisting of one primary and two backups are used for the acceptance test and timeout combination system.

By using equation 4.121 with $N = 2$ we get the following equation for the

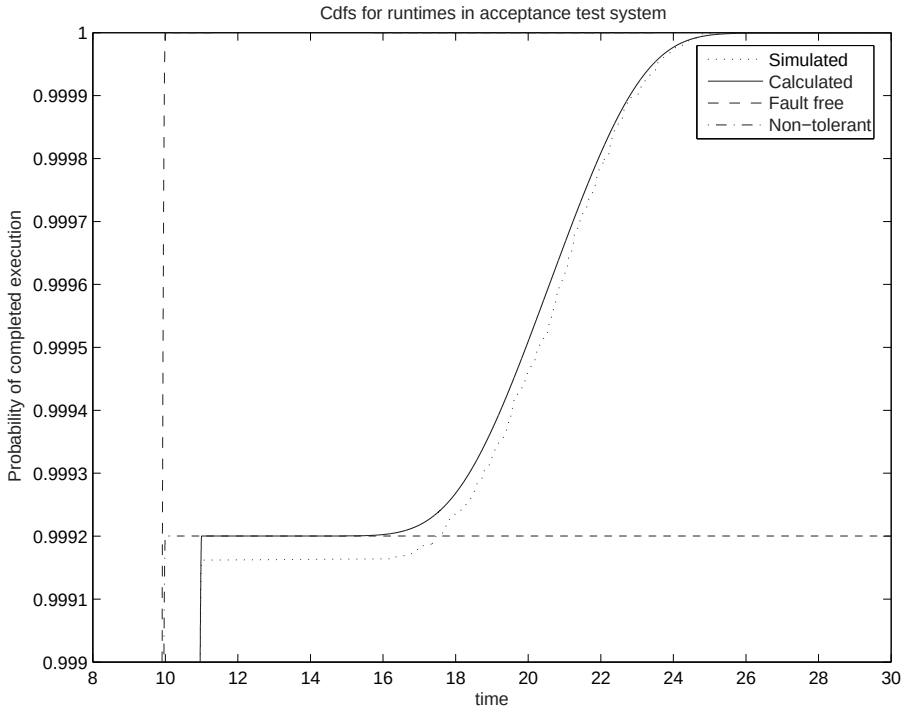


Figure 5.17: Cumulative density functions for the acceptance test system in example 5.7.

examples.

$$
\begin{aligned}
&\mathbf{G}(s) = \\
&\quad \kappa_{\mathrm{mo}0}\kappa_{\mathrm{v}0}\mathbf{M}_0(s)\mathbf{D}(s) \\
&\quad + \kappa_{\mathrm{co}1}\kappa_{\mathrm{mo}1}\kappa_{\mathrm{v}1}\mathbf{M}_1(s)\mathbf{C}_1(s)\mathbf{D}(s) \\
&\qquad ((1-\kappa_{\mathrm{mo}0})e^{-s\tau_{\mathrm{m}0}} + (1-\kappa_{\mathrm{v}0})\kappa_{\mathrm{mo}0}\mathbf{M}_0(s)\mathbf{D}(s)) \\
&\quad + \kappa_{\mathrm{co}2}\kappa_{\mathrm{mo}2}\kappa_{\mathrm{v}2}\mathbf{M}_2(s)\mathbf{C}_2(s)\mathbf{D}(s) \\
&\qquad ((1-\kappa_{\mathrm{mo}0})e^{-s\tau_{\mathrm{m}0}} + (1-\kappa_{\mathrm{v}0})\kappa_{\mathrm{mo}0}\mathbf{M}_0(s)\mathbf{D}(s)) \\
&\qquad ((1-\kappa_{\mathrm{co}1})e^{-s\tau_{\mathrm{c}1}} + (1-\kappa_{\mathrm{mo}1})\kappa_{\mathrm{co}1}\mathbf{C}_1(s)e^{-s\tau_{\mathrm{c}1}} \\
&\qquad + (1-\kappa_{\mathrm{v}1})\kappa_{\mathrm{co}1}\kappa_{\mathrm{mo}1}\mathbf{M}_1(s)\mathbf{C}_1(s)\mathbf{D}(s))
\end{aligned}
\tag{5.24}
$$

For comparison, systems with the same fault mechanisms, but without the

detection mechanisms is used, where an undetected fault leads to the failure of the system

For a system without timeout mechanisms, the equation for the runtime model can be found by using equation 5.24 and letting the timeout values τ_{co} and τ_{mo} approach ∞, i.e. let $e^{-s\tau} = 0$.

$$\begin{aligned}&\mathbf{G}_{\mathrm{noto}}(s) = \\ &\quad \kappa_{\mathrm{mo}0}\kappa_{\mathrm{v}0}\mathbf{M}_0(s)\mathbf{D}(s) \\ &\quad + \kappa_{\mathrm{co}1}\kappa_{\mathrm{mo}1}\kappa_{\mathrm{v}1}\mathbf{M}_1(s)\mathbf{C}_1(s)\mathbf{D}(s)(1-\kappa_{\mathrm{v}0})\kappa_{\mathrm{mo}0}\mathbf{M}_0(s)\mathbf{D}(s) \\ &\quad + \kappa_{\mathrm{co}2}\kappa_{\mathrm{mo}2}\kappa_{\mathrm{v}2}\mathbf{M}_2(s)\mathbf{C}_2(s)\mathbf{D}(s)(1-\kappa_{\mathrm{v}0})\kappa_{\mathrm{mo}0}\mathbf{M}_0(s)\mathbf{D}(s) \\ &\quad (1-\kappa_{\mathrm{v}1})\kappa_{\mathrm{co}1}\kappa_{\mathrm{mo}1}\mathbf{M}_1(s)\mathbf{C}_1(s)\mathbf{D}(s)\end{aligned} \tag{5.25}$$

For a system with no detection on value failures, equation 4.98 is used with $N = 2$ and with $\mathbf{M}_i(s)$ substituted with $\kappa_{\mathrm{v}i}\mathbf{M}_i(s)$. In this model, tasks suffering from undetected value failures will appear as having an infinite runtime.

$$\begin{aligned}&\mathbf{G}(s) = \\ &\quad \kappa_{\mathrm{mo}0}\kappa_{\mathrm{v}0}\mathbf{M}_0(s) \\ &\quad + \kappa_{\mathrm{co}1}\kappa_{\mathrm{mo}1}\kappa_{\mathrm{v}1}\mathbf{M}_1(s)\mathbf{C}_1(s)(1-\kappa_{\mathrm{mo}0})e^{-s\tau_{\mathrm{m}0}} \\ &\quad + \kappa_{\mathrm{co}2}\kappa_{\mathrm{mo}2}\kappa_{\mathrm{v}2}\mathbf{M}_2(s)\mathbf{C}_2(s)(1-\kappa_{\mathrm{mo}0})e^{-s\tau_{\mathrm{m}0}} \\ &\quad (1-\kappa_{\mathrm{co}1})e^{-s\tau_{\mathrm{c}1}} + (1-\kappa_{\mathrm{mo}1})\kappa_{\mathrm{co}1}\mathbf{C}_1(s)e^{-s\tau_{\mathrm{c}1}}\end{aligned} \tag{5.26}$$

Example 5.8 A simple system combining timeout and acceptance test

In this example, the triangular distribution m_α is used for primary and backups, the uniform distribution c_α is used for correction time distribution, and d_β is used for the test time distribution. The timeout for method execution, $\tau_{\mathrm{m}i}$, is set to 10, and the timeout for correction, $\tau_{\mathrm{c}i}$ is set to 5 for all the replicas.

The probability that replica i should fail due to an omission or crash failure during method execution, $1-\kappa_{\mathrm{mo}i}$, or a value failure, $1-\kappa_{\mathrm{v}i}$, is both set to 4×10^{-4}, and the probability of an omission or crash during correction, $1-\kappa_{\mathrm{co}i}$ is set to 2.5×10^{-4} for all i. These values gives approximately the same fault probabilites as if the faults were generated by a poisson process with mean time between fault occurrences of 10 000, which is used in most of the other examples presented here.

The results is shown in figure 5.18, together with the results of system having only one of the detection mechanisms.

As can be expected, the combined fault tolerance mechanism performs better than the systems having only one of the mechanisms. As both the systems with only one mechanism have a probability of an *undetected* failure of 4×10^{-4} in the

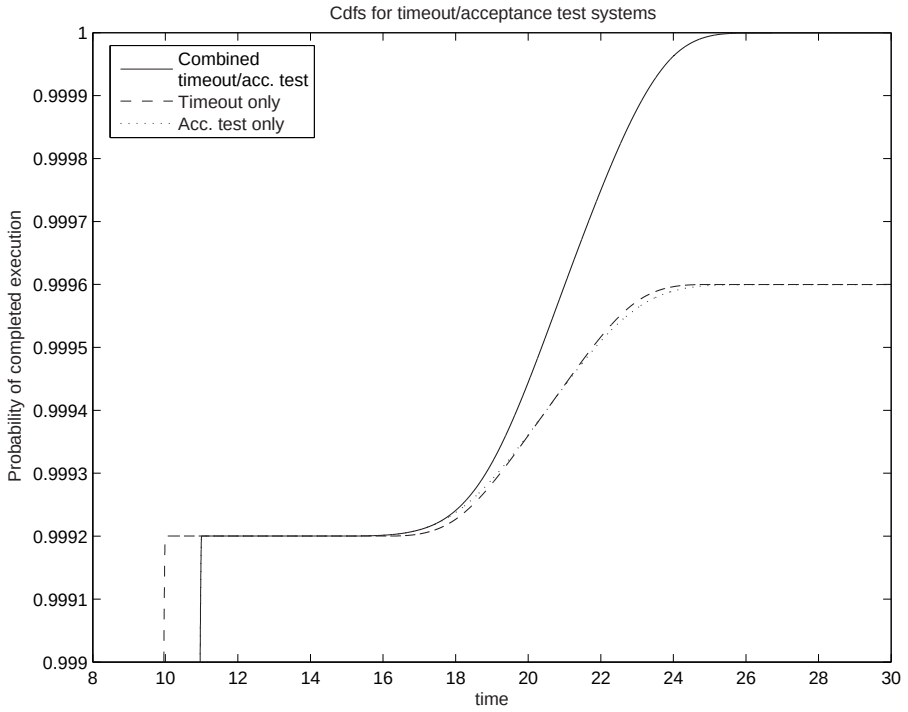


Figure 5.18: Cumulative distribution function for the combined timeout and acceptance test system described in example 5.8, compared to systems with the same fault probabilities, but with only one of the fault detection methods.

primary run, tasks in these systems will never have a probability of successful completion above $1 - 4 \times 10^{-4}$.

For systems hard deadlines earlier than $t = 20$, the improvement in reliability is not very high, as there is too little time to detect failures and rerun the task within the deadline, This reliability increases steadily with higher deadlines, and at $t_{dl} = 26$, the improvement is quite good, with a failure probability of 7.8×10^{-7}.

Example 5.9 Use of checkpoints

In this example, a task consisting of 12 subtasks is investigated. The task

is partitioned into m parts consisting of n subtasks, so that $mn = 12$. The goal with the example is to examine the best way of partitioning the task for different deadlines.

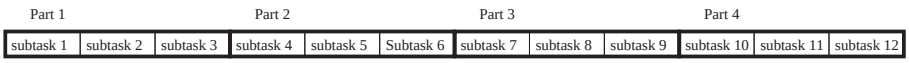


Figure 5.19: A task partitioned in 4 parts, each consisting of 3 subtasks.

Each of the 12 subtasks has a fault-free runtime that is uniformly distributed between 1 and 2.

$$\mathbf{M}_{\text{sub}}(s) = \frac{e^{-s} - e^{-2s}}{s} \tag{5.27}$$

For a part consisting of n subtasks, the fault-free runtime for the part will be distributed as a convolution of n subtasks, resulting in the moment-generating function

$$\mathbf{M}(s) = \mathbf{M}_{\text{sub}}(s)^n \tag{5.28}$$

The timeout is set at the maximum runtime for a part. As each subtask has a maximum runtime of 2, the timeout value of a part consisting of n subtasks becomes

$$\tau_{\text{m}} = 2n \tag{5.29}$$

The time used to run an acceptance test, in addition to other overhead, is 1 per part

$$\mathbf{D}(s) = e^{-s} \tag{5.30}$$

The correction time is also 1 per part

$$\mathbf{C}(s) = e^{-s} \tag{5.31}$$

The probability that a subtask experiences an omission failure during normal runtime $(1 - \kappa_{\text{mosub}})$ or a value failure $(1 - \kappa_{\text{vsub}})$ is both 10^{-4}. The probability of an omission failure during correction is also 10^{-4}.

$$\kappa_{\text{mosub}} = \kappa_{\text{co}} = \kappa_{\text{vsub}} = 1 - 10^{-4} \tag{5.32}$$

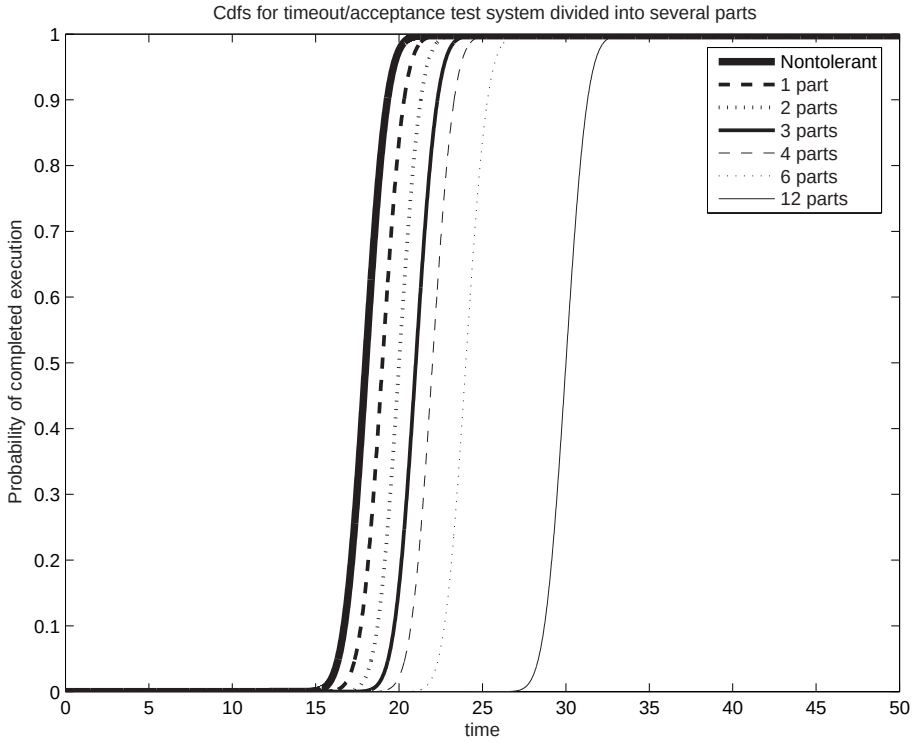


Figure 5.20: Comparison of the cdfs of distributions for a task divided into different numbers of parts

For parts consisting of n subtasks, a failure in one subtask means the whole part has failed. Thus, for a part consisting of n subtasks, the probabilities of not experiencing failures are

$$\kappa_{\mathrm{mo}} = \kappa_{\mathrm{mosub}}^{n} \tag{5.33}$$

$$\kappa_{\mathrm{v}} = \kappa_{\mathrm{vsub}}^{n} \tag{5.34}$$

The runtime distribution and fault probability is the same for primary and backups. If a given part fails more than 2 times during the execution of a single task, the whole task is considered failed. Thus, equation 5.24 can be used for the runtime distribution for each part.

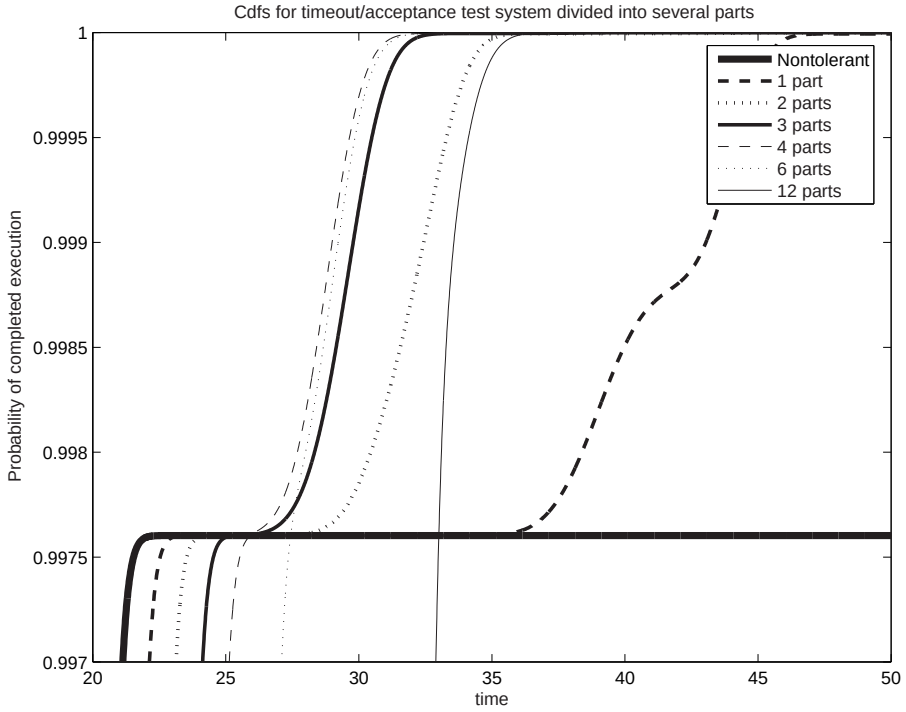


Figure 5.21: Comparison of the cdfs of distributions for a task divided into different numbers of parts, details

The runtime distribution for a task consisting of m parts, each with a runtime distribution $\mathbf{G}_i(s)$ is found using equation 4.122:

$$\mathbf{G}(s) = \prod_{i=1}^{m} \mathbf{G}_i(s)$$

The task partitions that are used in this example are

- 1 part of 12 subtasks (i.e., no division of the task)
- 2 parts of 6 subtasks
- 3 parts of 4 subtasks

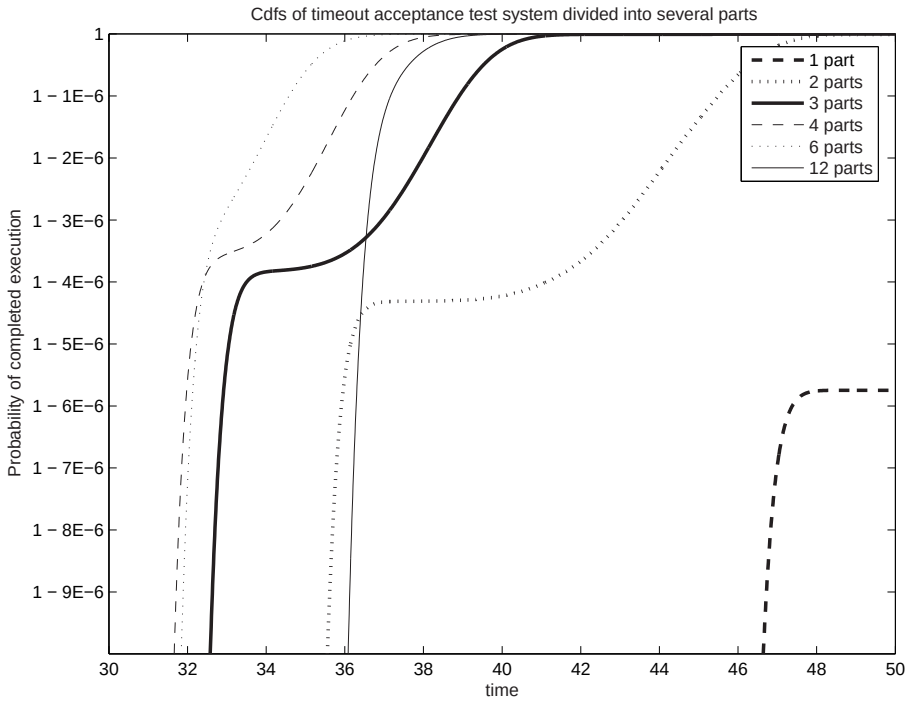


Figure 5.22: Comparison of the cdfs of distributions for a task divided into different numbers of parts, details

- 4 parts of 3 subtasks
- 6 parts of 2 subtasks
- 12 parts of 1 subtask

The cumulative distribution function of the results are shown in figures 5.20–5.22.

If there is a hard deadline at t_{dl}, and the only possibilities of a system failure is a deadline miss or a single task failing more than 2 times, examining the results show that a system without fault tolerance performs better than or as well as the fault tolerant systems at $t_{dl} < 25.4$. This system does not have the overhead from the fault tolerance mechanisms, and there is a high probability that if a

fault occurs in any of the other systems, it will not be tolerated within the time limits.

For $25.4 < t_{dl} < 26.0$, the 3-part system performs best, but only slightly better than a non-tolerant system.

For $26.0 < t_{dl} < 32.4$, the 4-part system performs best, and the improvement in fault tolerance begins to show; the system failure probability for this system is more than halved from the non-tolerant system at $t_{dl} = 29$.

The 6-part system performs best for $32.4 < t_{dl} < 40.9$, and the 12-part system performs best for $t_{dl} > 40.9$. At around $t_{dl} = 40$, the system failure probabilities for these systems are in the 10^{-9} range.

A summary of the deadline miss probabilities for different deadlines is shown in table 5.2

	Non-	Number of parts					
t_{dl}	tolerant	1	2	3	4	6	12
24	$\mathbf{2.40\times10^{-3}}$	2.40×10^{-3}	2.41×10^{-3}	3.40×10^{-3}	2.46×10^{-2}	0.501	1.00
25	$\mathbf{2.40\times10^{-3}}$	$\mathbf{2.40\times10^{-3}}$	2.40×10^{-3}	2.41×10^{-3}	3.40×10^{-3}	0.163	1.00
26	2.40×10^{-3}	2.40×10^{-3}	2.40×10^{-3}	$\mathbf{2.39\times10^{-3}}$	2.39×10^{-3}	2.46×10^{-2}	1.00
28	2.40×10^{-3}	2.40×10^{-3}	2.39×10^{-3}	2.14×10^{-3}	$\mathbf{1.81\times10^{-3}}$	2.00×10^{-3}	0.978
30	2.40×10^{-3}	2.40×10^{-3}	2.15×10^{-3}	8.36×10^{-4}	$\mathbf{3.05\times10^{-4}}$	4.10×10^{-4}	0.501
32	2.40×10^{-3}	2.40×10^{-3}	1.26×10^{-3}	3.56×10^{-5}	$\mathbf{5.59\times10^{-6}}$	7.25×10^{-6}	2.43×10^{-2}
35	2.40×10^{-3}	2.39×10^{-3}	3.15×10^{-5}	3.77×10^{-6}	2.41×10^{-6}	$\mathbf{6.02\times10^{-7}}$	1.20×10^{-4}
40	2.40×10^{-3}	1.49×10^{-3}	4.23×10^{-6}	2.48×10^{-7}	4.52×10^{-9}	$\mathbf{1.20\times10^{-9}}$	3.57×10^{-9}
45	2.40×10^{-3}	1.98×10^{-4}	1.38×10^{-6}	4.94×10^{-9}	1.21×10^{-9}	6.00×10^{-10}	$\mathbf{2.16\times10^{-10}}$

Table 5.2: Probability of deadline miss for the systems described in example 5.9 at different deadlines

Chapter 6

Discussions and suggestions for future work

This chapter contains discussions about the developed models, as well as some suggestions for future work.

6.1 Fault models

In the runtime models developed, the fault models have been relatively simple, either using a poisson arrival fault process, or a constant success probability for each replica. These fault models are fairly easy to understand and use in mathematical models, and are thought to be "good enough" for modeling in many systems.

The timeframe of the models is the runtime of one task, and the use of more complex fault models may often be unnecessary. For a larger timeframes, e.g. when using the results to calculate the reliability of the system as a function of the number of runs like in example 5.1, modeling effects like maturity and wear can be done by changing the fault rates for each run.

For all systems using the constant success probability and for one of the systems using the poisson arrival process (section 4.2.4, the fault probabilities can be changed within the same system, so that primaries and backups may have different fault probabilities. While it is not shown in the work, all the derived models using the poisson arrival fault process can be modified so that fault rates may vary between the replicas.

Faults resulting in failure modes that are not detected by the fault detection mechanism and faults that are not tolerated by the system are not part of any of the models, i.e. only the faults that the fault tolerant system is able to handle are modeled. The unmodeled faults may be important in the analysis of a system. In section 4.5.1, a system with several fault types was modeled, and in example 5.8 it was shown how the lack of a fault detection mechanism could be modeled. Similar expansions can be done with the other models. For models where faults leading to different failure modes are modeled, dependencies between failures of the same mode may cause the models to become very complex.

6.2 Models of other systems

While the models presented in chapter 4 cover a wide range of passively replicated systems, there will always be systems that do not fit into any of these models. In section 4.6, some concrete examples of system structures where the derived models cannot be used "as they are" are presented. It is believed that the method for deriving runtime distributions used in this work can be used for many other system models.

In the models presented here, the effect on the runtimes when running tasks in a computer system also running several other tasks is not considered. Scheduling and queuing of several differently prioritized fault tolerant tasks will of course affect the total time from a task is ready till it finishes. The models presented here can be used as an "actual processing time" model for tasks in a multitask scheduling or queuing system model.

6.3 Use of the models

A major problem with the models presented here is that the use of the models is not very easy. The use of moment-generating functions cannot be considered to be "well known", and the transforms between time-based distribution functions and moment generating functions may be difficult for many users. Creating computer tools that provide an easy to use interface to the models is therefore necessary.

To use the models, it is assumed that the fault rates and several distributions, like the distributions for fault free runtimes and the correction times are known, which may not be the case. Using approximations to fault rates and distributions may be necessary, which will result in some uncertainty to the re-

sults. It can, however, be assumed that the same approximations must be used for other models.

Chapter 7

Conclusion

In this work, mathematical run-time distribution models for several system classes using passive replication are developed.

The models are expressed as functions of the modeled system's fault rates, the fault free runtime distributions, the distributions of the times used to ready one of the backups, as well as the distributions of the time used to detect a fault, the time used to test the results with an acceptance test and/or the system's timeout values, depending on the fault detection mechanisms used.

It has been argued that the main differences between the models are the fault detection mechanisms used and whether the replicas are homogeneous or inhomogeneous. Other variants of the system structure can often be described as changes of the parameter distributions.

The models that were developed were models for systems using watchdogs, timeouts and acceptance tests as fault detection. For each test type, two models, one with homogeneous and one with inhomogeneous replicas, was developed using a poisson fault arrival process as fault model. For the watchdog fault detection system, an additional model for an inhomogeneous system using different fault rates for the different replicas was developed. For the systems using timeout and acceptance test, additional models using inhomogeneous replicas and a fixed fault probability for each of the replicas was developed. The model of a system using fixed fault probabilities and both timeout and acceptance test for fault detection was also developed.

How the models can be used has been demonstrated with simple examples, among these are examples on how the results can be used for finding probability of missed deadlines, determine the reliability of a system as a function of num-

ber of runs, and optimizing the "I'm alive" signalling frequency or checkpoint placement with respect to hard deadlines. Some of the results were compared to results from a discrete event simulator that was developed for this purpose.

Not only the models themselves, but also the method used to derive the models is important. While this method is similar to methods used in many models in queuing theory, this work has shown that it can also be used for deriving the runtime distribution models of tasks in passively replicated fault tolerant systems. It is also believed that the same method can be used for deriving the run-time distribution model for many systems not presented in this work.

Appendix A

Fault Tolerance Methods in Component-Based Real-Time Systems

By Åsmund Tjora and Amund Skavhaug

This extended abstract was presented at the work in progress session held in connection with the Euromicro DSD Symposium and Euromicro Conference in Dortmund, Germany, September 2002 [45].

A.1 Introduction

Many, if not most, of today's real-time applications will, in addition to timing requirements, also have fault tolerance requirements. It is therefore important to look at the combination of real-time and fault tolerant systems in order to determine if the use of a fault tolerance mechanism will cause a missed deadline. For a real-time system, a missed deadline is considered a fault, and it is therefore necessary to analyze the time used to tolerate a fault if the fault happens during a method call. If a fault is tolerated, but the mechanisms used to tolerate the fault leads to a missed deadline, the mechanisms may be unsuitable for a real-time system, depending on the real-time system's characteristics.

OMG's CORBA standard is the open standard technology base for heterogeneous systems, and it is used in a wide variety of applications. Therefore, we have studied the CORBA specification for fault tolerant systems. The replication styles described in Fault Tolerant CORBA we are looking at are COLD_PASSIVE, WARM_PASSIVE, ACTIVE and ACTIVE_WITH_VOTING.

For these, we want to find the time and resource overhead both in normal operation, and time use in a fault situation.

A.2 Analysis

We are currently using a very simple model for determining the time used in case of a fault. The worst case time use is calculated adding the time used to run a method, t_m, to the time used to detect a fault, t_d, and the time used for the fault tolerance mechanism to tolerate the fault, t_{lcp} for cold passive replication, and t_{lwp} for warm passive replication. We are only considering cases where only one fault occurs during a method call. This is a reasonable assumption if we don't consider correlated faults. (Correlated faults are not tolerated in the methods described in the FT CORBA specification.)

A.2.1 Cold passive replication

The COLD_PASSIVE replication style is based on having several backup (secondary) objects, and copying the state of primary object into a secondary if the primary object fails. The state of the primary object is logged, and when the primary fails, the state of the new primary is set from the log.

At normal operation, this replication style uses relatively little resources and time. There will be some overhead as the object's state is logged, but in most

cases, it should not be necessary to do this operation during a time critical method call.

If the primary object fails, the state of one of the secondary objects must be set from the log. The method must be run again on the new object. For a worst case, the time used to run a method on an object that fails is $t_d + t_{lcp} + 2t_m$ where t_d is the time used to detect a fault, t_{lcp} is the time used to set the state of the secondary object from log and make it the primary, and t_m is the time used to run the method. Depending on the relative length of these times and the type of deadlines, this replication style can be unusable (i.e. if there are hard deadlines, or if t_d and t_{lcd} are long compared to t_m) or they may be usable (i.e. the system tolerates some missed deadlines, and t_d and t_{lcd} are short compared to t_m)

A.2.2 Warm passive replication

The WARM_PASSIVE replication style is based on having a set of passive objects, and periodically copying the state of the active into the passive objects. Because of this, if the active object fails, the time to activate a passive object will be shorter than with the COLD_PASSIVE method.

At normal operation, the time and resource use will be small, as with COLD_PASSIVE. Updating the states of the passive objects will take more time than just logging the state of the active object, but as with COLD_PASSIVE, it is usually not necessary to do this during time-critical operations.

In case of a fault, this replication style has the advantage of having the secondary objects in a state that is close to the state of the primary object. Because of this, only the latest updates to the secondary object's state have to be made. The time to update the state of the secondary object, tlwp, is therefore smaller than the time used to set and update a secondary object using the COLD_PASSIVE replication style, tlcp. The worst case will have the same form as with COLD_PASSIVE, $t_d + t_{lwp} + 2t_m$.

A.2.3 Active replication

With the ACTIVE replication style, there are several objects that are running the method calls. Because of this, all objects in a replicated group will have the same state.

At normal operation, this replication style will have a higher time and resource use than the PASSIVE replication styles, as all the replicated objects

must run the method. The resource manager must also suppress duplicate requests and replies from the replica objects, so that the object group acts as only one object. For a case where all the objects in the group are running on one processor, the time used for a normal call will be n tm + trm where n is the number of objects in the group and trm is the resource manager overhead. Because it may be necessary to create new objects when one of the objects in the group fails, this replication style will also need a state logging mechanism.

In case of a fault, the replicas will still be running, and the time use will not change dramatically. This makes the time used for a method call deterministic, even in the case where one of the objects in the group fails. If the system tolerates the time used for normal operation, it should also tolerate the time used in case of a fault.

If the time used to run a method call is high with respect to the time used to make a passive object active, the PASSIVE methods should be more effective than the ACTIVE.

A.2.4 Active replication with voting

ACTIVE_WITH_VOTING is a replication style that is similar to ACTIVE, but results from the different replicas are voted upon. This method will thus tolerate some faults that the other methods don't tolerate. However, as the ACTIVE replication style, this style uses more time and resources in normal operation than the PASSIVE replication styles.

A.3 Discussion

As mentioned, the model we currently use to determine time use is very simple, and is only intended to get a rough overview of the replication styles described. For many cases, it may even be too simple to get a good overview of which of the replication styles are best suited for the real-time system.

Although we, for the simple model, consider it a reasonable assumption to ignore the possibility of multiple faults during one method execution, this possibility should be considered in more complex models.

A.4 Future work

In our future work, we are going to develop a better model for time use in fault situations. We want to create a probability density distribution for the time

use of a (possibly faulty) method execution. In this model, we will also look at the possibility for multiple independent faults occurring in the total method execution time. The goal of this model is to make it easier to determine the probability of fault recoveries leading to deadline faults.

We also want to look at other mechanisms in the FT CORBA specification, such as the fault detection and the logging mechanisms. As there are two ways of logging the state of an object, logging its entire state or logging the updates to the state, it may be useful to investigate what the best combination of these two are in different kind of systems.

Appendix B

A General Mathematical Model for Run-Time Distributions in a Passively Replicated Fault Tolerant System

By Åsmund Tjora and Amund Skavhaug

This paper was presented at the Euromicro Conference on Real-Time Systems in Porto, Portugal, July 2003 [46].

Abstract

In many systems, passive replication is used as a method for fault tolerance. A problem with using passive replication i real-time systems is that it can be difficult to analyze the time used by the system if a fault should occur.

In this paper we present a general mathematical model for the run-time distribution of a method in a fault tolerant system where a passive replication technique is used. The model gives this distribution as a function of the run-time distribution of the method in a fault-free system. We also demonstrate how this can be used to compute the probability of a missed deadline in a simple fault-tolerant system.

B.1 Introduction

Many of today's real-time applications will, in addition to the timing requirements, also have fault tolerance requirements. It is therefore important to look at the combination of real-time and fault tolerant systems, in order to determine if the use of a fault tolerance mechanism will cause a missed deadline. It may be necessary to analyze the time used by the fault tolerance mechanisms when a fault occurs, since a missed deadline is also considered a fault in a real-time system. If a fault is tolerated, but the mechanism used to tolerate the fault leads to a missed deadline, this mechanism may be unsuitable for a real-time system, depending on the characteristics of the system.

In real-time systems, it has been common to use some kind of active replication for fault tolerance, that is, several replicas of one object is running at the same time. This gives a more deterministic temporal behaviour in case of faults, and it also provides the possibility for detecting a wider range of faults, since the results from the different replicas can be compared. However, this replication strategy also uses a lot of resources.

For general systems, one of the more popular groups of fault tolerance mechanisms is the mechanisms based on passive replication. In passive replication, we have only one active object, and all the replicas are passive. This limits the resource requirements, as only one object in the replicated group runs at one time, and it is also easier to implement than many other replication mechanisms.

When the primary object fails in a passive replication system, one of the secondary objects must be brought up to date, and it must then rerun the method. A problem with using this kind of replication strategy in a real-time system is that it may be difficult to analyze the time used when a fault occurs.

It may therefore be hard to see if this replication strategy is suitable for the real-time system, or more generally, to which degree one can fulfill real-time and reliability requirements.

While both fault tolerance and real-time are quite mature fields, and there are other works that describe the performance of fault tolerant methods, like [10] and [51], we have not seen any other attempts to derive a general expression for the temporal behaviour of methods in passive replicated systems, which is the focus of our paper.

The rest of the paper is organized as follows:

- Section B.2 describes the model of a passive replication system that we have used as a base for deriving the mathematical model.
- Section B.3 describes the basic mathematical model, where the time used for fault detection and fault correction is ignored.
- Section B.4 describes how the mathematical model can be extended by adding the times used for fault detection and fault correction.
- In section B.5 we show an example on how the model can be used for a simple system.
- Section B.6 contains discussions and conclusion to the paper.

B.2 The passive replication method

In this section, we will describe the replication method model we use as a base for the mathematical formulae. This model is based on the cold passive and warm passive replication methods described in Fault Tolerant CORBA [34].

In a passive replication system, there will be one active primary object and several passive secondary objects. The state of the primary object is logged, and if the primary object fails, the state of one of the secondary objects is updated from the log. This object will then be the new primary, and it will rerun the method that the failed object was running when it failed.

In our model, fault detection is based on some kind of "heartbeat" system; the fault detector will either regularly ask the primary object if it is alive or it will listen to "heartbeats" created by the primary. If the fault detector doesn't get a response in what is considered a reasonable time, it will assume that the primary object has failed, and the fault correction system will begin to work.

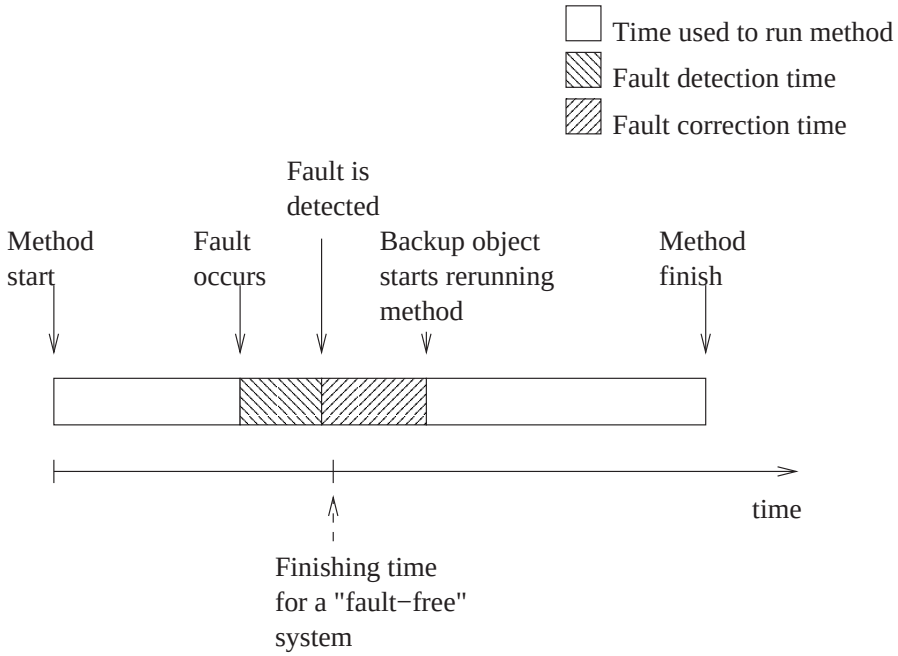


Figure B.1: Time used in case of a fault in a passive replication system

When a fault is detected, the fault correction system will prepare one of the secondary objects by updating its state from the log. When this object is up to date, it will be the new primary object, and it will start rerunning the method that the failed object was running.

The time use for one such fault occurence is illustrated in figure B.1.

B.3 A simple time model of a passively replicated fault tolerant method

This section describes how to model the time use of a fault tolerant method, using a passive replication strategy. For simplicity, we will assume that the time used for fault detection and fault correction are negligible. We will in later sections show how we can include these times to make a more complete model.

The method used to derive the time distribution is similar to the method used by Kleinrock [21] to derive the expression for the duration of a busy period. The idea is to derive the time distribution's probability generating function, defined by

$$\mathcal{F}(s) \equiv \mathcal{L}(\frac{dF(t)}{dt}) = \int_0^\infty e^{-st} dF(t) \tag{B.1}$$

where $\mathcal{L}(\cdot)$ is the laplace transform, by first assuming a given running time and a given number of faults, and then removing these assumptions.

In this paper, we will use function names with uppercase letters to describe distribution functions (example: $M(t)$), lowercase letters to describe probability density functions (example: $m(t)$), and calligraphy letters to describe probability generating functions (example: $\mathcal{M}(s)$).

B.3.1 The original fault-free system

We want to look at a method, which during normal operation has a running time described by the distribution function $M(t)$.

$$M(t) = P(Y \leq t | \phi = 0) \tag{B.2}$$

where $P(\cdot)$ is the probability function, Y is the finishing time of the method, and ϕ is the number of faults.

This function will have the corresponding probability density function (pdf) $m(t)$

$$m(t) = \frac{dM(t)}{dt} \tag{B.3}$$

and the probability generating function (pgf) $\mathcal{M}(s)$

$$\mathcal{M}(s) = \int_0^\infty e^{-st} dM(t) \tag{B.4}$$

B.3.2 The fault model

We model the faults as independent events. Many replication strategies have limits to the kind of faults they protect against, and independent faults are often a good enough model for the kind of faults the replication system is designed to tolerate. Using independent faults in the model will also make the model much simpler than using other fault models.

The faults are generated by a poisson process with intensity λ. The probability of at least one fault happening during a time interval $[0, \tau]$ is

$$p_\tau = 1 - p0_\tau = 1 - e^{-\lambda\tau} \tag{B.5}$$

where $p0_\tau = e^{-\lambda\tau}$ is the probability of no faults occuring in the interval.

Given that a fault occurs during the time interval $[0, \tau]$, the time used until the fault occurs is given by the distribution

$$H_\tau(t) = \begin{cases} \frac{1}{1-e^{-\lambda\tau}}(1 - e^{-\lambda t}) & , 0 \leq t \leq \tau \\ 1 & , t > \tau \end{cases} \tag{B.6}$$

This gives us the pdf

$$h_\tau(t) = \begin{cases} \frac{\lambda e^{-\lambda t}}{1-e^{-\lambda\tau}} & , 0 \leq t \leq \tau \\ 0 & , t > \tau \end{cases} \tag{B.7}$$

and the pgf

$$\mathcal{H}_\tau(s) = \frac{1 - e^{-\tau(\lambda+s)}}{1 - e^{-\lambda\tau}} \frac{\lambda}{\lambda + s} \tag{B.8}$$

B.3.3 The model of a system where faults may occur

We now want to make the model of the running time of a system where faults may occur.

When there is a possibility of faults occuring, the total running time of the system will be

$$Y = r + X_1 + X_2 + \cdots + X_\phi \tag{B.9}$$

where r is the running time for the fault free run, X_i is the time until abortion of a faulty run, and ϕ is the number of faulty runs.

We wish to find the distribution of the total time use,

$$G(t) = P(Y \leq t) \tag{B.10}$$

expressed by this functions probability generating function

$$\begin{aligned} \mathcal{G}(s) &= \int_0^\infty e^{-st} dG(t) = E[e^{-sY}] \\ &= E[e^{-s(r+X_1+X_2+\cdots+X_\phi)}] \end{aligned} \tag{B.11}$$

where $E[\cdot]$ is the expectation function.

We now introduce conditions on the running time and the number of faults.

$$\begin{aligned} E[e^{-sY}|r=x,\phi=k] &= E[e^{-s(x+X_1+\cdots+X_k)}] \\ &= E[e^{-sx}e^{-sX_1}\cdots e^{-sX_k}] \end{aligned} \tag{B.12}$$

Because of independency and since $E[e^{-sX_i}] = \mathcal{H}_x(s)$, we can write this as

$$\begin{aligned} E[e^{-sY}|r=x,\phi=k] &= E[e^{-sx}]E[e^{-sX_k}]\cdots E[e^{-sX_1}] \\ &= E[e^{-sx}](\mathcal{H}_x(s))^k \end{aligned} \tag{B.13}$$

We now want to remove the condition on the number of faults. This is done by taking the sum of the products of the expectation function (equation B.13) and probability of k faults, for each k.

$$E[e^{-sY}|r=x] = \sum_{k=0}^{\infty} E[e^{-sY}|r=x,\phi=k]P[\phi=k] \tag{B.14}$$

The probability function for the number of faults is given by

$$P[\phi=k] = e^{-\lambda x}(1-e^{-\lambda x})^k \tag{B.15}$$

This gives us the function

$$\begin{aligned} E[e^{-sY}|r=x] &= \sum_{k=0}^{\infty} e^{-xs}(\mathcal{H}_x(s))^k e^{-\lambda x}(1-e^{-\lambda x})^k \\ &= e^{-x(\lambda+s)}\sum_{k=0}^{\infty}((1-e^{-x(\lambda+s)})\frac{\lambda}{\lambda+s})^k \end{aligned} \tag{B.16}$$

To get to a final result, we will have to set an upper bound for the number of faults. As the probability of having a high number of faults is very low, and the fault tolerance mechanisms have a limited number of passive objects they can use anyway, this will be a reasonable approximation. To do this, we will need to modify the probability function for the number of faults, since we will not allow any more faults to happen if we have reached the upper bound.

$$\begin{aligned} &P[\phi=k|\phi\le k_{max}] \\ &= \begin{cases} e^{-\lambda x}(1-e^{-\lambda x})^k & ,k \le k_{max}-1 \\ (1-e^{-\lambda x})^{k_{max}} & ,k=k_{max} \end{cases} \end{aligned} \tag{B.17}$$

The new equation will then be

$$\begin{aligned} \mathcal{T}_{k_{max}}(s) &= E[e^{-sY}|r=x,\phi\le k_{max}] \\ &= e^{-x(\lambda+s)}\left(\sum_{k=0}^{k_{max}-1}((1-e^{-x(\lambda+s)})\frac{\lambda}{\lambda+s})^k\right) \\ &\quad + e^{-sx}((1-e^{-x(\lambda+s)})\frac{\lambda}{\lambda+s})^{k_{max}} \end{aligned} \tag{B.18}$$

The final step is to remove the condition on the time use for running the method.

$$\mathcal{G}(s) = \int_0^\infty \mathcal{T}(s) dM(x) \tag{B.19}$$

By using the definition of the probability generating function (equation B.1), we see that it will be possible to describe $\mathcal{G}$ in terms of $\mathcal{M}$.

For a maximum number of faults k_{max}, we get the expression

$$\begin{aligned}
&\mathcal{G}(s) \\
&= \left(\sum_{k=0}^{k_{max}-1} \frac{\lambda}{\lambda+s}^k \right. \\
&\quad \left. \left(\sum_{i=0}^{k} (-1)^i \binom{k}{i} \mathcal{M}((i+1)\lambda + (i+1)s) \right) \right) \\
&\quad + \frac{\lambda}{\lambda+s}^{k_{max}} \\
&\quad \left(\sum_{i=0}^{k_{max}} (-1)^i \binom{k_{max}}{i} \mathcal{M}(i\lambda + (i+1)s) \right)
\end{aligned} \tag{B.20}$$

Without assuming anything about the distribution of $M(t)$, this expression can not be transformed back to time domain.

B.4 Extending the model

In this section, we will show how the model can be extended, by incorporating times for fault correction and fault detection.

B.4.1 Adding time for fault correction

We assume the time used for fault correction can be described by the distribution function $C(t)$, with the corresponding pdf and pgf $c(t)$ and $\mathcal{C}(s)$.

We assume that a fault may happen during the fault correction process. As there is no correction during the initial run of the method, we have to distinguish between the initial run and the following runs. We now have to modify the expectation function (equation B.13) to

$$\begin{aligned}
&E[e^{-sY} | r = x, c = \epsilon, \phi = k] \\
&= \begin{cases} e^{-sx} & , k = 0 \\ e^{-s(x+\epsilon)} \mathcal{H}_x(s) \mathcal{H}_{x+\epsilon}(s)^{k-1} & , k > 0 \end{cases}
\end{aligned} \tag{B.21}$$

and the probability function for the number of faults (equation B.15) to

$$P[\phi = k] = \begin{cases} e^{-\lambda x} & , k = 0 \\ e^{-\lambda(x+\epsilon)}(1 - e^{-\lambda x})(1 - e^{-\lambda(x-\epsilon)})^{k-1} & , k \geq 1 \end{cases} \tag{B.22}$$

where c is the time used for fault correction.

When the condition on the number of faults is removed, we get the equation

$$\begin{aligned} & E[e^{-sY} | r = x, c = \epsilon] \\ = \; & e^{-xs} e^{-x\lambda} \\ & + \sum_{k=1}^{\infty} e^{-(x+\epsilon)s} \mathcal{H}_x(s) (\mathcal{H}_{x+\epsilon}(s))^{k-1} e^{-\lambda(x+\epsilon)} \\ & (1 - e^{-\lambda x})(1 - e^{-\lambda(x+\epsilon)})^{k-1} \\ = \; & e^{-x(\lambda+s)} \\ & + e^{-(x+\epsilon)(s+\lambda)}(1 - e^{-x(\lambda+s)}) \\ & \sum_{k=1}^{\infty} (1 - e^{-(x+\epsilon)(\lambda+s)})^{k-1} \frac{\lambda}{\lambda+s}^{k} \end{aligned} \tag{B.23}$$

As before, we need to put an upper bound to the number of faults. We modify the probability function for the number of faults

$$P[\phi = k] = \begin{cases} e^{-\lambda x} & , k{=}0 \\ e^{-\lambda(x+\epsilon)}(1 - e^{-\lambda x})(1 - e^{-\lambda(x-\epsilon)})^{k-1} & , 1{\leq}k{<}k_{max} \\ (1 - e^{-\lambda x})(1 - e^{-\lambda(x-\epsilon)})^{k_{max}-1} & , k{=}k_{max} \end{cases} \tag{B.24}$$

and get the new function

$$\begin{aligned} & \mathcal{T}_{k_{max}}(s) \\ = \; & e^{-x(\lambda+s)} \\ & + e^{-(x+\epsilon)(s+\lambda)}(1 - e^{-x(\lambda+s)}) \\ & \left(\sum_{k=1}^{k_{max}-1} (1 - e^{-(x+\epsilon)(\lambda+s)})^{k-1} \left(\frac{\lambda}{\lambda+s} \right)^{k} \right) \\ & + e^{-s(x+\epsilon)}(1 - e^{x(\lambda+s)}) \\ & (1 - e^{-(x+\epsilon)(s+\lambda)})^{k_{max}-1} \left(\frac{\lambda}{\lambda+s} \right)^{k_{max}} \end{aligned} \tag{B.25}$$

We can now get the equation for $\mathcal{G}(s)$.

$$\mathcal{G}(s) = \int_0^{\infty} \int_0^{\infty} \mathcal{T}(s) dM(x) dC(\epsilon) \tag{B.26}$$

By using the definition of the pgf, we see that $\mathcal{G}(s)$ can be expressed by products of $\mathcal{M}(\cdot)$ and $\mathcal{C}(\cdot)$.

$$\begin{aligned}
&\mathcal{G}(s)\\
&= \mathcal{M}(\lambda+s)\\
&\quad + \left(\sum_{k=1}^{k_{max}-1}\left(\frac{\lambda}{\lambda+s}\right)^k\right.\\
&\quad \left(\sum_{i=0}^{k-1}(-1)^i\binom{k}{i}(\mathcal{M}((i+1)\lambda+(i+1)s)\right.\\
&\quad - \mathcal{M}((i+2)\lambda+(i+2)s))\\
&\quad \left.\left.\mathcal{C}((i+1)\lambda+(i+1)s)\right)\right)\\
&\quad + \left(\frac{\lambda}{\lambda+s}\right)^{k_{max}}\left(\sum_{i=0}^{k_{max}-1}(-1)^i\binom{k_{max}-1}{i}\right.\\
&\quad (\mathcal{M}(i\lambda+(i+1)s)\\
&\quad \left.- \mathcal{M}((i+1)\lambda+(i+2)s))\mathcal{C}(i\lambda+(i+1)s)\right)
\end{aligned} \tag{B.27}$$

B.4.2 Adding the fault detection delay

We assume the time used from a fault occurs to the fault is detected fault described by the probability distribution $I(t)$, the density function $i(t)$ and the generating function $\mathcal{I}(s)$.

As the running object has already failed, and no new objects are running, we assume that new faults happening during fault detection time can be ignored.

While we still have the same fault number probability as in equation B.22, we have to modify equation B.21 to

$$\begin{aligned}
&E[e^{-sY}|r=x, c=\epsilon, i=\alpha, \phi=k]\\
&= \begin{cases} e^{-sx} & ,k=0\\ e^{-s(x+\epsilon)}\mathcal{H}_x(s)\mathcal{H}_{x+\epsilon}(s)^{k-1}e^{-s\alpha k} & ,k>0 \end{cases}
\end{aligned} \tag{B.28}$$

where i is the time used for fault detection.

By removing the condition on the number of faults we get the equation

$$\begin{aligned} & E[e^{-sY}|r=x, c=\epsilon, i=\alpha] \\ = \;& e^{-xs}e^{-x\lambda} \\ & + \sum_{k=1}^{\infty} e^{-(x+\epsilon)s}\mathcal{H}_x(s)(\mathcal{H}_{x+\epsilon}(s))^{k-1} \\ & e^{-s\alpha k}e^{-\lambda(x+\epsilon)}(1-e^{-\lambda x})(1-e^{-\lambda(x+\epsilon)})^{k-1} \\ = \;& e^{-x(\lambda+s)} \\ & + e^{-(x+\epsilon)(s+\lambda)}(1-e^{-x(\lambda+s)}) \\ & \sum_{k=1}^{\infty}(1-e^{-(x+\epsilon)(\lambda+s)})^{k-1}\left(\frac{\lambda}{\lambda+s}\right)^{k} e^{-s\alpha k} \end{aligned} \tag{B.29}$$

Putting an upper bound to the number of allowed faults gives us the equation

$$\begin{aligned} & \mathcal{T}_{k_{max}}(s) \\ = \;& e^{-x(\lambda+s)} \\ & + e^{-(x+\epsilon)(s+\lambda)}(1-e^{-x(\lambda+s)}) \\ & \left(\sum_{k=1}^{k_{max}-1}(1-e^{-(x+\epsilon)(\lambda+s)})^{k-1}\left(\frac{\lambda}{\lambda+s}\right)^{k} e^{-s\alpha k}\right) \\ & + e^{-s(x+\epsilon)}(1-e^{x(\lambda+s)}) \\ & (1-e^{-(x+\epsilon)(s+\lambda)})^{k_{max}-1}\left(\frac{\lambda}{\lambda+s}\right)^{k_{max}} e^{-s\alpha k_{max}} \end{aligned} \tag{B.30}$$

Finally, we can now get the equation for $\mathcal{G}(s)$ by removing the conditions of time use.

$$\mathcal{G}(s) = \int_0^{\infty}\int_0^{\infty}\int_0^{\infty} \mathcal{T}(s)dM(x)dC(\epsilon)dI(\alpha) \tag{B.31}$$

By using the definition of the pgf, we see that $\mathcal{G}(s)$ can be expressed by

products of $\mathcal{M}(\cdot)$, $\mathcal{C}(\cdot)$ and $\mathcal{I}(\cdot)$.

$$\begin{aligned}
&\mathcal{G}(s)\\
&= \mathcal{M}(\lambda+s)\\
&\quad + \left(\sum_{k=1}^{k_{max}-1}\left(\frac{\lambda}{\lambda+s}\mathcal{I}(s)\right)^k\right.\\
&\quad \left(\sum_{i=0}^{k-1}(-1)^i\binom{k}{i}(\mathcal{M}((i+1)\lambda+(i+1)s)\right.\\
&\quad -\mathcal{M}((i+2)\lambda+(i+2)s))\\
&\quad \left.\left.\mathcal{C}((i+1)\lambda+(i+1)s)\right)\right)\\
&\quad + \left(\frac{\lambda}{\lambda+s}\mathcal{I}(s)\right)^{k_{max}}\\
&\quad \left(\sum_{i=0}^{k_{max}-1}(-1)^i\binom{k_{max}-1}{i}(\mathcal{M}(i\lambda+(i+1)s)\right.\\
&\quad \left.-\mathcal{M}((i+1)\lambda+(i+2)s))\mathcal{C}(i\lambda+(i+1)s)\right)
\end{aligned} \tag{B.32}$$

This is our main result, which, as we will demonstrate, is a versatile tool.

B.5 Example

We will now show how this equation can be used for a simple system.

We have an object with a method that in normal operation has a deterministic running time τ_r. This object is replicated in a passive replication system with 2 backup objects. The fault detection and fault correction times are uniformly distributed in the intervals $[0,\tau_d]$ and $[0,\tau_c]$. The fault rate is λ

This gives us the system functions

$$M(t) = H(t-\tau_r) \tag{B.33}$$

$$\mathcal{M}(s) = e^{-\tau_r s} \tag{B.34}$$

$$C(t) = \begin{cases} t/\tau_c & ,0\le t\le\tau_c \\ 1 & ,t>\tau_c \end{cases} \tag{B.35}$$

$$\mathcal{C}(s) = \frac{1-e^{-\tau_c s}}{\tau_c s} \tag{B.36}$$

$$I(t) = \begin{cases} t/\tau_d & ,0\le t\le\tau_d \\ 1 & ,t>\tau_d \end{cases} \tag{B.37}$$

$$\mathcal{I}(s) = \frac{1 - e^{-\tau_d s}}{\tau_d s} \tag{B.38}$$

where $H(t)$ is the unit step function.

We will use equation B.32 with $k_{max} = 2$.

$$\begin{aligned} &\mathcal{G}(s) \\ &= \mathcal{M}(\lambda + s) \\ &\quad + \tfrac{\lambda}{\lambda+s}\mathcal{I}(s)((\mathcal{M}(\lambda + s) - \mathcal{M}(2\lambda + 2s))\mathcal{C}(\lambda + s)) \\ &\quad + (\tfrac{\lambda}{\lambda+s}\mathcal{I}(s))^2((\mathcal{M}(s) - \mathcal{M}(\lambda + 2s))\mathcal{C}(s) \\ &\quad - (\mathcal{M}(\lambda + 2s) - \mathcal{M}(2\lambda + 3s))\mathcal{C}(\lambda + 2s)) \end{aligned} \tag{B.39}$$

Because finding the inverse laplace transform may be difficult for many systems, we used Matlab and Simulink to numerically transform the expression to the time domain. The results for the system with parameters $\tau_r = 1$, $\tau_d = 0.5$, $\tau_c = 0.1$ and $\lambda = 0.01$ is shown in figure B.2.

The model can be used for various purposes, like finding the probability of missing the deadline. With the parameters above, $G(2) = 0.997$. This means that if the deadline is at $t = 2$, there will be a 0.3% chance that this system will miss the deadline each time the method is run.

The model can also be used to determine how sensitive the system is to changes in the parameters. Figure B.3 shows how the distribution changes when we change the times used for fault detection. The figure shows curves for $\tau_d = 0.5$, $\tau_d = 0.4$, $\tau_d = 0.3$, $\tau_d = 0.2$ and $\tau_d = 0.1$. For a deadline at $t = 2$, the chance of a deadline miss varies from 0.3% at $\tau_d = 0.5$ to 0.1% at $\tau_d = 0.1$.

B.6 Discussions and Conclusion

In our model, we are assuming a fault model where we will have independent faults. While this model is, in many cases, "good enough", and simple replication systems often cannot protect against many fault types that require other fault models, there may be cases where the fault model should be changed.

The model is also limited to the kinds of fault detection methods that it can represent. The model as it is described here can only be used if there is no dependency between the time used for fault detection and the time when a fault occurs. This makes the unmodified version of the model unsuitable for systems where fault detection is based on checking the return values, or where fault detection is based on timeout of the method return. It should, however, be possible to modify the model to suit these and other fault detection methods.

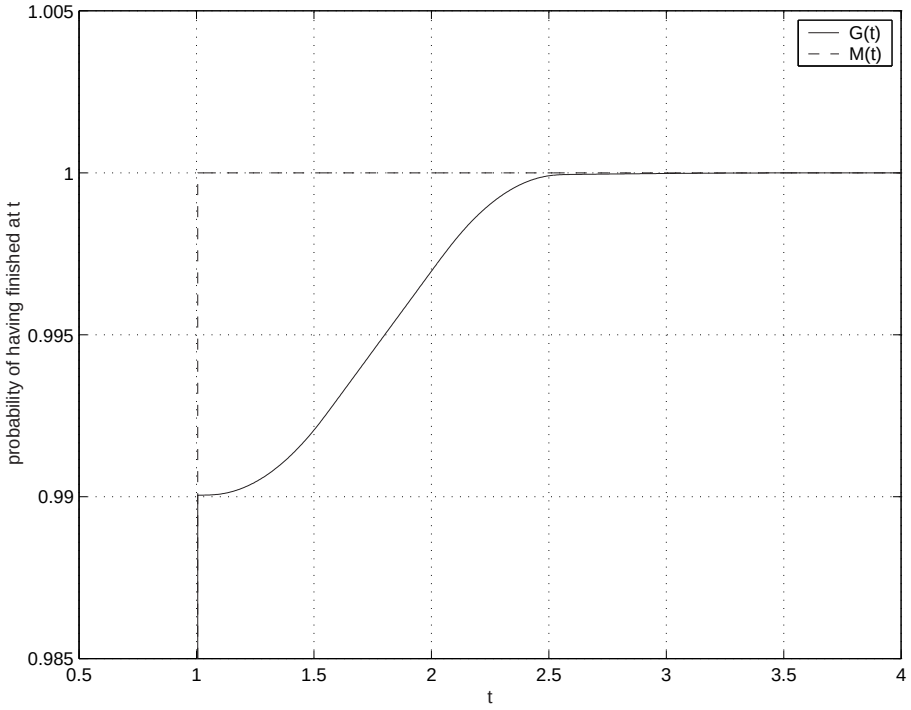


Figure B.2: $M(t)$ and $G(t)$ for the example system, details

We have derived an expression for the run-time distribution for a method in a fault tolerant system where passive replication is used. This expression, which is on the probability generating function form, is a function of the distributions of the running time in a fault-free system, the fault detection time, and the time used for fault correction.

We have also shown how this can be used for a simple system, and how we can use the results to determine the probability of a deadline fault and the sensitivity to parameter changes.

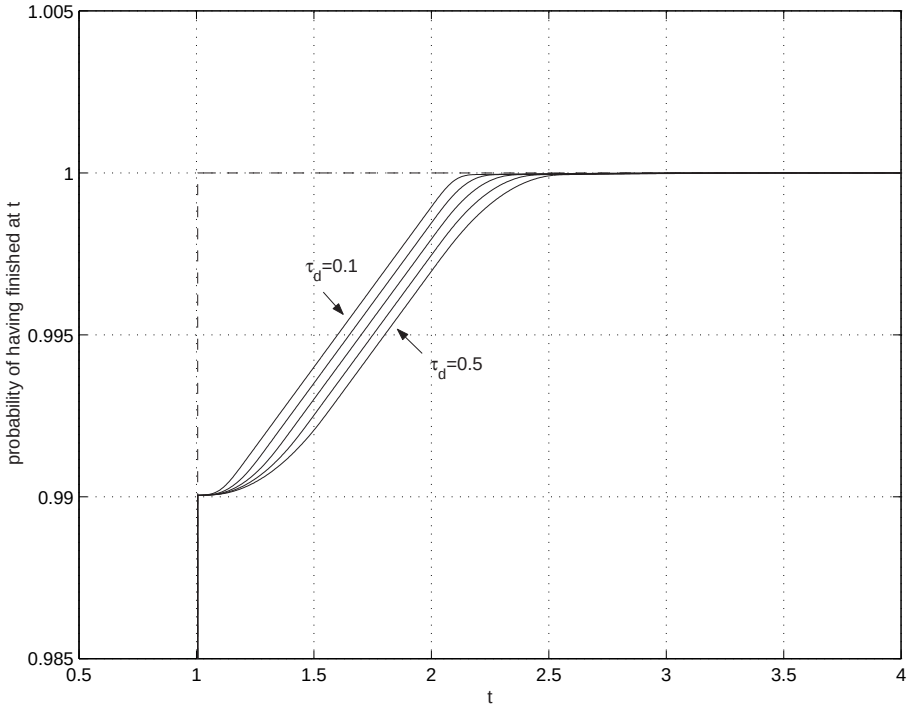


Figure B.3: The system's sensitivity to changes in fault detection time

Appendix C

Assessing Reliability of Real-Time Distributed Systems

By Åsmund Tjora and Amund Skavhaug

This paper was presented at the 1st ERCIM Workshop on Software-Intensive Dependable Embedded Systems in Porto, Portugal, August 2005 [47].

Abstract

Serial replication structures are used in fault tolerance mechanisms in systems. Analysing the timing behaviour of systems using these mechanisms can be difficult, however, it is necessary to do this kind of analysis if the systems are to be used in real-time applications.

In this paper, two ways of analysing the timing behaviour in such systems are studied, a simulator and an analytical model. Some of the strengths and weaknesses of these methods are discussed, and it is demonstrated with an example how they can be used.

C.1 Introduction

Many of today's real-time applications will, in addition to the timing requirements, also have reliability requirements. It is therefore important to study the combination of real-time and fault tolerant systems. A fault tolerance mechanism may be able to detect and correct a number of faults, however, in a real-time system, it is possible that the same mechanism introduces new faults if the time it uses for detection and correction causes the system to miss a deadline. Because of this, a fault tolerant mechanism may give very little improvement to the reliability of a system, depending on the system's characteristics.

For many real-time systems, it has been common to use a parallel fault tolerance structure, that is, several replicas of one object are run simultaneously. This will give a deterministic temporal behaviour even if a fault is detected in one of the replicas, and comparision of the results from the different replicas can be used as an extra fault detection mechanism. On the negative side, parallel fault tolerance mechanisms needs extra resources.

In a system using a serial fault tolerance structure, only one instance of an object is active at one time, while the other replicas are passive. If a fault is detected, the task running on the object is stopped, and one of the passive replicas is made active. The task is then rerun on this new active object. The mechanisms based on serial structures use fewer resources than those based on parallel structures, making them quite popular in general purpose systems. In real-time systems, the time used for detecting a fault, updating the states of the replica, and rerunning the task may cause deadline misses. When considering a fault tolerance mechanism based on a serial structure for a real-time system, it is therefore important to analyse the time used by such a system.

In this paper, two main methods of analysing the timing behaviour of a

fault tolerant system are presented: Analytical models of the behaviour, and simulation.

Analytical models give precise results, and once a model is developed for a system, the same model can often be used with little or no modification in similar systems. However, the mathematical expressions can become quite complicated, even for a simple system, and such models are therefore not always easy to understand or use.

Simulation may give good results, and is often easier to understand and use than analytical models. A problem with simulations is that they have to be run a large number of times if events with a low probability of occuring are to be studied.

C.2 The model of the fault-tolerant system

The system that we are modelling is a simple Client-Server system where the server is using a fault tolerant mechanism based on a serial structure. The model we are using for the fault tolerance mechanism is based on the cold and warm passive replication methods described in Fault Tolerant CORBA [34].

In the system, there is one active primary object and several passive secondary objects. The state of the primary object is logged, and if the primary object fails, the state of one of the secondary objects is updated from the log. This object will now be the new primary, and the task that was running on the failed object will be rerun.

In the system, fault detection is based on some kind of watchdog mechanism. A fault detector will either ask the primary object if it is "alive", or the primary object will generate "heartbeats" that the fault detector will listen to. In both cases, if the fault detector doesn't get a response in what is considered a reasonable time, it is assumed that the primary object has failed, and the fault correction mechanisms will start to work.

When a fault is detected, the fault correction mechanism will prepare one of the secondary objects by updating its state from the log. When the object is up to date, the task that was running will be rerun on it. The timing behaviour of the system in a case where a fault occurs is shown in figure C.1.

We also have to consider cases where several faults occurs during the service of one task.

If a new fault occurs after the correction of the previous fault has started, the whole detection-correction-rerun cycle has to be restarted. However, if the fault occurs before the previous fault is detected, it will have no extra effect on

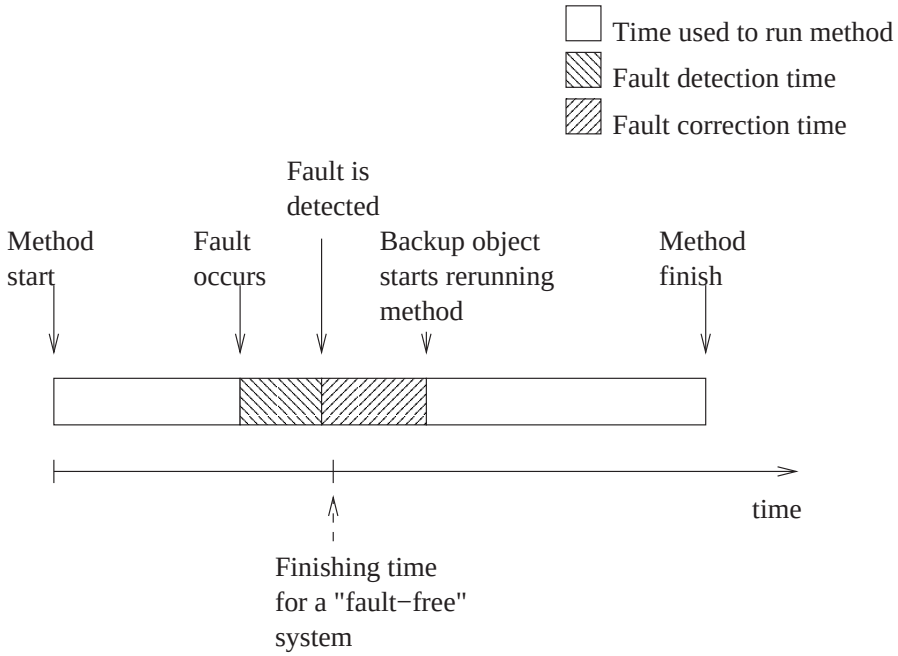


Figure C.1: Timing behaviour of system during a fault

the system, since it is still the already faulty object that is active.

C.3 The Analytical Model

The analytical model we use is the one presented in [46]. The expressions are derived using a method similar to the one used to find the distribution of the busy period in queuing systems, as presented by Kleinrock in [21].

In the expressions, we use the moment generating functions (mgf) of the different distributions. A distribution's moment generating function is the laplace transform of its probability density function (pgf), i.e. $\mathcal{F}(s) = \int_0^\infty e^{-st} f(t)dt$.

In this paper, we use calligraphic letters (i.e. $\mathcal{F}(s)$) to represent a distribution's moment generating function and normal lowercase letters (i.e. $f(t)$) to represent the distribution's probability density function.

The model gives us the moment generating function, $\mathcal{G}(s)$, of the run-time

distribution for a fault-tolerant system based on a serial structure as a function of the moment generating functions of the fault-free run-time, the fault detection and correction times and the fault intensity. Mathematical tools may then be used to transform the mgf back to the time domain.

We assume that the time used to run a fault-free method is given by the pdf $m(t)$, the time used to detect faults is given by the pdf $i(t)$, and the time used to correct a fault (i.e. updating the state of a passive object and restarting method on this object) is given by the pdf $c(t)$, and that these distributions have the moment generating functions $\mathcal{M}(s)$, $\mathcal{I}(s)$, and $\mathcal{C}(s)$. We also assume that the faults are independent and that fault arrivals can be modelled by a poisson process with intensity λ.

Given these parameters, the mgf of the run-time distribution in a system where faults may occur is given by

$$\begin{aligned}
&\mathcal{G}(s) \\
&= \mathcal{M}(\lambda+s) + \left(\sum_{k=1}^{\infty} \left(\frac{\lambda}{\lambda+s} \right)^k \mathcal{I}(ks) \right. \\
&\quad \left(\sum_{i=0}^{k-1} (-1)^i \binom{k-1}{i} (\mathcal{M}((i+1)(\lambda+s)) \right. \\
&\quad \left. \left. -\mathcal{M}((i+2)(\lambda+s)))\mathcal{C}((i+1)(\lambda+s)) \right) \right)
\end{aligned} \tag{C.1}$$

There is an infinite sum in the expression. This is explained by the fact that there always will be a possibility that the system will fail again while a task that has failed before is still running. This also makes it possible that an infinite number of faults may occur during the running of the same task. The infinite sum can be approximated to a finite sum in two ways:

- Assuming that only a finite number N faults will happen during the run-time of one task, so that no new faults will occur after this.

- Assuming that the task will fail if the number of faults occuring during the running of one task exceeds N.

The mgf for the first approximation is given by

$$\begin{aligned}
&\mathcal{G}_1(s) \\
&= \mathcal{M}(\lambda+s) + \Bigg(\sum_{k=1}^{N-1} \left(\frac{\lambda}{\lambda+s}\right)^k \mathcal{I}(ks) \\
&\quad \Bigg(\sum_{i=0}^{k-1} (-1)^i \binom{k-1}{i} (\mathcal{M}((i+1)(\lambda+s)) \\
&\quad -\mathcal{M}((i+2)(\lambda+s)))\mathcal{C}((i+1)(\lambda+s)) \Bigg) \Bigg) \\
&\quad + \left(\frac{\lambda}{\lambda+s}\right)^N \mathcal{I}(Ns) \Bigg(\sum_{i=0}^{N-1} (-1)^i \binom{N-1}{i} \\
&\quad (\mathcal{M}(i\lambda+(i+1)s) - \mathcal{M}((i+1)\lambda+(i+2)s)) \\
&\quad \mathcal{C}(i\lambda+(i+1)s) \Bigg)
\end{aligned} \tag{C.2}$$

while the mgf for the second is given by

$$\begin{aligned}
&\mathcal{G}_2(s) \\
&= \mathcal{M}(\lambda+s) + \Bigg(\sum_{k=1}^{N} \left(\frac{\lambda}{\lambda+s}\right)^k \mathcal{I}(ks) \\
&\quad \Bigg(\sum_{i=0}^{k-1} (-1)^i \binom{k-1}{i} (\mathcal{M}((i+1)(\lambda+s)) \\
&\quad -\mathcal{M}((i+2)(\lambda+s)))\mathcal{C}((i+1)(\lambda+s)) \Bigg) \Bigg)
\end{aligned} \tag{C.3}$$

If there is a low probability that a fault occurs during the running of a task, the probability of several independent faults occuring during the same task becomes so small that these approximations are reasonable. Also, since the systems we are analysing have deadlines, and each fault will take some time to detect and correct, there can only be a finite number of faults before we are guaranteed to *miss* the deadline.

A difference between the two approximations is that the first approximation models a system that will never fail completely, while the second will always have a small probability that the running of a task is never finished.

C.4 The Simulator

The simulator is designed as a simple server-client system, as shown in figure C.2. For the system presented in this paper, the network part is omitted.

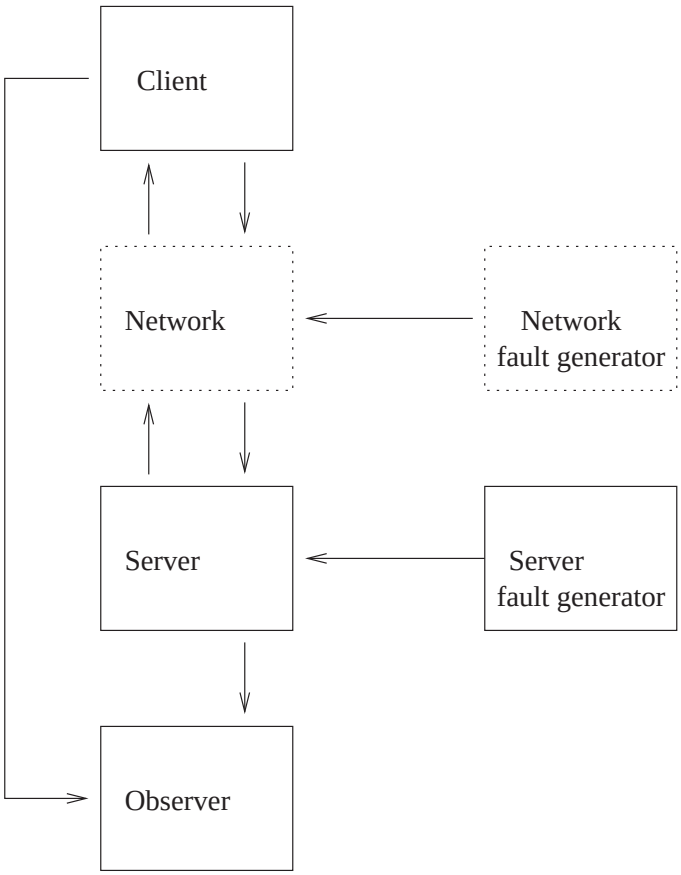


Figure C.2: Structure of a simple client-server simulator.

C.4.1 The Client

In the system presented here, the client part of the simulator functions mostly as a generator of an object type called *process*. At regular intervals, the client generates a new *process* object, which is time-stamped and sent to the server. When the network part of the model is used, it also receives *process* objects that are returned from the server, time-stamps them, and sends them to the observer.

C.4.2 The Fault Generator

The fault generator is a "pure" generator that creates *fault* objects at intervals drawn from a negative exponential distribution. These are sent directly to the server.

C.4.3 The Server

During normal operation, the server function is quite straightforward. When a *process* object arrives, it is placed in a queue. If the server is *idle* when this happens, or if it has just finished a service while the queue is not empty, it will take the next object in the queue. It will then set its status to *busy*, draw a service time from the service time distribution, and set the next event to happen when this time is up.

When a *fault* object arrives, the server will set its status to *error*, draw a time from the detection time distribution, and set the next event (i.e. the fault detection event) to happen when this time is up. When the fault is detected, the server sets is status to *correction* and draws the time to the next event from the correction time distribution. After the fault is corrected, the server returns to *busy*, and restarts service on the *process* object by setting the next event to happen after the service time.

A simplified state machine diagram for the server is shown in figure C.3

C.4.4 The Observer

The observer receives *process* objects that has finished running. The data from these objects are extracted and presented in a way that a program used for data analysis can read.

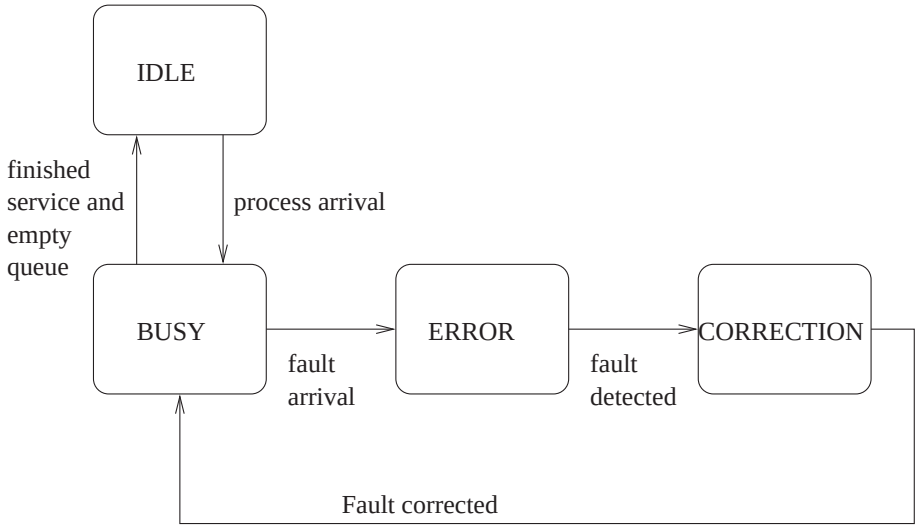


Figure C.3: Simplified state machine diagram for the server model.

C.4.5 Implementation

The simulator is implemented in C++, using the ADEVS discrete event simulator framework [1] as a base. For data analysis, Matlab is used.

C.5 Example

We will now look at the results from the mathematical model and the simulator for a simple system.

C.5.1 System parameters

For the fault free running time, we have chosen a triangular distribution with minimum time 6, maximum time 10 and the mode at 8.

$$m(t) = \begin{cases} 0 & ,0 \leq t < 6 \\ \frac{t-6}{4} & ,6 \leq t < 8 \\ \frac{10-t}{4} & ,8 \leq t < 10 \\ 0 & ,t \geq 10 \end{cases} \tag{C.4}$$

The fault detection time is uniformly distributed with a minimum time 1 and a maximum time 3.

$$i(t) = \begin{cases} 0 & ,0 \leq t < 1 \\ \frac{1}{2} & ,1 \leq t \leq 3 \\ 0 & ,t > 3 \end{cases} \tag{C.5}$$

The fault correction time is uniformly distributed with a minimum time 0 and a maximum time 5.

$$c(t) = \begin{cases} \frac{1}{5} & ,0 \leq t \leq 5 \\ 0 & ,t > 5 \end{cases} \tag{C.6}$$

These distributions have the following moment generating functions:

$$\mathcal{M}(s) = \frac{e^{-6s} - 2e^{-8s} + e^{-10s}}{4s^2} \tag{C.7}$$

$$\mathcal{I}(s) = \frac{e^{-s} - e^{-3s}}{2s} \tag{C.8}$$

$$\mathcal{C}(s) = \frac{1 - e^{-5s}}{5s} \tag{C.9}$$

The mean time between faults is set to 10000.

The maximum number of faults during the run of one method for the analytical model is set to 2. While this may seem like a very small number, the effect of modelling more faults in this system is very small, while it makes the expression much larger. Using equation C.3, the mgf for the run-time distribution for the system is found to be

$$\begin{aligned} &\mathcal{G}_1(s) = \\ &\mathcal{M}(\lambda + s) \\ &+(\tfrac{\lambda}{\lambda+s})\mathcal{I}(s) \\ &(\mathcal{M}(\lambda + s) - \mathcal{M}(2\lambda + 2s))\mathcal{C}(\lambda + s) \\ &+(\tfrac{\lambda}{\lambda+s})^2\mathcal{I}(2s) \\ &((\mathcal{M}(\lambda + s) - \mathcal{M}(2\lambda + 2s))\mathcal{C}(\lambda + s) \\ &-(\mathcal{M}(2\lambda + 2s) - \mathcal{M}(3\lambda + 3s))\mathcal{C}(2\lambda + 2s)) \end{aligned} \tag{C.10}$$

Since the inverse transform of the mgf will be quite complicated, Matlab and Simulink is used to numerically transform the results to the time domain.

The simulator is set to run 100000 tasks. Matlab is used to evaluate and present the results from the simulator.

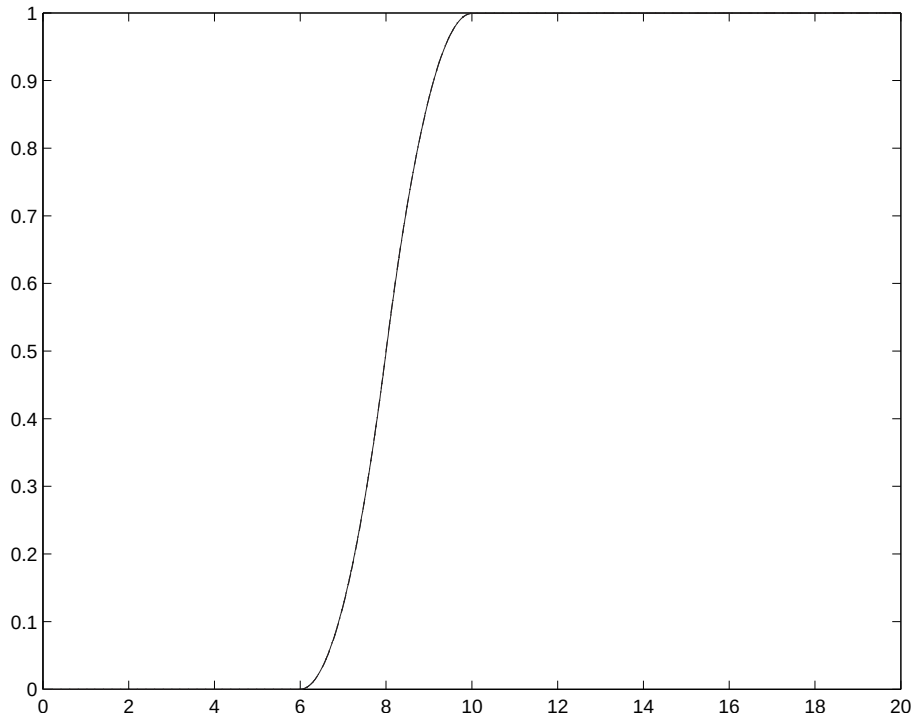


Figure C.4: Results from the simulation and the analytical model

C.5.2 Results

We plot the cumulative density functions (cdf) of the finishing times, using the results from the simulator and the analytical model together, as shown in figure C.4. If we zoom in on the plot as in figure C.5, we can se that the analytical results (solid line) and the simulator results (dashed line) follow each other quite closely, with an exception in the area around $t = 20$, where the simulator results show a somewhat lower probability of finishing at this time than the results from the analytical model. The curves also show the cdfs of a non-faulty system (top dotted line) and a non-fault-tolerant system (bottom dotted line).

The graphs clearly shows that the improvement caused by the fault tolerant mechanism is almost negligible if the deadline is close to 10, while the improve-

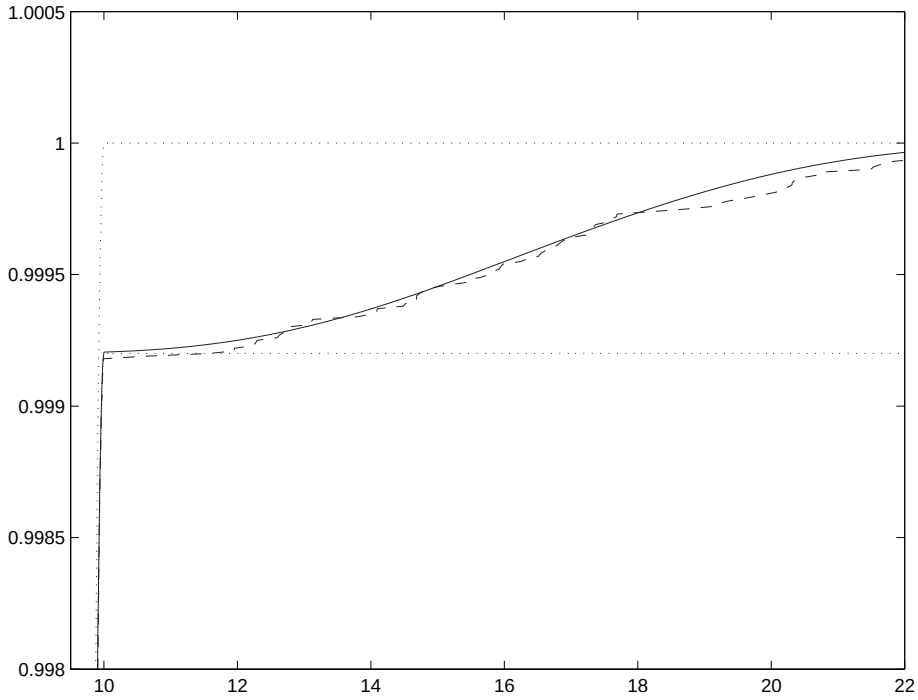


Figure C.5: Details of results from the simulation and the analytical model together with the fault-free and non-fault-tolerant distributions

ment is quite good if the deadline is around 20.

The results can be used to determine the probability that a task with a deadline will succeed or fail. If, for instance, we set the deadline to 20, the probability of a failure is $1.2 \cdot 10^{-4}$. Had there been no fault tolerance mechanism, this probability would have been $8.0 \cdot 10^{-4}$.

By changing parameters, we can see how the fault detection and correction times affect the system. As an example, we add 5 to the fault correction time, so it is uniformly distributed between 5 and 10:

$$c(t) = \begin{cases} 0 & , 0 \leq t < 5 \\ \frac{1}{5} & , 5 \leq t \leq 10 \\ 0 & , t > 10 \end{cases} \tag{C.11}$$

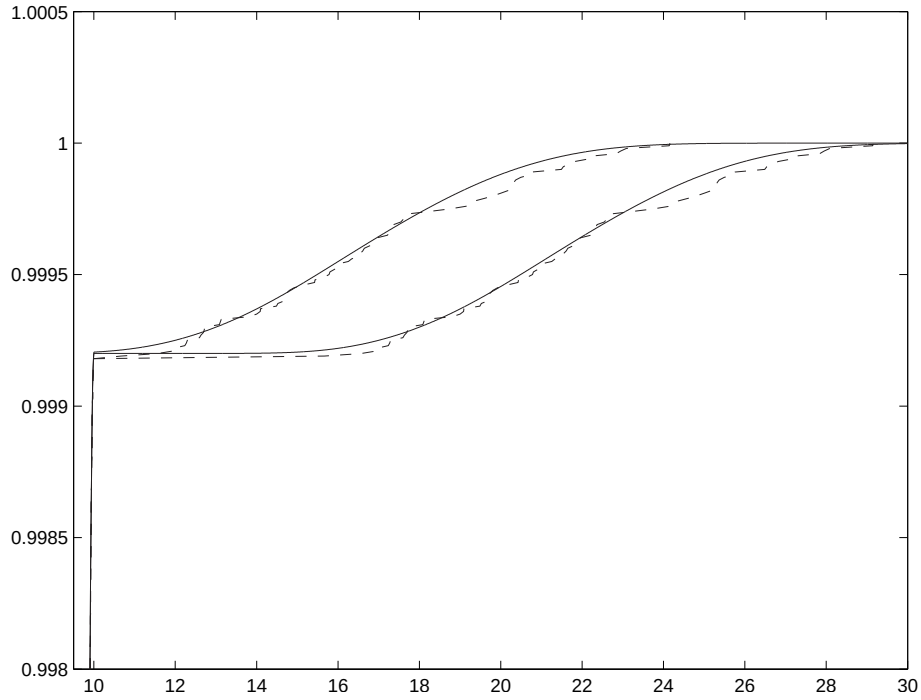


Figure C.6: Results showing the effect of adding a delay of 5 to the fault correction

$$\mathcal{C}(s) = \frac{e^{-5s} - e^{-10s}}{5s} \tag{C.12}$$

The results are plotted with the earlier results in figure C.6, showing that the delay in fault correction times causes the fault tolerant mechanism to be almost ineffective if the deadline is 20 or less.

C.6 Discussion and conclusion

In the models presented here, we assume that all faults are independent, and we also only model faults that can be detected and corrected by the simple fault tolerance mechanism described in section C.2. In many cases this is good enough for the analysis of a system.

If the behaviour of other fault classes, detection and correction mechanisms are known, expanding the simulator so it is able to handle these, is, in most cases, not very difficult. The difficulty of expanding the analytical model varies greatly, depending on how the faults affect the system and how the detection and correction mechanisms work.

In this paper, two different ways of creating models for the study of the timing behaviour of a fault tolerant real-time system, the simulator and the analytical model, and how the results from these can be used to find the failure probability of methods running on this kind of system are shown.

Some of the strengths and weaknesses of these models are discussed.

Both the simulator and the analytical models have the potential to be expanded, so they can be used to study a wider range of fault types, detection methods, correction methods and fault tolerant structures.

Appendix D

A Mathematical Model for Run-Time Distributions in a Fault Tolerant System with Nonhomogeneous Passive Replicas

By Åsmund Tjora, Amund Skavhaug, and Poul E. Heegaard

This paper was presented at the ERCIM/DECOS Workshop on Dependable Embedded Systems in Gdansk, Poland, September 2006 [49]. The proceedings of this workshop have not yet been published.

Appendix D. A Mathematical Model for Run-Time Distributions in a Fault Tolerant System with Nonhomogeneous Passive Replicas

Abstract

Structures based on passive replication are used as a fault tolerance mechanism in many systems, some times with variation in the replicas used. The analysis of timing behaviour in such systems is a difficult task, but this analysis is necessary if the systems shall be used in real-time applications.

In this paper, we consider services with requirements both to reliability and timeliness. The paper focuses on fault tolerant systems with passive replication at the component level, where a new replica is instantiated when the primary fails. The instantiation and take-over on failure will affect the service response time, and a mathematical model and expression is developed in order to quantify the effect of failures on the system response time distribution. The expression enables evaluation quantification of the service time guarantees given.

The use of the mathematical model is demonstrated on selected examples and its validity is compared to systems simulation of the same examples.

D.1 Introduction

Many real-time applications will, in addition to the timing requirements, also have reliability requirements. To fulfill these reliability requirements, some kind of fault tolerance mechanism might be neccessary. A fault tolerance mechanism may be efficient in detecting and correcting errors, however, if the extra time used for this causes deadline misses, the mechanism itself has introduced new faults. Because of this, some common fault tolerance mechanisms may give very little improvement to the system's reliability, depending on the system's real-time requirements. The effect of using fault tolerance mechanisms in real-time systems is therefore an important study, as noted in [43].

In many real-time fault tolerant systems, it has been common to use active replication structures, i.e., the task is run on several replicas simultaneously. This will give deterministic temporal behaviour even if errors are detected in one of the replicas, and comparision of results from the different runs can be used as a fault detection mechanism. On the negative side, these fault tolerance mechanisms requires extra hardware resources.

In fault tolerant systems systems using a passive replication structure, only one object running the task is active, while backup objects are passive. If an error is detected, the task is stopped, and one of the backup objects is prepared and made active. The task is then rerun on this object. Because only one of the replicas is running at a given time, these fault tolerance mechanisms use

less resources than the mechanisms based on active replication, making them popular, especially in general purpose systems. However, because of the extra time needed to detect an error, prepare a backup object and rerun the task, this kind of fault tolerance mechanisms are exactly the kind that may cause real-time applications to miss deadlines.

This does not mean that this type of fault tolerance mechanism is unsuitable for use in real-time applications; there is still a possibility that many of the faults can be tolerated within the time limits, thus improving the reliability of the system without the need for a more resource consuming fault tolerance mechanism. Analyzing the time use in these systems is therefore important in order to determine how much the reliability of the system can be improved while the tasks still meet their deadlines.

In many fault tolerant systems, the methods used to run the task are the same in all replicas. This makes the design of the system easier, and is often considered "good enough". In some system it is desirable to use different methods that perform the same task in the different replicas [2, 37]. Some faults may be a part of the program code itself, and will therefore appear in all replicas unless there is a difference between the replicas. Thus, varying the replicas may give protection against a wider range of faults.

In real-time systems, the timeliness of the results may be a higher priority than the precision of the results [28]. In some systems, using a fault tolerance system with backups that can provide results fast at the cost of the precision of the results, may be desirable, as it improves the chances of the task meeting its deadline. During normal operation, a method that gives results with high presision is used. When an error is detected, a faster, but less accurate method, is used.

In previous papers [46, 47], we have shown how timing in fault tolerant systems based on passive replication can be analyzed if the timing behaviour is the same in all replicas.

This paper, we focus on systems using passive replication with inhomogeneous replicas as a fault tolerance mechanism. We present a mathematical expression for the service times of tasks running in these kinds of systems, thus making it easier to decide if this kind of fault tolerance mechanism is applicable for the real-time system. Examples on using this expression are given, and the results are compared with simulations.

The paper is organized as follows:

- Section D.2 describes the class of passive replication fault tolerant systems that we have used as a base for the mathematical models.

- Section D.3 describes the mathematical model, and how it is derived.
- Section D.4 gives examples on how the mathematical model can be used as a method to analyse the timing behaviour in different systems, among them systems where the timing distributions are different between replicas and systems with imprecise backups.
- Secton D.5 contains discussions and conclusion of the paper.

D.2 The modelled system

The class of systems we are modeling can be described as a server that is using a fault tolerant mechanism based on passive replication. The model we are using was originally based on the cold and warm passive replication methods described in Fault Tolerant CORBA [34]. We have modified the model so that while the replicas used for backup perform the same services as the primary replica, they are not necessarily the same, so the service time distribution may vary between the replicas.

The different replicas as well as the components for managing the fault tolerance mechanism may be on the same node, or they may be distributed over several nodes, as shown in figure D.1. While the physical distribution of the system may affect the timing distributions in the model and the system's ability to tolerate certain faults, the mathematical model itself will not be affected by the physical structure of the server.

In the modelled system, there is one active primary object and several passive backup objects. The state of the active object is logged. If the primary object fails, one of the backup objects is updated to a state corresponding to the state the primary had before the task started to run, and this backup is made active and used to rerun the task the primary was running. The conversion of the states between the different kinds of replicas is either done as a part of the logging process, creating extra overhead during normal operation, or during fault correction, making the correction process take more time.

Fault detection is here supposed to be based on some kind of watchdog mechanism, which often is the case in real-time systems. There is regular communication between the active object and a fault detector, so the fault detector gets messages indicating that the active object "is alive". If the fault detector doesn't get a message from the active object in what is considered a reasonable time, it is assumed that the active object has failed, and the fault correction mechanism is activated. It is thus mainly faults that cause the running object

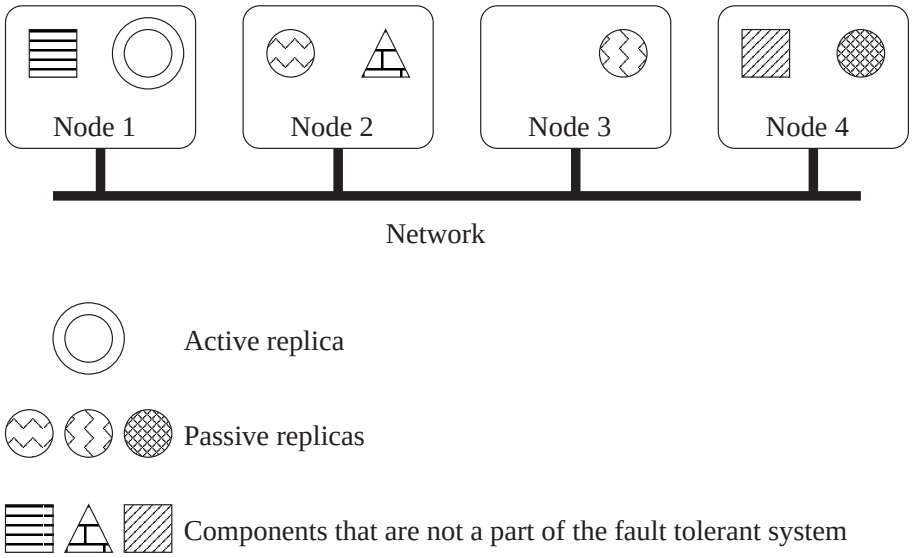


Figure D.1: Fault tolerant system with replicas distributed over several nodes

to be irresponsive or to crash that are tolerated. Some other faults that cause errors that can be observed by an external fault detection mechanism while the object is running, e.g. errors that cause the object to raise an exception, can also be covered by this model as long as the fault tolerance mechanism's way of correcting the faults is the same.

When a fault is detected, the fault correction mechanism will prepare one of the backup objects to run the task. The state of the object is updated from the log, and the task that was running on the failed object will be rerun on on the backup. The timing behaviour of the system when a fault occurs is shown in figure D.2.

There is a possibility that several faults will occur during the service of one task, something that must be reflected in the model. If a new fault occurs during the correction of the previous fault, or during the rerun of the task, the whole detection–correction–rerun cycle will recur. However, if the fault occurs before the previous fault is detected, it will have no additional effect, as the active object it affects has already failed.

Faults are modelled as independent, generated by a poisson process with a constant intensity, a common assumption in reliability modelling.

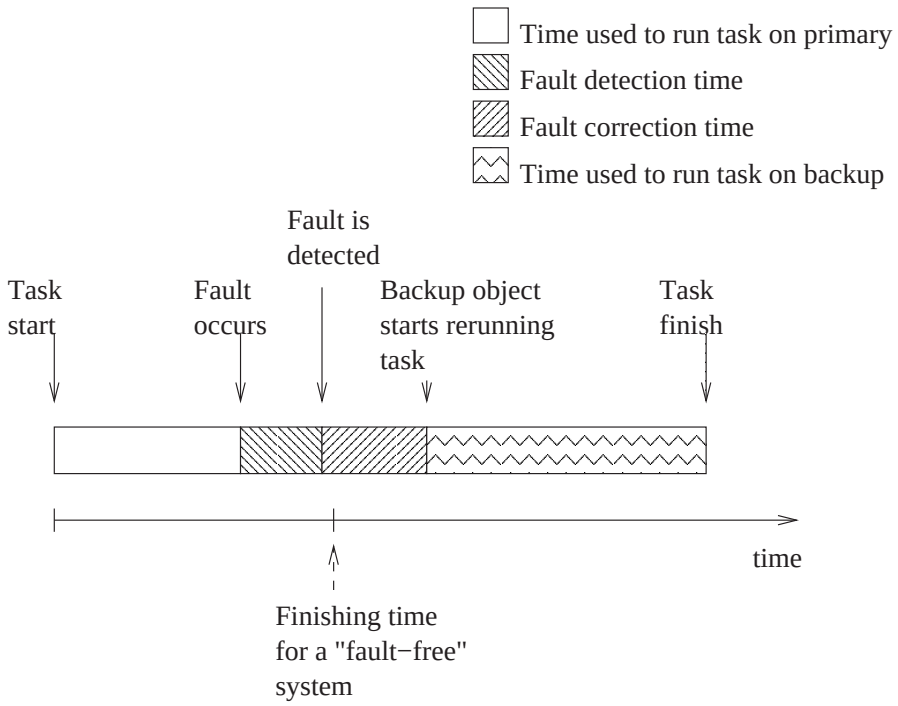


Figure D.2: Timing behaviour of system when a fault occurs

D.3 The mathematical model

The mathematical model is derived using the same method that we used for homogeneous replication systems in [46], which is based on the method used by Kleinrock [21] to derive the busy period in a queueing system.

D.3.1 Notation

For all equations, function names with normal capital letters ($F(t)$) indicate cumulative distribution functions, function names with normal lower-case letters ($f(t)$) indicate probability density functions, while function names with calligraphic letters ($\mathcal{F}(s)$) indicate moment generating functions. Function names with the same letter and index indicate the same distribution, e.g. $F_\alpha(t)$, $f_\alpha(t)$

and $\mathcal{F}_\alpha(s)$ are the cdf, pdf and mgf for the same distribution.

In the equations, we will use the following naming conventions for the different distributions:

$G(t)$, $g(t)$, $\mathcal{G}(s)$ The runtime distribution for the fault-tolerant system.

$M_i(t)$, $m_i(t)$, $\mathcal{M}_i(s)$ The fault-free runtime distribution for replica i. $i = 0$ indicates the runtime for the primary replica.

$C_i(t)$, $c_i(t)$, $\mathcal{C}_i(s)$ The fault correction time distribution for replica i. $i = 1$ indicates the time used to prepare the first backup object.

$I_i(t)$, $i_i(t)$ $\mathcal{I}_i(s)$ The fault detection time distribution for replica i. $i = 0$ indicates the time used to detect a fault in the primary object.

$H_\tau(t)$, $\mathcal{H}_\tau(s)$ The time-to-fault in a faulty run where the fault-free runtime would have been τ.

D.3.2 The fault model

The faults are modelled as generated from a poisson process with intensity λ. If the execution of a method takes the time τ, the probability that at least one fault happens before the execution ends is given by

$$p_\tau = 1 - e^{-\lambda\tau} \tag{D.1}$$

Given that a fault occurs during the execution, the cumulative density function (cdf) for the time from the start of the execution to the fault occurs is given by

$$H_\tau(t) = \begin{cases} \frac{1}{1-e^{-\lambda\tau}}(1 - e^{-\lambda t}) & , \quad 0 \le t \le \tau \\ 1 & , \quad t > \tau \end{cases} \tag{D.2}$$

This distribution has the moment generating function (mgf)

$$\mathcal{H}_\tau(s) = \frac{1 - e^{(\lambda+s)\tau}}{1 - e^{-\lambda\tau}} \frac{\lambda}{\lambda + s} \tag{D.3}$$

D.3.3 The system without detection and correction delays

First, we look at a model where the time used for detection and correction of the fault is 0, i.e., as soon as a fault occurs, the task will start running on a backup replica.

Appendix D. A Mathematical Model for Run-Time Distributions in a Fault Tolerant System with Nonhomogeneous Passive Replicas

For a system where the runtimes for the replicas are inhomogeneous x_i, where x_0 indicate the runtime for the primary replica, the number of faults ϕ before one of the replicas completes a fault-free execution is given by the probability function

$$P[\phi = k] = \begin{cases} e^{-\lambda x_0} & , \quad k = 0 \\ e^{-\lambda x_k} \prod_{i=0}^{k-1} (1 - e^{-\lambda x_i}) & , \quad k > 0 \end{cases} \tag{D.4}$$

The total runtime of a system with ϕ faults is given by

$$Y = r_\phi + X_0 + X_1 + \cdots + X_{\phi-1} \tag{D.5}$$

where X_i is the time to fault for each faulty run, and r_ϕ is the runtime for the fault-free run.

What we wish to find is the distribution of the total time use, expressed by the mgf

$$\begin{aligned} \mathcal{G}(s) &= \int_0^\infty e^{-st} dG(t) \\ &= E[e^{-sY}] \\ &= E[e^{-s(r+X_0+X_1+\cdots+X_{\phi-1})}] \end{aligned} \tag{D.6}$$

We start by introducing conditions on run-times and number of faults

$$\begin{aligned} &E[e^{-sY} | r_\phi = x_\phi, \phi = k] \\ &= E[e^{-s(x_k+X_0+X_1+\cdots+X_{k-1})}] \\ &= E[e^{-sx_k} e^{-sX_0} e^{-sX_1} \cdots e^{-sX_{k-1}}] \end{aligned} \tag{D.7}$$

Because the runtimes x_k and $X_0 \ \cdots \ X_{k-1}$ are independent, we can rewrite this as

$$\begin{aligned} &E[e^{-sY} | r_\phi = x_\phi, \phi = k] \\ &= E[e^{-sx_k}] E[e^{-sX_0}] E[e^{-sX_1}] \cdots E[e^{X_{k-1}}] \end{aligned} \tag{D.8}$$

Using $E[e^{-sX_i}] = \mathcal{H}_{x_i}(s)$, we can rewrite this as

$$E[e^{-sY} | r_\phi = x_\phi, \phi = k] = E[e^{-sx_k}] \prod_{i=0}^{k-1} \mathcal{H}_{x_i}(s) \tag{D.9}$$

We now remove the condition on the number of faults. This is done by multiplying the probability for a given number of faults with the runtime for

that number of faults, and summing these products:

$$\begin{aligned} &E[e^{-sY}|r_n = x_n] \\ &= e^{-sx_0}e^{-\lambda x_0} + \sum_{k=1}^{\infty} e^{-sx_k}e^{-\lambda x_k} \prod_{i=0}^{k-1} \mathcal{H}_{x_i}(s)(1 - e^{-\lambda x_i}) \\ &= e^{-(\lambda+s)x_0} + \sum_{k=1}^{\infty} e^{-(\lambda+s)x_k} \prod_{i=0}^{k-1} (1 - e^{-(\lambda+s)x_i}) \frac{\lambda}{\lambda+s} \end{aligned} \tag{D.10}$$

A problem with this equation "as it is" is the infinite sum in it, which makes it difficult to work with. This can be explained as no limit to the number of faults that can happen during the execution of a single task. To work around this problem, we set a maximum number of faults, N, that can happen to a single task and still be tolerated by the system. If more than N faults happen to the system during the execution of a single task, we consider the system to have failed. Even for a relatively low N, this is a reasonable approximation of the system's behaviour, since the probability of faults happening is usually very low. Also, because of the extra time used to tolerate faults, we can assume that a task will miss the deadline, and thus fail anyway, if there is more than N faults.

The equation with a maximum of N faults before a failure is given by:

$$\begin{aligned} &E[e^{-sY}|r_n = x_n] \\ &= e^{-(\lambda+s)x_0} + \sum_{k=1}^{N} e^{-(\lambda+s)x_k} \prod_{i=0}^{k-1} (1 - e^{-(\lambda+s)x_i}) \frac{\lambda}{\lambda+s} \end{aligned} \tag{D.11}$$

The condition on time use is removed by integrating the expression with respect to the runtime distributions

$$\begin{aligned} &\mathcal{G}(s) \\ &= E[e^{-sY}] \\ &= \int_0^{\infty} \cdots \int_0^{\infty} E[e^{-sY}|r_n = x_n] dM_0(x_0) \cdots dM_N(x_N) \end{aligned} \tag{D.12}$$

By using the definition of the moment generating function, we get

$$\begin{aligned} &\mathcal{G}(s) \\ &= \mathcal{M}_0(\lambda + s) \\ &\quad + \sum_{k=1}^{N} \left(\frac{\lambda}{\lambda+s} \right)^k \mathcal{M}_k(\lambda + s) \prod_{i=0}^{k-1} (1 - \mathcal{M}_i(\lambda + s)) \end{aligned} \tag{D.13}$$

Equation D.13 shows the mgf for the distribution of the runtime in a serial-structured fault tolerant system without the fault detection and fault correction times.

D.3.4 Adding the fault correction delay

We will now expand the expression by adding the delay for fault correction. The correction process consists of updating the backup objects, so that the state of these objects reflects the state of the primary object. We also assume that faults may happen during the correction process.

This can be done by changing the equations in D.3.3 so that the correction time is included. Equation D.4 is changed to

$$P[\phi = k] = \begin{cases} e^{-\lambda x_0} & , k = 0 \\ e^{-\lambda(x_k+y_k)} \prod_{i=0}^{k-1} (1 - e^{-\lambda(x_i+y_i)}) & , k > 0 \end{cases} \tag{D.14}$$

and equation D.9 is changed to

$$\begin{aligned} &E[e^{-sY} | r_n = x_n, c_n = y_n, \phi = k] \\ &= \begin{cases} e^{-sx_0} & , \quad k = 0 \\ e^{-s(x_k+y_k)} \mathcal{H}_{x_0}(s) \prod_{i=1}^{k-1} \mathcal{H}_{x_i+y_i}(s) & , \quad k > 0 \end{cases} \end{aligned} \tag{D.15}$$

The expression is otherwise derived as in D.3.3. This gives us the mgf for the system with fault correction time included:

$$\begin{aligned} &\mathcal{G}(s) \\ &= \mathcal{M}_0(\lambda + s) \\ &\quad + \sum_{k=1}^{N} \left(\frac{\lambda}{\lambda + s} \right)^k \mathcal{M}_k(\lambda + s) \mathcal{C}_k(\lambda + s) \\ &\quad \prod_{i=0}^{k-1} (1 - \mathcal{M}_i(\lambda + s) \mathcal{C}_i(\lambda + s)) \end{aligned} \tag{D.16}$$

where $\mathcal{C}_0(s) = 1$; since the primary replica is considered to be up to date, the correction time for this replica is always zero, i.e. $c_0(t) = \delta(t)$.

Equation D.16 can also be derived by combining the correction time with the run time for the replicas, i.e. let the run time for replica n be distributed with pdf $c_n(t) * m_n(t)$ where $*$ is the convolution operator, which in the mgf domain corresponds to a simple multiplication, i.e. $\mathcal{C}(s)\mathcal{M}(s)$.

D.3.5 Adding the fault detection delay

The equation is expanded further by adding the delay used for fault detection. During this delay, the method that is running is already in an erroneous state, so new faults happening during this time can be ignored.

The probability on the number of faults will thus be the same as for a system without fault detection, i.e. equation D.14.

The expected time use for a system where k faults occurs will be

$$\begin{aligned} &E[e^{-sY}|r_n = x_n, c_n = y_n, d_n = z_n, \phi = k] \\ &= \begin{cases} e^{-sx_0} & , k = 0 \\ e^{-s(x_k+y_k)}\mathcal{H}_{x_0}(s)\prod_{i=1}^{k-1}\mathcal{H}_{x_i+y_i}(s)e^{sz_i} & , k > 0 \end{cases} \end{aligned} \tag{D.17}$$

The mgf for a system with fault detection and fault correction times included is derived the same way as in D.3.3. This gives the final result:

$$\begin{aligned} &\mathcal{G}(s) \\ &= \mathcal{M}_0(\lambda + s) \\ &\quad + \sum_{\substack{k=1 \\ k-1}}^{N} \left(\frac{\lambda}{\lambda + s}\right)^k \mathcal{M}_k(\lambda + s)\mathcal{C}_k(\lambda + s) \\ &\quad \prod_{i=0}^{k-1}(1 - \mathcal{M}_i(\lambda + s)\mathcal{C}_i(\lambda + s))\mathcal{I}_i(s) \end{aligned} \tag{D.18}$$

Equation D.18 gives an expression for the total runtime for the class of fault tolerant systems described in section D.2 as a function of the distributions of the fault free runtimes, the correction times, and the fault detection times of the systems. We regard this as the main result of the paper.

We will now show how the results can be used for various systems.

D.4 Examples

In this section we will show how the equation derived in the previous section can be used for analysing the timing behaviour in several systems, and compare the results with simulator results.

For simplicity, we let all systems in the examples use the same structure; one primary and two backup objects. We can then use the same equation for all

systems, and vary the parameters and distributions according to the systems we are analyzing. The equation we use is found by setting $N = 2$ in equation D.18

$$\begin{aligned}\mathcal{G}(s) = \; & \mathcal{M}_0(\lambda+s) \\ & + \tfrac{\lambda}{\lambda+s}\mathcal{M}_1(\lambda+s)\mathcal{C}_1(\lambda+s)(1-\mathcal{M}_0(\lambda+s))\mathcal{I}_0(s) \\ & + \left(\tfrac{\lambda}{\lambda+s}\right)^2 \mathcal{M}_2(\lambda+s)\mathcal{C}_2(\lambda+s) \\ & \quad (1-\mathcal{M}_1(\lambda+s)\mathcal{C}_1(\lambda+s))\mathcal{I}_1(s) \\ & \quad (1-\mathcal{M}_0(\lambda+s))\mathcal{I}_0(s)\end{aligned} \tag{D.19}$$

Because the inverse transform of the moment generating functions we get from this expression will be quite complicated, Matlab and Simulink is used to numerically transform the results to the time domain.

Given that the fault-tolerant system makes the occurence of failures, except timing failures, negligible, and that the deadlines are hard (i.e., timing faults will lead to the failure of the system), the reliability function for the system can be expressed as

$$R(n) = G(t_{dl})^n \tag{D.20}$$

where t_{dl} is the deadline and n is the number of times the task is performed.

For each system, the calculated cdf of the system's completion time is plotted together with the cdf of the results from the simulations, and the reliability functions for the system with different deadlines are plotted and compared to the reliability function for a non-fault tolerant system.

D.4.1 The simulator

We have created a simple simulator for simulating the passive replication fault tolerant systems described in section D.2. The simulator was developed using C++ and the ADEVS framework for discrete event simulations [1]. A block diagram of the simulator is shown in figure D.3.

In the systems presented here, the client and fault generator functions as pure generators of respectively *process* and *fault* objects, which are sent to the server part of the simulator.

During normal operation, the server part of the simulator is quite straightforward. When the server starts the service on a *process* object, a running time is drawn from the running time distribution, and if no event occurs during this time, the service ends when the time is up.

If a *fault* object arrives while the server is busy, the server will enter an *error* state, and a fault detection time will be drawn from the fault detection time distribution. When the fault detection time is up, the server will enter a *correction*

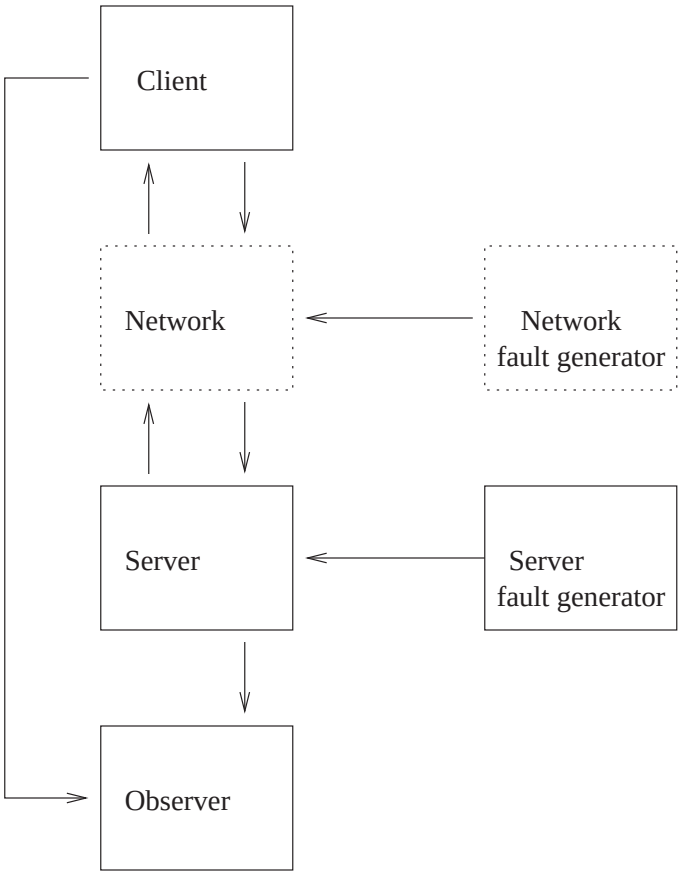


Figure D.3: Structure of a simple client-server simulator.

state, and draw a correction time from the correction time distribution. When the fault is corrected, the server will reenter the *busy* state, and a new running time is drawn for the *process* object. A simplified state machine diagram for the server part of the simulator is shown in figure D.4.

In both the fault-free and the faulty runs, the starting and ending time of the service is logged and sent to the observer part of the system, which presents the times in a format that can be read by Matlab.

The details of the simulator are discussed in [47].

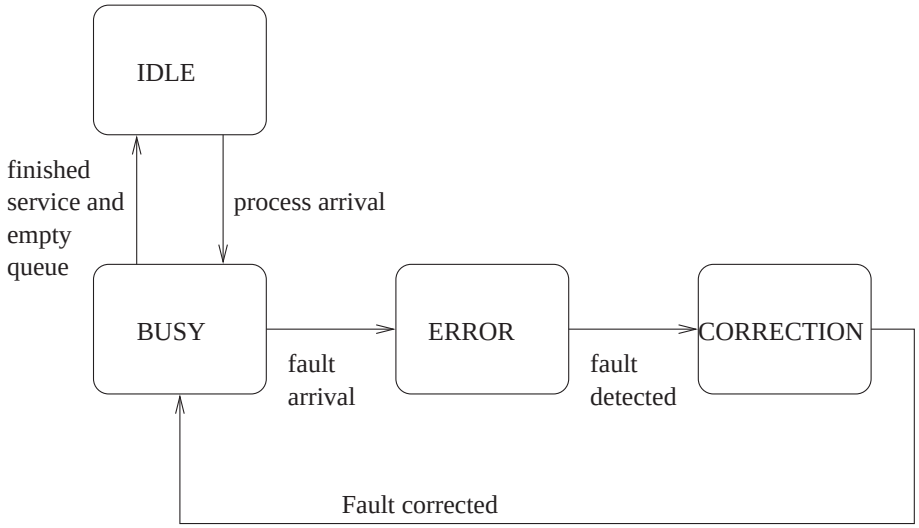


Figure D.4: Simplified state machine diagram for the server part of the simulator.

For each example, the simulator is set to run the task a reasonably large number of times, 500000. The results from the simulator are analyzed using Matlab.

D.4.2 A basic system

First, we look at a system where the distributions for run-time, fault correction and fault detection is the same for both the primary and the backup methods.

System parameters

For the fault-free running time, we choose a triangular distribution with a minimum time 6, maximum time 8, and mode 10.

$$m_0(t) = m_1(t) = m_2(t) = \begin{cases} 0 & ,0 \leq t < 6 \\ \frac{t-6}{4} & ,6 \leq t < 8 \\ \frac{10-t}{4} & ,8 \leq t < 10 \\ 0 & ,t \geq 10 \end{cases} \tag{D.21}$$

The fault detection time is uniformly distributed with a minimum time 1 and a maximum time 3.

$$i_0(t) = i_1(t) = \begin{cases} 0, & ,0 \leq t < 1 \\ \frac{1}{2} & ,1 \leq t \leq 3 \\ 0 & t > 3 \end{cases} \tag{D.22}$$

The fault correction time is uniformly distributed with a minimum time 0 and a maximum time 5.

$$c_1(t) = c_2(t) = \begin{cases} \frac{1}{5} & ,0 \leq t \leq 5 \\ 0 & ,t > 5 \end{cases} \tag{D.23}$$

These distributions have the following moment generating functions:

$$\mathcal{M}_0(s) = \mathcal{M}_1(s) = \mathcal{M}_2(s) = \frac{e^{-6s} - 2e^{-8s} + e^{-10s}}{4s^2} \tag{D.24}$$

$$\mathcal{I}_0(s) = \mathcal{I}_1(s) = \frac{e^{-s} - e^{-3s}}{2s} \tag{D.25}$$

$$\mathcal{C}_1(s) = \mathcal{C}_2(s) = \frac{1 - e^{-5s}}{5s} \tag{D.26}$$

The mean time between faults is set to 10000. This is an unrealistically high fault rate, but is chosen as it limits the number of simulation runs needed to get usable results.

Results

Details of the calculated and simulated cdf for this system is shown in figure D.5

The reliability functions for the system with different deadlines is plotted in figure D.6, together with the reliability function for a non-fault tolerant system. This shows that for this example, the improvement in reliability is negligible for deadlines near 10, while it is much improved for deadlines over 20.

D.4.3 Heterogeneous systems

In the second example, we look at systems where the primary is different from the backups.

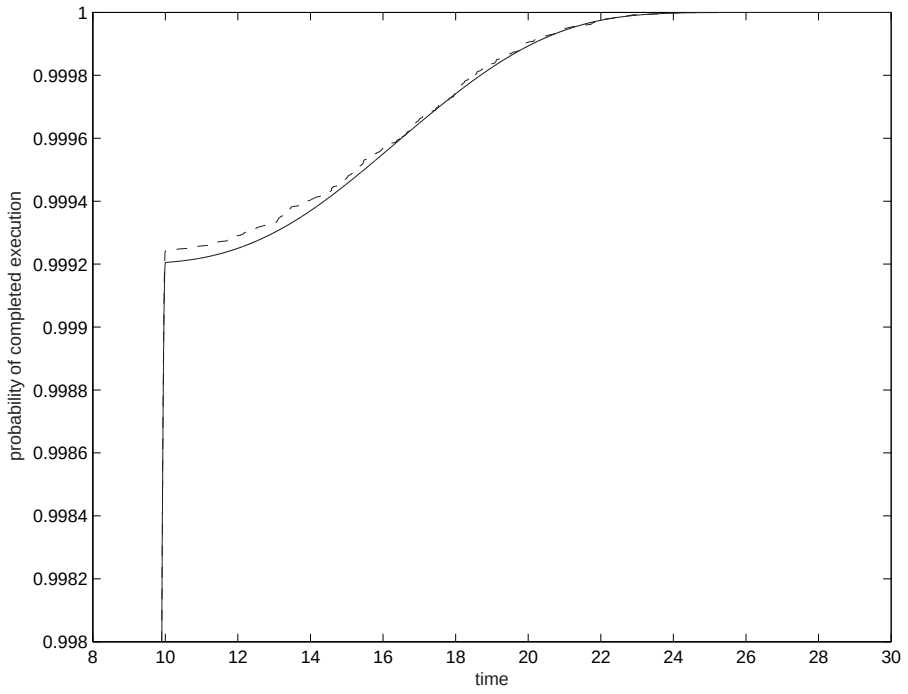


Figure D.5: Cumulative density function for the runtime distribution of the system described in D.4.2. The solid line shows the cdf calculated using the equation derived in the paper, the dashed line shows the cdf calculated from the simulator results.

System parameters

For the first system, we let the primary method have a deterministic running time of 9.

$$m_0(s) = \delta(t - 9) \tag{D.27}$$

$$\mathcal{M}_0(s) = e^{-9s} \tag{D.28}$$

The backups' running times are distributed with a triangular distrubution with the same parameters as in the previous example, i.e. equations D.21 and D.24 are used.

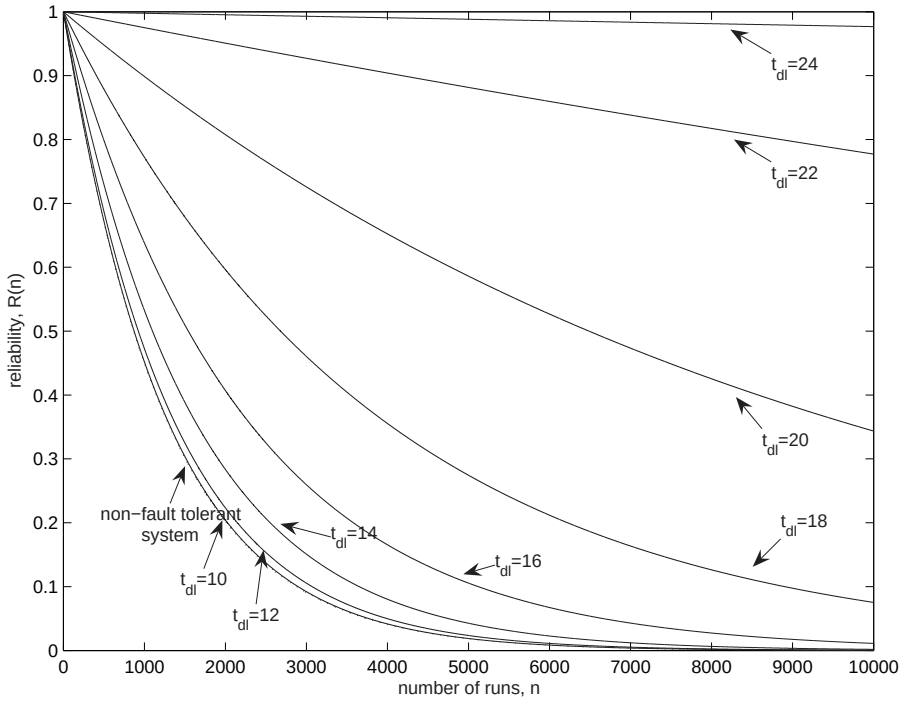


Figure D.6: The reliability functions for the system in D.4.2 with different deadlines, compared to the reliability function for a non-fault tolerant system

The fault correction and detection times are the same as in the previous example.

For the second system, we switch the distributions for the primary and backup running times, so that the primary has a triangular distributed running time and the backups have a deterministic running time.

Results

The calculated and simulated cdf of the first system is shown in figure D.7, while the calculated and simulated cdf of the second system is shown in figure D.8.

The reliability of the two systems at different deadlines is shown in figure D.9. Both systems have a poor improvement in reliability when the deadline is near

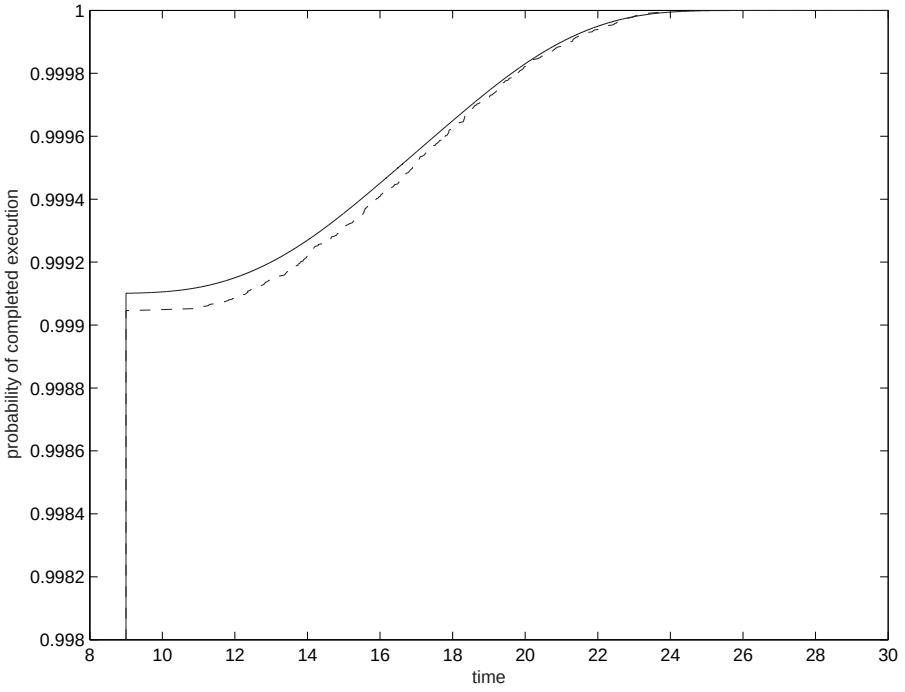


Figure D.7: Cumulative density function for the runtime of the first system described in D.4.3. The solid line indicates the calculated results, the dashed line indicates the simulated results.

10. At these deadlines the second system has a better reliability, because the average runtime of this system is better, and it will thus have a lower probability of faults with the fault model we use. For higher deadlines, the two systems has about the same reliability function, with a slightly higher reliability for the first system if the deadlines are over 20.

D.4.4 Imprecise systems

In the final example, we will look at a system with a precise, but slow, primary method and imprecise, but fast, backup methods.

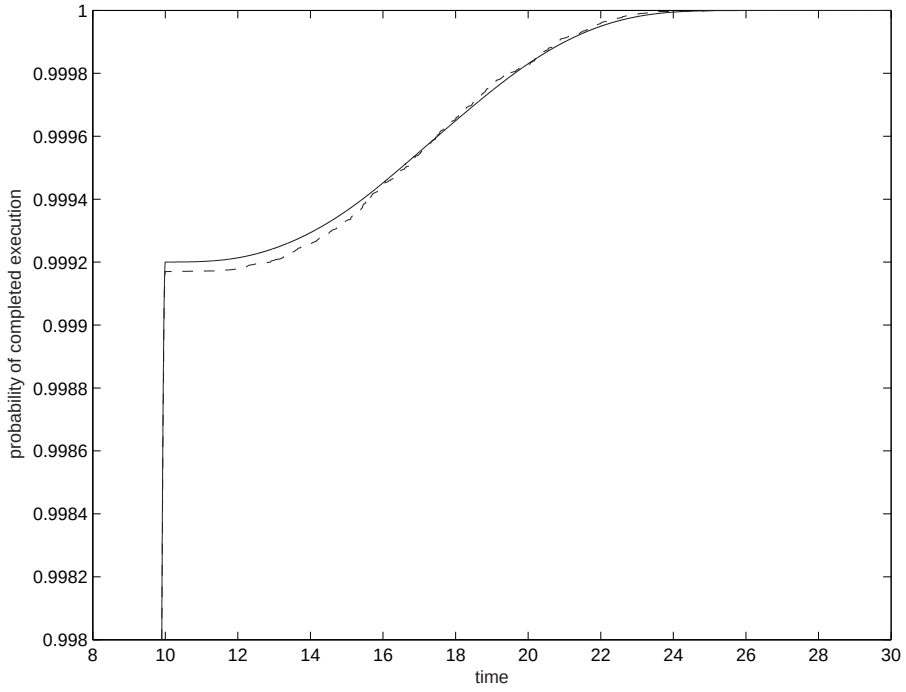


Figure D.8: Cumulative density function for the runtime of the second system described in D.4.3. The solid line indicates the calculated results, the dashed line indicates the simulated results.

System parameters

For the primary method, we use the same triangular distribution as described in earlier examples. The backups have a deterministic running time of 1.

$$m_1(t) = m_2(t) = \delta(t-1) \tag{D.29}$$

$$\mathcal{M}_1(s) = \mathcal{M}_2(s) = e^{-s} \tag{D.30}$$

Because of the relatively simple structure of the backups, we assume that the time used to bring them up to date is much shorter than for the backups in the

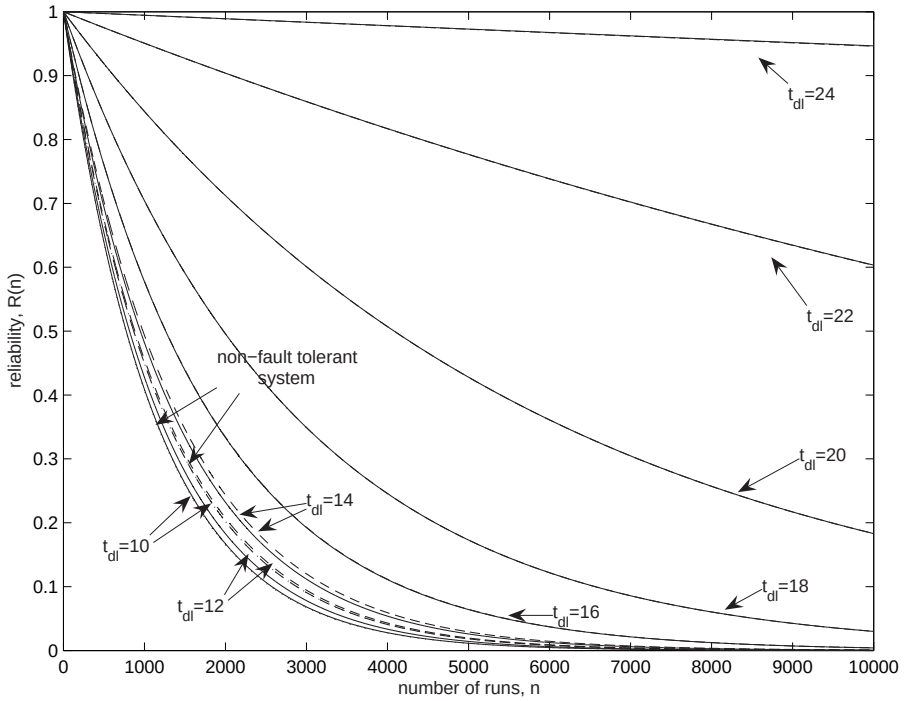


Figure D.9: Reliability functions for the systems described in D.4.3 at different deadlines. The solid lines indicate the first system, the dashed lines indicate the second system.

other example systems, and that it is distributed uniformly between 0 and 1.

$$c_1(t) = c_2(t) = \begin{cases} 1 & ,0 \leq t \leq 1 \\ 0 & ,t > 1 \end{cases} \tag{D.31}$$

$$\mathcal{C}_1(s) = \mathcal{C}_2(s) = \frac{1 - e^{-s}}{s} \tag{D.32}$$

The fault detection times are as in the previous examples.

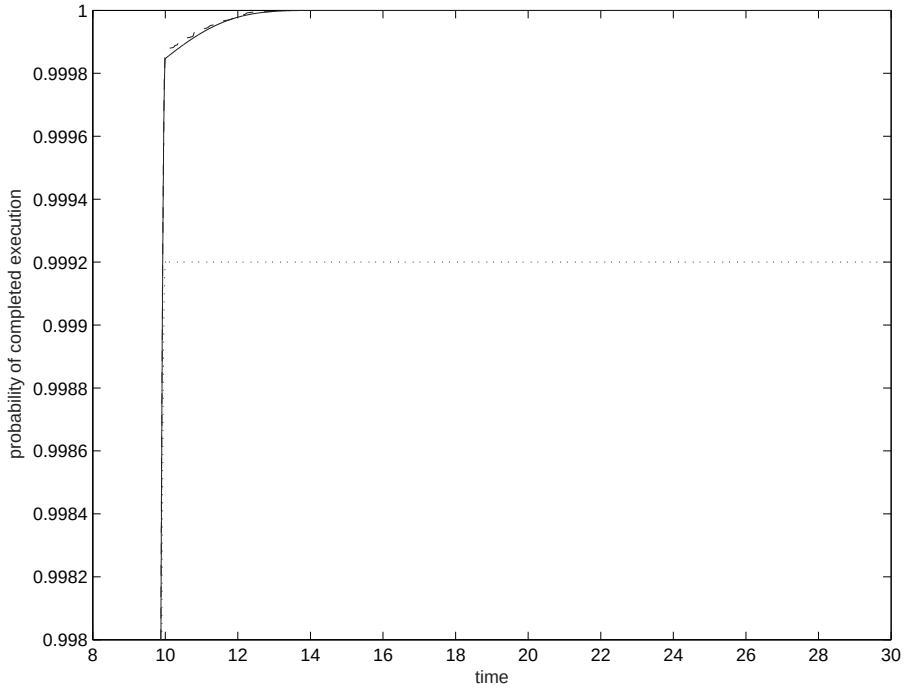


Figure D.10: Cumulative density functions for the runtime distribution of the system with imprecise backups described in D.4.4. The solid line indicates the calculated results, the dashed line indicates the simulated results. The dotted line indicates the probability of getting a precise result.

Results

The results are plotted in figure D.10, and show that we can get good timing results for this system if we allow results to be imprecise in the case of faults. The dotted line show the probability that the results are precise.

The reliability of the system with different deadlines are shown in figure D.11, showing a significant improvement in reliability even for a deadline of 10. For a deadline of 14, the system shows a very high reliability.

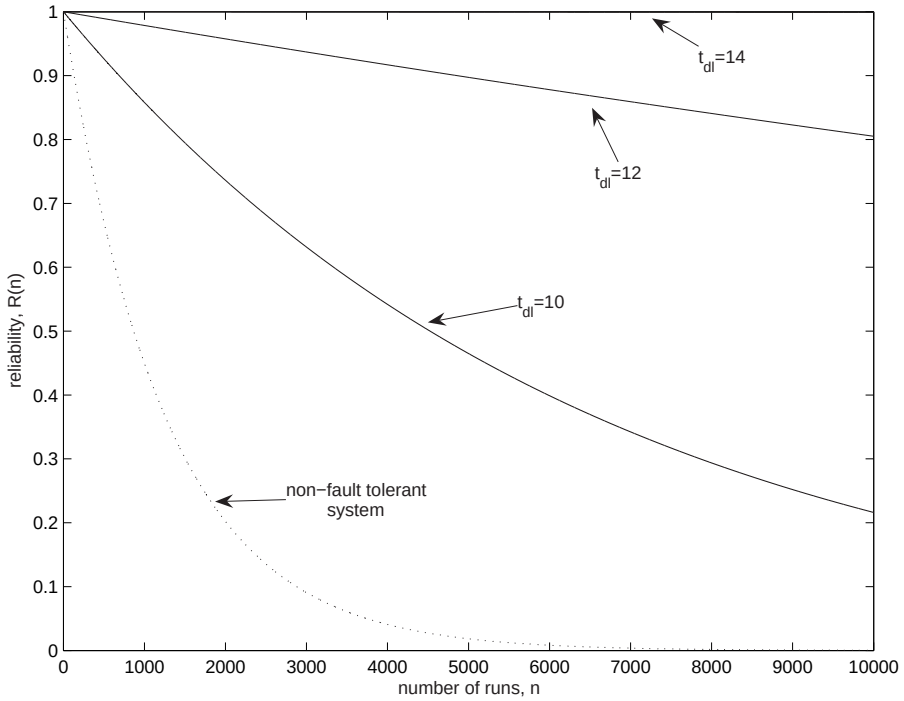


Figure D.11: Reliability for the system described in D.4.4 at different deadlines.

D.5 Discussions and conclusion

In the models we present here, we assume that faults are independent, and can be detected and corrected by the fault tolerance mechanism discussed in section D.2 or similar mechanisms. While this often will be good enough for the analysis of a system, there will be times when other fault behaviours and other detection and correction mechanisms must be modelled, e.g. non-independent faults or non-fail-silent behaviour, and fault detection by acceptance tests. While not covered by our present work, we believe that a mathematical model of the timing behaviour in such systems can be derived in a similar way as the one we used in section D.3. This will be pursued in future work.

In this paper, we have derived an expression for the run-time distribution of a method in a fault tolerant system based on passive replication where inhomo-

geneous replicas are used. This expression is a function of the distributions of the fault free runtimes, the fault detection times and the fault correction times in the system. We believe this to be a useful tool for the analysis of fault tolerant real-time systems. To demonstrate this, we have shown how these equations can be used for several example systems, and how they can be used to determine the probability of missed deadlines, and the corresponding reliability characteristics of the system.

Appendix D. A Mathematical Model for Run-Time Distributions in a Fault Tolerant System with Nonhomogeneous Passive Replicas

Appendix E

Run-time Distributions in Passively Replicated Systems Using Timeout and Acceptance Fault Detection

By Åsmund Tjora and Amund Skavhaug

This paper was presented at the ERCIM/DECOS Workshop on Dependable Embedded Systems in Lübeck, Germany, August 2007 [48]. The proceedings of this workshop have not yet been published.

Abstract

Fault tolerance based on passive replication is common in many systems. If this kind of fault tolerance mechanism is to be used in a real-time system, timing analysis is necessary, as the fault tolerance mechanism itself may cause timing faults.

There are different ways of detecting when the primary replica has failed, one of them is to use a timeout to detect crash and omission failures, another to run acceptance tests on the results, which detects some value failures. As these two detection strategies cover different kinds of failures, it can be useful to combine them.

In this paper, a mathematical model for the timing behaviour of a system with passive replication, where a combination of timeout and acceptance test is used for fault detection, is derived.

Examples are given to demonstrate how the model can be used for calculation of deadline miss probabilities, and further how this can be used for checkpoint placement optimisation.

E.1 Introduction

Many real-time systems will, in addition to the timing requirements, have requirements to reliability. To fullfill the reliability requirements, a fault tolerance mechanism might be necessary. While the fault tolerance mechanism may detect and correct many of the faults that occur to the system, it may also introduce new faults if the extra time used for detection and correction cause deadline misses. Because of this, the reliability improvement given by some fault tolerance mechanisms may be very small, depending on the system's real-time requirements. The effect of using fault tolerance mechainsms in real-time systems is therefore an important study, as noted in [43].

In fault tolerant real-time systems, using active replication structures, i.e. structures where a task is run on several replicas simultaneously, has been common. Even if errors are detected in some of the replicas, the non-erroneous replicas will still be able to produce results within the deadlines. In addition, comparision of the results from the different replicas can be used as fault detection, increasing the range of faults that can be detected. On the negative side, running several replicas of the same task simultaneously do require extra hardware resources, which can be costly.

Passive replication strucures are less resource consuming. In these struc-

tures, only one of the replicas running the task is active, while the other tasks are passive. If an error is detected in the active replica, it is stopped, and one of the backup replicas is prepared and made active. The task that ran on the failed replica is then rerun on the new active replica. Because of the extra time used to detect a fault, prepare a backup, and rerun the task, these fault tolerant mechanisms may cause deadline misses if used in a real-time system.

This does not mean that passive fault tolerant structures are unsuitable for all real-time applications. It is still possible that many of the faults are tolerated within the deadlines, thus improving the system's reliability without using a more resource consuming fault tolerance mechanism. Analysis of the time use becomes necessary to determine how much the reliability can be improved while the task still meets its deadlines.

For the fault tolerant system to function properly, it is necessary to have a mechanism that determines when the active replica has failed. Using a timeout on the execution of a task is a fairly simple fault detection mechanism. If the task has not finished its execution within a given time, it is considered an omission failure. Another fault detection mechanism is to run acceptance tests [37] on results from the task. These tests may detect some value failures, but are usually unable to determine the correctness of the results. Combining timeout and acceptance tests makes the system able to detect omission failures and some value failures.

In earlier works [46, 47], we have derived runtime models where we have focused on a fault detection mechanism where the task periodically signals that it is "alive" to the fault detector, and where it is assumed that the running replica has failed if the "alive" signals disappears. In those systems, the faults are detected a time after fault occurrence that is independent of the remaining execution time of the task. In this paper, the runtime model for systems using a combination of timeout and acceptance test for fault detection is derived.

The rest of the paper is organized as follows:

Section E.2 gives a textual description of the modelled systems.

Section E.3 contains the derivation of the mathematimal model for the runtime distribution.

Section E.4 contains examples of the use of the model

Section E.5 contains discussions and the conclusion of the paper.

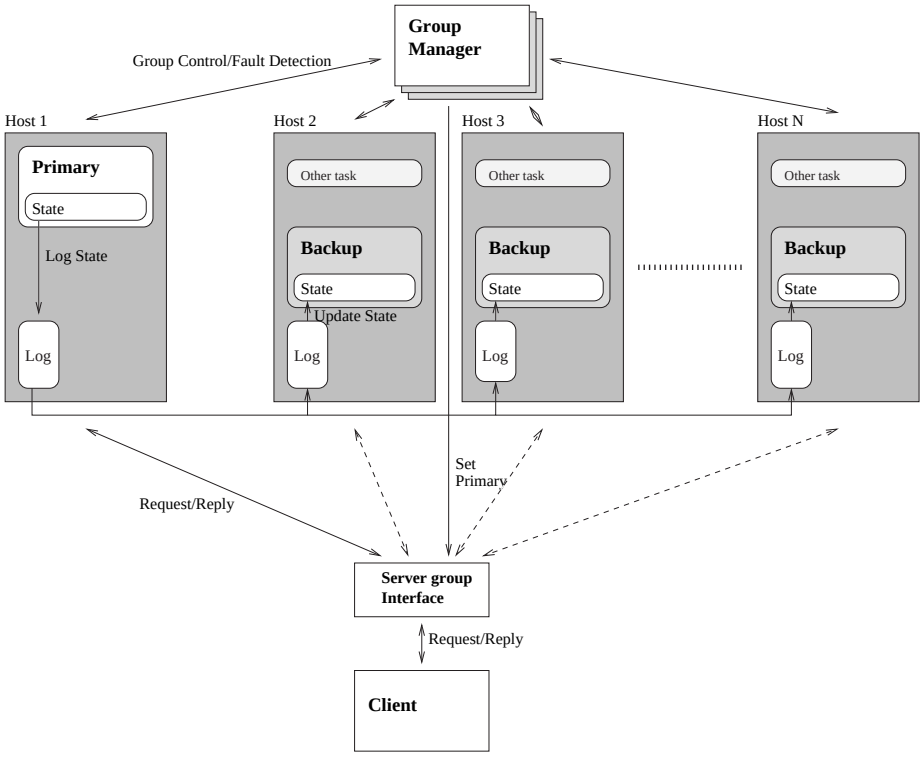


Figure E.1: A passively replicated task in a distributed system

E.2 Description of the modelled system

The class of systems that are modelled can be described as a server using passive replication mechanisms to achieve fault tolerance. In these systems, there are one active primary replica and several passive backup replicas. The replicas, as well as the fault tolerance mechanisms may be on the same node, or they may be distributed over several nodes, as shown in figure E.1. The physical distribution of the system will affect the timing distributions used in the models, as well as the system's ability to tolerate some faults, but the models themselves will not be affected by the physical structure of the server.

When operating normally, the state of the primary replica is logged. The log is used when updating the passive replicas and when creating new replicas.

The strategy for updating the passive replicas may vary. Frequent updates will cause more overhead during normal operation than infrequent updates, but as a result, the backups will have a state that are closer to the primary's state, so the time used to update the backups in a fault situation is shorter.

Two fault detection mechanisms are used. The timeout mechanism detects if the primary replica does not deliver results within a set maximum time, thus faults causing omission and silent failures are detected by this mechanism. The acceptance test checks the results to see if they meet criteria that all acceptable results must meet, and can thus detect some value failures.

When a fault is detected, the fault tolerance mechanism prepares one of the backups to run the task. The state of the backup is updated from the log, and this backup becomes the new primary. The task is then rerun. The timing behaviour of the system when faults occur is shown in figure E.2.

Faults are modeled using constant fault probabilities for each part of the execution (i.e. running the task on the primary, correction, and rerun on backup) of the system. As the faults are detected at either the timeout or after the acceptance test, we need to know that a fault has occured and whether it caused a value or an omission failure, but we do not need the exact time of the fault occurences.

There is a possibility that several faults occur during the service of one task. If a new fault occurs during the correction of previous faults or during the rerun of the task, a new detection–correction–rerun cycle will recur. If more than one fault occur before detection, the detection will be at the timeout if at least one of the faults cause an omission failure and at the acceptance test if all faults cause value failures.

E.3 Deriving the mathematical model

The mathematical model for the total runtime distribution of the tasks in the fault tolerant system is a function of the distributions for the fault free runtimes of the tasks, the acceptance test times, and the time between a fault detection and the rerun of the task (called correction time in this work), as well as the timeout values and the fault probabilities. The derivation of the expression is similar to the derivation of the busy period in queueing systems [21].

The distributions used in the expressions are represented by their moment-generating functions, which can be viewed as the laplace transform of the prob-

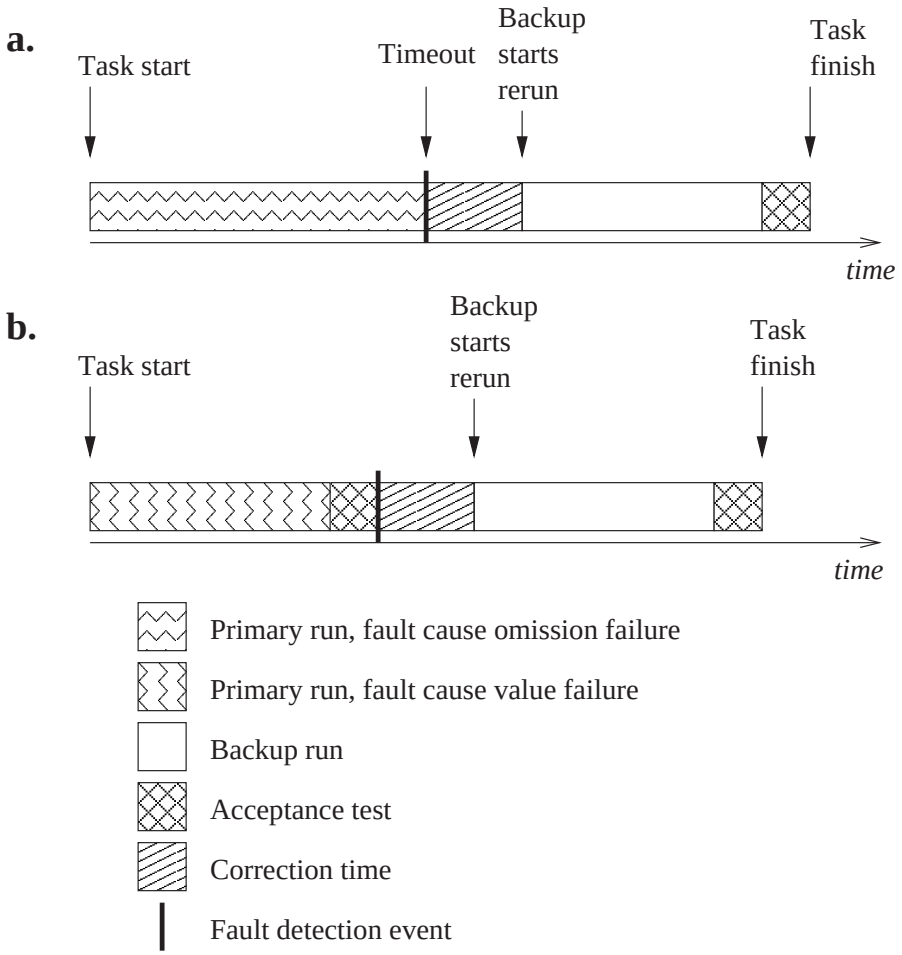


Figure E.2: Timing of the fault tolerant system when a fault is detected at timeout (a) and when a fault is detected by the acceptance test (b)

ability density function:

$$\mathbf{F}(s) = \mathcal{L}(f(t)) = \int_0^\infty e^{-st} f(t)dt = \int_0^\infty e^{-st} dF(t) \tag{E.1}$$

In this work, moment-generating functions are given boldface function names (e.g. $\mathbf{F}(s)$), probability density functions are given lower-case function names (e.g. $f(t)$), and cumulative distribution functions are given upper-case function names (e.g. $F(t)$). Different represenations of the same distributions are given the same letter and index (e.g. $\mathbf{F}_\alpha(s)$, $f_\alpha(t)$, and $F_\alpha(t)$ are different representations of the same distribution).

The function names used in this work are

$\mathbf{G}(s)$, $g(t)$, $G(t)$ The runtime distribution of a task in a system where faults may occur and be tolerated, i.e. the runtime distribution that are derived in this section.

$\mathbf{M}_i(s)$, $m_i(t)$, $M_i(t)$ The fault-free runtime distribution of a task on replica i.

$\mathbf{C}_i(s)$, $c_i(t)$, $C_i(t)$ The correction time distribution, i.e. the distribution of the time used from a fault in replica $i-1$ is detected to replica i is ready to rerun the task.

$\mathbf{D}(s)$, $d(t)$, $D(t)$ The distribution of the time used for acceptance test.

$\mathbf{A}_i(s)$, $a_i(t)$, $A_i(t)$ The distribution of the time used from a failed run of the task on replica i starts to the fault detection.

E.3.1 The fault model

As described in the previous section, a fault model with fixed fault probability for each part of the system is used. For each replica i, there is a probability $\kappa_{\mathrm{co}i}$ that there is no omission failure while readying the replica (i.e. the correction process), there is a probability $\kappa_{\mathrm{mo}i}$ that there is no omission failure while running the replica, and there is a probability $\kappa_{\mathrm{v}i}$ that there is no value failure detected when testing the results.

If a replica fails, there are three possibilities of where the failure is detected:

- Omission failure during correction, detection is at the timeout for the correction, $\tau_{\mathrm{c}i}$
- No omission failure during correction, but omission failure during the execution of the task. Detection is at the timeout for the execution, $\tau_{\mathrm{m}i}$ after the execution started.
- No crash or omission failure, but a value failure detected by the acceptance test.

The probability for at least one of these failures occur while running replica i is given by

$$\Pr(\phi \geq 1) = 1 - \kappa_{\mathrm{co}i}\kappa_{\mathrm{mo}i}\kappa_{\mathrm{v}i} \tag{E.2}$$

For a run where there is no correction time (e.g. the primary replica when the task execution starts), a fault may be detected either at the timeout, $\tau_{\mathrm{m}i}$ after the execution starts, or after the normal runtime x_i and the acceptance test time z. The time to failure detection is distributed with the pdf

$$\begin{aligned} &a_{\mathrm{nc}i}(t) = \\ &\frac{1-\kappa_{\mathrm{mo}i}}{1-\kappa_{\mathrm{mo}i}\kappa_{\mathrm{v}i}}\delta(t-\tau_{\mathrm{m}i}) \\ &+\frac{1-\kappa_{\mathrm{v}i}}{1-\kappa_{\mathrm{mo}i}\kappa_{\mathrm{v}i}}\kappa_{\mathrm{mo}i}\delta(t-(x_i+z)) \end{aligned} \tag{E.3}$$

which has the mgf

$$\begin{aligned} &\mathbf{A}_{\mathrm{nc}i}(s) = \\ &\frac{1-\kappa_{\mathrm{mo}i}}{1-\kappa_{\mathrm{mo}i}\kappa_{\mathrm{v}i}}e^{-s\tau_{\mathrm{m}i}} \\ &+\frac{1-\kappa_{\mathrm{v}i}}{1-\kappa_{\mathrm{mo}i}\kappa_{\mathrm{v}i}}\kappa_{\mathrm{mo}i}e^{-s(x_i+z)} \end{aligned} \tag{E.4}$$

If there is a correction time, and a separate timeout function for this, a fault may be detected at the timeout for the correction $\tau_{\mathrm{c}i}$, after the normal correction time y_i and the timeout for the runtime $\tau_{\mathrm{m}i}$, or after the correction, normal runtime and acceptance test, $y_i + x_i + z$. The time to failure detection is distributed with the pdf

$$\begin{aligned} &a_i(t) = \\ &\frac{1-\kappa_{\mathrm{co}i}}{1-\kappa_{\mathrm{co}i}\kappa_{\mathrm{mo}i}\kappa_{\mathrm{v}i}}\delta(t-\tau_{\mathrm{c}i}) \\ &+\frac{1-\kappa_{\mathrm{mo}i}}{1-\kappa_{\mathrm{co}i}\kappa_{\mathrm{mo}i}\kappa_{\mathrm{v}i}}\kappa_{\mathrm{co}i}\delta(t-(y_i+\tau_{\mathrm{m}i})) \\ &+\frac{1-\kappa_{\mathrm{v}i}}{1-\kappa_{\mathrm{co}i}\kappa_{\mathrm{mo}i}\kappa_{\mathrm{v}i}}\kappa_{\mathrm{co}i}\kappa_{\mathrm{mo}i}\delta(t-(y_i+x_i+z)) \end{aligned} \tag{E.5}$$

which has the mgf

$$\begin{aligned} &\mathbf{A}_i(s) = \\ &\frac{1-\kappa_{\mathrm{co}i}}{1-\kappa_{\mathrm{co}i}\kappa_{\mathrm{mo}i}\kappa_{\mathrm{v}i}}e^{-s\tau_{\mathrm{c}i}} \\ &+\frac{1-\kappa_{\mathrm{mo}i}}{1-\kappa_{\mathrm{co}i}\kappa_{\mathrm{mo}i}\kappa_{\mathrm{v}i}}\kappa_{\mathrm{co}i}e^{-sy_i}e^{-s\tau_{\mathrm{m}i}} \\ &+\frac{1-\kappa_{\mathrm{mo}i}}{1-\kappa_{\mathrm{co}i}\kappa_{\mathrm{mo}i}\kappa_{\mathrm{v}i}}\kappa_{\mathrm{co}i}\kappa_{\mathrm{mo}i}e^{-s(y_i+x_i+z)} \end{aligned} \tag{E.6}$$

E.3.2 Deriving the model

We start with a model of the number of detected faults, i.e. the number of reruns that is necessary to get an accepted result

$$\Pr[\phi = k] = \begin{cases} \kappa_{\mathrm{mo}0}\kappa_{\mathrm{v}0} & , k = 0 \\ \kappa_{\mathrm{co}k}\kappa_{\mathrm{mo}k}\kappa_{\mathrm{v}k}(1 - \kappa_{\mathrm{mo}0}\kappa_{\mathrm{v}0}) & \\ \quad \times \prod_{i=1}^{k-1}(1 - \kappa_{\mathrm{co}i}\kappa_{\mathrm{mo}i}\kappa_{\mathrm{v}i}) & , k > 0 \end{cases} \tag{E.7}$$

If there are k reruns, where the time to fault detection for a failed run of the task on replica i takes the time X_i, the correction time for replica k is y_k, the runtime for replica k is x_k, and the test time is z, the total runtime of the task is given by

$$Y = X_0 + X_1 + \cdots + X_{k-1} + y_k + x_k + z \tag{E.8}$$

We can now create an expectation function for E^{-sY} with conditions to runtimes r_n, correction times c_n and acceptance test time d and the number of faults ϕ

$$\mathrm{E}[e^{-sY} | r_n = x_n, c_n = y_n, d = z, \phi = k] = e^{-s(X_0 + X_1 + \cdots + X_{k-1} + y_k + x_k + z)} \tag{E.9}$$

As the times in this expression are independent, and by using the time to detection derived previously, i.e. $\mathrm{E}[e^{-sX_i}] = \mathbf{A}_i(s)$, the expectation function can be rewritten as

$$\mathrm{E}[e^{-sY} | r_n = x_n, c_n = y_n, d = z, \phi = k] = \begin{cases} e^{-s(x_0 + z)} & , k = 0 \\ e^{-s(x_i + y_i + z)} \mathbf{A}_{\mathrm{nc}0}(s) \prod_{i=1}^{k-1} \mathbf{A}_i(s) & , k > 0 \end{cases} \tag{E.10}$$

Removing the condition on the number of faults is done by multiplying the probability of a given number of faults (from equation E.7) with the expectation function (from equation E.10) for that given number of faults, and summing the results

$$\mathrm{E}[e^{-sY} | r_n = x_n, c_n = y_n, d = z] = \sum_{0}^{\infty} \mathrm{E}[e^{-sY} | r_n = x_n, c_n = y_n, d = z, \phi = k] \Pr(\phi = k) \tag{E.11}$$

This yields the expectation function

$$\begin{aligned}
&\mathrm{E}[e^{-sY}|r_n = x_n, c_n = y_n, d = z] = \\
&\kappa_{\mathrm{mo}0}\kappa_{\mathrm{v}0}e^{-s(x_0+z)} \\
&+\sum_{k=1}^{\infty}\kappa_{\mathrm{co}k}\kappa_{\mathrm{mo}k}\kappa_{\mathrm{v}k}e^{-s(x_k+y_k+z)} \\
&((1-\kappa_{\mathrm{mo}0})e^{-s\tau_{\mathrm{m}0}} + (1-\kappa_{\mathrm{v}0})\kappa_{\mathrm{mo}0}e^{-s(x_0+z)}) \\
&\prod_{i=1}^{k-1}(1-\kappa_{\mathrm{co}i})e^{-s\tau_{ci}} + (1-\kappa_{\mathrm{mo}i})\kappa_{\mathrm{co}i}e^{-sy_i}e^{-s\tau_{\mathrm{m}i}} \\
&\quad + (1-\kappa_{\mathrm{v}i})\kappa_{\mathrm{co}i}\kappa_{\mathrm{mo}i}e^{-s(x_i+y_i+z)}
\end{aligned} \tag{E.12}$$

A problem with this equation "as it is" is that there is an infinite sum in it. This can be explained as no limit to the number of faults that can occur, and thus no limit to the number of reruns that is necessary to get an acceptable result. To work around this, we set a maximum to the number of faults, N, that can occur and still be tolerated by the system. If more than N faults occur, we consider the system to have failed. Even for a relatively low N, this is a reasonable approximation of the system's behaviour, as the probability of faults occuring is usually very low. Also, because of the extra time used to tolerate faults, we can assume that a task will miss its deadline, and thus fail anyway, if more than N reruns are needed. It is also possible to model a system where there cannot be more than N faults, by setting $\kappa_{\mathrm{co}N} = \kappa_{\mathrm{mo}N} = \kappa_{\mathrm{v}N} = 1$

The conditions to time use are removed by integrating the expression with respect to the distributions, e.g., removing the condition to the acceptance test time is done with the integral

$$\begin{aligned}
&\mathrm{E}[e^{-sY}|r_n = x_n, c_n = y_n] = \\
&\int_0^{\infty}\mathrm{E}[e^{-sY}|r_n = x_n, c_n = y_n, d = z]dD(t)
\end{aligned} \tag{E.13}$$

Limiting the number of faults before failure to N and removing the conditions on time use for all runtimes, correction times and test times, gives the

moment generating function

$$\begin{aligned}
&\mathbf{G}(s) = \\
&\kappa_{\mathrm{mo}0}\kappa_{\mathrm{v}0}\mathbf{M}_0(s)\mathbf{D}(s) \\
&+\sum_{k=1}^{N}\kappa_{\mathrm{co}k}\kappa_{\mathrm{mo}k}\kappa_{\mathrm{v}k}\mathbf{M}_k(s)\mathbf{C}_k(s)\mathbf{D}(s) \\
&((1-\kappa_{\mathrm{mo}0})e^{-s\tau_{\mathrm{m}0}} + (1-\kappa_{\mathrm{v}0})\kappa_{\mathrm{mo}0}\mathbf{M}_0(s)\mathbf{D}(s)) \\
&\prod_{i=1}^{k-1}(1-\kappa_{\mathrm{co}i})e^{-s\tau_{\mathrm{c}i}} + (1-\kappa_{\mathrm{mo}i})\kappa_{\mathrm{co}i}\mathbf{C}_i(s)e^{-s\tau_{\mathrm{m}i}} \\
&\quad + (1-\kappa_{\mathrm{v}i})\kappa_{\mathrm{co}i}\kappa_{\mathrm{mo}i}\mathbf{M}_i(s)\mathbf{C}_i(s)\mathbf{D}(s)
\end{aligned} \tag{E.14}$$

Equation E.14 describes the moment generating function of the runtime of a combined timeout and acceptance test system, as a function of the fault-free runtimes, the correction times, the timeouts and the probabilities that failures does not occur, and is the main result of this work.

E.4 Example of use

In this section, we will give some examples on how the models can be used.

For the examples a system consisting of one primary and two backups is used, i.e. equation E.14 is used with $N = 2$:

$$\begin{aligned}
&\mathbf{G}(s) = \\
&\kappa_{\mathrm{mo}0}\kappa_{\mathrm{v}0}\mathbf{M}_0(s)\mathbf{D}(s) \\
&+\kappa_{\mathrm{co}1}\kappa_{\mathrm{mo}1}\kappa_{\mathrm{v}1}\mathbf{M}_1(s)\mathbf{C}_1(s)\mathbf{D}(s) \\
&((1-\kappa_{\mathrm{mo}0})e^{-s\tau_{\mathrm{m}0}} + (1-\kappa_{\mathrm{v}0})\kappa_{\mathrm{mo}0}\mathbf{M}_0(s)\mathbf{D}(s)) \\
&+\kappa_{\mathrm{co}2}\kappa_{\mathrm{mo}2}\kappa_{\mathrm{v}2}\mathbf{M}_2(s)\mathbf{C}_2(s)\mathbf{D}(s) \\
&((1-\kappa_{\mathrm{mo}0})e^{-s\tau_{\mathrm{m}0}} + (1-\kappa_{\mathrm{v}0})\kappa_{\mathrm{mo}0}\mathbf{M}_0(s)\mathbf{D}(s)) \\
&((1-\kappa_{\mathrm{co}1})e^{-s\tau_{\mathrm{c}1}} + (1-\kappa_{\mathrm{mo}1})\kappa_{\mathrm{co}1}\mathbf{C}_1(s)e^{-s\tau_{\mathrm{c}1}} \\
&\quad + (1-\kappa_{\mathrm{v}1})\kappa_{\mathrm{co}1}\kappa_{\mathrm{mo}1}\mathbf{M}_1(s)\mathbf{C}_1(s)\mathbf{D}(s))
\end{aligned} \tag{E.15}$$

For comparison, systems with the same fault mechanisms, but without one of the detection mechanisms, and where an undetected leads to the failure of the system is used.

A system without the timeout mechanisms can be modelled as a system where an omission failure will lead to an infinite response time, and can be modeled by letting the timeout values in equation E.15 approach ∞ (i.e. let

$e^{-s\tau} = 0$):

$$\begin{aligned}&\mathbf{G}_{\text{noto}}(s) = \\ &\kappa_{\text{mo0}}\kappa_{\text{v0}}\mathbf{M}_0(s)\mathbf{D}(s) \\ &+ \kappa_{\text{co1}}\kappa_{\text{mo1}}\kappa_{\text{v1}}\mathbf{M}_1(s)\mathbf{C}_1(s)\mathbf{D}(s) \\ &(1-\kappa_{\text{v0}})\kappa_{\text{mo0}}\mathbf{M}_0(s)\mathbf{D}(s) \\ &+ \kappa_{\text{co2}}\kappa_{\text{mo2}}\kappa_{\text{v2}}\mathbf{M}_2(s)\mathbf{C}_2(s)\mathbf{D}(s) \\ &(1-\kappa_{\text{v0}})\kappa_{\text{mo0}}\mathbf{M}_0(s)\mathbf{D}(s) \\ &(1-\kappa_{\text{v1}})\kappa_{\text{co1}}\kappa_{\text{mo1}}\mathbf{M}_1(s)\mathbf{C}_1(s)\mathbf{D}(s)\end{aligned} \tag{E.16}$$

The timing distribution for a system without acceptance test can be derived in a way similar to equation E.14, where a timing failure leads to the failure of the system, here modeled as an infinite response time:

$$\begin{aligned}&\mathbf{G}(s) = \\ &\kappa_{\text{mo0}}\kappa_{\text{v0}}\mathbf{M}_0(s) \\ &+ \kappa_{\text{co1}}\kappa_{\text{mo1}}\kappa_{\text{v1}}\mathbf{M}_1(s)\mathbf{C}_1(s)(1-\kappa_{\text{mo0}})e^{-s\tau_{\text{m0}}} \\ &+ \kappa_{\text{co2}}\kappa_{\text{mo2}}\kappa_{\text{v2}}\mathbf{M}_2(s)\mathbf{C}_2(s)(1-\kappa_{\text{mo0}})e^{-s\tau_{\text{m0}}} \\ &(1-\kappa_{\text{co1}})e^{-s\tau_{\text{c1}}} + (1-\kappa_{\text{mo1}})\kappa_{\text{co1}}\mathbf{C}_1(s)e^{-s\tau_{\text{c1}}}\end{aligned} \tag{E.17}$$

In the examples MATLAB and Simulink are used to numerically calculate the example systems' cumulative distribution functions from the moment-generating functions.

E.4.1 A basic system

In the first example, a basic replication system is modelled, to show how the expression derived in the previous chapter can be used.

The following parameters are used:

The fault-free runtimes for both the primary and the backups are distributed with a triangular distribution with minimum time 6, maximum time 10 and mode 8:

$$m_0(t) = m_1(t) = m_2(t) = \begin{cases} 0 & ,0 \le t < 6 \\ \dfrac{t-6}{4} & ,6 \le t < 8 \\ \dfrac{10-t}{4} & ,8 \le t < 10 \\ 0 & ,t \ge 10 \end{cases} \tag{E.18}$$

$$\mathbf{M}_0(s) = \mathbf{M}_1(s) = \mathbf{M}_2(s) = \frac{e^{-6s} - 2e^{-8s} + e^{-10s}}{4s^2} \tag{E.19}$$

The times used for correction is distributed uniformly between 0 and 5 for all replicas:

$$c_1(t) = c_2(t) = \begin{cases} \frac{1}{5} & ,0 \leq t < 5 \\ 0 & ,t \geq 5 \end{cases} \tag{E.20}$$

$$\mathbf{C}_1(s) = \mathbf{C}_2(s) = \frac{1 - e^{-5s}}{5s} \tag{E.21}$$

The acceptance test time is 1:

$$d(t) = \delta(t - 1) \tag{E.22}$$

$$\mathbf{D}(s) = e^{-s} \tag{E.23}$$

Timeout values are set to 10 for execution of a task on a replica and 5 for the correction. The omission failure probability is set to 4×10^{-4} for execution and 2.5×10^{-4} for correction, and the value failure probability is set to 4×10^{-4}.

The resulting cumulative distribution function for this system, compared to systems with only timeout or acceptance test as a fault detection method, is shown in figure E.3. The figure shows that there is very little improvement in the reliability of this system from a non-tolerant system if there is a hard deadline before $t = 20$. This is to be expected, as a task where a fault occurs must be run at least twice. Also as expected, the systems with only one of the fault detection mechanism will not achieve a failure probability lower than the probability of an undetected fault.

Between $t = 20$ and $t = 26$, the probability of completed execution increases steadily. If the combined detection mechanism is able to detect all faults except deadline faults, and there is a hard deadline, the system as a whole will only fail if all there is a missed deadline or if all the replicas fail. The probability of failure is 7.8×10^{-7} if the deadline is at $t = 26$, which is a significant improvement from the 8×10^{-4} failure probability of a non-tolerant system. For a deadline at $t = 30$, the failure probability is 5.7×10^{-7}, and for a deadline at $t = 50$ the failure probability is 8.8×10^{-10}.

E.4.2 A system with checkpoints

In this example, a task that can be partitioned into subtasks is investigated. Checkpoints can be placed between the subtasks, at each checkpoint, fault detection is performed, and the system's state is saved. If a fault is detected, the

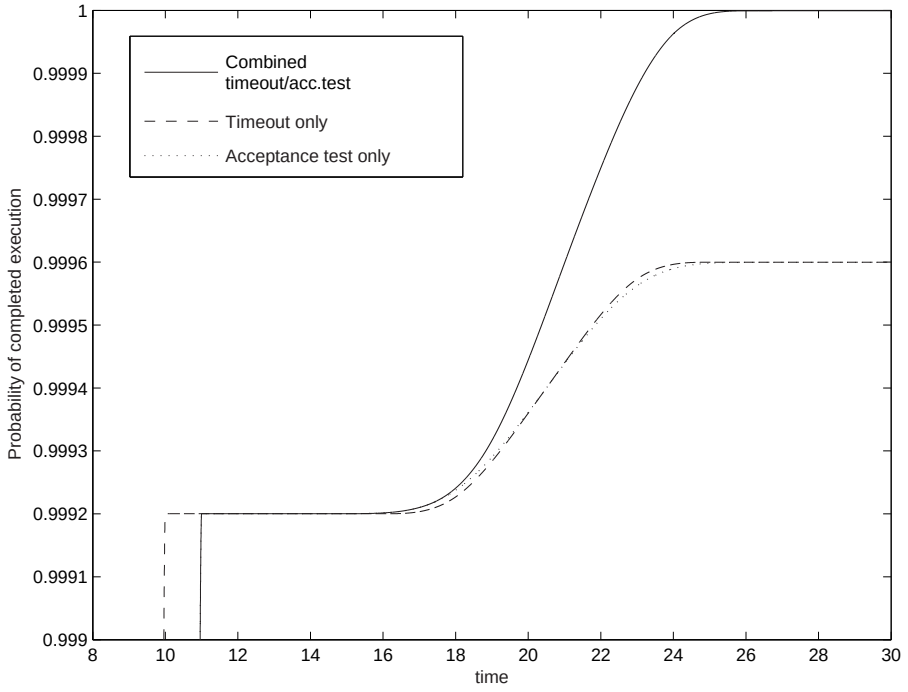


Figure E.3: Cumulative distribution functions for the system described in E.4.1, compared to systems with the same fault probabilities, but with only one of the fault detection methods.

execution restarts from the previous checkpoint. Using checkpoints generates much overhead, and several criteria and optimization methods for the number and placement of checkpoints exists [11, 25, 50]. In this example, we will show how the derived run-time distribution model can be used to optimize the number of checkpoints using the probability of deadline miss as a criteria.

The task consists of 12 subtasks, each with a fault-free runtime that is uniformly distributed between 1 and 2:

$$\mathbf{M}_{\text{sub}}(s) = \frac{e^{-s} - e^{-2s}}{s} \tag{E.24}$$

If the part between to checkpoints consists of n subtasks, the fault-free runtime of the part will be distributied as a convolution of the distribution of the

n subtasks, resulting in the moment-generating function

$$\mathbf{M}(s) = \mathbf{M}_{\text{sub}}(s)^n \tag{E.25}$$

The timeout value for a part is set to the maximum runtime of the part, i.e. if a part consists of n subtasks, the timeout is set to $2n$.

The time used to run an acceptance test, in addition to any overhead related to the checkpoint is set to 1 per part:

$$\mathbf{D}(s) = e^{-s} \tag{E.26}$$

The correction time is also set to 1 per part:

$$\mathbf{C}(s) = e^{-s} \tag{E.27}$$

For each subtask, the probability of an omission failure during normal execution or a value failure is both set to 10^{-4}, for a part of n subtasks, the probability of these failure types is $1 - (1 - 10^{-4})^n$. The probability of an omission failure during correction is also set to 10^{-4}.

As the execution goes back to the beginning of the part upon fault detection, each part can be viewed as a fault tolerant system with a timing distribution found by equation E.15. If a task consists of m parts, and each part i has the runtime distribution $\mathbf{G}_i(s)$, the total runtime distribution of the task becomes

$$\mathbf{G}(s) = \prod_{i=1}^{m} \mathbf{G}_i(s) \tag{E.28}$$

In this example, the task can partitioned in the following ways:

- 1 part of 12 subtasks (i.e. no partitioning of the task)
- 2 parts of 6 subtasks
- 3 parts of 4 subtasks
- 4 parts of 3 subtasks
- 6 parts of 2 subtasks
- 12 parts of 1 subtask

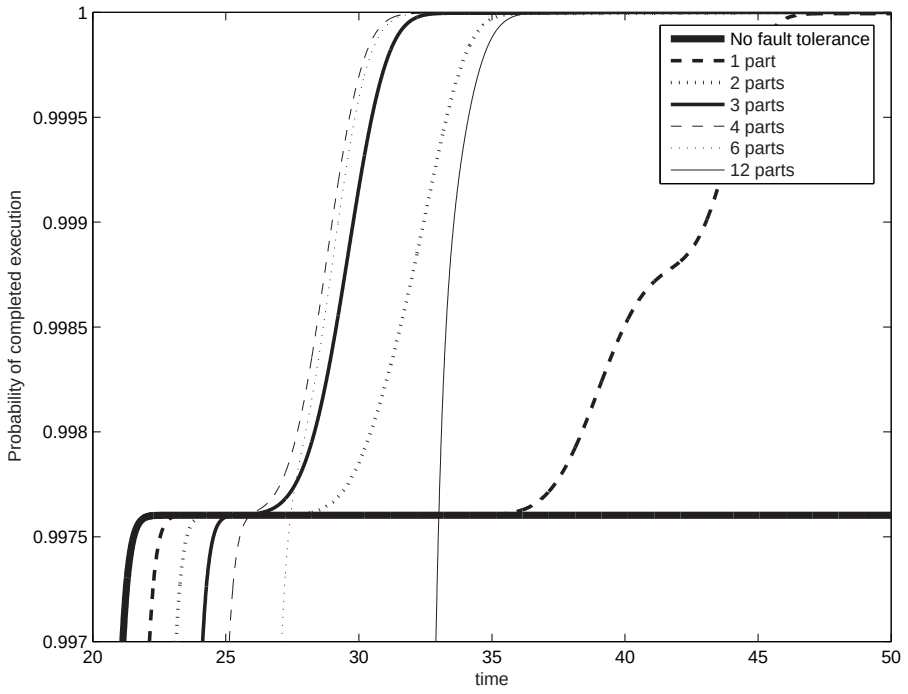


Figure E.4: Cumulative distribution functions for the systems described in E.4.2.

The resulting cumulative density functions are shown in figure E.4, with closer details in figure E.5.

If there is a hard deadline at t_{dl}, and the only possibilities of a system failure is a deadline miss or a single part failing more than 2 times, the results show that a nontolerant system performs better than or as good as the other systems if $t_{dl} < 25.4$, with a failure probability of 2.4×10^{-3}. The non-tolerant system does not have any overhead from the fault tolerance mechanisms, and there is a high probability that the other systems are not able to tolerate any fault occurences within this deadline.

For $25.4 < t_{dl} < 26.0$, the 3-part system has a failure probability that is slightly lower than the non-tolerant system.

For $26.0 < t_{dl} < 32.4$, the 4-part system has the lowest failure probability, and the improvement in the systems' reliability begins to show. For this system,

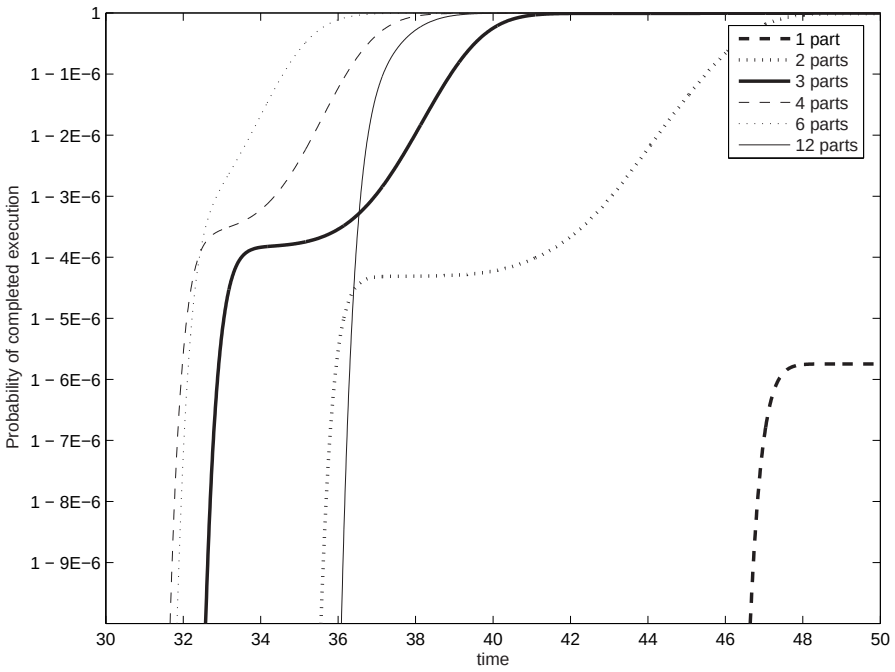


Figure E.5: Cumulative distribution functions for the systems described in E.4.2, details.

a deadline at 30 gives a failure probability of 3.1×10^{-4}, and for $t_{dl} = 32$, the failure probability is 5.6×10^{-6}.

The 6-part system has the lowest failure probability for $32.4 < t_{dl} < 40.9$, for $t_{dl} = 35$ the failure probability is 6.0×10^{-7}, and for $t_{dl} = 40$, the failure probability is down to 1.2×10^{-9}.

For deadilines higher than 40.9, the 12-part system has the lowest failure probability, below 10^{-9}.

E.5 Discussions and conclusion

In the model presented here, a very simple fault model is used, with constant probabilities for certain failure modes. This makes the models easier to derive and use, and it is possible to approximate some other fault models, like a poisson arrival fault process, to this model. Only replica failures that are detected by the timeout and acceptance test fault detection mechanisms are considered in the model. While the timeout will cover all silent and omission failures, there is a possibility that value failures pass the acceptance tests and lead to system failures. The equations can be expanded upon to also model this behavior.

In this paper, we have derived an expression for the run-time distributions of tasks in a fault tolerant system based on passive replication, where timeouts and acceptance tests are used as fault detection mechanisms. The expression is a function of the fault-free run-time distributions of the task on the replicas, the acceptance test time distribution, the distribution of the fault correction times, and the timeout values, as well as the probabilities of non-failure of the replicas. We believe that this can be a useful tool for the analysis of fault tolerant real-time system. The use of the model has been demonstrated on example systems, showing how the model can be used to determine the probability of missed deadlines, as well as how it can be used for optimization of checkpoint placement in a hard real-time system.

Bibliography

[1] Adevs: A discrete event system simulator. http://www.ece.arizona.edu/~nutaro/index.php.

[2] Algirdas Avižienis. The n-version approach to fault-tolerant software. *IEEE Transactions on Software Engineering*, SE-2(12), December 1985.

[3] Algirdas Avižienis, Jean-Claude Laprie, Brian Randell, and Carl Landwehr. Basic concepts and taxonomy of dependable computing. *IEEE Transactions on Dependable and Secure Computing*, 1(1):11–33, January–March 2004.

[4] Jens G. Balchen. *Reguleringsteknikk Bind 1.* Tapir Forlag, 1988.

[5] Jerry Banks, John S. Carson, II, and Barry L. Nelson. *Discrete-Event System Simulation.* Prentice Hall, second edition, 1996.

[6] Claude Berrou, Alain Glavieux, and Punya Thitimajshima. Near shannon limit error-correcting coding and decoding: Turbo-codes. In *Conference Record, IEEE International Conference on Communications*, volume 2, pages 1064–1070, Geneva, May 1993.

[7] A. Burns, S. Punnekkat, L. Strigini, and D.R. Wright. Probabilistic scheduling guarantees for fault-tolerant real-time systems. In *Proceedings of the 7th International Working Conference on Dependable Computing for Critical Applications.*, pages 339–356, San Jose, 1999.

[8] Alan Burns and Andy Wellings. *Real-Time Systems and Programming Languages: Ada 95, Real-Time Java and Real-Time POSIX.* Pearson Education Limited, third edition, 2001.

[9] Dave E. Eckhardt and Larry D. Lee. A theoretical basis for the analysis of multiversion software subject to coincident errors. *IEEE Transactions on Software Engineering*, SE-11(12):1511–1517, December 1985.

[10] Sachin Garg, Yennun Huang, Chandra M. R. Kintala, Kishor S. Trivedi, and Shalini Yajnik. Performance and reliability evalutation of passive replication schemes in application level fault tolerance. In *Digest of Papers of the Twenty-Ninth Annual International Symposium on Fault Tolerant Computing*, June 1999.

[11] Erol Gelenbe. On the optimum checkpoint interval. *Journal of the Association for Computing Machinery*, 26(2):259–270, April 1979.

[12] Sunondo Ghosh, Rami Melhem, and Daniel Mossé. Fault-tolerance through scheduling of aperiodic tasks in hard real-time multiprocessor systems. *IEEE Transactions on Parallel and Distributed Systems*, 8(3):272–284, March 1997.

[13] R. W. Hamming. Error detecting and error correcting codes. *The Bell System Technical Journal*, 26(2):147–160, April 1950.

[14] Ching-Chih Han, Kang G. Shin, and Jian Wu. A fault-tolerant scheduling algorithm for real-time periodic tasks with possible software faults. *IEEE Transactions on Computers*, 52(3), March 2003.

[15] Arnljot Høyland and Marvin Rausand. *System Reliability Theory.* John Wiley and Sons, 1994.

[16] Villy Bæk Iversen. *Data- og teletrafikteori.* Den Private Ingeniørfond, 1999.

[17] Hagbae Kim and Kang G. Shin. Evaluation of fault tolerance latency from real-time application's perspectives. *IEEE Transactions on Computers*, 49(1), January 2000.

[18] K. H. Kim. Designing fault tolerance capabilities into real-time distributed computer systems. In *Proceedings of the Workshop on the Future Trends of Distributed Computing Systems in the 1990s*, September 1988.

[19] K. H. Kim. Fair distribution of concerns in design and evaluation of fault-tolerant distributed computer systems. *Computer Communications*, 17(10):699–707, October 1994.

[20] K. H. Kim. Issues insufficiently resolved in century 20 in the fault tolerant distributed computing field. In *Proceedings of the 19th IEEE Symposium on Reliable Distributed Systems*, October 2000.

[21] Leonard Kleinrock. *Queuing Systems Vol 1: Theory*. John Wiley and Sons, 1975.

[22] Hermann Kopetz, Andreas Damm, Christian Koza, Marco Mulazzani, Wolfgang Schwabl, Christoph Senft, and Ralph Zainlinger. Distributed fault-tolerant real-time systems: The mars approach. *IEEE Micro*, 9(1):25–40, February 1989.

[23] Erwin Kreyszig. *Advanced Engineering Mathematics*. John Wiley & Sons, seventh edition, 1993.

[24] C. M. Krishna and Kang G. Shin. *Real-Time Systems*. McGraw-Hill, 1997.

[25] C.M. Krishna, Kang G. Shin, and Yann-Hang Lee. Optimization criteria for checkpoint placement. *Communications of the ACM*, 27(10):1008–1012, October 1984.

[26] Leslie Lamport, Robert Shostak, and Marshall Pease. The byzantine generals problem. *ACM Transactions on Programming Languages and Systems*, 4(3):382–401, July 1982.

[27] Jean-Claude Laprie and Karama Kanoun. Software reliability and system reliability. In Michael R. Lyu, editor, *Handbook of Software Reliability Engineering*, pages 27–69. McGraw-Hill, 1996.

[28] Jane W. S. Liu, Wei-Kuan Shih, Kwei-Jay Lin, Riccardo Bettati, and Jen-Yao Chung. Imprecise computations. *Proceedings of the IEEE*, 82(1), January 1994.

[29] Paul R. Lorczak, Alper K. Caglayan, and Dave E. Eckhardt. A theoretical investigation of generalized voters for redundant systems. In *Proceedings of the IEEE Fault-Tolerant Computing Symposium*, pages 444–451, Los Alamitos, 1989. IEEE.

[30] The MathWorks, Inc. *Getting Started with MATLAB 7*, 2007.

[31] The MathWorks, Inc. *Simulink 7 Reference*, 2007.

[32] Hein Meling. *Adaptive Middleware Support and Autonomous Fault Treatment: Architectural Design, Prototyping and Experimental Evaluation.* Dr.ing. thesis, NTNU, 2006.

[33] Meine J. P. van der Meulen and Miguel A. Revilla. Experiences with the design of a run-time check. In Janusz Górski, editor, *Computer Safety, Reliability, and Security. 25th International Conference, SAFECOMP 2006. Proceedings*, pages 302–315, Gdansk, September 2006. Springer.

[34] Object Management Group. *The Common Object Request Broker: Architecture and Specification, ver 2.6*, December 2001.

[35] Object Management Group. *The Common Object Request Broker: Core Specification, Version 3.0.3*, March 2004.

[36] M. Pease, R. Shostak, and L. Lamport. Reaching agreement in the presence of faults. *Journal of the Association for Computing Machinery*, 27(2):228–234, April 1980.

[37] Brian Randell. System structure for software fault tolerance. *IEEE Transactions on Software Engineering*, SE-1(2), June 1975.

[38] I. S. Reed and G. Solomon. Polynomial codes over certain finite fields. *Journal of the Society for Industrial and Applied Mathematics*, 8(2):300–304, June 1960.

[39] Daniel P. Siewiorek and Robert S. Swarz. *Reliable Computer Systems: Design and Evaluation.* Digital Press, second edition, 1992.

[40] Amund Skavhaug. *A Holistic Approach to Development of Dependable Industrial SCADA Systems. With Emphasis on Cost Effectiveness.* Dr.ing. thesis, NTNU, 1997.

[41] William Stallings. *Operating Systems: Internals and Design Principles.* Prentice Hall, third edition, 1998.

[42] William Stallings. *Data & Computer Communications.* Prentice-Hall, sixth edition, 2000.

[43] John A. Stankovic. Misconceptions about real-time computing: A serious problem for next-generation systems. *Computer*, October 1988.

[44] Andrew S. Tanenbaum. *Computer Networks*. Prentice-Hall, third edition, 1996.

[45] Åsmund Tjora and Amund Skavhaug. Fault tolerance methods in component-based real-time systems. In *Proceedings of the WIP-session held in connection with Euromicro Conference*, Dortmund, 2002.

[46] Åsmund Tjora and Amund Skavhaug. A general mathematical model for run-time distribution in a passively replicated fault tolerant system. In *Proceedings of the 15th Euromicro Conference on Real-Time Systems*, pages 295–301, Porto, July 2003.

[47] Åsmund Tjora and Amund Skavhaug. Assessing reliability of real-time distributed systems. In *Proceedings of the 1st ERCIM Workshop on Software-Intensive Dependable Embedded Systems*, pages 59–64, Porto, August 2005.

[48] Åsmund Tjora and Amund Skavhaug. Run-time distributions in passively replicated systems using timeout and acceptance fault detection. In *Proceedings of the ERCIM/DECOS Workshop on Dependable Embedded Systems*, Lübeck, August 2007. Proceedings not yet published.

[49] Åsmund Tjora, Amund Skavhaug, and Poul E. Heegaard. A mathematical model for run-time distributions in a fault tolerant system with nonhomogeneous passive replicas. In *Proceedings of the ERCIM/DECOS Workshop on Dependable Embedded Systems*, Gdansk, September 2006. Proceedings not yet published.

[50] John W. Young. A first order approximation to the optimum checkpoint interval. *Communications of the ACM*, 17(9):530–531, September 1974.

[51] W. Zhao, L. E. Moser, and P. M. Melliar-Smith. End-to-end latency of a fault-tolerant corba infrastructure. In *Proceedings of the Fifth IEEE International Symposium on Object-Oriented Real-Time Distributed Computing, 2002. (ISORC 2002)*, pages 189–198, 2002.

[52] Hengming Zou and Farnam Jahanian. A real-time primary-backup replication service. *IEEE Transactions on Parallel and Distributed Systems*, 10(8):533–548, June 1999.