

On Optimizing High-Latency Asymmetric Satellite TCP/IP

Charl Coetzee

Thesis presented in partial fulfillment of the requirements for the degree of
Master of Science, Engineering (M.Sc.Eng.), Electronics,
in the Faculty of Engineering
at the University of Stellenbosch



Thesis Supervisor: Prof. J.G. Lourens

October 1999

Declaration

I, the undersigned, hereby declare that the work contained in this thesis is my own original work and has not previously, in its entirety or in part, been submitted at any university for a degree.

Abstract

The Transmission Control Protocol (TCP), part of the TCP/IP Internet Protocol Suite, is responsible for the reliable unicast delivery of popular Internet services such as web browsing, e-mail, and file transfer.

Since satellite communication offers high-bandwidth links to geographically diverse areas, it becomes attractive to use satellite transmission (together with e.g. a remote dial-up modem as return path), to establish Internet services to a remote location. An asymmetric network is thus created.

For satellite transmission to be viable in this context, it must be able to deliver data much more quickly than would be possible via the dial-up modem. Unfortunately, it has been found in practice that TCP performs suboptimally in such environments, and that a single TCP sessions can only fill a fraction of the bandwidth available to it.

In fact, a single standard TCP connection will only be able to obtain around 53 kB/s throughput via satellite (roughly 8 times faster than dial-up modem), even if it has a satellite bandwidth of 54 Mbps all to itself. Other problems also exist, such as apparent unresponsiveness of connections.

This thesis studies these problems, and their possible solutions. It concentrates on examining various TCP enhancements in a network emulator, and shows how TCP can be used more efficiently over the asymmetric satellite channel.

Opsomming

Die 'Transmission Control Protocol' (TCP) uit die TCP/IP Internet Protokol Suite is verantwoordelik vir betroubare enkelbestemming data-aflowering. Dit word gebruik vir verskeie populêre Internet dienste, insluitend web snuffeling, e-pos en lêeroordrag.

Aangesien satelliet-telekommunikasie breëband skakels na geografies diverse areas bied, word dit aanloklik om sulke kommunikasie (te same met byvoorbeeld 'n modem as terugpad), te gebruik om Internet dienste aan afgeleë areas te voorsien. Sodoende kom 'n asimmetriese netwerk tot stand.

Die satellietkanaal moet egter dan die data heelwat vinniger kan lewer as wat via modem moontlik sou wees, andersins is so 'n stelsel nie lewensvatbaar nie. Ongelukkig wys die praktyk dat TCP suboptimaal funksioneer in sulke omgewings, en dat 'n enkele TCP verbinding net 'n fraksie van die beskikbare bandwydte kan vul.

Om die waarheid te sê, 'n enkele standaard TCP verbinding kan alleenlik ongeveer 53 kB/s via satelliet behaal (ongeveer 8 keer vinniger as per modem), selfs al sou die verbinding alleenreg tot 'n satellietkakel van 54 Mbps hê. Ander probleme kom ook voor, byvoorbeeld die stadige reaksie van TCP satelliet-sessies.

Die tesis bestudeer hierdie probleme, en moontlike oplossings word voorgestel. Daar word gekonsentreer op die ondersoek van verskeie TCP verbeteringe in 'n netwerk emuleerder, en gewys hoe TCP meer effektief oor die asimmetriese satellietkanaal gebruik kan word.

Acknowledgements

(of the non-TCP variety)

I would like to extend my heart-felt thanks to the international free software community: their efforts over the years have provided the tools which made possible not only this research, but some of TCP/IP itself.

Thanks to Shawn Ostermann and Tim Shepard for *tcptrace* and *xplot*, analysis tools which have proved invaluable. Thanks to Neal Cardwell for sending me Linux TCP Vegas and to Nihal Samaraweera for the original SIMULA ACK compaction code.

To Mark Allman, Sally Floyd, Vern Paxson, Craig Partridge and all at LBL, NASA LeRC, Ohio University, et al: many thanks for the high-quality research papers archived on-line. This has made literature study a true pleasure.

I would also like to thank Charl Botha, Gerhard Esterhuizen, Delmar Olivier, Garth Bond and Hans Grobler for numerous beneficial discussions. Especially to Hans, thanks for those hours spent poring with me over the Linux networking source code.

Thanks to Professor David Weber and the DSP Lab for providing a stimulating working environment with Linux expertise. Heart-felt thanks to my family and friends, for their support and for putting up with me and my strange hours.

I would like to extend my deepest gratitude to Professor Johan Lourens, my advisor, for his support, engineering wisdom and anecdotes. This has been of great benefit to this thesis, but perhaps more importantly, it has also been a wonderful, life-enriching experience.

List of Abbreviations

ACK	Acknowledgement
API	Application Programming Interface
ARP	Address Resolution Protocol
BCP	Best Current Practice
BSD	Berkeley Standard Distribution
<i>cwnd</i>	Congestion Window
CPU	Central Processing Unit
CSLIP	Compressed SLIP
DSL	Digital Subscriber Loop
DVB-S	Digital Video Broadcast - Satellite
FAK	Forward Acknowledgement
FEC	Forward Error Correction
FTP	File Transfer Protocol
HTTP	Hypertext Transfer Protocol
IETF	Internet Engineering Task Force
IMAP	Internet Message Access Protocol
IP	Internet Protocol
IRC	Internet Relay Chat
ISP	Internet Service Provider
LAN	Local Area Network

continued on next page ...

List of Abbreviations, continued from previous page

LBC	Limited Byte Counting
MSS	Maximum Segment Size
MTU	Maximum Transmission Unit
NNTP	Network News Transfer Protocol
PAWS	Protection Against Wrapped Sequence numbers
POP	Post-Office Protocol
PPP	Point-to-Point Protocol
<i>recvwin</i>	Receive Window
RED	Random Early Detection
RFC	Request For Comment (an Internet standards document)
RTO	Retransmission TimeOut
RTT	Round-Trip time
SACK	Selective Acknowledgement
SLIP	Serial Line Internet Protocol
SMTP	Simple Mail Transfer Protocol
<i>ssthresh</i>	Slow Start Threshold Size
TCP	Transmission Control Protocol
TOS	Type of Service
UDP	User Datagram Protocol
VJ	Van Jacobson

Contents

1	Introduction	1
1.1	TCP Weaknesses in High-Latency Asymmetric Networks	2
1.2	Spirit of the Research	3
1.3	Criteria for Evaluating Improvements	4
1.4	Approach to the Work	4
1.5	Working Method	6
1.6	Comments on Mathematical Modeling	6
1.7	Overview of the Literature	6
1.8	Structure of the Thesis	7
1.9	Intended Audience	8
1.10	Terminology	8
2	Relevant TCP Characteristics	9
2.1	TCP/IP Implementations	9
2.2	The TCP Packet and Header	10
2.3	TCP Sequence Numbers	10
2.4	TCP Acknowledgements	11

CONTENTS	viii
2.5 Selective ACKs (SACKs)	12
2.6 The Three-Way Handshake	12
2.7 The Maximum Segment Size (MSS)	13
2.8 Window-based Flow Control	14
2.8.1 Windows	14
2.8.2 ACK Self-Clocking	15
2.8.3 ACK Compression	15
2.9 Congestion Control	17
2.9.1 Slow Start	17
2.9.2 Congestion Avoidance	17
2.9.3 Timeouts	18
2.9.4 Fast Retransmit and Fast Recovery	18
2.9.5 SACK and FACK	19
2.10 The Bandwidth-Delay Product	19
2.11 Asymmetry	20
2.12 Conclusions	20
3 Satellite Network Architecture Design	21
3.1 Providing a Surrogate Sender	21
3.1.1 Spoofing	22
3.1.2 Proxy Server	22
3.2 Architecture Design	24
3.3 Issues of Multihoming and Routing	26
3.3.1 Single IP Address	26

<i>CONTENTS</i>	ix
3.3.2 Multiple IP Addresses	26
3.4 Assumptions	27
3.4.1 Packet Corruption and Error Rates	28
3.4.2 Congestion Control	28
3.4.3 Proxy Retrieval	29
3.4.4 Modem Settings	29
3.4.5 Latency	29
3.5 Conclusions	30
4 The Experimental Platform: Network Emulation	32
4.1 The Platform	33
4.2 Latency	33
4.2.1 Transmission Delay	34
4.2.2 Queuing Delay	34
4.2.3 Propagation Delay	35
4.2.4 Comments	35
4.3 The Emulator	36
4.4 Testing the Emulator	37
4.5 Measurements	39
4.6 Conclusions	40
5 Steady-State Performance	41
5.1 Window Size	41
5.1.1 Receiver Window Limitation	41

CONTENTS	x
5.1.2 Increasing the Window Size	42
5.2 Effects of Asymmetry	45
5.2.1 ACK Congestion	45
5.2.2 Undersized MSS	47
5.3 Improving ACK Congestion	47
5.3.1 ACK Suppression	47
5.3.2 Header Compression	48
5.3.3 ACK Compaction	50
5.3.4 Comments on Timestamps and SACKs	50
5.4 Obtaining the Maximum MSS	51
5.4.1 Setting Serial Interface MTU = Satellite Interface MTU	52
5.4.2 Use a Tunneling Network Device	52
5.4.3 Standardize True MSS Negotiation	53
5.4.4 Modify Default MSS in Absence of MSS Announcement	53
5.5 Measurements	53
5.5.1 Validations	53
5.5.2 Throughput Improvement	54
5.6 Conclusions	56
6 Dynamics: Packet Loss	58
6.1 The Experimental System	59
6.2 Packet Loss on the Reverse Link	59
6.2.1 Packet Loss on Dial-Up Link	59
6.2.2 Packet Loss on rest of Return Network	61

CONTENTS	xi
6.3 Long Outage on Forward Channel	61
6.4 Single and Multiple Dropped Packets	62
6.4.1 Exploding Serial Queues	62
6.4.2 Limiting Queue Length	64
6.4.3 Allowing Queue to Grow	67
6.4.4 Comments	71
6.5 Conclusions	73
7 Dynamics: TCP Slow Start	75
7.1 Queue Management	75
7.1.1 The Problem	76
7.1.2 Solutions	77
7.1.3 Experiments	77
7.1.4 Vegas Comment	79
7.2 Slow Start Speedup	80
7.2.1 Modified <i>cwnd</i> Increase Algorithms	82
7.2.2 Rate-Based Pacing	83
7.2.3 Vegas Comment	83
7.3 Conclusions	84
8 Example Designs	86
8.1 Baseline	87
8.2 Standard Return Link with Per-Client Throttling	87
8.3 Standard Return Link without Throttling	89

CONTENTS	xii
8.4 Header Compression with Shared ISP	90
8.5 Header Compression with Dedicated Dial-Up	93
8.6 ACK Compaction	95
8.7 A Custom Protocol	95
8.8 Conclusions	98
9 Conclusions	99
9.1 Network Emulation	99
9.2 Network Architecture	100
9.3 Header Compression	100
9.4 TCP Congestion Notification	100
9.5 Future TCP	101
9.6 Custom Paced Protocol vs. ACK Compaction	102
9.7 Final Conclusions and Recommendations	103
Bibliography	105
A ACK Compaction	110
A.1 Traditional ACK Compaction	110
A.1.1 Questions	111
A.1.2 Other Observations	111
A.2 Revised ACK Compaction	112
A.2.1 Timing Considerations	113
A.2.2 Observations	114
A.3 Conclusions	115

CONTENTS

xiii

B Reading TCP Time Sequence Graphs

116

List of Figures

1.1	The high-latency, asymmetric network configuration studied	2
2.1	The TCP three-way handshake	12
2.2	The TCP sliding window flow-control	14
2.3	TCP self-clocking in general	16
3.1	A network design for high-latency asymmetric TCP delivery	24
4.1	The emulation test-platform	36
4.2	Flood pinging the emulator	38
5.1	Example TCP throughput vs RTT	55
5.2	Example improved TCP session	56
6.1	Effect of outage on VJ header-compressed dial-up link	60
6.2	Overwhelmed queue leading to RTO	65
6.3	Overwhelmed queue leading to burstiness without RTO	67
6.4	Close-up of transmission burstiness	68
6.5	Overwhelmed queue leading to burstiness and RTO	69
6.6	Close-up of burstiness leading to RTO	69

<i>LIST OF FIGURES</i>	xv
6.7 Allowing the serial queue to grow without RTO	70
6.8 Burstiness after fast recovery	71
7.1 An extended slow start fills serial queue	78
7.2 A TCP Vegas transmission	79
8.1 Example design using standard return link and throttling	88
8.2 Example design with standard return link, no throttling	91
8.3 Example design: header compression, throttling and shared dial-up	92
8.4 Example design: header compression with dedicated dial-up	94
8.5 Example design using a custom paced protocol	96
A.1 Preliminary emulation of revised ACK compaction	112

List of Tables

1.1	Weaknesses of TCP	3
5.1	Typical example asymmetric throughput limits	54
5.2	Statistics for improved steady-state throughput experiment	56
7.1	Slow start times (sec) for $recvwin = 250$ kB and $MSS = 1460$ B . . .	82

Chapter 1

Introduction

The Transmission Control Protocol (TCP) [1], part of the TCP/IP Internet protocol suite, lies at the heart of reliable unicast data delivery over the Internet. As such, it is used for web browsing (HTTP), file transfer (FTP), e-mail transfer (SMTP, POP3, IMAP) and usenet news transfer (NNTP), among others.

Most users receive much more data than they originate, and in recognition of this many modern networks are designed to be asymmetric — this usually leads to lower cost.

Delivery of space TCP communications is becoming an increasingly active research area. TCP is being viewed as a mechanism via which earth stations may reliably communicate with space probes, or with computers on the moon and remote planets.

A subset of this is delivery of Internet services via satellite. Typically, a high-speed satellite downlink is used to transfer data to the user, while a low-speed dial-up phone line will transfer the TCP requests and acknowledgements from the client back to the server. This also enables Internet connectivity to the remotest of rural areas, where no terrestrial infrastructure may exist, but cellular or satellite telephony is accessible.

TCP is a one-size-fits-all solution, and while it functions over any network architecture, it does not always function efficiently. It is known that the latency involved in satellite connections can drastically degrade TCP performance, as can network asymmetry. The combination of *both* high latency and asymmetry makes for an even more explosive situation.

This thesis studies performance improvements to TCP in the context of delivery via geostationary satellite, with acknowledgements returning via a 9600 baud serial line (see figure 1.1). The 9600 baud speed was chosen as being the lowest common denominator, a communication speed which is almost always available. TCP improvements which perform adequately in a 9600 baud environment, should have no problems at higher speeds.

The thesis is structured in such a way to attempt to make the results relevant to other cases of asymmetric / space TCP communications.

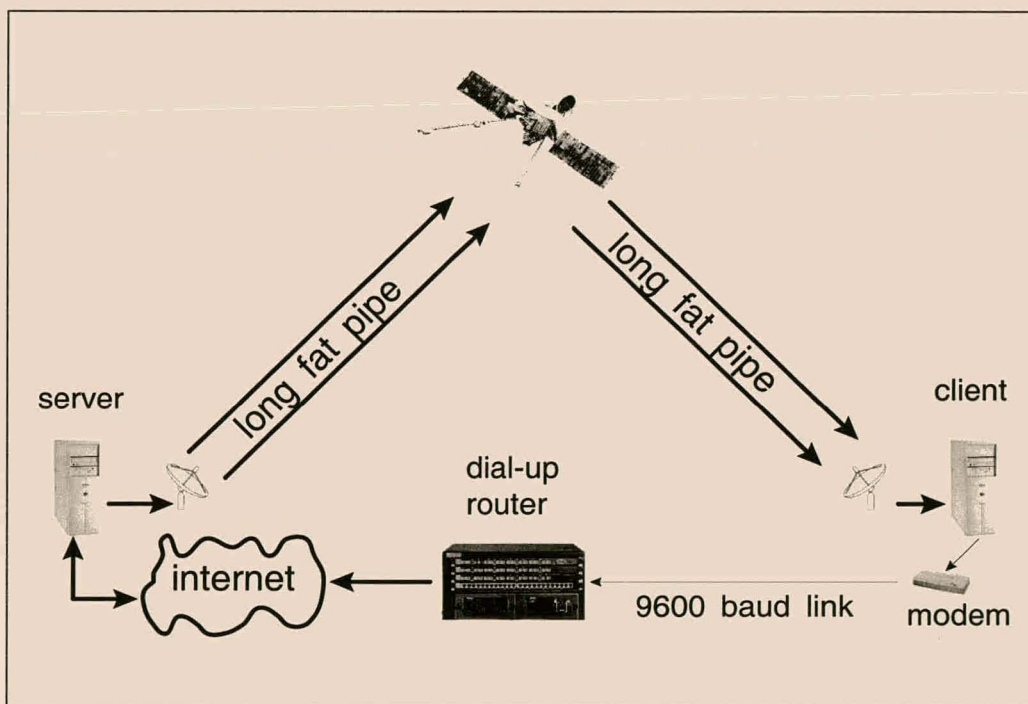


Figure 1.1: The high-latency, asymmetric network configuration studied

1.1 TCP Weaknesses in High-Latency Asymmetric Networks

The literature, practical experience and observations made using the test platform (all to be described in detail later), point to certain weaknesses in TCP performance caused

by the characteristics of this network. These are summarized briefly and intuitively in table 1.1.

Table 1.1: Weaknesses of TCP

Characteristic	Weakness
Latency	<ul style="list-style-type: none"> - Decreases forward throughput - sender unable to maintain sufficient in-flight data - Long slow start decreases responsiveness - Congestion control only becomes effective 1 RTT later
Asymmetry	<ul style="list-style-type: none"> - Decreases forward throughput - reverse link characteristics act as bottleneck to forward throughput - Forward transfer becomes highly sensitive to congestion on reverse link
Asymmetry + Latency	Under congestion, if the sender takes corrective action immediately, the effect is only seen 1 RTT later at the point of congestion. Any in-flight data has already been sent and will thus worsen congestion by placing additional stress on queues. Robustness decreases.

1.2 Spirit of the Research

To sketch the spirit within which this research is done, the following is quoted from [2]:

- Robustness is more important than efficiency.
- The Internet is characterized by heterogeneity on many levels.
- Complex systems are not best designed from scratch and on paper.
- Development and deployment in the infrastructure is of necessity incremental.
- Changes in the Internet can be unanticipated and uncontrolled.

- The Internet architecture and scale make requirements for global consistency problematic.

1.3 Criteria for Evaluating Improvements

In keeping with the spirit outlined in the previous section, *robustness* is the prime criterium. Principles of TCP congestion control should be adhered to, and not subordinated to the goal of more closely approximating optimal efficiency.¹

To help keep the underlying levels of congestion low, improvements should also adhere to *principles of active queue management*. Some of these issues are discussed in [3]: although this is done in terms of gateways, it will be seen that the serial queue on the client bears certain similarities to a bottleneck router. Behaviour such as lock-out can occur, where some traffic flows hog the serial queue so that other traffic (such as outgoing e-mail) cannot get space in the queue. Such examples will be seen in Chapter 6.

Another requirement in similar vein is that of *fairness*: improvements to certain types of TCP must lead to those streams being fair to other traffic streams, and to the network as a whole. This is closely related to active queue management.

The improvement in *overall session throughput* is the final criterium. As discussed in [4], there are no formal TCP performance standards, but experts generally agree that if a link is otherwise idle, a TCP connection should be able to fill it.² If a link is shared between N users, each TCP connection should get about $\frac{1}{N}$ of the bandwidth.

1.4 Approach to the Work

This project is approached from an engineering perspective: to build a system that has increased efficiency (in terms of throughput), yet maintains robustness.

¹If one person disabled their congestion control, they might notice an improvement in performance; but if everyone disabled congestion control, there would be no more Internet.

²This currently does not happen with satellite TCP.

Standard TCP is used as starting point, with modifications holding as closely as possible to standard TCP extensions and the spirit of TCP/IP research. In this fashion, the problems, solutions, results and conclusions of this study are also applicable to other related problems, e.g. asymmetric TCP over xDSL, or wireless links, or TCP via satellite telephone.

The TCP/IP research community normally tests protocols and enhancements by first running them within a software simulator such as *ns*. This allows easy observation of internal variables, and is known as *network simulation*.

The second step would be to build these enhancements into a physical laboratory test network to see how they perform in the context of actual hardware, kernels and network implementation. This is called *network emulation*.

Once emulation proves that the techniques are reasonably safe so as not to cause congestive collapse on the Internet (or in any other way affect the Internet robustness or reliability), the techniques should be tested via the global Internet. This step is important since there are no typical Internet configurations [2, 5, 6], and different TCP/IP implementations have differing idiosyncrasies [7].

Laboratory emulation by necessity restricts the number of possible configurations which can be observed, hence possible performance problems can be missed. However, if a system performs well in the emulator, it holds promise to perform well on the Internet.

The modifications / enhancements studied in this thesis, have mostly all been studied (in isolation) in network simulators. This thesis examines combinations of these enhancements in the context of a *network emulator*.

Although testing these results in the actual Internet is a vast amount of work on its own, it will be seen that the satellite provider has the parameters of the forward network under its control. Consequently, a satellite delivery network can be more controlled than most other Internet delivery environments. It could thus be argued that the notion of a 'typical configuration' holds slightly more relevance than in other Internet environments.

1.5 Working Method

Because of the closed-loop nature of TCP, TCP engineering can be a difficult, ad-hoc process.

The working method followed in this study is:

1. Firstly examine the modifications required to obtain steady-state throughput improvement.
2. Now, see how these modifications affect the various dynamic situations applicable to satellite networks. Evaluate in terms of the robustness criteria, etc. Go back and make changes as necessary.

It will be seen that many (sometimes unexpected) interactions exist between various modifications / enhancements. Hence, each change has to be tested against the previous changes in all the various dynamic situations. The work is thus effectively done in a closed-loop fashion.

1.6 Comments on Mathematical Modeling

The Internet is unparalleled in its growth, heterogeneity and unpredictable or even chaotic behaviour of its traffic. However, even though fractal mathematics has made a promising start toward solid characterizations of Internet traffic, it has still barely made a dent [5].

Consequently, mathematical models are of necessity mostly limited to first-order approximations, with the bulk of development work depending on experimental methods.

1.7 Overview of the Literature

Much research related to TCP via satellite is summarized in [8] and the ongoing work of [9]. This frequently relates to the isolated testing of certain extensions, often in

the context of network simulation, and often with respect to symmetric satellite links (e.g. [10], where the return path also traverses the satellite).

Some research has focused on long-haul space links (e.g. to space probes) [11]. Although this is not directly applicable, many shared problems do exist.

The effect of asymmetry has been studied to a lesser degree, with much research concentrating on terrestrial wireless IP networks, e.g. [12].

This thesis incorporates publications as recent as July 1999, which have focussed on the use of ACK compaction in satellite IP delivery via DVB-S dial-up networks [13].

This thesis concentrates on aspects which have not previously been studied much. This includes the quantification of asymmetric limits on forward throughput and issues of active queue management, especially relating to the effect of VJ header compression with dropped packets and the effect of slow start on the serial queue.

1.8 Structure of the Thesis

Chapter 2 gives an introductory overview of the basic TCP characteristics which lead to reduced performance. In Chapter 3, a network architecture is proposed which lends itself to the optimization of satellite TCP.

Chapter 4 discusses the creation of a network emulator to act as test platform for the thesis.

In Chapter 5, problems with and improvements to TCP's steady-state throughput are studied. The effects of dropped packets are examined in Chapter 6, and the slow start problems studied in Chapter 7.

Knowledge gained from these Chapters is used in Chapter 8 as a set of 'design guidelines' to propose network designs for optimized performance. General conclusions and recommendations follow in Chapter 9.

Appendix A discusses some preliminary results, obtained with a new variant of ACK compaction.

1.9 Intended Audience

Although this thesis is primarily intended to aid in improving TCP via asymmetric dial-up satellite networks, it contains research which may be applicable to other asymmetric or space networks as well.

It should also be of value to the TCP/IP research community as a whole, by pointing out issues which need to be addressed in future revisions.

This thesis may also give insight to designers of other delivery protocols (such as for real time media delivery), which have to function over high-latency asymmetric networks, and have to take into account issues of congestion and flow-control.

1.10 Terminology

In this thesis, the *forward*-direction will always refer to transmission from the server, via satellite, to the client. The *reverse*- or *return*-link will refer to transmission from the client, via the dial-up link, back to the server.

In any abbreviations such as *Mbps* or *kB/s*, the lower-case '*b*' refers to *bits* and the upper-case '*B*' to *bytes*.

This thesis defines $1 \text{ MB} = 1024 \text{ kB} = 1048576 \text{ bytes}$.

Chapter 2

Relevant TCP Characteristics

This chapter examines, in an introductory way, some of the design aspects of TCP/IP which have an impact on satellite performance.

2.1 TCP/IP Implementations

There are many different implementations of TCP/IP for the various computer platforms. Many are based on the various BSD NET releases (see [7]), while some (like Linux) were developed from scratch.

Consequently, each implementation has its own idiosyncrasies, bugs and differing interpretations of (and level of compliance with) the various standards. In fact, in many cases implementations adhere only partly to the standards because of programming difficulties. Some typical implementation problems are described in [14].

It is thus quite possible that an experiment may yield completely different results if performed with different implementations (as pointed out in [6]). One should thus always endeavour to think in terms of the spirit of TCP, and to seek to predict the results in terms of this spirit before experiments are carried out. This is the method followed in this thesis.

2.2 The TCP Packet and Header

A TCP packet (also called segment) is encapsulated within an IP packet. For IPv4, this adds an IP header (usually 20 bytes) to the TCP packet.

IP is responsible for making a best effort of delivering the packet. It does not guarantee error-free delivery.

The TCP packet thus contains a TCP header which includes, among other things, a checksum. Packets with incorrect checksums are simply discarded.

The header contains source and destination port numbers. Together with the IP source and destination address (from the IP header), this 4-tuple uniquely identifies the two communication end-points. A computer will typically associate traffic on a particular TCP port with the process *bound* to that port.

The header contains flags, such as SYN which indicates the start of a connection, FIN which indicates the end, and RST, used to terminate an active connection. An ACK flag is also present.

TCP can thus acknowledge received incoming data in the same packet with which outgoing data is sent to the remote port of the end-point 4-tuple. Acknowledgements thus piggy-back on data flowing in the opposite direction. Conversely, if a packet contains no data, but only acts as an ACK, that packet will only consist of the TCP header without additional overhead.

The minimum (and typical) TCP header is 20 bytes, but it can extend to 60 bytes carrying additional *options* (such as timestamps, selective acknowledgements, etc.).

2.3 TCP Sequence Numbers

The TCP header contains a 32-bit sequence number. This indicates the position in the data-stream of any data sent in the packet.

During the connection setup three-way handshake (SYN), the initial sequence number is indicated. Subsequent packets will have sequence numbers commencing from this

initial number.

A new connection normally has its initial sequence number chosen to be larger than the last sequence number used for the previous connection. This provides a measure of security: Sometimes, IP packets can go missing in the Internet¹, only to emerge again some time later.² At this later time, it is possible that another, *different* connection may now be active between the *same* 4-tuple end-points. The re-emerged packet does not form part of this data-stream, and is discarded because its sequence number does not fall within the range of sequence numbers for the new connection.

If transmission occurs at very high bandwidth, it is possible for the 32-bit sequence space to *wrap* in less than 2 minutes. Unfortunately, TCP's 32-bit wraparound sequence space, spread over 2 minutes, gives an effective data rate of only 286Mbps [4].

To counteract this problem, the timestamps-option can be used with the PAWS algorithm (Protection Against Wrapped Sequence numbers) [15]. This extension is sufficient for link speeds up to between 8Gb/s and 8Tb/s.

2.4 TCP Acknowledgements

The TCP header contains a 32-bit ACK sequence number, valid if the ACK flag is on. This is the sequence number plus 1 of the last successfully received data byte.

ACKs are cumulative. In other words, a single ACK acknowledges that *all* data up to the sequence number has been received. Say for example a receiver receives bytes 1–1024, followed by 2049–3072. It cannot ACK this new segment. All it can send is another ACK with 1025 as the ACK number, i.e. a *duplicate ACK*.

When TCP receives a new packet, it may *delay* the ACK for up to 500ms, either to allow it to accumulate that ACK with the next incoming packet's ACK, or to wait for possible outgoing data to piggy-back on. This is called *delayed ACKing*. Nonetheless,

¹E.g. because of dynamic routing changes

²The IP header contains a time-to-live field. It is generally assumed that an IP packet will live for up to 2 minutes.

TCP must ACK at least every second data packet, and every out-of-order packet must be ACKed with a duplicate ACK.

2.5 Selective ACKs (SACKs)

A new extension [16] allows selective ACKs to be sent as options in the TCP header, provided that this has been negotiated during the three-way handshake. In the previous example, when bytes 2049–3072 are received, the receiver would still reply with an ACK 1025, but could include a SACK 2049–3072 as option in the header.

2.6 The Three-Way Handshake

When a client opens a TCP connection to a server, a *three-way handshake* is executed with various flags set in the TCP header. This is illustrated in figure 2.1, and is used to synchronize sequence numbers between the two TCP stacks.

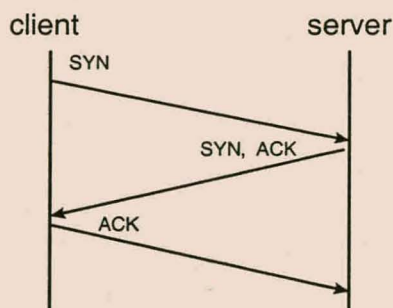


Figure 2.1: The TCP three-way handshake

SYN-segments options are also used to indicate if the respective peer is able to support extensions such as timestamps, selective acknowledgements (SACKs) and window scaling.

After the three-way handshake, transmission of data may commence in both directions.

2.7 The Maximum Segment Size (MSS)

The MSS is the maximum number of data bytes that TCP can send per packet (segment). During SYN segments, each end can indicate to the other the MSS it is willing to *receive*, using an option.³ If the MSS-option is not specified, the other end will assume that it should be 536 bytes.

This determines the overhead of packet headers vs. segment data. As indicated in [1], optimal MSS for any path is still very much an open question. Ethernet interfaces perform well with an MSS of 1460 bytes, but serial interfaces have been found to give better performance at 536 or 256 bytes.

Network interfaces have a limit on the number of bytes-per-packet that can be sent or received: the maximum transmission unit (MTU). The *advertised MSS* should be constrained by the MTU of the network interface over which the data is *received*. The maximum MSS is then given by

$$\max(MSS) = MTU - iph - tcph \quad (2.1)$$

where *iph* is the size of the IP header and *tcph* that of the TCP header (both typically 20 bytes).

For Ethernet, MTU=1500 bytes. Note that *tcph* can be larger than 20 bytes if TCP options are passed in the header. It is important to mention that in this case, most implementations will still advertise the MSS as if *tcph* = 20 — the onus is then on the remote end to adjust the actual MSS transmitted if it includes TCP options in the transmitted TCP headers.⁴

Note also that the MSS can be larger than that specified in eq. 2.1. This is avoided since IP packets will have to be fragmented. Fragmentation and defragmentation require extra time overhead, reducing performance.

Path MTU discovery should be used to determine the smallest MTU size of interfaces over which the forward data travels, to avoid fragmentation.

³This is often incorrectly referred to as a negotiated option. No negotiation takes place, however.

⁴In this thesis, MSS will always refer to the actual segment data size available.

The *recvwin* thus determines the amount of data which can be in flight (i.e. unacknowledged).

An additional window, the congestion window *cwnd*, is used in congestion control. The minimum of *cwnd* and *recvwin* is actually used to calculate the usable window.

It should be noted that the server must have sufficient socket buffer space to buffer the data which has not been ACKed. If this buffer space is too small, it will constrain the in-flight data.

2.8.2 ACK Self-Clocking

The sender can only open the usable window (and hence transmit new data) at the rate at which ACKs are received.

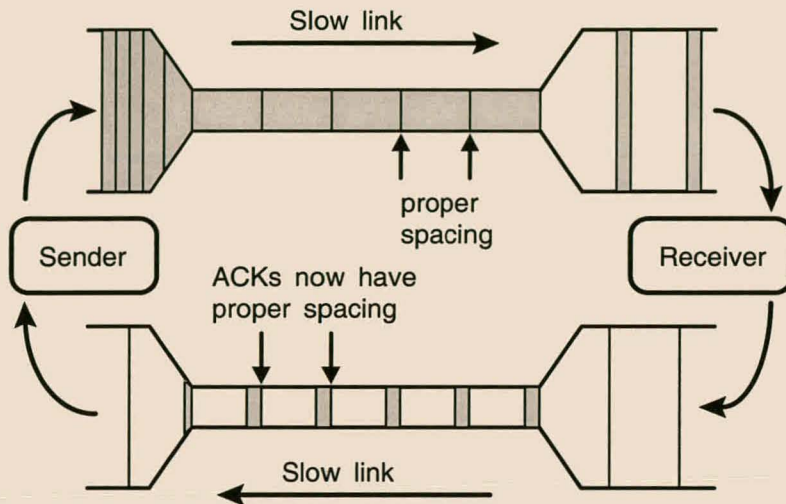
The network paths will ‘traffic-shape’ the forward data packets and return ACKs, so that ACKs will arrive at the steady-state rate at which the network can support it. New data will now be clocked out at this rate.

This feature of TCP is called *ACK clocking*, and is illustrated in figure 2.3 (after [17] and [18]). As explained in [17], this help keep the network in equilibrium by ‘conservation of packets:’ new packets are injected only at the rate at which packets leave the network.

In the asymmetric satellite architecture, it will be found that the picture is somewhat different: Instead of having a bottleneck in the forward path, the outgoing low-speed interface queue on the client effectively acts like a bottleneck router. This happens because the client essentially converts a stream of high-speed incoming segments into ACKs that have to traverse the (slow) output-queue.

2.8.3 ACK Compression

As discussed in [18], it is possible for the returning ACKs to traverse a router (such as the dial-up router) where the ACKs are queued with other traffic streams. This can *compress* the spacing between some ACKs, especially if the ACKs get routed to a



The horizontal dimension represents time; the shaded area is proportional to the packet size.

Figure 2.3: TCP self-clocking in general

higher bandwidth network (as is the case with dial-up routers). For example, if a group of ACKs are queued together behind some other traffic, they will eventually be bursted out at the speed of the outgoing interface.

This is called *ACK compression*, and may lead to burstiness in forward transmissions, which in turn leads to ACKs being generated in clumps. This may increase the probability of further ACK compression at the router.

Although the effect of ACK compression averages out during steady-state transfer, it does hold implications for bandwidth estimation during connection start-up.

2.9 Congestion Control

2.9.1 Slow Start

To reach the ‘equilibrium’ of steady-state, the network has to be gradually probed for capacity. This is accomplished by slow start and congestion avoidance.

Initially, at connection start, *cwnd* is set to an *initial window*, usually 1 packet.⁵ This allows the sender to transmit 1 packet. For each ACK the sender receives, *cwnd* is increased by one MSS. This allows the sender to transmit the number of packets which have been ACKed (since the window position slides to the right), *as well as* an extra packet for each ACK received.

The transmission rate thus increases exponentially with time.

2.9.2 Congestion Avoidance

TCP keeps track of an additional state variable, the slow start threshold size *ssthresh*, which is set to an arbitrary value at connection start (often equal to *recwin*). As soon as *cwnd* has increased sufficiently so that $cwnd > ssthresh$, slow start ends, and is replaced by *congestion avoidance*.⁶

The aim of congestion avoidance is gradually to probe the network for any additional capacity that may suddenly become available. During congestion avoidance, *cwnd* is increased by

$$segment\ size * \frac{segment\ size}{cwnd}$$

This means that *cwnd* will be incremented by one segment per round trip time (RTT), if all segments in the window have been ACKed. Growth of *cwnd* is thus *linear* during congestion avoidance.

⁵*cwnd* is actually stored in terms of bytes, hence 1 packet equals MSS bytes.

⁶Note that it is always the lesser of *cwnd* and *recwin* which determines the usable window.

2.9.3 Timeouts

The sender keeps track of the RTT of transmitted segments and their ACKs, and uses this to *estimate* a proper *retransmission timeout* (RTO).

If the ACK for a segment is not received before the RTO timer for that segment expires, TCP retransmits that segment. TCP now assumes that the network has to be congested, hence *cwnd* is reset to one segment size, and *ssthresh* is set to half of the current window size (the minimum of *cwnd* and *recvwin*).

The connection will thus re-enter slow start, but will maintain this exponential growth only to the much lower *ssthresh* value, whereupon it will change to the linear-growth (and hence much slower) congestion avoidance algorithm.

2.9.4 Fast Retransmit and Fast Recovery

If a segment is lost on the way to the receiver, the receiver will start generating *duplicate ACKs* for the successive out-of-order segments. After the sender receives the third duplicate ACK, it assumes that a segment has been lost.

To avoid the unnecessary retransmission of the out-of-order segments already held in the receiver's buffer, as well as the long wait associated with the retransmit timer, the sender uses the fast retransmit / fast recovery technique.⁷

The sender retransmits the missing segment and sets *ssthresh* to one half of *cwnd* (or *recvwin*, whichever is the smaller), rounded down to the nearest multiple of the MSS. Then *cwnd* is set equal to *ssthresh* plus three MSS (i.e. one half of the previous *cwnd*, plus the three received segments that have left the network).

The *cwnd* is updated by one MSS for each additional duplicate ACK received. If the *cwnd* allows, the sender transmits a new segment into the network. When a non-duplicate ACK is received, *cwnd* is set equal to *ssthresh* and the sender enters the congestion avoidance phase.

⁷If the sender is receiving duplicate ACKs, data is getting through the network and returning to slow start is unnecessary.

Effectively, this means that the sender allows half the pipe to drain (under the assumption that the network is congested), and then assumes transmission at half the previous rate, with linear congestion avoidance.⁸

2.9.5 SACK and FACK

SACK uses selective ACKs in an extension of the fast retransmit and fast recovery algorithms. It keeps track of the amount of data on the network and retransmits or sends new data when the amount of data on the network is less than *cwnd*. The use of SACKs allows more intelligent retransmissions.

Forward Acknowledgements (FACK) uses the SACK information, but is a departure from standard TCP. FACK separates the recovery and congestion control algorithms, and uses techniques such as rate-halving to inject new or retransmitted data into the network. The amount of data sent in rate-halving then determines the *cwnd* after recovery. This is an attempt to better estimate the proper window size after a congestive loss event.

2.10 The Bandwidth-Delay Product

Because of the propagation delay involved in transmitting data via a given channel, that channel can effectively ‘store’ a certain amount of data, or, in other words, it has a certain ‘capacity’ for in-flight data.

This quantity is given by

$$\text{capacity [bits]} = \text{bandwidth [bps]} \times \text{channel delay [sec]}$$

which is the bandwidth-delay product.

It will be seen that, to use a channel fully, TCP senders must be able to sustain this amount of data in flight.

⁸These multiplicative decrease, linear increase algorithms are argued by [17] (and its references) from control systems principles.

2.11 Asymmetry

From [12] the following definition of asymmetry is used: *a network is said to exhibit network asymmetry with respect to TCP performance, if the throughput achieved is not solely a function of the link and traffic characteristics of the forward direction, but depends significantly on those of the reverse direction as well.*

Although this thesis mostly uses the concept of bandwidth asymmetry, asymmetry can also refer to differences in error-rates, path propagation times, etc.

2.12 Conclusions

This chapter has given an overview of the basic design characteristics of TCP (which will be seen in later chapters to impact on asymmetric satellite performance), and has given a taste of the interdependent nature of TCP functioning.

Over the years, TCP has grown as the Internet has grown⁹, becoming a complex interwoven system, yet still reflecting some of its 'naive' origins.

Even today, network control primarily relies on the communications end-points being 'well-behaved.' Instead of the network itself supplying information about its status, much depends on TCP's ability to infer certain properties of the network from its observations at the 'edges' of the network. As will be seen, this approach has definite limitations, which translate into reduced efficiency on modern (e.g. satellite / asymmetric) networks.

⁹For example, congestion control was added to TCP after the Internet started suffering from *congestive collapses* in October 1986.

Chapter 3

Satellite Network Architecture Design

This chapter discusses the design of a satellite delivery network architecture which allows optimization of TCP/IP performance.

It is clear from research such as [8, 9] that control over *both the data sender and receiver* is required to optimize TCP performance effectively. In fact, it will be seen that most of the more substantial modifications have to be made to the sender.

Of course, in the Internet, senders are distributed globally, making it impossible to access all for modification. Furthermore, the enhancements for satellite performance may degrade terrestrial performance, hence unilateral modification of all TCP senders is not recommended.

The problem would be solved if the satellite delivery service could somehow provide a surrogate sender, which would act on behalf of the real TCP source. This surrogate sender would then be under full control of the satellite provider.

3.1 Providing a Surrogate Sender

This can currently be done in two ways:

- TCP Spoofing
- TCP Proxy Server

3.1.1 Spoofing

The basic idea calls for a router close to the satellite link to send back (spoof) acknowledgements for the TCP data, to give the actual sender the illusion of a short delay path. The router then suppresses ACKs returning from the receiver, and takes responsibility for retransmitting any segments lost over the satellite link.

This router could thus use a different protocol, more suited to the satellite environment, to deliver the data to the receiver.

Many problems with this scheme are pointed out in [4]. Specifically, the router must buffer the data segments to be able to retransmit them in case of loss over the satellite link. Secondly, the ACKs from the receiver must be diverted into this router. In many Internet paths, this can be problematical. Thirdly, if dynamic routing changes during a transfer, data may be lost.

All of these problems can be solved, but the solution always requires additional resources to track TCP state per connection. The magnitude of computing power required thus becomes similar to that needed for a proxy server.

3.1.2 Proxy Server

In this case, the client at the receiver does not make a TCP connection directly to the actual sender. Instead, it sends the TCP request to a proxy server. The proxy server then retrieves the data from the actual sender (using a separate TCP connection), and resends it to the client.

Enhancements, such as from [8], can then be made to the proxy sender stack. It is even possible to replace the TCP protocol on the proxy sender with a protocol more suited to the satellite environment.

The proxy server solves all of the problems exhibited by TCP spoofing, but requires extensive computer resources to buffer the incoming data and to keep track of the TCP state of all connections.

Proxy server technology has, fortunately, matured during the last few years, and is

already widely deployed by ISPs, mainly for the savings in international bandwidth provided. When a client requests a file that has recently been cached, the proxy will check with the actual sender whether the copy of the file is still valid. If possible, the proxy will then transmit the cached copy, without having to re-request it via the incoming link.

Proxy servers are also useful with certain Internet security technologies, such as firewalls.

It is possible to lighten the load on a single proxy server by replacing it with a proxy server farm. Proxy requests and cached data are then shared among the servers as needed.

TCP Services

Of course, a proxy system is now required for each type of TCP service that is to be enhanced. Proxy systems already exist for HTTP and FTP transfers, as well as for IRC and for e-mail.

For protocols such as News (NNTP) and e-mail (POP/IMAP), the satellite provider may already provide dedicated servers. In this case, the enhancements can be made directly to these servers without having to go via a proxy. This should be done with caution, however, since the terrestrial Internet may have to be isolated from the modifications made to the server TCP stack: refer to section 3.2.

Other TCP services (such as telnet) could transit the satellite without using the proxy server. These services would then not benefit from the TCP satellite enhancements. Even better, services which require minimum delay and low bandwidth (such as telnet), can and should be routed via the terrestrial dial-up network.

Breaking End-to-End Semantics

TCP spoofing and TCP proxy servers both effectively break the end-to-end semantics of TCP between the receiver and the original sender. However, the proxy server does

so in a clean way, by cleanly and clearly splitting the transfer into k separate end-to-end TCP connections.

3.2 Architecture Design

With proxy server technology now the norm at most ISPs, this is the solution chosen for the system architecture in this thesis.

For this thesis, TCP (with enhancements) is used to transmit the file from the proxy to the receiver. This enables the proxy server to run standard proxy software such as *squid*.¹ Henceforth, the term 'sender' shall refer to the proxy server sender stack.

It would be possible to replace TCP via the satellite with something more suitable for this last hop. Although this is outside the scope of this thesis, the issue is briefly examined in Chapter 8.

Figure 3.1 shows the design of a typical proxy-enhanced satellite architecture.

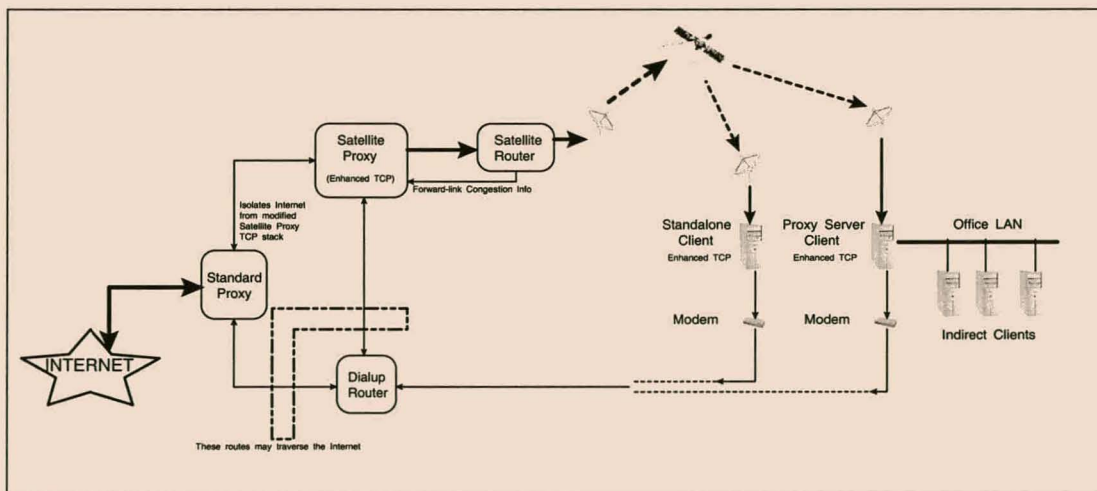


Figure 3.1: A network design for high-latency asymmetric TCP delivery

The client is connected to a dial-up or leased-line service via a modem. This is primarily to send out requests for data, and ACKs for the data received via the satellite. However, it can also be used to receive incoming data not routed via the satellite.

¹<http://squid.nlanr.net>

Bulk data transfers are served by the satellite proxy server (or server farm). This proxy server has modifications to its TCP sender stack to increase TCP performance via the satellite.

To isolate the Internet from these modifications, it is recommended that the satellite proxy obtain its data from a standard proxy server, rather than directly from the Internet.² This is especially true for enhancements such as slow start and congestion avoidance modifications, which may lead to congestive collapse on the global Internet.

The satellite proxy server routes the traffic via a router over the satellite network.

Multi-client Networks

The client may consist of a stand-alone system (such as a domestic user), or an entire client network (such as an office or Internet cafe). In the second case, use of a client proxy server is recommended, via which the actual clients request the data. This holds the following advantages:

- Enhancements are only required to the client proxy server stack, not to all office clients.
- By caching data, the client proxy server eliminates duplicate transmission of duplicate requested data via the expensive satellite network. Satellite providers normally limit the bandwidth to each of their clients, and levy a per-megabyte traffic charge.
- The client proxy server isolates the satellite-dial-up network from any congestion on the office LAN.

²Current TCP implementations can run only a single type of TCP to all addresses.

3.3 Issues of Multihoming and Routing

The client contains two network interfaces (the satellite interface and the dial-up interface). Each could have its own IP address, or could share a single IP address.

The network should have the ability to route only bandwidth-intensive traffic via the satellite, and the rest via the dial-up link. This improves interactivity for services such as *irc* and *telnet*, and saves valuable satellite bandwidth for the applications which require it.

The following summary is not intended to be exhaustive, but to place the two configurations into perspective.

3.3.1 Single IP Address

This is the simplest configuration. Routing is set up so that all packets from the Internet are routed to the dial-up interface, but that packets from the enhanced servers (satellite proxy servers, satellite mail servers, etc.), are routed via the satellite network.

The routing thus relies on the knowledge of the topology, and only specific services are routed via the satellite. Routing thus effectively happens at the TCP (or UDP) level.

Routing could also be done on the basis of the TCP port or type-of-service (TOS) field. Unfortunately, not all routers support TOS routing, and not all applications set the TOS correctly.

A disadvantage to this technique is that a new service which may benefit from the high-bandwidth satellite path (such as UDP streaming media), will be routed via the slow dial-up network until such time as special routing provision is made for it.

3.3.2 Multiple IP Addresses

This is the more flexible configuration. Packets are routed to an interface by simply routing to the respective IP address.

The actual functioning is dependent on the implementation, but with the BSD Sockets API [7], it works as follows:

The client application can bind to a particular *source* IP address. If it binds to the source IP associated with the dial-up interface, packets destined *to* that client application will then be routed by the Internet to the dial-up interface. If it binds to the source IP associated with the satellite interface, packets to the application will be routed to the satellite interface.

Unfortunately, most applications do not provide the ability to specify to which source IP address it should bind. In this case, the outgoing packets will be labelled with the *source* IP address of the interface through which the packet is sent.

This means that if a web browser (say), transmits a request for service to the satellite proxy, the IP packet will have as its source IP the address of the *dial-up* interface. This in turn means that the satellite proxy will try to route the resulting packets back to the dial-up interface.

To circumvent this problem, it may be necessary to use a network *tunnel*.

Unfortunately, it may still be necessary to provide specifically for each new high-speed service (except if the new client can bind to a particular source interface³).

Although this is the more flexible configuration, the application interface does not usually allow this flexibility to be exploited.

3.4 Assumptions

The following assumptions are made for the purposes of this thesis:

³Or if all outgoing traffic is routed via the tunnel, which negates the ability to route incoming traffic to the serial interface.

3.4.1 Packet Corruption and Error Rates

When a packet is dropped, TCP takes this as a signal that the network is congested. To uphold this assumption, the probability of packet loss due to *corruption* must be $\ll 1\%$, otherwise performance will be seriously degraded [17].

In this architecture, such a low packet corruption error rate is assumed. In practice, this can be achieved by Forward Error Correction (FEC).

In the case of a DVB-S network, the specification [19] gives the bit error rate as 10^{-10} to 10^{-11} , achieved by an underlying use of FEC. For a packet of 1500 bytes, this translates to a worst-case packet error rate of roughly 0.00012%.

Hence a DVB-S network without additional FEC is suitable for TCP delivery. Some providers, however, report a DVB-S IP packet-loss rate on the order of 1% (which is unacceptable), and hence use additional FEC on the IP packets. The reason for this is currently unclear, but may be related to the present generation of satellite receiver cards (which have known performance problems⁴) possibly dropping some of the underlying 187-byte DVB frames.

3.4.2 Congestion Control

Except for dropped packets due to corruption or signal-fading (the effect of which is studied in Chapter 6), it is extremely important to realize that congestion on the satellite link is completely under the satellite provider's control. By using techniques such as traffic shaping, pacing and ICMP Source Quench messages, the satellite router can supply feedback to the proxy and hence guarantee that congestion does not occur on the satellite path.⁵

Congestion can, however, occur on the return path and in the client machine on the outgoing low-speed queue.

⁴Such as not being able to sustain their advertised maximum throughput

⁵This in itself is a completely separate research field.

3.4.3 Proxy Retrieval

It is assumed that the proxy server retrieves the data quickly enough to send at any rate the studied enhancements can achieve. This is for illustrative purposes only; should this be untrue, the proxy will simply transmit data when it becomes available, hence the throughput to the client will become constrained by proxy's data retrieval rate.

3.4.4 Modem Settings

It is assumed that the dial-up link is error-corrected with standard modem error-correction (e.g. V.42, MNP 2–4).

Modems can introduce a substantial latency. For example, round trip times of 260ms may be measured when pinging over a standard 28800 baud dial-up link (whereas the packet transmission time over the modem is only about 26ms) [20].

As shown in [20], the modem latency can be reduced somewhat:

- Time is wasted in transmitting data from the computer via the serial port to the modem. *The serial port speed should be set as high as possible, irrespective of the modem line-speed.*
- Avoid the modem's on-board data compression. It is better to use the faster computer CPU for software compression (e.g. the compression in the PPP system). Note that pure ACKs are not very compressible [21].

3.4.5 Latency

The exact latency of a satellite connection depends on the specific configuration, with a round trip time of roughly 600ms a typical figure.

This consists of the following components:

- The geostationary satellite-hop delay of about 240 to 300ms, which may be increased by the presence of Doppler-buffers and interleavers for FEC. [22]

- The one-way serial latency of around 130ms. [20]
- Routing and processing overheads.

It is possible that delivery may take place via multiple satellite hops.

Comments

In a particular configuration, the total serial return-path latency should usually be less than the total satellite latency. In this case, TCP connections may be made more responsive by routing the second part of the three-way handshake (i.e. the server's SYN ACK to the client), via the serial dial-up link, rather than via the satellite path.

In calculating the bandwidth-delay product for a given network, the channel delay is normally taken to be the network RTT. It should be recognized, however, that in this asymmetric network, optimal efficiency depends on being able to fill the *satellite*-portion (the long fat pipe) of the network. Hence, if the serial return-path latency forms a substantial part of the total latency, the serial latency may have to be *omitted* from the RTT for certain calculations.

3.5 Conclusions

This chapter has examined the design of an overall satellite network architecture, which facilitates the effective enhancement of TCP by allowing the satellite provider to isolate and obtain full control of the TCP sender.

Such control has other important ramifications:

- If the bulk of the enhancements can be carried out on the sender, the cost and effort associated with client-side modification are substantially reduced.
- The client may also rely on standard (terrestrial) Internet connectivity. It would be unwise to make substantial modifications to client TCP stacks which may accidentally impact on terrestrial Internet stability.

Other facets of architecture design will be encountered in Chapter 8. These are the results of observations (to be discussed) of the effect of various TCP enhancements.

Chapter 4

The Experimental Platform: Network Emulation

This chapter discusses the creation of an emulation platform to test the TCP enhancements.

Modifications and enhancements to TCP are normally first simulated with a software package such as *ns*.¹ This allows the observation of parameters which are not so easily measured on an actual network.

As pointed out in [23], simulators usually run an independent specification of network code, rather than the code used in real networks. This can cause simulators to fail to mimic subtleties in the real code.

In a network emulator, the network topology under consideration is physically constructed. Code can then easily be moved from the emulator to the real Internet for final testing.

A drawback to using an actual network, since actual kernel-level code is involved, is that it can be difficult to measure certain internal variables without impacting on performance.

¹<http://www-nrg.ee.lbl.gov/>

4.1 The Platform

The main components which must be emulated are:

1. The network latency
 - Transmission Delay
 - Queueing Delay
 - Propagation Delay
2. The forward satellite bandwidth
3. The restricted return bandwidth

At the time when work commenced, code for the Ohio Network Emulator [23] was not available. Hence, an emulator was designed and implemented from scratch. In the mean time, similar general-purpose tools, such as *dummynet*², have appeared.

4.2 Latency

As discussed in [23], network latency consists mainly of the following:

1. Transmission Delay
2. Queueing Delay
3. Propagation Delay
4. Processing Overheads

²<http://www.iet.unipi.it/~luigi/ip.dummynet/>

4.2.1 Transmission Delay

The transmission delay is the amount of time it takes a network node to transmit a packet onto a given channel. It is determined using the channel bandwidth and the packet size:

$$\text{transmission delay} = \frac{\text{packet size}}{\text{bandwidth}} \quad (4.1)$$

Note that this is also the reception delay: a network interface must receive the entire packet before it can be processed (although it can receive the packet while the remote node is busy transmitting it, hence the reception delay is *not* an additional delay).

Also note that, for each router in the network path, an additional transmission / reception delay will be incurred, since the router must typically receive the entire packet before transmitting it again.

Because of the asymmetry, transmission / reception delays differ along the forward and return paths.

The transmission / reception delay is modeled in the emulator by the physical network interfaces which transmit and receive the data.

4.2.2 Queuing Delay

Queuing delay occurs when a packet arrives at a router which is already busy transmitting another packet. In this case, the packet is placed in a queue. The queue delay for a given packet P is dictated by the sum of the sizes of the packets in the queue when P arrives. The queue delay for the n -th packet in the queue is thus

$$\text{queue delay} = \begin{cases} 0, & n = 1 \\ \sum_{i=1}^{n-1} \frac{\text{packet size}}{\text{bandwidth}}, & n > 1 \end{cases} \quad (4.2)$$

As will be seen, a bottleneck occurs at the serial interface of the client. The client serial queue acts just like this router queue.

Although other routers may exist in the system, they are not modeled here. Instead, they may be modeled using a single bottleneck fluid approximation model, as discussed in [24], or by routing the return traffic via the Internet.

4.2.3 Propagation Delay

The propagation delay is imposed by the speed of light, and is the time it takes a packet to travel from one node to another along a physical channel.

4.2.4 Comments

Whereas the precise value of the propagation delay is non-critical, important information may be gleaned by monitoring the variations in queue delay, since this gives an indication of network congestion.

It should be noted that the concept of network latency (i.e. packet delay or RTT) is not rigorously defined, and that measurements invariably include the effect of transmission- and reception delays.

In fact, as discussed in [25], measurements of packet transit times are problematical. Echo-based techniques (such as *ping*) unavoidably conflate properties of the forward network path (which properties perturb the sender's original packets, and hence the times at which the target generates its replies), with the properties of the reverse direction. This is clearly undesirable, especially in asymmetric networks, and argues for receiver-based measurements where packet receivers cooperate with senders in order to accurately measure network traffic. This issue is revisited in section 7.1.4, where such packet delay measurements may be used to form estimates of the available network bandwidth.

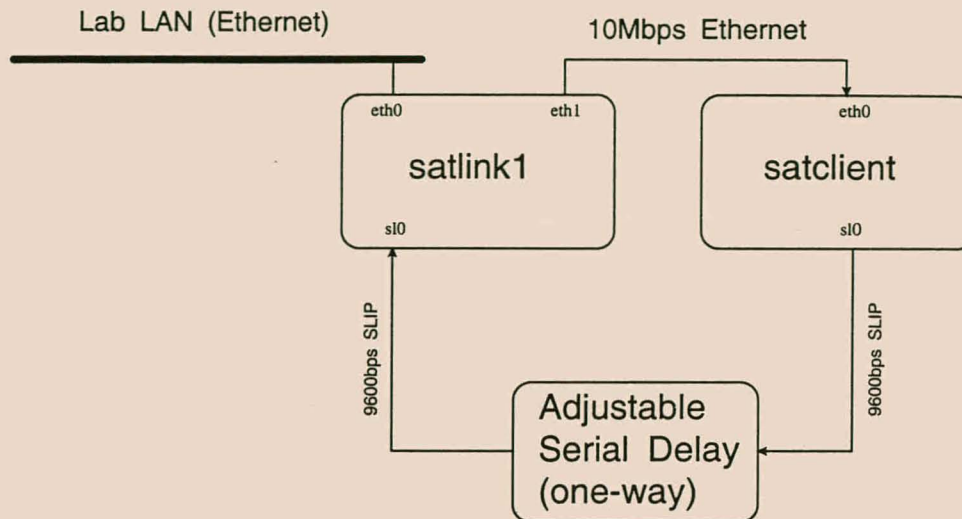


Figure 4.1: The emulation test-platform

4.3 The Emulator

The emulation platform, used to test the proposed TCP modifications, is shown in figure 4.1.

The client is emulated by *satclient*, which contains the client-side TCP modifications. *Satclient* is used to run the TCP client-side software.

The server is emulated by *satlink1*, which contains the server-side TCP modifications. Furthermore, it is configured to forward IP packets (as a router) for *satclient* from *eth0* to *eth1*, and from *sl0* to *eth0*, and to respond to Ethernet ARP requests on behalf of *satclient*.

The server-side software, such as the *squid* proxy server, is hosted on *satlink1*.

The forward satellite bandwidth is easily emulated using a 10Mbps Ethernet. Although DVB-S networks could transport more than this, a 10Mbps Ethernet is sufficient to illustrate the principles, and could easily be replaced by a 100Mbps Ethernet.

The return link is emulated by using a 9600 serial line. SLIP is used on this link rather than PPP, since SLIP is a unidirectional protocol while PPP requires two-way

handshaking. Use of a unidirectional serial system greatly simplifies the adjustable delay software.

The propagation delay is inserted into the serial return link, rather than the forward path. This makes no difference to the experimental network, and eases implementation of the delay, since the return path is much less speed-critical than the forward path.

The adjustable serial delay is implemented using two serial ports on a separate computer (also used for other purposes in the laboratory). Pseudo real-time scheduling techniques are used to ensure that received data are delayed accurately before being retransmitted.

For simplicity, the *satclient* Ethernet and serial interfaces are set to the same IP address (see section 3.3). This is sufficient to illustrate the principles.

The emulator does not induce packet errors, since it is assumed that the underlying forward satellite transmission is error-free enough so that $\ll 1\%$ of TCP packets are dropped due to corruption. Ethernet satisfies this requirement.

Forward dropped packets (i.e. due to rain fading) are emulated by briefly disconnecting the Ethernet cable at *eth1* of *satlink1* (see Chapter 6).

The emulator does not induce errors on the serial link. The effect of these errors are examined separately (see section 6.2.1).

All the machines run the *Linux-2.2* operating system.

4.4 Testing the Emulator

The delay can be verified by *pinging satclient* from *satlink1* (or vice versa). With the delay set to 1000ms, and using standard ping (echo request packets spaced 1 second apart), the echo reply packets arrive typically 1120ms later.

Since the serial transmission / reception time for a 84 byte ping packet is 89ms (see eq. 5.5), the additional 120ms consists of 89ms transmission time and roughly 31ms of processing overhead.³

³Because the serial delay process is not a network system, it does not incur its own packet reception

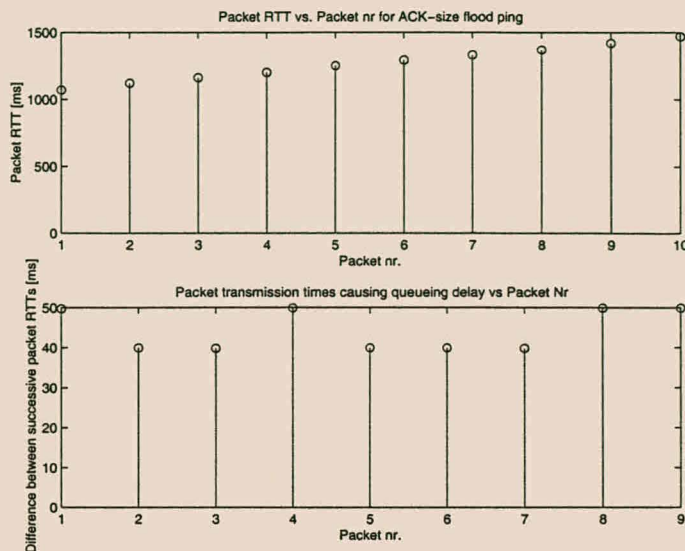


Figure 4.2: Flood pinging the emulator

To check the stability of the timing, flood pinging was used to fill the serial queue with back-to-back packets. The IP ping packet size was set to 40 bytes, to make it the same size as a typical TCP/IP ACK packet. In so doing, this test also illustrates the validity of equation 5.5 (as will be seen, the rate at which back-to-back ACKs can be transmitted constrains the maximum forward TCP throughput).

The result is shown in figure 4.2, with the delay set to 1000ms.

The top half of figure 4.2 illustrates the effect of the basic 1 second propagation delay, as well as the effect of queuing delay (eq. 4.2).

Flood ping generates echo request packets as rapidly as the transmitting interface can send them. Thus a set of echo request packets arrive virtually simultaneously at *sat-client*, which generates a set of back-to-back echo reply packets to fill the serial queue. The first reply is delayed for just over 1 second, while the next reply is delayed just over 1 second plus the time required to transmit the first. The third reply is delayed for just over 1 second, plus the time required to transmit the first two, and so forth, as expected.

/ transmission time (as would be expected for a router). Since the process uses non-blocking IO, data is read and written as soon as possible, i.e. on the byte (not packet) level.

This experiment clearly demonstrates the validity of the statement that the serial queue constitutes a bottleneck: The forward echo request packets are spaced back-to-back over the Ethernet, and then trigger echo reply packets which are spaced according to the serial transmission time.

The bottom half of figure 4.2 shows the difference between the arrival times of successive echo replies. This is approximately equal to the transmission time per packet via the serial link. The results agree well with eq. 5.5, which predicts a transmission time of 43ms for ACK-size packets.

The observed timing varies between 40ms and 50ms since the serial delay process obtains control of the CPU at a 10ms granularity. The delay was later moved to an architecture (DEC Alpha) which supports a process switching granularity of 0.977ms.⁴

The emulator reports any unexpected delays in timing (e.g. due to the scheduler delaying the process). Since the machine was being used for other purposes as well, the emulator sometimes reported variations of up to about 10ms in the timing. This is of no consequence, since the existence of all the traffic streams in the Internet causes timing variations to a particular packet stream.

4.5 Measurements

To observe the effect of TCP modifications, measurements are made as follows:

Packets are captured into a file and analyzed by executing *tcpdump*⁵ (normally on *satlink1*). This ‘eavesdrops’ on the specified network interface, and logs packets and their time of transmission or reception.

TCP connections are then analyzed as a whole by processing the captured files with *tcptrace* (which had to be modified to correctly read SACKs in the Linux environment). The resulting traces are plotted with *xplot*.⁶

⁴This is non-critical, and had to be done simply to accommodate a change in the physical layout of the laboratory.

⁵<http://www-nrg.ee.lbl.gov/>

⁶For *tcptrace* and *xplot*, see <http://jarok.cs.ohiou.edu/software/software.html>

Packet traces in this thesis are done from the viewpoint of the *sender*, i.e. traces show when the *sender* transmits packets and receives ACKs.

4.6 Conclusions

This chapter has discussed the key facets of network emulation. The creation, implementation and successful validation of such a platform was described, as well as the techniques to be used to measure the properties of TCP connections.

Chapter 5

Steady-State Performance

This chapter examines the performance of a single TCP connection under steady-state. For these purposes, a steady-state TCP connection is defined as being in full progress, having left the TCP start-up dynamics. Furthermore, a steady-state connection is not subject to packet errors or any congestion control algorithms.

It will be found that forward data throughput is limited by two main factors: the window-sizes of the forward TCP connection, and the impact of asymmetry on the connection.

Some details of enabling high-performance of TCP on hosts are examined. The findings of this chapter are illustrated with measurements made on the test-platform.

5.1 Window Size

5.1.1 Receiver Window Limitation

As discussed in [26], an upper bound for TCP throughput is given by

$$\text{max throughput} = \frac{\text{receive window size}}{\text{round trip time}} \quad (5.1)$$

This follows from the window-based flow control: TCP only buffers a receive window size (*recvwin*) worth of data in the network, even though the long fat pipe may hold much more.

Standard TCP allocates a 16 bit field to advertise the receive window size. Thus, when using the Standard TCP maximum *recvwin* of 64kB over a typical dial-up satellite channel ($RTT \approx 600ms$), the upper bound on throughput is

$$\begin{aligned} \max \text{ throughput} &\approx \frac{65535 \text{ bytes}}{600 \text{ ms}} = 109225 \text{ bytes/sec} \\ &\approx 106.7 \text{ kB/s} \approx 853 \text{ kbps} \end{aligned} \quad (5.2)$$

This bound is independent of the effects of asymmetry.

Unfortunately, a comment in the Linux source code points out that windows larger than 32767 bytes should not be used, since certain other implementations are buggy and use signed arithmetic during window calculations. Hence, the actual safe maximum throughput for a connection with $RTT = 600ms$ is actually about half that given by equation 5.2.

5.1.2 Increasing the Window Size

A new TCP extension, defined in RFC1323 [1, 15, 27], allows the two ends of a TCP connection to negotiate larger window sizes. This is done by sending a window scaling option during the SYN segments. If both ends indicate a willingness to honour this option, the factor specified is then used to scale the 16 bit window size in the normal TCP header.

This allows window sizes up to 1GB.

Most new TCP stacks support RFC1323. This includes Linux 2.2, Windows 98 and Windows NT 4.0.

To enable the use of large windows, the following must be done:

1. RFC1323 window scaling must be enabled on both server (proxy) and client.

2. The client application software (e.g. web browser) receive socket buffer size must be set to *bwdp* (described below). Most applications do not support this; in this case the client's default receive socket buffer size should be set to *bwdp*. This value sets the largest TCP receive window (*recvwin*) that the client will advertise. The receive window is the amount of data the client would like to be in flight.
3. The proxy server socket buffer size should be set to *bwdp* (described below). If the software does not support this, the server's default transmission socket buffer size should be set to *bwdp*. This value sets the maximum amount of data that TCP can have in flight at any time (i.e. that it can transmit during a window), even if the advertised *recvwin* is larger.

A list of RFC1323-compliant operating systems, as well as their procedures required to utilize large windows, is maintained on the performance-tuning list at [28].

Choice of Socket Buffer Sizes

The send and receive socket buffer sizes should be set to the amount of data that is buffered in the network itself. This is the *bandwidth * delay* product. The delay of the network can be measured using *ping* or *traceroute*, and is equal to the RTT.

The bandwidth refers to the maximum available forward transfer bandwidth for a particular system. This is not necessarily equal to the forward link bandwidth. It will be shown later that asymmetry places a fundamental limit on the forward throughput: this is then the bandwidth that should be used for the calculation.

For example, as will be seen in section 5.2.1, a client using a 9600 baud return link may have its total forward throughput limited to 304 kB/s, even though the satellite forward link may have a bandwidth of 54 Mbps. In this case, the bandwidth to be used for this calculation would be 304 kB/s.

Thus, the *bandwidth * delay* product (*bwdp*) is given by:

$$bwdp \approx RTT * (\text{maximum obtainable forward throughput}) \quad (5.3)$$

The receive socket buffer size should not be set too large, otherwise the connection may be overflowed, inducing congestion. This will be examined in section 7.1.1. The recommended limit is

$$\begin{aligned} \text{recvwin} &\leq RTT * (\text{maximum obtainable forward throughput}) \\ &\leq \text{bdwp} \end{aligned} \quad (5.4)$$

Usable limits for the maximum obtainable forward throughput on the test platform will be determined by the asymmetry (for example, see table 5.1 on page 54).

Server Memory Implications of Large Windows

The sender socket buffer size can be used to control per-connection bandwidth allocation. However, that means a given proxy server will allocate that amount of memory for each TCP connection. The reason this is necessary, is that the TCP sender must buffer all in-flight data (i.e. which has not yet been acknowledged), since any in-flight data could be lost, requiring retransmission.

Hence, if many connections are made from various clients, a proxy server may run out of memory. This problem can be combatted in two ways:

- A proxy farm can be used, so that different requests are sent to various proxy machines. This is commonly done in the industry.
- Modifications can be made to the kernel to allow dynamic resizing of socket buffers. In this case, the proxy server need only have enough memory to buffer the whole satellite link (say 54Mbps * RTT). As additional connections are made, the socket buffers for in-progress connections are decreased as they empty.

It would then also be possible to allocate larger socket buffers to clients who pay for a preferential or guaranteed service.

5.2 Effects of Asymmetry

5.2.1 ACK Congestion

As explained in section 2.8.2, TCP is ACK-clocked. The rate at which the sender can inject new packets into the network is constrained by the rate at which it receives the ACKs.

In the asymmetric network, the forward transfer thus becomes limited by the rate at which back-to-back ACKs can travel over the slow serial return-path. TCP effectively adjusts itself to clock data at only the rate at which the slowest link can manage.¹

Although this illustrates one of the reasons for TCP's robustness, it does cause somewhat of a dilemma: Most bulk data transfers are unidirectional. It is thus unfortunate that the unidirectional flow should be limited by ACK packets which in essence carry no information other than establishing the reliability of the connection.

Consider a serial system with line-speed *baud* and channel efficiency η (which may be > 1 when link-compression is applied).

Let D be the bits-to-byte divisor ($D = 10$ for an 8N1 connection).

The nominal channel throughput (*ctp*) is thus

$$ctp = \eta \frac{baud}{D} \quad [\text{bytes/sec}]$$

which implies a byte transit time (*btt*) of

$$btt = \frac{D}{\eta \cdot baud} \quad [\text{sec}]$$

An ACK packet without accompanying data (as in the case of unidirectional transmission only) consists of the IP header *iph* and the TCP header *tcp*. Normally $iph = 20$,

¹Note that this is steady-state, not dynamic congestion control behaviour.

while $tcph = 20$ for standard ACKs without options, or $tcph = 32$ if timestamps are turned on.

The serially transmitted data consists of the packet plus the serial protocol framing bytes spf . For SLIP, $spf = 2$ or $spf = 1$, depending on implementation.

The total transit-time for an ACK (att) is thus

$$att = \frac{D}{\eta \cdot \text{baud}} (spf + iph + tcph) \quad [\text{sec}] \quad (5.5)$$

Let each ACK acknowledge N segments of size MSS . Typically, implementations use $N = 2$ in accordance with the Hosts Requirements RFC (see [1, 29]).

The throughput limitation (TP) on data in the forward path is thus

$$\begin{aligned} TP &= \frac{N \cdot MSS}{att} \\ &= \frac{N \cdot MSS \cdot \eta \cdot \text{baud}}{D[spf + iph + tcph]} \quad [\text{bytes/sec}] \end{aligned} \quad (5.6)$$

As an example, consider a 9600 baud SLIP ($spf = 2$) return path with an Ethernet forward path, as in the test system. Take a nominal value of $\eta = 1$.

With timestamps disabled, $tcph = 20$ and $MSS = 1460$, hence $TP = 65.2$ kB/s. With timestamps enabled, $tcph = 32$ and $MSS = 1448$, hence $TP = 50.3$ kB/s.

If various connections are transferring data simultaneously, the return ACKs have to share the serial path. Note that the ACKs in eq. 5.5 can come from arbitrary connections, hence the throughput limitation of eq. 5.6 refers to the *total throughput on all forward data connections*. Thus, application-level modifications which split a TCP transfer into n simultaneous connections (as proposed in [9]) will not be able to overcome the maximum throughput set by eq. 5.6.

Solutions to this problem are studied in section 5.3.

5.2.2 Undersized MSS

As discussed in section 2.7, the client TCP will use the MSS-option during the three-way handshake, to inform the server of the MSS it is willing to receive. The problem is that the client must specify the MSS that the sender must transmit via satellite without knowledge that the data will be received via satellite.

TCP implementations typically examine the MTU of the outgoing route (in this case the serial link), assume that this will also be the incoming route, and use this to calculate the MSS it wishes to receive.

Many satellite receiver cards are based on Ethernet cards, and hence feature MTUs of around 1500 bytes. In comparison, serial interfaces have been found to perform better with MTUs of 576 or 296, and most modern operating systems set them as such.

Hence, if the client serial interface MTU is set to 576 (say), it will advertise an MSS of 536 to the satellite proxy server, even though the satellite interface could manage an MSS of around 1460. From equation 5.6, the maximum throughput limit is directly proportional to the MSS — thus it is clearly undesirable to run at an MSS lower than that achievable by the satellite interface.

Solutions to this problem are studied in section 5.4.

5.3 Improving ACK Congestion

This section studies solutions to the problems detailed in section 5.2.1. The asymmetric throughput limit is governed by equation 5.6.

5.3.1 ACK Suppression

Since TCP ACKs are cumulative, the asymmetric throughput limit can be improved by suppressing the transmission of some ACKs. Each ACK will then effectively acknowledge more sender packets, thereby increasing N in eq. 5.6. This has been studied in [30].

However, ACKs also function as other signals to TCP. For example, three successive duplicate ACKs signal the TCP sender to enter fast retransmit / fast recovery. Furthermore, during slow start, each ACK increases the amount of data sent. ACK suppression during slow start thus decreases performance since many Internet connections are short-lived and hence constrained by the *cwnd*.

Avoiding these problems would entail keeping track of the state of individual TCP connections - in general this is undesirable since it increases required CPU time.

Furthermore, larger ACK intervals (N) can lead to increased transmission bursts on the forward satellite network, which may decrease network stability. The conclusions in [29,31] is thus that ACK suppression is unwise.

A result from [30] suggests that increasing ACK intervals causes the RTT estimation to be inflated artificially. This causes the sender to take longer to recognize lost packets.

5.3.2 Header Compression

Quite a few fields in the TCP/IP headers remain constant from packet to packet, or change by small amounts. Using a differential encoding scheme, it is thus possible to greatly reduce this overhead.

VJ header compression [21] is widely used to improve performance on serial links, and is available as part of most TCP/IP implementations.² It typically compresses the TCP/IP headers to between 5 and 7 bytes, depending on how much information changes between headers. Use of header compression thus implies that the term $iph + tcph$ in eq. 5.6 be replaced by 7 (worst case).

Throughput is thus improved by a factor

$$\frac{spf + iph + tcph}{spf + 7} \approx 4.7 \quad (5.7)$$

for $spf = 2$ and $iph = tcph = 20$.

²Which fact makes its use very attractive.

Such an improvement is obtained since a pure ACK carries no data, consisting of only a TCP/IP header.

Herein lies a problem. Most ISPs budget their backbone to provide sufficient service for standard dial-up users. Since the dial-up forward throughput is low, the return ACK rate will be much lower than that required for satellite.

The bandwidth occupied on the backbone by the ACKs can be estimated as:

$$\begin{aligned} ACK\ bw &= (ACK\ size) * (ACKs\ per\ sec) \\ &= [iph + tcph] \frac{total\ forward\ throughput}{N.MSS} \end{aligned} \quad (5.8)$$

A 56kbps dial-up client (with MSS=536) will require about 209 Bps backbone-bandwidth for the returning ACKs.

A satellite user (MSS=1460) with a 9600 baud return link could sustain a forward throughput of 304 kBps using VJ header compression. This would require about 4264 Bps dial-up backbone-bandwidth for the returning ACKs. Because of the header compression, the bandwidth required on the backbone is much higher than that required on the serial link.

Clearly, standard dial-up ISPs who have not specifically budgeted for this bandwidth, would not be able to sustain it to multiple clients for any length of time. Hence, if a satellite provider wishes to use header compression to obtain improvement from the asymmetric limit, it should provide its own dial-up infrastructure.

As will be seen in Chapter 6, VJ header compression may cause performance problems when packets are dropped.

VJ header compression is intended for point-to-point links. This will cause complications when the dial-up interface IP address differs from the satellite interface IP (see section 3.3).

5.3.3 ACK Compaction

This new technique, proposed in [13,31], functions on a similar principle to ACK suppression by deleting ACKs from the return queue. However, information is appended to the remaining ACKs which allows a remote decompressor to *statelessly* reconstruct the original ACK stream (either exactly or approximately, depending on implementation). This requires very little overhead.

The decompressor uses the remaining ACK as a ‘master’ ACK, and modifies the appropriate header fields using the appended information. The decompressor then transmits the set of ACKs as if they originated from the client.

Consequently, sending 10 ACKs (say) is not much more expensive than sending only a single ACK. This has ramifications for the work discussed in Chapter 6 and 7.

The compactor *tunnels* the remaining ACKs and appended information to the decompressor, which resides at the satellite provider. The satellite provider can thus budget its own backbone for the bandwidth required to support the total ACK stream. The dial-up infrastructure only has to carry the packets to the decompressor. This requires only the capacity of the serial link, and hence does not place such a great strain on the dial-up network backbone. The problem from the previous section is thus avoided.

ACK compaction is discussed further in Appendix A.

5.3.4 Comments on Timestamps and SACKs

RFC1323 [15] also introduced the use of timestamps. Each transmitted data segment contains a TCP option carrying a timestamp. The ACK for that segment then contains a copy of that timestamp in a TCP option.

This serves two purposes:

- It allows for more accurate RTT estimation at the sender. Standard BSD-derived implementations often use only a single packet per window for the RTT estimate (hence a danger of aliasing exists), and may measure with coarse granularity.

- It protects the TCP connection against the wrapping of sequence numbers (PAWS).

However:

- [32] shows that it is the minimum RTO that has the most influence on the performance of the RTT estimator, and calls into question the need for timestamps to improve it.³
- According to [4], sequence number wrapping is not a factor for satellite links under about 100Mbps, although it is definitely required for links over 286Mbps.

It can thus be argued that timestamps can be turned off without negatively impacting on robustness, or, indeed, removing extra possible performance.

This is important, since VJ header compression [21] cannot compress headers which contain timestamps. ACK compaction could, in theory, be adapted to support timestamps, but at the cost of additional overhead.

Similarly, the presence of SACKs disables VJ header compression. SACKs play an important role via satellite [27] and should be used. Fortunately, SACKs mostly occur on duplicate ACKs, which (unfortunately) already act to disable VJ header compression in any case (refer to Chapter 6).

Since SACKs are advisory in nature [16], it does not matter if ACK decompaction reconstructs all ACKs with the SACK information contained in the ‘master’ ACK. To avoid missing any SACK blocks, the ACK compactor should examine all ACKs it deletes and include as many of their SACK blocks as possible in the remaining ‘master’ ACK.

5.4 Obtaining the Maximum MSS

This section studies methods of circumventing the problem detailed in section 5.2.2.

³For example, the Linux operating system measures all segment RTTs (except retransmission, according to Karn’s algorithm [1]) to high accuracy without requiring the timestamps option.

5.4.1 Setting Serial Interface MTU = Satellite Interface MTU

The MTU of the outgoing serial interface can be made the same as the MTU of the incoming satellite interface. There is, however, no guarantee that this MTU will be optimal for both paths.

Large MTUs on the serial path may cause problems if a forward transfer is taking place simultaneously with a reverse transfer (such as the client sending out e-mail). Consider that an outgoing packet of 1500 bytes will take 1.6 seconds to transmit over a 9600 baud link, delaying any ACKs associated with forward transfer streams. This in turn may trigger congestion control algorithms, which drastically reduces satellite performance.

5.4.2 Use a Tunneling Network Device

Because of the proxy server network architecture, the client will receive the enhanced satellite TCP connections from certain *known* IP addresses.

It is thus possible to create a 'dummy' network device, and modify the client's routing table so that traffic to the proxy-server is routed through this device. The MTU of this device is then set to the MTU of the satellite interface (say 1500). This device then forwards the packets through the normal serial interface, which can have its MTU set to 576 (say).

Hence, the enhanced TCP flows will receive an advertised MSS of 1460 (say), while outgoing traffic such as e-mail be subjected to a more optimal serial MTU of 576 (say).

Of course, the proxy requests routed through this dummy device could be as large as the MTU, hence the dummy device must be able to fragment its packets to be able to fit over the standard serial interface. The dummy interface would typically create an IP tunnel to a remote reassembler to achieve this.

Although this means additional (stand-alone) software at both client and at the satellite provider, it does tie in nicely with the use of ACK compaction. Two problems can thus be solved simultaneously.

5.4.3 Standardize True MSS Negotiation

As pointed out by Stevens [1], many people incorrectly refer to the MSS as being a negotiated option. It is not. Each end simply advises the other of the maximum segment size it is willing to receive. The other end may send less per segment, but not more.

It would be possible to standardize actual MSS *negotiation*, e.g. by introducing additional options on SYN segments so that each end not only advertises the MSS it is willing to receive, but also the MTU of the outgoing interface. The client could then use the last part of the handshake to readjust the MSS previously advertised. Together with path MTU discovery, the sender could then settle on a more optimal MSS.

This solution requires low-level modifications to the TCP implementation at both sender and receiver, as well as discussion in the TCP standards forums.

5.4.4 Modify Default MSS in Absence of MSS Announcement

Another solution is to turn off MSS announcement by the receiver. In absence of an MSS announcement, most implementations (e.g. BSD) assume an MSS of 536 bytes. This default value which the sender assumes can be changed to the MSS suitable to the satellite path.

Although this is (currently) completely non-standard, modern TCP receivers should be able to handle these MSS sizes without advertising them.

5.5 Measurements

5.5.1 Validations

The validity of eq. 5.5 has been demonstrated in figure 4.2, as discussed in section 4.4.

Figure 5.1 illustrates the validity of equations 5.1 and 5.6 for a $recvwin = 32\text{KB}$ and $MSS = 1448$. Timestamps were turned on and the MTU of the serial interface set to

Table 5.1: Typical example asymmetric throughput limits

MSS = 1460, timestamps disabled, $spf = 2$				
Serial speed [bps]	Standard		Header Compression	
	[kB/s]	[Mbps]	[kB/s]	[Mbps]
9600	65.2	0.51	304.2	2.38
14400	97.8	0.76	456.3	3.56
28800	195.5	1.53	912.5	7.13
33600	228.1	1.78	1064.6	8.32
64000	434.5	3.39	2027.8	15.84

1500.

Trip times were measured using *ping* (in the absence of other forward traffic, so as not to perturb the transit times), with throughput measured by *ttcp* using a packet size of 1448 bytes.

The measured values are always slightly lower than theoretical, since the client can advertise receive windows smaller than the maximum during the transmission.

It is interesting to note that the greatest discrepancy arises at the serial throughput limit. This is where ACKs are being transmitted back-to-back over the serial link. Hence the ACK-stream becomes more sensitive to variations in processing time at the client and sender, and especially to the process-switching granularity of the delay-software.

5.5.2 Throughput Improvement

The following experiment demonstrates the use of VJ header compression to optimize the return link. As a proof of concept of the architecture of Chapter 3, the server *satlink1* ran the *squid* proxy-server for this experiment.

Timestamps are disabled and MSS = 1460, obtained by setting the serial MTU to 1500. The delay is set to 1000ms, and the server and client socket buffers (and hence *recvwin*) are set to 256kB.

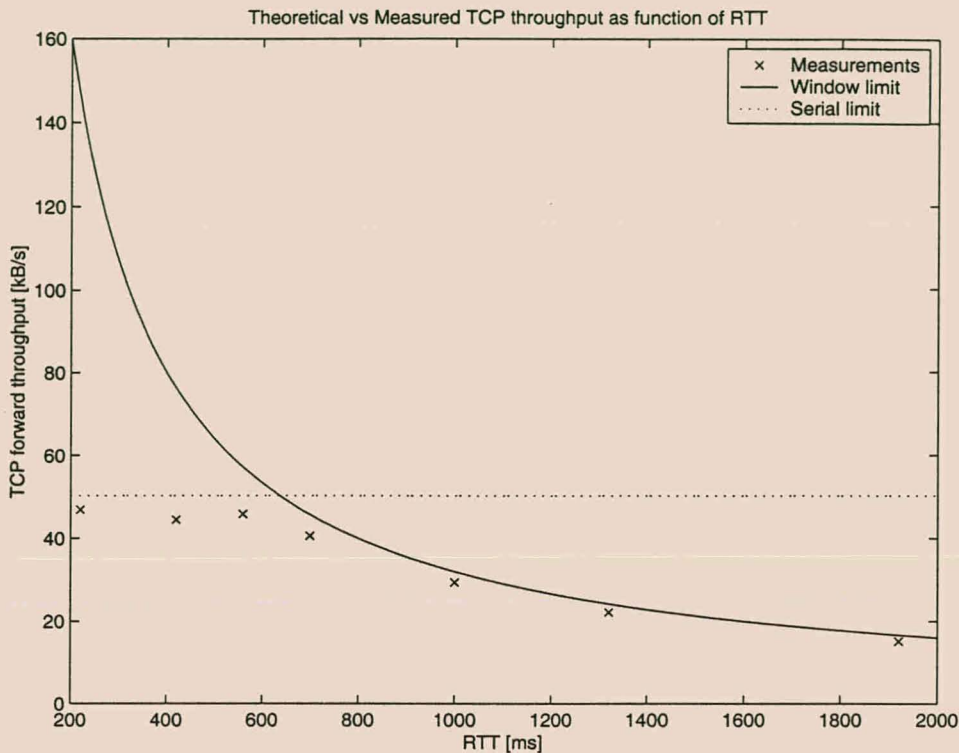


Figure 5.1: Example TCP throughput vs RTT

By tracing the FTP transfer of a 14MB file, the *time-sequence-number* graph of figure 5.2 is obtained (details on reading this type of graph can be found in Appendix B).

As seen from the figure, the connection spends an initial period in slow start, which is followed by smooth high-throughput transmission.

Some statistics for this transmission are given in table 5.2.

Although table 5.1 predicts a throughput limit of 304 kB/s for this experimental configuration, it was found in practice that transfers above 256 kB/s tended to be slightly less stable.

This is the same effect noted in sections 4.4 and 5.5.1: Variations in processing delay, especially the context-switching granularity of the serial delay process, cause back-to-back data to traverse the serial link less ‘smoothly.’

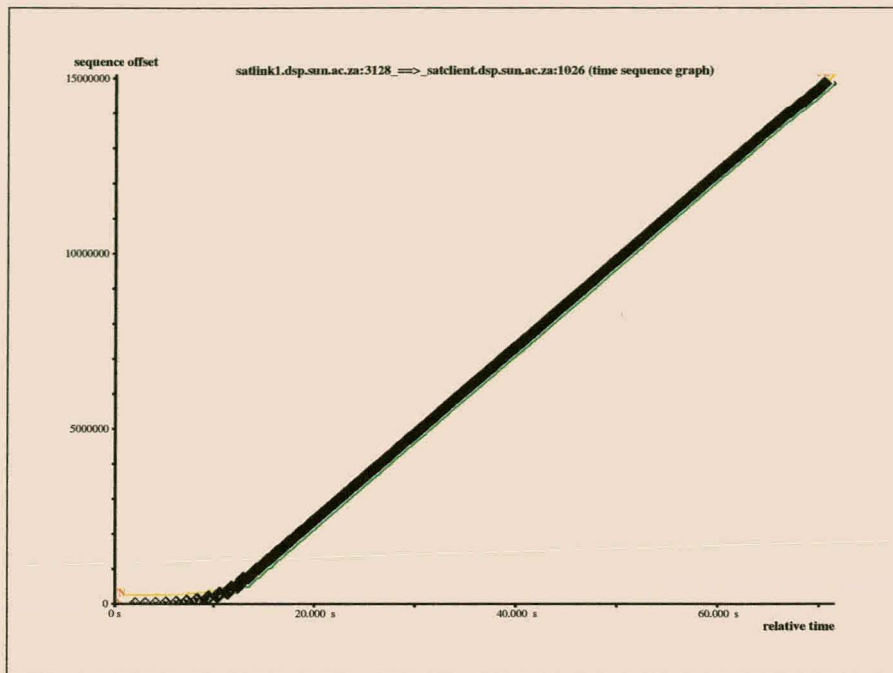


Figure 5.2: Example improved TCP session

Table 5.2: Statistics for improved steady-state throughput experiment

File size	14 850 814 bytes
Delay setting	1000ms
Total average session throughput (including slow start)	202.6 kBps
Throughput during steady-state	254 kBps = 1.98 Mbps
Average advertised <i>recvwin</i>	260 002 bytes
Slow-start time	\approx 13.5 sec

5.6 Conclusions

This chapter has examined the factors limiting steady-state forward TCP throughput. In addition to the known *recvwin*-limitation, it was found that the asymmetric nature of the network also imposes a limit on forward throughput.

This chapter has derived and quantified (to first order) the nature of this limit.

Among other things, it was found that the asymmetric limit is directly proportional to the advertised MSS. Consequently many modern operating systems, which automatically set lower MTUs on serial interfaces, will cause under-performance on satellite links.

Table 5.1 (page 54) gives some asymmetric limits for the experimental system at popular serial bit-rates.

Methods of side-stepping the throughput limitations were studied. Using one such method (VJ header compression), a steady-state transfer at 254 kB/s was demonstrated with an $RTT \approx 1000ms$. By comparison, plain vanilla TCP (with 32kB *recvwin*) would only be able to obtain 32 kB/s under the same circumstances, a factor 7.9 less.

This chapter also noted some potential pitfalls to these methods. Consequently, this chapter points out regions of TCP standardization and design which should be re-investigated in future revisions.

Although this chapter illustrates that substantial steady-state throughput improvement is possible, it should not be studied in isolation. For example, it will be found that header compression interacts negatively with dropped packets, and that a system running close to the serial throughput limit (e.g. because of multiple connections) may be in danger of packet-dropping by overflowing the serial queue.

Chapter 6

Dynamics: Packet Loss

This chapter examines the dynamic performance of TCP connections via satellite, in the event that packets are dropped because of corruption or rain-fading.

Although congestion control is important, it decreases performance over satellite networks [9]. Congestion control should not be disabled, since this could lead to large scale congestive collapse. As such, experimentation with congestion control should be done carefully with the knowledge that improvements may be obtained to single sessions, but may hold dire consequences for the rest of the network.

Unnecessary triggering of congestion control algorithms should be avoided. This is especially true of RTO, since it is followed by exponential slow start and linear growth (congestion avoidance) phases, both of which are very expensive over satellite networks due to the latencies involved.

Secondly, the serial queue length should be managed, in accordance with the spirit of [3], to decrease the underlying congestive scenario and to increase robustness.

It will be seen that an unfortunate interaction exists between packet loss, retransmission and the use of header compression. It is concluded that, although header compression greatly improves steady-state throughput, it can cause serious problems for packet loss dynamics.

6.1 The Experimental System

The experimental system for this chapter has been optimized for steady-state performance using VJ header compression over the 9600 baud link. The *recvwin* has been set to 250kB, and an RTT of 1 sec is used. MSS = 1460, obtained by setting the serial interface MTU to 1500.

The SACK options are enabled. As shown in [22,27], the use of SACKs give a distinct performance advantage over satellite links.

The experimental Linux system uses FACK for retransmissions. According to [27], FACK does not give as high a retransmission-throughput as SACK over satellite, but this is to be expected intuitively, since FACK is less aggressive. It is, however, not so trivial to disable FACK in the kernel in favour of SACK.¹

6.2 Packet Loss on the Reverse Link

There are two possible cases:

- Packets are lost on the serial (dial-up) link.
- Packets are lost in the Internet between the dial-up router and the proxy server.

6.2.1 Packet Loss on Dial-Up Link

Although most modern dial-up links can and should use error-correction, packets may still be corrupted from time to time.

The VJ header compression system is differential. If a packet is lost on the dial-up network, the VJ decompressor will lose synchronization with the compressor, and will hence reconstruct incorrect packets of which the checksums are invalid. ACKs will thus simply cease to arrive at the TCP sender.

¹New research [9] suggests that the real-world behaviour of FACK is much closer to the theoretical maximum for TCP than either Standard TCP (Reno) or SACK.

The VJ mechanism to resynchronize the compressor and decompressor [21] relies on TCP doing a retransmission. Hence, if a single packet is dropped or corrupted on the dial-up link, the sender will receive no ACKs and will be forced to an RTO. The retransmission will be detected by the compressor, which will then send the decompressor resynchronization information.

This induced RTO is unwanted, since the underlying reason is not true congestion but rather an idiosyncrasy of VJ header compression. This is an example of an RTO which will be tolerated in low-latency terrestrial networks, but is unacceptable in satellite networks because of the high expense involved in recovering from subsequent slow start / congestion avoidance.

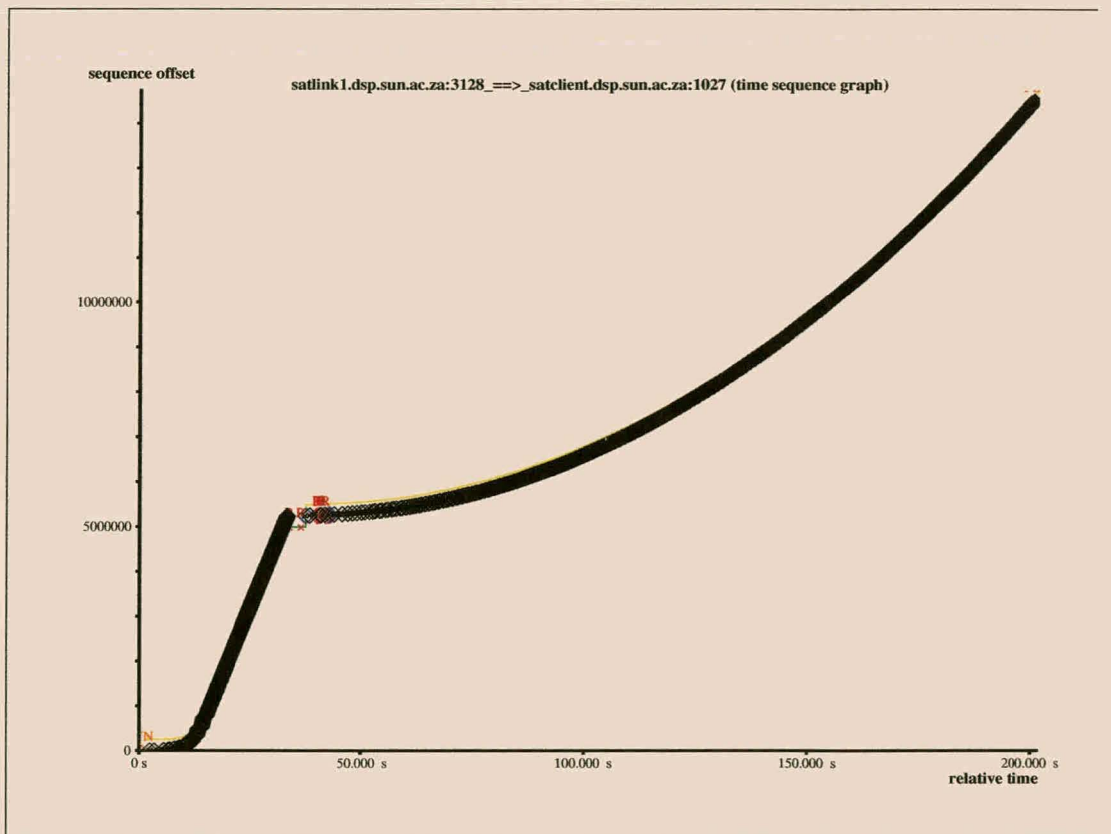


Figure 6.1: Effect of outage on VJ header-compressed dial-up link

Figure 6.1 demonstrates the effect of a link outage between the VJ compressor and decompressor. The outage is emulated by momentarily interrupting the serial-delay

program on the emulator. To illustrate the effect, this is done once the system is running at full steady-state throughput.

It is clearly seen that the sender reverts to slow start, and transits to linear growth at a much lower throughput rate (*ssthresh*) than previously, thereby greatly decreasing average throughput.

6.2.2 Packet Loss on rest of Return Network

When ACKs are lost on the return path in the rest of the Internet, this may or may not lead to RTO, depending on the sender's current RTO estimate.

However, under the assumption that the standard Internet drops or corrupts packets with probability $\ll 1\%$, loss of ACKs are indicative of congestion. According to the spirit of TCP, an induced RTO under such circumstances would be correct.

6.3 Long Outage on Forward Channel

During transmission, the client receives packets from the long fat pipe, and generates ACKs. These ACKs cause the sender to transmit additional packets into the pipe.

If the forward signal is dropped, the ACKs will momentarily not be generated, and hence the sender will not transmit more data. If the forward signal is dropped for long enough (typically 1 RTT or the window transmission time) so that the entire in-flight pipe drains, the ACK clock will break completely — once the pipe drains, no data remain to generate ACKs to trigger, in turn, the transmission of new data.

The only way to restart transmission, and the ACK clock, is to wait for an RTO. Since a long outage is indicative of changing network conditions, and hence a changing congestive environment, the induced RTO and slow start would be justified and required. According to the spirit of TCP, long periods of inactivity should be followed by a probe of network capacity (i.e. slow start) [33], since the congestive environment has effectively become unknown again.

6.4 Single and Multiple Dropped Packets

In this section, forward link outages are examined which are shorter than those of the previous section, and hence should preferably *not* give rise to RTO.

The packet drop is simulated by unplugging the forward Ethernet cable for a time less than the RTT.

6.4.1 Exploding Serial Queues

When an incoming packet is dropped, the following interaction between congestion control and header compression is observed:

Since the receiver now receives out-of-order packets, it will start generating duplicate ACKs for every new incoming segment (of which there is a pipe-full). Furthermore, it will ACK *every* new segment (not every second one) [33]. Hence the ACK-rate will double. Thirdly, duplicate ACKs invoke the header compression error-correction system, which turns the header compression off. SACKs may also be present, which will disable the header compression.

Effectively, N in eq. 5.6 changes from $N = 2$ to $N = 1$, while the throughput increase by header compression (eq. 5.7) is lost. Using these two equations, it is seen that if the serial path had been running at full capacity during steady-state, the return bandwidth has suddenly effectively dropped by a factor 9.4.

The serial transport time thus suddenly increases radically, placing great strain on the outgoing serial queue. In the steady-state case, packets enter the serial queue at the same rate that packets leave it. However, with the sudden drop in effective serial bandwidth, ACK packets are now placed on the queue much faster than they can leave.

Even if the sender retransmits the missing packets immediately, the client will still receive the whole in-flight window of data, all of which will trigger the duplicate ACKs.

For a single TCP connection, the serial queue length can be approximated to first order as follows: Assume that no packets can leave the queue during the time in which the (duplicate) ACKs for the in-flight data are dumped into the queue.

$$\begin{aligned} \text{max serial qlen} &\approx \text{num packets in in-flight data} \\ &\approx \frac{\text{recvwin}}{\text{MSS}} \quad [\text{packets}] \end{aligned} \quad (6.1)$$

For the experimental system of this chapter, with an MSS of 1460 bytes and $\text{recvwin} = 250$ kB, this equation predicts a maximum serial queue length of 176 packets.

By making a small modification to the Linux kernel, it is possible to track the maximum serial queue length. During normal steady-state behaviour, a maximum queue length of about 5 packets is observed.²

However, when forward packet loss is induced, the serial queue grows to between 100 and 155 packets, depending on the exact values of recvwin the client advertised during the previous RTT.

Note that if multiple TCP connections to the client are active, and all drop forward packets during the same time-frame (e.g. because of rain fading), eq. 6.1 is inaccurate. The queue length must be calculated over all the possible in-flight data of all the connections:

$$\text{max serial qlen} \approx \frac{\text{RTT} \cdot (\text{serial throughput limit})}{\text{MSS}} \quad [\text{packets}] \quad (6.2)$$

This equation is valid provided that the forward throughput is not constrained to a lower value by techniques such as throttling. *In that case, that maximum forward limit should be substituted for the serial limit.*

Since the test system can sustain 304 kB/s of throughput, this translates to a maximum queue length of 214 packets (for an RTT of 1 sec), irrespective of the number of connections.

(Note that other factors which influence the serial queue will be discussed in Chapter 7).

²This arises from timing variations dependent on how information changes between headers, i.e. on how effective header compression is on individual packets.

Another problem is that if one connection suffers dropped packets, the resulting change in effective serial bandwidth will affect all other connections — even though ACKs from those other connections may still be header-compressed, they have to wait in the serial queue while the uncompressed ACKs are slowly transmitted.

As soon as normal retransmission restarts, header compression will turn on and the problem will disappear.

During this period of decreased effective serial bandwidth, two options are available:

- Limit the queue to its standard size and drop the extra generated ACKs
- Allow the queue to grow as needed

6.4.2 Limiting Queue Length

The standard serial transmission queue length under Linux is 10 packets. The queue is drop-tail, in other words, if the queue already contains 10 packets, any additional packets are dropped until such time as the queue has space to accept more packets.

It should be noted that, in the case of one connection going into duplicate ACKs (hence growing the queue), packets from outgoing traffic (such as e-mail being sent) will also be dropped, as well as ACKs for concurrent incoming connections.

If two connections suffer RTO at similar times, both could enter slow-start during the same time-frame, which leads to the problem discussed in section 7.1.1.

Two variations are studied here:

- Using sender default minimum RTO
- Increasing sender minimum RTO

Default Minimum RTO

Because of the increase in transmission times for the ACKs in the queue, and the loss of the ACKs dropped off the end of the queue, the sender will suffer RTOs.

In the following experiment (figure 6.2), the standard Linux setup was used (i.e. minimum RTO of 200ms, serial queue length limit of 10 packets).

Packet loss causes the RTO timer to expire, hence retransmissions are triggered by RTO. The connection thus enters into slow start followed by linear congestion avoidance. Because of the reduction in *ssthresh*, transition to linear growth takes place at low throughput, leading to substantially degraded average throughput.

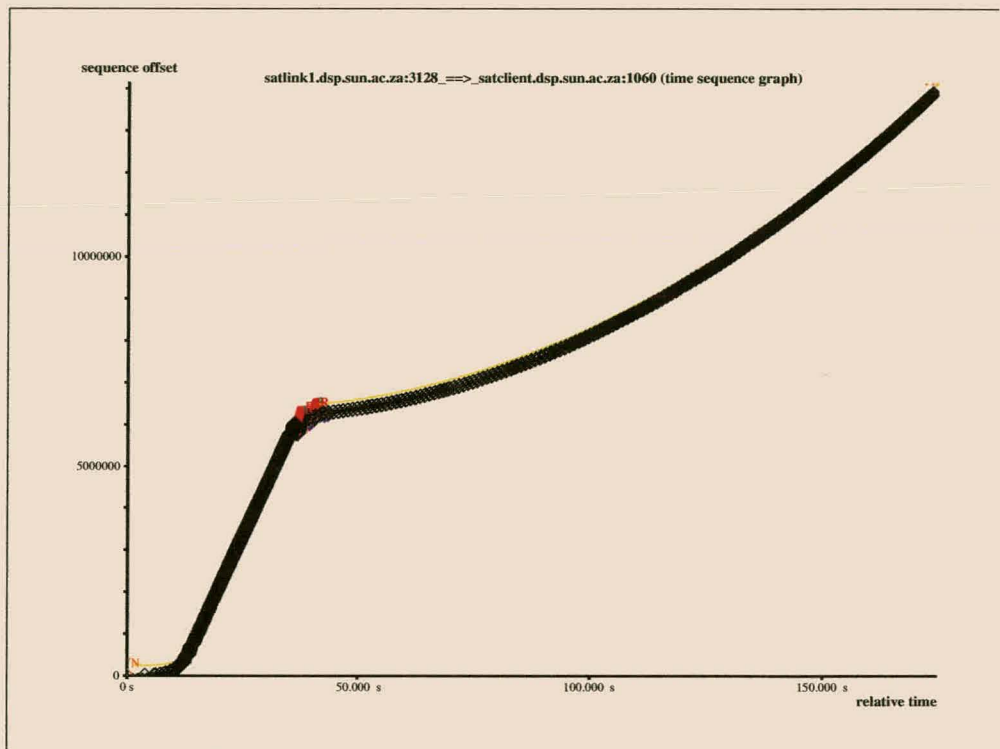


Figure 6.2: Overwhelmed queue leading to RTO

Increasing Minimum RTO

As discussed in [32], the minimum RTO dominates the performance of the RTO estimator. In an attempt to avoid the RTO encountered in the previous section, the minimum RTO can be increased at the sender. Conceptually, this may provide the opportunity for the information in the dropped ACKs to be replaced by the later ACKs, thereby allowing the sender to repair the damage with fast-retransmit, rather than RTO.

Once the out-of-order packets arrive, duplicate ACKs are queued at the serial interface, causing the effective serial throughput rate to decrease. The queue will thus start growing. Depending on the number of concurrent connections and the queue size, a varying number of the duplicate ACKs will thus be lost.

Assume that the queue is sufficiently long for at least three duplicate ACKs to reach the sender.³ The sender will use fast-retransmit and FACK to re-send the missing packets. However, since many of the duplicate ACKs have been dropped at the serial queue, the fast recovery will not transmit much new data from the new *cwnd* (since no ACK clock is being received).

When the client receives the missing packets, it will now send a cumulative ACK for *all* the packets received to date (including those packets for which the duplicate ACKs have been dropped). This will cause a jump in the position of *cwnd*⁴, so that the most of *cwnd* now covers unsent data. The sender will thus burst most of the *cwnd* onto the network.

Although it is assumed that this burst will not overwhelm the satellite network (because of the forward feedback congestion control assumption), it will indeed overwhelm the serial queue again. Hence most of the ACKs for the bursted *cwnd* will be lost, until such time as the queue has drained sufficiently to accept another packet. This ACK will in turn act cumulatively for all the packets sent to date, hence causing another jump in the position of *cwnd*, followed by another transmission burst.

The problem repeats in a series of gradually receding bursts. An example is shown in figures 6.3 and 6.4⁵, where the sender minimum RTO was set to 20 seconds.

Clearly, this situation is not very stable and will play havoc with concurrent TCP connections and outgoing data.

It is also possible for the burst-overwhelmed serial queue to drop sufficient packets to give rise to an RTO (despite the increased minimum RTO), with associated slow-start.

³Otherwise an RTO will occur.

⁴Assuming that the serial queue can accommodate this packet. If not, an RTO will occur.

⁵The apparent burstiness before the retransmissions is an artifact of the Ethernet retransmissions, caused by briefly removing the cable.

Such an example is shown in figures 6.5 and 6.6.

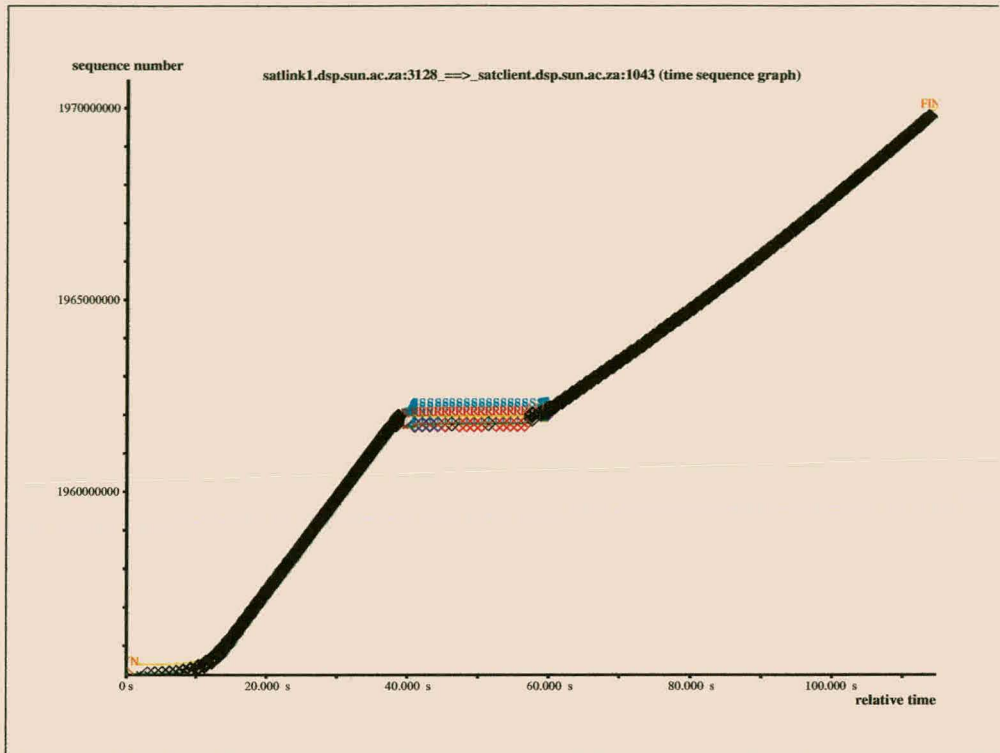


Figure 6.3: Overwhelmed queue leading to burstiness without RTO

6.4.3 Allowing Queue to Grow

If the serial queue is allowed to grow to queue all the ACKs from the in-flight data (as expressed in eq. 6.1 and 6.2), the sudden increase in serial transmission time is guaranteed to trigger an RTO at the sender.

To avoid this, the sender minimum RTO has to be set to a time longer than that required to clear the serial queue.

The queue clearing time can be estimated by combining eq. 6.2 and eq. 5.5, taking into account the space required for the SACK information:

$$qtime = \frac{RTT.D.(throughput\ limit)}{MSS.\eta.baud} (spf + iph + tcph) \quad [\text{sec}] \quad (6.3)$$

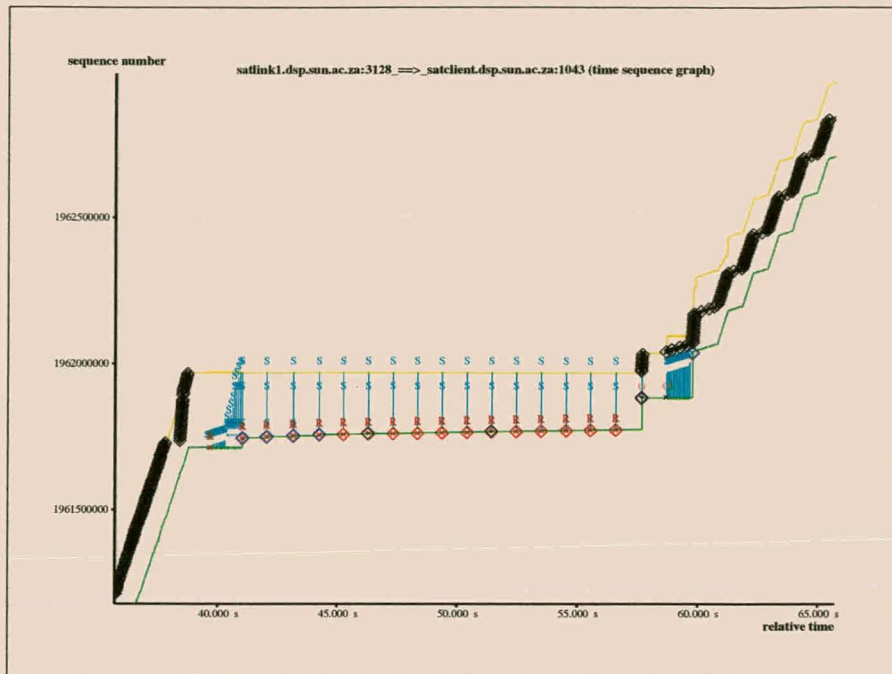


Figure 6.4: Close-up of transmission burstiness

The maximum TCP header size (including SACK blocks) is 60 bytes.

Thus, in the test system, to clear 214 packets at 9600 baud with $tcph = 60$ will take a maximum of 18.3 sec.

The idea thus is to allow the receiver to queue and send out all the duplicate ACKs (and SACK information), and to increase the sender minimum RTO to give the sender time to receive all of this information.⁶

The following experiment (figure 6.7) sets the sender minimum RTO to 20 seconds, and the serial queue length to 200 packets (to allow all 176 single connection packets to be queued).

When packet loss occurs, the receiver ACKs (with SACK blocks) the out-of-order packets arriving after the packet loss. Hence, retransmissions are triggered by the ACKs and SACKs, not by RTO. Therefore, the connection does not enter slow start, but continues at reduced throughput with linear growth (congestion avoidance).

⁶Other transfer streams will also be affected, since their packets will now become stuck in a long slow queue.

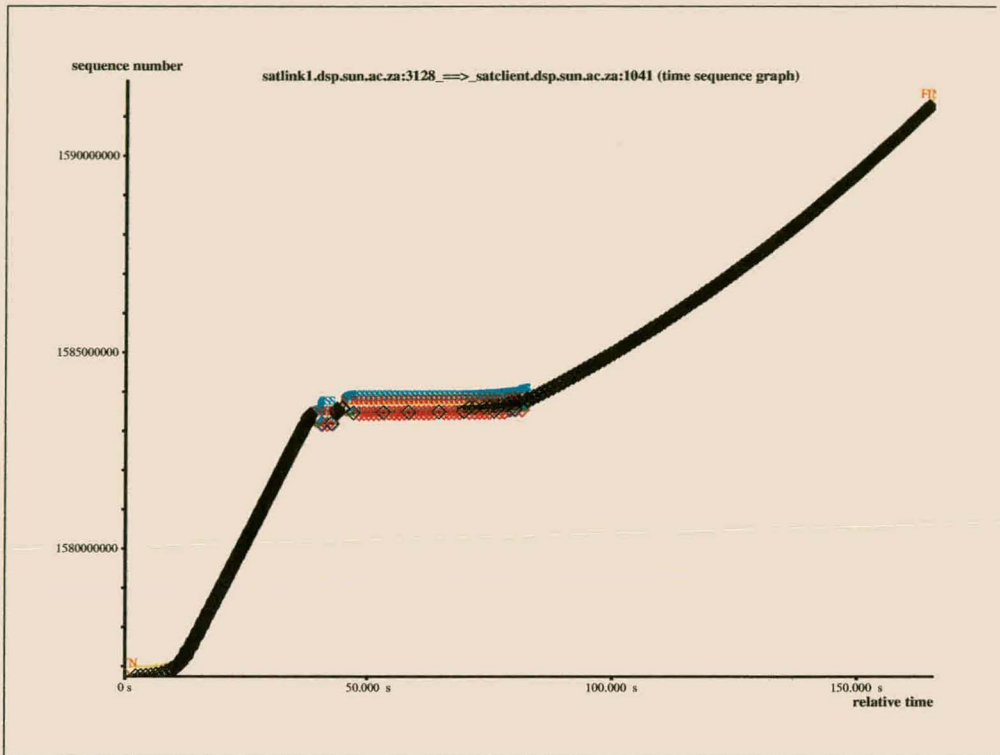


Figure 6.5: Overwhelmed queue leading to burstiness and RTO

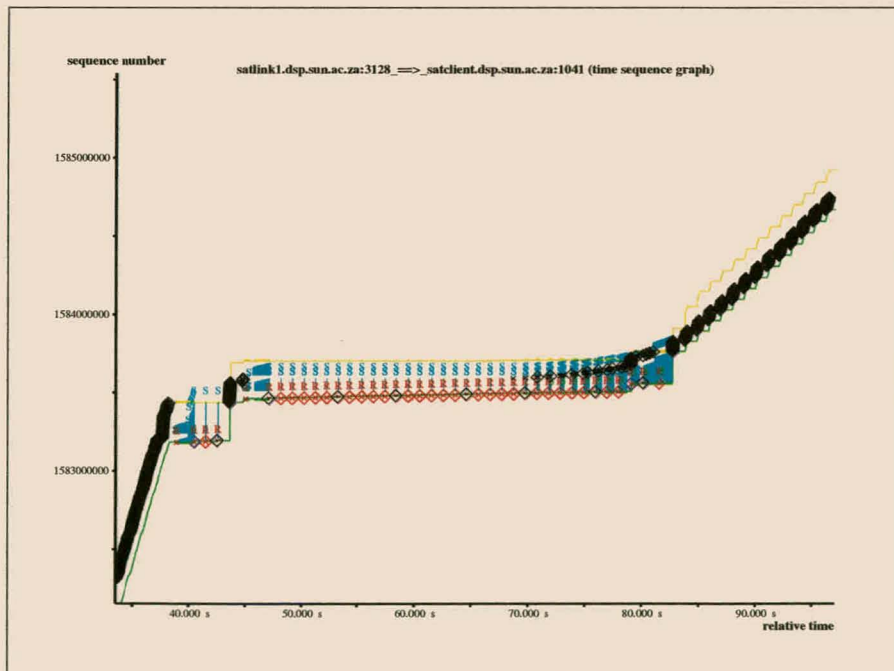


Figure 6.6: Close-up of burstiness leading to RTO

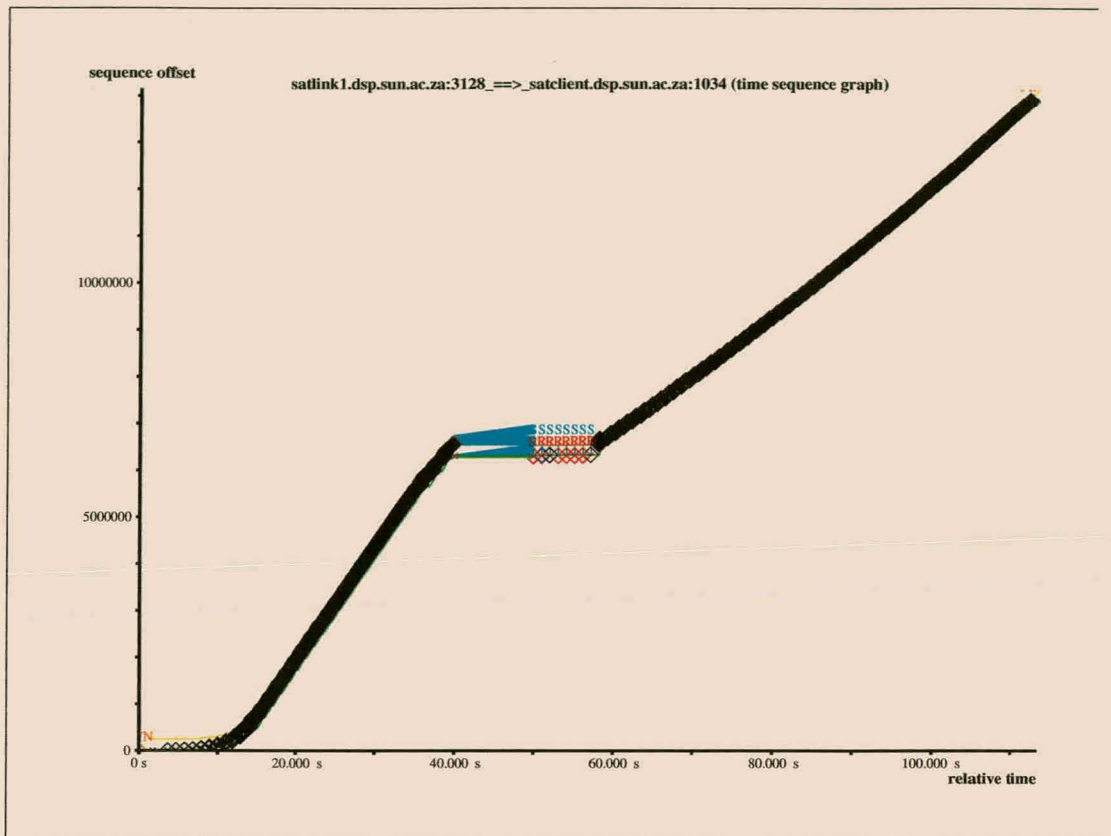


Figure 6.7: Allowing the serial queue to grow without RTO

It can be seen that a period of about 10 sec is spent on clearing the ACK queue, during which no new data is transmitted.⁷ This is followed by a time during which the FACK system transmits a single packet per RTT, until all lost packets have been retransmitted.

Note that a transmission burst also occurs in this case (close-up in figure 6.8). This arises from the design of fast recovery / fast retransmit when the transmission window is being kept close to its maximum size (i.e. to the *recvwin*), and has also been noted in [27]. However, in this case the burstiness does not overflow the serial queue, because the serial queue length is adequate.⁸

⁷This is less than that predicted by eq. 6.3 since the *recvwin* can be less than the maximum, and since SACK blocks do not necessarily extend the TCP header to its maximum of 60 bytes.

⁸The possibility that the burst may overflow the forward satellite queue is discounted here, under the assumption that the forward path is flow controlled.

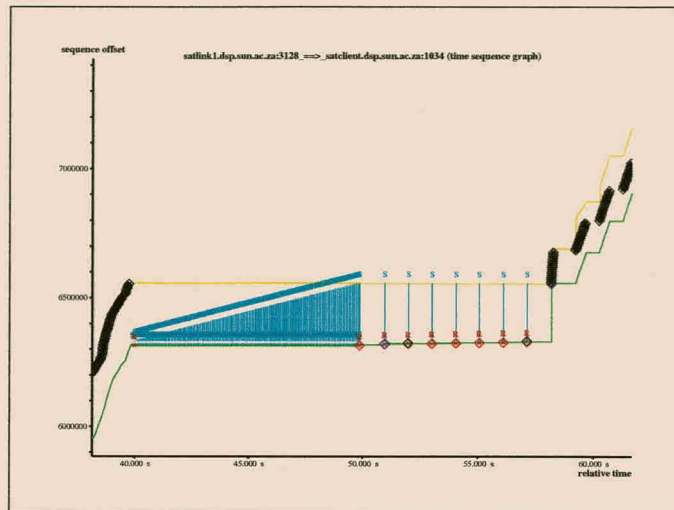


Figure 6.8: Burstiness after fast recovery

6.4.4 Comments

Since the limited queue length with increased minimum RTO leads to unpredictable behaviour and decreased robustness, it is not considered further.

By examining the growing queue case with increased minimum RTO, it is seen that the recovery period consists of two parts:

- Time to clear the queue of duplicate ACKs. This time is fixed and is about 10 seconds for the test system. It is estimated by using eq. 6.3 and depends on the serial baud-rate and *recvwin*.
- Time to retransmit lost packets. A single lost packet per RTT is repaired, thus the time varies. The fewer packets dropped, the less this time and the greater the advantage to using this method.

Best Average Throughput Performance

Measurements with a 12MB file suggests that use of a growing serial queue with increased minimum RTO mostly allows the file to be transferred in less time than RTO followed by slow start (typically 100 sec vs 170 sec). This of course depends on the

file size and the position of packet loss in the file. Specifically, the fewer the packets dropped, and the closer to the file beginning, the greater the advantage to using growing queues with increased RTO.⁹

Robustness: Changing Congestive Scenarios

This now causes a bit of a moral dilemma: After the packet loss is repaired, the connection resumes transmission at a high rate. However, this happens after a period of dormancy (at least 10 seconds in the examples). It is highly likely that network congestion conditions have changed during this time: [34,35] suggests that queuing on the Internet can change significantly over time-scales of 100 - 1000ms (with a wide range of behavior beyond this region).

It is well known that connections with lower RTTs are able to utilize more quickly transient extra bandwidth [9].¹⁰ Hence, in a normal dial-up environment, it is likely that other connections would have started to use the temporarily freed bandwidth.

In the case of rain fading, a substantial number of satellite connections to various clients (over a large geographical area) may have entered this dormant period. When these many connections suddenly resume retransmission at high rate, this could have a major impact on the dial-up network.

The basic dormancy period from eq. 6.3 could be decreased by increasing the serial speed and forfeiting the gain in steady-state throughput this could achieve by throttling the forward traffic. However, the dormancy may still be $> RTT$. According to RFC2581 [33] the network *should* be re-probed for capacity if the TCP has not sent data in an interval exceeding the default retransmission timeout (roughly equal to the RTT).

This method thus contravenes the robustness requirement of [2] for use in the global Internet. If a satellite provider wishes to deploy this method, it should provide its own dial-up infrastructure which is separate from the global Internet.

⁹Be aware, however, that techniques to increase slow start speed (see Chapter 7) may alter this.

¹⁰The lower the latency, the more rapid the changes during congestion avoidance

Increasing the minimum RTO substantially can also incur a performance penalty: sometimes (as seen in sections 6.2.1 and 6.3), an RTO is unavoidable to restart transmissions. A trade-off should thus be made between the desired minimum RTO, the serial baud-rate and a possible forward throttling limit (eq. 6.3). Incidentally, eq. 6.3 shows that choosing as large an incoming MSS as possible, also decreases the queue clearing time.

6.5 Conclusions

It is clear that, although VJ header compression improves steady-state throughput by increasing the asymmetric limit by a factor 4.7 (see eq. 5.7), it decreases the robustness of TCP by making it more sensitive to the effects of dropped packets.

The problem is partly that the steady-state performance requires the serial link to run at maximum capacity, hence leaving no margin to cope with the sudden doubling in ACK rate. This problem is exacerbated by the presence of SACKs and duplicate ACKs, which disable the header compression.

Because of the high latency, a large amount of data may be in flight, which increases the congestion. Subsequently, strain is placed on the serial queue, which in turn may lead to unnecessary RTO followed by expensive slow start and linear growth.

It is seen that rain fading (or single packet corruption) can unnecessarily trigger congestive control algorithms. This in turn may penalize other concurrent TCP connections (which are not experiencing congestion). Because of the doubling of the ACK rate, this may occur even if header compression is not used.

This chapter has illustrated some symptomatic treatments (lengthening the queue and increasing the minimum RTO) in an attempt to avoid these effects. However, as shown, these solutions require trade-offs to be made.

Clearly, the cleanest solution for these problems requires the return link to be able to handle the sudden doubling in ACK rate (and the appearance of duplicate ACKs and SACKs), yet be able to sustain the high ACK rate required for steady-state throughput. This is not true for VJ header compression.

A possible candidate is the new ACK compaction technique, by virtue of the fact that a variable number of ACKs can be compacted statelessly and non-differentially into a fundamental basic overhead. This also potentially side-steps the problem outlined in section 6.2.1.

Furthermore ACK compaction features a built-in serial queue management strategy. These issues are discussed further in Appendix A.

Chapter 7

Dynamics: TCP Slow Start

Slow start serves to establish the ACK clock and to probe the network for capacity, both at the beginning of a session and after an RTO or a long idle period [33].

Slow start presents various problems on satellite networks:

- Probing of network capacity can saturate the serial queue
- Long RTT lengthens slow start, decreasing responsiveness of small transfers

Short transfers (such as most web pages) take place almost entirely in the slow start domain, before the connection has reached steady-state. Hence, these short-lived connections may not actually benefit much from steady-state improvements

TCP start-up dynamics have been studied in detail in [6]. This chapter draws together the results from different disparate measurements to reach conclusions relevant to the satellite network.

7.1 Queue Management

As pointed out in Chapter 6, the situation can arise when more than one session is in slow start simultaneously.

Indeed, most web browsers will open a number of simultaneous sessions to retrieve various files and images associated with a web page ([36] suggests that for satellite links, a good number of simultaneous browser TCP sessions is 4).

7.1.1 The Problem

Section 5.2.1 shows that the serial system imposes a fundamental throughput limit on forward transfer. Of course, this capacity should be shared equally between all active TCP sessions. Similarly, if only a single session were active, it should use the full available capacity. Hence, the receive buffer sizes should be set so that a single connection will fill the available capacity.

This leads to the following problem: *ssthresh* (the point at which the TCP session changes from slow start to linear congestion avoidance) is initially set arbitrarily high, usually to the value of *recvwin*. The purpose of *ssthresh* is to reflect the capacity that the particular session should receive, but initially the sending TCP has no idea of the current link capacity. If N simultaneous connections start, *ssthresh* is thus initially typically set N times higher than it should be.

Each individual session is now increasing its *cwnd* exponentially, probing the capacity. When a session reaches its fair capacity limit¹, it should stop probing, but it doesn't since $cwnd < ssthresh$. Hence *cwnd* keeps on increasing exponentially.²

The serial queue cannot send its data out at faster than the capacity limit (equations 5.5, 5.6 and 5.7), but ACKs are entering the queue at higher rate since *cwnd* is still growing exponentially. Thus the queue grows and eventually overflows. This can cause slow start to recommence.

Although this seems strange, it is actually the standard behaviour of standard TCP:

- Capacity is probed by filling the queue until a packet is dropped

¹For example $\frac{1}{N}250kB/s$ for the test system considered so far

²A similar situation arises if only one TCP session is active, but the receive and sender buffer sizes are specified for a larger throughput than the serial link can actually support. E.g. in the test system if the buffer size is kept at 250kB, but the RTT decreased to 600ms.

- This indicates congestion
- which causes TCP to set a more appropriate *ssthresh* and slack off.

In effect, TCP thus has to induce congestion to measure network capacity to avoid congestion. This leads to many unnecessary retransmissions, which are expensive in a satellite network.

7.1.2 Solutions

A way is needed to estimate the connection bandwidth before packets are dropped, and thus to set the initial *ssthresh* to a more sensible, estimated level.

Such a solution is provided by TCP Vegas [24, 37–40]. Vegas effectively examines the spacing of returning ACKs to determine the bandwidth of the channel.

Vegas also does queue management during the linear growth phase, by comparing the actual throughput rate to the expected throughput (derived from RTT measurements). Vegas can thus decide if extra linear-growth packets should be transmitted into the network or not.

It has been demonstrated that Vegas is an effective, fair means of substantially decreasing unnecessary retransmissions and router queue lengths.

Standard TCP (Reno) is, of course, more aggressive than Vegas, and hence, if both Reno and Vegas are run concurrently over the same network, Reno tends to steal slightly more of the capacity. However, in the satellite proxy-based network, the provider can ensure that only Vegas is used for high-performance delivery.

7.1.3 Experiments

In the following experiment, two file transfers (13MB and 14MB) are started almost simultaneously. Figure 7.1 shows the time-sequence-number graph for the 14MB transfer.

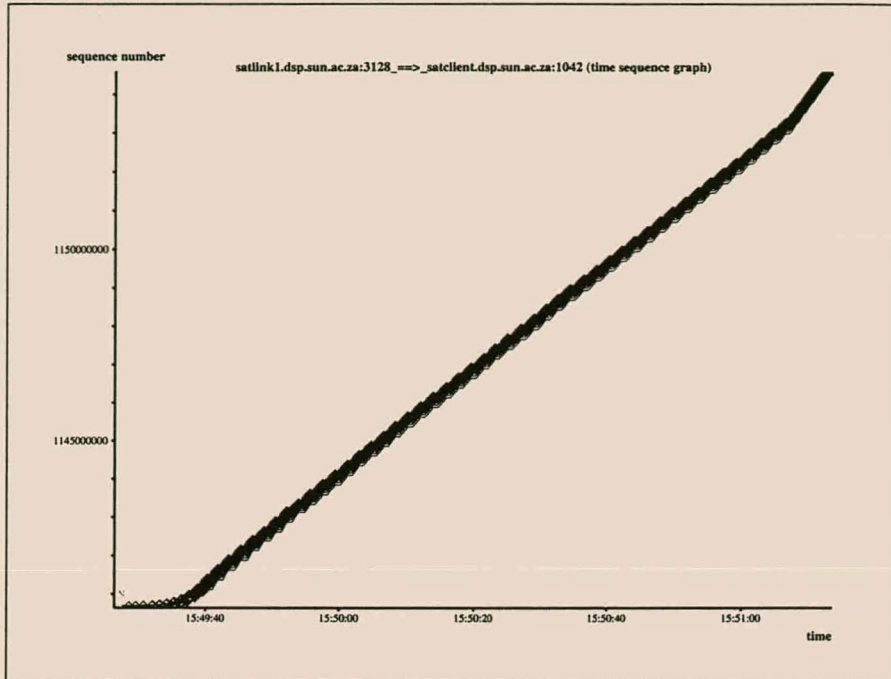


Figure 7.1: An extended slow start fills serial queue

Although the transfers seem to have reached linear growth, they have only reached the point where the serial system is functioning at maximum total throughput (eq. 5.6). Both connections remain in slow start for their duration: all the while the serial queue is growing, until it finally reaches a value of 66 – 88 packets (limited by the length of the file).

As can be seen in figure 7.1, as soon as the 13MB transfer stops, the 14MB transfer doubles its throughput. This happens because the queue has grown: when the 13MB transfer terminates, the queued ACKs of the 14MB transfer can utilize the full serial capacity.

Figure 7.2 illustrates the same transfer using Linux TCP Vegas [40] ($\alpha = 1$, $\beta = 3$, $\gamma = 1$). Here, the queue grows to a maximum of 15 – 30 packets, then does not grow beyond this.

In this case, when the 13MB transfer ends, the 14MB transfer continues at the same rate (while congestion avoidance slowly probes for extra capacity), instead of suddenly doubling. This happens because Vegas has set *ssthresh* to a suitable fair value for the

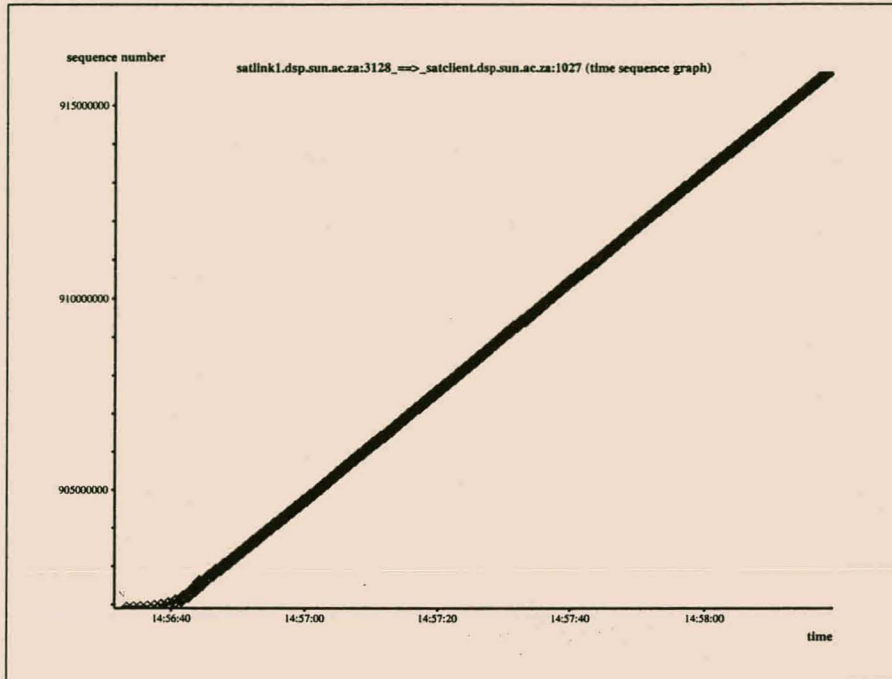


Figure 7.2: A TCP Vegas transmission

shared connection.

Thus, although Vegas solves the queueing saturation, it seems to be less desirable since it cannot rapidly use the freed capacity. This, however, is a dangerous viewpoint: the non-Vegas TCP can only react immediately because it has saturated the serial queue. Hence, the Vegas solution is more robust than the non-Vegas solution.

This illustrates a fundamental problem with high-latency TCP — any connection in congestion avoidance *cannot* adapt very speedily to capacity that becomes suddenly available. This is one of the main motivations to use a paced protocol for the satellite hop, which transmits data at the rate that the forward connection can support.

7.1.4 Vegas Comment

TCP Vegas is an example of where testing in the actual Internet is vitally important:

Allman [32] suggests that sender-based bandwidth estimation (such as in TCP Vegas) does not perform nearly as well in practice as it does in the laboratory. Low estimates

cause under-performance, while high estimates do not solve the problem.

One of the reasons cited, is the use of delayed ACKs. To compensate for this, [40] has built a min-RTT filter into Linux TCP Vegas to smooth the estimation.

It is possible that other solutions may exist, such as starting with an initial *cwnd* that is always a multiple of 2, and then increasing *cwnd* by a multiple of 2 on receipt of each ACK that acknowledges an even number of outstanding packets. Under the requirement that a host *must* acknowledge every 2nd packet, this should lead to no delayed ACKs being generated, hence increasing estimation accuracy.

Another effect is the alteration in ACK spacing which can occur in router queues (this is known as ACK compression [18]).³ Although this should not influence estimation during steady-state (the rates averaging out), it could have an effect during slow start on the estimation of the *ssthresh* transition point.

Consequently, [32] has proposed a receiver-based method where the receiver aids in measuring packet transit times more accurately. The argument for a receiver-based algorithm is backed up by findings such as from [25].

It is also noted that the interaction of Vegas and SACKs, and Vegas and ACK compression, require substantially more study. All of these are either active or future areas of research.

7.2 Slow Start Speedup

Many connections, such as typical web transfers, are short-lived. Because of long slow start times via satellite, these connections are usually constrained by the *cwnd* without obtaining the benefit of steady-state improvements.

Slow start occurs at various times. These forms of slow start are independent:

- Initial slow start

³ACKs may enter the router queue at a certain rate, but some may leave at an entirely different rate, depending on how many other packets the router has been buffering, and the speed of the outgoing interface.

- Slow start after RTO
- Slow start after idle time

The time spent in slow start to increase *cwnd* from an initial size of W_I to its final size W_F can be written compactly as follows (from [29, 41]):

$$\begin{aligned} \text{slow start time} &\cong n \text{ RTT } \log_2 \frac{W_F}{W_I} \quad [\text{sec}] \\ &= n \text{ RTT } \log_2 \frac{\frac{\min(\text{ssthresh}, \text{recwin}) \text{ [bytes]}}{\text{MSS [bytes]}}}{W_I \text{ [packets]}} \quad [\text{sec}] \end{aligned} \quad (7.1)$$

where the equality holds when $n = 1$ if every segment is ACKed. In the standard case of delayed ACKs with every 2nd segment being ACKed, $n = 2$, and the equation is approximate.⁴

Equation 7.1 demonstrates clearly another advantage of setting MSS as large as possible.

According to [33], W_I should be made equal to one of the following depending on the type of slow start:

- In the case of initial slow start: initial window IW where $IW = 1$ packet or $IW = 2$ packets.
- In the case of slow start after RTO: loss window LW where $LW = 1$ packet, irrespective of the value of IW .
- In the case of slow start after idle time: restart window $RW = IW$.

Clearly, slow start after RTO is the most expensive (followed by the even more expensive linear congestion avoidance), which is why such a premium was placed on avoiding RTO in the previous chapter.

⁴The non-linear behaviour of delayed ACKs makes it difficult to determine the exact time: A delayed ACK is sent as soon as the next ACK is due, or when the delayed ACK timer expires. The timer is 500ms according to the standard, but often implemented as 200ms.

It is important to realize [36] that using the new persistent HTTP⁵, modifications to initial slow start have very little impact on the overall page transfer rates, since only the first page transfer will initiate a new TCP connection. Furthermore, [36] shows that in many implementations, the client's request for the following page resets the server idle-timer, hence causing the following page to be sent at the full previous rate (rather than using slow start after idle time). *This may have serious consequences for the stability of a high-throughput network.*

Table 7.1: Slow start times (sec) for $recvwin = 250$ kB and $MSS = 1460$ B

W_I [packets]	RTT = 600 msec		RTT = 1000 msec	
	n = 1	n = 2	n = 1	n = 2
1	4.5	9	7.5	15
2	3.9	7.8	6.5	13
4	3.3	6.6	5.5	11

Table 7.1 summarizes a set of slow start times for the experimental system considered so far.

This table shows that these slow start times are long, causing a client to perceive the connection as 'unresponsive.' Each doubling of W_I decreases the slow start time by $n \times RTT$. Of course, the shortest times are obtained when $n = 1$ — decrease of n is obtained by opening $cwnd$ more aggressively during slow start.

7.2.1 Modified $cwnd$ Increase Algorithms

Slow start can be speeded by opening $cwnd$ by more than one segment size per ACK. For example, *limited byte counting* (LBC) [9, 29] uses the number of previously unacknowledged bytes to increase $cwnd$ each time an ACK arrives, with a maximum increase of 2 segments.

⁵Persistent HTTP allows a TCP connection to remain open after a web page transfer, allowing it to be used for subsequent web requests and hence saving the three-way handshaking time.

If this 2 segment limit is not enforced, large line-rate bursts can be generated when stretch ACKs (for large amounts of data) arrive (such as during loss recovery).

Although *cwnd* can be opened more aggressively, this holds certain consequences (see section 7.3).

7.2.2 Rate-Based Pacing

It has been suggested that a form of pacing TCP segments can be used for 1 RTT to start the ACK clock [9].

The rate at which segments are paced can be determined by prior estimation of the bandwidth, either from an initial estimate, or by using measurements from previous transfers. Instead of slow start (initially, after idle time or after an RTO), segments are paced out at a fraction (say half) of the previous rate for 1 RTT. This starts the ACK clock sufficiently for the remaining slow start to be rapid.

Unfortunately, any bandwidth estimate will at best be 1 RTT out of date when pacing begins, and as shown in [34,35], the congestive scenario can change drastically within this time.

In fact, it could be considered to measure characteristic times over which congestion varies on the dial-up network, and to try to calculate from that the factor by which the rate-pace should be adapted. However, the fractal, chaotic nature of Internet traffic [5] makes such attempts hopeless. Furthermore, deployment of paced TCP will alter the traffic balance in the Internet, effectively ‘skewing’ the original measurements.

7.2.3 Vegas Comment

The original TCP Vegas [37, 38] lengthens the slow start phase by only increasing *cwnd* on every second ACK. This is a result of the implementation of the *ssthresh* bandwidth estimator.

A slightly different implementation is used in Linux TCP Vegas [40], which allows *cwnd* to be increased on each ACK, thereby maintaining the same slow start times as

standard TCP.

7.3 Conclusions

The entire question of probing capacity and starting the connection to get it into steady-state, is a major research issue and heated debate. It raises many questions about stability, and hence deployment of new techniques is very slow. As such, it presents one of the major stumbling blocks to development of Internet technologies via satellite.

Large changes in *cwnd* (either from large initial windows, or from aggressive *cwnd* increases), will lead to the server bursting out data at its maximum rate. This will place strain on the serial queue, especially when multiple sessions are in slow start. Overflowing of the serial queue should be avoided especially during slow start — since the ACK clock may not yet be fully established, queue overflow may require an RTO to restart it.

The problem is mainly that in TCP, the sender is responsible for congestion control, and hence for the queue lengths along the network path. In this asymmetric environment, it is unfortunate that such overhead should be incurred in probing the dial-up network, since the bulk of the data will be sent over the forward path.

It would thus be ideal if the properties of the return path could be decoupled from the transmission over the forward path, thereby isolating control of the forward transmission from the shared Internet.

This is achieved to some extent both by a custom paced protocol (section 8.7) and by ACK compaction (Appendix A), the latter by its ability to send multiple ACKs with virtually the same number of bytes as a single ACK. ACK compaction may thus allow the sender to use large initial windows and a very aggressive *cwnd* increase algorithm, while incurring basically the same return traffic as standard TCP.

By providing its own queueing system, ACK compaction effectively carries out a form of queue management on the serial queue. This obviates the need for TCP Vegas. In fact, Vegas and ACK compaction should probably not be used together, since the timing variations introduced by ACK compaction may interfere with the Vegas estimation

algorithms.

In cases where a system such as ACK compaction is not used, and responsibility is on the sender to manage the client serial queue length, a system such as TCP Vegas should be used to estimate reasonable values for *ssthresh*. It is recommended that slow start be increased by no more than $IW=2$ and LBC.

Note that slow start speedup may change the results of section 6.4.4, so that an RTO followed by slow start may in fact deliver better responsiveness than growing serial queues and increased minimum RTO.

Chapter 8

Example Designs

This chapter uses the observations and conclusions of the previous chapters to propose a set of design guidelines and principles, which improve TCP performance in asymmetric satellite networks. These are illustrated by a set of tiered design examples.

The scenarios examined are:

- A standard return link with per-client throttling (96kB/s)
- A standard return link without throttling (192kB/s)
- A shared, header compressed dial-up link (356.4kB/s)
- A dedicated header compressed dial-up link (417.3kB/s)

As seen at various points in this thesis, it is advantageous to set the MSS as large as possible. This chapter will assume an MSS of 1460 bytes and an RTT of 600ms, with D taken as 10 and η as 1.

It is assumed that timestamps are disabled and that SACK or FACK is used. The term ‘shared ISP’ refers to dial-up infrastructure that is somehow shared with standard dial-up customers, or somehow uses the shared Internet for routing.

8.1 Baseline

To place the design examples of this chapter in perspective, recall that plain vanilla TCP should use a maximum window size of only 32kB (and may have a reduced MSS). This delivers a maximum throughput of 53.3kB/s (426.6 kbps) per TCP connection (for an RTT of 600ms).

In the absence of per-client forward throttling, with multiple TCP sessions per client (so that the client is running at over half the asymmetric throughput limit), the queuing problems of duplicate ACKs (section 6.4) and multiple slow starts (section 7.1.1) may arise.

8.2 Standard Return Link with Per-Client Throttling

In this scenario, header compression is not required (although it may be turned on — this will make the ACK stream co-exist more equitably with outgoing traffic such as e-mail). Shared ISP dial-up services can be used.

The provider chooses a serial bit-rate corresponding to *double* the TCP throughput it wishes to deliver. This is done using eq. 5.6 or table 5.1 for SLIP.

For example, the provider may pick a delivery rate of 0.75Mbps. Double this rate is 1.5Mbps, which requires a 28800 baud dial-up link.

Equation 5.1 is now used to calculate the window sizes. For a desired throughput of 0.75Mbps (96kB/s) with an RTT of 600ms, this translates to a window size of 57.6kB. The provider now sets its transmit socket buffers to 57.6kB, and has its clients set their receive socket buffers to 57.6kB. For anything larger than 32kB, RFC1323 window-scaling should be used.

The provider now throttles the forward traffic to each client to the desired throughput of 0.75Mbps. If a client opens a single TCP session, it will be able to obtain the full 0.75Mbps (96kB/s) throughput, while multiple sessions will share this available capacity.

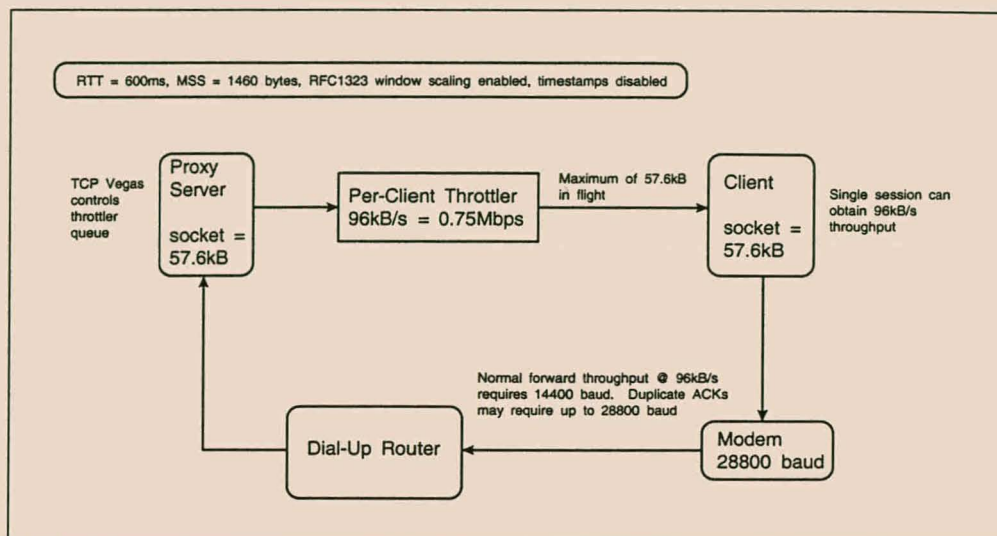


Figure 8.1: Example design using standard return link and throttling

A diagram of this example is shown in figure 8.1.

The rationale behind this is as follows: Because of the throttling, the provider can only have 57.6kB of data in flight to a client (across all of the client's connections). This requires an ACK rate capacity of about *half* that provided by the 28800 link.

In the absence of throttling, even by using TCP Vegas, multiple connections to the client would only be constrained by the maximum serial throughput, not the *recvwin*. In the worst case, if forward packets are dropped from all of the client's concurrent sessions (which will typically happen during rain-fading), the ACK rate will double. With throttling allowing only 57.6kB of data to be in flight to the client, the 28800 link is able to sustain this doubled ACK rate.

Thus, in the absence of outgoing data, the throttling ensures that data never enters the serial queue faster than it can leave. Many of the concerns of Chapter 6 are thus side-stepped.

Similarly, throttling avoids serial queue saturation by multiple slow starts (section 7.1.1). Nonetheless, the use of TCP Vegas is still recommended, otherwise *cwnd* will still grow without proper *ssthresh* limits during multiple slow starts, leading to large queues at the throttler. Effectively, the use of throttling moves many of the queueing issues from the serial queue to the throttler queue.

This design thus achieves a factor 1.8 throughput improvement (single connection) over the baseline, while avoiding queueing problems from duplicate ACKs and multiple slow starts — problems which may occur in the baseline system with many single-client concurrent connections.

Problems still remain: if the client concurrently does an outgoing transfer (such as sending e-mail), this transfer may be long and aggressive enough to overflow the serial queue and lead to satellite ACK-dropping. This may induce some of the scenarios of Chapter 6.

Similarly, if the dial-up link cannot sustain its speed (28800 baud in the example) because of flow control or line noise, doubling in ACK rate may lead to the problems of Chapter 6.

In these cases, the reliability of TCP should ensure that packets do get through eventually, although it will not be at optimal efficiency.

Slow start increase by LBC and $IW=2$ can be investigated.

8.3 Standard Return Link without Throttling

In this scenario, the provider allows the connection to be constrained by the maximum serial throughput. Hence, header compression should not be used, otherwise the dial-up ISP may be overflowed (if the provider cannot guarantee that header compression is turned off, it can throttle per-client the forward traffic to the serial throughput limit of eq. 5.6, i.e. 1.5Mbps for the following example).

The provider picks a suitable serial bit-rate and throughput rate using equation 5.6, leaving a slight throughput margin so that the serial link does not quite run at its maximum. For example, the provider may require a 28800 baud link for 1.5Mbps (192kB/s) throughput (from table 5.1).

The window size is now calculated using eq. 5.1, which equates to 115.2kB for the example. The server and client socket buffers are set to this size, and RFC1323 window scaling is enabled. A single client connection will thus be able to obtain a 192kB/s throughput, while multiple connections will share the capacity.

To avoid client serial queue strain during multiple slow starts, or if the RTT fluctuates downward (making the *recvwin* effectively too large for the capacity), TCP Vegas should be used on the sender. To enable Vegas to make effective bandwidth estimates, a slight lengthening of the serial queue may be required.

TCP Vegas will now regulate the amount of in-flight data to the client (across all concurrent sessions) to be properly constrained by the serial queueing rate. In the example, multiple connections will be regulated to about 1.53Mbps (from table 5.1).

In the event of forward packet loss, eq. 6.3 thus predicts a maximum clearing time of about 2.34 seconds for a growable serial queue. Since this is larger than the RTT, the growable serial queue / increased minimum RTO -solution (section 6.4.3) would be dangerous, as discussed in section 6.4.4.

Consequently, the standard RTO and limited queue (section 6.4.2) should be used. This is a viable option if the satellite provider is confident that forward packet loss (e.g. because of rain fading) is reasonably rare.

Note also that use of LBC during slow start may lead to speedier transfer than the congestion avoidance of the growable queue / increased minimum RTO -solution.

This design achieves a factor 3.6 throughput improvement (single connection) over the baseline, while avoiding the queueing problems of multiple slow starts. Duplicate ACKs will, however, lead to serial queue overflow, with consequent packet dropping for all the client's connections.

Other shortcomings are similar to those discussed in section 8.2.

Slow start modifications such as $IW=2$ and LBC can be investigated. More aggressive modifications may lead to congestive problems on the shared infrastructure.

8.4 Header Compression with Shared ISP

The satellite provider should reach an agreement with the shared dial-up provider, concerning the bandwidth required per client for ACKs on the dial-up backbone. The satellite provider now calculates the total forward throughput per client from eq. 5.8.

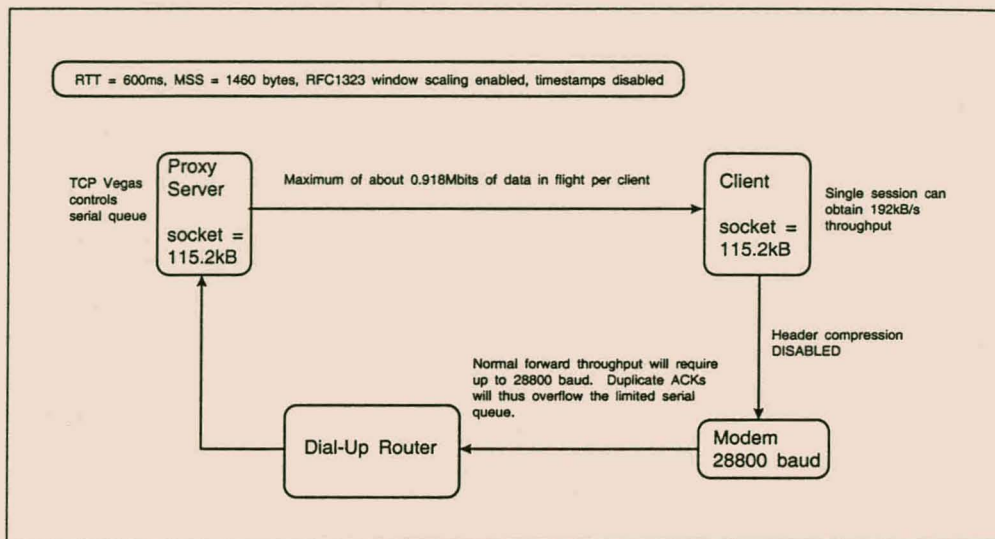


Figure 8.2: Example design with standard return link, no throttling

Notice that this rate will *not* double in the event of packet loss, since the duplicate ACKs will turn header compression off.

Say that the dial-up provider agrees to support an ACK bandwidth on its backbone of 5000 Bps from each satellite client. Using eq. 5.8, this translates to a total forward throughput of 365000 Bps, or 356.4 kB/s (2.78Mbps) per client.

From equations 5.6 and 5.7, this requires a serial link of at least 11250 bps, thus at least a 14400 baud modem (see table 5.1).¹

To ensure that the client cannot generate more than 5000 bytes/sec of ACKs on the backbone, the satellite provider has to throttle the forward traffic to the client to 356.4 kB/s (2.78Mbps). Otherwise, if the client opens multiple connections they will only be constrained by the maximum serial rate.

TCP Vegas should be used to regulate the bottleneck queues, whether this turns out to be the throttler or the dial-up ISP.

The window sizes are calculated from eq. 5.1 (213.84 kB in this case), and the socket buffer sizes set accordingly. A single TCP session to a particular client will now be able to obtain a throughput of 356.4 kB/s (2.78Mbps), while multiple connections will

¹Even higher modem speeds are of greater benefit to simultaneous outgoing data, e.g. e-mail

share this capacity.

The example is illustrated in figure 8.3.

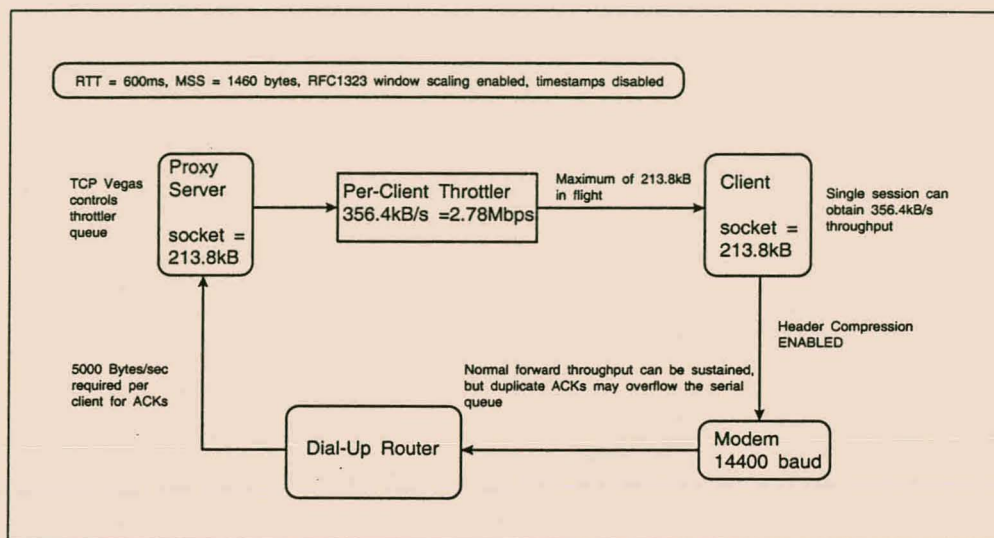


Figure 8.3: Example design: header compression, throttling and shared dial-up

Because of the high ACK rate², the standard limited serial queue and default RTO should be used (section 6.4.2). In the interest of dial-up stability (see section 6.4.4), any dormancy period (followed by high-rate transmission, section 6.4.3) will almost certainly be too long.

Slow start modifications of IW=2 and LBC can be investigated. This may also prove to give faster recovery from dropped packets than a growable serial queue with increased minimum RTO.

This design achieves a factor 6.7 throughput improvement (single connection) over the baseline, while avoiding queueing problems from multiple slow starts. Duplicate ACKs will, however, cause the serial queue to overflow, leading to packet dropping for all client connections. Furthermore, proper return-path bandwidth agreements have to be in place with the dial-up ISP.

Other shortcomings of this design are similar to those of section 8.2.

²A standard dial-up client's 33600 modem typically only requires a maximum byte-rate of around 4000 bytes/sec, uncompressed

8.5 Header Compression with Dedicated Dial-Up

For this scenario, the satellite provider should investigate slow start speedups of IW=2 and LBC. It should now decide which option delivers the better performance: a limited serial queue with standard RTO (section 6.4.2) or a growable serial queue with increased minimum RTO (section 6.4.3).

In the case of the former, design proceeds similarly to section 8.4. This section examines design of the latter option.

Firstly, the provider decides on a dormancy period and minimum RTO for the scenario of section 6.4.3. The minimum RTO should not be too large, because in certain cases an RTO may be required to resume transmission (see Chapter 6). Say that the provider chooses a minimum RTO of 5 seconds.

Equation 6.3 is now used to determine the ratio of per-client throughput to serial rate. Allowing that *tcph* may be 60 bytes while carrying SACKs, eq. 6.3 yields for this example:

$$\frac{D.(throughput\ limit)}{baud} = 148$$

Hence, if a 28800 baud modem is required, a throughput of 427317 Bps, or 417.3kB/s (3.26Mbps) can be sustained.

Examination of table 5.1 shows that this is about half the rate that a 28800 modem can actually support. This is the sacrifice made to obtain a minimum RTO of 5 seconds.

To ensure that the client cannot use more capacity by running multiple connections, the provider has to throttle the throughput to 3.26Mbps per client. TCP Vegas should now be used to manage the throttler queue.

From equation 6.2 (using the throttling limit), the maximum serial queue length is calculated as 176 packets. The client serial queue must thus be set to a maximum length of about 200 packets (to queue outgoing traffic as well).

In the worst-case event that packets are dropped from all a client's concurrent forward connections, a dormancy period of 5 seconds maximum will ensue, after which

congestion avoidance transmission will resume at high rate. Since the dial-up infrastructure is dedicated to the satellite provider, it cannot adversely affect low-RTT connections which have in the interim grabbed the transient capacity.

The socket buffers are set to the window size, calculated from eq. 5.1 as 250.38kB. A single TCP connection will thus be able to obtain throughput of 417.3kB/s (3.26Mbps), while multiple connections will share this capacity. This example is illustrated in figure 8.4.

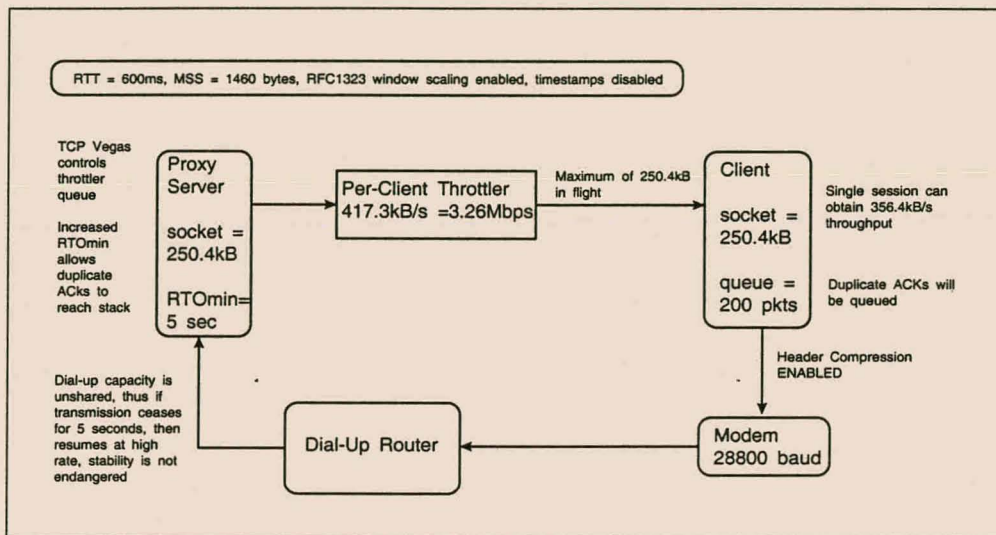


Figure 8.4: Example design: header compression with dedicated dial-up

This design achieves a factor 7.8 throughput improvement (single connection) over the baseline, while avoiding the serial queueing problems caused by duplicate ACKs and multiple slow starts. However, this requires increase of the minimum RTO (which in turn will decrease responsiveness in the cases where an RTO is required to restart transmission), and also relies on a completely dedicated dial-up network for stability — possibly a prohibitively expensive option.

Other shortcomings are similar to those of section 8.2.

8.6 ACK Compaction

ACK compaction has the potential to address the shortcomings of the previous designs. Since a large number of ACKs can be compacted into roughly the same overhead as a single ACK, large initial *cwnd* can be used as well as aggressive *cwnd* increase policies.

Furthermore, more aggressive congestion avoidance policies should be possible, without endangering the dial-up network. Subsequently, ACK compaction promises to provide the network with a way in which slow start can be very rapid, as can utilization of transient bandwidth.

Since ACK compaction can sustain varying ACK rates, the use of TCP Vegas should not be necessary. In fact, it may be problematical because of timing variations introduced by compaction. For this reason, it also appears that the use of ACK compaction requires a slight increase of the minimum RTO (see Appendix A).

ACK compaction allows the mismatched MSS problem to be solved cleanly, and since ACK compaction can manage the client serial queue directly (without having to rely on the TCP sender to do this), it could provide for the cleanest co-existence with outgoing traffic.

The limits of ACK compaction are not currently well understood and more research is required.

8.7 A Custom Protocol

As has been seen, TCP possesses certain fundamental characteristics (such as the sliding window flow control and congestion control) which makes its use via asymmetric satellite networks problematical. Specifically, to deliver data to the edge of the network, the maximum possible throughput should be limited only by the available forward capacity. TCP cannot immediately utilize available capacity but has to probe for it.

It may thus be tempting to replace TCP over the satellite hop with a custom protocol. This should be done transparently to the client. A possible method is illustrated in figure 8.5, which operates as follows:

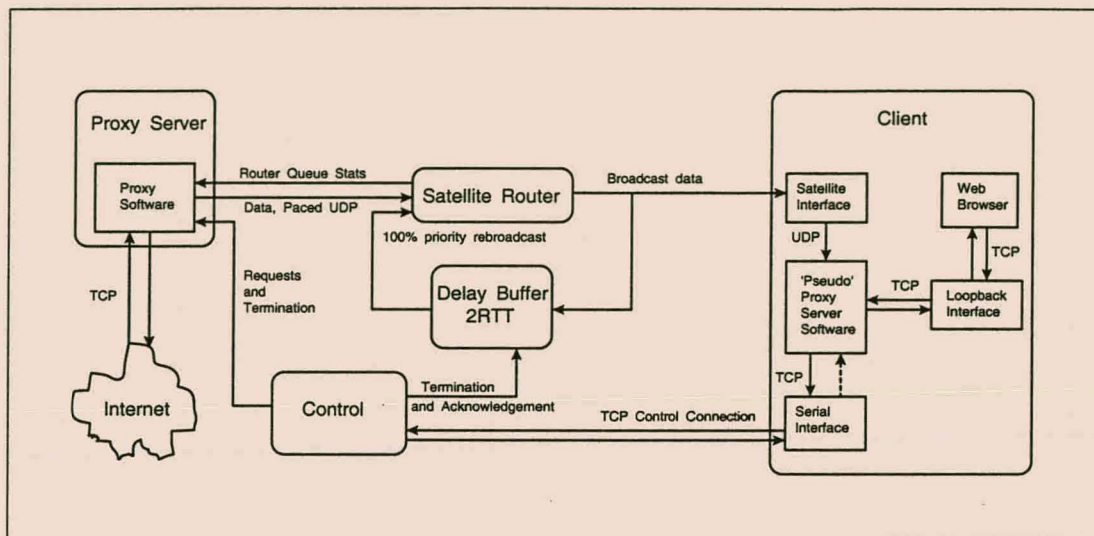


Figure 8.5: Example design using a custom paced protocol

The client runs software which acts like a proxy server. To download a web page, for example, the web browser would request the page from this pseudo proxy server, via the client's loopback interface.

This pseudo proxy server could then open a TCP connection via the serial link to the satellite proxy server. This requests the data, and acts as control connection for the satellite transmission. The use of TCP over the serial link makes this control (and also the custom protocol's acknowledgement-mechanism) reliable and congestion controlled in and of itself. The custom protocol can thus implement large-block acknowledgements without too much consideration of congestion issues.

Data received via this custom protocol is then sent by the pseudo proxy to the web browser, using the client's standard TCP stack via its loopback interface.

At the satellite provider, the proxy server obtains the data using standard TCP. However, a custom protocol is used to transmit this data via satellite, for example by encapsulation in UDP packets. The packets are paced out at a rate determined by factors such as:

- the available queue capacity of the satellite router.
- the rate allocated to the client.
- the rate at which the client's satellite card can reliably receive data.

To buffer data for retransmission, the following innovative scheme can be used:

Data emerging from the satellite router is copied, verbatim, to a buffer which delays this data for some time longer than the RTT (say for $2RTT$). This delay thus preserves the spacing of the data as transmitted via the satellite link.

When the controlling TCP connections indicate that blocks of data have been received, the relevant packets are cleared from the delay (while keeping the spacing of the other packets intact).

Packets emerging from the delay have thus not been received by the client within the delay time, and are resubmitted into the satellite queue. These packets from the delay have top access priority to the queue, even to the exclusion of packets from the proxy server.

In the event of rain fading, packets can thus circle around the delay-loop a few times until received, temporarily halting transmission of new data. Thereafter, transmission resumes at full rate.³

To prevent a packet from looping around the delay indefinitely, the buffer should decrement the IP time-to-live field of the packet and eventually discard it.

If the TCP control connection from a particular client is terminated, the proxy server is notified to terminate transmission, and any associated packets are removed from the delay.

In the event of a client crash, it may take some time for the control TCP connection to time out. The client pseudo proxy should thus regularly send 'keepalive' packets via the control link, the absence of which will trigger termination as above.

³Note that, in the outline of figure 8.5, packet loss between the proxy server and the satellite router may be irrecoverable. Hence, the server and router should be a self-contained unit, or additional robustness should be built into the system (such as feedback or erasure coding).

To provide protection against rain fading or data corruption, erasure codes could be used to generate cross-packet redundancy. For example, it is possible to code k packets into $n = 2k$ packets so that the original packets can be reconstructed from the reception of any k packets from this group.

8.8 Conclusions

This chapter has shown how to use the observations from previous chapters to improve TCP throughput, while maintaining as much network robustness and stability as possible.

It has been seen that the use of active queue management techniques are important for this stability, and consequently, TCP Vegas and per-client throttling feature prominently in this chapter.

The chapter has illustrated successive improvements to TCP throughput, which come at the cost of various trade-offs, and with successive increases in infrastructure costs. It was noted how ACK compaction may improve this situation (as well as potentially speeding up slow start and congestion avoidance) by providing a greater measure of 'decoupling' between the forward and return paths.

By becoming aware of the issues described in this and the other chapters, the provider becomes empowered to design a custom paced protocol able to side-step these issues. Consequently, this chapter has also suggested some outlines for just how such a protocol may function, in a fashion transparent to TCP and the client.

Chapter 9

Conclusions

Optimizing TCP on high-latency asymmetric networks is a multi-faceted science. It includes network- and architectural designs, modifications and enhancements of TCP stacks and optimization of the return link.

For conclusions on individual techniques, please refer to the conclusions of the individual chapters. These observations have been used in Chapter 8 to propose designs for enhancing TCP performance.

9.1 Network Emulation

The use of network emulation has been found to be fraught with problems, including having to keep track of multiple software revisions, upgrades with their own bugs and idiosyncrasies, and analysis tools which suffer from various minor bugs and incompatibilities.

It is also difficult to measure and log certain internal parameters using a network emulator. Furthermore, reconfiguring the system for various experiments requires significant turn-around time, since kernel recompilation and reboot may be required.

To aid in troubleshooting and validation, it is thus recommended that network emulation be carried out in tandem with network simulation.

9.2 Network Architecture

As discussed in Chapter 8, it may be undesirable to run the serial link at maximum rate. In the case of a single TCP connection to a client, this can be controlled by setting *recvwin*. However, for multiple TCP connections to a specific client, this control by *recvwin* becomes difficult because of the wide spread in possible numbers of simultaneous connections. Hence, forward per-client throttling may be needed.

This is especially true where the client is an office LAN.

9.3 Header Compression

Many of the problems with VJ header compression are related to its use of duplicate ACKs and TCP retransmissions to resynchronize the states of the compressor and decompressor.

It should be possible to create a header compression system where the decompressor can independently detect synchronization loss, and request a state-update via handshaking on the serial link without having to resort to TCP retransmissions and duplicate ACKs. Such a system could thus transport duplicate ACKs for dropped packets without loss of the high transfer rate as witnessed in Chapter 6.

Nonetheless, certain problems will remain, such as the possible strain on ISP bandwidth and the inability to increase slow start and congestion avoidance speed overaggressively, because of congestion implications.

9.4 TCP Congestion Notification

As seen in Chapter 6, packet loss degrades satellite performance. Consequently, satellite TCP is not very compatible with techniques such as Random Early Detection (RED), where routers drop packets at random when queues fill, in an effort to signal congestion to as many TCP streams as possible.

This is another motivation for the premium placed on active queue management techniques in this thesis.

9.5 Future TCP

The occurrence of transmission burstiness has been observed frequently enough¹ to recommend that a TCP sender utilize some form of pacing mechanism when *cwnd* undergoes large changes in size or position.

The ‘Vegas*’ technique [38] takes a step in this direction by *not* increasing *cwnd* immediately upon ACK reception, but rather by scheduling a *cwnd*-increase event at some future time, based on its available bandwidth estimate.

As discussed in section 7.1.4, sender-based delay measurements tend to perform better in the laboratory than in practice. Receiver-based aid for these measurements should be investigated [25, 32]. This is important for accurate bandwidth estimation, which aids in active queue management.

This thesis has highlighted the important impact of active queue management techniques on network robustness and reliability. These techniques are problematical from the edges of the network, since the end-point TCPs have to infer many characteristics of the intervening network only from observations at the end-points.

It can be concluded that, for asymmetric satellite TCP, management of each individual queue along the route is beneficial, and that this management is probably not best done only by the TCP sender. Hence it would be of benefit if every queue could provide some standard feedback mechanism by which the end-points could be kept informed about the status of the intermediate network.

The standardization of a true MSS negotiation method should be considered. As observed in this thesis, the forward MSS is a deciding factor of TCP performance over the satellite path. Currently, determination of an optimal asymmetric MSS is problematical.

¹E.g. after fast recovery, slow start, with ACK compaction, etc.

To increase TCP's throughput capability, the window sizes have to be increased. This increases the in-flight data, and hence also the noise sensitivity of the congestion control algorithms [17]. In the satellite scenario, the slightest congestion may dramatically reduce performance, with subsequent recovery being painfully slow.

These algorithms were primarily developed with small window sizes in mind, hence the increasing prevalence of large-window transfers argues for a re-evaluation of the TCP congestion control mechanisms.

Specifically, [17] suggests that during each RTT of linear congestion avoidance, the window size be increased by an amount much less than the pipe-size (the bandwidth-delay product minus protocol overheads), and then proceeds to fix this to 1 packet, according to the typical window sizes in use on the Arpanet at that time.

This places high-latency connections at a disadvantage since they cannot utilize transient bandwidth as rapidly as can lower-RTT connections. It presents the greatest single 'slow-growth' factor for satellite TCP.

Clearly the effect should be investigated of varying this additive increase according to the window size.

9.6 Custom Paced Protocol vs. ACK Compaction

The interconnectedness of TCP makes it difficult to work with, since a modification to one area invariably influences other areas. In fact, much of this thesis reads more like a litany of TCP problems, rather than TCP solutions, with many of the solutions being double-edged. Even though the techniques discussed here take a substantial step toward improving performance, the full satellite capacity can still not be utilized by a single TCP session, and capacity is still wasted during slow start and congestion avoidance.

Although ACK compaction has the potential to address many of these problems, the issues of forward congestion control required for ACK compaction (see Appendix A) are much akin to the forward pacing issues required for a custom paced protocol (refer to section 8.7).

ACK compaction clearly needs more study. Irrespective of the aggressiveness of *cwnd* increase algorithms, slow start and congestion avoidance will still incur some time penalties in comparison to a paced protocol. Using ACK compaction, it may ultimately be possible to replace slow start and congestion avoidance completely by a form of rate-based pacing.

In this case, the sender protocol no longer resembles TCP much. It is thus an open question which will provide the greater benefit for the least effort: design of a custom paced protocol, or refinement of ACK compaction.

9.7 Final Conclusions and Recommendations

This thesis has studied and quantified the following (which, as far as could be ascertained, has not been published elsewhere yet):

- The asymmetric throughput limit and its parameters (Chapter 5).
- The serial queueing implications of dropped packets during transmission at higher than half the serial throughput limit, especially as relates to the use of VJ header compression (Chapter 6).
- The serial queueing implications of multiple simultaneous slow start sessions (Chapter 7).
- The use of these foregoing observations to propose a set of design guidelines and example designs (Chapter 8), including the outlines of a transparent custom paced protocol to side-step the various problems encountered with TCP-based solutions.

It has been seen that optimization of TCP is a difficult area, with many unexpected interactions between various facets of TCP. Specifically, it is difficult to enhance TCP throughput performance, and yet remain within the somewhat subjective boundaries of robustness and the spirit of TCP/IP.

From Chapter 8 it was seen that various levels of TCP improvement can be obtained for various levels of trade-off. Specifically, a factor 7.8 throughput improvement (single connection) could be obtained over the baseline, but at the cost of a dedicated dial-up infrastructure.

It was concluded that ACK compaction may provide the cleanest TCP solution to these limitations. However, a preliminary investigation of ACK compaction (Appendix A), reveals that ACK compaction has its own set of problems, the resolution of which would most probably require as much effort as designing a new, custom, paced protocol.

Having identified many problems with enhancing TCP performance, this thesis thus also provides the knowledge which enables the design of a custom protocol specifically to side-step these issues.

Since such a protocol would give the provider full control over all aspects of the network, and may obviate any need to probe for network or dial-up capacity, it may potentially deliver even more responsive connections than ACK compaction would ever be capable of.

Consequently, it is finally concluded that design of a custom paced protocol may prove a cleaner, more gratifying option in the long term, with greater capability for expansion. This thesis has proposed the outlines of just such a system (see section 8.7), which is able to function transparently to TCP and to the client.

Bibliography

- [1] W. R. Stevens, *TCP/IP Illustrated*, vol. 1. Addison-Wesley, Reading, Massachusetts, 1994.
- [2] S. Floyd, "Internet Research: Comments on Formulating the Problem." Unpublished manuscript in progress, available from the LBL Network Research Group Website, <http://www-nrg.ee.lbl.gov/nrg-papers.html>, 21 January 1998.
- [3] B. Braden and D. Clark, e.a., "RFC2309: Recommendations on Queue Management and Congestion Avoidance in the Internet," April 1998.
- [4] C. Partridge and T.J. Shepard, "TCP/IP Performance over Satellite Links," *IEEE Network*, vol. 11(5), pp. 44–49, Sept/Oct 1997.
- [5] W. Willinger and V. Paxson, "Where Mathematics meets the Internet," *Notices of the American Mathematical Society*, vol. 45(8), pp. 961–970, Sept. 1998.
- [6] J. C. Hoe, "Start-up Dynamics of TCP's Congestion Control and Avoidance Schemes," Master's thesis, Dept. of Electrical Engineering and Computer Science, MIT, June 1995.
- [7] G. R. Wright and W. R. Stevens, *TCP/IP Illustrated*, vol. 2. Addison-Wesley, Reading, Massachusetts, 1995.
- [8] M. Allman, D. Glover, and L. Sanchez, "RFC2488 BCP28: Enhancing TCP Over Satellite Channels using Standard Mechanisms," January 1999.
- [9] M. Allman, S. Dawkins, and D. Glover, e.a., "Ongoing TCP Research Related to Satellites." IETF draft-ietf-tcpsat-res-issues-07.txt, work in progress, expires November 1999, May 1999.

- [10] M. Allman, C. Hayes, H. Kruse, and S. Osterman, "TCP Performance over Satellite Links," in *Proc. 5th International Conference on Telecommunications Systems*, (Nashville), March 1997.
- [11] R. C. Durst, G. J. Miller, and E. J. Travis, "TCP Extensions for Space Communication," in *ACM Proc. 2nd Annual International Conference on Mobile Computing and Networking (MobiCom)*, pp. 15–26, November 1996.
- [12] H. Balakrishnan, V. N. Padmanabhan, and R. H. Katz, "The Effects of Asymmetry on TCP Performance," in *Proc. 3rd Annual ACM/IEEE Intl. Conference on Mobile Computing and Networking (MobiCom), Budapest, Hungary*, pp. 77–89, Sept. 1997.
- [13] N. K. G. Samaraweera, "Return Link Optimization for Internet Service Provision Using DVB-S Networks," *ACM SIGCOMM Computer Communications Review*, vol. 29, July 1999.
- [14] V. Paxson, M. Allman, and S. Dawson, e.a., "Known TCP Implementation Problems." IETF draft-ietf-tcpimpl-prob-05.txt, work in progress, expires May 1999, November 1998.
- [15] V. Jacobson, R. Braden, and D. Borman, "RFC1323: TCP Extensions for High Performance," May 1992.
- [16] M. Mathis, J. Madhavi, S. Floyd, and A. Romanov, "RFC2018: TCP Selective Acknowledgement Options," October 1996.
- [17] V. Jacobson and M. J. Karels, "Congestion Avoidance and Control," in *Proc. SIGCOMM, Stanford, CA*, ACM, Aug. 1988. The paper used is a slightly revised version, Nov. 1988, from the LBL website.
- [18] J. C. Mogul, "Observing TCP Dynamics in Real Networks," Tech. Rep. 92/2, Digital Equipment Corp., Western Research Lab., April 1992.
- [19] European Telecommunications Standards Institute, "Digital broadcasting systems for television, sound and data services; Framing structure, channel coding and modulation for 11/12 GHz satellite services." ETS 300 421 Reference DE/JTC-DVB-6, December 1994.

- [20] S. Cheshire, "Latency and the Quest for Interactivity." <http://ResComp.stanford.edu/~cheshire>, November 1996.
- [21] V. Jacobson, "RFC1144: Compressing TCP/IP Headers for Low-Speed Serial Links," Febr. 1990.
- [22] T. R. Henderson and R. H. Katz, "Transport Protocols for Internet-Compatible Satellite Networks," *IEEE Journal on Selected Areas in Communications*, vol. 17(2), pp. 326–344, February 1999.
- [23] M. Allman, A. Caldwell, and S. Ostermann, "ONE: The Ohio Network Emulator," Tech. Rep. TR-19972, Ohio University, August 18 1997.
- [24] T. Bonald, "Comparison of TCP Reno and TCP Vegas via Fluid Approximation," Tech. Rep. 3563, Unite de recherche INRIA Sophia Anitpolis, November 1998.
- [25] V. Paxson, "On Calibrating Measurements of Packet Transit Times," in *Proc. SIGMETRICS, Performance Evaluation Review*, vol. 26, pp. 11–21, June 1998. The actual reference used is a longer version of this paper, obtainable from the LBL website.
- [26] M. Allman, "Improving TCP Performance over Satellite Channels," Master's thesis, Ohio University, June 1997.
- [27] C. Hayes, "Analyzing the Performance of new TCP Extensions over Satellite Links," Master's thesis, Ohio University, August 1997.
- [28] Pittsburgh Supercomputing Centre (PSC), "Enabling High Performance Data Transfers on Hosts." http://www.psc.edu/networking/perf_tune.html.
- [29] M. Allman, "On the Generation and Use of TCP Acknowledgements," *ACM Computer Communication Review*, vol. 28(5), pp. 4–21, October 1998.
- [30] S. R. Johnson, "Increasing TCP Throughput by Using an Extended Acknowledgement Interval," Master's thesis, Ohio University, June 1995.
- [31] N. K. G. Samaraweera, "High Speed Internet Access Using Satellite-Based DVB Networks," in *International Network Conference '98*, (Plymouth, UK), pp. 23–28, 1998.

- [32] M. Allman and V. Paxson, "On Estimating End-to-End Network Path Properties," in *Proc. ACM SIGCOMM*, (Cambridge, Mass.), September 1999. to appear.
- [33] M. Allman, V. Paxson, and W. Stevens, "RFC2581: TCP Congestion Control," April 1999.
- [34] V. Paxson, *Measurements and Analysis of End-to-End Internet Dynamics*. PhD thesis, Computer Science Division and Lawrence Berkeley National Laboratory, University of California, Berkeley, April 1997.
- [35] V. Paxson, "End-to-End Internet Packet Dynamics," in *ACM Proc. SIGCOMM, Computer Communication Review*, pp. 139–152, June 1997.
- [36] H. Kruse, M. Allman, J. Griner, and D. Tran, "HTTP Page Transfer Rates over Geo-Stationary Satellite Links," in *Proc. 6th International Conference on Telecommunications Systems*, March 1998.
- [37] L. S. Brakmo, S. W. O'Malley, and L. L. Peterson, "TCP Vegas: New Techniques for Congestion Detection and Avoidance," *ACM SIGCOMM, Computer Communication Review*, vol. 24(4), pp. 24–35, August 1994.
- [38] L. S. Brakmo and L. L. Peterson, "TCP Vegas: End to End Congestion Avoidance on a Global Internet," *IEEE Journal on Selected Areas in Communications*, vol. 13(8), pp. 1465–1480, Oct 1995.
- [39] J. S. Ahn, P. B. Danzig, Z. Liu, and L. Yan, "Evaluation of TCP Vegas: Emulation and Experiment," *ACM SIGCOMM, Computer Communication Review*, vol. 25(4), pp. 185–195, 1995.
- [40] N. Cardwell and B. Bak, "A TCP Vegas Implementation for Linux." <http://www.cs.washington.edu/homes/cardwell/linux-vegas/>, August 1999.
- [41] M. Allman, C. Hayes, and S. Ostermann, "An Evaluation of TCP Slow Start Modifications," *Submitted to Computer Communications Review*, 1998.
- [42] M. Allman, C. Hayes, and S. Ostermann, "An Evaluation of TCP with Larger Initial Windows," *ACM Computer Communications Review*, vol. 28(3), pp. 41–52, July 1998.

Many of these references can be obtained in electronic format from:

- <http://roland.lerc.nasa.gov/~mallman/papers/>
- <http://tcpsat.lerc.nasa.gov/tcpsat/>
- <http://jarok.cs.ohiou.edu/papers/>
- <http://ctd.lerc.nasa.gov/5610/inetprotocols.html>
- <http://www-nrg.ee.lbl.gov/nrg-papers.html>
- http://www.erg.abdn.ac.uk/public_html/publications/refs/index.html
- <http://www-scps.jpl.nasa.gov/scps/html/documents.html>

RFCs form part of the Internet Standards documents, and can be obtained from many mirrors, such as:

<ftp://ftp.sun.ac.za/info/rfc>

Note that RFCs may be updated, replaced or obsoleted by newer RFCs. Please examine the file *rfc-index.txt* at any RFC distribution for the current status of a particular RFC.

Appendix A

ACK Compaction

This appendix documents some preliminary findings obtained by using ACK compaction in the emulation platform.

A.1 Traditional ACK Compaction

ACK compaction, as introduced in [13, 31] basically functions as follows:

The serial queue is monitored. If a specific connection has more than a certain number of ACKs (typically 4) in the queue, all except the last of these ACKs are deleted (up to a high water mark, typically 60 ACKs).

Three bytes are now appended to this last ACK to indicate the number of ACKs deleted and the average amount of data acknowledged per ACK. The decompressor uses this to recreate approximately the original ACK stream. To attempt to maintain the approximate ACK spacing, the decompressor has to employ special traffic shaping techniques.

This method thus compacts a varying number of ACKs statelessly into a fixed overhead, simultaneously managing the serial queue length.

A.1.1 Questions

Two questions can be raised:

- The ability of this technique to ensure correct reconstruction of duplicate ACKs.
- The ability of this technique to manage the advertised *recvwin* properly.

The first is necessary for proper operation of fast retransmit / fast recovery.

The second is an interesting issue: The validity of the *recvwin* field is questionable, since this information is already 1 RTT out of date when the sender's response to it reaches the client.

However, the *recvwin* does have an important function: The client TCP can advertise a smaller *recvwin* on each segment, if the client process cannot read data fast enough from the network, or for some reason stops reading (e.g. because of a crash).

The incoming data at the satellite interface would normally not be dropped if there were no space on the receive socket (i.e., if current advertised *recvwin* is zero, but data is still arriving). It would simply be queued in the satellite interface device driver queue, until the TCP stack reads it from there into the receive socket buffer. Hence, the *recvwin* advertisement is the mechanism by which the *incoming satellite device driver queue* is managed.

A.1.2 Other Observations

Using traditional ACK compaction [13], the initial segments of a new session are transmitted more slowly than by using header compression, since full-size ACKs are traversing the serial link. As soon as the ACK compaction 'kicks in' or 'fires,' because multiple ACKs start to occupy the serial queue, performance increases.

This initial delay may be circumvented by using large initial *cwnd* and aggressive *cwnd* increase policies.

As documented in [13] by simulation, probably the most worrisome tendency is that ACK compaction builds large persistent queues at the satellite router, even despite the

use of traffic shaping on the returning ACKs. This indicates that ACK compaction should be used together with some form of forward queue management system. Because ACK compaction alters the timing patterns of the ACKs to some degree, the estimators of systems such as TCP Vegas may be negatively influenced. Hence, congestion monitoring and feedback methods in the forward path (such as ICMP source quench) should be investigated, even extending to the possible use of a pacing mechanism in the TCP server.

A.2 Revised ACK Compaction

To address the questions of section A.1.1, a different form of ACK compaction was designed and implemented for this thesis, and some preliminary tests were carried out. The experimental architecture is shown in figure A.1.

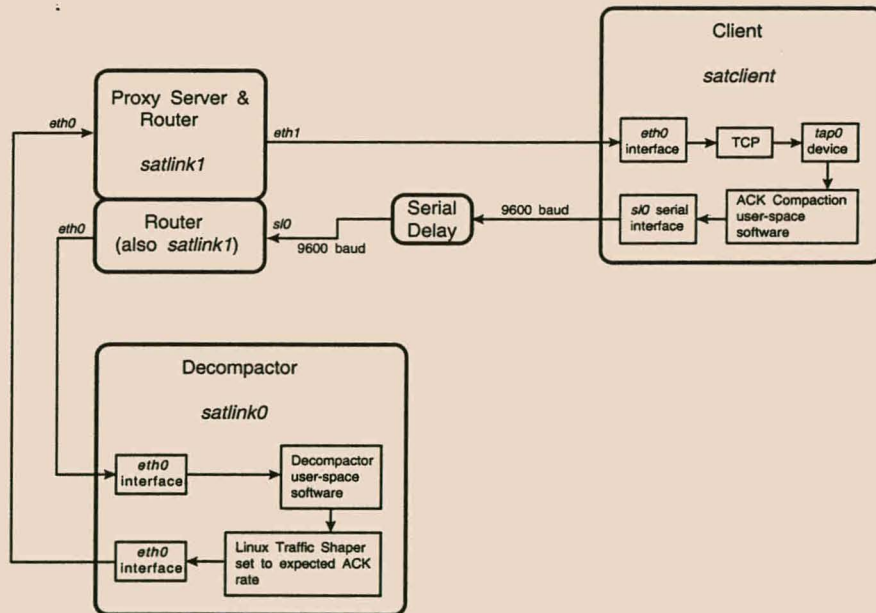


Figure A.1: Preliminary emulation of revised ACK compaction

Rather than monitoring the serial queue, client traffic to the proxy server is routed on the Linux client to a Linux *tap* device (instead of to *s10*). Here, the compactor (a

user-space program) encapsulates any traffic in UDP packets to be sent to the remote decompressor via the serial link.

In so doing, it was demonstrated quite clearly how the maximum MSS could be obtained on the forward link by setting the tap-device MTU to 1500, without affecting the serial device MTU of 576.

Any ACKs are placed in a queue, associated with that particular session's 4-tuple endpoints. Only one full ACK is kept in each queue: for each additional ACK, information is appended to the queue detailing a change in *recvwin* from the previous ACK, and the exact number of bytes ACKed.

Thus, for *each* additional ACK, either 2 or 4 bytes are appended.

Any ACK which cannot be compacted (e.g. because it contains data), causes the particular queue to be flushed out, followed by this new segment. In so doing, the remote decompressor can *exactly* reproduce the ACK packets, except in the event of a compactor-packet-loss between compactor and decompressor.

A.2.1 Timing Considerations

In order to manage the serial queue, the compactor waits for a time x , during which all received ACKs are queued (with the exceptions as noted above). After this period, all queued streams are sent out compacted.

For the preliminary examination, this amount of time was set to a fixed amount (from 50 to 100 ms, from an estimation of the time required to transmit a compacted packet via the serial link). In this method, ACK compaction incurs the cost of slight additional latency.

The idea is to put no strain on the serial queue. A better approach may be to calculate the serial transmission time for the compacted packets, and then wait for *this* amount of time, before transmitting the next compacted packet.

So that other traffic can also share the queue fairly, feedback from the serial queue could be incorporated, in order to take into account the time taken for all serial data on the queue, not only the compactor packets.

A.2.2 Observations

In contrast with traditional ACK compaction, this revised technique was tested with network emulation, rather than simulation.

The minimum RTO on the server had to be increased slightly to avoid timeouts caused by the timing variations introduced by ACK compaction.

Even with a plain traffic shaper (set to the expected ACK rate), limit-cycle-like behaviour was observed: ACKs tend to bunch up, so that some compacted packets carry only one or two ACKs, and some carry nearly all of the ACKs for a window. From this it can be concluded that the shaping system should be more adaptive to the actual current ACKing rate.

Furthermore, if a compacted packet containing many ACKs is lost, much of the ACK clock is lost. If sufficient ACKs are lost, an RTO may be required to restart transmission.

Of course, the larger a packet, the greater the probability of corruption and hence loss. Unfortunately, in this system, with extra information appended per original ACK, it is thus precisely these packets containing most of the ACK clock which suffer the highest corruption-loss probability.

Depending on the exact timing configuration, steady-state throughput rates up to 100 to 200 kB/s (with the 9600 return link) could be achieved. However, during most transfers over the laboratory network, a compacted packet containing most of the ACK clock would go missing, causing a tremendous slow-down.

Clearly, the ACK clock becomes very sensitive to the loss of compacted packets. It should be considered to transport these compacted packets reliably using TCP, rather than UDP encapsulation or IP tunneling. This could actually *decrease* the required header overhead, by enabling the use of VJ header compression. Of course, it does mean that compactor packets become subjected to slow start and flow control.

To retain better spacing between ACKs, and hence between subsequent sender segments, use of an adaptive ACK shaper should be investigated, as well as a data-pacing system at the TCP sender.

A.3 Conclusions

There are many possible approaches, and various subtleties, to implementing ACK compaction in an actual network. Although ACK compaction performs well in simulations [13], there is clearly some work still to be done before ACK compaction will perform reliably in an actual network.

Design of a full network architecture for ACK compaction, featuring congestion and queue management on the forward path, may turn out to be a task of similar magnitude to designing and building a custom paced protocol. However, the latter may deliver better performance and provide more control over the individual aspects of network performance.

Appendix B

Reading TCP Time Sequence Graphs

A TCP time sequence graph is a graphical representation of only one side of the full duplex TCP connection.

The x-axis is the time index of the connection. The y-axis of the graph is the TCP sequence number. As the sequence numbers increase, the graph will increase somewhat diagonally, growing from the lower left to upper right of the graph.

A transmitted data segment is represented by a vertical line, the length of which is determined by the segment size. The vertical placement of the line represents the position of those bytes in the data stream.

The (green) line or curve below the segments reflects the reception of ACKs, while the (yellow) line or curve above the segments represents the upper limit of the receiver's window. Hence, the movement of these two curves represent the movement of the receive window during the transmission.

A retransmitted data segment is a vertical line topped with an R (red).

If SACKs are present in a connection, they will be represented by a vertical line topped with an S (cyan-blue). This line covers the sequence numbers SACKed.