Institutionen för datavetenskap
Department of Computer and Information Science

Final thesis

Fault Diagnosis in Distributed Simulation Systems over Wide Area Networks using Active Probing

by

Filip Andersson

LIU-IDA/LITH-EX-A--15/028--SE

2016-02-02

Linköpings universitet
SE-581 83 Linköping, Sweden

Linköpings universitet
581 83 Linköping

Linköping University
Department of Computer and Information Science

Final Thesis

Fault Diagnosis in Distributed Simulation Systems over Wide Area Networks using Active Probing

by

Filip Andersson

LIU-IDA/LITH-EX-A--15/028--SE

2016-02-02

Supervisor: Johannes Schmidt, Åsa Falkenjack
Examiner: Ola Leifler

Abstract

The domain of distributed simulation is growing rapidly. This growth leads to larger and more complex supporting network architectures with high requirements on availability and reliability. For this purpose, efficient fault-monitoring is required. This work is an attempt to evaluate the viability of an Active probing approach in a distributed simulation system in a wide area network setting. In addition, some effort was directed towards building the probing-software with future extensions in mind. The Active probing approach was implemented and tested against certain performance requirements in a simulated environment. It was concluded that the approach is viable for detecting the health of the network components. However, additional research is required to draw a conclusion about the viability in more complicated scenarios that depend on more than the responsiveness of the nodes. The extensibility of the implemented software was evaluated with the QMOOD-metric and not deemed particularly extensible.

Keywords : Active Probing, Distributed Simulation, Fault localization, Master thesis, Extensibility

Acknowledgements

I would like to thank my examiner Ola Leifler as well as my two supervisors Johannes Schmidt and Åsa Falkenjack for all the great feedback and support during my thesis project. I am also grateful to Pitch Technologies and it's employees for having me and helping me to their best extent. Lastly, I would also like to thank my girlfriend for supporting me during the most stressful parts of the project.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

As modern technology is evolving at a very rapid pace, the networks required to support such technology are growing larger and more complex very fast. In any network, faults that can impact the stability and performance of the system are to be expected. These faults have to be managed and kept under control to ensure that the system can operate in a reliable way.

Fault detection and localization has been a hot topic over the last few decades. Many different approaches have been suggested and used. Recent years have seen a development towards enabling systems to autonomously detect and localize faults within themselves through the act of probing.

One growing field in technology that is very dependent on large and complex networks is the distributed simulation-domain, where the simulated scenarios can contain very high numbers of actors/entities being simulated together. One of the main issues is that different simulation environments developed by different vendors have to be able to communicate with one another to exchange simulation data. Fortunately there are standards available for this very purpose, the most popular one in recent years being the High Level Architecture (HLA). However, within the framework of HLA, there is not much support for fault management.

The fault detection and localization methods (mainly the probing-methods) have been shown to be beneficial for telecommunication and similar networks, this report will explore if these methods can also be used to improve the maintainability of a distributed simulation system using HLA.

1.1 Background

The available implemented solutions, as of today (no research in the HLA-domain has been found, therefore the implementation of Pitch Technologies will be used as the viewpoint), require great multitasking from the supervising party as the error logs are spread out in several different parts of the system. Hence, manually determining the root cause is not an easy task. Furthermore, in many cases, the information available in the error messages is not sufficient enough to pinpoint the root cause of several different errors. If there is no obvious root cause, the errors are usually handled by rebooting some part(s) of the system, hoping that it works as intended once the system is up and running again.

As an example, if a couple of software entities within the system are no longer responsive, the following issues are a few of the possible root-causes, and there might be no way of discerning the true cause by only looking at the errors:

- They all have unrelated issues that cause them to be unresponsive.
- Some unknown link that they all share might be down.
- If they all run on the same hardware, the hardware might be the cause.

Additionally there is currently no tool available that is able to showcase all errors in a single overview further hindering the mitigation work for the system overseer.

1.2 Motivation

Due to the complex nature of a large distributed simulation, the task of manually monitoring the health of the system is very time consuming. Especially in cases of fault-manifestation, where pinpointing and correcting the erroneous behaviour is required, preferably very quickly as to not delay the execution of the simulation. As the simulation scenarios grow, both in size and complexity, so does the effort required to keep the system healthy.

In multi-national catastrophe scenario exercises like the VIKING-series [1] even small faults that stall the progression of the scenario are extremely wasteful in regards to both time and economy. It is therefore vital to develop a faster and more efficient way of diagnosing faults in the system.

[1] VIKING14 was conducted in Enköping, Sweden 2014 with 2500 participants from over 50 countries.

1.3 Purpose

The thesis purpose is to explore the viability of automated fault detection and localization through Active probing [1] in regards to improving the maintainability of HLA-based systems. As part of the purpose a software implementation of Active probing in HLA-based systems will be created to be used for practical evaluation. It would be of interest to investigate the possibility of diagnosing more advanced faults than unavailable nodes. However, as it currently stands, there is very limited support for such experimentation in the target system. Instead efforts will be made to investigate how to build the software such that it is better prepared for future extensions, the extensibility of the final product will also be evaluated.

1.4 Research Questions

This report will attempt to answer the following questions:

- With regards to real world timing and bandwidth requirements, would an Active probing solution be viable in the target system and systems alike?
- What measures should be taken during the design step to ensure the extensibility of the system and how extensible is the final product?

The requirements will be presented and discussed in detail in chapter 3.

1.5 Delimitations

This thesis work will only focus on the detection and localization of faults, hence, no efforts will be spent on how to mitigate/repair the discovered faults.

An assumption about single fault per link-path is made. This means that the probing system will not be able to detect faults that are masked by other faults (i.e., if there is a problem with both the hardware and the software on one node, only the hardware fault will be discovered).

Halfway through the thesis project, the research questions were altered slightly. Most importantly, the question regarding extensibility was refocused to apply to a more general case of extending the software. How this change affected the appropriateness of the decisions made throughout the project will be brought up in chapter 5.

1.6 Report Structure

The report consist of the following six chapters:

1. **Introduction:** Introduces the topic of the thesis with the general topic of discussion, background, scope, research questions to be answered etc.
2. **Theory:** Describes the theoretical ground on which the thesis stands, explaining concepts used in the thesis work.
3. **Method:** Showcasing how the thesis work has been carried out, what directions were taken and why.
4. **Results:** Displays the performance of the implementation.
5. **Discussion:** Relates the results to the background, theory and research questions to discuss the success of the thesis work.
6. **Conclusion:** Gives the final conclusion of the report, ie. the answers to the research questions with support in the work that has been conducted.

1.7 Terminology

A short explanation of some of the most important concepts.

Probe : A test transaction that depends on one or more components in the network. A successful probe returns 0 (indicating that all dependencies are OK), failure in any of the dependencies returns 1.

Fault detection : The process of detecting the presence of a fault.

Fault localization : The process of pinpointing the exact source of the error that was detected during fault detection.

Node : A component in the network that is connected to the simulation environment (causing it to be a probing target).

Chapter 2

Theory

In this chapter, the theoretic basis for the thesis project conducted will be explained. This is to give the reader an understanding of the concepts used, making the subsequent chapters more understandable.

2.1 Fault diagnosis

As faults are ever present in the context of networking, fault diagnosis is of critical importance in a modern network. Unfortunately, most faults are not directly observable, but have to be inferred from gathered information either through alarms, probes or other means of information gathering. This is why the area of fault diagnosis has received plenty of attention in the research-domain during the last decades. Fault diagnosis has been defined as the three following steps by Steinder & Sethi [2]:

- Fault detection, discovering the existence of a fault.
- Fault localization, determining exactly what the fault is.
- Testing, verifying the output of the localization.

Fault localization is often described as the most difficult and time consuming part of the diagnosis, this due to the detection only needing information that something is amiss and testing already having a specific target. Whereas localization requires searching sometimes large areas of the network for a possible source for the error. This difference in complexity between detection and localization is especially evident in the area of Active probing [3], which will be discussed in a later section.

2.2 Probing

Before being able to diagnose the system health, sufficient information about the state of the system is required. This information can be gathered either passively or actively [4].

Passive information gathering is done by instrumenting system components to emit a message (or alarm) when its status/state change. The information in the messages can then be used in a correlation-process (event correlation [3]) to localize the fault, or as a basis to collect more information in a more active manner. This approach suffers from the requirement of heavy instrumentation of the components such that they are able to send out the appropriate messages for the right status-change [3]. Additionally, there is no way of ensuring that messages do get sent if a component is down [3]. Finally, some components may be proprietary and act as a "black box" in the network, removing the ability to receive event-data from all the system-components [3]. A positive aspect of the passive approach is that it does not generate high amounts of traffic, and therefore does not affect the performance of the network in any major way [5].

In contrast, the active approach consists of asking the components for information that is considered interesting. In many cases, the active approach is called "probing", where a probe is a test transaction carried out in the system, and the outcome of the transaction is dependant on one or more components [3]. Due to the end-to-end nature of the probes, they can be used to make very specific end-to-end measurements, enabling very flexible options on what to test for [6]. The main drawback of the probing-approach is its invasive nature; they can in some cases (caused by their own presence) disturb the very measurement they were intended to take [5].

In the most commonly used probing method, preplanned probing, probes are selected off-line in such a way that the probes together can diagnose any fault in the system. These probes are sent out at regular intervals (regardless if there are any errors present or not), the probe results can then be used to localize any possible faults through inference methods. Naturally, in a large and complex system, such a complete set of probes is often very large [7]. This raises an issue, with a large set of probes circulating in the network at regular intervals, performance can be negatively affected by the excessive probe-traffic [5]. This very reason is why recent research efforts have been focused towards Active probing [1], which aims to minimize the number of probes used for fault diagnosis.

2.2.1 Active probing

Active probing (also called Adaptive probing), introduced by the research team of Brodie et al. [1] and further researched by Natu & Sethi [6] and Lu et al. [8], utilizes probes in a more effective way than its predecessor, preplanned probing.

It does this through the distinction between detection probes and localization probes. Detection probes are selected as the minimal set of probes that cover the entire network, and can therefore detect **if** there is a fault present somewhere. When a fault is discovered, probes of finer granularity are selected to localize the error (only the regions corresponding to the failed detection probe need to be considered for the localization effort).

In most active probing solutions [1, 3, 8], some form of Bayesian belief network is used to support the decision-making as to what is the most informative probe, which should be sent next as well as what is the most probable fault.

Active probing often greatly decrease the amount of probes required for an accurate diagnosis compared to preplanned probing due to the significant reduction in probes that are issued at regular time intervals (all probes in preplanned probing vs. "a few" detection probes in Active probing). In comparison, the Active probing approach needs to do more on-line work when selecting which probe to send next [8] while the preplanned approach often has to do heavy inference work.

Different strategies of how to approach the localization step to further decrease the amount of probes used is discussed by Natu & Sethi [6], where they compare Max-, Min- and Binary-search versions for the probe selection step.

2.2.2 Hybrid solutions

Some researches have suggested a hybrid solution to fault reasoning by combining the active and preplanned probing [9]. This allows for a tradeoff between the issues in active probing (slow at reaching a conclusion) and preplanned probing (generates a large network overhead). The approach utilizes active probing to detect faults and narrow down the set of suspected root causes, then switch to preplanned probing once the set of possible faults goes below a predefined threshold.

Another hybrid solution is Active Integrated fault Reasoning (AIR) [10] that combines passive data collection for fault detection and enlists the help of the Active probing approach when passive data collection is not enough

to reach a definitive conclusion. AIR aims to improve the performance of fault localization while minimizing its intrusiveness [10].

2.3 Knowledge Inference

Whether preplanned or active probing is used, the probe results need to be analysed and the knowledge gained need to have some form of representation that can be used as a basis for further probing or to make a conclusion of the system state.

2.3.1 Bayesian belief network

Bayesian belief networks are, as mentioned in the previous section, the most commonly used knowledge-representation format (in Active probing). An example can be seen in fig 2.1. The reason for this is mainly that the Bayesian network provides insurance of choosing the optimal path as long as the probe with the highest conditional probability is chosen in each step [3]. The conditional probability of each node is calculated by combining known initial relational probabilities with recent probe results to generate the posterior probabilities. This is a process that is done iteratively with every new probe result until a certain threshold is reached, indicating that the state represented by this node is the current state of the system. In any iteration, the current conditional probability depends on all the previous probe results.

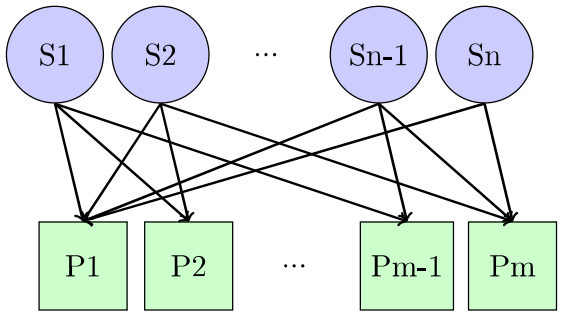

Figure 2.1: Bayesian network. (P=probes and S=states)

However, a major drawback is that the computational complexity of Bayesian inference is NP-hard [11].

2.3.2 Binary decision-tree

A binary decision-tree (similar to a flowchart), as seen in fig 2.2, is a different way to describe the knowledge acquired through the activity of probing. It consists of a binary tree with a question (probe) at each intersection and a conclusion (state) represented in each of the leaves. As probe-results are received, the tree is descended [1] until a leaf is reached, this leaf represents the current state of the system [12]. The binary decision-tree representation is somewhat closer to a preplanned approach than one using a Bayesian network, due to the predetermined nature of the tree.

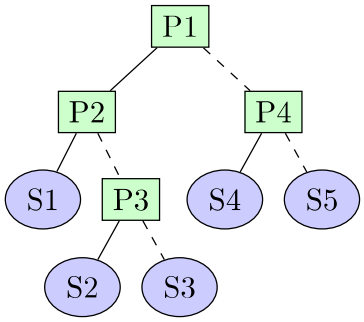

Figure 2.2: Binary desicion-tree. (P=probes and S=states)

The main advantage of decision-trees is that they are simple to implement and have a very low complexity, making the execution of the tree run in constant time [12]. Disadvantages are the inflexible structure, making maintenance of the tree cumbersome, as well as the inability to ask questions in a different order, which may in some cases be of interest.

2.4 High Level Architecture

High Level Architecture (HLA) is an interoperability standard for distributed simulation systems. It is a way to describe how data should be exchanged and who exchanges this data. An HLA-simulation consists of several different important concepts:

Runtime Infrastructure (RTI): This is the cornerstone of HLA. In short, the RTI is the interface to the world of HLA. Through the RTI a joined federate can send and receive standardized information (defined in a

[1] Taking solid paths on successful probe-results and dashed paths if probes fail.

Federation Object Model (FOM)). This information exchange is conducted through a Publish-Subscribe model on a service bus, see fig 2.3.

Federate: A federate is a component connected to the RTI and through it to a federation.

Federation: A federation is a collection of federates that are interconnected and exchange simulation data in accordance with a FOM.

Federation Object Model (FOM): A document describing how the data exchange in the federation should be described. It has been referred to as *"the language or the federation"* [13].

Federation Execution: A running session of the federation. Ex: A strategic training exercise scenario for military executives running several different military simulators for friendly and hostile tanks and aircraft. In this case, the Federation Execution would be the actual run-through of the scenario.

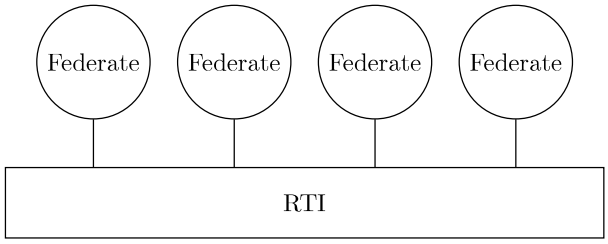

Figure 2.3: The service bus of HLA/RTI

There are a number of services supplied by the RTI, these are categorized into several different groups [14]). The group that is the most interesting in this case is the Management Object Model (MOM), which is a collection of inspection and management services for the federation. Through these services an observing federate can request reports about the other federates, making the services perfect probe candidates.

2.5 Pitch Systems

The system in which the experimentation part will be conducted is developed by the company at which the thesis project is done, Pitch Technologies. It is

set up with several different parts and an example can be seen in figure 2.4. In short, several sub-networks (often times over a single Local Area Network (LAN)) are connected through a number of ”gateways” (implemented as a software called PitchBooster). When large scale simulation exercises are conducted, different simulation sites (located at different geographical positions) connect behind these gateways to join the network that is supporting the exercise.

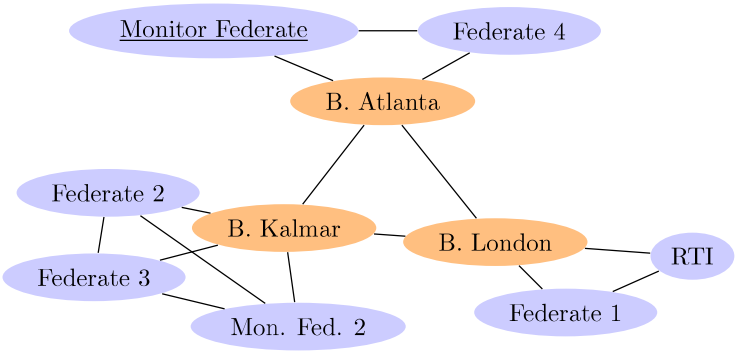

Figure 2.4: Example simulation network for a federation.

2.6 Extensibility in design

There are different levels that are interesting when speaking about software quality in general; System level, Technology level and finally the Unit level [15]. However, in this thesis the focus lies only on the unit level.

Extensibility, in the context of this thesis work, is the ability of a software to be adapted to changes to its specification that might or might not have been anticipated [16]. This should be possible with minimal or no changes to the existing internal structure and data-flow of the software. Extensibility is sometimes equated or linked to the concept of maintainability [16, 17].

Why is extensibility an important aspect of software design? Zenger [16] mentions the following reasons in his doctoral thesis:

- Software can be seen as a living organism that is constantly evolving with the help of the maintainers.
- Not all software ends up being used as it is intended when it is developed, it is therefore good if the software can be somewhat adapted to new areas of use.

- It is common for software product lines to heavily depend on earlier products in the line, creating new products that are based on the old ones. It is therefore very important that they are easy to extend.

In practice however, even though extensibility is a large boost for software reuse, software is generally not developed with extensibility as a main concern [16]. This due to:

- Extensible software is more complex and therefore harder to develop, test and deploy.
- As an additional effect of the previous point, it is more time-consuming and therefore more expensive (initially) to develop extensible software.
- It can affect performance negatively.

Despite the negative aspects presented, in practice the trade-off between performance and extensibility is most often a minor issue as it turns out that the parts of the software where extensibility is most beneficial are often not critical to the performance of the software as a whole. Likewise with the increased time/economical aspect of developing extensible software, if the software is known to be used for extension in the future, more often than not, it is cheaper to develop the initial software with a focus on extensibility such that it requires less of an effort to further develop it when the time comes.

A classification of different types of extensibility has been proposed, based in how invasive the extension work is in relation to the original software [16]:

White-box: White-box extensibility is split into two categories; Open-box and glass-box, where the first requires full access to the source code as changes are put directly into the original code, while in the former the source code is fully available but may not be modified.
Open-box in its unrestricted base-form can be rather unsafe as when code is changed by someone other than the original developer, that person might not be familiar enough with the core and introduce bugs or even break the software. It does, however, give the opportunity for real open-source software, which in some cases can be very beneficial where someone can copy the original software and change it to release an own, but slightly different version (ex. some of the Unix-systems, like Linux).
Glass-box is clearly a safer alternative, due to the separation of the

extension and the original software. This also makes it easier to understand and maintain the extensions since they are only that, an extension with a clear purpose. However, the freedom of modification is stripped away compared to open-box.

Gray-box: A middle ground between white-box and black-box extensibility, where the source code is not fully available, instead a more abstract documentation in the form of a system specialization interface is given. The interface specifies which abstractions can be extended and how they interact with the original software.

Black-box: This is the most restrictive form of extensibility, where only an interface specification is available. This is most commonly used in the case of proprietary software where the manufacturer wishes to hide all implementation details. Due to the restrictive format, all types of future extensions need to be anticipated when developing the original software for them to be available to extend. On the plus side, these limitations make black-box extensive software the easiest type of software to extend.

2.6.1 Achieving extensibility

Creating extensible software is often done by conforming with design principles and to some extent by applying software design patterns [16, 17]. Examples of design principles identified that support extensibility [17]:

- **Dependency Inversion Principle**
  Proposed by Martin [18] - *"Depend upon Abstractions. Do not depend upon concretions"*. Depending upon abstractions (interfaces) instead of concrete classes increases the extensibility of the software by enabling the swapping of a faulty or bad concrete implementation for a better one without having to modify the parts that depend on said abstraction.

- **Interface Segregation Principle**
  Another principle proposed by Martin [18] - *"Many client specific interfaces are better than one general purpose interface"*. When more than once client class use the same service (containing class specific functionality), that service should be represented by different interfaces for different class types, so as to only give the necessary access

to the client classes. This principle promotes extensibility by generating code that is easier to maintain and modify thanks to increased readability (less method-bloat when calling the service).

- **"Don't repeat yourself"**
  By having only one unique representation of a certain piece of information in the code (i.e. no repeats) there is no need to keep track of all the places some information is expressed. Naturally, this means that when something needs to be changed, it only needs to be changed in one place, witch reduces the risk of introducing new bugs and cause ripple effects that might end in the software crashing unexpectedly.

- **Law of Demeter**
  Presented by Lieberherr et al. [19], this principle states that software entities only should talk to their immediate neighbors (this is informally known as "only one dot per invocation"). A common example of this in practice is that when you wish to move forward, you don't tell your leg-joints to perform the specific angle-movements required to walk. By staying true to the Law of Demeter, the design will be tightly connected with the flow of data, causing identification of which parts of the system should be responsible for which tasks easier as well as simplifying identification of faulty design elements during implementation [17].

- **Modularity**
  Defined as: *"The property of a system that has been decomposed into a set of cohesive and loosely coupled modules"* by Booch [20]. Meaning that on a class-level, one class should have one clear purpose and only provide services that are closely related to this specific purpose. For example, a GUI widget that presents data, writes and reads data from a database as well as invoking remote services. This widget is filled with services that are useful together, but not very related [20]. The principle was referenced by Johansson and Lövgren [17], they argue that if a system is highly modular, it is easy to replace a component with another one, that might serve a slightly different purpose, and therefore the system is more extensible thanks to the property of modularity. Another good aspect of modularity is that it also increases the scalability of the software [17].

Even though the research-community is still somewhat divided on how well software design patterns affect the quality of software, there is support

for the claim that they help developers communicate better (increasing the readability of the code) [21], which according to the Consortium for IT Software Quality (CISQ) relates closely to maintainability (extensibility) [15]. Zenger [16] also supports the use of software design patterns to increase extensibility in software, often those that are derived from the *AbstractFactory* pattern [22].

Chapter 3

Method

In this chapter, the approach taken in this thesis project will be explained and motivated.

3.1 Prestudy

The project started by gathering information regarding the subjects relevant to the thesis by conducting a literature study. To compile an extensive base of academic literature well established databases [1] were searched for works on fault detection, fault localization, previously tried approaches as well as different types of knowledge representation and extensibility. Several different approaches were considered, but as the knowledge gained in the literature study was put into the perspective of the company software, most of them had to be dismissed and the direction and scope of the thesis was defined. Details on this is found in chapter 1 and chapter 2.

3.2 Requirements

The following three aspects of performance are considered important when judging whether the solution would be deemed viable in a real scenario [2] and they must therefore be taken into account during the design of the software.

[1] UniSearch, IEEE Xplore, Google Scholar etc.

[2] According to discussions with the systems administrators at Pitch, who are most familiar with the limitations of the systems that would benefit from a new solution to fault localization in a real scenario.

3.2.1 Localization time

A simulation execution being delayed is not only a bad user experience, it can (as mentioned in chapter 1) also be very costly. Therefore, it is important that this time is kept low.

3.2.2 Bandwidth

In some cases, the bandwidth can be limited, especially at remote simulation-sites. This limitation can be as low as 10 Mbps or even lower [23, 24]. For the simulation to work as intended, the diagnostic software needs to have a low enough bandwidth usage, such that it does not impact the simulation execution in a significantly negative way.

3.2.3 Correctness

Naturally, it is important that the results reported by the software can be relied upon to be correct. The correctness will be measured with the well established F-measure, described in [25] as:

$$F - measure = 2 * \frac{Precision * Recall}{Precicison + Recall}$$

where:

$$Precision = \frac{TruePositives}{TruePositives + FalsePositives}$$
$$Recall = \frac{TruePositives}{TruePositives + FalseNegatives}$$

3.3 Implementation

As previously described in chapter 2, HLA-communications are carried out between a number of federates belonging to a federation. It was therefore intuitive to create the software that would probe the system as a standalone federate, it would through this property gain access to the different functions that are granted to the HLA-federates (i.e., the ability to communicate with the other federates on a HLA-level).

The federate side is one aspect of the application, it also has to be able to communicate with the gateway-nodes (boosters) and the hardware of all nodes, as well as all additional endpoints. This communication will be further discussed in section 3.3.2.

Throughout the project, the intention was to create a software that should be easy to extend with additional functionality or adding new faults

to the detection/localization process. For this purpose, the design principles of modularity [20], dependency inversion [18] and "Don't Repeat Yourself" (DRY) have been used during the development. This was done with the intent of keeping the coupling in the final product low, making it easy to introduce new types of objects either by adding new or replacing old ones.

The implemented software is split into three major components, as seen in fig 3.1. The components are the following:

1. **System specific component** (SSC) **-** This component is the interface between the diagnostic software and the system on which it is applied. The main responsibilities of the component is creating and keeping the Network Model up to date as well as calling on the diagnostic process to run.

2. **Network Model** - Internal representation of the network, further explained in section 3.3.1.

3. **Fault Diagnostic Component -** Using the Network Model, detects and localizes faults in the network on request from the SSC, more details in section 3.3.2.

These three components are separated such that one can be easily exchanged with a new and different implementation, this is one of the steps taken towards a modular design.

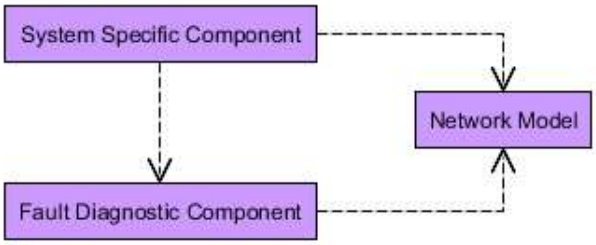

Figure 3.1: Overview of implementation architecture.

3.3.1 System Modeling

The first step in creating an automated fault diagnostic process is creating an internal representation of the network to be analyzed. Preferably the

representation should be flexible and easily changed (automatically) when new components are added and old components are removed or lost. This to make sure that the internal model is always updated insuring that the localization effort does not fail due to an inaccurate mapping of the system.

Due to the deterministic nature of the network (the path taken by the probe is known before it is sent), these requirements could be accommodated by transforming the network-graph into a tree. The monitoring federate (N0) is used as the root, and all the end-points (other federates etc.) make up the set of leaves. This structure made the probing process fairly straight forward, as will be seen further ahead in the chapter. As an example the network described in the theory chapter (fig 2.4) is simplified in fig 3.2 and used as an input producing the tree in fig 3.3.

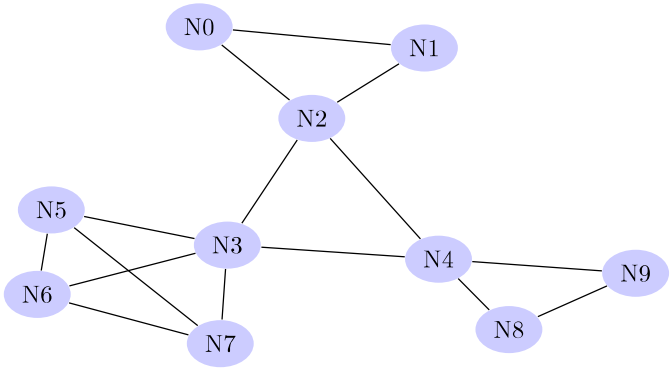

Figure 3.2: Simplified version of the network in fig 2.4.

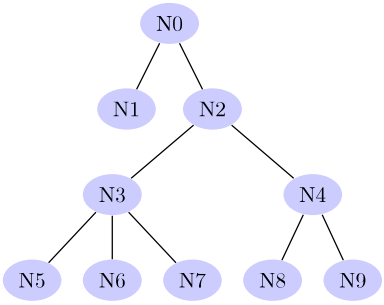

Figure 3.3: Tree-model representation of figure 3.2.

When new nodes are added or removed from the network, those changes are propagated to the tree-model, and the changes will be taken into account

during the next probing iteration.

Some objects in the system (like the nodes, probes..) share large parts of their behavior with other objects of the same type. In figure 3.4 the extraction of this common behavior to abstract classes (orange) that are realizations of the corresponding interfaces (green) can be seen. The specific implementations (blue) then extend the abstract classes with the behavior that is unique to that specific class.

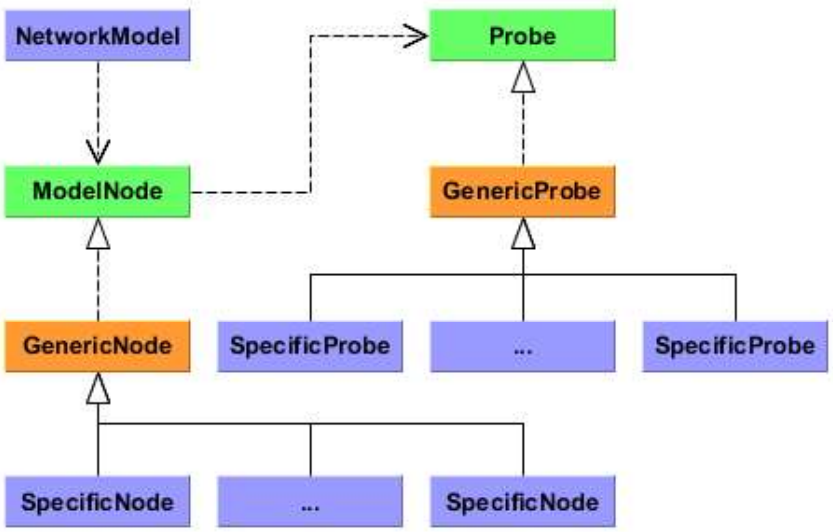

Figure 3.4: Relation between different levels of abstraction.

This illustrates all three of the design principles applied during the implementation. First: Instead of the Network Model depending upon the concrete implementations of different specific nodes and probes, it depends on their abstractions, in accordance with the dependency inversion principle. Second: Modularity - Due to the very low coupling achieved as a result of the dependency inversion, probes and nodes can be seen as modules that are easy to exchange during runtime. Making the implementation of those components modular on the class-level. Third: DRY - the abstract classes collect large parts of general behavior, if that behavior is to be altered, the changes only need to be applied in one place.

As reasoning about the state of the system was required, a form of knowledge-representation had to be chosen. The previously mentioned determinism of the network combined with the NP-hard computational com-

plexity of Bayesian inference [11] motivated the conclusion that the probabilistic approach of Bayesian belief-networks was unnecessarily complex for the purpose of the thesis project. Therefore, the decision to represent the knowledge of the current state of the system with a binary decision-tree was made.

3.3.2 Probing

As described by Rich et al. [3], the probes are either successful (represented by a 0) or unsuccessful (represented by a 1), this fits well with the binary decision-tree, where different paths are taken depending on the success of the probes used so far. Like in most of the literature [1], the probes that are used in this thesis are split into two groups, detection probes, used only to find the presence of a fault in the network, and localization-probes, used to localize the fault responsible in case of a failed detection probe.

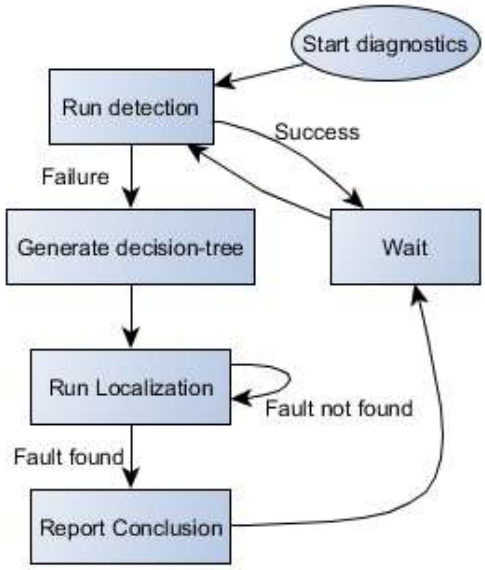

Figure 3.5: The flow of the fault-diagnostic process.

The flow of the diagnostic process conducted by the diagnostic component is presented in figure 3.5. When the component is instructed to start its process, the set of detection probes is gathered by asking the leaves of the model-tree for a detection probe, this will ensure that the entire tree is

penetrated by the detection probes and therefore, if there is a fault present, it will not go unnoticed. Each probe is selected to require a minimal amount of data to be transferred, such that it does not affect the network-load more than necessarily and different probes are used depending on what kind of node it is meant to probe. These are some of the probes that are used:

hlaFederateProbe: used to probe other federates with the help of the HLA-call ”requestInteractionsSent()”.

isBoosterAliveProbe: used to diagnose the health of a booster node (with an internal call to the booster; ”sayHelloAndGetCapabilities()”).

”ping”-Probe: formally called ICMP echo request, used to check if the hardware of a node is up. It is important to mention that this one can have some firewall issues (ICMP echo request is sometimes blocked in the firewall to prevent DDOS-attacks).

other endpoint-Probes: as several endpoints in the network might not be connected to the HLA-world, additional endpoint probes are therefore needed, as they cannot be probed by the hlaFederateProbe (some of these endpoints are the Pitch Commander Agent and Pitch Commander).

If all detection probes return successfully, the software waits for a predetermined probing interval and then restarts at the detection step. If, however, one (or more) failed detection probe(s) indicate that a fault has been discovered, a decision-tree is generated for each failed probe. Localization probes are then used in a combination of binary- and min-search [6] to further decrease the amount of probes used and thereby reducing the localization time as well as the impact on network-load. An example of a binary decision-tree for localizing a fault is showcased in fig 3.6. This decision-tree is a representation of the probes used to check the health status of the nodes on the path from the observer node, to the destination-node of the failed detection probe.

The non-leaves represent probes, solid lines are taken in the event of a successful probe result while dashed lines are used when an unsuccessful result is received. The leafs represent the states (conclusions) that the localization algorithm arrives at when it is finished. The states are the following:

1. False alarm, everything is OK.
2. The software of the endpoint is faulty.

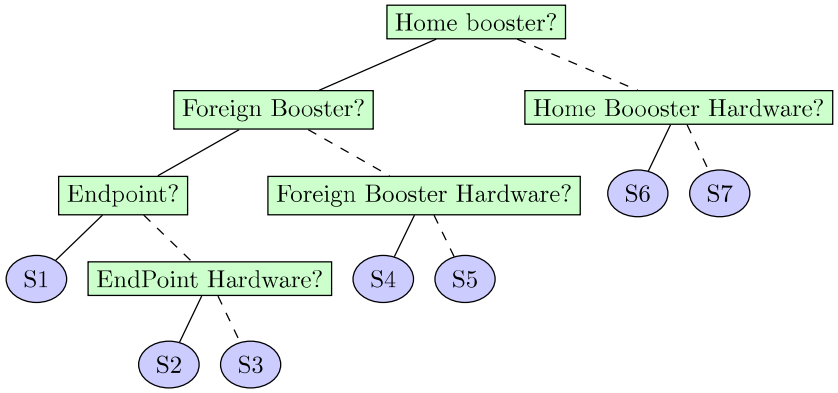

Figure 3.6: Decision tree for localizing a fault.

3. The hardware of the endpoint is faulty.
4. The software of the foreign booster is faulty.
5. The hardware of the foreign booster is faulty.
6. The software of the home booster is faulty.
7. The hardware of the home booster is faulty.

After the conclusion has been reported to the observing entity, the software waits for the probing interval to arrive again, and restarts the detection step.

3.4 Evaluation

The evaluation of the produced software will be split into two parts, one for each of the research questions; Performance and Extensibility.

3.4.1 Performance

To be able to answer the first research question, the specific numbers of the previously mentioned three performance-aspects have to be defined:

1. **Time -** When specifying the time allowed between a fault manifestation and the fault being reported to the system overseer it was considered important that the localization time should be low. However, as

Localization time	15s
Bandwidth/endpoint	5kbps
F-measure	1

Table 3.1: Performance requirements on probing software.

a human overseer cannot react or work very fast in comparison to a computer system, this localization time does not have to be faster than what a human could utilize efficiently. Other works mention sample frequencies of 10 minutes [26]. When time is of the essence, like in the previously mentioned VIKING-exercises, 10 minutes spent waiting to localize a fault is 10 minutes wasted. Although, a few seconds is negligible due to the human element of the mending process. The final time-requirement was settled on 15 seconds.

2. **Bandwidth use -** The limit of the network-load was estimated to no more than 0.5 Mbps for 100 endpoints (supported by [27], stating that no more than 10kbps should be used for each probe path), however that amount of endpoints on a 10 Mbps connection is extremely rare.

3. **Correctness** - Due to the highly deterministic structure of the software and the fact that it is only tested in a fully controlled environment, perfect precision and recall is expected, meaning an F-measure of 1. Because of this, one could consider this part of the evaluation to be of less significant than the other two. It will, however, show that the method is implemented correctly and that the effort to achieve the other two limitations does not negatively affect the correctness of the reports produced by the software.

The final values of the performance-requirements are displayed in table 3.1.

To test if the probing software can achieve these performance limits, several testing scenarios were defined. The tests were performed in a simulated environment with virtual machines that in some scenarios are affected by artificial network delay to achieve a realistic set-up for a Wide Area Network (WAN) [3].

The first two scenarios were automated with the help of the Pitch Commander software (this was not feasible in the other scenarios, they were

[3]Latency numbers retrieved from www.wondernetwork.com at 2015-09-11.

performed in a more manual manner), which allowed tapping into agents behind the respective boosters and switch the federates on and off at certain intervals with a python-script, making the testing of those scenarios significantly less cumbersome. Further details about the scenarios can be found in Appendix A.

Scenario 1: Single federate crash

A federate becomes unresponsive despite its hardware still running. The hardware of the crashed federate (and its booster) has a latency of 25-35ms (representing a node in Paris, France). This scenario was designed to test that the software was working and able to make correct conclusions about the current fault-state.

Scenario 2: Double federate crash

Similar to scenario 1, the difference being that two ”identical” faults occur at the same time at different locations. Like in the previous scenario, the first crashed federate has a latency of 25-35ms, in addition, the second is delayed by 70-100ms (representing a node in Huston, Texas). This scenario was created to see how much slower the localization process would become in the presence of more than one fault (this is not expected, but it is a point of interest).

Scenario 3: Hardware crash

The hardware of several nodes crashes (one hardware node), causing them to be unreachable. This node is ”placed” in Huston, Texas with a latency of 70-100ms. Similar to scenario 1 in the sense that the intent of the scenario was to showcase basic functionality, in this case the functionality of distinguishing between a software and hardware fault.

Scenario 4: No fault

In the last scenario there was no fault present, this is the most common state since faults are not the default state and only an occasional occurrence. This scenario was run with an increasing amount of federates to see how well the detection would scale, both in the context of localization time and bandwidth. Furthermore, in this scenario, there was no artificial latency since it is not relevant to the scaling aspect. Also relevant (for the test

of bandwidth usage) to mention that in this scenario, the detection probes were deployed at an interval of two seconds.

3.4.2 Extensibility

To be able to draw any form of conclusion of how extensible the software is in the end, an objective evaluation is required. This objective evaluation will be conducted with the help of the Quality Model for Object-Oriented Design (QMOOD). QMOOD is a method introduced by Bansiya and Davis [28] that can be used to assess the quality of different versions of a software-suite. It takes several different quality attributes into account, each of these attributes can also be evaluated separately and the one quality attribute that is of interest for the thesis is extensibility. To calculate the extensibility value of the software, the four following metrics are considered.

Average Number of Ancestors(ANA) : It is derived from the average number of classes that each class has as ancestors (i.e., how many classes it inherits from).

Direct Class Coupling(DCC) : Determined by the average number of classes that each class is directly dependent upon.

Measure of Functional Abstraction(MFA) : Calculated as the ratio of the number of methods that are inherited to the total number of methods available.

Number of Polymorphic Methods(NOP) : A count of the average amount of methods that can exhibit polymorphic behaviour.

Taking into account abstraction, coupling, inheritance and polymorphism, the final extensibility attribute value is calculated as:

$$Extensibility = 0.5 * Abstraction(ANA) - 0.5 * Coupling(DCC) + 0.5 * Inheritance(MFA) + 0.5 * Polymorphism(NOP)$$

However, the value does not mean much on its own, and can only be used to compare the evaluated software with earlier/later versions of itself or very similar software. Similarly to Goyal and Joshi [29], the values of the metrics will therefore be normalized (based on the highest achievable values) such that each metric value will be between 0 and 1. This results in the extensibility attribute value landing in the range of -1 to 3, which is then scaled so that the final value is between 0 and 1, making it easier to

interpret. This is done due to the fact that there is only one version available in this thesis project.

Chapter 4

Result

In this chapter, the results of the more practical parts of the thesis project will be presented.

4.1 Prestudy

The literature study resulted in a frame of what would be possible in regards to fault localization in the company systems. Likewise, experimenting with and discussing the company software gave a clear view of what would be possible to research in the environment of the systems. Combining the two, the direction and scope of the thesis could be decided.

Active probing [3] with the help of binary decision-trees [12] was the approach chosen to receive additional focus in the thesis by implementing the approach and evaluating its viability in the target system.

The result of the prestudy in a more concrete form is presented as the first two chapters of the report (chapter 1 and chapter 2).

4.2 Implementation

During the implementation-phase, a software was created that would represent the approach of Active probing [3]. In this software, binary decision-trees [12] were used as a representation of what is known about the state of the system at any given point in time. An overview of the three major software components of the final product was shown in fig 3.1 and the general structure of the approach is presented in the UML-diagram of figure 4.1. In

this diagram, the blue classes are full implementations, orange classes are partial implementations (abstract classes) and green classes are interfaces.

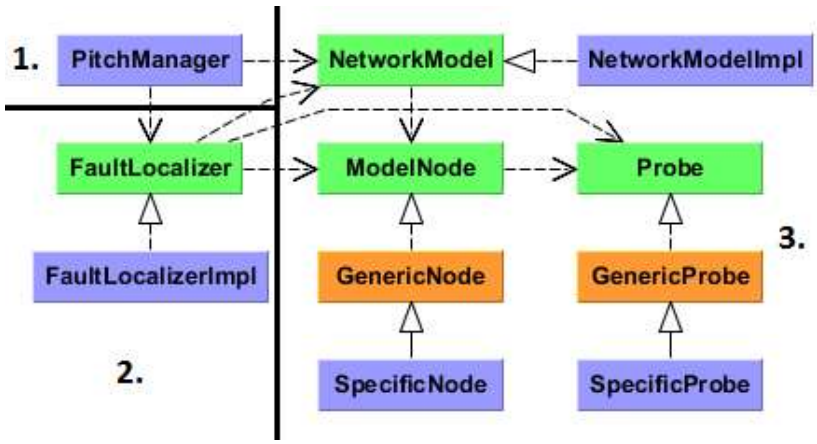

Figure 4.1: UML of the general approach.

The classes are also partitioned such that each partition belong to different major components:

1. System Specific Component.
2. Fault Diagnostic Component.
3. Model of network.

4.3 Evaluation

4.3.1 Performance

The implemented software was tested in four different scenarios to see how well it performed in regards to correctness, localization time and bandwidth usage. These test results are a summary of 100 (in scenario 1-3) to 500 (in scenario 4) test samples, the difference being due to the testing of scenario 4 going much faster. In all four of the test-scenarios, an F-measure of 1 was achieved meaning that all faults were found and there were no false positives.

Graphs of the results can be found in Appendix B.

	Total	Detection	Localization	Hardware
Average	1425.6	755.9	626.1	42.2
Median	1412	742	624	42
Max	1533	862	648	47
Min	1301	636	624	37

Table 4.1: Scenario 1, localization time (ms), sample size=100.

	Total	Detection	Localization	Hardware
Average	2161.7	765.1	1252.2	146.5
Median	2142	743	1249	142
Max	2332	862	1279	244
Min	2016	637	1249	123

Table 4.2: Scenario 2, localization time (ms), sample size=100.

Scenario 1

The first scenario was designed to test the localization time of a single fault that occurred at a remote location. Table 4.1 presents the results of this scenario.

Scenario 2

The second scenario was similar to the first scenario, the difference beeing that an additional fault was introduced simultaneously at another remote location. This to see how the localization time would vary. See table 4.2 for detailed results.

Scenario 3

In scenario three, the localization time of a hardware fault at a remote location was tested, details in table 4.3.

Scenario 4

The last test-scenario tested how well the software would perform in a scaling situation. The data collected in the scenario was:

- Time required (in table 4.4)..

	Total	Detection	Localization	Hardware
Average	5151.8	617.8	624	3854.5
Median	5190.5	649	624	3914
Max	5202	846	624	3926
Min	4186	641	624	2917

Table 4.3: Scenario 3, localization time (ms), sample size=100.

# federates	10	20	30
Average	72.4	140.3	182.4
Median	59	123	180
Max	171	248	270
Min	34	87	114

Table 4.4: Scenario 4, detection time (ms), sample size=500.

- Bandwidth required (downlink in table 4.5 and uplink in table 4.6)..

..for the detection step of the implemented algorithm.

4.3.2 Extensibility

The values in table 4.7 are the raw metric values, represented in their normalized form in table 4.8. Computing the final value for the extensibility of the software resulted in:

$$Extensibility = 0.5228 \approx 52.3\%$$

The largest contributors to this value is the very low coupling (DCC) and the somewhat high measure of functional abstraction (MFA), both of which

# federates	10	20	30	50
Average	0.66	1.37	2.0	3.46
Median	0.63	1.39	2.08	3.34
Max	0.68	1.42	2.11	3.48
Min	0.63	1.33	1.95	3.48

Table 4.5: Scenario 4, downlink bandwidth (kbps), sample size=500.

# federates	10	20	30	50
Average	2.08	7.9	17.94	48.29
Median	2.01	8.16	18.17	49.92
Max	2.15	8.16	18.17	50.2
Min	2.01	7.61	16.96	36.82

Table 4.6: Scenario 4, uplink bandwidth (kbps), sample size=500.

Metric	Value
ANA	2.3333
DCC	4.6944
MFA	0.6048
NOP	2.0857

Table 4.7: Metric values.

links back to the use of software design principles during the implementation.

Metric	Value
ANA	0.1333
DCC	0.1304
MFA	0.6048
NOP	0.4834

Table 4.8: Metric values (normalized).

Chapter 5

Discussion

This chapter will go through the different parts of the report and discuss them, linking back to relevant literature and give a basis for the conclusions that will be presented in the chapter after this one.

5.1 Results

5.1.1 Prestudy

The prestudy was meant to give an insight into the field of fault diagnosis and the inner workings of the target software system, this to give a frame of context in which the thesis project could be more clearly defined in the form of the research questions. The initial process of defining the direction and scope of the thesis was more difficult than expected and they shifted significantly in the first few weeks of the thesis. The thesis started as an investigation into how to report events that occur in certain geographical areas (in the simulated world) to an investigation into how faults can be localized with the help of error correlation tools to finally settle on the chosen approach of fault localization with Active probing.

One can consider the first question the main focus of the thesis, with the second question as a complement that deals with an interesting aspect, should the conclusion of the first question confirm that the chosen approach is indeed viable.

5.2 Implementation

5.2.1 Performance

The performance discussion will be held in three separate parts, one for each of the performance attributes defined in section 3.4.1.

Localization time

The time-requirement presented in table 3.1 states that the software should be able to find and report a fault that has occurred within 15 seconds from when it manifested. Given the response-time results, one can see that the first two scenarios are significantly faster than the third, where the hardware is faulty. This is due to the probe chosen (ping) waiting a certain amount of time before declaring that the probe had failed. This amount of time should be possible to set with a simple flag [1], however, I was unable to make it work properly in practice in a reasonable amount of time.

Despite the slow response of the hardware probe, scenario 3 shows promising numbers when comparing to the requirement. If the detection-probes are sent at an interval of 6 seconds, in the worst case the time from manifestation to report of fault would be under 12 seconds, the timing requirements are met even for the worst of the scenarios.

Additionally, scenario 4 shows that the detection time scales linearly with the number of endpoints that are probed. This time increase is only the additional time it takes to send out the detection probes (scenario 4 had no faults). In the event of a fault in an environment with a larger number of endpoints [2], the detection-step is not expected to take an increased amount of time for a reasonable number of endpoints.

Bandwidth usage

Looking at the results of scenario 4, where the bandwidth usage was measured with an increasing number of endpoints, it is clear that the downlink bandwidth is well within the requirement of 5 kbps/endpoint. The other data direction (uplink) present values that are significantly larger, although still within the requirement. Additionally, it is important to keep in mind the frequency used in the scenario (detection probes sent at an interval of

[1] ("-w") According to the manual entry for ping.
[2] Estimated number of end points where it gets significant $\sim$100

# federates	10	20	30	50
Average	0.69	2.63	5.98	16.1
Median	0.67	2.72	6.06	16.64
Max	0.72	2.72	6.06	16.73
Min	0.67	2.54	5.65	12.27

Table 5.1: Uplink bandwidth (kbps) after altered probing frequency.

two seconds), if that frequency is significantly lowered to the 6 seconds previously mentioned (in the section on localization time), the load on the uplink (and downlink) should shrink to roughly one third of what is shown in the result of scenario 4 resulting in table 5.1.

However, the numbers presented by the uplink results do not scale linearly and will at some point surpass the limit set by the requirement (although that would be at a fairly high number of endpoints [3]).

As a final note regarding scenario 4, the reason the scenario did not deal with any faults was that faults are assumed to be infrequent occurrences and the detection-probes are therefore considered to be the main load on the network, the localization-load is negligible.

Correctness

In all the different scenarios the probing software developed was able to conclude the correct state of the system in all of the cases, achieving perfect precision and recall and therefore also the perfect F-measure of 1.

One of the pitfalls that could cause a decrease in precision would be not accounting for the correct latency when pressing for a lower localization time by waiting for a shorter amount of time before declaring a probe as failed, thus causing false positives. Faults can also be misdiagnosed if the wrong conclusions are made from the probe results, which can happen if the inference-method is not implemented correctly (this includes the localization set being unable to identify all the faults uniquely). Some issues can also be caused by probabilistic inference, but since it is not used in this thesis, it is a non-issue.

While in the testing scenarios, most of these issues that can cause the correctness to decline are not present, the fact that all faults were found and

[3] An estimation calculates this number of endpoints to ∼85

that there were no false positives further strengthens the confidence in the solution to be correctly implemented.

5.2.2 Extensibility

As was mentioned in section 1.3, the reason the quality aspect of extensibility is of interest in this thesis project is the prospect of future additions to the faults the software is able to localize. In other words: *How does one introduce new fault definitions that the Active probing can start diagnosing?*

It is not of interest to modify the software, the only desire is the ability to extend the fault-definition such that it would include more possible faults (and as a result their occurrence could be localized). Therefore, this is not a case where open-box extensibility is relevant, and since the developers of a company software usually have access to the code base it is closer to a glass-box approach than gray/black-box.

During the implementation phase efforts were made to ensure that probes could be hot-swapped during runtime and thereby alter what is currently considered as "healthy" for the destination node in a dynamic way. This was achieved through the application of the design principles of modularity [20] and dependency inversion [18]. However, the structure of the binary decision-tree is very restrictive and only allows for a certain sequence of probe executions, which might not be suitable when detecting more sophisticated faults occurring in components at one simulation-site that might be caused by faulty components at another remote site.

A question, produced by this dilemma is: *Would it be possible to change the decision-tree?* (and thereby freely be able to design the order of questioning in whatever order is desired/required to localize a specific fault.)

Technically, this would be possible (with some non-major changes to the current code-base), however, the major issue in this case would be that when exchanging the old decision-tree with the new one, the localization will no longer be able to detect the faults that have been considered in the thesis (healthy/unhealthy).

For the sake of extending the fault-definition with additional, and more sophisticated faults Bayesian networks [3] would probably have been more suitable, even though it would not have been possible to test those sophisticated faults in the target system due to the lacking support structure. Had Bayesian belief networks been the choice of knowledge representation, the only addition that would be required to introduce a new type of fault would be the required probes that are used to diagnose it, and appending

the dependency-matrix [4] and the software would still be able to diagnose the old faults just as easy as the new ones.

The result of the extensibility evaluation carried out by applying the QMOOD-metric [28] for extensibility on the produced software show that some efforts have been made towards an extensible implementation. While the result of the evaluation (∼52.3% extensible) is by no means an incredibly high value, it seems to indicate that the effort to allow for hot-swapping of probes paid off (a logical conclusion stemming from the similarity of the design principles used and the way that the metrics are computed, coupling , functional abstraction etc.). This achieved extensibility is mainly noticed in practice when writing new probes or adding new types of nodes to the network that is to be diagnosed, both which is done very smoothly. Additionally, the use of "don't repeat yourself" results in writing new probes and nodes not requiring much coding-effort, as most of the behavior already is implemented. The application of dependency inversion and modularity has also made it easier to replace different modules with others if one would wish to do so, for example the model of the system is fairly well separated from the diagnosing algorithm. However, it is clear that there is plenty of room for improvement in a future version of the software.

5.3 Method

The first period of the thesis project was spent researching what has been done previously in the area of fault localization. During this initial phase of the project, the research direction was slightly different than it ended up. Bayesian networks were deemed unnecessarily complex for the purpose of determining the health of the nodes in a network following the structure of the target system. This decision was in large correct, however, when the research question regarding extensibility was altered to its current form, it soon became clear that the dismissal of Bayesian networks might have been a premature decision. As was mentioned in the previous section, extending the fault-definition would probably have been significantly simpler (or even possible at all) if Bayesian networks had been used.

For the same reason as Bayesian networks were dismissed, the Tree-model for the system was introduced. This to simplify the creation of the individual decision-tree for every node that was added. While it did help in this thesis work, the tree-structure might not have much use outside of the scope of the thesis project and might be a complicating factor when the

[4] A matrix describing which probes are affected by what fault-states [3].

work is continued towards extending the fault-definition with new and more complicated faults.

The evaluation of how viable an Active probing approach is depends mainly on the performance requirements set in collaboration with the Pitch employees. These requirements were set through the reasoning in chapter 3 (although supported by existing research) in an attempt to mirror an average (or low-end) real customer system such that it would be applicable in a more general case. However, it is important to keep in mind that these requirements are not fixed for all such systems and they may vary depending on the system that the approach is applied to.

One aspect of fault localization that has been excluded from the thesis work is how the fault localization is conducted in regards to distribution. Due to the bottle-neck that is the gateways (boosters) in the network architecture, an argument could be made for a distributed solution, a notion supported by Steinder and Sethi [2]. A solution would be a probing station behind each gateway monitoring the sub-network of that gateway to decrease the network load caused by the probing software. These local probing stations would then exchange information with each other to achieve a global comprehension about the health-status of the system. However, for more complicated faults, the distributed approach becomes much more complex as some fault in a specific region might be depending on something occurring at another location in the network [30]. This would require the local probing stations to handle more complex inference, the implementation of which would not fit into the timeframe of the project.

The argument can be made that the extensibility evaluation with QMOOD is, in a sense, futile. This due to the fact that a number (0.523) without much context does not tell the reader anything of interest. However, for further extension of the produced software a counter argument would be that a number like that helps the extension developer to estimate the how much time the extension will take. A very low value would for example indicate that more time would have to be spent on refactoring the code due to bad structure. For the general reader however, perhaps it would have been more interesting to include a step-by-step process on how to extend the software and through that give an indication on the extensibility of the product.

5.3.1 Source criticism

During the thesis project new information has been gathered continuously, which has lead to some unforeseen alterations to the focus of the report. Overall this has not been a major issue.

While the majority of the references used are primary sources, there are some exceptions, most importantly there is partial reliance on a candidate-thesis (although it is partially backed up by a doctoral thesis).

Additionally, in some cases it was hard to find reliable information, mainly the network characteristics of an HLA-network. In this case, discussions with the Pitch employees were instrumental to move forward without too much delay. However, the goal has always been to confirm everything concluded with their help by finding similar conclusions in published literature.

5.4 The work in a wider context

The probing software that has been developed in the project might not be ready to be taken into use directly in the realm of simulation networks. However, it is an indication of the usefulness of Active probing in a larger network. If further developed, it could come to help mitigate faults in large simulations for several different sectors that are important to the everyday life of almost everyone and thereby affect people all over the world. Among others, those sectors include defense, air- and train-route optimization etc.

As a counterpoint to the positive effects that might come as a result, since the probing software would be used in cooperation with software connected to such important organizations of society, the invasiveness of probing deserves a mention. Due to the nature of probes collecting information from the components that it is probing, it is not impossible to imagine a probing software being exploited to gain information regarding those organizations that might not otherwise be available.

It is important to note that although this thesis project was conducted in a context of HLA, the context is not central to the viability of the approach. The same results are to be expected in systems with similar network structure and performance requirement, regardless of what standards are used or not.

Chapter 6

Conclusion

The purpose of the thesis project was to explore the viability of an Active probing solution to fault localization in an HLA-based system. After studying the field of research and testing the Active probing approach compared to realistic requirements, the following conclusions to the research questions have been made:

With regards to real world timing and bandwidth requirements, would an Active probing solution be viable in the target system and systems alike?
For the scope of monitoring the health of the nodes, then **yes**, although some optimization could be made in regards to the amount of time to wait before declaring a probe failed and thereby lowering the total time significantly. The bandwidth requirement could also be relaxed significantly by applying a distributed approach instead of the centralized one implemented in the thesis project. In regards to the more advanced faults than unresponsive nodes, no real conclusion can be made until that path has been explored.

What measures should be taken during the design step to ensure the extensibility of the system and how extensible is the final product?
When looking at the literature, it is clear that adhering to the design principles presented in chapter 2 is the dominant way to currently build systems that are extensible by design. In the context of the thesis implementation, the efforts during the implementation phase, application of the design principles of dependency inversion, modularity and "don't repeat yourself" made the addition of new probes or replacement of modules easier. It also show in the QMOOD-measure of extensibility reaching the value of ∼52.3%. While this value is not significantly high, it is neither catastrophically low. It is clear that there is room for improvement on the extensibility side of the

software.

6.1 Future work

This research could be extended by experimenting with Bayesian networks, paving the way for inclusion of more sophisticated faults in the fault-definition. Another interesting aspect that would be on top of the priority list would be making a distributed version of the probing software and explore how that would affect the performance.

Bibliography

[1] Irina Rish, Mark Brodie, Ma Sheng, Natalia Odintsova, and Genady Grabarnik. Active probing. In *Active Probing*, 2004.

[2] Małgorzata Steinder and Adarshpal S. Sethi. A survey of fault localization techniques in computer networks. *Science of Computer Programming*, 53(2):165 – 194, 2004. Topics in System Administration.

[3] Irina Rish, Mark Brodie, Ma Sheng, Natalia Odintsova, Alina Beygelzimer, Genady Grabarnik, and Karina Hernandez. Adaptive diagnosis in distributed systems. *IEEE Transactions on Neural Networks*, 16(5):1088 – 1109, 2005.

[4] Li Deng, Xiaoyan Qu, and Dengwu Ma. The scheme design of distributed systems service fault management based on active probing. In *Systems and Informatics (ICSAI), 2012 International Conference on*, pages 1644–1649, May 2012.

[5] Maitreya Natu and Adarshpal S Sethi. Active probing approach for fault localization in computer networks. In *End-to-End Monitoring Techniques and Services, 2006 4th IEEE/IFIP Workshop on*, pages 25–33. IEEE, 2006.

[6] Maitreya Natu and Adarshpal S Sethi. Efficient probing techniques for fault diagnosis. In *Internet Monitoring and Protection, 2007. ICIMP 2007. Second International Conference on*, pages 20–20. IEEE, 2007.

[7] Mark Brodie, Irina Rish, and Sheng Ma. Optimizing probe selection for fault localization. In *12th International Workshop on Distributed Systems: Operations & Management*, 2001.

[8] Lu Lu, Zhengguo Xu, Wenhai Wang, and Youxian Sun. A new fault detection method for computer networks. *Reliability Engineering & System Safety*, 114:45 – 51, 2013.

[9] Likun Yu, Xuesong Qiu, Yan Qiao, Xingyu Chen, and Yanguang Liu. Optimizing probe selection algorithms for fault localization. In *Broadband Network and Multimedia Technology (IC-BNMT), 2010 3rd IEEE International Conference on*, pages 200–204, Oct 2010.

[10] Yongning Tang, E.S. Al-Shaer, and R. Boutaba. Active integrated fault localization in communication networks. In *Integrated Network Management, 2005. IM 2005. 2005 9th IFIP/IEEE International Symposium on*, pages 543–556, May 2005.

[11] Gregory F Cooper. The computational complexity of probabilistic inference using bayesian belief networks. *Artificial intelligence*, 42(2):393–405, 1990.

[12] Alina Beygelzimer, Mark Brodie, Sheng Ma, and Irina Rish. Test-based diagnosis: Tree and matrix representations. In *Integrated Network Management, 2005. IM 2005. 2005 9th IFIP/IEEE International Symposium on*, pages 529–542. IEEE, 2005.

[13] Björn Möller etc. *TheHLAtutorial*. Pitch Technologies, 2012.

[14] *IEEE Standard for Modeling and Simulation (M&S) High Level Architecture (HLA)– Federate Interface Specification*.

[15] Richard Mark Soley. How to deliver resilient, secure, efficient, and easily changed it systems in line with cisq recommendations. `http://www.omg.org/CISQ_compliant_IT_Systemsv.4-3.pdf`. Accessed: 2015-08-26.

[16] Matthias Zenger. *Programming language abstractions for extensible software components*. PhD thesis, ÉCOLE POLYTECHNIQUE FÉDÉRALE DE LAUSANNE, 2004.

[17] Niklas Johansson and Anton Löfgren. Designing for extensibility: An action research study of maximizing extensibility by means of design principles. `https://gupea.ub.gu.se/bitstream/2077/20561/1/gupea_2077_20561_1.pdf`, 2009. Accessed: 2015-08-26.

[18] Robert C Martin. Design principles and design patterns. *Object Mentor*, 1:34, 2000.

[19] K. Lieberherr, I. Holland, and A. Riel. Object-oriented programming: An objective sense of style. *SIGPLAN Not.*, 23(11):323–334, January 1988.

[20] Craig Larman. *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and the Unified Process*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2nd edition, 2001.

[21] Cheng Zhang and D. Budgen. What do we know about the effectiveness of software design patterns? *Software Engineering, IEEE Transactions on*, 38(5):1213–1231, Sept 2012.

[22] Erich Gamma. *Design patterns : elements of reusable object-oriented software.* Addison-Wesley professional computing series. Reading, Mass. : Addison-Wesley, cop. 1995, 1995.

[23] Attila Pásztor and Darryl Veitch. High precision active probing for internet measurement. In *Proceedings of INET' 01*, 2001.

[24] André N. Albagli, Djalma M. Falc ao, and José F. de Rezende. Smart grid framework co-simulation using HLA architecture. *Electric Power Systems Research*, 130:22 – 33, 2015.

[25] David L Olson and Dursun Delen. *Advanced data mining techniques*. Springer Science & Business Media, 2008.

[26] Jordi Meseguer, Josep M. Mirats-Tur, Gabriela Cembrano, Vicenç Puig, Joseba Quevedo, Ramón Pérez, Gerard Sanz, and David Ibarra. A decision support system for on-line leakage localization. *Environmental Modelling & Software*, 60:331 – 345, 2014.

[27] Atsuo Tachibana, Shigehiro Ano, Toru Hasegawa, Masato Tsuru, and Yuji Oie. Locating congested segments over the internet by clustering the delay performance of multiple paths. *Computer Communications*, 32(15):1642 – 1654, 2009.

[28] J. Bansiya and C.G. Davis. A hierarchical model for object-oriented design quality assessment. *Software Engineering, IEEE Transactions on*, 28(1):4–17, Jan 2002.

[29] P.K. Goyal and G. Joshi. Qmood metric sets to assess quality of java program. In *Issues and Challenges in Intelligent Computing Techniques (ICICT), 2014 International Conference on*, pages 520–533, Feb 2014.

[30] Irene Katzela, Anastasios T Bouloutas, and Seraphin B Calo. Centralized vs distributed fault localization. In *Integrated Network Management IV*, pages 250–261. Springer, 1995.

Appendices

Appendix A

Test Scenarios

The following is a detailed explanation of the testing scenarios used for the viability evaluation conducted in this thesis along with explanatory figures [1].

A.1 Scenario 1

Set-up(figure A.1):

- An example network with several gateway-nodes and federate nodes are set up.
- TestBooster5 and its sub-network have a delay of 25-35 milliseconds.

Test:

- AgentX5 is used to periodically connect and disconnect the federate encircled by red.
- The probing software (encircled by green) is running detection at the same frequency.

Expected result:
Every second iteration of the probing software should localize the fault of the "crashed" federate to be "software of federate *federate_name* is faulty". Time measurements are also taken and stored.

[1] All figures in this appendix are screenshots from the "BoosterGUI", a software developed by Pitch to give an overview of the simulation network structure.

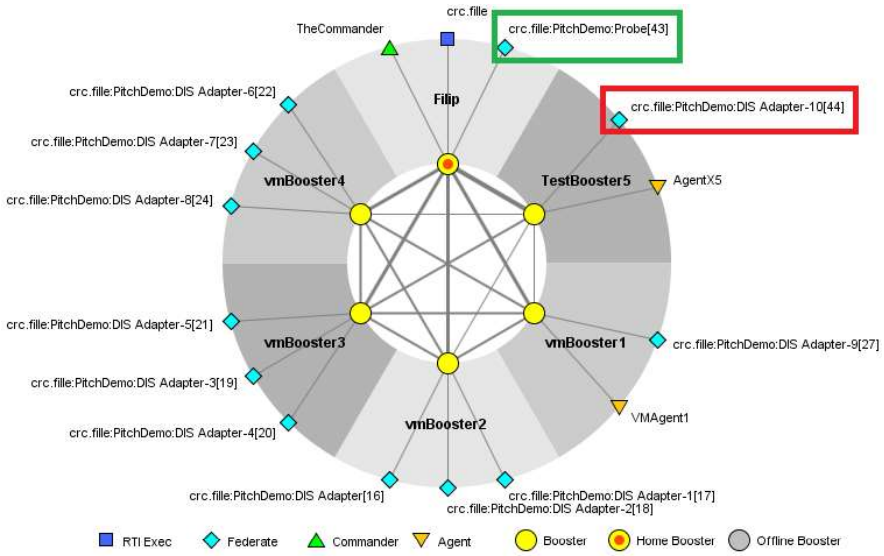

Figure A.1: Network set-up for scenario 1.

A.2 Scenario 2

Set-up(figure A.2):

- An almost identical network to that of scenario 1 is set up, on difference is the latencies.
- Delay of TestBooster5 (and its sub-network) is set to 25-35 milliseconds.
- Delay of vmBooster1 (and its sub-network) is set to 70-100 milliseconds.

Test:

- AgentX5 and VMAgent1 is used to periodically connect and disconnect the federates encircled by red.
- The probing software (encircled by green) is running detection at the same frequency.

Expected result:
Every second iteration of the probing software should localize the faults of

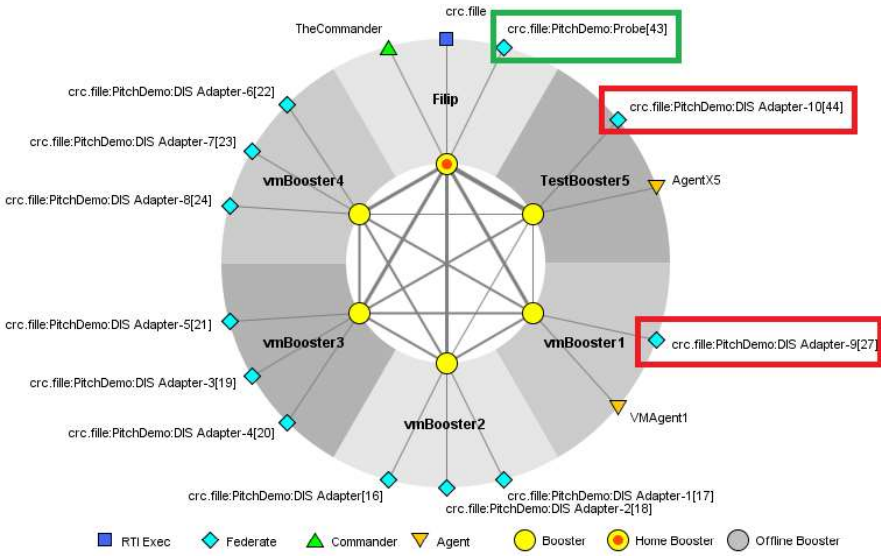

Figure A.2: Network set-up for scenario 2.

the "crashed" federates to be "software of federate *federate_name* is faulty". Time measurements are also taken and stored, a slight increase in time can be expected as there is more localization work involved (and the second federate has a longer delay).

A.3 Scenario 3

Set-up(figure A.3):

- An identical network to scenario 2 is used.

Test:

- The network-interface of the machine running the nodes encircled by red is periodically turned on and off (longer periods than the earlier tests to make sure that the interface has enough time to reconnect before it is shot down again).

- The probing software (encircled by green) is running detection at the same frequency.

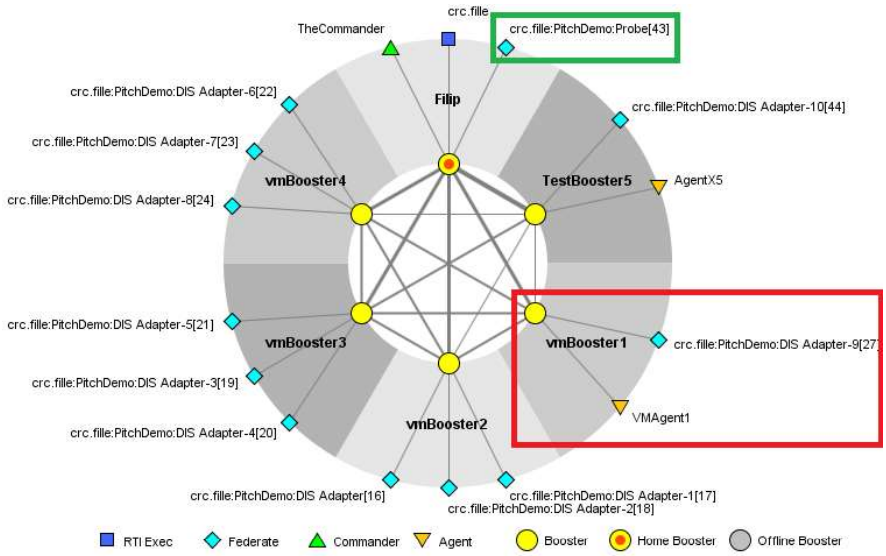

Figure A.3: Network set-up for scenario 3.

Expected result:
Every second iteration of the probing software should localize the faults of the unreachable nodes to be "hardware of nodes *node_names..* is faulty". Time measurements are also taken and stored.

A.4 Scenario 4

Set-up(figure A.2):

- A network of two gateway-nodes are set up to test the bottleneck of the gateway.
- An increasing amount of endpoint nodes are added each time the test scenario is executed.

Test:

- The probing software (encircled by green) is running detection every two seconds.

Expected result:
A collection of measurements on how the detection-step scales with and

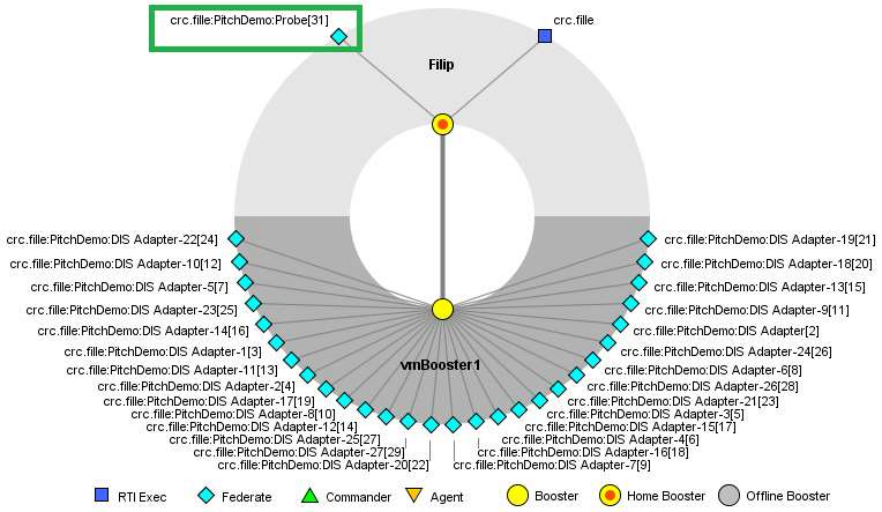

Figure A.4: Network set-up for scenario 4.

increasing amount of endpoints. Values that are measured is the detection time and the bandwidth use of the probing software.

Appendix B

Result-graphs

This is appendix is a collection of test results in graph-representation where n is sample size of each test run.

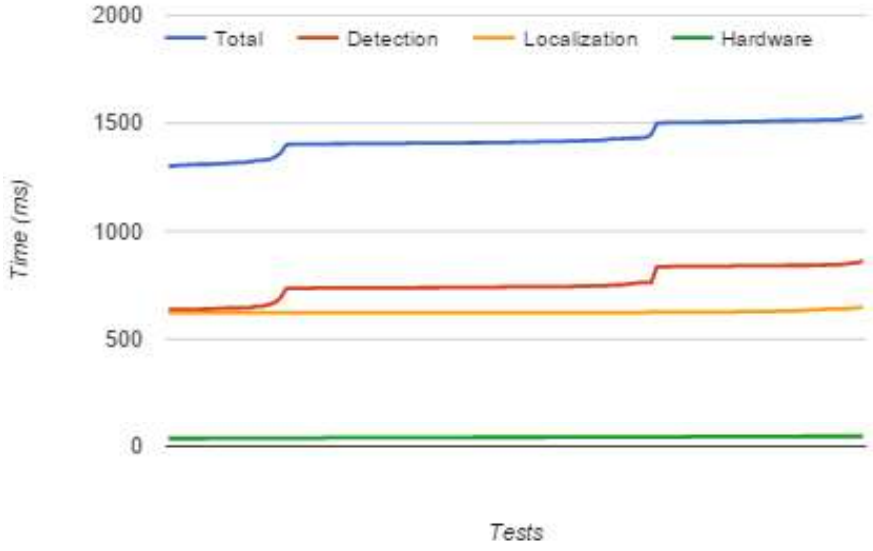

Figure B.1: Result distribution of scenario 1, n=100.

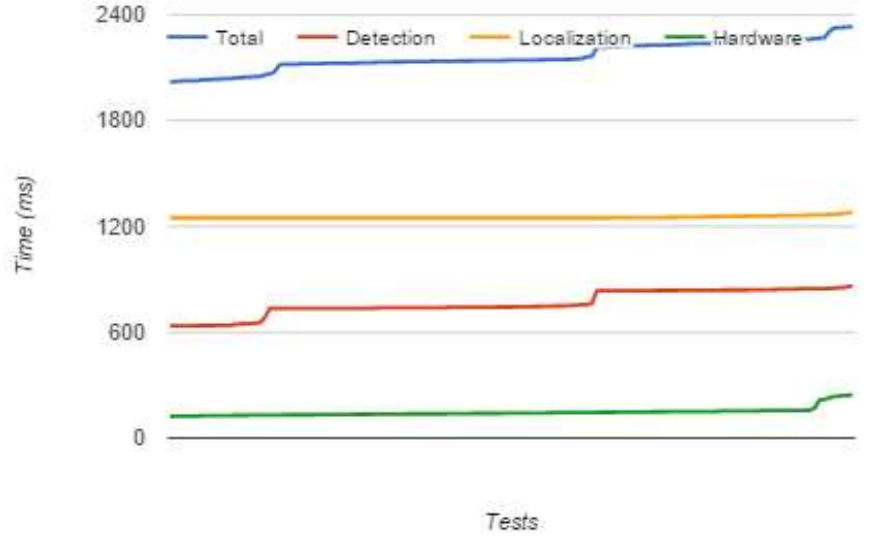

Figure B.2: Result distribution of scenario 2, n=100.

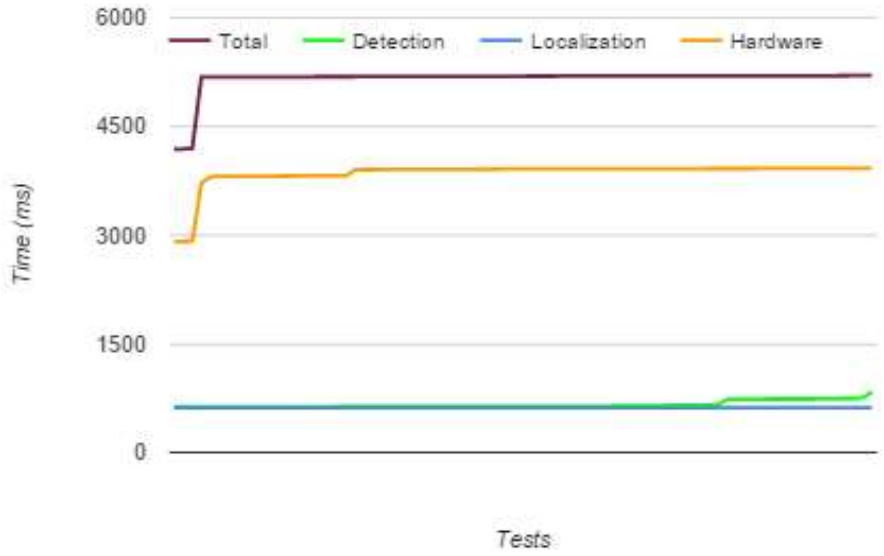

Figure B.3: Result distribution of scenario 3, n=100.

Figure B.4: Result distribution of scenario 4, detection time, n=500.

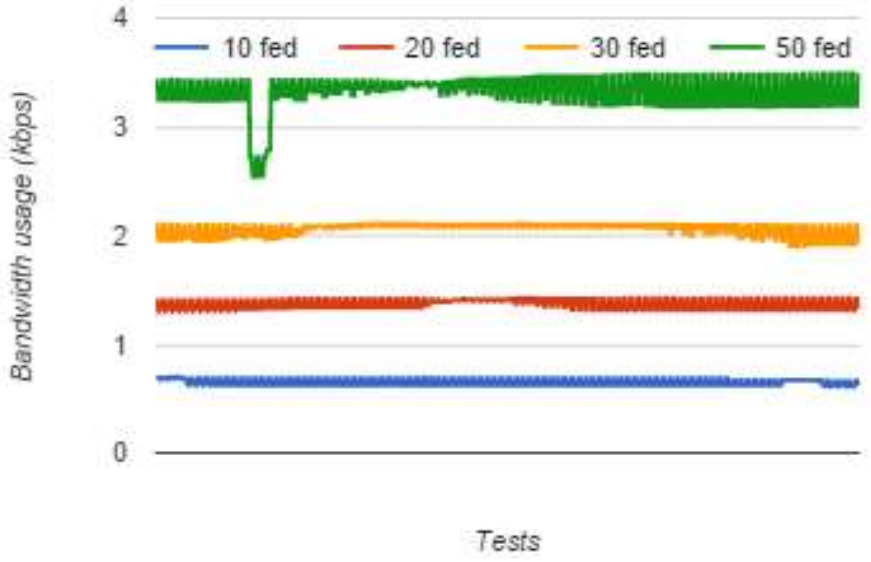

Figure B.5: Result of scenario 4, bandwidth use(downlink), n=500.

Figure B.6: Result of scenario 4, bandwidth use(uplink), n=500.

Copyright

Svenska

English