An Implementation and Experimental Study of the eXplicit Control Protocol (XCP)

Yongguang Zhang
HRL Laboratories, LLC, Malibu, California
Email: ygz@hrl.com

Thomas R. Henderson
Boeing Phantom Works, Seattle, Washington
Email: thomas.r.henderson@boeing.com

***Abstract*— The eXplicit Control Protocol (XCP) has been proposed as a multi-level network feedback mechanism for congestion control of Internet transport protocols. Theoretical and simulation results have suggested that the protocol is stable and efficient over high bandwidth-delay product paths, while being more scalable to deploy than mechanisms that require per-flow state in routers. However, there is little operational experience with the approach. Since the deployment of XCP would require changes to both the end hosts and routers, it is important to study the implications of this new architecture before advocating such wide scale changes to internets.**

This paper presents the results of an experimental study of XCP. We first implemented XCP in the Linux kernel and solved various systems issues. After validating previously reported simulation results, we studied the sensitivity of XCP's performance to various environmental factors, and discovered issues with TCP/IP configuration, capacity misestimation due to link sharing, handling of non-congestion losses, and the partial deployment of XCP queues in the network. These sensitivities can significantly reduce XCP's ability to control congestion and achieve fairness. Our contributions are twofold. First, through implementation we have revealed the challenges in platforms that lack large native data types or floating point arithmetic, and the need to keep fractions in the XCP protocol header. Second, through experiment and analysis we have identified several possibilities for XCP to enter into incorrect feedback control loops and adversely affect the performance. The challenges identified are deployment challenges intrinsic to the XCP design, and they suggest that the current proposal requires additional development and extension.

I. Introduction

The eXplicit Control Protocol (XCP) is a new Internet congestion control protocol developed by Katabi, Handley, and Rohrs [1]. XCP has attracted attention in the research community because of its promise to potentially obtain high utilizations in high bandwidth-delay product networks, while maintaining low standing queues in routers, without requiring that per-flow state be kept in routers. Rather, XCP keeps state in the packet headers, and requires routers to perform operations in aggregate.

Like the reliable transport protocols TCP and SCTP, XCP is a window-based protocol and implements congestion control at the endpoints of a connection. Senders maintain their congestion window (cwnd) and round trip time (RTT) and communicate these to the routers via a congestion header in every packet. Routers monitor the input traffic rate and persistent queue size for each of their output queues. Based on the difference between the output link capacity and the flow's cwnd and RTT, the router tells each flow sharing that link to increase or decrease its cwnd by annotating the feedback in the XCP headers. A more congested downstream router can further reduce this feedback by overwriting it. Ultimately, the packet will contain the feedback from the bottleneck router along the path. When the feedback reaches the receiver, it is returned to the sender (typically piggybacked on a transport protocol acknowledgement packet), and the sender updates its cwnd accordingly. The process is continuous and the responses by the sender to the network feedback take on the order of one RTT to take effect. The control laws ensure that the system converges to optimal efficiency and min-max fairness [1], [2].

Although the simulation results have shown that XCP controllers are stable and robust to estimation errors, and require only a few per-packet arithmetic operations [1], because it relies on floating-point operations, it has not been clear how the simulation models might translate to implementation code. Also, it is not well understood how XCP might perform in a partially deployed environment. We are aware of only one other ongoing XCP implementation effort, at USC's Information Sciences Institute. Initial implementation results were published in February 2004 [3]. The FreeBSD-based kernel does not have the same limitations on double long division that we encountered in the Linux kernel, and the initial results presented do not consider operation in mixed deployment scenarios. XCP itself is one of several proposals in the area of "high-speed" TCP extensions that have been recently proposed, including Scalable TCP [4], HighSpeed TCP [5], FAST TCP [6], and BIC-TCP [7]. The main difference between XCP and the other high-speed TCPs is that XCP relies on explicit router feedback, while high-speed TCPs are primarily end-to-end in nature.

We have conducted an experimental study of XCP and its deployment by implementing XCP in Linux and conducting a testbed evaluation. While our initial validation results match the previously reported simulation results, we do have some surprising findings. We first met an implementation challenge that arise from the lack of double-long or floating point arithmetic in Linux kernel. We then explored how the the choice of data types in the system (implementation variables and protocol header syntax) affects precision and dynamic range of XCP. Next, our experimental results revealed that XCP's performance can be adversely affected by environmental factors including TCP/IP configuration, link sharing, non-

congestion loss, and the presence of non-XCP queues. And finally, our analysis shows several possibilities for XCP to lose its ability to control congestion and achieve fairness.

This paper reports on the findings of this study. Section II describes our implementation approach, the challenges discovered in implementing the protocol in the Linux kernel, and our resolutions of those problems. Section III begins our experimental evaluation with a number of simple experiments that confirm the performance previously reported in simulation studies. Section IV extends the experiments to cover various operational and environmental factors that may affect XCP performance. Section V further analyzes the XCP control laws to identify areas of potentially incorrect feedback control loops that XCP must avoid. Finally, we discuss a number of deployment issues in Section VI, followed by the conclusions of this study.

II. Implementation Details

A. Overall Architecture

For evaluation purposes, we have implemented XCP as a TCP option. Other possible approaches would be to implement XCP as a separate transport protocol, to implement XCP as a "layer 3.5" header between the IP and transport headers, or to implement XCP as an IP header option. There are pros and cons to each approach, as discussed later in Section VI. Most of the implementation issues and performance results presented below are independent of this design choice.

For applications communicating with our XCP implementation, TCP is still the underlying transport mechanism that delivers data, but the use of XCP is negotiated upon connection setup. The effect of the XCP option is to modify the TCP sender's congestion window (cwnd) value according to the XCP protocol. Implementing XCP as a TCP option allowed us to borrow as much as possible from the existing protocol and software structure, resulting in a much faster development cycle, and backward compatibility with legacy TCP stacks.

In such an architecture, whenever TCP sends a data segment, the XCP congestion header is encoded as a TCP header option and attached to the outgoing TCP header. The feedback field of the XCP option can be modified by routers along the path from sender to receiver, and the receiver returns the feedback in TCP ACK packets, also in the form of an XCP option in the TCP header. Upon receiving an XCP option from an incoming ACK packet, the TCP sender updates its cwnd accordingly.

Our software architecture for the XCP implementation includes two parts. The first is a modification to the Linux TCP stack to support XCP end-point functions, and the second is a kernel module to support XCP congestion control in Linux-based routers. In addition, a simple API is provided so that any application can use XCP by opening a TCP connection and setting a special socket option.

B. XCP Option Format

Two XCP option formats are defined, one on the forward direction that is part of the data segments from sender to receiver, and the other on the return direction that is part of the ACK segments from receiver to sender. Since routers should only process the forward-direction XCP option, having two option formats makes it easy for the routers to know which direction the XCP option is traveling. Further, it paves the way to support two-way data transfer in a TCP connection, which will have forward-direction XCP options traveling in both directions (from both end-points).

Forward direction:

opt	optsize	
14	8	H_feedback
H_rtt		H_cwnd

Return direction:

opt	optsize	
15	4	H_feedback

Fig. 1. XCP option formats in both directions

Fig. 1 illustrates the two formats. As a prototype, we simply picked two unused values (14 and 15) for the TCP options without making any attempt to go through standardization. Other than `opt` and `optsize`, the remaining three fields are XCP congestion header fields [1]. They are each 16 bits long.

`H_cwnd` stores the sender's current cwnd value, which is measured in packets (segments). `H_feedback` is also measured in packets because this is the unit of change in TCP's cwnd in Linux kernel. However, we should not simply use a short integer type because that would limit the value to plus or minus 32,000 packets, which may not be sufficient in some cases with extremely large delay-bandwidth product. Further, as we will explain later in Section II-E.2, the XCP feedback value must not be rounded to an integer. Therefore, we choose a split mantissa-exponent representation that expresses the 16-bit `H_feedback` as the following:

- The most significant 13 bits are the signed mantissa (m)
- The remaining 3 bits are the unsigned exponent (e)
- The value stored in `H_feedback` is $m \cdot 16^{(e-3)}$

This format can represent a cwnd value from $\pm 2^{-12}$ (0.000244) to $\pm 2^{28}$ (268,435,456) and 0.

`H_rtt` is measured in milliseconds. Given that it is an unsigned 16-bit value, the maximum round trip time supported by this XCP implementation is around 65 seconds, which should be suitable for most cases.

C. XCP End-point Functions

1) XCP Control Block and cwnd updates: Like TCP, there is a control block for each XCP connection endpoint to store XCP state information. Since XCP is implemented as a TCP option, the XCP control block is part of the TCP control block (`struct tcp_opt`). It has the following major variables:

```
int  xcp;  /* whether to use XCP option */
struct xcp_block {
  __u16  rtt;  /* RTT estimate for xcp purposes */

  __s16  feedback_req;  /* cwnd feedback request */
  __s16  feedback_rcv;  /* received from XCP pkt */
  __s16  feedback_rtn;  /* returned by receiver */
```

```
  __s16  cwnd_frac;     /* cwnd fractions */
  __u32  force_cwnd;    /* restore cwnd after CC */
} xcp_block;
```

The three feedback variables correspond to the `H_feedback` field: `feedback_req` is the requested amount that XCP sender will put in outgoing XCP packets, `feedback_rcv` is the feedback amount that the XCP receiver receives before passing it back to the sender, and `feedback_rtn` is the amount that the XCP sender finally receives. To support delayed ACKs, feedback from several packets can be accumulated in both `feedback_rcv` and `feedback_rtn`. The same mantissa-exponent data type is used in these three variables.

We added steps in TCP's usual ACK processing to handle XCP options. Upon receiving a TCP packet with an XCP option, the host updates one of the above variables. At the sender, the `feedback_rtn` amount is added to TCP's cwnd (`snd_cwnd`). Since cwnd grows or shrinks in integer units, the leftover fractional part is stored in `cwnd_frac` and will be added back to `feedback_rtn` next time.

2) Integrating with TCP congestion control: An artifact of implementing XCP as an TCP option is that we have to integrate it with TCP congestion control. Since XCP should change cwnd only through router feedback, we could disable TCP congestion control altogether. It is, however, difficult to do so reliably because the Linux TCP implementation intermixes congestion control with other TCP functions. Further, we believe that mechanisms like fast retransmission are still useful in XCP. We therefore chose to preserve the TCP congestion control code but removed its effect on cwnd change, except in the timeout case in which cwnd is reset to one.

We do so with the `force_cwnd` variable in the XCP control block. After the sender updates its cwnd from XCP feedback, it stores the new cwnd value in `force_cwnd`. After subsequent TCP ACK processing and before any retransmission event, the sender restores the cwnd value from `force_cwnd`. This in effect allows fast retransmission but disables slow start and congestion avoidance (linear increase and multiplicative decrease). We will further discuss the issue of how XCP should respond to lost packets in Section VI.

3) Increasing advertised window: In TCP, the sender's cwnd is bounded by the receiver's advertised window size. Therefore, XCP may not be able to realize all of its cwnd value if the receiver's advertised window is not sufficiently large (meaning that the receiver cannot receive as much data). In the Linux implementation, the receiver grows its advertised window linearly by 2 packets per ACK. This is suitable for normal TCP but will be insufficient for XCP as XCP feedback may open up the sender's cwnd much faster. We therefore modified the TCP receiver so that the advertised window grows by the integer value of `feedback_rcv`. This modification is very important, as we will show later (Section IV-A) that there is an adverse effect if XCP cannot utilize all of its cwnd window.

4) Reducing the maximum number of SACK blocks: Although congestion loss is a diminishing event in an all-XCP network, there can be other sources of packet losses such as errors in wireless networks. As the selective acknowledgement (SACK) mechanism has been proven effective in dealing with losses in TCP, and many TCP implementation already include the SACK option, we should support SACK in our XCP implementation as well. In fact, we have validated this assertion through experiments (Section IV-C), and have concluded that it is very important for XCP to have the SACK mechanism to deal with non-congestion loss.

Our approach of implementing XCP as an TCP option has made this an easy task. The only code modification needed is to reduce the maximum number of SACK blocks allowed in each TCP ACK packet. Given that the maximum size of a TCP header (including options) is 60 bytes, with the timestamp and XCP options, we can now have at most 2 SACK blocks, compared to a maximum of 3 SACK blocks before we added XCP. We will demonstrate later that, even with a reduced number of SACK blocks, it still provides significant performance improvement in wireless networks. If XCP is implemented not as a TCP option but as a separate protocol header, this point will become moot.

D. XCP Router Module

Much of the complexity in the XCP protocol resides in the XCP router implementation. The router has to parse the congestion header in every XCP packet, compute individual feedback, and update the congestion header if necessary. This XCP router function is divided into two parts: the XCP congestion control engine that acts on the information encoded in each congestion header, and the kernel interface that retrieves such information from the passing XCP packets.

1) Kernel Support and Interface: The Linux kernel has two mechanisms, a loadable module framework and a device-independent queueing discipline layer, that allowed easy insertion and interfacing with the XCP engine. First, the entire XCP router function was implemented as a kernel loadable module with no change to the kernel source. Second, the Linux network device layer includes a generic packet queueing and scheduling mechanism called `Qdisc`. Its basic functions are to enqueue an outgoing packet whenever the network layer passes one down to the device layer, and to dequeue whenever the device is ready to transmit next packet. The Linux kernel includes several built-in Qdisc, including FIFO, CBQ, RED, etc.; more elaborate ones can be implemented and added as loadable modules. Our XCP router module provides two functions that any Qdisc can call to invoke the XCP engine (see Fig. 2): `xcp_do_arrival()` is called when an XCP packet is enqueued, and `xcp_do_departure()` is called when the XCP packet is ready to be transmitted in hardware. A new built-in Qdisc called "xcp" was included to operate XCP with a standard drop-tail queue (fifo).

To manage the new XCP Qdisc, we implemented a loadable module for the Linux traffic control command `tc`. With this command, it is very easy to use XCP Qdisc at any network

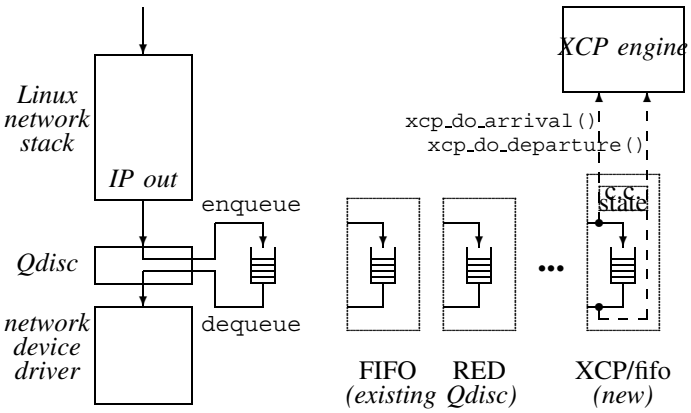


Fig. 2. Structure of XCP router module w.r.t. networking stack in Linux kernel

device or as part of another complex queueing system, such as at a leaf of a hierarchical token buffer. For example, the following command enables XCP for a 10Mbps link:

```
tc qdisc add dev eth2 root xcp capacity 10Mbit
```

The configuration parameter `capacity` should indicate the hardware-specified raw bandwidth. Proper adjustment must be made to account for the framing overhead when an XCP router calculates the true output link capacity (Section III-C).

2) Per-link State Information: XCP routers never maintain any per-flow state, and the XCP engine is completely stateless. Instead, the per-link congestion control state information is stored at each XCP-enabled Qdisc ("c.c. state" in Fig. 2). When the Qdisc invokes the XCP engine, it passes the packet (in an `skbuff` structure) along with the "c.c. state" structure.

3) Per-Packet and Per-Interval Computation: Following the algorithms described by Katabi et al [1], the XCP router implements two per-packet processing functions: `xcp_do_arrival()` updates the running traffic statistics when the packet arrives, and `xcp_do_departure()` calculates the individual feedback when the packet departs the queue. In addition, the XCP router must do per-interval processing to calculate the aggregate feedback, feedback distribution factors, and the length of the next interval. Usually, this should be done at the the end of a control interval, but it will require a variable-interval kernel timer. To avoid managing a high resolution timer in Linux kernel, we adopted a delayed computation approach – the end-of-interval computations take place when either of the two per-packet processing functions is called next. The advantage is that kernel timers were avoided and the XCP engine is completely packet-driven.

E. Integer Arithmetic and Scalability Issue

The computation in the XCP engine involves several real number multiplications and divisions, as well as summations over many packets. However, many kernels support limited integer arithmetic only. For example, 32-bit Linux has no double long arithmetic or floating point operations in kernel mode. We must therefore carefully design the data types to fit XCP feedback calculations into a 32-bit space. This unfortunately may limit the scalability, precision, and granularity in the XCP control. Below, we present an analysis of the impact of these arithmetic limitations on XCP's scalability.

1) Scaling Analysis: To implement all calculations in integer arithmetic, we must first determine the right unit, or scale, for each variable used in the XCP computation. By scale, we mean the range of values that a variable can represent as a function of the size of the variable (in terms of bits). That is, if a variable is size s, the range of values that it can represent after scaling is $[c_1 + c_2, (2^s - 1) \cdot c_1 + c_2]$, where c_1 and c_2 are the scaling factors: c_1 for granularity and c_2 for offset. For example, to represent a range from 1ms to 65sec round trip delay with 1ms granularity, we need a 16-bit variable using a c_1 of 1ms and c_2 of zero.

We start by taking as input the scales of XCP operational environment parameters – the number of flows ($\#flow$), bandwidth (bw), round trip time (rtt), and message transfer unit (MTU) (mtu). The following table lists these parameters and gives each a reasonable range that we think an XCP implementation should support.

Parameter	Corresponding variable and its typical range		
x (bits)	$\#flows$:	1, ..., 1M	($x = 20$)
y (bits)	bw:	1KBps, ..., 1TBps	($y = 30$)
z (bits)	rtt:	1ms, ..., 65535ms	($z = 16$)
t (bits)	mtu:	512, ..., 8000 bytes	($t = 4$)

We then estimate the scales for all other variables used in XCP calculations relative to these environmental parameters. We take into account how they are calculated, and their ranges and granularities. For example, the size for $cwnd_i$ (cwnd in flow i) should be $y + z - 9$, because its value converges to $bw_i \times rtt_i / mtu_i$, which can range from 0 to 1TBps×65s/512 with a granularity of 1 (packet). The following table lists such estimation for other XCP variables (due to space limitation, please refer to Katabi et al [1] for the meaning of each variable).

variable	range or approximation	est. scale
input_traffic	0 ... $bw \times rtt$	$y + z$
Queue	0 ... $bw \times rtt$	$y + z$
$cwnd_i$	0 ... $bw_i \times rtt_i / 512$	$y + z - 9$
$rtt_i / cwnd_i$	mtu_i / bw_i	$y + t$
$rtt_i^2 / cwnd_i$	$mtu_i \times rtt_i / bw_i$	$y + z + t$
sum_rtt_by_cwnd	$rtt_i / cwnd_i$ ... $\sum_i rtt_i$	$x + y + z$
sum_rtt2_by_cwnd	$rtt_i^2 / cwnd_i$ ... $\sum_i rtt_i^2$	$x + y + 2z$
ξ_p	bw/sum_rtt_by_cwnd	$x + 2y + z$
ξ_n	bw / rtt/input_traffic	$2y + 2z$

Based on this analysis, we are able to choose the scale and hence the data type for each variable based on the range of networking environments that we hope to support. For example, if we are to support the range given before the previous paragraph, we will need triple-long (96-bit) integer arithmetic. Of course, this assumes the extreme scenario of 1 million flows passing through the same router, some having 1ms RTT and 1TBps bandwidth while some others having 64s

RTT and only 1KBps bandwidth. If we are willing to give up the range or the precision (granularity), we can use a shorter integer, such as a double-long (64-bit).

Since the 32-bit Linux kernel does not even have native support for double-long operations, we designed a special data type called `shiftint` to accommodate the wide range of scales. It consists of a 32-bit integer to store the most significant bits, a short integer to store the scaling factor, and another short integer to store the shifting factor. Integers of any scale can be shifted and stored in a variable of this data type. Much of the XCP algorithm is implemented as operations on this data type. The trade-off is that we will lose precision in some calculations. A simulation study that compared the `shiftint` results with floating point operations puts the precision at $\pm 0.001\%$. The advantage is that we don't need to worry about manually scaling each variable as the scaling factor component automatically keeps track of the change.

2) Feedback Fractions: In Linux and many other TCP stacks, the unit of change in cwnd is a whole packet. Therefore, one would have easily used an integer type for `H_feedback`. However, the following analysis contradicts this intuition and shows that if `H_feedback` is measured in packets we must keep the fractional part and not round it off.

Consider a single flow with round trip delay rtt and bandwidth bw. Under XCP, its cwnd would converge at $cwnd = rtt \times bw/mtu$ packets. That is, during one XCP control interval, the router will encounter $cwnd$ packets from this flow. Now, let's assume that the available bandwidth for this flow has changed from bw to $(1+\Delta) \cdot bw$. If everything else is equal and unchanged, the individual feedback according to the XCP algorithm [1] will be

$$H\_feedback = \frac{h + max(\phi, 0)}{rtt \cdot \sum \frac{rtt \cdot s}{cwnd}} \cdot \frac{rtt^2}{cwnd} - \frac{h + max(-\phi, 0)}{rtt \cdot y} \cdot rtt$$

packets, where $h = max(0, \gamma \cdot y - |\phi|)$, $\phi = \alpha \cdot rtt \cdot (1+\Delta) \cdot S - \beta \cdot Q$, and $S = bw - y/rtt$. Considering that, at previous convergence, $\phi' = \alpha \cdot rtt \cdot S - \beta \cdot Q = 0$ and that input traffic y should equal $cwnd \cdot mtu$, we can deduce that $H\_feedback$ is $\alpha \cdot \Delta$ (packets).

If we look at this intuitively, when the bandwidth changes by a factor of Δ, the sender should match with a cwnd change by a factor of $\alpha \cdot \Delta$ (where $\alpha = 0.4$ is the stability constant [1]). This amount will be divided into small individual feedback shares among all packets during a control interval (one RTT). So if we only use integers to represent `H_feedback` and if the individual feedback share is small ($|\alpha \cdot \Delta| < 0.5$), we will lose all the individual feedback to rounding. And we will also lose the cumulative feedback since the sender can only accumulate cwnd changes by adding individual feedbacks together.

The above analysis justifies the mantissa-exponent data format as described in Section II-B. We further back this up with an experiment that compares XCP performance using this format and using an integer format to store feedback. The experiment setup and procedure are those described in Section III-A. The result (Fig. 3) shows that XCP sustains higher performance with the mantissa-exponent format.

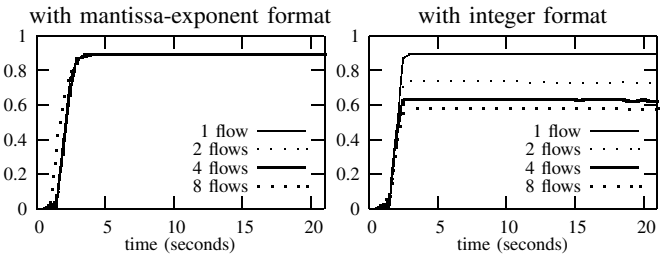


Fig. 3. Bandwidth utilization (total goodput of all flows as a ratio of raw bandwidth) comparison when 1, 2, 4, or 8 flows share the same bottleneck.

III. Experimental Evaluation

A. Experimental Setup

Our experimental study has been conducted in a real testbed illustrated in Fig. 4. XCP end-system code runs at end-hosts S1...S4 and D. XCP router code runs at router R. The end-to-end latency is emulated by placing a dummynet [8] host between R and D. The bottleneck link is between R and D, since the bandwidth between sources S and R is higher, and the transmission queue will build up at R's network interface to D. XCP router code operates on that interface to regulate flow throughput.

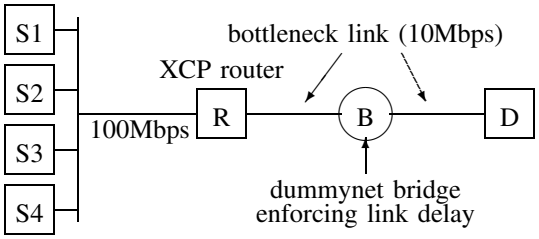


Fig. 4. Experimental network configuration

The round trip delay was set at 500ms and the bottleneck link was 10Mbps full-duplex, unless specified otherwise. Flows started at sources S1...S4 and terminated at D. We followed the TCP performance tuning guides (such as [9]) to set optimal values for TCP parameters. We used large buffer sizes and set the TCP window scaling option so that the connection could sustain the throughput in this large bandwidth-delay-product network. The maximum router queue size was set to twice that product, as is a common assumption for Internet networking.

We measured and compared the flow performance in each experiment. We were able to extract the following performance data from the traces that we collected during the experiments:

- *Bandwidth utilization* – the ratio (between 0 and 1) of total flow goodput to the raw bottleneck bandwidth, measured every RTT as time progresses. We expect to see a very high ratio (close to 1) if XCP is efficient.

- *Per-flow utilization* – the ratio of each flow's goodput to the raw bottleneck bandwidth. We expect to see the same ratio among all flows if XCP is fair.
- *cwnd value* – the progress of XCP sender's cwnd size (in packets), sampled by recording XCP packet's `H_cwnd` field. We expect to see a flat plot if XCP converges.
- *Router queue length* – the standing queue size at the bottleneck link, sampled whenever a packet enters the queue at router R. We expect to see a near empty queue if XCP is stable.
- *Packet drops* – the number of queue drops at bottleneck link, recorded every RTT at router R. We don't expect to see any drops if XCP is effective.

In the rest of this paper, we will frequently show the performance data as a function of elapsed time (seconds), unless marked otherwise.

B. Validation Experiments

The purpose of this set of experiments was to validate our XCP implementation and to validate the previously published simulation results on the XCP protocol. The XCP paper by Katabi [1] presented extensive packet-level simulations and showed that XCP achieves fair bandwidth allocation, high utilization, small standing queue size, and near-zero packet drops. We were able to arrive at similar conclusion through controlled experiments on a real network with our XCP implementation.

The first experiment compares XCP performance with TCP performance under FIFO and RED [10] queuing disciplines. We started four flows at the same time, one from each source S, to D. For the XCP test case, the flows were XCP flows (TCP with XCP options); otherwise they were normal TCP flows. For the RED test case, router R was configured with RED queue management with the drop probability set to 0.1 and with the minimum and maximum thresholds set to one third and two thirds of the buffer size, respectively.

Fig. 5 plots the performance measurement results, illustrating that XCP had the best and most stable performance, in terms of better bandwidth utilization, smaller queue buildups, and zero packet drops. The per-flow utilization charts further illustrate that XCP flows share the bandwidth equally and stably, but both TCP cases experienced significant fluctuations among flows. The per-flow cwnd chart also reveals that XCP flows can quickly converge to an optimal cwnd value.

We further studied XCP fairness toward flows that do not start at the same time or that have different RTTs but share the same bottleneck link – a well-known limitation of TCP congestion avoidance. In the next experiment, the setting was the same except that the first flow started at time zero and each additional flow started 30 seconds thereafter. Each flow lasted approximately 130 seconds. In the third experiment, we modified the dummynet configuration so that the delay was 50ms between S1 and D, 100ms between S2 and D, 250ms between S3 and D, and 500ms between S4 and D. That is, we changed the four flow RTTs to 100ms, 200ms, 500ms, and 1000ms, respectively. Fig. 6 shows that XCP exhibited

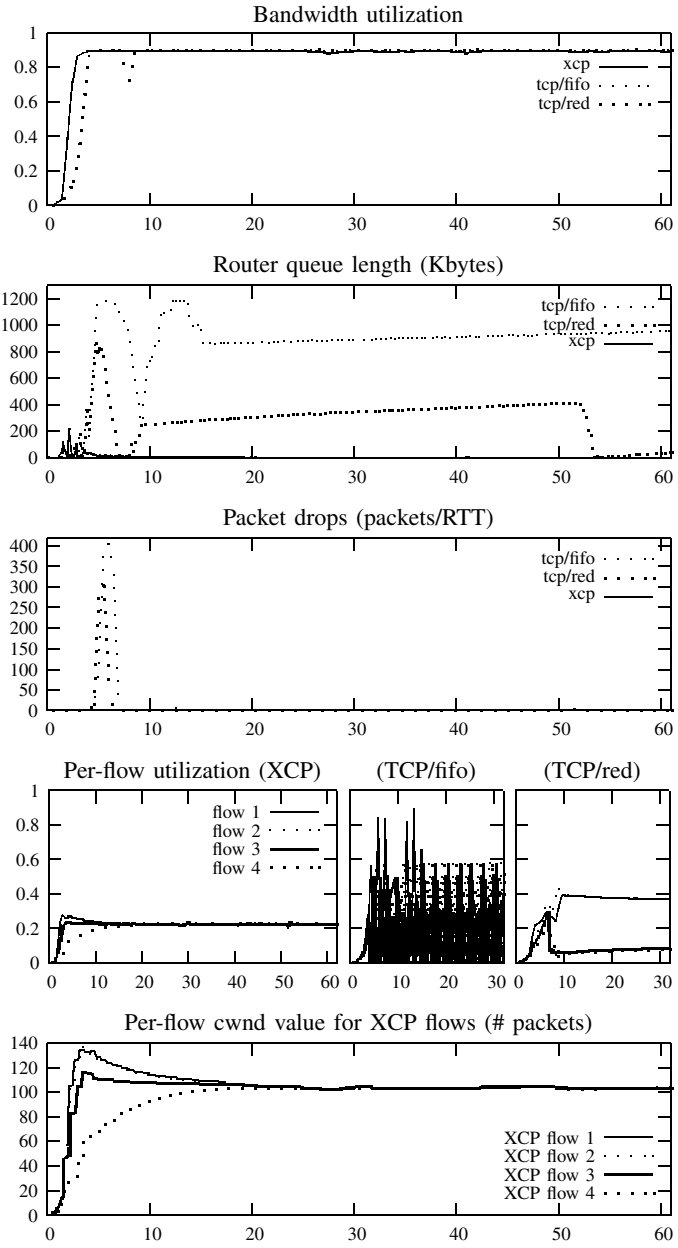


Fig. 5. Performance comparison between XCP and TCP.

fairness in both experiments. In summary, under controlled settings, we were able to demonstrate very good performance with XCP.

C. Fine-tuning the Link Capacity

To accurately calculate feedback, the XCP router must know the precise link capacity in advance. It must factor in proper framing and MAC overhead in the raw bandwidth estimate given in the `tc` command. This is important because if the XCP router overestimates the overhead, it will under-utilize the link and lower the performance. Likewise, if the router underestimates overhead, it will over-promise capacity, inflate feedback, and drive up its queue. Unfortunately, this overhead

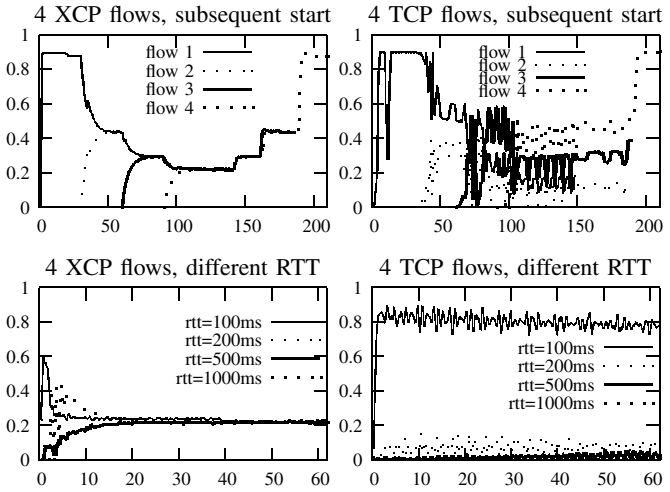


Fig. 6. Per-flow utilizations showing fairness between XCP flows (but not between TCP flows) with different start times or different RTTs.

cannot always be easily predicted because it varies by datalink format and sizes of actual data packets.

We have taken an empirical approach to estimate this overhead. In the same validation experiment setup, we varied the number of bytes to add as a per-packet overhead estimate and the packet sizes (through the MTU setting). We then compare the link utilizations and queue lengths. The result (Fig. 7) shows that 90-bytes per frame is a good estimate to balance both performance metrics – optimal bandwidth utilization and minimal queue buildup.

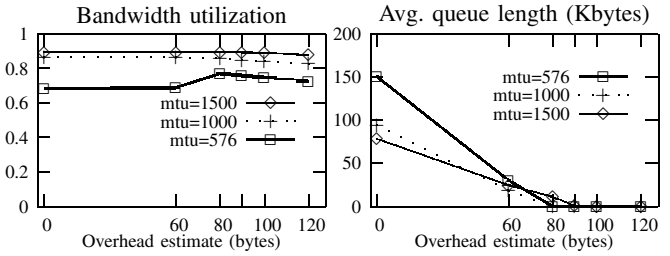


Fig. 7. Per-packet overhead estimates

In our experiments, we included this per-packet offset in the XCP router's link capacity calculation. We note that this value is from experiments using 10Mbps full-duplex Ethernet. The number may be different for other types of links, but the same approach can be followed to arrive at the best estimation.

IV. Sensitivity Study

XCP's control law requires the use of correct information. However, when XCP is deployed and integrated with a real network, unforeseen environmental factors like system configuration and network conditions may affect the accuracy of XCP's control information. We have given one such example in the previous subsection of how XCP should make proper assumptions of underlying link framing overhead. There can be additional and more subtle operational sensitivities. In this section, we look in more detail at other protocol sensitivity issues under the following four operational conditions:

- *TCP/IP parameter configuration.* Linux and many other operating systems provide mechanisms for the user to tune TCP/IP stack parameters, such as kernel buffer size, receiver's kernel buffer size, sender's socket buffer size, and receiver's socket buffer size. These memory allocations can affect XCP's performance because the throughput is limited not only by cwnd value but also by the buffer space available at both sender and receiver.
- *Link sharing.* Not all links in the Internet are point-to-point or have deterministic amounts of bandwidth. There are many contention-based multi-access links such as shared Ethernet and IEEE 802.11, and the true link capacity may be difficult or impossible to obtain. This can affect XCP feedback calculations.
- *Wireless networks.* Wireless networks almost always have the same link-sharing problem because the media is frequently shared multi-access and interference-prone. In addition, wireless networks often have non-congestion losses due to imperfect channel condition, interference, mobility, terrain effects, etc.
- *Hybrid networks*, where not all queues are XCP-capable. XCP is designed assuming an all-XCP network, but it will have to co-exist with non-XCP queues if it is to be incrementally deployed. Further, many link-layer devices (such as switches) have embedded queues for better traffic control and XCP will need to handle this type of hybrid network as well.

A. Sensitivity to TCP/IP Parameter Configuration

In XCP, the sender should include its current cwnd value in the XCP congestion header. For convenience of discussion, we call the value that the XCP sender puts in the `H_cwnd` field the *advertised* cwnd value. For the XCP control law to work properly, the routers must expect the sender to send the advertised amount during one RTT. However, if other factors limit XCP's sending rate, such as memory buffer shortage at the sender or receiver, the control law can be broken and XCP may not converge.

To prove this point, we repeated the above validation experiment with a single XCP flow. We first used the system default buffer size (64K) at the receiver. We then repeated the experiment with a larger value matching the bandwidth-delay-product (640K). Fig. 8 compares the bandwidth utilization of these two flows. Obviously, due to buffer limitation, the flow with small buffer size cannot fully utilize its cwnd to fully utilize the bandwidth.

If the sender fails to send as much as advertised, XCP routers will see the difference as spare bandwidth. Since XCP does not keep per-flow state to monitor the sending rate, when a router sees spare bandwidth, it will use positive cwnd feedback to increase a sender's allocation. This goes into a loop such that the XCP sender can keep increasing its cwnd value monotonically, well beyond the convergence point.

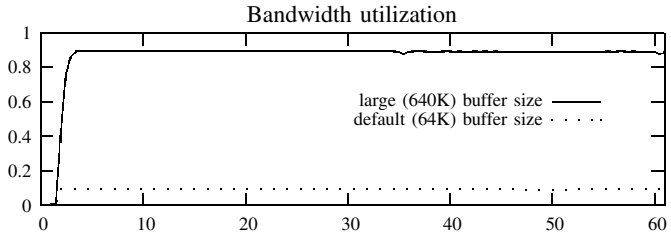


Fig. 8. XCP bandwidth utilization under different parameter tuning.

Fig. 9 shows such a negative effect. When we use large buffer sizes, the cwnd value has converged quickly to the optimal value around 400 packets. But when we use the default buffer size, the cwnd value increases linearly to well above 10,000 packets. Although the actual sending rate is still small due to buffer limitation, this limit may be transient and eased later. At that point, the congestion may become severe because the sender can suddenly send more than the network can handle under the inflated cwnd value.

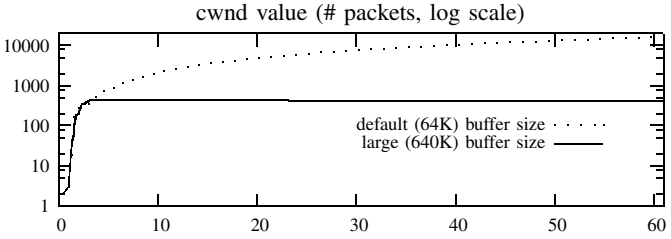


Fig. 9. XCP flow cwnd convergence under different parameter tuning.

One way to avoid this problem is for the XCP sender to advertise the true window limit instead of the cwnd face value. For example, if the additional limiting factor is the receiver's advertised window size, an XCP sender can put the lowest of either these limits or the current cwnd value as the advertised cwnd in `H_cwnd` field. The sender could also put the lowest limit in `H_feedback` field to set an upper bound on the positive feedback.

B. Sensitivity to Link Contention

All of the above experimental results are based on a point-to-point full-duplex Ethernet link (cross-over cable) at the bottleneck. However, if XCP operates in a multiple-access shared network, there will be cross traffic and media-access contention. Unfortunately, there is no easy way for one to predict and plan for such contention in calculating the true output capacity. XCP can only take link capacity at its face value or make a conservative estimate of how much capacity that it has fair access to. The damage of overestimation will be self-inflicted: XCP will generate inflated feedback, the senders will send more than the link can transfer, and the queue will build.

To demonstrate this, we repeated the above validation experiment with a 10Mbps Ethernet hub for the bottleneck link. We first set the hub to be half-duplex, in which case the XCP data packets had to compete with the ACKs for the link access. Then, we added another host to the same hub and dumped an additional 4Mbps of UDP traffic onto the link, creating media-access contention.

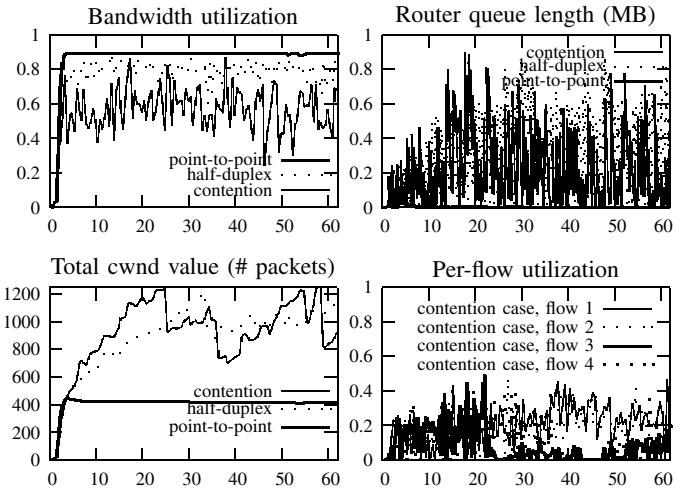


Fig. 10. XCP performances with different link environment.

The results (Figure 10) clearly show the limitations of the XCP algorithm when the link is half-duplex or is not contention-free. It is understandable that utilization is reduced because the link capacity is reduced, but since XCP does not know this, it will over-estimate the spare bandwidth and inflate the feedback. That is why the results show inflated cwnd values and significant queue buildups (compared to nearly zero queue length). If we look at per-flow utilization, we can see that individual XCP flows all have trouble converging (compared with Fig 5).

To remedy this deficiency, we need a new mechanism for the XCP router to detect link contention and to find out the proper capacity share, either by active measurement or if necessary by communicating with other routers that share this link.

C. Sensitivity to Non-congestion Loss

To study the effect of non-congestion losses on XCP, we conducted a set of experiments that injected losses artificially and randomly at a fixed probability, to emulate a lossy wireless channel. We injected packet losses at one of three locations: in the forward direction between the XCP sender and the XCP router, in the forward direction between the XCP router and the XCP receiver, and in the return direction. Since the XCP router does not process return-direction XCP options, it is unnecessary to make a distinction as to where to drop the return-direction XCP packets. We label these three different loss sources as "pre", "post", and "ack", and we varied them in different experiments to understand how to cope with different loss sources.

We also varied the loss probability (packet loss ratio), in different experiments. We chose ten different levels of loss ratios: 0.0001, 0.00025, 0.0005, 0.001, 0.0025, 0.005, 0.01, 0.025, 0.05, and 0.1. They cover a wide range of loss ratios

typically seen in a wireless network. In each experiment, we let XCP converge first, and then started the loss period after 15 seconds. The loss period lasted 60 seconds, and we measured the bandwidth utilization during this period. In addition to varying the loss source at "pre", "post", and "ack", we repeated all the experiments with the SACK option turned off, and included a TCP case (with SACK) as a comparison.

Fig. 11 compares the bandwidth utilization averages under all different settings. The label "nosack" in the legend denotes an XCP experiment with the SACK option turned off, and the data set "tcp" denotes the TCP case. Because the cwnd value converges around 400 in our network configuration, a loss ratio of 0.0025 corresponds to roughly 1 loss per RTT, and is marked by "1 loss/rtt" in the chart.

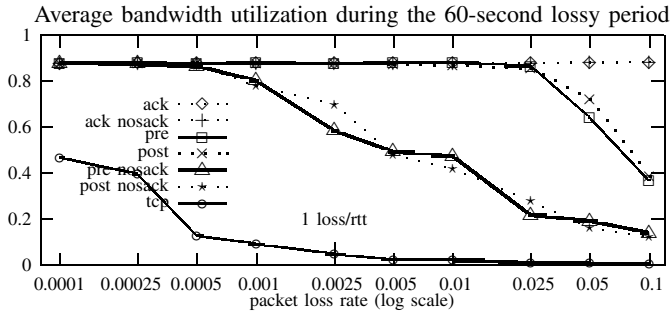


Fig. 11. XCP performance under non-congestion loss

The first clear observation is that XCP can handle non-congestion losses much better than TCP in all cases, if XCP makes the assumption that observed losses are not due to congestion. As indicated in many prior studies, packet losses on a high bandwidth-delay product path can substantially degrade TCP throughput because TCP cannot distinguish them from congestion loss. XCP, in this experiment, assumes all losses as non-congestion losses because it handles congestion separately through feedback.

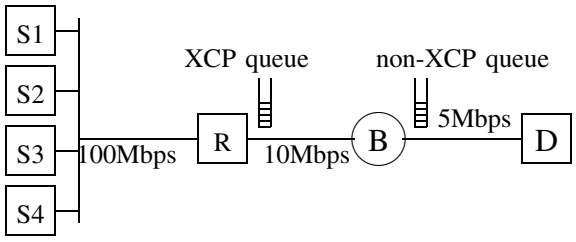
The next observation is that the loss of return-direction XCP packets (ACKs) does not have a noticeable impact across the whole range of loss rates, either with or without the SACK option. This is because ACKs are cumulative, so a loss of an ACK can be recovered by the subsequent ACK. Further, when XCP converges, the feedback carried in each ACK and the total feedback in a RTT is diminishing (see discussions in Section II-E.2); losing a small number of feedback packets (say, 10%) will not affect the throughput by much.

However, if forward-direction XCP packets are frequently lost, the impact on XCP performance can be significant because the lost segments must be retransmitted. Here, however, it makes a significant difference whether the SACK option is used or not. Without SACK, XCP's performance will suffer even with infrequent losses at a ratio as small as 0.001. But with SACK, XCP will not have noticeable degradation until the loss ratio is more than 25 times higher, at more than 0.025. And even at that high loss ratio, the degradation is much smaller with SACK than without SACK. This result validates our assertion earlier that it is very important for XCP-enabled TCP to include SACK to deal with non-congestion loss.

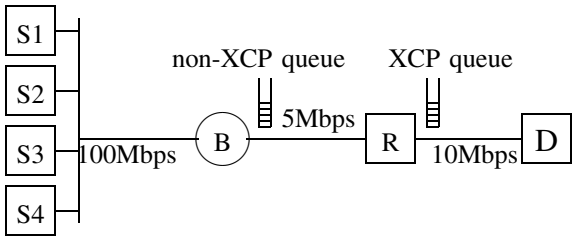
D. Sensitivity to Non-XCP Queues

We hypothesized that XCP will perform poorly in a hybrid network, worse than TCP, if the bottleneck is a non-XCP queue. Since the XCP router with the lowest per-flow link capacity in the path will dictate the sender's cwnd, if this capacity is still higher than the actual bottleneck link, the cwnd may still be high enough to cause congestion and packet losses. Unlike TCP which reacts to packet losses by reducing cwnd, XCP flows can only take commands from XCP feedback– they have no mechanism to react to this congestion.

To verify this hypothesis, we conducted two experiments: one put a tighter non-XCP queue after the XCP router and the other put it before (Fig. 12). In both cases, the non-XCP queue was a FIFO queue with a 100-packet queue limit and its output was limited to 5Mbps, half of the XCP link capacity at router R. The rest of the setup remained the same as in the previous experiments.



Case 1: non-XCP queue after an XCP queue

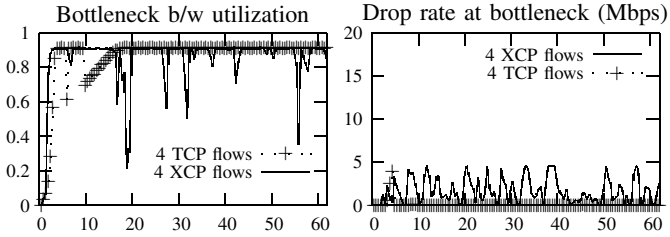


Case 2: non-XCP queue before an XCP queue

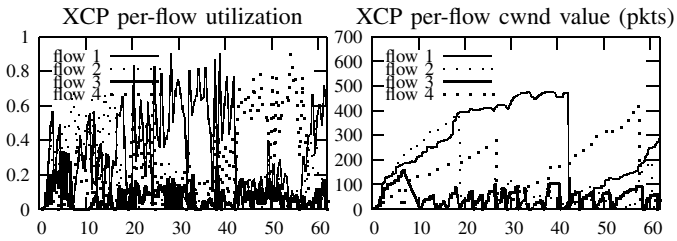
Fig. 12. Network configuration for hybrid network experiments

We measured utilization and packet drops as before but on the non-XCP queue because it was the new bottleneck. Results from the first experiment validate our hypothesis (see Fig. 13). Since XCP assumes that all losses are due to link impairments, it does not reduce its sending rate upon loss detection. As a result, XCP has lower utilization and much higher packet drops than with TCP. This is severe congestion on the bottleneck link and bandwidth waste elsewhere, as XCP senders transmit nearly 50% more than they should and XCP is not able to correct that. In addition, XCP flows fail to achieve fair bandwidth sharing, as shown by the per-flow utilization and cwnd charts in Fig. 13.

Results from the second experiment show a similar picture (see Fig. 14). The congestion is even worse this time – XCP

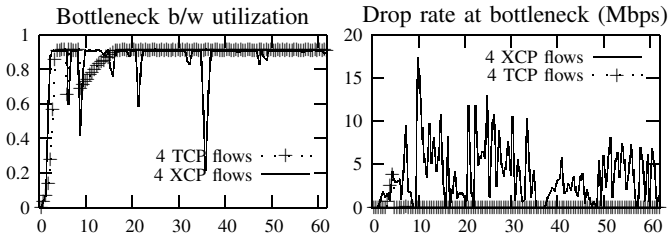


(a) Comparing XCP and TCP performance

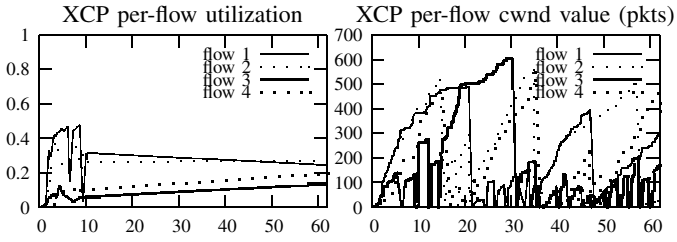


(b) XCP fails to converge

Fig. 13. Results of the non-XCP queue experiment (case 1)



(a) Comparing XCP and TCP performance



(b) XCP fails to converge

Fig. 14. Results of the non-XCP queue experiment (case 2)

senders over-transmit by nearly 100%. And as before, XCP fails to converge to a fair sharing state.

This study shows that XCP is incapable of dealing with the presence of a non-XCP queue at the bottleneck. To remedy this deficiency, we need a new mechanism for the XCP router to detect these queues and respond appropriately.

V. XCP Control Law Analysis

The above sensitivity study suggests that environmental factors, including system settings and network configuration, can have significant effects on the behavior of the XCP control law. To further understand this effect, we need to analyze the control law input and incorrect control loop scenarios.

A. Observations

The XCP feedback control operates on the following five input values: cwnd and RTT for each flow, link capacity, aggregated input traffic, and queue length. The first two are carried in each XCP packet and indirectly imply a flow's maximum rate. The third, link capacity, is a run-time configuration parameter. The last two, input traffic and queue length, are accurately measured at the XCP router itself.

This opens two possible ways for incorrect feedback calculations.

- *XCP can miscalculate the flow rate.* Since an XCP router doesn't keep per-flow information, it relies on the sender's advertised cwnd/RTT value. As we have seen above, the actual rate can be less than what the sender advertises if there are limiting factors at the end host (like low receiver-window or low buffer size) or in the network (like non-XCP queue). In this case, there is a mismatch between the perceived aggregated flow sending rate and the actual measured aggregated input traffic. The XCP router sees this as spare bandwidth and produces positive feedback to artificially drive the sender's cwnd over the stable value.
- *XCP can misestimate the link capacity.* XCP must know the true link capacity to produce correct feedback. However, as we have seen in many cases, the real output capacity can be dynamic and difficult to estimate. Link-layer factors like shared medium access and non-XCP queues can lower the capacity below the usual estimation (hardware-specific raw bandwidth). In this case, XCP will again over-estimate spare bandwidth and overly inflate the sender's cwnd.

In both cases, the network can be thrown into an unstable state, incurring more severe congestion than with TCP. Unfortunately, there is no mechanism within the existing XCP framework to remedy this situation. For XCP to be deployable, we must tighten the environment and remove these two possibilities, or we must find a fail-safe mechanism for XCP to detect and break out from a misbehaving control loop.

B. Control Theoretic Basis

We now give a control theoretic explanation of why capacity misestimation can lead to large fluctuations in XCP performance. In the original XCP paper [1], Katabi et al presented an XCP model as follows. Consider a single link of capacity c traversed by N XCP flows with a common round trip delay of d. Let $y(t)$ be the aggregate traffic rate at time t and $q(t)$ be the queue length. Using a fluid model, and ignoring boundary conditions, the XCP feedback mechanism can be expressed as a set of delay differential equations:

$$\dot{q}(t) = y(t) - c \tag{1}$$

$$\dot{y}(t) = -\frac{\alpha}{d}(y(t-d) - c) - \frac{\beta}{d^2}q(t-d) \tag{2}$$

where α and β are two control parameter constants.

Intuitively, the first equation implies that whenever the aggregate input traffic rate is different from the true capacity of the bottleneck, the excess traffic will accumulate in the persistent queue or any shortfall will be filled from the persistent queue. The second equation reflects how XCP calculates the aggregate feedback ($\phi = \alpha \cdot d \cdot (c - y) - \beta \cdot q$) that is used one round trip later to change the input traffic accordingly. With a variable change $x(t) = y(t) - c$, the above equations can yield a linear system independent of c:

$$\dot{q}(t) = x(t) \tag{3}$$

$$\dot{x}(t) = -K_1 x(t-d) - K_2 q(t-d) \tag{4}$$

where $K_1 = \alpha/d$ and $K_2 = \beta/d^2$ are constants. With simple control theory methods, this model can be proved stable [1].

However, this model has one oversight on the meaning of the variable c. If we look at how it was used in the above two delay differential equations (1,2), it actually has two different semantics. In the first equation (1), c is the actual *attainable* capacity – the rate at which an outgoing link can deliver XCP packets to the next hop. That is, the change of queue length depends on this actual capacity. In the second equation (2), c is the *estimated* capacity that is entered into the XCP router as a configuration parameter and used in the formula to calculate XCP feedback. To differentiate between these two, we should revise the above linear system as the follows:

$$\dot{q}(t) = x(t) + \epsilon(t) \tag{5}$$

$$\dot{x}(t) = -K_1 x(t-d) - K_2 q(t-d) \tag{6}$$

where $\epsilon(t)$ is the estimation error – the difference between the second c and the first c at time t.

This linear system model has other deficiencies as well. For example, this model assumes $q(t)$ has no bound but in practice $q(t)$ can never be negative or go above certain maximum queue limit. If this boundary condition is not considered, the system's steady state may require the ability to "buffer" unused bandwidth when the input traffic is below link capacity. This is obviously impractical in a real system as wasted bandwidth cannot be reclaimed later.

Coming up with a more accurate control system model for XCP and proving its properties is beyond the scope of this paper. We suffice to say here that the above revised linear system (5,6) will always fluctuate if ϵ is not zero, because it takes the form of a classic feedback control system with non-zero disturbance signal (Fig. 15). This explains what we have seen experimentally when XCP misestimated the link capacity.

VI. Deployment Issues

XCP faces a number of significant deployment challenges were it to be deployed on a wide scale. The most apparent impact is that it would require changes to the operating systems of the end hosts and also routers. With the rise of middleboxes such as firewalls and network address translators in the Internet, it has become increasingly difficult to deploy even new purely end-to-end protocols. XCP faces this and the following additional challenges.

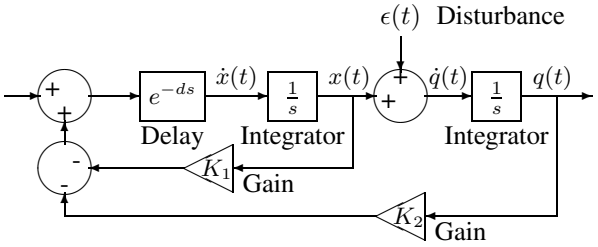


Fig. 15. XCP feedback loop with capacity estimation error

A. Incremental Deployment

As shown above in Section IV-D, XCP performance is sensitive to the presence of XCP-incapable queues along the path. This has serious deployment implications for upgrading existing networks or operating across heterogeneous networks. One potential solution is for the endpoint to switch to TCP if it detects a non-XCP queue across the end-to-end path, but there is no apparent way to do this reliably. Another idea is to detect any bottleneck non-XCP queue between two XCP routers and to calculate feedback based on an estimation of the queue's capacity or current level of congestion. This is also a non-trivial problem and further research is required.

B. XCP Responses to Lost Packets

An open issue is how XCP should respond to lost packets. Above, in Section IV-C, we showed that XCP would perform well in a lossy environment if the loss was not caused by congestion. However, if the loss was indeed due to operation over a congested non-XCP queue as described in Section IV-D, XCP would perform poorly. This is because XCP relies on its feedback loop to control congestion and considers all loss as rare events with no significance in congestion control.

An alternative, more conservative strategy would be to still follow TCP rules for cwnd reduction upon a loss, in case the loss was due to congestion. This however may not yield good performance in a truly lossy environment like wireless networks. Further, it has not been studied whether the XCP control laws can still hold with the addition of TCP responses due to packet losses.

This dilemma is difficult to solve. Studies have shown that it is often difficult to distinguish congestion loss from non-congestion loss in TCP. The same conclusion may apply to XCP.

C. XCP in the Internet architecture

As mentioned above, XCP could be deployed at different layers in the Internet architecture, including as a new transport protocol, as an existing transport protocol extension, as an interposed protocol layer between the IP and transport layers, or as an IP header option. Although we implemented XCP as a TCP option, for ease of experimental evaluation, it is not desirable to have to implement XCP extensions for every possible transport protocol, and routers will not likely want to have to parse different transport header formats. From an architectural standpoint, XCP is probably appropriate as

an interposed protocol layer, but there may be deployment issues with respect to middleboxes and firewalls that preclude that approach. If not coupled with a transport protocol, there are also issues relating to the piggybacking of XCP control information being returned to the XCP sender, without using additional packets.

As presently specified, XCP requires an additional 64 bits per data segment for the XCP header, plus potentially an extra 32 bits of XCP feedback header. The addition of an extra 64 or 96 bits per packet places additional load on the network, and can be particularly expensive for satellite links. Satellite and wireless links often use header compression, and any XCP approach should also be compatible with such compression. One possible avenue to explore is whether XCP headers are required on every packet or whether only some subset of the packets could carry an (aggregated) XCP header. This type of compression would likely be less robust to packet loss and may lead to a more complicated router algorithm.

D. Security considerations

XCP opens up another vector for denial-of-service attacks, because malicious hosts can potentially disrupt the operation of XCP by promising to send at one rate but sending at another, or by disrupting the flow of XCP information. Above in Section IV, we demonstrated the sensitivity of XCP's control algorithm to the use of incorrect information from well-intentioned hosts. When deploying XCP in public networks, it would seem that steps need to be taken to avoid malicious activity. This suggests that ingress nodes in each network area probably need to police the XCP flows to some degree, on a per-flow basis. Such a requirement might undermine one of the attractive features of XCP: its avoidance of requiring per-flow state in the network.

In addition, if it is defined above the IP layer, XCP would be incompatible with IPsec encryption, unless bypasses were defined to allow the XCP header to be copied over to the encrypted tunneled packet and back again to the plaintext side.

VII. Conclusion

This paper has reported on a study to implement XCP in the Linux kernel and to examine its performance in real network testbeds. The goal of this study was to identify issues important to XCP deployment and we have found several challenges. We first found that the implementation was challenging due to the lack of support for precision arithmetic in the Linux kernel, and that it was important to use a floating point data type in the XCP protocol header. When we studied the sensitivity of XCP to accurate reporting of the sending rate, to operation over contention-based media-access protocols, to non-congestion induced losses, and to incremental deployment, we found that XCP would have potentially significant performance problems unless misconfiguration, estimation errors, and XCP-induced congestion could be detected and prevented. Collectively, the experimental findings suggest the need for more refinement and extension, and careful consideration of its deployment impact on legacy protocols and infrastructure.

The results of this work, while obtained in a study of XCP, can apply to other router-based congestion controls such as AQM (Active Queue Management) schemes in general because an effective control often relies on accurate knowledge of the link capacity. These protocols, as well as XCP, will have difficulties operating in an environment in which the attainable capacity is not fixed. This includes cases that we have identified above, including links that are intrinsically variable, such as wireless networks in which the capacity is always affected by interference and propagation. Some are even multi-rate by design, such as 802.11b, which can dynamically adapt its rate between 1, 2, 5.5, and 11Mbps. All of these factors create estimation problems for feedback control. For future work, we plan to address this problem and develop techniques that can quickly and accurately estimate attainable capacity in such an environment.

Acknowledgments

The work described in this paper was supported by U.S. Army CECOM contract DAAB07-01-C-L845 (Program Manager: Gerald T. Michael) and also by Boeing IRAD and capital funds. The authors would like to acknowledge Sid Goel for his contributions in the floating point libraries and in the experiments, and Mohin Ahmed for helping us with the control theory and for coming up with the disturbance explanation. The authors would also like to thank Jae Kim and Debo Dutta for comments on the manuscript, and Aaron Falk and others on the ISI-XCP project for discussions during the course of this study.

References

[1] D. Katabi, M. Handley, and C. Rohrs, "Congestion control for high bandwidth-delay product networks," in *Proceedings of the ACM Conference on Communications Architectures and Protocols (SIGCOMM)*, Aug. 2002, pp. 89–102.

[2] D. Katabi, "Decoupling congestion control from the bandwidth allocation policy and its application to high bandwidth-delay product networks," Ph.D. dissertation, Massachusetts Institute of Technology, Cambridge, MA, Mar. 2003.

[3] G. Hains and F. Loulergue, "Preface: Special Issue on High-Level Parallel Programming and Applications," *Parallel Processing Letters*, Feb. 2004.

[4] T. Kelly, "Scalable tcp: Improving performance in highspeed wide area networks," *submitted for publication*, Dec. 2002.

[5] S. Floyd, "Highspeed tcp for large congestion windows," *IETF Internet Draft: draft-floyd-tcp-slowstart-01.txt*, Aug. 2002.

[6] C. Jin, D. Wei, and S. Low, "FAST TCP: motivation, architecture, algorithms, performance," *Proceedings of IEEE Infocom 2004*, Mar. 2004.

[7] L. Xu, K. Harfoush, and I. Rhee, "Binary increase congestion control for fast long-distance networks," *Proceedings of IEEE Infocom 2004*, Mar. 2004.

[8] L. Rizzo, "dummynet," `http://info.iet.unipi.it/~luigi/ip_dummynet/`.

[9] J. Mahdavi, "Enabling high performance data transfers on hosts," Dec. 1997, technical note, Pittsburgh Supercomputing Center. `http://www.psc.edu/networking/perf_tune.html`.

[10] S. Floyd and V. Jacobson, "Random early detection gateways for congestion avoidance," *IEEE/ACM Transactions on Networking*, vol. 1, no. 4, pp. 397–413, Aug. 1993.

[11] T. D. Dyer and R. V. Boppana, "A comparison of TCP performance over three routing protocols for mobile ad hoc networks," in *Proceedings of the 2001 ACM International Symposium on Mobile Ad Hoc Networking & Computing (MobiHoc'01)*, Oct. 2001.