Faculdade de Engenharia da Universidade do Porto

Implementation and performance evaluation of explicit congestion control algorithms

João Taveira Araújo

Master's Thesis in Electrical and Computer Engineering

Supervisor: Prof. Manuel Pereira Ricardo
INESC Porto Supervisor: Filipe Lameiro Abrantes

(President of the Jury)

March 2008

Abstract

As the Internet adapts to meet consumer demands for high performance in ubiquitous networking environments, congestion must be addressed using novel approaches to ensure system stability over increasingly unreliable and dynamic media. Our work focuses on implementing and studying the performance of two recent explicit congestion control protocols: XCP, eXplicit Control Protocol, and RCP, Rate Control Protocol.

This dissertation contains implementation details and subsequent test results of the XCP-b algorithm, a variant of XCP optimized for variable-capacity media, such as IEEE802.11. We show that, by inferring network capacity from queue dynamics, XCP-b proves more efficient than XCP when the link capacity is not explicitly known. Based on experimental feedback we present two alternative algorithms which further refine previous proposals.

Furthermore, this dissertation documents the implementation of RCP, for which no formal specification yet exists, using the existing XCP protocol as a framework, as well as comparing both protocols by emphasizing their benefits. We explore the core algorithm in an RCP system and the implications aggressively distributing bandwidth has on overall system stability. By quantifying conservative bounds within which such a system can withstand surges in network traffic, we demonstrate how to accurately predict system response to common arrival distributions.

Acknowledgements

Thanks are due to both my supervisors, Filipe Abrantes and Manuel Ricardo, for allowing me to roam freely for much of the time it took to complete this thesis. A special mention should also go out to Nuno Salta who shared the burden of hacking congestion control protocols into the BSD stack with me during our final year project, thereby pathing the way for much of my thesis. For his sins he is now enrolled as a PhD student.

Contents

List of Acronyms

AIMD Additive Increase, Multiplicative Decrease

API Application Programming Interface

BDP Bandwidth-Delay Product

BSD Berkeley Software Distribution

DCCP Datagram Congestion Control Protocol

DVB Digital Video Broadcasting

ECN Explicit Congestion Notification

FPGA Field-programmable Gate Array

ISI Information Sciences Institute, University of South California

IEEE Institute of Electrical and Electronics Engineers

IETF Internet Engineering Task Force

IP Internet Protocol

IPv4 Internet Protocol Version 4

IPv6 Internet Protocol Version 6

NTP Network Time Protocol

RCP Rate Control Protocol

RTT Round-trip Time

TCP Transmission Control Protocol

UDP User Datagram Protocol

UMTS Universal Mobile Telecommunications System

WLAN Wireless Local Area Network

XCP eXplicit Control Protocol

List of Figures

List of Tables

Auxiliary Listings

Chapter 1

Introduction

1.1 Overview

The core protocols that compose the Internet have remained relatively unaltered since their conception, composing a robust, scalable and dependable network that has since grown exponentially. As the Internet progressively distances itself from its humble beginnings, new requirements consistently challenge the existing network architecture, raising issues for which it was not originally designed and consequently cannot adequately handle.

Congestion control, on which this report will focus, is such a requirement. Initially overlooked, congestion control rapidly became a necessity as the Internet suffered a series of collapses in the mid 1980s [1]. The culprit was TCP [2], the reliable transport protocol over which most traffic is exchanged, and specifically the adopted retransmission policy. Rather than reducing transmission rates, TCP reacted to packet loss by doubling the data rate sent, further aggravating network congestion. This resulted in a network performing at full capacity but where no connection was making useful progress, endangering the very stability and scalability on which the Internet had thrived.

The threat of collapse was circumvented by embedding Van Jacobson's congestion control mechanisms in TCP . While this solution was far from ideal, its praticality is undeniable: backward compatible and incrementally deployable, it has managed to maintain the Internet stable for two decades of unprecedented growth. By being incorporated into TCP, these congestion mechanisms inherited one of TCP's most important design philosophies, the end-to-end principle [3], which advocates the shift of system complexity toward the end-host so as to achieve a dumb, cheap and efficient core network. As with the Internet however, the emergence of new

requirements raise issues which cannot be solved using the current congestion control scheme. Ever-higher bandwidth coupled with longer delays result in increasingly poor link utilization, slow adaptation to changing link loadings and significant congestion losses. Congestion is further exacerbated by a rise in demand for real-time multimedia applications which do not use TCP, therefore bypassing congestion control and generating an increasing amount of unresponsive traffic.

As pressure increases on adjusting congestion control to reflect these new expectations, multiple proposals have recently emerged with varying solutions, approaches and results. This report focuses on explicit congestion control and, specifically, protocols such as XCP [4] and RCP [5] that propose a new layer between the network and transport layers, providing an end-to-network platform which is used to exchange feedback information on congestion between end hosts and routers. While experimental results with both protocols present significant improvements over traditional congestion control schemes, there is still little understanding on how well such protocols would fare outside the environments within which they were developed.

1.2 Objectives

The objective of this thesis is to model, implement and enhance existing explicit congestion control algorithms so as to gain insight into theoretical and pratical limitations which may hinder widespread deployment. In particular, this thesis focuses on:

- **Shared-media access optimization** Both XCP and RCP share a feedback parameter which is calculated using the link capacity. In environments where this capacity is not known, such as IEEE 802.11 [6] and other wireless media, this algorithm results in reduced efficiency. Modifications to this behaviour, namely XCP-b[7], have been proposed and simulated, but are yet to be implemented. We propose both implementing and testing XCP-b on a real testbed, whilst simultaneously enhancing this algorithm based on experimental feedback.

- **Implementing RCP** At the time of writing RCP has no official implementation. Work carried out within this thesis included executing this task by extending the existing XCP implementation to support an experimental version of RCP, thereby taking advantage of the similarities between both explicit congestion control protocols.

– **Modelling RCP behaviour** The RCP algorithm achieves short flow completion times at the expense of queueing. By over-allocating bandwidth to new flows, RCP may be vulnerable when confronted with a significant and prolonged increase in flows. This thesis attempts to quantify the bounds within which RCP performs in a stable manner.

1.3 Document structure

This dissertation is organized as follows: chapter 2 discusses the nature of explicit congestion control and gives background on both XCP and RCP, two proposed congestion control protocols. The following chapters document the execution of our proposed objectives and the associated results. Chapter 3 describes the implementation of XCP-b and subsequent tests while chapter 4 explores further improvements to XCP-b based on initial experimental results. Chapters 5 and 6 relate to RCP: the former describes the complete implementation of the protocol over the existing XCP source code, while the latter studies the effect of flash crowds on queue stability. In chapter 7 we draw conclusions on the obtained results, as well as briefly discussing future work and trends in congestion control.

Chapter 2

Technical Background

2.1 Explicit congestion control

Introducing congestion control mechanisms at the transport layer allowed for both backward compatibility and incremental deployment, but its use was restricted to TCP flows. Although this approach made sense in a network where TCP connections were dominant and generated most traffic, this assumption can no longer be made. As demand for real-time distribution, particularly in streaming multimedia, rises, other transport protocols become increasingly responsible for network congestion. Addressing this issue is not trivial, resulting in different approaches. Transport protocols such as DCCP [8] provide session and congestion control without reliability, thereby positioning themselves as potential replacements for UDP [9]. Alternatively, congestion control protocols operating between the network and transport layer, such as XCP and RCP, are increasingly being seen as means of effectively managing all transport protocols equally and providing a cohesive solution.

Furthermore, the Internet as a whole is evolving towards ever higher bandwidth links, whilst simultaneously bearing witness to higher latency as wireless links become commonplace. The resulting high bandwidth-delay product of these emerging networks strain current congestion control mechanisms embedded in TCP.

By causing connections to "back off" once congestion is detected, TCP has become a cornerstone in the stability of today's Internet. With no explicit feedback from the underlying network however, TCP is limited to inferring congestion from packet loss. This renders TCP particularly inefficient over lossy mediums, which will become ubiquitous with the widespread deployment of wireless technologies such as WLAN, UMTS,

DVB and Bluetooth, as congestion is not the sole cause for packet loss. Additionally, once loss is detected TCP responds conservatively, since treating congestion as a binary value forces TCP to react without knowledge of the extent of congestion. The feedback control algorithm used, AIMD (additive increase, multiplicative decrease), probes for bandwidth through the linear increase of the congestion window, which is then exponentially reduced when loss occurs. Clearly, such a congestion avoidance algorithm is not suited for high-bandwidth medium: the increase policy is conservative whilst the decrease policy is aggressive, resulting in sub-optimal link utilization.

Future developments in congestion avoidance require explicit feedback to efficiently adapt the connection throughput to the underlying network link capacity. While protocol extensions such as ECN [10] use feedback to reduce packet loss by providing advance notification of congestion, protocols such as XCP and RCP use feedback to explicitly modulate transfer rates in order to avoid congestion. Both protocols offer significant advantages over Van Jacobson congestion control:

- Flow rates are more stable, particularly for long RTTs, providing better service for streaming applications;
- Queues are minimised, thereby reducing network latency;
- Rapid convergence to fair allocation;
- Improved bottleneck link utilization;
- Dynamic behaviour proportional to bandwidth-delay product rather than using fixed additive increase.

XCP and RCP will be discussed in more detail over the next two sections.

2.2 XCP - eXplicit Control Protocol

2.2.1 Protocol overview

The eXplicit Control Protocol (XCP) achieves maximum link utilization without resorting to packet loss through the inclusion of per-flow congestion state in packets. While senders specify their desired increase in throughput in outgoing packets, routers adjust this value to reflect both fairness and available bandwidth. Upon reaching the receiver, the resulting throughput

is returned as feedback, with the sender adjusting his congestion window accordingly.

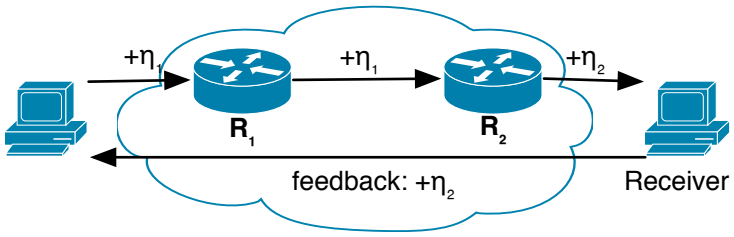


Figure 2.1: XCP flow path with corresponding feedback.

Figure 2.1 illustrates an overview of an XCP system. A sender requests an increase of η_1 to the current congestion window, signalling the request in the XCP congestion header. Router R_1 analyses and forwards the packet to R_2 without modifying the header, since there is enough capacity to cope with an increase in the flow's throughput rate. R_2 however considers η_1 excessive, and modifies the congestion header, replacing η_1 with a lesser value, η_2, the maximum permitted throughput change for this particular flow. The receiver copies η_2 and returns it to the sender as feedback, who then proceeds to adjust his congestion window accordingly. In this case, R_2 is the bottleneck in the path.

2.2.2 Congestion header

The XCP header is located between network and transport layers. The current draft version of the corresponding header for IPv4 packets is shown in 2.2.

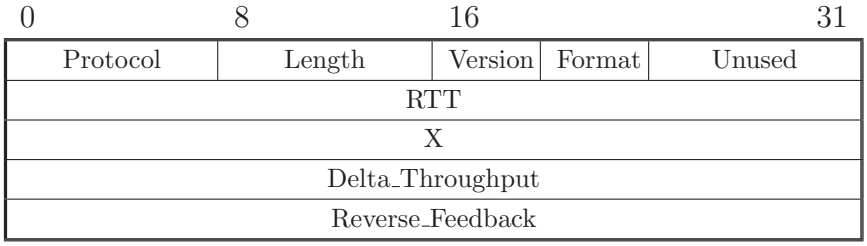


Figure 2.2: XCP congestion header, version 2.

The significance of each parameter is detailed in table 2.1.

Parameter	Definition
Protocol	Field indicating next-level protocol used in data payload.
Length	Header length in bytes.
Version	XCP version. Displayed format represents the second version of the congestion header.
Format	Indicates header format. Currently only the Standard Format and Minimal Format are defined.
RTT	Round trip time as measured by sender. Not used in Minimal Format.
X	Inter-packet time of the flow as calculated by sender. Not used in Minimal Format.
Delta_Throughput	Indicates desired change in throughput by sender. Not used in Minimal Format
Reverse_Feedback	Feedback value of *Delta_Throughput* received by the data receiver.

Table 2.1: XCP header parameters.

The most recent XCP draft specifies a third version of the protocol header where fields are rearranged to optimise performance on an experimental FPGA implementation. Throughout this report however it should be assumed that we are referring to version 2, as it is the only version available in the current kernel patch distributed by ISI [11], XCP's maintainers.

2.2.3 End-System functions

The sender must signal desired changes in throughput by calculating the appropriate *Delta_Throughput* value. This value reflects the per-packet distribution of the throughput change and is required in order to maintain XCP routers oblivious to per-flow congestion states. *Delta_Throughput* is obtained by calculating the difference between the desired and estimated throughput, which represents the appropriate change, and dividing the result by the number of packets in one round-trip time. The latter may be estimated by dividing the current throughput by the maximum segment size, thus obtaining an estimate for the current throughput in packets, and multiplying the resulting value by the round-trip time. The resulting equation is shown below (2.1):

$$Delta\_Throughput = \frac{t_p - t_a}{t_a \cdot \left(\frac{RTT}{MSS}\right)} \tag{2.1}$$

where t_p is the desired throughput, t_a is the current throughput, RTT is the current round-trip time estimate and MSS is the maximum segment size of outgoing packets. Additionally, the sender must set their current estimate of both RTT and X, the inter-packet time. X may be estimated by dividing the round-trip time by the number of outstanding packets or, alternatively, by obtaining the ratio between packet size and the current throughput.

The receiver simply copies the received *Delta_Throughput* value into the *Reverse_Feedback* field of outgoing packets, thus bringing closure to the system's feedback loop.

Upon receiving the feedback value, the sender adjusts its output rate accordingly. Under TCP, this adjustment is achieved by modulating the congestion window, *cwnd*, as described by equation (2.2):

$$cwnd = max\left(cwnd + Reverse\_Feedback \cdot RTT, MSS\right) \tag{2.2}$$

A minimum value of MSS is required to avoid the "Silly Window Syndrome" [12].

2.2.4 Router functions

Despite being an integral part of XCP end hosts perform few functions: the sender requests a change in throughput, while receivers merely return the resulting throughput as feedback. The core of XCP lies in the routers, where the de-coupling of utilization control from fairness control allows the system to converge to optimal efficiency.

Packet arrival On arrival at a router, the data contained in the packet's congestion header is used to update parameters used for further calculations. The router must update the total amount of incoming data, *input_traffic*, by incrementing the current value with size of the incoming packet in bytes. Additionally, the router must maintain an accurate estimate of the average RTT, d, across all flows, without maintaining per-flow state. This is achieved using (2.3):

$$d = \frac{\sum\left(X \cdot RTT\right)}{\sum X} \tag{2.3}$$

Consequently, the router must maintain both the summed total of incoming values of the inter-packet time X and the product of X with the corresponding RTT.

Utilization control Utilization is controlled by adjusting the aggressiveness according to the spare bandwidth and the feedback delay. To this end, the router must periodically calculate the fair capacity at regular control intervals. In the current draft specification of XCP, this interval is defined as being the average RTT, d, across all flows. The bandwidth F available for distribution amongst the flows is given by:

$$F = \alpha \cdot (C - input\_bw) - \beta \cdot \frac{q}{d} \tag{2.4}$$

where C is the capacity of the link, *input_bw* is the input bandwidth since the last control interval and q is the persistent queue. α and β are constants which guarantee system stability. This calculation ensures efficiency by making maximal use of the available link capacity whilst simultaneously draining the current queue.

Fairness control Fairness is ensured by reclaiming and reallocating bandwidth from flows with rates above their fair share. This requires the redistribution of previously allocated bandwidth by performing bandwidth shuffling, thereby certifying that new flows are attributed bandwidth even when the system is stable (F=0). The shuffling function used in the current implementation of XCP is presented in (2.5).

$$S_T = max\,(0, \gamma \cdot input\_bw - |F|) \tag{2.5}$$

where S_T represents the total bandwidth to be shuffled in the current control interval and $\gamma = 0.1$.

The fairness algorithm differentiates the total pool of capacity to be distributed between positive residue feedback, R_p, and the negative residue feedback, R_n.

$$R_p = S_T + max\,(F, 0) \tag{2.6}$$

$$R_n = S_T + max\,(-F, 0) \tag{2.7}$$

The final calculation carried out during the control interval timeout prepares the residue for usage on a per-packet basis. Since positive feedback is applied equally per-flow, the positive feedback scale factor, C_p, takes the positive residue feedback and divides it by the total sum of inter-packet time over the last control interval. Negative feedback, however, is proportional to capacity, therefore having a greater effect on flows occupying the most bandwidth. The negative feedback scale factor results from the division of the negative residue feedback by the total input traffic.

$$C_p = \frac{R_p}{\sum X} \tag{2.8}$$

$$C_n = \frac{R_n}{input\_traffic} \tag{2.9}$$

Before returning, the control interval timeout must schedule a new control interval in d seconds, using a newly calculated average RTT. Statistics collected during the control interval must also be reset.

Packet departure On packet departure, the router must compare the packet's *Delta_Throughput* value with the locally available capacity. To calculate this capacity, the positive and negative feedback associated to a packet must be known. The positive feedback, F_p, is calculated by multiplying the current estimate of the positive feedback scale factor C_p by the packet's declared inter-packet time X, thus allowing for fair per-flow distribution.

$$F_p = C_n \cdot X \tag{2.10}$$

Similarly, the negative feedback, F_n, is calculated by multiplying the current estimate of the negative feedback scale factor C_n by the packet's size, resulting in fairness by levelling the capacity attributed to each flow.

$$F_n = C_p \cdot Pkt\_size \tag{2.11}$$

The total feedback F_t which may be conceded to a packet may therefore be calculated as the difference between the packet's respective positive and negative feedback values:

$$F_t = F_p - F_n \tag{2.12}$$

Should F_t be lower than the packet's *Delta_Throughput*, *Delta_Throughput* is replaced with F_t before packet departure, otherwise the outgoing packet's congestion header remains unchanged.

2.3 RCP - Rate Control Protocol

2.3.1 Protocol overview

Rate Control Protocol (RCP) attempts to emulate processor sharing by using explicit feedback from the network to control end-system throughput. RCP however uses a different approach to congestion control: rather than calculating incremental changes to an end-system's congestion window, routers in RCP dictate the rate at which end-systems should operate. By using a single fair-share rate for all packets, routers have no need for per-packet calculations, resulting in a computationally lighter implementation than XCP. RCP is particularly well suited for bursty traffic: since bandwidth allocation is instantaneous, smaller transfers such as webpages take less time than using TCP or XCP. Such dynamic behaviour however comes at the cost of increased jitter, as queues oscillate to compensate the variation of flows over the network.

2.3.2 End-System functions

As with XCP, RCP requires senders to include a congestion header between the network and transport layer. Since no official implementation of RCP exists however, the congestion format is not yet formally defined. RCP requires the sender to include an estimate of the round-trip time and a desired rate, which by default may be set to ∞. Upon processing a RCP packet, receivers must copy the received rate value into the feedback field of an outgoing packet. The sender must adjust the outgoing flow rate to the value specified in the feedback field.

2.3.3 Router functions

Packet Arrival Packets arriving at the router contain the sender's estimate of the round-trip time, which is used to calculate d, the moving average of RTT across all packets. Additionally the router must increment the total input traffic, *input_traffic*, by the incoming packet's size in bytes.

Rate Control RCP attempts to emulate processor sharing, whereby the fair-share rate $R(t)$ is defined as the even distribution of capacity C amongst the total number of flows, $N(t)$:

$$R(t) = \frac{C}{N(t)} \tag{2.13}$$

This rate would have to be updated periodically during regularly scheduled control interval timeouts. As with XCP, d, the average RTT across flows is used as an interval. RCP however additionally introduces a user-defined update rate interval, τ, which may be used to drain queues at a faster rate. In this case, the update interval T is defined as:

$$T = min\left(\tau, d\right) \tag{2.14}$$

Obtaining an exact value for *N(t)*, as defined in (2.13), is not possible since routers maintain no per-flow state. An estimate however may be calculated by dividing the link capacity by the previously calculated rate:

$$\hat{N}(t) = \frac{C}{R\left(t - T\right)} \tag{2.15}$$

Similarly, the current rate *R(t)* may be obtained from the sum of the previously calculated rate and the amount of aggregate feedback, F, to be attributed to each flow:

$$R\left(t\right) = R\left(t - T\right) + \frac{F}{\hat{N}\left(t\right)} \tag{2.16}$$

F is also used in XCP and was previously defined at (2.4). The aggregate feedback must be scaled by $\frac{T}{d}$, since the control interval may differ from the average RTT, unlike XCP. By replacing (2.4) and (2.15) in (2.16), the complete fair-share rate algorithm becomes:

$$R\left(t\right) = R\left(t - T\right) \cdot \left(1 + \frac{\frac{T}{d} \cdot \left(\alpha \cdot \left(C - input\_bw\right) - \beta \cdot \frac{q}{d}\right)}{C}\right) \tag{2.17}$$

Packet Departure The *rate* parameter on outgoing packets, R_p, is then checked against the locally calculated rate R. If R is lower than R_p the router constitutes a bottleneck in the flow path and must replace R_p with R, otherwise the packet remains unmodified.

2.4 Capacity estimation in XCP

XCP requires that each queue controller knows the exact capacity of the underlying link in order to calculate the aggregate feedback (2.4) correctly. In shared-access media such as IEEE 802.11, an accurate estimate of the actual capacity is difficult to obtain, leading to a loss of efficiency in throughput.

This may easily be deduced by introducing an estimation error, ϵ, to the actual capacity, C_{real}:

$$C = C_{real} + \epsilon \tag{2.18}$$

By replacing (2.18) in (2.4) we obtain the aggregate feedback F with capacity estimation error, (2.19):

$$F = \alpha \cdot (C_{real} + \epsilon - input\_bw) - \beta \cdot \frac{q}{d} \tag{2.19}$$

where *input_bw* is the bandwidth used by incoming traffic over d seconds, the average round-trip time of incoming flows. The length of the persistent queue is represented by q and both α and β are constants. Full utilization of the real link capacity, $C_{real} = input\_bw$, results in an aggregate feedback of zero, since there is no bandwidth to distribute amongst flows. Under these conditions, (2.19) may be further simplified to:

$$q = \frac{\alpha}{\beta} \cdot d \cdot \epsilon \tag{2.20}$$

This result has two important consequences. If ϵ is positive, due to an overestimated link capacity, (2.20) represents the value of the queue length required to stabilise the system. If ϵ is negative, the queue length becomes negative, which physically represents the unused bandwidth during a control interval, provoked by under-estimation of a link's capacity, resulting in a loss of efficiency. A control theory analysis of such behaviour is developed further in [13].

2.4.1 WXCP

In [14], Wu *et al.* propose the Wireless eXplicit Control Protocol (WXCP) to overcome such limitations by adapting XCP for IEEE 802.11 media. By analysing link layer properties, such as the MAC busy and idle times, WXCP consistently estimates the current link capacity. In each control interval a WXCP router distributes an amount of bandwidth F according to:

$$F = \alpha \cdot \frac{B}{n+1} - \beta \cdot \frac{q}{d} - \zeta \cdot \frac{\bar{R}}{d} \tag{2.21}$$

where B is the estimation made by the router of the available bandwidth, n is the number of active neighbor stations during the control interval, q is the persistent queue length, and $\bar{R}$ is a running average of the number of link-layer retransmissions.

B estimates the available bandwidth based on the MAC idle time (d_{free}) and busy time (d_{busy}) over control interval d:

$$B = \frac{d_{free} \cdot bw}{d} \tag{2.22}$$

where bw represent the current output rate of the station. It is obtained by dividing the number of transmitted bytes $b_{transmitted}$ during the last interval, by the air time d_{self} used by the station to transmit that amount of information:

$$bw = \frac{b_{transmitted}}{d_{self}} \tag{2.23}$$

Because each station uses its own output bandwidth, the WXCP algorithm provides air time fairness. This means that stations will get the same amount of air time to transmit, being that stations using higher data rates will also achieve higher output bandwidths.

The number of active neighbor stations n is obtained by inspecting the MAC source address of all packets sent to the medium, irrespective of packet destination, and counting the number of unique addresses. The persistent queue length of the router station, q, is the minimum length of the queue observed during the control interval. Obtaining the average number of link layer retransmissions $\bar{R}$ is not specified by the authors, hence we assume that it consists in a moving average of the number of link layer retransmissions performed by that station alone in each control interval.

α, β, and ζ are system tunable parameters. α controls the amount of unused bandwidth that is distributed in each control interval, β controls the amount of built-up queue that is drained in each control interval, and ζ controls the portion of link layer retransmissions eliminated in each control interval. The authors propose $\alpha = 0.2$, $\beta = 0.1$, and $\zeta = 67$, however they do not provide general support nor stability proof for these values.

Additionally, WXCP proposes a pacing mechanism and a loss recovery mechanism which improve throughput smoothness and robustness to packet loss. These mechanisms, however, are minor improvements in our opinion,

since the core of the algorithm behavior will be driven by the explicit feedback.

While representing an important contribution to addressing limitations present in XCP, the choices taken in designing WXCP lead to some obvious shortcomings in their own right. For one, WXCP relies on link layer information and is therefore invariably tied to the transmission media it was tailored for - IEEE 802.11. While most key concepts may prove portable to different media, such a process is time-consuming and will most likely require fine-tuning. Additionally, WCXP requires stations to monitor and inspect the MAC header of every packet - even if it is not destined for the monitoring station - as well as maintaining look-up tables for existing MAC addresses, foreshadowing a high computational cost and, more worryingly, excessive power consumption. Finally, the WXCP algorithm was insufficiently tested, with no stability proof provided by the authors.

2.4.2 XCP-b

To improve XCP over shared-access media, an alternative algorithm for calculating the feedback aggregate has been specified [7], named XCP-blind, or XCP-b. This algorithm uses the persistent queue length as feedback for estimating link capacity by calculating spare bandwidth as the variation of the queue length over time:

$$F = -\alpha \cdot \frac{\Delta q}{d} - \beta \cdot \frac{q}{d} \tag{2.24}$$

Intuitively, if q is draining over a control interval, the link is under-utilised. Inversely, an increase in the queue level indicates a capacity overload. Under empty queues however (2.24) would not return any feedback at all, which leads to an alternative formula in times of link under-utilization.

$$F = \chi \cdot \frac{Q_{max}}{d} \tag{2.25}$$

where Q_{max} is the queue size and χ is a constant which ensures system stability, defined as $\chi = \frac{1}{5-\alpha-\beta}$. This ensures capacity is distributed amongst flows whilst simultaneously guaranteeing that the queue buffer never overflows unless drastic changes in channel capacity occur. Faced with constant queue fluctuations however, identifying link under-utilization is difficult, leading to erratic behaviour. The authors propose two additional modifications.

Above-zero queue stabilisation By stabilising the queue length above zero, under-utilization is easily identified since small fluctuations are absorbed by the queue. For a value κ at which the queue will stabilise, the aggregate feedback F is given by:

$$F = -\alpha \cdot \frac{\Delta q}{d} - \beta \cdot \frac{q - \kappa}{d} \tag{2.26}$$

Late reaction To avoid reacting impulsively when faced with rapid queue oscillations, the queue length is observed over n intervals and λ, the value of the weighted average of q is maintained (2.27).

$$\lambda_n = 0.25 \cdot q + (1 - 0.25) \cdot \lambda_{n-1} \tag{2.27}$$

The resulting aggregate feedback can therefore be calculated as:

$$F = \begin{cases} \chi \cdot \frac{Q_{max}}{d} & \text{if } \lambda < \tau \cdot \kappa, \\ -\alpha \cdot \frac{\Delta q}{d} - \beta \cdot \frac{q-\kappa}{d} & \text{if } \lambda \geq \tau \cdot \kappa. \end{cases} \tag{2.28}$$

where τ is a function of n:

$$\tau = (1 - 0.25)^n \tag{2.29}$$

Other methods for late reaction, as well as considerations on design parameters, are discussed in [7]. Unlike WXCP, XCP-b is designed to work over any shared-access medium, irrespective of link-layer technology, and does so without adding significant complexity.

Chapter 3

Implementing XCP-b

In this chapter we detail the implementation of XCP-b under the current network stack supporting XCP, available for FreeBSD 6.0, as well as testing the algorithm's performance on a real-life testbed.

3.1 Implementing control parameters

Our implementation of XCP-b allows users to manipulate design parameters in realtime through the *sysctl* interface, which allows kernel tweaking. Users may vary κ, the queue stabilisation length, and n, the number of control intervals before reacting to under-utilization. We chose to implement n rather than τ for two reasons. On one hand, n has a significance which is much easier to grasp than τ, since it indicates the quantity of control intervals the queue controller delays reaction. Additionally, the *sysctl* interface only allows integers to be assigned, thereby greatly limiting the granularity of τ.

This required some optimising at the kernel level, since no interrupt is generated once a variable is changed. Rather than repeatedly calculate τ using (2.29) at every iteration of the feedback algorithm, an additional variable containing the previous value of n, n_p was maintained. For every control interval, n is compared to n_p. If the values differ, τ is recalculated, and n_p retains the value of n.

3.2 Altering the aggregate feedback

The novelty of XCP-b resides in using variations of queue drainage as an indication of available capacity. Since such a feature was not needed in the original XCP algorithm, additional variables were associated to XCP's

queue control block, *xqcb*, to enable calculations involving queue oscillation, namely:

- *queue_w*, the weighted average of q, as described in (2.27).
- *queue_prev*, the value of q in the previous control interval.

Applying XCP-b also implies modifying the control interval timeout, *xcp_ctl_timeout()*, to calculate the aggregate feedback according to (2.28), by implementing the following steps after having calculated *input_bw*:

1. **Update** τ Verify change in n, as described in 3.1, and update τ accordingly. This operation must be forced when τ is equal to zero, which will occur on system boot.
2. **Calculate queue and update weighted average** The current length of the persistent queue must be obtained using (5.7), which takes into account the link overhead, after which the weighted average of the queue must be update.
3. **Calculate aggregate feedback** Having obtained or updated q, λ and τ, the aggregate feedback may be calculated by directly applying (2.28). Care must be taken in avoiding buffer overflow, in particular with negative values since all values are unsigned.
4. **Store queue value** Once the feedback has been obtained, the current queue value must be stored so the queue variation may be deduced on the next control interval

3.3 Logging extensions

A correct analysis of XCP-b requires extracting variables at every control interval onto a logging device. This required coding a new structure (listing 3.1 to *xcp_records.h*, and implementing the appropriate device interface for writing and reading the structure from the device by modifying the appropriate logger tools, namely *loggerd.c* and *decipher.c* respectively.

Of particular interest is the weighted value of the queue, *queue_w*, which should stabilise close to the value of *b_kappa* throughout simulations. Kernel queue length parameters, such as *ifq_len* and *ifq_drv_len*, are extracted in order to provide information on queue limits, which must be understood in order to avoid queue overflows.

```
149 struct router_bi{
150     int32_t     feedback;
151     uint32_t     avg_rtt;
152     uint64_t     queue;              /* Current instanteous queue, B */
153     uint64_t     queue_w;            /* weighted average of queue */
154     uint32_t    b_kappa;          /* permanent queue size (set by user)*/
155     uint32_t    ifq_len;
156     uint32_t    ifq_drv_len;
157     uint32_t    b_n;
158 };
```

Listing 3.1: */sys/dev/xcp/xcp_records.h* with XCP-b extension.

3.4 Test results on a fixed-capacity testbed

In order to validate our implementation we first analyze XCP-b performance over a wired, fixed-capacity setting. This allows us to eliminate spurious behaviour inherent to wireless media which might otherwise disguise the algorithm's shortcomings.

3.4.1 Testbed setup

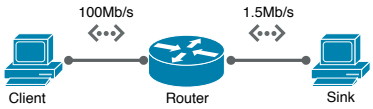


Figure 3.1: Wired testbed setup.

Our testbed was composed by a total of three PCs running FreeBSD 6.0 patched to support XCP and modified with our custom implementation of XCP-b. These nodes were split into different tasks: a client, from which outgoing connections are to be established, one sink, to receive all client flows, and a gateway node with two network interfaces, acting as a router between the client and the sink. All nodes possess 100Mbit/s ethernet network cards and are interconnected by 100Mbit/s switches. The router was configured so as to establish a bottleneck in capacity on the outgoing interface toward the sink, as well as inducing a specific delay on all flows, thereby allowing control over the round-trip time.

Test procedures

Our main objective is to characterise the dynamic behaviour of the underlying congestion control algorithms in adjusting flow throughput to available capacity. With this in mind, we defined a simple procedure which could easily be replicated:

- The client uses the client program *xstream* to establish multiple minute long flows to the server *xserver*, which runs continuously on the sink terminal. Both tools are provided with the XCP patch developed by ISI.
- Four flows are initiated 10 seconds apart, allowing for flow throughput to stabilise and convergence of flows toward optimal link utilization to be easily visualised.
- Link bottleneck is artificially controlled by manipulating the capacity of the router's outgoing interface, toward the sink, to 1.5Mbit/s, using the *pfctl* utility, which permits parameter configuration of the packet filter device.
- All flows are delayed to achieve a round trip time of approximately 100 ms.

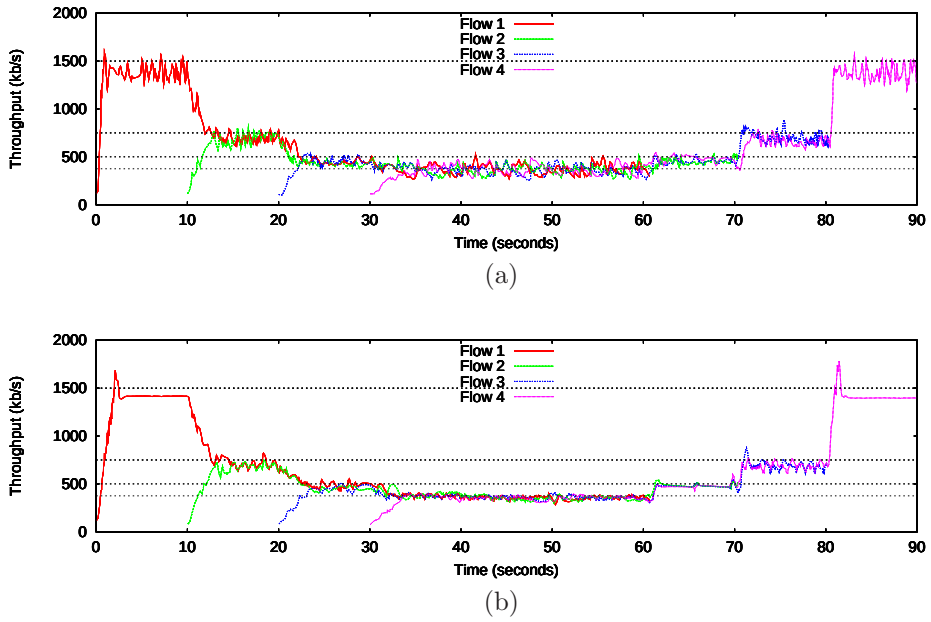
Since only the feedback algorithm was changed, only the router was compiled with XCP-b, whilst both end-hosts maintained the XCP vanilla kernel.

Restricting the bandwidth to such an extent may seem unreasonable given explicit congestion control is touted as improving performance over high bandwidth pipes. Our testing however will not benchmark performance but rather characterize the behaviour of different algorithms, whereby bandwidth becomes secondary. Extensive simulation results comparing TCP, XCP and RCP are presented in [5]. Additionally, by reducing the bandwidth to this extent we gain a tighter bottleneck, generating smaller logs from which we can extract the same traffic patterns used to validate our implementation.

Initially flows were triggered from multiple clients spread over different subnets. Early on this approach was abandoned as impractical. By increasing the number of nodes we were in effect reducing our accuracy in replicating tests under different conditions, whilst synchronizing logs was prone to errors derived from skewed timers, even when regularly updated using NTP.

Unless otherwise stated, all tests implement the same values for α and β, constants used in the calculation of the aggregate feedback. These were set to 0.4 and 0.226, respectively, as recommended in [4].

3.4.2 Test overview



(a)

(b)

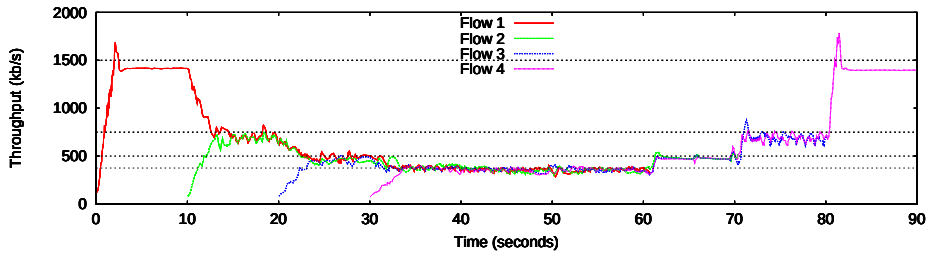
Figure 3.2: A comparative overview of end-host throughput using (a) XCP and (a) XCP-b

Analysing XCP-b requires that it be compared with the standard XCP implementation. Despite being engineered for shared-access media, XCP-b should adequately replicate XCP behaviour with minimal cutbacks in performance. Figure 3.2 displays client throughput in a system with an XCP-b router and the equivalent test results in a regular XCP system, revealing a remarkably similar performance from both algorithms. As expected, by modifying only the aggregate feedback algorithm, XCP-b retains the fairness controller responsible for bandwidth distribution amongst flows, resulting in slow increases to the congestion window until the optimal throughput has been attained. As desired, the utilization controller adequately estimates the available link capacity but reveals some difficulty in adapting to variations in the number of active flows, resulting in limited overshoot. It may seem surprising that despite being oblivious to network capacity

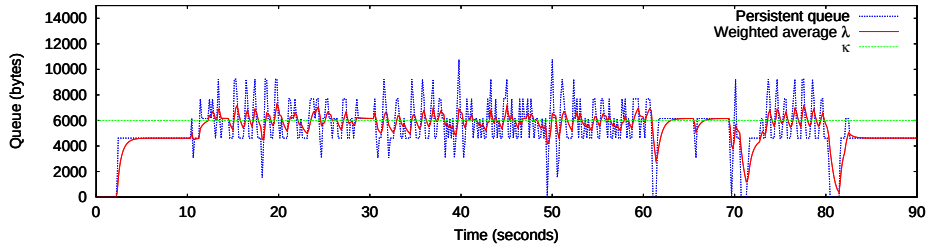
XCP-b achieves more stable throughput than the original XCP algorithm. This result however is a natural consequence of XCP-b design, whereby fluctuations in feedback are buffered in the queue rather than reflected on flow throughput.

The reliability of system results compromising a sole active flow cannot be ensured since routers are dependent on flow statistics to evaluate available bandwidth. As such we will focus on test results where two or more flows are present. Even within such bounds however, some overshoot is visible in the XCP-b test results once the third flow terminates. Understanding the causes which trigger such a reaction requires further analysis of the persistent queue, which provides feedback on link congestion in the absence of an explicit value for the media capacity.

3.4.3 Queue buffering



(a)



(b)

Figure 3.3: An overview of (a) XCP-b test results and the associated (b) queue stabilisation.

Link capacity is estimated by buffering the queue to a predetermined offset κ and interpreting fluctuations in queue size as an indicator of capacity variation. The performance of this feedback loop ultimately defines XCP-b: systematic underestimation of a link's capacity results in sub-optimal

utilization, while overestimation leads to increased queues and network latency.

Figure 3.3 shows the global evolution of end-host throughput and the resulting queue size present in the XCP-b router. The queue offset, κ, was set to 6000 bytes. Despite significant oscillation, the persistent queue size is kept in check by λ, the weighted average that is used to delay reaction. The value of λ closely follows κ. Deviations from this behaviour denote changes in incoming traffic and are usually associated to a decrease in active flows, which in turn spikes end-host throughput.

The resulting queue dynamics are consistent with previous simulations and constitute a significant step in adapting such congestion control protocols to variable-capacity media. As a proof-of-concept, the use of a wired testbed is acceptable in the extent that results are cleaner and anomalies more easily detected. Applying XCP-b on a wired testbed does not clearly emphasise the significant benefits the algorithm offers over the standard XCP implementation. These results do however provide some insight into how such an algorithm will behave in such environments, namely in addressing variations in active flows, which will be covered over the next sections.

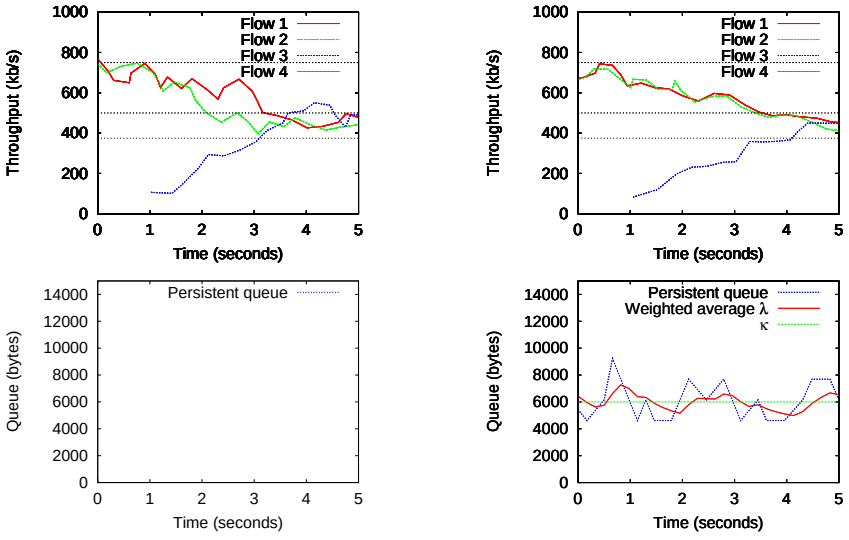


Figure 3.4: The throughput of end-hosts and the router queue as a third flow starts, using (a) XCP and (b) XCP-b.

Flow increase

XCP-b maintains the fairness controller used by XCP and, as such, reacts in much the same approach as the latter when faced with flow increase. Since there is no change in capacity, only the manner in which it must be redistributed amongst flows, XCP-b has no difficulty in converging to optimal utilization in approximately the same time interval as XCP, as shown in figure 3.4.

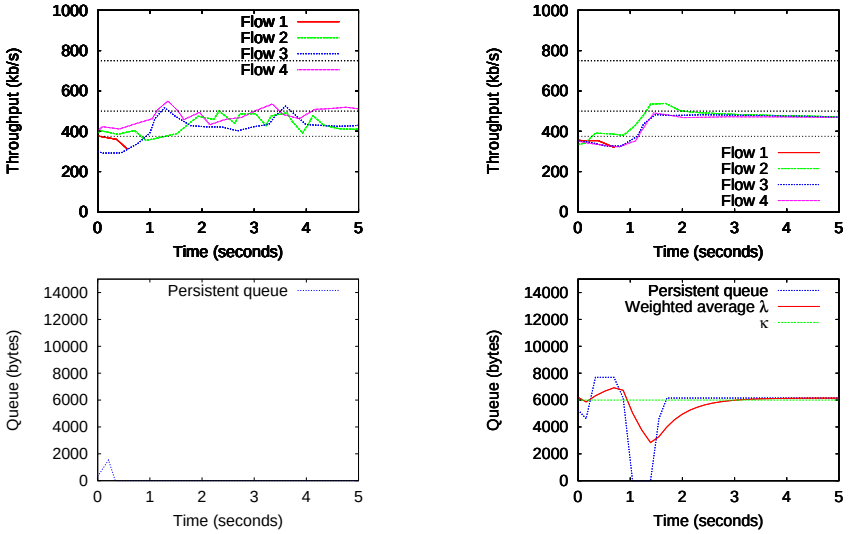
As expected, XCP successfully brokers the arrival of a new flow with no queuing. Interestingly XCP-b achieves similar performance, in that the queue remains very close to offset κ. The queue spikes before the third flow starts, which seems to indicate that queue fluctuations are recurrent and are not correlated with flow arrival. This is intuitive: since the medium is fully utilized, XCP-b must simply redistribute throughput amongst flows without increasing the total bandwidth. A consequence of increased queuing is a slightly more conservative adaptation rate, as additional bandwidth is reserved for queue drainage if λ rises above κ. This results in step increases to the transfer rate, alternating between periods of queue build-up, with rapidly increasing throughput, and queue drainage, where throughput remains stable. Since positive feedback is distributed equally amongst all flows, while negative feedback is attributed proportionally to throughput, the third flow increases throughput at a faster rate than the remaining rates even when the queue is below the predefined offset. The shuffling of traffic then ensures that all flows converge to the same rate.

Flow decrease

Unlike flow increase, a decrease in active flows poses problems in media where the capacity is not known. XCP-b must probe the network for bandwidth and adjust feedback once the queue exceeds the offset level κ, aiming for rapid convergence to optimal throughput whilst avoiding congestion.

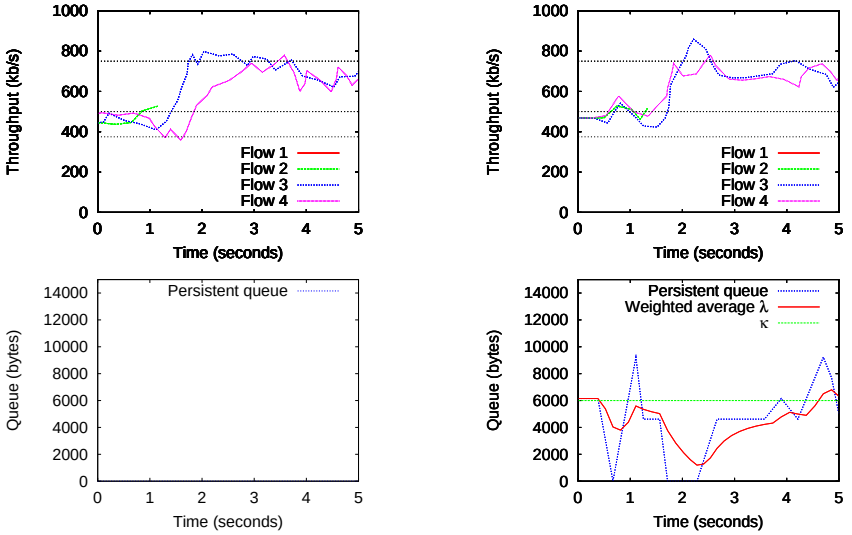
Achieving such performance is not trivial. Figure 3.5 shows throughput adjustment and the respective queue controller as the first flow ends, using both XCP and XCP-b. End-host rates are similar using both utilization control algorithms, which on first analysis would seem to indicate that XCP-b is reasonably robust when reacting to a decrease in active flows.

Interpreting the persistent queue levels would seem to further support this observation. Upon flow completion, the queue experiences a brief underflow, whereby capacity previously used by the completed flow was draining the queue. As the weighted average λ decreased, the aggregate feedback



(a) (b)

Figure 3.5: The throughput of end-hosts and the router queue as the first flow ends, using (a) XCP and (b) XCP-b.



(a) (b)

Figure 3.6: The throughput of end-hosts and the router queue as the second flow ends, using (a) XCP and (b) XCP-b.

available for existing flows was slowly increased, once more filling the queue to match the offset value κ. In this case, the transition to three flows occurred extremely smoothly, with very little overshoot. In part however, these results may present some bias, as immediately before the first flow ends, the queue is above κ. By dropping from a higher value, λ may have compensated a potential overestimation of link capacity. While this is a common occurrence, it does not address the worst-case scenario, where a flow is completed with the queue drained below offset level.

The completion of the second flow, shown in figure 3.6, reflects some of these concerns. As the flow ends, the queue is stable, with λ following the value of κ. The sudden decrease in active flows frees a significant amount of capacity, which completely drains the queue. As λ slowly drops, the throughput of the remaining flows increases, but not fast enough for the queue to build up. With no feedback, λ continues to drop, prompting end-host to increase rates excessively causing an overshoot in throughput. Despite this, the queue itself does not peak which indicates that κ is sufficiently high to provision the queue controller against flow decrease under test conditions.

The above example illustrates many of the challenges facing XCP-b, namely how control parameters should be tweaked to obtain optimal performance. An increase in κ avoids queue depletion, providing the utilization controller with some feedback when faced with abrupt changes, but at the cost of increased network latency. Likewise, λ could be calculated over an increased number of intervals, further delaying reaction, at the cost of increased convergence times. Alternatively, the weighted average could differentiate between both cases, attributing greater importance to symptoms of congestion, when the queue size is above κ, than signs of available bandwidth.

3.4.4 Open issues

While our results are encouraging, the performance of XCP-b should inevitably be tied to the value of κ used. We used the value of 6000 bytes, the equivalent of four packets, to avoid queue depletion. When quantifying κ we should compare it to the bandwidth-delay product (BDP). In our test setting, κ is 32% of the BDP. Although this value is high, we must consider packet, rather than byte, granularity. If we were to use 10% of the BDP, 1875 bytes, the persistent queue would fluctuate between one and two packets as λ tries to accompany κ. At such a low offset, queue depletion would also occur far too often.

On the other hand, if the BDP were higher it is feasible that less than

10% of the total value could be an adequate choice for the queue offset. Unfortunately, the initial specification of XCP-b provides no recommendation on appropriate values for κ.

Lastly, the influence of the number of flows must be adequately studied and subject of future research. The completion of a flow with considerable ratio of the overall capacity resulted in overshoot in our last example. The net effect of multiple flows must be actively validated through experimental results in order to understand how XCP-b must react in order to provide stability over variable-capacity media. Over the next section we validate XCP-b in a wireless environment, this time highlighting improvements over the original XCP algorithm rather than characterising XCP-b system response, which has been our focus thus far.

3.5 Test results on variable-capacity media

Upon successfully verifying the correct behaviour of the XCP-b algorithm over fixed-capacity media, we turn our attention to variable-capacity media, for which the algorithm was originally designed.

3.5.1 Testbed setup

In order to draw comparisons with previous results, we repeat test conditions and procedures previously described in section 3.4.1, replacing the link between router and sink with a IEEE 802.11g connection, as illustrated in 3.7.

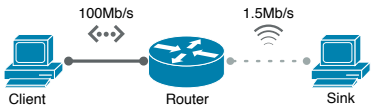


Figure 3.7: Wireless testbed setup.

Despite being throttled down to 1.5Mbit/s we expect available capacity on the wireless interface to vary over time, mainly due to interference from other wireless media. While the randomness of such interference can not be replicated between tests, this does not constitute a significant drawback as we intend to understand how each algorithm responds to such a change in capacity, irrespective of the moment at which it occurs.

3.5.2 Test results

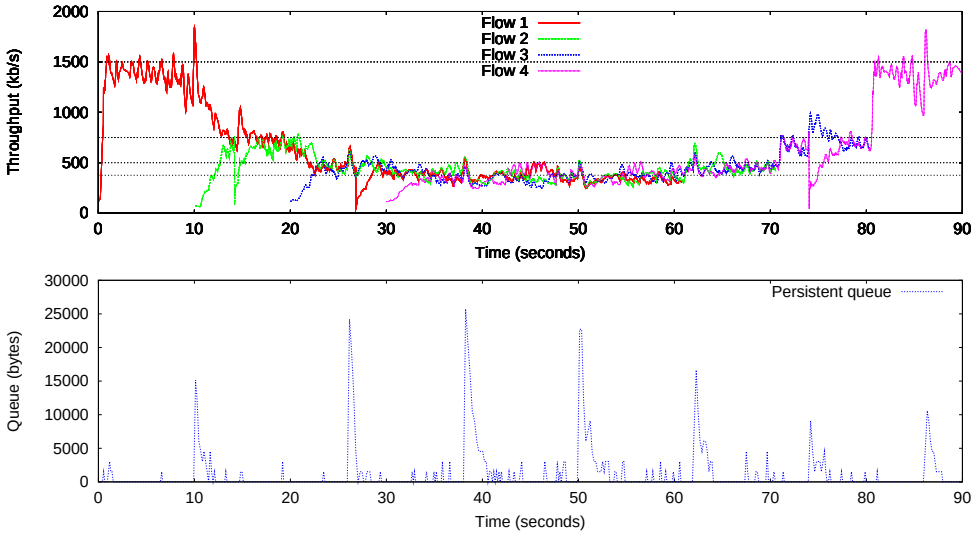
Figure 3.8 compares performance of both XCP and XCP-b over a wireless testbed. As expected, XCP performance is hindered by inaccurate knowledge of media capacity. By overestimating available capacity, XCP becomes vulnerable to queue spikes which it cannot predict when faced with varying conditions. The queue peaks at 26kB, which is significant, particularly when compared to previous performance in a fixed-capacity testbed, where the queue did not exceed 4500 bytes. The resulting jitter is particularly harmful for streaming applications, where latency is acceptable if kept constant. As such, XCP over variable-capacity media does not improve on one of TCP's shortcomings, and one it strived to correct.

On the other hand, XCP-b fares remarkably well, with few limitations when compared to results on a wired testbed. Despite not being immune to changes in capacity, XCP-b successfully absorbs such variations in the queue level, peaking at little over 12kB, double the offset κ.

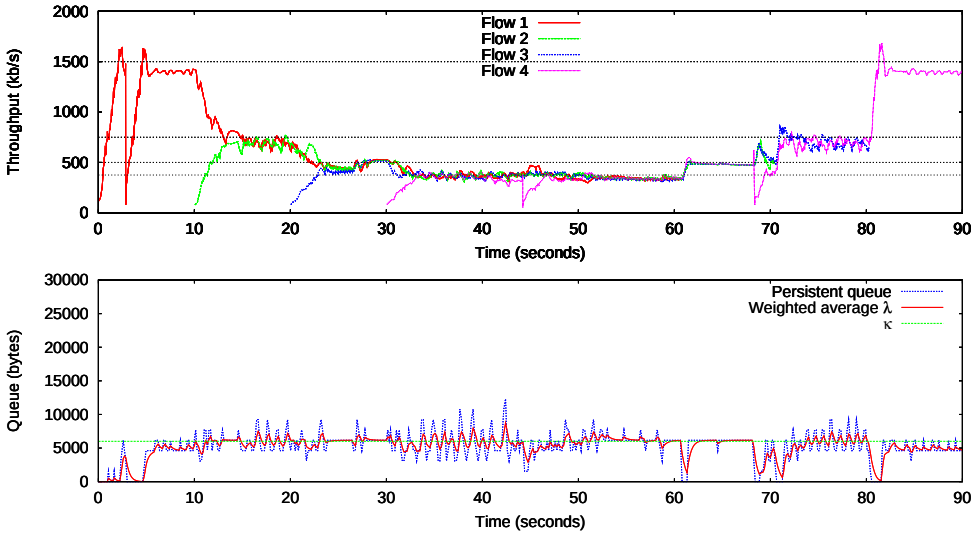
Both algorithms are naturally unable to avoid packet loss over wireless media and react in a conservative manner. For XCP-b packet loss has a greater impact however, since it influences the rate of queue depletion significantly. Once again, the value of κ plays a vital role in adequately reacting to such occurrences. In terms of overall throughput, packet loss is functionally equivalent to the reentry of a flow: capacity is initially freed and then slowly reclaimed by the flow which experienced packet loss. In our tests, recovering from packet loss affecting one flow only triggered a small buffer underrun of the queue. If, however, channel interference were to affect all flows equally, it is unclear how well XCP-b would fare in responding to such widespread packet loss.

Physically limiting the interface to a bottleneck capacity might also skew test results to some extent, as we are effectively throttling the media to a capacity it can withstand, albeit small variations over time and channel interference. Had we discarded the packet filter and run the tests at nominal capacity (i.e. 54Mbps) it is likely XCP performance would be further deteriorated with the increase of the error margin between perceived and real capacity of the media. Likewise, the probability of packet loss would increase as the router aggressively tries to achieve rates which the channel cannot withstand. In contrast, we expect XCP-b to saturate the medium whilst recognizing the limit in capacity from queue growth, consequently stabilising flow throughput to achieve a steady state.

We conclude that inferring link characteristics from queue dynamics is a powerful tool in improving explicit congestion control algorithms over unpredictable media, at the expense of increased latency. In a typical



(a)



(b)

Figure 3.8: A comparative overview of end-host throughput and resulting queue using (a) XCP and (b) XCP-b on a wireless testbed

setting composed of a wireless access point serving as a gateway to a wired infrastructure, such a concession may be acceptable, but it is not clear such algorithms would be as efficient in other environments, particularly ad-hoc networks where latency would become proportional to the hop count a packet takes. If we are willing to introduce some latency, our results suggest XCP-b provides significant improvements over XCP for variable-capacity media whilst maintaining two of its most important characteristics, namely high link utilisation and fairness. The efficiency of such an algorithm is inevitably tied to the size of the queue we are willing to buffer and while there are no clear guidelines on how such a system should be designed, over the next chapter we will present variations of the XCP-blind algorithm which focus on highlighting alternative approaches in enhancing system response.

Chapter 4

Beyond XCP-b

Based on the initial test results using XCP-b, this chapter explores variants to the core algorithm which could further enhance performance in varying network conditions.

4.1 Self-tuning κ

4.1.1 Reformulating the aggregate feedback

The original XCP-b algorithm, detailed in [7], demonstrates some erratic queue behaviour and a design parameter, κ, with an important influence in system performance yet with very few guidelines on what values it should be set to. Ideally an XCP-b system would adapt its permanent queue threshold as a function of queue fluctuation. With this in mind, a self-tuning κ would free system designer's from the responsibility of explicitly setting κ, but rather limit its value between upper and lower bounds.

This is achieved by adjusting κ to approximate the queue's standard deviation. During under-utilization periods the persistent queue will be zero, thus we calculate the standard deviation as if the queue is at some upper bound target length Q_{max}. As a result, during under-utilization, κ grows up to Q_{max}, enabling the distribution of bandwidth until the medium becomes saturated again. κ at iteration n can be expressed as:

$$\kappa_n = \begin{cases} \sigma_n(q) = \rho \cdot |q - \bar{q}_n| + (1 - \rho) \cdot \kappa_{n-1} & \text{if } q > 0, \\ \rho \cdot |Q_{max} - \bar{q}_n| + (1 - \rho) \cdot \kappa_{n-1} & \text{if } q = 0. \end{cases} \tag{4.1}$$

where $\sigma_n(q)$ represents the standard deviation of the queue in control interval n. Q_{max} is the target queue length mentioned above and should be set at most to the total buffer size of the queue to avoid excessive

packet loss. $\bar{q}$ is the exponential moving average of the queue calculated as $\bar{q} = \rho \cdot q + (1 - \rho) \cdot \bar{q}_{n-1}$. ρ is the constant that controls the pace of the exponential moving average of both the queue and its standard deviation.

Using (4.1) we can now revisit the initial XCP-b specification, where a "blind" amount of bandwidth is distributed when queue speed cannot be measured.

$$F = -\alpha \cdot \frac{\Delta q}{d} - \beta \cdot \frac{q - \kappa}{d} \tag{4.2}$$

where Δq is the variation of the persistent queue within a control interval d.

In the original specification of XCP-b, the controller would switch between the feedback function presented in (4.2) and a fixed increment, depending on the value of the exponential weighted average of the queue. If κ itself is adjusted to vary according the the queue's standard deviation however, switching between queue speed and fixed feedback strategies become embedded directly, allowing for smoother queue dynamics.

Controlling delayed reaction

As with XCP-b, delaying reaction to queue drainage is necessary in order to prevent false-positive identification of media under-utilization. Waiting for the queue to be empty for a few control intervals before actually considering the link under-utilized greatly reduces the number of unnecessary oscillations of both $\kappa(n)$ and the queue itself. The expression of $\kappa(t)$ including the delayed reaction feature is:

$$\kappa_n = \begin{cases} \sigma_n(q) = \rho \cdot |q - \bar{q}_n| + (1 - \rho) \cdot \kappa_{n-1} & \text{if } n > \mathit{counter}, \\ \rho \cdot |Q_{max} - \bar{q}_n| + (1 - \rho) \cdot \kappa_{n-1} & \text{if } n \leq \mathit{counter}. \end{cases} \tag{4.3}$$

where *counter* is a counter of the number of consecutive intervals during which the queue has been completely empty, and n is the threshold level for considering the link to be under-utilized. Alternatively, the identification of under-utilization may be implemented using the exponential average of the queue, which is already calculated by the router:

$$\kappa_n = \begin{cases} \sigma_n(q) = \rho \cdot |q - \bar{q}_n| + (1 - \rho) \cdot \kappa_{n-1} & \text{if } \bar{q} \geq \tau \cdot \kappa\,(t - d), \\ \rho \cdot |Q_{max} - \bar{q}_n| + (1 - \rho) \cdot \kappa_{n-1} & \text{if } \bar{q} < \tau \cdot \kappa\,(t - d). \end{cases} \tag{4.4}$$

τ should be set so temporary periods of queue depletion, due to medium access randomness or transient situations, do not immediately trigger an increase in κ, and has the following relationship with n:

$$\tau \approx (1-\rho)^n \tag{4.5}$$

Setting τ is inherently associated to choosing the number of intervals before reaction, n. n has to be high enough to cover regular bandwidth fluctuations during the transient period while at the same time not preventing the system from detecting under-utilization in a timely manner. A reasonable assumption is to consider that typical bandwidth fluctuations oscillate at the system fundamental frequency of $w_{xcp} = \frac{\beta}{\alpha \cdot d}$, as shown in [13]. Since the controller should wait for at least half of this period before considering the medium under-utilized, we may set n as:

$$n \geq \pi \cdot \frac{\alpha}{\beta} \tag{4.6}$$

which, for the recommended values of α and β results in $n \geq 5.54$. For our test results we will use $n = 6$ and, as a consequence of (4.5), $\tau = 0.225$.

Likewise, ρ should be set so that the pace of the moving average is slower than the system fundamental dynamics. Considering the exponential moving average is in effect a low-pass filter, we achieve a slower pace by setting the cut-off frequency of this low-pass filter lower than the fundamental frequency of the feedback system. Setting ρ to achieve a cut-off frequency n-times lower than the open-loop response is therefore simply a case of applying the rules of a low-pass filter. Noting that $w_{xcp} = \frac{\beta}{\alpha \cdot d}$ is the cut-off frequency of the open-loop system and that $\pi \cdot \frac{\alpha}{\beta} \cdot d$ represents half of the period of the cut-off frequency w_{xcp}, ρ should be calculated using (4.7):

$$\rho = \frac{d}{\frac{n}{w_c} + d} = \frac{1}{n \cdot \frac{\alpha}{\beta} + 1} \tag{4.7}$$

which, for $n = 6$ and the recommended values of α and β results in $\rho = 0.22$.

4.1.2 Test results

Using a self-tuning κ requires minimal code changes to the original XCP-b implementation, which is thoroughly described in chapter 3. As such, we implement the new algorithm and re-run our tests under the same conditions as those used to validate XCP-b over wireless media. For this test we limit κ to κ_{min},a minimum threshold set to 6000 bytes, which again is

similar to previous values chosen upon testing XCP-b. The resulting performance, displayed in figure 4.1.2 shows a more dynamic approach to queue buffering, with the moving average adapting itself between our predefined bounds, defined by Q_{max} and κ_{min}.

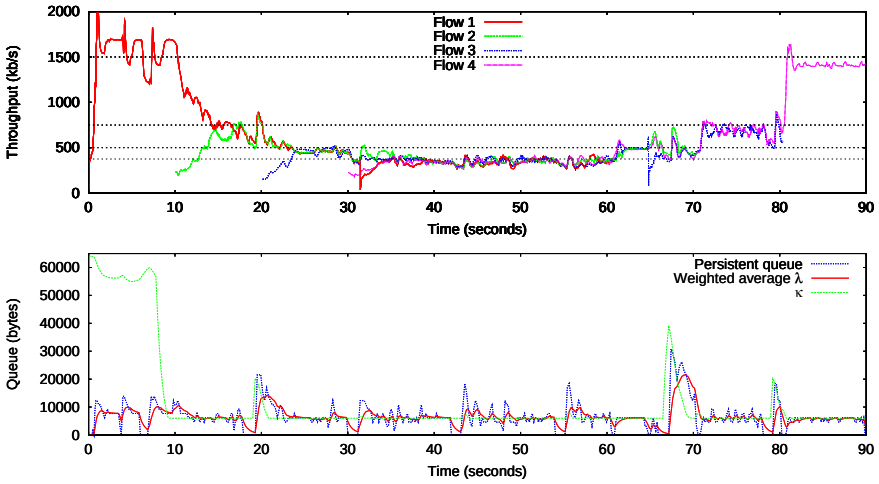


Figure 4.1: An overview of end-host throughput and resulting queue using XCP-b with a self-tuning κ.

By setting Q_{max} to equal the buffer limit, we are in effect turning our algorithm more aggressive in the distribution of bandwidth, which in turn spikes the queue build-up. While queueing is naturally an undesirable consequence, a self-tuning κ allows far greater flexibility in harnessing system behaviour. By establishing bounds within which we accept the queue to oscillate, rather than establishing a fixed value, we establish a more practical framework which can in turn focus on convergence time or queue minimization, thereby reflecting design choices, rather that design constraints.

The benefits of having a variable queue level should be more visible under heterogeneous conditions, where a greater variance amongst flows is reflected in greater fluctuations in the persistent queue. Under such conditions, a fixed threshold is unlikely to perform as successfully as using a threshold which naturally reaches a state of equilibrium. Future work should focus on testing both algorithms under different scenarios and with greater diversity of cross-traffic, which we expect will highlight the versatility using a self-tuning κ has over the original XCP-b proposal.

It should also be noted that thus far we have considered queue length in terms of bytes rather than packets. For shared-access media this may not be ideal, particularly in the case where the medium provides fair access in

terms of packets, e.g. IEEE 802.11. A station contending for access in this type of transmission medium, when granted medium access, is only able to send one packet, not a certain amount of bytes. For this reason, specifying κ in terms of bytes is a bad choice. If a router has been queuing packets 1500 bytes long, for $\kappa = 4500$ bytes, should the router start receiving small packets (e.g. 40 byte long TCP Acknowledgements), it will allow an extremely large number of these small packets to be queued. This large number of small packets will cause large queuing delay because the router is only able to dispatch one packet whenever it is able to grab the medium for transmission (assuming the medium is saturated). Future refinements of our algorithm should propose measuring all queue-related variables in terms of packets in order to further improve robustness in such cases.

4.2 Error Suppression algorithm

An alternative algorithm which builds on XCP-b would be to assume a capacity estimation error and attempt to estimate the associated error by interpreting changes in queue size. This Error Suppression (ErrorS) algorithm therefore consists in assuming the original calculation of the aggregate feedback F, using an arbitrarily set capacity C. If this value differs from the true capacity, then the queue dynamics will incorporate this error and we may use the queue growth to estimate the difference between $C - C_{real}$. The relationship between the capacity estimation error and queue length has already been discussed in section 2.4 and is quantified in (2.20), which is now solved in order to the error ϵ and taking κ into consideration:

$$\epsilon = \frac{\beta}{\alpha} \cdot \frac{q - \kappa}{d} \tag{4.8}$$

meaning that from the value of the persistent queue we obtain an estimation of the instantaneous error. If we can calculate the instantaneous error in our capacity estimation we should be able to approximate ourselves to a media's real capacity by integrating this error over successive periods:

$$\xi(t) = \frac{\psi}{d^2} \frac{\beta}{\alpha} \int_0^t q(t)dt \tag{4.9}$$

$$\xi(n) \approx \frac{\psi}{d} \left(\epsilon_n + \epsilon_{n-1}\right) \cdot \frac{d}{2} + \xi_{n-1} \tag{4.10}$$

where ψ is a control parameter responsible for throttling integration over time. We should pause for a moment to verify the significance of (4.10): we

approximate the integration of successive values of ϵ, which is the rate of queue oscillation in bytes per second. By dividing this result by d, we obtain ξ, the capacity offset estimate, which is also, as is obvious, in bytes per second. By now feeding the estimate back into the original XCP aggregate feedback algorithm (2.4) we are able to correct the difference between the estimated and real capacity, thereby eliminating the systematic offset which is reflected in the persistent queue size.

$$F = \alpha\left(C - y(t)\right) - \xi - \beta \cdot \frac{q - \kappa}{d} \tag{4.11}$$

The ErrorS algorithm also adopts the technique used by the Blind algorithm of stabilizing the queue length at a positive value, so that variations can be measured around a threshold. As with the Blind algorithm the impossibility of reading negative queue values can be a problem in cases where the real capacity has been under-estimated ($C_{real} > C$) leading to the link being under-utilized. In these cases an arbitrary positive feedback is given. For the ErrorS algorithm we use the approach described in section 4.1, providing a self-tuning κ which switches between feedback strategies, as described in (4.4).

Upon implementing the ErrorS algorithm it became obvious that the value of ξ had to be bounded. Since κ is equal to Q_{max} in the absence of flows, the error estimate ϵ will naturally be negative. As a consequence, our capacity estimate ξ would decrease systematically until a flow enters the system, by which time the capacity estimate was so skewed that it could not be recovered in an useful time frame (i.e. within a flows lifetime). To avoid such cases, and rather than defining a minimum capacity estimate which we could not possibly know, we simply defined that the capacity estimate is set to zero unless there is input traffic.

4.2.1 Test results

We implemented the Error Suppression algorithm as defined by equations (4.11), (4.10) and (4.4) over our previous XCP-b implementation and repeated the test procedures described in 3.4.1 on a wireless testbed, as described in 3.7. For initial testing we set conservative values in order to verify system behaviour. As such, α was set to 0.6, β to 0.2 and ψ to 0.15. We also limited the values of κ, setting Q_{max} to 20kB, and κ_{min} to 6000 bytes.

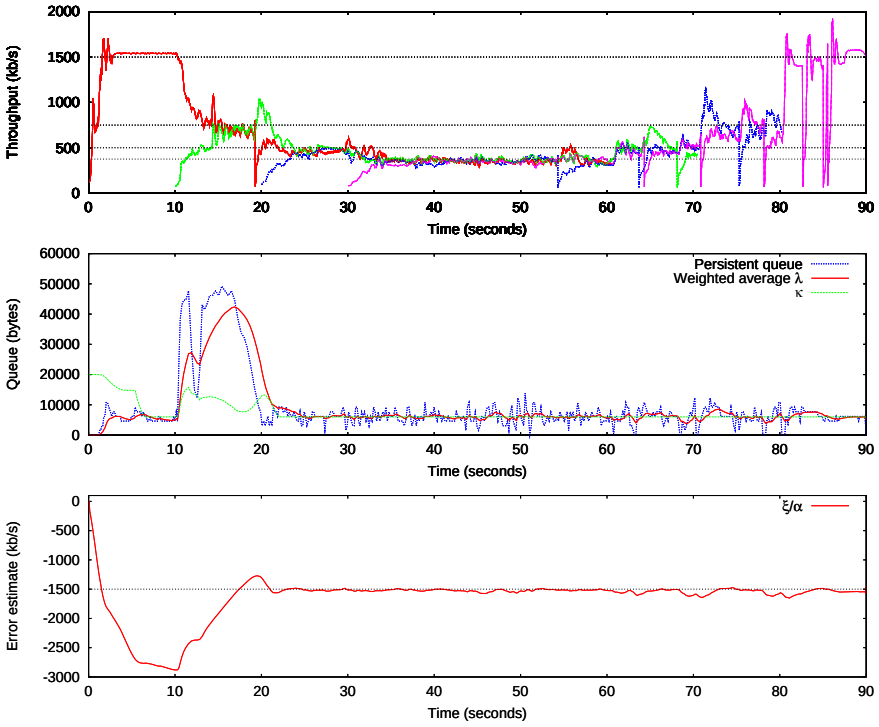


Figure 4.2: An overview of end-host throughput and resulting queue using the ErrorS algorithm with no capacity estimate.

Capacity underestimation

In our first test we cripple the algorithm's estimate of the capacity by deliberately setting C to 0. As can be seen in figure 4.2, our capacity estimate, when divided by α, approaches the value of our bottleneck capacity. The reason we must divide the value of ξ by α is a direct consequence of the aggregate feedback calculated in (4.11), which weighs capacity with α.

While ultimately we are interested in overall throughput, we will focus for now on the behaviour of the queue and the correlation with the capacity estimate. Clearly these results show some difficulty in adjusting throughput rates while an accurate capacity estimate is not yet reached. As the first flow start, ξ ramps downwards extremely rapidly as a consequence of the value of κ. This results in a gross overestimation of system capacity, which is only reversed once the second flow starts to inject traffic, causing the queue to build up to 50kB. With this feedback in place, ξ slowly builds up to the correct estimate whilst draining the queue. Once the queue has been drained the capacity estimate remains stable, as does the queue and throughput rates, which confirms that the Error Suppression algorithm works well in principle.

It is interesting to note that once ErrorS has achieved a stable plateau, the queue becomes extremely robust. In particular, despite flow completion and packet loss, the weighted queue average never drops significantly below κ, which may hint that the algorithm is less prone to erroneously detecting link underutilization.

Using an accurate capacity estimate

Next, we try to understand what would have should we have an accurate estimate for the capacity. Figure 4.3 shows test results obtained running the ErrorS algorithm with the capacity set to 1.5Mbps, which is the same capacity configured at the bottleneck. Improvements over figure 4.2 are clearly visible. Despite and undershoot (which represents and overestimate in network capacity), the capacity estimate steadily builds up before stabilizing, causing a mild, but acceptable, overshoot in the persistent queue.

By having an accurate initial estimate of a link's capacity ErrorS is able to quickly converge to the correct error estimate. This positions ErrorS apart from XCP-b, since it uses, but does not depend on, information on capacity as opposed to discarding the information altogether. In this case, even with an accurate capacity estimate, results are promising, but still a long way from being adequately optimized. For one, convergence of the estimated capacity takes too long, in part because the integration starts

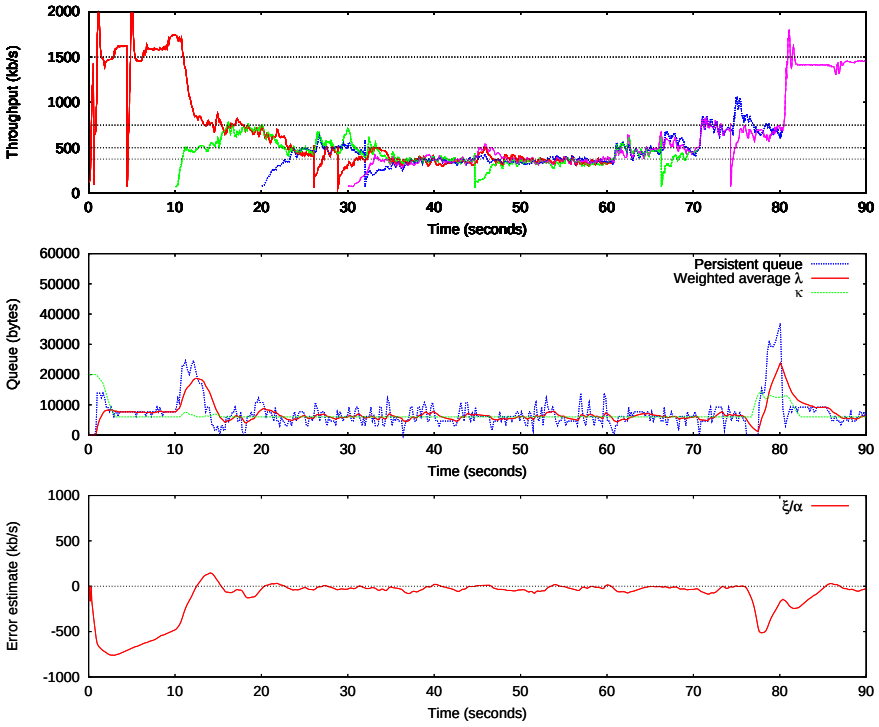


Figure 4.3: An overview of end-host throughput and resulting queue using the ErrorS algorithm with a capacity estimate equal to the real capacity.

before queue build up, producing an overly optimistic estimate for the capacity offset. Likewise, despite recognizing under-utilization well, ErrorS seems to have some difficulty in adapting throughput once the estimate has become inaccurate. This is clearly visible when the third flow ends: the queue underrun causes an increase of κ in an attempt to probe for bandwidth. This in turn causes the capacity estimate to increase. By coupling a high κ and an overestimation of link capacity ErrorS becomes vulnerable to far more queue build-up than XCP-b under similar circumstances. Before proposing any changes however we will first see what happens if we overestimate capacity in a systematic manner.

Capacity overestimation

To test capacity overestimation we insert an estimated capacity corresponding to twice the real capacity. By expecting a link capacity of 3Mb/s, the original XCP algorithm would consistently over allocate bandwidth, thereby inducing queue growth and packet loss. With ErrorS however, as seen in figure 4.4, once the capacity offset estimate, weighed by α, settles at 1.5Mb/s, the queue level is contained and throughput is stable, despite some worrying packet loss.

As with previous tests, the initial queue build-up as ξ converges to the correct estimate is alarmingly high. One obvious explanation is the weight of system parameters, which have not yet been sufficiently studied, and in particular the value at which we should set ψ. If this control parameter were higher, our capacity estimate would adapt to changing conditions more rapidly, which is a benefit during the starting phase, but may lead to excessive oscillations during the stable state. An adaptive ψ might be feasible to distinguish between both states at the cost of increased complexity. Finally, dropping the value of Q_{max} would improve the initial undershoot, which drops to low values due to κ. Unfortunately ErrorS adds additional complexity to the original feedback aggregate function and therefore is responsible for causing queueing alongside κ. This means that Q_{max} loses much of its original meaning as a queue upperbound limit, as is visible from our ErrorS test results were the queue frequently spikes beyond Q_{max}.

Future work should address how ErrorS functions in this transient state. We show that it is possible to infer link capacity accurately over time by inducing successive error estimates, and that by integrating these values we can achieve a more stable queue level. In our initial approach we use values which we know to be stable but with no guarantees on how well they would perform. While the results are acceptable, we intend to further improve heuristics and system parameters in order to minimize queue spikes in the

system queue. As is, ErrorS is an important approach for media where we have an approximate estimate of capacity, but are unable to provide a precise value due to changing conditions or variable overhead.

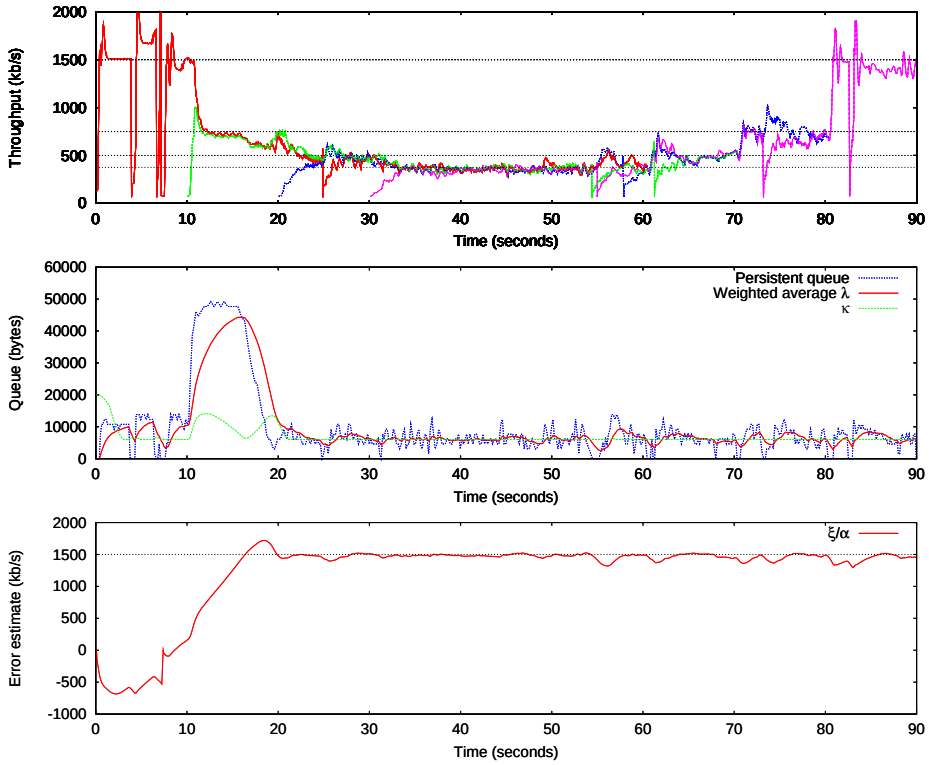


Figure 4.4: An overview of end-host throughput and resulting queue using the ErrorS algorithm with a capacity estimate double the real capacity.

Chapter 5

Implementing RCP

Rate Control Protocol provides explicit congestion control in a similar manner to XCP: both use feedback to relay information on congestion from the network back to the source. RCP however differs from XCP in how the aggregate feedback is calculated and distributed, achieving faster fair rate convergence at the expense of increased queuing. This chapter describes the implementation of RCP by focusing on changes relative to XCP.

5.1 RCP header format

To reduce redundant code, the current XCP congestion header, described in 2.2.2, was used as a template for the RCP header. Our implementation therefore treats RCP as a variant of XCP, differentiating packets based on the value of the *Version* parameter rather than on the protocol value.

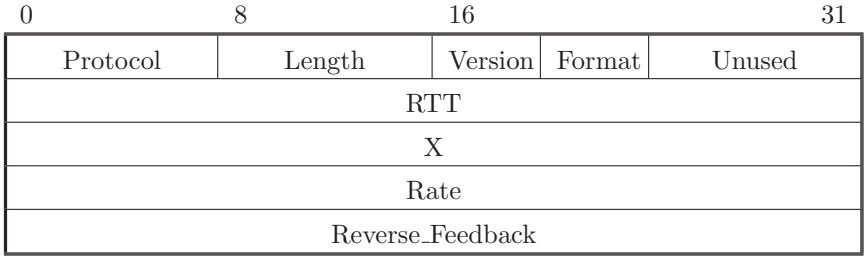


Figure 5.1: RCP congestion header.

As illustrated by figure 5.1, the only difference between RCP and XCP header is the *Rate* parameter, instead of *Delta_Throughput*. This is consistent with how RCP functions, explicitly notifying intermediate nodes of

an absolute value for the desired throughput rather than variations of the latter. On a technical note, this requires interpreting both *Rate* and *Reverse_Feedback* values as unsigned integers, since throughput may not be negative.

RTT and *X* have been included in the RCP congestion header and retain their previous functionality. Although not specified as requirements for RCP, the underlying algorithm for calculating the fair-share rate requires an accurate estimate of the average round-trip time over all flows. XCP obtains this estimate using (2.3), which requires values for both the round-trip time and the inter-packet interval on all packets. We decided to apply the same method for use with RCP, thereby maximising the amount of code shared by both protocols.

5.2 Inserting the RCP header

The sender must insert the congestion header on every outgoing packet as with XCP, with two important modifications to standard XCP behaviour:

- ***Version*** The congestion header must establish itself as an RCP header by altering the *Version* field to the appropriate identifier.

- ***Rate*** As opposed to sending a desired variation in throughput, the sender must set *Rate* to the highest possible value so it may be subsequently reduced by routers until the bottleneck rate for the path is set. This field is measured in bytes per second and is unsigned. Since *Rate* is a 32-bit field, this implementation of RCP limits throughput to approximately 34Gbps.

Both changes required modifying the existing *xcp_insert_header()* function in *xcp.c*.

5.3 Routing RCP packets

Routers with RCP perform two concurrent functions: they must process packets, updating local statistics and analysing the feasibility of the requested rate, and periodically update a fair-share rate to apply to all flows. Although this section will dissect both functions separately, both are mutually dependent. Forwarding requires the knowledge of the fair-share rate, while calculating this rate requires data collected from incoming packets.

5.3.1 Forwarding

Upon verifying that the destination IP address of an incoming packet is not assigned to a local interface, XCP-enabled routers queue datagrams for processing by the forwarding function associated with XCP, *xcp_forward()*. Queue processing may conceptually be divided into operations performed on packet arrival and packet departure.

Packet arrival

Upon packet arrival, the packet size registered in the IP header, *packet_len*, is summed to a local counter registering the volume of data arriving during a control interval, *input_bytes* (5.1). A variable registering the total number of packets received, *packet_num* is also incremented. Both operations assume a correct formation of the IP header since the incoming packet has already passed the IP layer. The congestion header must then be validated, namely by verifying the received header has the correct size and contains a recognised version of XCP. XCP routers without RCP support will drop the packet at this point since the version number will not match any defined standard. Otherwise, both the RTT and X fields will be extracted in order to update variables contained in the XCP queue control block (*xqcb*), namely *sum_x* (5.2) and *sum_xrtt* (5.3).

$$input\_bytes = \sum packet\_len \tag{5.1}$$

$$sum\_x = \sum X \tag{5.2}$$

$$sum\_xrtt = \sum (X \cdot RTT) \tag{5.3}$$

Packets with an RTT of zero should be ignored in both equations, as they hold no physical significance. The RTT is set to zero when the sender does not have an estimate of the round-trip time, usually during connection establishment. Using these packets would inevitably lead to skewed estimates.

Packet departure

On packet departure, the *Rate* field of the RCP packet, $Rate_p$, must be replaced by the current fair-share rate, $Rate_l$, if the latter has a lower value

(5.4). Once this operation has been processed the packet must be queued for output as processing of the congestion header has been completed.

$$Rate_p = min\left(Rate_p, Rate_l\right) \tag{5.4}$$

The variable responsible for containing the value of $Rate_l$ must be included in the *xqcb* structure and will be discussed in the next section.

It should be highlighted that RCP differs significantly from XCP on packet departure: XCP requires per-packet calculations using aggregate feedback scaling, as demonstrated in (2.10), (2.11). Packet arrival merely requires adequate version recognition to support RCP.

5.3.2 Calculations on control timeout

As described in section 2.3, a control timeout must be regularly scheduled, using the average round trip time across all flows as the timeout interval. The tasks executed on each timeout are detailed over the following paragraphs, all related to modifications of the *xcp_ctl_timeout* function.

Calculate average *RTT* An accurate estimate of the average RTT is vital, both for calculating a fair rate for all flows and assuring that control timeouts remain adaptive to flows. The average RTT over a control interval is calculated as follows:

$$avg\_rtt = \frac{sum\_x}{sum\_xrtt} \tag{5.5}$$

where *sum_x* and *sum_xrtt* were previously defined in (5.2) and (5.3).

Calculate input bandwidth The input bandwidth, *input_bw*, over a control interval must be calculated in order to estimate the amount of available capacity. Doing so requires dividing the total amount of incoming traffic by the previous control interval, *ctl_interval* (5.6).

$$input\_bw = \frac{input\_bytes + packet\_num \cdot link\_overhead}{ctl\_interval} \tag{5.6}$$

The incoming traffic must take into account the link-layer overhead which is not accumulated in *input_bytes*. This overhead is proportional to the number of packets and, when using for links using Ethernet (IEEE 802.3) [15], this value should be set to 24 bytes.

Calculate drainage rate of persistent queue The value of the persistent queue in bytes, *queue_b* and the number of packets which compose the queue, *queue_p*, are periodically calculated and stored in the *xqcb* structure. On control timeout the value of the persistent queue must be updated considering link overhead and divided by the latest average RTT (5.7), thus obtaining a estimate of the rate required to drain the persistent queue in one control interval.

$$queue\_rate = \frac{queue\_bytes + queue\_p \cdot link\_overhead}{avg\_rtt} \tag{5.7}$$

Calculate current fair-share rate The fair-share rate may be calculated by replacing (5.5), (5.6), (5.7), in (2.17). This results in:

$$rate = rate\left(1 + \frac{\alpha \cdot (capacity - input\_bw) - \beta \cdot queue\_rate}{capacity}\right) \tag{5.8}$$

where both *rate* and *capacity* are variables included in the queue control block *xqcb*. The current implementation does not support the arbitrary rate interval τ, as defined in (2.14), and therefore feedback scaling in (2.17) is not necessary since the control interval T coincides with the average RTT, *d*.

Implementing the RCP algorithm allows insight into some of the finer details overlooked in the theoretical approach to the problem of congestion control. The initial value of the rate to be attributed to flows in particular is not trivial. Since *rate* is constantly updated by multiplying it's previous value by a factor, coding must assert that *rate* is never zero, otherwise throughput will never increase. Choosing *rate* to equal the value of a link's capacity, as we have done, seems more logical, but may lead to some initial jitter as flows are attributed excessive bandwidth. The probability of such an occurrence is inversely proportional to the number of nodes in a path, since it is unlikely that a router's capacity would be considered a bottleneck in such a setting.

Reset variables Data variables used for calculations must be reset so as to accumulate data over a new control interval. Variables reset include *input_bytes*, *packet_num*, *sum_x* and *sum_xrtt*.

Reschedule timer Before returning, the control timeout must reschedule itself using *avg_rtt*, which replaces the value of the former control interval (5.9).

$$ctl\_interval = avg\_rtt \tag{5.9}$$

5.4 Relaying feedback

The receiver, on parsing the RCP header of incoming packets, must relay the *Rate* value as the *Reverse_Feedback* of the next outgoing packet. This differs slightly from standard XCP behaviour. XCP must return as feedback the sum of all received values of *Delta_Throughput* between outgoing packets, since the total number of incoming packets may exceed the number of outgoing packets. RCP in contrast only requires the most recent *Rate* value to be returned to sender as it represents the most up to date calculation of the available capacity.

Changing feedback behaviour required modifications in *xcp_input_callback()*, the function responsible for header parsing contained in *xcp.c*.

5.5 Congestion window adjustments

The feedback loop is complete once the sender adjusts his congestion window to reflect the received feedback rate. Currently, XCP achieves this by overriding TCP congestion control, namely in *tcp_input.c*. The following modifications for RCP support were performed in *tcp_input()*:

Access packet version For differentiated treatment between RCP and XCP, TCP must access the packet's *Version* parameter for inspection. By including the appropriate variable in XCP's control block, *xqcb*, the version identifier may be retrieved at the TCP layer by accessing the XCP tag.

Verify *RTT* The size of the resulting congestion window, *feedback_cwnd*, may be calculated by multiplying the rate contained in the *Reverse_Feedback* field by the current round-trip time estimate (5.10).

$$feedback\_cwnd = Reverse\_Feedback \cdot RTT \tag{5.10}$$

This requires initial caution on establishing connections, as feedback may be received before an estimate of RTT is available. If RTT is zero, the congestion window must not be adjusted.

```
134 struct router_rcp{
135     uint32_t        ctl_interval;
136     uint32_t        avg_rtt;
137     u_long          queue_b;
138     u_long          queue_p;
139     uint64_t        input_traffic_bytes;
140     uint64_t        input_bw;
141     uint64_t        capacity;
142     uint64_t        rate;
143 };
```

Listing 5.1: */sys/dev/xcp/xcp_records.h* with RCP extension.

Modulate congestion window Using the congestion window calculated in (5.10), the outgoing congestion window is defined as the minimum value between *feedback_cwnd* and the maximum permitted window by XCP and TCP, *XCP_TCP_MAXWIN* (5.11).

$$snd\_cwnd = min\,(feedback\_cwnd, XCP\_TCP\_MAXWIN) \qquad (5.11)$$

This concludes all modifications required for the current kernel implementation of XCP to support RCP. Additional modifications on logging tools are briefly mentioned in the next section.

5.6 Logging extensions

Adequate analysis of the performance of RCP requires extensions to the current logging platform provided with XCP. This was achieved by adding a new structure (listing 5.1) to *xcp_records.h* responsible for redirecting relevant data, described in section 5.3, onto a virtual device on control timeout.

5.7 Results

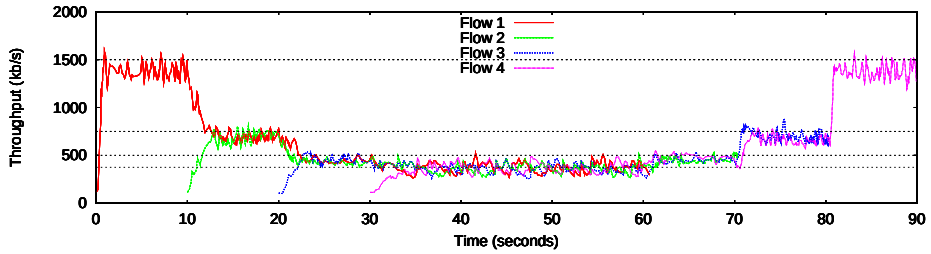
5.7.1 Testbed setup

Our testbed is described in section 3.4.1, maintaining a bottleneck capacity of 1.5Mb/s and a round-trip time of 100ms. Unlike XCP-b, RCP involves changes in both the feedback algorithm and end-host functions, requiring all nodes to be running RCP. The congestion control protocol used could be switched between XCP and RCP on all nodes by manipulating a kernel

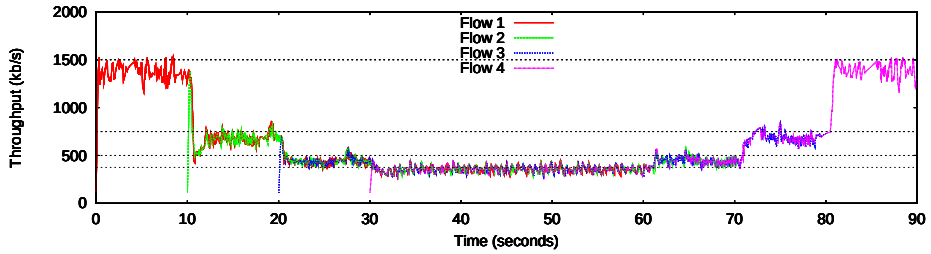
variable accessible through the *sysctl* utility. This allowed us to repeat tests with relative ease for both protocols and consequently compare both algorithms. All tests implement the same values for both α and β, constants used in the calculation of the aggregate feedback. These were once again set to 0.4 and 0.226, respectively.

5.7.2 Test overview

Test results using both XCP and RCP were obtained following the procedure described in section 3.4.1. Figure 5.2 compares the throughput used by end-hosts throughout testing in both scenarios. A preliminary analysis indicates RCP behaves as expected, emulating processor sharing with greater precision than XCP, with increased stability when the number of simultaneous flows remains constant. Additionally, RCP converges to a fair-share rate in a far shorter time frame than XCP, which progressively adjusts the end-hosts' congestion window to reflect changes in flows.



(a)



(b)

Figure 5.2: A comparative overview of end-host throughput using (a) XCP and (b) RCP.

Significant overshoot in adapting to link loading is apparent on few occasions. As the second flow starts, RCP underestimates the available capacity and modualtes end-hosts to a considerably lower throughput rate

before converging to the fair rate, resulting in visible negative overshoot. XCP on the other hand seems unable to adjust throughput appropriately for single flows, as demonstrated when the first flow starts by overshooting.

All these occasions however share a common factor: they occur immediately before or after a time frame in which a single flow was present. Such cases should be treated with caution as they do not accurately portray system dynamics, particularly since routers are dependent on flow statistics to judge available bandwidth, therefore becoming prone to errors in the presence of few active flows. For this reason, our analysis of test results will consider data spanning from the moment throughput has converged to a fair-share rate after the second flow has started up until the third flow is completed.

5.7.3 Fair-share rate estimation

By constituting the only bottleneck for all flows, the RCP router would be expected to dictate the fair-share rate for all end-hosts. When comparing the rate as calculated by the router on successive control intervals, shown in figure 5.3, with the throughput used by end-hosts, shown in figure 5.2b, it becomes apparent that our implementation is consistent with previous simulations. End-hosts constantly adjust their respective congestion windows to reflect the fair-share rate, which remains stable once the number of flows has been correctly estimated by the router.

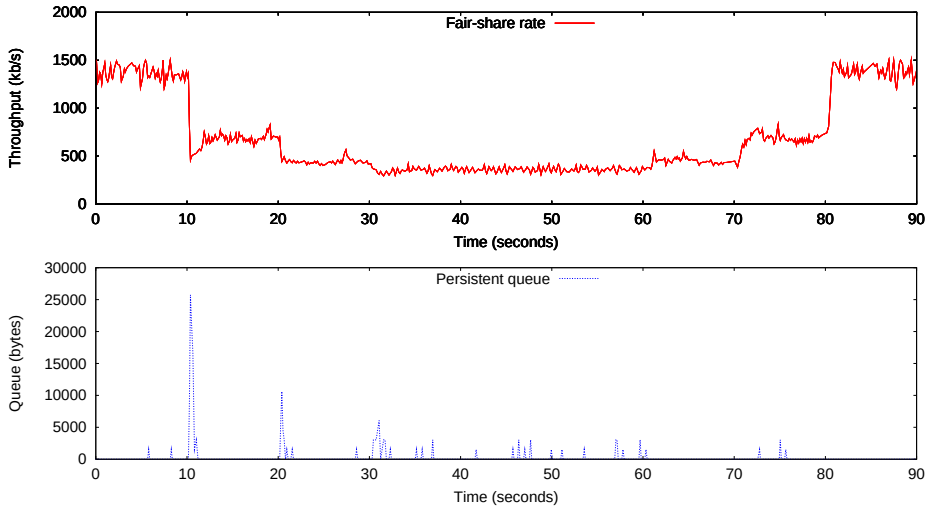


Figure 5.3: Fair-share rate as calculated by the RCP router.

Intuitively RCP has some difficulty in adapting the fair-share rate when faced with an increase of flows. By announcing a rate to all flows whilst maintaining no per-flow state, RCP must deal with the over-allocation of bandwidth when new flows are initiated. This behaviour may be observed in figure 5.3: each new flow causes the fair-share rate to undershoot. Even if we disregard the significant undershoot as the second flow starts, for reasons stated in section 5.7.2, such behaviour is visible with the start of the third and fourth flow, indicating that the router is compensating for over-allocation of bandwidth by draining the accumulated queue.

The spikes in the persistent queue are also of interest. As the second flow starts, RCP will overestimate available capacity at most by 100%, effectively allocating twice the throughput the bottleneck can handle. In reaction, the queue peaks to its highest value, reaching 26KBs. As the third flow starts, RCP once again distributes excessive bandwidth, this time over-allocating up to 50% of the available capacity. Likewise, as the fourth flow starts, RCP distributes up to a third of the capacity in excess. This is relevant in that RCP becomes more capable of sustaining new flows with a rise in the number of estimated flows since impact on queue size diminishes. When coupled with the lack of per-packet calculations, this makes RCP particularly attractive to core routers, which can handle thousands of flows at any given time.

In contrast, a decrease in throughput is adequately handled irrespective of the number of flows: overshoot is non-existent and convergence to the optimal rate value is comparable to the performance demonstrated by XCP in figure 5.2. A more detailed analysis on the effects of active flow increase will be discussed over the next section.

While RCP is visibly fairer than XCP, it is important to quantify this improvement. Measuring fairness can be achieved by using Jain's fairness index [16] (5.12):

$$J = \frac{\left(\sum_{i=1}^{n} \bar{x_i}\right)^2}{n \cdot \sum_{i=1}^{n} \bar{x_i}^2} \tag{5.12}$$

where $\bar{x_i}$ is the average throughput of source i and n is the number of active sources during the interval considered to calculate the index value. We calculate throughput average over 100ms intervals and plot the corresponding results in figure 5.4, limiting our time range to moments with over one flow, between 10 and 80 seconds, since Jain's fairness index is necessarily 1 when only one flow is present. Clearly, RCP is consistently fairer than XCP and, when faced with an increase in flows, converges to fairness in one RTT. It is important to stress that Jain's Index quantifies fairness between flows

and does not reflect utilisation in any measure. Therefore RCP converges to fairness far faster than it converges to a fair-share rate, since the first reflects that all flows share the same rate, whilst the second reflects that all flows share an equal portion of the overall capacity. Also of note is that, by choosing such small intervals for computing the Jain's index, we may be introducing some bias toward XCP. Since XCP distributes bandwidth incrementally it is natural that sources may not share the same instantaneous throughput rate but, over time, will bear similar throughput averages. Had we used an interval larger than RTT we would have seen smaller fluctations in the Jain's index associated to XCP once its value approaches 1.

In this case however we choose to align the interval value with that of the round-trip time in order to gain insight with greater precision on how long each algorithm takes to achieve fairness. Both algorithms achieve fairness but do so in different manners which hint at their subjacent design philosophies. While XCP privileges existing flows as it strives to redistribute bandwidth, an RCP system will consistently benefit short over long-lived flows.

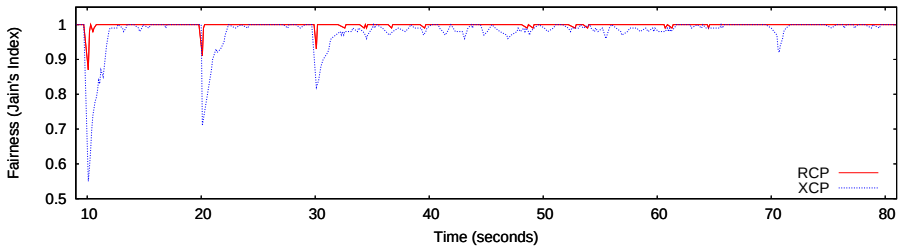
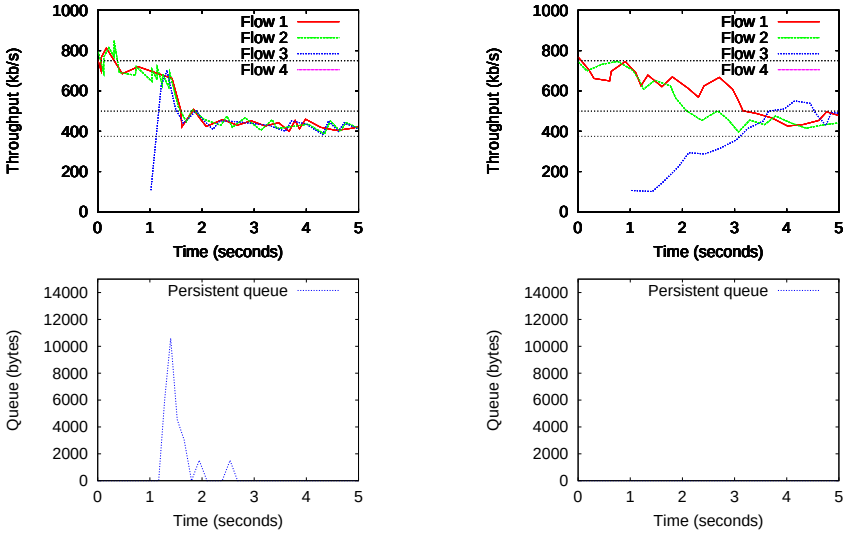


Figure 5.4: Jain's Fairness index for XCP and RCP.

5.7.4 Flow increase

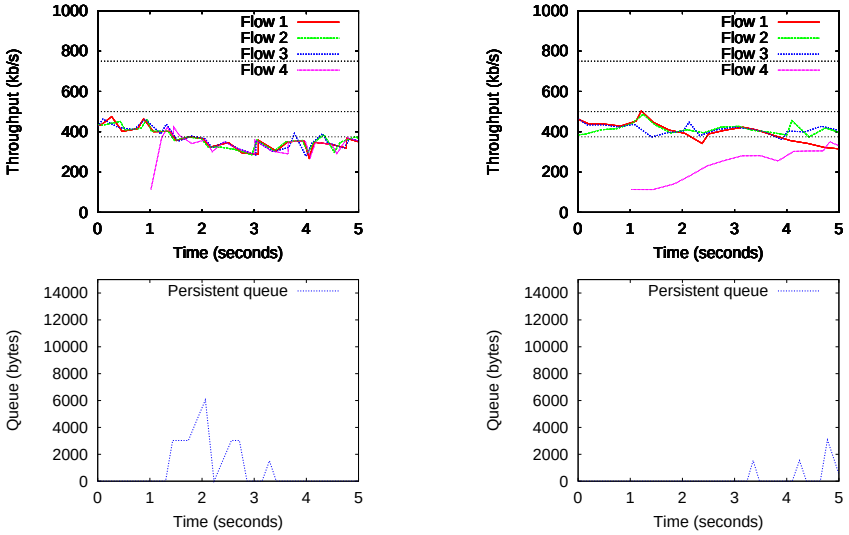
A defining characteristic of a congestion control algorithm is how it responds in the presence of increased demand. RCP allows new flows full allocation of the fair-share rate before adjusting this rate accordingly, absorbing excess throughput in queues along the flow path. While this suits short data transfers, as the congestion window almost immediately achieves optimal size, the subsequent capacity overload increases both network latency and jitter. Such characteristics penalise mostly real-time or interactive applications.

XCP on the other hand only allocates spare bandwidth to new flows. The fairness controller makes such spare bandwidth available, ensuring



(a) (b)

Figure 5.5: The throughput of end-hosts and the router queue as a third flow starts, using (a) RCP and (b) XCP.



(a) (b)

Figure 5.6: The throughput of end-hosts and the router queue as a fourth flow starts, using (a) RCP and (b) XCP.

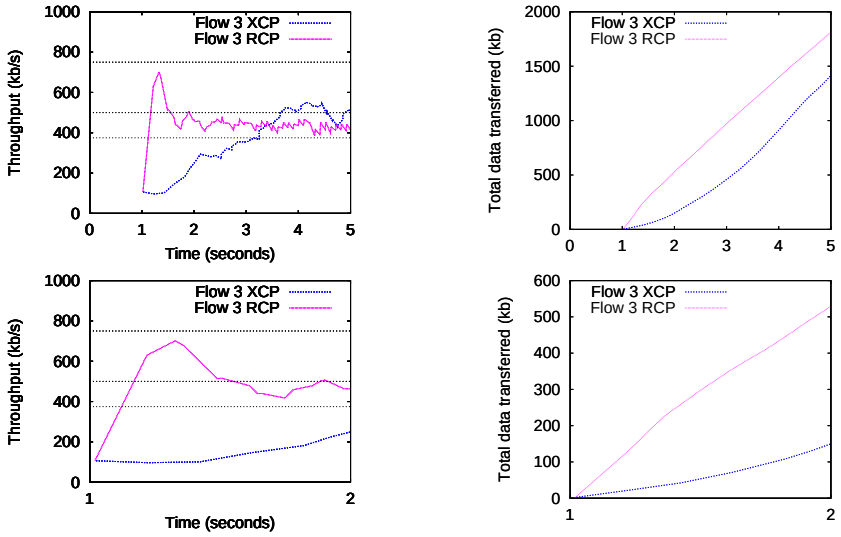


Figure 5.7: The (a) throughput of the third flow as it starts using both RCP and XCP and the respective (b) total data transferred.

that flows converge to the same rate by reducing bandwidth from flows with more than the fair share. As a result, XCP flows are slow in reaching the optimal throughput rate, but do so with no significant queuing. XCP is therefore adequate for long flows requiring low latency.

This is clearly visible in figure 5.5, where both protocols are compared. Using RCP, the flow initiated by client C takes approximately ten RTTs to both converge to the fair-share rate and completely drain the accumulated queue. In contrast XCP takes twice as long to stabilize at the same rate but with no persistent queue size.

Figure 5.6 compares RCP and XCP as the fourth flow starts. RCP shows signs of improved performance over the previous case, which is natural. As the number of flows increase, the net effect of a new flow decreases, resulting in less abrupt change and with a peak queue size considerably inferior than in figure 5.5a. XCP maintains slow convergence with some queuing, but apparently not correlated with the entrance of a new flow.

These results also confirm RCP's competence in short flows. While the previous figures are not ideal in evaluating total throughput, figure 5.7 shows the significant difference in data transferred over the first seconds of a flow's lifetime. Over the first two seconds in particular, corresponding to approximately twenty RTTs, RCP makes better use of the channel capacity

and transfers more data, at the cost of increased queuing. Once both rates stabilise, data transferred increases in a linear fashion with both protocols making full use of the available bandwidth. Over the first second (figure 5.7b), RCP transfers over three times more data than XCP. This performance could however affect overall system stability. RCP allows extremely fast, small flows, whilst needing time to adjust the fair-share rate to reflect the total number of active flows. The effect of a large amount of small flows on the accuracy of the fair-share rate estimate and the size of the persistent queue should be a subject of further study. XCP in contrast is not as affected by bursty traffic since bandwidth is distributed in an incremental fashion.

As an initial approach, our results confirm previous simulations, soundly supporting the validity of our implementation. Extrapolating meaningful conclusions for large networks from such small-scale flow dynamics is however neither trivial, nor appropriate. The observed behaviour raises interesting issues which must be addressed in the near future, in particular quantifying the effect of flows on system response and understanding the consequences that mass fluctuations in active flows have on both system latency and stability. Over the next chapter we attempt to comprehend the limits within such an aggressive protocol as RCP maintains stable.

Chapter 6

Flash crowds in RCP

Testing the implementation of RCP in the previous chapter made clear the underlying design philosophy which focuses on flow-time completion as the most relevant metric for congestion control. By aggressively distributing bandwidth before flows are accounted for however, RCP may become vulnerable when faced with large increases in the number of flows which is recurrent among Internet phenomena. In this chapter we model an RCP system using control theory and attempt to quantify the bounds within such a system may perform in a stable manner when subjected to a flash crowd, whilst verifying our results with experimental data retrieved from simulations using ns-2.

6.1 Modelling flow arrivals

An RCP system can be studied using a fluid model. Following (2.17), and (1) assuming a constant number of flows in the network, (2) considering all flows have the same RTT and (3) ignoring queue boundaries, the set of equations below characterizes an RCP system:

$$F(t) = \alpha \cdot (C - y(t-d)) - \beta \cdot \frac{q(t-d)}{d} \tag{6.1}$$

$$\dot{y}(t) = \frac{F(t)}{d} \tag{6.2}$$

$$\dot{q}(t) = y(t) - C \tag{6.3}$$

where the system delay d can be expressed by the sum of the propagation RTT d_0 and queuing delay:

$$d = d_0 + \frac{q(t)}{C} \tag{6.4}$$

To introduce the effect of the variation of the number of flows, we need to write (6.2) as:

$$\dot{y}(t) = \frac{1 + L(t)}{d} \cdot F(t) + \frac{L(t)}{d} \cdot y(t - d) \tag{6.5}$$

where $L(t)$ represents the growth rate of the number of flows:

$$L(t) = \frac{N(t) - N(t - d)}{N(t - d)} \tag{6.6}$$

To shed some light on (6.5), we should emphasize that in an RCP system flows acquire an instantaneous rate which is composed by the sum of the previous rate and the resulting aggregate feedback F divided by over all pre-existing flows. To account for the global effect new flows have on the variation of input bandwidth we must therefore take into account not only a growth rate in the aggregate feedback, bandwidth being freshly distributed, but also a growth in the previous total bandwidth allocated amongst flows, $y(t - d)$.

Note that we define the growth rate of the number of flows as being normalized to the system delay d. As such it represents the ratio between the number of new flows during an interval of d seconds and the number of active flows in the previous interval.

To understand the limits of RCP, we analyse its behaviour in the presence of a constant growth rate of the number of flows. This means that we consider $L(t) = L$ to be constant and establish steady-state properties and limits as a function of L. For example, considering $L = 0.5$ results in an increase in the number of flows by 50% over each interval of d seconds. As we will show soon enough, L itself influences the system delay d and for that reason we will also define stationary properties of the system as a function of L_0. L_0 is a particular case of L calculated for the network minimum delay d_0 which is constant, allowing us to define flow growth more objectively.

We start the analysis by assuming steady-state conditions of the system represented by (6.5), (6.1), (6.3), (6.4). Steady-state conditions are $y(t) = C$, $\dot{y}(t) = 0$, $L(t) = L$. Under these conditions, we can rewrite (6.5) as:

$$q_c = \frac{C \cdot L}{\beta \cdot (1 + L)} \cdot d \tag{6.7}$$

which, using $d = d_0 + \frac{q}{C}$ from (6.3), results in:

$$q_c = \frac{C \cdot L}{(\beta - 1) \cdot L + \beta} \cdot d_0 \tag{6.8}$$

where d_0 is the network RTT excluding queuing delay at the router. This is an interesting result, assuming that the system is able to achieve steady-state. In the presence of a constant growth rate in the number of flows in the network, the queue length of the bottleneck router will grow to a point where it neutralizes the effect of the arrival of new flows. We call this queue length the *compensation queue* or q_c. The compensation queue q_c required to balance flow growth rate is proportional to the network bandwidth delay product $C \cdot d_0$, and grows with the flow growth rate L, while decreasing with an increase of β. An interesting remark is that the parameter α does not influence the compensation queue. This is somewhat expected as α controls the weight given to the spare bandwidth in the feedback given to the sources. In steady-state the link is fully utilized, thus there is no spare bandwidth. Another interesting conclusion is that the RCP system can only sustain the flash crowd if $(\beta - 1) \cdot L + \beta > 0$. If this condition is not met, q_c will tend to infinity, meaning that utilization will be persistently above the network capacity and the system will be unstable. Fig. 6.1 shows q_c as a function of L, β. The stability limits are shown in the figure as vertical lines.

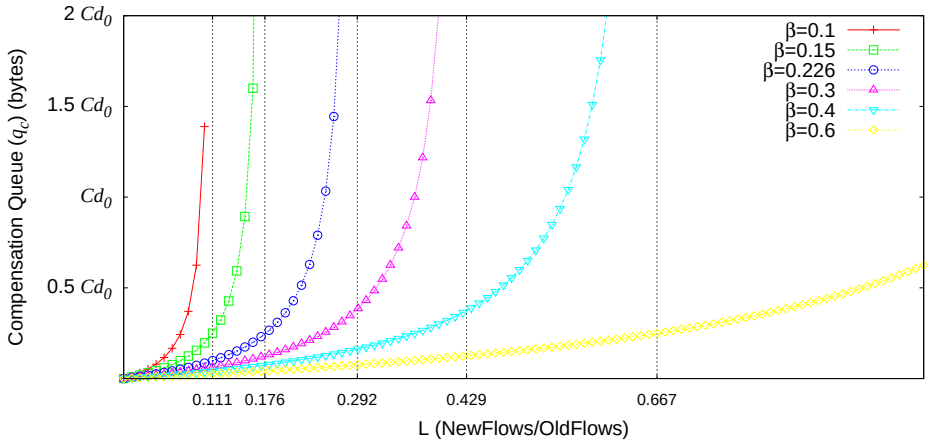


Figure 6.1: The compensation queue q_c required to neutralize a growth rate L of the number of flows for various values of β.

We have seen that the RCP system tries to neutralize the growth of the number of flows by building up the queue, stabilizing queue length up to a certain growth limit. These results also show a more subtle connection.

We have established a relationship between the compensation queue q_c and the growth rate L. The growth rate L, however, is defined as the growth rate of the number of flows each d seconds, while d itself depends of L. This does not allow us to define a constant growth rate. To overcome this problem we define L_0, which has the same meaning as L, but refers to the growth on a fixed interval of d_0 seconds. Additionally, we can represent L as a function of L_0:

$$1 + L = (1 + L_0)^{\frac{d}{d_0}} \tag{6.9}$$

upon simplification:

$$L = (1 + L_0)^{1+\frac{q}{C \cdot d_0}} - 1 \tag{6.10}$$

d_0, as previously stated, is the network RTT excluding queuing delay. Using this new definition of L in (6.8) we obtain:

$$q_c = \frac{C \cdot \left[(1 + L_0)^{1+\frac{q_c}{C \cdot d_0}} - 1\right]}{(\beta - 1) \cdot \left[(1 + L_0)^{1+\frac{q_c}{C \cdot d_0}} - 1\right] + \beta} \cdot d_0 \tag{6.11}$$

and now we have q_c defined only in terms of initial conditions, allowing us to determine q_c for a given constant growth rate of the number of flows. Unfortunately, this equation is not easily reducible to a closed form so we will just leave it as is, solving it numerically. The resulting plot is shown in Fig. 6.2, which exhibits a similar pattern to that of Fig. 6.1. One difference is the marking of stability limits. In Fig. 6.2 the maximum y vertex of each curve corresponds to the highest growth rate L_0 for which RCP is able to absorb the flash crowd.

In conclusion our analysis shows that, within certain limits, RCP is able to stabilize queue length even in the presence of a constant L or, in other words, if an exponential growth of the number of flows occurs. We have shown how to calculate the length at which the queue stabilizes given a growth rate L_0, a network minimum RTT of d_0, a link capacity C, and the β parameter of RCP. Likewise, we have shown how to calculate the maximum growth rate L_0 which RCP is able to sustain whilst remaining stable.

6.2 Response to typical arrival distributions

We have derived steady-state properties and conditions as a function of a constant growth rate of the number of flows L_0. As such we can calculate

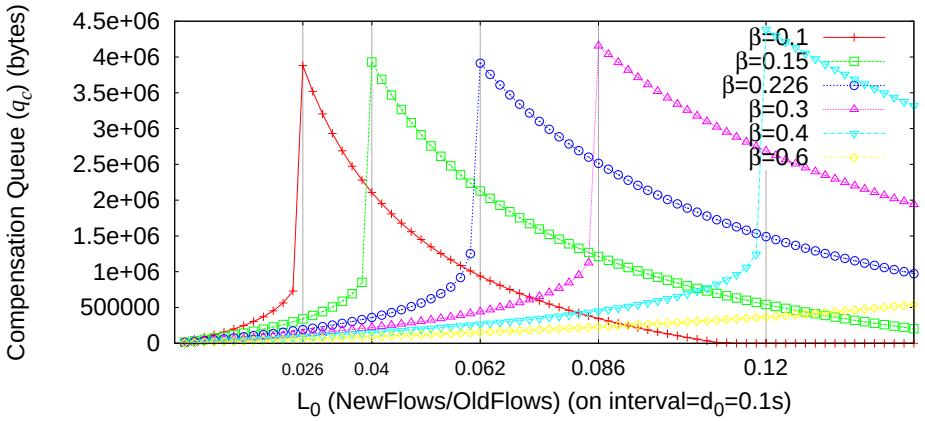


Figure 6.2: The compensation queue q_c required to neutralize a growth rate L_0 of the number of flows for various values of β. This refers to the particular case of $C = 100$ Mbit/s and $d_0 = 0.1$ s.

the compensation queue q_c if the number of flows in the network grows by a factor of $(1 + L_0)$ in each interval of d_0 seconds - an exponential increase. It is equally interesting to understand how an RCP system responds to other types of growth of the number of flows, namely in the presence of typical flow arrival distributions. To this end, we need to find how $L_0(t)$ behaves for these distributions. We analyse $L_0(t)$ for 3 types of flow arrival distributions: Laplace, Normal, and Erlang. The Laplace and the Normal distributions refer to the case of scheduled events, e.g. sports match, where arrivals may start before the event. The Erlang distribution refers to the case of unplanned events, e.g. *Slashdot* article, where there is a strong ramp-up reaction shortly after the event occurs, which then fades away in time. The probability density function (PDF) of the Laplace distribution is defined as:

$$f(x) = \frac{1}{2b} \cdot e^{-\frac{|x-\mu|}{b}} \tag{6.12}$$

the PDF of the Normal distribution is defined as:

$$f(x) = \frac{1}{\sigma \cdot \sqrt{2 \cdot \pi}} \cdot e^{-\frac{(x-\mu)^2}{2 \cdot \sigma^2}} \tag{6.13}$$

the PDF of the Erlang distribution is defined as:

$$f(x) = \frac{\lambda^k \cdot x^{k-1} \cdot e^{-\lambda \cdot x}}{(k-1)!} \tag{6.14}$$

where x represents the arrival time. In Fig. 6.3 we plot the evolution of $L_0(t)$ over time for some cases of the three distributions. Those plots are obtained for flash crowds of 5000 flows, and considering $d_0 = 0.1$s. Also, the initial number of flows in the system, i.e. before the flash crowd, is set to 1. The results obtained can be generalized for a flash crowd with any number of flows, as long as the ratio between the number of flows of the flash crowd and the initial number of flows in the network is kept constant. Analysing $L_0(t)$ for a PDF of an arrival distribution allows us to infer the queue response to that PDF. Queue length will follow $L_0(t)$ dynamics if $L_0(t)$ is below the stability limit (shown in Fig. 6.2), however if $L_0(t)$ is above the stability limit, then the queue length will increase exponentially. We will see this in more detail in the next section.

6.3 Simulation Results

The purpose of our simulation results is twofold: we wish to both 1) validate the theoretical limits extracted from the model presented in the previous section as well as 2) understand the limitations such a model has in fully representing an actual RCP system. To this end, we present results performed with ns-2 using our own implementation of the RCP algorithm based on the existing XCP source code included in the ns-2 package. The setup, shown in 6.4, is composed of wired nodes connected to a sink S via a router, R. To ensure the same bottleneck is shared across all flows, nodes connected to R have twice the bandwidth available between R and S, which was set at 100Mbit/s. The propagation delay, d, was set to 25ms unless otherwise stated, resulting in a total round trip time of 100ms for each flow.

Since our main emphasis is on understanding queue dynamics under a sustained increase of flows, we first populate the system with flows from nodes M_i to the sink S. This allows the system to both stabilise the flow rate attributed to every flow and drain the queue, which naturally builds up as the first flows enter the network. The number of initial nodes m is the minimum value of flows to which the growth rate L can be successfully applied, obtained using $m = round\left(\frac{1}{L} - 1\right)$. Once the queue has been depleted, the node responsible for simulating the flash crowd effect, F, initiates flows at the desired rate.

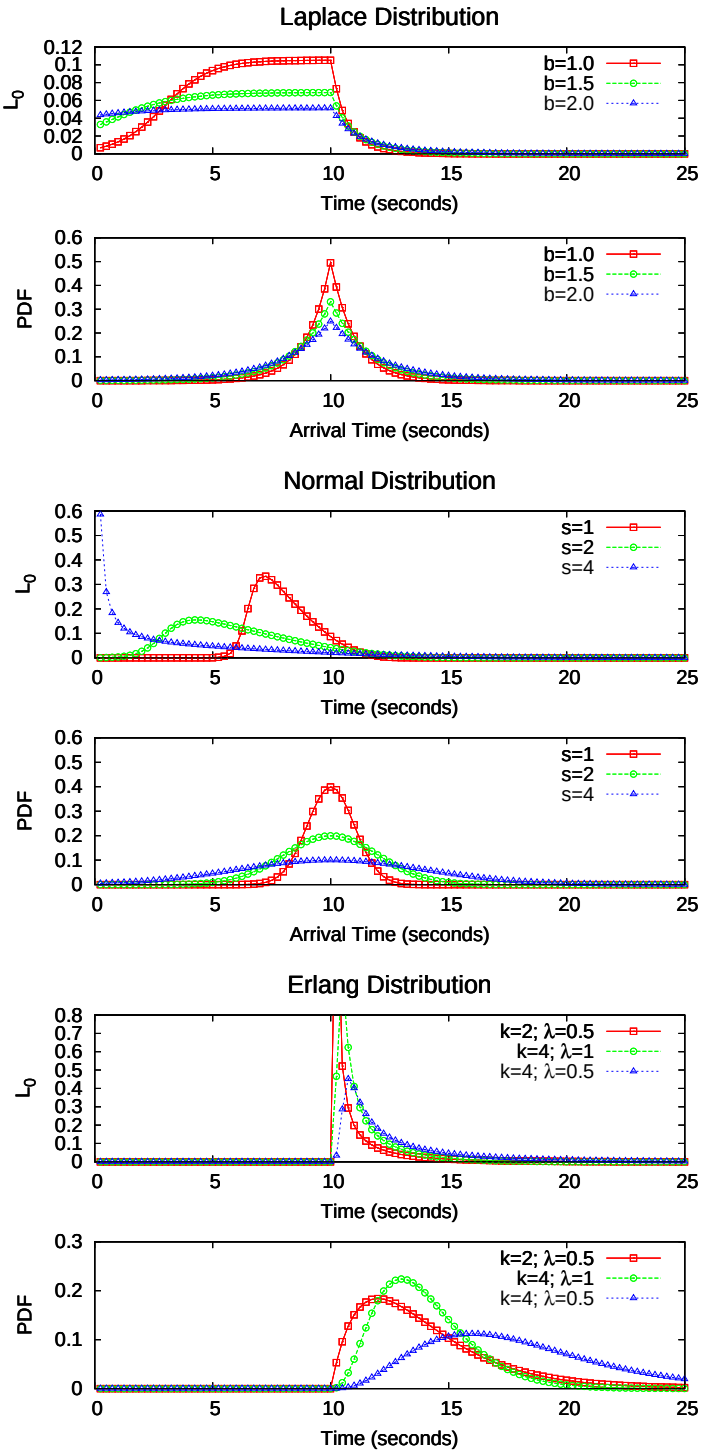


Figure 6.3: The relationship between L_0 and the PDF of the Laplace, Normal and Erlang distributions.

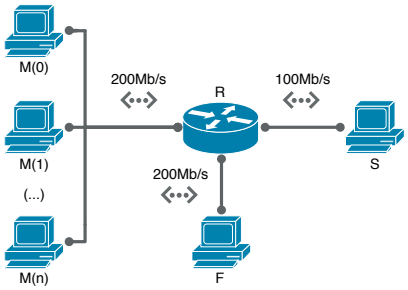


Figure 6.4: Simulation setup

6.3.1 Model Validation

We begin by validating the relationship between the compensation queue q_c and L_0 present in (6.2). We run a set of experiments for several values of L_0, β. In these experiments we use $\alpha = 0.4$. Fig. 6.5 plots the values obtained through ns-2 simulation overlapped with the theoretical values. The values obtained through simulation are represented by lines with points, while the theoretical results are shown in simple lines. The simulation results support that our analysis is valid and accurate. The bottom plot of Fig. 6.5 represents the queue length dynamics over time for various L_0 and $\beta = 0.226$, where we observe that the queue length converges to a vicinity of the value we have derived in the analysis. The plot above, shows the compensation queue required for a given pair of L_0, β. The curves obtained through simulation mirror those obtained through the analysis with only a small error.

6.3.2 Laplace Distribution

Finally, we analyse how RCP adapts to a surge in the number of flows following a Laplace distribution, as described in (6.12). We centre the peak of the Laplace distribution at instant $t = 20$ s and experiment with various values of the scale parameter b of the Laplace distribution. b controls the degree of concentration of flow arrivals around $t = 20$ s, being that for $b = 1$ arrivals are more concentrated than for $b = 2$. 5000 flows are injected in the network for each simulation run and RCP was configured with $\beta = 0.226, \alpha = 0.4$. The resulting figures (Fig. 6.6) show how $L_0(t)$ evolves over time, and also the consequent evolution of the queue length. It is observable that Laplace distributions tend to produce $L_0(t)$ function that converge to a constant value. The queue length follows the dynamics of $L_0(t)$ if $L_0(t)$

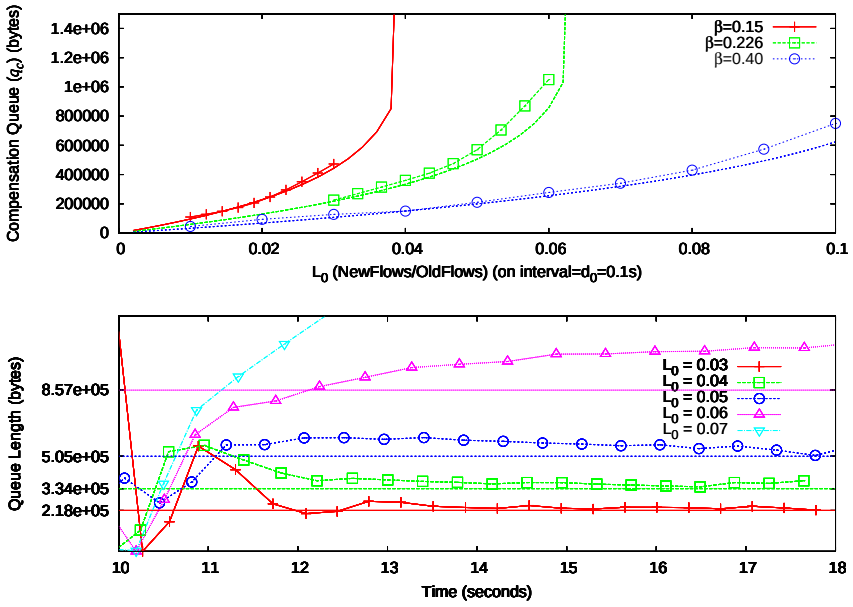


Figure 6.5: **Above)** The compensation queue q_c as a function of L_0 for various β. The theoretical values are plotted with lines without points. **Below)** Queue length vs. time for $\beta = 0.226$ and a range of L_0. The compensation queue theoretical values are represented by horizontal lines.

stays below the stability limit, which for this case is $L_0 \approx 0.062$ (marked in the figure as an horizontal line). If $L_0(t)$ goes above the stability limit, then the queue will grow exponentially, as happens in this experiment for $b = 1, b = 1.5$. For $b = 1.5$, $L_0(t)$ tends to 0.068 which is only slightly above the stability limit. For this reason the exponential growth of the queue in this case is timid.

6.3.3 Normal & Erlang Distributions

Finally, we analyze how RCP adapts to a surge in the number of flows following Normal and Erlang distributions (Eq. 6.13, and Eq. 6.14, respectively). We center the peak of the Normal distribution at $t = 10$ s, while the peak of the Erlang distribution varies between 5 and 10 s. We also vary the parameters of the distributions that regulate the concentration of flow arrivals. The number of injected flows was, again, 5000. The results, shown in Fig. 6.7 (Normal), and Fig. 6.8 (Erlang), indicate that both distributions produce periods of higher acceleration of the number of flows than the Laplace distribution. The value of $L_0(t)$ required to inject 5000

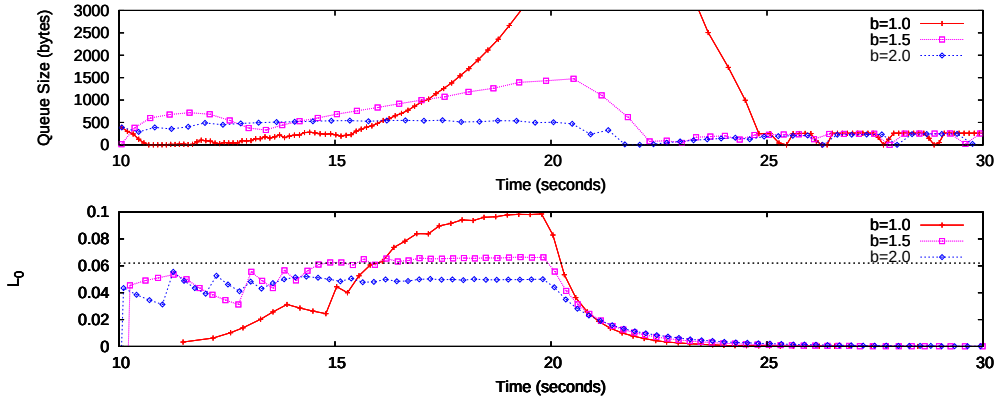


Figure 6.6: **Above)** The queue response to flash crowds following Laplace distributions with $b = 1, b = 1.5, b = 2$. **Below)** The growth rate $L_0(t)$ of the number of flows throughout time.

flows easily surpasses the stable limit of $L_0 < 0.06$ for the tested scenarios. For the Normal distribution, queue build-up is conservative before L_0 hits the threshold, after which the queue increases exponentially. Due to the extent of queue growth, figure 6.7 does not adequately convey the system's state the stability limit has been breached due to the differences in scale between the queue in its stable and unbound condition.

Surpassing the upper-bound limit is even more worrying for the Erlang distribution as it may even cause large spikes of $L_0(t)$ right at the beginning of the flash crowd. The periods of high acceleration experienced in these distributions, even if only for a short time, prove to be much harder to control by RCP, than the continued acceleration experienced by flash crowds following Laplace distributions. Whenever $L_0(t)$ goes above its stable limit, the queue length grows exponentially causing system delay to increase accordingly. For the system to recover from this unstable period, $L_0(t)$ must dive well below the initially established stability limit, because that limit was valid assuming a much lower base delay. How low $L_0(t)$ must go after an unstable period, depends on how much the system delay has grown during the period of instability. This fact is most clearly observable in the results from the experiment with a Normal distribution of flow arrivals (Fig. 6.7), where the longer the unstable period is, the lower $L_0(t)$ must go before queue length starts to decrease.

Another aspect of these experiments that stands out, is the high value of the queue length (in the order of tens of Mbyte) obtained when the stability limit of L_0 is breached. In our experiments we did not limit the

size of queue length, mainly because our objective was to validate the mathematical model. However, we do not expect real systems to have such large buffers. If the stability limit is breached, massive packet loss is expectable and additional measures are required to guarantee decent network performance. Such measures might include adopting some sort of admission control mechanism, or increasing the value of the parameter β of the RCP controller, whenever $L_0(t)$ crosses to the unstable region.

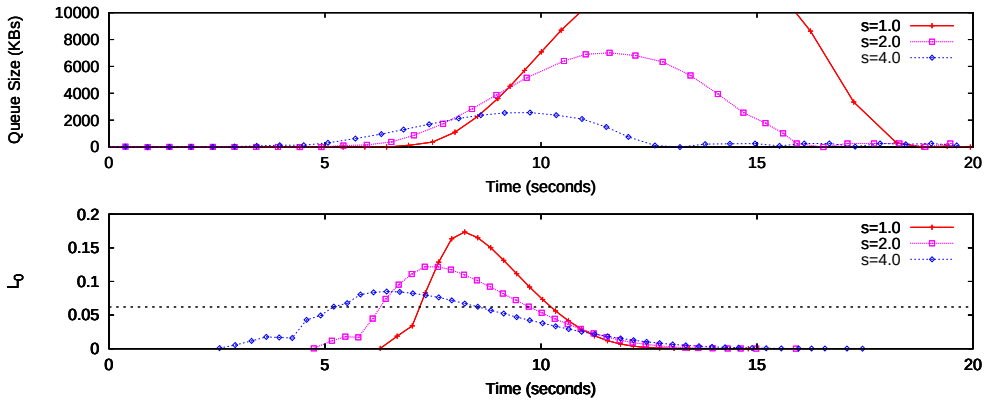


Figure 6.7: **Above)** The queue response to flash crowds following Normal distributions with $s = 1.0, s = 2.0, s = 4.0$. **Below)** The growth rate $L_0(t)$ of the number of flows throughout time.

We have studied the effect that the persistent and significant increase of the number of flows has in an RCP system. By introducing the variation of the number of flows in the differential equations that characterize the behaviour of RCP, we were able to determine properties of the steady state of RCP. We found that RCP is able to stabilize queue length if the growth rate $L_0(t)$ of the number of flows does not exceed a certain limit. The queue length required to stabilize the system is proportional to the BDP of the network and decreases with β. The maximum growth rate for which RCP is stable is obtained by identifying the maximum of Eq. 6.2. With these results, the designer of an RCP system is better prepared to choose RCP parameters and also to predict the system response in the presence of a flash crowd. Additionally, we have the studied how the growth rate $L_0(t)$ behaves for the case of three typical arrival distributions: Laplace, Normal, and Erlang. Flash crowds following a Laplace distribution have shown to be the easier to control by RCP, while those following an Erlang distribution where more prone to drive the system to instability - assuming the flash crowds are composed by the same number of flows and have similar

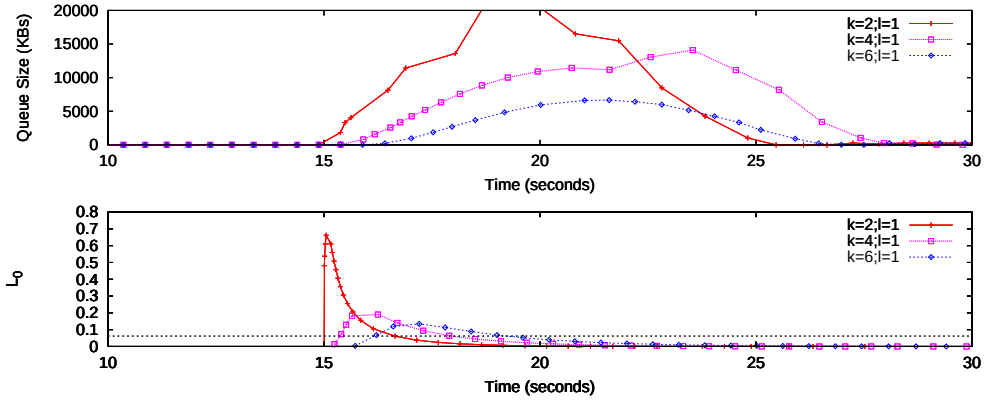


Figure 6.8: **Above)** The queue response to flash crowds following Erlang distributions with $k = 2, k = 4, k = 6$ with $l = 1$. **Below)** The growth rate $L_0(t)$ of the number of flows throughout time.

duration. It remains to be understood how realistic large, constant growth rates are. For $\beta = 0.226$, we have shown RCP system can withstand a growth rate of $L_0 = 0.062$. At this growth rate, after only 20 seconds, there is an increase of almost 170,000 times the original number of flows. How frequently such situations occur is not clear or documented and would be of interest in order to shed some light on how well suited RCP is in a real deployment setting.

Chapter 7

Conclusion

This dissertation has documented the design, implementation and evaluation of explicit congestion control algorithms with the intent of aiding the transition of such models from theory to practice. By extending explicit congestion control to perform over media for which it was not originally designed and providing an analytical model which can predict system response for worst-case scenarios, we attempt to scale explicit congestion control to all facets of the Internet, a network increasingly pervasive and heterogeneous at the edges with ever-higher performance at its core.

In this final chapter we draw conclusions on each of our proposed goals, presenting our achievements, shortcomings and suggestions on future work on congestion control.

Deploying explicit congestion control in variable-capacity media Explicit congestion control protocols struggle to perform efficiently over variable-capacity media as they rely heavily on a precise estimate of link capacity. We implement XCP-blind, an alternative algorithm which attempts to eliminate the correlation between link utilisation and such knowledge.

Initial testing presents promising results. XCP-b provides similar performance to XCP in a wired testbed, constituting a sound proof-of-concept for the use of queue oscillation as an indicator of network congestion. We interpret system response to flow variation by establishing the effect, and indirect implications, of control parameters. Whilst we conclude modifying such parameters is dependent on application requirements, we provide general guidelines on achieving specific patterns in behaviour. Results on a wireless testbed show XCP-b provides increased stability in both flow throughput and latency. From here we study how the queue buffer can be dynamically adapted to link loading so as to provide greater robustness to flow variance. The resulting algorithm performs according to our initial

expectations despite the limited nature of our testbed. Finally we design an alternative approach in queue inference of link capacity by estimating the accumulated error which approaches the real capacity of link media. Our results show an algorithm which is particularly useful when we have an approximate value for capacity. This is the case of media with relatively stable capacity but where the exact value is not known.

Both protocols are tested and shown as proof-of-concept, since more work is still required in accurately modelling algorithm behaviour and improving performance further. In particular, stability proof for ErrorS is lacking and further research into appropriate values for control parameters is necessary. Finally, we believe these changes in capacity estimation should be applied to RCP, where jitter is recurrent, and performance comparisons with XCP-b should be drawn.

Implementing alternative explicit congestion control protocols As TCP becomes increasingly unable to cope with existing requirements in congestion control, new approaches for achieving flow-rate fairness emerge.

We study, implement and analyse RCP, comparing the algorithm's performance with that of XCP. Test results are consistent with previous simulations, proving RCP rapidly converges to a fair-share rate at the expense of increased, erratic network queuing. While the use of such a protocol seems appropriate for small transfers, such as caching web pages, we suggest further testing in order to comprehend the implications a significant amount of intermittent bursty flows has on network latency and stability.

Our approach in implementing RCP draws on the overarching concepts it shares with XCP. We suggest separating the architecture for explicit congestion control from the underlying algorithms which dictate its dynamic response. Future work should focus on how different algorithms within this framework can coexist over the same network. Both XCP and RCP have significant, albeit different, merits and the choice between either should be scaled to user needs rather than imposed by network infrastructure. The existence of such an explicit congestion control layer providing a suite of interoperable algorithms would harness a far greater momentum in deployment than introducing each protocol individually.

Modelling the effects of large flow dynamics on aggressive explicit congestion control algorithms Analysis of RCP behaviour portrayed an aggressive algorithm, optimized for fast flow completion time at the expense of queue build-up. Upon realizing the impact of flow arrival diminishes as the number of flows increases, we set out to explore how RCP

copes with a constant increase in flows, thus exploring the limits within which the algorithm performs in a stable manner. While this objective was not part of our initial proposal it was a natural consequence of testing RCP and trying to establish the effect of flow arrival on the persistent queue.

The results presented are interesting since they are not immediately intuitive: we show RCP is able to withstand an infinite continuous growth with a constant queue within well defined limits. Such performance is particularly useful for core routers, which must cope with thousands of simultaneous flows which vary considerably over time. We also study the effect of different arrival distributions on RCP system stability, which is particularly useful for defining traffic admission policies which guarantee limited queueing.

Closing Remarks This dissertation analyses explicit congestion control performance not only from the edge, where the last hop tends to be performed over variable-capacity media, but also from a core perspective, where an increasing load must be processed with few performance trade-offs. Despite the fact the initial proposal focused on implementing explicit congestion control in order to carry out testing in real-life environments, the feedback from such work quickly worked its way upstream into the design process, allowing us to explore new approaches with the intention of further diversifying the context within such protocols can perform effectively. The strict time constraints within which this dissertation was bound ultimately limited the extent to which each subject was explored. As such, this dissertation contains some algorithms which should be viewed as proof-of-concept and are notably missing stability analysis and deeper parameter recommendations aimed at aiding potential system designers. Ultimately we feel this is an acceptable trade-off for the quantity of inroads developed in further understanding the potential and versatility of explicit congestion control algorithms over the increasingly diverse and unique group of networks that currently compromise the Internet.

Bibliography

[1] V. Jacobson and M. Karels, "Congestion Avoidance and Control," tech. rep., ACM Computer Communication Review, August 1988.

[2] I. S. Institute, "Transmission Control Protocol - Darpa Internet Program Protocol Specification." RFC 793, September 1981.

[3] J. H. Saltzer, D. P. Reed, and D. D. Clark, "End-to-end arguments in system design," *ACM Transactions on Computer Systems 2*, vol. 4, pp. 277–288, November 1984.

[4] A. Falk, D. Katabi, and Y. Pryadkin, "Specification for the Explicit Control Protocol (XCP) - draft-falk-xcp-02." Internet-Draft, November 2006.

[5] N. Dukkipati and N. McKeown, "Processor Sharing Flows in the Internet," Tech. Rep. TR04-HPNG-061604, Stanford University, June 2004.

[6] IEEE, "802.11 Part 11: Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) Specifications," 1999.

[7] F. Abrantes and M. Ricardo, "XCP for shared-access multi-rate media," *ACM SIGCOMM Computer Communication Review*, vol. 36, pp. 27–38, July 2006.

[8] E. Kohler, M. Handley, and S. Floyd, "Datagram Congestion Control Protocol (DCCP)." RFC 4340, March 2006.

[9] J. Postel, "User Datagram Protocol." RFC 768, August 1980.

[10] K. Ramakrishnan, S. Floyd, and D. Black, "The Addition of Explicit Congestion Notification (ECN) to IP." RFC 3168, September 2001.

[11] "XCP @ ISI." http://www.isi.edu/isi-xcp/.

[12] D. Clark, "Window and Acknowledgement Strategy in TCP." RFC 813, July 1982.

[13] Y. Zhang and M. Ahmed, "A Control Theoretic Analysis of XCP," in *Proc. of IEEE GLOBECOM*, March 2005.

[14] Y. Su and T. R. Gross, "WXCP: Explicit Congestion Control for Wireless Multi-hop Networks," in *IWQoS*, pp. 313–326, 2005.

[15] IEEE, "802.3 Part 3: Carrier sense multiple access with collision detection (CSMA/CD) access method and physical layer specifications," March 2002.

[16] R. Jain, D. Chiu, and W. Hawe, "A quantitative measure of fairness and discrimination for resource allocation in shared computer systems," 1998.