

UNIVERSITY OF OSLO
Department of Informatics

**A Linux implementation
and analysis of the
eXplicit Control Protocol
(XCP)**

Master thesis

Petter Mosebekk

24th May 2005



Abstract

The eXplicit Control Protocol (XCP) was released as an RFC draft on October 17, 2004. In this paper we have looked into the theory behind the development of XCP. In addition, we created a working Linux implementation of XCP as part of our investigation of the protocol. This paper also discusses the performance results we got by using our implementation of XCP, and compares them to TCP and other papers regarding XCP.

The XCP protocol has been developed as a new way to improve the congestion control, bandwidth utilization and congestion avoidance algorithm in TCP. In contrast to most other enhancements of the TCP protocol, XCP does not try to be backward compatible with existing TCP implementations. XCP introduces a new layer in the TCP/IP stack, which is used to add an XCP header in front of the regular TCP packet. The main difference between TCP and XCP is that an XCP stream allows routers to explicitly set how much bandwidth to allow in the XCP packets passing through them. By using explicit feedback from the network, XCP promises to prevent queue buildup in routers and packet drops caused by congestion. XCP allows the use of more aggressive algorithms, than in use by TCP, in order to quickly distribute any available bandwidth amongst XCP-enabled TCP-flows.

Our Linux implementation was based on the original XCP RFC draft. We implemented XCP as a separate protocol, situated between TCP and IP in the Linux kernel. Our goal was to create an implementation of XCP that would run without the need to change the existing TCP or IP code. By creating the XCP protocol as a Linux kernel module, we managed to achieve this goal. However, the very nature of XCP made implementing it as a separate protocol difficult, as the XCP protocol needed intimate knowledge of various TCP concepts; such as the congestion window and TCP-flows.

Simulations have shown XCP to be able to prevent queue buildup in routers, while at the same time maximizing throughput. Our tests managed to confirm these results, showing that XCP can be superior to TCP in some environments. However, our tests also show that XCP is vulnerable to a number of common scenarios, where it fails to work as intended. Scenarios such as half-duplex links, bursty applications and incorrect router setup make the XCP routers return invalid feedback back to the XCP hosts. Incorrect feedback from the XCP routers can lead to queue build-up, packet drops and oscillatory behavior, which are the same problems as XCP set out to solve.

Table of Contents

1	Introduction.....	1
2	XCP – The basic idea	5
2.1	Senders.....	9
2.1.1	Calculating the delta throughput.....	9
2.1.2	Reacting to XCP feedback	10
2.2	Routers.....	11
2.2.1	Packet arrival	11
2.2.2	Control Interval Timeout.....	12
2.2.3	Packet departure.....	14
2.2.4	Queue estimation timeout.....	14
2.3	Receivers.....	15
2.4	Summary	15
3	Implementing XCP in Linux.....	17
3.1	XCP protocol layering issues	19
3.2	XCP specification issues	20
3.3	Implementing the host side of the XCP protocol	22
3.4	Implementing the XCP router	24
3.4.1	Calculations on packet arrival	25
3.4.2	Control Interval Calculations.....	26
3.4.3	Calculations on packet departure.....	27
3.5	Summary	28
4	XCP Performance results.....	29
4.1	Overestimating the congestion window	30
4.2	XCP Router performance.....	34
4.3	Per flow performance.....	37
4.4	Half duplex XCP router	38
4.5	Sender not utilizing the full out-bandwidth	41
4.6	Summary	45
5	Conclusions and remaining challenges	47
6	References.....	51

1 Introduction

In recent years computer networks have become ubiquitous and wireless networks have seen an exponential increase in popularity. As the speed and availability of data networks continues to increase, consumers demand increased complex application contents to run over these networks. Applications that stream video are examples of network intensive applications, often preferring constant transfer speeds in addition to large amount of bandwidth. With the rapid increase in bandwidth of conventional and wireless networks, the *Bandwidth-Delay* product of the network will increase as well. The *Bandwidth-Delay* product is a measurement of how much data that is “in transit” in the network. When the delay or bandwidth increases in a network, the number of unacknowledged bytes that is in transit between the sender and receiver increases as well.

For the last two decades, the TCP [5] protocol has been the most widely used protocol on the Internet. During these two decades the world of computing has changed dramatically, and in ways not foreseeable by anyone. Computing power, network speeds and storage capacities have all increased at an enormous rate. Amidst all these changes, the TCP/IP protocol has been quite resilient, and although modifications to the protocol have been done over the years, it is still based on the same fundamental principles.

The original TCP specification did not put much thought into the question of congestion control, as it was little known issue at the time of development. Multiple enhancements of TCP has been introduced later in order to alleviate various problems discovered over the years. In order for multiple TCP flows not to oscillate under high loads and for TCP to work better in high speed environments, various algorithms such as Random Early Discard (RED) [6], Random Early Marking (REM) [8], Fast Recovery [9], Slow Start [9], Fast Retransmit [9], Forward Acknowledgment (FACK) [11], Selective Acknowledgment (SACK) [12] and Highspeed TCP [13] have been invented. However, mathematical analyses of current congestion control algorithms reveal that, as the *Bandwidth-Delay* product increases, TCP becomes oscillatory [7]. Oscillatory behavior can be counter productive for applications such as multimedia streaming applications, requiring a high constant bit rate.

Another issue with TCP is the way it allocates bandwidth on networks with high *Bandwidth-Delay* products. TCP’s AIMD (Arithmetic Increase, Multiplicative Decrease) algorithm prevents TCP from quickly using the available bandwidth in such scenarios. AIMD prevents TCP from increasing the bandwidth used by more than one packet per round trip time (RTT). If a packet is lost, TCP will use the “Multiplicative Decrease” part of the AIMD algorithm to reduce (halve in the case of TCP Reno) its throughput. From that half it will “Additively Increase” its sending rate by adding one packet per RTT. Networks with high RTT will therefore use quite

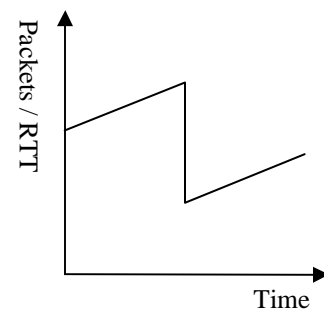


Figure 1: TCP Reno AIMD

some time to get up to speed (Figure 1). The different variations of the TCP protocol use different algorithms to detect packet drops, and react somewhat different when a drop has been determined. However, they all employ the arithmetic increase algorithm (except Highspeed TCP) so increasing sending speed when in the congestion avoidance phase can be time consuming. As Example 1 shows detecting and preventing TCP from entering the congestion avoidance phase could speed up a transfer significantly.

Consider a high speed network, with an RTT of 100ms. Given a packet size of 1.500 bytes, the speed TCP would increase the sending rate during congestion avoidance phase using AIMD is at about 15.000 bytes / second, which is very slow if you have a fast (Gigabit/second)-link.

Example 1: Ineffectiveness of the AMID algorithm

TCP uses packet drops as a way of detecting congestion, and thereby limiting its sending rate. This implicit signal gives the sender of data very little information on how to best react to a network problem. Not only does the “signal” give very little feedback, but it is not possible to detect a packet loss quickly either. As the timeout period is basically equal to the RTT of the network, TCP will use longer time to detect packet loss when the RTT is high. Using packet drops to adjust the available bandwidth becomes problematic as TCP has to wait for a timeout to occur before adjusting its sending rate. This process limits the bandwidth utilization, and additionally this ineffectiveness increases with increased bandwidth and RTT.

From a TCP point of view, the network is a “black box” and TCP does not do any assumptions on the capabilities of the network. When TCP was invented one felt that it was important not to assume anything about the capability of the network in order for the protocol to work over any network. As computer networks and knowledge have evolved this assumption can be challenged, and the approach taken by Dina Katabi, the developer behind the eXplicit Control Protocol (XCP) [1], was to rethink the whole congestion control problem from scratch. Instead of suggesting a minor improvement to any existing protocol, XCP was developed with an open mind to the question:

“Given our current knowledge of congestion control, how would we ideally like congestion control to work?”

Katabi’s answer to this question was to have the network give explicit messages back to the hosts of the current congestion levels in the network. By actively involving the routers, the senders would get exact feedback on how to dynamically adjust to the load in the network. This detailed control of the senders allowed routers to more or less prevent packet loss altogether, thereby improving the utilization of the available resources in the network.

In this paper we will in Section 2 give an introduction to the design rationale behind XCP and discuss the structure of the XCP protocol. As the XCP protocol takes a revolutionary approach to congestion avoidance and control, we further explain the XCP functions

needed in the end systems and routers. In order for XCP to work, routers and end systems must coordinate the distribution of the available bandwidth, how this is done is also discussed in Section 2.

To be able to verify and investigate the XCP protocol further we created a Linux implementation of XCP. Section 3, presents some issues and problems we encountered while creating this implementation. As XCP bases much of its protocol on floating point arithmetic, which is not allowed in the Linux kernel, workarounds had to be introduced. Creating the XCP protocol as a separate protocol in the already existing TCP/IP stack caused various implementation issues, especially since XCP needs to control features of the TCP protocol.

Section 4, contains different test results we got with our Linux implementation, comparing XCP to the default TCP version running on Linux. The default TCP configuration in Linux kernel 2.6 uses the New Reno, SACK and FACK enhancements. The results show that the XCP protocol can deliver on its promises, but only in situations where the XCP router have a correct model of the network it is running on.

Summary of our findings and conclusions, with thoughts on the future for XCP, can be found in Section 5. All references are located in Section 6.

2 XCP – The basic idea

XCP takes a fundamentally different approach than TCP to congestion control. It assumes that the network consists of routers capable of calculating the current network load, and thereby letting the sender know how much bandwidth is available for it to use in the network. By letting the network give more information back to the sender, XCP tries to prevent congestion and packet drops. The use of packet drops as a signal of network congestion is inaccurate and slow. By nearly eliminating packet loss, XCP seems to be able to outperform TCP significantly [5]. Additionally, the fact that TCP interprets packet loss as network congestion makes it less than ideal for usage on wireless links where the loss of a packet might come from other sources than network congestion. The observation that a packet loss is a poor way to signal congestion is the basis for the XCP protocol.

The specification for XCP is currently available as a draft RFC [2] and is aiming towards becoming an experimental RFC. The protocol has been developed in an effort to improve the performance of TCP in network with high *Bandwidth-Delay* products. XCP was conceived without any backward compatibility features, but as a separate protocol layered between IP and TCP. The reason XCP was not implemented as an IP-options, but as a separate protocol, was to prevent IP packets to follow the “slow” path in routers. Routers noticing IP packets with IP options will need to inspect the option thoroughly. This inspection is quite a bit slower than the normal fast packet forwarding, and would have to be done on each IP packet; therefore XCP does not utilize this approach.

In order to get a less oscillatory protocol a more precise feedback mechanism than in use by TCP is needed. As the feedback delay increases with high RTT, the protocol needs to take this feedback delay into account, by having the sender change its sending rate more seldom. The important question is how the protocol should adapt to changing feedback delay in order to achieve stability even when the feedback delay gets very high. XCP will automatically slow down its adjustment rate of the sending speed when the feedback delays (RTT) increase. This adaptation to increased network delay prevents the protocol from becoming unstable and oscillating, in contrast to TCP [7].

Fairness is another issue with TCP, as it is biased towards low RTT flows. XCP in contrast to TCP decouples flow control from utilization control. This decoupling has multiple benefits, as one can specify what considers a “fair” sharing of bandwidth between multiple flows. This allows for service differentiation using schemes that are either too aggressive, or too weak to be used for controlling congestion. It also allows XCP to use a much more aggressive utilization control algorithm. On high bandwidth networks, the time it takes the TCP AIMD algorithm (Figure 1) to “fill the pipe” can often be longer than the flows duration, leading to poor utilization and performance. XCP uses an MIMD (Multiplicative Increase, Multiplicative Decrease) algorithm instead for utilization control. This leads to much faster allocation of available bandwidth. XCP will

allocate bandwidth proportionally with the available bandwidth (“Multiplicative Increase”), and equally reduce the bandwidth proportionally if too much bandwidth is used (“Multiplicative Decrease”). XCP must also determine how to distribute the allocated bandwidth amongst the active XCP flows. In contrast to TCP, XCP separates bandwidth allocation and per flow allocation. XCP uses an AIMD algorithm to determine fairness between flows, thereby still managing to be TCP-friendly. This is possible because how bandwidth is distributed between different flows is not dependent on how much bandwidth XCP distributes.

The XCP protocol is based on creating a new protocol layer and header in the protocol stack between IP and TCP. This header is 20 bytes long and is placed before the TCP header, but after the IP header. The XCP routers do not keep any state information about each flow, but calculates feedback values on a per packet basis. As the number of flows in a router is an unknown and quickly changing parameter, the congestion control mechanism should not be dependent on it. This allows for quite simple implementations in routers, and makes the protocol more scalable.

The XCP protocol works by having the sender set extra network information in the XCP packet. This information is calculated by the sender, and adjusted by the routers along the way. The receiver copies the data back to the sender, and the sender can then use this feedback to adjust its sending speed. The sender supplies the packet with information regarding the current estimated RTT, the current throughput and the delta throughput. (The sender's desired change in throughput).

For XCP to work there needs to be at least one XCP aware router along the flows' path in the network. However, unless all routers are XCP compliant, the protocol will not work optimally (but it will still work to some degree). If no XCP routers are located between two XCP hosts, the sender would just send data as fast as possible, as no XCP router would reduce the sender's requests for bandwidth. This would be comparable to using TCP without any congestion control, as the XCP protocol overrides TCP's congestion control scheme.

Each XCP router only needs to check the aggregated load on itself, and can then calculate if it will allow any change in throughput as specified by the sender. If the router is not overloaded, it will allow the sender to use the bandwidth they request, if this still does not overload the router. If the router would become overloaded by a sender's request for bandwidth, it reduces the request to be within the limits of the router's capacity. The recipient of the XCP message will copy back the allowed throughput in a specific reverse-feedback field. The sender will then know what throughput to use for the next packet. In the case where the XCP enabled router is experiencing congestion, it can reduce the allowed throughput from the sender, making the sender send less data. When the value is copied back to the sender by the receiver, the sender is able to adjust the bandwidth in accordance with the router's request in one RTT.

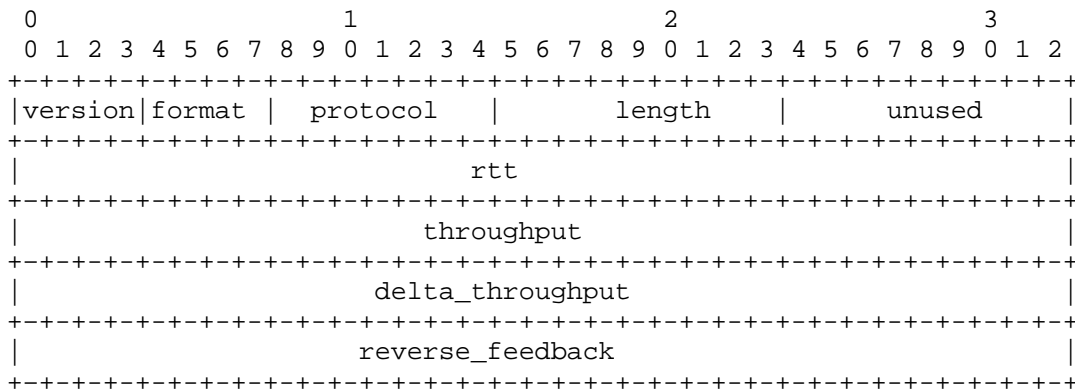


Figure 2: the XCP header

The XCP header (Figure 2) has a fixed size of 20 bytes and is specified in a draft RFC [2]. The header consists of the following fields:

- **Version: 4 bits**
Current version of XCP in use, value = 0x01
- **Format: 4 bits**
This field indicates the congestion header format. Two formats are currently defined, a standard format and a minimal format. The standard format indicates that the RTT, throughput, delta_throughput fields are in use. (Typically when a message has been sent from the sender, and is on its way to the recipient). A value of 0x01 indicates standard format.
The minimal format (value 0x02) is used when data is returned from the recipient back to the sender. The RTT, delta_throughput and reverse_feedback should be 0. Any XCP capable router seeing the minimal format header must not do any XCP calculation on these packets.
- **Protocol: 8 bits**
This indicates the next level protocol, typically TCP, to use in the data portion of the packet. The values for various protocols are specified by IANA (Internet Assigned Numbers Authority).
- **Length: 8 bits**
This field indicates the length of the congestion header, measured in bytes. For the current version of XCP this field has a constant value of 20.
- **Unused: 8 bits**
This field is currently unused and must be set to 0.

- RTT: 32 bits**
 This field indicates the smoothed round – trip time measured by the sender in milliseconds. This field is an unsigned integer. The minimum value expressible is 1 ms (values always rounded up). A 0 value indicates that the sender does not yet know the RTT. The maximum value is $4.3 * 10^9$ seconds or approximately 49 days.
- Throughput: 32 bits**
 This field indicates the current throughput for the flow, as measured by the sender, in bytes per millisecond. Throughput values should be rounded up. The maximum value expressible in this field is $4.3 * 10^9$ bytes/ms or 34.360 Gbps, in steps of 8000 bits/sec.
- Delta_throughput: 32 bits**
 This field indicates the desired change in throughput. It is set by the sender to indicate the amount by which the sender would like to adjust its throughput; this value may be subsequently reduced by routers along the path. It is measured in bytes per second and is a signed, 2's complement value. The minimum throughput change expressible in this field is -17 Gbps. The maximum value expressible in this field is 17 Gbps, in steps of 8 bits / second.
- Reverse_feedback: 32 bits**
 This field indicates the value of delta_throughput received by the data receiver. The receiver copies the field delta_throughput into the reverse_feedback field of the next outgoing packet in the same flow.

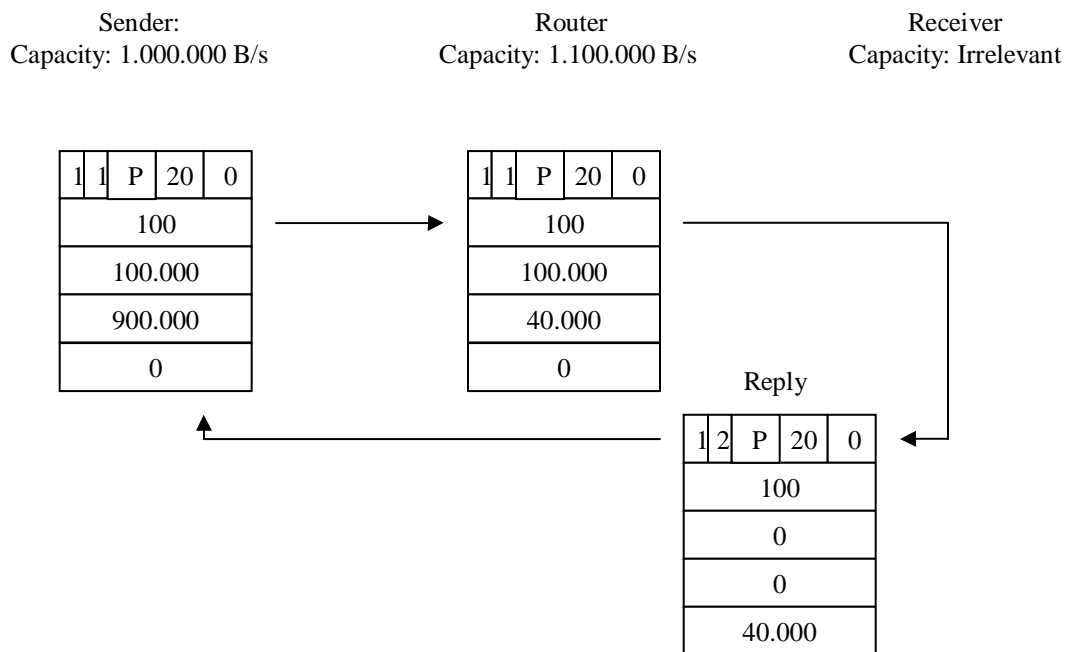


Figure 3: XCP header values at different stages in a network

The XCP protocol works by having the sender, receiver and all participating routers along a flows path read and edit XCP messages. In order for XCP to work, each of these systems needs to do some computations to allow for meaningful exchange of data. Figure 3 shows how a message containing an XCP header flows between the sender and receiver and back, via the XCP router. At each stage the XCP header fields are recalculated and updated. The following sections will describe in detail how these calculations and updates are done.

2.1 Senders

The senders are the computers actively transmitting data into the network. They are responsible for adhering to the XCP protocol and must not send more data than allowed by the XCP routers into the network. Most of the complexities of the XCP protocol are located at the senders, as they must set various parameters that the rest of the protocol needs to work correctly.

The sender is responsible for maintaining these four parameters:

- (1) A desired throughput value (bytes / second).
- (2) A current estimate of the actual throughput (bytes / second).
- (3) The maximum throughput allowed by XCP (bytes / second).
- (4) A current estimate of the RTT (milliseconds).

The desired throughput (1) value may be chosen by the sender to be any reasonable value, for example the speed of the local interface, or a maximum throughput value given through an API. It is the speed of which this flow wishes to run, if there were no congestion in the network.

2.1.1 Calculating the delta throughput

To be able to calculate the “delta throughput” in the XCP header, the sender needs to know its current throughput (2). Using the value for the sender’s desired throughput (1) and its current throughput (2) it can use the following formula to find the value of the delta_throughput to use in the XCP header:

$$d = \frac{C - (t * 1000)}{t * \frac{RTT}{MSS}} \quad (1)$$

Where:

- d* Delta throughput per packet, measured in bytes/second (B/s).
- C* The capacity, or maximum speed, of the sender, measured in B/s
- t* Current sending throughput at sender, measured in bytes/milliseconds.
- RTT* Round trip time, as seen from sender, measured in milliseconds.
- MSS* Maximum Segment Size is measured in bytes.

Formula (1) states that the delta throughput should be the difference between the current throughput and the desired throughput. This change in throughput is then divided on an estimated number of packets per RTT, given by the denominator: $t * (RTT / MSS)$. By spreading the total change in throughput onto each individual packet, the XCP router(s) can allow (or disallow) the throughput on a per packet basis. The reason for splitting the delta throughput amongst the packets leaving the sender is that the XCP routers do not keep any state information about each flow. The XCP routers base their allowance of bandwidth on the sum of all individual packets' request for bandwidth. Example 2 gives an example of how this calculation would work at the sender in a real network.

Given $RTT = 100\text{ ms}$ and $MSS = 1000\text{ bytes}$ and a desired speed of $1.000.000\text{ B/s}$. Assume the current speed is one packet per RTT, or $1000\text{ bytes per }100\text{ ms} \Rightarrow 10.000\text{ B/s}$

*The sender will then calculate the delta throughput per packet as follows:
 $(1.000.000\text{ B/s} - (10\text{ B/ms} * 1000)) / (10\text{ B/ms} * (100\text{ ms} / 1000\text{ B})) = 990.000\text{ B/s}$*

Example 2: Calculating delta throughput

In Example 2 the sender wishes to increase the sending speed by 990.000 bytes per second, and marks this in the packet. This is all the available bandwidth at the sender. The router will then use this request for bandwidth as a basis for allowing or disallowing the sender to increase its sending rate.

2.1.2 Reacting to XCP feedback

As the delta_throughput header field is returned to the sender by the receiver in the reverse_feedback field of the XCP message, the sender adjusts its TCP congestion window (cwnd) in order to adapt to the new bandwidth allowance. As a receiver can return fewer ACK-messages than received messages, the receiver needs to accumulate each packet's delta_throughput, and return the accumulated value with the next ACK message to the sender in the reverse_feedback field.

The formula used by the sender to adjust the congestion window is given by:

$$w = \max(w + r * RTT / 1000, MSS) \quad (2)$$

Where:

w TCP's current congestion windows, measured in bytes.

r Reverse_feedback field from received packet, measured in B/s.

RTT Sender's current round trip time estimate, measured in milliseconds.

MSS Maximum segment size, measured in bytes.

Formula (2) sets the minimum value of cwnd to MSS in order to avoid the “silly window syndrome” [10]. The “silly window syndrome” appears when the minimum cwnd becomes too small to allow TCP to send any data at all. To prevent cwnd to become 0, it is always set to be at least MSS bytes large. Example 3 shows how the congestion window is increased in response to the XCP feedback received from the network by using Formula (2).

Given $RTT = 100\text{ ms}$ and $MSS = 1000\text{ bytes}$ and desired speed = $1.000.000\text{ bytes / sec}$. The current cwnd is 1000 bytes . The sender receives a packet with the reverse_feedback field set to $900.000\text{ (bytes/sec)}$.

The sender's new cwnd therefore should be:

$$cwnd = \max(1.000\text{ B} + 900.000\text{ B/s} * (100\text{ ms} / 1000), 1000\text{ B}) = \underline{91.000\text{ B}}$$

Example 3: Calculating new congestion window based on feedback from receiver

2.2 Routers

In order for XCP to work, there must be at least one XCP capable router in the flows' path. This XCP router must calculate how to allocate bandwidth to each packet independently of the flow concept. In order to do that, the router keeps track of 4 different events. These events are occurring when packets arrive, when a control interval timer times out, on packet departure, and on a queue-assessment timer timeout. These four events all require calculations by the router and are done either at specific timeout intervals or when packets are entering / leaving the router. The following sections describe these events in more detail.

2.2.1 Packet arrival

The main calculations in the router are done during the Control Interval Timeout (see Section 2.2.2). In order to do these calculations, the XCP router needs to collect data from arriving XCP packets. The data is collected from the IP and XCP layers in each individual data packet, and the information gathered include:

- The total amount of bytes received (sum of all IP packet sizes)
- Total sum of weighted throughput (IP packet size / XCP throughput field)
- Total sum of weighted RTT ($RTT * IP\text{ packet size} / XCP\text{ throughput field}$)

The calculations are very simple which they need to be as they are done for each individual packet arriving at the router. The sums calculated are reset after each Control Interval.

2.2.2 Control Interval Timeout

The Control Interval Timeout appears at regular intervals, set to the average RTT experienced during the previous Control Interval. The main purpose of the calculations done during the Control Interval is to calculate the aggregate feedback, and how to distribute this feedback on a per packet basis. The XCP protocol will use the result of these calculations to adjust, if applicable, the delta_throughput field in the XCP messages during the next interval. During the previous Control Interval, statistics of the all packets arriving have been collected (see Section above), and the new feedback values are calculated based on this data. As the XCP protocol decouples utilization control from fairness control, these two calculations are done individually (and different formulas can be conceived to achieve best performance or to allow individual adjustments for special systems).

The calculations done are somewhat complex and time consuming (see below). Even though these calculations are computing intensive, they are only done once per average RTT of all flows and should therefore not affect the performance as much as if the calculations were needed for each packet.

2.2.2.1 The Efficiency Controller (EC)

The Efficiency Controller's task is to maximize the aggregated throughput through the router, without causing packet drops. It is not concerned with how any change in aggregated throughput is distributed among the individual flows. This is the job of the Fairness Controller (FC) (see Section 2.2.2.2). During the Control Interval Timeout the EC calculates what change in throughput that is needed to maximize throughput. The formula used is given by:

$$F = \alpha * d * S - \beta * Q \quad (3)$$

Where:

- F* The aggregated feedback (in bytes/second) during a Control Interval
- α* Constant. Suggested value is 0.4 [2]
- d* Average RTT of all flows during last Control Interval (in seconds)
- S* Spare bandwidth (bytes/second). This is the difference between the link capacity and link usage. Note that this value can be negative if the link is over – utilized.
- B* Constant. Suggested value is 0.226 [2]
- Q* The persistent queue size (in bytes). The persistent queue is the queue that does not drain in one round trip propagation delay.

Formula (3) allows spare bandwidth to be allocated proportionally to how much of the total bandwidth is available. Example 4 shows how the XCP feedback formula works, by calculating feedback based on the current load.

Assume a router with a total link capacity of 1.100.000 B/s, a current usage of 100.000 B/s, and the average RTT for all flows is 0,1 seconds (100 ms). Further assume there is no queue in the router. Formula (3) then gives:

$$F = 0,4 * 0,1 s * 1.000.000 B/s - 0,226 * 0 B = \underline{40.000 B/s}$$

In other words; for each RTT the aggregated sending speed of all senders should increase with 40.000 B/s. With an RTT of 100 ms there are 10 RTT's per second; hence this change equals 400.000 B/s per second. Formula (3) states that, given no queue, 40 % of the available bandwidth should be allocated to the flows passing the router during the next second.

Example 4: XCP's feedback function

2.2.2.2 The Fairness Controller (FC)

The Fairness Controller's task is to make sure that each packet in each flow passing through the router receives its fair share of any bandwidth feedback. The formulas used by the FC are not connected to the EC in any way, so what the FC considers "fair" can be tailored as one wishes.

In XCP the FC relies on the same principals as TCP, namely AIMD. The policy used by the FC compute the feedback per packet is dependent on whether the router is under- or over utilized.

If $F > 0$, (see formula (3)), the router is underutilized and the FC will increase the throughput of all flows with the same amount, regardless of the previous bandwidth usage. This leads to a relatively higher increase in throughput for flows running at low bandwidth, compared to high bandwidth flows. Example 5 shows how this distribution affects a router with multiple flows passing through it.

A router having one flow using 1.000.000 B/s, and ten flows using 10.000 B/s each, is going to distribute 440.000 B/s. This will give each flow additional 40.000 B/s, so the large flow will get 1.040.000 B/s and the small ones 50.000 B/s.

Example 5: XCP's additive increase algorithm

If $F < 0$, (see formula (3)), the router is over-utilized and the FC will decrease each flow's throughput proportionally to its current throughput. In amounts of actual bandwidth reduced, the ones that had most bandwidth will loose the most. As Example 6 shows, this algorithm helps to bring all flows to get the same share of the bandwidth eventually.

A router having one flow using 1.000.000 B/s, and 10 flows using 10.000 B/s each, needs to reduce the throughput by 120.000 B/s (10%). The large flow will be reduced to 900.000 B/s while the 10 small flows to 9.000 B/s each.

Example 6: XCP's multiplicative decrease algorithm

In addition to using the AIMD algorithm to ensure equality and convergence between the flows, the FC also introduces the concept of bandwidth shuffling. If $F \approx 0$, henceforth the efficiency is nearly optimal, bandwidth shuffling allows for redistribution of bandwidth between streams. The redistribution will assure each flow its fair share, even in a system that has already achieved an uneven equilibrium. Up to 10% of the bandwidth is redistributed between the flows according to AIMD in a fully loaded system.

2.2.3 Packet departure

When the calculations have been done during the Control Interval Timeout, the amount of bandwidth to allow or disallow has been calculated. This change in bandwidth is distributed on the expected number of packets arriving in an interval. Based on these values each packet might need to adjust its advertised `delta_throughput`. Only a few calculations are required per packet on departure to check its `delta_throughput` towards the allowed throughput for that packet. If the packet's advertised `delta_throughput` is bigger than what the router allows, the `delta_throughput` will be adjusted accordingly. In the case where the advertised `delta_throughput` is less than what the router is willing to allow, the `delta_throughput` is left unchanged.

2.2.4 Queue estimation timeout

An important part of XCP's internal algorithms is the queue size in the router. The queue size tells the algorithms used during the Control Interval Timeout how congested the router is, and indirectly how the feedback should be allocated to each flow. The persistent queue size is estimated over a timed interval, somewhat shorter than the average RTT, in order to avoid a feed-forward loop. The timeout takes into account the size of the queue, reducing the timeout when the queue grows.

$$T = \max(Q, (a - i/C) / 2) \quad (4)$$

Where:

T Next timeout, measured in milliseconds.

Q Standing queue size one is willing to maintain (milliseconds).

a The estimated average RTT during previous control interval (milliseconds).

i Instantaneous queue length measured in bytes.

C The capacity of the outbound link (B/s)

If average RTT is used as the length of the Control Interval, it must not be used as the interval used to estimate the queue size, because if a queue develops, the average RTT will increase. This will make the system reacting slower to the growing queue and the queue gets even larger, i.e. leading to instability.

2.3 Receivers

When the receiving system gets an XCP-message, it only needs to copy back the value in the `delta_throughput` field back into the `reverse_feedback` field. By setting the format to 0x02 the receiver prevents the XCP routers from processing the XCP packet on its way back to the sender. The receiver should send the XCP message out with the next TCP-ACK message. However, TCP implementations often do not send and TCP-ACK message for each received message. To be able to handle this situation, the receiver must accumulate all the `delta_throughput` it has received since the last TCP-ACK message was sent, in order for the XCP sender to get the correct feedback. The sender will in this case receive fewer XCP messages, but each will contain the aggregated feedback values of multiple XCP messages sent.

2.4 Summary

The XCP protocol introduces an entirely new way of dealing with network problems such as congestion control and fairness. By introducing a new protocol header, the XCP protocol is capable of receiving and reacting quickly to direct feedback from the network it operates on. XCP promises fairness between flows, virtually no packet loss while at the same time maximizing the bandwidth utilization [1]. The XCP routers makes sure all flows get their fair share of the bandwidth, while at the same time preventing senders from overloading the network (see Figure 4 below). Its critics will claim that introducing a layer between IP and TCP in the protocol stack is a violation of the established TCP/IP protocol hierarchy. Katabi challenges the established view of protocol layering by letting routers read and edit fields above the network (IP) layer. However, it is very unlikely that XCP would ever become a standards-track RFC as XCP removes itself entirely from the end-to-end solution to congestion control as used by TCP. This explains why XPC aims at becoming an experimental RFC and not a standard RFC.

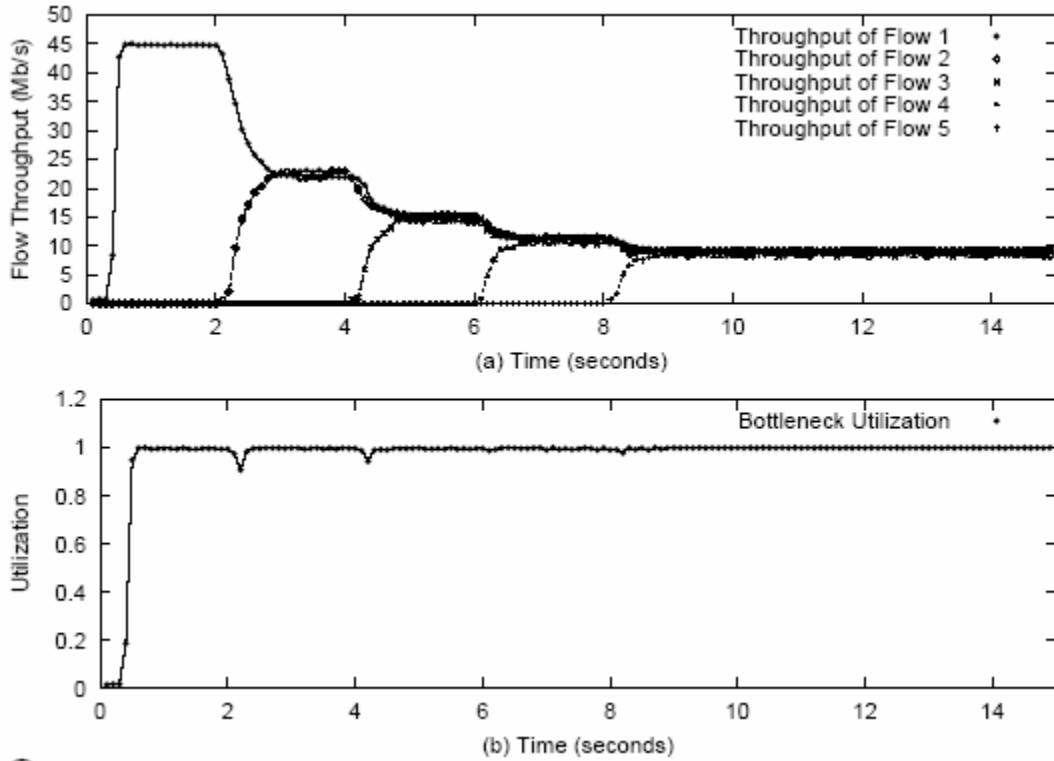


Figure 4: XCP simulated performance by Katabi [1]

Figure 4 is taken from [1] and shows the simulated performance of XCP gotten by Katabi. Figure 4.a shows how introduction of new XCP flows into a fully utilized network changes the equilibrium, while at the same time maintaining a fair bandwidth allocation between each flow. Each flows bandwidth usage is kept constant and the introduction of multiple competing flows does not lead to any oscillations at any point. Figure 4.b shows how the bandwidth utilization is kept maximized throughout the process. The introduction of more flows does not affect the total throughput, as the XCP routers make sure to distribute the available bandwidth fairly amongst the flows.

The XCP protocol, being fundamentally different from the current TCP/IP protocol implementation, could pose a real challenge to implement. It was therefore necessary to see if we could implement the XCP protocol in the Linux kernel, in order for us to be able to verify the simulation results. In Section 3 we investigate further the issues and challenges we faced when trying to create a real, working XCP protocol under Linux.

3 Implementing XCP in Linux

As part of our investigation of the XCP protocol, we wanted to create a real working Linux implementation of the XCP protocol that could be tested out in a real network. Previously the University of Southern California's Information Sciences Institute (ISI) has created a FreeBSD implementation [3], which has shown results supporting the simulations done of XCP [1]. We wanted to see if we could replicate the results gotten by ISI and by XCP authors' simulations by following the XCP specification directly [2], i.e. without looking at ISI's FreeBSD implementation. Another group lead by Zhang and Henderson [4] was in the progress of creating a Linux implementation when we first started out on our Linux implementation. Their implementation approach to the XCP protocol was different than ours and our implementation and tests have been conducted independently of Zhang and Henderson. In this chapter we describe our implementation and the choices and tradeoffs we took.

We wanted to implement XCP as a real protocol layer between IP and TCP as proposed by the XCP specifications, and not as an extension to either the TCP or the IP protocol. According to the XCP specification [2], XCP should add its protocol header between the IP and TCP headers, and therefore we felt that creating XCP as a new protocol layer between TCP and IP was the most "correct" implementation option (see Figure 5). By creating XCP as a separate protocol layer in the TCP/IP stack we could avoid changing the existing IP or TCP implementation in the Linux kernel. It would also serve as a test to see if such a fundamentally different approach to congestion control could be implemented in the Linux kernel without doing serious rewriting of the entire TCP/IP stack.

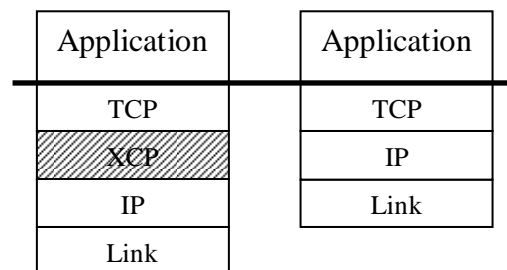


Figure 5: TCP/XCP/IP vs. TCP/IP Stack

XCP could have been implemented as a TCP option as well. This approach was followed by Zhang and Henderson [4] in their Linux implementation. The advantage of such an approach was primarily that it would give XCP intimate control of TCP. As XCP tries to change the whole congestion control scheme of TCP, knowledge of TCP's inner workings could be very important for XCP. However, by implementing XCP as a TCP extension, the XCP header would need to be converted to a TCP option. As we wanted to follow the XCP specification as closely as possible, this approach was dropped in favor for having XCP as a separate protocol.

Implementing XCP at the IP layer would also be possible. This would lead to having the XCP header implemented as an IP option instead. From the XCP routers point of view, this is equally elegant as having XCP as a separate protocol layer. However, an implementation at the IP layer would not give XCP any more control over the inner

workings of TCP, than implementing XCP as a layer in between XCP and IP. This approach would have solved problems with hardware checksums that we came to experience in our implementation (see Section 3.1 below). Since there were no other real advantages to this approach, it was dropped in favor of the separate layer option.

The XCP implementation was done on machines running Linux Fedora Core 3 [14] distributions, running the 2.6.9 version of the Linux kernel [15]. The XCP protocol was implemented as a Linux Kernel Module, so it could be loaded and unloaded from the kernel manually, without the need for rebooting. This is a quite common technique when implementing optional protocols under Linux, and has multiple benefits from a development and usage point of view. The primary development advantage is that the module can be compiled without the need for compiling the entire Linux kernel. The fact that the kernel will boot without the module loaded speeds up development, as a bug in the Linux kernel and its modules, often will bring the entire system down. When loaded the XCP kernel module will override all TCP traffic, appending the XCP header before the IP layer gets hold of the packet.

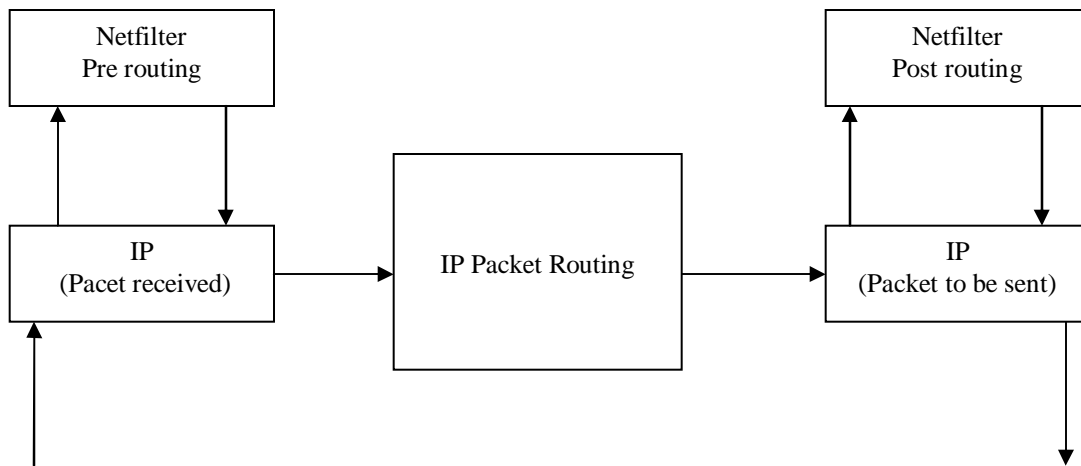


Figure 6: Netfilter facility in Linux kernel

XCP requires routers to read the contents of the XCP header contained in the IP packet. However, the default IP routing behavior in Linux only routes packets based on the IP source and destination address. Unless IP options are detected, the router will not inspect the packet any further. Implementing XCP requires changes to the role of the router as this is one of the structural differences between XCP and TCP/IP. Instead of just routing IP packets, routers under XCP need to inspect every packet and adjust the XCP packets' contents along the way. The Linux kernel already contains a feature called "netfilter" (see Figure 6) that allows a router to do custom processing of packets as they are processed by the IP layer. This feature is primarily used for creating different types of firewalls, and it is invoked by the kernel when IP packets are processed. By creating a special "netfilter-hook", packets containing the XCP protocol number, set to 200 for test purposes, was caught by the XCP "netfilter" and processed accordingly. The XCP router was also

created as a Linux Kernel Module that when loaded would process XCP packets passing the router. When unloaded the XCP router would run as a normal IP router, just routing XCP packets based on their IP header, without doing any additional calculations on them.

3.1 XCP protocol layering issues

Implementing the XCP protocol as a separate protocol between TCP and IP brought along quite a few issues, which were caused by the fact that the XCP protocol violates the layered protocol structure in the TCP/IP stack. Most notably XCP tries to adjust the way TCP works by directly manipulating TCP's congestion window parameter (cwnd). By manipulating the congestion window, XCP can make the TCP protocol allow more or less packets to be unacknowledged in the network, thereby indirectly increasing or reducing TCP's transmission speed. Our implementation adjusts the congestion window of TCP directly when packets are received or sent through the XCP layer, overriding any adjustments done by TCP to the congestion window.

However, the congestion window is only one of the parameters that TCP use to control its transmission speed, the "advertised window size" of the peer is equally important. TCP will not under any circumstances allow more packets to be sent than what the recipient advertises its window size to be, and it is up to the receiver to increase or decrease this value as it sees fits, regardless of what XCP might want. This functionality is one of TCP's main features, and it is used to prevent a slow receiver from becoming overloaded by a fast sender. So XCP has, by being a separate protocol to TCP, not total control over TCP's sending speed. XCP cannot override the way in which TCP handles its advertised window, as this would invalidate the TCP protocol entirely. One could have changed the way TCP handles its advertised window size, but this would be counter productive to our goal of trying to implement the XCP protocol as a separate layer and not changing any existing TCP/IP code.

One of the XCP protocol's main features is its ability to rapidly increase sending speed in environments with a high *bandwidth-delay* product, especially outperforming TCP's slow start algorithm that uses considerable time to increase TCP's performance in such an environment. However, under Linux the peer's advertised window is increasing at maximum two packets per ACK, preventing XCP from improving TCP's performance during startup. Even though XCP might change TCP congestion window parameter, allowing large amounts of unacknowledged data to be sent, it is ultimately TCP's decision how much to send, furthermore as long as XCP does not have ultimate control over TCP's transmission speed and how it chooses to increase its advertised window, XCP will not under any circumstances be able to outperform TCP during startup.

Another layering issue that arises with XCP is its need to know about TCP's flow concept. The bandwidth usage of one TCP flow can be notably different from another flow. So the XCP protocol needs to know to which flow a TCP packet belongs to before setting information such as RTT and current throughput. This knowledge is needed on both the sending and receiving side of the XCP protocol. The receiver is responsible for

accumulating feedback and returning it with the next TCP- ACK message in a specific flow. If multiple XCP flows are handled by the receiver, the receiver will need to be able to separate these flows from each other, in order to let feedback be returned with the correct flow. Again this is a violation of the protocol hierarchy. As the flow concept is an internal concept used by TCP it should not be necessary for XCP to know about it in order to work as intended. Our implementation solves this issue by letting the XCP layer read out the port numbers from the TCP header, and the IP addresses from the IP layer, and uses this information to decide which flow a packet belongs.

Section 3.1.3.2 “Response to Packet Loss” in the XCP specification [2] contains thoughts about how XCP should handle packet loss. Implementing the recommendations would further seriously violate the protocol layering between XCP and TCP. XCP would in addition to the information listed above also need to know about dropped packets and any TCP options used by TCP. We opted not to implement any of these features, so our XCP specification does not adjust TCP’s congestion window if a packet drop occurs. In an “all-XCP” environment it is the responsibility of the XCP routers to prevent packet drops due to network overload. If a packet was dropped due to corruption or other circumstances, TCP would need to retransmit the packet, but the XCP protocol would still maintain TCP’s congestion window, thereby maintaining a high transfer speed.

We also encountered problems with the hardware checksum feature in the Linux kernel. Some network interface cards create TCP and IP checksums in hardware, in order to speed up the creation of data packets. When placing the XCP header between the IP and TCP header, the hardware checksum fails. The most likely reason for this failure is that the hardware checksum feature fails to detect the presence of a new header between IP and TCP. As the TCP checksum includes the IP source and destination address, reading out the wrong header data when trying to fetch the IP source and destination address, will cause the checksum to become invalid.

3.2 XCP specification issues

We based our implementation on the latest specification of XCP publicly available [2]. After thorough reading of the specification, some flaws appeared. They were minor in the sense that it was easy to see that there was something wrong. However, they were serious in the sense that the XCP protocol would not work at all using a straight forward implementation following the specification. The XCP protocol measures throughput using two different scales, and these seemed to be confused by the authors of the specification. The `delta_throughput` field in the XCP header is measured in bytes/second, while the `current_throughput` field is measured in bytes/millisecond.

The first error in the specification can be found in the formula that should be used to calculate the new congestion window (`cwnd`) at the sender, when feedback is returned from the receiver. The original specification [2] states the following in section 3.1.3:

$$cwnd = \max (cwnd + feedback * RTT * 1000, MSS) \quad (5)$$

Where:

cwnd Current congestion window (bytes)

feedback reverse_feedback field from received packet, (B/s, may be +/-)

RTT Sender's current round-trip time estimate (ms)

MSS maximum segment size (bytes)

The value of *cwnd* has a minimum of *MSS* to avoid the "Silly Window Syndrome"

To see why this formula (5) is false, one only needs to fill in some data, as is done in Example 7:

$$cwnd = \max (3.000 B + 1.000.000 B/s * 100 ms * 1.000, 1.500B)$$

$$cwnd = \max (3.000 B + 100.000.000.000B, 1.500 B)$$

$$cwnd = \underline{100.000.003.000 B}$$

Example 7: Invalid feedback formula

What Example 7 shows is that in order to allow an increase in bandwidth of 1.000.000 B/s on a 100 ms delay link, you would supposedly need around 100 GB of memory, which clearly is wrong. The error is to multiply the RTT which is in milliseconds by 1000 to get seconds, while it should have been divided by 1000.

The correct formula is given by:

$$cwnd = \max (cwnd + feedback * RTT / 1000, MSS) \quad (6)$$

To see why this formula is correct, one only needs to fill in some data. Using the same values as in Example 7:

$$cwnd = \max (3.000 B + 1.000.000 B/s * 100 ms / 1.000, 1.500 B)$$

$$cwnd = \max (3000 B + 100.000 B, 1.500 B)$$

$$cwnd = \underline{103.000 B}$$

Example 8: Correct feedback formula

Example 8 shows that a buffer of 103.000 bytes is needed to be able to allow a bandwidth of 1.000.000 B/s on a link with 100ms delay. The default TCP memory setting in Linux provides TCP with approximately 100 kilobytes of buffer space. This allows a bandwidth of approximately 1 Megabyte/second (MB/s) on a 100 ms delay link, or 10 MB/s on a 10 ms delay link. A small LAN with delays around 1 ms could run as fast as 100 MB/s using the default memory settings. However, the default settings are inadequate in a high speed, high delay network.

Another similar error in the specification is found in the crucial feedback formula given in a XCP router as given in section 3.2.2 in the XCP specification [2]. The original pseudo code for calculating the routers feedback is:

$$F = a * (capacity - input_bw) - b * queue / avg_RTT$$

The original explanation from the XCP specification states:

*The aggregate feedback, F, is calculated. The variable `capacity' is the ability of the outbound link to carry IP packets, in bytes/second. The variable avg_RTT was calculated in line 7. The variable queue is the persistent queue and is defined in section Section 3.2.5. The values a and b are constant parameters. The constant a may be any positive number such that $a < (\pi/4 * \sqrt{2})$. A nominal value of 0.4 is recommended. The constant b is defined to be $b = a^2 * \sqrt{2}$. (If the nominal value of a is used, the value for b would be 0.226.) Note that F may be positive or negative.*

As stated ‘F’ (feedback) should be measured in bytes/second, as well as the capacity of the link. With ‘a’ and ‘b’ being constants, ‘input_bw’ needs to be measured in B/s in order for this formula to work at all. However, that is not the case: ‘input_bw’ is measured in bytes/milliseconds (B/ms). As well as mixing different measurements of scale, this is clearly wrong. Given a fully loaded system with a capacity of 1.000.000 bytes/second and an input bandwidth of 1.000 B/ms, feedback should be 0 (given no queue). Using the original formula F would be $0.4 * (1.000.000 - 1000)$ or approximately 400.000 B/s, basically the same as an unloaded system.

Therefore the correct formula is to have ‘capacity’ in bytes/milliseconds and ‘F’ given as:

$$F = 1000 * (a * (capacity - input_bw) - b * queue / avg_RTT) \quad (7)$$

The feedback calculation is now in B/ms, and need to be scaled by 1000 in order to be converted to B/s as required by the rest of the calculations during the control interval.

3.3 Implementing the host side of the XCP protocol

Most of XCP’s logic is based on parameters set by the sending side of a TCP/XCP flow. It is while sending data to the peer that the XCP header is filled with information about the current throughput, delta-throughput and current RTT values. On the receiving end, the delta_throughput is just returned back to the sender in the reverse_feedback field.

As TCP sends a packet, the XCP protocol layer adds the 20 byte XCP header before the IP layer adds its IP headers to the packet. Before the XCP header is passed on to the IP

layer, the XCP layer fills in the RTT field of the header by using TCP's smoothed RTT value. This value is readily available from the TCP layer and can be used as is. In the Linux kernel, the smoothed RTT value is expanded (multiplied) by 8 to allow TCP to do adjustments of the RTT values with integer arithmetic, so the RTT value is divided by 8 by the XCP layer before it is filled in into the header.

A major part of the XCP protocol is to calculate the sender's current throughput, and use this calculation to set how much more (or less) bandwidth one requires. This change is then divided amongst all packets leaving the sender. We used the recommended approximation from section 3.1.1 in the XCP specification [2]:

$$\text{throughput} = \text{cwnd} / \text{RTT} \quad (8)$$

Where:

throughput Current throughput measured in bytes/ms
cwnd TCP's current congestion window, measured in bytes.
RTT TCP's currently smoothed RTT value measured in milliseconds..

However, using formula (8) as an estimation of the current throughput prevents the implementation of section 3.1.3.1, "Aging of Allowed Throughput" in the specification [2]. This section is introduced to prevent the XCP host from being able to send large bursts of data into the network. The section specifies that each RTT in which the sender sends with actual throughput which is less than the allowed throughput, the allowed throughput must be reduced by the following exponential averaging formula:

$$Alw = Alw * (1 - p) + Act * p \quad (9)$$

Where:

Alw Allowed_Throughput. The allowed throughput for the sender to use
Act Actual_Throughput. The actual or current throughput
p is a parameter controlling the speed of aging, ranged between 0 and 1. A nominal value of 0.5 is recommended.

The reason why it cannot be calculated is that we are using *cwnd / RTT* as both the actual and allowed throughput. Since the XCP protocol changes the *cwnd* parameter in order to set how much bandwidth that is allowed, it cannot at the same time use this value to know how much data is actually sent. The congestion window indirectly specifies a maximum transfer value for TCP; it does not give XCP any ideas of how much data is really sent.

This formula is introduced into the specification to prevent sudden high bursts of data into the network allowed by high a *cwnd* value. The XCP protocol is developed for usage on networks with high *bandwidth-delay* products, in such networks the congestion window will need to be very high in order to allow many packets unacknowledged in the

network. However, section 3.1.1 in the specification might seem a bit hastily introduced, as it is incompatible with the $cwnd / RTT$ approach to throughput calculation, and neither `Allowed_Throughput` nor `Actual_Throughput` are variables that are mentioned any other places in the specification. The effect of not being able to implement this section allows senders that might have been idle for some time to overload the network with data, which in turn could lead to queue build-up and possibly packet loss in the network.

In addition to the current throughput estimate, XCP uses the field `delta_throughput` to request change in bandwidth. This field is in bytes/second, and thereby allows changes in bandwidth, in theory, to be quite small (minimum 1 B/s). However, the Linux implementation of TCP uses a congestion window measured in packets, not bytes. Each packet is up to MSS (maximum segment size) bytes long. On an Ethernet, the maximum segment size is usually 1500 bytes. The granularity of bandwidth change is therefore much higher than 1 byte/second. An increase of the congestion window by one packet would on a network with a delay (RTT) of 100 ms, increase the bandwidth with 15.000 bytes/second. Precisely tuned adjustments of the bandwidth using the XCP protocol are therefore not possible as long as the sending speed is dependent on the TCP implementation. The discussion in the specification under section 3.1.1 “Sending Packet” where the question of what to do if the delta throughput is calculated to be less than 1 byte per second can seem a little bit “theoretical” given the lack of granularity in TCP.

3.4 Implementing the XCP router

Another major part of the XCP protocol is the XCP capable router. A computer running Linux can function as a router by enabling IP packet forwarding. Most routers only route IP packets based on the IP header, and will not check the contents of an IP packet at all. This code is highly optimized so that routing of multiple IP packets with the same source and destination address can be handled very effectively. This is the so-called fast path in the router. XCP on the other hand need the router to read the XCP header in order for it to manipulate the XCP header. The argument Katabi [1] uses for not implementing XCP as an IP option is that IP packets with options would need to follow the “slow path” in non-XCP routers. The “slow path” is more computing intensive as the router needs to inspect the full packet header with all options. However, if XCP is implemented as a separate protocol, TCP routers would not process the XCP header at all, and since there are no IP-options, this would be very effective. On the other hand XCP is only “guaranteed” to work when all routers in a flow’s path support the XCP protocol. In a scenario where all the routers supports the XCP protocol, they must all read and parse the XCP header and do the required per packet processing. The calculation required by having XCP as a separate protocol will not be any faster than processing XCP data as IP options in this scenario. From the XCP-router’s point of view XCP could just as easily be implemented as an IP option instead of being a separate protocol.

The XCP router consist of 4 main parts, each with it clearly defined responsibility.

- 1) *Calculations on packet arrival.*

- 2) A control interval timer that does calculations based on data received during the previous RTT milliseconds.
- 3) Calculations on packet departure.
- 4) A queue estimation timer that estimates the persistent queue in the router during the previous RTT/2 milliseconds.

Each of these parts is described in more detail the following sections.

3.4.1 Calculations on packet arrival

By using a specific protocol number for XCP (200) in the IP packet we can sort out the XCP packets in the router and check the XCP header and extract the information required. For each packet arriving at the XCP router, some calculations (see Code 1) are done and these calculations are used as a basis for calculating the feedback for the next control interval. The calculations we do are based on the XCP protocol specification [2] section 3.2.1, given here in pseudo code.

```

input_traffic += Pkt_size
sum_inv_throughput += Pkt_size / Throughput
if (Rtt < MAX_INTERVAL) then
    sum_RTT_by_throughput += Rtt x Pkt_size / Throughput
else
    sum_RTT_by_throughput += MAX_INTERVAL x Pkt_size / Throughput

```

Code 1: Input traffic calculations

What is clear from the pseudo code is that without using floating points the code can lead to serious miscalculations, as shown in Example 9. Because integer arithmetic always has rounding errors, the pseudo code would magnify these errors if not extra care is taken. Unfortunately the Linux kernel does not allow any floating point calculations, only 32 bit integer arithmetic. In order for a real XCP implementation to work in the Linux kernel, these calculations need to be factored up, in order for integer arithmetic to be more precise.

*If Pkt_size is 1500 bytes and Throughput is 2500 bytes/ms, and there are 1000 packets during an RTT – interval. This would make the sum_inv_throughput = 0 (1000 * (1500/2500)), while the correct value is 600.*

Example 9: Integer arithmetic rounding errors

To gain maximum precision from integer arithmetic, divisions should be done with as large values as possible. However, the amount a variable can be scaled up to is dependent on the initial size and range of values that variable can have. If a variable's minimum value is 0 and maximum value is 32768 (13 bit), it can be scaled up by a factor of 131072 (15 bit) without passing the 32 bit integer limit. In order to allow the maximum amount of precision the scaling factor should be dynamically decided based on knowledge of the network where the XCP protocol is implemented. Quite elaborate schemes can be

invented to adjust and maximize the precision of these calculations. However, in our implementation, we choose not to use any such schemes, but resorted to a simple hard coded scaling factor that we knew would work under most normal loads and test purposes. A hard coded scaling factor is easy to implement, and does not lead to any big computing overhead, as no calculations are needed to adjust to different network scenarios. In a production system, it would be natural put more effort into this aspect of the implementation. Example 10 shows how increasing the scaling of the calculations done in Example 9, improves the accuracy enormously.

*Assume a scaling the Pkt_size by **1024**. Given an original Pkt_size of 1500 bytes and Throughput is 2500 bytes/ms, and 1000 packets during an RTT – interval. This would make the sum_inv_throughput = 599 ($1000 * ((1024 * 1500)/2500) / 1024$), much closer to the correct value of 600.*

Example 10: Effect of scaling

3.4.2 Control Interval Calculations

For each “average-RTT” milliseconds an XCP router needs to make a new average load calculation. This calculation is done based on all the data collected from packets arriving during the last RTT milliseconds. The XCP specification assumes that even though these calculations are somewhat complicated, the fact that they are only done once each RTT millisecond reduces their performance implication.

Pseudo code giving the calculations to be done during the “control interval” timeout:

```

avg_rtt = sum_rtt_by_throughput / sum_inv_throughput
input_bw = input_traffic / ctl_interval
F = 1000 ( a * (capacity - input_bw) - b * queue / avg_rtt)
shuffled_traffic = shuffle_function(...)
residue_pos_fbk = shuffled_traffic + max(F,0)
residue_neg_fbk = shuffled_traffic + max(-F,0)
Cp = residue_pos_fbk / sum_inv_throughput
Cn = residue_neg_fbk / input_traffic
input_traffic = 0
sum_inv_throughput = 0
sum_rtt_by_throughput = 0
ctl_interval = max(avg_rtt, MIN_INTERVAL)
timer.reschedule(ctl_interval)

```

Code 2: Control Interval Timeout calculations

Our implementation uses a timer that times out each average RTT, as this is the method depicted in the XCP specification. A drawback of using timers is that one needs to handle the exception when no traffic is passing the router. In this case a lot of the variables, like

input_traffic, will be 0 and lead to division by 0 if care is not taken by the feedback calculation routine. Another issue is that timers are not 100% accurate, as one is only guaranteed that the timer will timeout after the amount of milliseconds set, but not exactly after the amount of milliseconds set.

Another way of implementing a “control interval” is to check the amount of time passed since last “control interval” when a packet is received. If more than average RTT milliseconds have passed, one does the “control interval” calculation. The advantage of this method is that one does not need to use timers in the kernel, and one does not need to handle the exception where no data is received during an RTT-interval. The disadvantage is one can get somewhat skewed intervals if no packets arrive at the time of the timeout. This approach was used by Zhang Y., Henderson T [4] in their Linux implementation.

Regardless of the timeout implementation chosen, these calculations need to be done on per out-device basis, and not only on per “average-RTT” basis. In addition, all variables used by XCP needs to be kept on a per out-devices basis as well. The reason for this is that the calculations contain references to the queue length in the router. A router serving many different flows usually have multiple input and output interfaces which in turn can have very different load, and queues, on them. It is therefore vital for the XCP router to know which flows are running on a given interface, so that it can reduce the load for those flows only, and not all flows passing through the router. This per interface calculation linearly increases the processing power needed to run the XCP protocol with the number of interfaces in routers.

3.4.3 Calculations on packet departure

For each packet that leaves the router (unless it has got a minimal header), the delta_throughput field of the XCP header is read and maybe modified. The following calculations are done per packet, given this pseudocode:

```
pos_fbk = Cp * Pkt_size / Throughput
neg_fbk = Cn * Pkt_size
feedback = pos_fbk - neg_fbk
if(delta_throughput > feedback) then
    delta_throughput = feedback
else
    neg_fbk = min(residue_neg_fbk, neg_fbk + (feedback - delta_throughput))
    pos_fbk = delta_throughput + neg_fbk
residue_pos_fbk = max(0, residue_pos_fbk - pos_fbk)
residue_neg_fbk = max(0, residue_neg_fbk - neg_fbk)
if (residue_pos_fbk <= 0) then Cp = 0
if (residue_neg_fbk <= 0) then Cn = 0
```

Code 3: Calculations done on packet departure

This code will distribute the allowed feedback calculated during the control interval from the router amongst on all packets leaving the router during the next RTT milliseconds.

The code in line 20 and 21 enforces how XCP distributes changes in bandwidth between different flows (The Fairness Controller, Section 2.2.2.2).

Due to the use of integer arithmetic, these calculations need to be scaled up in order for the XCP protocol to work correctly. Each packet in an individual flow can receive quite a small change in the feedback, since the total feedback is distributed amongst all packets in that flow. So rounding errors due to integer arithmetic will again play an important role and need to be reduced by scaling up the computations before the result is given to each individual packet.

3.5 Summary

Implementing XCP in the Linux kernel was complicated by the placement of the XCP header, and by our desire to implement the protocol without changing the existing TCP/IP code. In addition, the fact that XCP is dependent on internal knowledge of the TCP protocol further complicated the implementation. The Linux kernel has been designed to allow for more protocols to be added as modules. However, based on the existing kernel TCP/IP code, it does not seem like it has been anticipated that an entirely new protocol layer should be situated between TCP and IP. We managed, by using some creative hacking, to allow the XCP protocol to become seamlessly integrated into the Linux network protocol stack. However, because of XCP's reliance on TCP we feel like this was probably not the right way to go for future implementations. As XCP tries to improve the performance of TCP, it seems more natural to implement it as a TCP option rather than as a separate protocol.

By using the “netfilter” facility to implement the XCP router, we greatly simplified the XCP router development. In addition, this allowed the processing of XCP header information to be done at the correct level in the protocol hierarchy. The extra overhead introduced by this solution was minimal, as the Linux kernel netfilter code is already highly optimized to work with firewalls, which also does processing based on the contents of IP packets.

The lack of floating point arithmetic in the Linux kernel complicated the implementation of the XCP protocol as well. In addition, the inability to do fine grained adjustments to the congestion window clearly showed that XCP has been developed without much thought on limitations that might be inherent in real computer kernels.

We followed the XCP specification [2] that recommended using timers to control the different control timeout intervals. In hindsight, this could have been implemented more elegantly without timers. Each packet passing through the XCP layer could just as easily have checked the time elapsed since last control interval, and if enough time had passed, done the required calculations.

4 XCP Performance results

In addition to implementing the XCP protocol in Linux, we also wanted to test the protocol's performance. In this section we describe the tests and results we got from our Linux implementation, compared to the original simulation results [1] and results gotten by Zangh and Handerson [4]. XCP promises to improve TCP's performance especially in networks with a high *bandwidth-delay* product. By introducing a new header and by receiving explicit signals from the network, the XCP protocol tries to prevent packet loss and oscillatory behavior. To be able to test the XCP protocol, the following test setup was created using real computers and network interface cards, running on a 10/100 Mbit Ethernet:

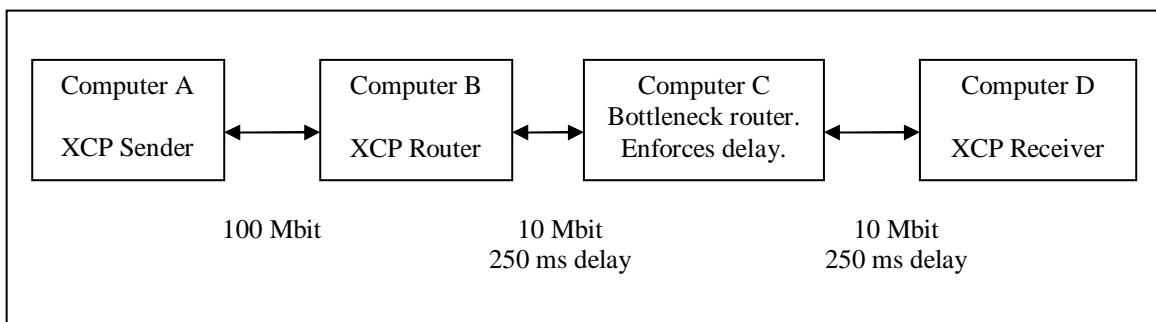


Figure 7: Test setup

Unless otherwise specified, the tests were performed by creating one or more TCP/XCP streams from computer A to computer D. Computer C used the standard built-in network emulator 'netem' [17] in the Linux kernel to create a total of 500 millisecond delay in the path from B to D and back. The 'netem' program delays packets for 250 milliseconds when the packets are leaving each of the Ethernet interfaces. In addition, Computer C was set up using two 10Mbit full duplex network cards to allow for a queue buildup in the XCP router (Computer B). The rest of the network used 100Mbit full duplex Ethernet network cards. The XCP router (Computer B) used a queue size of twice the *bandwidth-delay* product (833 packets), a common assumption for Internet networking.

TCP's sending speed is indirectly controlled by the amount of unacknowledged packets in the network. The more packets TCP allows to be unacknowledged, the faster the transfer-speed. As formula (8) states, the higher the RTT, the higher the number of unacknowledged bytes is needed to be in transit in a network, in order to achieve the same throughput. Three factors determine the amount of bytes that can be in transit for a TCP flow. First, the buffer-space available for the TCP protocol can limit the sending speed. In order for TCP to be able to resend packets, in case of packet drops or other network problems, TCP needs to buffer all packets sent that have yet to be acknowledged. In other words, TCP cannot have more packets in transit, than it has buffer space. Another fundamental feature of the TCP protocol is the sliding window algorithm as part of TCP's credit-based flow control. In order for a fast sender not to overload a slow receiver, the TCP protocol continuously reports the number of bytes that the peer is willing to receive. This variable is known as the advertised window, and TCP

will not under any circumstances allow more packets to be unacknowledged than what the peer’s advertised window has given permission to. Finally, in order to adjust to congestion problems in the network, TCP uses a variable, called the congestion window (cwnd) that also sets a maximum limit to the number of packets that can be in transit. Different TCP variations and implementations use different values and algorithms for changing the advertised window and congestion window in order to optimize TCP’s performance.

One of the first problems we needed to handle, before we could make any performance test, was to increase the buffer space available for TCP. The default TCP parameters in the Linux kernel are tuned towards LANs with very low RTT delays. However, our test-environment (Figure 7) was running with high delay and bandwidth. We followed the TCP tuning guide [16], and increased the kernel network buffers associated with each TCP flow. We increased TCP’s maximum buffer size to 16MB, and increased the Linux auto-tuning TCP buffer limits to (minimum, default, maximum): 4.096 B, 87.380 B and 16.777.216 B. With the increased buffer sizes, TCP was now capable of allowing enough packets to be in transit to fill the network pipe.

4.1 Overestimating the congestion window

The original XCP specification [2] does not address the problem of invalid adjustments of the congestion window (cwnd) parameter. As mentioned in Section 3.1 the advertised window cannot be adjusted by the XCP protocol, which leads to problems in the XCP protocol if this is not accounted for. The Figure 8 shows the congestion and advertised window (in packets) during a single XCP transfer as measured at Computer A (Figure 7).

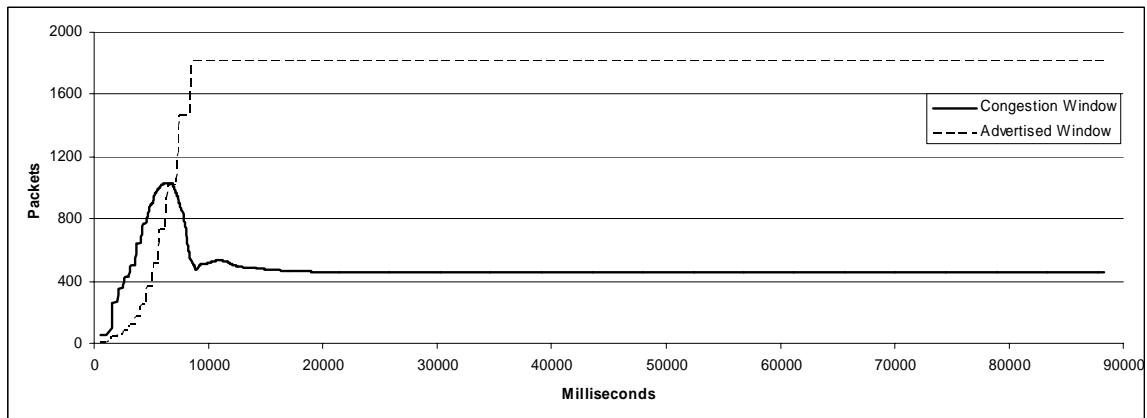


Figure 8: XCP Congestion Window vs. TCP Advertised Window

If the XCP client only adjusts the congestion window without taking into account the receiver’s advertised window, the congestion window, dictated by XCP, will easily outgrow the advertised window. The feedback from the XCP router is based on available bandwidth, and as long as this is below what the router can handle, it will return positive

feedback. The advertised window will grow based on the TCP implementation in Linux, and is outside XCP's control. Since the XCP protocol uses a "multiplicative increase", it will try to increase A's bandwidth usage by about 40% of the available bandwidth per RTT. However, the actual sending speed of TCP is limited by the advertised window and not by the congestion window. This leads the XCP router to receive less data than the XCP client thinks it is sending, and will in turn make the XCP router ask the XCP client to increase its sending speed even more. The only thing the XCP client can do is to further increase its congestion window, assuming this will lead to increased output. However, this only leads to inflated congestion window values, which are not used before the advertised window catches up with the congestion window. When this happens the router will become overloaded and the queue builds up in the router. Figure 8 clearly shows that around 5000 milliseconds, the advertised window catches up with the congestion window. The router is then overloaded with data from the XCP host, and quickly downgrades the XCP hosts sending speed. This effect happens because we are using formula (8) for throughput estimation and not adjusting for actual throughput (as specified in section 3.1.3.1 in the original XCP specification). The following example gives a more thorough walkthrough of the problem:

We have a XCP network containing 3 machines: A (sender), B (router) and D (receiver). A and B are connected with an 80Mbit network card (10.000 B/ms) and a RTT of 1 ms. B and D are connected with an 8 Mbit network card (1.000 B/ms) and a RTT of 499 ms. The total A to D RTT is therefore 500ms.

Assume A is transmitting to D using one TCP/XCP flow. For the simplicity, disregard XCP router shuffling and packet queuing. A's starting cwnd is 1. The MSS of the network is 1500 bytes and the congestion window (cwnd) is measured in packets.

Using these formulas:

current_throughput: cwnd / RTT

delta_throughput:

*(Capacity - (current_throughput * 1000)) / (current_throughput * (RTT/MSS))*

Feedback at sender:

*cwnd = max (cwnd + feedback * RTT / 1000, MSS)*

1. A starts by sending one packet to D during the current RTT interval (500ms):

A_{sender}:

current_throughput: 1 (b/ms)

*1 packet * (Packet_{delta} = 30.000.000 B/s)*

In other words, the sender sends one packet with the XCP delta_throughput field set to 30.000.000. This value is 3 times the capacity of the network as we only send 1/3 of a packet during RTT milliseconds.

2. B does calculations on the packet received and gets the following numbers:

B_{router} :

$$F = 400.000 \text{ B/s (approx)}$$

$$C_p = 266$$

$$1 \text{ packet} * (\text{Packet}_{\text{delta}} = 400.000 \text{ B/s})$$

When the router receives this packet, which is the only packet it receives in its 500 ms control interval, it can by using the pseudo code (4) calculate what delta to actually allow for this packet. This value, 400.000 B/s, is then set in the packet when it leaves the router.

3. A gets the packet back on return from D and recalculates its cwnd:

A_{sender} :

$$cwnd = cwnd + 1 * \text{Packet}_{\text{delta}} * RTT / (1000 * MSS)$$

$$cwnd = 1 + 1 * 400.000 * RTT / (1000 * MSS) = 1 + 133 = \underline{134}.$$

As the packet is returned from D back to A, A will read the reverse_feedback field. A only receives one packet (acknowledge) from D, with this field set to 400.000 B/s. The new congestion window at A will be 134 packets large, an increase in congestion window of 133 packets.

4. During the next 500 ms A now sends the 134 packets it has been allowed to send:

A_{sender} :

$$\text{current_throughput: } 400 \text{ (b/ms)}$$

$$134 \text{ packets} * (\text{Packet}_{\text{delta}} = 72.000 \text{ B/s})$$

Using the given formulas A will send 134 packets, each with the XCP header field, delta_throughput set to 72.000 B/s. (For a total of approximately 9.600.000 B/s, the currently available bandwidth at A).

5. B does calculations on the 134 packets received and gets the following numbers:

B_{router} :

$$F = 240.000 \text{ B/s (approx)}$$

$$C_p = 477$$

$$134 * (\text{Packet}_{\text{delta}} = 1.788 \text{ B/s})$$

B now receives the 134 packets, and calculates the feedback to allow per packet. Each packet gets its delta_throughput field reduced from 72.000 B/s to 1.788 B/s. (To allow an increase in bandwidth of 240.000 B/s in total).

6. A gets the 134 packets back on return from D and recalculates its cwnd:

A_{sender} :

$$cwnd = cwnd + 134 * \text{Packet}_{\text{delta}} * RTT / (1000 * MSS)$$

$$cwnd = 134 + 134 * (1.788) * RTT / (1000 * MSS) = 134 + 80 = \underline{214}.$$

As D returns the 134 packets back to A, A uses the value in the reverse_feedback fields to increase its congestion window. However, as the router now have increased load, the new increase is less than before. The increase in cwnd was reduced from 133 to 80 packets to fit the reduced bandwidth available in the router, giving a new congestion window for A of 214 packets.

However, assume instead that after the first packet was received from D by A (point 3), A does not send out 134 packets (like in point 4), but only 10 packets. (Due to lack of data, buffer space, or perhaps D's advertised window are only 10 packets big).

7. A now sends 10 packets instead of 134 during the 500ms interval:

A_{sender}:

current_throughput: 400 (b/ms)
 10 packets * (Packet_{delta} = 72.000 B/s)

Each of these 10 packets will contain the same delta_throughput as the packet sent under point 4, as the delta_throughput calculation is based solely on the congestion window of A, and the RTT of the flow.

8. B does calculations on the 10 packets received and gets the following numbers:

B_{router}:

$F = \text{approx. } 400.000 \text{ B/s}$
 $C_p = 10.666$
 $10 * (\text{Packet}_{\text{delta}} = 40.000 \text{ B/s})$

B only receives 10 packets during this interval, and considers itself very lightly loaded, which indeed is correct. Since it is under very light load, it will give each of the 10 packets a much higher feedback than under point 5 (40.000 B/s vs. 1.788 B/s).

9. A gets the 10 packets back on return from D and recalculates its cwnd:

A_{sender}:

$cwnd = cwnd + 10 * \text{Packet}_{\text{delta}} * RTT / (1000 * MSS)$
 $cwnd = 134 + 10 * 40.000 * RTT / (1000 * MSS) = 134 + 133 = \underline{267}$

D returns the 10 packets back to A, and A recalculates its congestion window. Notice how the increase in cwnd was not reduced but stayed constant at 133 packets as under point 3, thereby allowing A to send much more data than it should in the future. A's congestion window is not 267 packets, while the correct value should have been 214.

Example 11: Wrong feedback from XCP router.

The problem depicted above, with inflated cwnd values at the XCP host, will happen each time an XCP host sends less data than it is allowed by the XCP router. This scenario can occur in quite a lot of realistic situations, such as when a receiver is doing computing

intensive processing of data received, or when a sender is waiting for data to be sent. To remedy this problem, our Linux implementation sets the congestion window to minimum of the advertised window, and the congestion window allowed by XCP. This prevents the XCP protocol from wrongly calculating its current throughput when the throughput is limited by the advertised window, and not the congestion window.

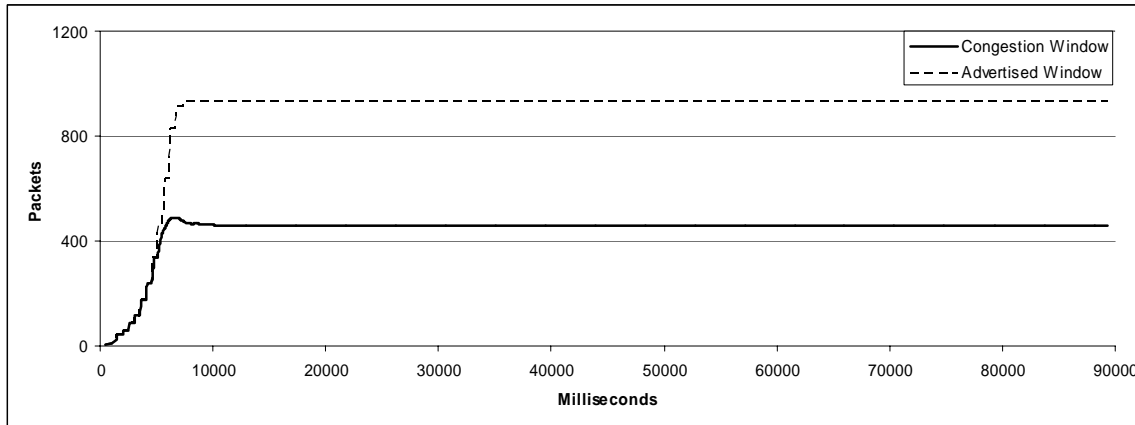


Figure 9: XCP Congestion Window limited by TCP Advertised Window

In contrast to Figure 8, Figure 9 shows that the congestion window is only slightly reduced when the sending speed reaches the capacity of the router (as can be seen from the curve at around 6000 milliseconds). In Figure 8 the congestion window is almost halved as the XCP router is overloaded with data due to incorrect feedback values given during the startup. As Example 11 shows, the XCP router is capable of giving incorrect feedback if the sender is under-utilizing its allowed bandwidth. This problem also arises if too little buffer space is available for the TCP to “fill the pipe”. When this happens the XCP router will always allow the sender to increase its throughput, thereby increasing the sender’s congestion window. However, this will not increase the sending speed, as it is the buffer space, and not the congestion window that is the limiting factor. The net result being that the cwnd-variable will grow indefinitely at the sender. It is therefore imperative that enough buffer space is available for TCP, in order to make XCP to work as intended.

4.2 XCP Router performance

According to the simulations presented in [1], one of the benefits of the XCP protocol was its ability to prevent queue build-up in the XCP routers, thereby preventing packet loss. This was perhaps the most important design rationale behind the XCP protocol, as packet loss on high speed, high bandwidth networks is very damaging to TCP’s performance.

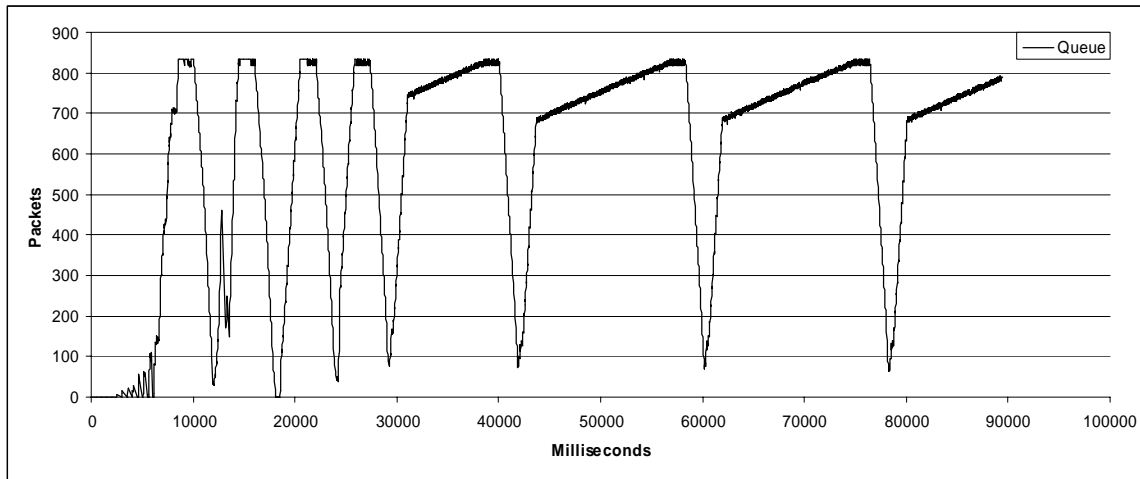


Figure 10: TCP router queue length

Figure 10 shows the queue size in the router for one TCP flow passing from computer A to D. As the queue size reaches the maximum of 833 packets, the router will start to drop packets. This will reduce the sending speed of computer A, leaving the router time to drain its packet queue. The TCP protocol will eventually adjust its congestion window to minimize packet drops. As the slow arithmetic increase in the congestion window finally causes the bandwidth to increase too much, packets will again be dropped. TCP will then again reduce its congestion window, reducing the queue size in the router, before slowly increasing it again.

In comparison, XCP manages to keep the routers queue as good as empty during the entire transfer of data (Figure 11). Compared to TCP (Figure 10) for the same flow, it is clear that XCP is superior to TCP when it comes to preventing queue build-up in the router. TCP encounters packet drops as the queue size tries to grow over the 833 packets limit in the router, XCP on the other hand does not encounter any packet drops, as the queue does not even pass 80 packets. This collaborates well with the simulations, and shows that the XCP router manages to reduce A's sending speed well in time to prevent the queue from becoming too large.

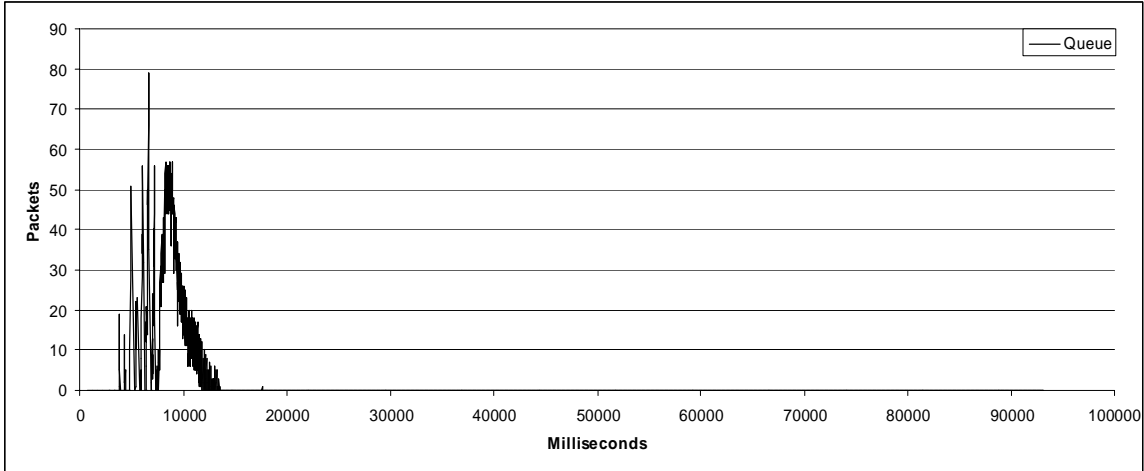


Figure 11: XCP router queue with one XCP flow passing. Router capacity set to 1180 B/ms.

However, as Figure 12 shows, XCP is dependent on the router being setup correctly in order to avoid queuing problems. The XCP router uses a predefined ‘capacity’ setting to set how much bandwidth is maximally available for it to use. If set too high, the XCP router will return inflated feedback values to the XCP host, driving up the queue size. Using the theoretical bandwidth as the ‘capacity’ of the XCP router leads to a small constant queue size. This happens because there are different kinds of low level headers that are appended to the IP level header, so the XCP router will need to take these into account as well before setting its ‘de facto’ capacity.

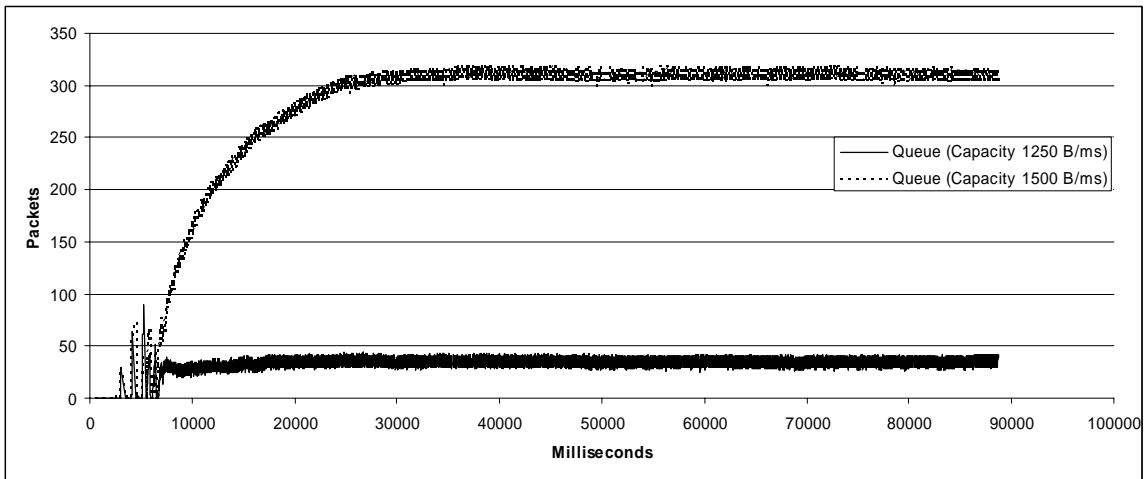


Figure 12: XCP router size with different ‘capacity’ setting

Figure 12 shows the effect of wrongly setting the XCP router’s ‘capacity’. The physical link has a capacity of 1250 bytes/ms (10Mbit/s). At this setting the router queue size does not drain. As Figure 12 shows, XCP depends on setting this value correct, as it clearly has an impact on the queue size in the router. As the queue grows, XCP’s feedback

formula will reduce the feedback to the XCP host, reaching an equilibrium dependent on the ‘capacity’ the XCP router assumes it has. However, finding the optimal value for XCP’s ‘capacity’ is not easy, as it can depend on what type of media and links XCP is running on. If the ‘capacity’ is set too low the XCP router will waste bandwidth which it could otherwise have distributed amongst the flows. On the other hand, if set too high the XCP router will not be able to remove the queue. These results are equivalent to the results reported by Zhang and Henderson [4].

4.3 Per flow performance

XCP also promises to share bandwidth between flows equally and to converge quickly. As Figure 13 shows, this is also the case. In this test, each new flow was started 30 seconds after the first one, and as we can see, the XCP router reallocates bandwidth to each new flow as they are created. This allocation is done quickly, and the flows converge nicely and all flows get an equal share of the bandwidth. As each flow stops sending, XCP redistributes the available bandwidth fairly between the remaining flows. Notice how the redistribution of bandwidth (at about 230,320 and 360 seconds) is quicker than the initial distribution between the flows (at about 30, 60 and 90 seconds). Each flow will at startup gradually increase its congestion window, like in Figure 9, taking longer time to reach equilibrium with existing flows. In addition, the XCP protocol uses its “shuffling function” to allow new flows to get a share of the bandwidth in a fully loaded system. The shuffling function allows up to 10% of total router bandwidth to be redistributed each RTT. When a flow is finished sending, the XCP protocol can react very quickly to redistribute the available bandwidth amongst the remaining flows, as it is the congestion window of each flow that is the only factor regulating the speed of the transfer. This result is in accordance with the simulation results gotten by Katabi [1].

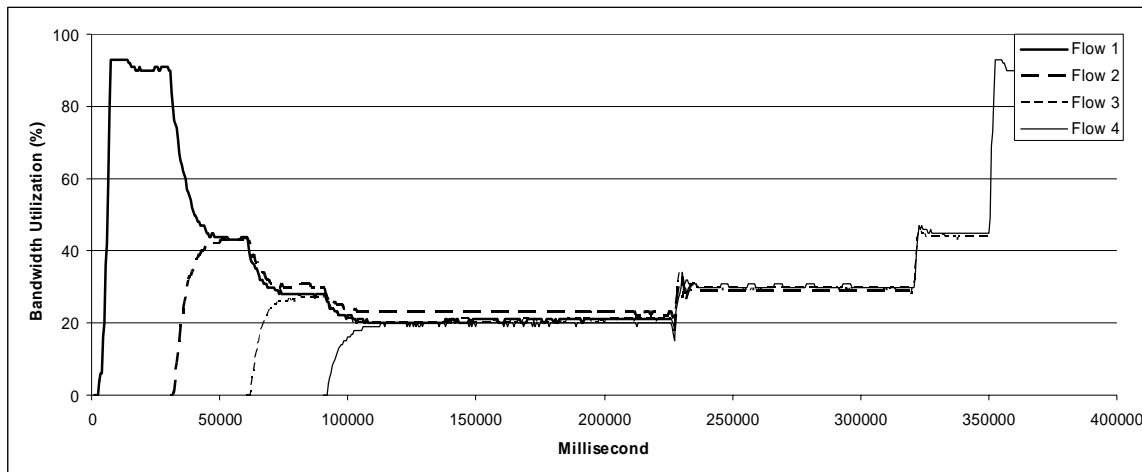


Figure 13: XCP flow allocation

This result is in sharp contrast to TCP, as seen in Figure 14. Each TCP flows’ bandwidth utilization oscillates as each flow is competing with all the other flows for bandwidth.

The rapid change in throughput for each TCP flow can be problematic for applications needing a constant bit rate. XCP clearly outperform TCP in this scenario, these results are in line with what the original simulations [1] and Zhang and Henderson [4] reports.

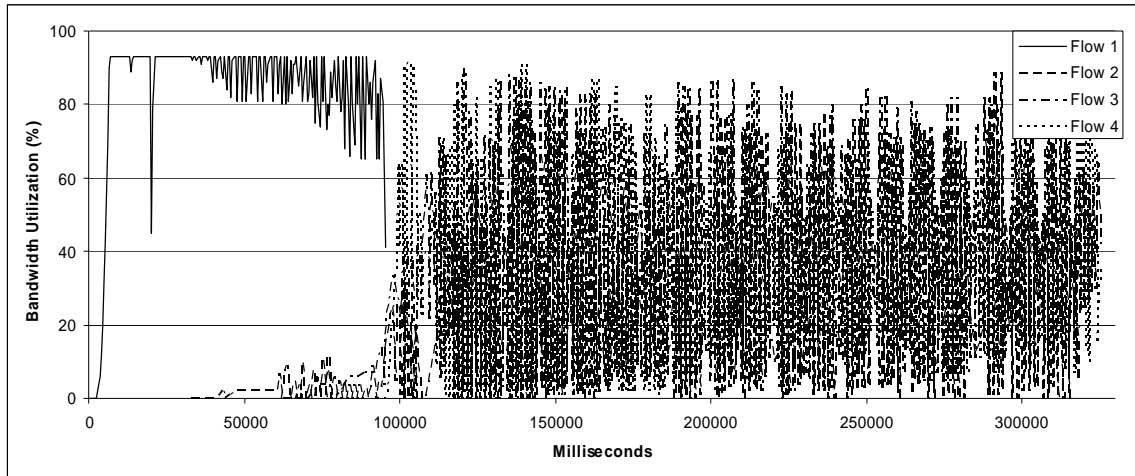


Figure 14: TCP flow allocation between 4 flows

4.4 Half duplex XCP router

The XCP feedback formula is based on the assumption that the XCP router easily can calculate the total bandwidth, and uses this calculation to figure out how much capacity is available at all times. If the router is full duplex this assumption appears to be valid under normal circumstances. However, if the XCP router is connected to the network on half-duplex links, the packets containing data will need to contend with returning ACKs on the link. This makes it impossible for the XCP router to know exactly how much bandwidth is actually available, leading to imperfect feedback being returned to the XCP host. In order to test this hypothesis we changed the 10Mbit full duplex links around Computer C (Figure 7) to half duplex. This makes the XCP router's (Computer B) bottleneck link half duplex as well.

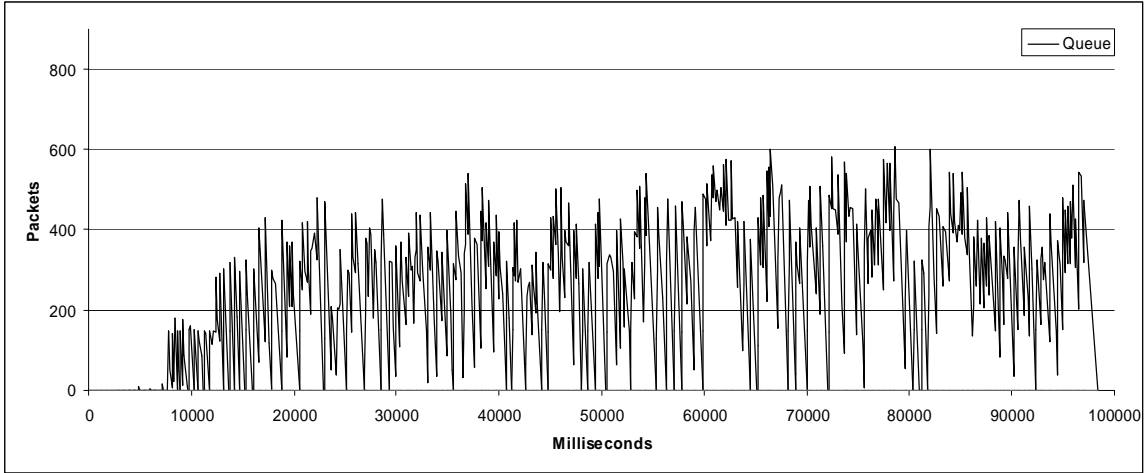


Figure 15: XCP router queue when running on half duplex network.

As Figure 15 shows, the XCP router fails to provide correct feedback to the XCP host, leading to a large, unpredictable queue size. There is very little difference compared to how TCP performs under the same circumstances, as shown in Figure 16.

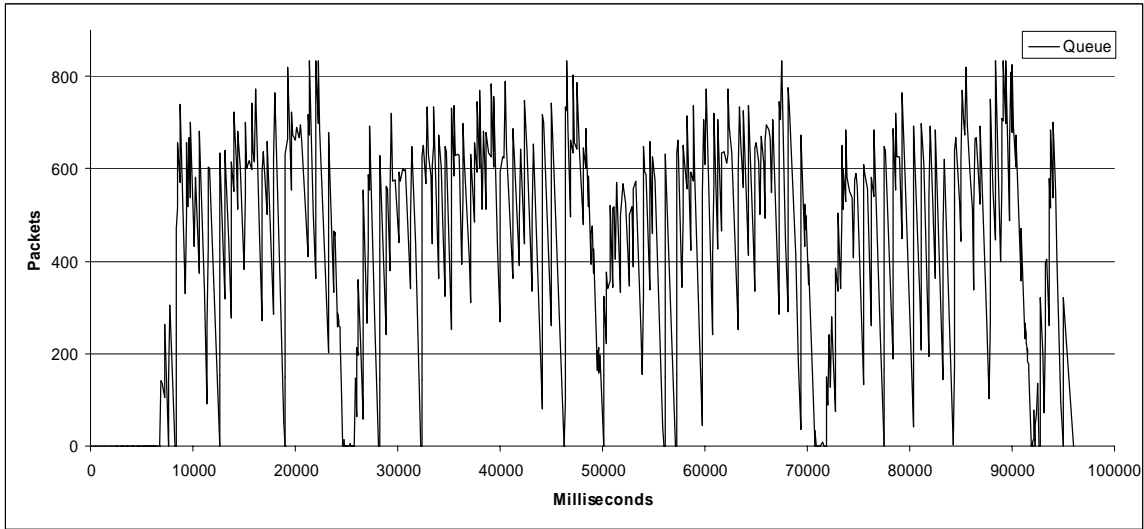


Figure 16: TCP router queue size when running on half duplex network

The TCP protocol will ultimately cause packet drops, as it is how TCP knows it has maximized its throughput. In this experiment the XCP queue size never reached high enough values to cause packet drops, as the maximum queue created was around 600 packets and the router could buffer 833 packets. If the XCP router had been equipped with a smaller packet queue, the router would be forced to drop packets as well, in an unpredictable fashion, undermining one of the major advantages of the XCP protocol.

The next two figures show the difference between bandwidth utilization of 4 XCP flows, started at the same time, when the router is running on a half duplex link, versus a full duplex link. These tests were done running the XCP router with the capacity set to 1250 b/ms. The small, but constant queue that a XCP router has when running in 1250b/ms

leads to minor fluctuations in each flows bandwidth utilization (Figure 17). However, when the same router is running on a half-duplex link (Figure 18), the XCP bandwidth utilization becomes much more unpredictable between each flow, reducing the ability of the XCP protocol to maintain constant bit rate drastically. This effect was also reported by Zhang and Henderson [4] in their Linux implementation.

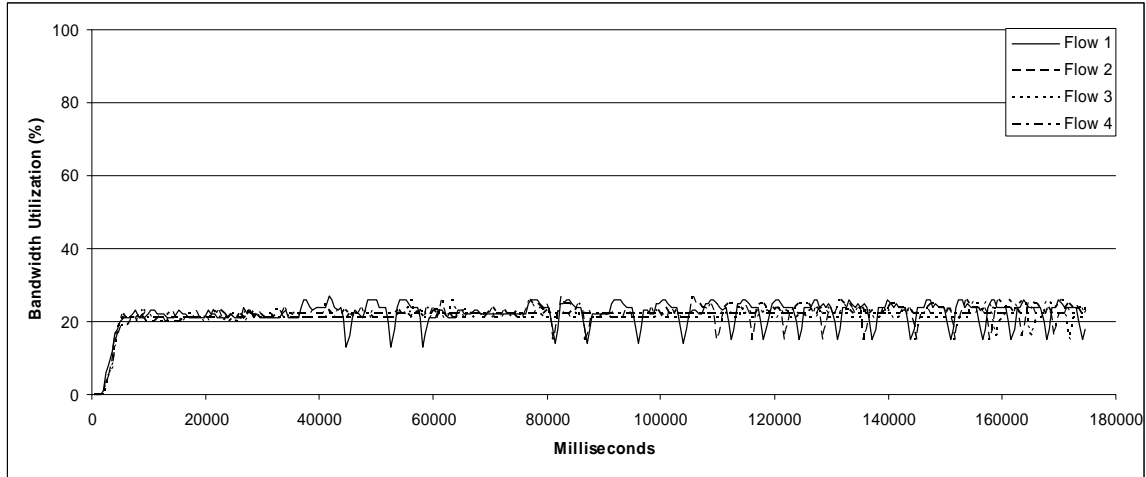


Figure 17: XCP Bandwidth utilization for full duplex router.

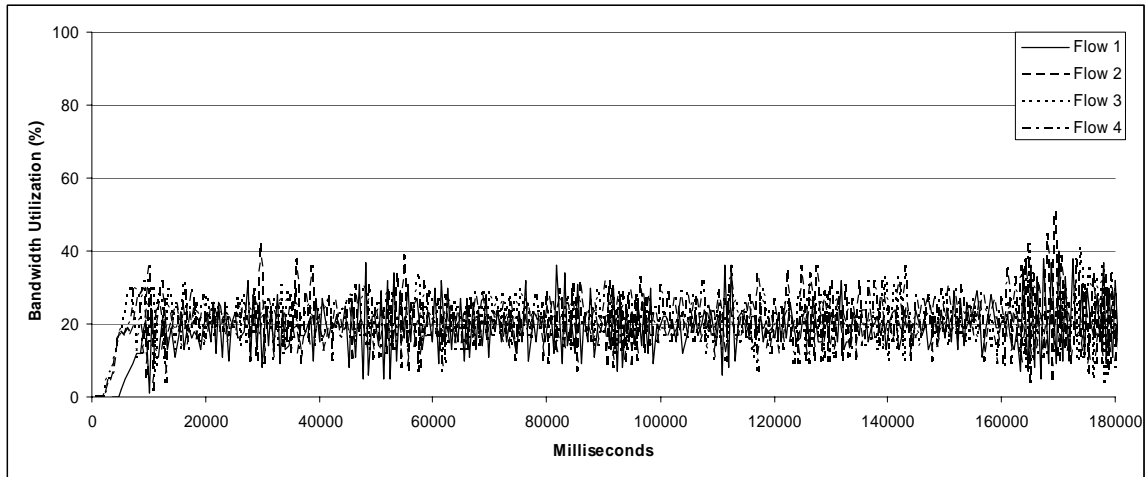


Figure 18: XCP Bandwidth utilization for half duplex router.

4.5 Sender not utilizing the full out-bandwidth

In the next experiment, we let the XCP host (Computer A) use less bandwidth than its maximal outbound network bandwidth. This is a quite common scenario, where the producer of data stops sending data for some unknown amount of time. An application program can for example be doing some processing of data before sending them.

To test how XCP performs under such a scenario, the test program is set up to repeatedly transfer one megabyte of data at full speed, before pausing for one second. In order to stress the XCP protocol further, we halved the queue length in the XCP router down to *bandwidth-delay* bytes (415 packets). By halving the queue length, the router will start to drop packets if it tries to buffer the entire 1 MB of data sent by the client. The rest of our network is set up as indicated in Figure 7.

Because we have used Formula (8) to estimate our current throughput, we are only able to reduce the congestion window if we receive negative feedback from the XCP router (see Formula (9) for details). As the XCP host does not reduce the congestion window while the application is not sending data, the XCP router will easily be flooded with data due to the large congestion window that the host has opened during the previous control interval. This is basically the same effect as described in Example 11.

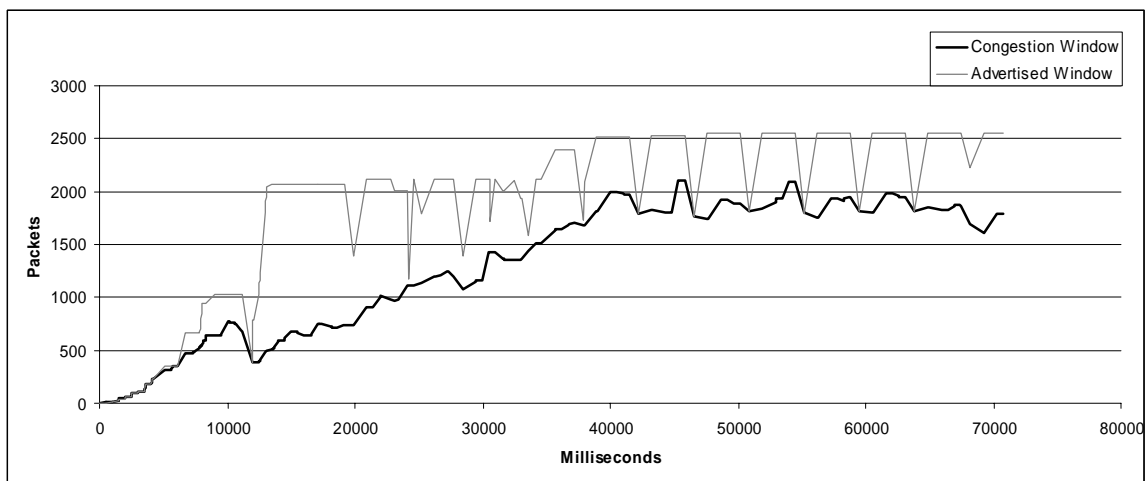


Figure 19: Congestion Window in XCP host sending at alternating speed

As the graph in Figure 19 shows, the congestion window is now continually growing at the XCP host, because there is no effective mechanism preventing it from being increased. When sending at maximum speed for this network, the congestion window converged around 500 packets (Figure 9). In this example, we are sending less data per second, but the congestion window is reaching values of over 2000 packets. The reason for this behavior is that each time the XCP host is slowing down (taking a pause), and not overloading the router, the XCP router will tell the host to increase its sending speed, thereby increasing the XCP host's congestion window. However, the congestion window

will continue to stay at the previous high value as long as the XCP router is not overloaded. The graph shows that the congestion window often is only prohibited by TCP's advertised window from growing even further. The inflated congestion window allows the XCP host to transfer large amounts of data without any restrictions into the network, easily overloading the XCP router (Figure 20).

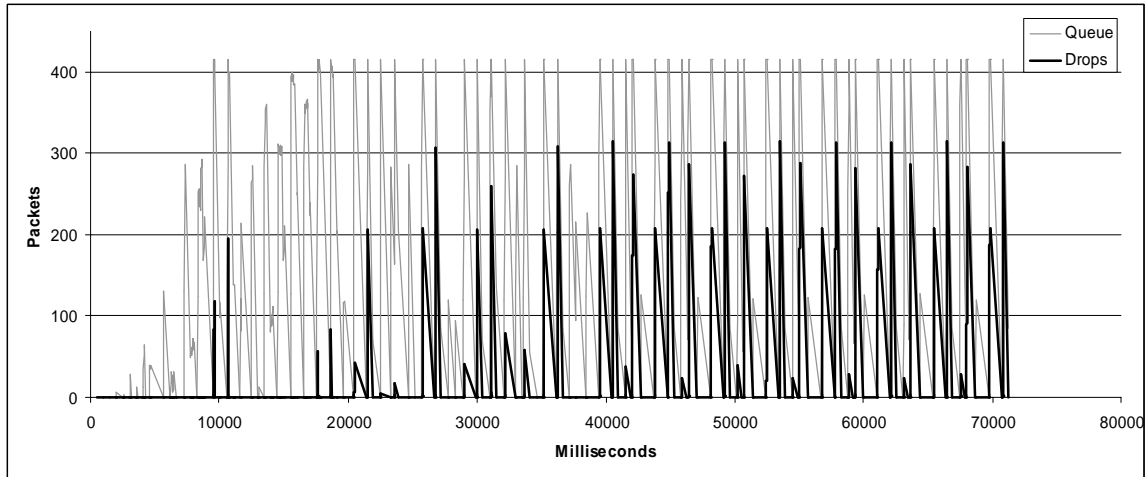


Figure 20: Queue and packet drops in XCP router

Under normal circumstances, overloading the XCP router would lead the XCP protocol to reduce the host's sending speed. However, the application's "bursty" behavior completely takes the XCP router by surprise. The XCP router is not able to adjust the feedback fast enough to prevent the XCP host from creating massive queues in the router. And when the queues get large enough, the router starts to drop packets, which is the one thing that the XCP router is designed to prevent.

There are multiple reasons for the failure of XCP to work under this scenario. First of all there is nothing in the XCP formulas that take into account the maximum length of a router's queue. The XCP feedback formula (3), only takes into account the length of the queue, not how large the queue can be. The XCP router will return exactly the same feedback regardless of how full (in percent) the router's queue is.

Another problem leading to the failure of the XCP protocol is the increased RTT as a result of queue build-up and packet drops (Figure 21). The XCP routers control timeout interval is initially around 500ms. However, as the queue builds up, and packets are dropped, the round trip time increases to between 700 and 1100 ms. Again the XCP feedback formula will fail to detect that it has allowed the sender to send too much data. Since the XCP router calculates the feedback based on the average load during the last RTT ms, an increased RTT reduces the perceived average load on the router. Each burst of 1MB of data is therefore measured at an average of between 909-1428 B/ms, around the 1250 B/ms that the router thinks it can handle. This means that the XCP feedback formula will not try to reduce the congestion window noteworthy of the XCP host, since the router does not think it is overloaded.

The XCP feedback Formula (3) does take into account the queue size when calculating the aggregated feedback. However, the queue size used is the minimum standing queue during the control interval (Formula (4)). As Figure 20 shows, the queue size is oscillating because of the oscillatory sending behavior of the XCP host. This leads the minimum queue size during a 700–1100 ms interval to be very low, often zero packets. By using a very low estimate for queue size, the feedback formula is again incapable of slowing down the XCP host.

Finally, the XCP protocol does not use packet drops as a way of reducing the sending rate, since XCP uses its own congestion control algorithm. When this algorithm fails, as seen above, the XCP protocol will continue sending data too fast, as neither packet drops nor XCP feedback will reduce the sending speed of the protocol.

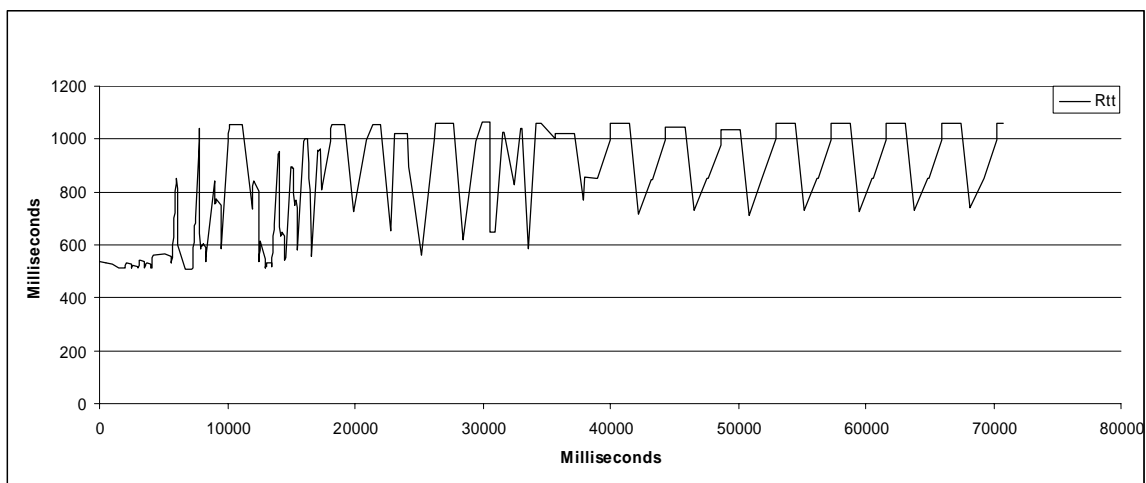


Figure 21: Round trip time as measured by XCP sender

TCP on the other hand, does not rely on precise feedback from the network, and is just sending at the allowed rate given by the host’s congestion window. However, TCP reacts to packet drops and combined with the use of a much less aggressive feedback algorithm (AIMD) than XCP, manages to prevent the large build-up of queues in the router. As can be seen from Figure 22, TCP’s congestion window usually stays well below the maximum of 600 packets. A congestion window of 600 packets leads to packet drops, which causes a rapidly decreased in sending speed to fit the properties of the network. The packet drops can be seen in Figure 23, and in contrast to XCP, packet drops leads to reduced congestion window at the sender. In addition, TCP reduces its congestion window when idle and this further prevents the application from bursting large amounts of data into the network after an idle period. By keeping the queue size modest in the router, TCP also manages to keep the RTT lower and less oscillatory.

In this example, TCP clearly outperforms XCP, even though XCP continuously receives feedback from the router. XCP bases all its decisions on the feedback returned from the XCP router, and if this feedback is incorrect, the XCP protocol stops working as intended.

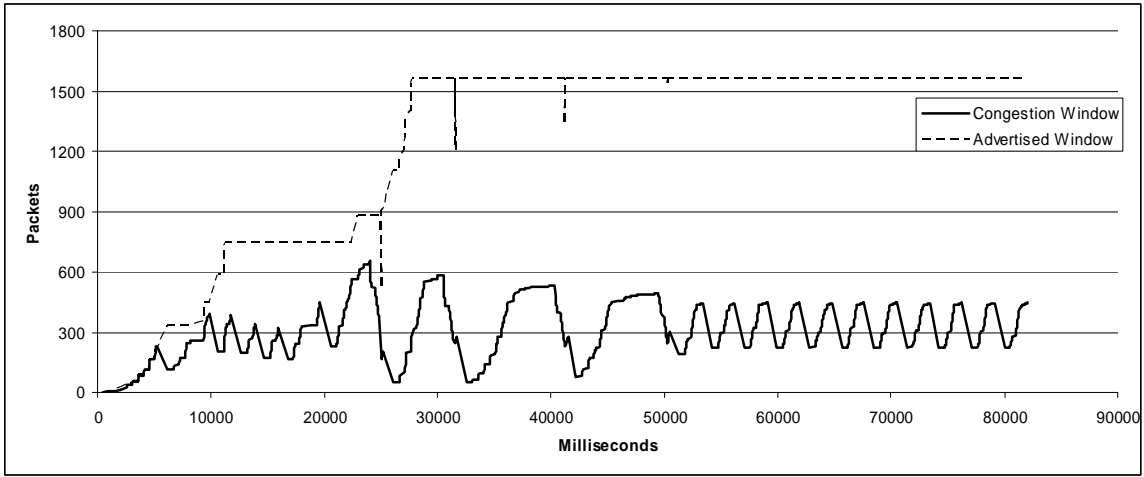


Figure 22: Congestion and advertised window measured from TCP host

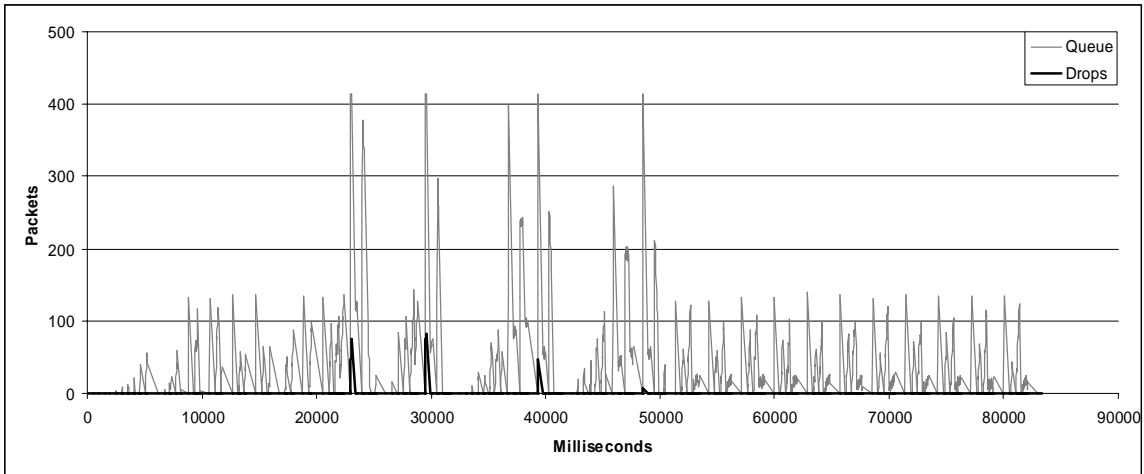


Figure 23: Queue and packet drops in router, with 1 TCP flow

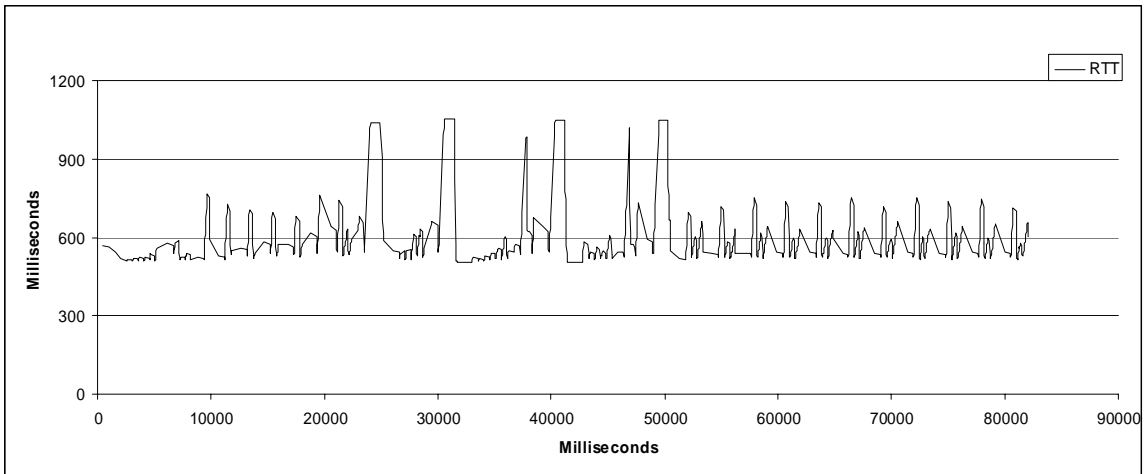


Figure 24: Round Trip Time measured by TCP host

4.6 Summary

The XCP protocol tries to improve TCP's congestion control and bandwidth utilization by explicitly having the routers participate in the sender's decision to send more or less data. Our results show that XCP successfully can manage to prevent queue build-up, and also manages to share the available bandwidth fairly and effectively between different TCP flows in a stable manner. However, our results also show that XCP in its current form is very vulnerable to an incorrect setup and clients that does not send with a constant rate. XCP does not function as intended in situations where the XCP router fails to correctly model the load on itself and the network.

The usage of Formula (8), as an estimation of throughput clearly leads to invalid XCP host behavior. The XCP protocol is dependent on hosts knowing their exact actual bandwidth utilization, in order for the hosts to correctly adjust their feedback from the XCP routers. Not being able to implement Formula (9) as specified in the XCP specification [2] has dramatic repercussions when XCP hosts do not utilize their entire out-bandwidth. Formula (9) is needed in order to prevent XCP hosts from bursting data into the network, as well as reducing the sending speed when little or no XCP feedback is received by the host.

Our test-setup (Figure 7) reveals another problem with using TCP's congestion window in any throughput calculations. Computer A's connection towards the XCP router (Computer B), is very fast and has a low RTT. In contrast, the connection towards Computer D is slower and has a significantly higher RTT. If the congestion window is used as a measurement of allowed transfer-speed, it allows Computer A to send out data in bursts, which can be problematic for the XCP router. For example; sending 0,5MB of data on a 100 Mbit link, with an RTT of 1 ms, requires 40ms and a congestion window of 12.500 bytes (9 packets). If the XCP router has allowed a bandwidth of 10 Mbit over the flows entire length (from A to D), having an RTT of 500ms, it needs to allow a congestion window at A of 625.000 bytes (417 packets). This congestion window is more than enough to allow Computer A to burst out the data at full speed of its local interface (100 Mbit). The XCP router is not capable of processing the incoming data at this speed, leading to queue build-up in the XCP router. In the worst case this could lead to packet drops, causing much of the same problems as described in Section 4.5. This shows that the XCP router can experience packet drops even if the XCP host is sending within the allowed speed limit, because the sender is sending data in burst. Therefore, for XCP to work as intended an XCP host must not use TCP's congestion window, but instead calculate the actual transfer speed and make sure that it does not send out packets in large bursts.

Another issue is the way the XCP router allocates feedback to each packet. The existing feedback formula does not take into account what previous feedback has been given at all. If clients do not send as much data as they are allowed to by XCP, it can easily lead to overestimation of the available feedback in the router, and thereby granting XCP hosts the right to overload the XCP router at a later point in time.

XCP routers need be able to correctly measure the load on the network in order to work as intended. If the XCP router fails to measure the load correctly, invalid feedback will be returned to the hosts, leading to unpredictable protocol behavior. Half-duplex links are problematic for XCP routers, as it is not possible to for the XCP router to calculate the correct load based only on the amount of bytes flowing in one direction. The current XCP router feedback formulas have no way of taking into account the reduced bandwidth based on the number of ACK-packets flowing through the router in the opposite direction.

In the common scenario where a program does not continuously send data at the given maximum rate, the XCP routers feedback mechanism fails. The efficiency controller base its feedback decision entirely on what has happened during the last RTT. It does not take into account any previous feedback decisions given in previous intervals, which could have reduced the oscillations caused by such application. In addition, the queue estimation timer uses minimum standing queue as a measurement of the queue length in the router. When the queue is oscillating because of hosts transmitting data in bursts, this algorithm will fail to see the real queue size. Using average queue, size instead of minimum queue size, might lead to more correct feedback in such situations.

5 Conclusions and remaining challenges

In this thesis we have investigated the XCP protocol and taken a more thorough look at the XCP specification and its issues. By creating a working Linux implementation based on the specification, we managed to see if the XCP protocol could take the step from theory to practice. As XCP was conceived from a theoretical view of the world, it had problems fitting into existing network implementations of the TCP/IP stack. The XCP protocol's main goal was to improve the way TCP work on networks with high *bandwidth -delay* products, by using a fundamentally different approach than the currently existing TCP/IP implementations. This new approach made it very difficult to implement the protocol without doing modifications to the existing network code. It was not given that a protocol looking good on paper and in theory, would actually perform well in real-life scenarios. Our tests show that, while the XCP protocol managed to deliver on some of its promises, it failed to deliver in key situations that might arise in the real world. However, if comprehensive and correct network information was available to the XCP router the XCP protocol would outperform TCP, as shown in section 4 and in the simulations [1]. In such situations the XCP protocol can accurately predict its load at all times, and if all the XCP hosts participating also reports correctly on current throughput and other network parameters, the XCP protocol seems live up to its promises.

TCP's main responsibilities are guaranteed packet delivery, ordering, congestion prevention and at the same time maximizing throughput, i.e., all this without assuming anything of the network it runs on. XCP wants to handle TCP's congestion and throughput control, thereby replacing two of the major parts of the TCP protocol. We feel that creating a new protocol entirely between IP and TCP pretending to be a real protocol layer is not the right way to go. However, we did manage to get XCP to work in the Linux kernel, without changing any other network code. Unfortunately this approach introduces various "hacks" in order to bypass the layered architecture of the TCP/IP stack. As noted in Section 3.1, there are quite a few challenges with having XCP as a protocol layer below TCP. In order for XCP to work as intended by the authors, it should be clear that XCP needs to be implemented at the same layer as TCP, replacing major parts of the TCP protocol. To be in full control of the actual transmission speed, XCP cannot be implemented below the TCP layer, as this implies that XCP is dependent on whatever sending speed TCP use. Adjusting the congestion window as a way of controlling the transfer rate from anywhere in the protocol stack, other than from within the TCP protocol, is a serious violation of the protocol layering structure.

If XCP was implemented as an extension to TCP it would be possible to control the sending speed more correctly, in addition, "Aging of Allowed Throughput", in the XCP specification [2] (given by Formula (9)), would also be possible to implement, as the TCP layer can know its actual sending speed. In addition, XCP needs to know to which flow a packet belongs in order to provide the correct feedback. Implementing XCP as an extension to the TCP layer removes this problem in its entirety, as the flow concept is an

integral part of the TCP protocol. Therefore, such an implementation would not need to worry about which flow a given packet belongs. Implementing section 3.1.3.2, “Response to Packet Loss” in the specification, can only be done if XCP is implemented as a TCP extension as well. Otherwise, the XCP protocol would need to check every packet leaving the TCP layer, and basically duplicate the TCP sliding window algorithm. In addition, the problems we encountered with hardware checksums (see Section 3.1) would also disappear as no data would be placed between the IP and TCP header.

With XCP implemented as a TCP extension, it would be natural to have its protocol header appended to the TCP header as a TCP option. However, in order for the XCP routers to work they must process the XCP header for each packet they receive. Most routers will only route packets based on the IP header, so something in this header must alert the router that it should examine the packet further and look for an XCP header. Our implementation worked by using a different protocol number for XCP, thereby allowing the router to react on this number in order to do the required processing. A TCP header containing the XCP header as TCP options could in theory be implemented in this way. However, it would mean that we had created a new protocol (a TCP-XCP protocol) and not extended the TCP protocol. Another approach would be to use the netfilter facility to examine all IP packets passing the router, searching for a TCP header with a given XCP option. Yet another possibility would be to set an IP option notifying any XCP aware router of the existence of an XCP option in the TCP header. In all cases the router must depart from its fast-path packet routing algorithm and do further investigation of the packet. This packet processing is much slower than the fast-path processing in a router. It would make little difference to the overall performance of the XCP router, compared to the departure from the fast-path, whether the XCP protocol was implemented as a TCP option, an IP option or as a new protocol.

The scalability of the XCP protocol is also something that could be a potential problem. Each XCP packet requires some extra calculation in the router and prevents the usage of the fast-path. XCP is conceived to solve problems on networks with a high *bandwidth-delay* product. For future networks it is likely that the bandwidth component increases much more rapidly than the delay component. As the speed of networks increases, the number of packets the routers need to process increases. XCP being dependent on calculations made on a per packet basis is likely to quickly become a bottleneck for high-performance routers. In addition the added load of calculating feedback for each output device each RTT will also require quite a lot of work on high-speed routers with multiple output interfaces.

The XCP protocol has been submitted as a draft RFC. Its initial simulation results have shown some promising capabilities, preventing packet loss and maximizing throughput in a stable manner. As an experimental idea XCP challenges the view that the network should be considered a “black box”. However, there are some obstacles left for XCP to pass before it can be deployed in the real world. As reported in our study, and in the paper released by Zhang Y., Henderson T. [4], the XCP protocol fails to converge under circumstances where it does not have total control over all the relevant network parameters. The XCP protocol is very sensitive to clients not using their allowed

bandwidth, to inaccurate router settings and other network settings, such as half – duplex networks, that foils the XCP routers attempts to create a valid model of the exact load. Another issue is the performance penalty that the increased complexity of XCP adds to the routers implementing the protocol. Increasing a router’s workload is probably not the way to go, when trying to cope with very high bandwidth links.

The biggest change from most other protocols is to actually assume the routers can supply information about their congestion level. This assumption can hinder XCP’s deployment as a protocol. In addition, XCP capable routers and hosts needs to be in place in the entire path of a flow for the protocol to fully work as intended. As this is unlikely to happen all at once in the Internet, it is therefore very likely that XCP routers will need to coexist with non-XCP capable routers. Exactly how this is supposed to work is an open issue, as congestion in such networks can go undetected by XCP. It has been suggested to let XCP fall back to classic TCP behavior when packet loss is detected, and that an endpoint that is limited by using its end-to-end congestion algorithm (typically TCP) would mark the XCP header with a flag, and thereby allowing routers to process its packets differently than those being limited by XCP.

As long as XCP is not capable of running correctly in “less than ideal” situations, the future for the protocol looks dire. Additionally, more work is needed to create a more solid efficiency controller algorithm that can handle incorrect and unpredictable network information and misbehaving clients. The XCP clients also need a more fine-grained measurement of current throughput, as using TCP’s congestion window is not suitable for precise adjustments of the data sending rate.

We also see security challenges for the XCP protocol. The XCP routers base the feedback on values set by the XCP hosts. A malicious host could easily use invalid XCP header values to make the XCP router return invalid feedback values to all participating hosts. We have not analyzed the consequences such attacks could have for the XCP protocol in detail, but note that it is inherently dangerous to base computing decisions on input that cannot be verified.

6 References

- [1] Katabi D., “Congestion Control for High Bandwidth – Delay Product networks”, *SIGCOMM’02, August 19 – 23, 2002, Pittsburgh, Pennsylvania, USA*, <http://www.cs.stonybrook.edu/~amohr/cse646/papers/katabi-xcp-sigcomm2002.pdf>
- [2] Katabi D., “Specification for the Explicit Control Protocol (XCP)”, <http://www.isi.edu/isi-xcp/docs/draft-falk-xcp-spec-00.txt>, October 2004
- [3] Falk A. “Experimental Measurements of the eXplicit Control Protocol (XCP)”, *Information Sciences Institute*, <http://www.isi.edu/isi-xcp/>
- [4] Zhang Y., Henderson T., “An Implementation and Experimental Study of the eXplicit Control Protocol (XCP)”, <http://www.isi.edu/isi-xcp/share/extended.pdf>, April 2005
- [5] Postel J., “Transmission Control Protocol”, *Internet Request For Comments, RFC 793, September 1981*
- [6] Floyd S., Jacobsen V., “Random early detection gateways for congestion avoidance”, *In IEEE/ACM Transactions on Networking*, 1(4):397- 413, August 1993
- [7] Low S.H., Paganini F., Wang S., Adlakah S., Doyle J.C., “Dynamics of TCP/RED and a scalable control”. *In Proc. Of IEEE INFOCOM, June 2002*. <http://netlab.caltech.edu/FAST/publications/infocom02.pdf>
- [8] Athuraliya S., Li V. H., Low S. H., Yin Q., “REM: Active Queue Management”, *IEEE Network Magazine*, vol. 15, pp. 48-53, May 2001
- [9] Stevens W. “TCP Slow Start, Congestion Avoidance, Fast Retransmit, and Fast Recovery Algorithms”, *Internet Request For Comments, RFC 2001, January 1997*
- [10] Clark D., “Window and acknowledgment strategy in TCP”, *Internet Request For Comments, RFC 813, July 1982*
- [11] Mathis M., Mahdavi J., "Forward Acknowledgment: Refining TCP Congestion Control", *Proceedings of ACM SIGCOMM, Stanford, CA, August, 1996*
- [12] Mathis M., Mahdavi J., Floyd S., Romanov A., “TCP Selective Acknowledgment Options”, *Internet Request For Comments, RFC 2018, October 1996*
- [13] Floyd S. “HighSpeed TCP for Large Congestion Windows”, *Internet Request For Comments, RFC 3649, December 2003*

- [14] The Fedora Project. *February 2005.*
<http://fedora.redhat.com/>

- [15] The Linux Kernel. *February 2005.*
<http://www.kernel.org/>

- [16] TCP Tuning Guide. *February 2005.*
<http://www-didc.lbl.gov/TCP-tuning/linux.html>

- [17] Network Emulator, netem. *February 2005.*
<http://developer.osdl.org/shemminger/netem/index.html>