Chapter 3

Sync-TCP

Sync-TCP is a family of end-to-end congestion control mechanisms that are based on TCP Reno, but take advantage of the knowledge of one-way transit times (OTTs) in detecting network congestion. I use Sync-TCP as a platform for studying the usefulness of adding exact timing information to TCP.

This chapter describes the development of Sync-TCP, which consists of the following:

- identifying parts of TCP congestion control that could be improved,
- developing a mechanism so that computers with synchronized clocks can exchange OTTs,
- developing mechanisms that can detect network congestion using OTTs, and
- developing mechanisms to react appropriately to network congestion.

Qualitative evaluations of the congestion detection mechanisms are provided in this chapter, while evaluations of selected congestion detection mechanisms coupled with appropriate congestion reaction mechanisms are described in Chapter 4.

3.1 Problem and Motivation

On the Internet today, there is no deployed end-to-end solution for detecting network congestion before packet loss occurs. Detecting network congestion before packet loss occurs is the goal of active queue management (AQM), in general, and Random Early Detection (RED), in particular. These solutions, though, require changes to routers and are not universally deployed. The emergence of cheap Global Positioning System (GPS) receiver hardware motivates the study of how synchronized clocks at end systems could be used to address the problem of early congestion detection. In this section, I will discuss network congestion and problems with TCP congestion control.

3.1.1 Network Congestion

Network congestion occurs when the rate of data entering a router's queue is greater than the queue's service rate for a sustained amount of time. The longer a queue, the longer the queuing delay for packets at the end of the queue. It is desirable to keep queues small in order to minimize the queuing delay for a majority of packets. Because of the large numbers of flows that can be aggregated in a queue (and because of protocol effects such as TCP slow start), transient spikes in the incoming rate (and, therefore, in the router queue size) are expected and should be accommodated. In order for these spikes to be handled, the average queue size at a router should be small relative to the total queuing capacity of the router.

3.1.2 TCP Congestion Control

TCP's approach to congestion control is to conservatively probe the network for additional bandwidth and react when congestion occurs. End-to-end protocols, such as TCP, treat the network as a black box, where senders must infer the state of the network using information obtained during the transfer of data to a receiver. A TCP sender infers that additional bandwidth is available in the network by the fact that acknowledgments (ACKs) return for data it has sent. The sender infers that segments have been dropped by the expiration of the retransmission timer (reset whenever an ACK for new data is received) or by the receipt of three duplicate ACKs. TCP senders infer that network congestion is occurring when segments have been dropped. One problem with this inference is that TCP has no other signal of congestion. The only time a TCP sender knows to decrease its sending rate is in response to a segment drop. Thus, TCP's congestion control mechanism is tied to its error recovery mechanism. By using segment loss as its only signal of congestion and the return of ACKs as a signal of additional bandwidth, TCP drives the network to overflow, causing drops, retransmissions, and inefficiency. This inefficiency results from some segments being carried in the network only to be dropped before reaching their destination.

3.2 Goals

The main goal of Sync-TCP is to show that information from computers with synchronized clocks could be used to improve TCP congestion control. One of the ways that TCP congestion control could be improved is to provide more efficient transfer of data by reducing segment loss. If Sync-TCP can accurately detect and react to congestion without having to wait for segment loss, a network of Sync-TCP flows should see lower loss rates than a network of TCP Reno flows. Additionally, the information from synchronized clocks could be used to provide a richer congestion signal than the binary (congestion or no congestion) signal of TCP Reno. Given a richer congestion signal, a Sync-TCP sender could react in different ways to different degrees of congestion.

3.3 One-Way Transit Times

If two end systems have synchronized clocks, they can accurately determine the OTTs between them. If the OTTs are known, the queuing delay of the path can be computed. The difference between the minimum OTT and the current OTT is the forward-path queuing delay. As queues in the network increase, the queuing delay will also increase and could be used as an indicator of congestion.

Using queuing delays as a signal of congestion would allow a sender to reduce its sending rate before queues in routers overflow, avoiding losses and costly retransmissions. Monitoring OTTs for indications of congestion also has the benefit of allowing senders to quickly realize when periods of congestion have ended, where they could increase their sending rates more aggressively than TCP Reno.

Sync-TCP, an end-to-end approach to congestion control, uses OTTs to determine congestion, because OTTs can more accurately reflect queuing delay caused by network congestion than round-trip times (RTTs). Using RTTs there is no way to accurately compute the forward-path queuing delay. If there is an increase in the RTT, the sender cannot distinguish between the cause being congestion on the forward path or congestion on the reverse path.

There are a few limitations to using OTTs to compute queuing delays. If there is congestion in the network that exists when a flow begins and this congestion lasts for the entire duration of the flow, then the flow's minimum-observed OTT would not represent the propagation delay of the path. In this case, the difference between the current OTT and the minimum-observed OTT would not be a clear picture of the actual queuing delay.

Assuming, though, that routes are stable[1] and persistent congestion is not occurring, the minimum-observed OTT is (or, is close to) the propagation delay. This minimum-observed OTT may decrease if the flow began when the network was already congested. As the queues drain, the minimum-observed OTT will decrease. Because of the fluctuations in most queues, the minimum-observed OTT probably will be determined fairly early and remain stable for the remainder of the flow's duration.

3.4 Sync-TCP Timestamp Option

If two computers have synchronized clocks, they can determine the OTTs between them by exchanging timestamps. For Sync-TCP, I added an option to the TCP header, which is based on the RFC 1323 timestamp option [JBB92]. Figure 3.1 pictures the full TCP header along with a cut-out of the Sync-TCP timestamp option[2]. This TCP header option includes the OTT in μs calculated for the last segment received (OTT), the current segment's sending

[1] Paxson reported that in 1995 about 2/3 of the Internet had routes that persisted on the time scale of days or weeks [Pax97].

[2] See [Ste94] for a description of the header fields.

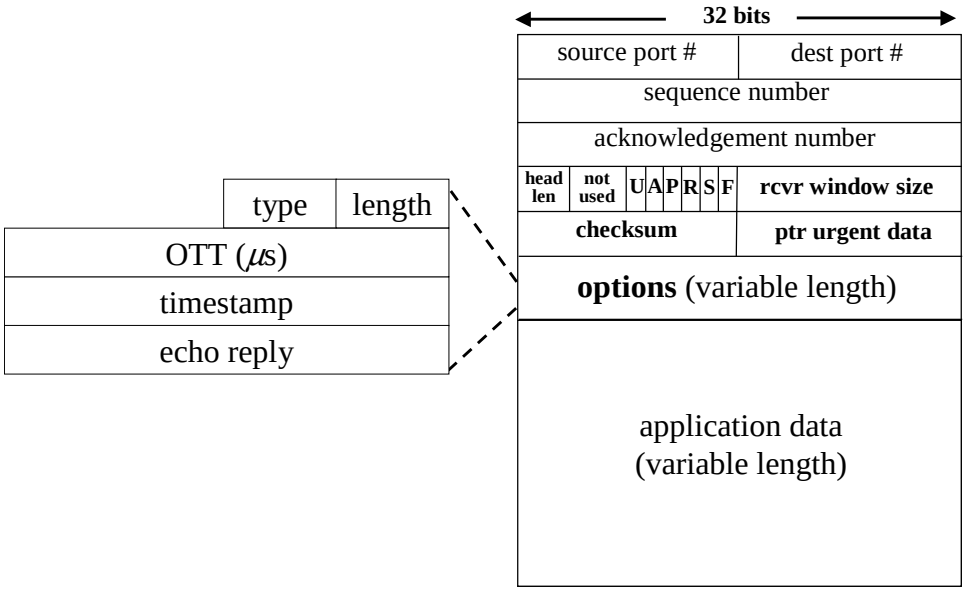

Figure 3.1: TCP Header and Sync-TCP Timestamp Option

time (*timestamp*), and the sending time of the last segment received (*echo reply*).

When the receiver sees any segment with the Sync-TCP timestamp option, it calculates the segment's OTT by subtracting the time the segment was sent from the time the segment was received. The OTT is then inserted into the next segment the receiver sends to the sender (generally an ACK). Upon receiving the ACK, the sender can calculate the following metrics:

- OTT of the data segment being acknowledged – This is provided in the OTT field of the Sync-TCP timestamp option.
- OTT of the ACK – This is the receive time minus the time the ACK was sent.
- The time the data segment was received – This is the time the data segment was sent plus the OTT.
- The time the ACK was delayed at the receiver – This is the time the ACK was sent minus the time the data segment was received.
- The current queuing delay – This is the current OTT minus the minimum-observed OTT.

Figure 3.2 provides an example of Sync-TCP timestamp option exchange and the computation of the OTTs. The first data segment is sent at time 2 and, for initialization, contains -1 as the entries for the OTT and the echo reply fields. When the data segment is received at time 3, the OTT is calculated as 1 time unit. When the ACK is generated and sent at time 4, the OTT of the data segment, the timestamp of the ACK, and the timestamp found in the data segment's option are sent in back in the Sync-TCP timestamp option to the sender. At time 5, when the ACK is received, the sender can calculate the OTT of the data segment it

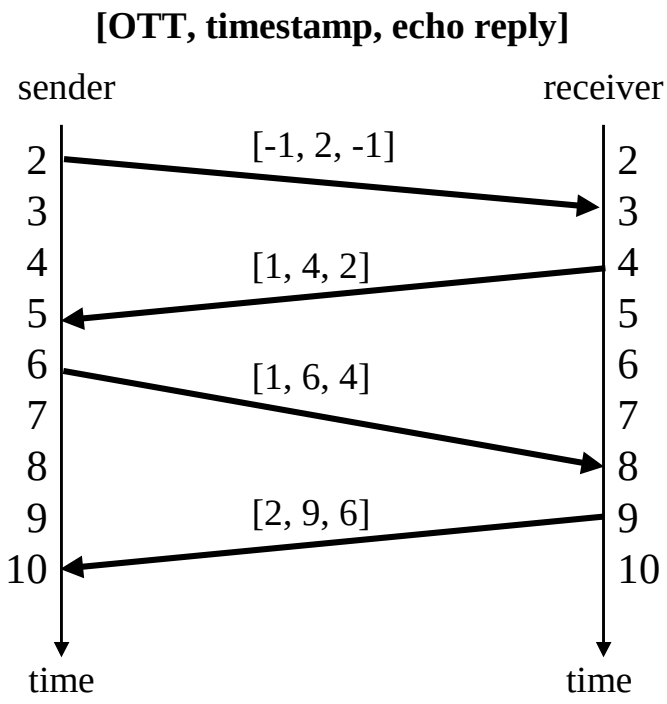

Figure 3.2: Sync-TCP Example

sent at time 2 (1 time unit), the OTT of the ACK (1 time unit), the time the data segment was received (time 3), and the amount of time the ACK was delayed (1 time unit). At time 10 when the second ACK is received, the OTT of the data segment is 2. Since the sender previously saw an OTT of 1, it can determine that the current queuing delay is 1 time unit.

3.5 Congestion Detection

In all of the congestion detection mechanisms I discuss, queuing delays are calculated from the exchange of OTTs in order to detect congestion. I began my study by looking at four different methods for detecting congestion using OTTs and queuing delays (each of these will be discussed in greater detail in the following sections):

- SyncPQlen (percentage of the maximum queuing delay) – Congestion is detected when the estimated queuing delay exceeds 50% of the maximum-observed queuing delay (which is an estimate of the maximum amount of queuing in the network).
- SyncPMinOTT (percentage of the minimum OTT) – Congestion is detected when the estimated queuing delay exceeds 50% of the minimum-observed OTT
- SyncAvg (average queuing delay) – Congestion is detected when the weighted average of the estimated queuing delays crosses a threshold, by default set to 30 ms.

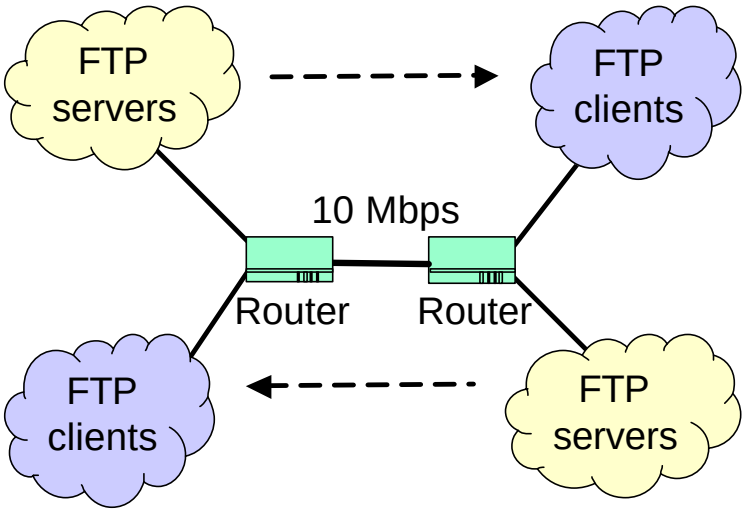

Figure 3.3: Network Topology for FTP Experiments

- SyncTrend (trend analysis of queuing delays) – Congestion is detected when the trend of the estimated queuing delays is increasing.

After evaluating these methods, I developed a congestion detection mechanism called SyncMix that is a combination of SyncPQlen, SyncAvg, and SyncTrend. In this section, I will describe each of the five congestion detection mechanisms I studied and the shortcomings of the first four mechanisms that led me to develop SyncMix.

To aid in the discussion of each of the congestion detection mechanisms, I will use a graph of the actual queuing delay over time at a bottleneck router as an example to illustrate the mechanism at work. All of the graphs are from simulations run with one 10 Mbps bottleneck link and 20 long-lived FTP flows transferred in each direction. Figure 3.3 shows the topology for these experiments. Different base (minimum) RTTs were used for the FTP flows. The motivation and source of these RTTs is explained in Chapter 4. The minimum RTT was 16 ms, the maximum RTT was 690 ms, and the median RTT was 53 ms. All of the FTP flows were run with TCP Reno as the underlying protocol and drop-tail queues at routers on each end of the bottleneck link. The congestion detection mechanisms only reported congestion but did nothing to react to those congestion indications. TCP Reno controlled all congestion window adjustments. For all of the experiments, the queue size seen at the bottleneck router was the same. Additionally, the estimated queuing delay for all of the experiments was the same (since the only variation among experiments was the congestion detection mechanism). Figure 3.4 shows the queuing delay as seen by packets entering the bottleneck router over a particular 5-second interval. The queuing delay at the router is calculated by dividing the number of bytes in the queue when the packet arrives by the speed of the outgoing link. Since

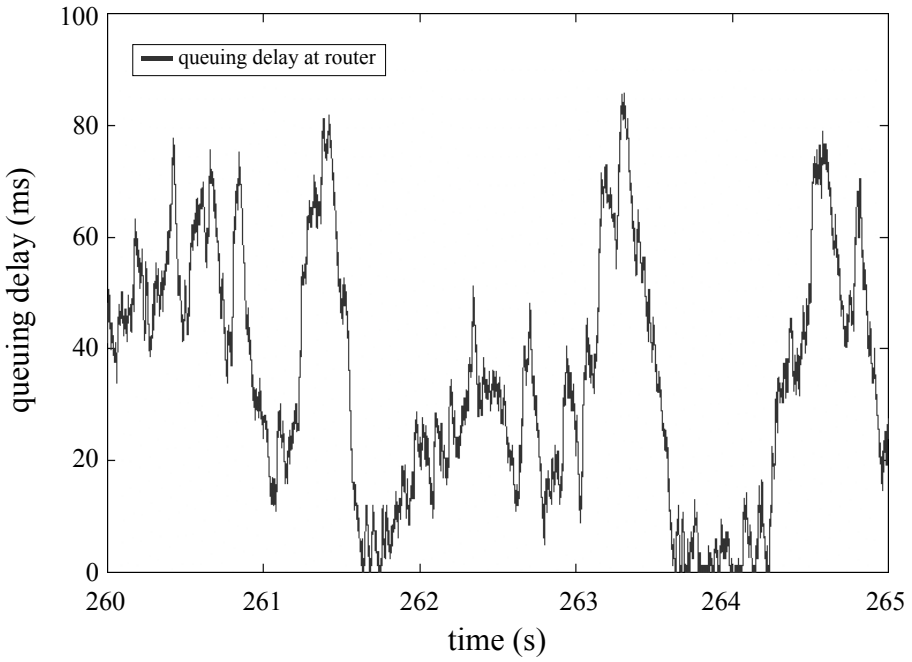

Figure 3.4: Queue Size at Bottleneck Router

these experiments were performed in simulation, the queue size (and queuing delay) can be monitored each time a new packet enters the queue. As can be seen even with a small number of long-lived flows, congestion and queuing delay varies widely. When data from a single FTP flow is shown on the graph (as will be done in the discussions of the individual congestion detection mechanisms), it is from the flow with the median RTT (53 ms).

Since Sync-TCP uses ACKs to report forward-path OTTs to the sender, the delay that a segment experiences will not be reported to the sender until its ACK is received. All of the FTP experiments were performed with delayed ACKs turned on, so in most cases, one ACK is sent to acknowledge the receipt of two data segments. If the congested link is close to the sender and the two data segments to be acknowledged arrive more than 100 ms apart, then it may take almost one RTT + 100 ms from the time congestion was observed to the time the sender knows about it. This is the worst-case delay notification. The best-case delay notification is the reverse-path OTT and would occur if the congested link was close to the receiver. If the paths are symmetric, this best-case delay is approximately 1/2 of the RTT.

Note that although the evaluation of the congestion detection mechanisms presented here considers only one congested link, experiments were run with two congested links and the same conclusions were reached. Even when there are multiple congested links, the difference between the OTT and the minimum-observed OTT estimates the total amount of queuing delay that a segment experiences. If the total amount of queuing delay increases over time,

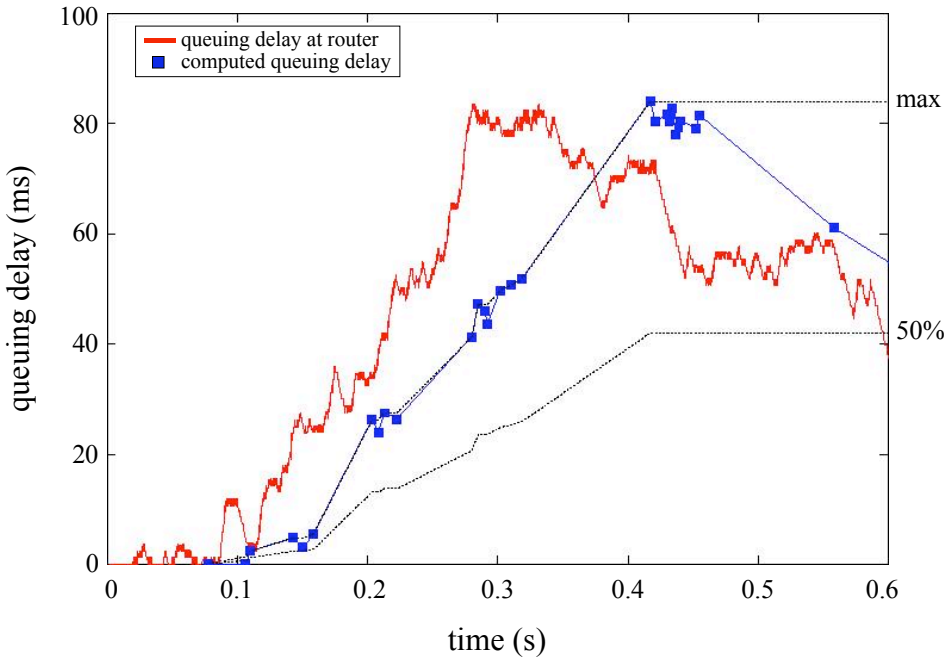

Figure 3.5: SyncPQlen Threshold Startup

then at least one of the intermediate queues is building up, signaling that congestion is occurring.

3.5.1 SyncPQlen

SyncPQlen records the minimum and maximum observed OTTs and uses them to estimate both the maximum and current queuing delay. SyncPQlen does not make congestion decisions until a segment has been dropped, so that the algorithm has a good estimate of the maximum amount of queuing in the network. The conjecture is that once a segment has been dropped, there is some segment in the flow (*i.e.*, the segment preceding or following the dropped segment) that will have seen a computed queuing delay close to the maximum amount of delay on the path, assuming no route changes. Each time a new OTT is received (*i.e.*, each time an ACK is received), SyncPQlen estimates the current queuing delay by subtracting the most recent OTT from the minimum-observed OTT. When this queuing delay exceeds 50% of the maximum-observed queuing delay, SyncPQlen signals that congestion is occurring. Frequently, both the maximum and minimum OTTs are observed fairly early in a flow's transfer. The minimum is often observed for one of the first segments to be transferred, and the maximum is often observed at the end of slow start when segments are dropped.

Figure 3.5 shows how the 50% of maximum-observed queuing delay threshold value changes with the maximum-observed OTT before segment loss occurs. The gray line is the queuing

delay as observed at the bottleneck link. The black squares are times at which ACKs were received and queuing delays were calculated at the sender for the corresponding ACKs. (The line between the squares is just a visualization aid. The queuing delay is only computed at discrete points.) The upper dotted line is the maximum-observed queuing delay. The lower dotted line is the SyncPQlen threshold of 50% of the maximum-observed queuing delay. A segment drop occurred just before time 0.3 (though it is not pictured). Around time 0.4 the sender receives a queuing delay sample reflecting the high queuing delay associated with the full queue. Until the segment loss is detected, the maximum-observed queuing delay is close to the current computed queuing delay as the queue builds up. After loss occurs, the computed and actual delays decrease but the maximum stays constant and will never decrease. Once segment loss has occurred, the assumption is that if the computed queuing delay gets close to the maximum-observed delay, the potential exists for additional segment loss to occur. Note the lag between the time the segment experienced a delay (on the gray line) and the time the sender was notified of that delay (at the black squares), *i.e.*, the horizontal distance between the gray and black lines at a given queuing delay. This lag is a function of the RTT and the actual queuing delay and will have a great impact on the algorithm's ability to sense congestion and react in time to avoid loss.

Discussion

Figure 3.6 shows the SyncPQlen congestion detection for one long-lived FTP flow with a 53 ms base RTT. The gray line is the queuing delay experienced by packets entering the router and is the same as in Figure 3.4. The black squares are the queuing delays computed by the sender. The top dotted line is the maximum-observed queuing delay, which occurred at some point previously in the flow's lifetime. The bottom dotted line is the 50% of maximum-observed queuing delay threshold. Whenever the most recent computed queuing delay is greater than the threshold, congestion is indicated. Whenever the most recent computed queuing delay value is lower than the threshold, the network is determined to be uncongested. By time 260, the flow has experienced at least one dropped segment, so the flow has a good estimate of the maximum-observed queuing delay. This is also evidenced by the fact that the maximum-observed queuing delay remains constant.

If there is a good estimate of the maximum amount of queuing available in the network, 50% of this value should be a good threshold for congestion. The goal is to react to congestion before packets are dropped. Using 50% of the maximum allows for there to be some amount of queuing, but allows flows to react before there is danger of packet loss. Using a threshold based on the amount of queuing delay in the network allows for multiple congested links. There is no predetermined threshold value, so the threshold can adapt to the total amount of queuing capacity in the path. As the number of bottlenecks increases (*i.e.*, between different paths) the queuing capacity in the network also increases. There is no way to pick a constant

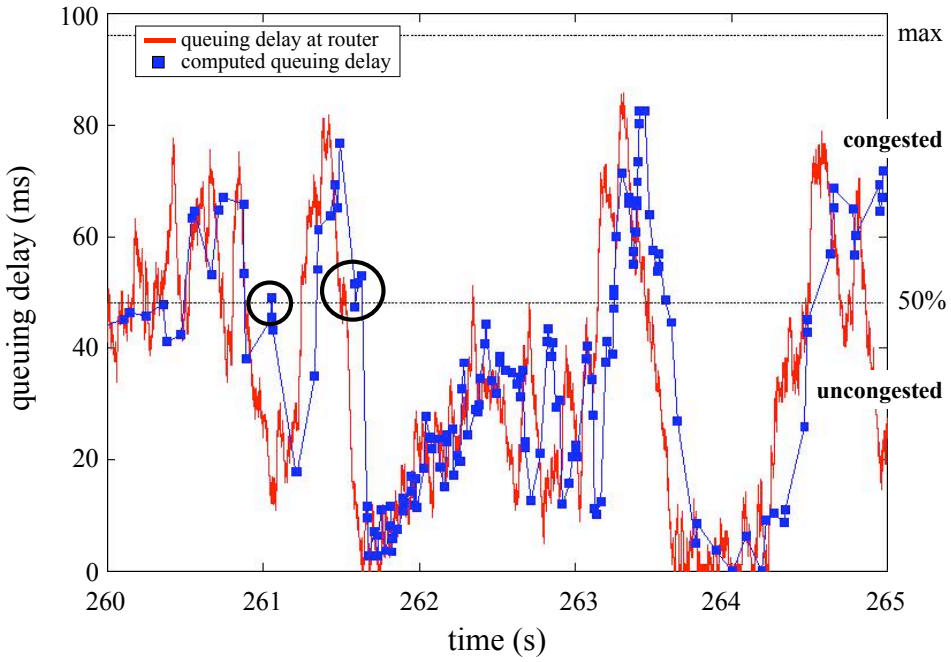

Figure 3.6: SyncPQlen Congestion Detection

queuing delay threshold without knowing the characteristics of the path ahead of time. Basing the threshold on a percentage of the maximum-observed queuing delay allows SyncPQlen to use a threshold without requiring *a priori* knowledge of the path.

One disadvantage of this algorithm is that until a segment is lost, SyncPQlen does not have a good estimate of the maximum queue size. Because of this limitation, this algorithm cannot operate during a flow's initial slow start phase. This may prevent the algorithm from operating on many short-lived flows, such as those prevalent in HTTP traffic.

Another disadvantage is that SyncPQlen operates on instantaneous values of the computed queuing delay. The circled areas in Figure 3.6 point out situations where the algorithm fails because of the noise in the instantaneous queuing delays. Just one sample above the threshold (*e.g.*, in the areas circled on the graph) would cause a congestion notification and could cause a reduction in the flow's sending rate, depending on the reaction mechanism. The algorithm is reacting to transient spikes in computed queuing delay due to using instantaneous OTTs. These spikes are a problem because they are not indicators of long-term congestion but would cause congestion to be detected.

3.5.2 SyncPMinOTT

SyncPMinOTT records the minimum-observed OTT for each flow. Anytime a new computed queuing delay is greater than 50% of the minimum-observed OTT, congestion is indi-

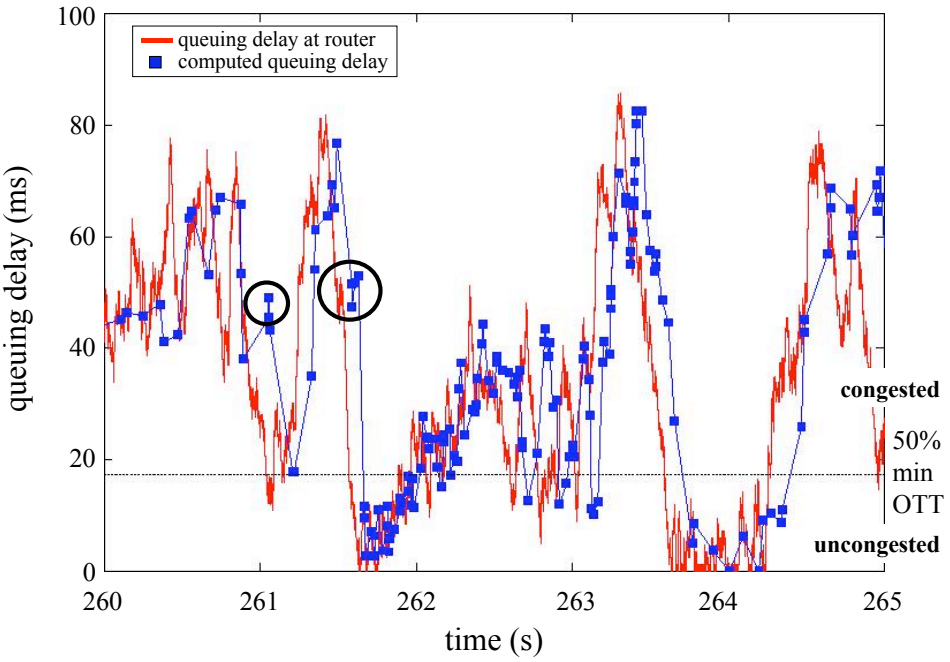

Figure 3.7: SyncPMinOTT Congestion Detection

cated. This algorithm was proposed in order to avoid waiting for a segment loss before having a good indicator of congestion. Basing the congestion detection mechanism on the minimum-observed OTT allows the congestion window of flows with large propagation delays to grow larger before SyncPMinOTT signals congestion than flows with short propagation delays. For example, if a flow had a 80 ms minimum-observed OTT, SyncPMinOTT would allow it to see computed queuing delays of up to 40 ms before signaling congestion. On the other hand, if a flow had a 20 ms minimum-observed OTT, SyncPMinOTT would signal congestion if the computed queuing delay ever reached above 10 ms. This could improve fairness between flows sharing the same path that have different RTTs, since it takes longer for flows with long base RTTs to recover from sending rate reductions than flows with short base RTTs. A flow with a long RTT would have a higher congestion threshold and hence would signal congestion less frequently than a flow with a shorter RTT.

Discussion

Figure 3.7 shows the SyncPMinOTT algorithm at work for one FTP flow with a 53 ms base RTT. As with the example in the previous section, the gray line is the queuing delay as seen by packets entering the router, and the black squares are the queuing delays as computed by the sender. (Note that since there is no congestion reaction, both the queuing delay at the router and the computed queuing delays here are exactly the same as that in Figure 3.6.) The

dotted line here is the 50% of minimum-observed OTT threshold. The minimum-observed OTT for this flow was 35 ms, so the threshold is 17.5 ms. If the most recent computed queuing delay is over the threshold, congestion is detected, otherwise the network is considered to be uncongested.

Unlike SyncPQlen, the threshold in SyncPMinOTT is not based on the amount of queuing capacity in the network, but rather on the propagation delay of the flow. No matter how many queues the flow passes through, the threshold remains the same. There could be small queues at each router, but congestion could be signaled because the sum of the queuing delays at each bottleneck would be large.

SyncPMinOTT also shares a potential disadvantage with SyncPQlen. Both algorithms react to transient spikes in computed queuing delay due to using instantaneous OTTs. These spikes are a problem because they are not indicators of long-term congestion but would cause congestion to be detected. In Figure 3.7, the circled areas are the same as in Figure 3.6. Even though the spikes in computed queuing delay are present, they are less critical because they occur well above the 50% minimum-observed OTT threshold.

Another potential disadvantage to SyncPMinOTT is how flows with asymmetric paths would be treated. For example, a flow with a large base RTT but small forward path OTT would be treated like a flow with a small base RTT, *i.e.*, one with a RTT of twice the small forward path OTT. The expectation is that flows with short forward path OTTs would have more frequent congestion indications than flows with larger forward path OTTs. This is because the flows with shorter RTTs can recover from rate reductions faster than flows with longer RTTs. A flow with a short forward path OTT but a large base RTT would then see these frequent congestion indications, but it would also take longer than a flow with a short RTT to recover. So, the flow would be unfairly penalized because of the asymmetry in its path and the large reverse path OTT.

3.5.3 SyncTrend

SyncTrend uses trend analysis to determine when computed queuing delays are increasing or decreasing. The trend analysis algorithm is taken from Jain *et al.*, which uses the trend of one-way delays to measure available bandwidth [JD02]. SyncTrend uses this trend analysis algorithm on the queuing delay samples that are computed by the sender.

Jain's trend analysis requires that samples be gathered in squares (*e.g.*, 100 samples gathered and divided into 10 groups of 10 samples each). I chose to use nine queuing delay samples for trend analysis in SyncTrend. This decision marks a tradeoff between detecting longer term trends and quickly making decisions. SyncTrend first gathers nine queuing delay samples and splits them into three groups of three samples, in the order of their arrival. The median of each of the three groups is computed, and the three medians are used to compute the trend. There are three tests that are performed on the medians:

- Pairwise Increasing Comparison Test (PCT_I).
- Pairwise Decreasing Comparison Test (PCT_D).
- Pairwise Difference Test (PDT).

Let the median computed queuing delays be $\{m_k, k = 1, 2, 3\}$. Let $I(X)$ be 1 if X is true and 0 if X is false. The PCT and PDT equations are as follows:

$$PCT_I = \frac{I(m_2 > m_1) + I(m_3 > m_2)}{2}$$

$$PCT_D = \frac{I(m_2 < m_1) + I(m_3 < m_2)}{2}$$

$$PDT = \frac{m_3 - m_1}{|m_2 - m_1| + |m_3 - m_2|}$$

PCT_I determines what percentage of the pairs of medians are increasing. With only three medians (and therefore, only two pairs of medians), there are only three possible values: 0, 0.5, and 1. Likewise, PCT_D determines what percentage of the pairs of medians are decreasing and can have the values 0, 0.5, or 1. PDT indicates the degree of the trend. If both pairs of medians are increasing, $PDT = 1$. If both pairs are decreasing, $PDT = -1$. Jain *et al.* performed an empirical study to determine that a strong increasing trend is indicated if $PCT_I > 0.55$ or $PDT > 0.4$ (in Jain's study PCT and PDT took on a wider range of values). I used these tests to also detect decreasing trends, and I make the claim that there is a strong decreasing trend if $PCT_D > 0.55$ or $PDT < -0.4$. If the medians have neither a strong increasing nor a strong decreasing trend, the trend is reported as undetermined.

SyncTrend computes a new trend every three queuing delay samples by replacing the three oldest samples with the three newest samples. SyncTrend returns the direction of the trend to the congestion reaction mechanism. The reaction mechanism could then choose to increase the flow's sending rate more aggressively than TCP Reno when the trend was decreasing.

Discussion

Figure 3.8 shows the SyncTrend congestion detection for one FTP flow with 53 ms base RTT. This figure is much like Figures 3.6 and 3.7, with the addition of the trend indications. (Note that since there is no congestion reaction, both the queuing delay at the router and the computed queuing delays here are exactly the same as that in Figures 3.6 and 3.7. In particular, some number of points are now marked to show congestion indications.) Whenever an increasing trend is detected, the computed queuing delay point is a circle. If a decreasing trend is detected, the computed queuing delay point is a star. If no trend could be determined or if there are not enough samples gathered to calculate the trend, the point remains a square. The boxed region on the graph is enlarged and pictured in Figure 3.9.

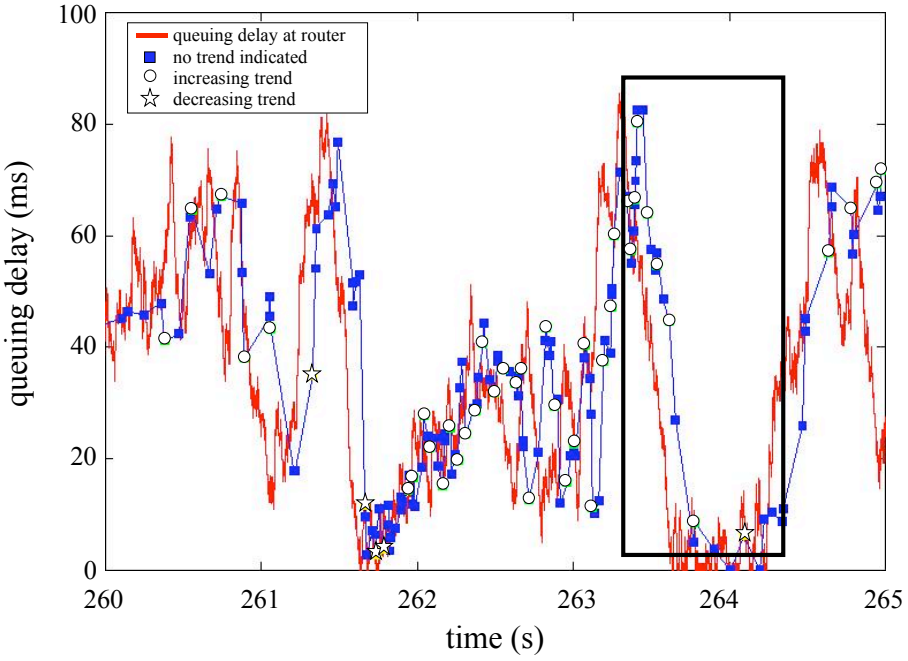

Figure 3.8: SyncTrend Congestion Detection

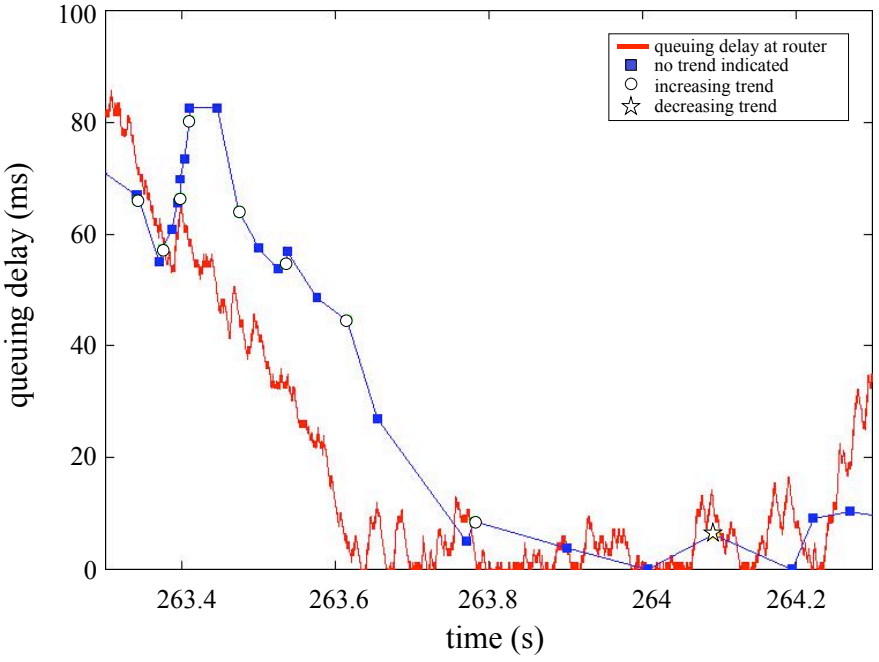

Figure 3.9: SyncTrend Congestion Detection - boxed region

Jain *et al.* used ten groups of ten one-way delay samples in their analysis. SyncTrend collects nine queuing delay samples (three groups of three) before evaluating the trend. This allows the algorithm to make decisions relatively quickly: at startup, after nine ACKs, and from there on, every three ACKs. This also results in the detection of small-scale trends, rather than longer-scale events. The algorithm might indicate small-scale trends where trend measurements fluctuate back and forth between increasing and decreasing. In Figure 3.9, the computed queuing delays are clearly decreasing, but because of the noise in the computed queuing delay signal, the trend analysis algorithm incorrectly detects that the trend is increasing. This could cause a congestion indication when congestion is actually subsiding. A tradeoff exists between collecting a larger number of samples to detect longer-scale trends and collecting few samples to be able to quickly make trend decisions. I chose to design the algorithm for quick detection.

3.5.4 SyncAvg

SyncAvg calculates the weighted average of the computed queuing delays, which it uses to detect congestion. Each time the average computed queuing delay exceeds a certain threshold, SyncAvg indicates that congestion is occurring. The averaging algorithm used by SyncAvg is based on the averaging algorithm used in RED. The weighted average queue length in RED is based on a single queue aggregating traffic from many flows. In contrast, SyncAvg smoothes the queuing delay estimates from only one flow. Therefore, the weighting factor SyncAvg uses should be much larger than RED's default of 1/512. SyncAvg uses the same smoothing factor as TCP's RTO estimator, which is 1/8. Both of these metrics (average queuing delay and RTT) are updated every time an ACK returns. The threshold set in SyncAvg is also based on RED. Christiansen *et al.* showed that using a minimum threshold of 30 ms was a good setting for RED [CJOS01]. In RED, the goal of the threshold is to keep the average queue size no larger than twice the threshold (*i.e.*, a target queuing delay of 60 ms). In SyncAvg, I set the default threshold to 30 ms so that, like RED, congestion will be indicated when the average queue size is greater than 30 ms.

Discussion

Figure 3.10 shows the SyncAvg congestion detection for one FTP flow with a 53 ms base RTT. In this figure, the black squares are the average queuing delays as computed at the sender. Previously all of the black squares of computed queuing delays were the same. Now the values are different (since they represent the average), but they are still computed at the same times as before. The average computed queuing delay tracks pretty well with the queuing delays reported at the router (the gray line) but with a noticeable lag.

In the boxed region (pictured in Figure 3.11), the lag between the actual queuing delay at the router and the average queuing delay computed at the sender is particularly striking.

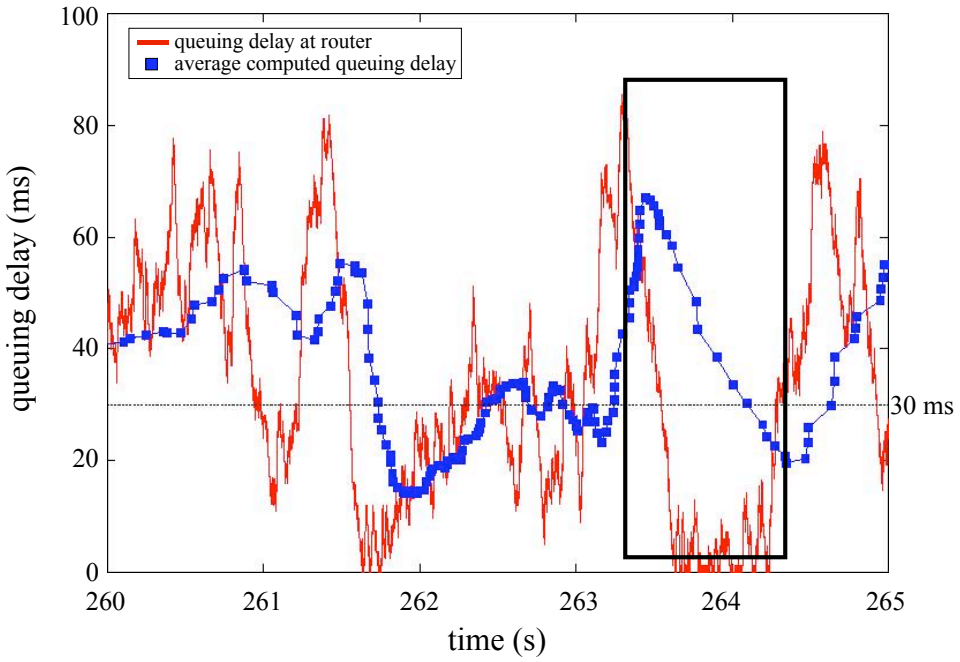

Figure 3.10: SyncAvg Congestion Detection

By the time the average computed queuing delay decreases below the threshold, it has been steadily decreasing for a significant amount of time and the actual queuing delay at the router is very low. Each black square represents the return of an ACK. Increases in the gaps between points are directly related to the gaps between the receipt of ACKs. The averaging weight is 1/8, so the last eight queuing delay samples have a large impact on the average. In the boxed region, there were a large number of high queuing delay samples. This can be seen when comparing the boxed region of SyncTrend (which shows computed queuing delays) and SyncAvg (which shows the average queuing delay) in Figure 3.12. When averaged, the large queuing delay samples, pictured on the left side of the SyncTrend graph, keep the average queuing delay high and increase the lag between the delay at the router and the computed delay. In fact, the average queuing delay value at a certain height on the y-axis does not directly correspond to values of queuing delay seen by the router as the computed queuing delay samples do in Figure 3.12a. When looking at the distance between the gray line and the black squares points in Figure 3.12a, the lag is directly related to the current RTT of the flow. When looking at the same distance in Figure 3.12b, the lag is related to the value of the previous computed queuing delays.

Another disadvantage of SyncAvg is that the fixed threshold value cannot adjust with increases in the total amount of queuing in the network (*e.g.*, with additional congested

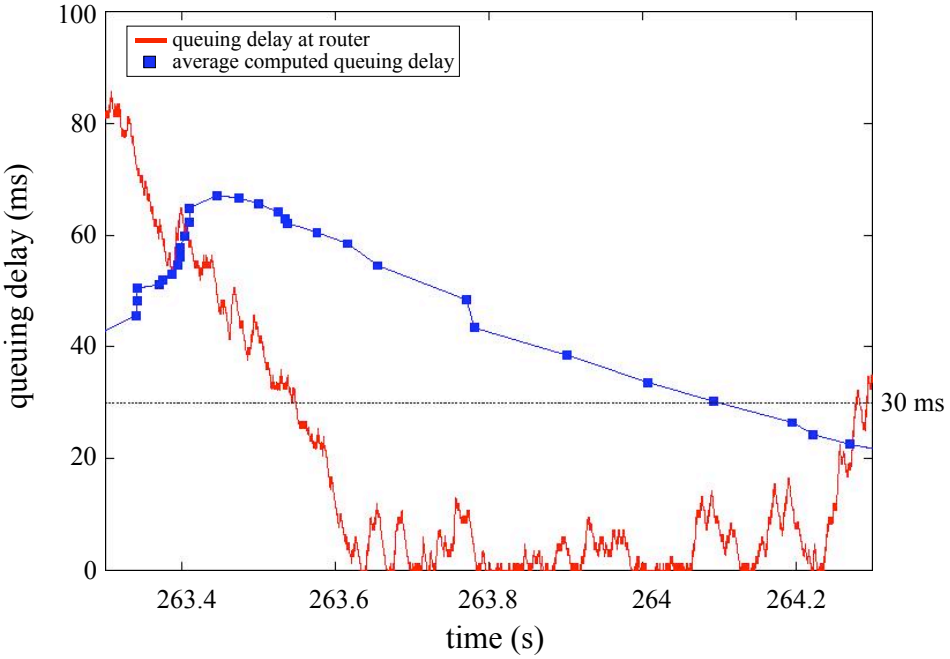

Figure 3.11: SyncAvg Congestion Detection - boxed region

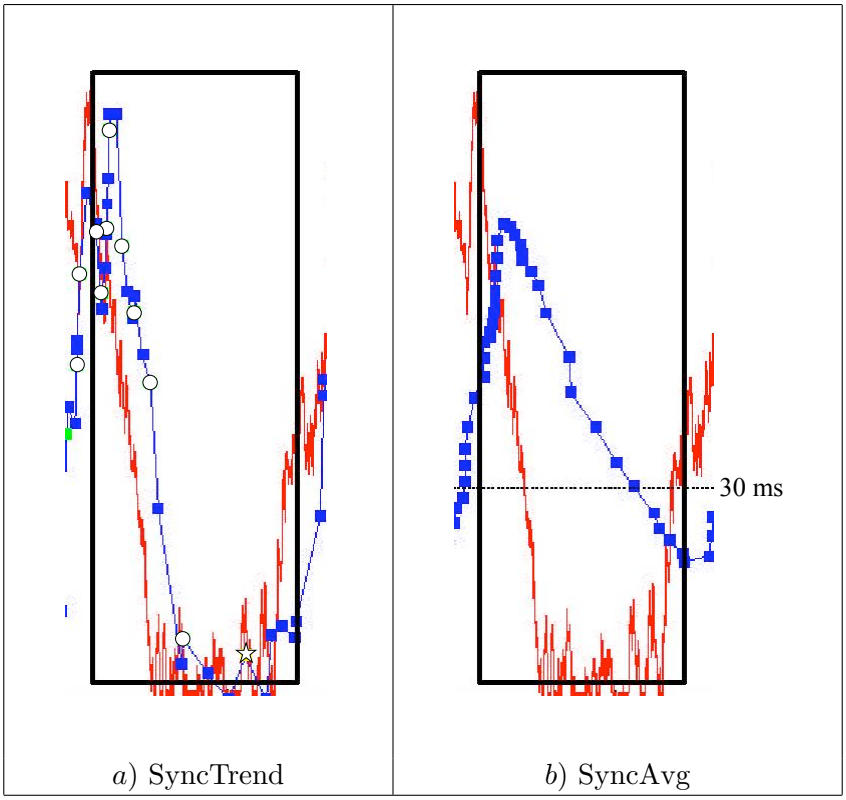

Figure 3.12: Boxed Regions from SyncTrend and SyncAvg

links). Since this method deals with end-to-end queuing delays, there is no one value that will be suitable for a threshold. This limitation is similar to that of SyncPMinOTT.

3.5.5 SyncMix

There were several problems discovered with the previous approaches to detecting congestion:

- SyncPQlen – The instantaneous computed queuing delay is too noisy.
- SyncPMinOTT – The instantaneous computed queuing delay is too noisy, and the threshold is not based on total amount of computed queuing delay in the network.
- SyncTrend – The instantaneous computed queuing delay is too noisy.
- SyncAvg – The average computed queuing delay introduces additional lag, and the threshold is not based on total amount of queuing delay in the network.

SyncMix uses a mixture of the previously described algorithms for congestion detection. SyncMix calculates the weighted average of the computed queuing delay and performs trend analysis on this average. SyncMix also compares the weighted average to a threshold based on an estimate of the maximum amount of queuing delay in the network to report the trend and where the average computed queuing delay lies.

Threshold

SyncMix uses a variation of the SyncPQlen threshold. Instead of the one 50% threshold, SyncMix divides the average computed queuing delay into four regions: 0-25%, 25-50%, 50-75%, and 75-100% of the maximum-observed queuing delay. Since SyncMix does not have a good estimate of the maximum queuing delay until a segment loss occurs, a different metric is used for congestion detection before the first segment loss. If there has been no segment loss and the maximum-observed queuing delay is less than 10 ms, SyncMix uses the absolute value of the percentage difference between the last two estimated queuing delays to determine the region. The 10 ms threshold is used to provide a transition to the original method of computing the region when there is no segment loss for an extended amount of time. If the maximum-observed queuing delay is at least 10 ms, then the four regions are at least 2.5 ms long. For example, if the maximum-observed queuing delay was 10 ms, then the 0-25% region would capture average queuing delays between 0-2.5 ms, the 25-50% region would capture average queuing delays between 2.5-5 ms, and so on. Letting maximum-observed queuing delays under 10 ms be the basis for setting the regions would cause very small regions at the beginning of a flow. Looking at the percentage difference between two consecutive queuing delay samples allows the algorithm to have an idea of the magnitude of change, while letting the trend analysis algorithm decide the direction of the change.

Number of Queuing Delay Samples	Weighting Factor
1	1
2	1
3	3/4
4	1/2
5	1/4
6 +	1/8

Table 3.1: SyncMix Average Queuing Delay Weighting Factors

Trend Analysis

SyncMix uses a simplified version of the trend analysis used in SyncTrend. Instead of computing the trend of the instantaneous queuing delay estimates, SyncMix computes the trend analysis on the weighted average of the queuing delays. SyncMix first gathers nine average queuing delay samples and splits them into three groups of three samples, in the order of their computation. The median, m_i, of each of the three groups is computed. Since there are only three medians, the trend is determined to be increasing if $m_0 < m_2$. Likewise, the trend is determined to be decreasing if $m_0 > m_2$. SyncMix computes a new trend every three ACKs by replacing the three oldest samples with the three newest samples for trend analysis. Due to the nine-sample trend analysis, no early congestion detection will occur if the server receives fewer than nine ACKs (*i.e.*, sends fewer than eighteen segments in the HTTP response, if delayed ACKs are turned on). More precisely, since OTTs are computed for every segment (including ACKs), each data segment from the client (*i.e.*, HTTP request segments) will carry an OTT sample that will be used in the server's trend analysis. So, the trend could be computed if the HTTP response were smaller than eighteen segments, but the early congestion detection would not have much of an effect.

Weighting Factor

SyncAvg used a weighting factor of 1/8 for the average computed queuing delay. This weighted averaging was inspired by RED. When RED averages its queue size for smoothing, it can assume that the process begins with an empty and idle queue. Therefore, the queue average is initialized to 0. Subsequent samples are weighted with a constant weight. Since SyncMix is not looking directly at a queue and a flow may begin when the network is in various states, it cannot assume that zero should be the basis for the average. SyncMix bootstraps the average with queuing delay samples rather than averaging them. Table 3.1 shows the weighting factors for the first few queuing delay samples. Each time a new queuing delay value is computed, the weighted average is calculated: $avg = (1 - w_q)avg + w_q(qdelay)$, where avg is the average and $qdelay$ is the newly-computed queuing delay. The weighting factors

Average Queuing Delay Trend	Average Queuing Delay as a % of the Maximum-Observed Queuing Delay
increasing	75-100%
increasing	50-75%
increasing	25-50%
increasing	0-25%
decreasing	75-100%
decreasing	50-75%
decreasing	25-50%
decreasing	0-25%

Table 3.2: SyncMix Return Values

given in Table 3.1 are used both at startup and whenever a timeout has occurred, which are times when the previous computed queuing delay average is considered to be out-of-date.

Congestion Indications

Each time an ACK is received, SyncMix either reports one of eight indications or that not enough samples have been gathered to compute the trend. The congestion indications are a combination of the trend direction and the region of the average computed queuing delay (Table 3.2).

Discussion

Figure 3.13 shows the congestion indications for SyncMix. The black squares are the average computed queuing delays. An increasing trend is indicated by a white circle, and a decreasing trend is indicated by a white star. The four average computed queuing delay regions are outlined with dotted lines at 25%, 50%, 75%, and 100% of the maximum-observed queuing delay.

Notice that the trend analysis works rather well for detecting trends in the average computed queuing delay. Particularly, in the boxed region of Figure 3.13 (enlarged in Figure 3.14), the detection of a decreasing trend indicates that congestion is abating even though there is still a large lag between the average queuing delay and the actual queue size.

3.5.6 Summary

Figure 3.15 shows the boxed regions from SyncTrend, SyncAvg, and SyncMix for comparison. SyncMix detects that congestion is abating (the trend is decreasing) earlier than either SyncTrend or SyncAvg. Due to noise, SyncTrend does not detect the trend decreasing until the computed queuing delay has been low for some time. SyncAvg detects that congestion is subsiding about the same time as SyncTrend when the average queuing delay passes below the

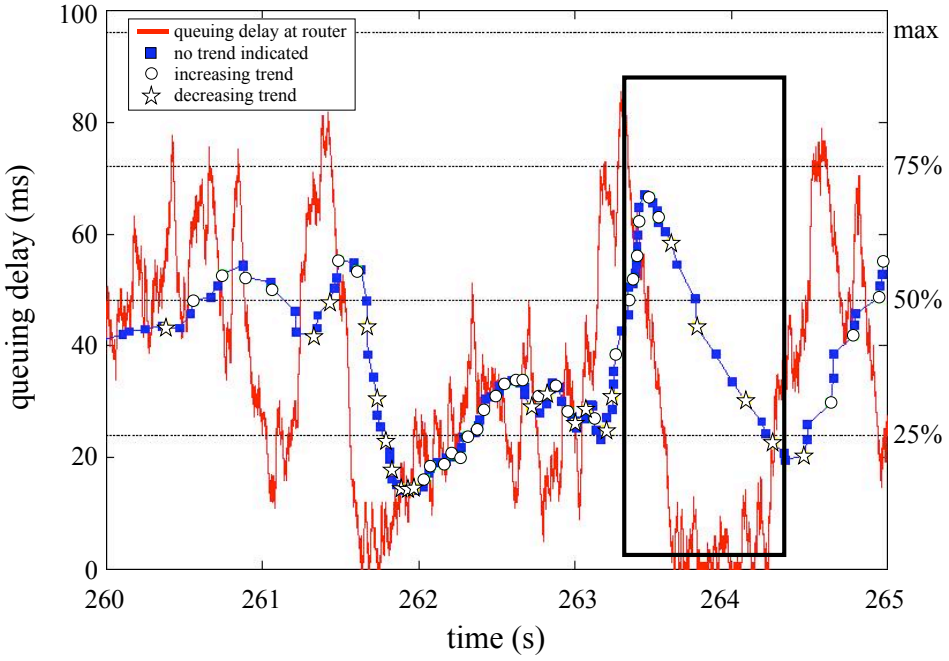

Figure 3.13: SyncMix Congestion Detection

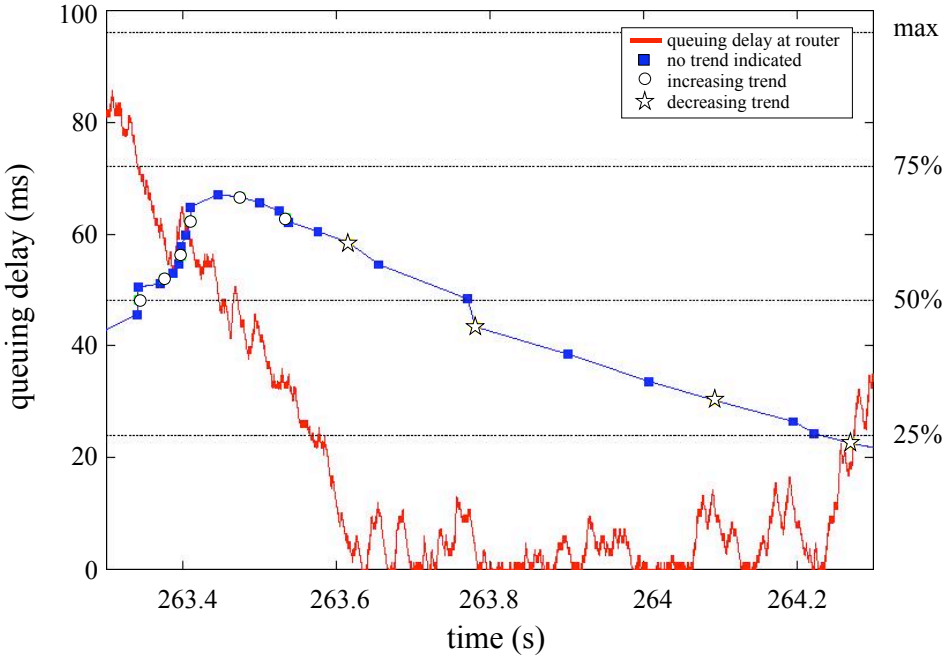

Figure 3.14: SyncMix Congestion Detection - boxed region

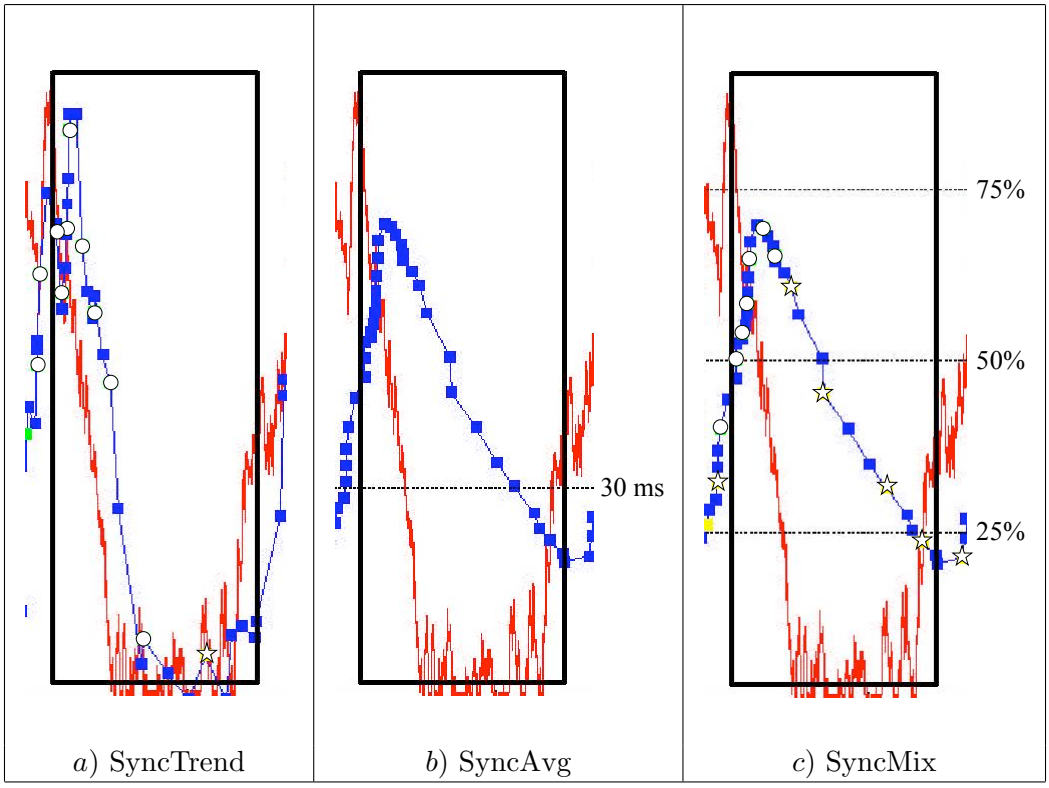

Figure 3.15: Boxed Regions from SyncTrend, SyncAvg, and SyncMix

30 ms threshold. For this reason, one of the congestion reaction mechanisms was specifically designed to be used with SyncMix and no congestion reaction mechanism will be evaluated with SyncTrend or SyncAvg as the congestion detection mechanism.

3.6 Congestion Reaction

Congestion detection is an important part of any congestion control algorithm. It is essential to be able to know when congestion is occurring so that the sender can react appropriately. The sender's reaction to congestion notification is also important. The reaction should be appropriate and should have the effect of reducing congestion and maintaining a low computed queuing delay. A Sync-TCP protocol is made up of a congestion detection mechanism and a congestion reaction mechanism. Since the congestion reaction mechanism cannot be evaluated apart from the congestion detection mechanism, I will present the evaluation of these methods in Chapter 4. I will consider two mechanisms for reaction to congestion indications, SyncPcwnd and SyncMixReact. SyncPcwnd is tuned for a binary congestion indicator and SyncMixReact for a more continuous signal. These algorithms will also be discussed in more detail in Chapter 4.

3.6.1 SyncPcwnd

SyncPcwnd reduces the congestion window by 50% whenever congestion is detected. Like the congestion window reduction policy in ECN, the reduction of *cwnd* is limited to at most once per RTT. The amount of the reduction is like that when an ECN notification is received or when TCP Reno detects segment loss through the receipt of three duplicate ACKs. When there is no congestion signal, SyncPcwnd increases *cwnd* exactly as TCP Reno does. SyncPcwnd can be used with any detection mechanism that gives a binary signal of congestion, such as SyncPQlen, SyncPMinOTT, SyncTrend, or SyncAvg.

3.6.2 SyncMixReact

SyncMixReact is specifically designed to operate with SyncMix congestion detection. SyncMixReact uses the additive-increase, multiplicative decrease (AIMD) congestion window adjustment algorithm but adjusts AIMD's α and β parameters according to the congestion state reported by SyncMix. Additive increase is defined as $w(t+1) = \alpha + w(t)$, where $w(t)$ is the size of the current congestion window in segments at time t and the time unit is one RTT. In TCP Reno's congestion avoidance, $\alpha = 1$. Multiplicative decrease is defined as $w(t+1) = \beta w(t)$. In TCP Reno, $\beta = 0.5$, with *cwnd* being set to 1/2 *cwnd* during fast retransmit. Following AIMD, when *cwnd* is to be increased, SyncMixReact incrementally increases *cwnd* for every ACK returned, and when *cwnd* is to be decreased, SyncMixReact makes the decrease immediately. Figure 3.16 shows the changes SyncMixReact makes to α and β based upon the result from the detection mechanism. The adjustments made to *cwnd* if the trend of average queuing delays is **increasing** are described below:

- If the average queuing delay is less than 25% of the maximum-observed queuing delay, *cwnd* is increased by one segment in one RTT (α is set to 1). This is the same increase that TCP Reno uses during congestion avoidance.
- If the average queuing delay is between 25-50% of the maximum-observed queuing delay, *cwnd* is decreased immediately by 10% (β is set to 0.9).
- If the average queuing delay is between 50-75% of the maximum-observed queuing delay, *cwnd* is decreased immediately by 25% (β is set to 0.75).
- If the average queuing delay is above 75% of the maximum-observed queuing delay, *cwnd* is decreased immediately by 50% (β is set to 0.5). This is the same decrease that TCP Reno would make if a segment loss were detected by the receipt of three duplicate ACKs and that an ECN-enabled TCP sender would make if an ACK were returned with the congestion-experienced bit set.

The adjustments made to *cwnd* if the trend of average queuing delays is **decreasing** are described below:

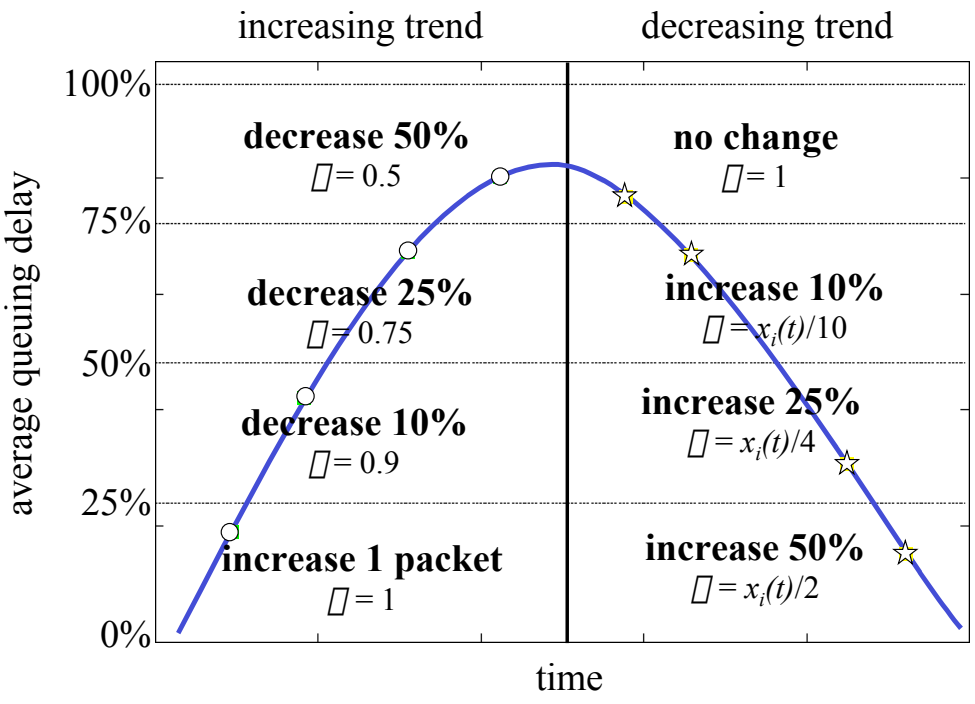

Figure 3.16: SyncMixReact Congestion Reactions

- If the average queuing delay is less than 25% of the maximum-observed queuing delay, *cwnd* is increased by 50% in one RTT (α is set to $x_i(t)/2$, where $x_i(t)$ is the value of *cwnd* at time t). If the congestion detection signal remained in this region, *cwnd* would be doubled in two RTTs. This increase is slower than TCP Reno's increase during slow start, but more aggressive than during congestion avoidance.
- If the average queuing delay is between 25-50% of the maximum-observed queuing delay, *cwnd* is increased by 25% in one RTT (α is set to $x_i(t)/4$).
- If the average queuing delay is between 50-75% of the maximum-observed queuing delay, *cwnd* is increased by 10% in one RTT (α is set to $x_i(t)/10$). This would typically be a very slow increase in the congestion window.
- If the average queuing delay is above 75% of the maximum-observed queuing delay, *cwnd* not adjusted (β is set to 1). The congestion window is not modified in this region, because if the trend begins to increase and the average queuing delay remains above 75% of the maximum-observed queuing delay, then *cwnd* would be reduced by 50%, a large decrease. If the average queuing delay decreased and the trend remained decreasing, then congestion is subsiding.

The additional congestion window decreases in SyncMixReact (as compared to TCP Reno) when the average queuing delay trend is increasing are balanced by the more aggressive congestion window increases when the average queuing delay trend is decreasing.

The congestion detection mechanism in SyncMix will return a new congestion detection result every three ACKs (according to the trend analysis algorithm), causing SyncMixReact to adjust the congestion window.

3.7 Summary

In this chapter, I described the development of Sync-TCP, a family of congestion detection and reaction mechanisms based on the use of OTTs. The OTTs are obtained via the exchange of timestamps between computers with synchronized clocks. I compared five different methods of congestion detection that use forward-path OTTs and computed queuing delays to detect when network queues are increasing and congestion is occurring. I also looked at two methods for reacting to the congestion indications provided by the congestion detection methods.

In Chapter 4, I will evaluate the performance of two Sync-TCP congestion control algorithms as compared with standards-track TCP congestion control algorithms. I chose to look at the combination of the SyncPQlen congestion detection mechanism and the SyncPcwnd congestion reaction mechanism. I will refer to this combination as Sync-TCP(Pcwnd). The second Sync-TCP congestion control mechanism, called Sync-TCP(MixReact), is a combination of the SyncMix congestion detection mechanism and the SyncMixReact congestion reaction mechanism.