ABSTRACT

Title of Thesis:	IMPROVING TCP PERFORMANCE OVER HIGH-BANDWIDTH GEOSTATIONARY SATELLITE LINKS
Degree candidate:	Vijay G. Bharadwaj
Degree and year:	Master of Science, 1999
Thesis directed by:	Professor John S. Baras Department of Electrical and Computer Engineering

The Transmission Control Protocol (TCP) is the most widely used transport protocol in the Internet today. The problem of poor TCP performance over satellite networks has recently received much attention, and much work has been done in characterizing the behavior of TCP and proposing methods for improvement. Meanwhile it remains hard to upgrade the majority of legacy host and gateway systems in the Internet that are running old and outdated software so that they can perform better in the changing networks of today.

In this thesis we consider an alternative network architecture, where large heterogeneous networks are built from small homogeneous networks

interconnected by carefully designed proxy systems. We describe the design and implementation of such a proxy and demonstrate marked performance improvements over both actual and simulated satellite channels. We also discuss some benefits and drawbacks of using proxies in networks and explore some tradeoffs in proxy design.

IMPROVING TCP PERFORMANCE OVER HIGH-BANDWIDTH GEOSTATIONARY SATELLITE LINKS

by

Vijay G. Bharadwaj

Thesis submitted to the Faculty of the Graduate School of the
University of Maryland, College Park in partial fulfillment
of the requirements for the degree of
Master of Science
1999

Advisory Committee:

Professor John S. Baras, Chair
Professor Leandros Tassiulas
Dr. M. Scott Corson

DEDICATION

To Dad.

ACKNOWLEDGMENTS

I am grateful to my advisor, Professor John S. Baras for his advice, support and encouragement. I would also like to thank Dr. Leandros Tassiulas and Dr. M. Scott Corson for agreeing to serve on my committee and to review this thesis.

Thanks are also due to Spyro Papademetriou, Manish Karir, Mingyan Liu, Stephen Payne, Roshni Srinivasan, Ravi Vaidyanathan and the numerous other colleagues who contributed to a great working environment with their constant help and support, both on a professional and a personal level. I am also grateful to Tina Vigil, for pulling off the impossible on more than one occasion.

Finally, I would like to express my appreciation for the support received from NASA under cooperative agreement NASA NCC-3528 under the Commercial Space Center program, and from Lockheed Martin Global Telecommunications under a separate contract. Without their support much of this work would not have been possible. Large parts of this work were done during summer internships arranged through the CSHCN at the Interactive Technologies Center of Lockheed Martin Telecommunications. I would like to thank Mark George, Norm Butts, Bill Maruyama, Bob Randall and Kouroush Saraf - I benefited greatly from their patience and wisdom.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Chapter 1

Introduction

1.1 The Internet Architecture

The Internet suite of communication protocols follows a four-layer model, as described in [1]. All Internet hosts and gateways use the Internet Protocol (IP, [2]) at the network layer. Various higher layer protocols run over IP; the most important transport layer protocols are the Transmission Control Protocol (TCP, [3]) and the User Datagram Protocol (UDP, [4]). A significant fraction of Internet traffic is TCP; important application layer protocols built over TCP include the Simple Mail Transfer Protocol (SMTP), the HyperText Transfer Protocol (HTTP), the File Transfer Protocol (FTP), the Network News Transfer Protocol (NNTP) and the Telnet Protocol.

When the Internet protocols were originally designed, the communication speeds supported by networks were relatively low, communication links were unreliable and of low quality by today's standards, and routers were also unreliable. The protocols were intended to be able to support communication

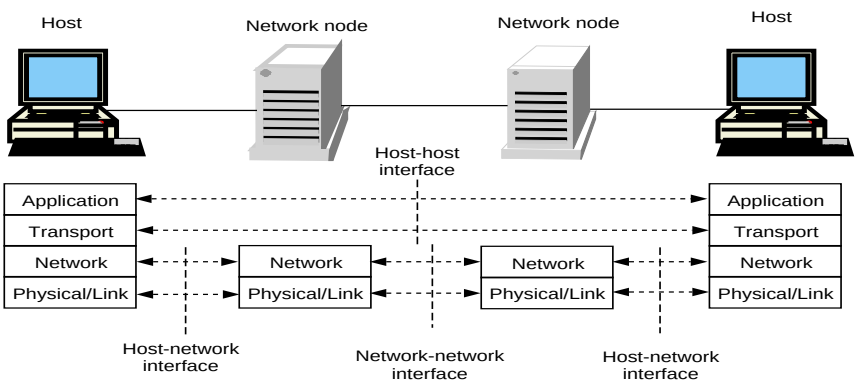


Figure 1.1: Protocol layering in the Internet architecture. Note that the host-network interface is identical to the network-network interface. Also note the clear separation between the network-network interface and the host-host interface.

even in extreme cases such as large-scale network failures or the physical destruction of large parts of the network. Therefore the design emphasizes an end-to-end philosophy; all intelligence is located in the hosts, who are at the edges of the network, whereas nodes inside the network are very simple in terms of both their functioning and their interfaces.

The assumptions of the Internet architecture are set out in [1]:

1. The Internet is a network of networks.

2. Gateways do not keep connection state information.

3. Routing complexity should be in the gateways.

4. The system must tolerate wide network variation.

The first three requirements imply a clear separation between the responsibilities of hosts and gateways (also known as routers). Transport layer

and application layer functionality is present only in hosts, while routing functionality is present mainly in gateways and is transparent to hosts. The first requirement makes the overall design of the Internet independent of link layer technologies, and leads to a hierarchical organization, which makes for scalability. It also simplifies the functionality required at the network layer and allows routing to be carried out in a stateless manner - IP is a "next hop" routing protocol. Thus the Internet is resilient to failures of individual nodes as these nodes are independent of each other.

The fourth requirement is more vague. It does not imply any particular design choice but instead states a general philosophy. Later we will argue that in the current Internet, this requirement sometimes contradicts one or more of the others, and that it may be worthwhile to slightly alter the other requirement in such a case.

1.1.1 The Transmission Control Protocol

The Transmission Control Protocol (TCP) is the transport protocol used for reliable data transmission in the Internet. It is a connection-oriented, byte-stream based, reliable, sequenced delivery protocol. TCP connections are full-duplex, and each byte of data transmitted in a given direction of a TCP flow is assigned a unique sequence number by the sender. This number is used for resequencing at the receiver and in acknowledgments.

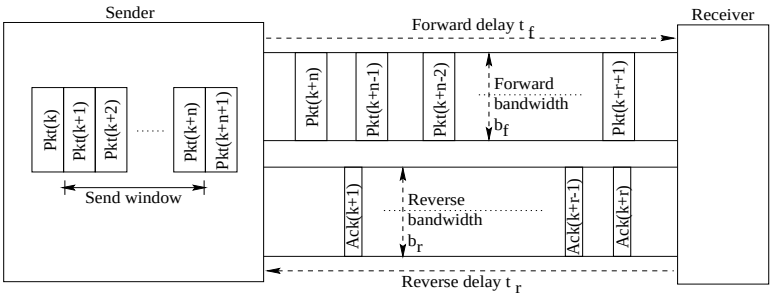


Figure 1.2: Bandwidth, delay and optimal window size. Here the window is equivalent to n packets and fully utilized. If the sender is to steadily send data without having to pause and wait for acknowledgments, the window must be at least $b_f(t_f + t_b)$ - i.e. the product of forward bandwidth and round-trip delay.

A host initiates a TCP connection with another host by sending a segment with the SYN flag. This segment also contains an initial sequence number and any TCP options the host might wish to negotiate. The second host responds with an acknowledgment - a SYNACK, with its own initial sequence number and the options it is prepared to accept. When the initiating host receives this message it can start sending data. The purpose of this exchange, known as a three-way handshake, is to allow hosts to set up the data structures associated with the connection, and to achieve synchronization with respect to sequence numbers in both directions of the connection.

During the data transfer phase, data can be sent in both directions at once. A selective retransmission strategy is used to provide reliable delivery of data - the receiver of data returns acknowledgments for data that is successfully received, and unacknowledged data is assumed lost and is retransmitted. Transmissions use a self-clocking mechanism - the acknowledgments provide the clock by which the sender paces its transmissions.

A window-based flow control scheme is used. A sender's window represents the maximum amount of data that can be sent without waiting for an acknowledgment. Since it takes one round trip time to receive an acknowledgment after data is sent, a network path can only be fully utilized when the window is at least as large as the product of the bandwidth (strictly speaking, forward bandwidth) and round-trip delay of the path (see Figure 1.2).

The sender's window is the smaller of the receiver's offered window and the sender's congestion window. The offered window serves as an upper bound on the sender's window, and is determined by the receiver based on the memory available. It is advertised by the receiver in the header of every data or acknowledgment segment it sends. The congestion window represents the sender's estimate of network congestion, and is calculated based on feedback received from the network. TCP uses implicit feedback - packet losses or large increases in round trip delays are interpreted as indications of congestion.

The evolution of the congestion window occurs in two phases [5]. In the first phase, known as slow start, the congestion window is initialized to one segment (resulting in a transmission rate of one segment per round trip time), and is increased by one segment for every new acknowledgment received. When the congestion window passes a threshold, the congestion avoidance phase is started, and the window is increased by one segment every time a complete window of data is acknowledged. All packet losses are assumed to be due to congestion. In effect, TCP assumes that there are no bit errors in the network. When a packet

loss is encountered, the lost packet is retransmitted and the rate of sending is reduced (at least halved) [3, 5]. Thus the window grows exponentially with time in slow start, and linearly in the congestion avoidance phase, unless restricted by the offered window.

When a host has finished sending data, it sends a FIN message to close its half of the full-duplex connection. Data transfer can still proceed in the other direction until the other host also sends a FIN. When both FINs have been received and acknowledged, the connection is closed and the associated data structures are destroyed by both hosts. In some cases one or both hosts enter a state known as TIME_WAIT, during which they retain the data structures for a certain time and consume any old duplicate segments that might be received.

Some other special signaling messages are also used in TCP. The RST message is used to abort a connection. The PSH flag on a data-bearing segment elicits an immediate acknowledgment, and the URG flag provides a means of sending urgent data.

1.1.2 Problems with TCP

Most TCP implementations in common use have deficiencies that cause them to perform poorly in some environments. We describe the best-known ones below.

Limited size of receiver window. The receiver's offered window is represented in the TCP header by a 16-bit field, which restricts its value to 64 kilobytes. Some

implementations limit the maximum window size even further. This limitation reduces throughput over paths with high bandwidth-delay products, such as geostationary satellite links, as TCP can only transfer one window of data every round-trip time.

Dependence of window growth on RTT. The self-clocking mechanism leads to unfairness between connections that traverse widely differing paths in the network - connections with smaller round-trip times can increase their rate of sending more rapidly, and so end up capturing most of the network bandwidth, at the expense of long-delay connections [6]. In other words, connections with lower round-trip delays are at an advantage as their clocks run faster. To make matters worse, the optimum window size on paths with large bandwidth-delay products is also larger, and many round trips are required before the congestion window grows to its optimal size. Over long-delay links such as satellite, this process takes a very long time. In many cases data transfer completes before the window can grow large enough, and the transfer rate obtained is very low.

Poor performance over bandwidth-asymmetric links. TCP's reliance on self-clocking leads to another problem - in order to keep a smooth flow of data going, frequent acknowledgments are required. This wastes reverse channel bandwidth and leads to problems on bandwidth-asymmetric paths. For example, typical TCP connections use a TCP segment size of 1480 bytes, giving an IP

packet of size 1500 bytes. Nearly all current TCP stacks adopt a policy of acknowledging every other full-sized segment received. Since a minimal ACK segment is 20 bytes, this gives us an IP packet of 40 bytes returned for every two 1500-byte packets sent. Ignoring lower layer headers for simplicity, this means that on paths with more than a 150:1 ratio of forward bandwidth to reverse bandwidth, the reverse path becomes a bottleneck and prevents full utilization of the forward path. For connections using the default IP packet size of 576 bytes (a fairly common occurrence) the problem sets in at bandwidth ratios of about 50:1.

Sensitivity to bit errors. The assumption that all loss is congestion-based leads to many problems on links on which the probability of corruption loss is comparable to or larger than the probability of congestion loss. When a packet is lost due to bit errors, the sender's congestion window is halved even though no congestion is present, thus under-utilizing the link. Moreover, TCP's cumulative acknowledgment scheme can discover only one segment loss every round trip. Thus if multiple segments are lost in one window of data, throughput is reduced sharply.

Implementation complexity. From an implementation point of view, TCP is a fairly complex protocol. It requires the host to maintain multiple timers per TCP flow, and consumes a fair amount of system resource in terms of memory and processing power. This is typically not a problem for desktop computers, but is a

significant drawback for small handheld and mobile devices, which are usually limited to small amounts of battery power and cannot afford large memories or fast processors. Therefore, for these devices, a complete TCP stack is impractical.

1.1.3 Proposed solutions

Many of the above problems appear in networks with satellite links. Such links have high delays and, often, high bandwidth-delay products. They are often asymmetric, with the bandwidth on the downlink being much greater than that on the uplink. User terminals are often mobile, and are power-constrained. Finally, many satellite channels have bit error rates exceeding those of terrestrial channels.

A number of methods have been proposed to solve, mitigate or avoid the above problems by modifying existing networks or existing TCP implementations [7, 8]. Some TCP enhancements have been standardized as TCP options by the Internet Engineering Task Force (IETF). Of these, the Window Scaling option [9] allows the effective size of the offered window to be increased to 30 bits - network applications can be modified easily to use the larger window sizes. The Selective Acknowledgment option [10] prevents serious performance degradation when multiple (up to three) packets are lost in one round trip, and transfer rates over long-delay connections can be improved by increasing the initial value of the congestion window during slow start [11].

The problem of TCP sensitivity to bit errors can often be bypassed by using link-layer techniques such as Forward Error Correction (FEC), power control, or Automatic Repeat Request (ARQ) to reduce the packet loss rate seen by TCP.

Asymmetric links remain a problem for TCP. One proposed solution is to change the frequency of acknowledgments when bandwidth asymmetry is detected, but that leads to much burstier TCP traffic, which is undesirable. Another solution is to use larger segment sizes, but that is not always practical as it may have other adverse effects, such as increasing the packet loss rate of TCP.

There is at present no solution for the problem of TCP unfairness to flows over long-delay paths. At best, network-level practices such as appropriate queueing schemes in routers, can be used to avoid exacerbating the situation in times of congestion.

The problem of implementation complexity is being studied - TCP implementations today are much more optimized than those of some years ago - but typical implementations are still large and do not scale well. The typical case, or fast path, in a TCP implementation is often well optimized and fast; the slower path, which includes handling of errors and various unusual but essential cases such as connection setup and closing, is however bulky and complex. Unfortunately, a TCP implementation consisting solely of the fast path is completely unusable.

1.2 Thesis Organization and Contributions

In this thesis we look at an approach which attempts to solve many of these problems by decomposing large, heterogeneous networks into smaller, nearly homogeneous parts and by using proxies to perform transport-layer protocol translations between these smaller networks. We describe the principle behind these TCP Performance Enhancing Proxies (or TCPPEPs as they are now known), and describe the design and implementation of a particular TCPPEP. We present results showing that in many realistic situations, deploying such proxies can deliver significant gains in performance.

The rest of this thesis is organized as follows: in Chapter 2 we describe the functioning of TCPPEPs and look at various types of such proxies, and in Chapter 3 we describe the design of our proxy implementation. Chapter 4 describes implementation issues, and Chapter 5 presents some results from this implementation. In Chapter 6 we discuss some issues that arose during implementation and testing, and derive a flow control scheme for our architecture. We then use the results to explore some tradeoffs in proxy design. Finally, some possibilities for future work are presented in Chapter 7.

Chapter 2

A Modification of the Internet Architecture

We have seen that TCP performs poorly over some network topologies, such as those with highly asymmetric links, high-bandwidth satellite links, or terrestrial wireless links. Often these problems arise from inadequacies in the layered communication model - in many situations, the layers are not independent but depend on each other in complicated ways. For example, the transport layer depends on link layer parameters such as delay and error rate, the network layer is affected by the stability of individual links, and so on. Even the application layer is not independent of the link layer - interactive applications, for instance, require short round trip times. Problems also arise when functionality is duplicated across layers; for example, TCP may perform poorly over networks that use link-layer ARQ mechanisms, due to the round trip time variance induced by link ARQ.

If a network is homogeneous or nearly so, then layered protocol design yields simple and effective solutions. For example, the Internet protocols work extremely well in a network comprising symmetric bidirectional links with low or

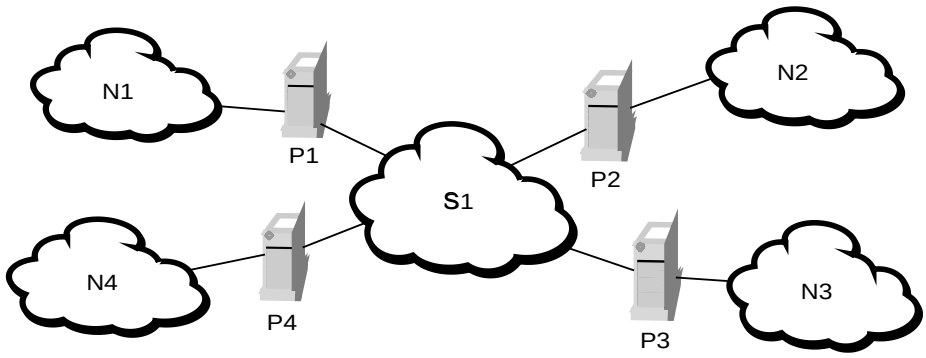


Figure 2.1: "Network of networks" using modified architecture. S1, N1, N2, N3 and N4 are all networks consisting of homogeneous elements. S1 is significantly different from N1 through N4, so P1 through P4 provide a layer of indirection that translates between the different networks.

medium latencies, rare and uncorrelated errors, and low failure rates, especially if the links are in a richly connected topology so that alternate routes can usually be found between hosts. However, a layered protocol suite designed for a given type of network may perform badly in a network with significantly different characteristics, and this is the case with the Internet protocols over link layer technologies such as high-bandwidth satellite.

2.1 Modified Architecture

This observation suggests a way to modify the Internet for better performance - partition the network into parts that are homogeneous or nearly so, and apply layered protocol design to each part. This involves introducing a layer of indirection between different parts of the network (Figure 2.1). The modified architecture is still a "network of networks", with routing functionality in the

gateways. The only change is that we may allow some gateways (in this case, P1 through P4) to perform higher layer functions, which may involve keeping some connection state information. In a way, we violate the second of the four design principles in Section 1.1, in order to strengthen the fourth.

The proxies P1 through P4 mediate between different networks by performing some translations that attempt to insulate the hosts on N1 through N4 from the special link-layer characteristics of S1. For example, they may use a different protocol within S1 for carrying out some transport level functions, and then perform a translation so that the change is transparent to the hosts. For example, many limitations of TCP arise from the intertwining of its error recovery and congestion control functions, and the proxies could try to change this by handling congestion control on a local basis.

This model can be viewed as a generalization of the Internet architecture: instead of restricting the functionality of gateways to the network layer and below, the dividing line between host and network functionality can be chosen arbitrarily, allowing the designer to trade network node complexity for host complexity. In practice this approach is advantageous because host software tends to be a commodity, while network nodes are fewer and more complex. Thus network nodes are better managed and easier for a network provider to maintain and upgrade.

Application proxies such as web caches are examples of this architecture. They put the line of separation between network and host functionality at the

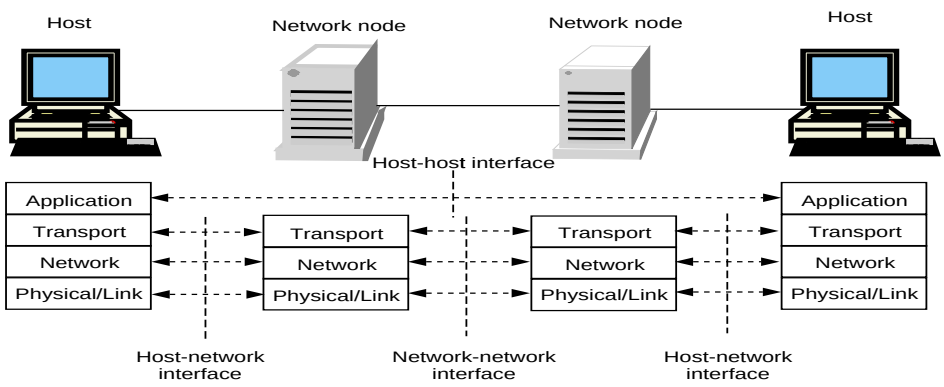


Figure 2.2: Protocol layering in proposed architecture.

application layer. Proxies have been written for many applications that have been found to perform poorly under various sets of circumstances. Each such proxy is specific to an application, and the application must be modified to use the proxy. In our case, however, a closer look at the problem reveals that almost all TCP applications suffer from similar shortcomings, all of which arise from limitations of TCP. Further, since TCP is the only connection-oriented transport protocol in the Internet suite, it is the only transport protocol that suffers from such problems. Application built on UDP, for instance, may also perform poorly in similar circumstances, but these problems are not attributable to UDP. Thus it appears worthwhile to introduce proxies in the network that allow gateways to perform functions up to the transport layer. Such a change can be made transparent to hosts, so that it does not require applications to be rewritten. Devices that implement this approach are known as TCP Performance Enhancing Proxies (TCPPEPs).

Any modification to the present Internet must be interoperable with the large

installed base of systems using traditional TCP/IP protocols, without harming the network as a whole. It should also be at least as flexible as IP; it should allow for flexibility in designing and provisioning the network, and should not restrict network expansion. Finally, the benefits of the change must be large enough to make implementation worthwhile. We will attempt to show that TCPPEPs can be designed to meet all these criteria.

2.2 TCP Performance Enhancing Proxies

We define TCP Performance Enhancing Proxies as agents residing in the network that perform specific actions which modify the behavior of the transport layer in order to improve the performance seen by the end user. Network nodes that perform link-level functions such as ARQ or Forward Error Correction (FEC) to improve TCP performance will be excluded from our definition, as they are in no way TCP-specific, nor do they use any knowledge of TCP in their working. Application-layer entities such as web proxies will also fall outside our definition.

PEPs may be classified into two categories depending on their relationship with the end user. User-visible proxies are similar to conventional proxies such as web proxies, and require some explicit configuration or similar action by the end user in order to be effective. Transparent proxies on the other hand work without user intervention and in many cases without the end user's knowledge.

PEPs can also be classified by their mode of functioning into passive and

active proxies. Passive proxies do not change the contents of the data or control stream, but may alter other characteristics, such as the timing of TCP segments or of acknowledgments on the link. Examples of such proxies are ACK pacing agents and rate-pacing agents. Usually passive proxies are deployed singly, and do not collaborate with other proxies of the same type. Active proxies alter the TCP data or control stream in some way, and may perform actions such as protocol translation or connection splitting. Often these proxies occur in sets of two or more, and the alteration takes place on the links between pairs of proxies. For this reason sets of active proxies are often thought of as "shielding" the network between them from the rest of the network path, and they often use fairly detailed knowledge of the characteristics of the link or network between them.

2.3 Review of Existing PEP Designs

One of the earliest PEPs to be proposed and implemented was the DirecPC^{TM} Turbo Internet^{TM} product [12, 13]. This product was designed to provide home Internet access through a receive-only satellite dish, utilizing a phone line for the reverse channel. Thus the user was linked to the Internet through a link with asymmetric bandwidth and delay characteristics. A user-visible active PEP, deployed at the satellite uplink, performs a form of TCP connection splitting. Data received over the terrestrial link is acknowledged as it is received. It is then transmitted, using a separate TCP connection over the satellite link, to the

receiver. The PEP identifies TCP flows by watching for SYN and FIN exchanges. ACK compression is used to overcome the problem of link bandwidth asymmetry. If multiple acknowledgments are queued for transmission, all but the last one are dropped. Thus the cumulative nature of TCP acknowledgments is used to conserve bandwidth on the reverse channel.

This system delivers transfer rates up to 400 kbps to users over a receive-only satellite dish, even with only a 9.6 kbps modem return link. Most other commercial products providing similar services today also use PEP-based architectures. This architecture does not require any changes to the TCP implementation in the user terminal. The PEP software can be upgraded, and new services can be added easily, without changes to the user's operating system software.

The Snoop architecture [14] was an early implementation of a transparent active PEP. It was mainly intended for enhancing TCP performance for users of terrestrial wireless links. The wireless link is assumed to be the last hop to the user, and a PEP (the Snoop agent) is deployed at the network end of this link. The agent watches for duplicate TCP acknowledgments, and uses them as an indication of corruption loss on the wireless link. It discards these duplicate acknowledgments, and retransmits the lost segment from its cache. Thus Snoop is functionally identical to link layer ARQ in its design and objectives, except that TCP acknowledgments are used as the triggering mechanism.

The major advantage of this design is that the PEP uses soft state - if the

lost segment is not in the Snoop agent's cache, the duplicate ACKs are propagated back to the sender, who retransmits the missing segment. This makes the architecture robust to system failures at the Snoop agent. The major disadvantage of Snoop vis-a-vis link-layer ARQ is that while link layer ARQ can safely be used anywhere in the network, Snoop can only be used at the edges of the network in last-hop links. If a Snoop agent is deployed in the middle of a network and congestion occurs downstream of it, the Snoop agent will absorb the duplicate ACKs and prevent the sender from discovering the congestion, and may lead to network collapse.

A recent commercial implementation of a transparent active PEP is described in [15]. This product is primarily a traffic management tool. It performs many complex bandwidth management functions, including admission control for TCP connections, TCP rate pacing and priority-based bandwidth sharing. Its strengths are its ease of administration and the ability for central control and administration of an enterprise-level network. The primary disadvantage is that the PEP relies on a large amount of hard state. Thus it cannot easily deal with richly-connected network topologies, where packets may be routed around the proxy. Also a system failure in the PEP may cause data loss and failure of communications.

Chapter 3

Design of a TCP Performance Enhancing Proxy

Each of the TCPPEP designs described so far assumes a particular kind of network topology, and may behave unpredictably if this assumption is violated. These proxies are also not transparent to applications - in many common situations, they can cause unusual application behavior and may even cause connection failures or data loss. These proxies were all intended for use in last-hop situations, and can deliver benefits in such situations, but they place constraints on future network expansion, and have limited applicability. For example, when a satellite link is deployed in the middle of the network, it would not be appropriate to use such proxies around the link. Performance may be improved for some users, but there is a potential for many undesired side-effects.

In this chapter we describe the design of a TCPPEP that is completely transparent to end-user applications under nearly all circumstances. This proxy was originally designed to improve TCP performance for users of a high-bandwidth geostationary satellite link deployed in the middle of a Wide Area Network. The TCPPEP model was chosen for the design so that the proxy

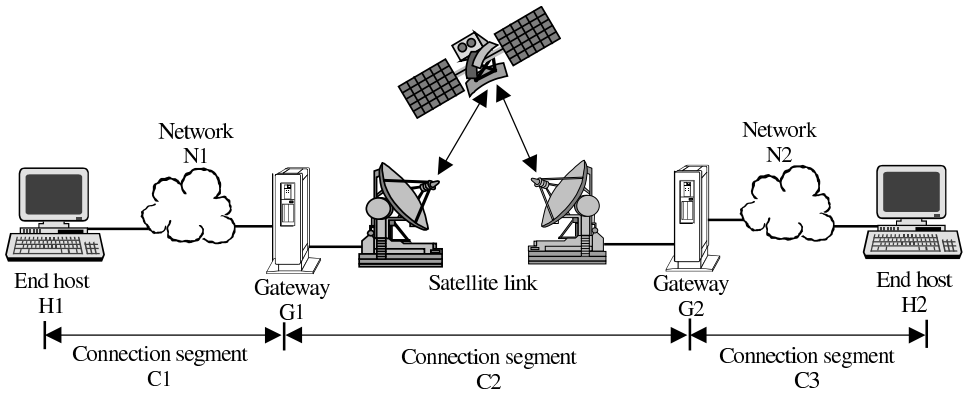


Figure 3.1: TCP connection splitting.

could be deployed by the network provider operating the satellite link, and immediately offer enhanced performance to end-users without requiring any intervention from them.

3.1 TCP Connection Splitting

Our proxy implements connection splitting to shield end-users from the effect of delay and bit errors on the satellite link. It is a transparent active TCPPEPs, intended to be deployed at the ends of the satellite hop. When deployed in this manner, the proxies split end-to-end TCP connections passing through them into separate "connection segments". A typical scenario is shown in Figure 3.1. Connections C1 and C3 are TCP connections, with the respective gateway proxying for the remote host by generating TCP acknowledgments on its behalf. For instance, whenever G1 receives data from H1, it immediately sends an acknowledgment to H1, and takes responsibility for delivering the data all the

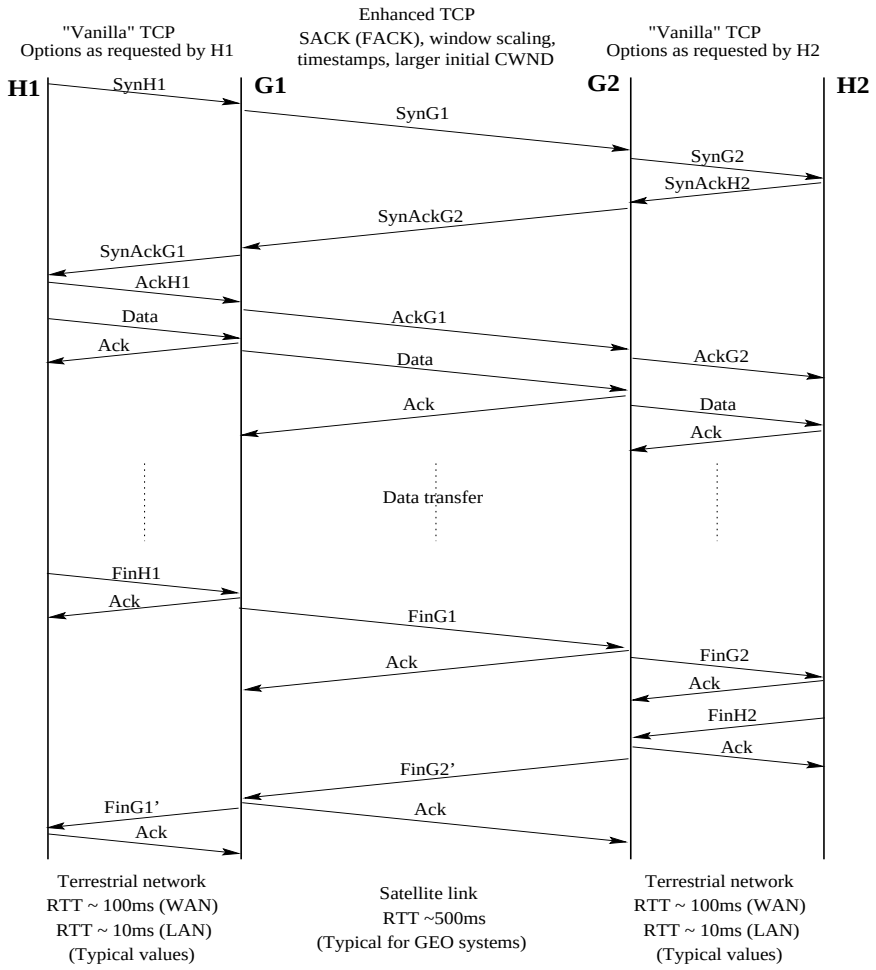


Figure 3.2: Timing diagram for connection splitting in the network of Figure 3.1. This diagram shows a simple unidirectional data transfer from H1 to H2.

way to H2. The connection segment C2 involves only the proxies G1 and G2, and so may use any transport layer protocol to ensure reliable transmission of data. Our implementation used TCP with all the enhancements recommended in [9], [10] and [11].

This system can provide performance gains even compared to deploying enhanced TCP implementations on all end hosts. An obvious advantage is that error recovery can be carried out locally on each connection segment instead of

end to end, potentially reducing delay and conserving satellite bandwidth. Some other benefits will be discussed in Chapter 6.

The operation of the PEP is illustrated in Figure 3.2. Whenever a proxy sees a connection request (i.e. a SYN segment), it intercepts the request and originates a similar connection request with an enhanced option set. When all downstream connections are completed, an acknowledgment (i.e. a SYN ACK) is returned to the host that originated the original request. Both the proxies always negotiate and accept all the TCP options referenced above during connection setup, so that the connection between G1 and G2 will always use all these options. TCP implementations are required to simply ignore unrecognized TCP options, so this mechanism allows a proxy to detect and utilize the capabilities of the other end-point of its connection segment, without having to know beforehand if the other end-point is another proxy or a host.

Once the connection has been set up, the proxy intercepts all data on that connection, returns acknowledgments to the sender bearing the address of the destination, and buffers the data for downstream transmission. When a proxy receives a FIN segment, it immediately closes the corresponding half-duplex connections. When a FIN has been received and relayed for both directions of a TCP connection, all the resources for the corresponding connection segments are freed.

There is a variable delay due to buffering at G1 as well as G2, which is not shown in Figure 3.2. Also, a host may choose to piggyback ACKs on other TCP

packets. In the figure, AckH1 may be piggybacked on the data following it, and FinH2 may be combined with the ACK immediately preceding it.

3.2 The Design Process

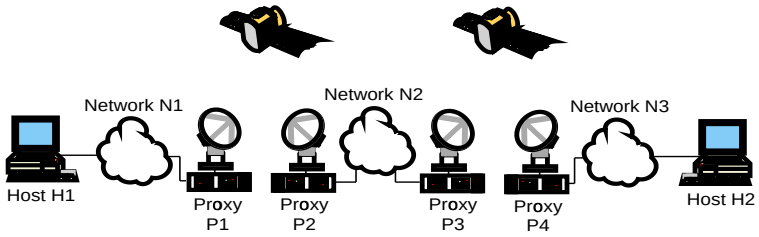
The design objective was to make a PEP that could be used as a plug-in replacement for a router, and would be indistinguishable from a router to end-user TCP applications. During the design phase, all the deployment scenarios shown in Figure 3.3 were considered, in addition to the topology of Figure 3.1, to evaluate the correctness of the proxy's behavior in this respect.

In this section we describe the broad principles of the design and discuss their application to specific issues, pointing out possible problem cases.

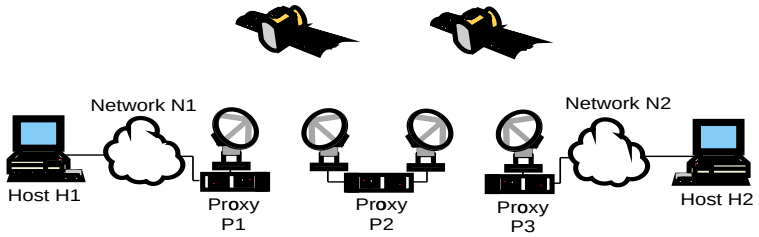
3.2.1 General Considerations

It is often argued that a connection splitting proxy violates the semantics of TCP and therefore can never be deployed without undesirable side-effects. This is not true. RFC 793, which specifies the TCP standard, explicitly acknowledges the possibility of connection splitting, and carefully limits the semantics of TCP to provide for such a situation.

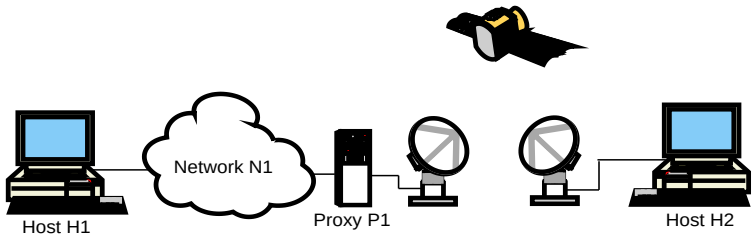
> Some computer systems will be connected to networks via front-end computers which house the TCP and internet protocol layers, as well as network specific software. The TCP specification describes an



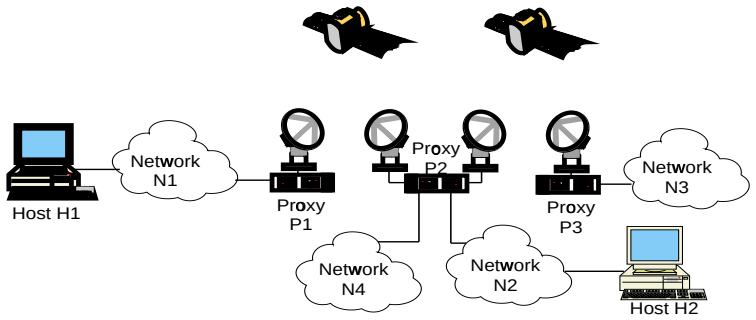
(a) Multiple satellite links with PEPs.



(b) PEP bridging two satellite links.



(c) Last-hop satellite link ($DirecPC^{TM}$ architecture).



(d) Proxy with multiple interfaces.

Figure 3.3: Alternative network topologies considered in PEP design.

> interface to the higher level protocols which appears to be implementable even for the front-end case, as long as a suitable host-to-front end protocol is implemented.
>
> (RFC 793, Section 1.1)

> An acknowledgment by TCP does not guarantee that the data has been delivered to the end user, but only that the receiving TCP has taken the responsibility to do so.
>
> (RFC 793, Section 2.6)

We try to define some characteristics of the "suitable host-to-front-end protocol" referred to above. Our goal is to design the proxies so that each proxy appears to all its adjacent TCP entities as a front-end for the remote host. For example, in Figure 3.1, G1 must appear to H1 as a front-end for H2, and to G2 as front-end for H1. Similarly G2 must appear to G1 as a front-end for H2 and to H2 as a front-end for H1. Thus the main contribution of this design is to show that TCP itself is suitable for use as a host-to-front-end protocol as envisaged in RFC 793.

3.2.1.1 Protocol Correctness

For the PEP to be transparent to hosts, it must guarantee that the various TCPs associated with a connection are consistent. In other words the proxy should not

put the TCPs at the connection endpoints in a combination of states that would not occur normally in TCP.

The proxy must not disturb the end-to-end flow of TCP control and signaling information. It should never try to anticipate the actions of a host. Specifically, a proxy must not originate any segment containing a SYN, RST, URG or FIN flag; these flags only meaningful in a host-to-host context. The proxy should correctly relay all TCP header information sent by the end host, including special flags, lest it affect application behavior in unpredictable ways.

The congestion control algorithms used on networks outside the ones shielded by the PEPs must not be more aggressive than TCP. Thus a connection from a proxy to a host must maintain the conservative approach of TCP. Otherwise installing the PEPs could cause unfairness to other flows and lead to network instability.

3.2.1.2 Robustness

The Internet is a robust system - the hosts and the network are each resilient to failures of the other. Data can be transparently routed around failures in the network without causing active TCP connections to fail. Similarly, network nodes can quickly recover from the effects of a host failure - router buffers used up by connections to or from that host are quickly freed up, as the packets are transmitted downstream. A proxy must maintain this characteristic. It must also deal gracefully with asymmetric routes and with route changes during the

lifetime of a connection.

A truly transparent proxy should not exercise stricter admission control policies for packets than an IP router would. Violating this requirement might create a "black hole" situation wherein the endpoints of a connection would completely lose connectivity even though a link existed between them. The proxy should also be able to detect host failures and reclaim resources committed to the associated connections.

Additionally, if the proxy modifies the TCP stream on some set of links, it must be designed to prevent this modified stream from leaking into the larger Internet, and to minimize undesirable effects in case this happens.

3.2.1.3 Security Issues

The proxy should avoid weakening the security of the end-to-end TCP connection - it should not make attacks such as sequence number spoofing attacks easier than they would be in the absence of the proxies. The proxy must also be able to withstand attacks directed against itself.

3.2.1.4 Implementation And Performance Issues

The most important issues in a proxy implementation are configurability, scalability, performance and efficiency.

The proxy should be flexible - an administrator should be able to control its behavior as much as possible. For example, telnet streams do not gain much from

connection splitting - it should be possible to configure the proxy to simply forward such streams without acting on them. The ability to turn splitting "on" or "off" on certain interfaces, or for connections to or from certain hosts or networks, would also be desirable.

The proxy should be capable of maintaining high rates of data transfer under a wide range of conditions, and should be capable of scaling smoothly to large numbers of TCP connections. It should have fallback mechanisms for cases when it is temporarily overloaded, so that it can ride out temporary spikes in its workload.

The proxy must be computationally efficient. The operation of the proxy should be as simple as possible, and preferably not more complex than that of a typical TCP implementation.

3.2.2 Specific Issues and Discussion

3.2.2.1 Choice of Protocol

As remarked earlier, with reference to Figure 3.1, the connection segment between G1 and G2 can use any protocol, as long as the proxies can translate this protocol back to TCP for the benefit of the hosts. In our implementation, we chose to use TCP on this segment as well for a number of reasons:

- TCP provides a flexible and extensible framework for alternative protocols. TCP options can be negotiated at connection setup, and all TCP

implementations are required to ignore any options they do not recognize. Thus if we want to run a different protocol on proxy-to-proxy links, we can set up the proxy to always negotiate this by setting a special TCP option during connection setup. If we are communicating with another proxy, it will accept this option; otherwise it will ignore the option. Since this process can be done during the three-way handshake, it does not add any appreciable overhead (except perhaps for a small processing delay). Thus the proxy becomes self-configuring, and can also be used in all the topologies of Figure 3.3 without any modification.

- If one of the proxies fails or is overloaded, the system will still work, as the other proxy will discover that it is communicating with a host and adapt accordingly. For example, in Figure 3.1, assume the proxy at G2 fails (but G2 is still functioning as an IP router). Then if H1 originates a connection request, the end-to-end connection will be split into two segments instead of three (namely, H1-G1 and G1-H2) and data transfer will proceed normally.

- If due to some reason a packet gets routed around one of the proxies, it is still in a format that a host can understand; thus connectivity is not lost when routes change and the proxy can no longer perform protocol translation.

- Protocol translation at the proxies is simplified. The TCP header information is passed through unchanged, so it does not need to be

reconstructed at the second proxy.

In sum, we use TCP over the satellite link mainly as a data format. Alternative flow control or congestion control methods can still be negotiated at connection setup, and any extra information required by these methods can be conveyed as TCP options, at any stage during the lifetime of the connection.

3.2.2.2 Connection setup

By default our proxy does not perform admission control for TCP connections. However, it can be configured to do so. When performing admission control, if a connection is rejected, an ICMP message is returned to the originating host. Using RST messages to perform this function (as some commercial proxy implementations do) violates the semantics of the RST flag.

When a proxy receives a connection request (SYN) for a new TCP connection, it makes an entry in an internal cache and sends a similar SYN downstream. This new request has the same sequence number but has enhanced TCP options added to it. Once this SYN is acknowledged by the remote host the proxy returns an acknowledgment to the originating party (Figure 3.2). If no response is received for a specified interval, the entry is removed from the cache.

If at any stage in the above process the proxy does not have the resources to set up connection splitting, the SYN (or SYNACK) is simply forwarded. This way the end-to-end TCP connection will still be completed end-to-end, though it

may not perform as well as connections for which splitting was successful. This method is used by the proxy to reduce its workload when necessary.

Retaining sequence numbers on TCP segments avoids problems with asymmetric routing. If we change the sequence number on a SYN and the SYNACK bypasses the proxy, the acknowledgment will be invalid and the originating host will abort the connection with an RST. Another reason for maintaining the sequence numbers is that the randomness of the initial sequence number is a sort of security measure which prevents some security attacks in TCP [16]. By altering it we may make it more predictable, thus weakening its security.

The proxy will not retransmit a SYN for which no response has been received, unless a retransmitted SYN is received from the originating host. Again, this is to allow for asymmetric routing. Consider the network topology of Figure 3.1, and assume that there is another route (not shown) from H2 to H1 that does not pass through P1 and P2. Suppose Host H1 sends a SYN. The SYN propagates to host H2, which returns a SYNACK that bypasses P1 and P2. This SYNACK is valid when it is received by H1, and this host starts sending data. Now if a proxy keeps retransmitting the SYN, H2 keeps responding with acknowledgments, none of which the proxy sees. This creates unnecessary network traffic.

There are many reasons for delaying the SYNACK from the proxy until a SYNACK is received from the remote host:

- Returning a SYNACK too early could lead to an "impossible" combination

of states in the hosts. If the SYNACK reaches the originating host before the SYN reaches the destination, the origin TCP will be in the ESTABLISHED state, which is a synchronized state, while the destination TCP will be in the CLOSED state, which is an unsynchronized state. This would lead to problems if the proxy failed, or if subsequent packets were routed around the proxy.

- It is a violation of the end-to-end semantics of the SYNACK. In the Internet, when a host receives a SYNACK, it infers that the remote host exists and is ready to receive data. Therefore a proxy must return a SYNACK only when it can reasonably take responsibility for delivering any data received on the connection to its destination, i.e. when it is sure that the destination host exists and will accept the connection.

- When a proxy returns a SYNACK, the originating host can start sending data. A malicious party could use this fact to launch a denial-of-service attack against the proxy. By sending a large number of SYN segments with bogus destination addresses and following them up with large amounts of data, the attacker could use up all the memory at the proxy, thus preventing other users from getting any service.

The cache of SYN requests prevents an attack similar to the above in the reverse direction. In the absence of such a cache, an attacker can send large numbers of SYNACKs with bogus source and destination addresses and follow

them up with data, thus exhausting all the memory on the proxy. If it were not for this consideration, the cache would be unnecessary, as all the information required to set up connection splitting can be deduced from the SYNACK.

3.2.2.3 Data Transfer

The proxy preserves the TCP sequence numbers on data passing through it. If sequence numbers are not preserved and if a packet somehow gets routed around the proxies due to a routing change or a proxy failure, this packet will probably be outside the window when it arrives at the receiver, and the receiver will generate an ACK and drop the packet. This would cause silent data loss. A second argument against changing the sequence numbers is that it puts hard state in the network - in case of a system crash at the proxy, the state information required to do the sequence number translation would be lost, and the connection would be irrecoverable even if a link existed between the communicating hosts.

The proxy also preserves URG flags and URG pointers, if any are present, to preserve the end-to-end semantics of this flag.

From the point of view of congestion control, splitting an end-to-end TCP connection into multiple connections is equivalent to breaking one feedback loop into multiple smaller loops. In order to achieve the same effect as end-to-end congestion control, we implement mechanisms to allow the different connection segments to sense and adapt to congestion on other segments. These are discussed in later chapters.

3.2.2.4 Connection Closing

When a host TCP has no more data to send, it sends a segment with the FIN flag set, signifying that it is closing its half of the full-duplex connection. When both hosts have sent a FIN and had it acknowledged, the connection is closed. Thus a proxy cannot generate a FIN because it does not know when a host has finished sending data.

Strictly speaking, to ensure that the hosts are never in a combination of states not permitted in TCP, the proxy would have to delay acknowledging a FIN until it has been acknowledged by the remote host, as it does in the case of a SYN. However, the situation here is somewhat different than for the SYN. Firstly, there is no way for the application to know through the TCP API when the FIN has been acknowledged, whereas an application can ascertain if the SYN exchange has been completed. Thus any application that needs to know if a transfer has completed successfully must implement its own application-layer acknowledgment mechanism - FTP is an example of this behavior. Secondly, the sender now has an estimate of the round-trip delay of its connection segment, and will retransmit the FIN unless it receives an acknowledgment in this time, creating unnecessary network traffic. Due to these differences, our implementation acknowledges a FIN as soon as it is received.

Different TCP implementations behave differently when in the TIME-WAIT state, especially with regard to incoming SYN packets. Some implementations

accept a new connection which has the same source and destination addresses and port numbers as one in TIME-WAIT, reusing the data structures associated with the old connection. Other stacks drop all packets received on a connection in TIME-WAIT, including SYN packets. Our proxy respects the decision of the end system in this matter. Connections are never placed in TIME_WAIT; their state information is destroyed as soon as the full-duplex close has been performed. This also makes the implementation simpler, as it avoids the need to keep a TIME_WAIT timer for the connection.

3.2.2.5 Errors and Exceptional Events

If a pair of hosts uses IP-level encryption, such as IPSEC, between them, the proxy does not have access to the TCP header, and so cannot extract the information needed to set up connection splitting. In this case the proxy forwards the packets, and does not attempt to split the connection.

If the proxy receives a TCP segment on a connection for which it has no state information, it simply forwards the segment. This segment may belong to a connection that was using another route and whose route has now changed, or it may belong to a connection for which connection splitting could not be set up due to the proxy being overloaded at the time. Alternatively the segment may belong to a connection that was set up before the proxy was activated. Yet another possibility is that the segment belongs to a connection for which splitting could not be set up due to asymmetric routing, as described earlier.

If the proxy receives an RST segment on a connection, it will forward the RST and destroy all data and state information associated with that connection. Since RSTs are used only in exceptional circumstances, and indicate that an irrecoverable error has occurred, continuing to send queued data would merely waste network resources as the data will be dropped at the destination anyway.

Chapter 4

Proxy Implementation

4.1 Overview

Our implementation of the TCP connection splitting proxy consists of the following components:

- Minor patches to the Linux kernel to fix various TCP bugs, export some symbols and to bypass unnecessary processing for sockets created by the proxy module.
- A loadable kernel module for the proxy functionality.
- A script for starting up and shutting down proxy operation, as well as setting various parameters at run time.
- A utility for obtaining profiling information about the proxy module. The proxy module includes a profiling facility which uses the cycle counter on the Pentium microprocessors to gather highly accurate data on the time taken to complete each of its function calls.

- Miscellaneous documentation files.

The proxy is implemented as kernel code for efficiency; this avoids the overhead of extra data copies and context switches, and it allows us to take advantage of the TCP/IP code already present in the kernel. Implementing the proxy as a loadable module allows us to develop it separately from the kernel itself, and simplifies its testing and use.

Our first implementation used version 2.1.36 of the Linux operating system. The proxy was implemented as an IP protocol handler. This was considered the best option available at the time, but it had the disadvantage that the proxy module also had to duplicate the IP forwarding functionality included in the Linux kernel.

When this version was being tested, version 2.1.95 of the Linux kernel was released, and further development of the proxy was moved to this version. The proxy now uses the firewalling capabilities of Linux; this new architecture eliminates the duplication of functionality between the proxy module and the kernel. The current version of the proxy module uses the kernel TCP and IP routines whenever possible, and only includes the minimal functionality required for connection splitting. Thus the design is compact and efficient - the module code requires less than 10 Kilobytes of memory, and adds no more than a few hundred clock cycles to the processing time of a packet.

Much of the operation of the proxy can be controlled using the Linux firewall administration tools. Thus the proxy can be configured with rules to select the TCP connections on which to carry out splitting. For example, we could set up the proxy not to carry out connection splitting on telnet connections, as these do not benefit much from connection splitting. A special class of rules known as accounting rules can be used to gather statistics on specific connections or groups of connections. The proxy can be dynamically reconfigured to collect statistics on any group of connections in this manner.

The proxy module also provides an interface for setting some internal parameters during its operation. These parameters are discussed in a later section. Finally, the module includes facilities for accurately profiling various parts of its execution, so that the actual processing time taken to perform various actions can be determined.

During the implementation, four bugs in the Linux kernel were found and fixed. Two of these related to the handling of zero-window probes and other out-of-window data fragments, one related to the handling of received SACK blocks, and one affected duplicate ACK counting when SACK was enabled. These bugfixes were merged into the mainstream Linux kernel, and were incorporated in version 2.1.126.

Our current implementation of the proxy runs on the Linux operating system, kernel version 2.1.95 (with the above bugfixes added). The implementation uses TCP, enhanced with timestamp, window scaling and SACK options, as well as

the FACK [17] congestion control algorithm and an increased value for the initial congestion window during connection startup.

4.2 Program Flow

The following sequence of events takes place when a packet is received at one of the proxies:

1. The interface card raises a hardware interrupt, causing Linux to execute the appropriate interrupt handling routine.

2. The interrupt handler performs a few basic functions, usually related to transferring data from the interface card to memory, schedules a software interrupt (also known as an interrupt bottom half) and exits. This is because further interrupts are disabled during the execution of the interrupt handler, so it must be very fast in order to avoid losing interrupts.

3. The bottom half is scheduled by the OS kernel in such a way that atomicity is ensured - specifically, once a bottom half is scheduled, no other instance of the same bottom half handler can execute until it completes.

All TCP/IP processing, including the proxy module, is done within the bottom half handler. The bottom half handler first performs appropriate link layer protocol processing (for example, removing the Ethernet header after verifying the checksum). Then it begins the IP processing.

First the IP checksum is verified and any necessary defragmentation performed by the kernel. Then the packet is checked against a set of input firewall rules. If it is rejected by any of these rules, the packet is discarded and processing ends. Otherwise the destination address of the packet is checked. If the packet is bound for the local machine, it is delivered to the appropriate higher-layer protocol for further processing.

If the packet is not destined for the local machine, it is checked against a series of forwarding rules. Each rules is checked in order of priority. The proxy module defines a medium-priority forwarding rule. This setup allows for higher-priority rules which might elect to forward the packet without performing connection splitting, and also allows for lower priority rules which can take appropriate actions on packets not processed by the proxy module. Once the proxy module receives a valid packet, it performs various TCP-level functions on it, and when this processing is completed the bottom half handler exits. If unexpected errors are encountered or system resources are unavailable for processing, the packet is skipped and handed down the firewall rule chain. In this case the packet will usually be forwarded (if it passes the output firewall rules), thus completing the bottom half processing.

A simplified flow diagram of the proxy module is shown in Figure 4.1. This diagram does not include the monitoring and profiling facilities present in the module. It also does not show the various parameters that can be set when the module is running. The functional units are described below.

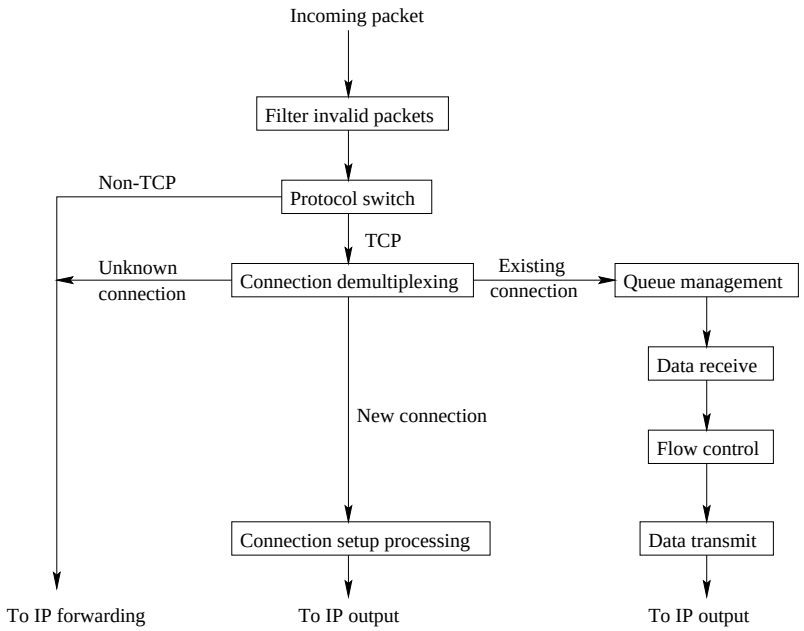


Figure 4.1: Functional flow diagram of proxy module.

- Filter invalid packets. Corrupted packets and other invalid requests are detected and filtered out.

- Protocol switch. TCP packets are separated from non-TCP packets. Non-TCP packets are handed down the forwarding firewall chain.

- Connection demultiplexing. A connection corresponding to the packet is searched for. If none exists and the packet is not a SYN, the packet is forwarded on the firewall chain. If a connection exists, the packet is delivered to the appropriate queue for processing by the kernel TCP functions. If the packet is a new SYN, data structures for handling a new connection are set up.

- Connection setup processing. If the packet is a SYN, it is added to the cache of pending SYN requests. If it is a SYNACK and matches a pending SYN, data structures are set up to handle the new connection and a SYNACK is relayed on. Otherwise the packet is handed down on the firewall chain.

- Queue management. If there is too much data already queued and pending on this connection, the packet is dropped without acknowledgment.

- Data receive. The packet is acknowledged and queued for downstream transmission.

- Flow control. Data is queued for transmission if there is a window available on the downstream connection segment. Otherwise it is left in the receive queue. This gives rise to a back-pressure effect, which allows congestion information to propagate upstream.

- Data transmit. Data is sent downstream with appropriate TCP options added.

In our first implementation we used very simple schemes for queue management and flow control. During the lifetime of a connection, the "back-pressure" algorithm described above is used for flow control; incoming segments on the upstream connection are served at a rate that matches the rate of transmission on the downstream connection. Thus when the downstream path

gets congested, the offered window on the upstream connection reduces correspondingly, and congestion information is propagated back to the sender.

However, the above scheme leads to very long queues, and may cause occasional stalls when the buffer on a proxy fills up. Therefore an additional mechanism is used to limit the size of the buffers at the gateways. If data is arriving on a connection at a much higher rate than it can be sent out, and if a large enough amount of data is already buffered for that connection, then an arriving packet is discarded without acknowledgment. This causes the sender to retransmit the segment and reduce its congestion window, and so keeps the buffers small without appreciably affecting end-to-end throughput. The implementation is careful not to drop more than one packet per window, to avoid causing serious performance degradation.

The threshold for dropping a segment in this way was set to the larger of half the receive window on the upstream segment and twice the congestion window on the downstream segment. This choice of threshold was motivated by the observation that in our setup, most of the queueing was happening at the satellite uplink. Therefore, using this threshold ensured that data transfer would never be interrupted for lack of data (as the upstream link, which had a much shorter round trip delay, would recover quickly and supply the data in time). In our implementation, once a packet is dropped, no more packets are dropped until a retransmission of the dropped packet has been received. This ensures that the upstream sender can recover from the loss using fast retransmit and so avoids a

TCP timeout.

This scheme gave us good results in our test setup. Stalls were eliminated, and shorter queue lengths were observed. However, this is not a general solution and certainly not an optimal solution - it reduces memory requirements by half, but it does not change the fact that the total memory requirement increases linearly with the number of active connections. We discuss flow control and memory issues further in Chapter 6.

4.3 Limiting Factors

We now look at some of the factors that limit the capacity of the proxy in terms of the number of connections and the bandwidth it can handle. In this discussion we leave out factors that are common to IP routers, such as interrupt service latency and link-layer protocol processing.

In order to perform splitting, the proxies must perform IP fragment reassembly. This is done by the IP firewalling code - firewall rules are applied to whole IP packets. Fragment reassembly is a complex operation that may require an extra data copy, and therefore can limit the maximum bandwidth handled by the proxy. However, IP fragmentation in the Internet is rare, since most current TCP implementations use path MTU discovery. Most hosts that do not perform path MTU discovery use the default MTU of 576 bytes, which is small enough to avoid fragmentation in nearly all cases. In our testing, which involved hosts

running five different popular operating systems, no fragmentation was observed.

The data structures required for connection splitting are stored in a static hash table. This can slow down lookup operations when the number of connections becomes large enough that a sizeable number of hash collisions begin to occur. This was not found to be a significant limitation for the link speeds at which our tests were carried out, because sufficient processing power was available to handle each packet. At higher link speeds this might be a significant constraint.

Running the proxy module was also found to increase the IP processing time (even for non-TCP traffic) in some cases. This is because when the gateways are working as IP routers, the IP forwarding rules are all present in cached memory. However, connection splitting requires the proxies to perform frequent lookups of TCP connections, which are represented by bulky data structures that may not all fit in cache. The increased probability of cache misses mainly increases the variance of IP processing times. We observed this effect while profiling the proxy, but it was not easily repeatable and so proved difficult to quantify.

The biggest constraint on the proxy is the large amount of memory required for the TCP transmit and receive buffers of the different connection segments. In the absence of efficient flow control algorithms, memory usage grows linearly with the number of concurrent connections. An associated problem is that brief stalls can occur in the data transfer whenever the receive buffer on a connection segment fills up, making the proxy advertise a zero window on its upstream

connection. Such stalls have also been reported earlier in the literature [18]. In our testing we found various reasons for these stalls.

- The Linux TCP implementation had bugs that caused incorrect handling of zero window probes and a failure to send window updates correctly.
- TCP generates ACKs containing window updates in response to zero window probes, but these are spaced at long and increasing intervals, so feedback to the sender about window opening is often delayed.
- If the zero window condition persists for a long period, then that TCP connection must restart from idle [5], i.e. reduce its congestion window to one segment and enter slow start. This causes a period of very slow data transfer, which appears similar to a stall.
- Stalls were more frequent when adjacent connections had drastically different round trip times. If there was congestion on a low-delay connection which was being fed by a long-delay connection, a stall would be much more likely to occur, as the low-delay connection would quickly drain the backlog in its buffers before the high-delay connection could start sending new data.

The first two problems can be eliminated by fixing the bugs and ensuring that the proxies generate prompt and correct window updates. However, the last two problems point to the need for a better flow control mechanism at the proxies, which prevents the zero window condition from occurring very often and which

maintains a smooth flow of data through the proxies. Such a flow control scheme would also reduce average queue lengths and hence memory requirements at the proxies.

4.4 Parameter tuning issues

4.4.1 Proxy parameters

The following parameters can be configured for the proxy module:

1. Send buffer size: This is the maximum size that can be allocated for the send buffer on a single TCP connection. For all the tests reported here, this was set equal to the product of the bandwidth and round-trip delay of the satellite link.

2. Receive buffer size: This is the maximum size that can be allocated for the receive buffer of a single TCP connection. For the tests reported here this was set to the product of the bandwidth and round-trip delay of the satellite link.

3. Default congestion window: This is the initial value of congestion window used on TCP connections over terrestrial and low-bandwidth satellite links. In all our tests this was set equal to one segment.

4. Higher congestion window: This is the initial value of the congestion window used on TCP connections over high bandwidth-delay product links. We tried values of one, four and sixteen segments.

5. Use of packet dropping for flow control: This determines whether the proxy's queue management algorithm, which uses packet drops to slow down the sender during periods of congestion on the downstream link, is used. Packet dropping was enabled during all the tests reported here.

The settings of these parameters have a significant effect on TCP performance. In particular, the send and receive buffer sizes should be set at least as large as the bandwidth-RTT product of the satellite link, otherwise the TCP on the proxies may be window-limited, leading to lower throughput.

When a TCP connection is set up, an estimate of round-trip delay is used along with any past knowledge of bandwidth usage on that route to calculate the initial offered window. If the other endpoint of the connection supports window scaling, then a suitable window scaling factor is chosen for the connection. If the initial offered window is large enough that the window scaling parameter is nonzero, then the higher congestion window is used on this connection, otherwise the default congestion window is used. This has the benefit of reducing startup times on connections with high bandwidth-delay products.

Therefore the default congestion window should be kept at one segment; it should definitely not be increased beyond four segments as this would risk

increased congestion for no real benefit. The higher congestion window can be set as high as desired, depending on the bandwidth-delay products expected from the network. However, setting it to very large values may cause congestion if a large number of connections are using the satellite link. The duration of slow start is reduced by one round trip each time this value is doubled; this fact, along with an estimate of the average traffic patterns on the link, can be used to decide an appropriate value for this parameter.

The use of the queue management policy appears to reduce the amount of queued data at the proxies without noticeably affecting throughput in our setup. However, its use can cause problems with hosts that do not implement the fast retransmit algorithm. When such hosts encounter a packet loss, they incur a TCP timeout, and so do not send any data for a significant length of time (the exact duration depends on round-trip time estimates). The proxy is then unable to fully utilize the send window on the downstream link, and throughput for the connection is reduced. Almost all servers in the present-day Internet implement the fast retransmit algorithm; it is part of the TCP Tahoe implementation, which is estimated to account for the majority of Internet hosts.

4.4.2 External effects

In our setup, two Cisco 1600 routers were used to convert the Ethernet frames output by the proxies to a serial format for transmission over the satellite link. If

the queue sizes on these routers were not configured large enough, many packets were dropped at the routers. This reduced the throughput as the congestion window was restricted to small values.

This effect is not directly attributable to the proxies. It is due to the fact that when a TCP segment is dropped due to congestion, it takes at least one round trip time for the sender to detect the loss and reduce its transmission rate. If during this time more packets are dropped, then the sender may suffer multiple losses within the same window, and may be unable to recover without a timeout. Configuring the router buffers to be roughly as large as the bandwidth-delay product of the satellite link appeared to reduce the loss rates to acceptable levels in all the cases we tested.

The behavior of the system with bit errors on the satellite link showed some limitations of TCP. At low error rates with uncongested links, the congestion window on the satellite link can grow quite large. However, when a single packet is lost due to bit errors, the congestion window is halved, and TCP enters congestion avoidance. In this case the link is underutilized for a long time, until the congestion window can grow back to its former value. Throughput decreases appreciably, even at moderate BER values.

At higher error rates, when the rate of errors is close to one error per round trip, the congestion window is halved very frequently, even when the fast recovery algorithm [5] is used. TCP throughput degrades very sharply at this point. These effects are explored in the next chapter.

Chapter 5

Performance of Proxy Implementation

5.1 Single TCP Connection

5.1.1 Test methodology

The test configuration is shown in Figure 5.1. The proxy module was run on a pair of Pentium PCs clocked at 166 MHz. The server host was a PC running Microsoft Windows NT Workstation 4.0 with the default TCP/IP parameters, and the client host was running Microsoft Windows 95 with the default parameters. We used the FTP server and the HTTP server from the NT Peer Web Services software. The client for the FTP testing was the standard FTP

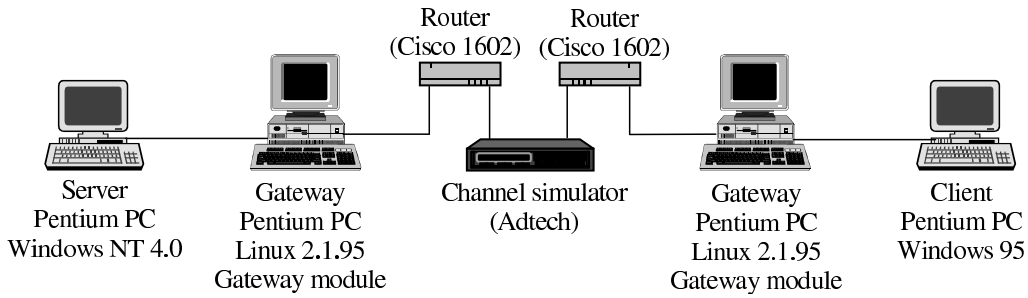


Figure 5.1: Test configuration. The proxy module was run on the two gateways.

client supplied with Windows 95, while the HTTP client was Netscape Communicator 4.05. Tests were carried out using a data channel simulator to simulate the satellite channel, as well as over a commercial Ku band satellite link.

We measured throughput for single FTP connections using different file sizes, with different data rates, delays and error rates on the simulated satellite link, with no other traffic on the link. Various combinations of the following parameter values were used:

- File sizes: 10 KB, 100 KB, 1000 KB, 10000 KB, 100000 KB
- Link rates: 384 kbps, 1.536 Mbps, 8 Mbps
- One-way link delay: 0, 250 ms
- Bit Error Rates: 0, 10^{-9}, 10^{-8}, 10^{-7}, 10^{-6}

To provide a baseline for comparison, identical tests were carried out with connection splitting disabled on the proxy machines.

HTTP tests using different kinds of webpages were carried out over the same range of link rates and delays. In this case we measured the total time required to load a page. Throughput is not meaningful in this situation, since the request-response mechanism of HTTP causes unavoidable periods of zero link utilization.

Three webpages were used for testing:

- Test Page 1 comprised an HTML document of 356 bytes, which referred to one image of size 391 KB.
- Test Page 2 comprised an HTML document of 1.7 KB, which referred to 16 images with sizes varying from 19 KB to 100 KB. The total size of the images was 669 KB.
- Test Page 3 comprised an HTML document of 1.6 KB, which referred to 16 small images of nearly equal sizes, with total size 262 KB.

It can be argued that these tests are quite simplistic and do not reflect real-life situations. However, they do represent important benchmarks, and serve to establish some basic capabilities of the proxies. More realistic situations are explored in the next section.

5.1.2 Results - Simulated Satellite Channel

Test results for FTP transfers are presented in Table 5.1, Table 5.2, Table 5.3 and Table 5.4.

Figure 5.2 shows the degradation of TCP performance when delay and errors are added to a link. The end-to-end TCP transfer is limited to a constant maximum transfer rate, independent of link bandwidth, by the fact that its offered window is restricted to small values. Thus link utilization decreases with increasing link bandwidth. The connection splitting approach, which uses much larger window sizes on the satellite link, yields better throughput, and always

Line rate (kbps)	File size (KB)	End-to-end		Proxy, IW=1		Proxy, IW=4		Proxy, IW=16	
		Time (s)	Throughput (KB/s)	Time (s)	Throughput (KB/s)	Time (s)	Throughput (KB/s)	Time (s)	Throughput (KB/s)
384	10	0.22	46.55	0.22	46.55	0.33	31.03	0.22	46.55
384	100	2.36	43.39	2.20	46.55	2.20	46.55	2.20	46.55
384	1000	22.41	45.69	22.57	45.37	22.58	45.35	22.52	45.47
384	10000	224.42	45.63	229.42	44.63	225.91	45.33	229.42	44.63
1536	10	0.06	170.76	0.16	64.00	0.22	46.55	0.16	64.00
1536	100	0.66	155.15	1.15	89.04	0.49	208.98	0.49	180.50
1536	1000	5.55	184.50	5.99	170.95	5.76	177.58	5.77	177.47
1536	10000	56.30	181.88	57.18	179.08	56.41	181.53	56.36	181.69
1536	100000	562.16	182.15						
8000	10	0.02	513.19	0.17	60.24	0.16	64.00	0.17	60.24
8000	100	0.55	186.18	0.44	232.73	0.33	310.30	0.39	262.56
8000	1000	1.98	517.17	1.26	812.70	1.49	687.25	1.32	775.76
8000	10000	19.39	528.11	12.41	825.14	12.08	847.68	12.09	846.98
8000	100000	190.48	537.59	123.09	831.91	120.56	849.37	120.40	850.50

Table 5.1: FTP test results for simulated channel with channel simulator set to zero delay and zero BER.

Line rate (kbps)	File size (KB)	End-to-end		Proxy, IW=1		Proxy, IW=4		Proxy, IW=16	
		Time (s)	Throughput (KB/s)	Time (s)	Throughput (KB/s)	Time (s)	Throughput (KB/s)	Time (s)	Throughput (KB/s)
384	10	1.70	6.02	1.70	6.02	1.71	5.99	1.70	6.02
384	100	8.68	11.80	4.01	25.54	4.07	25.16	4.01	25.54
384	1000	79.92	12.81	25.87	39.58	25.86	39.60	25.87	39.58
384	10000	787.80	13.08	251.01	40.80	243.43	42.07	238.87	42.87
1536	10	1.54	6.65	1.54	6.65	1.04	9.85	0.60	17.07
1536	100	7.58	13.51	3.68	27.83	2.19	46.76	1.27	80.63
1536	1000	68.38	14.98	9.34	109.64	7.86	130.28	6.97	146.92
1536	10000	672.78	15.22	59.81	171.20	58.17	176.04	57.28	178.77
1536	100000	7017.45	14.59						
2048	10	1.60	6.40	1.54	6.65	1.05	9.75	0.55	18.62
2048	100	10.66	9.61	3.63	28.21	2.64	38.79	1.21	84.63
2048	1000	67.45	15.18	8.07	126.89	6.64	154.22	5.77	177.47
2048	10000	663.00	15.44	46.19	221.69	44.27	231.31	43.40	235.94
2048	100000					425.83	240.47		
8000	10	1.60	6.40	1.54	6.65	0.99	10.34	0.55	18.62
8000	100	8.19	12.50	3.52	29.09	2.09	49.08	1.10	93.09
8000	1000	65.53	15.63	5.93	172.68	4.50	227.56	4.07	251.60
8000	10000	644.22	15.90	15.82	647.28	13.84	739.88	15.00	650.00
8000	100000			130.45	784.98	122.26	837.56	123.20	831.70

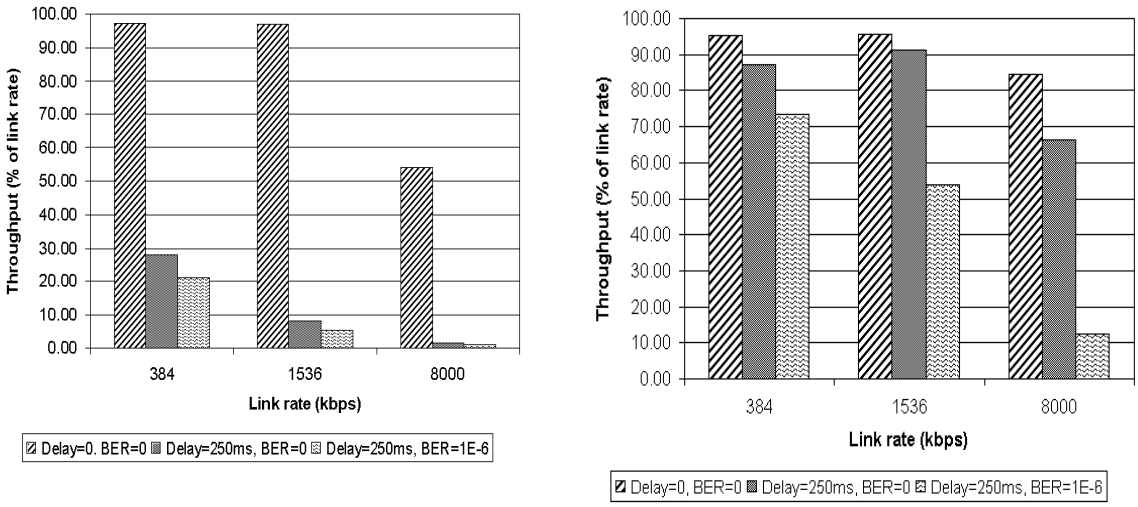
Table 5.2: FTP test results for simulated channel. Delay=250 ms each way, BER=0.

Line rate (kbps)	File size (KB)	End-to-end		Proxy, IW=1		Proxy, IW=4		Proxy, IW=16	
		Time (s)	Throughput (KB/s)	Time (s)	Throughput (KB/s)	Time (s)	Throughput (KB/s)	Time (s)	Throughput (KB/s)
384	10	1.94	5.28	1.70	6.02	1.76	5.82	1.70	6.02
384	100	16.09	6.36	4.72	21.69	4.61	22.21	4.94	20.73
384	1000	105.34	9.72	30.04	34.09	31.58	32.43	37.18	27.54
384	10000	1032.98	9.91	296.98	34.48	299.34	34.21	293.03	34.95
1536	10	1.54	6.65	1.54	6.65	1.04	9.85	0.60	17.07
1536	100	10.71	9.56	3.74	27.38	2.59	39.54	1.49	68.72
1536	1000	111.39	9.19	19.83	64.48	11.10	92.25	11.59	88.35
1536	10000	1039.68	9.85	101.28	101.11	110.34	92.80	100.24	102.15
8000	10	1.54	6.65	1.54	6.65	1.05	9.75	0.55	18.62
8000	100	8.18	12.52	3.68	27.83	2.09	49.08	1.10	93.09
8000	1000	119.73	8.55	8.62	133.16	8.78	116.63	11.70	87.52
8000	10000	1005.47	10.18	84.04	121.85	85.85	119.28	83.44	122.72
8000	100000			869.25	117.80	715.74	143.07	832.34	123.03

Table 5.3: FTP test results for simulated channel. Delay=250 ms each way, BER=10^{-6}.

Line rate (kbps)	Bit Error Rate	Time (s)	Throughput (KB/s)
384	10^{-9}	242.93	42.15
384	10^{-8}	242.06	40.39
384	10^{-7}	259.25	39.50
1536	10^{-9}	58.16	176.07
1536	10^{-8}	58.16	176.07
1536	10^{-7}	63.33	161.69
8000	10^{-9}	15.99	640.40
8000	10^{-8}	16.26	629.77
8000	10^{-7}	21.75	470.80

Table 5.4: FTP test results for simulated channel. In these tests connection splitting was enabled with the proxies using an initial congestion window (IW) of 1 segment. Delay=250 ms each way, BER=0. File size=1000 KB.



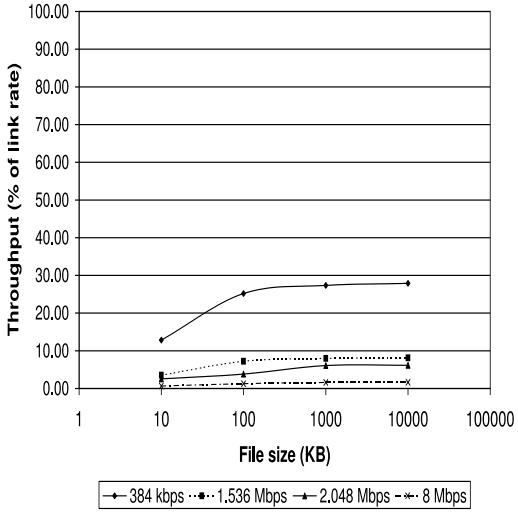
(a) End-to-end TCP

(b) Proxy, IW=1

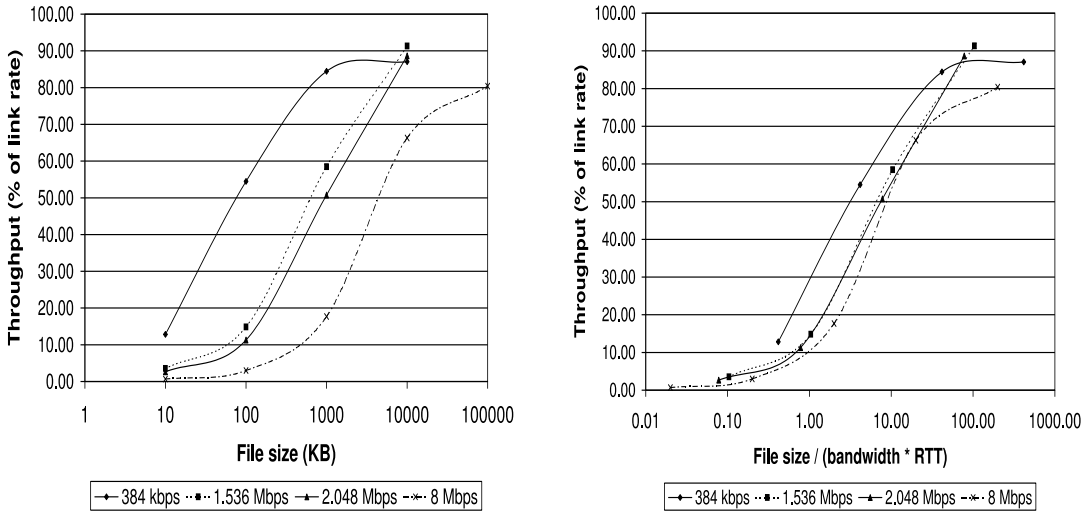
Figure 5.2: Effect of delay and errors on FTP performance. File size=10 MB.

uses a large fraction of link bandwidth. Both approaches suffer significantly from bit errors, especially at higher link speeds. This is due to TCP assuming that all losses are caused by congestion.

Figure 5.3 shows the dependence of throughput on the size of the file transferred. As expected, the end-to-end approach is limited to a constant rate due to the small size of the window. The proxy approach, which uses larger windows, does considerably better. As the file size is increased, the cost of low utilization during slow start is amortized over a longer interval of near-line-rate transfer, and the utilization improves. This is shown clearly in Figure 5.3(b), which plots the throughput against the ratio of file size to bandwidth-RTT product. Clearly, for good performance, the file size should be an order of magnitude larger than the bandwidth-RTT product.

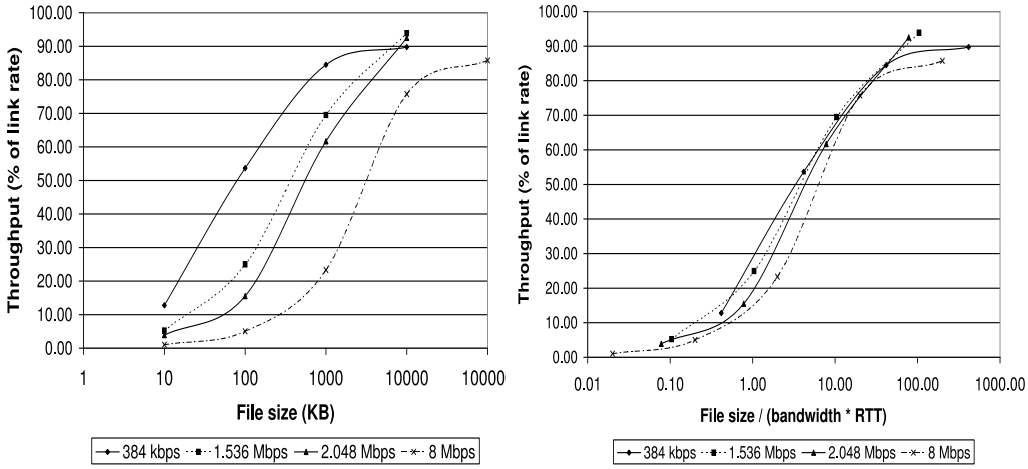


(a) End-to-end TCP

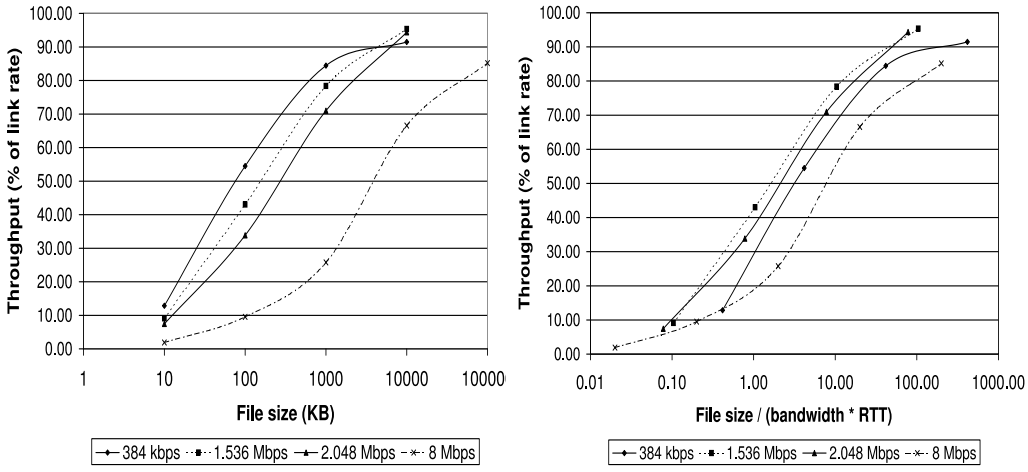


(b) Proxy, IW=1

Figure 5.3: Effect of file size on TCP performance at different link speeds. Delay=250 ms each way, BER=0.



(a) Proxy, IW=4

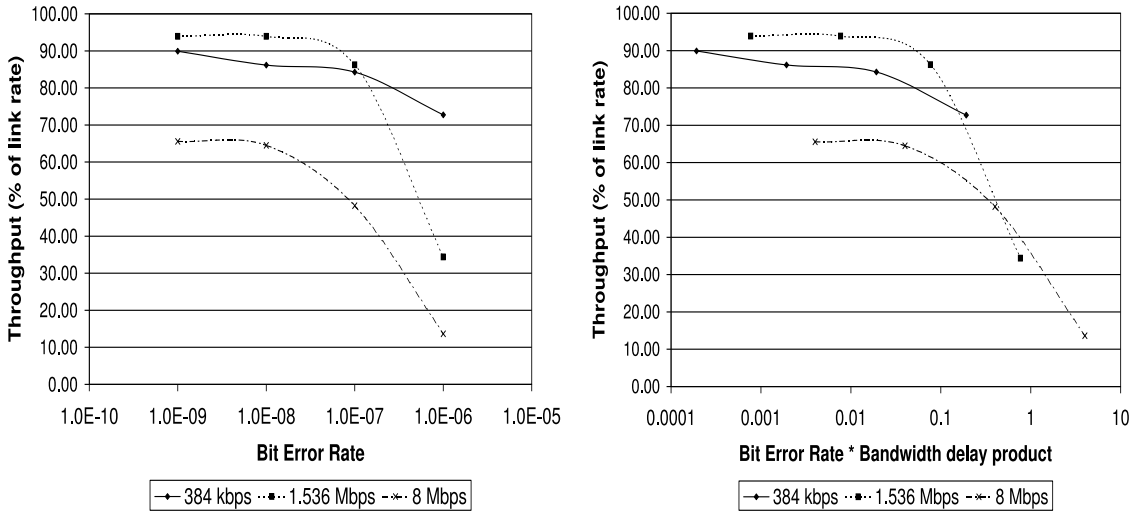


(b) Proxy, IW=16

Figure 5.4: Effect of increasing IW on TCP performance at different link speeds. Delay=250 ms each way, BER=0, proxies enabled.

The effect of slow start on throughput can be reduced by increasing the value of the initial congestion window from its standard value of one segment. As Figure 5.4 shows, this improves throughput at all link rates, but the increase is appreciable only for medium-sized files at higher link speeds. This is because each time the initial window is doubled, the slow start phase is shortened by one round trip. Thus increasing the initial window from one segment to four segments reduces the duration of slow start by two round trip times, and reduces the time required for the transfer by at most two round trips. For very large transfers, this is a small fraction of the total number of round trips required, and has little effect on average throughput. For small transfers, increasing the window fails to make any difference beyond a point, since the transfer time cannot be reduced to less than one round trip time. For medium-sized files (i.e. those with size approximately equal to the bandwidth-RTT product), the entire transfer is completed during slow start, so doubling the initial window has a significant effect.

In other words, doubling the initial congestion window can at most double the average throughput. However this increase is achieved at file sizes for which throughput is low to begin with, and not at large file sizes where utilization is relatively high. Thus increasing the initial congestion window, at least up to sizes that are small compared to the bandwidth-RTT product, does not change the essential characteristic of TCP that high link utilization can be achieved only when the transfer size is an order of magnitude larger than the bandwidth-RTT



(a) Throughput vs. BER

(b) Throughput vs normalized BER

Figure 5.5: Effect of bit errors on FTP performance at different link speeds. File size=1 MB, delay=250 ms each way. Proxies enabled, IW=1.

product.

Figure 5.5 shows the link utilization achieved by the split-connection system when bit errors are present on the satellite link. The split-connection system performs fairly well even at relatively high error rates. This is mostly due to our use of SACK information and the FACK algorithm on the satellite link. Throughput is more affected by bit errors at higher link speeds, since the susceptibility of TCP to errors (in terms of reducing the congestion window) depends on the number of errors per round trip time and not on the absolute BER. Figure 5.5(b) plots the throughput as a function of the error rate per round trip. We see that performance drops sharply when the error rate approaches one error per round trip, as is characteristic of TCP.

Line rate (kbps)	Test Page	Transfer time (s)	
		End-to-end TCP	Proxy, IW=1
384	1	9.60	9.60
384	2	16.50	17.50
384	3	6.87	7.40
1536	1	3.10	3.06
1536	2	4.85	4.85
1536	3	2.42	2.73
8000	1	1.15	1.35
8000	2	3.78	4.00
8000	3	1.50	1.50

Table 5.5: HTTP test results for simulated channel. Delay=0, BER=0.

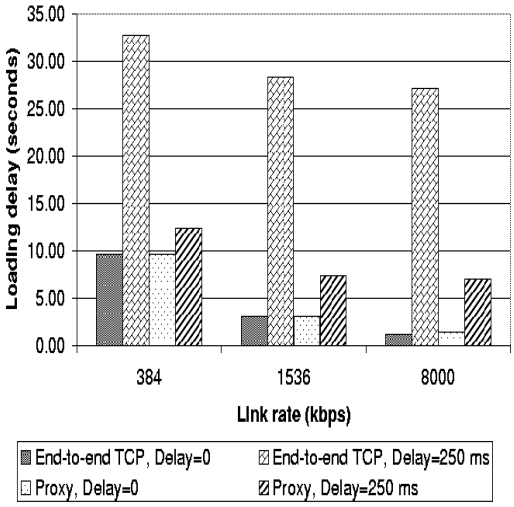
Line rate (kbps)	Test Page	Transfer time (s)			
		End-to-end TCP	Proxy, IW=1	Proxy, IW=4	Proxy, IW=16
384	1	32.69	12.34	11.78	11.65
384	2	37.21	12.72	11.28	11.34
384	3	18.48	12.00	10.81	12.20
1536	1	28.34	7.37	6.74	6.59
1536	2	32.90	11.61	10.80	10.84
1536	3	16.78	9.30	8.18	7.96
8000	1	27.10	7.05	6.09	4.98
8000	2	30.98	9.87	9.62	8.90
8000	3	16.00	7.91	7.91	7.36

Table 5.6: HTTP test results for simulated channel. Delay=250 ms each way, BER=0.

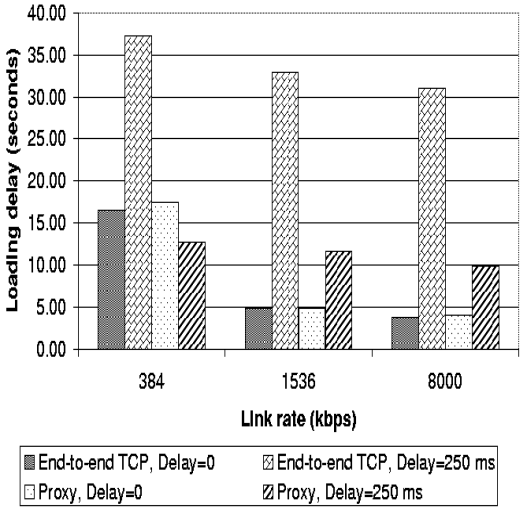
Test results for HTTP transfers are presented in Table 5.5 and Table 5.6. In this case performance is measured by the total time required for each webpage to load, as measured by a stopwatch. Due to the imprecise nature of this measurement method, as well as due to the variance introduced by the HTTP client (which requires a significant amount of time and processing power for its image manipulation and rendering algorithms), these results are by no means exact. However they serve very well to illustrate some general trends.

HTTP, like FTP, uses a request-response mechanism, wherein the client requests one object at a time from the server. However, each FTP transfer consists of a single file, whereas a single webpage typically consists of several smaller objects, each of which must be requested separately by the client. Due to the request-response mechanism, there is an interval of one round trip between the time that the client finishes receiving an object and the time that it begins to receive the next object. During this time, there is no traffic on the link except for the request by the client. Therefore the traffic generated by an HTTP session is intermittent in nature, with long pauses. These pauses are the major limiting factor for HTTP performance over long delay paths, so increasing the link rate or the initial congestion window does not improve performance. The only way to overcome this problem seems to be to add request aggregation capabilities to HTTP.

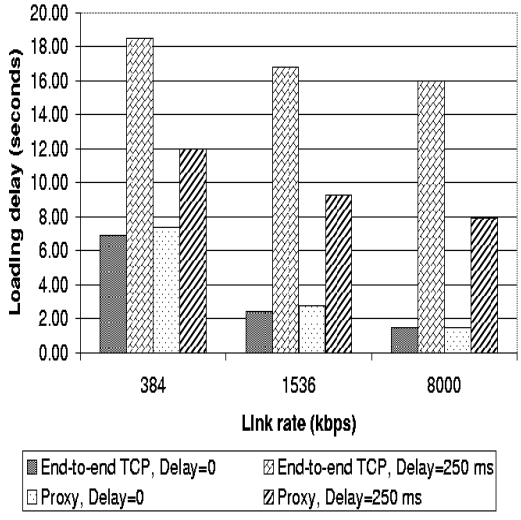
Figure 5.6 bears out this analysis. HTTP performance declines markedly with increasing link delay, and in this case the improvements due to using the proxy,



(a) Test Page 1

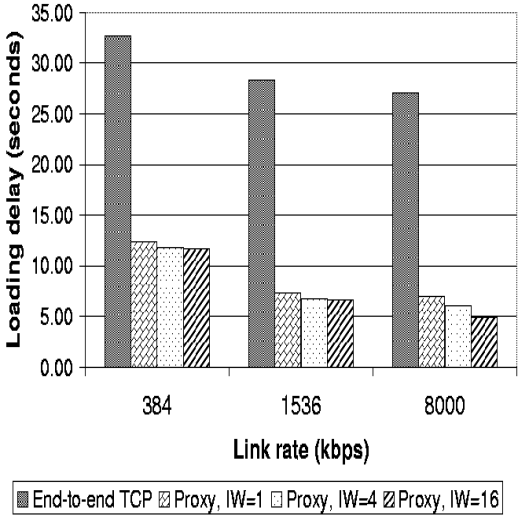


(b) Test Page 2

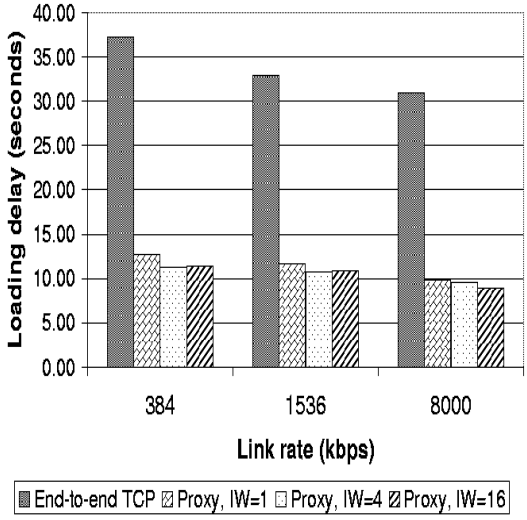


(c) Test Page 3

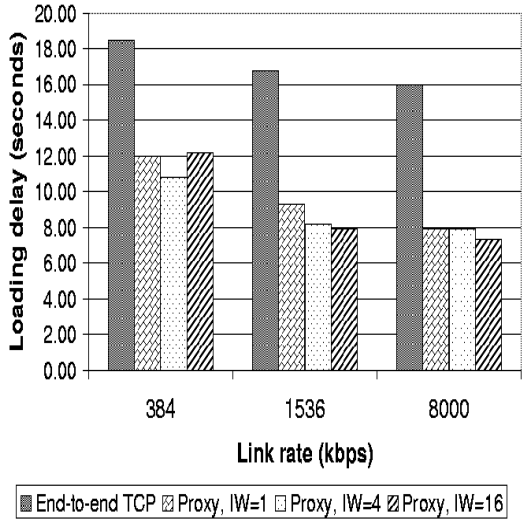
Figure 5.6: Effect of delay on HTTP performance for three sample webpages. For these tests, the proxies were set to IW=1 and the channel simulator to BER=0.



(a) Test Page 1



(b) Test Page 2



(c) Test Page 3

Figure 5.7: Effect of increasing IW on HTTP performance over simulated satellite link on three sample webpages. For these tests, the channel simulator was set to BER=0.

File Size (KB)	End-to-end TCP		Proxy, IW=1		Proxy, IW=4		Proxy, IW=16	
	Time (s)	Through put (KB/s)	Time (s)	Through put (KB/s)	Time (s)	Through put (KB/s)	Time (s)	Through put (KB/s)
10	1.60	6.40	1.65	6.21	1.10	9.31	0.60	17.07
100	10.65	9.62	3.24	31.60	2.80	36.57	1.27	80.63
1000	69.54	14.73	7.69	133.16	6.70	152.84	5.87	174.45
10000	683.66	14.98	45.31	226.00	44.27	231.31	43.44	235.73
100000					426.00	240.38		

Table 5.7: FTP test results for Ku band satellite link.

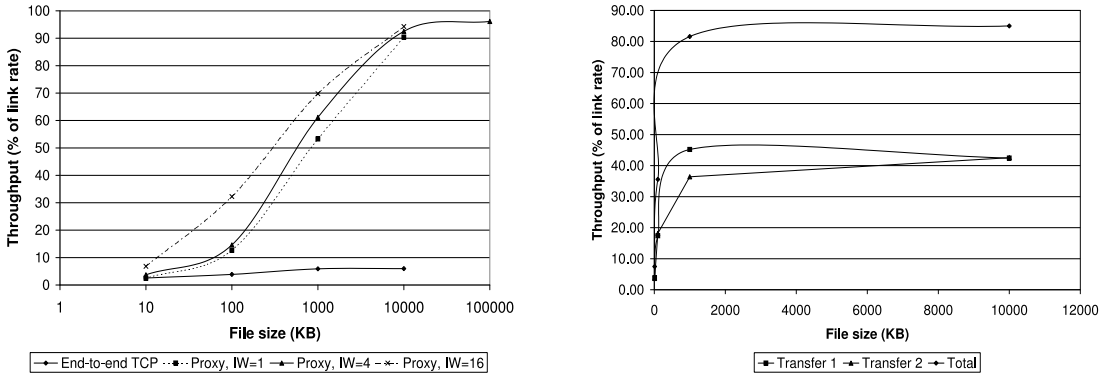
though still large, are not as remarkable as in the case of FTP. Further, as seen from Figure 5.7, increasing the initial congestion window does not have much effect on performance, which indicates that the performance is limited mainly by the request-response delays when the proxy is used. It is worth noting that for test pages number 2 and 3, the request-response delays contribute 8 seconds (16 round trips of 0.5 second each), which is a major portion of the total loading delay.

5.1.3 Results - Ku band satellite

Table 5.7 shows the results obtained when the above FTP tests were repeated over a commercial Ku band satellite link. The bandwidth of the satellite link was E1 (2.048 Mbps). During the tests, the satellite modems on the two ends of the link reported bit error rates of 10^{-6} and 4.5×10^{-5} respectively on the raw channel. The Reed-Solomon coding used by the modems reduced the error rate

File Size (KB)	Transfer 1		Transfer 2	
	Time (s)	Through put (KB/s)	Time (s)	Through put (KB/s)
10	1.04	9.85	1.10	9.31
100	2.30	44.52	2.20	46.55
1000	8.85	115.71	10.99	93.18
10000	94.26	108.64	93.98	108.96

Table 5.8: FTP test results for Ku band satellite link. Two simultaneous identical transfers.



(a) Single FTP session.

(b) Two simultaneous FTP sessions.

Figure 5.8: FTP Performance over Ku band satellite link.

to 10^{-12} or better in both directions, and so for all practical purposes the channel was error-free. Thus the results obtained were very similar to those obtained with the channel simulator at the same link speed with the BER set to zero.

Figure 5.8(a) shows the variation of throughput with changing file size. As expected, performance with proxies enabled is much better than with end-to-end TCP. A simple calculation shows that the bandwidth-RTT product of the satellite link is approximately 125 KB. We see that performance picks up when

file size is about an order of magnitude larger than the bandwidth-RTT product, and that only medium-sized file transfers are affected by increasing the initial congestion window.

Figure 5.8(b) shows the performance achieved by two identical simultaneous FTP connections sharing the satellite link. We see that for large enough file sizes, link utilization is still high, with each of the two connections getting a roughly equal share of bandwidth.

5.2 Multiple TCP Connections

5.2.1 Test Methodology

A setup similar to the single connection case with the channel simulator was used. The server and client machines in this case were PCs running Linux. As before, all links were 10 Mbps Ethernet links, except for the simulated satellite link, which was an 8 Mbps serial link. The TCP benchmarking tool DBS (Distributed Benchmarking System, [19]) was used to create multiple connections between these two machines. In the tests for measuring fairness to flows with different RTTs, a third machine was used as an additional client. This machine was also a Pentium PC running Linux connected to the server through a Cisco 7000 series router.

We measured application-level throughput at the receiver for different

numbers of parallel connections transferring data from the server to the client(s). Receiver throughput is measured instead of sender throughput because the latter merely measures how quickly the program could write its data to the TCP send buffer, and is in no way related to the actual rate of data transfer. All connections were made before the measurement was started, so that differing connection setup times did not affect throughput measurements. This was done using the BEFORE connection mode of DBS.

We measured throughput for transfers of size 10 KB and 100 KB. These sizes were chosen to approximate the typical size range of HTTP objects, so that the measurements would be somewhat realistic. Tests were carried out with the channel simulator set to 250 ms delay each way. In none of the tests were any bit errors introduced by the channel simulator. This was done for two reasons: firstly, the preceding tests show that bit errors do not have an appreciable effect on performance on typical Ku band satellite channels. Secondly, at the file sizes we tested, the probability of any single transfer encountering a bit error, even at relatively high error rates such as 10^{-6}, is so low that any results obtained from a small number of tests would not be statistically significant.

We performed tests with 10, 20, 30 and 40 simultaneous connections. These numbers were chosen so a wide range of system behavior could be explored. We know that the total bandwidth-RTT product of the link is 500 kB, or approximately 343 segments at our chosen MTU of 1500 bytes. Therefore when 10 connections share the channel, the optimal window for each of them is about

No. of conns.	Mean throughput (Mbps)				Standard deviation (Mbps)			
	End-to-end TCP	Proxy IW=1	Proxy IW=4	Proxy IW=16	End-to-end TCP	Proxy IW=1	Proxy IW=4	Proxy IW=16
1	0.0653	0.0611	0.1015	0.2146				
10	0.0613	0.0581	0.0938	0.1882	0.0002	0.0010	0.0008	0.0044
20	0.0617	0.0585	0.0979	0.1490	0.0004	0.0006	0.0023	0.0059
30	0.0609	0.0580	0.0953	0.1374	0.0014	0.0011	0.0027	0.0059
40	0.0606	0.0581	0.0901	0.1244	0.0016	0.0011	0.0094	0.0045

Table 5.9: Throughput for simultaneous TCP connections. Individual transfer size=10 KB, delay=250 ms each way.

50 kB, and we can expect to see some effects of limited receive window sizes, as the client and server were using the Linux default window sizes of 32 KB. On the other hand, when 40 connections share the channel, the optimal window size for each is around 12.5 kB, or about 9 segments. So at this point we do not expect to see any effects due to limited receive window sizes; instead we expect to see severe congestion when we increase the initial window on the proxies to 16 segments.

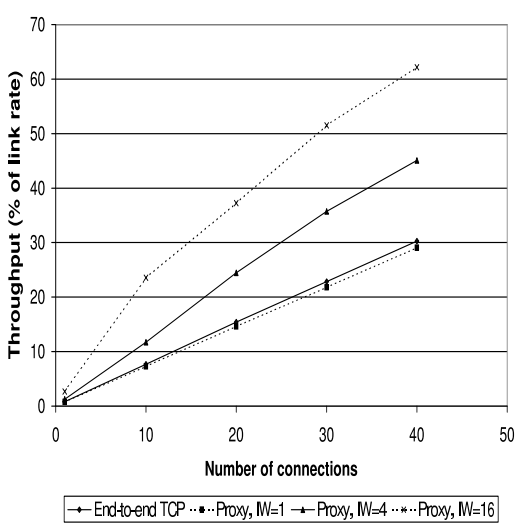
We note that some of the results shown here, especially for the 10 KB transfer size, may not be very accurate, due to limits imposed by timer granularity on the test systems. However, the general trends are quite reliable, and reveal some interesting insights.

5.2.2 Results - Simulated Channel

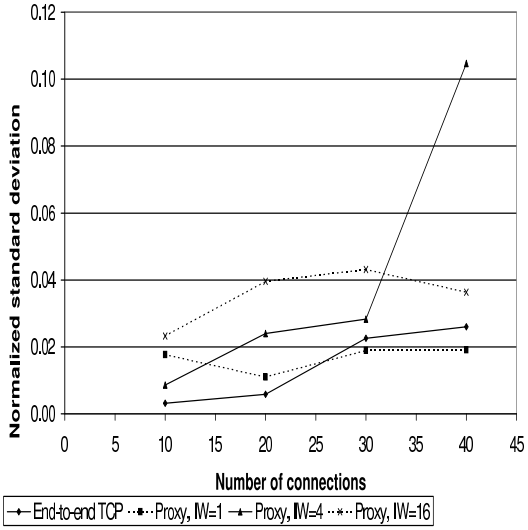
Table 5.9 and Table 5.10 shows the results when different numbers of parallel TCP connections transfer data simultaneously over the simulated link. These

No. of conns.	Mean throughput (Mbps)				Standard deviation (Mbps)			
	End-to-end TCP	Proxy IW=1	Proxy IW=4	Proxy IW=16	End-to-end TCP	Proxy IW=1	Proxy IW=4	Proxy IW=16
1	0.2222	0.2644	0.3286	0.8039				
10	0.2168	0.2517	0.3201	0.5387	0.0023	0.0034	0.0071	0.1293
20	0.2017	0.2127	0.2415	0.3916	0.0037	0.0133	0.0400	0.1206
30	0.1170	0.1496	0.2024	0.2706	0.0292	0.0370	0.0306	0.1037
40	0.1030	0.1519	0.1770	0.1971	0.0310	0.0229	0.0357	0.0452

Table 5.10: Throughput for simultaneous TCP connections. Individual transfer size=100 KB, delay=250 ms each way.

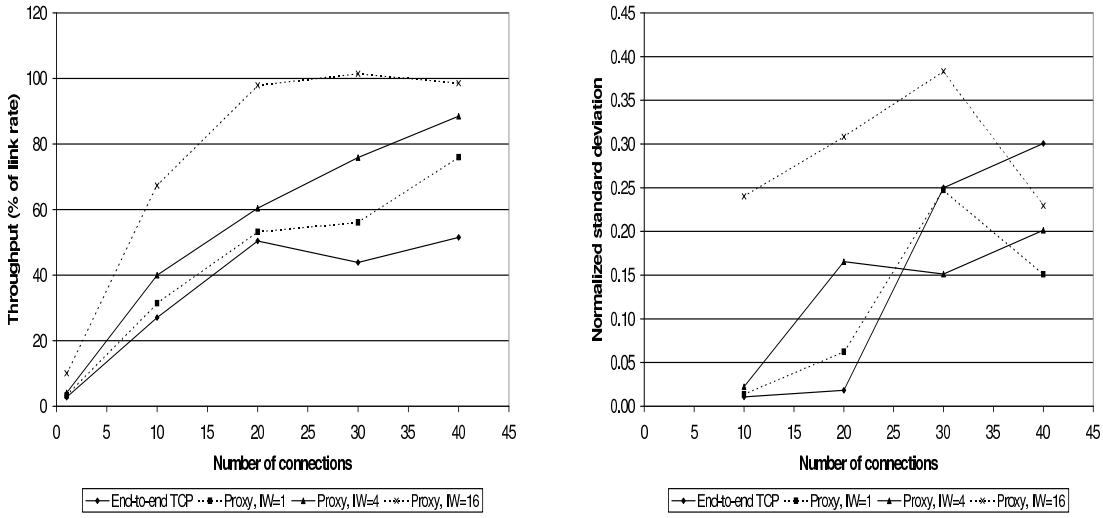


(a) Mean utilization.



(b) Normalized standard deviation.

Figure 5.9: Throughput for simultaneous 10 KB transfers. Delay=250 ms each way.



(a) Mean utilization.

(b) Normalized standard deviation.

Figure 5.10: Throughput for simultaneous 100 KB transfers. Delay=250ms each way.

results are plotted graphically in Figure 5.9 and Figure 5.10. As expected, throughput is low for small transfers, as the transfer time is dominated by delays due to slow start. Even 40 transfers of 10 KB each only represent a total transfer of 400 KB, which is less than the bandwidth-RTT product of our link, so it is not surprising that the use of proxies does not help much. Increasing the initial window makes a large difference to throughput - when IW is increased to 16 segments, the entire transfer can be accomplished in a single round-trip time.

More interesting results are observed for the larger transfer size of 100 KB. Here each transfer involves approximately 70 segments, which means that, starting with IW=1, the transfer takes about six round trips to complete, with the sender's congestion window growing to 32 segments. Increasing the IW to

4 segments reduces the transfer time by two round trips, and so yields significant benefit. Further increasing IW to 16 segments reduces the transfer time by another round trip, improving throughput even more.

With 40 simultaneous transfers of 100 KB each, the total amount of data transferred is 4000 KB, which is about an order of magnitude larger than the bandwidth-RTT product of the link. Therefore it is not surprising that the proxies perform well. However, it is remarkable that end-to-end TCP does not do as well, since for this case the optimal window size is only 12.5 KB. A possible explanation is provided by Figure 5.10(b). These plots show the standard deviation of the throughput of individual connections in each experiment, normalized by the mean throughput for connections in that experiment. We see that when the number of connections becomes large enough for the sum of the senders' windows to exceed the path bandwidth-RTT product, the standard deviation rises sharply. This occurs when the number of connections is about 20 (20 connections, each with window 32, make a total window of 640 segments, which is slightly less than twice the bandwidth-RTT product of the simulated satellite link). This indicates that, at least for the transfer sizes and link parameters involved, the bandwidth sharing mechanisms of TCP do not work very well when utilization is high. This observation, if true, would indicate that increasing the IW too much may have harmful effects on throughput.

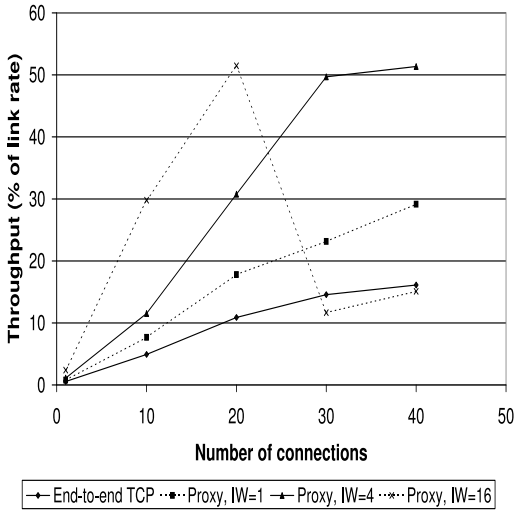
The measurements in Table 5.11 and Table 5.12, plotted in Figure 5.11 and Figure 5.12, show how proxies can improve performance by performing localized

No. of conns.	Mean throughput (Mbps)				Standard deviation (Mbps)			
	End-to-end TCP	Proxy IW=1	Proxy IW=4	Proxy IW=16	End-to-end TCP	Proxy IW=1	Proxy IW=4	Proxy IW=16
1	0.0436	0.0618	0.0885	0.1919				
10	0.0393	0.0614	0.0923	0.2382	0.0001	0.0003	0.0079	0.0043
20	0.0436	0.0712	0.1230	0.2058	0.0025	0.0015	0.0081	0.0064
30	0.0388	0.0617	0.1325	0.0311	0.0098	0.0043	0.0137	0.0163
40	0.0322	0.0583	0.1027	0.0302	0.0098	0.0046	0.0204	0.0240

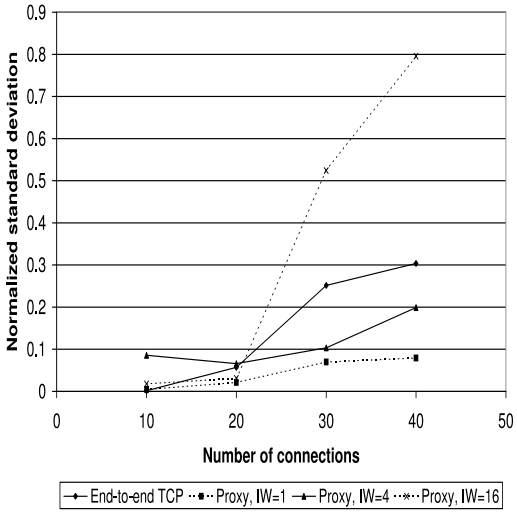
Table 5.11: Throughput with congested terrestrial link. Individual transfer size=10 KB, delay=250 ms each way.

No. of conns.	Mean throughput (Mbps)				Standard deviation (Mbps)			
	End-to-end TCP	Proxy IW=1	Proxy IW=4	Proxy IW=16	End-to-end TCP	Proxy IW=1	Proxy IW=4	Proxy IW=16
1	0.1546	0.2584	0.3239	0.9274				
10	0.1091	0.2431	0.2087	0.3201	0.0188	0.0122	0.1027	0.1772
20	0.0877	0.1268	0.1068	0.1111	0.0320	0.0371	0.0692	0.1092
30	0.0712	0.1167	0.1091	0.0966	0.0314	0.0341	0.0467	0.0624
40	0.0662	0.0948	0.0928	0.0975	0.0358	0.0304	0.0431	0.0567

Table 5.12: Throughput with congested terrestrial link. Individual transfer size=100 KB, delay=250 ms each way.

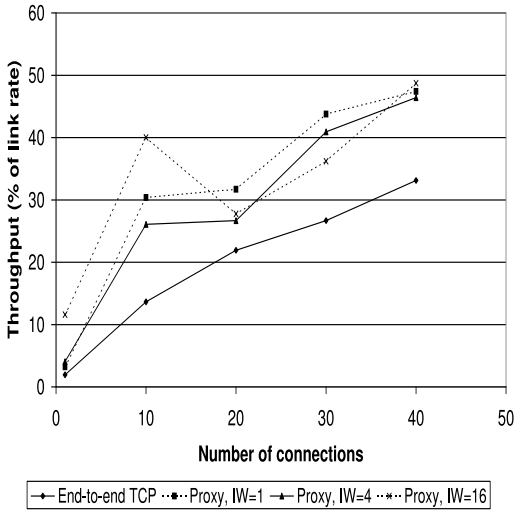


(a) Mean utilization.

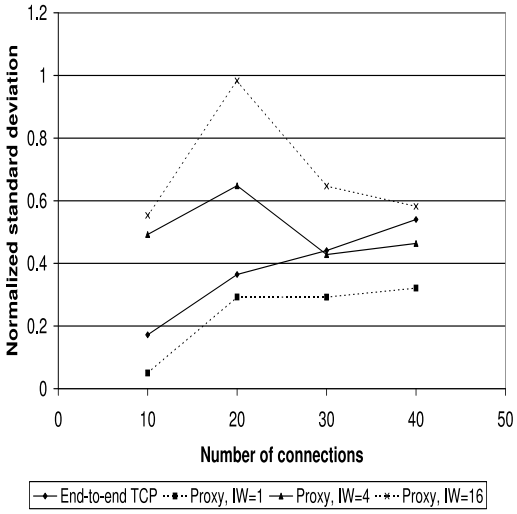


(b) Normalized standard deviation.

Figure 5.11: Throughput for 10 KB transfers with congested terrestrial link. Congested link is downstream of satellite link, which has delay=250 ms in each direction.



(a) Mean utilization.



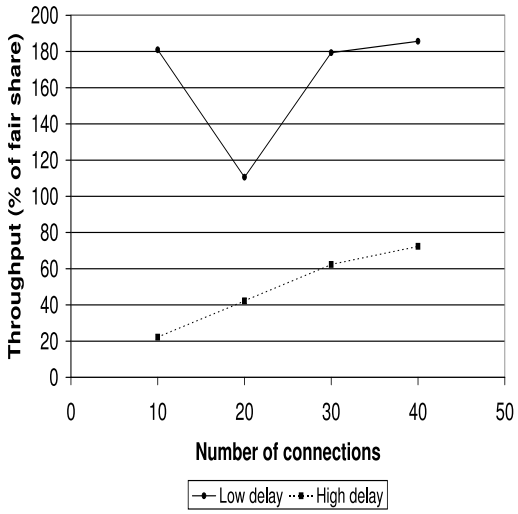
(b) Normalized standard deviation.

Figure 5.12: Throughput for 100 KB transfers with congested terrestrial link. Congested link is downstream of satellite link, which has delay=250 ms in each direction.

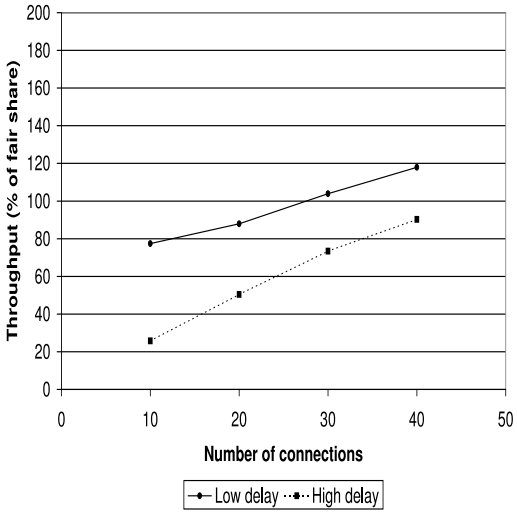
error recovery. During these tests, two simultaneous flood pings were carried out from the client host to its adjacent proxy, alongside the TCP transfers. A flood ping consists of ICMP Echo Request messages sent from one machine to the other 100 times a second or as fast as Echo Response messages are received by the sender, whichever is greater. We used Echo Request messages with 1472 bytes of data, so that the final IP packets containing these messages were exactly 1500 bytes long, i.e. sized equal to the Ethernet MTU. This arrangement simulates congestion on the terrestrial link, making it the bottleneck instead of the satellite link. As expected, the proxies perform better, since any packets dropped due to congestion can be retransmitted locally without traversing the satellite link.

In this case, increasing IW does not help much; it may even be harmful for throughput, though our data are not clear enough to say for certain. It is worth noting that with increased IW the standard deviation of throughput among simultaneous connections becomes very large, and of the order of the mean throughput.

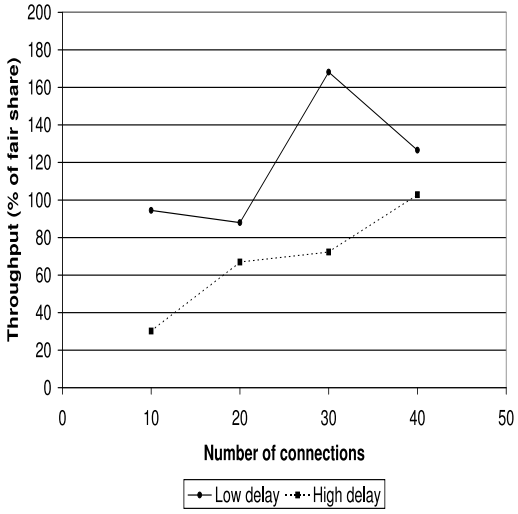
For the experiments in Table 5.13, a third host was used in addition to the setup in the preceding tests. This machine was connected to the server by Ethernet through a Cisco 7000 router, and served as an additional client host. In each case, half the total number of transfers (referred to as set L, for low delay) were to this additional client while the rest (set H, for high delay) were to the client on the other side of the simulated satellite link. Thus the only link common to set L and set H was the Ethernet link connected to the server.



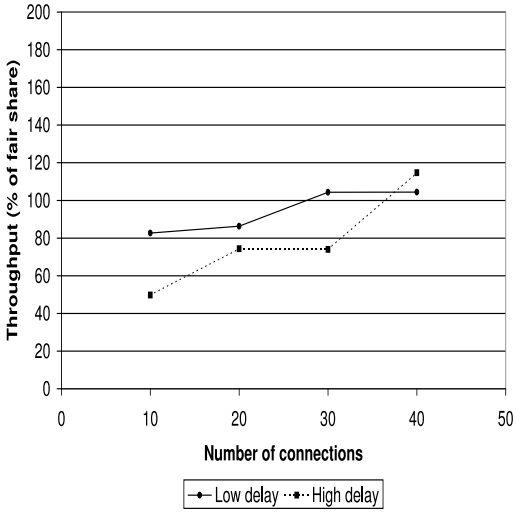
(a) End-to-end TCP.



(b) Proxy, IW=1.

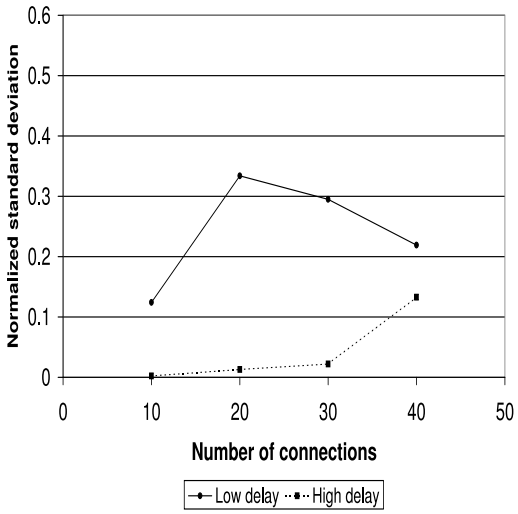


(c) Proxy, IW=4.

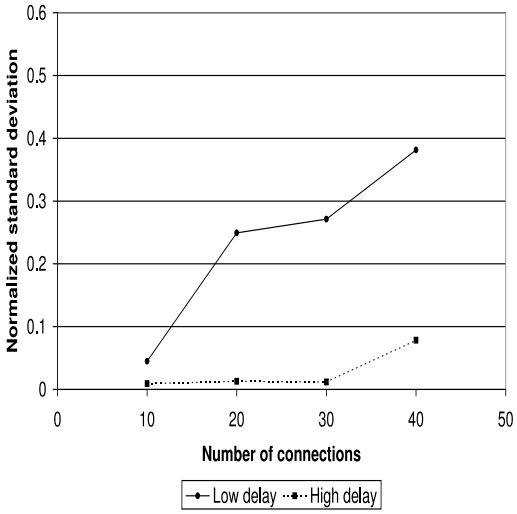


(d) Proxy, IW=16.

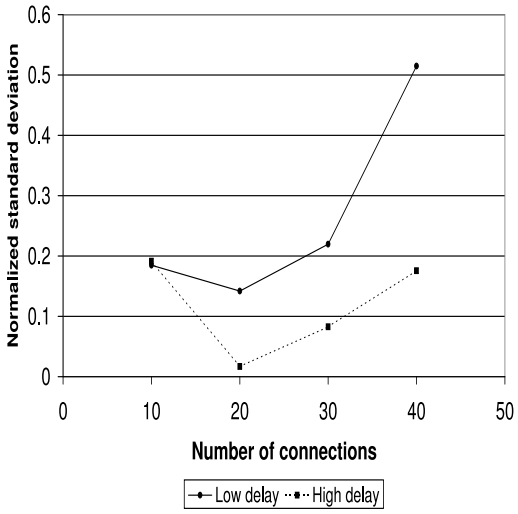
Figure 5.13: Bandwidth sharing among flows with different RTT.



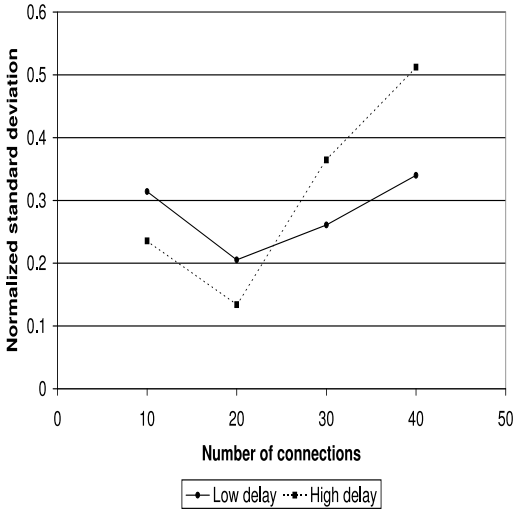
(a) End-to-end TCP.



(b) Proxy, IW=1.



(c) Proxy, IW=4.



(d) Proxy, IW=16.

Figure 5.14: Bandwidth sharing among flows with different RTT.

Total no. of conns.	Set	Mean throughput (Mbps)				Standard deviation (Mbps)			
		End-to-end TCP	Proxy IW=1	Proxy IW=4	Proxy IW=16	End-to-end TCP	Proxy IW=1	Proxy IW=4	Proxy IW=16
10	L	1.8090	0.7738	0.9450	0.8269	0.2248	0.0348	0.1746	0.2598
10	H	0.2215	0.2579	0.3017	0.4974	0.0005	0.0024	0.0577	0.1171
20	L	0.5529	0.4396	0.4397	0.4317	0.1846	0.1096	0.0624	0.0886
20	H	0.2112	0.2521	0.3345	0.3718	0.0027	0.0033	0.0056	0.0498
30	L	0.5979	0.3465	0.5605	0.3479	0.1764	0.0939	0.1230	0.0907
30	H	0.2077	0.2447	0.2409	0.2467	0.0046	0.0029	0.0199	0.0899
40	L	0.4640	0.2947	0.3164	0.2610	0.1017	0.1124	0.1629	0.0887
40	H	0.1809	0.2257	0.2570	0.2868	0.0240	0.0177	0.0451	0.1469

Table 5.13: Comparison of fairness to long-delay flows. Individual transfer size=100 KB. In each case, half the connections (set L) went over a low-delay path, while the others (set H) went over a path with delay 250 ms in each direction.

The results are plotted in Figure 5.13 and Figure 5.14. Due to the much larger round-trip delay experienced by clients in set H, end-to-end TCP is seen to be extremely unfair to them. However the proxies do a much better job of sharing bandwidth between the connections by enabling long-delay connections to compete on equal terms with low-delay wired links. As the initial window is increased, the sharing behavior improves. The remaining unfairness is due to the fact that the proxies use TCP over the satellite link, where the window evolves much slower than over the terrestrial low-delay link. This points to the need for a more efficient protocol over the satellite link. One obvious solution to this problem could be to use TCP with different window increments during slow start and congestion avoidance depending on the round-trip time.

Figure 5.14 shows an interesting trend - the standard deviation for set L is much higher than that for set H. This seems to strengthen the argument that TCP does not share bandwidth equally between identical connections when congestion is present.

5.3 Some Comments

From the above, we see that the value of IW plays a major role in determining throughput. However, it is not possible for a host to know such a value when it sets up the connection. This suggests that it might be useful to have the network participate in flow control by determining a good value for the initial window at connection setup, and informing the host about this value. In any event, it is clear that at least for the transfer sizes tested, the transfer does not last long enough for the window to stabilize or for equilibrium to be reached.

Looking closer at Figure 5.13, a disturbing fact becomes apparent. The throughput achieved by set H in the proxy case with IW=1 is only slightly higher than that achieved under end-to-end TCP. However, the throughput for set L is much lower. This suggests that with the proxies enabled, high-RTT connections become unnecessarily aggressive. Further investigation shows that the use of proxies for set H causes the window for these connections to grow as rapidly as for set L, and so all the data is transferred very quickly to the first proxy on the path, where it must wait for transmission when the congestion window on the

satellite link grows sufficiently large. This behavior unnecessarily reduces the throughput of set L, and uses up large amounts of memory on the proxy.

The above observation motivates us to consider methods for flow control at the proxies, so that they are never unnecessarily aggressive, and can keep their memory requirements low without sacrificing throughput. Such issues are the subject of the next chapter.

Chapter 6

Discussion: Flow Control Methods

6.1 Motivation

Every connection splitting proxy maintains, at least notionally, two sets of buffers for each direction of a split TCP connection - the receive buffer, into which data is received from the upstream data source, and the transmit buffer, which is used to hold data waiting to be sent downstream or data that has been sent downstream but not yet acknowledged. Often these are physically the same area of memory, but even then the proxy must keep track of them separately in order to maintain the splitting correctly.

The simplest way to achieve flow control, then, is to use a "back-pressure" mechanism. In its simplest form, this method consists of placing a fixed constraint on the size of the transmit buffer of each connection segment at a proxy, so that when this limit is exceeded a backlog accumulates in the receive buffers of the upstream connection segment and slows down the transfer on that portion accordingly. In this way congestion indications finally propagate back to

the sending host. This simple method is used by nearly all current implementations.

Our proxy implementation uses a slightly improved version of this method. The fixed constraint on the transmit buffer is replaced with a variable constraint, equal to the current value of the TCP send window on that connection segment. This speeds up the propagation of congestion information to the sender, as the receive buffers begin to get backlogged as soon as the inflow rate into the proxy exceeds the outflow rate from it. This modification also reduces memory requirements at the proxy. The storage required for transmit buffers at the proxy becomes equal to the sum of the send windows of all active connections, which is directly proportional to the sum of bandwidth-RTT products of the outgoing network paths at the proxy. In the simple scheme, on the other hand, the memory required for transmit buffers is proportional to the number of active connections.

This improved scheme still has some of the same drawbacks as the simple scheme, in that the total memory required for receive buffers is still directly proportional to the number of active split connections. We now explore methods of reducing this requirement, preferably to a quantity independent of the number of active connections, so that the proxy can scale easily to handle large numbers of split connections. In the rest of this chapter, we will consider only unidirectional transfers for simplicity. However, the methods derived can be easily extended to full-duplex TCP connections by using the same method in each direction independently.

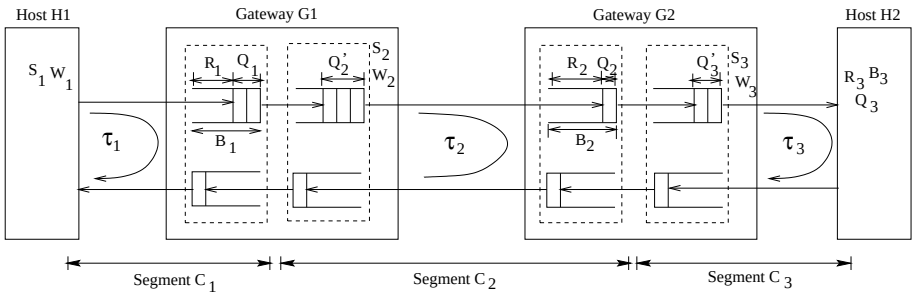


Figure 6.1: Back-pressure in split TCP connections. This diagram shows a unidirectional data transfer from H1 to H2 in the topology of Figure 3.1. Queues belonging to the same TCP connection are enclosed by dotted lines.

6.2 Problem Constraints

In Figure 6.1, for $i \in \{1, 2, 3\}$, let B_i represent the receive buffer (i.e. maximum possible offered window) on segment C_i, R_i the available receive buffer (i.e. current offered window) on segment C_i, Q_i the amount of data queued after acknowledgment on connection C_i, W_i the congestion window on segment C_i, S_i the send window on segment C_i (corresponding to SND.WND in [3]), Q'_i the amount of data sent or queued for transmission on segment C_i, and τ_i the round-trip delay on segment C_i. Note that these quantities are all functions of time, except for τ_i, which we assume constant.

So using this terminology, we can say that our proxy achieves flow control by keeping Q'_i as large as possible under the constraint $Q'_i \leq S_i$, i.e. by sending as much data on C_i as possible with the current window sizes, and buffering the rest in the receive buffer on C_{i-1}. The simple back-pressure scheme corresponds to keeping $Q'_i \leq M_i$, where M_i is some constant. The problem with both schemes is that they allow the queues at the proxy to grow very large (up to $Q_i = B_i$, where

B_i is a constant chosen at connection setup time) before the sender finally stops sending data and waits for the proxy to clear its backlog. To explore the issues involved in changing this behavior, let us try to design a more aggressive flow control scheme to reduce the memory required by actively regulating the flow on the upstream connection segment.

We look at some of the desirable characteristics such a scheme should have. Most importantly, we would like a scheme that uses only standard TCP mechanisms, as every network path of interest to us will include at least one segment between a proxy and a host, where we would like to use flow control. Such a scheme would also maintain the ability of the proxy to be deployed in any of the topologies in Figure 3.3 without modification.

While designing our new scheme we will assume that congestion patterns on a TCP connection vary on time scales that are at least of the order of the round-trip time of the connection and are short compared to its total lifetime. The first of these assumptions is usually a reasonable one, due to the dynamics of TCP congestion control. The second is more debatable, as it requires long-lived TCP connections, but it is necessary otherwise we will be in a position of trying to control transients that die out before we can observe them.

An obvious approach is to try and design a scheme that equates the transfer rates into and out of each proxy at all times. Under ideal conditions, such a scheme would make memory requirements for a proxy equal to the sum of the bandwidth-delay products of its outgoing paths, as memory would only be

required for transmit buffers and receive buffers could always be empty or nearly so. Thus memory requirements would be reduced while maintaining uninterrupted end-to-end transfer of data at the optimal rate.

For example, in Figure 6.1, consider proxy G1 and assume the bottleneck is downstream of it. Then S_1/τ_1 represents the rate at which H1 is sending data on C_1, and S_2/τ_2 represents the rate at which G1 is sending data on C_2. If we make these equal, very little buffering will be required at G1. If the bottleneck is upstream of G1, very little needs to be buffered at G1 anyway - data can be sent on C_2 as soon as it arrives on C_1. The major challenges in implementing such a scheme using standard TCP mechanisms are

- There is a delay in the feedback loop. Whenever G1 takes any action to try to make H1 change the rate of transmission on C_1, there is a delay of τ_1 before the change can take effect, due to the propagation delay between G1 and H1. However, in this time, S_2 might have changed.

- The receiver may not have an estimate of the round trip delay on a connection. This is the case for unidirectional transfers. Thus G1 may not know τ_1.

- In TCP, no direct information about the sender's congestion window is available to the receiver. Also, the congestion window keeps evolving with time, so it is hard for the receiver to track.

- A TCP receiver does not have a simple mechanism to tell the sender to increase the sending rate - it can only tell the sender to decrease its rate by reducing the offered window or by dropping a packet.

Thus we need a scheme that does not require any knowledge of W_1 or τ_1. We will now look at a simple algorithm that satisfies these conditions.

6.3 A Simple Algorithm

We observe that if τ_1 be known, it is easy to design a good flow control scheme for G1. Essentially, at any time t, the scheme would need to predict the amount of data that will be sent on C_2 in the time interval $[t+\tau_1, t+2\tau_1)$, and set the receive window on C_1 equal to this value. An upper bound on the amount of data sent in $[t+\tau_1, t+2\tau_1)$ can be calculated easily given τ_1 and the current state of C_2, since the evolution of the TCP congestion window in a given interval can be bounded.

In our case we do not know τ_1, but we can make two simple observations about the behavior of the quantities in Figure 6.1:

1. In any period $[t, t+\tau_1)$, the amount of data received by G1 on C_1 cannot exceed $R_1(t)$, unless the value of B_1 has been reduced in $(t-\tau_1, t]$. Thus the total amount of data available at G1 in $[t, t+\tau_1)$ is at most $B_1(t)$, and G1 cannot have sent more than $(B_1(t) - Q_1(t+\tau_1))$ on C_2 in this period.

2. When the inflow and outflow rates at G1 are perfectly balanced,

$$\frac{B_1}{\tau_1} = \frac{S_2}{\tau_2} \tag{6.1}$$

Let us construct a flow control scheme based on these observations. We assume that TCP is used on all segments, and that our improved back-pressure scheme is in operation, i.e. $Q'_i \leq S_i$ for all i at all times, and $Q_{i-1} > 0 \Rightarrow Q'_i = S_i$.

There are four possible cases at G1:

1. $Q_1 = 0$ and $Q'_2 = S_2$, i.e. there is no queueing at the proxy and the downstream link is fully utilized.

2. $Q_1 > 0$ and $S_2 < W_2$, i.e. there is queueing at G1 and G1 is limited by G2's offered window.

3. $Q_1 > 0$ and $S_2 = W_2$. i.e. there is queueing at G1 and G1 is limited by its congestion window.

4. $Q_1 = 0$ and $Q'_2 < S_2$, i.e. there is a bottleneck upstream.

We now consider these cases one by one. In each case we assume that the value of B_1 has been held constant for at least $2\tau_1$.

Case 1: $Q_1 = 0$ and $Q'_2 = S_2$. In this case the system is working perfectly, and there is no need to change anything.

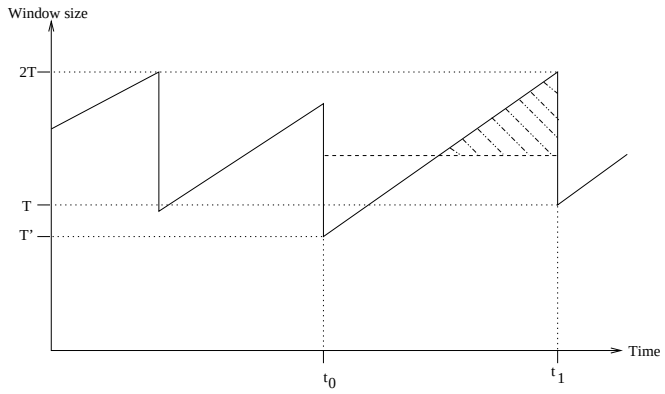


Figure 6.2: Example evolution of congestion window.

Case 2: $Q_1 > 0$ and $S_2 < W_2$. During the last τ_1, G1 can have sent at most $(B_1(t) - Q_1(t))$ on C_2. Since the outflow rate is limited to a constant, we can safely reduce B_1 to a value b such that

$$b = B_1(t) - Q_1(t) = R_1(t)$$

without reducing the throughput of the connection. Also, from (6.1), we have

$$\tau_1 \leq \frac{R_1(t)}{S_2(t)}\tau_2$$

Case 3: $Q_1 > 0$ and $S_2 = W_2$. We look at the dynamics of W_2. In steady state, it has cycles as shown in Figure 6.2. Consider the interval $[t_0, t_1)$. The average rate in this interval is as shown by the flat dotted line. One possible approach is to use B_1 to limit H1 to this rate. However, to maintain $Q_2' = S_2$, G1 will need to buffer an amount of data equivalent to the shaded area on the graph, i.e. a second-order polynomial of the rate. Also, B_1 must be continuously adjusted so that R_1 stays constant. Therefore this approach is not practical.

If the value of T were somehow known to us beforehand, then we could allocate B_1 large enough so that G1 would have enough data to fill its send window at all times. Then back-pressure would slow down H1 whenever G1 had a backlog in its queue, so that G1 would not have to buffer too much data. In other words, we could assign $B_1 = b$ such that

$$\frac{b}{\tau_1} = \frac{2T}{\tau_2} \tag{6.2}$$

Since the next value of T is not known to us beforehand, the easiest solution is to use the previous value of the slow start threshold, say T', as an estimate. Given some assumptions on system behavior, we could even derive a bound for the error of this estimate. However, in practice, we will have problems whenever the window grows larger than $2T'$, as we will not be able to maintain full link utilization. An alternative estimation technique would be to use a weighted average of past threshold values, though it has the disadvantage that we must keep track of an extra state variable for each connection.

We know that G1 has sent $W_2(t)$ on C_2 in the last τ_2. If it is in congestion avoidance, its window is increasing linearly so in the last τ_1 the amount of data sent by G1 is bounded by

$$\bar{W}_2 \geq \left[W_2(t) - \frac{l\tau_1}{2\tau_2} \right] \frac{\tau_1}{\tau_2}$$

where l is the segment size for the connection. Now, we also know that G1 must have sent at most $R_1(t)$ in the last τ_1, so

$$R_1(t) \geq \left[W_2(t) - \frac{l\tau_1}{2\tau_2} \right] \frac{\tau_1}{\tau_2}$$

Thus we get a quadratic expression in τ_1 on the right-hand side, which would require a significant amount of processing to solve. However, if we already know an upper bound for τ_1, we can substitute it into the negative term inside the parentheses and so get a linear inequality from which a new upper bound for τ_1 can be found. This new bound can then be substituted into (6.2) to give an upper bound on b.

Case 4: $Q_1 = 0$ and $Q_2' < S_2$ G1 cannot know if this is due to B_1 being set too small, or if it is due to some other bottleneck in C_1. If it is due to B_1 being too small, we have the inequality

$$\tau_1 \leq \frac{B_1(t)}{Q_2'(t)} \tau_2$$

This gives us an upper bound on τ_1. Using (6.1), we set buffer size to b where

$$b = B_1(t) \frac{S_2(t)}{Q_2'(t)}$$

6.4 Implementing the Scheme

Using the above ideas we can easily formulate an algorithm for performing flow control. As before we use G1 and the associated quantities for clarity.

In order to formulate an algorithm based on the above observations we must specify how often B_1 should be updated. We note that once the value of B_1 is changed, an interval of $2\tau_1$ is required for the queue lengths at G1 to stabilize. Returning to the equations in the four cases above we find that in each case we also get an upper bound on τ_1 as a result of the calculation.

Finally, we note that if a bottleneck is present upstream of G1, repeated application of the inequality in Case 4 may cause B_1 to be increased indefinitely. Therefore, if increasing B_1 once does not cause Q'_2 to increase, we deduce that the bottleneck is upstream.

Thus the implementation of the algorithm is very simple. We keep four state variables - τ holds the time when the algorithm was last invoked, q_1 is the value of Q_1 at τ, t_1 is our current estimate of τ_1, and q_2 holds the last value of Q'_2. The algorithm consists of the following steps:

1. Initialization. At connection setup, we can either start with an estimate of τ_1 and set B_1 accordingly or we can start with B_1 set to its maximum value, and with the initial estimate of τ_1 set to the Maximum Segment Lifetime of the Internet. Set all the other three state variables to zero.

2. Do nothing until Q_1 becomes greater than zero. When it does, go to the next step.

3. If $q_2 = 0$ (bottleneck is upstream), or $Q'_2 = 0$ (connection is idle), or $R_1 = 0$ (last reduction in B_1 has still not stabilized), set τ to the current time and

skip to step 5. Otherwise go to step 4.

4. Adjust B_1 according to the appropriate case. Then calculate the bound on τ_1 and assign it to t_1. Set τ to the current time, and q_1 to the current value of Q_1. If $q_1 = 0$, set q_2 to the current value of Q'_2, otherwise set it to zero.

5. Whenever a packet arrives on C_1, check if the current time exceeds $(\tau + 2t_1)$. If so, go back to step 3.

This scheme keeps the queues in the receive buffer at G1 much smaller than the previous back-pressure scheme. Also, since at each step we only used upper bounds on b to update B_1, we are assured that we will never have a situation where we reduce the offered window too much.

This scheme was added to our proxy implementation. It reduced memory requirements in many situations, but its failings were also interesting and they give some insight into the construction of TCPPEPs in general.

6.5 Proxies and Scalability

In designing the above scheme, we have overlooked three key factors. First, the memory requirements of a connection splitting proxy can never be truly upper bounded if the proxy is to maintain optimal link utilization. This is because the proxy cannot discard data it has acknowledged until it has successfully transmitted it and received an acknowledgment. There can be many cases, such

as a failure on the downstream host, in which the proxy would have to buffer data for an unusually long time.

A second problem is that any such flow control scheme can only work well when the upstream RTT is much less than the downstream RTT. For instance in our example above, it works when $\tau_1 < \tau_2$, but not when the reverse is true. The reason for this is that the proxy can only change the behavior of its upstream segment over intervals of the order of τ_1, whereas the downstream segment changes over intervals of the order of τ_2. In a sense, having $\tau_1 > \tau_2$ is like trying to control a system with a controller that can only vary at a rate much slower than the system itself. In our test system, the satellite link had an RTT of 500 ms while the terrestrial Ethernet had an RTT of about 2 ms, and this problem was easily observed. Whenever the satellite link was the bottleneck on a connection, our flow control scheme worked well, as there was very little to do on the proxy downstream of the satellite link. However when the bottleneck was downstream of the satellite link (as in the tests where we simulated terrestrial congestion) very large queues built up at the proxy downstream of the satellite link.

A third problem is purely practical - as observed in Chapter 5, typical TCP flows on the Internet are relatively short and do not last for a large enough number of round trips to allow schemes such as this to reach steady state. Therefore we often see fairly large queues even with our scheme in place, because data transfer often completes while the scheme is still in a transient stage.

It bears repetition that none of the above drawbacks is specific to our scheme

- they are general problems that apply to any connection-splitting proxy implementation. Therefore it is now relevant to take a closer look at the limitations of this architecture.

It is clear from the results in Chapter 5 that both localized flow control and localized error recovery are useful techniques in satellite networks. In nearly all the experiments with multiple TCP connections, the most important factor in ensuring maximum throughput was the choice of a good initial window size. Since a host is not expected to know the actual path a TCP connection will traverse in the network, it follows that it is useful for the network to be able to give feedback to hosts regarding a good initial window. This could be done, for example, during the initial three-way handshake to set up a TCP connection.

Localized error recovery is also seen to help, but in a proper connection-splitting implementation it is also the reason for high, and in fact unbounded, memory requirements. However, error recovery and flow control are so inextricably linked in TCP that it is not possible for the network to participate in flow control without also taking over at least some error recovery functions. Therefore it is an interesting topic for future research to explore ways to delink error recovery from flow control in transport protocols such as TCP while maintaining other desirable characteristics of TCP. It seems that a proxy that did semi-reliable local error recovery, similar to a Snoop agent, and also participated in flow control might significantly improve throughput while keeping memory requirements bounded.

Chapter 7

Conclusions and Future Work

In this thesis we demonstrated the feasibility of implementing a TCP Performance Enhancing Proxy in any part of the Internet by showing an implementation of such a proxy for a geostationary satellite link in a general network topology. The results obtained from testing this proxy show large improvements in the data transfer rates achieved by users under a wide range of conditions, and also point out many interesting issues for future work.

It is shown that TCP has many drawbacks for geostationary satellite links, and that using proxies to allow different methods of flow control on the satellite link would have significant benefits. We also see that it is useful to separate the functions of error recovery and flow control in transport protocols, and to allow the network to handle some part of this functionality. The search for methods to achieve this separation, and for good localized flow control schemes is a topic for future investigation.

Recent months have seen increasing deployment of next-generation Internet protocols such as IPSEC and IPv6 in the Internet. These protocols change some

of the characteristics of Internet traffic, and also the information available to network nodes about end-to-end traffic. The impact of these changes on proxy architectures such as the one described here is also an important aspect for future study.

BIBLIOGRAPHY

[1] B. Braden, ed. Requirements for Internet hosts – communication layers. RFC 1122, October 1989.

[2] J. Postel ,ed. Internet Protocol – protocol specification. RFC 791, September 1981.

[3] J. Postel, ed. Transmission Control Protocol – protocol specification. RFC 793, September 1981.

[4] J. Postel, ed. User Datagram Protocol. RFC 768, September 1980.

[5] M. Allman, V. Paxson, and W. Stevens. TCP congestion control. RFC 2581, April 1999.

[6] T.V. Lakshman and U. Madhow. The performance of TCP/IP for networks with high bandwidth-delay products and random loss. *IEEE/ACM Transactions on Networking*, June 1997.

[7] M. Allman, D. Glover, and L. Sanchez. Enhancing TCP over satellite channels using standard mechanisms. RFC 2488, January 1999.

[8] M. Allman, editor. Ongoing TCP research related to satellites. Internet Draft draft-ietf-tcpsat-res-issues-06.txt (work in progress), March 1999.

[9] V. Jacobson, R. Braden, and D. Borman. TCP extensions for high performance. RFC 1323, May 1992.

[10] M. Mathis, J. Mahdavi, S. Floyd, and A. Romanow. TCP selective acknowledgment options. RFC 2018, October 1996.

[11] Mark Allman, Sally Floyd, and Craig Partridge. Increasing TCP's initial window. RFC 2414, September 1998.

[12] A. D. Falk. A system design for a hybrid network data communications terminal using asymmetric TCP/IP to support internet applications. Master's thesis, Institute for Systems Research, University of Maryland, 1994.

[13] V. Arora, N. Suphasindhu, J.S. Baras, and D. Dillon. Asymmetric internet access over satellite-terrestrial networks. In *Proceedings of the AIAA: 16th International Communications Satellite Systems Conference and Exhibit, Part 1*, pages 476–482, Washington, D.C., February 1996.

[14] H. Balakrishnan, S. Seshan, and R. Katz. Improving reliable transport and handoff performance in cellular wireless networks. *ACM Wireless Networks*, 1(4), December 1995.

[15] Packeteer - intelligent bandwidth management. http://www.packeteer.com.

[16] S.M. Bellovin. Security problems in the TCP/IP protocol suite. *Computer Communication Review*, 19(2):32–48, April 1989.

[17] M. Mathis and J. Mahdavi. Forward Acknowledgment: Refining TCP congestion control. In *Proceedings of SIGCOMM '96*, Stanford, CA, August 1996.

[18] Hari Balakrishnan, Venkat Padmanabhan, Srinivasan Seshan, and Randy H. Katz. A comparison of mechanisms for improving TCP performance over wireless links. *IEEE/ACM Transactions on Networking*, December 1997.

[19] DBS: A TCP benchmark tool. http://shika.aist-nara.ac.jp/member/yukio-m/dbs/.