Sistema de Apoio à Gestão de Redes e Serviços
(Network Management Support System)

Bruno Mota Pena
(Licenciado)

Dissertação para obtenção do grau:
Mestre em Engenharia de Redes de Comunicações

Comité de Avaliação

Presidente: Prof. Doutor Rui Jorge Morais Tomaz Valadas
Orientadora: Prof.ª Doutora Teresa Maria Sá Ferreira Vazão Vasques
Vogais: Prof. Doutor Luis Filipe Lourenço Bernardo

Outubro de 2010
This thesis is dedicated to my parents and my sister, for their unconditional support and encouragement.
Abstract

IP networks have had an explosive growth in the past years, in both size and complexity. Today they are used everywhere, from the Internet to private companies and even in our homes, connecting all types of devices you can imagine: computers, printers, game consoles, televisions, mobile phones, sensors, etc.

Accompanying this growth are the Network Management Systems. Today they’re much more efficient and optimized than the first systems, but also very complex and difficult to deploy.

This thesis proposes a new, simpler and less intrusive solution to detect faults in IP networks, by analyzing and correlating pre-captured data from different network points in the same period of time. It uses a Framework that fulfills the following requirements:

- **Modular** – Composed by modules that provide the framework features.
- **Expandable** – Support for plugins to ease the inclusion of new features (application level).
- **Flexible** – Easily add or remove components (either modules or plugins).
- **Multi-threaded** – Ability to run multiple plugins at the same time.
- **Plugin isolation** – Plugins must not inherit the complexity of the frameworks multi-threading environment. It is necessary to isolate and provide a secure single-threaded environment for each plug-in to run.
- **Simplicity** – The framework should provide the simplest working environment for plugins, implementing only the essential features.

This Framework is used as the basis to an architecture consisting of three applications:

- **Collector** – Collects data from the network.
- **Analyzer** – Analyzes a Collectors data.
- **Correlator** – Correlates data from all Analyzers.

The Collector can be configured to be completely passive, so it doesn’t interfere with normal network operation, or to actively collect large amounts of information
about the network operation. In order to obtain a complete view of the network, several Collectors can be scattered over the network.

For each Collector, there is an Analyzer. It represents the first stage of data analysis and correlation, and is used to generate new, more rich and refined, data.

The Correlator is the architecture gathering point. It merges the data from all Analyzers, and is the second stage of data analysis and correlation. It is used to determine the root cause of the alarms reported by the Analyzers.

All three applications were tested with data captured from the network of Instituto Superior Tecnico – Taguspark.

**Keywords:** Network Management Systems, IP networks, Faults, Alarms, Event Correlation
Resumo

As redes IP têm tido um crescimento explosivo nos últimos anos, tanto em tamanho como em complexidade. Actualmente são utilizadas em quase todos os domínios, desde a Internet até às empresas privadas, e até mesmo nas nossas casas, interligando todos os tipos de dispositivos que possamos imaginar: computadores, impressoras, consolas de jogos, televisores, telemóveis, sensores, etc..

A acompanhar este crescimento estão os Sistemas de Gestão de Redes que, embora actualmente muito eficiente e optimizados quando comparados com os primeiros sistemas, estão também a tornar-se cada vez mais complexos e difíceis de implementar.

Esta tese propõe uma nova solução mais simples e menos intrusiva para detectar falhas em redes IP, através da análise e correlação de dados pré-capturados em pontos distintos no mesmo período de tempo. Esta solução tem por base uma Framework que respeita os seguintes requisitos:

- **Modular** – Composta por módulos que constituem as funcionalidades da framework.
- **Expansível** – Suporte para plugins de modo a facilitar a inclusão de novas funcionalidades (ao nível da aplicação).
- **Flexível** – Facilidade na inclusão e exclusão de componentes (sejam eles plugins ou módulos).
- **Multi-tarefa** – Capacidade de executar múltiplos plugins ao mesmo tempo.
- **Isolamento dos Plugins** – Os plugins não devem herdar a complexidade da programação multi-tarefa. É necessário isolar os plugins e fornecer-lhes um ambiente mono-tarefa seguro onde se possam executar.
- **Simplicidade** – A framework deve ser o mais simples possível, providenciando apenas as funcionalidades essenciais necessárias aos plugins.

Esta Framework é utilizada como base numa arquitectura composta por três aplicações:

- **Collector** – Captura dados da rede.
Resumo

- **Analyzer** – Analisa os dados do Collector.
- **Correlator** – Correlaciona os dados de todos os Analyzers.

O **Collector** pode ser configurado para ser completamente passivo, de modo a não interferir com o normal funcionamento da rede, ou para activamente proceder à recolha de grandes quantidades de informações relacionadas com o funcionamento da rede. A fim de obter uma visão completa sobre o funcionamento da rede, vários **Collectors** podem ser espalhados por diversos pontos distintos da rede.

Para cada Collector existe um **Analyzer**. **Este** representa a primeira fase da análise e correlação de dados, sendo utilizado para gerar novos dados mais refinados.

O **Correlator** é o ponto de encontro da arquitectura. **Representa a segunda fase** de análise e correlação de dados, onde são agregados os dados de todos os **Analyzers**. É utilizado para determinar a causa dos alarmes gerados pelos vários **Analyzers**.

Todas as três aplicações foram testadas com dados capturados na rede do **Instituto Superior Técnico – Taguspark**.

**Palavras-chave:** Sistemas de Gestão de Redes, Redes IP, Falhas, Alarmes, Correlação de Eventos
# Contents

1 Introduction ................................................. 1
   1.1 Motivation ........................................... 1
   1.2 Objectives and expected contributions ................. 2
   1.3 Structure of the document ............................. 2

2 Related work ............................................ 3
   2.1 Techniques for collecting data ......................... 3
      2.1.1 Passive techniques .............................. 3
      2.1.2 Active techniques .............................. 4
   2.2 Anomaly detection .................................. 6
      2.2.1 Hard-failures .................................. 6
      2.2.2 Soft-failures .................................. 6
   2.3 Event correlation ................................... 8
      2.3.1 Artificial intelligence ......................... 9
      2.3.2 Data mining .................................. 11
      2.3.3 Fault propagation ............................ 11
   2.4 Dynamic configuration ............................. 12
      2.4.1 SNMP – Policy Management MIB ................. 13
      2.4.2 Common Open Policy Service ................... 13

3 Architecture and Implementation – Framework .......... 15
   3.1 Bindings ............................................. 16
   3.2 Network & Protocols ............................... 17
   3.3 Metaplugiin ......................................... 19
      3.3.1 Plugin ........................................ 19
      3.3.2 Dependency loop detection .................... 21
      3.3.3 Metaplugiin guarantees ....................... 22
   3.4 Looplog ............................................. 26
   3.5 Persistence ......................................... 26
      3.5.1 Objects and instances ......................... 27
      3.5.2 Services provided ............................ 27
List of Figures

2.1 Template and envelopes ........................................ 8
2.2 Techniques used in event correlation.......................... 10
2.3 Architecture of a Policy-based Networking .................... 13

3.1 Application making use of the common Framework .............. 16
3.2 Bindings between Lua applications and the Operating System ... 16
3.3 Conceptual view of a plugin .................................. 19
3.4 Plugins as building blocks for other plugins .................. 20
3.5 Threads calling the plugin interface .......................... 22
3.6 Flowchart of the Metaplugins load method .................... 23
3.7 Example of plugin dependency loop ............................ 24
3.8 Diagram of all threads involved in the Metaplugins start method 25
3.9 Instances of different objects (seen as unknown/opaque data structures) ........................................ 27
3.10 Transparent and sorted access to instances on multiple input folders . 28
3.11 Flowchart of the Persistence setup method ................... 29
3.12 Write of an object instance .................................. 30
3.13 Reading object instances from files ........................... 32
3.14 Exchange of information between plugin instances ............ 34
3.15 Subscription of virtual queues by real queues ................ 35
3.16 Serialization of simultaneous calls to shared functions ....... 38
3.17 Range of each sample, using fixed limits ..................... 39
3.18 Range of each sample, using extended limits ................. 39
3.19 Event filtering to discover the root cause event .............. 42
3.20 Using Scheduler module to delay repeated actions .......... 43

4.1 Overview of the proposed architecture ......................... 49
4.2 Flowchart of all three applications ............................ 51
4.3 Collectors scattered over the network .......................... 52
4.4 Merge of topology from several Analyzers by the Correlator ... 54
<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.1</td>
<td>Network topology built using data from <em>Collector A</em></td>
<td>59</td>
</tr>
<tr>
<td>5.2</td>
<td>Comparison between the three Lua flavors (1 Thread)</td>
<td>62</td>
</tr>
<tr>
<td>5.3</td>
<td>Comparison between the multi-threaded Lua flavors (2 Threads)</td>
<td>63</td>
</tr>
<tr>
<td>5.4</td>
<td>Comparison between the multi-threaded Lua flavors (4 Threads)</td>
<td>63</td>
</tr>
<tr>
<td>5.5</td>
<td>Comparison between the Protocols and Network modules (1 Thread)</td>
<td>64</td>
</tr>
<tr>
<td>5.6</td>
<td>Comparison between the Protocols and Network modules</td>
<td>65</td>
</tr>
<tr>
<td>B.1</td>
<td>Comparison of execution time with and without CPU relinquish</td>
<td>78</td>
</tr>
</tbody>
</table>
List of Tables

5.1 Execution time (in seconds) using Original Lua (no multi-thread support) ........................................ 61
5.2 Execution time (in seconds) using Permissive Lua .................. 61
5.3 Execution time (in seconds) using Aggressive Lua ................. 62
5.4 Comparison of the execution time against Original Lua (1 Thread) . 63
A.1 Summary of the scripting languages that have been considered ..... 72
# Acronyms

<table>
<thead>
<tr>
<th>Acronym</th>
<th>Full Form</th>
</tr>
</thead>
<tbody>
<tr>
<td>AADS</td>
<td>Address Anomaly Detection System</td>
</tr>
<tr>
<td>ANSI</td>
<td>American National Standards Institute</td>
</tr>
<tr>
<td>ARP</td>
<td>Address Resolution Protocol</td>
</tr>
<tr>
<td>ARPANET</td>
<td>Advanced Research Projects Agency Network</td>
</tr>
<tr>
<td>ASN.1</td>
<td>Abstract Syntax Notation One</td>
</tr>
<tr>
<td>COPS</td>
<td>Common Open Policy Service Protocol</td>
</tr>
<tr>
<td>CPU</td>
<td>Central Processing Unit</td>
</tr>
<tr>
<td>DBSCAN</td>
<td>Density-Based Spatial Clustering of Applications with Noise</td>
</tr>
<tr>
<td>FPM</td>
<td>Fault Propagation Model</td>
</tr>
<tr>
<td>HTTP</td>
<td>Hypertext Transfer Protocol</td>
</tr>
<tr>
<td>ICMP</td>
<td>Internet Control Message Protocol</td>
</tr>
<tr>
<td>IETF</td>
<td>Internet Engineering Task Force</td>
</tr>
<tr>
<td>IP</td>
<td>Internet Protocol</td>
</tr>
<tr>
<td>IPsec</td>
<td>Internet Protocol Security</td>
</tr>
<tr>
<td>ISO</td>
<td>International Organization for Standardization</td>
</tr>
<tr>
<td>ITU</td>
<td>International Telecommunication Union</td>
</tr>
<tr>
<td>MIB</td>
<td>Management Information Base</td>
</tr>
<tr>
<td>MTU</td>
<td>Maximum Transfer Unit</td>
</tr>
<tr>
<td>OID</td>
<td>Object Identifier</td>
</tr>
<tr>
<td>OS</td>
<td>Operating System</td>
</tr>
<tr>
<td>OSI</td>
<td>Open Systems Interconnection</td>
</tr>
<tr>
<td>OSPF</td>
<td>Open Shortest Path First</td>
</tr>
<tr>
<td>PAMS</td>
<td>Performance and Anomaly Monitoring System</td>
</tr>
<tr>
<td>PBN</td>
<td>Policy-based Networking</td>
</tr>
<tr>
<td>PDP</td>
<td>Policy Decision Point</td>
</tr>
<tr>
<td>PEP</td>
<td>Policy Enforcement Point</td>
</tr>
<tr>
<td>PR</td>
<td>Policy Repository</td>
</tr>
<tr>
<td>QoS</td>
<td>Quality of Service</td>
</tr>
<tr>
<td>RAM</td>
<td>Random Access Memory</td>
</tr>
<tr>
<td>RCA</td>
<td>Root Cause Analysis</td>
</tr>
<tr>
<td>SLL</td>
<td>Link-layer pseudo-protocol (SLL stands for sockaddr_ll)</td>
</tr>
<tr>
<td>Acronym</td>
<td>Description</td>
</tr>
<tr>
<td>---------</td>
<td>-------------</td>
</tr>
<tr>
<td>SMI</td>
<td>Structure of Management Information</td>
</tr>
<tr>
<td>SMTP</td>
<td>Simple Mail Transfer Protocol</td>
</tr>
<tr>
<td>SNMP</td>
<td>Simple Network Management Protocol</td>
</tr>
<tr>
<td>TCP</td>
<td>Transmission Control Protocol</td>
</tr>
<tr>
<td>TLS</td>
<td>Transport Layer Security</td>
</tr>
<tr>
<td>UDP</td>
<td>User Datagram Protocol</td>
</tr>
<tr>
<td>VLAN</td>
<td>Virtual Local Area Network</td>
</tr>
</tbody>
</table>
Chapter 1
Introduction

1.1 Motivation

Since its first appearance on the ARPANET, in 1983, there has been an explosive growth of IP networks. Today they are used everywhere, from the Internet to private companies and even in our homes, connecting all types of devices you can imagine: computers, printers, game consoles, televisions, mobile phones, sensors, etc... Associated with this growth is the increased complexity of the networks. Not only becomes more difficult to understand the dynamics of these heterogeneous networks, but also to configure and maintain them [1].

Today’s scenario of massive technology adoption and wide use of Internet by organizations, has put an enormous pressure on the correct operation of all networks. Organizations are willing to invest significant amounts of money in network infrastructures to create more value in a highly competitive environment, which makes them less tolerant to any malfunction that affects its performance. Consequently, network management is a task that’s becoming more and more important.

The first Network Management Systems were created before this massive technology adoption, and could only perform very simple tasks. It quickly became obvious that to follow the growth of IP networks, it was necessary to increase the efficiency and autonomy of these systems. Several studies were made to find out how to use the knowledge from other areas of research in the network management systems. Since then, several techniques have been adopted: from artificial intelligence to simulate human behavior [2], from theory of computation to model fault propagation [2], from data mining to find patterns on large amounts of data [21], among others.

These researches have been quite successful, and we now find many of those techniques on several different network management systems. Yet, today’s network management systems, which were initially created to reduce the complexity of configuring and maintaining various network devices and services, have become themselves very complex and difficult to manage. Most of them requires a full-scale deployment (with customized settings and equipment throughout the network) and are only capable of real-time processing. To be efficient and scalable this type of
analysis has to operate on a very limited amount of information, that is collected on each node and transferred over the network periodically [3]. This makes it very hard to get a detailed analysis of how the network is operating, since the collected data contains only a partial amount of information [3].

This thesis proposes a new, simpler and less intrusive solution, that analyzes and correlates pre-captured data from different network points in the same period of time. To capture network data, it uses a set of capture probes that can be configured to be completely passive, so they don’t interfere with normal network operation, or to actively collect large amounts of information about the network.

1.2 Objectives and expected contributions

The main objective of the thesis is the implementation of a Framework to aid in the creation of an architecture capable of detecting and analysing faults on IP networks.

The Framework consists in a set of modules that provide basic functionality to the three applications that compose the architecture:

- **Collector** – Collects data from the network.
- **Analyzer** – Analyzes a Collector’s data.
- **Correlator** – Correlates data from all Analyzers.

Both the applications and Framework should fulfill the following requirements:

- Capture network data passively and actively;
- Off-line analysis and correlation of network data;
- Provide statistics about network usage;
- Flexible and easily extensible (through plugins);
- Easy configuration through high-level policy.

If all of this requirements are fulfilled the architecture should be able to evolve, by expanding the number of supported protocols and algorithms through the creation of new modules and plugins.

1.3 Structure of the document

This document has six chapters, this being the first one. In the second chapter it will be presented an analysis of the work that is related to theme of this thesis. The third and forth chapters will introduce the architecture and implementation of the framework and applications, respectively. The fifth describes the tests that have been performed and theirs results. Finally, the sixth chapter ends this document with the conclusion of all the work that has been done.
Chapter 2
Related work

In this chapter it will be presented an overview of the standards and researches related to the topic of this thesis. It’s divided into four main sections:

- **Techniques for collecting data**: Introduces the different types of data collection techniques and gives some examples.
- **Anomaly detection**: Defines the types of anomalies that can happen in a network, and how they can be detected at the different layers of the OSI model.
- **Event correlation**: Gives an overview of the available techniques for use in event correlation.
- **Dynamic configuration**: Presents the most used protocols for dynamic configuration.

### 2.1 Techniques for collecting data

There are several techniques that can be used to collect data from several different types of data sources, which can be divided into two categories:

- **Passive**: Don’t interfere with the normal network operation (e.g. capture network packets using the libpcap library [4]).
- **Active**: Interact with other network devices to obtain information (e.g. ping, traceroute, SNMP polling [5], etc.).

#### 2.1.1 Passive techniques

Passive techniques don’t interfere with the normal network operation, as they don’t generate any (additional) network traffic to obtain information about the network status.
The most known and used technique is listening to network packets (e.g. with the libpcap library [4]. The analysis of these packets provide invaluable information about the network: maximum transfer unit, round trip time, packet loss rate, congestion, unreachable destinations, among many others. But this is not the only technique available. In fact, almost every single network application can be used to infer information about the network status. Whenever an application deviates from its normal behavior (e.g. application that cannot connect to the server), it’s possible to extract some data (e.g. unreachable server). This data can then be used in a correlation engine to extract possible causes for the abnormal application behavior.

2.1.2 Active techniques

Active techniques interact with other network devices to obtain information about the network status. There are many possible techniques that can be used, and most of them uses sampling to collect network data.

2.1.2.1 Sampling

Accurate measures of the network status (e.g. usage, performance, etc.) are required for network management. For example, it may be necessary to monitor the bandwidth consumption of several hundred links in a distributed system to pinpoint bottlenecks. If the monitoring is too aggressive, it may create artificial bottlenecks. With too passive a scheme, the network monitor may miss important events [3]. Network query rates must strike a balance between accurate performance characterization and low bandwidth consumption, to avoid changing the behavior of the network while still providing a clear picture of the behavior. This balance is often achieved through sampling. Sampling techniques are used to study the behavior of a population of elements based on a representative subset. In general, the samples are taken periodically at some fixed interval or in some random distribution. Such sampling reduces bandwidth and storage requirements for the monitored data.

In a high-performance network, sampling overhead must have a minimal impact on the low-latency application transactions, while high network throughputs require frequent sampling to capture transient behavior. For example, many distributed applications (e.g. parallel discrete-event simulation, database systems, etc.) may require frequent synchronization, large data transfers, or both. Network management queries could delay critical data on a single link and thus reduce the efficiency of the entire application.

Under some traffic loads, simple periodic sampling may be poorly suited to the monitoring task. For example, during periods of idle activity or low network loads, a long sampling interval provides sufficient accuracy at a minimal overhead. However, bursts of high activity require shorter sample intervals to accurately measure
network status at the expense of increased sample traffic overhead. To address this issue, adaptive sampling techniques can be employed to dynamically adjust the sampling interval and optimize accuracy and overhead [3]:

- **Linear prediction**: Uses previous samples to estimate or predict a future measurement. Can be used with a set of rules defining the adjustments to make when the prediction is inaccurate.
- **Fuzzy logic**: Mimics the decision processes employed by the human brain. For example, a network administrator may reason that when network load is low, the sample interval can be increased.

### 2.1.2.2 Network tools

The most used tools in the detection and identification of problems in IP networks are **ping** and **traceroute**. They can detect packet loss, jitter, find the path MTU (Maximum Transfer Unit), and discover the network topology [6].

But there are many other tools that can also be very helpful. For example, in the Linux operating system we have:

- **arp**: Display the list of known neighbors, with their MAC addresses and IP.
- **route**: Shows the complete routing table, and allows to change it.
- **dig**: Perform queries on a DNS name server.
- **iperf**: Measures TCP and UDP bandwidth performance.
- **whois**: Display information about a specific IP address (network, ownership, etc.).

### 2.1.2.3 Simple Network Management Protocol

The SNMP protocol [5], defined by IETF as part of the Internet Protocol Suite, is the most widely used protocol for managing IP networks. It covers the functional areas of performance and failure management (OSI/ISO Network Management Model [7]), and implements the communication and information models.

SNMP runs over UDP/IP and supports push and pull interactions, which allows real-time monitorization. For the management of the various network devices is used a virtual database, called Management Information Base. This database consists of a hierarchy of objects, identified and accessed through their OID (Object Identifier). The objects are defined using a subset of ASN.1 (Abstract Syntax Notation One) called SMI [8] (Structure of Management Information), and are available in the form of variables that can be read and written by the management applications. ASN.1 [9] is a standard and flexible notation defined by ISO and ITU-T. It describes a precise and formal set of rules and data structures that are used to represent, encode, transmit and decode data.

A network managed by SNMP consists of three main components:
• *Managed Device*: A device that implements an SNMP agent.
• *Agent*: Software component running on a Managed Device, that converts SNMP data to its native format (and vice-versa).
• *Network Management System*: Software component that runs on one or more servers, monitoring and managing several Managed Devices.

There are four basic operations in SNMPv1:

- *GetRequest*: Get the value of a given variable.
- *GetNextRequest*: Iterator that gets the value of the next variable.
- *SetRequest*: Set the value of a variable and/or execute commands.
- *Trap*: Used by the Agent to report asynchronous events to the Network Management System.

Currently in its third version, the main differences are the inclusion of new data types (SMIv2), new operations (GetBulk, Notification, and Inform Report), and security mechanisms for authentication, access control and privacy.

### 2.2 Anomaly detection

Anomaly detection is an algorithm that raises alarms whenever a network anomaly is detected. These anomalies can be [10]:

- **Hard-failures**: Inability to send or receive packets. These can be caused by power failures, hardware failures, etc.
- **Soft-failures**: Performance degradation or loss available bandwidth. There can be many reasons for this kind of failure to happen, including: inappropriate use of the network, temporary congestion, packet loss, etc. Please note that when these failures occur the network is still usable, albeit with a lower performance than usual.

#### 2.2.1 Hard-failures

Due to its catastrophic consequences, Hard-failures are somewhat easy to detect. They don’t require any special model or algorithm to be detected.

#### 2.2.2 Soft-failures

Soft-failures on the other hand, are more difficult to detect since their only consequence is a slight change in the networks performance [10].

To detect Soft-failures, a generic three step model is used [1]:

- Step 1: Monitor network parameters (e.g., packet loss, bandwidth)
- Step 2: Compare current values to historical data
- Step 3: Raise alarms if significant deviations are detected
2.2 Anomaly detection

1. **Data sampling and filtering**: All data sources are sampled and the results filtered.
2. **Data analysis** [1, 11]: Analysis of the filtered data to create statistical indicators, capable of detecting subtle changes in the network.
3. **Anomaly detection** [1, 11]: Correlation (using a reactive or proactive model) of the various indicators. If an anomaly is found, an alarm is generated.

Due to its simplicity, it’s suggested in [12] the implementation of this model in all network nodes, using the nodes MIB as the data source. This implementation, if successful, allows the construction of distributed anomaly detection systems.

### 2.2.2.1 Link Layer – 802.3 Ethernet

The detection and identification of Soft Failures on Ethernet networks can be made using two complementary algorithms:

- **Performance and Anomaly Monitoring System** [10]: Identifies anomalous points in time series data by developing a prediction of normal behavior, called a template, and tolerance limits, called envelopes, based on a model of data variance. Current data that falls outside of the tolerance envelopes is considered anomalous. PAMS uses time varying envelopes, has different models for weekdays and weekends, and updates the model daily to adapt to changes in the environment. PAMS also has four envelope levels (with values from -2 to 2) to indicate degrees of anomalous behavior, as illustrated in Fig. 2.1 [10].

- **Address Anomaly Detection System** [10]: This is an anomaly detection algorithm based on activity records. For each Ethernet address two records are kept (one for source address, and another destination address). These records contain the network usage mean and standard deviation, which are updated every five minutes. The AADS detects and generates alarms for three types of events: increase in network usage, new/illegal address, and decreased in network usage. Just like PAMS, AADS also has an adaptive model. Its thresholds vary by time of day, week and year.

### 2.2.2.2 Network Layer – IP

To implement a mechanism for anomaly detection at the IP level, first it’s necessary to choose which data sources to use. There are several possibilities [13]:

- **Network probes**: Devices that provide information about latency and packet loss (through the use of tools such as Ping and Traceroute). Due to the differentiation that is made to the ICMP protocol [27] (low priority, and often blocked in the firewalls), this may not be a reliable method.

- **Network sniffers**: Devices that captured packet headers and use them to infer network properties. Due to the high volume of data that is require to capture and process, this method requires a powerful computer.
Fig. 2.1: Template and envelopes

- **Routing protocols**: An OSPF [14] node receives routing table updates in (almost) real-time. This data can be used to build the network topology, with information about the usage and status of the various links. Due to the necessity of keeping the routing tables small, the information contained in these updates is usually limited, and may not be sufficient for anomaly detection.

- **Network Management Protocols**: Standardized protocols (e.g. SNMP [5]) that provide information about network usage and performance. Due to their simplicity, ease of use, and amount of information available, they’re usually the data source of choice.

The combination of these data sources determines the available indicators that can be used for anomaly detection. These indicators allow, through their statistical analysis [13], to detect subsets of problems at the IP level.

### 2.3 Event correlation

Event correlation algorithms are used to analyze a set of alarms and determine the fault that originated them.

Fausts may be classified according to their duration time as [2]:

- **Permanent**: A permanent fault exists in a network until a repair action is taken.
- **Intermittent**: Intermittent faults occur on a discontinuous and periodic basis causing a degradation of service for short periods of time.
- **Transient**: Transient faults cause a temporary and minor degradation of service.
Alarms can be classified into one of the following two categories [2]:

- **Primary**: Alarms generated by the detection of an anomaly in the network (e.g. router failure).
- **Secondary**: Alarms are generated as a result an existing fault (e.g. a service that fails to respond due to the router failure).

Both types of alarms have the following properties [2]:

- **Ambiguity**: The same alarm may be generated as an indication of many different faults (1:N).
- **Inconsistency**: Disagreement among agents with regard to the facts related to network operation. One may have the perception that a component is operating correctly, while other may consider the component faulty.
- **Incompleteness**: Some alarms may be lost or delayed due to faults that originated them.

The number of alarms generated by a fault may depend on many factors such as dependencies among network devices, current configurations, services in use since fault occurrence, presence of other faults, values of other network parameters, etc. [2] Due to this non-determinism the system knowledge may be subject to inaccuracy and inconsistency. Fault evidence may also be inaccurate because of spurious alarms, which are generated by transient problems or as a result of overly sensitive fault detection mechanisms. When spurious symptoms occur, the management system may not be sure which observed alarms should be taken into account in the fault localization process [2].

The additional complication of fault localization results from the fact that different related and unrelated faults may happen within a short time period [2]. The event management system should be able to isolate primary alarms from secondary alarms, and identify all the faults.

Another important aspect that should be taken into consideration is that events are not only causally related. They are also time related [2]. For that reason it’s very important to maintain a time representation that is consistent among all agents.

For all of the above reasons, it’s possible that the correlation algorithms cannot clearly determine the root cause from a set of alarms. To help solve this complex problem, several techniques from other areas of research have been used [2], as shown in Fig. 2.2.

### 2.3.1 Artificial intelligence

The most widely used techniques in the field of fault localization and diagnosis are expert systems, which use techniques from *Artificial Intelligence*. Expert systems try to reflect actions of a human expert when solving problems in a particular domain [2]:
### Event Correlation

- **Neural Networks**: Systems composed of interconnected nodes called neurons, try to mimic the operation of a human brain. They are capable of learning and resilient to noise or inconsistencies in the input data. The disadvantage of neural network systems is that they require long training periods, and that their behavior outside their area of training is difficult to predict [2].

- **Decision Trees**: Allow a simple and expressive representation of the expert knowledge. However, their applicability is limited by the dependence on specific applications and the degraded accuracy in the presence of noise [2].

These techniques can be used in various types of systems [2]:

- **Rule-based Systems** [2]: Represent the knowledge base in the form of rules (conditions and actions). This type of systems, although useful in small networks, is rather limited. It cannot learn from experience, solve unknown problems (i.e. for which there are no specific rules), rules may overlap each other, and is hard to maintain in large dynamic environments.

- **Model-based Systems** [2]: Uses a representation model that consists of the system structure (static), its behavior (dynamic), and a set of rules. These rules are used to test the relationship between components. Such systems, though more prepared to deal with unknown issues, introduces new challenges like the difficulty in obtaining and maintaining a complete model of the system.

- **Case-based Systems** [2, 15]: Take decisions based on evidence and past events. Can learn how to deal with unknown problems which makes it very resilient to changes. However, case-based systems require an application-specific model for the resolution process. Also, they are very time inefficient, which make them unusable in real-time alarm correlation.
2.3 Event correlation

2.3.2 Data mining

There are other areas of research whose techniques can be used in event correlation. One of those areas is data mining. Data mining is an analytical process for the analysis of large amounts of data in an attempt to find consistent patterns and/or systematic relationships between variables.

In [18, 19, 20], multiple clustering algorithms are used to identify network patterns. They divide the network traffic into classes according to their properties. This information is very useful to plan, optimize and improve the service quality.

The most commonly used algorithms are:

- **Header Inspection**: This is the "classic" technique to cluster network traffic. It’s based on the information of the packet headers (source and destination addresses, and their ports), and is commonly used to group packets with the same application layer protocol (e.g. HTTP, SMTP, etc.).

- **Payload Inspection**: Today many protocols try to masquerade themselves of others (such as HTTP) in an attempt to bypass firewalls restrictions. Payload inspection is used to identify protocols by looking for known patterns in the packets payload.

- **Statistical analysis**: Recently there’s been an increase of encrypted data which invalidates payload inspection. Statistical analysis was introduced to overcome this problem. It consists on the analysis of several different properties (packet sizes, response times, etc.) to be used as parameters in clustering algorithms such as K-Means and DBSCAN.

In [21] it’s proposed a theoretical model for storing large volumes of network data in a Data Warehouse. This data contains important information about the network and its users that can be used for several different management tasks. For the moment this is still a theoretical model, but in the future it may become an important tool in network management.

2.3.3 Fault propagation

Graph-theoretic techniques rely on a graphical model of the system, called Fault Propagation Model (FPM), that describes which symptoms may be observed for each fault. Observed symptoms are mapped into FPM nodes which are then analyzed to identify the best explanation for the observed symptoms [2].

Graph-theoretic techniques require a priori knowledge of how a failure condition or alarm in one component is related to failure conditions or alarms in other components. To create such a model, an accurate knowledge of current dependencies among abstract and physical system components is required. Both the efficiency and accuracy of the fault localization algorithm are dependent on the accuracy of this a priori specification [2].

There are two types of graphs widely used:
• **Dependency Graphs** [2, 16]: Directed graphs that represent the dependencies between the various nodes. This representation is very useful to distinguish between primary and secondary alarm types.

• **Causality Graphs** [2, 17]: Directed acyclic graphs that represent causal relationships between the various alarms. That is, given two alarms it’s possible to check if one is a consequence of the other.

There are many systems that can be built using these graphs. The most important are:

• **Code-based Techniques** [2, 16, 17]: Each fault is represented by a code that corresponds to a vector of N positions. Each position of the vector represents a symptom (alarm) of the fault. In a deterministic context, the value of each position can assume a value of 0 or 1, depending on whether the alarm is or is not generated by this fault. In a probabilistic context, each position vector takes a value between 0 and 1, which represents the probability this symptom be present when the fault occurs. Symptoms can be divided into two groups: significant and candidates. A symptom is considered significant if it is above a given threshold of probability. A fault is detected whenever all of its significant symptoms are observed. The candidate symptoms are optional, so they may or may not be observed. These codes are an optimized matrix representation of a causality graph.

• **Bayesian Networks** [2, 15, 17]: Uses directed acyclic graphs where each node (random variables) represents the network state or the occurrence of events. A fault is identified by the analysis of the Bayesian network. The main problem of this type of systems is that it requires a significant amount of computer resources when applied to a large number of nodes.

• **Phrase-structured Grammars** [2]: The main advantage of these systems is to be able to build expressions from sub-expressions, which is used to construct a hierarchical model of the system with dependencies between objects. However, the algorithms of phrase-structured grammars are quite complex to use.

### 2.4 Dynamic configuration

The usual way to configure various network devices (routers, switches, etc.) is to do it one by one. Not only this is an inefficient way of doing it, but is also a very tedious job which is prone to errors. For these reasons, new alternatives for configuring network devices have been developed. The most important are based on high-level policies, which can be applied to different types of devices independently of their particular configuration. This speeds up the configuration process and avoids mistakes.

The architecture of a **Policy-based Networking** (PBN) [28], is presented in Fig. 2.3.
The **Policy Decision Point** (PDP) is the server where decisions are made about what policies to implement on the various **Policy Enforcement Points** (PEP). There’s also a **Policy Repository** (PR) where all policies are defined and stored.

The **Policy Enforcement Points** (PEP) represent agents (implemented in network devices) that implement policies based on decisions of the PDP.

Communication between the PDP and the various PEP can be done via SNMP or COPS (*Common Open Policy Service*).

It is possible to have policies that conflict with each other. To help in these situations, it’s proposed in [29] the implementation of a conflict resolution system based on rules derived from the **policy domains**. The concept of **policy domain** [29, 30] divides large networks into smaller logical groups that implement common policies. This division can be based on geographic location, application, responsibility, authority or other. A **policy domain** consists of network devices but may also contain other domains (called sub-domains).

### 2.4.1 SNMP – Policy Management MIB

SNMP already includes mechanisms for device configuration, uses the same architecture as presented in Fig. 2.3, and is the most used protocol for managing IP networks. This makes it an excellent candidate for use in a **Policy-based Networking**. The **Policy Management MIB** [24] was created to help mapping high-level policies into device-specific configuration commands. With this new MIB it’s very simple to define high-level policies that can be applied to any network device via SNMP.

### 2.4.2 Common Open Policy Service

COPS [25] is a request-response protocol used to exchange policy information between a **Policy Decision Point** (PDP) and the **Policy Enforcement Points** (PEP).
The highlighted features are:

- Uses a client-server model between the PDP and PEPs;
- Uses TCP for reliable message exchange;
- Implements mechanisms to securely exchange messages, and is also able to use existing security protocols (e.g. IPsec or TLS);
- It’s extensible (can send any type of information without having to change the protocol);
- It’s a stateful protocol, shared between the clients and server.

There are two models for COPS:

- **Outsourcing** [25]: In this model all policy decisions are performed by the PDP. When a PEP needs to make a decision, it sends the relevant information to PDP. PDP analyzes that information, makes a decision based on the policies it has, and sends the response back to the PEP.
- **Provisioning** [26]: In this model all PEPs make decisions based on the policies they know. For obtain policies, each PEP sends its decision-making capabilities to PDP, which selects the best subset of policies to return.

Although initially designed for policy control over QoS (Quality of Service), COPS is a very flexible protocol that can be used in other applications.
Chapter 3
Architecture and Implementation – Framework

To ease the implementation and expansion of the architecture’s applications, it has been established that they should all be based on a common Framework — implemented in Lua (see Appendix A) — with the following requirements (see Fig. 3.1):

- **Modular** – Composed by modules that provide the frameworks features.
- **Expandable** – Support for plugins to ease the inclusion of new features (application level).
- **Flexible** – Easily add or remove components (either modules or plugins).
- **Multi-threaded** – Ability to run several plugins at the same time.
- **Plugin isolation** – Plugins must not inherit the complexity of the frameworks multi-threading environment. It is necessary to isolate and provide a secure single-threaded environment for each plug-in to run.
- **Simplicity** – The framework should provide the simplest working environment for plugins, implementing only the essential features.

Modules provide basic functionality to applications, including the ability to load and manage plugins (see Sec. 3.3), save and load Lua objects (see Sec. 3.5), among others.

The following sections will present all the modules that were implemented to create the framework.
3.1 Bindings

Since Lua has a very simple and compact core, it lacks a few required features. In order to get those features in Lua, some bindings have to be created. These bindings provide a bridge between Lua programs and native operating system libraries (see Fig. 3.2).
3.2 Network & Protocols

Bindings are written in C and have the (almost exactly) same API as the system libraries:

- **Libpcap** – Bindings to the linux network packet capture library.
- **Pthread** – Bindings to the linux posix thread library.
- **Signal** – Bindings to handle application signals.

3.2 Network & Protocols

Both the Network and Protocols modules provide the same features. They both decode network packets from an array of bytes, and are both capable of generating an array of bytes from a packet representation.

Their difference is the language in which they are written. Protocols is written entirely in Lua, and was the first module to be coded. Network, on the other hand, is written in C. The main reason for Networks existence is that Protocols is rather slow when decoding thousands of packets (see Sec. 5.2).

The currently supported protocols are: SLL, Ethernet (802.3), VLAN (802.1Q), ARP, IP, ICMP, TCP and UDP.

The following example uses the Network module and the Libpcap binding module to decode arrays of bytes:

```lua
> require("libpcap")
> require("network")
> function loop(packet, cap)
    local header = packet:header()
    local bytes = packet:bytes()
    local type = cap:get_datalink()

    print("Timestamp: " .. header.ts.sec .. "." .. header.ts.usec)
    print("Length: " .. header.len)
    print("Capture Length: " .. header.caplen)

    if (type == libpcap.DLT_LINUX_SLL) then
        local sll = network.sll(bytes)
        type, bytes = sll:protocol(), sll:payload()
    elseif (type == libpcap.DLT_EN10MB) then
        local eth = network.ethernet(bytes)
        type, bytes = eth:type(), eth:payload()
    else
        -- UNSUPPORTED LINK LAYER PROTOCOL
    end
```
if (type == network.ETHERNET_TYPE_8021Q) then
  local vlan = network.vlan(bytes)
  type, bytes = vlan:type(), vlan:payload()
end
if (type == network.ETHERNET_TYPE_IP) then
  local ip = network.ip(bytes)
  print(ip:source() .. " => " .. ip:destination())
end
> cap = libpcap.open_live("eth0", 96, false, 1000)
> cap:loop(5, loop, cap)
Timestamp: 1282646136.784648
Length: 1400
Capture Length: 96
123.123.123.123 => 231.231.231.231
Timestamp: 1282646137.426978
Length: 1436
Capture Length: 96
231.231.231.231 => 123.123.123.123
Timestamp: 1282646137.938874
Length: 1100
Capture Length: 96
123.123.123.123 => 231.231.231.231
Timestamp: 1282646138.726026
Length: 1280
Capture Length: 96
231.231.231.231 => 123.123.123.123
Timestamp: 1282646139.035335
Length: 1090
Capture Length: 96
123.123.123.123 => 231.231.231.231
> cap:close()

The first two lines load the required modules, libpcap and network.
Next, a loop function is defined that receives a packet (array of bytes) and the user argument cap (libpcap capture handle). This function converts the array of bytes to a packet representation and prints some information about it.
A call to the function open_live is used to create a libpcap capture handle. This capture handle will listen for packets on the eth0 interface, capture only the first 96 bytes, not put the card into promiscuous mode, and set the timeout event to 1000 milliseconds.
The next line starts the capture loop for 5 packets, using loop as the callback function with the user argument cap (the previously created libpcap capture handle).
Finally, the last line closes the capture handle.
3.3 Metaplugin

The Metaplugin module adds plugin support feature to the framework. It uses a multi-threaded model to execute several plugins concurrently, can handle plugin dependencies, and is able to spawn multiple instances of a plugin.

3.3.1 Plugin

A plugin is a generic block of code that implements a well defined interface.

In the context of the proposed architecture, plugins can be seen as a black-box that receives one or more inputs and produces one or more outputs (see Fig. 3.3).

![Plugin Conceptual View](image)

Fig. 3.3: Conceptual view of a plugin

Plugins are very useful because they allow to develop new features more rapidly. Plugins can also be built on top of others to provide new advanced features (see Fig. 3.4).

Each plugin consists of two Lua files: one with the configuration, and another with the implementation code. Filenames should be in the form of `plugin.conf.lua` and `plugin.lua`, for the configuration and implementation respectively.

The configuration is a Lua table with all the possible options and tweaks that can be made to the plugin. Advanced Metaplugin features can be used by including a `metaplugin` table in the plugin configuration:

```lua
metaplugin = {
  depends = STRING | TABLE
  spawn = STRING | NUMBER
}
```

The `depends` keyword indicates the dependencies of the plugin. Multiple dependencies can be specified with a table.

The `spawn` keyword indicates on many instances of the plugin should be created. A string indicates which configuration item (must be a Lua table) should be used to
spawn instances, while a number explicitly indicates the number of instances.

In the following configuration example, it’s defined that there’s a dependency on pluginB and that Metaplugin should spawn three instances. Each instance will have a different host item: 123.123.123.123 for the first instance, 213.213.213.213 for the second instance, and 145.145.145.145 for the third instance.

```.lua
local conf = {
  metaplugin = {
    depends = "pluginB",
    spawn = "host"
  },

  host = {
    "123.123.123.123",
    "213.213.213.213",
    "145.145.145.145"
  },

  port = 80
}

return conf
```
3.3 Metaplugin

The implementation is also a Lua table, that will be used as a Lua *metatable* to spawn instances of the plugin, with the following interface (see Fig. 3.5):

- *initialize* – used to initialize the required data structures
- *start* – execute the plugin
- *stop* – asynchronous event that instructs the plugin to stop its execution
- *finalize* – used to free all the remaining data structures

Below is an example of a plugin implementation that uses the configuration example already presented:

```lua
local pluginA = {}
pluginA.__index = pluginA

function pluginA:initialize()
    print("Initializing pluginA...")
    -- self.host is already defined by Metaplugin
    self.port = self.conf.port
end

function pluginA:start()
    print("Running pluginA with: \" \self.host \":\" .. self.host .. ":\"
          .. self.port)
end

function pluginA:stop()
    print("Stopping pluginA...")
end

function pluginA:finalize()
    print("Finalizing pluginA...")
end

return pluginA
```

3.3.2 Dependency loop detection

When a plugin is loaded to memory, all of its dependencies are also loaded. Since we can check if a plugin has already been loaded, that are no problems with dependency loops at this stage (see Fig. 3.6).

Problems arise later, when synchronous operations must be made between dependencies and dependent plugins. If there are dependency loops, it will result in a
deadlock state. To avoid this, dependency loop detection is executed for each plugin at load time.

The algorithm uses the Looplog module (see Sec. 3.4) to track plugin loops (see Fig. 3.7). Since all branches of a certain node are independent from each other, each receives a copy of the current looplog. Whenever a plugin name is seen more than once (e.g. Plugin B in the Fig. 3.7), a loop is detected and Metaplugin raises an error exception.

### 3.3.3 Metaplugin guarantees

As stated before, Metaplugin runs a multi-threaded environment to execute several plugins concurrently. However, writing multi-thread aware code is difficult and unnecessary for most plugins. For that reason, the entire Framework tries to emulate a single-threaded environment for plugins. This way plugin developers don’t have to worry about access to shared states or other complexities of multi-threaded programming.

To aid in the emulation of a single-threaded environment, Metaplugin gives the following guarantees:
3.3 Metaplug in

Fig. 3.6: Flowchart of the Metapugins load method

- initialize – All dependencies will be initialized before the dependent plugins, and all plugins will be initialized before any plugin starts.
- start – No specific order will be used to start the plugins, they all start at the same time.
- stop – It’s called asynchronously, and may be called even if the plugin has already finished.
• finalize – No specific order will be used, but will only be called after all plugins have finished their execution.

These guarantees are enforced by the plugin manager. The plugin manager is responsible for the initialization, start and finalization of plugins, as well as all of the required synchronization between them (see Fig. 3.8).

The start method of Metaplugin executes the following algorithm (see Fig. 3.8):

1. Launch plugin manager threads (one for each plugin)
2. Wait for all managers to get control of their thread
3. Wait for all managers to initialize their instances
4. Wait for all managers to start their instances

Each plugin manager thread executes the following algorithm (see Fig. 3.5 and Fig. 3.8):

1. Initialize required data structures
2. Signal that it has control of thread and wait for all remaining plugin managers to do the same
3. If its plugin has dependencies, wait for all dependencies to initialize
4. Spawn and initialize all plugin instances
5. Signal waiting plugin managers (dependent plugins) that the plugin has been initialized
6. Wait for all remaining managers to initialize their plugins

Fig. 3.7: Example of plugin dependency loop
7. Launch a new thread for each plugin instance to run
8. Signal that it has started the plugin and wait for all remaining plugin managers to do the same
9. Wait for its plugin instance threads to finish
10. Signal that its instances have finish and wait for all remaining plugin managers to do the same
11. Finalize all plugin instances

Plugins are started on a separate dedicated thread that should only return when the plugin has finished its work or when it has been asked to stop (see Fig. 3.5 and Fig. 3.8).

### 3.4 Looplog

Looplog is an auxiliary module to help in loop detection. It uses a FIFO list, called log, to keep track of the already seen identifiers. Whenever an identifier is added to this list, Looplog checks if it’s already there and informs whether it has detected a loop or not. Looplog is also capable of cloning itself to help in situations where multiple sub-trees are independent between themselves (e.g. see Sec. 3.3)

The Looplogs API is as follows:

- **islooping** – adds an identifier to the end of the list and checks whether it’s looping (already on the list) or not
- **clone** – returns a clone of itself (a copy of the current log)
- **first** – used to get the first identifier on the list
- **last** – used to get the last identifier on the list
- **tostring** – returns a string representation of the loop (e.g. \( A \rightarrow B \rightarrow C \rightarrow A \))

### 3.5 Persistence

The Persistence module adds persistent storage features to the framework. It uses an object representation of data (simple way of representing unknown/opaque data structures) and stores it in files.

The requirements that lead to the creation of this module are the following:

- Manage all input and output files.
- Allow multiple input folders, and one output folder.
- Avoid filename collisions (which would result in overwriting already existing data).
- Store multiple instances of an object.
- Manage the coexistence of multiple objects on a single output folder.
- Sequential access to object instances (in the same order that they were written).
- Transiently read from several input folders.
- Allow to purge all from the output folder when initializing, but otherwise there is no need to modify or delete any of the already written data.
3.5 Persistence

3.5.1 Objects and instances

Objects are identified by their name and can have multiple instances (see Fig. 3.9).

Each instance is given an unique filename. It is composed by the object name, a timestamp and sequence number (resulting in objname-timestamp[seqn]). This choice of organization allows for instances from different input folders to be easily sorted and read sequentially (see Fig. 3.10).

3.5.2 Services provided

The Persistence module provides the following services:

- Ability to write instances of an object.
- Sequential read of all instances of an object.
- Reserve a unique filename to be used outside the Persistence module (e.g. by other modules).
- List all files of a given object to be used outside the Persistence module (e.g. by other modules).

Before any of this services can be used, the module must be initialized. This ensures that the module is aware of all the files that exists in both input and output folders, reducing the number of disk accesses and preventing it from overwriting existing output files (see Fig. 3.11).

Reserve filename

By reserving filenames to be used outside of the Persistence module, one ensures that they will not be used/overwritten by any other module or plugin.

The algorithm receives an object name and returns an unique filename:

1. Get exclusive access to database
2. Set timestamp to the current time
3. Find the first available filename: output/objname-timestamp[seqn]
4. Store filename in the database
5. Release exclusive access to database
6. Return the filename

List files

Listing all input files of a given object is used when raw access to files is required outside of the Persistence module (e.g. other modules).

The algorithm receives an object name and returns a sorted iterator over all input files:

1. Get exclusive access to database
2. Create a list of all input filenames of the given object name
3. Release exclusive access to database
4. Sort the list by timestamp and sequence number
5. Return an iterator for the list
Writing instances

As already stated, the Persistence module uses an object representation of data. This means that it doesn’t know the structure of the data, and yet it needs to write it to files. To do this, it uses the Serializer module (see Sec. 3.6) to convert objects into strings (see Fig. 3.12).

Example of how to write an object instance:
Fig. 3.12: Write of an object instance

```
writer = persistence.writer(objname)
writer.write(instance1)
```
3.6 Serializer

The Serializer module adds to the framework the ability to serialize any basic Lua data type. The serialized data is a single line string of Lua code, terminated with a newline character.

3.6.1 Supported data types

The basic Lua data types that can be serialized are: nil, boolean, number, string, table and function. Any other data type (e.g. thread, userdata, ...) will result in an error exception. Metatables are also serialized, which means that Lua objects (table with both data and code) are also serializable. The only limitation with Lua objects is that they cannot use up-values. Since up-values are part of the context of the object, they are not stored on the object itself or even accessible outside that context. This means that they are unknown/invisible to the Serializer module (or any other module for that matter).

```lua
-- up-values cannot be serialized
local upvalue = "Example of an up-value"
```
Fig. 3.13: Reading object instances from files
local obj = {}
obj.__index = obj

function obj:tostring()
    -- will always print "Example of internal value"
    print(self.objvalue)
    -- will print an empty line on deserialized objects
    print(upvalue)
end

function new()
    local o = { objvalue = "Example of internal value" }
    return setmetatable(o, obj)
end

### 3.6.2 Serialize

To serialize data, the module uses the following algorithm:

1. Create a new table with the data to be serialized
2. Traverse table to identify all tables (including metatables) and functions
3. For each function found: convert it to a string
4. For each table found: convert all of its keys and values to a string representation
5. For each table found: check if it has a metatable and convert the association to a string
6. Escape all newlines from the resulting concatenated strings and append a newline to the end

### 3.6.3 Deserialize

To deserialize data, the module uses the following algorithm:

1. Unescape string representation of table with data
2. Convert string to a Lua table
3. Extract data from table
3.7 Message Queue

The Message Queue module enables the exchange of information between plugin instances (see Fig. 3.14). Information is exchanged in the form of messages and stored in queues.

![Fig. 3.14: Exchange of information between plugin instances](image)

3.7.1 Queues

There are two types of queues in this module: virtual queues and real queues. Virtual queues don’t store any messages. Their purpose is to bind real queues (owned by plugins) to a known shared name, used to send messages. Whenever a message is sent to a virtual queue it is delivered to all the real queues that subscribe it (see Fig. 3.15).

Real queues is where messages are delivered and stored. Each plugin instance can have any number of real queues, and real queues can subscribe any number of virtual queues.

Queues have a few important properties:

- Virtual queues don’t need to be explicitly created. Whenever a non-existing virtual queue is used, it is automatically created.
Virtual queues don’t need to be explicitly destroyed. When not needed anymore, existing virtual queues are automatically destroyed.

Messages are never lost. They are always delivered to all subscribing real queues.

Whenever a real queue is full, the sender blocks until the message is delivered.

Messages are only delivered to queues that are subscribed at the time the message was sent.

Messages are not copied or duplicated. All real queues receive the same message.
3.7.2 Sending messages

Sending messages to virtual queues is done with a Lua object called \texttt{sender}. This object is able to send messages to several virtual queues at once:

1. For all its registered virtual queues:
   a. get exclusive access to virtual queue
   b. get all real queues (receivers)
   c. release exclusive access to virtual queue

2. For each real queue:
   a. get exclusive access to real queue
   b. while queue is full: wait (releases exclusive access and reaquires it)
   c. insert message in the end of queue
   d. signal receiver of waiting message
   e. release exclusive access to real queue

To use a \texttt{sender}, just create it and register the virtual queues to where messages will be sent:

```lua
sender = msgqueue.sender("vqueue1", "vqueue2", ...)
sender.send(message)
...
sender.destroy()
```

3.7.3 Receiving messages

To create real queues (where messages are delivered and stored), there is a Lua object called \texttt{receiver}. This object can subscribe several virtual queues and has a maximum number of waiting messages (queue size). A value of \texttt{-1} for queue size indicates it has an infinite size.

Receiving a message on a real queue executes the following algorithm:

1. Get exclusive access to real queue
2. Wait for messages on queue (releases exclusive access and reaquires it)
3. Remove first message from queue
4. Signal waiting senders that there is a free slot to deliver messages
5. Release exclusive access to real queue
6. Return the message

To use a \texttt{receiver}, just create it, register the virtual queues and wait for incoming messages:
3.8 Remote Call

Remote Call module enables plugins to share pieces of code that may be useful to other plugins. These pieces of code are available as functions with a known shared name.

3.8.1 Multiple simultaneous calls

As already said before, the framework works in a multi-threaded environment but tries to emulate a single-threaded environment for all plugins. For this to remain true in the situation where multiple plugins call the same shared function at the same time, Remote Call must serialize the calls that are made to the same function (see Fig. 3.16).

There are, however, situations where multiple simultaneous calls are allowed (e.g. stateless functions, multi-thread aware plugins, ...). For these situations, it is possible to instruct Remote Call to not serialize the calls.

```literate
local counter = 0
-- *not* safe for multiple simultaneous calls
function statefull(x)
    counter = counter + 1
    return (x + counter) * 2
end

-- safe for multiple simultaneous calls
function stateless(x)
    return x * 2
end
```
3.8.2 Registering and calling shared functions

To register a shared function, all that is required is a unique name, the calling restrictions (single or multi-threaded) and the function itself:

```python
remotecall.register("sf", remotecall.SINGLE, statefull)
remotecall.register("sl", remotecall.MULTI, stateless)
```

To call a shared function, only the known shared name (and the function arguments) is required:

```python
remotecall.call("sf", 10)    -- returns 22
remotecall.call("sf", 10)    -- returns 24
remotecall.call("sf", 10)    -- returns 26
remotecall.call("sl", 100)   -- returns 200
remotecall.call("sl", 100)   -- returns 200
```
If necessary, it’s also possible to unregister shared functions:

```python
remotecall.unregister("sf")
remotecall.unregister("sl")
```

### 3.9 Timeline

Polling mechanisms yield discrete samples over a period of time at a constant or dynamic rate. When multiple data sources are not synchronized between themselves, choosing a set of samples for a given time can be hard. Timeline helps in this situation, by converting a discrete sample set to a continuous sample set (see Fig. 3.17 and Fig. 3.18).

![Timeline Diagram](image)

**Fig. 3.17: Range of each sample, using fixed limits**

![Timeline Diagram](image)

**Fig. 3.18: Range of each sample, using extended limits**

As you can see from the Fig. 3.17 and Fig. 3.18, Timeline tries to pick the best sample for a given time. This can either be a sample from the past or, if the past is too old, a sample from the future.

#### 3.9.1 Limits

There are two types of limits that can be used with Timeline:

- **Fixed limits**: A query to a time before the first sample will not return any value, and a query to a time after the last sample will also not return any value.
- **Extended limits**: A query to a time before the first sample will return the first sample, and a query to a time after the last sample will return the last sample.
3.9.2 Using Timeline

Timeline has a small and easy API:

- **add**: Adds the sample if no other sample exists at the given time.
- **set**: Adds the sample overwriting any existing sample at the given time.
- **first**: Returns the first sample.
- **last**: Returns the last sample.
- **exactmatch**: Returns the sample at the exact given time if it exists.
- **bestmatch**: Searches for the best sample to the given time (respecting the imposed limits).
- **iterator**: Iterates through all the samples.

Here’s an example of how to use Timeline module to convert a discrete set of samples to a continuous set:

```plaintext
> require("timeline")
> tl = timeline.new(timeline.FIXED_LIMITS)
> tl:add(100, "100")
> tl:add(150, "150")
> tl:add(200, "200")
> =tl:bestmatch(80)
  nil
> =tl:bestmatch(100)
  100
> =tl:bestmatch(120)
  100
> =tl:bestmatch(140)
  150
> =tl:bestmatch(160)
  150
> =tl:bestmatch(180)
  200
> =tl:bestmatch(220)
  nil
> =tl:exactmatch(120)
  nil
> =tl:exactmatch(150)
  150
> =tl:first()
  100
> =tl:last()
  200
```
3.10 Root Cause Analysis

Root Cause Analysis is a module that helps in the process of event correlation. Event correlation can be decomposed in five simpler steps:

1. **Event filtering**: Consists in discarding events that are deemed to be irrelevant.
2. **Event aggregation**: Consists in merging duplicates of the same event.
3. **Event masking**: Consists in ignoring events pertaining to systems that are downstream of a failed system.
4. **Root cause analysis**: Is considered the most complex step of event correlation and consists in analyzing dependencies between events.
5. **Action triggering**: Events may trigger corrective actions or further investigations automatically.

This module focuses on the third and forth step, *Event masking* and *Root cause analysis*. Both this steps are implemented with event filters (see Fig. 3.19). Event filters try to find an explanation for an event existence. If there is a reason for the event (e.g. alarm about a non-responsive service due to a failed router), then the event is filtered. Events for which there isn’t any reason that justify their existence are considered indicators of a root problem.

3.10.1 Using Root Cause Analysis

The API used by this module is very simple:

- **addfilter**: Adds a filter function to the end of the chain of event filters.
- **unlock**: Unlocks a filter function, making it available to process events.
- **filter**: Runs a event through the chain of event filters.

When an event filter is added to the chain, it’s added in a locked state. This is done to avoid the filter being used before it’s ready. This allows plugins to register their filters and process any require data before starting to filter events. When the plugin is ready to process events it must call the unlock method. To maintain the single-threaded environment emulation for plugins, all filter calls are serialized.

To recognize the type of an event, each event must be assigned with a unique type identifier. To avoid clash or ambiguity, type identifiers are defined on the module.

Here is an example of how to use the Root Cause Analysis module:

```plaintext
function eventfilter(event)
    if (event.type == rca.UNREACHABLE_HOST) then
        return true -- filtered
    else
        return false -- not filtered
```
Fig. 3.19: Event filtering to discover the root cause event

```python
end
end
rca.addfilter(eventfilter)
-- do some work
rca.unlock(eventfilter)
-- do some more work
for event in events do
```
3.11 Scheduler

The Scheduler module is a timer loop that calls a specific function every few seconds or nanoseconds (see Fig. 3.20).

![Scheduler diagram](image)

**Fig. 3.20: Using Scheduler module to delay repeated actions**

It’s possible to define an initial waiting interval, a minimum and maximum limits (for interval adjustments) and to asynchronously stop the loop. The API of the module is as follow:

- **start**: Starts the wait-call loop.
- **stop**: Allows asynchronous break of the loop.
- **get**: Returns the current wait interval.
- **set**: Sets the next wait interval (ignoring the limits).
- **update**: Adjusts the waiting interval with the provided *delta* (respecting the minimum and maximum limits).

The following example demonstrates how to use the Scheduler module:
> require("scheduler")
> init = 1 -- seconds
> min = 1 -- seconds
> max = 10 -- seconds
> function f(arg)
>   local sched = arg.sched
>   print(sched:get())
>   local aux = (max - min)
>   local delta = math.random() * (aux * 2) - aux
>   sched:update(delta)
> end
> arg = {}
> sched = scheduler.new(f, arg, init, min, max)
> arg.sched = sched
> sched:start() -- thread busy on scheduler loop
> ...
> sched:stop() -- asynchronous call (another thread)

### 3.12 Running average

Running average module is used by plugins to keep track of both mean and standard deviation. It uses a *running algorithm* which means that it does not have to store any values to keep the mean and standard deviation up-to-date.

It provides the following API:

- **update**: Updates the mean and standard deviation with the provided sample.
- **get**: Returns the mean, standard deviation and number of processed samples.
- **reset**: Resets all values.

```
> require("runavg")
> ra = runavg.new()
> ra:update(50)
> =ra:get()
50 0 1
> ra:update(63)
> =ra:get()
56.5 9.1923881554251 2
> ra:update(29)
> =ra:get()
47.333333333333 17.156145643277 3
```
3.13 Fuzzy Logic

This module provides a Lua implementation of a Fuzzy Logic algorithm.

It consists of several sub-modules:

- `fuzzylogic`: main module with the client API
- `fuzzylogic/set`: handles fuzzy sets
- `fuzzylogic/term`: handles fuzzy terms
- `fuzzylogic/grammar`: handles fuzzy rules
- `fuzzylogic/membf`: contains several precoded membership functions
- `fuzzylogic/normf`: contains several precoded normalization functions
- `fuzzylogic/defuzf`: contains several precoded defuzzification functions

This is a complex module, with many options and setup steps, that is better explained with an usage example. The next sections demonstrate how to setup and use the Fuzzy Logic module with a simple restaurant tip example. The restaurant tip example tries to answer the following question: Given two numbers between 0 and 10 that represents the quality of service and the quality of food at a restaurant (where 10 is excellent), what should the tip be?

### 3.13.1 Instantiate Fuzzy Logic

The implemented Fuzzy Logic algorithm has quite a few configuration options:

- **Operand AND**
- **Operand OR**
- **Operand NOT**
- **Operand WEIGHT**
- **Function IMPLICATION**
- **Function AGGREGATION**
- **Function DEFUZZIFICATION**
- **Number of points for integration**

To instantiate a fuzzy logic configuration:
> require("fuzzylogic")
> conf = {
    opand = fuzzylogic.normf.min,
   opor = fuzzylogic.normf.max,
    opnot = fuzzylogic.normf.comp,
    opwei = fuzzylogic.normf.prod,
    impf = fuzzylogic.normf.min,
    aggf = fuzzylogic.normf.max,
    defuzf = fuzzylogic.defuzf.cog,
    npoints = 1000
}  
> fl = fuzzylogic.new(conf)

Not all options have to be specified. Those not specified in the configuration table will use the default values.

### 3.13.2 Adding Fuzzy Sets (inputs and outputs)

To add Fuzzy Sets to the instantiated fuzzy logic:

```lua
> -- two inputs
> service = fl:addinput("service", 0, 10)
> food = fl:addinput("food", 0, 10)
> -- one output
> tip = fl:addoutput("tip", 0, 30)
```

The first parameter is the name of the Fuzzy Set (will be used in the Fuzzy Rules). The second and third parameters are the lower and upper limits, respectively.

### 3.13.3 Defining Fuzzy Terms

Now that Fuzzy Sets exist in the instantiated Fuzzy Logic, it’s necessary to define their Fuzzy Terms:

```lua
> service:addterm(
    "poor",
```
3.13 Fuzzy Logic

The first parameter is the name of the Fuzzy Term (will be used in the Fuzzy Rules). The second and third parameters are the Membership Function and its parameters, respectively.

3.13.4 Adding Fuzzy Rules

Fuzzy Rules are added in the form of text. They describe the operations that Fuzzy Logic must do to compute the output value:
> rule1 = "IF service is poor OR food is rancid THEN tip is cheap [1]"
> rule2 = "IF service is good THEN tip is average [1]"
> rule3 = "IF service is excellent OR food is delicious THEN tip is generous [1]"
> fl:addrule(1, rule1)
> fl:addrule(2, rule2)
> fl:addrule(3, rule3)

The first parameter is the Fuzzy Rule number, while the second is its textual expression.

Rules are divided in three blocks. First, there is the IF statement, that indicates the conditions that will activate the rule. The second part is the THEN statement, that has the consequences of the rule (in case it’s activated). Finally, the third part is the weight of the rule, expressed between square brackets.

### 3.13.5 Computing output values

Now that the Fuzzy Logic instance is completely setup, it can be used to compute output values:

> service:setvalue(3)
> food:setvalue(7)
> fl:solve()
> tip:getvalue()
11.701603788948

The first two lines feed Fuzzy Logic with the values for each input Fuzzy Set. The third line instructs Fuzzy Logic to solve the problem. Finally, the fourth line prints the computed value for the output Fuzzy Set named tip.
Chapter 4  
Architecture and Implementation – Applications

The proposed architecture to capture, analyze and detect problems on IP networks consists of three different applications (see Fig. 4.1):

- **Collector** – Collects data from the network.
- **Analyzer** – Analyzes a Collector’s data.
- **Correlator** – Correlates data from all Analyzers.

![Fig. 4.1: Overview of the proposed architecture](image)

All three applications — implemented in Lua (see Appendix A) — are very similar (see Fig. 4.2), mostly differing only on their plugins.

The default directory structure used by all applications is the following:

```
|-- conf
    |-- application.conf.lua
    |-- application.lua
|-- conf
```

49
local conf = {
  paths = {
    conf = "./conf/",
    plugins = "./plugins/",
    modules = "./modules/",
    input = { "./input1/", "./input2/" },
    output = "./output/"
  },

  plugins = {
    "plugin1",
    "plugin2",
    "plugin3"
  }
}

return conf
Additionally, every application accept one command line parameter that specifies where the applications configuration file is located. By default, if no parameter is specified, it is assumed that it’s on the same directory as the executable.
4.1 Collector

The Collector is responsible for gathering packets and data from the subnetwork where it’s located. These are stored into a file to later be processed by the Analyzer. To maximize the amount of information that is captured, several Collectors may be scattered over the network (see Fig. 4.3).

By using plug-ins, it is possible to make the Collector work as a passive entity (e.g. only capturing data packets from the network), or an active entity (e.g. interacting with other nodes, such as routers or switches).

These are plugins developed for the Collector application:

- **pcap**: Collects packets from the network.
- **ping**: Pings one or more hosts on the network, and is capable of dynamically adapting the polling interval for each host individually (using the mean and standard deviation to determine the adjustments).
- **traceroute**: Performs a traceroute to one or more hosts on the network, adapting the polling interval for each host using their reachable state to choose the interval.
- **topology**: Uses SNMP to query routers about their routing tables and subnets.
- **usage**: Queries one or more hosts, using SNMP, about their link usage. Is also capable of dynamically adapt the polling interval for each host individually, using Fuzzy Logic.
4.3 Correlator

4.2 Analyzer

There’s one Analyzer assigned to each Collector. It represents the first stage of data analysis and correlation, and is used to generate new, more rich and refined, data and alarms (e.g. network usage statistics, unreachable host alarms, etc.). These will be then used by the Correlator for further processing.

The plugins developed for the Analyzer application:

- **pcap**: Reads the Collector saved packets, decodes them (from byte array to packet structure), and sends them to the pcap virtual queue.
- **mtu**: Listens on pcap virtual queue for decoded network packets. Whenever an IP packet is received, checks if it’s violating the network MTU. If a violation is found, it generates an alarm event.
- **statistics**: Gathers several statistics about the network packets that are received on the pcap virtual queue.
- **ping**: Searches the Collector data for ping timeouts. Whenever a timeout is found, it generates an alarm event.
- **topology**: Translates the Collector SNMP raw data table to a more usable representation, using a Timeline (see Sec. 3.9) representation of the network routers.

4.3 Correlator

The Correlator is the central point of the architecture. It merges the data from all Analyzers, and is the second stage of data analysis and correlation. It’s used to determine the root cause of the alarms generated by the Analyzers. This information can then be forwarded to, for example, a frontend (whose implementation is beyond the scope of this thesis) or any other application.

The plugins developed for the Correlator application:

- **topology**: Merges the topology from different Analyzers into one single consolidated topology (see Fig. 4.4), and sends it through the topology virtual queue.
- **statistics**: Merges statistics from all Analyzers and generates new statistics using the information received from the topology virtual queue.
- **digraph**: Generates a visual representation of the network routing tables (received from the topology virtual queue) using directed graphs.
- **routing**: Uses the information received from the topology virtual queue to check for looping and unreachable networks. If it finds an unreachable or looping network, it generates an alarm event that will be verified by the RCA module. It also inserts a filter on the RCA module (see Sec. 3.10) to filter unreachable hosts that are inside unreachable networks.
- **ping**: Checks its alarm events by sending them through the filter chain of the RCA module (see Sec. 3.10).
• `mtu`: Uses the information from the *topology virtual queue* to remove invalid alarm events (e.g. violations whose source IP is outside the network boundaries), and checks them with the RCA module (see Sec. 3.10).

The resulting alarm events — which have not been filtered by the RCA module (see Sec. 3.10) — are considered root cause alarms and saved on the output folder. Additionally, information about network statistics and directed graphs representing the network routing tables will also be saved on the output folder.
Chapter 5
Tests

5.1 Applications

In order to verify that all three applications (and their plugins) were working properly, some tests have been performed:

- Build network topology
- Verify MTU of network packets
- Gather statistics about network protocols and usage
- Raise alarms when a monitored host stops responding
- Hide secondary alarms (that result from another anomaly)

5.1.1 Setup

To execute these tests, three Collectors were placed on the network of Instituto Superior Tecnico – Taguspark. This is a medium size network with 4 routers, 43 network switches, 60 access points, and a few hundred hosts scattered in about 20 – currently in use – VLANs.

The Collectors had the following configuration:

- **Collector A**
  - CPU: AMD Athlon 64 X2 Dual Core Processor 4200+
  - RAM: 1GB DDR2 533Mhz
  - OS: Ubuntu 9.10
  - Lua: Version 5.1.4
  - VLAN: Administration
  - Plugins: ping and topology
  - Objective: Detect unreachable hosts (monitoring host 172.20.22.4), build network topology.
• **Collector B**
  - CPU: Intel Atom N280 Single Core Processor 1.66GHz (with Hyper-threading)
  - RAM: 2GB DDR2 667Mhz
  - OS: Ubuntu 10.04
  - Lua: Version 5.1.4
  - VLAN: Wireless
  - Plugins: ping and topology
  - Objective: Detect unreachable hosts (monitoring host 8.8.8.8), build network topology.

• **Collector C** (running in the main router)
  - CPU: Intel Xeon Dual Processor 3.00GHz (with Hyper-threading)
  - RAM: 4GB DDR2 800Mhz
  - OS: Debian 5.0
  - Lua: Version 5.1.4
  - VLAN: All
  - Plugins: pcap (snapshots of 96 bytes)
  - Objective: Capture network packet headers from all VLANs.

### 5.1.2 Timetable

Due to the large number of packets that go through the main router, **Collectors** only ran for five minutes.

In those five minutes some events have been simulated using *iptables* rules, to test if both **Collector A** and **Collector B** were able to detect and raise alarms whenever their monitored host stopped responding:

- **0:00** — Start
- **1:30** — Host 8.8.8.8 stops responding
- **1:45** — Main router stops responding
- **2:00** — Host 172.20.22.4 stops responding
- **2:30** — Main router recovers
- **3:00** — Host 8.8.8.8 recovers
- **4:00** — Host 172.20.22.4 recovers
- **5:00** — Stop

### 5.1.3 Results

The obtained results were the expected ones, showing that all applications (and respective plugins) are working properly.
5.1 Applications

Build network topology

Due to firewall restrictions on the Wireless VLAN (SNMP protocol blocked) where Collector B was located, only Collector A was able to gather data about the network topology (see Fig. 5.1).

Verify network packets MTU

Although the network packets capture by Collector C had some MTU violations, detected by the Analyzer, these were all committed by external hosts¹ and were discarded by the Correlator.

Gather statistics about network protocols and usage

In the five minutes that Collector C ran, it captured 6658098 network packets with a total of about 4.7GB:

<table>
<thead>
<tr>
<th>Protocol</th>
<th>Packets</th>
<th>Bytes</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Global statistics:</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6658098 packets</td>
<td></td>
<td>5060960246 bytes</td>
</tr>
<tr>
<td><strong>Layer 2 statistics:</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ETHERNET_TYPE_802_3:</td>
<td>49 packets</td>
<td>3118 bytes</td>
</tr>
<tr>
<td>0.001 %</td>
<td>0.000 %</td>
<td></td>
</tr>
<tr>
<td>ETHERNET_TYPE_ARP:</td>
<td>24144 packets</td>
<td>1105932 bytes</td>
</tr>
<tr>
<td>0.363 %</td>
<td>0.022 %</td>
<td></td>
</tr>
<tr>
<td>ETHERNET_TYPE_UNKNOWN:</td>
<td>806 packets</td>
<td>1029640 bytes</td>
</tr>
<tr>
<td>0.012 %</td>
<td>0.020 %</td>
<td></td>
</tr>
<tr>
<td>ETHERNET_TYPE_IPV6:</td>
<td>28004 packets</td>
<td>15754048 bytes</td>
</tr>
<tr>
<td>0.421 %</td>
<td>0.310 %</td>
<td></td>
</tr>
<tr>
<td>ETHERNET_TYPE_802_2:</td>
<td>7764 packets</td>
<td>513568 bytes</td>
</tr>
<tr>
<td>0.117 %</td>
<td>0.010 %</td>
<td></td>
</tr>
<tr>
<td>ETHERNET_TYPE_IP:</td>
<td>6597276 packets</td>
<td>5042550450 bytes</td>
</tr>
<tr>
<td>99.086 %</td>
<td>99.638 %</td>
<td></td>
</tr>
<tr>
<td>ETHERNET_TYPE_AOE:</td>
<td>5 packets</td>
<td></td>
</tr>
<tr>
<td>0.000 %</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

¹ Hosts that are outside topology boundaries
310 bytes (0.000 %)
ETHERNET_TYPE_IPX:
50 packets (0.001 %)
3180 bytes (0.000 %)

Layer 3 statistics
IP_PROTO_UNKNOWN:
2004 packets (0.030 %)
218526 bytes (0.004 %)
IP_PROTO_TCP:
3789546 packets (57.441 %)
3153204415 bytes (62.305 %)
IP_PROTO_IGMP:
235 packets (0.004 %)
12630 bytes (0.000 %)
IP_PROTO_ICMP:
292743 packets (4.437 %)
41883483 bytes (0.828 %)
IP_PROTO_UDP:
2287224 packets (34.669 %)
1655237658 bytes (32.706 %)
IP_PROTO_PIM:
189 packets (0.003 %)
15118 bytes (0.000 %)
IP_PROTO_ESP:
225335 packets (3.416 %)
210388416 bytes (4.157 %)

Usage per network
...

Usage per host
...

*Raise alarms when a host stops responding*

Both Collector A and Collector B detected that their respective hosts stopped responding to their ping requests:

[Tue Aug 7 16:40:36 2010] UNREACHABLE_HOST: 8.8.8.8
...
Fig. 5.1: Network topology built using data from Collector A
**Hide secondary alarms (that result from another anomaly)**

When the main router stopped responding (see Sec. 5.1.2), the alarms from both Collector A and Collector B were filtered by the RCA module (see Sec. 3.10):

```
[Tue Aug 7 16:40:36 2010] UNREACHABLE_HOST: 8.8.8.8
[Tue Aug 7 16:40:52 2010] UNREACHABLE_HOST: 8.8.8.8
### alarms filtered by the RCA module ###
[Tue Aug 7 16:41:48 2010] UNREACHABLE_HOST: 172.20.22.4
[Tue Aug 7 16:42:12 2010] UNREACHABLE_HOST: 172.20.22.4
[Tue Aug 7 16:42:35 2010] UNREACHABLE_HOST: 172.20.22.4
```

### 5.2 Bottlenecks ###

As already stated earlier sections (see Sec. 3.2 and Appendix B), while developing the applications it became obvious that there were some bottlenecks on the Framework.

These were detected during the analysis of network packets which, depending on the number of packets to analyze, could take any value between a few minutes to several minutes to execute.

To confirm these bottlenecks, it was measured the execution time of several operations, using the Linux `time` command:

- **Read**: Read a network capture file.
- **Decode**\(^2\): Decode all layers of all packets (convert from byte array to packet representation).
- **MTU**\(^3\): Check IP packets for MTU violations.
- **Statistics**\(^3\): Gather statistics about the network protocols (layer 2 and 3).
- **Everything**: Do all of the above.

All of the operations were tested on three different Lua flavors (see also Appendix B):

- **Original**: Original Lua implementation without multi-threading.
- **Permissive**: First Lua patch to support multi-threading, allowing CPU relinquish.
- **Aggressive**: Second Lua patch to support multi-threading, not allowing CPU relinquish.

\(^2\) Imply the **Read** operation.

\(^3\) Imply the **Read** and **Decode** operation.
5.2 Bottlenecks

On the Lua flavors that support multi-threading, it has also been tested with two and four threads (all executing the same operations).

5.2.1 Setup

The computer used to gather the statistics had following specifications:

- **CPU**: Intel Atom N280 Single Core Processor 1.66GHz (with Hyper-threading)
- **RAM**: 2GB DDR2 667Mhz
- **OS**: Ubuntu 10.04
- **Lua**: Version 5.1.4

The network capture file that was used contained about 187,000 packet snapshots of 96 bytes each. The MTU used for checking the size of IP packets was the standard for 802.3 Ethernet, 1500 bytes.

5.2.2 Results

<table>
<thead>
<tr>
<th></th>
<th>1 Thread</th>
<th>2 Threads</th>
<th>4 Threads</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Network</td>
<td>Protocols</td>
<td>Network</td>
</tr>
<tr>
<td>Read</td>
<td>0.763</td>
<td>0.763</td>
<td>-</td>
</tr>
<tr>
<td>Decode</td>
<td>19.299</td>
<td>56.330</td>
<td>-</td>
</tr>
<tr>
<td>MTU</td>
<td>19.511</td>
<td>57.414</td>
<td>-</td>
</tr>
<tr>
<td>Statistics</td>
<td>23.257</td>
<td>73.684</td>
<td>-</td>
</tr>
<tr>
<td>Everything</td>
<td>23.232</td>
<td>75.007</td>
<td>-</td>
</tr>
</tbody>
</table>

Table 5.1: Execution time (in seconds) using *Original* Lua (no multi-thread support)

<table>
<thead>
<tr>
<th></th>
<th>1 Thread</th>
<th>2 Threads</th>
<th>4 Threads</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Network</td>
<td>Protocols</td>
<td>Network</td>
</tr>
<tr>
<td>Read</td>
<td>1.407</td>
<td>1.407</td>
<td>3.979</td>
</tr>
<tr>
<td>Decode</td>
<td>36.228</td>
<td>244.513</td>
<td>196.189</td>
</tr>
<tr>
<td>MTU</td>
<td>36.949</td>
<td>249.220</td>
<td>201.119</td>
</tr>
<tr>
<td>Statistics</td>
<td>42.159</td>
<td>282.951</td>
<td>224.422</td>
</tr>
<tr>
<td>Everything</td>
<td>42.989</td>
<td>287.590</td>
<td>220.632</td>
</tr>
</tbody>
</table>

Table 5.2: Execution time (in seconds) using *Permissive* Lua
<table>
<thead>
<tr>
<th></th>
<th>1 Thread</th>
<th></th>
<th>2 Threads</th>
<th></th>
<th>4 Threads</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Network</td>
<td>Protocols</td>
<td>Network</td>
<td>Protocols</td>
<td>Network</td>
<td>Protocols</td>
</tr>
<tr>
<td>Read</td>
<td>0.902</td>
<td>0.902</td>
<td>3.709</td>
<td>3.709</td>
<td>5.744</td>
<td>5.744</td>
</tr>
<tr>
<td>Decode</td>
<td>32.740</td>
<td>59.528</td>
<td>180.590</td>
<td>193.830</td>
<td>358.498</td>
<td>329.455</td>
</tr>
<tr>
<td>Statistics</td>
<td>37.150</td>
<td>79.132</td>
<td>205.633</td>
<td>248.229</td>
<td>381.797</td>
<td>400.066</td>
</tr>
<tr>
<td>Everything</td>
<td>37.449</td>
<td>81.984</td>
<td>207.153</td>
<td>263.767</td>
<td>392.620</td>
<td>404.366</td>
</tr>
</tbody>
</table>

Table 5.3: Execution time (in seconds) using Aggressive Lua

5.2.2.1 Comparison of Lua flavors – Original vs Permissive vs Aggressive

As we can see from Table 5.1, 5.2, 5.3 and Fig. 5.2, 5.3, 5.4, there is a big difference in the execution time of all three Lua flavors for all operations. The fastest is the Original, which makes sense since no time is wasted on locking and unlocking the Lua mutex.

The slowest is Permissive. As explained in Appendix B, this happens because there are too many calls to relinquish the CPU, causing threads execution to ping-pong, wasting CPU time on context switches.

The Protocols module (written entirely in Lua), compared to the Network module (written in C), causes a lot more context switches which wastes valuable CPU time (see also Appendix B).

Table 5.4 compares the execution time of Permissive and Aggressive flavors against Original Lua.

Fig. 5.2: Comparison between the three Lua flavors (1 Thread)
5.2 Bottlenecks

The presented data supports the decision to use Aggressive Lua over Permissive Lua.
5.2.2.2 Comparison of Modules – Protocols vs Network

There is another big difference between the execution time of the Protocols and Network modules.

![Aggressive Lua (1 Thread)](image)

Fig. 5.5: Comparison between the Protocols and Network modules (1 Thread)

As we can see in Fig. 5.5, the Network module is a lot faster to execute than the Protocols module. This happens because of the following reasons:

- **Memory allocation**: In Lua, tables grow dynamically which means that memory is also allocated dynamically. This translates into several small calls to `malloc`, and possibly some to `realloc`. In the Network module, the data structures are allocated in one single call to `malloc`, which is faster.

- **Data access**: In the Protocols module, data is not directly available. It is necessary to perform some bitwise operations to extract the data from its byte array representation. With the Network module, data is directly available through the use of `bitfields`.

When executing more than one thread, the differences start fade (see Fig. 5.6). This happens because calling external modules require additional changes to the internal data structures of Lua, which involves locking the global shared mutex (see Appendix B). This affects the module performance because it serializes calls to the external module. Nevertheless, Protocols is still a lot more expensive when accessing data, as we can see by the operations `Statistics` and `Everything` in Fig. 5.6.

For this reasons, the Protocols module (Lua implementation) has been replaced by the Network module (C implementation).
5.2 Bottlenecks

Fig. 5.6: Comparison between the Protocols and Network modules

5.2.2.3 Other bottlenecks

No other significant bottlenecks were found on the Framework.
Chapter 6
Conclusions

The main objective of this thesis was to create a Framework to be the basis of an architecture capable of detecting, analysing and correlating anomalies on IP networks. To achieve this goal, a set of modules were created:

- **Bindings** – Provide a bridge between Lua programs and native operating system libraries.
  - *Libpcap* – Bindings to the linux network packet capture library
  - *Pthread* – Bindings to the linux posix thread library
  - *Signal* – Bindings to handle application signals
- **Network & Protocols** – Decode network packets from a byte array and vice-versa (C and Lua implementation, respectively).
- **Metaplugin** – Plugin support for applications.
- **Looplog** – Auxiliary module to help in loop detection (e.g. used by Metaplugin).
- **Persistence** – Persistent storage used by plugins.
- **Serializer** – Serialize any basic Lua data type.
- **Message Queue** – Exchange information between plugins.
- **Remote Call** – Share plugin code that may be useful to other plugins
- **Timeline** – Transform a discrete sample set into a continuous sample set
- **Root Cause Analysis** – Coordination module used in the event correlation.
- **Scheduler** – Timer loop that calls a specific function every few seconds or nanoseconds.
- **Running average** – Keep track of both mean and standard deviation of samples.
- **Fuzzy Logic** – Lua implementation of a Fuzzy Logic algorithm.

These modules compose the Framework, and were used to build three different applications:

- **Collector** – Collects data from the network.
- **Analyzer** – Analyzes a Collectors data.
- **Correlator** – Correlates data from all Analyzers.
Each application has its own set of plugins that perform very specific tasks (with a total of sixteen plugins). Together, these applications (and respective plugins) are capable of detecting a small subset of faults on IP networks.

The performed tests have shown that the plugins performed as expected, but some bottlenecks were found on the Framework. The first was related with the Lua patch that enables multi-threading, called *Permissive*. This was happening because Lua relinquishes the CPU too often, which wastes too much time in context switching. It has been fixed with the *Aggressive* Lua patch, that removes the `threadyield` feature, so threads are forced to use all of their CPU time slice. The other bottleneck found was on the *Protocols* module (Lua implementation). It became obvious that this module was very expensive (time-wise) when accessing packet data. This was also fixed, by reimplementing the module in C, which was named *Network*.

As for future work, this was only the first iteration of a very complex project, so there’s still plenty of room for improvement:

- Support more protocols, including IPv6.
- Support for automatic configuration through high-level policies.
- New algorithms for anomaly detection.
- New modules to further ease the implementation of plugins.
References

9. ITU-T X.680 (07/94) ASN.1 Specification of Basic Notation
24. S. Boros, Policy-based network management with SNMP, EUNICE 2000 Summer School, University of Twente, Netherlands, pp. 1-3 (2000)
Appendix A

Programming language

For the development of the Framework, Applications and Plugins, a choice of programming language had to be made. The first decision was whether to use a scripting language only for Plugins, or for all the code.

Advantages of using a scripting language:

- Easier to write.
- Easier to modify.
- Easier to understand.
- Rapid development of new features.
- Usually less lines of code to do the same functions.
- Smaller file size.

Disadvantages of using a scripting language:

- Slower execution.
- Potentially limited performing certain tasks.
- Depends on the interpreter.
- Slightly harder to debug.

The main disadvantages for using a scripting language are the slower execution and being potentially limited. The slower execution can easily be overcome by using a faster computer, and by wisely choosing the scripting language. Limitations of the scripting language can also be overcome if there’s the possibility to extend it with some external modules.

With the Framework requirements in mind (see Sec. 3) and by evaluating both pros and cons, it was decided that all code (Framework, Applications, and Plugins) was to be written in a scripting language. Nevertheless, it was also decided that a non-scripting language (e.g. C programming language) could be used whenever high performance code was required (see Sec. 3.2).

Table A.1 summarizes the scripting languages that have been considered and their characteristics.
Table A.1: Summary of the scripting languages that have been considered

<table>
<thead>
<tr>
<th></th>
<th>Javascript</th>
<th>Lua</th>
<th>Perl</th>
<th>Python</th>
<th>Ruby</th>
</tr>
</thead>
<tbody>
<tr>
<td>Easy to learn</td>
<td>★★★</td>
<td>★★★</td>
<td>★</td>
<td>★★★★</td>
<td>★★</td>
</tr>
<tr>
<td>C integration</td>
<td>★</td>
<td>★★★★</td>
<td>★</td>
<td>★</td>
<td>★★★</td>
</tr>
<tr>
<td>Multi-thread support</td>
<td>×</td>
<td>☑️️️</td>
<td>☑️️</td>
<td>☑️️️</td>
<td>☑️️️</td>
</tr>
<tr>
<td>Speed and Memory</td>
<td>★</td>
<td>★★★★</td>
<td>★★★</td>
<td>★★★★</td>
<td>★★★</td>
</tr>
<tr>
<td>Core size</td>
<td>★★★</td>
<td>★★★★</td>
<td>★</td>
<td>★★★★</td>
<td>★★★</td>
</tr>
<tr>
<td>Support</td>
<td>★</td>
<td>★</td>
<td>★★★★</td>
<td>★★★★</td>
<td>★★★</td>
</tr>
</tbody>
</table>

1 Requires Lua to be recompiled and an external module

From the Table is easy to understand why Lua was chosen. Although it lacks a big user base and support libraries when compared to the others, it’s still the best in all the other categories.
Appendix B

Lua multi-thread patch

Lua does not implement true multi-threading, that is preemptive threads sharing memory. The reason for this is that Lua only implements ANSI C features\(^1\) which does not specify multi-thread.

Nevertheless, the code includes some basic (non-optimized) multi-threading support in the form of function stubs. These stubs have to be reimplemented to include specific calls to operating system libraries:

- `lua_lock`: Locks a global mutex to get exclusive access to internal data structures.
- `lua_unlock`: Unlocks the global mutex that gives exclusive access to internal data structures.
- `luai_threadyield`: Used by the running thread to relinquish the CPU.

To reimplement the stubs, two files were created: `luser.h` and `luser.c`.

Contents of `luser.h`:

```c
#ifndef LUSER_H_
#define LUSER_H_
#define lua_lock(L) lua_Lock(L)
#define lua_unlock(L) lua_Unlock(L)
#define luai_threadyield(L) {lua_Unlock(L); \ luai_ThreadYield(L); \ lua_Lock(L);}

#include "lua.h"

void lua_Lock (lua_State * L);
```

\(^1\) One of Lua objectives is to be as portable as possible
Later, while implementing the applications it became obvious that this patch was not performing very well (see Sec. 5.2). After some investigation, it was discovered that `luai_threadyield` was the culprit. `luai_threadyield` is called whenever the running thread wants to relinquish the CPU for another thread to run. A quick search in Lua's source code revealed the following:

```
#define dojump(L,pc,i) {((pc) += (i);  
   luai_threadyield(L);)}

[...]

case OP_JMP: {
   dojump(L, pc, GETARG_sBx(i));
   continue;
```
case OP_EQ: {
  TValue *rb = RKB(i);
  TValue *rc = RKC(i);
  Protect(
    if (equalobj(L, rb, rc) == GETARG_A(i))
      dojump(L, pc, GETARG_sBx(*pc));
  )
  pc++;
  continue;
}

case OP_LT: {
  Protect(
    if (luaV_lessthan(L,RKB(i),RKC(i)) == GETARG_A(i))
      dojump(L, pc, GETARG_sBx(*pc));
  )
  pc++;
  continue;
}

case OP_LE: {
  Protect(
    if (lessequal(L, RKB(i), RKC(i)) == GETARG_A(i))
      dojump(L, pc, GETARG_sBx(*pc));
  )
  pc++;
  continue;
}

case OP_TEST: {
  if (l_isfalse(ra) != GETARG_C(i))
    dojump(L, pc, GETARG_sBx(*pc));
  pc++;
  continue;
}

case OP_TESTSET: {
  TValue *rb = RB(i);
  if (l_isfalse(rb) != GETARG_C(i)) {
    setobjs2s(L, ra, rb);
    dojump(L, pc, GETARG_sBx(*pc));
  }
  pc++;
  continue;
}
case OP_FORLOOP: {
    lua_Number step = nvalue(ra+2);
    lua_Number idx = luai_numadd(nvalue(ra), step);
    lua_Number limit = nvalue(ra+1);
    if (luai_numlt(0, step) ? luai_numle(idx, limit) :
        luai_numle(limit, idx)) {
        dojump(L, pc, GETARG_sBx(i));
        setnvalue(ra, idx);
        setnvalue(ra+3, idx);
    }
    continue;
}

case OP_FORPREP: {
    const TValue * init = ra;
    const TValue * plimit = ra+1;
    const TValue * pstep = ra+2;
    L->savedpc = pc;
    if (!tonumber(init, ra))
        luaG_runerror(L, LUA_QL("for") \ " initial value must be a number");
    else if (!tonumber(plimit, ra+1))
        luaG_runerror(L, LUA_QL("for") \ " limit must be a number");
    else if (!tonumber(pstep, ra+2))
        luaG_runerror(L, LUA_QL("for") \ " step must be a number");
    setnvalue(ra, luai_numsub(nvalue(ra), \
        nvalue(pstep)));
    dojump(L, pc, GETARG_sBx(i));
    continue;
}

case OP_TFORLOOP: {
    StkId cb = ra + 3;
    setobjs2s(L, cb+2, ra+2);
    setobjs2s(L, cb+1, ra+1);
    setobjs2s(L, cb, ra);
    L->top = cb+3;
    Protect(luaD_call(L, cb, GETARG_C(i))); \
    L->top = L->ci->top;
    cb = RA(i) + 3;
    if (!ttisnil(cb)) {
    [...]
}
The snapshot above shows that `luai_threadyield` is being called on the op-codes: `OP_JMP`, `OP_EQ`, `OP_LT`, `OP_LE`, `OP_TEST`, `OP_TESTSET`, `OP_FORLOOP`, `OP_FORPREP`, `OP_TFORLOOP`. This causes too many context switches and wastes CPU time (see Fig. B.1).

To solve this issue, the patch was modified so that `luai_threadyield` does not relinquish the CPU (causing the running thread to use all of its CPU time, minimizing context switches):

Contents of `luser.h`:

```c
#ifndef LUSER_H_
#define LUSER_H_

#define lua_lock(L) lua_Lock(L)
#define lua_unlock(L) lua_Unlock(L)
#define luai_threadyield(L) {}

#include "lua.h"

void lua_Lock (lua_State * L);
void lua_Unlock (lua_State * L);

#endif /* LUSER_H_ */
```

Contents of `luser.c`:

```c
#include <pthread.h>
#include "lua.h"

static pthread_mutex_t lua_mutex = PTHREAD_MUTEX_INITIALIZER;

void lua_Lock (lua_State * L) {
    pthread_mutex_lock(&lua_mutex);
}
```
With this modification the performance improved significantly (see Sec. 5.2).

The patches were named *permissive* and *aggressive*, representing their policy regarding CPU sharing.