A Network Congestion Control Protocol to More Quickly Finish Flows

Abstract

The transmission control protocol (TCP) is the major transport protocol in the Internet. TCP and its variants have the drawback of not accurately knowing rate share of flows at bottleneck links. Some protocols proposed to address these drawbacks are not fair to short flows, which are the majority of the Internet traffic. Other protocols result in high queue length and packet drops which translate into a high average flow completion time (AFCT).

In this paper we present the design and analysis of a Quick congestion Control Protocol (QCP). QCP can quickly give flows their fair share rates hence allow them to quickly finish. Unlike existing schemes, QCP uses an accurate formula to calculate the number of flows sharing a network link. This enables QCP to get fair share rates to flows without over or under-utilization of bottleneck link capacities. We also present an efficient sharing mechanism which QCP uses to assign capacity which is not used by some flows bottlenecked elsewhere to other flows which need the capacity. We show how QCP can be implemented by extending the emerging OpenFlow architecture. Simulation results confirm the design goals of QCP in achieving reduced AFCT (by upto 30%).

Categories and Subject Descriptors C.2.2 [*Network Protocols*]: Transport Protocol Design

General Terms Computer Systems Organization

Keywords Congestion, fairness, efficient sharing, fast download

1. Introduction

General Internet and data center traffic is dominated by a large number of small flows (mice) and a few number of large flows (elephant) [1, 3, 34]. For certain small size time sensitive requests (flows) in applications such as financial and database transactions, even a small increase in average flow completion time (AFCT) is significant. Multimedia (streaming) applications also require a more smooth, fair and predictable rate allocation. Video streaming services such as YouTube and Netflix require an average download rate which is slightly larger than video encoding rate [35]. Otherwise if elephant flows significantly delay the periodic transfer of fixed size blocks (64KB for instance) of video content [35], the video playback may interrupt due to empty buffers. So the AFCT of flows of specific sizes becomes an important performance metric as also discussed in [9]. Hence an efficient congestion control protocol should able to achieve small AFCT for majority of network flows.

The majority of network traffic uses the transmission control protocol (TCP) [17] as a congestion control protocol. TCP was successful managing congestion in the early stages of the Internet and before the emergence and vast expansion of other types of network and networking technologies. With the growth of Internet and network technologies TCP encountered many performance problems [14, 22]. As discussed in [22], a random packet loss can for instance result in a significant TCP throughput degradation over high bandwidth-delay product networks. TCP is also not fair to short flows (mice) which are the majority of network traffic as few big size (elephant) flows can significantly delay many mice flows [1, 34, 36].

There have been numerous research efforts to deal with the weaknesses of TCP. The current modifications to TCP such as HighSpeed TCP [11] inherit the main problems of TCP in not quickly knowing the bottleneck link share of flows. This results in flows taking longer to finish than necessary [9].

The eXplicit congestion Control Protocol (XCP) [18] is designed to achieve full link utilization and hence high per flow throughput. However, XCP is not fair to short flows (mice) resulting in higher average flow completion time (AFCT) [9]. The Rate Control Protocol (RCP) [9] on the other hand was designed to finish flows quickly. But as shown later in this paper, RCP under or over estimates the number of active flows which it needs to obtain the rate at which flows send packets. This results in under or over utilization of bottleneck link capacity which in turn results in high queue length, packet drops and high AFCT.

To overcome the drawbacks of existing congestion control schemes and hence finish the majority of network flows more quickly, we present the design and analysis of a Quick congestion Control Protocol (QCP). The QCP approach derives a simple and effective congestion control rate metric which routers calculate and at which sources send data. Unlike TCP, this rate metric can quickly obtain a very high link utilization and a low queue size and hence results in smaller AFCT. It can also achieve fairness (equal fair and proportional fair) among all flows quickly unlike XCP. QCP also uses an accurate derivation of the number of active flows and hence doesn't suffer from such estimation errors of RCP. QCP can be easily implemented using (extending) the OpenFlow [26, 27, 32] architecture.

QCP can emulate processor sharing (PS) [31] by dividing available link capacity fairly among flows. The PS scheme may not allocate a link capacity, unused by flows which are bottlenecked at other links, to other (local) flows which need the capacity. This is because in a multi-bottleneck network scenario, a traditional PS scheme does not have a good way to find out whether or not an active flow is bottlenecked locally or at another link. To deal with this drawback, we design an *Efficient Sharing* (ES) scheme by extending the QCP approach. The ES algorithm achieves this efficiency by treating each flow bottlenecked at other links (not locally) as a *fraction* of a flow.

Previous results [9] have shown how RCP outperforms XCP [18] and TCP. The simulation results we present in this paper show how QCP outperforms both XCP and RCP.

The main contributions of this paper are as follows.

- We show that the performance of the Rate Control Protocol (RCP) degrades with network congestion.
- We propose a Quick congestion Control Protocol (QCP), which is a novel congestion control scheme, that overcomes the weaknesses of existing congestion control schemes to achieve reduced flow completion time and high link utilization.
- We implement QCP in the NS2 simulator and present results which show how QCP outperforms RCP and XCP.
- We introduce a new resource sharing scheme called Efficient Sharing (ES) and generalized ES (GES). ES can be more efficient than the traditional Processor Sharing (PS) in utilizing unused network resources without the need of multiple queues and complex schemes for flows. We show that QCP is an ES and GES protocol.
- We show how QCP can be implemented using (extending) the emerging OpenFlow [32] architecture which is currently being deployed in the backbone of major enterprise networks such as Google.

The rest of the paper is organized as follows. We first discuss more related work in section 2. The QCP algorithm is explained in section 3. In section 4 we present the derivation of the QCP rate. Sections 5 and 6 show how QCP can achieve processor sharing and even more efficient sharing (ES) than the traditional processor sharing (PS) [19]. The QCP packet header format is presented in section 7. We extend QCP to be a weighted fair share allocation in section 8. The stability of QCP is shown in section 9. We show how QCP can be gradually implemented in section 10. A description of how QCP can co-exist with TCP and TCP like algorithms is presented in section 11. After validating the performance of QCP using simulation in section 12, we give a brief summary in section 13.

2. Related Work

In this section we discuss previous work on congestion control protocols. We start with TCP and its variants and then analyze the major clean-slate congestion control protocols.

2.1 TCP and its Variants

The performance limitations of TCP over high bandwidth delay product networks has been reported in [22]. They showed that a random packet loss can result in a significant throughput degradation. The same paper also shows that TCP is unfair towards flows with higher round trip delays. TCP is also not fair for short-lived flows as shown in [14] as the bottleneck bandwidth is dominated by long-lived flows whose window size has grown so large. As has been extensibly reported in the literature [7], TCP is also not suitable for wireless networks. The main reason is that TCP assumes that all packet losses are due to network congestion while in the case of wireless networks it can be due to some wireless link errors which may soon correct themselves. TCP also either under-utilizes or over-utilizes the network bandwidth resulting in a download time much longer than necessary as discussed in [9].

The current modifications to TCP inherit the main problems of the original TCP and have not properly addressed the main challenges. The Datagram Congestion Control Protocol (DCCP) [20] which is primarily designed to replace the User Datagram Protocol (UDP) whose unreliable nature can cause congestion collapse is for instance based on the TCP algorithm. There are also many other variants of and modifications to TCP [4, 6, 11, 13]. Nonetheless these modifications of TCP inherit the basic limitations of TCP of not quickly knowing the bottleneck link share of flows in spite of some improvements over the original TCP. They mainly rely on packet loss and packet delay as congestion signals. Hence, they take longer than necessary to fully utilize the link capacity and to achieve fairness among flows. This in turn results in higher average flow completion (download) time (AFCT).

2.2 Major Clean Slate Protocols

In this section we discuss how QCP differs from RCP and XCP which are the two other major clean slate congestion control protocols.

2.2.1 On Performance of RCP

The rate update equation of the rate control protocol (RCP) [9] is given by

$$R(t) = R(t-d) + \frac{(\alpha(C - y(t)) - \beta\frac{q(t)}{d})}{N(t)} \quad (1)$$

where d is a moving average of the RTTs measured across all packets, $R(t-d)$ is the last (previous) updated rate, C is the link capacity, $y(t)$ is the measured input traffic rate during the last update interval, $q(t)$ is the instantaneous queue size, $N(t)$ is the router's estimate of the number of ongoing flows (i.e. number of flows actively sending traffic) at time t and α, β are parameters chosen for stability and performance.

In RCP and in the rate control protocol with acceleration control (RFC-AC) [10], the number of ongoing flows, $N(t)$ is estimated as

$$N(t) = \frac{C}{R(t-d)}. \quad (2)$$

But this is a heuristic estimate and is where the major limitation of RCP lies.

RCP either over-estimates or under-estimates the allocated rate $R(t)$. When the initial value of $R(t-d)$ from which $N(t)$ is obtained is too small, then $N(t)$ is too large . This in turn results in the router unnecessarily dividing the capacity into too many flows resulting in link under-utilization. Let's consider an initial rate of $R(t-d) = C/200$ whose corresponding $N(t) = 200$. If the link receives only 40 flows/sec for an RTT of 0.1 sec, we have an actual number of 4 flows. If the router allocates each of these flows only $C/200$ bytes/sec, then the total arrival rate for the next round becomes $C/50$ bytes/sec which is 1/50 of the available link capacity. In this case RCP significantly under-utilizes the link capacity.

On the other hand if the initial value of $R(t-d)$ is too large, then $N(t)$ becomes too small. As a result the router divides the capacity into fewer number of flows and hence over-estimates the rate allocation. This causes link over-utilization, more queuing delays and packet losses. In fact, the simulation setup of RCP uses a big buffer capacity (to try to deal with this).

For example, let the initial sending rate $R(t-d) = C/4$. Then the corresponding $N(t) = 4$. If the flow arrival rate is 200 flows/sec for an RTT of 0.1 sec, the actual number of flows is 20. The router then tells each of these 20 flows to send at the rate of $R(t-d) = C/4$. If they all send at this rate, then the total arrival rate $\Lambda = 20C/4 = 5C$. Hence the link receives 5 times more packets than it can handle.

2.2.2 On Performance of XCP

The fact that XCP is not fair to short flows (flows with small data to send) makes its average flow completion (download) time (AFCT) much higher than TCP as shown in [9]. For example, let us consider three short lived flows (mice) which just started with a congestion window size of 1 packet and need to send 50 packets each and one long lived flow (elephant) which needs to send 500 packets and already reached a congestion window size of 60 packets. Without loss of generality let's assume that they all have the same round trip time (RTT). If the spare link capacity is 20 packets per RTT, then XCP shares it equally among all four flows allowing each flow to increase its congestion window by 4 packets per RTT. This implies that the window size of the three short lived (mice) flows is now set to 5 packets per RTT. Hence, it takes 50/5 = 10 rounds (RTT) to download each of the short lived flows and hence a longer AFCT for most of the flows. But QCP can reduce this FCT of majority of the flows by dividing the entire link capacity (say 60+20 = 80 packets/RTT) equally among all four flows. This implies that each flow sets (resets) its window size to 80/4 = 20 packets per RTT. This implies that each of the short lived flows (the majority) will have a file download time of about 2.5 rounds (RTT).

3. QCP Algorithm

The QCP algorithm at the end-hosts and at the routers can be described as follows:

- A source sends each byte j with its desired rate $\hat{R}_j$ carried in the corresponding packet of the byte.
- Each router in the network calculates $R(t)$ using equation 4 or the simplified QCP version equation 14 every control interval $d,\ 0 < d \leq RTT_{max}$. Here RTT_{max} is the maximum RTT of the flows which can be known or estimated offline.
- Each router in the path of a packet associated with byte j checks if $R(t) < \hat{R}_j$ in which case it overwrites $\hat{R}_j$ and forwards it unchanged otherwise.
- The destination then copies the $\hat{R}_j$ in the data packet to the ACK packet.
- The source sets its current window size $w'_j = \hat{R}_j RTT_j$ upon receipt of the ACK packet where RTT_j is the RTT of the flow of byte j.
- Each router updates its $R(t)$ value every control interval d.

4. QCP Rate

To define and derive the QCP rate metric, we first give notation in table 4.

The per flow fair QCP rate allocation at a bottleneck router is derived as follows.

The intuition behind QCP is the assertion that the total number L of bytes sent to a router (link) during a control interval d shouldn't exceed the bandwidth-delay product minus the queue size at the router during the interval. Denote w_j to be the windows size of (number of bytes sent by) a flow of byte j. Byte j is carried by its packet which arrives at the router. Define the *per byte cwnd* to be the number of bytes to be sent in the next round trip time (RTT) for each

Parameters	Description
C	Link capacity in bytes per sec
d	Duration of control interval in sec
$q(t)$	Queue size from the current interval in bytes
$R(t)$	Per flow rate allocation of the current interval in bytes per sec
N	Number of flows in the current interval
L	Total number of bytes which arrive to the router during a control interval of length d
m_i	Number of packets of flow i which arrive to the router
ϵ_i	The packet size of flow i in bytes
$\hat{L}$	Total number of packets which arrive to the router
R_i	Sending rate of flow i during the current round in bytes per sec
α, β	Stability parameters

Table 1. QCP Notations

of the w_j bytes sent by a source of byte j during the current RTT. If a source is sending at the rate R_j bytes/sec and plans to send at the fair share rate $R(t)$ bytes/sec in the next RTT, then the *per byte cwnd* is $R(t)/R_j$ bytes. The objective is to find the fair rate $R(t) = w'_j/RTT'_j$ using the current rate share $R_j = w_j/RTT_j$ of a flow associated with its byte j which arrives at the router.

The total number of bytes sent to a router from all sources in the next interval is then the sum of the *per byte cwnd* of all flows. With the notations defined in table 4, if $R_j = w_j/RTT_j$ denotes the rate associated with the jth of the L bytes which arrive to the router,

$$\sum_{j=1}^{L} \frac{R(t)}{R_j} = \sum_{k=1}^{\hat{L}} \frac{\epsilon_k R(t)}{R_k} = \sum_{i=1}^{N} \frac{m_i \epsilon_i R(t)}{R_i} = \alpha C d - \beta q(t) \tag{3}$$

This implies that

$$R(t) = \frac{\alpha C d - \beta q(t)}{\sum_{j=1}^{L}(1/R_j)} = \frac{\alpha C - \beta \frac{q(t)}{d}}{\frac{1}{d}\sum_{j=1}^{L}\frac{1}{R_j}}. \tag{4}$$

5. QCP Can Achieve Processor Sharing (PS)

In this section we discuss how QCP achieves PS. The inter-byte time σ_j is defined as the time between two consecutive bytes for a flow associated with byte j. It is given by $\sigma_j = \frac{1}{R_j}$. Now suppose a router has seen L bytes within the control time interval d. If n_i of these bytes carrying σ_i (in their corresponding packet) from source i are received by the router during the control interval d, then taking the denominator of equation 4 we have

$$\frac{1}{d}\sum_{j}^{L}\frac{1}{R_j} = \frac{1}{d}\sum_{i=1}^{N} n_i \frac{1}{R_i} = \sum_{i=1}^{N}\frac{n_i}{d}\frac{RTT_i}{w_i} \tag{5}$$

where N is the number of active flows and w_i is the congestion window size (cwnd) of flow i which is the number of bytes source i sends during its round trip time (RTT_i). The variable n_i is the total number of flow i bytes which arrive at the router during the control interval d.

In the case where all bytes sent from a source i at the rate of $R_i = w_i/RTT_i$ arrive to the next hop router (switch) at the same rate (as all the bytes of a flow to a router can be spaced at an equal interval of σ_i on average) we have that

$$\frac{n_i}{d} = \frac{w_i}{RTT_i}. \tag{6}$$

This implies that $\frac{1}{d}\sum_{j}^{L}\frac{1}{R_j} = \sum_{i=1}^{N}(\frac{n_i}{d})/(\frac{w_i}{RTT_i}) = N$ which means that QCP can achieve PS.

In the next section we will discuss scenarios where $\frac{n_i}{d} \neq \frac{w_i}{RTT_i}$ and where QCP can perform better than the traditional processor sharing (PS) in a scheme we define as *efficient sharing (ES)*.

6. Efficient Sharing (ES)

Before we define *Efficient Sharing (ES)*, we will first describe the following notations. The resource to be shared has a capacity of $X\ units/sec$. Different sources use a sequence (chain) of different resources one after the other. Some sources are currently requesting a total of M units of the current resource of interest per interval τ. If there are N such sources and if source k requests n_k units then $M = \sum_{k}^{N} n_k$. A source associated with unit j has a bottleneck resource share rate denoted by $\Re_j\ units/sec$. So the source associated with unit j is sending requests to the current resource at the rate of $\Re_j$. The current rate allocation at the current resource of interest is denoted with $\Re(t-d)$. To define ES, set

$$\check{\Re}_j = max\left(\Re_j, \Re(t-d)\right) \tag{7}$$

where max is a maximum function.

DEFINITION 1. *The Efficient Share allocation $\Re(t)$ for each source at the current resource of interest for the next interval is defined as*

$$\Re(t) = \frac{X}{\frac{1}{\tau}\sum_{j}^{M}\frac{1}{\check{\Re}_j}}. \tag{8}$$

In the case of QCP, $X = \alpha C - \beta\frac{q(t)}{d}$. We next show how ES outperforms the traditional PS and GPS by allocating capacity unused by some flows bottlenecked elsewhere to other flows which need the capacity. We also discuss how QCP is an ES protocol.

6.1 ES vs PS and GPS

A processor sharing (PS) which is also called a uniform processor sharing allocates a resource capacity of $X\ units/sec$ into N users equally. Its generalization is called Generalized Processor Sharing (GPS) [31] and was first proposed in [8] as weighted fair queueing (WFQ). GPS shares the resource X among the N users according to their weights ϕ_j. A source (user) i gets the share $\Re_i(t)$ given by

$$\Re_i(t) = \frac{\phi_i}{\sum_{k}^{N}\phi_k}X. \tag{9}$$

The case where all the ϕ_j are the same is a PS scheme. The weight values of ϕ_j are picked by the GPS scheduler. However GPS does not give any specific approach to obtain the *weights* in such a way that a resource m which can not be used by a user which is bottlenecked at another resource n is allocated to another user which can use the resource m. On the other hand, ES uses the weights given in the denominator of equation 8 to implicitly assign unused resource to all flows which can use it.

Even though ES obtains the same rate to all flows as shown in 8, the flows which do not need the assigned rate implicitly release the resource to other flows which require it by not using the capacity of the resource beyond what they can use (bottlenecked elsewhere). For example if $X = 100\ units/sec$, $R_1 = 10\ units/sec, n_1 = 10\,units$, $R_2 = 50\ units/sec, n_2 = 50\,units$ and $R_3 = 70\ units/sec, n_3 = 70\,units$ for a control interval of $\tau = 1\ sec$, and $R(t-d) = 40\,units/sec$ then since $max(10, 40) = 40$ from equation 7,

$$\Re(t) = \frac{100}{\frac{10}{40} + \frac{50}{50} + \frac{70}{70}} = \frac{100}{2.25} = 44.44\,units/sec.$$

Here, all three users are assigned the same rate $\Re(t) = 44.44$. However user 2 implicitly releases the resources it can not use by sending only at $10\ units/sec$. A PS mechanism would result in a share of $100/3 = 33.3$ and would not assign the resource unused by user 1 to the other users (sources). The GPS on the other hand does not provide a mechanism to obtain proper weight values at a multi-bottleneck level to allow efficient use of resources. Hence ES can also be viewed as a special case GPS where weights are automatically and adaptively calculated at a distributed network level in such a way that what some sources (users) *can not use* is *equally allocated* to other users in a work conserving manner (utilizing available resource if there is a demand for it).

A resource some users cannot use can also be allocated to other users which can use it proportionally based on some weights ϕ_i as the case of GPS. We call this generalization of ES a generalized efficient sharing (GES). The same argument as equation 3 can be used to find a new rate $\Re_j(t)$ with weight ϕ_j associated with unit j as follows.

$$\sum_k^M \frac{\Re_k(t)}{\Re_k} = \sum_k^M \frac{\phi_k \Re(t)}{\Re_k} = X\tau. \qquad (10)$$

This implies that

$$\Re(t) = \frac{X}{\frac{1}{\tau}\sum_k^M \frac{\phi_k}{\Re_k}}. \qquad (11)$$

Then the GES rate $\Re_j(t) = \phi_j \Re(t)$.

In addition to achieving PS, QCP can also handle ES scenarios which a traditional PS scheme cannot handle. This enables QCP to achieve more efficient sharing (ES) than PS. We will next use two QCP cases to describe this ES.

6.2 Single Bottleneck Scenario

There can be a scenario where $\frac{n_i}{d} < \frac{w_i}{RTT_i}$. This happens for instance when a new bottleneck link is formed in the flow path before the location of the previous bottleneck link which allocated $R_i = \frac{w_i}{RTT_i}$ to flow i. The new bottleneck link then drops or delays packets of flow i resulting in smaller rate $\frac{n_i}{d}$ arriving to the previous bottleneck link. In this case, QCP in the previous bottleneck link counts flow i as less than one flow (*fractional flow*) which is equal to $\frac{n_i}{d}/R_i$. On the other hand, the traditional PS counts each of such flows as one flow. In this case, the PS approach at the previous bottleneck link divides the capacity by more than the *actual* number of flows. This results in PS allocating less rates to some flows which need more. Dividing the capacity by the *exact fractional number* of flows, QCP however gives the capacity unused by some flows to flows which can use it without causing buffer overflow or resource underutilization. To do this, QCP doesn't require any special queues or complicated operations as the allocation is done using QCP rate equation. On the other hand, the scenario where $\frac{n_i}{d} > \frac{w_i}{RTT_i}$ may occur, for instance when bursts of packets of a flow arrive to a link. In this case, QCP temporarily counts such flows as $\frac{n_i}{d}/R_i$ which is more than one flow. Hence, QCP assigns less rate to flows to absorb the bursts of packets.

Another important result from equations 6 and 5, is that unlike RCP [9] and XCP [18] the estimation of the control interval, d, in QCP *doesn't need the exact flow RTTs*. The value of d can be set to some reasonable value between maximum and minimum RTT values. It can be user-defined and obtained from reasonable offline experiments. The smaller the value of d, the more recent bottleneck rate values the packets carry back to their sources. QCP is less sensitive of the choice of d. This is because if the choice of d results in $\frac{n_i}{d} \neq \frac{w_i}{RTT_i}$, the ES nature of QCP temporarily counts the flow as a fractional flow resulting in an accurate rate calculation as discussed above. QCP can also use flow RTTs to obtain d like XCP and RCP.

6.3 Multi-Bottleneck Network

In a multi-bottleneck network where different flows are bottlenecked at different links, some flows may not be able to utilize their equal share allocation at a link which is a bottleneck to other flows. If the bottleneck link allocation of flow i is R_i and if its current equal rate share at its non-bottleneck link is $R(t-d) > R_i$, then flow i can waste its non-bottleneck link capacity which can otherwise be used by other flows bottlenecked at that link. This can result in QCP not achieving ES.

To deal with this scenario, QCP uses

$$\check{R}_j = max(R_j, R(t-d)) \qquad (12)$$

instead of the R_j in the denominator of equation 4, where R_j is the source rate carried by a packet associated with byte j

of a flow and $R(t-d)$ is the rate allocation of flows at the link for the current interval.

QCP uses expression 12 only if the flow associated with byte j is in its second RTT sending packets at its bottleneck link rate. QCP can check this by comparing R_j against the initial QCP rate R_{init} of flows which can be known before hand as the ratio of initial *cwnd* and some average flow RTT. In this case if $R_j \leq R_{init}$, QCP doesn't use the expression 12 as the flow may be just starting. QCP packet header can also carry a single bit to indicate the start of a flow. If possible, SYN packet can also be used to indicate the start of the flow. OpenFlow switches (routers) can also detect the first packet of a flow if the packet does not belong to any of the flow table entries [32].

Here is some explanation of why the approach in expression 12 can achieve ES (Efficient Sharing). If $R_j < R(t-d)$, then a flow which owns byte j should be treated as a partial (fractional) flow by the router which allocated $R(t-d)$ to the flows (including the flow of byte j). This enables QCP to assign the unused resource to other flows bottlenecked at that router.

On the other hand, if $R_j > R(t-d)$, QCP achieves ES by treating the flow of byte j as at least one flow as it can cause temporary queue spikes (being late to learn its new allocation). This occurs for instance because the allocation R_j was much older than $R(t-d)$ as the flow has an RTT too long (longer than the control interval) to know about its latest rate allocation.

If we approximate R_j used in equation 4 with $R(t-d)$ even if $R_j > R(t-d)$, we get

$$\begin{aligned} N_a &= \frac{1}{d}\sum_{j}^{L}\frac{1}{R_j} \approx \frac{1}{d}\sum_{j}^{L}\frac{1}{R(t-d)} \\ &\approx \frac{1}{R(t-d)}\frac{L}{d} = \frac{y(t)}{R(t-d)} \end{aligned} \tag{13}$$

where $y(t) = \frac{L}{d}$ is the total input traffic rate in bytes during the control interval d at the router, and $R(t-d)$ and R_j are rates per flow.

When QCP uses equation 13, it can overestimate the actual number N_a of flows when $\frac{1}{R_j} < \frac{1}{R(t-d)}$. This overestimation of N_a can result in a lower rate allocation to all flows which in turn can result in link underutilization. This is specially true with a misbehaving flow. If QCP uses equation 13, the misbehaving flow can continue to increase its rate at the expense of the other flows as the router continues to count the flow as more than one flow. Such behavior is not fair to the other flows which quickly obey the QCP rate rule.

In a simplified version of QCP, we also use equation 13 in the denominator of equation 4 as an estimation of the actual number of flows. The resulting simplified QCP rate is then given by

$$R(t) = \frac{\alpha C d - \beta q(t)}{d N_a} \tag{14}$$

The derivation in equation 13 shows that the main strength of this simplified version of QCP lies on its use of the *fractional flow* concept where flows can be counted as partial flows unlike the case of PS. Hence, the simple expression given by equation 13 is an estimator of ES. This implementation allows QCP packet header to be even smaller (about 8 bytes) as shown in section 7. In the simulation experiments of this paper we used the exact QCP rate given by equation 4.

7. QCP Packet Header Format

The QCP header can be placed as shim layer between the TCP and IP headers. QCP can have two packet header implementation schemes. The first one which is shown in figure 1 has a 12 byte header.

0 1 2 3 ... 14 15 16 ... 30 31 32
Inter–Byte Interval (Inverse of Flow Bottleneck Rate)
QCP Bottleneck Rate
QCP Reverse Bottleneck Rate

Figure 1. QCP header with 12 bytes

The first field is the *Inter-Byte Interval* length $\sigma_j = 1/R_j$, where R_j is the current sending rate attached to a packet associated with byte j of the corresponding flow. The routers in the path of byte j (its associated packet) use this field to obtain the QCP rate given by equation 4. The second field is the *QCP Bottleneck Rate* $\hat{R}_j$ which is the rate initialized to be the desired rate by source. The bottleneck router in the path of the packet associated with byte j can then overwrite the value. This rate is the minimum of all the rates in the path of the packet associated with byte j. The third field is *QCP Reverse Bottleneck Rate* which is the same QCP bottleneck rate which the receiver copies to its outgoing packets (ACK packets for example). The simulation results for QCP used in this paper use this implementation scheme of the QCP header.

The second implementation scheme of the QCP header shown in figure 2 is without the σ_j field. This implementation can reduce the QCP packet header to 8 bytes.

0 1 2 3 ... 14 15 16 ... 30 31 32
QCP Bottleneck Rate
QCP Reverse Bottleneck Rate

Figure 2. QCP header with 8 bytes

In this implementation scheme each source sets the value of the *QCP bottleneck rate* ($\hat{R}_j$) to its desired rate. Each router in the path of the packet associated with byte j calculates the rate using equation 14. If this rate is smaller than the $\hat{R}_j$ in the packet header, then the router replaces the $\hat{R}_j$ in the packet header with what it obtains using equation 14. The receiver then copies the value of the *QCP bottleneck*

rate which routers may have changed, into the ACK (returning) packets. The source which receives the ACK packets then adjusts its *cwnd* to the product of the rate it gets from the ACK packets and its RTT.

8. Weighted QCP

The QCP rate given by equation 4 can be extended to be a weighted share metric. Such a metric allows different flows to get different shares based on their weights without causing router buffer overflow or link under-utilization. If packet j of a flow carries the weight information ω_j of its flow, then using a derivation similar to the one used in equations 10 and 11, we have

$$\sum_{j=1}^{L} \frac{\omega_j R(t)}{R_j} = \alpha C d - \beta q(t). \tag{15}$$

This implies that

$$R(t) = \frac{\alpha C d - \beta q(t)}{\sum_{j=1}^{L} (\omega_j / R_j)}. \tag{16}$$

A flow of packet j can then set its rate $R_j = \omega_j R(t) = \omega_j \hat{R}_j$ where $R(t)$ is the bottleneck link rate.

Different policies can be set for different classes of flows. For instance, if a flow which just received the rate of $\hat{R}_k$ from its ACK packet k wants to achieve a target rate of $R_j^{\hat{T}}$ for the next round, it sets its weight ω_j as $\omega_j = \frac{R_j^{\hat{T}}}{\hat{R}_k}$.

Different levels of priority can be used by adding a few more bits in the QCP header or using the current IP header fields (ECN bits). The source can also send ω_j / R_j in the QCP header. Each source i can then set its congestion window as $w_i = \omega_i \hat{R}_i RTT_i$ packets where $\hat{R}_i$ is obtained from the ACK packets.

9. Stability Analysis

In this section we present stability analysis of QCP using control theory.

9.1 Lyapunov Stability

The rate allocation by QCP queue at a bottleneck router is done every control interval d. This allocation is received by each source sharing the bottleneck link after a round trip time (of each of the sources). This new rate allocation changes the congestion window w_j of each source j. So the aggregate feedback sent per unit time is the sum of the derivatives of the congestion windows. This feedback is similar with the XCP feedback and hence we have

$$\sum_j \frac{dw_j}{dt} = C - \Lambda(t-d) - \frac{q(t-d)}{d} \tag{17}$$

where $\Lambda(t-d)$ and $q(t-d)$ are the total arrival rate and queue size in the previous control interval and C is the link capacity.

Adding the control parameters α and β for stability, Equation 17 becomes

$$\sum_j \frac{dw_j}{dt} = \alpha(C - \Lambda(t-d)) - \beta \frac{q(t-d)}{d}. \tag{18}$$

As shown in [2] and [24], the QCP feedback mechanism is given by the delay differential equations

$$\begin{aligned} \Lambda'(t) &= \frac{\alpha}{d}(C - \Lambda(t-d)) - \frac{\beta}{d^2} q(t-d) \\ q'(t) &= \begin{cases} \Lambda(t) - C, & q(t) > 0 \\ max\{\Lambda(t) - C, 0\}, & q(t) = 0. \end{cases} \end{aligned} \tag{19}$$

As the QCP feedback mechanism can be written in Equation 19, appropriate Lyapunov functions can be used to find stable values of the control parameters α and β. For instance the work [24] shows that $\beta/d^2 = \alpha/d$ gives stability. This for instance implies that if $\alpha = 1.0$, $\beta = d$. Previous work [2] also shows a wide range of stable values for protocols whose feedback mechanism can be written in the form of Equation 19. Our detailed simulation results also show that $\alpha = 1 = \beta$ gives stable values for QCP. We are also working on using the Lyapunov functions in [24] to find even wider stable regions for α and β.

10. Gradual Deployment of QCP

In the implementation of QCP, routers (router-like boxes) read the QCP packet headers, calculate rate and modify the packet headers of flows. QCP can be easily implemented by extending the OpenFlow [26, 32] which enables clean slate schemes to be implemented in big networks such as the Google backbone network [15, 16]. In this section we will discuss how QCP can be implemented using (by extending) the OpenFlow architecture.

There are different ways QCP can be implemented using the OpenFlow switch and protocol specification [32]. For the rest of this section we will use the terms switch and router interchangeably. As specified in [32], each flow entry of an OpenFlow switch contains a set of *instructions* that are executed when an arriving packet matches the entry. One of the instruction types is *Meter* which directs the packet to a specified meter. Each meter has one or more *meter bands*. Each band specifies the *rate* at which the band applies and the way packets should be processed. Packets are processed by a single meter band based on the *current measured meter rate*. The meter applies the meter band with the highest *configured rate* that is lower than the current measured rate. If the *current measured rate* is lower than any *specified meter band rate*, no meter band is applied.

In the case of QCP, the configured rate can be obtained by taking the *QCP bottleneck rate* ($\hat{R}_j$) from the QCP packet header. The *measured meter rate* associated with a specific link can be replaced with the QCP rate in equation 21 obtained as discussed in section 10.1. The OpenFlow switch

can then invoke the *Apply-Actions* instruction to apply the *Set-Field* action which overwrites the QCP bottleneck rate in the packet header.

10.1 QCP Rate Using OpenFlow

The flow table of an OpenFlow switch can maintain a per flow packet counter. By polling the packet counter every control interval d the number n_i of packets of each flow i during the interval d (for each of the N flows sharing a given link) can be obtained. The flow table also maintains the numbers L_r of received and L_s served bytes for each flow from which the queue size $q(t)$ of a link can be obtained. The $q(t)$ can also be obtained by reading the OpenSwitch queue length. By reading n_i at every control interval d, the current sending rate r_i of flow i is given by $r_i = n_i/d$. If the QCP rate of the link obtained from the previous control interval is $R(t-d)$, then setting

$$\check{r}_i = max\,(r_i, R(t-d)), \tag{20}$$

the QCP rate can then be calculated as

$$R(t) = \frac{\alpha C - \beta \frac{q(t)}{d}}{\sum_{i=1}^{N} \frac{r_i}{\check{r}_i}}. \tag{21}$$

This counters can be obtained and the rate can be calculated using the OpenFlow switches (decentralized) with off-ASIC or on-ASIC CPU [23, 28, 33]. It can also be calculated by the OpenFlow controllers (centralized) and sent to the OpenFlow switches. The OpenFlow switches then apply the *Set-Field* action using the instruction discussed above. The *Set-Field* action can be applied to the first packet of each flow allowing each source to jump start its sending rate as what Quick start TCP [37] aims to do. The *Set-Field* action can also be applied to some randomly selected or to all packets of a flow. This implementation scheme can be done using the 8 byte QCP header scheme as discussed in section 7.

10.2 Using Stateless OpenFlow

Another QCP implementation scheme using the OpenFlow concept can use the 12 bytes or 8 bytes QCP header scheme discussed in section 7. In this implementation scheme, the OpenFlow switches do not even need to read the bytes counts from the flow tables. In this case, the source rate R_j in the QCP packet header field which carries $\sigma_j = \frac{1}{R_j}$ with equations 4 and 12 is used instead of the r_i in equations 20 and 21. This implementation scheme does not require OpenFlow switches and controllers to maintain per flow states (such as counters), which is the main OpenFlow scalability issue [29]. The per link rate which can be calculated using equation 4 (with end-host assistance) or equation 14 (with no end-host assistance) is all the switches and controllers need to achieve QCP and other OpenFlow objectives. For instance OpenFlow rate limiting and max-min routing can be done as briefly discussed in sections 10.4 and 10.3 using this rate.

10.3 Using OpenFlow Edge Switches

The QCP rate given by equation 21 or 4 can also be used as a link weight metric in a max-min routing algorithm. The max-min routing algorithm finds the minimum rate of each path and takes the path with the highest such rate. The OpenFlow controller can run the max-min algorithm and provide the edge OpenFlow switches with such max-min *path* and the corresponding *rate*. Parallelism and other approaches are being used to scale the OpenFlow controller[5, 21, 38]. The max-min can also be done by the routers (OpenFlow switches) in a distributed manner by exchanging the rate as a link metric. The edge switches can then use the *Set-Field* action to replace the rate at the QCP packet headers. The QCP packet headers then do not have to be changed until they reach another edge switch (router) which has rate of its link. Using this approach saves the core switches (routers) more packet processing time. This approach can be easily performed using emerging network virtualization architectures such as [30] with intelligent edge open vSwitches and controller cluster.

10.4 QCP Rate Limiting

Rate limiting of a flow can also be done using the QCP rate. If the *measured rate* R_M of a flow at a switch is higher than the QCP rate $R(t)$ obtained using equation 21, packets of the flow can be dropped (sent to low priority queue) with probability $(R_M - R(t))/R_M$. Otherwise, the packets are served with no drops. Different flows sharing a link can also get different rate allocations to support Quality of Service (QoS). By associating a weight ω_i to flow i the QCP rate can be obtained as

$$R(t) = \frac{\alpha C - \beta \frac{q(t)}{d}}{\sum_{i=1}^{N} \frac{\omega_i r_i}{\check{r}_i}}. \tag{22}$$

Flow k can then get a share of $R_k = \omega_k R(t)$. In this case, if $R_M > R_k$, packets of flow k are dropped (sent to low priority queue) with probability $(R_M - R_k)/R_M$. Otherwise packets of flow k are served with no drops.

11. QCP with TCP Flows

To allow QCP to be implemented with other protocols such as TCP for incremental deployment, the QCP router can be modified as follows. The QCP router creates separate fair queues for TCP and QCP traffic. The router serves packets from the TCP and QCP queues based on weights, for instance, using round robin. The weights ω_T and ω_Q of the TCP and QCP queues can be calculated the same way as the weights of TCP and XCP queues are calculated in [18]. To force its flows to be fair to TCP, QCP also uses the remaining capacity $\omega_Q C$ instead of the entire link capacity C in the calculation of the QCP rate using equations 4 and 22.

Table 2. Baseline parameters for experiments on estimation of N

Parameter	Default value
Link capacity	20 Mbps
Link propagation delay	50 ms
Number of flows	10
File size	4 MB

12. Simulation Analysis

In previous studies [9], RCP was shown to outperform TCP and XCP. In this section, we evaluate the performance of QCP comparing it with RCP and XCP using NS2 [25] which is a widely used network simulator.

To validate the performance of QCP, we have implemented the QCP source as a sub-class of TCP-Reno and QCP queue as a subclass of DropTail Queue in NS2.

Similar to previous work on RCP, we first use a simple topology which contains sources and a destinations connected by one single bottleneck link. Unless specified we use a router buffer size of 1 bandwidth-delay-product (BDP).

In the first set of experiments, we show how RCP and QCP make estimation on the number of flows. We generate a fixed number of big size flows which all start at the same time. The baseline parameters are summarized in Table 2.

Figure 3 plots the estimation of number of flows versus time for QCP and RCP. We use the same value of $\alpha = 1 = \beta$ for QCP in all experiments while RCP uses different values of α and β in different experiments. The estimation of N from QCP virtually matches the real value. In contrast, depending on the choice of parameters, the estimation of N from RCP either needs much longer time to converge or even never converges. This in turn results in flows taking longer to finish than necessary. Two important messages conveyed here are: (1) QCP gives a more accurate and reliable estimation of the number of flows than RCP; (2) the performance of RCP is sensitive to the setting of parameters and there is no specific rule on how to set these parameters. In the rest of the paper we use $\alpha = 0.1$ and $\beta = 1$ for RCP experiments.

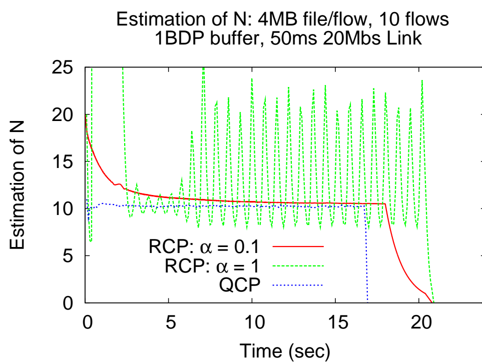

Figure 3. Estimation of N versus time with $\alpha = 0.1$, $\alpha = 1$ for RCP

To compare the average flow completion time (AFCT) of QCP against RCP we have also considered a different numbers of flows with a fixed file size. As can be seen from Figure 4 the AFCT of QCP is smaller than that of RCP.

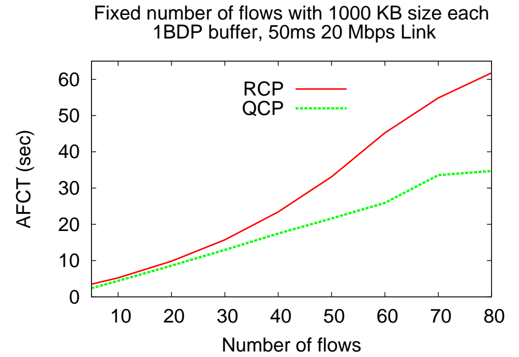

Figure 4. AFCT versus number of flows

RCP performs badly when the number of flows grows as shown in figure 4. This is because RCP either under or over estimates the number of flows into which the bandwidth is divided.

Most of the experiments used to validate RCP in [9] were obtained using a non-congestion scenario with an average link load of around 90%. However such a simulation scenario doesn't properly evaluate the performance of RCP. In fact as in a *Naive QCP* approach, where we set the initial *cwnd* of every flow equal to the file size of the flow for the cases, where the link on average is not fully utilized (similar to many RCP experiments in the [9]), the network doesn't get congested on average as shown in figure 5. In this scenario a congestion control protocol is not even strictly needed as all flows can send all the packets they have in one round and retransmit some of their lost or delayed packets to get very small AFCT. As can be seen from the plot, even Naive QCP outperforms RCP.

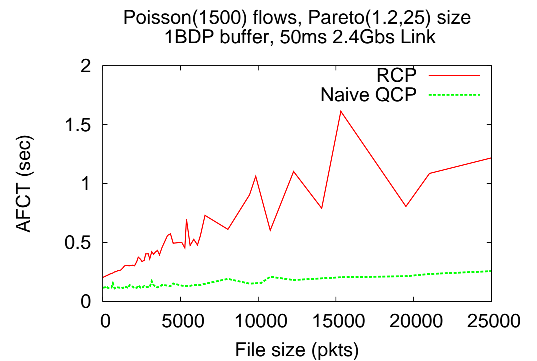

Figure 5. AFCT of *Naive QCP* vs RCP: Poisson(1500 flows/sec), Pareto(1.2, 25pkts)

However, under a congestion scenario, the performance of RCP is worse when compared with QCP as shown in the

Protocol	Number of finished flows (in 26.061 sec)
RCP	17280
QCP	66212

Table 3. QCP versus RCP under high load scenario: Poisson(8333.3 flows/sec), Pareto(1.2,30 pkts)

next experimental results. In these experiments Poisson flow arrivals where the file sizes are Pareto distributed are used as is also the case in [9, 18, 36] to emulate Internet and datacenter traffic [3]. As shown in table 3, within a simulation time of 26.061 seconds only 17280 RCP flows finished due to the increasingly high file completion time (FCT) as shown in figure 6. On the other hand as can be seen from table 3, 66212 QCP flows finished during the same time.

The y-axis of figure 6 shows how the average completion time of flows that complete within each progressing 2 second interval grows with simulation time in a loaded link scenario. As the simulation time progresses, RCP results in higher file completion time of the flows that finish. This in turn results in less flows finishing in RCP than in QCP as shown in table 3.

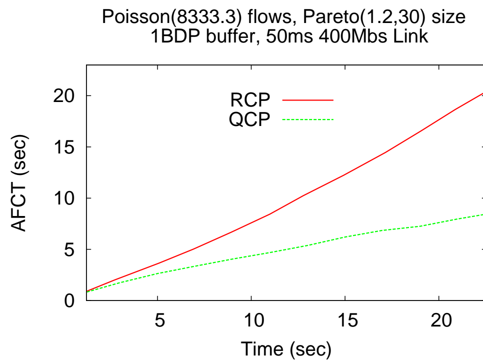

Figure 6. FCT of flows versus simulation time (with 2 sec aggregation): Poisson(8333.3 flows/sec), Pareto(1.2, 30 pkts)

We also compared the FCT of the 17280 RCP flows (all RCP flows) which finished against the first 17280 QCP flows which finished. As shown in figure 7 the FCT of QCP flows is much smaller than that of RCP. This small FCT helped more QCP flows finish in a shorter time as shown in table 3.

We next give comparison of QCP against XCP which is another major clean slate congestion control protocol. As discussed in section 2.2.2, XCP is not fair to small size flows (called mice) which are the majority of Internet flows. This is because the link bandwidth is dominated by a few large size flows (called elephants).

Table 4 shows that the AFCT of 20 flows about one-third of which are elephant flows and the remaining 13 flows are mice flows. The file size of each of the elephants is 1MB

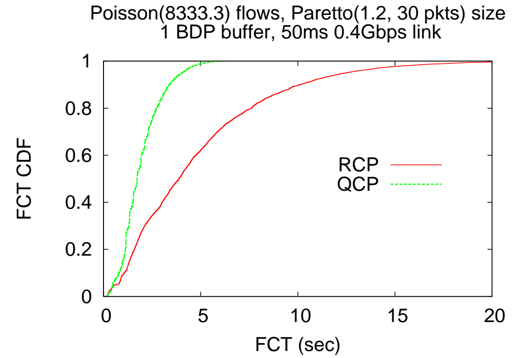

Figure 7. FCT CDF of finished RCP flows vs QCP flows: Poisson(8333.3 flows/sec), Pareto (1.2, 30pkts)

QCP_M	XCP_M	QCP_E	XCP_E	QCP_{Avg}	XCP_{Avg}
0.4001	0.8971	3.8176	3.5130	1.5962	1.8127

Table 4. AFCT (seconds): QCP versus XCP

and that of each mice is 50KB. The single bottleneck link bandwidth is 20 Mbps with a propagation delay of 50ms. All the elephant flows start at the same time and the mice flows start about 1 second after the elephant flows start.

As shown in table 4, the AFCT of XCP mice flows (XCP_M) is more than twice the AFCT of QCP mice flows (QCP_M). This shows how unfair XCP is to short flows when compared to QCP. While achieving this fair allocation to small file size (mice) flows, QCP does not compromise the overall average flow completion time (QCP_{Avg}) when compared with XCP. QCP is also fair to the big file size (elephant) flows (QCP_E).

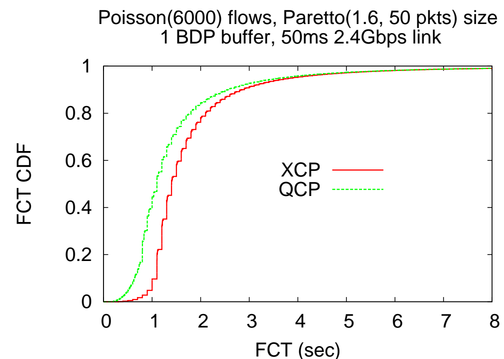

Figure 8. FCT of XCP flows vs QCP flows: Poisson(6000 flows/sec)

Figures 8 and 9 present the CDF of QCP versus XCP for a link capacity of 2.4Gbps with a propagation delay of 50ms. In figure 8 flows arrive following a Poisson distribution with mean 6000 flows/sec and file sizes are Pareto distributed with mean 50 packets (packet size = 1000 Bytes) and shape parameter of 1.6. In figure 9 flow arrival is Poisson with

mean alternating between 5400 flows/sec for 3 seconds and 30000 flows/sec for 2 seconds. This simulates a link where the average load fluctuates between high load and low load. In figure 9 file sizes are also Pareto distributed but with mean 30 packets and shape parameter of 1.2.

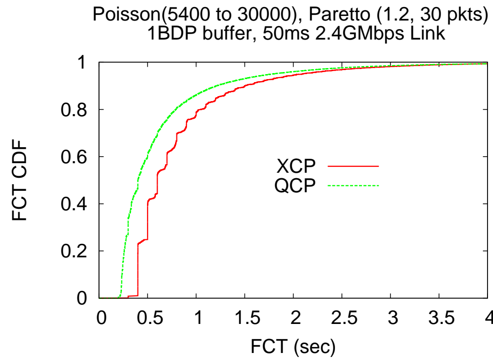

Figure 9. FCT of XCP flows vs QCP flows: Poisson(5400 to 30000 flows/sec)

As can be seen from both figures 8 and 9 the file completion time of of the majority of QCP flows is smaller than that of XCP flows. As file sizes are Pareto distributed to emulate the Internet flows, most of the flows in this simulation setup are small size flows (mice).

We have also compared the performance of QCP against RCP for a two bottleneck network topology as shown in figure 10.

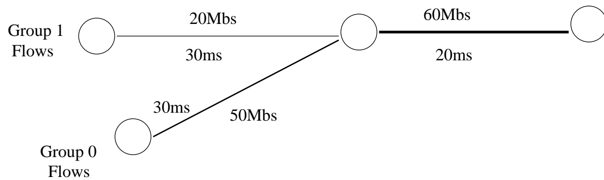

Figure 10. Two bottleneck network topology

As shown in figure 11, QCP gives lower FCT for the two groups of flows crossing two different bottleneck links as shown in the topology of figure 10.

We have also simulated QCP against RCP and XCP using flow inter-arrival times and flow size traces taken from [3] and [12]. In figure 12 we use a bottleneck link capacity of 0.4Gbps and flow inter-arrival times uniformly distributed between 10 and 100 micro seconds taken from [3] to evaluate QCP against other protocols under a high load (congestion) scenario. Like the previous experiments, QCP flows finish quicker resulting in more QCP flows finishing. Within 8.6 simulation time, 48511 QCP flows and 37762 RCP flows completed. Figure 12 shows the FCT CDF of all RCP flows that finished against the corresponding QCP flows that finished.

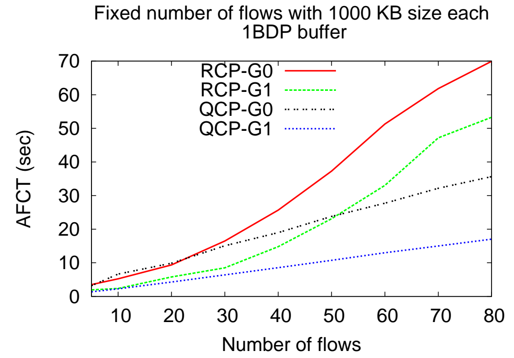

Figure 11. FCT of Group 0 (G0) and Group 1 (G1) RCP and QCP flows

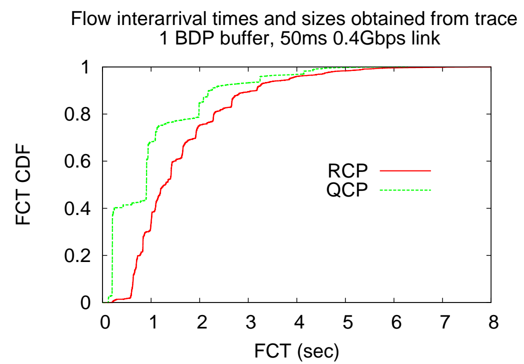

Figure 12. FCT CDF of RCP flows vs QCP flows: Flow inter-arrivals and sizes taken from traces

13. Summary

In this paper we presented the design of a Quick congestion Control Protocol (QCP). QCP uses a *fair rate metric* to determine *the rate at which flows send data*. We have shown how QCP can achieve a more efficient sharing (ES) scheme than the traditional processor sharing (PS). We have described how QCP can be implemented in the new OpenFlow networking architecture which is currently being deployed in major enterprise networks.

Simulation results show that QCP can outperform RCP and XCP which are the two major clean slate congestion control protocols.

References

[1] M. Al-Fares, K. Elmeleegy, B. Reed, and I. Gashinsky. Overclocking the Yahoo!: CDN for faster web page loads. In *IMC '11*, pages 569–584, New York, NY, USA, 2011. ACM.

[2] H. Balakrishnan, N. Dukkipati, N. McKeown, and C. Tomlin. Stability analysis of explicit congestion control protocols. *Communications Letters, IEEE*, 11(10):823–825, October 2007.

[3] T. Benson, A. Akella, and D. A. Maltz. Network traffic characteristics of data centers in the wild. In *IMC '10*, pages

267–280, New York, NY, USA, 2010. ACM.

[4] L. S. Brakmo, S. W. O'Malley, and L. L. Peterson. TCP vegas: new techniques for congestion detection and avoidance. *SIGCOMM Comput. Commun. Rev.*, 24(4):24–35, Oct. 1994.

[5] Z. Cai, A. L. Cox, and T. S. N. Eugene. Maestro: A System for Scalable OpenFlow Control. Technical Report: Rice University, 2010.

[6] C. Casetti, M. Gerla, S. Mascolo, M. Y. Sanadidi, and R. Wang. TCP Westwood: end-to-end congestion control for wired/wireless networks. *Wirel. Netw.*, 8(5):467–479, Sept. 2002.

[7] S. Dawkins, G. Montenegro, M. Kojo, V. Magret, and N. Vaidya. End-to-end performance implications of links with errors. United States, 2001. RFC3155 Editor.

[8] A. Demers, S. Keshav, and S. Shenker. Analysis and simulation of a fair queueing algorithm. *SIGCOMM Comput. Commun. Rev.*, 19(4):1–12, Aug. 1989.

[9] N. Dukkipati and N. McKeown. Why flow-completion time is the right metric for congestion control. *SIGCOMM Comput. Commun. Rev.*, 36(1):59–62, 2006.

[10] N. Dukkipati, N. McKeown, and A. G. Fraser. RCP-AC: Congestion Control to Make Flows Complete Quickly in Any Environment. In *INFOCOM*, 2006.

[11] S. Floyd. HighSpeed TCP for Large Congestion Windows. United States, 2003. RFC Editor.

[12] A. Greenberg, J. R. Hamilton, and *et.al*. VL2: a scalable and flexible data center network. *Commun. ACM*, 54(3):95–104, Mar. 2011.

[13] S. Ha, I. Rhee, and L. Xu. CUBIC: a new TCP-friendly high-speed TCP variant. *SIGOPS Oper. Syst. Rev.*, 42(5):64–74, July 2008.

[14] T. R. Henderson, E. Sahouria, S. McCanne, R. H. Katz, and Y. H. Katz. On Improving The Fairness Of TCP Congestion Avoidance. In *IEEE Globecomm*, pages 539–544, 1997.

[15] S. Higginbotham. Google's next OpenFlow challenge: taking SDNs to the consumer. http://gigaom.com/cloud/how-google-is-using-openflow-to-lower-its-network-costs/, apr 2012.

[16] S. Higginbotham. How Google is using OpenFlow to lower its network costs. http://gigaom.com/cloud/how-google-is-using-openflow-to-lower-its-network-costs/, apr 2012.

[17] V. Jacobson. Congestion avoidance and control. In *SIGCOMM '88*, pages 314–329, New York, NY, USA, 1988. ACM.

[18] D. Katabi, M. Handley, and C. Rohrs. Congestion control for high bandwidth-delay product networks. *SIGCOMM Comput. Commun. Rev.*, 32(4):89–102, 2002.

[19] L. Kleinrock and R. R. Muntz. Processor sharing queueing models of mixed scheduling disciplines for time shared system. *J. ACM*, 19:464–482, July 1972.

[20] E. Kohler, M. Handley, and S. Floyd. Designing DCCP: congestion control without reliability. *SIGCOMM Comput. Commun. Rev.*, 36(4):27–38, 2006.

[21] T. Koponen, M. Casado, and *et.al*. Onix: a distributed control platform for large-scale production networks. OSDI'10, pages 1–6, Berkeley, CA, USA, 2010. USENIX Association.

[22] T. V. Lakshman and U. Madhow. The performance of TCP/IP for networks with high bandwidth-delay products and random loss. *IEEE/ACM Trans. Netw.*, 5(3):336–350, 1997.

[23] G. Lu, R. Miao, Y. Xiong, and C. Guo. Using cpu as a traffic co-processing unit in commodity switches. HotSDN '12, pages 31–36, New York, NY, USA, 2012. ACM.

[24] Z. Lu and S. Zhang. Stability analysis of XCP congestion control systems. In *WCNC 2009. IEEE*, pages 1–6, april 2009.

[25] S. McCanne and S. Floyd. NS-2. `http://www.isi.edu/nsnam/ns/`.

[26] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner. Openflow: enabling innovation in campus networks. *SIGCOMM Comput. Commun. Rev.*, 38:69–74, March 2008.

[27] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner. OpenFlow: enabling innovation in campus networks (OpenFlow White Paper), March 2008.

[28] J. C. Mogul and P. Congdon. Hey, you darned counters!: get off my asic! HotSDN '12, pages 25–30, New York, NY, USA, 2012. ACM.

[29] J. C. Mogul, J. Tourrilhes, and *et.al*. DevoFlow: cost-effective flow management for high performance enterprise networks. Hotnets '10, pages 1:1–1:6, New York, NY, USA, 2010. ACM.

[30] Nicira. Its Time to Virtualize the Network. http://nicira.com/en/network-virtualization-platform, 2012.

[31] A. K. Parekh and R. G. Gallager. A generalized processor sharing approach to flow control in integrated services networks: the single-node case. *IEEE/ACM Trans. Netw.*, 1:344–357, June 1993.

[32] B. Pfaff and et. al. Openflow switch specification v1.3.0. https://www.opennetworking.org/, April 2012.

[33] PluribusNetworks. F64 Series Server-Switch with Netvisor OS. http://www.pluribusnetworks.com/solutions/, 2012.

[34] F. Qian, A. Gerber, and *et.al*. TCP revisited: a fresh look at TCP in the wild. In I*MC '09*, pages 76–89, New York, NY, USA, 2009. ACM.

[35] A. Rao, A. Legout, and *et.al*. Network characteristics of video streaming traffic. In C*oNEXT '11*, pages 25:1–25:12, New York, NY, USA, 2011. ACM.

[36] J. Rojas-Mora, T. Jiménez, and E. Altman. Simulating flow level bandwidth sharing with pareto distributed file sizes. In VALUETOOLS '11, pages 265–273, ICST, Brussels, Belgium, Belgium, 2011. ICST.

[37] M. Scharf and H. Strotbek. Performance evaluation of quick-start TCP with a Linux kernel implementation. In I*FIP NETWORKING 2008*, pages 703–714, Berlin, Heidelberg, 2008. Springer-Verlag.

[38] A. Tavakoli, M. Casado, T. Koponen, and S. Shenker. Applying NOX to the Datacenter. HotNets, 2009.