2016

Study of Queueing Delay in the Single-Hop Case

Muhsin Sulaiman King
Bard College

Study of Queueing Delay in the Single-Hop Case

Senior Project submitted to
The Division of Science, Mathematics, and Computing
of
Bard College

by
Muhsin King

Annandale-on-Hudson, New York
May, 2016

Abstract

Through the study of network characteristics it is possible to predict the overall behavior of network traffic, an aim which has significant implications with respect to network scheduling, traffic shaping, congestion control, and flow control. One of these characteristics is the delay that packets experience in the output queue of routers along the network path. This project attempts to implement and verify an untested scheme for measuring queueing delay in the single-hop case. The scheme uses a pair of probing packets, and estimates queueing delay by measuring the intra-probe gap between them after they traverse the router under test. Applications were created to test this scheme both in simulation and on a closed physical network. The scheme was verified in simulation, and, when tested on the physical network, probing packets experienced the gap compression that indicates queueing delay. However, due to a failure to generate stable queuing at the output interface of the router, and the inability to measure intermittent queueing from the router's console, these results are ultimately inconclusive.

Table of Contents

Dedication

To my parents. Thanks for everything.

Acknowledgements

I would like to express my gratitude to my project adviser Prof. Khondaker Salehin, for his guidance throughout this study.

… and to my academic adviser, Sven Anderson, for helping me when I most needed help.

… and to the rest of the Bard College computer science department (past and present):
Keith O'Hara, for encouraging my love of programming;
Robert McGrail, for making me work hard despite my best efforts;
and Rebecca Thomas, whom we all miss a great deal.

… and to my family, who are great.

… and to my friends, who are great.

1 - Background

1.1 - Introduction

Network characterization is a problem that is both intellectually stimulating and of eminent practical importance. The ability to predict the behavior of aggregate network traffic has significant implications with respect to network traffic scheduling, traffic shaping, congestion control, and flow control. This is increasingly important due to the rising popularity of high-bandwidth real-time applications (such as video/audio streaming, and video/audio chat software), and the exponentially increasing population of the Internet. However, the sheer scale of the Internet, the traffic volume, and the increasing diversity of network technologies makes network characterization a challenging task.[1]

1.2 - Network Characterization Overview

A network can be characterized by studying its various components. First among these are the properties of the physical devices that make up the network. The physical components of the Internet are comprised of routers, links, and end hosts (workstations, personal computers, servers, etc). These are the most basic building blocks of every network, and the network of networks that is the Internet. Of these, only the routers and links are relevant to this project.

[1] Ayyorgun, Sami, and Wu-chun Feng. "A Deterministic Characterization of Network Traffic for Average Performance Guarantees."

1.2.1 - Physical properties

Links are the transmission medium through which data passes from source to destination. This may be a single point-to-point medium (such as a direct Ethernet link or telephone line), a series of multiple connections (as in multiple Ethernet segments), or a broadcast medium (as in WiFi). When characterizing a network, each link has a number of potentially interesting properties. The propagation delay, for example, representing the time required for a packet of data to traverse the link, must be known to accurately estimate the behavior of traffic throughout the network. The capacity of the link must also be determined, describing the maximum rate at which packets of data can be sent through the link (ie. maximum throughput).

Meanwhile, routers transfer data between links in the network. While the internal architecture of routers varies, the primary purpose of a router is to examine each incoming packet, and determine which outgoing interface to transmit it on, based on the packet's final destination. The components of a router are shown in Figure 1.1. Packets arrive on each input interface that includes an input queue, in case a burst of traffic causes packets to be accepted faster than the router can process them. After packets are processed by the input interface, they are inspected at the switching fabric before they are placed on the appropriate output interface. The switching fabric allows packets from any input link to be transmitted on any output link, and uses a routing table to determine the best path to the packet's final destination. The rate at which the output interface transmits packets is identical to the outgoing link's capacity. As with the input interface, packets may arrive at an output interface more quickly than it has the capacity to process them, so each output interface also contains a buffer known as the output queue, where packets wait for their turn to be transmitted. Generally these buffers are designed as First In First

Out (FIFO) queues.[2] There are a number of properties that one may want to measure concerning routers in the network, including its overall traffic load and the various delays suffered by packets relayed through it.

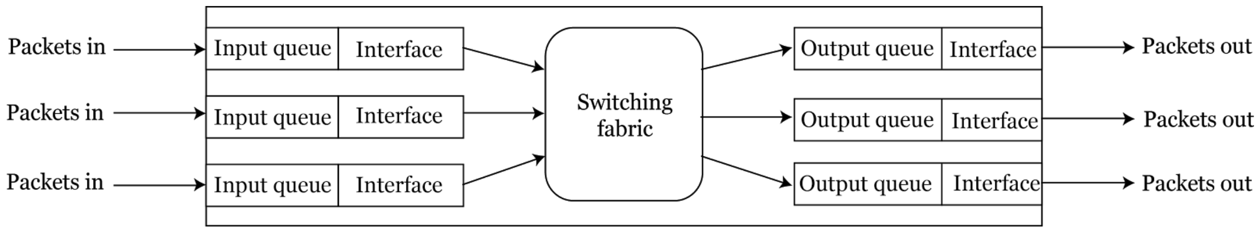


Figure 1.1: Internal components of a router. Packets are accepted via an input interface, after which a switching fabric forwards them to the optimal output interface to reach their final destination.

1.2.2 - Network Traffic and Latency

Aside from a network's physical properties, one must also characterize its interactions with traffic. Certain aspects of this interaction will be constrained by the physical properties of the network—in particular, the minimum possible delay and the maximum possible throughput of packets.[3] Within these limits, the actual conditions that packets will experience can vary greatly. In particular, the delay that a packet suffers as it passes through the network is the sum of the effects of many parameters.

As already mentioned, propagation delay is the time a packet spends passing from one end of a link to the other. More specifically, it is the time between the moment the packet's

[2] Sudjiman, David. "Router Output Queue."

[3] Kozierok, Charles M. "Performance Measurements: Speed, Bandwidth, Throughput and Latency."

leading bit leaves the output interface at the source, and when that same bit arrives at the input interface at the destination. Simply put, for a link of length *d* in which bits propagate at a speed of *v*, the propagation delay can be assumed to be *d/v*.

Second, the transmission delay a packet experiences is the time required to place a packet onto a link, whether from the output interface of a router or end host. More specifically, the time between when the packet's first and last bits are placed on the interface. For a packet of size *s* and outgoing link capacity *t,* the transmission delay will be *s/t.*

Finally there is routing delay, which is the time that packets spend inside routers, or, the time between the first bit of the packet arriving at the router and that same bit arriving on the output link. This delay is broken up into several parts. First is there is processing delay, which is the time takes the router to look up the packet's destination address on its forwarding table, and determine the appropriate output interface. The packet must then be moved to the chosen interface, which incurs further delay. Both of these operations incur delays that may or may not be proportional to packet size. Lastly, there is queueing delay, which is the time a packet spends waiting in queues in the router. Generally, significant queueing only occurs at the output queue,[4] as shown in Figure 1.2. When an output interface receives traffic from the switching fabric more quickly than it can be processed, excess traffic is move onto the output queue, where it waits for its turn to be place on the link. Packets in this queue experience a queueing delay proportional to the number of packets ahead of them in the output queue, and the sum of their combined transmission delays.[5]

[4] Crovella, Mark, and Balachander Krishnamurthy. *Internet Measurement: Infrastructure, Traffic, and Applications,* 112.

[5] Marbach, Peter. "Queueing Delay."

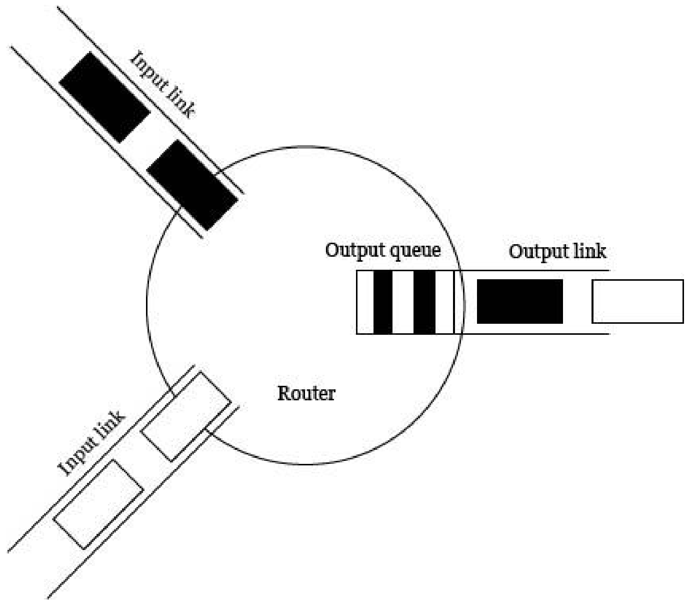


Figure 1.2: Output queuing in a router. Packets enter the router through input links faster than the output link can accept them, requiring them to queue before the output interface

1.3 - Queueing Delay

Under ideal conditions (ie. when traffic is scheduled such that a router's inputs never exceed its output capacity), the router would never develop an output queue, and thus packets would not be delayed in this manner. However, a queue can be built up in one of two ways: first, if traffic to the router is arriving faster than the router can process it. Second, if multiple traffic streams flow into the router and are diverted to the same output link, their constituent packets can collide, forcing one packet to wait on the output queue while the other is transmitted. In either of these scenarios, an output queue can sometimes build up rapidly in the router to the point that it is completely filled. When the output queue is full, any other packets destined for the output interface are dropped with no further processing.

Failure to drop packets when the queue is full can cause bufferbloat, a phenomenon whereby an excess of buffering packets causes high latency and packet delay variation (also known as jitter)[6,7]. However, dropping packets can cause its own set of issues—for one thing, all information in the packet is lost. If the protocol used to send the dropped packet does not require that the recipient send an acknowledgement of its receipt (as in the User Datagram Protocol (UDP) commonly used in live video and audio streaming applications[8]) the information is effectively irretrievable.

Due to the potential seriousness of queue buildup, and the related effect of queueing delay on latency throughout the network, queueing delay is a significant metric that must be accounted for to properly characterize a network. It is also one of the most complicated variables to measure—unlike transmission and processing delay, which are relatively static based on the network's link capacity and the header information of incoming packets, queuing delay is a dynamic parameter determined by the ever-changing amount of traffic the network is experiencing. Without direct access to the configuration terminal of every router in the network, measuring the effect of queuing delay on a network can be extremely challenging.[9]

[6] Gettys, Jim, and Kathleen Nichols. "Bufferbloat: Dark Buffers in the Internet."

[7] Demichelis, C., and P. Chimento. "IP Packet Delay Variation Metric for IP Performance Metrics (IPPM)."

[8] Kurose, James F., and Keith W. Ross. *Computer Networking: A Top-down Approach Featuring the Internet*.

[9] Kay, Rony, Ph.D. "Pragmatic Network Latency Engineering Fundamental Facts and Analysis."

1.4 - Network Characterization Techniques

There are several existing methods to measure various network parameters discussed above. These can be classified into two broad categories: passive measurement and active packet probing techniques.[10]

1.4.1 - Passive Measurement

Passive measurement tools track data transmissions through the network, deriving the necessary information from existing traffic. These are potentially very efficient and accurate, as they do not add any additional traffic to the network. However, this same fact limits the applicability of these techniques to network paths that have recently carried traffic. A good example of a passive measurement tool is the Remote Network Monitoring (RMON) protocol, an industry standard that enables multiple network monitors to exchange data collected by analyzing the traffic flow on a distributed local network (LAN). RMON passively collects information on traffic throughout the network, while not making a significant impact on network traffic itself.[11]

1.4.2 - Active Packet Probing

In contrast, active probing techniques are more invasive, while allowing for a greater depth of network exploration. This is achieved by sending packets, individually or in pairs, and tracking their transmission end to end. Because active techniques introduce new traffic into the

[10] Cottrell, Les. "Passive vs. Active Monitoring."

[11] Rouse, Margaret. "What Is RMON (Remote Network Monitoring)."

network, they can analyze network paths even where no prior traffic exists. One can use data collected in this manner to help characterize the network as a whole, and gather information about its component devices and traffic. The downside is that adding extra traffic to the network dilutes existing traffic, potentially decreasing the accuracy of the measurements. The Ping protocol is a great example of an active technique. Ping sends and Internet Control Message Protocol (ICMP) echo request to from one host to another to determine the reachability of the host and measure the round-trip time between the endpoints. Though Ping uses active packet probing, it is minimally invasive, adding only a single packet at a time to the network load. On top of this, ICMP packets have minimal overhead, and are generally not used to carry data as in UDP or TCP.[12]

1.5 - Active vs. Passive Measurement of Queueing Delay

Passive measurement of queueing delay is difficult because it can require infrastructural support from the Internet[13] and clock synchronization[14] to collect data of sufficient accuracy. To avoid this problem this project use an active packet probing technique—specifically, a packet pair probing technique. This helps reduce the need for infrastructural support, as measurements are only taken at the sending and receiving hosts rather than on hardware dispersed throughout

[12] Postel, J. "Internet Control Message Protocol."

[13] Anagnostakis, K.g., M. Greenwald, and R.s. Ryger. "Cing: Measuring Network-internal Delays Using Only Existing Infrastructure."

[14] Papagiannaki, K., S. Moon, C. Fraleigh, P. Thiran, and C. Diot. "Measurement and Analysis of Single-hop Delay on an IP Backbone Network."

the network. The need for clock synchronization is eliminated because each host takes these measurements without reference to the other, based on their own internal timing.

1.6 - Objectives

In the following chapters we present a scheme for measuring queueing delay in the single-hop case.[15] This probe-gap model was previously tested in a simulation environment, but never verified experimentally, which is the ultimate goal of this project. The scheme is relatively simple, measuring queueing delay by sending a pair of UDP packets from one host to another, passing through a single router (ie. single-hop). At the receiving end, the queueing delay can be inferred by the change in the intra-probe gap between the packets. The intra-probe gap is the time difference between the receipt of the last bits of the first (head) packet, and the last bits of the second (tail) packet. Ultimately when tested on real network traffic, this scheme should allow for accurate measurements without a great deal of infrastructural support or instrumentation at the end hosts.

[15] Salehin, Khondaker M., and Roberto Rojas-Cessa. "Scheme for Measuring Queueing Delay of a Router Using Probe-Gap Model: The Single-Hop Case."

2 - Related Work

There have been several prior attempts to develop schemes for measuring queueing delay. One is pathchar, an active scheme that measures the link capacity and queueing delay over a network path using a linear-regression model on the round-trip times of ICMP packets.[16] While extremely simplistic in design, pathchar is considered network-intrusive, which is to say that the probing packets themselves can have a significant effect on the existing flow of traffic on the end-to-end path. On top of this, it requires a long measurement time, and while pathchar has been shown to be an accurate measure of link capacity, it did not prove be an accurate measurement of queueing distribution for a single link.[17]

Another method, cing, measures queueing delay using pairs of ICMP packets. Though this scheme is relatively simple, its measurement accuracy and deployment feasibility depend heavily on collecting timestamps for every probing packet from the routers in the path while under test.[18]

Another study created a scheme to measure queueing delay for the purpose of detecting Distributed Denial of Service (DDoS) attacks.[19] Similar to this project, they used a UDP packet pair probing technique. By separately measuring the difference between the intra-probe gaps at

[16] Jacobson, Van. "Pathchar — a Tool to Infer Characteristics of Internet Paths."

[17] Downey, Allen B. "Using Pathchar to Estimate Internet Link Characteristics."

[18] Anagnostakis, K.g., M. Greenwald, and R.s. Ryger. "Cing: Measuring Network-internal Delays Using Only Existing Infrastructure."

[19] Lu, Wei-Zhou, Wei-Xuan Gu, and Shun-Zheng Yu. "One-way Queuing Delay Measurement and Its Application on Detecting DDoS Attack."

the input and output links along the path, they deduced the Fourier magnitude of end-to-end queuing delay probability density function (PDF). Because they measured intra-probe gaps separately at either end, they were similarly able to avoid any reliance on clock synchronization or infrastructure support. They tested their scheme both in simulation and experimentally in a closed network.

There also exist passive schemes for queueing delay measurement. These can be divided into instrumentation-based and analysis-based schemes. Instrumentation-based schemes measure queueing delay by instrumenting its input and output links using a specialized packet or kernel process, and a packet monitor.[20] Analysis-based schemes apply a stochastic model on captured traffic data to characterize the queueing delay of a router. These schemes are complex, however, and require prior insight about traffic load, the number of traffic flows, and the rate of packet loss.

[20] Papagiannaki, K., S. Moon, C. Fraleigh, P. Thiran, and C. Diot. "Measurement and Analysis of Single-hop Delay on an IP Backbone Network."

3 - Problem Statement

3.1 - The Probe Gap Model for Measuring Queueing Delay

The scheme implemented in this project utilizes active packet probing to measure queueing delay in the single-hop case. Packets are sent in pairs, called a compound probe, with the minimum possible time gap between them. In each compound probe, the heading packet is small and the trailing packet is large, as shown in Figure 3.1. This difference in size causes a dispersion gap between the two packets when they enter the link. A dispersion gap is the space between last bit of the heading packet and the first bit of the trailing packet. However, because the input link at the receiving host will have just started the process of receiving the trailing packet when its first bits arrive on the input link, what is measured instead is the intra-probe gap, or the time between the receipt of the first bits of the heading packet and the receipt of the last bits of the trailing packet (also shown in Figure 3.1). Unlike the dispersion gap, the intra-probe gap also includes the transmission time of the trailing packet. The transmission time of a packet T_p is the direct result of its size *s* and the capacity of the link *c,* as shown in the equation:

$$T_p = \frac{s}{c}$$

Equation 3.1

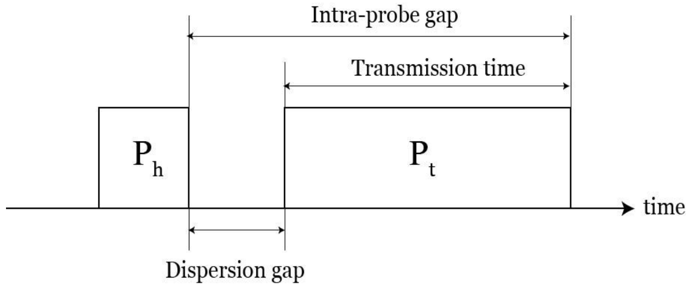


Figure 3.1: Intra-probe and dispersion gaps in a compound packet probe

As shown in Figure 3.2, the dispersion gap D_i between packets P_h and P_t is proportional to the difference in their sizes s_h and s_t, and the capacity of the links c_i and c_{i+1} before and after the router. This can be described by the equation:

$$D_i = \frac{s_t}{c_i} - \frac{s_h}{c_{i+1}}$$

Equation 3.2

This dispersion can then be determined at the receiving host by measuring the intra-probe gap G_{i+1}. In the case of two equally sized packets and two links with equal capacity, this proportional dispersion gap between them be would be nonexistent:

$$s_h = s_t = s$$

Equation 3.3

$$c_i = c_{i+1} = c$$

Equation 3.4

$$D_i = \frac{s}{c} - \frac{s}{c} = 0$$

Equation 3.5

Thus, we need to use a probe configuration with a small heading and large trailing packet.

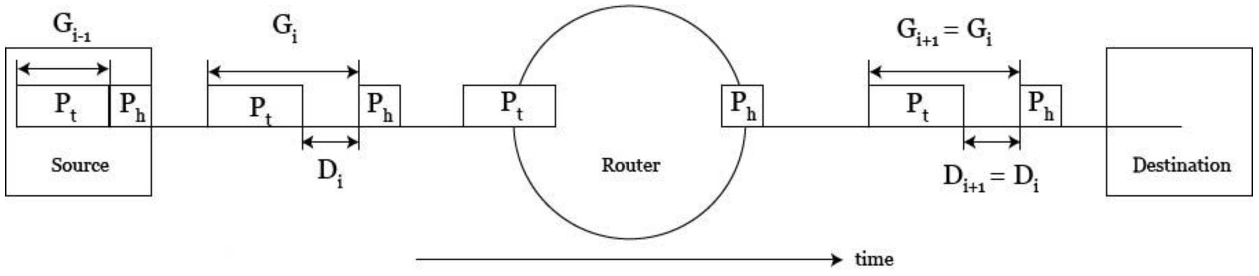


Figure 3.2: Dispersion and intra-probe gaps in a compound packet probe before and after a router experiencing no cross traffic

The dispersion gap generated by the different sizes of the packets can then be used to measure queueing delay in the router in the single-hop case. If the router is under sufficient load due to cross traffic, a queue will emerge before the output interface. If a queue exists in the router at the time when the heading packet arrives, it will be placed at the back of the queue rather than being transmitted onto the output link. Here, the heading packet will suffer a queueing delay equal to the sum of the transmission times of the packets in the queue when it arrives. While the heading packet is waiting in the queue, the trailing packet will begin to catch up to it, reducing the dispersion gap between them, as shown in Figure 3.3. This reduction in dispersion can then be measured at the receiving host by comparing the measured inter-probe gap $M[G_{i+1}]$ after a router under cross traffic and the expected intra-probe gap without cross traffic $E[G_{i+1}]$. If a reduction in dispersion is observed, we can use this information to infer the amount of queuing delay the packets have experienced on the end-to-end path.

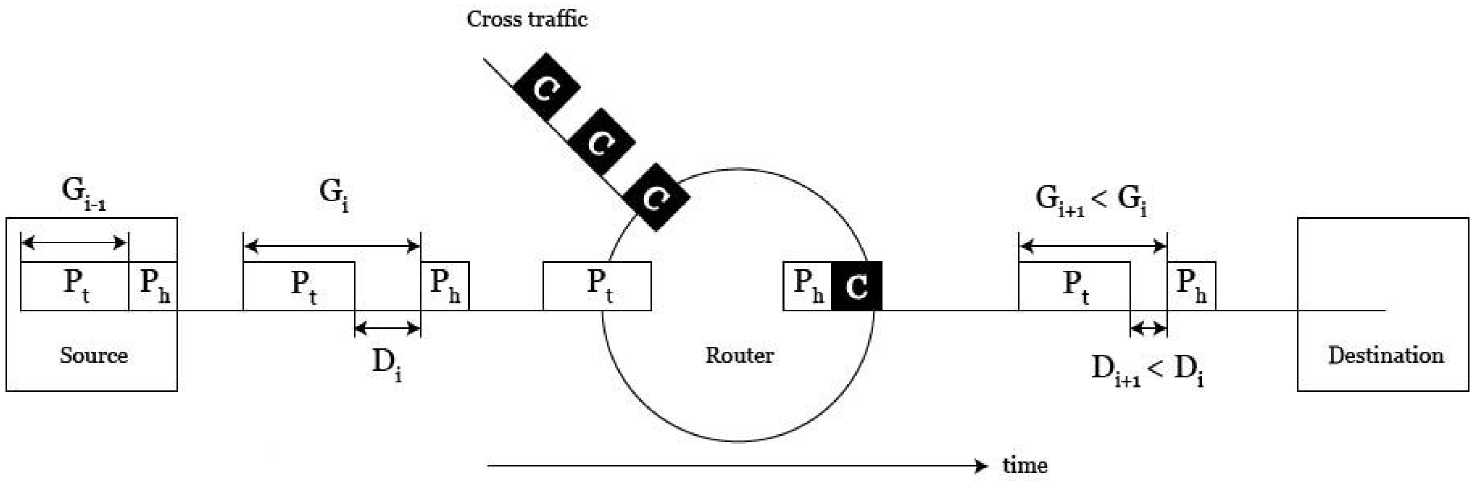


Figure 3.3: Cross traffic causes a queue at the output link in the router, reducing the dispersion gap between packets in the compound probe

The detailed steps used to implement the proposed scheme are listed below:

1. Identify the link capacities c_i and c_{i+1} of the links before and after the router on the end to end path.
2. Determine packet sizes s_h and s_t such that their predicted dispersion gap is greater than zero by Equation 3.2.
3. Estimate the expected intra-probe gap between probing packets P_h and P_t of sizes s_h and s_t respectively at the router as:

   $E[G_{i+1}] = T_t + D_i$
   Equation 3.6

   Where T_t is the transmission time of P_t given by Equation 3.1 and D_i is the predicted dispersion gap given by Equation 3.2
4. Send P_h and P_t with the minimum possible gap between their transmission (equal to the transmission time of P_t given by Equation 3.1) from the host connected to the first link.

5. Timestamp P_h and P_t upon receiving them at the host connected to the second link and measure the intra-probe gap $M[G_{i+1}]$ where $M[G_{i+1}] \geq s_t \div c_{i+1}$.
6. Calculate the change in the intra-probe gap with reference to $E[G_{i+1}]$ as:

   $\Delta[G_{i+1}] = E[G_{i+1}] - M[G_{i+1}]$,

   Such that $\Delta[G_{i+1}] > 0$ when the intra-probe is compressed, while $\Delta[G_{i+1}] \leq 0$ when in the intra-probe gap is either unchanged or increased. Discard the measurements in the latter category.
7. Finally, also discard measurements where $M[G_{i+1}] = s_t \div c_{i+1}$, as this suggests that the queueing delay in the router could be larger than D_i.

3.2 - Probe Filtering

Given a continuous flow of cross traffic through the router, not every compound probe will be able to extract meaningful data about the queueing delay that packets experience in the router. There are several possible configurations of packet probes and their interaction with cross traffic, which are shown in Figure 3.4. Case 1 shows what happens if the cross traffic fails to generate an output queue in the router. In this case, the intra-probe gaps before (G_i) and after (G_{i+1}) the router will be the same, showing that no queueing delay was experienced. Case 2 shows the ideal case, where queueing in the router causes the heading packet in the probe to be delayed, reducing the dispersion gap when measured at the destination. Cross traffic does not insert itself inside the packet probe, and causes no interference with the trailing packet. Case 3 is the same as Case 2, except some cross traffic has inserted itself between the heading and trailing packet. However, the inserted cross traffic is small enough that the dispersion gap is still reduced by the heading packet's queuing delay.

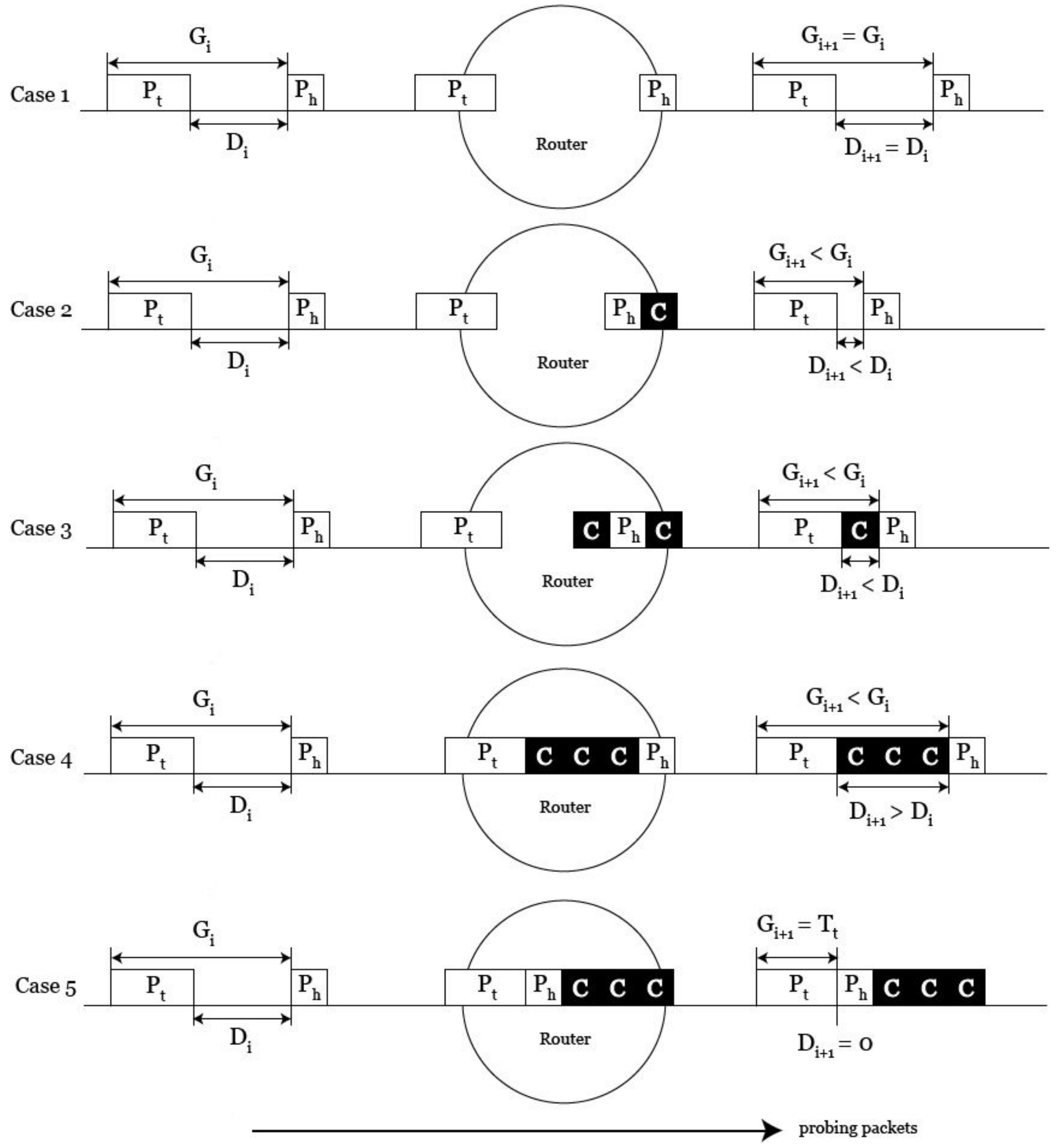


Figure 3.4: Possible interactions between the compound probe and cross traffic packets, and their effect on intra-probe and dispersion gaps.

In Case 4 enough packets are inserted between the heading and trailing packets to widen their dispersion gap on the second link. This can be measured at the destination as an increase in the intra-probe gap relative to the estimated value by Equation 3.6 value. In Case 5, there is so much queueing at the router that the dispersion gap between the heading and trailing packet vanishes completely, and they are bunched together.

Only Case 2 provides meaningful data about queueing delay in the router that can be measured at the end of the path. In Case 2 there is some queueing delay, and the amount can be inferred from the reduction in the dispersion gap between the two packets.

Cases 1, 3, 4, and 5 all represent results that need to be filtered. In Case 1 there is simply no queueing delay, which is identifiable if $E[G_{i+1}] = M[G_{i+1}]$. Cases 4 and 5 can also both be identified easily at the end of the path. Case 4 can be identified if the intra-probe gap measured is greater than the expected value without cross traffic, while Case 5 can be seen if the intra-probe gap is equal to the transmission time of the trailing packet, given by Equation 3.1

Case 3 is trickier, because it will still register at the destination as a reduction in the dispersion gap between the packets. However, because traffic has been inserted between the probing packets, the reduction in dispersion is not as great as it should have been given the amount of queueing delay experienced by the heading packet. This Case will be impossible to detect without a statistical model for predicting queueing behavior in the router.

3.3 - Building an Output Queue

To test the scheme described above, a controlled testbed needs to be designed. To determine the effectiveness of the probe-gap model, an output queue must be generated in the router where probing packets are transferred onto the next link (L_{i+1} in section 3.1). As mentioned in the section 1.3, output queues can be created in one of two ways. First, if the rate of incoming packets is greater than output link capacity, packets in excess of the capacity will be placed on the queue. Second, if two or more packets try to enter the output link at the same time, only one will go through while others wait their turn in the output queue.

Generating a queue simply by increasing the rate of cross traffic such that it is greater than the output queue is problematic. If the rate of cross traffic is left constant, the output link will never be able to “catch up” with the rate of input, and packets will continue to build up in the queue faster than the output interface can process them. This will cause the queue to fill completely, and ultimately cause severe rates of packet loss. This is an issue for the proposed scheme because this active probing technique requires that both packets in the compound probe reach their final destination. An overly large rate of cross traffic also increases the likelihood of Cases 4 and 5 shown in Figure 3.4 (which one depends on the initial size of the dispersion gap).

This leaves packet collision as the remaining method of queue generation. Packet collision occurs in live network traffic, but to procure it using artificial traffic multiple traffic streams must be utilized, as shown Figure 3.5. With several cross traffic streams feeding into the router at certain rates, collisions will occur, causing some packets to be moved into the output queue. However, these streams must be carefully tuned such that the rate of collision is great enough to cause measurable queuing, but not so great that the queue overflows.

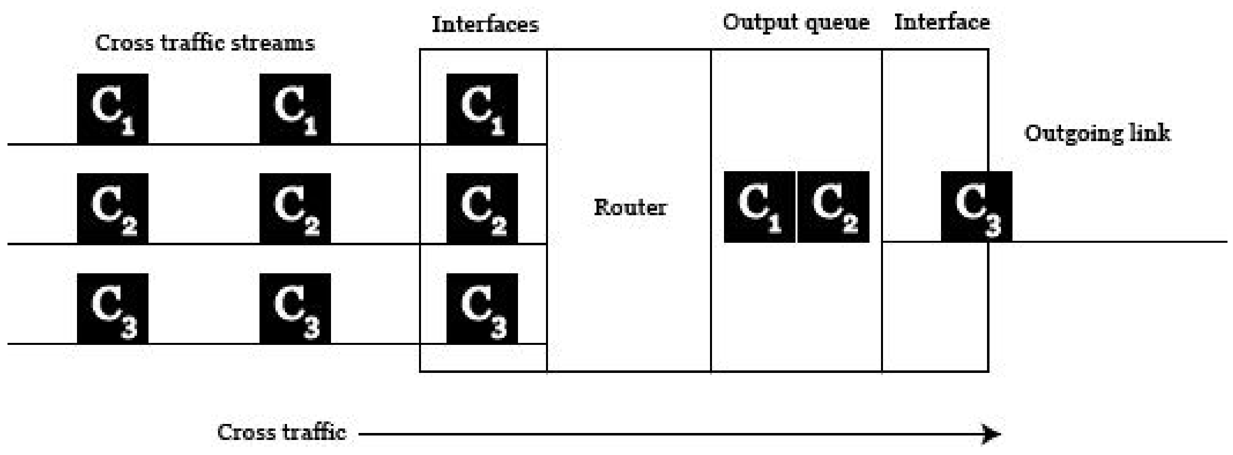


Figure 3.5: Multiple cross traffic streams send packets to the same output interface, causing a collision. Two packets are forced to wait in the queue while the other is transmitted.

4 - Creating the Testbed Network

This section describes the creation and configuration of the closed network used to test the probe-gap model for measuring queueing delay.

4.1 - Equipment

4.1.1 - Hardware

The network uses four computers and one router, set up in the configuration shown in Figure 4.1. Two of these workstations take the role of Client and Server in the packet-probing application, while the other two are be dedicated to cross traffic generation. The Client host can also send cross traffic packets, allowing for up to three simultaneous streams directed at the server host to generate a queue in the intermediate router. While it is possible to send multiple cross traffic streams from a single host, this is not desirable as it will not result in collision queueing at the router's output interface, but rather at one of the input interfaces. Each computer needs an Intel-compatible network interface card (NIC), and must be installed with Intel's Linux e1000 Ethernet Base Driver.

The router used is a Cisco 3640 Integrated Services Router. It has five Fast Ethernet interface ports, allowing for simultaneous transmission between all three. Fast Ethernet interfaces allow for a maximum bit rate of 100Mbps. Four of these interfaces are connected via ethernet to the four hosts. Setting up the router requires a serial-to-Ethernet cable, which allows access to the router's configuration terminal. For this project router configuration is conducted through the Client host (this is a matter of preference, and the host used for router configuration does not impact experimental results).

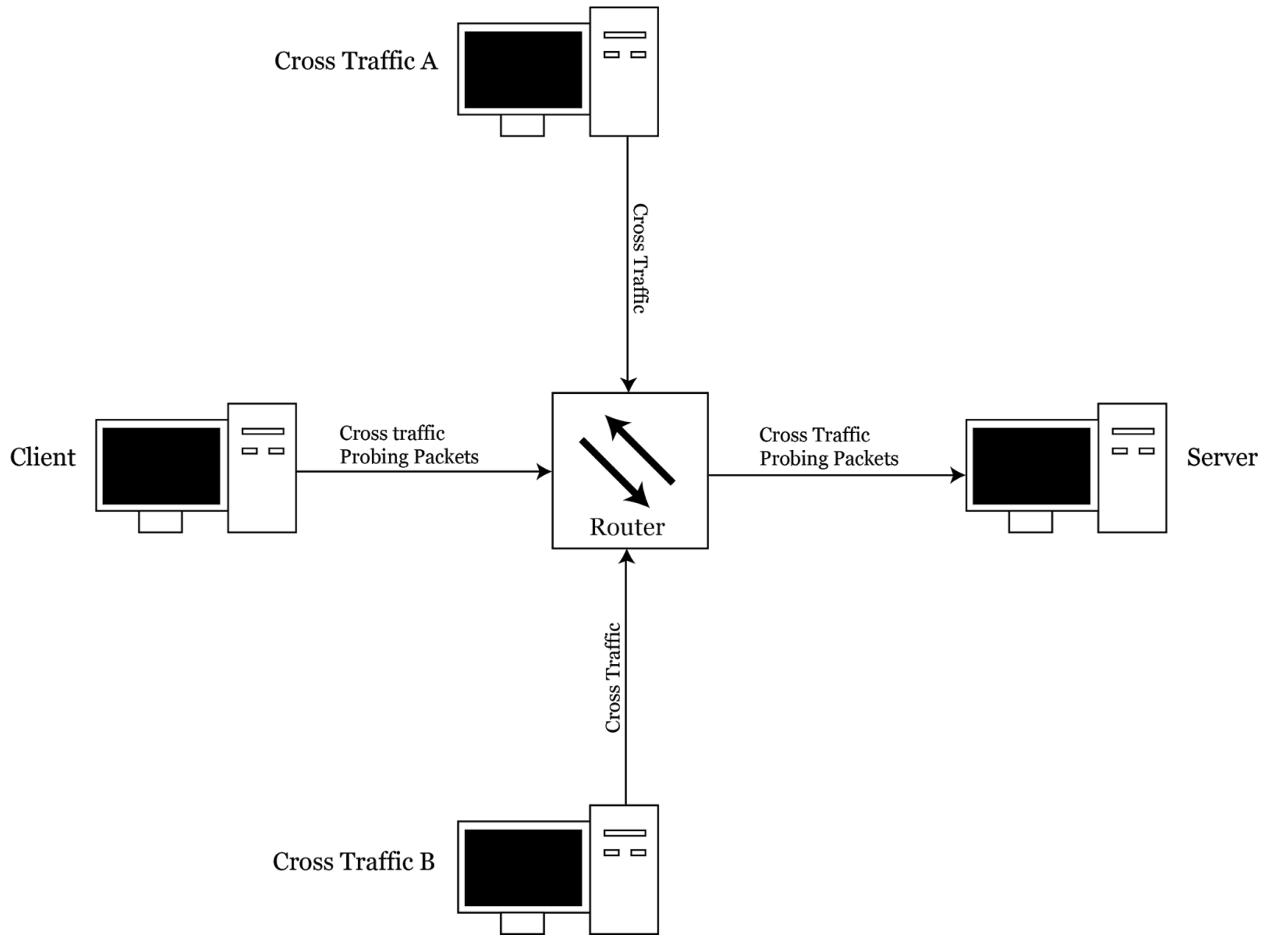


Figure 4.1: Map of devices and traffic flows in the testbed network

4.1.2 - Software

Each workstation is installed with Ubuntu 14.04, along with the Ethernet driver detailed above. The Client and Server stations must also have installed the Java Development Kit (JDK) version 8. Every host also must be installed with the ethtool utility, which is used to configure the Ethernet interfaces of each host. All hosts that are used to generate cross traffic (the Client and Cross Traffic A/B hosts) are installed with Iperf, a utility used to create traffic streams and measure the throughput of the network that is carrying them. While developing the application, it was also necessary to measure traffic on the end-to-end path as it is sent and received by the Client and Server hosts. This was done using the Wireshark utility. Finally, the workstation used to configure the router is installed with Minicom, a text-based modem control and terminal emulation program.

4.2 - Host Configuration

4.2.1 - Network Interface Card Parameters

Each host, equipped with an Intel NIC, drivers, and the appropriate Linux utilities, is then configured with certain Ethernet link parameters. Using ifconfig, we configure the IP addresses of all hosts such that they are in the same network, but different subnets, as shown in Table 4.1. Each host is also given the same subnet mask. Finally the speed and duplex mode of each Ethernet connection is configured. Using the ethtool utility, speed (the maximum bitrate at which packets can be transferred) is set to 10 Mb/s, while the duplex mode is set to full (allowing for simultaneous send/receive on a single link).

Host	IP address	Subnet mask
Server	169.254.147.222	255.255.0.0
Client	169.254.7.190	255.255.0.0
Cross Traffic A	169.254.203.42	255.255.0.0
Cross Traffic B	169.254.5.216	255.255.0.0

Table 4.1: IP address and subnet mask assignments for all hosts in the network

4.2.2 - Network Interrupt Coalescence

The final parameter configured with ethtool is network interrupt coalescence. Many network interfaces use interrupt coalescence (IC) to generate a single interrupt for multiple packets received in a short time interval. IC is helpful in high-bandwidth systems, as lumping packets into a single interrupt reduces the interrupt processing required on a per-packet basis[21]. However, IC introduces additional delay to packets received, and alters the dispersion between packet pairs.[22] In effect, as the rate of interrupt increases latency increases as well, while CPU utilization decreases.[23] The probe gap model implementation is not CPU intensive, but relies heavily on consistent and predictable dispersion between packet pairs. As such, it is necessary to minimize the effect of IC for receiving hosts in the network. A short experiment was devised to test the effects of IC on the testbed network, described in section 6.1. Ultimately an interrupt value was found that eliminates the effect of IC on the network. This is applied by setting the rx-usecs IC parameter to 75 via ethtool.

[21] "Interrupt Coalescence at the NIC Layer." *IBM Knowledge Center*.

[22] Prasad, Ravi, Manish Jain, and Constantinos Dovrolis. "Effects of Interrupt Coalescence on Network Measurements."

[23] Brouer, Jesper Dangaard. "[net-next,2/5] Ixgbe: Increase Default TX Ring Buffer to 1024."

4.3 - Router Configuration

After each host is configured, the router's interfaces must be set up to facilitate communication between the hosts. The router's five interfaces are labeled fa0/0, fa2/0, fa2/1, fa3/0, and fa3/1. All of these are used except fa3/1.

4.3.1 - Accessing the Router Configuration Terminal

The router can be configured from a workstation connected to it via an Ethernet-to-Serial cable. From that computer, the Minicom Linux utility can be used to access the router's configuration terminal.[24] Using Minicom, a new serial port setup is created to access the relevant serial connection. The setup is configured to use a baud rate of 9600 bits per second, and 8-N-1 encoding. This gives access to the router's configuration terminal, from which router interface parameters can be set.

4.3.2 - Router Interface Parameters

All the interfaces used are given identical network parameters, with the exception of their IP address. Each interface is assigned an address such that it shares both its network and subnet with the host that it shares an Ethernet connection with. All interfaces are given a uniform subnet mask. The assignments used are shown in Table 4.2. Each interface is also assigned the same link speed of 10Mb/s and full duplex mode as in the network's end hosts. A uniform link speed is used, as variable link speeds along the network path will affect the dispersion gap of probing packets.

[24] Sudjiman, David. "Installing Minicom to Connect to Cisco Router Using Console Cable."

Interface	IP address	Subnet mask
fa0/0	169.254.147.221	255.255.255.0
fa2/0	169.254.7.189	255.255.255.0
fa2/1	169.254.203.41	255.255.255.0
fa3/0	169.254.5.215	255.255.255.0

Table 4.2: IP address and subnet mask assignments for relevant router interfaces

When all hosts and interfaces are configured and connected, their connectivity is tested with ping packets. Every host in the network should have a clear line of communication to every other host. A complete map of the testbed network with all IP address assignments is shown in Figure 4.2.

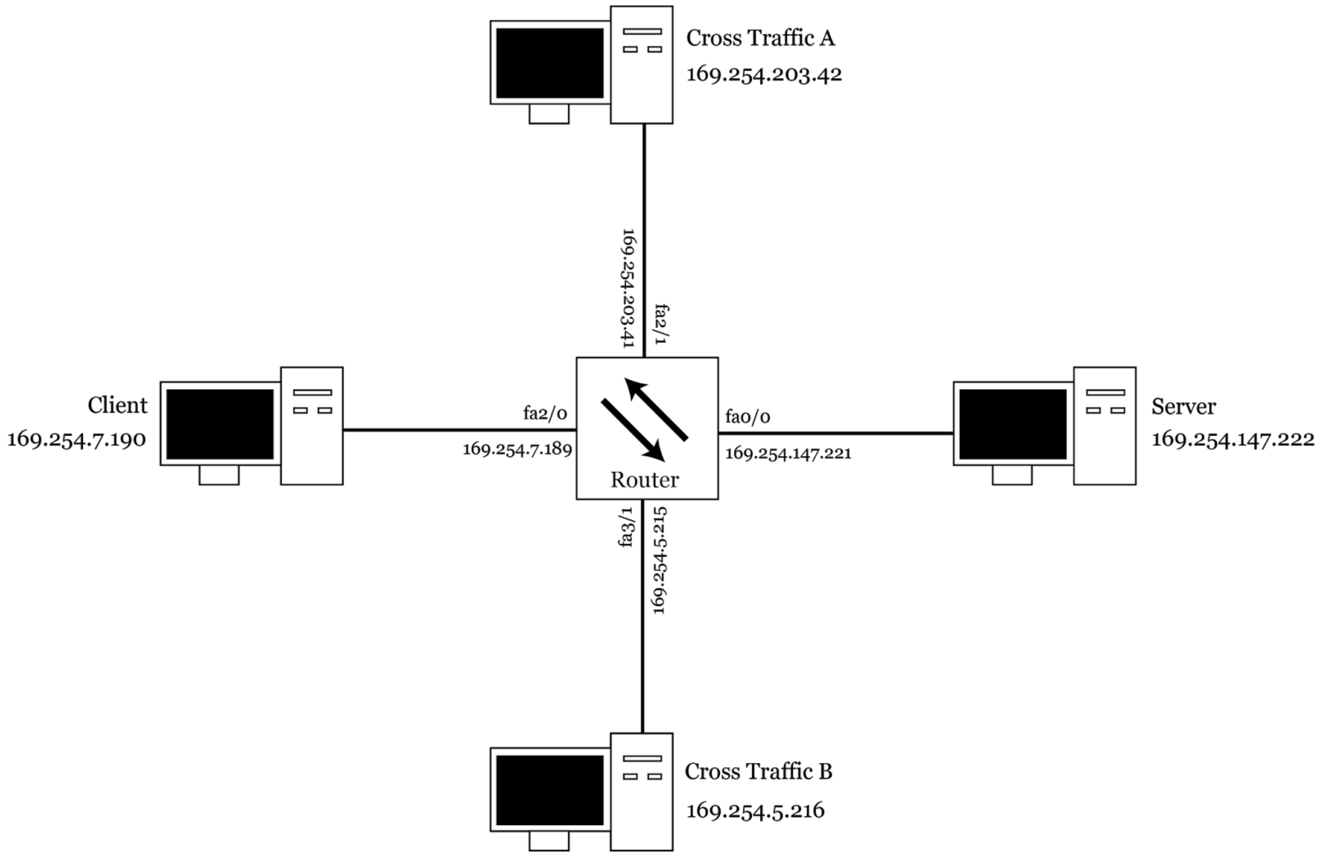


Figure 4.2: Map of all devices in the testbed network and their assigned IP addresses.

5 - Application Design

5.1 - Implementing the Probe Gap Model for Measuring Queuing Delay

5.1.1 - Application Overview

After the network is built, the compound probe scheme for measuring queueing delay is implemented in Java. The basic design of the application is to send trains of packet probes from a process running in the Client host to another process in the Server host. Each train contains some number of identical packet pairs. The Server process measures the intra-probe gap of each packet pair, and then calculates an average value for each packet train. Between packet trains, the size of probing packets may or may not change, allowing for multiple probe configurations to be tested in a single run. When building these applications, there were a number of design goals in mind:

1. **Simplicity:** the probe gap model is simplistic by design, and the code should reflect this. No modules are required outside of standard Java libraries. Complexity should be minimized, and unnecessary functions should be eliminated.
2. **Efficiency:** packet pairs must be sent with as little delay as possible between them, and a large number of probing packets will be necessary to gather meaningful data. As such, the code should be efficient, and not induce a significant load on the host. Ideally it should run in *O(nm)* time, where *n* is the number of probe trains, and *m* is the number of packet pairs per train. This means that the overhead for each additional probe sent should be constant.

3. **Modularity:** the code should be able to adapt to the changing parameters of the experiment. Lower level methods dealing with packet delivery and byte manipulation should be abstracted such that top level classes never need to deal with the details of UDP/TCP communication.

Data is sent between the Client and Server processes in two streams. The first is a Transport Control Protocol (TCP) loop, which the Client uses to relay any relevant experimental parameters that the Server needs before it can accept probing packets. A TCP connection is used for experimental controls, as the TCP protocol includes measures for error correction and packet receipt acknowledgement that make it significantly more reliable than UDP traffic. The second stream is make up of the UDP probing packets which are sent from Client to Server after the server has relayed acknowledgement of the control parameters. Unlike the Client, the Server-side application runs continuously, waiting to for notification from the Client of an incoming set of packet trains..

5.1.2 - TCP Control Loop

The control loop consists of two simple classes, TCPClient and TCPServer, both of which are capable of sending and receiving data to and from one another. There are a number of parameters that need to relayed from Client to the Server before packet probing can begin. These include the number of packet trains the server should expect, the number of packet pairs in each train, the size of the heading and trailing packets for the first train, and the amount that these sizes should decrement in each successive train. When the Server has received all the parameters necessary to set up the experiment, it sends an acknowledgement message back to the Client, which initiates the UDP loop.

5.1.3 - UDP Packet Probing

The UDP experimental loop begins with the generalized superclass UDPSendReceive, which contains all methods necessary for a host to send or receive UDP packets. It also contains the class variable for the host socket, as these are a uniform type for both clients and servers (unlike in TCP). Its two subclasses are the UDPClient and UDPServer classes, which use these methods to send or receive packet pairs. The UDPClient class sends two packets of given sizes, with no delay between them, to a given IP address, over a given port. Meanwhile, the UDPServer class is programmed to receive two packets of given sizes, and calculate the time between their receipt, ie. the intra-probe gap.

These UDP and TCP Client/Server classes are combined into a single Client and Server pair. To run the experiment, the end-user starts the Server on one host, with no additional arguments given. The Client meanwhile, takes several arguments. It must be told the IP address of the Server host, the number of packet trains to send, and the length of each packet train, along with the size of each packet. It will also be provided with an inter-probe gap, or the time it should wait between sending packet pairs. A good value for the inter-probe gap is half of the round-trip time of an individual packet pair on the end-to-end path. Any smaller and probes may interfere with one another, while a larger gap will cause unnecessary delay in experimentation.

5.2 - Traffic Collision Simulation

5.2.1 - Simulation Overview

On top of testing the scheme in a physical network, a simulation was also created to test the probe gap model for measuring queueing delay. The simulation emulates the behavior of a network similar the physical network created in sections 4.1-3. The simulated network consists of up to three cross traffic hosts directing traffic toward a single output link in a router. Each simulated traffic stream uses the same network parameters as in the testbed: a full-duplex, 10Mb/s link, with no interrupt coalescence. The simulation is designed such that each step simulates one microsecond of real time. The codebase consists of three classes: TrafficSimulator, ProbeSimulator, and RouterSimulator.

5.2.2 - TrafficSimulator

The TrafficSimulator class simulates a single cross traffic stream. Similar to an Iperf UDP client process, it is initialized with the desired bit rate and the size of the datagrams that are sent along the path. Because all the simulated links have a uniform capacity of 10 Mb/s but the desired bit rate may be lower than this, packets will be spaced to create, on average, the desired rate of traffic. For example, to create a traffic stream that averages 5 Mb/s on a 10 Mb/s link, the stream adds a delay between each packet (ie. inter-packet gap) equal to the transmission time of each packet. This is to say that it spends half of the time transmitting packets, and half the time waiting. This is the same process that an Iperf client uses to create the traffic streams at bit rate different from the link speed. The ratio of delay to packet transmission is directly proportional to the difference between the desired traffic rate *R* and the link's actual capacity *L*. The transmission time of each packet is equal to its size *s* divided by the 10Mb/s link speed. The intra-packet gap (the time between when the simulated interface begins transmitting one packet and the time when it begins transmitting the next) is equal to the size of the packet, in bits divided by the desired traffic rate. Thus, the spacing between packets is equal to the intra-packet gap minus the transmission time of each packet, shown in the equation:

$$\text{spacing} = \frac{s}{R} - \frac{s}{L}$$

Equation 5.1

5.2.3 - ProbeSimulator

The ProbeSimulator class attempts to reproduce the behavior of the probe-gap model for measuring queueing delay. Similar to the Client application created for the physical testbed, it requires a number of arguments, including the size of the heading and trailing packets, and the inter-probe gap. At each simulation step, the ProbeSimulator determines how much longer to delay before transmitting another heading or trailing packet. After a heading or trailing packet is sent, the delay time before sending the next heading or trailing packet is equal to the inter-probe gap given at initialization, plus or minus a random value, used to prevent the probing streams from synchronizing with the traffic streams. The inter-probe gap is randomized between probes—not between the individual packets in the probe. The time between transmission of heading and trailing packets is equal to the transmission time of the trailing packet, calculated by Equation 3.1. Thus, the initial intra-probe gap is also equal to this value as there should be no gap between the packets as they are sent from the simulated network interface.

5.2.4 - RouterSimulator

The RouterSimulator class utilizes TrafficSimulator and ProbeSimulator objects to simulate the testbed that was created in sections 4.1-3. At each simulated step, it increments time steps for each of its cross traffic streams and its simulated packet probing application. The output link, which is the destination of all traffic in the simulation, is represented by a simple FIFO queue, implemented with a Java LinkedList. The first position (index = 0) in this queue represents the packet that is currently being processed by the output link. Any remaining packets (index > 0) represent the output queue in the router. Each time the delay counters in the traffic or probe simulators reach zero, their step functions return the appropriate packet size to be transferred into the router, destined for the output link. At each time step, the simulated router processes a number of bits equal to the link capacity, 10Mb/s, divided by the time elapsed during the step (one microsecond):

$$\frac{10 \times 10^{6} \text{ bits}}{\text{second}} \times \frac{1 \times 10^{-6} \text{ seconds}}{\text{step}} = \frac{10 \text{ bits}}{\text{step}}$$

Equation 5.2

When the packets arrive in the router, the packets will have a predictable intra-probe gap, G_i, given by Equation 3.6. Based on this, we can determine whether or not queuing caused by simulated traffic resulted in a compression in the dispersion gap. The results from this simulation are shown in section 6.4.

6 - Procedure and Results

6.1 - Testing the Effects of Interrupt Coalescence on Compound Probes

Early on in the design of the physical testbed, the interrupt coalescence of network interface cards throughout the network was interfering with probe-gap measurements, as described in section 4.2.3. An experiment was designed to isolate the effects of IC on packet pairs in the testbed network, and to determine a rate of interrupt that would not disrupt probe-gap measurements of queueing delay. In this test, two hosts are connected via Local Area Network (LAN) connection, and shoot identically sized packets in pairs between them in pairs, with no delay between the delivery of each packet.

6.1.1 - Theoretical Intra-Probe Gap

Because the packets are the same size, the output interface of the sending host will produce no dispersion gap between them, as shown in Equations 3.3-5. With no dispersion gap, the intra-probe gap between the packets as measured at the receiving host should be equal to the transmission time of the second packet. This can be estimated by Equation 3.1. On top of this, these will be a small amount of dispersion caused by the interframe gap induced by the network interface card. The interframe gap is necessary for Ethernet devices to allow them to switch into receiving mode between packet transmissions, as otherwise they may miss a frame destined for them. For a 10Mb/s link, the interframe gap will add 9.6 μs to the predicted intra-probe gap value.[25]

6.1.2 - Procedure

1. Connect the Client and Server hosts configured in section 4 directly by via Ethernet wire. Ensure connectivity by sending ping packets between them.
2. On the Server host, compile the Sever.java class and run it with the command: java Server
3. On the Client host, compile the Client.java class and run it with the command:

   ```
   java Client 169.254.147.221 500 10 1000 1000 100 100 10000
   ```

   This begins a packet probing client process, with UDP packets directed at the Server host's address. It will send 10 packet pair trains of size 500. Both the heading and trailing packets begin with a length of 1000 bytes, and both values decrement by 100 bytes after each probe train. After each train, the Server will report the average intra-probe gap measured from all packet pairs in the train. Record these results.
4. Adjust the interrupt coalescence parameter "rx-usecs" on the Server host with the command:

   ```
   sudo ethtool -C eth0 rx-usecs N
   ```

   where N is an integer between 1 and 512.
5. Repeat steps 2-4, recording results for different IC parameters.

[25] "Interframe Gap and Spacing." *Savvius*.

6.1.3 - Results and Discussion

The default value for the rx-usecs interrupt coalescence parameter is 100. This resulted in the intra-probe gap distribution shown in Figure 6.1 below:

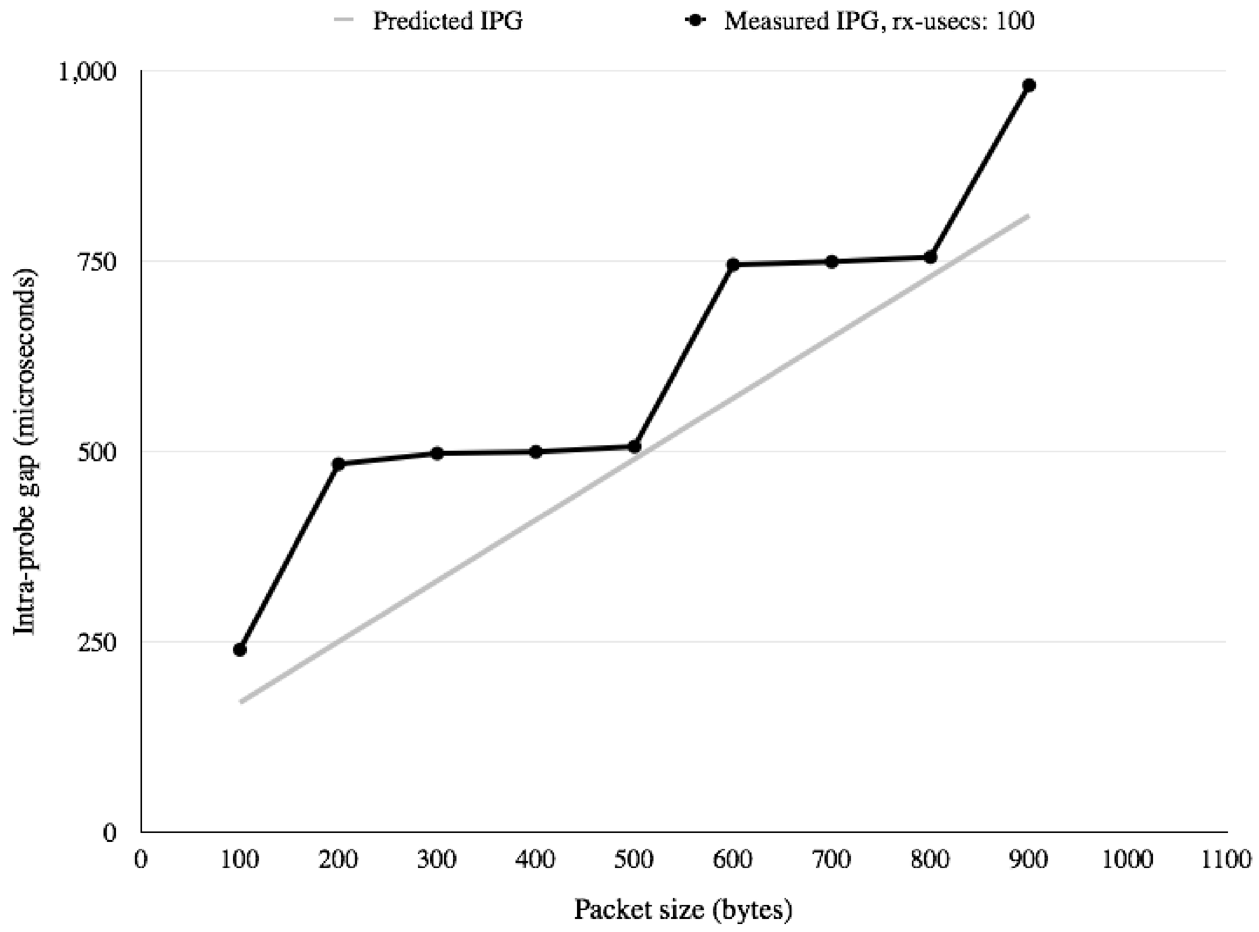


Figure 6.1: Server-side intra-probe gap measurements using identical packet pairs of varying size, and an rx-usecs IC parameter of 100. Link speed: 10 Mb/s.

After some trial and error, an IC value was found that eliminates the effect of interrupt coalescence on compound probes in the network. Figure 6.2 shows the packet size vs. IPG distribution for this case, using an rx-usecs value of 75:

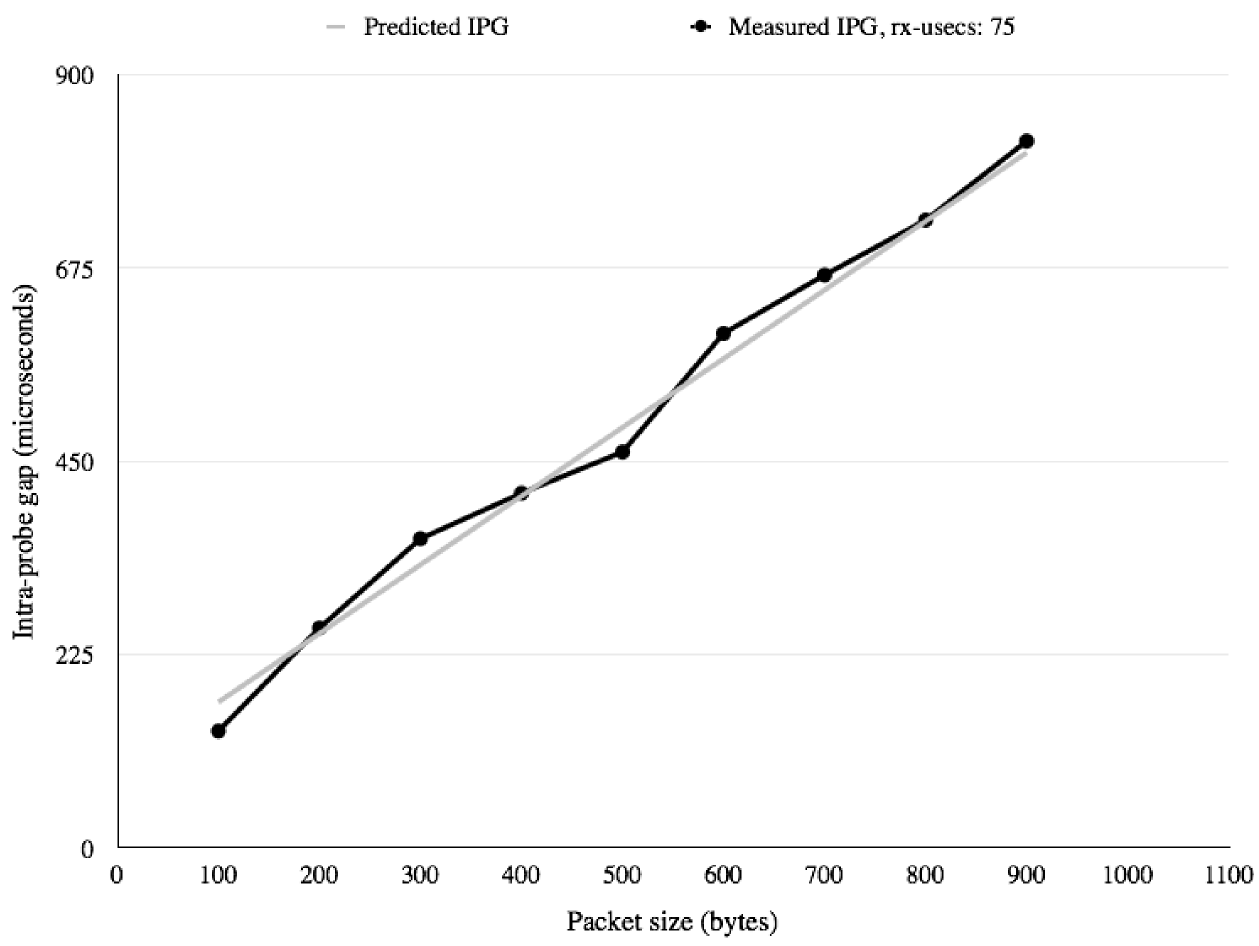


Figure 6.2: Server-side intra-probe gap measurements using identical packet pairs of varying size, and an rx-usecs IC parameter of 75. Link speed: 10 Mb/s.

6.2 - Generating a Stable Output Queue

To test the probe gap model for measuring queueing delay, a queue needs to be generated at one of the output links in the router. Three cross traffic streams were used to try and generate a stable queue in the router—one stream running from the Client host to the Server host, and two running from dedicated cross traffic hosts (Cross Traffic A and B in Figure 4.1-2) to the Server host.

6.2.1 - Procedure

An experiment was devised to attempt to generate stable queueing through trial and error, using three Iperf streams. These streams are generated by Iperf running in client mode, while the Server workstation runs Iperf in server mode, giving acknowledgement of traffic packets received. The detailed steps of this experiment are given below:

1. On the Server host, start Iperf using the command:

   ```
   iperf -s -u
   ```
2. Open up the router's console using the command "minicom cisco" from the connected workstation.
3. Prepare iperf commands on the remaining three workstations (Client, Cross Traffic A, and Cross Traffic B) with the template:

   ```
   iperf -c 169.254.147.221 -u -b Xm -l Y -t 1200
   ```

   where X is the desired traffic rate, and Y is the desired datagram size in bytes. Each host may or may not sure different values.
4. Initiate all three cross traffic streams simultaneously, and begin a timer.
5. In the router's console, use the command:

   ```
   show interface fa0/0 summary
   ```

   to display information about the router's Ethernet interface connected to the Server host. Track of the values of the "OHQ" (output holding queue) and the "OQD" (output queue drops) columns.

6.

   a. If the OQD value begins to rise and the OHQ value is near or equal to 40 (the maximum length of the output queue), stop the timer, and then stop the cross traffic streams. Record the time measured, which is the time until the first output queue packet drop.
   b. If the output queue is filled in the first iteration of the router's show interface fa0/0 summary command, record the time until the first packet drop as less than one second.
   c. If the cross traffic streams expire before any queueing is generated, record record a "timeout" for that trial.
   d. If the OHQ value rises above zero and does not continue to rise steadily, a stable queue has been generated. Record the average length of the queue.

7. Repeat steps 3–6, varying the bit rates and datagram sizes for the cross traffic streams.

6.2.2 - Results and Discussion

Using the procedure above, none of the cross traffic parameters tested successfully generated a stable output queue in the router—or, at least, a queue that was measurable from the router's console. There were also no cases of intermittent queueing observed, which is to say, a queue that builds and is then depleted. For the purposes of testing the proposed scheme for queueing delay measurement, either of these scenarios would have been useable. Instead, all configurations tested fell into three categories of queueing behavior:

1. The queue builds instantly, filling entirely before the first instance queue scan from the router console. Packets in excess of the router's maximum queue depth are dropped. This

suggests that the rate of collision that is so high that the router is unable to process the packets and deplete the queue before it fills.

2. The queue starts empty. After a certain number of seconds, the queue begins to build slowly until it is filled, after which excess packets are dropped as above. This suggests an initial state of zero collision, followed by period of collision in excess of the output link capacity. This is most likely caused by traffic synchronization, which is to say that after a certain period, asymmetric traffic flows align such that their packets enter the router simultaneously for an extended period.
3. The queue remains empty for the entire duration of the experiment (ten minutes). This suggests that no traffic collision occurred, or that any time there was collision, the time that any packets spent in the queue did not align with the instant of queue measurement from the router console.

Table 6.1 shows all cross traffic parameters that were used to try and generate a stable queue in the router, using the three-stream model described above. Columns 2-7 show the Iperf bit rate and datagram size parameters used on each host connected to the listed router interface. The last column shows the time until the output holding queue (OHQ) is filled.

Trial	fa2/0 bit rate (Mb/s)	fa2/0 datagram size (bytes)	fa2/1 bit rate (Mb/s)	fa2/1 datagram size (bytes)	fa3/0 bit rate (Mb/s)	fa3/0 datagram size (bytes)	time to fill OHQ (seconds)
1	1	64	1	64	1	64	(timeout)
2	4	120	2	100	1	64	(timeout)
3	4	120	2	150	3	64	< 1
4	4	120	3	150	1	64	< 1
5	4	120	3	200	1	64	< 1
6	4	120	3	400	1	64	(timeout)
7	4	120	3	250	1	64	4.5
8	4	120	3	270	1	64	12.5
9	4	120	3	280	1	64	94.7
10	4	121	3	280	1	64	19
11	4	120	3	280	1	65	(timeout)
12	4	120	3	281	1	64	(timeout)
13	5	201	3	303	1	64	(timeout)
14	5	500	2	303	1	64	6.0
15	5	500	2	303	1	103	(timeout)
16	5	500	2	303	1	70	22.2
17	5	500	2	303	1	71	46.3
18	5	500	2	303	1	72	(timeout)
19	5	501	2	303	1	71	37.2
20	5	500	2	304	1	71	54.6
21	5	500	2	305	1	71	64.2
22	5	500	2	306	1	71	78.6
23	5	500	2	306	1	71	106.4
24	4	128	2	128	1	64	(timeout)
25	4	128	3	192	1	64	< 1
26	4	128	3	192	0	-	(timeout)
27	4	128	3	192	1	128	(timeout)
28	4	128	3	192	1	96	2.1
29	4	128	3	192	1	112	(timeout)
30	4	128	3	144	1	112	< 1
31	4	64	2	64	1	64	(timeout)
32	4	64	2	64	2	64	< 1
33	3	64	3	64	2	64	< 1
34	3	64	3	64	1	64	(timeout)

35	3	128	3	64	2	64	< 1
36	3	128	3	128	2	64	< 1
37	3	128	3	128	2	128	< 1
38	3	128	2	64	2	128	< 1
39	3	128	2	128	2	128	< 1
40	3	196	2	128	2	128	2.2
41	3	196	2	128	1	64	(timeout)
42	5	501	2	306	1	71	56
43	5	500	2	307	1	71	101
44	5	500	2	307	1	72	(timeout)
45	5	500	2	308	1	71	151.3

Table 6.1: Cross traffic parameters used to attempt to generate a stable queue. Larger time to fill OHQ is better.

While a stable queueing configuration was not discovered, there were several interesting results from the experiment. While it was not known which parameters might result in stable queueing prior to the experiment, there were several configurations tested that were predicted to result in intermittent queueing. In any configuration with one or more identical cross traffic streams (trials 1, 32-34, 36, 37, 39, and 40), intermittent queueing ought to have been guaranteed. As long as the streams are started simultaneously, they should be immediately and perpetually in sync as they enter the router. The lack of intermittent queueing observed at any time during the experiment throws into the doubt the queue monitoring technique used by the router's internal computer. Because queue values are not published constantly, but once per 0.7 seconds, it is possible that the queue readouts in the router are an average of several queue scans taken during this period, rather than an instantaneous value. No documentation was found as to router's method for queue monitoring, so this is pure speculation.

As for general queueing trends, it was found that the most stable queues (ie. longest time to fill OHQ) were created from traffic configurations with combined bit rates approaching the 10Mb/s limit (a bitrate any larger will cause the queue to fill instantly) and medium-sized packets (between 200 and 500 bytes). Smaller packet sizes (64-100 bytes) combined with larger individual bitrates (3-5 Mb/s) resulted in a short time to fill OHQ (usually unmeasurably small), while a combined bit rate that was too small (< 6 Mb/s) was generally unable to generate any observable queuing before the timeout period.

6.3 - Testing the Probe-Gap Model With Symmetric Cross Traffic Streams

While a stable queueing configuration was not discovered through trial and error, there were several intermittent queueing configurations observable in the router simulation described in section 5.2. When the simulation was tested, it was found that symmetric cross traffic streams were an effective way to generate consistent intermittent queueing in the router. Directing two 3 Mb/s cross traffic streams with 100-byte datagrams at the simulated router resulted in the consistent intermittent queueing behavior shown in Figure 6.3.

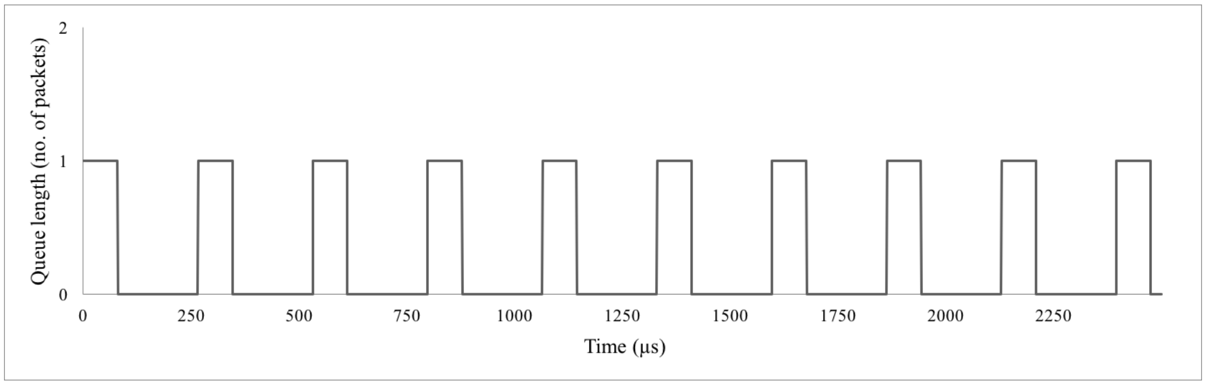


Figure 6.3: Output queue length of the simulated router while under symmetric cross traffic.

This makes intuitive sense, as, if the streams are in sync, their packets will collide every time they try to enter the router, causing regular queueing. Using this model, an experiment was designed to determine the efficacy of symmetric cross traffic for creating a queue measurable with the proposed queueing delay measurement scheme. The network used is the same as described in sections 4.1-3, except that the Client host is not used to send cross traffic packets. Only two Iperf client processes are utilized, originating from the Cross Traffic A and Cross Traffic B hosts, shown in Figures 4.1 and 4.2.

6.3.1 - Procedure

1. Connect four workstations to the Cisco 3640 router, as described in sections 4.1-3.
2. On the Server host, start Iperf using the command:
   ```
   iperf -s -u
   ```
3. Compile and run the server-side Java application, Server.java, on the Server host.
4. Prepare the following Iperf command on the Cross Traffic A and Cross Traffic B workstations:
   ```
   iperf -c 169.254.147.221 -u -b 2m -l 100 -t 1200
   ```
5. On the Client host, compile the client-side Java application, Client.java, and prepare the following command:
   ```
   java Client 169.254.147.221 500 22 1408 1500 64 0 10000
   ```
6. Initiate the two Iperf streams simultaneously
7. Run the Java Client application.
8. Record the average intra-probe gap results of each packet train, outputted from the Server.java process.

9. Close both Iperf processes.
10. Repeat steps 4-9 twice, changing the Iperf bitrate of the two streams to 3Mb/s, and then 4Mb/s. Finally, repeat steps 5, 7 and 8, running the application without any Iperf streams to determine the probing behavior without cross traffic.

6.3.2 - Results and Discussion

Without cross traffic, the intra-probe gap should be equal to estimated value given by Equation 3.6. This is then compared to the the intra-probe gaps measured in the router without cross traffic, shown in Figure 6.4:

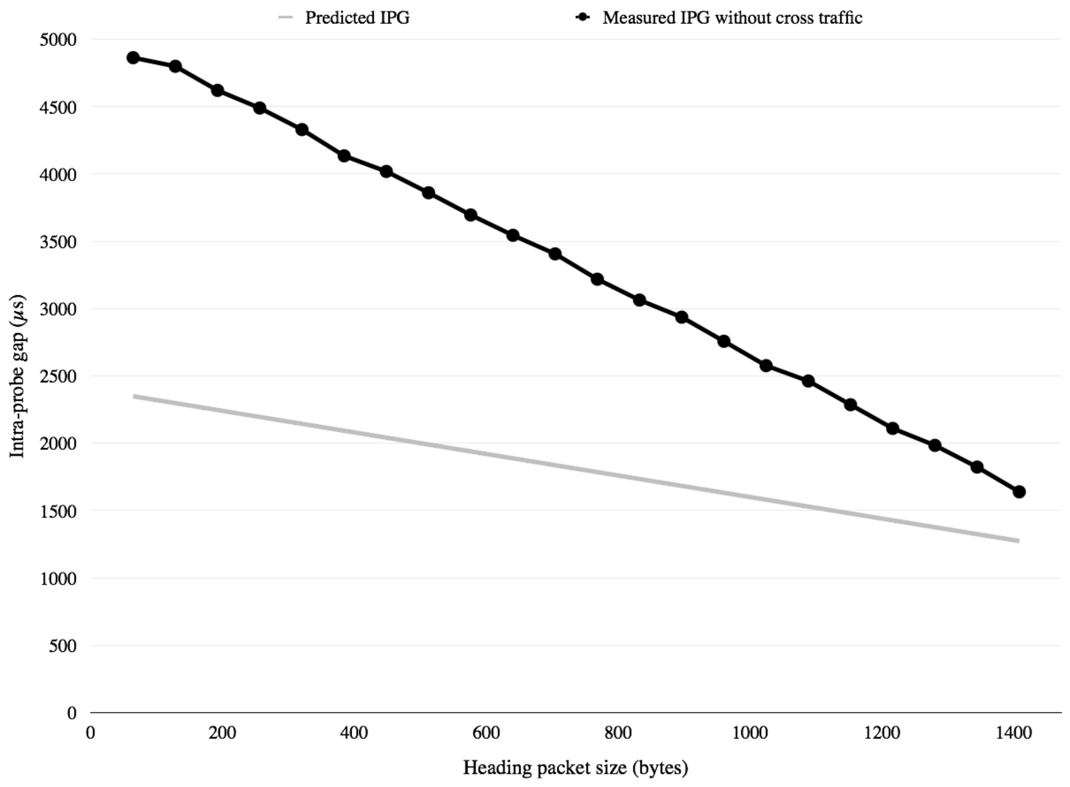


Figure 6.4: Predicted and measured intra-probe gaps for a compound probe composed of a heading packet of 64-1408 bytes, and a trailing packet of 1500 bytes. No cross traffic is present in the router.

Altogether, it appears seems that the intra-probe gap varies in the linear manner expected when cross traffic is absent from the router, though at different rate than the theoretical value. Despite the disparity between the theoretical and observed dispersion values, it is clear that dispersion is proportional to the difference in heading and trailing packet size.

Next, compare the intra-probe gaps measured without cross traffic, to those measured with varying configurations of symmetric cross traffic affecting the router, shown in Figures 6.5-7:

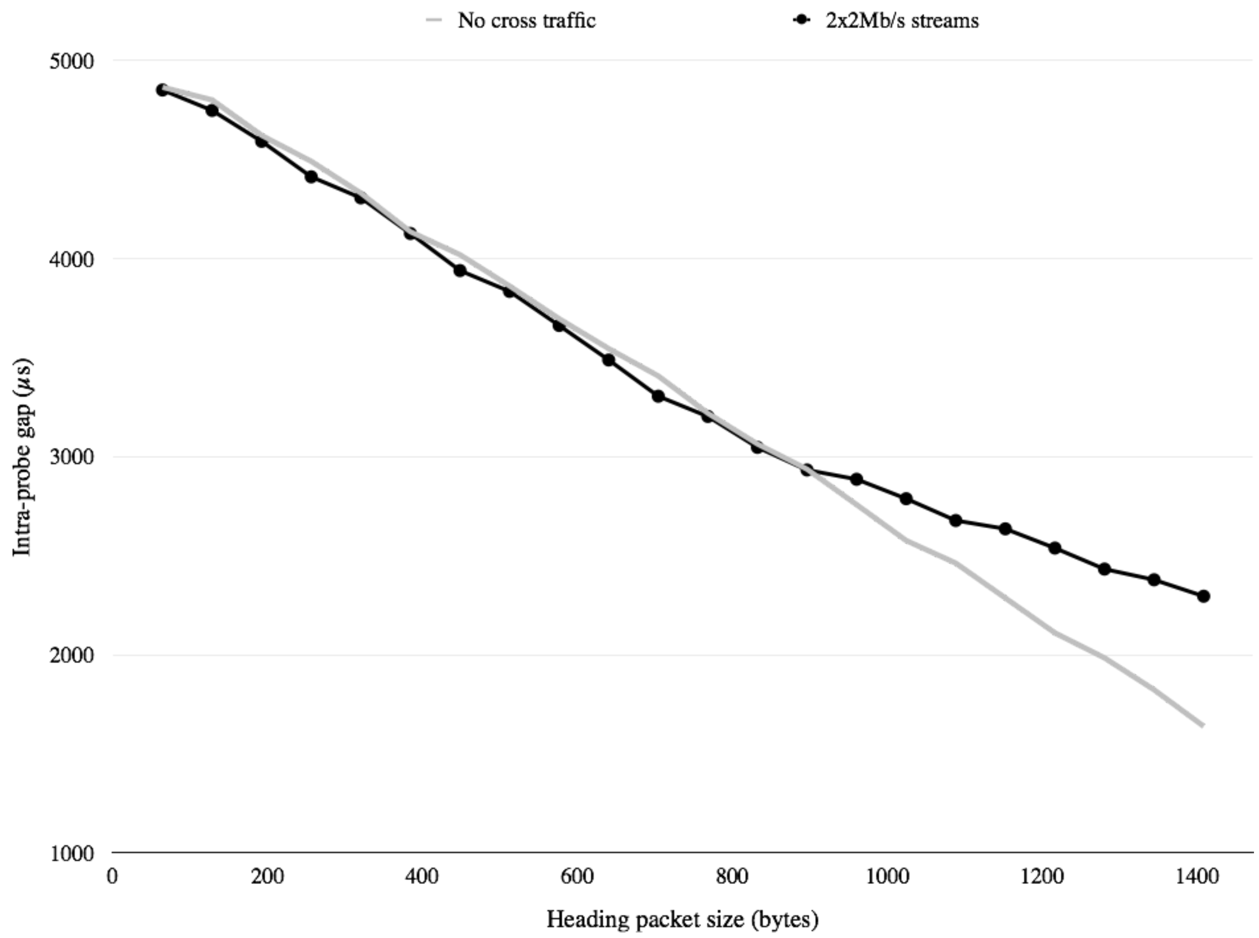


Figure 6.5: Measured intra-probe gaps for a compound probe composed of a heading packet of 64-1408 bytes, and a trailing packet of 1500 bytes, directed at a router under no cross traffic (grey) or under two cross traffic streams using 100-byte datagrams, sent at a rate of 2 Mb/s (black)

When the router is subject to two 2 Mb/s cross traffic streams, no dispersion reduction is observed, implying that no queueing delay is experienced by the packets in the router. However, as the size of the of the heading packet increases past 900 bytes (and the dispersion gap reduces by Equation 3.2), the IPG observed deviates above the value measured without cross traffic. This implies that enough cross traffic packets are being inserted between the probing packets that their dispersion is increased beyond the amount generated by passing through network interfaces along the path. This behavior is shown in Case 4 of Figure 3.4, and is to be expected as the dispersion gap between packets approaches zero. As the intra-probe gap between packets approaches the transmission time of the trailing packet, there is less space between the packets for cross traffic to get caught without increasing the dispersion gap in the probe.

Figure 6.6 shows the intra-probe gap distribution using two cross traffic streams with an average bitrate of 3 Mb/s, using 100-byte datagrams:

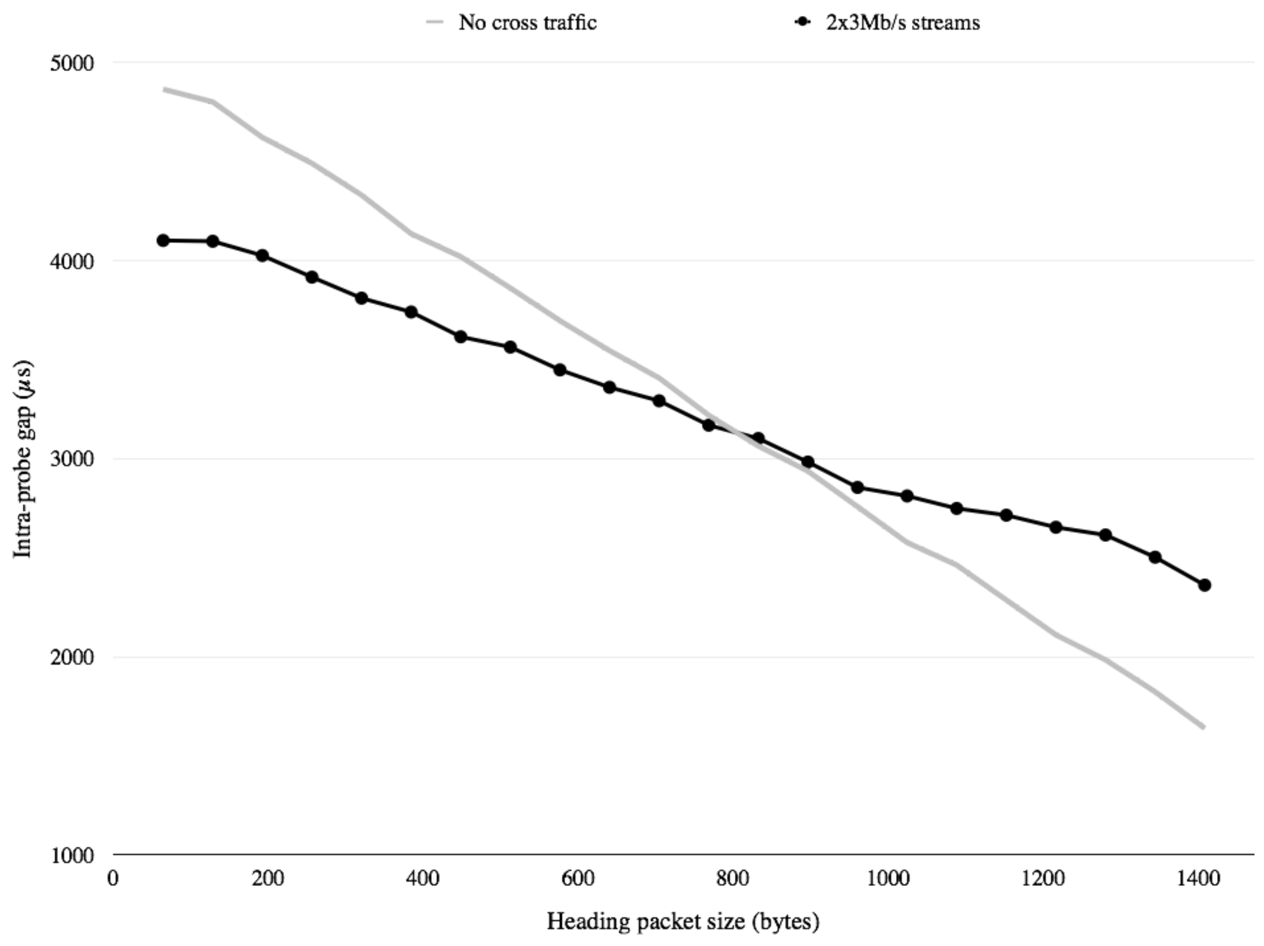


Figure 6.6: Measured intra-probe gaps for a compound probe composed of a heading packet of 64-1408 bytes, and a trailing packet of 1500 bytes, directed at a router under no cross traffic (grey curve) or under two cross traffic streams using 100 byte datagrams, sent at a rate of 3 Mb/s (black curve)

In this experiment, compound probes with a large trailing packet (1500 bytes) and a small heading packet (~800 bytes or less) experience significant dispersion reduction (equal to the space between the curves in Figure 6.6). The 64/1500 byte compound probe, for instance, sees a dispersion reduction of 763 µs. If we take this reduction to be equal to the queueing delay experienced by the heading packet, we can determine the number of packets ahead of the heading packet when it reaches the queue. Each 100-byte cross traffic packet has a transmission time of 80 µs (by Equation 3.1), implying that around 10 packets are ahead of the heading packet in the queue.

For compound probes with heading packets larger than 800 bytes, the intra-probe gap begins to increase relative to the control (ie. the traffic-free case), as it did in the 2 Mb/s cross traffic case. Once again, this implies that enough cross traffic packets are inserting themselves in between the probing packets to increase the dispersion gap between the two.

There is, however, a major caveat to these results: if the router is inspected via using its configuration terminal, it does not register any queueing at any point during the experiment. So either this dispersion reduction, while clearly caused by the presence of cross traffic in the router, is not actually the result of queueing delay, or, as speculated in section 6.2.3, it is possible that the router's queue monitor is not precise enough to pick up brief, intermittent queueing.

Figure 6.7 shows the intra-probe gap distribution using two cross traffic streams with an average bitrate of 4 Mb/s, using 100-byte datagrams:

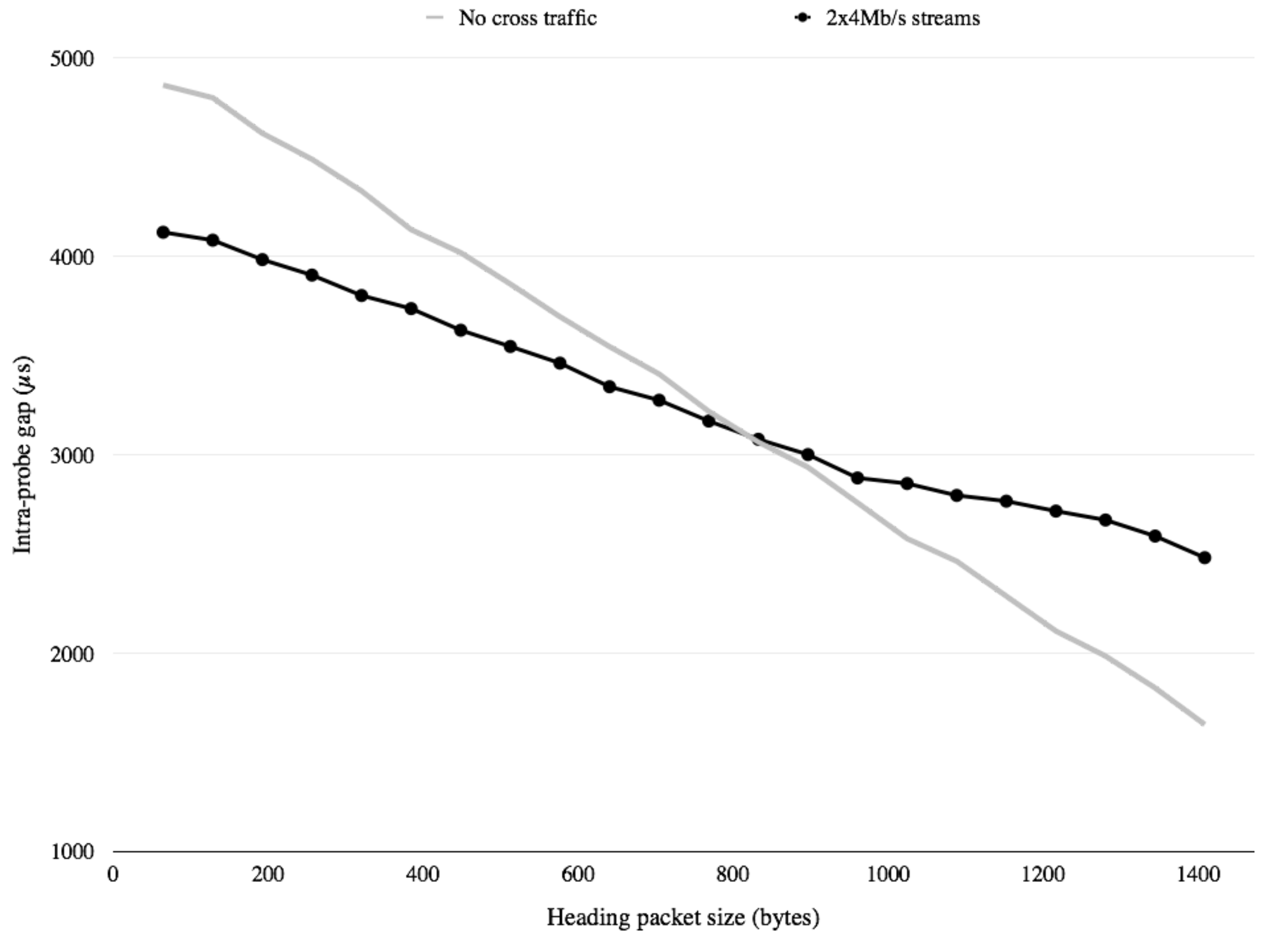


Figure 6.7: Measured intra-probe gaps for a compound probe composed of a heading packet of 64-1408 bytes, and a trailing packet of 1500 bytes, directed at a router under no cross traffic (grey curve) or under two cross traffic streams using 100 byte datagrams, sent at a rate of 4 Mb/s (black curve)

This trial results in an almost identical intra-probe gap distribution as the 3 Mb/s case. Probes with disparate heading and trailing packet sizes show a significant dispersion reduction, sometimes slightly more, sometimes slightly less than in the 3 Mb/s case. Once again, as the heading packet increases past ~800 bytes, interfering cross traffic packets increase the intra-probe gap past that of the control. Once again, no queue is visible in the router when monitored via the router console, making these results, for all three symmetric cross traffic configurations, ultimately inconclusive.

Besides the lack of queueing observed from the router console, another issue with the results in Figures 6.6 and 6.7 is the size of gap compression observed. If we examine the first probe configuration listed (consisting of a 64 byte heading packet and a 1500 byte trailing packet), in the case of 3 and 4 Mb/s symmetric cross traffic streams, there is significant intra-probe gap reduction. For both of these traffic configurations, around 760 μs of gap reduction is observed. A 100 byte cross traffic packet should require an 80 microsecond transmission time (by Equation 3.1), giving us an average of 9.5 packets in the output queue when the heading packet arrives. This is far more queueing than the router should ever be experiencing under two symmetric cross traffic streams at 3 or 4 Mb/s. These traffic configurations should never produce more than one packet of queueing at a time (as shown in Figure 6.3). This makes sense, as the total incoming bit rate (6 or 8 Mb/s) is significantly less than the capacity of the output link. Queueing is caused by packet collision, but never sufficient queueing such that a packet from one traffic stream should queue with the packet behind it. There should be sufficient capacity in the router’s output link to process any intermittent collision queueing before the next wave of packets arrives. As such, the large dispersion reduction observed cannot be adequately explained by the predicted intermittent queueing behavior.

6.4 - Simulation Evaluation

Leaving aside the difficulties experienced with the physical testbed, the simulation of the proposed scheme was also evaluated experimentally. To validate the application, simulated probing packets are first sent through the simulated router to see if they experience the expected dispersion behavior when using different heading and trailing packet sizes. As in Figure 6.4, Equation 3.6 is used to create a theoretical model of intra-probe gaps based on packet pair size differential. Figure 6.8 shows that the intra-probe gaps of simulated probes at the end of the path vary exactly as predicted by this equation:

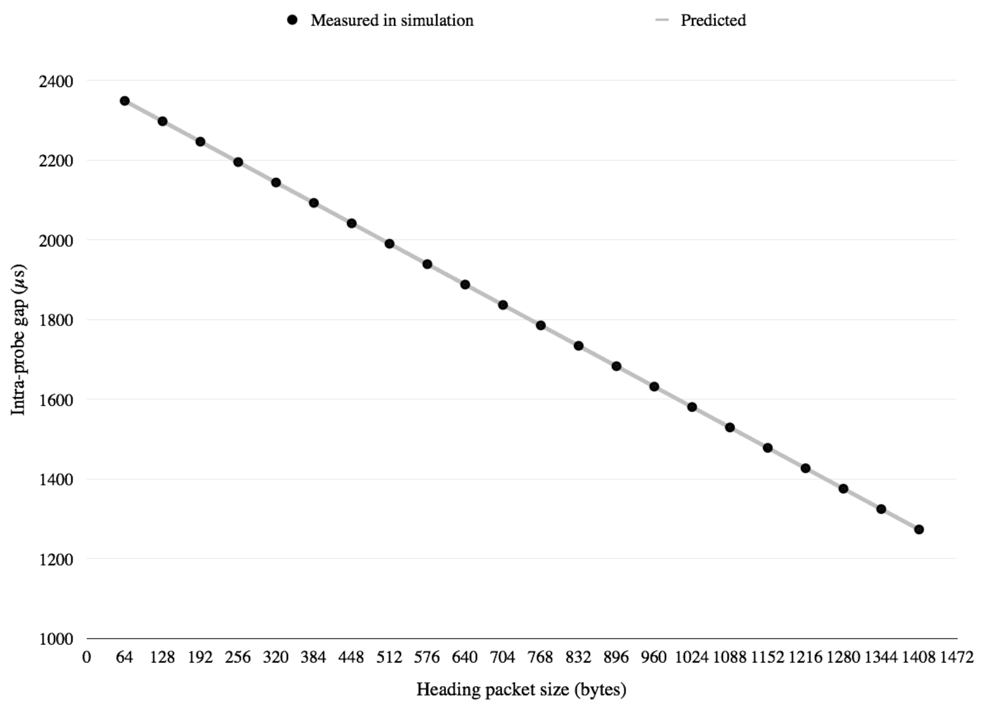


Figure 6.8: Predicted and measured intra-probe gaps in simulation, for a compound probe composed of a heading packet of 64-1408 bytes, and a trailing packet of 1500 bytes. No cross traffic is present.

Next, a simple test was devised to determine if this simulated probe-gap scheme can accurately predict queue length based on changes in the intra-probe gap. Because the simulated dispersion gap is identical to the theoretical value when traffic is absent, dispersion changes can be measured by comparing the theoretical value to measured value when cross traffic is used. Gap compression was measured with three cross traffic streams directed at the simulated router—one 4 Mb/s stream carrying 64-byte datagrams, one 2 Mb/s stream with 150-byte datagrams, and a 1 Mb/s stream with 100-byte datagrams. This cross traffic configuration produces intermittent collision-queueing over time as shown in Figure 6.9.

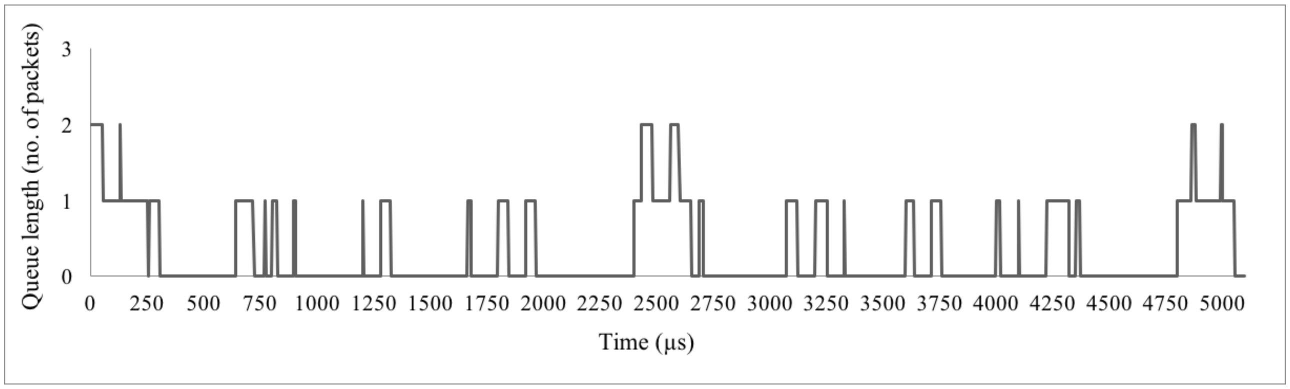
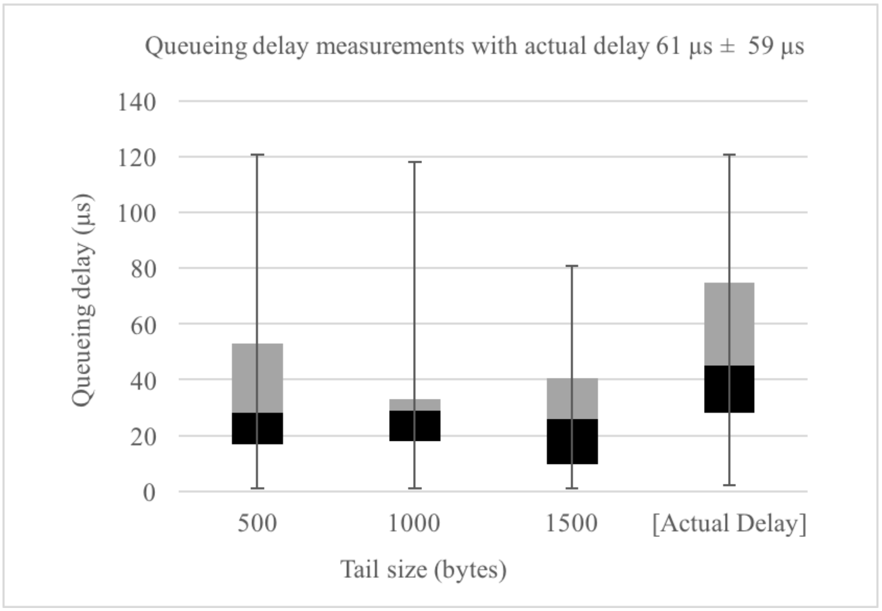


Figure 6.9: Queue length in the simulated router while under 3 asymmetric cross traffic flows.

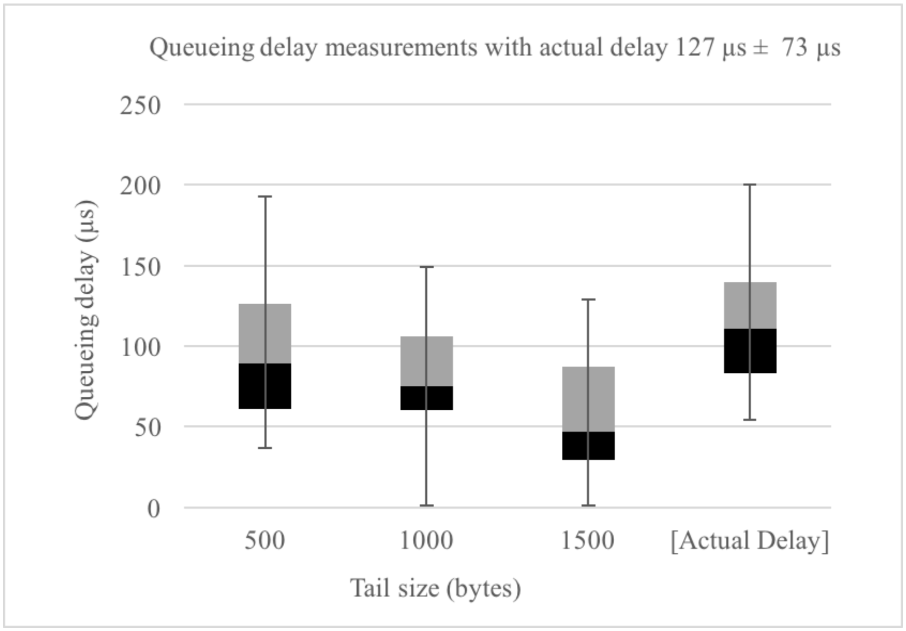
As this timeline shows, the queueing behavior under asymmetric cross traffic does not result in an obvious queueing pattern. Compare to Figure 6.3, where queueing follows a regular rhythm. Queueing is also intermittent—an output queue is only present for 28.7% of the simulation's duration. This means that the proposed scheme needs to be sensitive to small, brief queueing delays.

We evaluate the performance of the proposed scheme by sending 1000 pairs of probing packets from the simulated source host to the router, where each pair is separated by 10,000 µs ±

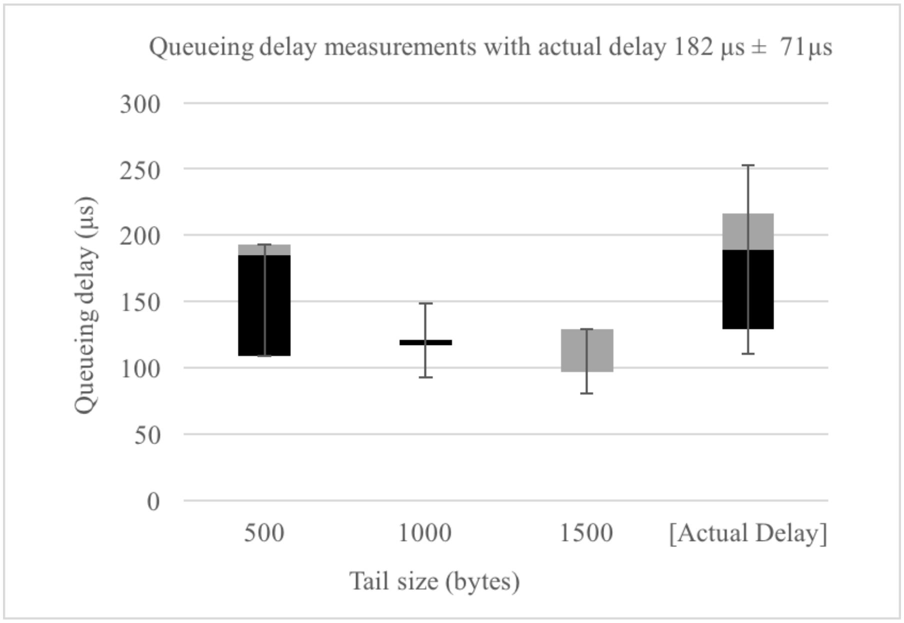
1000 µs. Figure 6.10 shows the results of the probes in the simulation, with queueing delays in the router of: (a) 61 µs ± 59 µs, (b) 127 µs ± 73 µs, and (c) 182 µs ± 71 µs. In Figure 6.10, the first three box-and-whisker plots of each graph represent queueing delays measured using the proposed scheme, using tail packets of 500, 1000, and 1500 bytes. The rightmost plot represents the actual queueing delay that the simulation is trying to measure. In each plot, the whiskers, represent the range of queueing delay, while the black and grey boxes represent the second and third quartiles respectively. The median value is the line between the black and grey boxes.



(a)



(b)



(c)

Figure 6.10: Queueing delay measurements in simulation, using varying tail packet sizes and asymmetric cross traffic streams, compared with the actual delay value.

The results in Figure 6.10 show that the simulation achieves high accuracy for the tested queueing delays. The range of the measured queueing delay, illustrated by the upper and lower whiskers, overlaps with the range of actual queueing delays in the simulation router in every instance. The error of each measurement is also calculated, shown in Table 6.2, which is the ratio between the true queueing delay value and the delays measured with the simulated probe. The accuracy of measurement is high in the majority of cases (usually 100%). The error observed is caused by cross traffic interfering with the trailing packet. Thus, for these small queueing delay values, compound probes with smaller tail packets were somewhat less prone to error. The smaller the intra-probe gap, the less time in which cross traffic can interfere. However, smaller trailing packets will generate less packet dispersion, and a probe cannot measure queueing delay that is larger than its dispersion gap. However, these results show that the probe gap model is sensitive to small amounts of queueing delay. Considering the constantly fluctuating value queueing delay shown in Figure 6.9, the simulated compound probes consistently achieve high measurement accuracy for different levels of delay.

Head size (bytes)	**Tail size (bytes)**	**Overall error**
64	500	21.05%
64	1000	24.33%
64	1500	33.29%

Table 6.2: Overall error percentage for each compound probe configurations under asymmetric cross traffic.

7 - Conclusions and Future Work

Without the ability to create a stable queue at the router's output interface that is measurable from the router console, it is difficult to draw meaningful conclusions from the results shown in section 6.3.2. While significant dispersion reduction was observed when using a heading packet size of less than 800 bytes and symmetric cross traffic streams of 3 or 4 Mb/s, the scheme cannot be validated without the ability to empirically monitor the length of the router's output queue. Any speculation as to the actual contents of the queue is pure guesswork.

As to the inability to create stable, or even intermittent queueing that is measureable, this brings up another set of issues. If two symmetric cross traffic streams are initiated simultaneously (as they were in sections 6.2 and 6.4) and directed at the same host, with the same incoming link speed, there is no clear reason that the router should not experience intermittent queueing. Indeed, cross traffic streams with an identical bit rate and packet size should by definition collide as they attempt to enter the output interface, and thus generate consistent intermittent queueing (as they did in simulation, shown in Figure 6.3). Yet, despite numerous trials, this was never measureable in test. While a possible explanation for this is the variability of propagation delay between the two hosts, it should not have been significant enough to prevent intermittent queueing in every case. In the case of two 4Mb/s, cross traffic streams, the spacing between traffic packets is only slightly greater than the transmission time of the packets themselves, making it extremely unlikely that propagation delay would prevent packet collision in every trial.

Leaving aside symmetric cross traffic, intermittent queueing was not observed in any of the 45 trials listed in section 6.2 (most of which used asymmetric traffic streams), at any point during their 10 minute lifespan. The only behavior that was observed was either an instantly full queue, a completely empty queue, or a queue that built slowly until filling. The fact that intermittent queueing never actually occurred in a single one of these trials is extremely unlikely. Many of these configurations used three asymmetric cross traffic streams, meaning that propagation delay cannot, by chance, cause packets to be permanently out of sync. Even in a configuration with a relatively small total bitrate, it simply does not make sense that random packet collisions would never occur. Ultimately, all of the queueing behavior observed via the router console runs counter to what was observed in simulation. In simulation, both symmetric and asymmetric cross traffic streams were able to generate consistent intermittent queueing (see Figures 6.3 and 6.9).

One possible explanation for this is that queue measurements from the router console are not an instantaneous value. Rather, it is possible that the router monitors the queue for several milliseconds, and reports the average value over that period. This is conceivable, as the router only reports queue measurements every 700 milliseconds, at the fastest—but as there is no documentation available as to the router's queue polling process, this is pure speculation. However, this would still serve the queue monitor's chief purpose, which is to troubleshoot output queue drops.[26] Stable or building queues would be clearly visible, but intermittent queuing of one or two packets would go undetected.

[26] "Troubleshooting Input Queue Drops and Output Queue Drops." *Cisco*.

Even if it is assumed that the router's queue monitoring procedure is designed in such a way that intermittent queueing is not observable (and this cannot be assumed), this is not the only issue with the results in section 6.4.2. Even if the expected queueing behavior were occurring, the size of the the dispersion reduction observed in Figures 6.6 and 6.7 is far too great to be caused by this intermittent queueing alone. The observed reduction was as high as 760 μs, but, with at most two 100 byte cross traffic packets ahead of the probe, it should never have been higher than 160 μs.

The proposed scheme was much more successful when tested in the simulated network. The simulated compound probes were shown to be sensitive to small queueing delays, even if they are brief and intermittent. While all of the compound probe configurations tested were by no means error-free, the majority of measurements taken using a 500-byte tail packet were accurate within one microsecond.

In the end this project raised many more questions than it answered. How to generate stable queueing in the router, why intermittent queueing is never observable, whether the router monitors queueing as an instantaneous value or as an average, and what causes the extreme dispersion reduction observed in the router under symmetric cross traffic. Finally, the question that this project aimed to answer in the first place remains untouched—whether or not the probe gap scheme for measuring queueing delay is practical for deployment on live networks.

Several approaches could be designed to attack these questions. If output queueing could be phrased as an equation of time, with the precise effects of each cross traffic parameter on the test network isolated, the equation could be balanced in such a way that stable queueing is attainable. However, as Table 6.1 shows, the behavior of the output queue is susceptible to extremely small changes in traffic behavior. Changing the size of traffic packets by a single byte can be the difference between a queue populating completely after a couple of seconds and the queue never filling at all.

As to the possible invisibility of intermittent queueing from the perspective of the router's console, the best way to test this theory would be to use a previously validated scheme for measuring queueing delay, such as pathchar or cing. Using the same cross traffic parameters and one of these schemes, it should be possible to observe the expected intermittent queueing behavior, assuming that it exists. Neither of these schemes are flawless measures of queueing delay, they should at least determine if there is any queueing at all. This could then confirm or deny whether probe-gap scheme as implemented has any merit when applied to live network traffic.

However, even if the probe-gap scheme was shown to be capable of detecting queueing, the results would still need to be validated as described in section 3.2. While most invalid results are easy to detect and filter (some of which are already filtered by the application implemented in this project), Case 3 of Figure 3.4 will require a statistical model to be created that determines the likelihood of traffic insertion between the probing packets, even in cases where a dispersion reduction was observed. This aspect was completely untouched by this project, as it is simply not relevant until it is certain that the implementation is capable of detecting queueing at all (of which, to reiterate, there can be no certainty). At the very least this project can hopefully serve as a jumping off point for further study of the probe gap scheme for measuring queueing delay.

References

Anagnostakis, K.g., M. Greenwald, and R.s. Ryger. "Cing: Measuring Network-internal Delays Using Only Existing Infrastructure." *IEEE INFOCOM 2003. Twenty-second Annual Joint Conference of the IEEE Computer and Communications Societies (IEEE Cat. No.03CH37428)*. Web.

Ayyorgun, Sami, and Wu-chun Feng. "A Deterministic Characterization of Network Traffic for Average Performance Guarantees." *The 38th Annual Conference on Information Sciences and Systems* (2004). Web. 30 Apr. 2016.

Brouer, Jesper Dangaard. "[net-next,2/5] Ixgbe: Increase Default TX Ring Buffer to 1024." *Patchwork*. 14 May 2014. Web. 30 Apr. 2016. <http://patchwork.ozlabs.org/patch/348793/>.

Cottrell, Les. "Passive vs. Active Monitoring." *Stanford Linear Accelerator Center*. 11 Mar. 2011. Web. 30 Apr. 2016. <https://www.slac.stanford.edu/comp/net/wan-mon/passive-vs-active.html>.

Crovella, Mark, and Balachander Krishnamurthy. *Internet Measurement: Infrastructure, Traffic, and Applications*. Chichester, England: Wiley, 2006. Print.

Demichelis, C., and P. Chimento. "IP Packet Delay Variation Metric for IP Performance Metrics (IPPM)." *IETF Tools*. Nov. 2002. Web. 30 Apr. 2016. <https://tools.ietf.org/html/rfc3393>.

Downey, Allen B. "Using Pathchar to Estimate Internet Link Characteristics." *Proceedings of the Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication - SIGCOMM '99* (1999). Web.

Gettys, Jim, and Kathleen Nichols. "Bufferbloat: Dark Buffers in the Internet." *Communications of the ACM* 55.1 (2012): 57-65. Web. 30 Apr. 2016.

"Interframe Gap and Spacing." *Savvius*. Web. 30 Apr. 2016. <http://www.wildpackets.com/resources/compendium/ethernet/interframe_gap>.

"Interrupt Coalescence at the NIC Layer." *IBM Knowledge Center*. Web. 30 Apr. 2016. <http://www.ibm.com/support/knowledgecenter/SSQPD3_2.6.0/com.ibm.wllm.doc/batchingnic.html>.

Jacobson, Van. "Pathchar — a Tool to Infer Characteristics of Internet Paths." Mathematical Sciences Research Institute, Berkeley, CA. 27 Apr. 1997. Web. 30 Apr. 2016. <ftp://ftp.ee.lbl.gov/pathchar/msri-talk.pdf>.

Kay, Rony, Ph.D. "Pragmatic Network Latency Engineering Fundamental Facts and Analysis." *Network Visibility & Test Solutions*. CPacket Networks. Web. 30 Apr. 2016.

Kozierok, Charles M. "Performance Measurements: Speed, Bandwidth, Throughput and Latency." *The TCP/IP Guide*. 20 Sept. 2005. Web. 30 Apr. 2016.

Kurose, James F., and Keith W. Ross. *Computer Networking: A Top-down Approach Featuring the Internet*. Boston, MA: Pearson Education, 2010. Print.

Lu, Wei-Zhou, Wei-Xuan Gu, and Shun-Zheng Yu. "One-way Queuing Delay Measurement and Its Application on Detecting DDoS Attack." *Journal of Network and Computer Applications* 32.2 (2009): 367-76. Web.

Marbach, Peter. "Queueing Delay." *University of Toronto Department of Computer Science*. Web. 30 Apr. 2016. <http://www.cs.toronto.edu/~marbach/COURSES/CSC358_S14/delay.pdf>.

Papagiannaki, K., S. Moon, C. Fraleigh, P. Thiran, and C. Diot. "Measurement and Analysis of Single-hop Delay on an IP Backbone Network." *IEEE J. Select. Areas Commun. IEEE Journal on Selected Areas in Communications* 21.6 (2003): 908-21. Web.

Postel, J. "Internet Control Message Protocol." *EITF Tools*. Sept. 1981. Web. 30 Apr. 2016. <https://tools.ietf.org/html/rfc792>.

Prasad, Ravi, Manish Jain, and Constantinos Dovrolis. "Effects of Interrupt Coalescence on Network Measurements." *Lecture Notes in Computer Science Passive and Active Network Measurement* (2004): 247-56. Web.

Rouse, Margaret. "What Is RMON (Remote Network Monitoring)." *SearchMobileComputing*. Nov. 2010. Web. 30 Apr. 2016. <http://searchmobilecomputing.techtarget.com/definition/RMON>.

Salehin, Khondaker M., and Roberto Rojas-Cessa. "Scheme for Measuring Queueing Delay of a Router Using Probe-Gap Model: The Single-Hop Case." *IEEE Communications Letters IEEE Commun. Lett.* 18.4 (2014): 696-99. Web.

Sudjiman, David. "Installing Minicom to Connect to Cisco Router Using Console Cable." *David Sudjiman*. 27 Mar. 2006. Web. 30 Apr. 2016. <http://www.davidsudjiman.info/2006/03/27/installing-minicom-to-connect-to-cisco-router-using-console-cable/>.

Sudjiman, David. "Router Output Queue." *David Sudjiman*. 20 May 2013. Web. 30 Apr. 2016.

"Troubleshooting Input Queue Drops and Output Queue Drops." *Cisco*. 28 July 2008. Web. 30 Apr. 2016. <http://www.cisco.com/c/en/us/support/docs/routers/10000-series-routers/6343-queue-drops.html>.

Appendix A - Compound Packet Probe Implementation

Client.java

```java
import java.io.IOException;
import java.net.InetAddress;

public class Client {
   private static UDPClient client;
   private static TCPClient control;

   public static void main(String[] args) throws IOException, InterruptedException {
       int header = 42;
       String address = args[0];
       int trainLength = Integer.parseInt(args[1]);
       int numTrains = Integer.parseInt(args[2]);
       int sH = Integer.parseInt(args[3])-header;
       int sT = Integer.parseInt(args[4])-header;
       int hDec = Integer.parseInt(args[5]);
       int tDec = Integer.parseInt(args[6]);
       int IPGmicro = Integer.parseInt(args[7]);
       InetAddress IP = InetAddress.getByName(address);
       client = new UDPClient();

       // calculate inter-probe gap
       int IPGnano = IPGmicro * 1000;
       int IPGmilli = IPGnano / 1000000;
       IPGnano = IPGnano % 1000000;
       System.out.println("Inter-probe gap: " + ((IPGmilli*1000)+(IPGnano/1000)) + "
microseconds.");

       // send control variables to server
       control = new TCPClient(address,6789);
       control.send(control.socket, numTrains);
       control.send(control.socket, trainLength);
       control.send(control.socket, sH);
       control.send(control.socket, sT);
       control.send(control.socket, hDec);
       control.send(control.socket, tDec);

       // primary experimental loop
       for (int i = 0; i < numTrains; i++) {
           for (int j = 0; j < trainLength; j++) {

               client.packetPairIPG(sH, sT, IP, 9876);
               Thread.sleep(IPGmilli,IPGnano);
           }
           // reduce packet size by the given decrements for each successive train
           sH -= hDec;
           sT -= tDec;
       }
       client.socket.close();
   }
}
```

Server.java

```
import java.io.IOException;
import java.io.PrintWriter;
import java.net.Socket;

public class Server {
   public static void main(String[] args) throws IOException {
       TCPServer control = new TCPServer(6789);
       UDPServer server = new UDPServer(9876);
       int header = 42, div = 1000;
       PrintWriter writer = new PrintWriter("AvgIPG.txt", "UTF-8");

       // main server loop, continually accepts new packet train series
       while(true){
           // receive control variables from client
           Socket connection = control.socket.accept();
           int numTrains = control.receive(connection);
           int trainLength = control.receive(connection);
           int sH = control.receive(connection);
           int sT = control.receive(connection);
           int hDec = control.receive(connection);
           int tDec = control.receive(connection);

           for(int i = 0; i < numTrains; i++){
               int sum = 0;
               int numValid = 0;
               System.out.print("\nHead " + (sH+header) + " bytes\tTail " + (sT+header) + "
bytes\t");

               for(int j = 0; j < trainLength; j++){
                   long IPG = server.packetPairIPG(sH,sT);
                   if(valid(IPG,sH,sT)){
                       sum += IPG;
                       numValid ++;
                   }
               }

               sum /= numValid;
               writer.println(sum);
               System.out.print("Intra-probe gap " + sum/div + " " + "micro" + "seconds");
               sH -= hDec;
               sT -= tDec;
           }
       }
   }

   public static boolean valid(long IPG, int sH, int sT){
       // estimated transmission time of the tail packet through a 10Mbps link
       int sTTrasmission = sT * 8 / 10;
       if(IPG < sTTrasmission){
           return false;
       }
       return true;
   }
}
```

UDPClient.java

```
import java.io.*;
import java.net.*;

public class UDPClient extends UDPSendReceive{

   public UDPClient() throws SocketException {
       socket = new DatagramSocket();
   }

   // sends two probing packets of given sizes to the server application at the given
address
   public void packetPairIPG(int sizeH, int sizeT, InetAddress IP, int port) throws
IOException {
       byte[]  headTx = new byte[sizeH],
               tailTx = new byte[sizeT];

       send(headTx, IP, port);
       send(tailTx, IP, port);
   }
}
```

UDPServer.java

```
import java.io.*;
import java.net.*;

public class UDPServer extends UDPSendReceive{

   public UDPServer(int port) throws SocketException{
       socket = new DatagramSocket(port);
   }

   public void closeServer(){if(socket != null) socket = null;}

   //returns time difference between two packet receipts, the "intra-probe gap", in
nanoseconds
   public long packetPairIPG(int sizeH, int sizeT){
       long intraProbeGap = -1;
       try{
           DatagramPacket head = receive(sizeH);
           long headRxTime = System.nanoTime();
           DatagramPacket tail = receive(sizeT);
           long tailRxTime = System.nanoTime();
           intraProbeGap = (tailRxTime - headRxTime);
       }
       catch (IOException err){
           System.out.println("Error establishing connection "+err.getMessage());
       }
       return intraProbeGap;
   }
}
```

UDPSendReceive.java

```
import java.io.IOException;
import java.net.*;

public class UDPSendReceive {
   DatagramSocket socket;

   // sends data to a specified address over a specified port
   public void send(byte[] data, InetAddress IP, int port) throws IOException {
       DatagramPacket packet = new DatagramPacket(data, data.length, IP, port);
       socket.send(packet);
   }

   // sends a single int to a specified address over a specified port
   public void sendInt(int n, InetAddress IP, int port) throws IOException {
       byte [] num = ByteUtils.intToBytes(n);
       send(num, IP, port);
   }

   // returns the next packet received of a given size
   public DatagramPacket receive(int size) throws IOException {
       byte[] data = new byte[size];
       DatagramPacket packet = new DatagramPacket(data, data.length);
       socket.receive(packet);
       return packet;
   }

   public DatagramSocket getSocket(){return socket;}

   public void setSocket(int port) throws SocketException {
       socket = new DatagramSocket(port);
   }
}
```

TCPClient.java

```
import java.io.*;
import java.net.*;

public class TCPClient {
   Socket socket;

   TCPClient(String address, int sock) throws IOException {
       socket = new Socket(address, sock);
   }

   // sends a given integer over the TCP connection
   public void send(Socket connection, int n)throws IOException {
       DataOutputStream out = new DataOutputStream(connection.getOutputStream());
       out.writeInt(n);
   }
   // returns an integer received over the TCP connection
   public int receive(Socket connection) throws IOException {
       DataInputStream in = new DataInputStream(connection.getInputStream());
       return in.readInt();
   }
}
```

TCPServer.java

```
import java.io.*;
import java.net.*;

public class TCPServer {
   ServerSocket socket;

   TCPServer(int sock) throws IOException {
       socket = new ServerSocket(sock);
   }

   // send an integer over the TCP connection
   public void send(Socket connection, int n)throws IOException {
       DataOutputStream out = new DataOutputStream(connection.getOutputStream());
       out.writeInt(n);
   }

   // returns an integer received over the TCP connection
   public int receive(Socket connection) throws IOException {
       DataInputStream in = new DataInputStream(connection.getInputStream());
       return in.readInt();
   }
}
```

Appendix B - Collision Simulation Implementation

TrafficSimulator.java

```
public class TrafficSimulator {
   int packetSize;
   int linkSpeed;
   int delay;
   int interPacketGap;

   public TrafficSimulator(int size, int speed){
       packetSize = size;
       linkSpeed = speed;
       interPacketGap = (packetSize*8)/linkSpeed;
       delay = 1;
   }

   // returns in bits the size of the frame that is put on the output link, or -1 if no such
frame is sent
   public int step(){
       delay --;
       if(delay == 0){
           delay += interPacketGap;
           return packetSize*8;
       }
       return -1;
   }
}
```

ProbeSimulator.java

```
public class ProbeSimulator {
   int headSize;
   int tailSize;
   int interProbeGap;
   int initialInterProbeGap;
   int intraProbeGap;
   int headDelay;
   int tailDelay;

   public ProbeSimulator(int hs, int ts, int interPG){
       initialInterProbeGap = interPG;
       headSize = hs;
       tailSize = ts;
       intraProbeGap = tailSize*8/10;
       headDelay = 1;
       tailDelay = intraProbeGap+1;
   }

   // returns in bits the size of the frame that is put on the output link, or -1 if no such
packet is sent
   public int step(){
       headDelay--;
       tailDelay--;

       if(headDelay < 1){
           interProbeGap =  initialInterProbeGap + (-1000) + (int)(Math.random() * 1000);
           headDelay += interProbeGap;
           return headSize * 8;
       }

       if(tailDelay < 1){
           tailDelay += interProbeGap;
           return tailSize * 8;
       }

       return -1;
   }
}
```

RouterSimulator.java

```
import java.util.ArrayList;
import java.util.LinkedList;
import java.util.List;
import java.io.FileWriter;
import java.io.IOException;

/*
   simulates a single output queue in a router, supporting up to three cross traffic streams
   also simulates the probe-gap model for measuring queueing delay
   each iteration simulates one microsecond
*/

public class RouterSimulator {
   public static final int TRAFFICID = 0;
   public static final int HEADID = 1;
   public static final int TAILID = 2;
   public static void main(String[] args) throws IOException {

       FileWriter writer1 = new FileWriter("simulationQL1.csv");
       FileWriter writer2 = new FileWriter("simulationQL2.csv");
       FileWriter writer3 = new FileWriter("simulationQL3.csv");

       LinkedList<List<Integer>> queue = new LinkedList<List<Integer>>();

       int OutputLC = 10;
       int elapsedTime = 0;
       int headSize = 64;
       int tailSize = 1500;
       TrafficSimulator ct1 = new TrafficSimulator(64,4);
       TrafficSimulator ct2 = new TrafficSimulator(100,1);
       TrafficSimulator ct3 = new TrafficSimulator(150,2);
       ProbeSimulator probe = new ProbeSimulator(headSize, tailSize, 10000);

       // dispersion gap plus transmission time of tail packet
       int IPGTheoretical = (tailSize*8/10) - (headSize*8/10) + (tailSize*8/10);
       System.out.println(IPGTheoretical);
       int last = 0;
       int queueSize = 0;
       int zeroTime = 0;
       int runs = 10000000;
       boolean IPGCounting = false;
       int IPGCounter = 0;
       int probeCounter = 0;
       int reducedIPGCounter = 0;
       int compressionTotal = 0;
       int trueDelayTotal = 0;
       int headQueueSize = 0;
       int headQueueDelayCounter = 0;
       int headQueueDelay = 0;
       double errorTotal = 0;

       // tracks the total amount of dispersion for each size of queue at heading packet
entry
       int[][] queueSizeCompressionTotals = new int[10][2];

       // set up CSV headers
       writer1.append("time (micoseconds),dispersion reduction,actual delay,packets ahead at
entry,\n");
       writer2.append("time (micoseconds),dispersion reduction,actual delay,packets ahead at
entry,\n");
       writer3.append("time (micoseconds),dispersion reduction,actual delay,packets ahead at
entry,\n");

//       Uncomment to write queue length to file instead of dispersion reduction
//       writer1.append("time (micoseconds),output queue length (number of packets)\n");

       for(int i = 0; i < runs; i++){
           // the first position in the queue represents the output interface
           // each member of the queue contains two values:
           // the amount of the packet that still needs to be processed (index 0)
           // and the total size of the packet (index 1)
           // heading packets in a probe include a third value: the size of the queue when
they first enter
           if(queue.size() > 0){
               int tempLC = OutputLC;
               while(queue.size() > 0 && tempLC > 0){
                   List<Integer> packet = queue.remove();
                   int dif = packet.get(0) - tempLC;
                   headQueueDelayCounter ++;


                   // if packet was not completely processed given the available link
capacity, put it back into the queue
                   if(dif > 0){
                       packet.set(0,dif);
                       queue.addFirst(packet);
                       tempLC = 0;
                   }

                   else{
                       // if it was a heading packet, begin measuring the intra-probe gap
                       if(packet.get(1) == HEADID){
                           IPGCounting = true;
                           headQueueSize = packet.get(2);
                           // actual queueing delay is the time it took from entering the
queue to leaving the queue, minus the transmission time of the head
                           headQueueDelay = headQueueDelayCounter - (headSize*8/10);
```

```
                        }
                        // if it was a trailing packet, finish measuring the intra-probe gap
                        else if(packet.get(1) == TAILID){
                            if(IPGCounter < IPGTheoretical && headQueueSize > 0){

                                System.out.println(elapsedTime + " microseconds, dispersion
reduction: " + (IPGTheoretical - IPGCounter) +
                                        ", actual queueing delay of head: " + headQueueDelay
+ " packets ahead at head entry: " + headQueueSize);


                                // output to appropriate CSV file based on queue length
                                if(headQueueSize == 1) writer1.append(elapsedTime + "," +
(IPGTheoretical-IPGCounter) + "," + headQueueDelay + "," + headQueueSize + "\n");
                                if(headQueueSize == 2) writer2.append(elapsedTime + "," +
(IPGTheoretical-IPGCounter) + "," + headQueueDelay + "," + headQueueSize + "\n");
                                if(headQueueSize == 3) writer3.append(elapsedTime + "," +
(IPGTheoretical-IPGCounter) + "," + headQueueDelay + "," + headQueueSize + "\n");

                                errorTotal += Math.abs(headQueueDelay - (IPGTheoretical -
IPGCounter)) / (double) headQueueDelay * 100;

                                queueSizeCompressionTotals[headQueueSize-1][0] ++;
                                queueSizeCompressionTotals[headQueueSize-1][1] +=
IPGTheoretical - IPGCounter;

                                reducedIPGCounter++;
                                compressionTotal += IPGTheoretical - IPGCounter;
                                trueDelayTotal += headQueueDelay;
                            }

                            probeCounter++;
                            IPGCounting = false;
                            IPGCounter = 0;
                        }
                        // continue loop with the remaining link capacity
                        tempLC = dif;
                    }
                }
            }

            if(IPGCounting) IPGCounter++;

            int packet1 = ct1.step();
            int packet2 = ct2.step();
            int packet3 = ct3.step();
            int probePacket = probe.step();

            if(packet1 > 0){
                List<Integer> p1= new ArrayList<Integer>();
                p1.add(packet1);
                p1.add(TRAFFICID);
                queue.add(p1);
            }

            if(packet2 > 0){
                List<Integer> p2= new ArrayList<Integer>();
                p2.add(packet2);
                p2.add(TRAFFICID);
                queue.add(p2);
            }

            if(packet3 > 0){
                List<Integer> p3= new ArrayList<Integer>();
                p3.add(packet3);
                p3.add(TRAFFICID);
                queue.add(p3);
            }

            if(probePacket > 0){
                List<Integer> pp= new ArrayList<Integer>();
                pp.add(probePacket);
                if (probePacket == headSize*8){
                    pp.add(HEADID);
                    pp.add(queue.size()); // add the current size of the queue when the
heading packet enters
                    headQueueDelayCounter = 0;
                }
                else pp.add(TAILID);
                queue.add(pp);
            }

            queueSize = queue.size() > 0 ? queue.size() - 1 : 0;
            if(queueSize == 0) zeroTime++;

//          Uncomment to print queue size changes (and output to CSV)
//          if(queueSize != last){
//              writer1.append(elapsedTime + "," + queueSize + "\n");
//              System.out.println(elapsedTime + " microseconds, OHQ = " + queueSize);
//          }
//          last = queueSize;

            elapsedTime++;
        }

        double percentZeroTime = (double)zeroTime/runs * 100;
        System.out.println("percentage of time with zero queue: " + percentZeroTime);
        System.out.println("Intra-probe gap reduced for " + reducedIPGCounter + " out of " +
probeCounter + " probes.");

        if(reducedIPGCounter > 0){
            System.out.println("Average compression: " + compressionTotal/reducedIPGCounter +
", average true delay: " + trueDelayTotal/reducedIPGCounter);
        }

        for(int i = 0; i < queueSizeCompressionTotals.length; i++){
            if(queueSizeCompressionTotals[i][0] > 0){
                System.out.println("Average compression for queue of length " + (i+1) + ": "
+ queueSizeCompressionTotals[i][1] / queueSizeCompressionTotals[i][0]);
            }
        }

        System.out.println("Error: " + errorTotal/reducedIPGCounter + "%");

        writer1.flush();
        writer1.close();
        writer2.flush();
        writer2.close();
        writer3.flush();
        writer3.close();
    }
}
```