University of Innsbruck

Institute of
Computer Science

Research Group
Distributed and Parallel Systems

Master Thesis

A split connection PEP in ns-2

Author
Philipp Gschwandtner

Supervisor
Dr. Michael Welzl

November 30, 2009

"The most exciting phrase to hear in science
— the one that heralds new discoveries —
is not 'Eureka!' but 'That's funny...'. "

Isaac Asimov

Abstract

The TCP protocol has been defined over 25 years ago. Although there have been updates that increase its performance in a variety of network scenarios, there are still topologies in which standard TCP does not achieve the maximum performance that is theoretically achievable. An example are network topologies that consist of long fat pipes, i.e. links with a high bandwidth-delay product, like satellite links or submarine cables that connect continents. Standard TCP can suffer from underutilisation in such topologies. This thesis discusses the field of performance enhancing proxies (PEPs) for TCP protocols. It describes several techniques on increasing TCP's performance along with a discussion of previously proposed PEPs. Furthermore a survey of a split-connection PEP with different protocol and queueing technique combinations is demonstrated. The results show that a split connection PEP using a separate queue for buffering can increase the performance of the overall system, depending on the queue sizes of the network.

Contents

Chapter 1

Introduction

1.1 Motivation

Today there are a great number of wireless networks with constantly increasing bandwidths. Satellite links can be used for home Internet connections, downstream as well as upstream, often requiring no separate return path via a phone line as it was custom in the past. These technologies provide services for a vast number of end users and not just special institutions, requiring more than proprietary protocols that cannot be deployed on a wide basis. Scientific progress allows for a constant increase of bandwidth for almost all link types and transport media, whereas the link delay is often reduced to little more than the signal propagation delay, hence representing a fixed lower limit that is unchangeable. For this reason, the so-called bandwidth-delay product, an important characteristic of links that are used by TCP connections, also increases, especially for links with high delays such as satellite links.

However TCP's original draft is almost 30 years old, designed to meet the requirements and resources of network topologies at that time. But network hardware has evolved over time, leading to higher bandwidths and the widespread usage of transport media that protocol designers did not consider then. Furthermore the usage of such topologies has changed, since today's industry aims to create technologies that can be used by end users more directly — the aforementioned home Internet connections over satellite links or HSDPA-capable cell

phones are such an example. Of course the original design of TCP has been extended many times, to overcome multiple design flaws and disadvantages, with particular regard to new technologies. However, all these changes and modifications are limited by the very design principle of network traffic, namely protocol fairness. New variants of TCP that are to be deployed on a wide basis need to improve TCP's performance in some area, while at the same time remain fair regarding unmodified versions of TCP that are still being used — in short, modifications need to be backwards compatible with concurrent network traffic. A new aggressive variant of TCP that is able to use all available resources is useless if flows without this modification stall due to starvation.

Furthermore, there are network scenarios which require more than simple protocol extensions to increase the performance on a reasonable level. For these reasons, a huge area of research has been done in the past (and will no doubt be done in the future) regarding performance enhancing proxies (in short: PEPs). Any single mechanism or complete system that increases the performance of network applications could be regarded as a PEP, however usually — as the name suggests — their tasks are carried out by network entities somewhere along a path connecting end hosts, not by the end hosts themselves. Therefore, the main idea is to enhance the performance by performing tasks inside the network, not at the end points.

There are various approaches, ranging from simple one-way mechanisms that re-transmit packets to complete systems that sever the TCP connections of end hosts and use completely different, specially tuned protocols in the network to transmit the application data that needs to be delivered.

This thesis aims to give an overview of existing PEP research, together with an implementation of a simple split connection PEP system that can increase the performance of TCP connections over satellite links. This PEP system is evaluated and compared to standard network scenarios that use unmodified TCP flows (see section 1.3 for a more detailed outline of the thesis and its contents).

1.2 Simulating with ns-2

The performance enhancing proxy discussed in this thesis was implemented and evaluated in the discrete network simulator ns-2[1]. This project, originally a variant of the REAL network simulator, has been developed by multiple organisations including the University of Southern Carolina since 1996 and provides a variety of resources with which to simulate network scenarios, spanning from multiple routing techniques over multicast implementations to wireless and multimedia protocols. While there can be drawbacks of doing simulations with network topologies that include special components or characteristics such as satellite connections (further explained in chapter 5), it is still the main choice to evaluate new network systems. It supports many of today's protocols that are also used in real life, together with an implementation design that allows new protocols to be added or existing mechanisms to be modified easily, while remaining very precise in its resemblance of reality.

1.3 Outline

The thesis is structured as follows: Chapter 2 will give an overview of performance enhancing proxy mechanisms, how to categorise and classify them, explain advantages and disadvantages together with security implications and discuss related work in the field of performance enhancing proxies. Chapter 3 explains the theory behind the PEP mechanism of this thesis in detail and how it works. Furthermore the protocols and queuing techniques that are used are discussed and compared. In addition an explanation of the means of flow control is given (i.e. how to control sending rates). Chapter 4 deals with the simulation results obtained in the ns-2 network simulator. The topology of the simulation setup is illustrated and measurement results are presented and discussed for various scenarios. Chapter 5 deals with the actual implementation of the PEP in the ns-2 simulator and describes the modifications and new configurable parameters.

[1] http://www.isi.edu/nsnam/ns/

Moreover it shows how the PEP can be used in TCL simulation scripts and offers a short documentation of the sample script that is provided with this thesis and that was used for obtaining the measurement results discussed in chapter 4. Finally chapter 6 concludes with the findings of this thesis, summarises the measurement results obtained and provides an outlook for future work on the topic of performance enhancing proxies.

Chapter 2

PEP Overview

This chapter gives an overview of current performance enhancing proxy mechanisms. Section 2.1 explains the different characteristics by which to classify PEPs in general and their effects. Afterwards, existing research and experiments regarding performance enhancing proxies will be discussed in section 2.2.

2.1 Characteristics and Classification

There are several ways of implementing mechanisms which increase the performance of TCP in certain situations without modifying the protocol to a point of incompatibility with existing implementations. These mechanisms can be classified by the following characteristics as discussed in [9].

Layers

Generally, a performance enhancing proxy may operate at any number of layers, however usually their function is restricted to one or two layers. Most of them modify the standard behaviour at the transport level or application level. Nevertheless lower-level PEPs that are working at the link layer — as discussed in [29] — are also possible and in use (e.g. ARQ, FEC, etc.), although they can also worsen the behaviour of TCP. On the transportation layer — for TCP, these PEPs are sometimes called TCP PEPs — one may perform various tasks

like ACK spacing (the modification of the time delay between multiple ACK packets to reduce burst behaviour). On the application layer, a great variety of performance enhancements is in place like web caches or mail transfer agents (MTAs).

Although PEPs situated at the transportation layer or below usually do not modify the payload data of TCP segments, they can still violate the end-to-end argument. The end-to-end argument — one of the main principles of the Internet — states that certain end functions can only be correctly performed by the end systems and not inside the network [31]. This is one of the main reasons why performance enhancing proxies are put only in special network scenarios and not recommended for general use.

Distribution

Another characteristic of PEPs is the distribution fashion. PEPs may work at a single node or they may be distributed among multiple nodes. An example for a single node distribution would be a conversion point between different physical media, e.g. an implementation providing impedance matching between a wired and a wireless link. A possible application for distributed PEPs might be at each end of a link with certain characteristics (e.g. a satellite link).

Symmetry

Furthermore, one can classify PEP implementations as either symmetric or asymmetric. Symmetric PEPs work in both directions, i.e. they apply the same performance enhancing mechanism for packets, regardless of the interface on which they were received or their destination. Asymmetric PEPs act differently, depending on the direction of the packet. The direction can be derived from the receiving interface or from the protocol itself (e.g. the direction of a TCP flow, also called TCP data channel or the direction of the ACK flow, often called TCP ACK channel). Asymmetric PEPs are usually in use where the protocol traffic

is asymmetric, or at network points where the link characteristics change (e.g. a node where wired and wireless links meet).

Transparency

The degree of transparency is another quality by which PEPs can be characterised. A performance enhancing proxy may be fully transparent, i.e. end nodes do not notice the PEP and therefore do not require any modification at any layer in order for the performance enhancement to take effect. Moreover this way, the compatibility with the current network is fully preserved. PEPs with less transparency may require changes at one or both end nodes at one or more protocol layers.

Disadvantages and security implications

Despite the fact that PEPs may enhance the performance of the network, there are also some negative implications that come with their use. In addition to the aforementioned end-to-end violation, there is the principle of fate sharing which is also violated [14, 21]. Normally state information about the connection between two endpoints is only stored at those endpoints, not within the network. This allows for the state information to be separated from the connection path and for a fast protocol recovery. If the connection breaks down, both end-points will be aware of it. However, if certain PEPs are in use at some intermediate node, they may also hold information about the connection. For example if a split-connection PEP is in place, there will be two connections, each between one endpoint and the intermediate node. If one connection for end node A breaks down, end node B still has an open, working connection to the midpoint and may not notice that the path of data delivery has been severed. For this reason, the principle of fate sharing is broken. Moreover if the PEP midpoint breaks down, the connection is also severed, while packets can simply be rerouted upon failure of a node if no split-connection PEP is used.

Furthermore one must consider that a PEP implementation must be able to read information from a packet's header or possibly even modify it. If this is the case for a lower-layer PEP, it does not pose a problem. However some PEPs base their performance enhancing decisions of some layer on the information inside a higher-layer header (e.g. prioritisation of certain protocols for multimedia purposes). For those and for PEPs that generally work at a higher layer, end-to-end security protocols like IPSec that work with encrypted payloads render the PEP useless and must therefore be avoided. If the PEP is not transparent, it is possible to maintain two encrypted connections, however the data will still needs to be decrypted and re-encrypted at the PEP node. This is not only less secure than end-to-end security, it implies that the end users trust the network in terms of security which generally is not the case. Moreover if the two connections are negotiated with different levels of security, end users may act on the false assumption that they know the level of security for the entire end-to-end path. Naturally a PEP can force the same level of encryption for both connections, however — apart from the less security provided, compared to real end-to-end security — this increases the complexity of its implementation [9].

2.2 Related Work

TCP was first standardised in RFC 793 in 1981, providing a number of features to facilitate the use of networks. It supports connection-oriented, stateful communication, reliable and ordered transport of data, the ability to cope with network congestion and it aims to utilise the available network bandwidth to the best of its abilities. But despite the fact that there have been a number of updates to TCP, most of which changed the congestion control or acknowledgement behaviour, there are many use cases and network topologies that emerged from scientific progress, in which TCP's performance is far from the theoretically achievable maximum concerning network utilisation and throughput.

Since changing the basic behaviour of TCP would have either required completely replacing the original or affected TCP friendliness in mixed protocol situ-

ations, it was not an option. Therefore other means of increasing its performance — without changing the standard — have been investigated.

2.2.1 Splitting the Connection

One of the first performance enhancing proxies is described in [5]. Bakre and Badrinath point out the problem of TCP not addressing "the distinctive features of wireless mobile computing". They suggested a new protocol, indirect-TCP or I-TCP, to increase both the performance on wireless networks as well as the support of the mobility of end hosts. Normally, a fixed end host communicating with a mobile device (also called mobile host) pertains a direct connection to it. The problem that arises with such a setup is the fact that TCP assumes packet loss to be an effect of congestion, although there can be other causes (e.g. packet dropping due to wireless cell handovers or corruption). When I-TCP is used, this connection is split in two, with a mobile support router (MSR) in the middle (which connects to the fixed host on behalf of the mobile host, e.g. the fixed host is unaware of the MSR). Their testbed to simulate this modification includes three hosts connected via a 2 Mbit/s wireless radio connection. Their performance evaluation shows a 7% increase of throughput compared to regular TCP for their simulation in local area networks. In wide area networks — with a higher number of hops and thus a longer connection delay — the performance increase is almost 100% over normal TCP. Moreover this performance increase in WANs is tripled if they include cell switching, which presents the second advantage of this system: If the mobile host moves, only the connection on the path to the MSR must be re-established to a new MSR, which gets the complete connection state from the old MSR and resumes the connection to the fixed host. Therefore the fixed host is unaware of such handovers since it keeps communicating with what it believes to be the mobile host. The connection between the MSR and the mobile host can be tuned to the specifics of the wireless medium in use, therefore maximising the performance. In addition, the fixed end host does not need any modification since I-TCP is implemented only at the mobile support routers. The mobile host however needs to be aware of this modification. Furthermore, it should be

mentioned that this approach — as any split connection mechanism — violates the end-to-end principle (see section 2.1).

However, there are two ways of splitting a connection. The approach of Bakre and Badrinath presented in [5] separates the original connection in two, with one intermediate station, since the link characteristics (or in their example, even the transport media) are different for the sender and the receiver. Nevertheless there are situations where the end nodes are situated in similar networks (regarding the link characteristics), but the connection between them uses one or more links with different characteristics or transport media. Hence, the sender and receiver are subject to the same network conditions and it might be counterproductive to use different protocols for them. For this reason, the performance enhancing proxy can be distributed among more nodes than just one, enabling the use of different protocols between the performance enhancing nodes while the protocols at the end nodes remain unaltered.

Durst, Miller and Travis suggest a complete protocol suite with this approach in [15] for space communication, called Space Communications Protocol Standards (SCPS), with four parts — namely a file transfer, transport, security and a network protocol. The transport protocol SCPS-TP they developed is a modification of TCP "to improve the operation in the space environment". For this, they investigated the problems that arise with communication between terrestrial endpoints and those on spacecrafts. Standard TCP assumes that packet loss is an effect of congestion, not corruption. But some degree of packet corruption is usual and expected on wireless links. As a result upon receiving three duplicate acknowledgements (DUPACKs) it reduces the congestion window and thereby drops its sending rate. Their solution for this problem distinguishes between congestion and corruption. For former situations, standard TCP Vegas is used as a congestion algorithm. However if corruption is detected — with lower-layer mechanisms measuring the number of packets that were received but corrupted — the receiver sends ICMP messages holding a *corruption experienced* flag as well as acknowledgements with the same option. During this phase, TCP will not back off due to the alleged congestion but keeps sending at its current rate.

If packet loss is still indicated by DUPACKs without the option set, SCPS-TP switches to TCP Vegas' congestion control. If no congestion is encountered or congestion is experienced (indicated by the arrival of DUPACKs) but without the *corruption experienced* information, SCPS-TP behaves like normal TCP Vegas. A performance evaluation of the modifications was done by simulating a link with a bandwidth of 1.5 Mbit/s and delays of 50 and 100 ms together with bit error rates between 10^{-5} and 10^{-8}. The results indicate an increasing throughput advantage of SCPS-TP over TCP when increasing the error rate. While SCPS-TP manages to keep a performance of roughly 90% when increasing the error rate by a factor of 10^{-3}, TCP drops its performance by a factor of 10. Furthermore, doubling the delay from 50 to 100 ms has very little impact on SCPS-TP, but costs standard TCP 25% of its throughput. Their results indicate that the main reason for TCP's poor behaviour in error prone environments is its false assumption that packet loss is always caused by congestion. In addition, they suggest that non space-related wireless and mobile links might also benefit from their modifications.

Moreover there are split connection approaches that offer more complex enhancements than just separating a TCP connection into two or three connections. The approach proposed in [16] by Ehsan et al., for example, uses two split connections and works on the application layer. An HTTP server is connected to an intermediate node via the Internet, and this node is connected via a one-way satellite link to a node sending HTTP requests over a phone line return path. However, contrary to the split connection approaches priorly discussed, the authors included an application level cache at the intermediate node (in their case, a web cache). They suggest that in topologies that include long fat pipes with comparatively high channel error rates such as satellite links, TCP often suffers from high sensitivity to random packet losses, even with a split connection mechanism in use. To mitigate this effect, they install a cache on the intermediate node. If a sender establishes a connection to the cache and transmits an HTTP request and the requested document is present in the cache (cache hit), it is directly transmitted to the sender without involving the server. However if the requested document is not present or outdated (cache miss), the intermediate node will

establish a connection to the server, fetch the requested document, cache it and transmit it to the sender. Therefore the connection is not only split at the transport layer, but also at the application layer, since the HTTP requests from the sender are explicitly sent to the cache, not just intercepted by it. Likewise, the server gets incoming connections and requests explicitly from the cache. Measurements were taken using a commercial satellite connection of up to 24 Mbit/s downstream and a regular phone line return channel of 56 Kbit/s upstream. The results show various performance increases regarding throughput ranging from 40% up to 140% for cache misses and up to 170% for cache hits. In addition, larger files show a higher benefit gain. Measurements regarding the number of simultaneous connections between the requesting node and the cache show smaller performance gains for increasing number of connections, however for any number of connections the PEP proposal of Ehsan et al. still offers performance gains compared to using a single connection without any cache involved.

Caini et al. evaluate the performance of PEPsal, a transparent connection splitting TCP PEP designed for satellite connections, in [13]. PEPsal is open source and according to the authors in use by a commercial satellite Internet provider. Their topology of interest consists of two senders connected via a link with a short RTT to a midpoint node, which is again connected to a satellite receiver via a link with a long RTT and a wired receiver via a link with a small RTT. One of the senders transmits data to the satellite receiver, one transmits data to the wired receiver, therefore the link between the senders and the midpoint node is their common bottleneck. The PEP they propose transparently splits the TCP connection from the sender to the satellite receiver and uses TCP Hybla, an enhanced TCP variant for satellite links developed by the authors, as the transport protocol between the midpoint and the satellite receiver. TCP Hybla was first proposed in [12] and grants connections with a large RTT the instantaneous transmission rates of comparatively fast reference connections. Furthermore they investigated the effect of congestion on the link that connects the senders and the midpoint node, which has the same bandwidth as the satellite connection. They compared this proposal to using non-split NewReno/SACK TCP and TCP Hybla connections, together with simply splitting a SACK TCP connection (i.e.

without the use of TCP Hybla for the connection to the satellite receiver) for various round trip times between 50 ms and 600 ms. Their results regarding throughput show that without any errors on the satellite channel, a non-split NewReno/SACK TCP connection deteriorates by a factor of up to 10 times with increasing RTTs in the presence of congestion on the bottleneck link. According to the authors, the main reason for this degradation is the fact that standard TCP congestion control is only fair regarding bandwidth utilisation, if all competing flows have roughly the same round trip time. Since having background traffic from a sender to the wired receiver with a very small RTT compared to the RTT of the connection using the satellite, this fairness is no longer given. Introducing errors on the satellite channel leads to even worse performance of NewReno/-SACK and also decreases the performance of using split connections with SACK TCP, while both PEPsal and Hybla TCP are only marginally affected. When encountering congestion in addition to random losses due to errors, TCP Hybla's performance also decreases slightly for RTTs higher than 400 ms while PEPsal is able to maintain its throughput.

There has been some more research pertaining to satellite connections. RFC 2488 [4] and RFC 2760 [3] discuss not the technique of splitting connections but rather the parameters of an actual TCP connection over a satellite link or links with high bandwidth-delay products in general. They suggest modifications for the start-up phase like eliminating the delay caused by TCPs three-way handshake. Further proposals include values for initial window sizes or disabling delayed ACKs since they decrease the growth rate of TCPs congestion window (although resulting in a higher utilisation of the return path). But changes with a much higher impact are the use of timestamps [23] together with TCPs sequence numbers to allow very high bandwidths and the introduction of the TCP window scaling option as discussed in [27]. First of all, since the maximum lifetime of an IP packet was originally assumed to be 120 seconds, a TCP sequence number cannot be reused for this time frame to ensure that no two segments with the same sequence number are present in the network. Since sequence numbers are represented by 32 bits in the TCP header, spreading them over 120 seconds gives a maximum data rate of only 286 Mbit/s. Using timestamps in conjunction

with sequence numbers to identify a segment, this limitation is removed. Second, standard TCP window sizes cannot exceed 64 KBytes, which limits TCP's effective throughput to about 1 Mbit/s (assuming a round-trip-time of 0.5 seconds for GEO stationary satellites) [26]. Further research includes new start-up and congestion control mechanisms that use low priority dummy segments to probe satellite network conditions even if there is no actual data transfer [2], or using the Packet Pair algorithm to monitor link utilisation and reflect it in the advertised windows for TCP senders using the PEP [32]. In [25], Marchese et al. proposed PETRA, a PEP transport architecture for satellite communications, which uses a special transport protocol (STPP) for satellite links while keeping standard TCP interfaces at the ends of those links for transparency. In addition PETRA divides the transport layer into two sublayers to separate flow control and the maintenance of the end-to-end semantics of TCP connections. Their results show performance increases between 20% and 400%, depending on the channel error rate and network conditions.

Also PEP research pertaining to different fields of application than satellite communication has increased over the last few years, due to the spread of technologies like UMTS, WLAN and Bluetooth. Fiorenzi et al. suggest in [17] that the lack of knowledge of HSDPA link characteristics is one of the main performance concerns for TCP in such topologies. They propose a split connection approach together with cross-layer signalling, to inform a proxy that relays the TCP payload of a sender of the wireless network conditions and let TCP reflect them in its scheduling and sending rate decisions. Their experiments in ns-2 for an HSDPA link with a mean maximum bandwidth of about 1.3 Mbit/s show a well-increased throughput compared to unmodified Reno TCP connections, which can occasionally break down to a sending rate of 0 Kbit/s while their PEP is able to maintain an average throughput of about 1.1 Mbit/s.

2.2.2 Other Approaches

However splitting connections (whether done in combination with additional enhancement mechanisms or not) is not the only option to improve TCP's per-

formance over links with distinct characteristics. Since splitting the connection practically always violates the end-to-end principle, no matter the measures taken to mitigate this problem, further research has been done in the field of PEPs that use other means to improve performance.

Balakrishnan et al. reach similar results in [7] as Bakre and Badrinath did with I-TCP in [5], but comply with the end-to-end argument because they do not split the TCP connection. Instead their approach, called *snoop*, relies on local retransmission. Their topology of interest consists of a fixed host which is connected to a wireless base station via Ethernet. The base station in turn is connected wirelessly to a mobile host via an AT&T WaveLan card supporting 2 Mbit/s. The *snoop* mechanism introduces a data cache sitting at the wireless base station. It maintains copies of TCP segments coming from the fixed host while routing the original segments normally to the mobile host. But in case of a duplicate acknowledgement or a timeout (i.e. no acknowledgement arriving within a certain time frame), segments from the cache can be re-sent without involving the original sender. This is completely transparent from the sending host's point of view and hides possible packet loss on the wireless link. Performance evaluations were done with the REAL[1] network simulator, with a wireless bandwidth of 2 Mbps and a cache size of 20 packets. The results show that the caching and retransmission of segments either can increase the throughput by a factor of up to 20 with wireless bit error rates ranging between $5 * 10^{-7}$ (one erroneous bit every 2 megabit) and $1.5 * 10^{-5}$ (one bit error every 64 kilobit), or it can cope with up to 20 times higher error rates compared to normal TCP.

The advantage of this approach compared to that of I-TCP is the decreased complexity of packet processing required at the midpoint. I-TCP needs to acknowledge a segment, re-pack the payload in a new packet and transmit the new packet, whereas the protocol proposed by Balakrishnan et. al. in [7] only needs to copy the segment to a cache and possibly retransmit it without any further processing. They also compare it with established retransmission mechanisms such as the fast-retransmit technique discussed in [11] and find that fast-retransmit is

[1] http://www.cs.cornell.edu/skeshav/real/

less effective since it only deals with wireless handovers, not error characteristics of wireless links. Another reason is the need for the packets to be re-sent from the source and not some point closer to the error prone wireless link, introducing unnecessary delay. Furthermore they suggest that the use of link-level retransmissions in parallel with end-to-end retransmission protocols can greatly decrease performance on links with higher error rates if they are not highly coordinated.

PEP research other than split connections and local transmission include simple scheduling modifications. In [8], Pravin Bhagwat et al. also study packet burst errors in wireless LANs and their effect on the performance of TCP. They point out that when a wireless channel that is used for transmitting data is in a burst error state, lower layer protocols may try to retransmit lost frames multiple times, resulting in poor link utilisation (since immediate retransmissions during a burst error period are very likely to fail again). To illustrate this phenomenon, one of their experiments included a file transfer of approximately 600 Kilobytes between two machines, both connected to an intermediate station via infrared wireless LAN and Ethernet respectively. Under normal, no-loss conditions the file transfer took approximately 10.5 seconds, resulting in a throughput of 0.508 Mb/s. In their simulations including packet loss, the wireless channel switches to burst error mode 3 times (at 4.3, 5.8 and 12.8 seconds). This yields in a decreased throughput of 0.321 Mb/s, which is almost 40% less. They found the main problem source to be TCP's long delay when sending retransmissions and resuming slow start, for cases in which lower level retransmissions fail during burst error periods. As a result, TCP reacts by exponentially backing off its higher level retransmission, therefore resulting in the poor utilisation mentioned above.

Contrary to using split connections however, they suggest the use of a new packet scheduling mechanism to mitigate this problem. Their scheduler includes the state and characteristics of the wireless channel in its packet scheduling and dispatching decisions. For radio-based wireless LANs, their channel state dependent packet (CSDP) scheduler maintains queues for each wireless destination. When packet loss occurs due to bad wireless channel conditions, retransmissions

of lost packets are delayed until the conditions return to normal. However while packets to this destination are delayed, packets to other destinations can be transmitted during this period, since they found that wireless channels to different destinations are statistically independent. Their experiments with the new CSDP scheduler show an overall approximate performance increase of 15%.

A different approach to enhance the performance of TCP is called Ack Congestion Control (ACC). In [6], Balakrishnan et al. proposed a congestion control mechanism for acknowledgement segments in their discussion of the performance of asymmetric TCP connections. An example for an asymmetric connection would be a home Internet connection that uses a satellite for its downstream path, but a dial-up modem for the upstream path. This is quite frequent for home satellite connections since the sending equipment is expensive compared to the dial-up modem solution which requires just a phone line. The bandwidths of the directions could differ as much as 10 Mbit/s and 28.8 Kbit/s. The authors define a bandwidth ratio k between the ratio of the bandwidths of both directions (forward for data, reverse for acknowledgements) and the ratio of the packet sizes (e.g. 1000 Bytes for data segments, 40 Bytes for ACKs). For example a downstream/upstream bandwidth ratio of 100 and a packet size ratio of 25 would result in $k = 4$. This means that more than one ACK every 4 packets would lead to congestion on the reverse path, which in turn limits the growth of the sender's congestion window.

One method of mitigating this effect is header compression. It can reduce the size of ACKs considerably (assuming they do not carry payload data) and free up bandwidth. Another way of solving the problem is called *ACK filtering*, which is similar to a split connection approach. The two nodes at each end of the bottleneck link could group enqueued ACKs of the same connection together and delete the original ACKs, therefore acknowledging more data with a single ACK. Upon arrival of such a cumulative ACK, the second node reconstructs the original ACKs and forwards them to the sender. This way the required bandwidth decreases. The third solution proposal — designed for two-way TCP flows (i.e. the ACKs for the reverse path also carry payload data) — is simple but effective.

The authors suggest that congestion control should be used for these ACKs as well, similar to the standard congestion control of TCP senders to reduce burst effects. Simulations were done with a bottleneck link that supports 10 Mbit/s upstream and 9.6 Kbit/s or 28.8 Kbit/s downstream. Their results generally show throughput performance increases between 4% and 300%. Without header compression, the use of ACC can double the achieved throughput. Ack filtering provides even higher performance increases with a factor ranging between 2 and 4. Header compression itself also greatly increases the achievable throughput, up to a factor of 4. However either systems used in conjunction (ACC + compression or filtering + compression) only double the performance compared to the single enhancement or show almost no performance increase at all. Generally speaking, the topology with the slower 9.8 Kbit/s return path shows higher benefits from the enhancements compared to the faster 28.8 Kbit/s setup.

Chapter 3

Theory

This chapter discusses the theory behind the performance enhancing proxy implemented during the course of thesis. Section 3.1 gives a basic explanation of its operation and characterises the PEP. Afterwards section 3.2 gives an understanding of the flow control mechanism. Finally section 3.3 discusses the protocols and section 3.4 the queues that are used.

3.1 General

The PEP implementation of this thesis is intended for enhancing the performance of TCP connections that share a common bottleneck link with a high bandwidth-delay product. An extreme example for such a link would be a satellite connection. Geostationary satellites, as used for TV broadcasts and global communications in general, orbit the Earth at an altitude of approximately 36.000 km. The main advantage of this GEO (Geostationary Earth Orbit) is a constant and relatively large field of coverage, hence eliminating the need for protocols that can cope with connection breaks. The coverage of satellites on a lower orbit (e.g. GPS, satellite phone systems, etc.) changes with their position, a fact that needs to be taken into account when designing and deploying protocols for them (e.g. handovers).

In addition, GEO satellites do not need to be tracked by antennas or dishes since they appear to remain in a fixed position when viewed from Earth (hence

greatly reducing costs for Earth-based communication equipment). The disadvantages of GEO satellites are their increased transmit power requirements (and more sensitive receiver equipment) due to the large distances involved. However low-level factors like error characteristics, modulation techniques or power requirements — because of having little or none impact on the transport or application level — would go beyond the scope of this thesis and are therefore neglected.

Another problem that arises with connections that span over such large distances is the delay. The signal propagation delay that covers the distance from a ground station directly below to a GEO satellite and back is at best 240 ms (assuming the speed of light in vacuum as the propagation speed and an overall distance of 72.000 km) — neglecting any processing time needed by the satellite to perform tasks like error correction. Usually, one can assume propagation delays of 250–280 ms, resulting in a round trip time of up to 600 ms for bidirectional satellite links [1]. Such high delays and RTTs have a considerable impact on the convergence of protocols which use a congestion response that depends on the RTT, like TCP (see section 3.3).

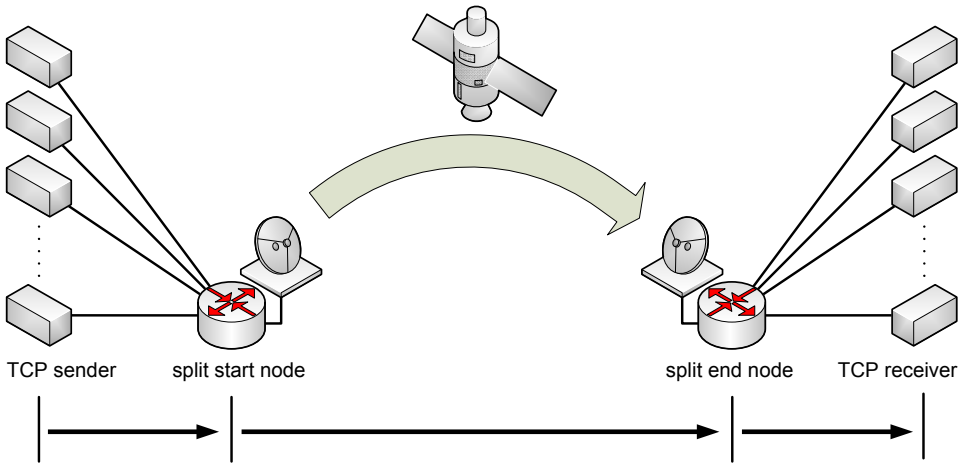

Figure 3.1: The topology shows connection splitting performed by the two split nodes.

The main technique of the PEP presented in this thesis to increase the performance over such a link is based on the split connection approach. Instead of a TCP flow spanning from the end sender to the end receiver, the connection is split into three separate connections (as illustrated in figure 3.1) when the PEP is in use. The sender sends its packets to the split start node, which — aside from immediately acknowledging the data — extracts the payload data. This data is then retransmitted like normal application data from the split start node to the split end node. The split end node again acknowledges the segment and forwards the payload in a new packet to the end receiver. To be able to determine the original destination of the data at the split end node, the source address of incoming packets is saved into the PEP id table. When the data is ready to be transmitted at the split start node, the id is removed and saved in a header field of the outgoing packet. Hence the split end node can determine its intended target. This forwarding mechanism is illustrated in figure 3.2. Furthermore it should be mentioned that regardless of the number of connections held by the senders and receivers, the PEP always uses a single TCP connection over the satellite link.

3.1.1 Classification

According to the classification provided in section 2.1, the PEP acts on the transport layer, it is distributed, asymmetrical, and non-transparent for the following reasons:

Both split nodes take incoming packets, extract the payload data and forward it without any further modification or processing. Thereby the PEP operates at the transport layer only — although the payload is copied at the application layer, there is no modification or further performance enhancement done and therefore this layer can be neglected. Since the approach splits the original TCP connections in three and both end senders and receivers are unaware of any packet processing by the split nodes, the PEP might be extended in the future to support further tasks and enhancements at the application layer (e.g. compression).

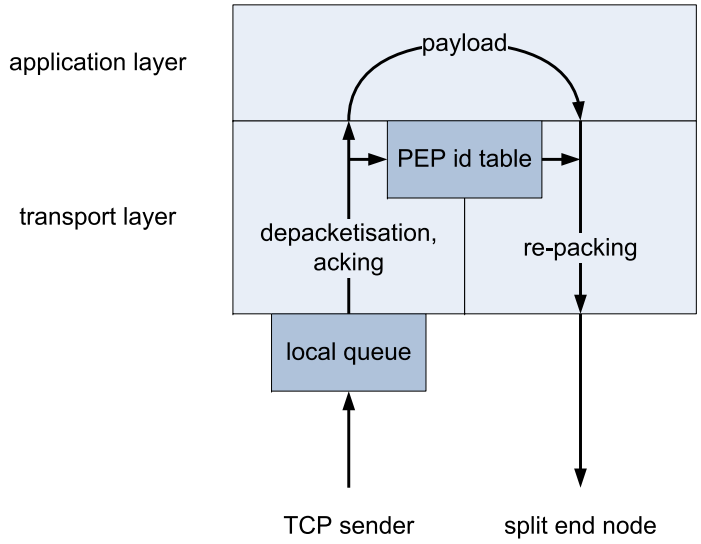

Figure 3.2: The main packet forwarding behaviour of the split start node.

Since the connection is split and the split start node immediately sends ACKs to the TCP sender upon the reception of packets, the split start node is responsible for the delivery of the data. While the general approach can be realised in a transparent or a non-transparent manner, the TCP senders of this ns-2 implementation of the PEP connect explicitly to the performance enhancing proxy, therefore the implementation of the PEP is not transparent. But nevertheless the sender has no information about the successful delivery of its data since it gets acknowledgements for the reception of payload that has not reached its intended destination yet. For this reason the PEP (more precisely the nodes at which the PEP is situated) must ensure safe data transport, which is guaranteed by TCP itself. Hence there is no need for additional measures to ensure guaranteed data delivery.

Still, as mentioned in subsection 2.2.1, this approach violates the end-to-end argument since the responsibility of guaranteed data transport is no longer carried by the end nodes alone. Furthermore the principle of fate sharing is also broken since splitting a TCP connection also results in storing state information

about the connections inside the network, not just the end nodes. For example the connections between the TCP senders and the split start node could be severed but the connections between the split end node and the end receivers are still valid. Despite the fact there is no more data arriving in such a case, the end receivers have no further information about the state of their respective sender's connections (which would not be the case for unsplit connections spanning between the senders and receivers).

The end nodes do not need any modification regarding their protocols. Since the connection is split in three, the protocol used between the two split nodes does not necessarily need to be the same as the one used by the end nodes. Due to the fact that the payload data at the split nodes is extracted and re-enveloped into new packets, the protocol on the split link can be exchanged for some other that meets the specific requirements of a link with a high bandwidth-delay product (or, for other applications, links with other special requirements). This will be further discussed in section 3.3.

The approach itself is asymmetrical, since it only deals with one-way flows regarding the transport of data. However there is no restriction that precludes installing the PEP twice, one for each direction. Since TCP connections are fully isolated from each other on the TCP stack of a network device, two PEPs installed for opposite data directions can be used in parallel and should not interact.

Finally, the PEP is distributed since it must be deployed on two nodes, at each side of the bottleneck. Non-distributed PEPs that use two split connections have been discussed in 2.2.1, however they are mostly used in network scenarios where error-prone links pose the main problem regarding performance. However transmission errors are not a concern for the scope of this thesis and are assumed to be dealt with by lower-layer mechanisms not requiring retransmissions (e.g. forward error correction). By this reason and due to the fact that a split connection approach resulting in two separate connections instead of three requires a protocol change either at the sender or at the receiver, those techniques are not an option for the scenario discussed in this thesis.

3.2 Flow Control

Explicit flow control of the PEP is realised only at the split start node, since the split end node simply forwards the data to the end receivers and the split link is considered to be the bottleneck. Also because the split link is the main concern regarding link utilisation and the split start node is the point of merging the data of multiple flows into a single TCP flow, only the split start node needs an additional local queue to buffer data. This queue is logically placed in front of the split start node and enqueues packets before they are received by the receiving TCP stack of the split start node. If flow conditions allow it, packets are dequeued and processed by the receiving stack. The queue drops packets according to its mechanism (depending on the queue type, e.g. drop tail, RED, etc.). However this setup still lacks a form of control to regulate the rate of packets that are dequeued (since there is no physical link). To control the packet rate that is accepted at the split start node, the sender of the satellite connection needs to be involved since we need information on when to send how much data — if any.

For this reason, the TCP sender of the split start node holds a value *buffer* which is similar to a receiver window. Whenever the sender transmits data, *buffer* is increased by the number of bytes that have been sent. If there are packets in the local queue, and *buffer* is at least the size of one packet, packets can be dequeued and handed to the receiving TCP stack of the split start node while *buffer* is decreased by the size of each packet that has been dequeued. If *buffer* is smaller than the size of one packet, the sender waits a predefined time interval (see chapter 5) — by default 1 ms — and checks again. When packets are handed to the receiving stack, they are processed and acknowledged without any modification (i.e. standard TCP behaviour). Hence, *buffer* approximately represents how much data can still be taken into the buffer of the TCP stack of the split start node.

This mechanism basically relies on TCP to determine how much data can be processed. If the split start node is able to send more data, then in turn more data

can be accepted. If the sending rate on the split link decreases — for whatever reason — the number of packets accepted for forwarding at the split start node is also decreased. Since this is realised by a simple queue that drops packets, the TCP end senders will encounter packet loss and adjust their sending rates accordingly (independent of the actual queuing mechanism in use). Therefore, the system is self-regulating, depending only on the data flow rate on the split link.

3.3 Protocols

An important factor that affects the performance of a PEP deployed under extreme network conditions (like a satellite connection) is the protocol that is being used. Since it can easily be exchanged on the middle link without modification on the end-to-end nodes, it stands to reason to compare protocols regarding their performance on long delay links with high bandwidth. In the following subsections, Standard TCP and CUBIC are explained and compared with each other.

3.3.1 TCP

Standard TCP uses two thresholds by which it limits its sending rate — the advertised window of the receiver and the congestion window *cwnd*. The first is disclosed in the acknowledgements sent back to the sender. The second is maintained by the sender itself, representing the network conditions as stated below. Hence, the maximum sending rate is the minimum of those two thresholds.

To increase its sending rate, TCP uses two mechanisms, depending on the situation (for the purpose of this discussion, we assume that the advertised window is very high and thus not a limiting factor). When a TCP flow starts sending application data (after the three-way handshake), standard TCP uses *slow start*. As packets are sent, the sender receives ACKs acknowledging them. For each ACK received, the sender sends two packets — therefore, despite the name of

the mechanism, the growth rate is exponential. This continues until a threshold *ssthresh* (usually 64 KB) is reached, after which the growth rate slows to the *congestion avoidance* phase.

During congestion avoidance, TCP increases its sending rate by determining a new *cwnd* value as follows:

$$cwnd = cwnd + MSS * \frac{MSS}{cwnd}$$

MMS denotes the maximum segment size. The equation shows that the growth rate is linear and dependent on the round trip time (*cwnd* is approximately increased by one packet every RTT). However, as soon as packets are dropped (i.e. duplicate acknowledgements (DUPACKs) arrive or the sender's retransmission timer expires), *cwnd* is reset to 1, *sstresh* to half the window size and TCP falls back to slow start again.

Since this means that TCP effectively performs a restart and starts probing for network resources. However arriving DUPACKs indicate that — although at least one segment was lost — segments are arriving at the receiver. Therefore, a new variant called Reno uses this information. It sets *sstresh* and *cwnd* to half of the bytes in flight (i.e. travelling on the path), immediately retransmits the lost segment and increases *cwnd* for every arriving DUPACK (including the three DUPACKs that indicated packet loss in the first place) since a DUPACK means that a segment has left the network. Upon arrival of a normal ACK (acknowledging the receiver's reception of a packet previously transmitted), the congestion window is set to *sstresh* and Reno continues with its congestion avoidance phase. Figure 3.3 illustrates that behaviour (note: since the advertised receiver window is very high and not a limiting factor, slow start is not visible).

Since congestion is indicated by DUPACKs or expiring timers, the round trip time (and its determination) is crucial. For smaller RTTs compared to larger ones, the retransmission timer can also be lower, resulting in a faster detection and recovery from packet loss (meaning the opposite effect for larger RTTs). However high round trip times limit TCP's progress even further, since the send-

ing rate is a function which is time-dependent on the reception of ACKs. Since the arrival of ACKs is delayed with higher RTTs (since the delivery of both data segments and their ACKs takes longer for higher delays), the rate increase of TCP is effectively slowed.

To improve the performance of TCP over links with high bandwidth and long delay, a new mechanism called SACK was introduced in RFC2018 [18]. It basically provides TCP receivers with the means of notifying senders of the arrival of specific segments, thus allowing the senders to keep a so-called scoreboard of which specific segments have arrived — compared to the old variant where only the last successfully received segment was known. Furthermore it holds an explicit variable for estimating the number of bytes in flight. These changes allow TCP senders to transmit more than just one missing segment per RTT (as TCP Reno does), therefore clearly providing benefits for long fat pipes. For this reason, it is used as the standard TCP variant for all the performance measurements done in this thesis.

3.3.2 CUBIC

To mitigate the fact that TCPs achievable throughput is greatly affected by the round trip time, new means of controlling the sending rate have been discussed. CUBIC is such an example, since its window growth is a function of the maximum window size (explained below) and the time of the last packet loss — not the round trip time [30].

The growth rate function of CUBIC — as the name suggests — is a cubic function, as illustrated by figure 3.3. The basic idea is to probe the network bandwidth and upon packet loss to remember the maximum window size w_max during that event. Later on, when increasing its rate, CUBIC approaches this limit with a decreasing growth until w_max is reached (in figure 3.3 just below 500). Above that, CUBIC increases its growth rate again. The slow growth when $cwnd < w\_max$ ensures the stability of the protocol, the fast growth for higher values ensures the scalability and utilisation of available bandwidth.

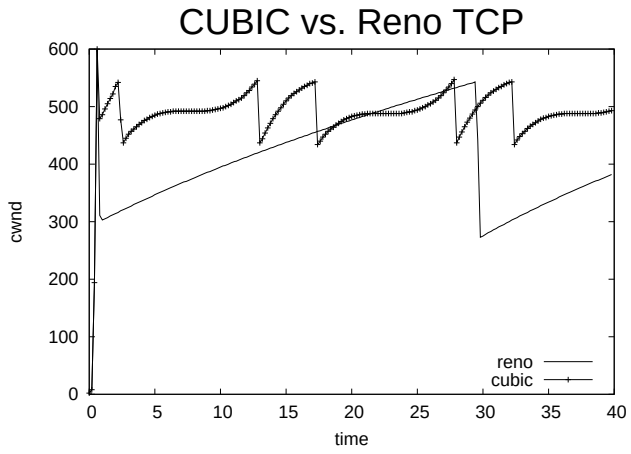

Figure 3.3: The congestion window of CUBIC and Reno TCP over time.

3.4 Queuing

Queues are means of the network topology and its entities to deal with the unsteadiness of today's traffic or more specific, "to compensate for sudden traffic bursts" [33]. The Internet provides the user with a best-effort strategy — if there are enough resources on a link to accommodate a packet, it is forwarded on this link. However when the resources do not suffice, the packet must be either buffered or dropped. Since immediately dropping the packet would basically mean that we can only cope with single packet traffic bursts and that it would ultimately result in very small link utilisation, packets are generally buffered in queues (based on a first-in-first-out strategy). By reason that a good portion of Internet traffic is usually based on spikes (e.g. an HTTP request followed by the transfer of an HTML web page), queues enable routers or other packet forwarding entities to achieve a better link utilisation and compensate for traffic bursts. During traffic spikes, the packets can be buffered and during phases in which the link would normally be underutilised, these packets can be forwarded.

However there has been a great deal of research in the area of queues sizes and what mechanisms and strategies to use.

The right queue size has been a topic of research for some time. Ordinarily, the alleged solution is simple: Since using queues results in better link utilisation together with a more smooth form of traffic, the queue size should be as long as possible. However the issue is more complex. Naturally it is impossible for queue sizes to be infinite due to hardware constraints. But even very long queues that can buffer an extreme amount of packets are not a good idea. Apart from hardware costs of the memory needed inside routers, there are several network implications as discussed in [33]:

First of all, queues introduce delay. Depending on the length of a queue, packets may be delayed long enough for it to have an impact on the performance of the end applications (e.g. real-time traffic of a video conference). For this reason, the queue length should rather be shorter than longer — however the limits depend on the link the queue is used for as well as the complete path of the connection. A queuing time of 10 ms may be relatively huge for a link with a delay of 1µs, however if the complete path of the TCP flow has a one way delay of 100 ms, it might be acceptable.

Second, the queue will be filled by TCP senders up to its limit in any case, no matter how long the queue is. Since protocols like TCP use packet loss as an indicator for congestion, they will increase their sending rate until that point is reached. Furthermore since queues delay the effect of congestion (i.e. it is possible to increase the sending rate over the link's bandwidth without congestion for a limited period of time since the queue can buffer these additional packets), the convergence time of TCP flows (e.g. the time it takes for a flow to adjust its sending rate to a point where the link utilisation is acceptable) increases in proportion to the length of the queue. Furthermore if the aforementioned increase of the delay of the path is high enough it will slow the speed of convergence even further (since the sending rate of TCP is a function of the arrival of acknowledgements and therefore of the round trip time). While suggestions in the past were to set the queue size to some value between the bandwidth-delay

product and twice the bandwidth-delay product (which should approximately be the bandwidth-RTT product), these guidelines may not be applicable for today's links that pose similar signal propagation delays as in the past, but provide a much higher bandwidth — therefore leading to comparatively large queue sizes. Therefore, various suggestions have been made to reduce the queue size to smaller values. This is one of the reasons why this thesis investigates queue sizes up to the bandwidth-delay product [33].

3.4.1 Drop Tail

Apart from the length of the queue, the dropping strategy is another major field of research. The simplest form is the standard first-in-first-out approach that drops new packets when the queue has reached its limit. Such queuing strategies are called "drop tail" or "tail drop". While they are easy to implement and execute, it is not always a good idea to use this simple approach since there are two main drawbacks: Firstly, it is possible for one or more flows to exclusively utilise the queue which leads to starvation of other flows. This phenomenon is caused by the synchronisation of flows or other timing effects. The second disadvantage is also the main point of the strategy: Arriving packets are dropped if the queue is already full. However due to the bursty nature of some traffic (e.g. web traffic), this can lead to the loss of multiple packets in a row. In turn, the sender will back off only to increase its rate later on when the queue is emptied. This again leads to a burst behaviour of the sender and furthermore to a synchronisation of multiple flows regarding their traffic spikes. The result can be an underutilised link, which is the exact opposite of the intended effect of using a queue [10].

In [24], Kevin et al. have done some research regarding the length of a drop tail queue. Their experiments include a dumbbell topology testbed with a 10 Mbit/s bottleneck on which queuing has been investigated. All nodes on one side of the bottleneck act as HTTP request generators, therefore simulating web browsing by users, the nodes on the other side of the bottleneck act as web servers, generating and sending HTTP responses. They calculated the bandwidth-delay product of the bottleneck to be approximately 96 packets, and used queue sizes between 30

and 240 together with network loads between 80% and 110%. They measured the cumulative response probability in relation to the response time (i.e. how many requests completed within intervals between 0 and x milliseconds). Their results show that there is very little difference between the queue sizes at 80% load. But for example at 98% load, the queue size with the highest number of successful responses under 500 ms was 30, the lowest setting (approximately one third of the bandwidth-delay product). Increasing it only increases the number of successful responses for higher intervals than up to 500 ms, at which point 190 seems to be the best queue size (approximately 2 times the bandwidth-delay product). However for 110% load, setting the queue size to 60 (approx. two thirds of the bandwidth-delay product) seems to achieve the best results. Overall the highest performance gains (regarding the number of successful responses within a certain time interval) are about 20%.

3.4.2 RED

To mitigate these problems and allow queuing that achieves better throughput and fairness results, a more sophisticated solution is necessary. Random Early Detection (RED), proposed in RFC2309 [10], is such a queuing mechanism since its decision is not based on a binary state (i.e. "queue is full" or "queue is not full") but rather on a probability function. The basic idea behind the system is to drop more packets as the queue length increases. This behaviour is illustrated in figure 3.4.

As long as the queue length is shorter than *min_thresh*, no packets are dropped. If the queue length is larger than *min_thresh* but still shorter than the limit represented by *max_thresh*, packets are dropped with the probability p. As figure 3.4 shows, p increases as the queue length increases, up to a maximum packet dropping probability of *max_p*. If the queue length is longer than *max_thresh*, p is 1 and therefore every incoming packet is dropped until the queue length decreases below that limit.

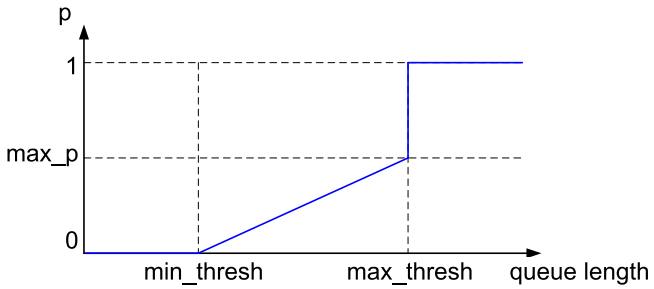

Figure 3.4: The dropping strategy of RED. p denotes the probability of dropping a packet, *min_thresh* and *max_thresh* represent queue length limits for the mechanism.

However it is important to note that RED does not use the instantaneous queue length for its decision making but rather an average queue length that has been determined over a time interval. Thus, if the queue has been almost empty in the past, RED does not immediately drop packets if *max_thresh* has been reached. This also means that *max_thresh* should represent a value that is smaller than and not equal to the maximum queue size to still allow some packets to be enqueued in such cases.

One of the most challenging problems of RED queues is setting the parameters. As discussed for the drop tail mechanism, the queue size is an important setting that can affect the performance. And since RED is more sophisticated and more complex, there are more parameters that can or must be configured properly to achieve good performance. Table 3.1 lists the most important RED parameters. The following suggestions for setting these parameters are documented in [19]. The authors recommend default values, but the optimal values are widely unknown since it depends on the topology and traffic characteristics of the specific scenario. Therefore the RED implementation in ns-2 allows for an automatic configuration of its parameters (the use of which is recommended in [19]) with respect to the given link characteristics (bandwidth and delay) and a specified target queue delay (by default 0.005 seconds).

Table 3.1: Configurable parameters of RED.

Name	Description
length	The maximum length of the RED queue
min_thresh	Threshold above which packets are dropped probabilistically
max_thresh	Threshold above which every packet is dropped
q_w	Weight factor for calculating the average queue length
max_p	Maximum probability of dropping packets
gentle	Toggles gentle mode of RED (boolean)

The optimal value for *min_tresh* depends on the link properties. Generally setting it to something very small like one or two packets is not suggested since it will lead to very early dropping, opposes burst traffic (which should be smoothened instead) and can lead to low link utilisation. When using RED's automatic configuration, it is set to five packets unless the target delay of the queue and the bandwidth of the link require a higher value, by performing the following calculation:

$$min\_tresh = max\left(5, \frac{target\_delay * ptc}{2}\right)$$

ptc is a constant packet count value, which resembles the maximum number of packets that can be placed on a link with a given bandwidth per second. *target_delay* denotes the desired queueing delay for packets and is set to 5 ms by default. Therefore, using links with higher bandwidths or working with longer queueing delays increases the average queue size of RED queues. The suggestion for the maximum threshold value is $max\_thresh = 3 * min\_tresh$ and should also only be changed if the queuing delay is non-trivial. Moreover the default setting for *max_p* should be 0.1 according to [19], since investigations have shown that end-to-end packet drop rates range somewhere above 5%. A higher value is not suggested, by reason that a general packet drop rate above 10% might suggest to end systems that it is a result of network problems rather than just active queue

management. Finally w_q was originally suggested to be 0.002, however RED's automatic configuration also determines this parameter by evaluating

$$rtt = 3 * \left(link\_delay + \frac{1}{ptc} \right)$$

$$w\_q = 1 - e^{\left(-\frac{1}{10 * rtt * ptc} \right)}$$

where *ptc* again denotes the packet count value as described above and *link_delay* is the one-way delay of the link. If the chosen value for w_q is too small, the queue will react too slowly on changing traffic conditions. If it is too large, the computed average queue length will converge to the instantaneous queue length, therefore possibly reacting too sensitive to traffic bursts. By letting RED automatically configure w_q, the mechanism ensures a feasible queue weighting with regard to the link's throughput, meaning a slower adaptation for links with high bandwidths and/or delays.

In [24], Kevin et al. also compared RED with drop tail using the testbed priorly described. They simulated work loads at 90%, 98% and 110% together with queue sizes of 120 and 190 for drop tail and 120 and 480 for RED with varying parameters for w_q, max_p, *min_thresh* and *max_thresh*. Their results show that using RED leads to a higher number of successful responses in every simulation, regardless of the time frame for which successful responses were counted or the queue sizes.

Another option of RED that should be mentioned is RED's *gentle mode*. Normally, as illustrated by figure 3.4, if the average queue length exceeds *max_thresh*, every packet is dropped ($p = 1$). In *gentle mode*, RED does not immediately jump to $p = 1$ but rather increases p slowly up to 1 as the average queue length approaches $2 * max\_thresh$. While this does not increase the maximum achievable performance of RED, it increases its performance on average since it makes RED more robust against suboptimal parameter settings [20].

Using RED is generally suggested, even with suboptimal parameters. Since it is partly based on random decisions, it eliminates the traffic synchronisation phase effects mentioned in subsection 3.4.1. RED keeps the average queue length short, which results in short queuing delays, but can still cope with short bursts of traffic without packet loss. Moreover as a side-effect, it provides some degree of quality of service, since the probability of dropping a packet from a sender is roughly proportional to its sending rate. Therefore, RED penalises senders with high rates.

Chapter 4

Evaluation

The effect of using the PEP proposed in this thesis (cf. chapter 3 for the theoretical architecture and chapter 5 for the implementation in ns-2) has been thoroughly examined. This chapter will discuss and illustrate the results of the measurements that have been taken.

4.1 Methodology and Simulation Setup

To measure the effect of using the PEP and compare the performance between enabling and disabling it under a variety of network conditions, 450 simulations were done in ns-2 with the sample TCL script provided on CD (see section 5.2.1 for a description of the parameters and figure 3.1 for an illustration of the topology). The script creates a dumbbell topology, with a specifiable number of nodes that act as TCP senders. They are connected to the split start node via Ethernet links with a delay of 5 ms and a specifiable bandwidth (100 Mbit/s unless specified otherwise). The split start node is connected to the split end node via a link with a bandwidth of 100 Mbit/s and a fixed delay of 250 ms, representing a link over a geostationary satellite [1]. The split end node is connected via Ethernet links to a number of nodes acting as TCP receivers, again with the bandwidth specified and a fixed delay of 5 ms. The number of TCP senders and TCP receivers is equal, as are the characteristics of their links to the split start and split end nodes respectively. The queues for all links except for the satellite link are

Table 4.1: Parameters for RED when the (recommended) automatic configuration is used, for a link with a bandwidth of 100 Mbit/s and a delay of 250 ms.

Name	Value
min_thresh	62.5
max_thresh	187.5
max_p	0.1
q_w	0.000005

DropTail queues with a maximum length set to 100 packets, which equals to 1.5 times the bandwidth-delay product for 100 Mbit/s and 5 ms. As explained in section 3.1, the error rate on all links is 0 by definition and the only cause for packet loss is packet dropping in queues.

The queue size used by the split start node is given for each experiment. If the PEP is disabled, this size is simply the queue size for the split link. If the PEP is enabled, the queue size stated in the discussion and figure descriptions represents the size of the local queue used by the PEP on the split start node — in this case, the size for the split link queue is set to 500. The only additionally configurable DropTail parameter, head dropping, is disabled. For RED queuing, the standard parameter suggestions of Sally Floyd as stated in [20, 19] were used, with the gentle mode enabled. These standard settings cause the ns-2 implementation of RED to automatically configure its parameters, the values of which are listed in table 4.1.

The simulations run with the PEP disabled use end-to-end SACK TCP flows. With the PEP enabled, the end senders and receivers still use SACK TCP, but the TCP connection between the split nodes is either using SACK or CUBIC, as stated.

The applications used for producing traffic by the end hosts are FTP connections with an unlimited sending rate (i.e. the sending rate is only limited by TCP, not the application), and simulated web traffic by use of a standard Pareto

distribution. Every sending node has exactly one application installed. Therefore every application transmits its data over a separate link and queue, which minimises any undesired delay and phase effects and drops on the links that connect the end nodes with the split nodes, since it might affect simulation results. The receiver window is set very high as not to be a limiting factor. The payload size for all packets is 1000 bytes, which — together with an additional 20 bytes for the TCP header and 20 bytes for the IP header — results in a total packet size of 1040 bytes. This leads to a bandwidth-delay product on the satellite link of 3250 packets.

4.2 General

The PEP system presented in this thesis has been evaluated with both TCP SACK and CUBIC as protocols used between the split nodes. However, as illustrated by figure 4.1, TCP SACK exhibits poor behaviour for long fat pipes such as satellite connections. The huge delay coupled with the relatively large bandwidth leads to a slow convergence rate of TCP SACK, which is also the main reason to investigate the usage of a PEP. In the example illustrated by figure 4.1, the first back-off of TCP happens at approximately 5.500 seconds. It takes over 2.500 seconds (approximately 42 minutes) for TCP's AIMD mechanism to reach full link utilisation again. Therefore, although all measurements have also been done with TCP SACK as the split link protocol, the results of these simulations are neglected from the evaluation discussion since it does not provide any performance improvements. Furthermore the first few seconds of the evaluation graphs are omitted from the discussion since they represent the initial start-up phase of the system, not the steady state behaviour that is of more interest (see section 5.3 for a more detailed explanation).

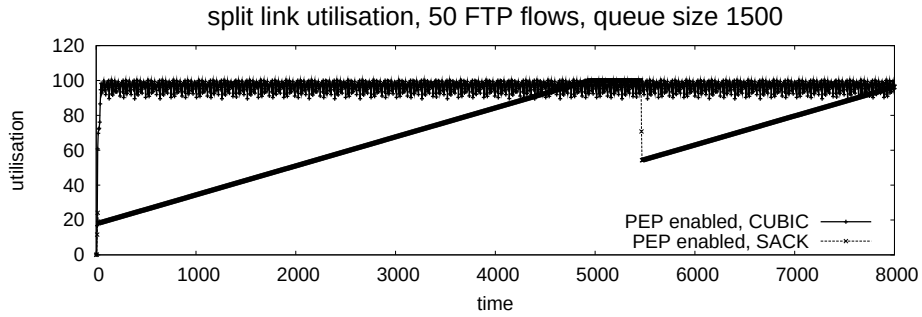

Figure 4.1: A comparison of the link utilisation of the PEP system with TCP SACK and CUBIC as protocol in use. The figure shows TCP SACKs poor behaviour upon packet loss over links with high bandwidth-delay products.

4.3 FTP

The measurements with simulated FTP traffic show a performance increase regarding both end-to-end throughput and link utilisation with decreasing queue sizes. As an example, an illustration and discussion of a scenario involving 50 FTP flows is given. Later on, this scenario will be compared with others where a varying number of flows was used. For relatively large queue sizes, the queue is able to mitigate the effect of multiple TCP flows fighting over the available bandwidth as expected. Since the *max_thresh* limit for RED is 187.5 packets (see table 4.1), the limit for $p = 1$ (i.e. dropping every packet) is 375 packets in gentle mode. Hence, on average, the queue can hold up to 2625 additional packets during traffic peaks.

However when the overall queue size is decreased, this ability is reduced, which is illustrated by figure 4.2. When the PEP is not in use, as the queue size decreases, the burstiness of the TCP flows leads to multiple flows backing off either simultaneously or within a small interval (the former should only happen by accident since traffic synchronisation effects should be eliminated by RED). Therefore, in the following few seconds, the link is underutilised (represented

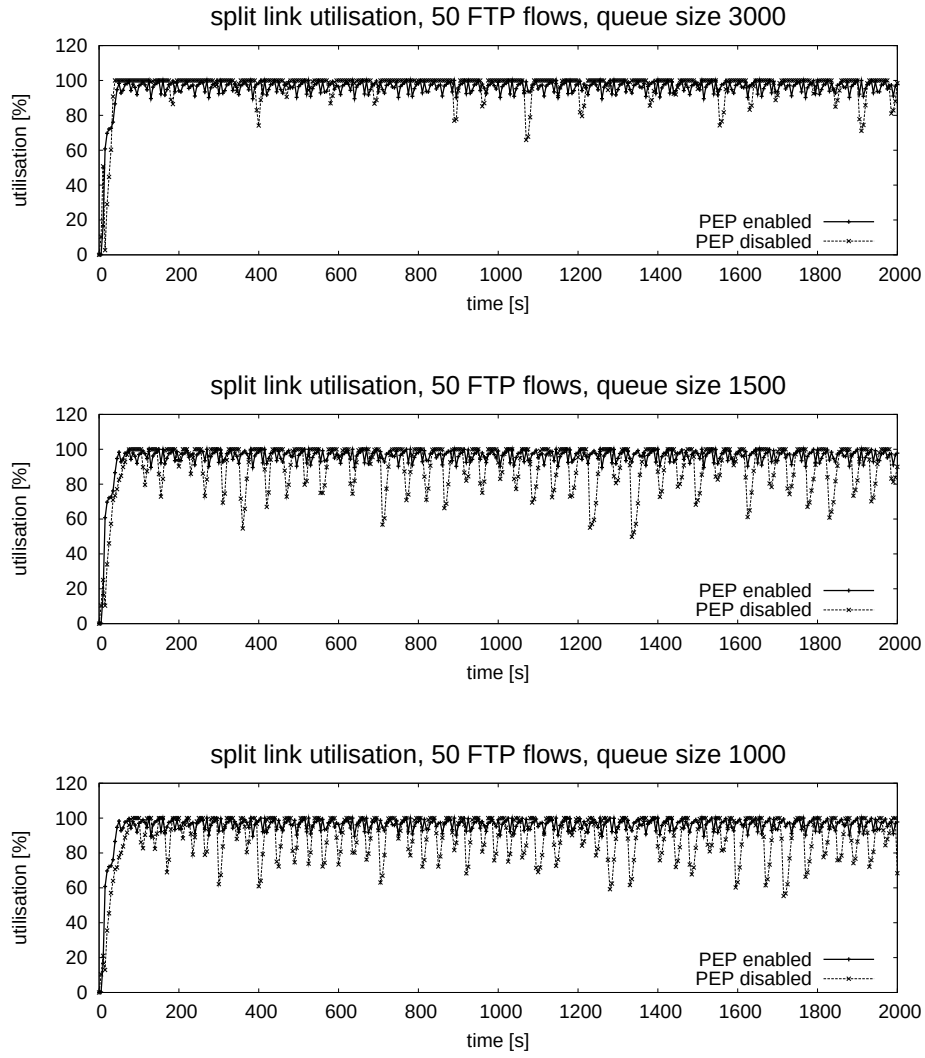

Figure 4.2: Split link utilisation for RED queueing with queue sizes 1000, 1500 and 3000 with FTP traffic using 50 SACK flows with the PEP disabled, and the PEP enabled with CUBIC as the transport protocol. The figures show an increasing performance improvement, when the PEP is used, for decreasing queue sizes.

by the valleys in the figure). However this is not the case for the scenario with the PEP in use, since there is only a single TCP flow that can utilise all the available bandwidth. Hence, for smaller queue sizes, the network can greatly benefit from using the PEP. The average split link utilisation of the PEP system is 94.2 Mbit/s, almost independent on the size of the local queue. The scenarios without the PEP in use show an average split link utilisation of approximately 95 Mbit/s, 88.9 Mbit/s and 87.9 Mbit/s for queue sizes 3000, 1500 and 1000 respectively.

As the utilisation shows, it should be mentioned that due to CUBIC's bandwidth probing behaviour, the average and the maximum bandwidth for limited time frames can be smaller compared to that of SACK TCP flows, as visible in figure 4.2 for a queue size of 3000. Since enabling the PEP means a link queue size of 500, CUBIC's bandwidth probing behaviour cannot be smoothened as much as would be possible with larger queue sizes. Therefore, for very large queue sizes, separate SACK flows can still outperform the PEP system in some cases, which is also visible in the throughput figure 4.3. This figure shows the average throughput received at the end nodes with both the PEP system enabled and disabled. First of all, the graph illustrating the throughput with a queue size of 3000 shows that the bandwidth of CUBIC periodically drops down, which is a result of CUBIC's previously discussed bandwidth probing function. As a result, the average end-to-end throughput is only 1.8 Mbit/s when using the PEP, compared to 1.92 Mbit/s without the PEP.

However also the system without the PEP in use shows some drops when using a queue size of 3000 (e.g. at approximately 1075 seconds). This effect grows stronger for smaller queue sizes, as illustrated by the second graph, which represents the same simulation for a queue size of 600. Here, the PEP system outperforms the non-PEP system, since the PEP system remains at its 1.8 Mbit/s whereas the average throughput of the non-PEP system drops to 1.68 Mbit/s. Figure 4.3 also shows how the performance of single flows is affected, as illustrated by the third graph. With the PEP disabled, flows try to gain bandwidth until packet loss occurs — after which they fall back. Therefore, there are intervals

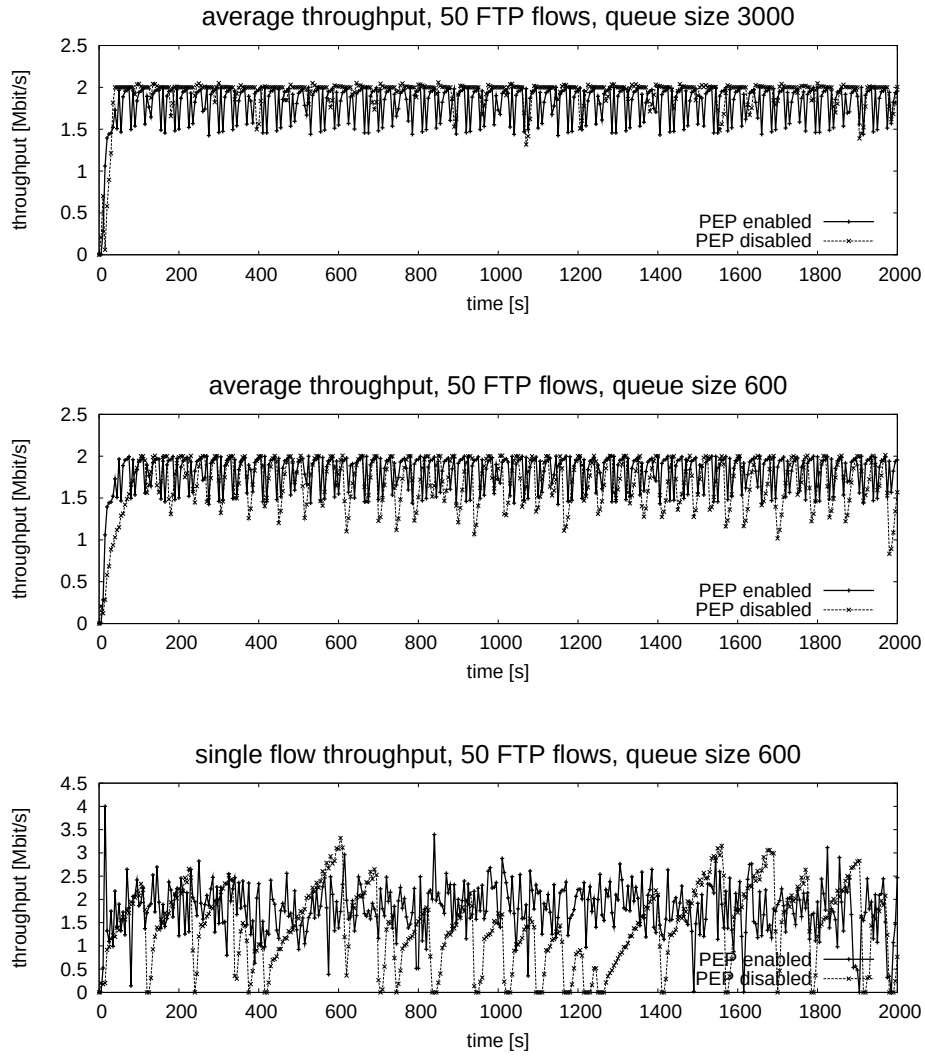

Figure 4.3: Average throughput and throughput for a single randomly selected flow for RED queueing with queue sizes 3000 and 600 with 50 FTP traffic flows. The figures show that the average throughput of the scenario without PEP is higher for a queue size of 3000. The third illustration shows the throughput of a single flow (no. 23) for both variants, with and without PEP in use.

where the throughput bandwidth borders at 0.0 Mbit/s, leading to a practically non-existent minimum bandwidth. The scenario with the PEP in use however shows that the flow only occasionally falls below 1 Mbit/s, but stays averagely at 2 Mbit/s (which is exactly a fiftieth part of 100 Mbit/s, the total bandwidth of the split link). Therefore the PEP system with a local RED queue in use seems to distribute the bandwidth more evenly between the flows compared to using 50 flows and without the PEP in use.

Changing the local queue of the PEP system to a DropTail queue has no effect regarding the link utilisation, since the 50 flows always provide enough data to saturate the link whether active queue management is used or not (i.e. the queue is never empty). Therefore the illustration of the link utilisation for RED as a local queue also applies for DropTail. The throughput as seen by the end receivers however differs, hence the illustration is given in figure 4.4. The average throughput shows almost no difference compared to the throughput when using RED, however this is also expected since the queueing mechanism — over long intervals — only differs in which packets are accepted at the split start node, not their general number (which is rather limited by the available bandwidth on the split link and the PEP's flow control, not the queueing mechanism). However the queueing mechanisms differ in the distribution of the bandwidth, as indicated by the third graph of figure 4.4. Like the measurements with RED, the bandwidth is — although with random spikes — overall more controlled compared to the scenario without the PEP in use, however one can see higher peaks of up to 5 Mbit/s. Therefore we can deduce that RED introduces a more evenly distributed random fashion than DropTail, which results in a smoother end-to-end throughput. In addition, RED provides more fairness between competing flows than DropTail. Using the latter can lead to phase effects and even starvation of single flows. Since RED's dropping decision is partly random-based, it introduces a random component into the network that effectively eliminates phasing effects. This leads to a higher level of fairness.

The performance gain is also increasing with increasing flow numbers, as indicated by figure 4.5. The measurements were taken in a scenario with 500

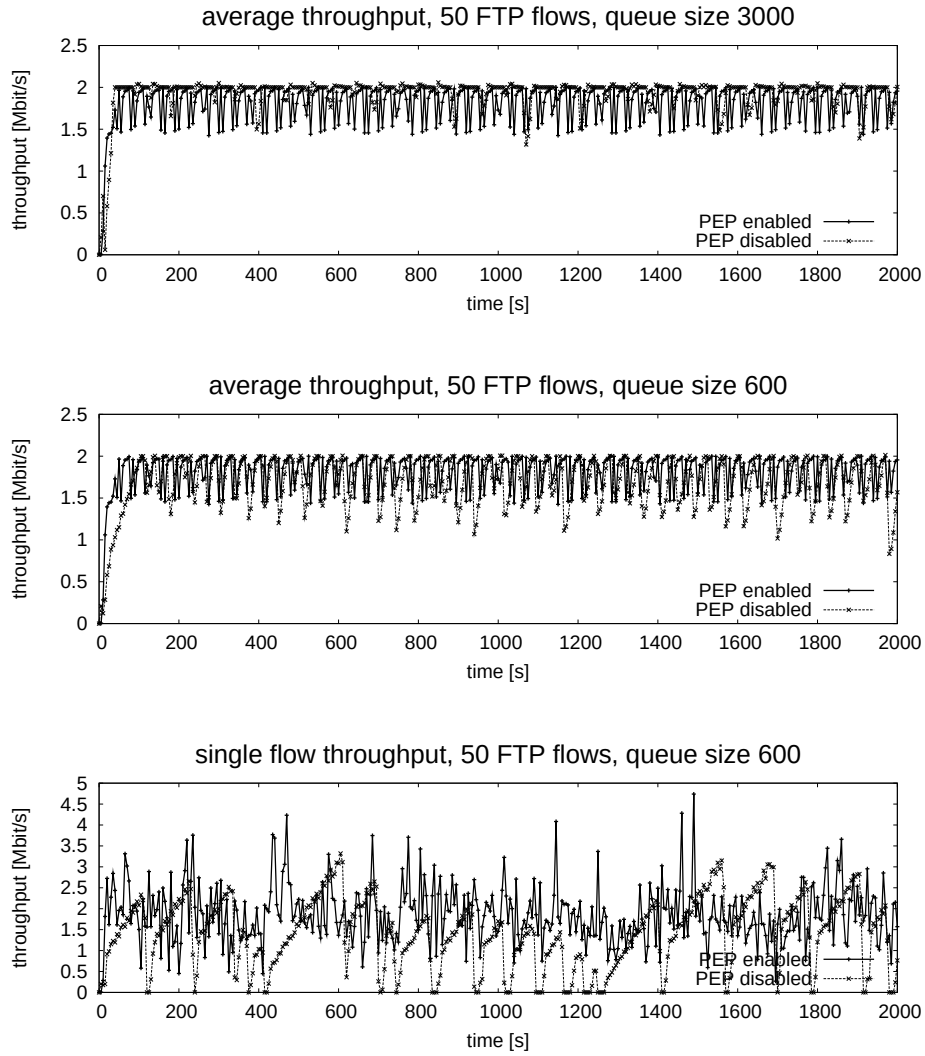

Figure 4.4: Average throughput and throughput for a single randomly selected flow for DropTail queueing with queue sizes 3000 and 600 with 50 flows transporting FTP traffic. The figures show that the average throughput of the scenario without PEP is higher for a queue size of 3000. The third illustration shows the throughput of a single flow (no. 23) for both variants, with and without PEP in use.

FTP traffic flows. Even for standard queue sizes like 3000 (approximately the bandwidth-delay product of the split link), the PEP system is able to provide a minimum utilisation of roughly 90 Mbit/s, whereas the PEP-disabled system shows underutilisation peaks down to 70 Mbit/s. Furthermore decreasing the queue size has no visible effect on the performance of the CUBIC flow of the PEP system (as expected, since the queue on the split link is fixed to 500 packets), it provides an average utilisation of about 94.1 Mbit/s. The simulation without the PEP in use however already has slightly lower average utilisation, approximately 93.3 Mbit/s, for a queue size of 3000. Decreasing the queue size to 1500 and 1000 results in an average utilisation of 85.1 Mbit/s and 81.5 Mbit/s respectively.

For smaller flow numbers, 5 in the example illustrated in figure 4.6, the queue size-dependent effect of the performance enhancement is even stronger visible. For a queue size of 3000 packets, the PEP system cannot outperform the standard scenario with 5 competing flows, since the DropTail queue used for the PEP-disabled scenario is able to buffer enough packets to mitigate the effect of TCP's fall back upon congestion. However this effect is increased greatly upon decreasing the queue size, as illustrated by the graphs representing simulations with a queue size of 1500 and 1000. As the queue size decreases, the interval at which the utilisation stays at its theoretical maximum shrinks and the performance drops become more frequent. This leads to an average utilisation of 87.8 Mbit/s and 78.4 Mbit/s for queue sizes of 1500 and 1000 respectively. For a queue with a limit of 3000 packets, the PEP-disabled system reaches 96.2 Mbit/s whereas the scenario with the PEP in use averages at 94.2 Mbit/s.

Further measurements regarding the behaviour of the end senders have been done, the results of which are illustrated in figure 4.7. It shows the average congestion window of 50 FTP senders over the course of the simulation. The first and obvious observation is that — without a PEP system — the congestion window is directly proportional to the queue sizes and bandwidth-delay products on the path and therefore shrinks when the queue size for the bottleneck link is decreased. Furthermore the split connection approach is directly visible, since

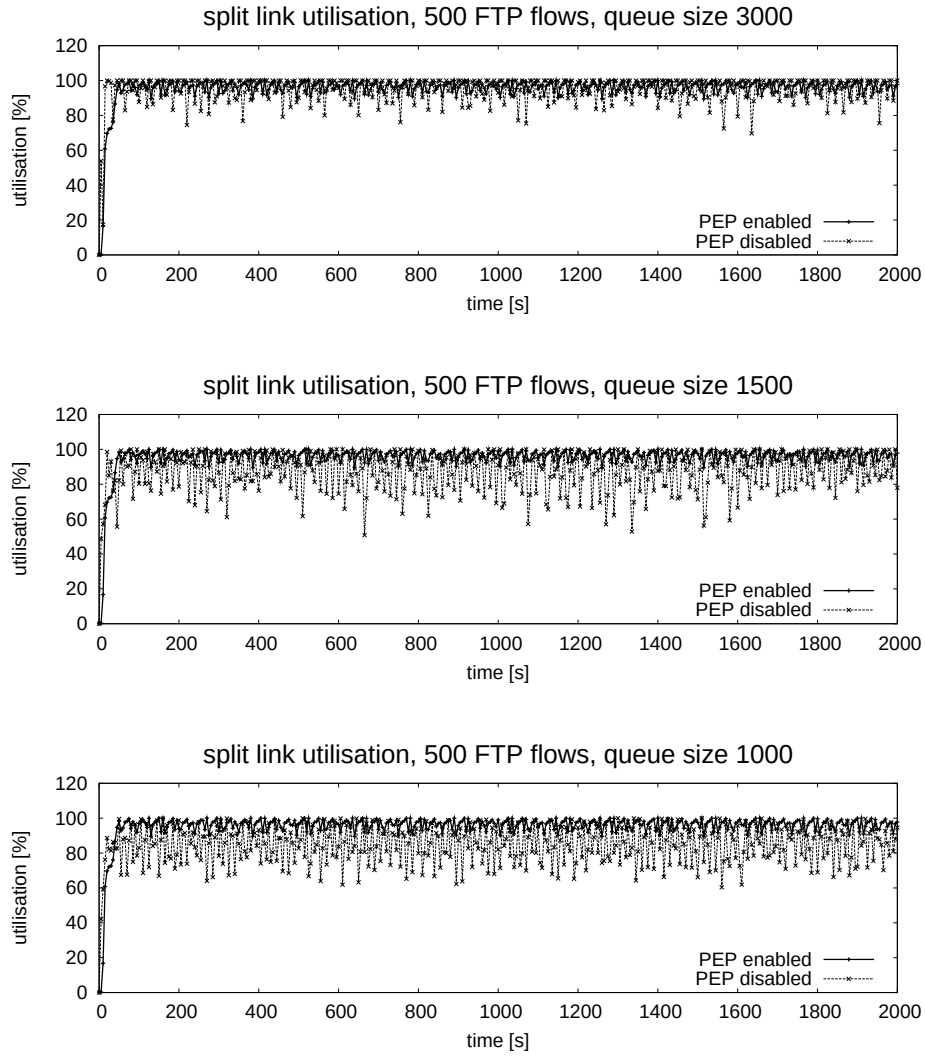

Figure 4.5: Split link utilisation for RED queueing with queue sizes 1000, 1500 and 3000 with 500 FTP traffic flows. The graphs show an increasing performance gain by use of the PEP system with decreasing queue sizes.

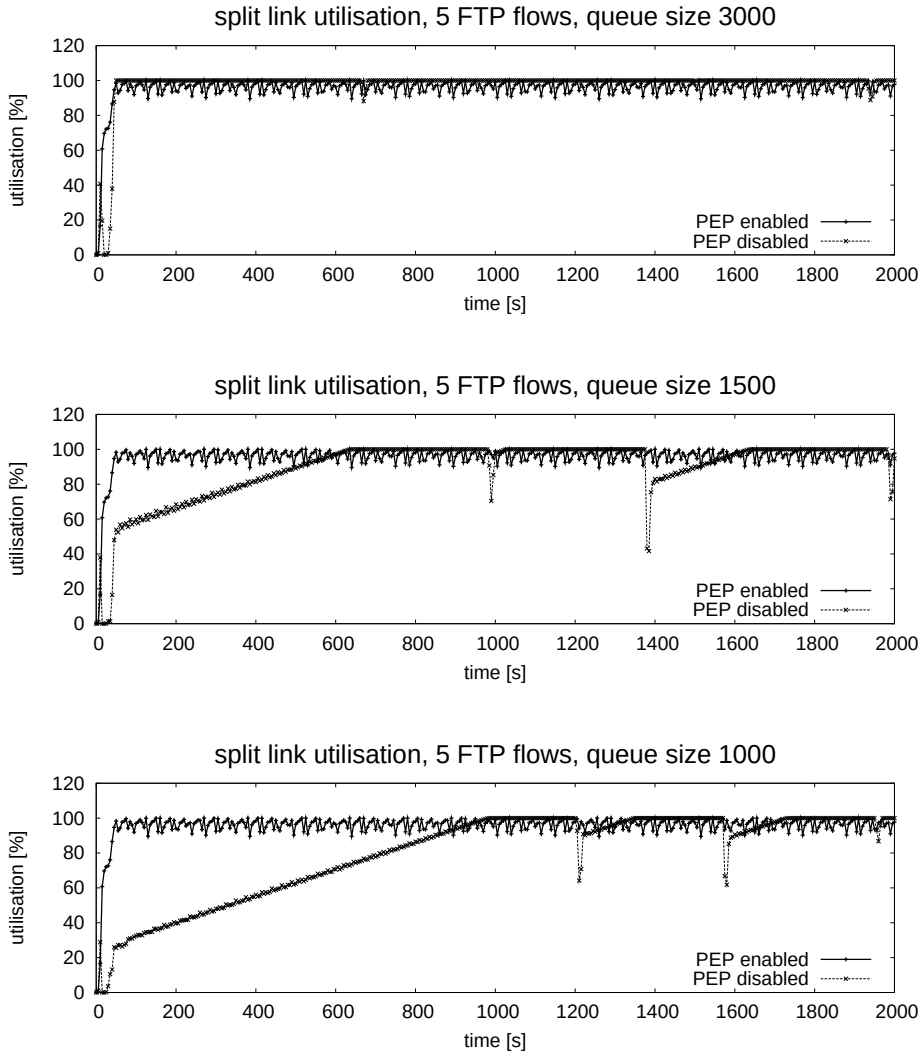

Figure 4.6: Split link utilisation for RED queueing with queue sizes 1000, 1500 and 3000 with 5 FTP traffic flows. The graph with queue size 3000 shows that the PEP system cannot always outperform standard scenarios with competing flows. However it shows performance gains for smaller queue sizes. Furthermore the scenario without the PEP in use shows performance degradation when reducing the queue size, since the DropTail queue in use leads to multiple flows simultaneously backing off.

the average congestion window of the senders is significantly smaller when the PEP is in place (approximately 30% for DropTail and 3% for RED compared to the cwnd of unsplit flows). This is an indication of the smaller control loop and round trip time of TCP, since its delay and number of hops have decreased (one of the main reasons for deploying a split connection PEP). In addition, the differences between the DropTail and RED mechanism are clearly visible. The large DropTail queue allows for comparatively high congestion windows since the queue can be at any fill level for arbitrary time intervals. For this reason, the congestion window is directly proportional to the size of the local queue used on the split start node. However this is not the case for RED, since its primary goal is to mitigate bursty traffic but at the same time keep an overall small average queue to minimise queueing delay. As previously explained, this is the reason why RED's *max_thresh* value is set to 187.5 packets by its automatic configuration (see table 4.1), whether the queue size is 600 or 3000. This leads to a much better responsiveness of TCP, which is the reason for a much higher utilisation and throughput of the PEP system for small flow numbers, as illustrated by figure 4.6.

General results of the measurements are illustrated and compared to each other in figure 4.8. All scenarios for these results were run with a simulation time of 2000 seconds. The graphs shows the split link utilisation in Mbit/s for the queue sizes 3000, 1500, 1000, 600 and 300 for increasing numbers of parallel TCP flows (1, 2, 3, 4, 5, 10, 25, 50, 100, 250 and 500). This allows for a comparison of TCP's behaviour and resource utilisation, while numerous effects can be observed. First it shows — as previously mentioned — that there are cases in which the PEP system does not provide any performance gains compared to separate, parallel TCP flows that are competing for bandwidth. The graph for the queue size 3000 illustrates this observation. The reason for this effect is the queue's ability to smoothen TCP's fighting effect (i.e. multiple flows backing off and leaving a link underutilised). This is also the cause for the different results obtained for smaller queue sizes. A queue size of 1500 is already enough to aggravate this effect, leading to poor throughput compared to the one obtained by the PEP's single CUBIC flow, depending on the number of flows. A further observation

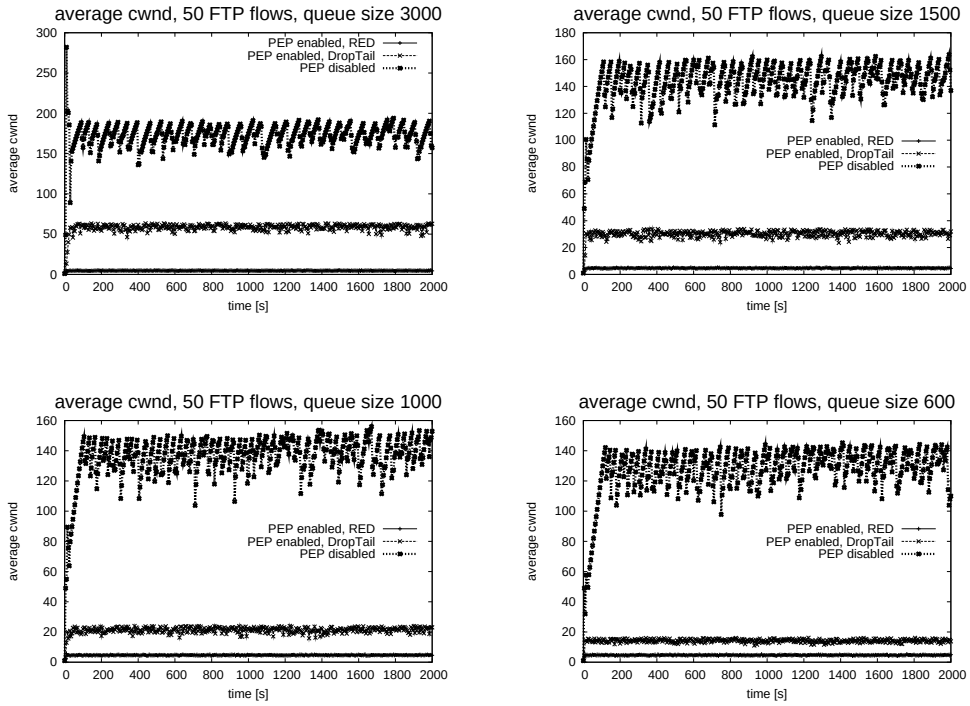

Figure 4.7: Average congestion window size of the end senders for queue sizes 1000, 1500 and 3000 with 50 FTP traffic flows. The graphs show the large congestion window for scenarios without a split connection mechanism, which shrinks for decreasing queue sizes. The windows when the PEP system is in use indicate that the average length of the RED queue is independent of the total queue size.

that can be made is that the peak performance of single TCP SACK flows in such topologies (i.e. links with high bandwidths and high delays) can be achieved by using 10 - 25 flows. Decreasing or increasing the number beyond this range seems to lead to less overall throughput.

A further interesting effect is visible in the simplest comparison, namely a scenario including the use of the PEP, and a scenario with a single unmodified TCP SACK flow. Although the PEP was originally considered to mitigate the fighting effect of multiple TCP flows, its comparison to standard TCP reveals the ineffectiveness of TCP SACK over long fat pipes. Therefore, one can gain performance benefits by using the PEP even for a single flow. The reason for this lies in the very design of CUBIC's congestion avoidance mechanism, as described in section 3.3.2. However, as illustrated by figure 4.8, it still depends on the queue size whether any performance gain is to be expected at all. A queue size of 3000 for the satellite connection seems to be enough to mitigate this effect and delay packet loss due to congestion beyond the 2000 second marker. However despite the fact that TCP SACK will eventually show such performance drops, it is still questionable whether they will have a significant impact on the overall performance. Together with CUBIC's bandwidth probing behaviour, which can keep it from achieving the absolute theoretical maximum of the bandwidth that is available over longer intervals, it is possible that a split connection PEP system using CUBIC will not show enough performance improvements regarding throughput in such scenarios to justify its deployment.

Moreover the graphs of figure 4.8 show that a local queue size of 300 seems to be reasonable enough as increasing the queue size further does not have any effect on the performance of the PEP — most probably due to the reason that enough flows supply the PEP with enough data to be forwarded, therefore the queue is never fully emptied, even if limited to a comparatively small size of 300. Hence the effect of various queue sizes (300 and above) for the local queue used by the split start node of the performance enhancing proxy is minimal.

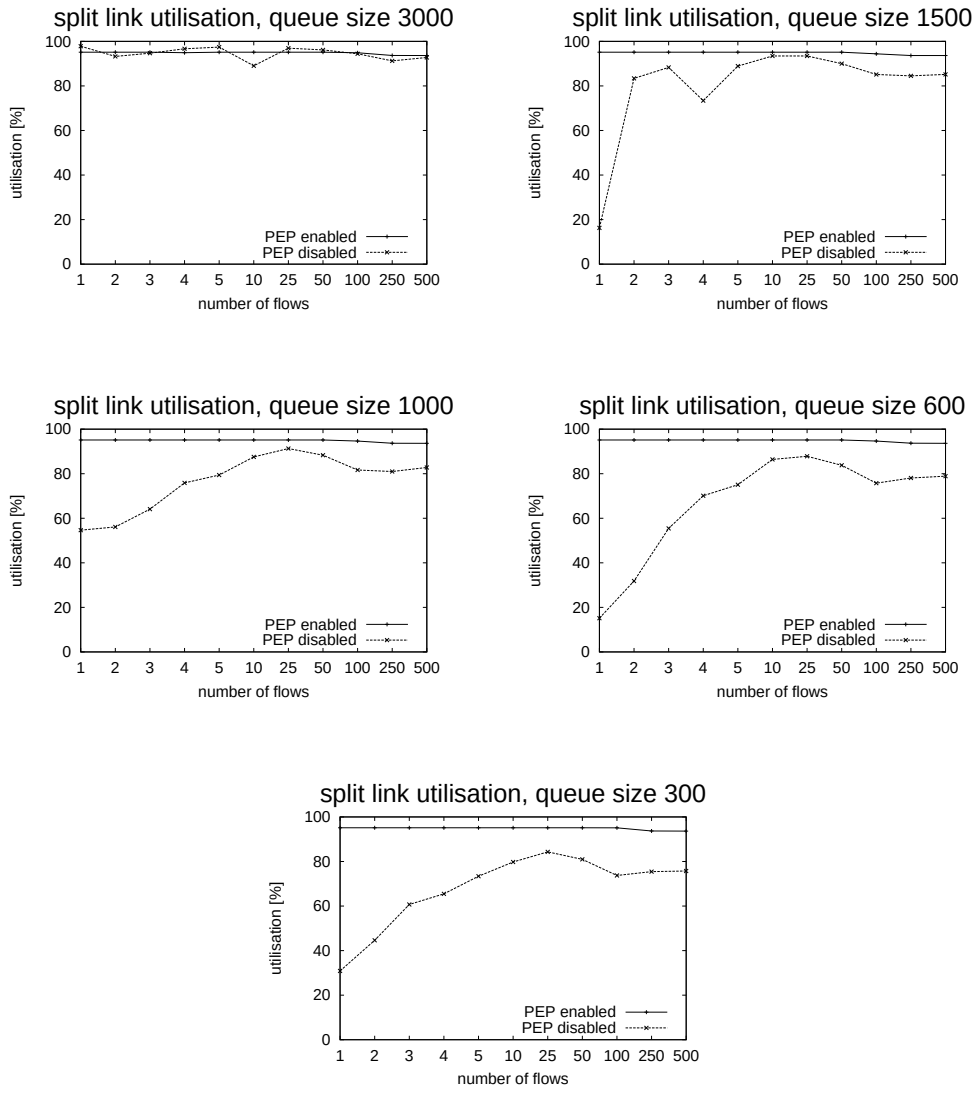

Figure 4.8: Overall FTP traffic measurement results of using the PEP system compared to unmodified parallel TCP SACK flows for different flow numbers and queue sizes. The results show performance gains for decreasing queue sizes, as well as a general performance deficit for 1–10 flows and 50–500.

Table 4.2: Default Pareto parameters in ns-2, which were used for the simulations with Pareto traffic.

Name	Value
burst_time_	500 ms
idle_time_	500 ms
rate_	64 Kb
packetSize_	210
shape_	1.5

4.4 Pareto

Simulations have also been done to investigate the effect of using the PEP for web traffic. For this, a standard Pareto distribution has been used as a traffic generator in ns-2, with 500, 750, 1000 and 1500 as numbers of users. Table 4.2 lists the parameters that were used for the Pareto distribution (which are also the default values for Pareto in ns-2) Due to the nature of Pareto web traffic, 500 flows do not fully saturate the link but rather produce traffic of approximately 80 Mbit/s, as can be seen in figure 4.12 — to be able to fully saturate a link with a bandwidth of 100 Mbit/s, one needs approximately 650 such flow. Therefore these simulations represent approximately 80%, 115%, 153% and 230% workload regarding the bottleneck's throughput.

Figure 4.9 shows the bursty nature of web traffic. Even scenarios with the largest queue size, 3000 packets, result in drops of the utilisation on the split link since the queue is unable to mitigate the irregularities. Still, due to CUBIC's bandwidth probing behaviour, the PEP system is unable to achieve better performance than the separate parallel TCP flows, which show an average utilisation of 95.3 Mbit/s compared to the PEP's 93.8 Mbit/s. However — as we have already seen for FTP traffic — the PEP can easily outperform concurrent flows for smaller queue sizes, since the CUBIC flow of the PEP manages to stay at 95.4 Mbit/s for queue sizes of 1500 and 1000, whereas the unmodified TCP flows drop to 89.9 Mbit/s and 87.2 Mbit/s respectively. Still, it can be observed

that the PEP — using a local RED queue for this experiment — seems unable to achieve its nominal performance for the largest queue size of 3000. This effect is further illustrated and discussed together with figure 4.12, when the overall Pareto results are presented.

The effect of using the PEP with different queueing mechanisms is illustrated in figure 4.10. It shows the throughput of a single flow without the PEP in use, and with the PEP in use together with RED and DropTail as local queueing mechanisms respectively. First of all, the great number of TCP flows without a performance enhancing proxy seems to lead to intervals of starvation for single flows (e.g. between 460 and 570 seconds for the selected flow). However it should be mentioned that smaller intervals which lack any throughput can also be caused by the nature of Pareto's traffic and simply be a result of no browsing activity by the user. This effect is still present when using the PEP, but greatly mitigated. Furthermore one can observe that using RED as a local queue for the PEP seems to result in higher peaks regarding throughput, although shorter in time. On the other hand, DropTail queueing — for which the randomness of packet dropping is a result of the sending rate of the TCP flows themselves — leads to a slightly more consistent but smaller throughput. However it should be noted that the average throughput of this flow is 103 Kbit/s without the PEP in use, whereas enabling the PEP results in an average throughput of 105 Kbit/s for DropTail and 152 Kbit/s for RED queuing. Since the average of the throughput of all flows shows little variance (119 Kbit/s without the PEP and 120 Kbit/s with the PEP for both queuing variants), the increase in performance of 50% for this single flow when using RED must be a random phenomenon (i.e. other flows will show a similar decrease in performance). This also leads to the conclusion that — although it usually punishes flows with more packets more gravely — RED provides no bandwidth guarantee for small, bursty traffic.

Figure 4.11 shows the effects of using RED and DropTail as local queueing mechanisms respectively by illustrating the cumulative throughput one flow achieves over time. The selected flow is the same that was used for figure 4.10. In the cumulative graph, the effect of DropTail, which allows a smaller but more

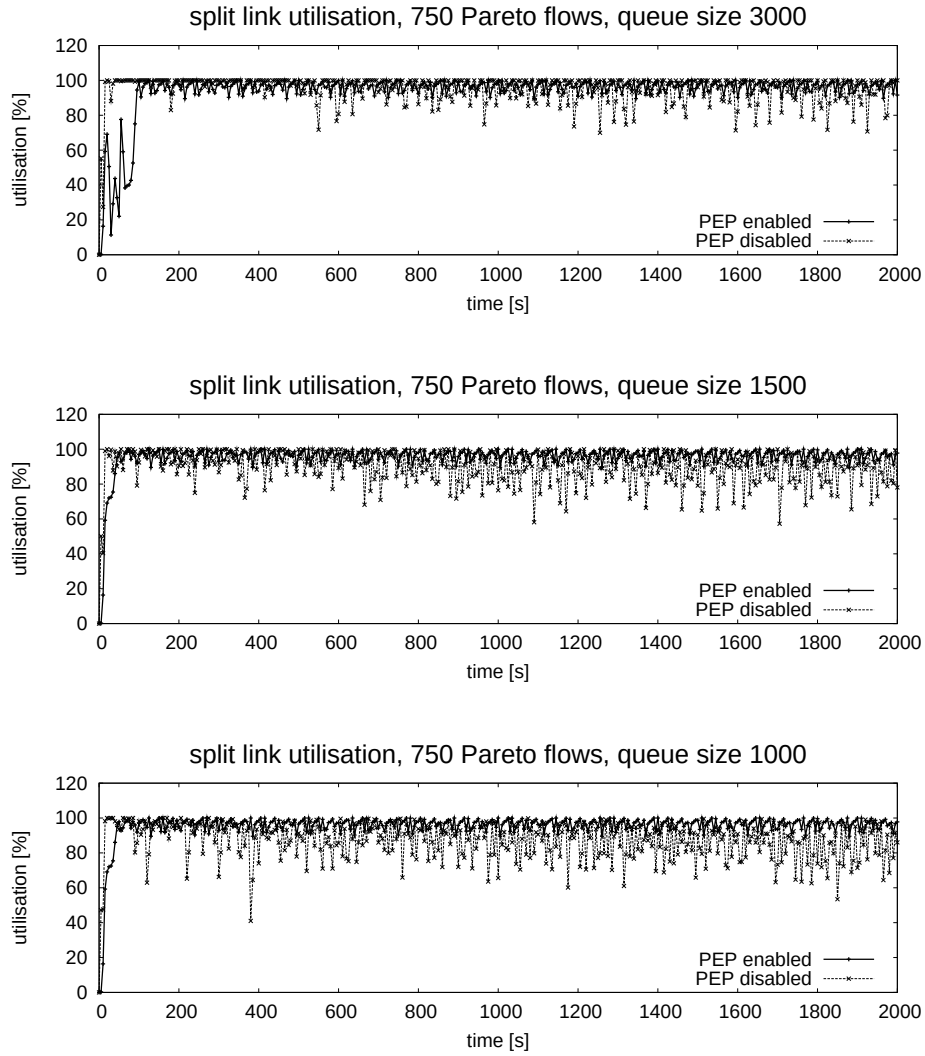

Figure 4.9: Split link utilisation for RED queueing with queue sizes 1000, 1500 and 3000 with 750 web traffic flows, represented by a Pareto distribution. The graphs show decreasing performance regarding utilisation for parallel TCP flows with decreasing queue sizes. However even a queue that can hold 3000 packets is unable to fully mitigate the bursty nature of web traffic.

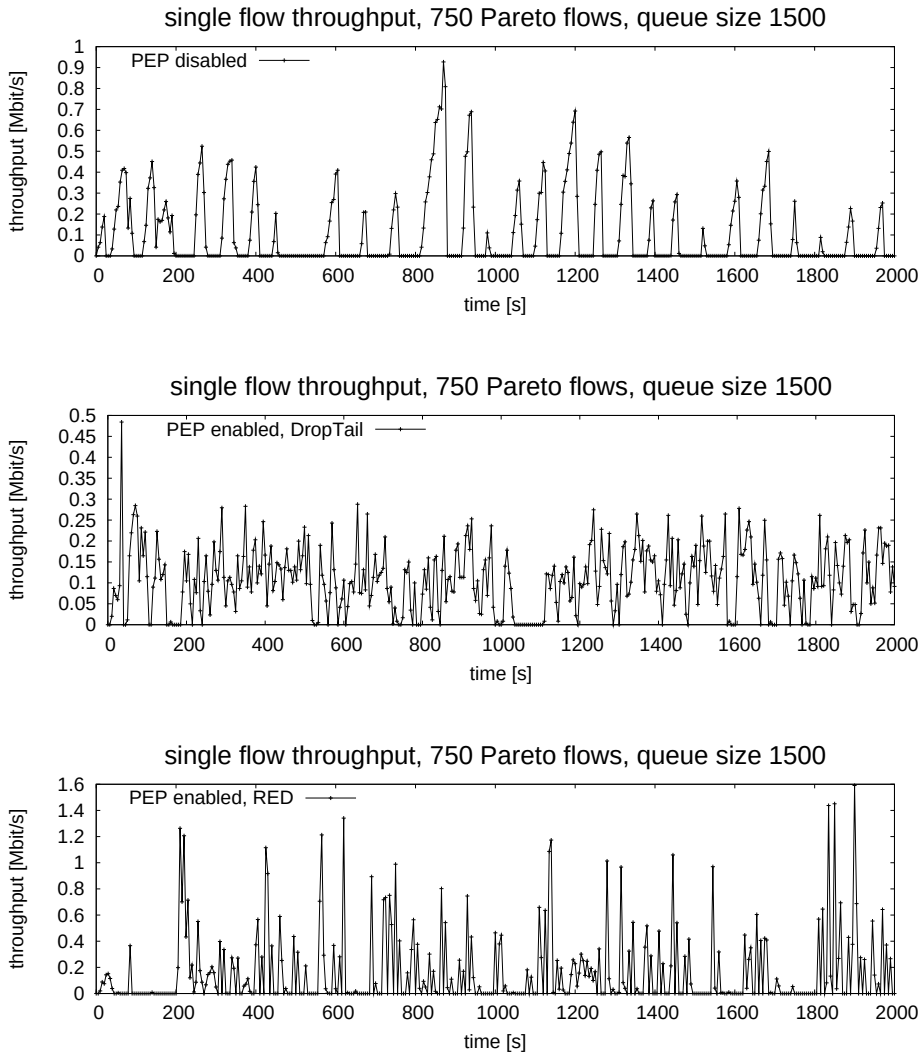

Figure 4.10: Throughput for a single randomly selected flow (no. 547) with the PEP disabled and enabled with both RED and DropTail as local queues for a queue size of 1500 and 750 web traffic flows. The graphs show the different effects of RED and DropTail. RED seems to lead to higher, shorter peaks in throughput for a single flow, compared to DropTail which results in single flows achieving a smaller throughput but over longer intervals.

steady throughput, is more visible since it leads to a small but steady increase in the amount of transferred data. In contrast, using RED leads to sporadic intervals in which more data is transferred, followed by intervals during which less data is transferred. This is also the case for the scenario without using the PEP. Comparing this simulation with reality leads to the conclusion that RED allows for short time frames in which more data (e.g. contents of a web page) can be transferred to the user that is browsing the web, possibly resulting in some delay after which large parts of the requested web page are received. DropTail on the other hand seems to provide longer time frames with less bandwidth, which might result in less delay but longer loading times once the transfer starts. Therefore RED might be preferred over DropTail for web traffic since the chances are higher that the transfer of a web page will be completed faster, once the transfer has started.

The general results of simulations done with Pareto web traffic, illustrated by figure 4.12, show that generally using RED has no advantage over using DropTail as a queueing mechanism for the local queue used by the split start node of the PEP. For the scenario with a queue size of 3000, DropTail even seems to outperform RED marginally. This might be caused by RED dropping packets earlier than DropTail (which drops only if the queue is full) - therefore packets of short and bursty traffic can have a bigger chance of getting forwarded with DropTail queues compared to RED (assuming the same queue size for both mechanisms). Therefore, the flows might be forced by RED to back off sooner. While it was expected for the PEP system with a local RED queue not to show significant performance gains for large queue sizes (similar to the results obtained with FTP traffic), the PEP system with RED actually shows worse link utilisation for a queue size of 3000, achieving between 93.8 and 91.0 Mbit/s whereas the PEP system using DropTail manages to stay at 95.4 Mbit/s. This phenomenon is less visible for a queue size of 1500 (for which the PEP-enabled system achieves better throughput results, between 95.4 and 94.4 Mbit/s) and completely disappears for queue sizes 1000 and below.

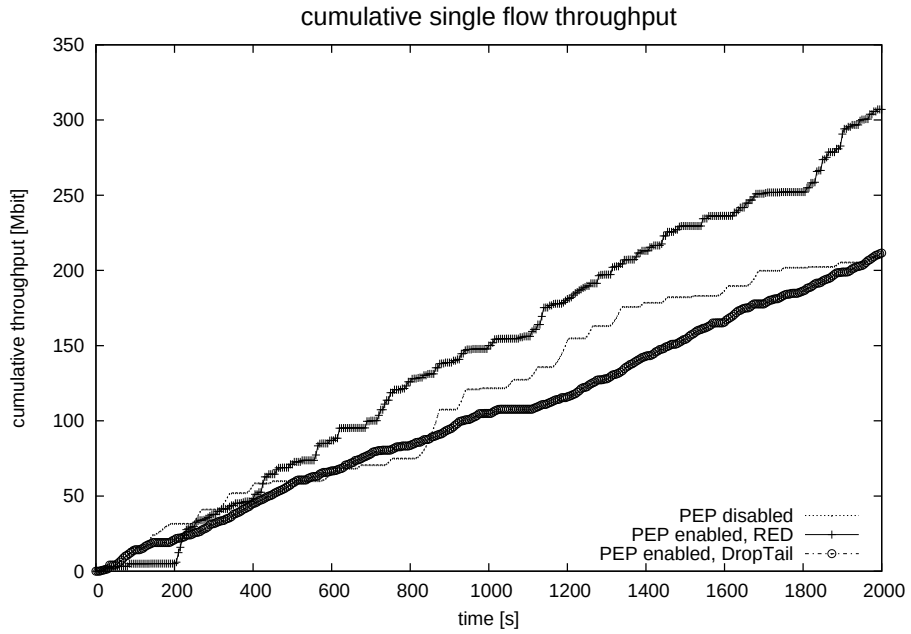

Figure 4.11: Cumulative throughput for a single randomly selected flow (no. 547) with the PEP disabled and enabled with both RED and DropTail as local queues for a queue size of 1500 and 750 web traffic flows. The graph shows that DropTail leads to a smoother increase in the amount of transferred data over longer periods of time compared to RED, which results in the flow transferring higher amounts of data but for shorter intervals.

Apart from this small observation, the overall results show a similar performance of the PEP system compared to FTP traffic. For 500 flows, which cannot saturate the link, using the PEP system shows no positive or negative effects. For higher flow numbers, both PEP systems show performance gains since they stay at 95.4 Mbit/s, independent of the flow number. Without the PEP, the average link utilisation drops to approximately 88.5 Mbit/s for a queue size of 1500, 86.0 Mbit/s for 1000, 82.0 Mbit/s for 600 and 80.0 Mbit/s for a queue size of 300.

The overall results of all the simulation scenarios show that using the PEP described in this thesis can lead to performance gains for small queue sizes. Since it uses CUBIC as the transport protocol for delivering the data, there is no need for large queue sizes in the range of the bandwidth-delay product or twice this value. The experiments presented in this chapter have been done with a link queue size of 500 for the PEP system. Furthermore the measurement results show that the PEP system achieves good performance even for small local queue sizes of 300 and 600. Therefore this performance enhancing proxy might be useful in situations where hardware costs, power requirements or other constraints lead to the usage of small buffers. As the illustration and discussion of the measurements show, using a split connection PEP in such cases can raise the performance up to 500%, depending significantly on the traffic and the number of flows that are used.

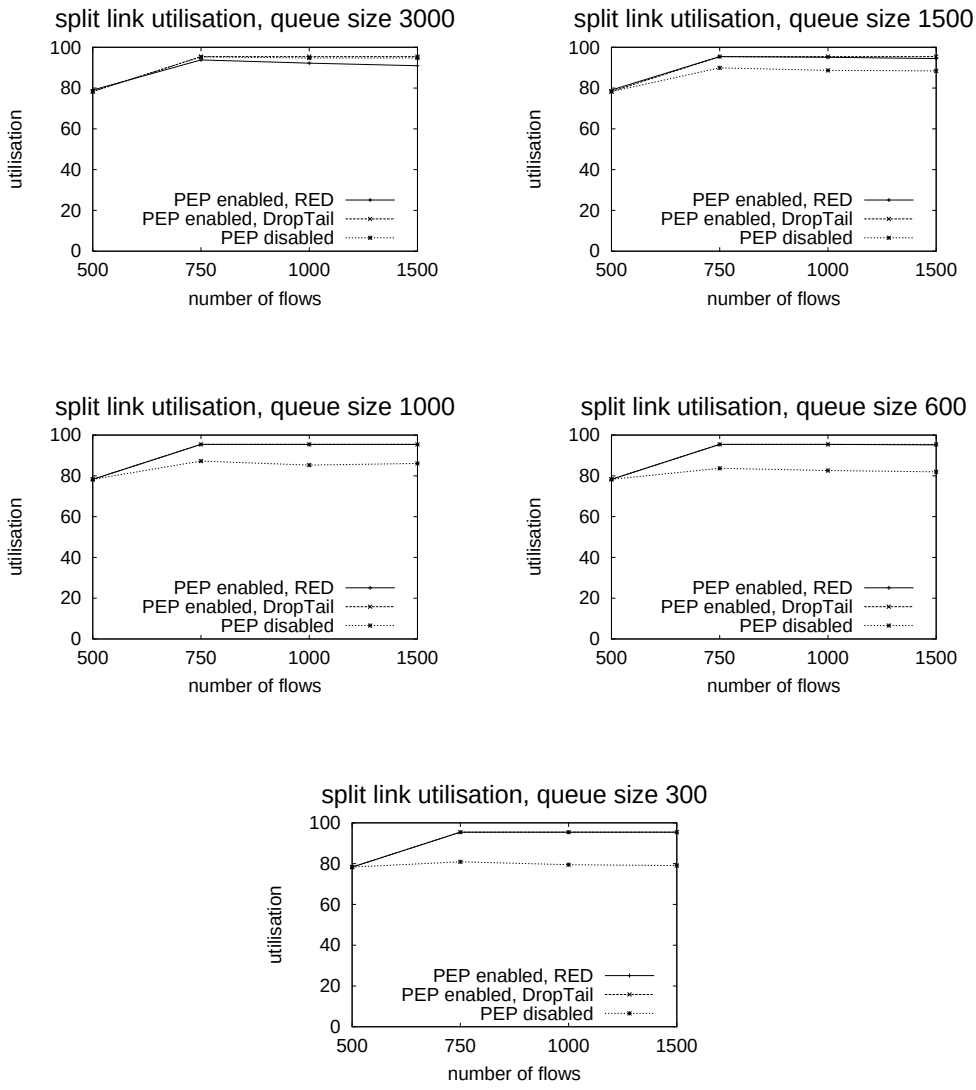

Figure 4.12: Overall Pareto web traffic measurement results of using the PEP system compared to unmodified parallel TCP SACK flows for different flow numbers and queue sizes. There is no performance gain for a queue size of 3000, however for decreasing queue sizes the PEP system shows higher link utilisation compared to the standard scenario with separate parallel TCP flows.

Chapter 5

Implementation

This chapter documents the ns-2 implementation details of the PEP. Section 5.1 presents the newly introduced C++ classes and discusses the modifications of pertaining class files of ns-2. Moreover section 5.2 will show the usage of the new agents in TCL and how they can be configured. Section 5.3 finally deals with the resource requirements of ns-2 and how they can be reduced.

5.1 Implementation in ns-2

Basically the PEP implementation consists of three parts: The sender-side modifications, the receiver-side modifications and the so-called *SplitManager*. The first two ensure the correct PEP processing of the TCP segments and regulate flow control. The SplitManager is used to configure the system and change various parameters that affect the performance and behaviour of the performance enhancing proxy. In addition, it holds references to objects and parameters commonly used by the sender and receiver agents. A basic illustration of the consisting parts of the PEP nodes is given in figure 5.1.

Since the sender- and receiver-side modifications are essentially changes to pertaining TCP agents of ns-2, one must choose between two possible implementations that are available for TCP. First there are the one-way TCP agents. Together with the Linux wrapper classes, they provide a variety of TCP variants spanning from the original TCP protocol to versions with newer congestion con-

trol mechanisms like compound TCP or CUBIC. However they do not provide means to carry real payload data, they are only able to transmit packets that hold a size-field in the packet's common header *hdr_cmn*, that can be used for resource-checking in queues accurate to a byte for example. But using such packets (and the agents generating them) to transport application data (e.g. HTTP requests and responses) is not possible. Furthermore the TCP flow is a one-way flow, i.e. the virtual data can only be transmitted from a specific source to a specific sink. In addition, one-way TCP agents normally do not support a specific connection establishment or tear-down, as TCP does in reality. Hence, TCP connections are simply available throughout the entire simulation as long as the agents are in use.

Second there are ns-2's *FullTCP* agents, which do exchange packets carrying real payload data. These agents can be used to establish two-way connections and transport application layer data between them in both directions. However they do not provide the newer congestion control mechanisms mentioned above, but only Tahoe, Reno, NewReno, and SACK. This renders them useless for the purpose of this thesis, since the main application for this PEP are long fat pipes, i.e. links with a high bandwidth-delay product. The TCP variants provided by *FullTCP* behave poorly in such environments, as discussed in chapter 3, which is the main reason why this performance enhancing proxy needs mechanisms like CUBIC. Furthermore *QuickStart*[1], which would be able to mitigate this disadvantage, is also not available for *FullTCP* in ns-2.

For these reasons — and since *FullTCP* is incompatible with ns-2's one-way TCP agents — the implementation modifications extend the one-way TCP agents rather than *FullTCP*, which — although no real data transfer is possible — allows measurements to show if there are benefits that can be gained from using such a performance enhancing proxy in topologies with long fat pipes.

The implementation of this PEP has been done and tested for ns-2.33 only. It is not guaranteed that the modifications will build or produce correct results with older or newer versions of ns-2.

[1] http://www.icir.org/floyd/quickstart.html

5.1.1 SplitManager

The *SplitManager* (TCL name: *Application/SplitManager*; C++ class files `ns-base-dir/tcp/split-manager.*`) is a control entity for the whole PEP system. Both nodes at the beginning and end of the split connection hold a reference to the same SplitManager instance, ensuring that only valid parameters are used. Since this PEP system is only simulated in ns-2 this is a valid solution. In reality however it might be necessary to use one control object for each node and to arrange for some communication between the two to coordinate their actions. For example the number of TCP end flows between the senders and the split start node as well as the receivers and the split end node should always be the same. The implementation uses a single SplitManager object for both the split start node and the split end node, therefore preventing any misbehaviour due to configuration errors.

Both split nodes hold references to the SplitManager object and use these to get current values for queue and buffer sizes for example. This allows future simulation scripts to change parameters like the queue length or the packet size on-the-fly during the simulation and observe the effects. Presently there are five parameters that can be configured via TCL:

number_of_connections_ This value represents both the number of flows as well as the number of senders and receivers respectively.

buffer_start_ The parameter buffer_start_ is used as a starting value for flow control. The split start node sets its buffer_count_ variable to this value during the initialisation phase, to allow some data to be received at the beginning.

header_size_ Normally the header size for TCP/IP simulations is 40 Bytes, 20 Bytes for the IP header and 20 Bytes for the TCP header. However to accommodate deviations, the header size of the packet can be changed with this parameter.

Table 5.1: Default values for the variables of SplitManager when they are not set through TCL (*number_of_connections_* must always be set).

TCL/C++ variable name	Default value
number_of_connections_	0
buffer_start_	200 * 1040 bytes
header_size_	40 bytes
packet_size_	1000 bytes
send_check_interval_	0.01 seconds

packet_size_ Also, the packet size is usually 1000 Bytes. To be able to simulate other packet sizes, this value can be modified.

send_check_interval_ This value denotes the time interval between checking whether there is data to be forwarded at the split start node (see section 5.1.2).

In addition to these five parameters, a SplitManager object also holds a reference to a queue that is used locally for buffering incoming packets at the split start node (see section 3.2 for an explanation of how the flow control works). It is defined as a *Queue* object. *Queue* is the superclass which all ns-2 C++ classes implementing queues are derived from. Basically it is composed of a PacketQueue (a simple FIFO queue that can hold packets), methods for enqueuing and dequeuing packets, and methods to keep statistics. Usually the *enque* method is then re-implemented in subclasses like DropTail to perform certain tasks (e.g. only insert a given packet into the underlying PacketQueue if the queue length is below a certain limit, drop the packet otherwise). By using a *Queue* object as the queue for SplitManager, we ensure compatibility with any queuing mechanism currently implemented in ns-2. This eliminates the need to re-implement queuing techniques which are both already present and well-tested in ns-2. One simply needs to specify the queue to be used in the TCL simulation script (see section 5.2). This queue is then used locally on the split start node, and packets are enqueued before they are received by the receiving TCP stack of the split start node. Only when flow control allows further packets to be received, these packets

are dequeued and handed to the *recvFinal()* method of the receiving TCP stack (see subsection 5.1.3).

Furthermore SplitManager also holds references to two *Table* objects (the classes for which are also defined in the C++ class files for SplitManager, `ns-base-dir/tcp/split-manager.*`). They are necessary to ensure that the data arriving at the split start and split end nodes is forwarded correctly. Every TCP connection in ns-2 needs a sending TCP stack and a receiving TCP stack (called a TCP sink). Therefore — if a node holds 10 incoming connections — there are 10 TCP sinks installed on that node. When a packet is received by this node, the receiving TCP stack simply hands the size of the payload data over to the application layer. There is no information about where the data came from. To be able to forward the data accordingly, SplitManager uses two tables *send_* and *recv_*, one for each split node.

Basically a *Table* is a simple FIFO list that can hold addresses (i.e. integer values). For every packet that arrives at the split start node, the source address is copied into the list by its receiving TCP sink. The payload data is then handed to the application layer, which again hands it to the sending TCP stack to forward it in form of a new packet to the split end node. When this new packet is ready for transport, the address previously inserted into the table is removed again and saved into a special field of the IP header (see subsection 5.1.4). Hence, the split end node can forward the data accordingly. If there were no such field, the split end node would be unable to determine the receiver for the payload. At the split end node, the field is read and inserted to another table. When the application layer gets the data, it removes the first entry from the table to determine which sending TCP stack needs to forward the data to the end receiver (since we have multiple TCP sender agents each holding a connection to a single node, we need to determine the respective agent, not the destination address).

Since the insertion and removal of addresses needs to be done every time a packet is received, *Table* has been implemented as a resizeable ring buffer array for performance reasons regarding code execution.

Table 5.2: Function call chain that illustrates packet processing when receiving a packet at the split start node.

Function name	Description
SplitTcpSinkAgent::recv()	This function is called whenever a packet is received. It calls the enque() function of the local queue.
Queue::enque()	Normal enque() function of *Queue*; enqueues a packet and exits.
SplitTcpAgent::checkSend()	Called whenever the timer fires (see subsection 5.1.2); if there are packets in the queue and the buffer value is high enough, a packet is dequeued and handed to SplitTcpSinkAgent::recvFinal().
SplitTcpSinkAgent::recvFinal()	Adds the source address of the packet to the *recv_* table by calling SplitManager::add(); calls DelAckSink::recv() afterwards, which in turn calls SplitTcpAgent::recvBytes() with a number of payload bytes that are delivered (*nbytes*, see figure 5.1).
SplitTcpAgent::recvBytes()	Forwards the payload to the split end node by calling *SplitTcpAgent*'s inherited sendmsg() function.
SplitTcpAgent::output()	Slightly modified version of TcpAgent::output(), removes previously added addresses from the *recv_* table and saves them in the *pepid_* header field of outgoing packets; increases the buffer value every time a packet is sent.

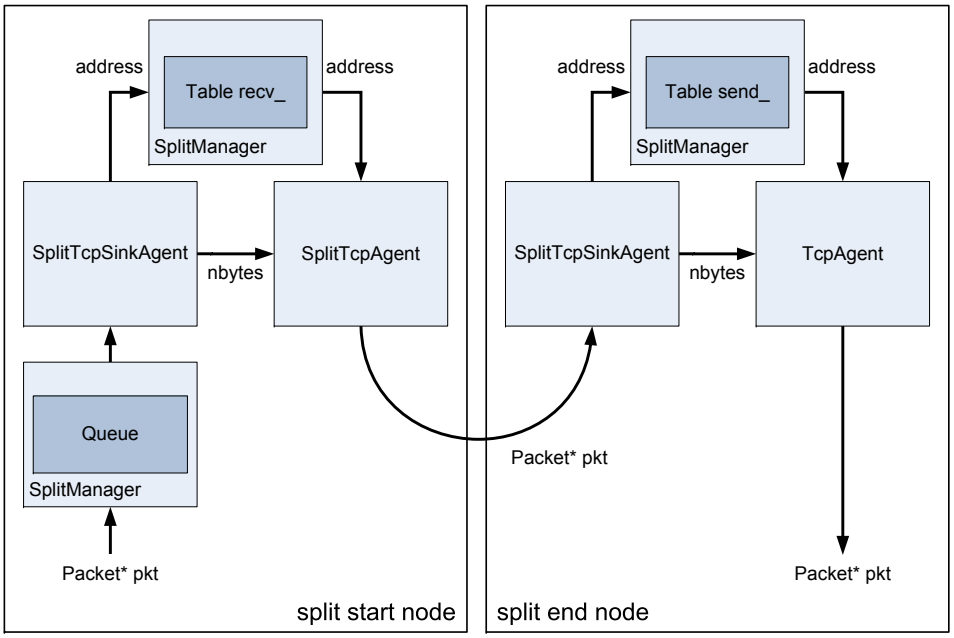

Figure 5.1: The split start and split end nodes use modified versions of *Sack1TcpAgent* (*LinuxTcpAgent* for Linux variants) and *DelAckSink* for the data forwarding mechanism. As illustrated by the figure, the *Queue* and *Table* objects are accessed through *SplitManager*.

5.1.2 Modified TCP Source

The second part of this performance enhancing proxy is a modified TCP sender called *SplitTcpAgent* (TCL name *Agent/TCP/Sack1/Split*; C++ class files `ns-base-dir/tcp/split-tcp.*`), which is a subclass of *Sack1TcpAgent*. It is used on the split start node only. When the receiving side of the node gets a packet, its payload is handed to the application layer and immediately forces the SplitTcpAgent to send a new packet on the split link with the payload data. However the SplitTcpAgent instance on this node is in control of the flow of packets that are received by the split start node. When a packet arrives at the node, it is immediately enqueued in the node's local queue without any further processing. SplitTcpAgent checks this queue periodically (the time interval can

Table 5.3: Function call chain that illustrates packet processing when receiving a packet at the split end node.

Function name	Description
SplitTcpAgent::recv()	Called whenever a packet is received; adds the address contained in the *pepid_* field to the *send_* table and hands the packet to DelAckSink::recv(), which in turn calls SplitTcpAgent::recvBytes() with a number of payload bytes that are delivered (*nbytes*, see figure 5.1).
SplitTcpAgent::recvBytes()	Removes addresses from the *send_* table and forwards the payload to the corresponding addresses by calling TcpAgent::sendmsg().

be configured through the SplitManager via the parameter *send_check_interval_*) and only if the current situation allows data to be received and forwarded, it dequeues the packet from the queue and hands it to the *recvFinal()* method of the receiving TCP stack.

The periodic checking is implemented via a ns-2 *TimeHandler* object called *SendTimer*. Every time the timer fires, the SplitTcpAgent method *checkSend()* checks whether there are any packets in the queue and whether the flow conditions allow data to be forwarded. If both conditions are true, it hands the packet to the corresponding TCP sink of the node for acknowledging (since there are multiple TCP sinks installed on the split start node, the correct TCP sink is determined by the packet's destination port). However since this is done in a loop, we can check and forward multiple packets during one interval. Therefore the sending rate is only indirectly limited by the timer interval (i.e. too long intervals can cause burst behaviour and possibly result in bad link utilisation). But one must keep in mind the additional delay this timer can introduce if set too high. A timer interval of 0.03 for example in combination with the split link's queue may result in smooth sending rates, however if — at some point in time — the queue

is empty or the buffer indicator is too low, the loop ends. Packets arriving just after the loop has ended experience an additional delay of up to 30 milliseconds. Therefore the default value for this interval is chosen small enough (0.01 seconds, see table 5.1). When there are no more packets or if the flow control does not allow the forwarding of more data (i.e. the buffer indicator is too low), the loop ends and the timer is rescheduled to be fired again in the time provided by *check_send_interval*.

Checking periodically for new packets is a safer way of implementing the PEP than relying solely on an event-based structure, since only using events can cause the PEP system to become bursty or even completely stall for some time. Let us assume that we use an event-based implementation, that forwards the payload of packets upon their reception at the split start node (i.e. such a forwarding event is a simple function call). Furthermore we assume a scenario in which the end senders have sent enough packets to completely fill the local queue and that they do not need to send more packets for some time. Since enqueueing packets on a queue in ns-2 itself cannot cause further external action outside the queue, an event-based implementation would have to generate forwarding events upon the reception of packets before they are enqueued or dropped. However this means that the forwarding of data of enqueued packets on the split start node is exclusively driven by the end senders' transmission of packets. If — for any reason — the forwarding on the split start node is delayed (e.g. due to congestion on the split link), buffered data on the split start node would then only be forwarded again when new packets arrive (whether they are dropped or not). But under the previous assumption that the senders do not need to send additional data, "new" packets will only arrive when the retransmission timer of a sender expires, therefore introducing an unnecessary delay to the forwarding path (since the RTO timeout should rather be larger than smaller as suggested in [28]). Furthermore if the forwarding over the split link is delayed, incoming events would have to be neglected (if they were accepted despite any congestion on the split link, there would be no flow control at all). This means that there can be packets in the queue, the forwarding events for which got lost. As a result, assuming a scenario including long-lived traffic and a queue size of 100, the last

100 packets might not be forwarded at all since the corresponding events that cause the forwarding are missing. To avoid such problems, the forwarding on the split start node is scheduled via *SendTimer* to separate buffered packets from their forwarding events. On the split end node, there is no need for such a timer since there is no local queue and data can be handed to the sending TCP agents immediately.

SplitTcpAgent also holds a reference to the SplitManager to get information like packet sizes and to access the local queue, and it also holds references to all TCP sinks on that node since it needs to be able to forward the dequeued packets to them for acknowledging. Both references are assigned through the TCL script.

The only method of TcpAgent that is overridden is *output()*, which is responsible for assembling and sending packets. This method is almost a complete copy of the original, with two small modifications. First we need to set the *pepid_* field of the outgoing packet, so the split end node knows which end receiver is the intended destination of this data. This is determined by removing the first entry from the *send_* table of the SplitManager. Second, the SplitTcpAgent needs to increase the buffer_count_ parameter since it sends data from its buffer and therefore can receive new data to be sent.

To enable simulations with the PEP that use the Linux TCP variants supported by ns-2, the same modifications apply for ns-2's *LinuxTcpAgent* classes, enabling the same functionality for any Linux variant of TCP (e.g. CUBIC). The C++ class for the Linux variant is called *SplitTcpLinuxAgent* (TCL name *Agent/TCP/Linux/Split*; C++ class files `ns-base-dir/tcp/split-tcp-linux.*`).

5.1.3 Modified TCP Sink

The third part the PEP consists of a modified version of ns-2's *DelAckSink*, a TCP sink using delayed ACKs, called *SplitTcpSinkAgent* (TCL name: *Agent/TCPSink/Sack1/DelAck/Split*; C++ class files `ns-base-dir/tcp/split-tcp-sink.*`). It is used on both split nodes, at the

start and at the end of the split link, although exhibiting different behaviour depending on the position. On the split start node, when receiving a packet, the packet is simply enqueued in the queue held by SplitManager. Whenever the SplitTcpAgent allows this packet to be received (see chapter 3 for a description of the mechanism), it is handed to SplitTcpSinkAgent's *recvFinal()* method. It adds the packet's source address into the *recv_* address table (also held by SplitManager) and invokes the normal TCP sink *recv()* method (responsible for acknowledging). When there is data that can be handed over to the application, the *recvBytes()* method is called. Here, the data is handed to the SplitTcpAgent with the second split node as its destination.

At the split end node, the SplitTcpSinkAgent simply adds the packet's *pepid_* to the *send_* table of SplitManager and calls the *recv()* method of DelAckSink. When the *recvBytes()* method is ultimately called, the corresponding address of the data to be forwarded is removed from the table, and the packet is forwarded to its destination.

Since the same SplitTcpSinkAgent implementation is used on both nodes of the split link, the mode of operation (*SPLITSTART* or *SPLITEND*) has to be configured. Furthermore the agent used at the split start node needs a reference to the destination agent (i.e. the TCP sink installed on the split end node) — and likewise, the split end node implementation needs references to all sending TCP stacks to be able to forward the data to the end receivers. This is all configured via TCL (see section 5.2).

5.1.4 Other Changes

Further changes to the existing ns-2 code include an extension of the IP header (contained in the C++ files `ns-base-dir/common/ip.*`). A new field *pepid_* has been added to store the original source address of the data after it is re-packed and sent on the split link. When the packet is received by the split end node, this field is read to ensure correct forwarding to the respective destination. For simulation purposes, introducing a new field in the IP (or any other) header poses

no danger of interfering with normal operations since the existing implementation is simply unaware of it and the header sizes are saved in separate variables and are not directly related to the number or size of the header fields.

5.2 TCL Usage Examples

A SplitManager TCL object can be created and configured as follows:

```
set split_manager [new Application/SplitManager]
$split_manager local-queue $manager_queue
$split_manager set number_of_connections_ 10
$split_manager set buffer_start_ [expr 800*1040]
$split_manager set header_size_ 40
$split_manager set packet_size_ 1000
$split_manager set send_check_interval_ 0.003
```

The values for the parameters buffer_start_, header_size_ and packet_size_ are specified in bytes. The buffer_start_ variable in the example is set to 800 packets with 1040 being the total packet size. The unit for send_check_interval_ is seconds. The specified queue *manager_queue* can be any queue implemented in ns-2, for example *Queue/DropTail*. It can be created like any other ns-2 TCL object, however it should be noted that there may be some additional requirements to be met for certain queueing mechanisms. *Queue/RED's* automatic configuration for example uses knowledge about the bandwidth and delay of the link it is normally installed on (e.g. to automatically calculate its threshold values, see section 3.4.2 for a more detailed description of the automatic configuration). For that, it references the link to access its delay and bandwidth parameters. Since there is no link for the local usage of the queue, one may need to create a dummy link object — with the appropriate delay and bandwidth properties of the satellite link — for such queues to work properly. Since the sole purpose of this reference is the retrieval of bandwidth and delay information, this solution may be necessary for such mechanisms to work as desired, however the fact that

it is only a dummy link has no impact on the simulation results. The creation of such a dummy link together with a RED queue is shown below:

```
set local_link [new DelayLink]
$local_link set bandwidth_ 50Mb
$local_link set delay_ 250ms

set manager_queue [new Queue/RED]
$manager_queue link $local_link
$manager_queue set limit_ 1000
$manager_queue reset
```

The following TCL example code creates a TCP sender for the split connection and sets its reference to the manager:

```
set tcp_splitstart_send [new Agent/TCP/Sack1/Split]
$tcp_splitstart_send manager $split_manager
```

The creation of a TCP sink for the split connection is similar, but the type (*SPLITSTART* or *SPLITEND*) needs to be specified. Furthermore the TCP sender needs references to the TCP sinks for packet acknowledging:

```
for {set i 0} {$i < $numOfConns} {incr i} {
 set tcp_splitstart_recv($i) [new Agent/TCPSink/Sack1/DelAck/
     Split]
 $tcp_splitstart_recv($i) type SPLITSTART
 $tcp_splitstart_recv($i) manager $split_manager
 $tcp_splitstart_send acker $tcp_splitstart_recv($i)
}
```

Furthermore, the forwarders need to be set:

```
for {set i 0} {$i < $numOfConns} {incr i} {
 $tcp_splitstart_recv($i) forwarder start $tcp_splitstart_send
 $tcp_splitend_recv forwarder end $tcp_splitend_send($i)
}
```

If the PEP should use Linux TCP, there are two modifications to be done. First, the TCP sender object must be of the type *Agent/TCP/Linux/Split* in-

stead of *Agent/TCP/Sack1/Split*. Second, one must specify the congestion control algorithm to be used by means of a scheduled command as in the following example:

```
$ns at 0.0 "$tcp_splitstart_send select_ca cubic"
```

5.2.1 Sample TCL Script split.tcl

The sample TCL simulation script creates a dumbbell topology (similar to the one shown in figure 3.1) with a configurable number of end nodes that use FTP to generate traffic over a shared bottleneck link. It uses five parameters that need to be specified when the simulation script is run:

usePEP This parameter is used as a boolean flag to enable or disable the use of the PEP. When the PEP is not in use, all TCP sender simply connect to all TCP receivers over the split link with no flow intervention from the split nodes. If the PEP is enabled, the TCP senders send their data to the split start node, it forwards the data to the split end node and the split end node forwards it to the TCP receivers.

bandwidth With this value, one can specify the bandwidth of the split link in megabit per second. The delay is unchangeable and set to 250 ms.

numOfConns The parameter numOfConns specifies the number of TCP flows, hence also the number of TCP sender and receiver end nodes.

transferEndTime The duration of the file transfers in seconds can be specified with this parameter.

tcpMode This option specifies the protocol to be used between the split nodes. One can either set the mode to "sack", thereby using standard TCP behaviour, or one can specify one of the Linux congestion control variants that are implemented in ns-2 (e.g. "cubic", "compound", etc.).

A sample scenario with the PEP in use, a split link bandwidth of 50 megabit/s, 20 TCP flows, a data transfer time of 600 seconds and CUBIC being the TCP

variant used for the split connection can therefore be simulated by calling ns-2 as follows:

```
ns split.tcl 1 50 20 600 cubic
```

The sample script generates output files with measurement results collected during the simulation. Measurements are taken every 1.0 seconds. The data is saved in a subdirectory *data/**A**pep-**B**mb-**C**num-**D**time-**E***, where the letters **A**-**E** denote the parameters with which the script was run. The content of the output files is as follows:

xfbw_recv(i).tr contains the application throughput of end receiver i

xfbw_recv_avg.tr contains the average application throughput of all end receivers

xfcw_send(i).tr contains the congestion window of end sender i

xfbw_splitstart_recv.tr contains the application throughput at the split start node (only available when the PEP is enabled)

xfcw_splitstart.tr contains the congestion window of the split start node (only available when the PEP is enabled)

xfbw_splitend_recv.tr contains the application throughput at the split end node (only available when the PEP is enabled)

xftracebw_splitend_recv.tr is a tracefile of the utilisation of the split link

The trace data for the split link utilisation is accumulated by a TCL procedure called *trace_splitend_recv* {*a*}. To be able to evaluate this data like all other measurement variables by the *record* { } procedure, the sample script installs the procedure as a *trace-callback*. The procedure is then called every time a packet is received at this node and adds the packet size to a variable called *trace_bw_splitend_recv*. Installing such a *trace-callback* can be done as shown below:

```
$ns duplex-link $splitstart $splitend $split_bandwidth
    $split_delay DropTail
```

```
set split_link [$ns link $splitstart $splitend]
$split_link trace-callback $ns trace_splitend_recv
```

5.3 ns-2 Resource Requirements and Limitations

In reality, a network packet holds a small number of headers. For example for a packet belonging to Internet traffic being transmitted on an Ethernet cable, it is reasonable to assume that — in short — the Ethernet frame consists of an Ethernet header (and footer), an IP header and a TCP header, followed by the TCP payload which can carry any application information. The size of this packet is the sum of the sizes of the application-level data and all its headers. If routers need to process, store or deal in any way with this packet, the resources must suffice for this packet size. Furthermore, routers need time to process packets, whether they are just routing them or modifying information as a performance enhancing proxy possibly would.

Since ns-2 is a simulator and does not exactly resemble network reality, there are three major differences which need to be taken into account. First, packets in ns-2 do not carry real payload data (unless special transport agents are used which explicitly allow the transfer of application data, see section 5.1), they only consist of headers and a simple integer value representing their size. Therefore the virtual size of the payload has no impact on the resource requirements to simulate that packet. However, compared to real networks, ns-2 has much higher resource requirements regarding the packet's headers. Contrary to a real packet, only holding the packet header information of the protocols that are actually in use, ns-2 reserves — by default — memory for every header that might be used for a packet. Therefore, every packet — no matter which information it carries or what protocols are used for the current simulation — holds headers for every protocol known to and implemented in ns-2 (e.g. multicast protocols, wireless sensor and mobility protocols, multimedia extensions, etc.), which represents a huge overhead for most simulations.

Furthermore ns-2 keeps almost all packets that occur in its simulation in the memory until the simulation has ended, hence the memory requirement is directly proportional to the simulation time. To mitigate this issue, it is necessary to command ns-2 to remove or rather omit unnecessary packet headers. This is documented in the file `ns-base-dir/tcl/lib/ns-packet.tcl`. Headers can be omitted in `ns-packet.tcl` or directly in the simulation script prior to instantiating the simulator object `ns` with the commands `remove-all-packet-headers` and `add-packet-header IP TCP` for example. This can greatly reduce the memory requirements of ns-2, depending on the simulation scenario [22].

The third difference is that ns-2 does not simulate processing time. Routers naturally need a small amount of time to process information and act accordingly, even for simple tasks as just forwarding packets. Moreover data cannot be generated at arbitrarily high rates. This is not the case for simulations done with ns-2. In its simulations there is no processing time, complex tasks can be performed with zero time consumption and data can be generated at any rate. For this reason, the discussion of the simulation results assumes a steady state system, skipping the initial start-up phase of the network scenarios.

A further limitation of ns-2 is the duration of the simulation time. Although the source code of ns-2 is generally written with respect to large scenarios and advice is provided in [22] on how to cope with large simulations (both regarding topology and time), the simulation time is limited in some way. Time is generally represented by a *double* variable, in the ns-2 scheduler and otherwise. However depending on the number of TCP Agents (and their workload), it is possible that some time values still overflow the relatively large number range provided. As a result, timestamps can suddenly hold negative numbers at some point in simulation time. One sign of such an overflow is ns-2 exiting with an error message such as *"TcpAgent: negative RTO! (-429496729.600000)"*. Furthermore, this error does occur in a peculiar manner. Let us assume for example that a specific simulation scenario completes successfully for a total simulation time of 4000 seconds. If the error occurs for longer simulation times, it does not necessarily occur after the 4000 second point in time has passed (after which one

would expect the overflow to happen since the simulation complete successfully for a duration of 4000 seconds), but at any time — sometimes also just after a few seconds of simulation time. After extensive investigation, the only solution that is both suggested and working for such errors is to reduce the overall simulation time, hence avoiding such overflows. Therefore, this poses a time constraint for simulating scenarios that depend on long simulation times, such as simulations involving large round trip times as they were done for this thesis.

Chapter 6

Conclusion

This thesis has shown that despite many extensions to TCP over the years, there is still room for improvement that cannot be achieved by small modifications of the protocol. Related work has been presented, which has shown both the need for performance enhancing proxies and the performance gains one can achieve by deploying them. The PEP presented in this thesis — using a split connection approach — shows that using such a mechanism can improve the performance of transport protocols like TCP significantly, although depending on the specific circumstances. The measurement results that have been illustrated and discussed in chapter 4 show that the PEP shows little to no improvement for queue sizes which resemble the bandwidth-delay product of a link. However since this product is comparatively large for links with a high bandwidth and a long signal propagation delay such as satellite connections, the effect of smaller queue sizes has also been investigated. The outcome of the simulation experiments done in ns-2 indicates that a split connection PEP, that uses different protocols than standard TCP variants like TCP SACK, can lead to performance improvements between a few percent up to a factor of 5 for flows that transfer large amounts of data. For web traffic — resembled by a standard Pareto distribution — the use of this PEP still shows performance gains for queue sizes that are smaller than the bandwidth-delay product, up to 20%.

Furthermore investigating the use of different queuing mechanisms used locally to buffer the data to be forwarded by the PEP system has shown the different

characteristics of those queueing techniques and their effect on the end-to-end throughput of single flows. Since DropTail is based on a simple decision without any random factors, it can lead to phase effects that reduce the overall effectiveness of a network and leaves resources unused. Furthermore these phase effects reduce the fairness between TCP flows and can even lead to the starvation of flows. Using RED mitigates these disadvantages, since its decision making (on whether to drop a packet) is partially probability-based. Hence, it provides higher fairness for multiple competing TCP flows. For these reasons — although DropTail provides the same overall utilisation and average throughput — RED seems to be the preferred mechanism for the local queue used by the PEP

The field of PEPs offer a great variety of future work to be done, despite the fact that there has already been much research in the past. Even if one limits their research to PEPs implementing the idea of split connections, there are still many characteristics that can be investigated. Future efforts regarding this topic might include new means of flow control, different transport protocols on the split link or different queuing techniques besides those presented and discussed in this thesis.

List of Figures

List of Tables

Bibliography

[1] Ian F Akyildiz and Seong-Ho Jeong. Satellite ATM Networks: A Survey. *IEEE Communications Magazine*, (5):1–3, 1997.

[2] Ian F. Akyildiz, Giacomo Morabito, and Sergio Palazzo. TCP-Peach: a new congestion control scheme for satellite IP networks. *IEEE/ACM Trans. Netw.*, 9(3):307–321, 2001.

[3] M. Allman, S. Dawkins, D. Glover, J. Griner, D. Tran, T. Henderson, J. Heidemann, J. Touch, H. Kruse, S. Ostermann, K. Scott, and J. Semke. Ongoing TCP Research Related to Satellites. RFC 2760 (Informational), February 2000.

[4] M. Allman, D. Glover, and L. Sanchez. Enhancing TCP Over Satellite Channels using Standard Mechanisms. RFC 2488 (Best Current Practice), January 1999.

[5] Ajay Bakre and B.R. Badrinath. I-TCP: Indirect TCP for Mobile Hosts. pages 136–143, 1995.

[6] Hari Balakrishnan, Venkata N. Padmanabhan, Randy H. Katz, and Y H. Katz. The Effects of Asymmetry on TCP Performance, 1997.

[7] Hari Balakrishnan, Srinivasan Seshan, Elan Amir, and Y H. Katz. Improving TCP/IP Performance over Wireless Networks. pages 2–11, 1995.

[8] Pravin Bhagwat, Partha P. Bhattacharya, Arvind Krishna, and Satish K. Tripathi. Using channel state dependent packet scheduling to improve TCPthroughput over wireless LANs. *Wireless Networks*, 3(1):91–102, 1997.

[9] J. Border, M. Kojo, J. Griner, G. Montenegro, and Z. Shelby. Performance Enhancing Proxies Intended to Mitigate Link-Related Degradations. RFC 3135 (Informational), June 2001.

[10] B. Braden, D. Clark, J. Crowcroft, B. Davie, S. Deering, D. Estrin, S. Floyd, V. Jacobson, G. Minshall, C. Partridge, L. Peterson, K. Ramakrishnan, S. Shenker, J. Wroclawski, and L. Zhang. Recommendations on Queue Management and Congestion Avoidance in the Internet. RFC 2309 (Informational), April 1998.

[11] Ramon Caceres and Liviu Iftode. Improving the Performance of Reliable Transport Protocols in Mobile Computing Environments. *IEEE Journal on Selected Areas in Communications*, 13:850–857, 1994.

[12] C. Caini and R. Firrincieli. TCP Hybla: a TCP enhancement for heterogeneous networks. *International Journal of Satellite Communications and Networking*, 22(5), 2004.

[13] C. Caini, R. Firrincieli, and D. Lacamera. PEPsal: a Performance Enhancing Proxy for TCP satellite connections. *IEEE Aerospace and Electronic Systems Magazine*, 22(8):7–16, 2007.

[14] D. Clark. The design philosophy of the DARPA internet protocols. *SIGCOMM Comput. Commun. Rev.*, 18(4):106–114, 1988.

[15] Robert C. Durst, Gregory J. Miller, and Eric J. Travis. TCP Extensions for Space Communications. pages 15–26, 1996.

[16] N. Ehsan, M. Liu, and R.J. Ragland. Evaluation of Performance Enhancing Proxies in Internet over Satellite. *International Journal of Communication Systems*, 16(6), 2003.

[17] M. Fiorenzi, D. Girella, N. M
"oller, Å. Arvidsson, R. Skog, J. Petersson, P. Karlsson, C. Fischione, and KH Johansson. Enhancing TCP over HSDPA by cross-layer signalling. In *IEEE Global Telecommunications Conference*, volume 1, pages 5348–5352, 2007.

[18] S. Floyd, J. Mahdavi, M. Mathis, M. Podolsky, and A. Romanow. An extension to the selective acknowledgement (SACK) option for TCP, 1999.

[19] Sally Floyd. RED: Discussions of Setting Parameters. `http://www.icir.org/floyd/REDparameters.txt`, 1997.

[20] Sally Floyd. Recommendation on using the gentle variant of RED. `http://www.icir.org/floyd/red/gentle.html`, 2000.

[21] Christian Huitema. *Routing in the Internet*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1995.

[22] Information Sciences Institute, University of Southern California. The Network Simulator ns-2: Tips and Statistical Data for Running Large Simulations in NS. `http://www.isi.edu/nsnam/ns/ns-largesim.html`, 2002.

[23] V. Jacobson, R. Braden, and D. Borman. TCP Extensions for High Performance. RFC 1323 (Proposed Standard), May 1992.

[24] Mikkel Christiansen Kevin, Kevin Jeffay, David Ott, and F. Donelson Smith. Tuning RED for Web Traffic. In *in Proceedings of ACM SIGCOMM 2000*, pages 139–150, 2000.

[25] M. Marchese, M. Rossi, and G. Morabito. PETRA: Performance Enhancing Transport Architecture for Satellite Communications. *IEEE JOURNAL ON SELECTED AREAS IN COMMUNICATIONS*, 22(2), 2004.

[26] Christopher Metz. TCP over Satellite... The Final Frontier. *IEEE Internet Computing*, 3(1):76–80, 1999.

[27] C Partridge and T J Shepard. TCP/IP Performance Over Satellite Links. In *Internet Node Technology Development Project Protocol Architecture Model Report*, 1997.

[28] V. Paxson and M. Allman. Computing TCP's Retransmission Timer. RFC 2988 (Proposed Standard), November 2000.

[29] Fei Peng, Lijuan Wu, and Victor C. M. Leung. Cross-layer enhancement of TCP split-connections over satellites links. *International Journal of Satellite*

Communications and Networking, 24(5):405–418, 2006.

[30] I. Rhee and L. Xu. CUBIC: A new TCP-friendly high-speed TCP variant. In *Proc. PFLDnet*, volume 2005, 2005.

[31] J. H. Saltzer, D. P. Reed, and D. D. Clark. End-to-end arguments in system design. *ACM Trans. Comput. Syst.*, 2(4):277–288, 1984.

[32] D. Velenis, D. Kalogeras, and B. Maglaris. SaTPEP: A TCP performance enhancing proxy for satellite links. *Lecture notes in computer science*, pages 1233–1238, 2002.

[33] Michael Welzl. *Network Congestion Control: Managing Internet Traffic (Wiley Series on Communications Networking & Distributed Systems)*. John Wiley & Sons, 2005.

Acknowledgements

I would like to thank my supervisor, Dr. Michael Welzl for his continued help and support for this thesis. Furthermore I thank Dragana Damjanovic for her assistance and many insightful conversations during the research and investigation of this topic.