pcapstitch: A Tool to Collect Singleton One-way Delay and Loss Measurements

A Thesis

Submitted to the Faculty

of

Drexel University

by

Daniel William DaCosta

in partial fulfillment of the

requirements for the degree

of

Masters of Science

May 2011

Dedications

For my parents; the source of my resilience, tenacity, passion, and pride.

Acknowledgments

I would like to thank Dr. Spiros Mancoridis for his time and encouragement. His positivity, organization, and efficiency are second to none. I would also like to thank my committee, Dr. Regli, Prof. Kain, and Dr. de Oliveira for their time.

Many, many things, including this thesis, would be impossible without Jess's love and support. I will pay this debt in kindness, always kindness.

Contents

List of Tables

List of Figures

Abstract

pcapstitch: A Tool to Collect Singleton One-way Delay and Loss Measurements
Daniel William DaCosta
Dr. Spiros Mancoridis

Network measurement is an established engineering principle[1]. However, there are still some aspects that lack general tools for measurement. A simple, general tool to measure one-way traffic characteristics passively does not exist. Currently, measuring one-way delay and loss requires instrumented applications, estimation, complex infrastructure setup, or active measurement techniques. This makes one-way delay and packet loss difficult to measure, yet these measurements directly indicate network influenced application utility degradation.

pcapstitch is a tool that collects singleton one-way delay and loss measurements. pcapstitch collects these measurements passively requiring no application instrumentation (*i.e.*, it can be used with general network traffic). pcapstitch associates semantically equivalent packets in trace files collected from multiple observation points. Packets must be associated in this way because packets can be modified in transit. The only other tool with this capability known to this author is OpenIMP[2] which is a comprehensive measurement suite. It can measure many different network characteristics both passively and actively. It relies on multiple independent software components, probes and a measurement controller. In contrast, pcapstitch has a simple setup and is designed solely to collect singleton one-way delay and loss measurements from trace files. Simplicity, few dependencies, and operation consistent with the UNIX philosophy are advantages of using pcapstitch.

Chapter 1: Introduction

Computer applications (herein after referred to as applications) are pervasive in modern life. They automate tasks that previously required human effort and expertise. Word processing applications (*i.e.*, Microsoft Word, OpenOffice Writer, and AbiWord) exemplify the potential of computer applications. Automated spell checking and grammar components improved editing efficiency.

Application utility characterizes an application's usefulness. An objective analysis of application utility can identify and aid in the diagnosis of application problems. Application utility measurement is specific to an application, category of applications, and/or application usage context. For example, one measure of utility for word processing applications might be the frequency of misspellings.

Computer networks (herein after referred to as networks) allow applications to communicate with each other remotely over some channel. Using networks, documents that can be created and edited more efficiently using word processing applications, can be sent across the globe in seconds using email applications. Email applications, or any other application that relies on networks, are referred to as network applications. However, networks are made up of imperfect channels. A perfect channel presents a one-to-one function between sent messages and received messages[3]. Networks are composed of imperfect channels making networks imperfect. Protocol stacks suffer performance degradation due to network imperfections. A protocol stack is an implementation (also referred to as an instance) of a set of reference protocols necessary for communication over a network. Network applications rely on protocol stacks. Thus, network application utility is dependent on the network characteristics and their effect on protocol stacks.

Network application utility is influenced by network characteristics through protocol stacks. Network characteristics describe a networks particular quality or lack thereof. A web browser (*i.e.*, Microsoft Explorer, Mozilla Firefox, and Google Chrome) is a type of network application that is strongly influenced by network characteristics. Network characteristics can largely influence browsing delay, which is one measure of web browser application utility[4]. Galletta *et. el.* define browsing

delay as the difference between the time of the hyperlink click and the time of the hyperlink content presentation [5]. Chat, voice, video, and bulk data transfer are types of network applications with sensitivities to network characteristics. Delay and loss degrade video quality. Congested network paths can cause excessive data transfer times. Therefore, measuring network characteristics is necessary to diagnosing application utility degradation.

Network measurement is an established engineering principle[1]. However, there are still some aspects that lack general tools for measurement. Current tools can be divided into two major categories, active and passive. Active tools introduce traffic into the network for measurement, passive tools measure organic network traffic. Passive measurement induces less bias and is preferred when possible. A simple, general tool to measure one-way traffic characteristics passively does not exist. Currently, measuring one-way delay and packet loss requires instrumented applications, estimation, complex infrastructure setup, or active measurement techniques. This makes one-way delay and packet loss difficult to measure, yet these measurements directly indicate network influenced application utility degradation.

pcapstich (/*p*•*cap*•*stitch*/, capitalization intended) is a simple, general tool to measure one-way traffic characteristics passively. pcapstitch *stitches* identical packets from network trace files collected simultaneously from multiple network observation points. The non-uniqueness of packets over time is mitigated through the *Packet Equivalency Functions* and *stitch horizon*. pcapstitch can measure one-way traffic characteristics accurately without application instrumentation. Consequently, pcapstitch can measure network characteristics responsible for application utility degradation.

This dissertation discusses the construction and usefulness of pcapstitch. In Chapter 2, an overview networks and the TCP/IP protocol suite is given. This chapter then goes on to discuss the current state of the art tools and why pcapstitch is a useful tool. Chapter 3 discusses, how to use pcapstitch, how pcapstitch is built, what issues may effect accuracy, and a exemplary use of pcapstitch. Chapter 4 concludes with how pcapstitch should be used and why it is useful.

Chapter 2: Background

A network is a set of objects that are interconnect by some medium. A social network is a network where the objects are people and the medium is some relationship between pairs of people. A family tree is an instance of a social network. Another type of network is a communication network, which facilitates the transfer of data between objects (*e.g*, personal computers, routers, switches, cell phones) using a medium (*e.g*, conductive material, fiber-optical material, electromagnetic frequency).

Graphs are commonly used to represent networks. A graph ($G = (V, E)$) is a set of vertices V and a set of binary relations or edges E[6]. In a network graph, objects are represented by $v \in V$ and medium interconnects are represented by $(v_1, v_2) \in E$ where $v_1, v_2 \in V$ Figure 2.1, shows a graph representation of a communication network where the vertices are computing devices and the edges are medium interconnects.

The Internet is an example of a relatively recent man-made communication network. This work is concerned with computer communication networks (hereinafter referred to as network(s)). Networks are mode of devices interconnected by a channel. A channel is a means for data transmission. More specifically, it is a system to transmit choices represented as symbols from one point to another[7]. Ethernet[8] is an example of a channel used to transmits bits between computers. In networks represented as graphs, channels are edges or sets of edges.

This master's dissertation describes pacpstitch, a tool to measure one-way network characteristics. pcapstitch associates the same packet recorded in different locations within a network. pcapstitch's name describes its operation; it stitches two or more packets recorded in multiple libpcap formatted files. This section contains the background material necessary to understand pcapstitch operation and construction. Section 2.1 describe basic structure and terminology of modern networks. Section 2.2 details the TCP/IP protocols suite, arguably the most popularly used protocol suite. Network traces are described in Section 2.3. Section 2.4 explains the importance of measuring delay and loss. Section 2.5 goes over community standards for measuring delay and loss. Section

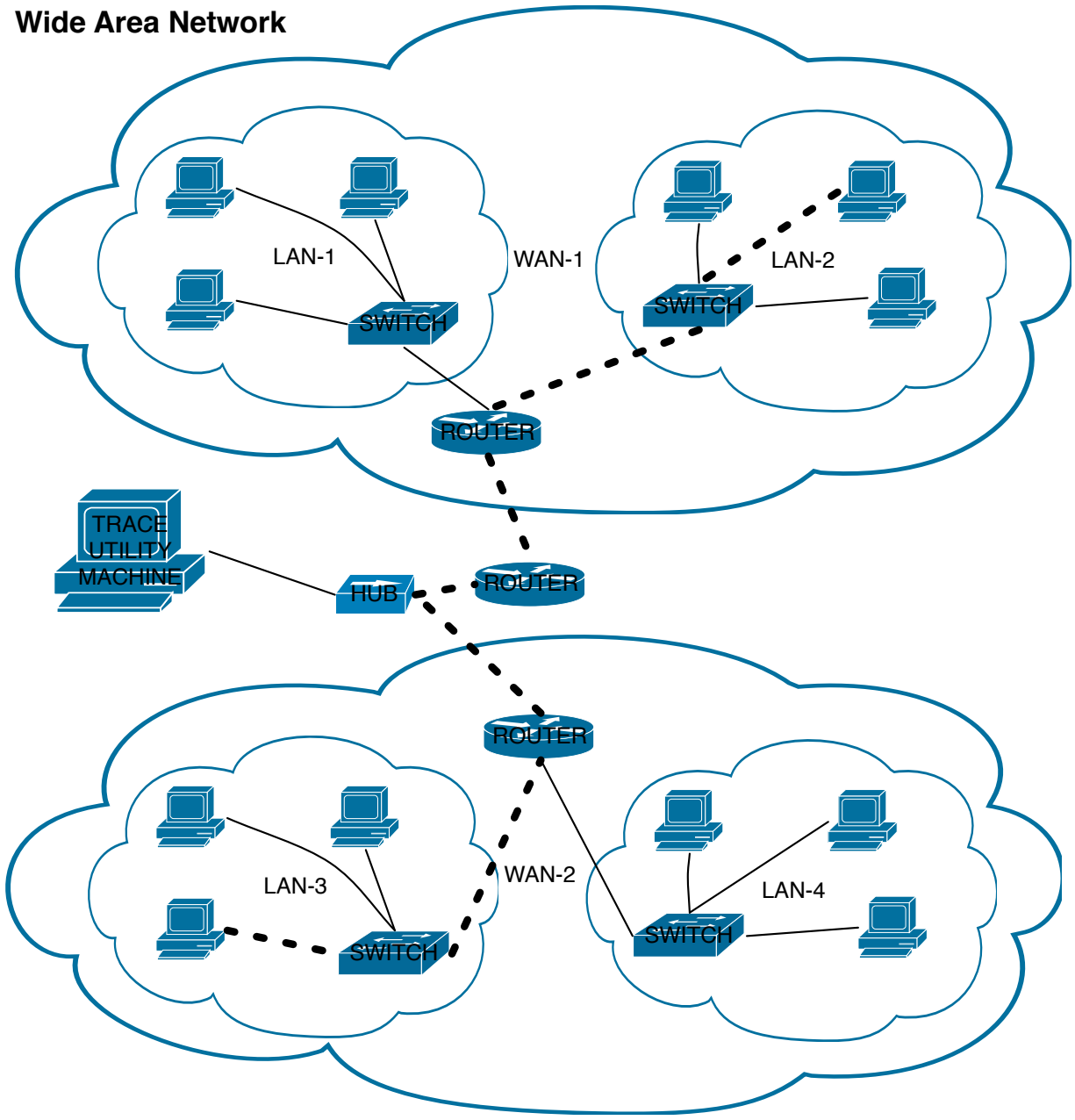

Figure 2.1: This diagram shows the basic network building blocks. The diagram is one WAN (composed of the entire figure) with two constituent WANs. Each constituent WAN contains two LANs. Switches allow devices on LANs to communicate. Routers allow inter-LAN communication. A WAN is made up of LANs,WANs, and devices. All edges are referred to as "links". The series of links that are dotted represent a "path" from a device in LAN-3 to LAN-2. A hub is on the path between two routers. Hubs duplicate traffic to all connected devices. Therefore, the trace utility machine will observe all traffic between these routers.

2.6 surveys the most popular network measurement tools for measuring delay and loss. Finally, the problem that pcapstitch is addressing and solves is succinctly described in Section 2.7.

2.1 Overview of Modern Networks

One model of networks has two device types, end-hosts and core devices. End-hosts generate data and core devices transfer data. End-hosts (hereinafter referred to as hosts) are devices that reside on network edges. People interact with hosts. Example hosts are laptops, printers, smart cellular phones, and video game systems. HTTP servers and DNS servers are also hosts, however most users rarely physically interacts with them. Core devices are all devices that are not hosts. Generally, a core devices primary purpose is to facilitate data transfer. Some examples of core devices are routers, satellites, cellular towers, firewalls, and switches.

Networks can be divided into two broad categories, circuit switched networks and packet switched networks [9]. These categories are separated by their mechanism for sharing resources. The primary network resource is bandwidth. Bandwidth is the maximum data amount allowable within a period of time. Bandwidth is traditionally measured in bits per second (bps).

In a circuit switched network, the network resources along a path between two communicating devices are reserved before the session begins. A session is a continuous period of communication between two devices. Reservation of resources is referred to as call setup. Circuit switched networks provide sessions with bandwidth guarantees. Inefficient use of channel resources occurs when sessions do not fully utilize resources at all times. These inefficient periods are referred to as *silent periods*.

During a silent period no data is transmitted in a session. In a circuit switched network, sessions have reserved resources, therefore other calls may be delayed even though the resources are theoretically available. Figure 2.2 illustrates this phenomena in circuit switched networks. Networks following this pattern are referred to as *barbell network* because of their resemblance to barbells. Suppose two simultaneous sessions must take place ($lt \leftrightarrow rt$ and $lb \leftrightarrow rb$) both reserving x bps. Only one session could take place if channel L has less bandwidth than $2x$. In a circuit switched network this is true regardless of the duration or frequency of silent periods..

Packet switched networks were designed to utilize silent periods more efficiently. In packet

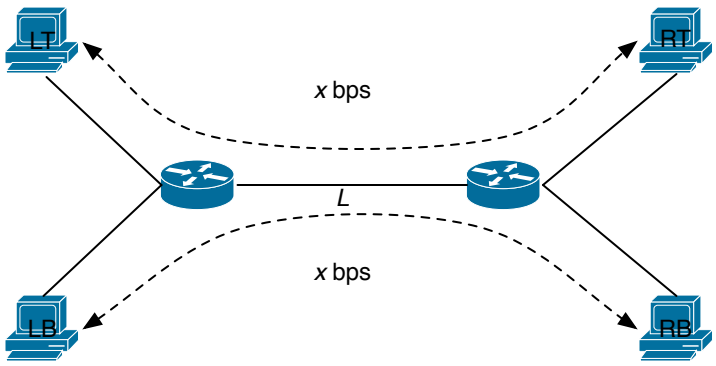

Figure 2.2: Two pairs of hosts are trying to communicate through one channel L. Communication between the left top host and the right top host, respectively lt and rt, reserves x bps. Communication between the left bottom host and the right bottom host, respectively lb and rb, reserves x bps. If L cannot provide $2x$ bps, one of these sessions cannot occur regardless of whether the active session is sending data.

switched networks data is parceled into *packets*. A packet can also refer to the unit of data at the network layer (discussed in Section 2.2.3). Throughout this documents *packet(s)* refer to parceled data on packet switched networks and *network layer packet(s)* refer to network layer data units. Packets are distributed through the network in a store-and-forward fashion, one link at a time. A link is a channel that connects any two devices directly through a physical medium. Generally, a device that delivers packets locally is a host and one that forwards is a core device. The decision to deliver locally or forward a packet is a function of packet addressing information and device configuration. Information within a packet that determines ultimate packet destination can also be referred to as control information.

Packet switch networks are built using a layered architecture. Each layer is a collection of functionality targeted at a similar goal. An implementation of a layer, is a layer instance. For example, the goal of the link layer (present in both the TCP/IP model and OSI reference model, descriptions follow shortly) is to coordinate communication between two directly connnected devices. An instance of a the link layer would be Ethernet (IEEE 802.3)[8].

Any particular layer presents a delivery interface to the layer directly above and a reception interface to the layer directly below. The top-most and bottom-most layers are exceptions, respec-

tively missing the delivery interface and reception interface. The delivery interface marshalls the data from the above layer and adds information for intra-layer communication. The reception interface reverses the marshalling transformation of the delivery interface and removes any intra-layer information.

The OSI[10] reference model and TCP/IP[10] model are the two most popular layered architecture representations. The OSI reference model is primarily used in conceptual understanding of networks. It has a corresponding implementation (referred to as the OSI stack) that is not in general use. The TCP/IP model describes a practical, implemented, layered architecture. It models, with some loss of precision, how most modern packet-switched networks operate.

Functionality increases when traversing the layers from lowest to highest (physical layer to the application layer). Each layer provides new guarantees and functionality to its subsequent layer. The top most layer is an application running on a device. The bottom most layer encodes and decodes symbols into physical phenomena.

2.2 A Brisk Introduction to The TCP/IP Reference Model

Table 2.1 summarizes the TCP/IP model and OSI reference model. A network can be described using either. This document prefers the TCP/IP model because it closely resembles network implementations, something pcapstitch is sensitive to. This model has five layers, each layer has a device responsible for implementing the layer instance. The five primary devices (respectively from lowest layer to highest layer) are Communication Devices, Switch, Router, Host and NAPT Router, and Host. Additionally, each layer has two well-known instances:

- Physical Layer - Ethernet Physical and 802.11 Physical
- Link Layer - Ethernet and 802.11
- Network Layer - IPv4 and IPv6
- Transport Layer - TCP and UDP
- Application Layer - HTTP and DNS

Table 2.1: TCP/IP model and OSI Reference Model. The *Responsible Device* column describe network devices that typically maintain the instance for each layer. The *TCP/IP Layer Name* column are the TCP/IP Layers. The *OSI Layer Name* column are the OSI Layers. The *Message Unit* column gives the name of the payload at each layer.

Responsible Device	Common Protocols	TCP/IP Layer Name	OSI Layer Name	Message Unit
Host	HTTP,DNS	Application	Application Presentation Session	Data
NAPT router	TCP,UDP	Transport	Transport	Segment
Router	IPv4,IPv6	Network	Network	Packet
Switch (alternatively Hub or Bridge)	Ethernet,802.11	Link	Link	Frame
Communication Device	Ethernet Physical,802.11 Physical	Physical	Physical	Bit

Layer instances can also be referred to as protocols. This document attempts to consistently use the term protocol preceded by the particular TCP/IP layer if not obvious. For instance, TCP is a transport layer protocol.

This section is not exhaustive, it describes devices and protocols to the resolution necessary to understand pcapstitch. It provides a general description, but in many cases exceptions to these description can be made. For example, a router often has a remote management interface accessed through transport and application protocols to facilitate configuration. This would technically make the router a host; such exceptions are hidden in the interest of clarity.

The remainder of this section contains sub-sections providing a high-level overview of the responsible device and layer instance. Section 2.2.1 describes communication devices, the Ethernet physical protocol, and the 802.11 physical protocol. Section 2.2.2 describes switches, the Ethernet protocol, and the 802.11 protocol. Section 2.2.3 describes routers, the IPv4 protocol, and the IPv6 protocol. Section 2.2.4 describes NAPT routers, the TCP protocol, and the UDP protocol. Section 2.2.5 describes the host, the HTTP protocol, and the SIP protocol.

2.2.1 Physical Layer: Communication Devices, Ethernet Physical Protocol, and 802.11 Physical Protocol

Communication devices are associated with the physical layer. They convert symbols representing information into some physical data representation. Bits are the fundamental symbols of digital communication devices.

Physical mediums (hereinafter referred to as mediums) can provide one or more channels. A channel is method for communication. Communication devices usually adhere to standards that detail wiring, transmission, reception, frequency bounds, valid mediums, and modulation schemes.

Ethernet Physical and 802.11 Physical are common examples of physical layer protocols. The Ethernet[8] physical layer is a very common wired computer network physical protocol. It has a number of variants that can create channels on co-axial, twisted pair, or fibre-optic medium. Most Ethernet Physical protocol variants are defined as a IEEE 802.3 standard. The IEEE 802.11 physical layer is a common wireless network physical protocol. 802.11 variants (*i.e.*, a, b, g, n)

defines encoding and decoding across Direct Sequence Spread Spectrum, Frequency Hoping Spread Spectrum, and Infrared.

pcapstitch relies on network trace files collected in the libpcap format (both are discussed in more detail in 2.3). This file format does not contain any information regarding the physical layer. However, the physical layer can be the source of delay and loss (discussed in 2.4). pcapstitch can aid diagnosis of such physical layer issues.

2.2.2 Link Layer: Switches, Ethernet, and 802.11

Switches are devices that implement link layer protocols. They deliver frames between devices on a Local Area Network (LAN). A LAN is a collection of devices with unique link addresses within a single address space. In other words, a LAN is the set of devices that can communicate solely using link layer addresses.

A switch is composed of a set of ports and an interconnect fabric. When referring to network devices, ports are physical interfaces that accept some medium. The interconnect fabric uses link addresses to transfer a frame from a device on one port to another device on a different port. Generally, this can be referred to as *routing*. A similar procedures is performed by routers, therefore this process will be referred to as *switching* to avoid confusion.

Switches compromise channel contention and channel cost. Channel cost can be reduced by forcing all devices in a LAN to share one channel. No two devices can use this channel at the same time increasing the potential for contention. Contention can be reduced by creating a channel between all-pairs of devices, this is impractical as the number of devices increases. Switches compromise the two concerns by sharing multiple channels and distributing contention among them.

In practice the link layer has two sublayers, Medium Access Control (MAC) and Logical Link Control (LLC). The LLC sublayer is positioned above the MAC sublayer within the link layer. The MAC sublayer is responsible for negotiating usage among multiple devices on the same channel. The LLC is responsible for presenting a consisting interface to the network layer and multiplexing/demultiplexing frames for multiple network protocols.

Ethernet and 802.11 are common examples of link layer protocols. Both protocols use the IEEE

802.2 LLC sublayer. Network protocols interacting with Ethernet or 802.11 therefore can use the same High-Level Data Link Control[10] (HDLC) interface.

Ethernet and 802.11 share similarities at the MAC layer as well. Both protocols rely on a MAC addresses embedded in the link layer frames. Both protocols rely on a 32 bit Cyclic Redundancy Check (CRC32) to detect errors within a frame that may have occured during transmission. The protocols differ in channel control methods. Ethernet performs Carrier Sensing Multiple Access with Collision Detection (CSMA-CD). Collision detection(CD) allows the Ethernet MAC sublayer to detect when two devices are transmitting on the same channel at the same time. If this event is not avoided collision occurs. Carrier Sensing Multiple Access(CSMA) is an uncoordinated procedure for accessing a channel. CSMA determines appropriate transmission time based on a silent period and a backoff period determined by previous collisions. 802.11 performs Carrier Sensing Multiple Access with Collision Avoidance (CSMA/CA). 802.11's CSMA algorithm offers similar functionality as Ethernet's CSMA algorithm.

Collision detection in wireless networks is not feasible. This arises from practicalities of wireless transmissions:

1. Wireless antennas are half-duplex and therefore cannot receive and send simultaneously.

2. Devices with multiple antennas will be located closer together, so it is likely that the any received signal will be dominated by the locally sent signal.

Collision avoidance modifies CSMA slightly by adding a probabilistic wait time after a silent period. By staggering all potential senders collisions are reduced. When a collision does occur it will result in a failed CRC32 check and the frame will be dropped.

pcapstitch has full visibility into link layer frames. Control information in link layer frames is not used for packet stitching. pcapstitch can measure delay and loss which can occur at the link layer.

2.2.3 Network Layer: Routers, NAPT Router, IPv4, and IPv6

Routers are devices that implement network layer protocols. Routers deliver packets between devices on Wide Area Networks (WAN). A WAN is a collection of LANs or WANs. Network layer addressing within a WAN is hierarchical. Figure 2.1 illustrates the difference between LANs and WANs.

A router is composed of a set of ports and an interconnect fabric. The interconnect fabric uses network addresses to direct packets between two ports. A WAN may contain WANs; intra-WAN routing reduces the set of possible destinations at any given router.

Unlike MAC addresses (discussed int Section 2.2.2), network addresses are assignable. Routers require a way of dynamically resolving routes between device pairs as network address are reassigned. This is why network layer routing is usually hierarchical. If it were not, a router would require enough memory to store route information about every unique network address. Hierarchical routing solves this problem by allowing routers to have one address that generalizes many individual addresses.

A packet may go through multiple routers on its journeys to a destination device. The channels and routers packets traverse en route to destination devices is called a path. A single channel between between two routers (or between any two devices) is referred to as a link. Throughout this document, link is used to mean a single physical channel between two devices. This name is convenient because communication between any two devices, directly connected or through a switch, is primarily handled by the link layer. Consequently, a path is made up of devices and links between those devices. The difference between links and paths is illustrated in Figure 2.1.

The Internet Procotol(IP[11]) is the most commonly used network protocol. It has two major variants: Internet Protocol Version 4 (IPv4[11]) and Internet Protocol Version 6 (IPv6[12]). The main differnce between the two version is addressing space; IPv4 uses a 32 bit representation for addresses (2^{32} possible addresses) and IPv6 uses a 128 bit representation for addresses (2^{128} possible addresses). Other significant IPv6 differences include the lack of packet fragmentation, a simpler packet header, and tighter coupling with security mechanisms.

pcapstitch has full visibility into network layer packets. Currently, pcapstitch only supports IPv4. pcapstitch relies heavily on control information within the IPv4 packet to stitch packets correctly.

Control information in IPv4 that can change from device to device along a path is not used by pcapstitch. IPv4 packet fields such as Time-to-Line(TTL)[11], checksum[11] and various optional fields are example of fields that may change in transit.

Network layer devices that change field devices in unusual ways may negatively effect pcapstitch accuracy. A Network Address Translation (NAT) device is an example of such a device. NAT devices provide a one-to-one mapping between network addresses. Information about IPv4 packet control information used by pcapstitch is found in Section 3.1. Details about packet transformations that can effect pcapstitch accuracy is found in Section 3.4.

2.2.4 Transport Layer: Host and NAPT Router, TCP, and UDP

The transport layer is responsible for delivery of segments to hosts. Generally, these protocols are implemented in the host network stack. The host network stack consists of layered protocol implementations used for communication. There are core devices that also implement transport layer protocols. Firewalls, throughput throttling devices, and Network Address-Port Translation Routers (NAPT routers) are examples of such devices.

As in the network layer (Section 2.2.3), pcapstitch uses control information from transport layer protocol segments to stitch packets. Devices that change certain control information fields in transport layer segments can effect pcapstitch stitching accuracy adversely. A NAPT router is an example of such a device. NAPT routers use source address (network layer) and source port (transport layer) information to map to another source address and source port. Unlike NAT devices, NAPT routers can provide many-to-one mappings. An algorithm maps many source address and source port pairs to one source address and different source ports. NAPT routers can coordinate traffic from many different hosts with many different source address through one address. Figure 2.3 illustrates NAPT router functionality.

Transmission Control Protocol[13](TCP) and User Datagram Protocol[14](UDP) are two very common transport layer protocols. TCP provides reliable in-order delivery. UDP provides unreliable user-segment-preserving delivery. Both protocols offer application multiplexing through port numbers. Port numbers are transport layer addresses. Multiple application sessions are possible

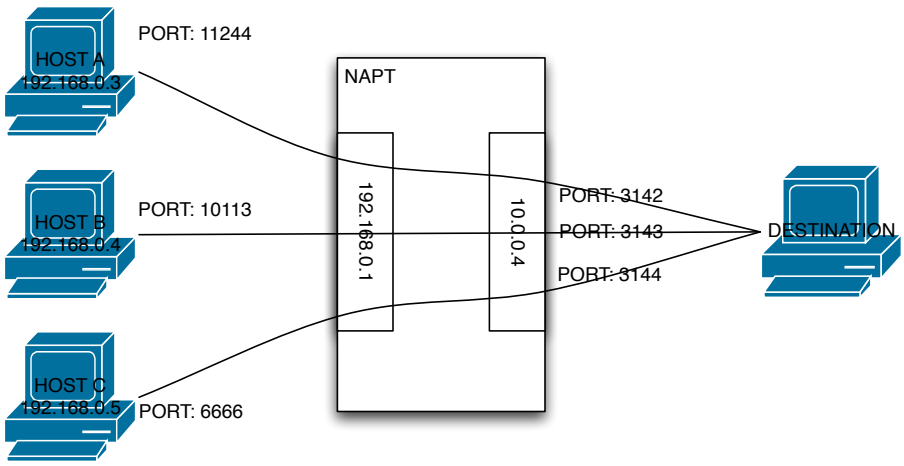

Figure 2.3: In this figure three hosts (A,B,and C) are trying to communicate to *Destination* through $NAPT$. $NAPT$ is a Network Address and Port Translation router. Each host is labeled with a source IP Address. Traffic from each host bound for *Destination* has its source port labeled prior to entering $NAPT$. The host IP and port prior to $NAPT$ carries original addressing information. The destination IP and port of *Destination* is irrelevant for explain NAPT router functionality. Each host routes traffic through $NAPT$ interface labeled as 192.168.0.1. $NAPT$ translates the addresses and ports of these connections such that all traffic source addresses read as 10.0.0.4 from *Destination*'s perspective. $NAPT$ differentiates return traffic through port translation; connections associated with host A,B, and C map to ports 3142,3143, and 3144 respectively. In this way NAPT routers can make traffic from many different hosts appear to come from only one.

over a single channel due to port numbers.

TCP reliability is provided through an initiated connection using acknowledgements. Acknowledgements notify a sender that a segment has been received. After some amount of time, the lack of an acknowledgement can indicate that a segment was not successfully transmitted. During TCP connection initiation two hosts negotiate initial byte count offsets. Sent Segments (measured in bytes) increment the byte count offset. Upon reception of a segment an acknowledgement is sent that indicates the last contiguous byte offset. Closing a connection is also negotiated to ensure that all sent segments have been received. At any state, there are mitigation strategies for dealing with lost segments; segment retransmits at increasing intervals followed by ungraceful disconnection.

UDP only guarantees that application data separation is preserved. UDP differs from TCP in this regard where data is transmitted as a stream requiring applications to encode there own separation. Another difference is UDP's lack of reliable data transfer.

pcapstitch has full visibility into transport layer segments. Currently, pcapstitch only supports TCP and UDP. pcapstitch uses TCP and UDP source and destination port control information to stitch packets. NAPT routers will negatively impact pcapstitch accuracy, more information can be found in Section 3.4.3.

2.2.5 Application Layer: Host, HTTP, and DNS

The application layer is the endpoint of any session. A host is any device that implements an application layer protocol. Consequently, a device on the edge of network, a device with only one link, or an end system is a host. Traditionally, hosts are clients and servers. Clients are host that request most of the data in a session. Servers are host that deliver most of the data in a session.

Application protocols transmit data. Packet payload is another way of referring to application protocol data. Payload is data in a packet that is not control information (i.e. addressing, checksums, protocol directives).

Exactly whether a device is referred to as a host is context specific. Using the web configuration mechanism of a switch makes it a host. However, it is still also a switch. In general the role of a device can be determined by its purpose for existence. If its primary purpose is to execute applications it

is a host, otherwise it is a core device.

There are many application protocols. Hyper-Text Transfer Protocol[15](HTTP) and Domain Name Service[16] are two common application protocols. HTTP is what facilitates web browsing with applications like Firefox, IE Explorer, and Safari. HTTP essentially supports a request/response protocol. A client specifies what is desired or what is required respectively through GET and POST methods. A server responds with an error code or the content of the request.

DNS converts textual names to IP numbers usable by IPv4 or IPv6. `www.google.com` is an example of such a textual name. DNS subtleties regarding domain name resolution are unnecessary for a discussion of pcapstitch.

pcapstitch has full visibility into application layer data. Application data is used for packet stitching by default using a hash function. This can be changed by customizing the PEF (Section 3.1).

2.3 Network Traces

A network trace (hereinafter referred to as trace) records activity on a channel. Network capture, packet capture, network sniff, and packet sniff are alternative terms for trace. A device that is running a trace is a network observation point (hereinafter referred to as an observation point). Traces can be recorded to network trace files (hereinafter referred to as trace file).

Trace files contain information about packets sent and received over a channel. Exactly what data is recorded and the format of the trace file depends on the specific network trace utility (hereinafter referred to as trace utility). pcapstitch requires trace files recorded on different channels simultaneously.

Trace files are effective for monitoring network health. Network health is composed of security, configuration, and performance.

Understanding the characteristics of malware is useful for prevention, detection, and mitigation. Malicious software or malware, is software that gives covert access of your device to an unauthorized party. It is common for malware to coordinate efforts via network communication. Trace files collected on devices running malware are useful for determining malware behavior, command and

control infrastructure, and other infected devices.

Various device protocol implementations must be configured to inter-operate on a network. Subtle assumption violations can result in insidious global behavior. Slight configuration changes may drastically alter overall network behavior. Trace files can be used to diagnose configuration errors and verify that networks are operating as expected.

Networks are ultimately designed to allow inter-application communication. Network application utility is therefore based on network performance. Network performance can be estimated through the analysis of trace files. Trace files with per-packet meta-data (*i.e.*, time, packet offset) can be used to measure capacity and per-application bandwidth. Capacity or Nominal Physical Link Capacity, "is the theoretical maximum amount of data that a link or path can support"[17]. Throughout this document capacity is defined this way unless otherwise noted. For example, capacity can be measure by sending excessive UDP traffic from one device to another. Received packets over a one second window (measured using time meta-data) can calculate maximum throughput. pcapstitch is primarily designed to measure network performance using trace files.

Timestamps recorded in meta-data are very important to network measurements. Ideally, these timestamps store the time that a packet enters "the wire" or exits "the wire". This is called "wire time", and specifically describes the time of the first packet bit entering a physical medium or the time of the last packet bit exiting a physical medium. Most trace utilities record "host endpoint time" where a host is an observation point. Host endpoint time is equal to wire time. It is later than wire time (when receiving), or earlier than wire time (when transmitting). Host endpoint time is less accurate than wire time because host delay incurred between wire time and trace utilities. Among many other delays inherent to modern computer hardware and operating systems, this delay includes the time required to retrieve packets from a physical medium, the time required to move the packet over the IO bus, and the time required for the software to retrieve the packet and timestamp it. The difference between wire time and host endpoint time is processing delay (processing delay is discussed in Section 2.4.1.

Trace utilities can be software or hardware implements. Software trace utilities take advantage

of network traffic that already exists on a device. Software utilities operate by duplicating packets that move between the Network Interface Card (NIC) and Operating system. tcpdump[18], Wireshark[19], ncap[20], and Snoop[21] are some example of software trace utilities.

Hardware trace utilities are built specifically to collect traces. Hardware trace utilities are also called network taps. Network taps can be built inexpensively by taking advantage of the physical properties of a medium. For example, http://hackaday.com/2008/09/14/passive-networking-tap/ describes how to build a network tap for Ethernet. These simple network taps still require a software trace utility to collect the data. The network tap duplicates network traffic on another link.

pcapstitch requires that trace files be stored in libpcap[18] file format. This file format is generated by the libpcap library that pcapstitch and many other trace utilities rely on. The file format specifies how packets are stored and how they should be accessed.

2.4 Delay and Loss: TCP/IP performance killers

Packet delay and Packet loss (hereinafter referred to as delay and loss) affect the performance of all TCP/IP model layers. All layers either expose the subsequent layer to the current delay and loss characteristics or transform them. There are two possible transformations that can be applied to delayed or lost packets. A delayed packet can be dropped or further delay added. A lost packet can be converted to a delayed packet via error correction mechanisms that adds delay.

The TCP/IP model offers modularity and information hiding. Different protocol implementations are interchangeable at any given layer of the model. This modularity is demonstrated in Section 2.2 where each layer is described with two example protocol implementations. TCP/IP model layers do not expose packet formatting or implementation details protecting other components from modification, exemplifying information hiding.

Unfortunately, these properties of the TCP/IP model permit inefficiencies. Delay transformations, loss transformations, and unexposed protocol assumptions can cause degraded performance. Consider a protocol stack with two protocols that engage in retransmission. Specifically, a link layer and a transport layer both using an Automatic Repeat reQuest[10] (ARQ) protocol. ARQ uses acknowledgments (or lack of acknowledgements) to determine if packet retransmission is required.

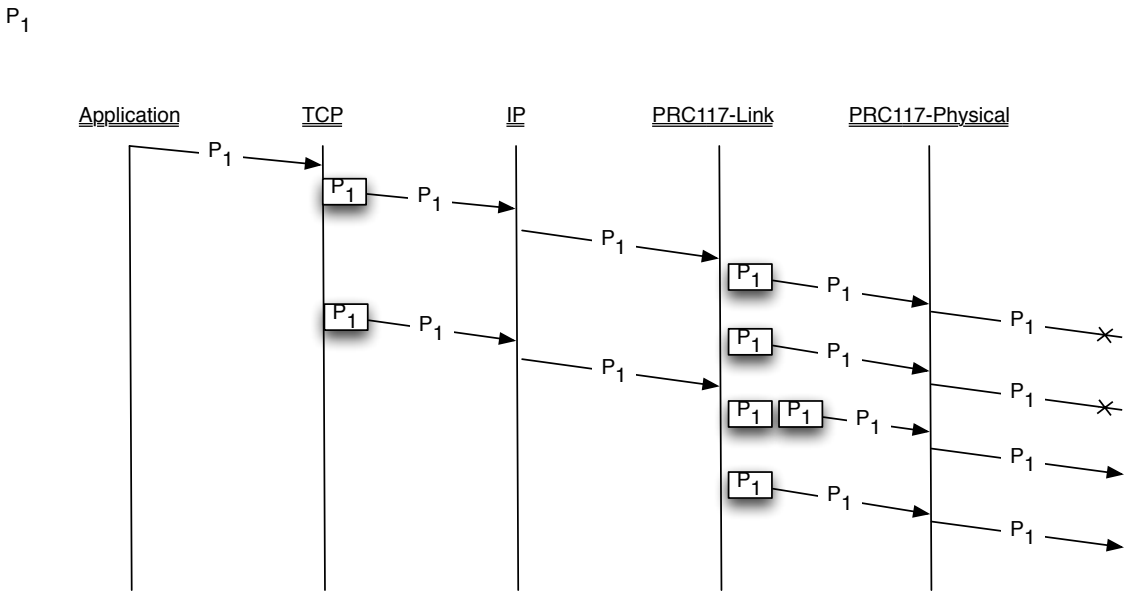

Figure 2.4: In this figure an application is trying to send a packet P_1 and processes are protocols layers. When TCP receives this packet it queues it for potential retransmission. The packet is then handled by IP and sent to the PRC117F link layer where it is queued again. The PRC117F link layer also queues this packet and tries to send it twice unsuccessfully. Time elapses and TCP, having failed to receive an acknowledgement, retransmits P_1. When the PRC117F link layer receives this retransmission it has two duplicate packets in its queue. This causes P_1 to use twice the bandwidth as necessary and if more packets needed to be sent could cause significant performance degradation.

If the transport layer retransmits a packet before the link layer has successfully transmitted the original packet, duplicate data will be transmitted.

This example is a practical concern, tactical modems such as those embedded in the AN/PRC-117F Multiband Manpack Radio[22] have ARQ capability. Modulator-Demodulators or modems are often used as link layer protocols. TCP, a transport layer protocol, also implements a form of ARQ. Using both of these protocols can result in emergent performance degradation when simply considering the TCP/IP model abstraction. Figure 2.4 illustrates the potential problems of this scenario.

Protocols in isolation can cause degraded performance if implementation assumptions are violated. TCP was built when wired, low error networks were common. TCP thus assumes that packet loss occurs as a result of congestion (how congestion causes packet loss is discussed in Section

2.4.2. It implements a congestion mitigation strategy that reduces throughput as more packet loss is observed. Over time, this reduces congestion. Unfortunately, wireless and extremely high-speed networks violate this assumption resulting in silent periods artificially imposed by TCP.

Delay and loss affect the TCP/IP protocol stack. The TCP/IP protocol stack affects application utility. Understanding application utility therefore requires the measurement of delay and loss. pcapstitch provides a general way for delay and loss to be accurately measured from network trace files.

The remainder of this section discusses delay and loss in more detail. Section 2.4.1 describes the different types of delay. Section 2.4.2 describes the different types of loss.

2.4.1 Delay

Delay is the unit of network measurement. All network measurements can be expressed in delay. Loss is equivalent to infinite delay. Bandwidth is equivalent to delay per-bit. A channel with a bandwidth of 16Kbps transmits a bit every 60 micro-seconds. The theoretical minimum delay therefore is 60 micro-seconds.

Processing delay, queueing delay, transmission delay and propagation delay are the four types of delay. Concisely put, they are, delay incurred on a device, delay incurred waiting for resources, delay incurred to put data into the channel, and delay incurred for data to traverse the channel respectively. Together they are referred to simply as delay. Delay incurred from one device to another is one-way delay. The adjective "One-way" when modifying network measurements means the measurement is taken from packets moving in one direction on a path. Delay incurred from one device to another and back is Round-Trip Delay or Round-Trip Time(RTT). Through out this document *delay* indicates one-way delay unless otherwise specified.

In Section 2.2.2 and Section 2.2.3 interconnect fabric was the mechanism used to move a packet from one port to another. The duration a packet spends in the interconnect fabric is the processing delay. The time elapsed processing a packet by network and link protocols on host is also processing delay. Processing delay is the time needed by a device to read control information and perform necessary operations. A router's processing delay is primarily routing. Processing delay in firewalls,

devices used to verify and censor incoming and outgoing data, occurs during packet inspection.

The fundamental cause of queueing delay is serial channel transmission. While a packet transmission is in progress, other packets must wait for channel availability. During this time packets must be stored in memory. The memory holding waiting packets is the queue and packets waiting in it are suffering from queueing delay. It is convenient and generally accurate to think of the queue as being First-In, First-Out (FIFO). There are many queueing disciplines more complicated than FIFO, such as Random Early Detection [23] and Adaptive Virtual Queuing [24].

Transmission delay is the time it takes a packet to enter a channel. Dividing packet size (in bits) by the bandwidth of a channel (in bits-per-second) will expose the transmission time for any packet. For example, given a 1442 byte packet sent on a 16Kbps (16384bps) channel, 0.7 seconds of transmission is incurred.

Propagation delay is the time it takes a packet to traverse a channel. The lower bound on propagation is the physical length of the channel divided by the speed of light. Transmission delay is the entry cost to a channel and is solely influenced by channel capacity. Propagation delay is the transmission cost of the channel and is influenced by channel transmission mechanisms and physical distance between end points of the channel.

Figure 2.5 illustrates the different types of delay. Transmission delay is constant and rarely adjustable for a given channel. Given the speed of modern devices processing delay can be approximated as constant unless under severe load. Changing transmission delay or processing delay usually requires hardware changes. Therefore when trying to reduce delay, queueing delay is usually addressed first. Queueing delay can be addressed by protocols and device configuration. Increasing delay is a good indication of network congestion. Queues grow due to channel contention and as they grow the latest packet waits longer than all previous packets in the queue.

2.4.2 Loss

Packet loss occurs when delay is infinite. In other words, the packet never reaches its intended destination. Error and congestion are the two causes of loss and how types of loss are categorized. Together they are referred to simply as loss. When addressing them separately they are referred to

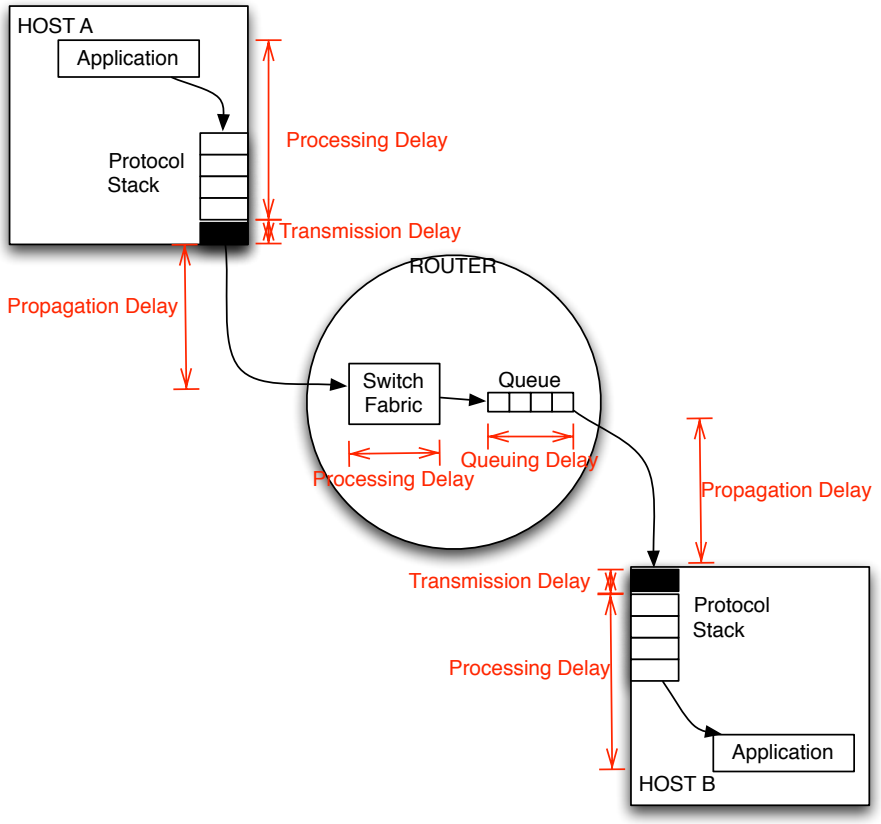

Figure 2.5: This figure shows a packets path from host A to host B through a router. Sections along the path are labeled with what type of delay is occurring on that section.

as loss due to error or loss due to congestion.

Spurious modification of one or more bits in a packet can cause loss. These spurious modifications are called bit errors. Physical phenomena, faulty hardware, and improper configuration all can cause bit errors.

These errors indirectly cause the packet to be discarded. Various protocols across all layers implement error detection algorithms. Checksums and CRC are examples of error detection algorithms. These algorithms can detect certain errors. A packet is discarded upon detection of an error. A discarded packet will never reach its destination and therefore is categorized as a lost packet. Practical network engineering favors packet discarding over processing spuriously modified packets. The latter can result in unknown behavior.

Loss due to congestion is intimately related to queueing delay (Section 2.4.1). Device queues are limited by memory. Memory limits dictate how many packets can be queued for a channel. If a queue is saturated, a new packet that requires queuing must be dropped. When this occurs it is considered packet loss due to congestion. Congestion on the channel filled the queue leading to a packet loss.

pcapstitch measures loss by counting unstitched packets. Unstitched packets are packets that have not been associated with any other packet. Loss measured by pcapstitch is also bounded in time. The lower bound for the loss is the time the lost packet was sent. The upper bound for the loss is the time of last successful packet reception where the paths for both packets (last successful packet and lost packet) were identical. Packet loss bounding is illustrated in Figure 2.6.

2.5 Network Performance Metrics

Consider the three charts in Figure 2.7. The top-most chart shows the performance of TCP (Linux 2.4.20) under optimal conditions (100Mbps, 0% packet loss, 1 <ms delay). Middle-most and bottom-most show the same TCP implementation on a link with 5% and 10% packet loss respectively. Such throughput degradation would likely impact application utility. Without additional information (like specification of link packet loss rate shown in these charts), observing throughput degradation is insufficient to indicate packet loss. Measurements are needed to identify the cause of throughput

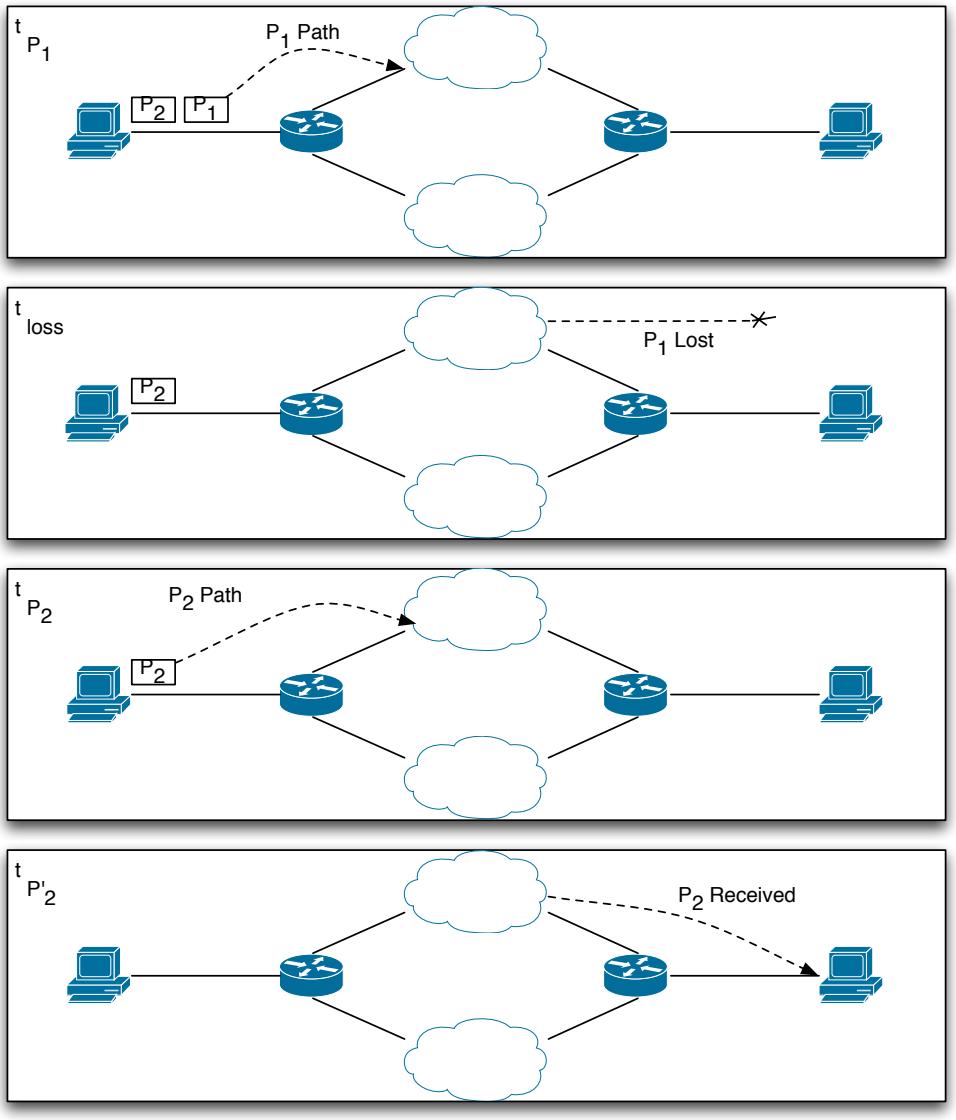

Figure 2.6: An illustration of how loss time can be bounded. P_1 and P_2 are two packets. At t_{P_1}, P_1 is sent along the upper path. At t_{loss}, P_1 is lost. At t_{P_2}, P_2 is sent along the upper path. At $t_{P_2'}$, P_2 is received at the same destination that P_1 was destined for. Assuming that P_1 and P_2 would have shared the same path and P_1 isn't being held somewhere in the network, we can bound the time of loss by t_{P_1} and $t_{P_2'}$.

degradation.. Measuring performance metrics like packet loss rate is important to understand how network characteristics affect application utility.

The Internet Engineering Task Force[25] (IETF) IP Performance Work Group has standardized network performance metrics (hereinafter referred to as metrics) that measure network characteristics. pcapstitch is a tool for collection of some of these metrics; particularly singleton one-way delay and loss. In Section 2.6 competing tools are evaluate for their ability to retrieve these metrics passively. Currently, there is only one tool which can do this (discussed in Section 2.6.6). It has a number of infrastructure requirements where pcapstitch only requires libpcap, a tool available on virtually any platform.

In this section, the metrics pcapstitch can collect are described. For convenience they may be given shortened names. Each metric has one or two short examples of how it could indicate performance issues at various layers. These examples are not exhaustive, but are given for additional intuition about metric usefulness.

Methodologies describe how metrics are collected. Reading a sun-dial or a digital clock are two different methods for measuring time. The method of metric collection in pcapstitch relies on network trace files and stitching. Network trace files are discussed in Section 2.3 and stitching is discussed in Section 3.1.

A method can rely on active measurement or passive measurement. The conservatism of a method refers to the likelihood of the methodology influencing the measured results. Passive measurements are more conservative than active measurements. Passive measurements rely on network traffic that arises from natural use. In other words, traffic not generated solely to take measurements. Active measurements rely on artificial network traffic, that is traffic injected into the network solely for measurement.

pcapstitch is passive in its measurement methodology. It uses traffic recorded at multiple observation points simultaneously; no extra traffic is required. However, care must be taken during collection because even passive measurement methodologies can influence measurements. Descriptions of tools that perform passive or active measurement can be found in Section 2.6

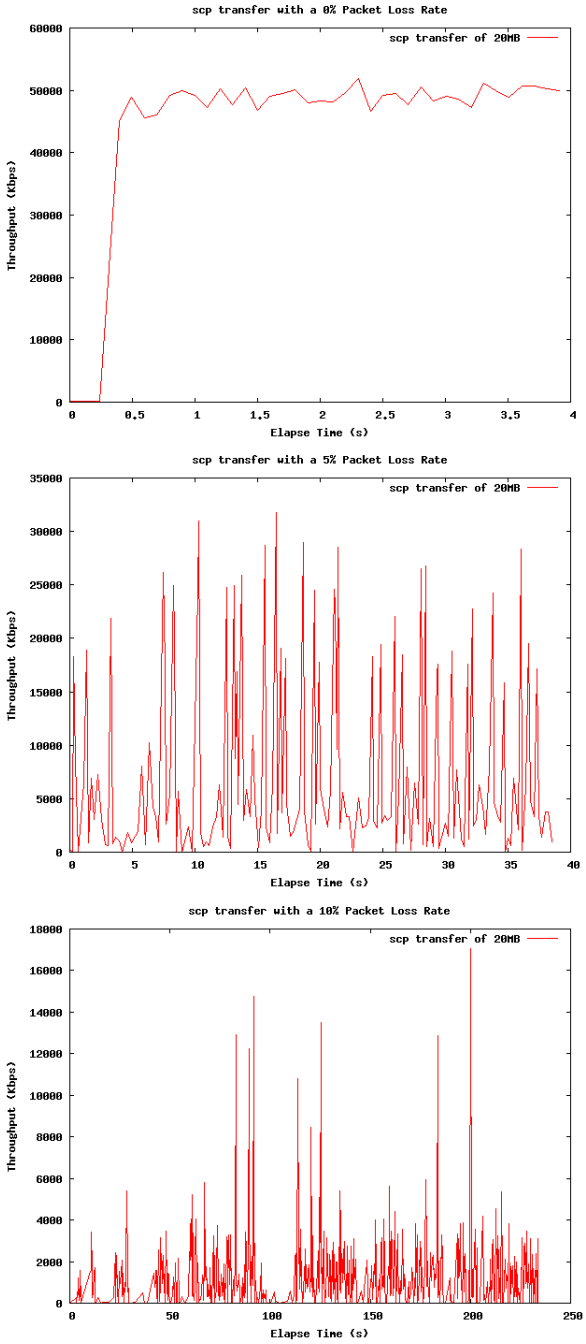

Figure 2.7: This figure shows TCP performance measured with respect to throughput. It was done using SCP an application that relies on TCP. Along the x-axis is time. All three charts were measured on a link with a capacity of 100Mbps and 1 <ms delay. In the top-most chart, the TCP performance measurement is on a link with 0% packet loss. In the middle-most chart, the TCP performance measurement is on a link with 5% packet loss. In the bottom-most chart, the TCP performance measurement is on a link with 10% packet loss. (NB: Data throughput data was retrieved from trace files using pcapstitch.)

Metrics can be of singleton, sample, or statistical type. A singleton metric is a unit measurement. pcapstitch collects singleton metrics, that is, a measure of one-way delay and loss is reported for each packet record in a trace file. A sample metric is a collection of singletons. Running pcapstitch over trace files with more than one packet will result in a sample. A statistical metric derives some useful information from a sample or samples. The minimum delay of all results reported by pcapstitch would be statistical.

Statistical metrics are derived metrics and singleton metrics are original metrics. By way of example, human height is an original metric, average height by race is a derived metric. Collection methods of original metrics involve actual measurement. Derived metrics are synthesized from one or more original metrics.

Metrics always refer to data in terms of bits. This is the unit of data on networks. However, packets are usually discussed in terms of bytes. A byte is eight bits, throughout this document conversion is not necessarily explicit. Time is in seconds possibly with metric modifiers (*e.g.*, milliseconds, microseconds). Rates will always be measured in some unit per seconds.

The rest of this section discusses metrics that pcapstitch can measure. Section 2.5.1 describes a one-way delay metric. Section 2.5.2 describes a packet loss metric.

2.5.1 Delay

pcapstitch can measure Type-P-One-way-delay[26]. As explained in Section 2.4.1, throughout this document one-way delay is referred to as delay unless otherwise noted. Delay measures the duration taken for a packet to travel from a source device to a destination device. When discussing or measuring delay, the terms source and destination will be used to indicate the source device of the packet and the destination device of the packet.

RFC2679[26] describes an active method to collect sample metric Type-P-One-way-Delay-Poisson-Stream. This method collects singletons measured by injecting traffic into the network according to a Poisson distribution into a sample. The Poisson distribution is used because it has been shown to be robust to synchronization and minimize measurement influence.

pcapstitch measures passively via trace files. Therefore, a sample discipline to avoid synchroniza-

tion and measurement influence is unnecessary. A sample metric that contains all recorded delays within a time frame replaces it. This sample metric is referred to as a delay sample set. Using the nomenclature from RFC2679, Type-P-One-way-Delay-Stream metric.

Type-P-One-way-Delay-Median and Type-P-One-way-Delay-Minimum, delay central tendency and minimum delay respectively, are two common metrics pcapstitch can be used to collect easily. These metrics as defined in RFC2679 using Type-P-One-way-Delay-Poisson-Stream as the sample source. When using pcapstitch, these metrics will use Type-P-One-way-Delay-Stream. Again, since no active measurement is being used, there is no need to apply a sample discipline.

Delay measurements can be used to estimate delay central tendency, calculating delay fluctuations, estimating link capacity, and estimating congestion. Estimating delay central tendency before deploying applications and protocols with potential for conditional delay intolerance is crucial to maintain application utility. If a protocol using an ARQ based algorithm with a maximum retransmission time of 50ms and 4 maximum retransmission attempts is on link with an estimated delay central tendency of 200ms throughput, performance will suffer. Redundant data would be retransmitted irrespective of whether it was successfully received.

Applications or protocols that depend on consistent delay measurements need to understand delay fluctuations. Consistent delay, even if very long, is preferable to delay fluctuations when running Voice over IP applications. Delay fluctuation is also referred to as jitter.

The minimum delay along a path can provide a lower bound on capacity. Recall that capacity is the maximum theoretical throughput and a path is a collection of links a packet traverses from source to destination. Any link on a path cannot have a lower capacity than the minimum measured delay times the maximum packet size in bits. Figure 2.8 illustrates this be example.

2.5.2 Packet Loss

pcapstitch can measure Type-P-One-way-Loss[27]. Throughout this document this measure is referred to simply as loss. Loss measures whether a packet was successfully received at some destination.

Type-P-One-way-Average can be used to understand central tendency of loss along a path or

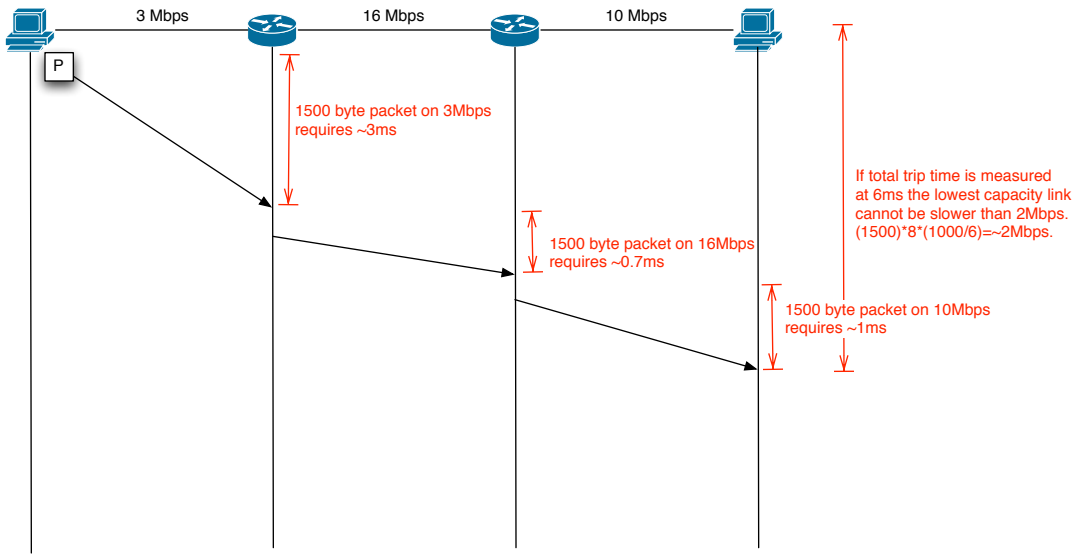

Figure 2.8: This sequence diagram shows a packet traveling along a path composed of links of differing bandwidths. Measuring the delay along the path can give us a lower bound on the capacity of any given link.

link. This metric is defined in RFC2680 using Type-P-One-way-Packet-Loss-Poisson-Stream as the sample source. When using pcapstitch, these metrics will use Type-P-One-way-Loss-Stream. Again, since no active measurement is being used, there is no need to apply a sample discipline.

Loss measurements are most useful in understanding performance degradation of protocols and applications. For example, as discussed in Section 2.2.5, TCP retransmits packets when loss is estimated. TCP assumes loss is due to congestion, thus when a retransmission occurs it increases it retransmission interval exponentially. TCP allows a finite amount of outstanding unacknowledged transmitted packets. Because an acknowledgement represents the last contiguously received byte, a lost packets can significantly degrade transmission performance. Figure 2.9 provides an example of how loss can degrade TCP performance.

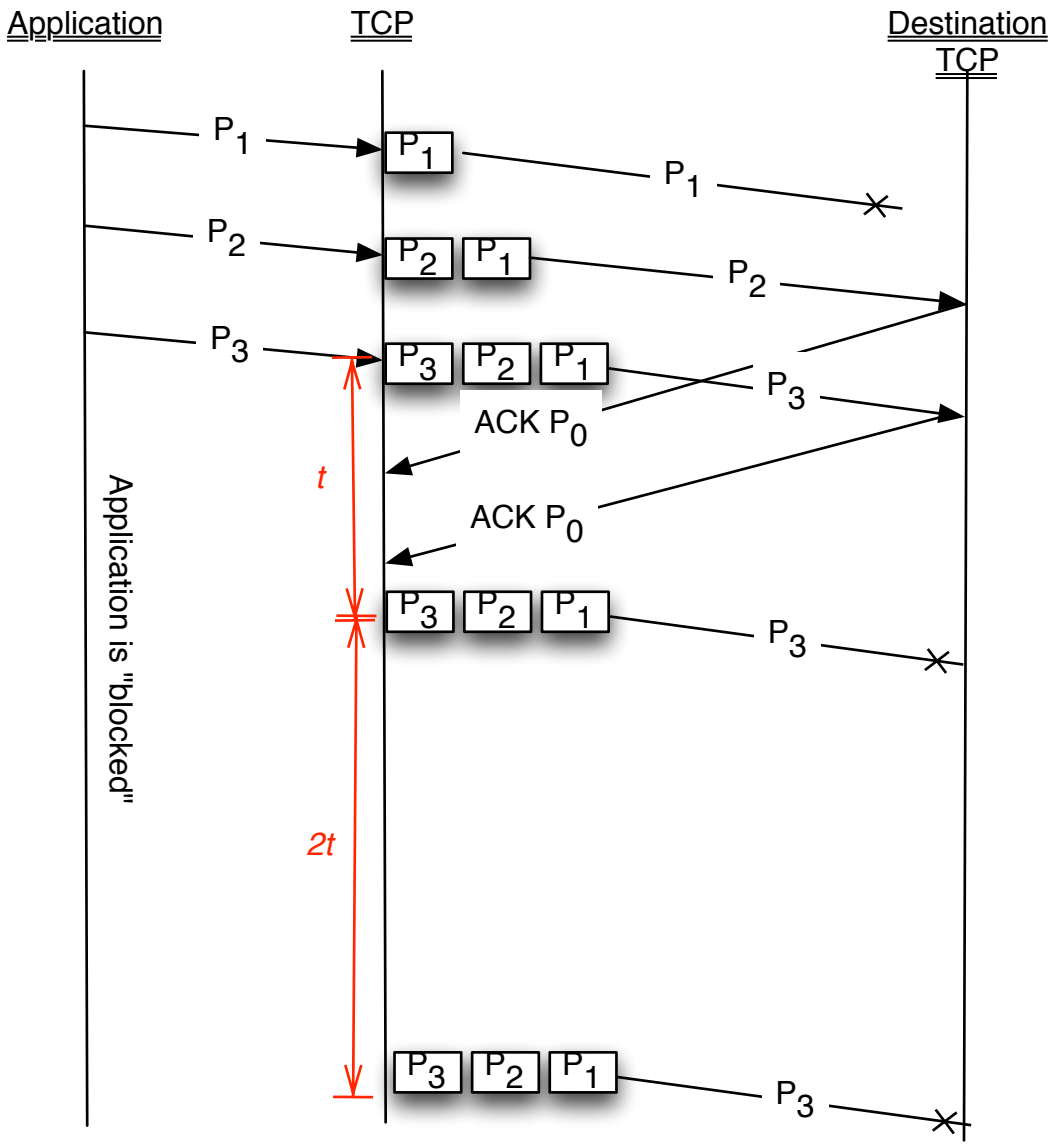

Figure 2.9: This sequence diagram represents an application using TCP to communicate to a destination host. It demonstrates how loss can negatively effect TCP throughput performance. Intermediate layers and exact TCP operation details are elided for clarity. P_0 is the initial byte sequence. The maximum number of outstanding packets permitted by this TCP implementation is 3. P_1 is sent and lost, P_2 is sent and acknowledged with P_0 since the receiving host is missing P_1, likewise with P_3. At this point the application can send no more data because the TCP send window is full. TCP enters backoff period of t before retransmitting P_1 which is again lost causing a $2t$ backoff period. An exponential backoff pattern will be repeated until P_1 is successfully received or a maximum retransmission limit is exceeded. During this time, the application can send no more data.

2.6 Measurement Tools Related to pcapstitch

Network measurement is important toward understanding application utility and protocol performance. There are many tools that can be used to passively or actively approximate or measure one-way delay and/or loss. This section will choose the most popular variants of these tools.

pcapstitch was built to fill a particular role; simple, UNIX friendly, passive measurement of one-way delay and loss from trace files. To our knowledge OpenIMP[2] and pcapstitch are the only tools that can measure delay and loss passively for general application data using only libpcap trace files. pcapstitch differentiates itself from OpenIMP by being simple, lightweight, and aligning closely with the UNIX philosophy[28].

The rest of this chapter is separated into sections that describe popular tools that can measure or approximate one-way delay and/or loss. Section 2.6.1 describes ping and traceroute, tools that can actively approximate one-way delay and loss using ICMP messages. Section 2.6.2 describes tcptrace, a tool that can passively approximate one-way delay and loss of TCP traffic from trace files. Section 2.6.3 describes iperf, a tool that can actively measure one-way loss. Section 2.6.4 describes itg, a tool that can actively measure one-way loss and delay of UDP traffic. Section 2.6.5 describes OWAMP, a protocol with a number of implementations that is intended for active one-way measurements. Section 2.6.6 describes Trajectory Sampling and OpenIMP, OpenIMP implements a form of Trajectory Sampling that can be used to measure one-way delay and loss.

2.6.1 Ping and Traceroute

The Internet Control Messaging Procotol (ICMP) [29] is used for network diagnostics. It can be classified as a transport protocol since it relies on the network layer. However, in many cases the network layer and ICMP are more intimately involved than the other layers. For example, if a router on LAN L receives a packet destined for host A on L that was sent from host B on L, a ICMP redirect message may be sent to B. It would indicate that future messages should be sent directly to A via the link layer. This interaction between the network layer and ICMP intentional breaches modularity and information hiding principles.

The ICMP echo request is a simple but useful ICMP message type. A network device configured appropriately will reply to an echo request. This reply will be sent to the host that generated the original echo requst. Echo requests and replies can be used to identify and debug network and device issues.

Ping[30], an active network measurement tool common to many platforms, uses ICMP echo messages for network characteristic discovery. Ping periodically sends echo requests to a host specified by an IP address. Replies that can be matched with previously sent requests are successful *pings*. Each request is time-stamped so that a round-trip-time (RTT) can be calculated upon receiving a reply. If a reply to a request is never received, it is considered a loss.

Traceroute[31] is used identify hops along a path. It uses ICMP echo message and manipulation of the time-to-live (TTL) IP field. The TTL field is used to prevent messages from remaining in network loops indefinitely. The field with is 8 bits and can have a valued between 0 and 255. Each time an IP packet moves through a core device, its TTL field should be decremented (refer to the next sentence for why "should" is used instead of "is"). If a core device (described in Section 2.1) receives an IP packet with a TTL value of 1, the packet has expired and should be dropped (in both case "should" is used because there is no way to enforce externally that core devices behave in this way). A device may send an ICMP message to the sender notifying them that some packet has expired. Therefore, it is possible to "trace" the links in a path by sending ICMP echo messages with incrementing TTL fields.

Ping and Traceroute can be used to approximate one-way delay by halving singleton RTTs. Asymmetric to and from paths would cause such approximations to be error prone. Assuming symmetric paths would not exclude such delay approximations from errors; asymmetric queue delays occur on symmetric paths.

Ping and Traceroute can also be used to approximate one-way loss. When a loss occurs it is impossible for ping to determine if the request or reply was dropped. Thus, the average loss rate measured using ping is only an upper bound on the one-way loss rate.

Finally, ping and traceroute are not a general tools and can only be used to measure or approx-

imate network characteristics for ICMP echo messages.

2.6.2 tcptrace

tcptrace[32] can gather detailed information from a variety of trace files. It is used to monitor and debug tcp performance behavior passively. Unlike pcapstitch, it cannot stitch sent packets in one trace file with received packets in another. tcptrace assumes all traffic was collected at a single observation point and recovers information based on RFC specified TCP behavior. tcptrace can output a number of formats including ones that can be input into other utilities to create TCP performance and behavior charts.

tcptrace can approximate one-way delay by halving singleton RTTs. Asymmetric to and from paths would cause such approximations to be error prone. Assuming symmetric paths would not exclude such delay approximations from errors; asymmetric queue delays occur on symmetric paths.

All retransmitions reported by tcptrace could be considered one-way loss. Packets taking longer than TCP retransmit time for arrival would be erroneously counted as loss. Determining whether an echo request or reply was dropped is analgous to determining whether a TCP segment was dropped or an acknowledgement was dropped. In this respect, both ping and tcptrace one-way loss approximations need to form some assumption about assymetry (or lack thereof) on a path. This technique of loss approximation is only applicable when the trace file is collected from the source host. Ostermann *et el.*[33] describe a more complex mechanism for approximating loss given TCP variants that improves accuracy.

tcptrace can also detect segment re-ordering. Segment re-ordering occurs when a host receives TCP segments with a byte offset less than the greatest byte offset seen thus far (details to handle byte sequence overflow are elided for simplicity). Two scenarios cause packet re-ordering: somewhere on the path between the source and destination host, that segment was moved with respect to its sending sequence or the first segment was determined lost by the sender and retransmitted. Approximating one-way loss using tcptrace requires assumptions with respect to how to handle segment re-ordering. This technique of loss approximation is only applicable when the trace file is collected from a destination host.

tcptrace can analyze all TCP traffic collected in a trace file but cannot be used on UDP traffic. For more information please refer to the tcptrace website: http://www.tcptrace.org.

2.6.3 iperf

iperf[34] is an application layer tool that is used to measure bandwidth for UDP and TCP traffic. Unlike ping and tcptrace, it coordinates two running iperf instances on potentially different hosts. iperf can actively measure one-way loss under certain configurations. iperf does not permit one-way delay measurements easily.

From an applications perspective, loss cannot be detected when using TCP as the the transport layer. TCP segments sent by one iperf instance will either eventually be delivered or the TCP connection will be dropped. Segments delivered by TCP may be retransmitted a number of times; iperf will not have information about these retransmissions. iperf cannot approximate one-way loss when operating in TCP mode.

iperf permits application data transfer using UDP. UDP (discussed in Section 2.2.4) does not guarantee datagram delivery and therefore will not hide loss from applications. A datagram labeled with a unique application payload can be sent and if that packet was not received, it can be considered loss. In this way, iperf can detect one-way loss when operating in UDP mode.

iperf cannot be used to measure one-way delay and loss on general application traffic. The data iperf sends is determined solely by iperf.

2.6.4 ITG

ITG[35] is a active network measurement tool built to generate network traffic in a repeatable way. One-way loss and delay can be measured from ITG generated traffic. Like iperf (see Section 2.6.3), ITG coordinates multiple ITG instance on potentially different hosts. ITG can measure one-way delay and loss under certain configurations. DITG[36] is a variant of ITG with improved perfomance, concurrency, and logging capabilities. In this section we refer to them both as ITG and choose the most competitive features with respect to pcapstitch.

ITG can generate network, transport, and application layer traffic according a stochastic process.

The process may vary with inter-departure time(IDT) or packet size (PS). IDT is the time between two consecutivetively sent packets from the same host. PS is self explanatory.

ITG can measure one-way delay and loss while operating in UDP mode. Collecting singleton measurements requires that ITG run in a mode that records sender log file (using ITGSend) and a receiver log file (using ITGRecv). ITGDec can decode these log files and output text containing singleton one-way delay and loss measurements. ITGDec can also output statistical measurements where the entire log file is treated as a sample measurement.

One-way loss cannot be calculated and one-way delay must be approximated when ITG is run in TCP mode. Loss cannot be calculated because segments sent by one ITG instance will either eventually be delivered or the TCP connection will be dropped. Segments delivered by TCP may be retransmitted a number of times; ITG will not have information about these retransmissions. One-way delay in TCP mode is approximated by the difference in payload transmission and reception time at the application layer. TCP's retranmission mechanism will convert loss to delay; this will cause error in delay measurements. TCP segment acknowledgements dropped may cause senders to retransmit unnecessarily. This will cause delay measurement error in subsequent packets; data delivery to TCP (where ITG applies time-stamps) and actual packet transmission time may diverge as previous segments are retransmitted (Figure 2.9 illustrates this phenomena).

ITG can only simulate general application traffic using stochastic processes with IDT and PS variables. Certain measurement scenarios may call for realistic application protocols and payload or accurate TCP one-way delay and loss measurement; ITG could not be used in these cases.

2.6.5 OWAMP

OWAMP (One-Way Active Measurement Protocol) is a one-way active measurement protocol describe in RFC 4656[37]. There are a number of different tools that implement OWAMP such as J-OWAMP[38] and owamp[39]. Previous to OWAMP, there was not an standard that allowed traffic exchanges for the collection of singleton one-way metrics. Like pcapstitch, OWAMP relies on high accuracy time synchronization. Such synchronization is available through GPS and CDMA devices that can be used as high accuracy clocking sources. Streams of UDP packets sent to and from hosts

running OWAMP servers is the mechanism for singleton collection. OWAMP can measure one-way delay and loss accurately.

OWAMP relies on schedules and slots to make singleton measurements. Schedules describes when packets will be sent within a slot. Slots are time frames where packets can be sent. Schedules can be fixed quantity or probabilistic. A fixed quantity scheduled slot sends a packet after a constant delay. A probabilistic scheduled slot sends a packet after a delay determined by a Poisson distribution. Probabilistic schedules can be calculated exactly by multiple parties through the use of a pseudo-random number generator seeded to the same value.

Schedules and slots allow OWAMP test packet receivers to calculate one-way delay and loss. The difference between a packets slot schedule send time and its receive time is a singleton one-way delay measurement. Mitigation of send time error and skipped packets is done through test packet control information. Loss is calculated through sequence numbers within the test packets. Upon completion of a test session singleton measurements can be retrieved.

OWAMP cannot emulate application protocols or data and cannot use TCP. Consequentely, it is not an effective tool when trying to understand how one-way loss and delay impact application utility.

2.6.6 Trajectory Sampling

Trajectory sampling[40] samples packets at different devices within a network. The samples can be used to determine flow route path and flow performance. A packet hash function is used to identify the same packet at different observation points. Packet hash functions and the packet equivalency functions (described in Section 3.1) are nearly identical concepts. In both, the goal is to make a unique packet signature from network, transport, and application fields. These fields must be chosen based on their likelihood of changing in transit and ability to discriminate different packets.

Zseby *et el.*[41] explains how packet hash functions can be used to measure one-way delay. They evaluate how different packet hash functions perform with respect to processing time and collision. One-way delay singletons are measured by collecting hashes at multiple observations points in the network. When a packet hash is collected the time of its collection is recorded. The difference in

collected time of two packets with the same packet hash is one-way delay. Singleton one-way loss measurements can be collected if a packet hash has less collisions than expected.

Trajectory Engine[42],impd4e[43], and OpenIMP[2] are all tools that utilize trajectory sampling concepts to collect various measurements. Tools based on trajectory sampling are used primarily for network engineering and thus sampling is preferred due to the volume of traffic. They can passively measure one-way delay and loss.

2.7 Problem Statement

Measuring networks is an important component toward understanding application utility. Application utility characterizes an application's usefulness. Objective analysis of application utility can identify and aid in then diagnosis of application problems. Network characteristics directly (through application protocol interaction) and indirectly (through transport, network, link, and physical layer protocol interactions) impact application utility. The relationship between network phenonema and application utility must be understood to predict application usefulness in untested networks, to identify areas for application improvement, and to generally improve network performance.

Accurate one-way delay and loss singleton measurements are fundamental to measuring networks. Virtually any other network metric can be derived from these two singleton metrics. It is common for these metrics to be approximated from traffic with known round-trip behavior like ICMP Echo messages (discussed in Section 2.6.1) and TCP segment acknowledgements (used by tcptrace, discussed in Section 2.6.2). These approximations are inherently error prone due to asymmetric network paths and queuing delay.

Simulation and active measurement are reasonable for approximating application utility in various network scenarios. Two common tools that can make such measurements are iperf and ITG (respectively discussed in Sections 2.6.3 and 2.6.4). Eventually, live tests on actual networks are required to understand exact application utility. Measuring one-way delay and loss singletons are critical for such tests. For any application, this can be done through application instrumentation. This approach has drawbacks, it must be done for every application that requires measurement and the instrumentation overhead may change application and network behavior.

In the past, accurate one-way delay and loss singleton measurements lacked accurate time synchronization. These measurements require infrastructure to synchronize clocks at multiple points of observation. Now, synchronization is widely available through clock information in GPS or CDMA; both are common in almost any mobile device. When such sources are unavailable, NTPv4[44] can provide time synchronization with bound error.

OpenIMP[2] is a passive, general network measurement infrastructure that can collect one-way

delay and loss measurements. It consists of N distributed Measurement Servers (impd), a Result Data Collector (impcol), and the OpenIMP Shell (impsh) that can be used to send commands to the distributed components and retrieve their return values. Data can be collected as raw files or using the IPFIX[45] protocol. Setting up and configuring OpenIMP is non-trivial.

pcapstitch is a lightweight passive, general network measurement tool that can collect one-way delay and loss measurements. Libpcap[18] is its only dependency and can be found easily for many platforms and operating systems. It outputs data in a simple human readable text files using a space delimited format. This format ensure that further processing by arbitrary applications is easy.

pcapstitch fills the gap left in the absence of a simple tool for passive, general network measurement. It has been used successfully to measure various network applications. It has few dependencies and relies on standard trace file libraries. pcapstitch is also a case study in building system tools using Haskell[46], a pure (*i.e.*, side effect free), strongly typed, programming language. pcapstitch contains an embedded domain specific language that parses network headers demonstrating a practical use for simulated dependent types. Most importantly, pcapstitch adheres to a number of the precepts from the UNIX philosophy, namely:

- Make each program do one thing well.
- Store data in flat text files.
- Use shell scripts to increase leverage and portability.
- Avoid captive user interfaces.

Chapter 3: pcapstitch

pcapstitch is a tool that collects singleton one-way delay and loss measurements. pcapstitch collects these measurements passively and requires no application instrumentation (*i.e.*, it can be used on general network traffic). pcapstitch associates semantically equivalent packets in trace files collected from multiple observation points. Packets must be associated in this way because segments can change in transit. The only other tool with this capability known to this author is OpenIMP[2] which is a comprehensive measurement suite. It can measure many different network characteristics both passively and actively. It relies on multiple independent software components. In contrast, pcapstitch has a very simple setup and is designed solely to collect singleton one-way delay and loss measurements from trace files. Simplicity, few dependencies, and operation consistent with the UNIX philosophy are advantages of using pcapstitch.

Chapter 2 provided the background necessary to understand relevant construction details, use cases, and practical concerns of pcapstitch. It provided the reader with some intuition about why one-way delay and loss is important for understanding application utility. In Section 2.6, related tools are discussed and in Section 2.7 the technological gap pcapstitch fills is exposed.

This chapter discusses pcapstitch in detail. Section 3.1 defines the terminology that is necessary for understanding the operation of pcapstitch and provides an abstract description of how pcapstitch collects measurements from trace files. Section 3.2 goes through a simple experiment using pcapstitch that describes pcapstitch invocation, heuristics for stitch horizon determination, and examples of how to use pcapstitch output in a unix environment. Construction of pcapstitch is discussed in Section 3.3; general program control flow, how stitching is implemented and pcapstitch's embedded domain-specific language for header parsing are the main technical issues addressed. Section 3.4 describes issues that should be considered when using pcapstitch. Finally 3.5 outlines the steps necessary to install pcapstitch.

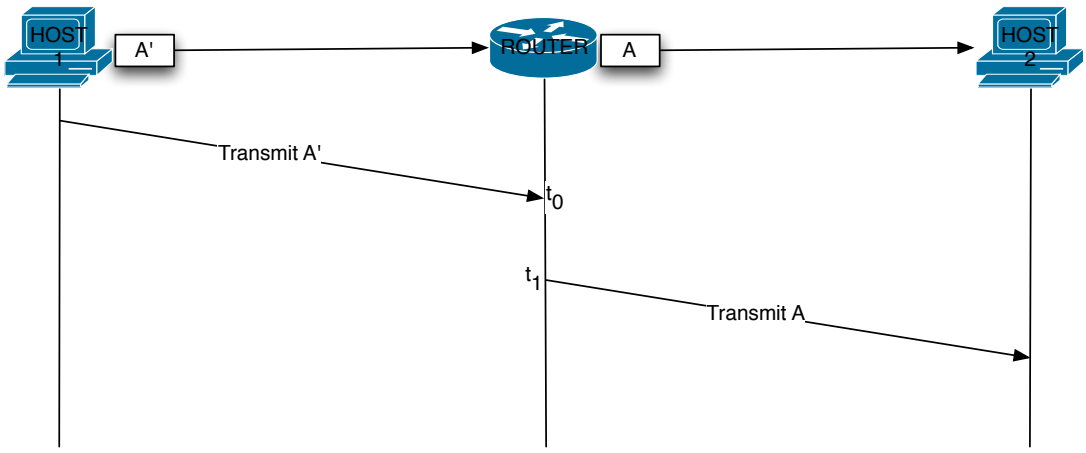

Figure 3.1: A simple network with three observation points, $Host_1$, *Router*, and $Host_2$. A packet A' is routed from $Host_1$ to $Host_2$ via *Router*. Between time t_0 and t_1, A' is processed by *Router*. This processing will change certain portions of A' creating A. A is a different packet when comparing bytes but equivalent when comparing meaning. That is, it represents the original packet A' sent from $Host_1$. Thus we say that A' and A are semantically equivalent packets.

3.1 Pcapstitch Overview

According to the cloc tool[47] pcapstitch contains just under five thousand lines of code (4892) throughout twenty-two different files. All of this code directly or indirectly supports the creation of a stitch. A stitch is a set of semantically equivalent packets, each from different observation points. Semantically equivalent packets have one point of origin within a network. Semantically equivalent packets is illustrated in Figure 3.1.

A Packet Equivalency Function (PEF) determines packet semantic equivalence. A PEF compares expected static structures from two packets. Referring again to Figure 3.1, an appropriate PEF would merge A' and A into a stitch. Throughout this document, merging means to place a packet from an observation point into a stitch's semantically equivalent set. If pcapstitch were run on two trace files collected at $Host_1$ and $Host_2$, given an appropriate PEF, a single stitch would be created by merging A' and A into that stitch's semantically equivalent set.

Protocol fields within a packet are used by a PEF. Understanding how networks might change these fields is key to constructing the right PEF. For instance in Figure 3.1, Ethernet source and

destination address could not be used because routers on the transmission path between a source and destination device will alter those fields. However, in Figure 3.3, Ethernet source and destination address could be used because the two hosts are directly connected on a LAN. pcapstitch contains two additional non-protocol field components that can be used in a PEF. One ensures that merges only take place between packets from different trace files and another will only allow merges between packets with equal application payload hashes. An application payload hash is a hash of the portion of data in the application layer of a packet.

pcapstitch creates stitches from one or more trace files collected at one or more observation points. Stitches are used to measure singleton one-way delay and loss. The pcapstitch output format maintains all recorded trace file times of merged packets in each stitch. Merge packet times are timestamps indicating when a particular packet was observed by a trace utility. The merge packet recorded times in a stitch allow measurement of:

- One-way latency through the difference of the earliest merged packet and the latest merged packet.
- One-way loss by noting if a stitch's semantically equivalent set contains less merged packets than expected.

The stitch creation process starts by reading a packet from a trace file. This packet is then checked for semantic equivalency against a list of stitches using a PEF. If a semantically equivalent packet is found in a stitch's previously merged packets, then the recently read packet is merged with that stitch. If no semantically equivalent packet is found in any stitches, the recently read packet is merged into an empty stitch.

Stitches are removed from the list according to the stitch horizon. The stitch horizon defines stitch list length by time. The latest merged packet in any stitch in the list minus the earliest merged packet in any stitch in the list defines the list time length. Earliest stitches are removed and outputted from the stitch list until the list time length is less than or equal to the stitch horizon. The stitch horizon operation is illustrated in Figure 3.2.

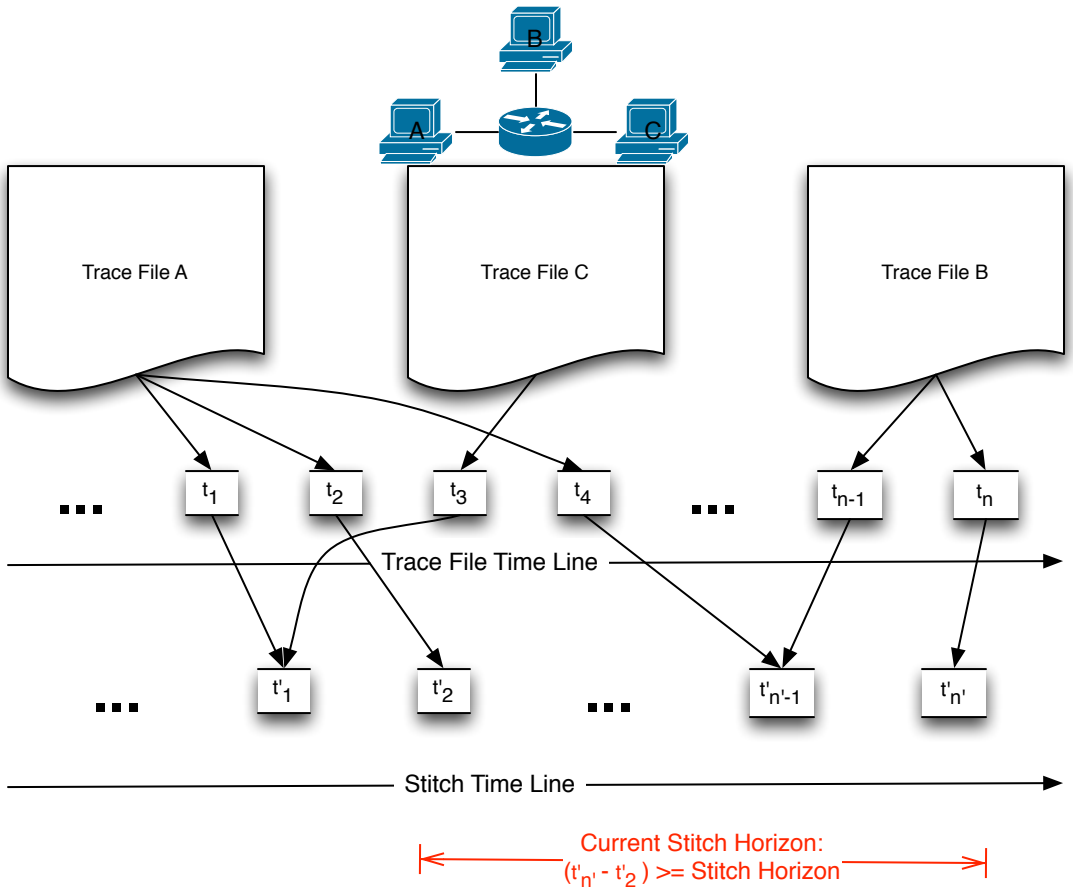

Figure 3.2: An illustration of stitch horizon operation. Packets are read in from multiple trace files. Each packets has a timestamp associated with it, $t_1, \ldots, t_n$ that indicates when the packet was recorded in the trace file. These packets are merged in chronological order and their timestamp changes to that of the latest merged packet, $t'_1, \ldots, t'_n$. This second queue (highlighted) consists is the stitch list. The stitch horizon are all stitches that satisfy the predicate $t'_n - t'_i < h$ where $i \leq n$ and h is the specified stitch horizon time.

A stitch horizon is necessary for an accurate measurement of loss. As discussed in Section 2.4.1, loss is equivalent to infinite delay. It is impossible to distinguish delay and loss using this definition for any finite time experiment. In practice, an upper bound time can be used to assume a delayed packet has been lost. A reasonable upper bound could be derived by finding the lowest throughput link in the network, calculating the time necessary to send the largest possible packet using the lowest throughput link, multiplying that time by the number of links in the network, multiplying the number of nodes in the network by some estimate on processing time, and summing these last two values. This values estimates the time it would take a packet to traverse every link and device in the network and can be used for the stitch horizon. Assuming a perfect PEF, a stitch horizon greater than any occurring packet delay will allow pcapstitch to measure loss accurately.

Perfect PEFs are generally not possible. Given enough packets, two packets will eventually be incorrectly identified as semantically equivalent regardless of the PEF. Packets that are incorrectly identified as semantically equivalent are referred to as collisions. The number of packets is not the primary cause of collisions. To simplify this discussion, the network is treated as a probabilistic process generating packets according to some distribution. Using this model, the likelihood of a collision increases over time and with maximum network throughput because more packets can be generated. Assuming uniqueness of packet fields within a time-frame can mitigate the risk of collisions. This is another function of stitch horizon.

Making the stitch horizon as short as possible can mitigate collisions. A stitch horizon that mitigates value repetition of the smallest field used by a PEF will reduce collision probability. An estimate for this time can be calculated using $fv/(l/8*1/m)$ where fv is the number of unique values of the field, l is the fastest link in the network, and m is the minimum packet size allowable. For very fast networks this value may be prohibitively small. In these cases it is important to remember that multiple fields from the PEF must collude to cause collision. Therefore, this estimate will likely be conservative.

Shorter stitch horizons bound memory usage and improve performance. Trace files can be large.

A trace file recorded on a $100Mbps$ link at full utilization for an hour can reach $40GB$. Setting a stitch horizon of an hour when running pcapstitch on this trace file will cause the stitch list to possibly contain $40GB$ of data. pcapstitch **does not** store entire packet payload in the stitch list, it only stores control information and application payload hashes. However, it is possible that this $40GB$ trace file consists entirely of minimum size packets with no application payload. Under these conditions the stitch list would still approach $40GB$ in size. More data in a list generally means more packets in a list. More packets in a list means more PEF comparison on every new packet read from a trace file. This increases the amount of time it takes pcapstitch to complete. In general, a stitch horizon should be set as long as possible given memory requirements, performance requirements, and PEF collision fragility such that a delayed packet will never be erroneously calculated as lost.

pcapstitch's usefulness with respect to the state of the art is described in Section 2.7. The stitch horizon is why pcapstitch is simple, fast, and accurate. Conceptually, the stitch horizon and the embedded domain-specific language for parsing network headers are the most technically interesting pieces of pcapstitch.

3.2 Operation Demonstration

pcapstitch is best demonstrated through an experiment. This experiment will determine one-way median delay between two nodes in a network from passive application traffic. This demonstration will provide:

- An overview of how to use pcapstitch
- Intuition for choosing the correct pcapstitch options
- Examples of how to generate formatted results from pcapstitch using common unix utilities

This experiment will be run on the two node barbell network illustrated in Figure 3.3. This network is an "experiment" running on the University of Utah's Emulab System[48]. Emulab is a network testbed that allows for actual networks to be instantiated through NS2[49] topology configuration

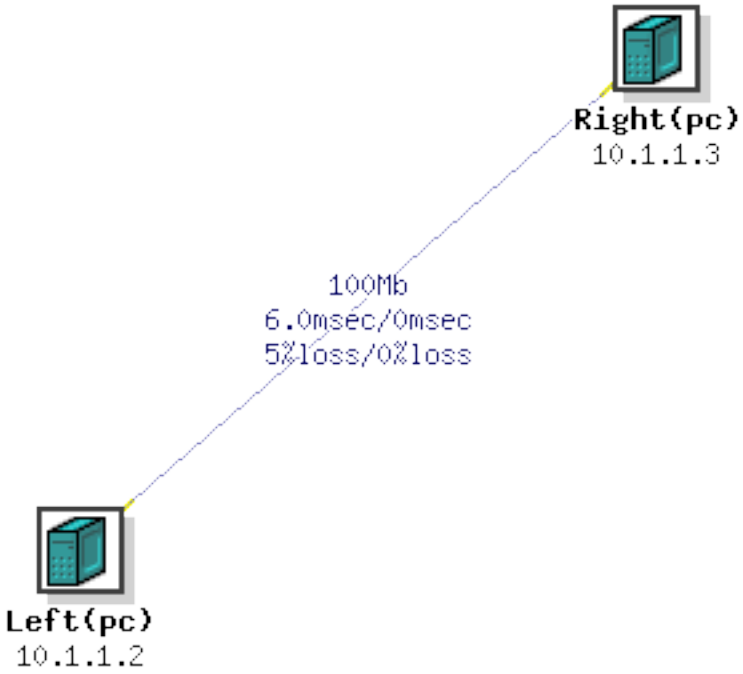

Figure 3.3: A two node barbell network that our experiment is being run on. The link has a throughput of $100Mbps$. Traffic moving from the right node to the left node will suffer an additional (with respect to than transmission and queuing delay) $6ms$ delay and an additional packet loss rate of 5% (this will be above any packet losses that may be caused by a queue overrun). Traffic moving from the left node to the right node will suffer no additional delay (other than transmission and queuing delay) and no additional loss (other than any that might be caused by a queue overrun).

files. Link conditions, network devices (routers and switches), and host operating systems can all be specified. When swapped into Emulab, actual machines are allocated and the network topology is controlled through sophisticated virtual LAN setups. Link conditions are emulated using DummyNet[50].

In this network both nodes are considered observation points, that is, each node records sent and received network traffic in a trace file. Both nodes are running RedHat Linux 9.0. Consistent with Figure 3.3, we will refer to the two nodes (observation points) as left and right. Application traffic will be generated using Secure CoPy (scp), an application available through the OpenSSH Suite[51]. scp allows for data to be transferred between two host over a network. scp is secure because it encrypts this data while in transit.

The experiment will use scp to move a $20MB$ file from left to right. A random file can quickly be generate via `dd if=/dev/urandom of=/tmp/random.data bs=1M count=20`. During the transfer, trace files from the two nodes will be generated, therefore trace utilities on both nodes should be started before performing the experiment. This can be done by `sudo tcpdump -w /tmp/trace.pcap -s 0 -i eth3` on both nodes. The `-s 0` directs tcpdump to collect all data from the packet. If this is excluded, application payload may be truncated in the trace files. Once the trace utilities are running on both nodes, the experiment can be started using `scp /tmp/random.data right:/tmp/`. After the transfer is complete, both trace utilities can be stopped.

Trace files from the experiment can now be processed by pcapstitch. pcapstitch will stitch packets sent from one node with packets received at the other node. As pcapstitch is processing, uniformly formatted output will be generated. For all stitches, each merged packet has the time it was observed at different observation points. As noted earlier, this allows singleton one-way delay measurements from each stitch. Stitches are parameterized by the stitch horizon and the packet equivalency function, both are options for pcapstitch.

Three options can be specified at invocation to modify how stitches are created:

1. The stitch horizon time specified with `-H`

2. The maximum stitch count specified with `-m`.

3. The packet equivalency function specified with `-p`.

pcapstitch can be invoked without any specified options by executing `pcapstitch` $pcapfile_1$ $pcapfile_2$ ... $pcapfile_n$ where $pcapfile_1$, $pcapfile_2$, ..., $pcapfile_n$ are trace files. This will set a default stitch horizon time of 30 seconds and a maximum stitch count of 2. The default PEF will use network and transport source and destination addressing information, network ID, network offset, application payload hashes, and will ensure that all merges occur between packets from different trace files.

The maximum stitch count specifies the maximum number of merges a stitch can have. When a stitch has a packet merge count equal to the maximum stitch count new packets cannot be merged with it. This can be exploited to improve performance; stitches with a merge count greater than or equal to the maximum stitch count need not be compared for potential merging with the PEF. Given the network in Figure 3.3, the maximum packet stitch count would be two. Maximum stitch count can be modified by using `-m x` where `x` is a positive integer greater than 0. If 1 is used all stitches will only have one merge.

Stitch horizon specifies how long a stitch can be merged with newly read packets from trace files. Figure 3.2 illustrates stitch horizon operation. The link in figure 3.3 is an Ethernet link with a throughput rate of $100Mbps$. Ethernet's maximum frame length is 1518 bytes, minimum transmission time will be approximately $0.130ms = (1518 * 8)/(10^8)$. Therfore, $0.130ms$ is the minimum stitch horizon time for this experiment. Anything less is likely to generate stitches with a merge count of one because any mergable packet will be outside the stitch horizon upon reading the next packet. This calculation only accounts for transmission delay. In Section 2.4.1 components of delay are discussed; specifically, processing delay, queuing delay, transmission delay and propagation delay. The stitch horizon must account for all of these delays and PEF collision fragality. The maximum segment length of Ethernet is 100 meters. This experiment has one segment, assuming propagation at the speed of light makes our propagation delay less than a microsecond. Thus, propagation delay is estimated to be 0. Queue delay can be estimated using the NIC transmission queue size. Reception queueing delay can be ignored because devices in this experiment can handle traffic as fast as the network can deliver it. On left and right the transmission queue can grow to

100 packets making maximum queuing delay $13ms$ ($0.130ms * 100$). Finally, we add $6ms$ because of the induced link delay and get $20ms = ceil(13ms + 0.130ms + 6)$. An alternative to analytically deriving the stitch horizon, which requires intimate knowledge of the network, would be to derive it empirically before or after the experiment with ping. Running `ping right` reports a RTT of $6.19ms$, adding this to previously calculated propagation and queueing delay results in $20ms = ceil(6.19ms + 0.130ms + 13ms)$

A stitch horizon of $20ms$ needs be shown as less than $fv/(l/8*1/m)$. This will provide confidence that PEF collisions will not occur in the $20ms$ stitch horizon. The 2 byte network ID field will be used to measure PEF collision fragility. Most modern operating systems increment the IPv4 ID field with every packet. When fragementation is required, two packet fragments may have the same ID but the IPv4 offset field will differ (the IPv4 offset field is also included in the default PEF). Choosing only the network ID field is conservative; if the time window for uniqueness is greater than $20ms$ confidence with respect to few collision will be improved. Therefore, $fv = 65536$, $l = 100Mbps$ and $m = 66$(Minimum Ethernet frame size = 12[gap]+8[preamble]+14[header]+4[trailer], Minimum IPv4 packet size = 20, Minimum Transport segement size [UDP] = 8). Based on this information, $346ms$ is expected before two packets with the same network ID are sent. The current $20ms$ stitch horizon holds because it is less than $346ms$. Stitch horizon time can be modified by using `-H x`(where `x` is a positive real number) on the command line. In the current pcapstitch revision, stitch horizon must have a precision greater than or equal to a micro-second. In this experiment a stitch horizon of $20ms$ can be set with the -H 0.02 option.

A packet equivalency function (PEF) is a function that tests if two packets are semantically equivalent. The PEF can be modified using `-p s` where `s` is a string that describes the PEF. The PEF is specified in a syntax referring to protocol layer fields and composed using `+`. Possible layer fields are:

- *Link.Source* - Link layer source address
- *Link.Destination* - Link layer destination address
- *Link.Payload* - The protocol being carried by the link layer

- *Net.Source* - Network layer source address
- *Net.Destination* - Network layer destination address
- *Net.ID* - Network layer ID
- *Net.Offset* - Network layer fragementation offset
- *Net.Payload* - The protocol being carried by the network layer
- *Trans.Source* - Transport layer source address
- *Trans.Destination* - Transport layer destination address
- *FileMatch* - Packets will only be merged if they did not come from the same trace file
- *App* - Hash of the data in the transport layers payload.

This experiment uses a PEF of `FileMatch+Link.Payload+Net.Payload+Net.Source+Net.Destination+Net.ID+Net.Offset+Trans.Source+Trans.Destination+App`. This is the default pcapstitch PEF and specifies that the PEF will use Network and Transport source and destination addressing information, Network ID, Network Offset, application payload hashes, and will ensure that all merges occur between packets from different trace files. The comparisons will occur in the order specified from left to right.

The current revision of pcapstitch requires that trace files be in a libpcap[18] format. Trace files should be captured from obversation points that are time synchronized. Stitches from unsynchronized trace files can cause incorrect results due to:

- A stitch leaving the stitch list too early causing a subsequent semantically equivalent packet to not be merged
- A stitch entering the stitch list too early causing a PEF collisions
- One-way delay measurement error

Consider two packets that should be merged and a horizon time set appropriately, given the first noted case, these two packets will be removed from the stitch horizon without being merged. Consider three packets where two of those packets should be merged and a horizon time set to avoid PEF collisions, given the second noted case, time synchronization inaccuracy can cause collisions. The third case is obvious; one-way delay is calculated by the difference in time recorded at two different observation points. Sensitivity to the time synchronization requirement can be managed with proper parameters based on network detail specifics. This experiment uses NTPv4[52] for time synchronization. NTPv4 over a LAN can be expected to achieve better than $5ms$ accuracy for time synchronization[44]. The stitch horizon option should be adjusted to account for this by setting a stitch horizon of $25ms$ via *-H* 0.025.

pcapstitch outputs results in a space delimited text file. Figure 3.4 gives a sample of the output from pcapstitch when run on trace files collected from this experiment. The output columns are described as:

1. Flow ID - This is a unique integer associated with a quintuple containing network source address, network destination address, transport protocol, transport source address and transport destination address. This ID will be associate with traffic bi-directionally by defining the quintuple as $(src_l, dst_l, proto, src_t, dest_t)$ where:
    - src_l is the greater of the source and destination network address when compared in integral form
    - dst_l is the lesser of the source and destination network address when compared in integral form
    - $proto$ is the transport layer protocol
    - src_t is the transport source address if the network source address is greater than the network destination address and the transport destination address otherwise
    - dst_t is the transport destination address if the network source address is greater than the network destination address and the transport source address otherwise

```
0 42 Ethernet 00:02:b3:3f:75:b1 ff:ff:ff:ff:ff:ff UnknownLink NoAddress NoAddress UnknownNetwork NoAddress NoAddress (1302656911.867952,left.pcap,1)
0 60 Ethernet 00:02:b3:3f:75:b1 ff:ff:ff:ff:ff:ff UnknownLink NoAddress NoAddress UnknownNetwork NoAddress NoAddress (1302656911.868160,right.pcap,1)
0 42 Ethernet 00:02:b3:23:79:15 00:02:b3:3f:75:b1 UnknownLink NoAddress NoAddress UnknownNetwork NoAddress NoAddress (1302656911.868207,right.pcap,2)
0 60 Ethernet 00:02:b3:23:79:15 00:02:b3:3f:75:b1 UnknownLink NoAddress NoAddress UnknownNetwork NoAddress NoAddress (1302656911.874159,left.pcap,2)
1 74 Ethernet 00:02:b3:3f:75:b1 00:02:b3:23:79:15 IPv4 10.1.1.2 10.1.1.3 TCP 32817 22 (1302656911.874178,left.pcap,3)|(1302656911.874339,right.pcap,3)
1 74 Ethernet 00:02:b3:23:79:15 00:02:b3:3f:75:b1 IPv4 10.1.1.3 10.1.1.2 TCP 22 32817 (1302656911.874413,right.pcap,4)|(1302656911.880432,left.pcap,4)
1 66 Ethernet 00:02:b3:3f:75:b1 00:02:b3:23:79:15 IPv4 10.1.1.2 10.1.1.3 TCP 32817 22 (1302656911.880486,left.pcap,5)|(1302656911.880644,right.pcap,5)
1 89 Ethernet 00:02:b3:23:79:15 00:02:b3:3f:75:b1 IPv4 10.1.1.3 10.1.1.2 TCP 22 32817 (1302656911.882381,right.pcap,6)|(1302656911.888414,left.pcap,6)
1 66 Ethernet 00:02:b3:3f:75:b1 00:02:b3:23:79:15 IPv4 10.1.1.2 10.1.1.3 TCP 32817 22 (1302656911.888451,left.pcap,7)|(1302656911.888623,right.pcap,7)
1 88 Ethernet 00:02:b3:3f:75:b1 00:02:b3:23:79:15 IPv4 10.1.1.2 10.1.1.3 TCP 32817 22 (1302656911.902846,left.pcap,8)|(1302656911.903021,right.pcap,8)
1 66 Ethernet 00:02:b3:23:79:15 00:02:b3:3f:75:b1 IPv4 10.1.1.3 10.1.1.2 TCP 22 32817 (1302656911.903055,right.pcap,9)|(1302656911.909062,left.pcap,9)
1 610 Ethernet 00:02:b3:23:79:15 00:02:b3:3f:75:b1 IPv4 10.1.1.3 10.1.1.2 TCP 22 32817 (1302656911.905254,right.pcap,10)|(1302656911.912512,left.pcap,11)
1 610 Ethernet 00:02:b3:3f:75:b1 00:02:b3:23:79:15 IPv4 10.1.1.2 10.1.1.3 TCP 32817 22 (1302656911.909098,left.pcap,10)|(1302656911.910523,right.pcap,11)
1 66 Ethernet 00:02:b3:3f:75:b1 00:02:b3:23:79:15 IPv4 10.1.1.2 10.1.1.3 TCP 32817 22 (1302656911.943752,left.pcap,12)|(1302656911.943916,right.pcap,12)
1 66 Ethernet 00:02:b3:23:79:15 00:02:b3:3f:75:b1 IPv4 10.1.1.3 10.1.1.2 TCP 22 32817 (1302656911.943957,right.pcap,13)|(1302656911.949951,left.pcap,13)
1 90 Ethernet 00:02:b3:3f:75:b1 00:02:b3:23:79:15 IPv4 10.1.1.2 10.1.1.3 TCP 32817 22 (1302656911.949973,left.pcap,14)|(1302656911.950137,right.pcap,14)
1 490 Ethernet 00:02:b3:23:79:15 00:02:b3:3f:75:b1 IPv4 10.1.1.3 10.1.1.2 TCP 22 32817 (1302656911.957247,right.pcap,15)|(1302656911.964447,left.pcap,15)
1 66 Ethernet 00:02:b3:3f:75:b1 00:02:b3:23:79:15 IPv4 10.1.1.2 10.1.1.3 TCP 32817 22 (1302656911.964467,left.pcap,16)|(1302656911.964634,right.pcap,16)
1 482 Ethernet 00:02:b3:3f:75:b1 00:02:b3:23:79:15 IPv4 10.1.1.2 10.1.1.3 TCP 32817 22 (1302656912.005890,left.pcap,17)|(1302656912.007267,right.pcap,17)
1 66 Ethernet 00:02:b3:23:79:15 00:02:b3:3f:75:b1 IPv4 10.1.1.3 10.1.1.2 TCP 22 32817 (1302656912.039600,right.pcap,18)|(1302656912.045595,left.pcap,18)
1 802 Ethernet 00:02:b3:23:79:15 00:02:b3:3f:75:b1 IPv4 10.1.1.3 10.1.1.2 TCP 22 32817 (1302656912.070443,right.pcap,19)|(1302656912.077701,left.pcap,19)
1 66 Ethernet 00:02:b3:3f:75:b1 00:02:b3:23:79:15 IPv4 10.1.1.2 10.1.1.3 TCP 32817 22 (1302656912.077717,left.pcap,20)|(1302656912.077885,right.pcap,20)
1 82 Ethernet 00:02:b3:3f:75:b1 00:02:b3:23:79:15 IPv4 10.1.1.2 10.1.1.3 TCP 32817 22 (1302656913.569964,left.pcap,21)|(1302656913.570157,right.pcap,21)
1 66 Ethernet 00:02:b3:23:79:15 00:02:b3:3f:75:b1 IPv4 10.1.1.3 10.1.1.2 TCP 22 32817 (1302656913.570190,right.pcap,22)|(1302656913.576203,left.pcap,22)
1 114 Ethernet 00:02:b3:3f:75:b1 00:02:b3:23:79:15 IPv4 10.1.1.2 10.1.1.3 TCP 32817 22 (1302656913.576241,left.pcap,23)|(1302656913.576415,right.pcap,23)
1 66 Ethernet 00:02:b3:23:79:15 00:02:b3:3f:75:b1 IPv4 10.1.1.3 10.1.1.2 TCP 22 32817 (1302656913.576431,right.pcap,24)|(1302656913.582372,left.pcap,24)
1 114 Ethernet 00:02:b3:23:79:15 00:02:b3:3f:75:b1 IPv4 10.1.1.3 10.1.1.2 TCP 22 32817 (1302656913.576604,right.pcap,25)|(1302656913.582581,left.pcap,25)
1 66 Ethernet 00:02:b3:3f:75:b1 00:02:b3:23:79:15 IPv4 10.1.1.2 10.1.1.3 TCP 32817 22 (1302656913.582599,left.pcap,26)|(1302656913.582763,right.pcap,26)
1 130 Ethernet 00:02:b3:3f:75:b1 00:02:b3:23:79:15 IPv4 10.1.1.2 10.1.1.3 TCP 32817 22 (1302656913.582755,left.pcap,27)|(1302656913.584000,right.pcap,27)
1 146 Ethernet 00:02:b3:23:79:15 00:02:b3:3f:75:b1 IPv4 10.1.1.3 10.1.1.2 TCP 22 32817 (1302656913.588693,right.pcap,28)|(1302656913.595718,left.pcap,28)
1 434 Ethernet 00:02:b3:3f:75:b1 00:02:b3:23:79:15 IPv4 10.1.1.2 10.1.1.3 TCP 32817 22 (1302656913.595885,left.pcap,29)|(1302656913.596187,right.pcap,29)
1 386 Ethernet 00:02:b3:23:79:15 00:02:b3:3f:75:b1 IPv4 10.1.1.3 10.1.1.2 TCP 22 32817 (1302656913.604066,right.pcap,30)|(1302656913.610157,left.pcap,30)
1 66 Ethernet 00:02:b3:3f:75:b1 00:02:b3:23:79:15 IPv4 10.1.1.2 10.1.1.3 TCP 32817 22 (1302656913.643766,left.pcap,31)|(1302656913.643939,right.pcap,31)
1 706 Ethernet 00:02:b3:3f:75:b1 00:02:b3:23:79:15 IPv4 10.1.1.2 10.1.1.3 TCP 32817 22 (1302656913.667146,left.pcap,32)|(1302656913.668595,right.pcap,32)
1 98 Ethernet 00:02:b3:23:79:15 00:02:b3:3f:75:b1 IPv4 10.1.1.3 10.1.1.2 TCP 22 32817 (1302656913.679190,right.pcap,33)|(1302656913.685213,left.pcap,33)
1 66 Ethernet 00:02:b3:3f:75:b1 00:02:b3:23:79:15 IPv4 10.1.1.2 10.1.1.3 TCP 32817 22 (1302656913.685259,left.pcap,34)|(1302656913.685438,right.pcap,34)
1 130 Ethernet 00:02:b3:3f:75:b1 00:02:b3:23:79:15 IPv4 10.1.1.2 10.1.1.3 TCP 32817 22 (1302656913.685517,left.pcap,35)|(1302656913.686763,right.pcap,35)
1 114 Ethernet 00:02:b3:23:79:15 00:02:b3:3f:75:b1 IPv4 10.1.1.3 10.1.1.2 TCP 22 32817 (1302656913.687231,right.pcap,36)|(1302656913.693186,left.pcap,36)
1 130 Ethernet 00:02:b3:3f:75:b1 00:02:b3:23:79:15 IPv4 10.1.1.2 10.1.1.3 TCP 32817 22 (1302656913.693368,left.pcap,37)|(1302656913.693557,right.pcap,37)
1 114 Ethernet 00:02:b3:23:79:15 00:02:b3:3f:75:b1 IPv4 10.1.1.3 10.1.1.2 TCP 22 32817 (1302656913.703685,right.pcap,38)|(1302656913.709650,left.pcap,38)
1 66 Ethernet 00:02:b3:3f:75:b1 00:02:b3:23:79:15 IPv4 10.1.1.2 10.1.1.3 TCP 32817 22 (1302656913.743750,left.pcap,39)|(1302656913.743930,right.pcap,39)
1 114 Ethernet 00:02:b3:23:79:15 00:02:b3:3f:75:b1 IPv4 10.1.1.3 10.1.1.2 TCP 22 32817 (1302656913.803048,right.pcap,40)
```

Figure 3.4: Output from running pcapstitch on trace files collect during this experiment. The lines that have values of *UnKnownLink*, *UnknownNetwork*, and *NoAddress* indicate packets with headers that pcapstitch cannot parse.. For this experiment they are Address Resolution Protocol(ARP) packets.. Packets that contain unhandled protocols will have those column values and will never have a stitch count of more than one.

2. Wire length - This is size in bytes of the packet at the link layer.

3. Link Layer Type - The protocol used at the link layer.

4. Link layer Source Address - The link layer source address.

5. Link Layer Destination Address - The link layer destination address.

6. Network Layer Type - The protocol used at the network layer.

7. Network Layer Source Address - The network layer source address.

8. Network Layer Destination Address - The network layer destination address.

9. Transport Layer Type - The protocol used at the transport layer.

10. Transport Layer Source Address - The transport layer source address.

11. Transport Layer Destination Address - The transport layer destination address.

12. Merge Records - Each merge is separated by `|`. Each merge record is a triple containing time of packet recording, trace file that the packet was recorded in, and offset of the packet within that trace file (starting at 1). The merge records are chronologically order from left to right.

This text format makes processing and analyzing within a unix environment straightforward. Figure 3.5 shows a script that converts the pcapstitch output from this experiment into a gnuplot[53] graph of one-way latencies. One-way singleton delay is calculated by subtracting the earliest packet record time and latest packet record time from a stitch. In this experiment, a lost packet would only have one merge record.

Figure 3.6 shows one-way delay consistent with the experiment topology and analytic propagation delay. The right to left traffic indicates a one-delay around $6ms$ which is the additional delay specified in the Emulab topology. The left to right traffic shows a number of one-way delay measurements near $0.100ms$ which is consistent with the analytic calculation for propagation delay of $0.130ms$.

Figure 3.7 demonstrates how aggregate packet loss rate can be calculated. If this script was stored as `plr.sh`, running it twice via `grep '10.1.1.2 10.1.1.3' /tmp/scp-output.stitch |`

```
#!/bin/bash
TFILE=‘tempfile‘
I=‘cat $1 | sed -e "s/|\|,/ /g" -e "s/(\|)//g" \
| awk ’BEGIN{i=0;}{if(i==0){i=$12}}END{print i;}’‘
cat $1 | grep ’10.1.1.2 10.1.1.3’ |\
sed -e "s/|\|,/ /g" -e "s/(\|)//g" |\
awk -v i=${I} ’
BEGIN{init=i;
print "set xlabel \"Experiment Time(s)\"";
print "set ylabel \"One-way delay(s)\"";
print "plot \"-\" using 1:2 title \"Left->Right\" \
with lines,\"-\" using 1:2 title \"Right->Left\" with lines";}
{
  if(NF>14){printf("\t%f %f\n",$12-init,$15-$12);}
}END{print "end";}’ > ${TFILE}
cat $1 | grep ’10.1.1.3 10.1.1.2’ |\
sed -e "s/|\|,/ /g" -e "s/(\|)//g" |\
awk -v i=${I} ’
BEGIN{init=i;}
{
  if(NF>14){printf("\t%f %f\n",$12-init,$15-$12);}
}END{print "end";}’ >> ${TFILE}
cat ${TFILE} | gnuplot  -persist
rm \${TFILE}
```

Figure 3.5: A bash script to convert the data presented in figure 3.4 into a one-way latency graph.

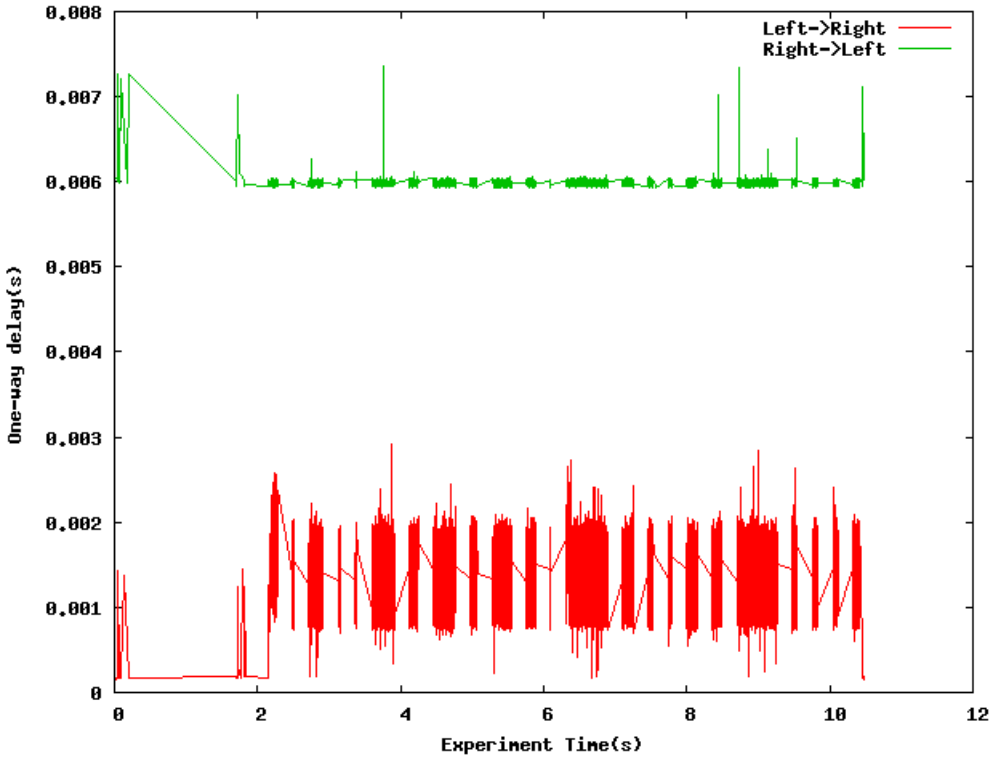

Figure 3.6: The graph produced by running Figure 3.5 on the pcapstitch output. Figure 3.4 shows a small sample of that output.

```
#!/bin/bash
grep -v UnknownLink | sed -e "s/|\|,/ /g" -e "s/(\|)//g" |\
awk '
BEGIN{}
{
 if(NF>14){gcount=gcount+1;}
 else{bcount=bcount+1;}
}END{
 print bcount/(gcount+bcount);
}'
```

Figure 3.7: A bash script that will calculate aggregate one-way packet loss rates from pcapstitch output.

`/tmp/plr.sh` and `grep '10.1.1.3 10.1.1.2' /tmp/scp-output.stitch | /tmp/plr.sh` would calculate the aggregate packet loss rate in both directions. Those results are 0% for left to right traffic and 0.049208% for right to left traffic. These values are consistent with the network topology defined in Figure 3.3.

This section started with a demonstration experiment to measure one-way median delay on a network illustrated in Figure 3.3. Figure 3.8 shows a script to collect this measurement. If this script was stored as `median-delay.sh`, running it twice via `grep '10.1.1.2 10.1.1.3' stitchfile.out | median-delay.sh` and `grep '10.1.1.3 10.1.1.2' stitchfile.out | median-delay.sh` would calculate the one-way delay in both directions. The result of running this script on the output in Figure 3.4 returns a median one-way delay of $0.001333s$ for traffic sent from left to right and $0.00598693s$ for traffic sent from right to left. NTP reported time synchronization accuracy should be used to find a range for each of these values. Both nodes listed an accuracy of at least $0.300ms$. Therefore, true one-way median delay is between $0.001033s$ and $0.001633s$ for left to right traffic and between $0.00568693s$ and $0.00628693s$ for right to left traffic.

3.3 Construction of pcapstitch

pcapstitch was tested and built on a Linux[54] x86 environment using the Haskell[46] programming language. The most recent version of pcapstitch is compiled with version 6.12.3 of the Glasgow Haskell Compiler (GHC) [55]. Detailed information about how to install pcapstitch can be found in

```
#!/bin/bash
sed -e "s/|\|,/ /g" -e "s/(\|)//g" |\
awk '
BEGIN{count=1;}
{
 if(NF>14){values[count]=$15-$12;count=count+1;}
}END{
 asort(values);
 if(count%2==0){o=(values[count/2]+values[count/2+1])/2;}
 else{o=values[(count-1)/2+1];}
 print o
}'
```

Figure 3.8: A bash script that will calculate one-way median delay from pcapstitch output.

Section 3.5.

Haskell is a static, pure, functional, type-safe, lazy evaluated programming language. Haskell is statically typed and static; all typing information must be known (or must permit inferencing) during compilation and it is not interpreted. Haskell's purity highly discourages side-effects. Common imperative mechanism such as variable mutation are not available. Side-effects can occur through a set of unsafe operations or using monads; the former option is highly discouraged. Mathematically, monads are constructs from category theory. In Haskell, monads allow the composition of computations and can be used to simulate side-effects. Haskell is functional because it is pure and because, in many cases, Haskell computation can be carried out through lambda calculus (*i.e.*, substitution). These properties provide the basis for a strong type-system that gaurantee's safety. Haskell's type systems is an extended version of the inferencing Hindley-Milner[56] type system. It supports polymorphism through type classes. Type classes offer an interface which types must implement to be included within a type class. In addition, it supports a number of type extensions that provide richer capability. Some of these extensions and capabilities are demonstrated in Section 3.3.2 which describes pcapstitch's embedded domain specific language for network header parsing. Haskell supports lazy evalution; values will not be computed until they are needed. This is opposite of how a strict language behaves. Consider the pseudo-code `x = 2 + f(); print x;` . A strict language will calculate the first statement before proceeding to the second statement. Haskell and other lazy

languages will store a deferred operation `2 + f()` as the value of `x`. Evaluation of `x` will not be forced until `print x` is called.

From a programming perspective, three components of pcapstitch make it's construction technically interesting:

1. pcapstitch was constructed in Haskell
2. pcapstitch's embedded domain specific language for header parsing
3. pcapstitch's process for stitching efficiently

This section will examine these three components. Components 2 and 3 will have dedicated subsections while component 1 will be discussed throughout this section. Section 3.3.1 gives an overview of pcapstitch construction. Section 3.3.2 explains pcapstitch's embedded domain specific language (item 2). Finally, Section 3.3.3 details how pcapstitch stitches packets efficiently (item 3).

Throughout this section, specific Haskell modules are referred to by name and italicized. With the exception of the *Main* module, pcapstitch modules can be identified with the prefix *Network.Pcapstitch*. The *Main* module is the pcapstitch entry point..

3.3.1 Construction Overview

A high-level pcapstitch control diagram is shown in Figure 3.9 and the software architecture is shown in Figure 3.10. Control flow starts when pcapstitch is invoked with command line arguments. Upon successfully parsing the command line options, pcapstitch enters its main processing loop. The processing loop reads, chrononologically, the next packet from the trace files, constructs a Message from the packet by parsing packet headers, removes and outputs any expired stitches from the stitch list according to the stitch horizon, and finally attempts to merge the Message somewhere into the stitch list. This section will go over the control points shown in Figure 3.9 in some detail and refer the reader to specific portions of the source code.

As discussed in section 3.1, pcapstitch is invoked with a list of trace files and optional command line flags. pcapstitch handles command line arguments according to [57] in *Network.PcapStitch.Options*. Flags are parsed from the command line using the *System.Console.GetOpt* module. Each possible

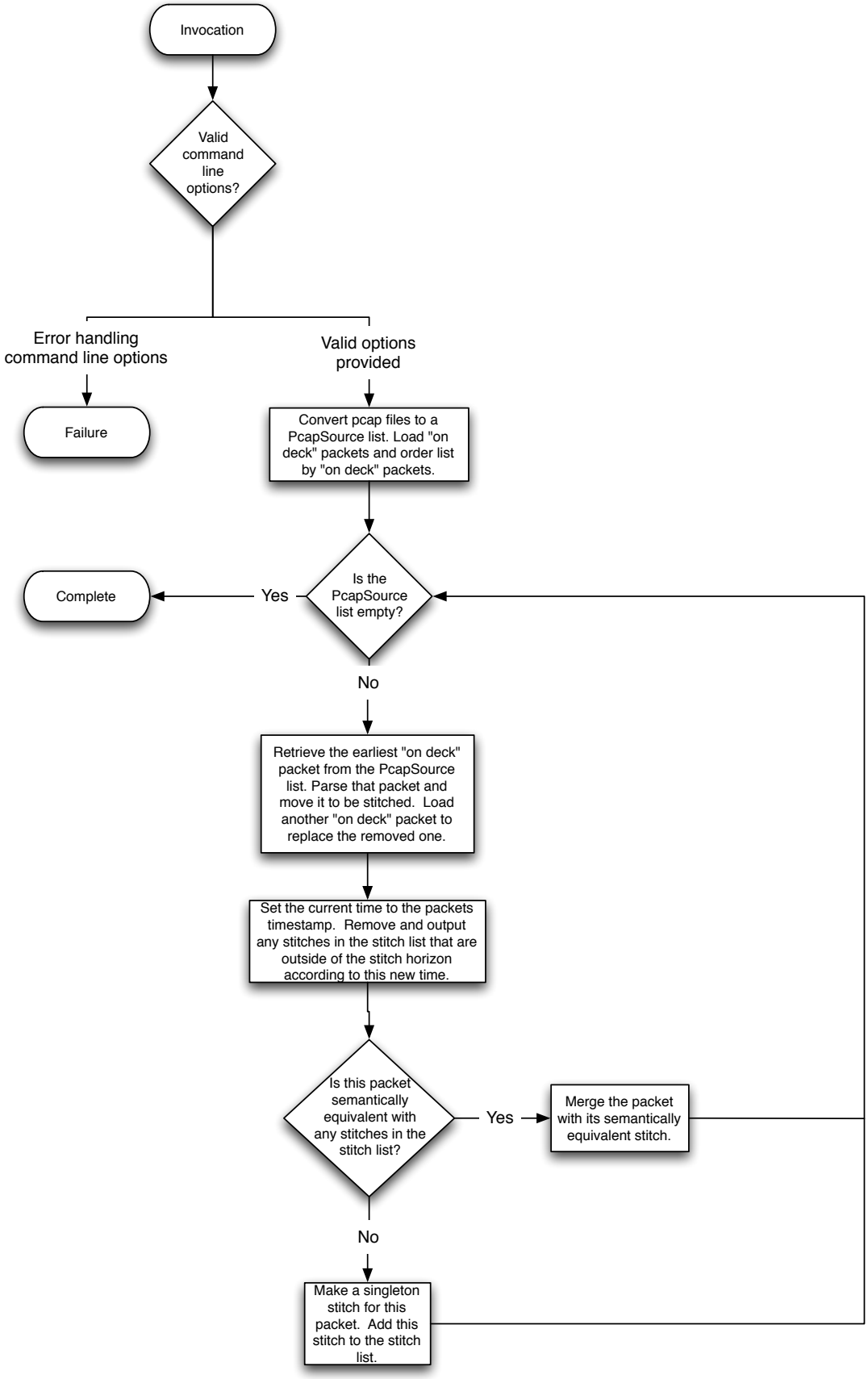

Figure 3.9: pcapstitch control diagram.

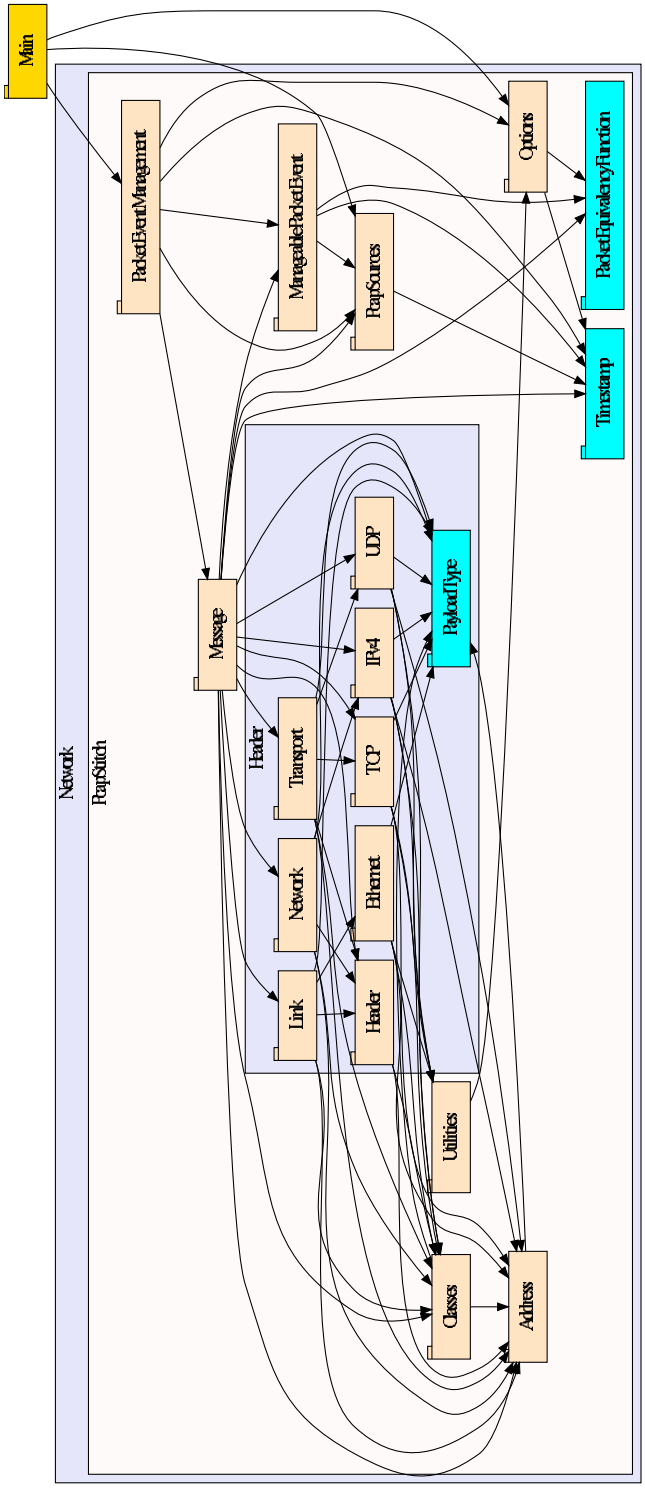

Figure 3.10: This high-level software architecture diagram shows the interaction between pcapstitch's major components. Yellow indicates an entry module and turquoise indicates a leaf module (*i.e.*, it depends on no other modules).

flag results in a lambda or a string documenting an error. This lambda list is threaded using monadic composition. In this way, parsing the command line is accomplished through a number of state transitions. The initial state is defined in *Network.Pcapstitch.Options* (refer to Section 3.1 for the default options). A completed options data structure is constructed by each lambda receiving an old option state, performing some operation according to a command line flag, and returning an new option state.

Network trace files are converted to PcapSource data structures within *Network.Pcapstitch.PcapSource*. Each PcapSource maintains four fields:

- Trace file name - The trace file name provided to pcapstitch through the command line.
- Trace file handle - The file handle provided by *Network.Pcap*.
- Packet count offset - The offset into the file handle, in packets. For instance, a packet count offset of 4 indicates that the next packet read would be the $5th$ packet in the file.
- The "On Deck" packet - An already extracted packet that has yet to be processed.

A PcapSource is created for each trace file; thus a list of PcapSources is the source of packets to merge.

Each PcapSource has one packet loaded "on deck", in other words, read from a trace file but not processed. Processing creates a data structure from a packet that is suitable for stitching. Loading "on deck" packets allows for chronological processing, a requirement for efficient stitching. Given two trace files, the next packet from both must be loaded to observe their timestamps for chronological processing. This assumes that packets are recorded in the trace file in chronological order. This assumption can be violated on occasion which results in loss of accuracy (discussed in Sub-Section 3.4.5).

The next packet to merge is selected from a PcapSource with the earliest "on deck" packet. The list of PcapSources is kept ordered according to their "on deck" packet timestamps. When an "on deck" packet is removed from a PcapSource, it is replaced by a new packet from that PcapSource's trace file, and that PcapSource is re-inserted into the list of PcapSources. This ensure that the

earliest "on deck" packet is always at the front of the list of PcapSources. When a PcapSource has no further packets it is closed and removed from the list of PcapSources. Thus, when the list of PcapSources is empty, no further packets can be read, pcapstitch can output all remaining stitches in the stitch list, and exit.

Processed "on deck" packets from the list of PcapSources are converted to Messages using *Network.PcapStitch.Message*. A Message is the implementation of a stitch. Thus all packets are converted to Messages that represent singleton stitches. The Message contains all information necessary for a PEF to evaluate semantic equivalence. It also contains a list of triples, that represent a merge. The triple contains the same content as a single merge record within a stitch outputted by pcapstitch (discussed in Section 3.2).

Processing the "on deck" packet is primarly focues on parsing the protocol layer headers. It is initiated by calling `getPacketEvent` from the ManageablePacketEvent type class (discussed shortly). `getPacketEvent` parses the network headers using *Network.PcapStitch.Header.Header*, *Network.PcapStitch.Header.Link*, *Network.PcapStitch.Header.Network*, *Network.PcapStitch.Header.Transport*, *qualified Network.PcapStitch.Header.TCP*, *qualified Network.PcapStitch.Header.UDP*, *qualified Network.PcapStitch.Header.IPv4*. These header parsing modules heavily rely on pcapstitch's embedded domain specific language for header parsing found in *Network.PcapStitch.HeaderNinja* and further discuessed in Section 3.3.2.

Messages implement the ManageablePacketEvent type class from *Network.PcapStitch.ManageablePacketEvent*. ManageablePacketEvent presents a uniform interface for stitching packets. The ManageablePacketEvent type class provides the following important operations:

1. Merge testing - Using the PEF to determine if two messages are semantically equivalent.

2. Equality testing - Determine if two Messages are identical. That is, the Message's are actually the same Message. This is different than semantically equivalence. In a impure language a pointer would be used to reference the same Message in different data structures. Haskell is pure, therefore every data value is a copy, resulting in multiple *identical* copies that reside in different data structures. Equality testing finds these identical copies.

3. A merge operator - Will merge one Message with another.

4. Output a Message - Will output a line conformant with the stitch output described in Section 3.2.

From the stitching perspective, all operation are done through this interface. In this way, the actual implementation of a stitch (currently a Message) can be altered or replaced.

ManageablePacketEvent stitching occurs in *Network.PcapStitch.PacketEventManagement*. For clarity, packets are all converted to Messages (singleton stitches), and Messages are instances of the ManageablePacketEvent type class which PacketEventManagement uses for stitching. When PacketEventManagment retrieves a ManageablePacketEvent, it checks for a semantically equivalent ManageablePacketEvent, using merge testing, within the stitch list according to the PEF. If a semantically equivalent ManageablePacketEvent is found, it is merged with its using the merge function. If no semantically equivalent ManageablePacketEvent is found, the ManageablePacketEvent is added to the stitch list. ManageablePacketEvents are purged and outputted when the stitch horizon dictates their removal from the stitch list.

3.3.2 Type Safe Header Parsing

Network header (subsequently referred to as headers) parsing is a common task in many different types of software. Network drivers, network monitoring tools, routers and switches all typically contain some software header parsing mechanisms. Parsing headers involves separating byte strings into fields and converting those fields to more appropriate types.

In practice, parsing headers can be a tedious task requiring boilerplate infrastructure. Incongruence between bytes, typically the smallest transmission parcel in networks, and fields specified in bits requires error prone bit manipulation. Additionally, the fields themselves need not be byte aligned but the sum of all the fields must be.

Network Header Parser (NHP) is a prototype library that addresses these issues. NHP is written in Haskell[46] using type-level programming, a flexible record infrastructure[58] and Template Haskell[59]. It reduces the amount of boilerplate code by allowing specifications of headers through

individual field combination. NHP automatically performs bit manipulation and type conversion guided by field declarations. Guaranteeing byte alignment in headers relies on dependent types[60] faked[61] in Haskell. In other words, if fields are combined in such a way that the combined length modulo eight does not equal zero the code will not compile. The parsed headers are returned in flexible records with a type specific to the header.

NHP is a prototype evolving embedded domain specific language to simplify and strengthen the safety of network header parsers. A user of this language constructs network header parsers by declaring fields and combining them. NHP is contain solely withing the *Network.PcapStitch.HeaderNinja* module within the pcapsitch.

Declaring Fields and Building Headers

A `Field` is a data structure that holds three values: field name, field data conversion function, and bit size. The field name is self-explanatory, however the name itself does require some initial setup due to the nature of the records package. The template function `makeName` reduces boilerplate code by generating field names automatically from a provided string.

The data conversion function converts a byte string to a desired data type. The byte string is only as long as the bit size of the ceiling of the bit size of the field divided by eight. The specific type signature of the data conversion function is `B.ByteString -> BitGet fieldtype`. b The bit size value specifies how long the field is in bits. Bit sizes are represented through a generalized algebraic data type[62] and have a number of convenience values as well as the ability to specify an arbitrary bit size. The `RequestFieldSize` data structure hides a type-level programming mechanism that uses Peano numerals[63]. This mechanism is used during header construction to ensure byte alignment at compile time via type class constraints. The template function `arbitraryBits` can be used to generate fields of arbitrary bit size, using it avoids building a verbose Peano numeral by hand. An example field specification can be found in Figure 3.11. `RequestFieldSize` (convenience constructors removed for brevity) and an example field specification follows:

Fields are combined using the infix, right associative `.++.` operator. This operator folds the fields from right to left to build headers. `headerTail` is an empty header that must be placed at

```
-- A view of how bit sizes are represented.
data RequestFieldSize a where
  RFSStop ::RequestFieldSize FZero
  RFSNBits::Peano b=>b -> RequestFieldSize a -> RequestFieldSize (Add a b)
-- An example field specification
$(makeName "FragOff")
bits = $(arbitraryBits 14)
fof = Field FragOff getbgword16  (RFSNBits bits)
```

Figure 3.11: A shortened view of the field size data structure and an example of how to construct a field.

```
ipv4header =
  verf.++.hlenf.++. dscpf.++.tlenf.++.ipidf.++.flagsf.++.fof
  .++.ttlf.++.protof.++.chksumf.++.sourcef.++.destf.++.headerTail
```

Figure 3.12: Constructing a header through field combination and how to use recordType and wrapInData.

the right most position of any field combination expression. `.++.` keeps a running bit length total of the growing header through the use of the `RequestFieldSize` data structure. Combining fields to form headers is demonstrated in Figure 3.12. An example of how fields are combined follows:

Parsing a Header from Byte String

Byte string are parsed with `getHeader`. Byte alignment is ensured by the `ProperByteAlignment` type class constraint using a type function that checks if the header bit length modulo eight equals zero. `getHeader` requires as input header and byte string. It returns a type-specific record based on the header. Errors are handled through the `Either` data structure. Fields are accessed by field name through the `!!!` operator. The types of these records can be unwieldy, however, using the `recordType` function and the template function `wrapInData` address this issue. The `recordType` function ties a header record type to a variable and `wrapInData` creates a data structure that contains the header record. The `getHeader` type signature and a full example of Ethernet header parsing follows:

Conclusions and Future Work

NHP is a prototype-embedded-domain-specific language for automatically generating network header parsers. It does so from network header specifications relying on Haskell's type system to ensure at

```
getHeader :: ProperFormedRequestField t a  => Header (t a) rec
-> B.ByteString -> Either String (rec (IdKindStar),B.ByteString)
getHeader = ...
$(makeName "Source")
sourcef = Field Source getbgword64 RFSWord48
$(makeName "Destination")
destf = Field Destination getbgword64 RFSWord48
$(makeName "Protocol")
protof = Field Protocol (liftM EthernetType . getbgword16) RFSWord16
ethheader =  destf .++. sourcef .++. protof .++. headerTail
ethheadertype = recordType ethheader
$(wrapInData "EthernetFrame" 'ethheadertype)
getEthHeader = getHeader ethheader
handleError = handleHeaderError "Error Parsing EthernetFrame"
getEthernetFrame :: B.ByteString -> (EthernetFrame,B.ByteString)
getEthernetFrame bs = (EthernetFrame rec,rest)
  where
    tuple = handleError . getEthHeader $ bs
    rec   = fst tuple
    rest  = snd tuple
source (EthernetFrame x) = x !!! Source
destination (EthernetFrame x) = x !!! Destination
payloadtype (EthernetFrame x) = x !!! Protocol
```

Figure 3.13: The type signature of the getHeader function.

compile time that headers are byte aligned. NHP demonstrates a practical use of a rich, type-safe, declarative language, namely Haskell. Future work includes a more elegant solution for handling optional fields within headers (NHP currently allows bolt on options that circumvent byte-alignment constraints), exploring how the record package can be used to enforce constraints between fields, making the data conversion input parameter more specific to the bit size of the field, and making the specification language more concise and readable.

3.3.3 Packet Stitching

Stitching is implemented in *Network.PcapStitch.PacketEventManagement* (referred to hereafter as PacketEventManagement). This module is show in Figure 3.14. PacketEventManagement has three primary concerns: accuracy, performance and memory usage. Section 3.1 describes how stitch horizon manages these concerns. In this section, the implementation details are shown.

Main initiates PacketEventManagement by calling `runPacketEventManager` with the list of PcapSources and options. This first step is to initialize the `PacketEventManagementMutableHash`

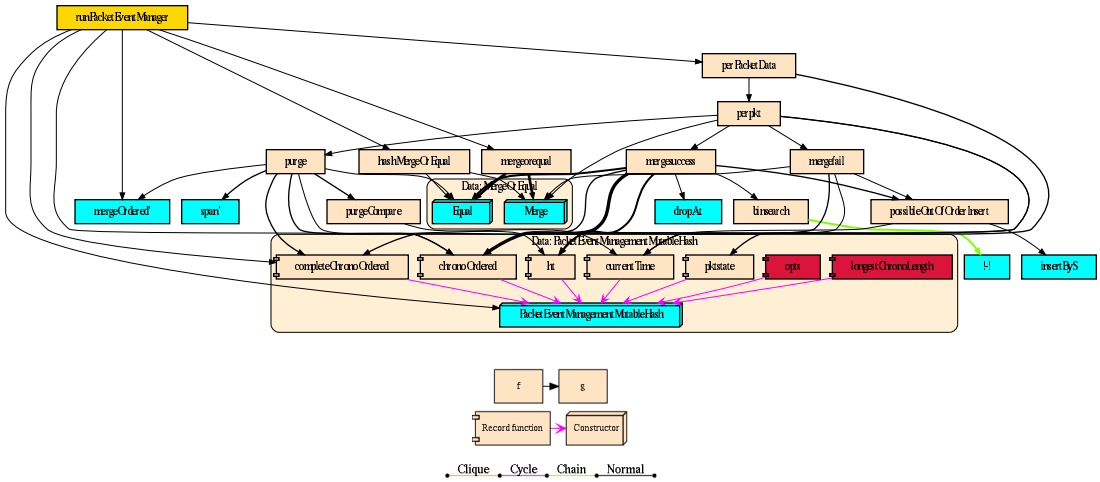

Figure 3.14: This is a visualization of PacketEventManagement. All figures following the first are legends. The second figure shows two normal functions where f calls g. The third figure shows data type declaration. The forth figure shows edge classification.

data structure. `PacketEventManagementMutableHash` contains all the data to keep the stitch list and the stitch horizon consistent. The following descriptions represent fields within `PacketEventManagementMutableHash` that are pertinent to stitching:

- Options - Contains the options provided to pcapstitch on invocation. This will not change once `PacketEventManagementMutableHash` has been initialized.
- Chronologically ordered ManageablePacketEvent list (COM)- This is the stitch list.
- Chronologically ordered Completed ManageablePacketEvent list (COCM) - A stitch list where all stitches have a merge count equal to the maximum stitch count.
- ManageablePacketEvent Hash Table (MHT) - A hash table that contains all the ManageablePacketEvents that are in the COM. This is how semantically equivalence is tested for efficiently.
- Current packet time - The time of the current ManageablePacketEvent that is being merged.

Both the COM and the COCM are implemented using the Haskell *Data.Sequence* (Sequences) package based on Finger Trees[64]. Sequences have amortized constant time lookup of elements at

the front and back and $O(log(min(i, n-i)))$ time lookups for elements at index i where n is the size of the sequence. If the assumption of chronological ordering within one trace file always held, inserting ManageablePacketEvent, merged or singleton, would always have constant time. However, in some cases, this assumption will not hold. Due to this, a merged or singleton ManageablePacketEvent must be re-inserted into the list chronologically. This accomplished through a binary search. Thus, every new ManageablePacketEvent will require $O(logn)$ time where n is the maximum size of the stitch list given stitch horizon and network parameters.

Stitching increments `PacketEventManagementMutableHash` state. The initial `PacketEventManagementMutableHash` state is one where the COM, COCM, and MHT are empty. This state is then folded across all packets from the list of PcapSources using `foldPacketDataWith` located in *Network.PcapStitch.PcapSources*. A fold is a high-order function that applies a function to a data structure, accumulates a result at each element, and returns a final value. Given the list $[1, 2, 3, 4, 5]$ folding addition over would result in a value of 15. Similarly, `PacketEventManagementMutableHash` is folded over packets chronologically from trace files using the `perPacketData` function located in *Network.PcapStitch.PacketEventManagement*.

`perPacketData` increments `PacketEventManagementMutableHash` state by one packet. It performs five major ordered actions each time it is called with a new packet:

1. Conversion - Converts the packet to a ManageablePacketEvent. This is where the packet is processed.

2. Purge - Removes and outputs any ManageablePacketEvents from the COM or COCM that have expired according to the stitch horizon. This is where stitch horizon consistency is maintained.

3. Look-up - Looks for a ManageablePacketEvent in the MHT that is semantically equivalent according to the PEF. This is where semantically equivalence checks are carried out.

4. Handle Look-up Result - If the look-up failed, insert the ManageablePacketEvent into the COM and MHT. If the look-up succeeded, merge the ManageablePacketEvent or ,if the Man-

ageablePacketEvent has a merge count equal to the maximum stitch count, move the ManageablePacketEvent to the COCM and remove it from the MHT. This is where merging takes place.

5. Increment Current packet time - Change the current time to the timestamp of the input packet. This is how the stitch horizon is slid into the future.

The COM and COCM maintain stitch horizon consistency through the purge action. Semantical equivalence is checked during the look-up action. Each ManageablePacketEvent has a hash generated from the components referred to by the PEF. That is, if a PEF was specified of `Net.Source+Net.Destination+Net.ID+Net.Offset`, a hash would be generate from the network layer source and destination address,the network layer id, and the network layer offset for each ManageablePacketEvent. Hash tables that do not assume a perfect hash function have lists at each hash entry. Finding a hash table entry involves matching a hash entry and, if the hash entry list is greater than one, another equivalence function to locate the correct list element. HMT is no different, the hash of a ManageablePacketEvent is used to locate the list, and the PEF to locate the correct list element.

When a ManageablePacketEvent is removed from COM (because it is outside the stitch horizon or because it is complete) it must also be removed from the HMT. Not doing so can cause collisions with future ManageablePacketEvents. Removing a ManageablePacketEvent from the HMT is different than looking for a merge. When a merge takes place a hash and PEF are used. When a removal takes place a hash and equality are used. That is, a removal needs to remove identical ManageablePacketEvents from the COM or COCM and the HMT. This prevents removal of spurious ManageablePacketEvents in the HMT.

The MergeOrEqual data structure in *Network.PcapStitch.PacketEventManagement* manages these two types of look-ups. All ManageablePacketEvents are entered into the HMT wrapped in a merge type. When a look-up according to PEF is desired the hash table is queried with a Hash and a ManageablePacketEvent wrapped in a merge type as well. If a query is made against the HMT with a hash and a ManageablePacketEvent wrapped in a equal type, equality is used for look-up.

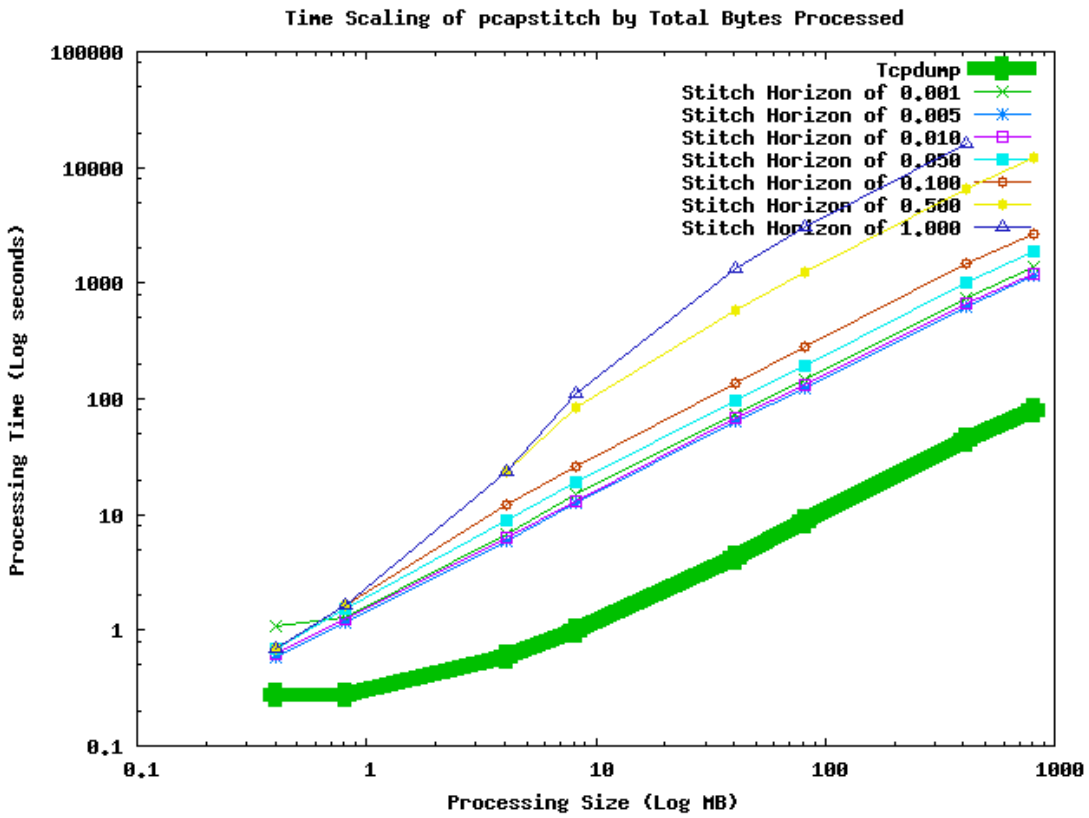

Figure 3.15: Performance of running pcapstitch on various trace files with various stitch horizons.

ManageablePacketEvent is the work-horse of pcapstitch. Consequently, it is where the bulk pcapstitch run-time is spent. This section will conclude with performance measurements. Real time was coarsely measured using `time`. The data for this performance experiment was measured on a network similar to Figure 3.3. pcapstitch output was direct to `/dev/null`. The baseline comparison is tcpdump outputting the file to `/dev/null`. Tcpdump uses libpcap to read packets from trace files as pcapstitch does. Therefore, its performance should be the lower bound of pcapstitch's performance. Figure 3.15 shows different file sizes and how they perform as stitch horizon increases. Figure 3.16 shows different packet counts and how they perform as stitch horizon increases. Both figures generally show multiplicative scaling with tcpdump performance. However, larger stitch horizons start to impact performance exponentially.

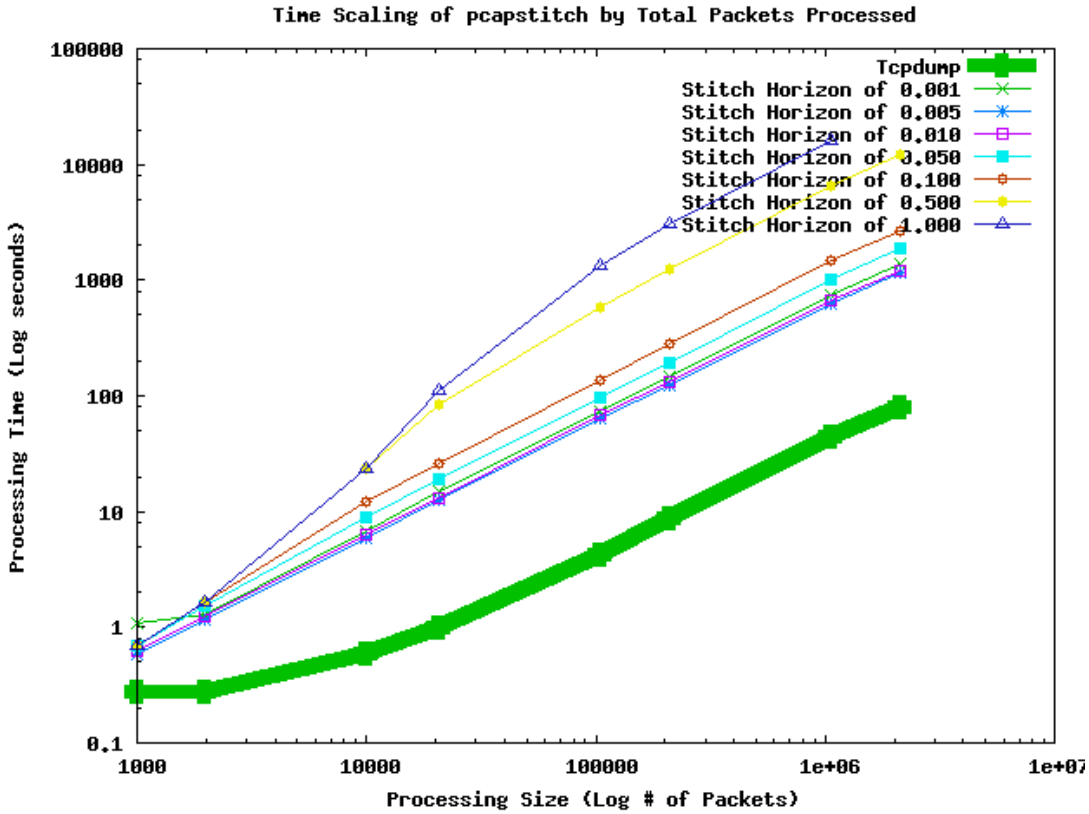

Figure 3.16: Performance of running pcapstitch on various trace files with various packet counts.

3.4 Miscellaneous pcapstitch Issues

pcapstitch can measure singleton one-delay and loss on networks. The term "network" is vague; in practice when measuring on networks every detail matters. When collecting trace files, details are equally important. The detail depth of a network and its constiuent compenents (*e.g.*, devices, software, link types, usage patterns) relate to confidence about measurement accuracy. This section discusses some of the details that can affect pcapstitch.

3.4.1 Difference of Wire Time and Trace File Record Time

Trace files record a timestamp of all packets. The semantics of this timestamp should be taken to mean "when the packet was received at the host" However, there is error in the timestamp because it includes processing delay. This processing delay aggregates as a packet moves from the port on a device's NIC, through the kernel, to the userland software stack and finally at the software that records the timestamp. Processing load on a device running a trace utility can cause variable processing delay. It is also important to note where the trace utility is getting its timestamp; in some operating systems it will be from when the kernel receives the packet and in others it will be when the trace utility received the packet. Depending on the accuracy required, modelling this processing delay may be necessary.

3.4.2 Assumption of PEF Component Relative Uniqueness

The components used to construct PEFs were chosen because they are relatively good at avoiding collisions. In the default PEF, the network layer id, network layer offset, and application payload hash are the components that are most likely to generate a unique signature. When packets have no application payload the application payload hash will always be the same. Network layer id and offset provide uniqueness because most operating systems increment one of them on a per-packet basis. This behavior is not standardized, it is valid to send two subsequent packets with the same network layer id as long as they are not fragemented. When using pcapstitch a quick check to verify that PEF components will generate unique values should be done.

3.4.3 In Transit Modifications of Packets May Impact Accuracy

For a PEF to merge packets accurately there must be some part of the packet readable by the PEF that does not change in transit. For instance, a router will completely replace a link layer header and modify some fields at the network layer (*e.g.*, time-to-live, checksum); a PEF that relies on any of these modified fields will fail to properly stitch. A network device that is engaged in port/network address translation(PNAT) will change all layers except the transport payload and thus only a PEF of *App* (application payload hash) will allow proper stitching. When relying solely on application payload hashes, understanding frequency of payload repetition is necessary to avoid PEF collisions. A trace file collected at a device or in a network path that has the potential to change all specifiable PEF components cannot be stitched. In the current pcapstitch implementation, packet fragmentation in the network will garuantee that packets cannot be stitched. However if all observation points see the fragmented packets, stitching will occur normally. In summary, care must be taken when collecting network trace files to be stitched with regards to whether a PEF will be able to merge packets.

3.4.4 Kernel Dropped Packets and Loss Accuracy

It is not uncommon for trace utilities to report packets dropped at the kernel. A packet that was dropped at the kernel, most likely made it to its proper application destination. However, the kernel did not send this packet to the trace utility. Kernel dropped packets indicate that the kernel is having difficulty keeping up with network throughput. This can occur because of significant processing load on the kernel or because the connecting link is very fast in comparison to the hardware the kernel is running on. A suggestion to avoid kernel droppage due to processor load is to employ dedicated observations points.

3.4.5 Packets Recorded In Trace File That Are Not Chronologically Ordered

Trace files have the potential for receiving packets that are out of order chronologically. Suppose we have received traffic for a TCP stream, suppose the queue of received data is a duplicate syn packet and data packet with its push flag set. Both of these packets will have chronologically correct

timestamps. However, because of the push flag the data packet may be processed by the kernel first according to implementation details. This would cause tcpdump to write the data packet before the syn packet in a libpcap file. This behavior impacts efficiency because to properly detect when packets have moved outside the packet horizon they must be ordered. If packets were always read from trace files in chronological order, insertion into the stitch list could be accomplished by added the packet to the end. Because it is possible that packets will not be chronologically ordered in the trace file, precautions must be taken that require less efficient handling of packets. Specifically when a packet is going to be added to the stitch list, a check is done to see if the packet is chronologically out of order (is the packets timestamp earlier than the latest timestamp on the stitch list). If the packet is in chronological order, it can be added to the end of the stitch list. If the packet is out of chronological order, it must be inserted in order which involves a search.

3.4.6 Synchronization, Timing, and pcapstitch Accuracy

pcapstitch relies heavily on the ability of observation points to be time synchronized. Relatively accurate time synchronization can be had through use of common frameworks such as NTPv4. Very accurate one-way delay measurements may require polling the NTPv4 processes at all obersation points and recording offset. This information could then be used to adjust raw singleton one-way delay measurements from pcapstitch.

One-way delay can be defined as:

$OneWayDelay_{measured} = OneWayDelay_{true} + SystemicError + RandomError.$

When performing any experiment one must try to account for $SystemicError$ and bound $RandomError$.

TCP/IP Offloading

Some modern NICs do additional processing onboard; this is referred to generally as offloading. Some examples of offloading are, network and transport layer checksum calculation and partial TCP protocol negotiation. Moving such processing to a NIC can free a CPU from some responsibilities and reducing traffic on the PCI bus. Some offloading can change packet structure and PEF sensitive

fields. Large TCP segment send and receive offloading can cause erroneous packet loss. Send offloading may cause trace files to record a bigger than MTU packets being sent which the NIC will fragment by creating appropriate TCP headers. Receive offloading behaves similarly with respect to received packets. When using pcapstitch any NIC offloading should be used with caution.

3.5 Installing pcapstitch

pcapstitch source code can be found at `http://patch-tag.com/r/chaosape/pcapstitch`. Darcs[65], a distributed source control program, can be used to retrieve it. Specifically, the darcs command `darcs get http://patch-tag.com/r/chaosape/pcapstitch` will retrieve pcapstitch.

pcapstitch was written in the Haskell[46] programming language and is distributed as a cabal package. The Common Architecture for Building Applications and Libraries (Cabal) is a package management system for Haskell. The most convenient way to get a working Haskell environment with Cabal is by downloading The Haskell Platform (`http://hackage.haskell.org/platform/`).

Upon successfully installing The Haskell Platform, cabal-dev should be installed via the command `cabal install cabal-dev`. Cabal-dev provides similar functionality to cabal with the exception that when libraries are installed they will be sandboxed. This ensure any current cabal repositories will remain undisturbed.

Once cabal-dev is installed enter the top-level pcapstitch package directory and download a cabal package database via the command `cabal-dev update`. pcapstitch dependencies can then be built via the command `cabal-dev install-deps`. The command

```
cabal-dev configure && cabal-dev build && cabal-dev install
```

will configure, compile, and install pcapstitch. At this point pcapstitch should be installed in `./cabal-dev/bin/` within the top-level pcapstitch package directory. This program is relatively monolithic and will have no shared library dependencies with any libraries from cabal-dev. Therefore, moving the binary to other compatible machines should be possible.

pcapstitch has only been tested on Linux variants. If problems are encountered acquiring or building pcapstitch contact Daniel William DaCosta at chaosape@chaosape.com

Chapter 4: Conclusion

pcapstitch is a tool that collects singleton one-way delay and loss measurements. These measurements are recovered from trace files collected from multiple observation points in a network. pcapstitch use is demonstrated in Section 3.2 where a simple network experiment is performed.

pcapstitch construction is interesting because of it's use of Haskell, the NHP library, and the concept of a stitch horizon. Construction of practical, performance sensitive, system tools in Haskell provides stronger garuantee's of correctness and weaken's the position that "system tools require c". The NHP library relies on Haskell's type system extensions to provide a prototype embedded domain-specific language that can provide *provable* garuantee's about byte alignment of network headers. It demonstrates how dependent types can be used to enforce assumptions between software components, specifically, that network data is always byte aligned. Finally, and most importantly, pcapstitch uses the concept of a stitch horizon to manage the practical issues concerned with merging semantically equivalent packets; conversion of delay to loss, efficiency of packet merging, and bounding memory.

OpenIMP and pcapstitch are the only tools known to the author that can collect singleton one-way delay and loss measurements passively. OpenIMP is a robust, professional, comprehensive tool suite for measuring one-way delay and loss as well as other metrics. OpenIMP's setup requires probes and a measurement controller. Acquiring and processing data requires a web server and database. pcapstitch is a minimal, prototype application built only to measure one-way delay and loss from libpcap formatted files collected from multiple observations points. It outputs a flat text file that is easily parsed and analyzed by common UNIX utilities. pcapstitch does one thing well, stores data in flat text files, and relies on subsequent filter processes. In this way, pcapstitch's functionality is useful.

There are many other measurement tools that exists. However, they either require active probing of the network or must assume network symmetry.

All network characteristics can be measured with respect to delay. A lost packet, is one whose delay has exceeded some maximum expected time. pcapstitch's stitch horizon specifies such a time. One-way measurements are important because networks are not symmetric. A path used by a packet to arrive at a destination may not be used by response packets and queues on devices do not fill symmetrically. Therefore, accurately measuring network characteristics requires one-way delay and loss measurements.

A network's utility is derived from application utility. However, networks are imperfect. These imperfections affect the performance of the protocol stacks used by devices to communicate. Application utility is influenced by network characteristics through protocol stacks. The application specific protocols can be influenced as well. pcapstitch accurately measures network characteristics and measurement can be done easily with common UNIX utilities. This allows application utility, protocol stack performance, and, ultimately, network utility degradation to be diagnosed accurately.

Bibliography

[1] D. Awduche, A. Chiu, A. Elwalid, I. Widjaja, and X. Xiao. Overview and principles of internet traffic engineering, 2002.

[2] openimp.

[3] Klaus Krippendorff. *Information theory: structural models for qualitative data*. Sage, Thousand Oaks, CA, 1986.

[4] Fui Hoon Nah and Kihyun Kim. *Managing Web-enabled technologies in organizations*. IGI Publishing, Hershey, PA, USA, 2000.

[5] Dennis F. Galletta, Raymond M. Henry, Scott McCoy, and Peter Polak. Web site delays: How tolerant are users? *J. AIS*, 5(1):0–, 2004.

[6] Thomas H. Cormen, Clifford Stein, Ronald L. Rivest, and Charles E. Leiserson. *Introduction to Algorithms*. McGraw-Hill Higher Education, 2nd edition, 2001.

[7] Claude Elwood Shannon. A mathematical theory of communication. *Bell Systems Technical Journal*, 27:379–423,623–656, 1948.

[8] Xerox. The ethernet: a local area network: data link layer and physical layer specifications. *SIGCOMM Comput. Commun. Rev.*, 11:20–66, July 1981.

[9] James F. Kurose and Keith W. Ross. *Computer Networking: A Top-Down Approach*. Addison-Wesley Publishing Company, USA, 5th edition, 2009.

[10] Indra Widjaja Alberto Leon-Garcia. *Communicatio Networks: Fundamental Concepts and Key Architectures, Second edition*. McGraw Hill Higher Education, Boston, 2004, ISBN 0-07-119848-2. Hardcover, pp 900, plus XXVII, 2005.

[11] Rfc 791 internet protocol - darpa inernet programm, protocol specification, September 1981.

[12] S. Deering and R. Hinden. RFC 2460 internet protocol, version 6 (IPv6) specification, December 1998.

[13] Information Sciences Institute. RFC 793, 1981. Edited by Jon Postel. Available at http://rfc.sunsite.dk/rfc/rfc793.html.

[14] J. Postel. User datagram protocol, August 1980.

[15] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee. Hypertext transfer protocol – http/1.1, 1999.

[16] P. V. Mockapetris. Domain names - implementation and specification, 1987.

[17] P. Chimento and J. Ishac. Defining Network Capacity. RFC 5136 (Informational), February 2008.

[18] Lawrence Berkley Laboratory. tcpdump.

[19] Gerald Combs and many contributors. wireshark.

[20] nmon. ncap.

[21] Sun Microsystems. snoop.

[22] Harris Corporation. An/prc-117f(c).

[23] Sally Floyd and Van Jacobson. Random early detection gateways for congestion avoidance. *IEEE/ACM Trans. Netw.*, 1:397–413, August 1993.

[24] Srisankar S. Kunniyur and R. Srikant. An adaptive virtual queue (avq) algorithm for active queue management. *IEEE/ACM Trans. Netw.*, 12:286–299, April 2004.

[25] Internet Engineering Task Force. Ietf.

[26] G. Almes, S. Kalidindi, and M. Zekauskas. A One-way Delay Metric for IPPM. RFC 2679 (Proposed Standard), September 1999.

[27] G. Almes, S. Kalidindi, and M. Zekauskas. A One-way Packet Loss Metric for IPPM. RFC 2680 (Proposed Standard), September 1999.

[28] Eric Steven Raymond, Thyrsus Enterprises, Copyright Eric, and S. Raymond. The art of unix programming. Addison-Wesley, 2003.

[29] J. Postel. Internet Control Message Protocol. RFC 792 (Standard), September 1981. Updated by RFCs 950, 4884.

[30] Mike Muuss. Ping.

[31] Van Jacobson. Traceroute.

[32] Shawn Ostermann. tcptrace.

[33] Mark Allman, Wesley M. Eddy, and Shawn Ostermann. Estimating loss rates with tcp. *ACM Performance Evaluation Review*, 31:2003, 2003.

[34] NLANR/DAST : Iperf - the TCP/UDP bandwidth measurement tool. http://dast.nlanr.net/Projects/Iperf/, Accessed 2007.

[35] S. Avallone, A. Pescapè, and G. Ventre. Analisys and experimentation of internet traffic generator. In *Proceedings of NEW2AN 2004*, pages 70–75, St. Petersburg, February 2004.

[36] Analysis and experimentation of an open distributed platform for synthetic traffic generation. In *Proceedings of the 10th IEEE International Workshop on Future Trends of Distributed Computing Systems*, pages 277–283, Washington, DC, USA, 2004. IEEE Computer Society.

[37] S. Shalunov, B. Teitelbaum, A. Karp, J. Boote, and M. Zekauskas. A One-way Active Measurement Protocol (OWAMP). RFC 4656 (Proposed Standard), September 2006.

[38] Jose Luis Oliveira Paulo Salvador Antnio Nogueira Joao Silve Helder Veiga, Rui Valadas. jowamp.

[39] owamp.

[40] N. G. Duffield and Matthias Grossglauser. Trajectory sampling for direct traffic observation. *IEEE/ACM Trans. Netw.*, 9:280–292, June 2001.

[41] Tanja Zseby. Evaluation of building blocks for passive one-way-delay measurements, 2001.

[42] N. G. Duffield, A. Gerber, and M. Grossglauser. Trajectory engine: A backend for trajectory sampling. In *In Proc. Network Operations and Management Symposium (NOMS*, pages 15–19, 2002.

[43] impd4e.

[44] David Mills. Ntp accuracy.

[45] B. Claise. Specification of the IP Flow Information Export (IPFIX) Protocol for the Exchange of IP Traffic Flow Information. RFC 5101 (Proposed Standard), January 2008.

[46] Simon Peyton Jones, editor. *Haskell 98 Language and Libraries: The Revised Report*. http://haskell.org/, September 2002.

[47] Al Danial. cloc.

[48] Brian White, Jay Lepreau, Leigh Stoller, Robert Ricci, Shashi Guruprasad, Mac Newbold, Mike Hibler, Chad Barb, and Abhijeet Joglekar. An integrated experimental environment for distributed systems and networks. Boston, MA, December 2002.

[49] The Network Simulator NS-2. http://www.isi.edu/nsnam/ns/.

[50] Luigi Rizzo. Dummynet: A simple approach to the evaluation of network protocols. *ACM Computer Communication Review*, 27:31–41, 1997.

[51] Openssh.

[52] D.L. Mills. Network Time Protocol (NTP). RFC 958, September 1985. Obsoleted by RFCs 1059, 1119, 1305.

[53] Gnuplot.

[54] Linus Torvalds. The linux operating system.

[55] Glasgow haskell compiler.

[56] Luis Damas and Robin Milner. Principal type-schemes for functional programs. In *Proceedings of the 9th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '82, pages 207–212, New York, NY, USA, 1982. ACM.

[57] High-level option handling with getopt. http://www.haskell.org/haskellwiki/High-level_option_handling_with_GetOpt.

[58] Wolfgang Jeltsch. A flexible record system.

[59] Tim Sheard and Simon Peyton Jones. Template meta-programming for haskell. In *ACM SIGPLAN Haskell Workshop 02*, pages 1–16. ACM Press, 2002.

[60] Per Martin-Lof. An intuitionistic theory of types.

[61] Connor McBride. Faking it: Simulating dependent types in haskell, 2001.

[62] Simon Peyton Jones. Simple unification-based type inference for gadts. pages 50–61. ACM Press, 2006.

[63] Ralf Hinze. Church numerals, twice!, 2002.

[64] Ralf Hinze and Ross Paterson. Finger trees: a simple general-purpose data structure. *J. Funct. Program.*, 16:197–217, March 2006.

[65] Darcs. http://darcs.net/, 2009.

Appendix A: Novelty of pcapstitch

A.1 Correspondance with Dr. Vern Paxson

Dr. Vern Paxson is recognized as a world expert in network characterization. In 2008, he was awarded the Association for Computing Machinery's Grace Murray Hopper Award for his work in measuring and characterizing the Internet. In the interest of understanding the uniqueness of pcapstitch for measurement he was contacted. What follows is our email correspondance.

Email to Dr. Paxson:

```
Dr.  Paxson,

My Name is Dan DaCosta.  I am currently working on a tool I call
PcapStitch for my Master's dissertation.  PcapStitch identifies the
same packet from packet captures that have been collected at multiple
points in a network simultaneously.  This is done by using header
fields and a packet horizon ( a time based on maximum one-way latency
in the network ).  Identification is done after collection has been
completed.  I have written this tool in Haskell and it is near
complete.

In my limited network testing experience and background research, I
have found no tool that had this functionality.  I know there exist
network testing suites that can provide this functionality assuming
particular payload formats but I have found none that will provide
this functionality for arbitrary TCP/IP packet captures.

As I begin to write my dissertation my anxiety that I will stumble
```

```
upon a tool that accomplishes the same thing grows.  I expect that you
are extremely busy, but I would be very appreciative if you could give
me your opinion about the novelty of such a tool (e.g. "Tool x, y and
z all do that", or "Nope, haven't seen a tool that does that").

Thank you very much for your time!
-d.
```

Email response from Dr. Paxson:

```
> PcapStitch identifies the
> same packet from packet captures that have been collected at multiple
> points in a network simultaneously.

I don't know of any such tool already out there.  One person you might ask
would be Nick Duffield <duffield@research.att.com>, who has worked a lot
on "trajectory sampling" whereby copies of the same packet are recorded
at different routers as it travels.  He may have run across such a tool
while doing studies for the mechanisms he's worked on in this regard.

Best wishes,

Vern
```

A.2 Correspondance with Dr. Nick Duffield

Dr. Nick Duffield is AT&T and IEEE fellow for his work network in measurement, analysis, sampling and inference. What follows is our email correspondance:

Email to Dr. Duffield:

My Name is Dan DaCosta. I sent an identical inquiry to Dr. Vern Paxton, he suggested I contact you with this question. I am currently working on a tool I call PcapStitch for my Master's dissertation. PcapStitch identifies the same packet from packet captures that have been collected at multiple points in a network simultaneously (Dr. Paxson implied this might also be referred to as "trajectory sampling"). This is done by using header fields and a packet horizon (a time based on maximum one-way latency in the network). Identification is done after collection has been completed. I have written this tool in Haskell and it is near complete.

In my limited network testing experience and background research, I have found no tool that had this functionality. I know there exist network testing suites that can provide this functionality assuming particular payload formats but I have found none that will provide this functionality for arbitrary TCP/IP packet captures.

As I begin to write my dissertation my anxiety that I will stumble upon a tool that accomplishes the same thing grows. I expect that you are extremely busy, but I would be very appreciative if you could give me your opinion about the novelty of such a tool (e.g. "Tool x, y and z all do that", or "Nope, haven't seen a tool that does that").

Thank you very much for your time!

-d.

Email response from Dr. Duffield:

Dan,

I don't know of any such tool.

I'm not sure if trajectory corresponds exactly to what you are doing. It uses a packet hash (which is generally over payload as well as header) and time horizon to associate multiple observations of the same packet. We did some work (see second paper below) on evaluation of trajectory sampling, but this was more a proof of concept for backend evaluation of measurements, how to choose timeouts, database issues. It did not result in a tool for general use.

Also, I suggest that you contact Tanja Zseby Tanja.Zseby@fokus.fraunhofer.de because she has done work on empirical evaluation of trajectory sampling for performance measurements and may possibly have some relevant work in this area.

Here are some of our papers in this area:

(original paper)
Trajectory Sampling for Direct Traffic Observation, N.G. Duffield and M. Grossglauser, IEEE/ACM Transactions on Networking, v. 9 no. 3 (June 2001) pp. 280-292. Earlier version appeared in: Proceedings ACM SIGCOMM'2000, Computer Communications Review, Vol 30, No 4, October 2000, pp. 271--282
http://www2.research.att.com/~duffield/papers/DG-TS-ToN.pdf

(more on backend issues including associating different observations of

```
the same packet, timeout issues)

Trajectory Engine: A Backend for Trajectory Sampling, N.G. Duffield,

A. Gerber, M. Grossglauser, IEEE Network Operations and Management

Symposium 2002, Florence, Italy, April 15-19, 2002.

http://www2.research.att.com/~duffield/papers/DGG01-engine.pdf



Please do send me a pointer to anything you do / have done in this area,


Regards,


Nick
```

This email led me to OpenIMP[2], a more sophisticated and complicate (relative to pcapstitch) tool that can measure one-way delay and loss passively. It is developed by the Fraunhofer Institute for Open Communication Systems (FOKUS).