UNIVERSITY OF OSLO
Department of Informatics

Passive Traffic Characterization and Analysis in Heterogeneous IP Networks

Master thesis

Håvard Wik Thorkildssen

May 10, 2005

Passive Traffic Characterization and Analysis in Heterogeneous IP Networks

Håvard Wik Thorkildssen

May 10, 2005

Abstract

In this thesis we revisit a handful of well-known experiments, using modern tools, to see if results yielded from earlier experiments are valid for today's heterogeneous networks. The traffic properties we look at are relevant for designing and optimizing network equipment, such as routers and switches, and when building corporate networks. We have looked at the characteristics of two different heterogeneous networks; a university network, and an ISP network. We have captured traffic from different weeks, and at different times of the day. We first describe the challenges involved with collecting, processing and analyzing traffic traces from high-speed networks. Then we then look at the various factors that contribute to uncertainty in such measurements, and we try to deduct these factors. The experiments involve collection and analysis of high-resolution traffic traces from two operative networks, each of which contains several gigabytes of network traffic data. We look at properties such as: Packet inter-arrival time distributions, packet size distributions, modeling packet arrivals (self-similarity versus Poisson), traffic per application (egress traffic per destination port), and protocol distributions. A simplistic attempt to quantify the volume of Peer-to-Peer (P2P) traffic inspecting both header data and payload is conducted to evaluate the efficiency of today's methodology for identification (port numbers only). We have used freely available tools like TCPDump, Ethereal, TEthereal, Ntop, and especially the CAIDA CoralReef suite. The shortcomings of these tools for particular tasks have been compensated for by writing custom-made Perl scripts, proving that it is possible to do advanced analysis with fairly simple means. Our results reveal that there are in fact measurable differences in terms of packet inter-arrival time distributions and statistical properties in the two networks. We also find significant differences in the application distribution, and the deployment of new technologies such as Multicast.

Acknowledgements

First and foremost, I would like to thank my supervisor, assistant professor Hårek Haugerud, for his invaluable help during this stressful period. Second, I would like to thank Dag Langmyhr for making sure coordination between all involved parties went smoothly, and for his excellent introduction to LaTeX. I would also like to thank: Are Gravbrøt, Jon Suphammer, Steinar Haug, Chris Qvigstad and Morten Kjønnø from CATCH, for their technical expertise and for being sympathetic guys. Frode Eika Sandnes and Kyrre M. Begnum, for being academic inspirations. Mark Burgess, for fruitful discussions and for being the person he is. Magnus R. Solberg, for reading through this thesis. I would also like to express my gratitude to Stig Jarle Fjeldbo, Ole Rindalsholt, Trond Aspelund, and the rest of the master group, for useful feedback and healthy criticism during these last two years. Furthermore, I would like to thank my mother, Kirsten Elshaug Wik, for academic guidance, for helping me solve the more practical problems, and for always standing up for me. Iver Kjekshus and Atle Eriksen, for letting me sleep at night, and Hans Henrik Clementz, for putting my own mental quirks into perspective. I also owe a great deal to my beautiful girlfriend Kristin Øhlckers, for her patience with me on stressful late-hours, and for her unconditional love and friendship. Last, but not least, I would like to thank the rest of my friends and family, and the people whom I have forgotten to mention here, for being supportive and bearing over with me when I am the most stressed out.

Oslo, May 2005

Håvard Wik Thorkildssen

Table of Contents

List of Figures

List of Tables

Preface

Project Background

The idea for this thesis began sometime in the fall of 2004, when the author regained a previous interest in monitoring and analyzing traffic passing through servers in his home network. A quick search on Google revealed that there are several thousand software packages available, each of them meeting its own need. And the best part; most of them are available at no charge. In the case of Open Source/Free Software [ope99], you can even further customize the software by altering and fitting the source code to specific needs[i]! Secondly, the author was curious about the differences in network traffic characteristics of different heterogeneous networks, for example an ISP and a university network, and whether or not these differences could be quantified. The users of a university network are in many respects similar to the average home-user. However, this is not necessarily reflected how they use the network — people do different things when they are at home than when they are working or studying. This notion was formalized to a project plan, which was accepted by the university at the end of 2004. The total time available to this project was approximately 17 weeks.

Target Audience

Basic knowledge about how (inter-)networks and higher-level protocols work is clearly an advantage for the reader. However, a well-informed average reader should be able to follow most of the experiments without any prior knowledge of these subjects. We shall give a brief introduction to networking concepts, as well as measurement and analysis methodology, that will hopefully enable the networking-novice to understand the basic concepts. Some examples are placed directly in the text to ease prospective reproduction of the experiments. Additionally, verbose output and code is included in the appendices.

The scope of this thesis is to take a closer look at network traffic behavior from a low-level perspective. Albeit with a few exceptions, the design and functionality of user-oriented services, such as HTTP, DNS, SMTP, and streaming multimedia, will not be elaborated unless it has absolute relevance to the

[i] Most of which are released under open source/free software licenses, such as the GNU Public License or Berkley BSD license and derivatives.

methodology or results.

Terminology

The terms *captures*, *data-traces*, *traces*, and *dumps*, are used interchangeably throughout this thesis. These terms refer to packet header traces. Terms like *P2P* and *file-sharing* are both used, although file-sharing is only a subset of the P2P concept. We use the term *high-speed networks* frequently. This term is a subjective and perhaps diffuse term, however we generally mean networks with throughput that exceed around 10Mbit/s.

Thesis Outline

In this thesis outline, we look at the most important sections of the thesis. Refer to the table of contents for a more detailed overview of the thesis.

Chapter 1 introduces the main motivation for doing Internet and LAN measurements and analysis, and describes the challenges involved with conducting studies of this nature. In Section 1 and 2, we have tried to place the subject into context, along with a preliminary description of the problem. In Section 4 we introduce the Cooperative Association for Internet Data Analysis (CAIDA).

Chapter 2 contains the background material and literature survey, where we provide an overview of previous work on these subjects along with some fundamental theory. In Section 1, we introduce the reader to basic networking concepts. Section 3 elaborates on the various measurement techniques that are used, and under what circumstances they are suitable. In Section 4, the most popular data collection and analysis tools available are described. In Section 5, we look at a selection of network properties. Section 6 describes the CoralReef suite, and how it can be utilized by system and network administrators. In Section 7, we look at Peer-to-Peer and file-sharing applications.

Chapter 3 discusses the methodology of the experiments. In Section 1, we look at the basic methodology for conducting network measurements. In Section 2 and 3, we look at constraints and limitations for this study. Section 4, 5, and 6, describes the methodology for capturing, processing and analyzing network traces, respectively. Section 7 describes two widely used traffic models, and In Section 8, we elaborate on a methodology for estimating the traffic volume of P2P traffic.

Chapter 4 is dedicated to discuss the results from the experiments.

Chapter 1

Introduction

When people thought the Earth was flat, they were wrong. When people thought the Earth was spherical they were wrong. But if you think that thinking the Earth is spherical is just as wrong as thinking the Earth is flat, then your view is wronger than both of them put together. –Isaac Asimov

1.1 The Big Picture

The field of network traffic characterization and analysis dates back to research on the first switched telephone networks in the beginning of the 20th century. The research conducted by pioneers on the field, such as Erlang [JW99], formed the foundation for modern network traffic analysis.

Recent advances in network technologies have far outpaced our abilities to effectively manage and engineer them. However, through the efforts of several research communities, such as the Cooperative Association for Internet Data Analysis, CAIDA, we have come a long way in the field of effectively characterizing and modeling networks and network traffic. The data rates are in magnitudes higher than those of telephone networks, and the networks have become so complex that it is impossible to grasp even for the most experienced network administrator. Due to the nondeterministic and decentralized nature of the Internet, one can say that the Internet has become a being of its own, and in many ways, the Internet has grown out of control.

In recent years, the demand for bandwidth[i] on the Internet has sky-rocketed as a result of the deployment of new applications that are capable of utilizing the capacity of modern networks. The bandwidth demand is driven by several factors, and one should not assume that it is a fixed quantity. First of all, users tend to utilize the network more if the network is well functioning, and there is a relatively loose policy. This encourages the users to find new ways to utilize the network. On the contrary, if the network does not function well, users will steer away from it and find alternative forms of communications [Peu02]. With

[i] The term bandwidth was originally a term used to describe the width, usually measured in hertz, of a frequency band. However, in a digital context it refers to he amount of data that can be transferred through a digital connection in a given time period (i.e., the connection's bit rate, which is usually measured in bit/s).

increasing bandwidth demand comes higher data rates, and with higher data rates comes complexity, and the importance of proper network administration becomes apparent.

The design and construction of networks and network equipment is not based on arbitrary assumptions about the environment in which it is to operate. They are constructed after thorough research about the characteristics of the traffic. Since these characteristics vary with the network environment, the topological design and configuration is often made-to-measure on a case-by-case basis. The traffic that passes through a given network node, for example a border router, is often in magnitudes of several gigabytes of data per second. Hence, the equipment deployed and general design of the network has to be optimized in order to process every packet without malfunctioning or skipping packets. However, the processes that trigger network traffic are, by nature, random processes. By *random*, we mean that there are too many unknown factors to be able to trace the channels of cause and effect [Bur04]. The complexity of the networks is often so comprehensive that it is impossible for the human mind to grasp. In order to reduce the complexity, experiments can be performed on a selected location in the network, and conclusions about the system as a whole can be drawn from the findings. Several factors influence the value of the results from such experiments; hence one shall not underestimate the value of planning, and assessment of uncertainties. We shall look closer at factors contributing to uncertainty in network measurements later in this thesis.

For a network administrator, it is vital to have an adequate understanding of the characteristics of the network, not only from a scientific point of view, but also in order to be able to follow changing trends in usage-patterns and volume, and thus to be able to handle the network demand of both the near and distant future. By measuring and analyzing networks, you get an objective record or benchmark of how it behaves. This will make it easier to deduct cause and effect when implementing changes in the network, and to judge whether or not changes in the network have improved its performance or degraded it [Gog00].

1.2 Networks

A network is a collection of hosts, or nodes, connected together so they can exchange information. These hosts can be special-purpose hardware such as routers or printers, or regular computers, which may run several different operating systems and services, e.g., Microsoft Windows, Mac, or UNIX. The hosts communicate through an agreed set of protocols, such as the TCP/IP protocol suite. These protocols define how the flow of information is to be exchanged. We will look closer at the TCP/IP suite in the methodology section. Fig. 1.1 and Fig. 1.2 shows a user network (LAN) and an ISP network (WAN/MAN) respectively [Gro01].

An internetwork, such as the Internet, is a collection of networks that are interconnected in a mesh. The nodes do not have to be directly connected to

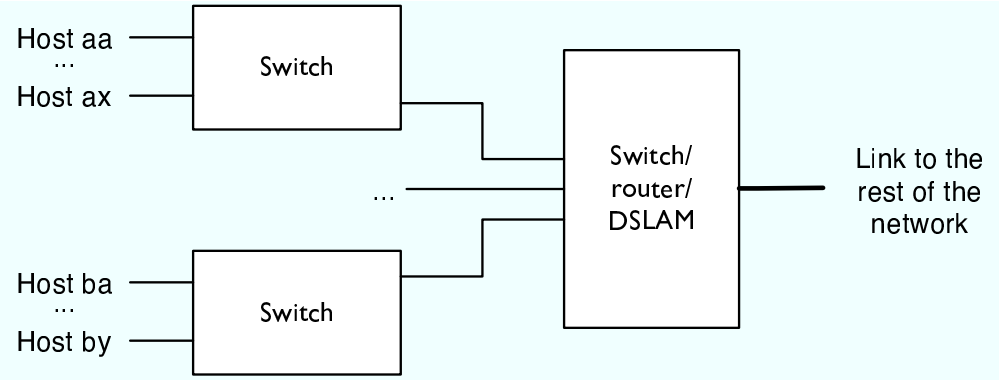

Figure 1.1: Example of a (moderate-sized) user network

each other in order to communicate. Routers are connected to more than one network and *routes* traffic in the form of network packets between them. A central router in the Internet may be connected to as many as several hundred networks simultaneously, holding routing tables for all networks and seamlessly routing traffic between all its connected networks.

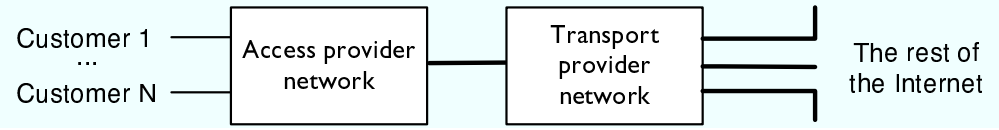

Figure 1.2: Example of a (small) ISP network

Transport providers are ISPs who run their own wide-area network, or WAN, and provide connectivity to its customers via that network [Gro01]. Transport providers maintain high-speed links that cover large geographical areas, and, through a concept called *peering*, connects to other providers. This interconnection between transport providers and transit providers constitutes the previously mentioned mesh, where all hosts are, in theory, capable of reaching any other host in the internetwork through other nodes. This route through several nodes in a network is often referred to as a *path*.

The network traces analyzed in these experiments are from the OUC student (user) network and from a leaf-node in the CATCH Communications ISP network.

1.2.1 CATCH Communications

CATCH Communications own and operate their own infrastructure, and their network has nodes throughout Norway, covering approximately 65 % of the population. Subsequently, their backbone networks carry an enormous amount of data. In order to limit the amount of data, and hence making it easier to process with our fairly modest equipment, the traces will be dumped at a rel-

atively low-traffic node in the Lillestrøm area, just outside Oslo. The node is connected directly to the backbone network and hosts primarily private end-users. The maximum throughput of the uplink pipe is 20Mbit/s.

1.2.2 Oslo University College

Oslo University College has a high-speed fiber connection to the Internet through a high-speed node at the college administration, which is directly connected to NIX1 and NIX2[ii]. The student network, at which the traces are to be captured, is on a separate VLAN (Virtual LAN). The maximum throughput of the network is 100Mbit/s.

1.3 Measuring the Immeasurable

Measuring the Internet is, at first glance, an impossible task. The Internet today contains several million hosts, and no computer or piece of hardware exists that is fast enough to process and interpret the statistics from all core routers on the Internet simultaneously. It is, therefore, vital to state a precise definition of what you want to measure, why, and what it can tell you. A natural question that arises is: What is all this data and who are requesting it? The Cooperative Association for Internet Data Analysis (CAIDA) are continually working on finding answers to questions of this nature. They are difficult to answer, due to a number of reasons including:

- The amount of data involved are extremely large. Even through forcing the capturing device to discard payload, and only capture the TCP/IP headers from a single link for a few minutes, may generate several gigabytes of data to be stored and/or analyzed.
- The Internet is, by nature, decentralized and there is no single place at which to make measurements. Moreover, there is no single organization responsible for coordinating and controlling the Internet at this level.
- There are several large organizations that own the resources that make up the Internet, however not all of these perceive it to be in the commercial interests of the company to do measurements of this nature, or they might not have the resources required to do so.

Measurements of the Internet are spawned by independent parties all over the globe. However, these measurements are often *end-to-end measurements*, performed by people wanting to verify the the performance of their Internet service. These measurements are for the most part active measurements, a concept that will be discussed later, where the user is actively probing their network with packets, measuring the delay until the packet returns to its source. Conducting such measurements can be useful in many situations, however it is difficult, if not impossible, to take all contributing factors into consideration

[ii] The Norwegian Internet eXchange, located at Oslo Innovation Center.

and draw any real conclusions from them. An example of such measurements is the so-called *Internet speedometers*, a service that involves timing the download of a file, for example an arbitrary image, fetched from a remote server. The actual timing of such a process can be biased by several uncorrelated sources, for example overhead caused by heavy load on links operated by transport providers, and it does not give you a good measurement of the service level provided by the ISP.

1.4 The Cooperative Association for Internet Data Analysis

CAIDA is a project of the National Laboratory for Applied Network Research (NLANR) within the University of California, San Diego. The project is heavily involved with research that has inspired the experiments conducted in this thesis. Subsequently, literature written and published by CAIDA is essential background material to this thesis.

CAIDA is a collaborative undertaking to promote greater cooperation in the engineering and maintenance of a robust, scalable global Internet infrastructure. It will address problems of Internet traffic measurement and performance, and of inter-provider communication and cooperation within the Internet service industry. It will provide a neutral framework to support these cooperative endeavors. Tasks are defined in conjunction with participating CAIDA organizations [MC97]. They develop and maintain software that is deployed at large Internet junctions, and that is used by researchers and ISPs in all parts of the world. We will use the CoralReef suite from CAIDA extensively in our experiments.

Chapter 2

Background Material and Previous Work

In this chapter, we shall introduce the reader to some fundamental networking concepts, such as the Ethernet, the xDSL family of technologies, and the TCP/IP protocol suite. We shall also look at previous work on the field, and discuss the software that is to be used in our experiments.

2.1 Internet Connectivity

Universities and other academic institutions used to be the junctions of the Internet, and have therefore traditionally been connected to the Internet with high-speed Ethernet links and fiber optics. This is still the case in most parts of the world. However, when building infrastructure for private end-users, these technologies have proven far too expensive to implement. Thus, cheaper technologies, like xDSL and Internet over the cable-TV infrastructure, have become widespread. These technologies scale better over large geographical areas — however, they can not offer the same network throughput or availability, albeit they have become far better in recent years [BA99].

2.1.1 Ethernet

Ethernet is a frame-based computer networking technology for local area networks (LANs), and has in the later years also been deployed in metropolitan area networks (MANs) and wide-area networks (WANs). It is a shared medium, and collision management is handled by an algorithm known as carrier sense multiple access with collision detection (CSMA/CD) [MB76].

2.1.2 The Ethernet Frame

Ethernet traffic is transported in units of a frame, where each frame has a definite beginning and end. The Ethernet frame consist of five elements: the Ethernet header, the IP header, the TCP header, the encapsulated data, and the Ethernet trailer. A model of the frame is provided in Fig. 2.1.

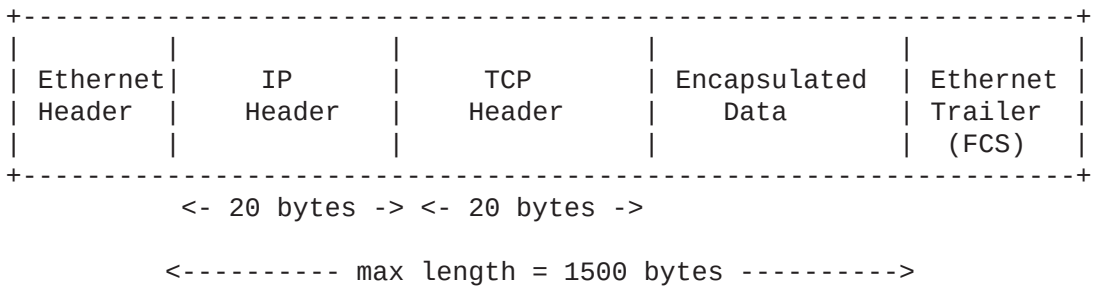

Figure 2.1: The Ethernet frame

2.1.3 The TCP/IP Reference Model

The TCP/IP model is an abstract model that describe the design of communications and computer networks. It was designed as a simpler, more Internet-oriented model, to replace the aging OSI model. We shall not discuss the OSI model further in this survey, as it has been replaced by the TCP/IP reference model for all practical purposes. The model has four layers, as opposed to the seven-layered OSI model. The four levels are the application layer, the transport layer, the network layer, and the data link layer. In Table 2.1 we provide an overview of the TCP/IP model.

Layer #	Layer # (OSI)	Layer	Services
4	5,6,7	Application layer	HTTP, SMTP, FTP
3	4	Transport layer	TCP, UDP, SCTP
2	3	Network layer	IPv4, IPv6, ICMP
1	1,2	Data link layer	Ethernet, ARP, 802.11a

Table 2.1: The TCP/IP Model

The data to be sent is encapsulated by each layer, from the application down to the physical, and each layer adds its own header information. When data is received, each layer strips off the header, and then passes the packet up to the next layer. The transport layer includes source and destination hosts and ports, and a sequence number, so that a file can be disassembled into multiple packets and assembled at the receiving end. How the frames are to be delivered is determined by the network layer. The Maximum Transmission Unit defines the maximum size of the packet, with IP header, TCP header and payload combined. The network layer makes sure that packets that are to be sent along paths with a smaller MTU are fragmented. Most network interface cards are configured by default with a MTU of 1500 bytes, and in a LAN, under normal conditions, packets are not fragmented. The Network layer also provides the encapsulation of the datagram into the frame that is transmitted over the network. Since Ethernet addresses (MAC addresses) are not routable, the network layer rewrites the Ethernet addresses with each hop.

2.1.4 The TCP/IP Protocol Suite

The TCP/IP protocol suite, also known as the Internet protocol suite, is a set of network communication specifications that is implemented in equipment operating in networks that range from small home networks, with a couple of hosts, to the globe-spanning Internet. The protocol is referred to as a *suite* since it includes two protocols:

TCP The Transmission Control Protocol, and

IP The Internet Protocol

A packet contains two parts: a header part, and a payload part. We can think of the IP and TCP layer as two independent packets, where the IP packet has encapsulated the TCP packet. Both the TCP and the IP layer have its own header. A full IP and TCP header is 20 bytes long each, without options and padding. The payload part can be of variable length. In Fig. 2.2 and Fig. 2.3 we provide an overview of the IP and TCP headers, respectively. We shall not go into detail on all fields of the header, but we will provide a brief overview of the fields that are relevant to our experiments.

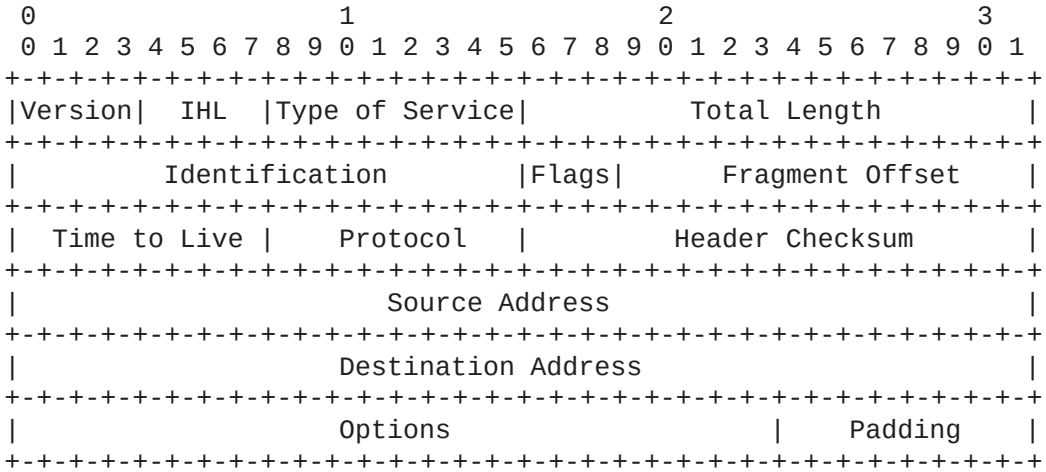

Figure 2.2: IP Header Format

Relevant Fields in the IP Header

The following fields in the IP header are relevant to our experiments:

- Source address
- Destination address
- Total length
- Protocol

The *source address* is the IP address of the original sender of the packet. The format of an IP address is a numeric 32-bit address written as four numbers, separated by periods. Each number can be zero to 255.

The *destination address* is the IP address of the final destination of the packet.

The *total length* of the packet is the size of the datagram, and is a value given in bytes. This is the combined length of the header and the data.

The *protocol* indicates the type of transport packet being carried. These protocols are represented by a decimal number, as we can see in e.g., [ip96]. The most common protocols on the Internet are TCP (6), UDP (17) and ICMP (1).

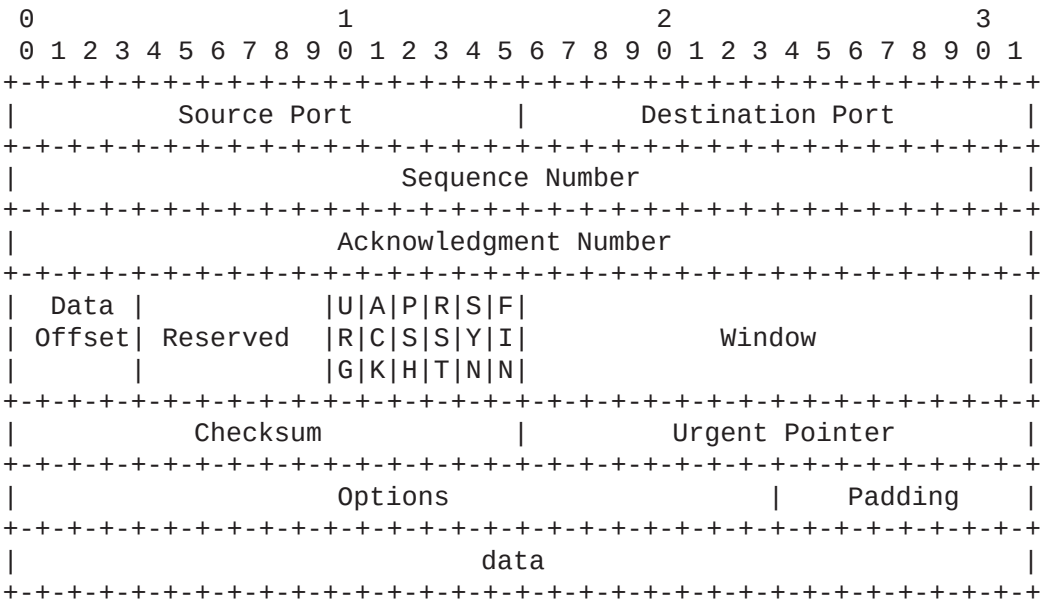

Figure 2.3: TCP Header Format

Relevant Fields in the TCP Header

The following fields in the TCP header are relevant to our experiments:

- Source port
- Destination port
- Data

Ports are used to separate between independent TCP or UDP flows; either between the same host pairs, or between different hosts. In modern heterogeneous networks, hosts may run several networked services, creating the need for several "local addresses" on the same hosts. Sending hosts are connected to the destination port of the receiving host, and the receiving host identifies the sending host by their source port. Hence:

The *source port* is the outgoing port from the sending end.

The *destination port* is the incoming port on the receiving end.

The *data*, often referred to as the *payload*, is the actual information of the packet. This data can be represented in hex, binary or decoded in ASCII. We shall use the ASCII form of the data to inspect the payload.

2.1.5 The xDSL Family of Technologies

xDSL is a carrier technology on the data link layer, dating back to research performed at Bell laboratories in the late 1980s. xDSL, or simply DSL, refers to a whole family of technologies. The researchers at Bell found out that by utilizing unused frequency spectra on copper wires, the wires could carry both ordinary telephone traffic and digital broadband transmissions without interference. The theoretical capacity of a copper wire is related to the Shannon capacity, formulated in Shannon's theorem, of the wire. However, this is beyond the scope of this document. The xDSL family of technology includes:

- ADSL (Asymmetric Digital Subscriber Line)
- HDSL (High Bit Rate Digital Subscriber Line)
- RADSL (Rate Adaptive Digital Subscriber Line)
- SDSL (Symmetric Digital Subscriber Line, a standardized version of HDSL)
- SHDSL (Single-pair High Speed Digital Subscriber Line)
- VDSL (Very high speed Digital Subscriber Line)

Each of these technologies has different properties and areas of utilization. However, they are common in that they provide a digital connection over the copper wires of the local telephone network, connecting them to a xDSL gateway (DSLAM). The DSLAMs are connected to the Internet through the backbone network, and traffic to and from DSLAMs are usually carried over ATM networks. xDSL is an "always-on" service, and most charge a fixed-rate fee for their service, albeit a few actors (Telenor, Tiscali) have experimented with a volume-oriented price model with very limited success.

We shall not delve further into the design or technical specifications of xDSL, as the traffic we shall capture is Ethernet frames, captured on the backbone network of the ISP.

2.2 Internet Measurement and Data Analysis

Claffy [CM99] et al. compares the Internet to a cybernetic equivalent of an ecosystem. The last mile connections from the Internet to private end-users and enterprises are supplied by thousands of capillaries, and the different ISPs maintain the arteries, the backbone network. As the Internet becomes more and more complex, an adequate understanding of the processes behind networks becomes more and more important. An insight into the overall health

and scalability of the system is critical to the Internet's successful evolution, Claffy elaborates. Network measurements involve the collection, processing, analysis, and post-processing of the data. In the next section, we shall look at a few techniques for collecting data from networks.

2.3 Measurement and Capturing Techniques

2.3.1 Monitor Placement

Having a clear understanding of the network topology is an important prerequisite to monitor placement. In these experiments, we shall only look at a small piece of the networks; hence the results yielded from them will not necessarily reflect the whole network. While it might be tempting to measure traffic between every pair of sites, the cost does not scale with the benefit [BCea01].

2.3.2 Active versus Passive Measurements

Active and passive measurement techniques are often used in combination, since the two techniques yields different properties of the network [MJ98].

Active measurements perturb the network, for example by probing the network with a ICMP ping and measuring the time it takes, or measuring the loss along a datagram stream. Brownlee, Claffy et al [BCea01] notes that, in order to yield significant results, passive monitors must process the full load of the link, which can be a challenge on high-speed links. Passive measurements are measurements where one infer performance data from the underlying network flows without perturbing the network or infrastructure [MJ98]. Hence, passive measurements do not suffer from this constraint.

More practically, active measurement techniques are used for measuring e.g.: Availability, error rate, response time and data throughput. On the other hand, passive measurement techniques are used for measuring e.g.: Interpacket arrival times, packet length distributions, length of activity periods, length of silence periods, time between connections, and duration of connection, as defined by [Gog00]. We shall elaborate on these properties later in this text.

2.3.3 Physical Tapping versus SPAN port

Passive measurements involves tapping into a network and recording traces from it. There are basically two ways of tapping into a network. As you can see from Fig. 2.4, you can either tap into the network via a physical splitter (2.4a), or configure a mirror, or SPAN port (2.4b) on the appropriate switch or router. SPAN is an acronym for Switched Port Analyzer, and it was originally a feature from the Cisco Catalyst line of switches [Hea00]. However, this feature has now become a standard feature on advanced network equipment from vendors such as Juniper, Nortel. Extreme networks, and Lucent, as well.

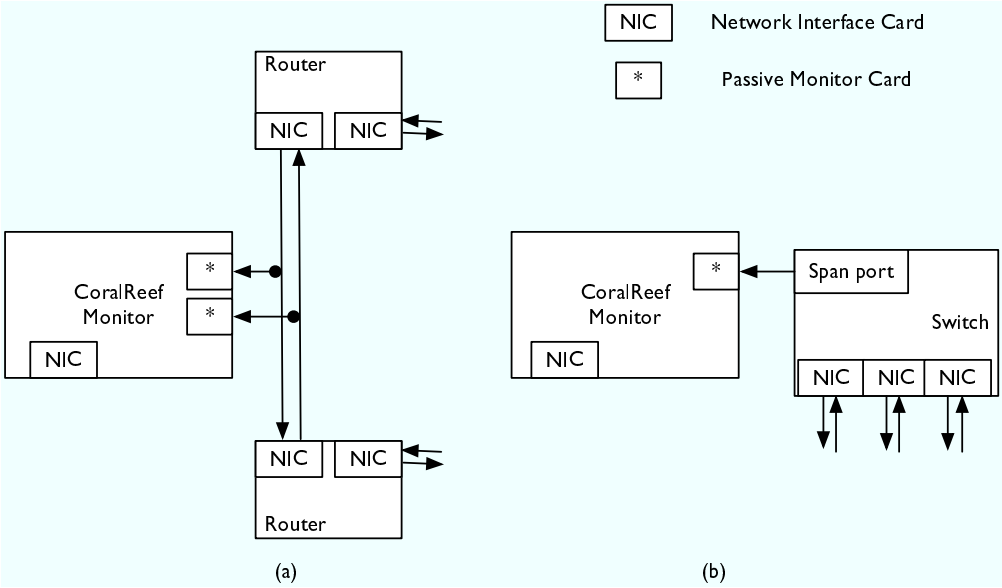

Figure 2.4: Network tapping with: (a) Physical splitter, (b) SPAN port

2.3.4 Software versus Hardware Monitors

There are two major types of network monitors; software- and hardware monitors [Gog00]. Most monitors, however, will be characterized as something in between. Generally speaking, a hardware monitor is a special piece of network equipment designed specifically for the task, namely capturing and analyzing network traffic. These are off-the-shelf units that can be acquired from any of the big network equipment manufacturers, like Cisco Systems or Juniper Network, or be custom built. A software monitor, on the other hand, is usually some sort of PC with a network interface card (NIC). PCs can easily be expanded and customized to perform a specific task, e.g., expanding the box with a number of network interface card or with a WAN card. For analysis in networks with relatively low throughput, one can use an unmodified GNU/Linux desktop system without degrading the accuracy of the results. Most software monitors are running a variant of the UNIX operating system, for example GNU/Linux or FreeBSD. Several people have experimented with using the Microsoft Windows operating systems, but UNIX and clones are generally more effective. Empirical evidence has shown that the GNU/Linux perform better in high-throughput networks than other operating systems, due to the way system calls are handled by the kernel and the TCP/IP stack implementation of the OS [Rot01].

2.3.5 Software Monitors

Gogl [Gog00] provides a summary of the advantages and disadvantages of software and hardware monitors.

Advantages

The simplest software monitor is available at no costs beyond a standard networked computer. A lot of free software and open source tools are available, which are capable of doing very advanced analysis, both active and passive, and real-time and offline. Excellent examples of such tools are the network dump and analysis tools TCPDump and Ethereal, and Ntop [DS00a] – a network traffic probe. We shall look closer at these tools later in this thesis.

Maintenance of software monitors is easy and cheap. Newer versions of the monitor programs can easily be upgraded through standard package management tools, and they are available at no cost. The most popular tools are often rapidly developed and enhanced to include new functionality by a large community of programmers, and the patches are spread throughout Internet mirrors almost instantly. The user of the monitor can normally handle the replacement or upgrade of the software without causing long disruptions of ongoing measurements due to hardware updates. If the software is developed by an open source model, the source code is also available, making it possible to perform on-site customizations for specific tasks. An example of such customizations is to modify the output format of which the monitor uses. However, this does not necessarily involve modifying the source code of the tool itself, but can be accomplished by using standard UNIX tools like `grep`, or by using free development frameworks, such as `Net::Pcap`[i].

The main characteristic of a software monitor is that they run on the monitored system itself. Software monitors are therefore able to access internal data of the system, in contrast to hardware monitors, where the monitor is tapped onto the network. An example of data that only a software monitor can measure is the packet delay caused by the packaging process within a host or the queue length of internal buffers [Gog00].

Disadvantages

However, software monitors have some drawbacks. Software monitors have no direct access to the media, only to a host connected to the media [CDG00, Gog00]. Therefore, they are not suited for measuring hardware-near events, like signal errors on the network line. Software monitors are also prone to uncertainty in the results caused by the sharing of resources. During network peak hours, the monitor may have enough problems coping with its own traffic, neglecting monitoring tasks. This may cause inaccuracy and uncertainties in the form of e.g., displaced time stamps. Gogl [Gog00] suggests that software monitors are only to be used for relatively low input and sample rates. However, faster computers are able to cope with more throughput without causing inaccuracy in the measured results. A software monitor may require hundreds of operations per network packet, and thus the input rate is limited by the host's processor speed. Nevertheless, the accuracy of our fairly modest equipment is more than sufficient for our experiments.

[i] Perl bindings for libpcap.

2.3.6 Hardware Monitors

Advantages

The circuitry of a hardware monitor is designed specifically to monitor and analyze network traffic data, hence they are often able to process larger volumes of data, and with higher sample rates than software monitors [MHK+03]. Moreover, hardware monitors are directly connected to the network media, and are therefore able to detect and monitor low level events like signal errors and signal degradation, and they can even be used to identify specific failures of network component interfaces [Gog00].

Hardware monitors are external boxes that, in contrast to software monitors, run independently from the monitored system. As a result of this, hardware monitors do not interfere with the resource consumption or depend on the availability of the system.

Disadvantages

The advantages of a hardware monitor come at a cost; the boxes are often very expensive, both in purchase cost and in maintenance. Examples of maintenance are firmware/software upgrades and hardware upgrades, e.g., memory upgrades. It is not given that these upgrades can be performed *on-site*, and several manufacturers require that the customer send in the equipment for maintenance [Gog00].

These systems often run proprietary, non-standardized operating systems, and must be operated by a qualified user, i.e., a competent UNIX administrator is not necessarily adequate. In contrast to the software monitor, these boxes appear to the user as a closed system, and there is often no information available about their inner workings.

2.3.7 Choosing a Hardware Platform

Dumping and processing traffic in high-speed networks puts the system under serious stress, and it is crucial to choose a hardware platform that is able to cope with the throughput, especially with respect to disk I/O and CPU. Cleary and Donelli [CDG00] have found that, although the IDE (ATA-66) specification defines a maximum bandwidth of 22MB/s, experiments have shown that the maximum data rate achieved with a standard IDE disk is far lower – 5-6 MB/s in their experiments. According to Moore, Keys et al [MKea01], the choice of hardware depends on the utilization of the links being monitored and the amount of aggregation desired. For normal packet traces, the main constraint is usually disk performance and capacity. They recommend ultra-wide SCSI rather than IDE, although the newer S-ATA interfaces with faster disks are probably able to cope with high throughput equally good. For flow collection and analysis, CPU, and memory capacity are usual constraints. However, the networks monitored in these experiments do not yield throughputs that will make these constraints bottlenecks.

2.3.8 Choosing a Software Platform

The hardware platform of choice for our experiments is a networked PC. Software that is to be deployed in high-speed environments should be scalable up to gigabit/s speeds, and be able to handle fluctuations in data rate. The software we shall use has been tested in environments with significantly higher throughput than in our networks. In the next section we will look at the software tools that are to be used in our experiments.

2.4 Data Collection and Analysis Tools

2.4.1 TCPDump

TCPDump [JLM04] is a utility that allows a user to capture and store packets passing through a network interface[ii]. This is a handy utility, which can prove invaluable for a network administrator interested in monitoring or debugging the network. It has some fairly powerful features, such as the extensive filtering capabilities. As a result of being powerful, this utility has also been used for unlawful purposes such as password sniffing.

Under normal conditions only packets that are addressed to a network interface are intercepted and passed onto the upper layers of the TCP/IP stack. Packets which are not addressed to the interface are ignored. However, in *promiscuous mode* on a shared media network (for example with a hub), or connected to a SPAN port, this utility can capture all packets on a network. TCPDump supports operating both in promiscuous mode and normal mode, although the default behavior is to place the card in promiscuous mode when started.

TCPDump uses the libpcap [MLJ94] library, from Lawrence Berkeley National Labs, as the storage format for its capture files. This open source framework serves as a back-end for several network packet tools. The format has become the industry standard for network analysis and packet manipulation tools, and it is supported by e.g., CoralReef and IPAudit. The library is highly versatile and works with both the BSD packet filter and the GNU/Linux sock packet interface.

While TCPDump is an extremely powerful tool, it focuses mainly on TCP/IP protocol, where it does its job well. However, Ethereal is much more versatile and can understand and follow streams of a variety of protocols.

2.4.2 Ethereal and TEthereal

Ethereal [Com04a] and Tethreal are two popular applications for data retrieval and analysis. The first sports a graphical interface, whereas the latter uses a text-mode interface. Hence, TEthereal is similar to TCPDump in many respects. Ethereal is visually pleasing, and the GUI presents the information in a hierarchical way. Perhaps the best feature of Ethereal is that it can follow different IP fragments of the communication between two hosts, and separate

[ii] A.K.A a network sniffer.

that particular stream in a new window for further analysis. The text-mode counterpart supports most of the same features, however it does not provide the same user-friendliness for particular tasks as Ethereal does.

Both of the above-mentioned tools use the Pcap-format for storage, and traces captured with TCPDump and Ethereal/TEthereal can be used interchangeably. An excellent introduction to the capabilities of Ethereal can be found in [Com04b].

2.4.3 Ntop

Ntop [DS00a, DSS+99, DCS+01, DS00b] is a real-time (online) network traffic probe that displays network usage, and a set of network properties, in a way that resembles the UNIX `top` command. Ntop is based on `libpcap` and it has been written in a portable format in order to run on virtually every UNIX platform as well as Microsoft Windows.

Ntop can be interfaced either through a web browser (where Ntop features a stand-alone a web server), where traffic information is presented in a nice and clean GUI, or in a text-mode environment. Ntop will be utilized in the first experimental phase of this thesis, where we want to get a preliminary overview of the traffic patterns. We will also compare results derived using other tools to those of Ntop, due to the fact that we have limited time to redo and verify the results from our experiments. The drawback with Ntop is that it cannot be scripted; hence making it unsuitable for doing analysis where the data needs to be processed before, or after analysis.

2.4.4 Perl and Net::Pcap

Perl [WS90], an acronym for *Practical Extraction and Report Language*, is a powerful scripting language that includes several thousand libraries, thus making it easy to adapt to system administration tasks [BE00]. Perl excels in that it allows for rapid prototyping and testing of advanced functionality. However, due to the fact that Perl is a *scripted language*, it is not particularly fast for extensive numerical calculations. For such tasks, `C` is usually the fastest programming language.

The `Net::Pcap` module is a framework for developing scripts that use the `libpcap` library to interface the trace files directly. We will also use additional libraries, e.g., `Net::Netpacket` for unpacking and working with Ethernet frames and IP packets. Perl shall be the language for our experimental implementations.

2.5 Measurement Properties

2.5.1 Active Performance Metrics

The Internet Engineering Task Force (IETF)'s IPPM Working Group [Gro04] has developed a framework for performance metrics. These metrics will serve as a measure for Quality of Service (QoS) when providers implement different

QoS in their networks. Several others [Jai92, Gog00] are working by some commonly agreed metrics for transport networks. These include:

- Availability
- Error rate
- Response time
- Data throughput

2.5.2 Statistical Properties

However, these metrics are difficult to measure from a passive measurement point-of-view, as they will require active probing of the networks to be of any use. Gogl [Gog00] also concludes that these properties are not adequate for the coarse high-level monitoring and analysis of operational network behavior, and for revealing the internal dynamics of a network. He suggests the following suitable statistical quantities:

- Inter-cell and inter-packet arrival times
- Packet length distributions
- Length of activity periods
- Length of silence periods
- Time between connections
- Duration of connections

We will discuss these properties in the Methodology.

2.5.3 Flows and Packet Trains

The notion of *flows* and *packet trains* are discussed by Claffy and Jain [CBP95, Jai92], and they are closely related[iii]. A flow is a burst of traffic from the same source and heading to the same destination. If the space between two packets exceeds some inter-flow gap, they are said to belong to separate flows. This approach is also known as *timeout-based flow profiling*. Flows are identified by a five tuple consisting of source IP address, source port, destination IP address, destination port, and transport layer protocol. Others have suggested alternative approaches to profiling flows, however these are beyond the scope of this survey, since they are not relevant to our experiments. The motivation for using a flow timeout for profiling flows instead of *state*[iv], is that not all transport

[iii] The differences between the two definitions are, in this context, insignificant, and we will stick with the term flow for the rest of this document.

[iv] TCP support connection states through the SYN-FIN mechanisms.

layer protocols support this. In other words, identifying flows by state is not practically feasible with these protocols.

The motivation for distinguishing between flows and single packets is that routers maintain flow state in order to remember the nature of flows that are passing through them. A single flow can be thought of as a unique *channel* through the network, and there is cost associated with the creation and tear-down of these channels. Hence, understanding the effect of the packet train phenomena is essential to optimizing router efficiency.

2.5.4 Protocol and Application Distributions

Caceres [Cac89] was the first to popularize the idea of counting packets and bytes of data per protocol and application (TCP/UDP port), and presenting the information via histograms. Visualizing and interpreting tables of such information is valuable for network administrators, as it enables the administrator to gain an insight into the usage-pattern on the transport and application layers. It is also useful for implementing QoS and traffic shaping in networks where this is vital to the service level of the network. These tables and histograms are crucial to the building of traffic models, since they are related to other properties of the system, such as bandwidth consumption patterns and the inter-packet arrival time distribution[v]. Morin [Mor03] has developed a framework for shaping P2P traffic in a DOCSIS network[vi], which is based on both the statistical properties of the traffic and application distributions.

2.5.5 Modeling Packet Arrivals – Poisson versus Self-Similarity

The packet inter-arrival time between two packets, Δt_i, is defined as [FHH02]:

$$\Delta t_i = t_{i+1} - t_i \tag{2.1}$$

The distribution of arrival time frequencies is often referred to as the inter-packet arrival time distribution, and has been subject to studies since the early days of networks [JW99]. In networking hardware, such as router, switches and router-switches[vii], there is a fixed overhead per packet being processed. Therefore, knowing the distribution of when packets arrive is of interest to both network gear manufacturers, and the network administrators that configure the equipment. We shall move on to look at different approaches to modeling packet arrivals.

Network packet arrivals have traditionally been modeled as Poisson processes. The Poisson, or exponential, distribution is most commonly used to model a number of random occurrences of some phenomenon in a specified unit of time. Refer to Fig. 2.5 for a plot of the Poisson distribution for a set of continuous observations. There are historical and practical reasons behind

[v] This correlation becomes clear if we look at the inter-packet arrival time distribution of streaming multimedia applications, where we the arrival pattern exhibits large quantities of high-frequency UDP datagrams.

[vi] The technology deployed by cable-modem ISPs.

[vii] Often referred to as *layer 3 switches*.

the widespread acceptance of this assumption – the most prominent being the analytic simplicity of the Poisson distribution. However, a number of traffic studies have shown that packet inter-arrivals are not exponentially distributed [PF95, JR86, DJea92].

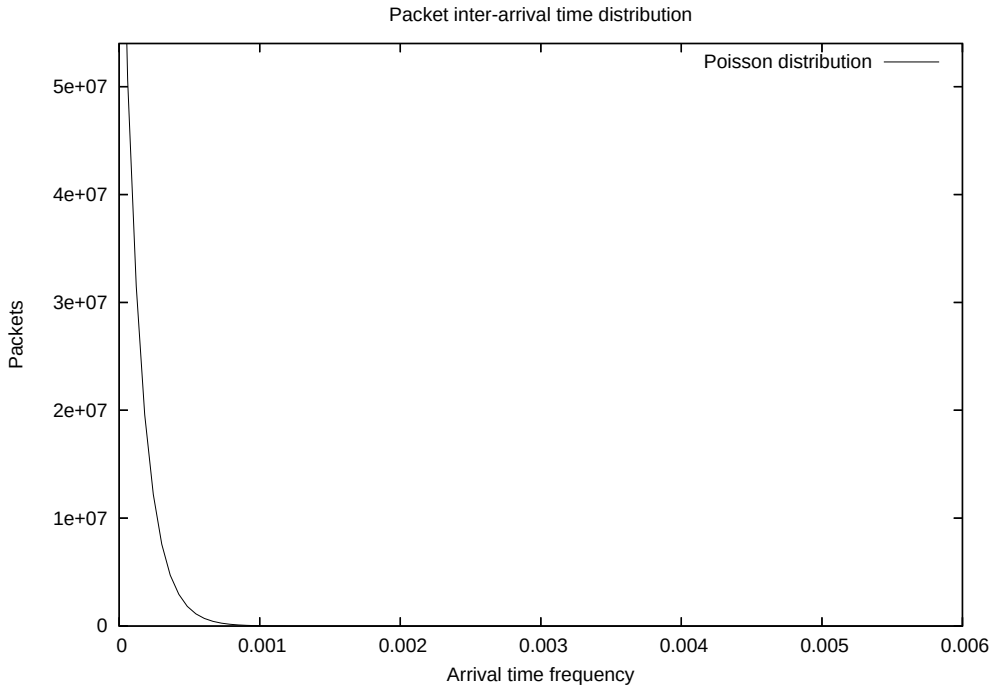

Figure 2.5: Example of the Poisson distribution

Newer studies [KMFB04] have concluded that network traffic can be well represented by the Poisson model for sub-second time scales. At multi-second scales, packet arrivals are better modeled as self-similar processes.

The arrival of packets is assumed to be Poisson distributed if the packets arrive completely at random.

We divide the time t after the previous packet arrival into N parts. This gives N time intervals of duration $\Delta t = \frac{t}{N}$. Assuming that the probability p of a new packet arrival is the same for any interval (random process) gives $p = \lambda \cdot \Delta t$ for some constant λ.

If $P(t)$ is the probability of a packet arriving at time t, then

$$P(t) = (1-p) \cdot (1-p) \cdot ... \cdot (1-p) = (1-p)^N \tag{2.2}$$

is equal to

$$(1 - \lambda \cdot \Delta t)^N = \left(1 - \frac{\lambda \cdot t}{N}\right)^N \tag{2.3}$$

and the limit when $N \to \infty$ is a well known mathematical identity:

$$\lim_{N \to \infty} P(t) = e^{-\lambda t} \tag{2.4}$$

Self-similarity is an ubiquitous phenomenon present in both local area and wide area traffic traces [PKC97, WPT96, PKC97, CB97, EPW95, Gog00]. Self-similar processes with parameters N and s are described as a power-law such as [Wei05]:

$$N = s^d, \tag{2.5}$$

where

$$d = \frac{\ln N}{\ln s} \tag{2.6}$$

is the "dimension" of the scaling law, known as the Hausdorff dimension. A random irregularity is termed self-similar if it remains statistically similar upon a change of length scale [And80]. Self-similarity implies "fractal-like" behavior. For self-similar traffic, there is no natural length for a burst of traffic and traffic bursts appear on a wide range of time scales [WPT96]. Paxson et al [PF95] have found that user-initiated TCP-session arrivals, such as remote login and file-transfers, are well modeled as Poisson processes, whereas other connection arrivals deviate considerably from Poisson. According to [PKC97], transport layer mechanisms are important factors in translating the application layer causality into link traffic self-similarity. Their studies have shown that network performance in terms of throughput, packet loss rate, and packet retransmission rate degrades gradually with increased heavy-tailedness. Queuing delay, response time, and fairness deteriorate more drastically. How much heavy-tailedness affects self-similarity is determined by how well congestion control is able to shape a source traffic into an on-average constant output stream while conserving information.

2.6 CoralReef as a Tool for Network and System Administrators

There are several tools available that perform passive capture and analysis, however they have lacked the feature of flexible real-time network traffic flow monitoring [MKea01, Tea01]. CoralReef evolved from OCXmon monitors that ran on the MS-DOS platform, and supports real-time and offline analysis of both ATM and Ethernet links. Existing tools are typically narrow in scope, designed for specific tasks, e.g., TCPDump [JLM04] and NeTraMet [BZ01]. CoralReef is designed with modularity and easy customization in mind. It provides a clean, consistent user interface for a wide range of network analysis applications, both offline and in real-time. CoralReef is a package of device drivers, libraries, classes and applications [KMea01]. CoralReef was developed and originally only used on FreeBSD, but has since been ported to run on GNU/Linux, Sun Solaris and other UNIX variants. To avoid any compatibility issues, we shall use FreeBSD as the underlying operating system for our CoralReef monitor.

CoralReef includes bindings for C, C++, and Perl, however we shall focus on the command line utilities of CoralReef, as these tools are more than

adequate for our experiments. A detailed overview of the applications in the CoralReef suite can be found in Appendix B.

We will utilize CoralReef to gather the following statistical properties from our traces:

- Packet inter-arrival times, to find the packet inter-arrival time distribution.
- Data rates and rate development.
- Protocol distributions.
- Application distributions.
- Packet size distributions.

The CoralReef applications fall into either of these two categories: The `crl`-applications, which operate on raw packet data, and the `t2_`-applications, which operate on aggregated flow data. The filenames of the binaries are all prefixed by their category [MKea01], and thus by their functionality.

CoralReef supports the `libpcap` library, and subsequently it can read pcap files. The relationship between CoralReef applications, and the interaction with other tools is visualized in Fig. 3.3 in the Methodology chapter. CoralReef also supports a proprietary format, `crl` which is utilized by the suite's own capturing application, `crl_trace`.

2.7 Peer-to-Peer and File-sharing

Peer-to-peer (abbreviated to P2P for the rest of this thesis) networks have become one of the biggest bandwidth consumers on the Internet [Coo04]. The technology provides improved robustness, scalability, and diversity over the standard client/server model, by utilizing methods that reduce the need for central servers to transfer data among the users [AHTea04, RC04]. In Fig. 2.6 we provide a model of a flat (pure) P2P network.

The growing use of P2P is a controversial subject in mainstream media. It is commonly accepted that most of these networks exist for the sole purpose of spreading copyrighted material. However, there are several legitimate areas of utilization for P2P technology, and we have probably only seen the beginning of creative ways of employing it. More and more people are experimenting with use of P2P technology for purposes like telephony, distributed backup, load balancing and more. The distributed IP telephony solution Skype [BS03] is a brilliant example of applied P2P technology for legitimate purposes. RIAA[viii], and other representatives for copyright holders tend to use P2P volume as a direct measure of illegal activities on the Internet, but this is not necessarily an adequate assumption [Coo04].

P2P technology is a double-edged sword for end-user ISPs: Although it fuels the demand for broadband at home, it also drives up the ISP's expenses

[viii] The Recording Industry Association of America (RIAA), `http://www.riaa.com/`.

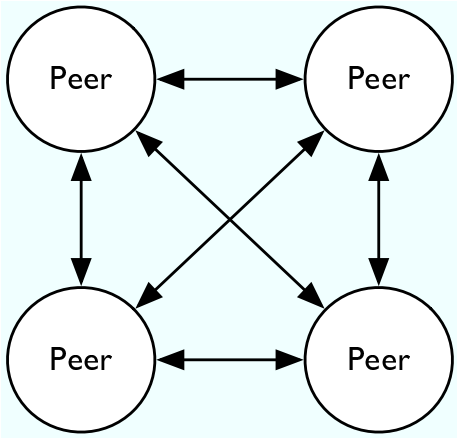

Figure 2.6: A pure P2P network

caused by volume-based charges of upstream providers, and is hence in direct conflict with the flat rate charged by the ISPs [Coo04].

Oslo University College, along with several other academic institutions have prohibited the use of P2P applications entirely (without explicit permit from the board of directors, on a case-by-case basis).

Chapter 3

Methodology

In this chapter we shall introduce the reader to the methodology of our experiments. We begin with the methodology for capturing and processing the data, and move on to discuss the CoralReef suite, which is used for analysis. Several scripts have developed, for the purpose of processing raw trace files and post-processing CoralReef output, and we will look at the use of these scripts further on.

3.1 Process Workflow

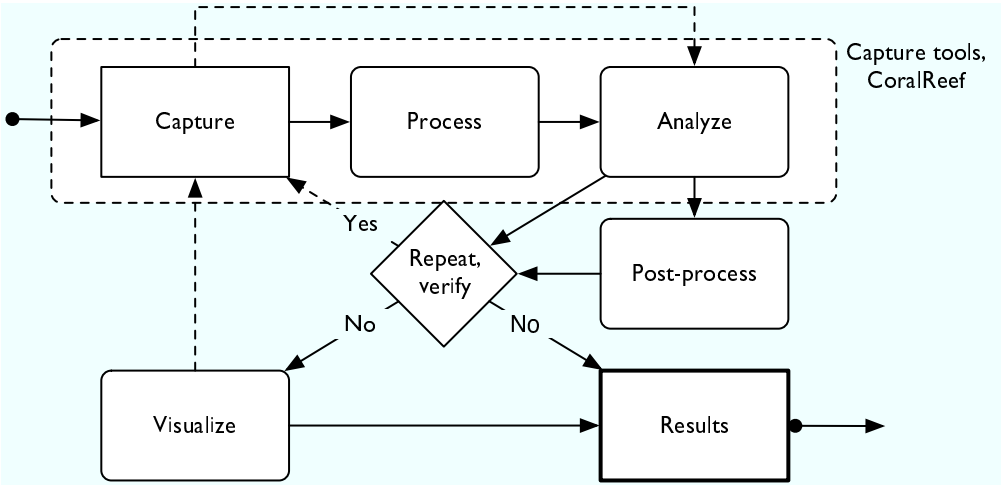

Figure 3.1: Process flow model with five states.

We have developed a model that describes the high-level methodology for performing network measurements in Fig. 3.1. This model incorporates six states: capture, process, analyze, post-process, repeat/verify, visualize, and interpret the results.

Offline measurements start with the capturing of data. The data can range from sub-second measurements, on high-speed links, to long-term measure-

ments. Not even the most sophisticated equipment is able to store all data from high-speed links such as OC48 links for a long time, where data rates often exceed several gigabits per second. On the other hand, it is perfectly feasible to capture and store long-term traffic from a slower link, e.g., a on a link closer to the edge of the network.

The most widely used tools store the traces in a cross-program and cross-platform format such as the popular `pcap` format. The `pcap` format has to a large extent become the industry standard format for traffic traces. The use of cross-program formats enables the network administrator to use output from one program and feed it directly into another program without processing.

Processing includes, but is not limited to:

Converting between formats for cross-program operation, for example between the CoralReef specific `crl` format and `pcap`.

Post-capture filtering to remove irrelevant information, such as filtering only ingress or egress traffic. In our experiments we shall start out with monolithic traces, and break them down to smaller pieces before further analysis.

Sorting and processing raw text data from analysis software, so that it can be fed into a script that does calculations based on the data, or be visualized directly in a program such as `gnuplot`.

When the traces have been processed, the data can be analyzed. This can be accomplished either with off-the-shelf software like Ethereal or the CoralReef software suite, or with custom-made scripts. As our experiments require more than off-the-shelf software, we shall utilize the programming language Perl, and for our payload-inspection, the `Net::Pcap` library as well.

Network analysis software spews out verbose statistics. In many situations, it is necessary to post-process before visualization or interpretation. This includes performing numerical operations on the data, and text-formatting.

A single sample analysis can provide a snapshot of the state of the network. Unless the network is entirely static, results yielded from such an experiment can not be generalized per se. The properties of a network are subject to fluctuating trends, both hourly and seasonal. If the scope of one's experiment is to gain an understanding of how the network behaves over time, it is necessary to redo the experiments several times, at regular and irregular intervals[i].

Raw data are often hard to interpret for the human mind. When the amount of data is fairly small, the results can be visualized with a table, or a bulleted list. In the case of network measurements, the amount of data is usually large — sometimes overwhelming. In such cases, it is easier to interpret the results from looking at a graph, a time-series, a histogram or a pie-chart.

[i] As Albert Einstein said to one of his student assistants who was preparing for an incoming class; "Professor Einstein, what test are we giving them?" To which Einstein replied, "The same test we gave them last week." Bewildered, the student assistant replied, "But Professor Einstein, we already gave that test." Einstein simply said, "Yes, but the answers are different this week."

3.2 System Constraints and Limitations

3.2.1 System Constraints

Network measurements and analysis have several constraints that should be taken into consideration in advance of conducting the experiments. Although online analysis does not require storing traffic traces, it is still subject to the same constraints as offline analysis. However, it does not require storing the capture files. We can summarize these constraints as follows:

- High data rate
- High data volume
- High processing load.

These constraints can make measurements troublesome when the data rate and volume is high, and they can lead to thrashing or contention of monitor resources. In order to handle these constraints, we can consider alternative ways of doing measurements and analysis:

Samples We can perform capture samples, at a fixed or random frequency, instead of continuous measurements.

Filter at capture time In order to reduce the traffic volume, we can filter at capture time to reduce the disk space required, e.g., only specific addresses or ports, the first n bytes or the packet, et cetera.

Reduce the capture time In many high-speed networks, captures are performed at sub-minute scales to reduce the data volume. This can provide a snapshot of the network properties, however fluctuations in traffic patterns will not be revealed.

The high data rate places serious stress on the hardware, in terms of CPU, system I/O, memory and disk throughput. If the monitor hardware is not adequately powerful, we can place the monitor on a link with lower throughput.

There are space limitations to main memory and on disk. When the analysis requires connection tracking or flow tracking, the amount of memory available to the application is a constraint. Using a lower timeout period for flows can help reduce the amount of memory needed.

In certain situations, the data throughput is simply higher than our equipment can process. In such cases, we can perform online analysis to sample, filter, and aggregate. Online analysis can reduce the disk thrashing. However a fast CPU, or even *several* CPUs in a symmetrical multiprocessor environment, is in many situations necessary in order to cope with the packet rate. If the CPU(s) or memory cannot cope with the packet rate, this can result in contention, where none of the packets are processed.

The correlation between the rate of incoming packets, a queue Q_i, and the processing rate (service rate), S_o can be expressed by a ratio R:

$$R = \frac{Q_i}{S_o} \tag{3.1}$$

If the rate of the incoming packets is high, and the processing rate is lower than the incoming packet rate, then $R > 1$, and the queue will grow. A system that is not able to cope with the rate, is by definition an unstable system, where packets will fail to be processed. On the other hand, if $R \leq 1$ the system copes with the data rate without skipping packets, and subsequently we have a stable system.

When observing packets on a network we are dependent on using hardware and measurement tools that are able to keep up with the traffic rate at the measurement point without skipping packets. If the rate is too high for the monitor, the measurement tool should at least report the number of packets which were dropped [Gro01].

Another important constraint, often neglected by network administrators [Gro01], is the use of efficient and robust system software and hardware for the monitor. The cost associated with maintaining broken software and repairing hardware can be a frustrating and time-consuming task, stealing valuable time for the network administrator.

3.3 Assessment of Error and Uncertainty

When conducting network measurements, the results are subject to both random and systematic errors [Bur04].

In our measurements, random errors are usually small and do not contribute significantly to our results. Moreover, random errors tend to fall into stable distributions, hence they will zero out over time. For a vector of measured properties, Σ, we can write the error as $\Delta\Sigma$. So:

$$\Sigma = \langle\Sigma\rangle + \Delta\Sigma \tag{3.2}$$

where

$$\Sigma_i = \begin{pmatrix} \text{Inter-arrival time distribution} \\ \text{Packet size distribution} \\ \text{Protocol distribution} \\ \text{Application distribution} \\ \text{...} \end{pmatrix}$$

and where $\langle\Sigma\rangle$ is the *mean* or *expectation* value of the vector of measurements. On the other hand, systematic errors can contribute significantly to the results, and they are represented by a systematic shift in the true value. However, these errors are much easier to deal with — that is, if they are identified.

We can estimate the effect on our results of an error in Σ, through the following formula:

$$\Sigma = \Sigma(\overline{S}, \overline{R}) \tag{3.3}$$

where Σ is a function of two vectors, $\overline{S}$ and $\overline{R}$. The vectors represent factors that contribute to error and uncertainty, and are defined as:

$$\overline{S} = \begin{pmatrix} \text{Software bus} \\ \text{Resource thrashing/contention} \\ \text{Clock inaccuracy} \\ \text{Packet encryption/encapsulation} \end{pmatrix}.$$

and

$$\overline{R} = \begin{pmatrix} \text{Software bug} \\ \text{Clock synchronization error} \\ \text{Sample rate} \\ \text{Filter expression} \\ \text{Packet encryption/encapsulation} \end{pmatrix}.$$

Note that packet encryption and encapsulation, and software bugs can contribute to both random and systematic errors. We shall not try to put probabilities on these factors, but a general formula is provided to calculate the uncertainty [Bur04]:

$$\Delta\Sigma = \sqrt{\left(\frac{\delta\Sigma}{\delta\overline{S}}\right)\Delta\overline{S}^2 + \left(\frac{\delta\Sigma}{\delta\overline{R}}\right)\Delta\overline{R}^2} \tag{3.4}$$

3.3.1 Analysis Error and Uncertainty Cause Tree

In Fig. 3.2 we have developed a cause tree for errors and uncertainties that are associated with conducting network analysis.

3.3.2 Limitations

When interpreting the results from our experiments, it is important to be aware of the following limitations on the scope of our experiments:

Filtering at capture-time The analysis is performed on two separate networks, over 8 separate traffic traces. We shall capture 6 smaller traces from different hours of the day on different days from the OUC network, and 2 larger traces with long-term traffic data from both networks. Unfortunately we are not able to do short-term captures of the CATCH network. The traffic volume on the OUC network is large, due to Novell LAN traffic, and large quantities of UDP Multicast packets. We are unable to capture all of these data for longer periods of time, so we have chosen to filter out everything except TCP, UDP, and ICMP for our long-term captures. Due to the enormous amount of UDP Multicast packets, these are filtered at capture time. This way we are able to analyze TCP, UDP, and ICMP traffic alone, and unfiltered traffic, independently. Moreover, this provides a more fair basis for comparison between the two networks, since the traffic on the CATCH network is mainly carried over these protocols.

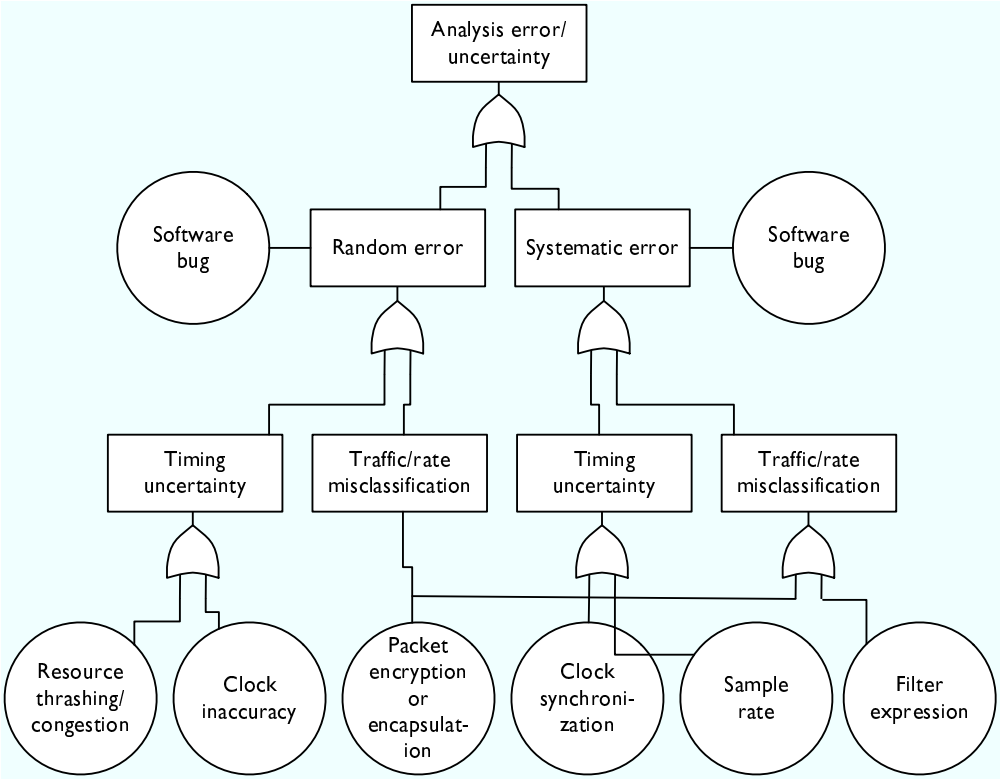

Figure 3.2: Cause tree for errors or uncertainties.

Sampling rate and error The monitor at which the data is captured is not a state-of-the-art computer. It is a regular desktop computer with a standard network interface card. This limits the sampling rate and the accuracy of timestamps to the speed of the processor. The resolution of our capturing hardware is 10 ms. However, the data rates are fairly low, so the computer should have no problems processing the data with sufficient accuracy.

Utilization of the link and time of the day The traces were captured at different days, and at different times of the day. However, the times were selected arbitrarily, and we make certain reservations about special network occurrences on these days. The links were not particularly strained on the days of capture, so the picture may have looked somewhat different if the traces were captured on days with more link utilization.

Verifying results Due to the time limitations on these experiments, repeating and verifying measurements as much as we would like is not feasible. The long-term captures are conducted over the coarse of three days, and can therefore be used for snapshot analysis of that time and place. Such analysis will not reveal fluctuating seasonal trends.

48-byte packets When the captures are recorded, a snap-length of 48 bytes is used for each packet. The headers and payload of a packet combined is usually significantly larger than this. However, we shall only need about 4 bytes of the payload in order to recognize strings associated with P2P traffic. The four extra bytes are included for overhead.

Encapsulation Packets that are encapsulated, e.g., with IP-IP encapsulation, will not include any payload as a result of the snap-length. Hence, the payload can not be inspected in our P2P characterization experiment. This can result in false negatives, and represents an uncertainty in our measurements.

Encryption Some P2P protocols have implemented encryption technologies in order to provide additional privacy for the users of the network. This is analogous to the encapsulation and is not accounted for, and may result in false negatives. This represents an uncertainty in our measurements.

3.4 Capturing Methodology

3.4.1 Point of Measurements

If we take a look back at Fig. 1.1 and Fig. 1.2, our captures are recorded at the end-switch, and on the access provider network, for the OUC traces and the CATCH traces, respectively. We have access to a SPAN port on the Cisco Catalyst switch that connects the student network at Oslo University College. The data traces from the CATCH network is pre-captured for us.

3.4.2 Scale of Measurements

We have chosen to distribute the analysis over 8 sets of data traces. We shall acquire six sets of short-term captures (approximately 6 hours each) from different days from the OUC network, and two sets of short-term captures (over the coarse of several days), one from each network.

3.4.3 Hardware

The university has no hardware monitor equipment, so we shall build a software monitor for our experiments. Moreover, a standard PC with a fast network interface card is adequately powerful to cope with the throughput of our networks, and they operate at a sufficient sample rate for an academic study, where absolute timing accuracy is not critical. In Table 3.1, we provide the specifications of our capturing and analysis hardware. We move on to discuss how the software can be used to achieve our goals.

3.4.4 Software

All data traces are captured and analyzed using Free Software or Open Source tools.

They are captured using a computer running the Debian flavor of GNU/Linux, with a dedicated NIC for traffic capturing. By placing the network interface card in promiscuous mode, the card dumps all packets on the wire, not only packets directed to the host. TCPDump and Ethereal are used interchangeably, as they both store the data in the `pcap` binary format. We shall take a look at the cross-program format `pcap`, and the `libpcap` library in general in the next section.

The platform used for analysis with the CoralReef suite is a computer with similar specifications, although running the FreeBSD 4.11 OS with an optimized kernel.

Platform	CPU	RAM	OS	Disk interface	Size
Capture/OUC	Intel 1.0 Ghz	512MB	Linux 2.6	ATA-66	120 GB
Capture/CC	Intel 2.4 Ghz	2GB	Linux 2.6	UW-SCSI	272 GB
Analysis	AMD 1.0 Ghz	768MB	FreeBSD 4.11	ATA-100	320 GB

Table 3.1: Hardware specifications

3.4.5 Libpcap

Libpcap is a portable library that provides a packet filtering mechanism based on the BSD packet filter (BPF). It is the core component of TCPDump, and is developed by the same team. We shall interface `libpcap` through a Perl module known as `Net::Pcap` available. This module has been in beta version

for several years now, and has sufficient functionality to make it usable for our experiments. The Ethereal network analyzer also utilizes the `libpcap` library.

3.4.6 Capturing with TCPDump

In this section we will discuss how we can utilize TCPDump to capture traffic from a network interface in promiscuous mode. The manual page [JLM04] provides more verbose reference material for the curious reader. We will cover the relevant functionality to ease prospective reproduction of our experiments.

A Simple Example

TCPDump puts the network interface card in promiscuous mode by default (i.e., capturing *all* traffic, not only traffic directed to the host itself). However, it can also run without being in promiscuous mode, using the `-p` option. Continuous capturing on a given network interface card can be achieved with the following command:

Example 1 — *Capturing with TCPDump*

```
$ sudo tcpdump -i eth1 -w outfile.dump
```

This captures `everything` on the wire. For our short-term captures, we want to capture all traffic, except that we want to limit the size of the packet to 48 bytes. This can be achieved by adding the `-s 48` option.

Capturing TCP, UDP ICMP, and Dropping Multicast

As we have outlined above, we shall use a packet snap-length of 48 bytes. Continuous capturing on the `eth1` interface card, enforcing a snap-length of 48 bytes, writing to a dump-file, filtering out everything except TCP, UDP, ICMP, and ignoring UDP Multicast traffic, can be achieved with the following command:

Example 2 — *Capturing with TCPDump*

```
$ sudo tcpdump -i eth1 -s 48 -w ouc.dump ip proto \
 \(\\tcp or \\udp or \\icmp\) and \(not Multicast\)
```

Note that the identifiers `tcp`, `udp`, and `icmp` are keywords and must be escaped with backslash (\), or \\ in the C-shell.

In the default setup of both GNU/Linux and FreeBSD, access to the packet capture pseudo device, e.g., `/dev/le` is restricted to the superuser (`root`). The `sudo` prefix tells the command to run TCPDump as superuser, without being logged in as `root`.

3.4.7 Using Ethereal and TEthereal

The same result as we produced above could also be achieved using TEthereal. The advantage with TEthereal and Ethereal over TCPDump, is that we can utilize a *ring buffer mode*, where TEthereal times out, and starts on a new file, after a given criteria, such as after n minutes or gigabytes. The output-file is annotated with a timestamp. We can use ring buffer mode by passing the `-b n` option, where `n` is the number of ring buffer files, unsuitable for analysis by our scripts without further processing.

Example 3 — *Capturing with TEthereal*

```
$ sudo tethereal -i eth1 -s 48 -b 5 duration:270000 -w \
ouc.dump -f ip proto \(\\tcp or \\udp or \\icmp\) and
\(not Multicast\)
```

The same result can be achieved using the point-and-click interface in Ethereal.

Ethereal and TEthereal have a few more advanced features than TCPDump, making it a more attractive application in many respects. However, TEthereal has a legacy built-in file-size limit of 2 GB, making it unusable for these environments without using a ring buffer. We shall use TEthereal for our short-term traces, so we can try the experiments on smaller files. On the other hand, we shall use TCPDump for our long-term traces. Using ring buffer mode for long-term captures would result in numerous trace files.

3.5 Processing Methodology

3.5.1 Filtering Ingress and Egress Traffic

When we look at application distributions, we want to look at ingress (inbound) and egress (outbound) traffic separately. In our monitor setup, we capture traffic going in both directions. This can ultimately result in a distorted port distribution, as a result of hosts communicating with random source ports. This example filters out egress traffic from the CATCH trace:

Example 4 — *Filtering out ingress traffic with TCPDump*

```
$ sudo tcpdump -n -r catch.dump -w catch-inbound src \
net 201.153/19 and not dst net 201.153/19
```

In the data traces from CATCH, the IP-addresses were systematically scrambled. TCPDump will hang for a long time when trying to look up invalid IP addresses in DNS. By using the `-n` option, we force TCPDump to not look up DNS names for the IP addresses. The methodology is nearly identical for filtering out ingress packets in OUC data traces, except we filter out the class B networks `128.39.73/24`, `128.39.74/24`, and `128.39.75/24` instead.

3.6 Analysis and Post-Processing Methodology

We will start off by looking more formally on the properties we are to measure, and will move on to discuss how this can be achieved using CoralReef and our own scripts.

3.6.1 Traffic Rate and Volume

Traffic rate quantities can be measured in three different ways: packet rate, byte rate, and flow rate. We want to differentiate between these quantities, because neither of these alone is a sufficient measure of the data rate. We shall explain why in this section.

Per Byte

In network equipment there is a fixed overhead for processing each byte. The volume in bytes is therefore an important measure for the network administrator. Knowing the byte rate in a network is also a question of dimensioning and planning. Moreover, upstream providers usually charge by byte. We shall look at the byte rate development of the long-term data traces.

Per Packet

There is also a fixed overhead for processing each packet in a router or switch. The number of packets passing through the node is therefore relevant along with the volume in bytes. We shall look at the packet rate development of the long-term data traces.

Per Flow

The notion of a flow represents communication where two hosts communicate beyond one packet. In most situations, two hosts send several packets both ways. A flow is built up from a train of IP packets. There is cost associated with the creation and tearing down of these virtual channels. The flow can be characterized by various criteria [Peu02, CBP95]; packet inter-packet arrival time (timeout-based), address granularity, and states. In these experiments we shall use a timeout-based approach, with a flow timeout of 64 seconds. This is the most commonly used timeout period for timeout-based flow-characterization, and the choice is supported by the methodology in [CBP95]. We shall look at the flow rate development in the long-term data traces.

3.6.2 Traffic per Protocol

The protocol distribution is the percent-wise distribution of protocols in our two environments. We shall look at the distribution of TCP, UDP, and ICMP, and see if we can find differences between the networks.

3.6.3 Traffic per Application (TCP Destination Port)

Several sources have reported that we are seeing a shifting trend in the usage pattern of the Internet, where people are moving from traditional applications, such as HTTP (the web), FTP (file-transfer), and SMTP (email), to newer and more bandwidth demanding protocols like P2P file-sharing applications and streaming multimedia. We want to investigate whether or not this hypothesis is correct, and to see if there are measureable differences between the two networks. We shall look at the destination ports of the egress traffic of both sets of data traces.

3.6.4 Packet Size Distributions

Matsumoto et al [MTH90] has investigated the effect of using different functions for estimating the packet size distribution under various loads, and how it affects a system's performance. There are both per-packet and per-byte components of the cost of routing or switching a packet, so knowing the packet size distribution allows designers to optimize hardware and software architectures around relevant benchmarks.

3.6.5 Packet Inter-Arrival Time Distribution

The packet inter-arrival time distribution is the distribution of inter-arrival times between packets seen on the network. Designers of a high-speed networking equipment might want to know the shortest and most common packet inter-arrival times and the percentages of packets at those times. This, so that they can design a switch that can switch at the appropriate number of packets per second. From a scientific point-of-view, it is also desirable to determine a suitable model of when packets arrive over different time scales. We shall find the packet inter-arrival time distribution for both sets of data traces.

3.6.6 Using the CoralReef Suite Applications

CoralReef is a comprehensive software suite developed by CAIDA to collect and analyze data from passive Internet traffic monitors. CoralReef supports both capturing and reading in its own format, `crl`, and reading from trace files in the `pcap` format. It supports capture and analysis of both ATM cells and Ethernet packets. Fig. 3.3 shows the flow to and between the CoralReef applications. In this section we shall look at how CoralReef can be utilized to measure the properties we have listed above.

The CAIDA toolkit consists of several tools, both command line tools and libraries for C, C++ and Perl. However, the command line tools provide sufficient functionality for our experiments. We have provided a table of the functionality of all `crl` applications used in our experiments in Appendix B. Considering our *offline* approach, we shall not use CoralReef's online capabilities, and only feed the applications with pre-captured data traces. The CoralReef

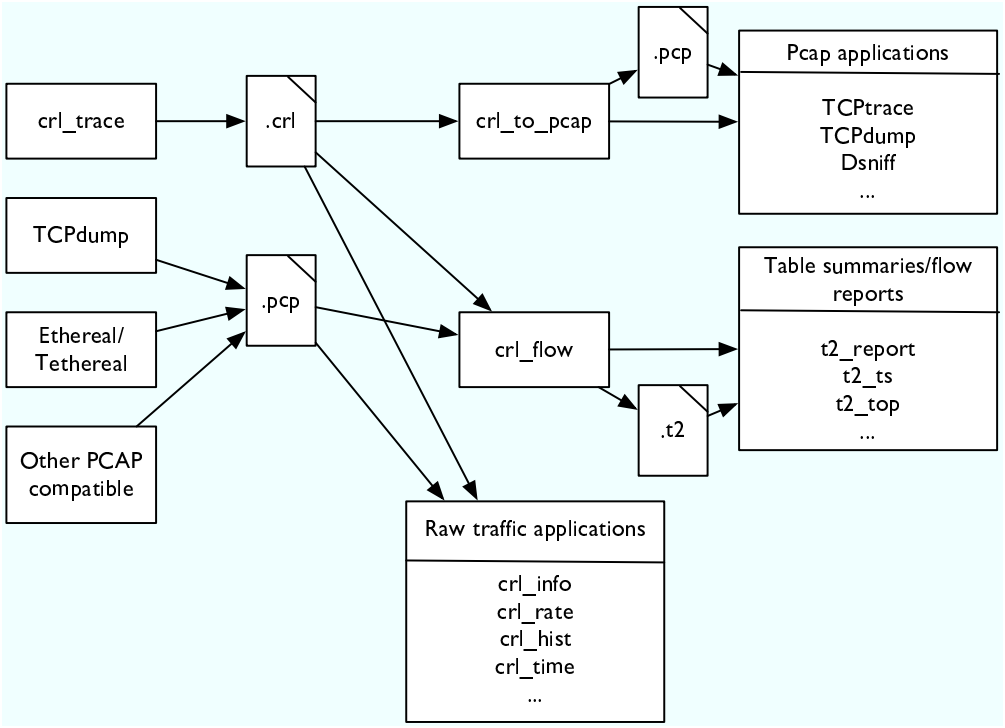

Figure 3.3: Overview of flow between CoralReef applications

suite is split up into several classes. We shall cover applications from the following classes: Utilities, Dynamic reports, Static reports, and Traffic flow applications.

The utilities include applications for capturing traces, converting between formates, and to extract inter-arrival times.

The dynamic reports can generate reports from live data on the fly. However, the dynamic reports can also be generated from static data traces.

The static reports can only be generated from data trace files stored in `pcap` or `crl`, and includes applications for generating inter-AS matrices, packet and byte counts by IP length and protocol, port summary matrices, and more.

Traffic flow applications extracts and aggregates flow data from a series of measurements.

In the coming sections, we shall look at how the applications are used at the command line, and on how we can utilize our custom scripts to extract and process the raw output.

Several options are common to the `crl` applications, e.g., the `-C` option. This option takes different parameters, such as `-C 'filter expression'`. The application can also be stopped after a specified duration using `-Cd=time`. The option `if:fxp0` tells the applications to use the interface `fxp0`, a native Ethernet interface. If `if:` is not specified, the `crl` application assumes we are working on a trace file. The applications that start with `t2_` are different from the `crl` applications in that it generates reports from the output of `crl`

applications.

Sample output from all relevant applications are provided in Appendix A.2.

Utilities

crl_info The `crl_info` is a utility that prints out information about trace files, such as hardware, the iomode it was in, the interface it is on, the hardware revision, the bandwidth of the link, et cetera.

Example 5 — *Using the* `crl_info` *application.*

```
$ crl_info -di tracefile
```

Running this command will return general information about the trace, along with the first and last timestamp of the packets.

crl_time `crl_time` traverses a `pcap` file and spews out one line per packet or cell with: Interface, relative cell/packet number, timestamp, and time difference between each cell/packet.

Example 6 — *Using the* `crl_time` *application.*

```
$ crl_time -p tracefile
```

This command will cause `crl_time` to use packet timestamps instead of ATM cell timestamps.

Dynamic Reports

crl_rate The `crl_rate` application count packets and bytes on interfaces, subinterfaces or IEEE802.1Q VLANs at a given interval. The default interval is 60 seconds. In the example, the option `-Ci=30`, specifies a local averaged interval of half a minute (30 seconds). This application is used to extract data rates, per packet and per byte, from our data traces.

Example 7 — *Using the* `crl_rate` *application.*

```
$ crl_rate -Ci=60 -4 tracefile -o output
```

Running this command will print the IPv4 rate over an interval of 60 seconds to an output file `output`.

Static Reports

crl_hist The `crl_hist` generates a report on packet and byte counts by IP length and protocol, port summary matrices for TCP and UDP, fragment counts by protocol, packet length histograms for the entire trace and for a list of applications, and the top 10 source and destination port numbers seen for TCP and UDP traffic. We use the `crl_hist` application to extract the protocol, application and packet size distributions.

Example 8 — *Using the* `crl_hist` *application.*

```
$ crl_hist tracefile
```

Running this command will cause `crl_hist` to generate a histogram that can be post-processed or visualized by a program like `gnuplot`.

Traffic Flow Applications

crl_flow The `crl_flow` application generates summaries of traffic flow data for post-processing by `t2_rate` or other `t2` applications.

t2_rate The `crl_rate` aggregates flow data from `crl_flow` and presents flow count and rate on a given interval. The interval is specified with `-Ci=interval`. The timeout for each flow is specified with `-Tf<time>`. Other criteria than timeout are also supported. We use both these application to extract flow counts and flow rates from our data traces.

Example 9 — *Using the* `crl_flow` *and* `t2_rate` *applications.*

```
$ crl_flow -Ci=1000 -b -Tf64 tracefile |t2_rate -s \
-o flows.txt
```

This command will report the number of flow count and rate over an interval of 1000 seconds, with a flow timeout period of 64 seconds. The `-b` option tells `crl_flow` to use a binary output format that is interpreted by `t2_rate`.

3.6.7 Analysis and Post-Processing Scripts

CoralReef applications will often spew out a plethora of information about several properties of the traffic simultaneously. We are normally not interested in all fields of the output at the same time, so we have developed a set of scripts that extracts relevant fields from CoralReef, and that presents the data in a format that is post-processing and visualization-friendly. The scripts are simple shell pr Perl scripts, however they will significantly ease mass-reproduction of the experiments on several data traces.

Packet and Byte Rate

rate.sh The `rate.sh` script extracts the relevant fields for calculating packet and byte rates and presents it in a format that can be plotted directly using e.g., `gnuplot`. The data is normalized through the `normalizer.pl` script that is described below.

Example 10 — *Using the* `rate.sh` *script.*

```
$ rate.sh tracefile|normalizer.pl > output
```

Packet Inter-Arrival Time Distribution

inter-arrival.sh The script `inter-arrival.sh` extracts the relevant fields from `crl_time` and puts the inter-arrival times into bins. Then, the number of unique inter-arrival times are counted, and the information is presented in a format that can be plotted as a distribution with e.g., `gnuplot`.

Example 11 — *Using the* `inter-arrival.sh` *script.*

```
$ inter-arrival.sh tracefile > output
```

3.6.8 Sorting and Processing Output

We have developed a set of scripts for processing the data, either for further analysis or for visualization. These scripts are placed in Appendix C. However, we will discuss the most important scripts here, and how they relate to the CoralReef suite.

normalizer.pl The `normalizer.pl` script is used for normalizing output from `rate.sh`. The output from `rate.sh` rate is expressed in megabits/s, kilobit/s, and bytes/s, respectively. This script converts all rates to megabit/s, to ease visualization or further post-processing. Data is read from STDIN.

utilization.pl This script is a simple script that calculates the mean utilization of a link. It takes output from `rate.sh`. The output file from `rate.sh` is specified with the `-r` option, and the capacity of the link is specified with the `-b` option.

Example 12 — *Using the* `utilization.pl` *script.*

```
$ utilization.pl -r ratefile -b 100
```

local-avg.pl The script `local-avg.pl` takes output from `rate.sh` and calculates local average values for a given time interval. In addition to the time in minutes and the byte or packet rate, this script outputs a third column that can be used for plotting error bars. The `ratefile` and `output` files are specified with the `-r` and `-o` options, respectively. The interval in which the script shall coarse-grain over is specified with th `-i` option. We can add an offset to the x-axis using the `-f` option to visualize several traces in the same time series.

Example 13 — *Using the* `local-avg.pl` *script.*

```
$ local-avg.pl -r ratefile -o outputfile -i 6000 \
-f 25
```

3.7 Determining a Suitable Packet-Arrival Model

We have developed a script, `self-sim.pl`, that takes input from `crl_rate` and calculates the $\frac{R(n)}{S(n)}$ values for each interval, where

$$R(n) = \max(0, W_1, W_2, \ldots, W_i, \ldots, W_n) - \min(0, W_1, W_2, \ldots, W_i, \ldots, W_n) \tag{3.5}$$

and

$$W_i = \sum_{k=1}^{i} (X_k - \langle X \rangle) \tag{3.6}$$

$S(n)$ is the sample variance for the interval.

The output from this script can be used to calculate the Hurst exponent [MVN97, GB99, DC98, Bur04, Gog00] of the time series.

Example 14 — *Using the* `self-sim.pl` *script.*

```
$ self-sim.pl timeseriesfile
```

3.8 A Methodology for Estimating P2P Usage

We have developed a Perl script that uses the `Net::Pcap` interface to `libpcap`. This script is known as `inspect.pl` and can be found in Appendix C.3.3. This script inspects all packets in a `pcap` file for a series of criteria. These criteria are derived from previous work in [Ora01]:

- TCP destination port number.
- UDP destination port number.
- Payload content.

If the TCP or UDP destination port number matches that of any given P2P network, the packet is classified as P2P. If the port number is not matched, we inspect the payload for a set of known ASCII strings associated with P2P traffic. We want to evaluate if inspecting the packet content will yield results in identifying P2P traffic beyond those identified by port number. The script is generic in that it is trivial to add more protocols, ports or ASCII strings to the arrays of known P2P protocols.

Example 15 — *Using the* `inspect.pl` *script.*

```
$ inspect.pl -p -r tracefile -o reportfile.txt
```

Running the script with the command `-p` option will enable packet inspection.

When the script is finshed inspecting all packets in the `pcap` file, it generates a report with some statstics. Fig. 3.4 shows the output from the report.

```
                General statistics
.-#-Total--------#-Identified--pct--#-Non-TCP----.
| 148890          15020      10.08  480          |
`-----------------------------------------------'

         Protocol breakdown by P2P network
.-Network----------------#-Packets------Cum-%----.
| Kazaa/Fasttrack:       6074             4.08    |
| Edonkey/clones:        2948             1.98    |
| WinMX/Napster:         0                0.00    |
| Bittorrent:            5985             4.02    |
| Gnutella:              0                0.00    |
| DirectConnect          0                0.00    |
|-------------------------------------------------|
| Total:                 148890          10.08    |
`-------------------------------------------------'
```

Figure 3.4: A report generated by the inspect.pl script

Chapter 4

Results

In this chapter, we shall look at the results from our experiments. We will start off with a few words about what we expect the results to be like, and we move on to look at the actual results.

4.1 Expected Results

In our experiments, we expect to find that:

Proposition 1 *There are measurable differences between the traffic characteristics of an ISP network (mostly WAN/MAN traffic) and a college network (mixed LAN/WAN traffic), in terms of network traffic attributes and statistical properties.*

Given the nature of the two networks, we expect to see measurable differences between the two environments. First and foremost, because the OUC network has both LAN and WAN traffic. And secondly because the user population is quite different. However, most of the LAN traffic was filtered out at capture time, as a result of the fact that most of the LAN traffic is associated with the Novell network, which is carried over a proprietary Novell protocol. This conclusion was drawn after studying the characteristics of the traffic in Ntop.

Proposition 2 *Newer and more bandwidth-demanding applications, such as streaming multimedia and file-sharing applications, are responsible for more of the aggregated traffic volume than traditional services such as HTTP, FTP, and SMTP.*

Several sources in the mainstream media have reported that we are seeing a shift in the usage-pattern of the Internet. New ways of utilizing network bandwidth are fronted by dubious P2P networks whose content are of a varying degree of lawfulness. We suggest that the introduction of new applications and ways of using the network is reflected in the application distribution. We expect to see diverging patterns between the OUC traces and the CC traces. We shall compare our findings with a trace analyzed by Sprint in San Jose, August, 2000(SJ-00) [Cor00].

In addition, the protocol distribution reveals interesting properties of the traffic, because traffic such as streaming multimedia exhibits large quantities of low-frequency state-less UDP packets, as opposed to user-initiated stateful TCP sessions. Researchers have predicted that IP Multicast will take over as the primary means of transport for high-quality streaming multimedia [Alm00]. Multicast is more effective than Unicast in that it can send a stream of datagrams to a group of hosts within the same subnet through a single stream, rather than to single hosts through several streams. However, Multicast has not yet been implemented by most end-user ISPs. Multicast implies some technical challenges, as it has problems with traversing the ever-more-popular NAT "firewalls". The OUC network supports IP Multicast — The CATCH network does not. We shall evaluate if there exists an actual demand for IP Multicast.

Proposition 3 *In contrast to claims from mainstream media, popular P2P applications are as popular as ever. However, they have evolved from using fixed port numbers for communication, to using arbitrary ports, thus making it difficult to achieve accurate measurements of the scale of usage.*

The use of P2P applications have moved from being generally accepted by the public in the days of Napster, to being regarded, by most, as a shady enterprise run by cynical criminals. This comes as a result of the ongoing information war between the file-sharers and the copyright holders[i]. Enterprises are introducing strict firewall rules and heavy traffic queuing on known P2P ports, and P2P software developers are having an increasingly hard time trying to evade these mechanisms. Most of the P2P protocols, perhaps only except from the open source P2P protocol Gnutella, are based on a non-disclosed architecture, hence information about the inner workings of such protocols will have to be acquired through reverse-engineering the software. Recent surveys from independent parties have concluded that P2P usage has dropped significantly over the last few years [Ora01]. This is supported through studies conducted in [Coo04]. We investigate if developers of such software have adapted to stricter policies and learnt how to evade the filters by using dynamic port numbers — or even reserved ports like the www port (80) to cloak the actual traffic. We will evaluate if inspecting payload, in addition to port-based identification, will yield results in revealing packets that that belong to P2P streams.

4.2 General Remarks about Visualization

Most the results from the analysis have been visualized with tables, graphs, time seris, and pie charts. The graphs and pie charts are created using interactive plotting and graphing software such as `gnuplot` or `xmgr` in combination with shell scripts (Bash).

[i] The most prominent spokesmen being the music and movie industry.

A common problem with visualization is that the amount of information is too large to display all relevant data in a single graph. Either because there are too many data sources, or that, in the example of time series, peaks distort the overall trends. Therefore, some of the graphs have been reduced to capture only the important details. This is stated explicitly in the text. The scripts used for plotting the data are available in the appendices.

4.3 Data Traces

The traces analyzed in these experiments were captured from two separate networks: the student network at Oslo University College, and on a medium-sized router in the ISP infrastructure of CATCH Communications, more precisely in the Lillestrøm area in Norway. The long-term CATCH trace was recorded over the course of approximately 75 hours, or 4500 minutes. The OUC trace was recorded over 48 hours, or 2800 minutes. The CC1L trace was started March 22, 2005 at 13:00, and the OUC1L trace was started on March 29, 2005 at 13:30. Table 4.1 summarizes some relevant information about the traces analyzed in this study. CC and OUC are acronyms for CATCH Communications and Oslo University College, respectively. The -L and -S postfix indicates if it is a long-term or short-term capture. All traffic is captured on the Ethernet layer, although Multicast traffic is filtered out at capture-time in the long-term captures.

Set	Date	Start	Dur. (h)	Flows	Packets	Bytes	Mn. util.
CC1L	2005-03-22	13:12	76	10.5M	234M	134.9G	22%
OUC1L	2005-03-30	13:30	48	1.9M	139M	69.7G	4%
OUC1S	2005-02-28	21:10	6	60K	13M	11.6 G	4.5%
OUC2S	2005-03-01	03:08	6	54K	13M	11.8G	4.5%
OUC3S	2005-03-01	09:07	6	52K	13M	11.3G	4.5%
OUC4S	2005-03-01	15:04	6	52K	13M	11G	4.5%
OUC5S	2005-03-01	21:01	6	53K	13M	11G	4.5%
OUC6S	2005-03-01	03:00	6	44K	13M	11G	4.5%

Table 4.1: The traces used in this analysis

In order to maintain privacy for the customers, the IP addresses from the CATCH data-set has been anonymized and substituted by the fictitious network 201.153.0.0/19. Additional CATCH addresses are also scrambled, however, the relative topology of all CATCH networks has been preserved throughout the trace-file. Table 4.2 represents the CATCH address space:

24.37.128.0/18	214.80.0.0/16	201.153.0.0/19
24.152.0.0/16	214.82.0.0/16	201.234.32.0/19
106.96.64.0/18	214.92.0.0/17	212.253.0.0/16
194.18.160.0/19	214.71.0.0/16	212.7.0.0/16
212.6.0.0/17	214.70.0.0/17	

Table 4.2: Anonymized IP adresses in CC1L

Example 16 — *Sample output from he CC1L trace*
The command `tcpdump -r catch.dump` *yields (output altered to fit here):*

```
154.134.223.193.51736 > 201.153.10.80.2546: [|tcp]
154.134.223.193.51736 > 201.153.10.80.2546: [|tcp]
25.18.97.175.5662 > 201.153.10.97.2119: [|tcp]
201.153.10.104.1405 > 81.82.244.50.2341: [|tcp]
4.162.76.40550 > 201.153.10.84.1025: [|tcp]
201.153.10.77.16450 > 194.28.20.182.17244: udp 172
201.153.10.117.62042 > 204.47.192.4.6112: udp 23
...
```

These networks generate between 6 and 7 gigabytes of header data every twenty-four hours.

As we can seen from Fig. 4.1, the OUC trace files contain less data than the CATCH data trace due to the shorter capture time. However, the OUC network generates more traffic on average (measured in bytes).

4.4 Traffic Rate

In the following sections we shall look at the traffic rates in packets, bytes and flows and comment on each of them.

The OUC network generates more traffic on average, than the CC network, and expresses more bursty behavior interspersed with non-bursty behavior. We shall come back to discuss these properties later.

We have measured data rate development for our long-term data traces in three different quantities: per packet, per byte and per flow. The per packet and per byte time series are plotted both in their original form and with error bars. The error bars indicate the standard deviation, $\pm\sigma$, of the interval, and are generated using the `local-avg.pl` script. The error bars make it easier to interpret the rate development, and it gives a good picture of the burstiness within the interval.

The flow rate is plotted every 1000 seconds, and hence we have not plotted error bars.

4.4.1 Per Packet

The total number of packets is $N_c = 2.34 \cdot 10^8$ in the CC1L trace, and $N_o = 1.39 \cdot 10^8$ in the OUC1L trace. The average number of packets per second is $R_c = 852.33$ and $R_o = 801.93$ respectively. In Fig. 4.1 we have visualized the raw packet rate as reported by CoralReef for each 60 second interval. The time series is hard to interpret due to the overwhelming amount of plotting points.

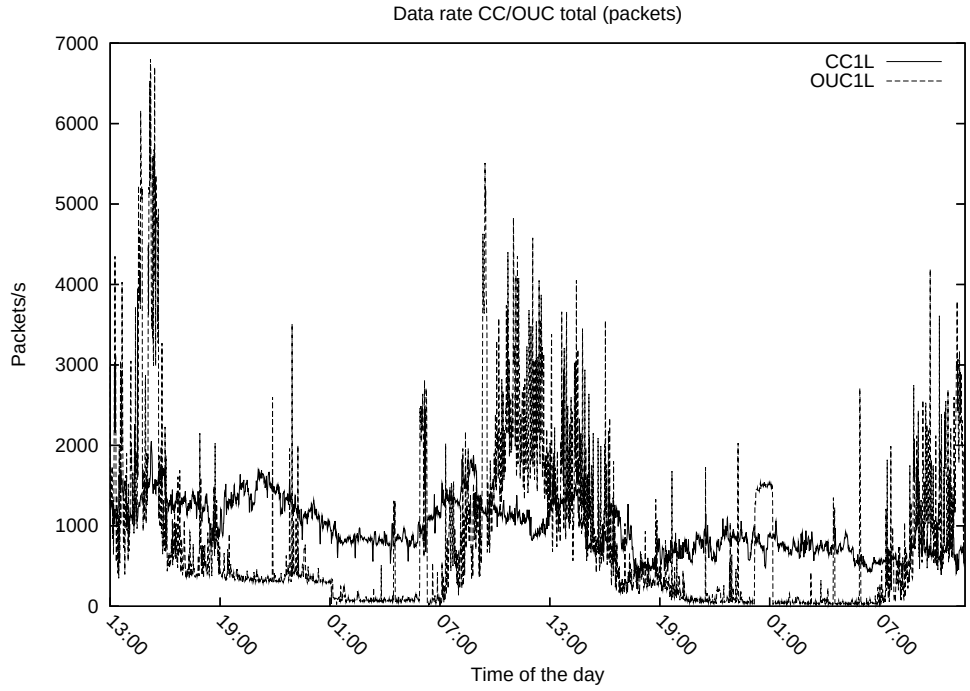

Figure 4.1: 60s average packet rate of CC1L and OUC1L

The CC1L and OUC1L traces has an average standard deviation of $\overline{\sigma}_c = 217.34$ and $\overline{\sigma}_o = 308.47$, respectively. Put simply, this shows that the OUC1L trace exhibits more bursty behavior in terms of packet rate than the CC1L trace
.

Using the `local-avg.pl` script like in the example below, we have calculated local average values and standard deviations, or more precisely the square root of the bias-corrected variance, of each interval in the long-term traces. The coarse-grained time series is visualized in Fig. 4.2. Note that we have added a 25 minute offset to the CC1L trace to make it easier to interpret both data traces in the same time series.

Example 17 — *Coarse-graining with* `local-avg.pl`.

```
$ local-avg.pl -r ratefile -o outfile -i 6000 -f 25
```

We notice that the rate in both data traces peak at around noon. However, the OUC network has longer periods of inactivity during the night. We believe

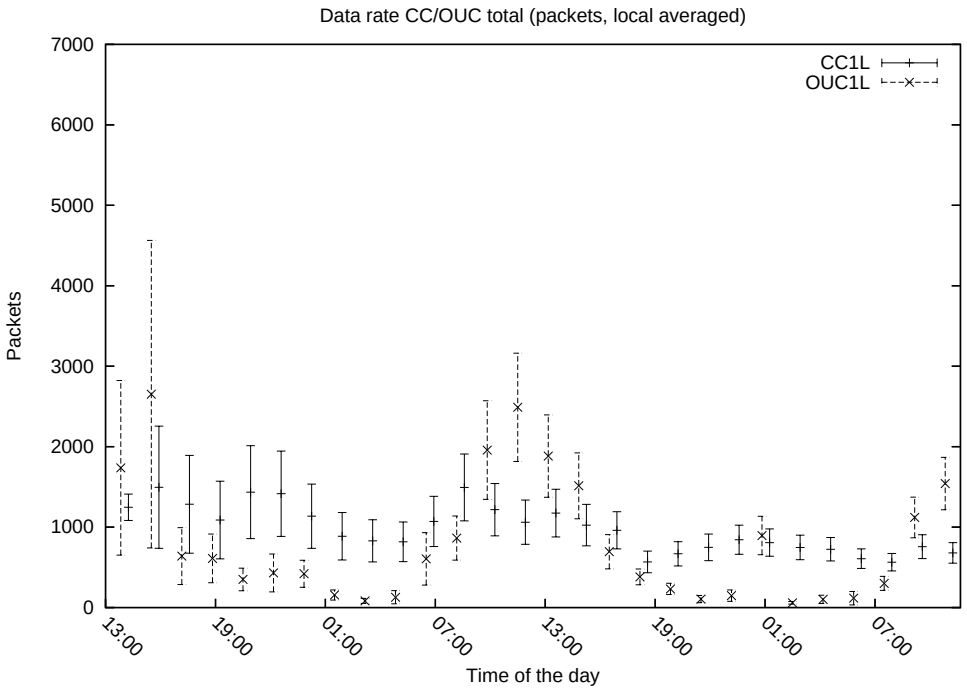

Figure 4.2: 6000s average packet rate of CC1L and OUC1L with error bars

this is caused by P2P traffic, which normally generate an almost constantly high packet rate.

4.4.2 Per Byte

The total volume of the data traces is $S_c = 134.85$ GB for the CC1L trace and $S_o = 69.72$ GB for the OUC1L trace. The average bit rate is therefore $R_c = 2.09$ megabit/s for the CC1L trace, and $R_o = 3.29$ megabit/s for the OUC1L trace.

The raw data rate development has been plotted in Fig. 4.3. As with the packet rate, the time series is hard to interpret due to the overwhelming amount of plotting points.

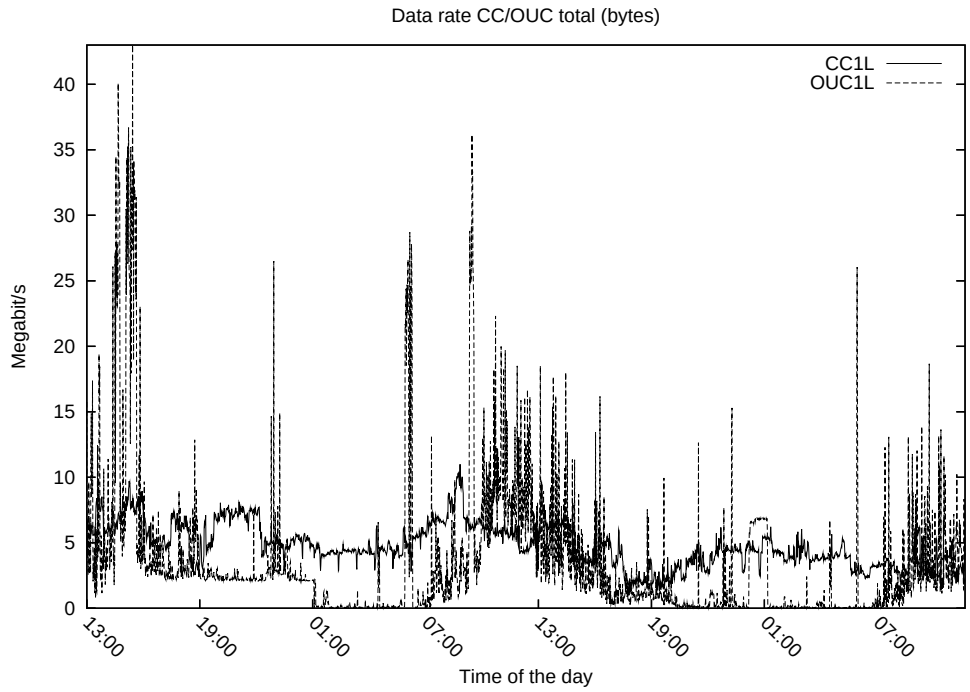

Figure 4.3: 60s average data rate of CC1L and OUC1L

In Fig. 4.4 we have provided a coarse-grained time series. The time series is produced using the same methodology as above.

The CC1L and OUC1L traces has an average standard deviation of $\overline{\sigma}_c = 8.56$ and $\overline{\sigma}_o = 13.20$, respectively (in megabit/s). Again, this shows that the OUC1L trace exhibits more bursty behavior.

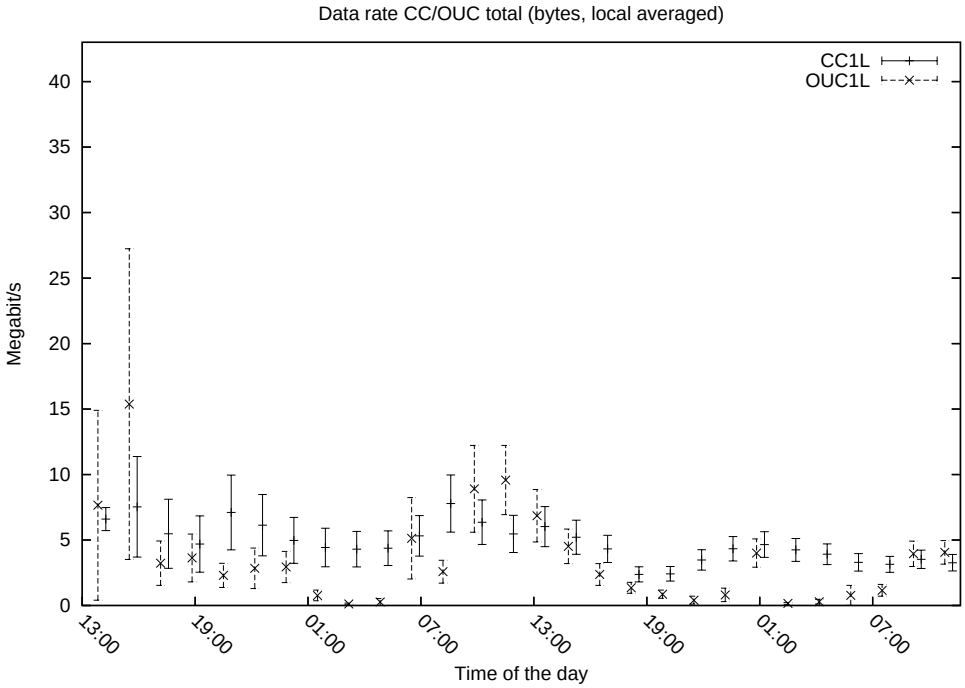

Figure 4.4: 6000s average data rate of CC1L and OUC1L with error bars

4.4.3 Per Flow

The total number of flows in the CC1L trace is $F_c = 1.02 \cdot 10^7$, and $F_o = 1.78 \cdot 10^6$ for the OUC1L trace.

The average number of flows per second is $R_c = 37.37$ in the CC1L trace, and $R_o = 10.24$ in the OUC1L trace.

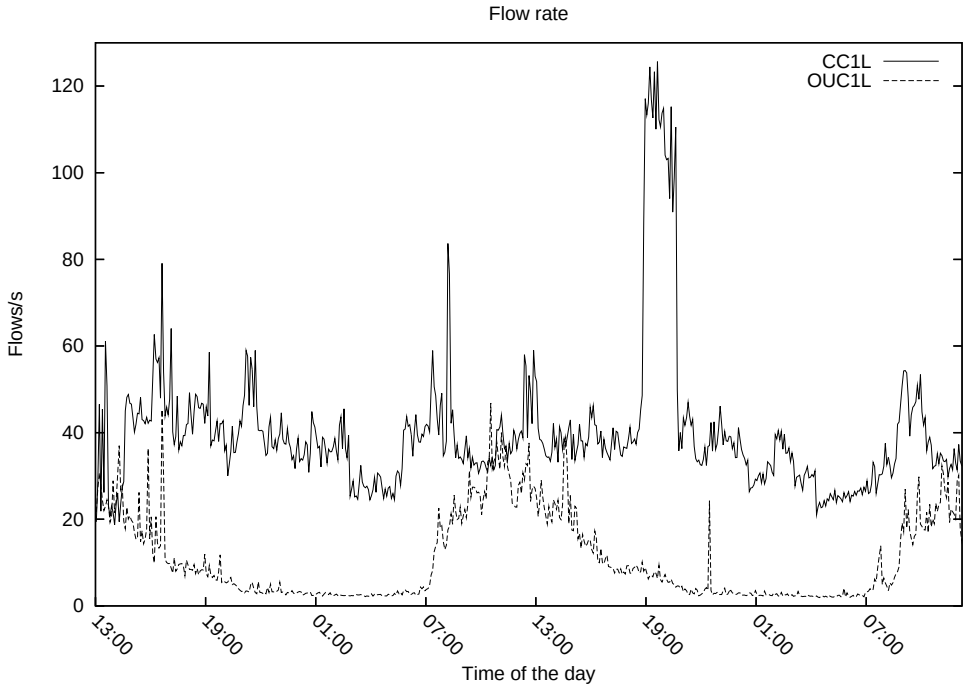

Figure 4.5: 300s average flow rate for CC1L and OUC1L (64s timeout)

As we see from Fig. 4.5 the CC1L trace exhibits a more constant flow rate, whereas the OUC1L trace follows a more traditional pattern, with a high flow rate during business hours, and a low flow rate during off-hours. Note that the combined time series only shows the first 48 hours of the traces, since this is the total length of the OUC1L trace. We can see that although the CC1L trace has an average packet and byte rate that is lower than the OUC1L trace, the CC1L trace has an average flow rate that is nearly 4 times higher. Bittorrent and other P2P applications are characterized [PNB03] partly by a constantly high flow rate. It is not unusual for the Bittorrent protocol to maintain several hundred `ESTABLISHED` sockets simultaneously. We believe that the high number of flows in the CC1L trace is primarily caused by the widespread use of P2P applications in this network, and this is supported by our findings in the application distribution for the data trace (Fig. 4.4). Traditional applications such as HTTP and FTP do not exhibit the same pattern.

We also see that the number of flows peaks at different times in the two long-term traces. While the OUC1L trace has a peak around 13:00, the CC1L trace has the most flows in the early evening, around 19:00. However, this dif-

fers significantly from the packet rate peak, where both traces peak at around 13:00. We suggest that this burst is caused by people surfing the web and using P2P applications when they come home from school or work. This generates a great number of flows.

4.5 Traffic per Protocol

We have identified the protocol distribution for all sets of data traces. We provide the results in Table 4.3. The protocol fields that contains *N/A* indicate that the protocol was filtered out during capture time.

Prot.	CC1L	OUC1L	OUC1S	OUC2S	OUC3S	OUC4S	OUC5S	OUC6S
TCP	89.634	93.829	0.154	0.006	0.207	0.201	0.004	0.264
UDP	10.146	6.095	99.493	99.631	99.348	99.368	99.597	99.337
ESP	0.135	N/A	0.000	0.000	0.000	0.000	0.000	0.000
ICMP	0.081	0.076	0.016	0.031	0.024	0.025	0.017	0.019
GRE	0.003	N/A	0.000	0.000	0.000	0.000	0.000	0.000
IGMP	0.000	N/A	0.331	0.327	0.416	0.401	0.376	0.375
PIM	0.000	N/A	0.006	0.006	0.005	0.005	0.006	0.005

Table 4.3: Protocol Distribution

If we look at the short-term traces (OUC1-6S), we see that UDP is by far the most used protocol. After some investigation, we have found that these UDP packets are mostly Multicast traffic, and they come at an almost constant rate. Common for all these packets is that they come at a low frequency, 0.0007 seconds, and that they originate from two hosts at Østfold University College. The Multicast streams alone generate around 40-50 megabit/s. Less common protocols like ICMP, IGMP, and PIM are also present. IGMP is a protocol that is used to establish host memberships in particular Multicast groups on a single network. The PIM protocol is also associated with the Multicast traffic, as it is a Multicasting routing protocol that runs over the existing Unicast infrastructure.

The overwhelming amount of UDP Multicast traffic is fairly surprising, and it is in stark contrast to what we initially expected to find. We believe that this reflects the demand for Multicast, and that it, when introduced, can save ISPs a lot of money in upstream bandwidth cost.

If we look at the relative distribution between TCP, UDP, and ICMP (Fig. 4.6) for the two long-term data traces, where the multicast traffic is filtered out, we find that the OUC1L trace has a slightly higher percentage of TCP packets than the CC1L trace. The number of ICMP packets is nearly equal.

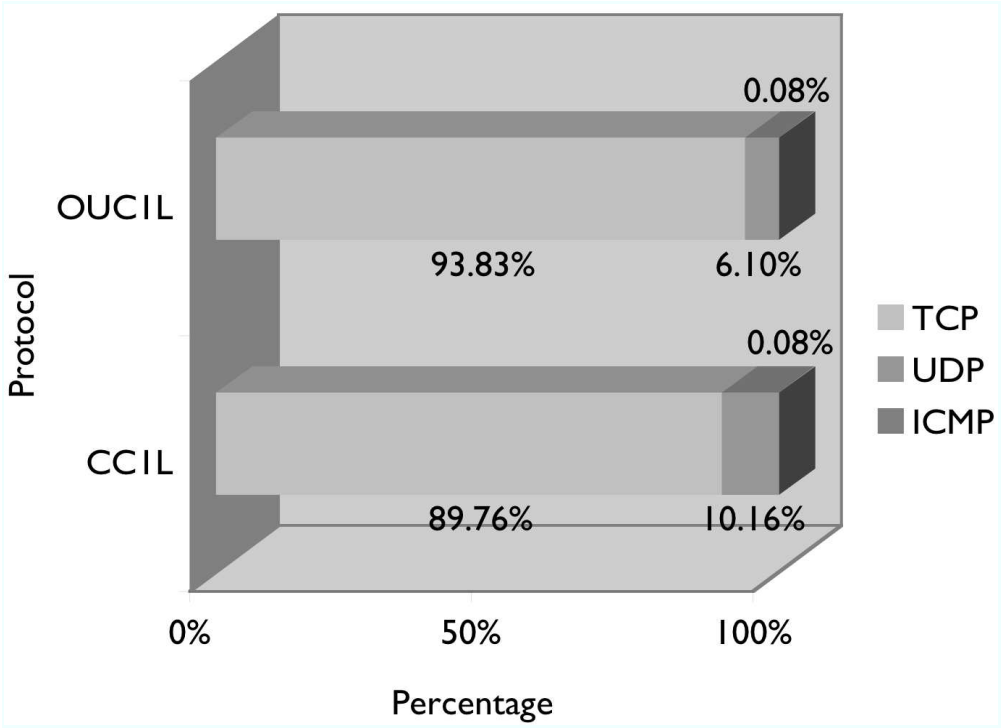

Figure 4.6: Protocol distribution, OUC1L/CC1L (TCP, UDP, and ICMP)

4.6 Traffic per Application

We have measured the TCP destination port distribution for egress traffic in the OUC1L and CC1L data traces. The procedure for filtering out ingress traffic is described in the post-processing methodology. We have visualized the results in Table 4.4 and Table 4.5, although only the 10 most interesting ports are included. The table provides information about port, the resolved service (where available), the number of packets, the number of packets in percentage, the number of bytes, the number of bytes in percentage, and average packet size.

Port	Resolved service	Packets	Pct.	Bytes	Pct.	Avg. size
4662	edonkey	10M	9.93	8.2G	10.69	768
64850	Not known	3M	2.79	3.7G	4.79	1223
6881	bittorrent	1.7M	1.54	1.4G	1.75	809
6346	gnutella-svc	1.4M	1.34	1G	1.34	712
63943	Not known	614K	0.57	811M	1.06	1322
3256	cpqrpm-agent	610K	0.57	796M	1.04	1304
15642	Not known	575K	0.53	758M	0.99	1319
1996	trrsrbport	531K	0.49	751M	0.98	1414
11095	Not known	528K	0.49	743M	0.97	1405
4888	Not known	511K	0.47	677M	0.88	1326

Table 4.4: TCP packet and byte counts by dport — CC1L (egress)

Edonkey is a popular P2P network with clients for all the major operating systems. Clients include Emule, Edonkey2000, MLDonkey, and Shareaza. In the CC1L data trace we see that Edonkey is the service that, by far, generates the most traffic, with around 10% measured in both packets and bytes. Bittorrent is another popular P2P network, and it is responsible for around 2% of the aggregated traffic. Gnutella generates approximately 1.5% of the packets. The accumulated percentage of positively identified P2P traffic is around 12.81% of the packets, and 13.78% of the byte volume. However, we must assume that a lot of the traffic is not identified correctly with this fairly primitive methodology.

Port 64850, which generated 3.5 GB of the traffic, could not be identified, but chances are high that this is outgoing traffic to a random source port on a single host. The OUC network differs from the CATCH network in that it actually offers services to the public, through e.g., the student UNIX login server `cube.iu.hio.no`. However, there are no technical obstructions that prevent private end-users in the CATCH network from running services on their own computers.

Port	Resolved service	Packets	Pct.	Bytes	Pct.	Avg. size
20	ftpdata	6.86M	32.64	9.7G	58.13	1415
80	http	2.91M	13.84	379M	2.27	130
3793	Not known	188K	0.89	199M	1.19	1062
61597	Not known	178K	0.85	189M	1.14	1065
54626	Not known	151K	0.72	161M	0.96	1066
4318	Not known	1.7M	7.92	131M	0.78	78
443	https	297K	1.41	127M	0.76	426
16461	Not known	85K	0.41	126M	0.75	1476
15105	Not known	80K	0.38	115M	0.69	1440
1613	netbill-keyrep	82K	0.39	87M	0.53	1071

Table 4.5: TCP packet and byte counts by dport — OUC1L (egress)

In the OUC1L trace we see a more traditional distribution, with FTPdata and HTTP being responsible for a total of nearly 45% of the aggregated traffic in packets and around 60% of the traffic in bytes. FTPdata generates by far the most traffic in bytes, with 58% of the data, and HTTP on second place with 2.3%. Several ports are unidentified (3793, 61597, 54626, ...), however we can assume that they are random source ports at foreign hosts using services in the OUC student network.

As we can see there are significant differences between the application distribution in the two networks. While the OUC1L trace is dominated by FTP, HTTP, and HTTPS traffic, the CC1L trace has mostly P2P traffic. Subsequently, the results are in accordance with our propositions.

If we compare our findings with the Sprint SJ-00 data trace, we see that there are considerably less HTTP traffic. The amount of FTP traffic is significantly higher in the OUC1L trace than in the Sprint trace, and lower than in the CC1L trace. The use of file-sharing applications is almost the same in the Sprint trace and the OUC1L trace, albeit considerably less than in the CC1L trace.

If we look at the short-term traces and compare them to the Sprint trace, we see that streaming multimedia is far more widespread in the OUC network. Streaming multimedia is not on the top 10-list in the CC1L trace, and neither is DNS or SMTP (email) in any of the traces.

4.7 Packet Size Distributions

The packet size distribution has been plotted in Fig. 4.7 and Fig. 4.8 for the CC1L and OUC1L traces. Packet sizes vary between 1 and 1500 bytes with an MTU of 1500 in both networks. In the first figure, the cumulative percentage of packets against packet size has been plotted, and in the second figure, the cumulative percentage of bytes against packet size.

In the CC1L trace, the most common packet size is 40 bytes, with $\langle N_c \rangle = 6.72 \cdot 10^7$, or 27.04% of the packets. In the OUC1L trace, the most common packet size is 1500 bytes, with $\langle N_c \rangle = 3.76 \cdot 10^7$ packets, or 28.88% of the packets.

We can see from the figure that over 60% of the packets in the OUC1L trace, and around 50% in the CC1L trace, are less than 200 bytes in size. Approximately 70% of the packets are shorter than 1450 bytes in both traces, and the remaining 30% of the packets are between 1450 and1500 bytes long.

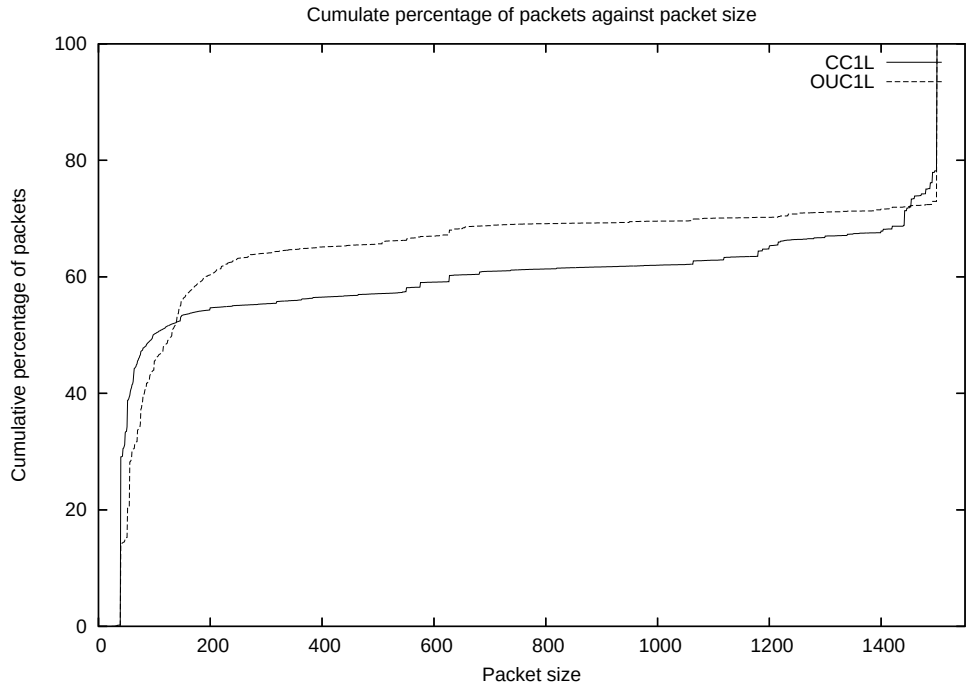

Figure 4.7: Cumulate percentage of packets against packet size

If we look at the cumulative percentage of bytes per packet size in Fig. 4.8, we see that only 20% of the traffic in bytes is carried with packets that are 1200 bytes or shorter in both traces. The remaining 80% is carried with packets that are between 1200 and 1500 bytes long. However, in the OUC1L trace 40% of the total number of bytes is carried with packets 1450 bytes or shorter, in contrast to 22% in the CC1L trace.

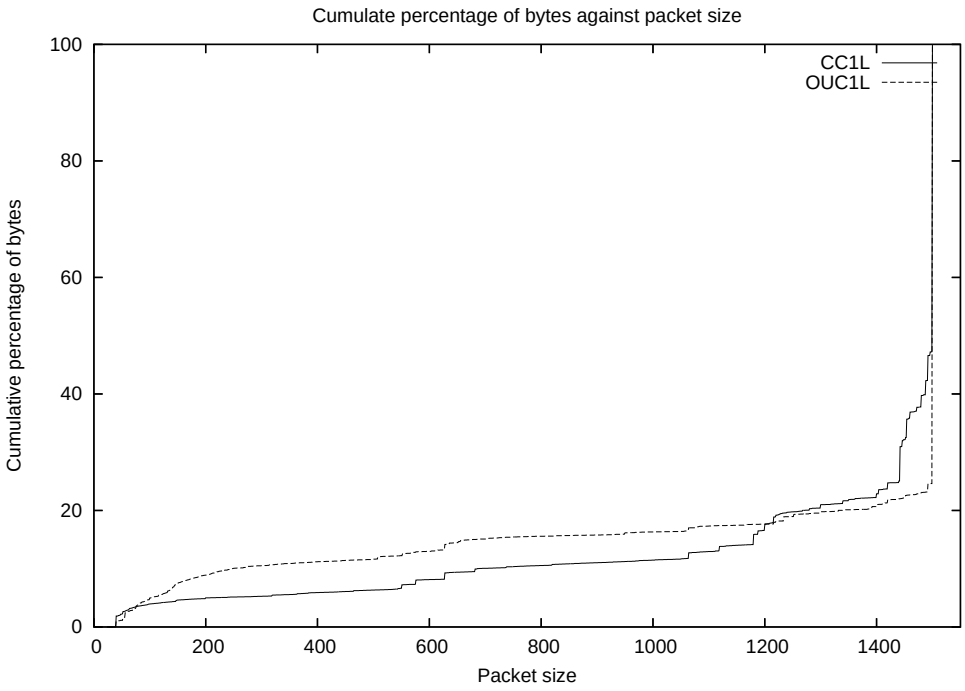

Figure 4.8: Cumulate percentage of bytes against packet size

4.8 Packet Inter-Arrival Times

We have discussed the presence of large quantities of UDP Multicast packets in the OUC network. As we can see from Fig. 4.9, these packets cause the inter-arrival pattern of the network to deviate significantly from the Poisson distribution. The distribution for all of the small data traces have been plotted on the same figure, since they exhibit more or less the same pattern.

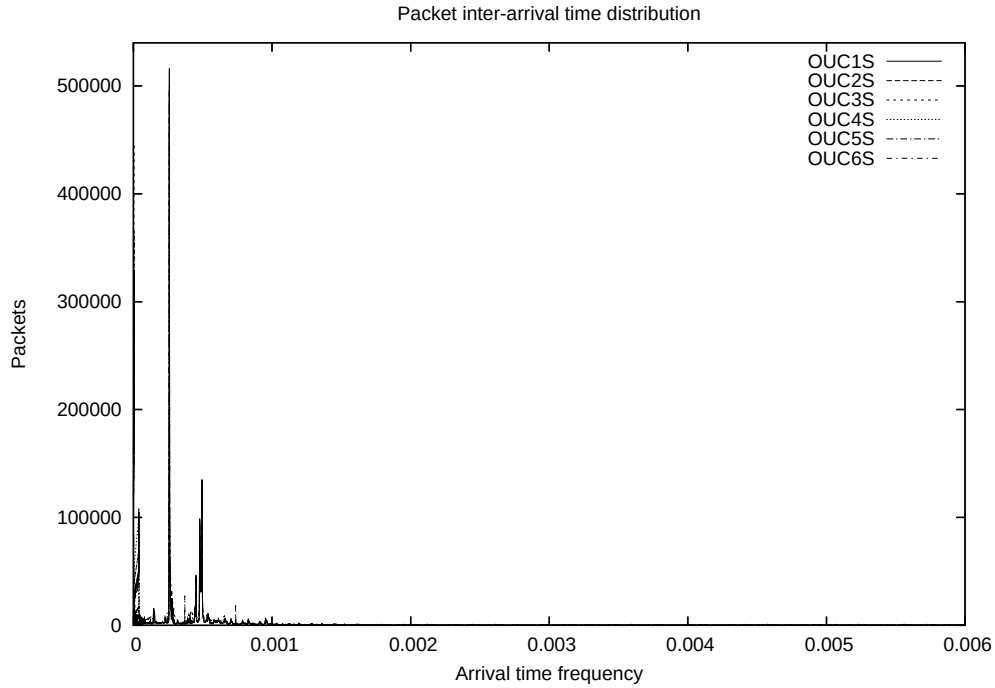

Figure 4.9: Packet inter-arrival time distribution OUC1-6S

4.8.1 Modeling as Poisson Process

If we look at the long-term captures in Fig. 4.10 and Fig. 4.11, we can see that pattern looks more like the Poisson distribution. This is because only TCP, UDP, and ICMP traffic has been captured in the long-term traces, and Multicast traffic is filtered out. The Poisson model fits well for the OUC1L trace, except for some long-tailed behavior. However, the CC1L trace exhibits a significantly more heavy-tailed distribution on the same scale, making the Poisson model unsuitable for the WAN traffic in the trace.

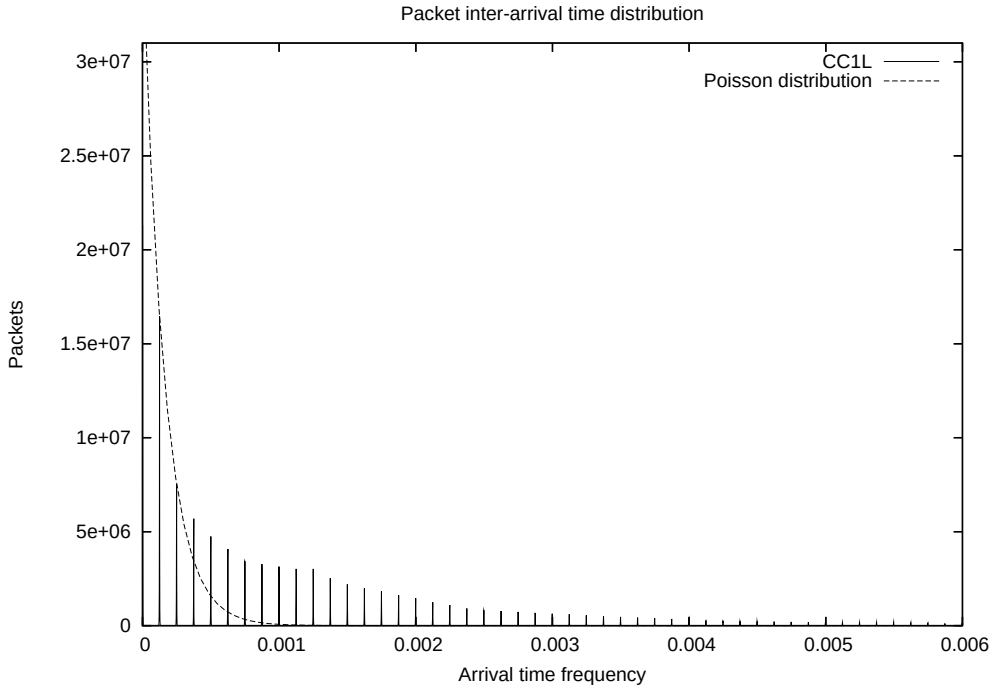

Figure 4.10: Packet inter-arrival time distribution CC1L

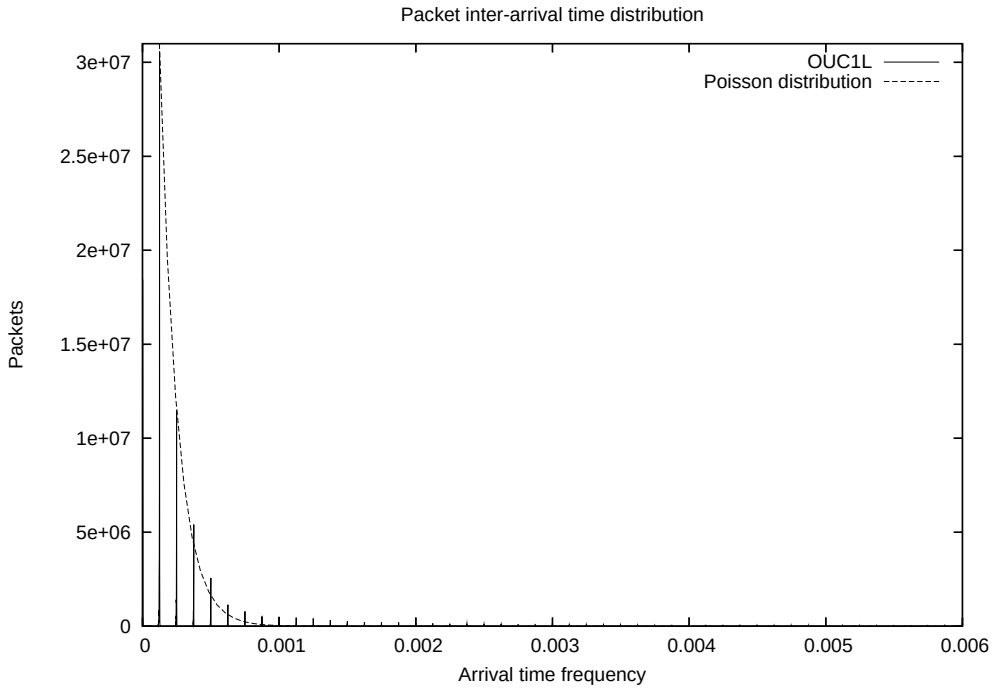

Figure 4.11: Packet inter-arrival time distribution OUC1L

4.8.2 Timestamp Oddities

As we can see from the figures, the traffic appears to come at fixed time intervals, as opposed to the almost continuous graph in Fig. 4.9. It appears almost as if the resolution of the time stamps is higher in the first distribution. This is not correct, as there are several packets that arrive at intervals between the peaks in the graph. However, these packets are not numerous enough to show along the y-axis, due to the enormous amount of packets in the trace that comes at these fixed intervals. We are unsure as to why the traffic pattern appears this way, although we operate with three theories: This could be due to uncertainty in the placement of timestamps, when the packets are received from the SPAN port on the monitor. Another theory is that: In the long-term traces, traffic was filtered out at capture time. It could be that the capture hardware is not capable of the same time stamp accuracy when filters are applied during capture time, although we could not find evidence that supports this in the documentation of the capture software.

We believe it is more likely that the data is buffered internally at the monitor, and placed into bins before it is given a timestamp. When the packet rate is high, the buffer fills up faster, and traffic is passed on through to the timestamp mechanism at higher frequencies. On the other hand, when the packet rate is low, it takes some time before the buffer is filled, and data is sent along for timestamping. This is analogous to the way TCPdump caches data up before writing to disk, although we could not find hard evidence to prove it. Subsequently, it could be an interesting subject for further research.

4.8.3 Modeling as a Self-Similar Process

We have tried modeling the inter-arrival time distribution as a self-similar process. This is achieved using the `self-sim.pl` script, and the data is plotted on a log-scale in Fig. 4.12 and Fig. 4.13. The Hurst exponent is defined as the difference quotient of the regression line. The OUC1L trace has a Hurst exponent $H_o = 0.30294$, and the CC1L has $H_c = 0.14858$. Processes with a Gaussian profile has $H \simeq \frac{1}{2}$. According to Burgess [Bur04], observations with a $0 < H < \frac{1}{2}$ have short-range dependencies and the correlations sum to zero.

Local- and wide-area-network-driven measurements tend to show a value of $H > \frac{1}{2}$.

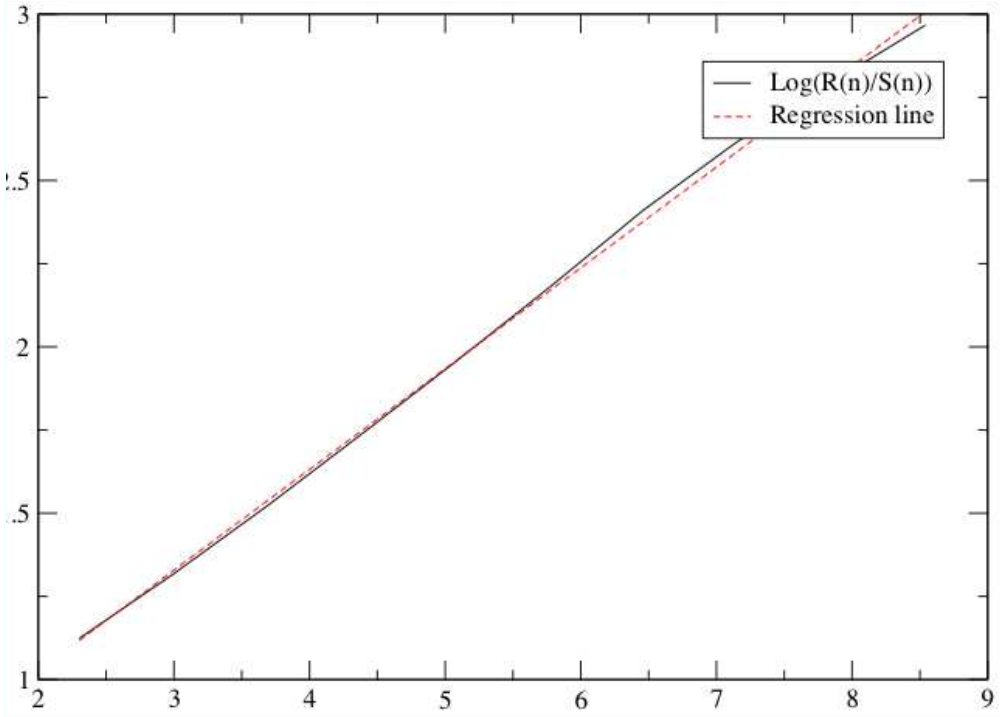

Figure 4.12: Modeling as self-similar process – OUC1L

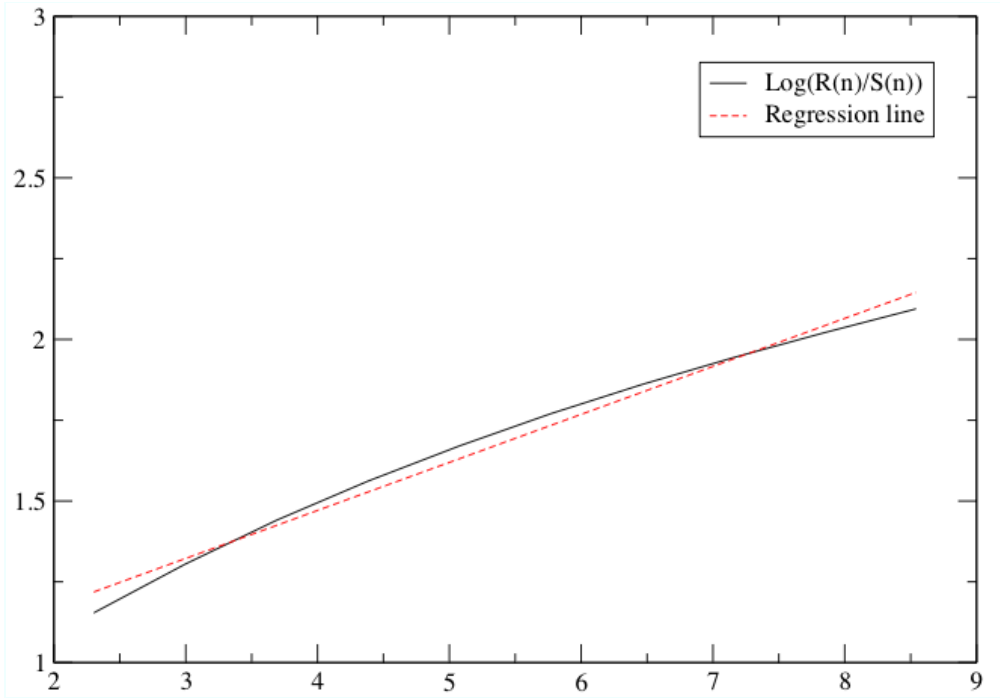

Figure 4.13: Modeling as self-similar process – CC1L

4.9 Estimating P2P Volume

We have identified P2P traffic using our `inspect.pl` script. The script does classification based on destination port number and payload. In this experiment we have not differentiated between ingress and egress traffic, and we ran the script on both the long-term traces. Our approach is a simple per packet method, and we have not implemented connection tracking or byte counting, although the script can easily be expanded with this functionality.

Network	Packets	Cum pct.
Kazaa/Fasttrack	6429	0
Edonkey/clones	33365	0.02
WinMX/Napster	1393	0
Bittorrent	106547	0
Gnutella	4711005	3.39
Total	139102237	3.41

Table 4.6: P2P volume (ingress and egress)– OUC1L

As we can see from Fig. 4.6 and Fig. 4.7, the CC1L trace has slightly more P2P traffic than OUC1L, with 10.04 %, as opposed to 3.41 % in the OUC1L trace, using this methodology. Edonkey is by far the most widely used P2P protocol in the CC1L trace, whereas the Gnutella protocol dominates the OUC1L trace.

Network	Packets	Cum pct.
Kazaa/Fasttrack	19391	0.01
Edonkey/clones	21827413	9.34
WinMX/Napster	20185	0.01
Bittorrent	2757660	0.01
Gnutella	1562699	0.67
Total	233613782	10.04

Table 4.7: P2P volume (ingress and egress) – CC1L

When we ran the `inspect.pl` script on the long-term traces, we could not find a single packet that contained payload associated with P2P traffic which was not already identified by the destination port. This could mean either of three things; the protocols have changed their routines for negotiating and initiating file transfers, the protocols have implemented encryption to cloak the traffic, or — the protocols are still using known ports.

Packets with a destination port that is known to be P2P traffic are automatically classified as P2P, and payload is not inspected. We modified the script to do payload inspection, regardless of destination port and ran it on a smaller

trace. Our results revealed that at least some protocols are still using the same mechanisms for negotiating and initiating file transfer, as several packets contained the strings our script searches for. Although this is a strong indication that P2P networks have not moved to use random port numbers, we can not be absolutely sure, as the number of packets containing these strings was surprisingly low. This is also a subject for further research.

Chapter 5

Conclusions and Further Work

In this thesis we have investigated the characteristics of two heterogeneous networks. We have limited the study to offline passive analysis, distributed over 8 sets of network traffic traces. Our results have revealed that there are measurable differences between the networks, both in terms of statistical properties, and in protocol and application distributions.

If we break down the three propositions we made in the beginning of the results chapter, we find the following:

In terms of statistical properties, the heavy-tailed packet inter-arrival time distribution of the CATCH network reveals that the CATCH network has longer periods of network silence than the OUC network, and that it expresses more bursty behavior. This is due to the WAN nature of the CATCH network. The distribution declines much faster as the inter-arrival time increases, in the OUC network. The Poisson model fits well for the OUC1L trace, however we see strong indications that the Poisson model does not fit well for the CC1L trace. Further investigation of the self-similar properties of the two networks is necessary in order to say anything certain about their patterns, although we see signs of short-range dependence.

In terms of application usage, we have established that P2P applications are more widely used among private end-users than among the students at OUC. The OUC network had a more traditional distribution, with HTTP, HTTPS, and FTP, on the top 10 list. In contrast, the CC1L trace HTTP, SMTP, and FTP, were not even on the percent-wise top 10 list, measured both in packets and bytes. Additionally, new technologies such as Multicast have become widely used in the OUC network, and it is responsible for a substantial amount of the aggregated traffic.

If we consider proposition 3, we cannot conclude that P2P protocols have evolved from using fixed port numbers to using dynamic ones. On the contrary, our results revealed that P2P applications are still using well-known ports. Therefore, we disregard the hypothesis that payload inspections will reveal P2P traffic beyond port-based identification. However, as P2P protocols evolve fast, this is clearly a subject for further investigation.

Although the experiments were not repeated enough times to make the

results statistically significant, we have provided a snapshot insight into the properties of these networks. The results may or may not have looked different if we observed the networks for a longer period of time.

The total time available for this project was only 17 weeks, hence it was impossible to cover all aspects of network analysis. At an early stage of the project period, we realized that the study had to be limited to focus on a relatively small number of network properties. The field of network monitoring and analysis is an evolving research field, and researchers are finding new ways to analyze and represent network traffic characteristics by the day. We have chosen to focus on a combination of properties that we deemed interesting after reading a series of papers on the subject.

CoralReef is a comprehensive software suite, which offers several powerful features. However, during the progress of this project we have discovered that there exists several other applications that provide more or less the same functionality. Given more time, it could have been interesting to look at these applications as well, and compare them to the CoralReef suite. CoralReef is not perfect, and it is not yet ready for all network analysis tasks. We have one grave objection with the CoralReef suite, namely that we found it unnecessarily hard to compile and install. Subsequently, we wasted several days trying to find an operating system that the applications would compile on. We are unsure as to why we had such a hard time compiling the applications, but apparently CoralReef is picky about Perl and Libc versions.

The output form the applications is wide in scope, and offers a plethora of information. However we are often only interested in a small subset of that information. Subsequently, we had to develop a set of custom scripts to post-process the output from CoralReef. It would probably have been more effective, in terms of processing speed, to modify the applications themselves to include more options for narrowing down the scope of the output.

During these experiments we have only touched upon the *offline* capabilities of the CoralReef suite. We placed this limitation on the study for several reasons: First of all, we did not have access to the CATCH network for a long period of time, and we did not have permission to install custom software on the capturing hardware; secondly, we had limited time to conduct the experiments, and it was crucial to get started with the actual data as soon as possible.

We have developed a set of utilities that is intended to supplement existing software and ease the day-to-day operations for system and network administrators working with analysis and debugging of networks. The post-processing scripts are more or less straight-forward, and they work as intended. However, tools such as the `inspect.pl` script could preferably have been expanded with functionality that also included per protocol, and aggregated volume in bytes, of P2P traffic. The script could also have been expanded with connection tracking, enabling the network administrator to do flow analysis of the P2P traffic. We chose to limit the functionality of the script to offline analysis of pre-captured traffic traces only. However, the procedure for opening a trace file and a live stream is more or less the same, so enabling the script to also read from live streams is trivial. We could also have modified the output to include more information, or perhaps present the data in a more

accessible format, such as on a visually pleasing web page.

Additionally, it would have been interesting to reverse-engineer more P2P protocols, in order to find out their exact behavioral pattern, and to see if other criteria can be used to identify the traffic. This is analogous to the way spam filters are moving from using simple pattern-matching of the contents of the email, to using statistical properties of the email, e.g., utilizing Bayesian classification, to separate spam from legitimate email [AHTea04].

During the progress of this thesis, we have realized that there exists several intriguing possibilities in combing existing tools in a creative way. Given more time, it would have been possible to develop an online monitoring and analysis platform that combined features offered by CoralReef, with data-storage and visualization tools such as RRDTool and GNUplot. RRDTool can store long-term statistics in round-robin databases, whereas GNUplot can be used for live or semi-live visualization of statistical data. This information could have been presented in a visually pleasing web interface, and could, in the author's opinion, exceed monitoring and analysis platforms such as Ntop — with respect to both functionality and usability. CoralReef sports a web front-end for online analysis, through the `t2` applications, however these offer less information than Ntop.

As with all projects of this nature, the learning period never ends, and subsequently the focus has shifted several times during the experimental phase, as well as during the writing of this thesis. Although we started out with a much wider scope, we quickly acknowledged that the limited time did not allow us to cover *everything*. As a result of this, we narrowed the scope down sufficiently to be able to dig deep enough into the material for the results to be of any value. Network analysis is time-consuming, both because we are working on huge data files, that takes time to process — and also because it requires interpretation of the results. In order to be able to learn anything from the results it is crucial that one has a clear understanding of what one is measuring, and what the numbers *mean*. Moreover, it is important to be aware of constraints and limitations, and to consider these in the interpretation of the results.

Appendix A

CoralReef Extras

A.1 CoralReef Applications

Application	Functionality
`crl_info`	- Information about selected file, type of hardware, iomode, interface and more.
`crl_time`	- Outputs one line per packet. Eache line contains ifnumber, pkt number, timestap, time difference between pkts and more.
`crl_guess`	- Attempts to guess file format, interface type, link layer encapsulation protcol and more.
`crl_encode`	- Encodes IP addresses to protect privacy and removes any pkt payload. This tool was used for encoding the IP addresses from the CATCH data-set.
`crl_rate`	- Collects IP-level stats, IP lengths, IP-packet counts, non-IP packet counts and IPv6 packet counts per interface/subinterface. It outputs summaries every interval seconds (default: 60).
`crl_hist`	- Outputs a report on packet and byt counts by IP length and protocol, port summary matrics for TCP and UDP, packet length histograms, top 10 source and destination ports, and more.
`crl_flow`	- Creates summaries of traffic flow for port-processing by t2_report and other scripts.After each interval, crl_flow outputs tables of flows which expired during the interval and tables of flows which are still active.
`t2_rate`	- Similar to crl_rate, outputs information about every interface/subinterface, with the addition of a tuple (unique connection) rate, and works on crl_traffic2 output instead of on CoralReef devices or tracefiles.
`crl_flowscnt`	- Groups packets into bins in order to measure flows. In addition to a final report, it will report partial results every 100000 packets. Basis of a framework for comparing different definitions of flows and analyzing which definition gives more useful data.

A.2 CoralReef Application Sample Output

A.2.1 crl_info

```
testfile.crl:
 type:     file
 version:  9 (0x80000009)
 comment:  OC12 ATM RFC1483 point
 encoding: cidr  version 10
 iomode:   unspecified
 interface 0
   timestamps:  little endian, not normalized
   hardware:  point  version 0
   software:  Coral version 9
   physical:  ATM
   data link: ATM_RFC1483
   captured:  1999-10-15 15:06:40.000000 UTC (940000000.000000)
   tz offset: -08:00
   iomode:    first=48,rx
   bandwidth: OC12c (599040 Kbps)
```

A.2.2 crl_time

```
if   cell   native timestamp        seconds         difference  comment
 0      0: 00000189 00001763  937347057.030465400
 0      1: 00000189 0000187b  937347057.030476600  0.000011200
 0      2: 00000189 00001a4b  937347057.030495160  0.000018560
 0      3: 00000189 00001b05  937347057.030502600  0.000007440
 0      4: 00000189 00001e7f  937347057.030538200  0.000035600
 .
 .
 .
 0    151: 00000189 0000fdba  937347057.032824080  0.000007440
 0    152: 00000189 0000fed7  937347057.032835480  0.000011400
 0    153: 0000018a 0000038e  937347057.032883760  0.000048280  high stamp +1
 .
 .
 .
 0    198: 0000018a 00004462  937347057.033547600  0.000007440
 0    199: 0000018a 000045fd  937347057.033564040  0.000016440
 0    200: 00000000 000003f2  937347056.000040400 1.033523640
 0    201: 00000000 00000981  937347056.000097320  0.000056920
 0    202: 00000000 00000c68  937347056.000127040  0.000029720
 0    203: 00000000 00000eec  937347056.000152800  0.000025760
 .
 .
 .
 0  13835: 000005b0 0000f74a  937347059.819348880  0.000142640
 0  13836: 00000000 0000064e  937347056.000064560 3.819284320
 0  13837: 00000000 0000077a  937347056.000076560  0.000012000
 0  13838: 00000000 00000872  937347056.000086480  0.000009920
 .
 .
 .
# 2 anomalous records found.
```

A.2.3 crl_rate

```
# time 937345564.000001 (11.723567s), packets lost: 0
# if v4pkts v4bytes v6pkts v6byte non-IP v4pkts/s v4bits/s v6pkts/s v6bits/s
```

```
0[1:257]         0          0        0      0      1    0.00     0.00     0.00
0.00
0[1:4350]    19460    6914362        0      0      0    1.66k    4.72M    0.00
0.00
0[1:258]         0          0        0      0      1    0.00     0.00     0.00
0.00
0[1:4351]    27302    7199757        0      0      0    2.33k    4.91M    0.00
0.00
0[1:4346]    29923   15786019        0      0      0    2.55k   10.77M    0.00
0.00
0[1:4096]     2343    1802366        0      0      0  199.85     1.23M    0.00
0.00
0[1:4097]        1         40        0      0      0    0.09    27.30     0.00
0.00
0   TOTAL    79029   31702544        0      0      2    6.74k   21.63M    0.00
0.00
```

A.2.4 crl_hist

```
#Trace File: testfile.crl
#$Id: crl_hist.pl,v 2.15 2002/02/28 21:26:14 dmoore Exp $
#$Author: dmoore $
#$Name:  $
#$Revision: 2.15 $
#Report Library
#$Id: CRL_report.pm,v 2.10 2001/04/11 17:06:08 kkeys Exp $
#$Author: kkeys $
#$Name:  $
#$Revision: 2.10 $
#Interesting TCP Ports: 20 25 80 110 119 139 443 554 1051 1062 1085 1144 1755
3128 5501 6000 6688 6697 6699 7070 8000 8080
#Interesting UDP Ports: 53 137 2049 3130 6112 6770 6970 6971 6972 7070 7648
7777 9000 9001 9005 27001 27005 27015 27500 27901
27910 27960 37370
#Starts at 937345564.0000016689, Ends at 937345575.7235686779,
Duration 11.7235670090
#
937345575.7235686779    # end_time      = Latest timestamp
0       # fragcount     = Fragments of IP Datagrams
63418   # nofragcount   = IP packets with DF set
0       # nonip = Non-IP packets
0       # opcount       = IP Packets with options
937345564.0000016689    # start_time    = Earliest timestamp
0       # tcpfragcount  = Fragments of TCP Datagrams
31702544        # total_bytes   = Total IP Bytes
79029   # total_packets = Total IP Packets
11.7235670090   # total_time    = Duration of trace(s)
0       # udpfragcount  = Fragments of UDP Datagrams
#
#Packet and byte counts by IP length
#Size   Num     CumPkts pct.    CumByte pct.
28      52      52      0.06579863      1456    0.00459269
32      45      97      0.12273975      2896    0.00913491
36      24      121     0.15310835      3760    0.01186025
37      7       128     0.16196586      4019    0.01267722
38      36      164     0.20751876      5387    0.01699233
39      196     360     0.45552898      13031   0.04110396
40      26783   27143   34.34561996     1084351 3.42039112
41      214     27357   34.61640664     1093125 3.44806713
    .
    .
    .
540     8       54389   68.82157183     4492596 14.17108987
541     6       54395   68.82916398     4495842 14.18132879
```

```
542     9       54404   68.84055220     4500720 14.19671557
543     4       54408   68.84561364     4502892 14.20356675
544     33      54441   68.88737046     4520844 14.26019313
545     7       54448   68.89622797     4524659 14.27222686
546     4       54452   68.90128940     4526843 14.27911590
    .
    .
    .
1497    6       68618   86.82635488     16086063        50.74060618
1498    6       68624   86.83394703     16095051        50.76895722
1499    7       68631   86.84280454     16105544        50.80205551
1500    10398   79029   100.00000000    31702544        100.00000000
#
#Traffic breakdown by protocol:
#proto  pcount  pct.    bytes   pct.    avg. size
6       70224   88.85852029     30603213        96.53235715     435
17      6756    8.54876058      899399  2.83699314      133
1       1755    2.22070379      105028  0.33129203      59
50      186     0.23535664      60744   0.19160607      326
94      45      0.05694112      25184   0.07943842      559
2       10      0.01265358      4849    0.01529530      484
4       49      0.06200256      3832    0.01208736      78
47      4       0.00506143      295     0.00093052      73
#
#TCP Traffic matrix:
#src    dst     packets pct.    bytes   pct     avg. size
80      0       26276   37.41740715     18246167        59.62173645     694
0       25      7832    11.15288221     4257019 13.91036621     543
443     0       2327    3.31368193      2149294 7.02309918      923
0       0       5181    7.37781955      1974318 6.45134222      381
0       80      19254   27.41797676     1602202 5.23540453      83
20      0       700     0.99681021      509613  1.66522711      728
119     0       1054    1.50091137      312526  1.02121957      296
139     0       230     0.32752335      298589  0.97567860      1298
8000    0       168     0.23923445      212091  0.69303507      1262
0       119     225     0.32040328      179785  0.58747100      799
110     0       482     0.68637503      141464  0.46225212      293
25      0       1867    2.65863522      102136  0.33374273      54
0       20      570     0.81168831      90626   0.29613230      158
554     0       64      0.09113693      75897   0.24800337      1185
1755    0       85      0.12104124      71476   0.23355718      840
22      0       734     1.04522670      70676   0.23094307      96
    .
    .
    .
#
#UDP Traffic matrix:
#src    dst     packets pct.    bytes   pct     avg. size
0       7070    480     7.10479574      251055  27.91364011     523
27015   27005   609     9.01420959      146224  16.25796782     240
53      0       528     7.81527531      107169  11.91562366     202
0       0       1045    15.46773239     94942   10.55616028     90
53      53      742     10.98283008     72567   8.06838789      97
27500   27001   445     6.58673771      59586   6.62509076      133
27901   0       707     10.46477205     40299   4.48065875      57
27005   27015   703     10.40556542     38317   4.26028937      54
27901   27910   610     9.02901125      37891   4.21292441      62
6112    6112    362     5.35820012      18423   2.04836785      50
0       53      201     2.97513321      13081   1.45441567      65
    .
    .
    .
#
#Fragment breakdown by protocol:
#proto  pcount  pct.    bytes   pct.    avg. size
#
```

```
#Packet and byte counts from 554/tcp by IP length
#Size   Num     CumPkts pct.    CumByte pct.
40      1       1       1.56250000      40      0.05270301
116     1       2       3.12500000      156     0.20554172
271     1       3       4.68750000      427     0.56260458
349     1       4       6.25000000      776     1.02243830
600     1       5       7.81250000      1376    1.81298339
649     1       6       9.37500000      2025    2.66808965
653     1       7       10.93750000     2678    3.52846621
654     11      18      28.12500000     9872    13.00710173
663     3       21      32.81250000     11861   15.62775867
1268    2       23      35.93750000     14397   18.96912921
1500    41      64      100.00000000    75897   100.00000000
#
#Packet and byte counts from 80/tcp by IP length
#Size   Num     CumPkts pct.    CumByte pct.
40      4998    4998    19.02115999     199920  1.09568218
41      36      5034    19.15816715     201396  1.10377155
42      2       5036    19.16577866     201480  1.10423192
43      5       5041    19.18480743     201695  1.10541025
44      2069    7110    27.05891308     292731  1.60434244
    .
    .
    .
1496    600     18996   72.29410869     7326186 40.15191793
1497    1       18997   72.29791445     7327683 40.16012240
1498    6       19003   72.32074897     7336671 40.20938206
1499    4       19007   72.33597199     7342667 40.24224375
1500    7269    26276   100.00000000    18246167        100.00000000
#
#Packet and byte counts from 8000/tcp by IP length
#Size   Num     CumPkts pct.    CumByte pct.
41      10      10      5.95238095      410     0.19331325
46      1       11      6.54761905      456     0.21500205
77      1       12      7.14285714      533     0.25130722
    .
    .
    .
1430    1       52      30.95238095     38123   17.97483156
1468    1       53      31.54761905     39591   18.66698728
1500    115     168     100.00000000    212091  100.00000000

    .
    .
    .
    .
    .

#
#ICMP traffic breakdown
#Type   Code    packets pct.    bytes   pct     avg. size
5       0       1218    69.40170940     68208   64.94268195     56
8       0       286     16.29629630     19822   18.87306242     69
3       3       75      4.27350427      6406    6.09932589      85
11      0       53      3.01994302      2980    2.83733861      56
3       13      38      2.16524217      2128    2.02612637      56
69      0       25      1.42450142      1972    1.87759455      78
0       0       29      1.65242165      1776    1.69097764      61
3       1       31      1.76638177      1736    1.65289256      56
#
#IGMP traffic breakdown
#Type   Code    packets pct.    bytes   pct     avg. size
19      2       9       90.00000000     4813    99.25757888     534
19      1       1       10.00000000     36      0.74242112      36
#
#TCP packet and byte counts by source port (top 10)
```

```
#Port   packets pct.    bytes   pct     avg. size
80      26276   37.41740715     18246167        59.62173645     694
443     2327    3.31368193      2149294 7.02309918      923
1069    520     0.74048758      566063  1.84968487      1088
20      700     0.99681021      509613  1.66522711      728
119     1054    1.50091137      312526  1.02121957      296
139     230     0.32752335      298589  0.97567860      1298
8000    168     0.23923445      212091  0.69303507      1262
2858    116     0.16518569      169704  0.55453001      1462
2995    132     0.18796992      168962  0.55210543      1280
1999    134     0.19081795      165993  0.54240383      1238
#
#TCP packet and byte counts by destination port (top 10)
#Port   packets pct.    bytes   pct     avg. size
25      7853    11.18278651     4268116 13.94662711     543
80      19254   27.41797676     1602202 5.23540453      83
1087    1570    2.23570289      1540886 5.03504648      981
38776   1055    1.50233538      989135  3.23212795      937
1215    531     0.75615174      577984  1.88863829      1088
1142    287     0.40869219      346306  1.13160014      1206
1512    218     0.31043518      297705  0.97279001      1365
57596   198     0.28195489      292166  0.95469061      1475
1294    192     0.27341080      216961  0.70894844      1130
50109   133     0.18939394      190164  0.62138573      1429
#
#UDP packet and byte counts by source port (top 10)
#Port   packets pct.    bytes   pct     avg. size
53      1270    18.79810539     179736  19.98401155     141
27015   609     9.01420959      146224  16.25796782     240
2000    176     2.60509177      113459  12.61497956     644
27901   1317    19.49378330     78190   8.69358316      59
27500   445     6.58673771      59586   6.62509076      133
1892    85      1.25814091      50760   5.64376878      597
27005   703     10.40556542     38317   4.26028937      54
21470   52      0.76968620      33275   3.69969279      639
47774   84      1.24333925      26812   2.98110182      319
16519   84      1.24333925      26628   2.96064372      317
#
#UDP packet and byte counts by destination port (top 10)
#Port   packets pct.    bytes   pct     avg. size
6970    335     4.95855536      209109  23.24985907     624
27005   609     9.01420959      146224  16.25796782     240
53      943     13.95796329     85648   9.52280356      90
27001   445     6.58673771      59586   6.62509076      133
27930   707     10.46477205     40299   4.48065875      57
27015   712     10.53878034     38664   4.29887069      54
27910   610     9.02901125      37891   4.21292441      62
7029    84      1.24333925      26812   2.98110182      319
3568    67      0.99171107      20538   2.28352489      306
2233    132     1.95381883      20400   2.26818131      154
#
```

A.2.5 crl_flow

```
# crl_flow output version: 1.1 (text format)
# generated by:  crl_flow -A -Tf5 -Ci=10 testfile.crl

# begin trace interval: 937345564.000001
# trace interval duration: 10.000000 s
# Layer 2 PDUs dropped: 0
# Packets dropped: 0
# IP: 21.3840 Mbit/s
# Non-IP: 0.0000 pkts/s
# Table IDs: 0[1:4097], 0[1:4096], 0[1:4351], 0[1:4350], 0[1:258], 0[1:4346]
```

```
11
12 # ID: 0[1:4097]
13 # unknown_encaps: 0
14 #      ip_not_v4: 0
15 #           pkts: 1
16 #          bytes: 40
17 #          flows: 1
18 #          first: 937345569.195527825
19 #         latest: 937345569.195527825
20 # Table types: Tuple Table (active)
21
22 # ID: 0[1:4096]
23 # unknown_encaps: 0
24 #      ip_not_v4: 0
25 #           pkts: 2008
26 #          bytes: 1545956
27 #          flows: 185
28 #          first: 937345564.004452850
29 #         latest: 937345573.984448775
30 # Table types: Tuple Table (expired), Tuple Table (active)
31
32 .
33 . (etc)
34 .
35
36
37 # begin Tuple Table (active) ID: 0[1:4097]
38 #src dst proto ok sport dport pkts bytes flows first latest
39 198.32.200.43   198.32.200.31   6 1 23889 179 1 40 1 937345569.195527825
40 937345569.195527825
41 # end of text table
42
43 # begin Tuple Table (expired) ID: 0[1:4096]
44 #src dst proto ok sport dport pkts bytes flows first latest
45 216.65.39.234 212.4.196.34 6 1 80 34336 2 80 1 937345566.724943350
46 937345566.725142800
47 216.65.55.119 195.202.34.83 6 1 80 1801 3 197 1 937345564.201072575
48 937345565.233968925
49
50 . (etc)
51 .
52 # end of text table
```

A.2.6 t2_rate

```
1  # time 937345564.000001 (5.000000s)
2  # if[subif]     pkts      bytes    flows  entries   pkts/s   bits/s  flows/s
3  0[1:4346]      12707    6842517     1852     1852    2.54k   10.95M  370.40
4  0[1:4351]      11803    2975471     1599     1599    2.36k    4.76M  319.80
5  0[1:4096]        774     505411      116      116  154.80   808.66k   23.20
6  0[1:4350]       7399    2534397     1190     1190    1.48k    4.06M  238.00
7  0   TOTAL      32683   12857796     4757     4757    6.54k   20.57M  951.40
8
9  # time 937345569.000001 (5.000000s)
10 # if[subif]     pkts      bytes    flows  entries   pkts/s   bits/s  flows/s
11 0[1:258]           0          0        0        0    0.00     0.00     0.00
12 0[1:4346]      12633    6537015     1926     1926    2.53k   10.46M  385.20
13 0[1:4351]      11533    3061945     1641     1641    2.31k    4.90M  328.20
14 0[1:4096]       1234    1040545      129      129  246.80     1.66M   25.80
15 0[1:4097]          1         40        1        1    0.20    64.00     0.20
16 0[1:4350]       9035    3232644     1390     1390    1.81k    5.17M  278.00
17 0   TOTAL      34436   13872189     5087     5087    6.89k   22.20M    1.02k
18
19 # time 937345574.000001 (1.723567s)
```

```
# if[subif]     pkts      bytes    flows  entries   pkts/s   bits/s  flows/s
0[1:4346]       4583    2406487     1037     1037    2.66k   11.17M  601.66
0[1:4351]       3966    1162341      889      889    2.30k    5.40M  515.79
0[1:4096]        335     256410       60       60  194.36     1.19M   34.81
0[1:4350]       3026    1147321      720      720    1.76k    5.33M  417.74
0   TOTAL      11910    4972559     2706     2706    6.91k   23.08M    1.57k
```

Appendix B

Tables

B.1 Application breakdown from Sprint SJ-00, August 2000

Category	Packets (%)	Bytes (%)	Flows (%)
Web	71.12	83.49	63.55
File Sharing	3.34	2.44	1.8
FTP	0.75	1.01	0.3
Email	2.44	1.58	2.11
Streaming	5.25	4.34	2.44
DNS	1.18	0.21	5.92
Games	0.92	0.13	0.18
Other TCP	7.46	4.63	14.89
Other UDP	6.74	1.92	6.56
Not TCP/UDP	0.81	0.26	2.26

Table B.1: Application breakdown from Sprint SJ-00, August 2000

Appendix C

Scripts

C.1 Shell Scripts

C.1.1 inter-arrival.sh

```
#!/bin/sh

USAGE="Usage: $0 [input trace]"
CRLTIME="/usr/local/Coral/bin/crl_time"

# Check that two command line arguments are given.
if [ -z "$1" ]; then
   echo $USAGE
   exit
fi

TMP1=‘mktemp tmp.1.XXXXXX‘ || exit 1
TMP2=‘mktemp tmp.2.XXXXXX‘ || exit 1
TMP3=‘mktemp tmp.3.XXXXXX‘ || exit 1

echo "Created temp-files: ‘basename $TMP1‘, ‘basename $TMP2‘, \
‘basename $TMP3‘.."
echo "Fetching deltas..."
$CRLTIME $1|awk ’{print $6}’ > $TMP1

echo "Sorting deltas..."
sort -n -T /data/ $TMP1 > $TMP2

echo "Calculating uniques..."
uniq -c $TMP2 > $TMP3

echo "Swapping X and Y axis..."
cat $TMP3|awk ’{ print $2 " " $1 }’ > inter-packet-$1.txt

echo "Cleaning up..."
rm -f $TMP1
rm -f $TMP2
rm -f $TMP3
```

C.1.2 rate.sh

```
#!/bin/sh

USAGE="Usage: $0 [input trace]"
```

```
CRLRATE="/usr/local/Coral/bin/crl_rate"

# Check that command line arguments are given.
if [ -z "$1" ]; then
   echo $USAGE
   exit
fi

echo "Fetching rates (bytes/min)..."
$CRLRATE -s $1|awk '{print $8}' > rate.txt
```

C.2 Gnuplot Scripts

C.2.1 psd_packets.gnuplot

```
set terminal postscript eps
set title "Cumulate percentage of packets against packet size"
set autoscale
set xlabel "Packet size"
set ylabel "Cumulative percentage of packets"
set output "psd_packets.eps"
plot "catch_psd_ppct.txt" title "CC1L" with lines, "ouc_psd_ppct.txt" \
title "OUC
```

C.2.2 rate-ouc-cc-pkt.gnuplot

```
set terminal postscript eps
set title "Data rate CC/OUC total (packets)"
#set autoscale
set xrange [0:2800]
set yrange [0:7000]
set xtics nomirror rotate by -45("13:00"0,"19:00"360,"01:00"720, \
"07:00"1080,"13:00"1440,"19:00"1800,"01:00"2160,"07:00"2520,\
"13:00"2880)
set xlabel "Time of the day"
set ylabel "Packets/s"
set output "rate-ouc-cc-pkt.eps"
plot "cc-pkt-rate.txt" title "CC1L" with lines, "ouc-pkt-rate.txt" title \
 "OUC1L" with lines
```

C.3 Perl Scripts

C.3.1 self-sim.pl

```
: # *-*-perl-*-*
eval 'exec perl -w -S  $0 ${1+"$@"}'
  if 0;    # if running under some shell

# Check that the correct number of command line arguments are given.

use POSIX;
die "Usage: $0 [input-file]\n" if @ARGV == 0;

# Pop the last argument
my $infile = pop(@ARGV);

```

```
$length = `wc -l $infile|awk '{print \$1}'`;
chomp($length);

#print "$length length\n";

for ( $incr = 10 ; $incr <= 10000 ; $incr *= 2 ) {
    open( INFILE, "<$infile" ) or die "Cannot read file $infile; $!\n";

    #print "i $incr\n\n";
    my $Gtotal = 0;
    my $antall = 0;

    # Outer for-loop.
    for ( $k = 0 ; $k < $length ; $k = $k + $incr ) {

        #print "#### Iteration: $k ####\n";

        # Declare some variables.
        my $highest = 0;
        my $lowest  = $highest;
        my $start   = 0;             # $k;
        my $end     = int($incr);    # $k + $incr;
        my $total   = 0;
        my $mean    = 0;
        my $var     = 0;
        my $varsum  = 0;
        my @line    = [];
        my @w       = [];

        # Open infile for reading.

        $num = 0;
        for ( $i = $start ; $i < $end ; $i++ ) {
            $line[$i] = <INFILE>;
            if ( defined $line[$i] ) {
                $num++;
                chomp( $line[$i] );
                $total += $line[$i];
                $var   += $line[$i] * $line[$i];

                #print "$i $line[$i]  \n";
                $w[$i] = 0;
            }
            else {

                # print "End of file\n\n\n";
                last;
            }
        }

        #print "$end ($num - $start)\n";
        # Calculating mean value.
        $mean = $total / ( 1.0 * $num - $start );
        $varsum = ( $var / ( 1.0 * $num - $start ) ) - ( $mean * $mean );
        if($varsum < 0)
        {
            $varsum = -$varsum;
        }
        $sigma  = sqrt($varsum);

        for ( $i = $start ; $i < $num ; $i++ ) {
            $w[$i] += ( $line[$i] - $mean );
            if ( $w[$i] > $highest ) { $highest = $w[$i]; }
            if ( $w[$i] < $lowest )  { $lowest  = $w[$i]; }
        }

```

```
79          #print "Mean: $mean ";
80          #print "Variance: $var ";
81          #print "Std. dev: $sigma\n";
82
83          # Closing INFILE.
84          $rnsn = ( $highest - $lowest ) / $sigma;
85
86          #       print "\t$highest $lowest P(n) / S(n) = $rnsn\n";
87          #       printf("k: %d %4.2f\n",$k,$rnsn);
88          $Gtotal += $rnsn;
89          $antall++;
90      }
91
92      $Gtotal = $Gtotal / ( 1.0 * $antall );
93      $ilog   = log($incr);
94      $Glog   = log($Gtotal);
95
96      printf( "%6.5f %6.5f\n", $ilog, $Glog );
97      close(INFILE);
98  }
```

C.3.2 normalizer.pl

```
1
2   : # *-*-perl-*-*
3     eval 'exec perl -w -S  $0 ${1+"$@"}'
4           if 0;  # if running under some shell
5
6   # Set to "1" if you want the script
7   # to print the value regardless of
8   # the input format.
9   $force = 0;
10
11  while(<STDIN>) {
12
13          # Reset variable each round.
14          my $normal = 0;
15
16          # If in Mb/s.
17          if($_ =~ /([0-9]+\.[0-9]+)M/) {
18                  print "$1\n";
19          }
20
21          # if in kb/s.
22          elsif($_ =~ /([0-9]+\.[0-9]+)k/) {
23                  $normal = $1/1000.0;
24                  print "$normal\n";
25          }
26
27          # if in b/s.
28          elsif($_ =~ /([0-9]+\.[0-9]+)/) {
29                  $normal = $1/1000000.0;
30                  print "$normal\n";
31          }
32
33          # else die or print line.
34          else {
35                  die "Parsing failed. The script halted on \
36                  the following line:\n\t $_\n" \
37                  unless($force == 1);
38                  print "$_";
39          }
40  }
```

C.3.3 inspect.pl

```
: # *-*-perl-*-*
  eval 'exec perl -S  $0 ${1+"$@"}'
        if 0;  # if running under some shell

use Getopt::Std;
use Net::Pcap;
use NetPacket::Ethernet;
use NetPacket::IP;
use NetPacket::TCP;
use NetPacket::UDP;

$| = 1; # unbuffer STDIO

my $usage = "USAGE: inspect.pl -r <dump-file> [-o <summary-file>] -p | \
-h (verbose help)\n";
my $usagelong = <<USGLONG;
USAGE: inspect.pl -r <dump-file> [-o <summary-file>]|-h (verbose help)
OPTIONS:
    -r <dump-file> Input-file in pcap format.
    -o <summary-file> Output-file for report.
    -p Do packet inspection in addition to port identification.
    -h Print this message.
USGLONG

my $opt_string = 'hr:o:p';

getopts( "$opt_string", \my \\%opt );

if ($opt{h}) {
    print $usagelong and exit;
}

if (!$opt{r}) {
    print $usage and exit;
}

my $pinsp = "yes" if $opt{p};

my $inputfile   = $opt{r};
my $outputfile  = $opt{o};

if ($outputfile) {
    open(OUTFILE,">$outputfile") ||
        die "Failed to open outputfile: $outputfile\n";
}

# The largest P2P networks.
my $kazaa           = 0;
my $edonkey         = 0;
my $winmx           = 0;
my $bittorrent      = 0;
my $gnutella        = 0;
my $directconnect   = 0;

# Some counters.
my $identc      = 0;
my $nontcpudp   = 0;
my $count       = 0;

# Strings to look for during packet inspection.
my @knownstrings    = ("GIVE", "GNUT", "GO!!", "\$MyN",
                       "\$Dir", "\$SR");

# Known P2P ports.
```

```
65  my @kazaaports          = (1214);
66  my @edonkeyports        = (4661, 4662, 4663, 4664,
67                             4665, 4666, 4667, 4668,
68                             4669, 4670, 4671, 4672);
69  my @winmxports          = (6257, 6699);
70  my @bittorrentports     = (6881, 6882, 6883, 6884,
71                             6885, 6887, 6888, 6889);
72  my @gnutellaports       = (6346);
73  my @directconnectports  = (411, 412);
74
75  my $object;
76
77  # Initializing packet object.
78  $object = Net::Pcap::open_offline($inputfile, \$err);
79
80  # Validating object.
81  unless (defined $object) {
82      die "Something went wrong when reading capture file: $err\n";
83  }
84
85  # Running loop with callback function.
86  Net::Pcap::loop($object, -1, \&process_pkt, '') ||
87      print "Reached EOF: $inputfile.\n";
88
89  # Closing pcap stream.
90  Net::Pcap::close($object);
91
92  # Calculating percentages.
93  my $kazaapct            = sprintf ("\%.2f", ($kazaa / $count) * 100.0);
94  my $edonkeypct          = sprintf ("\%.2f", ($edonkey / $count) * 100.0);
95  my $winmxpct            = sprintf ("\%.2f", ($winmx / $count) * 100.0);
96  my $bittorrentpct       = sprintf ("\%.2f", ($winmx / $count) * 100.0);
97  my $gnutellapct         = sprintf ("\%.2f", ($gnutella / $count) * 100.0);
98  my $directconnectpct    = sprintf ("\%.2f", ($directconnect / $count) * 100.0);
99  my $totalpct            = $kazaapct + $edonkeypct + $winmxpct + $bittorrentpct
100                           + $gnutellapct + $directconnectpct;
101
102 # Printing summary.
103 @summary = <<BREAKDOWN;
104
105                 General statistics
106 .-#-Total--------#-Identified--pct--#-Non-TCP-------.
107 | $count\t\t $identc\t\t $totalpct $nontcpudp\t |
108 '-------------------------------------------------'
109
110         Protocol breakdown by P2P network
111 .-Network---------------#-Packets------Cum-\%----.
112 | Kazaa/Fasttrack:\t $kazaa\t\t $kazaapct\t |
113 | Edonkey/clones:\t $edonkey\t\t $edonkeypct\t |
114 | WinMX/Napster:\t $winmx\t\t $winmxpct\t |
115 | Bittorrent:\t\t $bittorrent\t\t $bittorrentpct\t |
116 | Gnutella:\t\t $gnutella\t\t $gnutellapct\t |
117 | DirectConnect\t $directconnect\t\t $directconnectpct\t |
118 |-----------------------------.-----------------
119 | Total:\t\t $count\t\t 100\t |
120 '-----------------------------------------------'
121
122 BREAKDOWN
123
124 print @summary;
125
126 if($opt{o}) {
127     print OUTFILE @summary;
128     close(OUTFILE);
129 }
130
```

```
131 # Callback function.
132 sub process_pkt {
133     # Total packets.
134     $count++;
135     my $packet = $_[2];
136     # Strip packet.
137     my $ether_data = NetPacket::Ethernet::strip($packet);
138     # Strip Ethernet frame.
139     my $ip = NetPacket::IP->decode($ether_data);
140 
141     # Is TCP?
142     if ( $ip->{'proto'} == 6 ) {
143         # Strip IP.
144         my $tcp = NetPacket::TCP->decode($ip->{data});
145         my $dest_port = $tcp->{dest_port};
146 
147         # Is it Kazaa?
148         if (grep {$tcp->{dest_port} == $_} (@kazaaports)) {
149             $kazaa++; $identc++;
150         }
151         # Is it Edonkey?
152         elsif (grep {$tcp->{dest_port} == $_} (@edonkeyports)) {
153             $edonkey++; $identc++;
154         }
155 
156         # Is it WinMX?
157         elsif (grep {$tcp->{dest_port} == $_} (@winmxports)) {
158             $winmx++; $identc++;
159         }
160 
161         # Is it Bittorrent?
162         elsif (grep {$tcp->{dest_port} == $_} (@bittorrentports)) {
163             $bittorrent++; $identc++;
164         }
165 
166         # Is it Gnutella?
167         elsif (grep {$tcp->{dest_port} == $_} (@gnutellaports)) {
168             $gnutella++; $identc++;
169         }
170 
171         # Is it Direct Connect?
172         elsif (grep {$tcp->{dest_port} == $_} (@directconnectports)) {
173             $directconnect++; $identc++;
174         }
175 
176         # Not known port.
177         else {
178             # Is -p option set, if so inspect.
179             if ($pinsp eq "yes") {
180                 my $dest_port = $tcp->{dest_port};
181                 my $string = $tcp->{data};
182                 $string =~ s/\r\n/\n/g;
183                 chomp ($string);
184 
185                 if (grep {$tcp->{data} =~ /$_/} (@knownstrings)) {
186                     print "Recognized string on $dest_port/TCP: $string\n";
187                     $identc++;
188                 }
189 
190                 else {
191                     #print "No luck\n";
192                 }
193             }
194         }
195 
196     }
```

```
197     elsif ( $ip->{'proto'} == 17 ) {
198         # Strip IP.
199         my $udp = NetPacket::UDP->decode($ip->{data});
200         my $dest_port = $udp->{dest_port};
201 
202         # Is it Kazaa?
203         if (grep {$udp->{dest_port} == $_} (@kazaaports)) {
204             $kazaa++; $identc++;
205         }
206 
207         # Is it Edonkey?
208         elsif (grep {$udp->{dest_port} == $_} (@edonkeyports)) {
209             $edonkey++; $identc++;
210         }
211 
212         # Is it Gnutella?
213         elsif (grep {$udp->{dest_port} == $_} (@gnutellaports)) {
214             $gnutella++; $identc++;
215         }
216         # Is it Direct Connect?
217         elsif (grep {$udp->{dest_port} == $_} (@directconnectports)) {
218             $directconnect++; $identc++;
219         }
220 
221         # Not known port.
222         else {
223             # Is -p option set, if so inspect.
224             if ($pinsp eq "yes") {
225                 my $dest_port = $udp->{dest_port};
226                 my $string = $udp->{data};
227                 $string =~ s/\r\n/\n/g;
228                 chomp($string);
229 
230                 if (grep {$udp->{data} =~ /$_/} (@knownstrings)) {
231                     print "Recognized string on $dest_port/UDP: $string\n";
232                     $identc++;
233                 }
234 
235                 else {
236                     #print "No luck\n";
237                 }
238             }
239         }
240     }
241 
242     else {
243         # NonTCP/UDP.
244         $nontcpudp++;
245     }
246 }
```

C.3.4 local-avg.pl

```
1  : # *-*-perl-*-*
2    eval 'exec perl -S  $0 ${1+"$@"}'
3            if 0;  # if running under some shell
4 
5  use Getopt::Std;
6 
7  $| = 1; # unbuffer STDIO
8 
9  # Sensible default.
10 my $offset = 0;
11 my $interval = 10;
```

```
12
13  # Usage.
14  my $usage = "USAGE: local-avg.pl -r <input-file> -o <output-file> \
15  [-i <interval>] [-f <offset>]\n";
16
17  # Opt string.
18  my $opt_string = 'r:i:o:f:';
19
20  # Put opts in hash.
21  getopts( "$opt_string", \my %opt );
22
23  # Check that both -r and -o are present.
24  unless($opt{r} and $opt{o}) {
25      print $usage and exit;
26  }
27
28  # Has user specified own interval?
29  if ($opt{i}) {
30      $interval = $opt{i};
31  }
32
33  # Has user specified own interval?
34  if ($opt{f}) {
35      $offset = $opt{f};
36  }
37
38
39  # Fetching in and out-files.
40  my $inputfile   = $opt{r};
41  my $outputfile  = $opt{o};
42
43  # Open file-handles.
44  open(INFILE,"<$inputfile")    or die "Cannot read file $inputfile; $!\n";
45  open(OUTFILE,">$outputfile")  or die "Cannot write to file $outputfile; $!\n";
46
47  # Put infile in array.
48  @lines = <INFILE>;
49
50  # Length of file.
51  $size = @lines;
52
53  # Header.
54  print OUTFILE "#Lines\tLoc avg $interval\tStd.dev.\n";
55
56  # Counter.
57  my $i = 0;
58
59  # Outer loop.
60  while ($i < $size) {
61      # Some variables.
62      my $j       = 0;
63      my $locsum  = 0;
64      my $var     = 0;
65      my $stddev  = 0;
66      my $sumofsquares = 0;
67
68      # Inner loop.
69      while ($j < $interval) {
70          # Local sum for inner loop.
71          $locsum         += $lines[$i];
72
73          # Sum of squares.
74          $sumofsquares   += ($lines[$i]*$lines[$i]);
75          $i++;
76          $j++;
77      }
```

```
78
79      # Mean value for inner loop.
80      $mean   = $locsum / $interval;
81
82      # Variance
83      #$var    = ($sumofsquares - (($locsum * $locsum) / $i)) / ($i - 1));
84      $var     = ($sumofsquares - (($locsum * $locsum) / $i)) / ($i - 1);
85      # Standard devation.
86      $stddev = sqrt($var);
87
88      # Setting position in between the interval.
89      $pos    = ($i - ($interval / 2)) + $offset;
90      print OUTFILE "$pos\t$mean\t\t$stddev\n";
91  }
92
93  # Closing file-handles.
94  close(INFILE);
95  close(OUTFILE);
```

C.3.5 utilization.pl

```
1   : # *-*-perl-*-*
2     eval 'exec perl -S  $0 ${1+"$@"}'
3             if 0;  # if running under some shell
4
5   use Getopt::Std;
6
7   # Sensible default.
8   my $bandwidth = 100; # 100Mb/s LAN
9
10  my $usage = "USAGE: utilization.pl -r <ratefile> -b <bandwidth> (Mb/s)\n";
11
12  my $opt_string = 'r:b:';
13  getopts( "$opt_string", \my %opt );
14  print $usage and exit unless $opt{r};
15
16  my $inputfile = $opt{r};
17  open(INFILE,"<$inputfile") or die "Failed to open file: $inputfile\n";
18
19  if (defined $opt{b}) {
20      my $bandwidth = $opt{b};
21  }
22
23  my @rates = <INFILE>;
24  my $length = @rates;
25
26  foreach $rate (@rates) {
27      $totalrate += $rate;
28  }
29
30  my $mean = sprintf ("%.2f", $totalrate / $length);
31
32  my $meanutil = sprintf ("%.2f", ($mean / $bandwidth) * 100.0);
33  print "Mean rate: $mean Mb/s, Mean utilization: $meanutil %\n";
```

Bibliography

[AHTea04] T Aspelund, IA Hassan, HW Thorkildssen, and W S Tam et al. *Network and system administration: research surveys*, volume 1 of *HiO-report ; 2004 nr 24*. Oslo University College, 2004.

[Alm00] KC Almeroth. Evolution of multicast: From the mbone to interdomain multicast to internet 2 deployment. *IEEE Network*, 2000.

[And80] DJ Andrews. A stochastic fault model 1. static case. *J. Geophys. Res*, 1980.

[BA99] VK Bhagavath and GA Alpharetta. Emerging high-speed xdsl access services: Architectures, issues, insights, and implications. *IEEE Communications Magazine*, 1999.

[BCea01] N Brownlee, K Claffy, and M Murray et al. Methodology for passive analysis of a university internet link. *Proc. of Workshop on Passive and Active Measurements PAM2001*, 2001.

[BE00] DN Blank-Edelman. *Perl for system administration*. O'Reilly, 2000.

[BS03] SA Baset and H Schulzrinne. An analysis of the skype peer-to-peer internet telephony protocol. *Arxiv preprint cs.NI/0412017*, 2003.

[Bur04] M Burgess. *Analytical Network and System Administration — Managing Human-Computer Systems*. J. Wiley & Sons, Chichester, 2004.

[BZ01] N Brownlee and N Zealand. Using netramet for production traffic measurement. *Intelligent Management Conference*, 2001.

[Cac89] R Caceres. Measurements of wide area internet traffic. *Report UCB/CSD 89/550, Computer Science Division, University of California, Berkeley, California*, 1989.

[CB97] ME Crovella and A Bestavros. Self-similarity in world wide web traffic: Evidence and possible causes. *IEEE/ACM Transactions on Networking*, 1997.

[CBP95] KC Claffy, Hans-Werner Braun, and George C. Polyzos. A parameterizable methodology for internet traffic flow profiling. *IEEE Journal of Selected Areas in Communications*, 1995.

[CDG00] J Cleary, S Donelly, and I Graham. Design principles for accurate passive measurement. *Proceedings of Passive and Active Measurement Workshop*, 2000.

[CM99] KC Claffy and S McCreary. Internet measurement and data analysis: passive and active measurement, 1999.

[Com04a] G Combs. *The Ethereal Manual Page*, 2004.

[Com04b] G Combs. Ethereal user's guide. *http://www.ethereal.com/docs/user-guide/*, 2004.

[Coo04] Cooperative Association for Internet Data Analysis (CAIDA). *Is P2P dying or just hiding?* Globecom 2004, November, December 2004.

[Cor00] Sprint Corporation. Sprint ipmon dms - packet trace analysis. *http://ipmon.sprint.com/packstat/packet.php?040206*, 2000.

[DC98] J Drobisz and KJ Christensen. Adaptive sampling methods to determine network traffic statistics including the hurst parameter. *LCN*, 1998.

[DCS+01] L Deri, R Carbone, S Suin, N SpA, and I Italy C Serra. Monitoring networks using ntop. *Proc. of IM*, 2001.

[DJea92] P Danzig, S Jamin, and R Caceres et al. An empirical workload model for driving wide-area tcp/ip network simulations. *Internetworking: Research and Experience*, 1992.

[DS00a] L Deri and S Suin. Effective traffic measurement using ntop. *IEEE Communications Magazine*, 2000.

[DS00b] L Deri and S Suin. Practical network security: experiences with ntop. *Computer Networks*, 2000.

[DSS+99] L Deri, S Suin, C Sera, L Pacinotti, and I Pisa. Ntop: Beyond ping and traceroute. *DSOM*, 1999.

[EPW95] A Erramilli, P Pruthi, and W Willinger. Self-similarity in high-speed network traffic measurements: Fact or artifact. *Proc. in 12th Nordic Teletraffic Seminar*, 1995.

[FHH02] Tony Field, Uli Harder, and Peter Harrison. Network traffic behaviour in switched ethernet systems. *10th IEEE International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunications Systems*, 2002.

[GB99] M Grossglauser and JC Bolot. On the relevance of long-range dependence in network traffic. *IEEE/ACM Transactions on Networking*, 1999.

[Gog00] H Gogl. *Measurement and Characterization of Traffic Streams in High-Speed Wide Area Networks*. PhD thesis, Institut für Informatik, Technische Universität München, 2000.

[Gro01] CAIDA Metrics Working Group. Network measurement faq. *http://www.caida.org/outreach/metricswg/faq.xml*, 2001.

[Gro04] The IETF IPPM Working Group. Ip performance metrics website. *http://www.ietf.org/html.charters/ippm-charter.html*, 2004.

[Hea00] C Headquarters. *Voice-Over IP Monitoring 4.6. 0 Best Practices Deployment Guide*. Cisco Systems, Inc., 2000.

[ip96] *IPng and the TCP/IP protocols: implementing the next generation Internet*. ohn Wiley & Sons, 1996.

[Jai92] R Jain. Myths about congestion management in high speed networks. In *Proceedings of the IFIP TC6 International Conference on Information Network and Data Communication, IV*, pages 55–70. North-Holland, 1992.

[JLM04] V Jacobson, C Leres, and S McCanne. *The Tcpdump Manual Page*. Lawrence Berkeley Laboratory, Berkeley, CA, 2004.

[JR86] R Jain and S Routhier. Packet trains: measurements and a new model for computer network traffic. *IEEE Journal on Selected Areas in Communications*, 1986.

[JW99] JL Jerkins and JL Wang. From network measurement collection to traffic performance modeling: challenges and lessons learned. *Journal of the Brazilian Computer Society*, 1999.

[KMea01] K Keys, D Moore, and R Koga et al. The architecture of coralreef: An internet traffic monitoring software suite. Technical report, Cooperative Association for Internet Data Analysis - CAIDA, San Diego Supercomputer Center, University of California, San Diego, 2001.

[KMFB04] T Karagiannis, M Molle, M Faloutsos, and A Broido. A nonstationary poisson view of internet traffic. *Proc. IEEE Infocom*, 2004.

[MB76] R Metcalfe and D Boggs. Ethernet: Distributed packet switching for local computer networks. *Communications of the ACM*, 1976.

[MC97] T Monk and KC Claffy. Internet data acquisition and analysis: Status and next steps. Technical report, University of California, San Diego/National Laboratory for Applied Network Research (NLANR), 1997.

[MHK+03] A Moore, J Hall, C Kreibich, E Harris, and I Pratt. Architecture of a network monitor. *Passive and Active Measurement Workshop 2003*, 2003.

[MJ98] G R Malan and F Jahanian. An extensible probe architecture for network protocol performance measurement. *SIGCOMM*, 1998.

[MKea01] D Moore, K Keys, and R Koga et al. The coralreef software suite as a tool for system and network administrators. Technical report, Cooperative Association for Internet Data Analysis - CAIDA, San Diego Supercomputer Center, University of California, San Diego, 2001.

[MLJ94] S McCanne, C Leres, and V Jacobson. *Libpcap manual*. Lawrence Berkeley National Labs, 1994.

[Mor03] M Morin. Managing p2p traffic on docsis networks. Technical report, Sandvine Incorporated, 2003.

[MTH90] Y Matsumoto, Y Takahashi, and T Hasegawa. The effects of packet size distributions on output and delayprocesses of csma/cd. *IEEE Transactions on Communications*, 1990.

[MVN97] S Molnar, A Vidacs, and AA Nilsson. Bottlenecks on the way towards fractal characterization of network traffic: Estimation and interpretation of the hurst parameter. *Proceedings of PMCCN*, 1997.

[ope99] *Open Sources: Voices from the Open Source Revolutio, The GNU Operating System and the Free Software Movement*. The Free Software Foundation, 1999.

[Ora01] Andy Oram, editor. *Peer-to-peer: Harnessing the Power of Disruptive Technologies*. O'Reilly, Sebastopol, California, 2001.

[Peu02] M Peuhkuri. Internet traffic measurements – aims, methodology, and discoveries. Master's thesis, Helsinki University of Technology, 2002.

[PF95] V Paxson and S Floyd. Wide area traffic: the failure of poisson modelling. *IEEE/ACM Transactions on networking*, 3(3):226, 1995.

[PKC97] K. Park, G. Kim, and M. E. Crovella. Effect of traffic self-similarity on network performance. In *Proc. SPIE Vol. 3231, p. 296-310, Performance and Control of Network Systems, Wai Sum Lai; Hisashi Kobayashi; Eds.*, pages 296–310, 1997.

[PNB03] FL Piccolo, G Neglia, and G Bianchi. The effect of heterogeneous link capacities in bittorrent-like file sharing systems. *IEEExplore*, 2003.

[RC04] ME Renda and J Callan. The robustness of content-based search in hierarchical peer to peer networks. *Proceedings of the Thirteenth ACM conference on Information*, 2004.

[Rot01] J Rothman. Which os is fastest for high-performance network applications. *SysAdmin Magazine*, 2001.

[Tea01] The CoralReef Team. Coralreef. *http://www.caida.org/tools/measurement/coralreef/*, 2001.

[Wei05] EW Weisstein. Self-similarity. *From MathWorld–A Wolfram Web Resource. http://mathworld.wolfram.com/Self-Similarity.html*, 2005.

[WPT96] W. Willinger, V. Paxson, and M.S. Taqqu. Self-similarity and heavy tails: structural modelling of network traffic. *in* A practical guide to heavy tails: statistical techniques and applications, pages 27–53, 1996.

[WS90] L Wall and R Schwarz. *Programming perl*. O'Reilly & Assoc., California, 1990.