INSTITUTO SUPERIOR TÉCNICO
Universidade Técnica de Lisboa

A Fault-Tolerant Network Intrusion Detection System

Tiago Sampaio de Faria Picado

Dissertação para obtenção do Grau de Mestre em

Engenharia Informática e de Computadores

Jurí

Presidente:	Prof. Cjgldadlkh Njgjkjala Bklalsjdla Ylkhjçasdj
Orientador:	Prof. Carlos Nuno da Cruz Ribeiro
	Prof. Paolo Romano
Vogais:	Prof. António Casimiro

Maio de 2012

To Sofia, Inês and Miguel

ACKNOWLEDGMENTS

First and foremost I would like to thank my supervisors, Prof. Carlos Ribeiro and Prof. Paolo Romano 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. Thank you also for your patience in dealing with a student with a somewhat busy life outside academia.

A few people in my former school, Instituto Superior de Agronomia, have helped me complete this task by giving me their vote of confidence. I must specially thank Prof. António StAubyn and Prof. Luísa Brito, who were kind enough to write recommendation letters at the time of application for the Master's course.

Many colleagues in my new school have made going through the course much easier than it could have been, by integrating the "outsider guy" as one of their own. Not meaning to leave any of them out, I must remember in particular Pedro Filipe Silva and Vasco Fernandes.

I also need to thank all my co-workers at CIISA, Ardil (current and former), and elsewhere, as completing this course many times meant extra work for them, and still I felt the encouragement to proceed!

To Filipe Ribeiro, Rodrigo Almeida and Tiago Ferreira, my friends of all times. Even if in later years too many times so far apart, your encouragement is still felt!

If someone ever kept pushing me to reach this goal, it was my father and his companion. I thank you both for continuing to push.

A very special Thank You to the (grand)mothers! Without your enormous support taking care of the kids, I most likely wouldn't be writing this page. A thank you also to my sister for her encouragement.

Finally, I am grateful to my family, Sofia and the kids, for giving me the strength and inspiration to pursue this to the end!

RESUMO

Os sistemas de deteção de intrusão de rede podem desempenhar um papel determinante na segurança de uma organização; é por isso importante que eles possam analizar o máximo possível dos dados disponíveis. É no entanto sabido que, em certas circunstâncias, em particular quando sujeitos a carga elevada, eles podem perder algum do tráfego de rede. Nesta tese foi investigada a possibilidade de minimizar falhas por omissão por parte destes sistemas, utilizando multiplas réplicas de um sistema de deteção de intrusão. A ideia subjacente é o facto de a probabilidade de um conjunto de réplicas perder um dado pacote ser menor do que a de um sistema composto por uma única instância. Foi desenvolvida uma camada de sincronização, colocada entre a camada de captura de pacotes e a camada de deteção de intrusão (Snort). Esta camada deteta desalinhamentos nos conjuntos de pacotes recebidos por cada réplica, utiliza o algoritmo diff para comparar os pacotes recebidos e recupera os pacotes perdidos por cada uma. Apesar de o protocolo de sincronização ser normalmente necessário precisamente quando as condições são mais exigentes, e a capacidade de recuperação poder por isso ser limitada, os resultados sugerem que pelo menos em alguns cenários o mecanismo de sincronização poderia revelar-se útil. Os casos em que as réplicas conseguiram recuperar pacotes e manter o alinhamento, em situações em que um sistema único perderia pacotes, foram o de tráfego de rede com picos, e o da breve execução de um processo com necessidades elevadas de processamento, numa das réplicas.

Palavras-chave:

Sistema de deteção de intrusão de rede; falhas por omissão; replicação; protocolo de sinchronização; diff

ABSTRACT

Network intrusion detection systems can play a determinant role in providing adequate security to an organization, and so it is important that those systems are given the chance to analyze as much of the available data as possible. However, it is known that under certain conditions, in particular when under heavy load, they may lose some network packets. In this thesis, the possibility of minimizing intrusion detection systems omission failures by using multiple replicas of an intrusion detection system was researched. The underlying idea is the fact that the probability of loss of a given packet by the set of replicas should be lower than the probability of loss of that packet by a system composed by a single intrusion detection instance. A synchronization layer, sitting between the packet capture layer and the intrusion detection layer (Snort) was developed. This layer detects misalignments in the received packet sets, uses the diff algorithm to compare packets received by each replica, and recovers packets missing from each one. While the synchronization protocol is usually required precisely when conditions are more demanding and the ability to recover packets may for that reason be limited, results suggest that at least on some scenarios the synchronization mechanism could prove to be useful. Cases where the replicas were able to recover packets and maintain alignment, where a single intrusion detection system would lose packets, included the case of short network traffic bursts and the case of a short-lived processor-consuming process launched on one of the replicas.

Keywords

Network intrusion detection system; omission failures; replication; synchronization protocol; diff

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF ACRONYMS

CRC	–	Cyclic Redundancy Check
CIDF	–	Common Intrusion Detection Framework
CPU	–	Central Processing Unit
DMTCP	–	Distributed MultiThreaded CheckPointing
DNS	–	Domain Name System
FIFO	–	First In First Out
HIDS	–	Host Intrusion Detection System
ICMP	–	Internet Control Message Protocol
IDS	–	Intrusion Detection System
IP	–	Internet Protocol
IPC	–	Inter-Process Communication
IPS	–	Intrusion Prevention System
MAC	–	Media Access Control
MD5	–	Message Digest 5
MTCP	–	MultiThreaded CheckPointing
NIDS	–	Network Intrusion Detection System
PPP	–	Point-to-Point Protocol
SSO	–	Site Security Officer
TCP	–	Transmission Control Protocol
UDP	–	User Datagram Protocol
WEP	–	Wired Equivalent Privacy
WIDS	–	Wireless Intrusion Detection System
WPA	–	WiFi Protected Access

1 INTRODUCTION

From country-sized corporations to home and mobile users, today's world is increasingly reliant on information systems and on the communications networks which connect them. The Internet in particular, and its related set of technologies have become nothing short of ubiquitous, and increasing convergence between Information Technology and Telecommunications worlds is taking this reality even further.

Hand in hand with this usage growth, came an increase in the number of attacks to those systems and networks. Besides the more traditional attacks where intruders, either originating from inside or outside of an organization, will direct their attacks at particular networks, with specific goals regarding those networks, the current scenario dictates that every connected system is a target, as armies of so-called zombie machines are needed for the propagation of junk e-mail messages, phishing schemes or to be participants of massive coordinated attacks against selected targets.

Protection from network attacks is therefore also increasingly important. In the context of network security, besides implementing secure protocols (cryptographically sound, DoS resistant), further defenses are provided by two kinds of artifacts: those that provide access control mechanisms on the flow of network traffic - firewalls - and those that are capable of detecting and alerting on unwanted network activities - intrusion detection systems [1].

Ideally, access control policies would allow all legitimate traffic to pass through the policy enforcing artifacts, while blocking all unwanted traffic. Since this is not possible, however, network intrusion detection systems can prove to be an essential part of network security, either by alerting on intrusion attempts in real time, allowing for countermeasures to be taken before attempts succeed or at least prevent further attacks from succeeding, or by logging the necessary data for later analysis.

1.1 Motivation

Since the data gathered by intrusion detection systems in general, and by network intrusion detection systems in particular, can be viewed as essential for providing adequate security to an organization, it is important that those systems are given the chance to analyze as much of the available data as possible.

Network intrusion detection systems are known to drop packets is some situations, namely in those situations where they are unable to process with sufficient speed the traffic stream that is presented to them for analysis. This may happen if the system has insufficient resources, namely processing resources, to handle all the traffic. It should be highlighted that network intrusion detection can be very resource-demanding, given the high-speed links intrusion detection systems may be required to monitor.

Considering the deployment of an adequately dimensioned system, packet loss could still occur on rare occasions, either due to unusual spikes in the normal network traffic pattern, or to a momentarily high demand of system resources, for instance due to the execution of a task by some process running in the system.

Even if rare, packets that were lost could prove to be important in detecting a relevant intrusion or intrusion attempt. To minimize this possibility, two or more intrusion detection systems could be deployed for analyzing the same network traffic. However, since a number of network traffic analysis depend on the availability of a sequence of related packets, and not of just one packet, it could happen that in a given situation both of them would lose packets, and while parts of the sequence would be missed by one intrusion detection system, other parts would be missed by the other, even though together they could have the complete sequence.

1.2 Thesis Goals

In this thesis, the possibility of minimizing network intrusion detection systems omission failures by using multiple synchronized replicas of an intrusion detection system was researched. The underlying idea is the fact that the probability of loss of a given packet by the set of replicas should be lower than the probability of loss of that packet by a system composed by a single intrusion detection system instance.

For this purpose, the widely disseminated *open-source* Snort [2] intrusion detection system was used. Snort is the most wide-spread intrusion detection system and this together with the availability of its source code, turned it into an ideal candidate for the research.

Furthermore, the current stable version of Snort can for the most part be considered single-threaded. This means that it is not able to take advantage of the fact that most computer system today, even single-user desktop systems, are equipped with multi-core processors, which results in the availability of spare processing capacity that could be used for other tasks.

The main goal of the thesis was to develop a synchronization protocol able to detect misaligned replicas in regard to the stream of packets each of them recovered from the network for analysis, to determine which specific packets were different across replicas, and to recover required packets, so that all the intrusion detection system replicas could analyze the same set of packets. For this, a synchronization layer sitting between the network traffic capture library and the intrusion detection system was developed.

This means that the synchronization layer is intended to replace the usual packet source of the intrusion detection system, and becomes responsible for passing it the packets for analysis. The thesis also aimed at evaluating the possibility of optimistically providing the IDS with unverified packets, at the cost of needing to perform a rollback operation if it was found that the packets processed by the IDS were incomplete. This could help minimize the latency introduced by synchronization, which could be critical if the intrusion detection system was running as an intrusion prevention system.

Finally, the thesis also aimed at providing a solution for saving and transferring the state between replica instances. This would be a requirement for allowing a new replica to join a running replica set, and for the state of the intrusion detection component of that replica to become aligned with the state of the already running ones. For this, solutions in the field of process checkpointing were researched.

1.3 Document Organization

The remaining of the document is organized in the following manner:

In chapter 2, background information on a number of topics is reviewed, which includes the field of intrusion detection, a discussion about some of Snort's details, the presentation of important concepts of fault tolerance, and the problem of saving and restoring the state of a process.

In chapter three, design concepts of the proposed solution are presented. The methods used for replica alignment verification and for packet recovery and the passing of packets to the intrusion detection system are explained. At the end of the chapter, the proposed synchronization algorithm is presented.

In chapter four, some details of the implementation are discussed. An overview of the system architecture is given, and the major system components are described. Finally, some comments about the modifications made to the source code of Snort and of the MTCP checkpointing library are made.

In chapter five, an evaluation of the system effectiveness and efficiency is carried out. The results of tests performed to the system are presented and discussed.

In chapter six, the thesis ends with some concluding remarks and suggestions for future work.

2 BACKGROUND

Before turning attentions to the problem being addressed in the present thesis, background knowledge on the areas which are needed for discussing the proposed solution needs to be revisited.

First, a summary of the state of the art regarding the field of intrusion detection is presented. Commonly used terminology in this field is introduced, and so are some of the main goals and challenges involved in developing systems adequate for intrusion detection. This is followed by a brief discussion of some of the major aspects used when classifying intrusion detection systems, which helps in understanding their functioning, as well as some of the key design choices that need to be made. The general architecture of an intrusion detection system is presented, and the purpose of each of its components is explained. This section is ended by presenting some of the practical difficulties which need to be tackled when deploying this type of technology in production environments.

Next, the *open source* intrusion detection system Snort, which was used during the research, is introduced. A general presentation of the software and its importance in the intrusion detection landscape is given, followed by a look into Snort's inner workings, highlighting its layered and modular architecture, a feature which favors its usage in research, and then describing the purpose of each of its individual layers.

The chapter continues with a discussion on fault-tolerant systems and of some related fundamental concepts, such as failure and communications models. Attention is then turned to software replication strategies. Replication is a key aspect for ensuring fault-tolerance and high-availability, and the classic strategies for accomplishing this task are presented. The problem at hand is then viewed in light of the presented concepts, and it is shown how this problem presents some unique challenges, compared with the assumptions made in the classical scenarios just described.

Finally, the problem of saving and transferring process state is introduced, and the area of process check-pointing is reviewed. Process checkpointing technology provides a possible solution for two issues which arise in the research: transferring state from a running replica to a newly launched replica that is to become part of a cluster, and rolling back the process state of a replica which is optimistically analyzing traffic before synchronization is ensured.

2.1 Intrusion Detection Systems

When guarding computer systems or networks against attacks, the conventional approach is to deploy a number of protective mechanisms in order to secure them. However, this prevention-based approach has a number of limitations, including [3]:

- It is difficult, or perhaps impossible to build useful systems which are absolutely secure, as the possible existence of design or administrative flaws cannot be excluded in a system with a large number of components;
- It is impractical to assume that a vast existing and possibly insecure infrastructure will be replaced in favor of a new, secure one, given the tremendous investment already made in the current infrastructure;
- The prevention-based security philosophy constrains user's activities, making them less productive, while a more open mode of operation is regarded by many as highly useful for promoting user productivity;
- Crypto-based systems cannot defend against lost or stolen keys, or cracked passwords;
- Finally, secure systems can still be vulnerable to insiders misusing their privileges since they cannot fully guard against the insider threat.

These limitations justify the use of other anti-intruder approaches[1]. Since even the best prevention-focused mechanisms will ultimately fail, detection of intrusions or intrusion attempts can provide a second line of defense. Like burglar alarms, detection-focused mechanisms do not prevent intrusions, but provide this defense by [4]:

- Enabling early detection of intrusion activities. Measures can then be taken to eject the intruder from the system before damage is done, or to limit the amount of damage if the intrusion has already partially succeeded;
- Dissuading the intruder's intentions, therefore preventing intrusions;
- Enabling the collection of information about intrusion techniques that can be used to strengthen the prevention facilities.

Tools which perform the detection of intrusion activities followed by recovery measures are known as Intrusion Detection Systems. However, present intrusion detection systems are increasingly focusing on real-time attack detection and blocking, diminishing the difference between intrusion prevention and detection. Systems with such focus are known as Intrusion Prevention Systems [5].

[1] Halme and Bauer [41] provide an anti-intrusion taxonomy comprised of six techniques: *prevention* (precluding or severely handicapping the likelihood of an intrusion's success), *preemption* (striking against a threat before an intrusion attempt occurs, decreasing the likelihood of its later occurrence), *deterrence* (avoiding the initiation or continuation of an attack by increasing the necessary efforts or risks associated with launching the attack, or by devaluing its perceived gains), *deflection* (letting an intruder believe he has succeeded, while attracting him to where harm is minimized), *detection* (discriminating intrusion attempts and intrusion preparation from normal activity and alerting authorities) and taking *countermeasures* (actively and autonomously counter intrusions as they are being attempted).

2.1.1 Terminology

Seminal work on the design of a security monitoring mechanism, for managing the detection of system misuse through the analysis of activity, done by J. P. Anderson in 1980 [6], launched the field of intrusion detection and introduced a number of important definitions. Since they have not since lost validity, they are reproduced here, as found in the original report:

Threat: The potential possibility of a deliberate unauthorized attempt to[2]:

a) access information
b) manipulate information
c) render a system unreliable or unusable

Risk: Accidental and unpredictable exposure of information, or violation of operations integrity due to malfunction of hardware or incomplete or incorrect software design.

Vulnerability: A known or suspected flaw in the hardware or software design or operation of a system that exposes the system to penetration of its information to accidental disclosure.

Attack: A specific formulation or execution of a plan to carry out a threat.

Penetration: A successful attack; the ability to obtain unauthorized (undetected) access to files and programs or the control state of a computer system.

Other terms have since Anderson's report entered the intrusion detection lexicon [7]:

Exploit: The process of using some vulnerability to violate a security policy. A tool or a defined method that could be used to violate a security policy is often referred to as an exploit script.

Incident: A collection of data representing one or more related attacks. Attacks may be related by a number of properties, including attacker, type of attack, objectives, sites, or timing.

Intruder: An intruder or attacker is a person who carries out an attack. A potential intruder may be referred to as an adversary.

Intrusion: A common synonym for attack, but more precisely, a successful attack.

False negative: An event that the intrusion detection system fails to identify as an intrusion or an intrusion attempt, when one has in fact occurred.

False positive: An event that is incorrectly identified by the IDS as being an intrusion or an intrusion attempt, when none has occurred.

2.1.2 Goals and Requirements

In order to fulfill their intended purpose, intrusion detection systems must not only be able to detect attacks, but to do so with adequate accuracy and timeliness, so that the impact of

[2] In other words, the CIA of computer security: *Confidentiality*, *Integrity* and *Availability*

intrusions and intrusion attempts may be minimized. This imposes a number of goals and requirements that such systems must meet. The major ones that such systems must meet include [5], [8]:

1. They should be able to detect a wide variety of intrusions, originating from both inside and outside the site. Also, they should be able to detect both previously known and unknown attacks. This later requirement suggests the need to have some learning or adapting mechanism.
2. They should be accurate in their detection. The goal is to develop a system which has a broad attack detection coverage, minimizing the number of false negatives, while keeping the rate of false alarms as low as possible, because in a live network the number of false positives may exceed the number of correctly detected attacks, which undermines the confidence in the system.
3. Detection should be accomplished in a timely fashion. This does not mean it is required that it takes place in real-time, as real-time intrusion detection may raise issues of responsiveness, while it is often sufficient to discover an intrusion within a short period of time. But this may become a requirement if the goal of a given system is to prevent attacks, rather than simply detecting them.
4. Intrusion detection systems may be required to be able to handle large amounts of data without affecting performance and without dropping data. In particular, they may need to process audit data and make decisions at a rate which is equal to or greater than the rate of arrival of new audit patterns. This becomes a critical factor it today's high speed networks.
5. Presentation of the analysis should be done in a simple and easy to understand format, ideally as simple as a green light if no attacks were detected and a red light otherwise. In reality intrusions are rarely clear-cut, and intrusion detection systems must therefore present more complex data for the site security officer to determine if and what action is to be taken. It is desirable that the presented data can help in a quick analysis of the intrusion, for example by identifying the type of attack.
6. They should be scalable an easily customizable, in order to be able to adapt to the specific requirements of the environments in which they are deployed, which may vary considerably.
7. Finally, intrusion detection systems should themselves be resistant to attacks, because if they fail this requirement, they can themselves become targets of the attackers, with the purpose of thwarting their ability to adequately provide intrusion detection functionality.

2.1.3 Classification

Different intrusion detection systems take different approaches to the way they try to identify intrusion attempts. This leads to fairly different systems, but of course common aspects also exist. Over the years, a number of authors have tried to systematize various aspects related to intrusion attempts and intrusion detection, by providing taxonomies or ontologies, usually centered on one of the elements involved (intruders, their techniques, intrusion detection systems, targets or even audit data are examples of this). For taxonomies focused on intrusion detection systems, their detection methods and operational characteristics see, for example, [9], [10] or more recently, [11]. Here a brief summary of some of the main characteristics used to classify IDSs is presented.

2.1.3.1 Detection Principle

Arguably, the most important aspect for defining an intrusion detection system is its detection principle. Mainly, two complementary trends for the detection principle exist: to try to detect a deviation from the normal usage pattern of a system, which is referred to as *anomaly*, *behavior-based* or *statistical* intrusion detection, and to use accumulated knowledge about attacks and try to look for evidence of the presence of those attacks, which is referred to as *misuse*, *knowledge-based* or *signature-based* intrusion detection.

1. In **anomaly, behavior-based or statistical IDSs**, a model of what is considered normal behavior has to be built, and this model is usually updated on a regular basis. The main advantage of this approach is that, in theory, it is possible to use it to detect previously unknown types of attacks. However, in practice, some drawbacks make anomaly systems harder to deal with than with misuse systems: "normal behavior" is not easy to define and so operators can easily be swamped with false-positives, which discredit the IDS. An intruder may even use this deliberately, to lower the operator’s reactivity to the system, or since the model is updated regularly, he can train the IDS so that it considers the intrusion attempts as "normal". Finally, a behavior-based IDS may identify that something is wrong, but has no knowledge allowing it to identify what it is.
2. In **misuse, knowledge-based or signature-based IDSs**, on the other hand, it should be possible to identify precisely the type of attack the system is under, while keeping the false-positive rate at a much lower level (but, do read "the base-rate fallacy", later in this section). On the other hand, this kind of system is unable to detect previously unknown attacks, which may lead to a high false-negative rate.

Even though the dichotomy between anomaly and misuse systems has existed since the early days of intrusion detection, Axelsson [12] shows how in light of classical detection and estimation theory, this dichotomy vanishes, or is at least weakened.

2.1.3.2 Audit Data Source

A second characteristic of great importance to define an intrusion detection system is the source of audit data. When the analyzed audit data refers to events on a single host, the intrusion detection system is said to be a *Host based IDS* or *HIDS*. When the analyzed audit data is gathered from traffic collected on a network, the intrusion detection system is said to be a *Network based IDS* or *NIDS*.

Some authors, such as [7], consider four categories instead of two: *Application IDS* or *AIDS*, which focus their analysis on the audit data of a specific application, the already mentioned *HIDS* and *NIDS*, and *Multi-network/Infrastructure IDS*, which usually take the form of an incident response team.

Discussing the two main types, we find that both have relative strengths and weaknesses:

1. **Host based IDSs** where the first systems to be designed, at a time when the target environment was a mainframe computer. HIDSs have the advantage of being close to the monitored system, which gives them access to security information not available to NIDSs, such as system status, file system alterations and log files. Some of this information might provide confirmation of the success of an attack, whereas NIDSs are only able to report that an intrusion attempt was performed. Finally, HIDSs are able to overcome the problem of ciphered network traffic (see below). On the other hand, HIDSs require host by host configuration, can degrade the performance of the monitored host, can't usually detect probing activities before the actual attacks, and don't have an overview of all events that might be happening in their surroundings.
2. **Network based IDSs** can have this overview; plus, a single NIDS can monitor events related to a large number of hosts and does not degrade the production systems performance, as NIDSs are usually deployed on dedicated hardware. It also has access to information contained in raw network packets which can be of high value for the analysis.

2.1.3.3 Response on Detection

Regarding their response to detection events, intrusion detection systems can be classified as being either passive or active. Passive response systems just log and alert on a detection event, so that an SSO (Site Security Officer) may take the appropriate measures, while active systems autonomously take countermeasures to thwart ongoing attacks. The actions that such systems can take when suspicious activities are identified are varied. Some are discussed bellow, when intrusion prevention systems are presented.

2.1.3.4 Detection Time

Regarding the time when detection takes place, two main types of operation exist: on-line or real time detection, when intrusion detection systems try to detect attacks and intrusions as they occur, and off-line, batch or *a posteriori* detection, when audit data is stored for analysis at some latter time. As discussed earlier, this later mode of operation is not suitable for active response systems.

Some authors consider that there are in reality two categories for real time detection: actual real time detection and virtual real time detection [13]. Systems in the first category are capable of detecting attacks as they occur, and can prevent them from succeeding. Systems in the second category are capable of detecting attacks shortly after they succeed, and therefore give time for reactions which prevent the success of further attempts, either by eliminating the vulnerabilities which allowed the success of the attack, or by isolating the vulnerable system.

2.1.3.5 Architecture

Most intrusion detection systems use a centralized architecture and monitor events on a single host or network. But this architecture is not well suited for what are today common distributed network attacks, and neither for dealing with complex network architectures. For these, a distributed architecture, with data collection and/or analysis in multiple locations, is best suited. More recently, new DIDS architectures, such as GRID IDSs (e.g. [14], [15]) or Peer-to-Peer IDSs (e.g. [16]) have generated interest in the research community.

2.1.3.6 Special Types

To end this discussion on intrusion detection system classification, two types of intrusion detection systems which are sometimes viewed as belonging to a class of their own, need to be mentioned:

Intrusion Prevention Systems

Intrusion detection systems with a strong emphasis on the response component are referred to as Intrusion Prevention Systems (IPSs). As was previously mentioned, the response nature can vary considerably, ranging from defensive actions, such as shutting down affected systems or restoring them to an uncompromised state, reconfiguring firewalls, resetting network connections or blocking them, if working as Inline IDSs, to counter-attack actions, which are highly controversial, and may also be legally dubious.

Honeypots

Honeypots and honeynets are decoy systems which are deployed to achieve a number of goals: divert the attacker's attention from critical systems, buy time for administrators to respond to intrusion activities, or study the behavior and techniques of attackers. Their intrusion detection abilities steam from the fact that honeypots/honeynets are systems not advertised among legitimate users, and therefore any interest in them is to be suspected.

2.1.4 Generalized IDS Architecture

As becomes apparent from the previous section, the inventors of intrusion detection systems are presented with a wide number of implementation choices, which lead to fairly different intrusion detection systems. Even so, it is possible to form some sort of notion of a “typical” intrusion detection system, and its constituent parts. Any generalized architectural model of an intrusion detection system would contain at least the following elements [17][3]:

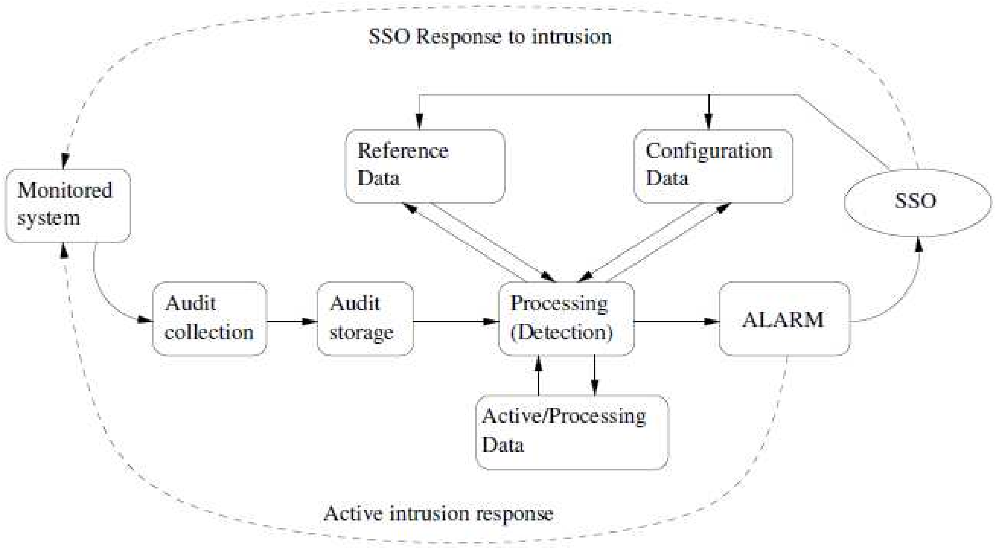

Figure 1 – Organization of a generalized intrusion detection system (from [17]).

Audit collection: Collection of low level audit data for intrusion detection analysis. Sources of data can be keyboard input, log files, system loads, file system changes, network traffic, etc. In the CIDF framework, this element would correspond to the Event generators (*E-Boxes*). The collection of low level events can be performed by one or many "E-Boxes". In the latter case, the fusion of events from different sources has to be performed.

Audit storage: Low level audit events are typically stored somewhere (in files or in a database), either awaiting for processing (in addition to the cases where data is collected from multiple sources and has to be fused before analysis, since processing can take significant time, sometimes the tasks of collecting and processing data are separated, even if only one source of data is used) or indefinitely for later reference. The amount of stored data can become quite large, due to the amount of raw events. In the CIDF framework, this element would correspond to the Event databases (*D-Boxes*).

[3] For an earlier IDS architecture proposal, see The Common Intrusion Detection Framework Architecture [17]. The Common Intrusion Detection Framework (CIDF, `http://gost.isi.edu/cidf/`) ideas later originated the now concluded IETF Intrusion Detection Working Group. The goals of this group where to define data formats and exchange procedures for sharing information of interest to intrusion detection and response systems, and to management systems which might need to interact with them.

Processing: The processing block is the heart of the intrusion detection system. Here is where audit data is fused (if needed) and analyzed by one or more algorithms, in search for intrusion evidence. In the CIDF framework, this element would correspond to the Event analyzers (*A-Boxes*).

Configuration data: Data which determines the operating mode of an instance of the intrusion detection system: which data sources to use, which and how detection components are used, how to respond to detected intrusion events. This is the main means of control of the intrusion detection system by the SSO (Site Security Officer), and it represents data which can be considered sensitive, as it could provide an intruder with information on which forms of attack would likely go undetected.

Reference data: The reference data storage stores information about known intrusion signatures for misuse systems or profiles of normal behavior for anomaly systems. Most often, intrusion signatures are updated by the SSO from an outside source[4], and are usually tied to a specific intrusion detection system. In the case of anomaly systems, profiles of normal behavior are usually updated by the processing element, at regular intervals.

Active/processing data: Intermediate results used by the processing element, for example, partial assemblies of a data flow. These can become quite large. In Snort, many configuration options can be found, which are related to limits on the amount of data a particular detection component will hold before starting to discard data, based on some criteria.

Alarm: The part of the system which handles all system output. This can be an automated response to suspicious activity, in which case it corresponds to the function of the CIDF framework Response Units (*R-Boxes*), a warning to an SSO or the storage of the high level event in permanent storage (once again, usually in files or in a database).

2.1.5 IDS hurdles

Intrusion detection technology is not an easy technology to deploy. A number of practical difficulties prevent it from being more generally widespread and from being truly effective when it is indeed deployed. Here, some of those difficulties are reviewed.

The base-rate fallacy: Both false-positive and false-negative alerts represent serious problems in intrusion detection systems. Recall that a false-positive means that an alert occurred but there was no attack, while a false-negative means that the IDS failed to generate an alert for an attack. Before [18], false-negatives where considered more serious, as they mean missing attacks while providing a false sense of security. However, the cited paper shows that due to a phenomenon known as "the base-rate fallacy", and for a

[4] In the case of Snort, the "Snort rules" would be provided either by *Sourcefire* or by the Snort community. More information on this topic can be found in the section where Snort is discussed.

reasonable set of assumptions, false-positives can be a more limiting factor in IDS performance. This is because an adequately low false-positive rate per event is very hard to achieve, due to the sheer number of events. The problem with high false-positives, or false alarms, as has been previously mentioned, is that they unnecessarily consume the operator’s time and discredit the IDS, which then starts getting ignored.

Performance: Intrusion detection, and especially network intrusion detection, can be a highly resource demanding activity. Even without considering local area traffic where currently even a relatively modest network switch can forward a number of packets in the order of tens of millions per seconds, these days, just monitoring the typical corporate Internet link often means inspecting hundreds of Mbit/s of data, or more. In high-speed links, besides the problem of being able to hold enough processing data in memory, NIDSs often start dropping packets, as they are unable to process all of them. Aside from resorting to schemes involving scaling the number of used IDSs, some research in using hardware solutions for packet inspection has also been done (e.g. [19], [20]).

Complexity/Scalability: As corporate networks grow larger, internal monitoring becomes more difficult, because of the need to monitor multiple high-speed segments. Common options include using various intrusion detection systems scattered throughout the network, raising the amount of investment and administrative work, or to send data collected through sensors to a single IDS, which tends to overload the switch/hub port to which the IDS is connected. For networks which use a star topology, a possible solution is given in [21].

Ciphered traffic: Ciphered traffic has become the norm, especially for communication with the more mission-critical services. This adds to security by providing confidentiality and integrity to communications, but it also means that the more common class of knowledge-based network intrusion detection systems can no longer detect many attacks. Even though some attacks against ciphered services can be detected (see, for example, Snort's SSH preprocessor), many others, such as pattern-matching of HTTP requests, cannot.

Cost: Intrusion detection, when taken seriously, can become fairly costly, and the business case to justify that cost is not always an easy to make. In Network Intrusion Detection [22], an example is given of how quickly an investment on intrusion detection can grow from requiring "two fast computers" at 2.500 dollars each, to requiring hundreds of thousands of dollars on licenses, salaries and specialized equipment. Costs will be proportional to the size of the organization, but not all will be willing to support them.

2.2 Snort

Snort is a highly popular intrusion prevention and intrusion detection system (IPS/IDS) which, according to its home website[5], is the most widely deployed IDS/IPS technology worldwide, and has become the *de facto* standard for this kind of technology.

Due to its popularity, feature set and source code availability, the Snort IDS/IPS became a natural choice for this project. In this section, some aspects specific to this intrusion detection system are presented. This section is for the most part based on [23], [24] and on analysis done on the software source code.

Snort began its life in 1998 as a project by Martin Roesch for a packet sniffer. In 1999 it was already being described as a "Lightweight Intrusion Detection for Networks" [2], and today it is considered a full-edged IDS/IPS, with a feature set comparable to those of a commercial-grade IDS[6]. It still retains its early nature, though, and at present can be configured to run in four different modes:

- *Sniffer* mode, in which it reads packets from the network and displays them on the console;
- *Packet Logger* mode, in which it logs the network packets to disk;
- *Network Intrusion Detection System* mode, where it analyses network traffic looking for matches against a user-defined rule set;
- *Inline* mode, by getting packets from `iptables` (Linux) or `ipfw` (FreeBSD), instead of from its usual packet capture source, `libpcap`, and eventually dropping packets if some inline-specific rules match.

The heart of Snort as an intrusion detection system is its signature-based, or rules-based detection engine, which compares each network packet against a set of rules, looking for matches (IP addresses, ports, packet headers, payload characteristics, etc.), but the usage of packet preprocessors for traffic normalization is essential to avoid techniques for evading the IDS itself, such as those that were described in a seminal paper by Thomas Ptacek and Timothy Newsham [25].

One of the aspects which make Snort appropriate for usage in a research project is its architecture. The software uses a layered architecture, which in some layers is plug-in based. This makes the task of changing some parts of the program without affecting other parts somewhat easier. Figure 2 shows a high-level representation of this architecture. Next, each of these layers is briefly described.

[5] `http://www.snort.org/`

[6] Snort itself is commercially supported, as it is the core product of *Sourcefire* (`http://www.sourcefire.com/`), a company founded by Martin Roesch for this purpose.

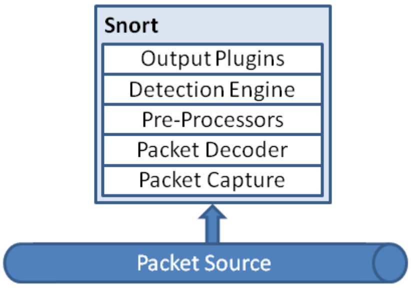

Figure 2 – Snort's layered architecture.

Packet Capture: Snort traditionally used `libpcap`[7] to capture raw packets from the network. This choice favored Snort's portability, as it released its developers from having to worry about operating system specific interfaces for retrieving packets. More recently, Snort started using Linux's `iptables` and FreeBSD's `ipfw` as packet sources, for *Inline* mode of operation. This allows it to actively drop packets if an attack is detected.

Packet Decoder: The packet decoder is the component responsible for analyzing a packet and setting up a Packet structure, which is essentially a structure with pointers to various parts of a packet. The decoder layer actually consists of a number of different decoders which are called according to the packet source interface type (Ethernet, PPP, etc) and the protocols encapsulated in the packet (TCP, UDP, etc).

Packet Preprocessors: Preprocessors are plug-ins which normalize and/or examine traffic in ways beyond the abilities of the Snort rules. Most of the program state (the active/processing data in the generalized IDS architecture) can be found in the preprocessors (IP fragments, partial reassemblies of TCP sessions, etc.). The amount of state data kept by each preprocessor is usually configurable and can become quite large. For example, the maximum memory cap just for TCP packet storage can be set up to 1 GB. Preprocessors let the original packet through, and when a reassembly is complete, generate a pseudo-packet, containing the normalized data.

Detection Engine: The detection engine is in reality a preprocessor which has the privilege of being the one which is called last. The detection engine is where the Snort rules are evaluated. It is also plug-in based, where each keyword option of a rule is a different plug-in. Most of the alerts should come from the detection engine, but the preprocessors and even the packet decoder can also generate alerts.

Output Plug-Ins: Output plug-ins, as the name implies, handle Snort's output, which is divided in alert and logging subsystems, for handling intrusion detection system alerts and for logging all captured network traffic, respectively. Different output plug-ins handle different data formats (text, binary, XML) and storage back-ends (files, databases).

[7] `http://www.tcpdump.org/`

Program flow is closely related to this architecture. After an initialization phase, the program enters a loop in which each packet captured by `libpcap` passes through all layers, possibly getting logged or generating one or more alerts. Housekeeping tasks such as signal handling are deferred until there are no packets left to read.

The current stable version of Snort uses threads, but only in a limited number of situations, all of which are related to configuration initialization or configuration reloading. It is therefore unable to take advantage of today’s very common multi-core processor architectures, leaving available processing capacity somewhat unused, even for typical dual core desktop computers.

2.3 Fault Tolerance

Fault tolerance is an essential requirement for achieving dependable systems. The original definition of dependability is the ability to deliver service that can justifiably be trusted [26], which in the end is what one intends to obtain. As mentioned in the beginning of this thesis, the world is increasingly reliant on information systems and communications networks, and so the need for dependable systems is generally increasing.

One possibility for achieving fault-tolerant services is to build software on top of fault-tolerant hardware, at least for some application classes, and this has been successfully pursued by companies such as Tandem and Stratus [27]. However, economic factors have motivated the development of cheaper software-based fault-tolerance solutions, and in particular in solutions based on replication.

In this section, some fundamental fault-tolerance concepts are briefly presented, followed by a discussion of the two main classes of existing replication techniques, the *primary-backup replication* model and the *active replication* model, which are able to ensure the correctness criterion of *sequential consistency*.

2.3.1 Fault Tolerance Concepts

Before continuing the discussion, a number of relevant concepts in the field of fault tolerant systems need to be introduced. These include the concepts of faults, errors and failures, of failure and communications models and of some correctness criteria:

Faults, errors and failures: Starting by defining what constitutes a fault, it must be said that this not a totally unambiguous term. As Cristian [28] notes, "what one person calls a failure, a second person calls a fault, and a third person might call an error". A fault is usually defined as a defect at the lowest level of abstraction that may cause an error, which in turn is a category of the system state that may lead to a failure, meaning that the system is no longer functioning according to its specification [29].

Failure model: When trying to build a fault tolerant distributed system, it is important to define what failures the system is supposed to withstand. Failures are commonly grouped in *failure models* or *failure classes*. In a typical distributed system, where a set of processes communicates by exchanging messages, processes are usually considered to either be correct (behaving according to their specifications), crashed (stopped receiving or sending messages) or behaving maliciously (sending messages that do not follow the specification) [27], which is also usually referred to as *arbitrary* or *byzantine* behavior. For a more complete set of failure classes (both of process and communication channel), see for example [30].

System model for communications: The communications channel for the exchange of messages between processes may have a known bound on the delay of a message transmission. In this case, the channel is said to be *synchronous*. If no such bound exists, and messages can suffer arbitrary delay in their transmission, the channel is said to be *asynchronous*. The asynchronous model is more general than the synchronous one, and more adapted to what can be expected of the behavior of most existing communications channels.

Correctness criteria: When dealing with replicated objects, several correctness criteria can be applied. The strictest correctness criterion is called *linearizability*, and aims at transmitting to clients the illusion of a non-replicated object, which is why it is also called *one-copy equivalence*. A replicated service is said to be linearizable if any sequence of operations done by multiple clients on the replicated objects meets the specification of a single correct copy of those objects, and the order of operations is consistent with the real times at which the operations occurred. A less strict criterion, *sequential consistence*, keeps the first condition (correctness), but only demands that the order of operations done by each client must be kept [30].

2.3.2 Primary-Backup Model

In the *primary-backup*, or *passive replication* model, one replica is selected as the primary replica for the service, meaning that, in its purest form[8], this replica receives requests from clients, processes the requests, responds to clients and propagates changes to the secondary or backup replicas. If the primary replica fails, one of the backup replicas is promoted to the role of primary replica.

Figure 3 shows a representation of the way the primary-backup system handles an incoming request from a client process p_j, if no failures occur:

[8] A common variation of this model is to allow clients to query backup replicas in read-only operations.

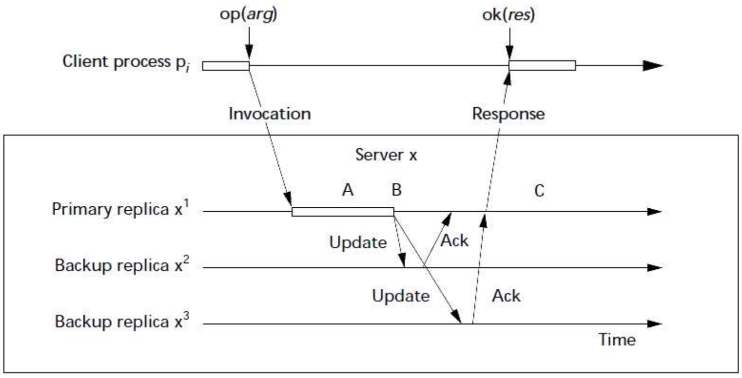

Figure 3 – Primary-Backup replication model (from [27]).

It is easy to see that this model guarantees linearizability, as long as the primary replica is correct, as it behaves in the same fashion as if only a single replica existed. Also, since the primary waits for the acknowledgements of the other replicas, before returning the response to the client, it is guaranteed that all correct (non-crashed) replicas have the same state.

In a failure scenario, however, things get more complicated. For the purposes of this discussion, it is sufficient to say there are ways of guaranteeing the system correctness, despite failure of the primary, using either synchronous or asynchronous system models. But one other problem still exists: if the primary replica fails, during the time it takes for the backup replicas to detect the failure, to agree on the correct state and to select the new primary replica, we are left with problem of service availability.

2.3.3 Active Replication Model

In the *active replication*, or *state-machine* model, replicas are seen as state machines and all of them play equivalent roles, unlike the ones in the primary-backup model. A state machine consists of state variables, which encode its state, and commands, which transform its state. Each command is implemented by a deterministic program, and execution of the command is atomic with respect to other commands, modifying the state variables and/or producing some output [31].

Figure 4 shows a representation of the way the active replication system handles an incoming request from a client process p_i, if no failures occur:

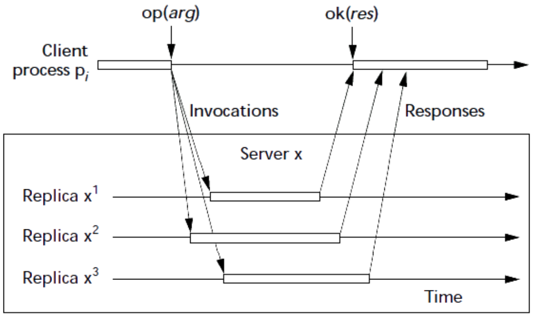

Figure 4 – Active replication model (from [27]).

In the context of this discussion, commands result from the processing of received messages by the replicas (servers). From the properties described above for state machines, it is easy to see that as long as correct replicas receive the same set of messages, and process them in the same order, their internal state will be the same. Active replication therefore relies on a communication primitive that guarantees the required properties of order and atomicity, and this primitive is called *total order broadcast* (also known as *atomic broadcast*; see [32]).

Active replication does not achieve linearizability (the total order in which correct replicas process client requests can differ from the real time order at which clients made those requests), but does achieve sequential consistency (if a client sent request 1, then request 2, the total order primitive guarantees that all correct replicas process request 1, then request 2) [30]. It is also able to cope with the byzantine failure of up to *f* replicas: if at least *2f+1* replicas exist, the group, the client or a front-end/back-end server can select the right result.

Comparing the classic replication models with the replication scheme implemented in this thesis, the later bears some resemblance with the active replication model, as in both cases all participating replicas behave as state machines. However, while active replication relies on the total order broadcast communication primitive to ensure message ordering, the replicas in the proposed system will instead rely on the underlying hardware to ensure this ordering, but have to deal with omission failures.

2.4 The State Saving/Transferring Problem

One of the issues that would need to be solved to implement a completely working system would be to have the ability to add new replicas to a running cluster. This would be required, either for increasing the number of active replicas in the cluster, or to recover a crashed replica.

For the state of the intrusion detection component of a replica to become aligned with the state of the already running replica intrusion detection systems, a newly launched IDS would need to process the same packets that were previously processed by its peers. Even putting aside issues concerning the required time to process all previous packets, most likely, the majority of those packets would no longer be available in the buffers of the other replicas, and so this solution does not appear to be an option.

A different approach could be taken: to transfer the state of an already running intrusion detection system to the newly launched replica. This presents a problem which can be split in two: saving the state of one replica and actually transferring it to another one. Trying to save the state of a program, and in particular of an already existing and complex program, can prove to be challenging. Two approaches for solving this problem were considered, each with its advantages and drawbacks: in this thesis, they are referred to as the "transparent box" approach and the "black box" approach.

2.4.1 "Transparent Box" Approach

In this approach, the program code needs to be analyzed as extensively as necessary to ensure all state-related data structures are known. This requires a deep understanding of the program's internals, namely of program flow and of state-related data structures, so no important state variable is missed, and no irrelevant ones are saved. This applies to the first part of the problem (saving the state), and would present a highly demanding task, considering Snort's complexity.

Concerning the second part of the problem (transferring the state), it should be noted that while many modern programming languages such as Java and C# provide support for serialization, C (in which Snort is written), does not. Serialization, or mashalling, is the process of taking a collection of data items and assembling them into a form suitable for transmission in a message [30]. Deserialization, or unmarshalling is the reverse process, which is performed on the receiver side.

While a few libraries exist that attempt to support serialization in C, such as tpl[9], libc11n[10] and gSOAP[11], it was not certain if any of these would able to deal with any data structure that could appear in a program as complex as Snort, or just with relatively simple ones. This means that functions for serializing unsupported data structure types might have to be done by hand.

Besides the amount of work that would be required, this approach has one further disadvantage: it would require keeping track of all future modifications done to the official program tree, so that newer versions could be used without missing newly introduced relevant state variables.

2.4.2 "Black Box" Approach

In this approach, the program is viewed as extensively as possible as a "black box", and a process checkpointing package is used to capture the process memory image. This approach makes particular sense if one considers that the bulk of the process's memory image is used to keep state related data (in a network intrusion detection system, appears to be the case, considering the need to keep numerous IP fragments, partial TCP streams, and so on, as long as possible – with long in this case being translated into "available memory space").

Process checkpointing packages, also known as process checkpoint/restart, or C/R, have been available for over 15 years (e.g. [33]). The usefulness of process checkpointing is varied: checkpointing a long running process, such as a batch job, for the event that it or the computer in which it is running crashes, saving the workspace of an interactive session, migrating processes between nodes in a cluster, or submitting an application with a CPU-intensive first

[9] `http://tpl.sourceforge.net/`

[10] `http://s11n.net/c11n/`

[11] `http://gsoap2.sourceforge.net/`

phase to a high performance node or cluster, and later migrating it to a laptop for an interactive analysis of the results (see [34] for further examples).

A number of approaches to process checkpointing can be taken: kernel-level, user-level and application-level checkpointing. A closely related topic is that of a complete virtual machine saving and restarting, of course a much more "heavy-weight" task. While application-level checkpointing refers to a checkpointing implementation which requires non-trivial modifications to the target application [35], kernel-level and user-level checkpointing packages intend to provide a more general support.

Comparing just the two types which can be used for supporting a generic application, they both have different levels of transparency, complexity, portability and performance characteristics. Transparency means whether user applications need to be modified, recompiled or re-linked. Generally speaking, adding support to the kernel leads to better transparency, but more implementation complexity and less portability [36].

One problem with both checkpointing types is that their implementation is to a high degree connected to the platform in which it is intended to run, making them hard to maintain. This is especially true in kernel-level checkpointing, since kernel modules are hard to maintain because they directly access internals of the kernel, which change more rapidly than standard APIs [34].

As a result of this problem, many implementations stop being maintained, and developers no longer work in subsequent versions of the targeted platform. See for example the website `http://www.checkpointing.org/`, which shows a large number of implementations that do not work in current platform versions. Currently, the most promising implementation for the Linux platform, in terms of features, actuality and maintenance seems to be the Distributed MultiThreaded CheckPointing (DMTCP) project [35], [34], and its stand-alone MultiThreaded CheckPointing (MTCP) component.

Comparing the "black box" approach with the "transparent box" approach, it should be noted that while the "black box" approach seems more appealing concerning implementation simplicity and compatibility with future software versions, it become a much more heavyweight solution in the state-transfer phase, as it's not possible to distinguish what portions of state are unnecessary to save and transfer. As was discussed, it makes sense in particular if the bulk of the process's memory image is used to keep state.

Such is the case in the problem at hand, as was previously mentioned, and so the "black box" approach and the MTCP implementation were the options selected for this project.

3 DESIGN

In this chapter, the general design concepts for implementing the synchronization mechanism between replicas are presented. The chapter starts with a description of the way replicas confirm their alignment, and of how they recover when packet loss occurrence is detected. Next, the way the intrusion detection system receives packets for analysis, from the synchronization layer, is explained. The chapter ends with the presentation of the proposed synchronization algorithm, which is the central algorithm of the project.

3.1 Packet Synchronization Mechanism

As previously illustrated, a single intrusion detection system may be setup in a number of different ways. The developed fault-tolerant system was designed considering the common setup of an intrusion detection system which listens to only one network interface, from which it receives packets to be analyzed, connected to some sending interface. It is assumed that the listening network interface is not used by the system for any purposes other than traffic analysis, and in particular that the system does not send out traffic that was generated by itself, through that interface.

In such a setup, the single intrusion detection system can be replaced by the fault-tolerant system, composed by a set of replicas, each running a synchronization layer which in turn feeds its IDS instance. All replicas should receive the same network packets, sent by the sending interface, and duplicated by some device which behaves as a multiport repeater, such as an Ethernet hub. A second, separate interface is used by each replica for the required system synchronization tasks, and any other functions, such as system control, monitoring or reporting.

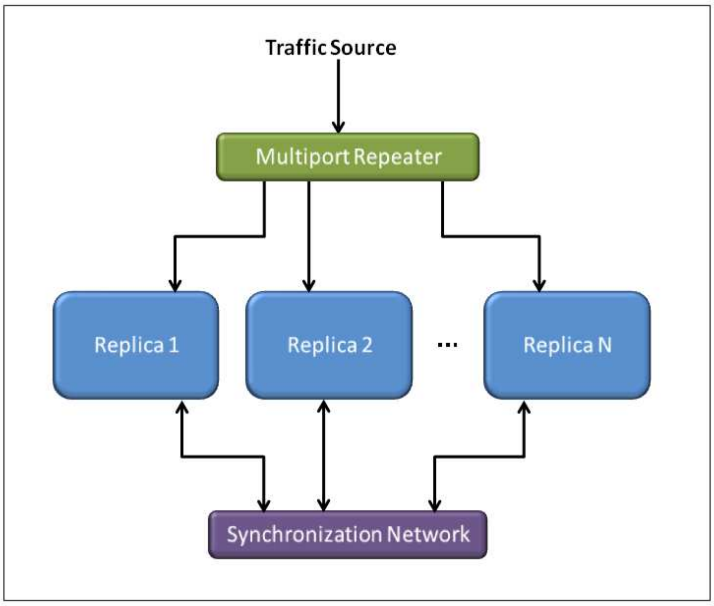

Figure 5 – Fault-tolerant system overview.

Considering that all network components behave as FIFO's, then all replicas synchronization layers receive the same packets, and in the same order, unless some component is unable to deliver one or more of them to the synchronization layer – an omission failure. A common reason for such a failure is that the consumer component – the stand-alone IDS or in this case the synchronization layer – is unable to pick up and process packets quickly enough, leading to the filling of buffers of other components, causing them to drop subsequent packets.

In a system composed by multiple replicas, each receiving and processing the same network traffic, omissions, if they occur, will not necessarily happen in the same way on each replica, due to differences in the load and pace of independent, asynchronous systems, even if the main pace-setting factor, the stream of packets to be analyzed, is the same for all the replicas.

The proposed fault-tolerant systems attempts to take advantage of this by combining the data received by each of the replicas and trying to detect if packets were lost by one or more of them. If it is able to detect that packet loss has occurred, it then tries to infer, as best as it can, what the original packet stream must have been. To achieve this, a compact representation of each received packet is computed and stored, using a hash function. For every *k* packets, a compact representation of the corresponding set of hashes is computed in the same way, and then exchanged and compared by all replicas.

If the system is running in the absence of faults, all replicas are able to receive and process the same packets. Since the hash function is deterministic, the computed hashes will be the same in all replicas, and so will be the hash of a given cluster of *k* hashes.

Figure 6 illustrates a cluster where packet loss has not occurred:

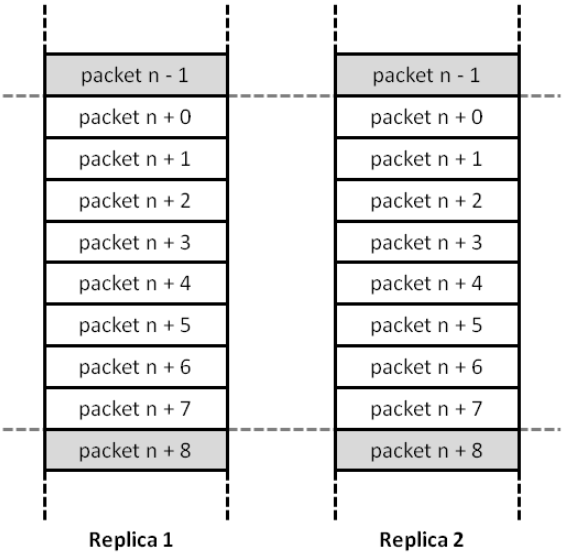

Figure 6 – In a two-replica system running in the absence of omissions, all packets are received by both. Each computes and exchanges a hash of *k* = 8 hashes of individual packets, which will match.

This will also hold true if all replicas miss exactly the same sub-set of original packets, but in this case the fault cannot be detected. If, on the other hand, omissions occur which do not involve exactly the same sub-set of packets, the hash of *k* hashes will differ between replicas, and the fault becomes apparent.

Once detected, replicas exchange the hashes of individual packets and run a deterministic algorithm to determine which packets are common, which are different, and how should the differences be merged. Such an algorithm could have been developed during this project, but by viewing the sequence of hashes of each replica as a sequence of lines on a text file, the job of determining the best way to merge the sequences can be left to a highly established tool for this type of task: the `diff` utility.

Figure 7 show a cluster where packet loss has occurred:

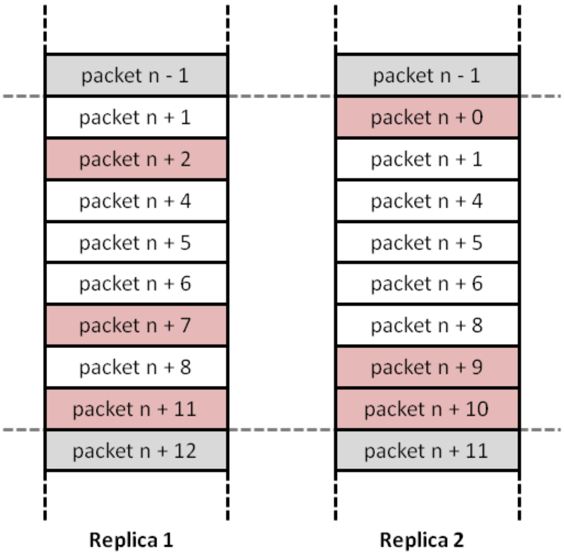

Figure 7 – A system loosing packets. A given packet, present in the original stream, may be missed by neither, one or both replicas.

One issue arises from the fact that the synchronization algorithm is working in blocks of packets: once misaligned, a point of realignment may not be found within a given block. In this case, the *k* hashes of packets from that block may be merged differently by `diff`, according to the choice of letting or not letting it see hashes of packets beyond that block. In the above figure, for example, `diff` could choose to merge packets starting from *n + 8* in the order *n + (8, 11, 9, 10)*, if it was not given the chance to see that packet hash *n + 11* was in fact present in the next block of hashes of replica 2.

3.1.1 “Anchor” Definition

For this reason, the concept of an “anchor” is introduced: an “anchor” is defined as a block of a chosen size of *l* contiguous packets which are present in all replicas. If such a block exists, the replicas are considered to be aligned in that block of *l* packets, and `diff` can be used to determine the best way to merge all hashes of packets which came before it. If such a block does not exist, on the other hand, then replicas are considered to be misaligned in such a way that using `diff` to merge the hashes is not regarded as valid, and further *k* packets need to be considered before attempting to realign the replicas. If multiple "anchors" exist, then the one used for alignment is the one closest to the end of the block being evaluated.

Supposing that replicas are aligned up until packet *n - 1*, if the loss of packets is detected by the replicas, due to differences between the hashes of hashes from the current block, and replicas retrieve the individual hashes of that block, then three different cases can occur:

i. An "anchor" exists at the end of the packet set of one or more replicas, while those replicas are not missing packets from any other replicas, before the occurrence of that "anchor", as is shown in figure 8. In this case, it is concluded that those replicas did not miss any packet, and have no recovery to do, while the others need to retrieve the missing packets (before the "anchor") from their peers which are complete in their packet set. It should be noticed however, that the fact that an "anchor" exists at the end of the packet set of a replica is not enough to conclude it has not missed packets.

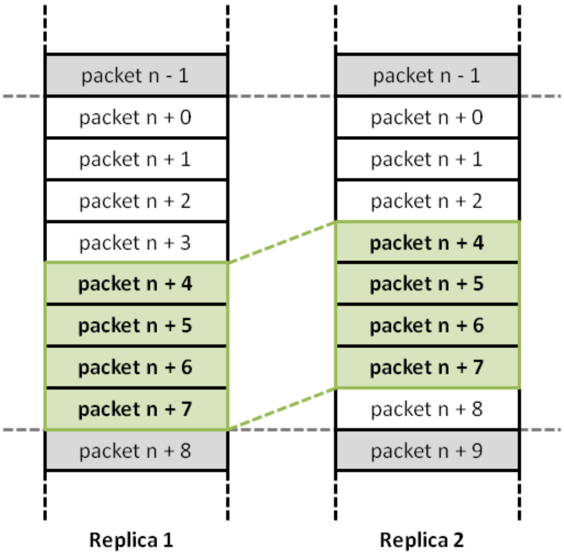

Figure 8 – An "anchor" of *l* = 4 packets is found at the end of the packet set of replica 1, while it is not missing packets from replica 2, before that "anchor". In this case, replica 2 is the one missing packets, and all extra packets it has in the current block, compared with those of replica 1, appear after the "anchor".

ii. An "anchor" exists, but it does not meet both previous conditions. In this case, it is possible that one or more replicas did not miss any packets, but this cannot be concluded yet. It is safe, however, to realign replicas up until the "anchor". To conclude replicas are again completely realigned, at least the next *k* packets need to be evaluated, as one or more replicas could have lost packets after the "anchor".

This is what happens in the case show in figure 9. Only before the "anchor" can packets be realigned (in this case, packet *n + 1*). Packets after *n + 7* will only be realigned after this round.

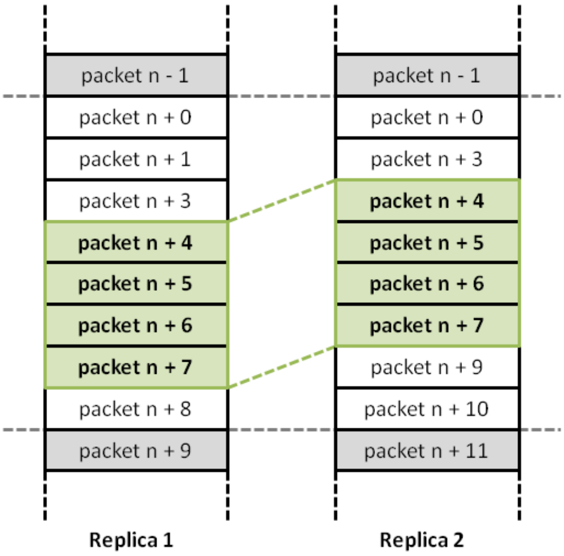

Figure 9 – One or both replicas have lost packets, but none of them has an "anchor" at the bottom of its packet set. While it is certain that replica 2 has lost at least packet *n + 1*, which can be recovered from replica 1, it is not know, until further packets are considered, whether or not replica 1 has lost packets. In reality, in the given scenario, both have lost packet *n + 2*, but this they will never be able to find out.

iii. An "anchor" is not found. By the given definition, no recovery can be done by any replica, until at least the next *k* packets are evaluated and one is found.

3.1.2 Desynchronization Level

When replicas are misaligned, they are said to be desynchronized. When a desynchronization first appears after a synchronization round, a level 0 desynchronization is said to occur. If it is possible to conclude that at least one replica did not lose any packets, the system can be realigned immediately, up until the current block, using that information. If not, the next *k* packets are evaluated, together with the first misaligned block, in a level 1 desynchronization, and so on, until an "anchor" can be found and it is possible to re-synchronize some or all of the missed packets.

Regarding the anchor, the same conclusions that could be taken in a desynchronization level 0 can be taken if that level is higher: a desynchronization level 1 is for the most part equivalent to a desynchronization level 0 of a block of *2k* packets. It is possible however, that the realignment cannot be completed, but that the next step happens again at desynchronization level 1 (or even 0), if an "anchor" is found and enough packets are resynchronized, so that the first block becomes completely synchronized.

Figure 10 illustrates a desynchronization level 1:

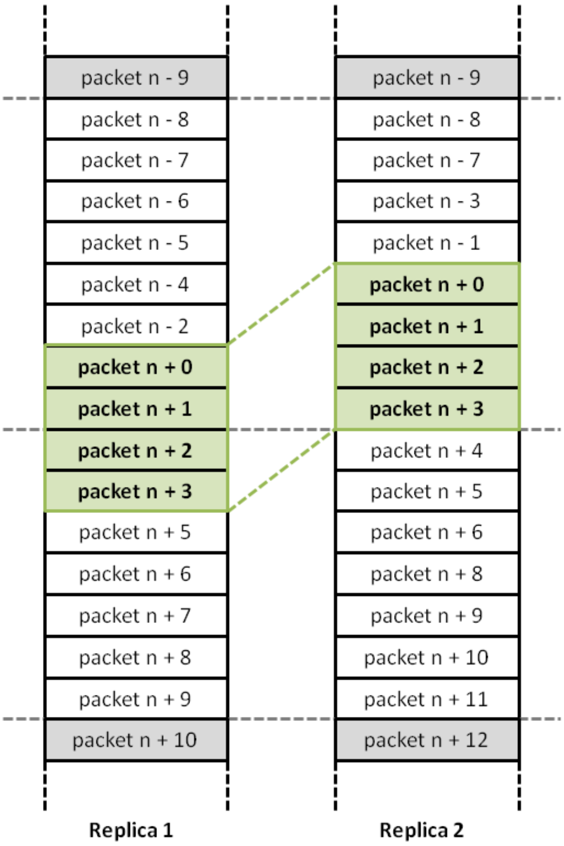

Figure 10 – Replicas at desynchronization level 1, in block *i*. An “anchor” is found and some packets can be realigned. After that, block *i – 1* will become completely realigned, and so the next round will again be at desynchronization level 1, not level 2.

3.1.3 A Few Notes

Starting from an aligned state, the synchronization mechanism verifies if replicas have lost their alignment every *k* packets. This requires them to exchange, as was mentioned, only a single hash representing those *k* packets, in order to check if a resynchronization is necessary, instead of having to exchange all *k* hashes or even worse, the actual packets, until that exchange is confirmed to be necessary. It should be noted, however, that verification and realignment cannot be done independently for every block of *k* packets: a single omission on any given block will cause all subsequent blocks to be misaligned. This means that only after block *i* is verified and aligned can block *i* + *1* be tested for alignment.

Figure 11 gives an example with k = 2 packets:

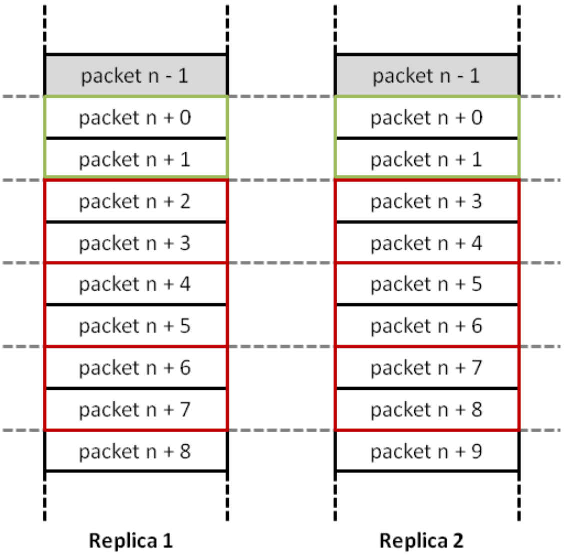

Figure 11 – Two replicas using a block size of *k = 2*. If comparison between corresponding blocks is made before synchronization, all blocks will appear misaligned after the first omission, unless by some chance both replicas were to miss the same number of packets on a given interval. This means that they cannot be compared and resynchronized independently.

Once misaligned, it is therefore necessary to find a new synchronization point, which translates into finding an "anchor", and to transfer missing packets prior to that "anchor", between replicas, before restarting the process of comparing only the hashes of hashes. Packet hashes could be added one by one, until such an "anchor" was found. The choice, however was to always add *k* more hashes until one is found.

When an "anchor" is finally found and packets are realigned, the next block can be set to start at the first packet after the "anchor", as replicas will be aligned up to its last packet. In the current implementation, however, the synchronization mechanism will again always move in blocks of *k* packets, which means that sometimes hashes of packets known to be aligned will appear in a next synchronization round.

3.1.4 Bootstrapping

So far, the way replicas verify if they remain aligned, and the way they recover from misalignment, once detected, was described. In this section, the bootstrapping of replicas is explained.

Since in most production environments replicas would start analyzing traffic on a live network link, it is easy to see, that only by a small chance would two or more replicas start their processing work at exactly the same packet, once launched. This means one cannot let them compare the hash of hashes of the first *k* packets each has collected and expect them to be correctly aligned.

Currently, bootstrapping the system is performed if the following manner: the first replica is launched as the leader replica. Instead of immediately starting to process packets, it waits until a specified minimum number of peers have connected to it, or a specified number of seconds have elapsed since it was launched. Each connecting peer is required to start capturing packets before connecting to the leader. This ensures that, once the leader starts collecting packets, the other replicas participating in the system bootstrap have at least as may packets as the leader replica.

Once the leader starts the boot process, it first collects a specified number of packets, say *m* packets, and then sends a list of all their corresponding hashes to the other peers. Each peer attempts to find a sequence of hashes of packets matching the received list, starting from the last *m* hashes in its own hashes list, and moving to the beginning of the list. If the match exists, the replica is considered to be aligned with the leader up to the last hash of the received list, and the process of comparing hashes of blocks of *k* hashes starts with the first packet captured after that.

In the current implementation, the number *m* of packets used for the bootstrap is taken from the number of *k* packets which make up a block, but there is no requirement for this. The "anchor" size, or another unrelated value, could have been used. Using *k* to find a match in the list of hashes may in fact be quite expensive, in particular if the value chosen for *k* is large.

3.2 Feeding the IDS

A stock Snort intrusion detection system will read a live packet stream using the `libpcap` packet capture library. The developed synchronization layer is meant to sit between the `libpcap` packet capture library, which it also uses, and a modified version of the IDS. It captures packets, synchronizes them between replicas, and provides them to the intrusion detection system.

The choice for this later task was to use a shared memory segment and shared semaphores, which provide a fast and flexible form of inter-process communication (IPC). Captured or resynchronized packets are copied into the shared memory segment, and the intrusion detection system is signaled whenever a new packet is available to be analyzed.

3.2.1 Writing Packets in the Shared Memory Segment

Packets are written sequentially in the packet queue area as they are made available to the enqueueing function. In this way, resynchronized packets do not need to be inserted between packets captured by the capture library, but this also means that they are not placed in the shared memory segment in the same order that they were sent by the sending interface. For this reason, in addition to copying all the data that the IDS would receive from `libpcap`, information regarding the location of the next packet that it should consume is also inserted.

Together with the actual packet data, the intrusion detection system reads this information, which it will use to find the next packet. The task of ensuring the sequence it will follow is correct is left for the synchronization layer, which needs to update the address information of packets whenever they are resynchronized from other replicas.

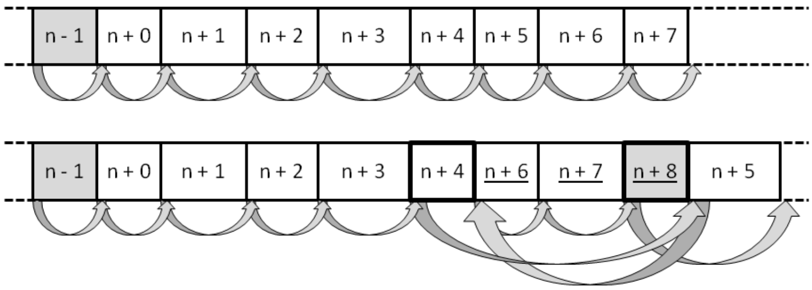

Figure 12 – Shared memory packet insertion. When a newly resynchronized packet *n + 5* is inserted in the shared memory queue, address information needs to be updated. Next address information for packet *n + 5* is taken from the previous packet slot, *n + 4*, which in turn is updated to point to the newly inserted packet. Also, the address information on the previously last inserted packet needs to be updated to point to the first free memory address, after *n + 5*. The old packets *n + x*, with x >= 5, become packets *n + x + 1* in the sequence that will be read by the IDS.

The shared memory segment serves the purpose of buffering packets until they are synchronized and the intrusion detection system is able to process them. It also serves the purpose of extending the buffering ability provided by the capture library, which is important if at some moment the IDS cannot be fed by the synchronization layer, or itself cannot process packets at the rate with which they arrive from the network. Obviously, this buffering ability is not unlimited.

3.2.2 Optimistic Modes

Packet enqueueing in the shared memory segment and the point at which that task becomes too slow, leading to the dropping of packets by the capture library, depends on various factors including queue size, traffic rate, synchronization speed, intrusion detection system consumption rate, and also the way the latter is allowed to consume them.

These factors led to the provisioning of four different operating modes, named optimistic operation modes, which are described in the next sections. The first two operating modes are considered pessimistic modes of operation, while the latter two are considered optimistic modes. The first one was created for the most part as a simplification to facilitate development, while the last one is currently not implemented.

What separates the pessimistic modes from the optimistic ones is that in the first case the IDS is only allowed to process packets after they have been synchronized between replicas, while in the second case packets may be analyzed by the intrusion detection system even before that happens, optimistically considering that replicas will be confirmed to be aligned, at the risk of having to roll back the IDS up to a point where it had only processed correctly aligned packets.

3.2.2.1 Optimistic Mode 0 or "Very Pessimistic Mode"

In this mode, on each replica, a single thread is responsible for collecting from the packet capture library the necessary number of packets for a given synchronization round - either *k*, if no packets are left over from the previous round, or *k* minus the number of packets that were pushed forward in the previous round(s), due to insertion of resynchronized packets - and then for synchronizing with its peers. This later part includes confirming whether or not replicas are synchronized, and if not, getting all the required data (hashes and packets) for realignment.

Another thread, or threads, if more than two peers exist, are also involved, which are the threads responsible for answering to the corresponding requests arriving from the replica's peers.

3.2.2.2 Optimistic Mode 1 or "Pessimistic Mode"

In optimistic mode 1, another thread is introduced. It is a thread dedicated to the task of capturing packets using `libpcap`, and enqueueing them in the shared memory segment. This for the most part decouples the task of picking up packets from the tasks related to replica synchronization, reducing the chances that packets are dropped by the capture library due to delays in the synchronization thread.

Optimistic mode 1 is considered less pessimistic than the previous mode in the sense that packets are collected and enqueued from `libpcap` at a pace which may prove to be unsustainable, in case the synchronization thread is not able to keep up with that pace and the packet queue ends up filling completely, before the first collected packets are no longer necessary.

3.2.2.3 Optimistic Mode 2 or "Optimistic Mode"

This mode is the first of two in which packets may be consumed by the intrusion detection system before alignment verification. Since the main goal of the project is to study approaches in which the IDS packet loss is minimized, the consumption of unverified packets is not introduced as a failback mechanism, in cases where the synchronization pace is too slow. Instead, it is introduced as a way to cope with scenarios where analysis delay may be critical.

In a typical deployment of an intrusion detection system strictly as an IDS system, as long as the synchronization mechanism does not fall back in pace, compared with traffic capture pace, in such a way it is never able to recover, the delay introduced by this mechanism should be irrelevant. For example, in a "slow" link with a packet rate of one thousand packets per second, and with a cluster size of one thousand packets, the first packet would suffer a delay of one second before being processed, and so would the generation of an eventual alert generated by that packet. Either a higher traffic rate or a smaller chosen cluster size would lead to an even smaller delay.

While a delay of one second to alert a human operator may be deemed as irrelevant, to a machine it may be very significant. If the intrusion detection system is used as an in-line IPS, or to control an external firewall, any delay may allow the passing of many packets that would, without usage of the synchronization layer, be blocked. This provides a reason for letting the IDS analyze packets as soon as possible.

Since the main goal, as was mentioned, is to minimize packet loss, replicas are considered to be “incorrect” whenever it is found they have processed packets while missing some. To recover from this “incorrect” state, checkpoints of the intrusion detection system state are periodically taken, and the IDS is rolled back to a “correct” one, and allowed to consume aligned packets, whenever misalignment is detected and the IDS has processed packets beyond a point where they had been confirmed to be aligned.

In optimistic mode 2, the intrusion detection system is instructed to take a new checkpoint, replacing the previous one, whenever a new synchronization point is established and the IDS reaches that point. If the IDS is too behind and has not yet reached the previous synchronization point, then the current one is skipped. This is due to the fact that check-pointing introduces delay, and if the intrusion detection system is falling behind, skipping checkpoints favors its recovery.

On the other hand, the previous synchronization point is not moved forward to the current one, as doing this could result in always moving the synchronization point before the IDS could reach it, thereby never taking checkpoints.

However, if the intrusion detection system is instead able to process packets ahead of the current synchronization point, it is not allowed to consume those that are beyond the current synchronization round. This is because doing so, once a new synchronization point was established, the IDS could already be ahead of it, and taking a checkpoint would include packets beyond that point.

3.2.2.4 Optimistic Mode 3 or “Very Optimistic Mode”

As previously mentioned, optimistic mode 3 was not implemented in this project. In this mode, the intrusion detection system would be permitted to consume packets as soon as they were received. This would mean letting it consume packets beyond the current synchronization round, and the implementation would consist of not only optimistically letting it consume packets unknown to be aligned, but also optimistically taking checkpoints of the IDS every *k* consumed packets, checkpoints whose correctness would be unknown.

This optimistic mode would allow, for the most part, the decoupling of the packet consumption pace from the packet synchronization pace.

3.2.3 Recovering Shared Memory Space

Once packets confirmed to be aligned have been used by the intrusion detection system, and that the intrusion detection system will not need even if rolled back, the corresponding space can be made available for other packets to be written. This is done by dividing the packet queue in virtual slots, each slot corresponding to the maximum space a packet may take.

The required space for a packet includes the address of the next packet and the information provided by `libpcap`, which consists, besides the packet itself, of a `char` pointer and a header structure `struct pcap_pkthdr` (defined in `pcap.h`) with information regarding the time of packet capture and the length of the capture. The length of the capture may differ from the size of the packet, because the maximum capture size may be limited by setting a *snaplen* value, which specifies the maximum number of bytes to capture [37]. For Snort, the default maximum packet capture size is 1514 bytes.

Slots are virtual in the sense that, to optimize space, each next packet is written right after the end of the last one, and not written in the next slot, which would waste a lot of space, due to the fact that many captured packets will be smaller than the size of a slot. Taking the case of Ethernet frames, using the default Snort *snaplen* value (which was in fact used in this project) of 1514 bytes is suitable for capturing packets on a typical Ethernet link not using *jumbo frames* (which are Ethernet frames exceeding 1500 bytes of payload [38]). But for example the designers of Snort's Stream5 pre-processor base some of their default values on an estimated average size of 400 bytes per TCP segment [39]. In cases where this estimation is verified, using a fixed slot space would waste more than one Kbyte (depending on IP header size) per packet, on average.

The choice of using a fixed slot size to free space relates to the usage of a semaphore to control the progress of the enqueueing function. If slots are available, the semaphore is loaded with the corresponding value. If not, the function has to wait for slots to become available, at the risk of having packet dropped by `libpcap`.

Since the packet sequence in shared memory may include jumps due to the enqueueing of missed packets, free space cannot be computed by using the distance between the last written packet and the first packet that the intrusion detection system may need to (re)read in the packet sequence, as it may also need to jump back in the sequence.

Calculating free space is therefore done in three steps:

1. Starting from the packet where the previous iteration ended, the sequence of packets is followed up to the last packet included in the intrusion detection system's last checkpoint (the IDS registers these points, even in pessimistic modes). While following the packet sequence, since these packets will no longer be needed, the next packet address information is cleared.

2. In a second step, the physical packet sequence in shared memory is followed, starting from the point where the last iteration ended. The next packet location is computed from the packet capture length in the `libpcap` header, registered for the current one. Packets are followed until the first one without a cleared next address information is found, meaning it is not yet free.
3. The amount of freed memory slots is computed using the distance between the previous and current last free packet in the shared memory queue.

3.3 The Synchronization Algorythm

In the previous sections, the general ideas for the implementation of a synchronization mechanism for a set of replicas capturing and analyzing a common source of network traffic were presented. In this section, the main steps of the core algorithm of the implementation, the synchronization algorithm, are described.

The proposed algorithm is a recursive function, which on each call executes a given synchronization round, that is, it uses the next block of *k available* packets to test for replica alignment and to realign replicas if packet loss has occurred and it is possible to partially or completely execute that task.

It should again be highlighted that if a specific packet was missed by all the replicas in the system, then of course it cannot be recovered. From the system's point of view, and by extension, from the synchronization algorithm's point of view, it is as if that packet had never existed.

The synchronization algorithm is for the most part independent from the previously presented optimistic modes. It leaves the job of handling those differences for another function, whose purpose is precisely to coordinate the synchronization layer and the intrusion detection system layer, which is where the differences between optimistic modes are reflected.

The synchronization function receives as arguments the number of the synchronization round, the desynchronization level and the amount of packets left from the previous round. It is called from a perpetual loop, which starts from a synchronized state, without any credit in packets. It will only return when replicas are synchronized and there is no credit left, and so it is called from the loop with the last two arguments set to 0 (zero).

The recursive synchronization algorithm pseudo-code is represented in figure 13:

```
forever do
  cluster_num = cluster_num + SyncWithPeers(cluster_num, 0, 0);


synced SyncWithPeers(cluster_num, desynced, credit);
  synced = 1;

  if (credit < CLUSTER_SIZE)
    EnqueuePackets(CLUSTER_SIZE - credit);
    credit = CLUSTER_SIZE;

  UnlockPeersWaitingClusterHash();

  if (PeersSendClusterHash(cluster_num) || desynced)
    MyHashesSet(cluster_num, desynced);        // unlocks peers waiting to get my hashes
    all_hashes = AllHashesGet(cluster_num, desynced);

    anchor = DiffRun(all_hashes);

    switch (anchor)
      case -1:
        desynced = 0;
        SyncUpperLayer(OK);

      case -2:
        credit += EnqueueMissingPackets(CLUSTER_SIZE – ANCHOR_SIZE);
        desynced = 0;
        SyncUpperLayer(NOT_OK);

      case -3:
        desynced = desynced + 1;
        SyncUpperLayer(UNKNOWN);

      default case:
        credit += EnqueueMissingPackets(anchor);
        desynced = desynced + 1;
        if ((desynced + 1 - CountResynchronizedBlocks()) > 0)
          desynced = desynced + 1 - CountResynchronizedBlocks();
        else
          desynced = 0;

        SyncUpperLayer([UNKNOWN | NOT_OK]);

  else
    SyncUpperLayer(OK);

  if ((desynced > 0) || (credit > CLUSTER_SIZE))
    synced = synced + SyncWithPeers(cluster_num + 1, desynced, credit - CLUSTER_SIZE);

  return synced;
```

Figure 13 – Synchronization algorithm pseudo-code.

The function starts by setting its return value to 1, as at least one more round will have passed once it returns. It then makes sure at least CLUSTER_SIZE packets are available. These will need to be retrieved from the packet capture library, in optimistic mode 0, or may already be in the shared memory queue, in other modes.

It then unblocks any peers possibly already waiting to get the current hash of hashes. To be able to answer such a request, the corresponding packets must already be available, which was just ensured by the previous call to EnqueuePackets().

Next, the replica sends its own hash of hashes of the current cluster to the other peers. PeerSendClusterHash() will return true if at least one of the peers replies with a different hash. Either in this case, or if the desynchronization level at which SyncWithPeers() was called is greater than 0, replicas are considered misaligned and are required to exchange all individual packet hashes, corresponding to the desynchronization level, so that `diff` may be executed.

It should be noted that in the current round, replicas may happen to be aligned, but still have misaligned packets from previous rounds, and so checking if PeerSendClusterHash() returns false is not enough: the desynchronization level also needs to be false for replicas to avoid exchanging individual hashes.

If this needs to occur, the next step is to retrieve from the hashes database the list of hashes that will be given to other peers and only after that list is set may the replica proceed to retrieve other's hashes. This is because if others are too slow to retrieve those hashes, the local replica may not only already have retrieved other's hashes but also ran `diff`, gotten missing packets, and have modified its own hashes database, by the time others ask for the list, which will then correspond to the already synchronized packet list.

Once hashes are retrieved, `diff` is run. DiffRun() will return either the position of the last anchor found, or a special case value.

If that value is -1, then the local replica has not lost packets while other(s) have. At the end of the round the system will be aligned (as there is at least one peer who has not lost packets). If the value is -2, then it's the reverse situation and the local replica needs to retrieve packets it has missed. A special value of -3 indicates that an anchor was not found and so the desynchronization level is simply raised by one.

The default case means that an anchor was found, but realignment may not yet be completed. Packets up to the anchor are retrieved, and if enough packets were already resynchronized, the next desynchronization level may be adjusted according to the number of resynchronized blocks, instead of just being incremented by one.

If many blocks are resynchronized, computing the desynchronization level could even lead to a negative value. Setting such a value is not allowed, of course, but what it means is that in the next round replicas are sure to be aligned, and that check could be skipped. In fact, the next cluster could always start after the last anchor packet. This is left as a possible future optimization.

After the replica synchronization tasks are complete for a given round, SyncWithPeers() calls SyncUpperLayer() with the result, which may be used differently according to the optimistic mode. For example, OK means the replica has not lost packets in all modes, while NOT_OK has the same semantics as OK for the pessimistic modes, but indicates the need for a roll back in the optimistic ones. UNKNOWN translates into letting the intrusion detection system process *k* more packets in the optimistic modes, and doing nothing in the pessimistic ones.

If the desynchronization level has become greater than 0, or if the number of packets in credit is greater than CLUSTER_SIZE (which are the ones corresponding to the current cluster of packets), SyncWithPeers() is called again.

Finally, the number of completed rounds is returned.

4 IMPLEMENTATION

In the previous chapter, the main ideas used for designing a synchronization layer able to verify replica alignment, to resynchronize missing packets and to provide them to an analysis layer were presented. In this chapter, the main components of the system are described and some details concerning the implementation of those ideas are explained.

The system is comprised of a number of equally behaving replicas, with the exception of the system bootstrapping phase which, as described in chapter 3, uses the first launched replica as a leader in that phase. Increasing the number of replicas in theory reduces the chance of a given packet to be lost by all replicas in the system, but it also increases the amount of work that needs to be done in the synchronization process, which in some cases may lead to more packets being dropped, therefore producing an effect opposite to the one which was desired.

Each replica is composed by two independent processes: one running the developed synchronization layer and another running the network traffic analysis layer. For this layer, a modified version of the *open source* industry-standard intrusion detection system Snort was used. The original IDS source code was modified in a number of ways, to accommodate the specific needs of this project.

It is important to mention that some elements which would be required to exist on a production system, but that were not considered fundamental for the purposes of this study, are missing in the current implementation. One such element is replica management: once the system bootstraps, it does not currently support the joining or leaving of replicas. Replica crashes are also not supported, and neither are byzantine failures. While algorithms for dealing with these issues have been studied and can be found in the literature, it is admitted that supporting crash failures may prove to be significantly challenging to handle in the problem at hand. This is because any small delay in crash detection may result in a very quick growth of the unsynchronized packets queue.

A second aspect which needs to be mentioned is the fact that currently, only two replicas are supported in a running system. This is because an n-way `diff` algorithm was not developed. The standard `diff` utility which was used is limited to comparing only two text inputs. While a `diff3` utility also exists, which takes three text inputs, it purpose is to reconcile two new versions of a file which were derived from a common ancestor [40]. This does not apply to the problem at hand, and `diff3` is not appropriate even for supporting three replicas.

A third element of the system which was not completely implemented was the alert handling infrastructure. In a production system, implementing this infrastructure would be required to filter out duplicate alerts generated by the system. The system may generate duplicate alerts due to the following reasons:

1. If the goal of the synchronization layer is to allow a number replicas to process the same set of packets, a given alert will be generated across all replicas upon detection, assuming they are running with the same configuration and the same set of rules;
2. In optimistic synchronization modes, some packets may be processed more than once, and because of this, a given set of packets may generate an alert more than once, whenever a roll back is involved.

Finally, due to the current existence of unsolved issues concerning the implementation of the freed slots recovery algorithm, all results presented in chapter 5 were made with a packet queue large enough to make its usage unnecessary.

4.1 Architecture Overview

As mentioned earlier, each replica is composed by two processes, which communicate via a shared memory segment channel. While the current stable version of Snort is single-threaded, the synchronization layer is multi-threaded, and coordination not only between the two processes, but also between threads belonging to the synchronization layer, was an important aspect of the implementation.

Figure 10 presents a representation of the major system components:

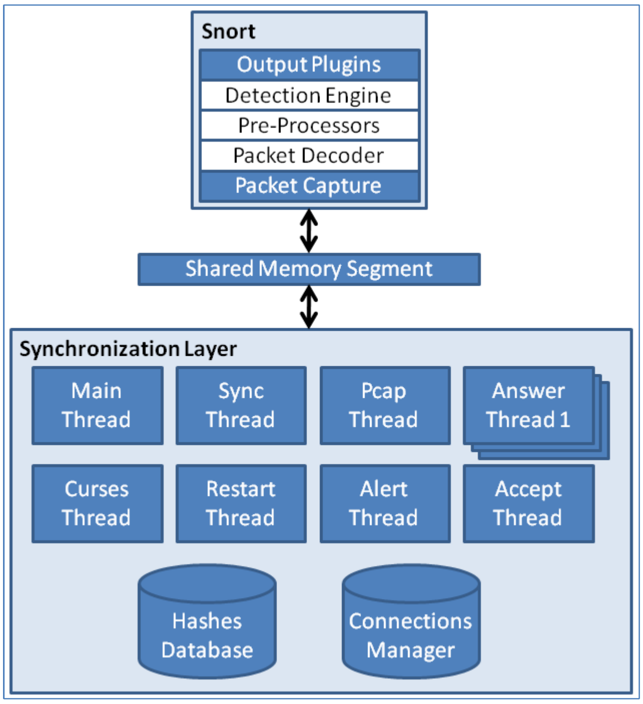

Figure 14 – Major replica components. Each replica is composed by two separate processes, a synchronization layer process and an analysis layer process. The two processes communicate using a shared memory segment and shared semaphores. For the analysis layer, the *open-source* industry-standard Snort intrusion detection system was used. The bottom and top layers of Snort required some level of modification for this project, and are for this reason highlighted in the figure.

As shown, the synchronization layer may execute up to eight separate threads, even if the system is run with only two replicas. When the implementation is able to support more than two replicas running in the system, several instances of the Answer Thread may be executed on each replica, one for each connected peer. Some threads, such as the Curses Thread, the Restart Thread and even the Pcap Thread may not be run in a given setup. Two main information repositories, in addition to the shared memory segment, are also important components of the system: the Packet Hashes Database and the Connections Manager.

4.2 System Components

The major components of the system can be divided in two categories: processes/threads, which execute some task, and information repositories, which keep data that needs to be shared between threads.

The first category includes the following threads in the synchronization layer:

Main Thread: Starts and ends replica execution, launches the intrusion detection system, and monitors incoming alerts.

Curses Thread: Interactively presents the execution progress of a replica.

Accept Thread: Accepts incoming connections from peers and launches answer threads.

Answer Thread: Processes the requests made by the peer connected to the thread.

Sync Thread: Keeps replicas synchronized by exchanging cluster hashes, hashes of packets and packets, with peer replicas.

Pcap Thread: Captures packets from the network and enqueues them in shared memory.

Restart Thread: Restarts the IDS when a roll back needs to be performed.

Alert Thread: Receives alerts generated by the IDS and filters out duplicates.

The second category includes the following repositories:

Packet Hashes Database: Stores the computed hashes of captured packets, as well as other important data, such as packet order information and the location of the actual packet data in the shared memory segment.

Curses Thread: Keeps track of peers currently present in the system, as well as of the channels which are open between the local replica and the other peers.

The other major component of the system is of course the analysis layer, composed by the Snort intrusion detection system, which runs in a separate process from the synchronization layer. Next, the role of each of the components belonging to the synchronization layer is described in detail. Later in the chapter, the modifications introduced in the intrusion detection system, as well as the issue of checkpointing, are discussed.

4.2.1 Main Thread

The Main Thread is the thread responsible for starting and ending replica execution. When a replica is created, the thread processes command line arguments, launches the intrusion detection system process, creates the shared memory segment and sets up its initial values, which are written in a set of known addresses of the segment so that the intrusion detection system process can find them, initializes shared semaphores (the process shared semaphores and those shared between threads inside the synchronization layer process), sets up signal handling, opens log files and/or the system log facility, daemonizes the process if requested, initializes the Packet Hashes Database and the Connections Manager, opens the interface specified for packet capture and launches all the required threads, with the exception of the synchronization thread and also of the packet capture thread, which is created by the Sync Thread after the system bootstraps. Once this phase is completed, the next task of the Main Thread differs according to whether the replica is configured to be the leader replica or not.

If the replica was launched as the leader, it waits until a specified number of peers connect to it, or a specified number of seconds have elapsed. Once one of the two has occurred, the interface for packet capture is closed and reopened. This ensures that the leader has no packets which the other peers participating in the system bootstrap do not have, since it was the last one to open the interface for packet capture. The purpose of opening the capture interface at launch, in this case, is only to test whether or not the specified interface can be used, instead of doing so when other peers have already connected to it.

If the replica is not the leader, it tries to connect to the leader replica and in case of success, retrieves the list of peers currently logged into the system, and proceeds by connecting to all other peers. Logging into a peer opens a channel for requests to that peer. Upon acceptance of a connection, a peer must open a reverse connection to the peer it accepted, so it in turn can make requests to that peer.

Next, each replica proceeds by creating the Sync Thread, which will be responsible for bootstrapping and for the execution of the perpetual synchronization loop.

Finally, the Main Thread enters a loop where it sleeps for a specified amount of time and periodically wakes up to check for existing alerts or an exit signal. If an exit signal exists, it performs the required tasks for an orderly exit of both the synchronization process and of the intrusion detection system process.

4.2.2 Curses Thread

The Curses Thread is a thread which is not meant to be used in a production system. Its purpose is to allow a user to interactively follow execution progress on a console screen. For this it uses the `curses` library for terminal control, from which it inherits the name.

The user is presented with various runtime information, such as the current synchronization round and the total number of rounds to be executed, if configured, various packet enqueueing totals and rates (from packet capture, from peers, and total), statistics of packets captured and dropped by the packet capture library, and a number of desynchronization statistics.

Information concerning the amount of randomly dropped packets is also presented. Each replica may be configured to randomly discard certain packets for testing purposes. This feature will be further explained later in the chapter.

4.2.3 Accept Thread

The Accept Thread is a thread which waits for incoming connections and registers those connections next to the Connections Manager. If connection slots are available and the registration succeeds, the Connections Manager then creates an Answer Thread which will serve the connecting peer. The Accept Thread then proceeds with its wait for a new incoming connection.

4.2.4 Answer Thread

If the Connections Manager is able to register a new connection, an Answer Thread is created to process requests made by the corresponding client peer. As previously mentioned, a reverse channel is also opened if the incoming peer successfully logs in, and only if both logins succeed is a connection between peers considered to be firmly established.

The Answer Thread will serve a number of commands sent by the client peer. In the current implementation, all commands and answers to commands are sent in plain text, except for raw data of resynchronized packets.

Packet hashes are encoded and decoded using the `openssl` (`http://www.openssl.org/`) library, which has functions for the encoding and decoding of data in Base64 format, commonly used for encoding e-mail messages. Converting hashes to a plain text format, as they are sent to a peer, means they arrive in a format suitable to be used when running the `diff` utility.

The list of commands which are currently implemented is presented in table 1:

Table 1 – List of commands supported by the server thread of a replica.

`login`	The client attempts to perform a login providing its replica ID and a login secret, while piggybacking its own server IP address and port information, for the opening of the reverse channel. If the login is successful, the server replies providing two running parameters, namely the anchor size and the packet cluster size.
`peerlist`	The client uses this command to retrieve the list of peers currently connected to the server replica.

Table 1 (continued) – List of commands supported by the server thread of a replica.

Boothash	When the client is the leader, it uses the **boothash** command to send the list of hashes which correspond to the packets that are used for the initial synchronization.
boothashfatal	When the client is the leader, it uses the **boothashfatal** command if due to an internal error the **boothash** cannot be sent. In this case, bootstrapping the system fails.
chash	This command is used by the client peer to retrieve a given cluster hash of hashes.
getclhashes	If the client needs to retrieve all individual packet hashes of a given cluster, it invokes **getclhashes** on the server peer.
getpacket	This command is used to retrieve a given packet (libpcap header and raw packet data) that the client replica has missed and needs to resynchronize.

4.2.5 Sync Thread

The purpose and workings of the synchronization thread have for the most part been explained on the previous chapter. Once created, the Sync Thread of the leader replica starts by capturing *k* packets whose individual hashes together form the “boot cluster hash”. It then sends those hashes to the other peers participating in the system bootstrap.

The peers receiving the boot cluster hash search in their Packet Hashes Database for a group of consecutive packets whose hashes match the received boot cluster hash, starting from the last captured packet, instead of the first, as they may have already captured a large number of packets while waiting for the leader to initiate the bootstrap process. If a match is found, they are considered aligned with the leader, and the first packet of the boot cluster hash is registered in a global variable (for the leader, this packet always corresponds to the first captured packet, after closing and reopening the capture interface).

The Sync Thread of replicas other than the leader, once created, starts collecting network packets one by one, after which it checks if the boot cluster hash has already been found. Once found, it then completes this phase by registering in shared memory the address of the packet where the IDS should start analyzing packets. This step in not required for the leader, as in the leader’s case the IDS always starts at the beginning of the packet queue.

To complete the system bootstrap, SyncWithPeers() is called once in optimistic mode 0, so that any packets in credit are consumed. The system then switches to the configured optimistic mode, and the packet capture thread is launched by the Sync Thread, for optimistic modes above 0. In mode 0, packet capture is done by the Sync Thread itself.

Finally, the synchronization thread starts the perpetual loop which continuously calls SyncWithPeers(), as described in the previous chapter.

4.2.6 Pcap Thread

The packet capture thread is responsible for enqueueing packets in the shared memory segment. For this it calls the `pcap_dispatch()` function of `libpcap`, which in turn collects packets from the capture network interface, and calls the enqueueing function ShmEnqueue(). This latter function receives data provided by `pcap_dispatch()`, which includes a `char` pointer, the capture header and the captured packet data. All this is written in shared memory by ShmEnqueue(), together with the computed offset value of the packet to be written next in the packet queue. ShmEnqueue() is also responsible for computing the hash of the packet, which is by default a Message Digest 5 hash. After this is done, the written packet is registered in the Packet Hashes Database, by providing its packet number, shared memory address and computed MD5 hash.

Pcap Thread calls ShmEnqueue() concurrently with the function which retrieves and enqueues packets which were missed by the replica, EnqueueMissingPackets(). This function appeared already in the synchronization algorithm presented in chapter 3, and is used by Sync Thread. For this reason, one thread sometimes needs to wait for the other to finish. While the shared memory enqueueing function is the same, registering a packet in shared memory differs slightly, as in the case of resynchronized packets the position where the packet is to be inserted needs to be taken into account, while in the case of packets captured from the network capture interface, it is always inserted as the last packet in the database.

Besides calling `pcap_dispatch()`, a function which randomly drops packets, RandomDrop(), may also be called by the packet capture thread, for testing purposes. The RandomDrop() function, in its simplest usage form, drops packets according to a specified probability. For this it calls `pcap_dispatch()`, passing it the ShmNullEnqueue() function, instead of the ShmEnqueue() function, as argument. The former does nothing with the captured packet, and so the packet is lost.

RandomDrop() may also be used in more complex ways. Namely, it is possible to configure it to alternate periods in which it drops packets with periods in which it does not. Those periods may be specified in terms of seconds or packets. The start of a period where it drops packets may also be specified in terms of a probability.

As a final remark, it should be noted that the tasks performed by the packet capture thread are instead done by the EnqueuePackets() function (see the synchronization algorithm in chapter 3), if the system is running in optimistic mode 0.

4.2.7 Restart Thread

When running optimistically, checkpoints of the IDS are periodically taken. If the IDS has consumed misaligned packets, it needs to be rolled back. Both check-pointing and rolling back consumes time. In some cases rolling back requires even more time than the roll back action itself because it is necessary to wait for the previously taken checkpoint to be complete, before using it to roll back (these details will be further discussed later in the chapter).

The purpose of the Restart Thread is to minimize, as much as possible, the delay imposed to the Sync Thread, whenever a roll back needs to be performed. When a roll back is required, the Sync Thread will instruct the incorrect instance of the intrusion detection system to exit, but then leaves to the Restart Thread the task of restoring a correct instance, leaving the synchronization thread free to continue its work.

4.2.8 Alert Thread

The Alert Thread's purpose is to receive alerts generated by a customized output plug-in of the Snort intrusion detection system, and to filter out the duplicate alerts which are generated by the system. As noticed in the beginning of the chapter, the alert infrastructure implementation is currently incomplete.

4.2.9 Packet Hashes Database

Each packet which is enqueued in the shared memory segment needs to be registered in the Packet Hashes Database, including both packets captured from the network and packets that were missed, and later recovered from other peers. The database keeps not only the hashes themselves, but also the order of packets in the queue and also their address in the shared memory segment.

While the intrusion detection system does not require the services of the Packet Hash Database, as it simply follows the pointers to the next packets in the queue, the synchronization threads, and in particular the Sync Thread, the Pcap Thread and the Answer Threads, use its services extensively. To access those services, the threads use the PacketHash() interface.

The following operations are supported by the database:

Table 2 – List of operations supported by the Packet Hash Database.

HASH_INIT	Packet Hash Database initialization.
HASH_REGISTER	Registers a packet captured from the network capture interface.
HASH_MISSED_REG	Registers a missed packet, recovered from a peer replica.
HASH_CL_FIND	Used to find the boot cluster hash, sent by the leader replica.

Table 2 (continued) – List of operations supported by the Packet Hash Database.

HASH_CL_GET	Returns the cluster hash of the requested cluster number.
HASH_GET_HASHES	Retrieves a set of hashes from the database, for example the boot cluster hash, or all the hashes belonging to a cluster of hashes.
HASH_GET_OPCKT	Returns the shared memory location of a packet with a given originally registered packet number (the original reference is kept, as its reference may change due to reordering, before a peer is able to retrieve it, during a synchronization round).
HASH_GET_PCKT	Returns the shared memory location of a packet with a given packet number.
HASH_POST_SYNC	Performs post-synchronization tasks.
HASH_FREE	Frees unnecessary packet entries from the database.
HASH_DUMP_ALL	Dumps all packet entries onto the screen (for debugging).

In order to ensure integrity of the database and correctness of replies to requests, each request is processed in isolation. For this reason, there are concurrency issues in accessing the database, and it can become a bottleneck. For example, the packet capture thread and the synchronization thread concur in access to the database, as one tries to register new packets captured from the network, while the other tries to retrieve packet data required for testing alignment.

Concurrency becomes even more critical if replicas discover that some packets were missed and need to retrieve packets from each other. In this case, three threads will concur in usage of the database, as EnqueuePackets() used by Pcap Thread and EnqueueMissingPackets() used by Sync Thread both try to register packets, while the Answer Thead attempts to retrieve information required for serving missed packets requests coming from other peers. This situation may provide conditions for further loss of packets from the network, even if there are sufficient processing and memory resources available for the packet capture thread.

4.2.10 Connections Manager

The Connections Manager keeps track of the peer replicas connected to the local replica, as well as of the communication channels opened between them and the local replica. It has also the task of managing the launch and closure of the Answer Threads.

This information repository, accessed by using the ConnectionManager() interface, is used mostly when the local replica needs to execute a request to the other peers, and uses it to retrieve the list of existing peers and respective communication channels (i.e. their respective socket descriptors).

4.3 Snort IDS Modifications

For integration with the synchronization layer, a number of modifications were introduced to the original Snort intrusion detection system source code. The most significant modification was the replacement of the perpetual packet capture loop provided by the InterfaceThread() function, so that the IDS could read packets to be analyzed from the shared memory segment, instead of from LibPcap. Modifications were introduced in such a way that the modified version can be run either coupled with the synchronization layer or as a standalone intrusion detection system, as if it had no modifications.

The new perpetual loop function will start by registering the loop start time, similarly to InterfaceThread(), as required for Snort to report statistics, and then by testing if the shared memory segment is open, otherwise it will result in a fatal error. If the shared memory segment is available, it then reads necessary configuration values from shared memory known addresses, such as the synchronization ("feeder") process ID, the start and end of the packet queue and the operation mode. It then computes the number of available memory slots and initializes the checkpointing library, if necessary. After this initialization phase, it waits for the first incoming packet, after which it registers the actual packet processing start time, before entering the perpetual packet analysis loop.

There are actually two loop versions, a simpler one for pessimistic modes and another for the implemented optimistic mode. In the pessimistic version, for each packet, the loop only passes a critical section concerning exclusive access to the shared memory address where the signal checking flag is registered. After checking if there is some signal that needs to be handled, it simply calculates the new packet data location in shared memory and passes the necessary values to Snort's PcapProcessPacket() function. No exclusive access is required for reading packet data in the shared memory segment as for pessimistic modes, if the synchronization thread allows the packet to be consumed, this means there will be no further modifications to this memory area, and so there is no concurrency between the two processes in access to this area. Note that, for reporting memory freed by the IDS, additional critical sections could be required.

The optimistic version of the loop requires additional features, as in addition to signal checking and packet reading, the loop is required to report packets consumed since the last rollback, and needs to verify it a checkpoint was reached, and if so, to take the checkpoint. Both of these require exclusive access to some areas of the shared memory segment, and so it now does reading the next packet address location, as the synchronization layer may be in the process of reordering packet location pointers. Once the next packet location is established, it is safe to let PcapProcessPacket() read it without exclusive access however, as the packet data itself will not change, even if it is later discovered that packets were misaligned and that packet should not have been processed.

4.4 Checkpointing

In optimistic modes of operation, the state of the intrusion detection system needs to be saved whenever a correct synchronization point is established, so that it can be rolled back to that state, if it is discovered that the IDS has consumed misaligned packets. The initial implementation used the MTCP standalone component of the Distributed MultiThreaded CheckPointing project[12], which saves a process image to a file that may be used to restart process execution from the taken checkpoint. This image provides the means not only for rolling back an incorrectly aligned intrusion detection process, but also for another task, which would be required in a complete system implementation, even in pessimistic modes: to transfer the state to a replica newly added to an already running set of replicas. The alternative method would be to transfer all packets already processed by the running replicas, but most of those packets would probably no longer be available in replica’s buffers.

Using MTCP is highly performance penalizing, however, as performing a checkpoint takes a significant amount of time. Even if MTCP uses the `fork()` system call to spawn an image to be saved, and even if it would save each checkpoint to a different file, avoiding having to wait for the previous checkpoint to be taken before performing the next, a rollback would still require waiting for the completion of the previous checkpoint.

For this reason, and since during normal operation the checkpoint file image is not required, the intrusion detection system instead uses the `fork()` system call directly, to spawn a child process that is suspended from execution, and may later be restarted, if it is found that its parent is incorrect. Checkpointing using `fork()` provides a much faster way of creating a process copy image, as the operating system implements the system call using copy-on-write, which means it is not even necessary to wait for copying the process image in memory.

Some modifications to MTCP were also introduced, for better control of the timing of checkpoints (out of the box, MTCP will only take checkpoints periodically). An MTCP image would still be needed if it was required to transfer intrusion detection system state between replicas, and it was verified that a given image could in fact be restarted on another replica in a different machine. This requires that replicas write a file with its ID in a known location, so that when resumed, the IDS image can check if it is still on the same feeder replica, otherwise it needs to adjust shared memory segment address values (known locations are registered as offsets of the start of the memory segment, but the starting address will change across different computers).

Mechanisms for actually transferring the state were never implemented, however, so testing was done by manually transferring and restarting the image file.

[12] `http://dmtcp.sourceforge.net/`

5 RESULTS AND DISCUSSION

In this chapter, an evaluation of the system effectiveness and efficiency is carried out. First, an evaluation is made to verify the correctness of the synchronization algorithm and how, not considering performance limitations, it is in theory able to improve on what a single replica would be able to achieve, in scenarios where either only one or all replicas lose packets. Next, performance tests are carried out to verify system limits and in which conditions the replica system would be able to provide better results than a single unmodified IDS system.

5.1 Setup

The setup of the test environment involves using a combination of hardware for the system replicas and for traffic injection, specific software versions and testing conditions.

5.1.1 Replica Hardware

Two sets of different hardware for the replica intrusion detection systems were used. The first set uses two desktop-class computers, each with the characteristics outlined in table 2:

Table 2 – Summary of desktop computer replicas hardware.

Processor	Single Dual Core Processor at 2.33GHz (Intel® Core™ 2 Duo E6550)
Memory	2 GB RAM
Hard Disk	Single SATA HDD at 7200 RPM
Network	Two Gigabit Ethernet network interface cards, one for traffic capture, one for replica synchronization and management

A deployment using two desktop computers would be a common scenario in many organizations, where a combination of limited budget and available network bandwidth would mean it would be difficult to justify the acquisition of dedicated high-end hardware for the task of network intrusion detection.

The second set of hardware for replica intrusion detection systems uses two server-class computers, and represents a setup that would be deployed in organizations where the cost of such hardware could be justified. Its characteristics are presented in table 3:

Table 3 – Summary of server computer replicas hardware.

Processor	Dual Quad Core Processor at 2.50GHz (Intel® Xeon® L5420)
Memory	8 GB RAM (due to kernel limitations, only 3.2GB are available)
Hard Disk	Dual SAS HDD at 15000 RPM (RAID 1)
Network	Three Gigabit Ethernet network interface cards, one for traffic capture, one for replica synchronization and one for management

5.1.2 Traffic Source

For organizations wanting to monitor network links with bandwidth up to 100 MBit/s, a simple setup for feeding the traffic source to replicas could consist of configuring an Ethernet switch with a network port which that would mirror the traffic of the link or links to be monitored. This port could then be connected to a network repeater hub, which would then replicate traffic to all the replicas in the cluster.

For higher bandwidth requirements, such as Gigabit Ethernet links, commercial hubs seem to be rare to nonexistent, however. While commercial hardware tailored for this type of task exists, since tests performed were based on the repeated injection of previously captured traffic, for test repeatability, instead of analysis of live traffic, a cheaper solution consisting of using only the injector computer and the replicas was used.

The injector hardware is comprised of a low-end server-class computer, with the characteristics described in table 4:

Table 4 – Summary of traffic injector computer hardware.

Processor	Single Dual Core Processor at 2.33GHz (Intel® Xeon® 3065)
Memory	4 GB RAM
Hard Disk	Dual SATA HDD at 7200 RPM (software RAID 1)
Network	Three Gigabit Ethernet network interface cards, two for traffic injection, one for management

Two interfaces are used for traffic injection, each of which is directly connected to the traffic capture interface of an IDS replica. These interfaces are bonded together using the Linux kernel bonding driver, in broadcast mode. This mode sets up a virtual bond interface which transmits all traffic outgoing from that interface on all slave interfaces [5].

5.1.3 Software Versions

Replicas were installed with the current stable 32 bit version of the Debian GNU/Linux distribution, Debian *Squeeze*. However, due to noticing degraded performance in the included LibPcap software, this library's shared object and header files were sourced from the previous version, Debian *Etch*. Software versions for the Snort IDS, checkpoint library and diff library were sourced from their respective web sites.

Regarding the diff utility, the project implementation initially used the widely used GNU `diff` command, included in the Debian distribution, calling it as an external command. To avoid the performance penalty incurred in the frequent execution of an external command, it was replaced by Michael B. Allen's `libmba` library implementation. Tests confirmed the two versions of the synchronization layer produced the same ordering results (two replicas, each using a different version of the utility, would be able to run compatibly), and usage of the external command was therefore abandoned.

Table 5 summarizes the software versions used in the project:

Table 5 – Used software versions.

Operating System	Debian GNU/Linux 6.0.3 *Squeeze*, 32 bit version, with kernel 2.6.32-5-686 (except for LibPcap library, sourced from Debian 5) Source: `http://www.debian.org/`
Intrusion Detection System	Snort, version 2.8.5 (Build 106) Source: `http://www.snort.org/`
Check-pointing Library	Distributed MultiThreaded CheckPointing, MTCP standalone component, version 1.2.3 Source: `http://dmtcp.sourceforge.net/`
Diff Utility Library	libmba, version 0.9.1 Source: `http://www.ioplex.com/~miallen/libmba/`

5.1.4 Testing Conditions

Tests were run with only a minimum set of operating system related processes running on each replica, such as the `syslog`, `udev` and `dbus` daemons. The notable exception was the `ssh` daemon, used for interactive connection to replicas. This resembles the conditions of dedicated NIDS production boxes, which should ideally have only a minimum set of processes running, besides the intrusion detection system itself, but which usually provide some form of remote management interface.

For the same reason, the network interface for capturing traffic was setup as a dedicated traffic capture interface, with no associated IP address. This means traffic originating from the IDS should never be sent from that interface, which is a key requirement for the stealthiness of a network intrusion detection system.

No attempt was made to customize the default Snort configuration, so tests were run with out-of-the-box configuration and rules provided by the used Linux distribution. The only exception to this was tuning the home network parameter, so it would match the network to which packet samples belonged to.

5.1.5 Tools for Network Traffic Capture and Replay

For capturing traffic samples used during the tests, the widely used `tcpdump`[13] tool was chosen. Many tools are able to replay traffic captured by `tcpdump`. In this project, the `tcpreplay`[14] tool was used.

[13] `http://www.tcpdump.org/`

5.2 Running Options

The synchronization layer software can be executed with a combination of different options. Typing the command ‘`sink --help`’ provides a brief help and summary of available options:

```
# sink --help
Usage: sink [OPTIONS]
  or:  sink --version
  or:  sink --help

Notes:
  Optimistic mode has four levels, using checkpointing in the last two:
    * 0: very pessimistic - capture n packets, then synchronize with peers
    * 1: pessimistic - continuously capture packets in a separate thread
    * 2: optimistic - feed Snort as soon as current cluster packets are captured
    * 3: very optimistic - feed Snort as soon as packets are captured (TODO)

  Random drop option format can be:
    1. <probability>
    2. p<probability>,(d|D)<duration>,(f<frequency>|(i|I)<interval>)

    Values mean:
      * <probability>: probability that a packet is dropped (1-100000)
      * <duration>:    duration of a random drop period (d: secs; D: pkts)
      * <frequency>:   probability that a random drop period starts (1-100000)
      * <interval>:    interval between random drop periods (i: secs; I: pkts)


Options:
  -a <id>               Sink ID
  -A                    Only synchronization layer (don't run Snort)
  -B                    Buffer size in packets, for pcap (0 to disable, 32767 max)
  -C <interface>        Interface for packet capture (max: 31)
  -c                    Curses mode
  -d                    Daemon mode
  -D                    Dump packet hash database in the end (don't free hash slots)
  -e <snort path>       Snort executable file path
  -E <snort conf>       Snort configuration file path
  -f                    Shared memory file path (default: /tmp/apart_segment_id)
  -F                    MT CheckPoint file path (default: /tmp/apart_snort_mtcp)
  -g <group name>       Run Snort using this group name GID
  -G                    Use Snort grandchildren, not MTCP, for checkpoint images
  -h <hashes file>      Log packet hashes to hashes file (replace)
  -H <hashes file>      Log packet hashes to hashes file (append)
  -i <interface>        Listen only to this interface
  -l <log file>         Log messages to log file
  -L                    Log messages to the system log
  -m <max clients>      Maximum number of connections (1-128)
  -M <memory size>      Shared memory size in MB. (Min: 2, default: 32)
  -n <n packets>        Use n packets for anchor  size (only on leader; default: 4)
  -N <n packets>        Use n packets for cluster size (only on leader; default: 56)
  -o <mode>             Optimistic mode: 0, 1 or 2 (TODO: 3)
  -p <port number>      Listen to this port number
  -r <rd format>        Randomly drop packets collected from pcap_dispatch
  -R <n rounds>         Run only n synchronization rounds
  -S <address>          Leader sink address (requires -x)
  -s <port>             Leader sink port
  -q                    Quiet. Same as -v 0
  -u <user name>        Run Snort using this user name UID
  -U                    Feed UP packets even during resynchronization
  -v <level>            Verbosity level (0-10)
  -V                    Verify packets in DB and shmem (requires enough shmem)
  -W <n seconds>        If leader, wait for n (0-60) seconds to start
  -w <n sinks>          If leader, wait for n (0-16) sinks to start
  -x <mtcp_restart>     'mtcp_restart' executable file path
  -X <secret>           Cluster secret
  -Z <hash function>    Use 'murmurhash3', 'one-at-a-time' or 'crc32' instead of md5
  -z <n bytes>          Use only the first n packet bytes for the hash (default: 56)
```

Figure 15 – Synchronization layer executable help menu.

[14] `http://tcpreplay.synfin.net/`

5.3 Packet Samples

To test the system, two types of datasets were considered. One type consists of capturing real world traffic, while the other consists of generating artificial packet streams.

For the first type, a random sample of one million packets was collected from the Internet connection of a ~1500 student higher education institution, during a normal working period. This will be referred throughout this chapter as the *Dataset 1*.

For the second type, the tool packETH[15] was used to generate streams using a repetition of a specific UDP packet from *Dataset 1*. This tool is able to send out repetitions of a given Ethernet packet, while (optionally) randomly changing its source MAC and IP addresses. The purpose of randomization is to create a homogenous traffic stream, both in rate and in content, while introducing the necessary differences for the proper operation of the synchronization protocol.

A UDP reply to a Domain Name System query was chosen, so that Snort would be provided with a data packet that could result in some analysis work done by the IDS, unlike for example a single TCP packet, which would necessarily belong to an incomplete TCP stream, and would for that reason probably be for the most part ignored by Snort: for example, Papadogiannakis *et al.* [6] note that 4627 of the 9276 rules in the default Snort rule set contain the `flow:established` keyword, which defines that the detection engine should process the rest of the rule only if the packet belongs to an established TCP connection.

5.4 Packet Hashes and Hash Algorithms

Using the synchronization layer imposes an extra overhead over a system running only the intrusion detection system software. Not considering the synchronization mechanism itself, simply capturing the packets for feeding Snort is already costlier than direct capture by the IDS, due to three reasons:

1. Snort analyses packets stored from the buffer provided by the packet capture library. Because the synchronization layer requires additional control of the packet queue, packets have to be copied between the packet capture buffer and the synchronization layer's own packet queue, before being made available to the IDS.
2. A hash of each packet needs to be computed and stored in a hash database, as this is a key element of the synchronization algorithm.
3. There is some concurrency between the two layers, implying sometimes one has to wait for the other until access to the queue can be performed.

While the overhead of copying the packets to the shared memory queue cannot be avoided, and concurrency can be optimized only up to a point, computing and storing the hashes offers a number of chances for optimization.

[15] `http://packeth.sourceforge.net/`

Two factors influence the performance of hash computation: the chosen algorithm and the amount of data provided as input. Given the purpose served by the hashes, which is to provide a compact but ideally unique representation of each packet, several algorithms were tested for hash collisions and performance, using different maximum packet data as input.

5.4.1 Collisions

As input, the one million packets *Dataset 1* was used. This dataset was found to have 997864 unique packets, meaning a total of 2136 packets were equal to some other packet. This is not considered to be unexpected, as for example any repeated DNS query or ICMP sequence between a given pair of hosts could result in exactly similar packets, not to mention TCP packet retransmissions due to packet loss in a TCP connection.

Four different hash/checksum algorithms were tested. Results are shown in table 6:

Table 6 – Analysis of hash algorithms collision performance, using different maximum lengths of data from each packet, starting at the beginning of each one. The tested dataset has a total number of 1 million packets, 2136 of which are equal to a previously seen packet.

	16 Byte Hashes		**4 Byte Hashes**	
Bytes	**MD5**	**MurmurHash3**	**Jenkins One-at-a-Time**	**CRC32**
16	999924	999924	999924	999924
32	29796	29796	29920	29864
40	2758	2758	2866	2879
48	2169	2169	2277	2288
56	2136	2136	2266	2252
64	2136	2136	2250	2262
Full	2136	2136	2244	2270

As expected, 16 byte hash functions showed a lower collision rate than the 8 byte ones. On the other hand, results show that, for the tested dataset, using the first 56 bytes of each packet to compute the corresponding hash was enough to achieve a collision rate equal to that of the collision rate already present in the that dataset.

5.4.2 Performance

To test the differences in performance between algorithms, various trials were done for each algorithm, varying the maximum length of packet data used for calculation of its hash. The system was run in optimistic mode 1 with a large shared memory buffer and without Snort. Packets from the *Dataset 1* were injected by the `tcpreplay` tool at maximum speed (between 905 and 910 Mbps, as reported by the tool), using the `‘—loop=10’` option, and the formation of a queue of outstanding packets (that is, of packets in shared memory, not yet synchronized) and/or the dropping of packets from the LibPcap queue were observed.

The replicas were run with the following options:

```
   # sink -a <id 1> -C <interface> -i <own IP> -p <own port> -X <secret> -M 1000 -o 1
          -A -c [-z <hash function>] -z <size> -w 1
```

and,

```
   # sink -a <id 2> -C <interface> -i <own IP> -p <own port> -X <secret> -M 1000 -o 1
          -A -c [-z <hash function>] -z <size> -S <leader IP> -s <leader port>
```

Results, using the server-class hardware, showed that a queue of outstanding packets and/or the dropping of packets would start to build at a value of about 1380 bytes as maximum packet length used for input in MD5; for MurmurHash3, at about 745 bytes; for Jenkins One-at-a-Time, at about 400 bytes; and for CRC32, at about 780 bytes.

5.4.3 Algorithm Selection

In light of the collision and performance results, all subsequent results presented in the next sections were performed using the MD5 algorithm, providing it with only up to the first 56 bytes of each packet as input for the hash calculation. These options would most likely become non-tunable parameters on a production implementation.

5.5 Implementation Correctness

If the synchronization algorithm and its implementation work according to the intended goals, then it is expected that, for all packets up to the last completed synchronization round:

1. If at least one replica has not lost packets, then all replicas should present the complete set of packets, in the correct order, to the upper layer;
2. All replicas should present the same set of packets, and in the same order, regardless of the degree of packet loss that has occurred on each individual replica.

To test the implementation correctness, the replica system was tested by introducing different levels of random packet loss, using the RandomDrop() function. One-million packets *Dataset 1* was injected in the network, followed by a set of 10 thousand repeated UDP packets, generated without address randomization, and the resulting hashes of each replica were saved into files.

Hashes resulting from the UDP packets were then removed from the files and a count of different lines using the `diff` utility to compare the saved files was produced. The purpose of injecting the additional UDP packets is to provide a chance for the protocol to align the trailing packets of *Dataset 1*. It was verified that the hash of these packets does not occur in the hashes of packets which are present in *Dataset 1*.

The first test consisted of running the system using the RandomDrop() function on only one of two replicas, for various hash cluster sizes and target packet loss values.

The replicas were run with the following options:

```
# sink -a <id 1> -C <interface> -i <own IP> -p <own port> -X <secret> -M 1000 -o 1
       -A -w 1 –h <file 1> [-N <cluster>]
```

and,

```
# sink -a <id 2> -C <interface> -i <own IP> -p <own port> -X <secret> -M 1000 -o 1
       -A -S <leader IP> -s <leader port> –h <file 2> [-N <cluster>] [-r <loss>]
```

Figure 12 shows the results:

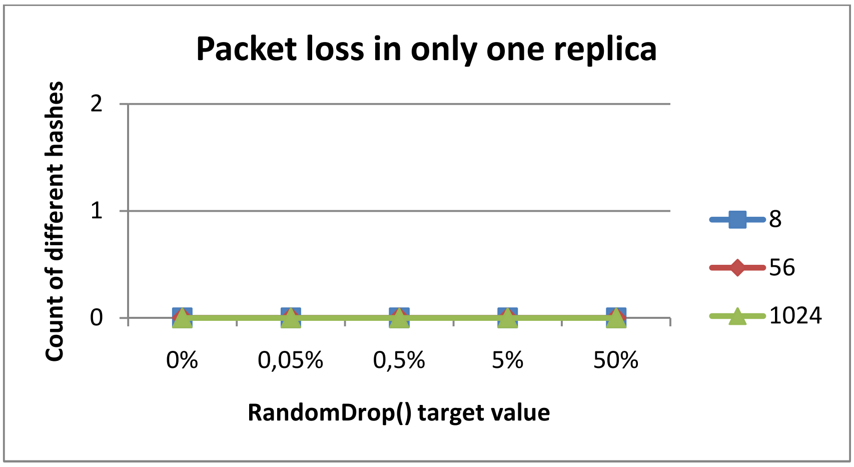

Figure 16 – Count of differences between hash files from two replicas, one which does not drop packets and one which drops packets, using the RandromDrop() function, after stripping hashes resulting from the crafted UDP packets sample. The resulting files are always similar. Tested hash cluster sizes were 8, 56 and 1024.

As shown, the hash file produced by the packet-loosing replica, after UDP-crafted packets are stripped, is always similar to the file of the replica to which packet loss was not imposed. Also, as expected, comparing each of these files with the hashes file produced from a single, non-clustered replica to which packet loss was not imposed, equally results in no differences between files, which seems to confirm that if at least one replica has not lost packets, then both replicas are able to order and deliver the original set of packets to the upper layer.

In the second test, the system was run using the RandomDrop() function on both replicas, with equal target packet loss values on each run, for various hash cluster sizes:

The replicas were run with the following options:

```
# sink -a <id 1> -C <interface> -i <own IP> -p <own port> -X <secret> -M 1000 -o 1
       -A -w 1 –h <file 1>  [-N <cluster>] -r <loss>
```

and,

```
# sink -a <id 2> -C <interface> -i <own IP> -p <own port> -X <secret> -M 1000 -o 1
       -A -S <leader IP> -s <leader port> –h <file 2> [-N <cluster>] -r <loss>
```

Figure 13 shows the results:

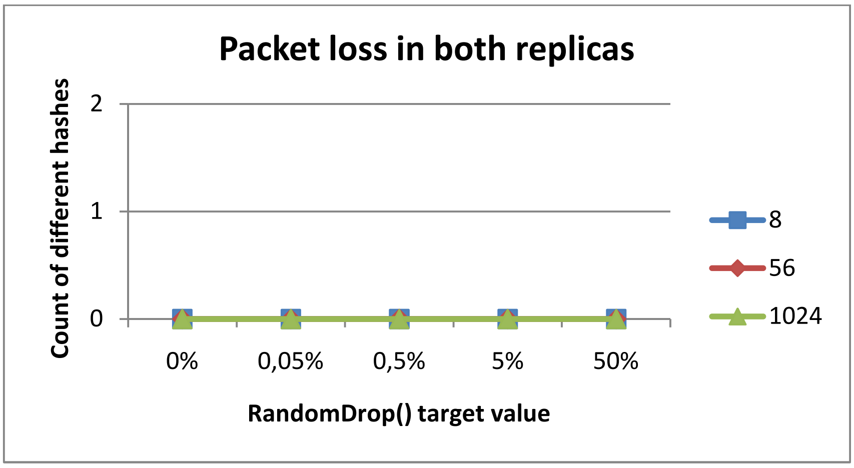

Figure 17 – Count of differences between hash files from two replicas, both dropping packets using the RandromDrop() function, after stripping hashes resulting from the crafted UDP packets sample. The random drop target parameter was the same for both replicas, on each run. The files resulting from each are always similar (but different from that of a replica where packets were not lost).

As shown, the hash files produced by both packet-loosing replicas, after UDP-crafted packets are stripped, are similar, which seems to confirm that both replicas are able to align and present the same set of packets to the upper layer. It should be noted that, while both replicas use the same packet drop target value, this does not mean they lose the same set of packets, due to the random character of the packet-dropping function. During these tests, a given packet could have been missed by one, the other, neither or both replicas.

5.6 Synchronization Protocol Effectiveness

Given that the goal of the synchronization protocol is to ensure that the replica cluster provides the intrusion detection system with a set of packets as close as possible to the original set, and in particular by providing to all replicas each packet that was received by at least one replica, the effectiveness of the protocol can be evaluated by its ability to avoid packet loss, which can be estimated theoretically, if considering some simplifications:

1. Considering that each replica loses packets independently from the others, not taking into account the factors which led to the occurrence of packet loss;
2. Considering that, as long as the upper layer receives a given packet, it is not relevant for its correct operation whether or not it was received in the same order as it was originally seen on the network.

If these simplifications can be taken as true, then the theoretical effectiveness of the protocol can be estimated from calculating the probability that a given packet is lost by all the replicas:

$$P_{(Packet\ loss)} = P_{(R_1\ packet\ loss)} \times P_{(R_2\ packet\ loss)} \times \ \dots \times P_{(R_n\ packet\ loss)}$$

And, if all replicas have an equal packet loss probability then, for n replicas:

$$P_{(Packet\ loss)} = P_{(Replica\ packet\ loss)}{}^{n}$$

Figure 14 shows the theoretical synchronization protocol effectiveness for a one-million packet sample, using different replica cluster sizes in which each replica would have an equal packet loss probability:

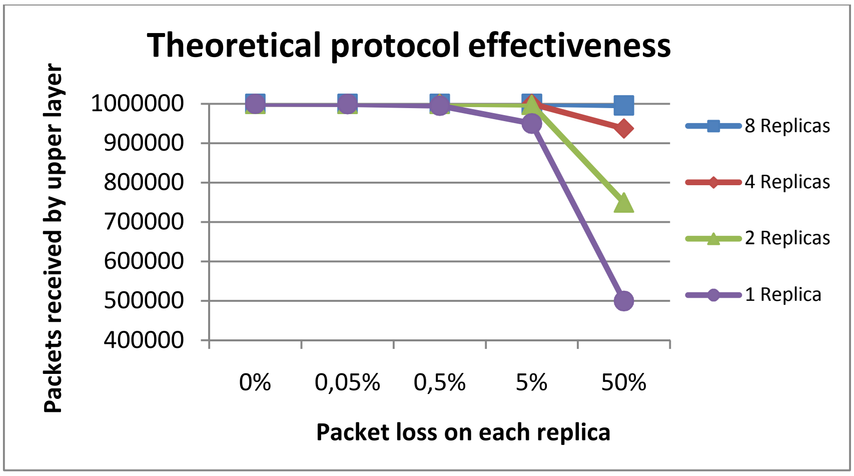

Figure 18 – Theoretical effectiveness of the synchronization protocol, considering that replicas lose packets with an independent probability and that all have an equal packet loss probability.

The figure illustrates how, even for high packet loss probabilities, with a high number of replicas, the final packet loss by the cluster would be relatively low. For a 50% packet loss probability on an individual replica, a cluster with four replicas would lose 6,25% of all packets, and 0,39% if it had eight. Even a two replica system would still lose only half of the traffic lost by a single replica system.

To analyze the synchronization protocol effectiveness in practice, the replica system was again tested by introducing different levels of random packet loss, using the RandomDrop() function. One-million packets *Dataset 1* was injected in the network, followed by a set of 10 thousand UDP packets, generated without address randomization, and the resulting hashes of each replica were saved into files. Hashes resulting from the UDP packets were again removed from the files and a count of different lines using the `diff` utility to compare the saved files was produced.

It should be noted, however, that the RandomDrop() function is not very accurate: the number of actually dropped packets can be significantly different from the specified packet loss target. It does, however, accurately report the actual number of dropped packets in the end of each run, and it is fairly consistent when run repeatedly under the same conditions. Therefore, before testing the system for effectiveness, conditions close to the desired packet loss target were found for *Dataset 1* (without using the UDP packets):

Table 7 – Determining actual packet loss rates for a given set of conditions. Results shown are three-run averages. Controllable conditions which influence the actual packet loss rate of the dataset traffic are the RandomDrop() function configuration option value and the rate of traffic injection.

Target	Replica 1	Replica 2	Average	Conditions
0,05%	0,0498%	0,0515%	0,0507%	-r 25, ~30 Mbps
0,5%	0,513%	0,514%	0,513%	-r 250, ~30 Mbps
5%	5,03%	5,06%	5,04%	-r 2500, ~30 Mbps
50%	52,9%	5,29%	52,9%	-r 50000, ~10 Mbps

The system was then tested under the determined conditions of packet loss target rates and traffic injection rate. Running options were the same as in the previous test. The results are represented in figure 15:

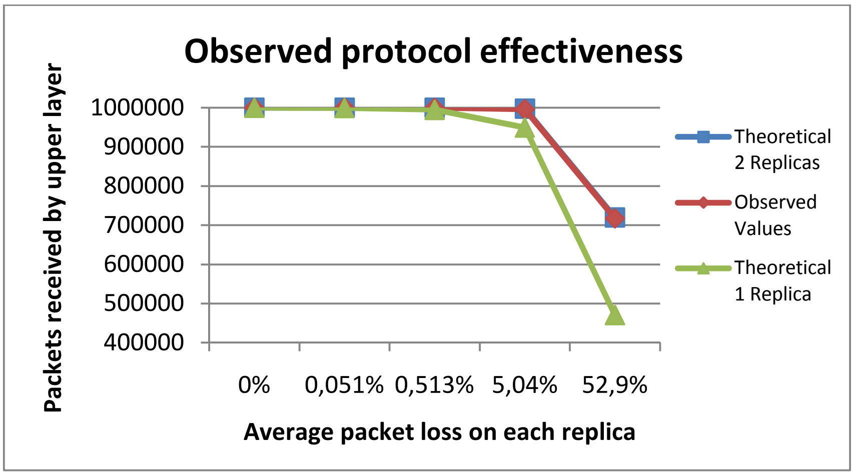

Figure 19 – Theoretical and observed synchronization protocol effectiveness. The results shown are three-run averages for each packet loss rate condition.

The results show a very close behavior between the expected and actual results. The actual numbers however, show that the real packet loss was always slightly higher than the theoretical value. This may indicate that even for these test conditions, where packet loss occurs solely due to the RandomDrop() function, loss probability may not be completely independent between replicas. It is easy to see that in a live traffic monitoring situation, with packet loss occurring due to traffic spikes, loss probability would not be independent between replicas, and would bring the loss rate closer to that of a single replica.

The number of packets received by the upper layer does not, however, highlight the fact that the synchronization algorithm, which relies on `diff` to order those packets that are not lost, will not always align them in accordance with the original order. The previous results are therefore again repeated in figure 16, with an added line representing the total differences between a run with no packet loss and runs where packet loss was imposed. These differences result from packets lost, plus out-of-order packets.

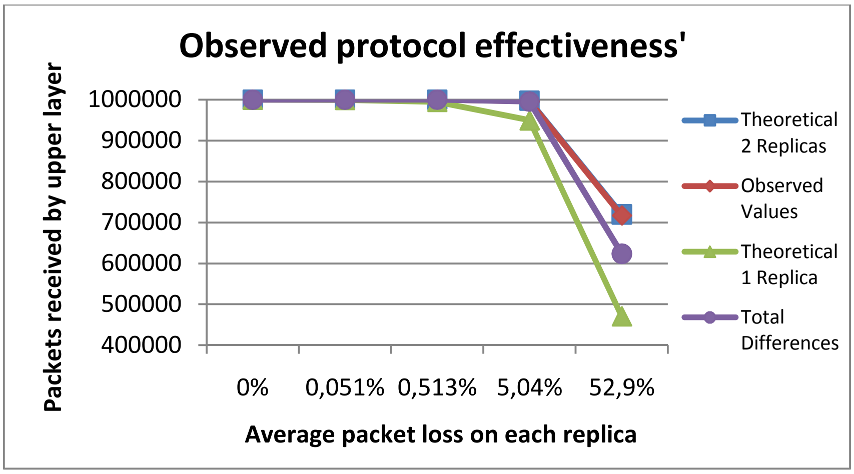

Figure 20 – Theoretical and observed synchronization protocol effectiveness, including data representing the total number of differences between hash files of synchronized replicas which have lost packets and a file from a lossless replica. The results shown are a three-run averages for each packet loss rate condition.

As it's possible to observe, the total differences line approaches the theoretical line for one replica (where differences would be due only to packet loss), but this effect is more noticeable for very high packet loss rates. For small to moderate loss rates, results suggest that `diff` has, in frequent cases, enough data to align packets according to the original order.

5.7 System Performance

The previously presented tests were conducted under conditions in which the system was run without having to deal with performance issues. Given the fact that intrusion detection systems are often required to monitor high-speed network links, the performance of the replica system is of paramount importance for its applicability. As discussed in 5.4, the synchronization layer will represent a performance overhead, when compared with running an IDS without this layer. This performance overhead may be justified by the likely existence of spare processing capacity, in today's world of multi-core processors. Snort in particular, and in its current stable version is, as previously discussed, single-treaded, leaving a full processing core available for other tasks, in a today's very common dual-core processor computer.

5.7.1 Hash Cluster Size

Hash cluster size is one parameter which is expected to influence performance. Intuitively, a smaller hash cluster size will be costlier than a larger one, in regard to the number of required cluster hash exchanges between replicas, to confirm alignment, while a larger cluster size will require a higher number of individual hashes to be exchanged, in the cases where replicas discover they are misaligned.

Performance was evaluated by calculating throughput. Three hash cluster values were tested: a "small" value of 8, a "medium" value of 56, and a "large" value of 1024 hashes per hash cluster.

With a medium value of 56 it should still be possible to exchange the individual hashes in a single IP packet, in case of desynchronization: a Base64 encoded 16 Byte hash takes 24 Bytes, so the total size for 56 hashes is 1344 Bytes. With the addition of the various protocol headers, the final packet size will be close to the typical 1500 Bytes MTU limit on Ethernet networks.

Trials were done on the server-class hardware, for various degrees of packet loss, imposed using the `RandomDrop()` function. The default value of four hashes for the anchor size was used. Throughput was calculated by measuring the time required to synchronize and provide the upper layer with a total of 501760 packets, which requires 62720 synchronization rounds for 8-hash clusters, 8960 rounds for 56-hash clusters and 490 rounds for 1024-hash clusters.

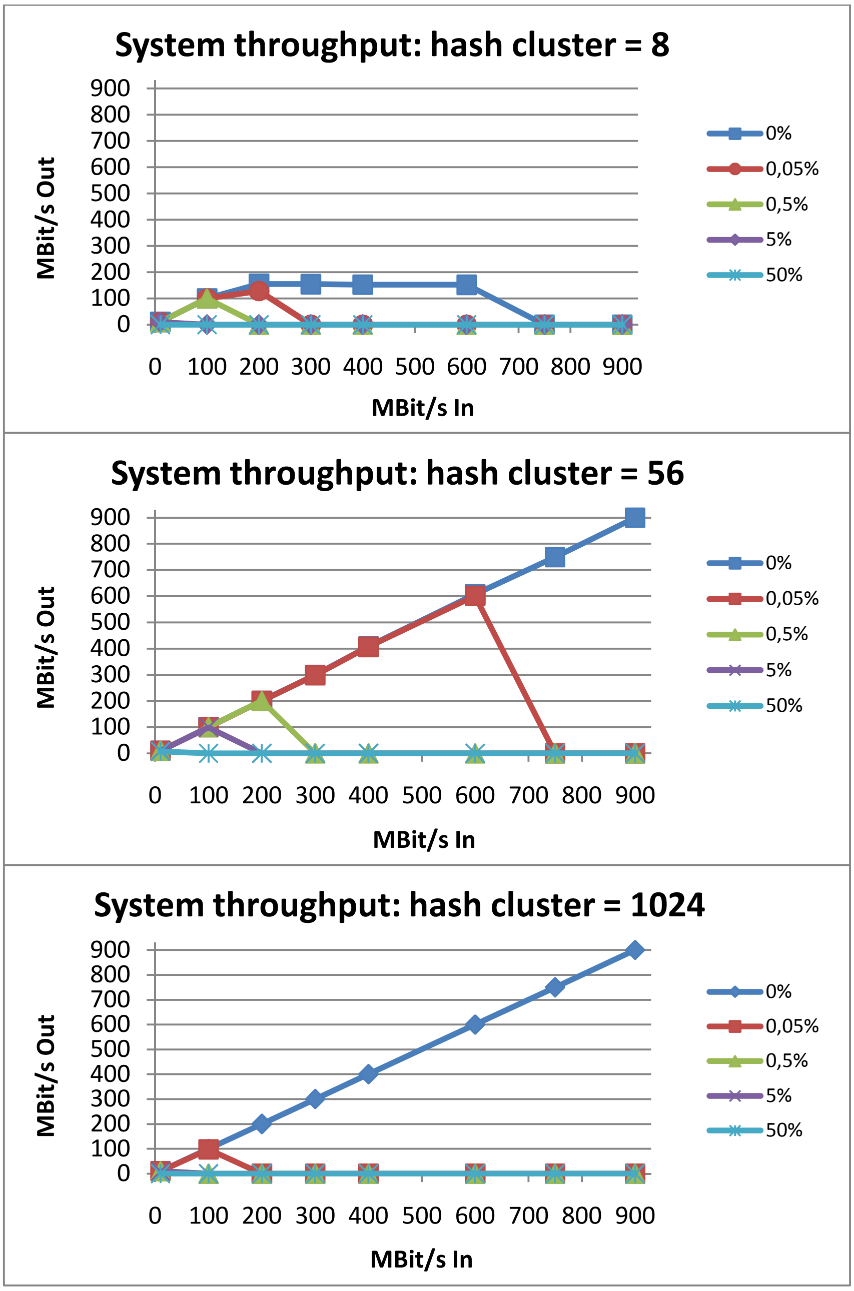

Figure 21 – Replica system throughput for hash clusters of 8, 56 and 1024 hashes. Traffic was injected at speeds of 10, 100, 200, 300, 400 and 900 MBit/s. Imposed levels of packet loss were 0%, 0,05%, 0,5%, 5% and 50%.

The replicas were run with the following options:

```
# sink -a <id 1> -C <interface> -i <own IP> -p <own port> -X <secret> -M 1000 -o 1
       -A -w 1 [-N <cluster>] –R <rounds> [-r <loss>]
```

and,

```
# sink -a <id 2> -C <interface> -i <own IP> -p <own port> -X <secret> -M 1000 -o 1
       -A -S <leader IP> -s <leader port> [-N <cluster>] –R <rounds> [-r <loss>]
```

Results in figure 17 suggest that the medium size of 56 hashes per cluster is the most favorable value, as it outperforms the smaller and larger values in all situations, and is the only cluster value able to handle a 50% packet loss rate at 10 MBit/s. While a value of 8 seems to provide better results than a value of 1024 when packet loss is involved, as was previously intuited, the overheard of frequent cluster hashes exchanges seems to limit its maximum throughput to about 150 MBit/s, before it finally fails. On the other hand, while under packet loss conditions the 1024-hash value performs worse than the 8-hash value, with it, the system is able to handle the maximum traffic rate that `tcpreplay` is able to inject, if loss does not occur. It should be noted, however, that the system is unable to bootstrap under this traffic rate with a 1024-hash cluster, unlike what happens with a 56-hash cluster. Recall that bootstrapping involves the comparison of all individual packet hashes to confirm alignment, not just the comparison of the hashes computed from the cluster of individual hashes.

Results also highlight two things: that the system is able to synchronize packets at “wire speed” until close to the point where it fails (except in the 8-hash cluster case), and that it is not able to continue operating sustainably beyond that point and re-synchronization becomes unrecoverable. This is in the nature of the protocol, and a production system would need to include mechanisms to stop attempting synchronization once it was detected it could not handle the current bandwidth.

5.7.2 Running with the IDS

In the previous section, the performance of the synchronization system running without running the intrusion detection layer was evaluated, with the purpose of selecting a good value for the size of the cluster of hashes. A complete system, running with the intrusion detection system on top of the synchronization layer, will have a smaller throughput, due to two reasons: first, due to concurrency issues between the two layers; secondly, because while without artificial packet loss the synchronization layer was able to handle “wire speed” traffic for the tested traffic rates, Snort is not; if Snort is not allowed to drop those packets which it is not able to handle at “wire speed”, it will require the intrusion detection system to use additional time to process them, thereby reducing throughput.

Figure 18 shows the average rate of packet loss, on the two classes of hardware used during the tests, under different network traffic rates, for Snort running as a stand-alone IDS (i.e., without the synchronization layer) when using the out-of-the-box configuration and rules:

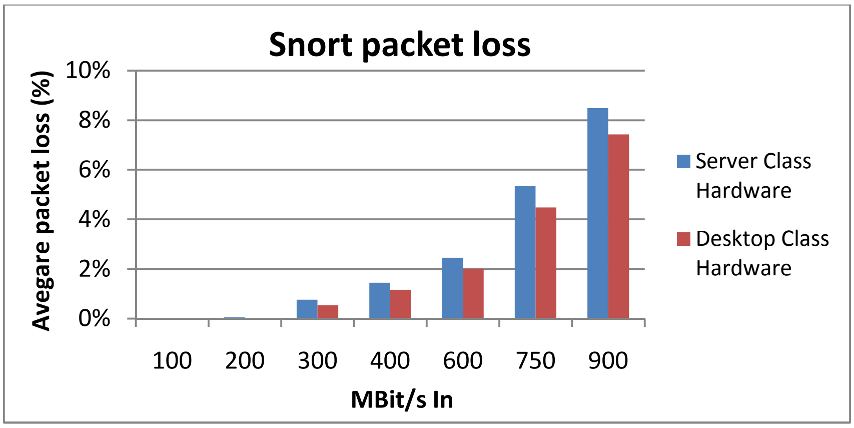

Figure 22 – Six-run averages (three runs per machine) packet loss of the Snort intrusion detection system running in stand-alone mode (without the synchronization layer) and with the out-of-the-box configuration and rules provided by the Debian distribution.

Interestingly, while the results between the two different hardware classes are relatively similar, the desktop-class hardware seems to perform slightly better than the server class one. It should be noted, however, that both sets of processors used are of a similar generation and have a very approximate CPU clock working frequency, and since Snort is not able to take advantage of the higher number of cores available, approximate results were to be expected.

Throughput was calculated as previously, but on this occasion running with the intrusion detection system on top of the synchronization layer, in both optimistic modes 1 and 2, for the default value of 56-hash clusters, using the following options:

```
# sink -a <id 1> -C <interface> -i <own IP> -p <own port> -X <secret> -M 1000 (-o
       1|-o 2 –G) –R 8960 –e <snort executable> -E <snort conf> -u <snort user> -g
       <snort group> [-r <loss>] –w 1
```

and,

```
# sink -a <id 2> -C <interface> -i <own IP> -p <own port> -X <secret> -M 1000 (-o
       1|-o 2 –G) –R 8960 –e <snort executable> -E <snort conf> -u <snort user> -g
       <snort group> [-r <loss>] -S <leader IP> -s <leader port>
```

Throughput was found to be very similar to that of the system running without the intrusion detection system, in particular in optimistic mode 1, which means that the system, complete with the IDS, was still able to perform at “wire speed”, up to the points of failure. Only on high traffic rates, from 300 MBit/s onwards, was the overhead effect noticeable. This effect was more pronounced in optimistic mode 2, whose other difference was the fact that it was not able to sustain a level of 5% packet loss at 100 MBit/s, unlike optimistic mode 1. Recall that, comparing the two optimistic modes, mode 2 is more demanding, due to the existence of added threads, the overhead of checkpointing, and the additional work of re-processing already process packets, when rollbacks occur. This latest factor, however, did not come too much into play, as at traffic injection rates higher than 300 MBit/s, only at very small levels of packet loss (0,05%) was the system able to process traffic without failing.

To highlight the differences between the replication system running with and without Snort, figure 19 compares the relative throughput of the system without the IDS and with the IDS in the two optimistic modes:

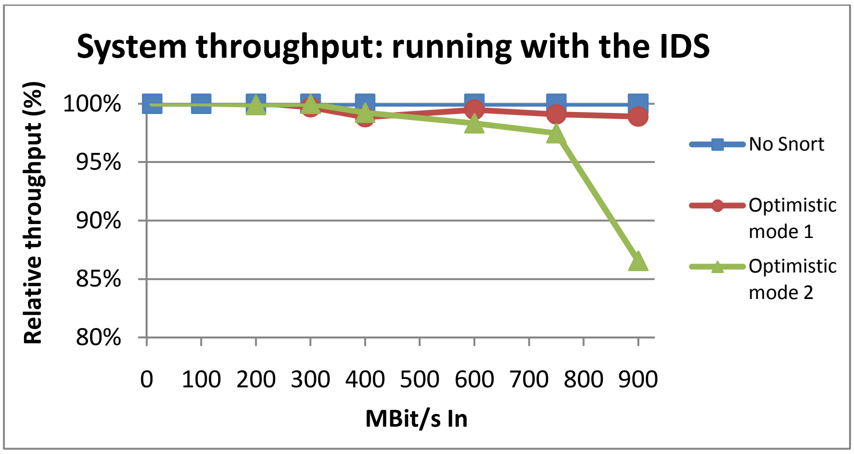

Figure 23 – Comparison of the relative throughput of running the system without the IDS (with '-o 1') and running the system complete with the IDS, in optimistic modes 1 and 2, without packet loss. Throughput is very similar at lower traffic injection rates, and differences become more apparent as this rate increases, in particular in mode 2.

Only throughput without packet loss is shown, since with packet loss it is very similar for all cases, with the exception of optimistic mode 2 at 100 MBit/s with a 5% packet loss level, as was previously mentioned. The figure illustrates how throughput suffers at higher traffic rates, in particular in mode 2.

5.7.3 Optimistic Mode 2 Checkpoints

Remembering that the purpose of the optimistic modes is to give the intrusion detection system a chance to analyze packets even if their alignment is not yet confirmed, it is important to evaluate whether or not it is ever a position to do so, or if in reality it is always too far behind the synchronization mechanism, and the overhead of the check-pointing feature is unnecessary.

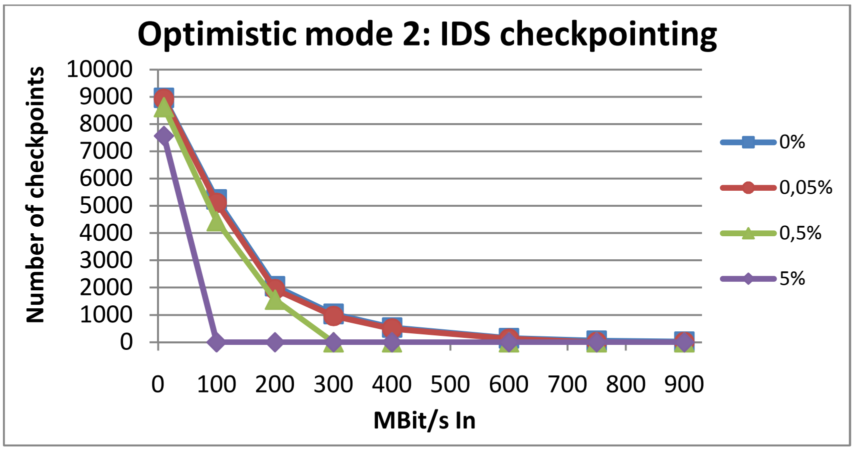

Figure 24 – Number of average checkpoints taken by the intrusion detection systems in optimistic mode 2. Checkpoints decrease as traffic rate increases.

Analysis of the number of checkpoints taken by the intrusion detection systems gives an indication that, in particular at lower network traffic rates, they are often close to the current synchronization round. Recall that in optimistic mode 2, the intrusion detection systems will execute a checkpoint if they are just ahead or behind the last synchronization point that was established by the synchronization layer.

Confirmation that the intrusion detection systems are sometimes ahead of the last synchronization point comes from analyzing the number of rollbacks that were performed, as illustrated in figure 21:

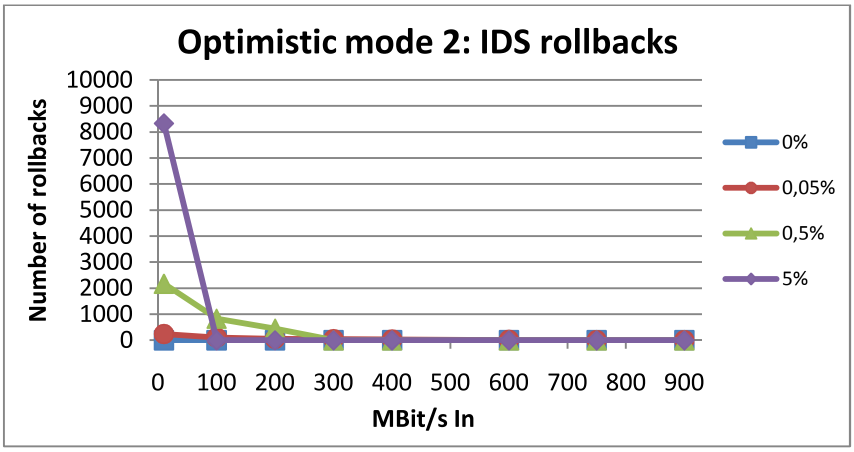

Figure 25 – Number of average rollbacks performed by the intrusion detection systems in optimistic mode 2. Rollbacks decrease with the increase of traffic rate, even if the level of packet loss is the same.

Rollback occurs when an intrusion detection system is ahead of last synchronization point, and the system verifies that misaligned packets were consumed. If the IDS was behind such point there would be not need to perform the rollback. At a 0% packet loss rate, there is never the need for a rollback, as even if the IDS is ahead of the last synchronization point, it always consumes aligned packets. On the other hand, looking at a given packet loss rate line, it is clear that the number of performed rollbacks decreases with the increase of traffic injection rate, which indicates that in less often occasions is the IDS in a position to consume misaligned packets, as the smaller number of rollbacks cannot be explained by a lower probability of occurrence of misaligned packets.

This suggests that the system could perhaps benefit from the ability to switch modes, using mode 2 at lower network traffic rates, and mode 1 at higher ones. Notice, however, that even at 600 MBit/s, the system had the need to perform rollbacks on a few occasions.

5.7.4 Hardware Resources

Besides contention for system resources, and in particular of processing resources, between the synchronization layer and the intrusion detection layer, there is also contention for these resources between the various threads of the synchronization layer. For this reason, the proposed solution should be able to benefit from the availability of a higher number of processors/processor cores on each system in which it is run.

To confirm these benefits, a comparison was made between the throughput achieved by the 8-core server-class set of replicas and the one achieved by the 2-core desktop-class set. As previously mentioned, the two sets of computers available for testing have processors with similar, even if not equal, capabilities and the desktop-class computers were even able to fare slightly better in the intrusion detection system packet loss test.

The comparison was made in optimistic mode 2, as this mode is more processor-demanding, as described in section 5.7.2. Figure 22 shows the results:

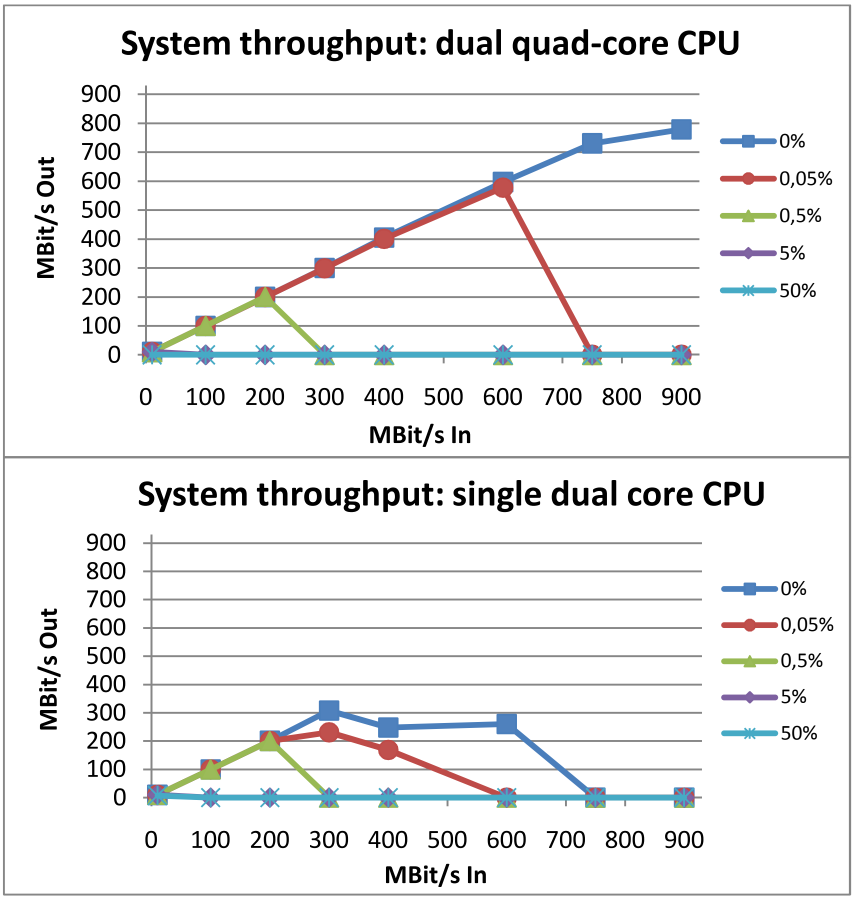

Figure 26 – Effect of the availability of more processing cores on system throughput, in optimistic mode 2. Throughput suffered significantly on the 2-core replica set, when compared with the values achieved by the 8-core replicas, and even at %0 packet loss rates, where resynchronization and rolling back does not occur, the system was no longer able to sustain "wire speed" throughput beyond 300 MBit/s traffic injection rates.

As depicted above, in this test the 2-core hardware set performed significantly worse when compared with the 8-core one, which seems to confirm that the replication systems benefits significantly from the availability of more processing resources for the efficient parallelization of the various tasks it needs to execute.

5.8 Application Scenarios

This section presents two scenarios where the synchronization mechanism could prove useful in real-life situations. During the previous tests, the replica system used the benefit of having, in practice, an unrestricted amount of available memory, allowing it to buffer, if necessary, all packets injected into the network, for later synchronization. This resulted in the occurrence of packet loss only by artificially introducing losses, for traffic rates up to the saturation of a Gigabit Ethernet link. This is of course not sustainable in the long run, if the system is consistently unable to handle the network traffic rates it needs to keep synchronized.

For this reason, in the following examples, the system is restricted from continuously capturing new network packets, if it is unable to dispatch already captured ones in a timely fashion. This is done by limiting the number of captured but unsynchronized packets on each replica. It is not a cap on the amount of shared memory, however, as this would make it unfeasible to retrieve missed packets from other replicas, which in turn is necessary to allow the completion of resynchronization, at which point replicas are allowed to capture further packets from the network.

5.8.1 Traffic Spikes

The first considered application scenario is the case of network traffic with the occurrence of spikes, where the system is unable to handle all traffic, resulting in packet loss. Since these spikes are followed by periods of much lower traffic rates, the system is given time to recover some of the packets each individual replica has lost, and to empty filled up buffers.

The test traffic has the characteristics shown in figure 23:

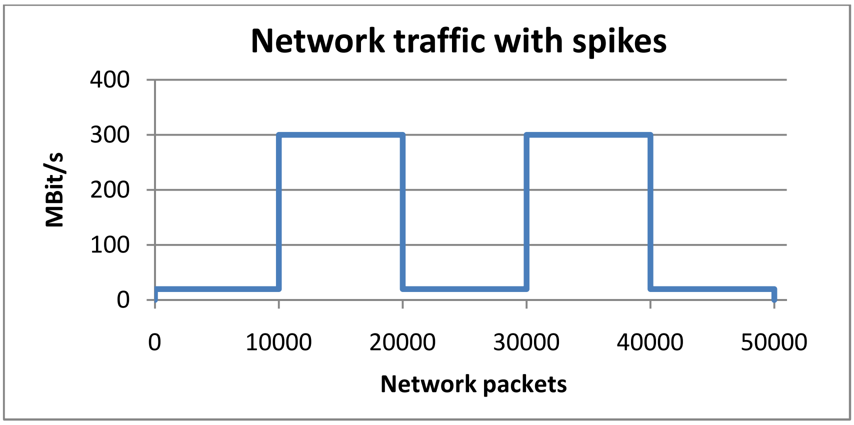

Figure 27 – Traffic spykes.

A batch of 10 thousand packets is injected at a rate of about 20 Mbit/s, followed by another batch injected at about 300 MBit/s. This pattern is repeated and the test is ended by a final batch of 10 thousand packets, injected at 20 MBit/s. Packets used are the UDP-crafted packets, with MAC and IP address randomization. In this test, the server-class hardware was used.

The replicas were run with the following options:

```
# sink -a <id 1> -C <interface> -i <own IP> -p <own port> -X <secret> -B 1024 -M
       10 -o 1 –e <snort executable> -E <snort conf> -u <snort user> -g <snort
       group> -w 1
```

and,

```
# sink -a <id 2> -C <interface> -i <own IP> -p <own port> -X <secret> -B 1024 –M
       10 -o 1 –e <snort executable> -E <snort conf> -u <snort user> -g <snort
       group> -S <leader IP> -s <leader port>
```

The replicas were therefore given the chance to buffer only 1024 unsynchronized packets, and to use a total amount of 10 MB of shared memory. The small buffer works as a manageable extension of the buffering provided by LibPcap. With this setup, the replicas lose some of the packets injected according to the pattern described above, even without using the RandomDrop() function, and so does a stand-alone Snort IDS. Results are given in table 8:

Table 8 – Synchronization of traffic with spikes. Results show three-run averages, in 3 runs where replicas were able to keep alignment.

	Replicas Average	IDS Average
Received	50000	50000
Lost from LibPcap	1266	1070
Unprocessed by Snort	552	1070

It should be noted that not in all runs were replicas able to keep alignment. But the example shows how in some cases the synchronization protocol, even in a situation where individual replicas lose more packets than a stand-alone IDS, is able to recover some of them between replicas, and the replicas intrusion detection system ends up analyzing more packets than the stand-alone version.

5.8.2 Busy Systems

The second application scenario is the case where one system is subjected to a temporary high processor load. This load could be due to some process separate from the intrusion detection process, a there are usually at least a few additional processes running on system dedicated to intrusion detection, for example for system management.

In this test, a batch of 50 thousand packets is injected at a rate of about 50 Mbit/s, followed by the injection of 8 additional packets, to allow for synchronization round completion (56 x 993 = 5008). Packets used are the UDP-crafted packets, with MAC and IP address randomization. Half-way through the run, a processor-consuming process is launched in one of the replicas, which runs for about 10 thousand packets. Here, the desktop-class hardware was used, since systems with a low number of processing cores would be more adversely affected by such situation.

The replicas were run as before. With this setup, the replica where the processor-consuming process is launches loses some of the packets injected according to the pattern described above, even without using the RandomDrop() function, and so would a stand-alone Snort IDS. Results are given in table 9:

Table 9 – Synchronization of traffic with a processor-consuming process. Results show three-run averages, in 3 runs where replicas were able to keep alignment.

	Replica 1 Average	**Replica 2 Average**
Received	50008	50008
Lost from LibPcap	385	0
Unprocessed by Snort	0	0

While the replica where the CPU-intensive process was launched lost a few packets, it was able to recover them from its peer, with the intrusion detection system ending up analyzing all packets in both replicas.

5.9 A Word on Checkpointing

During project development, it became clear that using the Multi-Threaded CheckPoint library would become a performance bottleneck too big to overcome. For this reason, during the evaluation tests, only the `fork()` method of checkpointing was used.

Feasibility of using MTCP to create a process image file that could be transferred between replicas was tested, however, as described in section 4.4.

6 CONCLUSION

The loss of packets by a network intrusion detection system may compromise its ability to detect intrusions or intrusion attempts. Even if the number of packets missed by the intrusion detection system is very low, when compared with the number of packets that were analyzed, some dose of misfortune may dictate that the ones that were missed were the ones that were needed for detecting an attack that would cause significant harm to the organization.

In this project, an attempt was made for providing a solution to avoid or at least minimize these omission failures, by developing a system composed by a set of replicas that could together reduce or avoid packet loss by the system, based on the fact that the probability of loss of a given packet by all the replicas should be lower than the probability of loss of that packet by a system composed by a single replica. The higher the number of replicas in the system, the lower should this probability be. Once detected that a given network packet was missed by one or more replicas, it could be recovered from one of the replicas which had not missed the packet.

To achieve this, replicas periodically exchange a compact representation of the packets that they received, in an attempt to detect if there are differences in the set of packets received by each of them. Once a difference is detected, the replicas try to determine which individual packets in the set are different, and proceed by retrieving from other replicas the ones that were missed.

Results, using a system with two replicas, suggest that this approach could be feasible in practice, given the availability of spare processing capacity which is common in today's multi-core processor computers. This spare capacity could be used for the synchronization tasks, without too much penalty to the network packet analysis tasks. In addition, the resource demands for the synchronization tasks are probably much lower than the demands for the analysis tasks, in particular while the rate of packet loss is very low: as was shown, confirmation of replica alignment introduces very little latency, when comparing the time taken for injecting packets with the time required for confirming alignment, even at the highest tested traffic injection rates, which were of a virtually filled up Gigabit Ethernet link. At this traffic rate, the tested intrusion detection system dropped packets at a rate that comes close to a 10% figure, while the synchronization mechanism was able to continue alignment confirmation without accumulating unverified packets, which indicates it could sustainably continue to do so at those traffic rates, even with a limited amount of available memory buffer for the packet queue.

Obviously, the overhead of the synchronization protocol increases significantly, once packet loss starts to increase, as in addition to detecting misalignment, replicas need to determine which specific packets were missing by each of them, and need to exchange and reintegrate the missed ones in their packet queue. This became evident in tests where packets were artificially dropped, with the added amount of work eventually leading to failure of the synchronization mechanism, for higher traffic injection rates.

The project also explored the alternatives of either allowing the analysis layer to only access packets confirmed to be aligned, or to allow it to analyze packets whose correctness is yet to be confirmed. In the latter case, to ensure alignment between replicas analysis processes, a checkpoint and rollback mechanism was introduced. Allowing an intrusion detection system to analyze yet-to-be-confirmed packets should reduce analysis latency, and could be crucial in the case of intrusion prevention systems. Results suggest that, while this feature seems less relevant at higher traffic rates, as the intrusion detection system lags behind the synchronization mechanism, and ends up analyzing packets at a time where alignment has already been confirmed, for lower traffic rates it does seem relevant, as indicated by the number of performed rollbacks, which are required only when the IDS has already consumed misaligned packets, meaning it was ahead of the last confirmed synchronization point.

Two typical application scenarios were also tested, where the amount of available memory resources were limited, to simulate situations where, regardless of the initial amount of memory available for the queue, that resource would be close to exhaustion. Those application scenarios were the case of a network link with a usually low traffic rate, but in which an occasional traffic spike would occur, and the case where some process running in one replica would briefly require a large amount of processing resources. Results suggest that, in these situations, which ultimately forced one or both replicas to drop packets, some of the packets missed by each replica were in fact different and recoverable from each other, at least in some specific conditions, which illustrates the protocol usefulness.

The developed synchronization algorithm always tries to recover and align all detected missed packets. It became evident that, once approaching resource exhaustion, a production system should refrain from trying to keep synchronizing all the packets captured by the set of replicas, as resource exhaustion indicates that replicas are unable to sustainably handle the amount of traffic passing through the network. A production system should therefore include mechanisms for throttling the amount of resynchronized packets and/or for switching between synchronization and no synchronization modes.

It should be noted, however, that the current implementation does not allow one to conclude where the limits of the synchronization protocol lie. As a project started from scratch, as opposed to one trying to optimize a specific aspect of an existing system, a number of inefficiencies exist in different sub-systems, which may be the subject of optimization. These range from network message exchange efficiency, to packet registration and recovered packets reordering optimization, and to concurrency improvements.

Finally, it also should be highlighted the fact that the limitations of the synchronization protocol become apparent as testing focused on performance under very demanding conditions, as those are the conditions where intrusion detection systems packet loss typically becomes a problem. If packet loss occurrence between the traffic source and the traffic analyzers was due to factors not connected with performance limitations of the analysts system, the protocol could more easily prove to be useful, even with its various inefficiencies.

6.1 Future Work

The initial work undertook in this thesis suggests a number of paths for future work that could be pursued. While the efficiency issues could perhaps be left to the implementers of a production system, a number of possibilities for further research are also presented.

One possible research path would be the development of an efficient n-way version of the (re)synchronization algorithm. While in theory, increasing the number of replicas should reduce the probability of losing one specific packet, in practice the realignment phase becomes much more complex, as replicas now have to compare and decide upon a much higher number of packet sets. This overhead can in turn facilitate further packet loss, thereby defeating the original goal.

A second path would be in the field of replica management, and in particular of crash failure detection. Waiting for the timeout of a reply to a request to suspect of a crash failure of a peer could result in the accumulation of a very large number of packets in the queue, and to the loss of some. This simple approach is therefore probably not suited for this task.

Another option for further research could lie in the development of failback mechanisms, once determined that resynchronization of packets could not continue. How to optimize the balance between trying to recover packets and controllably allow the loss of some? Also, resuming synchronization would probably be costlier than maintaining it. This is at least what happens in the current bootstrapping implementation.

Finally, opportunities for the applicability of the protocol to wireless environments could perhaps be explored. Wireless network environments are becoming increasingly important, and they provide a noisy environment where a wireless intrusion detection system can miss packets that could be picked up by another in a more favorable location. While wireless intrusion detection would present a number of challenges for a WIDS analyzing radio signals, namely the opacity of ciphered traffic from a client associated with a given infrastructure access point, perhaps this approach could be useful in detection of pre-association attacks. Also, a number of wireless service providers do not rely on mechanisms such as WEP or WPA/WPA2 for allowing client access, but instead on authentication on a Web portal, and leave the wireless communication channel un-ciphered.

7 REFERENCES

[1] Gollman, D. (2006). *Computer security, 2nd Edition.* New York, NY, USA: John Wiley & Sons, Inc.

[2] Roesch, M. (1999). Snort - lightweight intrusion detection for networks. *LISA '99: Proceedings of the 13th USENIX conference on System administration* (pp. 229-238). Berkeley, CA, USA: USENIX Association.

[3] Mukherjee, B., Heberlein, L. T., & Levitt, K. N. (1994, May/Jun.). Network intrusion detection.

[4] Stallings, W. (2003). *Network Security Essentials: Applications and Standards* (Second Edition ed.). (I. Pearson Education, Ed.) Upper Saddle River, New Jersey: Prentice Hall.

[5] Gupta, K. K. (2009). *Robust and Efficient Intrusion Detection Systems.* PhD Thesis, The University of Melbourne, Department of Computer Science and Software Engineering.

[6] Anderson, J. P. (1980). *Computer security threat monitoring and surveillance.* Technical Report Contract 79F296400, James P. Anderson Co, Box 42 Fort Washington, Pennsylvania.

[7] Allen, J., Christie, A., Fithen, W., McHugh, J., Pickel, J., & Stoner, E. (2000). *State of the practice of intrusion detection technologies.* Technical Report CMU/SEI-99-TR-028, Carnegie Mellon, Software Engineering Institute.

[8] Bishop, M. (2003). *Computer Security: Art and Science.* Boston, USA: Addison-Wesley Professional.

[9] Axelsson, S. (2000). *Intrusion-detection systems: A Taxonomy and Survey.* Chalmers University of Technology, Department of Computer Engineering. SE-412 96, Goteborg, Sweden.

[10] Debar, H., Dacier, M., & Wespi, A. (2000). *A revised taxonomy for intrusion-detection systems.* Technical report, Zurich Research Laboratory, IBM Research Division.

[11] Sabahi, F., & Movaghar, A. (2008). Intrusion detection: A survey. *Systems and Networks Communications, 2008. ICSNC '08. 3rd International Conference on, Oct. 2008*, (pp. 23-26).

[12] Axelsson, S. (2000). *A preliminary attempt to apply detection and estimation theory to intrusion detection.* Technical Report 00-4, Chalmers University of Technology, Department of Computer Engineering, SE-412 96, Goteborg, Sweden.

[13] Zúquete, A. (2008). *Segurança em Redes Informáticas* (2ª Edição ed.). FCA -Editora de Informática.

[14] Leu, F.-Y., Lin, J.-C., Li, M.-C., Yang, C.-T., & Shih, P.-C. (2005). Integrating grid with intrusion detection. *AINA '05: Proceedings of the 19th International Conference on Advanced Information Networking and Applications* (pp. 304-309). Washington, DC, USA: IEEE Computer Society.

[15] Senger, H., & Nakahara Junior, J. (2008). Applying computational grids for enhancing intrusion detection systems. *CSEWORKSHOPS '08: Proceedings of the 2008 11th IEEE International Conference on Computational Science and Engineering - Workshops* (pp. 149-156). Washington, DC, USA: IEEE Computer Society.

[16] Janakiraman, R., Waldvogel, M., & Zhang, Q. (2003). Indra: A peer-to-peer approach to network intrusion detection and prevention. *WETICE '03: Proceedings of the Twelfth International Workshop on Enabling Technologies* (p. 226). Washington, DC, USA: IEEE Computer Society.

[17] Axelsson, S. (2005). *Understanding Intrusion Detection Through Visualisation.* PhD thesis, Chalmers University of Technology, School of Computer Science and Engineering, Goteborg, Sweden.

[18] Axelsson, S. (2000). *ACM Transactions on Information and System Security (TISSEC) , 3* (3), 186-205.

[19] Clark, C. R., & Schimmel, D. E. (Sep. 2003). Efficient reconfigurable logic circuits for matching complex network intrusion detection patterns. *Proceedings of International Conference on Field-Programmable Logic and Applications (FPL).* Lisbon, Portugal.

[20] Mitra, A., Najjar, W., & Bhuyan, L. (2007). Compiling pcre to fpga for accelerating snort ids. *ANCS '07: Proceedings of the 3rd ACM/IEEE Symposium on Architecture for networking and communications systems* (pp. 127-136). New York, NY, USA: ACM.

[21] Andrade, P. F. (Sep. 2007). *Port hopper - an efficient intrusion detection system for networks with centralized routing.* Master's thesis, Instituto Superior Técnico.

[22] Northcutt, S., & Novak, J. (Sep. 2002). *Network Intrusion Detection.* New Riders Publishing.

[23] *Snort (R) users manual 2.8.5, the snort project*. (Oct. 2009). Retrieved from http://www.snort.org/assets/125/snort_manual-2_8_5_1.pdf.

[24] Beale, J., Baker, A. R., & Esler, J. (2007). *Snort IDS and IPS Toolkit.* Burlington, Ma, USA: Syngress Publishing, Inc.; Elsevier, Inc.,.

[25] Ptacek, T. H., & Newsham, T. (Jan. 1998). *Insertion, evasion, and denial of service: Eluding network intrusion detection.* Technical report, Secure Networks, Inc.

[26] Avizienis, A., Laprie, J.-C., Randell, B., & Landwehr, C. (2004). Basic concepts and taxonomy of dependable and secure computing. *IEEE Trans. Dependable Secur. Comput. , 1* (1), 11-33.

[27] Guerraoui, R., & Schiper, A. (1997). Software-based replication for fault tolerance. *Computer , 30* (4), 68-74.

[28] Cristian, F. (1991). Understanding fault-tolerant distributed systems. *Commun. ACM , 34* (2), 56-78.

[29] Gartner, F. C. (1999). Fundamentals of fault-tolerant distributed computing in asynchronous environments. *ACM Comput. Surv. , 31* (1), 1-26.

[30] Coulouris, G., Dollimore, J., & Kindberg, T. (2005). *Distributed Systems: Concepts and Design (International Computer Science)* (4th Edition ed.). Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc.

[31] Schneider, F. B. (1990). Implementing fault-tolerant services using the state machine approach: a tutorial. *ACM Comput. Surv. , 22* (4), 299-319.

[32] Défago, X., Schiper, A., & Urbán, P. (2004). Total order broadcast and multicast algorithms: Taxonomy and survey. *ACM Computing Surveys , 36* (4), 372-421.

[33] Plank, J. S., Beck, M., Kingsley, G., & Li, K. (1994). *Libckpt: Transparent checkpointing under unix.* Technical report, Knoxville, TN, USA.

[34] Ansel, J., Aryay, K., & Cooperman, G. (2009). Dmtcp: Transparent check-pointing for cluster computations and the desktop. *IPDPS '09: Proceedings of the 2009 IEEE International Symposium on Parallel&Distributed Processing* (pp. 1-12). Washington, DC, USA: IEEE Computer Society.

[35] Rieker, M., Ansel, J., & Cooperman, G. (2006). Transparent user-level check-pointing for the native posix thread library for linux. In H. R. Arabnia (Ed.), *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications & Conference on Real-Time Computing Systems and Applications, PDPTA, June 26-29, 2006. 1*, pp. 492-498. Las Vegas, Nevada, USA: CSREA Press.

[36] Kim, B.-J. (Oct. 2005). *Comparison of the existing checkpoint systems.* Technical report.

[37] *PCAP manual page*. (n.d.). Retrieved from http://www.tcpdump.org/pcap3_man.html.

[38] *Jumbo Frame*. (n.d.). Retrieved from http://en.wikipedia.org/wiki/Jumbo_frame.

[39] *README.stream5 document*. (n.d.). Retrieved from http://cvs.snort.org/viewcvs.cgi/snort/doc/README.stream5?rev=1.19&content-type=text/vnd.viewcvs-markup.

[40] Khanna, S., Kunal, K., & Pierce, B. C. (2007). A Formal Investigation of Diff3. *Foundations of Software Technology and Theoretical Computer Science*, (pp. 485-496).

[41] Halme, L. R., & Bauer, R. K. (n.d.). *AINT Misbehaving: A Taxonomy of Anti-Intrusion Techniques*. Retrieved 04 28, 2012, from http://www.sans.org/security-resources/idfaq/aint.php