Implementation of an Available Bit Rate Service for Satellite IP Networks using a Performance Enhancing Proxy

by

Pavan K. Reddy

A Thesis

Submitted to the Faculty

of the

WORCESTER POLYTECHNIC INSTITUTE

In partial fulfillment of the requirements for the

Degree of Master of Science

in

Electrical and Computer Engineering

by

May 2004

APPROVED:

_____

Professor David Cyganski, Advisor

_____

Professor Brian King

_____

Professor William Michalson

Abstract

The transport control protocol (TCP) is one of the most heavily used protocols on the Internet, offering a reliable, connection oriented transport service. However, the quality of service (QoS) provided by the TCP protocol deteriorates when it is used over satellite IP networks. With the increased usage of Internet applications by the military in remote geographical regions, there is an increased need to address some of the shortcomings of TCP performance in satellite IP networks. In this research we describe our efforts at designing and testing a performance enhancing proxy (PEP) that can be used improve the QoS provided by the TCP service in large latency networks. We also show how one can use such a proxy to create a new transport service similar to the Available Bit Rate (ABR) service provided by ATM networks without needing ATM infrastructure, this new service offers a connection oriented, reliable, best effort transport service with minimal queuing delay, jitter and throughput variation.

Acknowledgements

I would like to express my gratitude to my advisor Prof. David Cyganski for helping me for the last year in this research and, without whose insight this research would not have been done. I am also grateful to the members of my thesis committee Prof. Brian King and Prof. William Michalson for their thoughtful suggestions that have helped improve this thesis. I would also like to thank the Network Centric Systems division of the Raytheon corporation for sponsoring the research that lead to this thesis.

I would also like to thank all the members of the Electrical Engineering Department's Machine Vision Laboratory during my stay there Dan Breen, Nicholas Hatch, David Holl and Ben Woodacre for providing a friendly, helpful and lively environment to work in. I would also like to appreciate my roommates Jens-Peter Kaps and Gunnar Gaubatz for the encouragement and help they have provided over the last few years. Finally and most importantly, I want to thank my parents from whom I have learnt so much about the need for dedication and perseverance and who have always supported me in pursuing my dreams.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The Transport Control Protocol (TCP) is one of the most popular Internet protocols in use today. The design of TCP has evolved significantly from its original specification. One of the major modifications made to the TCP design was the inclusion of a congestion control scheme. However, as a previous study has shown [Hol03], TCP is incapable of meeting the quality of service (QoS) requirements of certain applications that require low jitter. This is attributed to the behavior of the current congestion control mechanism used in most TCP stack implementations. This shortcoming is more evident when TCP is used in networks with large end to end latency such as satellite IP networks. None of the existing techniques used to provision QoS in IP networks are adequate in improving the QoS of TCP traffic in satellite IP networks.

In this thesis, we provide a detailed account of our effort at designing, implementing and testing a performance enhancing proxy that can be used to create an Available Bit Rate (ABR) service from ordinary TCP flows, without requiring any modifications to existing TCP stacks. We also show that when such a Performance Enhancing Proxy is used in a satellite IP network to offer IP-ABR service, the resulting IP-ABR service can provide a considerable improvement in the quality of service than that is offered by TCP.

This service will allow applications to take advantage of TCP's reliable transport service without having to compromise QoS.

In Chapter 2 of this report we start by providing some background on the TCP protocol and examine the QoS of TCP traffic in large latency networks, we also look at the effect TCP traffic has on the QoS of non-TCP traffic. In Chapter 3, we propose the creation of an IP-ABR service using performance enhancing proxies to improve the QoS of TCP traffic. In Chapter 4 we provide further details on the various mechanism developed to create an IP-ABR PEP. In Chapter 5 we provide an account of various tests simulated on an early IP-ABR PEP model developed in the NS simulation environment. We also provide an account of some of the changes that were made to the IP-ABR PEP to improve its performance. Chapter 6 provides details on the implementation of a prototype IP-ABR PEP for the LINUX operating system. In Chapter 7, we provide details on the various laboratory tests conducted on the IP-ABR PEP implementation in order to evaluate its performance.

Chapter 2

Background

Despite the growth of terrestrial wired and wireless networks, satellite based communication is still commonly used to communicate to many geographically isolated regions. A satellite with an orbital altitude of around 36,000 Km above the equator will maintain a stationary position relative to any point on the earth, because its orbital period matches that of the earth [Tan96], such satellites are referred to as geostationary satellites. Most satellites used for communication are geostationary earth orbit (GEO) satellites. These GEO satellites provide very wide coverage, however, they also have long delay characteristics, attributed to the vast distances that a radio signal needs to travel to and from a geostationary satellite.

Using the Internet Protocol (IP) over satellite links allows network designers to seamlessly interconnect remote subnetworks to each other and to the Internet. The Internet protocol (IP) is a connection-less datagram protocol that facilitates data transportation across heterogeneous subnetworks. The IP protocol essentially allows the creation of the Internet. IP is also the network layer or layer 3 in the 7 layer OSI hierarchy model used to describe computer network protocol stacks [Tan96]. IP is an important protocol in data communications, due to the increasing popularity of Internet applications. However, most

Internet applications rarely use the IP protocol alone, instead they use so called transport protocols. The Transmission Control Protocol (TCP) is the most commonly used transport protocol.

In this chapter we shall provide a brief overview of the Transmission Control Protocol and highlight some of the modifications made to the TCP design since its inception. We shall also detail the shortcomings of TCP in meeting the quality of service demands of certain applications in large latency networks such as satellite IP networks.

2.1 TCP

The TCP protocol was designed to provide a reliable, connection oriented, best effort, transport service over an IP network. It was initially designed as part of the ARPANET project. The initial specification of the Transmission Control Protocol was published in 1980 as RFC 793.

2.1.1 TCP Evolution

The University of California, Berkeley implemented one of the first TCP stacks in its version of the UNIX operating system. This early version of UNIX is commonly referred to as the Berkeley Software Distribution (BSD) [MKMQ96]. BSD UNIX and the TCP/IP stack implementation were very popular and usage proliferated. Over the years the BSD group was also the first to implement recommended modifications to TCP, hence most UNIX TCP/IP stack implementations were based on the BSD implementation. Table 2.1 details the various modifications made to TCP over the years and the BSD UNIX version that they first appeared in.

Feature	First Implementation	Year
First widely available implementation of TCP/IP	4.2 BSD	1983
Congestion control mechanism (RFC 1222 compliance)	4.3 BSD Tahoe (1988) or Net/1	1989
Fast recovery	4.3 BSD Reno (1990) or Net/2	1991
Long fat pipe (Window scaling)	4.4 BSD Net/3	1993

Table 2.1: TCP versions

2.1.2 TCP Mechanisms

TCP stacks transmit data in blocks, each of these blocks of data is referred to as a "segments". Segments transmitted are numbered sequentially in terms of Octets, this numbering scheme facilitates reordering of data at the receiver, in case they arrive out of order. Data packets received at the receiver/destination are acknowledged by sending an acknowledgement back to the sender/source. The acknowledgement (ACK) may either be a TCP frame with no payload or may piggy back on a existing TCP frame with payload that is bound to the sender.

Unlike the Internet protocol (IP), TCP guarantees reliable transport of data between end hosts. In order, to create a reliable protocol over an unreliable connection-less protocol such as IP, TCP uses an Automatic Repeat Request (ARQ) mechanism, which allows it recover from packet loss or corruption.

TCP is a best effort service, this means that TCP stacks self regulate their flow rate in a manner such that they can achieve the maximum throughput between two end hosts. TCP uses a sliding window mechanism for self-regulated flow control. According to the initial design of TCP described in RFC 793 the end hosts establish an initial connection by participating in a three way handshake, after which the TCP sender sends a complete window of data, then enters self-pacing mode, wherein further segments are transmitted only after receiving acknowledgements for previously transmitted ones [Pos81]. As we shall see in

the following sections, this flow control mechanism has been changed significantly from the original design specified in RFC 793.

Using TCP over satellite networks poses unique problems due to the long delay characteristics of such networks. One of the earlier problems noticed when using TCP over such satellites networks is that of link under-utilization. It was realized that the default window size used in most TCP implementation was not large enough to scale to TCP throughput when used over links with large bandwidths and delay, such as satellite links. In order avoid underutilization of satellite links it was proposed that the window sizes be enlarged, by using a scaling factor [VJB92]. As stated in RFC 1323 a window scaling feature was subsequently introduced, which can allow stacks to scale their window by up to factor of 2^{14} when transmitting across networks with large bandwidth delay products.

2.2 TCP Congestion Control

The original specification for TCP used a window based flow control mechanism where only the receiver could regulate the flow, thereby preventing the sender from overflowing the receiver's buffer. However, in this early design the sender did not take into account network congestion. This in turn lead to "congestion collapses" [Nag84] on the Internet. Proper congestion control, that takes into account congestion experienced in the networks, is essential to implement a best effort transport service in a packet network.

2.2.1 Detecting Congestion

In order to create a congestion control mechanism it is essential that the end hosts be capable of detecting congestion within the network. There are a couple of ways by which the end hosts can detect congestion within the network [Sta02]:

1. Explicit Feedback Signaling: In this technique, the end host is notified of conges-

tion by the network, typically an intermediate router experiencing congestion on its queue. Once congestion is detected the end hosts explicitly notified of it by some signaling scheme. The disadvantage of this method is that in order to be effective it requires all routers to participate in this technique.

2. Implicit Feedback Signaling: Congestion is detected by the end host without any help from the network. Detection of packet loss is one means by which this can be achieved. The advantage of this method is that it does not require the routers to support congestion monitoring, since they are not used. However, a disadvantage of using congestion events such as packet drops is that the end hosts react to congestion only after it occurs.

2.2.2 Current TCP Congestion Avoidance Mechanism

In order to prevent TCP from causing congestion collapses on the Internet, in 1988, Van Jacobson proposed modifications to the TCP design [JK88]. These proposals were later made a requirement in RFC 1122 after it was proven that they are successful in preventing congestion collapses occurring when TCP is used. In that proposal the existing ARQ mechanisms and the flow control mechanism were redesigned so that, TCP end hosts can alleviate congestion in the network upon detecting a congestion event in the form of packet loss.

The original TCP design specified in RFC 793 used an ARQ mechanism, in which a TCP sender retransmits a segment if it encounters a timeout of it's retransmission timer (RTO) which signifies a lost segment. However, retransmitted packets are more likely to encounter congestion, since the sender does not provide sufficient time for the congestion to clear. This in turn increases the probability of an RTO timeout unless the RTO estimate is increased. It was proposed that the sender use a backoff process during segment retrans-

mission, thereby providing the network with time to decongest itself. In order to achieve this, an exponential backoff mechanism was proposed, wherein, the sender multiplies the RTO by a constant value for each retransmission[Sta02].

In addition to modifying the ARQ mechanism, other modifications were made to the flow control mechanism. In particular four of these techniques are used in most TCP stacks that are compliant with RFC 1122[Bra89]:

1. Slow start
2. Dynamic Window Sizing
3. Fast retransmit
4. Fast recovery

The dynamic window sizing technique allows the transmit window size to vary dynamically in response to congestion, thereby allowing the window to be shrunk on detecting congestion and enlarged afterwards. The “slow-start” technique was proposed to prevent the sender from causing congestion while ramping up the window either at the connection start or after congestion backoff. The slow-start technique grows the TCP transmit window size exponentially. These techniques were effective in preventing congestion collapses, however as we shall see in the next section, these same techniques also degrade the QoS provided to applications.

2.3 TCP QoS and Fairness

As mentioned previously, the lack of congestion control mechanisms in the initial version of TCP lead to congestion collapses on the Internet. In order to prevent congestion collapses, congestion control techniques were initially proposed by Van Jacobson and others and were later incorporated into various TCP implementations [Ste97a][JK88]. This

congestion control mechanism allows end hosts to self regulate themselves, thereby preventing congestion collapses from occurring. They do so by reducing throughput upon detecting congestion. To detect congestion they make use of implicit feedback from congestion events such as packet loss. A key feature of this mechanism is that of dynamic window sizing. Adjustment of the window size indirectly causes the TCP stacks to throttle back throughput in response to congestion events.

The effectiveness of this scheme in preventing congestion collapses has resulted in its wide spread adoption in various TCP stack implementations. However, despite its success it has a few important shortcomings which are becoming increasingly evident. More and more Internet applications that are in use today have specific quality of service (QoS) needs. In other words, these applications have specific network requirements in terms of throughput, delay, jitter and packet-loss. The current congestion control mechanism in TCP cannot guarantee to meet the QoS needs of applications.

In addition to QoS related problems, the existing mechanism is built upon a greedy algorithm, wherein TCP hosts compete against one another for bandwidth. Now, with the widespread adoption of TCP, it is increasingly common to have multiple TCP flows sharing a link. This in turn leads to a lack of fair usage of bandwidth, since the current algorithm cannot guarantee fairness [Flo00].

2.3.1 TCP Congestion Control Mechanism's effect on QoS

According to the initial scheme (RFC 793) used in TCP, throughput should rapidly ramp up and then enter a steady-state mode in which the source end host transmits a new segment only upon receiving an acknowledgement for a previously transmitted segment [Pos81]. In this steady-state period, sender transmissions are driven by ACK arrivals from the receiver, which in turn are driven by segment transmission from the sender. This results in a "self-pacing" behavior between sender and receiver, which is also sometimes

called “ACK-clocking”.

However, ramping the window size up, too fast, at the start of a TCP session may lead to congestion. In order to avoid a congestion collapse due to rapid expansion of the transmit window a new mechanism called “slow start” was proposed. The slow start mechanism governs TCP throughput the ramping stage from the initial moments of a TCP session until it enters self-pacing mode. Slow start involves exponentially ramping up the transmit window size, and of the throughput until congestion is detected in the form of packet loss. Upon detecting congestion, TCP sets a slow start threshold to half of the current window size or the congestion window size. It then backs off by reducing the window size to 1 and enters slow start mode again. In slow start mode it ramps up the throughput until it reaches the slow start threshold, which was previously set to half the congestion window size. After reaching the slow start threshold, TCP breaks out of slow start mode and resumes window growth at a linear pace.

During self-pacing mode the throughput is proportional to the transmit window size. If this window is non variable the throughput should be approximately constant. However, in the current mechanism the window size varies because a dynamic window is used for congestion control. We can summarize the dynamic window sizing behavior with the following rules [Sta02].

1. If congestion is detected set slow start threshold to half the current window size. Then back off by reducing window size to 1.

2. After backing off, TCP resumes window growth as in slow-start mode, until the window reaches the slow start threshold, unless we encounter congestion again.

3. If the slow start threshold is reached, stop exponential growth and continue to expand the window linearly until congestion is detected again.

As can be seen, a TCP transmitter constantly searches for an upper bound on bandwidth by linearly incrementing its transmission window. This leads to an increase in throughput until the stack detects congestion again. Upon detecting congestion a back-off is triggered, followed by slow-start. These repeated swings in the window size lead to oscillations in the achieved throughput. Also, at the peak of each oscillation packets are lost due to congestion.

This expansionist behavior associated with TCP's dynamic window sizing algorithm is necessary, since TCP is a best-effort service. The best-effort nature requires that stacks be greedy, since there is no explicit means by which they can estimate available bandwidth. However, this behavior also leads to highly variable throughput and packet retransmissions. This is especially noticeable in networks with large bandwidth delay products, since they have larger window sizes resulting in wider swings in the throughput. This variability also affects other QoS parameters such latency and jitter, because of the periodic filling and overflow of router queues. Another significant problem is that any non-TCP traffic sharing a queue will also be affected by TCP's behavior and likewise display degraded QoS.

2.3.2 TCP Fairness

The TCP flow-control mechanism exhibits a greedy behavior when it comes to bandwidth usage. In situations wherein multiple TCP flows share a link, TCP flows compete against each other for bandwidth. This creates a situation wherein fair-sharing of bandwidth cannot be guaranteed. This might be acceptable for certain applications. However, there are an increasing number of applications that have specific throughput requirements. For instance video and audio streaming applications have strict throughput requirements.

2.4 QoS tools for IP networks

2.4.1 Random Early Detection (RED)

TCP utilizes packet loss as an implicit means of determining the maximum available bandwidth. Hence it frequently creates congestion as a means of probing. Most routers drop no packets untill the queue is full, then they drop "all" incoming packets, such a queuing discipline is referred to as drop-tail mechanism. Each packet dropped by the router effectively signals an end host to reduce its throughput. In the drop-tail scheme the router will drop any packet that cannot be placed in the queue. These drops, however, may be disproportionate to what is needed to reduce congestion. Now, all those connections whose packets have been dropped may reduce their throughput and ramp up again simultaneously. The resulting surge in traffic may cause these connections to begin the next cycle of slow-start, thereby causing previously unaffected connections to join in the process. This reduction in throughput results in under utilization of link. This phenomena is called "global synchronization".

In order to prevent the global synchronization phenomena, a new queuing discipline called Random Early Detection (RED) was developed [FJ93]. Unlike the drop-tail case wherein routers simply react to congestion, with RED, routers are proactive about congestion management. In others words, when using RED, the router will drop packets before severe congestion is created, as a means of reducing congestion. The operation of RED is based on the fact that a router's queue growth can be used as an indication of future congestion. The RED technique involves first defining minimum and maximum queue thresholds. In operation, packets are randomly dropped with increasing probability starting upon the router queue size exceeding the minimum threshold until it reaches the maximum threshold, at which point all packets are dropped.

RED is effective in preventing global synchronization. However, it is not effective

in solving any of the problems associated with TCP throughput variability and fairness. Since RED is purely a congestion management technique for TCP, it does not strive to improve the QoS of TCP.

2.4.2 Class Based Queuing

Class Based Queueing (CBQ) is a queuing discipline. CBQ enabled routers separate traffic into different classes based on their QoS needs. Each class is provided with a separate queue and each queue is processed in an order and fashion that depends on the priority given to the class. In essence a CBQ enabled router provides differential service to the various traffic classes passing through the router [1].

Class Based Queueing (CBQ) is a popular tool used in networks which have mixed traffic with different QoS requirements. However, CBQ is not effective in actually changing the behavior of TCP itself. The problems associated with TCP's behavior require a solution that directly addresses the throughput variability exhibited by the current congestion control mechanisms. However, there are constraints to any possible solution. Due to the wide spread deployment of TCP technology, any solution requiring modification of existing TCP stack implementations will be infeasible, which means any new solution will have to be backward compatible with legacy TCP implementations. So we can use CBQ to meet the higher QoS needs of non-TCP traffic in shared networks.

2.5 IP service classes

As we have mentioned previously, IP is a connectionless, unreliable datagram service. By using CBQ in IP networks bandwidth can be guaranteed to the end user. Therefore by using UDP and CBQ one can easily implement a Constant Bit Rate (CBR) service for

[1] In order enjoy the benefits of CBQ, policies should be shared among all routers in a management domain.

IP networks. By segregating CBR traffic from non-CBR traffic and giving it the highest priority, the network can guarantee fixed bandwidth for the duration of the CBR flow, providing nearly zero packet loss due to congestion, low jitter and delay, hence providing high QoS. However, since IP-CBR requires using UDP, reliable transport service similar to TCP cannot be provided. IP-CBR will only be appropriate for applications that have strict constant bit rate requirements without strict reliability requirements such as for digital voice traffic.

TCP is reliable and connection oriented, but offers little QoS especially in networks with large bandwidth delay products. Regular best-effort TCP service does not guarantee bandwidth, latency or jitter, which makes it similar to the Unspecified Bit Rate (UBR) service provided in ATM. In a previous study it was shown how CBQ can be used to segregate traffic with TCP from constant bit rate traffic [Hol03]. Applications such as e-mail and FTP which only require reliable transport can use such a service. In addition to CBR and UBR, there exist two more service classes for ATM; Variable Bit Rate (VBR) and Available Bit Rate (ABR). In the following chapter we shall also look at how an ABR service for IP can be used to improve TCP quality of service in satellite networks.

Chapter 3

IP-ABR Concept

As we have seen in the previous chapter, when TCP is used over satellite IP networks the QoS provided by it is degraded due to the behavior of TCP's congestion control mechanism in large latency networks. We have also seen how certain tools such as class based queuing (CBQ) can be used to improve the QoS of non-TCP traffic. However, there are no existing tools or techniques that can be used for applications that require the connection oriented, reliable service and also require better QoS than that offered by TCP. In this chapter, we propose a new service that will offer a connection oriented, reliable transport service, with better QoS than that offered by TCP. We shall also show how one can create such a service by modifying the behavior of existing TCP service using a Performance Enhancing Proxy (PEP).

3.1 IP-ABR service

3.1.1 ATM's ABR service

The Available Bit Rate (ABR) service is an ATM service for applications which have both flexible bandwidth needs and also require better QoS than that provided by services

such as ATM's Unspecified Bit Rate (UBR) service. In an ATM network an application using the ABR service at the time of connection setup will specify a minimum cell rate (MCR) that it requires and a peak cell rate (PCR) that it can use. Upon receiving a request, bandwidth is allocated by the network in such a manner that each ABR source will receive at least the minimum requested cell rate. Later on if more bandwidth becomes available (due to the reduced needs of higher priority traffic such as constant bit rate traffic) it will be redistributed to the ABR sources. If any extra bandwidth exists after meeting the peak cell rates (PCR) of all of the ABR sources, it is reallocated to the UBR traffic [Sta02]. Since ABR service regulates bandwidth usage by explicitly allocating bandwidth to ABR sources it reduces the congestion normally associated with bursty traffic. This allows the ABR service to provide better QoS to the application, while still allowing applications to take advantage of excess bandwidth as it becomes available.

ABR and TCP service have a similarity since both allow end hosts to dynamically adjust to available bandwidth. However, the congestion control scheme of these two transport services differ significantly. ABR service is proactive in congestion mitigation: bandwidth is allocated in a manner such that congestion is minimized, whereas, TCP's congestion control scheme is reactive: congestion reduction measures kick in after a congestion event such as packet loss occurs. Therefore, for bursty traffic type applications ABR is an ideal service. The disadvantages of ABR is that it requires the usage of ATM, which in many cases is not feasible from an operations point of view.

3.1.2 ABR for IP networks

We propose to create a new service for IP networks that will provide a service that is similar to ATM's ABR, we shall refer to this new service as IP-ABR in order to distinguish it from ATM's ABR. Like ATM's ABR service IP-ABR will also provide higher QoS than that offered by regular TCP for applications with flexible bandwidth needs. To create this

new IP-ABR service we intend to use the existing TCP protocol, instead of defining a new transport protocol. In order to create an IP-ABR service out of ordinary TCP service, we borrow from ATM's ABR the idea of dynamic resource allocation.

Just as in ATM's ABR in IP-ABR we will require a service that can dynamically determine bandwidth available to the IP-ABR traffic given the requirements of higher priority traffic. We shall refer to this service as the Bandwidth Management Service (BMS). The BMS can be a service built into either the routers themselves or be instantiated as a separate standalone application. The BMS can create an IP-ABR service by dynamically determining the available bandwidth and then redistributing the available bandwidth to end hosts in such a manner that each flow is allocated a bandwidth that will meet the end host's minimum requirement. However, since these flows are TCP flows, their throughput must be regulated in a manner such that they do not exceed the bandwidths that were dynamically allocated by the BMS. In order to facilitate bandwidth allocation to end hosts, we propose to use a Performance Enhancing Proxy (PEP) which can regulate TCP flows by window limiting them. The resulting window-limited TCP flows should mitigate congestion and thereby induce the TCP flows into a steady self-pacing mode. The resulting IP-ABR service can provide the application with the following:

1. Reliable transport service: Because it uses TCP's ARQ mechanism.

2. Connection oriented: Uses TCP existing connection scheme.

3. Best Effort bandwidth usage: Variable Bandwidth allocation by BMS.

4. Reduced throughput variation compared to regular TCP service: Due to better congestion control done by the PEP.

5. Better QoS (i.e., lower delays, jitter compared to regular TCP service): This also due to better congestion control.

3.2 Performance Enhancing Proxies

In order to create an IP-ABR service as we have proposed, we will require a performance enhancing proxy which is capable of regulating TCP throughput to a specified bandwidth. In this section we explore in further detail how one can design a Performance Enhancing Proxy to regulate TCP flows.

3.2.1 Window Limited TCP

As a previous study [Hol03] has shown, an optimal window size can be estimated using the bandwidth delay product of the path between the end hosts. If the maximum window size is limited to this optimal window size it will limit the number of segments in transit between the end hosts, thereby preventing congestion and creating self-paced flows with minimal throughput variation, low packet loss, low delay and low jitter [Hol03].

The technique used in the previous study involved estimating an optimal window size from the bandwidth delay product of a specific route. Most TCP implementations allow users to configure the maximum window size. It was shown that if, at socket connection time, the estimated window is enforced for a connection, then the optimally limited window will induce TCP into a self-pacing mode for the duration of the flow, thereby achieving a flow with a steady non-variable throughput. However, this technique of using route specific windows can not be used to implement the proposed IP-ABR service, since window sizes cannot be varied after the connection is setup.

3.2.2 Flow control in TCP

The original TCP specification (RFC 793) included a flow control mechanism in its design. This flow control mechanism was incorporated in the original design to prevent a TCP sender from overflowing a receiving host's buffer (i.e., the receive window in the

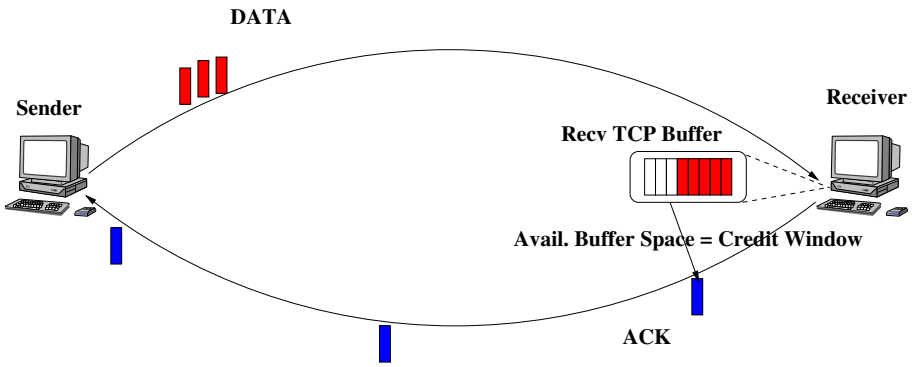

Figure 3.1: TCP Flow Control Mechanism.

TCP stack). The operation of this flow control scheme is illustrated in Figure 3.1. In this diagram we show how a receiving host informs the sender of the available buffer space by setting the "credit" window field in the ACK being transmitted back to the sender. Upon receiving the ACK packet the sender's TCP stack will limit the transmit window to the size of the credit window, unless it is greater than the congestion window, in which case the congestion window takes precedence over the credit window.

3.2.3 Dynamic credit limiting using a PEP

In order to regulate TCP flows to obtain a steady self-pacing mode without setting a window limit on the sending host, we propose to use the Performance Enhancing Proxy to transparently [1] intercept in-flight ACKs and limit the credit window to an optimal size as depicted by Figure 3.2. In order to do this the PEP will require access to the TCP header field. Overwriting of the credit window in in-flight ACKs will subsequently limit the sender's transmit window to the same size as the credit window set by the PEP. This optimal window limit set by the PEP will in turn prevent the TCP session from dropping out of self-pacing mode. In addition to this, the PEP can also reallocate the assigned bandwidth at any point in time for a particular connection, this allows the BMS

[1]The IP-ABR PEP does not split TCP connections, end to end connection integrity is preserved

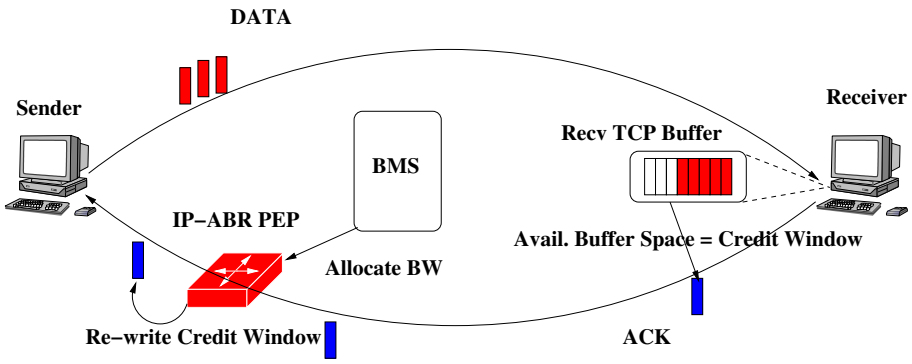

Figure 3.2: Using a Performance Enhancing Proxy to create IP-ABR service.

to dynamically vary the bandwidth as needed. Since the optimal window size assigned is estimated using the bandwidth delay product of the end to end network, the PEP will have to estimate the end to end delay for each flow, we shall provide further details on the mechanism developed to accomplish this in the next chapter.

3.2.4 Advantages of using a PEP

This novel approach of using a performance enhancing proxy to regulate TCP flows to create an IP-ABR service, has the following advantages.

1. The use of a PEP does not require modification of existing TCP stack implementations, which in turn allows this mechanism to be backward compatible with legacy TCP stack implementations.

2. By locating the PEPs closer to the edge of the network, we can distribute the work load of traffic regulation in the network thereby avoiding extra load on a few core router CPUs.

3.3 IP-ABR service operation

Figure 3.3 illustrates a network consisting of three remote subnetworks interconnected via a geostationary satellite using T1 speed links. This network has a large bandwidth delay product and is therefore ideal to deploy IP-ABR service to improve the QoS of TCP. There are a mixed variety of applications using this shared data network therefore it carries various kinds of data traffic such as voice, bulk data transfer and interactive data. Therefore in this network, the IP-ABR service is provided along side two other service classes UDP based IP-CBR and regular TCP. As described previously, CBQ is used to segregate the various traffic categories and process traffic on a priority basis, IP-CBR being the highest priority traffic followed by IP-ABR and then regular TCP.

Since this is a satellite network, link bandwidth is variable and depends upon environmental conditions such as weather and variation in satellite location. As mentioned previously, in order to create an IP-ABR service it is crucial that we have a bandwidth management service (BMS). In order to allocate bandwidth to the IP-ABR flows, the BMS makes use of Performance Enhancing Proxies (PEP). These IP-ABR PEPs are deployed either at the end hosts, or on an intermediate router. In Figure 3.3 the devices hosting the IP-ABR PEPs are depicted with red coloring. The IP-ABR PEP will regulate the corresponding TCP flows, so that the each IP-ABR flow will attain the bandwidth allocated to it.

In an IP network there are two different architectures that are used to provision QoS to the various types of applications, these two architectures are known as Integrated Services (IS) and the Differentiated Services (DS). In Figure 3.3 we illustrate how IP-ABR can be used in both Traffic Management Frameworks. It must be noted that using IP-ABR does not require the deployment of either framework. The following sections are merely suggestions of how IP-ABR services can be provisioned.

3.3.1 IP-ABR operation Integrated Services approach

In an IS framework, the end hosts request resources from the network prior to transmitting data using a protocol such as the Resource Reservation Protocol (RSVP) [RBS94]. Typically these requests are processed by an Admission Control Service that allocates the requested resources if available, in order to provision resources within the networks the admission control service interacts with a network management agent.

In order to provide IP-ABR service within an Integrated Services architecture, we suggest that applications wishing to use the IP-ABR service must send a request specifying its minimum and peak throughput requirement via the RSVP prior to transmitting data. In an IS framework an admission control service would normally process such a request from the client and reserve bandwidth in the network if available. In the case of IP-ABR service, we propose that a BMS along with IP-ABR PEP's be used to allocate bandwidth to end hosts. This makes the IP-ABR flows conform to the allocated bandwidth within the network.

The subnet depicted in the lower left portion of Figure 3.3 illustrates this. In this portion of the diagram we show an end host requesting IP-ABR service in an ISA framework. In the depicted network, prior to connection setup, the end host sends an IP-ABR service request message with the minimum and maximum bandwidths (100 kbps and 200 kbps respectively) specified. This request is processed by a BMS, which allocates the available bandwidth (132kbps) within the requested range via an IP-ABR PEP. Once the bandwidth is allocated to the PEP it will regulate the TCP flows in such a manner that they conform to this rate. IP-ABR service varies from other tradition ISA services, since the allocated bandwidth can also vary after connection depending upon the available bandwidth in the network at any given time, this can be done with the BMS reassigning bandwidth.

3.3.2 IP-ABR operation Differential Services approach

As we have shown, deploying IP-ABR service in an Integrated services architecture is feasible. However, the integrated services architecture has not become very popular since they are quite complex to deploy. The Differential Services Architecture (DSA) approach has become a more popular approach to provisioning QoS in IP networks.

In a Differential Services (DS) Architecture, bandwidth allocation is usually based upon pre-configured bandwidth allocation policies that are defined by a Network Administrator. End hosts do not dynamically request resources from a network, these QoS requirements are pre-negotiated by End Users and Service Providers using Service Level Agreements (SLA). The Network Administrator subsequently uses these SLA to define bandwidth allocation policies. QoS requirements are defined on a service category basis. Service category can be identified by using the DS code-point set in the IP header by the end host. The network classifies and processes packets based on the code point. In this framework, resources are usually not handled on an individual basis as they are in an IS environment, instead all traffic with the same DS code point are treated similarly [Sta02].

In this model, applications using the IP-ABR service do not explicitly request bandwidth from the network. However, network administrators using pre-configured policies determine appropriate bandwidth allocation for different IP-ABR flows. As illustrated by the subnetwork in the upper left corner of Figure 3.3 the BMS allocates the available bandwidth in the ratio of 5:3:2 as specified by the preconfigured Bandwidth allocation policies. The IP-ABR PEP can map these categories to individual flows using code-points. In this scenario bandwidth allocation is not done on an individual connection basis, instead the IP-ABR PEP allocates bandwidth in such a manner that all flows belonging to a specific DS code-point attain the same bandwidth. In the case of IP-ABR traffic additional traffic conditioning is not required by routing nodes inside the network since end hosts will conform to the allocated bandwidths.

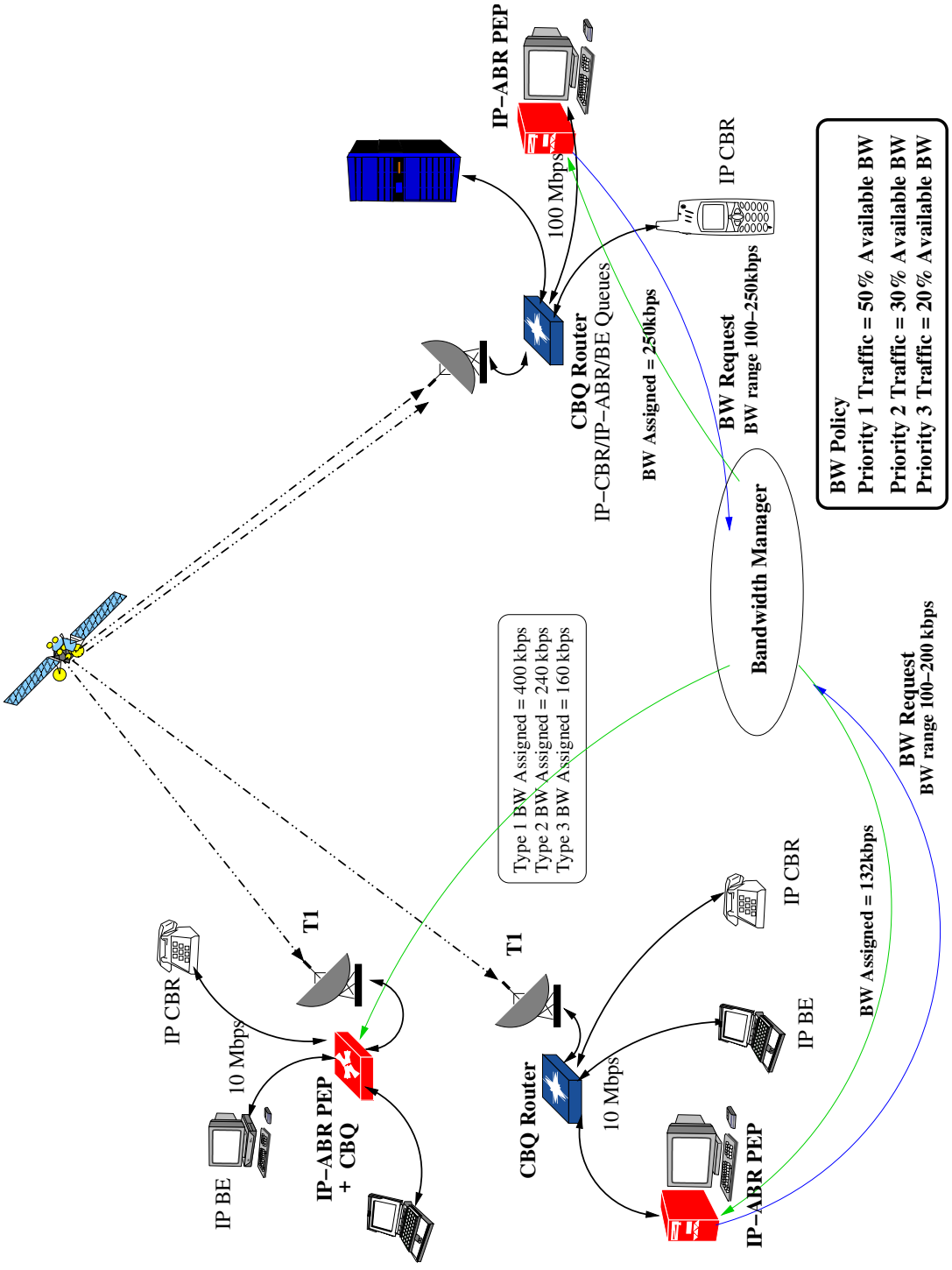

Figure 3.3: IP-ABR service operation in large latency IS and DS enviroment.

3.4 Benefits of IP-ABR service

IP-ABR service can offer many benefits to applications and network administrators some of which are listed below:

1. Advantages to applications:
   (a) Offers better QoS compared to that offered by TCP.
   (b) IP-ABR service can guarantee the minimal bandwidth needed by each individual flow.
2. Advantage to non IP-ABR traffic:
   (a) Reduces congestion, thereby reducing the impact that bursty IP-ABR traffic can have on other traffic types.
3. Network Management advantages:
   (a) Allows network operators to distribute available bandwidth on a priority basis using policies.
   (b) Allows end hosts to adapt quickly to changes in network bandwidth (useful in satellite networks where bandwidth varies over time.)

Chapter 4

IP-ABR Performance Enhancing Proxy - Mechanism

The IP-ABR service proposed earlier is intended to provide better quality of service by regulating TCP throughput so that congestion is prevented, thereby creating self-paced flows which have low throughput variation, low delay and low jitter. In the previous chapter we had proposed that an IP-ABR PEP be used to dynamically allocate bandwidth. In essence, the IP-ABR PEP acts as a TCP flow regulator, by which a bandwidth management service (BMS) can induce change in TCP transmission rates corresponding to variation of the available bandwidth, thereby creating IP-ABR flows. In this chapter we describe the rate control mechanism used by the IP-ABR PEP to regulate IP-ABR service flows.

4.1 Dynamic bandwidth allocation

The focus of this work is to design, implement and test a flow regulation system in the IP-ABR PEP and not the higher level protocols or implementation connected with the BMS.

However, we will briefly review the constraints that are placed upon bandwidth allocation that must be obeyed by the BMS.

In order to create the proposed IP-ABR service, we have proposed to use a bandwidth management service (BMS) which keeps track of the current bandwidth usage and from the available bandwidth dynamically "allocates" bandwidth to IP-ABR service flows. Now, let us assume that a given network segment or link has a physical bandwidth of BW_{max} all of which is used by IP-ABR traffic. Let us also assume that at a certain time t this network segment is shared by n IP-ABR service flows $f_1, f_2, f_3, ..., f_n$, which have been allocated flow rates of $r_1, r_2, r_3 ...., r_n$ by the BMS, so that each flow is guaranteed atleast the minimum bandwidth requested by it and total the bandwidth in use does not exceed the available bandwidth $BW_{available}$. This constraint upon the allocated rates can be expressed with the followin equation:

$$\sum_{1 \leq i \leq n} r_i \leq BW_{available} \tag{4.1}$$

Similarly, in the case where $r_1, r_2, r_3, ..., r_n$ are expressed as a fraction of $BW_{available}$, we can express the contraint upon allocated rates using the following equaation.

$$\sum_{1 \leq i \leq n} r_i \leq 1 \tag{4.2}$$

In the scenario depicted above the available bandwidth is equal to the bandwidth of the physical link, since we assumed that the network segment only carries IP-ABR service traffic, and therefore the IP-ABR traffic can use all of it. However, in networks with mixed traffic, the available bandwidth for IP-ABR service traffic varies due to the needs of traffic such as CBR traffic, which may have a higher priority than IP-ABR traffic. Therefore, an increase in bandwidth needs of higher priority traffic, leads to a reduction in available bandwidth. Now, if we assume that at time t the total bandwidth needed for higher priority

traffic is BW_{hp}, then the available bandwidth for IP-ABR flows can be estimated by the following equation:

$$BW_{available} = BW_{max} - BW_{hp} \tag{4.3}$$

As the available bandwidth changes on a given network segment, the BMS must reallocated bandwidth for each flow and with the aid of an IP-ABR PEP. However, in a packet network a "channel" between two end hosts is a virtual path through the physical network. The channel path may pass through more than one network segment, where each segment along the channel path has a different available bandwidth at any given time. Therefore, in the case of IP-ABR service, the bandwidth allocated to an IP-ABR service flow by the BMS, can not be greater than the available bandwidth on the segment with the least available bandwidth.

4.1.1 Estimating optimum window size

In the case of protocols such as TCP, which use a sliding window mechanism for flow control, the window size used for the sliding window will limit the achieved bandwidth. In a sliding window protocol, if we assume there are no errors, then a source may keep transmitting data until it reaches the end of the transmit window. Now, if the transmit window size is limited to a particular size, say W_L, then the bandwidth achieved can be estimated by the following Equation [Sta02]:

$$\beta = \frac{W_L}{2a + 1} \tag{4.4}$$

In the above Equation a is the propagation delay between end hosts. Using this relationship for a sliding window mechanism we can compute the optimal window size for a

given bandwidth β, where β is less than the bandwidth of the physical link.

$$W_L = \lfloor \beta \times (2a + 1) \rfloor \tag{4.5}$$

In the above Equation the term $2a + 1$ can be approximated by the round trip time (RTT) of a segment, since we have assumed there are no queuing delays or retransmission delays due to packet loss. We define the round trip time as the time interval between a packet transmission and arrival of an acknowledgement from the receiver. Now in the case of an IP-ABR service TCP flow the optimal window size Wnd_{opt} required to achieve the allocated throughput of $BW_{allocated}$ is given by the following Equation:

$$\text{Wnd}_{opt} = \lfloor BW_{alloc} \times \frac{data_{size}}{MTU} \times RTT \rfloor \tag{4.6}$$

In the above Equation the value of Wnd_{opt} is rounded down to the nearest octet, since TCP windows are expressed in terms of octets. This Equation is valid as long as the throughput allocated by the BMS, $BW_{allocated}$, is greater than $\frac{MSS}{RTT}$. Since, when $BW_{allocated}$ is equal to $\frac{MSS}{RTT}$ only a single segment is in transit between the end hosts, any further reduction in throughput will require segments of fractional MSS size, which cannot be transmitted, because of the silly-window prevention measures used in most stack implementations [Ste94a].

4.2 Estimating RTT

In order to compute the optimal window size for a particular flow the IP-ABR PEP will need to accurately estimate the round trip time (RTT). In order to reduce overhead, TCP transmits data in blocks, each of these blocks is referred to as a segment. The smallest block size that can be transmitted is referred to as the minimum segment size (MSS). In

most operating systems the MSS is a user configurable setting and on most systems is set to the default value of 536 bytes.

Assuming there are no packets lost, the RTT for a TCP segment is the time it takes from when a segment is transmitted to when an acknowledgement is received for it. In other words when a TCP segment is transmitted with a sequence number X at time t_1, an acknowledgement will be sent back with an ACK number equal to $X + MSS + 1$. The acknowledgement number informs the sender of the starting octet of the data that the receiver is next expecting. If the corresponding ACK is received at t_2, than the $RTT = t_2 - t_1$. In order to estimate the RTT for each flow the proxy must keep track of arrival times of all segments that have not been acknowledged yet.

TCP acknowledgements can sometimes be grouped together in a single ACK. For example consider Figure 4.1 in which data segments with sequence numbers 1 and 101 are transmitted by the sender. Here, the destination ACKs back both packets simultaneously, by only returning a single ACK for the latest segment it received (which in this case is 101). We account for this scenario by modifying the RTT estimation technique originally described, by estimating the RTT as the time difference between the time of transmission of the "latest" segment whose sequence number is less than an acknowledgement number and the arrival time of that acknowledgement.

4.2.1 RTT and retransmissions

Earlier we had approximated RTT as twice the propagation delay, where the propagation delay is equal to the link/channel latency. However, the RTT is greater than the propagation delay, since in addition to propagation delay the RTT, also includes the queueing delays and retransmission delay experienced by a transmitted segment and its corresponding ACK.

When a packet gets lost during transmission in a channel, TCP's ARQ mechanism

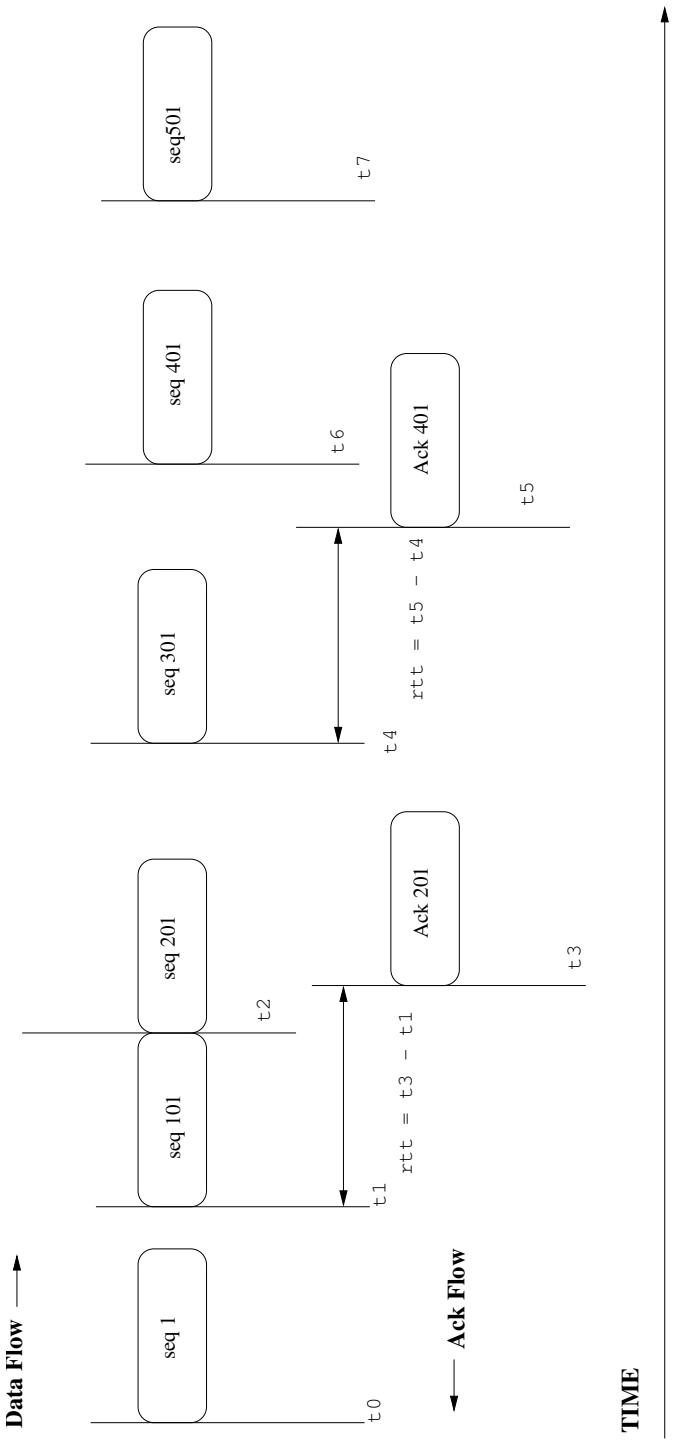

Figure 4.1: RTT Estimation.

retransmits the packet. However, there is no guarantee of successful transmission even when a segment is retransmitted, therefore multiple retransmissions might be required, thereby resulting in more than one segment with the same sequence number passing by the IP-ABR PEP. However, when an acknowledgement for one of the many retransmitted segments is received, it is not possible to match it to a particular retransmission, since we can not determine which of the transmitted duplicates have been dropped/lost and which have been not. This problem prevents us from being able to correctly estimate the RTT for retransmitted segments. Hence, we avoid estimating RTT for segments that get retransmitted. So if the IP-ABR PEP encounters a segment that is a duplicate of one it has already seen it raises a flag, so that RTT will not be estimated when an ACK corresponding to the retransmitted segment arrives.

4.2.2 RTT averaging

RTT estimation using the scheme described in the previous section includes both the channel latencies and queuing delays. However, we are concerned with estimating just the channel latency. Hence, we need to develop a scheme that can separate the queuing delays from the estimated RTT. If we do not separate queuing delays from our RTT estimate it will be detrimental to IP-ABR operations. This is due to the fact, that using RTT estimates that include queuing delay will result in an inflated bandwidth delay product, thereby, giving the IP-ABR PEP an impression of a channel with longer latency than the real latency. This in turn causes the end hosts to inject more segments into the channel, which in turn creates further congestion and leads to larger queuing delays. This increase in queueing delay will again factor into the IP-ABR PEP's window estimation, thereby, creating a feedback loop which will eventually lead to congestion and packet loss, as result of which the IP-ABR service flow breaks out of self-pacing mode.

To prevent this, we developed a scheme in which we "skim off" the queuing delay.

The idea for this is based on the fact that for a given flow the link/channel delays are mostly constant, assuming that the route between end hosts does not change. Queuing delays on the other hand tend to fluctuate, causing variation in estimated RTT values. Thus we can attribute any increase in estimated RTT to an increase in queuing delay, and similarly any decrease in RTT estimation is attributed to a decrease in queueing delay. Therefore in order to estimate the link latency we look for the smallest RTT estimate; the smaller the RTT estimation the closer it is to the link latency. As can be seen, it is quite essential for this technique that the link latency does not vary with time. Route variation is a phenomena in connectionless packet switched networks wherein packets belonging to the same flow (i.e., packets with the same source and destination), may take different routes before they arrive at the destination. Of course we need to accommodate slow variation in route delay such as induced by movement of satellite relay stations. It is for this reason that our estimation function will involve weighted averaging with a lesser but non-zero weight given to RTT values greater than the current RTT estimate. To estimate the smallest RTT we can compute a weighted average RTT as follows

$$artt_n = wt \times rtt_{est} + (1 - wt) artt_{n-1} \tag{4.7}$$

Where $artt_n$ is the average RTT after the nth sample, rtt_{est} is the estimated RTT of the nth sample, $artt_{n-1}$ is the average RTT of the $n-1$ sample and wt is the weight given to the nth RTT sample. We use different weights depending on whether the sample RTT is greater than the average RTT or not, since we want $artt_n$ to be closer to the smallest RTT. Therefore, larger weight is given when rtt_{est} is less than $arrt_{n-1}$ and a smaller weight is used when rtt_{est} is greater than $artt_{n-1}$. From early simulation experiments we find that the following ranges for weights work well.

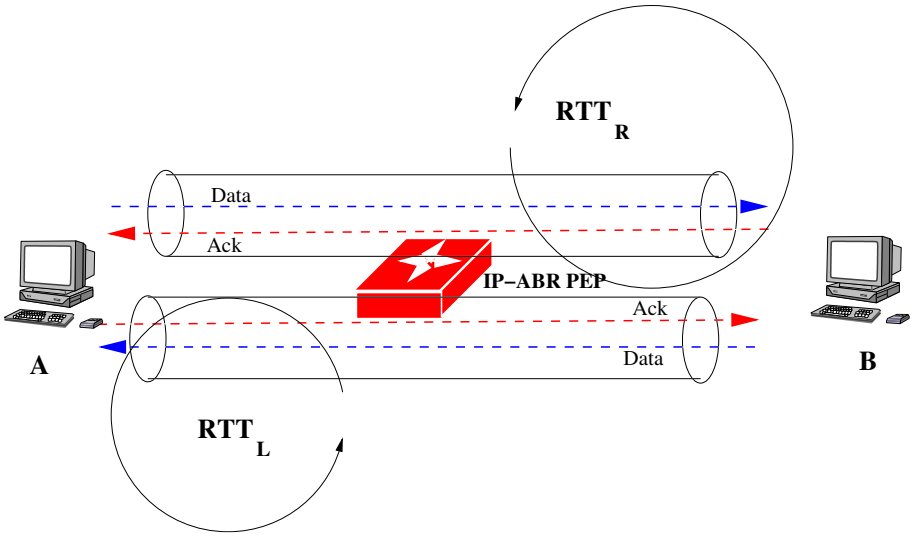

Figure 4.2: RTT Estimation

$$wt = \begin{cases} 0 < x \leq 0.002 & \text{if } rtt_{est} \geq artt_{n-1}, \\ 0.6 \leq x < 1 & \text{if } rtt_{est} < arrt_{n-1}. \end{cases}$$

The RTT estimation techniques mentioned here were first implemented and tested in the NS simulation environment. During this early testing we only verified that the RTT estimates were within bounds of the maximum theoretical queuing delay and network delay for the test network. We did not conduct any further statistical analysis on the variance of the RTT estimates. However, since our RTT estimates are used to estimate the window sizes used to regulate TCP flows, any failure on the part of RTT estimation would reveal itself as incorrect division of bandwidth between competing sources. As we shall describe in the following chapter, we have conducted more rigorous tests to verify the accuracy of the bandwidth achieved by TCP flows that are regulated using the proxy which do confirm the functionality of the RTT estimator.

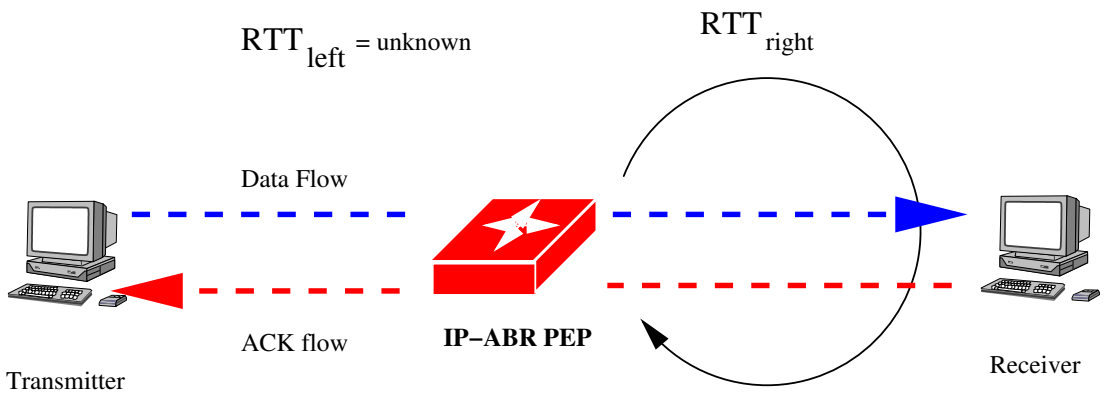

Figure 4.3: RTT estimation for asymmetric data flow.

4.2.3 Estimating RTT and PEP location

The mechanism described previously assumes that the PEP is located at the sender. However, when a PEP is located midstream as depicted in Figure 4.2, the RTT estimation has to be done on both sides of the router. Let us refer to these RTT estimates as RTT_{left} and RTT_{right}. RTT_{left} is estimated using data flowing from B to A and ACKs flowing from A-B. Similarly RTT_{right} can be estimated using the flows in the opposite direction. As a result of this split RTT estimation, the total RTT for a flow is the sum of RTT estimates on both sides of the PEP as shown below:

$$RTT = RTT_{left} + RTT_{right} \tag{4.8}$$

We can make use of this scheme, even if the PEP is located at the end host. Since, if the PEP is located at the transmitter we see that RTT_{left} is essentially zero then $RTT = RTT_{right}$. However, there is a limitation here. As can be seen, in order to estimate both RTT_{left} and RTT_{right} the PEP requires that data flows in both direction. However, if the connection is asymmetric as shown in Figure 4.3, where data flows in only one direction, we can estimate the RTT on only the side of the PEP. However, by placing the PEP as close as possible to the sender we can avoid this problem. Another

advantage of distributing the IP-ABR PEPs closer to the edge is that the processing load is more disbursed. Hence a good general strategy is to always place IP-ABR proxies as close as possible to the end host even though in a general sense this in not a fundamental requirement for implementing this service. Throughout this report and the implementation code we assume the latter strategy and take the full link latency to be reflected by the proxie's ongoing traffic RTT estimate.

Chapter 5

IP-ABR PEP simulations

The initial implementation of the IP-ABR Proxy was built and simulated using NS. This allowed us to validate the algorithm and mechanisms to be used to build the IP-ABR proxy before actual proxy implementation. The NS simulation environment also gave us the flexibility to test more complex test scenarios. This chapter details the tests simulated, the results and modifications we had to make to the IP-ABR PEP.

5.1 Network Simulator (NS) background

Network Simulator (NS) is an event driven simulation tool developed by the Lawrence Berkeley Labs (LBL) and the University of California at Berkeley to simulate and model network protocols. NS has an object oriented design, and is built with C++, but also has an ObjectTCL API as a front end [FV02].

As stated earlier, the goal of the IP-ABR PEP is to regulate TCP flows to attain a pre-determined bandwidth such that TCP flows are held in a self-paced mode for the duration of test, thereby, achieving throughput with minimal variation for the duration of the flow. To validate our claims we conducted two sets of tests. The first of these tests involves

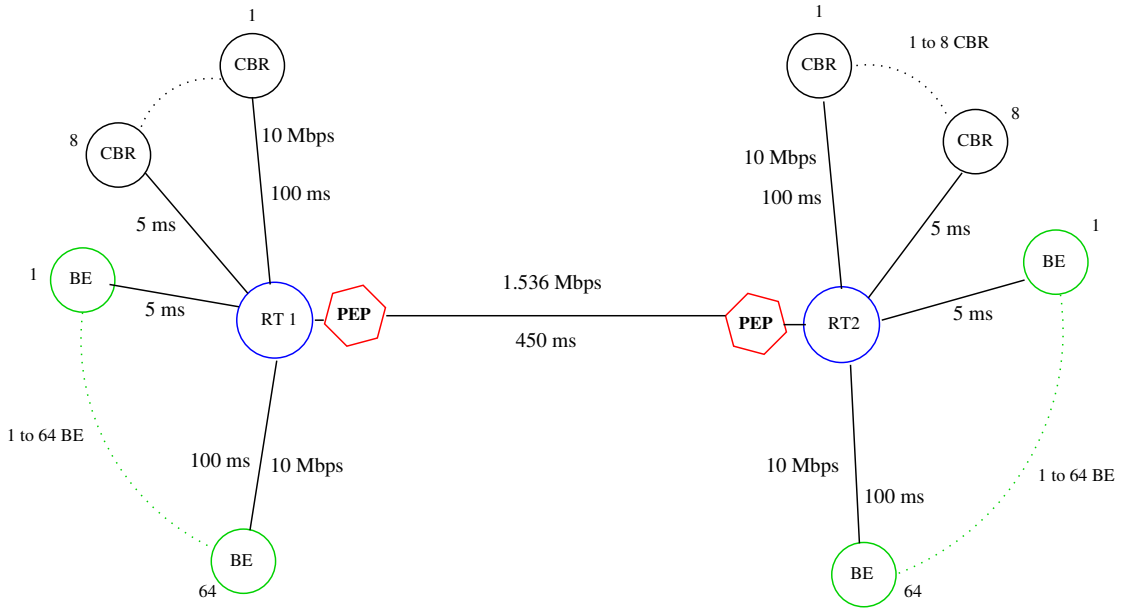

Figure 5.1: NS simulation topology for IP-ABR PEP Testing.

capturing throughput variability statistics over time. In addition to looking at throughput variability, we must also verify that bandwidth distribution across several flows is fair for which a second set of tests were conducted.

5.2 Throughput variability test

5.2.1 Test Configuration

The simulations for these tests were configured as follows: As depicted in Figure 5.1 a total of 72 hosts (64 best-effort and 8 constant bit rate) were placed on either end of a satellite link. Routers (RT1 & RT2) were used to connect the 72 hosts on each side to the satellite link. The link itself is similar to a bi-directional T1 satellite link in which the link has a latency of 450 ms and a 1.536 Mbps bandwidth in each direction. Each host was connected to the router over a high speed link (10 Mbps). The link latencies of the high speed links range between 5 and 100 ms. We varied the link latencies on the end links

so that we could simulate a scenario where each TCP flow has a different round trip time (ranging from 910 to 1100 ms). In addition to this we attached an IP-ABR PEP to each of the routers. The proxy node has access to the associated router's queue so that it can allocate bandwidth to each flow by modifying the ACK frames in transit.

The hosts were configured such that, for each host at one end of the link, there was a corresponding host at the other side of the link, forming a pair. During our tests, a host from each end host pair would send a constant stream of data to the paired host over the bottleneck satellite link over a TCP connection. This results in bi-directional TCP traffic flows between hosts, which we shall refer to as forward and reverse flows. The "full stack" New Reno implementations of TCP were used to simulate the TCP traffic. The constant bit rate traffic simulated was intended to be similar to voice traffic, so, we configured the CBR hosts to transmit 64 bytes of data, that is a corresponding 92 byte IP packet, every 8 ms using UDP. Correspondingly, each CBR pair had a bandwidth requirement of 92 kbps.

Voice traffic requires minimal jitter and packet loss, since UDP is an unreliable protocol and any packet loss will lead to a degradation in voice quality. Thus, if TCP traffic shares a queue with CBR traffic it will cut into the bandwidth needed for the CBR traffic, thereby degrading CBR traffic QoS. Therefore, in order to meet the QoS needs of the voice traffic we must guarantee the required bandwidth and segregate it from TCP traffic. To do so, we used class base queuing (CBQ) at the router. In our simulation, CBQ was configured to segregate the two different traffic classes (CBR and BE) into separate queues. The CBR queue was managed with a drop-tail queuing discipline. The queueing discipline for the BE traffic was either drop-tail or Random Early Detection (RED) depending on the test. The CBR traffic was given a higher priority than the TCP traffic. The queue size for the TCP traffic was 64 packets long. The queue size for the CBR flows was

computed as follows [Hol03]:

$$qsize_{cbr} = A + cbr_{num} \cdot \frac{8 \cdot cbr_{size} \cdot A}{\beta \cdot cbr_{interval}} \tag{5.1}$$

In this case $cbr_{num} = 8$, $cbr_{size} = 64$, $cbr_{interval} = 8ms$ and A is the number of CBR packets that arrive in the time it takes to transmit a BE packet, which can be estimated as follows.

$$A = cbr_{num} \cdot \lceil \frac{8 \cdot be_{size}}{\beta \cdot cbr_{interval}} \rceil \tag{5.2}$$

This CBQ configuration guarantees that the bandwidth needs of CBR traffic are met, and it also isolates it from the TCP traffic, thereby offering better QoS. As mentioned previously, using CBQ in the network is a typical technique used to meet the various QoS requirements of different traffic classes.

5.2.2 Test description

In the tests conducted, TCP traffic was stagger started in such a manner that 8 flows were started at a time every 0.5 secs and for the first 120 seconds TCP end hosts were given the entire bandwidth of the satellite link. This provided sufficient time for all flows to go through the initial slow start phase. The 8 CBR pairs started transmitting at 120 secs from the start of the test. The bandwidth needs of the CBR traffic were guaranteed by using CBQ. This results in a reduction of the available bandwidth for the TCP traffic. TCP hosts must adjust to this reduction in available bandwidth in order to prevent congestion.

What we sought from these tests, was to see how TCP throughput varies over the duration of the test. In particular we are interested in seeing how TCP throughput behaves in situations wherein available bandwidth is constant and also when it varies in response to needs of higher priority traffic. However, unlike CBR traffic, TCP traffic is bursty by nature. If we were to look at the instantaneous throughput rate we would always see large

variations. However, instead of looking at the instantaneous throughput, if we were to look at throughput over an appropriate interval we would find less variation. Therefore, we use a technique of window averaging developed in a previous study [Hol03]. This method involves averaging the throughput sampled over a 5 second window interval, every time a block of data is received at the receiver application. This moving window approach dampens the variations attributed to TCP's bursty nature, but still allows us to observe variations in throughput caused by the dynamic window sizing we described previously.

We conducted two types of tests, in the first type we enable the Random Early Detection (RED) on the TCP traffic queue with the minimum and maximum thresholds set to 32 and 64 respectively. In the second test we used a drop-tail queuing disciple with the maximum queue size set to 64 packets, and we also enable the IP-ABR PEP attached to each router. Each of the IP-ABR PEPs regulates the TCP sender on its side of the link. In this simulation the IP-ABR PEP is configured to distribute the available bandwidth equally amongst the competing TCP flows.

5.2.3 Throughput variability test results

Figure 5.2 shows a plot of TCP throughput measured for each flow when we used RED as the queuing discipline for the TCP queue. The New Reno version of the TCP stack was used in these simulations. From Figure 5.2 we can clearly see, that despite available bandwidth being constant for the periods between 0-120 secs and 120-240 secs, the throughput of each individual flow varies. TCP throughput seems to repeatedly climb and drop exhibiting an irregular oscillatory behavior. The reduction in the steepness of the peaks in the later half (i.e., after 120 secs) has to do with reduction in available bandwidth to TCP flows from 1.536 Mbps to 0.736 Mbps due to the 8 CBR sources coming online as described above.

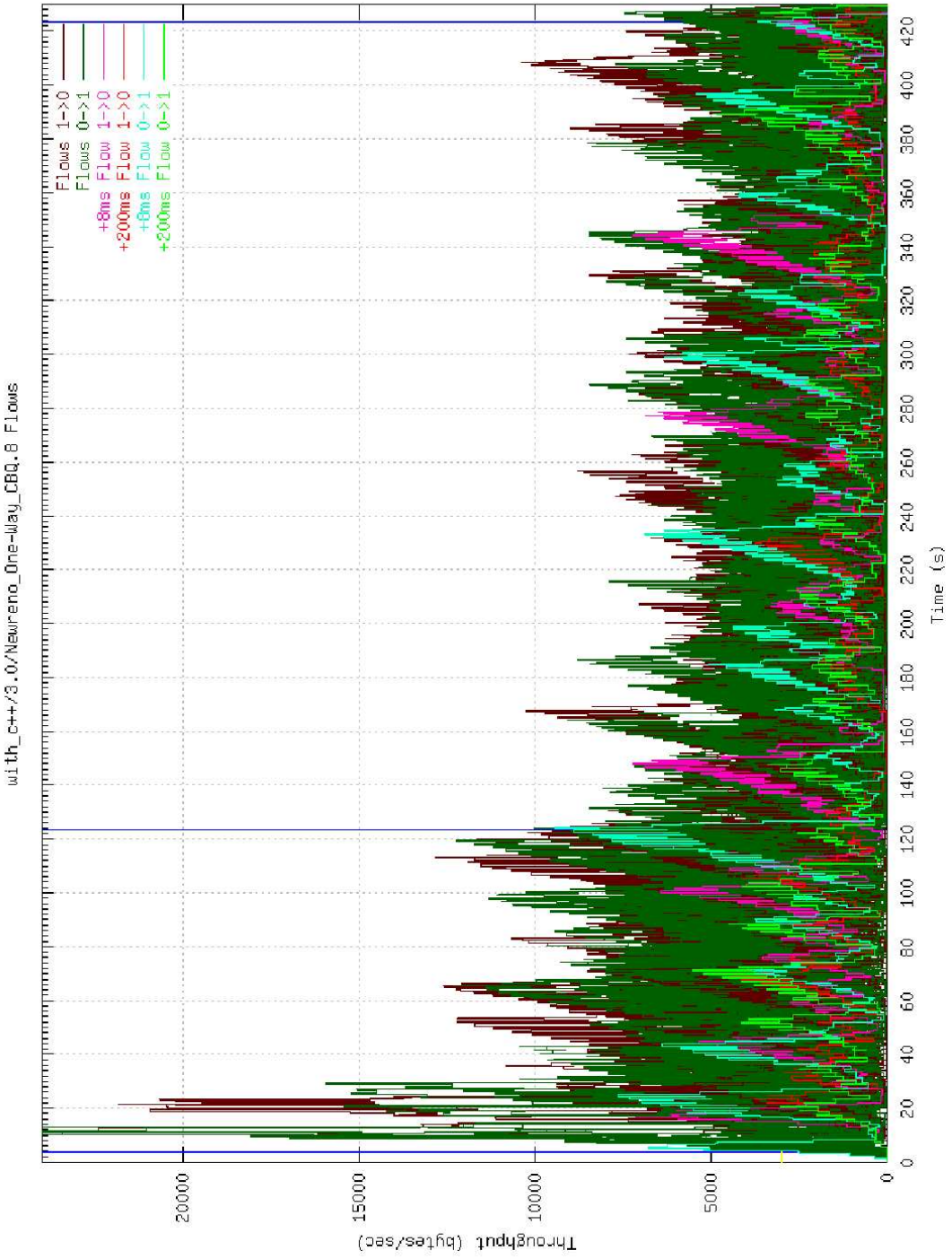

Figure 5.2: Throughput vs. Time (RED + CBQ)

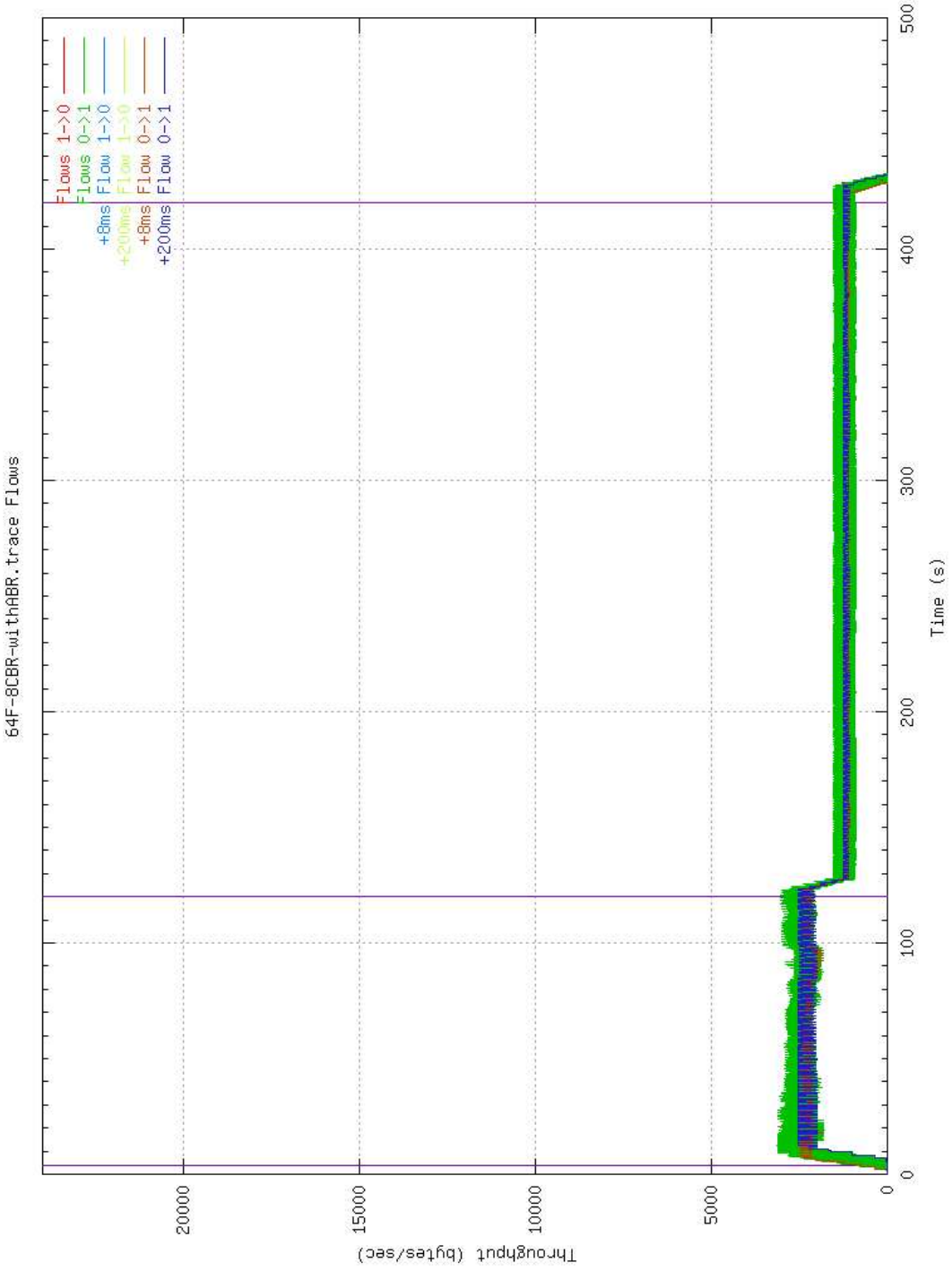

Figure 5.3: Throughput vs. Time (IP-ABR PEP)

The irregular oscillations that the throughput exhibits was expected, since after an initial ramp up during slow start TCP does not maintain a steady-state, instead TCP stacks constantly search for an upper bound by linearly incrementing their windows, which in turn causes the throughput to grow and will eventually create congestion at the queue at the bottleneck link, which in turn triggers congestion alleviation, causing the drops that we see. This pattern is irregular, since we have multiple TCP flows sharing a link, each competing against the others the with congestion reaction times that are dependent on the traffic rates and propagation delays. This results in some flows more often getting a higher throughput than others, which leads to unfair average distribution of bandwidth.

Figure 5.3 shows the results of the test using IP-ABR PEP, which paints a contrasting picture to the previous test. We can see that the TCP throughput does not vary throughout the test except for the moment at which the available bandwidth changes. As soon as the IP-ABR PEP can estimate the RTT for a TCP flow, the flow is band-limited by the PEP, so that the end hosts do not inject more than the optimal number of segments required to meet the allocated bandwidth. Just as in the previous case, the TCP flows go through a slow start period where the window is exponentially ramped up, however, before any congestion occurs it will hit the window limit set by the IP-ABR PEP since the window is clipped to the optimal size. At this point, TCP will enter into self-pacing mode for the duration of the test, thereby achieving a throughput with minimal variation. The only time the window size varies again is when the IP-ABR PEP modifies the window limit corresponding to changes in the allocated bandwidth. In the case of this test, the bandwidth of the IP-ABR service flows is reduced at 120 seconds to meet the needs of the higher priority CBR traffic.

5.3 Fairness tests

5.3.1 Configuration

In this test, we are interested in verifying that the IP-ABR PEP can guarantee fair bandwidth usage. In conducting this test we used a similar configuration to that of the previously described throughput variability tests. However, we removed the 8 constant bit rate (CBR) hosts from each end. Another change we made is to keep the available bandwidth constant for the duration of the test. This allows the TCP traffic to use all of the satellite link's bandwidth. We configure the IP-ABR PEP to equally distribute the available bandwidth to each connection. The test duration itself was shortened to a period of 100 secs. We repeated the test for the various TCP MSS settings of 256,536,1024 and 1452 bytes.

5.3.2 Fairness metric

Fairness is a vague concept, since it is subjective with respect to the needs of the end host. So what may be fair to one application may not be fair to another. This makes it hard to come up with a single measure for fairness that quantifies it. However, in our case what we are interested in, is the scenario wherein an IP-ABR PEP is trying to regulate all flows, such that the link bandwidth is equally shared amongst the various BE flows. The closeness of the achieved throughput (or goodput to be precise, since we are only interested in packet traffic that successfully enters the receiver's delivered data stream) to the desired rate should be reflected by our measure of fairness.

In other words If we have N hosts sharing a link with a bandwidth β, the throughput utilization of each of the flows should be $\frac{\beta}{N}$. We can measure per-flow link utilization achieved for each flow as follows.

$$Util_{link} = \frac{N \cdot \sum_{1 \leq i \leq k} MSS}{\beta \cdot T} \tag{5.3}$$

Where $\beta = 1.536Mbps$ (satellite bandwidth), $N = 64$ (number of BE flows) and k is the number of segments successfully delivered to an application by the receiver over the duration of the test (T secs).

5.3.3 Results of fairness tests

After measuring the utilization of the various flows we came to the conclusion that the flow utilization values were spread over a wide range between 0.72 to 0.95 of the normalized bandwidth allocated by the IP-ABR PEP (which in this case is equal for all the flows). To determine if there is a pattern to this distribution, we examined the variables that are involved in computing the window size. Of all the variables, only the RTT varies significantly between the flows, since we had configured the test network topology so that the link latencies between end hosts vary between 910 and 1100 ms.

We then plotted the utilization against the respective channel/link latency. As shown in Figure 5.4 the utilization appears to be distributed in a pattern with respect to the link latency. This pattern was noticed in all tests irrespective of the MSS, but as we can see there also seems to be some relationship to the MSS. This relationship between utilization, RTT and MSS prevents this first implementation of the IP-ABR PEP from guaranteeing fair usage of bandwidth.

In order to identify the cause of the utilization skew we critically examined the bandwidth delay algorithm used in the IP-ABR PEP to compute the optimal window size. As stated previously, the optimal TCP window size can be computed using Equation 4.6. This formula computes the window size rounded down to the nearest octet. However, from examination of TCP stack implementations we discovered that, when a TCP sender receives a window size advertised by the receiver, it will round down the window to the nearest MSS. This is to avoid phenomena called “silly window syndrome,” to which sliding window based flow control schemes are vulnerable, wherein the sender sends several

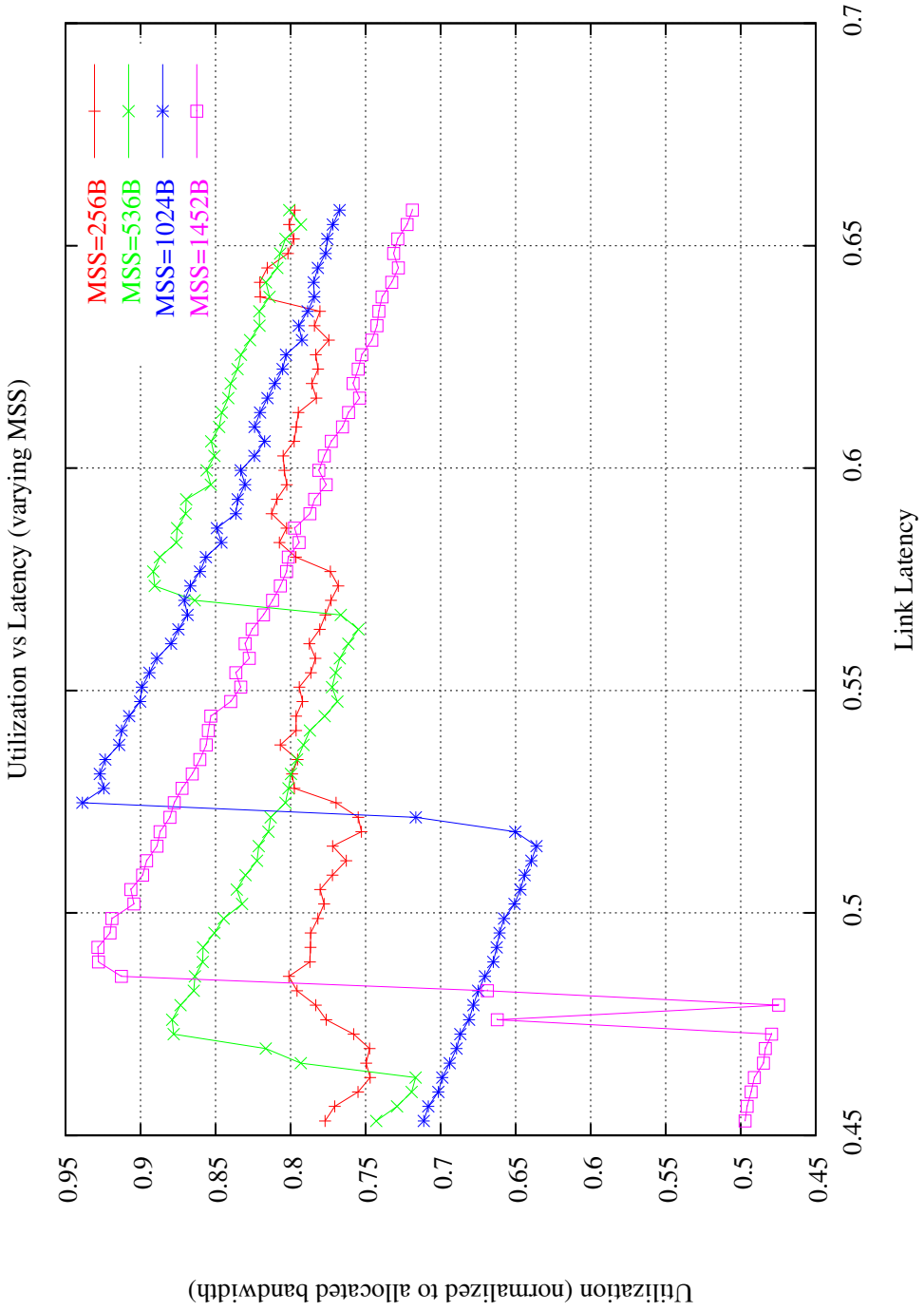

Figure 5.4: Utilization vs. Latency (Measured)

packets smaller than the MSS size, instead of waiting to send a single larger MSS size packet [Ste94b].

To prove that this assumption is in-fact the cause of the utilization skew, we tried to theoretically estimate the effects on utilization from rounding down of the optimal window to MSS boundaries. To do so, we computed the optimal window size using the RTT values from our test configuration. We then rounded the windows down to the actual window size that TCP stacks use, and then computed the actual achieved throughput. From the actual achieved throughput we can also compute the normalized throughput or utilization. We finally plotted the utilization against the link latency, so that we could compare it to the measured results from the earlier simulation. The following MATLAB program was used to theoretically compute the throughput achieved by each flow under such circumstances:

```
function a = actual_throughput(link, num_users)
MSS=536;
MTU=576;
bw = 1536000/8;
[rows cols] = size(link);
a = zeros(rows, cols);
for r=1:rows
    for c=1:cols
                window = 2*bw*(2*link(r, c)+.45)*MSS/(MTU*num_users);
                tcpwnd = floor(window/MSS)*MSS;
                a(r, c) = tcpwnd/(2*(2*link(r, c)+.45));
        % normalize the throughput
        a(r, c) = a(r, c)/(bw/num_users)*MTU/MSS;
    end
```

end

From Figure 5.5 we can see that the theoretical estimates of utilization seem to closely resemble the results from measured values shown in Figure 5.4. This proves our assumption that the silly window avoidance measures implemented in TCP skew the per flow utilization.

5.4 Modification of the IP-ABR PEP

As we have shown in the previous section, rounding the optimal TCP windows size to the nearest MSS skews the utilization, but, TCP senders round down the optimal window size to the nearest MSS in order to avoid the silly window syndrome. In order to prevent this from affecting the IP-ABR PEP's ability to guarantee fair bandwidth usage, we propose to modify the existing algorithm used to estimate the optimal window size, so that it can compensate for window rounding.

5.4.1 Silly window compensation

In the IP-ABR PEP we will estimate the optimal window size for a particular flow as before. However, we will then round down the optimal window size to the nearest MSS to get the actual window size (wnd_{actual}). Rounding down the optimal window size creates a deficit in the allocated bandwidth. To make up for this deficit, we first estimate the deficit and then carry it forward as credit and apply it to subsequent window computations. This is done by computing the difference between optimal window size and actual window size as δ_{credit}, and then carrying the credit forward. Upon receiving the next subsequent ACK packet for the flow, we apply it the optimal window size before rounding again. The

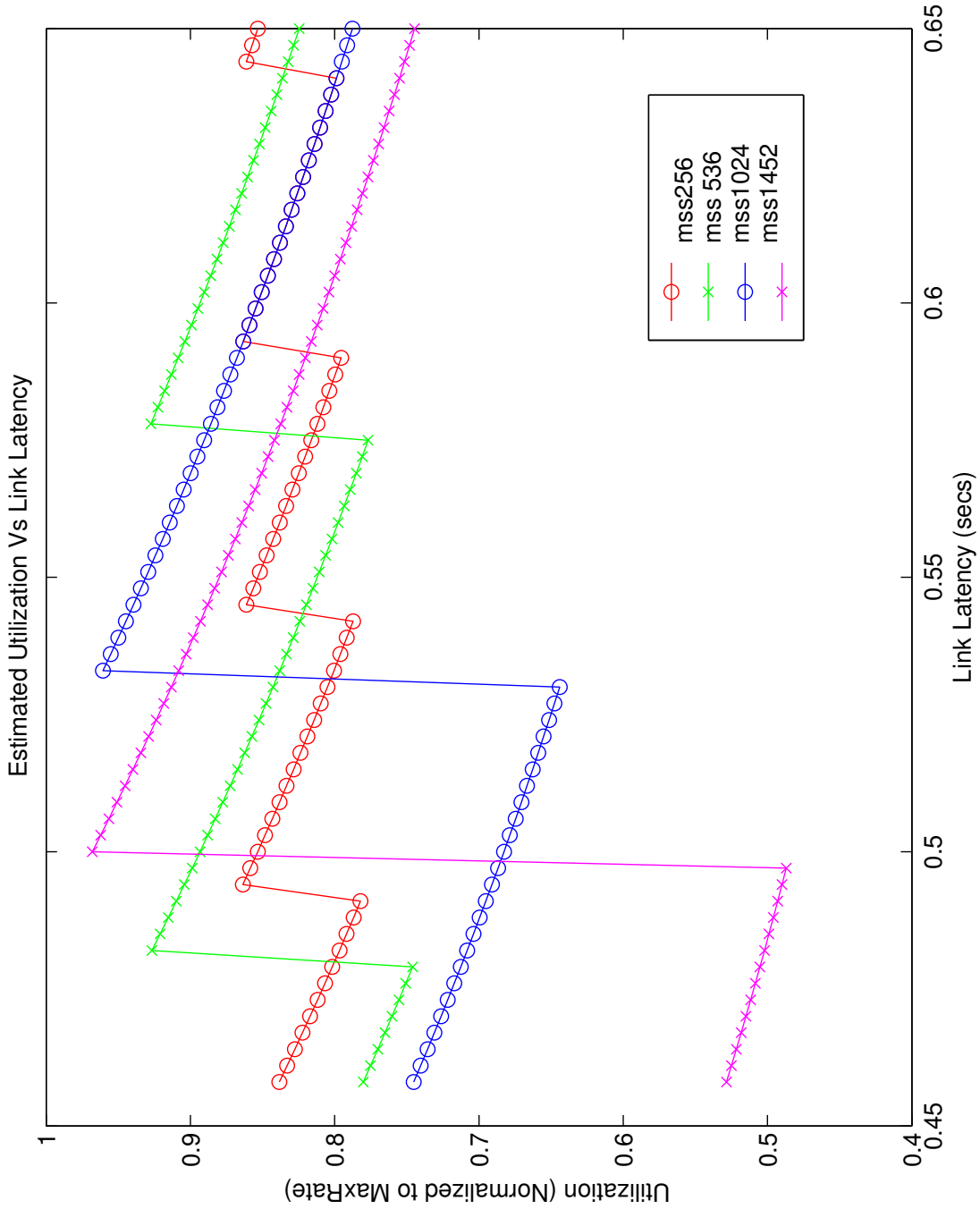

Figure 5.5: Utilization vs. Latency (Estimated).

new modified algorithm can be expressed as follows:

$$wnd = \delta_{credit} + (\beta_{available} \times \frac{data_{size}}{MTU} \times RTT) \tag{5.4}$$

$$\delta_{credit} = wnd \mod MSS \tag{5.5}$$

$$wnd_{actual} = wnd - \delta_{credit} \tag{5.6}$$

Prior to this modification, the value of the optimal window size was fairly constant (unless either there was a change in the available bandwidth or the RTT). With this modification, window sizes set by the IP-ABR will periodically fluctuate between W and W + MSS. Where W is the actual window size or the optimal window size rounded down to the nearest MSS. However, this small variation in burst sizes will not lead to a noticeable variation in throughput.

5.4.2 Fairness after using silly window compensation

After making the modification to the optimal window size estimation algorithm we repeated the previous test. Figure 5.6 shows the results of that test in comparison with the results prior to the modification. For the sake of clarity in this figure, we only plotted the results of the test with the MSS set to 536 bytes (the most commonly encountered WAN MSS value), however, similar improvements were noticed for MSS settings of 256 and 1440 bytes. From Figure 5.6 we can clearly see that the modification succeeds in breaking any relationship between utilization, latency and minimum segment size. We also noticed that the overall utilization (i.e., the mean utilization of all flows) improves from 86% of the link capacity to 90%.

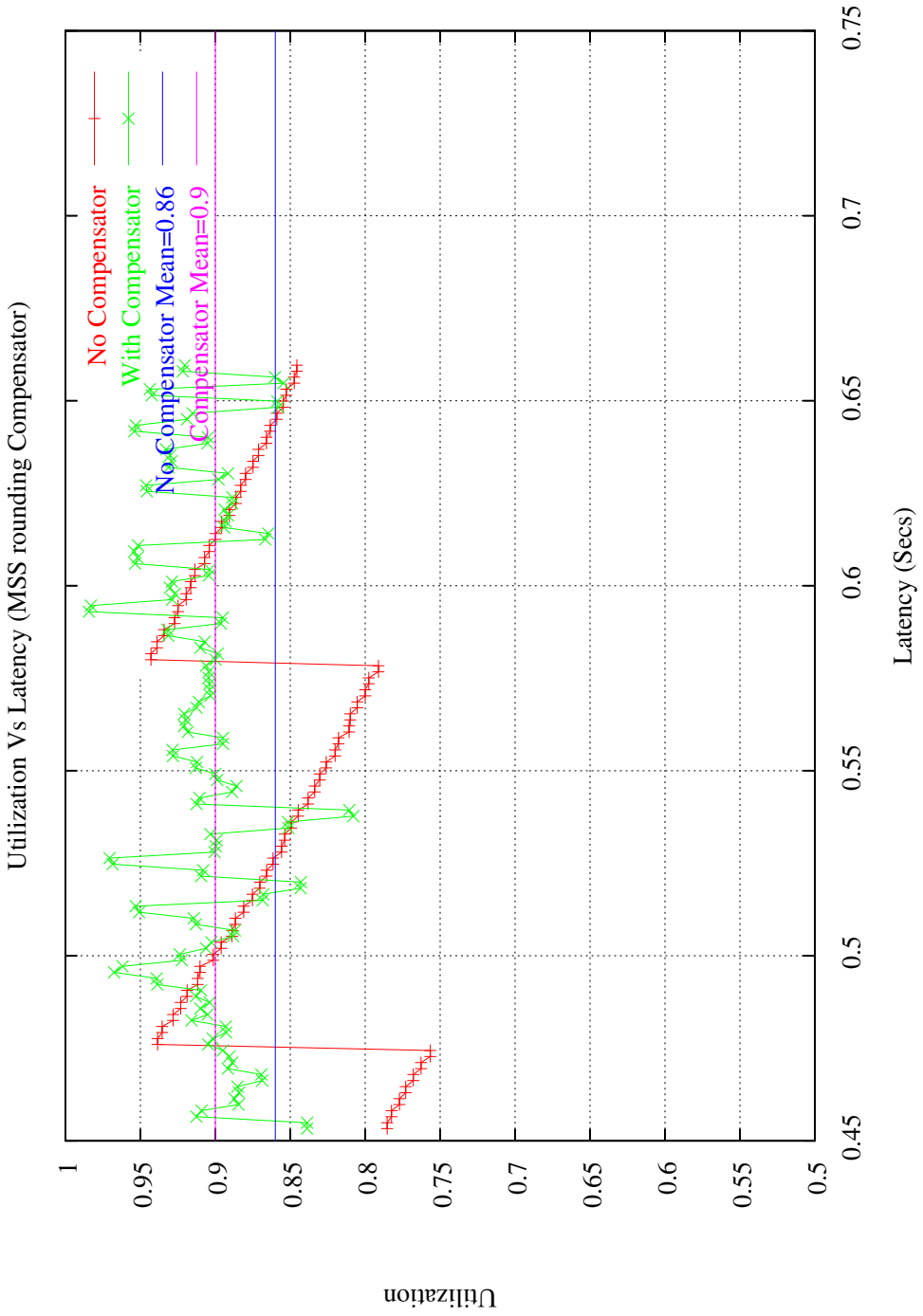

Figure 5.6: Utilization vs. Latency with and without silly window compensation.

Chapter 6

IP-ABR PEP Implementation

Having been able to successfully implement and test the IP-ABR PEP in the NS simulation environment, our next task was to build a prototype of the IP-ABR PEP for the LINUX platform. This chapter details the design of the prototype IP-ABR performance enhancing proxy.

6.1 Design

In order to create an IP-ABR service we proposed using performance enhancing proxies to dynamically allocate bandwidths to TCP hosts. As described earlier the IP-ABR PEP dynamically allocates bandwidth by means of dynamic window limiting of TCP flows, which as we have shown, can be achieved by modification of in flight ACK packets. In conjunction with a bandwidth management service which determines the bandwidth to allocate, IP-ABR PEP induces flows that have lower delay, lower jitter and less packet loss then compared to regular TCP flows. It also allows the bandwidth management service to dynamically adjust flows to use available bandwidth as services change over time.

The prototype IP-ABR proxy has an object-oriented design and was developed using C++. This allowed us to reuse most of the code from the simulated version without major modifications, since most of the algorithms and techniques used in the simulated version were developed using C++ standard template libraries. The prototype IP-ABR proxy was designed to run as daemon process.

The flow chart shown in Figure 6.1 illustrates the various steps involved in processing a TCP packet by the IP-ABR Proxy. For each flow, the IP-ABR proxy determines the optimal window size corresponding to allocated bandwidth and the channel latency. The bandwidth allocation is determined by a BMS, however the IP-ABR proxy must monitor the flows and determine the channel latency as we have described previously in Chapter 4.

6.2 Flow tracking

6.2.1 Flow monitors

In order to estimate the RTT for each flow, the proxy must keep track of data packets that have not been acknowledged previously. Then, using the RTT we try to estimate the link latency as described previously, by estimating the weighted average RTT ($ARTT$) for the first n-1 samples per flow, wherein higher weighting is given to smaller RTT estimates, thereby giving us an $ARTT$ that approaches the natural link latency. Another variable that must also be tracked on a per flow basis is the credit, δ_{credit} the difference between optimal window size and actual window size described earlier. We keep track of this so that we can apply any reminder to the subsequent packet.

To handle operations on a per flow basis, we make use of a Flow Monitor object as illustrated in Figure 6.2. For each flow being regulated by the IP-ABR PEP we instantiate an IP-ABRFlowMonitor object. Each flow monitor object instantiated keeps track

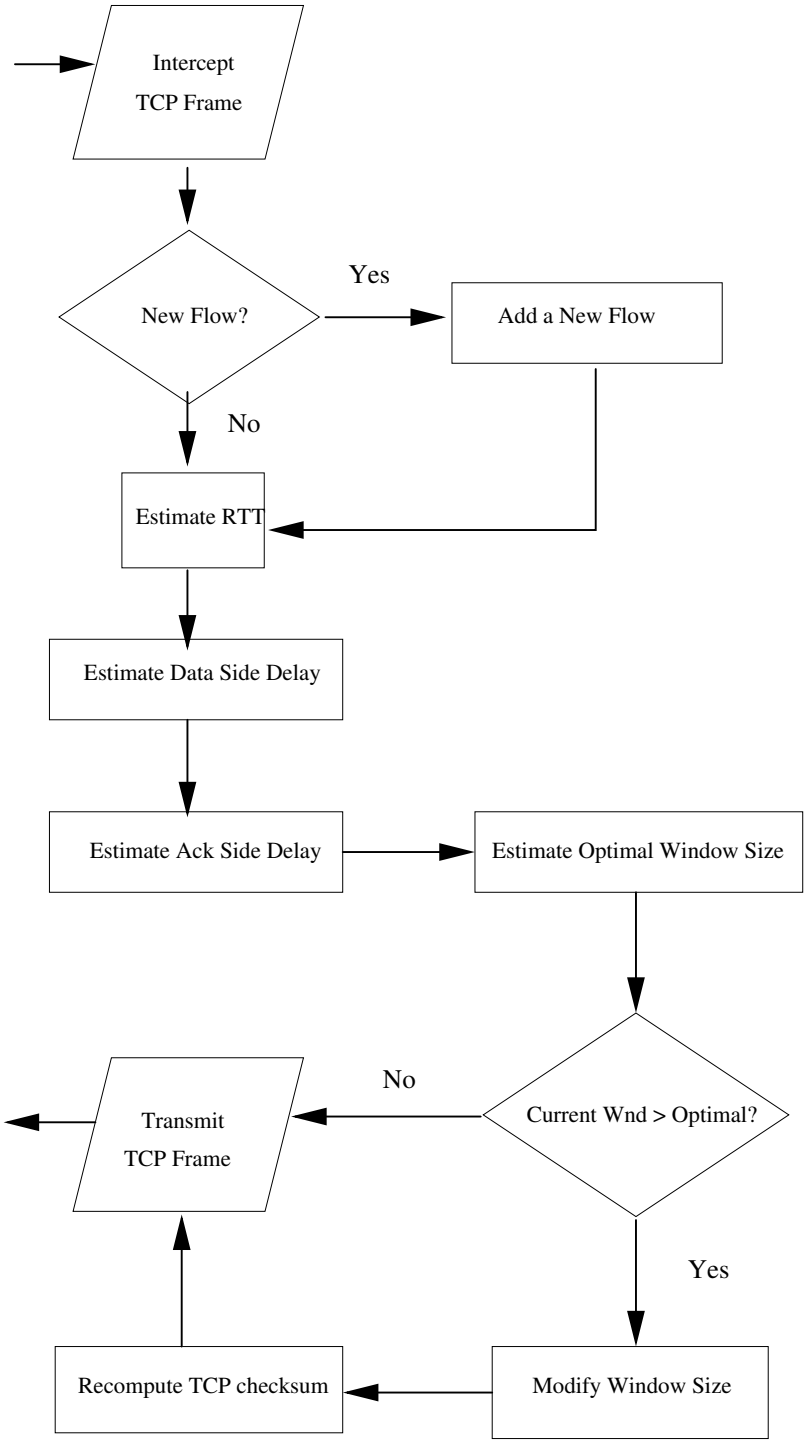

Figure 6.1: IP-ABR PEP Packet Process Flow Chart

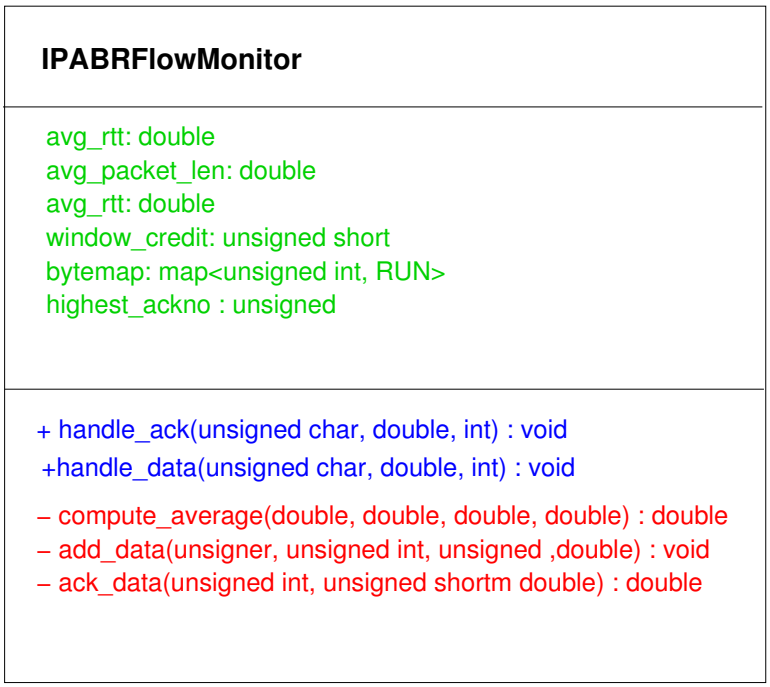

Figure 6.2: IPABRFlowMonitor Class

of the average RTT (avg_rtt), the average packet size (avg_pkt) and the window credit (window_credit). In addition to these attributes, each flow monitor object also maintains a list of unacknowledged data packets with their corresponding arrival time stamps. As previously described, the arrival time is used to estimate the RTT and subsequently the $ARTT$ when an ACK is received. Access to these variable is restricted by listing them in the Flow monitor class as private. The FlowMonitor class provides a few public member functions (all functions in Figure 6.2 with a + prefix).

As shown in Figure 6.3, when an IP-ABR proxy receives a new TCP frame that is bound from source A to destination B, the proxy first retrieves the flow monitor object for the left flow (i.e., B-A) and the right flow (i.e., A-B). Let us refer to them as FlowMonitors 1 and 2. FlowMonitor 1 can estimate the RTT on the “right” side to the PEP by using Data flowing from A to B and ACKs from B to A. Similarly FlowMonitor 2 can estimate the RTT on the left side using the flow in the opposite direction.

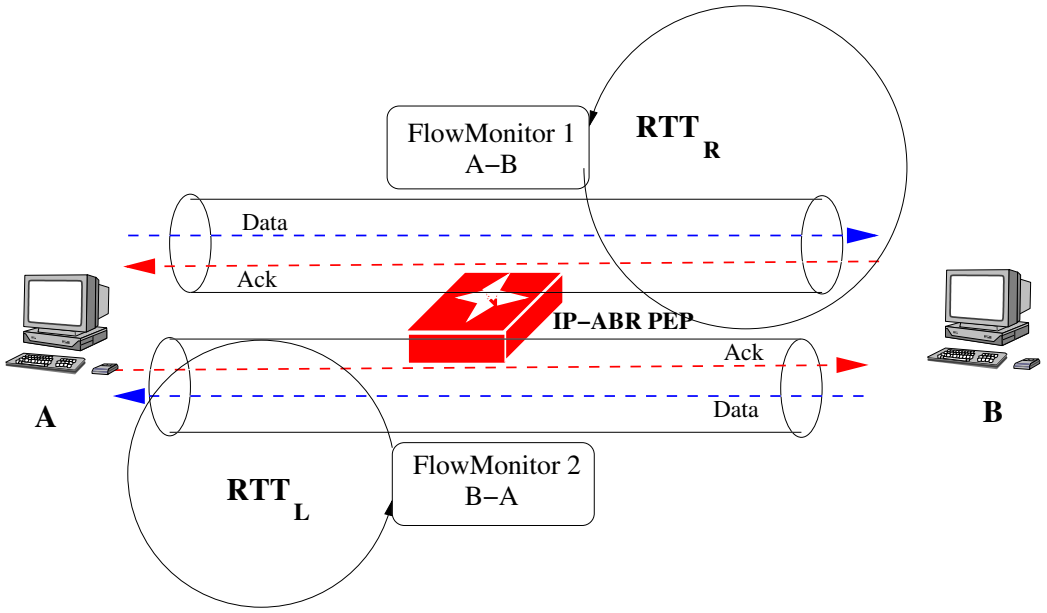

Figure 6.3: RTT estimation using FlowMonitor Objects

When a packet from host A arrives, it contains data flowing from A to B but may also contain an ACK from B to A. In order to process the packet, the PEP will first call the handle_data function within the flow monitor 1 object. If this frame contains a new sequence number, the flow monitor will make a new time stamped entry to keep track of the packet's arrival. This function then returns the last RTT measured on the right side of the PEP (RTT_R). After calling handle_data the IP-ABR proxy calls the handle_ack function within the flow monitor 2 object (i.e., B-A). The handle_ack function is passed the packet and the RTT measured on the right side of the proxy. The handle_ack function retrieves the acknowledgement number within the packet and using the acknowledgement number estimates the RTT on the downstream side is estimated. As described previously, this is done by finding the time interval between the acknowledgement arrival and the arrival of the latest data packet corresponding to the acknowledgement.

Now, the proxy estimates link latency between the end hosts by adding both left and right sides. In other words $RTT = RTT_L + RTT_R$. Using the RTT estimate the han-

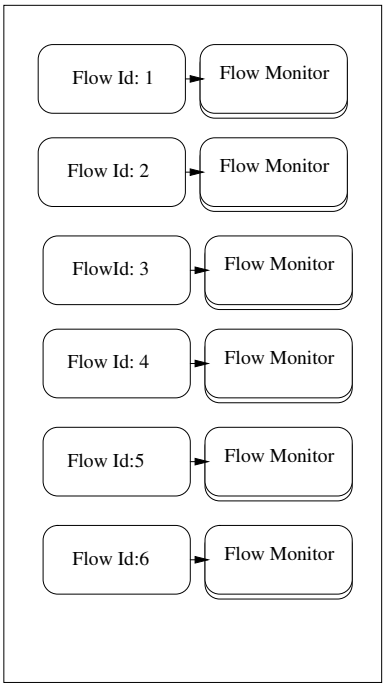

Figure 6.4: Flow Tracking Hash Table

dle_ack function computes the optimal window as described previously. If the optimal window size is less than the current window size within the packet, it is replaced with the optimal window limit, otherwise it is not.

As we have seen in this description, in order to process a frame the IP-ABR PEP needs two flow monitor objects. Whenever a new flow is encountered for which there are no flow monitor objects, the proxy instantiates, two new FlowMonitor objects. FlowMonitors objects are destroyed when a flow terminates, which occurs when the PEP detects a FIN packet, signaling connection termination.

6.2.2 Flow tracking Table

The IP-ABR PEP will usually have to manage multiple flows simultaneously. Since each flow will require two FlowMonitor objects the proxy will have to keep track of them all. To facilitate flow tracking, the IP-ABR proxy uses a hash table of FlowMonitor objects

as illustrated in Figure 6.4. A C++ standard template library "Map" class was used to implement this flow tracking table. Flows can be uniquely identified by their source address, source port, destination address and destination port. So, we use all of these to construct a flow identity, which in turn is used as key in the hash table. Therefore, each flow-Id has a one-to-one relationship with a FlowMonitor object. By making use of a flow tracking table we greatly simplify tracking and accessing FlowMonitor objects.

6.3 Packet interception in Linux

We have mentioned before that the IP-ABR PEP can be deployed on either an end host or on an intermediate router. In both scenarios, it is essential that the IP-ABR Proxy be capable of "transparently" intercepting the TCP frames. In most operating systems, access to packets is normally restricted to the kernel. However, a number of techniques are available to work around these restrictions.

1. Kernel Modules: The idea here is to design the IP-ABR PEP as a kernel module, since there are very few restrictions placed on kernel modules, because they operate in the kernel memory space. An immediate concern with taking this approach is that any instability in the module would result in crashing the system. It is also relatively more complicated to develop the IP-ABR as a kernel module, since the C++ libraries we used can not be used in the kernel. But by far the biggest disadvantage of this approach is that porting to another platform would require extensive changes to most of the program. This is not an appealing solution for an early prototype.

2. Raw Sockets: Raw sockets are feature that were first introduced in the BSD socket library. However, Raw socket implementations vary between platforms. On some platforms, access to TCP frames is not allowed [Ste97b]. However, both Linux and Windows operating systems support access to TCP frames via this interface, which

would make the code portable. However, in order to use raw sockets, source routing must be supported on the platform. This is not a feature supported by the Windows operating system.

3. Firewall API: Just like our transparent proxy, firewall programs also require access to packets passing through the kernel. On most operating systems firewall programs are implemented as kernel modules. In the past, most of these systems were mostly closed to modification by end users. However, because of the increasing complexity of rules governing firewall operation firewall programs are being made extensible. Some of these implementations provide interfaces through which user space applications can access packets. Linux provides such a mechanism in its Netfilters firewall subsystem. The advantage of using this scheme is that it allows a large portion of the code to be platform independent, with only the small portion of coding that interfaces with the firewall API requiring porting.

6.3.1 Netfilters

The firewall architecture used in Linux has dramatically changed over the years. Prior to version 2.4 of the kernel, Linux used the "ipchains" program to implement firewalls. This is similar to the implementation of ipchains on the various BSD platforms of FreeBSD, Open BSD etc. A common problem with the ipchains architecture was that it did not have a proper API that could facilitate easy modification and extension [Rus01].

The Firewall subsystem was redesigned during the development period prior to kernel version 2.4. As a result of this development, a new Firewall subsystem called netfilters was developed. The old ipchains program was replaced with a new program called "iptables". The new netfilters architecture also provided people with a new API to extend the existing functionality. One of the early extensions developed is a program called IP-

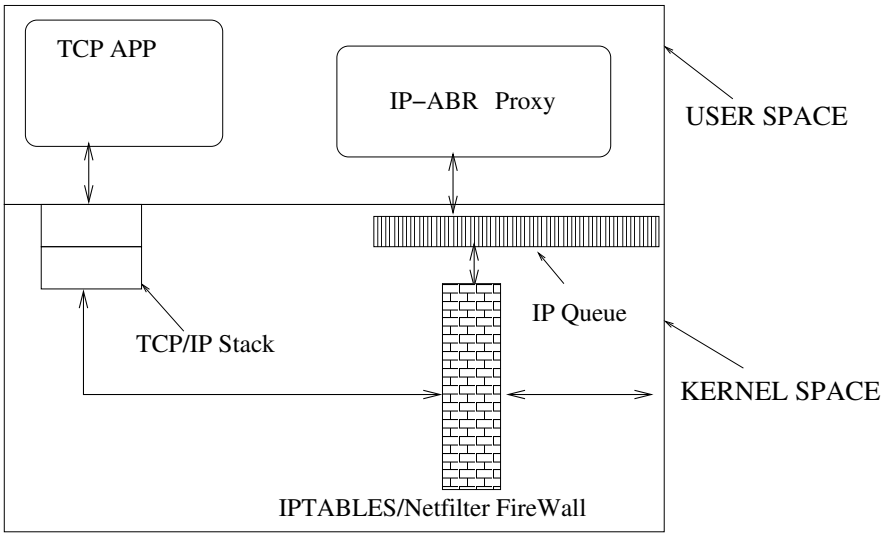

Figure 6.5: IP-ABR PEP and Netfilters

Queue. IP-Queue is a kernel module. It has been included in the kernel source since version 2.4. As illustrated in Figure 6.5 the IP-Queue module allows a user to configure firewall rules that instruct the kernel to forward packets to a user space application.

The prototype IP-ABR PEP developed uses the IP-Queue program to access TCP frames as illustrated in Figure 6.5, thereby, allowing us to operate the PEP in the user space, while at the same time being able to access packets. The Netfilters API provides a number of locations at which packets rules can be applied. Here is a list of locations where firewall rules can be applied.

1. PRE-ROUTING: This location provides access to all incoming packets.

2. LOCAL IN (INPUT): Packets destined to the local host will not be routed, therefore this is the last point to access them before they are passed to a user space application.

3. IP FORWARD: This point is before the routing decision has been made.

4. POST ROUTING: After the routing decision has been made.

5. LOCAL OUT (OUTPUT): Last point before packets are transmitted. Provides access to all out-going packets.

In our case, where we would apply these rules depends on the type of packets the PEP intends to intercept. If the IP-ABR PEP is located on an end host and is used to manage local TCP flows, then we access both the OUTPUT and INPUT channels. If the proxy is installed up on a router and is used to manage "all" TCP flows passing through it, we need only access the OUTPUT channel.

Various predefined rules exist that instruct the firewall subsystem on how to process a packet. Simple rules such as ACCEPT and DROP are used to handle packets. To redirect packets to the IP-ABR PEP we make use of the QUEUE rule. For instance, using setting the rule show below using the iptables program instructs the kernel to pass all outgoing TCP packets to an application.

```
iptables -A OUTPUT -p tcp  -j QUEUE
```

The QUEUE rule is used in conjunction with the IP-Queue module. This rule instructs the kernel to redirect the packet to a user space application for processing. However, the application exists in the user space memory and the packet in the kernel space memory and access to kernel space is restricted to the kernel and kernel modules. To overcome this restriction the packet is temporarily "copied" into user space by the IP-Queue module. The packet is queued in user space so that an application such as the IP-ABR PEP can access it. Once the packet is processed it is passed by kernel with a "verdict" issue. The verdict instructs the kernel on how to handle the packet. More specifically the verdict allows the application to instruct the kernel to either drop or allow the packet.

6.4 Window modification

After the window field in the ACK header is modified by the IP-ABR PEP one of the last things that needs to be done before transmitting the packet, is to recompute any packet header checksums. The IP header checksum does not need to be updated, since the IP-ABR does not modify IP header fields. However, the TCP checksum needs to be re-computed, since we are modifying the window field in the header. Unlike the IP header checksum the TCP checksum covers both the TCP header and the payload. This checksum is computed by padding the data block with 0, so that it can be divided into 16 bit blocks and then computing the ones-complement sum of all 16 bit blocks [1].

6.4.1 Recomputing TCP checksum

Computing the checksum over the entire length of the TCP frame, for every ACK the IP-ABR PEP modifies, is computationally expensive. However, if only a single field changes, the checksum can be recomputed incrementally as described in RFC 1624 [Rij94]. To compute a new checksum by incrementally updating the checksum one must add the difference of the 16 bit field that has changed to the existing checksum. In other words, if HC_{old} is the old checksum in the header prior to modification, then the new header checksum HC_{new} can be computed with the following equation[2]:

$$HC_{new} = \sim (\sim (HC_{old}) + (-m_{old}) + m_{new}) \tag{6.1}$$

Where m_{old} is the 16 bit field before modification m_{new} is the new value

For further clarity, let us consider the following example. Let us consider sequence 00f1, 13a4, 527b, 0041 and 7612. We can compute the ones complement sum of this

[1] This is also known as Fletcher's checksum

[2] $\sim$ is a ones complement operation

00f1	
13a4	
527b	
0041	change to 805a
7612	
dd63 = C_{old}	

Table 6.1: Checksum before modification

00f1
13a4
527b
805a
7612
5d7d = C_{new}

Table 6.2: Checksum after modification

sequence as the value dd63 as shown in Table 6.4.1. Now, let us designate the old sum as C_{old}. Now, the actual checksum in the header is the complement of C, let us designate this HC. In the case of this example $HC = \sim (C_{old}) = 229c$. Now, if we replace 0041 with 805a, then the corresponding ones complement sum is now 5d7d as shown in Table 6.2. Let us designate this as C_{new}.

Now if HC_{new} is the new checksum of the modified block. We can compute it as $HC_{new} = \sim (C_{new})$, substituting the value of C_{new}, we can see that $HC_{new} = \sim (5d7d) = a282$. However, we can also compute HC_{new} using the alternative method described in Equation 6.1.

$\sim (\sim (HC_{old}) + (m_{new} - m_{old})) = \sim (dd63 + 805a - 0041) = \sim (5d7d) = a282 = HC_{new}$

Therefore, by using the incremental update method we can recompute the checksum in 4 operations as opposed to n operations, where n (i.e., $n - 1$ addition operations, and more operation 1 for attain in the complement) is the number of 16-bit blocks that make up the checksum block.

Chapter 7

Performance Evaluation

Having implemented a prototype IP-ABR performance enhancing proxy (PEP), our next task was to evaluate its performance. Specifically we were interested in seeing how an IP-ABR service implemented using the PEP, would succeed in reducing throughput variability, decreasing packet loss, lowering delays and jitter. In this chapter, we provide a detailed account of all the tests conducted on the proxy.

In order to test the IP-ABR PEP, we conducted a series of tests in a test network which had a large bandwidth delay product, similar to a satellite IP network. In these tests, we compared the performance of regular TCP service to IP-ABR service implemented with the prototype PEP, by comparing metrics such as throughput variability, packet loss and delays. We also used two different queuing disciplines, drop-tail and Derivative Random Drop [Gay96].

7.1 Test Environment

In this section we provide an overview of the test environment used for testing the prototype IP-ABR PEP.

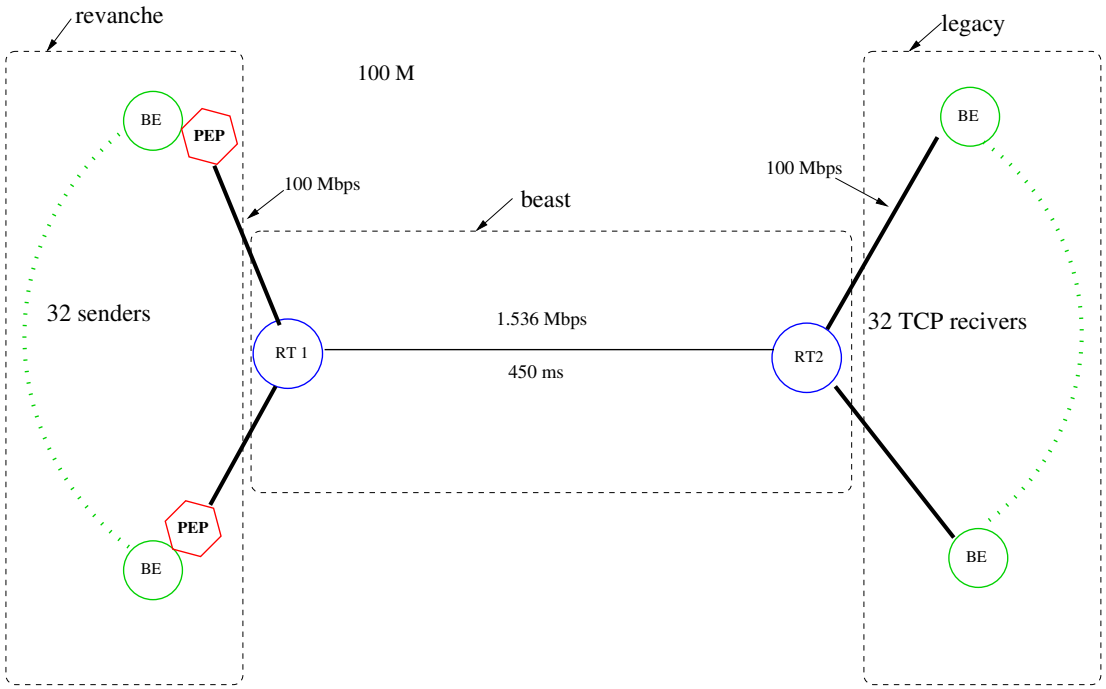

Figure 7.1: Test Bed Emulated Topology

7.1.1 Test Topology

Figure 7.1 depicts the network topology that was used to design a testbed for testing the IP-ABR PEP. The test network topology is similar to that used in the NS simulations (Figure 5.1) conducted earlier. However, instead of using multiple end hosts, one for each TCP flow, in this topology we have two PCs on either end of an emulated satellite link. We tested the IP-ABR PEP in a network similar to a satellite network, since we are interested in using IP-ABR service to improve the service quality in networks with large bandwidth delay products, such as satellite IP networks. The end hosts are connected to a satellite link via a router; the link between the end hosts and the router is 100Mbps Ethernet link. The bandwidth of the satellite link is that of a T1 link (i.e., 1.536 Mbps), therefore it serves as a bottleneck link.

A problem with implementing this testbed, is that it calls for connecting both hosts

Source	Destination	BW Bytes/sec	Delay ms
sender	receiver	192000	450
receiver	sender	192000	450

Table 7.1: NISTnet emulator configuration

over a satellite link, which is not is not easily accessible in our lab environment. However, instead of using an actual satellite link to connect the hosts, we can emulate the link using a network emulator.

7.1.2 Satellite Link Emulation

In order to emulate the satellite link in our test-bed, we used the NISTnet network emulator, that was developed by the National Institute of Standards and Technology. NISTnet can emulate link delays and bandwidth constraints. NISTnet also provides various queuing disciplines such as Derivatice Random Detection (DRD) and Explicit Congestion Notification (ECN). The NISTnet software is available for the Linux operating system at the NIST website [http://www.antd.nist.gov/tools/nistnet/].

The NISTnet emulation program is a Linux kernel module. NISTnet is usually deployed on an intermediate router between two end hosts. Once installed, the emulator replaces the normal forwarding code in the kernel. Now, instead of forwarding packets as the kernel normally does, the emulator buffers packets and forwards them at regular clock intervals that correspond to the link rates of the emulated network. Similarly, in order to emulate network delays, incoming packets are simply buffered for the period of the delay interval.

The NISTnet emulator acts only upon incoming packets and not upon outgoing packets, therefore, in order to properly emulate a network protocol such as TCP, which has bidirectional traffic flow, the emulator must be configured for traffic flowing in both directions as shown in Table 7.1. To emulate a network with specific bandwidth and delay

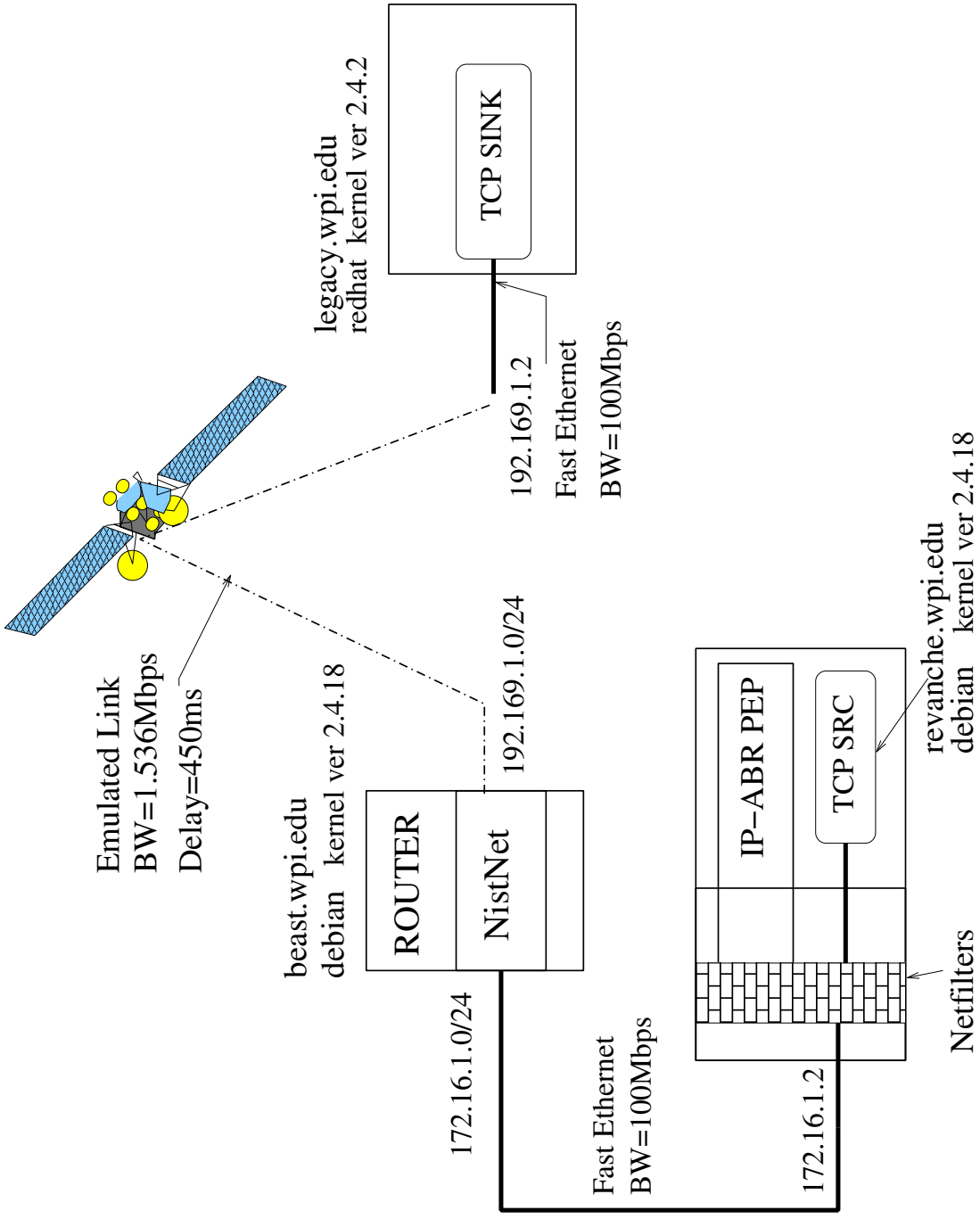

Figure 7.2: IP-ABR PEP Test bed

characteristics, we specify the IP address of the source and destination end hosts, and we the bandwidth and delay of the emulated link between the end hosts. NISTnet also allows either of two queuing disciplines to be specified: Derivative Random Drop (DRD) or Explicit Congest Notification (ECN).

Note that in Figure 7.1 we have two routers, RT1 and RT2, on either end of the satellite link, where RT1 is the router closer to the sender and RT2 is the router closer to the receiver. However, in our configuration we use a single router instead of two as depicted in Figure 7.2. The one router in the middle is configured to emulate the bandwidth constraint of the satellite link, but is not configured to emulate delays. However, each end host has a NISTnet emulator that is configured to emulate the satellite link delay for incoming packets. In this configuration the single router in the middle appears to be RT1 to the sender and RT2 to the receiver. Because congestion does not occur on the router located on the far side of the link, we do not need to emulate the routing queue of RT2.

7.1.3 Test Equipment Configuration

In order to emulate the test environment shown in Figure 7.1, we only require three machines, therefore, we used three PCs with the Linux operating system to conduct our tests. Table 7.2 details the configuration and purpose of each of these machines.

Two of the machines served as the end hosts in a TCP connection, and the third machine was used as router. The end host machines were connected to the router host with a 100 Mbps fast Ethernet LAN as shown in Figure 7.2. The end host “revanche”, acted as a TCP sender and was therefore on the congested side of the link. The IP-ABR PEP also resided upon this host. The NISTnet emulation program was installed on the router box “beast” and is configured to emulate the satellite link bandwidth for traffic going in either direction. The third PC (legacy) is used as the receiver end host. As mentioned previously, each end host also has a NISTnet emulator configured to emulate the delay of

Hostname	revanche.ece.wpi.edu	beast.ece.wpi.edu	legacy.ece.wpi.edu
Purpose	End Host/TCP source	Router	End host/TCP sink
Processor	AMD Athlon MP 1800+	AMD Athlon 1.1 GHz	Pentium 2 (400 MHz)
OS	Linux (Debian)	Linux (Debian)	Linux (Red Hat 7.3)
TCP Version	NewReno	NewReno	NewReno

Table 7.2: Test Equipment Configuration

the satellite link.

7.1.4 Test traffic generation

In our tests we create multiple TCP flows that compete for the bandwidth of the shared bottleneck link. Using the Python programming language we developed two programs, one for the sender/client side, and, the other for the receiver/server side. The sender side program is used to spawn n concurrent threads. Each thread makes a request for a socket connection from the receiver/server side. When the receiver/server gets this request, it spawns a corresponding thread to service that sender/client. Once the connection is established, the sender will continuously send MSS size blocks of data to the receiver by writing to the socket buffer as fast as possible. By keeping each TCP data buffers full, we ensure that TCP flows compete against each other by trying to gain the maximum possible bandwidth.

7.2 Drop Tail

We have been proposing to create an IP-ABR service to improve the service quality in satellite IP networks. As previously described, this is done by using a IP-ABR PEP to dynamically window limit TCP flows, so that it does not exceed the available bandwidth

which is determined by a bandwidth management service (BMS), thereby preventing congestion and creating self paced flows which have minimal throughput variations and enjoy better QoS than regular TCP flows. The tests described in next few sections verify the IP-ABR PEP's performance in this regard.

7.2.1 Description

As mentioned previously, in all of the tests conducted, we generated traffic by using 32 TCP flows, which compete for the shared single bottleneck link. The 32 concurrent TCP flows transfer data in one direction, thereby, asymmetrically loading the link, therefore ACKs do not experience queuing delays.

There are a number of queuing disciplines that influence TCP's performance and, of these the NISTnet environment allows us apply Drop Tail, DRD and ECN queuing disciplines on the router's queue. So which queuing discipline do we use choose for our tests? We first tested using a simple drop tail queuing discipline. However, we also conducted a set of tests with DRD later. NISTnet does not directly support drop-tail. However, by setting the minimum and maximum limits of DRD to coincide we can achieve a 95% drop probability when the queue size reaches the limit. This is good approximation of drop-tail behavior.

Using a drop-tail queuing discipline on the router, we conducted two tests. In both tests we had 32 TCP flows competing for the bandwidth of the satellite link. In the first test we used ordinary TCP flows. In the second, test we created an IP-ABR service using the prototype PEP. The PEP was configured to equally distribute the link's bandwidth amongst the flows. As seen in the early NS simulations, the IP-ABR service implemented using the PEP should result in fair, non-varying throughput with low packet loss, delay and jitter.

The first metric used in comparing IP-ABR to regular TCP is throughput variability.

However, the application throughput achieved at the receiver is of more importance than the sender's achieved throughput. The receiver side application throughput is sometimes referred to as "goodput". In order to compare throughput variability between IP-ABR and TCP, we measure goodput and plot it against the duration of the test. As mentioned earlier TCP traffic is bursty in nature, this results in highly variable instantaneous throughput. We avoid plotting this instantaneous throughput and instead use the technique previously developed to estimate average throughput in our NS simulations, wherein we use a 5 sec. running window over which we determine the average throughput.

7.2.2 Configuration

The following is a list of test parameters used for the drop-tail tests.

1. Local link bandwidth = 100 Mbps
2. Bottleneck link bandwidth = 1.536 Mbps
3. Bottleneck link one way latency = 450 ms (emulated with NISTNET in both direction)
4. Queuing disciplines: drop-tail.
5. Max drop-tail queue size = 15,30 packets
6. MTU = 1500 Bytes, MSS = 1460 Bytes (This is the maximum MSS size that can be used on Ethernet and is the default size on Linux).
7. Flow Duration for test = 120 secs.

Metric	TCP	with IP-ABR
Mean throughput of a randomly selected flow	66.5591 kbps	48.50 kbps
Standard deviation of throughput for randomly selected flow	25.09 kbps	2.258 kbps

Table 7.3: Comparison with and without IP-ABR, using drop-tail queue limit 15 packets.

7.2.3 Observations on throughput variability tests

In the first test, we consider the plot of the throughput of the 32 TCP flows for the duration of the test. From Figure 7.3 we can see that the throughput is highly variable. This is expected and is similar to what we have seen in early simulations of TCP in NS (Figure 5.2). As explained earlier, these oscillations in throughput are caused by the current congestion control mechanism used in TCP. More specifically the linear growth of the TCP window during periods of congestion avoidance will cause periodic queue overflows on the bottleneck router. The resulting packet drops will in turn trigger TCP congestion avoidance behavior. This periodic breakaway from steady state causes oscillations in throughput. We expect the IP-ABR service to eliminate these oscillations.

We repeat the test with the IP-ABR PEP enabled on the end host. The IP-ABR PEP is configured to equally allocate the available bandwidth amongst all 32 competing flows. In other words, each flow is allocated a bandwidth of 6000 bytes per second or 48 kbps. As in the previous test, we plot the throughput for each of the IP-ABR service flows over the duration of the test. As can seen from Figure 7.4, the TCP throughput of IP-ABR service traffic exhibits less variation when compared to the throughput of regular TCP traffic. In order to compare the two, we picked a random flow and estimated the standard deviation of the flow's throughput about the mean throughput for the duration of the test as shown in Table 7.3. However, this is not an ideal comparison, since in the case of regular TCP some flows exhibit more variation in throughput than others, but it does provide us with

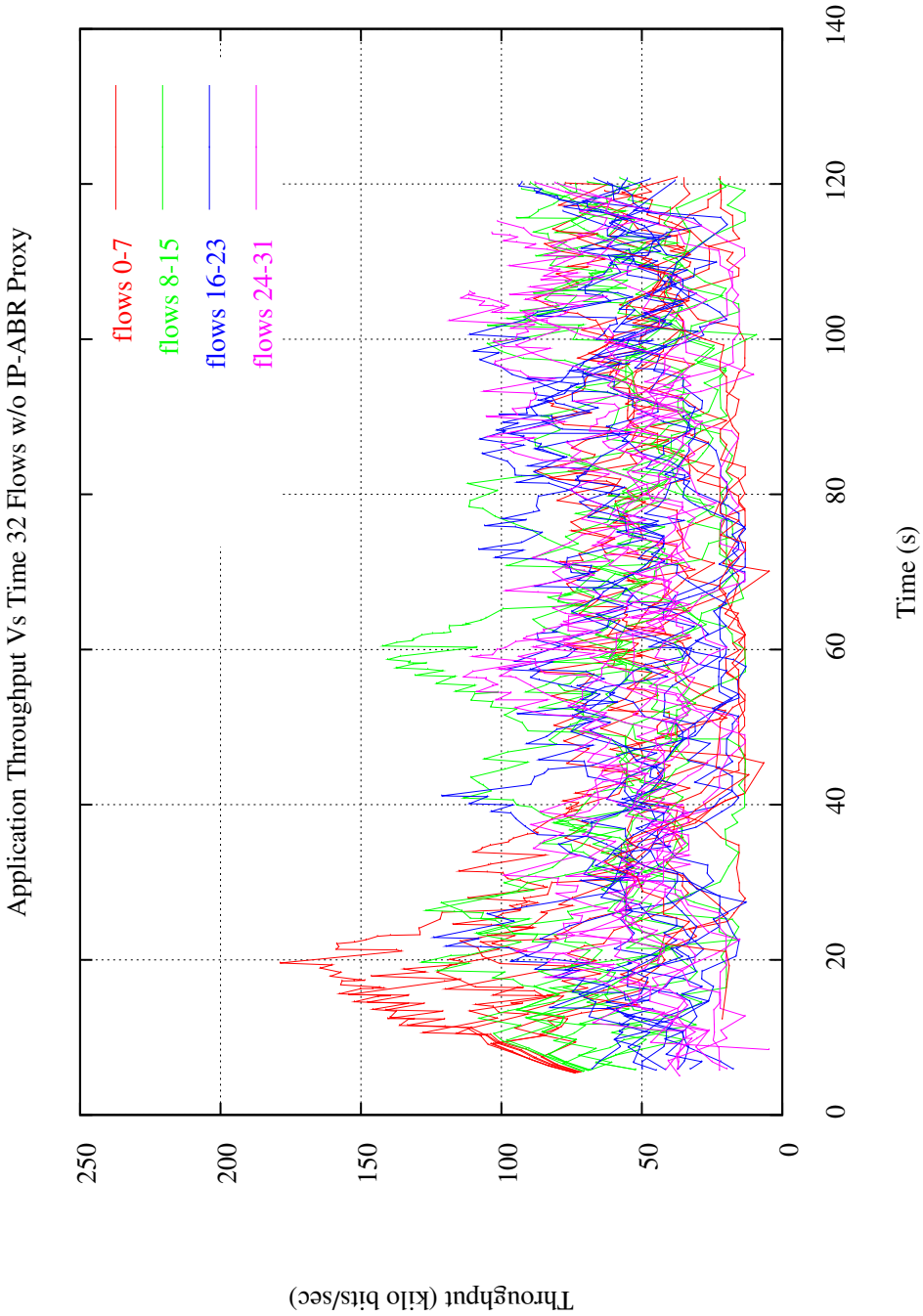

Figure 7.3: Throughput variability for 32 TCP flows, using a drop-tail queue of size 15 packets.

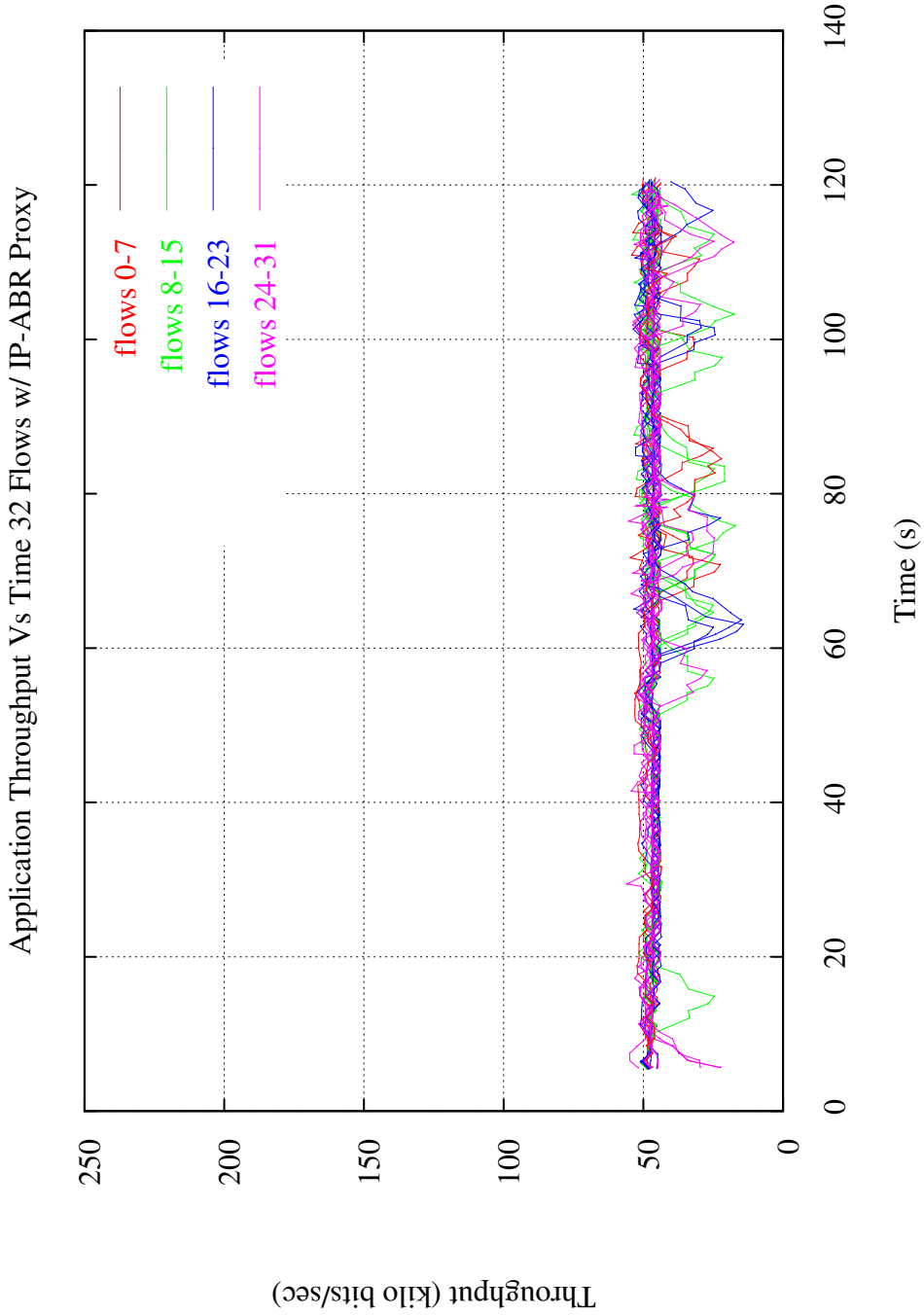

Figure 7.4: Throughput variability for 32 IP-ABR flows, using a drop-tail queue of size 15 packets.

an idea of the reduction in throughput variability achieved by using an IP-ABR service.

We notice that in the case of IP-ABR service traffic, the window limitation executed by the IP-ABR PEP, lowers the upper bound of the throughput oscillations. However, the throughput seems to drop at irregular intervals. As we have seen in early NS simulation tests, if the bandwidth is allocated in such a manner, that it does not exceed the bottleneck link, then each flow is induced into self-pacing mode for the duration of the connection. It is possible that the queue size used in this test is insufficient to accommodate the congestion caused by the bursty TCP traffic, thereby leading to packet loss. However, in order to substantiate this claim, we need to estimate the packet drops during the test.

7.2.4 Observations on QoS metrics

Until now, we have just compared the IP-ABR service to regular TCP, by looking at the throughput variability. However, in addition to throughput variability there are also other QoS metrics such as packet loss and delays which we have not compared yet. However, we can not measure these QoS metrics by measuring delay and packet loss of IP packets, because TCP uses an automatic request repeat (ARQ) mechanism in order to recover from packet loss. This allows TCP to provide a reliable transport service to the application. Therefore, an application using TCP never experiences packet loss directly. However, because of the ARQ mechanism used in TCP, packet loss will manifest itself in the form of excessive packet delays and a drop in throughput. In other words, the delay and jitter that TCP application experience are related retransmission delays.

In order to properly measure QoS metrics such as packet loss, delay and jitter for TCP service traffic we define a new metric, that we call “segment delivery latency” (SDL). The segment delivery latency is the time it takes for a segment to be successfully delivered to the destination from the time it was “first” transmitted by the sender. The segment delivery latency should be equal to the sum of link latency, queuing delay and retransmission delay.

Therefore, the SDL should quantify the delivery delay experienced by an application that is using TCP as a transport service.

$$SDL = latency_{link} + delay_{queue} + delay_{retransmisssion}. \tag{7.1}$$

In order to estimate segment delivery latency (SDL) we measure the time difference between when a TCP segment is "first" transmitted and when the first corresponding ACK is received at the sender. Let us denote this time as $\delta_{segtime}$. If there are minimal queuing delays and drops on the ACK channel, we can approximate segment delivery latency by $\delta_{segtime} - latency_{link}$. The minimum segment delivery latency experienced will be equal to the link latency, because under such circumstances there niether is queuing delay nor is there retransmission delay. We can normalize the segment delivery latency by subtracting the link latency from it. Let us refer to this normalized segment delivery latency as excess segment delivery latency (ESDL). Where ESDL is thus given by the following formula.

$$ESDL = \delta_{segtime} - (2 \cdot latency_{link}) \tag{7.2}$$

In the case of our test, we captured TCP traffic at the sender host using the "TCPdump" packet capture program. TCPdump provides a time-stamp of packet arrival and departure times. We then used another program that could use this captured data to provide us with ESDL estimates for each segment. Now, by estimating the mean ESDL values for each flow we can obtain a measure for the mean delay experienced by the receiving side application. In addition to this, we can use the largest ESDL estimate as a measure of the "delay jitter" experienced by the application, since delay jitter is defined as the maximum variation in delay [Sta02].

When a segment is not dropped/lost the ESDL it experiences is equal to just the queuing delay experienced due to all routers between end hosts. If all segments are of a

Metric	TCP	with IP-ABR
Segments with a delivery latency $\geq$ 117ms	6.9 %	0.5%

Table 7.4: Delivery latency comparison with and without IP-ABR, using a drop-tail queue size of 15 packets.

constant size, then, the maximum theoretical queuing delay experienced by a segment in our test can be estimated approximately as the queue size in terms of packets divided by the satellite link bandwidth in terms of packets per second. Now, given the maximum queuing delay we can assume that any packet with a ESDL greater than the maximum queuing delay is a dropped packet, since it is experiencing retransmission delay, thereby providing us with a means by which we can estimate packet drops and losses.

$$QueuingDelay_{max} = \frac{QueueSize_{max}}{Bandwidth_{link}} \tag{7.3}$$

In Figure 7.5 we show the excess segment delivery latency (ESDL) measured for each segment transmitted (denoted by the red dots), plotted against the duration of the test. In this plot the green trace shown is the ESDL experienced by a single randomly selected flow. As mentioned before, any segment that has been retransmitted will have an ESDL which will be equal to the sum of queuing delay and retransmission delay. Similarly for segments that do not have to be retransmitted their ESDL will be equal to only the queuing delay. We can use this fact to determine the number of segments that have been retransmitted.

The maximum queuing delay for a drop tail queue size of 15 packets is approximately 117 ms. From visual observation we can see that in the case of TCP service traffic there are a large number of segments that have an ESDL greater than the maximum queuing delay of 117 ms with a maximum ESDL of approximately 3.5 secs. By comparing the ESDL of each segment to the maximum queuing delay, we estimated that approximately

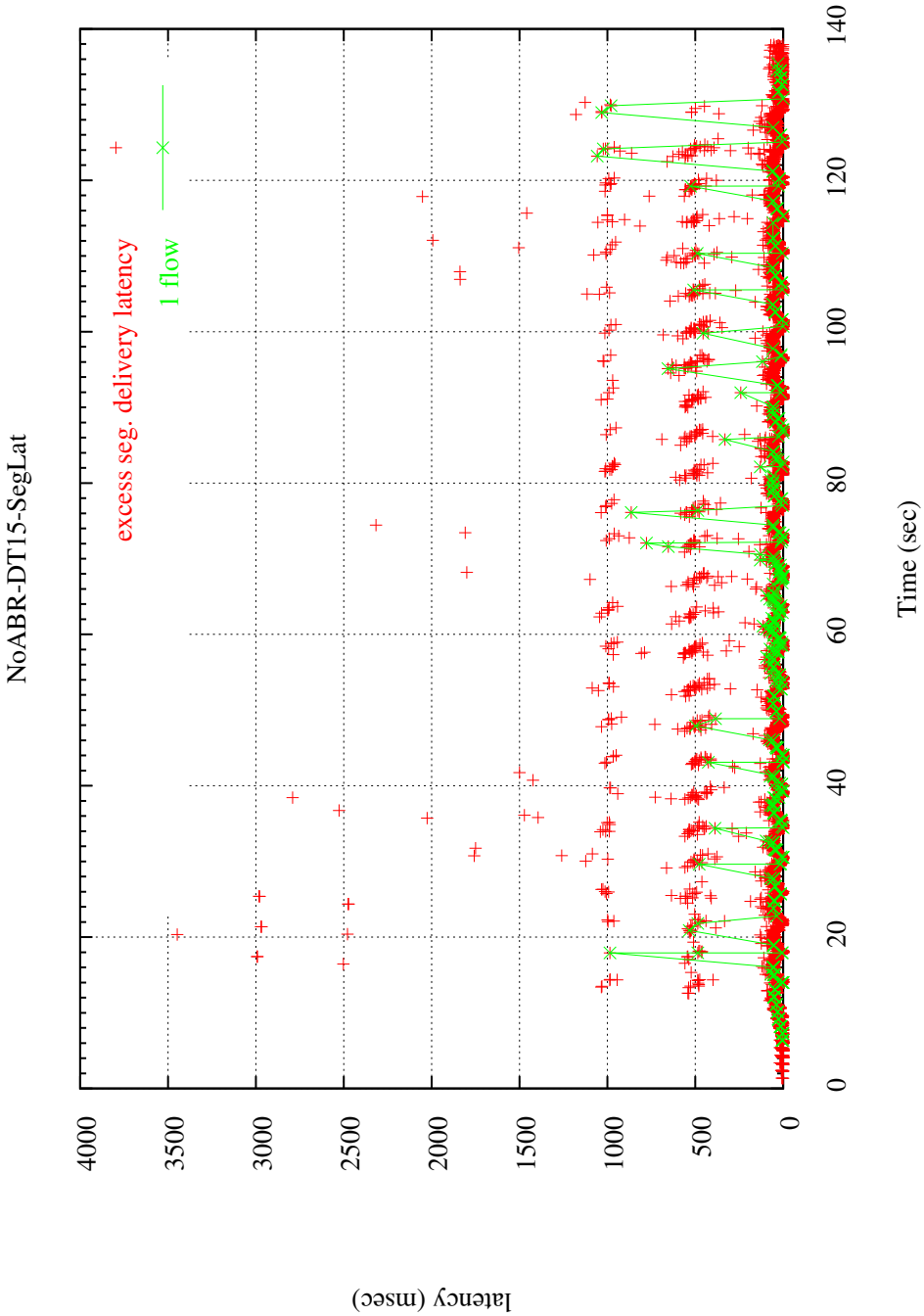

Figure 7.5: Excess segment delivery latency vs. test duration (32 TCP flows, using a drop-tail queue of size 15 packets).

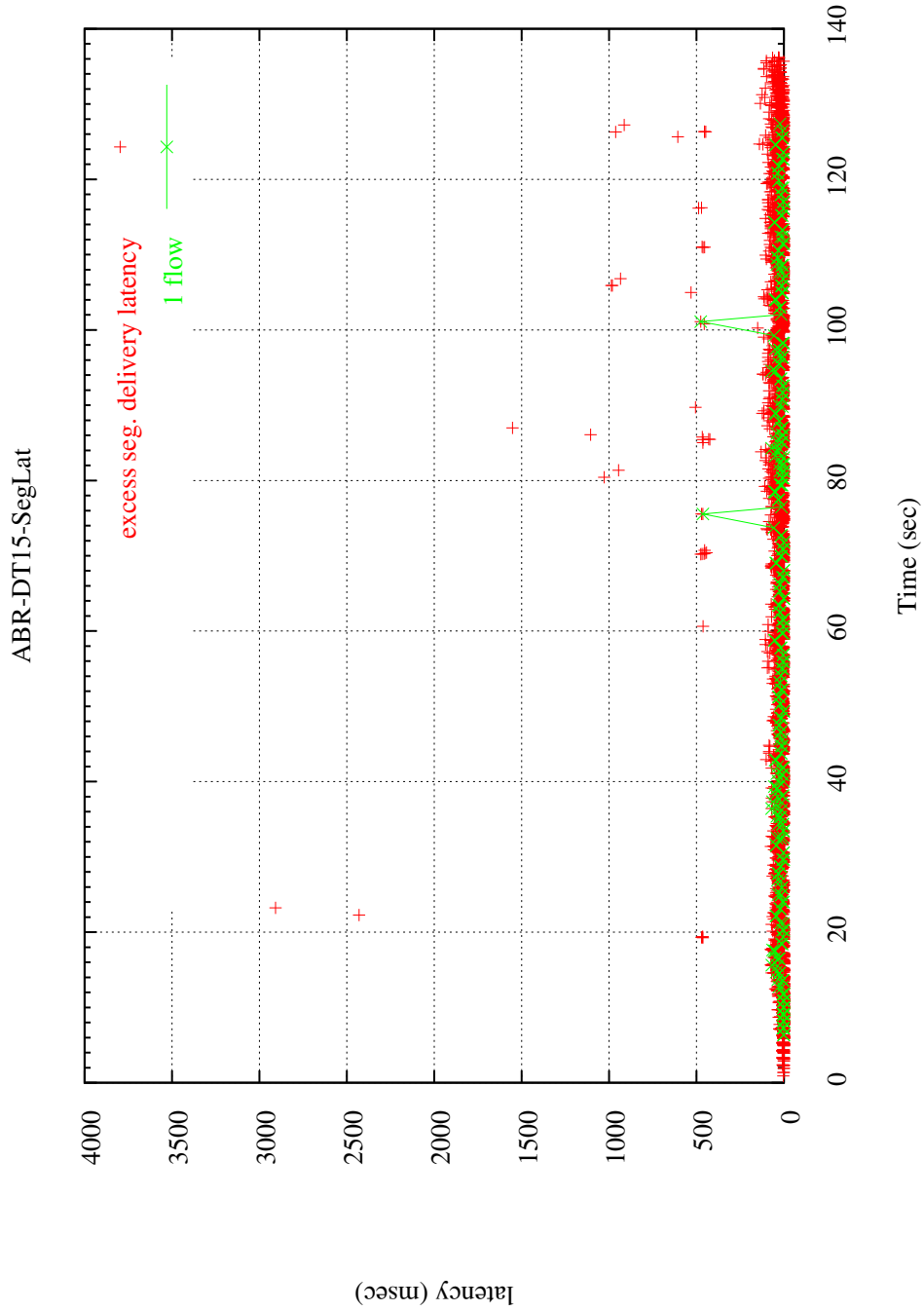

Figure 7.6: Excess segment delivery latency vs. test duration (32 IP-ABR service flows using a drop-tail queue of size 15).

Metric	TCP	with IP-ABR
Mean Throughput of a Random Flow	49.672 kbps	49.19 kbps
Standard Deviation of Throughput for a random flow	25.16 kbps	2.6 kbps

Table 7.5: Throughput comparison with and without IP-ABR, using a drop-tail queue size of 30 packets.

6.9% of segments transmitted get dropped at least once with the TCP service. As shown in the Table 7.4 in comparison to 6.9% for TCP traffic, only 0.5% of packets get dropped due congestion with the IP-ABR service.

7.2.5 Increasing drop-tail queue limit

From the results of the previous test we can see that by enabling the IP-ABR service there is a definite reduction in packet drops due to congestion. However the drop tail queue size is still too small to prevent all packet drops in the IP-ABR service case. Since all flows are rate limited by the proxy, any drops that trigger TCP congestion avoidance will lead to under utilization of the channel. Hence we need to eliminate these packet drops. However, can we further reduce it by increasing the queue size? To test this assumption we conducted one more drop-tail test, wherein we doubled the queue limit on the bottleneck router to a size of 30 packets.

Figure 7.7 shows the throughput of regular TCP traffic when we use a larger queue on the router. As in the previous test, we see large variations in the throughput of TCP traffic. The larger queue size has not made any difference on regular TCP traffic. Figure 7.8 shows the throughput of the IP-ABR service traffic. This time, by using a larger queue size, we see that there is negligible variation in the throughput of the IP-ABR traffic. We can also see that all drops in throughput noticed in the previous test have been eliminated. However, have packet losses and retransmission delays been eliminated? To answer this

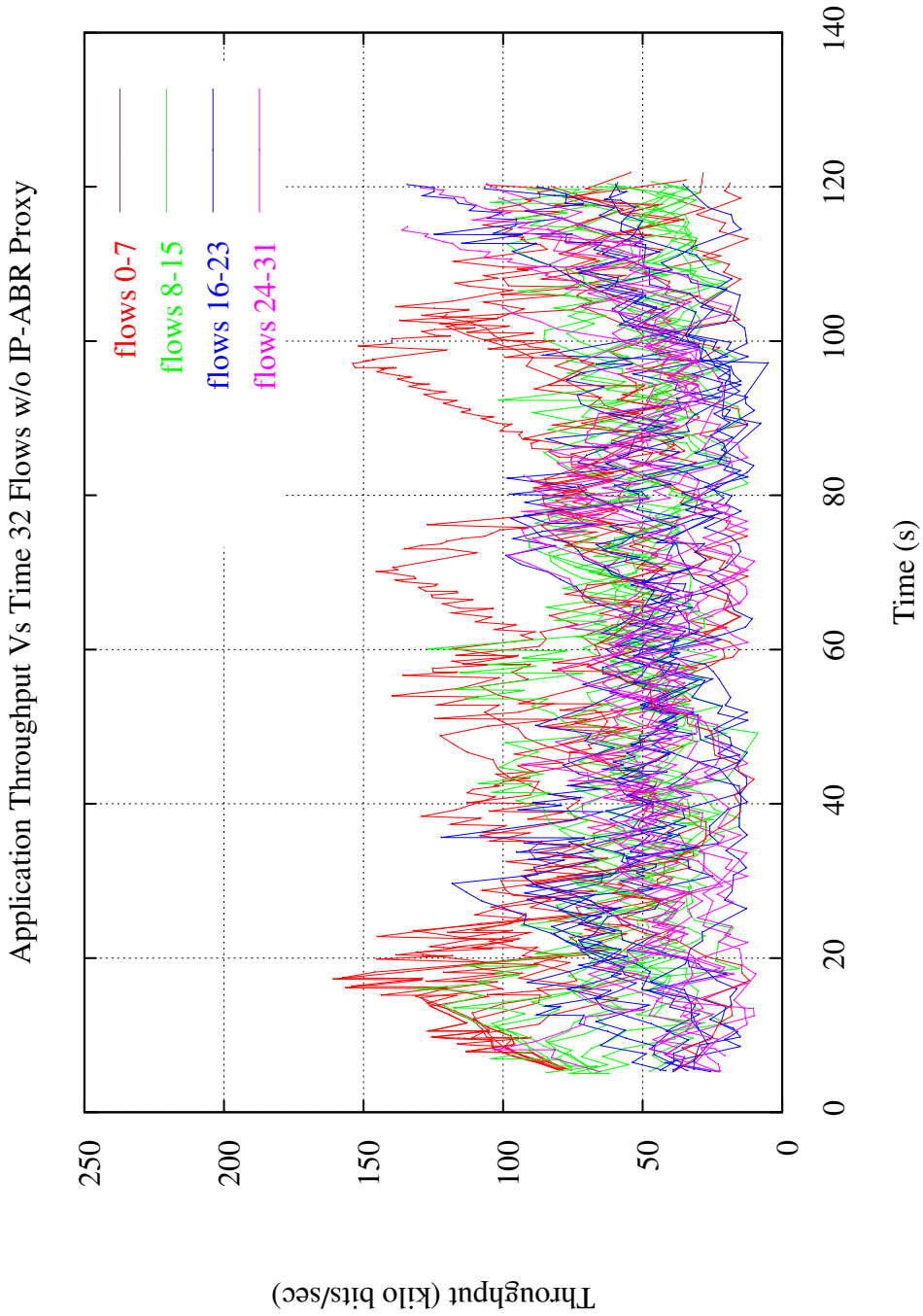

Figure 7.7: Throughput variability for 32 TCP flow without IP-ABR service, using a drop-tail queue of size 30 packets.

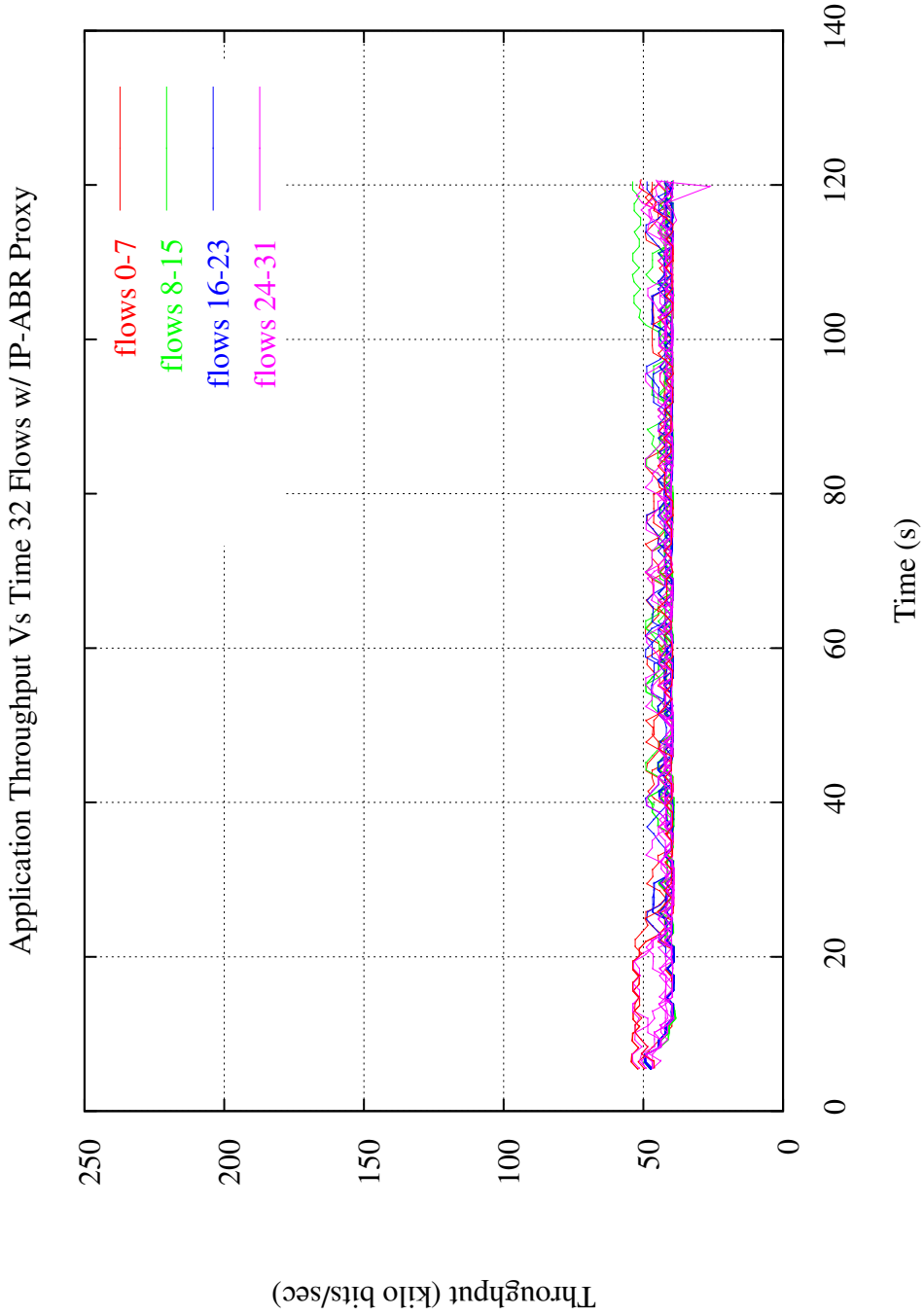

Figure 7.8: Throughput vs. variability for 32 IP-ABR flows, using a drop-tail queue of size 30 packets.

Metric	TCP	IP-ABR
Segments with a delivery latency $\geq$ 117ms	12 %	0.07%
Segments with a delivery latency $\geq$ 234ms	6.7 %	0%

Table 7.6: Delivery latency comparison with and without IP-ABR using a drop-tail queue size of 30 packets.

Metric	TCP	IP-ABR
Mean ESDL	134 ms %	30.9 ms%
Standard Deviation of ESDL	1137.0 ms %	19.7 ms%

Table 7.7: ESDL for a single random flow comparison of TCP and IP-ABR using a drop-tail queue size of 30 packets.

we plotted the measured ESDL as was previously done.

In Figure 7.9, shows a scatter plot of the measured ESDL over the duration our test for each transmitted segment by regular TCP service. Similarly, Figure 7.10 shows the ESDL experienced for IP-ABR service traffic. Now, since the queue size was doubled to 30 packets in this test, the maximum theoretical queuing delay limit used to estimate the number of retransmitted segments also doubles to 234 ms instead of 117 ms. Table 7.6 shows a comparison of the packet drop counts for the IP-ABR service versus the TCP service. We can see that by using a larger queue the retransmissions have been completely eliminated for the IP-ABR service. By doubling the drop tail queue limit from 15 to 30 on the bottleneck router, the packet drops were reduced from 0.5% to 0. The larger queue has not helped TCP service, since the packet drops seem to be approximately the same (6.7% with the larger queue as opposed to 6.9% with a smaller queue size). In fact, the large queue size has increased the queuing delays experienced, since around 12% of the segments transmitted experience a delay of 117 ms or more, as opposed to just 6.9% of packets when we were using a queue of half the size.

From Figure 7.10 we can see that the queue is sufficiently large enough to accommo-

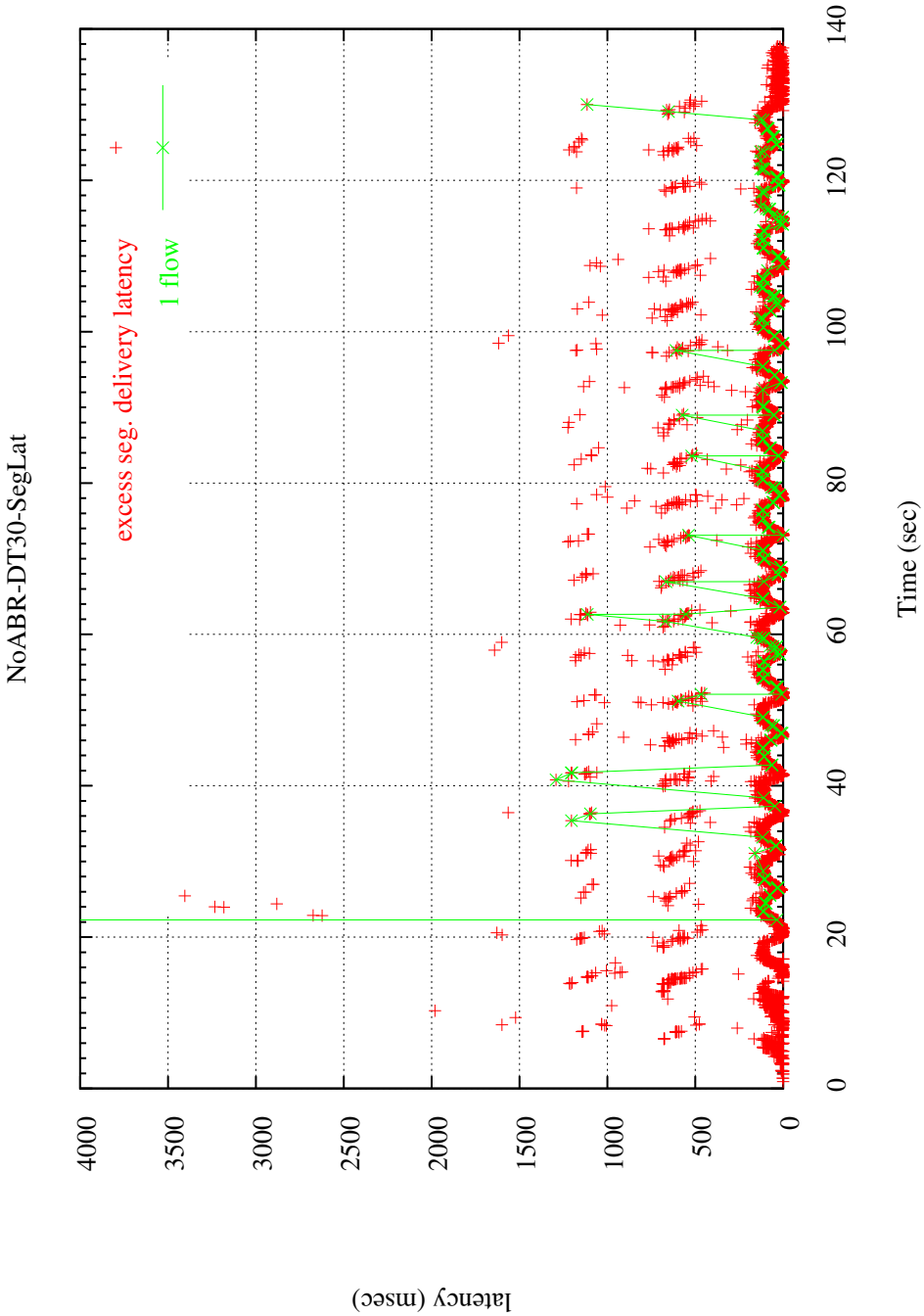

Figure 7.9: Excess segment delivery latency vs. test duration (32 TCP flow using a drop-tail queue of size 30 packets).

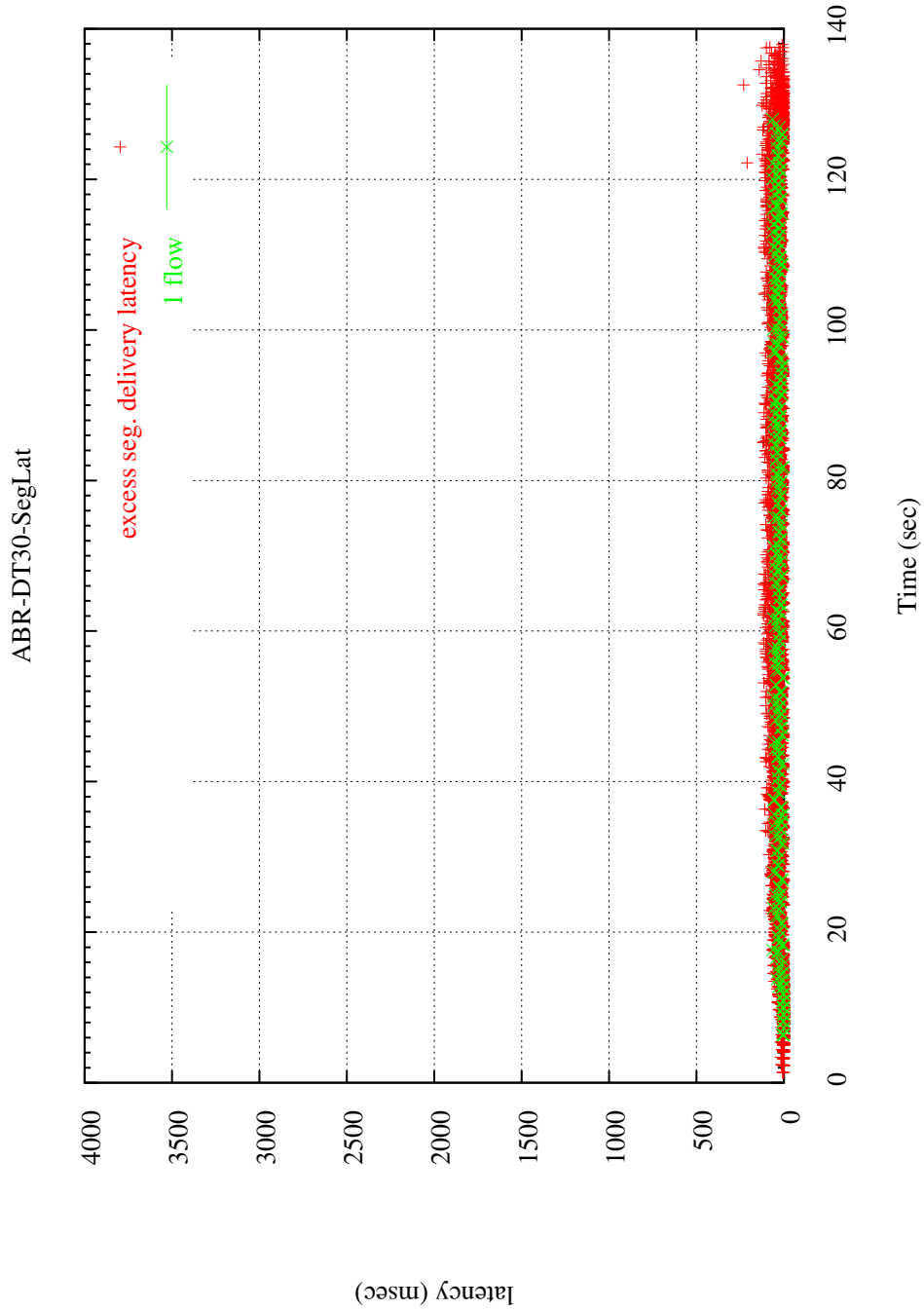

Figure 7.10: Excess segment delivery latency vs. test duration (32 IP-ABR service flows using a drop-tail queue of size 30 packets).

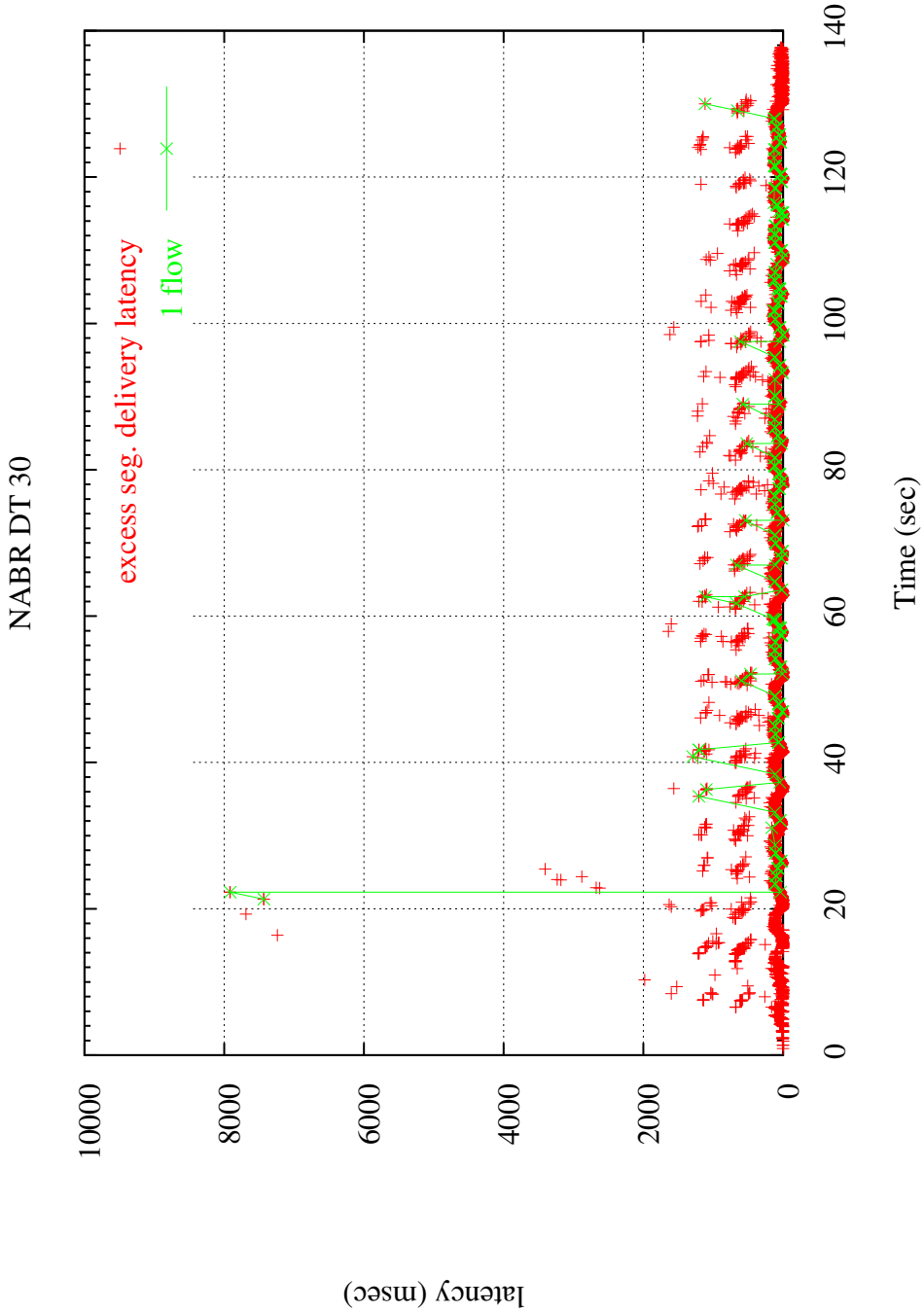

Figure 7.11: Delay Jitter experienced by regular TCP (32 TCP flow using a drop-tail queue of size 30 packets).

Metric	TCP	IP-ABR
Largest ESDL	7920.770 ms	229.824 ms

Table 7.8: Maximum delay jitter (max ESDL) TCP vs IP-ABR (Drop-tail = 30 packets).

date the 32 IP-ABR flows This allows us to properly compare the delay jitter experienced by regular TCP traffic and IP-ABR traffic. As explained earlier in order to compare "delay jitter", we compare the maximum ESDL experienced by any one flow. In order to clearly see the larger ESDL estimates which fell outside the plot range of Figure 7.9, we re-plotted the same ESDL estimates with a large range in Figure 7.11. In this plot we can clearly see the largest ESDL estimates for any of the 32 TCP test flows. Comparing these values to the estimates for IP-ABR traffic in Figure 7.10 we can see a significant drop in delay jitter. Table 7.8 also provides a comparison of actual worst case ESDL estimates.

By plotting the previous two measures of ESDL shown in figures 7.9 and 7.10 again in Figures 7.12 and 7.13 over a smaller range (i.e., between 0 and 500 ms) we provide a better view of the actual queuing delays experienced with the two services.

Figure 7.12 shows the ESDL experienced by each segment in TCP service. In this plot we do not see many of the retransmitted segments, since the maximum theoretical queuing delay was estimated to be around 234 ms. However, we can clearly see how queuing delay varies with time. We can notice a oscillation pattern in the lower ESDL values for TCP service. This repetitive rise and fall in delay can be attributed to TCP congestion control mechanisms. The queueing delays grow because of the linear growth of TCP transmission windows. This leads to increased congestion and therefore longer queuing delays; the queue delay keeps growing until the queue overflows, which is followed by a drop in queuing delays, which, as we know, is caused by the end hosts going into congestion back-off mode. This results in some segments with excessively long delays/ESDL due to the extra time involved in retransmitting the frame (packets gets retransmitted only after

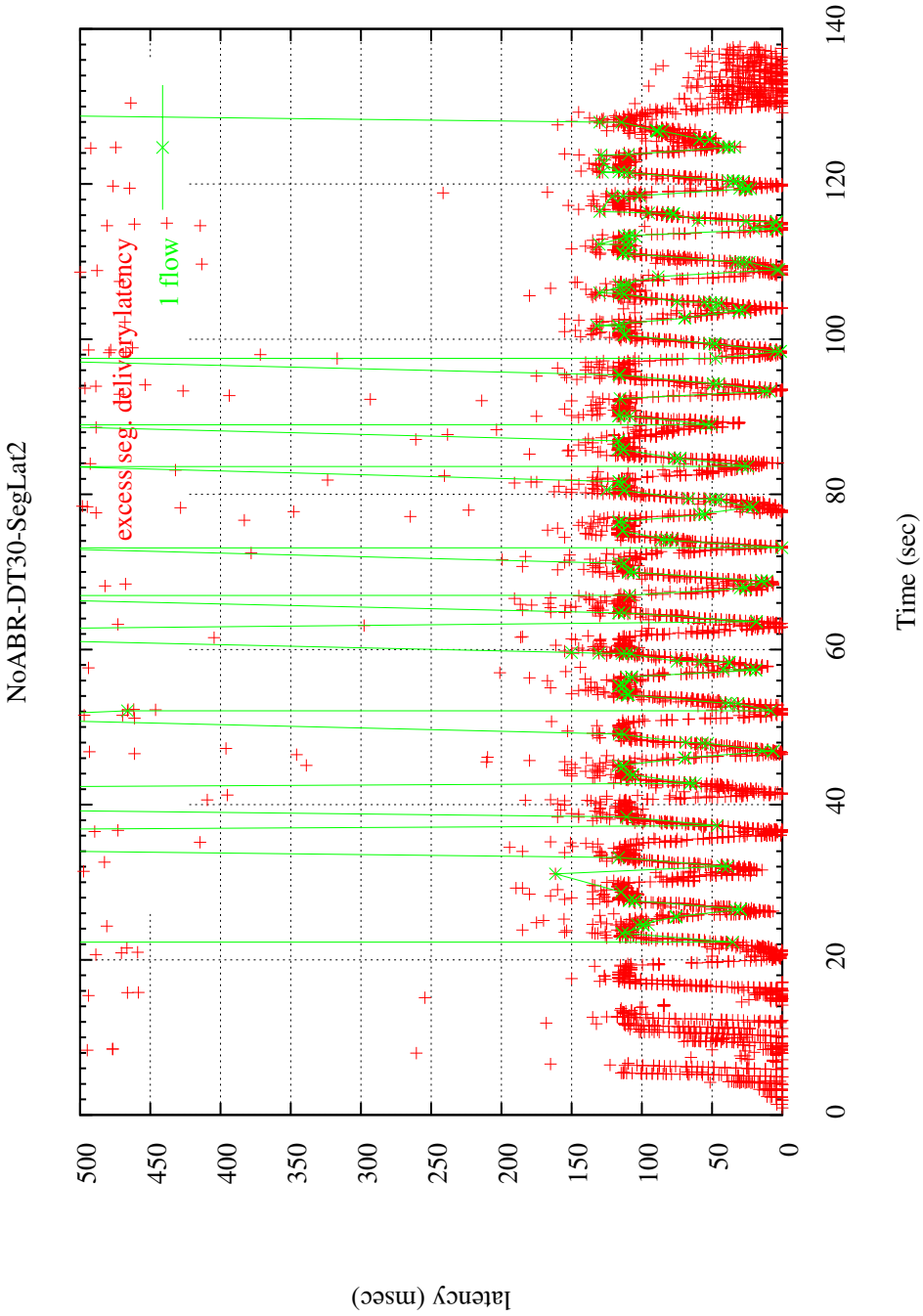

Figure 7.12: Closeup view of excess segment delivery latency vs. test duration (32 TCP flow using a drop-tail queue of size 30).

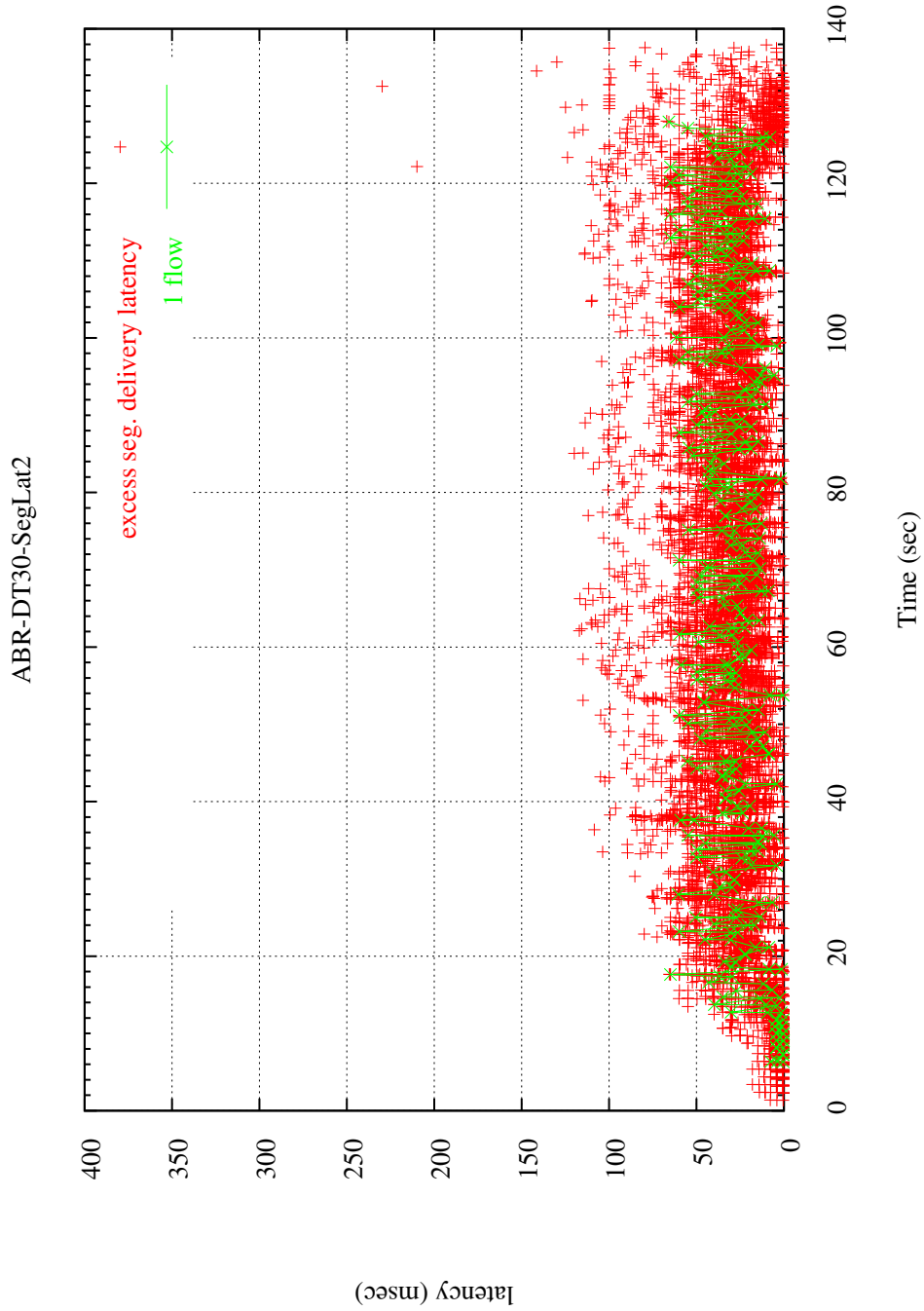

Figure 7.13: Close up view of excess segment delivery latency vs. test duration (32 IP-ABR service flows using a drop-tail queue of size 30).

the RTO times out, given the large RTT for these networks lead to large retransmission delays). The larger ESDL estimates for TCP traffic shown in Figure 7.9 are caused by a single segment being retransmitted multiple times. In other words, the price of this congestion control mechanism is that the few segments that get dropped will have very high delays, since they have to be retransmitted.

The green trace plotted in Figure 7.12 is a trace of the ESDL estimates for a single flow. From this we can see that regular TCP's behavior leads to very large variations in ESDL estimates which in turn leads to higher jitter experienced by the application. As shown in Table 7.7, for a single randomly selected flow, we measured the mean ESDL experienced for the duration of the flow for TCP flows to be around 134 ms with a standard deviation of 1134 ms. In comparison, an IP-ABR flow had a mean ESDL of only 30.9 ms with a standard deviation of just 19.78 ms. Note however, that in the case of TCP traffic the standard deviation of ESDL estimates vary from flow to flow, therefore, the jitter experienced by applications using regular TCP will vary.

From these drop-tail tests we have shown that in the case of TCP traffic, increasing the queue size results in longer queuing delays, without a noticeable reduction in the number of packets dropped. However, when we use an IP-ABR proxy in conjunction with a large enough queue size, we succeed in regulating TCP flows and, thereby, reducing congestion and eliminating packet drops. The reduction in congestion due the regulation done by the IP-ABR PEP leads to lower queuing delays than compared to regular TCP traffic. In addition to this by eliminating packet drops, we also eliminate segment retransmission thereby significantly reducing the jitter experienced by the application.

Another point worth mentioning is that very few segments experience a queuing delay of zero. Even during congestion reduction mode, we see that the queuing delay start to rise after briefly touching zero. When a segment experiences zero delay, it means that there was no segment queued in front of it when it arrived at the router. This may appear

to be a good thing, on the contrary, no packets in a routers queue lead to underutilization of the link (which in this case is the satellite link).

7.3 DRD

7.3.1 Configuration

Until now we have used a drop tail queuing discipline on the bottleneck router. However, as we mentioned earlier, when a drop-tail queuing discipline is used on routers carrying TCP traffic it causes a phenomena called global-synchronization, wherein congestion leads to simultaneous slow-start of many flows leading to further congestion that causes the TCP stacks to cyclically reduce their throughput, thereby resulting in under utilization of the link. Queuing disciplines such as RED were developed to avoid this cyclic congestion and the excessive packet loss it causes. Therefore, using RED as a queuing discipline in our tests should result in better performance than when drop-tail is used.

However, the NISTnet emulator does not support RED; instead it uses a variation of RED called Derivative Random Drops [Gay96]. DRD is was designed to improve RED performance in high speed networks at lower computation cost compared to the previous mechanism. However, the behavior of DRD is similar to existing RED implementations in the sense that packets get dropped from the queue with increasing probability as the queue length increases. So, using DRD should meet our goal of using a more proactive queuing disciple than drop-tail to prevent global-synchronization. In our DRD test we configured the derivative random drop (DRD) policy so that the minimum threshold was set to 15 and maximum threshold was 40. The rest of the test parameters were not changed from the previous tests.

Metric	TCP	with IP-ABR
Mean Throughput of a Random Flow	53.58 kbps	48.91 kbps
Standard deviation of Throughput for Random flow	16.61 kbps	2.93 kbps

Table 7.9: Throughput comparison with and without IP-ABR (Drop-tail = 30).

Metric	TCP	IP-ABR
Segments with a delivery latency $\geq$ 117ms	8.53 %	0.9 %
Segments with a delivery latency $\geq$ 312ms	5.47 %	0.01 %

Table 7.10: Delivery latency comparison with and without IP-ABR (Drop-tail = 30 packets).

7.3.2 DRD test observations

As in previous tests the throughput variability was the first metric we examined. Figure 7.14 shows the measured throughput of ordinary TCP traffic without any IP-ABR service. From this we can see that, as in the previous tests, there are still large variations in throughput, despite the usage of a more proactive queuing discipline. However, the size of the largest peaks have diminished.

Figure 7.15 shows the throughput in the case of IP-ABR based TCP traffic. As can be seen, there are negligible variations in throughput. As shown in Table 7.9, when we examine a random flow we found the standard deviation in the throughput to be about 2.93 kbps in the case of IP-ABR traffic and 48.91 kbps in the case of ordinary TCP traffic.

It is difficult to estimate the number of packet drops in a scenario where RED/DRD is used. Unlike the drop-tail case wherein packet drops occur when the queue overflows, in the RED/DRD case randomly selected packets are dropped, beginning when the queue size crosses the minimum threshold set, with increasing probability as the queue approaches the maximum threshold. Hence, instead of estimating the number of packets

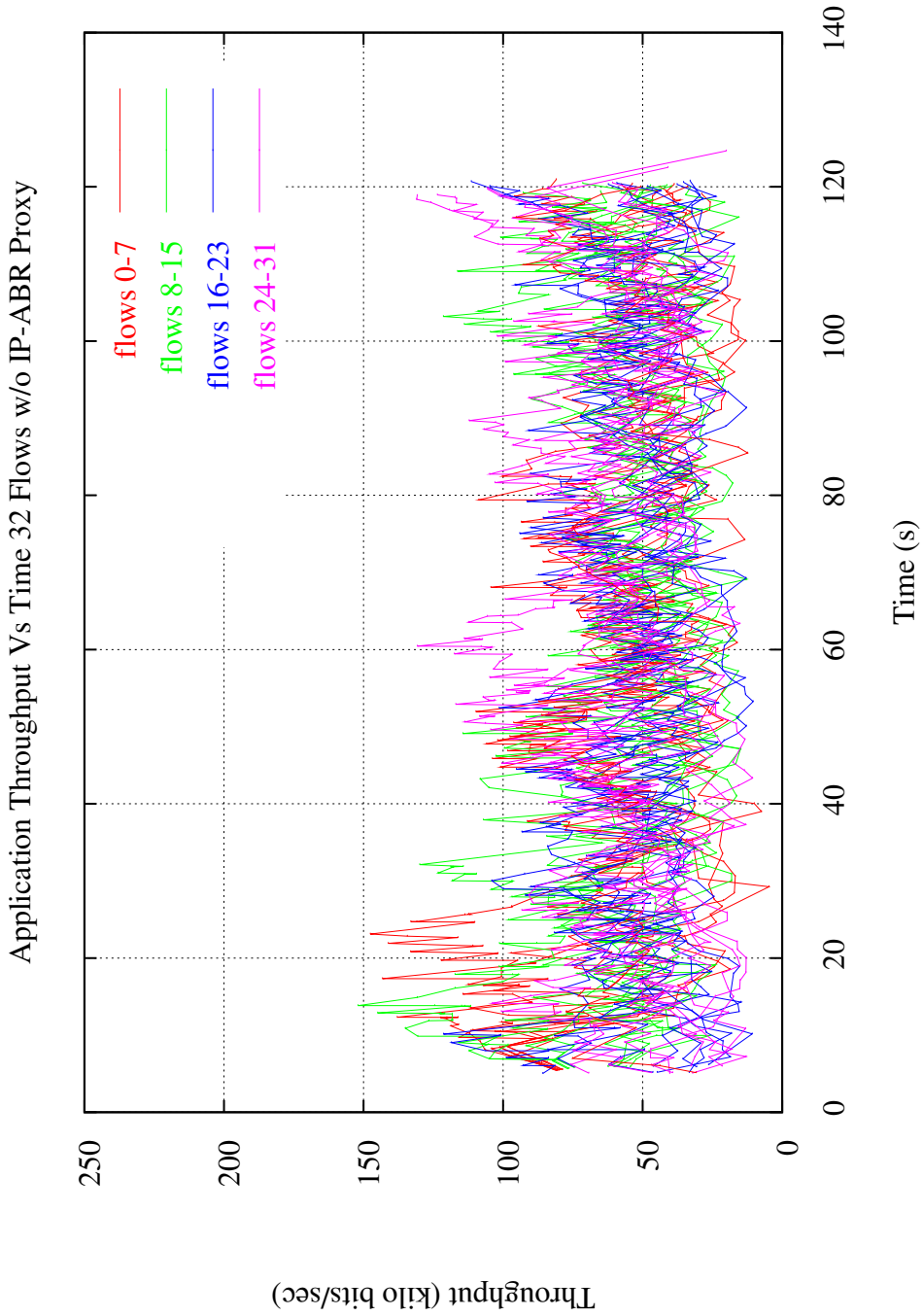

Figure 7.14: Throughput vs. flow duration for 32 TCP flows using DRD with minimum and maximum thresholds of 15 and 40 packets.

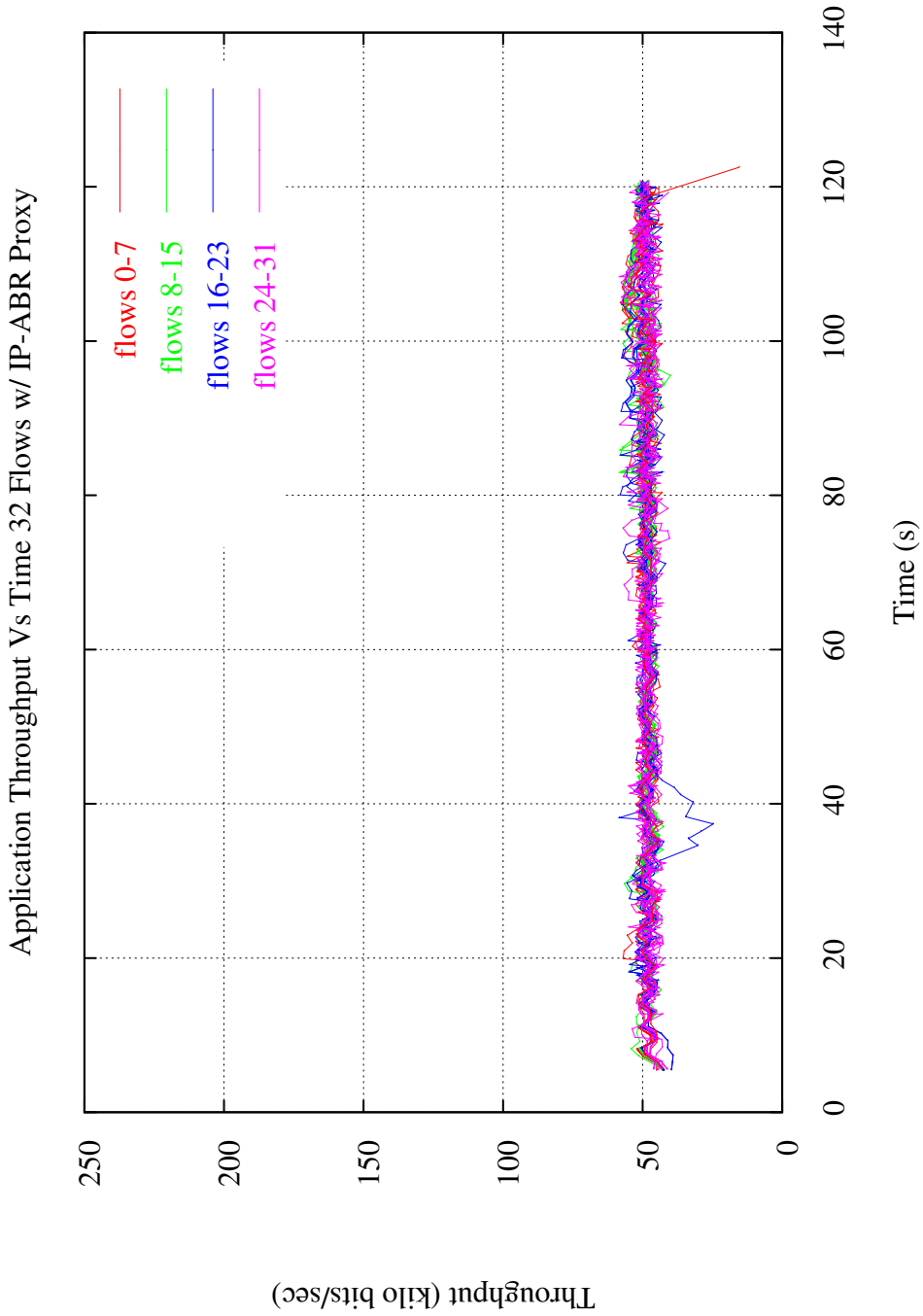

Figure 7.15: Throughput vs. flow duration for 32 IP-ABR service flows using DRD with minimum and maximum thresholds of 15 and 40 packets.

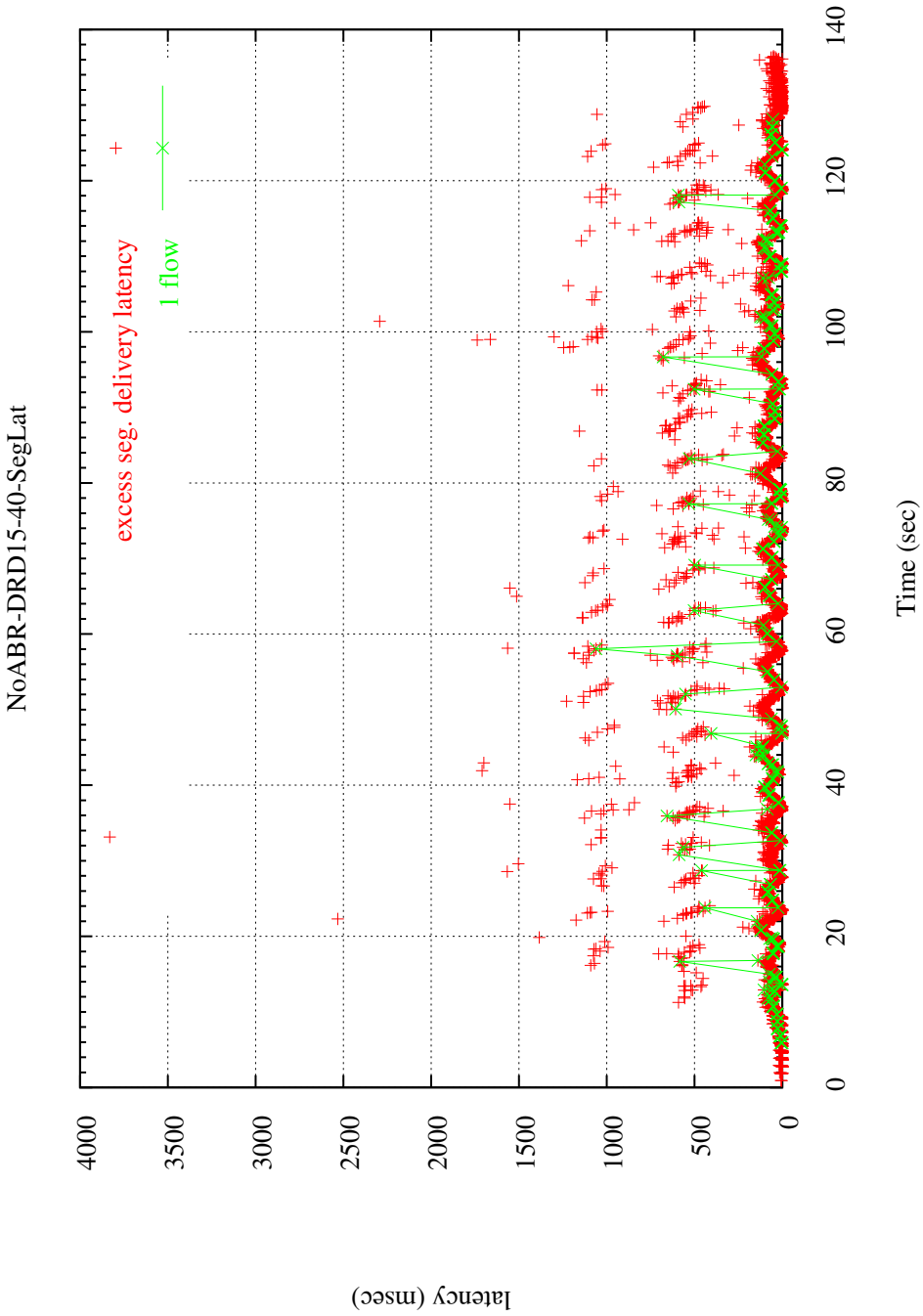

Figure 7.16: Excess segment delivery latency vs. test duration (32 TCP flows using DRD).

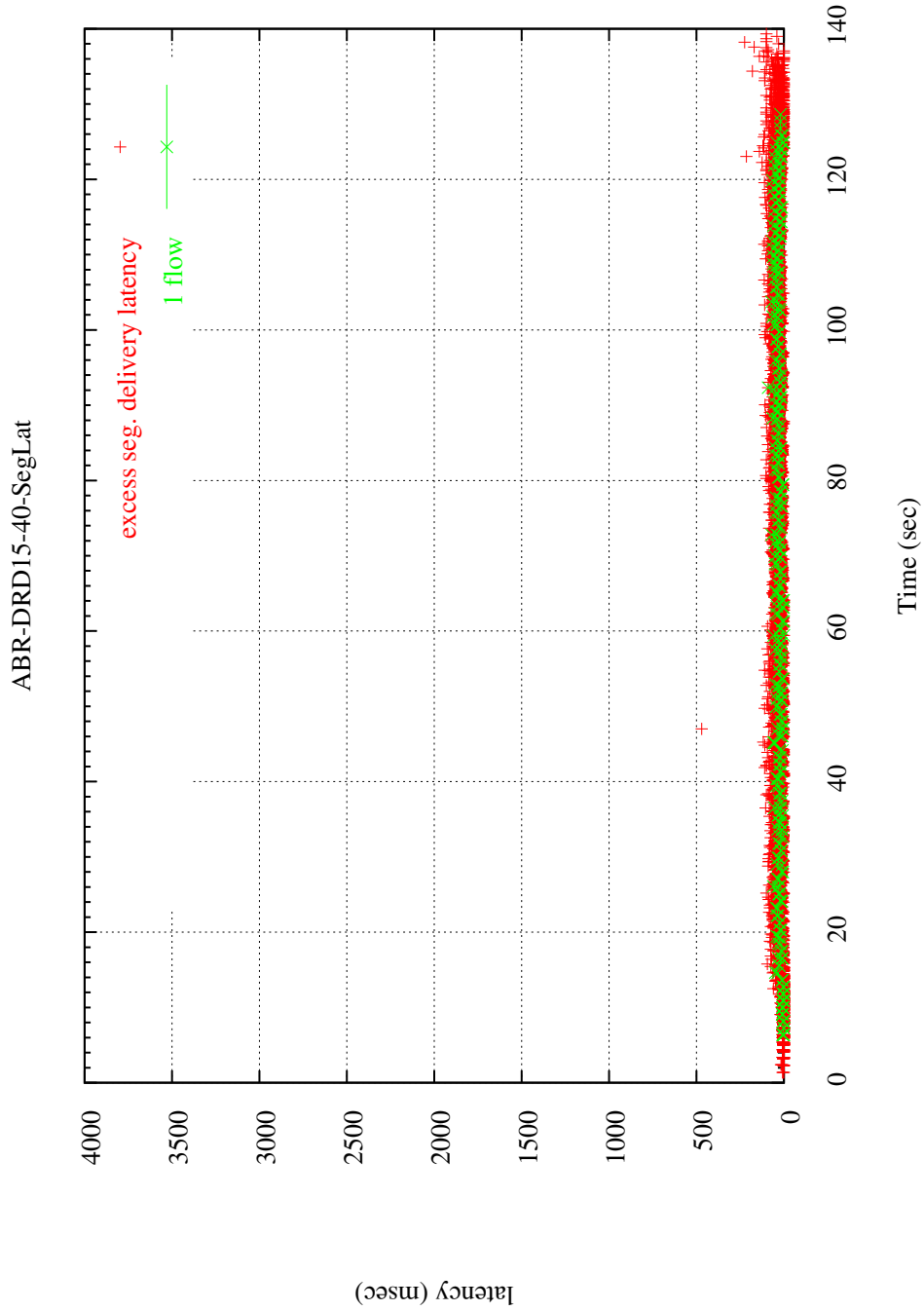

Figure 7.17: Excess segment delivery latency vs. test duration (32 IP-ABR service flows using a DRD).

lost as in previous tests, we provide an estimate of total packets with ESDL greater than the queuing delays corresponding to the minimum and maximum threshold queue length settings. The actual count of packets dropped and retransmitted ranges between these two numbers.

As shown in the Table 7.10, using DRD for TCP service we estimated that 8.53% of all transmitted segments experienced an ESDL of greater than 117 ms and 5.47% had experienced an ESDL greater than 312 ms. A large number of packets still get retransmitted as can be seen from Figure 7.16. Where as with the IP-ABR service enabled we see only 0.9% of packets with a delay greater than 117 ms and only 0.01% retransmitted with a delay greater than 312 ms. In fact from Figure 7.17 we can see that only 1 packet has been retransmitted. This single drop corresponds to the drop in throughput of a single flow shown in Figure 7.8.

With the set of tests conducted so far with the prototype IP-ABR PEP, we have shown that we can create an IP-ABR service that offers throughput with minimal variation and with minimal delay, jitter and packet loss. In the drop tail tests we have seen that IP-ABR service improves QoS considerably, as long as the queue size used on the congested router, is sufficiently large enough, to accommodate the minimal congestion due to router's link bandwidth difference.

7.4 Fairness

In the tests conducted so far we have seen how well the prototype IP-ABR proxy succeeds in improving the QoS when compared to ordinary TCP. We also stated previously that, in addition to offering better QoS, the IP-ABR service can also guarantee fair bandwidth usage. This is something that regular TCP cannot offer at all.

7.4.1 Equal sharing test

In the tests conducted so far we have configured the IP-ABR proxy to equally allocate the available bandwidth amongst the flows. From throughput graphs seen so far we can see that the IP-ABR proxy does in fact succeed in band limiting the throughput of the flows to a very narrow bandwidth range over time. In this section we will gauge the fairness of this division of this bandwidth between different flows.

So, in order to draw a fair comparison of IP-ABR and TCP fairness, we used the data from the previous drop-tail test for which the queue size of 30 packets was used in both cases. From this test data, we estimated the flow utilization over the duration of test, for which computation the utilization was normalized to the bandwidth of the satellite link. Figure 7.18 the normalized utilization of each IP-ABR flow was plotted with green and the normalized utilization of each TCP flow was plotted with red. As we can see the IP-ABR utilization numbers are almost equal, whereas the regular TCP utilization numbers vary widely.

7.4.2 Weighted bandwidth distribution test

Until now, we have looked at scenarios in which the IP-ABR proxy distributes bandwidth equally amongst the flows. However, the proxy is also effective in scenarios for which bandwidth needs to be distributed unevenly. To verify the proxy's effectiveness in creating a weighted distribution of bandwidth we conducted a test. On the bottleneck router we configured a drop-tail queuing discipline with a queue size of 30. The duration of the test was 480 secs. The remaining test parameters and configurations were the same as in the previous tests. Using the IP-ABR proxy assign we bandwidth in the ratio of 1:2:4:8 to 4 groups of flows, each group consisting of 8 flows.

With this test we wish to first verify that the bandwidth was distributed as we desired.

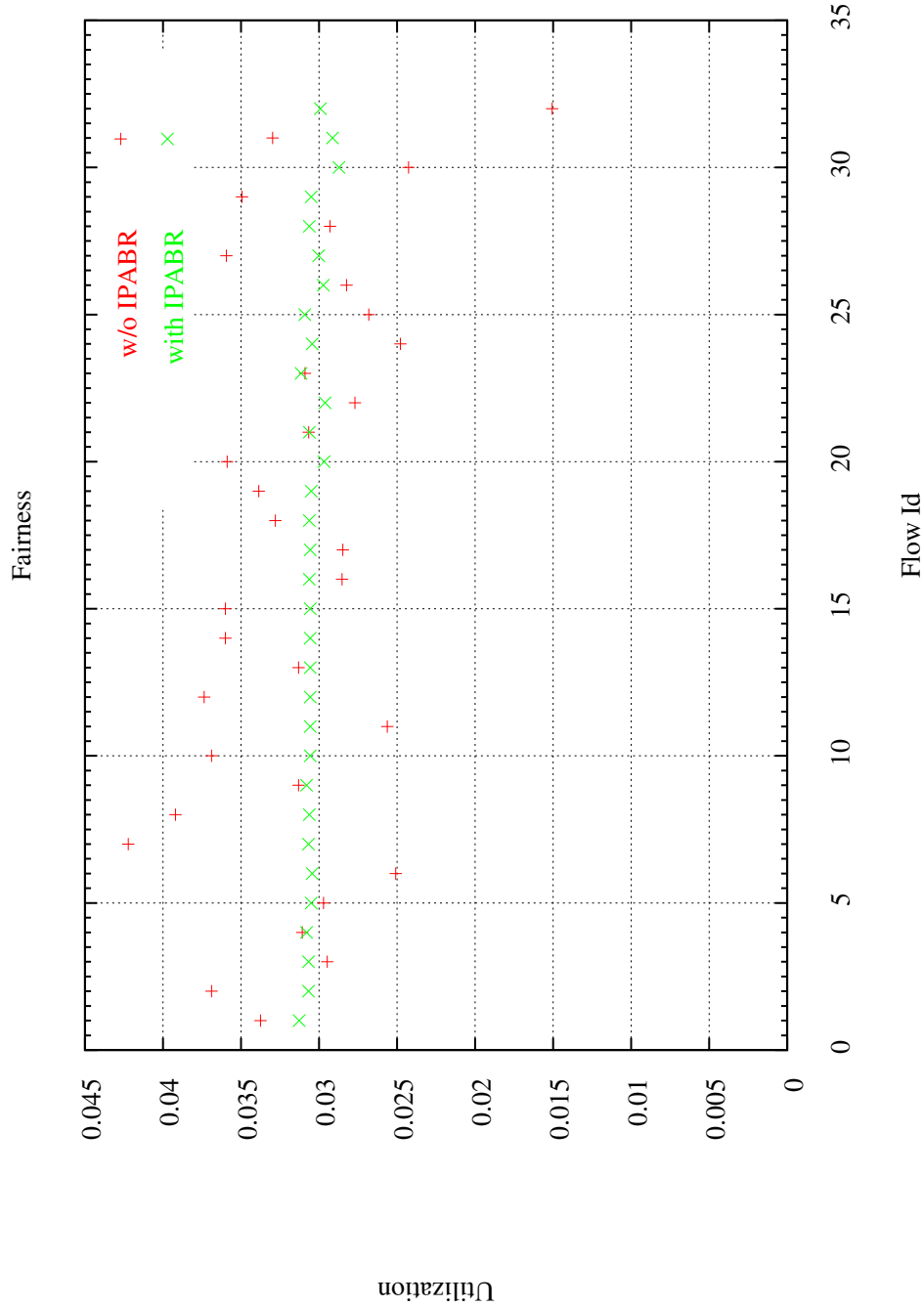

Figure 7.18: Normalized Link Utilization comparisons of IP-ABR to TCP.

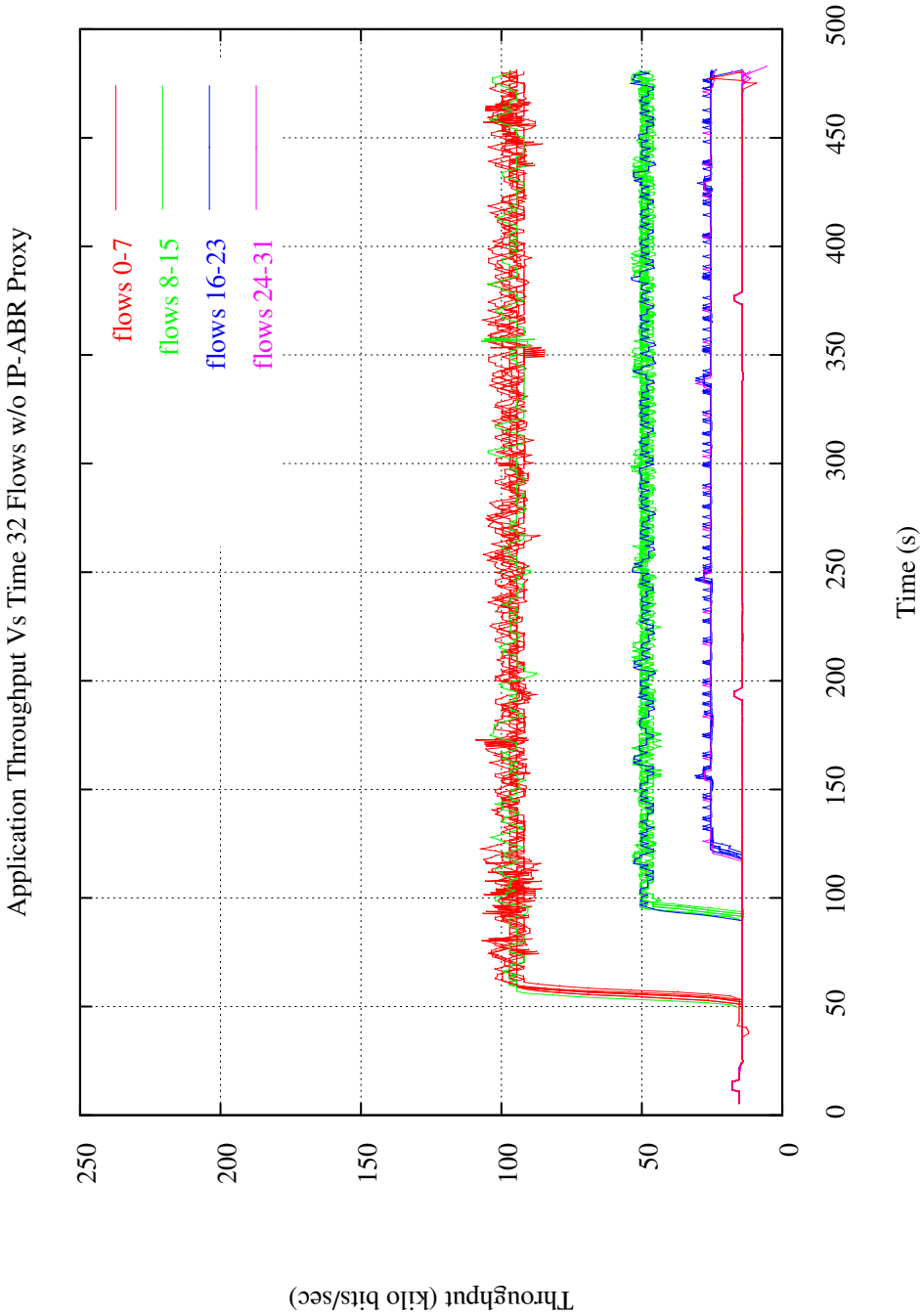

Figure 7.19: Throughput variation during weighted bandwidth distribution test.

In addition to this, we wish to also verify that each group of flows exhibits the characteristics of IP-ABR service flows, namely low throughput variation, low delay and low jitter. The utilization plot used in the previous test (i.e., Figure 7.18) does not show the throughput variation. Therefore instead of examining the utilization per flow of we examine the throughput variability.

Figure 7.19 shows the measured throughput for each of the 32 flows over the duration of the test. From this Figure we can clearly see that the 32 flows are grouped into 4 groups and that the achieved throughput of each group was appropriate for the ratio we have allocated to them via the IP-ABR PEP. In addition to this, the achieved throughputs display minimal variation, similar to other IP-ABR tests seen so far. From these two fairness tests we have seen that the IP-ABR PEP can be used to fairly distribute bandwidth whether the distribution of bandwidth is equal or weighted.

7.5 Available bandwidth variation response test

As mentioned earlier in Chapter 2, one of the advantages offered by an IP-ABR service over IP-VBR service, is that end hosts using an IP-ABR service can take advantage of bandwidth as it becomes available unlike end hosts using IP-VBR. In our earlier simulation studies we have conducted a test (i.e., Figures 5.2 and 5.3) wherein the end hosts respond to reduction in available bandwidth via the IP-ABR PEP. However, none of the tests conducted on the prototype until this point, show how end hosts respond to changes in available bandwidth. Therefore we conducted an additional set of tests to demonstrate this.

We use the same topology as in previous tests, in which we have two end hosts separated by a high latency bottleneck link. We conducted two tests, the first without the IP-ABR service and second with IP-ABR service enabled. We once again fixed the queue

Test duration	240 secs
Concurrent unidirectional flows	32 flows
Drop tail queue limit	30 packets
Available bandwidth variation period	20 secs

Table 7.11: Parameter for available bandwidth variation response test

size to a length of 30 packets for both cases and again used a drop-tail queuing discipline. In these tests we increased the duration of the test to 240 secs from 120 secs. During the duration of the test we vary the available bandwidth of the satellite link between 192000 Bytes/sec to 64000 Bytes/sec periodically at an interval of 20 secs. Therefore, the lower and upper values of the available bandwidth are in a 1:3 ratio. This can be accomplished by modifying the emulated link bandwidth in NISTnet.

Figure 7.20 shows the measured throughput for the duration of the first test which was conducted without the IP-ABR service. We can see that the congestion detection algorithms used in TCP allow the end hosts to respond to variations in available bandwidth. We can see that end hosts reduce or increase throughput in response to corresponding changes in the available bandwidth. However, as seen before in our NS simulation, we see irregular variations in throughput even during the periods when available bandwidth is constant. We can also observe how some flows achieve much higher throughput at the price of the bandwidth of other flows.

In the next test we enabled IP-ABR service so that the IP-ABR PEP regulates flows to match the available bandwidth. We then measured the throughput over the duration of test. From Figure 7.21 we can see that end hosts respond to the changes in available bandwidth by correspondingly varying their throughput. However, unlike in the previous case, here the throughput variation is brought about by the IP-ABR PEP. The IP-ABR PEP reduces or increases the window limits on the end hosts corresponding to changes in available bandwidth. This causes the end hosts to adjust their throughput without causing

congestion or without causing them to break out of self-pacing mode. This creates flows with better QoS and fairness as we desired from an IP-ABR service.

From the plot we also notice a phasing effect. It appears that some flows respond to changes in available bandwidth faster than others. This difference is approximately 5 secs and is also noticeable in the NS simulations conducted earlier in Figure 5.3. This 5 sec lag time seems to be independent of the period of the bandwidth oscillations.

With the tests conducted so far we have shown that the prototype IP-ABR PEP can be used to successfully create an IP-ABR service in networks with large bandwidth delay products such as satellite networks. From the results of these tests we see that the resulting traffic has reduced throughput variability, reduced packet loss, reduced delays and jitter. In addition to the improvement in QoS, IP-ABR also offers better fair bandwidth usage, whether it be equal sharing of bandwidth or a weighted distribution.

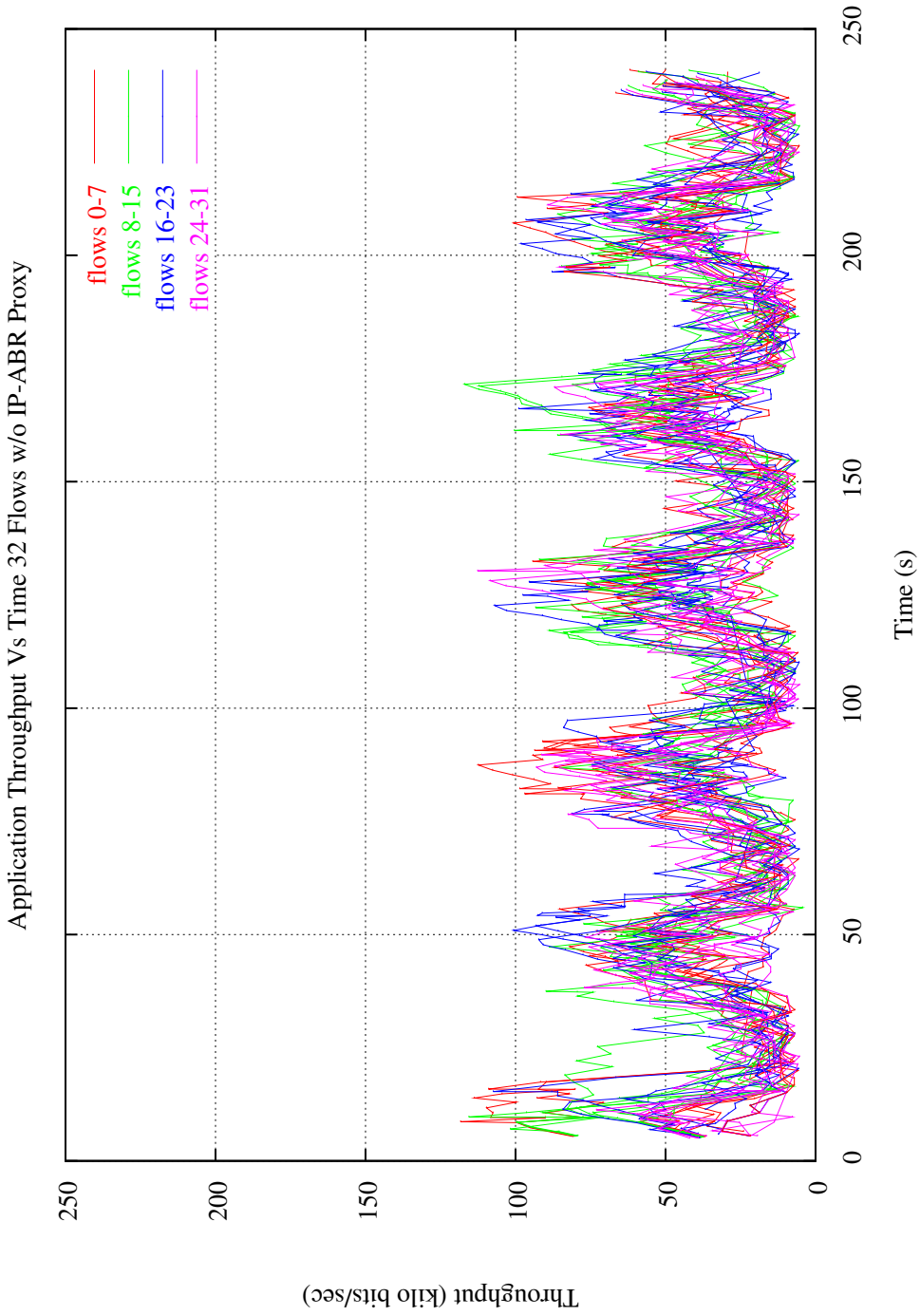

Figure 7.20: TCP response to variations in available bandwidth.

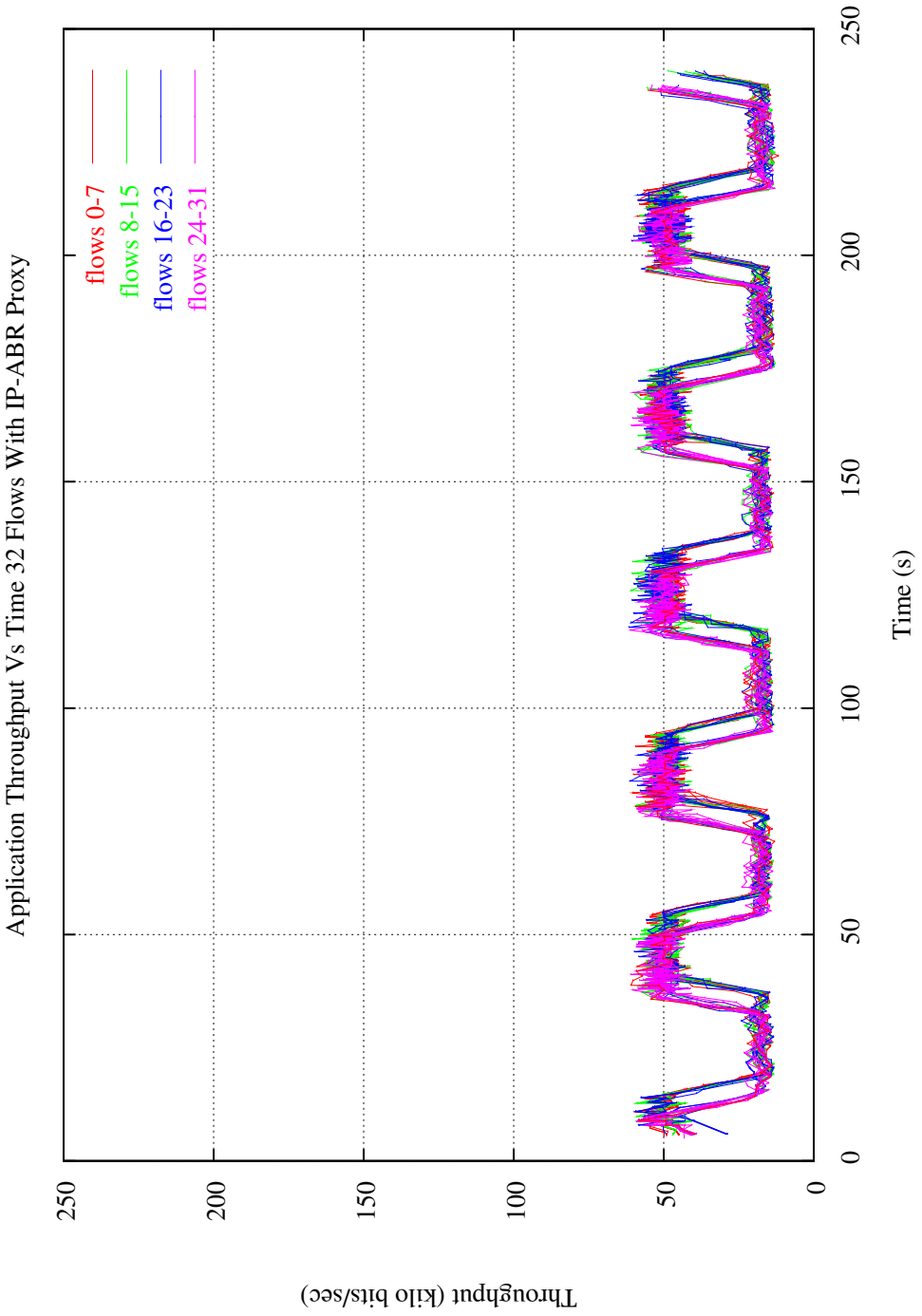

Figure 7.21: IP-ABR service flows response to variations in available bandwidth.

Chapter 8

Conclusion

The transport control protocol (TCP) is one of the most popular protocols in use on the Internet today, providing a reliable, connection oriented service. However, the current congestion control scheme employed in most TCP implementations causes TCP flows to exhibit very high throughput variability and jitter. This degradation in QoS is even more pronounced when TCP is used in networks with large end to end latencies such as satellite networks.

For certain types of applications in use today there is a need for a transport service that can provide a higher QoS than currently provided by TCP. In order to meet the needs of such applications in this research we started off by proposing a new service. The goal of this new service is to provide better QoS compared to TCP while still retaining the reliable connection oriented characteristics of TCP. This new service is similar to ATM's ABR service and is therefore referred to a IP-ABR. In order to create this IP-ABR service we made use the existing TCP protocol, instead to developing a new transport protocol.

As highlighted in Chapter 3, the key idea behind IP-ABR is dynamic bandwidth allocation. However, in order achieve this TCP flows must be regulated in a manner so they they are locked into a steady self-pacing mode where they attain the bandwidth allocated

by the Network, thereby reducing throughput variability and jitter. In order to achieve this, we have developed the novel scheme of using a Performance Enhancing Proxy (PEP) to create an IP-ABR service by regulating existing TCP flows. As mentioned previously in Chapter 3 this new scheme involves using an IP-ABR PEP to dynamically window-limit TCP flows by modifying in-flight ACK packets.

In Chapter 4 we have provided details on the window estimation algorithm and RTT estimation scheme that is used by the Performance Enhancing Proxy to regulate TCP flows to a specific bandwidth with minimal throughput variation. In Chapter 5, we proceeded to use the previously developed mechanisms in developing a proof of concept implementation of the Performance Enhancing Proxy in the NS simulation environment. From the results of our test simulations we have shown how an IP-ABR PEP can significantly reduce TCP throughput variability by successfully locking TCP flows into a steady self-pacing mode for the duration of the connection. From these early tests we have also discovered how the "silly window" avoidance schemes used in TCP prevents the IP-ABR flows from achieving the full bandwidth allocated by the IP-ABR PEP. In order to rectify this problem we modified the optimal window estimation algorithm to compensate for TCP stack behavior.

Finally we have also implemented a working prototype of the IP-ABR PEP for the Linux platform. In Chapter 6 we have detailed how we implemented the prototype using the LINUX Netfilters Firewall API to intercept in-flight TCP. We have also conducted a variety of tests on the prototype IP-ABR PEP to measure performance of the prototype. In order to conducts tests on the IP-ABR PEP we developed a test-bed using PCs and network emulation software that can provide an environment similar to a satellite IP network.

From the throughput variability tests conducted using the prototype IP-ABR PEP we have successfully shown that given sufficient queue sizes on the router the prototype IP-

ABR PEP is successful in inducing TCP flows into a steady self-paced mode with each IP-ABR flow attaining a throughput equal to corresponding allocated bandwidth. We have shown the resulting IP-ABR flows exhibit minimal throughput variability in comparison with regular TCP flows. We have also established a new metric called Excess Segment Delivery Latency (ESDL) which can be used to measure effects of packet retransmission delays upon the end application. From the results of the tests conducted we have shown that IP-ABR is successful in eliminating retransmissions and reducing queuing delay, which also leads to reduced jitter when compared to regular TCP flows. We have also shown from the bandwidth variation tests conducted that, IP-ABR flows can rapidly adapt to variations in allocated bandwidth, which allows the end applications to use excess available bandwidth, or relinquish bandwidth if needed by higher priority traffic. From the weighted bandwidth distribution tests we have demonstrated IP-ABR's unique ability to allocate bandwidth on an individual flow basis. This unique ability of allowing bandwidth allocation policies to be specified on a per-flow basis is extremely beneficial to network administrators.

In this research, we have successfully demonstrated that a service like ATM's ABR can be created in an IP network using existing TCP implementations and a performance enhancing proxy without the need for deploying ATM. We have also shown that such a service can offer a reliable connection oriented service with minimal throughput variation, lower packet loss, lower delay even in networks with large bandwidth delay products.

Bibliography

[Bra89] R. Braden. Requirements for internet hosts – communication layers. RFC 1122, IETF, January 1989.

[FJ93] Sally Floyd and Van Jacobson. Random early detection gateways for congestion avoidance. *IEEE/ACM Transactions on Networking*, 1(4):397–413, 1993.

[Flo00] S. Floyd. Congestion control principles. RFC 2914, IETF, September 2000.

[FV02] Kevin Fall and Kannan Varadhan. *The ns Manual*. The Virtual InterNetwork Testbed Project, April 2002.

[Gay96] M. Gaynor. Proactive packet dropping methods for tcp gateways. available at `http://www.eecs.harvard.edu/~gaynor/final.ps.`, October 1996.

[Hol03] David Holl. Performance analysis of tcp and stp implementations over satcom links. Master's thesis, Worcester Polytechnic Institue, Worcester, MA 01609, May 2003.

[JK88] Van Jacobson and Michael J. Karels. Congestion avoidance and control. In *ACM SIGCOMM '88*, pages 314–329, Stanford, CA, August 1988.

[MKMQ96] Micheal J. Karels Marshall Kirk McKusick, Keith Bostic and John S. Quarterman. *The Design and Implementaion of the 4.4BSD Operating System*. Addison-Wesley, Reading, Massachusetts, 1996.

[Nag84] John Nagel. Congestion control in ip/tcp internetworks. RFC 896, IETF, June 1984.

[Pos81] Jon Postel. Transmission control protocol. RFC 793, IETF, September 1981.

[RBS94] D. Clark R. Braden and S. Shenker. Integrated services in the internet architecture: an overview. RFC 1633, IETF, July 1994.

[Rij94] A. Rijsinghani. Computation of the internet checksum via incremental update. RFC 1624, IETF, May 1994.

[Rus01] Rusty Russell. *Linux Netfilter Hacking HOWTO rev 1.11*. Netfilter Project, October 2001.

[Sta02] William Stalling. *High-Speed Networks and Internets: Performance and Quality of Service*. Prentice Hall, Upper Saddle River, New Jersey, second edition, 2002.

[Ste94a] W. Richard Stevens. *TCP/IP Illustrated: implementation*. Addison-Wesley, Reading, Massachusetts, 1994.

[Ste94b] W. Richard Stevens. *TCP/IP Illustrated: the protocols*. Addison-Wesley, Reading, Massachusetts, 1994.

[Ste97a] W. Stevens. Tcp slow start, congestion avoidance,fast retransmit, and fast recovery algorithms. RFC 2001, IETF, January 1997.

[Ste97b] W. Richard Stevens. *UNIX Network Programming Vol1*. Prentice Hall, Upper Saddle River, New Jersey, second edition, 1997.

[Tan96] Andrew Tanenbaum. *Computer Networks*. Prentice Hall, Upper Saddle River, New Jersey, 3 edition, 1996.

[VJB92] R. Braden V. Jacobson and D. Borman. Tcp extensions for high performance. RFC 1323, IETF, May 1992.