CHALMERS
UNIVERSITY OF TECHNOLOGY

UNIVERSITY OF GOTHENBURG

Distributed Systems verification using fault injection approach

Master of Science Thesis in Software Engineering

Zhenxiao Hao
Khaled Alnawasreh

Department of Computer Science and Engineering
Chalmers University of Technology
University of Gothenburg
Gothenburg, Sweden 2016

Master's thesis 2016

Distributed Systems verification using fault injection approach

Zhenxiao Hao
Khaled Alnawasreh

CHALMERS
UNIVERSITY OF TECHNOLOGY

Department of Computer Science and Engineering
Division of Software Engineering
Chalmers University of Technology
University of Gothenburg
Gothenburg, Sweden 2016

Distributed Systems verification using fault injection approach

Khaled Alnawasreh
Zhenxiao Hao

Supervisor: Patrizio Pelliccione, Department of Computer Science and Engineering
Supervisor: Mårten Rånge, Ericsson
Examiner: Miroslaw Staron, Department of Computer Science and Engineering

Master's Thesis 2016
Department of Computer Science and Engineering
Division of Software Engineering
Chalmers University of Technology
University of Gothenburg
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Cover: A research of how to test distributed systems which consist of embedded components by injecting random faults

Typeset in LaTeX
Printed by [Chalmers University of Technology]
Gothenburg, Sweden 2016

Distributed Systems verification using fault injection approach
Zhenxiao Hao
Khaled Alnawasreh
Department of Computer Science and Engineering
Chalmers University of Technology

SUMMARY

Software nowadays becomes more complex and the number of the components that is involved in an application is externally large. If a fault occurs, the fault can easily propagate, become larger and take more time to detect and reproduce. Therefore, having a robust system that is able to perform normally even with the existence of faults is very important, but at the same time is very challenging. Different researches have been involved in handling and improving the robustness by using fault injection techniques presented in [23], [31]. Fault injection is mainly used in order to detect the unexpected faults as well as the dependencies bottleneck. Fault injection approaches work by sending fault messages to the components within a distributed system and observing how the system can handle them.

This study presents a fault injection approach for testing the robustness of the embedded distributed system in the RBS (Radio based station) at Ericsson. RBS is a distributed system that consists of components that communicate with each other via messages. One characteristic of the distributed system at Ericsson is the possibility to work and provide services even though some components fail. Since the components are stateful and have complex protocol, verifying that the system is robust is not a trivial task. The new approach is inspired from Netflix's ChaosMonkey. When Netflix moved their data center to amazon web service, they had the need to use fault injection technique for testing the reliability of the distributed system. After deep analysing of the Performance Management(PM) framework documentations at Ericsson, some potential bottlenecks have been discovered and some strategies on how the faults can be triggered have been implemented. A fault injection tool have been developed in this study for testing the robustness of the distributed system. Moreover, unexpected faults were detected after generating two fault types, which were sending random messages as well as delaying messages. This study illustrates the potential of utilizing fault injection approach that comes as a complementary to traditional software testing.

The report is written in English.

Keywords: Distributed Systems, Fault Tolerance, Fault Injection Testing, Embedded Systems.

Acknowledgements

We, the authors of this thesis would like to express our deepest thanks to the following people who guided us and gave us invaluable feedback during our work on the thesis.

Mårten Rånge, supervisor at Ericsson

Patrizio Pelliccione, supervisor at Chalmers

Miroslaw Staron, examiner at Chalmers

We would like also to thank Ericsson for giving us the valuable opportunity to conduct this research.

Khaled Alnawasreh & Zhenxiao Hao, Gothenburg, March 2015

"The best way to avoid failure is to fail constantly".
– Netflix Engineers [25].

Contents

1

Introduction

This chapter gives a brief introduction of fault injection and how it is currently used in the industry. Furthermore, the current problem that Ericsson has and the motivation of introducing a new fault injection technique are presented. This chapter ends with the goal of the thesis and the structure of the report.

1.1 Background

Due to the evolution of social media and mobile devices, massive network data is produced. Distributed systems are at the core of this evolution. The need for efficient manipulation and usage of network data is a major issue. In order to process network data in parallel and perform tasks efficiently, many computing nodes are connected, providing users with services characterized by fault tolerance, ease of use and speed [30].

The usage of the Internet is continuously growing in the industry due to the fact that more and more devices are connected, as shown in Figure 1.1. As a result, more faults are introduced to the systems, and the systems are more likely to be affected by those faults. For this reason, distributed systems need to be more resilient to different types of faults and combinations of them as part of the system's normal operating procedure. Combination of faults may have a major impact on the performance of the system, sometimes even leading to systems being temporarily out of service, which has dramatic effects of embedded distributed systems, because such systems are more fragile and safety critical [22]. Distributed systems tend to have more unexpected faults than other types of systems when used in reality [15].

Chaos Monkey, an open source project which adopts the fault injection technique, was developed by Netflix when they moved their data center to the cloud. The main function of Chaos Monkey is to inject random faults to the distributed system hosted on the cloud. While Chaos Monkey is used, fault injection is becoming a more popular technique for testing distributed systems, various researches on fault injection are springing out. At same time, other types of fault injection tools were also introduced, but most of them are used to test systems hosted on the cloud.

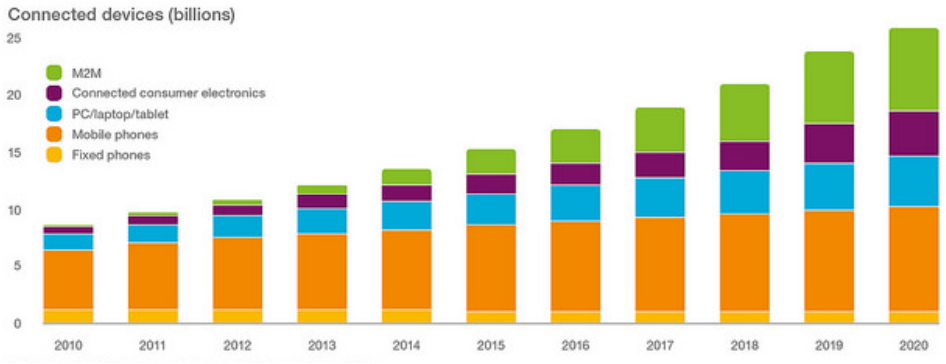

Figure 1.1: Ericsson Mobility Report June 2015 [14]

1.2 Problem description

RBS (Radio Base Station) is an integral part of the mobile data solutions delivered by Ericsson. RBS consists of embedded components that are connected together in order to support high network availability to wide area network. Mobile network is a distributed system that consists of many RBS. Due to the fact that operators as well as subscribers expect that network services support almost everything in their daily life, so high availability of network services is an essential part in daily life. As long as network signaling volumes increasing and the operators demanding more complex configuration for supporting more scenarios, the need for improving the resiliency is also increasing. The network should be flexible so that it responses to the challenge of the dramatic increasing in the network traffic and signaling created by the operators and subscribers [12].

The PM framework, one key component of RBS, is a framework used to count phone calls and network usages of subscribers. Same as other components of RBS, the PM framework is tested using the traditional approach like unit testing and functional testing (mock testing) [4], such software testing techniques are deterministic[8]. Unit testing will not catch all errors in the code, since it cannot evaluate every execution path of the program. Unit testing only tests the functionality of the units themselves, it will not catch integration errors or system-level errors [5]. Although functional testing combines different test cases and tests the code at the system level, such a combination is usually limited. In fact, functional and unit testing approaches are deterministic testing where random faults are very hard to detect. Therefore, there is a need of implementing a non deterministic approach, fault injection approach uses random mechanism for detecting the unexpected faults at the same time can be utilized in large scale systems. So fault injection approaches can be complementary to the existing testing techniques.

In order to do the testing in a non-deterministic way, the testing should be randomized; such a testing can be conducted with fault injection [26]. Some projects like Chaos Monkey already implement fault injection, but they were developed for testing the distributed systems hosted on the cloud. RBS is a type of distributed sys-

tem which consists of small embedded components. Previous approach like Chaos Monkey is not feasible for such an embedded distributed system. The new fault injection tool for such an embedded distributed system has new requirements; for instance the injection in the new fault technique is done near to the hardware layer, the programming language will be c++ instead of Java. On the other side, the new fault injection technique uses some shared libraries. Moreover, the communication paradigm between the components done using inter process communication.

1.3 Motivation

Services running in distributed systems are difficult to be tested using traditional methods [15]. When faults occur in running services, faults can exponentially propagate [24]. In this case, identifying faults in a running application in a production environment would go a long way. Testing in large scale distributed system is very hard due to the fact that the number of faults that can occur at any time is very large as well as it is very hard to imagine the scenarios that can accrue. Traditional testing techniques are not sufficient to predict the effects of faults on realistic applications running on large distributed systems [15].

As software for distributed systems becomes more complex, ensuring that a system meets its prescribed specification is a growing challenge that confronts software developers [11]. Due to the complexity of testing the services that are running on large scale distributed systems, verifying the robustness of distributed system is not trivial. Furthermore, the amount of faults that can occur is large. In particular, traditional testing will not provide a high confident testing level and it lacks of detecting unexpected faults. For this reason, traditional testing should be extended to run-time testing [17]. Run-time testing helps to detect the faults earlier, prevents faults from propagation and provides consistence tracing of faults. Testing at run-time can help to verify and validate that the system meets its requirements continuously.

The fault injection technique has been proposed as a possible way to address the fault tolerance challenges of distributed systems. This technique has been utilized for many years in order to verify the fault tolerance level of critical systems. Robustness of the system is important for Ericsson products and for embedded distributed systems. There is a need for Ericsson to have a fault injection tool in order to provide more robust products, the embedded industry is also calling for more research on utilizing fault injection on embedded distributed systems.

1.4 Goals

The main purpose of this study is to test the robustness and availability of the distributed systems. In this thesis we present a fault injection approach that provides non-deterministic testing method for testing the robustness of the distributed sys-

tems. The main idea is to check if the fault injection approach can be applicable at Ericsson embedded distributed system. This study is expected to benefits Ericsson who has an interest to test the robustness of their distributed system.

The thesis was developed in collaboration with a subset of the teams at Ericsson AB in Gothenburg, Sweden. Due to the complexity of distributed systems and the challenges of testing distributed systemds with traditional approaches, a new approach based on some existing distributed systems testing technique and researches is needed. The main investigation of this thesis is how to develop a new fault injection approach and implement a new fault injection tool to help the development team at Ericsson in testing their embedded distributed system. Moreover, we aim to make the fault injection approach as general as possible so it can be used in other companies and bring benefits to the embedded industry in general.

1.5 Report structure

In this section the structure of the report is specified.

Related Work
In the related work, we describe the previous related work. For each previous study we present the limitation, delimitation as well as the challenges that have been discovered on different literature study. Different concepts and terminology will be also presented for instance fault injection, failure as a service and the let it crash philosophy. Finally, we compare the previous related works with our thesis study.

Research Approach
In the research approach section, we describe the research methodology used in this thesis Furthermore, we list the design and creation process starting from the awareness of the problem and ending with final conclusions. Additionally, we presents the implementation of the suggestion and solution. Finally, data gathering and analyzing were also described in this section.

Results
In this section, the final results of running the fault injection approach are listed. Furthermore, from the extracted results an evident is drown on how well the fault injection approach work in this context. Finally, the tool is evaluated by comparing the results that we got with the previous techniques that Ericsson has.

Discussion
In this section the results and the benefits of this study are discussed. Bottleneck of the system architecture will be also discussed. Moreover, key improvements of the architecture will be suggested.

Conclusion and future work
In this section, we list the final conclusion. Moreover, we present the lesson

learned from this study. Finally, different aspects on how the work can be improved are suggested.

2
Related Work

This section describes the background of this thesis and some previous related work that aimed to provide fault injection approaches as well as implementations of fault injection tools.

2.1 Fault injection

A software system does not always behave as it is expected. The more distributed system grows the more it suffers from different dependability issues for instance in embedded flight software, the presence of faults affects its dependability and can even have a big impact on the entire mission. Fault avoidance, fault removal and fault tolerance techniques are used for achieving high dependability. For preventing and minimizing the faults, faults removal and tolerance should be adapted in all software development stages staring from the design and implementation and ending with the verification and validation[29]. Dependability is an important issue especially when the software is critical safety system, faults can propagate and have a consequent impact on the rest of the components. Furthermore service availability is also a very important aspect nowadays especially when the society becomes more Internet society [12]. Fault injection has been widely used in order to verify and validate the dependability and the availability of the software system. There are two types of fault injection, the first is hardware fault injection and the second is software fault injection [27].

In hardware fault injection the faults are injected at the physical level by controlling the parameters of the environment variables. In this case fault types can be like disturbing the power supply, voltage sags, heavy ion radiation and electromagnetic interference etc [27].

In software fault injection there are some techniques that inject the fault and this can be by implementing different fault types such as register and memory faults, missing messages, delayed messages, faulty messages and system overload. The faults can be inserted within the application, between the target and operating system or between the critical components in the system. Software fault injection techniques can be classified into two classifications: the first one is compile-time faults and the second one is run-time based on when and where the faults are injected. Faults can be triggered with different mechanism, for instance time-out, exception, code modification and sending different random messages [27].

2.2 Chaos Monkey Testing

CMT has been developed when Netflix moved their data center to Amazon Web Services (AWS). The main reason of developing CMT is to assess how potential failures in AWS would affect their ability to produce continuous services. Netflix has a big challenge to maintain a continuous testing to their service reliability when they move their data center to AWS, in order to evaluate how potential failures in AWS would affect their ability to provide continuous services. They introduce failures to the system and stress it in order to investigate how the system acts, where does it fail, when it fails and under which circumstances [15].

This research paper has shows that the more types of fault injected, the more errors discovered. the fault injection has been shown well preperation on how the algorithms works. Faults can easily propagate in large scale distributed system, especially on a system that has high dependencies between its components.

CMT works by sending fault commands to the components that are hosted in the cloud. On receiving the commands, the instance of the cloud will fail itself. By introducing faults CMT enables Netflix to discover the bottlenecks and weaknesses in their systems. Moreover, this helps them to strength the weak areas that are critical in their system. As a result of their study, they have discovered that CMT helps to detect different failure scenarios. CMT helps to find unexpected failure that can not be detected using the traditional methods. consequently, CMT helps in building a robust system that is resilient against failures.

CMT software design is flexible and can be used for other cloud service providers. However, it is just applicable for testing the availability and robustness for the running services in the cloud. Additionally, CMT suffers from the limitation in terms of computational power. In case of having a lot of test cases to run, it might even dominate the power consumption. Utilizing CMT in large scale systems will be costly and will consume extra resources such as network bandwidth, storage space and processing power[18]. When utilizing this approach, we should consider hardware reliability, memory managements as well as the running environment.

2.3 Let it Crash Philosophy

The research of Let-it-Crash Paradigm is presented in [33], this research has been conducted in order to assess the applicability for safety related software, check how error handling perform, as well as identify potential improvements for future work. Erlang was used as an implementation language for safety related functionality [33].

The main challenge in the Let it Crash paradigm study is to identify different software fault types in order to improve error handling in safety critical systems.

The evaluation of the LiC approach mainly depends on how well the system meets

those requirements and does the system acts as it is expected. The following assumptions acted as requirements for developing flexible software architecture that can manage different fault scenarios and at the same time for assessing the applicability of the LiC:

- Ensure the execution of critical functions.
- Prevent the unintended execution of a function.
- Define and monitor the conditions for carrying out a critical function.
- Ensure carrying out critical functions at a specific time and in a specific order.
- Unexpected failures either have no influence or result in a safe state.

Chaos Monkey Testing is not be applicable in traditional system with a small number of complex tasks or threads, since this can lead to complete failure within a short time. Chaos Monkey is more applicable to a large scale system that has the ability to perform even on the presence of failures. Instead, LiC only triggers the monitoring process, then fast replacement of the terminated software part will be performed.

This research can be efficiently utilized in safety related software development that have low number of complex tasks where Erlang is the implementation language, but it does not cover large scale embedded distributed system. Furthermore, this approach has a problem when it comes to time critical safety system, due to the fact that Erlang is missing the hard real time ability [33].

2.4 Failure Scenario as a Service (FSaaS)

A research of fault injection is described in [15], this research conducts a new model called Failure Scenario as a Service (FSaaS). The main purpose of this model is to test the resilience of the cloud application. This can be done by monitoring the result of generating different failure scenarios on the cloud. Their main research focus is to investigate the impact of the failure on the jobs running in Hadoop for analyzing the MapReduce jobs. Hadoop is an open source software framework written in the Java language. The main purpose of Hadoop is to process large distributed data set on huge computer clusters [2]. MapReduce is an efficient model for processing and generating large data set that is running on the distributed system on computer clusters [3].

A fault injection tool called AnarchyApe [1] was used to inject faults into Hadoop clusters. In order to evaluate the effects of individual fault and combination of faults, some sample fault scenarios and different types of workloads were performed on different types of Hadoop jobs. As a result of their study, they discovered that the resulting behavior of the distributed system depends much on the fault types, combination of faults, job types, and the time when the faults are injected. Furthermore, with traditional testing, it is very hard to test the reliability of running

services on the cloud. In large system, the number of faults that can happen are relatively large and therefore it is hard to test large scale system. Their study presents a new model that can deal with large cloud applications such Hadoop to be tested efficiently. The model had helped them to identify the week spots of the system and trying to fix them as well as monitor the resources for efficient utilization.

The limitation of the model is that it can be only used by cloud service providers and clients who rely on Hadoop. However, their goal is to develop a system that can be applicable to general cloud scenarios [15]. Indeed, this thesis aims to create an approach and develop a fault injection tool that has the ability to be applicable to small embedded distributed systems. Since embedded distributed systems are an essential part of most nowadays application, having an efficient tool for evaluating their robustness will be very beneficial and valuable.

2.5 Fault Injection and Mutation Testing with Model Based Development

A research of Increasing Efficiency of Verification and Validation by Combining Fault Injection and Mutation Testing with Model Based Development is presented in [28]. The main focus of this research is to proof that testing effectiveness and efficiency will be improved by applying fault injection techniques combined with a mutation testing approach for verifying and validating the automotive software at the model level. The main improvements of such testing is to detect the faults early in the development cycle thus providing enough time to be aware of the defects as well as less cost to fix them.

ISO 26262 standard stated the requirements for using the fault injection techniques. The study presented in [28], describes that those requirements are challenging due to the fact that they are related to complete functionality rather that components of sub-component of the software. Considering that the fault injection techniques are used in the automotive at one electric component (ECU) or one software system, and seldom at the function level. Therefore, utilizing fault injection technique is not enough since fault injection techniques are used late in the development (when ECUs are being developed), and any detection of faults at this stage will be difficult and costly.

In the same study [28], their research proposes a solution that the detection should be done at the model level when the ECUs functionality is still under design, in this case fault handling will be easy and cheap to resolve. Fault injection techniques are successfully used to detect the fault and identify the dependability problem of hardware and software when applied to complete system [19]. In order to increase the effectiveness of fault injection techniques and identify whether the fault should be injected at the model, software or ECU level, mutation testing should be utilized in order to verify the adequacy of the test cases. The combination of those approaches will improve the fault detection on an early design stage. A roadmap has been pro-

vided and described in [28]. The roadmap shows how a combination of fault injection and mutation testing approaches can be applied to modelling of automotive software for avoiding cost effects as well as increase the safety of modern and future cars.

The software development in the automotive industry and other similar industries widely adapted the paradigm of Model-Based Development (MBD) and the functionalities are safety critical functionality. MBD provides a significant functional testing in an early stage of the development process. Combination of fault injection and mutation testing approaches can be adapted to effectively verify and validate the functionality of the system. By detecting faults earlier, ability to perform much of verification and validation of the needed functionality and robustness the quality of the software will be exponentially improved. On the other hand, from this thesis we ensure that the cost for fixing the faults will be decreased when detecting faults in an early stage of the development cycle described in the discussion section "Quality of findings".

2.6 Improving Fault Injection in Automotive Model Based Development

Model based development (MBD) has been widely used in the automotive software development. MBD allows the verification and validation of the functionality and their dependencies in an early stage of the development cycle. Fault injection can be used for dependability evaluation at the model level for the hardware artifacts. However, the interdependencies between the system and its environment at the model level may affect the result reliability to be unrealistic. This research paper presents an approach in order to come over this problem that they observed. This problem makes it difficult to utilizing fault injection at an early stage of development. Fault the bypass modeling (FBM) framework has been presented and evaluated in [27]. The FBM framework can be used in order to observe the system behavior under fault injection modes. With this framework the system behavior will be monitored and analyzed in an early stage of the development cycle.

Analyzing system behavior in an early stage will reduce the amount of defects in late stage, improve the quality, as well as reduce the development time. Those advantages are very important especially when the system is safety critical system. Fault injection techniques have been proven to be an efficient technique for testing the robustness of the system. However, fault injection has a limitation when using it in a close loop model testing, the reason behind that is the interdependences between the system and its environment at the model level may affect the result to be unrealistic with unrealistic system behavior. Using the FBM will resolve this problem thus make it possible to use fault injection correctly in order to test the behavior model for dependability and robustness evaluation. The FBM helps an early defect detection which will not only save time and cost, but also will increase the quality of the development. Using the fault injection techniques at the model

level will allow discovering the dependability issues on an early stage. As a conclusion more robust system will be developed an evaluated continuously from the early stage [27].

3
Research Approach

This section presents the research methodology used in this thesis. First, the main purpose of the research and the research questions are presented. Furthermore, design research process including awareness of problem, suggestions and solutions, implementation, evaluation and conclusion is described. The section ends with list of validity threats that should be considered.

3.1 Research Purpose

The purpose of this thesis is to investigate how a fault injection approach can help development teams at Ericsson to verify and test the fault tolerance abilities of the distributed systems (the Radio Base Station). The main characteristic of such systems is that they consist of small embedded components, whose computation power is very limited. Based on the experiences of such a fault injection approach, another research purpose of this thesis is how this approach can be generalized so that it can be adapted by companies who have the need to test embedded distributed systems, like the systems on the vehicles or airplanes.

3.2 Research Question

This section presents the questions that this thesis aims to answer.

Main Question:

- Due to the limitations of traditional methods used by Ericsson for testing distributed systems, the high availability requirements of the products delivered by Ericsson, and the complexity of testing embedded distributed systems. The first research question is how to develop a fault injection approach for testing the robustness of the embedded distributed systems at Ericsson?
- As distributed systems is becoming more complex and the difficulty of testing the services that are running on large scale distributed systems, as well as the fault injection technique has been proposed as a possible way to address the fault tolerance challenges of distributed systems. The second research question is how the above approach can be utilized in other companies that have similar and common context?

3.3 Research Methodology

After a deep studying of different research methodologies together with our thesis context, the design and creation methodology was selected. Design and creation research was developed to address theoretical questions about the nature of learning in context. It is mainly used when a creation of new knowledge is seeking in order to solve a particular phenomenon through designing innovative artifact [10]. The phenomenon being investigated is mainly to verify the robustness of the RBS components at Ericsson as well as investigate the bottleneck of their distributed system. In order to address the problem that Ericsson has, a new artifact has been introduced and implemented using design and creation research methodology. After that, the new design was evaluated. Finally, a conclusion was carried out on how well the new model works and final results were presented. Rules and constraints were well identified for further use in studies with similar context.

The development process of this thesis was done in agile way. List of well specified requirements were identified with priorities. Work delivery was done in iterations. Considering that each iteration is three weeks. We started this thesis on the 19th of February, we used the first two weeks to get familiar with the PM framework. Each iteration took three working weeks, the first iteration was started on the 1st of March and ended on the 21th of March, the implementation was finished at the beginning of June. The work was done in four iterations. During the first iteration, we implemented the random messages sending, then we integrated the random messages sending into the automatic testing environment in the second iteration. Messages delaying was implemented during the third iteration and was integrated into the automatic testing environment in the last iteration.

Weekly meeting with the supervisor from Ericsson for delivering what has done, getting feedback as well as planning for next iteration. Getting continuous feedback from the supervisors helped us in gaining more information about the system, discussing the obstacles, getting some suggestion and solutions, at the same time validating the work continuously. The following sections present the stages of the design and research process including awareness of the problem, suggestion and solution, implementation, evacuation and lastly the final conclusion as shown in Figure 3.1.

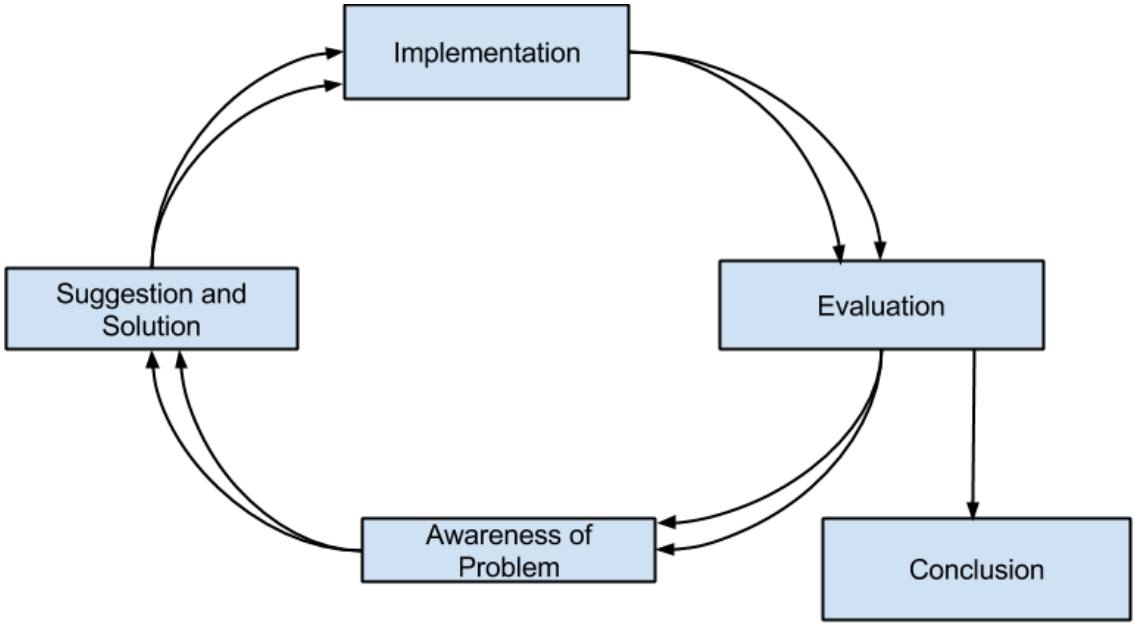

Figure 3.1: Design Research Process

3.3.1 Awareness of Problem

The first strategy used in this phase was collecting information about different fault types through deep literature studying. By studying the previous related work, we took an inspiration for developing the new approach. Furthermore, we had meetings with the supervisors at Ericsson in order to gather more information on how the system works as well as some suggestions about the injection. We also analyzed Ericsson PM framework documents for identifying the bottleneck of the system and determining the injections.

Additionally, Ericsson's internal fault reports and other available documents helped in gathering more information about the weakness of the system and identifying where problems can occur. After deep studying of system weakness together with possible faults which might happen in reality, random faults were designed. At the same time the system was experimented by running with different scenarios and inspecting the log files.

3.3.2 Suggestion and Solution

The main strategies that we used in this phase were weekly meeting with the supervisor and observation. During the weekly meeting, the supervisor presented the PM framework and some libraries it uses. At the same time we consulted the supervisor regarding our doubts. Some suggestions were also proposed though our meetings with the supervisor. Observation includes reading the documents, the source code and log files of the PM framework. Getting feedback from the supervisor helped us on finding reasonable solutions on the obstacles that we met. Tracking the be-

haviour of the system through the log files also helped us on finding the reason behind undesirable system behaviours.

Based on Ericsson fault report and designed fault tolerant ability of the PM framework, potential fault types were discussed with the supervisor. We decided to have two fault types, the first one was sending random messages and the second one was delaying messages. In sending random messages, we got the idea from the supervisor in order to test the filter ability of the PM framework against random invalid messages. Delaying the messages were performed between the critical components of the PM framework where faults can occur.

3.3.3 Implementation

This section describes the implementation of the fault injection tool. The implementation contains two steps: tool development and tool integration. The first step mainly concerns the work of development the fault injection tool using an Ericsson internal shared library called Inter Process Communication. The second step mainly concerns the work of integrating the developed tool into the automatic testing environment of the PM framework. Both fault development and fault integration are described in details in the implementation chapter.

3.3.4 Evaluation

Evaluation is a very important phase in the development process. As a matter of fact, evaluation gives an indication of the conformance level of how the tool works. In this study tool evaluation included the following:

- Weekly meeting with the supervisor for getting feedback.
- Well identified requirements for the expected conformance level on how the tool should work.
- Continuous validation of the requirements.
- Observing the results of running the fault injection tool through the log files.
- Comparing the observed results with the expected (how the system should perform in its normal state).
- If faults were detected, evaluating the faults by tracing back to the cause that made them.

The implementation of the fault injection tool was conducted in four iterations. During the first iteration, we implemented the random messages sending, then we integrated the random messages sending into the automatic testing environment in the second iteration. Messages delaying was implemented during the third iteration and was integrated into the automatic testing environment in the last iteration. Since we utilized agile development process, the fault injection approach was evaluated continually. Continuous evaluation of the approach helped us in developing

the work as needed, decrease the uncertainty level, and increase the work quality. The source code of each iteration was pushed to the remote repository where the supervisor had access to it. During the weekly meeting, we got feedback from the supervisor of the fault injection tool. Furthermore, we presented the fault injection tool after the last iteration to the development team and got some feedback from them.

The faults caused by the new fault injection tool were monitored through log files, by analyzing the behaviors of the system through the log files, the new fault injection approach was evaluated continually. Everything appears in the log files introduced by the fault injection tool was analyzed in order to evaluate the tool. An example of unwanted system behavior can be like system interrupts, messaging delaying, time-out and so on.

Model evaluation mainly depended on the results of running the fault injection tool. If the PM framework crashes after running the fault injection tool, the actual injection and the reason of the crashing will be evaluated first. If the random message sending triggers the PM framework crashing, the message types, contents, amounts and the frequency of the sending will be evaluated first. Based on the expected fault tolerant ability of the PM framework and the possible fault types which might exist in reality, some unnecessary provoking will be filtered out. For example, sending random message A with content B triggered the system crashing, but in reality, message A can never contain content B, then we just filter such a random message away. If sending message A with content C triggered some bugs in the PM framework, and in reality message A might contain content C, then we keep such a random message.

The fault injection approach works based on random manner, random selection can be both realistic and not in its nature. In order to get a reasonable indication of the approach evaluation, we made the input selection of the tool less randomized. By doing so, the approach is evaluated in a reasonable way and the results helped in discovering the unexpected faults. As a result, evaluating the detected faults helped us in evaluating the internal code as well as system architecture. After that, some suggestions were proposed on how internal code and architecture can be improved.

3.3.5 Conclusion

This section describes the conclusion of system behavior after each fault injection scenario. Based on the running results of each iteration and continuous feedback from the supervisor as well as the development team, conclusion was constructed on how well the approach performed. Final evaluation of the approach mainly depended on final results of all iterations together.

After the final iteration, results were listed for the main research questions. Furthermore, the robustness and availability of the embedded distributed system were tested using the new fault injection approach. Conclusion was built on how well the approach worked, how did the system behave after fault injection and what

are the keys of improvements for future work. As a conclusion of this study, we showed that fault injection technique can be applied to the embedded distributed system. Furthermore, unexpected faults were detected using this approach where non deterministic testing is applied. This approach also gave us well indication of the dependencies bottleneck where faults might occur, some keys of improvements were built from the observed results. Finally, this approach came as a complementary of the existing traditional testing approach.

3.4 Validity Threats

The validity of the study shows the trustworthiness of the final result and to which limit can the results be extended. Validity threats should not be biased by the researchers. Empirical research shows different thread to validity described in [16].This section shows the validity threats that were encountered during performing this study at the same time using different strategies in order to mitigate them. The threats were addressed not only at the final phase, but also during all the phases of the design research, in order to mitigate the risk of having unreliable results as much as possible [20].

3.4.1 Internal validity

Internal validity concerns about the causal relation between different factors when doing the testing. Factors can be the time that the faults were triggered, fault types, a combination of different fault types at different time, etc. Those factors have a causal relation in a way that they affect each others as well as the final result of testing [32].

Internal validity can also be how well the factors were considered when doing the test. The following questions should be considered when doing internal validity. Were all the factors considered? Could some factors impact on the reliability of the result? Were there any factors that should be triggered in order to make other factors work? List of internal validity including history, maturation and ambiguity about direction of causal influence are described as the following:

History: Doing the same test at the same object but on different time when running the fault injection tool on the real environment can have different results. For instance, when the testing happened on a normal day or when it happened on a holiday considering the number of events that can happen are varies from day to another. This could be due to the fact that the circumstances were not the same on both occasions. To mitigate this threat, the test should be repeated at different time to assure the result reliability. All test cases on this study were done on the simulator and not on the real environment. For this reason, there will be a risk of not having the same result when doing it on the real environment.

Maturation: This can be when the subject reacts differently over time. For instance, when the software scale over the time and the object relations become more complex, the behavior of the object could also change. When developing the fault injection tool, the XML messages name and message types were hard coded. Since part of the code were hard coded, does this have any effects when system scale. Ericsson distributed system can scale exponentially over time. Since we are doing the testing on small distributed system, being confident on that the testing technique will scale is very important. Having such scalable technique is a big challenge in software industry. Random testing can scale much better than other testing techniques on large systems and thus become more cost-effective on the long term [7]. Utilizing random testing technique in this case will mitigate maturation threat.

Ambiguity about direction of causal influence: Ambiguity about direction of causal influence is the difficulty to determine the causal relation between variables. In case of fault happened, it can be hard to verify the cause of it. When having a large and complex system identifying the causal relationship between the variable is not trivial task. For instance if A causes B or B causes A or even an external X causes A and B. When the fault injection implementation was integrated to the automatic testing environment, some of the injection which was ran manually triggered the crash of the automatic testing environment. This happened because of the strict dependencies in the automatic testing environment. The more complex the system is, the more ambiguous causal relation will be. Well understanding of the distributed system, the correlation between the components and dependencies will mitigate from this threat.

3.4.2 External Validity

This section points out how much can the result be generalized. There are a lot of things should be considered when it comes to generalizing the model. The external validity cares about to which extend the result can be generalized. The main concerns is how to make the result general as much as possible so it can work on other similar environmental setting. [32].

Interaction of selection and treatment: This case can be when it comes to generalizing, having subjective populations that are not representative of the population, for instance when selecting just programmers in an inspection experiments when programmers, testers, and system engineer should take part of the inspection. When doing the testing in this thesis different members were involved, including software developer, tester, software engineer, as well as supervisors from the academy in order to make the study objective as much as possible. However, when experimenting the test, the tester can insert some values into the variables on the configuration file such as number of messages, message types and time interval. By letting the user inserting those values, the result of the test can be biased.

Interaction of setting and treatments: When doing the testing on toy environment not on a real environment. In our case doing the testing on the simulator

instead of doing it on the real environments. Will the results be affected or changed?

3.4.3 Conclusion validity

This validity threat focus on the relationship between the treatment and the outcome. List of the conclusion validity is described in the following:

Fishing and the error rate: This study aims to detect at least single fault by fishing through random extermination. But on the other side this can not verify that fishing could generalize the reliability of the measurements.

Reliability of measures: By experimenting the test more than once, we assured the reliability of the results. We got the same result at each time we repeat the experiment.

3.4.4 Construct validity

Construct validity is focusing on how well the design constructed when doing the experiment (Testing). Moreover, this threat focuses more about does the result of the study covers the purpose as well as the research questions. In order to mitigate this threat the rules should be well identified at the same time the result should not be interpreted in a wrong way.

3.4.5 Reliability

The main focus of this section is to identify how reliable is the final result. Will the results be effected if the experiments repeated again and again at different time or environments. Furthermore, in case of conducting the study by other researchers, will they come up with the same results [20]. In order to mitigate from this threat we have listed the rules needed to conduct the study at different companies with similar context. Moreover, as a non-expert of Ericsson embedded distributed system, gray box testing is performed by randomizing different test cases at different times. In this case, predicting unexpected faults will increase. Random testing seems to be more unbiased and has higher faults predictability than other testing techniques [7]. Finally, by taking guidance and feedback from an external developers and researchers in both Ericsson and Chalmers, this will increase the unbiased judgment as well as the reliability of the results.

4

Implementation

This chapter describes the implementation of the fault injection tool. The implementation contains two steps: tool development and tool integration. The first step mainly concerns the work of development the fault injection tool using an Ericsson internal shared library called Inter Process Communication, the results of this step are some compiled C++ object files. The second step mainly concerns the work of integrating the developed tool into the automatic testing environment of the PM framework, where Shell Scripts and Ruby were used.

4.1 Inter Process Communication

Inter Process Communication(IPC) is used as a communication paradigm in some components of the PM framework. Collection of information from the system is done by handling the collection of the counters and events between its components [13]. IPC is used to send messages between mailboxes, processes and processors. A processor is an execution unit that is handled by one instance of an operating system, it can have multiple cores and several simultaneously executing contexts. A process has its own memory map and can have several running threads. A thread is a single execution context that can coexist with other threads within a process. We worked on a simulated environment, where the processor was the CPU of the computer, a process was a simulation of a processor on the embedded components, and a thread was a simulation of a process in the real operating system. Figure 4.1 shows the relationship between the real operating system and the simulated environment. In our implementation, only processes and threads were involved, they were identified by the process namespace and the thread name. Both the namespace and the thread name have human readable names.

IPC is a widely used library of RBS. The PM framework is built on top of IPC, it uses the APIs of IPC for the communication within the system. Based on this feature, we started our implementation without touching the code of the PM framework, but made use of IPC APIs. In this way, we were able to inject random faults into the PM framework from outside the framework, so that there was no extra computation for the framework to do in order to cater for fault injection. Which is contradict to CMT.

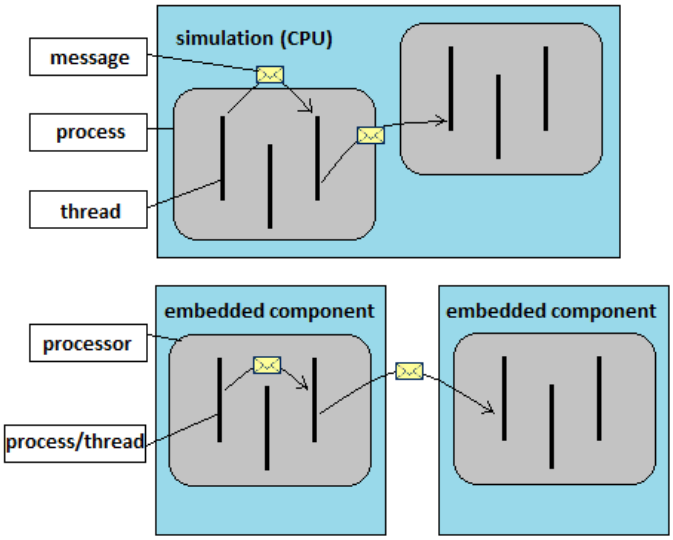

Figure 4.1: Relationship between the real operating system and the simulated environment

4.2 Automatic testing environment

In order to test the functions of the PM framework, a set of automated scenarios which test the PM use cases that operation in reality have been developed. The automatic testing environment is used to test the PM framework with such scenarios. The automated test scenarios will be used by the fault injection tool to verify the PM works even in the presents of faults.

The fault injection tool was integrated into the automatic testing environment using Shell Scripts, which injects random faults into the PM framework while running with the scenarios. The automatic testing environment also controls the operations of the the fault injection tool like code generation, compiling and linking. By integrating the tool into the automatic testing environment, the operation of the fault injection tool is more user-friendly and errors are more easily to be detected. The running results of fault injection are presented in the end of the automatic testing environment and traced in different log files. Figure 4.2 shows one type of running results in the end of the automatic testing environment, where the errors and corresponding log files are presented.

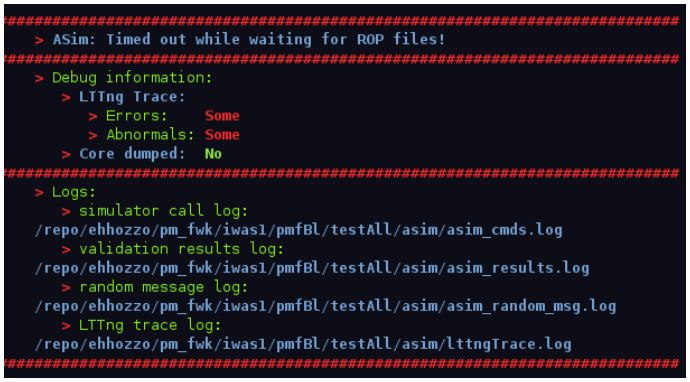

Figure 4.2: Running results of automatic testing environment

4.3 Sending random messages

In IPC, different messages were constructed with C++ structs, after initialization, the message objects were encapsulated in a message union and sent together. The fields of the messages are specified in some XML files shown in List 4.1. We first ignored those files and constructed the messages with random fields, this means that the messages might miss some necessary fields and might contain some extra unnecessary fields. After that we constructed the messages following the XML files but with randomly filled contents, for instance, making the field called messages size to be a random number instead of the real message struct.

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <signal name="CountersRcfm" osename="COUNTERS_RCFM" size="12">
3         <element name="sigNo" size="4" stereotype="signal_number" />
4 </signal>
```

Listing 4.1: XML that specified the messages fields

Constructing the message structs was based on the skeleton of the XML files, different categories of messages had different message fields and required different C++ header files. In order to implement the randomization mechanism, all the request messages structs and their corresponding required header files needed to be considered. The existing request messages might change in the future, to make the messages which were to be sent controllable, the message were specified in a configuration file. A code generator written in Ruby was used to create the message structs. By reading the configuration file and the XML file which specified the message fields, different message structs code were generated and compiled to different binaries.

After the random messages being constructed, the next step was to create the senders. The senders were actually some mailboxes located in the same namespace, which means that each mailbox has a separate thread and they all located in the same process. In order to send the messages to some targets, the mailbox address of the targets should be known. In the PM framework, the mailbox addresses can

be located by using combinations of the namespace names and the mailbox names. In our implementation, we put all the namespace names and mailbox names into vectors and tried to locate all possible combinations of them by putting them in a map as shown in List 4.2. So if there are n namespace names and m mailbox names, then we have m*n combinations in total. If a TID is successfully located, it will be saved in another vector which will be used as the target to send. The messages were sent in two different ways, synchronously and asynchronously. Sending the messages synchronously was implemented by joining the mailbox to the main thread.

```
1 for (auto && ns : namespaces)
2   for (auto && mb : mailboxes)
3     ivs.mboxes.insert(std::make_pair(ns, mb));
```

Listing 4.2: Code that creates all combinations of namespaces and mailboxes

Sending random messages was integrated into the automatic testing environment, which was written in Shell script. Every error triggered by the random messages was logged, the logging was implemented with a watch dog. The watch dog kept track of which message was sent and how many of it were sent. The watch dog also kept pinging all the previously located TIDs and if any of them dead during the time when the messages were being sent, the watch dog would also log such a TID. The log file provided a conclusion of all the relevant events when injecting random messages to the PM framework, which made the work easy to detect errors that were not found with unit testing.

4.4 Delaying messages

The message sending of the PM framework uses IPC by linking some dynamic libraries during run time. In C and C++, it is possible to wrap the dynamic library functions. Such a wrapping is implemented by creating shared libraries which override the functions of the original dynamic libraries. Such a dynamic library file can be linked by setting an environment variable called LD_PRELOAD.

We implemented the delaying messages by wrapping the IPC dynamic libraries which was responsible for sending messages. As shown in code 4.3, the name of original message sending function was called ___itc_send, the main parameters of this function were the message contents named "msg", the receiver named "to" and the sender named "from". The wrap function had exactly the same function name and parameters with the original function. At Line 13, a function called "dlsym" took a handle of the dynamic library which contains the ___itc_send function, this handle was returned to a function pointer called ___real_itc_send which had the same parameter types as the original ___itc_send function. By calling the ___real_itc_send function with the original parameters of ___itc_send, the original ___itc_send got called. But before calling ___real_itc_send, we let the current thread sleep for 1 second as shown at Line 14, this means that the ___itc_send function was always executed 1 second after being called, which implemented the delaying of message sending.

```
int __itc_send(
  union itc_msg **msg,
  itc_mbox_id_t to,
  itc_mbox_id_t from,
  const char *file,
  int line)
{
  int (*__real_itc_send)(
    union itc_msg **,
    itc_mbox_id_t,
    itc_mbox_id_t,
    const char *,
    int) = dlsym(RTLD_NEXT,"__itc_send");
  sleep(1);
  __real_itc_send(msg, to, from, file, line);
}
```

Listing 4.3: Code that wraps the sending function of IPC

After being compiled, the shared library can be loaded by setting the environment variable LD_PRELOAD to the path of this shared library. The nodes of the PM framework are started with an Erlang shell, which is different from a normal Linux shell. The environment variables are set in an XML file as shown in List 4.4 and 4.5. At Line 4 of List 4.4 the environment variable called LD_PRELOAD is set to the path of a shared library file called "wrap.so". List 4.5 is an XML configuration file of a node in the PM framework, Line 5 contains the path to the environment variable configuration file, the environment variable is set by adding this line into the node configuration file and restarting the node.

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<config >
  <env var="DEBUG" value="true"/>
  <env var="LD_PRELOAD" value="/PATH/TO/wrap.so"/>
</config>
```

Listing 4.4: XML for setting the environment variable

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<appdata target="appm">
   <loadmodule tag="dynamic" name="PMTESTAGGREGATOR" id="CXC1737528">
       <file type="i686" relpath="SOME_PATH" />
       <file type="config" relpath="PATH_TO/env_config.xml" />
   </loadmodule>
</appdata>
```

Listing 4.5: XML configuration file of a node in the PM framework

5
Results

This chapter introduces the results of performing fault injection with the fault injection tool. First, the steps of performing fault injection, the variables and the expected outcome are illustrated. Then, the real outcome, the performance of the PM framework and the detected bugs are introduced.

Design and creation research methodology was selected for creating the fault injection approach. Design and creation methodology is used for investigating a certain phenomena by creating and designing a new artifact. The phenomena in this study is to test the robustness of the embedded distributed systems and the artifact is the fault injection approach.

Agile development process was utilized for creating the fault injection approach. Fault injection tool developed in four iterations. Random sending messages was implemented in the first iteration, then we integrated the random messages sending into the automatic testing environment in the second iteration. Messages delaying was implemented during the third iteration and was integrated into the automatic testing environment in the last iteration.

In order to evaluate if the fault injection approach reached the conformance level against its requirements, the approach was evaluated at the end of each iteration. The evaluation was done by comparing the results of running the tool and the expected ones. At the end of the final iteration, conclusion was built on how well the approach worked. Final evaluation of the developed approach described at the result part in details, we could see that fault injection approach has detected unexpected fault that were not seen on the previous testing techniques. As matter of fact, we could also see that this approach is able to identified the bottleneck of the existing embedded distributed system.

5.1 Sending random messages

The communication within the system follows some specific protocols. Sending random messages is mainly developed to test the fault tolerance ability of the protocols. The messages are categorized into three types: request, confirmation and rejection. Based on the designed fault tolerance feature and the possible message types in reality, we only performed sending of random request messages. We focused on three aspects of the invalid messages, the origination, the contents and the amount of

sending.

The system is designed to tolerate invalid request messages, one main filtering standard of invalid messages is the origination of the messages. We focused on three aspects of the message origination, the namespace name, the mailbox name and combination of namespace and mailbox names. We first created multiple mailboxes which have the same namespace name, this means that the random messages are generated from different threads but the same process. Then we created mailboxes with same mailbox names but different namespace names, this means that the random messages are generated from both different threads and processes. Lastly, we created mailboxes with both different mailbox names and namespace names. Three cases of mailbox and namespace names are shown in Table 5.1.

	mailbox	namespace
case 1	S	D
case 2	D	S
case 3	D	D

Table 5.1: Combinations of mailbox and namespace names, S means same, D means different

The message contents inside the system varied dramatically, some contents describe the actual payload, some describe the protocol version, some specify the size of the messages. The fields of the messages were also different from each other. Based on the above characteristics, we constructed the random messages from two different aspects, random fields and random contents.

The amount of random messages would affect the filtering ability of the protocols. For instance, with sending one random message, the system is able to filter it, but with sending 1000 random messages, the system might only be able to filter 500 of them. With a huge amount of random messages, the CPU load of the systems would be high. In this case, would the system still be able to filter invalid messages? Would a high CPU load affect other parts of the system? To answer the above two questions, we sent random messages in two ways, synchronously and asynchronously.

Filtering of invalid messages is conducted by checking the properties of the messages. Such checking consumes time and processing power. So the filtering ability can also be affected by the frequency of the random messages. To test the filtering ability of the PM framework in terms of random message frequencies, the random messages are sent to the PM framework with different frequencies. This was implemented by giving different time intervals between each message sending. The number of invalid messages that need to be generated in order to let the system behaves differently were varies depending on the time interval between the messages as well as the time where the messages were triggered. Sending a lot of invalid messages without having any time interval between messages will lead the system to not behave as expected. This happened due to the fact that the computation power of the embedded com-

ponents is limited, the number of the messages are large and there is no enough time between message sending. Thus, the system will not be able to recognize if the message is valid or not.

Figure 5.1 shows the test chains of the PM framework. One sample test chain from the left to right can be read like this: test case1, with same fields but different contents, send the random messages synchronously, the time interval between each random message is 100ms. In order to perform the test randomly, we created different C++ classes to represent different variables in the chain, when the test starts, variable objects of the test chains are initialized and connected randomly. There are four variable classes, which are Cases, DataType(fields or contents), Concurrency(synchronously or asynchronously) and Intervals.

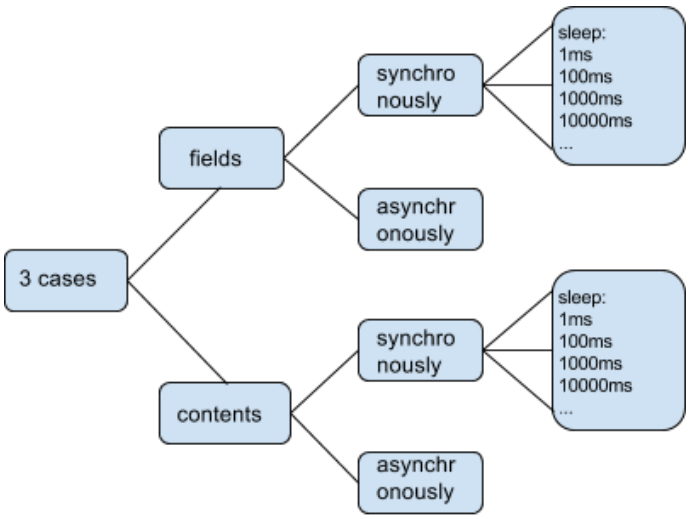

Figure 5.1: Testing chains

By performing the random message sending to the PM framework, we helped Ericsson tested the PM framework at two different aspects. The first aspect is the message types, which tested the PM framework at functional level. The second aspect is the message amount, which tested the PM framework at the performance level. The module test of the PM framework has covered almost all the invalid message sending, but the test is not random and the messages are sent one at a time. Our test sends random messages randomly and messages are also combined randomly, this helped the PM framework developers at Ericsson find many corner cases which the module tested does not cover. The random messages sending also plays a performance test role. By adjusting the number of random messages and the time interval between the random messages sent, a performance threshold of the PM framework was found, based on the demand in reality, this test helped the PM framework developers find a secure hardware requirements, which provides a optimal and economic solution for Ericsson.

Figure 5.2 shows a snapshot of sending random messages to one node in the PM framework. The response time of messages sent to this node should be less than 200

million seconds, if the there's no response after 200 million seconds, the sender will timeout. Figure 5.3 shows a snamshot of sending random messages synchronously with different time intervals between each sending which won't trigger timeout. Figure 5.2 shows that the node can maximally receive 1200 random messages at the same time. Figure 5.3 shows that in order to send 6000 messages, each message sending should have at least 80 million seconds interval. These snapshots have been saved as benchmarks for improving the performance of the nodes.

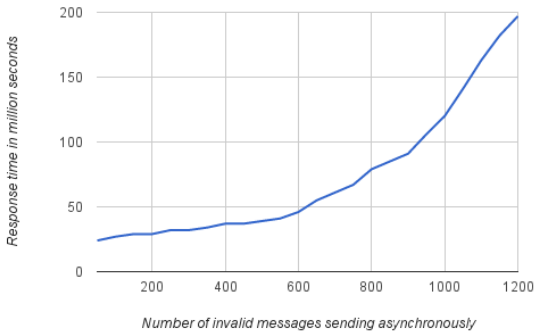

Figure 5.2: Sending invalid messages asynchronously

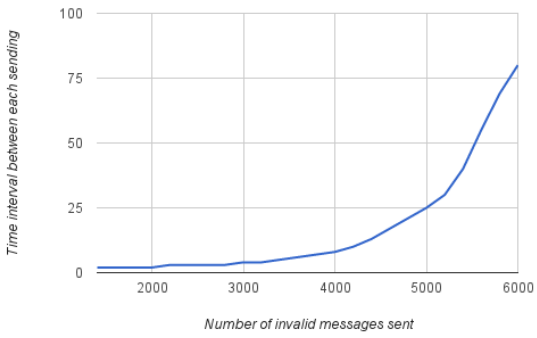

Figure 5.3: Sending invalid messages synchronously

5.2 Delaying messages

The second fault injection scenario was delaying the messages between the system components. The delay mechanism used default and randomize time interval between different components on the system in order to check if the system can handle different timeout intervals. There were multiple types of messages sending within the PM framework, for instance, request message, response message, rejection, confirmation message and messages describing the data. Some types of the messages

followed some time out mechanism strictly. For instance, when the connection request is sent, the node waits for the confirmation or rejection messages for a period of time, if neither a confirmation nor a rejection message is received, the node just times out the current request.

Some messages sent within the PM framework have relationships with each other, those messages usually work as a chain shown in Figure 5.4. In this case, node1 sent a request to node4 for some data, at the same time a clock started for the time out mechanism. In order to receive the data, some messages were sent to node2 and node3, but node1 only waited for the response from node4, delaying can happen on any of the messages sent between node1 and node4. In reality, such a chain can be really long and the delay of the messages sent between each nodes can also be very unpredictable.

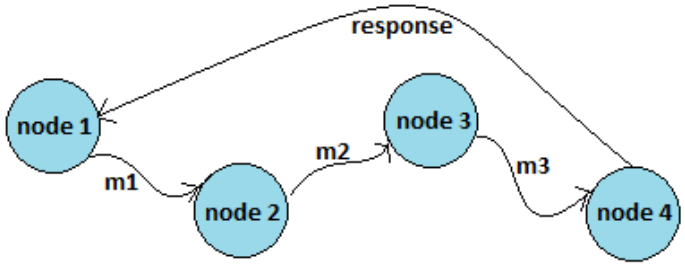

Figure 5.4: Message delaying chain

Delaying messages also tests the PM framework in performance level. By delaying messages in the PM framework, the network performance of the PM framework is tested. Currently the PM framework network performance test is conducted by adjusting the network bandwidth. This approach has two main drawbacks. First, the test has some hardware requirements, which makes the test more complicated. Second, adjusting the network bandwidth makes the test uncontrolled, because the delaying of the message caused by a low network bandwidth is very hard track. The current network performance test if more like a black box testing, what is really going on in the system during the test is not peered. Our test provides a white box testing for the PM framework network performance. During the message delaying, each message delaying is recorded in the log file, the PM framework developers at Ericsson can easily track the message delaying. A future network bandwidth benchmark has been created based on the experiences of the message delaying.

Figure 5.5 shows the time of delaying in million seconds and the corresponding network bandwidth in Mbit/s. The nodes uses wireless communication and Ericsson uses the international standard of Wireless 802.11n [6], whose normal bandwidth is 600 Mbit/s. This benchmark is used when the performance test by adjusting network bandwidth is performing abnormally, the corresponding delaying will be run to check which process is responding slowly.

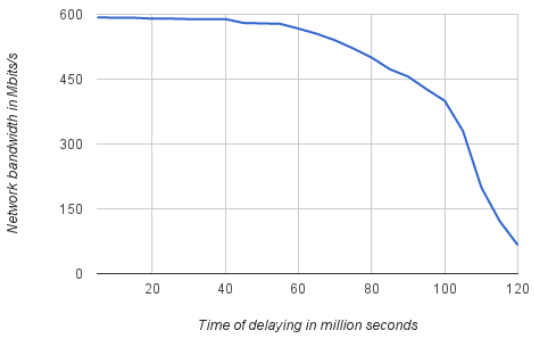

Figure 5.5: Message delaying and corresponding network bandwidth

5.3 Generalizability

The implementation of the fault injection tool was designed for the RBS at Ericsson. Two fault types have been implemented and integrated to the automatic testing environment. The first fault type is sending random messages and the second one is delaying the messages.

Fault injection techniques are widely used for testing the availability, robustness and the dependability of the software systems. Fault injection techniques are mainly used late on the development cycle after the software has been developed; there are some limitations when utilizing the fault injection in an early stage presented in [27]. To come over those limitations fault injection shall be combine with other testing like mutation testing for adapting it in an early stage of the development process [27].

The approach that has been developed in this thesis can be adapted late in the development cycle after the software has been implemented. Moreover, it is only applicable on distributes system that consists of small embedded components.

There are some constrains and requirements that should be considered when utilizing this approach in other context which are:

- The programing language should be C/C++ while the injections are performed on the top of the HW layer.
- Embedded distributed environment.
- Considering the computation power, since the computation power on the small embedded components are limited.
- The platform type that should be (Linux/Unix) environment.
- Some shared libraries that support some existing functionality such as send/receive signals.

- Process and thread communication paradigm.
- Wrapping the sharing library.
- There must be some existing test cases/suites.

The below figure 6.1 shows some general components that should be included in order to adapt this approach.

As a result of our thesis we have discovered that fault injection techniques can be adapted to the embedded distributed components. Keeping on mind some consideration when adapting this approach, when sending a lot of random messages without any time interval between the messages then the system will crash, due to the fact that the the computation power is limited on the embedded components. Moreover, the testing method used in the fault injection approach is non-deterministic provocation thus the testing can catch unexpected faults that are seldom happened. Finally, when utilizing fault injection approach the results of the provoking can be both realistic and not realistic and this can be due to the fact that the input of the test is totally randomized, in order to mitigate from this limitation we made our test cases less randomized such as having time interval, specify the messages type, number of messages and delaying time. By doing this the result of the fault injection tool was more reasonable and at the same time it gave a realistic indicate when evaluating the system.

6
Discussion

In this section the fault injection model is discussed. Furthermore, the challenges of adopting the fault injection approach into the PM framework are outlined. The section continues by discussing the quality of finding and what benefits could be gained from the study. Finally future work and some keys of improvements are suggested.

6.1 Fault injection model

Figure 6.1 presents the fault injection model, which consists of the target system (PM_fwk), fault injector, monitor (log files), IPC library as well as the controller that has the automatic testing and the configuration files.

After running the fault injection tool all data are monitored from the log files using stack trace, these data are collected and analyzed for the evaluation of the tool as well as for answering the main research questions. The user can run and control the tool through the controller, where the automatic testing environments and the configuration files are located. Through the automatic testing environment different testing techniques are listed. The fault injection tool supports random mechanism where the number of faulty messages, message delay and the time interval between sending the messages are randomized based on reasonable manner. On the other hand, the user can manually adjust the test cases through the configuration files.

The fault injection approach provides different fault types at different location and time. We have developed two fault types the first fault type is sending random messages and the second one is delaying the messages. Fault types such as sending invalid messages and delaying the messages were done by linking some dynamic libraries of IPC during run time. IPC library is a separate component, which consist of different system functionalities such as sending the messages between different system component. After deep study of the PM framework documentation and discussion with the supervisor, system targets were identified. The system targets were some critical components of the PM framework where they have strong dependencies. The faults were injected on the system target where there is strong coupling. Due to the fact that time factor can have an effect on running the experiments and to make sure that the result we got are reliable, we have repeated the experiments more than once at different time. For non deterministic testing as pointed in [21], as long as the number of the repetition of test sequence increases, the quality of testing

will also be increased. In actual testing this number is limited due to economical and practical reasons. However, when executing the testing large number of time, the probability that not all possible execution paths are executed will be close or even equal to zero.

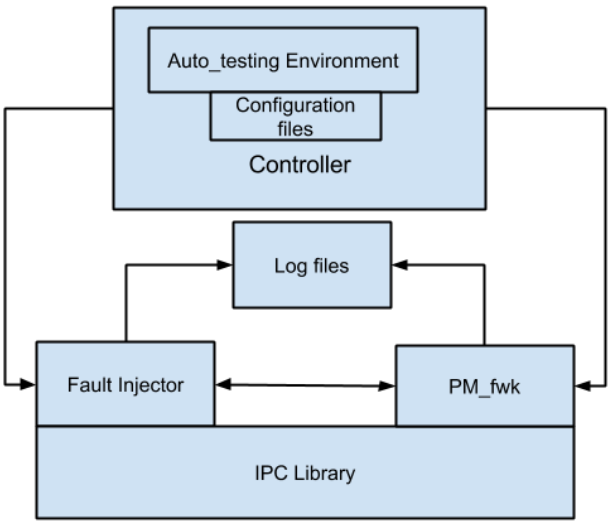

Figure 6.1: Fault Injection Model

6.2 Challenges in adopting the fault injection approach

During this study, we have met some challenges and barriers when adopting the fault injection approach. Challenges started from understanding the existing system, since the software at the RBS in Ericsson is huge and complex as well as the number of the connected components are relatively high. Furthermore, the idea of the fault injection is just available for cloud based application, but not to the embedded distributed system. So this study presents a new approach of fault injection that can be applicable to the embedded distributed system.

Another challenge was when we implemented the fault types, we implemented them from the scratch as well as imported some shared libraries in order to call some

existing functionalities from the lower level. We met some difficulties in loading the IPC libraries as well as setting up the environment variables. After the implementation of the fault types, integrating the fault injection tool with the automatic testing environments was not a trivial task due to the strict conditions on the automatic testing environments.

Under the fact that fault injection technique is non deterministic testing that follow a randomized manner, the result of running such technique can be misleading. In case of having a complete randomization of the test cases, then the results can be unreasonable, because of generating test case that can't happen in the reality. For instance, when we implemented the first fault type which was sending random messages, the system act always differently. That was due to the fact that the injection performed on the embedded distributed system where the computation power is limited, sending a lot of random messages without any time between them was misleading. We come over this problem by inserting sometime between each massage sending. As a conclusion, having the time between each sending gave us a reasonable and realistic indication.

Moreover, when we implemented the second fault type which was message delay, the system usually did not act as it should be, this was due to the fact that the time interval of delaying the messages was completely randomized. We come over this challenge by making it less randomized through identifying the default time and specifying the time interval. After that, the results were realistic and also gave us a reasonable indicate of approach evaluation. Finally, continuous testing of the new implemented fault injection tool was time consuming. There are some steps that should be performed such as running the simulation and restarting the Erlang nodes in order to get and observe the result of testing in the log files.

6.3 Quality of findings

The PM framework tested using the traditional testing methods such as unit and functional testing. In traditional testing, test cases might not cover all the needed cases on how the system should perform. Therefore, the results will mainly depend on how well the test cases are written and how well do they cover what should be covered. For this reason, traditional testing can lack of detecting the unexpected faults. Utilizing the fault injection approach for testing the PM framework at the RBS will complement the traditional testing and add some value, since the fault injection approach is used to detect the dependability issues as well as the unexpected faults based on a random manner. Moreover, as described in [21] deterministic testing usually covers smaller test suites with full fault coverage, on the other side, non deterministic testing covers larger test suites.

As a result of running the fault injection tool, unexpected faults were detected. Furthermore, the faults were identified and diagnosed through observing the stack trace on the log files. Diagnosing the faults helped us in identifying the bottleneck of the

distributed system. This has been shown after injecting the message delay fault type on the weak spots of the architecture that has strong coupling. After identifying the weak points, any improvement in the architecture will be easier to do. In fact, detecting and diagnosing the faults also helped in discovering more faults to inject. Detecting the unexpected faults and discovering the bottleneck of the software architecture at this stage will save money and time. Since the distributed system at Ericsson grows exponentially over the time, so detecting the faults as early as possible will be very beneficial. On the other side, the confidence level of testing and code quality is increased. Consequently, customer satisfaction will also be increased.

As conclusion, this study indicated that fault injection can be feasible and applicable for testing the robustness the dependability of the embedded distributed system. Furthermore, unexpected faults were detected using the fault injection approach. Moreover, this study can be taken as an inspiration on how to utilize fault injection techniques on embedded distributed systems in general.

6.4 Future work

This section shows some key improvements that can be considered for future work. This study focused on Ericsson environment. It would be interesting to involve more companies in the study for making more general approach to embedded distributed system, at the same time the result will be more reliable and general. In order to expand this approach and make it more general, the following suggestions can be taken into the consideration.

- Integrating this approach with another testing technique such as mutation testing.
- Injecting the system with more fault types.
- Make the same study on other companies and compare the result to see if it can be generic.

6.4.1 Integrating fault injection approach with mutation testing

Fault injection technique can be used in order to detect dependencies faults at the late stage of the development process. Utilizing fault injection approaches in an early stage is not the best practice. Therefore, combining fault injection techniques with mutation testing will improve the efficiency of detecting the faults at an early stage, thus reducing the cost as well as the time spend in detecting faults. Previous research has described the strength of combining the fault injection with mutation testing in model based development [28].

Combining fault injection with mutation testing can be very effective and efficient, since fault injection tests the dependencies between the components and mutation

testing tests the functionality of the system. Thus combining them will increase the confidence level of the internal and external code of the components as well as the consistencies between them. Moreover, the combined approach can be applied in an early stage and late stage of the development cycle. By doing this, the code will be validated continuously, the certainty level will be increased and bugs will be fixed in an early stage with lower cost.

6.4.2 Injecting the system with more fault types

Due to time constrains we just focused on the two most potential fault types which are sending random messages as well as messages delaying. However, more fault types can be injected such as register and memory faults, killing process, CPU overloads, slow down network, fork bomb at a particular node, and drop network packets for duration of period at a certain rate, etc. By injecting more fault types, the confidence level will be increased at the same time the total quality of the system will also be increased.

It would also be an area of interest to test a combination of faults. This can be by injecting a combination of faults in different order and at different time. In this case the probability of detecting faults will increase for the reason of a larger set of input are involved in testing. Furthermore, faults can act differently when injecting them in different order and different time [15]. Testing a combination of faults in a random based can detect even more unexpected faults. Moreover, this will increase the subset of the test suite that could be covered.

While fault injection is non deterministic testing that based on random manner, random testing can scale much more than deterministic testing [7]. Particularly, the Ericsson system scales fast and using random testing will be cost effective in the long run. In this case the maintainability cost will not be at risk in case of updating the test suite.

6.4.3 Improvements in the software architecture

Since the complexity of the software is dramatically increasing and systems are scaling over the time, the dependency level will also be increased. In the distributed system, there are always dependencies between the components, some dependencies are more critical than others on how the system should perform and this can be due to the fact that the amount of coupling between the components varies. In order to perform a particular action, the action should follow a specific sequence performed at different components that are connected with each other. Whenever a critical component has failed, the rest of the connected components can also fail.

After a deep study of the distributed system architecture at Ericsson, we identified some critical components that failure can be potential due to the strong dependencies. Message delaying for testing the timeout mechanism is a potential failure type in such dependencies. We delayed the messages between the critical components as

well as tracked the system behavior through the log files. Under those circumstances, we have discovered that the system will not act as it should be when delaying the messages at a specific time. In the long run, software architecture can be improved by having lower dependencies on those critical components. By having lower coupling, components will be easier to replace, reduce the chance that a change in one component causes a problem in the other components which enhances reusability, reduce the chance that a fault in one component cause failure in other components which enhances robustness [9].

7
Conclusion

This study aimed to verify the robustness of the distributed system at Ericsson. In particular, traditional testing techniques are used at Ericsson. However, traditional testing lacks of detecting the unexpected faults and dependency issues. We come over this problem by developing a new fault injection approach that performed based on non deterministic testing. Furthermore, this approach came as a complementary to the traditional testing. As can be seen, fault injection approach was developed and unexpected faults were detected. In the final analysis, we observed that fault injection can be adopted to the embedded distributed system and dependencies bottleneck can be identified. As has been noted, this approach helped the development team at Ericsson to utilize a better verification technique and reduced from the unpredicted faults that appeared during the operations on the customers sides. In the long run, using fault injection increases code quality, confidence level as well as reduces the cost. Academically, this thesis gave attention on the benefits of adopting such approach for fault tolerance ability of embedded distributed systems. In industry, generalizing this approach can help different companies that have a common interest in verifying the robustness of their embedded distributed system.

Bibliography

[1] Anarchyape. https://github.com/david78k/anarchyape/.

[2] Apache Hadoop. http://en.wikipedia.org/wiki/Apache_Hadoop.

[3] MapReduce. http://en.wikipedia.org/wiki/MapReduce.

[4] Mock Object. http://en.wikipedia.org/wiki/Mock_object.

[5] Unit testing. http://en.wikipedia.org/wiki/Unit_testing.

[6] Wireless 802.11n. https://en.wikipedia.org/wiki/IEEE_802.11n-2009.

[7] Arcuri, A., Iqbal, M. Z., and Briand, L. Random testing: Theoretical results and practical implications. *IEEE Transactions on Software Engineering 38*, 2 (2012; 2011), 258–277.

[8] Avritzer, A., and Larson, B. Load testing software using deterministic state testing. *ACM SIGSOFT Software Engineering Notes 18*, 3 (1993), 82–88.

[9] Bass, L., Clements, P., and Kazman, R. *Software architecture in practice*, 3. ed. Addison-Wesley, Upper Saddle River, N.J, 2012.

[10] Collins, A., Joseph, D., and Bielaczyc, K. Design research: Theoretical and methodological issues. *Journal of the Learning Sciences 13*, 1 (2004), 15–42.

[11] Dawson, S., Jahanian, F., Mitton, T., and Tung, T.-L. Testing of fault-tolerant and real-time distributed systems via protocol fault injection. In *Fault Tolerant Computing, 1996., Proceedings of Annual Symposium on* (1996), IEEE, pp. 404–414.

[12] Ericsson. High Availability is more than five nines. http://www.ericsson.com/real-performance/wp-content/uploads/sites/3/2014/07/high-avaialbility.pdf, 2014.

[13] Ericsson. PM FrameWork Documentation. https://vkia.rnd.ericsson.se/proj/crbs/hd_pm_fwk/doc/LATEST/introduction.html, 2014.

[14] Ericsson. Mobility Report. http://www.ericsson.com/res/docs/2015/ericsson-mobility-report-june-2015.pdf, 2015.

[15] Faghri, F., Bazarbayev, S., Overholt, M., Farivar, R., Campbell, R. H., and Sanders, W. H. Failure scenario as a service (fsaas) for hadoop clusters. In *Proceedings of the Workshop on Secure and Dependable Middleware for Cloud Monitoring and Management* (New York, NY, USA, 2012), SDMCMM '12, ACM, pp. 5:1–5:6.

[16] Feldt, R., and Magazinius, A. Validity threats in empirical software engineering research-an initial survey. In *SEKE* (2010), pp. 374–379.

[17] Fredericks, E., Ramirez, A., and Cheng, B. Towards run-time testing of dynamic adaptive systems. In *Software Engineering for Adaptive and Self-Managing Systems (SEAMS), 2013 ICSE Workshop on* (May 2013), pp. 169–174.

[18] Gunawi, H. S., Do, T., Hellerstein, J. M., Stoica, I., Borthakur, D., and Robbins, J. Failure as a service (faas): A cloud service for large-scale, online failure drills. *University of California, Berkeley, Berkeley 3* (2011).

[19] Hsueh, M.-C., Tsai, T., and Iyer, R. Fault injection techniques and tools. *Computer 30*, 4 (Apr 1997), 75–82.

[20] Kitchenham, B. A., Pfleeger, S. L., Pickard, L. M., Jones, P. W., Hoaglin, D. C., El Emam, K., and Rosenberg, J. Preliminary guidelines for empirical research in software engineering. *IEEE Transactions on Software Engineering 28*, 8 (2002), 721–734.

[21] Luo, G., von Bochmann, G., and Petrenko, A. Test selection based on communicating nondeterministic finite-state machines using a generalized wp-method. *IEEE Transactions on Software Engineering 20*, 2 (1994), 149–162.

[22] Malaiya, Y. K. Antirandom testing: getting the most out of black-box testing. pp. 86–95.

[23] Mariani, R., and Boschi, G. A system-level approach for embedded memory robustness. *Solid State Electronics 49*, 11 (2005), 1791–1798.

[24] Metzger, A., Schmieders, E., Sammodi, O., and Pohl, K. Verification and testing at run-time for online quality prediction. IEEE, pp. 49–50.

[25] Netflix. 5 Lessons Weve Learned Using AWS. `http://techblog.netflix.com/2010/12/5-lessons-weve-learned-using-aws.html`, December 2010.

[26] Ramachandran, P., Ramachandran, P., Kudva, P., Kudva, P., Kellington, J., Kellington, J., Schumann, J., Schumann, J., Sanda, P., and Sanda, P. Statistical fault injection. IEEE, pp. 122–127.

[27] Rana, R., Staron, M., Berger, C., Hansson, J., Nilsson, M., and Törner, F. Improving fault injection in automotive model based development using fault bypass modeling. In *GI-Jahrestagung* (2013), pp. 2577–2591.

[28] Rana, R., Staron, M., Berger, C., Hansson, J., Nilsson, M., and Törner, F. Increasing efficiency of iso 26262 verification and validation by combining fault injection and mutation testing with model based development. In *ICSOFT* (2013), pp. 251–257.

[29] Samuel, A. A., N, J., B, V., C.A, I., and Zachariah, J. P. Software fault injection testing of the embedded software of a satellite launch vehicle. *IEEE Potentials 32*, 5 (2013), 38–44.

[30] Tinetti Fernando, G. Distributed systems: principles and paradigms (2nd edition): Andrew s. tanenbaum, maarten van steen pearson education, inc., 2007 isbn: 0-13-239227-5. *Journal of Computer Science and Technology 11*, 2 (2011), 115–116.

[31] Vadlamudi, S. G., and Chakrabarti, P. P. Robustness analysis of embedded control systems with respect to signal perturbations: Finding minimal counterexamples using fault injection. *IEEE Transactions on Dependable and Secure Computing 11*, 1 (2014), 45–58.

[32] Wohlin, C., Ohlsson, M. C., Wesslén, A., Höst, M., Runeson, P., Regnell, B., of Computing, S., Högskola, B. T., of Technology, B. I., and för datavetenskap och kommunikation, S. *Experimentation in Software Engineering*. 2012.

[33] Woskowski, C., Trzeciecki, M., and Schwedes, F. Assessing the applicability of the let-it-crash paradigm for safety-related software development.