Forecaster: Implementation and Evaluation

Anuj Ladkani, Mansoor Aftab, Prasanna Jeevan, and Santosh Munirathna
maftab,adladkan,pdevanu,skmunira@ncsu.edu

***Abstract*—In this Report we propose to show the work we have done in implementing and evaluating the Forecaster Algorithm, on our setup topology. Our goal was to effectively predict by means of probing and calculation the amount of available bandwidth on an end-to-end path and test its convergence in various traffic condition. We address the current limitation with the implementation we have finished and the work required to be done for future work in this area.**

I. Introduction

Bandwidth Estimation is an important field of study in Computer Networks. The Estimation of available bandwidth and maximizing its utilization is very important to modern high bandwidth network applications. Most protocol suites offer their own set of primitives with in-built functionality to estimate the available bandwidth[1]. Many methods exist to measure available bandwidth. Some techniques rely on searching by trail & error[1] while others rely on estimation by calculation[2][3][4] we take up one such technique of the latter type, Forecaster[5].

II. Methodology

The Forecaster algorithm is a method of determining the available bandwidth on an end-to-end path, and also determining the tightest link(most congested link) along the path, together we refer to this as the signature of the link. Forecaster essentially works by sending two streams of probe packets at rates that are much lower than the available bandwidth of the path in question. The key idea is calculate the end to end path utilization through a binary test that measures the fraction of probe packets that experienced queuing. Available bandwidth and speed of the tight link is then calculated with the help of derived link utilization. Once the signature of the link is available, this information can be used by Congestion Control Mechanisms at the transmitter, to make optimal utilization of the link

A. Implementation

I. Estimating End to End Utilization.The key idea of the algorithm is in the approximation of the measure of path utilization and not in the actual measurement of utilization. The Forecaster approzimates the end-to-end path utilization by a simple binary test based on the queuing experienced by the two streams of probe packets in the end-to-end path. The queuing factors of these two probing streams which are pumped into the end-to-end path at different rates are captured by the algorithm and the resulting approximate utilization is derived . This approximate utilization is then used to project the tight link bandwidth in the end-to-end path. The projected bandwidth of the tight link gives the clear idea or an approximation of the available bandwidth in the end-to-end path.

1. Path Utilization ρ(r).

The algorithm aims as explained at determining the path utilization. The first step towards this goal is to measure the path utilization in the end-to-end path between the two host machines. The path utilization factor is represented by ρ(*r*). The factor represents the fraction of packets that experience queuing in the probing stream, which are pumped into the path at different rates. The factor *r* represents the rate at which the packets are sent. The probe packets (hence forth referred to as probes) are sent into the path with exponential inter-departure rates with an average value of *r*. This emulates the PASTA property and according to the PASTA property, the probe packets arriving in the queuing system will sample the system queues, on average, as an outside observer would at an arbitrary point in time.

2. Estimating Bandwidth.

ρ(r) Can now be used to estimate the bandwidth A along the path and also the speed of the tight link. We need two additional factors *c0* and *c1*. These two factors represent the raw utilization of a single link and 1/*c1* represents the link speed.

The key idea of the algorithm is that when rate *r* becomes reached the available bandwidth A, then ρ(r) reaches it bounds i.e. ρ(r) ≈ 1.

$$1 \approx c0 + Ac1.$$

We now compute the values of *c0* and *c1* with the two probing streams. The calculation of *c0* and *c1*

$$\rho(r1) \approx c0 + r1.c1$$

$$\rho(r2) \approx c0 + r2.c1$$

$$c1 = \frac{\rho(r2) - \rho(r1)}{(r2 - r1)}$$

$$c0 = \rho(r1) - r1.c1.$$

The value of *c1* gives a conservative estimate of the tight link speed C. The pseudo code for the forecaster algorithm is listed below.

Algorithm Forecaster:

(1) Measure ρ(r1) and ρ(r2) resulting from two probe sequences, at rates of r_1 and r_2 , respectively

(2) $c1 = (\rho(r2) - \rho(r1))/(r2 - r1)$

(3) $C = 1/c1$

(4) $c0 = \rho(r1) - r1.c1$

(5) $A = (1 - c0)/c1$

return A, C

III. Related Work

Previously work has been done in this area, using a technique called Packet-pair[6][7].Packet pair technique which aims to estimate the capacity of a path (bottleneck bandwidth) from the dispersion of two equal-sized probing packets sent back to back. It has been also argued that the dispersion of longer packet bursts (packet trains) can estimate the available bandwidth of a path. In another piece of work in the area of band width measurement [8], one way delay in the link was used to measure the bandwidth. One way delay of a periodic packet stream show an increasing trend when the stream's rate is higher than the available bandwidth. A tool FEAT[9] uses a similar approach of using probe packets. FEAT features a new dynamic pattern of probes called a Fisheye Stream. One fisheye stream covers a range of packet probing rates. A fisheye stream consists of a focus region, where the probing rates are sampled more frequently and the number of packets used at each rate is larger. This creates a fisheye effect that the focus region enables an easily identified turning point for accurate measurements. When the dynamic available bandwidth is outside the region, the surrounding regions enable the tool to automatically refocus.

IV. Test Setup

We propose to setup up controlled traffic between two Linux end systems running our application/tool/protocol, and also introduce background traffic, these end systems will be separated by 3 hops, and our goal is to time stamp packets and determine using Forecaster algorithm if queuing was experienced by these packets.(See Fig. Below)

V. Detailed Plan of Work

The plan of work is mainly divided into 2 parallel task sets, with one involving the development and implementation of the forecaster code and the other involving setting up of the test bed to execute the forecaster code.

A. Development

i. Pre-development Setup

The initial setup was done on virtual systems. Two Linux (Ubuntu 8.04) virtual machines were configured using the SUN xVM VirtualBox. They are connected to each other by virtual interfaces. Iperf 2.0.2 has been used to send packets between the virtual machines. The setup was used to send probe packets without any cross traffic. One of the virtual machine acted as a client and the other acted as a server. The packet generation was done using Iperf. 400 probe packets were considered and are time stamped at the sender and the receiver. These values will be then processed into source IP, destination IP, IP ID and Time stamp value. Then the processed output from both the sender and receiver are merged, where the time stamp of the sender and receiver with the same IP ID are put together. This is used as input to calculate the slope and the average number of points above the slope. The points above the threshold are the packets that have experienced queuing.

ii. Sender Kernel Module Design

This kernel module will be inserted at the sender. It is capable of time stamping the packets using the netfilter hooks, storing the timestamp in the probe packet's data field and sending the probe packets at different rates. The design of sender kernel module is given below.

1. Packet generation: The kernel module will generate the required probe packets as shown in the figure 1. We are generating 400 probe packets by default and the number of probe packets sent can be easily varied. An UDP client program is implemented and integrated with the sender kernel module. The probe packets are sent from the client to the receiver.

   The probe packets will be sent at two different rates R1 and R2. R1 is of the rate 50 Kbps and R2 is 10 Mbps. These rates can also be easily varied. We have considered the rates R1 and R2 based on the heuristics given in Measuring Bandwidth Signatures of Network Paths, Mradula Neginhal, et al, [5]. The packet size of the UDP probe packets is set to 1428 bytes less than the 1500 bytes MTU.

   After all the probe packets are sent 20 marker packets are also sent. The marker packets are sent to indicate the receiver that all the probe packets have been sent and also to take care of any lost probe packets since the receiver will be waiting for 400 probe packets.

2. Time stamping the probe packets: Before the probe packets are sent they are captured using the netfilter hooks. Then the time stamp is obtained using the Intel's RTDSC (Read Time Stamp Counter) instruction. This time stamp counter provides the high resolution timer that can be accessed from kernel. Then this time stamp value is stored in the data filed of the packet and the packet is sent to receiver.

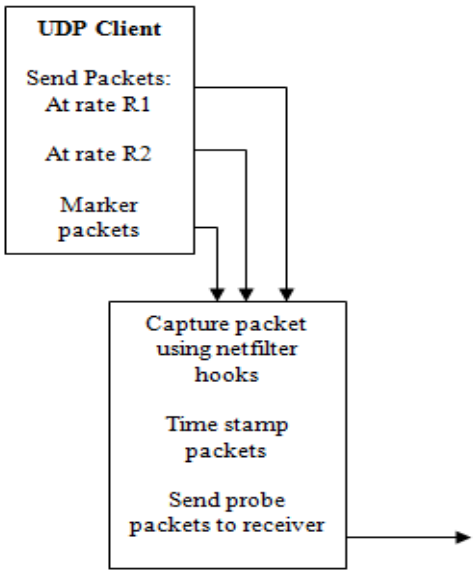

Figure 1: Sender kernel module design

a) Packet Generation Implementation Details

The probe packets are generated using a UDP client built into the packetfilter kernel module. We have used some of the ksocket[10] wrapper functions to access the kernel level socket interfaces. Ksocket is a Linux 2.6 kernel module which provides the kernel developers with bsd style socket interfaces. We are not using the whole ksocket as a different module since we do not need all the socket interfaces, so we have ported only few required socket wrapper functions like ksocket(), ksetsockopt() and ksendto() into our module.

The UDP client generates 400 probe packets each of size 1048 bytes. The size of the packets can be changed later as per our requirements. Before sending the probe packets they are captured using the netfilter hooks. We use the NF_IP_POST_ROUTING hook, which captures the packet just before it hits the wire. Using netfilter, a call back function is registered which will be called when a packet traverses the hook. In the call back function the IP ID of each packet is set to a counter value which is incremented from 0 to 399. This is done because all the UDP packets sent from the UDP client had their IP ID set to zero. Then the new checksum of the modified packet is recalculated using ip_fast_csum() function. After this the time stamp of the probe packet is recorded and the packet is sent to the receiver. This also reduces the delay incurred before the packet is sent out and when the time stamp value is recorded.

The first 200 probe packets are generated as soon as the packetfilter kernel module is inserted at the rate R1. We then wait for five seconds before sending the next 200 probe packets at the rate R2. The waiting in between R1 and R2 rates is to make sure that the network properties are is not affected by the first 200 probe packets.

b) Time Stamping Implementation Details

The time stamping of the packet is done using the RTDSC instruction. The assembly code to get the time stamp is [20]

```
__asm__ volatile (".byte 0x0f, 0x31" : "=A" (tmpstmp)).
```

Where the op-code specifies RTDSC and the time stamp will be stored in the tmpstmp variable.
The time stamp value is also stored in the packet data field and hence the UDP check sum is calculated using the csum_tcpudp_magic() function.
The time stamping will not be accurate if the CPU is multi-core and also the power saving measures taken by operating system which involves voltage frequency scaling of the CPU.
We disabled the multi-core functionality by accessing the BIOS setup. The power saving measures can be disabled by using the following command in Linux systems.

```
sudo sh -c "echo performance > /sys/devices/system/cpu /cpu0/cpufreq/scaling_governor"
```

The frequency of the CPU can be checked by using the below command. The current frequency of the CPU should be equal to its maximum frequency.

```
cat /proc/cpuinfo
```

The CPU ticks can be converted to time in seconds by using the below formula.

Time (in sec) = Time stamp (in CPU ticks) / Frequency of the CPU

iii. Receiver Module Design

The receiver module consists of a receiver kernel module and an application level UDP server. The receiver kernel module captures and time stamps the received probe packets and the application level UDP server calculates the bandwidth based on the packets queued.

a) Receiver Kernel Module Design

The receive kernel module is similar to the sender kernel module except that it does not have the UDP client as shown in figure 2. The receiver kernel module captures the probe packets using the netfilter hooks, obtains the received time stamp value and stores it in the data field of the probe packet and sends the packet to the application level UDP server. The receiver calculates the UDP checksum since the data field of the probe packet is updated using the csum_tcpudp_magic() function.

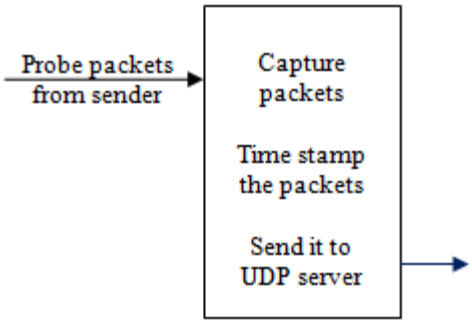

Figure 2: Receiver kernel module design

b) Receiver UDP Server Design

This consists of application level UDP server which accepts the probe packets sent from the sender as shown in figure 3. The time stamp at the sender and receiver are extracted from the data field of the probe packet and stored by the UDP server. This is done for all the probe packets until it receives the marker packets. Once the marker packets it does the calculation to identify the queued packets which are fed into forecaster algorithm to obtain the available bandwidth and tight link speed.

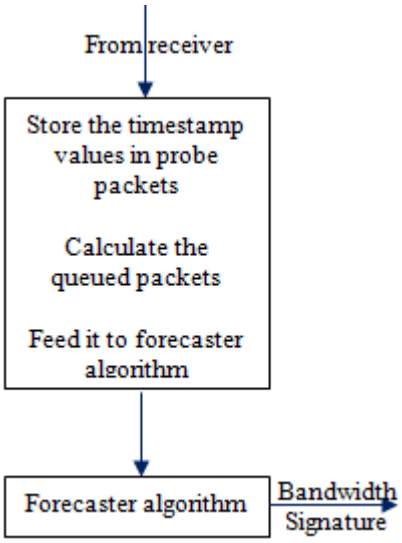

Figure 3: Receiver UDP server design

c) Identifying the queued packets

Firstly the obtained time stamp at the receiver will be used as x-axis values and the difference between the time stamp value at the sender and receiver is used as the y-axis values. The goal is to identify a band and the packets which fall above the band are considered to have experienced queuing. This is done using the following steps.

For each 200 probe packets of the same rate the first 30 packets and the last 30 packets are considered. Then they can be assumed as plotted on a graph as specified above. Then the slope between each of the point is calculated using the slope formula

m = (y2-y1)/(x2-x1)

Then the averages of all these slopes are taken.

This value is used to identify the y intercepts of each of the first and last 30 points using the formula

c = y-mx

These y-intercept values are stored and sorted. Then the 95th percentile value is taken as the band value.

Then the y-intercept for each of the remaining 170 points in the middle will be calculated and compared to the band value. If the y-intercept value is greater than the band value then the packet is considered to have experienced queuing. This is fed into forecaster algorithm to obtain the bandwidth signature.

B. Test & Execution

To start with the setup of the test bed we read the commands for CLI of Cisco Routers. We used the Terminal Server (Cisco 2800) which was given to us to setup the other 3 routers (Cisco 3825). The 3825s had 2 GigE links and 4 L2 ports. These L2 ports by default belong to VLAN 1. We gave this VLAN 1 IP address and binded a L2 port with it. After giving IPs to the L2 port and the two GigE interfaces we configured OSPF routing protocol. We repeated this on all three of the routers. The final setup is shown in the figure at the end of the report. Our Forecaster kernel module was installed on two laptops running Ubuntu 8.10 with 100Mbps NICs on them. And 3 desktops were used to generate cross traffic and each had a GigE card and OS was Ubuntu 8.10.

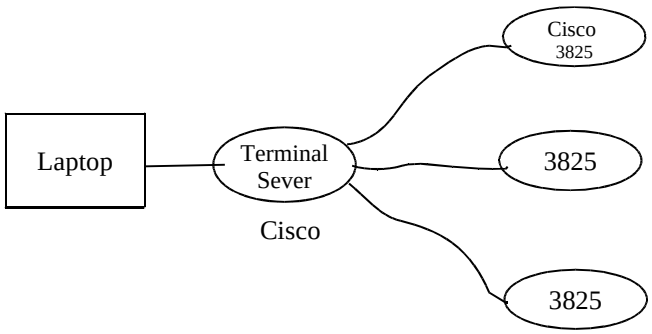

Figure 4

After the topology setup our aim was to get enough traffic to get queuing on the router ports. This requires the traffic generators to pump data at maximum rate. As mentioned we had Desktops with GigE link, but that doesn't mean each desktop will contribute to 1 Gbps of traffic. Commodity OSes are not traffic generating software plus there are other reasons too which can make these NICs pump traffic at a rate way below their pumping capacity. We researched on these reasons and tried to get maximum throughput from these NICS.

1. Network cards are normally connected to the PCI bus bus via a free PCI slot. In older workstation and non server-class motherboards the PCI slots are normally 32 bit, 33MHz. This means they can transfer at speeds of 133MB/s. Since the bus is shared between many parts of the computer, it's realistically limited to around 80MB/s in the best case. Gigabit

network cards provide speeds of 1000Mb/s, or 125MB/s. If the PCI bus is only capable of 80MB/s this is a major limiting factor for gigabit network cards. The math works out to 640Mb/s, which is really quite a bit faster than most gigabit network card installations, but remembers this is probably the best-case scenario. Many people recommend setting the MTU of your network interface larger. This basically means telling the network card to send a larger Ethernet frame. While this may be useful when connecting two hosts directly together, it becomes less useful when connecting through a switch that doesn't support larger MTUs. At any rate, this isn't necessary. 900Mb/s can be attained at the normal 1500 byte MTU setting..

2. For attaining maximum throughput, the most important options involve TCP window sizes. The TCP window controls the flow of data, and is negotiated during the start of a TCP connection. Using too small of a size will result in slowness, since TCP can only use the smaller of the two end system's capabilities.
3. We used sysctl utility of Linux to change the following parameters, net.core.wmem_max (max window size), net.core.wmem_default, net.core.rmem_default (default window size), net.ipv4.tcp_window_scaling (window scaling according to rfc 1323).
4. After we changed these values the data rate we could get from the GigEs went upto 800Mbps. To confirm this we used nttcp. The nttcp program measures the transferrate (and other numbers) on a TCP, UDP or UDP multicast connection.

We then used Iperf to generate UDP packets. We used this because Iperf in UDP mode allows changing the bandwidth of traffic generated. We also used netem, it provides Network Emulation functionality for testing protocols by emulating the properties of wide area networks. The current version emulates variable delay, loss, duplication and re-ordering. We used it to get some drop rate at the sender side to make some packet loss forcefully, we did few experiments like this to see if the bandwidth calculated by forecaster changes in proportion with the change in loss. We didn't add any loss after these initial experiments. For rest of the readings, we generated Iperf cross traffic and ran our kernel module on two laptops to measure the bandwidth from client to server.

VI. Results

Our implementation was tested on the setup shown in Figure 4. We used two streams of UDP packets each of 1428 bytes send at two different rates R1 and R2. R1 had 200 probe packets sent at 50 Kbps and R2 had 200 probe packets sent at 10 Mbps. The Rate of R2 is selected to be one tenth of the capacity of the sender link. Then we plot each of the packets with time at receiver as the X axis and the time difference between the sender and receiver time stamps in Y axis.

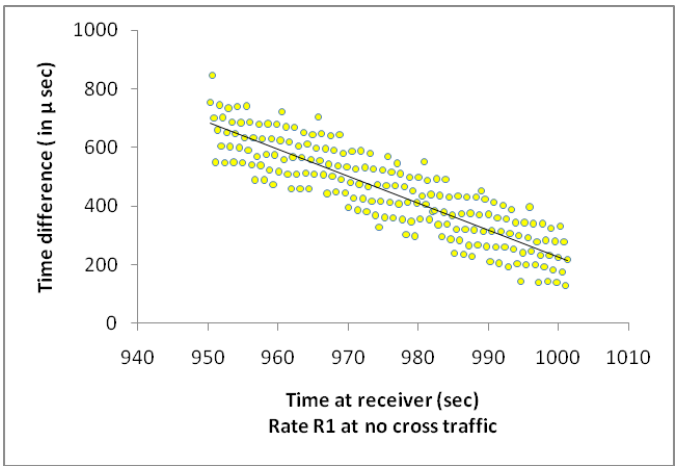

Figure 6(a)

Figure 6(a) shows the obtained graph with rate R1 and no cross traffic. In this the number of probe packets queued is zero.

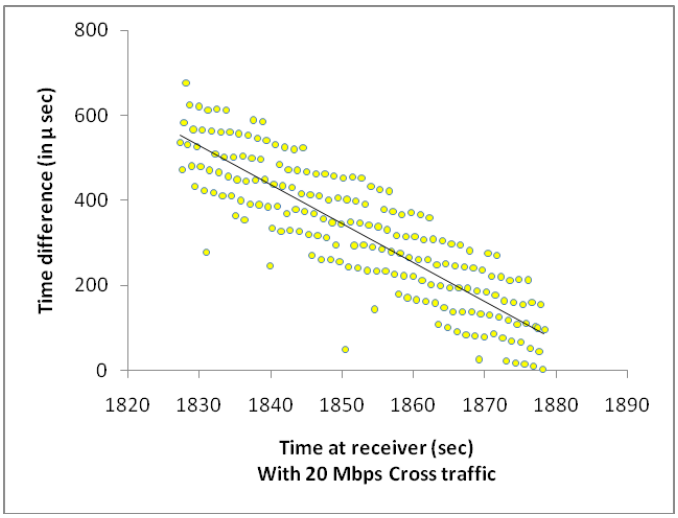

Figure 7(a)

Figure 7(a) shows the obtained graph with rate R1 and 20 Mbps cross traffic. In this the number of probe packets queued is zero.

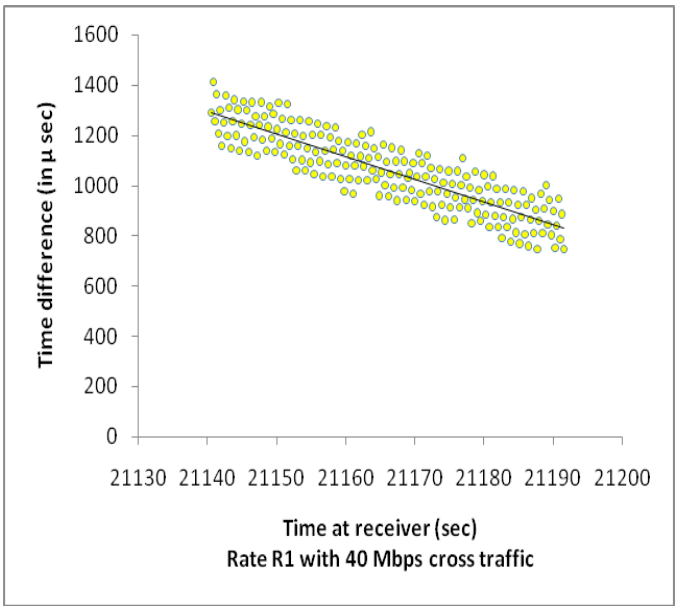

Figure 8(a)

Figure 8(a) shows the obtained graph with rate R1 and 40 Mbps cross traffic. In this the number of probe packets queued is zero.

Reasoning for Slope in the above shown graphs.

As mentioned before we are using the RTDSC instruction. This instruction returns a 64-bit value incremented every ticks, this presents probably the most granular method to calculate time, since we need to convert the ticks value to seconds we need to divide the value of ticks with number of ticks per second, since the system information only reports and approximate rounded figure this presents us with a problem, We illustrate it with the following example, here a we use 1.5 ticks as well, in reality there is no 1.5 tick but this is used to merely illustrate the situation when the clock speed is mhz, and the effect of points after the decimal place.

Assume the Sender has a clock frequency of 1 tick per second, and the receiver has a clock frequency of 1.5 clock ticks per second, but the system information rounds of this to 1 tick per second for the receiver, also assume that the sender sends packets every 2 seconds, and the initial tick count in sender is 9 and in receiver is 13.5. The timer difference calculation will work out as follows-

Sender Sends a packet at tick 9, time is tick/rate therfore timestamp is 9 seconds. Assume packet takes 3 seconds to reach the receiver, the receiver receives at tick 18, time is tick/ rate therefore we consider time as 18, therefore calculated propagation time is 9.

Sender again sends a packet at tick 11, time is tick/rate therefore timestamp is 11 seconds, packet takes 3 seconds to reach the reciver, the reciver receives at tick 22.5, time is tick/rate therefore we consider time as 22, therefore calculate propagation time is 11.

We have shown in 2 steps an increasing value for interpacket time, this error will continue to build up and will present us a slop when the interpacket times are plotted.

VII. Issues Faced

1. Floating point operation at Kernel Level

The kernel does not support floating point operations for efficiency reasons. So we could not do the calculation part which involved calculating the slope and also the forecaster algorithm in kernel module. We have moved these functionality to an application module. There are other ways to do floating point operations such as using fixed point arithmetic[21].

References

[1] TCP Congestion Control, http://www.ietf.org/rfc/rfc2581.txt

[2] V. Jacobson. pathchar - A Tool to Infer Characteristics of Internet Paths, 1997. ftp://ee.lbl.gov/pathchar

[3] A. B. Downey. clink: A tool for estimating internet link characteristics, 1999. rocky.wellesley.edu/downey/clink/

[4] B. A. Mah. pchar: A Tool for Measuring Internet Path Characteristics, 2001

[5] Measuring Bandwidth Signatures of Network Paths, Mradula Neginhal, Khaled Harfoush, and Harry Perros

[6] Techniques and a Capacity Estimation Methodology: C. Dovrolis, P. Ramanathan, and D. Moore.

[7] Evaluation and characterization of available bandwidth probing techniques: Ningning Hu; Steenkiste, P.

[8] M. Jain and C. Dovrolis. End-to-End Available Bandwidth: Measurement Methodology, Dynamics, and Relation with TCP Throughput.

[9] FEAT: Improving Accuracy in End-to-end Available Bandwidth Measurement, Qiang Wang and Liang Cheng, Laboratory Of Networking Group

[10] http://ksocket.sourceforge.net/

[11] http://www.protocoltesting.com/trgen.html

[12] http://kb.pert.geant2.net/PERTKB/AdvancedGuide

[13] http://dpnm.postech.ac.kr/research/01/ipqos/

[14] http://www.protocoltesting.com/trgen.html

[15] http://kb.pert.geant2.net/PERTKB/AdvancedGuide

[16] http://www.pcausa.com/Utilities/pcattcp.htm

[17] http://rude.sourceforge.net/

[18] http://www.ubuntugeek.com/bandwidth-monitoring-tools-for-ubuntu-users.html

[19] http://www.enterprisenetworkingplanet.com/nethub/article.php/3485486

[20] http://www.mcs.anl.gov/~kazutomo/rdtsc.html

[21] http://www.embedded.com/98/9804fe2.htm

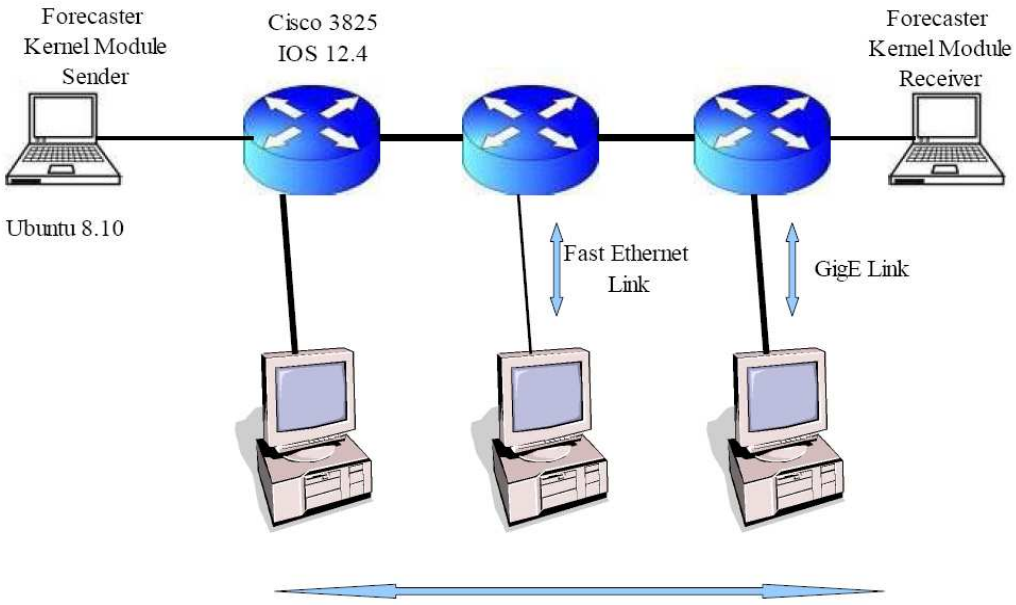
Forecaster
Kernel Module
Sender
Cisco 3825
IOS 12.4
Forecaster
Kernel Module
Receiver
Ubuntu 8.10
Fast Ethernet
Link
GigE Link
Traffic generators (Each desktop with GigE NIC)

Figure 9