EXPLORING DELAY-BASED TCP CONGESTION CONTROL

A THESIS SUBMITTED TO

THE NATIONAL UNIVERSITY OF IRELAND

FOR THE DEGREE OF

MASTER OF SCIENCE

by

Gavin D. McCullagh

Based on research carried out in

The Hamilton Institute,

N.U.I. Maynooth

under the direction of **Professor Doug Leith**

N.U.I. Maynooth *February 2008*

Contents

Chapter 1

Background

The internet is undoubtedly one of the most pervasive, revolutionary technologies to gain widespread use in the last two decades. It's effects and influence can be seen throughout the world, particularly in the first world but increasingly also in developing countries, many of whom see it as having an economic levelling effect.

The network itself really only provides a (usually) moderately reliable signalling and routing system for sending small chunks (packets) of data from one host to another. Packets may be lost or reordered without warning. Hosts who wish to communicate across the network must work within these constraints. Communicating hosts must provide for themselves any greater level of reliability they need. This design is based on the "end-to-end" principal [Saltzer et al., 1984] which places as much of the communications protocol operations as possible at the end points.

There are currently two very common "transport protocols" running on the end hosts. UDP simply sends packets and provides no extra reliability. If the user doesn't get a response to a request, they are usually expected to request again if they so choose. TCP by contrast ensures that arrival of

all packets at their destination is confirmed, retransmitting any which are lost; ensures that any reordering of the packets is corrected, keeps different communication sessions from interfering and attempts to send at the highest rate it can without causing excessive packet loss due to congestion in the network. This last responsibility of TCP is called "congestion control" and was added in the late eighties in response to several instances of congestion collapse on the early internet.

In the intervening years, TCP's standard congestion control (Tahoe, Reno) has evolved in small ways, but no major redesign has taken place. Various problems have been shown, particularly in its achieved throughput on large bandwidth-delay product (BDP) networks and its propensity to cause high latency on network links with large available queues. In response to this, various experimental schemes have been proposed and implemented. In particular, CUBIC [Xu and Rhee, 2005] is now the default congestion control in Linux and Compound TCP [Tan et al., 2005] is now available in Microsoft Windows Vista. These algorithms have not as yet been ratified by the standards body.

Both the current standard and most of the experimental congestion control methods are fundamentally loss-based, that is they rely on packet loss to detect that the network is above full capacity. It is presumed that the loss is caused by a full network queue. However, a second means of detecting incipient congestion proposed in the late eighties is to observe that the round trip time of a packet increases as network queues build [Jain, 1989]. This has the advantage of warning early rather than merely waiting until the network is over-utilised and packets are lost. This method has rarely been used in real networks and is the main subject of this thesis.

1.1 Structure of Thesis

The thesis begins with a short review of TCP and new congestion control schemes in Chapter 2. Following this, chapter 3 explores the question of whether correlation between congestion and the delay signal of each flow on a link is really necessary for delay-based congestion control to function. It also explores the behaviour of the delay-based AIMD (DB-AIMD) algorithm in various network environments. Chapter 4 presents an experimental study into the effectiveness of a particular delay-based methodology for emptying network queues where congestion is detected. Finally, chapter 5 discussed some practical issues involved in how one measures RTT in practice for the purposes of congestion control.

1.2 Contribution of this Work

The contribution of this thesis includes:

- Establishing substantial experimental evidence that while the correlation between the delay measured by an individual flow on a link and the congestion on that link, this is not a fundamental barrier to succesful control of congestion.
- A detailed study of the DB-AIMD congestion control scheme. This explores the circumstances under which it successfully maintains operation at the "knee of the curve" and the current limitations such as a (albeit slow) linear scaling of peak network latency with number of flows on the link.
- Experimental confirmation of the efficacy of the dynamic queue emptying methodology used in H-TCP [D.J.Leith and R.N.Shorten, 2004]

and DB-AIMD [Leith et al., 2007] which is also to that used in FAST [Jin et al., 2004].

- Detailed information on some tricks, pitfalls and potential problems which must be overcome in order to accurately measure network queueing delay in TCP for the purpose of congestion control.

The work has resulted in submission of several representative publications. A paper entitled *Delay-based AIMD congestion control* appeared in the Proceedings of the Workshop on Protocols for Fast Long-Distance Networks (PFLDNet) 2007. A further two papers have been submitted: *Delay-based Congestion Control: Sampling and Correlation Issues Revisited* to Transactions on Networking; and *Making available Base-RTT for use in congestion control applications* to IEEE Communications Letters.

The work has also led to a number of patches to the linux TCP stack including

- a small bug fix accepted to 2.6.18 to H-TCP for very large BDP flows [McCullagh, 2006]
- a patch to the TSO code bounding the amount of time which sending could be deferred, written by John Heffner [Heffner, 2006]
- a change to the congestion control code and API by Stephen Hemminger which moved to using internal timing for congestion control RTT instead of RFC1323 timestamps [Hemminger, 2007]
- a patch considerably improving the way RTT is calculated internally for the purposes of congestion control [McCullagh, 2007]

most of which are discussed in Chapter 5.

Chapter 2

Congestion Avoidance in TCP

2.1 Introduction to TCP

Computer networking is usually conceived of as a hierarchy of (in this case seven) layers. The top (application) layer interfaces directly with computer programs which use the network. The bottom (physical) layer defines the details of how signalling is carried out on specific media (cable, optical, etc). The other layers in between look after everything else including how computers coexist and communicate on the medium, how they choose paths between each other, how data is broken up in transit, how the data is encoded and perhaps encrypted in transit, etc.

In practice, applications commonly pass data to the application layer which feeds it down to the next layer, etc. before the data gets sent on the medium. Each "middle-box" device involved in the transfer implements a subset of the layers. In the case of switches and routers, this is usually only the bottom two or three layers as they are all that is required to route and transfer packets. The two end points generally implement the full set, as an application on each end communicates.

Layer	Data Unit	Function
7. Application	Data	Network process to application
6. Presentation		Data Representation and Encryption
5. Session		Inter-host communication
4. Transport	Segment	End-to-end connections and reliability (TCP)
3. Network	Packet/Datagram	Path determination and logical addressing (IP)
2. Data Link	Frame	Physical Addressing (MAC & LLC)
1. Physical	Bit	Media, signal and binary transmission

Figure 2.1: OSI Model of Network Layers

TCP/IP is the most commonly used pair of complementary protocols used to implement layers three (IP) and four (TCP) on the current internet.

IP, the internet protocol defines the logical address scheme used by every internet host, the routing header for each data packet containing the source and destination address and how the path is determined from one host to another. Higher layers can therefore assume data will be routed correctly if they provide an IP address and data in suitably sized units.

TCP and UDP (among others) both implement the transport layer. UDP, the user datagram protocol is a simple method used to send datagrams to another host. It is stateless and does not ensure any reliability or ordering. Applications using UDP cannot assume each packet arrives without duplication or at all and cannot assume the order the packets will arrive in. For this reason it is commonly used in time sensitive applications (e.g. VoIP) where retransmission of lost data is useless and in short request/response applications such as DNS and broadcast/multicast applications where there may be many receivers unknown to the sender.

TCP, by contrast, is designed for reliable, ordered data transmission. TCP is responsible for breaking up data into pieces (segments) which are suitable for IP [1], reassembling the stream in order at the receiver, retransmission of lost segments and flow control. TCP is used by many applications,

In order to achieve this, the receiver advertises an amount of data which it is currently willing to accept (known as the "advertised" or "receive" window). The sender then sends no more data than the receive window allows, breaking it up into suitably sized segments. The sender places a sequence number in the header of each segment to tell the receiver at what position this packet fits in the data stream. The receiver can then reassemble the data stream on arrival using the sequence numbers and send back short acknowledgement (ACK) packets containing the sequence number of the last byte of continuous data received. This ACK packet allows the sender to verify receipt of data and the receiver to revise the receive window. If a segment arrives out of order (i.e. before a previous one in the sequence), the receiver responds by sending back a duplicate ACK, repeating the sequence number up to which it has full receipt. If the sender sees three such duplicate ACKs or a segment goes unacknowledged for a substantial time, a loss is detected and the lost packet is retransmitted.

In order that TCP can be bi-directional, every packet can simultaneously carry data being sent and an acknowledgement of data received.

2.2 Congestion Control

In the late 1980s the internet experienced a number of "congestion collapses", where the amount of data being sent began to exceed the available bandwidth

[1] The maximum segment size, MSS is the amount of data which with IP and TCP headers added will fit under the maximum transmission unit or MTU.

of the core internet backbone. This resulted in widespread packet loss, in response to which the senders resent, further exacerbating the issue. The only guide the sender had to control flow was the receive window and the speed of its local network interface, neither of which bore any relation to the available bandwidth over the link as a whole. As a result, it sent data in bursts as the receive window changed.

In response to this, Jacobson et al proposed a series of measures which the sender would implement, known collectively as "Congestion Avoidance" for TCP[Jacobson, 1988]. The main modification was to introduce a new limit to control the rate of data sending called Cwnd, the congestion window. The purpose of the congestion window is to restrict the amount of data sent so as to avoid congestion collapse. The amount of unacknowledged data is then constrained above both by the receiver window (which prevents overloading the receiver) and by the congestion window (which prevents overloading the network).

The algorithm used to calculate the congestion window was very important. It was recognised that packet loss was usually a signal that the network was congested and therefore when a loss was detected, rather than simply retransmitting, the sending rate (i.e. the congestion window) should be lowered to relieve the congestion. It was also recognised that the rate should be increased in a controlled manner. A principle of 'packet conservation' was employed in that a new packet would only be sent onto the network in response to the return of an existing packet.

The initial standard algorithm, 'Tahoe' has two modes of operation, 'slowstart' and 'congestion avoidance'. Slowstart mode is used to quickly increase up to a reasonable equilibrium rate, then congestion avoidance mode probes gradually above that. An estimated equilibrium rate is stored in sshthresh

```
if(cwnd < ssthresh)
        /* if we're still doing slow-start
         * open window exponentially */
        cwnd += 1
else
        cwnd += 1/cwnd
```

Figure 2.2: The Tahoe Increase algorithm as described in [Jacobson, 1988]

(the slowstart threshold). When Cwnd exceeds ssthresh, the flow switches to congestion avoidance mode.

Initially, Cwnd is set to one and slowstart mode operates, Cwnd is increased by one packet for each ACK received. This causes Cwnd to double each round trip time and corresponds to an exponential increase in send rate, see fig. 2.3(a). In Tahoe TCP, when loss is detected, ssthresh is set to half the current Cwnd, Cwnd is reset to one and slowstart begins again. Once Cwnd reaches ssthresh, the flow switches to congestion avoidance mode.

In congestion avoidance mode, Cwnd is only increased once each round trip time, so as to increase the sending rate roughly[2] linearly in time. At some point, the rate will again exceed the available bandwidth, the router queue will fill and packet loss will again be detected. Each time this happens, ssthresh is set to half of Cwnd, Cwnd resets to one, and slowstart begins again.

'Tahoe' distinguishes loss detected due to three duplicate ACKs (where ACKs are clearly still arriving) and loss detected due to a timeout. In the former case, congestive loss is supposed and a mechanism known as fast retrasmit is employed, immediately retransmitting the lost packet.

The next increment of the standard, 'Reno', adds the 'Fast Recovery'

[2]In reality, the increase is linear in RTT, but as the router queue length also increases with Cwnd, the RTT gets longer, hence the slightly sub-linear increase in fig. 2.3(b).

[Stevens, 1997] mechanism so that rather than resetting Cwnd to one after fast retransmit, instead it is set to the same value as ssthresh. Slowstart is therefore skipped and the algorithm proceeds in congestion avoidance mode.

This steady state behaviour, linearly increasing to congestion, then halving is usually called "Additive Increase, Multiplicative Decrease" (AIMD).

A number of extra tweaks have since been added to 'New Reno', including improvements to fast recovery as well as SACK (selective acknowledgements). However, the broad congestion control scheme remains the same.

2.2.1 Problems in Reno

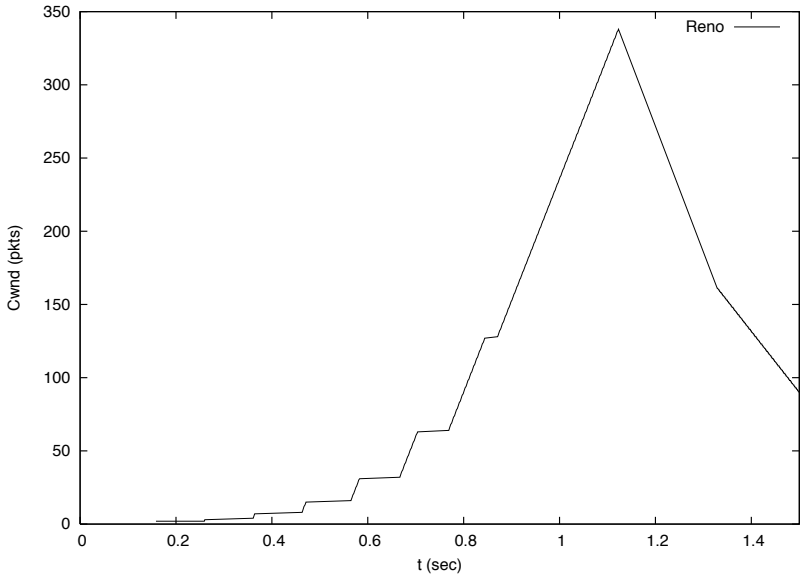
A number of problems have been identified in New Reno, particularly on large BDP links

Throughput Performance on large BDP links is poor as the linear increase function can take a long time to reach full utilisation of a link after a congestion event.

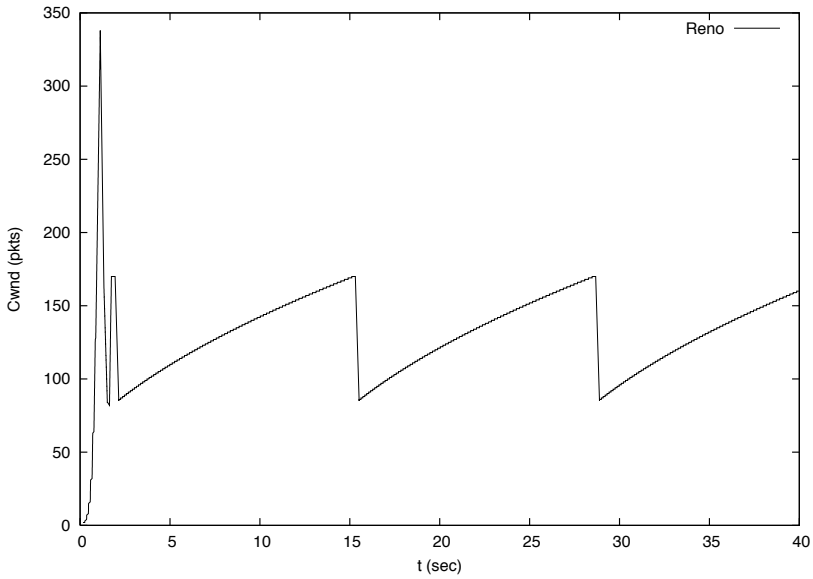
Convergence When an additional flow starts up, convergence to a steady state takes a long time, due to long times between congestion events

RTT Unfairness There is some debate as to what share of bandwidth two competing flows should have if their RTTs differ. Reno gives greater throughput to flows with shorter RTTs.

Buffer Provisioning Reno's performance is sensitive to the amount of buffering in the bottleneck link but the optimal amount of buffering depends on the RTT of a given path. If the queue is too short for a given path, throughput will suffer. If it is too long, network latency will be high as Reno will maintain a standing queue.



(a) Reno Slowstart



(b) Reno Slowstart and Congestion Avoidance

Figure 2.3: Cwnd time-history for an example Reno TCP flow on an experimental testbed with the linux kernel (v2.6.24) as sender. The initial exponential slowstart phase can be seen in the first couple of seconds, after which the linear increase phase takes over. Congestion (a lost packet) is detected at around 16 and 28 seconds and Cwnd is halved each time. Bottleneck bandwidth: 10Mbps, round trip propagation delay: 102msec, router queue length: 125KB, delayed ACKing disabled on the receiver for simplicity.

With the above problems in mind, a number of new loss-based congestion avoidance algorithms have been suggested by various researchers, including Cubic [Xu and Rhee, 2005] (the current default in Linux), H-TCP [D.J.Leith and R.N.Shorten, 2004], HS-TCP [Floyd, 2003], Scalable TCP [Kelly, 2003], and Westwood [Mascolo et al., 2001] (all of these are available in linux). Most are similar loss-based schemes which steepen the cwnd AIMD additive increase rate (α) and modify the multiplicative decrease factor (β) during the congestion avoidance phase. Some novel ideas used within them include:

Behave like Reno for small Cwnd Many of these algorithms behave like or attempt to emulate the behaviour of Reno on paths with low BDP. This can give backward compatibility on links where Reno behaviour is satisfactory (Scalable, Cubic, HS-TCP).

AIMD Increase as a function of Cwnd Scalable TCP increases Cwnd by a constant amount on each ACK (not each RTT), so Cwnd increase is a linearly increasing function of Cwnd. Similar to slowstart, this leads to an exponential increase, albeit with a lower exponent. HS-TCP increases Cwnd as a sub-linear (usually logarithmic) function of Cwnd.

AIMD Increase as a function of time between congestion events H-TCP increases Cwnd as a polynomial function of Δ, the time between congestion events. Cubic increases Cwnd as a function both of Δ and Cwnd.

Mitigate RTT unfairness H-TCP (optionally) and Cubic attempt to increase Cwnd as a function of RTT, in order that two flows competing with different RTTs get the same Cwnd.

Empty queue at congestion H-TCP estimates the minimum round trip propagation delay and the current RTT and attempts to empty the queue by setting β, the multiplicative decrease factor to the ratio of the propagation delay and the round trip time.

These algorithms are all fundamentally loss-based, i.e. loss is the primary congestion signal used[3]. Several implementations have however, made use of the queueing delay either as part of their increase function (H-TCP, CUBIC) or as part of their decrease (H-TCP, FAST).

2.3 Delay-Based Congestion Control

An alternative form of congestion control was proposed by Jain in the early nineties. In [Jain, 1989], it was suggested that the throughput and round trip time delay on a network relate to network load as shown in figure 2.4. He also pointed out that the throughput had a "knee" bend around the point where the queue began to fill and that increasing network load beyond this point would result in a minimal gain in throughput, but at the cost of extra latency due to queueing. It was therefore proposed that congestion avoidance schemes (distinct from congestion "control" schemes) should be designed to keep the network operating at the knee of the curve regardless of the amount of buffering in the network.

2.3.1 Vegas

A number of schemes were suggested including CARD [Jain, 1989], Tri-S [Wang and Crowcroft, 1991], Vegas [Brakmo et al., 1994] and later FAST [Jin et al., 2004]. Among these, one of the most thoroughly analysed and

[3]Apart perhaps from ECN, which foretells of impending loss.

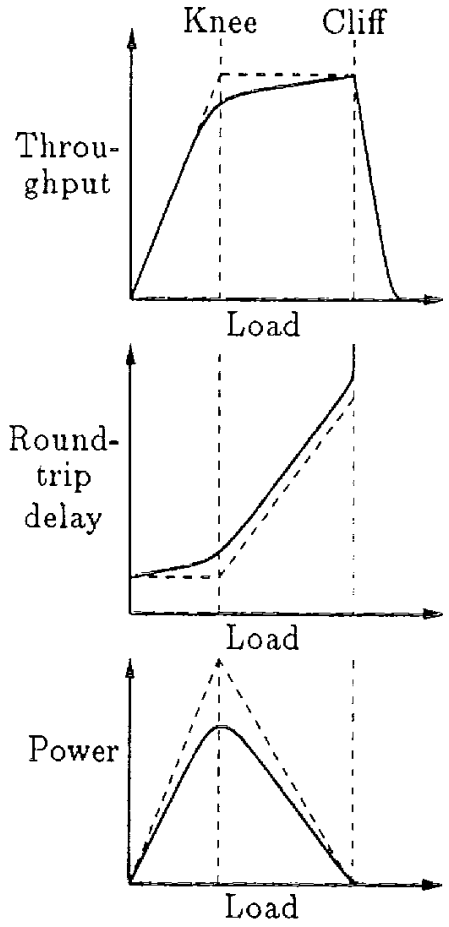


Figure 2.4: Jain's "knee of the curve" figure, taken from [Jain, 1989]

implemented algorithm is Vegas. What Vegas proposes is to estimate the expected throughput and the actual throughput being achieved based on τ, the queueing delay, $Cwnd$, the congestion window and $minRTT$, the minimum observed round trip time, which is used an estimate of the round

trip propagation delay, $baseRTT$.

$$ExpThruput = \frac{Cwnd}{minRTT} \tag{2.1}$$

$$ActThruput = \frac{Cwnd}{minRTT + \tau} \tag{2.2}$$

$$\epsilon = (ExpThruput - ActThruput) * minRTT \tag{2.3}$$

The algorithm then adjusts the window in response to changes in queueing delay as follows:

$$Cwnd \leftarrow \begin{cases} Cwnd + 1 & \epsilon < a \\ Cwnd & \epsilon \in [a, b] \\ Cwnd - 1 & \epsilon > b \end{cases} \tag{2.4}$$

where a and b are design parameters (often 2 and 4 respectively). These effectively define an upper and lower bound on the number of packets a vegas flow will attempt to maintain in the queue.

The Vegas algorithm has been observed to have a number of flaws. Some of these are specific to Vegas, but some are more general problems which are exhibited in other delay-based congestion avoidance schemes.

Knee of the Curve Each Vegas flow which operates across a bottleneck will attempt to maintain a positive number of packets in the queue (in the range a–b), regardless of the delay it experiences. As a result, the queue length increases linearly with the number of flows sharing the link. Operation at the knee of the curve is therefore only achieved where there is a small number of flows.

Measuring $baseRTT$ As Vegas flows seek always to maintain non-zero queueing delay, the queue may never empty while flows operate. New flows

must estimate the throughputs based on the minimum round trip time, but will overestimate this value if the queue is never empty. They will then overestimate the expected throughput leading to unfairness between competing flows.

Coexistence For practical reasons, it is necessary that any new algorithm can coexist reasonably with the existing ones (particularly Reno) for it to gain adoption on the internet. When competing with loss-based flows such as Reno, Vegas consistently reduces its window too early and achieves a very poor share of the available bandwidth [Ahn et al., 1995].

The first two issues noted above are specific to the Vegas algorithm, but the third seems to be a more general issue with any scheme which proposes to back off before the queue fills. Some other general issues which have been noted in the context of delay-based algorithms are:

Measuring $baseRTT$ **#2** Obtaining a reliable estimate of $baseRTT$ can be troublesome when it may change over time due, for example, to routing changes. Also, if the link is shared with loss-based flows, a standing queue may develop regardless of the actions of the delay-based flow. This issue is partially investigated in chapter 4.

Correlation Some measurements have been made [Prasad et al., 2004; Martin et al., 2003] suggesting that the correlation between the delay-signal observed by a single flow may be only very weakly correlated to the network congestion, e.g. where many flows share a single link. It is suggested that congestion control may be problematic using delay-based methods under these conditions. This issue is investigated further in chapter 3.

Reverse Path Queueing Assuming accurate $baseRTT$ estimation, the delay signal measured on a link is the sum of the queueing delay on both the forward and reverse paths. Unless these components can somehow be separated, it seems likely that delay-based flows will reduce $Cwnd$ in response to congestion on the reverse path, even though their actions may not be able to affect this and there may not be congestion on the forward path.

Noise Round-trip delay is affected not just by queueing but also by other causes unrelated to congestion, such as delayed acks, TCP segmentation offload and wireless MAC delay. Some of these issues are approached in chapter 5.

2.3.2 Delay-Based AIMD

Recently, a new experimental delay-based congestion control algorithm has been developed [Leith et al., 2007] called Delay-based AIMD. This new algorithm aims to take the existing Reno AIMD scheme and make minimal changes to produce a delay-based congestion control algorithm.

The main modification to the Reno scheme is to define an extra congestion event. The algorithm stores the minimum observed RTT (assumed to be the $baseRTT$), the current RTT and thereby infers the queueing delay, τ. When the queueing delay exceeds some threshold value, τ_0, $cwnd$ is backed off.

The congestion window therefore evolves according to:

$$Cwnd \leftarrow \begin{cases} Cwnd + \alpha/Cwnd & \tau \leq \tau_0 \\ \beta * Cwnd & \tau > \tau_0 \\ \beta * Cwnd & \text{packet loss} \end{cases} \tag{2.5}$$

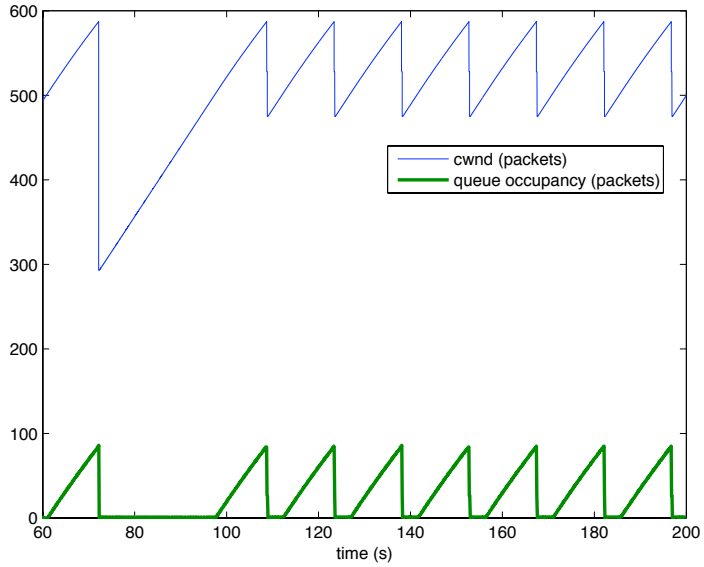


Figure 2.5: Illustrating delay-based AIMD algorithm. (*ns* simulation, delay 120ms, link rate 50Mbps, 400 packet queue, τ_0 20ms).

A suggested value for the queueing delay threshold, τ_0 is 50 msec. In order to have quick recovery from congestion events, α, the increase applied on each ACK has been initially suggested to be the same as H-TCP's cubic increase function [D.J.Leith and R.N.Shorten, 2004], though Reno's has also been used in some tests and in principle most of those used in proposed loss-based congestion control schemes could be used. For those tests used in this thesis, the Reno linear increase is used as shown in equation 2.5.

The back off factor, β is similar to that in H-TCP and is defined as

$$\beta(t) = \delta * \frac{minRTT}{RTT(t)} \tag{2.6}$$

with $RTT(t)$ the current observed round trip time and $minRTT$ the minimum observed round trip time – an estimate of the round trip propagation delay, $baseRTT$. The factor $\delta < 1$ (e.g. 0.8) is multiplied to ensure that if

$baseRTT$ were overestimated and the ratio is too large to empty the queue, each successive back off will empty further, forcing $minRTT$ down to its correct value. This choice of β was first proposed in [Leith et al., 2007] and is investigated in more detail in chapter 4.

2.3.3 Hybrid Congestion Control

In addition to algorithms which use delay as a congestion indicator, a further category of congestion control has been proposed which is in a sense a hybrid of delay and loss-based congestion control. The two best-known current examples are Compound TCP (which is suggested as an option in Microsoft Windows) and TCP Illinois (available in linux). These algorithms seek to be more aggressive than Reno where queueing delay is minimal, but to fall-back to a Reno behaviour in the presence of queueing delay, as detected by monitoring the round trip time.

Compound TCP [Song, 2006], for example uses a congestion window which is the sum of two components, the regular Reno Cwnd component and a Dwnd (delay window) value which is calculated in a similar same way to Vegas. TCP Illinois [Liu et al., 2006] uses loss as the congestion signal to decrease Cwnd, but sets α, the rate of increase and β the magnitude of decrease, as a function of averaged round trip delay.

2.4 Summary

This chapter has provided a brief introduction to TCP and its standard congestion control. Recently proposed extensions for large bandwidth delay product networks have also been summarisd including Cubic, the default linux scheme and Compound, Microsoft's congestion control scheme which

is available in Windows Vista.

Delay-based congestion control has also been explored, particularly Vegas and the experimental delay-based AIMD which is explored in more detail in later chapters.

Chapter 3

Delay-based Congestion Control: Sampling and Correlation Issues

3.1 Introduction

In this chapter we revisit the recently voiced concern that low correlation between measured delay and packet loss events means that delay may be fundamentally flawed as a signal for congestion control. A related concern is that on heavily multiplexed links the measured delay may be only weakly correlated with the congestion window of a flow [Prasad et al., 2004]. These concerns are particularly topical in view of a number of proposals to change the TCP congestion control algorithm to make use of delay information. Examples include not only FAST TCP [Jin et al., 2004] but also hybrid congestion control algorithms based on the use of both loss and delay, e.g. TCP Illinois [Liu et al., 2006] and Compound TCP [Tan et al., 2005]. The latter is now available in Windows Vista and is currently undergoing review

at the IRTF and IETF standards bodies.

A number of recent independent measurement studies [Biaz and Vaidya, 2003; Martin et al., 2003; Rewaskar et al., 2005] have indeed found that there may be only low correlation between packet loss events and the delay measured by a flow. That is, when packet loss occurs, and thus some network queue is full, nevertheless all flows need not observe high delay. In fact, any given TCP flow may observe high delay at only a small proportion of packet loss events. This behaviour is confirmed by our own experimental measurements.

[Martin et al., 2003], [Prasad et al., 2004] consider possible reasons for the low correlation observed and suggest that sampling issues are a fundamental factor. To see this, consider Figure 3.1 which presents an example queue occupancy time history at a bottleneck link carrying many flows. Packets from a single flow "sample" the queueing delay at the bottleneck link. However, when many flows share a link, the proportion of queued packets that are associated with a given flow can become small. The queueing delay is then only very sparsely sampled by that flow – for example, the "samples" for one flow are marked by solid squares in Figure 3.1. As a result, the flow cannot accurately estimate the state of the queue and may fail to detect even large changes in queue occupancy and, in particular, may fail to detect queue full events. Thus, in general, the correlation between the queueing delay measured by a given flow and the actual queueing delay at a bottleneck link may be low.

While the possibility of low correlation between packet loss events and the delay measured by a flow thus seems well established, our aim in this chapter is to investigate the implications for congestion control. A natural concern is that low correlation between measured delay and packet loss events

means that delay measurements are *prima facie* an inadequate indicator of network congestion and thus their use for congestion control could be fundamentally flawed. Indeed, precisely such concerns are raised in [Martin et al., 2003; Prasad et al., 2004; Rewaskar et al., 2005]. This potentially has direct implications not only for recent delay-based proposals (FAST, Compound TCP etc) but also for our understanding of the fundamental constraints on congestion control within the current Internet architecture.

Our main contribution in this chapter is to demonstrate that in fact what matters for congestion control is the *aggregate* behaviour of the flows sharing a link. Hence, perhaps somewhat surprisingly, while any given single flow may measure delay which is only weakly correlated with the actual queueing delay, this is not in itself an obstacle to congestion control. In this chapter we demonstrate this constructively via detailed experimental tests and also confirm analytically the general nature of our conclusions.

It is important to emphasise that this result does not preclude the existence of other factors that may limit the practical application of delay-based congestion control, it only states that low correlation is not itself an obstacle. For example, obtaining good delay measurements in the presence of "noise" such as delayed acking and hardware offload is also potentially an important issue – in fact we touch on this issue here although it is not the main focus of the present chapter. Despite the existence of such issues that require further study, we nevertheless argue that the work here is an important first step in exploring the nature of fundamental constraints on congestion control.

The chapter is organised as follows. In Section 3.3 we consider the requirement for correlation between delay and loss events, analyse the congestion control properties of the delay-based AIMD algorithm and establish conditions under which it achieves congestion control, in the sense of bounding

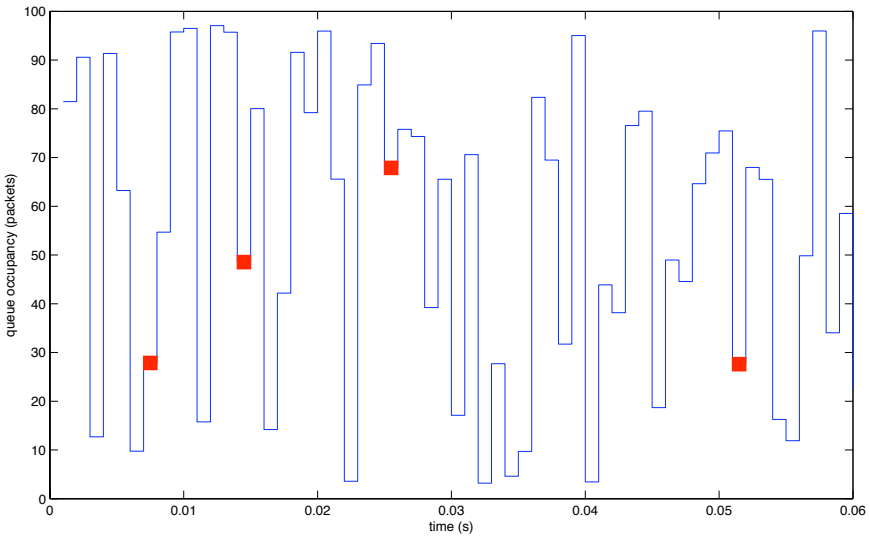


Figure 3.1: Illustrating sampling issues when estimating queue state – example queue occupancy time history with packets of one selected flow marked by solid squares.

the queue occupancy. Section 3.4 presents experiments to validate this analysis, including in heavily multiplexed regimes. We briefly summarise our conclusions in Section 3.6. Further experimental analysis of performance of delay-based AIMD in the presence of delayed acking and TSO is described separately in chapter 5.

3.2 Related Work

In parallel with work on the development of delay-based algorithms, a number of concerns have been raised as to practicality of delay as a congestion signal. Potentially one of the most serious is the observation that there can be low correlation between measured and actual queueing delay and loss. This is discussed in detail in the experimental studies in [Biaz and Vaidya, 2003;

Martin et al., 2003; Rewaskar et al., 2005]. These studies do not investigate any specific delay-based algorithm but rather make use of experimental measurements to evaluate correlation. [Prasad et al., 2004] considers a number of possible reasons for low correlation to occur, including in particular sampling issues.

3.3 Is low correlation an obstacle to congestion control ?

On the face of it, the existence of situations where low correlation exists between the queueing delay measured by a flow and the actual queueing delay appears to create an obstacle to congestion control. That is, how can we expect to succeed at congestion control if some flows are unable to reliably detect congestion events where the queue occupancy is high. We demonstrate that in fact what matters for congestion control is the *aggregate* behaviour of the flows sharing a link. Hence, while any given single flow may measure delay which is only weakly correlated with the actual queueing delay, this is not in itself an obstacle to achieving congestion control.

To explore this question we use the delay-based AIMD algorithm as an example and investigate its congestion control behaviour in more detail. We note that our purpose here is not to advocate use of the delay-based AIMD algorithm. Rather the delay-based AIMD algorithm is simply one approach to delay-based congestion control that happens to provide a useful vehicle for demonstrating some fundamental issues in a concrete manner. Our analysis makes use of the following basic observation.

Key Observation. Sparse sampling of the queue occupancy means that

packets from some flows may not detect an event where the queueing delay rises above a threshold τ_0. Nevertheless, if the queueing does rise above τ_0 then we *always* have that some packets do experience queueing delay greater than τ_0 – namely, the very packets that are responsible for filling the queue above threshold τ_0.

We immediately have that as long as the queueing delay remains above threshold τ_0, then each RTT the delay-based AIMD algorithm will lead to at least one flow backing off. Moreover, since every packet participating in the overshoot in queue occupancy above τ_0 measures the high queueing delay, the magnitude of the aggregate flow backoff is roughly proportional to the overshoot in queue occupancy (we make this statement more precise below). Intuitively, this creates pressure to drain the queue occupancy below τ_0, thereby achieving congestion control. This argument does not require that every flow be able to detect events when the queueing delay becomes high, it only requires that in aggregate the flows respond to each such congestion event. By the foregoing key observation, the latter is always the case. We develop this argument in more detail next.

3.3.1 Congestion Control Analysis

Consider n delay-based AIMD flows sharing a common bottleneck. Let w_i, $baseRTT_i$ be the respective cwnd and round-trip propagation delay of flow i. Let $W(k) = [w_1(k), \cdots, w_n(k)]$ where k corresponds to the time of the k'th congestion event, i.e. a network event where at least one flow backs off its cwnd. Figure 3.2 illustrates an example cwnd and queue time history. From

the AIMD algorithm we have that

$$w_i(k+1) = \beta_i(k)w_i(k) + \alpha_i T_i(k) \tag{3.1}$$

where α_i is the AIMD increase parameter in packets/RTT, β_i is the AIMD backoff factor with $\beta_i(k) = 1$ corresponding to no back off of flow i at event k and $\beta \leq \beta_{max} < 1$ otherwise. $T_i(k)$ is the number of flow i RTTs between congestion events k and $k+1$.

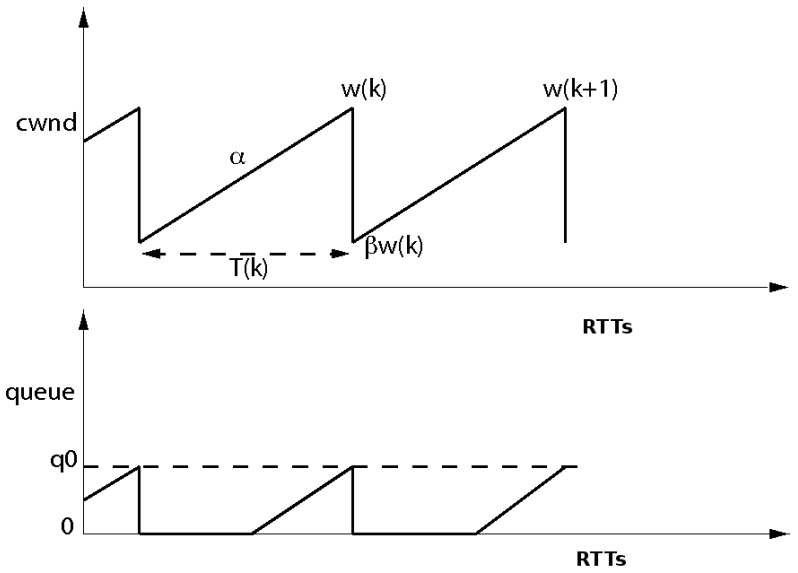


Figure 3.2: Illustrating cwnd and queue time histories.

The queue occupancy $q(k)$ at the k'th congestion event is

$$q(k) = \sum_{i=1}^{n} [w_i(k) - b_i(k) baseRTT_i] \tag{3.2}$$

where $b_i(k)$ is the bandwidth being consumed by flow i and $\sum_{i=1}^{n} b_i(k) = B$ with B the link bandwidth in packet/s.

We let q_0 denote the queue occupancy corresponding to the backoff delay threshold τ_0. Note that this implicitly assumes that a one to one correspondence exists between queue occupancy and queueing delay. This assumption is satisfied, for example, for any link with constant service rate.

Also assume, for the moment, that a delay-based flow backs off its cwnd whenever one or more packets measures queueing delay above threshold τ_0. We return to this assumption later in Section 3.3.4. We then have the following identity:

$$\begin{aligned} \sum_{i=1}^{n} \beta_i(k) w_i(k) &\leq \beta_{max}(q(k) - q_0) + \sum_{i=1}^{n} w_i(k) - (q(k) - q_0) \quad (3.3) \\ &\leq \sum_{i=1}^{n} w_i(k) + (\beta_{max} - 1)(q(k) - q_0) \quad (3.4) \end{aligned}$$

Hence, combining (3.2) and (3.4),

$$\begin{aligned} q(k+1) &= \sum_{i=1}^{n} [w_i(k+1) - b_i(k+1) * baseRTT_i] \\ &= \sum_{i=1}^{n} [\beta_i(k) w_i(k) + \alpha_i T_i(k) - b_i(k+1) * baseRTT_i] \\ &\leq \beta_{max} q(k) + (1 - \beta_{max}) q_0 + \sum_{i=1}^{n} \alpha_i T_i(k) \\ &\quad + \sum_{i=1}^{n} [w_i(k) - b_i(k+1) * baseRTT_i - q(k)] \quad (3.5) \end{aligned}$$

When flows are synchronised, in steady state[1] $b_i(k+1) = b_i(k)$ and the terms in the square brackets sum to zero. Hence, the queue length is constrained

[1]The existence of a unique, stable steady-state solution for synchronised AIMD-based TCP networks is shown in, for example, [Shorten et al., 2004].

by

$$q(k+1) \leq \beta_{max} q(k) + (1-\beta_{max}) q_0 + \sum_{i=1}^{n} \alpha_i T_i(k)$$

Since $\beta_{max} < 1$ this recursion is convergent and so as $k \to \infty$

$$q(k) \leq q_0 + \sum_{i=1}^{n} \alpha_i T_i(k)/(1-\beta_{max}) \tag{3.6}$$

Since the time between congestion events is necessarily bounded (trivially, since the network capacity is bounded the flow cwnds cannot increase indefinitely), we have that $q(k)$ is upper bounded.

The bound in (3.6) is potentially very conservative. Nevertheless, it establishes that a network of synchronised delay-based AIMD flows will always converge to operation with bounded queues and so achieve congestion control. This result holds even when the delay measured by any given single flow may be weakly correlated with queue excursions above the threshold τ_0. Thus we have established that low correlation is not in itself an obstacle to achieving congestion control. Of course we need to validate this theoretical analysis via experimental measurements and this is the subject of Section 3.4. Before that, however, we first consider some important extensions of our analysis.

3.3.2 Two important cases

Tighter bounds on the queue occupancy can be readily obtained in two common cases.

Case 1: Queue occupancy falls below q_0 (i.e. below the delay threshold τ_0) on flow cwnd backoff following congestion event k. In the worst case (i.e.

corresponding to peak queueing delay), congestion event $k+1$ occurs when all flows increase their cwnd's at the point where the queue is just below q_0. That is,

$$q(k+1) \leq q_0 + \sum_{i=1}^{n} \max(\alpha_i, 1) \tag{3.7}$$

The bound (3.7) has a natural interpretation. Due to network delays, no flow can detect an event where the queue exceeds τ_0 until at least one RTT after the event. During this RTT it can therefore happen that every flow increases its Cwnd and injects additional packets into the network. As a result, the size of the resulting queue overshoot can, in the worst case, be proportional to the number n of flows.

Case 2: Queue occupancy remains above q_0 following congestion event k. In this case the time until the next congestion event is $\leq max_{i\in\{1,2,..,n\}} t_i$ and so

$$q(k) \quad \leq \quad q_0 + \sum_{i=1}^{n} \max(\alpha_i, 1)/(1-\beta_{max}) \tag{3.8}$$

3.3.3 Unsynchronised Flows

The foregoing analysis is for networks where flow backoffs are synchronised. The analysis can be readily generalised to situations where the flows are not synchronised. In more detail, taking expectations in (3.5) yields

$$E[q(k+1)] \leq \beta_{max} E[q(k)] + (1-\beta_{max}) q_0 + \sum_{i=1}^{n} \alpha_i E[T_i(k)] + \sum_{i=1}^{n} [E[w_i(k)] - E[b_i(k+1)] t_i - E[q(k)]]$$

In steady state[2] $E[b_i(k+1)] = E[b_i(k)]$, the terms in the square brackets sum to zero and length is constrained by

$$E[q(k+1)] \;\leq\; q_0 + \sum_{i=1}^{n} \alpha_i E[T_i(k)]$$

Thus our conclusion that low correlation is not an obstacle to congestion control remains unchanged in unsynchronised networks.

3.3.4 Filtered delay measurements

The foregoing analysis assumes that the delay-based AIMD algorithm backs off its cwnd whenever one or more packets measure queueing delay above threshold τ_0. However, in practice we expect to use a smoothed estimate of queueing delay in order to avoid backing off in response to spurious delays. In other words we expect to backoff cwnd only after a sufficient number of packets have experienced high queueing delay.

The choice of an appropriate smoothing filter is a design question that is outside the scope of the present chapter. Instead, our interest here is in understanding the impact such smoothing may have on the congestion control properties of a delay-based algorithm. One direct approach is to seek a class of filters under which the previous analysis remains applicable. It can be seen immediately that as long as we maintain the identity (3.4), then the general bound (3.6) remains unchanged. For example, this holds when we

[2]Under mild assumptions, the existence of a unique stationary distribution for unsynchronised TCP networks is shown in, for example, [Shorten et al., 2006].

modify the delay-based AIMD algorithm to be

$$cwnd \leftarrow \begin{cases} cwnd + \alpha/cwnd - \gamma, & \text{on each ACK} \\ \min[\beta(cwnd + S_\gamma), cwnd], & \text{if } \tau \geq \tau_0 \\ \beta \times cwnd, & \text{if packet loss} \end{cases}$$

where $\gamma > 0$ if the currently acknowledged packet has experienced queueing delay above τ_0 (this may be estimated via the packet time stamp), otherwise $\gamma = 0$. Thus the algorithm makes a small reduction in cwnd when individual packets experience high delay. This satisfies (3.4) by ensuring that cwnd is always backed off at least in proportion to the number of packets with high queueing delay. S_γ is a running total of γ values and is reset to zero on a full backoff (i.e. when $\tau \geq \tau_0$). It is therefore the overall amount, since the last full backoff, subtracted from cwnd due to individual delayed packets. We use S_γ to adjust the decrease in cwnd at a full backoff to avoid double counting of delayed packets. The delay signal τ triggering full backoff may be any smoothed estimate of queueing delay, thereby decoupling the baseline congestion control behaviour of the algorithm from the choice of smoothing filter.

The per packet backoff factor γ may differ from the standard backoff factor β. One natural choice is $\gamma = 1 - \beta$ which ensures back off to $\beta \times cwnd$ when a full windows worth of packets are delayed. Since we might expect to choose the smoothing filter such that τ exceeds τ_0 when a full cwnd of packets exceeds τ_0 (indeed, perhaps when less than a full cwnd of packets exceed τ_0), then the per packet backoff simply acts as a conservative "safety net" .

Again, we have that low correlation is not an obstacle to congestion control even when a filtered delayed signal is used.

3.4 Experimental Measurements

To help build confidence in the validity of the foregoing analysis for real network traffic, in this section we explore delay-based congestion control behaviour using experimental measurements taken on a hardware testbed. Use of experimental tests seems particularly important in the context of delay-based control as issues such as scheduling granularity, hardware offload, packet bursts *etc* are difficult to model accurately yet may have a direct impact on delay measurements and performance.

This section is organised as follows. We begin by demonstrating that low correlation between delay and loss is prevalent on heavily multiplexed links. Initially delayed acking is disabled, as is TCP segmentation offload (TSO) in order to focus on correlation issues. We explore the congestion control behaviour of the delay-based AIMD algorithm and compare experimental measurements with the analysis in Section 3.3. In Section 3.4.3 we discuss in more detail some observations on asymptotic behaviour as the number of flows becomes very large. In Section 3.4.5 we enable delayed acking and TSO and consider the impact on delay measurement and congestion control. A full description of the experimental setup can be found in Appendix A.

3.4.1 Illustrating low correlation

We begin by considering a heavily multiplexed link where we expect low correlation between delay and loss to be prevalent. Figure 3.3 demonstrates the congestion control action of the delay-based AIMD algorithm on a 10Mbps link shared by 80 TCP flows. The flows have a range of round-trip times from 20-200ms. Figure 3.3(a) plots the correlation between the queueing

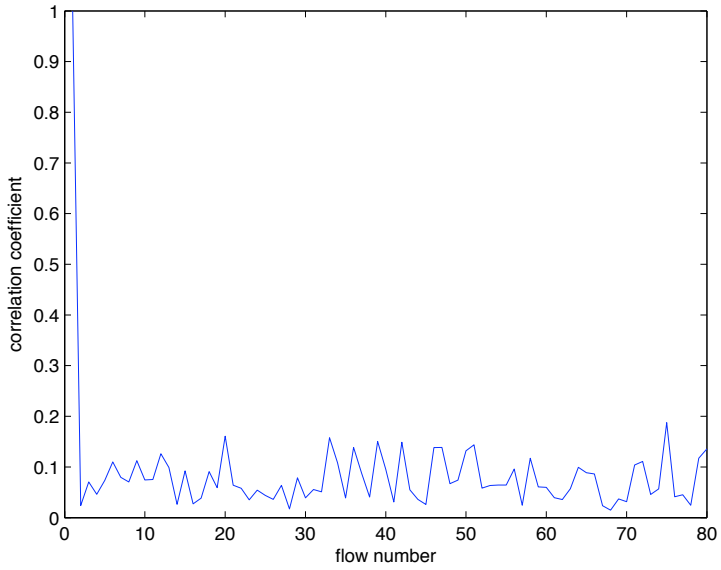
delay measured by flow 1 and the queueing delay measured by flows 2-80[3]. These results are representative, with similar correlation values obtained for flows other than flow 1. It can be seen that the correlation between measured queueing delay is close to zero for all flows. Figure 3.3(b) shows typical time histories of the measured queueing delay (i.e. measured RTT minus the known propagation delay), and the low level of correlation between measurements is evident.

Despite the low correlation between measured queueing delay, the delay-based algorithm successfully regulates the flow cwnds to prevent queue overflow and congestion. This is illustrated in Figure 3.4(a) which plots the sum of the flow cwnds while Figure 3.4(b) plots typical cwnd time histories for individual flows. During this test run no packet losses occurred. Note that the small flow cwnds in Figure 3.4(b) are feature of the link being highly multiplexed and reflect the fact that here we are intentionally seeking to explore situations where flow delay measurements may be weakly correlated.

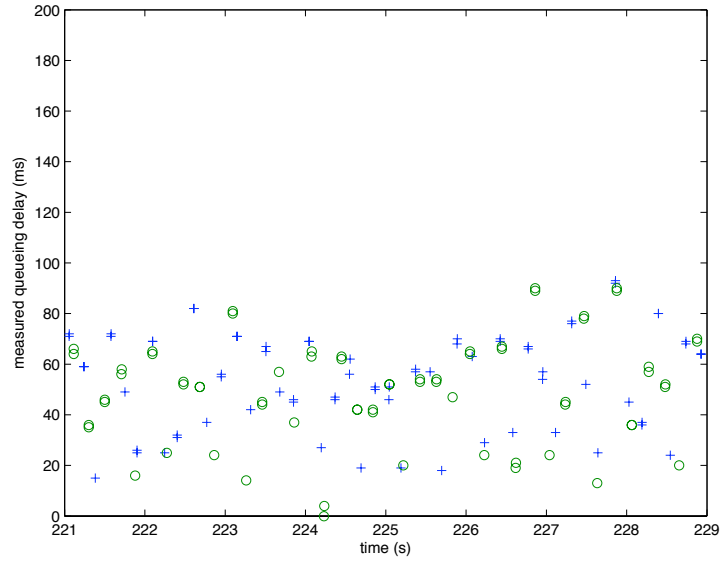
3.4.2 Peak queueing delay vs number of flows

To explore congestion control performance in more detail we consider the measured queueing delay as a function of the number of flows sharing a link. Figure 3.5 plots the mean and peak queueing delay as the number of competing flows is varied on a link. Also marked on Figure 3.5 is the analytic bound (3.7). This worst case bound captures the fact that no flow can detect a queue overshoot above τ_0 until one RTT after it occurs. Thus it can happen that all flows increase their cwnd and insert extra packets with one RTT, creating an overshoot above τ_0 that is proportional to the

[3]Queueing delay is the measured from the packet time-stamps less $baseRTT$, where $baseRTT$ is the minimum observed delay. We confirmed that flow $baseRTT$ estimates of propagation delay were accurate. Time histories are aligned based on the peak correlation.

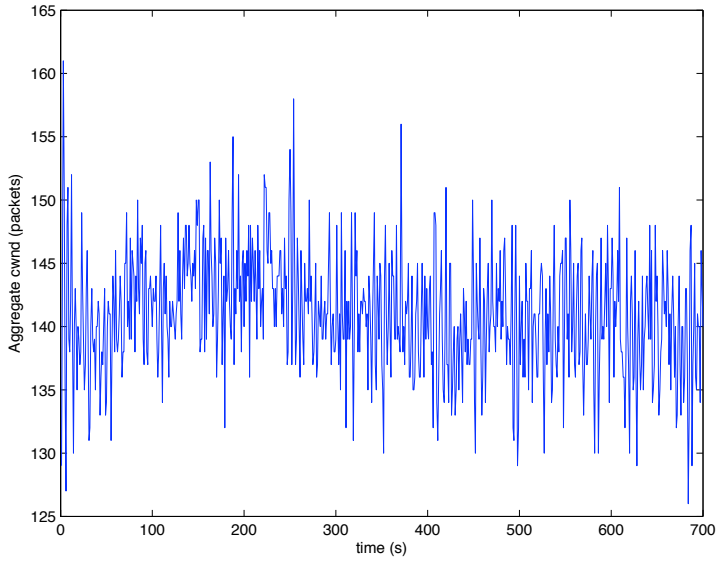


(a) Correlation between queueing delay measured by flow 1 and by flows 2-80.

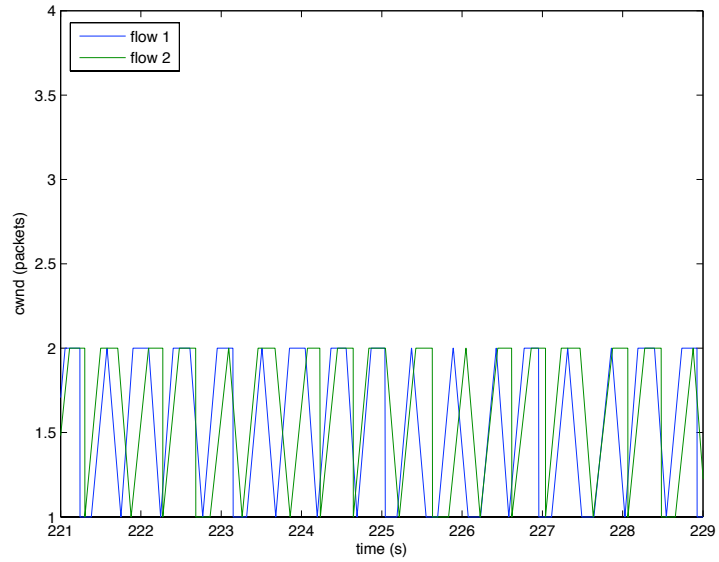


(b) Time histories of measured queueing delay for flow 1 and flow 2.

Figure 3.3: 80 concurrent, long-lived flows. 10Mbps link, flow baseRTTs uniformly randomly chosen from 20-200ms.



(a) Sum of flow cwnds.



(b) cwnd time histories for flow 1 and flow 2.

Figure 3.4: 80 concurrent, long-lived flows. 10Mbps link, flow baseRTTs uniformly randomly chosen from 20-200ms.

number of flows. It can be seen that as we increase the number of flows the peak queueing delay the analytic bound is generally quite tight. This not only provides a degree of validation of the analysis in Section 3.3 but also provides a concrete demonstration that a delay-based algorithm can indeed achieve effective congestion control under low correlation conditions.

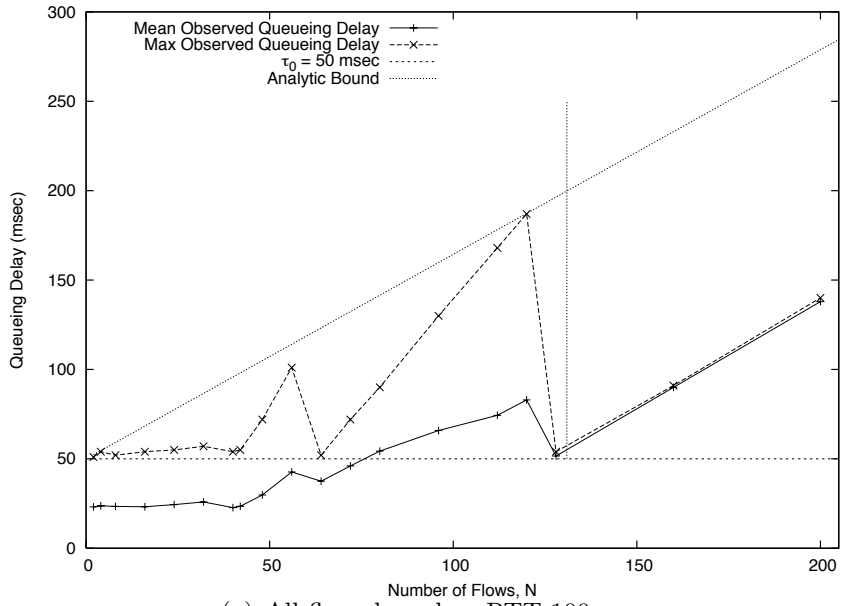
Investigating these experimental results in more detail, it can be seen from Figure 3.5 (and Figure 3.7) that the peak queueing delay does not increase monotonically with the number of flows but rather may decrease as the number of flows increases. This arises due to quantisation of the number of packets in flight. To see this consider n flows sharing a link with bandwidth-delay product P packets and with queue occupancy q_0 packets corresponding to the queueing delay threshold τ_0. Assume, for simplicity, that the flows are perfectly synchronised and have the same cwnd. Let $\underline{cwnd}(n) = \max\{j : j \times n - P < q_0, j \in \{0, 1, 2, ...\}\}$. That is, $\underline{cwnd}(n)$ is the largest cwnd such that the queueing delay still remains below the threshold τ_0. Overshoot in queue occupancy above q_0 will then occur on the next round-trip time when flows increase their cwnd by one packet, with the magnitude of the overshoot being $(\underline{cwnd}(n) + 1) \times n - P - q_0$. As we increase the number of flows to $n + k$, then the flow cwnd just before backoff initially remains unchanged i.e. $\underline{cwnd}(n + k) = \underline{cwnd}(n)$, provided $\underline{cwnd}(n) \times (n + k) - P < q_0$. Eventually, however, as $n + k$ increases further $\underline{cwnd}(n + k)$ must reduce to be smaller than $\underline{cwnd}(n)$. At this point the delay overshoot will also decrease, leading to non-monotonic behaviour of the overshoot. It can seen from Figures 3.5(a) that, as might be expected, this effect is most pronounced when all flows have the same RTT and so are almost synchronised. When flows have different RTTs, as in Figure 3.5(b), and are unsynchronised the overshoot tends to show a simple rising trend.

Also plotted in Figure 3.5 is the mean queueing delay vs number of flows. It can be seen that the mean delay also displays a rising trend. This has implications for performance and operation at the "knee of the curve". The potential exists to modify the delay-based algorithm to mitigate this effect e.g. by adjusting the threshold τ_0 based on observed overshoot in delay. However, we do not explore this possibility here.

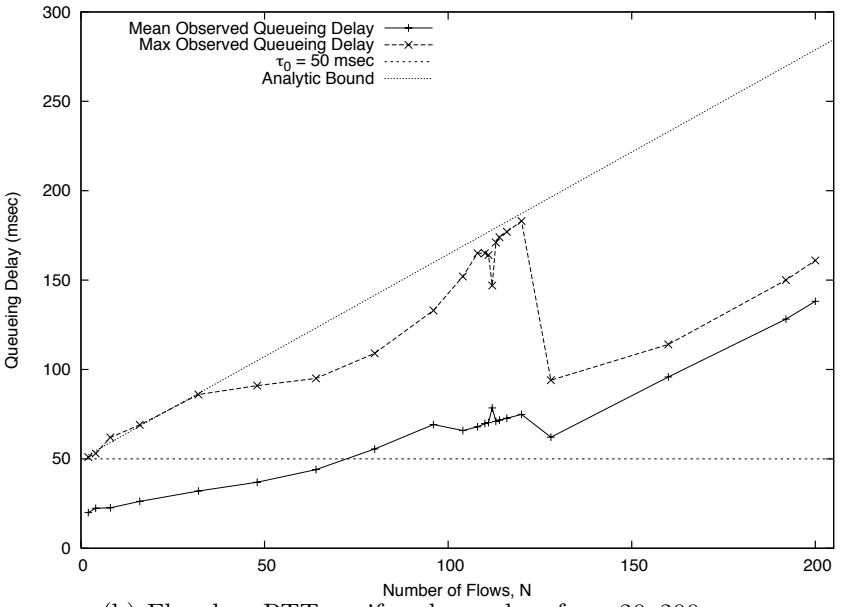
3.4.3 Asymptotic behaviour

While our experimental results demonstrate that low correlation need not be an obstacle to congestion control, they also highlight that other issues can arise on links with very large numbers of flows. In particular, as the number of flows is increased the flow cwnd's will eventually decrease until they are only one packet in size. This is inevitable since newly added flows must have a cwnd of at least one packet and therefore existing flows must, if possible, reduce their cwnd to make space for new flows (or rather backoff their cwnds to maintain the queueing delay below τ_0). Eventually, however, we will reach the situation where all flows have reduced their cwnd to one packet.

This behaviour is illustrated in Figure 3.6, which plots the flow cwnds as the number of flows is increased. It can be seen that as the number of flows is increased the flow cwnds tend to fall, until eventually all flows have cwnd of one packet. The point where this occurs can be calculated as follows. In Figure 3.6(a) flows with a base RTT of 100ms share a 10Mbps link. The number of packets required to fill the pipe and create a queueing delay of τ_0 is $B(baseRTT + \tau_0)$, where B is the link rate in packets per second. With τ_0=50ms and 1500 byte packets, $B(baseRTT+\tau_0)$=130 packets i.e. our limit is 130 flows with cwnd of 1 packet. This is confirmed by inspection of Figure 3.6(a). In the mixed base RTT case shown in Figure 3.5(b) it is harder to

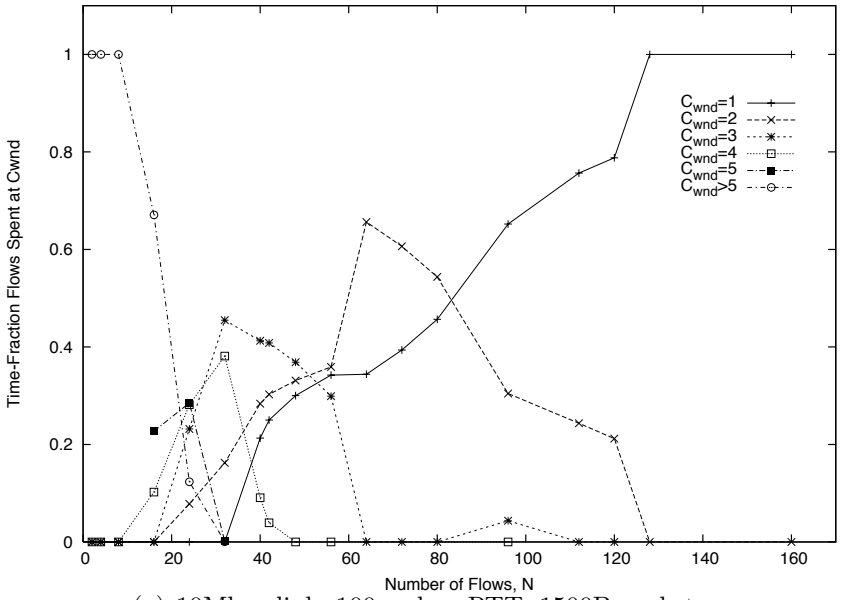


(a) All flows have baseRTT 100ms.

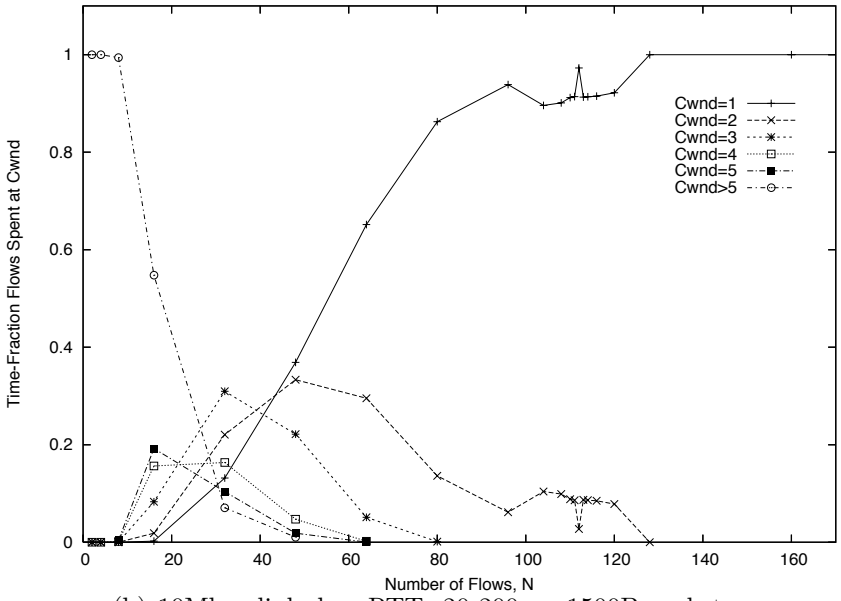


(b) Flow baseRTTs uniformly random from 20–200ms.

Figure 3.5: Peak and mean queueing delay vs number of flows. 10Mbps link. Bound (3.7) is labelled "Analytic Bound". The vertical line in (a) marks the point at which the number of flows equals the path bandwidth-delay product.



(a) 10Mbps link, 100ms baseRTT, 1500B packets.



(b) 10Mbps link, baseRTTs 20-200ms, 1500B packets

Figure 3.6: Fraction of time that flows take values of *cwnd* vs the number of flows. It can be seen that flow *cwnd*'s tend to decrease as the number of flows increases, until all flows have *cwnd* of one packet.

analytically calculate the number of flows where the limiting regime occurs. Nevertheless, the qualitative behaviour is similar as can be seen from Figure 3.6(b).

In this asymptotic regime, each additional new flow leads to an increase in the level of queue occupancy. This occurs despite the fact that the queueing delay may then remain persistently above τ_0, since flows all have cwnd of one packet and so cannot backoff their cwnd further to reduce the queueing delay. Note also that when the delay is persistently above τ_0, delay-based AIMD flows will not increase their cwnd. Hence, we have that the queueing delay simply increases linearly with the number of flows. This behaviour is evident in Figure 3.5(b) when the number of flows is greater than 130 – it can be seen that the delay rises linearly, parallel to the analytic bound (3.7).

We discuss SACK Reno in more detail in the next section, but note here that once the cwnd of a flow falls below three packets, fast retransmit no longer operates (since it requires two duplicate acks after loss of one packet) and congestion control falls back retransmit timeouts (RTO). SACK Reno behaviour is the asymptotic regime is thus complex and, for example, prone to prolonged unfairness between competing flows due to sensitivity to loss of retransmitted packets when in RTO.

The fact that flow cwnds are constrained to have a minimum value of one packet appears to place a limit on the use of delay-based congestion control. Namely, as the number of flows is increased to the point where a flow cwnd of less than one packet is needed to maintain low queueing delay, then low delay operation becomes impossible and packet loss eventually occurs as the number of flows becomes sufficiently large.

The limit is, however, not a fundamental one. For example, while cwnd is constrained to be at least one packet, we might reduce the packet size

to prevent queue buildup. For example, we re-ran our experiments using a packet size of 512 bytes rather than 1500 bytes. Figure 3.7 shows the corresponding results – we can now fit more than 390 flows on the link before the flow cwnds are all reduced to only one packet. Reducing packet size may not be an attractive solution, however, as it increases the transmission overhead (packet headers and link framing overhead remain unchanged as the packet payload is decreased in size). One alternative is to insert delays between packet transmissions so as to pace packets at a slower rate, which would soften the impact of the one packet lower bound on cwnd. However, situations where this limit is reached are essentially corner cases where flows are all getting very low throughput and the user experience is likely to be poor regardless of changes to TCP. For example, on a 10Mbps link with 130 flows, each flow is has a throughput share of less than 75Kbps, i.e. similar to a dialup modem. With 512B packets, for >390 flows the per share is less than 25Kbps.

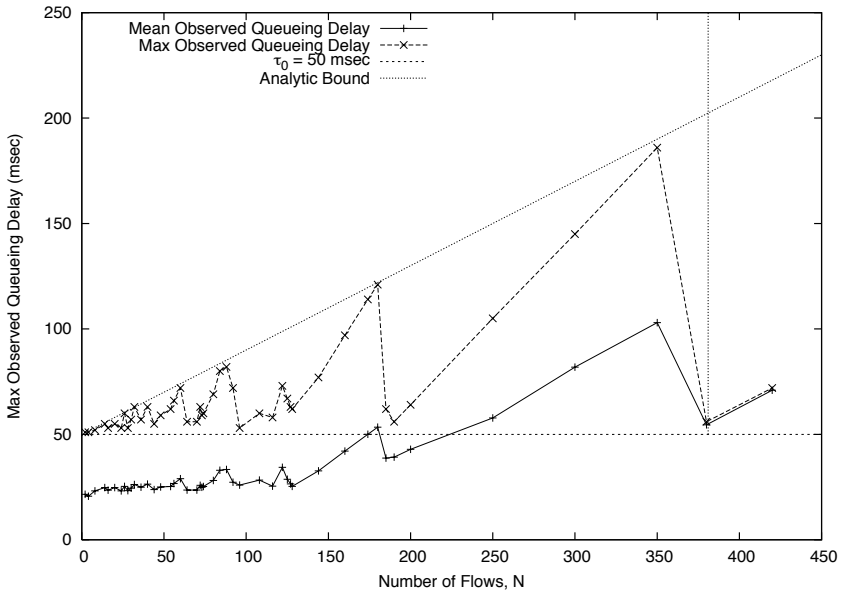
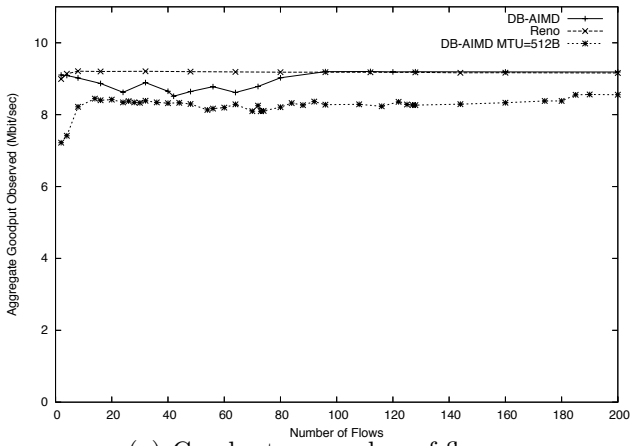


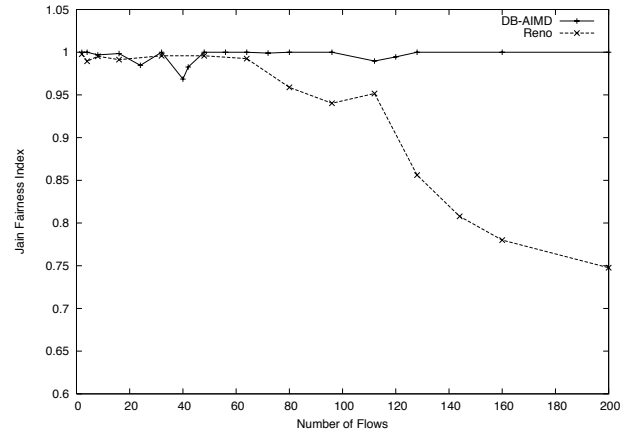
Figure 3.7: Peak and mean queueing delay vs number of flows. 10Mbps link, 100ms baseRTT, 512B packets.

3.4.4 Comparison with SACK Reno

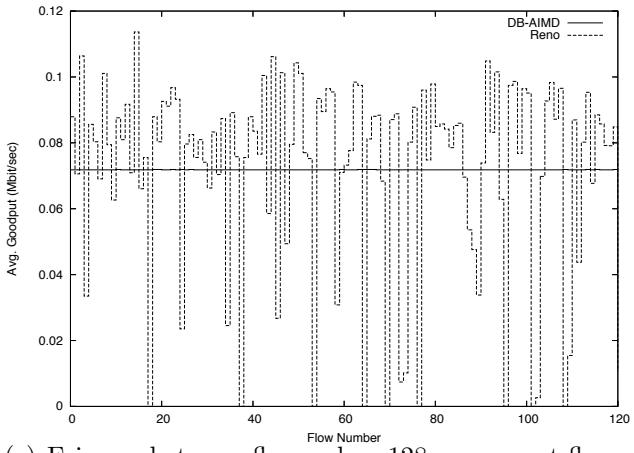
To illustrate that effective congestion control is indeed being achieved by the delay-based algorithm even when correlation is weak, it is informative to compare behaviour with that of the standard SACK Reno loss-based TCP. Figures 3.8 and 3.9 compare the performance of the delay-based algorithm with that of the standard Linux SACK Reno algorithm. Figure 3.8 show measurements when flows have the same base RTT of 100ms and Figure 3.9 shows measurements when the flow base RTTs are distributed between 20-200ms. It can be seen that for a given packet size link utilisation is at worst slightly reduced with delayed-based AIMD i.e. the low delay achieved by the delay-based AIMD algorithm does not come at the cost of a significant lowering of throughput. As discussed previously, the negative impact on throughput of using a smaller packet size is evident in Figure 3.8(a). While the aggregate link utilisation is similar, it can be seen from Figure 3.8(b) that the delay-based algorithm achieves better inter-flow fairness than Reno on heavily-multiplexed paths. It can also be seen from Figure 3.9(b) that the difference in fairness behaviour is less pronounced when there is wider mix of flow RTTs, and so of flow cwnds. At low cwnds, fast recovery is ineffective and congestion control in Reno reverts to RTO operation. Unfairness then arises because flows in RTO become very sensitive to packet loss – following an RTO, if the first retransmitted packet is lost then the RTO timer is doubled and can easily increase to several seconds duration. In contrast, the delay-based algorithm avoids packet loss even in quite extreme regimes.



(a) Goodput vs number of flows.

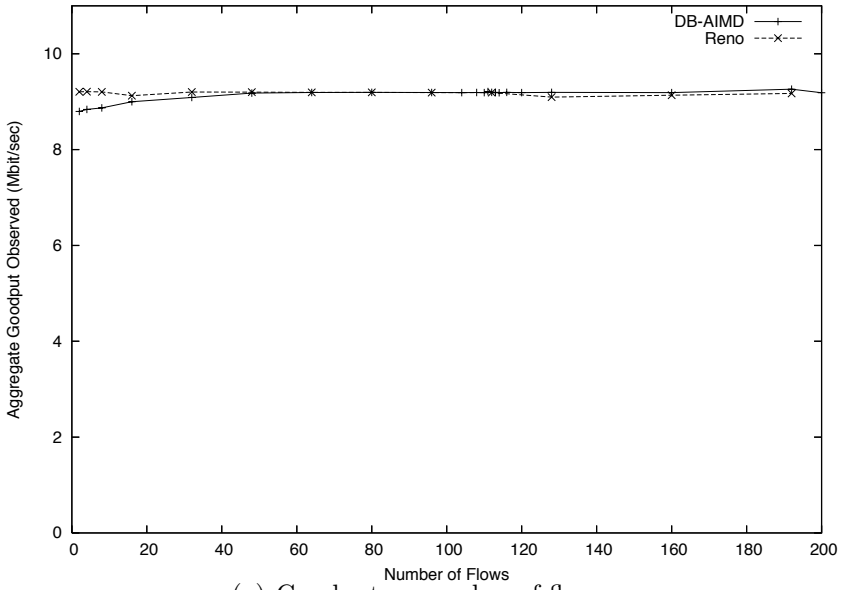


(b) Fairness vs number of flows.

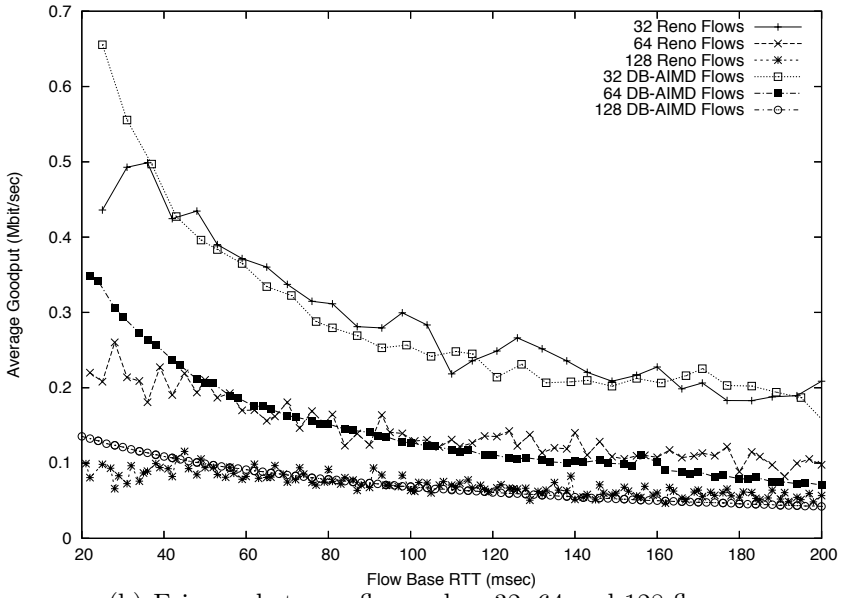


(c) Fairness between flows when 128 concurrent flows.

Figure 3.8: Goodput and fairness for Reno and delay-based AIMD algorithms. 10Mbps link, 100ms baseRTT.



(a) Goodput vs number of flows.



(b) Fairness between flows when 32, 64 and 128 flows.

Figure 3.9: Goodput and fairness for Reno and delay-based AIMD algorithms. 10Mbps link, 20-200ms baseRTT.

3.4.5 Delayed ACKing and TSO

The foregoing experimental results disable delayed acking and TSO in order to focus clearly on the fundamental performance of the delay-based algorithm. In this section we consider the impact of delayed acking and TSO in more detail. Although our primary focus in this chapter is on the impact of low correlation in congestion control, we found that our experimental tests also helped to throw some light on the issue of obtaining clean delay measurements. In particular, we found that some significant sources of "noise" in delay measurements can be removed by simple sender side changes to delay measurement within the network stack. These changes have since been incorporated into the Linux kernel and are discussed in detail in chapter 5.

With the changes to RTT measurement and TSO handling discussed in chapter 5, Figure 3.10(a) plots the measured queueing delay vs number of flows with delayed acking and TSO enabled. In these tests we also make a change to the delay-based AIMD algorithm to prevent delay backoff reducing the cwnd below two packets (loss backoff and RTO behaviour is unchanged). This is because with delayed acking enabled we need at least two packets in flight in order to obtain a clean delay measurement using the approach discussed above. It can be seen from Figure 3.10(a) that the performance is similar to that without delayed acking and TSO – compare with Figure 3.5(b). In particular, the queueing delay remains bounded even for large numbers of flows where the correlation between measured and actual queueing delay is low. The point at which the link enters the asymptotic regime is changed, however, from around 130 flows to around 90 flows owing to the lower bound on cwnd of two packets in Figure 3.10(a) while in Figure 3.5(b) the lower bound is one packet.

Also shown in Figure 3.10(b) is the corresponding measured link goodput.

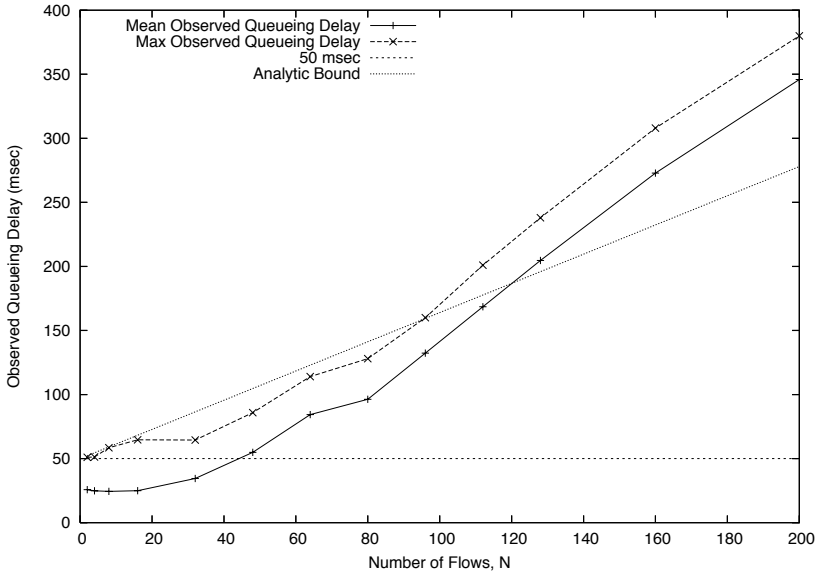
It can be seen that the goodput is almost identical to that without delayed acking and TSO. Although not shown here, throughput fairness between flows is also very similar to that without delayed acking and TSO.

3.5 Scope

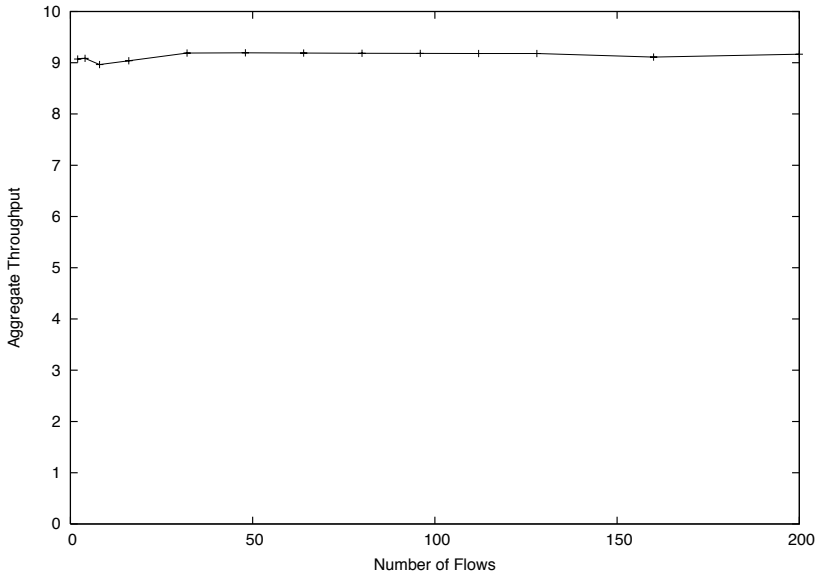
This work focusses on the specific question of whether low correlation between measured and actual queueing delay and loss is a fundamental obstacle to congestion control. This issue is of particular interest as it is a commonly voiced concern and has been the subject of a number of published studies. We emphasise that many other issues that are not addressed here remain to be considered before any decision could be made as to the suitability or otherwise of delay as a congestion signal for use other than on a purely experimental basis. For example, we do not consider the issue of co-existence between flows operating loss-based and delay-based congestion control, performance over multiple bottlenecks, performance over wireless links and so on. Nevertheless, we argue that the work here is an important first step in exploring the nature of fundamental constraints on congestion control.

3.6 Conclusions

In this chapter we revisit the commonly voiced concern that low correlation between measured delay and packet loss events means that delay may be fundamentally flawed as a signal for congestion control. This concern is particularly topical in view of a number of proposals to change the TCP congestion control algorithm to make use of delay information. Examples include not only FAST TCP [Jin et al., 2004] but also hybrid congestion



(a) Peak and mean queueing delay vs number of flows



(b) Goodput vs number of flows

Figure 3.10: Performance with delayed acking and TCP segmentation offload enabled. 10Mbps link. Flow baseRTTs uniformly random from 20-200ms.

control algorithms based on the use of both loss and delay, e.g. TCP Illinois [Liu et al., 2006] and Compound TCP [Tan et al., 2005]. The latter is now available in Windows Vista and is currently undergoing review at the IRTF and IETF standards bodies.

The main contribution of this work is to demonstrate that in fact what matters for congestion control is the *aggregate* behaviour of the flows sharing a link. Hence, perhaps somewhat surprisingly, while any given single flow may measure delay which is only weakly correlated with the actual queueing delay, this is not in itself an obstacle to congestion control. We demonstrate this constructively via detailed experimental tests and also confirm analytically the general nature of our conclusions. This potentially has direct implications for our understanding of the fundamental constraints on congestion control within the current Internet architecture.

Chapter 4

Estimation of Round-Trip Propogation Delay

4.1 Introduction

Estimation of round-trip propagation delay, also referred to as $baseRTT$, is a fundamental part of many congestion control algorithms. It is usually estimated using $minRTT$, the minimum observed round trip time for a flow. Apart from its evident importance in delay-based algorithms such as FAST TCP [Jin et al., 2004] and TCP Vegas [Brakmo et al., 1994], it also plays an important role in recently proposed loss-based (and hybrid) schemes such as TCP Westwood [Mascolo et al., 2001], Compound TCP [Song, 2006], and H-TCP [D.J.Leith and R.N.Shorten, 2004] in which flows aim to adaptively set their backoff factor to $\beta = baseRTT/RTT_{max}$, where RTT_{max} is related to the measured RTT at backoff. In this latter context, the ability to estimate $baseRTT$ effectively decouples the congestion control algorithm from the issue of queue provisioning and enables high utilisation to be achieved with small buffers [Shorten and Leith, 2006].

Accurate estimation of $baseRTT$ is, however, known to potentially be problematic. A primary issue is interactions between $baseRTT$ estimation and the congestion control algorithm itself. For example, in TCP Vegas and related algorithms a standing queue is induced as part of the correct operation of the congestion control algorithm. Thus, when flow start times are staggered, later flows tend to over-estimate $baseRTT$ due to the standing queue created by earlier flows. Similar issues can also arise with loss-based algorithms. For example, if the AIMD backoff factor used is $\beta = minRTT/RTT_{max}$ (as in H-TCP and some versions of Westwood), then over-estimation of $baseRTT$ may mean that flows do not empty network queues, allowing the overestimate to persist indefinitely. Statistical multiplexing of flow backoffs on links shared by many loss-based flows can also lead to later flows experiencing a standing queue and so overestimating $baseRTT$.

In this chapter we revisit the interaction between $baseRTT$ estimation and congestion control action. We develop a simple AIMD-based scheme that allows network buffers to drain and thus demonstrate in a constructive manner that, with proper design, it is indeed possible for flows traversing a bottleneck link to estimate their base RTT reliably.

The full experimental testbed setup is described in appendix A

4.2 Background

As mentioned in 2.3.2, the delay-based AIMD algorithm has the following backoff factor, β

$$\beta(t) = \delta * \frac{minRTT}{RTT(t)} \tag{4.1}$$

with $minRTT$, an estimation of $baseRTT$ as before. As mentioned in chapter 2, the factor $\delta < 1$ (eg 0.8) is multiplied to aid in corect measurement of

$baseRTT$, as described in Leith et al. [2007]. If the measured $minRTT$ were initially an overestimate of $baseRTT$, the queueing delay would be underestimated, the ratio would therefore be too large to empty the queue and the error in $minRTT$ would be maintained. The additional δ factor causes a back off by slightly more than necessary to empty the "apparent" queueing delay. Each successive back off will empty further, lowering the estimate until $minRTT = baseRTT$.

The impact of this choice of β is the primary focus of the present work. While we often illustrate results with reference to the delay-based AIMD algorithm, all our analysis extends to general AIMD algorithms including loss-based algorithms. To make this explicit, we therefore also include examples illustrating loss-based AIMD operation. Our main result is that with the choice of backoff factor (4.1) only very mild conditions are needed for the bottlenecked buffer to drain and for the true value of $baseRTT$, to be available to network flows regardless of initial estimation errors. This fact is shown both analytically and experimentally.

4.3 Draining network buffers

To help gain some insight into the mechanics of the backoff algorithm, consider for the moment a network with a single flow. Let B denote the link bandwidth in packets/s, $baseRTT$ the round-trip propagation delay. Consider the k'th backoff event and let $w(k)$ denote the congestion window of the flow at backoff and Q_k the network buffer occupancy. At backoff, we have that $RTT(k) = baseRTT + Q_k/B$ and $w(k) = B * RTT(k)$. Following

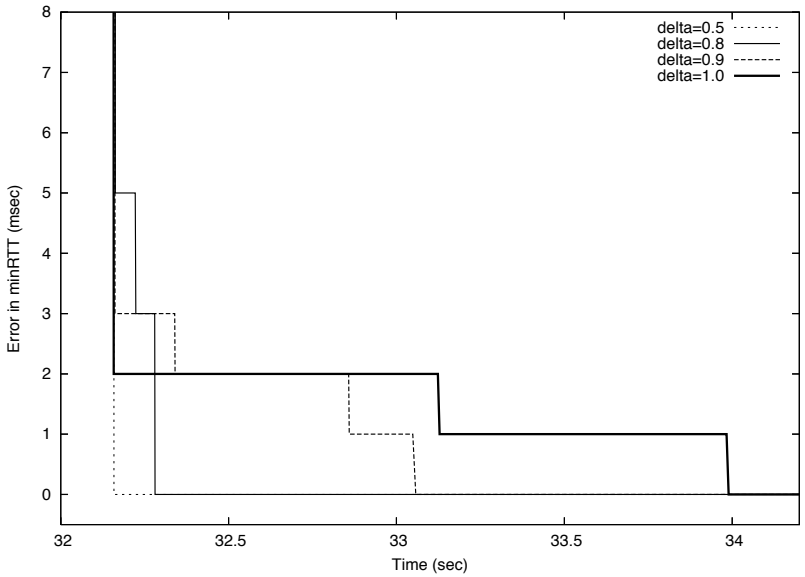
backoff, the flow cwnd is $\beta(k)w(k)$. Selecting $\beta(k)$ according to (4.1),

$$\beta(k)w(k) = \delta\frac{minRTT(k)}{RTT(k)}B * RTT(k) = \delta * B * minRTT(k)$$

If $minRTT(k) = baseRTT$, then since $\delta < 1$ it can be seen that cwnd falls *below* the link bandwidth-delay product $B * baseRTT$. Thus the queue empties thereby providing an opportunity for the flow to observe the propagation delay $baseRTT$. If $minRTT(k) > baseRTT$ then the queue need not empty after backoff. The buffer occupancy after backoff is $q_k = \beta(k)w(k) - B * baseRTT = B * (\delta * minRTT(k) - baseRTT)$ and the round-trip delay is $baseRTT + q_k/B = \delta * minRTT(k)$. Since $\delta < 1$, the round-trip delay is *lower* than the previous lowest observed delay $minRTT(k)$. Hence, the flow can update $minRTT$ to a value that is closer to the true propagation delay $baseRTT$. In effect, we are using the multiplicative decrease action to probe the network to discover whether an RTT *below* our current best estimate $minRTT$ is possible. After a number of congestion events (the number being dependent on the size of the initial error in $minRTT$ and on the value of δ), we can see the flow is eventually guaranteed to obtain an accurate estimate of the propagation delay $baseRTT$. This is illustrated in Figure 4.1, which shows experimental measurements of $minRTT$ converging to $baseRTT$.

4.3.1 Detailed Analysis

Consider n flows sharing a bottleneck link. Let $w_i(k)$ denote the cwnd of flow i at the k'th backoff event, let $baseRTT_i$ be the round-trip propagation delay of flow i. Let Q_k be the queueing delay at the k'th congestion event. Note that this need not be the maximum buffer size when delay-based AIMD



(a) Delay-based AIMD

Figure 4.1: Experimental measurements of estimation error $minRTT - T$ vs time. Measurements are shown for a range of values of the design parameter δ. Initial estimate of base RTT is hard-wired to an incorrect value to illustrate convergence. 10Mbps link, baseRTT 200ms, one delay-based AIMD TCP flow.

is used. When delay-based congestion control it also need not be the same at every congestion event (due to burstiness etc). At congestion we have that the aggregate flow rate equals the link rate, i.e.

$$\sum_{i=1}^{n} \frac{w_i(k)}{baseRTT_i + Q_k} = B \tag{4.2}$$

Following backoff, the aggregate rate becomes $\sum_{i=1}^{n} \beta_i(k) \frac{w_i(k)}{baseRTT_i + q_k}$, where q_k is the queueing delay after backoff ($q_k < Q_k$) and $\beta_i(k)$ is the backoff factor of flow i.

If the queue empties on backoff, then $q_k = 0$ and flows have the opportunity to measure their base round-trip time $baseRTT_i$. If the queue does not empty on backoff, then the aggregate flow rate continues to equal the link rate, i.e.

$$\sum_{i=1}^{n} \beta_i(k) \frac{w_i(k)}{baseRTT_i + q_k} = B \tag{4.3}$$

Assume that flow backoffs are synchronised i.e. every flow backs off at each congestion event (this assumption is relaxed later). Also assume for the moment that each flow observed the RTT at the $k-1$'th backoff when the queueing delay was q_{k-1} (again, we relax this assumption later). The flow backoff factors then satisfy

$$\beta_i(k) \leq \delta \frac{baseRTT_i + q_{k-1}}{baseRTT_i + Q_k} \quad \forall i \in 1, .., n$$

Substituting into (4.3),

$$B = \sum_{i=1}^{n} \beta_i(k) \frac{w_i(k)}{baseRTT_i + q_k} \leq \sum_{i=1}^{n} \delta \frac{baseRTT_i + q_{k-1}}{baseRTT_i + q_k} \frac{w_i(k)}{baseRTT_i + Q_k} \tag{4.4}$$

Using (4.2), it then follows that $\exists i$ such that

$$\delta \frac{baseRTT_i + q_{k-1}}{baseRTT_i + q_k} \geq 1$$

i.e.

$$q_k \leq \delta q_{k-1} - (1 - \delta) * baseRTT_i$$

Thus, provided $\delta < 1$ the queue occupancy at backoff q_k decreases monotonically until eventually the queue empties, providing an opportunity for flows to measure their base round-trip time. This is illustrated for example in Figure 4.2 which shows the error in $baseRTT$ for 16 reno flows on a 10Mbps link. Further illustration is in figure 4.3 which shows the $baseRTT$ error histories for a variety of different BDP links all heavily loaded with DB-AIMD flows.

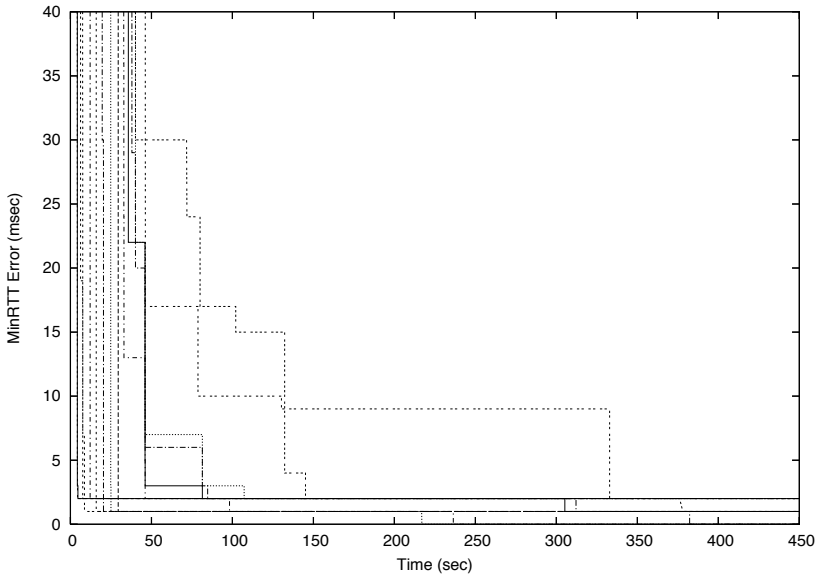
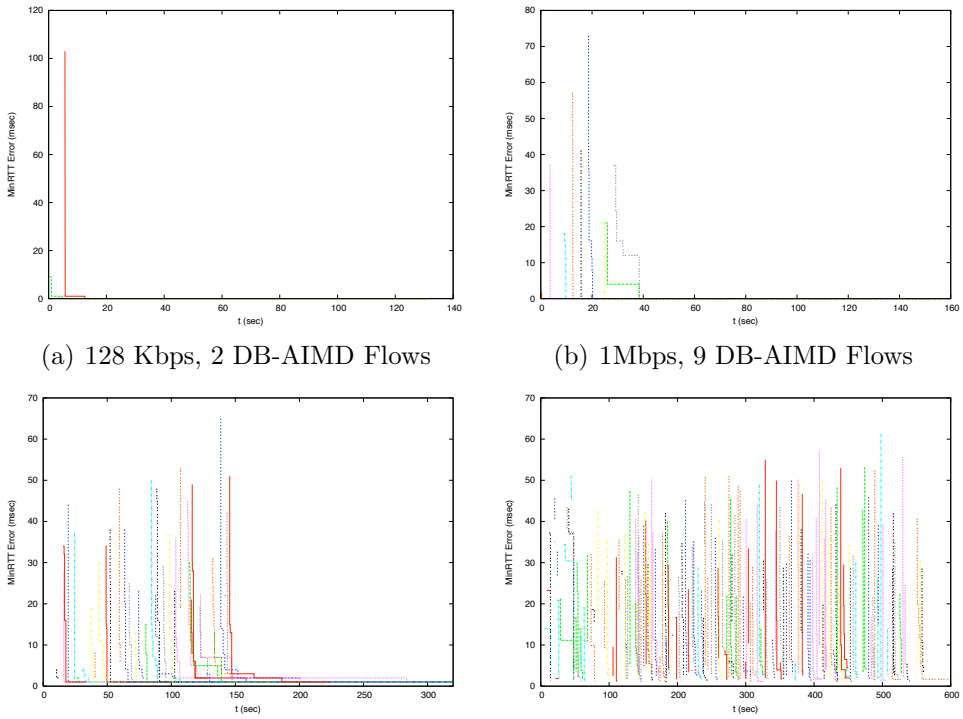


Figure 4.2: Experimental measurements illustrating queue draining with multiple flows. 10Mbps link, 125KB buffer, mix of flow baseRTTs 20-200ms, $\delta = 0.8$, 16 TCP Reno flows with adaptive backoff and randomised start times. Initial base RTT estimates for all flows are hard-wired to incorrect values to confirm insensitivity of convergence to estimation errors.



(a) 128 Kbps, 2 DB-AIMD Flows

(b) 1Mbps, 9 DB-AIMD Flows

(c) 10Mbps, 48 DB-AIMD Flows

(d) 50Mbps, 192 DB-AIMD Flows

Figure 4.3: Further experimental measurements showing the queue draining with multiple flows. In each case there is 1xBDP of queueing on the router and the baseRTT is spread evenly within 20–200msec.

4.3.2 Discussion

Convergence Rate

The rate of decrease is evidently influenced by the choice of δ, decreasing δ increasing the rate at which the queue drains. This can be seen, for example, in Figure 4.1.

Unsynchronised Drops

We can capture unsynchronised backoffs by setting $\beta_i(k) = 1$ for flows which do not backoff at the k'th congestion event. The foregoing analysis can then be immediately extended to the case of unsynchronised flows under mild assumptions. Specifically, assume that at congestion events synchronised backoffs occur with probability lower bounded by $p_s > 0$. That is, it occasionally happens that all flows backoff together at a congestion event. This assumption can be relaxed in various ways but this is beyond the scope of the present work.

Observability

Our analysis assumes that each flow observes the RTT after the k'th backoff. It is easy to see that this assumption may, however, be further relaxed to the much weaker requirement that there is a non-zero probability p_i that over a congestion event flow i observes an RTT less than or equal to the RTT after the k'th backoff.

Quantisation of cwnd

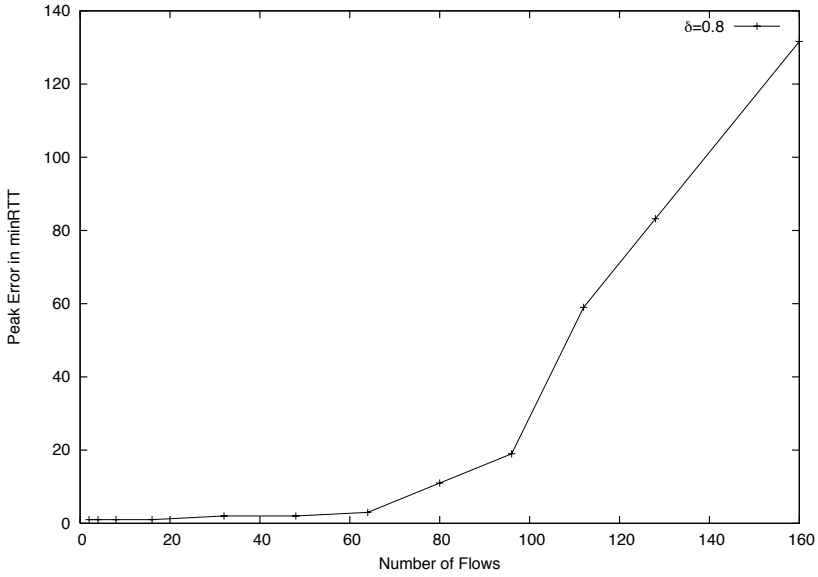
Our analysis assumes that the specified backoff factor (4.1) is successfully applied to the flow cwnd. A notable exception to this occurs when the flow

cwnd is only one packet in size. Since this is the lowest admissible cwnd, the backoff factor specified by (4.1) cannot be applied. This is illustrated, for example, in Figure 4.4(a) which plots the worst-case (over all flows) error in estimated baseRTT as the number of flows is increased. Also shown in Figure 4.4(b) is the distribution of flow cwnd values vs the number of flows. It can be seen that the worst case estimation error begins to rise as the number of flows increases above 60. This corresponds to a regime where around 60% of flows have a cwnd of only one packet and around 35% have cwnd of two packets. Above around 100 flows, $> 90\%$ of flows have a cwnd of one packet. Since flows can no longer backoff their cwnd, a standing queue develops at the link buffer and the estimation error of later flows inevitably increases. We note that this issue can potentially be resolved by introducing more fine-grained control of the flow send rate at low cwnd via, for example, pacing. Consideration of such extensions is, however, beyond the scope of the present work.

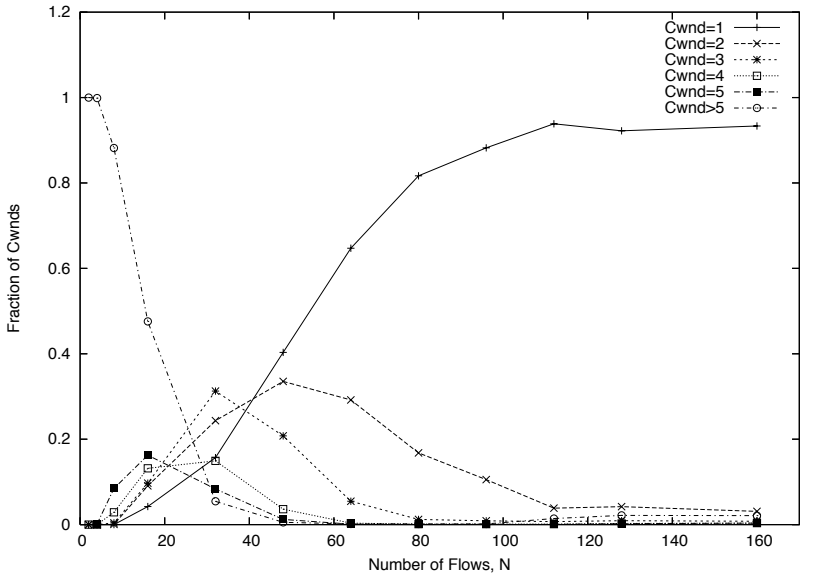
Fairness

When modifying the back off factor, it is important to consider the effect this will have on the relative throughput achieved by competing flows (the "fairness"). There is some debate as to what fairness is desirable (window, throughput, network resource, ...). This will not be entered into here. However, it is important to note where a modification changes the fairness behaviour.

With Reno, when a source i receives an ACK, $cwnd$ is incremented by $\frac{1}{cwnd}$, or by one packet per round trip time. When a packet loss is detected $cwnd$ is backed off by a factor $\beta = \frac{1}{2}$. When $w_i(k)$ is the congestion window size for flow i at congestion event k, we have that at the $k+1$th congestion



(a) Worst case estimation error



(b) Flow cwnd distribution

Figure 4.4: Illustrating quantitisation issues as the number of flows on a link is increased and flow cwnds tend towards one packet. 10Mbps link, mix of flow baseRTTs 20-200ms, $\delta = 0.8$, delay-based AIMD (similar results are obtained for Reno with adaptive backoff).

event

$$w_i(k+1) = \beta_i(k)w_i(k) + \alpha_i T(k) \tag{4.5}$$

where $T(k)$ is the time, in seconds between congestion events k and $k+1$ and α_i is the *cwnd* increase rate in pkt/sec. α_i is therefore roughly $\frac{1}{RTT_i}$, the inverse round trip time of that flow. If flow i detects the congestion event $\beta_i = \frac{1}{2}$ and otherwise $\beta_i = 1$.

If we represent the average Cwnd of flow i as $E[cwnd_i]$, and the packet drop ratio per flow as λ_i, the ratio of windows for two flows with differing round trip times [Leith and Shorten, 2006] is

$$\frac{E[w_i]}{E[w_j]} = \frac{\alpha_i/(1-E[\beta_i])}{\alpha_j/(1-E[\beta_j])} \approx \frac{1/\lambda_i RTT_i}{1/\lambda_j RTT_j} \tag{4.6}$$

Where congestion events are synchronised, $\lambda_i = 1$ for all flows, ie all flows experience all congestion events, regardless of their *cwnd*.

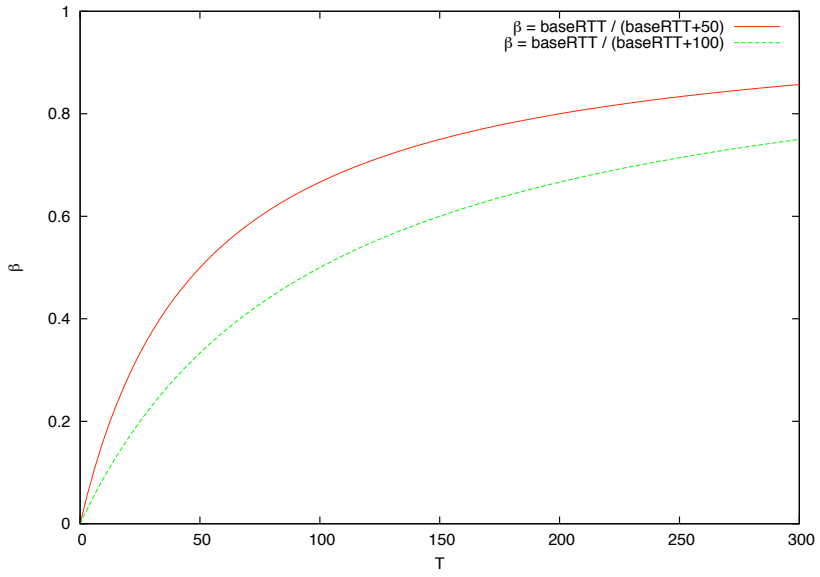


Figure 4.5: Variation of the back off factor, β with base round trip time, $baseRTT$ for two example propagation delays, 50msec and 100msec.

The scheme proposed here modifies the back off factor, β to be a function of the ratio of base round trip time and round trip time at congestion (eqn. 4.1). For a given queueing delay, this ratio increases with the base round trip time, see fig. 4.5. In the unsynchronised case, β and $E(cwnd_i)$ will therefore be greater for flows with larger round trip times.

It would therefore be necessary to also correspondingly modify the increase rate α_i if window fairness is to be maintained in unsynchronised environments.

$$\frac{E[w_i]}{E[w_j]} = 1 \tag{4.7}$$

$$\frac{\alpha_i/(1-E[\beta_i])}{\alpha_j/(1-E[\beta_j])} = 1 \tag{4.8}$$

One simple solution to which is

$$\alpha_i = n(1-\beta_i) \tag{4.9}$$

for some constant factor n, eg for compatibility with reno $n = 2$.

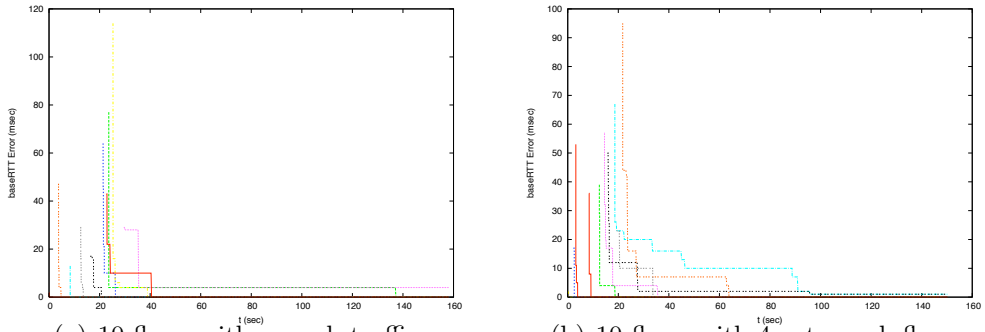
Transient Flows

In previous tests, we have shown flows successfully detecting $baseRTT$ where N such long-lived flows share a bottleneck. In most network environments however, long-lived flows are seldom seen in isolation and indeed are very often the exception. Much of the TCP traffic consists of short-lived flows (eg web, email, file-sharing traffic), lasting at most a few seconds.

Such short-lived flows may never exit slowstart so many aggregated together can be quite an aggressive use of bandwidth. On the other hand, as they come and go they force the long-lived flows to back off, then dis-

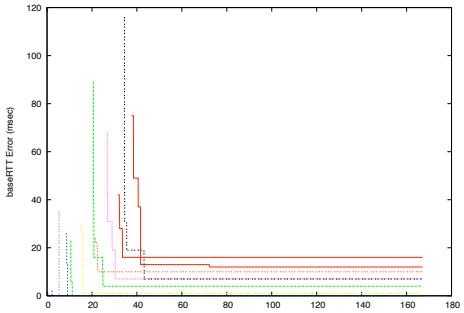
appear, which is akin to choosing $\beta = 0$ so they could potentially improve measurement of $baseRTT$ in some instances.

With this in mind, some measurements have been done on long-lived flows sharing a link with multiple http flows whose data size is Pareto distributed, as suggested in [Willinger et al., 1997]. Figures 4.6(a), 4.7(a) and 4.8(a) show the error in observed round trip time for a set of long-lived flows saturating a bottleneck link. Data from a similar experiment is shown in figures 4.6(b), 4.7(b) and 4.8(b) but with the addition of a number of extra web flows in each case. Despite the reduced bandwidth share per flow, it can be seen that flows with error in their $baseRTT$ estimate seem to correct themselves at least as often and sometimes more often in the presence of web flows. The 32-flow example with 40 web flows (fig. 4.8(b)) seems to finish with less error in $baseRTT$ than without web flows. This is perhaps due to the web flows forcing the long-lived flows to back off and then emptying the queue when they finish shortly afterward.

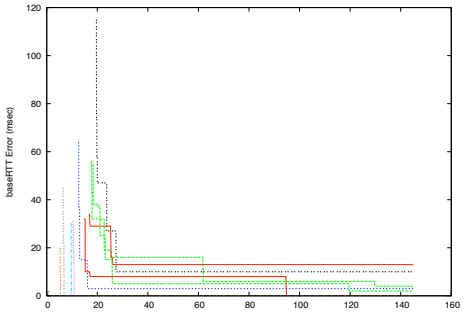


(a) 10 flows with no web traffic (b) 10 flows with 4 extra web flows

Figure 4.6: The effect of extra transient flows. BW=1Mbit/sec, N=10, $\delta = 0.8$, delay-based AIMD (similar results are obtained for Reno with adaptive backoff). Where the extra web flows are added, although one flow initially gets stuck with a higher error in $baseRTT$, it is clear that queue emptying events are more common and $baseRTT$ gets corrected.

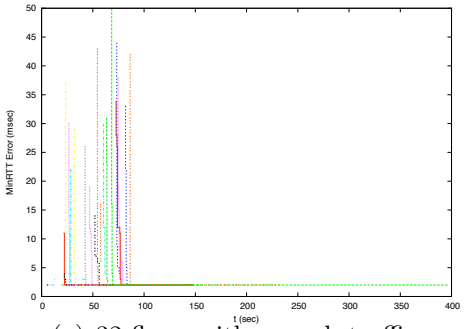


(a) 12 flows with no web traffic

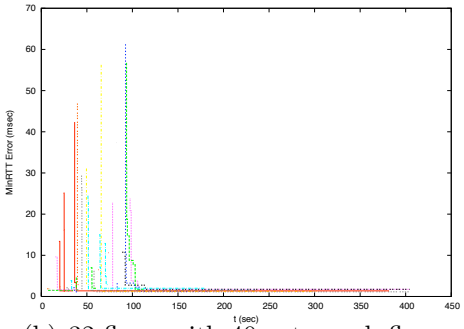


(b) 12 flows with 4 extra web flows

Figure 4.7: The effect of transient web flows. BW=1Mbit/sec, N=12, δ = 0.8, delay-based AIMD.



(a) 32 flows with no web traffic



(b) 32 flows with 40 extra web flows

Figure 4.8: The effect of transient web flows. BW=10Mbit/sec, N=32, δ = 0.8, delay-based AIMD.

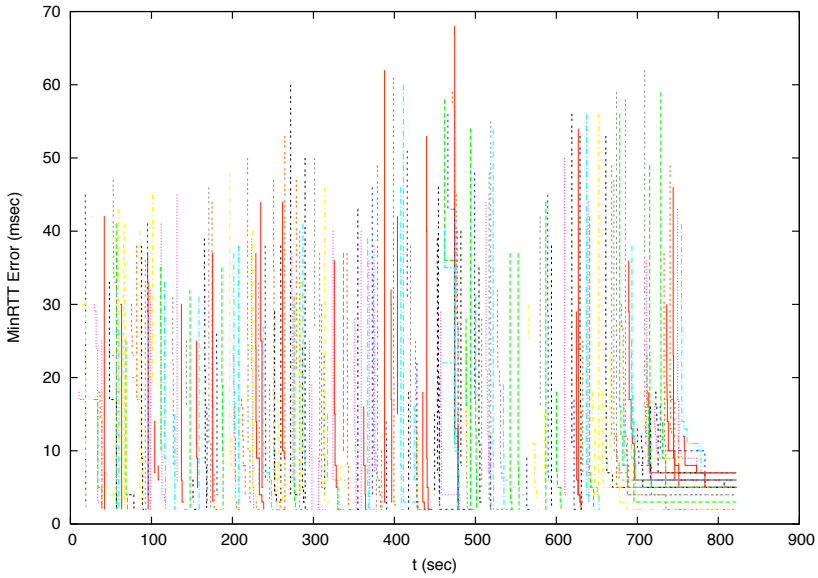


Figure 4.9: Illustration of the adaptive backoff breaking down. The first 200 or more flows quickly correct their estimate of $minRTT$, but subsequently some flows begin tp show persistent errors, though only of the order of 10msec. BW=50Mbit/sec, N=256, $\delta = 0.8$, delay-based AIMD. Points are omitted where the error becomes less than 2ms.

4.4 Conclusions

In this chapter we revisit the interaction between *baseRTT* estimation and congestion control action. We use the previously described delay-based AIMD scheme that allows network buffers to drain and thus demonstrate in a constructive manner that, with proper design, it is indeed possible for flows traversing a bottleneck link to estimate their base RTT reliably.

Chapter 5

Measurement of RTT for Congestion Control

While implementing delay-based AIMD, it became clear that obtaining a clean, reliable measure of network round trip times was not trivial and while all TCP stacks currently measure round trip time, this value was probably not well suited to use in congestion control. A number of hurdles have been overcome which we document here to save others from repeating this work. Most of the solutions have since been incorporated into the linux kernel's core TCP congestion avoidance code so all delay-based algorithms developed on linux will benefit. However, other TCP implementors may still benefit from this information.

5.1 Trusting Echoed Timestamps

Since the earliest days of TCP, hosts have measured the evolving round trip time of each flow in order that they could correctly determine the RTO (retransmit timeout). Following [Jacobson, 1988], modern TCP implemen-

tations maintain a smoothed round trip time ($sRTT$) which is updated with each raw sample (RTT) using a simple smoothing mechanism.

$$sRTT = \frac{1}{8}RTT + \frac{7}{8}sRTT \tag{5.1}$$

In order to reduce the work of the sender in timing each packet, RFC1323 introduced the TCP Timestamps option. This allows either end of the connection to request that two 32-bit timestamp fields be included in the tcp headers, one for each host. On sending a packet, each host places a timestamp in the header. The receiver then echoes this timestamp back to the sender in the acknowledgement. On receiving the ACK, the sender can then infer the round trip time from the echoed timestamp and the current time. The RFC states that the timestamp values "must be at least approximately proportional to real time, in order to measure actual RTT", but later states in the PAWS (protection against wrapped sequence numbers) that "values are monotone non-decreaasing in time".

By using echoed timestamps to calculate the RTT, the sender is trusting the receiver to honestly return these timestamps. There is probably little benefit for a receiver to alter the RTO so this seems safe. However, where the round trip time is used in congestion control to alter $Cwnd$, there is a clear benefit for a receiver who can alter the sender's estimate of the round trip time in order to increase his share of network bandwidth.

5.1.1 Loss-based Congestion Control

Some loss-based congestion control algorithms attempt to correct RTT unfairness by increasing $Cwnd$ more quickly for longer round trip times (e.g. Cubic, H-TCP). If they use timestamp-generated round trip times, they can

be fed an over-estimated RTT causing them to be more aggressive. This could potentially be used by a receiver either to increase share of bandwidth or perhaps as part of a distributed denial of service.

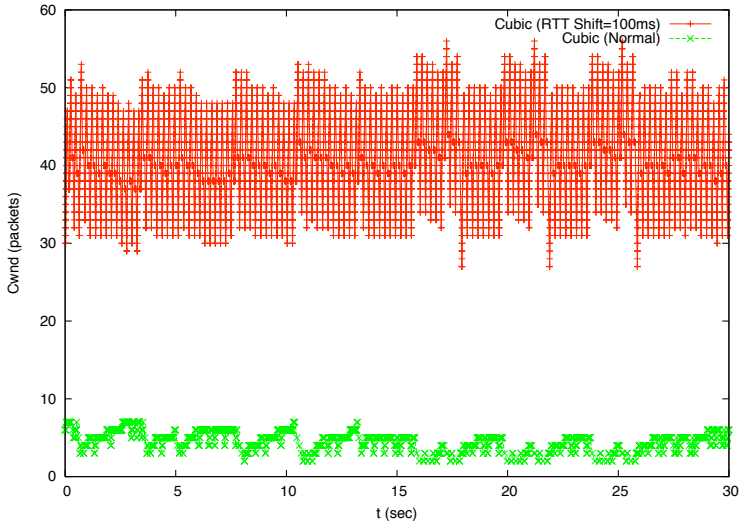
Prior to linux v2.6.22, Cubic made direct use of the echoed timestamps in order to calculate the round trip time which it used to scale Cwnd. We discovered that this allowed a dishonest receiver to gain substantially greater bandwidth over competitors, simply by subtracting a constant integer from the timestamp value it sent back in its acknowledgements, see Fig. 5.1.

5.1.2 Delay-based Congestion Control

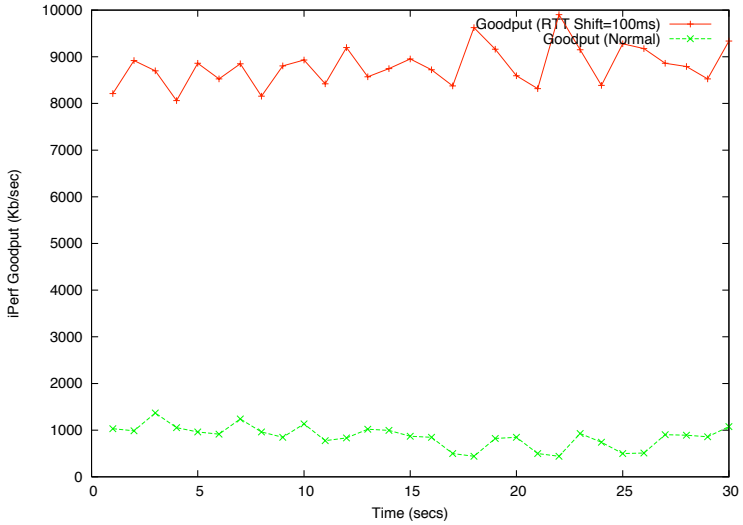
Both delay-based (e.g. Vegas, FAST) and hybrid (e.g. Compound, Illinois) congestion control algorithms measure the queueing delay for use as a signal of congestion and to some degree all back off when they measure an increasing delay. These potentially can have the increasing delay hidden from them. For example, a receiver can use a ping to measure the real queueing delay and then dynamically alter the sender's timestamp to reduce the apparent queueing delay. This can give them a greater bandwidth share over competing flows.

5.1.3 TCP-LP

A further case of timestamp trust is TCP-LP [Kuzmanovic and Knightly, 2003] which attempts to act as a bandwidth scavenger. By monitoring the queueing delay TCP-LP attempts to back off wherever there is queueing delay so as to only use bandwidth where there is spare capacity. In order to avoid backing off due to reverse-path congestion, TCP-LP attempts to estimate the one-way transmission time by monitoring the receiver's timestamps. Even if internal timing is used to calculate the round trip time, a cheating receiver



(a) Cwnd



(b) Bandwidth

Figure 5.1: Congestion window and resulting achieved bandwidth histories for two competing Cubic TCP flows. One of the receivers is subtracting 100msec from the timestamps it echoes back so the sender behaves as though the RTT is longer. Here we can see the effect of the different round trip times apparent to the sender (5msec, 105msec). Link bandwidth: 10Mbps, BaseRTT: 5msec, Queue Length: 50KB

could in principle craft its own timestamps to suggest that all of the queueing is on the reverse path. However, as a bandwidth scavenger, TCP-LP may still be useful where both sender and receiver timestamps can be trusted to behave predictably.

Following our report of these issues, the linux congestion avoidance code was modified in v2.6.22 [Hemminger, 2007] so as not to use timestamps for the calculation of RTT (though timestamps are still used for RTO). Cubic was also modified so as not to directly use the timestamps. Instead, either an internal timestamp is stored in each control block (`scb->when`) in the retransmit queue, or where necessary a microsecond-granular `ktime` is used where fine grain timestamps are desired (e.g. Vegas, Veno). The only exception is TCP-LP which by design must use the receiver's timestamps.

Linux has so far used internal timing to prevent a receiver from forging timestamps. An alternative approach would be to craft verifiable timestamps. One limitation is that timestamp values in the tcp headers are required to be monotone for each flow. Nevertheless, one or more of the rightmost bits of the timestamp could be reserved for use as a signature to verify the timestamp contained in the remaining bits. The signature could then be recalculated on return. Where the verification failed, the sender could choose a suitable recourse, e.g. backing off Cwnd or resetting the connection.

5.2 Delayed ACKing

Acknowledgements are cumulative in TCP. This means that a single ACK of byte n implies acknowledgement of all previous bytes in the session. It is therefore unnecessary to ackowledge every received packet. Based on this, an optimisation of TCP was proposed in RFC813 such that when data is

received an acknowledgement is not sent immediately. If a quick response is produced by the application (e.g. response to a keystroke in a telnet session) the ACK may be carried on the data response instead of on a separate ACK packet. If no quick response occurs, the next incoming packet will generate an ACK for both packets or if sufficient time passes the single ACK will eventually be sent. This optimisation is useful for reducing traffic, particularly in interactive sessions (e.g. telnet, ssh).

The optimisation, called "Delayed ACKing" was mandated for use in RFC1122 which stated:

> A TCP SHOULD implement a delayed ACK, but an ACK should not be excessively delayed; in particular the delay should not be more than 0.5 seconds, and in a stream of full-sized segments there SHOULD be an ACK for at least every second segment.

Almost all modern TCP stacks implement this. When RFC1323 introduced timestamps for RTO calculation it was decided that, in order that delayed ACKs would not cause timeouts, the echoed timestamp should be that of the earliest packet being ACKed:

> Many TCP's acknowledge only every Kth segment out of a group of segments arriving within a short time interval; this policy is known as "delayed ACKs". The data-sender TCP must measure the effective RTT, including the additional time due to delayed ACKs, or else it will retransmit unnecessarily. Thus, when delayed ACKs are in use, the receiver should reply with the TSval field from the easliest unacknowledged segment.

The result is that RTTs calculated from timestamps can considerably over-estimate the single-packet round trip time, particularly where Cwnd is

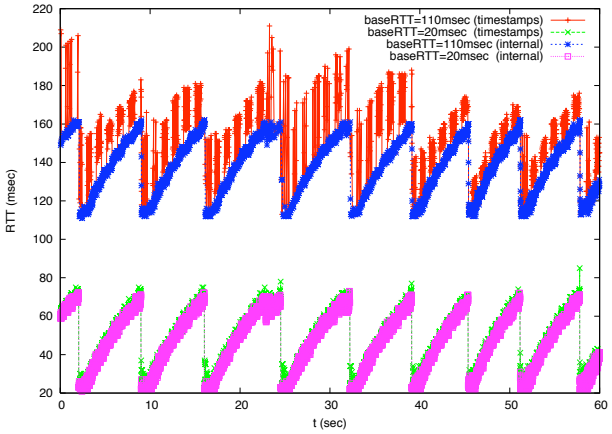
small, see fig. 5.2. This can incorrectly indicate congestion. Some RTT values will have an extra delay due to the time which passed between sending a pair of packets. Where there is a long time between two packets, a single packet may go unacknowledged until the maximum allowed time (the delayed ACK alarm) and be acknowledged singly. On FreeBSD, this time is tunable but 100msec by default. On Linux it varies between 40msec and 200msec.

As in section 5.1, one answer is to use internal timing. When each ACK arrives, if delayed ACKing is in operation, one can ignore ACKs which only acknowledge one packet (those triggered by the delayed ack alarm) and calculate RTTs based on the most recent of each pair of packets being acknowledged (i.e. the packet which directly triggers each ACK).

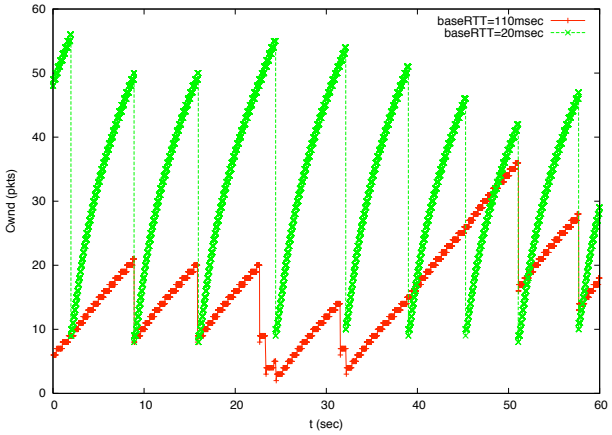
The initial internal RTT timing code in linux emulated the behaviour of the timestamps, using the timestamp of the oldest acknowledged packet as these RTTs were used for RTO where timestamps were unavailable. A fix has been accepted into Linux v2.6.24 separating the calculation of RTO and congestion avoidance RTTs as above [McCullagh, 2007], both for the microsecond and millisecond granular RTT code paths.

Figure 5.2(a) overlays rtt data calculated using the updated internal rtt calculation and that calculated from timestamps, alongside Cwnd for those flows in figure 5.2(b). Two flows with different round trip propagation delays (20msec, 110msec) compete on a 10Mbps link. The rtt data clearly shows the difference between the two delay signals and that the effect of delayed ACKing is more pronounced for the lower Cwnd flow.

Figure 5.3 shows overlaid timestamp and internal RTTs calculated using the microsecond-granular internal code. Figure 5.4 shows the same picture with the fixed microsecond timer logic. With the unfixed stack (fig. 5.3), delayed ack “noise” in the congestion signal causes Cwnd to back off con-



(a) RTT



(b) Cwnd

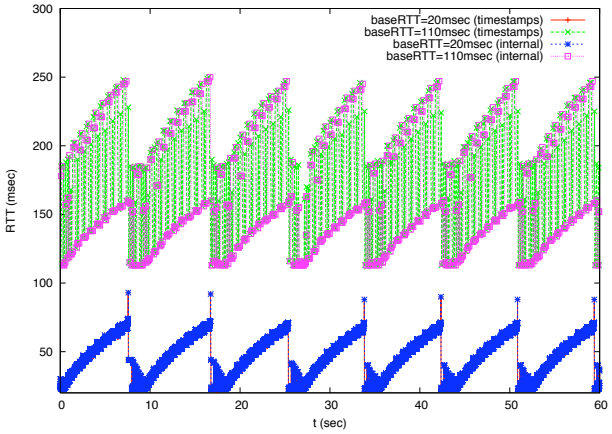
Figure 5.2: RTT and Cwnd histories for two competing flows with *baseRTT* of 110 and 20 msec. RTT histories are overlaid from both the timestamps (100msec: red, 20msec: green) and the internally calculated congestion control RTT values (100msec: blue, 20msec: pink). As Cwnd is smaller for the 110 msec flow, the interpacket time is greater so delayed ACKing has a much more noticeable effect. In this example the delayed acking noise is of the order of 50msec, the threshold delay for DB-AIMD, so it can cause a back off without any queueing at all. Bottleneck bandwidth: 10Mbps, *baseRTT*: 20,110msec, Queue Length: 5MB. Delay-based AIMD congestion control.

stantly. Cwnd becomes so small for the 100msec flow that the delayed ACK alarm (which seems to be about 90msec) is used regularly whereas in the fixed graph (fig. 5.4) the delayed ACK "noise" is filtered from the congestion signal so both flows have functional Cwnd and consequently the delayed ACK "noise" seen in the timestamp data reduces considerably.

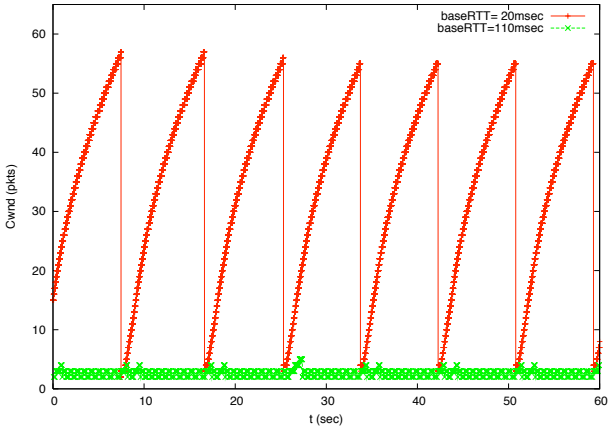
5.3 TCP Segmentation Offload

TCP Segmentation Offload (TSO) is an optimisation for TCP which is provided by some recent (usually GigE or faster) network cards. One task of the TCP stack is to break up (segment) data into MTU sized segments. At high speeds, this can be quite costly in cpu cycles. TSO allows the software TCP stack to offload this task to the network card which does it in hardware, freeing up cpu cycles for other work. TCP takes a large chunk of data (e.g. 64KB) to be sent and passes that to the network card along with template TCP, IP and data-link headers. The network card then breaks the chunk into segments (commonly 46 segments of 1448 Bytes) adds headers to each and transmits the packets.

TSO can affect round trip time calculations. Firstly, the TCP timestamp is usually fixed in the template which gets passed to the network card. Therefore, regardless of when the packets get sent, they will all have the same timestamp. It follows that the later the packet is in the chunk, the larger its timestamp-calculated RTT will be. If one uses internal timing, at least in the linux implementation, the control block which contains the internal timestamp corresponds to a single large unsegmented chunk, not to an individual packet, so any subsequent acknowledged packets after the first will have an extra "TSO delay" added. However, to discard these other packets

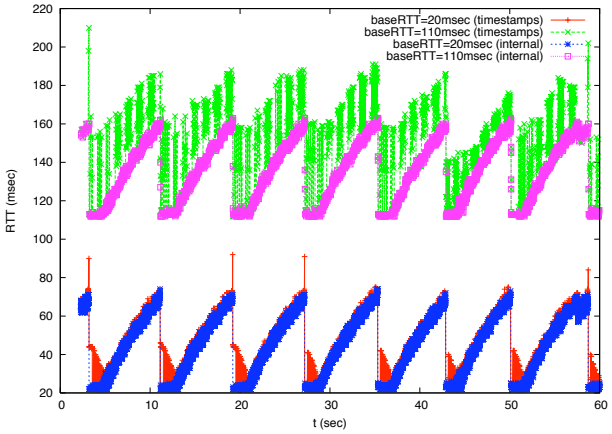


(a) RTT (unfixed)

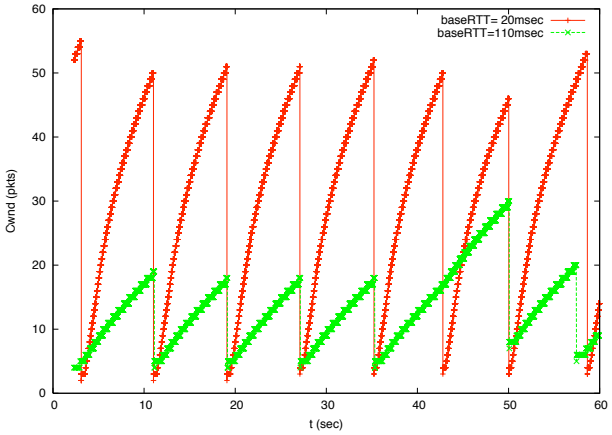


(b) Cwnd (unfixed)

Figure 5.3: RTT and resultant Cwnd histories for two competing flows calculated using timestamps (red, green) and the microsecond-granular internal method (blue, pink). The graphs show the data before the fix incorporated into linux v2.6.24. It can be seen that the microsecond-granular internal timing emulates delayed ack noise. The effect on DB-AIMD is severe, causing repeated back offs. The delayed ack noise is worse for smaller Cwnds (up to 90msec here), reinforcing the problem. It can be seen that there is delayed ack "noise" on the larger Cwnd flow, immediately after each back off. Bottleneck bandwidth: 10Mbps, $baseRTT$: 20,110msec, Queue Length: 5MB. Delay-based AIMD congestion control.



(a) RTT (fixed)



(b) Cwnd (fixed)

Figure 5.4: RTT and resultant Cwnd histories for two competing flows calculated using timestamps (red, green) and the microsecond-granular internal method (blue, pink). The two graphs show the data with the fix incorporated into linux v2.6.24-rc5. The microsecond-granular timestamps no longer emulate delayed ack noise but the timestamps still show it. For the smaller-Cwnd flow, the magnitude of the delayed ACK noise is of the order of 40-50msec, compared with 90msec in fig. 5.3. Bottleneck bandwidth: 10Mbps, $baseRTT$: 20,110msec, Queue Length: 5MB. Delay-based AIMD congestion control.

would reduce the number of available RTT samples substantially.

How TSO is implemented within the TCP stack is very important in minimising this "TSO delay". DB-AIMD was first implemented on v2.6.17.7 of the linux kernel. It was quickly discovered that TSO caused a severe "smearing out" of the RTT values, see figure 5.5(a). Closer inspection (figure 5.5(b)) showed that TSO was causing regular oscillation in the RTT values of the order of 50msec as each subsequent packet in a TSO chunk got a greater "TSO delay".

The reason for this "smearing" was that the linux kernel was attempting to use TSO too aggressively. In order to make full use of off-loading, linux would defer sending until a chunk of 40–45 packets was available. This would then be passed down to the network card. The transmission time for each of these chunks at 1Gbps is around 0.5msec so each leaves the sender almost simultaneously. However, at 10Mbps the transmission time is around 50msec. The result of this behaviour is that while the sender defers sending, the queue drains, though perhaps not down to zero. When the chunk gets sent, the first packet goes to the front of the queue and has a fairly short queueing delay. Each subsequent packet will queue behind the others in the chunk. The last packet will have the same queueing delay as the first, plus the queueing time of the entire chunk (50msec). The result is the steep linear spikes of 50msec seen in 5.5(b). This is not due to inaccurate measurement of the RTT, but rather due to TSO causing bursts which make the real RTT fluctuate.

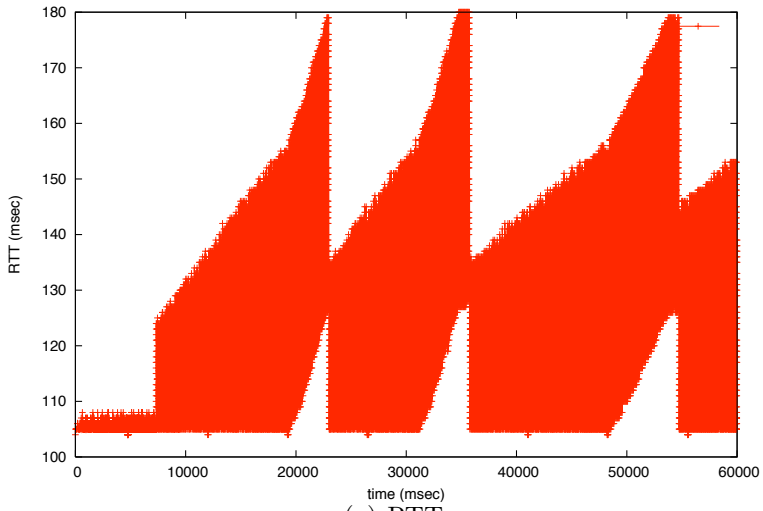
The effect of this can be to both overestimate or underestimate the congestion in the network. Figure 5.7 shows the RTT of two independent, similar Reno flows with TSO — one with a recent linux kernel (v2.6.24-rc6), one before this issue was fixed (v2.6.18.1). In both cases, there is only one flow on the link. There are two clearly different cases:

- Where Cwnd is small, the time TSO defers is long enough to empty the queue. Had the first packets in the chunk been sent earlier, some would have been through the router so the later packets would have seen less of a queue and therefore have a lower RTT. Instead, they must wait the full 50msec. In these regions, the 2.6.18 kernel sees higher RTTs than the 2.6.24 kernel, see figure 5.7(c).

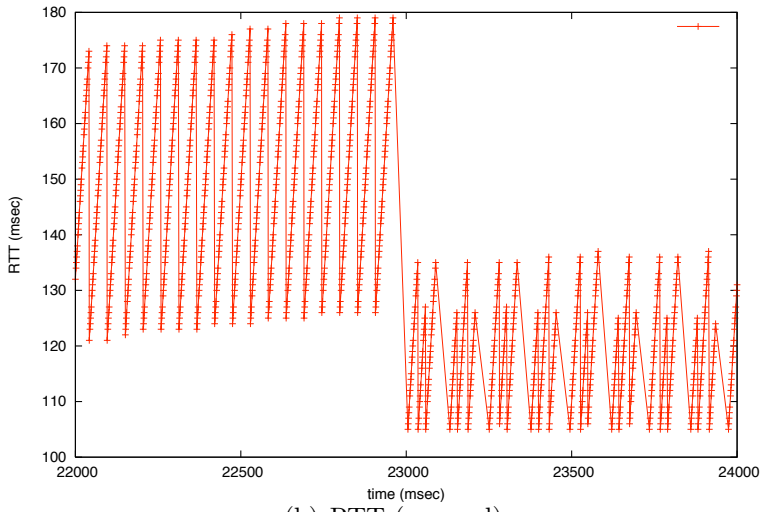
- For larger Cwnds, TSO does not defer sufficiently long to empty the queue. Therefore, had the first packet in the chunk been sent earlier, it would still be in the queue for the later packets and their RTT is unaffected. However, the queue has decreased in size during the deferral, so the RTT experienced by the first packet in the chunk is actually shortened compared to sending the packets evenly spaced. In these regions, the 2.6.18 kernel sees lower RTTs than the 2.6.24 kernel, as can be seen in figure 5.7(c).

On our discovery of this issue, a patch was written for the linux kernel [Heffner, 2006] to restrict the amount of time the kernel would defer the sending of data before passing a TSO chunk down to the network card (by default to 5msec). The effect of this initial patch was to restrict the size of each TSO chunk to as much data as has been ACKed in the past 5 msec (otherwise Cwnd would be exceeded). This patch reduced the noise considerably, though not quite to the level it was without TSO, see figure 5.6. The TSO code has changed substantially since then and the issue seems no longer to be a problem, see figure 5.7.

It was also discovered in v2.6.24-rc3 that the updated microsecond-granular, internal timer in the linux kernel was treating TSO in a special way, somewhat analogous to the situation with delayed acking. When a portion of a TSO chunk was acknowledged, linux would not calculate an RTT based on

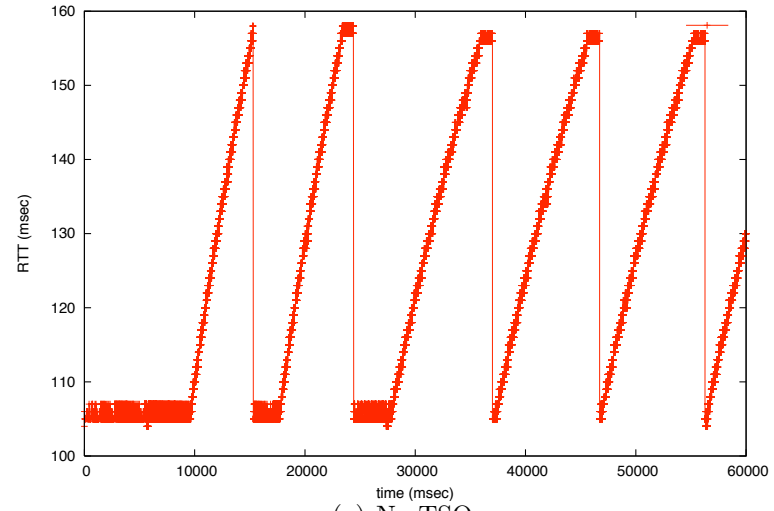


(a) RTT

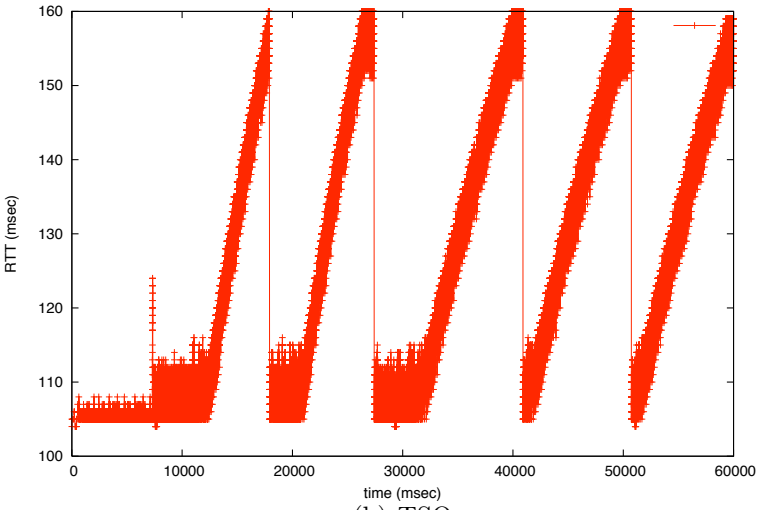


(b) RTT (zoomed)

Figure 5.5: An example RTT time history for DB-AIMD with TSO enabled on linux v2.6.17.7. The overall queue probing trend can be seen, but it is clear in the zoomed picture that the queueing trend is coupled to a second shorter time-scale oscillation caused by extra delays due to TCP Segmentation Offload. Bottleneck bandwidth: 10Mbps, $baseRTT$: 102msec, Queue Length: 512KB, single delay-based AIMD TCP flow. Delayed ACKing is disabled on the receiver for simplicity.

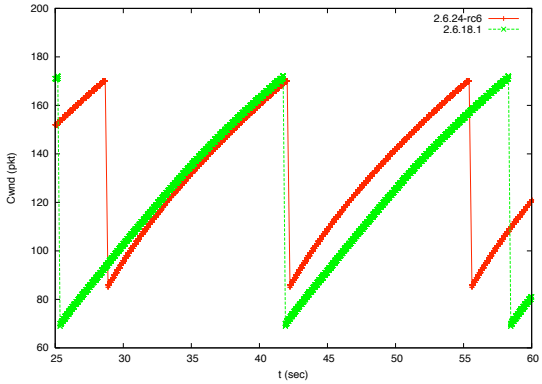


(a) No TSO

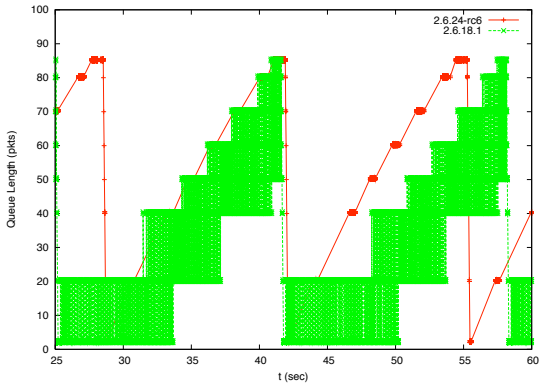


(b) TSO

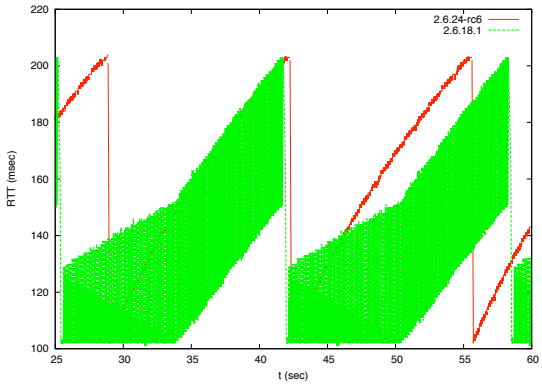
Figure 5.6: RTT time history for DB-AIMD with TSO disabled (left) and enabled (right) on linux v2.6.17.7 with John Heffner's TSO patch. It can be seen that the patch substantially reduces "TSO delay" by setting a maximum time which the kernel can defer sending data due to TSO. Bottleneck bandwidth: 10Mbps, $baseRTT$: 102msec, Queue Length: 512KB, single delay-based AIMD TCP flow. Delayed ACKing is disabled on the receiver for simplicity.



(a) Cwnd



(b) Queue Occupancy



(c) RTT

Figure 5.7: Example time histories of measured RTT with TSO enabled in Linux 2.6.18.1 and 2.6.24-rc6 kernels using Reno congestion control. Delayed ACKing disabled. Bottleneck bandwidth: 10Mbps, $baseRTT$: 100msec, Queue Length: 1xBDP, single Reno TCP flow.

each acknowledgement but instead wait until the final acknowledgement corresponding to that chunk even though the stored timestamp corresponded more accurately to the first packet in the chunk. This yields the maximum available RTT including the full TSO delay for that chunk[1].

As with delayed acking, TSO delay is not a helpful signal of congestion so this was patched in v2.6.24 [McCullagh, 2007]. To prevent problems like those in figure 5.5, the RTT based on the first acknowledgement would give the cleanest RTT signal. However, ignoring all subsequent acknowledgements in a chunk could severely hinder the system from detecting and responding to congestion. It seems also that the smearing problems are best addressed with the implementation of TSO. For these reasons an RTT is now calculated for every acknowledgement in a TSO chunk, rather than just the first.

[1]Curiously, this accidentally filtered some of the delayed ACKing noise — where a delayed ACK corresponded to the end of a chunk, the more recent timestamp would be preferred over the older one.

Chapter 6

Conclusions

The objective of this thesis is to investigate the possibilities and limitations of delay-based congestion control in general and to carefully examine criticisms which have been made in the literature. In so doing, we have

1. Presented experimental evidence to show that, while the correlation between network congestion and individual flows' estimate of the round trip time may be very weak, this is not a fundamental barrier to congestion control. We have shown congestion control being achieved in the practical examples of low correlation. While every flow may not detect each congestion event, what is important for congestion control is the aggregate detection and reaction of all flows sharing the link.

2. Studied in greater detail the behaviour of the delay-based AIMD congestion control algorithm. In particular, we have shown its ability to constrain the queueing delay with multiple concurrent flows and shown how the queueing delay scales as the number of flows increases to its practical limit — where the flows can no longer back off.

3. Experimentally examined the practical utility of adaptive back-off meth-

ods used in H-TCP to ensure that the minimum round trip propagation delay is measurable by all flows.

4. Examined the practical problems in accurately measuring network queueing delay, via the round trip time. In particular, we have looked at the problems associated with echoed timestamps, delayed acknowledgments and TCP segmentation offload and have offered some solutions which can be employed on the sender to minimise their effects. Most of these solutions are now in use in the standard Linux kernel.

While much of this work sounds some positive notes for delay-based congestion control, there are still many issues which must be addressed before it could be considered for use in real networks. Among other remaining concerns, further work is needed to

- examine the effects of other causes of increases in round trip time, such as wireless MAC delay
- design a delay-based scheme which can coexist with loss-based flows
- design a delay-based scheme which will fully utilise a link in the presence of queueing delay on the reverse path

before delay-based congestion control can truly be practical. Considerable further work lies ahead.

Appendix A

Experimental setup

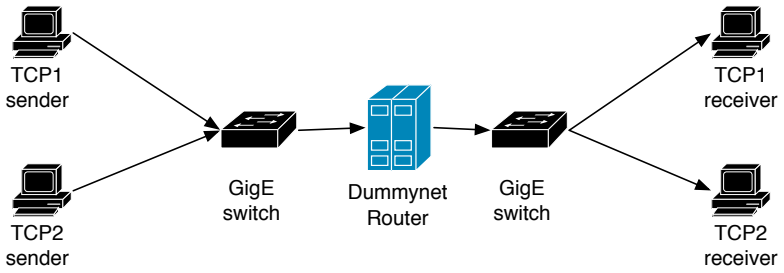


Figure A.1: The dumbbell network topology used in tests.

Experiments were carried out using a testbed consisting of commodity PCs connected to gigabit switches to form the branches of a dumbbell topology. All sender and receiver machines used in the tests have identical hardware and software configurations as shown in Table I and are connected to the switches at 1Gb/sec. The router, running FreeBSD v4 with the dummynet module, can be configured with various bottleneck queue-sizes, capacities and round trip propagation delays to emulate a range of network conditions.

We have implemented the delay-based AIMD algorithm in Linux 2.6. The kernel version used (unless otherwise stated) is 2.6.23. For delay-based AIMD, the minimum observed RTT measurement RTT_{min} is used as an

	Description
CPU	Intel Xeon CPU 2.80GHz
Memory	512 Mbytes
Motherboard	Dell PowerEdge 860
Kernel	Linux 2.6.23
txqueuelen	1,000
max_backlog	300
NIC	Intel 82540EM
NIC Driver	e1000

Table A.1: Hardware and Software Configuration.

estimate of propagation delay and queueing delay is then estimated as $RTT - RTT_{min}$.

TCP Flows are injected into the testbed using iperf. TCP stacks are instrumented using a modified version of the Linux tcpprobe module. Unless otherwise stated, the queueing delay threshold used is $\tau_0 = 50ms$.

Bibliography

Ahn, J. S., Danzig, P. B., Liu, Z., and Yan, L. (1995). Evaluation of TCP Vegas: emulation and experiment. *ACM Computer Communications Review*, pages 185–195.

Biaz, S. and Vaidya, N. (2003). Is the round-trip time correlated with the number of packets in flight? *Proceedings of the 3rd ACM SIGCOMM conference on Internet measurement*, pages 273–278.

Brakmo, L. S., O'Malley, S. W., and Peterson, L. L. (1994). TCP Vegas: New techniques for congestion detection and avoidance. In *Proceedings of SIGCOMM*, pages 24–35.

D.J.Leith and R.N.Shorten (2004). H-TCP protocol for high-speed long-distance networks. In *Proc. 2nd Workshop on Protocols for Fast Long Distance Networks. Argonne, Canada, 2004.*

Floyd, S. (2003). Highspeed TCP for large congestion windows. IETF RFC 3649, Experimental, Dec 2003.

Heffner, J. (2006). Bound tso defer time (resend). Linux netdev mailing list `http://www.mail-archive.com/netdev@vger.kernel.org/msg24484.html`.

Hemminger, S. (2007). Tcp congestion control rtt patches. Linux netdev mailing list `http://www.mail-archive.com/netdev@vger.kernel.org/msg42876.html`.

Jacobson, V. (1988). Congestion Avoidance and Control. *ACM Computer Communication Review; Proceedings of the Sigcomm'88 Symposium in Stanford, CA, August, 1988*, 18:314–329.

Jain, R. (1989). A delay-based approach for congestion avoidance in interconnected heterogeneous computer networks. *ACM Computer Communication Review*, 19(5):56–71.

Jin, C., Wei, D. X., and Low, S. H. (2004). FAST TCP: Motivation, architecture, algorithms, performance. In *IEEE INFOCOM 2004*.

Kelly, T. (2003). Scalable TCP: improving performance in highspeed wide area networks. *ACM SIGCOMM Computer Communication Review*, 33(2):83–91.

Kuzmanovic, A. and Knightly, E. (2003). TCP-LP: a distributed algorithm for low priority data transfer. *INFOCOM 2003. Twenty-Second Annual Joint Conference of the IEEE Computer and Communications Societies. IEEE*, 3.

Leith, D., Heffner, J., Shorten, R., and McCullagh, G. (2007). Delay-based aimd congestion control. In *Proc. Workshop on Protocols for Fast Long Distance Networks, Los Angeles.*

Leith, D. and Shorten, R. (2006). On rtt scaling in h-tcp. `http://www.hamilton.ie/net/rtt.pdf`.

Liu, S., Başar, T., and Srikant, R. (2006). TCP-Illinois: a loss and delay-based congestion control algorithm for high-speed networks. *Proceedings of the 1st international conference on Performance evaluation methodolgies and tools.*

Martin, J., Nilsson, A., and Rhee, I. (2003). Delay-based congestion avoidance for TCP. *IEEE/ACM Transactions on Networking*, 11(3):356–369.

Mascolo, S., Casetti, C., Gerla, M., Sanadidi, M., and Wang, R. (2001). TCP westwood: Bandwidth estimation for enhanced transport over wireless links. *Proceedings of the 7th annual international conference on Mobile computing and networking*, pages 287–297.

McCullagh, G. (2006). Fix integer overflow in h-tcp congestion control. Linux netdev mailing list `http://www.mail-archive.com/netdev@vger.kernel.org/msg25076.html`.

McCullagh, G. (2007). Tcp: use non-delayed ack for congestion control rtt. Linux netdev mailing list `http://www.mail-archive.com/netdev@vger.kernel.org/msg57778.html`.

Prasad, R. S., Jain, M., and Dovrolis, C. (2004). On the effectiveness of delay-based congestion avoidance. In *Second International Workshop on Protocols for Fast Long-Distance Networks.*

Rewaskar, S., Kaur, J., and Smith, D. (2005). Why dont delay-based congestion estimators work in the real-world. Technical report, Technical Report TR06-001, Department of Computer Science, UNC Chapel Hill, July 2005.

Saltzer, J., Reed, D., and Clark, D. (1984). End-To-End Arguments in System Design. *Technology*, 100:0661.

Shorten, R. and Leith, D. (2006). On queue provisioning, network efficiency and the delay-bandwidth product. IEEE Transactions on Networking, to appear.

Shorten, R., Leith, D., Foy, J., and Kilduff, R. (2004). Analysis and design of congestion control in synchronised communication networks. Automatica.

Shorten, R., Leith, D., and Wirth, F. (2006). Products of random matrices and the internet: Asymptotic results. *IEEE Transactions on Networking, 14(6), pp. 616-629.*

Song, K. (2006). Compound TCP: A Scalable and TCP-Friendly Congestion Control for High-speed Networks. *Proceedings of PFLDnet 2006.*

Stevens, W. (1997). RFC2001: TCP Slow Start, Congestion Avoidance, Fast Retransmit, and Fast Recovery Algorithms. *Internet RFCs.*

Tan, K., Song, J., Zhang, Q., and Sridharan, M. (2005). A compound TCP approach for high-speed and long distance networks. In *International Workshop on Protocols for Fast Long-Distance Networks.*

Wang, Z. and Crowcroft, J. (1991). A new congestion control scheme: Slow Start and Search (Tri-S). *ACM Computer Communication Review*, 21(1):32–43.

Willinger, W., Taqqu, M., Sherman, R., and Wilson, D. (1997). Self-similarity through high-variability: Statistical analysis of ethernet lan traffic at the source level. *IEEE/ACM Trans Networking*, 5.

Xu, L. and Rhee, I. (2005). CUBIC: A new TCP-Friendly high-speed TCP variant. In *Proc. Workshop on Protocols for Fast Long Distance Networks, 2005.*