Universitatea Politehnica Bucuresti
Facultate de Automatica si Calculatoare
Departamentul de Calculatoare

Masurarea capacitatii in retele de mare viteza. Integrarea in MonALISA

Student: Stefan-Cosmin Zota
Coordonator stintific: prof. dr. Ing. Valentin Cristea

Colaboratori: prof. dr. Ing. Fiz. Iosif Legrand
as. Catalin Carstoiu

Preface

Abstract

My diploma thesis deals with measuring end-to-end capacity for network links. This project is part of the MonALISA framework – complex dynamic service monitoring system. This document will describe the mechanisms and methods implemented for capacity estimation as well as the module structure and integration in the MonALISA framework.

Motivation

The Internet is changing rapidly. One of the consequences of this change is a growing need for higher quality of service. A corollary of the need for higher quality of service is the need for accurate and extensive measurement, including the need to measure bandwidth.

Currently the most accurate bandwidth measurement techniques are to directly measure the fastest rate that traffic can be sent through a network. Wide-scale deployment of these “heavy-weight” bandwidth tests can overwhelm the network with test traffic. Accurate measurement of bandwidth is difficult if simple large data volume techniques are not used but there is some current research in this area. Existing techniques attempt to estimate the capacity and bandwidth of both links and paths while attempting to use as small a quantity of data as possible. These techniques must operate from only the end points of a connection, and must not require specialist software be deployed into the core of

the network. Therefore the need for estimation tools is growing in the landscape of twenty – first century.

Overview of the Thesis

This thesis is structured in **eleven** chapters. This an overview of the contents of each chapter.

Chapter 1 - *Introduction*: In this chapter, we will discuss the importance of measuring network links. I will also provide a short historical of estimation tools categories.

Chapter 2 - *Basic notions for measuring network links*: This chapter includes a brief description of the major network parameters. The definitions from this chapter will help the reader to further understand the algorithms presented.

Chapter 3 – *Existing techniques*: In this chapter, I will clasify the existing techniques for measuring bandwidth and capacity. We will also provide an exhaustive list of the existing application with a few important traits.

Chapter 4 – *MonALISA. Monitoring Agents using Large Integrated Services Arhitecture*: An overall description of the MonALISA framework is provided. I will concentrate more on the existing bandwidth measurement module ABPing.

Chapter 5 – *ADR. Measuring end-to-end capacity*: This chapter covers all the aspects of the Average Dispersion Rate technique for capacity estimation. At the end of the chapter I will explain the methodology implemented in the ADR tool.

Chapter 6 – *Java NIO*: This chapter presents the important characteristics

of the new Java 1.4 IO library. I will focus on the classes and methods used in the application and underline the choice of using them.

Chapter 7 – *Technical specification - the architecture of the application*: In Chapter 7 it is given a full description of the implementation. I will go to the low level of application in the essential parts of the algorithms.

Chapter 8 – *The structure of the MonALISA module cap*: In this chapter I will describe the incorporation process in the MonALISA framework. Also I will detaliate the structure of the *cap* module, the interaction of the user with my application.

Chapter 9 – *Performance*: I will show the results obtained using this tool on Ethernet and GigaByte links. I will draw some concusions and define the future improvements.

Chapter 10 – *References*: A complete list of references.

Chapter 11 – *Code listing*: A few pages of essential code listed for future reference.

Credits

This diploma thesis was developed with the help of my advisors: prof. PhD. Vice Dean Valentin Cristea, prof. PhD. Fiz Iosif Legrand and as. Catalin Carstoiu. I would like to thank also the people who worked beside me in the Caltech laboratory for their advices, patience and most of all friendliness. I hope I will work in this environment the following years.

Contents

Chapter 1 - Introduction

1.1. The importance of measuring network links

As long as Internet bandwidth has increased, the amount of traffic sent over the Internet has grown to consume it. This means that despite the increasing link bandwidth in network backbones and into homes and offices, optimizing the use and allocation of bandwidth continues to be an interesting problem. Although many applications are more interested in available bandwidth than link bandwidth, knowing the link bandwidth along a path enables more accurate measurement of available bandwidth. In addition, several applications can directly use link bandwidth, including planning networks to minimize bottlenecks and analyzing network performance as a whole.

However, the Internet's current size, heterogeneity, and rate of change make determining link bandwidth a challenging research problem. This is true even though applications are usually only interested in the bandwidth along a particular path or even just the smallest bandwidth (the bottleneck bandwidth) along that path. A database to store bandwidth information would neither scale well nor cope with the rate at which routes change. Routers currently do not report link bandwidths. Since routers gain much of their speed by being as simple as possible, slowing them to answer link bandwidth queries is probably not acceptable. The easiest approach to deploy, and consequently the one in which we are most interested, is for end hosts to infer link bandwidth by actively probing

or passively listening to traffic. Hosts can share this information if the probing or listening is expensive.

With the increase in use of the Internet, more people are finding themselves dependent on it. Just as happened with the telephone system early last century, business and people are finding that the requirement of Internet for communication and gathering of information is something they cannot operate without. Increasingly more and more business models are based solely around the Internet.

However with this growth of dependency and use of the Internet, more and more demands are being placed on the performance of the network. Users require that consistent monitoring of the performance is carried out, in order to both detect faults quickly and predict and provision for the growth of the network.

Measuring the Internet is difficult some of the reasons for this are described below:

- Not all ISP's are forthcoming about details of the loading and performance of their network.
- Even with the support of the ISP, the complexity of the network means that normally multiple providers are involved in the end-to-end connection between hosts. This situation makes the monitoring of end-to-end performance by any one ISP nearly impossible.
- The servers and ISP's try to limit the traffic in the best way they can. A network administrator will not let ordinary people to flood a connection just to know the capacity of a particular path.
- In general there are big variations in small time stamps of the

network parameters. This means that we should use a twenty four hour day servers for monitoring network links. A plethora of problems need of expensive hardware.

- Detecting invisible network nodes (intelligent hubs, updated routers). These nodes will drop packets suspected to make Denial of Service attacks or act different with some packets.

One possible way to meet this need would be the deployment of special software or hardware on each router in the network. A solution such as this, however, is just not practical. The cost, time and security problems with this outweigh the gains from this type of instrumentation. The cost of this solution is involved in the man hours spent upgrading software on all of the routers in the network, the charges for this software by the vendor, and the price to upgrade older routers that are unable to run this software. This sort of upgrade is also not going to be instantaneous. The time required for upgrading the software on every router in the entire network would be huge. This would leave a substantial time where there are inconsistencies in the network when it may be possible to measure some of the paths, and not others.

An alternative approach is to use end-to-end software run on the end hosts. This allows the measurement to be run at the user's discretion and allows for simple deployment. However this approach requires the software to infer the characteristics of the links involved without being able to directly measure each link individually.

This realm is heading into end-host programs for estimating capacity and bandwidth. The results obtained with these tools are of huge significance for scientific applications in nuclear physics, astronomy and biotechnology. In a few

years, the scientists from all around the world will be connected by Gigabyte links. We need research projects like Grid Physics Network (GriPhyN) or Particle Physics Data Grid (PPDG). Surely we will be forced to invest in very expensive switches to assure a terrabyte transfer rate (for example a on a 622MBps link which is used in a data-intensive application we reach a troughput of several duzins of gigabytes, no matter the capacity of the network).

The capacity, bandwidth and troughput cuantify the transfer in a network. At the Application and Network levels, the existing protocols lack the ability to anticipate the troughput of a network path, that is why Transport level protocols such as TCP/IP try to determine adaptively and dinamically the maximum transfer rate. TCP uses techniques like *congestion avoidance* or *slow start.* Unfortunately the TCP generates a large quantity of traffic. We can avoid this downpoints by shifting to UDP and ICMP packets. The last two protocols adapt very quickly to the instability of real networks and most of all do not need to keep open a conection between the sender and the receiver (thousands of open conncetions cause congestion).

1.2. The history of measuring tools

Estimation tools for network properties have changed substantially in the last twenty years. In this period, we can distinguish three genereations:

1. ***First generation***(the 80's and the beggining of 90's): We have seen the first attempts for network managements. Programs like *ping, traceroute or ttcp* are familiar to everyday users. To recognize their impact, some of them have been included in operating systems. They made a low level monitoring and

debugging, determining round trip times, packet loss rate, hop number or bulk TCP throughput.

2. ***Second generation***(the mid 90's): In this decade several monitoring large scale activities have started such as PingER (SLAC), NMI (LBNL, PSC, ACIIR), Surveyor (Advanced Networks), AMP (NLANR) and so on. Numerous hosts have been used for testing these new frameworks. N probes can scan bidirectionally N*N network paths. The major disadvantage is that the results cannot correlate and cannot capture the network performance. Therefore decisive actions could not be taken starting from the obtained data. Moreover these data were available only to the user, other Session or Applecation level protocols could not use them.

3. ***The third generation****(late nineties and 21stcentury)*: The next generation of monitoring tools will inferr available bandwidth (maximum permitted troughput) or capacity (maximum possible troughput). We can get other interesting values like: jitter, cross traffic percents. The data can be used by middleware network applications and even high level applications in order to optimize the end- to-end transfer.

Chapter 2 - *Basic notions for measuring network links*

2.1. Definitions

Like most specialist research areas, bandwidth measurement and estimation has many specific terms that need to be explained. Unfortunately many of these terms are used differently by different authors. This section clarifies the main terms, and defines how we will use them in this paper.

We begin with the names for the components of a network. *Hosts* are the end points from which a packet either originates from or is destined to. A *router* is a machine with two or more network connections that forwards packets from one connection to another that will get the packet closer to its destination. A *link* refers to a single connection between routers or routers and hosts. A path is the collection of links, joined by routers, that carries the packets from the source to the destination host. Two paths are different if any intermediate router is different.

Link latency is the time it takes from the time the first byte of a packet is placed on the medium until the time that the first byte is taken from the medium. This delay is caused by the rate at which signals are propagated in the link (e.g. electrons in a cable) and distance of the medium.

The *link bandwidth* is the rate at which bits can be inserted into the medium. The faster the bandwidth the more bits can be placed on the medium in a given time frame.

An important thing to note about computer network routers is that they are

normally store-and-forward routers. This means that every byte of the packet must be received from the link and placed into a buffer in the router before the router will start to send it out on the destination link. If packets arrive at a router faster than they can be sent out the appropriate output port a packet queue will form for this port. The queue discipline used is almost always a FIFO queue.

In the next sections we go into more detail about the different variations of the terms latency and bandwidth.

2.1.1. Latencies

There is much separate latency in a network system. Each of these values is used throughout this paper so we describe them all here.

The *transmission delay* is the time it takes a packet to be placed on the medium. This time is proportional to the packet size and the bandwidth of a link. It is the time from the time the first byte is placed on the network until the time the last byte has been sent.

The *transmission time* is considered to be the combination of link latency and transmission delay. The transmission time is the time between the first byte being placed on the medium and the last byte being taken off. This is the sum of the link latency and the transmission delay.

The *path latency* is the sum of all of the individual transmission times as well as the queuing time inside the routers. This is the time that it takes from the sender issuing the packet until the destination receiving it. Path latency is often referred to as the one-way delay.

Path latency is hard to measure as it requires a distributed synchronized

clock to timestamp the time the packet was sent and the time the packet was received. The clocks need to be synchronized as the sending time will be time stamped from the sender's clock and the reception timestamp from the destination machines clock.

A more common measurement is the *round trip time* (RTT) latency. This is the sum of the path latency in the forward and reverse directions, and can be measured easily by timing the sending of a packet, have the destination machine respond to the packet immediately and the original sender timestamp the return of this packet.

Unfortunately as the forward and reverse paths and path latencies may differ, the RTT latency cannot differentiate between the forward and reverse delay.

2.1.2. Bandwidths

As with latency, there are many variants of the term bandwidth. This section will discuss these.

Unlike latencies, *link bandwidths* do not sum to result in the path bandwidth. The *path bandwidth* is defined by the minimum of the link bandwidths, as this is the fastest any traffic can make it through the path. The path bandwidth is also known as the path capacity.

The bandwidth of a path is shared by the traffic under consideration and other traffic. This reduces the amount of bandwidth available to the hosts. This other traffic is referred to as cross traffic.

Available bandwidth is the amount of bandwidth "left over" after the cross

traffic. The link with the lowest available bandwidth will not necessary is the link with the lowest capacity.

Dovrolis refers to the link that restricts the paths capacity as the narrow link and the link that restricts the available bandwidth the tight link.

2.2 Assumptions

A large amount of money and time is currently being spent implementing high speed, next generation networks. These networks are being constructed in-order to support the large growth in the Internet, as well as enabling higher bandwidth services to run over the network to more people. There is an increasing demand to know if the performance obtained from these networks is what is expected from them. The performance of a network is a complicated issue with many variables effecting different traffic in different ways. While poor values for some performance metrics may only have a strong effect on the performance of a small number of applications, there are a few metrics which are almost universally significant for all types of traffic. Bandwidth and latency are the two most commonly quoted of these performance metrics.

Most of the routers are store and forward and usually the transmission delay is proportional to the packet size. This means that they have to receive the whole packet before they can forward it to the next interface. The standard ordering mechanism is FIFO. The latency of the path is the sum of latencies of every link along the path, as opposed to bandwidth which is the minimum bandwidth for all links. This slow link determines the rate of transfer for packets because it dictates the maximum transfer capacity.

The bandwidth of a path is divided between the probing traffic and the cross-traffic. The cross-traffic reduces significantly the bandwidth of every path. We can derive the available bandwidth to be the bandwidth obtained after excluding the cross-traffic. The capacity of the path is determined by the link with the lowest transmission rate. This slow connection is a *narrow link* for capacity and a *tight link* for bandwidth estimation.

The presence of layer-2 store-and-forward devices causes consistent errors in per-hop measurement tools such as *pathchar* and *pchar*. Also, not all routers along an end-to-end path are equal: internal switches, different buffer configurations, and the relegation of classes of messages typically employed by tools to a router's "slow path" all add error into the measurements. Furthermore, tool accuracy tends to deteriorate on heavily loaded or high bandwidth paths where traffic characteristics differ from tool assumptions or network interface interrupts coalesce. Attempts to increase accuracy often require tools to saturate the path with measurement probes, a method that is inefficient and does not scale.

Chapter 3 - Existing techniques

3.1. A classification of existing techniques

There have been a number of techniques proposed in the area of bandwidth estimation. Most concentrate on measuring one of two values, either the individual link bandwidths of a path, or the capacity of a path. In general these techniques can be classified into two groups. *Single packet* and *packet pair* techniques. The names refer to the number of packets that are used in a single probe. A measurement of a link or path will consist of multiple probes, in the case of some implementations, this can be in the order of 10MB of data (14400 individual probes) to measure a 10 hop path. The following sections will detail the theory of these techniques, improvements suggested and example implementations.

3.1.1. By the number of probe packets

3.1.1.1. „Single Packet" or „One packet"

Single packet techniques concentrate on estimating the individual link bandwidths as opposed to end-to-end properties. These techniques are based on the observation that slower links will take longer to transmit a packet than faster links. If it is known how long a packet takes to cross each link, the bandwidth of that link can be calculated.

Calculations must also take into account the latency, which varies for each

link. As discussed in, latency is not dependent on the packet size or the bandwidth of that link, but the time that the signal takes to travel down the path. The transmission time of a packet is determined by the packet size (P), the bandwidth of the link (b) plus a fixed latency value (l).

$$t = \frac{P}{b} + l \quad (1)$$

If the time and packet size are known equation (1) can be rearranged to give the bandwidth. As latency is fixed for a particular link, latency can be considered as a fixed offset. When the transmission time for multiple, varied sized probe packets are taken, a graph such as Figure (1) can be produced.

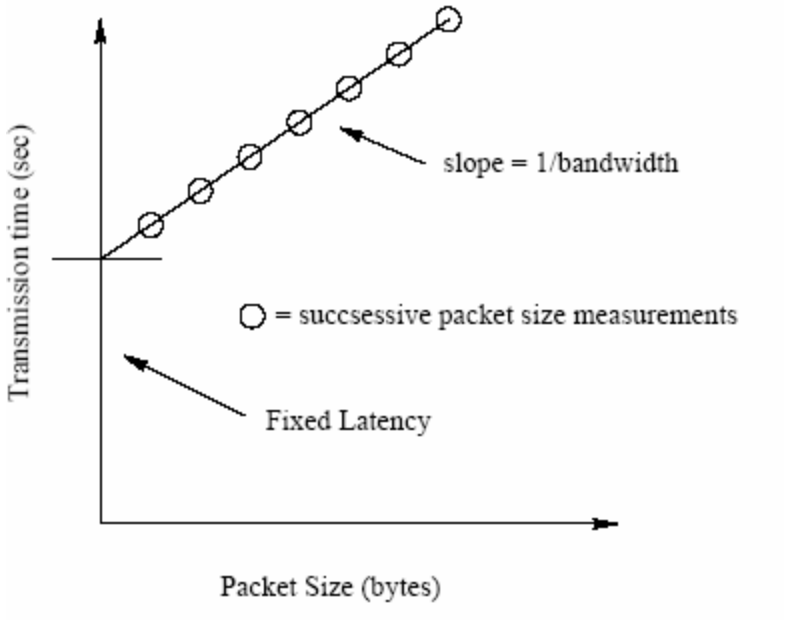

Figure 1: Calculating the slope of this graph forms the basis of how single packet techniques estimate bandwidth.

Each probe packet is plotted using packet size versus transmission time. The bandwidth can be calculated from this graph by performing a linear

regression to find the slope of the points, and the inverse of this value is the estimated bandwidth.

There are addition problems when conducting these measurements in the real world. The issue of measuring the time it takes a packet to cross each link in the network path separately is the first hurdle.

To avoid the need for special router instrumentation, single packet implementations take advantage of the IP time to live (TTL) field. This field is decremented by one by each router the packet passes through. Once the TTL is decremented to zero, the packet is discarded and the router will send an ICMP TTL expired error message to the original packet source. By setting the TTL to expire at the router at the end of the link to be measured, the sender can record the RTT of the packet to the end of the link and back by recording the sent time of the packet and the return time of the ICMP error message.

The TTL for the measurement packet is set to 2 as it leaves the machine. The TTL is decremented to 1 in the first router, and 0 in the second router. This router then generates an ICMP error message to be returned to host A. The error message may or may not follow the same path to return to host A for this reason is displayed as a dashed line. The effect of the return path differing from the forward path will be discussed later in this section.

Although the use of the TTL field allows measurements to be made from the end points without special software deployed in each router in the path, it also causes a number of problems. First there is no way of considering any link (with the exception of the first link in the path) independently. The measurements reported for all but the first link must also include all the effects of the previous links.

This problem is solved by representing a link as a sum of all the results for the previous links, plus the component of this link. The latency for a single link was given in equation.

Summing the links we derive:

$$t_x = \frac{P}{b_x} + l_x + \sum_{i=0}^{x} t_i \quad (2)$$

This means that the latency of the link being measured can be calculated by subtracting the latency measured on the previous links. The bandwidth of the link is calculated by subtracting the slope of the previous links from the slope of this link and taking the inverse of this value.

This process has the disadvantage that errors accumulate over each links measurements. This problem results in a limit to the accuracy that can be expected for measurements of distant links in long hop count paths.

However, this is only one of many more problems that single packet techniques encounter. Possibly the most significant of these problems is the effect of other traffic on the link (cross traffic). If a packet experiences delays, due to cross traffic on the link, then the estimation of time will be affected proportionally to the volume of this traffic.

Jacobson addresses this issue by assuming that cross traffic can only ever increase a delay seen by a packet. If enough packets are sent eventually one should get at least one through in the minimum time. These packets are said to have the shortest observed round trip time (SORTT) and is discussed by Downey. The graph shown earlier in Figure 1 could now be considered to be just plotting the SORTT values, and ignoring the delayed packets that would appear above each measurement.

More packets are required to discover the SORTT for more distant links. As the results from links further away from the source are combined to the effects of all of the previous links a packet must experience no queuing delay through every node, not just the node being measured.

While single packet techniques have many difficulties there are a number of common problems in the area of network measurement that do not strongly affect the bandwidth estimate results. Asymmetric routing is one such issue.

The term asymmetric routing refers to when the path too and from a node is different and, because of this, the delay in each direction can be different. This can cause problems for many types of active and passive measurement analysis. In the case of single packet bandwidth estimation techniques asymmetric routing causes few problems. The reason for this is while the ICMP error packet that we need for timing may come back via a different path the error packet is of fixed size for all sized measurement packets. Assuming the route doesn't change, the time it takes through the return path will, therefore, be fixed for all packet sizes. However, we cannot distinguish between added packet delay due to congestion on the forward and reverse paths. This means that the packet must experience zero delay through all routers on both the forward and reverse paths. This means that we need to send more packets to discover the SORTT.

Asymmetric routing will however cause problems with the latency estimation described earlier. As the return path may change speed for different hops, the subtraction of previous delay times will no longer hold true. This will result in an incorrect estimation of the latency added on by this link. An example of this is shown in Figure 2. In this measurement, packets returned from router three travels over a low latency link when compared to the links that

measurement two has to travel over. Using the example latencies shown in the figure, the measurement for link two will be the combined latency of the paths, summing to 44ms in this case. The measurement for link three however will be carried out over three 10ms paths, and back over two 2 ms paths and one 10ms path. This gives a sum of 44ms RTT. The estimated RTT latency for this link will then be 44ms - 40ms, 4ms. This compared to the correct value of the 10ms link (20ms RTT) latency.

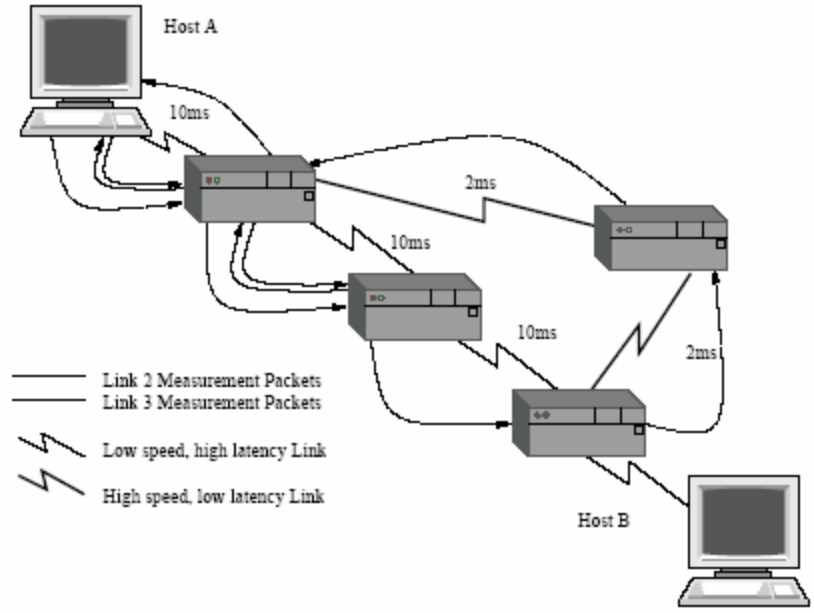

Figure 2: Latency errors caused by Asymmetric routing.

Bellovin and Jacobson use the one-packet delay model to develop a technique for measuring link bandwidths. Bellovin and Jacobson use the round-trip delay to successive routers along a path. The round-trip delay can be modeled as the sum of the one-way delay for the initial packet and that of its acknowledgement.

Bellovin and Jacobson resolve the problematic assumption about no queuing by observing that queuing caused by additional traffic can only increase delays. Therefore, the minimum of several observed delays of a particular packet size fits the model. Their technique is to send several packets for each of several different packet sizes, plot the delays of these packets versus their sizes, and then use linear regression (Figure 1) to obtain the slope of the graph. The inverse of the slope is the bandwidth.

In practice, the problems with this technique are that linear regression is expensive, routers are not built to send acknowledgments in a timely manner, some nodes are “invisible'', and the reverse path adds noise.

The first problem is that the linear regression described above must be done for every link measured. Many packets may be required to filter out the effect of other traffic and calculate a regression with high confidence. Jacobson provides pathchar as an implementation of the algorithms just described. Using its default settings, it will send 10MB of data in the course of measuring a 10 hop Ethernet path.

Downey uses statistical methods to reduce measurement traffic. Once he detects the convergence of a link bandwidth estimate, then he avoids sending further packets to measure this link. Methods such as this are complementary to our packet tailgating technique.

The second problem is that the one-packet technique requires getting timely acknowledgements from routers. Bellovin uses Internet Control Message Protocol (ICMP) Echo and Echo Reply packets sent to the routers, while Jacobson and Downey use UDP packets, successively incrementing the IP Time-To-Live (TTL) field to receive ICMP time exceeded responses from the

routers.

These approaches have the advantage that no special software needs to be deployed on routers to gather timing information, but unfortunately they may not work in all parts of the Internet. Because of malevolent use of ICMP packets, some routers and hosts either rate-limit them or filter them out, thus slowing down or precluding measurement.

Another problem is that bridges, host operating systems (OS s), and network interface cards (NIC s) are usually store-and-forward nodes but do not decrement the IP TTL and are not individually addressable in IP. Consequently, the links corresponding to these “invisible'' nodes cannot be detected or measured using the IP TTL decrement method cited above. There is a node between the source application and the source operating system because the sending OS usually must copy packets from the application's address space to the kernels. In addition, the source OS s network card driver usually must copy the sent packet from kernel address space across the system bus to the NIC. Finally, if the destination is a PC, the packet usually must be copied from the destination's NIC to the destination's kernel address space. The application-kernel, kernel-NIC, and NIC-kernel copies usually must be individually complete before the packet can be forwarded any further in the pipeline. These invisible nodes cause error in the measurement of the next link.

The final problem is that relying on acknowledgements and round-trip delays means that there is twice the possibility that queuing could corrupt a sample when compared to a technique that relies only on one-way delay. This is because queuing in the reverse path can delay the acknowledgement, even if there is no queuing in the forward path. As a result, many packets may be

required to filter out the effect of other traffic and calculate a regression with high confidence. To use one-way delay, one-packet-based techniques to use one-way delay would need new software at every router on a path, which would not be practical.

Another issue causing problems with the latency estimation is caused by the time taken by a router to generate the ICMP error message. ICMP can be used as a form of denial of service attack on a router. To reduce this risk many routers place a very low priority on the creation of ICMP packets to avoid overloading the routers CPU. This means that the latency observed will include this extra time that would not be seen by a packet traveling through a router. This will not affect the bandwidth estimation, however, as long as the router is consistent on the time introduced for all packet sizes. If the router handles packets of different sizes differently this would introduce errors into the slope calculation used to calculate bandwidth.

The combination of these problems means that this technique doesn't scale to higher hop counts. The amount of traffic required to accurately find the SORTT also reduces the usefulness of this method on busy paths.

The noise introduced into the measurement by cross traffic, and the other problems discussed here, is often larger than the difference between the transmission of the smallest packet (40 bytes) and the largest packet (normally 1500 bytes).

3.1.1.2. "Packet Pair"

Packet Pair models attempt to estimate the path capacity not the link capacity discovered by single packet techniques. These techniques have been in

use since at least 1993, when Bolot used them to estimate the path capacity between France and the USA. He was able to quite accurately measure the transatlantic capacity, which at that time was 128kbps.

Packet pair techniques are often referred to as packet dispersion techniques. This name is perhaps more descriptive. A packet experiences a serialization delay across each link due to the bandwidth of the link. Packet pair techniques send two identically sized packets back-to-back, and measure the difference in the time between the packets when they arrive at the destination.

For simplicity, we will initially ignore cross traffic, and discuss the effects it has later in this section. Figure 3 shows packet pair measurements diagrammatically.

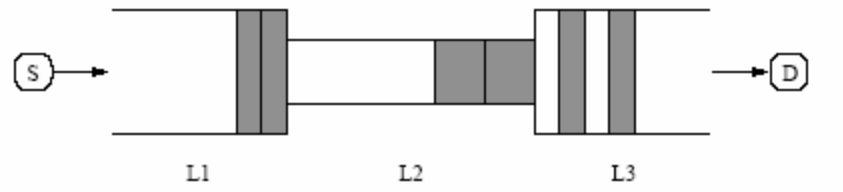

Figure 3: Packet pair measuring trough the limiting link.

Shown are 3 links of a path from source S to destination D, and two packets, shown once in each link. Link L1 and L3 have twice the capacity of L2; L2 is the capacity limiting link in the path. The first packet arrives at the router between L1 and L2 and is forwarded out on L2 without queuing delay as there is no other traffic present. L2 has a lower speed than L1 so the second packet arrives while the first packet is still being sent out and is queued. As soon as the first packet has been transmitted down L2 the second packet begins to be sent. As soon as the first packet is received by the router between L2 and L3 the router can forward it out on L3. The first packet will have finished being sent

before the second packet is fully received by the second router as L3 is faster than L2. As soon as the router does finish receiving the second packet it will forward it on. As the spacing can only be changed by a slower link, and we have defined L2 to be the capacity limiting (slowest) link, the spacing will remain the same through to the destination machine.

The spacing is equal to the time that the router at the end of the limiting link spent receiving the second packet after the first one was received. This because the first packet was sent as soon as it was fully received and then the router must wait until the second packet has fully arrived before it can be sent on. This value is actually equal to the transmission delay. The transmission delay (t) is proportional to the packet size (P) and the capacity of the link (b).

$$t = \frac{P}{b} \quad (3)$$

Note the difference between equation 1 and 3. Single packet techniques have to take into account the latency of the link in the calculations. Packet pair techniques do not need to estimate the latency of the link as it will be the same for both packets (in the absence of cross traffic), canceling it out.

Cross traffic effects are the most obvious and serious problem to affect packet pair measurements. If cross traffic delays the first packet it will compress the spacing between the packets and the bandwidth estimation will be high. This is referred to as time or probe compression. If cross traffic arrives in the queue between the first and second packet the spacing will be expanded resulting in underestimation of the bandwidth.

All recent research into packet pair techniques has focused on filtering out

the compressed or extended measurements to provide the closest estimation to the capacity of the path.

Lai and Carter both propose statistical methods to filtering the results to estimate the bandwidth. Both of these approaches assume that as the compression or extension values will be random, then the actual bandwidth should appear as the most common measurement.

However current research suggests that this may not be the case, and that there may be other, more common values than the actual bandwidth. This result is the focus of current research into packet pair techniques.

3.1.2 By the method applied

Considering this criterium, we have two big categories: Variable Packet Size (VPS) and Packet Train Dispersion (PTD). Steve Bellovin and Van Jacobson have proposed VPS for measuring metrics of each hop. In this technique, different size UDP and IP packets are generated and put in the medium in order to examine the round trip time, delay. From this we can extract the bandwidth and loss rate.

The VPS tools use variations of round trip time to estimate the characteristics of a path. An agent will send packets with specific dimmensions and it will use the round trip time to filter the effects of queuing on intermediar routers. By modulating the distance between packets, the VPS techniques can determine the distribution of round trip times.

PTD focuses on measuring the capacity on a path. The PTD methodology is based on the Packet Pair model in the context of congestion control. An

important side of PTD is the statistical aproximation. Paxsons has discovered that the distribution of bandwidth is multimodal and if we extract the global mode this value will provide us with a clue about the path capacity.

We can apply the PTD technique for measuring bandiwdth. We can assume that the dispersion of packet trains is invers proportional to the available bandwidth. By lengthning the packet train we can deduce that the values converge to Asymptotic Dispersion Rate (ADR), a value a little lower than the real capacity.

3.2. Existing applications

There are a plethora of host to host applications that measure network characteristics and their number grows everyday. It is very hard to cuantify and characterize each application, but some basic traits worth mentioning. We will browse fast trough the most representative applications and then we will present Table 1. which contains a complete list.

Allen Downey has implemented the PTD technique to create *clink*, and in a similar mode Bruce Mah has written *pchar (Network Characterization Service)*. Kevin Lai and Mary Baker have developed the tailgating technique and implemented in *nettimer.*

Clink and pathchar are updates of the original pathchar. Clink implements adaptiive testing techniques to vary the number of probes for each hop. We need only a few packets for links with low noise rate, but on busy links the number of probe packets increases exponentially,

Pchar focuses on poratbility for platforms and protocols such as Ipv6 or

linear regression methods.

Constantin Dovrolis has defined a method for measuring capacity implemented in many applications. It is based on the Paxsons observation about the multimodal distribution of the bandwidth. This method will be detaliate in the dedicated Chapter 5. In table 1 we mention and clasify the majority of existing applications.

Application	Author	Technique	Measuring	Short description
Abing		Single Packet	Point-to-Point	- pretty rudimentary; - the results vary a lot from an execution to another;
Bing	Pierre Beyssac	VPS	Point-to-Point	-it relies on the simpler ping module; - it determines the raw throughput of a link by using ICMP echo request for packets of different sizes;

Bprobe, Cprobe	Bob Carter	Packet Pair	Point-to-Point	- bprobe makes an estimation for the maximum path bandwidth; - cprobe makes an estimation of the congestion for a given path; - they rely on 2 characteristics of the IRIX OS: • a high precision timer which offers a great granularity for measuring the time between packets; • the ability to change the priority of a running process to avoid context switching;
Clink	Allen B. Downey	VPS/even-odd	Every link	- it is a reimplementation of pathchar; - it uses the even-odd technique; - when it reaches a routing instability, it gathers the data from al paths and makes an estimation; - many probe packets, it can be considered as a DoS attack;

Iperf	NLANR group	Path flooding	Point-to-Point	- it relies on ttcp and nettest; - it can use UDP and TCP packets; - it computes the bandwidth jitter, bandwidth and loss rate; - it shows MSS/MTU; - setting of the TCP window size by socket buffers; - multi-threaded application; - multicasting ability;
DummyNet	Luigi Rizzo			- flexbile tool for testing network protocols as well as for measuring bandwidth; - simulates the limitations of bandwidth, delay and multi-path effects;
Netperf	Rick Jones	Path flooding	Point-to-Point	- a benchmark for unidirectional throughput end-to-end latency;
Nettimer	Kevin Lai	Tailgating	Every link	- pasive (listen for other traffic in the network) or active; - uses a simple tailgating technique; - it can measure the bottleneck bandwidth and link bandwidth;

Pathchar	Van Jacobson	VPS/even-odd	Every link	- estimates performance for every link; - it uses the TTL field from IP packet and different sizes of UDP packets; - sums up incrementally values in the network;
Pipechar	Jin Guojon	Packet train	Every link	- measures maximum bandwidth and available bandwidth; - large number of packets required;
Pchar	Bruce A. Mah	VPS	Every link	- similar to patchar, on IPv6 protocol; - many SNMP facilities;
Pathchirp		Single Packet	Point-to-Point	- self induced congestion; - extracts mant possible bandwidth values; - it needs at least two congestioned queues from source to destination;

Sprobe	Saroiu, Savage	Packet Pair	Point-to-Point	- bidirectional bottleneck bandwidth estimation in a an uncooperative medium; - it requires three 3 RTT to make a bandwidth estimation; - 6 TCP SYN packets are send to ar inactive port and 6 pachete TCP RST are received; - Application level protocols;
Treno	Matt Mathis, Jamshid Mahdavi	TCP simulation		- throughput estimation from TCP packets; - independent estimation for every host;
Ttcp, Nttcp	Elmer Bartel	Path flooding		- benchmark for throughput and loading generator; - initially implemented for file transfer; - many versions, the most recent include suport for UDP, generating data patterns, page alignment or offset control;
Viznet	NLANR group			- a standalone java application;

Table 1: Tools for measuring the parameters of networks

There are are a lot of large-scale projects that target extracting network characteristics using supervising tools. The most important are:

- *AMP* (Active Measurement Program): the NLANR group (National Laboratory for Applied Network Research) develops this project for monitoring and diagnosticating high bandwidth and performance networks.

- *NIMI*: a project financed by NSF to gather data from all over the Internet. The NMI probes cand be used for configuration and coordination.

- *PingER*: a DOE/MICS project for monitoring end-to-end links and to gather data for Esnet, HENP (High Energy&Nuclear Phsycs).

- *RIPE*: a project for monitoring conectivity parameters such as delays or routing vectors.

- *Skitter*: used for forwarding IP paths from source till the destination. It is developed and maintaned by CAIDA (Cooperative Association for Internet Data Analysis).

- *Skping*: a hig precision tool for debugging. Uniform engine uniform for many operating systems.

- *Surveyor*: uses active delay testing and loss rate along a path between CSG servers (educational comunities in the U.S.).

Chapter 4 – MonALISA. Monitoring Agents using Large Integrated Services Arhitecture

4.1. General architecture

The MonALISA (Monitoring Agents in A Large Integrated Services Architecture) system provides a distributed ser-vice for monitoring, control and global optimization of complex systems. MonALISA is based on a scalable Dynamic Distributed Services Architecture (DDSA) implemented using Java / JINI and Web Services technologies. The scalability of the system derives from the use of a multithreaded execution engine to host a variety of loosely-coupled self-describing dynamic services or agents, and the ability of each service to register itself and then to be discovered and used by other services, or clients that require such information.

A service in the DDSA framework is a component that interacts autonomously with other services either through dynamic proxies or via agents that use self-describing protocols. By using dedicated lookup services, a distributed services registry, and the discovery and notification mechanisms, the services are able to access each other seamlessly. The use of dynamic remote event subscription allows a service to register an interest in a selected set of event types, even in the absence of a notification provider at registration time. The lookup discovery service will then automatically notify all the subscribed services, when a new service, or a new service attribute, becomes available.

The code mobility paradigm (mobile agents or dynamic proxies) used in the DDSA extends the approaches of re-mote procedure call and client- server. Both the code and the appropriate parameters are downloaded dynamically into the system. Several advantages of this paradigm are: optimized asynchronous communication and disconnected operation, remote interaction and adaptability, dynamic parallel execution and autonomous mobility. The combination of the service architecture and code mobility makes it possible to build an extensible hierarchy of services that is capable of managing very large systems.

4.2. Monitoring Service

An essential part of managing a global system, like the Grids, is a monitoring system that is able to monitor and track the many site facilities, networks, and the many task in progress, in real time. The monitoring information gathered also is essential for developing the required higher level services, and components of the Grid system that provide decision support, and eventually some degree of automated decisions, to help maintain and optimize workflow through the Grid. MonALISA is an ensemble of autonomous multi-threaded, agent-based subsystems which are registered as dynamic services and are able to collaborate and cooperate in performing a wide range of monitoring tasks in large scale distributed applications, and to be discovered and used by other services or clients that require such information. MonALISA is designed to easily integrate existing monitoring tools and procedures and to provide this information in a dynamic, self describing way to any other services or clients. MonALISA services are organized in groups and this attribute is used for registration and

discovery.

4.3. Data Collection Engine

The system monitors and tracks site computing farms and network links, routers and switches using SNMP, and it dynamically loads modules that make it capable of interfacing existing monitoring applications and tools (e.g. Ganglia, MRTG, LSF, PBS, Hawkeye ?.). The core of the monitoring service is based on a multithreaded system used to perform the many data collection tasks in parallel, independently. The modules used for collecting different sets of information, or interfacing with other monitoring tools, are dynamically loaded and executed in independent threads. In order to reduce the load on systems running MonALISA, a dynamic pool of threads is created once, and the threads are then reused when a task assigned to a thread is completed. This allows one to run concurrently and independently a large number of monitoring modules, and to dynamically adapt to the load and the response time of the components in the system. If a monitoring task fails or hangs due to I/O errors, the other tasks are not delayed or disrupted, since they are executing in other, independent threads. A dedicated control thread is used to stop properly the threads in case of I/O errors, and to reschedule those tasks that have not been successfully completed. A priority queue is used for the tasks that need to be performed periodically. A schematic view of this mechanism of collecting data is shown in Figure 1.

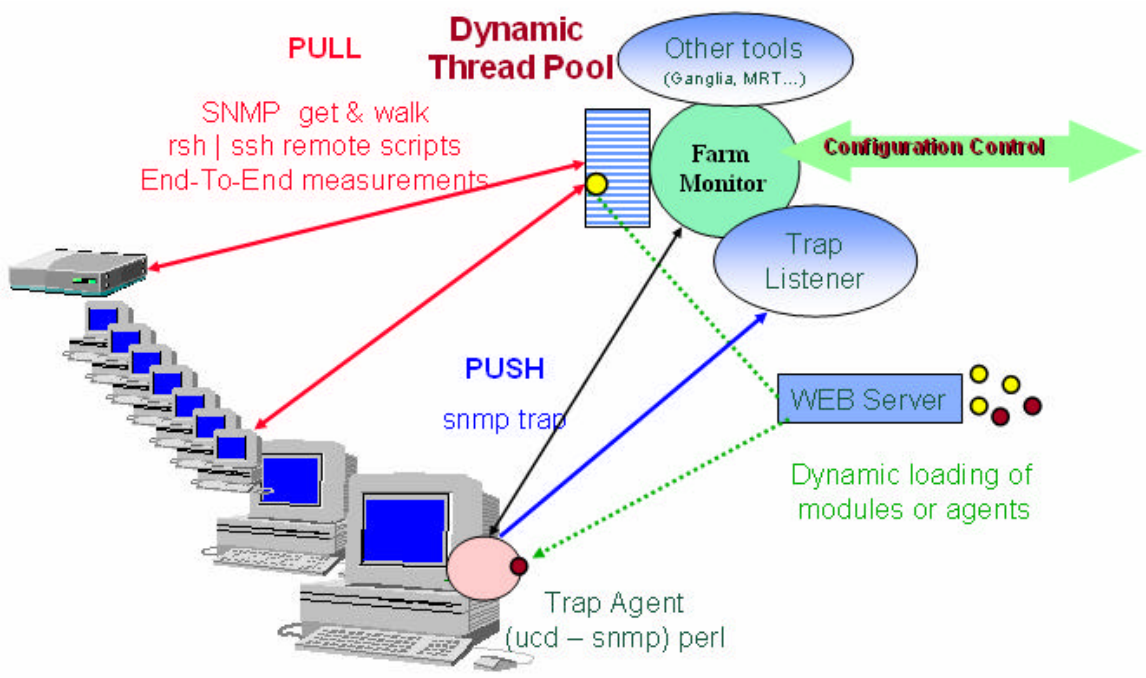

Figure 4: The Data Collection engine.

This approach makes it relatively easy to monitor a large number of heterogeneous nodes with different response times, and at the same time to handle monitored units which are down or not responding, without affecting the other measurements.

A Monitoring Module is a dynamic loadable unit which executes a procedure (or runs a scrip / program or performs SNMP request) to collect a set of parameters (monitored values) by properly parsing the output of the procedure. In general a monitoring module is a simple class, which is using a certain procedure to obtain a set of parameters and report them in a simple, standard format.

Monitoring Modules can be used for pulling data and in this case it is

necessary to execute them with a predefined frequency or to "install" (has to run only once) pushing scripts (programs) which are sending the monitoring results (via SNMP, UDP or TCP/IP) periodically back to the Monitoring Service. Allowing to dynamically load these modules from a (few) centralized sites when they are needed makes much easier to keep large monitoring systems updated and to provide new functionalities dynamically. Users can implement easily any new dedicated modules and use it the MonALISA framework.

4.4. Register and discover services

Each MonALISA service registers with a set of JINI Lookup Discovery Services (LUS) as part of a group, and having a set of attributes. The LUSs are also JINI services and each one may be registered with the other LUSs (Figure 2) If two LUSs have common groups any information related with a change of state detected for a service in the common group by one is replicated to the other one. In this way it is possible to build a distributed and reliable network for registration of services and this technology allows dynamically adding or removing LUSs from the system. Any service should also provide for registration the code base for the proxies that other services or clients need to instantiate for using it. This approach is used to make sure that the right proxies are used for each service while different versions may be used in a distributed organization at the same time. The registration is based on a lease mechanism that is responsible to verify periodically that each service is alive. In case a service fails to renew its lease, it is removed from the LUSs and a notification is sent to all the services or clients that subscribed for such events. Any monitor client services is

using the Lookup Discovery Services to find all the active MonALISA services running as part of one or several group ?communities?. It is possible to select the services based on a set of matching attributes. The discovery mechanism is used for notification when new services are started or when services are no longer available. The communication between interested services or clients is based on a remote event notification mechanism which also supports subscription. The client application connects directly with each service it is interested in for receiving monitoring information. To perform this operation, it first downloads the proxies for the service it is interested in from a list of possible URLs specified as an attribute of each service, and than it instantiate the necessary classes to communicate with the service. This procedure allows each service to correctly interact with other services or clients.

4.5. Predicates, filters and alarm agents

The clients can get any real-time or historical data by using a predicate mechanism for requesting or subscribing to selected measured values. These predicates are based on regular expressions to match the attribute description of the measured values a client is interested in. They may also be used to impose additional conditions or constrains for selecting the values. In case of requests for historical data, the predicates are used to generate SQL queries into the local database. The subscription requests will create a dedicated thread, to serve each client. This thread will perform the matching test for all the predicates submitted by a client with the measured values in the data flow. The same thread is responsible to send the selected results back to the client as compressed

serialized objects. Having an independent thread per client allows sending the information they need, fast, in a reliable way and it is not affected by communication errors which may occur with other clients. In case of communication problems these threads will try to reestablish the connection or to clean-up the subscriptions for a client or a service which is not anymore active.

Monitoring data requests with the predicate mechanism is also possible using the WSDL/SOAP binding from clients or services written in other languages. The class description for predicates and the methods to be used are described in WSDL and any client can create dynamically and instantiate the objects it needs for communication.

4.6. Client proxy

The architecture provides a proxy service which is used by clients to connect to different services. The proxy service is also a JINI service. In our service design we use the mutual discovery between services and proxies to detect when a certain service runs run behind a firewall or NAT. In this case the service initiates a connection to all the available proxies for a community and registers itself with the LUSs. Any client can interact now with such services via the proxy services. At the same time the proxy service does an "intelligent" multiplexing of subscribed data for multiple clients. We run multiple proxy services for redundancy and also for a dynamic load balancing of clients.

4.7. Pseudo-clients and repositories

A generic framework for building "pseudo-clients" for the MonALISA services was developed. This has been used for creating dedicated Web service repositories with selected information from specific groups of monitoring services. The pseudo-clients use the same LUSs approach to find all the active MonALISA services from a specified set of groups and subscribes to these services with a list of predicates and filters.

These predicates or filters specify the information the pseudo client wants to collect from all the services. A pseudo client stores all the values received from the running services in a local MySQL database, and is using procedures written as Java threads to compress old data. A Tomcat based servlet engine is used to provide a flexible way to present global data and to construct on the fly graphical charts for current or customized historical values, on demand. Dedicated servlets are used to generate Wireless Access Protocol (WAP) pages containing the same information for mobile phone users. Multiple Web Repositories can easily be created to globally describe the services running in a distributed environment.

4.8 The ABPing module

An important part of the MonALISA architecture are the VRVS reflectors, which enable setting video conferences. For every reflector we have modules to collect information, to monitor the traffic, to detect clients and active rooms, or to detect communication break-downs. All these data are available real-time. The modules are implemented in the Monitoring Module described above.

Every reflector must choose from a list of possible peers and possible

paths for the packets. The ABPing (Available Bandwidth Ping) module is used for these tasks. In ABPing we use UDP packets and a "Single Packet" system with acknowldgements. The quality of every link is computed using the formula:

RTimeQuality = OVERALL_COEF + RTT_COEF*rtt +

PKT_LOSS_COEF*loss% + JITTER_COEF*jitter (4)

This is a flexible formula, based on RTT(Round Trip Time) we can draw the packet loss percent, the jitter. The obtained values represent:

RTT – the time from the reflector to the peer and back;

LOSS – the percent of packets lost until the packet reaches the peer;

JITTER – the sum of probe train variations divided by an average RTT mediu and by the number of probes;

The list of peer coefficients is easily configurable. Every reflector extracts from these configuration files the needed information. The configuration files are checked periodically from a configurable URL and the peer list is kept in the ABPing module.

Chapter 5 – *ADR.* Measuring end-to-end capacity

5.1. Preliminary considerations

Let us define two important bandwidth metrics for a netwrok path. Consider a path P as a sequence of *First-Come First Served* (FCFS) store and forward links that transfer packets from the sender S to the receiver R. Assume thet the path is fixed and unique for the duration od the measurements, i.e. no routing changes or multipath forwarding occur. Each link i transmits data with a constant rate of C_i bits per second, referred to as *link capacity* or *transmission rate*. Two bandwidth metrics that are comonly associated with path P are the capacity C ad the available bandwidth A. The capacity is the minimum transmission rate among all links in P. Note that the capacity does not depend on the traffic load of the path. Available bandwidth, on the other hand, is the minimum spare link capacity, i.e., capacity not used by other traffic among all links in P.

More formally, if H is the number of hops (links) in P, C_i is the capacity of link i, and C_0 is the transmission rate of the sender, the the path capacity is:

$$C = \min_{i=0\ldots H} C_i \quad (5)$$

Additionally, if u_i is the utilization of link i $0 < u_i < 1$ over a certain time interval, the average spare capacity of link I is $C_i (1 - u_i)$. Thus the available bandwidth of P in the same interval can be defined as:

$$A = \min_{i=0...H} [C_i(1 - u_i)] \quad (6)$$

The link with the minimum transmission rate determines the capacity, while the link with the minimum spare bandwidth determines the available bandwidth. To avoid the term bottleneck link, that has been widely used for both metrics, we refer to the capacity limiting as *narrow link*, and to the available bandwidth limiting as *tight link*. These two may be different.

The packet pair technique presented before will be used for measuring the capacity of a path. When a packet is transmitted by a store-and-forward link it encounters a transmission (serialization) delay, related to the clock rate of the underlying transmission hardware. In a link of capacity C_i, the transmission delay for a packet of size L is $t = L/C_i$. For now, let us ignore the fact that a packet can carry different encapsulation headers in different links, and assume that the packet size L remains constant as the packet traverses the path. A packet pair measurement consists of two packets of the same size L sent back-to-back from S to R. Without any cross traffic in the path, the packet pair will reach R with a dispersion d equal to $t_n = L/C$. Note that t_n is the transmission delay at the narrow link. The receiver can then estimate the capacity C from the measured dispersion d as $C = L/d$.

5.2. “Packet Pair Dispersion”

5.2.1. Short historical notes

The concept of packet dispersion, as a burst of packets traverses the narrow link of a path, was originally described by Jacobson and it is closely

related to the TCP self-clocking mechanism. Jacobson did not consider however the effects of cross traffic, and so he did not distinguish between capacity and available bandwidth. Keshav explored the same concept in the context of congestion control, recognized that the dispersion of packet pairs is not related to the available bandwidth when router queues use the FCFS discipline, and so he focused on fair-queuing instead. Bolot used packet dispersion measurements to estimate the capacity of a transatlantic link and to characterize the traffic inter-arrivals. Early work on packet pair dispersion was followed by more sophisticated variations, focusing on statistical techniques that can extract an accurate capacity estimate from noisy bandwidth measurements.

Paxson was the first to observe that the bandwidth distribution is multi-modal and he elaborated on the identification and final selection of a capacity estimate from these modes. He used both the packet pair and packet trains to estimate the underlying bandwidth distribution. The complete methodology is called PBM ("Packet Bunch Modes").

More recently the packet pair technique has been largely revised, explaining the multiple modes that Paxson observed based on queuing and cross traffic. Additionally, Pasztor and Veitch revealed the negative effect of lower layer headers and it argued for peak detection as superior to mode detection for capacity estimation. Along a different research thread Harfoush, Bestavros and Byers showed that it is possible to measure the capacity of targeted path segments using packet dispersion techniques.

More recently, significant progress has been made in the estimation of available bandwidth trough end-to-end measurements. The TOPP and SLoPS techniques use packet streams of variable rates. When the stream rate exceeds

the available bandwidth, the stream arrives at the receiver with a lower rate than its rate at the sender. TOPP has the additional advantage that, together with available bandwidth, it can also estimate the capacity of the tight link in the path, and in some cases the capacity and available bandwidth of other links in the path. SLoPS, on the other hand, give more emphasis on the variability of available bandwidth, and on the resulting measurement uncertainty ("grey region").

5.2.2. The basic traits of the method

Consider an H-hop path defined by the sequence of capacities $P = \{C_0, C_1, \ldots, C_H\}$. Two packets of size L are sent back-to-back from the source to the sink; these packets are the *packet pair* or *probing packets*. The *dispersion* of the packet pair is the interval from the instant the last bit of the first packet is received at a certain path point to the instant the last bit of the second packet is received at that point. The dispersion is $?_0 = t_o = L/C_0$ after the source, and let it be $?_i$ after link i. When the packet pair reaches the sink, the dispersion is $?_H$ and the receiver computes a bandwidth estimate $b = L/?_H$. Since $?_H$ varies in general, if we repeat the experiment many times the b values will form a certain distribution ß. Our goal, then, is to infer a final path capacity estimate C from the distribution ß.

First, suppose that there is *no cross traffic in the path*. It is easy to see that the dispersion $?_i$ cannot be lower than the dispersion at the previous hop $?_{i-1}$ and the transmission delay $t_i = L/C_i$ at hop i, i.e., $?_i = \max\{?_{i-1}, t_i\}$. Applying this

Model recursively from the sink back to the source, we find that the dispersion at the receiver is:

$$\Delta_H = \max_{i=0\ldots H} \tau_i = \frac{L}{\min_{i=0\ldots H}\{C_i\}} = \frac{L}{C_n} = \tau_n \quad (7)$$

, where C_n and t_n are the capacity and the transmission delay of the narrow link, respectively. Consequently, when there is no cross traffic, all the bandwidth estimates are equal to the capacity ($b = C_n = C$).

When there is cross traffic in the path, the probing packets can experience additional queuing delays due to cross traffic. Let d_i^1 be the queuing delay of the first probing packet at hop i, and d_i^2 be the queuing delay of the second probing packet at hop i *after the first packet has been transmitted at that link*. The dispersion after hop i is:

$$\Delta_i = \begin{cases} \tau_i + d_i^2 & \text{if } \tau_i + d_i^1 \geq \Delta_{i-1} \\ \Delta_{i-1} + (d_i^2 - d_i^1) & \text{otherwise} \end{cases} \quad (8)$$

Note that when $t_i + d_i^1 < ?_{i-1}$ and $d_i^2 < d_i^1$ the dispersion decreases from hop i - 1 to hop I ($?_i < ?_{i-1}$). This effect can cause a dispersion at the receiver that is lower than the dispersion at the narrow link, i.e., $?_H < t_n < L/C$ if there are additional hops after the narrow link; we refer to such links as *post-narrow links*3. This observation means that *the capacity of the path cannot be estimated simply from the minimum measured dispersion*, as that value could have resulted from a Post - narrow link.

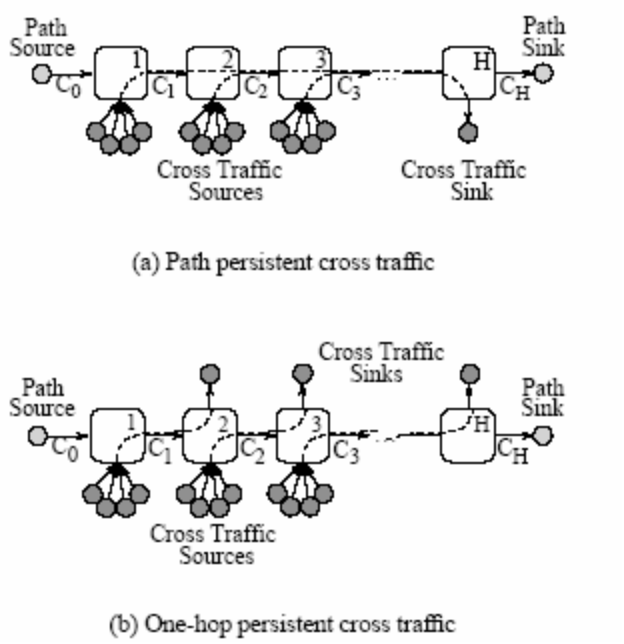

Figure 5: The Data two extremes of cross traffic routing.

An important issue is the routing of the CT packets relative to the packet pairs. The two extreme cases are shown in Figure 5; in Figure 5 - the CT packets follow the same path as the packet pairs (*path persistent CT*), while in Figure 5-b the CT packets always exit one hop after they enter the path (*one-hop persistent CT*). We simulate the one-hop persistent CT case. In the following experiments, the bandwidth distribution ß is formed from 1000 packet pair experiments.

Figure 6 shows the histogram of ß, with a bin width of 2 Mbps, for a path P = {100, 75, 55, 40, 60, 80} (all capacities in Mbps). Note that the path capacity is C = 40Mbps, while the post - narrow links have capacities of 60 and 80 Mbps, respectively.

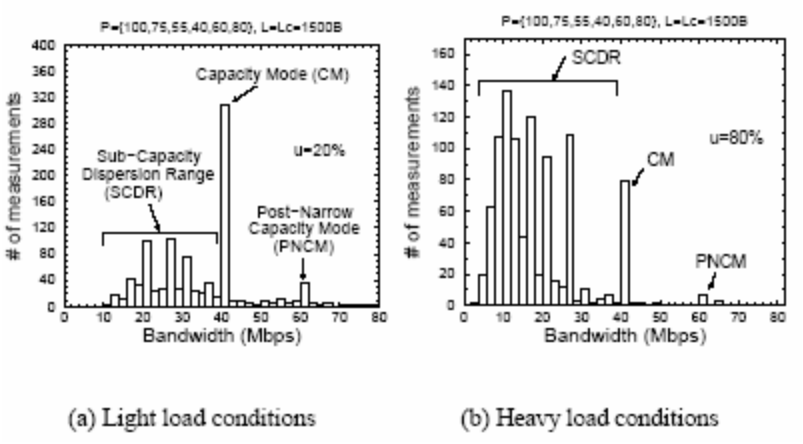

Figure 6: The ß distribution in two different path loads.

In Figure 6-a, each link is 20% utilized, whereas in Figure 6-b, all links are 80% utilized. When the path is lightly loaded (u=20%) the capacity value of 40 Mbps is prevalent in ß, forming the *Capacity Mode (CM)*, which in this case is the global mode of the distribution. Bandwidth estimates that are lower than the CM are caused by CT packets that interfere with the packet pair, and they define the *Sub-Capacity Dispersion Range (SCDR)*. For instance, the SCDR in Figure 6-a is between 10 and 40 Mbps; the cause of the local modes in the SCDR is discussed in the next paragraph. Bandwidth estimates that are higher than the CM are caused in the post-narrow links when the first probing packet is delayed more than the second; these estimates are referred to as *Post-Narrow Capacity Modes (PNCMs)*. Note a PNCM at 60 Mbps, which is the capacity of the link just after the narrow link; this local mode is created when the first probing packet is delayed long enough for the packet pair to be serviced back-to-back in that link.

In heavy load conditions (u=80%), the probability of CT packets interfering with the probing packets is large, and the CM is not the global mode of B. Instead, the global mode is in the SCDR, which now dominates the bandwidth

measurements. A key point here is that *the path capacity cannot be always correctly estimated by statistical techniques that extract the most common bandwidth value or range*. Instead, we must examine the resulting bandwidth distribution in queuing terms; analyze what causes each of the local modes, and what differentiates the CM from the rest of the local modes.

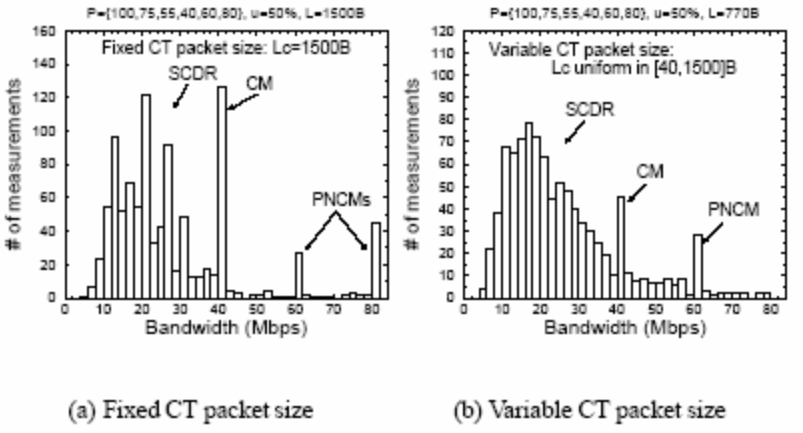

Figure 7: Fixed versus variable CT packet size L_C.

Figure 7 shows for the same path when the CT packet size L_C is fixed (1500 bytes) and when it varies uniformly in the range [40, 1500] bytes (u=50%). In the first case, the probing packet size L is also 1500 bytes, while in the second case it is 770 bytes, i.e., the average of the [40, 1500] range5. When all packets have the same size ($L_C = L = 1500B$), it is simpler to explain the local modes in the SCDR. For instance, consider the path P = {100, 60, 40}, and assume that all packets have the same size. A local mode at 30 Mbps can be caused by a packet interfering with the packet pair at the 60 Mbps link, since in that case the dispersion after the narrow link is $\Delta_n = \frac{L}{40} + (2 * \frac{L}{60} - \frac{L}{40}) = \frac{L}{30}$. Similarly, a mode at 20 Mbps is caused by a packet interfering with the packet pair at the 40 Mbps

link or by two packets interfering at the 60 Mbps link, and so on.

When the CT packet size varies uniformly in the range [40,1500]B though (Figure 7-b), the resulting dispersion is less predictable, since a single packet interfering with the packet pair can produce a range of dispersion values, depending on its size. However, *the CM and one or more of the PNCMs are still distinct in the distribution*, as they are caused by the probing packets being serviced back-to-back from the narrow or from post – narrow links, respectively. Several measurement studies have shown that the packet size distribution in the Internet is centered on three or four values. Specifically, about 50% of the packets are 40 bytes, 20% are 552 or 576 bytes, and 15% are 1500 bytes. These Common packet sizes would cause a packet pair bandwidth distribution that is more similar to the 'discrete dispersion' effects of Figure 7-a, rather than the 'continuous dispersion' effects of Figure 7-b.

5.3. "Packet Train Dispersion"

Extending the packet pair technique, the source can send N > 2 back-to-back packets of size L to the sink; we refer to these packets as a *packet train of length* N. The sink measures the total dispersion ?(N) of the packet train, from the first to the last packet, and computes a bandwidth estimate as $b(N) = \frac{(N-1)L}{\Delta(N)}$. Many such experiments form the bandwidth distribution ß (N).

If there is no cross traffic in the path, the bandwidth estimates will be equal to the capacity C, as in the packet pair case. Measuring the capacity of a path using packet trains is required when the narrow link is multi-channeled. In a k-channel link of total capacity C, the individual channels forward packets in

parallel at a rate of C/k and the link capacity can be measured from the dispersion of packet trains with N = k+1. Packet trains are also required to measure the *sustainable rate* of a traffic shaper.

It may appear at first that using packet trains, instead of packet pairs, makes the capacity estimation more robust to random noise caused by cross traffic. One can argue that this is true because packet trains lead to larger dispersion values, which are more robust to measurement noise. However, this is not the case due to the following reason. Although the dispersion ? (N) becomes larger as N increases, so does the 'noise' in the measured values of ?(N), since it becomes more likely that CT packets will interfere in the packet train. Packet trains should be less prone to noise, since individual packet variations are smoothed over a single large interval rather than N - 1 small intervals, *but* with a larger N the greater the likelihood that a packet trains will be dispersed by cross traffic, leading to bandwidth underestimation.

In this section, we present simulation and experimental results illustrating the effect of N in the bandwidth distribution ß (N) and make some general observations about this relation. Figure 8 shows the histograms of ß (N), for four increasing values of N, from simulations of the path P = {100,75,55,40,60,80} with u=80% in all links. Figure 9 shows the histograms of ß (N), for four increasing values of N.

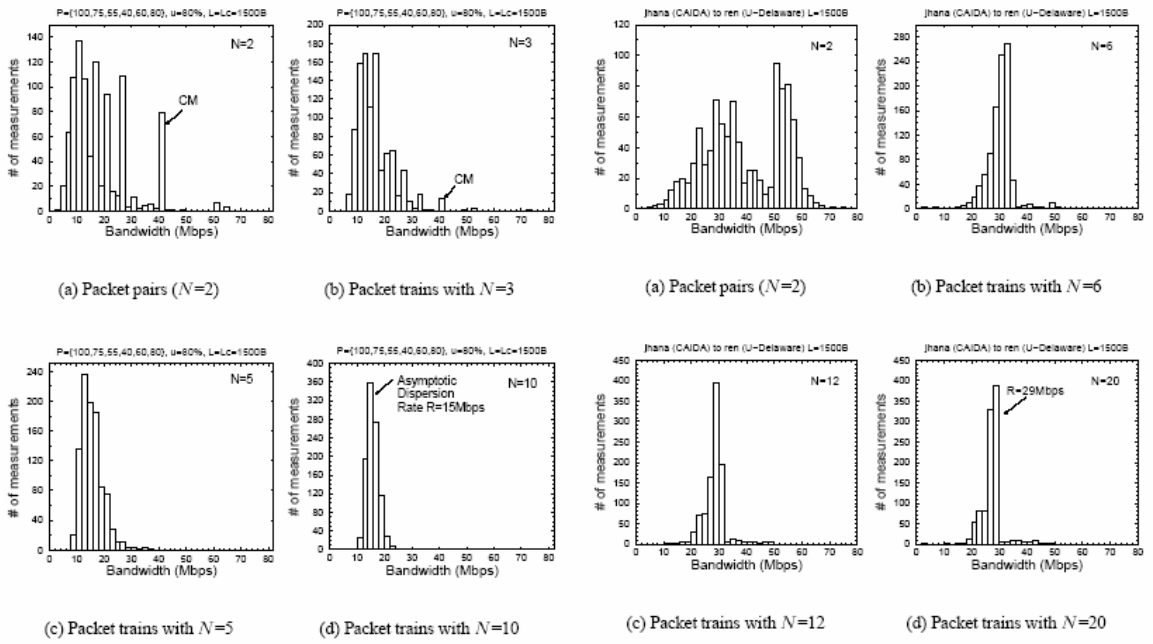

Figure 8, 9: The effect of packet train length (simulation and measurements).

A first observation is that, *as* N *increases, the CM and PCNMs become weaker, until they disappear, and the SCDR prevails in the bandwidth distribution* ß (N). The reason is that, as N increases, almost all packet trains encounter additional dispersion due to CT packets. This also means that *the best value of* N *for generating a strong capacity mode is* N*=2, i.e., to use packet pairs; anything longer than packet pairs is more likely to get additional dispersion due to cross traffic*.

A second observation is that, *as* N *increases,* ß (N) *becomes uni-modal*. This implies that, when N is large, the dispersion of packet trains by CT packets is not determined by distinct interference cases, forming local modes, but it is determined by the aggregate amount of CT interfering with the packet train.

A third observation is that *the range of the distribution, which is related to the measurement variance, decreases as* N *increases*. This means that the

variance in the amount of cross traffic interfering with the packet train decreases, as the length of the packet train increases.

A fourth observation is that, *when* N *is sufficiently large and* ß (N) *is uni-modal, the center of the (unique) mode is independent on* N. We refer to the center of this unique mode as the *Asymptotic Dispersion Rate (ADR) R*. The fact that ADR does not depend on the packet train length means that, for sufficiently large N, the dispersion of the packet train ? (N) becomes proportional to N - 1, and thus the packet train length cancels out from the bandwidth estimate

$$b(N) = \frac{(N-1)L}{\Delta(N)}.$$

5.4 Average Dispersion Rate (ADR)

In this section, we present a model for the dispersion of packet trains, taking into account the cross traffic in the path. First, consider a single hop path $P = \{C_0, C_1\}$ with $C_0 < C_1$, i.e., the C_1 link ('link-1') is the narrow link. A packet train of length N is sent from the source to the sink with initial dispersion $?_0 = L$ (N – 1) C0. Let r_1 be the average incoming rate of cross traffic in link-1. The average amount of cross traffic that arrives in link-1 during $X_1 = ?_0 r_1$. Assuming that the link-1 queue is serviced in a FCFS basis, the X_1 cross traffic interferes with the packet train packets, and so the average dispersion at the exit of the narrow link is:

$$\bar{\Delta}_1 = \frac{(N-1)L + \bar{X}_1}{C_1} = \frac{(N-1)L}{C_1}(1 + u_1 \frac{C_1}{C_0}) \quad (9)$$

, where $u_1 = r_1/C_1$ is the load (utilization) of the narrow link due to cross traffic.

Consequently, the average bandwidth estimate at the receiver that we refer to as the Asymptotic Dispersion Rate R, is:

$$R \equiv \frac{(N-1)L}{\bar{\Delta}_1} = \frac{C_1}{1 + u_1 \frac{C_1}{C_0}} < C_1 \quad (10)$$

, which is lower than the path capacity. Note that *the ADR is independent of* N, since the amount of interfering cross traffic X_1, and thus the overall dispersion $?_1$, is proportional to N - 1.

Some comments on Equation 10 follow. First, if the capacities C_0 and C_1 are known, we can measure R from the dispersion of long packet trains, compute the cross traffic utilization u_1 from Equation 10, and then compute the available bandwidth as $A = C_1(1 – u_1)$. So, the available bandwidth of single hop paths can be estimated, using the dispersion of packet trains that are sufficiently long to produce a narrow estimate of R. This also implies that the available bandwidth is not inversely proportional to the dispersion of long packet trains, even for single hop paths.

These results can be generalized to an H-hop path with $C_0 > C_1 > \ldots > C_H$, for the case of path persistent cross traffic. Let r_i be the average rate of cross-traffic that enters the path in link i. The average dispersion at the exit of link i, then, is:

$\bar{\Delta}_i = \bar{\Delta}_{i-1}(C_{i-1} + r_i)/C_i$, and the ADR becomes

$$R = \frac{(N-1)L}{\bar{\Delta}_H} = \frac{C_H}{\prod_{i=1}^{H}(1 + \frac{r_i}{C_{i-1}})} \quad (11)$$

For instance, for the path $P = \{C_0, C_1, C_2\}$ with $C_0 > C_1 > C_2$:

$$R = \frac{C_2}{1 + \frac{r_1}{C_0} + \frac{r_2}{C_1} + \frac{r_1 r_2}{C_0 C_1}} \quad (12)$$

When the capacities do not decrease along the path, the analysis is more complicated. In the single-hop case $P = \{C_0, C_1\}$ with $C_0 < C_1$, there would be an idle spacing of duration $L/C_0 - L/C_1$ at the exit of link-1 between any two probing packets, if there was no cross traffic. The cross traffic can fill in the idle space in the packet train, or cause additional dispersion without filling in all the idle space. A lower bound on the dispersion $?_1$ can be derived if we assume that the cross traffic increases the packet train dispersion beyond $?_0$ only after it fills in all the idle spacing. When this is the case, the dispersion at the receiver is

$$\bar{\Delta}_1 = \Delta_0 + \max\left\{\frac{\bar{X}_1}{C_1} - (N-1)L\left(\frac{1}{C_0} - \frac{1}{C_1}\right), 0\right\} \quad (12)$$

If the cross traffic load is sufficiently low ($r_1 < C_1 - C_0$), the dispersion is not increased at link-1 (i.e., $?_1 = ?_0$), and so $R = C_0$. Otherwise, the final dispersion becomes $?_1 = ((N - 1)\, L/C_0)\, (u_1 + C_0/C_1)$, which gives the same ADR value as Equation 10. These results can be extended for the case of H hops, when the cross traffic is path persistent. Specifically, a lower bound on the dispersion $?_H$ can be derived if we assume that the cross traffic increases the packet train dispersion only after it fills in all the idle spacing between probing packets. Then,

$$\bar{\Delta}_i = \begin{cases} \bar{\Delta}_{i-1}\frac{C_{i-1}+r_i}{C_i} & \text{if } C_{i-1} \geq C_i \\ & \text{or } r_i \geq C_i - C_{i-1} > 0 \\ \bar{\Delta}_{i-1} & r_i < C_i - C_{i-1} \end{cases} \quad (13)$$

Given the capacities and cross traffic rates in each hop, and since $?_0 = L\,(N - 1)/C_0$, we can solve recursively for $?_H$, and thus for R. When the cross traffic

is not path persistent, i.e., CT packets exit the path before the last hop, the dispersion of packet trains is hard to analyze for the same reason: CT packets an interfere in the packet train increasing its dispersion, and then exit the path leaving idle space, or 'bubbles', between probing packets. These bubbles can be filled in by CT packets in subsequent hops, or they can persist until the packet train reaches the sink. For the case of one-hop persistent cross traffic, an upper and a lower bound can be derived for R. Note that since the cross traffic is assumed to be one-hop persistent in this case, the utilization of link i is $u_i = r_i/C_i$. For an H-hop path in which all links have the same capacity C, it can be shown that the ADR is:

$$\frac{C}{\prod_{i=1}^{H}(1+u_i)} \leq R \leq \frac{C}{1+\max_{i=1\ldots H} u_i} \quad (14)$$

The lower bound corresponds to the case that bubbles are never filled in, while the upper bound corresponds to the case that the bubbles created at the link with the maximum utilization are the only ones that reach the receiver, and that the rest of the path links just fill in (partially) those bubbles.

5.5. A capacity estimation methodology

In this section, we present a capacity estimation methodology based on the insight developed so far in the paper. This methodology has been implemented in a tool called *ADR*. The ADR methodology requires the cooperation of both the source and the sink, i.e., it is a *two end-point methodology*. I prefer the two end-point methodology, even though it is less flexible, because it is more accurate.

5.5.1. Phase I: Packet pair probing.

One is more likely to observe the capacity mode using packet pairs than using packet trains. Consequently, in this phase we use a large number of packet pair experiments to 'uncover' all the local modes of the bandwidth distribution ß, expecting that one of them are the CM. There is a trade-off in the selection of the probing packet size L: smaller packets lead to stronger PNCMs, while larger packets lead to a more prevalent SCDR. A probing packet size of L=800 bytes usually leads to the strongest CM in the resulting bandwidth distribution. In ADR, Phase I consists of K_1=2000 packet pair experiments using a packet size of L=800 bytes.

From the resulting distribution of bandwidth measurements ß, we obtain all the local modes. The numerical procedure for the identification of the local modes is similar to the algorithm described before, but the user has to specify the *histogram bin width ?*, which is also the *resolution of the final capacity estimate*. If, for example, the resolution is ? = 2 Mbps, *ADR* will produce a final estimate that is a 2 Mbps interval. The resolution is a critical parameter for the accuracy of the final result.

The sequence of local modes, *in increasing order*, is denoted as $M = \{m_1, m_2, \ldots, m_M\}$. We expect that one of these local modes, say m_k, is the CM (i.e., $C = m_k$), with the larger modes being PCNMs, and the smaller modes being in the SCDR of ß. If the distribution ß is uni-modal, which happens in very lightly loaded paths, the measurement process terminates and the capacity estimate C is the unique mode m_1. Otherwise, Phase-II selects m_k from M.

5.5.2. Phase II: Packet train probing.

As N increases, the CM and the PNCMs are eliminated from the bandwidth distribution ß (N), and the SCDR accumulates all measurements. Gradually, ß (N) becomes uni-modal, centered at The Asymptotic Dispersion Rate R and the width of this unique mode is reduced as N increases. Let N' be the *minimum* value of N for which ß (N) is uni-modal. Also, let [?$^-$, ?$^+$] be the range of the unique mode, i.e., the bandwidth interval that includes all the significant values in the 'bell' around R. The heuristic rule with which the capacity estimate C is selected is that *the capacity mode is the minimum mode m_i in M which is higher than* ?$^+$, i.e.,

$$\hat{C} = m_k = \min\{m_i \in \mathcal{M} : m_i > \zeta^+\} \quad (15).$$

The heuristic is based on the following reasoning. First, when N is sufficiently large for ß (N) to be uni-modal, almost all packet trains have encountered dispersion due to cross traffic, and so ?$^+$ < C. Second, because N is the *minimum* packet train length that generates a unimodal ß (N), the range of the unique mode is still sufficiently wide to cover all the local modes in the SCDR of B between R and C. This heuristic resulted from long experimentation, and is evaluated later in this section.

In *ADR*, Phase II consists of a K_2=400 packet train experiments with L=1500B for each length N. If the resulting distribution ß (N) is not uni-modal, N is increased by two, and the process repeats. Note that K_2 is significantly lower than K_1, because in Phase II we only check whether the distribution is uni-modal, instead of estimating the local modes. When the length N = N' is reached, the

upper threshold $?^{+}$ is measured, and the capacity estimate m_k is determined from Equation 15.

Chapter 6 - *Java NIO*

6.1. Selectors

In this chapter, we'll explore selectors. Selectors provide the ability to do *readiness selection*, which enables *multiplexed I/O*. Readiness selection and multiplexing make it possible for a single thread to efficiently manage many I/O channels simultaneously. C/C++ coders have had the POSIX select( ) and/or poll( ) system calls in their toolbox for many years. Most other operating systems provide similar functionality. But readiness selection was never available to Java programmers until JDK 1.4. Programmers whose primary body of experience is in the Java environment may not have encountered this I/O model before.

The paradigm of quickly checking to see if attention is required by any of a set of resources, without being forced to wait if something isn't ready to go. This ability to check and continue is a key to scalability. A single thread can monitor large numbers of channels with readiness selection. The Selector and related classes provide the APIs to do readiness selection on channels.

6.1.1. Selector basics

You register one or more previously created selectable channels with a selector object. A key that represents the relationship between one channel and one selector is returned. Selection keys remember what you are interested in for each channel. They also track the operations of interest that their channel is

currently ready to perform. When you invoke select( ) on a selector object, the associated keys are updated by checking all the channels registered with that selector. You can obtain a set of the keys whose channels were found to be ready at that point. By iterating over these keys, you can service each channel that has become ready since the last time you invoked select( ).

At the most fundamental level, selectors provide the capability to ask a channel if it's ready to perform an I/O operation of interest to you. For example, a SocketChannel object could be asked if it has any bytes ready to read, or we may want to know if a ServerSocketChannel has any incoming connections ready to accept. Selectors provide this service when used in conjunction with SelectableChannel objects, but there's more to the story than that. The real power of readiness selection is that a potentially large number of channels can be checked for readiness simultaneously. The caller can easily determine which of several channels are ready to go. Optionally, the invoking thread can ask to be put to sleep until one or more of the channels registered with the Selector is ready, or it can periodically poll the selector to see if anything has become ready since the last check. If you think of a web server, which must manage large numbers of concurrent connections, it's easy to imagine how these capabilities can be put to good use.

At first blush, it may seem possible to emulate readiness selection with non-blocking mode alone, but it really isn't. Non-blocking mode will either do what you request or indicate that it can't. This is semantically different from determining if it's *possible* to do a certain type of operation. For example, if you attempt a non-blocking read and it succeeds, you not only discovered that a read( ) is possible, you also read some data.

This effectively prevents you from separating the code that checks for readiness from the code that processes the data, at least without significant complexity. And even if it was possible simply to ask each channel if it's ready, this would still be problematic because your code, or some code in a library package, would need to iterate through all the candidate channels and check each in turn. This would result in at least one system call per channel to test its readiness, which could be expensive, but the main problem is that the check would not be atomic. A channel early in the list could become ready after it's been checked, but you wouldn't know it until the next time you poll. Worst of all, you'd have no choice but to continually poll the list. You wouldn't have a way of being notified when a channel you're interested in becomes ready.

This is why the traditional Java solution to monitoring multiple sockets has been to create a thread for each and allow the thread to block in a read( ) until data is available.

This effectively makes each blocked thread a socket monitor and the JVM's thread scheduler becomes the notification mechanism. Neither was designed for these purposes. The complexity and performance cost of managing all these threads, for the programmer and for the JVM, quickly get out of hand as the number of threads grows.

True readiness selection must be done by the operating system. One of the most important functions performed by an operating system is to handle I/O requests and notify processes when their data is ready. So it only makes sense to delegate this function down to the operating system. The Selector class provides the abstraction by which Java code can request readiness selection service from the underlying operating system in a portable way.

6.1.2. Selection keys

Selector: The Selector class manages information about a set of registered channels and their readiness states. Channels are registered with selectors, and a selector can be asked to update the readiness states of the channels currently registered with it. When doing so, the invoking thread can optionally indicate that it would prefer to be suspended until one of the registered channels is ready.

Selectable Channel: This abstract class provides the common methods needed to implement channel select ability. It's the super class of all channel classes that support readiness selection. FileChannel objects are not selectable because they don't extend from SelectableChannel. All the socket channel classes are selectable, as well as the channels obtained from a Pipe object. SelectableChannel objects can be registered with Selector objects, along with an indication of which operations on that channel are of interest for that selector. A channel can be registered with multiple selectors, but only once per selector.

SelectionKey: A SelectionKey encapsulates the registration relationship between a specific channel and a specific selector. A SelectionKey object is returned from SelectableChannel.register( ) and serves as a token representing the registration. SelectionKey objects contain two bit sets (encoded as integers) indicating which channel operations the registrant has an interest in and which operations the channel is ready to perform.

6.1.3. Asynchronous closability

It's possible to close a channel or cancel a selection key at any time. Unless you take steps to synchronize, the states of the keys and associated channels could change unexpectedly. The presence of a key in a particular key set does not guarantee that the key is still valid or that its associated channel is still open.

Closing channels should not be a time-consuming operation. The designers of NIO specifically wanted to prevent the possibility of a thread closing a channel being blocked in an indefinite wait if the channel is involved in a select operation. When a channel is closed, its associated keys are cancelled. This does not directly affect an in-process select( ), but it does mean that a selection key that was valid when you called select( ) could be invalid upon return. You should always use the selected key set returned by the selector's selectedKeys( ) method; do not maintain your own set of keys.

If you attempt to use a key that's been invalidated, a CancelledKeyException will be thrown by most methods. You can, however, safely retrieve the channel handle from a cancelled key. If the channel has also been closed, attempting to use it will yield a ClosedChannelException in most cases.

6.1.4. Selection scaling

Selectors make it easy for a single thread to multiplex large numbers of selectable channels. Using one thread to service all the channels reduces complexity and can potentially boost performance by eliminating the overhead of

managing many threads. But is it a good idea to use just one thread to service all selectable channels? As always, it depends.

It could be argued that on a single CPU system it's a good idea because only one thread can be running at a time anyway. By eliminating the overhead of context switching between threads, total throughput could be higher. But what about a multi-CPU system? On a system with *n* CPUs, *n*–1 could be idling while the single thread trundles along servicing each channel sequentially.

Or what about the case in which different channels require different classes of service? Suppose an application logs information from a large number of distributed sensors. Any given sensor could wait several seconds while the servicing thread iterates through each ready channel. This is OK if response time is not critical. But higher-priority connections (such as operator commands) would have to wait in the queue as well if only one thread services all channels. Every application's requirements are different. The solutions you apply are affected by what you're trying to accomplish.

For the first scenario, in which you want to bring more threads into play to service channels, resist the urge to use multiple selectors. Performing readiness selection on large numbers of channels is not expensive; most of the work is done by the underlying operating system. Maintaining multiple selectors and randomly assigning channels to one of them is not a satisfactory solution to this problem. It simply makes smaller versions of the same scenario.

A better approach is to use one selector for all selectable channels and delegate the servicing of ready channels to other threads. You have a single point to monitor channel readiness and a decoupled pool of worker threads to handle the incoming data. The thread pool size can be tuned (or tune itself,

dynamically) according to deployment conditions. Management of selectable channels remains simple and simple is good.

The second scenario, in which some channels demand greater responsiveness than others, can be addressed by using two selectors: one for the command connections and another for the normal connections. But this scenario can be easily addressed in much the same way as the first. Rather than dispatching all ready channels to the same thread pool, channels can be handed off to different classes of worker threads according to function. There may be a logging thread pool, a command/control pool, a status request pool, etc.

Chapter 7 - *Technical specification - the architecture of the application*

7.1. General architecture

The application for measuring capacity along a network path was developed using Java 1.4. classes and methods. The structure of *ADR* is adapted to the new possibilities enabled by Java NIO library. It is fairly easy to use this structure for developing similar monitoring tools. Madalin Mihailescu used a similar structure for developing an available bandwidth estimation tool. All the classes are in the same package and there are three classes (Common, Globals and GlobalsRcv) which contain definitions for certain constants needed in the algorithm.

In the same package we have three libraries libudprcv, libudpsnd and libtime along with UDPRecv and UDPSend C implementations. In the next section we will explain the motivation for using C code. For integrating the C sources I used jini method and the headers are generated with the javah application.

The core of the application is the PathRateSnd and PathrateRcv classes. These two classes implement the server and the client needed for path estimation. All the other classes are used to enable the communication between the server and the client or to provide the client (receiver) with the appropriate methods for applying the ADR algorithm on obtained data.

The communication sender-receiver uses UDP packets of different sizes

and a TCP control connection. The TCP connection is the medium for forwarding control messages back and forth. The control message tell the receiver or the sender which action should it take (send packets, receive packets, compute time, change the value of global variables, suspend the algorithm and so on). The UDP packets are used for arrival time estimation. Generally the receiver receives a train of packets and time stamps the arrival time for each packet. All these data are then averaged, statistically modified in order to extract the Paxson bandwidth modes. In the next section I will describe in detail the class structure and the role of each class.

7.2. Class structure

We will provide a full package class description. The capacity tool estimation is layered implemented a nd we will go through succinctly trough each layer.

The first layer would be the sender and the receiver (***PathRateSnd*** and ***PathRateRcv***)or the server and the client. This is the only layer visible to a user launching the application. Meaning that if he chooses to test the tool he needs to start the execution of the sender (server) and then to start as many clients (receivers) as he wants.

The sender and receiver are not symmetrical like many socket programs. Instead the sender has very few lines which set up the future communication. The receiver is the largest class in the package (almost 1000 lines) and it implements the *ADR* algorithm. The sender has a general structure and may be used for measuring other network parameters.

The second layer is responsible of managing the client threads. The server can accept multiple requests from receivers. Therefore he needs an executing thread for each of these. This thread is implemented in the ***WorkerThread*** class. The *WorkerThread* waits for something to happen (a control message on a TCP connection checked by a Java NIO selector) and the objects of this class are managed by the ***ServiceThreadPool*** class. The *ServiceThreadPool* class uses a load balancing strategy implemented in the ***LoadBalancing*** class. This middleware layer leaves a lot of place to be improved. The classes are abstractly implemented and can be easily extended to enable further functionality.

The third layer is responsible with the high level communication between the clients. This level includes the classes ***ConnectionHandler***, ***ConnectionMethodsRcv*** and ***ControlListenThread***. The *ConectionHandler* class sets up the java socket communication and most importantly receives and decodes the control messages from the TCP connection. This class also calls the lower layer of communication for sending UDP packets. The *ConnectionMethodsRcv* is responsible for setting the UDP connection and for calling the lower layer of communication for receiving UDP packets. Finally the ControlListenThread listens on the TCP connection for a particular ending train control message.

The fourth and final level of the application is the lowest level of application. This level contains the ***UDPSend*** and ***UDPRecv***. These classes contain numerous C native code methods for setting low level communication. In the next section we will look closer at the need for C native code. The *UDPSnd* and *UDPRecv* call *udpsend* and *udprecv* libraries for compiling the C code.

Except this four levels which encapsulate layers of communication it is worth mentioning the rest of the classes and their use:

Common – a class for defining control code messages, communication ports and other important constants

GetTime – a class which uses a native C method for computing the current time in microseconds

Globals – a class which contains definitions of several constants

GlobalsRcv – a class with constant definitions needed by the receiver

UtilMethods – a sum of mathematical methods

UtilMethodsRcv – contains important methods related to the ADR algorithm implemented

7.3. The need for native C code

In this section we will attack the need for native C code implementation. At first when I tried to implement the algorithm I have chosen Java for many reasons. Java is more robust, platform independent, robust, flexible and a lot more. I have implemented all the application in this programming language. Unfortunately the results were very bad especially for links over 100Mbps. I studied this matter and discovered that the weak point was the rate at which Java can send UDP packets. More explicitly Java cannot send UDP packets lower than 40 microseconds and for Gigabyte links I needed 10 microseconds. This sounds like a secondary issue but if we think that I need to register the arrival times and put them in the algorithm, the issue becomes the core of the problem.

The C socket implementation provides the sending time needed for this

application. It is not easily configurable or provides methods for selection, but it does the low-level job. I implemented this part of communication in C and the results were conforming to what I expected. Therefore the communication uses the strong-points of these two influential programming languages.

7.4. Method summary

I will provide a list of the essential methods from every class and their point in the whole architecture. The classes are listed alphabetically and the methods are presented as they appear in the source code.

Common:

- public static void print(int ctr_code) – prints a control message

ConnectionHandler:

- public ConnectionHandler(SocketChannel sc, DatagramSocket socketUdp, WorkerThread wt) – constructor which sets the ConnectionHandler object on the SocketChannel sc, communicating trough the DatagramSocket socketUDP and managed by the WorkerThread wt
- public int readDataFromSocket() – reads and decodes a control message
- private int send_train() – calls the C code UDPSend procedure
- public int recv_ctr_msg() – returns the received control message
- public void send_ctr_msg(int ctr_code) – send the ctr_code message

ConnectionMethodsRcv:

- public ConnectionMethodsRcv(SocketChannel sc) – constructor which sets the ConnectionMethodsRcv object on SocketChannel sc
- public int getRecvLatency() – returns the receiver latency by calling the C procedure recvfromLatency
- public int getDelta() – returns a useful time difference for the sender
- public int recv_train(int exp_train_id, int[] timesec, int[] timeusec, int train_len) – calls the UDP correspondent C procedure and registers the arrival times in timesec and timeusec

ControlListenThread:

- public void run () – reads a control message and if the message is FINISHED_TRAIN than it sets the global variable finised_train

Gettime:

- public native void getTimeOfDay() – native C code method for extracting the current system time

LoadbalancingStrategy:

- public LoadBalancingStrategy(int unitsNo) – sets a load balancing vector of unitsNo objects
- public void addNewUnit(Object unit) – adds a new object to the vector
- public boolean ready() – returns the status for the usability of the objects
- public synchronized Object selectUnitForNewTask() – selects an object for a particular task
- public synchronized void taskFinished(Object o) – sets the flag for finished task and frees up the object
- private int findInsertPosition(int low, int high) – inserts a unit at the middle of the vector of objects

PathrateRcv:

- public PathrateRcv(String address) – constructs a receiver object which communicates with the server given by address
- public void init() – initialization for global receiver variables
- protected void registerForOperations(SocketChannel sc, int ops) – registers a socket channel for the operations coded in ops
- public void service () – executes the whole receiver part of the algorithm

PathrateSnd:

- public PathrateSnd(int unitsNo) – defines a new sender and initializes the service thread pool vector
- public void openChannel() – opens the TCP channel
- public void service() – waits for something to happen using the NIO selector. If something is received than it adds a new task to handle the request.
- protected void registerChannel (Selector selector, SelectableChannel channel, int ops) – registers the channel with the given selector for given operations

ServiceThreadPool:

- public ServiceThreadPool(int unitsNo) – initializes a new service thread pool
- public void addNewTask(SocketChannel sc) – selects a worker thread to handle the socket channel sc and adds the task to this worker

UDPRcv

- public native int getUDPSocket(int UDPRCV_PORT,int UDP_BUFFER_SZ) – sets up the UDP linked to the UDPRCV_PORT and with maximum buffer size UDP_BUFFER_SZ and returns it
- public native void closeUDPSocket(int sock_udp) – closes the UDP socket
- public native int recvTrain(int max_pkt_sz, int exp_train_id, int train_len, int sock_udp) – receives the train exp_train with the length of train_len packets and the maximum packet size of max_pkt_sz. Also it sets up the arrival time arrays attributes of this class
- native int delta(int max_pkt_sz, int sock_udp, int UDPRCV_PORT) – returns the average time between sending and receiving a UDP packet needed in the receiver
- public native int recvfromLatency(int sock_udp, int UDPRCV_PORT, int max_pkt_sz) – returns the receiver latency of the medium by sending 50 packets back and forth and returning the average latency time

UDPSnd:

- public native int getUDPSocket(int UDP_BUFFER_SZ) - sets up the UDP linked with maximum buffer size UDP_BUFFER_SZ and returns it
- public native void closeUDPSocket(int sock_udp) – closes the UDP socket
- public native int sendLatency(int max_pkt_sz) - returns the sender latency of the medium by sending 50 packets back and forth and returning the average latency time
- public native int sendTrain(int max_pkt_sz, int train_len, int train_id, int sock_udp, String address, int UDPRCV_PORT) – sends the train train_id with a length of train_len packets and maximum packet size max_pkt_sz

UtilMethods:

- public static int timeToUsdelta(int sec1, int usec1, int sec2, int usec2) – returns the time difference between (sec1, usec1) and (sec2, usec2)
- public static void orderInt(int unord_arr[], int ord_arr[], int num_elems) – order an integer vector using the bubble sort algorithm
- public static void minSleepTime() – returns the minimum sleep time for system calls

UtilMethodRcv:

- public static int check_intr_coalescence(int[] timesec,int[] timeusec, int len, int burst) – returns the coalescence for time stamps given in timesec and timeusec with a certain burst
- public static double getKurtosis(double bell_array[], int size) – returns the Kurtosis value from the bell array
- public int termint(int exit_code) – ends the measurements
- public static void happyEnd(double cap_lo, double cap_hi) – the algorithms finishes normally and this method prints the results
- public static double getAvg(double data[], long no_values) – returns the average of data array
- public static double getStd(double data[], long no_values) – returns the standard deviation of data array
- public double getMode(double ord_data[], int vld_data[], double bin_wd, int no_values) – returns the local mode for the data array with the bin bin_wd. It also sets up the attributes mode_cnt, bell_cnt, bell_lo, bell_hi and bell_kurtosis
- public int gig_path(int pack_sz, int round, double k_to_u_latency) – procedure called for GigaByte heavy-loaded paths with interrupt colescence

WorkerThread:

- public WorkerThread(LoadBalancingStrategy lbs) – initiates the worker thread with the specified load balancing strategy
- public void run() – waits for some action and when there are multiple active selector it iterates and reads data from the corresponding socket channels
- public void addNewTask(SocketChannel sc) – wakes up the selector and adds the task related to the socket channel sc
- protected void registerForOperations(SocketChannel sc, int ops) – register the channel with the specified operations
- protected void unregisterForOperations(SocketChannel sc, int ops) – unregisters the specified operations
- public void eraseChannel(SelectionKey key) – erases the channel

Chapter 8 - *The structure of the MonALISA module cap*

8.1. Incorporation in MonALISA

The tool for measuring capacity is useful as a standalone application, but its use increases considerably when integrated in a complex monitoring framework like MonALISA. Any user who accesses the site cacr.caltech.edu and launches the MonALISA client can check out the data from the *cap* module. Furthermore the team of programmers who maintain this complex project can use this tool in any moment to provide an estimate for the capacity of a path. There are thousands of network paths between farms supervised by MonALISA and each has a fixed capacity. On every network change the *cap* module can detect dropped nodes or the new upgraded path capacities.

8.2. Configuration parameters

For integrating the tool in the MonALISA I have done several configuration steps. At first I had to include my files in the package lia.Monitor.Farm.Pathrate. Then I have changed the headers of the C implementation files and recompiled them. At this point the sources were ready to be used in this framework.

I have implemented two sources:

- lia.Monitor.modules - the monPathrate class which handles the configuration, the connection settings and gets the results;

- lia.App.Pathrate – the AppPathrate class which handles the logic of

server farm for my module.

Further we need a configuration file on an URL containing the names of hosts that need to be supervised. We will take a close look at the two classes, the core of the integration issue.

8.2.1. The monPathrate class

This class extends the cmdExec and implements the MonitoringModule interface. The attributes of this class initialize basic variables for MonALISA integration: the logger file, the name of the module, the resource type string, the operating system, the configuration URL, the configuration reload interval, the reload interval of the module. The class uses three vectors which contain lists of hostnames, peers and removed peers. Naturally we have to instantiate a PathrateRcv (receiver - client) object for result extraction.

The configuration is loaded using the TimerTask class and the reload intervals are set. The PathrateReceiver class extends the TimerTask and it starts all the clients in the vector of peers for the server. The class ConfigLoader parses the configuration URL file and adds every receiver to the peer Vector. The most important method for this class is doProcess. This method iterates through the list of peers and gets the results from every measurement from the receivers. The result is added to the result vector that is fetched by AppPathrate for output.

8.2.1. The AppPathrate class

The AppPathrate implements the lia.app.AppInt Interface. It has attributes for the file name, the properties object, the configuration options, the execution

parameters the path to MonALISA. It uses the sender object of the sender. This is the server that needs to be started, stopped and restarted and get information about the server. This class extracts the needed execution information from configuration files or updates the files.

Chapter 9 - *Performance*

9.1. The accuracy of ADR, results

We have tested this tool and compared it to previous tests with other capacity estimation applications. Due to the consistency of the results and the precision of the estimation we can say that it performs admirably in various environments. It generates traffic of 1.56GB with an occupied bandwidth of 4MBps.

The average execution time is 400 seconds, but it can provide the result even in 350s, a good time for capacity estimation. We have tested the following 100MB paths:

- rogrid.pub.ro -> monalisa.cern.ch 96MB
- monalisa.cern.ch -> rogrid.pub.ro 88MB
- rogrid.pub.ro -> monalisa-starlight.cern.ch 80MB
- monalisa-starlight.cern.ch -> rogrid.pub.ro 88 MB
- monalisa.cern.ch -> monalisa-starlight.cern.ch 80 MB
- monalisa-starlight.cern.ch -> monalisa.cern.ch 93 MB

After testing on Gigabyte links we have obtained 900-930MBps. Next we will provide a history chart the output of the cap module during two hours of cyclic execution:

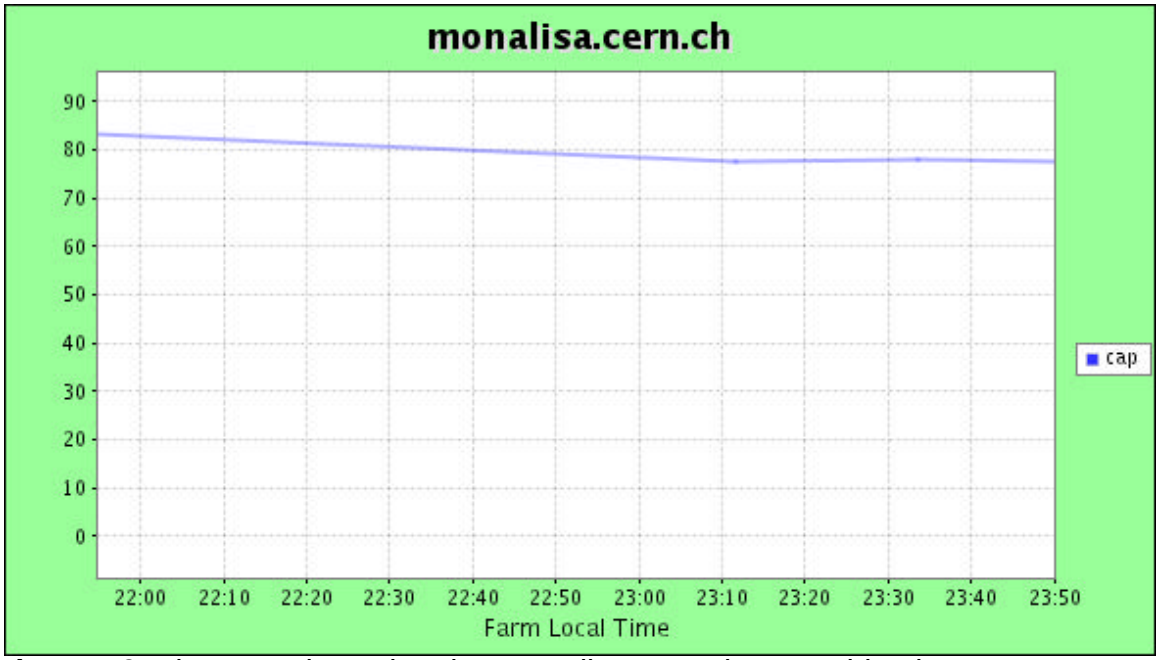

Figure 10: The capacity estimation monalisa.cern.ch -> rogrid.pub.ro

9.2. Generated traffic

We will present an estimation of the generated traffic. The traffic is computed starting from the receiver's code. The sender has the role of a server, for instance it sends a train of packets when he receives the Common.SEND control message on the message control TCP connection. So we examine closely the receiver.

The receiver main code is in the service method. The receiver initiates the sockets and then it computes the latency.

GlobalsRcv.recvLatency = cmrcv.getRecvLatency();

It calls the getRecvLatency method from ConnectionsMethodRcv and then the code is forwarded to the C implementation. Here 50 packets of maximum size are changed. The maximum packet size is 1472 bytes, so we change 50M packets (M = 1472 bytes).

Next in the receiver we compute the minimum possible delta. The UDP receiver calls the delta method which sends and receives 400 packets of maximum packet size. So we add 400M bytes

The receiver needs to discover the maximum feasible train length in the path. For that task it sends trains of packets with increased length and maximum size. The number of packets is:

(2 + 3 + 4 + 5 + 6 + 8 + 10 + 12 + 16 + 20 + 24 + 28 + 32 + 36 + 40 + 44 + 48)M = 338M bytes

The receiver checks for possible channel links and traffic shapers by increasing packet train lengths with 7 trains:

$$\frac{49 \bullet 48}{2} \bullet 7 \bullet M = 1176 \bullet M = 8232 bytes$$

For detecting possible capacity modes the client receives 1000/40 = 25 = no_trains_per_size. The pack_incr_step = 23, m = 572bytes. This adds to:

$$2 \bullet 25[m + m + 23 + .. + 36m + 36 \bullet 23 + 4M] = 50(36m + 4M + 23 \bullet 666) = 50(36m + 4M + 15318)$$
$$= 50 \bullet 30332630 = 1516631500 bytes$$

For detecting local modes we do not send or receive packets. The receiver calls the getMode method which uses the receiving time array to detect the local modes.

Next we estimate the asymptotic dispersion rate by receiving 500 trains of maximum train length. The standard maximum train length is 48, and the number of packets adds to $500 \bullet 48 \bullet M$.

For detecting the local modes in phase I, we do not change any packets. Now we add all the data obtained in these paragraphs before. We call MPC (maximum packet changed) the resulting value:

$$MPC = 50 \bullet 1472 + 400 \bullet 1472 + 338 \bullet 1472 + 8232 \bullet 1472 + 1516631500 + 3532800 =$$
$$73600 + 588800 + 497536 + 12117504 + 1516631500 + 3532800 =$$
$$1{,}565{,}236{,}940 bytes = 1565236 KB = 1565 MB = 1.565 GB$$

The generated traffic is considerable, but if we think that the algorithm is launched at very long periods of times (days or even weeks) the result is not so discouraging. Furthermore for slow links (under 100Mbps) we can configure the tool to generate traffic ten times smaller. The algorithm occupies:

bandwidth = 1565MB/400s = 3.91MBps (400s the estimated execution time). This bandwidth for measurement is significant for 10MBps link, but if we think that the tool is conceived for Gigabyte links the percentage is negligible.

9.3. Conclusions

To sum up, we need to take a more objective look at this tool. It has advantages and disadvantages, but as I will show it is very useful overall. Let's start with the disadvantages:

- for slow links the traffic is considerable;

- the obtained results can vary in a 3-4 Mbps capacity;

The first disadvantage has been detailed in the previous chapter in conclusion the traffic is small for links faster than 10MBps. Furthermore from all the tools tested that measure the path capacity this generates the smallest traffic. Generally for measuring the capacity we need to change a lot of packets between the server and the client.

The second and last disadvantage is negligible because the purpose of the tool is to guess the path capacity. If we obtain an 83MBps capacity it is clearly that the capacity is 100MBps or for 900MPps.

Now the main characteristics of the applications are:

- *Robust* – the tool is adaptable to many environments;

- *Easy configurable* – the application can be configured. There are three classes (Common, Global, GlobalsRcv) that contain all the used constants;

- *C core packet transfer* – the optimum packet UDP communication is assured by the C programming language;

- *Extendable* – the structure of the application is easily expandable. A programmer can extend the sender or receiver to add further functionalities;

- *Layered structured* – as I discussed earlier the application has a four layered structure;

- *Logs* – it generates exhaustive, comprehensive logs. The Common class contains the Verbose variable that controls the level of complexity for the log file;

- *Platform independent* – the application is implemented in Java so it is independent of the machine used;

- *Friendly interface* – it has a friendly interface and prints results easy to understand for any user, even if it is not in the Computer Science field;

There are numerous advantages of this tool as compared to other similar applications. I strongly believe that it will be a landmark in the field of network monitoring host tools and it will prove very useful to the MonALISA framework.

9.4. Future improvements

The most important improvement of this technique is to measure the capacity of every link along the path. I will propose another algorithm similar to the K. Lai nettimer. I believe that by using the tailgating algorithm we can improve dramatically the use of the *cap* module.

We describe a new deterministic model of packet delays that unifies previous models. Using this model, we derive a novel technique, called packet tailgating, for measuring link bandwidths along a path through the Internet. Packet tailgating captures link-specific characteristics by causing queuing of packets at particular links. For each link, the technique sends a large packet with a time-to-live (TTL) set to expire at that link followed by a very small packet that will queue continuously behind the large packet until where the large packet expires.

The advantages of the tailgating technique are its speed, unobtrusiveness

and robustness compared to the other link bandwidth techniques. Packet tailgating is potentially faster and less obtrusive than the previously discussed techniques because it performs the expensive linear regression step only once for the entire path instead of once for every link. The tailgating step has to be done for every link, but this only requires finding the minimum delay for a pair of packets compared to finding the minimum delay of 16 to 64 different packet sizes. We test this hypothesis using nettimer in the next section.

Packet tailgating is potentially more robust because it can detect multi-channel links, does not rely on timely delivery of ICMP packets, and can be run without acknowledgements.

Packet tailgating can measure multi-channel links because, like the packet bunch extension to packet pair, it can send multiple packets to fill the channels. To do this, we send the tailgated packet followed by any number of tailgaters. We can use this queuing to measure the bandwidth of one of the channels (we assume that all the channels have the same bandwidth).

Packet tailgating does not use ICMP time-exceeded packets from intermediate nodes for measurement. In fact, packet tailgating can work without acknowledgments at all from the destination. The tailgating source can continue to send later tailgater packets without knowing the delay of earlier tailgater packets. If the earlier delays can be occasionally transmitted back to the source, then the source can adaptively decide when to finish the two stages, but this is an optimization. Otherwise, the tailgater source can just send a fixed number of packets in each stage. Eventually the source and destination must communicate so that the source can specify which tailgated packets were prematurely dropped.

Measuring without acknowledgements avoids queuing in the return path, enabling packet tailgating to be twice as accurate as single-packet techniques. In addition, measuring without acknowledgements avoids ack-implosion on multicast trees, enabling packet tailgating to measure the bandwidth of several links simultaneously on a multicast tree. Single packet techniques used on a multicast tree would cause a flood of acknowledgements to flow back to the source. The queuing of these acks would likely destroy their usefulness for round trip delay measurements.

Chapter 10 – References

[1] O'Reilly – Java NIO

[2] C. Dovrolis, P. Ramanathanm si D. Moore – Packet Dispersion Techniques and a Capacity Estimation Methodology

[3] C. Dovrolis, P. Ramanathanm si D. Moore – What do packet dispersion techniques measure

[4] Active measurement project – NLANR group
[5] J. C. Bolot – Characterizing End-to-End Packet Delay and Loss in the Internet
[6] R. L. Carter si M. E. Crovella – Measuring Bottleneck Link Speed in Packet-Switched Networks
[7] A.B. Downey – Using Pathchar to estimate Internet Link Characteristics
[8] V. Jacobson – Congestion Avoidance and Control
[9] M. Jain si C. Dovrolis – End-to-End Available Bandwidth: Measurement methodology, Dynamics and Relation with TCP Throughput
[10] K. Lai si M. Baker – Measuring bandwitdh
[11] A. Pasztor si D. Veitch – The Packet Size Dependence of Packet Pair Like Methods
[12] V. Paxson – Measurement and Analysis of End-to-End Internet Dynamics
[13] A. B. Downey – A Tool for Estimating Internet Links Characteristics

Chapter 11 - Code listing

UtilMethodsRcv.java

```
public class UtilMethodsRcv{
    long mode_cnt;long bell_cnt; double bell_lo; double bell_hi; double bell_kurtosis;
```

```
ByteBuffer buffer;
CharBuffer charBuffer;
UDPRecv udpr = new UDPRecv();
PathrateRcv pr;
public UtilMethodsRcv(PathrateRcv pr){
buffer = ByteBuffer.allocateDirect(11);
charBuffer = CharBuffer.allocate(11);
this.mode_cnt = 0;this.bell_cnt = 0;this.bell_lo = 0;this.bell_hi = 0;this.bell_kurtosis = 0;this.pr = pr;
}
public static int check_intr_coalescence(int[] timesec,int[] timeusec, int len, int burst){
        double[] delta = new double[Common.MAX_STREAM_LEN];
        int b2b=0, tmp=0;int i;int min_gap;

    min_gap = GlobalsRcv.MIN_TIME_INTERVAL > 3 * GlobalsRcv.recvLatency?GlobalsRcv.MIN_TIME_INTERVAL :
3 * GlobalsRcv.recvLatency ;
        for (i = 2; i < len; i ++){
                delta[i] = (timesec[i] - timesec[i-1])*1000000 + timeusec[i] - timeusec[i-1];
                if ( delta[i] <= min_gap ){
                        b2b++ ; tmp++;
                }
                else{
                        if ( tmp >=3 ){
                                burst++;tmp=0;
                        }
                }
        }
        if ( b2b > .6*len ){
                return 1;}
        else return 0;
}
//Extracts the last part of the vector v starting at pos with max elements
public static double[] extractLast(double v[], int pos, int max){
        double[] newv = new double[max];
        for (int i = pos;i < max; i++){
                newv[i - pos] = v[i];
        }
        return newv;
}
// Order an array of int using bubblesort
public static double[] orderInt(double unord_arr[], int num_elems){
        int i,j ;double temp; double[] ord_arr = new double[num_elems];
        for (i=0;i<num_elems;i++)
                ord_arr[i]=unord_arr[i];
        for (i=1;i<num_elems;i++) {
```

```
                    for (j=i-1;j>=0;j--)
                            if (ord_arr[j+1] < ord_arr[j]){
                            temp=ord_arr[j];ord_arr[j]=ord_arr[j+1];ord_arr[j +1]=temp;
                            }
                            else break;
            }
            return ord_arr;
}
public static double getKurtosis(double bell_array[], int size){
            double kurtosis, var, mean, temp, diff;int i;
            if (size < 3)           return -99999;

            temp = 0;
            for(i = 0; i < size; i++)temp += bell_array[i];
            mean = temp / size;temp = 0;
            for(i = 0; i < size; i++){
                    diff = bell_array[i] - mean;
                    temp += diff * diff;
            }
            var = temp / size;
            if(var == 0)return -99999;
            temp = 0;
            for(i = 0; i < size; i++){
                            diff = bell_array[i] - mean;
                            temp += diff * diff * diff * diff;
                    }
            kurtosis = temp / (var * var);
            return kurtosis;
}
//Terminate measurements
public int termint(int exit_code){
            int ctr_code;
            ConnectionMethodsRcv cmrcv = new ConnectionMethodsRcv(GlobalsRcv.sc);
            ctr_code = Common.GAME_OVER;
            cmrcv.send_ctr_msg(ctr_code);
            try{ GlobalsRcv.sc.close();} catch (Exception ignored){}
            udpr.closeUDPSocket(GlobalsRcv.sock_udp);
            return exit_code;
}
/* Successful termination. Print result.  */
public void happyEnd(double cap_lo, double cap_hi){
            System.out.println("-------------------------------------------------");
            System.out.println("Capacity estimate : " + (int)cap_lo + " to " + (int)cap_hi);
            System.out.println("Final capacity estimate : " + (int)((cap_lo + cap_hi)/2) );
```

```
            System.out.println("-------------------------------------------------");
            //good measurement --> add it to the vector
            synchronized(pr.myPeers) {
                  Double cap = new Double(((cap_lo + cap_hi)/2));
                  System.out.println("[happy end]");
                  if (pr.myPeers.containsKey(pr.address)) {
                        Vector values = (Vector)pr.myPeers.get(pr.address);
                        values.add(cap);
                  }
                  else {
                        Vector values = new Vector();
                        values.add(cap);
                        pr.myPeers.put(pr.address, values);
                  }
            }
            UDPRecv udpr = new UDPRecv();
            udpr.closeUDPSocket(GlobalsRcv.sock_udp);
            try { GlobalsRcv.sc.close();} catch (Exception e) {}
            GlobalsRcv.po.close();
      }
      //Compute the average of the set of measurements <data>.
      public static double getAvg(double data[], long no_values){
            int i;double sum;sum_ = 0;
            for (i=0; i<no_values; i++) sum_ += data[i];
            return (sum_ / (double)no_values);
      }
      //Compute the standard deviation of the set of measurements <data>.
      public static double getStd(double data[], long no_values){
            int i;double sum_, mean;mean = getAvg(data, no_values);
            sum_ = 0;
            for (i=0; i<no_values; i++) sum_ += Math.pow(data[i] - mean, 2.);
            return Math.sqrt(sum_ / (double)(no_values-1));
      }
      /*
      Detect a local mode in the set of measurements <ord_data>.
      Take into account only the valid (unmarked) measurements (vld_data[i]=1).
      The bin width of the local mode detection process is <bin_wd>.
      The set has <no_values> elements, and it is ordered in increasing sequence.
      The function returns the center value of the modal bin.
      Also, the following call-by-ref arguments return:
      mode_cnt:     # of measurements in modal bin
      bell_cnt:     # of measurements in entire mode
      (modal bin+surrounding bins of the same `bell')
```

```
        bell_lo:    low bandwidth threshold of modal `bell'
        bell_hi:    high bandwidth threshold of modal `bell'
        */
        public double getMode(double ord_data[], int vld_data[], double bin_wd, int no_values){
                int i, j, done, tmp_cnt, mode_lo_ind, mode_hi_ind, bell_lo_ind, bell_hi_ind, bin_cnt, bin_lo_ind, bin_hi_ind, lbin_cnt, lbin_lo_ind=0, lbin_hi_ind=0, /* left bin */ rbin_cnt, rbin_lo_ind=0, rbin_hi_ind=0; /* right bin */
                double mode_lo, mode_hi, bin_lo, bin_hi;
                /*Weiling: bin_toler, used as */
                double bin_cnt_toler;

                j=0;
                for (i=0; i<no_values; i++) j+=vld_data[i];
                if (j==0)   return Common.LAST_MODE; /* no more modes */

                //Find the bin of the primary mode from non-marked values
                /* Find window of length bin_wd with maximum number of consecutive values */
                mode_hi=0; mode_hi_ind=0; mode_lo=0; mode_lo_ind=0;tmp_cnt=0;
                for (i=0;i<no_values;i++) {
                        if (vld_data[i] == 1)  { j=i;
                                while (j<no_values && (vld_data[j] == 1) && ord_data[j]<=ord_data[i]+bin_wd)
                                        j++;
                                if (tmp_cnt<j-i){
                                tmp_cnt = j-i;
                                mode_lo_ind = i;
                                mode_hi_ind = j-1;
                                }
                        }
                }
                this.mode_cnt = tmp_cnt;
                mode_lo = ord_data[mode_lo_ind];
                mode_hi = ord_data[mode_hi_ind];
                if (Common.Verbose)
                        System.out.println("Central mode bin from " + mode_lo +" to " + mode_hi + this.mode_cnt);
                this.bell_cnt = tmp_cnt;this.bell_lo = mode_lo;this.bell_hi = mode_hi;
                bell_lo_ind = mode_lo_ind;bell_hi_ind = mode_hi_ind;

                /*Find all the bins at the *left* of the central bin that are part of the same mode's bell. Stop when another local mode is detected*/
                bin_cnt = (int)this.mode_cnt;bin_lo_ind = mode_lo_ind;
                bin_hi_ind = mode_hi_ind;bin_lo = mode_lo;bin_hi = mode_hi;
                /* Weiling:  noise tolerance is determined by bin_cnt_toler, and it's
                * proportional to previous bin_cnt instead of constant BIN_NOISE_TOLER .
                */
```

```
done=0;
bin_cnt_toler = GlobalsRcv.BIN_CNT_TOLER_kernel_percent * (bin_cnt);
do{
        /* find window of measurements left of the leftmost modal bin with
        (at most) bin_wd width and with the maximum number of measurements */
        lbin_cnt=0;
        if (bin_lo_ind>0){
                for (i=bin_hi_ind-1; i>=bin_lo_ind-1; i--) {
                        tmp_cnt=0;
                for (j=i; j>=0; j--){
                        if (ord_data[j]>=ord_data[i]-bin_wd)
                                        tmp_cnt++;
                        else break;
                }
                        if (tmp_cnt >= lbin_cnt){
                                lbin_cnt = tmp_cnt;
                                lbin_lo_ind = j+1;
                                lbin_hi_ind = i;
                        }
                }
        }
        if (Common.Verbose)
                System.out.println("Left bin from " + ord_data[lbin_lo_ind] +" to " +
ord_data[lbin_hi_ind]);
        if (lbin_cnt>0){
                if (lbin_cnt < bin_cnt+bin_cnt_toler){
                        this.bell_cnt += bin_lo_ind-lbin_lo_ind;
                        this.bell_lo    = ord_data[lbin_lo_ind];
                        bell_lo_ind = lbin_lo_ind;
                        /* reset bin counters for next iteration */
                        bin_cnt = lbin_cnt;bin_lo_ind = lbin_lo_ind;
                        bin_hi_ind = lbin_hi_ind;bin_lo = ord_data[lbin_lo_ind];
                        bin_hi = ord_data[lbin_hi_ind];
                        bin_cnt_toler = GlobalsRcv.BIN_CNT_TOLER_kernel_percent
* (bin_cnt);
                }
                else{
                        /* the bin is outside the modal bell */
                        done=1;
                }
                if (bin_lo_ind <= 1)
                        done=1;
        }
        else done=1;
```

```
} while (done == 0);
/*Find all the bins at the *right* of the central bin that are part of the same mode's bell. Stop when another local mode is detected.*/
bin_cnt = (int)this.mode_cnt;bin_lo_ind = mode_lo_ind;
bin_hi_ind = mode_hi_ind;bin_lo = mode_lo;bin_hi = mode_hi;
done=0;
do{
    /* find window of measurements right of the rightmost modal bin with
    (at most) bin_wd width and with the maximum number of measurements*/
    rbin_cnt=0;
    if (bin_hi_ind<no_values-1){
        for (i=bin_lo_ind+1; i<=bin_hi_ind+1; i++){
            tmp_cnt=0;
            for (j=i; j<no_values; j++){
                if (ord_data[j]<=ord_data[i]+bin_wd) tmp_cnt++;
            else break;
        }
        if (tmp_cnt >= rbin_cnt){
            rbin_cnt = tmp_cnt;rbin_lo_ind = i;rbin_hi_ind = j-1;
            }
        }
    }
    if (Common.Verbose)
        System.out.println("Left bin from " + ord_data[lbin_lo_ind] +" to " + ord_data[lbin_hi_ind]);
    if (rbin_cnt>0){
        if (rbin_cnt < bin_cnt+bin_cnt_toler){
            /* the bin is inside the modal bell */
             /* update bell counters */
            this.bell_cnt += rbin_hi_ind-bin_hi_ind;
            this.bell_hi = ord_data[rbin_hi_ind];
            bell_hi_ind = rbin_hi_ind;
            /* reset bin counters for next iteration */
            bin_cnt = rbin_cnt;
            bin_lo_ind  = rbin_lo_ind;
            bin_hi_ind  = rbin_hi_ind;
            bin_lo = ord_data[rbin_lo_ind];
            bin_hi = ord_data[rbin_hi_ind];
            bin_cnt_toler = GlobalsRcv.BIN_CNT_TOLER_kernel_percent * (bin_cnt);
        }
        else{
        /* the bin is outside the modal bell */
        done=1;}
        if (rbin_hi_ind >= no_values-2) done=1;
```

```
                    }
                    else done=1;
                } while (done == 0);
                /* Mark the values that make up this modal bell as invalid */
                for (i=bell_lo_ind; i<=bell_hi_ind; i++) vld_data[i]=0;
                /* Report mode characteristics */
                if (this.mode_cnt > GlobalsRcv.BIN_NOISE){
                    if (Common.Verbose) {
                        System.out.println(mode_lo + mode_hi);
                        System.out.println(this.mode_cnt + "measurements");
                        System.out.println("Modal bell: " + this.bell_cnt + "measurements - low :"
+ this.bell_lo + "high:" + this.bell_hi);
                    }
                    /* Weiling: calculate bell_kurtosis*/
                    this.bell_kurtosis = getKurtosis(ord_data , bell_hi_ind - bell_lo_ind + 1);
                    if(this.bell_kurtosis == -99999){
                        System.out.println("\nWarning!!! bell_kurtosis == -99999\n");
                        return Common.UNIMPORTANT_MODE;
                    }
                    /* Return the center of the mode, as the average between the high and low
                    ends of the corresponding bin */
                    return (mode_lo + mode_hi)/2.;
                }
                else    return Common.UNIMPORTANT_MODE;
            }
            //Trying long trains to detect capacity in case interrupt coalescing detected.
            public int gig_path(int pack_sz, int round, double k_to_u_latency){
                int i, j, est = 0, ctr_code;
                int[] id = new int[Common.MAX_STREAM_LEN];
                int[] disps = new int[Common.MAX_STREAM_LEN];
                double[] cap = new double[Common.MAX_STREAM_LEN];
                double[] ord_cap = new double[Common.MAX_STREAM_LEN];
                int[] time_s = new int[Common.MAX_STREAM_LEN];
                int[] time_us = new int[Common.MAX_STREAM_LEN];
                int train_len=200, bad_train;
                System.out.println("  Testing for Gigabit paths\n  20 Trains of length: " +  train_len);
                for (j=0; j<10; j++){
                    int num_b2b = 1;int k=0;
                    ConnectionMethodsRcv cmrcv = new ConnectionMethodsRcv(GlobalsRcv.sc);
                    if (Common.Verbose) System.out.println("  Train id: " + j);
                    ctr_code = Common.TRAIN_LEN | (train_len<<16);
                    cmrcv.send_ctr_msg(ctr_code);
                    ctr_code= Common.PCK_LEN | (pack_sz<<16);
                    cmrcv.send_ctr_msg(ctr_code);
```

```
round++; ctr_code = Common.SEND | Common.CTR_CODE;
cmrcv.send_ctr_msg(ctr_code);
int train_id = 0;
bad_train = cmrcv.recv_train(train_id, time_s, time_us, train_len);
if (bad_train == 2)
        {/* train too big try smaller */
        train_len-=20;
        if (train_len < 100 && est < 5)  {
        System.out.println("Insufficient number of packet dispersion estimates.\n");
        System.out.println("Insufficient number of packet dispersion estimates.\n");
        System.out.println("Probably a 1000Mbps path.\n");
        System.out.println("Probably a 1000Mbps path.\n");
        termint(-1);
        }
}
else{
        for (i=1; i<train_len; i++){
                disps[i]  =  UtilMethods.timeToUsdelta(time_s[i],  time_us[i],
time_s[i + 1], time_us[i + 1]);
        }
        for (i=1; i<train _len; i++){
        if ( (num_b2b<5) ||(disps[i] < num_b2b*1.5*k_to_u_latency) ){
                num_b2b++;
        }
        else{
                id[k++]=i;
                num_b2b = 1;
        }
}
for (i=0; i<k-1; i++) {
        cap[est]                =                12000.0*(id[i+1]-id[i]-1)/
UtilMethods.timeToUsdelta(time_s[i], time_us[i], time_s[i + 1], time_us[i + 1]);
        if (Common.Verbose)          System.out.println(cap[est]);
        }
}
}
if ( est > 3 ){
        ord_cap = orderInt(cap,est–);
        happyEnd(ord_cap[est/2-1], ord_cap[est/2+1]);
}
else{
        System.out.println("Insufficient number of packet dispersion estimates.\n");
        System.out.println("Insufficient number of packet dispersion estimates.\n");
        System.out.println("Probably a 1000Mbps path.\n");
```

```
                    System.out.println("Probably a 1000Mbps path.\n");
                    termint(-1);
                }
                termint(0);
                return(1);
        }
}
```

PathrateRcv.java

```
public class PathrateRcv{
        int tcpSndPort = Common.TCPSND_PORT;
        static Selector selector;String address;InetSocketAddress socketAddress;
        Charset charset;CharsetDecoder decoder;CharsetEncoder encoder;
        ByteBuffer buffer;CharBuffer charBuffer;
        UtilMethodsRcv urcv = new UtilMethodsRcv(this);
        public Hashtable myPeers;
        UDPRecv udpr = new UDPRecv();
        public PathrateRcv() {
                System.out.println("Initializing receiver");
                myPeers = new Hashtable();}
        public void startClient (String addr) {
                address = addr;System.out.println("[PathrateRcv] : " + address);
                init();service();}
        public void fillResults(Result result) {
                synchronized (myPeers) {
                        Vector savb = (Vector) myPeers.get(result.NodeName);
                        result.param[0] = ((Double)savb.get(savb.size()-1)).doubleValue();
                        System.out.println("We have added " + result.param[0]);}
        }
        public void init() {
                socketAddress = new InetSocketAddress(address, tcpSndPort);
                charset = Charset.forName("ISO-8859-1"); decoder = charset.newDecoder();
                encoder = charset.newEncoder(); buffer = ByteBuffer.allocateDirect(1024);
                charBuffer = CharBuffer.allocate(1024); GlobalsRcv.po = new PrintOutput("client_cap.log");
                GlobalsRcv.interrupt_coalescence=0;GlobalsRcv.min_time_interval = 0;
                GlobalsRcv.sendLatency = 0; GlobalsRcv.recvLatency = 0;
                GlobalsRcv.max_rate = 0; GlobalsRcv.min_rate =0;
                GlobalsRcv.cur_pkt_sz = 0; GlobalsRcv.transmission_rate = 0;
                GlobalsRcv.time_interval = 0;
                UDPRecv udpr = new UDPRecv();}
                public void service (){
                int opt_len = 4, rcv_buff_sz, outlier_lim, pack_incr_step;
                boolean quick_term = false;int i, j, c, sum_rtt, ctr_code, round, train_len, prev_train_len = 2;
                int train_len_P1, max_train_len, no_pack_sizes, no_trains, no_trains_P1;
```

```
int no_trains_per_size, trains_msrd = 0, trains_rcvd = 0, train_no;
int trains_per_size, no_modes_P1 = 0, no_modes_P2 = 0;
int cap_mode_ind=0, bad_tstamps;
double train_spacing, min_train_spacing, max_train_spacing;
int max_pack_sz, min_pack_sz_P1, pack_sz;
int fsec, fusec, csec, cusec;
int exp_train_id = 0;
int adr_narrow, bad_train, bad_trains;
boolean enough_data, path_overflow, abort_phase1=false;
int[] measurs_vld_P1 = new int[GlobalsRcv.NO_TRAINS_P1];
int[] measurs_vld_P2 = new int[GlobalsRcv.NO_TRAINS_P2];
char[] random_data = new char[Common.MAX_PACK_SZ];
char[] pack_buf = new char[Common.MAX_PACK_SZ];
double bw_msr, bin_wd, bw_range, avg_bw, rtt, adr, std_dev, delta;
double[] min_OSdelta = new double[500];
double[] ord_min_OSdelta = new double[500];
double min_possible_delta, mode_value, merit, max_merit;
double[] measurs_P1 = new double[GlobalsRcv.NO_TRAINS_P1];
double[] measurs_P2= new double[GlobalsRcv.NO_TRAINS_P2];
double[] ord_measurs_P1 = new double[GlobalsRcv.NO_TRAINS_P1];
double[] ord_measurs_P2 = new double[GlobalsRcv.NO_TRAINS_P2];
double[] modes_P1_mode_value = new double[GlobalsRcv.MAX_NO_MODES];
long[] modes_P1_mode_cnt = new long[GlobalsRcv.MAX_NO_MODES];
long[] modes_P1_bell_cnt = new long[GlobalsRcv.MAX_NO_MODES];
double[] modes_P1_bell_lo = new double[GlobalsRcv.MAX_NO_MODES];
double[] modes_P1_bell_hi = new double[GlobalsRcv.MAX_NO_MODES];
double[] modes_P1_bell_kurtosis = new double[GlobalsRcv.MAX_NO_MODES];
double[] modes_P2_mode_value = new double[GlobalsRcv.MAX_NO_MODES];
long[] modes_P2_mode_cnt = new long[GlobalsRcv.MAX_NO_MODES];
long[] modes_P2_bell_cnt = new long[GlobalsRcv.MAX_NO_MODES];
double[] modes_P2_bell_lo = new double[GlobalsRcv.MAX_NO_MODES];
double[] modes_P2_bell_hi = new double[GlobalsRcv.MAX_NO_MODES];
double[] modes_P2_bell_kurtosis = new double[GlobalsRcv.MAX_NO_MODES];
int[] arrv_tvsec = new int[Common.MAX_STREAM_LEN];
int[] arrv_tvusec = new int[Common.MAX_STREAM_LEN];
long starts, startus, stops, stopus, timeUS;
no_pack_sizes = GlobalsRcv.NO_PACK_SIZES;
for(i = 0; i < Common.MAX_STREAM_LEN; i++){
        arrv_tvsec[i] = 0; arrv_tvusec[i] = 0;}
GetTime getTime = new GetTime();
getTime.getTimeOfDay();
starts = getTime.seconds;startus = getTime.useconds;
System.out.println("Starting the receiver");
try{
```

```
GlobalsRcv.sc = SocketChannel.open();
Socket socket = GlobalsRcv.sc.socket( );
socket.setReuseAddress(true);
GlobalsRcv.sc.configureBlocking(true);
GlobalsRcv.sc.connect(socketAddress);
ConnectionMethodsRcv cmrcv = new ConnectionMethodsRcv(GlobalsRcv.sc);
GlobalsRcv.sock_udp = udpr.getUDPSocket(Common.UDPRCV_PORT, Common.UDP_BUFFER_SZ);
GlobalsRcv.recvLatency = cmrcv.getRecvLatency();
GlobalsRcv.sendLatency = cmrcv.recv_ctr_msg();
GlobalsRcv.min_time_interval = GlobalsRcv.SCALE_FACTOR*((GlobalsRcv.recvLatency > GlobalsRcv.sendLatency) ? GlobalsRcv.recvLatency : GlobalsRcv.sendLatency) ;
GlobalsRcv.min_time_interval = GlobalsRcv.min_time_interval > GlobalsRcv.MIN_TIME_INTERVAL?GlobalsRcv.min_time_interval : GlobalsRcv.MIN_TIME_INTERVAL;
// Estimate round-trip time
sum _rtt=0;
ctr_code = Common.BOUNCE | Common.CTR_CODE;
for(i=0; i<10; i++){
        getTime.getTimeOfDay();
        fsec = GetTime.seconds;fusec = GetTime.useconds;
        cmrcv.send_ctr_msg(ctr_code);
        ctr_code = cmrcv.recv_ctr_msg();
        getTime.getTimeOfDay();
        csec = GetTime.seconds;cusec = GetTime.useconds;
        if (i>0) sum_rtt += UtilMethods.timeToUsdelta(fsec, fusec, csec, cusec);
}
rtt = (double) (sum_rtt/9000.);
System.out.println("Average round-trip time:" + rtt);
// Determine minimum train spacing based on rtt
min_train_spacing = Common.MIN_TRAIN_SPACING/1000; /* make msec */
if (min_train_spacing < rtt*1.25){
/* if the rtt is not much smaller than the specified train spacing,
* increase minimum train spacing based on rtt */
        min_train_spacing = rtt*1.25;}
max_train_spacing = min_train_spacing;train_spacing = min_train_spacing;
ctr_code = Common.TRAIN_SPACING | ((int)train_spacing<<16);
cmrcv.send_ctr_msg(ctr_code);//send minimum train spacing
max_pack_sz = Common.MAX_PACK_SZ;
ctr_code = Common.MAX_PCK_LEN | (max_pack_sz<<16);
cmrcv.send_ctr_msg(ctr_code);//send the maximum packet size
pack_sz = Common.MAX_PACK_SZ;
ctr_code = Common.PCK_LEN | (pack_sz<<16);
cmrcv.send_ctr_msg(ctr_code);//send the the packet size
/* Also set the minimum packet size for Phase I */
min_pack_sz_P1 = GlobalsRcv.MIN_PACK_SZ_P1;
```

```
if (min_pack_sz_P1 > max_pack_sz) min_pack_sz_P1 = max_pack_sz;
// Measure minimum latency for kernel-to-user transfer of packet.
int count = 0;

/* Send train length to sender */
train_len = 0; exp_train_id = 0;
ctr_code = Common.TRAIN_LEN | (train_len<<16);
cmrcv.send_ctr_msg(ctr_code);
min_possible_delta = cmrcv.getDelta();
min_possible_delta *= GlobalsRcv.MULTIPLICATIVE_FACTOR;
System.out.println("Minimum acceptable packet pair dispersion: " +
min_possible_delta + " usec (min_possible_delta)");
/* The default initial train-length in Phase I is 2 packets (packet pairs) */
train_len_P1 = 2;
/* Keep a unique identifier for each round in pathrate's execution.
This id is used to ignore between trains of previous rounds.*/
round = 1;
/* Discover maximum feasible train length in the path
(stop at a length that causes 3 lossy trains) */
System.out.println("Maximum train length discovery");
/* Send number of trains to sender -> only one train at this phase */
no_trains = 1;
ctr_code = Common.NO_TRAINS | (no_trains<<16);
cmrcv.send_ctr_msg(ctr_code);
/* Send packet size to sender -> maximum at this phase */
pack_sz = max_pack_sz;//maximum packet size at this phase phase = 0
ctr_code = Common.PCK_LEN | (pack_sz<<16);
cmrcv.send_ctr_msg(ctr_code);
/* Initial train length */
train_len = train_len_P1;path_overflow = false;
bad_trains = 0; trains_msrd = 0;
/* Repeat for each train length */
while (train_len <= GlobalsRcv.MAX_TRAIN_LEN && !path_overflow){
        /* Send train length to sender */
        ctr_code = Common.TRAIN_LEN | (train_len<<16);
        cmrcv.send_ctr_msg(ctr_code);
        /* Tell sender to start sending packet trains */
        round++;
        ctr_code = Common.SEND | Common.CTR_CODE;
        cmrcv.send_ctr_msg(ctr_code);
bad_train = cmrcv.recv_train(exp_train_id, arrv_tvsec, arrv_tvusec, train_len);
        /* Compute dispersion and bandwidth measurement */
        if (bad_train == 2)   path_overflow = true;
        else if (bad_train != 1) {
```

```
delta = UtilMethods.timeToUsdelta(arrv_tvsec[1], arrv_tvusec[1], arrv_tvsec[train_len], arrv_tvusec[train_len]);
bw_msr = ((28 + pack_sz) << 3) * (train_len-1) / delta;
/* Very slow link; the packet trains take more than their spacing to be transmitted */
                                                    if (delta > train_spacing*1000){
                                                            path_overflow=true;
                                    /* Send control message to increase train spacing by one second */
                                            ctr_code = Common.PATH_OVERFLOW | Common.CTR_CODE;
                                                            cmrcv.send_ctr_msg(ctr_code);
                                                            max_train_spacing = train_spacing;
                                            ctr_code = Common.TRAIN_SPACING | ((int)train_spacing<<16);
                                                            cmrcv.send_ctr_msg(ctr_code);
                                                    }
                                                    /* Increase train length */
                                                    prev_train_len = train_len;
                                                    if (train_len<6) train_len ++;
                                                    else if (train_len<12)train_len += 2;
                                                            else train_len += 4;
                                                }
                                    }
                                    if (path_overflow || train_len > GlobalsRcv.MAX_TRAIN_LEN) max_train_len =
prev_train_len;
                                    else max_train_len = train_len;
                                    System.out.println("Maximum train length " + max_train_len + "packets");
                                    /* Tell sender to continue with  next phase phase 1*/
                                    ctr_code = Common.CONTINUE | Common.CTR_CODE;
                                    cmrcv.send_ctr_msg(ctr_code);
                                    /*Check for possible multichannel links and traffic shapers*/
                                    System.out.println("Preliminary measurements with increasing packet train lengths");
                                    /* Send number of trains to sender (fiarrv_tvsecve of them is good enough..) */
                                    no_trains = 7;
                                    ctr_code = Common.NO_TRAINS | (no_trains<<16);
                                    cmrcv.send_ctr_msg(ctr_code);
                                    /* Send packet size to sender -> maximum at this phase */
                                    pack_sz = max_pack_sz;
                                    ctr_code = Common.PCK_LEN | (pack_sz<<16);
                                    cmrcv.send_ctr_msg(ctr_code);
                            for (train_len = train_len_P1; train_len <= Common.MIN_V(max_train_len,10); train_len++){
                                            /* Send train length to sender */
                                            if (Common.Verbose) System.out.println("Train length: " + train_len);
                                            ctr_code = Common.TRAIN_LEN | (train_len<<16);
                                            cmrcv.send_ctr_msg(ctr_code);
                                            /* Tell sender to start sending packet trains */
                                            round++;
                                            trains_rcvd=0;
```

```
                    do{
                        /* Wait for a complete packet train */
                        ctr_code = Common.SEND | Common.CTR_CODE;
                        cmrcv.send_ctr_msg(ctr_code);
                bad_train = cmrcv.recv_train(exp_train_id, arrv_tvsec, arrv_tvusec, train_len);
                        /* Compute dispersion and bandwidth measurement */
                        if (bad_train !=1){
        delta = UtilMethods.timeToUsdelta(arrv_tvsec[1], arrv_tvusec[1], arrv_tvsec[train_len], arrv_tvusec[train_len]);
        bw_msr = ((28+pack_sz) << 3) * (train_len-1) / delta;
                if (Common.Verbose) System.out.println("Bandwidth " + bw_msr + "Mbps");
                            /* Acceptable measurement */
                            if (delta > min_possible_delta * (train_len - 1)){
                            measurs_P1[trains_msrd++] = bw_msr;}
                            trains_rcvd++;
                        }
                    }while(trains_rcvd<no_trains);
                    /* Tell sender to continue with next phase */
                    ctr_code = Common.CONTINUE | Common.CTR_CODE;
                    cmrcv.send_ctr_msg(ctr_code);
                }
                trains_msrd--;
                if (trains_msrd < GlobalsRcv.MAX_MEASUR_GIGA)
                    /* Possible interrupt coalescing, try different test */
                    urcv.gig_path(max_pack_sz, round++, min_possible_delta/3);
                /*Estimate bandwidth resolution (bin width) and check for "quick estimate"*/
/* Calculate average and standard deviation of last measurements ,ignoring the five largest and the five smallest values  */
                ord_measurs_P1 =  UtilMethodsRcv.orderInt(measurs_P1, trains_msrd);
                enough_data=true;
                if (trains_msrd > 30){
        avg_bw = UtilMethodsRcv.getAvg(UtilMethodsRcv.extractLast(ord_measurs_P1,5,trains_msrd), trains_msrd-10);
        std_dev = UtilMethodsRcv.getStd(UtilMethodsRcv.extractLast(ord_measurs_P1,5,trains_msrd), trains_msrd-10);
                    outlier_lim=(int) (trains_msrd/10);
            bw_range=ord_measurs_P1[trains_msrd-outlier_lim-1]-ord_measurs_P1[outlier_lim];
                }
                else{
                    avg_bw = UtilMethodsRcv.getAvg(ord_measurs_P1, trains_msrd);
                    std_dev = UtilMethodsRcv.getStd(ord_measurs_P1, trains_msrd);
                    bw_range=ord_measurs_P1[trains_msrd-1]-ord_measurs_P1[0];
                }
                /* Estimate bin width based on previous measurements */
                /* Weiling: bin_wd is increased to be twice of earlier value */
                if (bw_range < 1.0) bin_wd = bw_range*0.25;
                else bin_wd = bw_range*0.125;
                if (bin_wd == 0) bin_wd = 20.0;
```

```
                    if (Common.Verbose) System.out.println("Capacity Resolution: " + bin_wd);
                    /* Check for quick estimate (when measurements are not very spread) */
                    if( (std_dev/avg_bw < GlobalsRcv.COEF_VAR_THR) || quick_term){
                            if (Common.Verbose) if (quick_term)
                                                    System.out.println("Requested Quick Termination");
                    else System.out.println("Quick Termination' - Sufficiently low measurement noise");
            if (Common.Verbose) System.out.println("Coefficient of variation:" + std_dev/avg_bw);
                            urcv.happyEnd(avg_bw-bin_wd/2., avg_bw+bin_wd/2.);
                            urcv.termint(0);
                    }
                    ctr_code = Common.CONTINUE | Common.CTR_CODE;
                    cmrcv.send_ctr_msg(ctr_code);
                    /*Phase I: Discover local modes with packet pair experiments*/
                    System.out.println("Phase I: Detect possible capacity modes");
            // Phase I uses packet pairs, i.e., 2 packets (the default; it may be larger)
                    train_len = train_len_P1;
                    if (train_len > max_train_len) train_len = max_train_len;
                    /* Compute number of different packet sizes  in Phase I */
                    pack_incr_step = (max_pack_sz - min_pack_sz_P1) / no_pack_sizes + 1;
                    pack_sz = min_pack_sz_P1;
                    /* Compute number of trains  per packet size */
                    no_trains_per_size = GlobalsRcv.NO_TRAINS_P1 / no_pack_sizes;
                    /* Number of trains in Phase I */
                    no_trains = GlobalsRcv.NO_TRAINS_P1;
                    /* Send train spacing to sender */
                    train_spacing = min_train_spacing;
                    ctr_code = Common.TRAIN_SPACING | ((int)train_spacing<<16);
                    cmrcv.send_ctr_msg(ctr_code);
                    trains_msrd = 0;
                    /* Compute new packet size (and repeat for several packet sizes) */
                    for (i = 0; i < no_pack_sizes; i++){
                            ctr_code = Common.PCK_LEN | (pack_sz<<16);
                            cmrcv.send_ctr_msg(ctr_code);
                            ctr_code = Common.NO_TRAINS | (no_trains_per_size<<16);
                            cmrcv.send_ctr_msg(ctr_code);
                            ctr_code = Common.TRAIN_LEN | (train_len<<16);
                            cmrcv.send_ctr_msg(ctr_code);
                            /* Tell sender to start sending */
                            round++;
if (Common.Verbose) System.out.println("Train length: " + train_len + "/ Packet size: " + (pack_sz+28) + "/ " + i*100/no_pack_sizes + "% completed");
                            bad_tstamps = 0; // reset bad time stamps counter
                            trains_per_size = 0;
                            do{
```

```
                                        ctr_code = Common.SEND | Common.CTR_CODE;
                                        cmrcv.send_ctr_msg(ctr_code);
                        bad_train = cmrcv.recv_train(exp_train_id, arrv_tvsec, arrv_tvusec, train_len);
                                        /* Compute dispersion and bandwidth measurement */
                                        if (bad_train != 1) {
        delta = UtilMethods.timeToUsdelta(arrv_tvsec[1], arrv_tvusec[1], arrv_tvsec[train_len], arrv_tvusec[train_len]);
        bw_msr = ((28+pack_sz) << 3) * (train_len-1) / delta;
                                                if (Common.Verbose) {
                                System.out.println("Measurement " + trains_per_size+1);
                                System.out.println("Bandwidth " + bw_msr + "Mbps");}
                                                /* Acceptable measurement */
                                                if (delta > min_possible_delta*(train_len-1)) {
                                                        measu rs_P1[trains_msrd++] = bw_msr;
                                                else{bad_tstamps++;}
                                                /* # of trains received in this packet size iteration */
                                                trains_per_size++;
                                        }
                                } while(trains_per_size < no_trains_per_size);
                                /* Tell sender to continue with next phase */
                                pack_sz += pack_incr_step;
                                if (pack_sz > max_pack_sz)
                                        pack_sz = max_pack_sz;
                                /***** dealing with ignored measurements *****/
                        if (bad_tstamps > no_trains_per_size/GlobalsRcv.IGNORE_LIM_FACTOR)
                                {
                                        train_len+=1;
                                        if (train_len > Common.MAX_V(max_train_len/4,2)){
                                                abort_phase1=true;}
                                        pack_sz+=275;
                                        if (pack_sz > max_pack_sz) pack_sz = max_pack_sz;
                                        if (!abort_phase1) {
        if (Common.verbose) System.out.println("Too many ignored measurements " + train_len + " packets. Adjust packet
size: " + (Common.MIN_V(pack_sz,max_pack_sz)+28));}
                                                else break;
                                }
                                ctr_code = Common.CONTINUE | Common.CTR_CODE;
                                cmrcv.send_ctr_msg(ctr_code);
                        }
                        /* Compute number of valid (non-ignored) measurements */
                        no_trains = trains_msrd-1;
                        /*Detect all local modes in Phase I*/
                        if (!abort_phase1){
                                /* Order measurements */
                                ord_measurs_P1 = UtilMethodsRcv.orderInt(measurs_P1, no_trains);
```

```
                    /* Detect and store all local modes */
                    System.out.println("Local modes : In Phase I -- ");
                    /* Mark all measurements as valid (needed for local mode detection) */
for (train_no = 0; train_no < no_trains; train_no++)  measurs_vld_P1[train_no] = 1;
                    no_modes_P1 = 0;
                    while (true){
          mode_value = urcv.getMode(ord_measurs_P1, measurs_vld_P1, bin_wd, no_trains);
                        if (mode_value == Common.LAST_MODE) break;
/*the modes are ordered based on the number of measurements in the modal bin (strongest mode
first) */
                    if (mode_value != Common.UNIMPORTANT_MODE) {
                    modes_P1_mode_value[no_modes_P1] = mode_value;
                    modes_P1_mode_cnt[no_modes_P1] = urcv.getMode_cnt();
                    modes_P1_bell_cnt[no_modes_P1] = urcv.getBell_cnt();
                    modes_P1_bell_lo[no_modes_P1] = urcv.getBell_lo();
                    modes_P1_bell_hi[no_modes_P1] = urcv.getBell_hi();
                    modes_P1_bell_kurtosis[no_modes_P1] = urcv.getBell_kurtosis();
                    no_modes_P1++;
                    if (no_modes_P1 >= GlobalsRcv.MAX_NO_MODES) {
                        System.out.println("Increase MAX_NO_MODES constant");
                        urcv.termint(-1);}
                    }
          }
}
else {
          System.out.println("Aborting Phase I measurements..");
          System.out.println("Too many ignored measurements");
          System.out.println("Phase II will report lower bound on path capacity.");
          getTime.getTimeOfDay();
          stops = getTime.seconds; stopus = getTime.useconds;
          timeUS= (stops - starts)*1000000 + (stopus - startus);
          System.out.println("Execution time " + (int)(timeUS/1000) + " miliseconds");
}
no_trains_P1 = no_trains;
/* Tell sender to continue with next phase */
ctr_code = Common.CONTINUE | Common.CTR_CODE;
cmrcv.send_ctr_msg(ctr_code);
/* Phase II: Packet trains with maximum train length*/
System.out.println("Phase II: Estimate Asymptotic Dispersion Rate (ADR) -- ");
/* Train spacing in Phase II */
train_spacing = max_train_spacing;
ctr_code = Common.TRAIN_SPACING | ((int)train_spacing<<16);
cmrcv.send_ctr_msg(ctr_code);
/* Determine train length for Phase II. Tell sender about it. */
```

```
/*The following constraint is only used in very low capacity paths.*/
train_len = max_train_len;
train_len = (int)(((avg_bw*0.5) * (max_train_spacing*1000)) / (max_pack_sz*8));
if (train_len > max_train_len) train_len = max_train_len;
if (train_len < train_len_P1) train_len = train_len_P1;
ctr_code = Common.TRAIN_LEN | (train_len<<16);
cmrcv.send_ctr_msg(ctr_code);
/* Packet size in Phase II. Tell sender about it. */
pack_sz = max_pack_sz;
ctr_code = Common.PCK_LEN | (pack_sz<<16);
cmrcv.send_ctr_msg(ctr_code);
/* Number of trains in Phase II. Tell sender about it. */
no_trains = GlobalsRcv.NO_TRAINS_P2;
ctr_code = Common.NO_TRAINS | (no_trains<<16);
cmrcv.send_ctr_msg(ctr_code);
if (Common.Verbose)System.out.println("Number of trains: " + no_trains + " - Train length: " +
train_len + " - Packet size: " + (pack_sz+28));
/* Tell sender to start sending */
round++; trains_msrd=0;trains_rcvd=0;bad_tstamps=0;
do{
        ctr_code = Common.SEND | Common.CTR_CODE;
        cmrcv.send_ctr_msg(ctr_code);
        /* Wait for a complete packet train */
        bad_train = cmrcv.recv_train(exp_train_id, arrv_tvsec, arrv_tvusec, train_len);
        /* Compute dispersion and bandwidth measurement */
        if (bad_train != 1){
delta = UtilMethods.timeToUsdelta(arrv_tvsec[1], arrv_tvusec[1], arrv_tvsec[train_len], arrv_tvusec[train_len]);
bw_msr = ((28+pack_sz) << 3) * (train_len-1) / delta;
        if (Common.Verbose) {
        System.out.println("Measurement " + (trains_rcvd + 1) + "  out of " + no_trains);
        System.out.println("Bandwidth " + bw_msr + "Mbps");}
        /* Acceptable measurement */
        if (delta > min_possible_delta*(train_len-1)){
                measurs_P2[trains_msrd++] = bw_msr;}
                else bad_tstamps++;
        trains_rcvd++;
        }
}while(trains_rcvd<no_trains);
/* Tell sender to continue with next phase */
ctr_code = Common.CONTINUE | Common.CTR_CODE;
cmrcv.send_ctr_msg(ctr_code);
/* Compute number of valid (non-ignored measurements) */
no_trains = trains_msrd - 1;
/* Detect all local modes in Phase II */
```

```
/* Order measurements */
ord_measurs_P2 = UtilMethodsRcv.orderInt(measurs_P2, no_trains);
/* Detect and store all local modes in Phase II */
        System.out.println("Local modes : In Phase II");
        /* Mark all measurements as valid (needed for local mode detection) */
        for (train_no = 0; train_no < no_trains; train_no++)  measurs_vld_P2[train_no]=1;
        no_modes_P2=0;
        while (true) {
        mode_value = urcv.getMode(ord_measurs_P2, measurs_vld_P2, bin_wd, no_trains);
        if ( mode_value == Common.LAST_MODE) break;
                /* the modes are ordered based on the number of measurements
                in the modal bin (strongest mode first) */
                if (mode_value != Common.UNIMPORTANT_MODE) {
                        modes_P2_mode_value[no_modes_P1] = mode_value;
                        modes_P2_mode_cnt[no_modes_P1] = urcv.getMode_cnt();
                        modes_P2_bell_cnt[no_modes_P1] = urcv.getBell_cnt();
                        modes_P2_bell_lo[no_modes_P1] = urcv.getBell_lo();
                        modes_P2_bell_hi[no_modes_P1] = urcv.getBell_hi();
                modes_P2_bell_kurtosis[no_modes_P1] = urcv.getBell_kurtosis();
                        no_modes_P2++;
                        if (no_modes_P2 >= GlobalsRcv.MAX_NO_MODES) {
                        System.out.println("Increase MAX_NO_MODES constant");
                        urcv.termint(-1);
                        }
                }
        }
        /*
        If the Phase II measurements are distributed in a very narrow fashion
        (i.e., low coefficient of variation, and std_dev less than bin width),
        and the ADR is not much lower than the earlier avg bandwidth estimate,
        the ADR is a good estimate of the capacity mode. This happens when
        the narrow link is of significantly lower capacity than the rest of the links,
        and it is only lightly used.*/
        /* Compute ADR estimate */
adr = UtilMethodsRcv.getAvg(UtilMethodsRcv.extractLast(ord_measurs_P2,10,no_trains), no_trains-20);
std_dev = UtilMethodsRcv.getStd(UtilMethodsRcv.extractLast(ord_measurs_P2,10,no_trains), trains_msrd -20);
        adr_narrow=0;
if (no_modes_P2==1 && std_dev/adr<GlobalsRcv.COEF_VAR_THR && adr/avg_bw>GlobalsRcv.ADR_REDCT_THR){
        if (Common.Verbose)System.out.println("The capacity estimate will be based on the ADR mode.");
        adr = modes_P2_mode_value[0];
        adr_narrow=1;
}
if (no_modes_P2 > 1) {
```

```
                System.out.println("WARNING: Phase II did not lead to unimodal distribution. The ADR estimate may be wrong. Run again later.");
                max_merit=0.;
                for(i=0;i<no_modes_P2;i++) {
                        System.out.println(modes_P2_mode_value[i]-bin_wd/2.);
                        System.out.println(modes_P2_mode_value[i]+bin_wd/2.);
                        // Weiling: merit is calculated using kurtosis as the narrowness of the bell
                        merit = modes_P2_bell_kurtosis[i] * (modes_P2_mode_cnt[i] / (double)no_trains);
                        System.out.println("Figure of merit:" +  merit);
                        if (merit > max_merit){
                                max_merit = merit;
                                cap_mode_ind = i;
                        }
                }
                adr = modes_P2_mode_value[cap_mode_ind];
        }
        System.out.println("Asymptotic Dispersion Rate (ADR) estimate:");
        System.out.println(adr);
                /* The end of the story... Final capacity estimate */
                /* Final capacity estimate: the Phase I mode that is larger than ADR,
                and it has the maximum "figure of merit". This figure of metric is the
                "density fraction" of the mode, times the "strength fraction" of the mode.
                The "density fraction" is the number of measurements in the central bin of the mode,
                divided by the number of measurements in the entire bell of that mode.
                The "strength fraction" is the number of measurements in the central bin of the mode,
                divided by the number of measurements in the entire Phase I phase.*/
                if (!abort_phase1){
                        System.out.println("Possible capacity values:");
                        max_merit=0.;
                        for(i=0;i<no_modes_P1;i++){
                                /* Give possible capacity modes */
                                if (modes_P1_mode_value[i]+bin_wd/2. > adr){
                                System.out.println(modes_P1_mode_value[i]-bin_wd/2.);
                                System.out.println(modes_P1_mode_value[i]+bin_wd/2.);
                                // Weiling: merit is calculated using kurtosis as the narrowness of the bell
                        merit = modes_P1_bell_kurtosis[i] * (modes_P1_mode_cnt[i] / (double)no_trains_P1);
                                System.out.println("Figure of merit: " + merit);
                                if (merit > max_merit){
                                        max_merit =  merit;
                                        cap_mode_ind = i;}
                        }
                }
                if (max_merit>0.){
        urcv.happyEnd(modes_P1_mode_value[cap_mode_ind]-bin_wd/2.,
```

```
modes_P1_mode_value[cap_mode_ind]+bin_wd/2.);
                getTime.getTimeOfDay();
                stops = getTime.seconds;stopus = getTime.useconds;
                                timeUS= (stops - starts)*1000000 + (stopus - startus);
                                System.out.println("Execution time " + (int)(timeUS/1000) + " miliseconds");
                                urcv.termint(0);
                        }
                        /* If there is no Phase I mode that is larger than the ADR or
                         if Phase II gave a very narrow distribution
                        the final capacity estimate is the ADR.*/
                        else{
                        /* If there are multiple modes in Phase II (not unique estimate for ADR),
                        * the best choice for the capacity mode is the largest mode of Phase II */
                                System.out.println(adr-bin_wd/2.);
                                System.out.println(adr+bin_wd/2.);
                                getTime.getTimeOfDay();
                                stops = getTime.seconds; stopus = getTime.useconds;
                                timeUS= (stops - starts)*1000000 + (stopus - startus);
                                System.out.println("Execution time " + (int)(timeUS/1000) + " miliseconds");
                                System.out.println("ADR mode");
                                urcv.happyEnd(adr-bin_wd/2., adr+bin_wd/2.);
                                urcv.termint(0);
                        }
                }
                else {
                        max_merit=0.;
        System.out.println("Phase I was aborted.--> The following estimate is a lower bound for the path capacity.");
                        getTime.getTimeOfDay();
                        stops = getTime.seconds;stopus = getTime.useconds;
                        timeUS= (stops - starts)*1000000 + (stopus - startus);
                        System.out.println("Execution time " + (int)(timeUS/1000) + " miliseconds");
                        urcv.happyEnd(adr-bin_wd/2., adr+bin_wd/2.);
                        urcv.termint(0);
                }
        } catch (Exception e){ System.err.println(e);}
         finally{ udpr.closeUDPSocket(GlobalsRcv.sock_udp);
                if (GlobalsRcv.sc != null){
                                try{GlobalsRcv.sc.close();} catch (Exception ignored){}}
                        }
                }
        }
```

PathrateSnd.java

```
        public class PathrateSnd{
```

```
	Server srv;
	public PathrateSnd(int unitsNo) {srv = new Server(unitsNo);}
	public void startServer(){
		srv.start();System.out.println("[Pathrate] ----> start server");}
	public void stopServer() { srv.stopServer();System.out.println("[Pathrate] ----> stop server");}
}
class Server extends Thread {
	Selector selector; ServerSocketChannel serverChannel;ServiceThreadPool stp;boolean boolSelect;
	public Server(int unitsNo){
		super();boolSelect = true;
		try {
			stp = new ServiceThreadPool(unitsNo);
			Globals.po = new PrintOutput("server.log");
		} catch (IOException e) {}
	}
	public void run(){ startServer();}
	public void startServer(){
		try{ selector = Selector.open();}catch (IOException e){}
		UtilMethods.minSleepTime();
		/* gettimeofday latency */
		GetTime getTime = new GetTime();
		Globals.getTimeOfDayLatency = getTime.getTimeOfDayLatency();
		Globals.po.write("DEBUG :: gettimeofday latency(usec) = " + Globals.getTimeOfDayLatency);
		UDPSend udps = new UDPSend();
		Globals.sendLatency = udps.sendLatency(Common.MAX_PACK_SZ);
		Globals.po.write("DEBUG :: send latency(usec) = " + Globals.sendLatency);
		// open channels for communication
		try { openChannel();  } catch (IOException e) {}
		serviceClients();
	}
	public void stopServer() {
		Globals.po.close();
		try {
			boolSelect = false;
			selector.wakeup();
			selector.close();
			serverChannel.close();
		} catch (IOException ex) { }
		stp.stopWorkers();
	}
	public void openChannel() throws IOException {
		int tcpSndPort = Common.TCPSND_PORT;
		// TCP control connection
		// Allocate an unbound server socket channel
```

```
            serverChannel = ServerSocketChannel.open();
            // Get the associated ServerSocket to bind it with
            ServerSocket serverSocket = serverChannel.socket();
            // Set the port the server channel will listen to
            serverSocket.bind (new InetSocketAddress (tcpSndPort));
            serverSocket.setReuseAddress(true);
            // Set nonblocking mode for the listening socket
            serverChannel.configureBlocking (false);
            // Register the ServerSocketChannel with the Selector
            serverChannel.register (selector, SelectionKey.OP_ACCEPT);
}
public void serviceClients() {
            try {
                        // Wait for something of interest to happen
                        while (boolSelect) {
                                    // This may block for a long time. Upon returning, the
                                    // selected set contains keys of the ready channels.
                                    int n = selector.select();
                                    if (n == 0) {continue;     // nothing to do   }
                                    // Get set of ready objects
                                    Set readyKeys = selector.selectedKeys();
                                    Iterator readyItor = readyKeys.iterator();

                                    // Walk through set
                                    while (readyItor.hasNext()) {
                                                // Get key from set
                                                SelectionKey key = (SelectionKey)readyItor.next();
                                                // Remove current entry
                                                readyItor.remove();
                                                if (key.isAcceptable()) {
                                                            // Get channel
                                    ServerSocketChannel keyChannel = (ServerSocketChannel)key.channel();
                                                            // Get the socket channel
                                                            SocketChannel s = keyChannel.accept();
                                                            s.configureBlocking(false);
                                                            // ok now put the task in someone's queue
                                    // each worker has a selector registered on a number of socket channels
                                                            stp.addNewTask(s);
                                                }
                                                else {throw new IllegalStateException();}
                                    }
                        }
            } catch (IOException e) {Globals.po.close();
                        try { selector.close();  serverChannel.close();} catch (IOException ex) { }}}}
```