NetOdyssey: A Framework for Real-Time Windowed Analysis of Network Traffic

Fábio Duarte Beirão

Submitted to the University of Beira Interior in candidature for the degree of
Master of Science in Computer Science and Engineering

Supervised by Mário Marques Freire

June 2010

Departament of Computer Science
University of Beira Interior
Covilhã, Portugal
http://www.di.ubi.pt

ii

Acknowledgments

"This one goes out to the one I love..."

R.E.M. – The One I Love

The ones we love are the ones who make us stand tall and step up to the challenges of life. Yet, they do not just shove us into these challenges: they are there, with us, every step of the way. They suffer with us, they understand our deepest frustrations and they rejoice with us on our victories, no matter how small or big they are. To all of you, the ones I love, Mother, Father, Brother, my forever loved Marisa, my friends, Dr. Pedro Inácio, João Gomes, and my always supportive supervisor Professor Mário Freire; to all of you who made it possible for me to overcome this challenge, I thank you. You were the ones who made this journey possible, because as life teaches us so well, no man is an island.

There is always much to be done, there is always much to be said, but there is one thing that can't remain undone, that can't remain unsaid: to tell you how much I appreciate all your effort. Remain assured: not one single moment you spent was wasted. My deepest gratitude goes to each of you.

Fábio Duarte Beirão

Abstract

Traffic monitoring and analysis is of critical importance for managing and designing modern computer networks, and constitutes nowadays a very active research field. In most of their studies, researchers use techniques and tools that follow a statistical approach to obtain a deeper knowledge about the traffic behaviour. Network administrators also find great value in statistical analysis tools. Many of those tools return similar metrics calculated for common properties of network packets. This dissertation presents *NetOdyssey*, a framework for the statistical analysis of network traffic. One of the crucial points of differentiation of *NetOdyssey* from other analysis frameworks is the windowed analysis philosophy behind *NetOdyssey*. This windowed analysis philosophy allows researchers who seek for a deeper knowledge about networks, to look at traffic as if looking through a window. This approach is crucial in order to avoid the biasing effects of statistically looking at the traffic as a whole. Small fluctuations and irregularities in the network can now be analyzed, because one is always looking through window which has a fixed size: either in number of observations or in the temporal duration of those observations. *NetOdyssey* is able to capture live traffic from a network card or from a pre-collected trace, thus allowing for real-time analysis or delayed and repetitive analysis. *NetOdyssey* has a modular architecture making it possible for researchers with reduced programming capabilities to create analysis modules which can be tweaked and easily shared among those who utilize this framework. These modules were thought so that their implementation is optimized according to the windowed analysis philosophy behind *NetOdyssey*. This optimization makes the analysis process independent from the size of the analysis window, because it only contemplates the

observations coming in and going out of this window. Besides presenting this framework, its architecture and validation, the present Dissertation also presents four different analysis modules: *Average and Standard deviation*, *Entropy*, *Auto-Correlation* and *Hurst Parameter* estimators. Each of this modules is presented and validated throughout the present dissertation.

Keywords

Analysis of Traffic Behavior, Auto-correlation estimator, Average and Standard Deviation, Entropy estimator, Hurst Parameter, Mersenne Twister pseudo-random generator, Modular Approach, Random Capture Generator, Real time Analysis, Statistical Traffic Analysis, Windowed Analysis of Network Traffic

Contents

List of Figures

Acronyms and Abbreviations

Acronyms

AWS   Analysis Window Size

AWT   Analysis Window Time

BCTU   Bit Count per Time Unit

CPU   Central Processing Unit

CSV   Comma Separated Values

CLR   Common Language Runtime

DoS   Denial of Service

DPI   Deep Packet Inspection

DLL   Dynamically Linked Library

DTLS   Datagram Transport Layer Security

GUI   Graphical User Interface

HEAF   Hurst Exponent by Autocorrelation Function

HTTP   Hypertext Transfer Protocol

ICMP   Internet Control Message Protocol

IEEE   Institute of Electrical and Electronics Engineers

IETF   Internet Engineering Task Force

IOS   Internetwork Operating System

IP   Internet Protocol

IPFIX   Internet Protocol Flow Information Export

IPv4	Internet Protocol version 4
IPv6	Internet Protocol version 6
IPX	Internetwork Packet Exchange
LSM	Least Square Method
NMCG	Network Multimedia and Computing Group
NIC	Network Interface Card
OS	Operating System
PC	Personal Computer
PR-SCTP	Partial Reliability Stream Control Transmission Protocol
RFC	Request For Comments
RMON	Remote Network Monitoring
SCTP	Stream Control Transmission Protocol
STD	Standard
SSL	Secure Sockets Layer
SMTP	Simple Mail Transfer Protocol
SNMP	Simple Network Management Protocol
SNMPv3	Simple Network Management Protocol version 3
TCP	Transmission Control Protocol
TLS	Transport Layer Security
UDP	User Datagram Protocol
XML	Extensible Markup Language

Abbreviations

Please consider the meaning of the following abbreviations when you find them later in the text:

a.k.a.	also known as;
e.g.	for example;
i.e.	that is to say; in other words;
vs.	versus.

Chapter 1

Introduction

1.1 Focus and Scope

Internet, the great network that connects everything and everyone who has access to it, is a game-changing paradigm for many. Nowadays, there are jobs exclusively on the Internet, social networks that allow us to *know* other people, video and voice conferences that make physical distances as short as the available bandwidth. In the past 30 years, ever since the first "hello world" Internet communication, the Internet has grown in ways that could not have been predicted. This is evident, for instance, nowadays, when we are realizing that the addressing space of Internet Protocol version 4 (IPv4) is becoming scarce. In the early days of the Internet, it was never thought that it would reach so many people and affect so many lives. The Internet of today is no longer available for a minority of experts only. There is a growing usage by children and elder people, that either seek for entertainment and gaming, or for some companion, which many find *on-line*. The ever growing popularity and ease of access to the Internet brings with it questions that are constantly arising: *how safe is the Internet?*, *is there any privacy on the Internet?*, *is it possible to remain anonymous on the Internet?*, among other questions, whose answers might not remain constant throughout time.

On the Internet, just as outside it, there are always those persons who have malicious intentions. These malicious intentions break laws and rules,

disrupting the harmonious coexistence of society. On the Internet, there are several ways to disrupt the normal way of working, for instance, Denial of Service (DoS) attacks, *phishing* scams, identity theft, amongst other threats. With the convergence of services on the Internet such as banking, grocery shopping, renting and buying music, videos or books, among many other free or paid on-line services, the potential damage caused by these threats becomes much more significant.

The Internet can be seen as a variety of things, depending on how people use it: some people see the Internet as a way to navigate on websites; some people see the Internet as a way to talk to other people; others see the Internet as a way to read news; others see the Internet as a mean to freely easily obtain content that otherwise would need to be paid; among other ways that the Internet may be seen. This happens because so many of the actual Internet users do not know (and do not need to know) how everything works *under-the-hood*. It is because the Internet allows for this ease of access that it has a growing popularity and it is also because of this needed ignorance that many attacks are possible and cause potential damage.

All of these issues lead to a lot of research and development, in order to try to make the Internet a safer place. This is a constant struggle, one which is necessary, because challenges are a requirement for evolution. In order to understand the problems of the Internet of today and the challenges of the Internet of tomorrow, one needs to comprehend the behavior of Internet users and their needs. With the increasing bandwidth availability and decreasing Internet access prices, Internet users are becoming more demanding and with greater expectations than ever. With the advent of on-line radio, video and television streaming, standard Internet communication protocols faced a challenge: deliver quality or deliver quantity. These questions are always actual, because tomorrow a new need will emerge, which was not predictable today, so there is a need to understand how can the Internet respond to that challenge. This is why it is important to have means to understand how the Internet behaves and how users on the Internet use it: what is their experience, what are their problems, and what can be done to solve or prevent those problems.

In this dissertation, the author describes a new tool, which helps in this process of understanding the Internet. Although there are several research communities and tools, the author felt the need to analyze the Internet in a particular manner that, to the best of the knowledge of the author, was not yet addressed. Thus, this dissertation presents and describes *NetOdyssey*, a tool to facilitate the process of statistically analyzing the behavior of the Internet traffic.

1.2   Problem Definition and Objectives

As described in section 1.1, the need for understanding the behavior the usage of the Internet, and other networks, in order to better comprehend their expectations, is constant throughout the evolution of the Internet. This behavior can be assessed by long term analysis, performed in some Internet nodes, which are able to gather Internet traffic from several sources. Some of the currently available tools that permit this analysis are presented in Chapter 2, but, to best knowledge of the author of this dissertation, none of them presents an approach to analyzing Internet traffic as *NetOdyssey* (the outcome of this dissertation) does.

This dissertation seeks the easiness of the process of understanding the behavior of any given network (Internet or other), but he is also looking for freedom in this process of analysis. One of the problems the author was faced with was the need to analyze non–conventional network parameters. This means that while a researcher may wish to analyze one specific network metric today, she/he might wish to analyze a radically different network metric tomorrow. Also, the author of this dissertation wishes to look at the network in a non–traditional way. Traditionally, network communications may be seen as unidirectional or bidirectional flows, but within the scope of this dissertation, there is a wish to look at the network as a whole. So, in order to avoid losing information with vast analysis periods, the author wishes to analyze the network as if he was looking through a window, and network packets where passing through that window. This analysis philosophy will be throughly explained in section 3.3.

Specifically, the main objective of this dissertation is to present a framework which:

- Allows researchers to understand the behavior of a network by statistically analyzing it;

- Looks at the network as if one was through a *window*; this *window* should be parameterizable in size and fashion of sliding;

- Provides modularity and expandability, so the analysis is not confined to predefined metrics;

- Allows for researchers to easily exchange analysis modules and results, in order to share knowledge and conclusions.

1.3 Main Contributions

This section presents the main contributions of the present dissertation, in accordance with the opinion of the author.

The first and main contribution of this dissertation is the *NetOdyssey* network analysis framework itself. This framework is the outcome of a research and engineering effort, which provides researchers with special interest in statistically analyzing network traffic with a means to perform this analysis in a windowed manner. Also, the *NetOdyssey* framework facilitates the sharing of knowledge and information, due to its open nature. This framework was presented to the scientific community at the 9th Institute of Electrical and Electronics Engineers (IEEE) International Symposium on Network Computing and Applications (IEEE NCA10) [1]. As it is described in Chapter 3, *NetOdyssey* provides a modular approach, which means that the analysis is performed by independent modules, allowing them to be easily tweaked and adjusted as the researcher sees fit.

The other main contribution of this dissertation is the pack of available modules, presented and validated in section 4.3 which are useful not only for the analysis they preform, but also as a basis for other researchers to learn and understand how to create their own *NetOdyssey* analysis modules. This pack of modules include an average and standard deviation

calculator, an entropy estimator, an auto-correlation estimator and a Hurst parameter estimator.

Although the contributions of this dissertation may seem short in number, the author strongly believes that *NetOdyssey* is a framework that may grow and, through the contributions of those who will use it, mature even further. In section 5.2 directions for future work of *NetOdyssey* are proposed.

1.4   Organization of the Dissertation

The *body* of this dissertation is constituted by three main chapters, preceded and succeeded by the *Introduction* and *Final Conclusions and Future Work*, respectively. There are also two Appendixes in the end of this dissertation. The compilation of the bibliographic references used along this work is included after Chapter 5. The contents of each one of the chapters of this dissertation can be summarised as follows:

Chapter 1 elucidates the context for the subject on which this dissertation is going to elaborate on, identifying the main objectives and the problem to be solved. In this chapter it is also included the description of the main contributions resulting from this dissertation.

Chapter 2 provides an overview of the current state of the art regarding the subjects of interest of this dissertation. The three main subjects of interest reviewed in this chapter were network tools for packet capturing and analysis, network information protocols and plotting tools.

Chapter 3 is the core of this dissertation, presenting the *NetOdyssey* framework. The tools user for the development of *NetOdyssey* are enumerated and the calculation philosophy and modular approach are also depicted. In this chapter there is also an example of an user analysis module, which is created and explained step by step.

Chapter 4 presents the whole process of validating the *NetOdyssey* framework, from the creation of a random capture generator, to the validation of the analysis modules provided with this dissertation.

Chapter 5 wraps up the most important conclusions of this dissertation, while also providing some directions for future research and work.

Appendix A depicts the detailed description of the class model in a high-level analysis, detailing what is the purpose of each programmatic class of *NetOdyssey*.

Appendix B with has a pretty print of the source code of the implemented modules that are one of the contributions of this dissertation.

Chapter 2

State of the Art and Critical Review of Tools for Network Monitoring and Analysis

2.1 Introduction

The main idea behind this section is to briefly present some important and popular tools that are – each in their own way – related to the scope of this dissertation. This dissertation has three subjects of interest and the current state of the art related to each of them is presented in this section.

The first subject of interest in this dissertation is the process of capturing and analyzing packets traveling in a network. In section 2.2 some of the most well known tools for analysis of network traffic will be presented [2–9]. Due to the relatively large number of available tools, only some of the most important and well-known tools are presented. Because there are a lot of published research and tools available the focus of this chapter is the analysis of those works, in order to gather a deeper understanding of the focus of other researchers.

The second subject of interest in this dissertation is the process of statistical analysis of data gathered by the packet capturer. In section 2.3 some network protocols will be presented that, with the help of active network equipment, allow for some analysis of network metrics. This section is

presented here, because it was important for the author to understand which network metrics are being analyzed in real-world scenarios and how are they gathered and processed.

The third subject of interest in this dissertation is the representation of the results. This representation is usually done by creating a chart, using plot tools. In section 2.4 a brief description of some available plotting tools is presented, but since this is the least important subject of this dissertation, the analysis of the state of the art concerning this section was more shallow.

2.2   Overview of Network Tools for Packet Capturing and Analysis

This section presents some of the most relevant tools available for capturing and analyzing network packets.It focuses on tools that, in the opinion of the author, have a closer scope to the goal of this dissertation, allowing a better understand about what has been done and what may remain to be done.

2.2.1   Wireshark

Wireshark is widely known as the *de facto* standard across educational institutions, as claimed in [4]. Although *Wireshark* is known as a network analysis tool, it is not the kind of analysis that one is looking for at the Network Multimedia and Computing Group (NMCG) [10]. The analysis provided by *Wireshark* is aimed at the packets themselves instead of statistical analysis. *Wireshark* enables the understanding of packets per network layers and conversation flows. Still, this section presents *Wireshark* in this state-of-the art, because it is one of the most well-known network analysis tools and it helped in the process of validating his dissertation.

The analyze menu provided by the Graphical User Interface (GUI) of *Wireshark* contains the following features:

– Display filters and display filter macros: this feature allows the creation and management of filters that define which of the captured packets will be displayed. Filters can be simple expressions, like

```
ip.src == 192.168.0.1
```

(display packets which Internet Protocol (IP) address source is `192.168.0.1`), or more complex expressions, like

```
not (tcp.port == 80) and
not (tcp.port == 25) and
   ip.addr == 192.168.0.1
```

which displays non Hypertext Transfer Protocol (HTTP) and non Simple Mail Transfer Protocol (SMTP) packets to and from IP address `192.168.0.1`. Display filter macros are a mechanism to create shortcuts for complex filters. It is possible for example to create a filter called `myFilter`, whose expression is

```
(ip.src == $1 and ip.dst == $2)
```

and then use the filter like

```
${myFilter:192.168.0.1;192.168.0.254}.
```

For very complex and repetitive filters, it becomes very useful to use macros in order to minimize accidental typos and ease the analysis process;

– Follow Transmission Control Protocol (TCP) (defined in Standard (STD)007 [11]), User Datagram Protocol (UDP) (defined in STD0006 [12]) and Secure Sockets Layer (SSL) stream: this feature allows the following of the conversation stream, for the currently selected packet. TCP streams are well defined, because they have a sequence number, a stream beginning and a stream ending. When requested to follow a TCP stream, *Wireshark* applies a filter such as

```
(tcp.stream eq 29).
```

This implies that there is an identifying number for TCP streams generated by *Wireshark* when it captures a new TCP stream initialization. UDP streams are more tricky to follow, because they are just theoretical streams, due to the session-less nature of UDP. So, when requested to follow an UDP stream, *Wireshark* applies a filter like

```
((ip.addr eq ip1 and ip.addr eq ip2) and
 (udp.port eq port1 and udp.port eq port2)).
```

Thus, it is possible to understand that *Wireshark* considers UDP streams as a combination of IP addresses (source and destination) and UDP ports;

– Expert info and Expert info composite: this feature presents a window with a log of errors, warnings and notes about the current capture. The log lines are categorized with four severity levels, described as follows:

* Chat: information about usual workflow, e.g. a TCP packet with the SYN flag set, or a TCP connection reset (RST flag set);

* Note: notable information, e.g. an application returned an "usual" error code like HTTP 404, or there was a TCP duplicate ACK;

* Warn: warning, e.g. application returned an "unusual" error, like a connection problem, or there was a TCP segment lost (which may be usual at capture start);

* Error: serious problem, e.g. a malformed packet. Expert info composite displays the same information as expert info, but grouped in a tabular fashion (one tab for each severity level).

2.2.2   Analyzer 3.0 (alpha)

Analyzer 3.0 (alpha) is a network packet analyzer for the Win32 platform. It is claimed in [7] that *Analyzer* includes several functionalities that are needed by a network management operator. *Analyzer* is based on Win-Pcap [3], thus making it able to capture packets on most Win32 platforms. *Analyzer* is currently being developed at Politecnico di Torino [13], although

at the time this dissertation was written, the latest version of *Analyzer* was 3.0a12, dated November 15th, 2007. The source code of *Analyzer* is available. Just as WireShark, *Analyzer* is able to capture and display packets on both local machines and remote probes. *Analyzer* is able to parse network packets according to a protocol descriptor contained in an external file. This protocol descriptor is written in NetPDL language [14], a markup language describing network packets, maintained by NetBee Library group [8].

Some of the main features of *Analyzer* are:

- Advanced capture settings: it is possible to define how many packets should be captured (or capture until session is stopped), or the duration of the capture in seconds. It is also possible to define a custom size for the packet snapshot length, allowing for instance to capture only the packet headers (98 bytes, according to the default settings of *Analyzer*). Simple packet sampling options are also available: capturing 1 packet every N packets or 1 packet every N milliseconds

- Remote capturing: based on the remote capturing capabilities of Win-PCap itself, *Analyzer* allows for packet capturing on remote sites. These remote probes must also be running *Analyzer* and be ready to accept a remote monitoring session. Remote capturing allows for UDP transport and automatically ignoring packets belonging to the current remote capture session;

- NetInject: this module allows to inject a capture file created by *Analyzer* itself. It provides options like infinite looping or defining the number of times to inject the capture, respecting capture timings, sending as fast as possible, setting the number of packets per second or Kbytes per second;

- NetMiner: this module allows some data mining to be performed on captured network traffic. Its algorithm allows for the detection of frequent itemsets (e.g. which are the top 10 couples of Hosts/Ports?) and association rules (e.g. `IP 192.168.0.1 -> MAC AA11BB-223344`). The definable data mining parameters are: minimum support (e.g. a value of 0.1% means that only results referring to

more than 0.1% of traffic will be considered) and minimum confidence (e.g. 80% means that only rules valid for more than 80% of applicable traffic will be considered).

– End-to-end reachability monitor: based on Internet Control Message Protocol (ICMP) echo (a.k.a. ping) packets, this module is able to periodically check for hosts ping reachability, HTTP and SSL avail-ability and respond with one or more of the following actions: ring a bell, start traffic capture and send an e-mail to several recipients.

The problems that the author of this dissertation encountered during his analysis of *Analyzer 3.0* alpha were:

– Although *Analyzer* allows for pcap (libpcap and winpcap) captures to be opened, it does not allow the recording of its captures in any other format than *Analyzer* capture format. This is a big limitation, because the interoperability of capture files allows for more applications to analyze them;

– It was not possible to test NetMiner module, because an error was displayed, while opening the source database. Several different con-figurations were tried but there was no success;

– Starting a new NetLogger instance or opening a NetLogger database file caused *Analyzer 3.0* to simply crash without any warning. Several different configurations were tried but there was no success;

– Although the menu for link-layer statistics exists, it simply shows a message box saying it is not implemented.

2.2.3   ntop

ntop is a network probe based on libpcap [2]. *ntop* may be seen as a simple Remote Network Monitoring (RMON)-like agent, with an embedded web interface. *ntop* architecture is described in [9], and, at the time this dissertation was written, the team behind *ntop* was working on *PF_RING*, a module that allows the enhancement of the packet capturing process, by allowing packets to travel directly from the Network Interface Card (NIC)

to *PF_RING*, instead of taking the traditional path within the Operating System (OS) kernel. This approach is enabling *ntop* to be able to capture packets that travel in 10Gb links running in commodity Personal Computers (PCs), while optimizing the overall capturing process.

The main features of *ntop* are:

- Displaying traffic statistics, such as unicast, multicast and broadcast counters, packet size grouping, bad packet checksum count, total IP traffic, IP fragmented traffic, and non-IP traffic, number of hops (distance) for each packet, as well as other statistics;

- Sorting of network traffic according to many protocols, such as IPv4, Internet Protocol version 6 (IPv6), Internetwork Packet Exchange (IPX), AppleTalk, and others, and showing the distribution of IP traffic among those protocols;

- Identifying computer users (e.g. email address, through the capture of SMTP packets);

- Passive host OS identification (though the OS fingerprint and patterns contained in some packets);

- Displaying IP traffic Subnet matrix (who is talking to who);

- Acting as a NetFlow (see 2.3.2) collector;

- Protocols are user configurable, meaning a user can teach *ntop* about a specific protocol he wishes to be understood when captured, so it can be included in statistics.

ntop has other features, but none of them fulfill the requirements of the research work behind this dissertation. *ntop*, like other tools analyzed and experimented by the author, does not have the capability of measuring non-conventional network metrics, and also does not allow for that analysis to be done with a sliding window manner.

2.2.4   CoMo – Continuous Monitoring

CoMo, described in [6], is a passive monitoring system, designed to be the basic building block of an open network monitoring infrastructure. Because

CoMo was though to allow the capturing and analysis of traffic over multiple sites, it faces a great challenge to provide privacy and security guarantees to the owner of the monitored link. *CoMo* tries to fulfill the requirements of *Openness*, allowing users to customize the system and the software platform to their specific needs; and *Resilience*, since the system should be able to monitor and analyze the traffic in any load condition, specially under unexpected traffic anomalies that may overload the system resources.

The philosophy behind *CoMo* is to allow researchers to create several analysis module and them run them (with due permission) on the remote site. This means that the traffic itself is not available to researchers, but instead only the analysis returned by the module.

2.2.5   The NetBee Library

NetBee is a library intended for packet sniffing, packet decoding and traffic classification. *NetBee* was created and is being developed by the same research group that created WinPCap [3]. However, it is claimed in [8] that the WinPcap architecture is rather old and does not fit for nowadays needs. This claim emphasizes the lack of modularity and extendability of WinPcap (and libpcap) architecture. This limitation, combined with the extremely hard to upgrade nature of WinPcap (it is very easy to break backward compatibility), lead to the choice of creating a new library, from scratch, with a new architecture, object–oriented and open to extensions.

The main problem around *NetBee* is its lack of maturity. In fact, the creators of *NetBee* warn that current releases must be intended as a proof of concept. They also assure that due to the try and error nature of early development, new releases of *NetBee* will almost surely break backward compatibility. These issues lead us to exclude *NetBee* for traffic analysis, even if *NetBee* would be, theoretically, a good choice.

2.3   Overview of Network Information Protocols

This section describes some network protocols that facilitate the process of analyzing network traffic and behavior. These protocols exist because typically, network analysis needs to be conducted on terminal network nodes, which are able to gather packets from several sources. These terminal nodes are usually active network equipments, such as switches and routers. These active network equipments run an OS that allows them to gather and analyze network information. The protocols presented in this section enable the process of exporting this data, so they can be stored, presented and further analyzed.

2.3.1   SNMPv3

Simple Network Management Protocol version 3 (SNMPv3) is described in STD0062 [15] and it provides a framework for managing networks. An *SNMPv3* management system contains several nodes (traditionally called agents), each with access to management instrumentation; at least one *SNMPv3* entity (traditionally called a manager), containing command generator and/or notification receiver applications; and a management protocol, used to convey management information between the *SNMPv3* entities.

The main goals that drove the architecture behind *SNMPv3* were [15]:

- Use existing materials as much as possible;

- Address the need for security, which was considered the most important flaw with previous implementations of this protocol;

- Allow for architecture evolution in the standards track, even if consensus has not been reached on all pieces, making it possible to upgrade portions of *Simple Network Management Protocol (SNMP)* without disrupting an entire *SNMP* framework;

- Keep *SNMP* as simple as possible (*S* stands for Simple).

STD0062 [15] describes the main security requirements of *SNMPv3*, but a more thorough analysis of these security measures would fall outside the

scope of this state of the art review. Thus, we will consider *SNMPv3* to be a secure and reliable protocol.

Although *SNMP* allows the gathering of many network properties and statistics (such as number of packets, erroneous packets, packet drops, among other management statistics), these statistics fall outside the purpose of this dissertation.

2.3.2   Cisco NetFlow

NetFlow is a protocol developed by Cisco Inc., that provides a key set of services for IP applications. Some examples of these services are network monitoring, DoS monitoring and network traffic accounting. Although *Net-Flow* was initially a proprietary protocol, it is now a mature protocol. *Net-Flow* is now an Internet Engineering Task Force (IETF) standard described in informational Request For Comments (RFC)3954 [16]. The actual version of *NetFlow* by the time the present dissertation was written was version 9 [17]. An IETF standard, known as Internet Protocol Flow Information Export (IPFIX) (described in subsection 2.3.3) is emerging inspired on this version of *NetFlow*.

The *NetFlow* version 9 export format uses templates to ease the task of observing IP packet flows, in an extensible and flexible manner. A template defines the collection of fields to be exported, with corresponding descriptions of structure and semantics. Active network elements, such as routers and switches (those that implement Cisco Internetwork Operating System (IOS)) gather informations about IP flows and export it to collectors, using UDP (defined in STD0006 [12]) or Stream Control Transmission Protocol (SCTP) (defined in RFC4960 [18]) packets. In this particular context, a flow is defined as an unidirectional sequence of packets, with some common properties. The collected data about these flows provides fine–grained metering for highly flexible resource usage accounting. This data is very granular, containing information such as IP addresses, packet and byte counts, timestamps, application ports, input and output interfaces, etc.

The template–based approach of *NetFlow* version 9 provides the following advantages:

- Just like – for instance – in Extensible Markup Language (XML), new fields can be added to the export records, without changing the structure of the export record format. This was a limitation of versions prior to version 9;

- Structural information about the exported flow records are contained in the export, thus, if the *NetFlow* collector does not understand the semantics of new fields, it can still interpret the flow record;

- Due to the flexibility provided by the template mechanism, it is possible to only export the fields that are required. This helps to reduce the volume of the exported data and also reduce network load.

A *NetFlow* Exporter (e.g. a Cisco IOS router or a Cisco IOS switch) gathers informations about flows and exports them using the *NetFlow* protocol. A flow can be exported under some conditions, such as:

- If an explicit end of flow is detected. For example, TCP (defined in STD007 [11]) has special flags (called FIN (as in finish) and RST (as in reset)) that explicitly terminate a TCP connection (because in this context flows are unidirectional, tcp connections represent 2 flows);

- If the flow has been inactive (no packets observed corresponding to this flow) for a given period of time;

- If it is a long-lasting flow, exports should be made on a regular basis;

- If the Exporter is experiencing internal constraints, such as low memort or counters wrapping.

Although the exports of previous versions of *NetFlow* were strictly encapsulated into UDP packets, *NetFlow* version 9 has been designed to be transport protocol independent, allowing it to operate over congestion-aware protocols, such as SCTP.

These exports should be collected by *NetFlow* Collectors and these are responsible for analysis, storage and/or presentation of the results to the user/network administrator. The *NetFlow* version 9 was designed with the

assumption and expectation that the Explorers and Collectors would remain within a single private network. Thus, no great security measures were implemented, no impositions on confidentiality, integrity nor authentication requirements. This greatly reduces the implementation complexity and also increased the efficiency of *NetFlow* version 9 protocol. As described in subsection 2.3.3, IPFIX (described in RFC5101 [19]) addresses these security considerations.

2.3.3   IPFIX

IPFIX is described in RFC5101 [19] and, at the time this dissertation was written, it had a status of Proposed Standard. RFC3917 [20] provides the requirements for the IPFIX and RFC5655[21] is also a Proposed Standard for a file format designed to facilitate interoperability and re-usability among a wide variety of flow storage, processing and analysis tools. Because IPFIX is based on Cisco NetFlow version 9 (see subsection 2.3.2), they share some similarities, such as the existence of export processes and collecting processes and the existence of data and template records. Nevertheless, *IPFIX* may be considered the version 10 of NetFlow. In fact, the version field in the *IPFIX* header contains the value 0x000a, which increments by one the value used in the NetFlow services export version 9.

Just as NetFlow version 9, *IPFIX* has been designed to be transport protocol independent. It is also notable that the exporter can export to multiple collecting processes using different transport protocols. RFC5101 [19] specifically states that in order to guarantee compliance and interoperability with different *IPFIX* implementations, SCTP (defined in RFC4960 [18]) using the Partial Reliability Stream Control Transmission Protocol (PR-SCTP) (defined in RFC3758 [22]) must be supported. UDP (defined in STD0006 [12]) and TCP (defined in STD007 [11]) are referred to as optional implementations. TCP may be used in environments susceptible to congestion, although there is a strong recommendation for the usage of PR-SCTP, due to its ability to limit back pressure on exporters.

The security considerations for the *IPFIX* protocol have been derived from RFC3917 [20], where an analysis of potential security threats is made.

Thus, the requirements for *IPFIX* security define that *IPFIX* must provide a mechanism to ensure the confidentiality of *IPFIX* data, in order to prevent disclosure of Flow Records; *IPFIX* must provide a mechanism to ensure the integrity of *IPFIX* data, in order to prevent the injection of incorrect data or control information into an *IPFIX* stream and *IPFIX* must provide a mechanism to authenticate Collecting and Exporting Processes, in order to prevent collection of data from an unauthorized Collecting Process, or the exportation of data from an unauthorized Exporting Process. Transport Layer Security (TLS) (defined in RFC4346 [23]) and Datagram Transport Layer Security (DTLS) (defined in RFC4347 [24]) were designed to meet the aforementioned security requirements of *IPFIX*, without the need for pre-shared keys. Since a deeper security analysis would fall outside the purpose of this protocol review, we will just assume *IPFIX* is a secure and reliable implementation, even though RFC5101 [19] presents some scenarios that could compromise the reliability hereby assumed.

2.4   Overview of Plotting Tools

This section presents some plotting tools that are well-known to the scientific community. These tools are important in the scope of this dissertation, because plotting data greatly helps the process of analyzing it, so conclusions and details can be assessed.

2.4.1   Microsoft Excel and OpenOffice Calc

Microsoft Excel and OpenOffice Calc are two well-known spreadsheet applications, with intuitive and simple GUIs, which allow for data analysis and processing. Although they both support extensions that enable much more analysis methods than those already implemented, they both have a physical limitation of 65.536 rows and 1.024 columns, which is predictable to soon become a limitation, given the amount of data intended to be analyzed in the scope of this dissertation. For instance, a 24 hour network capture, in a 2Mb link can contain about 22.000.000 100B packets: a much greater number of values than those which these tools are designed to handle.

2.4.2 RRDtool

In [25] it is claimed that *RRDtool* is the open source industry standard, high performance data logging and graphing system for time series data. RRD stands for Round Robin Database. *RRDtool* is available for several linux distributions and Microsoft Windows platforms. Source code is also available, so it is possible to compile it on other OSs.

RRDtool supports 2D plotting of captured data, as opposed to gnuplot (see 2.4.3) which also supports 3D plotting and mathematical functions as input. *RRDtool* supports different types of outputs, such as *png*, *svg* or *eps*.

2.4.3 gnuplot

Gnuplot [26] is a graphing utility available for several OSs such as Linux, Microsoft Windows, OSX and others. It is an open source utility and was originally created to allow scientists and students to visualize mathematical functions and data.

Gnuplot supports several types of plots, in either 2D or 3D representations. Several demos for the different plots are available, facilitating the learning process to use *gnuplot*. It is possible to export the plots to several file formats, such as *eps*, *jpg*, LaTeX, *png*, *svg*, and other formats and *gnuplot* is extensible, so new output modes can be added.

2.5 Conclusion

In the introduction of this chapter, the three subjects of interest for this dissertation were introduced. After gathering information about tools related to the scope of this dissertation, it was provided a new insight and a deeper comprehension about related works and the current state of the art.

About the first subject of interest of this dissertation, the process of capturing and analyzing packets traveling in a network, one may learn that, among the tools presented in this chapter, *CoMo* (see section 2.2.4) is the one that was author found to be closest to the tool developed by the author. Still,

the scope and goals of *CoMo* are different from the ones of *NetOdyssey* in the following aspects:

- *CoMo* was designed for multi site (different locations/organizations) traffic analysis. This raises security and privacy issues that are not relevant in the scope of this dissertation;

- *CoMo* was also designed to be used by multiple users, so there is a need for fair resource sharing, in order to guarantee those users are able to run the analysis they wish, without compromising the system and other users.

In this dissertation, these needs are relaxed, as the developed tool (see Chapter 3) was designed to fulfill our research needs, although also being modular and open enough for it to be used by other researchers.

Regarding the second subject of interest, the process of statistically analyzing the captured packets, it was learned that many current approaches are based on Cisco NetFlow (see section 2.3.2), IPFIX (see section 2.3.3) and SNMPv3 (see section 2.3.1). These approaches present tools that act as gatherers of information provided by these protocols, and then run some analysis on the gathered data. These approaches were outside the scope of this dissertation, because we wish to enable researchers to analyze any kind of metric, conventional and non-conventional.

Finally, for the third subject of interest, the representation of the results of the traffic analysis, the author decided to use *gnuplot* because of the high plotting versatility it provides and because this is a well-known tool in the research community.

Chapter 3

The NetOdyssey Framework

3.1 Introduction

The main purpose of the *NetOdyssey* framework was initially to support
some research work at the NMCG, but, as *NetOdyssey* evolved, it became
clear that it be could easily used by other researchers for network traffic
analysis. Thus, *NetOdyssey* has evolved to a modular approach, described
in section 3.5 , in order to make its usage easier. *NetOdyssey* encompasses
the actual trend of Central Processing Unit (CPU) manufacturers to deploy
more than one core inside the processors, by using a multi–thread approach.
As described in section 3.4, each analysis module, e. g. the packet capturing
code and other core components, all run in separate threads. Thus, if one
analysis module is slower to process than the others, it will not become a
bottleneck, since other modules can keep running, given there are enough
resources in the host machine. Even if an analysis module encounters
an exception and needs to be stopped, it will stop alone, and it will not
interrupt the whole analysis process of other modules. In Chapter 4, the
validation process of *NetOdyssey* is described.

3.2 Tools for Development of *NetOdyssey*

NetOdyssey was developed using Microsoft .NET Framework 3.5. Although
this framework was designed to develop applications for the Microsoft

Windows OS, it is possible to use Mono Project to run *NetOdyssey* on other OSs (namely Linux and Apple Mac OS). *NetOdyssey* was entirely written in C#.

In order to capture network packets within the Microsoft Windows platform, winPcap was used. Since winPcap is not natively object-oriented (as C# is), sharpPcap - a winPcap C# wrapper - was used.

3.2.1   Microsoft .NET Framework 3.5

According to [27], the .NET Framework is an integral Microsoft Windows component that supports building and running applications and XML Web services. One of the main objectives behind the .NET Framework is to provide a consistent object-oriented programming environment. The .NET Framework has two main components: the Common Language Runtime (CLR) and the .NET Framework class library. The CLR is the foundation of the .NET Framework, as it provides core services such as memory and thread management, strict type safety, providing accuracy, security and robustness. On the other hand, the class library is a comprehensive, object-oriented collection of reusable types that a developer can use on its applications.

3.2.2   Mono Framework

According to [28], Mono is a software platform designed to allow developers to easily create cross platform applications. It is an open source implementation of Microsoft .NET Framework based on ECMA [29] standards for C# and the CLR, sponsored by Novell [30]. The main motivation behind Mono is to lower the barriers of production of applications for Linux, embracing a successful and standardized software platform. On [28] it is also claimed that the benefits of Mono are:

– Popularity: built on the success of .NET, there are millions of developers that have experience building applications in C#;

- High-level programming: all Mono languages benefit from the features of the runtime, like automatic memory management, reflection, generics and threading;

- Base Class Library: a comprehensive class library provides thousands of built in classes to increase productivity;

- Cross Platform: Mono runs on Linux, Microsoft Windows, Mac OS X, BSD, Sun Solaris, Nintendo Wii, Sony Playstation 3, Apple iPhone. It also runs on x86, x86-64, IA64, PowerPC, SPARC (32), ARM, Alpha, s390, s390x (32 and 64 bits). It is claimed that applications designed with Mono are able to run on nearly any computer in existence;

- CLR: the use of the CLR allows developers to choose the programming language they best like to work with, and it can inter-operate with code written in any other CLR language.

3.2.3 winPcap

In [3], it is claimed that winPcap is the industry-standard tool for link-layer network access in Windows environments. WinPcap consists of a driver and a library that provide easy access to low-level network layers.

In [31] it is claimed that the main features of winPcap are the following:

- Freedom: there is a total freedom to modify winPcap and use it within any application, even commercial ones;

- High performance: there are classic optimizations, described in the packet capture literature and some original ones like just-in-time filter compilation and kernel-level statistic processing.

- Popularity: there is a vast list of tools – free and commercial ones – that use winPcap. Within this list there are some well known tools such as Wireshark, Nmap, Snort, WinDump and ntop. It is also claimed that winPcap has thousands of downloads per day;

- Reliability: with a rock-solid development approach, and through the contributions of many users, winPcap has grown to be a reliable and stable software;

- Ease of use: either in a programmers point of view and the final user perspective, winPcap is well documented and simple to setup and start to use;

- Portability: winPcap is compatible with libpcap [2], meaning current applications can b ported either to or from other operative systems, without the hassle of porting the capture library.

3.2.4 SharpPcap

In order to use winPcap with Microsoft .NET Framework, a wrapper needs to be used. This is due to the object oriented nature of the programming language used to develop *NetOdyssey*, namely C#. WinPcap is not object–oriented *per se*, but an object oriented approach increases the reliability and re–usability of *NetOdyssey*. At the time this dissertation was written, SharpPcap was open source software and it was freely available at [32].

3.3   The Calculation Philosophy of *NetOdyssey*

Equation 3.1, the Strong Law of Large Numbers, shows that, given enough observations (possibly infinite), the values from a random variable tend to the expected value (the average). This leads to a lost of information, because any variation in the observed values is absorbed and smoothed by all other values. In order to avoid this loss of information, *NetOdyssey* relies in analyzing all data in a windowed fashion. *NetOdyssey* supports the two following types of analysis windows:

- Analysis Window Size (AWS) – the analysis window is filled with $AWS = n$ observations and then analyzed. Once all results are calculated, the analysis window slides one observation to the right (this means the first observation is removed, and a new observation is inserted in the window), and this process repeats all over again (see Figure 3.1);

- Analysis Window Time (AWT) – the analysis window is filled with observations for $AWT = t$ seconds, and when it is full, it is analyzed.

Once all results are calculated, the analysis window is *cleared*, and this process repeats all over again (see Figure 3.2).

$$P \left(\lim_{n \to \infty} X_n = \mu \right) = 1 \tag{3.1}$$

NetOdyssey may gather the network packets themselves, one by one, or a statistical information provided by winPcap: Bit Count per Time Unit (BCTU). These two capture modes are not compatible with each other, because they require the network adapter to work in different capturing modes. When *NetOdyssey* is gathering BCTU information, it is not possible to access the packets themselves. This happens because a captured packet, by definition, has a well defined start time and size, but this does not allow us to know how long did it actually take to transmit/receive, or how many bits were generated by it, because of situations like re-transmits and malformed packets. Thus, in order to gather BCTU information, the adapter is requested to work in statistic gathering mode, which allows the adapter to return precise information about the amount of bits traveling through it.

Figure 3.1: A representation of a sliding analysis window (AWS of 3 observations).

In Figure 3.1 it is easily perceivable that in an analysis window of $AWS = n$ analysis, when there is a right-shift of the analysis window, $n - 1$ packets are common to the previous analysis window. If every window needed to be fully analyzed (although this may be the case for some analyses), this would lead to a slow and potentially not real-time analysis. In order to optimize this process, all of the analysis methods of *NetOdyssey* must be implemented with a *per input* and *per output* analysis in mind. When correctly implemented, this approach makes the analysis independent from

Figure 3.2: A representation of a temporal analysis window (AWT).

the size of the AWS (this does not apply to AWT, because after t seconds the analysis window is cleared (as seen in Figure 3.2)).

In a high-level description, *NetOdyssey* provides an analysis window constructor that works in the following manner:

– Until analysis window is *full* (definition of *full* varies according to analysis method: $AWS = n$ observations or $AWT = t$ seconds of observations):

  * Enqueue an observation (may be a network packet or a BCTU statistic) in the analysis window;

  * Send this observation to all analysis modules;

– When analysis window is *full*:

  * Request all analyzers to report their analysis from the current analysis window;

  * If analyzing in a sliding fashion (AWS): Dequeue an observation from the analysis window, sending this observation to all analysis modules;

  * If analyzing in a temporal fashion (AWT): Clear all observations from the analysis window, requesting all analysis modules to clear their analysis;

– Now that the analysis window is not *full*, repeat the process again.

3.4   The Architecture of *NetOdyssey*

NetOdyssey has been developed with expansion capability and easiness of use in mind. *NetOdyssey* was thought with a multi-thread approach, in order to allow it to take full advantage of current multi-core CPUs. The core behind *NetOdyssey* provides the tools and the threads responsible for compiling expansion modules, capturing network packets or BCTU statistics, constructing and managing the analysis window, reporting application status and delays through a Health Reporter and providing task queuing support methods for Expansion Modules.

- *Expansion Modules Compiler*

  Microsoft .NET Framework provides methods and classes that allow for *runtime* compilation of C# or VB.NET source code. As described in section 3.2.1, the CLR is the foundation of the .NET Framework, so these classes are able to compile plain-text code (determining syntax-errors and compiler warnings), and run this code inside the scope of the application. This is the same process as invoking a Dynamically Linked Library (DLL), though the only difference relies on the code being compiled *on-demand*, instead of being *pre-compiled*. This approach facilitates the openness philosophy behind *NetOdyssey*, allowing for all of those who utilize this framework to share their modules and allowing them to be updated, adapted and tested with different settings. The *NetOdyssey* class that provides this functionality is `clsModules` and it is described in Appendix A.1.4;

- *Capturer Thread*

  NetOdyssey relies on a thread to capture the observations that are going to be analyzed. It is important to run this capturing process in a separate thread, in order to minimize the number of dropped observations that may arise due to processing delays. As described in section 3.3, these observations may be the network packets themselves or BCTU statistics. From a high level point of view, the capturer thread is responsible for:*a*) configuring the capture device (*live* capture device or *off-line* capture device: a *pcap* file); *b*) capturing the

observations; *c*) sending each observation to the analysis window constructor thread. It is also this capturer thread that is responsible for verifying if the configured stopping conditions (time limit and/or observation count) have been reached. The *NetOdyssey* class that provides this functionality is `clsCapturer` and it is described in Appendix A.1.5;

– *Analysis Window Constructor Thread*

NetOdyssey constructs and manages the analysis window according to the configured settings in a separate thread. The capturer thread dispatches the observations to this analysis window constructor and this thread constructs the analysis window, sending the observations to the analysis modules and requesting them to report their analysis according to the algorithm described in section 3.3. The *NetOdyssey* class that provides this functionality is `clsAnalysisWindow` and it is described in Appendix A.1.6;

– *Health Reporter Thread*

NetOdyssey provides a thread that allows the monitoring of the status (also known as health) of other *NetOdyssey*'s threads. This mechanism is useful to understand if any loss of observations is happening or any analysis module is falling behind on it's analysis process. In order to be able to report it's health, a class must implement the interface `IHealthReporter`, described in Appendix A.1.9. The *NetOdyssey* class that provides this functionality is `clsHealthMonitor` and it is described in Appendix A.1.7;

– *Expansion Modules Base Threads*

Each of *NetOdyssey*'s analysis modules is an implementation of either interface `INetOdysseyBCTUAnalyzerModule` (described in Appendix A.2.4) or `INetOdysseyPacketAnalyzerModule` (described in Appendix A.2.5) and an extension of base class `NetOdysseyModuleBase` (described in Appendix A.2.1). This base class provides essential services, such as: *a*) threaded execution of the module; *b*) queuing of the analysis tasks the module must perform; *c*) streaming the module reports to a text file; *d*) reporting the module's

status, namely the size of it's task queue . It is essential to provide these services, in order to permit an easier utilization of *NetOdyssey*. For example, an user programming a *NetOdyssey* module does not have to worry about writing the analysis to a file: all the output from the analysis report method is automatically streamed to a file, one for each module. Thus, the module developer only has to worry about the format of his output, which can of course be adapted to his needs, e.g.: Comma Separated Values (CSV).

3.5   A Modular Approach

As stated in Chapter 1, *NetOdyssey* aims to become a well known network analysis framework.   One of the main advantages of *NetOdyssey* is presented in this section: modularity. This feature is possible due to a precise differentiation between the core of *NetOdyssey* and the analysis modules. While the core of *NetOdyssey* provides basic functionality and services, such as those described in section 3.4, it is the modules that are responsible for actually analyzing network information.

The analysis modules of *NetOdyssey* are nothing more than plain–text code files (C# or VB.NET) that get compiled during *NetOdyssey*'s start. Each module is a class that extends a base class and implements an analysis interface.   This base class is `NetOdysseyModuleBase` (described in Appendix A.2.1) and the currently available analysis interfaces are `INetOdysseyBCTUAnalyzerModule` (described in Appendix A.2.4) and `INetOdysseyPacketAnalyzerModule` (described in Appendix A.2.5).

Section 3.3 explains the philosophy behind the *per observation* analysis: with the windowed analysis approach of *NetOdyssey*, it is much more efficient to analyze only the observations that enter and exit the window instead of always analyzing the whole window. This is even more obvious for sliding analysis windows ($AWS = n$), where n is large.

The core services of *NetOdyssey* run in separate threads and each analysis module also run on it's own thread. This is required because there may be different modules performing an analysis and it is not desirable that mod–

ules delay each other. Thus, when extending `NetOdysseyModuleBase`, a module is actually gaining access to a set of the core services of *NetOdyssey*, such as threaded execution, task management and report streaming. Whoever wishes to implement a *NetOdyssey* analysis module must only worry about how to process the observations entering and leaving the analysis window. An example of a *NetOdyssey* analysis module is presented in subsection 3.6.

3.6 An Example Module

This section presents an example module. In this example, one wishes to analyze the *mean* and *standard deviation* of the network packet sizes. Thus, the analysis will be performed in a *per-packet* fashion (as opposed to a *per-BCTU* analysis).

In order to perform a *per-packet* analysis, this example module needs to implement the interface `INetOdysseyPacketAnalyzerModule` (Appendix A.2.5). All of this module is written in C# code.

```
class example : NetOdysseyModuleBase,
                INetOdysseyPacketAnalyzerModule
```

As described in Appendix A.2.5, `INetOdysseyPacketAnalyzerModule` requires the classes that implement it to implement the following methods and functions:

1. `void AnalysePacketIn(Packet Packet, int WindowSize)`
2. `void AnalysePacketOut(Packet Packet, int WindowSize)`
3. `void Clear()`
4. `string ModuleEnd()`
5. `string ModuleStart()`
6. `string ReportAnalysis()`

Function `string ModuleStart()` is invoked once, before *NetOdyssey* starts the analysis process. This function returns a string which is automatically added to the top of the module output file. In this example,

the output will be separated by a semi–colon. It is convenient to print the header names in the first line, so the implementation of `ModuleStart()` is as follows:

```
public override string ModuleStart() {
        return "average; stdDev" + Environment.NewLine;
}
```

Function `string ModuleEnd()` is invoked once, after the current *NetOdyssey* module finishes the analysis process. This function returns a string which is automatically added to the end of the module output file. In this example, there is no need for a special footer output, so the implementation of `ModuleEnd()` is as follows:

```
public override string ModuleEnd() {
        return "";
}
```

The analysis performed by this module resides inside the `AnalysePacketIn()` and `AnalysePacketOut()` methods. These are the methods responsible for performing the real–time analysis of observations (in the particular case of this module these observations are network packets), as they enter and leave the analysis window. In this module, the author seeks to know the average and standard deviation of the network packets' size. The equation for determining the average of a random variable ($\bar{x}$) is presented in 3.2 and the equation for determining the standard deviation of a random variable (σ) is presented in 3.3.

$$\bar{x} = \frac{1}{N} \sum_{i=1}^{N} x_i \tag{3.2}$$

$$\sigma = \sqrt{\frac{1}{N} \sum_{i=1}^{N} (x_i - \bar{x})^2} \tag{3.3}$$

Equation 3.2 is fairly simple to implement, but equation 3.3 needs to be approached in a different manner, in order to facilitate the windowed

analysis approach of *NetOdyssey*. Thus, re-writing equation 3.3, it is possible to obtain equation 3.4.

$$\sigma = \sqrt{\frac{1}{N} \sum_{i=1}^{N} x_i^2 - \bar{x}^2} \tag{3.4}$$

In order to optimize the execution of the module, the following variables are declared and initialized inside the scope of the module's class:

1. `int _packetLenght`
2. `int _currentCount = 0`
3. `double _sum = 0`
4. `double _sumOfSquares = 0`
5. `double _average`
6. `double _sigma`

Variable `_packetLenght` is an auxiliary variable that seeks to minimize the overhead of accessing inside the `Packet` object received in the analysis methods. Variable `int _currentCount` holds a counter to the current number of packets inside the analysis window. This variable represents N in equation 3.2. Variable `double _sum` holds the sum of the packets length inside the analysis window. This variable represents $\sum_{i=1}^{N} x_i$ in equation 3.2. Variable `double _sumOfSquares` holds the sum of the square value of the packets length inside the analysis window. This variable represents $\sum_{i=1}^{N} x_i^2$ in equation 3.3. Variables `double _average` and `double _sigma` are auxiliary variables that facilitate the output of the analysis function described ahead.

Method `void AnalysePacketIn(Packet Packet, int WindowSize)` is called every time a network packet enters the analysis window. The implementation of this method is as follows:

```
public override void AnalyzePacketIn(
  PacketDotNet.Packet Packet, int WindowSize) {
      _packetLenght =
            Packet.BytesHighPerformance.Length;
      _currentCount++;
```

```
        _sum += _packetLenght;
        _sumOfSquares += _packetLenght * _packetLenght;
}
```

Method `void AnalysePacketOut(Packet Packet, int WindowSize)` is called every time a network packet leaves the analysis window. The implementation of this method is as follows:

```
public override void AnalyzePacketOut(
  PacketDotNet.Packet Packet, int WindowSize) {
        _packetLenght =
                Packet.BytesHighPerformance.Length;
        _currentCount--;
        _sum -= _packetLenght;
        _sumOfSquares -= _packetLenght * _packetLenght;
}
```

Method `void Clear()` is called every time the network analysis window is cleared. This happens particularly in the case of a temporal analysis window ($AWT = t$ seconds). This method exists to provide additional optimization, since if the analysis window was cleared, there is no need to analyze the removal of each packet, just reset the analysis module altogether. The implementation of this method is as follows:

```
public override void Clear() {
        _currentCount = 0;
        _sum = 0;
        _sumOfSquares = 0;
}
```

Finally it is necessary to implement the `string ReportAnalysis()` function. This function is called every time the analysis window is full, in order to allow the method to output a string containing it's results of the analysis of the current window. This output is automatically appended to the module report file, as mentioned in section 3.4. The implementation of this function is as follows:

```
public override string ReportAnalysis() {
        if (_currentCount > 0)
                _average = _sum / _currentCount;
        else
                _average = 0;


        if (_currentCount > 1)
                _sigma = Math.Sqrt(
                        (_sumOfSquares / _currentCount) -
                        (_average * _average)
                );
        else
                _sigma = 0;


        return _average + "; " + _sigma +
                Environment.NewLine;
}
```

This module was tested with the following randomly generated packets (generated with the tool described in section 4.2):
{569; 153; 1188; 64; 768; 339; 892; 1357; 435; 384}.

The output generated by this module for a sliding analysis window of 5 packets (AWS= 5) is:

```
average; stdDev
548,4; 412,000291262033
502,4; 419,896463428784
650,2; 400,511872483201
684; 448,703465553811
758,2; 362,421522539708
681,4; 391,710403231775
```

Using equation 3.2 to verify the average calculation results:

$(569 + 153 + 1188 + 64 + 768)/5 = 548.4;$

$(153 + 1188 + 64 + 768 + 339)/5 = 502.4;$

$(1188 + 64 + 768 + 339 + 892)/5 = 650.2;$

$(64 + 768 + 339 + 892 + 1357)/5 = 684;$

$(768 + 339 + 892 + 1357 + 435)/5 = 758.2;$

$(339 + 892 + 1357 + 435 + 384)/5 = 681.4.$

As expected, all results match.

Using equation 3.4 to verify the standard deviation calculation results:

$\sqrt{(569^2 + 153^2 + 1188^2 + 64^2 + 768^2)/5 - 548,4^2} = 412.002913;$

$\sqrt{(153^2 + 1188^2 + 64^2 + 768^2 + 339^2)/5 - 502.4^2} = 419.8964634;$

$\sqrt{(1188^2 + 64^2 + 768^2 + 339^2 + 892^2)/5 - 650.2^2} = 400.5118724;$

$\sqrt{(64^2 + 768^2 + 339^2 + 892^2 + 1357^2)/5 - 684^2} = 448.7034656;$

$\sqrt{(768^2 + 339^2 + 892^2 + 1357^2 + 435^2)/5 - 758.2^2} = 362.4215225;$

$\sqrt{(339^2 + 892^2 + 1357^2 + 435^2 + 384^2)/5 - 681.4^2} = 391.7104032.$

As expected, all results match.

The full and pretty printed source code of this module is available in Appendix B.1.

3.7 Conclusion

The present section presented a framework whose purpose is to ease the process of statistically analyzing network traffic. The philosophy behind this framework allows for a windowed analysis approach. This windowed analysis lets us at the NMCG and all others who use this framework, to look at the traffic as if looking through a window. Observations slide through this window, entering and leaving one at a time, as in a queue.

Because it is not easy for one to predict which metrics she/he will be analyzing throughout her/his research, the presented framework, *NetOdyssey* allows for analysis modularity. This modular approach means that those who use *NetOdyssey* may effectively develop their own analysis modules tuning them to their personal analysis necessities.

Since *NetOdyssey* looks at the network as if looking through a window, it is important to analyze only the observations that *enter* and *leave* this analysis window, since those are the ones that bring new information to the metrics being analyzed. The user analysis modules of *NetOdyssey* must be implemented with this approach in mind. If this approach was not taken, one would need to analyze the whole window, for each entering and leaving packet, which for an analysis window of size n would mean $n - 1$ observations with duplicated analysis.

As of the actual implementation, *NetOdyssey* may fill the analysis windows with two types of observations: the network packets themselves and BCTU observations. Analyzing the network packets allows for researchers to look at headers and payloads, while analyzing BCTU observations allows for researchers to understand fluctuations in bandwidth usage and other metrics. Due to the incompatible nature of these two observation types (libPcap must be configured to capture in either packet mode or BCTU mode), it is not possible to analyze them both at the same time.

Chapter 4

Results and Validation

4.1 Validation of NetOdyssey

NetOdyssey supports capturing live packets on any available NIC (provided they are compatible with winPcap), but it also supports opening an offline capture. The source of the packets is independent from all the calculation methods, so in order to validate *NetOdyssey* and assure coherent and valid results, an offline capture was used. Also, in order to avoid potentially biased captures, a random capture generator was created. Whenever possible, all calculations presented in this chapter were performed by well-known and valid mathematical tools, such as the *R-project* tool [33].

4.2 Random Capture Generator

In order to have a statistically random and unbiased offline capture to aid the validation process of *NetOdyssey* and all the implemented modules, a random capture generator was created. This random capture generator utilizes the well-known *Mersenne-Twister* [34] algorithm to randomly generate different packet sizes and different inter–arrival times. Although it is possible to perform Deep Packet Inspection (DPI) with *NetOdyssey* (the payload of the packets is available to the modules), this is outside of the scope of this dissertation, and, based on this, only packet sizes and packet

inter-arrival times are considered here. This random capture generator has different parameters for minimum and maximum packets sizes, minimum and maximum inter-arrival times, capture size and random payload generation (otherwise payload will be full of zeros).

This random capture generator outputs all packet sizes and inter-arrival times to a plain text file, which can be used to validate the generator. Both packet sizes and packet inter-arrival times are compared against these randomly generated values, using WireShark [4], a well-known, impartial and scientifically valid tool. In order to produce statistically significant results, several captures of more than 500.000 packets were generated and validated.

Throughout the present dissertation and the process of validating *NetOdyssey* and its modules, it was used one randomly generated capture of 5.000 packets, with a minimum packet size of 64 bytes and maximum packet size of 1518 bytes (as recommended by STD0041 [35]). For the inter-arrival times, it was decided to use a minimum value of 0 (zero) and a maximum value of 2.000.000 microseconds (2 seconds).

In order to assess the degree of randomness of this generated capture, the entropy calculation for an AWS of 250 observations is presented in subsection 4.3.1, and in subsection 4.3.3, the auto-correlation function of this generated capture is included.

4.3   Implemented Modules

One of the contributions of this dissertation, mentioned in section 1.3, is the pack of analysis modules. These modules are presented and validated in this current section. In order to obtain a good degree of scientific validation, all the results provided by currently implemented modules were mathematically validated with a well-known and impartial mathematical tool: R [33]. The implemented modules available and explained in this dissertation are:

- Average and Standard Deviation estimator, described in section 3.6 (source code available in Appendix B.1);

  - Entropy estimator, described in section 4.3.1 (source code available in Appendix B.2);

  - Auto-Correlation estimator, described in section 4.3.3 (source code available in Appendix B.3);

  - Hurst parameter estimator, described in section 4.3.5 (source code available in Appendix B.4).

4.3.1   Entropy Estimator

Entropy (*H*) is the measure of the uncertainty of random variable. Equation 4.1 presents the definition of *Entropy*, also known as *Shannon Entropy*. The maximum value of the *Entropy* is the *logarithm* of the different number of possible observations.

$$H(X) = -\sum_{x \in \chi} p(x) \log p(x) \tag{4.1}$$

With the present dissertation, and, as stated in section 1.3, one of the analysis modules available is an entropy estimator module. Equation 4.1 is not directly implementable in an optimized way to perform a windowed analysis, as described in section 3.3. Thus, there were two possible approaches for an entropy analysis:

  - For every full analysis window, calculate the entropy of the whole window: this approach is not optimized at all and, as such, it was not implemented;

  - For every observation entering and leaving the window, calculate it's particular effect on the entropy of that window: this approach is optimized to work in accordance with *NetOdyssey*'s calculation philosophy, so it was implemented.

In the following explanation, please consider *w* as the analysis window size and *c* as the number of times that the current observation's value is present inside the analysis window. When an observation enters the analysis window, this module performs the following analysis:

- If $c = 0$:
    * $c = 1$;
    * Add $\frac{c}{w} * \ln(w)$ to the entropy;

- Else, if $c > 0$:
    * Subtract $\frac{c}{w} * \ln(\frac{w}{c})$ from the entropy;
    * $c = c + 1$;
    * Add $\frac{c}{w} * \ln(\frac{w}{c})$ to the entropy;

In a similar way, when an observation leaves the analysis window, this module performs the following analysis:

- If $c = 1$:
    * Remove $\frac{c}{w} * \ln(w)$ from the entropy;
    * $c = 0$;

- Else, if $c > 1$:
    * Subtract $\frac{c}{w} * \ln(\frac{w}{c})$ from the entropy;
    * $c = c - 1$;
    * Add $\frac{c}{w} * \ln(\frac{w}{c})$ to the entropy;

Thus, when reporting the analysis, this module only needs to output the current value of the entropy, because it is always up-to-date.

The main idea behind this algorithm is to update the value entropy only when observations enter and leave the observation window. If an observation value already existed inside the window, its previous effect on the entropy must be removed, before adding the new effect on entropy. This is observable in the algorithm above.

The validation of this module is presented in subsection 4.3.2, and the source code for this module is available in Appendix B.2.

4.3.2 Validation of Entropy Estimator

Figure 4.1 presents the R source code utilized to validate the Entropy calculation module available with the present dissertation.

```
1   function (aws,count) {
2       for(aux in seq(count-aws+1)) {
3           w <- packets[aux:(aux+aws-1)]
4           a <- round(entropy(table(w)),5)
5           b <- round(results[aux],5)
6           if (a != b) {
7               print(c("Failed in ",aux))
8               return(FALSE)
9           } else {}
10      }
11      return(TRUE)
12  }
```

Figure 4.1: R code used to validate results from Entropy Estimator module.

Entropy() is a function available in R, that estimates the Shannon entropy of a random variable from its observation counts. It is necessary to use the table() function, in order to return the observation counts of a random variable. By comparing all R entropy calculation results for a sub-vector of initial packet sizes, rounded to the 5th decimal place for different window sizes, and observing that all values match, it is possible to access the validity of the implementation of this module. As it possible to understand from the validation source code, in the first occurrence of a mismatch, the validation immediately stops and warns about the error. The execution of this validation ran flawlessly for different AWS sizes and origin values, thus proving that this module is correctly estimating the Entropy values.

In order to assess the degree of randomness of the captures generated by the tool presented in section 4.2, the *entropy* calculation for an AWS of 250 observations was calculated and the results are presented in Figure 4.2. As it is possible to visualize, the entropy values are always very close to the maximum value (ln(250) = 5.521), thus confirming the high-degree of randomness of this randomly generated capture.

4.3.3 Auto-Correlation Estimator

The auto-correlation function [36] is a commonly-used tool for assessing randomness in a data set. This randomness is measured by calculating

Figure 4.2: Entropy of 5.000 randomly generated packet sizes, with AWS=250.

the correlations of the values within the data set with each other. The distance of the packets utilized for determining the correlation is called *lag*, and is represented by k. Equation 4.2 presents the equation for the auto-correlation function of a random variable X, for a *lag k* and observation count w. When the data set has a high degree of randomness, the auto-correlation values are near to zero for all lag separations. Likewise, if there is a low degree of randomness, autocorrelation values will be far from zero. The auto-correlation function has a co-domain of $[-1{:}1]$.

$$acf(X, k) = \frac{\sum_{i=0}^{w-k} X_i X_{i+k} - \mu \sum_{i=0}^{w-k} X_i - \mu \sum_{i=0}^{w-k} X_{i+k} + (w - k)\mu^2}{\sum_{i=0}^{w} X_i^2 - w\mu^2} \qquad (4.2)$$

As an example of the auto-correlation function, Figure 4.3 presents the plot of the example equation 4.3. Figure 4.4 presents the plot of the auto-correlation function for a maximum lag of $K = 200$. For small values of K, the auto-correlation of the function values is evident. This is expected, because the origin vales have a very small degree of randomness, yet they

are not constant.

$$f(x) = 2\sin\left(2\pi\left(x - \frac{1}{4}\right)\right), x = [0, 2] \tag{4.3}$$

Figure 4.3: Plot of equation 4.3 .

Figure 4.4: Autocorrelation of equation 4.3, maximum lag $K = 200$.

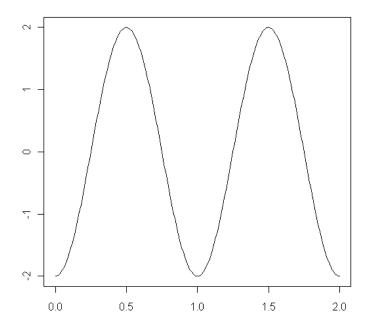In order to access the degree of randomness of the random capture gener-
ated in section 4.2, and utilized in the process of validating the analysis
modules available in this dissertation, the auto-correlation function for all
values of K is presented in Figure 4.5. It is possible to observe that
for all values $K > 0$ the value of the auto-correlation is very near to 0,
meaning a very good degree of randomness (auto-correlation of $K = 0$ is
1 by convention). This is the desired and expected behavior, because, as
stated in 4.2, a well-known and valid pseudo-random generator was used
to generate these values.

Figure 4.5: Autocorrelation of 5.000 randomly generated packet sizes, for all Ks.

With the present dissertation, and, as stated in section 1.3, one of the
analysis modules available is an auto-correlation estimator module. This
module was implemented in order to take advantage of the optimization the
analysis philosophy behind *NetOdyssey*, presented in section 3.3 provides.
To accomplish this, it was used the methods to analyze the observations
entering and leaving the analysis window to update all sum values and then
in the analysis report method calculate and return the result of equation 4.2.

Since it is desirable to calculate the auto-correlation function to different
lag values, there were two approaches:

  – Either create a module that would calculate the auto-correlation for

one specific *lag* and then run this module several times for different
*lag*s;

– Or create a module capable of calculating different *lag*s at the same
time.

The second approach was the one followed in this particular implementa-
tion, because as the number of *lag*s to calculate increases, so could the
overhead of running different concurrent *threads* (see section 3.5) at the
same time increase.

The validation of this module is presented in subsection 4.3.4, and the
source code for this module is available in Appendix B.3.

4.3.4   Validation of Auto–Correlation Estimator

Figure 4.6 presents the R source code utilized to validate the Auto–Correlation
calculation module available with the present dissertation.

```
1   function (aws, count, Ks) {
2       r <- 1
3       for(aux in seq(count-aws+1)) {
4           w <- packets[aux:(aux+aws-1)]
5           for (k in Ks) {
6               a <- round(acf(w,lag.max=k, plot=FALSE)$acf[k+1],5)
7               b <- round(results[r],5)
8               if (a != b) {
9                   print(c("Failed in k,aux,r", k, aux, r))
10                  return(FALSE)
11              } else {
12                  r <- r + 1
13              }
14          }
15      }
16      return(TRUE)
17  }
```

Figure 4.6: R code used to validate results from Auto–Correlation Estimator module.

Acf() is a function available in R that computes estimates of the auto-
correlation function. In the scope of this validation process, the arguments
that are important in this function are *lag.max=k* and *plot=FALSE*. These
arguments are important, because by default, the acf() function utilizes
$10\log_1 0(N)$, where N is the number of observations; *plot=FALSE* is a flag
that tells acf() not to present it's results in a plot, as it does by default.
Input argument Ks is a vector with the different values of k that will be

calculated. By comparing all R `acf()` calculation results for a sub-vector of initial packet sizes, rounded to the 5th decimal place for different window sizes, and observing that all values match, it is possible to access the validity of the implementation of this module. As it possible to understand from the validation source code, in the first occurrence of a mismatch, the validation immediately stops and warns about the error. The execution of this validation ran flawlessly for different AWS sizes and origin values, thus proving that this module is correctly estimating the Auto-Correlation values.

4.3.5   Hurst Exponent by Autocorrelation Function Estimator

In [37], Gubner and Kettani proposed a *new* method for the estimation of the Hurst parameter, naming it Hurst Exponent by Autocorrelation Function (HEAF). This proposal is based on the autocorrelation function $\gamma(k)$ of a self-similar process, which can be simplified as equation 4.4.

$$\gamma(k) = \frac{1}{2} \left(|k+1|^{2H} - 2|k|^{2H} + |k-1|^{2H} \right) \qquad (4.4)$$

Calculating the auto-correlation for $k = 1$, equation 4.4 degenerates into equation 4.5, which can be solved for H, thus providing an immediate estimate for the Hurst parameter, presented in equation 4.6.

$$\gamma(1) = 2^{2H-1} - 1 \qquad (4.5)$$

$$\Leftrightarrow H = \frac{1}{2} \log_2 (\gamma(1) + 1) + \frac{1}{2} \qquad (4.6)$$

These were the basic principles followed in the implementation of the Hurst parameter estimation module, available with the present dissertation. Because of the novelty of the method, this particular implementation has not yet been implement in R software, thus it is not possible to validate in the same fashion as the previous modules were validated. Nevertheless, and as it is possible to observe by comparing the source code of the

Auto-Correlation Estimation Module (Appendix B.3) and the source code of the Hurst Parameter Estimation Module (Appendix B.4), this module utilizes the already validated auto-correlation code, only expanding it on the `ReportAnalysis()` function. This extension is nothing more than the application of the Least Square Methods (LSMs) to the points with coordinates constituted by the base 2 logarithm of the values of lag k and the autocorrelation also for lag k.

The main idea behind this implementation of the Hurst Estimation is to plot the following values for, as a term of example an analysis window of 200:

x-axis: $\log_2(2) = 1$, $\log_2(4) = 2$, $\log_2(8) = 3$, $\log_2(16) = 4$, $\log_2(32) = 5$, $\log_2(64) = 6$, $\log_2(128) = 7$;

y-axis: $\log_2(acf(k = 2))$, $\log_2(acf(k = 4))$, $\log_2(acf(k = 8))$, $\log_2(acf(k = 16))$, $\log_2(acf(k = 32))$, $\log_2(acf(k = 64))$, $\log_2(acf(k = 128))$.

From this plot, it is possible to assess the *slope* of the line connecting these points, through the LSMs method. Once this *slope s* is estimated H is determinable by equation 4.7.
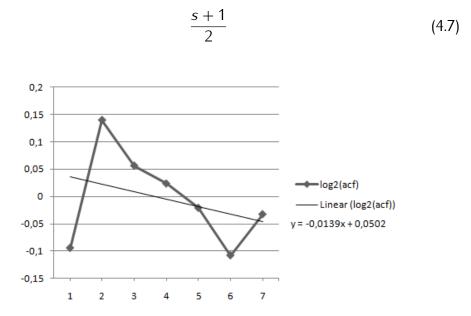
$$\frac{s + 1}{2} \tag{4.7}$$

Figure 4.7: Estimation of the Hurst parameter based on autocorrelation function using linear regression.

Figure 4.7 depicts an example of the plot for the results from the first AWS of 200 observations, from the random capture generated with the tool presented in 4.2. As it is possible to observe, the slope of the line from the linear regression of the values is close to 0 ($-0,0139$). Through equation 4.7 it is possible to determine that the Hurst parameter H is 0.49307. This value is very close to 0.5, meaning there is no long-range dependence between values in this particular observation window. When H is close to 1, there is strong evidence of long-range dependence (also known as long memory). Likewise, when H is close to 0, there is strong evidence of anti-persistence.

4.4   Conclusions

One of the main contributions of this dissertation is the pack of available user analysis modules. Throughout the present chapter, this pack of modules was presented and validated. These modules are: average and standard deviation calculation, entropy estimation, auto-correlation estimation and HEAF estimation. A random capture generator was developed, in order to provide some statistically random data (non biased), so these analysis modules can be validated.

This pack of modules has their C# source code available in Appendix B. It was important to us to provide the scientific community with this dissertation explaining these analysis modules and their source code, because this provides a basis for the development of more *NetOdyssey* modules. These modules depict the windowed analysis philosophy behind *NetOdyssey* and provide easiness to its understanding.

Everyone who implements a *NetOdyssey* analysis module should take special care in the validation process, because only with valid and correct results it is possible to obtain knowledge about network behavior. Section 5.2 presents some guidelines for those who wish to develop analysis modules for *NetOdyssey*.

Chapter 5

Conclusions and Future Work

5.1 Main Conclusions

In section 1.2 it was stated that this dissertation aims to facilitate the process of understanding any given network, while providing researchers with freedom in this understanding process. Chapter 2 presented the revision of most of the state of the art available at the time this dissertation was written. From this analysis of the state of the art, it was put in evidence the need for the development of a tool . Throughout the present dissertation, it was described the design and development of *NetOdyssey*, a framework for network traffic analysis, with a windowed analysis philosophy in mind (as described in section 3.3). This windowed analysis philosophy is one of the crucial differentiation points of *NetOdyssey* from other analysis frameworks. This framework has grown in such a way, that it could be used not only by the NMCG research group, but also by the scientific community in general. This lead to a change in the scope of this dissertation, that instead of focusing in traffic analysis itself, it focused on the development and validation of this framework. Throughout Chapter 3 and Chapter 4, the author presented and validated this framework, *NetOdyssey*. Throughout all sets of laboratory tests that the author submitted *NetOdyssey* to, its reliability increased notably. There is a great confidence in this tool, since it has been able to surpass all initial expectations.

The *NetOdyssey* framework is a very good addition for the ever grow–

ing community of researchers that wish to analyze network metrics and statistics. Its modular approach and object oriented development facilitate further expansions and improvements. The analysis modules developed and presented with the current dissertation also facilitate the understanding and implementation of further analysis modules. This is a crucial requisite in order to guarantee the future utilization and improvement of *NetOdyssey*.

Special care was taken while scientifically validating both *NetOdyssey* and the available analysis modules. This validation was possible thanks to a meticulous way of developing the analysis modules and then comparing the output of these modules against a well known and impartial statistical tool. The choice for using the R software [33] was due to the fact that it is indeed very useful, has a non-steep learning curve and it is freely available for everyone to use.

5.2   Directions for Future Work

While *NetOdyssey* has already proven to be a robust and reliable framework for network analysis, many improvements may still take place. Because of the open and modular approach inherent to *NetOdyssey*, researchers and developers, who wish to use this framework to perform their analysis, are able to expand and improve the functionalities of *NetOdyssey*.

Because user analysis modules need to be written in plain source code, they can be shared within the scientific and interested community. This is only possible, of course, if those who utilize this framework are willing to share their analysis modules, but there are great expectations for the future of *NetOdyssey*.

About the future work and progress of *NetOdyssey*, the following guidelines are proposed:

- Assessment of the statistical metric one wishes to analyze;

- Evaluation if this metric can be evaluated in a windowed manner (i.e., following the analysis philosophy described in section 3.3);

- Development of the analysis module, according to the documentation in this dissertation and other available modules;

- Validation of the results with a well known and impartial tool;

- Use the developed and validated module to perform the desired analysis, obtaining results and conclusions;

- Share of these results, conclusions and preferably also the analysis module with the scientific community.

NetOdyssey was developed with a good degree of object–orientation and class separation in mind. This allows for the development and extension of additional core functionalities. For example, if one wishes to analyze network flows, one may develop a capturer module that captures flows and sends them to the already developed analysis modules.

In short, the future work of *NetOdyssey* depends greatly on those who utilize it. Those who wish to implement analysis modules may focus on that scope, and those who wish to improve its core services and functionalities may focus on that scope. The author feels it will be a thrilling experience to watch *NetOdyssey* develop and mature even further.

References

[1] F. D. Beirão, J. V. Gomes, P. R. M. Inácio, M. Pereira, M. M. Freire. NetOdyssey - a new tool for real-time analysis of network traffic. *9th IEEE International Symposioum on Network Computing and Applications (IEEE NCA 2010), Cambridge, USA*, 15-17 July 2010, 4 pages.

[2] tcpdump/libpcap Web Page. http://www.tcpdump.org/ accessed February 22nd, 2010.

[3] CACE Technologies. WinPcap: The Windows Packet Capture Library. http://www.winpcap.org/ accessed February 22nd, 2010.

[4] CACE Technologies. Wireshark About Page. http://www.wireshark.org/about.html accessed March 3rd, 2010.

[5] Analyse-it. Analyse-it Web Page. http://www.analyse-it.com/ accessed March 8th, 2010.

[6] G. Iannaccone, C. Diot, D. McAuley, A. Moore, I. Pratt, L. Rizzo. The CoMo White Paper. September 2004.

[7] Politecnico di Torino. Analyzer 3.0 (alpha) Wep Page, November 2007. http://analyzer.polito.it/ accessed March 2nd, 2010.

[8] Politecnico di Torino. The NetBee Library Web Page. http://www.nbee.org/ accessed March 2nd, 2010.

[9] ntop.org. ntop Web Page. accessed March 16th, 2010.

[10] Network multimedia and computing group. http://floyd.di.ubi.pt/nmcg/ accessed June 25th, 2010.

[11] J. Postel. STD0007: Transmission Control Protocol. September 1981.

[12] J. Postel. STD0006: User Datagram Protocol. August 1980.

[13] Politecnico Di Torino Web Page. http://www.polito.it/ accessed March 2nd, 2010.

[14] The NetBee Library PDL. http://www.nbee.org/doku.php?id=netpdl:index accessed March 2nd, 2010.

[15] B. Wijnen D. Harrington, R. Presuhn. STD0062: An Architecture for Describing Simple Network Management Protocol (SNMP) Management Frameworks. December 2002.

[16] Ed. B. Claise. RFC3954: Cisco Systems NetFlow Services Export Version 9. October 2004.

[17] Cisco. http://www.cisco.com/en/US/products/ps6601/ products_ios_protocol_group_home.html accessed March 9th, 2010.

[18] Ed. R. Stewart. RFC:4960: Stream Control Transmission Protocol. September 2007.

[19] Ed. B. Claise. RFC5101: Specification of the IP Flow Information Export (IPFIX) Protocol for the Exchange of IP Traffic Flow Information. January 2008.

[20] B. Claise S. Zander J. Quittek, T. Zseby. RFC3917: Requirements for IP Flow Information Export (IPFIX). October 2004.

[21] L. Mark T. Zseby A. Wagner B. Trammell, E. Boschi. RFC5655: Specification of the IP Flow Information Export (IPFIX) File Format. October 2009.

[22] Q. Xie M. Tuexen P. Conrad R. Stewart, M. Ramalho. Stream Control Transmission Protocol (SCTP) Partial Reliability Extension. May 2004.

[23] E. Rescorla T. Dierks. RFC4346: The Transport Layer Security (TLS) Protocol Version 1.1. April 2006.

[24] N. Modadugu E. Rescorla. RFC4347: Datagram Transport Layer Security. April 2006.

[25] T. Oetiker. RRDTool Web Page. http://oss.oetiker.ch/rrdtool/index.en.html accessed March 17th, 2010.

[26] gnuplot. gnuplot Wep Page. http://www.gnuplot.info/ accessed March 17th, 2010.

[27] MSDN. .NET Framework Conceptual Overview. http://msdn.microsoft.com/en-us/library/zw4w595w.aspx accessed February 22nd, 2010.

[28] Mono-Project Web Page. http://www.mono-project.com/ accessed February 22nd, 2010.

[29] ECMA International. ECMA Web Page. http://www.ecma-international.org/ accessed February 22nd, 2010.

[30] Novell Web Page. http://www.novell.com/ accessed February 22nd, 2010.

[31] CACE Technologies. WinPcap features. http://www.winpcap.org/misc/features.htm accessed February 22nd, 2010.

[32] SharpPcap Sourceforge Project Page. http://sourceforge.net/projects/sharppcap/ accessed February 22nd, 2010.

[33] Bell Laboratories. R-project. http://www.r-project.org/ accessed April 19th, 2010.

[34] M. Matsumoto, T. Nishimura. Mersenne twister: a 623-dimensionally equidistributed uniform pseudo-random number generator. *Special issue on uniform random number generation*, 8:3–30, 1998.

[35] C. Hornig. STD0041: A Standard for the Transmission of IP Datagrams over Ethernet Networks. April 1984.

[36] G. E. P. Box and G. Jenkins. Time series analysis: Forecasting and control. 1976.

[37] J.A. Kettani, H.; Gubner. A novel approach to the estimation of the long-range dependence parameter. *IEEE Transactions on Circuits and Systems II: Express Briefs*, 53:463–467, 2006.

Appendix A

Class Model

A.1 Base Classes of NetOdyssey

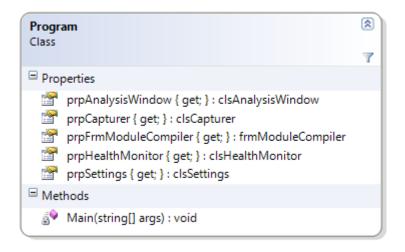### A.1.1 Program

Figure A.1: `Program` – The main class containing the `main()` method.

`Program` (Figure A.1) is the default class where the `main()` method resides. This class contains the following properties:

- *prpAnalysisWindow*: this property contains the instance of `clsAnalysisWindow` (see A.1.6);
- *prpCapturer*: this property contains the instance of `clsCapturer` (see A.1.5);

59

– *prpFrmModuleCompiler*: this property contains the instance of the GUI form that contains the instance of `clsModules` (see A.1.4);

– *prpHealthMonitor*: this property contains the instance of `clsHealthMonitor` (see A.1.7);

– *prpSettings*: this property contains the instance of `clsSettings` (see A.1.2).

In a no error execution, the `main()` method goes through the following steps:

1. read and parse command-line settings;
2. check current settings;
    – if current settings are not valid or incompatible, show `frmSettings`(see A.1.3);
3. compile user modules;
4. start user modules;
5. start *AnalysisWindow* thread;
6. start *HelathMonitor* thread;
7. start Capturer thread;
8. start capturing and analysis process;
9. wait for user input, or wait for stopping conditions to be reached, if set;
    – when there is any user input, request Capturer thread to stop;
10. gracefully terminate application

A.1.2   clsSettings

The class `clsSettings` (Figure A.2) is responsible for holding all of *NetOdyssey*'s settings. The description of these settings is also available if *NetOdyssey* is started with -h command-line flag.

The available settings and their command-line flags are as follows:

– *AnalysisWindowSize* -aws=*n*: the size in packets or statistics of the AWS;

– *AnalysisWindowTime* -awt=*t*: the number of seconds of the AWT;

- *AutoStartCapture* `-asc`: automatically start capture if all settings are valid (will show compile results window anyway);
- *BitCountPerTimeUnit* `-bctu=`*t*: the number of milliseconds between BCTU readings;
- *CaptureDevice* `-d=`*d*: the zero-based index of the device to capture on; use flag `-ld` to list available devices;
- *CaptureMode*: an internal property to facilitate the differentiation between packet capturing and gathering of statistics;
- *DumpCapture* `-dump`: a flag to dump the capture to an offline file, allowing for posterior processing and further analysis; dumping the capture is not available in statistics gathering mode;
- *HealthMonitorInterval* `-hmi=`*t*: the number of milliseconds between health monitor reports;
- *IsStopScheduled*: an internal property set to true when there is any kind of capture stop scheduled (time or packet count);
- *ListDevices* `-ld`: a flag signaling whether devices will be listed or not;
- *ModulesFolder* `-modulesFolder=`*string*: the path for the folder from where user modules will be compiled;
- *Packets_StatisticsToCapture* `-c=`*n*: the amount of packets or statistics (depending on capture mode) to capture before automatically stopping the analysis;
- *PrintHelp* `-h`: a flag signaling whether help will be printed or not;
- *prpArguments*: an internal list of parsed command-line arguments;
- *prpDevices*: an internal list of `PcapDevice`s;
- *prpDumpFile*: an internal string containing the default path for the dump file;
- *RealTimePriority* `-rtp`: a flag indicating that *NetOdyssey* should try to raise it's priority to *real-time*;
- *ReportsFolder* `-reportsFolder=`*string*: the path for the folder to where user modules' reports will be written;
- *SecondsToCapture* `-t=`*t*: number of seconds to capture, before automatically stopping the analysis;
- *ShowCompileWindow* `-nsc`: a flag signaling whether or not to show

the compile results window;

- *TcpDumpFilter* `-filter=`*string*: a string written with the syntax of *tcpdump* to filter the captured packets;
- *Verbose* `-v`: a flag signaling whether *NetOdyssey* will produce verbose output.

`clsSettings` also provides the following methods:

- `checkSettings()`: a method that checks all settings for invalid or incompatible configurations;
- `createReportsFolder()`: a method that verifies the existence of the given report folder and creates it, if possible;
- `dumpSettings()`: a method that will output the current settings to the reports folder, for future reference.

A.1.3 frmSettings

`frmSettings` (Figure A.3) is shown either when the current settings are invalid or incompatible (e.x. if both AWS and AWT are set) or if the flag *AutoStartCapture* (see A.1.2) is not set. The default settings include *AutoStartCapture* set to false, thus `frmSettings` is shown if *NetOdyssey* is run with not command-line options.

A.1.4 clsModules

The abstract class `clsModules` (Figure A.4) provides the method `compileModules()` which is responsible for looking for `*.cs` and `*.vb` files inside `inSourceDirectory` and compile them. Once compiled, these modules can be accessed through the public field `prpModules`. The type of this public field is *List* of `NetOdysseyModuleBase` (see A.2.1). This class is utilized by the `main()` method inside `Program` class.

In general guidelines, the process of compiling a *NetOdyssey* user module written in C# is as follows:

- Create an array `_ra` of *String*s with the necessary refer-
  ence assemblies, i.e.: `{"System.dll"`, `"SharpPcap.dll"`,
  `"NetOdysseyModule.dll"`, `"PacketDotNet.dll"}`
- Instantiate an object `_cscp` of type
  `Microsoft.CSharp.CSharpCodeProvider`;
- Instantiate an object `_cp` of type
  `System.CodeDom.Compiler.CompilerParameters`, passing
  `_ra` as argument;
- Instantiate an object `_cr` of type
  `System.CodeDom.Compiler.CompilerResults` from
  `CSharpCodeProvider.CompileAssemblyFromFile()`,
  passing `_cp` and the full path to the file to be compiled as arguments;
    * Collection `_cr.Errors` contains all compile errors and warn-
  ings;
- Instantiate an object `_a` of type `System.Reflection.Assembly`
  from `_cr.CompiledAssembly`;
- For each object `_t` of type `System.Type` from `_a.GetTypes()`:
    * If `_t.IsClass` and
  `_t.IsSubclassOf(typeof(NetOdysseyModuleBase))`:
        * Instantiate an object `_module` of
  type `NetOdysseyModuleBase` by casting
  `System.Activator.CreateInstance(_t)` to
  `NetOdysseyModuleBase`;

The compiled user module, of type `NetOdysseyModuleBase` is now
accessible through `_module`. `clsModules` places all instances of user
modules inside the list `prpModules`.

A.1.5   clsCapturer

When `clsCapturer` (Figure A.5) is instantiated, it receives a `PcapDevice`
(an object that represents a winPcap device). This `PcapDevice` may be
a *Live* device or an *Offline* device.

This class implements the interface `IHealthReporter` (see A.1.9), mean-
ing it is able to report it's status through the method `HealthReport()`.

In this health report, `clsCapturer` outputs the following informations, according to their applicability: total captured packets/gathered statistics, total packets/statistics to be captured and adapter statistics, such as dropped packets.

When the method `Start()` is called, a new thread of the method `Work()` is launched. If the capture has a temporal limit, a new thread of the method `WorkthrCapturerStop()` is launched. This last thread sleeps for the amount of time of the capture temporal limit, and when this time has elapsed this thread stops the `Work()` thread.

When working with BCTU statistics, these informations are processed asynchronously, because it is the *Pcap* driver who provides *NetOdyssey* with this data. Thus, the method `prpLiveDevice_OnPcapStatistics()` is called every time a BCTU reading is available.

The method `Work()` works in the following manner:

- *if capturing packets*: prepare the capture device, applying filters and preparing dump files, if applicable. Then, in an endless loop (terminated by `main()` or by reaching stopping conditions):
    * Capture a packet;
    * Parse the packet (convert it from a *PacketDotNet.RawPacket* to a *PacketDotNet.Packet*);
    * Enqueue this packet in the analysis window;
    * If applicable, dump the packet;
- *if gathering BCTU statistics*: prepare the capture device, setting it's capture mode to Statistics and then setting the statistics callback method to `prpLiveDevice_OnPcapStatistics()`. The device is then requested to start the capture and this thread terminates, because the capture will run in a thread inside winPcap (this is why we need to utilize the callback method).

When stopping conditions are met (user input, packets captured or statistics gathered reach count limit or defined time has elapsed, whichever comes first), a `null` element is enqueued in the analysis window queue, signaling it to stop.

A.1.6   clsAnalysisWindow

The class `clsAnalysisWindow` (Figure A.6) is responsible for queuing either network packets or network statistics and send each of these items to all modules.

When the method `Start()` is called, a new thread of the method `Work()` is launched. If AWT is being used, a new thread of the method `AWTWork()` is launched. This thread runs in an infinite loop, sleeping for $AWT = t$ seconds, and when it awakes, it requests all modules to report their analysis and *clears* the current analysis window. (See subsection 3.3 for better understanding of the AWT analysis mechanism).

`clsAnalysisWindow` has an *input queue*, because `clsCapturer` (see A.1.5) does not contain any queuing mechanism. This *input queue* helps minimize the loss of network packets/statistics due to delays in handling and processing this data. The method `Enqueue()` (there are two signatures according to whether *NetOdyssey* is capturing packets or BCTU statistics) is called by `clsCapturer` and simply enqueues the information in the aforementioned *input queue*. A semaphore is also used to notify the `Work()` thread that there is new information available.

The method `Work()` works in the following manner, inside an endless loop (terminated when a `null` element is found in the *input queue*):

  – Read information *i* from *input queue*;
       * If *i* is `null`, this means that a stopping order has been given, so request all user modules to finish their analysis and terminate the current thread;
       * Else, place *i* inside the `analysisWindow` queue, requesting all user modules to analyze *i*;
       * If `analysisWindow` has reached the maximum $AWS = n$ size then:
            Request all user modules to report their current analysis;
            Remove information *t* from the tail of the `analysisWindow` queue (dequeue);
            Request all modules to remove *t* from their analysis;

A.1.7   clsHealthMonitor

The class `clsHealthMonitor` (Figure A.7) is responsible for request-
ing current status (health) from classes that implement the interface
`IHealthReporter` (see A.1.9). This class receives the instances of the
objects to monitor through the method `addModule()` and maintains a
thread (method `Work()`) that sleeps in an endless loop for a parameterized
time (see refclsSettings, parameter *HealthMonitorInterval*). Every time this
thread wakes up, it requests all modules to report their status (health) and
prints the outputs to *stdout*.

This class is very useful to determine if any user module is having pro-
cessing the inputs slower than expected and to know if packets are being
dropped.

A.1.8   clsMessages

The abstract class `clsMessages` (Figure A.8) contains the methods for
printing messages to the *stdout*. This class exists in order to facilitate the
future internationalization of *NetOdyssey*.

A.1.9   IHealthReporter

`IHealthReporter` (Figure A.9) is the interface that all classes which
support reporting their current status (health) must implement. This inter-
face is easy to implement, since it only mandates the implementation of the
`HealthReport()` method, which returns a *string*. It is up to each class
that impleme

A.2 Base Classes of the User Modules of NetOdyssey

A.2.1 NetOdysseyModuleBase

`NetOdysseyModuleBase` (Figure A.10) is the base class for every user module, thus providing all basic functionalities, such as threaded running and task queuing. `NetOdysseyModuleBase` has an internal queue of `NetOdysseyModuleBaseTask` (see A.2.2), so each module may run at it's own pace.

All user modules are compiled by `clsModules` (see A.1.4) and are instantiated in `clsModules.prpModules` list. The following methods are `virtual`, meaning they can be overridden by a later re-implementation of those methods:

```
- void AnalyzeBCTUIn(ulong BCTU, int WindowSize);
- void AnalyzeBCTUOut(ulong BCTU, int WindowSize);
- void AnalyzePacketIn(ulong BCTU, int WindowSize);
- void AnalyzePacketOut(ulong BCTU, int WindowSize);
- void Clear();
- void ModuleStart();
- void ModuleEnd();
- void ReportAnalysis();
```

When writing an analysis module, one should implement these methods with the `override` keyword, in order to override the default implementations. These default implementations simply print a warning message saying these methods were not re-implemented.

A.2.2 NetOdysseyModuleBaseTask

`NetOdysseyModuleBase.Task` (Figure A.11) is a class that provides a mechanism to store the arguments and actions that need to be queued. This class has several instantiation methods, which are called according to the action *NetOdyssey* wishes to enqueue.

A.2.3   NetOdysseyModuleBaseModuleTask

`NetOdysseyModuleBase.ModuleTask` (Figure A.12) is an enumerator
with the possible actions to enqueue with `NetOdysseyModuleBase.Task`
(see A.2.2). This `enum` exists in order to facilitate the object–oriented
approach, instead of utilizing a not strongly typed mechanism, such as
strings or integers to identify these actions.

A.2.4   INetOdysseyBCTUAnalyzerModule

`INetOdysseyBCTUAnalyzerModule` (Figure A.13) is the interface that
must be implemented when programming a user module that will ana-
lyze BCTU observations. The main difference between this interface and
`INetOdysseyPacketAnalyzerModule` (see A.2.5) is the name and
input type of the *AnalyzeIn* and *AnalyzeOut* methods.

A.2.5   INetOdysseyPacketAnalyzerModule

`INetOdysseyPacketAnalyzerModule` (Figure A.14) is the interface
that must be implemented when programming a user module that will
analyze packet observations. The main difference between this interface
and `INetOdysseyBCTUAnalyzerModule` (see A.2.4) is the name and
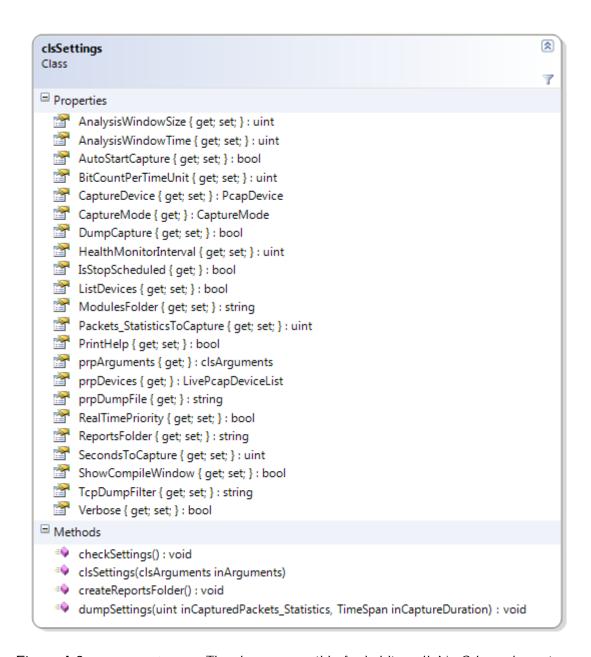input type of the *AnalyzeIn* and *AnalyzeOut* methods.

clsSettings
Class

Properties

- AnalysisWindowSize { get; set; } : uint
- AnalysisWindowTime { get; set; } : uint
- AutoStartCapture { get; set; } : bool
- BitCountPerTimeUnit { get; set; } : uint
- CaptureDevice { get; set; } : PcapDevice
- CaptureMode { get; } : CaptureMode
- DumpCapture { get; set; } : bool
- HealthMonitorInterval { get; set; } : uint
- IsStopScheduled { get; } : bool
- ListDevices { get; set; } : bool
- ModulesFolder { get; set; } : string
- Packets_StatisticsToCapture { get; set; } : uint
- PrintHelp { get; set; } : bool
- prpArguments { get; } : clsArguments
- prpDevices { get; } : LivePcapDeviceList
- prpDumpFile { get; } : string
- RealTimePriority { get; set; } : bool
- ReportsFolder { get; set; } : string
- SecondsToCapture { get; set; } : uint
- ShowCompileWindow { get; set; } : bool
- TcpDumpFilter { get; set; } : string
- Verbose { get; set; } : bool

Methods

- checkSettings() : void
- clsSettings(clsArguments inArguments)
- createReportsFolder() : void
- dumpSettings(uint inCapturedPackets_Statistics, TimeSpan inCaptureDuration) : void

Figure A.2: `clsSettings` – The class responsible for holding all *NetOdyssey*'s settings.
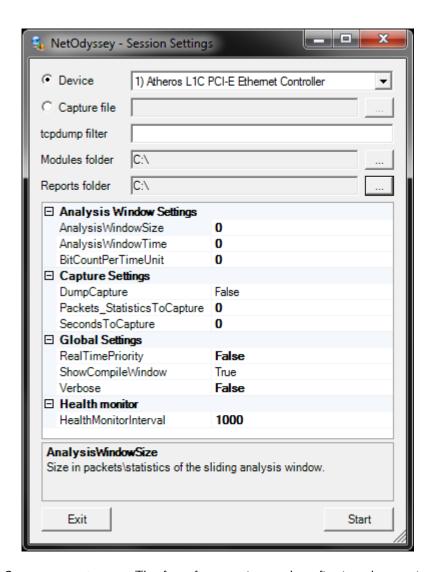
Figure A.3: `frmSettings` – The form for entering and confirming the session settings.
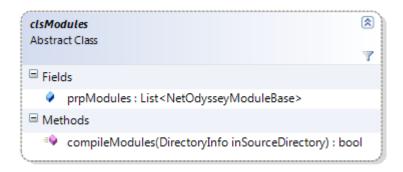

Figure A.4: `clsModules` – The class responsible for compiling `*.cs` and `*.vb` files.
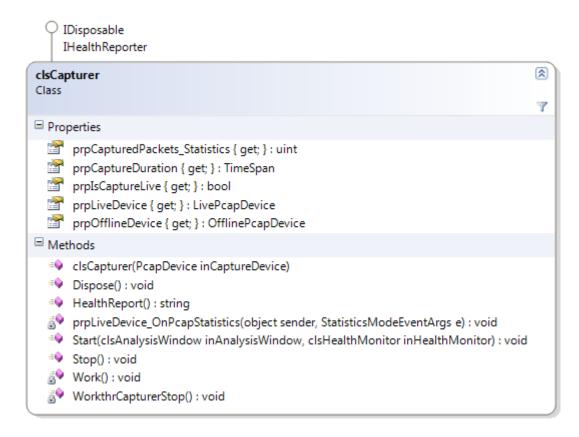
IDisposable
IHealthReporter

clsCapturer
Class

Properties
- prpCapturedPackets_Statistics { get; } : uint
- prpCaptureDuration { get; } : TimeSpan
- prpIsCaptureLive { get; } : bool
- prpLiveDevice { get; } : LivePcapDevice
- prpOfflineDevice { get; } : OfflinePcapDevice

Methods
- clsCapturer(PcapDevice inCaptureDevice)
- Dispose() : void
- HealthReport() : string
- prpLiveDevice_OnPcapStatistics(object sender, StatisticsModeEventArgs e) : void
- Start(clsAnalysisWindow inAnalysisWindow, clsHealthMonitor inHealthMonitor) : void
- Stop() : void
- Work() : void
- WorkthrCapturerStop() : void

Figure A.5: `clsCapturer` – The class responsible for capturing network packets or statistics, according to the analysis mode.

IDisposable
IHealthReporter

clsAnalysisWindow
Class

Methods
- AWTWork() : void
- clsAnalysisWindow()
- Dispose() : void
- Enqueue(Packet inPacket) : void
- Enqueue(ulong inBCTU) : void
- HealthReport() : string
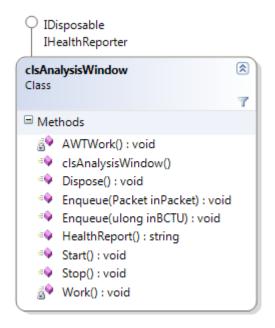- Start() : void
- Stop() : void
- Work() : void

Figure A.6: `clsAnalysisWindow` – The class responsible for queuing values in a windowed manner, and sending them to user modules.
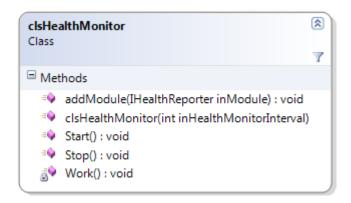
Figure A.7: `clsHealthMonitor` – The class responsible for requesting the current status of *NetOdyssey*'s threads, from time to time.
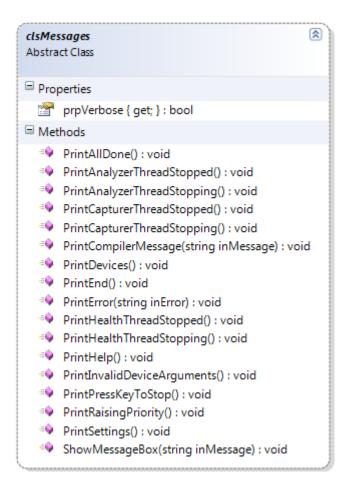

Figure A.8: `clsMessages` – The abstract class responsible for printing *NetOdyssey*'s outputs to *stdout*.

Figure A.9: `IHealthReporter` – The interface that must be implemented by classes who are able to report their current status (health).
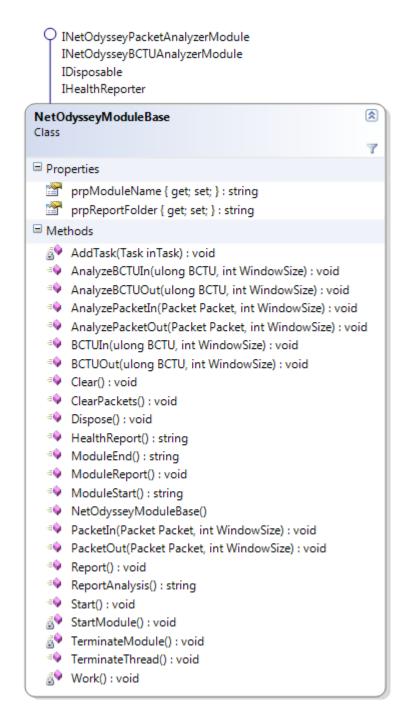
○ INetOdysseyPacketAnalyzerModule
   INetOdysseyBCTUAnalyzerModule
   IDisposable
   IHealthReporter

NetOdysseyModuleBase
Class
⊟ Properties
   🔧 prpModuleName { get; set; } : string
   🔧 prpReportFolder { get; set; } : string
⊟ Methods
   AddTask(Task inTask) : void
   AnalyzeBCTUIn(ulong BCTU, int WindowSize) : void
   AnalyzeBCTUOut(ulong BCTU, int WindowSize) : void
   AnalyzePacketIn(Packet Packet, int WindowSize) : void
   AnalyzePacketOut(Packet Packet, int WindowSize) : void
   BCTUIn(ulong BCTU, int WindowSize) : void
   BCTUOut(ulong BCTU, int WindowSize) : void
   Clear() : void
   ClearPackets() : void
   Dispose() : void
   HealthReport() : string
   ModuleEnd() : string
   ModuleReport() : void
   ModuleStart() : string
   NetOdysseyModuleBase()
   PacketIn(Packet Packet, int WindowSize) : void
   PacketOut(Packet Packet, int WindowSize) : void
   Report() : void
   ReportAnalysis() : string
   Start() : void
   StartModule() : void
   TerminateModule() : void
   TerminateThread() : void
   Work() : void

Figure A.10: `NetOdysseyModuleBase` – The class responsible for providing all the basic methods for a user module.
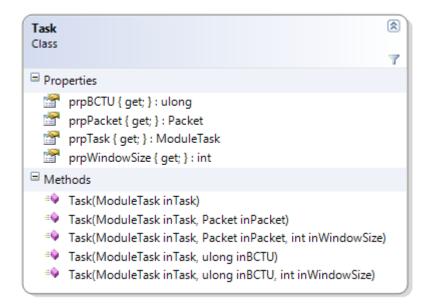
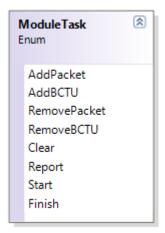**Figure A.11:** `NetOdysseyModuleBaseTask` – The class that holds a *NetOdyssey* module task.

Figure A.12: `NetOdysseyModuleBaseModuleTask` – The *enum* that represents the type of possible`NetOdysseyModuleBaseTask`s.
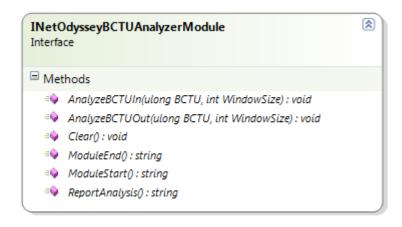
INetOdysseyBCTUAnalyzerModule
Interface

⊟ Methods
  ≡● *AnalyzeBCTUIn(ulong BCTU, int WindowSize) : void*
  ≡● *AnalyzeBCTUOut(ulong BCTU, int WindowSize) : void*
  ≡● *Clear() : void*
  ≡● *ModuleEnd() : string*
  ≡● *ModuleStart() : string*
  ≡● *ReportAnalysis() : string*

Figure A.13: `INetOdysseyBCTUAnalyzerModule` – The interface that must be implemented by user modules that perform a BCTU analysis.

INetOdysseyPacketAnalyzerModule
Interface

⊟ Methods
  ≡● *AnalyzePacketIn(Packet Packet, int WindowSize) : void*
  ≡● *AnalyzePacketOut(Packet Packet, int WindowSize) : void*
  ≡● *Clear() : void*
  ≡● *ModuleEnd() : string*
  ≡● *ModuleStart() : string*
  ≡● *ReportAnalysis() : string*

Figure A.14: `INetOdysseyPacketAnalyzerModule` – The interface that must be implemented by user modules that perform a *per-packet* analysis.

Appendix B

Implemented Modules Source Code

B.1 Average and Standard Deviation Estimator

```csharp
 1   using System;
 2   using System.Collections.Generic;
 3   using NetOdysseyModule;
 4
 5   namespace AverageAndStdDev
 6   {
 7       class NetOdysseyAvgStdDevPacketsModule :
 8           NetOdysseyModuleBase, INetOdysseyPacketAnalyzerModule {
 9
10           int _packetLenght;
11           int _currentCount = 0;
12           double _sum = 0;
13           double _sumOfSquares = 0;
14           double _average = 0;
15           double _sigma = 0;
16
17           public override string ModuleStart()
18           {
19               return "average; stdDev" + Environment.NewLine;
20           }
21
22           public override string ModuleEnd()
23           {
24               return "";
25           }
26
27           public override void AnalyzePacketIn(PacketDotNet.Packet Packet,
28                                             int WindowSize) {
29               _packetLenght = Packet.BytesHighPerformance.Length;
30               _currentCount++;
31               _sum += _packetLenght;
32               _sumOfSquares += _packetLenght * _packetLenght;
33           }
34
35           public override void AnalyzePacketOut(PacketDotNet.Packet Packet,
36                                              int WindowSize) {
37               _packetLenght = Packet.BytesHighPerformance.Length;
38               _currentCount--;
39               _sum -= _packetLenght;
40               _sumOfSquares -= _packetLenght * _packetLenght;
41           }
42
43           public override void Clear() {
44               _currentCount = 0;
45               _sum = 0;
46               _sumOfSquares = 0;
47           }
48
49           public override string ReportAnalysis() {
50               if (_currentCount > 0)
51                   _average = _sum / _currentCount;
52               else
53                   _average = 0;
54
55               if (_currentCount > 1)
56                   _sigma = Math.Sqrt(
57                       (_sumOfSquares / _currentCount) -
58                       (_average * _average)
59                   );
```

```
60              else
61                  _sigma = 0;
62              return _average + "; " + _sigma + Environment.NewLine;
63          }
64      }
65  }
```

B.2   Entropy Estimator

```csharp
1    using System;
2    using System.Collections.Generic;
3    using NetOdysseyModule;
4
5    namespace EntropyOnTheFly
6    {
7        class NetAnalyzerEntropyPacketsModule :
8                NetOdysseyModuleBase, INetOdysseyPacketAnalyzerModule {
9
10           int PacketLength;
11           double _ws;
12           double _entropy = 0;
13
14           Dictionary<int, int> _occurences = new Dictionary<int, int>();
15
16           public override string ModuleStart()
17           {
18               return "entropy" + Environment.NewLine;
19           }
20
21           public override string ModuleEnd()
22           {
23               return "";
24           }
25
26           public override void AnalyzePacketIn(PacketDotNet.Packet Packet,
27                                                 int WindowSize) {
28               PacketLength = Packet.BytesHighPerformance.Length;
29               _ws = (double)WindowSize;
30               lock (_occurences)
31               {
32                   if (_occurences.ContainsKey(PacketLength)) {
33                       // If this packet size already exists,
34                       // remove its previous weight and add
35                       // the new one to the entropy
36                       _entropy -= (_occurences[PacketLength] / _ws) *
37                               Math.Log(_ws / _occurences[PacketLength]);
38                       _occurences[PacketLength]++;
39                       _entropy += (_occurences[PacketLength] / _ws) *
40                               Math.Log(_ws / _occurences[PacketLength]);
41                   }
42                   else {
43                       // If this packet size didn't exist already,
44                       // add it to the entropy
45                       _occurences.Add(PacketLength, 1);
46                       _entropy += (1 / _ws) * Math.Log(_ws);
47                   }
48               }
49           }
50
51           public override void AnalyzePacketOut(PacketDotNet.Packet Packet,
52                                                 int WindowSize) {
53               PacketLength = Packet.BytesHighPerformance.Length;
54               _ws = (double)WindowSize;
55               lock (_occurences)
56               {
57                   if (_occurences[PacketLength] == 1) {
58                       // If this is the last occurence of this packet size,
59                       // remove it from the entropy
```

```
60                      _occurences.Remove(PacketLength);
61                      _entropy -= (1 / _ws) * Math.Log(_ws);
62                  }
63                  else {
64                      // If this packet size still exists in the window,
65                      // update its value
66                      _entropy -= (_occurences[PacketLength] / _ws) *
67                              Math.Log(_ws / _occurences[PacketLength]);
68                      _occurences[PacketLength]--;
69                      _entropy += (_occurences[PacketLength] / _ws) *
70                              Math.Log(_ws / _occurences[PacketLength]);
71                  }
72              }
73          }
74
75          public override void Clear() {
76              _entropy = 0;
77              lock (_occurences)
78                  _occurences.Clear();
79          }
80
81          public override string ReportAnalysis() {
82              return _entropy + ";" + Environment.NewLine;
83          }
84      }
85  }
```

B.3   Auto-Correlation Estimator

```csharp
1   using System;
2   using System.Collections.Generic;
3   using NetOdysseyModule;
4
5   namespace AutoCorrelations
6   {
7       class NetOdysseyACFPacketsModule :
8           NetOdysseyModuleBase, INetOdysseyPacketAnalyzerModule
9       {
10          int _Kcount; // check method ModuleStart for K generation
11          List<int> _Ks = new List<int>();
12
13          List<Queue<int>> _awPackets = new List<Queue<int>>();
14          List<Queue<int>> _XiPackets = new List<Queue<int>>();
15          List<Queue<int>> _nextKPackets = new List<Queue<int>>();
16          List<List<int>> _XikPackets = new List<List<int>>();
17          List<Queue<int>> _XiXikProducts = new List<Queue<int>>();
18
19          int _packetLenght = 0;
20          int _currentCount = 0;
21          int _K = 0;
22
23          List<int> _XSum = new List<int>();
24          List<int> _XiSum = new List<int>();
25          List<int> _XiXikSum = new List<int>();
26          List<int> _XiXikProductsSum = new List<int>();
27          List<int> _XikSum = new List<int>();
28          List<int> _XiXikProduct = new List<int>();
29          List<int> _XSquareSum = new List<int>();
30          List<int> _aux = new List<int>();
31
32          List<double> _XMean = new List<double>();
33          List<double> _EX2 = new List<double>();
34          List<double> _VarX = new List<double>();
35          List<double> _ac = new List<double>();
36
37          bool _errFlag = false;
38          string _report;
39
40          public override string ModuleStart()
41          {
42              string _Kstring = "";
43              // K=2,4,8,16,32,64,128,256,512,1024
44              for (_K = 2; _K <= 2000; _K *= 2)
45              {
46                  _Ks.Add(_K);
47                  _Kstring += _K + ",";
48              }
49              // Remove last comma from _KString
50              _Kstring = _Kstring.Remove(_Kstring.Length-1);
51              _Kcount = _Ks.Count;
52
53              // Initialize all vectors
54              for (_K = 0; _K < _Kcount; _K++)
55              {
56                  _awPackets.Add(new Queue<int>());
57                  _XiPackets.Add(new Queue<int>());
58                  _nextKPackets.Add(new Queue<int>());
59                  _XikPackets.Add(new List<int>());
```

```
60                   _XiXikProducts.Add(new Queue<int>());
61
62                   _XSum.Add(0);
63                   _XiSum.Add(0);
64                   _XiXikSum.Add(0);
65                   _XiXikProductsSum.Add(0);
66                   _XikSum.Add(0);
67                   _XiXikProduct.Add(0);
68                   _XSquareSum.Add(0);
69                   _aux.Add(0);
70
71                   _XMean.Add(0);
72                   _EX2.Add(0);
73                   _VarX.Add(0);
74                   _ac.Add(0);
75               }
76           Console.WriteLine("Auto-correlation _Ks=[{0}] (total:{1}) started",
77                             _Kstring , _Kcount);
78           return "";
79       }
80
81       public override string ModuleEnd()
82       {
83           return "";
84       }
85
86       public override void AnalyzePacketIn(PacketDotNet.Packet Packet,
87                                            int WindowSize)
88       {
89           _currentCount++;
90
91           if (_Ks[_Kcount - 1] >= WindowSize)
92           {
93               _errFlag = true;
94               return;
95           }
96
97           _packetLenght = Packet.BytesHighPerformance.Length;
98           for (_K = 0; _K < _Kcount; _K++)
99           {
100              _awPackets[_K].Enqueue(_packetLenght);
101              _XSum[_K] += _packetLenght;
102              _XSquareSum[_K] += _packetLenght * _packetLenght;
103              if (_currentCount <= WindowSize - _Ks[_K])
104              {
105                  _XiPackets[_K].Enqueue(_packetLenght);
106                  _XiSum[_K] += _packetLenght;
107              }
108              else
109              {
110                  _nextKPackets[_K].Enqueue(_packetLenght);
111              }
112              if (_currentCount > _Ks[_K])
113              {
114                  _XikPackets[_K].Add(_packetLenght);
115                  _XikSum[_K] += _packetLenght;
116
117                  _XiXikProduct[_K] = _awPackets[_K].Dequeue() * _packetLenght;
118                  _XiXikProductsSum[_K] += _XiXikProduct[_K];
```

```
119                          _XiXikProducts[_K].Enqueue(_XiXikProduct[_K]);
120                      }
121                  }
122              }
123
124          public override void AnalyzePacketOut(PacketDotNet.Packet Packet,
125                                               int WindowSize)
126          {
127              _packetLenght = Packet.BytesHighPerformance.Length;
128              _currentCount--;
129
130              if (_errFlag) return;
131
132              // Assuming that the whole analysis window is full
133              // (this is the expected behavior)
134
135              for (_K = 0; _K < _Kcount; _K++)
136              {
137                  _XSum[_K] -= _packetLenght;
138                  _XSquareSum[_K] -= _packetLenght * _packetLenght;
139
140                  _aux[_K] = _XiPackets[_K].Dequeue();
141                  _XiSum[_K] -= _aux[_K];
142
143                  _aux[_K] = _nextKPackets[_K].Dequeue();
144                  _XiPackets[_K].Enqueue(_aux[_K]);
145                  _XiSum[_K] += _aux[_K];
146
147                  _aux[_K] = _XikPackets[_K][0];
148                  _XikSum[_K] -= _aux[_K];
149                  _XiXikProductsSum[_K] -= _XiXikProducts[_K].Dequeue(); ;
150                  _XikPackets[_K].RemoveAt(0);
151              }
152          }
153
154          public override void Clear()
155          {
156              _currentCount = 0;
157              _XSum.Clear();
158              _XSquareSum.Clear();
159
160              _XiSum.Clear();
161              _XiXikSum.Clear();
162              _XiXikProductsSum.Clear();
163              _XikSum.Clear();
164              _XiXikProduct.Clear();
165
166              _awPackets = new List<Queue<int>>(_Kcount);
167              _XiPackets = new List<Queue<int>>(_Kcount);
168              _XikPackets = new List<List<int>>(_Kcount);
169              _XiXikProducts = new List<Queue<int>>(_Kcount);
170
171              _errFlag = false;
172          }
173
174          public override string ReportAnalysis()
175          {
176              if (_errFlag)
177                  return "!there are Ks greater than analysis window;" +
```

```
178                          Environment.NewLine;
179                   _report = "";
180                   for (_K = 0; _K < _Kcount; _K++)
181                   {
182                       if (_currentCount > 0)
183                           _XMean[_K] = (double)(_XSum[_K]) / _currentCount;
184                       else
185                           _XMean[_K] = 0;
186
187                       _ac[_K] = (
188                                   _XiXikProductsSum[_K] -
189                                   (_XMean[_K] * _XiSum[_K]) -
190                                   (_XMean[_K] * _XikSum[_K]) +
191                                   (_currentCount - (_Ks[_K])) *
192                                       (_XMean[_K] * _XMean[_K])
193                               ) / (
194                                   _XSquareSum[_K] -
195                                   _currentCount *
196                                       (_XMean[_K] * _XMean[_K])
197                               );
198
199                       _report += "ACF(K=" + (_Ks[_K]) + ")=" +
200                           _ac[_K] + ";" + Environment.NewLine;
201                   }
202               return _report + ";;" + Environment.NewLine;
203           }
204       }
205   }
```

B.4 Hurst Parameter Estimator

```
1   using System;
2   using System.Collections.Generic;
3   using NetOdysseyModule;
4
5   namespace HurstParameter
6   {
7       class NetOdysseyHurstPacketsModule :
8           NetOdysseyModuleBase, INetOdysseyPacketAnalyzerModule
9       {
10          int _Kcount; // check method ModuleStart for K generation
11          List<int> _Ks = new List<int>();
12
13          List<Queue<int>> _awPackets = new List<Queue<int>>();
14          List<Queue<int>> _XiPackets = new List<Queue<int>>();
15          List<Queue<int>> _nextKPackets = new List<Queue<int>>();
16          List<List<int>> _XikPackets = new List<List<int>>();
17          List<Queue<int>> _XiXikProducts = new List<Queue<int>>();
18
19          int _packetLenght = 0;
20          int _currentCount = 0;
21          int _K = 0;
22
23          // Hurst parameter variables
24          double _slope;
25          double _hurst;
26          double _dYMean;
27          double _dXMean;
28          double _dAggXX;
29          double _dAggXY;
30          double _log2K;
31
32          List<int> _XSum = new List<int>();
33          List<int> _XiSum = new List<int>();
34          List<int> _XiXikSum = new List<int>();
35          List<int> _XiXikProductsSum = new List<int>();
36          List<int> _XikSum = new List<int>();
37          List<int> _XiXikProduct = new List<int>();
38          List<int> _XSquareSum = new List<int>();
39          List<int> _aux = new List<int>();
40
41          List<double> _XMean = new List<double>();
42          List<double> _EX2 = new List<double>();
43          List<double> _VarX = new List<double>();
44          List<double> _ac = new List<double>();
45
46          bool _errFlag = false;
47          string _report;
48
49          public override string ModuleStart()
50          {
51              string _Kstring = "";
52              // K=2,4,8,16,32,64,128
53              for (_K = 2; _K <= 200; _K *= 2)
54              {
55                  _Ks.Add(_K);
56                  _Kstring += _K + ",";
57              }
58              // Remove last comma from _KString
59              _Kstring = _Kstring.Remove(_Kstring.Length-1);
```

```
60                  _Kcount = _Ks.Count;
61
62              // Initialize all vectors
63              for (_K = 0; _K < _Kcount; _K++)
64              {
65                  _awPackets.Add(new Queue<int>());
66                  _XiPackets.Add(new Queue<int>());
67                  _nextKPackets.Add(new Queue<int>());
68                  _XikPackets.Add(new List<int>());
69                  _XiXikProducts.Add(new Queue<int>());
70
71                  _XSum.Add(0);
72                  _XiSum.Add(0);
73                  _XiXikSum.Add(0);
74                  _XiXikProductsSum.Add(0);
75                  _XikSum.Add(0);
76                  _XiXikProduct.Add(0);
77                  _XSquareSum.Add(0);
78                  _aux.Add(0);
79
80                  _XMean.Add(0);
81                  _EX2.Add(0);
82                  _VarX.Add(0);
83                  _ac.Add(0);
84              }
85              Console.WriteLine("Hurst _Ks=[{0}] (total:{1}) started",
86                              _Kstring , _Kcount);
87              return "";
88          }
89
90          public override string ModuleEnd()
91          {
92              return "";
93          }
94
95          public override void AnalyzePacketIn(PacketDotNet.Packet Packet,
96                                               int WindowSize)
97          {
98              _currentCount++;
99
100             if (_Ks[_Kcount - 1] >= WindowSize)
101             {
102                 _errFlag = true;
103                 return;
104             }
105
106             _packetLenght = Packet.BytesHighPerformance.Length;
107             for (_K = 0; _K < _Kcount; _K++)
108             {
109                 _awPackets[_K].Enqueue(_packetLenght);
110                 _XSum[_K] += _packetLenght;
111                 _XSquareSum[_K] += _packetLenght * _packetLenght;
112                 if (_currentCount <= WindowSize - _Ks[_K])
113                 {
114                     _XiPackets[_K].Enqueue(_packetLenght);
115                     _XiSum[_K] += _packetLenght;
116                 }
117                 else
118                 {
```

```csharp
119                    _nextKPackets[_K].Enqueue(_packetLenght);
120                }
121                if (_currentCount > _Ks[_K])
122                {
123                    _XikPackets[_K].Add(_packetLenght);
124                    _XikSum[_K] += _packetLenght;
125
126                    _XiXikProduct[_K] = _awPackets[_K].Dequeue() * _packetLenght;
127                    _XiXikProductsSum[_K] += _XiXikProduct[_K];
128                    _XiXikProducts[_K].Enqueue(_XiXikProduct[_K]);
129                }
130            }
131        }
132
133        public override void AnalyzePacketOut(PacketDotNet.Packet Packet,
134                                              int WindowSize)
135        {
136            _packetLenght = Packet.BytesHighPerformance.Length;
137            _currentCount--;
138
139            if (_errFlag) return;
140
141            // We assume that the whole analysis window is full
142            // (this is the expected behavior)
143
144            for (_K = 0; _K < _Kcount; _K++)
145            {
146                _XSum[_K] -= _packetLenght;
147                _XSquareSum[_K] -= _packetLenght * _packetLenght;
148
149                _aux[_K] = _XiPackets[_K].Dequeue();
150                _XiSum[_K] -= _aux[_K];
151
152                _aux[_K] = _nextKPackets[_K].Dequeue();
153                _XiPackets[_K].Enqueue(_aux[_K]);
154                _XiSum[_K] += _aux[_K];
155
156                _aux[_K] = _XikPackets[_K][0];
157                _XikSum[_K] -= _aux[_K];
158                _XiXikProductsSum[_K] -= _XiXikProducts[_K].Dequeue(); ;
159                _XikPackets[_K].RemoveAt(0);
160            }
161        }
162
163        public override void Clear()
164        {
165            _currentCount = 0;
166            _XSum.Clear();
167            _XSquareSum.Clear();
168
169            _XiSum.Clear();
170            _XiXikSum.Clear();
171            _XiXikProductsSum.Clear();
172            _XikSum.Clear();
173            _XiXikProduct.Clear();
174
175            _awPackets = new List<Queue<int>>(_Kcount);
176            _XiPackets = new List<Queue<int>>(_Kcount);
177            _XikPackets = new List<List<int>>(_Kcount);
```

```
178                     _XiXikProducts = new List<Queue<int>>(_Kcount);
179
180                 _errFlag = false;
181             }
182
183         public override string ReportAnalysis()
184         {
185             if (_errFlag) {
186                 return "!there are Ks greater than analysis window;" +
187                     Environment.NewLine;
188             }
189             _report = "";
190             _dYMean = 0;
191             _dXMean = 0;
192             _dAggXX = 0;
193             _dAggXY = 0;
194
195             for (_K = 0; _K < _Kcount; _K++)
196             {
197
198                 if (_currentCount > 0)
199                     _XMean[_K] = (double)(_XSum[_K]) / _currentCount;
200                 else
201                     _XMean[_K] = 0;
202
203                 _ac[_K] = (
204                             _XiXikProductsSum[_K] -
205                             (_XMean[_K] * _XiSum[_K]) -
206                             (_XMean[_K] * _XikSum[_K]) +
207                             (_currentCount - (_Ks[_K])) *
208                                 (_XMean[_K] * _XMean[_K])
209                         ) / (
210                             _XSquareSum[_K] -
211                             _currentCount *
212                                 (_XMean[_K] * _XMean[_K])
213                         );
214
215                 _log2K = Math.Log(_Ks[_K], 2);
216
217                 if (_Kcount > 1)
218                 {
219                     _dYMean += _ac[_K] / (double) _Kcount;
220                     _dXMean += _log2K / (double) _Kcount;
221                     _dAggXX += _log2K * _log2K;
222                     _dAggXY += _log2K * _ac[_K];
223
224                     _slope = (
225                                 _dAggXY -
226                                 (double) _Kcount *
227                                 _dXMean *
228                                 _dYMean
229                             ) / (
230                                 _dAggXX -
231                                 (double) _Kcount *
232                                 _dXMean *
233                                 _dXMean
234                             );
235                 }
236
```

```
237                        _report += "log2(K=" + (_Ks[_K]) + ")=;" +
238                                   _log2K + "; " +
239                                   "log2(A(K=" + (_Ks[_K]) + "))=" +
240                                   _ac[_K] + ";" +
241                             Environment.NewLine;
242                }

243

244            if (_Kcount > 1) {
245                _hurst = (_slope + 1) / 2.0;
246                _report += "_slope=;" + _slope + "; " +
247                           "_hurst=;" + _hurst + ";" +
248                           Environment.NewLine;
249            }

250

251            return _report + ";;;" + Environment.NewLine;
252        }
253    }
254 }
```