"Politehnica" University of Bucharest
Computer Science and Engineering Department
Faculty of Computer Science

Measuring the available bandwidth in a network (end to end). Integration in MonALISA

Student name:
Madalin Mihailescu
Coordinator:
PhD-Professor Ing. Valentin Cristea (UPB)
Adviser:
PhD-Professor Iosif Charles Legrand (Caltech)

Bucharest
- 2004 -

Preface

Abstract

The available bandwidth in a network path is of major importance in congestion control, streaming applications, quality of service verification, server selection and in several other areas. The ability for an application to adapt its behavior to changing network conditions depends on the underlying bandwidth estimation mechanism that the application or transport protocol uses. So, from accurate bandwidth estimation algorithms and tools, can benefit a large class of data-intensive and distributed scientific applications.

The accuracy in measuring the available bandwidth is difficult to achieve if estimation techniques that use small amount of data, instead of large data volume techniques, are used but there is current research in this area. Existing techniques attempt to estimate the capacity and bandwidth of both links and paths using 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.

The MonALISA (Monitoring Agents in A Large Integrated Services Architecture) system provides a distributed service architecture which is used to collect and process monitoring information. Besides the monitoring service, MonALISA provides a videoconferencing system.

VRVS is a videoconferencing system based on a set of servers called reflectors that route the audio/video streams to the participating clients. A goal for the VRVS system is to enhance the quality of the service, so the quality of the alternative connections in the system is important. The available bandwidth is the most significant parameter that describes the quality of a connection.

This paper presents a method for estimating the available bandwidth end to end, its implementation and the integration in MonALISA

Table of contents

Chapter 1 - Introduction

1.1. The importance of measuring network links

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 inability for users to be confident in the performance in the network is causing a demand for tools to enable users to asses the performance without assistance. These tools need to quickly and easily measure the end-to-end performance of the network, while not placing anymore load on the network than is absolutely necessary.

Extra load may restrict the times that the measurement may be made, and depending on the charging system, could create large extra traffic charges.

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 users 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.

The need for accurate bandwidth measurement is stronger for scientific applications in nuclear physics, astronomy or biotechnology.

Scientific experiments in the fields of high energy nuclear/particle physics, astronomy, or biotechnology, can generate terabytes of data; experiments planned for the near future are expected to produce petabytes. Such large data sets will be made available to thousands of scientists around the world via a high performance computing and communication infrastructure. Scientists must be able to analyze results of these experiments as if these petabytes were stored in their desktop hard drive, to collaborate in `labs without walls', and to access remote instruments as if these instruments were located next door. Research projects such as the Grid Physics Network (GriPhyN) and the Particle Physics Data Grid (PPDG) are motivated by the computing and networking needs of this new generation of scientific experiments.

Expensive links and switches graded as `terabyte-capable' do not guarantee that applications will be able to use all this potential bandwidth. It has been demonstrated that, especially in high latency and high bandwidth paths, data intensive applications often achieve throughput of no more than tens of Mbps, regardless of network capacity graded orders of magnitude higher.

Several factors contribute to mismatches between the capacity of underlying network technology and application throughput. The latter depends on the transport protocol design (flow and congestion control algorithms), transport protocol implementation and selection of its tunable parameters, intensity and behavior of network cross traffic, throughput of network interfaces and operating system at end-hosts, and the capacities and MTUs (Maximum Transmission Units) of underlying link technologies.

Capacity, bandwidth, and throughput all quantify aspects of data transfer rate across a network, which is the main performance consideration for data intensive scientific applications. *A major component missing from today's applications and transport protocols is the ability to measure and predict the throughput achievable across a network path.* Instead, transport protocols, mainly TCP, attempt to dynamically and adaptively search for the maximum possible rate using techniques such as slow start or congestion avoidance, which often lead to network underutilization and low application throughput. We could avoid these problems if we had measurement methodologies that could accurately monitor the maximum possible bandwidth (*capacity*) in a network path, as well as the maximum allowable bandwidth (*available bandwidth*). With accurate bandwidth-related information, the transport protocol and applications could achieve higher throughput and react faster to changing network conditions. Accurate bandwidth assessment techniques would also allow a quantum jump in the functionality of Internet traffic engineering, e.g., agents in routers or proxies that could inform routing protocols of overloaded paths, and/or provide guidance to service differentiation mechanisms (scheduling, input traffic limiting) for overloaded classes of traffic.

These problems can be avoided by finding accurate monitoring methods based on UDP or more optimal on ICMP packets. These two last protocols adapt faster and better to network changes and they don’t need keeping a connection opened during the measurement process (with a few thousands connections you can get to congestion).

1.2. History of measuring tools

Numerous network measurement projects in the last decade, mostly focusing on performance evaluation, have defined two generations of network performance measurement activities. We now have a third generation. Each generation builds on the measurement tools and experiences of the previous ones.

1. **First generation:** In the eighties and early nineties, several active performance measurement tools were developed, e.g., *ping*, *traceroute*, *ttcp*, primarily for use by network managers when installing equipment, debugging the network, or monitoring basic performance metrics, such as round-trip times, loss rates (with periodic packet probing), number of hops, or bulk TCP throughput. These tools have

demonstrated enormous practical operational value for network managers, but they do not measure more advanced performance metrics, such as capacity or available bandwidth across a path or of a certain link in the path.

2. **Second generation:** Starting in the mid-nineties, several large-scale measurement activities used first-generation tools (and slightly modified derivatives) to evaluate and monitor the performance of various infrastructures: *PingER* (SLAC), *NIMI* (LBNL, PSC, ACIRI), *Surveyor* (Advanced Networks), RIPE's *Test Traffic*, *AMP* (NLANR), Wolski's *NWS*, *Skitter* (CAIDA), and several commercial efforts, e.g., Keynote, MIDS, Internet Weather Services. A common characteristic of these projects is the large number of probing hosts distributed around the US or world. An important point in these projects is that N probes allow bi-directional monitoring of N^2 network paths. Results from these projects have been of solid (yet limited) value in *monitoring* and *visualizing* approximate network performance. Their results are, however, virtually impossible to correlate, and they do not do well at capturing and analyzing performance data in time to take immediate operational actions based on the measurements. Further, none of the activities involved mechanisms for making measurement results available to transport protocols, applications, or middleware to improve end-to-end performance.

3. **Third generation:** To make an innovative contribution to the field, the next generation of measurement technology captures more advanced performance metrics, such as capacity (maximum possible throughput) and available bandwidth (maximum allowable throughput) of a network path. Such metrics are more useful to an application than simple round-trip time or loss rate estimates. Other advanced performance metrics related to available bandwidth include: delay variations (jitter), stationarity of cross traffic, and distribution of queue sizes along the network path. In addition, third generation tools must make gathered information available to applications, transport protocols, and network middleware for use in modulating end host behavior to optimize end-to-end performance. I will cover the existing techniques and tools in chapter 3.

Chapter 2 - Basic notions for measuring network links

2.1. Definitions

For a better understanding of the essence of the problems and the solutions proposed, it's necessary to define and explain some specific terms:

Hosts → end points from which a packet either originates from or is destined to.

Router → 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.

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

Link → refers to a single connection between routers or routers and hosts.

Path → 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 → 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.

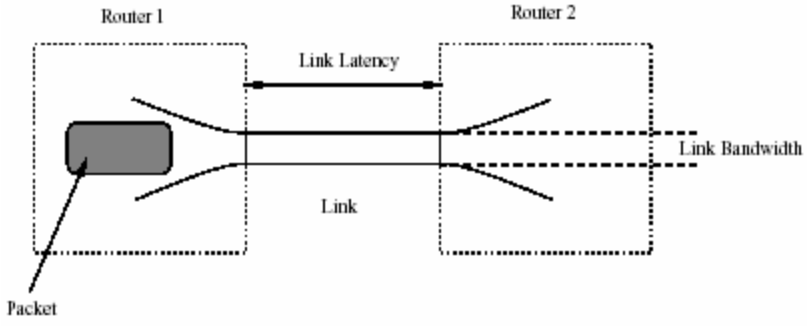

Figure 1. Bandwidth and latency

Link bandwidth → 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 (**Figure 1**).

The transmission delay → 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 → 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 → the sum of all of the individual transmission times as well as the queueing 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 .

Round trip time (RTT) latency → 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.

The path bandwidth → 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 → the amount of bandwidth ``left over" after the cross traffic. The link with the lowest available bandwidth will not necessary be the link with the lowest capacity.

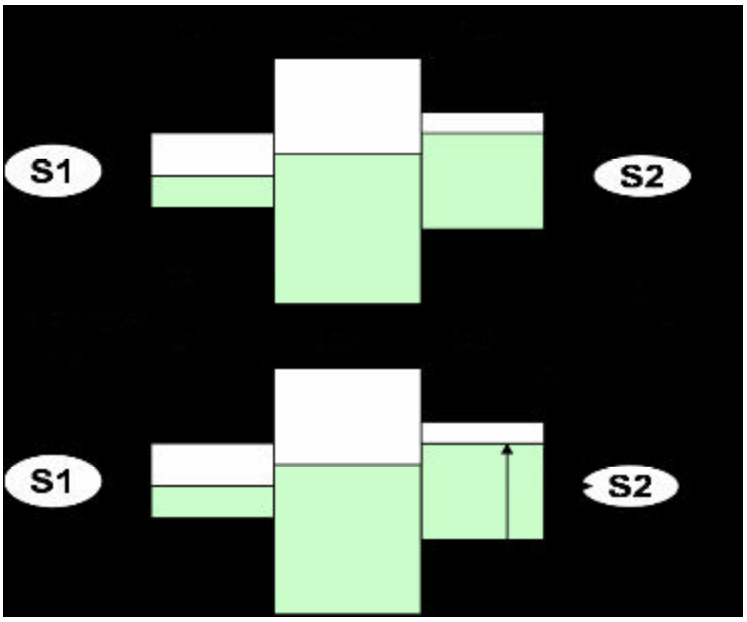

Figure 2. Capacity and available bandwidth

The *capacity* of a path is determined by the link with the minimum capacity (narrow link) → C = C1.

The *available bandwidth* of a path is determined by the link with *the minimum unused capacity* (tight link). → A. (**Figure 2**)

Chapter 3 - Existing techniques and tools

3.1. Existing bandwidth estimation techniques

This section describes existing bandwidth measurement techniques for estimating capacity and available bandwidth in individual hops and end-to-end paths. We focus on four major techniques: variable packet size (VPS) probing, packet pair/train dispersion (PPTD), self-loading periodic streams (SLoPS), and trains of packet pairs (TOPP). VPS estimates the capacity of individual hops, PPTD estimates end-to-end capacity, and SLoPS and TOPP estimate end-to-end available bandwidth. There is no currently known technique to measure available bandwidth of individual hops.

In the following we assume that during the measurement of a path P its route remains the same and its traffic load is stationary. Dynamic changes in routing or load can create errors in any measurement methodology. Unfortunately, most currently available tools do not check for dynamic route or load changes during the measurement process.

3.1.1. Variable Packet Size Probing

VPS probing aims to measure the capacity of each hop along a path. S. Bellovin and V. Jacobson were the first to propose and explore the VPS methodology. The key element of the technique is to measure the RTT from the source to each hop of the path as a function of the probing packet size. VPS uses the time-to-live (TTL) field of the IP header to force probing packets to expire at a particular hop. The router at that hop discards the probing packets, returning ICMP time-exceeded error messages back to the source. The source uses the received ICMP packets to measure the RTT to that hop. The RTT to each hop consists of three delay components in the forward and reverse paths: serialization delays, propagation delays, and queuing delays.

The serialization delay of a packet of size L at a link of transmission rate C is the time to transmit the packet on the link, equal to L/C. The propagation delay of a packet at a link is the time it takes for each bit of the packet to traverse the link, and is independent

of the packet size. Finally, queuing delays can occur in the buffers of routers or switches when there is contention at the input or output ports of these devices.

VPS sends multiple probing packets of a given size from the sending host to each layer 3 device along the path. The technique assumes that at least one of these packets, together with the ICMP reply it generates, will not encounter any queuing delays. Therefore, the minimum RTT measured for each packet size will consist of two terms: a delay that is independent of packet size and mostly due to propagation delays, and a term proportional to the packet size due to serialization delays at each link along the packet's path. Specifically, the minimum RTT $T_i(L)$ for a given packet size L up to hop i is expected to be:

$$T_i(L) = \alpha + \sum_{k=1}^{i} \frac{L}{C_k} = \alpha + \beta_i L,$$

where:

• C_k : capacity k-th hop

• a : delays up to hop i that do not depend on the probing packet size L

• ß$_i$: slope of minimum RTT up to hop i against probing packet size L, given by

$$\beta_i = \sum_{k=1}^{i} \frac{I}{C_k}$$

Note that all ICMP replies have the same size, independent of L; thus, the a term includes their serialization delay along with the sum of all propagation delays in the forward and reverse paths.

The minimum RTT measurements for each packet size up to hop i estimates the term ß$_i$. Repeating the minimum RTT measurement for each hop i = 1, …, H, the capacity estimate at each hop i along the forward path is:

$$C_i = \frac{1}{\beta_i - \beta_{i-1}}$$

Unfortunately, VPS probing may yield significant capacity underestimation errors if the measured path includes store-and-forward layer 2 switches. Such devices introduce serialization delays of the L/C type, but they do not generate ICMP TTL-expired replies because they are not visible at the IP layer. Modifying VPS probing to avoid such errors remains an active research problem.

3.1.2 Packet Pair/Train Dispersion Probing

3.1.2.1. Packet Pair Probing

Packet pair probing is used to measure the end-to-end capacity of a path. The source sends multiple *packet pairs* to the receiver. Each packet pair consists of two packets of the same size sent back to back. The dispersion of a packet pair at a specific link of the path is the time distance between the last bit of each packet. Packet pair techniques originate from seminal work by V. Jacobson, S. Keshav, and J. C. Bolot.

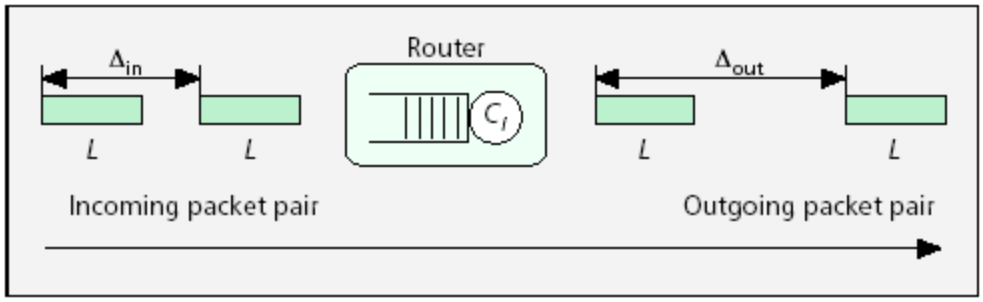

Figure 3. Packet pair dis pertion

Figure 3 shows the dispersion of a packet pair before and after the packet pair goes through a link of capacity C_i assuming that the link does not carry other traffic. If a link of capacity C_0 connects the source to the path and the probing packets are of size L, the dispersion of the packet pair at that first link is $?_0 = L/C_0$. In general, if the dispersion prior to a link of capacity C_i is $?_{in}$, the dispersion after the link will be:

$$\Delta_{out} = \max\left(\Delta_{in}, \frac{L}{C_i}\right)$$

, assuming again that there is no other traffic on that link.

After a packet pair goes through each link along an otherwise empty path, the dispersion R the receiver will measure is:

$$\Delta_R = \max_{i=0,\ldots,H}\left(\frac{L}{C_i}\right) = \frac{L}{\min_{i=0,\ldots,H}(C_i)} = \frac{L}{C}$$

, where C is the end-to-end capacity of the path. Thus, the receiver can estimate the path capacity from $C = L/?_R$. Admittedly, the assumption that the path is empty of any other traffic (referred to here as *cross traffic*) is far from realistic. Even worse, cross traffic can either increase or decrease the dispersion $?_R$, causing underestimation or overestimation, respectively, of the path capacity. Capacity underestimation occurs if

cross traffic packets are transmitted between the probing packet pair at a specific link, increasing the dispersion to more than L/C. Capacity overestimation occurs if cross traffic delays the first probe packet of a packet pair more than the second packet at a link that follows the path's narrow link.

Sending many packet pairs and using statistical methods to filter out erroneous bandwidth measurements mitigates the effects of cross traffic. Unfortunately, standard statistical approaches such as estimating the median or the mode of the packet pair measurements do not always lead to correct estimation. **Figure 4** illustrates why, showing 1000 packet pair measurements at a path from the University of Wisconsin to CAIDA (at the University of California, San Diego, UCSD), for which the path capacity is 100 Mb/s. Note that most of the measurements underestimate the capacity, while the correct measurements form only a local mode in the histogram. Identifying the correct capacity-related mode is a challenging task.

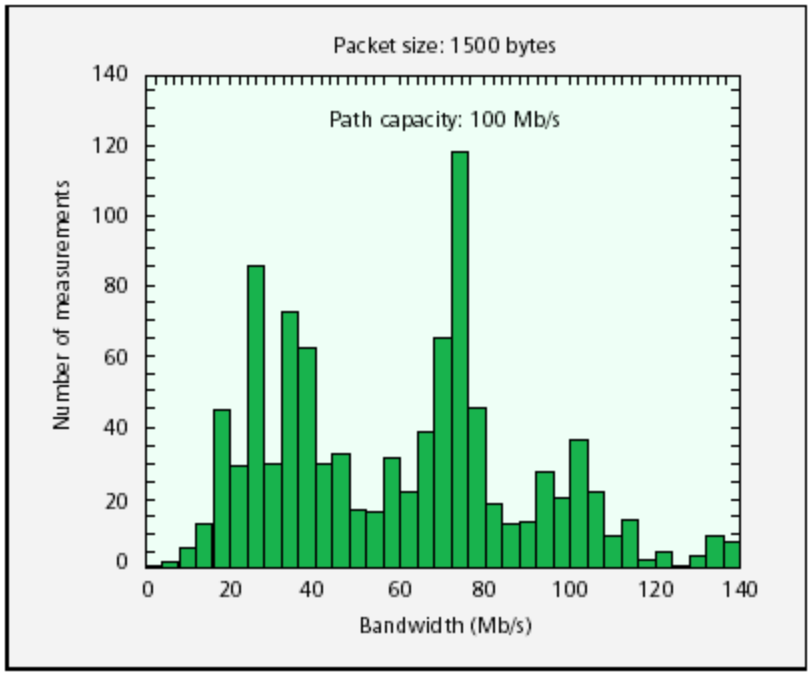

Figure 4. A histogram of capacity measurements from 1000 packet pair experiments in a 100 Mb/s path

3.1.2.2. Packet Train Probing

Packet train probing extends packet pair probing by using multiple back-to-back packets. The dispersion of a packet train at a link is the amount of time between the last

bit of the first and last packets. After the receiver measures the end-to-end dispersion $?_R(N)$ for a packet train of length N, it calculates a *dispersion rate D* as:

$$D = \frac{(N-1)L}{\Delta_R(N)}$$

What is the physical meaning of this dispersion rate? If the path has no cross traffic, the dispersion rate will be equal to the path capacity, the same as with packet pair probing. However, cross traffic can render the dispersion rate significantly lower than the capacity. To illustrate this effect, consider the case of a two-hop path. The source sends packet trains of length N through an otherwise empty link of capacity C_0. The probing packets have a size of L bytes. The second link has a capacity $C_1 < C_0$, and carries cross traffic at an average rate of $R_c < C_1$. We assume that the links use first come first served (FCFS) buffers. The dispersion of the packet train after the first link is $?_1 = L(N-1)/C_0$, while the train dispersion after the second link is:

$$\Delta_2 = \frac{(N-1)L + X_c}{C_1}$$

where X_c is the amount of cross traffic (in bytes) that will arrive at the second link during the arrival of the packet train at that link. The expected value of X_c is:

$$E[X_c] = R_c\Delta_1 = R_c\frac{(N-1)L}{C_0}$$

, so the average dispersion rate ADR the receiver measures is:

$$ADR \triangleq \frac{(N-1)L}{E[\Delta_2]} = \frac{C_1}{1+\frac{R_c}{C_0}} \leq C_1$$

As the train length N increases, the variance in the amount of cross traffic X_c that interferes with the probing packet train decreases, reducing also the variance of the dispersion rate *D*.

This last equation shows the following important properties for the average dispersion rate ADR. First, if $R_c > 0$, ADR is less than the path capacity. Second, ADR *is not related to the available bandwidth in the path*, which is $A = C1 - Rc$ in this example. In fact, it is easy to show that ADR is larger than the available bandwidth (ADR > A) if $Rc > 0$. Finally, ADR *is independent of the packet train length* N. However, N affects the variance of the measured dispersion rate D around its mean ADR, with longer packet

trains (larger *N*) reducing the variance in *D*.

PPTD probing techniques typically require double-ended measurements, with measurement software running at both the source and the sink of the path. It is also possible to perform PPTD measurements without access at the sink, by forcing the receiver to send some form of error message (e.g., ICMP port-unreachable or TCP RST packets) in response to each probe packet. In this case the reverse path capacities and cross traffic may affect the results.

3.1.3. Self-Loading Periodic Streams (SLoPS)

SLoPS is a recent measurement methodology for measuring end-to-end available bandwidth. The source sends a number *K* of equal-sized packets (a *periodic packet stream*) to the receiver at a certain rate *R*. The methodology involves monitoring variations in the one-way delays of the probing packets. If the stream rate *R* is greater than the path's available bandwidth *A*, the stream will cause a short-term overload in the queue of the tight link. One-way delays of the probing packets will keep increasing as each packet of the stream queues up at the tight link. On the other hand, if the stream rate *R* is lower than the available bandwidth *A*, the probing packets will go through the path without causing increasing backlog at the tight link, and their one-way delays will not increase. **Figure 5** illustrates the two cases.

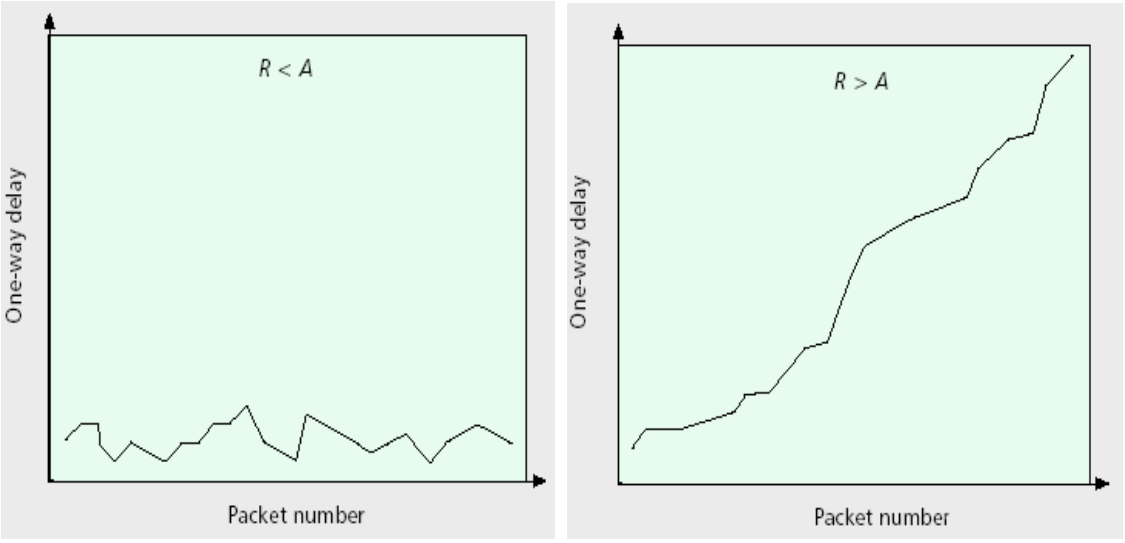

Figure 5. ***One-way delays increase when the stream rate* R > *available bandwidth* A, *but do not increase when* R < A.**

In SLoPS the sender attempts to bring the stream rate R close to the available bandwidth A, following an iterative algorithm similar to binary search. The sender probes the path with successive packet trains of different rates, while the receiver notifies the sender about the one-way delay trend of each stream. The sender also makes sure that the network carries no more than one stream at any time. Also, the sender creates a silent period between successive streams in order to keep the average probing traffic rate to less than 10 percent of the available bandwidth on the path.

The available bandwidth estimate A may vary during the measurements. SLoPS detects such variations when it notices that the one-way delays of a stream do not show a clear increasing or nonincreasing trend. In that case the methodology reports a *grey region*, which is related to the variation range of the available bandwidth A during the measurements.

This technique will be presented in detail in Chapter 5, as it is the technique that will be implemented in this project.

3.1.4. Trains of Packet Pairs

B. Melander, M. Bjorkman, and P. Gunningberg proposed a measurement methodology to estimate the available bandwidth of a network path. TOPP sends many packet pairs at gradually increasing rates from the source to the sink.

Suppose a packet pair is sent from the source with initial dispersion $?_S$. The probing packets have a size of L bytes; thus, the offered rate of the packet pair is $R_o = L/?_S$. If R_o is more than the end-to-end available bandwidth A, the second probing packet will be queued behind the first probing packet, and the *measured rate* at the receiver will be $R_m < R_o$. On the other hand, if $R_o < A$, TOPP assumes that the packet pair will arrive at the receiver with the same rate it had at the sender ($R_m = R_o$). Note that this basic idea is analogous to SLoPS. In fact, most of the differences between the two methods are related to the statistical processing of the measurements. Also, TOPP increases the offered rate linearly, while SLoPS uses a binary search to adjust the offered rate.

An important difference between TOPP and SLoPS is that TOPP can also estimate the capacity of the tight link of the path. Note that this capacity may be higher

than the capacity of the path if the narrow and tight links are different. To illustrate TOPP, consider a single-link path with capacity C, available bandwidth A, and average cross traffic rate $R_c = C - A$. TOPP sends packet pairs with an increasing offered rate R_o. When R_o becomes larger than A, the measured rate of the packet pair at the receiver will be:

$$R_m = \frac{R_o}{R_o + R_c} C$$

, or

$$\frac{R_o}{R_m} = \frac{R_o + R_c}{C}$$

TOPP estimates the available bandwidth A to be the maximum offered rate such that $R_o \sim R_m$. The last equation is used to estimate the capacity C from the slope of R_o/R_m vs. R_o.

Unfortunately, in paths with multiple links, the R_o/R_m curve may show multiple slope changes due to queuing at links having higher available bandwidth than A, so the accuracy is unclear.

3.2. Existing available bandwidth estimation tools

Now, I will try to summarize the main existing tools for available bandwidth estimation.

Cprobe was the first tool to attempt to measure end-to-end available bandwidth. *Cprobe* measures the dispersion of a train of eight maximum-sized packets. However, it has been previously shown that the dispersion of long packet trains measures the *dispersion rate*, which is not the same as the end-to-end available bandwidth. In general, the dispersion rate depends on all links in the path as well as on the train's initial rate. In contrast, the available bandwidth only depends on the tight link of the path.

Pathload implements the SLoPS methodology. It requires access at both ends of the path and support a bandwidth range rather than a single estimate. It sends a periodic packet trains with different rates and measures the one-way delays (OWD). In case of an

overload link there will be an increasing OWD trend (as queue builds up) otherwise there will be a non-increasing trend. The trends are detected using two algorithms.

IGI/PTR uses the PPD (packet pair dispertion) mechanism. Multiple packet pairs are sent with increasing gap size. The IGI algorithm computes the rate of the competing traffic while the PTR algorithm computes the available bandwidth.

PathChirp uses packet trains with exponential spaced packets called chirps. It measures interarrival times and therefore does not require time synchronization. Delay signatures measured are separated into excursions (segments) where all packets are part of the same busy period.

Spruce uses Poisson distributed packet pairs (exponentially inter probe times) and a sliding window average of the sample measurements to continuously provide available bandwidth information. Spruce requires knowledge about the bottleneck capacity, as it computes the available bandwidth as a difference between the capacity and a calculated rate.

Netest sends periodic packet trains. It determines amount of cross traffic with an algorithm called feedback adaptive control. The algorithm sends with a certain rate, measures the received rate and adapts the sender rate until the receiver rate equals the sender rate. It can also be used to calculate capacity.

I have to mention here three tools that measure the achievable TCP throughput for a path. TTCP, NetPerf, and Iperf are all tools that use large TCP transfers to measure the achievable throughput in an end-to-end path. The user can control the socket buffer sizes and thus the maximum window size for the transfer. *TTCP* (Test TCP) was written in 1984, while the more recent *NetPerf* and *Iperf* have improved the measurement process and can handle multiple parallel transfers.

All three tools require access at both ends of the path, but do not require superuser privileges.

Iperf gives you the possibility to calculate the available bandwidth, and I will use it as reference for my results.

Tool	Author	Measurement metric	Methodology
cprobe	Carter	End-to-end available bandwidth	Packet trains
pathload	Jain-Dovrolis	End-to-end available bandwidth	Self-loading periodic streams
IGI	Hu	End-to-end available bandwidth	Self-loading periodic streams
pathChirp	Ribeiro	End-to-end available bandwidth	Self-loading packet chirps
ttcp	Muuss	Achievable TCP throughput	TCP connection
Iperf	NLANR	Achievable TCP throughput	Parallel TCP connections
Netperf	NLANR	Achievable TCP throughput	Parallel TCP connections

Figure 6. Main existing tools

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

4.1. The MonALISA Services architecture

The MonALISA (Monitoring Agents in A Large Integrated Services Architecture) system provides a distributed service architecture which is used to collect and process monitoring information. While its initial target field of application is networks and Grid systems supporting data processing and analysis for global high energy and nuclear physics collaborations, MonALISA is broadly applicable to many fields of “data intensive” science, and to the monitoring and management of major research and education networks. MonALISA is based on a scalable Dynamic Distributed Services Architecture (DDSA), and is implemented in Java using JINI and WSDL technologies. The scalability of the system derives from the use of a multi threaded engine to host a variety of loosely coupled self-describing dynamic services, the ability of each service to register itself and then to be discovered and used by any other services, or clients that require such information. The framework integrates many existing monitoring tools and procedures to collect parameters describing computational nodes, applications and network performance. Specialized mobile agents are used in the MonALISA framework to perform global optimization tasks or help and improve the operation of large distributed system by performing supervising tasks for different applications or real time parameters. MonALISA is currently running around the clock monitoring several Grids and distributed applications on around 150 sites.

A service in the DDSA framework is a component that interacts autonomously with other services through dynamic proxies or 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 to be notified of a selected

set of event types, even if there is no provider to do the notification 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.

When a service is created, both the code and the appropriate parameters are downloaded dynamically. 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 DDSA service features and code mobility makes it possible build an extensible hierarchy of services capable of managing very large Grids, with relatively little program code.

The services are managed by an efficient multithreading engine that schedules and oversees their execution, such that data handling operations are not disrupted if one or more tasks (threads) are unable to continue. The system design also provides reliable ``non-stop'' support for large distributed applications under realistic working conditions, through service replication, and automatic re-activation of services. These mechanisms make the system robust against the failure or inaccessibility of multiple Grid components.

MonALISA services are organized in groups and this attribute is used for registration and discovery. Each MonALISA service registers with a set of JINI Lookup Discovery Services (LUS) as part of a group, and having a set of dynamic attributes. The LUSs are also JINI services and each one may be registered with the other LUSs. 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 or 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 done using a MonALISA proxy service and is based on a remote event notification mechanism which also supports subscription.

4.2. The Monitoring Service

An essential part of managing a global Data Grid 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, self-describing 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.

4.2.1. The 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, Hawkeye).

The core of the monitoring service is based on a multi-threaded 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 5**. 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.

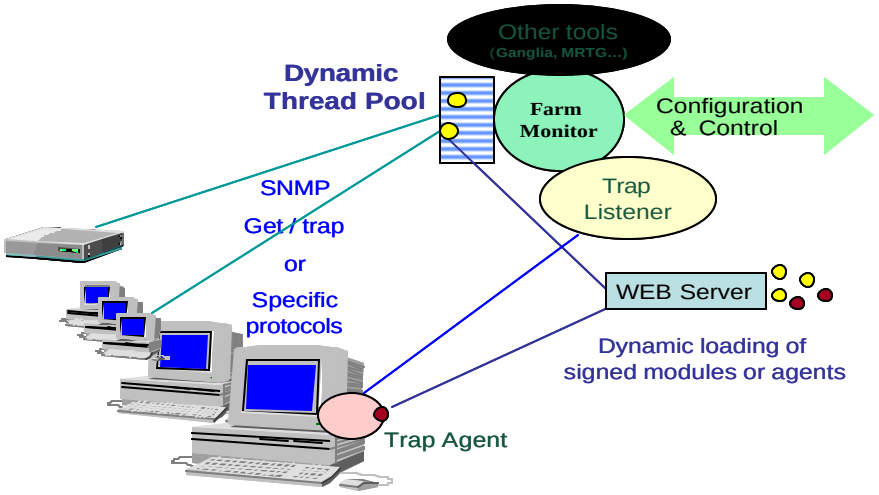

Figure 5. A schematic view of the data collection mechanism based on a multi-threaded engine.

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. As an example, we monitored 500 compute nodes performing a request for ~200 metric values per node every 60 seconds. This provided a sustained rate of ~1600 metric values per second collected, using an average of 20 active threads. The number of threads necessary to monitor a complete site is dynamically adjusted, and very

dependent on the response time for each node, which is related to its load as well as to the quality of the network connections.

4.2.2. Data Storage

The collected values are stored in a relational database, locally for each service. The JDBC framework in JAVA offers the flexibility to dynamically load any driver and connect to virtually any relational database. A normalized scheme is used to store the result objects provided by the monitoring modules in indexed tables, which are themselves generated as needed, dynamically. As data get older, we compress the values stored in the database by evaluating the mean values over larger time intervals, while keeping the fluctuation range for each parameter.

4.2.3. Registration and Discovery

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. 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.

4.2.4. Effective Data Handling: Predicates, Filters and Agents

The clients, other services or agents 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 constraints for selecting the values. In case of requests for historical data, the predicates are used to generate SQL queries to the local database. The subscription requests create a dedicated thread, to serve each client. This thread performs a 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 and 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 no longer 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. Currently, Web Services technology does not provide the functionality to register as a listener, and to receive the future measurements a client may want to receive.

Other applications or clients may also use the Agent Filters to receive the

information they need. The Agent Filter is a java module which can be dynamically deployed to any MonALISA service. It is designed to perform a dedicated data processing task on local data (by subscribing with a predicate to the data flow) and to return the processed information periodically. The MonALISA service provides the run time environment for these agents, which must be digitally signed by a trusted certificate. As an example, such filters are used to compute the aggregate I/O traffic in a farm, or to provide the number of nodes which are free. The same thread used for handling the predicate subscription is used for sending the filtered results back to each client. Dynamically loadable alarm agents, and agents able to take actions when abnormal behavior is detected, are currently being developed to help with managing and improving the working efficiency of the facilities, and the overall Grid system being monitored.

The clients, or any other services, use a set of proxies to connect and get information from the monitoring services. These proxy services are used to allow monitoring services to run behind firewalls, and to control the connections performed by services. At the same time, these services are used to provide an intelligent multiplexing of the same information if requested by more than one client or service. The way clients connect to monitoring information using the MonALISA proxy services is presented in **Figure 2** In general, clients discover the nearest proxy service and use it to get the information, but a dynamic load-balancing mechanism is also used to distribute the load among the available proxies, so that monitoring information is served to many clients or services without increasing the number of connections or load on individual monitoring services.

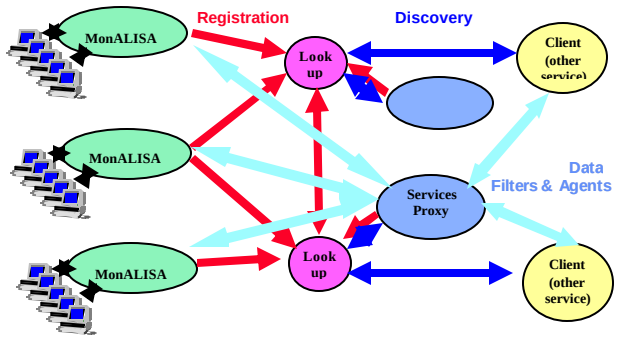

Figure 6. MonALISA proxy services are used for accessing monitoring information from clients or other services.

4.3. Clients and Data Access

We have developed a global graphical client which uses the discovery mechanism to find all of the active services from a list of user defined groups. This graphical client is implemented as a Web Start application that can be started and used from any web browser with little effort.

A MonALISA service may provide its own GUI to any client as a complex proxy containing the marshalled components as an attributed to the service. This GUI communicates with each service from which the user wants detailed information and plots the requested values. MonALISA provides flexible access to real-time or historical monitoring values by using either a predicate subscription mechanism or dynamically loadable filter agents. These mechanisms are used by any interested client to query and subscribe to only the information it needs, or to generate specific aggregate values in an appropriate format. When a client subscribes with a predicate to certain values, the GUI will automatically update as new values matching the subscriptions are collected.

The graphical user interface allows users to visualize global parameters from multiple sites, as well as detailed tracking of parameters for any individual site or component in the entire system. The graphical clients also use the remote notification mechanism, and thus are able to dynamically show when new services are started or when services become unavailable. Dedicated filers are used to provide global views with real time updates for all of the running services.

In **Figure 7**, we present a few examples in how real-time and historical data are presented in MonALISA.

4.4. Monitoring the VRVS System

The Virtual Rooms VideoConferencing System (VRVS) is an enhanced web based video conferencing system which is using a set of reflectors distributed world wide for an efficient real-time distribution of the audio and video streams.

For each VRVS reflector, a MonALISA service is running using an embedded Database, for storing the results locally, and runs in a mode that aims to minimize the

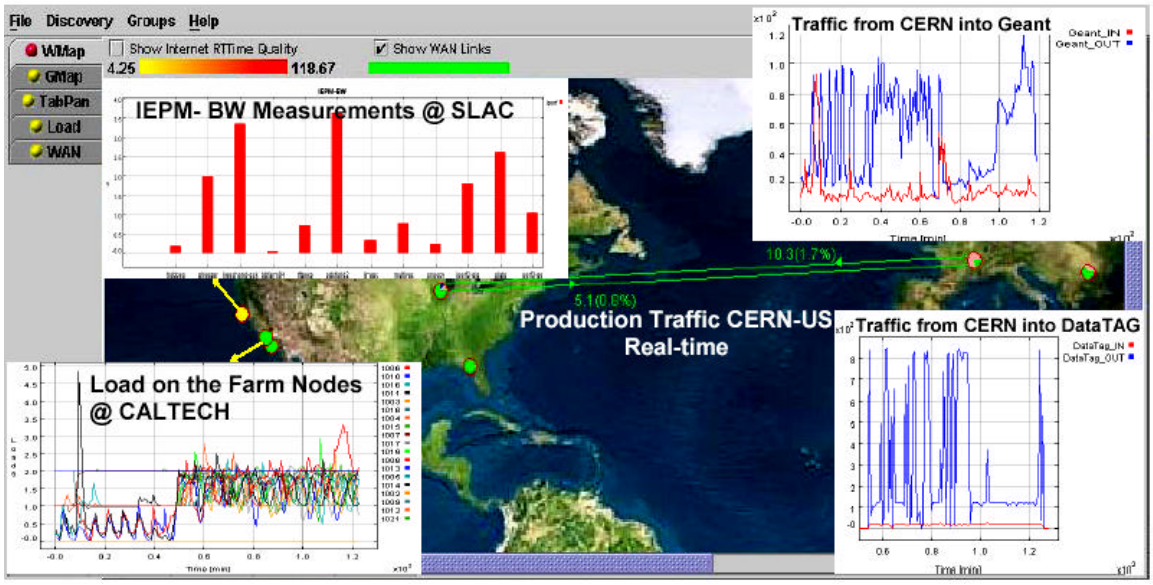

Figure 7. The main GUI in MonALISA: it provides globals views of the system as well as real time and historical plots

reflector resources it uses (typically less than 16MB of memory and practically without affecting the system load).

Dedicated modules to interact with the VRVS reflectors were developed: to collect information about the topology of the system; to monitor and track the traffic among the reflectors and report communication errors with the peers; and to track the number of clients and active virtual rooms. In addition, overall system information is monitored and reported in real time for each reflector: such as the load, CPU usage, and total traffic in and out.

A dedicated GUI for the VRVS version was developed as a java web-start client. This GUI provides real time information dynamically for all the reflectors which are monitored. If a new reflector is started it will automatically appear in the GUI and its connections to its peers will be shown. Filter agents to compute an exponentially mediated quality factor of each connection are dynamically deployed to every MonALISA service, and they report this information to all active clients who are subscribed to receive this information.

It provides real-time information about the way the VRVS system is used (number of conferences or clients) the topological connectivity of the reflectors and the quality of

it and system related information (IO traffic CPU load). Clients can also get historical data for any of these parameters.
The subscription mechanism allows one to monitor in real time any measured parameter in the system as all the updates are dynamically displayed on the open windows.

4.5. Optimized Dynamic Routing

We have developed agents able to provide an optimized dynamic routing of the videoconferencing data streams for the VRVS system. These agents use information about the quality of the alternative connections in the system to produce, in real-time, a minimum spanning to optimize the data flow at the global level.

Monitoring agents perform ping-style measurements using UDP probes to measure the quality of the connection with possible peer reflectors. These agents are deployed on all MonALISA services that run on the reflectors. They perform the measurements continuously with a set of peers that are dynamically configured for each reflector. The probe packets are small UDP datagrams sent back and forth and are used to compute the RTT, jitter, and the percentage of lost packages.

4.5.1. Minimum Spanning Trees

The reflectors and all of the possible peer connections define a graph. The best routing path for replication of the multimedia streams is defined as a Minimum Spanning Tree (MST). The task is to find the tree that contains all the reflectors (vertices in the Graph G) for which the total connection "cost" is minimized:

$$MST = \min\left(\sum_{(v,u)\in G} w((v,u))\right)$$

The "cost" of the connection between two reflectors (w) is evaluated using the UDP measurements from both sides. This cost function is build with an exponentially mediated RTT and if lost packages are detected or the jitter of the RTT is high the cost function will increase rapidly.

Based on these values provided by the deployed agents, the MST is calculated nearly in real - time. It has been implemented the Baruvka's Algorithm, as it is well suited for a parallel/distributed implementation. Once a link is part of the MST a momentum factor is attached to that link. This is to avoid triggering reconnections for small fluctuations in the system. Such cases may occur when two possible peers have very similar parameters (or they may be at the same location). In **Figure 8** an example of a dynamically MST for connecting the VRVS reflectors is presented.

This is an example of a high level service developed to optimize a real-time world wide distributed application

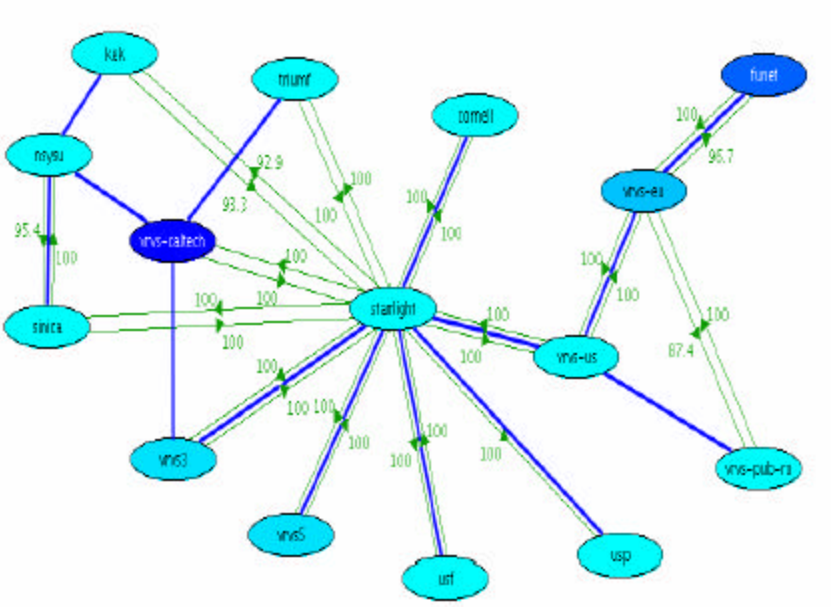

Figure 9. MST connections and peer link qualities for a set of VRVS Reflectors

The MST computation is an example of a high level service developed to optimize a real-time world wide distributed application and to help in operating such complex systems. The use of the MST peers optimization strategy has proved that MonALISA can successfully be used to monitor and control a distributed application, making the application more robust and efficient.

4.5.2. Getting qualities of internet links between reflectors – the ABPing module

The ABPing module computes the "cost" of the connection between two reflectors. The module sends UDP packets to the other reflectors. The other reflectors respond sending back the received packet. This way we can determine simple, but important factors that influence the quality of each link. The quality is computed with the following formula:

RTimeQuality = OVERALL_COEF + RTT_COEF * rtt
+ PKT_LOSS_COEF * loss% + JITTER_COEF * jitter

This formula is flexible enough to permit calculating any kind of quality, based on RTT, Packet Loss and Jitter. The values obtained by pinging peers are:

- rtt – the round trip time for packets to travel to the peer and back;
- loss – percent, ranging from 0 to 1 of lost packets sent to the peer;
- jitter – sum of the variations of rtt for a set of samples, divided by the average rtt and number of samples.

The list of available peers for each reflector and the *_COEF coefficients should be highly configurable to allow easy reconfiguration. To reach this goal, the configuration file is the same for all reflectors, each one knowing to extract only the information that is needed. The coefficients must be the same for all reflectors in order to obtain comparable RTime qualities.

The configuration file is loaded at start, and then it is periodically checked, from a URL configured when starting MonALISA service on the reflector. If there is a new peer for a reflector, it is added to the list of peers in the monABPing module. Similarly, if a known peer isn't found anymore in the configuration file, it is deleted from the peer list. If at least one of the coefficients modify, all measurements are reset and the new values are computed using the previous formula.

Practicly, my project estimates another "cost", quality of the link, which is the available bandwidth.

Chapter 5 – SLoPS. Measuring end-to-end available bandwidth

5.1. Basic Idea

Next, I will describe the measurement methodology that I will implement and that is Self-Loading Periodic Streams (SLoPS).

A periodic stream in SLoPS consists of K packets of size L, sent to the path at a constant rate R. *If the stream rate R is higher than the avail-bw A, the one-way delays of successive packets at the receiver show an increasing trend*. I first illustrate this fundamental effect in its simplest form through an analytical model with stationary and fluid cross traffic. Then, I show how to use this 'increasing delays' property in an iterative algorithm that measures end-to-end avail-bw. Finally, I depart from the previous fluid model, and observe that the avail-bw may vary during a stream. This requires to refine SLoPS in several ways, that is the subject of the next section.

First I have to make some basic assumptions. I consider a network path P as a sequence of H store-and-forward links that transfer packets from a sender *SND* to a receiver *RCV*. I assume that the path is fixed and unique (no routing changes or multipath forwarding occur during the measurements). Each link i can transmit data with a rate C_i bps (the *link capacity*). The two throughput metrics that are commonly associated with P are the end-to-end *capacity* C and *available bandwidth* A. As I stated in Chapter 2, *the end-to-end avail-bw is defined as the maximum rate that the path can provide to a flow, without reducing the rate of the rest of the traffic in P*.

Mathematically speaking, let's suppose that link i transmitted $C_i u_i(t_0, t_0 + t)$ bits during a time interval $(t_0, t_0 + t)$. The term $u_i(t_0, t_0 + t)$, or simply $u^t{}_i(t_0)$, is the average *utilization* of link i during $(t_0, t_0 + t)$, with $0 = u^t{}_i(t_0) = 1$. Intuitively, the avail-bw $A^t{}_i(t_0)$ of link i in $(t_0, t_0 + t)$ can be defined as the fraction of the link's capacity that has not been utilized during that interval:

$$A_i^\tau(t_0) \equiv C_i \left[1 - u_i^\tau(t_0)\right]$$

Extending this concept to the entire path, the end-to-end avail-bw $A^t{}_i(t_0)$ during $(t_0, t_0 + t)$ is the minimum avail-bw among all links in P,

$$A^{\tau}(t_0) \equiv \min_{i=1\ldots H} \{C_i\,[1 - u_i^{\tau}(t_0)]\}$$

5.2.1. SLoPS with fluid cross-traffic

Consider a path from *SND* to *RCV* that consists of H links, $i = 1, \ldots, H$. The capacity of link i is C_i. We consider a stationary (time invariant) and fluid model for the cross traffic in the path. So, if the avail-bw at link i is A_i, the utilization is $u_i = (C_i - A_i)/C_i$ and there are $u_i C_i t$ bytes of cross traffic departing from, and arriving at, link i in any interval of length t. Also, assume that the links follow the First-Come First-Served queueing discipline, and that they are adequately buffered to avoid losses. We ignore any propagation or fixed delays in the path, as they do not affect the delay variation between packets. The avail-bw A in the path is determined by the tight link t ? $\{1, \ldots, H\}$ with

$$A_t = \min_{i=1\ldots H} A_i = \min_{i=1\ldots H} C_i\,(1 - u_i) = A$$

Suppose that *SND* sends a periodic stream of K packets to *RCV* at a rate R_0, starting at an arbitrary time instant. The packet size is L bytes, and so packets are sent with a period of $T = L/R_0$ time units. The One-Way Delay (OWD) D^k from *SND* to *RCV* of packet k is:

$$D^k = \sum_{i=1}^{H}(\frac{L}{C_i} + \frac{q_i^k}{C_i}) = \sum_{i=1}^{H}(\frac{L}{C_i} + d_i^k)$$

, where $q^k{}_i$ is the queue size at link i upon the arrival of packet k ($q^k{}_i$ does not include packet k), and $d^k{}_i = q^k{}_i /C_i$ is the queueing delay of packet k at link i. The OWD difference between two successive packets k and $k + 1$ is:

$$\Delta D^k \equiv D^{k+1} - D^k = \sum_{i=1}^{H} \frac{\Delta q_i^k}{C_i} = \sum_{i=1}^{H} \Delta d_i^k$$

, where

$$\Delta q_i^k \equiv q_i^{k+1} - q_i^k$$

$$\Delta d_i^k \equiv \Delta q_i^k / C_i$$

We can now show that, if $R_0 > A$ the K packets of the periodic stream will arrive at *RCV* with increasing OWDs, while if $R_0 = A$ the stream packets will encounter equal OWDs. This property is stated next, and proved in Appendix A.

PROPOSITION 1. *If* $R_0 > A$, *then* $?D^k > 0$ *for* $k = 1, \ldots, K - 1$. *Else, if* $R_0 = A$, $?D^k = 0$ *for* $k = 1, \ldots, K-1$.

One may think that the avail-bw A can be computed directly from the rate at which the stream arrives at *RCV*. This is the approach followed in packet train dispersion techniques.

The following result, however, shows that, in a general path configuration, this would be possible only if the capacity and avail-bw of all links (except the avail-bw of the tight link) are *a priori* known.

PROPOSITION 2. *The rate* R_H *of the packet stream at RCV is a function, in the general case, of* C_i *and* A_i *for all* $i = 1, \ldots, H$.

This result follows from the proof in Appendix A (apply recursively Equation 19 until $i = H$).

5.2.2. An iterative algorithm to measure *A*

Based on Proposition 1, we can construct an iterative algorithm for the end-to-end measurement of A. Suppose that *SND* sends a periodic stream n with rate $R(n)$. The receiver analyzes the OWD variations of the stream, based on Proposition 1, to determine whether $R(n) > A$ or not. Then, *RCV* notifies *SND* about the relation between $R(n)$ and A. If $R(n) > A$, *SND* sends the next periodic stream $n + 1$ with rate $R(n + 1) < R(n)$. Otherwise, the rate of stream $n + 1$ is $R(n + 1) > R(n)$.

Specifically, $R(n + 1)$ can be computed as follows:

→ $R(n) > A, R^{max} = R(n)$

→ $R(n) \leq A, R^{min} = R(n)$

$R(n+1) = (R^{max} + R^{min})/2$

R^{min} and R^{max} are lower and upper bounds for the avail-bw after stream *n*, respectively. Initially, R^{min}=0 and R^{max} can be set to a sufficiently high value $R_0^{max} > A$. The algorithm terminates when $R^{max} - R^{min} = ?$, where ? is the *avail-bw estimation resolution*. If the avail-bw *A* does not vary with time, the previous algorithm will converge to a range $[R^{min}, R^{max}]$ that includes *A* after $[\log_2(R^{max}/?)]$ streams.

5.2.3. SLoPS with real cross-traffic

We assumed so far that the avail-bw *A* is constant during the measurement process. In reality, the avail-bw may vary because of two reasons. First, the avail-bw process $A^t(t)$ may be non-stationary, and so its expected value may also be a function of time. Even if $A^t(t)$ is stationary, however, the process A^t can have a significant statistical variability around its (constant) mean $E[A^t]$, and to make things worse, this variability may extend over a wide range of timescales t. How can we refine SLoPS to deal with the dynamic nature of the avail-bw process?

From the tests made, I concluded that there are three clear situations in analyzing the OWD variations of a stream.

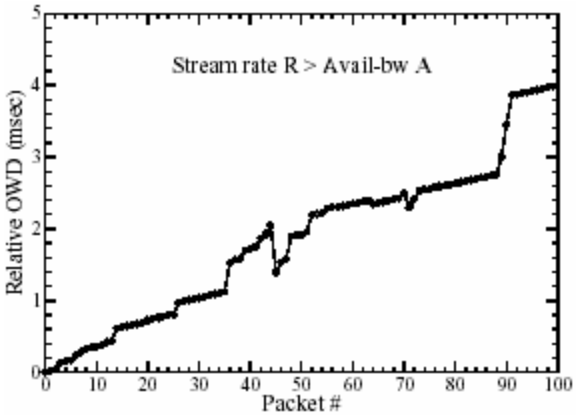

Figure 10.a. **R > A**

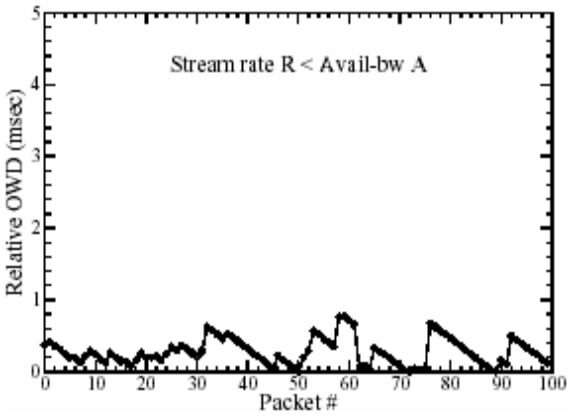

Figure 10.b. **R < A**

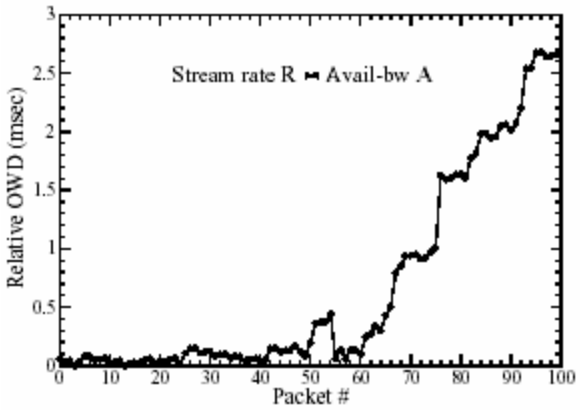

Figure 10.c. R <> A

Figure 10. OWD variations for a periodic stream of 100 packets

In **Figure 10.a**, the stream rate *R* is higher than the long-term avail-bw *A*. Notice that the OWDs between successive packets are not strictly increasing, as one would expect from Proposition 1, but *overall, the stream OWDs have a clearly increasing trend*. This is shown by the fact that most packets have a higher OWD than their predecessors.

On the other hand, the stream of **Figure 10.b** has a rate *R* lower than the long-term avail-bw *A*. Even though there are short-term intervals in which we observe increasing OWDs, *there is clearly not an increasing trend in the stream*.

The third stream, in **Figure 10.c** the stream does not show an increasing trend in the first half, indicating that the avail-bw during that interval is higher than *R*. The situation changes, however, after the 60-th packet. In that second half of the stream there is a clear increasing trend, showing that the avail-bw decreases to less than *R*.

The previous example motivates two important refinements in the SLoPS methodology. First, instead of analyzing the OWD variations of a stream, expecting one of the two cases of Proposition 1 to be strictly true for every pair of packets, we should instead watch for the presence of *an overall increasing trend during the entire stream*. Second, we have to accept the possibility that the avail-bw may vary around rate *R* during a probing stream. In that case, there is no strict ordering between *R* and *A*, and thus a third possibility comes up, that we refer to as 'grey-region' (denoted as *R* <> *A*).

In the next chapters I will describe the technologies that I used, the design and the implementation of my application, its integration in MonALISA, test results and an analysis of its accuracy.

Chapter 6 – Used Technologies

6.1. Java NIO

The Java Development Kit 1.4 provides developers non-blocking I/O on both sockets and files. For Java network programmers, non-blocking I/O is very exciting, because it makes writing scalable, portable socket applications simpler.

Previously, Java programmers would have to deal with multiple socket connections by starting a thread for each connection. Inevitably, they would encounter issues such as operating system limits, deadlocks, or thread safety violations. Now, the developer can use selectors to manage multiple simultaneous socket connections on a single thread. I will talk about selectors later.

6.1.1. Buffers

Starting from the simplest and building up to the most complex, the first improvement to mention is the set of *Buffer* classes found in the *java.nio* package. These buffers provide a mechanism to store a set of primitive data elements in an in-memory container. Basically, imagine wrapping a combined DataInputStream/DataOutputStream around a fixed-size byte array and then only being able to read and write one data type, like char, int, or double. There are seven such buffers available: *ByteBuffer, CharBuffer, DoubleBuffer, FloatBuffer, IntBuffer, LongBuffer, ShortBuffer.*

The *ByteBuffer* actually supports reading and writing the other six types, but the others are type specific. To demonstrate the use of a buffer, the following snippet converts a String to a CharBuffer and reads a character at a time. You convert the String to a CharBuffer with the wrap method, then get each letter with the get method.

```
CharBuffer buff = CharBuffer.wrap(args[0]);
for (int i=0, n=buff.length(); i<n; i++) {
        System.out.println(buff.get());
}
```

When using buffers, it is important to realize there are different sizing and

positioning values to worry about. The length method is actually non-standard, specific to CharBuffer. There is nothing wrong with it, but it really reports the *remaining* length, so if the position is not at the beginning, the reported length will not be the buffer length, but the number of remaining characters within the buffer. In other words, the above loop can also be written as follows:

```
CharBuffer buff = CharBuffer.wrap(args[0]);
for (int i=0; buff.length() > 0; i++) {
        System.out.println(buff.get());
}
```

Getting back to the different sizing and positioning values, the four values are known as *capacity, limit, position,* and *mark*:

capacity → the maximum number of data elements the buffer can hold; the capacity is set when the buffer is created and can never be changed

mark → A remembered position. Calling mark() sets *mark* = *position*. Calling reset() sets *position* = *mark*. The *mark* is undefined until set.

position → the index of the next element to be read or written; the position is updated automatically by relative get() and put() methods

limit → the first element of the buffer that should not be read or written; in other words, the count of live elements in the buffer

The following relationship between these four attributes always holds:

0 <= *mark* <= *position* <= *limit* <= *capacity*

The *position* is an important piece of information to keep in mind when reading from and writing to a buffer. For instance, if you want to read what you just wrote you must move the position to where you want to read from, otherwise, you'll read past the *limit* and get whatever just happens to be there. This is where the flip() method comes in handy, changing the limit to the current position and moving the current position to zero. You can also rewind() a buffer to keep the current limit and move the position back to zero.

The wrap() mechanism shown above is an example of a non-direct buffer. Non-direct buffers can also be created and sized with the allocate method, essentially wrapping the data into an array. Memory areas that are targets of I/O operations must be contiguous sequences of bytes. For this reason, the notion of a direct buffer was introduced. Direct buffers are intended for interaction with channels and native I/O routines. They make a best effort to store the byte elements in a memory area that a

channel can use for direct access by using native code to tell the operating system to drain or fill the memory area directly.

A direct ByteBuffer can be creates using the allocateDirect(int capacity) method. Direct buffers rely on the system's native I/O operations to optimize access operations.

6.1.2. Channels

Channels are the second major innovation of java.nio. They provide direct connections to I/O services. A Channel is a conduit that transports data efficiently between byte buffers and the entity on the other end of the channel (usually a file or socket).

A good metaphor for a channel is a pneumatic tube, the type used at drive-up bank-teller windows. Your paycheck would be the information you're sending. The carrier would be like a buffer. You fill the buffer (place your paycheck in the carrier), "write" the buffer to the channel (drop the carrier into the tube), and the payload is carried to the I/O service (bank teller) on the other end of the channel.

The response would be the teller filling the buffer (placing your receipt in the carrier) and starting a channel transfer in the opposite direction (dropping the carrier back into the tube). The carrier arrives on your end of the channel (a filled buffer is ready for you to examine). You then flip the buffer (open the lid) and drain it (remove your receipt). You drive away and the next object (bank customer) is ready to repeat the process using the same carrier (*Buffer*) and tube (*Channel*) objects.

The new socket channels can operate in nonblocking mode and are selectable. These two capabilities enable tremendous scalability and flexibility in large applications. It's no longer necessary to dedicate a thread to each socket connection (and suffer the context-switching overhead of managing large numbers of threads). Using the new NIO classes, one or a few threads can manage hundreds or even thousands of active socket connections with little or no performance loss.

All the socket channels (*SocketChannel*, *ServerSocketChannel*, and *DatagramChannel*) create a peer socket object when they are instantiated. These are the familiar classes from java.net (*Socket*, *ServerSocket*, and *DatagramSocket*), which have been updated to be aware of channels. The peer socket can be obtained from a channel by

invoking its socket() method. Additionally, each of the java.net classes now has a getChannel() method.

While every socket channel (in java.nio.channels) has an associated java.net socket object, not all sockets have an associated channel. If you create a *Socket* object in the traditional way, by instantiating it directly, it will not have an associated *SocketChannel*, and its getChannel() method will always return null.

To place a socket into nonblocking mode, we look to the common superclass of all the socket channel classes: *SelectableChannel*. The following methods are concerned with channel's blocking mode:

```
public abstract class SelectableChannel extends AbstractChannel implements Channel {
    // This is a partial API listing
    public abstract void configureBlocking (boolean block) throws IOException;
    public abstract boolean isBlocking();
    public abstract Object blockingLock();
}
```

Nonblocking sockets are usually thought of for server-side use because they make it easier to manage many sockets simultaneously. But there can also be benefits to using one or a few sockets in nonblocking mode on the client side. For example, with nonblocking sockets, a GUI application can pay attention to user requests and carry on conversations with one or more servers simultaneously. Nonblocking mode is useful across a broad range of applications.

6.1.3. Selectors

Next, 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.

For an illustration of readiness selection, let's return to the drive-through bank example. Imagine a bank with three drive-through lanes. In the traditional (nonselector) scenario, imagine that each drive-through lane has a pneumatic tube that runs to its own teller station inside the bank, and each station is walled off from the others. This means that each tube (channel) requires a dedicated teller (worker thread). This approach doesn't scale well and is wasteful. For each new tube (channel) added, a new teller is required, along with associated overhead such as tables, chairs, paper clips (memory, CPU cycles, context switching), etc. And when things are slow, these resources (which have associated costs) tend to sit idle.

Now imagine a different scenario in which each pneumatic tube (channel) is connected to a single teller station inside the bank. The station has three slots where the carriers (data buffers) arrive, each with an indicator (selection key) that lights up when the carrier is in the slot. Also imagine that the teller (worker thread) spends as much time as possible reading an interesting book. At the end of each paragraph, the teller glances up at the indicator lights (invokes `select()`) to determine if any of the channels are ready (readiness selection). The teller (worker thread) can perform another task while the drive-through lanes (channels) are idle yet still respond to them in a timely manner when they require attention.

While this analogy is not exact, it illustrates 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 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.

Selectors represent the most powerfull aspect of java NIO, as readiness selection is essential to large-scale, high-volume server-side applications.

6.2. JNI

The Java Native Interface (JNI) is the native programming interface for Java that is part of the JDK. By writing programs using the JNI, you ensure that your code is completely portable across all platforms.

JNI allows Java code that runs within a Java Virtual Machine (VM) to operate

with applications and libraries written in other languages, such as C, C++, and assembly. Programmers use the JNI to write native methods to handle those situations when an application cannot be written entirely in the Java programming language.

Programming through the JNI framework lets you use native methods to do many operations. Native methods may represent legacy applications or they may be written explicitly to solve a problem that is best handled outside of the Java programming environment.

The JNI framework lets your native method utilize Java objects in the same way that Java code uses these objects. A native method can create Java objects, including arrays and strings, and then inspect and use these objects to perform its tasks. A native method can also inspect and use objects created by Java application code. A native method can even update Java objects that it created or that were passed to it, and these updated objects are available to the Java application. Thus, both the native language side and the Java side of an application can create, update, and access Java objects and then share these objects between them.

Native methods can also easily call Java methods. Often, you will already have developed a library of Java methods. Your native method does not need to "re-invent the wheel" to perform functionality already incorporated in existing Java methods. The native method, using the JNI framework, can call the existing Java method, pass it the required parameters, and get the results back when the method completes.

It is easy to see that the JNI serves as the glue between Java and native applications. **Figure 11** shows how the JNI ties the C side of an application to the Java side.

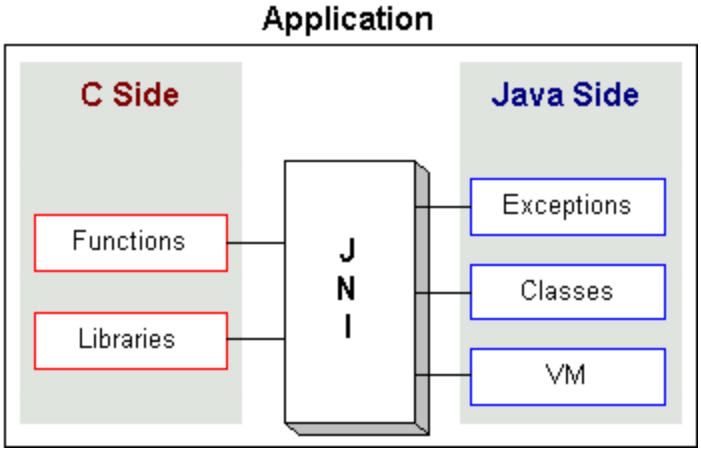

Figure 11. The interaction C- JNI -Java

Chapter 7 – Implementation and Design

In this chapter I will describe the implementation of this project, class structure and the integration in MonALISA.

7.1. Implementation

My implementation consists of two components: process SND running at the sender and process RCV running at the receiver. The tool uses UDP for the periodic packet streams. Additionally a TCP connection between the two end – points serves as a 'control channel'. The control channel transfers messages regarding the characteristics of each stream, the abortion or end of the measurement process, etc. In the followings I describe the implementation in detail.

A. Clock and timing issues

SND timestamps each packet upon its transmission. So, RCV can measure the *relative OWD* D^k of packet k, that differs from the actual OWD by a certain offset. This offset is due to the non-synchronized clocks of the end-hosts.
Since I am only interested in OWD differences, a constant offset in the measured OWDs does not affect the analysis. Clock skew can be a potential problem, but not here.

B. Selection of T and L

The transmission period T and the packet size L are two important parameters. First, the transmission rate R of a stream is:

$R = L/T$

Given the stream rate R, I will select values for L and T to satisfy the previous relation.

There are some practical constraints in the selection of L and T, however. Specifically, L can not be less than a certain number of bytes and it should not be more than the path's MTU (to avoid fragmentation). Also, if L is too small the possibility of

zero padding in certain layer links may cause a signifficant change in the layer packet size, and thus in the stream rate at those links.

On the other hand the transmission period *T* should be as small as possible. The reason is that as *T* increases, so does the duration of each stream. Ideally the transmission of each stream should complete before the processes SND or RCV get interrupted by a context switch at the end – hosts. Additionally, a lower value of *T* leads to a shorter duration for the entire measurement process. The minimum possible value of T depends on the hardware and operating system of the measurement hosts.

C. Selection of stream length K

There is a trade-off in the selection of the number of packets *K* in a stream. First, if *K* is too large the stream may overflow the queue of the tight link when $R > A$, causing losses in both the stream and the cross traffic packets.

On the other hand, if *K* is too small, the stream will not provide RCV with enough samples to infer in a robust manner whether there is an increasing trend in the measured OWDs. I use $K = 100$ packets, because this stream length rarely causes packet losses, while it provides an adequate number of OWD measurements to detect an increasing trend.

D. A fleet of streams

My project does not determine whether R > A based on a single stream. Instead, it sends a fleet of N streams. Each stream consists of K packets of size L bits, transmitted periodically in every *T* seconds. All streams in a fleet have the same rate $R = L/T$. Each stream is sent only when the previous stream has been acknowledged. This introduces an idle interval of one round-trip time ? between streams. The objective of this idle period is to let the path 'drain' the last stream before sending a next one.

There are two main reasons that I use *N* streams of *K* packets each, instead of a single fleet of $N \times K$ packets. First, having *N* streams allows us to examine *N* consecutive times whether $R > A$ or not. This is because RCV checks the measured OWDs for an increasing trend independently in each stream. Second, the use of multiple streams separated by a 'silence' period ? allows the queues in the network to drain our

measurement traffic and recover from the short-term overload that each stream causes. The default value for N is 12 streams.

N, K and T determine the duration U of a fleet, where:

$$U = N \times (K \times T + ?)$$

If a stream encounters excessive losses (>10%), or if more than a number of streams within a fleet encounter moderate losses (>3%), the entire fleet is aborted and the next fleet is send with the same rate.

E. Detecting an increasing OWD trend

Suppose that the (relative) OWDs of a particular stream are $D1, D2, \ldots, DK$. As a pre-processing step, I partition these measurements into $\Gamma = \sqrt{K}$ groups of G consecutive OWDs. Then, we compute the median OWD D^k of each group. Then I analyze the set $\{D^k, k = 1, \ldots, G\}$, which is more robust to outliers and errors.

I use two complementary statistics to check if a stream shows an increasing trend. The *Pairwise Comparison Test* (PCT) metric of a stream is

$$S_{PCT} = \frac{\sum_{k=2}^{\Gamma} I(\hat{D}^k > \hat{D}^{k-1})}{\Gamma - 1}$$

, where $I(X)$ is one if X holds, and zero otherwise. PCT measures the fraction of consecutive OWD pairs that are increasing, and so $0 = S_{PCT} = 1$. If the OWDs are independent, the expected value of S_{PCT} is 0.5. If there is a strong increasing trend, S_{PCT} approaches one.

The *Pairwise Difference Test* (PDT) metric of a stream is:

$$S_{PDT} = \frac{\hat{D}^{\Gamma} - \hat{D}^1}{\sum_{k=2}^{\Gamma} |\hat{D}^k - \hat{D}^{k-1}|}$$

PDT quantifies how strong is the start-to-end OWD variation, relative to the OWD absolute variations during the stream. Note that $-1 = S_{PDT} = 1$. If the OWDs are independent, the expected value of S_{PDT} is zero. If there is a strong increasing trend, S_{PDT} approaches one.

In my project, the PCT metric shows an increasing trend if $S_{PCT} > 0.55$, while the PDT shows increasing trend if $S_{PDT} > 0.4$. These two threshold values for S_{PCT} and S_{PDT} (0.55 and 0.4, respectively) were chosen after the tests made.

There are cases in which one of the two metrics is better than the other in detecting an increasing trend. Consequently, if either the PCT or PDT metrics shows an 'increasing trend', the stream is considered as *type-I*, increasing. Otherwise, the stream is considered of *type-N*, non-increasing.

F. Grey-region

If a large fraction f of the N streams in a fleet are of type-I, the entire fleet shows an increasing trend and we infer that the fleet rate is larger than the avail-bw ($R > A$). Similarly, if a fraction f of the N streams are of type-N, the fleet does not show an increasing trend and we infer that the fleet rate is smaller than the avail-bw ($R < A$).

It can happen, though, that less than $N \times f$ streams are of type-I, and also that less than $N \times f$ streams are of type-N. In that case, some streams 'sampled' the path when the avail-bw was less than R (type-I), and some others when it was more than R (type-N). Then, the fleet rate R is in the 'grey-region' of the avail-bw, and write $R <> A$. The interpretation that we give to the grey-region is that *when $R <> A$, the avail-bw process $A^t(t)$ during that fleet varied above and below rate R, causing some streams to be of type-I and some others to be of type-N*. The averaging timescale t, here, is related to the stream duration V. In my project, f is set to 50%.

G. Rate adjustment algorithm

After a fleet n of rate $R(n)$ is over, I determine whether $R(n) > A$, $R(n) < A$, or $R(n) <> A$. I will present the iterative algorithm that determines the rate $R(n+1)$ of the next fleet.

First, together with the upper and lower bounds for the avail-bw R^{max} and R^{min}, I also maintain upper and lower bounds for the grey-region, namely G^{max} and G^{min}.
When $R(n) <> A$, one of these bounds is updated depending on whether $G^{max} < R(n) < R^{max}$ (update G^{max}), or $G^{min} > R(n) > R^{min}$ (update G^{min}). If a grey-region has not been detected up to that point, the next rate $R(n + 1)$ is chosen, as half-way between R^{min} and R^{max}. If a grey-region has been detected, $R(n + 1)$ is set half-way between G^{max} and R^{max} when $R(n) = G^{max}$, or half-way between G^{min} and R^{min} when $R(n) = G^{min}$. The complete rate adjustment algorithm, including the initialization steps, is given in . It is important to

note that this binary search approach succeeds in converging to the avail-bw, as long as the avail-bw variation range is strictly included in the $[R^{min}, R^{max}]$ range.

The measurement terminates not only when the avail-bw has been estimated within a certain resolution ? ($R^{max} - R^{min} = ?$), but also when $R^{max} - G^{max} = ?$ and $G^{min} - R^{min} = ?$, meaning when both avail-bw boundaries are within ? from the corresponding grey-region boundaries. The parameter ? is referred to as *grey-region resolution*.

The tool eventually reports the range $[R^{min}, R^{max}]$.

H. Detection of a sender context switch

In my implementation I check whether a context switch occurred at SND while a stream was being sent.

Suppose that t_i is the transmission time of packet i from SND. t_i is carried in packet i RCV compares the sending times of consecutive packets to see whether $t_{i+1} - t_i > T + W$, where W is maximum allowed deviation from the target period T. If $t_{i+1} - t_i > T + W$, I assume that SND was switched out after sending the i-th packet of a stream. Then, RCV splits the received stream into two substreams, one between packets 1 and i, and another between packets i +1 and K. If a substream includes less than K+1 packets, it is discarded from the OWD analysis.

I. Detection of a receiver context switch

I also check whether a context switch occurred at RCV while a stream was being received. Suppose that a_i is the arrival time of packet i at the RCV process. If RCV is switched out while receiving a stream, some of the stream packets will be accumulated in a kernel buffer at the receiving host. When RCV runs again, those packets are transferred from kernel to user space with a spacing of Q µs, where Q is the latency of the recvfrom system call. Typically, Q is a few microseconds and it can be measured at RCV before the measurements start. So, RCV can detect a local context switch comparing the arrival times of consecutive packets. If $a_{i+1} - a_i \sim Q$, packets i and $i+1$ are discarded from the OWD analysis.

J. Measurement latency

Since this implementation is based on an iterative algorithm, it is hard to predict how long will a measurement take. For the default tool parameters, and for a path with *A* ~ 100Mbps and ? = 100ms, the tool needs ~ 15 seconds to produce a final estimate.

7.2. Class structure

My project consists of two parts: the sender and the receiver. It is practicly a client-server model, and I implemented it using java NIO.

The server first initializies a *ServiceThreadPool* for handling the messages from the clients. Then, it opens a *ServerSocketChannel*, sets it in non-blocking mode, and registers it with a *Selector* for accepting connections from the clients:

```
// 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));

// Set nonblocking mode for the listening socket
serverChannel.configureBlocking (false);

// Register the ServerSocketChannel with the Selector
serverChannel.register (selector, SelectionKey.OP_ACCEPT);
```

When a client connects the server puts the ‘task’ in a worker’s queue. The ‘task’ represents the client’s socket channel. I have implemented a *LoadBalancingStrategy* for the workers’ queues.

Each W*orkerThread* has a *Selector*. Working with java NIO allows me to unify a worker’s jobs, meaning that when adding a ‘task’ in a worker’s queue, I register the socket channel with the worker’s selector so at the next iteration the worker will also test this socket channel. Each client has a unique *ConnectionHandler* object attached to the socket channel, that deals with the client’s control messages.

```
synchronized (internalLock) {
        selector.wakeup();
       // Register interest in when connection
        SelectionKey key =
```

```
                sc.register(selector, SelectionKey.OP_WRITE, new
                        ConnectionHandler(sc, this, sock_udp));
        }
```

In **Figure 12**, there is a diagram of the server's classes.

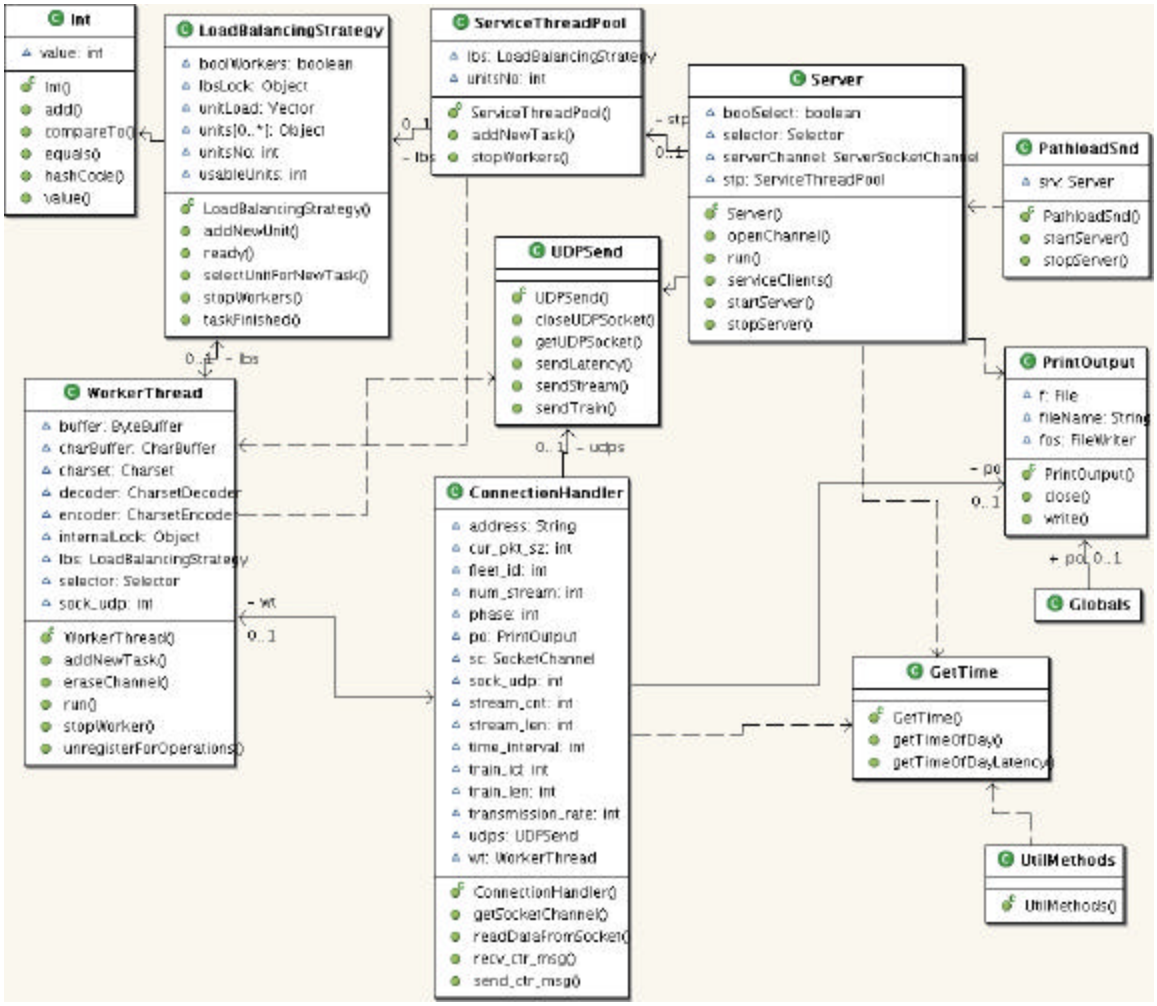

Figure 12. The server's class diagram

The *client* opens a *SocketChannel*, and connects to the server. The client' socket channel works in blocking mode as it does not need the non-blocking advantages.

After getting the UDP socket it then starts the algorithm described in the previous chapters.

The client's class diagram is described bellow in **Figure 13.**

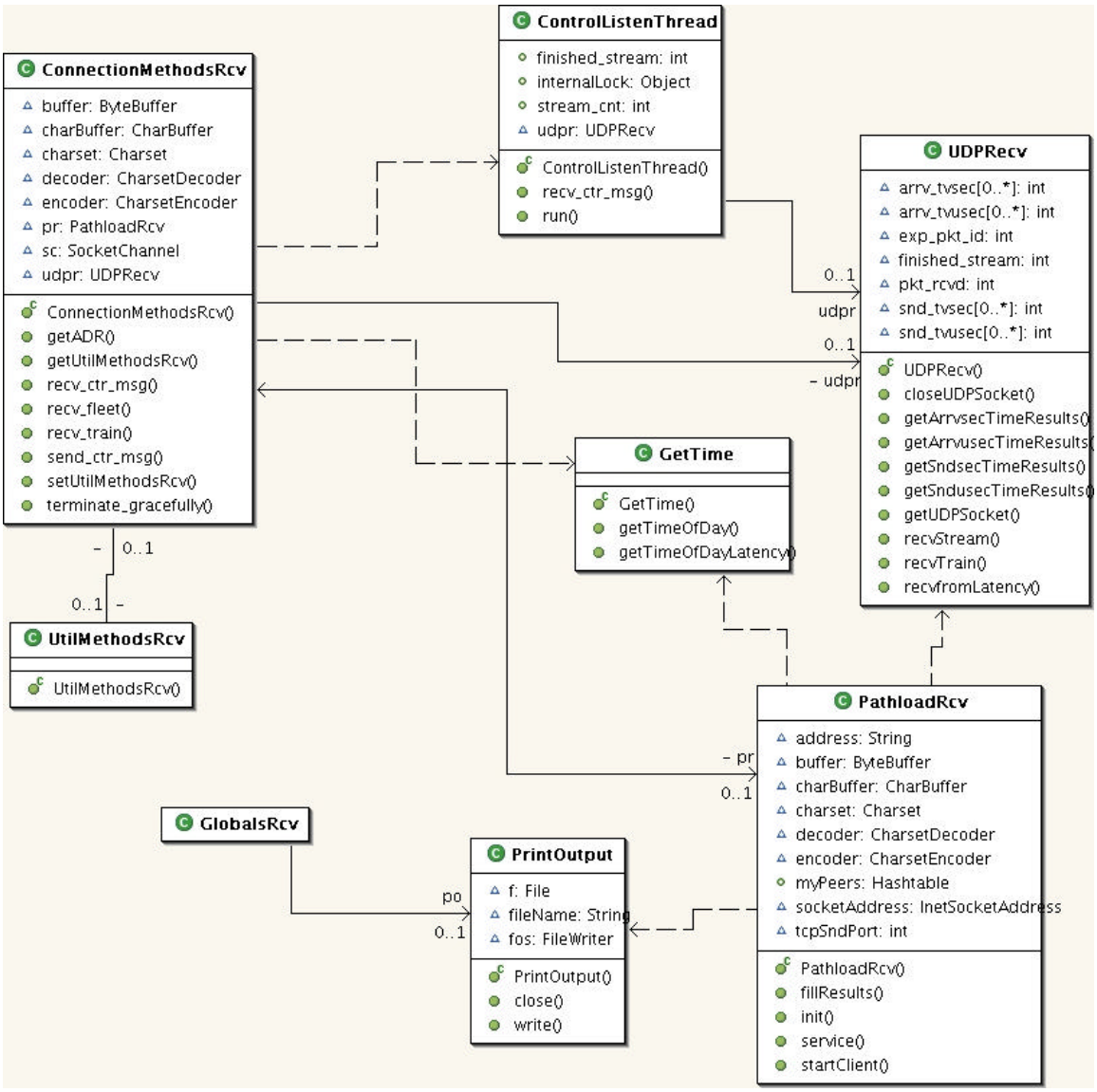

Figure 13. The client's class diagram

7.3. The utility of JNI

An important part of my implementation is the usage of native code C. The classes *GetTime*, *UDPRecv* and *UDPSend* have native methods defined and implemented in C.

```
// GetTime.java
public native void getTimeOfDay();
public native int getTimeOfDayLatency();

static {
        System.out.println("Loading time library...");
        System.load(Common.libpath + "/libtime.so");
}
```

```
// UDPRecv.java
public native int getUDPSocket(int UDPRCV_PORT,int UDP_BUFFER_SZ);
public native void closeUDPSocket(int sock_udp);
public native void recvStream(int cur_pkt_sz, int exp_fleet_id,
                              int stream_cnt, int MAX_STREAM_LEN,
                               int UDPRCV_PORT, int UDP_BUFFER_SZ,
                               int sock_udp);

static {
        System.load(Common.libpath + "/libudprecv.so");
}

//UDPSend
public native int getUDPSocket(int UDP_BUFFER_SZ);
public native void closeUDPSocket(int sock_udp);
public native int sendStream(int cur_pkt_sz, int fleet_id,
                             int stream_cnt, int stream_len, int time_interval,
                              int min_sleep_interval, int min_timer_intr,
                             int gettimeofday_latency, int UDPRCV_PORT,
                             String address, int sock_udp);

static {
        System.out.println("Loading udpsend library...");
        try {
                System.load(Common.libpath + "/libudpsend.so");
        } catch (Exception e) {
                System.out.println("Library udpsend not found");
         }
}
```

The need for native code C comes from the fact that I need very accurate timing. In jdk 1.4 you can only get time im milliseconds. In C, by using the gettimeofday system call you can get it in microseconds.

UDPSend and *UDPRecv* classes provide the methods for the UDP communication and they are written also in C. The idea is that, assuming that you use the path's MTU size for the packet to avoid fragmentation (1.500 bytes = 12.000 bits), for a path with ~100 Mbps capacity you have to obtain a transmission rate of ~100 Mbps, and this cand be obtained by sending 12.000 bits packets at a time interval of ~120 microseconds. So for a 1Gbps path the time interval must be of ~12 microseconds.

The latency of a send in Java is ~40 microseconds and so to high for testing the gigabit paths. On the other hand, the latency of a send in C is ~6 microseconds, enough to test those paths.

Chapter 8 – The Integration in MonALISA

8.1. Monitoring Modules

The main component that gathers data, injecting it into the system is a monitoring module. A monitoring module is a Java class that can be dynamically loaded from any location specified by a URL. At the same time with importing, data is also translated (usually by parsing) to a format understood by the MonALISA. With numerical data received from the monitored device, information about monitored node, such as name, cluster and farm, is also added.

Usually, these modules are invoked at fixed time intervals, using a priority queue. They can extract SNMP data, run rsh and ssh scripts where this is possible, connect through a TCP socket and query a device etc. In order to maintain up-to-date large distributed systems, these modules are built to be dynamically instantiated from certain, possible fixed, URLs.

All modules implement a *MonitoringModule* interface that allows different implementation for each module.

When it is invoked, the module returns a vector of *Results* that are passed further to the MonALISA core.

A monitoring module is a Java class that must implement the following interface:

```
public interface MonitoringModule extends lia.util.DynamicThreadPoll.SchJobInt {
        public MonModuleInfo init(MNode node, String args);
        public String[] ResTypes();
        public String getOsName();
        public Object doProcess() throws Exception;
        public MNode getNode();
        public String getClusterName();
        public String getFarmName();
        public boolean isRepetitive();
        public String getTaskName();
        public MonModuleInfo getInfo();
}
```

The *SchJobInt* is an interface that represents a job that can be scheduled for

execution. A monitoring module is such a job that monitors the activity on a certain *MNode* (monitored node that is part of a *Cluster*, on a *Farm*). It is invoked at configurable time intervals – the doProcess() method. If the job fails, it throws an exception. If it succeeds, it returns an object: a *Result*, or a vector of *Results*. These results are serialized and passed to the listening clients.

My module is called *MonPathload*. When this module is initialized, the configuration must be read from a certain URL, passed as a parameter in the *ml.properties* configuration file. Then, at fixed intervals, the configuration is reread from the same URL. This can be easily achieved by defining an inner class to handle this problem:

```
// the scheduling is made in the constructor
// timer task for reloading the config file
TimerTask task = new ConfigLoader();
Timer ttask = new Timer(true);
ttask.schedule(task, 0, CONFIG_RELOAD_INTERVAL);

class ConfigLoader extends TimerTask {
        public void run(){…}
}
```

In the doProcess() method we just call for each peer, the FillResults method of *PathloadRcv*. This is the "worker" class for this module (the client). *PathloadRcv* has a *Hashtable* with all its peers as key and a vector with measurements as values for each key. The FillResults method returns the last measurement made.

So the client resides in a *MonitoringModule*. The server, on the other hand is called from the AppControl and can be started or stoped remotely.

8.1. AppControl

All the modules must implement the lia.app.AppInt interface and must be packaged in .jar files that exactly respect the package structure.

The definition for lia.app.AppInt is:

```
package lia.app;

public interface AppInt {

        public boolean start();
        public boolean stop();
        public int status();
```

```
    public String info();
    public String exec(String sCmd);
    public boolean update(String sUpdate);
    public boolean update(String sUpdate[]);

    public String getConfiguration();
    public boolean updateConfiguration(String s);
    public boolean init(String sPropFile);
    public String getName();
    public String getConfigFile();

} // end of interface AppInt
```

start()

This function should start the service and return *true* if the service could be started and *false* if the service could not be started. Here I instantiate a *PathloadSnd* object, which represents the server and call its startServer() method. I do this if the object hasn't been already instantiated (two consecutive calls of start()).

stop()

This function should stop the service and return *true* if the service could be stopped and *false* if the service could not be stopped. Here I test whether the object has been instantiated and call its stopServer() method if it has.

restart()

This function calls stop() and then start().

status()

Returns one of the following codes:

- lia.app.AppUtils.APP_STATUS_STOPPED (0) - the application is not running
- lia.app.AppUtils.APP_STATUS_RUNNING (1) - the application is running
- lia.app.AppUtils.APP_STATUS_UNKNOWN (2) - application status could not be determined

info()

Returns a string with the application configuration files as an XML.

exec(String)

Executes the given command and returns the output of the command. You can return null if the application you are controlling does not accept any user commands.

update(String)

Changes the application configuration files according to the given argument. You should implement the commands explained in the Client-Server protocol document. The return value must be true if the requested update could be done or false if the configuration could not be updated.

update(String [])

Executes a set of updates.

getConfiguration()

Returns the content of the module's configuration file as a string value. I use lia.app.AppUtils.getConfig(Properties prop, String sFile)

updateConfiguration(String)

Replaces the content of the configuration file with the given string. I use lia.app.AppUtils.updateConfig(String sFile, String sContent)

init(String)

This function is called by the main program when the module is loaded. The parameter is the module's configuration file. I use lia.app.AppUtils.getConfig(Properties prop, String sFile) to read the contents of this file.

getName()

Should return the complete name of the module to make sure that there is no conflict in names.

getConfigFile()

Returns the configuration file name given as parameter to init(String).

Chapter 9 – Tests and Conclusions

9.1. Tests and evaluation

I have tested my algorithm between my computer from school and monalisa.cern.ch. It is a 100 Mbps connection and the path has 11 links.

The results are obtained from the MonALISA graphical client (**Figure 14**).

The results are from an hour analysis.

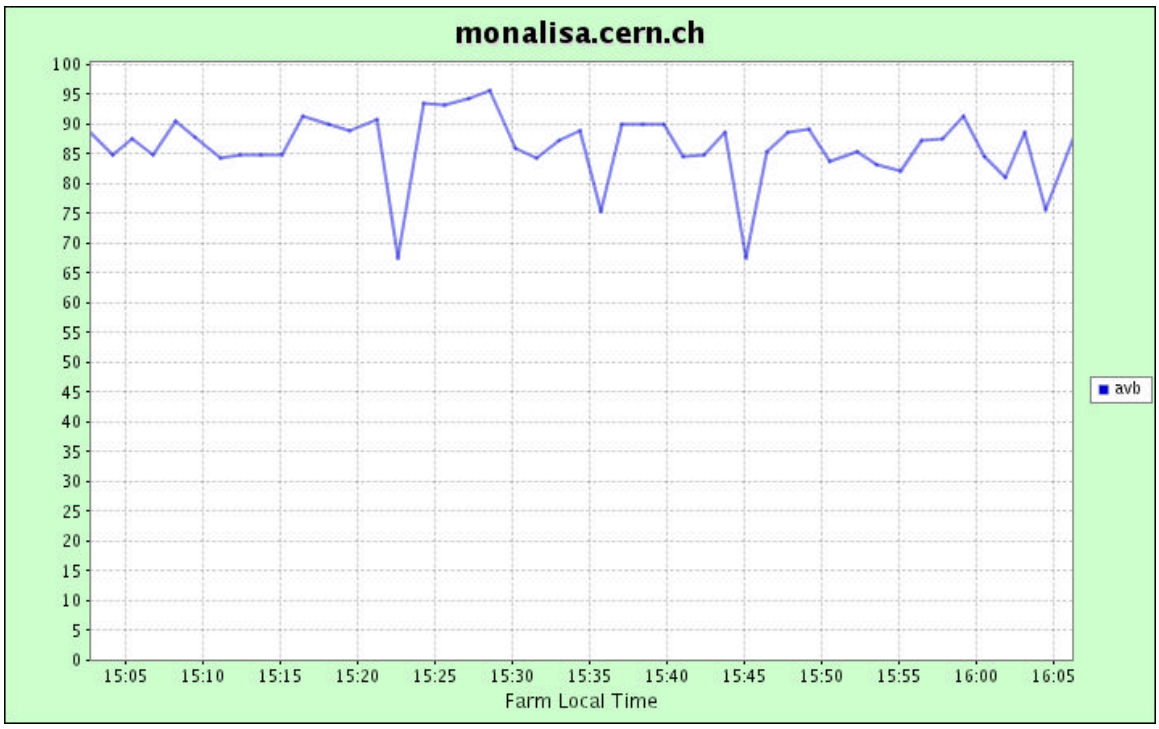

Figure 14. Tests between 141.85.99.167 – monalisa.cern.ch (100 Mbps connection)

The measurement latency was of about 15 seconds / measurement.

An important problem is the traffic generated. Each measurement needed about 10 fleets of streams to estimate the available bandwidth. The packets sent were about 1.000 bytes. A fleet has 12 streams and a stream has 100 packets. So the total traffic generated during a measurement process was ~1.000 * 100 * 12 * 10 bytes ~ 10Mbytes.

It seems high, but the fact is that this traffic is the total traffic; at a specific moment of time (sending a fleet), the traffic is about 1Mbyte.

9.2. Conclusions

My goal was to implement an accurate method for estimating the available bandwidth in a network end-to-end. From the results that I obtained, this implementation works pretty good.

By integrating it into the MonALISA system, I obtained more then an application that measures bandwidth end-to-end. I obtained a tool for monitoring different paths, and a tool that can improve the VRVS system.

By knowing the available bandwidth in real time, the VRVS reflectors can route the streams on the best path possible, so enhancing the performances of the system.

The tool must be tested more on the gigabit paths.

A future goal in this domain is to obtain the same accuracy with a less number of packets, and so with less traffic generated.

Besides adjusting the previous algorithm, we are analyzing a new idea: the simulation of the packet trains as impulse signals.

This can be done by considering a train of packets of different sizes (in SLoPS I use the same size for all the packets in a stream), and sent at different rates.(also, in SLoPE the rate is constant during a stream).

We can look at the train of packets as a sum of delta dirac impulses of amplitude l_i, where i is the i-th packet in the train.

$$I(t) = \sum_{i=1}^{n} l_i * d(t - t_i)$$

We are looking for a transfer function $H(L, A)$, where L is a latency and A is the available bandwidth so we can compute the output "signals":

$O(t) = H(t, L, A)$? $I(t)$ (the convolution)

$$O(t') = \int_{-\infty}^{\infty} H(t - t') * I(t) dt$$

This is the computed output. We also have the real output:

$$O(t') = \sum_{i=1}^{n} l_i * d(t' - t_i')$$

From these two relations we can adjust the available bandwidth in the transfer function.

Chapter 10 - References

- *J. C Bolot. - Characterizing End-to-End Packet Dispersion Delay and Loss in the Internet*
- *C. Dovrolis, P. Ramanathan si D. Moore – What do Packet Dispersion Tecniques Measure?*
- *A.B. Downey Using Pathchar to Estimate Internet Link Characteristics*
- *V. Jacobson – Congestion Avoidance and Control*
- *M. Jain si C. Dovrolis – End-to-End Available Bandwidth, Measurement Methodology, Dynamics and Relation with TCP Throughput*
- *Java NIO – Ron Hitchens; Publisher: O'Reilly*
- *Java Native Interface – Beth Stearns* http://java.sun.com/docs/books/tutorial/native1.1/index.html
- *MonALISA web page* http://monalisa.cacr.caltech.edu
- *H.B. Newman, I.C. Legrand, P. Galvez, R. Voicu, C. Cirstoiu (2003), "MonALISA: A Distributed Monitoring Service Architecture", CHEP03, La Jolla, California;*

Appendix - Code listing and demonstrations

A. Proof of Proposition 1

A.1 At the first link

Case 1: $R_0 > A_1$.

Suppose that t^k is the arrival time of packet k in the queue. Over the interval $[t^k, t^k + T)$, with $T = L/R0$, the link is constantly backlogged because the arriving rate is higher than the capacity ($R_0 + u_1 C_1 = C_1 + (R_0 - A_1) > C_1$). Over the same interval, the link receives $L + u_1 C_1 T$ bytes and services $C_1 T$ bytes. Thus,

$$\Delta q_1^k = (L + u_1 C_1 T) - C_1 T = (R_0 - A_1) T > 0$$

and so,

$$\Delta d_1^k = \frac{R_0 - A_1}{C_1} T > 0$$

Packet k +1 departs the first link ? time units after packet k, where

$$\Lambda = (t^{k+1} + d_1^{k+1}) - (t^k + d_1^k) = T + \frac{R_0 - A_1}{C_1} T$$

, that is independent of k. So, *the packets of the stream depart the first link with a constant rate* R_1, where:

$$R_1 = \frac{L}{\Lambda} = R_0 \frac{C_1}{C_1 + (R_0 - A_1)}$$

We refer to rate R_{i-1} as the *entry-rate* in link *i*, and to R_i as the *exit-rate* from link *i*. Given that $R_0 > A_1$ and that $C_1 = A_1$, it is easy to show that *the exit-rate from link 1 is larger or equal than* A_1 *and lower than the entry-rate* ($A_1 = R_1$ when $A_1 = C_1$):

$$A_1 \leq R_1 < R_0$$

Case 2: $R_0 > A_1$.

In this case, the arrival rate at the link in interval $[t^k, t^k + T)$ is $R_0 + u_1 C_1 = C_1$). So, packet k is serviced before packet $k + 1$ arrives at the queue. Thus,

$$\Delta q_1^k = 0, \quad \Delta d_1^k = 0, \text{ and } \quad R_1 = R_0$$

A.2 Induction to subsequent links

The results that were previously derived for the first link can be proved inductively for each link in the path. So, we have the following relationship between the entry and exit rates of link *i*:

$$R_i = \begin{cases} R_{i-1}\frac{C_i}{C_i+(R_{i-1}-A_i)} & \text{if } R_{i-1} > A_i \\ R_{i-1} & \text{otherwise} \end{cases}$$

, so

$$A_i \leq R_i < R_{i-1} \quad \text{when } R_{i-1} > A_i$$

Consequently, the exit-rate from link *i* is:

$$R_i \geq \min\{R_{i-1}, A_i\} \tag{A}$$

Also, the queueing delay difference between successive packets at link *i* is:

$$\Delta d_i^k = \begin{cases} \frac{R_{i-1}-A_i}{C_i} T > 0 & \text{if } R_{i-1} > A_i \\ 0 & \text{otherwise} \end{cases} \tag{B}$$

A.3 OWD variations

If $R_0 > A$, we can apply the result obtained before recursively for $i = 1, \ldots, (t\text{-}1)$ to show that the stream will arrive at the tight link with a rate $R_{t-1} = A_{t-1} > A_t$. Thus, $?d_t^k > 0$, and so the OWD difference between successive packets will be positive, $?d^k > 0$.

On the other hand, if $R_0 = A$, then $R_0 = A_i$ for every link i (from the deffinition of A). So, applying recursively the (A) formula from the first link to the last, we see that $R_i < A_i$ for $i = 1, \ldots, H$. Thus, (B) shows that the delay difference in each link i is $?d_i^k = 0$, and so the OWD differences are $?d^k = 0$.

B. Code Listing

```
//PathloadSnd

package lia.Monitor.Farm.Pathload;

import java.nio.channels.ServerSocketChannel;
import java.nio.channels.SocketChannel;
import java.nio.channels.Selector;
import java.nio.channels.SelectionKey;

import java.net.ServerSocket;
import java.net.InetSocketAddress;

import java.util.Iterator;
import java.util.Set;

import java.io.IOException;

public class PathloadSnd {

      Server srv;

      public PathloadSnd(int unitsNo) {
            srv = new Server(unitsNo);
      }

      public void startServer(){

            srv.start();
            System.out.println("[Pathload] ----> start server");
      }
      public void stopServer() {

            srv.stopServer();
            System.out.println("[Pathload] ----> 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_PKTSZ);
      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.TCP_SNDPORT;

      // 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));
        // 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
                                        // I kind of unify the
                                        //worker's jobs
                                        stp.addNewTask(s);

                                  } else {
                                        throw new
                                        IllegalStateException();
                                  }

                            }

                      }
                      // Never ends

                } catch (IOException e) {
                      Globals.po.close();
                      try {
                            selector.close();
                            serverChannel.close();
                      } catch (IOException ex) {
                      }
                }
          }
     }

}

// ServiceThreadPool
package lia.Monitor.Farm.Pathload;

import java.io.IOException;
import java.nio.channels.SocketChannel;

public class ServiceThreadPool {

     LoadBalancingStrategy lbs;
     int unitsNo;

     public ServiceThreadPool(int unitsNo) throws IOException {
          this.unitsNo = unitsNo;
          lbs = new LoadBalancingStrategy(unitsNo);
          for (int i = 0; i < unitsNo; i++) {
               WorkerThread wt = new WorkerThread(lbs);
               lbs.addNewUnit(wt);
               wt.start();
          }
     }

     public void addNewTask(SocketChannel sc) throws IOException {
          WorkerThread wt = (WorkerThread)lbs.selectUnitForNewTask();
          wt.addNewTask(sc);
     }
```

```
	public void stopWorkers() {
		lbs.stopWorkers();
	}
}
```

// WorkerThread

```
package lia.Monitor.Farm.Pathload;

import java.io.IOException;
import java.nio.channels.SelectionKey;
import java.nio.channels.Selector;
import java.nio.channels.SocketChannel;
import java.util.Iterator;
import java.util.Set;

import java.nio.ByteBuffer;
import java.nio.CharBuffer;
import java.nio.charset.Charset;
import java.nio.charset.CharsetDecoder;
import java.nio.charset.CharsetEncoder;

import java.net.Socket;

public class WorkerThread extends Thread {

	Charset charset = Charset.forName("ISO-8859-1");
	CharsetDecoder decoder = charset.newDecoder();
	CharsetEncoder encoder = charset.newEncoder();

	ByteBuffer buffer = ByteBuffer.allocateDirect (11);
	CharBuffer charBuffer = CharBuffer.allocate(11);

	LoadBalancingStrategy lbs;
	Selector selector;

	Object internalLock;

	int sock_udp;

	public WorkerThread(LoadBalancingStrategy lbs) throws IOException
	{
		this.lbs = lbs;
		selector = Selector.open();
		internalLock = new Object();

		// UDP Socket
		UDPSend udps = new UDPSend();
		sock_udp = udps.getUDPSocket(Common.UDP_BUFFER_SZ);

	}

	public void run() {
		while (true) {
			synchronized(lbs.lbsLock) {
				if (!lbs.boolWorkers) {
```

```
            try {
                  selector.close();
            } catch (IOException e) {}
            break;
      }
}
// Wait for something of interest to happen
/* If we have no connections to serve we
 * must block again
 */
//   System.out.println("Waiting   for   something   to
happen");
// This may block for a long time. Upon returning, the
// selected set contains keys of the ready channels.
try {
      //System.out.println("Worker      blocked      at
      select...");
      while (selector.select() == 0) {
            synchronized(lbs.lbsLock) {
                  if (!lbs.boolWorkers)
                        break;
            }
            //System.out.println("Worker   blocked   at
            internalLock...");
            synchronized (internalLock) {
            }
      }
} catch (IOException e) {
      // Selector exception
      break;
}

// 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();
      //  get  the  ConnectionHandler  associated  with
      this key
      ConnectionHandler   ch   =   (ConnectionHandler)
      key.attachment();
      SocketChannel sc = ch.getSocketChannel();

      // a control message from the client is on the
      channel
      if (key.isReadable()) {

            ch.readDataFromSocket();

            // remove interest for OP_READ

            //registerForOperations(sc,
```

```
                    (key.interestOps() & (~SelectionKey.OP_READ)));

                    }
                    else
                    if (key.isWritable()) {
                        /* tell this receiver our send latency */
                          try {

                          sc.write(encoder.encode
                          (CharBuffer.wrap(Integer.toString(Globals.
                    sendLatency))));
                          } catch (Exception e) {}

                          //  remove  interest  for  OP_WRITE  and
                          register interest in OP_READ
                          unregisterForOperations(sc,
                          (~SelectionKey.OP_WRITE));
                          registerForOperations(sc,
                          SelectionKey.OP_READ);
                    } else { // we did not register for this
                          throw new IllegalStateException();
                    }
              }
        }
}

public void addNewTask(SocketChannel sc) throws IOException {

        Socket s = sc.socket();
        String inetAddr = s.getInetAddress().toString();
        Globals.po.write(new  String("[]  New  connection  from  "  +
        inetAddr));

        synchronized (internalLock) {
              selector.wakeup();
              // Register interest in when connection
              SelectionKey key =
                    sc.register(selector, SelectionKey.OP_WRITE,
                                new    ConnectionHandler(sc,    this,
                                sock_udp));
        }
}

protected void registerForOperations(SocketChannel sc, int ops) {
        //synchronized (internalLock) {
        //    selector.wakeup();
        sc.keyFor(selector).interestOps(ops);
        //}
}

public void unregisterForOperations(SocketChannel sc, int ops) {
        //synchronized (internalLock) {
        //    selector.wakeup();
        SelectionKey key = sc.keyFor(selector);
        key.interestOps(key.interestOps() & ops);
        //sc.keyFor(selector).interestOps(ops);
        //}
```

```
	}

	public void eraseChannel(SelectionKey key) {
		// Get channel
		SocketChannel sc = (SocketChannel) key.channel();
		key.attach(null);
		key.cancel();
		lbs.taskFinished(sc);
		try {
			sc.close();
		} catch (IOException ex) {}
	}

	public void stopWorker() {
		selector.wakeup();
	}
}


// ConnectionHandler
package lia.Monitor.Farm.Pathload;

import java.nio.channels.SocketChannel;
import java.nio.channels.SelectionKey;

import java.net.Socket;
import java.net.InetAddress;

import java.nio.CharBuffer;

public class ConnectionHandler {

	SocketChannel sc;
	WorkerThread wt;
	PrintOutput po;

	int sock_udp;


	int phase;			// i need this to know what to do
					for each control message
					// represents the phase of the
					algorithm
					// ......? are the codes in
					Common.java sufficient ?......

	int train_id;

	int train_len;

	int fleet_id;
	int transmission_rate;
	int cur_pkt_sz;
	int stream_len;
	int time_interval;
	int num_stream;
	int stream_cnt;
```

```
String address;
UDPSend udps;

public ConnectionHandler(SocketChannel sc, WorkerThread wt, int
sock_udp) {
      this.sc = sc;
      Socket socket = sc.socket();
      InetAddress inetAddr = socket.getInetAddress();
      address = inetAddr.toString();
      address = address.substring(1);
      udps = new UDPSend();

      po = new PrintOutput(address);
      this.wt = wt; this.sock_udp = sock_udp;
      phase = 0; train_id = 0; train_len = 0; fleet_id = -1;
      stream_cnt = 0; num_stream = Common.NUM_STREAM;
}

public int readDataFromSocket () {

      // reads a control message from the client and analysis the
      faze of the algorithm
      // ......? should i create a JobPoolThread for the udp
      messages ?......
      int ctr_code = recv_ctr_msg();

      // check what control message the client has sent us
      if ( (((ctr_code & Common.CTR_CODE) >> 31) == -1) &&
      ((ctr_code & 0x7fffffff) == Common.SEND_TRAIN )
                  && phase == 0) {

            /* receiver starts ADR measurement */
            po.write("[PHASE 0 --> START ADR] : SEND_TRAIN");
            send_train();
            phase++;
            return 1;
      }
      if ( (((ctr_code & Common.CTR_CODE) >> 31) == -1) &&
      ((ctr_code & 0x7fffffff) == Common.GOOD_TRAIN)
                  && phase == 1) {
            po.write("[PHASE 1 --> ADR] : GOOD TRAIN");
            phase++;
            return 1;
      }
      if ( (((ctr_code & Common.CTR_CODE) >> 31) == -1) &&
      ((ctr_code & 0x7fffffff) == Common.BAD_TRAIN)
                  && phase == 1) {
            po.write("[PHASE 1 --> ADR] : BAD TRAIN " + train_id);
            train_id++ ;
            send_train();
            return 1;
      }
      if ( (((ctr_code & Common.CTR_CODE) >> 31) == -1) &&
      ((ctr_code & 0x7fffffff) == Common.SEND_FLEET)
                  && phase == 6) {
            po.write("[PHASE 6 --> START AVB] : SEND FLEET");
```

```
            phase = 2;

            ctr_code = Common.RECV_FLEET | Common.CTR_CODE ;
          send_ctr_msg(ctr_code );
          stream_cnt = 0;
          fleet_id++;
          po.write("\nSending fleet " + fleet_id);
          // start sending the fleet
          send_fleet();

            return 1;
        }
        if ( (((ctr_code & Common.CTR_CODE) >> 31) == -1) &&
        ((ctr_code & 0x7fffffff) == Common.CONTINUE_STREAM) ) {
            po.write("[PHASE x --> AVB] : CONTINUE STREAM");
            stream_cnt++ ;
            if (stream_cnt < Common.NUM_STREAM) {
                send_fleet();
            }

            return 1;
        }
        if ((((ctr_code & Common.CTR_CODE) >> 31) == -1 ) &&
        ((ctr_code & 0x7fffffff) == Common.ABORT_FLEET) ) {

            po.write("[PHASE x --> AVB] : ABORT FLEET");
            return 1;
        }

        if ((((ctr_code & Common.CTR_CODE) >> 31) == -1)
        &&((ctr_code & 0x7fffffff) == Common.TERMINATE)) {
            po.write("[PHASE 7 --> STOP AVB] : TERMINATE");
            po.write("\n\n");
            po.close();
          // remove interest for OP_READ
            wt.unregisterForOperations(sc,
            (~SelectionKey.OP_READ));
            wt.eraseChannel(sc.keyFor(wt.selector));
            return 1;
    }
        switch (phase) {
            case 2 :
                transmission_rate = ctr_code;
                po.write("[PHASE 2 --> PARAM] : TRANSMISSION
                RATE " + transmission_rate);
                phase++; break;
            case 3 :
                cur_pkt_sz = ctr_code;
                po.write("[PHASE 3 --> PARAM] : CUR_PKT_SZ " +
                cur_pkt_sz);
                phase++; break;
            case 4:
                stream_len = ctr_code;
                po.write("[PHASE 4 --> PARAM] : STREAM_LEN " +
                stream_len);
                phase++; break;
            case 5:
```

```
                  time_interval = ctr_code;
                  po.write("[PHASE 5 --> PARAM] : TIME_INTERVAL "
                  + time_interval);
                  phase++; break;

            default :
                  System.out.println("[PHASE > 2] CLIENT
                  TERMINATED");
                  po.write("\n\n");
                  po.close();
                  // remove interest for OP_READ
                  wt.unregisterForOperations(sc,
                  (~SelectionKey.OP_READ));
                  wt.eraseChannel(sc.keyFor(wt.selector));

      }

  return 1;
}

private int send_fleet() {

      int tmp1sec, tmp1usec, tmp2sec, tmp2usec;
      double  t1=0, t2 = 0 ;
      int ctr_code, ret_val ;

      po.write("#");
      // JNI call to send a stream of UDP packets in C
      udps.sendStream(cur_pkt_sz, fleet_id, stream_cnt,
      stream_len, time_interval, Globals.minSleepInterval,
            Globals.minTimerIntr, Globals.getTimeOfDayLatency,
                                    Common.UDP_RCVPORT, address,
                                    sock_udp);

      /* Wait for 2000 usec and send End of
       * stream message along with streamid.
       */
      //....? is this necessary ?...... don't think so
      GetTime getTime = new GetTime();
      getTime.getTimeOfDay();
      tmp2sec = GetTime.seconds;
      tmp2usec = GetTime.useconds;
      t1 = (double) tmp2sec * 1000000.0 +(double)tmp2usec ;
      do {
            getTime.getTimeOfDay();
            tmp2sec = GetTime.seconds;
            tmp2usec = GetTime.useconds;
            t2 = (double) tmp2sec * 1000000.0 +(double)tmp2usec ;
      } while((t2 - t1) < 2000 ) ;

      ctr_code = Common.FINISHED_STREAM | Common.CTR_CODE ;

      send_ctr_msg(ctr_code);
      send_ctr_msg(stream_cnt);

      return 1;
}
```

```
private int send_train() {

      int pack_id;
      byte train_id_n, pack_id_n ;
      int ctr_code, ret_val, i, train_len=0;

      if ( train_len == 5)
            train_len = 3;
      else
            train_len = Common.TRAIN_LEN - train_id*15;

      udps.sendTrain(Common.MAX_PKTSZ, train_len, train_id,
      sock_udp, address, Common.UDP_RCVPORT);

      // ......? is this sleep necessary ?.......don't think so
      try {
            Thread.sleep(0, 200000);
      } catch (Exception e) {}

      ctr_code = Common.FINISHED_TRAIN | Common.CTR_CODE ;
      send_ctr_msg(ctr_code );
      return 0 ;
}

/*
 * receive a tcp control message
 */
public int recv_ctr_msg() {
      int count;
      int ctr_code = -1;
      wt.buffer.clear( );           // Empty buffer
  try {
      count = sc.read (wt.buffer);
      wt.buffer.flip( );        // Make buffer readable
      wt.decoder.decode(wt.buffer, wt.charBuffer, false);
      wt.charBuffer.flip();
      ctr_code = Integer.parseInt(wt.charBuffer.toString());
      wt.buffer.clear();
      wt.charBuffer.clear();
  } catch (Exception e){}

  return ctr_code;
}

/*
 * sends a tcp control message
 */
public void send_ctr_msg(int ctr_code) {
      try {
            String code = Integer.toString(ctr_code);
            int len = code.length();
            for (int i = 0; i < 11 - len; i++) code = "0" + code;
            sc.write(wt.encoder.encode(CharBuffer.wrap(code)));
      } catch (Exception e) {}
}
```

```
      public SocketChannel getSocketChannel() {
            return sc;
      }
}

// PathloadRcv
package lia.Monitor.Farm.Pathload;

import java.net.Socket;
import java.net.InetSocketAddress;

import java.nio.ByteBuffer;
import java.nio.CharBuffer;
import java.nio.channels.SocketChannel;
import java.nio.charset.Charset;
import java.nio.charset.CharsetDecoder;
import java.nio.charset.CharsetEncoder;
import java.nio.channels.Selector;

import java.util.Hashtable;
import java.util.Vector;

import lia.Monitor.monitor.Result;

public class PathloadRcv {

      int tcpSndPort = Common.TCP_SNDPORT;
      static Selector selector;
      InetSocketAddress socketAddress;

      Charset charset;
      CharsetDecoder decoder;
      CharsetEncoder encoder;
      // Allocate buffers
      ByteBuffer buffer;
      CharBuffer charBuffer;
      String address;
      public Hashtable myPeers;

      public PathloadRcv() {
            myPeers = new Hashtable();
      }

      public void startClient (String addr) {
            address = addr;
            System.out.println("[PathloadRcv] : " + 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();
        }
    }
```

```
public void init() {
      socketAddress = new InetSocketAddress(address, tcpSndPort);
    charset = Charset.forName("ISO-8859-1");
    decoder = charset.newDecoder();
    encoder = charset.newEncoder();
    // Allocate buffers
    buffer = ByteBuffer.allocateDirect(1024);
    charBuffer = CharBuffer.allocate(1024);
    GlobalsRcv.po = new PrintOutput("client.log");

    GlobalsRcv.slow=0;
      GlobalsRcv.interrupt_coalescence=0;
      GlobalsRcv.bad_fleet_cs=0;
      GlobalsRcv.num_stream = Common.NUM_STREAM;
      GlobalsRcv.stream_len = Common.STREAM_LEN ;
      GlobalsRcv.exp_flag = 1;
      GlobalsRcv.num=0;
      GlobalsRcv.snd_time_interval=0;

      GlobalsRcv.converged_gmx_rmx = 0 ;
      GlobalsRcv.converged_gmn_rmn = 0 ;
      GlobalsRcv.converged_rmn_rmx = 0 ;
      //GlobalsRcv.counter = 0 ;
      //GlobalsRcv.prev_actual_rate = 0;
      //GlobalsRcv.prev_req_rate = 0 ;
      GlobalsRcv.cur_actual_rate = 0 ;
      GlobalsRcv.cur_req_rate = 0 ;
      GlobalsRcv.bw_resol=0;
      GlobalsRcv.increase_stream_len=0;
      GlobalsRcv.lower_bound=0;

      GlobalsRcv.exp_fleet_id = 0;

      GlobalsRcv.tr_min = 0; GlobalsRcv.tr_max = 0;
      GlobalsRcv.grey_min = 0 ; GlobalsRcv.grey_max = 0;
      GlobalsRcv.min_time_interval = 0; GlobalsRcv.sendLatency =
      0;
      GlobalsRcv.recvLatency = 0;
      GlobalsRcv.max_rate = 0; GlobalsRcv.min_rate = 0;
      GlobalsRcv.tr  = 0; GlobalsRcv.adr  = 0;
      GlobalsRcv.max_rate_flag  = 0; GlobalsRcv.min_rate_flag  =
      0;
      GlobalsRcv.converged_gmx_rmx_tm =
      0;GlobalsRcv.converged_gmn_rmn_tm = 0;
      GlobalsRcv.converged_rmn_rmx_tm = 0;
      GlobalsRcv.trend_idx = 0;
      GlobalsRcv.ic_flag = 0;
      GlobalsRcv.num_bursts = 0;
      GlobalsRcv.tmp_b2b = 0;
      GlobalsRcv.repeat_1 = 0; GlobalsRcv.repeat_2 = 0;
      GlobalsRcv.pct_metric = new double[50];
      GlobalsRcv.pdt_metric = new double[50];
      GlobalsRcv.bad_fleet_rate_mismatch = 0;
      GlobalsRcv.retry_fleet_cnt_cs = 0;
      GlobalsRcv.retry_fleet_cnt_rate_mismatch = 0;
```

```
        GlobalsRcv.TMP = 0;

        GlobalsRcv.cur_pkt_sz = 0; GlobalsRcv.transmission_rate = 0;
        GlobalsRcv.time_interval = 0;


    }

    public void service() {

        int trend = 0, prev_trend = 0, exp_start_timesec,
        exp_start_timeusec;
        GetTime getTime = new GetTime();
        getTime.getTimeOfDay();
        exp_start_timesec = GetTime.seconds;
        exp_start_timeusec = GetTime.useconds;

        UDPRecv udpr = new UDPRecv();

        try {

              // TCP Connection
              GlobalsRcv.sc = SocketChannel.open();
              Socket socket = GlobalsRcv.sc.socket( );
              socket.setReuseAddress(true);

              GlobalsRcv.sc.configureBlocking(true);
              GlobalsRcv.sc.connect(socketAddress);

              // UDP Socket
              GlobalsRcv.sock_udp =
              udpr.getUDPSocket(Common.UDP_RCVPORT,
                          Common.UDP_BUFFER_SZ);

              ConnectionMethodsRcv cmrcv = new
              ConnectionMethodsRcv(GlobalsRcv.sc, this);

              // get recvLatency
              GlobalsRcv.recvLatency =
                    udpr.recvfromLatency(GlobalsRcv.sock_udp,
                    Common.UDP_RCVPORT, Common.MAX_PKTSZ);

              // get SND send latency
              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;
              GlobalsRcv.po.write("[] Min Time Interval : " +
              GlobalsRcv.min_time_interval);
              GlobalsRcv.po.write("[] Send Latency : " +
              GlobalsRcv.sendLatency);
```

```
GlobalsRcv.po.write("[] Recv Latency : " +
GlobalsRcv.recvLatency);
//.............................
GlobalsRcv.max_rate = (Common.MAX_PKTSZ+28) * 8. /
GlobalsRcv.min_time_interval ;
GlobalsRcv.min_rate = (Common.MIN_PKTSZ+28) * 8./
GlobalsRcv.MAX_TIME_INTERVAL ;

// Estimate ADR
GlobalsRcv.adr = cmrcv.getADR();
GlobalsRcv.po.write("[] ADR : " + GlobalsRcv.adr);
if ( GlobalsRcv.bw_resol == 0 ) GlobalsRcv.bw_resol =
.02 * GlobalsRcv.adr ;

if (GlobalsRcv.interrupt_coalescence > 0) {
      GlobalsRcv.bw_resol = .05 * GlobalsRcv.adr;
}
if (GlobalsRcv.adr > 0) GlobalsRcv.tr =
GlobalsRcv.adr;
else GlobalsRcv.tr = 15 * GlobalsRcv.min_rate ;
GlobalsRcv.po.write("[] Max rate, Min rate : " +
GlobalsRcv.max_rate + ", " + GlobalsRcv.min_rate);

/* if ADR couldnot be estimated, then initialize tr =
1 mbps */
if ( GlobalsRcv.tr == 0 || GlobalsRcv.tr >
GlobalsRcv.max_rate )
GlobalsRcv.tr = (GlobalsRcv.max_rate +
GlobalsRcv.min_rate) / 2. ;
else if ( GlobalsRcv.tr < GlobalsRcv.min_rate )
      GlobalsRcv.tr = GlobalsRcv.min_rate ;

/* Estimate the available bandwidth.*/
GlobalsRcv.transmission_rate = (int)(1000000 *
GlobalsRcv.tr);
GlobalsRcv.max_rate_flag = 0 ;
GlobalsRcv.min_rate_flag = 0 ;

int ctr_code;

int fleet_aborted = 0;

double prev_tr;

while (true) {

// if the fleet was aborted resend it with the same
parameters
      // otherwise calculate the new parameters
      if (fleet_aborted == 0) {
          System.out.println("fleet not aborted!!!");
          if ( UtilMethodsRcv.calc_param() == -1 ) {
            //ctr_code = Common.TERMINATE |
         Common.CTR_CODE;
          //cmrcv.send_ctr_msg(ctr_code);
```

```
                    cmrcv.terminate_gracefully(exp_start_times
                    ec, exp_start_timeusec) ;
                  break;
                  }
                }
                System.out.println("-->TRANSM RATE : " +
                GlobalsRcv.transmission_rate);
                // a hack for the cases in which tha rate
                remains constant
                prev_tr = GlobalsRcv.tr;

              cmrcv.send_ctr_msg(GlobalsRcv.transmission_rate);

              cmrcv.send_ctr_msg(GlobalsRcv.cur_pkt_sz) ;

              if ( GlobalsRcv.increase_stream_len > 0 )
                GlobalsRcv.stream_len=3 * Common.STREAM_LEN;
              else
                GlobalsRcv.stream_len = Common.STREAM_LEN;

              cmrcv.send_ctr_msg(GlobalsRcv.stream_len);

              cmrcv.send_ctr_msg(GlobalsRcv.time_interval);

              ctr_code = Common.SEND_FLEET | Common.CTR_CODE ;
              cmrcv.send_ctr_msg(ctr_code);

          ctr_code = cmrcv.recv_ctr_msg();

              if ((((ctr_code & Common.CTR_CODE) >> 31) == -1)
              && ((ctr_code & 0x7fffffff) == Common.RECV_FLEET
              ))
                //GlobalsRcv.po.write("FROM SND --> RECV
              FLEET");
                System.out.println("FROM SND --> RECV FLEET");
              System.out.println("RECEVEING FLEET....");
              // recv fleet
                if (cmrcv.recv_fleet() == -1) {
                      fleet_aborted = 1;
                      // ? ..... is this for the best .... ?
                                                /*if (
                      GlobalsRcv.increase_stream_len == 0 ) {
                      trend = GlobalsRcv.INCREASING;
                      if ( GlobalsRcv.exp_flag == 1 &&
                      prev_trend != 0 && prev_trend != trend)
                            GlobalsRcv.exp_flag = 0;
                      prev_trend = trend;
                      if
                      (UtilMethodsRcv.rate_adjustment(GlobalsRcv
                      .INCREASING) == -1)
                cmrcv.terminate_gracefully(exp_start_timesec,
                exp_start_timeusec);
                      }*/
              }
              else {
                fleet_aborted = 0;
```

```
            UtilMethodsRcv.get_sending_rate() ;
            trend =
            UtilMethodsRcv.aggregate_trend_result();

            if ( (trend == -1) &&
            (GlobalsRcv.bad_fleet_cs > 0) &&
                        (GlobalsRcv.retry_fleet_cnt_cs
            >GlobalsRcv.NUM_RETRY_CS) ) {
      cmrcv.terminate_gracefully(exp_start_timesec,
      exp_start_timeusec) ;
            break;
            }
            else if(( (trend == -1) &&
            (GlobalsRcv.bad_fleet_cs > 0) &&
                        (GlobalsRcv.retry_fleet_cnt_cs
            <= GlobalsRcv.NUM_RETRY_CS) )) /* repeat
            fleet with current rate. */
              continue ;

            if (trend != GlobalsRcv.GREY) {
                  if (GlobalsRcv.exp_flag == 1 &&
            prev_trend != 0 && prev_trend != trend)
                        GlobalsRcv.exp_flag = 0;
                  prev_trend = trend;
            }

            if (UtilMethodsRcv.rate_adjustment(trend)
            == -1) {
      cmrcv.terminate_gracefully(exp_start_timesec,
      exp_start_timeusec);
            break;
            }
    }

      // the hack for the repeating of the rate
      if  (fleet_aborted == 0) {
            if (trend == GlobalsRcv.INCREASING &&
            prev_tr <= GlobalsRcv.tr) {
                  //System.out.println("I : " +
                  prev_tr + "-" + GlobalsRcv.tr);
                  GlobalsRcv.tr -= 5;
            }
            if (trend == GlobalsRcv.NOTREND && prev_tr
            >= GlobalsRcv.tr) {
                  //System.out.println("N");
                  GlobalsRcv.tr += 5;
            }
            if (trend == GlobalsRcv.GREY && prev_trend
            == GlobalsRcv.INCREASING
                        && prev_tr <= GlobalsRcv.tr) {
                  //System.out.println("GI");
                  GlobalsRcv.tr -= 5;
            }
            if (trend == GlobalsRcv.GREY && prev_trend
            == GlobalsRcv.NOTREND
                        && prev_tr >= GlobalsRcv.tr) {
```

```
                                    //System.out.println("GN");
                                    GlobalsRcv.tr += 5;
                              }
                        }
                  }

            } catch (Exception e) {
                  System.out.println("[pathload rcv] : exceptionnn");
                  System.err.println(e);
            } finally {
              udpr.closeUDPSocket(GlobalsRcv.sock_udp);

              if (GlobalsRcv.sc != null) {
                  try {
                        GlobalsRcv.sc.close();
                  } catch (Exception ignored) {
                  }
              }
              GlobalsRcv.po.close();
            }
      }

}
```