UNIVERSIDADE DE LISBOA
Faculdade de Ciências
Departamento de Informática

ANALYSIS AND IMPLEMENTATION OF CONSISTENCY AND FAULT TOLERANCE MECHANISMS IN SOFT REAL-TIME SYSTEMS

Pedro Miguel Ferreira Figueira

PROJETO

MESTRADO EM ENGENHARIA INFORMÁTICA

Especialização em Arquitectura, Sistemas e Redes de Computadores

2014

UNIVERSIDADE DE LISBOA
Faculdade de Ciências
Departamento de Informática

ANALYSIS AND IMPLEMENTATION OF CONSISTENCY AND FAULT TOLERANCE MECHANISMS IN SOFT REAL-TIME SYSTEMS

Pedro Miguel Ferreira Figueira

PROJETO

MESTRADO EM ENGENHARIA INFORMÁTICA
Especialização em Arquitectura, Sistemas e Redes de Computadores

Dissertação orientada pelo Prof. Doutor António Casimiro Ferreira da Costa
e por Eng. José António dos Santos Alegria

2014

Acknowledgments

First and foremost I would like to thank my supervisors, Prof. Dr. António Casimiro and Eng. José Alegria for granting me this opportunity and for mentoring during the duration of the project. A special thanks for the project's manager João Duarte for guiding my work, and for many hours of interesting and challenging discussions trying to find solutions for solving the problems presented during the thesis. I would also like to thank Pedro Inácio and the remainder of the PT Comunicações' team for all their advice and support during my time at PT.

I also need to thank all my friends that throughout the years helped me overcome any and all obstacles. Your friendship is invaluable and hopefully will stand the test of time.

Last but not least, I am grateful to my family, Alexandrina and Vasco, my parents and brother, for their patience and for giving me the strength and inspiration to pursue my goals and be successful.

To my brother.

Resumo

Neste projecto pretende-se explorar a implementação de soluções de tolerância a faltas num sistema de monitorização de rede. A monitorização é feita numa grande rede empresarial, com um volume de informação significativo devido ao elevado número de clientes que a empresa de telecomunicações tem a nível nacional.

Por norma, não podemos evitar todas as faltas, é preciso tolerá-las replicando os componentes do sistema para o tornar mais confiável. É necessário fazer uma análise detalhada a cada componente, verificar as falhas existentes e prováveis, e, se possível, alterar a arquitectura de modo a colmatar as falhas identificadas.

Para além de efectuarmos a análise à confiabilidade do sistema, que foi o foco principal do projecto, algum do trabalho incidiu sobre a necessidade de melhorar a monitorização do sistema, isto é, foi necessário extrair métricas e indicadores relevantes que possibilitam uma melhor introspecção permitindo despoletar acções de recuperação automáticas e uma visualização mais completa do estado do sistema.

Partindo do estado inicial, o objectivo foi analisar detalhadamente o sistema: a sua arquitectura, os componentes que o constituem e o seu funcionamento. Partindo dessa análise, foram elaborados casos de uso que serviram de base para a execução de testes ao sistema, permitindo retirar conclusões de como melhorar a confiabilidade e adicionar mecanismos de tolerância a faltas ao sistema.

A arquitectura do sistema foi analisada antes da realização dos testes, resultando na elaboração de diagramas representativos em diferentes níveis de abstracção, nomeadamente ao nível da infraestrutura e ao nível dos serviços. Adicionalmente, todas as ferramentas usadas no projecto foram revistas identificando os seus objectivos e propósitos.

A realização dos testes ao sistema permitiu delimitar a capacidade de processamento de cada um dos componentes e identificar o comportamento e consequências resultantes da injecção de faltas em situações predeterminadas. Os resultados e conclusões extraídos foram registados e serviram de base para o desenho de soluções aplicáveis ao sistema com o objectivo de corrigir as falhas identificadas.

Após as modificações iniciais a estabilidade do sistema foi melhorada, resultado da adição de capacidades de recuperação automática aos componentes do sistema, isto é, ao ocorrerem falhas identificadas e previsíveis, o sistema é capaz de retornar ao estado correcto de funcionamento sem qualquer necessidade de intervenção humana. Este tipo

de comparação foi realizada em pontos-chave do projecto com o objectivo de registar os resultados das implementações realizadas em cada etapa.

A fase final consistiu em desenhar soluções que permitissem adicionar redundância ao sistema com o objectivo de introduzir tolerância a faltas no sistema. Foi necessária uma máquina física adicional e, antes de ser escolhida a solução final, foi realizado o levantamento do estado da arte relativo às técnicas utilizadas nesta área.

Das várias opções estudadas, um subconjunto foi testado em ambiente de desenvolvimento. As soluções escolhidas para implementação tiveram em conta a facilidade e aspecto prático, bem como as vantagens e desvantagens de cada opção.

No final, temos um sistema tolerante a faltas, replicado em duas máquinas físicas que se supervisionam mutuamente tendo a capacidade de tolerar a falha de uma das máquinas.

Palavras-chave: confiabilidade, tolerância a faltas, sistemas distribuídos, replicação, disponibilidade, monitorização

Abstract

This project aims to explore the implementation of fault tolerance solutions in a network monitoring system. Monitoring is performed on a large corporate network, with a significant information volume derived from the large number of customers that the telecommunications company has nationwide.

As a rule we cannot avoid all faults, one has to tolerate them by replicating system components to make them more reliable. It is necessary to make a detailed analysis of each component, checking for flaws and possible improvements and, if possible, change the architecture so as to bridge the identified gaps.

In addition, our approach takes into account the necessity of monitoring of both the corporate network as well as the system's state. Regarding the monitored network, displaying a metrics panel is of high importance so that there is an efficient visibility of the system state.

The practical contributions began with the improvement of the system when there is a single machine deployed, testing and reporting any and all options to increase its reliability and fault tolerance. After the initial modifications, the behavior of the machine was much improved. The corrections and software updates made the node fully capable of returning to a working state in the event of the identified and predictable failures that each component is prone to.

After these improvements, the following work focused on adding redundancy to the system in order to introduce fault tolerance. It required an additional physical machine and, before committing to a final solution, surveying the state of the art techniques used in the field.

Of the several options studied, a subset was tested in the development environment. The implemented solutions took into account the easiness and practicality as well as the advantages and disadvantages of each option.

In the end, we have a fault tolerant system, replicated in two physical machines that monitor each other in order to tolerate the failure of one the them.

Keywords: dependability, fault tolerance, distributed systems, replication, availability, monitoring

Contents

List of Figures

List of Tables

Chapter 1

Introduction

This chapter serves as an introduction to the project's motivations and objectives, a summary of the contributions made and to present the structure of the document.

1.1 Motivation

With the growing competition in Portugal in providing internet services and IPTV[1], customer retention is crucial for the providing companies. On the other hand, the complexity of highly distributed systems that support these services tend to originate flaws in multiple layers of the OSI model [14]. The intersection of these realities is something that providing companies should care about and invest great efforts to improve customer service in the presence of faults.

A customer that experiences poor to faulty (or even inexistent) internet connection or television service can quickly become dissatisfied, possibly leading to a contact with the support call center, a visit to the customers home by a tech team or even the contract cancellation. In all these cases the company loses money in the process. The Network Event Trouble Seeker (NETS) is a project that originated and is developed at PT Comunicações that attempts to detect anomalies possibly affecting customer services and trigger automated correction procedures for solving them, hopefully before they are noticed by the customer. This system was designed to help oversee the wired network, and includes processes of filtering, aggregation and soft real-time correlation of basic network events. It applies soft real-time detection of relevant patterns to these events indicating suspected anomalies that affect either individual clients or groups of customers (common faults).

It is possible to take advantage of the real-time delivery of log flows sent by network equipments to reduce latency in the detection of problems, enrich or create new processes of detection, correct and / or deter problems affecting the quality of customer service. In PT Comunicações, detected situations and evidence are forwarded to the Sistema Geral de Alarmes (SGA) for analysis and correction by operational teams.

[1] Internet Protocol television

In order to provide a reliable service to the company's operation department, systems such as this must be robust and provide reliable and consistent information.

1.2 Objectives and Planning

Since its inception, the NETS system has been developed only in its functional aspect, with little focus on fault tolerance, i.e., any anomaly within the monitoring infrastructure could mean the interruption or discontinuation of its operation.

The work developed within our project was focused on achieving improvements in the following areas:

1. **Visualization**: in order to enhance the operation of the current system and the implementation of changes in the architecture, it is proposed to extract and expose in a clear and visible way, metric panels with indicators considered relevant;

2. **Single node fault tolerance**: it is essential to minimize the number of faults, and identified faults should be treated as soon as possible to ensure the continuity of service with the least possible loss of consistency;

3. **Introduce redundancy**: the NETS system is currently supported by a single physical machine, which represents a single point of failure (SPOF). To achieve this goal, new architectures should be designed and analyzed to introduce redundancy and implement an architecture with at least two physical machines.

Therefore, to achieve these improvements, the work encompassed the following activities:

1. **Study** the existing NETS platform, identifying weaknesses and potential sources of failures;

2. **Produce** software that extracts relevant metrics from the system to monitor its status;

3. **Define** solutions to the problems identified during analysis;

4. **Implement** the best suited solutions to the system;

5. **Test** solutions.

In short, the overall objective was to achieve a more robust and fault tolerant system whose availability and consistency are maintained even in the presence of anomalies. In the event of a system failure, the system detects this failure, logs relevant information and attempts to recover from the failure and return to a working state with the least disruption possible. In addition, the implementation of fault tolerance solutions based on redundancy allows the achievement of the intended robustness.

1.3 Contributions

The main contributions to improve the existing NETS system were the following:

- The source code of each component was thoroughly analyzed to correct software bugs and code refactoring in order to improve its performance. This analysis was also useful to learn what each component does, how he does it and its dependencies;
- New versions of several applications used in the NETS system were available, delivering better performance, stability and more features. Their update required code refactoring and some adaptations but the benefits were considered to be worth the extra work;
- The NETS system is constituted by a variety of components, in which the failure of a single one can potentially nullify the entire operation. Given that continuous operation is a requirement, each component began to be monitored and, if any should fail, they are restarted in order to return to a correct working condition;
- Several software applications were developed to test, monitor and report the NETS system. Testing the system required developing an application that simulates the incoming sources, in order to easily perform case studies and learn the system's performance capabilities. The monitoring and reporting software were implemented to improve the monitoring of the NETS system and to log its state and changes when a failure is detected in order to enable a better analysis and understanding of the fault's origin and details.
- Redundancy was introduced making the system fault tolerant. This required an extra physical machine and a careful analysis of how the incoming stream of events, the data processing and produced outputs are treated with the new architecture.

1.4 Document Structure

This document is structured in the following way:

Chapter 1 – Introductory chapter describing the objectives, motivation, a brief summary of the NETS project and the document structure;

Chapter 2 – A review of concepts related to the project area, ranging from monitoring and real time systems to a broad view of concepts and techniques used in fault tolerant computer systems.

Chapter 3 – Contains the analysis of the NETS system, architecture and applications/solutions used;

Chapter 4 – The proposed improvements and solutions are presented in this chapter, along with a review of their advantages and disadvantages;

Chapter 5 – The resulting implementations based on the solutions presented in the previous chapter are described here along with their details and review;

Chapter 6 – This chapter portraits a comparison of the system state between major improvements/milestones, showing the differences and benefits accomplished by the implementations made on the system;

Chapter 7 – The final remarks and conclusions are presented in this chapter. In addition to a summary of the work done, some possibilities for future work are presented, what needs to be done and what could be improved.

Chapter 2

Related Work and Technologies

In this chapter we will review theoretical concepts and technologies relevant to the project. Some of the topics covered include monitoring concepts, how it can be done and some of its strategies, real-time systems, which types and characteristics of each one, and fault tolerance, detailing dependability concepts, fault classification, strategies and mechanisms used, as well as redundancy and its place and importance in fault tolerance.

2.1 Monitoring Concepts

> *"Monitoring of a computer system is defined as the process of obtaining the status and configuration information of the various elements of a computer system and consolidating that information." D. C. Verma [58]*

The NETS system supervises the wired network allowing soft real-time detection of event patterns that indicate or allow inferring suspected anomalies using flows of network equipment logs, and then forward the detected situations and their evidence to a system accessed by operational teams responsible for resolving the identified situations.

In this domain, monitoring in soft real-time enables the identification and visualization of anomalous situations in a timely manner instead of having to rely on the information released several hours later after batch processing or through channels such as customers' complaints received via call center.

For information to be easy to see and noticeable, NETS uses metric panels to display the relevant elements in a simple way to understand, avoiding excess / showers of alarms that could undermine the monitoring of the system.

Monitoring has been discussed for a long time [3, 38, 39, 40], and its evolution grew remarkably in recent years [45, 46, 53], i.e., initially, the available tools were restricted to those provided by the operating system, such as *top* in Linux, *ping*, simple log files, etc. This type of limitation has led to the development of applications such as Nagios[1]

[1] http://www.nagios.org/

and Zenoss[2], created by the necessity to address the lack of monitoring software. Even though they stimulated the development of more tools dedicated to monitoring, they are renown to have some use and integration difficulties, problems that tend to be avoided and solved by the new generation of applications like statsd [19] and graphite [12], where there is an effort to decrease the learning curve and ease of operation of such tools.

Presently, the effort is focused in aggregating and exporting data to metric panels and presenting the information in a clear and noticeable way without generating too much visual clutter minimizing fatigue in the visualization of alarms.

Generic Model for Monitoring

According to [58], there are several categories of information retrievable while monitoring a system: status information, configuration information, usage and performance statistics, error information, topology information and security information. The collected information can pass through a chain of processes in order to clean the raw data, reduce its size into more manageable amounts, and generate reports to be consulted by system administrators.

This chain depends on the structure of the monitoring system. Although there are many different monitoring systems and products available in the market, there is a generic structure that can describe most of them. The generic structure, illustrated in Figure 2.1, consists in five layers of monitoring functions.

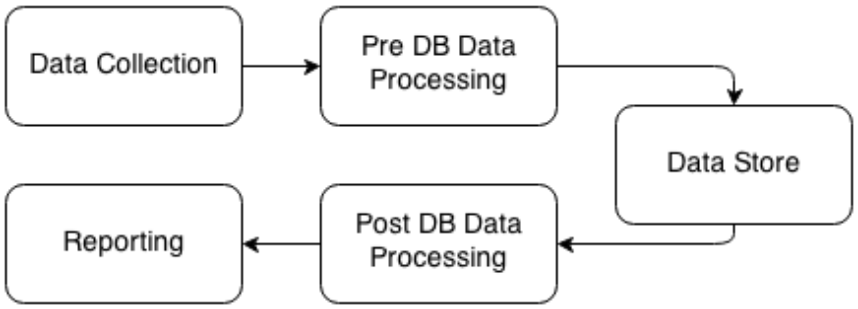

Figure 2.1: Generic structure of monitoring applications.

The *data collection* is performed through equipment monitoring and can be achieved in two ways: Active Monitoring and Passive Monitoring [58].

- **Passive Monitoring** - In this type of monitoring, information is collected without introducing additional work, that is, information is available through normal operation of the system. The only additional traffic introduced in the network is the necessary for agents or devices to send the generated data to the system management. This approach can be used in any computer system.

[2]http://www.zenoss.com/

Some examples of information that can be sent include log files generated by applications, the MIB[3], which describes the structure of the management data of a device subsystem, and information about the network topology which is usually maintained by network protocols such as OSPF[4] and BGP[5], among others.

This type of monitoring can generate a significant amount of useful information for the management system and there are several typical methods used depending on the environment and architecture used.

- **Active Monitoring** - In contrast to the passive model, this technique assumes an intrusive monitoring model explicitly requiring the direct inquiry of agents or target devices. It works by sending requests to the monitored equipments, and waiting for their response that contains the available information.

  In this case, we are only inquiring for specific data which could mean lower bandwidth requirements, but, in contrast, there is an added network load due to the need of constantly sending requests to the device and waiting for the answers. Furthermore, this method can provide information that could be difficult to obtain using passive methods.

In the NETS system, data streams arriving from agents who are sending data and messages with relevant events are considered passive, while monitoring within the NETS infrastructure, relative to processes, disk space, state of the JVM[6], etc., is made actively.

According to [58], the *Pre DB data processing* has three main objectives: data reduction, data cleansing and data format conversion.

- **Data Reduction** - Reducing the volume of information by reducing redundant information. It can be achieved by using the following methods: aggregation, thresholding and duplicate elimination;
- **Data Cleansing** - Cleaning the data by removing erroneous or incomplete data;
- **Data Format Conversion** - Converting the information to a format that will be stored in the database.

The resulting output of the *Pre DB data processing* is then stored in the data store, usually using a commercial off-the-shelf (COTS) database solution that can have different designs: rolling, load-balanced, hierarchical database federation, partitioned, and round-robin. The chosen database design is selected according to the expected volume of information, but it is possible to have multiple types of databases operating in the same

[3]Management Information Base
[4]Open Shortest Path First
[5]Border Gateway Protocol
[6]Java Virtual Machine

system to distribute the data load or handle different types of information, for example, one database for performance statistics and another for security information.

The *Post DB Data Processing* is responsible for analyzing the collected information and decide what is relevant to be reported. For instance, it correlates information from different sources, and uses patterns to determine specific alarm situations.

The last section is *Reporting*, which can be composed by viewing consoles and monitors that present the information in a simple and effective way, enabling system administrators to effectively visualize the monitored system's state.

2.2 Metrics and Measurements

> *"It is not the strongest of the species that survives, nor the most intelligent, but the one most responsive to change." Charles Darwin*

Metrics can be defined as a **system of parameters** or ways of **quantitative** and **periodic assessment** of a process that is to be measured, along with the procedures to carry out such measurement and the procedures for the interpretation of the assessment in the light of previous or comparable assessments. For example, they can represent the number of failures of a certain component, the load of a system, latency, etc., indicating how well behaviors and processes are functioning, as well as their performance while highlighting areas for improvement (weaknesses) and strengths within the system.

Monitoring can use metrics to diagnose problems, point to solutions, respond appropriately to change and allocate resources accordingly. But it is very important to know what information to retrieve and how often to collect it, which can introduce significant changes in the architecture of our monitoring system. While we may retrieve all imaginable metric every second, it may overload the network and make the metric panels more complex than necessary. On the other hand, if we retrieve too little or erroneous information it may harm our system rather than properly monitor it [58].

Some of the available techniques used in monitoring systems to aggregate and order data sets are listed below. Each one has its usefulness depending on the objective, values in question and situation.

The **average** is a standard aggregation technique which alone might not be useful depending on the objective, because two sets can have completely different values but with the same average. I.e., the set 3,3,3,3,3 possess the average value of 3, and another set 1,2,3,4,5 has the same average as the previous.

Using the **median** we can get more detail compared with the average, given that it separates the set in half after being ordered, indicating the values which separate the upper and lower half, resulting in a value closer to the typical value because it does not undergo many changes if there are many disparate values.

The **standard deviation** measures the dispersion of a data set in relation to their average, that is, if the values are all very close it will have a low standard deviation, instead of a data set with a highly dispersed set of values, which will have high standard deviation. It can be used along with the *average* as a complement, there are several probability distributions that are characterized by the *average* and *standard deviation*, e.g., Gaussian distribution.

The **percentile** is a measure used in statistics indicating the value below which a given percentage of observations in a group of observations fall. For example, the 20th percentile is the value (or score) below which 20 percent of the observations may be found. There are three special percentiles called the *quartiles* which divide the data into four groups of equal size, the 25th percentile is also known as the first quartile (Q1), the 50th percentile as the median or second quartile (Q2), and the 75th percentile as the third quartile (Q3).

The **summary of the five numbers** [21] is a descriptive statistic that represents information obtained from a number of observations, being necessary to have the percentages based on a random variable measured on a scale. It is composed of five percentiles, minimum sample, lower quartile, median, upper quartile and maximum sampling. This technique provides a concise summary of the observable distribution, being suitable for ordinal measurements, intervals and ratios.

Besides statistical concepts, **time series** analysis comprises methods for analyzing data in order to extract meaningful statistics and other characteristics of the data. It shows the variation over a period of time, noting a measured value at specific intervals. This type of observation should consist in a sufficient and meaningful time window to obtain visible and relevant data. The data analysis requires its aggregation and to be temporally ordered, and there is a large dependence on the time in which they occur. It's an analysis technique suitable for forecasting taking into account historical series.

2.3 Real-Time Systems

"Real-time System – system whose progression is specified in terms of timeliness requirements dictated by the environment" P. Veríssimo [57]

In terms of real-time systems, we have three distinct classes as we can see in [36, 57]:

Hard Real-Time - Systems of this class have to avoid any kind of time failure and can have catastrophic consequences when one happens. Example: On-board flight control system (fly-by-wire[7]).

[7]Fly-by-wire (FBW) - A system that replaces the conventional manual flight controls of an aircraft with an electronic interface, where the movements of flight controls are converted to electronic signals transmitted by wires (hence the fly-by-wire term).

Mission-critical Real-Time - Such systems should avoid time failures, where occasional failures are treated as exceptional events. Example: Air-traffic control system.

Soft Real-Time - In such systems, occasional failures are accepted and do not cause major problems. Example: Online flight reservation system.

Regarding monitoring systems, they can perform their operation in real-time and belong in one of the above mentioned classes, or, alternatively, perform in an off-line manner through batching or with large time intervals, e.g., hourly or daily.

Off-line monitoring can be used in contexts where the systems being monitored do not require immediate action should an alarm be triggered or and anomaly detected. While real-time monitoring systems allow a constant and continuous visualization of the monitored metrics, and enable the prompt detection of behavior deviations, outliers and warnings depending on what is being monitored. This also provides the chance to resolve the issue or forward to the appropriate field technicians enabling a faster corrective action.

2.4 Fault Tolerant Computing and Dependability Concepts

In this section we present and explain the main concepts for fault tolerant distributed systems, dependability and strategies to achieve robust systems.

In Figure 2.2 we can see a summarized view of the notions that will be introduced, grouped in three classes, attributes, means and impairments. These concepts are an important basis to enable the improvement of a system's dependability which is one of this project's main goals.

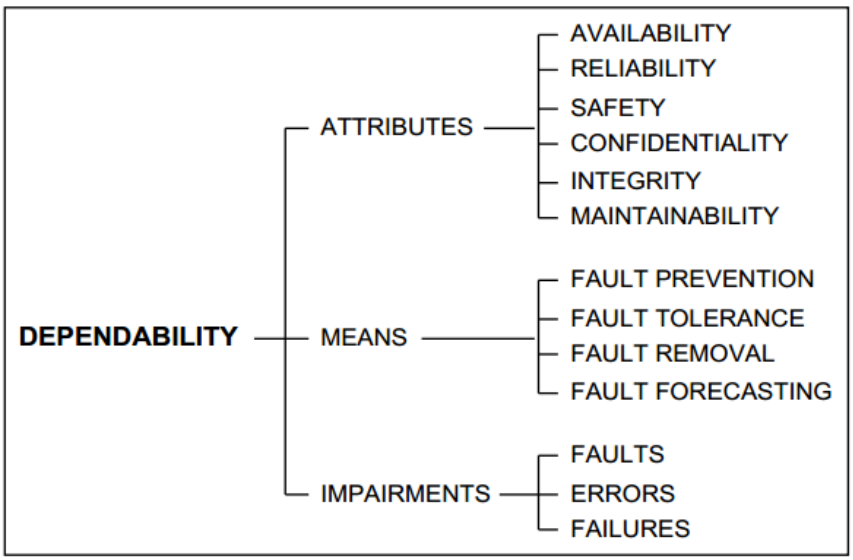

Figure 2.2: Dependability tree.

2.4.1 Dependability

"Dependability – the measure in which reliance can justifiably be placed on the service delivered by the system" J.-C. Laprie [34]

The **dependability** of a computer system is the ability to avoid system failures that are more frequent and more severe than acceptable [4]. Some of the biggest threats are faults, errors and failures. Next is a list of attributes that can be assessed to determine the dependability of a distributed system, as seen in [57, 34]:

Reliability – the measure of the continuous delivery of correct service, can be expressed by the mean time to failure (MTTF) or mean time between failures (MTBF);

Maintainability – the measure of the time to restoration of correct service and can be expressed by the mean time to repair (MTTR);

Availability – the measure of the delivery of correct service with respect to the alteration between correct and incorrect service, it is expressed as MTBF / (MTBF + MTTR);

Safety – the degree to which a system, upon failing, does so in a non-catastrophic manner;

Security – The measure of guarantees in terms of confidentiality, integrity and availability in service provision;

Integrity – Absence of improper system alterations;

Resilience – The ability to tolerate and recover from faults.

The most important aspects that will be addressed in this project will mainly focus in improving a system's reliability, availability and resilience. This is accordingly to the requirement of delivering a continuous and correct service (monitoring) even in the presence of faults.

2.4.2 Faults Classification

A **failure** means that the system presents an incorrect or abnormal behavior, resulting as a deviation from its regular service delivery. These failures result from **errors**, which are symptoms of existing system **faults**, representing invalid system states that should not be reached. A single **fault** can spawn multiple **errors** and consequently multiple system **failures** and a failure can range from different types of faults and even the combination of some of them.

The relationship between fault classes can be seen in Figure 2.3, where it is visible that some classes contain others, meaning that if a system tolerates a more general class, it can assume that it tolerates the faults of the contained classes.

Faults can be classified according to different criteria, such as their characteristics and how the component behaves once it has failed [57]. They can be **independent faults**, attributed to different causes, or **related faults** associated to a common cause, usually

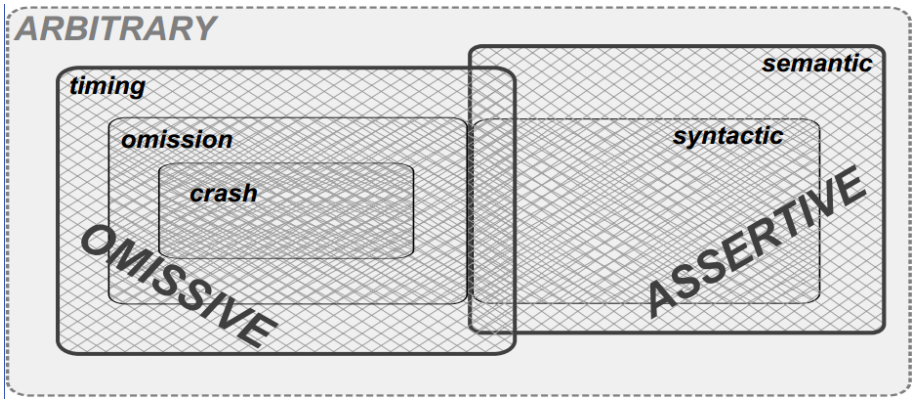

Figure 2.3: Relationship between fault classes.

originating **common-mode failures** which is the result of one or more events, causing coincident failures in multiple systems, e.g., single power supply for multiple CPUs, single clock, single specification used for design diversity [33], etc.

In relation to the domain, they can be **hardware** or **software faults**. Design faults occur more frequently in software than in hardware because of the difference in complexity between these two domains. This can be explained by the difference in internal states, as the hardware machines have a smaller number comparing to software programs [48, 49].

Regarding persistence, faults can be classified as **permanent** or **transient**. Hardware faults can be be of both kinds, but a software fault is always permanent. Software faults that appear transient are in fact permanent software faults with a complex activation pattern, which limits its reproducibility if it cannot be identified.

The most benign class of faults belong to the ***omissive fault*** group: they occur essentially in the time domain when a component does not perform an interaction as it was specified to do.

- **Crash faults** – Occurs when a given component permanently halts its operation, or never returns to a valid state;
- **Omission faults** – When a component occasionally omits an action, failing to perform its service;
- **Timing faults** – Occur when a component is late or does not perform its service on time.

On the other hand, ***assertive faults*** belong in the value domain, and are characterized by a component performing an interaction in a manner that was not specified.

- **Syntactic** – When the value's format is incorrect, e.g., a temperature sensor showing "+Ad" degrees;

- **Semantic** – When the format is correct, but the value is not, e.g., a temperature sensor showing 26ºC when it is snowing.

The last class is ***byzantine and arbitrary faults***, being a combination of the omissive and assertive faults, are the most dangerous and hardest to tolerate, usually involving heavy and complex techniques to face them successfully. In this case, the system is assumed to fail in any manner of way, possibly resulting in erratic behavior from processes, not following protocols, send contradictory messages, etc.

In general byzantine fault tolerance in a monitoring system is unnecessary given its non critical nature. But a continuous and correct delivery of service is expected. To this end, there is a great focus on treating crash and syntactic faults, without undervaluing the remainder types of faults.

2.4.3 Strategies and Mechanisms

To obtain a more reliable solution and increase the time between failures we can use some strategies and mechanisms that enable prevention, tolerance, removal and forecasting of faults [57, 4]. The use of these strategies and mechanisms should be encouraged in order to maximize the reliability of computer systems.

The **impairments** to dependability are faults, errors and failures. They are not desired and appear unexpectedly, causing and resulting in undependable states [34]. In particular, the complete elimination of software design faults is very hard through fault prevention and removal. Additionally, hardware faults, either permanent or transient, may happen during system operation. Therefore, only fault tolerance can cope with residual software faults and hardware operational faults. In the monitoring scope, fault prevention and removal are irrelevant whereas fault forecasting can be useful, but our main focus is in fault tolerance.

Fault Tolerance – Continuous operation despite the presence of faults. There are several mechanisms that can help a system to guarantee such capability:

- **Error Detection** – It targets the immediate detection of errors in order to contain them, avoiding propagation and triggering recovery and fault treatment mechanisms. It can be **reactive**, detecting and reporting error at run-time, or **proactive**, whose detection and report is done at development time.

- **Fault Recovery** – The main goal is to restore the system to a state without faults / errors [32].

- **Fault treatment** – Aims at preventing faults from being reactivated.

The main concern is having active redundancy through fault masking or semi-active redundancy using error detection and forward recovery.

Using fault masking it is expected that the system has enough redundancy to deliver a correct service without any noticeable glitch, whereas error detection would reactively trigger forward recovery causing the system to move forward to a predictable and ensured correct state. Either way, faults treatment can be performed by reinitializing key software components.

2.4.4 Redundancy

> *"A key supporting concept for fault tolerance is redundancy, that is, additional resources that would not be required if fault tolerance were not being implemented." Laura L. Pullum [47]*

The use of redundancy is fundamental to fault tolerance. It consists in the utilization of additional resources that are not required for normal system operation. In computer systems, redundancy can be applied in the space, time and value domains [57, 47].

- **Space Redundancy** - Consists on having several copies of the same component, e.g., storing the information on several disks, different nodes computing the same result in parallel, disseminating information along different network paths, etc.

- **Time Redundancy** - Consists in doing the same thing more than once, in the same or different ways, until the desired effect is achieved, e.g., the retransmission of a message to tolerate possible omissions, repeating computations, etc.

- **Value Redundancy** - Consists in adding extra information about the value of the data being sent or stored, e.g., parity bits and error correcting codes can be added to detect / correct data corruption.

In the present context, we will disregard time and value redundancy because they are not a requirement in this monitoring scope, and will focus on space redundancy. For example, it is necessary to have copies of the system's nodes in order to avoid losing state and prepare it for possible node faults, at which point a spare node will take over the operation minimizing the system's unavailability while the faulty node recovers.

2.4.5 Common-Mode and Common-Cause Failures

> *"A Common-Mode Failure is the result of an event(s) which because of dependencies, causes a coincidence of failure states of components in two or more separate channels of a redundancy system, leading to the defined systems failing to perform its intended function." Bourne et al.[6]*

The benefits of duplicating components can be defeated by common-mode and common-cause failures. Common-cause failures are **multiple component failures** having the **same cause**, and common-mode failures is when **an event** causes multiple systems to fail. These types of failure happen when the assumption of independence of the component's failure is invalid.

Common-cause failures can occur owing to **common external** or **internal influences**. External causes may involve operational, environmental, or human factors. The common cause may also be a (dependent) design error internal of the supposedly independent components.

- **Design Diversity** - To protect against common design errors, components with a different internal design (but performing the same function) may be used. This approach is called "design diversity"[44]. Multiple versions of software that are written from equivalent requirements specifications are examples of design diversity. That is, the component requirements are the same, but the way the requirement is achieved within the component may be different.
  Two pieces of software that compute a sine function but use different algorithms to do so are an example of design diversity.

- **Functional Diversity** - A second type of diversity called "functional diversity", involves components that perform completely different functions at the component level (although the components may be related in that they are used to satisfy higher-level system requirements). The crucial point is that the component requirements are different.
  In the case of software, functional diversity means that the behavioral requirements for the software are different. For example, one program may check to see whether two numbers are equal and another, functionally diverse program, might select the larger of two numbers.

Common-mode and common-cause failures can effectively halt a monitoring system's operation, therefore, it is important not to undervalue mechanisms and techniques that can aid in their treatment. For example, the input volume of a monitoring system is generally high, but it can take a single faulty message to potentially render every machine belonging to the system unavailable by crashing or generating an exception on them.

2.5 Existing Monitoring Solutions

Currently, there is a wide array of options and tools available to perform monitoring on computer systems and networks, process logs, manage messages, etc., both commercial and open source. This section presents some of the most relevant tools or solutions that could relate with the NETS system's work.

2.5.1 Nagios

Nagios was initially released in 1999, making it one of the oldest applications of its genre. It works as an overseer/monitor constantly regulating and monitoring the health of hardware, software and the network using SNMP [8] as the default communication protocol. Nagios can monitor parameters such as *system status*, if it is up and running, CPU/memory/disk usage etc., *service status*, whether a service is up and running, e.g., DNS, Web Server, mail server, etc., and other factors including room temperature, humidity, among others. Nagios can generate alerts through SMS/email when the monitored parameters exceed previously set thresholds.

The Nagios application is split in different parts, the main application is responsible for scheduling and executing the service checks in a concurrent fashion, while maintaining all state information and taking action when transitions require it. Additionally, the checks are called Nagios plugins and are available separately, in which their use is recommended but not necessary as they can be developed locally. The core add-ons include Nagios Remote Plugin Executor (NRPE) and Nagios Service Check Acceptor (NSCA). The first is used to execute indirect checks locally on target hosts, and the NSCA enables passive monitoring, where Nagios is split into client and server application.

The usage of Nagios by PT Comunicações is restricted to its usual objective which is monitoring computer systems, networks and infrastructures. But for the project at hand, it has too many useless features, lacks correlation capabilities, and requires installing software for the remote monitoring and script executor which is not an option given some of the monitored equipment cannot be modified.

2.5.2 Zenoss

Zenoss is a monitoring solution similar to Nagios, includes features like automatic host discovery / monitoring providing time series graphs on performance, event management which normalizes data into events allowing users to set alarms based on the frequency of an alert, it is ready for Syslog monitoring and SNMP/SSH.

The functional similarities between Nagios and Zenoss also mean that its usage by PT Comunicações would be limited to its natural scope, therefore, it is not a good candidate for the project.

2.5.3 Sensu

Sensu [9] is a monitoring framework written in Ruby [10] that aims to be simple, malleable and scalable. Sensu schedules the remote execution of checks and collects their results;

[8] Simple Network Management Protocol - http://www.snmplink.org/
[9] http://sensuapp.org/
[10] https://www.ruby-lang.org/

uses RabbitMQ [11] to route check requests and results. Checks always have an intended target, servers with certain responsibilities, such as web servers or data storage. A Sensu client has a set of subscriptions based on its server's responsibilities; the client will execute checks that are published to these subscriptions. A Sensu server has a result subscription where the clients publish their checks results.

Sensu has a messaging oriented architecture in which messages are JSON [12] objects, the ability to re-use existing Nagios plugins or other written in any language, it also supports sending metrics to various backends (Graphite [13], Librato [14], etc.).

This framework is an open source solution that has been actively developed in the last years. At the time of the project's creation it was not in a stable condition and lacked the required features. As of now, it still would not bring any advantages to the NETS system while having costs associated with its adoption.

2.5.4 TRONE

This project[15] is part of a collaboration between the University of Lisbon, University of Coimbra, Portugal Telecom and Carnegie Mellon University, and aims to solve problems associated with cloud infrastructure, ensuring persistent operations while tolerating accidental and malicious faults, and monitoring of the processes in an accurate and reliable way.

To achieve these objectives, the approach taken was the preparation of a robust architecture with fault diagnosis and automatic fast reconfiguration.

One of TRONE's components is the Fit Broker, whose implementation is in the prototype stage, following an approach of Publish / Subscribe, replication state machines using the platform BFT-SMaRt [16], public-key encryption, event channels with support for quality of protection (QoP) and quality of Service (QoS) with different levels of fault tolerance, publicizing the ability to handle large volumes of data, little intrusiveness and ease of integration.

This project was taken into account given it seemed to fit the NETS system context, especially the FIT Broker as a replacement for the NETS's message broker, RabbitMQ. After a detailed analysis we found little gains in its adoption. The main advantage was the addition of intrusion tolerance, which is not the focus of this project, and there were disadvantages regarding the broker performance because of the complexity inherited by its algorithm, and the cost of adopting it.

[11] http://www.rabbitmq.com/
[12] JavaScript Object Notation
[13] http://graphite.wikidot.com/
[14] https://metrics.librato.com/
[15] http://trone.di.fc.ul.pt/
[16] https://code.google.com/p/bft-smart/

2.6 Summary

This chapter introduced the definitions and context that will be the base of the project's work, including: a review of what is monitoring and how metrics and measurements take part in that process, a brief definition of the different types of real-time systems and the dependability and fault tolerance concepts that will be the main focus of the project's work.

Additionally, we perform a survey of related technologies that currently exist in the market characterizing them in terms of its monitoring type, real-time class and what metrics are treated. The objective of this survey was to present the solutions that are available, justifying why they were not used by PT Comunicações and, instead, motivated the creation and development of the NETS project.

Chapter 3

NETS System Analysis

This chapter presents an in-depth analysis of the NETS system aiming to improve it with respect to dependability, by applying fault tolerance solutions. The tools used within the system will be reviewed describing their purpose, the physical and software architectures are detailed and pictured, and a section dedicated to fault modeling and testing will identify some of the cases that need to be solved.

3.1 Methodology

The methodology consisted in:

- Analyzing the system's characteristics: its architecture, components and software;
- Translating the analysis into diagrams, picturing: the physical and software architecture, data flow, processes' use cases and dependencies;
- Developing software to aid in testing, monitoring, and injecting faults in the system;
- Executing performance tests to establish the system's capability bounds;
- Reviewing the reliability and recoverability of components, testing their resilience to fault injection and verifying, in case of failure, the consequences to the system;
- Analyzing fault tolerance mechanisms implementable in the system;
- Implementing and testing the chosen solutions;
- Comparing the system between milestones in order to expose the advantages and improvements made possible through the implementations in the system;

3.2 NETS System Tools

This chapter will present and describe the software choices that were made when the NETS project was designed to support the system's architectural solution. Therefore, they were already integrated in the system at the time of this analysis.

3.2.1 logstash

Logstash [25] is an open source tool for event and log files management and processing. It can be used to aggregate, analyze and store log files for later use. One of the key concepts of this application is functioning in a three stage pipeline, input, filters and output, that is, the events are received by the inputs, modified by the filters and transmitted through the outputs. There is a wide variety of both inputs and outputs, e.g., AMQP [1], elasticsearch [15], snmptrap, sqlite, TCP, UDP, etc. available for use.

3.2.2 RabbitMQ

RabbitMQ [20] is an open source message broker software written in Erlang [16], a programming language used to build scalable soft real-time systems with requirements on high availability and fault tolerance, that implements the AMQP.

RabbitMQ offers a variety of features that allows one to trade off reliability for performance, including persistence, delivery acknowledgment and high availability through clustering. It has flexible routing, as messages are routed through exchanges, which can be bound to one or more queues and are responsible for delivering the messages to the relevant queues depending on the data. After that, the consumers can pool the queues for data. It has a built in capability to form clusters on a local network, forming a single logical broker. It is equipped with an API with options to show information that include the currently open connections, exchanges/queue list, exchanges/queue size, etc. and a web interface.

3.2.3 Esper

Esper [17] is an open source event driven correlation engine written in Java created for financial systems. Correlation rules are described in syntax similar to SQL [2], supporting patterns that are defined and used in the detection of pre-defined situations.

In order to interface with the "outside world", Esper also provides an extensible library (EsperIO) of plugins for reading and writing events using protocols such as HTTP [3], JMS [4], AMQP and CSV [5] files.

3.2.4 Graphite

Graphite [12] is an open source tool for monitoring and graphing the performance of computer systems. Graphite collects, stores, and displays time series data in real time

[1] Advanced Message Queueing Protocol - http://www.amqp.org/
[2] Structured Query Language - http://www.sql.org/
[3] Hypertext Transfer Protocol
[4] Java Message Service
[5] Comma Separated Values

using round robin databases [58], providing a web interface and API to both generate graphics and extract values and claims to scale both horizontally and vertically. Uses a naming scheme based dot notation (e.g. “metrics.hostname1.cpu.avg”), and has an API to send data through TCP.

3.2.5 Statsd

Statsd [19] is a daemon that listens for metric measurements on an UDP port and aggregates them, outputting statistics to another service such as Graphite. It works by using buckets for each stat (which has a value, typically an integer) and periodically computes the aggregation after a configured flush interval. There are various types of metrics, including counters, timers and gauges. It is an open source solution.

3.2.6 Monit

Monit [37] is an open source utility for managing and monitoring, processes, programs, files, directories and file-systems on a UNIX system. Monit conducts automatic maintenance and repair and can execute meaningful causal actions in error situations.

Contrary to many monitoring systems, Monit can act if an error situation should occur, e.g., if sendmail is not running, Monit can start sendmail again automatically or if apache is using too much resources (e.g. if a DoS attack is in progress) Monit can stop or restart apache and send you an alert message. Monit can also monitor process characteristics, such as how much memory or cpu cycles a process is using.

Monit can also be used to monitor general system resources on local-host such as overall CPU usage, Memory and Load Average.

Additionally, Monit can be expanded with M/Monit, increasing its capabilities and providing monitoring and management of all Monit enabled hosts via a modern, clean and well designed user interface which also works on mobile devices.

3.3 NETS Architecture

This section explores the architectural details of the NETS system, describing the advantages and capabilities of the currently used tools and an analysis of the components that constitute it.

3.3.1 Physical View

The NETS system was originally designed to be based on a single physical machine (Figure 3.1) running two VMs[6] related to the system, one for data processing and the other as the metrics repository.

[6] Virtual Machines

The communication between the virtual machines is performed entirely within this physical machine without requiring an actual network, and only consist in metrics being sent from the processing VM to the metrics VM. Apart from that, there are only two data flows: one arriving to the NETS system from the sources and the other is the outgoing alarms generated by the NETS system that are sent to the Sistema Geral de Alarmes (SGA)/Web-server.

Regarding fault tolerance, the physical machine has no mechanisms in use. There is no auxiliary power source, redundant network access or even hardware replication / backups.

The failure of the physical machine or the processing VM halts the system and nullifies the system's work and objective.

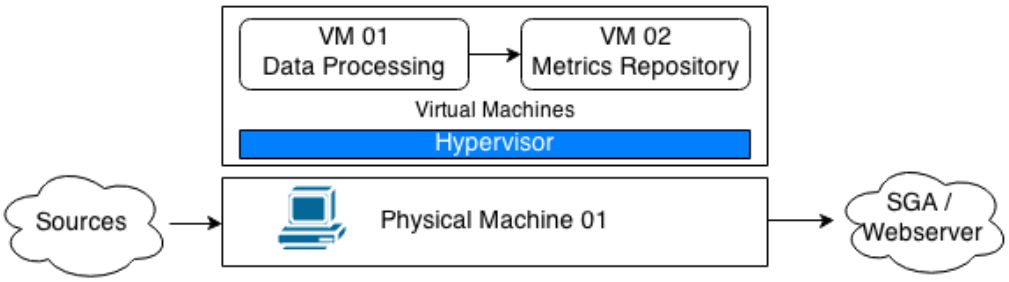

Figure 3.1: NETS Physical View.

3.3.2 Software View

The software architecture is illustrated in Figure 3.2, and the following description has key points related to this image in order to aid in its visualization.

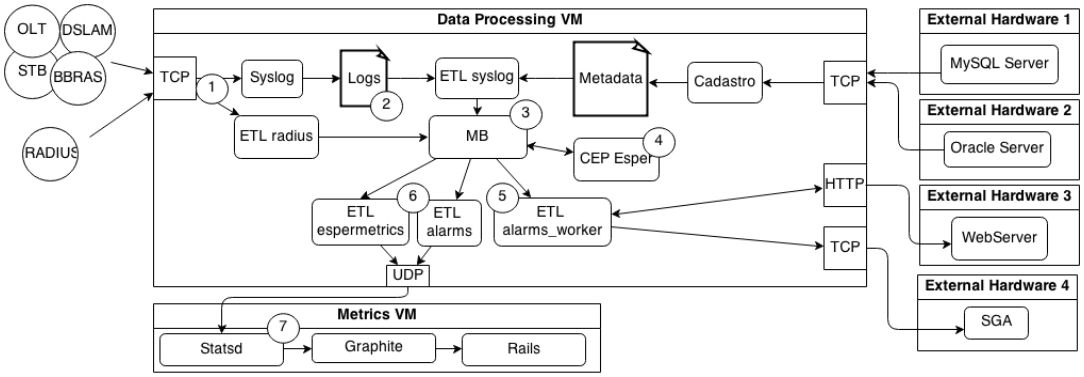

Figure 3.2: NETS Software Architecture.

The virtual machine responsible for data processing receives data via TCP (1), they have two distinct types: one according to syslog's standards which are stored in log files,

and the other type is relative to RADIUS messages. Both types of data are managed by the ETL[7] application, logstash, described in Section 3.2.

The ETL application is responsible for extracting the data from outside sources (log files) (2), transforming it to fit operational needs and send the result to the message broker (3), RabbitMQ, which in turn, communicates with the event processing engine (4), Esper, responsible for correlating events and alarms generation. In the end, the ETL application is responsible for sending the generated alarms to an external system (5), general alarms system (SGA - Sistema Geral de Alarmes), and related statistical information to the metrics repository (6) which is housed in the second virtual machine (Figure 3.1).

The metrics repository handles all the statistical data and measurements (7), which in turn can be visualized through graphical tools and a web back-end, enabling a constant monitoring of the current and historical states of the system.

The analysis of the NETS system life-cycle revealed the existence of three distinct phases, namely the Inputs phase, the Processing phase and the Outputs phase. Those phases are highlighted in Figure 3.3 along with the data flow within the data processing virtual machine. The remainder and complementary UML[8] diagrams can be consulted in Appendix A.

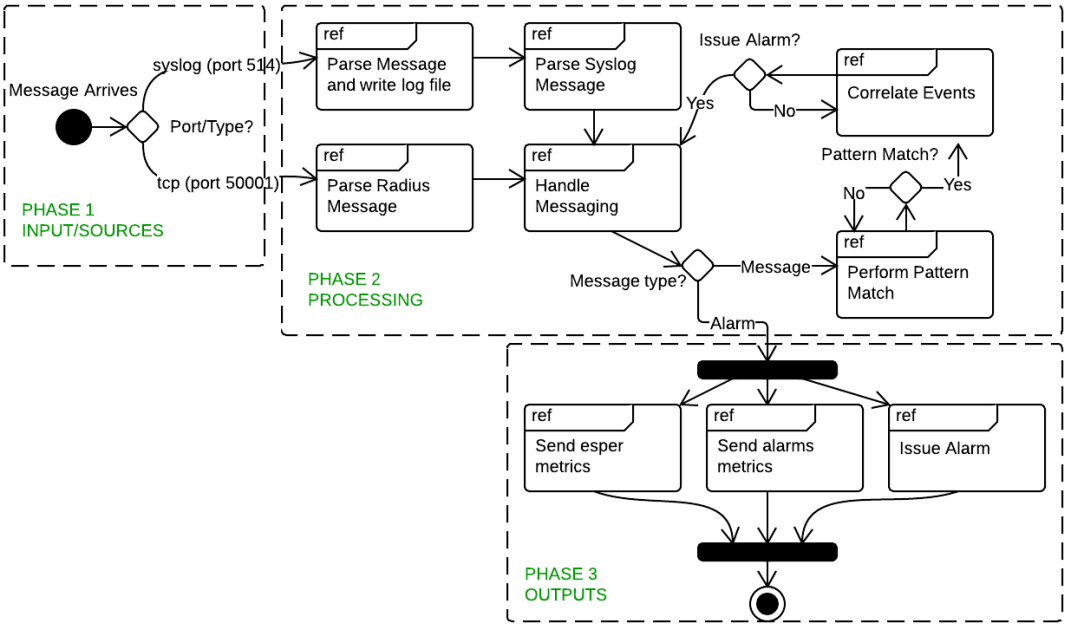

Figure 3.3: NETS Operation Phases.

[7]Extract, transform, load
[8]Unified Modeling Language

3.3.3 First Phase - Inputs

The **first phase** is mainly composed by receiving events sent by network devices through Syslog and TCP. These events have soft real-time requirements (Chapter 2.3) given that there are limited bounds to their use and validity, and they can be of five different types which are detailed in Table 3.1 along with their expected rate of arrival and description.

The events arriving by TCP go directly to ETL radius, whereas the ones received by Syslog are written do disk files, which are accessed by ETL syslog to parse them.

Type	Description	Syslog	TCP	Rate (events/s)
DSLAM	Digital subscriber line access multiplexers	X		20
OLT	Optical Line Terminals	X		5
BBRAS	Broadband remote access servers	X		50
STB	Set-top boxes	X		0.5
RADIUS	Remote Authentication Dial In User Services		X	150
		Total		225.5

Table 3.1: NETS Source types and volumes.

3.3.4 Second Phase - Processing

The **second phase** is comprised by **events processing** and **alarms generation**. Events are received, transformed and correlated to generate alarms in pre-configured cases. The process is explained in detail below, with the steps taken by each application.

- **Cadastro**

  Runs once a day to build client base information, writing the resulting meta-data to disk files.

- **Extract Transform Load (ETL) – logstash**

  ETL syslog parses messages from log files written by Syslog extracting relevant information to be sent to MB RabbitMQ using the meta-data file previously written by Cadastro to resolve the client's unique identifiers.

  ETL radius parses TCP messages extracting relevant information to be sent to MB RabbitMQ.

  ETL alarms and espermetrics send statistical information to the metrics repository.

  ETL alarms_worker sends generated alarms to the SGA and complementary information to the external web-server.

- **Message Broker (MB) – RabbitMQ**

The message broker serves as the main connection between components, behaving as a consumer from phase 1 processes, receiving parsed messages from ETL syslog and ETL radius, and as a producer for phase 2 processes, producing messages to be consumed by ETL espermetrics, alarms and alarms_worker.

- **Complex Event Processing (CEP) – Esper**

  Esper's sole responsibility is to correlate the received events, generating alarms under pre-configured circumstances, i.e., a client that is constantly connecting and disconnecting or several clients breaking up at roughly the same time and location will raise an alarm.

3.3.5 Third Phase - Outputs

The **third** and last **phase** is the **output** of the generated alarms. Alarms are divided in two parts. One that is sent to the SGA[9] through TCP, with the basic information required to initiate corrective measures. The second part is sent to an external web-server, by HTTP, with the evidences that support and justify the generation of the alarm itself and, this information is kept for future reference and consultation.

Parallel to this operation, statistical data is sent to the metrics repository for visualization and system monitoring.

With this information and having identified the dependencies between components, we designed diagrams that represent the software architecture (shown in Figure 3.2) and the operation phases (Figure 3.3) of the NETS system with the relationships and dependencies between components.

3.4 Fault Modeling

Fault modeling helps to identify targets for testing, making the analysis possible through experimentation and simulating modeled cases. After producing the architecture diagram, we started modeling the possible failure cases in the system's components.

3.4.1 Stream of Events

Assumption/Requirement: There is a constant and regular stream of events.

Fault: The rate of arrival is higher than the regular load.

Fault: The rate of arrival is null.

[9] Sistema Geral de Alarmes

3.4.2 Syslog

Assumption/Requirement: Syslog is always active, handles several sources of events, writing received events to disk in the form of log files.

Fault: Syslog crashes

Fault: Syslog corrupts messages

Fault: Lack of Disk Space

Fault: Log files are deleted/corrupted

3.4.3 ETL syslog

Assumption/Requirement: ETL syslog is always active

Fault: ETL syslog crashes

Assumption/Requirement: ETL syslog receives a constant stream of events

Fault: Load is above regular level

Fault: Load is below regular level or null

3.4.4 ETL radius

Assumption/Requirement: ETL radius is always active

Fault: ETL radius crashes

Assumption/Requirement: ETL radius receives a constant stream of events

Fault: Load is above regular level

Fault: Load is below regular level or null

3.4.5 Cadastro

Assumption/Requirement: Cadastro is generated once a day, and serves as a database for client information and to cross reference data received in events to identify uniquely the respective client.

Fault: Lack of Disk Space

3.4.6 MB RabbitMQ

Assumption/Requirement: RabbitMQ is always active, serving as the central message passing component, having a bridge like function linking the several components together, from ETL to CEP and back.

Fault: MB RabbitMQ crashes

Fault: MB RabbitMQ high load

Fault: Lack of Disk Space

3.4.7 CEP Esper

Assumption/Requirement: CEP Esper is always active and handles the event correlation to generate relevant alarms under specific conditions, it is a critical component due to being the sole responsible for the alarms generation.

Fault: CEP Esper crashes

3.4.8 ETL espermetrics and ETL alarms

Assumption/Requirement: ETL espermetrics and ETL alarms are always active and are consumers of MB RabbitMQ, processing statistical data and sending it to the metrics repository.

Fault: ETL espermetrics/alarms crashes

3.4.9 ETL alarms_worker

Assumption/Requirement: ETL alarms_worker is active and is responsible for sending the generated alarms to the SGA where they will be handled by operational teams. The details and link of alarms' information is sent to the external web-server.

Fault: ETL alarms_worker crashes

Fault: The SGA or the external web-server is not active

3.5 Fault Testing

Fault testing is performed based on the modeling made in the previous section. Faults were injected in the NETS system to simulate the described cases in order to register the behavior and consequences in each situation.

3.5.1 NETS Normal State With Regular Load

The system's state under regular load was outlined in order to have a point of comparison. In Figure 3.4 is shown the amount of messages received which is in accordance with the information already provided in Table 3.1, i.e., there is an average of 225-240 messages per second arriving to the system. This value has an expected fluctuation directly related to the time of day, e.g., at night the number of events is lower than in the morning or noon. In Figure 3.5 we detail the use of resources under regular load, i.e., the amount of CPU and memory used by each process.

A constant flow of events is expected as a natural consequence of network activity, both considering normal user actions and abnormal activities.

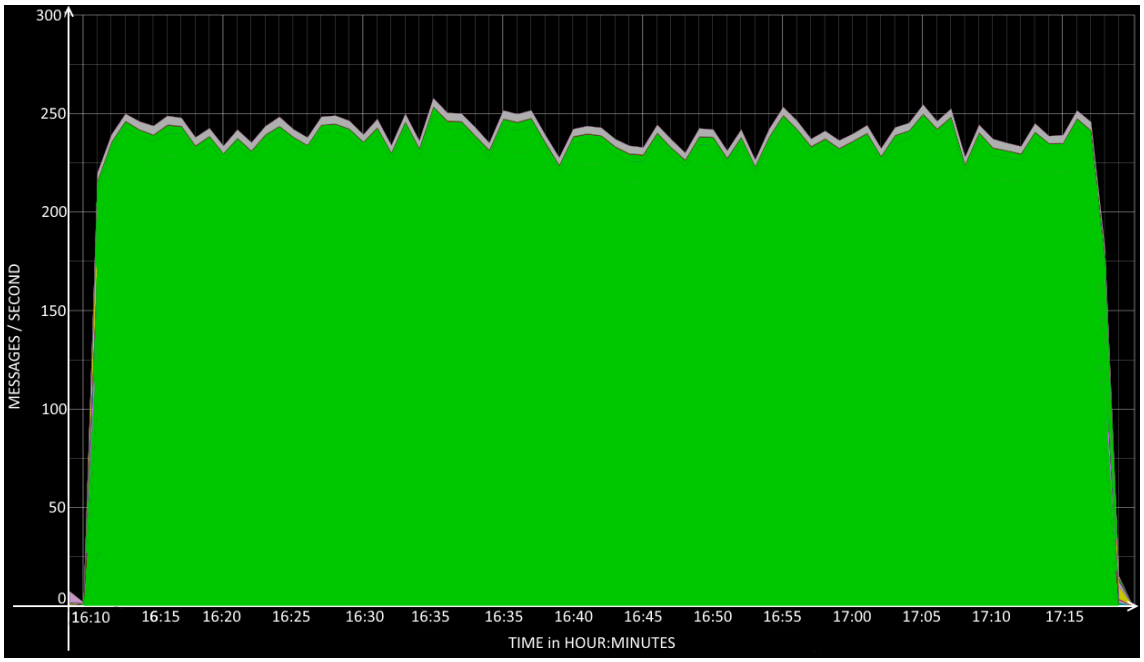

Figure 3.4: NETS regular load.

System	Status	Load	CPU	Memory	Swap
nets-vm-01	Running	[0.00] [0.00] [0.00]	3.7%us, 7.1%sy, 0.0%wa	27.1% [4439424 kB]	0.0% [0 kB]
Process	**Status**	**Uptime**	**CPU Total**		**Memory Total**
syslog-ng	Running	18h 2m	0.1%		0.0% [2480 kB]
rabbitmq	Running	18h 2m	2.2%		0.6% [100508 kB]
etl-syslog	Running	18h 2m	4.5%		3.0% [503092 kB]
etl-radius	Running	18h 2m	6.3%		2.6% [437732 kB]
etl-pg_worker	Running	18h 2m	0.4%		0.1% [22788 kB]
etl-espermetrics	Running	18h 2m	0.0%		2.4% [395728 kB]
etl-alarms_worker	Running	18h 2m	0.0%		0.1% [20108 kB]
etl-alarms	Running	18h 2m	0.0%		2.4% [395592 kB]
esper	Running	18h 1m	3.8%		14.1% [2318848 kB]

Figure 3.5: NETS regular load stats.

3.5.2 Stream of Events - Load Simulation

The initial tests were performed in order to explore and perceive the NETS system's load capacities. By simulating additional load in each component and the whole system, we can draw conclusions regarding their current capacity and future viability.

It is noteworthy to say that in this test we check for failures of all types, i.e., crash faults if any component fails due to the additional load, omission faults if there is loss of messages, queue clogging, network, etc., timing faults if the components are not able to process requests in a timely manner resulting in loss of messages, syntactic and semantic faults, given the extra volume and high processing requirements which may lead to invalid messages.

Test DSL

The evolution and behavior of the component when it was submitted to a performance test is illustrated in Figure 3.6.

This component handled a load of up to 300-330 requests/s without any signs of distress. When submitted to loads superior to that limit means that an increasing number of messages begin to be discarded due to being processed as out of time.

The complementary details are presented in Appendix B.

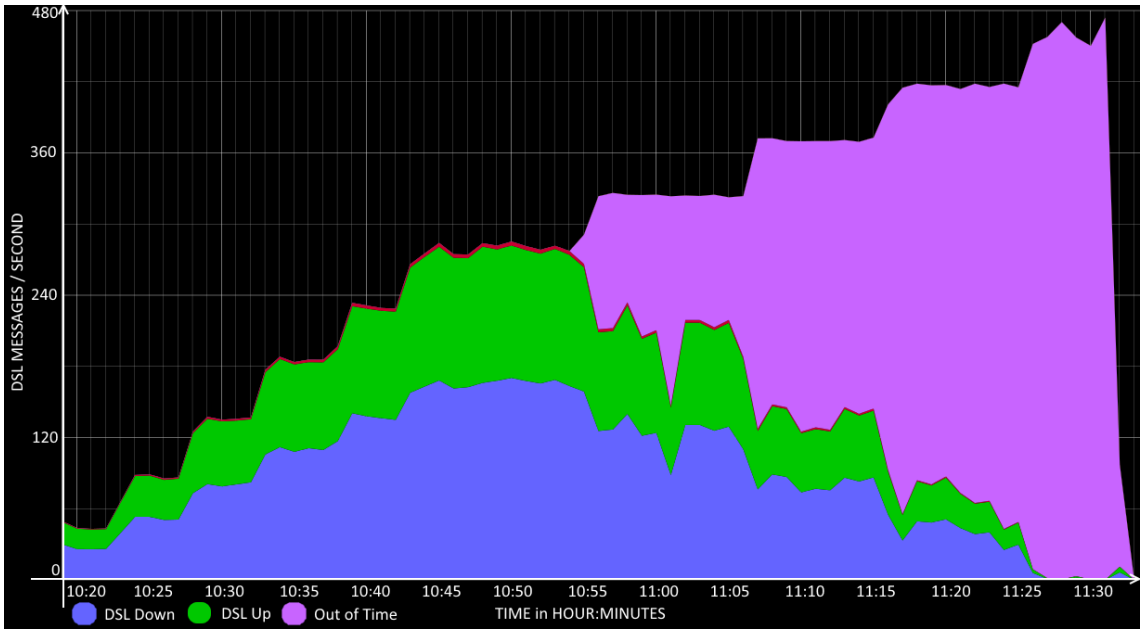

Figure 3.6: Test - DSL.

Test STB

This component has a similar behavior to DSL, as it is roughly the same process and methods performing the message filtering. It is capable of handling a load of up to 300 requests/s, but higher loads will translate in an increasingly number of messages being discarded due to being processed as out of time.

The evolution can be seen in Figure 3.7 and the details in Appendix B.

Test RADIUS

The RADIUS handler has no problematic situations up to a load of 1050-1155 requests/s. After that mark some messages are lost due to being processes as out of time. This is the component that receives the highest share of requests, but it is also the one with greater processing capabilities.

The evolution can be seen in Figure 3.8 and the details in Appendix B.

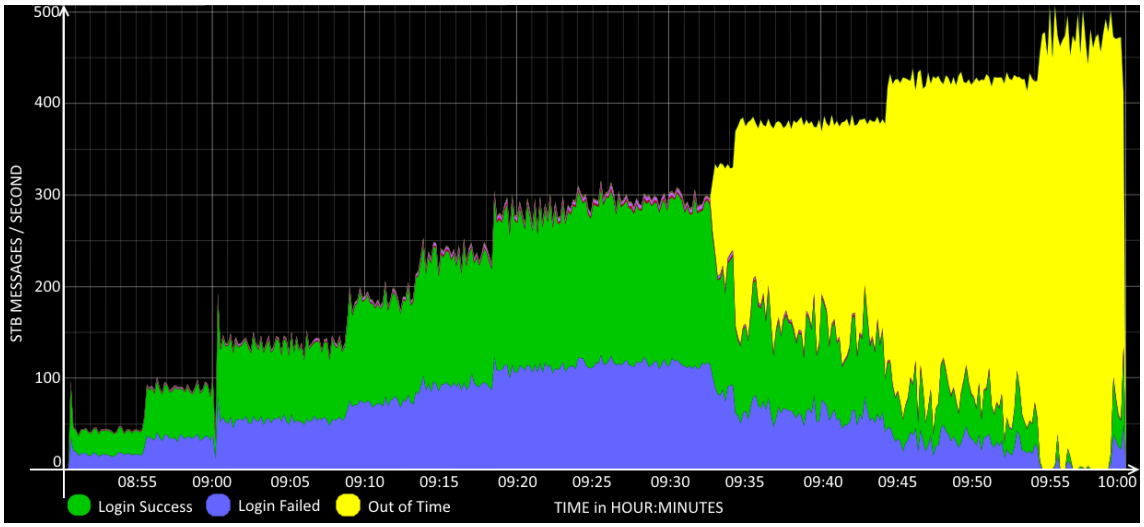

Figure 3.7: Test - STB.

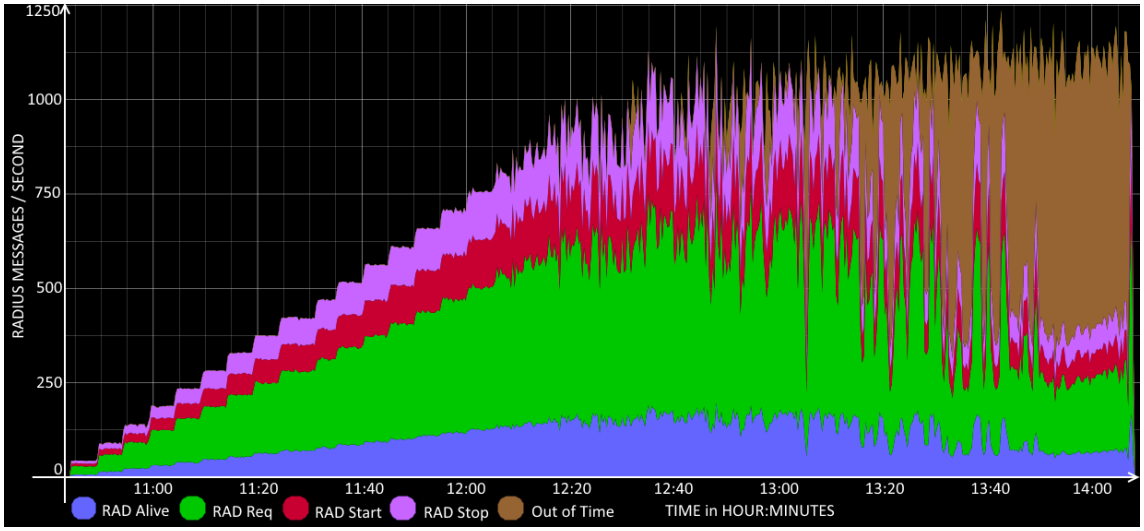

Figure 3.8: Test - RAD.

Test ALARMS (Using DSL-FLAP)

There are several patterns configured in CEP Esper that can trigger alarms' generation. One of these alarms is related to the DSL behavior of the clients. When a client is continuously connecting and disconnecting suggests a problem with its connection. This pattern is called "DSL flapping" and was used as base to test the alarms' processing capacity.

The component responsible for the alarms' handling can take up to 10-12 requests/s before the message broker starts queueing them as it cannot handle such load.

The evolution can be seen in Figure 3.9 and the details in Appendix B. This image is composed by two charts because they are intimately related. The upper chart shows the number of queued messages in the message broker, the ideal situation is having zero queued messages. The bottom chart represents the rate at which requests are being pro-

cessed. In this figure it is observable that when the rate peaks its capacity, messages start to be queued and will be delivered in an unknown delay.

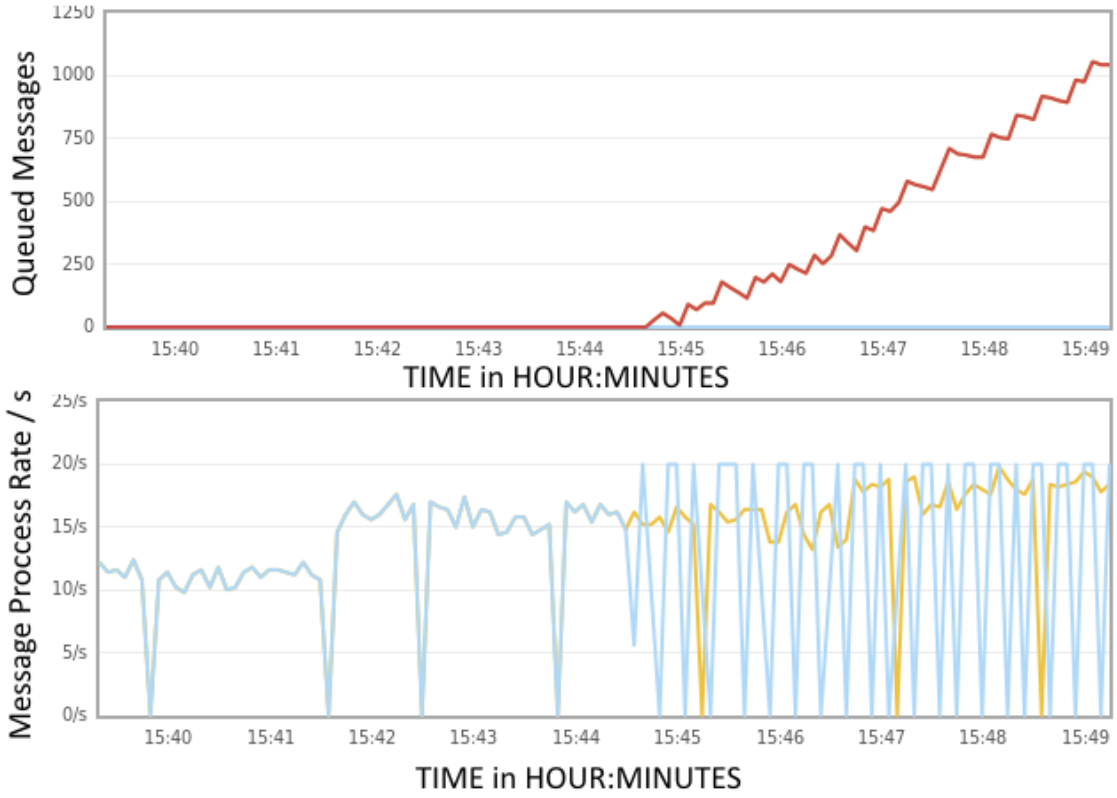

Figure 3.9: Test - ALARMS.

3.5.3 Stopped Stream of Events and Closed Ports

In this second test we focus our attention on omission faults, i.e., if events fail to arrive at the components for lack or loss of messages in the same network.

Stream of Events:

If there is no stream of events, the system simply waits for new messages to process without taking any action. When, for some reason, messages are not arriving successfully and the input is restored, there is a delay to process old messages that potentially generates an overflow of requests, in which there is an initial period likely to contain a large number of out of time messages, taking a few minutes to recover depending on how long and how many messages were sent in downtime.

Individual components:

Within the NETS system every component waits for new messages to process. Messages exchanged within the NETS system are primarily managed by RabbitMQ that offers de-

livery confirmation mechanisms. These are not in use.

Generated alarms are sent by TCP to the SGA, and the alarms' details are posted through HTTP to the web-server acknowledging their successful reception.

The metric's data is sent over UDP because the loss of some of these messages is not critical, but there are no mechanisms that detect this situation, temporary or permanent, e.g., if metrics are being sent to but not received by the metrics machine.

3.5.4 Crash Faults

Over the following tests the objective is to register the behavior when there is a crash fault on each individual component. The main goal is to determine the consequences for the system, its dependencies and if an automatic recovery is possible while measuring how long it takes.

Crash syslog-ng

In the event of a crash, messages sent by syslog will be accumulated and discarded after a set timeout. When and if the service recovers, some of the messages can be processed as out of time if syslog-ng takes too long to recover.

Syslog-ng has a supervisor, in practice the child is the main process, the supervisor is only there to restart it in the following cases: it was killed by a signal, it exited with a non-zero return value.

When shutting down syslog-ng, the child process needs to receive a TERM signal, which will exit with a zero return value and also brings away the supervisor process. Meaning that if syslog-ng crashes in certain ways it might not be restarted, e.g., "kill -9".

Crash Esper

When there is a crash, the state built with the past events kept in memory is lost. Esper is unable to retain the persistence / consistency in its base version, but it is available in a separate package, EsperHA.

Given this situation, some of the alarms can be lost or never issued because the history kept in memory is lost, limiting the correlations made.

While Esper is unavailable alarms are not issued, consequently, the components responsible for sending the alarms to the SGA, alarms and ETL-etl-alarms_worker, are without work, meaning that *PHASE 3 outputs* stops completely.

Automatic recovery managed by monit.

Crash etl-syslog

Syslog's related messages stop being processed, and there is an increased risk for them to be lost while being treated as out of time messages depending on how long it takes for the

process to recover.

After the process recovers, there is a period of time taken to catch up because there are queued messages waiting to be processed, directly related to how long the process was unavailable.

Automatic recovery managed by monit.

Crash etl-alarms

The statistical information regarding issued alarms stops being sent to the metrics repository. It is considered not critical.

The generated alarms stop being sent to the SGA and to the external web-server, those messages are held by the message broker until the service recovers.

These situations can be seen in Figure 3.10, where, after the crash, a large number of messages is kept on the RabbitMQ's queue. This is followed by some time without any alarms being issued, and then the recovery of the process is made. The queued alarms are then processed and sent to the SGA as usual.

Automatic recovery managed by monit.

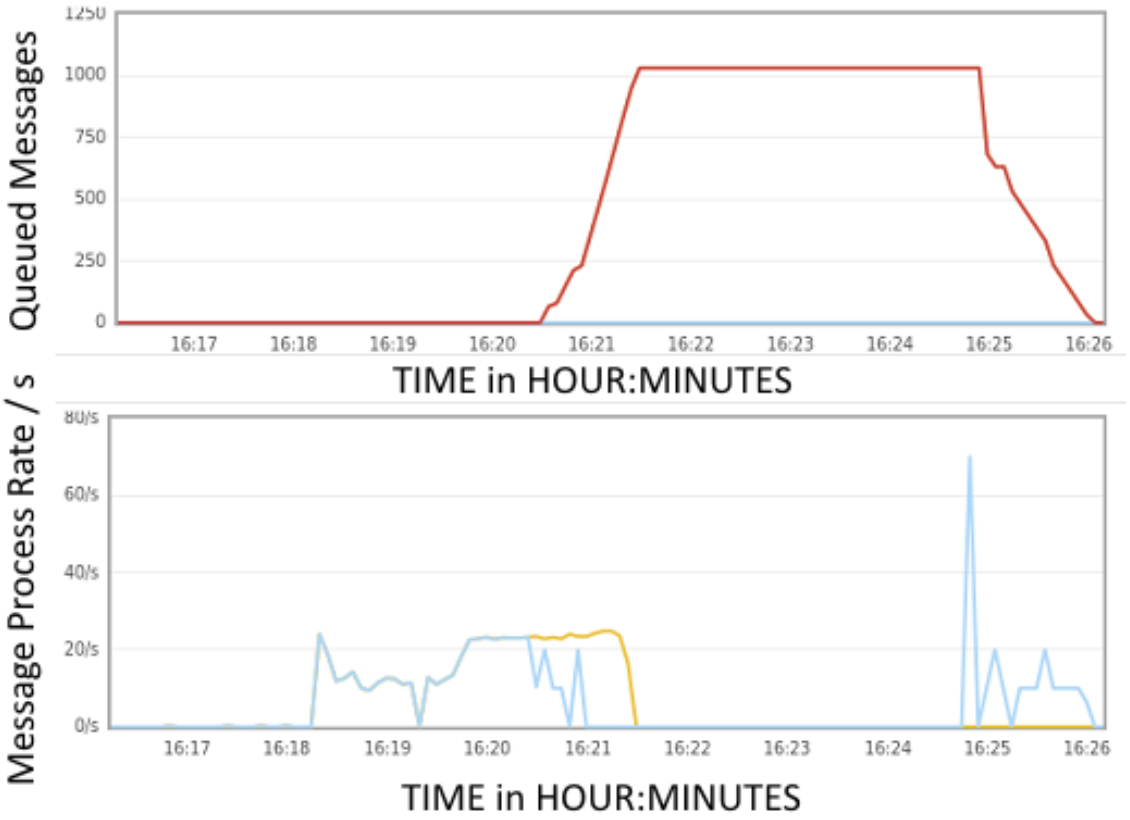

Figure 3.10: Test - ETL Alarms.

Crash RabbitMQ

In the event of RabbitMQ crashing, it loses all the information in memory (messages, queues and exchanges). With it, Esper also crashes, and cannot perform an automatic recovery due to not having all the exchanges declared, meaning that it keeps crashing and rebooting until there is a manual configuration of the missing queues in RabbitMQ.

Crash etl-alarms_worker

If this etl crashes, etl-alarms will also crash and the generated alarms cannot be sent to the SGA and to the external web-server. These messages are kept by the message broker until the service recovers.

Automatic recovery managed by monit.

Crash etl-espermetrics

The statistical information regarding Esper's activity stops being sent to the metrics repository. It is considered not critical.

Automatic recovery managed by monit.

Crash pg_worker

If there is a crash fail in pg_worker, etl-syslog also crashes. Therefore, the consequences are similar having an increased recovery time because two processes have to recover instead of one.

Automatic recovery managed by monit.

Crash etl-radius

Radius related messages arriving at the system stop being processed. Instead, they are queued up for a configured amount of time before they are discarded.

There is an increased chance of losing these messages by being processed as out of time depending on the time taken for the process to recover.

Automatic recovery managed by monit.

3.5.5 File Deletion/Corruption

In this test we take a closer look into assertive faults, both syntactic and semantic, and omission faults, but specifically in files.

To this end, we check what happens when files are deleted or become corrupt, and the behavior of each component when receiving corrupted or invalid messages.

1. File Deletion

- Logs at /var/logs/nets/etl

  No problems detected in the system performance, only the logging is affected as no new lines are written until the processes are resetted or the log file is rotated.

- Logs at /var/logs/nets/esper

  No problems detected in the system performance, only the logging is affected as no new lines are written until the processes are resetted or the log file is rotated.

- Logs at /srv/syslog-data/*

  From the moment a file is deleted, the events and data stream is lost and no longer registered, i.e., messages are not processed by NETS. This situation is normalized if the syslog-ng process is restarted or there is log rotation when the hour passes, forcing the opening of a new file or descriptor.

2. Corrupt Messages

   No problematic situations were identified regarding corrupt messages arriving to the system and files used by NETS, because they are filtered by rigorous pattern matching performed by the ETL applications and parsed as invalid messages.

3.5.6 Lack of Disk Space

In this test we check the system's behavior when disk space becomes scarce or null, if it originates crash faults, omission faults, etc. In order to perform this test, the disk was purposely filled until there was a small amount of space left, and then we let the remainder to be filled up by the system's normal operation.

The following events were recorded as a result of the system's behavior in face of the diminishing disk space:

1. First, RabbitMQ starts doing flow control which blocks connections. There are two types of flow control used, both work by exerting TCP back-pressure on connections that are publishing too fast. They are:

   - A per-connection mechanism that prevents messages being published faster than they can be routed to queues.
   - A global mechanism that prevents any messages from being published when the memory usage exceeds a configured threshold or free disk space drops below a configured threshold.

   Both mechanisms will temporarily block connections - the server will pause reading from the sockets of connected clients which send content-bearing methods (such as

basic.publish) which have been blocked. The intent is to throttle producers but lets consumers continue unaffected. Connection heartbeat monitoring will be disabled too.

2. If nothing is done, logging applications will consume the remaining free space. By then, logs will cease to be written and requests stop being processed by syslog/etl-syslog/etl-radius.

3. Cadastro cannot be written without enough disk space, meaning that Esper is left without its database and etl-syslog cannot verify the clients identity in incoming messages or it would do it in an out of date list.

4. The NETS system stops working completely.

At this point, recovery is possible after releasing disk space, rebuilding cadastro if necessary and resetting the affected components or the hour changes because of file descriptors/open files situation.

3.5.7 Bottlenecks

In this test we searched for potential bottlenecks in the NETS' processing, taking into account the time spent in each component. The objective is to identify the components that are most prone to delay the process and find ways to improve their performance to avoid timing faults.

The only component that was identified to be a significant concern was etl-alarms_worker.

For the analysis of etl-alarms_worker, small changes were made to its source code, adding log activity in key points enabling us to determine the time taken in each relevant action and the total time taken processing one message.

This resulted in the creation of an auxiliary script to perform the data gathering and processing to create the required statistics. It produced the following results:

Results:

Number of samples: 1450

Mean Time of first post: 41.442295

2 outliers: [3037.799835, 133.379936]

Mean Time of second post: 46.744604

0 outliers: []

Mean Time for socket: 1.041185

13 outliers: [4.679918, 4.669905, 6.410122, 3.959894, 4.499912, 4.950047, 4.119873, 3.760099, 3.289938, 3.540039, 3.530025, 3.250122, 3.299952]

Mean Total Time: 89.532596

Time is in milliseconds, average of about 90ms for each request, showing us that most of the time taken is used to make HTML posts.

We conclude that its processing capacity is limited at roughly 11 requests per second, given the mean total time for each request. Loads greater than this will result in alarms being queued and waiting to be processed for an undetermined time.

The closing of already open alarms also generates load and traffic. This can happen when an alarm is treated and manually closed, or there is an automatic closing given that the situation solved itself naturally or ceased to exist.

3.6 Risk Analysis

An important part of any system is having a good understanding of the possible risks and how to manage them in the case of their occurrence. There is a variety of risks not worth mentioning either for their extremely low probability or irrelevance to the system, therefore, we will try to summarize the ones that are important and/or relevant to the NETS system.

In Figure 3.11 we classify risks taking into account their probability and severity, enabling us to draw a line between the risks worth having a contingency plan and those who are not. For this system we are taking into account those who are considered ”High Risk”.

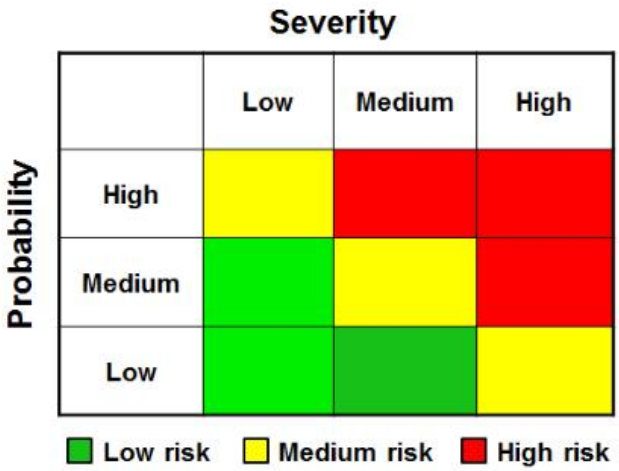

Figure 3.11: Risk Management Priority.

As a result of this analysis and in accordance with the tests described in the previous section, we have produced Table 3.2, in which it is visible the possible threats to the system’s dependability.

Having a risk table, we are left to produce a contingency plan to prevent, monitor and handle the occurrence of each risk with ”high” threshold. For that, we will produce an

#	Type	Probability	Severity	Risk
1	Higher rate of events in syslog	Medium	Medium	Medium
2	Higher rate of events in etl-syslog	Medium	Medium	Medium
3	Higher rate of events in etl-radius	Medium	Medium	Medium
4	Null rate of events in syslog	Low	High	Medium
5	Crash Syslog	Low	High	Medium
6	Syslog corrupts messages	Low	High	Medium
7	Lack of disk space	Low	High	Medium
8	Crash etl-syslog	Low	High	Medium
9	Null rate of events in etl-syslog	Low	High	Medium
10	Crash etl-radius	Low	High	Medium
11	Null rate of events in etl-radius	Low	High	Medium
12	Crash Esper	Low	High	Medium
13	File deletion/corruption	Low	Medium	Low
14	Crash RabbitMQ	Low	Medium	Low
15	Crash etl-alarms	Low	Medium	Low
16	Crash etl-alarms_worker	Low	Medium	Low
17	Crash etl-espermetrics	Low	Low	Low

Table 3.2: NETS Risk Analysis.

RMMM plan (Risk Management, Mitigation and Monitoring) for the risks that exceed the specified threshold if needed.

According to Table 3.2 the threat to the system is relatively moderate. The most concerning risks are related to the arrival rate of events and the crash of the processes that handle their reception and treatment. Both these cases have been modeled and will be addressed when presenting possible solutions to improve the system's dependability.

3.7 Summary

In this chapter the tools used in the system were presented along with the system's architecture and details of its workflow. Performance tests served both to realize the processing limitations and to verify the system's behavior in anomalous and/or extreme cases. And finally, through fault modeling and the tests made based on the analysis, we produced a risk analysis for NETS system.

In the following chapter we will present solutions in order to improve the system's dependability.

Chapter 4

Solution Design

Pursuant to the presented in the previous chapter, several proposals and possible solutions are presented in this chapter in order to improve the system's dependability, as well as a review of their advantages and disadvantages.

4.1 Proposals for individual component enhancements

During the analysis we identified several components that can be improved in order to increase their reliability and/or performance. The implemented solutions will be specified in Chapter 5 providing the implementation's details.

4.1.1 Stream of events

- Incoming Stream of Events

  Given that the incoming message flow is expected to be continuous and nonzero 24h per day, the implementation of an alarm if no messages are received in a defined time frame could help speed up recovery measures. Otherwise, the complete omission of events can be visually noticed in the NETS dashboard.

  Another concern is the availability of the ports used by the NETS system. There is the possibility of manually checking their status or using an application to monitor them.

  One way to check if a particular port is open:

  nc -z -w5 <host><port>; echo $?

  It tries to make a connection to the specified host and port and returns "0" if it is successful, or "1" if it cannot establish the connection.

- Messaging between components - RabbitMQ's Message Acknowledgments

Since networks are unreliable and applications fail, it is often necessary to have some kind of processing acknowledgment. Sometimes it is only necessary to acknowledge the fact that a message has been received. Sometimes acknowledgments mean that a message was validated and processed by a consumer, for example, verified as having mandatory data and persisted to a data store or indexed.

This situation is very common, AMQP has a built-in feature called message acknowledgments (sometimes referred to as acks) that consumers can use to confirm message delivery and/or processing. If an application crashes (the AMQP broker notices this when the connection is closed), if an acknowledgment for a message was expected but not received by the AMQP broker, the message is re-queued (and possibly immediately delivered to another consumer, if any exists).

4.1.2 Crash syslog-ng

A simple improvement that can increase syslog-ng's dependability is by adding it to monit along with the other processes that are already monitored in the NETS context.

The only modification needed is the creation of a monit file with syslog-ng's related rules and information regarding the file that maintains the pid of the current syslog-ng process.

4.1.3 Crash Esper

If there is the need for fail-over and/or recovery capability, then EsperHA (Esper High Availability) can be an option. EsperHA is a complete solution for zero-downtime CEP and event series processing. It combines Esper with high performance resilience options.

It uses incremental backup that only write state changes. Recovery algorithms read the newest state up to the latest state since last checkpoint and never replay or read old unnecessary state. EsperHA supports a full-backup-only mode. Throughput in events-per-second with resilience is very close to throughput without resilience [17].

There is a monetary concern in the adoption of this additional package, as it might not be mandatory or even necessary depending on the final system architecture.

4.1.4 Crash RabbitMQ

Similarly to syslog-ng, RabbitMQ's state can be monitored by monit, only requiring the addition of its rules and details to the monit's configuration.

One consequence of RabbitMQ's crash is Esper crashing along with it, but Esper can recover if there is a cluster of RabbitMQs, instead of losing everything, information about the exchanges and queues survive given that there is one correct machine in the cluster, enables Esper to recover automatically.

Additionally, queues and exchanges are configured as transient but can be declared as durable, which would prevent their disappearance when Esper crashes or restarts.

A durable queue stores messages if there are no connected consumers available to process the messages at the time they are published.

An exchange does not store messages. They can be marked as durable but all that really means is that the exchange itself will still be there if the broker restarts, but it does not mean that any messages sent to that exchange are automatically persisted.

From RabbitMQ documentation [20]:

- Durability (exchanges survive broker restart)
- Auto-delete (exchange is deleted when all queues have finished using it)
- Arguments (these are broker-dependent)

Exchanges can be durable or transient. Durable exchanges survive broker restart whereas transient exchanges do not (they have to be redeclared when broker comes back online).

4.1.5 File Deletion/Corruption

First, we need to understand how files and handlers work in Linux, which is the operating system used in the NETS system.

If the file is moved (in the same file-system) or renamed, then the file handle remains open and can still be used to read and write the file.

If the file is deleted, the file handle remains open and can still be used. The file will not really be deleted until the last handle is closed.

If the file is replaced by a new file, it depends exactly how. Either the file is overwritten, in which case the file handle will still be valid and access the new file, or the existing file is unlinked and a new one created with the same name, in which case it is the same as deletion (see above).

In general, once the file is open, the file is open, and nobody changing the directory structure can change that - they can move, rename the file, or put something else in its place, it simply remains open.

In Unix there is no delete, only unlink(), which makes sense as it does not necessarily delete the file - just removes the link from the directory.

One possible solution for the case in study is reloading syslog, forcing the service to read its configuration file again. This operation re-creates the files / links and everything goes back to normal. Files and logs that were deleted are re-created and work resumes.

Possible solution:

”sudo service syslog-ng reload” or ”sudo killall -HUP syslog-ng”

The only concern is how we can detect the need to do the reload.

4.1.6 Lack of Disk Space

Disk space is monitored by RabbitMQ, avoiding the need to implement additional mechanisms to do this task, although monit is also able to perform it easily. We could make a temporal prediction of disk space exhaustion using the already available metrics of free and used disk space.

Another detected situation is frequent log writing by Esper. The following lines are added to ”engine.log” and ”audit.log” whenever Esper makes a new connection:

2014-01-08,16:18:51.226,INFO,com.espertech.esper.epl.db.ConnectionCache,.makeNew Obtaining new connection and statement

2014-01-08,16:18:51.227,INFO,com.espertech.esper.epl.db.ConnectionCache,.close Closing statement and connection

This is embedded in Esper’s source code and, currently, there is nothing that can be done to prevent this from happening besides editing the source and recompiling the whole Esper’s jar file. We concluded that this procedure was unnecessary as there are plans to reduce the frequency of connections and the benefits are not significant.

4.2 Redundancy Analysis

In this section we will analyze if byzantine fault tolerance is needed, the challenges of common-cause and common-mode failures, and solutions to provide input and output safety and redundancy guarantees.

We will also review a number of architectures as possible candidates implementable in the NETS system, describing their details and comparing their advantages and disadvantages.

4.2.1 Byzantine Fault Tolerance / Arbitrary Faults

Byzantine fault tolerance was not an absolute necessity in NETS given that the system is not critical and its data output does not have catastrophic consequences if it fails for some reason.

Either way, the misbehavior of a component should be detected and adequate action performed to restore the system to a correct working state. If the fault is known, specific measures targeting that very fault should be used, but in the event of an arbitrary fault it should be enough to detect it and restart the component in order to correct its state.

How it could be done

When a byzantine fault occurs, the system may respond in any unpredictable way, unless it is prepared to have byzantine fault tolerance. These faults can range from invalid input being fed into the NETS system to alarms being erroneously produced in the inner data processing.

As seen on Lamport's byzantine generals problem [31], a possible approach would be adding processes to the system, i.e., a solution could be found with a system with n >= 3t + 1, where n is the number of the processes in the system and t is the number of faulty processes. A correct operation can be ensured if the faulty processes represent less than one third of the total.

These processes would work with each other, agreeing on the messages received and outputs produced, enabling voting on each generated message. A practical example would be a cluster (Figure 4.1) in NETS where all the machines process the flow of events and generate alarms. These alarms would be subject of a voting to ensure that the produced alarm is indeed correct. The inputs received by the NETS system could be treated in a similar fashion.

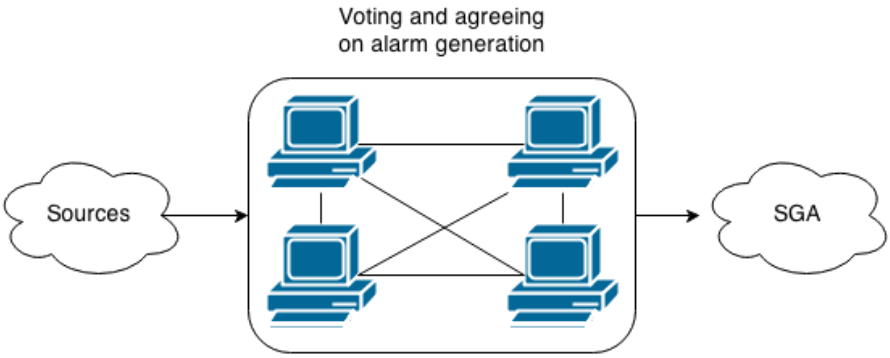

Figure 4.1: BFT Architecture.

4.2.2 Common-Cause / Common-Mode Faults

The NETS system's stream of events is sent from a small and restricted number of IP addresses, but can change throughout time. These machines are configured to send the event's data to a specific IP address corresponding to a NETS machine.

The data processing machine in the NETS system receives two types of messages that pass through syslog-ng's filter, and are discarded if they are not according to the template.

After this initial filter, the respective ETL (radius / syslog) applies another filter that checks the syntax and values of the received message. If the message is valid and has a proper client number it is forwarded to the message broker for processing, otherwise the message is discarded.

In the tests made (described in the previous section), no problematic cases were identified. In the event of an invalid input leading to the crash of the etl or syslog, this situation is promptly detected and the process is restarted in a matter of seconds and continues to work regularly with minimal impact to the system.

A failure in these components is not considered serious given their swift recovery. The machine itself does not crash because of these processes.

The use of diversification (described in section 2.4.5) in the NETS system may be conditioned due to the cost of producing this type of mechanism. The severity and probability of occurring versus resilience / need to run 24/7, means that a simpler but sufficiently reliable and stable system is preferable.

As seen in the literature [6], this type of system is generally used in NASA or in building nuclear reactors. In these cases the failure may have catastrophic consequences, which leads to mandatory prevention and fault tolerance in every component, including Byzantine faults.

4.2.3 Input Handling

As we have just described in the previous section, the stream of events comes from several sources. Given the nature of some of the equipments used, it is only feasible to configure these sources to send their information to one IP address.

We will now describe some of the options available for adding fault tolerance to the arriving stream.

Port Mirroring

Port mirroring is used on a network switch to send a copy of network packets seen on one switch port to a network monitoring connection on another switch port. This is commonly used for network appliances that require monitoring of network traffic, such as an intrusion detection system, passive probe or real user monitoring technology.

Network engineers or administrators use port mirroring to analyze and debug data or diagnose errors on a network. It helps the administrator keep a close eye on network performance and will alert them when problems occur. It can be used to mirror either inbound or outbound traffic (or both) on single or multiple interfaces.

In the NETS system specific case, port mirroring could be used to deliver the same stream of events to two different physical machines, effectively cloning it. Having the advantage that this would make the configured switch responsible for the distribution task, but being a low level hardware device, the probability of failure, or mean time before failure (MTBF), would be substantially lower than a regular server. This would also make this equipment a single point of failure in the NETS system.

An additional advantage is lacking the need of acquiring extra hardware, instead it requires reconfiguring an existing one.

Load Distributer

Another option involves an additional physical machine that would serve as a distributer. This node would receive the stream of events and propagate it to the remaining nodes of the NETS system.

The advantage of this approach is the fact that this machine would be a lighter and more reliable node in comparison to the remainders, making its probability of failure lower. But, either way, the cluster would depend on its correctness and availability, making it a single point of failure.

Dynamic IP

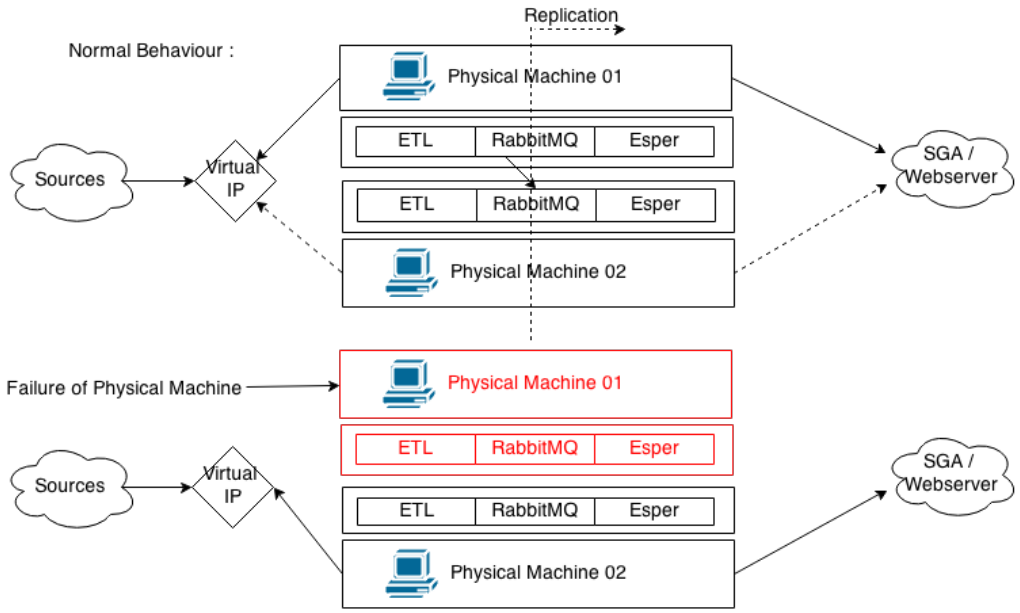

Figure 4.2: Dynamic IP.

Alternatively, given that the events are sent to a single IP address set to the primary machine on the NETS cluster, in the event of the crash or malfunction of this machine, that IP address is assigned to another correct machine, effectively taking its place as the primary (Figure 4.2). Only the ETLs of the primary node perform stream filtering, and we use RabbitMQ's mirrored queues to propagate the messages to the other cluster members in order to achieve an updated state regarding Esper.

This requires constant monitoring of the primary and a mechanism that can handle the necessary configuration changes when the IPs need to be changed.

The advantage is that no extra hardware is needed and all the configuration is done in NETS system's machines. Additionally, when one machine crashes, there is only a brief unavailability while the IP update is done, and the machine that took over should have the same or a very similar state to the primary before crashing.

4.2.4 Output Conciliation

Two options were considered: using a single output stream, assuming it is trustworthy, or using a more complex algorithm involving voting.

The alarms are generated taking into account the received events, if two machines receive the same stream simultaneously, they should produce exactly the same output. This situation is nearly impossible to guarantee, whereas a very similar or even the same result would be produced with less strict temporal limits.

There are some cases in which the output could diverge, i.e., a difference in the machine's clock could lead to missed alarms or different messages' order also leading to generating extra alarms or missing some.

These situations were discussed with the company's project manager and were considered to be acceptable. Losing some alarms is not critical, and most of them should not be affected because as they are detected by patterns that tend to repeat themselves, only leads to a slight delay in their detection.

Using a Single Output

The entire cluster processes data, but only the output of one machine is taken into account and sent to the SGA. In the event of a failure, one of the remaining machines takes over and starts sending its alarms to the SGA without delay.

With this solution, there can be a brief period, between detecting the failure of the current sender and switching that role to other machine, where alarms are not being produced. The alarms are not being produced but their loss can be avoided if we enable the acknowledgments option of RabbitMQ. This way, the alarms are sent but only erased when there is an acknowledge from the SGA.

Deciding the Output Through Consensus

We could implement a consensus algorithm where all the machines vote on the output. But it would generate additional overhead and tamper with the alarms' generation performance.

Having all the machines generating alarms and using a log or table to keep track of the issued alarms would provide a way to discard the already seen or sent alarms to the SGA.

4.2.5 Architectures

In this section we will present a number of architectures that were considered as potential options to be implemented in the NETS system. In each of them there will be an evaluation of the advantages and disadvantages of adopting each architecture.

The amount of detail poured in each architectural analysis will be related with its perceived viability and usefulness.

Option 1

The simplest architecture is composed by two physical machines to provide the minimum redundancy needed for tolerating a single fault.

Both machines receive the event stream from the sources and process the incoming messages, but only one of them produces an output, e.g., Esper processes and correlates events independently in each machine, but only the primary sends the generated alarms to the SGA.

This can be achieved with port mirroring described in the previous section and a representation of the architecture seen in Figure 4.3.

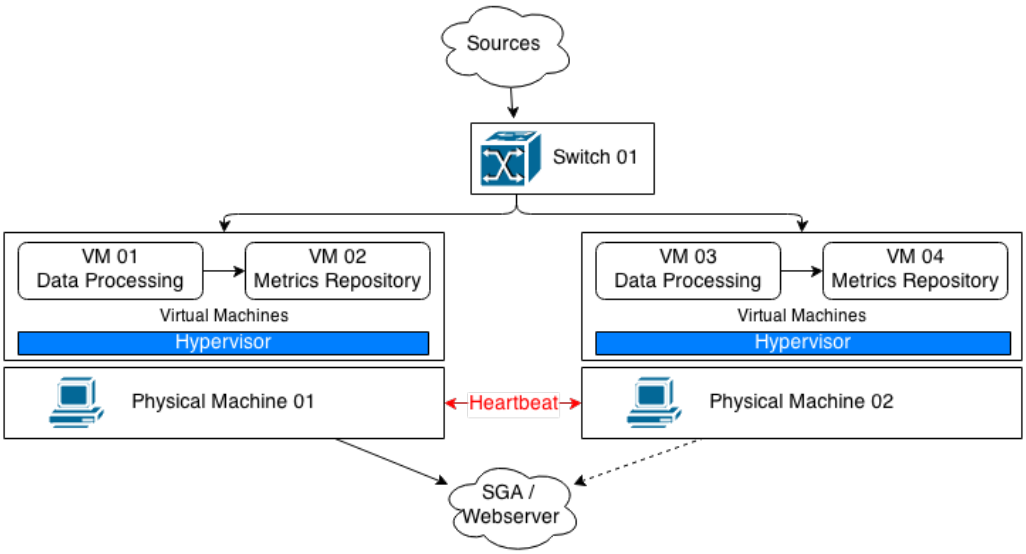

Figure 4.3: Architecture 1.

Option 2

A similar architecture to Option 1 would also consist in two physical machines, but if it is not possible to configure the network equipments to do port mirroring, we can configure dynamic IP addressing as described in the previous section (Figure 4.4).

The stream of events remains configured to one IP address which is the primary in the NETS system, in the event of the primary crashing, the IP would be taken over by an equivalent but correct machine. Both machines would have the same state taking advantage of RabbitMQ's mirrored queues.

This approach has the advantage that only one of the machines is filtering the incoming stream of events and sending the alarms to the SGA, but both of them perform event correlation to keep an updated Esper state.

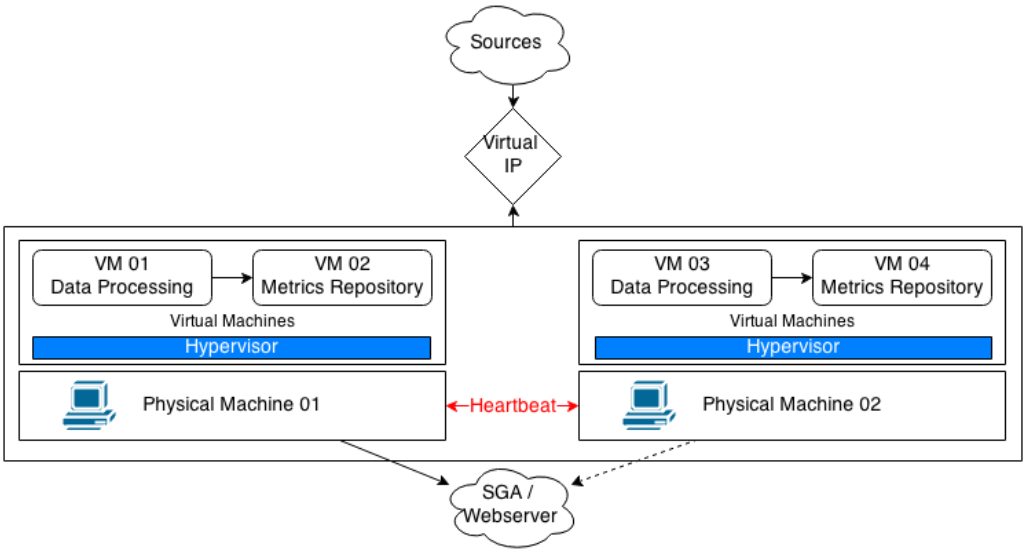

Figure 4.4: Architecture 2.

Option 3

An alternative would consist in adding an extra machine responsible for distributing the stream of events to the NETS' processing cluster (Figure 4.5).

This machine would only distribute the incoming stream of events to every node of the processing cluster, becoming the single point of failure. Its unavailability would compromise the whole operation but given the less complex software it is expected to have higher reliability than the rest, i.e., a bigger mean time before failure (MTBF).

Option 4

A more complex solution involves load distributors and two clusters, combining the elements from Option 2 and 3.

Basically, we could extend Option 3, but instead of only having a cluster of data processing machines, the distribution machine would also be replicated and given a cluster using Option 2's virtual IP for availability and redundancy.

This would effectively remove the single point of failure, but would require extra hardware, a more complex algorithm and applications to manage every replica.

An illustration of this architecture can be seen in Figure 4.6.

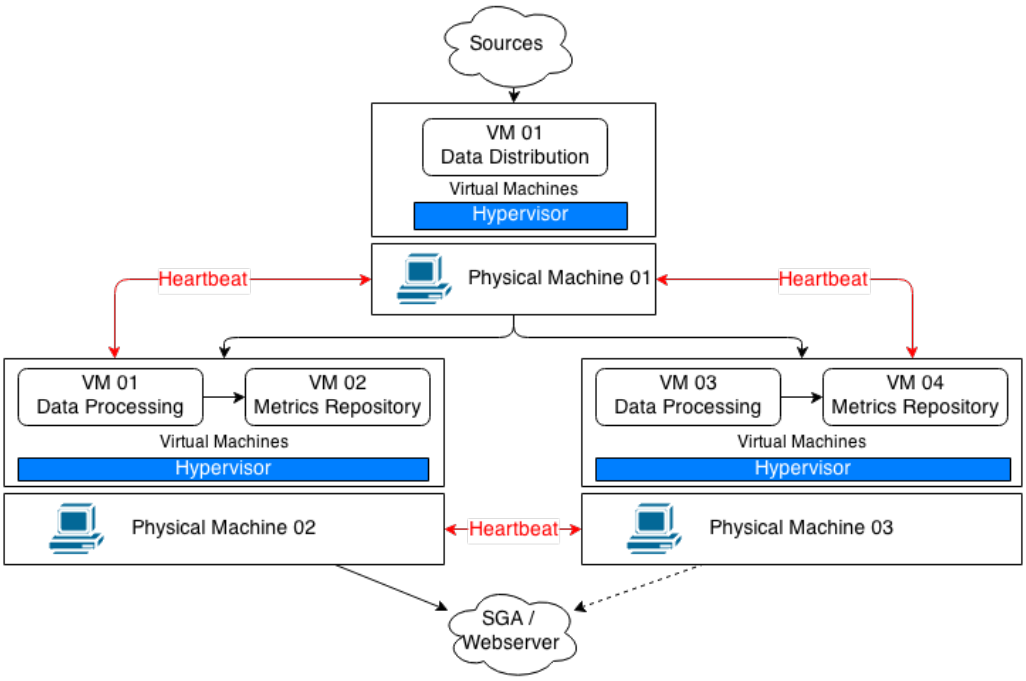

Figure 4.5: Architecture 3.

4.3 Proposals for Fault Tolerance Enhancements at System Level

4.3.1 Port Mirroring

Port mirroring is an option that requires the reconfiguration of the network switch where the NETS system servers are physically located. This process could be attempted but requires contacting other departments and teams to approve and perform the required changes.

The fact that it involves external actions can delay and complicate the adoption of this method, making the other options more suitable for the task at hand.

4.3.2 Load Distributor

This second option requires acquiring an extra machine to be assigned as a load distributor. One disadvantage is the additional management required to install, configure and maintain the new hardware.

There is also a bureaucratic process attached to grant the extra equipment that can be lengthy and difficult, making this an even less probable choice than port mirroring.

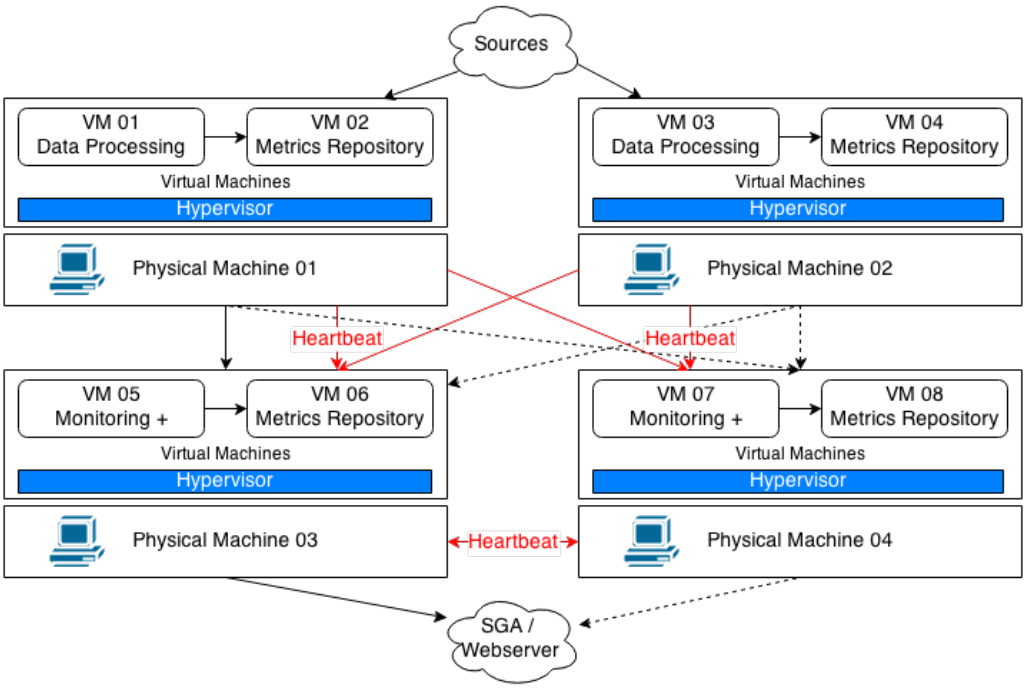

Figure 4.6: Architecture 4.

4.3.3 Virtual IP

A Virtual IP (VIP) can be used for connection redundancy by providing alternative fail-over options on one machine; a VIP address may still be available if a computer or NIC (Network Interface Controller) fails, because an alternative computer or NIC replies to connections replacing the failed one.

There is software available to perform this task, e.g., Keepalived[27], which is free and open source. Otherwise a simple implementation can be developed to customize the configuration and restrict the functionality to the NETS system's requirements.

4.4 Summary

In this chapter we reviewed possible solutions to implement in accordance with Chapter 3's analysis. In Table 4.1 is shown a comparison of the several architectures that were defined as potential candidates implementable in the NETS system.

	Min #	Input Conciliation	Output Conciliation	SPOF(Reliability)
Option 1	2	Port Mirroring	Single	Switch(High)
Option 2	2	Virtual IP	Single	-
Option 3	3	Load Distributor	Single	Server(Medium)
Option 4	4	Virtual IP + Load Distributor	Single/Vote	-

Table 4.1: Architectures Chart.

After this analysis we see that the system initially consisted of only one physical machine, not having any mechanisms for fault tolerance. Details of the resulting implementations or the lack of them will be presented and explained in the following chapter.

Chapter 5

Implementations

This chapter enumerates the changes and implementations made in the NETS system. A detailed description of their development and justification is made to review the advantages and possible disadvantages.

5.1 Enhancements of individual components

In this section we will describe the solutions regarding individual components, which were considered relevant or important and therefore implemented in the NETS system.

These modifications were deemed important to improve the reliability, recoverability and fault tolerance of each individual node belonging to the NETS system. Therefore, improving the overall system availability and resilience.

5.1.1 Visualization Improvement - Scripts

The initial interventions on the NETS system were comprised by the development of several Ruby scripts to extract relevant metrics on the operation of some components in order to improve the system's visualization. This information is intended to be displayed in dashboards and be used by other applications to issue alarms and reports should anomalies happen. They were:

- RabbitMQ: Extraction of statistical data of queue sizes, free disk space (in accordance with proposed solution design 4.1.6), number of connections, etc.;
- netstat: We measure metrics related to the TCP and UDP connections through netstat, including the number of connections, their state, etc.;
- Cron: Organization and preparation of a general script that is targeted by a cron to automatically run scripts associated with the NETS system;

- JVM: The correlation engine uses the JVM, so it was proposed to monitor some data related to the garbage collection (GC), sizes and proportions of the various generations of the GC, etc.

These metrics are aggregated by a daemon, statsd, which sends them to be graphed by Graphite.

5.1.2 Traffic Generator (Stresser)

We identified several tools to test each of the components. From simulating the sources / inputs with rsyslog[1], must[2], sysloggen[3] and loggen[4] to generate load on MB RabbitMQ with Tsung[5] and JMeter-Rabbit-AMQP[6], and finally CEP Esper with it is Performance Kit.

After reviewing the available applications, the decision of developing custom software was made for better control, option's flexibility and available parameters. The code was written using Ruby, an object-oriented, general-purpose programming language, to take advantage of the available expertise as a learning opportunity.

Stresser was created to generate load, taking into account several parameters for generating the messages to be sent. Its primary use is to add load to the sources / inputs processes and test the limits and behavior of the various components. It is used along the Fault Injection software detailed in the next topic, simulating the presence of faults and / or high load, helping to taking steps for the improvement and correction of the system's stability, consistency and robustness.

V1 - Basic generation and send mechanisms.

V2 - Added continuous send mode.

V3 - Added memcache support.

V4 - Added configuration files for application and messages.

V5 - Added options to send specific message types.

V6 - Added new type for alarm generation (DSL FLAP).

[1] http://www.rsyslog.com/
[2] http://sourceforge.net/projects/mustsyslog/
[3] https://subversion.assembla.com/svn/logzilla/scripts/contrib/sysloggen/
[4] http://www.balabit.com/
[5] http://tsung.erlang-projects.org/
[6] https://github.com/jlavallee/JMeter-Rabbit-AMQP

5.1.3 Fault Injection and Behavior Monitoring

Through research in the subject of fault injection and its techniques, Netflix's Chaos Monkey[11] crossed our path. It is a service which runs in Amazon Web Services (AWS) that seeks out Auto Scaling Groups and terminates random instances (virtual machines) in the group. This methodology is used because failures happen and they inevitably happen when least desired or expected, testing the capability to tolerate instance failures and its consequences as well as their recovery ability.

This tool, Chaos Monkey, was not used in NETS but was the inspiration to implement an application with similar features, and another to register the changes made to the relevant log files when a failure is detected in the system. Both these applications are detailed below.

NETS Monitor and Monit (Updated rules)

NETS Monitor and a set of scripts were developed to work together with monit in order to log events related to the NETS system's applications. When an application stops responding or crashes, monit resets the application and registers the recent logs that may have evidence about the problems' cause.

It is used along with monit to carry out its work, as monit performs the monitoring and detection of application changes, it calls specific scripts depending on the case, i.e., if it is a crash, a CPU overload, etc.

V1 - Rules added for processes, process termination and CPU usage limits.

V2 - Added pid files to detect reboot of processes.

V3 - Added monitoring of RabbitMQ's process and pid in accordance with proposed solution design 4.1.4.

V4 - Added monitoring of syslog-ng process and pid in accordance with proposed solution design 4.1.2.

NETS Faults

NETS Faults is an application to inject random faults in a set of defined processes. It requires adding the file holding the pid information relative to the application we want to add to our set which will be target of random faults.

V1 - List of processes, option to finish random or specific processes.

V2 - Added and option to continuously end processes with a defined percentage.

5.1.4 Configuring RabbitMQ

RabbitMQ's configuration changed in favor of turning the exchanges' durability to *true*. This modification involved editing the configuration files of etl-syslog and etl-radius, in which the definitions for the graphite and alarms exchanged were added.

An additional configuration change could involve enabling queue persistence. This would guarantee that queued messages waiting to be processed are kept even in the event of a crash.

Another possible configuration involves enabling message persistence and confirmation. With these options enabled, messages would be safe to a greater extent in case of problematic situations. This would have a slight impact on performance and was deemed to not being required because clustering solves this particular situation.

5.2 Enhancements at system level

5.2.1 Load Distributor

The Load distributor required the implementation of an application that distributes the receiving stream to configured and IP addresses. It can be used to support Option 2 of the architectures described in 4.2.5.

This application requires configuring the desired listening ports, and the IP or IPs to which a copy of the arriving stream is sent to. Its operation requires connecting to the target machines and, in the absence of a successful connection, tries to reconnect periodically.

The development of this software was halted before it was finalized because the chosen architecture will use Option 3, which uses Virtual IP instead of a load distributor.

5.2.2 Monitor for Alarm Transmission State

Standalone application that regulates the transmission of alarms to the SGA. It should be installed in each NETS machine.

Checks the state of the cluster through RabbitMQ and enables or disables the alarm transmission by the node taking into account if it should be sending or not, i.e., the one running correctly with biggest uptime in the cluster. In a correct state, one machine has the transmission active, the remainders have it disabled.

5.2.3 Introduce Fault Tolerance

Two of the purposed solutions were tested: using a load distributor and Virtual IP attribution.

The system is now comprised by two individual physical machines working side by side and is capable of tolerating a fault in any given machine.

The adopted implementation was Virtual IP attribution in accordance to the proposed solution design 4.3.3. Custom software was developed to manage the IP address management and assignment, given that additional configuration was needed in order for the cluster of RabbitMQs to keep working as intended.

This last implementation is still in test phase, requiring minor tweaking for it to work according to the system's needs and remove possible faults in the code.

5.3 Summary

Both individual and system wide improvements have been described in this chapter. The reasons for their implementations and any advantages are also mentioned.

In the next chapter we will present two evaluations. One that compares the system state before and after the individual enhancements, and another that makes a system wide comparison summarizing the main differences and achievements.

Chapter 6

Evaluation

This chapter presents a comparison of the system architecture and state between major improvements/milestones, showing the differences and benefits accomplished by the enhancements explained in the previous chapter.

6.1 Comparison between the initial state and after individual improvements

Given the modifications detailed in the previous chapter (Chapter 5), the behavior and response of the NETS system has improved comparing to the non Fault-tolerant version.

To validate the new implementations, the system was resubmitted to the fault tests performed while the system was analyzed (as seen in Chapter 3) in order to record its behavior given the new improvements. The following conclusions were taken:

- Corrections made in the message broker, RabbitMQ, improved its reliability and recovery. Consequentially, the complex event processor, Esper, also increased its reliability because of its dependency with the message broker;
- The addition of new rules and missing processes to the monit configuration enables a better visualization of their status and speeds up the system's recovery in the presence of faults;
- Every process in the NETS ecosystem is now capable of automatically restoring itself to a working condition in case of their failure.

6.2 Comparison between initial condition and improvements at the system level

Initially, the NETS system was composed by a single physical machine with two virtual machines responsible for the data processing and metrics respectively.

After the implementation of the new fault tolerant architecture composed by two physical machines, tests were performed to record the system's reaction under the new conditions. These tests consisted in simulating a node's failure in order to force the NETS system to react to that failure, i.e., the primary node fails and the passive node should be promoted to primary to ensure the system's continued work.

The following conclusions were taken:

- Added redundancy, going from a single physical machine to a working cluster composed by two different physical machines, but with the freedom to add more nodes to the system if needed.
- The stream of events is sent to a single machine, effectively preventing common-mode failures related to the input, but the filtered data is shared by all members of the cluster using the message broker's features. This maintains an updated state in every node enabling a quick takeover and recovery if the primary node fails.
- Fault tolerance enabled, in the event of a crash or unavailability of the primary node in the cluster, another node will take over briefly and continue to perform the work normally, minimizing the system's unavailability.
- Automatic recovery, should a passive node fail, the primary is unaffected by this situation and the failed node triggers recovery mechanisms in order to return to a working condition.

6.3 Summary

The implementations introduced in the system described in the previous chapter resulted in the improvement of NETS. A review of the expected changes in capabilities taking into account the context of fault tolerance and dependability is systematized in Table 6.1.

	Initial State	Individual Improvements	System Level Improvements
# Nodes	1	1	2
Self Monitoring Scale	Low	Medium	High
SPOF	X	X	-
Automatic Recovery	-	X	X
Fault Tolerance	-	-	X

Table 6.1: Milestone Comparison.

In the next chapter we will present the final remarks and conclusions taken during the project's time frame.

Chapter 7

Conclusion

The beginning of the project started with an in-depth analysis, what components constitute the system, their function and dependencies. The system's architecture was pictured in a diagram along with the use cases of each major operation that takes place during the process.

After exploring and having a good notion of the system, development began with small bug fixes and improvements. Several tools were implemented to test and monitor the system state while faults were injected, allowing further research and elaboration of potential strategies to solve the identified situations.

At this point, and after the final single node improvements were made, the node's reliability was greatly improved. Not only it became capable of restoring its status to a correct and working state after the failure of any of its components, it also became more resilient and fault free.

The final part of the work was related to the introduction of redundancy and thus making the system fault tolerant. An additional machine was added to compose a cluster that works together to generate the intended alarms. The adopted strategy was to use a Virtual IP that is assigned to the primary node of the cluster. This node is the only one that parses the stream of events and sends alarms to the SGA, and the resulting messages are mirrored through all the remaining nodes using the message broker's embedded capabilities.

In its current state, the system is capable of sustaining one node failure. In the event of a failure, the remaining node assumes the primary's role and starts parsing the stream of events becoming the sole alarms generator. When and if the failed node recovers, it will rejoin the cluster and serve as the backup if the newly appointed primary fails.

Several tools and methodologies used and developed in the NETS improvement can be exploited in other projects, especially regarding the configuration and management of RabbitMQ and Esper along with all the covered fault tolerance topics.

7.1 Achievements

At the end of the project's time frame, the work done in the NETS system resulted in the following achievements:

- The source code of each component was thoroughly analyzed to correct software bugs and code refactoring in order to improve its performance. This analysis was also useful to learn what each component does, how it does it and its dependencies;
- New versions of several applications used in the NETS system were available, delivering better performance, stability and more features. Their update required code refactoring and some adaptations but the benefits were considered to be worth the extra work;
- **Failure Detection and Recovery** - The NETS system is constituted by a variety of components, in which the failure of a single one can potentially nullify the entire operation. Given that continuous operation is a requirement, each component began to be monitored and, if any should fail, they are restarted in order to return to a correct working condition;
- **Failure Detection** - Several software applications were developed to test, monitor and report the NETS system state. Testing the system required developing an application that simulates the incoming sources, in order to easily perform case studies and learn the system's performance capabilities. The monitoring and reporting software were implemented to improve the monitoring of the NETS system and to log its state and changes when a failure is detected in order to enable a better analysis and understanding of the fault's origin and details.
- **Fault Tolerance** - Redundancy was introduced making the system fault tolerant. This required an extra physical machine and a careful analysis of how the incoming stream of events, the data processing and produced outputs are treated with the new architecture.

7.2 Future Work

At this point the NETS system can still be improved, some of the available options open to exploration are suggested taking into account their relevance and natural improvement progression:

- **From Development to Production**
  The latest implementations related to the addition of redundancy are still in the development environment, the next logical step is to commit the new modifications

and integrate them into the production environment. This will require further testing to assure everything is working as intended.

- **Port Mirroring**
  Given that the final architecture is composed by two physical machines, it would be interesting to experiment with port mirroring. Having the stream of events being delivered to the two machines independently would only require monitoring each other to determine the master without any other dependencies.

- **Back-office**
  The NETS back-office needs further development to expose and have a good monitoring view of the system and its alarms state as well as integrating some of the new metrics retrieved by recent implementations.

- **ETL-alarms_worker Refactoring**
  The improvement of ETL alarms_worker could be considered given its limited performance.

Appendix A

UML Activity and Sequence Diagrams

In this section we will be presenting the UML diagrams that cover the NETS system operation.

In Figure A.1 the operational phases are shown along with the main procedures that occur within the NETS system. The remainder figures detail the identified procedures in each phase.

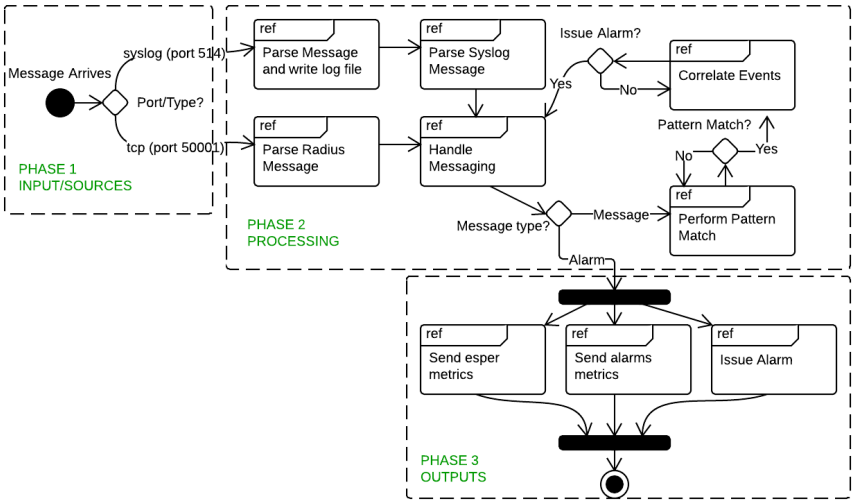

Figure A.1: NETS Phases.

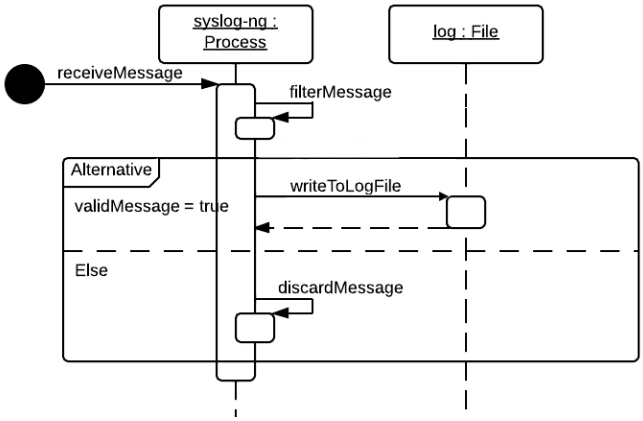

Figure A.2: Parse Message and Write Log.

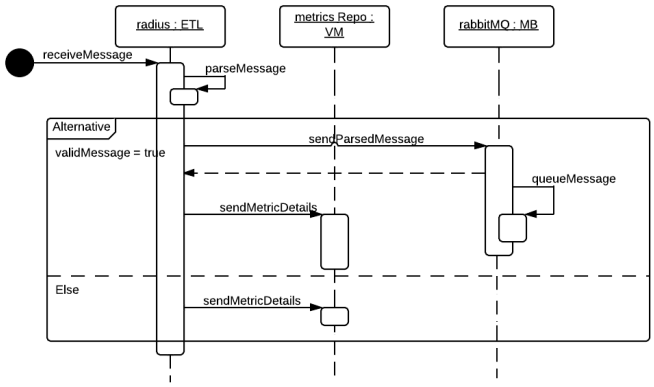

Figure A.3: Parse Radius Message.

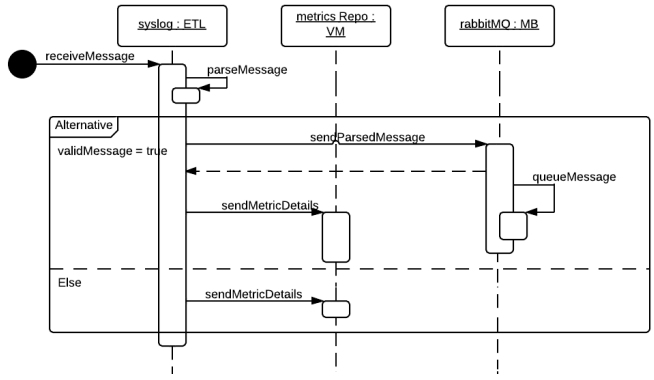

Figure A.4: Parse Syslog Message.

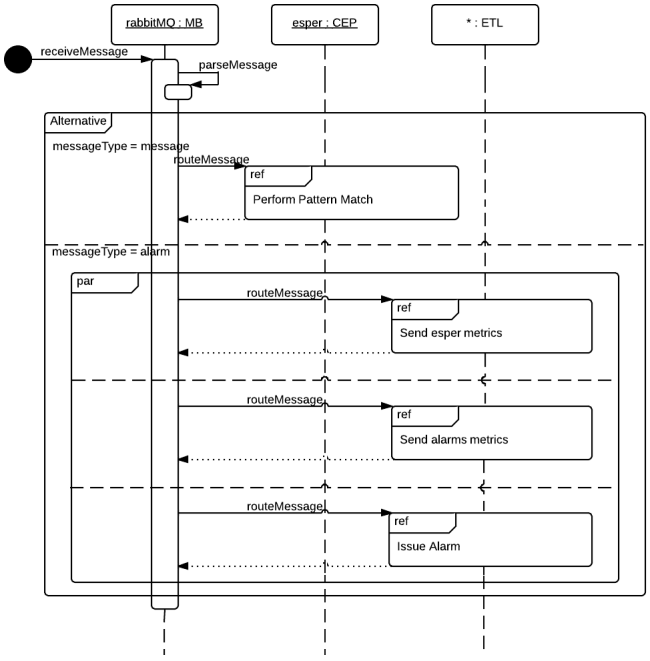

Figure A.5: Handle Messages.

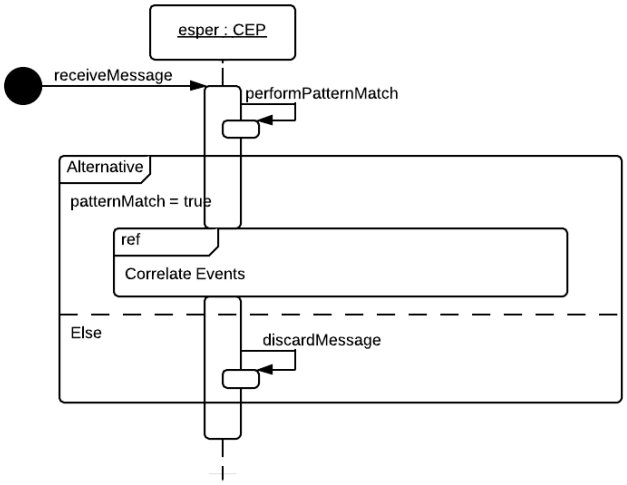

Figure A.6: Perform Pattern Matching.

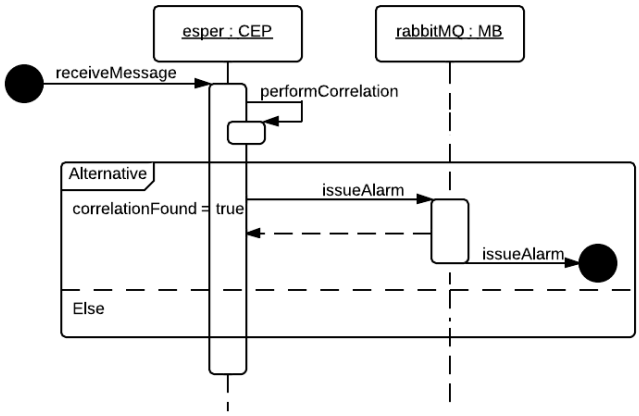

Figure A.7: Correlate Events.

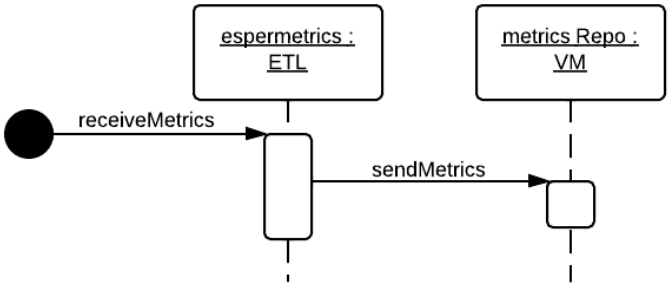

Figure A.8: Send Esper Metrics.

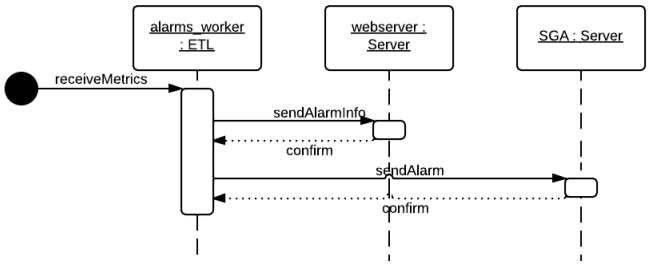

Figure A.9: Issue Alarm.

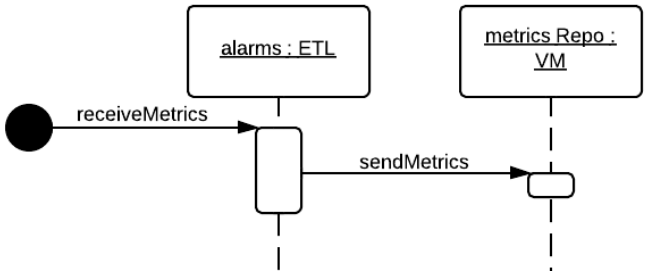

Figure A.10: Send Alarm Metrics.

Appendix B

Fault Testing Details

The performance tests were ran in the NETS system using Stresser as the load generator. The tests targeted each component, registering the amount of messages sent, the CPU and memory used by the relevant processes, and the consequences to the system.

INFO -r 27 sends about 50-55 messages/second

B.1 DSL

Test began 20 February 2014 at 10:18 using the following command:
ruby st.rb –ip 10.101.68.76 -t 0 -r 27 –type dsl

In Table B.1 there is the detailed progress of the performed test. Each line represents a record which can be seen as a picture of the system's current state. The first column shows the load and time stamp of the record, followed by the CPU and memory of the relevant processes, finalizing with a column showing if there are lost messages.

Load [Start Time]	CPU [Memory] RabbitMQ	CPU [Memory] etl-syslog	CPU [Memory] Esper	Request Loss Rate
50 [1018]	0.9% [88780 kB]	4.3% [451996 kB]	2.0% [2256400 kB]	0%
100 [1023]	1.1% [91004 kB]	7.1% [483028 kB]	2.5% [2256516 kB]	0%
150 [1028]	1.5% [89744 kB]	10.6% [486948 kB]	3.0% [2256608 kB]	0%
200 [1033]	2.0% [88400 kB]	15.4% [490200 kB]	3.7% [2257096 kB]	0%
250 [1038]	2.3% [88972 kB]	19.0% [488928 kB]	4.1% [2257100 kB]	0%
300 [1043]	2.6% [88336 kB]	22.2% [489504 kB]	4.6% [2257140 kB]	0%
350 [1058]	2.2% [88524 kB]	25.2% [501240 kB]	4.4% [2269552 kB]	33%
400 [1106]	1.6% [89268 kB]	26.6% [508004 kB]	4.1% [2269580 kB]	50%
450 [1115]	1.0% [88944 kB]	28.9% [509460 kB]	3.8% [2269576 kB]	66%
500 [1125]	0.7% [88528 kB]	30.2% [514536 kB]	3.6% [2269580 kB]	100%

Table B.1: DSL Fault Testing

B.2 STB

Test began 21 January 2014 at 08:50 using the following command:
ruby st.rb –ip 10.101.68.76 -t 0 -r 27 –type stb

Load [Start Time]	CPU [Memory] RabbitMQ	CPU [Memory] etl-syslog	CPU [Memory] Esper	Request Loss Rate
50 [0850]	0.9% [93848 kB]	3.3% [502992 kB]	1.9% [1624796 kB]	0%
100 [0855]	1.2% [91880 kB]	6.8% [511984 kB]	2.5% [1624848 kB]	0%
150 [0900]	1.7% [92488 kB]	11.3% [517892 kB]	3.1% [1624936 kB]	0%
200 [0908]	2.2% [93340 kB]	15.2% [515064 kB]	3.8% [1624940 kB]	0%
250 [0913]	2.5% [93440 kB]	18.7% [517836 kB]	4.2% [1627004 kB]	0%
300 [0918]	2.9% [92492 kB]	22.5% [523444 kB]	4.8% [1627288 kB]	0%
350 [0923]	3.0% [92296 kB]	23.9% [523592 kB]	5.0% [1641520 kB]	33%
400 [0933]	1.6% [93654 kB]	26.6% [534564 kB]	4.1% [1641564 kB]	50%
450 [0943]	1.2% [94792 kB]	29.3% [547248 kB]	3.8% [1641584 kB]	66%
500 [0953]	0.8% [93004 kB]	31.5% [546512 kB]	3.7% [1641584 kB]	100%

Table B.2: STB Fault Testing

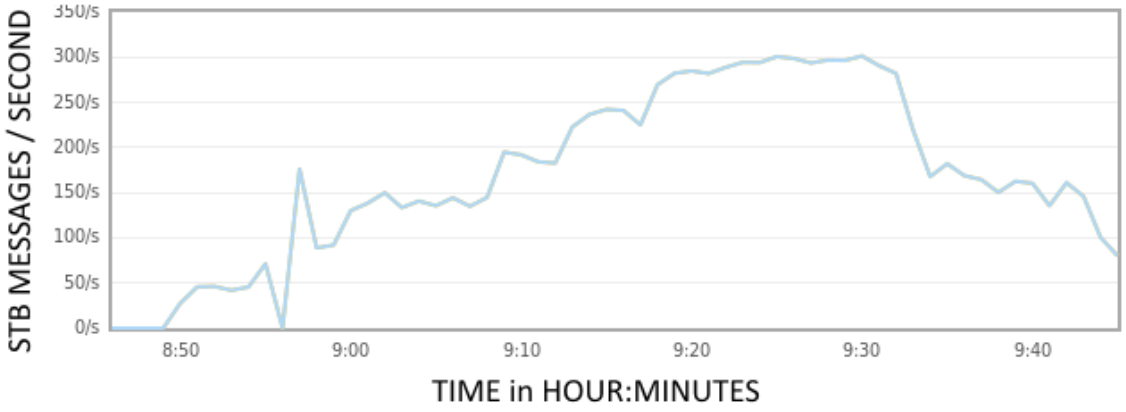

Figure B.1: STB Total Processed Messages' Rate During Test.

B.3 RADIUS

Test began 21 January 2014 at 10:44 using the following command:
ruby st.rb –ip 10.101.68.76 -t 0 -r 27 –type rad

B.4 DSL-FLAP

Test began 10 February 2014 at 15:23 using the following command:
ruby st.rb –ip 10.101.68.76 -t 0 -r 15 –type dsl-flap

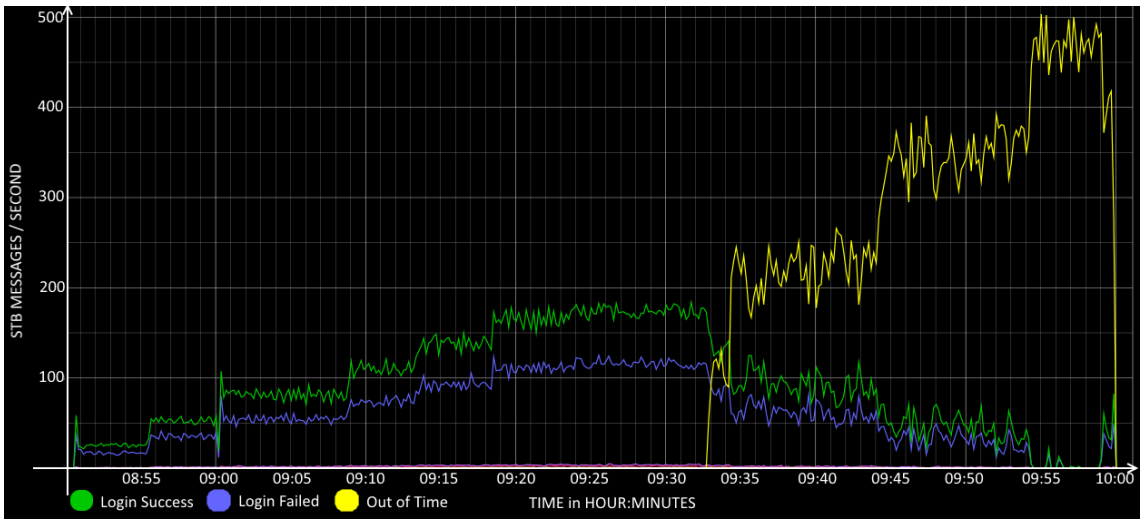

Figure B.2: STB Individual Processed Messages' Rate During Test.

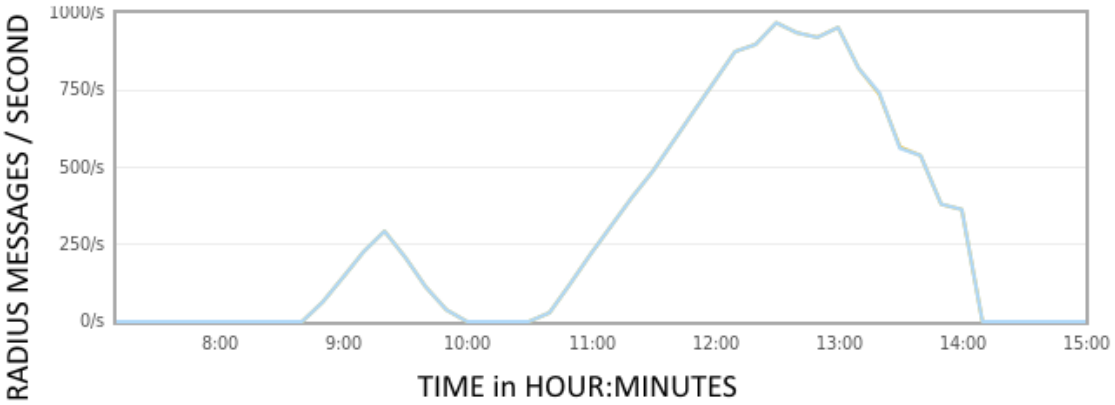

Figure B.3: RADIUS Total Processed Messages' Rate During Test.

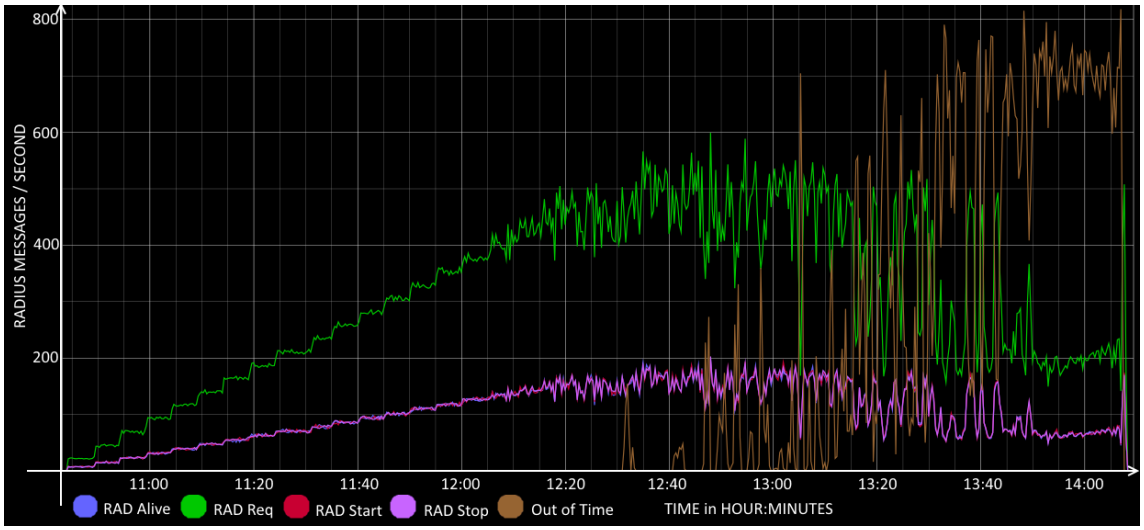

Figure B.4: RADIUS Individual Processed Messages' Rate During Test.

Load [Start Time]	CPU [Memory] RabbitMQ	CPU [Memory] etl-radius	CPU [Memory] Esper	Request Loss Rate
50 [1044]	0.7% [90840 kB]	5.3% [398404 kB]	2.1% [2249800 kB]	0%
100 [1049]	1.1% [93076 kB]	3.4% [438260 kB]	2.4% [2249840 kB]	0%
150 [1054]	1.6% [92748 kB]	5.6% [439004 kB]	3.1% [2260088 kB]	0%
200 [1059]	1.8% [93252 kB]	7.9% [439168 kB]	3.6% [2262600 kB]	0%
250 [1104]	2.1% [92708 kB]	10.1% [439356 kB]	4.2% [2264700 kB]	0%
300 [1109]	2.5% [93036 kB]	12.2% [441528 kB]	4.8% [2268808 kB]	0%
350 [1114]	2.7% [92592 kB]	14.2% [441676 kB]	5.2% [2272912 kB]	0%
400 [1119]	3.0% [94192 kB]	16.5% [443588 kB]	5.8% [2282704 kB]	0%
450 [1124]	3.4% [94016 kB]	18.6% [444044 kB]	6.3% [2299728 kB]	0%
550 [1135]	4.0% [93716 kB]	22.6% [446384 kB]	7.3% [2312020 kB]	0%
600 [1140]	4.2% [94132 kB]	24.5% [448548 kB]	7.9% [2328568 kB]	0%
650 [1145]	4.7% [94912 kB]	26.5% [450728 kB]	8.3% [2340876 kB]	0%
700 [1150]	5.0% [94632 kB]	28.8% [450888 kB]	8.9% [2347020 kB]	0%
750 [1155]	5.5% [95184 kB]	32.0% [453212 kB]	9.6% [2351116 kB]	0%
800 [1200]	6.1% [94720 kB]	36.5% [455208 kB]	10.6% [2359308 kB]	0%
850 [1205]	7.0% [94208 kB]	42.9% [455380 kB]	12.7% [2381836 kB]	0%
900 [1210]	7.2% [94496 kB]	42.6% [462564 kB]	12.6% [2387980 kB]	0%
950 [1215]	7.7% [96620 kB]	43.0% [464732 kB]	14.9% [2394132 kB]	0%
1000 [1220]	7.8% [94920 kB]	46.2% [464925 kB]	14.2% [2400292 kB]	0%
1050 [1225]	7.1% [94604 kB]	43.9% [467116 kB]	11.9% [2410556 kB]	0%
1100 [1230]	8.4% [93436 kB]	47.4% [471756 kB]	14.4% [2433084 kB]	>0%
1150 [1240]	7.4% [93852 kB]	45.5% [473988 kB]	14.6% [2482636 kB]	>0%
1200 [1250]	7.4% [94668 kB]	45.0% [478248 kB]	13.4% [2503124 kB]	>0%
1250 [1300]	8.3% [98396 kB]	47.6% [480448 kB]	15.5% [2517076 kB]	>0%
1300 [1310]	8.0% [95216 kB]	46.7% [482596 kB]	14.2% [2529392 kB]	>0%
1350 [1320]	7.7% [95020 kB]	46.4% [484804 kB]	13.5% [2543740 kB]	>0%
1400 [1330]	3.9% [94060 kB]	41.5% [489004 kB]	8.2% [2553980 kB]	>0%
1450 [1340]	6.5% [94068 kB]	46.3% [491428 kB]	12.3% [2562176 kB]	>0%
1500 [1350]	4.1% [94628 kB]	41.9% [494632 kB]	8.5% [2564244 kB]	>0%
1550 [1355]	3.9% [94140 kB]	42.2% [496772 kB]	8.3% [2566272 kB]	>0%
1600 [1400]	4.3% [93440 kB]	42.6% [498928 kB]	8.9% [2570368 kB]	>0%

Table B.3: RADIUS Fault Testing

Time	Load	Message Queueing
1523	2-2.5	No queueing
1528	4.5-5.5	No queueing
1533	6-8	No queueing
1538	10-12	No queueing
1543	14-16	Message queueing starts

Table B.4: DSL-FLAP Testing Times

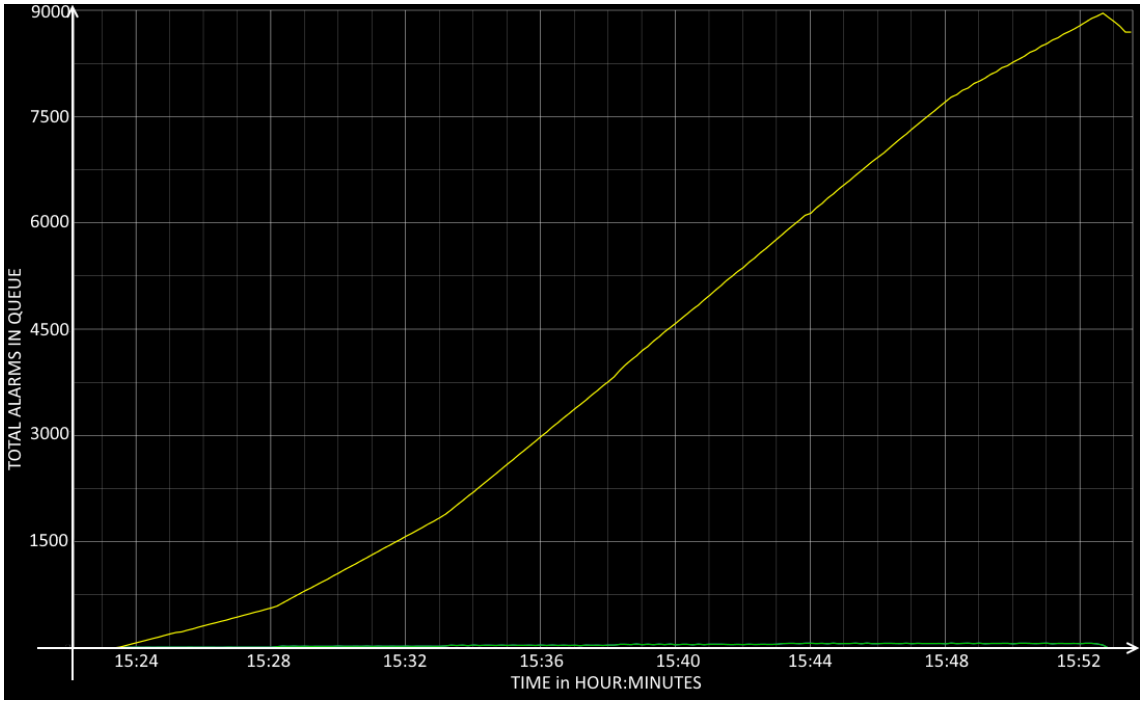

Figure B.5: DSL-FLAP Total Alarms in Queue.

Bibliography

[1] rsyslog project website. `http://www.rsyslog.com/`. [Online; accessed 06-October-2013].

[2] Advanced Message Queuing Protocol. `http://www.amqp.org/`. [Online; accessed 12-October-2013].

[3] Fred R Arndt and CGM Oliver. Hardware monitoring of real-time computer system performance. *Computer*, 5(4):25–29, 1972.

[4] Algirdas Avizienis, Jean claude Laprie, Brian R, Carl L, and Senior Member. Basic concepts and taxonomy of dependable and secure computing. *IEEE Transactions on Dependable and Secure Computing*, 1:11–33, 2004.

[5] BFT-SMaRt code repository. `https://code.google.com/p/bft-smart/`. [Online; accessed 12-October-2013].

[6] AJ Bourne, GT Edwards, DM Hunns, DR Poulter, and IA Watson. *Defences against common-mode failures in redundancy systems: a guide for management designers and operators*. UKAEA Safety and Reliability Directorate, 1981.

[7] John Bowles and Sharma Upadhyayula. Network management and performance monitoring. In *System Theory, 1991. Proceedings., Twenty-Third Southeastern Symposium on*, pages 529–534. IEEE, 1991.

[8] Alan Burns and Andrew J Wellings. *Real Time Systems and Their Programming Languages: Ada 95, Real-time Java and Real-time POSIX*. Pearson Education, 2001.

[9] António Casimiro, Paulo Verissimo, Diego Kreutz, Filipe Araujo, Raul Barbosa, Samuel Neves, Bruno Sousa, Marilia Curado, Carlos Silva, Rajeev Gandhi, et al. Trone: Trustworthy and resilient operations in a network environment. In *Dependable Systems and Networks Workshops (DSN-W), 2012 IEEE/IFIP 42nd International Conference on*, pages 1–6. IEEE, 2012.

[10] Miguel Castro and Barbara Liskov. Practical byzantine fault tolerance and proactive recovery. *ACM Transactions on Computer Systems (TOCS)*, 20(4):398–461, 2002.

[11] NetFlix's Chaos Monkey. http://techblog.netflix.com/2012/07/chaos-monkey-released-into-wild.html. [Online; accessed 10-April-2014].

[12] Graphite project website. http://graphite.wikidot.com/. [Online; accessed 06-October-2013].

[13] must project website. http://sourceforge.net/projects/mustsyslog/. [Online; accessed 06-October-2013].

[14] John D Day and Hubert Zimmermann. The osi reference model. *Proceedings of the IEEE*, 71(12):1334–1340, 1983.

[15] Elasticsearch homepage. http://www.elasticsearch.org/. [Online; accessed 12-October-2013].

[16] Erlang Programming Language. http://www.erlang.org. [Online; accessed 09-October-2013].

[17] Esper project website. http://esper.codehaus.org/. [Online; accessed 06-October-2013].

[18] Didier Essame, Jean Arlat, and David Powell. Padre: A protocol for asymmetric duplex redundancy. In *Dependable Computing for Critical Applications 7, 1999*, pages 229–248. IEEE, 1999.

[19] Statsd project website. http://github.com/etsy/statsd/. [Online; accessed 06-October-2013].

[20] RabbitMQ project website. http://www.rabbitmq.com. [Online; accessed 06-October-2013].

[21] David C Hoaglin, Frederick Mosteller, and John Wilder Tukey. *Understanding robust and exploratory data analysis*, volume 3. Wiley New York, 1983.

[22] Java Programming Language. http://www.java.com. [Online; accessed 09-October-2013].

[23] JRA Plugin project website. http://github.com/jlavallee/JMeter-Rabbit-AMQP. [Online; accessed 06-October-2013].

[24] logstash code repository. http://github.com/logstash/logstash. [Online; accessed 06-October-2013].

[25] logstash project website and documentation. http://logstash.net/. [Online; accessed 06-October-2013].

[26] JavaScript Object Notation. http://www.json.org/. [Online; accessed 11-October-2013].

[27] Keepalived homepage. http://www.keepalived.org/. [Online; accessed 20-April-2014].

[28] Hermann Kopetz. Real-time systems: design principles for distributed embedded applications. 1997.

[29] Israel Koren and C Mani Krishna. Fault-tolerant systems. 2007.

[30] Jaynarayan H Lala and Richard E Harper. Fault tolerance in embedded real-time systems: importance and treatment of common mode failures. In *Hardware and Software Architectures for Fault Tolerance*, pages 263–282. Springer, 1994.

[31] Leslie Lamport, Robert Shostak, and Marshall Pease. The byzantine generals problem. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 4(3):382–401, 1982.

[32] Jean-Claude Laprie. Dependable computing and fault-tolerance. *Digest of Papers FTCS-15*, pages 2–11, 1985.

[33] Jean-Claude Laprie. Dependability: Basic concepts and associated terminology. *PDCS First Year Report*, 1990.

[34] Jean-Claude Laprie. Dependable computing: Concepts, limits, challenges. In *FTCS-25, the 25th IEEE International Symposium on Fault-Tolerant Computing-Special Issue*, pages 42–54, 1995.

[35] Librato metrics storage and visualization. https://metrics.librato.com. [Online; accessed 12-October-2013].

[36] Chung Laung Liu and James W Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *Journal of the ACM (JACM)*, 20(1):46–61, 1973.

[37] Monit project website. http://mmonit.com/monit/. [Online; accessed 06-October-2013].

[38] David E Morgan, Walter Banks, Dale P Goodspeed, and Richard Kolanko. A computer network monitoring system. *Software Engineering, IEEE Transactions on*, (3):299–311, 1975.

[39] David Ernest Morgan and David James Taylor. Special feature a survey of methods of achieving reliable software. *Computer*, 10(2):44–53, 1977.

[40] David Ernest Morgan, David James Taylor, and G Custeau. A survey of methods for improving computer network reliability and availability. *Computer*, 10(11):42–50, 1977.

[41] SJ Mullender. Distributed systems. 1993.

[42] Nagios project website. `http://www.nagios.org/`. [Online; accessed 07-October-2013].

[43] Tsung project website. `http://tsung.erlang-projects.org/`. [Online; accessed 06-October-2013].

[44] Patrick O'Connor and Andre Kleyner. *Practical reliability engineering*. John Wiley & Sons, 2011.

[45] David M Ogle, Karsten Schwan, and Richard Snodgrass. Application-dependent dynamic monitoring of distributed and parallel systems. *Parallel and Distributed Systems, IEEE Transactions on*, 4(7):762–778, 1993.

[46] Anand K Ojha. Techniques in least-intrusive computer system performance monitoring. In *SoutheastCon 2001. Proceedings. IEEE*, pages 150–154. IEEE, 2001.

[47] Laura L Pullum. *Software fault tolerance techniques and implementation*. Artech House, 2001.

[48] Brian Randell. System structure for software fault tolerance. *Software Engineering, IEEE Transactions on*, (2):220–232, 1975.

[49] Brian Randell. *System structure for software fault tolerance*. Springer, 1978.

[50] Ruby Programming Language. `http://www.ruby-lang.org/`. [Online; accessed 11-October-2013].

[51] Sensu Monitoring Framework. `http://sensuapp.org/`. [Online; accessed 11-October-2013].

[52] Simple Network Management Protocol. `http://www.snmplink.org`. [Online; accessed 11-October-2013].

[53] Janusz Sosnowski and Marek Poleszak. On-line monitoring of computer systems. In *Electronic Design, Test and Applications, 2006. DELTA 2006. Third IEEE International Workshop on*, pages 5–pp. IEEE, 2006.

[54] SQL Programming Language. `http://www.sql.org/`. [Online; accessed 09-October-2013].

[55] Andrew S Tanenbaum and Maarten Van Steen. *Distributed systems*, volume 2. Prentice Hall, 2002.

[56] TRONE project website. `http://trone.di.fc.ul.pt`. [Online; accessed 06-October-2013].

[57] Paulo Verissimo and Luis Rodrigues. *Distributed systems for systems architects*, volume 1. Springer, 2001.

[58] Dinesh C Verma. *Principles of computer systems and network management*. Springer, 2009.

[59] Zenoss project website. `http://www.zenoss.com`. [Online; accessed 07-October-2013].