Chapter 2

Related Work

In this chapter, I will describe previous work related to my thesis. First, I will briefly discuss computer clock synchronization, including the Global Positioning System. Then, I will give an overview of several recent TCP congestion control algorithms. I will also discuss several congestion control algorithms that are not tied to TCP. In addition to end system congestion control mechanisms, I will describe variants of Random Early Detection, an active queue management algorithm that assists end systems in congestion control. Finally, I will look at network measurement and analysis research that pertains to detecting network congestion.

2.1 Synchronized Clocks

Mills defines synchronizing frequency as adjusting clocks to run at the same frequency and synchronizing time as setting clocks to have the same time relative to Coordinated Universal Time (UTC), the international time standard [Mil90]. Synchronizing clocks is defined as synchronizing both frequency and time. In a computer network, there are two popular methods of synchronizing clocks: the Network Time Protocol (NTP) and the Global Positioning System (GPS). NTP is a network protocol that can run on computers to synchronize clocks to a specific computer that serves as a time source. The GPS satellite network allows computers that have GPS receiver hardware to synchronize their clocks to the GPS satellites.

2.1.1 Network Time Protocol

The Network Time Protocol (NTP) is a distributed clock synchronization protocol currently used to provide accurate time to computers around the world [Mil92]. Primary NTP servers are synchronized to standard reference clocks via GPS, modem, or radio. Secondary NTP servers and clients synchronize to the primary servers. When a client requests a time update, the server responds with its current time. The client notes when the message was received and uses one-half of the round-trip time as an estimate of the offset between the

server and client. NTP is accurate to about one millisecond on a LAN and a small number of tens of milliseconds on a WAN.

2.1.2 Global Positioning System

The Global Positioning System (GPS) can provide accurate timing information as well as position measurements to receivers on Earth [DP90]. GPS satellites are on a fixed orbit around the earth, so that at all times, the position of any satellite is known. For a stationary GPS receiver on Earth, information from four satellites is needed to determine both position and time. The GPS satellites transmit signals to Earth, and a GPS receiver locks on to these signals and decodes the position of the satellites. A special code is transmitted every millisecond that helps the receivers keep accurate time. The time computed from GPS can fall to within one microsecond of UTC.

Currently, GPS is limited by the fact that receivers need line of sight to receive signals from the satellites. For computer clock synchronization via GPS, there are methods, such as NTP, for propagating the GPS information to many machines that do not have line of sight to the satellites or do not have GPS receivers. Using NTP as a propagation method, though, reduces the accuracy of clock synchronization to that of NTP. It is reasonable to expect that clocks of computers on a WAN could be synchronized to within 1-2 ms if there were GPS time sources that could be used as NTP servers located on managed networks with low delay variability.

2.2 TCP Timestamp Option

RFC 1323 describes extensions to TCP to improve performance on large bandwidth-delay product network paths [JBB92]. The document introduces two new TCP options: window scale and timestamp. Both of these options are additions to the TCP header. Without the window scale option, the maximum value of the receiver's advertised window (and thus, the maximum value of the send window) in a TCP connection is 65 KB (2^{16}). Using the window scale option allows the receiver window to grow to up to 1 GB (2^{30}). The timestamp option can be used to protect against wrapped sequence numbers in a single connection and to compute more accurate RTTs. The timestamp option adds 10 bytes to the TCP header and contains the time the segment was sent and, if the segment is an ACK, the timestamp contained in the last data segment received. This TCP option is the basis for the Sync-TCP timestamp option.

2.3 TCP Congestion Control

TCP's congestion control has evolved through a process of iterative refinement. Current standards include TCP Tahoe, TCP Reno, TCP NewReno, and Selective Acknowledgments. In the following sections, I will describe developments of TCP congestion control algorithms and mechanisms past the standards-track improvements discussed in Chapter 1. TCP Tahoe provides the basis for the TCP variants that I discuss. The TCP modifications are grouped here according to changes they make to TCP Tahoe:

- **Variations on Slow Start:** TCP Vegas, TCP Santa Cruz, TCP Peach.
- **Variations on Congestion Avoidance:** TCP Vegas, TCP Santa Cruz, TCP Peach, AIMD Modifications.
- **Loss Detection:** TCP Vegas, Forward Acknowledgments, TCP Santa Cruz.
- **Data Recovery:** Forward Acknowledgments, TCP Santa Cruz, TCP Westwood, TCP Peach.

2.3.1 TCP Vegas

To address the sometimes large amounts of segment losses and retransmissions in TCP Reno, Brakmo *et al.* proposed several modifications to its congestion control algorithm [BOP94]. The resulting protocol, TCP Vegas, includes three main modifications: a fine-grained RTT timer, congestion avoidance using expected throughput, and congestion control during TCP's slow start phase. With these modifications, Brakmo reported a 40-70% increase in TCP Vegas' throughput over TCP Reno. As a result, TCP Vegas has been studied and debated extensively in the networking community [Jac94, ADLY95, MLAW99, HBG00, LPW01].

Congestion Avoidance

TCP Vegas uses a decrease in throughput as an early indication of network congestion. This is done by keeping track of the expected throughput, which corresponds to the amount of data in transit. As the congestion window increases, the expected throughput should increase accordingly. The basis of TCP Vegas' congestion avoidance algorithm is to keep just the right amount of extra data in the network. Vegas defines "extra data" as the data that would not have been sent if the flow's rate exactly matched the bandwidth available in the network. This extra data will be queued at the bottleneck link. If there is too much extra data, the connection could become a cause of congestion. If there is not enough extra data in the network, the connection does not receive feedback quickly enough to adjust to congestion. Feedback is in the form of RTT estimates which only return if data continues to be sent and acknowledged. By monitoring changes in throughput and reacting to conditions other than segment loss, TCP Vegas pro-actively handles network congestion. Unfortunately, study of

this portion of TCP Vegas revealed that it contributed the least to the authors' claims of higher throughput than TCP Reno [HBG00].

Slow Start

TCP Vegas added its congestion detection algorithm to slow start. During slow start, the expected throughput and actual throughput are monitored as in congestion avoidance. When the actual throughput is less than the expected throughput by a certain threshold amount, TCP Vegas transitions from slow start to congestion avoidance. According to one study, the congestion detection in slow start accounted for a 25% increase in throughput over TCP Reno, making it the most beneficial of all of TCP Vegas' modifications [HBG00].

Data Recovery

TCP Reno sends retransmissions after receiving three duplicate ACKs or after the RTO timer has expired. TCP Vegas tries to improve upon this by using a fine-grained measurement of the RTT as the RTO and by recording the time each segment is sent. Upon receiving a duplicate ACK, if the time since sending the potentially lost segment is greater than the fine-grained RTO, TCP Vegas retransmits the assumed lost segment. This modification met with criticism from prominent members of the networking community. In a note to the end2end-tf mailing list, Van Jacobson argued that the RTO should never be set to a segment's RTT because of normal fluctuations in RTT due to competing flows [Jac94]. In particular, Jacobson said that the RTO should never be set less than 100-200 ms.

TCP Vegas has not yet been accepted as a standard part of TCP. Recent work [LPW01], though, has focused on reinterpreting TCP Vegas and casting it as a complement to Random Early Marking [LL99], an active queue management technique.

2.3.2 Forward Acknowledgments

The forward acknowledgment congestion control algorithm (FACK) was developed as a method of using the information in the SACK option for congestion control [MM96]. The term "forward acknowledgment" comes from using information about the forward-most data known to the receiver, which is the highest sequence number the receiver has seen. The idea behind FACK is to use the information in the SACK option to keep an accurate count of the amount of unacknowledged data in the network. This allows the sender to more intelligently recover from segment losses. FACK incorporates congestion control into fast recovery but makes no changes to TCP Reno's congestion control algorithms outside of fast recovery.

Data Recovery

FACK adds two state variables: $snd.fack$, the largest sequence number held by the receiver (also called the "forward-most" sequence number) and $retran\_data$, the amount of unacknowledged retransmissions in the network. FACK uses information from the SACK option returned in each ACK to update $snd.fack$. The amount of unacknowledged data is represented by $awnd$, which is $snd.nxt - snd.fack + retran\_data$, where $snd.nxt$ is the sequence number of the next new data segment to be sent. During recovery, while $awnd < cwnd$, data can be sent. This regulates how fast lost segments can be retransmitted and forces the recovery mechanism to follow TCP Reno's congestion window mechanism instead of rapidly sending a series of retransmissions of lost segments.

Loss Detection

FACK also changes the trigger for entering fast recovery. In TCP Reno, fast recovery is entered whenever three duplicate ACKs are received. FACK uses $snd.fack$ to calculate how much data the receiver has buffered in its reassembly queue (data waiting for a hole to be filled before delivery). If the sender determines that the receiver's reassembly queue is larger than three segments, it enters fast recovery. The receipt of three duplicate ACKs still triggers an entry into fast recovery. This modification to the fast recovery trigger is useful if several segments are lost prior to receiving three duplicate ACKs. If only one segment is lost, then this method would trigger fast recovery at the same time as TCP Reno.

2.3.3 TCP Santa Cruz

TCP Santa Cruz offers changes to TCP Reno's congestion avoidance and error recovery mechanisms [PGLA99]. The congestion avoidance algorithm in TCP Santa Cruz uses changes in delay in addition to segment loss to detect congestion. Modifications to TCP Reno's error recovery mechanisms utilize a SACK-like ACK window to more efficiently retransmit lost segments. TCP Santa Cruz also includes changes to the RTT estimate and changes to the retransmission policy of waiting for three duplicate ACKs before retransmitting.

Congestion Avoidance

TCP Santa Cruz attempts to decouple the reaction to congestion on the data path from congestion detected on the ACK path. This is done through the use of a new TCP option, which includes the time the segment generating the ACK was received and its sequence number. TCP Santa Cruz uses *relative delay*, or change in forward delay, to detect congestion. The relative delay for two segments is calculated by subtracting the difference in the receipt times of the segments from the difference in the sending times of the segments. These relative delays are summed from the beginning of the connection and updated every RTT. If the

sum of relative delays is 0, then the amount of data queued in the network is steady over the interval. The relative delay is then translated into an estimate of the number of packets queued at the bottleneck using a packet-pair-like method (as described in section 2.4.2).

The goal of the algorithm is to keep a certain number of packets, N_{op}, queued at the bottleneck. If the current number of packets queued, n_t, is within δ of N_{op}, the current window is unchanged. If $n_t < N_{op} - \delta$, the window is increased linearly. If $n_t > N_{op} + \delta$, the window is decreased linearly. This approach is much like the window adjustment in TCP Vegas.

In their experiments, the authors varied N_{op} between 1-5 packets. They found that using between 3-5 packets gave high utilization and lower delay than TCP Reno. These experiments were performed with a single source and sink over a single bottleneck. In a more complex environment, the best value for N_{op} would have to be determined experimentally.

Slow Start

TCP Santa Cruz makes a small change to TCP Reno's slow start algorithm. The congestion window is initially set to two segments. This allows for the calculation of relative delay during slow start. TCP Santa Cruz can force the transition from slow start to congestion avoidance before $cwnd > ssthresh$ whenever a relative delay measurement or n_t is greater than $N_{op}/2$. The idea is to end slow start once any build-up in the queue is detected.

Data Recovery

TCP Santa Cruz includes in each ACK the sequence number of the segment generating the ACK and that segment's retransmission number (*e.g.*, if the segment was the second retransmission of a segment, its retransmission number would be 2). With this change, the RTT estimate (using the same algorithm as TCP Reno) can be calculated for every ACK including those that acknowledge retransmissions. This also removes the need for an exponential backoff in setting the RTO for successive retransmissions of the same segment because the RTT of the individual retransmissions can now be tracked. Additionally, the RTT estimate can be updated for each ACK received instead of once per RTT as in TCP Reno.

An *ACK Window* field is also included in every ACK. The ACK Window can indicate the status (received or not received) of each segment in a window. Approaches like SACK are limited to indicating only 3-4 missing segments. The ACK Window is represented by a bit-vector, where each bit represents a certain number of bytes. The number of bytes represented by each bit is determined by the receiver and indicated in the first byte of the ACK Window option. With a maximum of 19 bytes for the option, the ACK Window indicates the status of a 64 KB window with each bit representing 450 bytes. Normally, each bit represents the

number of bytes in the network's maximum segment size. If there are no gaps in the sequence space, the ACK Window is not included.

Loss Detection

TCP Santa Cruz also makes changes to TCP Reno's retransmission policy. A sender does not have to wait for three duplicate ACKs. If a segment is lost (as noted in the ACK Window), it can be retransmitted when the first ACK for a later segment is received after a RTT has passed since sending the lost segment. This mechanism is also much like the loss detection used in TCP Vegas.

2.3.4 TCP Westwood

TCP Westwood consists of a change to fast recovery that improves the performance of flows that experience link error losses [MCG+01]. This is useful on physical links where losses due to link errors are common, such as on a wireless link. To assist in this "faster recovery," the bandwidth used by the flow is estimated each time an ACK is returned.

The key insight to bandwidth estimation is that as soon as segments are dropped, the subsequent return rate of ACKs represents the amount of bandwidth available to the connection. TCP Westwood uses a low-pass filter on the bandwidth estimates to ensure that the most recent information is being used without reacting to noise. This low-pass filter needs to be given samples at a uniform rate. Since ACKs arrive at a non-uniform rate, the filtering algorithm in TCP Westwood assumes that virtual ACKs arrive at regular intervals. These virtual ACKs are counted as acknowledging 0 bytes of data, and so, do not affect the bandwidth estimate.

Data Recovery

TCP Westwood's changes TCP Reno's data recovery mechanisms allow a flow that experiences a wireless link error to recover from the segment loss without reacting as if network congestion caused the loss. Instead of backing off by 50% after the receipt of three duplicate ACKs, a TCP Westwood sender sets *cwnd* and *ssthresh* to appropriate values given the estimated bandwidth consumed by the connection at the time of congestion. In particular, at the beginning of fast recovery, *ssthresh* is set to the bandwidth-delay product (BDP), where the bandwidth is the current estimate (as described above) and the delay is the minimum RTT observed. If $cwnd > ssthresh$, then *cwnd* is set to *ssthresh*, otherwise *cwnd* is set as in TCP Reno (*i.e.*, *cwnd* is reduced by 50%). If a loss is detected via the expiration of the RTO timer, *ssthresh* is set to the BDP and *cwnd* is set to 1. slow start will be entered, and the flow will quickly increase its congestion window to *ssthresh* if the bandwidth is available.

2.3.5 TCP Peach

TCP Peach is a set of modifications to TCP Reno to improve communication over satellite networks in a QoS-enabled Internet [AMP01]. The two main changes are the replacement of slow start with "sudden start" and the replacement of fast recovery with "rapid recovery." TCP Peach also includes a small change to congestion avoidance to guarantee TCP-friendliness.

TCP Peach uses the transfer of dummy segments to probe the network for additional bandwidth. These dummy segments have a low-priority bit set in the type-of-service (TOS) field in the IP header. QoS-enabled routers would process the TOS bits and know that in times of congestion they should drop dummy segments first. When a sender receives ACKs for dummy segments, it knows that there is more bandwidth available in the network.

Slow Start

In sudden start, *cwnd* is initialized to 1. Every $RTT/rwnd$ seconds, a dummy segment is sent, where *rwnd* is the receiver's advertised window. After sending $rwnd - 1$ dummy segments, congestion avoidance is entered, so slow start lasts for only one RTT. For each dummy segment that is acknowledged (which would occur during congestion avoidance), *cwnd* is incremented.

Congestion Avoidance

The modification that TCP Peach makes to congestion avoidance controls when to open the congestion window. The variable *wdsn* is initialized to 0 and is used to make sure that during congestion periods, TCP Peach exhibits the same behavior as TCP Reno. When an ACK for a dummy segment is received and $wdsn = 0$, *cwnd* is incremented. If $wdsn \neq 0$, *wdsn* is decremented and *cwnd* is unchanged. No further changes are made to congestion avoidance, and *cwnd* is increased as in TCP Reno.

Data Recovery

Rapid recovery, which replaces fast recovery, lasts for one RTT and is triggered by the receipt of three duplicate ACKs, as in TCP Reno. The congestion window is halved, just like in fast recovery, so $cwnd = cwnd_0/2$, where $cwnd_0$ is the congestion window when loss was detected. The source then sends *cwnd* segments along with $cwnd_0$ dummy segments. The first *cwnd* dummy segments sent during rapid recovery that are acknowledged represent the amount of bandwidth that was available before the loss occurred. The congestion window should only be incremented for each dummy segment acknowledged after the first *cwnd* dummy segments have been acknowledged. To avoid increasing the congestion window for the first *cwnd* dummy ACKs received, *wdsn* is initially set to *cwnd*. The rapid recovery

phase, like fast recovery, ends when an ACK is received for the retransmitted segment. If the retransmitted segment is not dropped, its ACK will return before ACKs for any dummy segments.

If the segment drop was due to congestion, then either none or very few of the second *cwnd* dummy segments will be acknowledged. In this way, the congestion window has been halved and congestion avoidance is entered, just like in TCP Reno. If the segment drop is due to a link error, then most of the second *cwnd* dummy segments will be acknowledged and the congestion window will quickly grow back to $cwnd_0$, the level it was before the segment was dropped. TCP Peach does not explicitly distinguish between link errors and congestion drops, but it allows flows that suffer link errors to recover quickly.

2.3.6 AIMD Modifications

Several modifications to the parameters of TCP's AIMD congestion window adjustment during congestion avoidance have been proposed, including Constant Rate and Increase-By-K. These changes were made to address TCP's unfairness to flows with long RTTs [FP92]. This unfairness stems from the fact that congestion window increases are based on the rate of returning ACKs. Flows with long RTTs will see fewer increases of *cwnd* than a flow with a shorter RTT in the same amount of time.

Congestion Avoidance

The Constant Rate algorithm was introduced by Floyd [FP92, Flo91a]. Constant Rate changes the additive increase of TCP's congestion window to

$$w(t+1) = w(t) + (c * rtt * rtt)/w(t),$$

where $w(t)$ is the size of the congestion window at time t, c is a constant controlling the rate of increase, and rtt is the average RTT in seconds. This change is equivalent to allowing each flow increase *cwnd* each second by c segments.

Henderson *et al.* modified the Constant Rate algorithm to address the problem of very bursty transmission if each ACK increased the congestion window by several segments [HSMK98]:

$$w(t+1) = w(t) + min((c * rtt * rtt)/w(t), 1),$$

where $w(t)$, t, c, and rtt are as before.

An appropriate value of c is not easy to determine. The value c can be thought of as regulating the aggressiveness of the flow to be the same as a flow with a RTT of rtt. If $c = 25$, a RTT of 200 ms makes the numerator equal to 1. So, the flows with $c = 25$ would be on average as aggressive as standard TCP Reno flows with a base RTT of 200 ms. Henderson

recommends a value of c less than 100. Additionally, Henderson found that if only some flows used the Constant Rate (or Modified Constant Rate) algorithm while other flows used TCP Reno, the benefit of Constant Rate was diminished.

Henderson *et al.* also proposed the Increase-By-K algorithm as an improvement to Constant Rate [HSMK98]. This algorithm allows flows with long base RTTs to improve their performance without requiring all flows to make the change. Whenever the congestion detection mechanism signals that the network is not congested, *cwnd* is increased in the following manner:

$$w(t+1) = w(t) + min((K/w(t)), 1).$$

Suggested values of K are 2-4. In TCP Reno, for example, $K = 1$. As with the Modified Constant Rate algorithm, the minimum term is added to ensure that the congestion window does not grow faster than during TCP Reno slow start.

2.4 Non-TCP Congestion Control

Many network applications, such as streaming media and networked games, do not require reliability and use UDP rather than TCP for data transfer. These applications may employ either no congestion control strategy or a congestion control strategy that is far different than TCP. Flows that do not respond to network congestion are called *unresponsive* and could harm the performance of TCP flows. Developers of applications that use these non-TCP flows are encouraged to include congestion control mechanisms, especially those that are *TCP-friendly*, meaning that they receive the same throughput as a TCP flow would have given the same network characteristics (RTT and drop probability).

Bansal *et al.* classify non-TCP congestion control algorithms based on their steady-state and transient state behaviors [BBFS01]. In steady state, an algorithm can be either *TCP-equivalent*, *TCP-compatible*, or *not TCP-compatible*. In a transient state, an algorithm can be *TCP-equivalent*, *slowly-responsive*, or *responding faster than TCP*. TCP-equivalent algorithms use a transmission window that grows according to an AIMD algorithm with the same parameters as TCP[1]. A congestion control algorithm is TCP-compatible if it obtains close to the same throughput as TCP in steady-state. TCP-compatible algorithms see the same performance as TCP given the same loss rate and RTT. TCP-equivalent algorithms are also considered TCP-compatible. Slowly-responsive algorithms reduce their sending rate less than TCP in reaction to a single segment loss or congestion notification (*i.e.*, these also transmit faster than TCP). An AIMD algorithm with different parameters than TCP is an example of a slowly-responsive algorithm. Slowly-responsive algorithms are not guaranteed to be TCP-compatible.

[1]From here on, TCP refers to TCP Reno.

2.4.1 Congestion Control Principles

In RFC 2914, Sally Floyd explains why congestion control is needed and what is required for correct congestion control [Flo00a]. Non-TCP protocols, such as streaming media protocols, which are unresponsive to network congestion, have the potential to be unfair to TCP flows. If flows are unresponsive to network congestion, they will continue to send packets at a rate that is unsustainable by the network. This could cause many packets to be dropped, including those from TCP flows that would then reduce their sending rates. Flows that are unresponsive may not cause congestion collapse, but they could cause starvation among TCP flows. TCP is designed such that flows with similar RTTs should see similar performance. Non-TCP flows are encouraged to be TCP-compatible in order to be fair to TCP flows.

2.4.2 Control-Theoretic Approach

Keshav describes a control-theoretic approach to flow control [Kes91]. This approach requires that a sender can observe changes in the state of the network. This can be achieved if routers use a round-robin queuing and scheduling discipline for packets of different flows and a "packet-pair" probing technique. A round-robin queuing mechanism reserves a separate FIFO queue for each flow and services one packet from each queue in a round-robin manner. Under round-robin queuing, the service rate of a single flow will depend upon the number of other flows on the link rather than the incoming rates of other flows on the link, as with FIFO queuing. Packet-pair probing is a method of determining the bottleneck bandwidth of a path. A source sends out two packets back-to-back. If the queuing scheme is round-robin, it is guaranteed that the bottleneck rate is $1/\alpha$, where α is the duration between the receipt of the packets at the destination. When the sender then receives an ACK for each packet in the packet pair, the sender estimates the bottleneck link speed from the resulting inter-ACK gap. It is assumed that the original packet spacing is preserved in the ACKs. ACKs are sent as data packets arrive and, with round-robin scheduling, their spacing does not change.

Keshav points out that such a control-theoretic approach would be very hard to implement on a network where all routers use FIFO queuing because there is no easy method for monitoring the network state. With round-robin queuing, a flow could determine how many other flows shared the link by observing the time between two of its packets being sent.

The goal of this flow control algorithm is to maintain a certain number of packets queued at the bottleneck router. For example, for maximum link utilization, there should always be at least one packet in the queue to ensure there is data to send when resources are available.

2.4.3 General AIMD

Chiu and Jain analyzed AIMD algorithms and showed that, for fairness and stability, α should be greater than 0 and β should be between 0 and 1 [CJ89]. Yang and Lam extend this

to show that to be considered TCP-friendly [YL00],

$$\alpha = \frac{4(1-\beta^2)}{3}.$$

Yang and Lam further found that $\alpha = 0.31$ and $\beta = 0.875$ was a TCP-friendly setting that lowered the number of rate fluctuations. The authors derive a general model for the sending rate of flow using general AIMD based on α, β, the loss rate, RTT, timeouts, and the number of segments acknowledged by each ACK[2].

2.4.4 DECbit

Ramakrishnan and Jain propose a scheme called DECbit where a router indicates congestion to a source by setting a bit in the packet header during times of congestion [RJ90]. The router indicates congestion when the average queue size is non-zero. If a receiver sees congestion bits set in an incoming packet, it echos those bits in the ACK to notify the sender of congestion. If a majority of ACKs from the latest window have congestion indication bits set, *cwnd* is decreased. Otherwise, *cwnd* is increased. The authors follow the AIMD congestion window adjustment policy and recommend setting $\alpha = 1$ and $\beta = 0.875$.

The average queue size is based upon the queue regeneration cycle, which is the time between busy and idle periods (build up and drain of the queue). The average queue size during a regeneration cycle is given by the sum of the queue size during the cycle divided by the time of the cycle. For making congestion decisions, the router computes the average queue size based on both the current regeneration cycle and the previous regeneration cycle. Since this computation is performed for every packet entering the queue, as the current regeneration cycle grows, the contribution of the previous regeneration cycle diminishes. This mechanism allows for the congestion signal to take into account long regeneration cycles and use more current information in this case.

The sender waits for a window's worth of packets to be acknowledged after a change to *cwnd* before making another change to *cwnd*. When a window update is made, the first packet to reflect the change in window size is the ACK of the first packet of the next window. The sender waits until the entire next window has been acknowledged before deciding how to update *cwnd*. For example, if W_p is the window size before the change and W_c is the window size after the change, a new change to the window is not made until at least $(W_p + W_c)$ packets have been acknowledged (which requires waiting approximately two RTTs). The sender only uses the information from the last W_c ACKs to make its window change decision. This reduces the oscillation of the window and keeps the sender from reacting prematurely to transient increases in delay. The authors also suggest that all history about the state of the network should be cleared after a window update has been made. Thus, after a window

[2]This takes into account delayed ACKs, where two segments can be acknowledged by each ACK.

update, only the congestion bits from the current window are used in deciding how to adjust the next window. There is a balance to be struck between the percentage of ACKs with congestion indications required for making a decision and the router's threshold for detecting congestion. The authors use a router threshold of 1 packet and a 50% percent threshold of ACKs returning with congestion indications. So, routers will mark packets with congestion indications if the average queue size is greater than 1 packet. If 50% or more of the ACKs in a window return with congestion indications, the sender will reduce *cwnd*.

2.4.5 Delay-Based Approach

Jain presents a delay-based approach to congestion avoidance where the RTT is used as a congestion signal [Jai89]. Jain also distinguishes between selfish optimum and social optimum strategies and discusses methods for deciding whether to increase or decrease the window, what increase/decrease algorithm to use, and how often to decide to change the window.

When multiple flows are competing for network resources, the flows can either choose to find the social optimum or the selfish optimum. A selfish optimum leads to packet loss because flows try to keep increasing their windows until there is no longer buffer space available. In a socially optimum scheme, some flows back off while other flows increase their windows in order for there to be fair sharing of network resources. To determine the social optimum with non-uniform packet service times, flows may need to know the number of the other flows in the network and their window sizes.

To determine how to change the window in a simplified system where there are no other flows, the *normalized delay gradient* (NDG) is computed:

$$NDG = (\frac{RTT - RTT_{old}}{RTT + RTT_{old}})(\frac{W + W_{old}}{W - W_{old}}),$$

where RTT is the RTT at window size W and RTT_{old} is the RTT at window size W_{old}. NDG is proportional to the load in the system. If $NDG > 0$, then the window is decreased. If $NDG \leq 0$, then the window is increased. As in DECbit [RJ90], this algorithm uses an AIMD adjustment with $\alpha = 1$ and $\beta = 0.875$. The window is also updated every other RTT.

Jain suggests that if the minimum delay is known, then the socially optimum window size can be determined without knowing the windows of the other flows. This is done by estimating the load put on the network by the other flows, which is proportional to the difference in the delay at startup (when $W = 1$) and the minimum delay.

2.4.6 Binomial Algorithms

Bansal and Balakrishan present binomial algorithms, where the window increase/decrease factors are proportional to a power of the current congestion window [BB01]. In general, the increase algorithm is $w(t + 1) = w(t) + \alpha/w(t)^k$, and the decrease algorithm is

$w(t+1) = w(t) - \beta w(t)^l$. AIMD is represented by $k = 0$ and $l = 1$. Multiplicative-increase/multiplicative-decrease (MIMD), used in TCP slow start, is $k = -1$ and $l = 1$. The authors show that as long as $k + l = 1$ and $l \leq 1$, the binomial algorithm is TCP-compatible. Bansal and Balakrishnan look particularly at two examples of the binomial algorithm: inverse increase/additive decrease (IIAD) where $k = 1$ and $l = 0$ and SQRT where $k = 1/2$ and $l = 1/2$. TCP flows using binomial algorithms can see higher throughput than competing TCP AIMD flows when drop-tail routers are used. This unfairness to TCP AIMD flows would be remedied if routers with active queue management (specifically RED, see section 2.5.1) were used instead of drop-tail routers. The authors argue that this unfairness is not a problem for binomial algorithms, but rather evidence that drop-tail queuing should not be used.

2.5 Active Queue Management

Internet routers today employ traditional FIFO queuing (called "drop-tail" queue management). Active queue management (AQM) is a router-based congestion control mechanism where a router monitors its queue size and makes decisions on how to admit packets to the queue. Traditional routers use a drop-tail policy, where packets are admitted whenever the queue is not full. A drop-tail router thus only monitors whether or not the queue is filled. AQM routers potentially drop packets before the queue is full. These actions are based on the fundamental assumption that most of the traffic in the network employs a congestion control mechanism, such as TCP Reno, where the sending rate is reduced when packets are dropped. Many AQM algorithms are designed to maintain a relatively small queue but allow short bursts of packets to be enqueued without dropping them. In order to keep a small queue, often many packets are dropped "early," *i.e.*, before the queue is full. A small queue results in lower delays for packets that are not dropped. The low delay resulting from a small queue potentially comes at the cost of higher packet loss due to early drops than would be seen with losses only due to an overflowing queue as with drop-tail routers.

2.5.1 Random Early Detection

Random Early Detection (RED) is an AQM mechanism that seeks to reduce the long-term average queue length in routers [FJ93]. Under RED, as each packet arrives, routers compute a weighted average queue length that is used to determine when to notify end-systems of incipient congestion. Congestion notification in RED is performed by "marking" a packet. If, when a packet arrives, congestion is deemed to be occurring, the arriving packet is marked. For standard TCP end-systems, a RED router drops marked packets to signal congestion through packet loss. If the TCP end-system understands packet-marking, a RED router marks and then forwards the marked packet towards its destination.

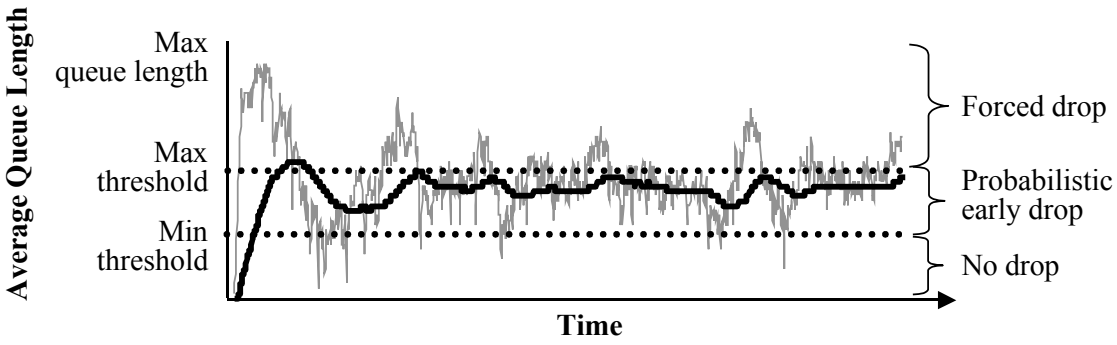

Figure 2.1: RED States. The gray line is the instantaneous queue size, and the black line is the weighted average queue size.

RED's congestion detection algorithm depends on the average queue size and two thresholds, min_{th} and max_{th}. When the weighted average queue size is below min_{th}, RED acts like a drop-tail router and forwards all packets with no modifications. When the average queue size is between min_{th} and max_{th}, RED probabilistically marks incoming packets. When the average queue size is greater than max_{th}, all incoming packets are dropped. These states are shown in Figure 2.1.

The more packets a flow sends, the higher the probability that its packets will be marked. In this way, RED spreads out congestion notifications proportionally to the amount of space in the queue that a flow occupies. The probability that an incoming packet will be dropped is varied linearly from 0 to max_p when the average queue size is between min_{th} and max_{th}. This is illustrated in Figure 2.2.

The following equations are used in computing the average queue size and drop probability in RED:

- exponential weighted moving average:
  $avg_{new} = (1 - w_q)avg_{old} + w_q *$ current queue length
- drop probability: $p_a = p_b/(1 - count * p_b)$, where
  $p_b = (max_p(avg - min_{th}))/(max_{th} - min_{th})$ and *count* is the number of packets that have been admitted to the queue since the last packet was marked

The RED thresholds min_{th} and max_{th}, the maximum drop probability max_p, and the weight given to new queue size measurements w_q, play a large role in how the queue is managed. Recommendations on setting these RED parameters specify that max_{th} should be set to three times min_{th}, w_q should be set to 0.002, or 1/512, and max_p should be 10% [Flo97].

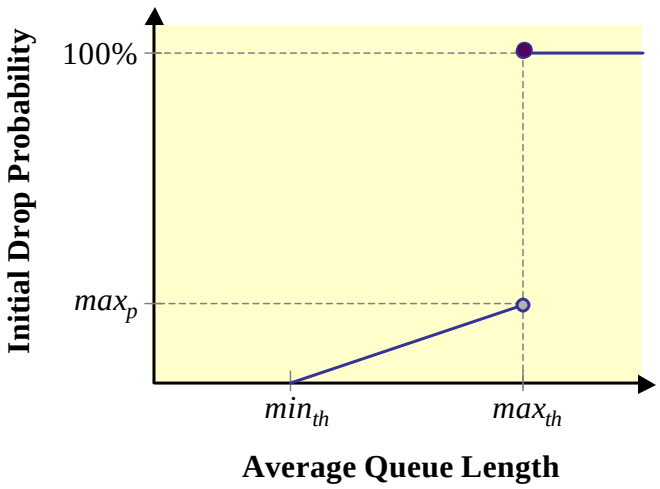

Figure 2.2: RED Initial Drop Probability (p_b)

There have been several studies [CJOS01, BMB00] of network performance with RED routers. Christiansen *et al.*, in particular, looked at the performance of HTTP traffic over RED routers [CJOS01]. Their original goal was to find RED parameter settings that were optimal for links that carried a large majority of HTTP traffic. What they found was that end-user metrics of performance were very sensitive to the parameter settings, the recommended settings performed worse than if the network consisted of all drop-tail routers, and determining optimal parameters for RED was non-intuitive.

2.5.2 Explicit Congestion Notification

Explicit Congestion Notification (ECN) is a form of AQM that allows routers to notify end systems when congestion is present in the network by setting a bit in the packet, as in DECbit [Flo94, RF99]. Ramakrishnan and Floyd set down the following design assumptions (or limitations) of ECN, which are applicable to any early congestion detection method [RF99]:

- ECN can only help alleviate congestion events that last longer than a RTT.
- ECN can only help alleviate congestion caused by flows that carry enough segments to be able to respond to feedback.
- It is likely that the paths followed by data and ACKs are asymmetric.

ECN is recommended for use in routers that monitor their average queue lengths over time (*e.g.*, routers running RED), rather than those that can only measure instantaneous queue lengths. This allows for short bursts of packets without triggering congestion notifications. Most studies of ECN have assumed that RED is the AQM scheme used.

When an ECN-capable RED router detects that its average queue length has reached a min_{th}, it marks packets by setting the CE ("congestion experienced") bit in the packets' IP

headers. When an ECN-capable receiver sees a packet with its CE bit set, an ACK with its ECN-Echo bit set is returned to the sender. Upon receiving an ACK with the ECN-Echo bit set, the sender reacts in the same way as it would react to a packet loss (*i.e.*, by halving the congestion window). Ramakrishnan and Floyd recommend that since an ECN notification is not an indication of packet loss, the congestion window should only be decreased once per RTT, unless packet loss does occur. Thus, a TCP sender implementing ECN receives two different notifications of congestion, ECN and packet loss. This allows senders to be more adaptive to changing network conditions.

ECN has the same feedback constraints as any closed-feedback loop scheme. It takes at least 1/2 RTT for the ECN notification to reach the sender after congestion has been detected by the router. Since congestion can shift to different parts of the network, ECN works optimally when both end systems and all intermediate routers are ECN-capable.

Studies of network performance of ECN-capable RED routers have shown that using ECN is much more preferable than dropping packets upon congestion indications [SA00, ZQ00]. Packet loss rates are lower and throughput is higher. Note that performance of ECN-capable flows is still closely tied to the parameters of the AQM scheme used.

2.5.3 Adaptive RED

Adaptive RED is a modification to RED which addresses the difficulty of setting appropriate RED parameters [FGS01]. Studies have found that the "best" parameter settings depend upon the traffic mix flowing through the router, which changes over time. RED performance depends upon how the four parameters min_{th} (minimum threshold), max_{th} (maximum threshold), max_p (maximum drop probability) and w_q (weight given to new queue size measurements) are set. Adaptive RED adapts the value of max_p so that the average queue size is halfway between min_{th} and max_{th}. The maximum drop probability, max_p is kept between 1-50% and is adapted gradually. Adaptive RED includes another modification to RED, called "gentle RED" [Flo00b]. In gentle RED, when the average queue size is between max_{th} and $2 * max_{th}$, the drop probability is varied linearly from max_p to 1, instead of being set to 1 as soon as the average is greater than max_{th}. These modifications to max_p and the drop probability are shown in Figure 2.3. Additionally, when the average queue size is between max_{th} and $2 * max_{th}$, selected packets are no longer marked, but always dropped. These states are pictured in Figure 2.4.

Adaptive RED adds two parameters to RED, α and β. α controls the increase rate of max_p, and β is the decrease factor of max_p. The authors suggest that $\alpha < \frac{max_p}{4}$ so that, in each interval, the average queue size will not fluctuate too wildly. The default setting of α is $min(0.01, \frac{max_p}{4})$. The same consideration should be made in setting β. The authors suggest that $\beta > 0.83$.

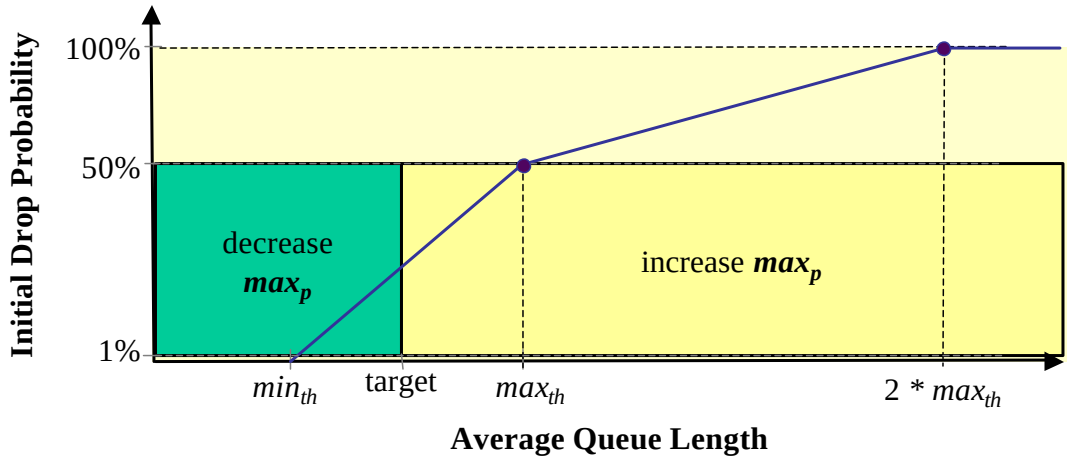

Figure 2.3: Adaptive RED Initial Drop Probability (p_b)

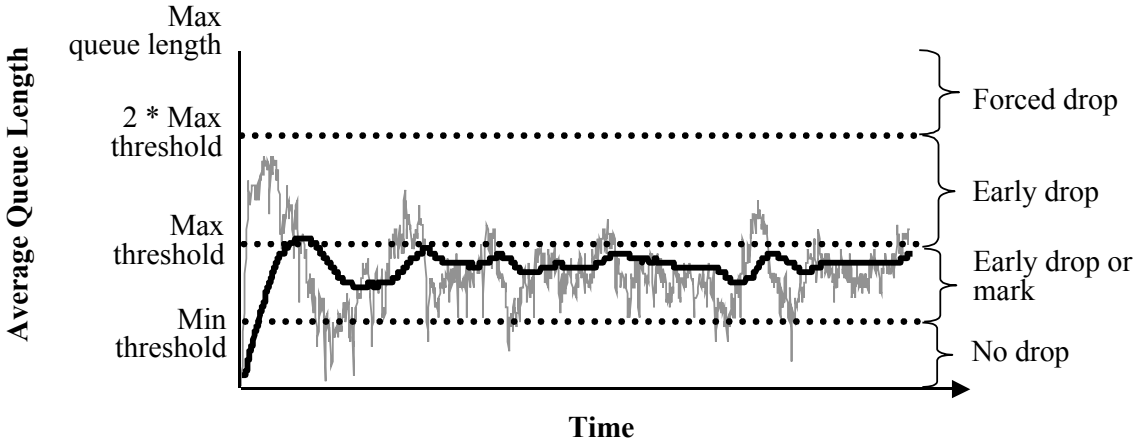

Figure 2.4: Adaptive RED States. The gray line is the instantaneous queue size, and the black line is the weighted average queue size.

Adaptive RED's developers provide guidelines for the automatic setting of min_{th}, max_{th}, and w_q. Setting min_{th} results in a tradeoff between throughput and delay. Larger queues increase throughput, but at the cost of higher delays. The rule of thumb suggested by the authors is that the average queuing delay should only be a fraction of the RTT. If the target average queuing delay is $delay_{target}$ and C is the link capacity in packets per second, then min_{th} should be set to $delay_{target} * C/2$. The guideline for setting max_{th} is that it should be $3 * min_{th}$, resulting in a target average queue size of $2 * min_{th}$. The weighting factor w_q controls how fast new measurements of the queue affect the average queue size and should be smaller for higher speed links. This is because a given number of packet arrivals on a fast link represents a smaller fraction of the RTT than for a slower link. It is suggested that w_q be set as a function of the link bandwidth, specifically, $1 - e^{-1/C}$.

2.6 Analysis of Internet Delays

Sync-TCP is based on the idea that having exact information about the delays that packets from a flow encounter can be used to improve congestion control. This section highlights previous work on how delays can be used to determine certain characteristics of a network.

2.6.1 Queuing Time Scales

Paxson looked at estimating the time scales over which queuing varies [Pax99]. The time scale of queuing delay variation is important because if queues fluctuate over a time scale less than a RTT, then rate adaptations based on queuing delay would not have time to acquire feedback about the effect of the adaptation, and therefore would be useless. Paxson obtained one-way transit time (OTT) data from off-line analysis of packet traces gathered from 20,000 TCP bulk transfers. Clock errors in the traces were detected using algorithms developed for finding clock adjustments and skew [Pax98].

Paxson found that, in the traces, flows typically experience the greatest variation in queuing delay on time scales between 0.1 - 1 seconds. In other words, if a connection was divided into intervals of time, the largest queuing delay variation would be seen in intervals whose length was between 0.1 - 1 seconds. This is encouraging because it indicates that the majority of congestion periods (where queuing delays are high) occur on time scales likely to be longer than most RTT values (typically 20-200 ms), so adaptations have the potential to have a positive effect on congestion.

2.6.2 Available Bandwidth Estimation

Paxson also looked at using OTTs to estimate the amount of available bandwidth in the network [Pax99]. Let λ_i be the amount of time spent by packet i at the bottleneck for transmission and queuing behind packets in the same flow as packet i. The service time of

a b-byte packet, Q_b, is b/B, where B is the bottleneck bandwidth in bytes per second. For all packets with equal b, variation in OTT will reflect the queuing delay of a packet behind packets in its own connection. Variation in OTT, ψ_i, is given by $\lambda_i - Q_b$. The packet's total queuing delay, γ_i, is calculated by subtracting the packet's OTT from the connection's minimum observed OTT. If the connection is the only traffic on the path, $\psi_i = \gamma_i$. All queuing delay is due to other packets of the same connection. Let

$$\beta = \frac{\sum_i (\psi_i + Q_b)}{\sum_j (\gamma_j + Q_b)},$$

where the summation range is over all packets in a connection. β is the proportion of the packet's delay due to packets from its own connection and reflects the amount of available bandwidth in the network. If $\beta \approx 1$, then the connection is receiving all of the bandwidth that it is requesting. All of the queuing delay experienced by the packets is due to that connection's own packets. If $\beta \approx 0$, then this connection's packets are spending the majority of their queuing time behind packets from other connections. So, the load contributed by this connection is relatively insignificant. $\beta \approx 1$ does not mean that the connection is consuming all of the network resources, just that any time it tried to consume more resources, they were available. By computing β from actual traces, Paxson reported that paths with higher bottleneck bandwidths usually carry more traffic (and more connections) and have lower values of β. Also, β is a fairly good predictor of future levels of available bandwidth. Paxson observed that for the same path, one observation of β will be within 0.1 of later observations for time periods up to several hours. Allman and Paxson mention the possibility of using a measure of the available bandwidth in the network to allow connections to ramp up to their fair share more quickly than slow start [AP99].

2.6.3 Delays for Loss Prediction

One of the problems with current congestion control is that on a wireless link it is difficult to distinguish between losses due to wireless link errors and losses due to congestion. The detection of congestion losses should trigger a reduction in the sending rate, while link error losses should not cause a reduction in the sending rate.

Samaraweera and Fairhurst looked at the feasibility of using changes in RTTs as a network congestion detector [SF98]. This would allow wireless senders to distinguish between congestion losses (those preceded by an increase in RTT) from link error losses (those that follow no discernible increase in RTT). Using changes in the RTT as an estimate of queuing delay requires obtaining a minimum RTT measurement close to the actual round-trip propagation delay of the flow. Subsequent increases in RTT can be compared to the minimum RTT to determine the degree of congestion. Samaraweera and Fairhurst noted three problems with such an end-system-based method of inferring the type of loss:

- persistent congestion will cause an incorrect estimate of the minimum delay,
- AQM techniques will cause packet drops without a large build-up of the queue, and
- routing changes will make consecutive delay estimates unrelated.

Biaz and Vaidya looked at how well congestion avoidance algorithms predict packet loss [BV98]. The idea was that a congestion avoidance algorithm was good at predicting loss if, when it indicated that the congestion window should be increased, no packet loss occurred. For a predictor to be accurate, the authors suggested that the following requirements should be met:

- packet losses follow a large build-up in the queue,
- a build-up in the queue usually results in packet loss, and
- the predictor correctly diagnoses a large build-up in the queue.

Through live Internet tests and *ns* simulations, they found that the level of aggregation prevented a single flow's congestion avoidance algorithm from accurately predicting packet loss. The changes in the congestion window of a single flow did not have a large enough effect on the size of the queue at the bottleneck link to prevent packet loss.

Bolot found that unless a flow's traffic comprised a large fraction of the bottleneck link traffic, packet losses appeared to occur randomly (*i.e.*, without a corresponding build-up in delay) [Bol93]. If a flow is a small fraction of the traffic, only a few of its packets would be in the congested router's queue at any one time. With so few samples of the queuing delay during a time of congestion, it would be difficult to observe a significant increase in the queuing delay before packet loss occurred.

2.6.4 Delays for Congestion Control

Martin *et al.* looked at how well TCP flows with delay-based congestion avoidance mechanisms (DCAs), such as TCP Vegas, could compete with TCP Reno flows [MNR00]. Martin showed that small numbers of DCA flows cannot co-exist in the same network with TCP Reno flows without receiving less bandwidth than if there were no TCP Reno flows. The problem is that DCAs and TCP Reno have opposite goals. DCAs react to increased delays by slowing down their sending rates. TCP Reno flows try to overflow the queue in order to determine the available bandwidth of the network. Any bandwidth that DCA flows give up will be taken by TCP Reno flows, which then would cause DCA flows to back off even further. This study looked at the impact that competing with many TCP Reno flows would have on a single DCA flow. It did indicate, though, that if DCA flows represented a significant percentage of traffic, overall packet loss rates decreased.

2.7 Summary

Researchers have approached TCP congestion control either by adding mechanisms to the queuing algorithm in routers or by changing the TCP Reno congestion control protocol itself.

Those who look at the queuing algorithm in routers come from the point-of-view that the main problem is found in drop-tail queuing rather than TCP Reno. Since packet drops indicate to TCP Reno that congestion is occurring, the queuing algorithm should drop some packets before the queue overflows, whenever the router detects that congestion is occurring. The router monitors its queue and makes a decision of which packets to drop at what point in time. This approach adapts the queuing mechanism according to the behavior of the prevalent transfer protocol (*i.e.*, TCP). A change is made at the router in order to produce the desirable behavior in the end systems. If, in the future, a new congestion control protocol were released that did not depend on packet loss for congestion signals, these "early drop" queuing mechanisms might be a hindrance.

The opposite approach is to look at what the congestion control algorithm on the end system can do to detect congestion before it results in packet loss. Researchers have looked at using RTTs and throughput estimates as indicators of congestion [BOP94, PGLA99, Jai89]. This previous work is based on the assumption that the flow's RTT is composed equally of the data path delay and the ACK path delay. There is no guarantee that this assumption would be true.

Sync-TCP, an end-system approach to TCP congestion control, is built on the foundation of TCP Reno and uses TCP Reno's loss detection and error recovery mechanisms. Sync-TCP is also built on the work of researchers who looked at changes in RTTs and throughput as indications of congestion. I have added the assumption that all computers have synchronized clocks (*e.g.* via GPS) in order to investigate the benefit that could come from being able to accurately measure OTTs and separate the detection of data-path congestion from ACK-path congestion. As will be shown in detail in Chapter 3, the TCP timestamp option is the basis for the format of the Sync-TCP timestamp option that will be used to communicate synchronized timestamps between computers.

The idea behind Sync-TCP is that flows can be good network citizens and strive to keep queues small while still achieving high goodput. Jain's work on delay-based congestion control introduced the idea of the selfish optimum vs. the social optimum congestion control algorithm. The social optimum allows for some flows to slow down their sending rates while other flows increase their sending rates in order to maximize the efficiency of the network without overflowing network buffers. What has resulted with Sync-TCP, and will be shown in Chapter 4, is that some longer flows will voluntarily back off when congestion is detected so that short flows (those that are too short to run the Sync-TCP congestion control algorithm) see short queues and complete quickly. This does not mean that the long flows are starved (many even see better performance than with TCP Reno), but that all of the flows in the

network are cooperating to provide a better network environment featuring less packet loss and shorter queue sizes.

With Martin's work on the deployability of DCAs in mind, Sync-TCP will be evaluated in a homogeneous environment, where all flows will use the same congestion control protocol. The goal of my study is to look at the usefulness of synchronized clocks and exact timing information in TCP congestion control. If synchronized clocks are found to be useful in an ideal environment (all clocks are perfectly synchronized and no competing TCP Reno traffic), then techniques for incorporating these changes into the Internet can be developed later.