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

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

Tom Henderson
Boeing Phantom Works, Seattle, Washington
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 do 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 identified two sources that can significantly reduce XCP's ability to control congestions and achieve fairness. Our contributions are two fold. 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 XCP protocol header. Second, through experiments and analysis we have identified several possibilities that XCP can enter into incorrect feedback control loops and adversely affect the performance. These are deployment challenges intrinsic to XCP design. More research is needed before we can advocate a wide scale adoption.

I. Introduction

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 allows the state to be kept 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 to each of their output queues. Based on their difference from 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 congestion headers. A more congested downstream router can further reduce this feedback by overwriting it. Ultimately, the packet will contain the feedback from the bottleneck along the path. When the feedback reaches the receiver, it is returned to the sender in an 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 round trip time 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 Science 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 end-to-end and can only indirectly infer the congestion state of routers along the path.

We have conducted an experimental study on XCP and its deployment. We have implemented XCP in Linux and conducted a comprehensive experimental 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 discovered that the choice of data type affects accuracy and the performance of XCP. Next, our experimental results revealed that XCP's performance can be adversely affected by environment factors including TCP/IP configuration, link sharing, non-congestion loss, and the presence of non-XCP queues. And finally, our analysis would show several such possibilities for XCP to lose its ability to control congestions 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 where XCP must avoid. Finally, we discuss a number of deployment issues in Section VI, followed by the conclusion 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 XCP, TCP is still the underlying transport mechanism that connects and delivers flow data. The effect of the XCP option is to modify TCP sender's cwnd (congestion window) 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 to legacy TCP stacks.

In such an architecture, whenever TCP sends a data segment, the XCP congestion header is encoded in a 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 an 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).

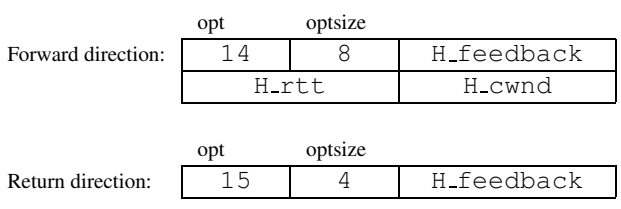

Fig. 1. XCP option formats in both directions

Fig. 1 illustrates the two formats. As a prototype, we simply pick two unused values (14 or 15) for TCP option without making any attempt to go through IETF standardization. Other than opt and optsize, the remaining three 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 a split mantissa-exponent representation that interprets the 16-bit `H_feedback` as the following:

- The most significant 13 bits is the signed mantissa (m)
- The remaining 3 bits is 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 16-bit, 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

C.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 purpose */

  __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 XCP receiver receives before passing back to the sender, and `feedback_rtn` is the amount that XCP sender finally receives. To support delayed ACK, feedback from several packets can be accumulated at both `feedback_rcv` and `feedback_rtn`. The same mantissa-exponent data type is used in these three variables.

We add steps in TCP's usual ACK processing to handle XCP options. Upon receiving a TCP packet with XCP option, it updates one of the above variables. At 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.

C.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 choose to preserve the TCP congestion control code but remove its effect on cwnd change, except in RTO case where cwnd is reset to one.

We do so with the `force_cwnd` variable in 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 disallows 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.

C.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 the 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 than that. We therefore modify the TCP receiver so that the advertised window grows by the integer value of `feedback_rcv`. This modification is very important, as we will see later (Section IV-A) that there is an adverse effect if XCP cannot utilize all of its cwnd window.

C.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 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 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 timestamp option and XCP option, we can now have at most 2 SACK blocks, compared to maximum 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 deployed not as a TCP option but as a separate protocol header, this point becomes 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.

D.1 Kernel Support and Interface

The Linux kernel has two mechanisms that will allow easy insertion and interface of the XCP engine: loadable module and device-independent queueing discipline layer. First, the entire XCP router function is 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 like FIFO, CBQ, RED, etc.; more elaborate ones can be implemented and added as loadable modules. The 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" is included to operate XCP with a standard drop-tail queue (fifo).

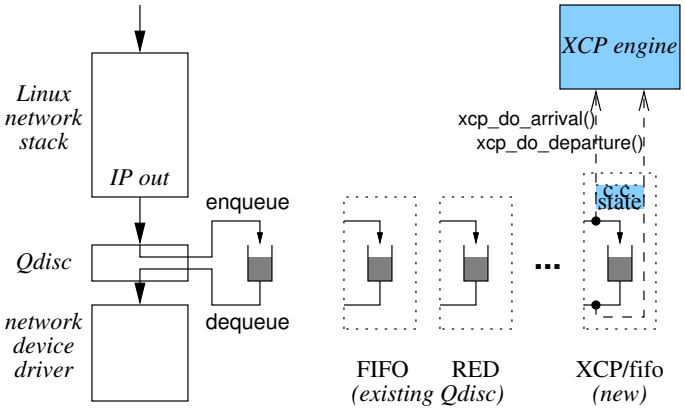

Fig. 2. Structure of XCP router module w.r.t. networking stack in Linux kernel

To manage the new XCP Qdisc, we have 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 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 has been

made to count for the framing overhead when XCP router calculates the true output link capacity.

D.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 skbuff structure) along with the "c.c. state" structure.

D.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 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 adopt a delayed computation approach – the end-of-interval computations will take place when either of the two per-packet processing functions is called next. The advantage is that we don't need to maintain kernel timers and the XCP engine is completely packet-driven.

E. Integer Arithmetic and Scalability Issue

The complex computation in 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 in 32-bit integer space. This unfortunately may limit the scalability, precision, and granularity in XCP control. Below, we present an analysis to study the impact of these arithmetic limitations on XCP's scalability.

E.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 and here c_1 is 1ms and c_2 is zero.

We start by taking as input the scales of XCP operation environment parameters – the number of flows ($\#flow$), bandwidth (bw), round trip time (rtt), and 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, 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 the granularity of 1 (packet). The following table lists such estimation for other XCP variables (due to space limitation, please refer to the XCP paper [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 environment 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 double-long (64-bit).

Since the 32-bit Linux kernel does not even have native support for double-long operations, we have to design 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 tradeoff 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.

E.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 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 (also a round trip time). 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 feedbacks to rounding. And we will also lose the accumulative 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 than with the alternative implementation.

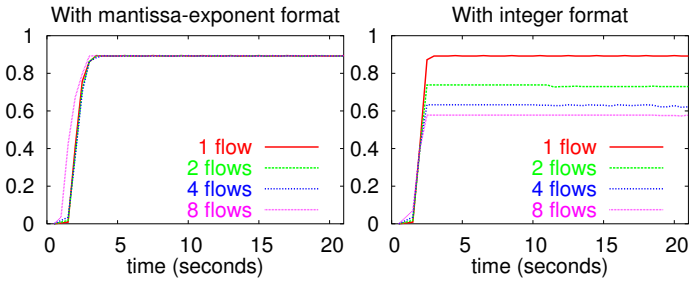

Fig. 3. Bandwidth utilization (total goodput of all flows as a ratio of raw bandwidth) comparison when 1, 2, 4, or 8 flows sharing the same bottleneck.

III. Experimental Evaluation

A. Experimental Setup

Our experimental study has been conducted in a real test network 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, whereas 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 will operate on that interface to regulate flow throughput.

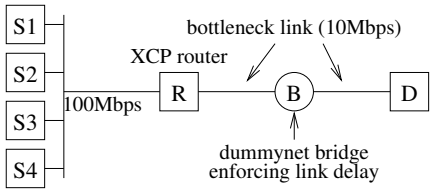

Fig. 4. Experimental network configuration

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

We measure and compare the flow performance in each experiment. We are able to extract the following performance data from the traces that we collect 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.
- *Per-flow utilization* – the ratio of each flow's goodput to the raw bottleneck bandwidth.
- *cwnd value* – the progress of XCP sender's cwnd size (in packets), sampled by recording XCP packet's `H_cwnd` field.
- *Router queue length* – the standing queue size at bottleneck link, sampled whenever a packet enters the queue at router R.
- *Packet drops* – the number of queue drops at bottleneck link, recorded every RTT at router R.

In rest of this paper, we will frequently show these performance data in time-progressive charts. Unless marked otherwise, the untitled horizontal axis is time progress in seconds.

B. Validation Experiments

The propose of this set of experiments is to validate our XCP implementation and to validate the previously published simulation results on the XCP protocol. The XCP paper by Katabi [1] presents extensive packet-level simulations and shows that XCP achieves fair bandwidth allocation, high utilization, small standing queue size, and near-zero packet drops. We are 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 start four flows at the same time, one from each source S, to D. For the XCP test case, the flows are XCP flows (TCP with XCP options); otherwise they are normal TCP flows. For the RED test case, router R is configured with RED queue management with the drop probability 0.1 and with the minimum and the maximum thresholds set to one third and two thirds the buffer size, respectively.

Fig. 5 plots the performance measurement results. We can see that XCP has 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

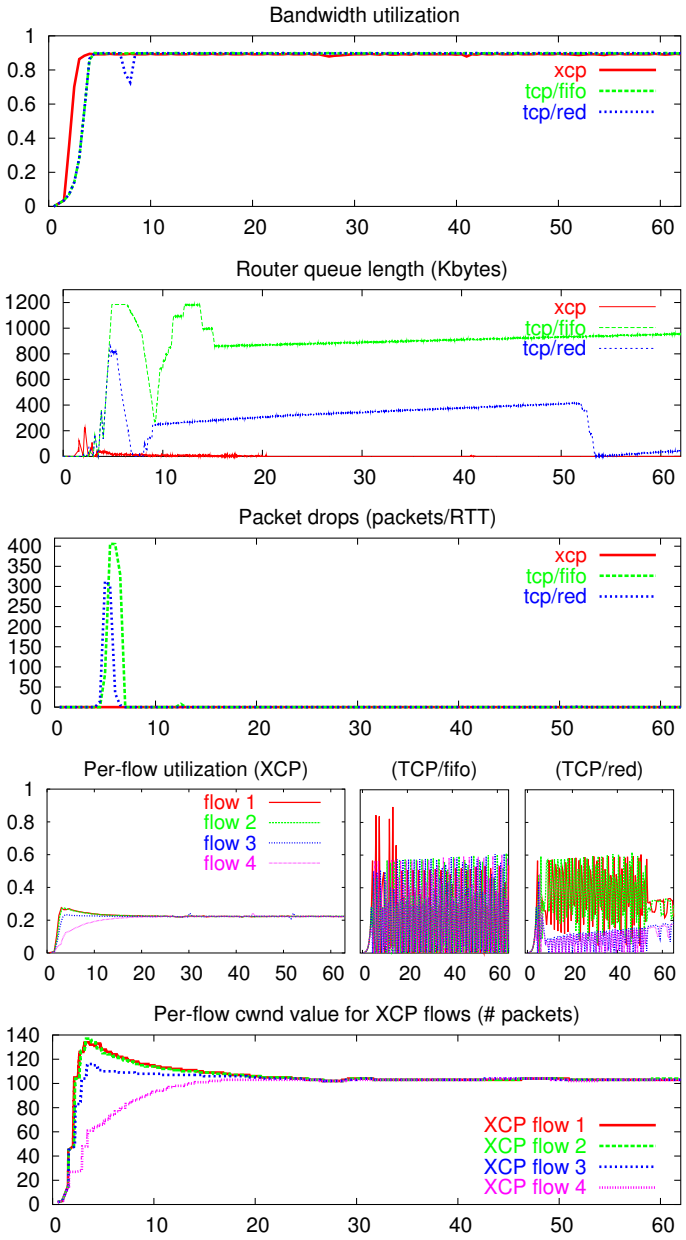

Fig. 5. Performance comparison between XCP and TCP.

that XCP flows share the bandwidth equally and stably, but both TCP cases experience significant fluctuations among flows. The per-flow cwnd chart also reveals that XCP flows can quickly converge to an optimal cwnd value.

We further study XCP fairness toward flows that do not start at the same time or have different RTTs but that share the bottleneck link – a well-known limitation of TCP congestion avoidance. In the next experiment, the setting is same except that the first flow starts at time zero and each additional flow starts at 30 seconds later. Each flow lasts approximately 130 seconds. In the third experiment, we modify the dummynet setting so that the delay is 50ms between S1 and D, 100ms between S2 and D, 250ms between S3 and D, and 500ms between S4 and D. That is, we change the four flow's RTTs to 100ms, 200ms, 500ms, and 1000ms, respectively. Fig. 6 shows that XCP exhibit fairness in both experiments. In summary, under controlled settings, we are able to demonstrate very good performance with XCP.

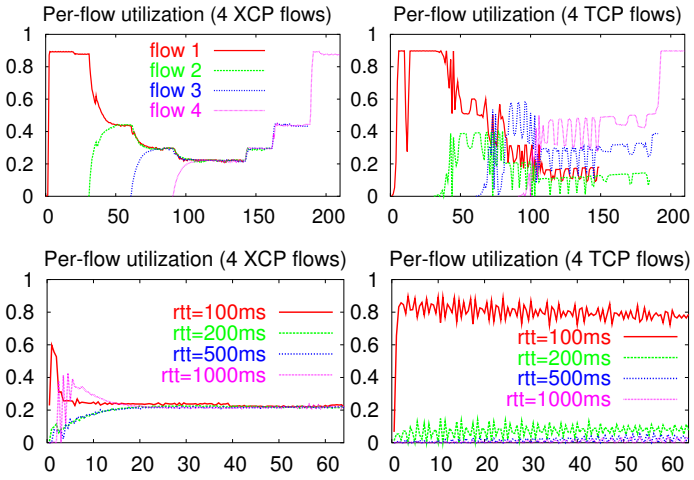

Fig. 6. Fair shares among 4 XCP flows with different start time or different RTT.

C. Fine-tuning the Link Capacity

To accurately calculate feedbacks, XCP router must know the precise link capacity in advance. It must factor in proper framing and MAC overhead in the raw bandwidth given in `tc` command. This is important because if we overestimate the overhead, we will under-utilize the link and lower the performance. Likewise, if we underestimate it, we will over-promise capacity, inflate feedbacks, and drive up the queue. Unfortunately, this overhead cannot 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 vary the number of bytes to add as per-packet overhead estimate and the packet sizes (through MTU setting). We then compare the link utilizations and queue lengths. The result (Fig. 7) shows that 90-bytes is a good estimate to balance both performance metrics – optimal bandwidth utilization and minimal queue buildup.

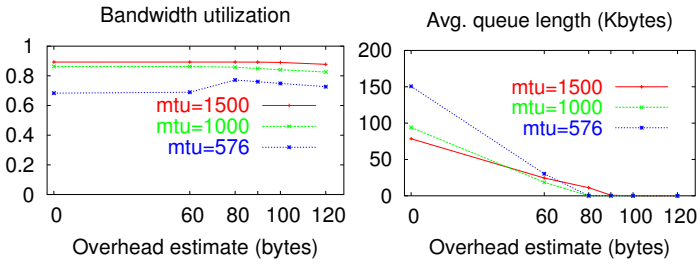

Fig. 7. Per-packet overhead estimates

We have included this estimation in 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 in a real system,

unforeseen environmental factors like system configurations and network connections may affect the accuracy of XCP's control information. We have given one such example in the previous subsection on how XCP should make proper assumption 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 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 Ethernet and IEEE 802.11, and the true link capacity may be difficult or impossible to obtain. This may affect XCP feedback calculations.
- *Wireless networks.* Wireless networks almost always have the same link-sharing problem because the media is shared multi-access and interference-prone. In addition, wireless networks often have frequent non-congestion losses due to imperfect channel condition, interference, mobility, terrain effects, etc. Wireless networks can also have temporary blockage or blackout period when the link experiences 100% loss for a short period of time.
- *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 in the Internet. 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 networks as well.

A. Sensitivity to TCP/IP Parameter Configuration

By XCP's control law, the XCP sender informs XCP routers of its current cwnd value and gets feedback for the new value to use during the next interval. For convenience of discussion, we call the value that XCP sender puts in the `H_cwnd` field the *advertised* cwnd value. For the control law to work properly, XCP routers must expect the sender to send the advertised amount during an 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 repeat the above validation experiment with a single XCP flow. We first use the system default buffer size (64K) at the receiver. We then repeat 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

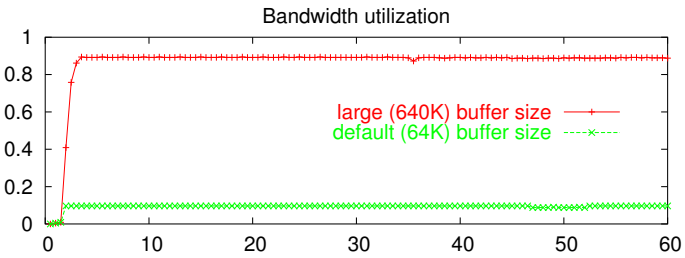

Fig. 8. XCP bandwidth utilization under different parameter tuning.

XCP sender can keep increasing its cwnd value monotonically, well beyond the convergence point.

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 removable when a possibly transient memory buffer shortage is 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.

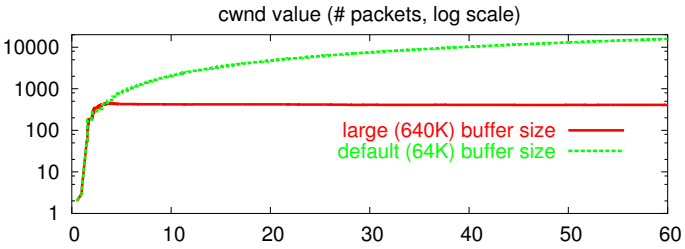

Fig. 9. XCP flow cwnd convergence under different parameter tuning.

One way to avoid this problem is for the XCP sender to advertise the true cwnd 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 the above experimental results are based 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. The damage will be self-inflicted: XCP will generate inflated feedbacks, the senders will send more than the link can transfer, and queue will build up.

To demonstrate this, we repeat 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 will have to compete with the ACKs for the link access. Then, we add another host to the same hub and dump an additional 4Mbps UDP traffic onto the link, creating media-access

contention.

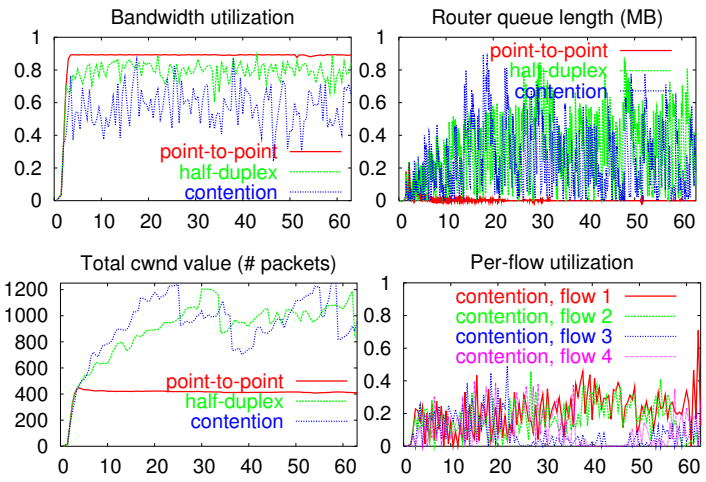

Fig. 10. XCP performances with different link environment.

The results (Figure 10) clearly fault 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 feedbacks. 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).

Unfortunately, there is not much one can do at the network layer to remedy this deficiency, unless we add MAC delays or conservative assumptions into the XCP control law.

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 the 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 on 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 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 SACK option turned off, and the data set "tcp" denotes the TCP case. Because the cwnd value converges around 400 in our network configuration, loss ratio 0.0025 is roughly 1 loss per RTT and is marked with "1 loss/rtt" in the chart.

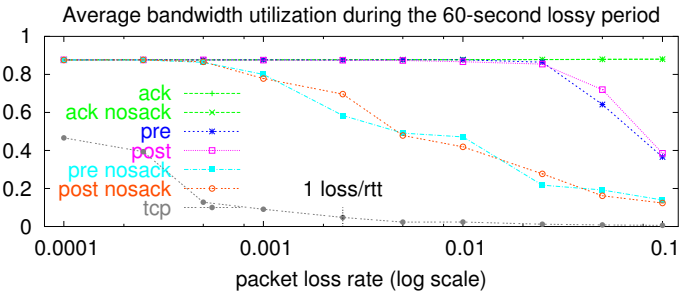

Fig. 11. XCP performance under non-congestion loss

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 it 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 in the whole range of loss rates, either with or without the SACK option. This is because ACKs are accumulative 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 to include SACK to deal with non-congestion loss.

D. Sensitivity to Blockage

Wireless networks can often have a temporary blockage or blackout period when the link experiences 100% packet loss for a short period of time. This can be caused by mobility when two nodes move out of range from each other, or by environmental effects like terrain or moving objects. It is obvious that no transport mechanism can communicate during the blockage period, but it is important to see how it can recover quickly to the previous state when the blockage ends.

In this experiment, we emulated blockage by setting firewall rules at the router to drop all packets during the blackout period. The blackout period started at 15 seconds after the flows start so as to let XCP first converge. We measured the bandwidth utilization and observed its behavior before, during, and after the blackout period. We repeated the measurement using 1, 2 or 4 XCP flows. The result of a 2-second blackout period is illustrated in Fig. 12. We also included 1, 2, and 4 TCP flows as a comparison.

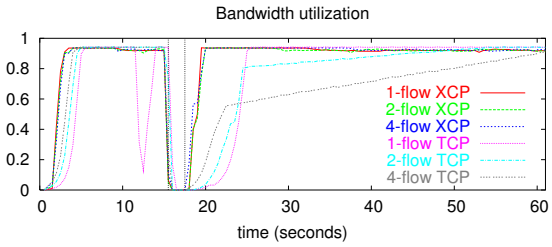

Fig. 12. XCP behavior around 2-second blockage (15th-17th second)

The result shows that XCP can climb out of a blockage quickly, as fast as during the start of a flow. In comparison, TCP is much slower as it must go through slow-start. Also, the number of flows does not make much of a difference in the XCP case.

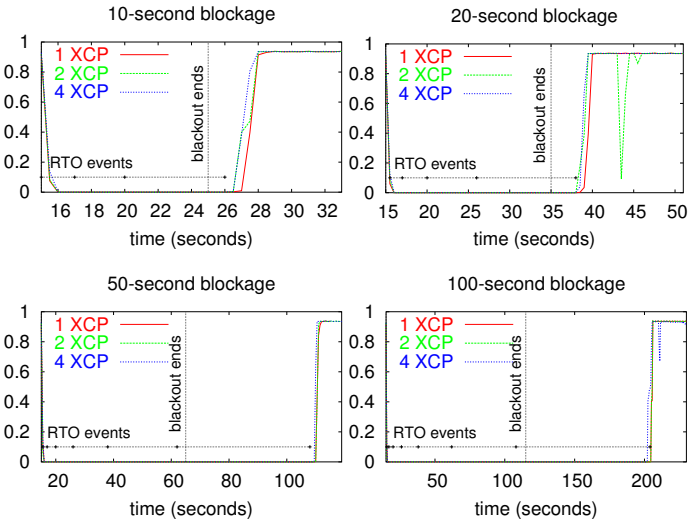

Fig. 13. XCP bandwidth utilization during and after blockage of various length

We further explored the blockage problem by varying the duration of blackout period, including 10, 20, 50, and 100 seconds. The results, in Fig. 13, show that there is a gap between the end of a blackout period and the start of XCP's climb when the blackout period is longer. A further study of the packet trace revealed that this is an effect of using our XCP implementation on top of TCP. Since XCP does not define its policy dealing with blockage, it inherits TCP's exponential back-off of its retransmit timer.

In TCP, when the blockage is much longer than round-trip time, a retransmission timeout (RTO) event will happen. Then the first unacknowledged packet will be retransmitted and the RTO value will double. To illustrates this, we plot the RTO expiration events along the bottom of each chart in Fig. 13. We can see that XCP will not recover until an RTO expires after the blackout period. The longer the blockage period lasts, the larger the RTO value will be, and the longer XCP may have to wait until it recovers.

Barring a notification from the network routers, XCP can only rely on retransmissions to learn when the blockage ends. One way to improve the inefficiency due to RTO growth is to replace the exponential back-off with a fixed RTO strategy similar to [11]. Since XCP handles congestion loss extremely well, consective RTO can only imply route failure or blockage. If RTO does not double, a retransmission event can happen much sooner after the blockage ends, and XCP can recover right after that.

E. 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 feedbacks – it has 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. 14). In both cases, the non-XCP queue is a fifo with a 100-packet queue limit and its output is limited to 5Mbps, half of the XCP link capacity at router R. Rest of the setup remains the same as the previous experiments.

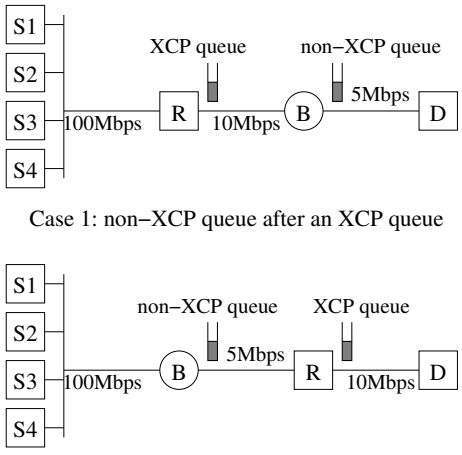

Fig. 14. Network configuration for hybrid network experiments

We measured utilization and packet drops as before but on the non-XCP queue because it is now the new bottleneck. Results from the first experiment validate our hypothesis (see Fig. 15). 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. 16.

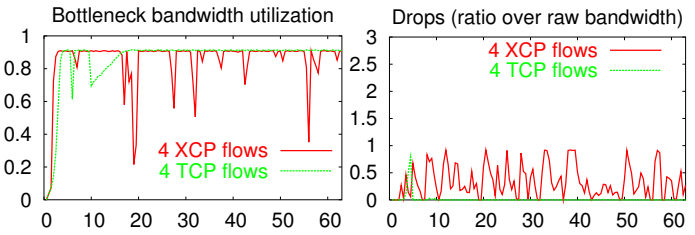

Fig. 15. XCP and TCP performance in the first experiment

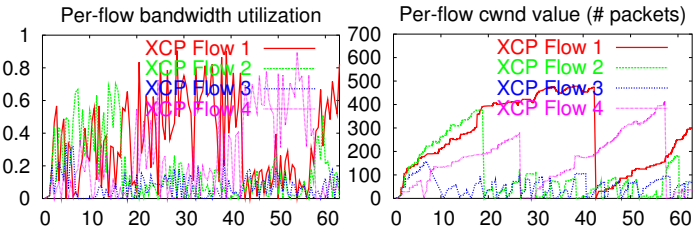

Fig. 16. XCP fails to converge (case 1)

Results from the second experiment show a similar picture (see Fig. 17). The congestion is even worse this time – XCP senders over-transmit by nearly 100%. And as before, XCP fails to converge to a fair sharing state (see Fig. 18).

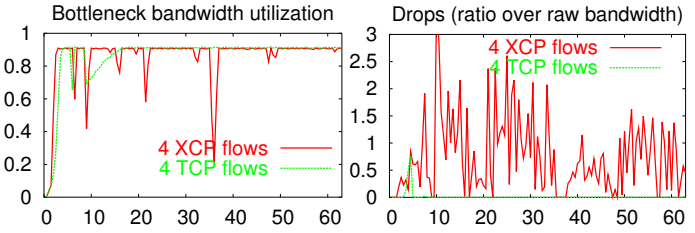

Fig. 17. XCP and TCP performance in the second experiment

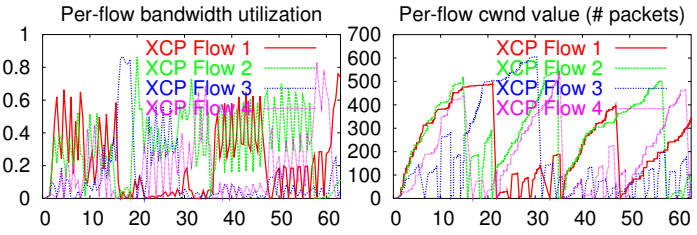

Fig. 18. XCP fails to converge (case 2)

This study shows that XCP is incapable of dealing with the presence of non-XCP queue if it becomes the bottleneck. This will have significant implications in XCP deployment and we will further discuss it in the next two sections.

V. Analysis of XCP Control Law Applicability

The above sensitivity study suggests that XCP must have accurate input for its control law to work properly. External factors, including OS setting and network configuration, can have significant effects on the behavior of XCP control law. To further understand this effect, we analyze the control law input and incorrect control loop scenarios.

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 are accurately measured at the XCP router itself.

This opens two possible ways for incorrect feedback calculations.

- *XCP can mis-calculate the flow rate.* Since an XCP router doesn't keep per-flow information, it relies on sender's advertised cwnd/RTT value. As we have seen above, the actual rate can be less than what the sender advertises if there is limiting factors at the end host (like low receiver-window or low buffer size) or at 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. XCP router sees this as spare bandwidth and produces positive feedbacks to artificially drive sender's cwnd over the stable value.
- *XCP can mis-estimate 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 acess and non-XCP queues can lower the capacity than the usual estimation (hardware-specific raw bandwidth). In this case, XCP will again over-estimate spare bandwidth and overly inflate 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.

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.

A. Incremental Deployment

As shown above in Section IV-E, XCP performance is sensitive to the absence of XCP-capable routers on the path. There is no apparent way for an XCP endpoint to reliably determine that it is running over XCP-capable queues across the end-to-end path.

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-E, 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 can holds with the addition of TCP congestion control AIMD rules.

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.

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 correct information. 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 likely undermines one of the attractive features of XCP: its avoidance of requiring per-flow state in the network.

In addition, XCP presently is incompatible with IPsec encryption, unless bypasses are 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 because of the lack of support for precision arithmetic in the Linux kernel. We also found that it is 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 has potentially significant performance problems unless mis-configuration, estimation errors, and XCP-induced congestion can be detected and prevented. We believe we are among the first to report such deployment challenges that XCP must overcome to see widespread deployment in IP-based networks. These deployment challenges are significant and appear to present a formidable barrier to acceptance of XCP unless additional XCP extensions are researched and developed.

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 experiments. 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] Dina Katabi, Mark Handley, and Charlie 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] Dina Katabi, *Decoupling Congestion Control from the Bandwidth Allocation Policy and its Application to High Bandwidth-Delay Product Networks*, Ph.D. thesis, 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] Luigi 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] Sally Floyd and Van Jacobson, "Random early detection gateways for congestion avoidance," *IEEE/ACM Transactions on Networking*, vol. 1, no. 4, pp. 397–413, Aug. 1993.

[11] Thomas D. Dyer and Rajendra 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.