5 Modeling Alternate Congestion-Control Mechanisms

The fundamental design of the Internet protocol suite [3] assumes that network elements, such as routers, are relatively simple – receiving, buffering and forwarding packets among connected links and dropping packets when buffers are insufficient to accommodate arriving packets. Under this assumption, computers connected to the Internet must implement decision algorithms to pace the rate at which packets are injected into the network. Such decision algorithms, known typically as congestion-control mechanisms, are implemented independently by each source with the goal of achieving a satisfactory network-wide outcome and a fair distribution of resources to all active sources. In the current state-of-the-practice, congestion-control mechanisms are implemented as part of the transmission-control protocol (TCP) [9-11] that operates within every computer attached to the global Internet. While TCP congestion-control procedures have proven quite successful [2] at achieving desired properties, numerous researchers [47-52, 64a] have postulated potential changes in relationships among bandwidth and propagation delay as the speed of network links increases toward 10s and 100s of gigabits per second (Gbps). Under such envisioned circumstances, researchers predict that TCP congestion-control procedures will prove insufficient, leading to substantial underutilization in network resources and preventing end users from achieving high transfer rates, potentially reaching or surpassing a Gbps. These predictions have stimulated researchers to propose alternate congestion-control mechanisms [53-62] that might achieve higher network utilization and better user performance as network speeds increase.

As part of proposing alternate congestion-control mechanisms, researchers typically model, simulate and implement prototypes and then explore how candidate congestion-control mechanisms might affect the Internet and its users. Given the increasing number of proposals, interest is growing [63-68] in developing procedures to fairly and effectively evaluate the properties of the proposals. A similar motive underlies the work reported in the current study; however, our approach is to simulate proposed congestion-control mechanisms within a reasonably large network that can support on $O(10^5)$ simultaneously active flows. To illustrate our methodology, we have chosen to investigate six proposed alternate congestion-control mechanisms [53-55, 59, 61-62], which have been simulated and studied empirically at smaller scales.

In this section of our study, we introduce the basic concepts underlying TCP congestion control and we explain the changes to those procedures that are proposed by six different research teams. Other research teams [56-58, 60] have also proposed changes to TCP congestion-control procedures. We chose to examine only six proposals in order to limit our study, which focuses on a methodology for conducting evaluations rather than on an exhaustive consideration of all published proposals. We selected five specific proposals because a recent study [67] reports empirical results from prototype implementations included within Linux. This enables us to validate our simulations of the proposals against the reported empirical measurements. We chose a sixth alternate congestion-control mechanism, Compound TCP [59], or CTCP, because it has been proposed by researchers at Microsoft and, thus, may be available in the future within a large number of computers attached to the Internet. Further, there are some recent empirical results [66] against which we can validate our model of CTCP. The methodology we define in our study can be applied to additional proposals for alternate congestion-control mechanisms, should there be sufficient interest.

The remainder of this section is organized into five major topics. We begin (in Sec. 5.1) by introducing TCP congestion control and then (in Sec. 5.2) define the procedures adopted by six, selected, alternate proposals for congestion control in the Internet. Next (in Sec. 5.3), we describe how we model congestion-control procedures within MesoNetHS. In Sec. 5.4, we describe the test configuration used to verify that we correctly model each congestion-control mechanism. We then present simulation results showing evolution of congestion-windows over time for each congestion-control mechanism that we model. The information reported in this section sets the stage for us to consider (in Secs. 6 through 9) whether proposed alternate congestion-control procedures might change macroscopic network behavior or user experience.

5.1 TCP Congestion Control

A typical TCP flow evolves through three phases: connection, transfer and graceful close. For purposes of congestion control, we limit our discussion to the connection phase and the transfer phase. Fig. 5-1 shows a high-level view of these two phases. During the connection phase, a source attempts to establish contact with an intended receiver. Inability to establish contact results in a connection failure, which prevents data from flowing between source and receiver; thus, connection-establishment procedures provide one form of congestion-control implemented by TCP. During the transfer phase, a source sends data (in the form of segments) on the flow until the required amount has been received successfully. A receiver signals receipt of data segments by sending acknowledgments (ACKs) to the source. By sending duplicate acknowledgments, a receiver may also indicate failure to receive specific segments, which the source must then retransmit. Further, a sender may fail to receive acknowledgments, which requires the sender to raise a timeout and to retransmit unacknowledged data. During the transfer phase, congestion-control procedures determine when a source may send data segments.

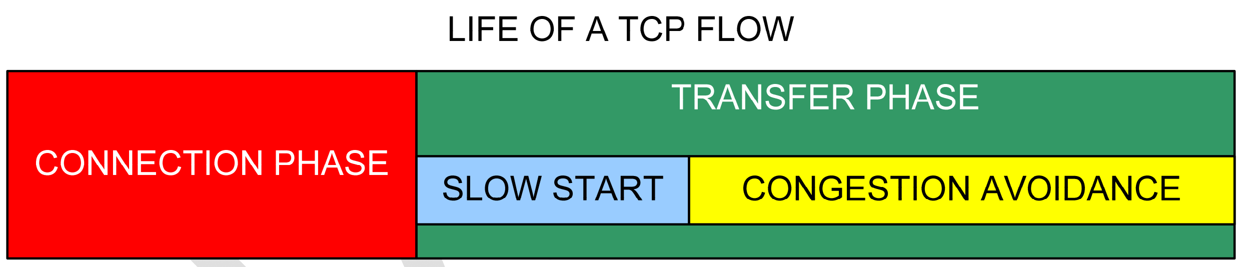

Figure 5-1. Main Phases and Congestion-Control Procedures in the Life of a TCP Flow

The transfer phase includes two congestion-control regimes: slow start and congestion avoidance. Slow start occurs when a source is uncertain about the transmission rate that might be achieved on a TCP flow. For this reason, after establishing a connection, a source begins the transfer phase using slow-start procedures. A source might also adopt slow-start procedures after a timeout. Slow-start procedures begin by sending data at a slow rate but then increase that rate quickly (e.g., exponentially) as ACKs arrive from the receiver. Once a source has a better idea about an achievable transmission rate, slow-start procedures are abandoned in favor of congestion-avoidance procedures, which attempt to increase the sending rate more slowly (e.g., linearly). Thus, congestion-control procedures during the transfer phase have three basic purposes: (1) find an achievable transfer rate on a flow; (2) maintain the achievable transfer rate if possible; (3) attempt to increase the achievable transfer rate. Proposals for revising TCP congestion-control procedures target mainly congestion-avoidance procedures within the transfer phase.

We model TCP flows as a connection phase, followed by a transfer phase. We adopt identical connection-establishment procedures regardless of the specific congestion-control mechanism being simulated. Within the transfer phase, we model congestion-control procedures as an initial slow-start period, followed by congestion avoidance. In addition, we employ slow-start procedures after a timeout. Generally, no matter which congestion-control mechanism we simulate, we adopt identical slow-start procedures. We alter congestion-avoidance procedures, which respond to acknowledgments, losses and timeouts, as required by the various congestion-control mechanisms that we simulate.

Below, we describe the procedures used in our model for connection establishment and slow start. Then we outline our model of standard (i.e., Reno) TCP congestion-avoidance procedures. Subsequently, in Sec. 5.2, we describe our model of congestion-avoidance procedures for each of the six alternate congestion-control mechanisms that we simulate.

5.1.1 Connection Phase

Typically, establishing a TCP connection (or flow) requires a three-way handshake involving a connection-request (SYN) segment sent from a source to a receiver, followed by a connection-confirm (SYN+ACK) segment sent from a receiver to a source and then ending with an ACK segment sent from a source to a receiver. Our model simulates connection-establishment as a two-way handshake – SYN followed by SYN+ACK – because TCP allows ACKs to be piggybacked on data (DT) segments. This implies that the first DT sent from a source to receiver during the transfer phase may also double as the final segment of connection establishment.

Of course, congestion may lead to lost SYN or SYN+ACK segments; thus, a source must implement error detection and recovery procedures, which typically involve retransmitting SYN segments. In our model, we simulate such procedures, while adopting default parameters typically used in TCP implementations within the Microsoft Windows® family of operating systems. We take this decision because many computers connected to the Internet use the Microsoft implementation of TCP. Fig. 5-2 illustrates schematically our connection-establishment model: showing one possible scenario leading to connection failure.

During connection establishment a source first sends a SYN segment and waits for a period of time (3 s in Fig. 5-2) for a SYN+ACK. If no SYN+ACK segment arrives, the source sends a second SYN and waits for a longer period of time (6 s in Fig. 5-2). If no SYN+ACK segment arrives, the source sends a third SYN and waits for a longer period of time (12 s in Fig. 5-2). This cycle repeats until the maximum number of SYNs (3 in Fig. 5-2) has been sent or until a SYN+ACK segment arrives. If no SYN+ACK segment arrives after the maximum number of SYNs is sent, then (as shown in Fig. 5-2) TCP raises a connection-failure signal. For the parameters we adopt, connection failures occur after 21 s without receipt of a SYN+ACK after the first SYN is sent. Arrival of a SYN+ACK segment during this timeout period (as illustrated in Fig. 5-3) results in successful connection establishment. Fig. 5-2 and Fig. 5-3 show several losses that can require retransmission of SYN and SYN+ACK segments.

We include connection-establishment procedures within our model because they provide one means to limit congestion when attempting to exchange data over severely congested paths within the Internet. We found that failure to model connection-establishment procedures leads to an unrealistic buildup of excessive congestion under model configurations that induce congestion at various points within a simulated topology.

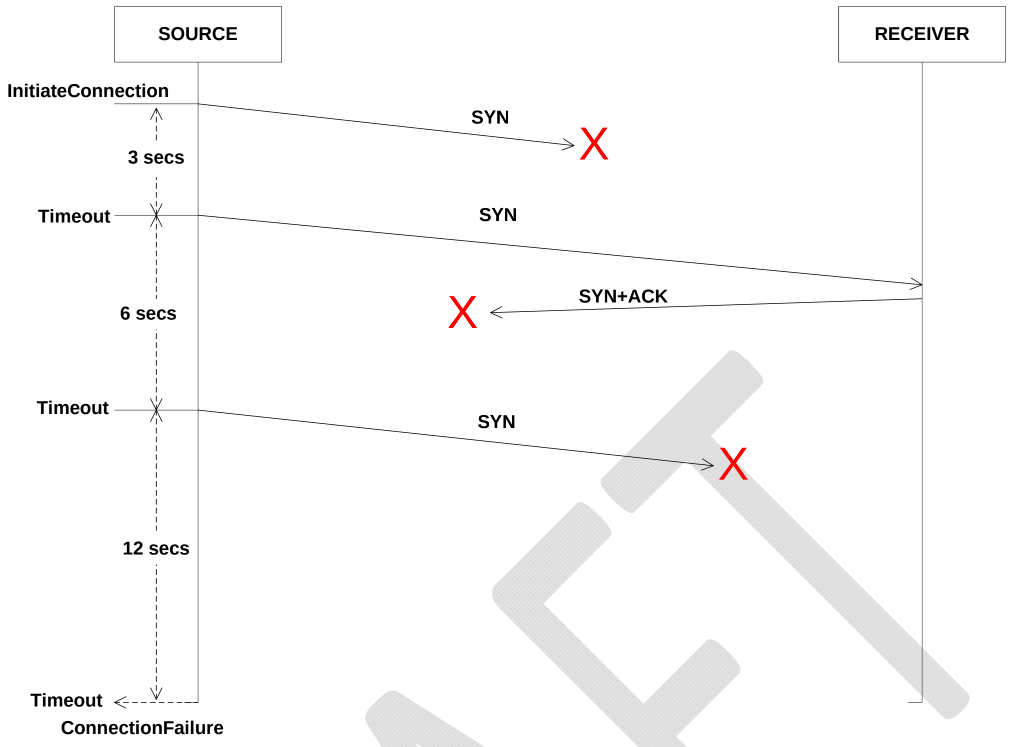

Figure 5-2. TCP Connection-Establishment Procedures Leading to Connection Failure

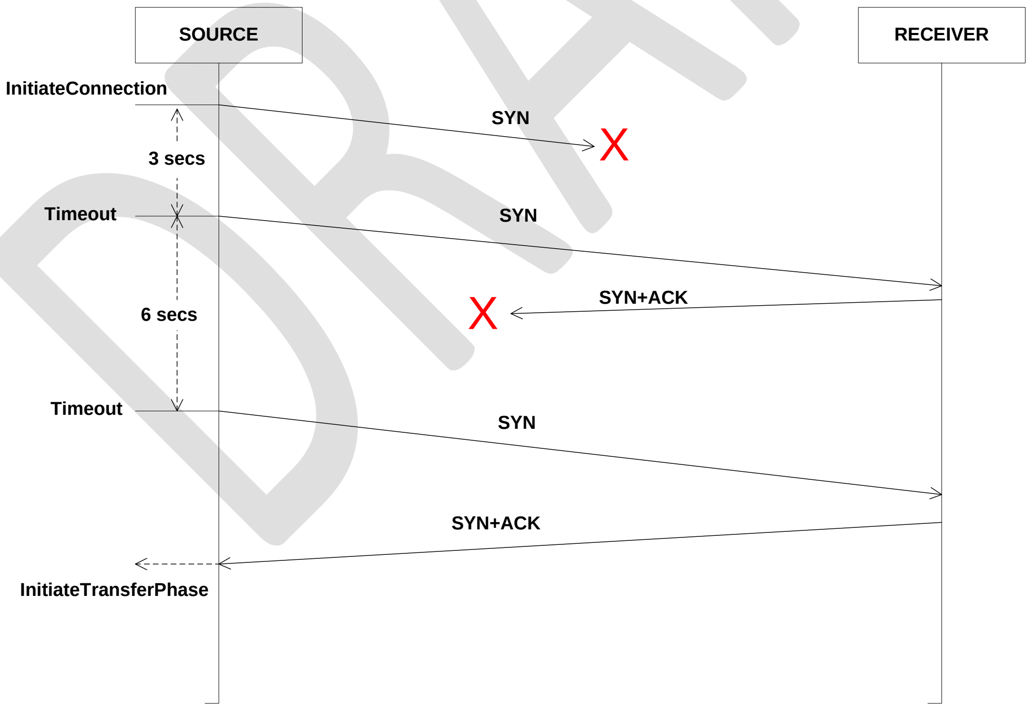

Figure 5-3. TCP Connection-Establishment Procedures Leading to Initiation of the Transfer Phase

Table 5-1. Definition of Symbols Used to Model Connection-Establishment Procedures

Symbol	Definition
syn_{INT}	Timeout interval for initial SYN
syn_{MAX}	Maximum number of SYNs to send
syn_{SENT}	Number of SYNs that have been sent
syn_{TO}	Timeout for current SYN
time	Current time

Our model of congestion-establishment procedures uses the variables identified and defined in Table 5-1. Initiation of the connection phase entails the following steps by a source.

```
InitiateConnection ≡ | $syn_{MAX} \leftarrow 3$
                     | $syn_{INT} \leftarrow 3\ s$
                     | $syn_{TO} \leftarrow \mathbf{time} + syn_{INT}$
                     | $syn_{SENT} \leftarrow 1$
                     | send(SYN)
```
(1)

Upon a timeout, a source implements the following procedures, which amount to an exponential back-off in the timeout period until the maximum number of SYN segments have been sent.

```
Timeout ≡ | if $syn_{SENT} < syn_{MAX}$
          |     | $syn_{INT} \leftarrow 2 \times syn_{INT}$
          |     | $syn_{TO} \leftarrow \mathbf{time} + syn_{INT}$
          |     | $syn_{SENT} \leftarrow syn_{SENT} + 1$
          |     | send(SYN)
          | signal(ConnectionFailure)   otherwise
```
(2)

If a SYN+ACK segment is received prior to connection failure, then the source initiates the transfer phase, which is discussed next in two parts: slow start and congestion avoidance.

$$\mathbf{SYN + ACKreceived} \equiv \mathbf{InitiateTransferPhase} \tag{3}$$

5.1.2 Transfer Phase – Slow Start

During the transfer phase a TCP flow establishes and adjusts a congestion-window (*cwnd*) and slow-start threshold (*sst*), which requires introducing and defining some additional symbols, as shown in Table 5-2. Our model permits two forms of slow-start: (a) standard TCP slow start or (b) limited slow start [8]. We explain each of these in turn.

Table 5-2. Definition of Symbols Used to Model Slow-Start Procedures

Symbol	Definition
cwnd	Current congestion window
$cwnd_{INT}$	Initial congestion window (we use $cwnd_{INT} = 2$)
sst	Current slow-start threshold
sst_{MAX}	Threshold to switch from exponential to logarithmic increase (varies with experiment)
sst_{INT}	Threshold to terminate initial slow start (varies with experiment)

5.1.2.1 Standard Slow Start. Upon entering slow start, a TCP flow adopts a small value ($cwnd_{INT}$) for the congestion window (*cwnd*). During standard slow start, a flow then increases *cwnd* exponentially as ACKs are received until reaching an initial slow-start threshold (sst_{INT}). After the congestion window reaches sst_{INT} (or upon a loss) the flow enters a congestion-avoidance regime. In our model, a flow initiates slow-start with the following procedures.

$$\textbf{InitiateTransferPhase} \equiv \left| \begin{array}{l} cwnd \leftarrow cwnd_{INT} \\ sst \leftarrow sst_{INT} \end{array} \right. \quad (3)$$

5.1.2.2 Limited Slow Start. During limited slow start, a flow increases *cwnd* exponentially as ACKs are received until reaching a maximum slow-start threshold (sst_{MAX}). After the congestion window reaches sst_{MAX} the flow increases *cwnd* logarithmically with each ACK received until reaching sst_{INT}. After the congestion window reaches sst_{INT} (or upon a loss) the flow enters a congestion-avoidance regime.

Our model distinguishes standard slow start from limited slow start based on the relationship between sst_{MAX} and sst_{INT}. Limited slow-start procedures are used if $sst_{MAX} < sst_{INT}$. Otherwise, standard slow-start procedures are used. These conventions are specified through the combination of configuration parameters for sst_{MAX} and sst_{INT}, initialization procedures (3) and the following procedures upon receiving an ACK.

$$\textbf{ACK} \wedge (cwnd < sst) \equiv \left| \begin{array}{l} cwnd \leftarrow cwnd + 1 \quad if \quad cwnd < sst_{MAX} \\ cwnd \leftarrow cwnd + \dfrac{1}{\left(\dfrac{cwnd}{0.5 \times sst_{MAX}}\right)} \quad otherwise \end{array} \right. \quad (4)$$

5.1.2.3 Setting Slow-Start Threshold. The literature indicates no widespread agreement on what value should be chosen for sst_{INT}. Some authors [7] recommend setting sst_{INT} to an arbitrarily large value, which implies that initial slow start will continue until a flow experiences its first loss or timeout. Other authors [11] recommend setting sst_{INT} to a small value, which means that slow start might terminate before a flow has determined its available bandwidth; thus, the maximum available bandwidth might not be achieved before the flow terminates (depending on the number of data segments in the flow). Others [4] suggest setting sst_{INT} selectively based upon properties maintained by the device driver for the network interface. In addition, some [5] suggest using the advertised window (*awnd*) returned from a receiver to set sst_{INT}.

Given such varying suggestions, we included sst_{INT} as a configuration parameter of our model. This allows sst_{INT} to be set to large and small values, as desired. Our model does not support setting sst_{INT} variably based on properties of the network interface. Our model does not simulate a receiver's *awnd*; thus, setting sst_{INT} based on that value is not supported.

5.1.3 Transfer Phase – Congestion Avoidance

In our model of TCP Reno, congestion avoidance, which begins once $cwnd \geq sst$, increases the congestion window linearly, at the rate of one per round-trip time. The increase accrues fractionally as ACKs are received. When the receiver signals a loss, the congestion window is cut in half. Upon a timeout, the slow-start threshold is set to half the congestion window and the congestion window is set to its initial value. Below we specify the procedures used by a source to increase *cwnd* on receipt of each ACK, to decrease *cwnd* upon each signaled loss and to decrease *cwnd* and *sst* at each timeout.

5.1.3.1 Increase Congestion Window after Acknowledgment. For each ACK received within each round-trip time up until a loss or timeout, a TCP source increases its congestion window by a fraction, using the following procedures.

$$\mathbf{ACK} \equiv cwnd \leftarrow cwnd + \frac{1}{cwnd} \qquad (5)$$

An actual TCP implementation will use *cwnd* as part of a decision function to determine its send window (*swnd*). The decision function is *swnd* = **min**(*cwnd*, *awnd*). Since our model does not simulate *awnd*, a source's *swnd* is always equal to its *cwnd*. This means the sending rate of sources in our model will be constrained by network congestion rather than by local policies within receivers.

5.1.3.2 Decrease Congestion Window after Signaled Loss. When a receiver signals a loss within a given round-trip time, a TCP source reduces its *cwnd* by half. In real TCP implementations, a loss is signaled by receiving three consecutive, duplicate ACKs. This convention was designed to accommodate cases where DTs are delivered out of order by the network. Reordering DTs can lead to duplicate ACKs even though a DT was not lost; thus, a TCP source defers any decision that a DT was lost until three duplicate ACKs arrive in sequence. Modern router vendors strive to ensure that packets are not reordered on a given flow [42-44]; however, some researchers [39-40, 46] have reported cases where packets are reordered within a router. Our simulation model permits packets to be lost, but not reordered. For this reason, our sources detect explicit losses upon receiving a single duplicate ACK, which we model as a negative acknowledgment (NAK).

Sources in our model reduce a flow's *cwnd* once in a round-trip time when a loss is signaled by a receiver. The reduction rule follows.

$$\textbf{Loss} \equiv \left| \begin{array}{l} cwnd \leftarrow \frac{cwnd}{2} \\ sst \leftarrow cwnd \end{array} \right. \tag{6}$$

The *sst* is reset to the *cwnd* so that the flow continues in congestion avoidance rather than reentering slow start.

5.1.3.3 Decrease Slow-Start Threshold and Reset Congestion Window after Timeout. A source encounters a timeout when no ACKs or NAKs have been received on a flow for the duration of a retransmission timeout (RTO). The RTO for a flow is maintained to be no less than twice and no greater than 32 times round-trip propagation delay between a source and receiver. Regardless of congestion-control mechanism, our model implements a single set of procedures for maintaining and increasing RTO. Upon initiation of a flow's transfer phase, RTO is set to twice the round-trip propagation delay. Upon receipt of an ACK or NAK, a flow's RTO is set to the maximum of 1.5 times the measured, smoothed round-trip time (SRTT) or twice the round-trip propagation delay. With each timeout the RTO is doubled, which leads to an exponential back-off, up to the maximum RTO.

Occurrence of a timeout indicates a significant interruption in the path between a source and receiver. For this reason, our model adopts a conservative strategy in responding to timeouts. The *sst* is reduced using the reduction rules required by the specific congestion-control mechanism in use for the flow. In addition, the *cwnd* is reset to its initial value. This implies that the flow will reenter slow start and then use rapid increase procedures until $cwnd \geq sst$. Subsequently, the flow returns to congestion-avoidance procedures. Our model for TCP Reno uses the following timeout procedures.

$$\textbf{Timeout} \equiv \left| \begin{array}{l} sst \leftarrow \textbf{max}\left(\frac{cwnd}{2}, cwnd_{INT}\right) \\ cwnd \leftarrow cwnd_{INT} \end{array} \right. \tag{7}$$

5.1.3.4 Combined Effects of Slow Start and Congestion Avoidance. To appreciate the combined effects of slow start and congestion avoidance, as implemented in our model of TCP Reno, we consider some schematic graphs of the time evolution of the *cwnd* for hypothetical flows. Fig. 5-4 depicts changes in *cwnd* assuming the use of standard slow start, with both sst_{MAX} and sst_{INT} set to 128. The *cwnd* increases exponentially in slow start until it reaches sst_{INT}. Subsequently, congestion avoidance commences and the *cwnd* increases linearly. Just after the *cwnd* reaches 150 (time 30), a loss occurs and the *cwnd* is reduced to 75. The *cwnd* then increases linearly until it reaches 100 (time 55). At about time 63 the source experiences a timeout and the *cwnd* is reduced to its initial value (two). At the same time *sst* is set to 50 (half the value of the *cwnd* when the timeout occurred). As ACKs resume the *cwnd* increases exponentially (in slow start) until 50 (the value of *sst*) after which the flow returns to congestion avoidance and the *cwnd* increases linearly.

Fig. 5-5 displays the same scenario as Fig. 5-4, with the exception that standard slow start is replaced by limited slow start, with $sst_{MAX} = 16$ and $sst_{INT} = 128$. Here, the *cwnd* increases exponentially until reaching 16 and then increases logarithmically until reaching 128. Subsequently, *cwnd* increases linearly in congestion avoidance until a loss occurs, just after the *cwnd* reaches 150 (about time 42). After the loss, the *cwnd* drops in half (to 75) and then increases linearly until reaching 100. At about time 80 the source experiences a timeout and the *cwnd* is reduced to two, while *sst* is reset to 50 (half the value of the *cwnd* when the timeout occurred). When ACKs resume the *cwnd* increases exponentially to 16 and then logarithmically to 50 (the value of *sst*) from which the *cwnd* increases linearly as the flow returns to congestion avoidance.

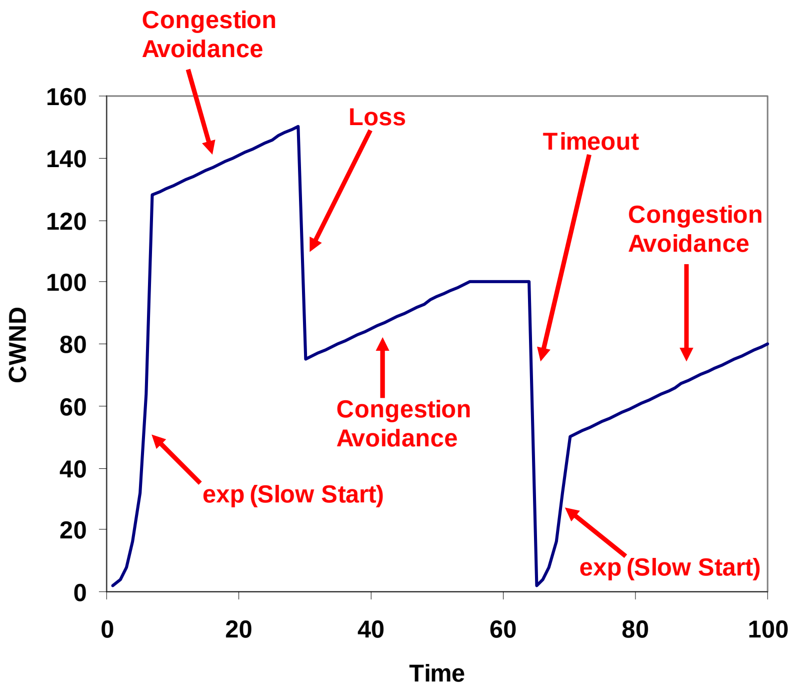

Figure 5-4. Sample Evolution of Congestion Window under Standard Slow Start and Congestion Avoidance

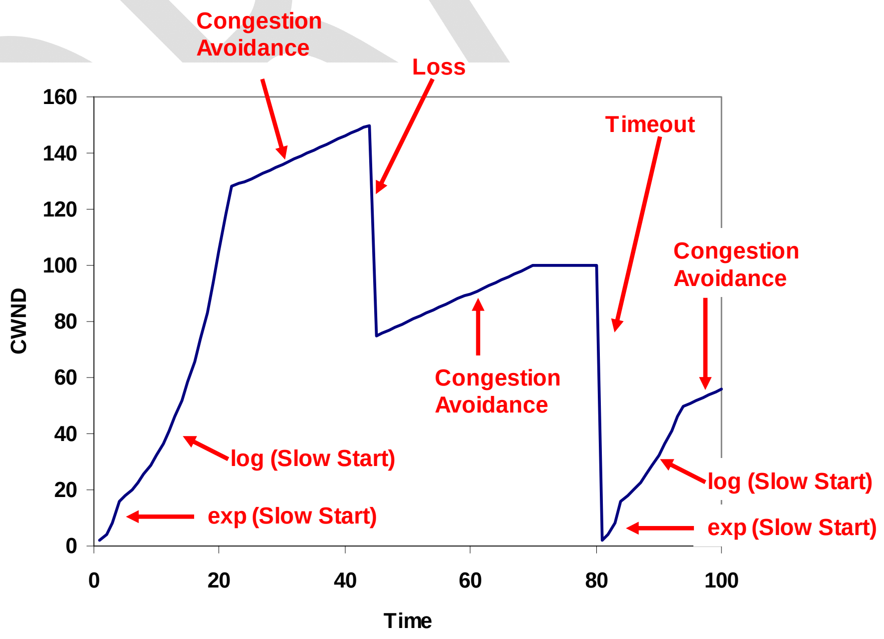

Figure 5-5. Sample Evolution of Congestion Window under Limited Slow Start and Congestion Avoidance

5.2 Congestion-Avoidance Procedures for Six Alternate Congestion-Control Mechanisms

In this section, we describe congestion-avoidance procedures defined by six proposed alternate congestion-control mechanisms: binary increase congestion control (BIC) [62], Compound TCP (CTCP) [59], Fast Active Queue Management (AQM) Scalable TCP (FAST) [61], High-Speed TCP (HSTCP) [53], Hamilton TCP (H-TCP) [55] and Scalable TCP [54]. For each congestion-control mechanism, we specify increase procedures taken upon receipt of an ACK, decrease procedures used upon explicit notification of a loss and timeout procedures. In addition, three of the congestion-control mechanisms (CTCP, FAST and H-TCP) require periodic actions, which we also specify.

All but two (FAST and H-TCP) of the alternate congestion-control mechanisms define a threshold, such that when the congestion window is below the threshold then normal TCP congestion-avoidance procedures are used. This means that the alternate congestion-avoidance procedures will be invoked only when a flow's congestion window surpasses the threshold. Further, whenever a congestion window passes and then falls below the threshold, normal TCP congestion-avoidance procedures will be resumed. Alternate procedures will be reactivated when the congestion window again passes the threshold. H-TCP also uses an activation threshold; however, the H-TCP threshold is defined in terms of elapsed time since the most recent loss on a flow. FAST does not use a threshold; thus, the alternate congestion-avoidance procedures are always applied for FAST flows. As appropriate, we specify the threshold values associated with each alternate congestion-control mechanism.

5.2.1 BIC

The congestion-avoidance procedures used by BIC aim to make aggressive increases in the *cwnd* when the current *cwnd* is far from a target *cwnd* and smaller increases as the current *cwnd* nears the target. BIC determines the target *cwnd* by conducting a binary search within some range around the current *cwnd*. When the target window falls beyond the search range, BIC increases the *cwnd* additively in a fixed increment and then reinitiates the binary search within the new range. Implementing this behavior requires rather complex logic; thus, BIC procedures for congestion-avoidance tend to be somewhat elaborate. The resulting *cwnd* evolution for BIC reflects its complexity – reproducing a function that appears to change in a pattern resembling a human heartbeat.

Specifying BIC congestion-avoidance procedures requires some additional symbols, as identified and defined in Table 5-3. Where applicable, the table denotes parameter values used within our model. We adopt parameter values that match the default values reported in an empirical study [67] of prototype implementations for alternate congestion-control mechanisms.

A given BIC flow uses normal TCP congestion-avoidance procedures or the alternate BIC procedures, defined below, depending on the relationship between *cwnd* size and a low-window parameter ($LW_B = 14$).

$$\textbf{SelectProcedures} \equiv \left| \begin{array}{ll} \textbf{TCPcongestionAvoidance} & \textit{if } \ cwnd < LW_B \\ \textbf{BICcongestionAvoidance} & \textit{otherwise} \end{array} \right. \qquad (8)$$

Table 5-3. Symbols and Definitions Used to Model BIC Congestion-Avoidance Procedures

Symbol	Definition
β_B	Multiplicative back-off factor for loss (BIC parameter β = 0.8)
B_B	Parameter for computing increase in congestion window (BIC parameter B = 4)
Δ_B	Candidate numerator for increase in congestion window
LW_B	Low-window threshold (LW_B = 14) for applying BIC procedures
MIN_B	Variable to track minimum value for computing BIC target window
MAX_B	Variable to track maximum value for computing BIC target window
$PREV_B$	Variable to track previous MAX_B
σ_B	Parameter for computing increase in congestion window (BIC parameter σ = 20)
$SMAX_B$	Threshold to begin rapid increase in congestion window (BIC parameter S_{max} = 32)
SS_B	Boolean indicating whether the flow is in BIC slow start (**true**) or not (**false**)
SST_B	BIC slow-start target
SSW_B	BIC slow-start congestion window
TGT_B	BIC target window (BIC variable w_1)

5.2.1.1 Increase Procedures. The window-increase procedures (9) used by BIC differ depending upon whether the current *cwnd* is below or beyond a previously determined maximum *cwnd*. As the *cwnd* passes the previously determined maximum, BIC invokes slow-start procedures that increase the *cwnd* until the *cwnd* approaches a new maximum (set to twice the previous maximum). As the *cwnd* nears the new maximum, slow-start procedures are abandoned and the *cwnd* is increased using a binary search. Once the new maximum is exceeded, then BIC reenters its slow-start procedures. Thus, during congestion avoidance, BIC increase procedures alternate between BIC slow-start and binary search.

During the BIC binary search, the *cwnd* is increased more quickly when further from the current midpoint and less quickly as it nears the midpoint. The rules controlling the increase pattern are encoded within a function (10).

5.2.1.2 Decrease Procedures. Upon notification of a loss (11), BIC sets the maximum *cwnd* for the binary search to the current *cwnd* and then multiplicatively decreases the congestion window. If the loss followed closely behind a previous loss, then BIC also multiplicatively decreases the maximum *cwnd* used for the binary search. In addition, BIC abandons its slow-start procedures to ensure a new binary search commences within the reduced range.

(9)

```
ACK ≡ if $SS_B$ = false
          $\Delta_B \leftarrow \frac{(TGT_B - cwnd)}{B_B}$
          $cwnd \leftarrow cwnd + \frac{f_\alpha(\Delta_B, cwnd, TGT_B)}{cwnd}$
          if $(cwnd < MAX_B)$
              $MIN_B \leftarrow cwnd$
              $TGT_B \leftarrow \frac{(MAX_B + MIN_B)}{2}$
          otherwise
              $SS_B \leftarrow$ true
              $MAX_B \leftarrow cwnd \times 2$
              $SSW_B \leftarrow 1$
              $SST_B \leftarrow cwnd + 1$
      otherwise
          $cwnd \leftarrow cwnd + \frac{SSW_B}{cwnd}$
          if $cwnd \geq SST_B$
              $SSW_B \leftarrow SSW_B \times \frac{B_B}{(B_B - 1)}$
              $SST_B \leftarrow cwnd + SSW_B$
          if $SSW_B \geq MAX_B$
              $SS_B \leftarrow$ false
              $TGT_B \leftarrow \frac{(MAX_B + MIN_B)}{2}$
```

with

$$f_\alpha(\Delta_B, cwnd, TGT_B) \equiv \begin{cases} \frac{B_B}{\sigma_B} & \text{if } (\Delta_B < 1 \wedge cwnd < TGT_B) \vee (TGT_B \leq cwnd < TGT_B + B_B) \\ \Delta_B & \text{if } (1 < \Delta_B < SMAX_B) \wedge cwnd < TGT_B \\ \frac{TGT_B}{(B_B - 1)} & \text{if } B_B < cwnd - TGT_B < SMAX_B \times (B_B - 1) \\ SMAX_B & \text{otherwise} \end{cases} \tag{10}$$

5.2.1.3 Timeout Procedures. For BIC timeout procedures (12) we adopt logic similar to that used for a loss, except that the *sst* is set to half the *cwnd* and the *cwnd* is set to its initial value ($cwnd_{INT}$). This ensures that standard (or limited) slow-start is used until the flow's *cwnd* reaches the new *sst*. After that, congestion-avoidance procedures resume.

$$\textbf{Loss} \equiv \left| \begin{array}{l} MAX_B \leftarrow \frac{(1+\beta_B)}{2} \times cwnd \quad if \quad cwnd < PREV_B \\ MAX_B \leftarrow cwnd \quad otherwise \\ PREV_B \leftarrow MAX_B \\ TGT_B \leftarrow \mathbf{0} \\ SS_B \leftarrow \mathbf{false} \\ cwnd \leftarrow \beta_B \times cwnd \\ sst \leftarrow cwnd \end{array} \right. \tag{11}$$

$$\textbf{Timeout} \equiv \left| \begin{array}{l} MAX_B \leftarrow \frac{(1+\beta_B)}{2} \times cwnd \quad if \quad cwnd < PREV_B \\ MAX_B \leftarrow cwnd \quad otherwise \\ PREV_B \leftarrow MAX_B \\ TGT_B \leftarrow \mathbf{0} \\ SS_B \leftarrow \mathbf{false} \\ sst \leftarrow \mathbf{max}\left(\frac{cwnd}{2}, cwnd_{INT}\right) \\ cwnd \leftarrow cwnd_{INT} \end{array} \right. \tag{12}$$

5.2.2 CTCP

Compound TCP, or CTCP, augments (or compounds) the congestion window with a second component, called the delay window (*dwnd*). (See Table 5-4 for a complete listing of symbols and parameter settings used to specify CTCP behavior.) The *dwnd* is added to the *cwnd* to establish the actual send window used for CTCP flows. CTCP defines rules for increasing *dwnd* aggressively when a flow is underutilizing the available transmission rate and also defines rules for reducing *dwnd* as a flow's transmission rate nears the available bandwidth. Upon detection of congestion, either through explicit losses or timeouts, CTCP reduces the delay window toward zero.

CTCP procedures update the *dwnd* periodically, typically once per round-trip time. As a flow's transmission rate nears equilibrium around some estimated available bandwidth, CTCP tends to cause the send window to oscillate by exponentially increasing the *dwnd* when the estimated number of packets queued for a flow falls below a threshold ($\gamma_C = 30$) and then linearly decreasing *dwnd* when the estimated number of queued packets exceeds the threshold. On the other hand, when the transmission rate is increasing on a flow, CTCP exponentially increases the *dwnd* without exerting a countervailing linear decrease. Consequently, the CTCP send window can reach a large size relatively quickly when a transmission path exhibits no congestion.

As shown below (13), CTCP uses normal TCP congestion-avoidance procedures for adjusting the *cwnd* whenever the *cwnd* is below a threshold ($LW_C = 41$). This means that the *dwnd* is used only when the *cwnd* is sufficiently large.

$$\textbf{SelectProcedures} \equiv \left| \begin{array}{l} \textbf{TCPcongestionAvoidance} \quad if \quad cwnd < LW_C \\ \textbf{CTCPcongestionAvoidance} \quad otherwise \end{array} \right. \tag{13}$$

Table 5-4. Symbols and Definitions Used to Model CTCP Congestion-Avoidance Procedures

Symbol	Definition
α_C	Window increase (α_C = 0.125) weight for CTCP
A_C	Actual throughput (*cwnd*/*SRTT*$_C$) experienced on CTCP flow
β_C	Window decrease (β_C = 0.5) weight for CTCP
CD_C	Boolean denoting whether early congestion has been detected (**true**) or not (**false**)
γ_C	CTCP gamma threshold (γ_C = 30) for detecting early congestion
D_C	Difference between expected and actual throughput experienced on CTCP flow
dwnd	CTCP delay window
E_C	Expected throughput (*cwnd*/*minRTT*$_C$) on CTCP flow
ζ_C	CTCP zeta parameter (ζ_C = 0.1) defining reduction speed in delay window
k_C	Exponent (k_C = 0.8) for CTCP window-increase procedures
LW_C	Low-window threshold (LW_C = 41) for applying CTCP procedures
$minRTT_C$	Minimum round-trip time experienced on CTCP flow
$SRTT_C$	Average Smoothed Round-Trip Time experienced on CTCP flow

5.2.2.1 Increase Procedures. CTCP increases the *cwnd* fractionally with each ACK received in a round-trip time without a loss. The increased *cwnd* is then added to the current *dwnd*. As with other congestion-avoidance procedures, CTCP suspends increases after a loss in a round-trip time until an ACK arrives associated with a subsequent round-trip. The increase procedures adopted by CTCP consider both the *cwnd* and the *dwnd*, as follows.

$$\text{ACK} \equiv \left| \begin{array}{l} cwnd \leftarrow cwnd + \dfrac{1}{(cwnd + dwnd)} \\ cwnd \leftarrow cwnd + dwnd \end{array} \right. \qquad (14)$$

5.2.2.2 Decrease Procedures. Upon a loss, CTCP decreases the *cwnd* by half and then notes that congestion was detected. As with other congestion-avoidance procedures, the *sst* is reset to the *cwnd* in order to ensure the flow remains in congestion avoidance. During the next periodic update cycle, CTCP will act on the loss notification by reducing the *dwnd*. Equation (15) specifies the precise decrease procedures used by CTCP upon an explicit loss.

$$\mathbf{Loss} \equiv \left| \begin{array}{l} cwnd \leftarrow \frac{cwnd}{2} + dwnd \\ CD_C \leftarrow \mathbf{true} \\ sst \leftarrow cwnd \end{array} \right. \tag{15}$$

5.2.2.3 Timeout Procedures. Given a timeout, CTCP adopts the same procedures used by TCP and then augments those procedures by resetting *dwnd* to zero and noting that congestion was detected. The precise procedures follow.

$$\mathbf{Timeout} \equiv \left| \begin{array}{l} sst \leftarrow max\left(\frac{cwnd}{2}, cwnd_{INT}\right) \\ cwnd \leftarrow cwnd_{INT} \\ dwnd \leftarrow \mathbf{0} \\ CD_C \leftarrow \mathbf{true} \end{array} \right. \tag{16}$$

5.2.2.4 Periodic Procedures. The remaining CTCP procedures are used periodically, every round-trip time, to update the *dwnd*. The update procedures (17) depend upon a number of parameters. We adopt the recommended settings for those parameters, as shown in Table 5-4.

$$\mathbf{every}(SRTT_C) \equiv \left| \begin{array}{l} E_C \leftarrow \frac{cwnd}{minRTT_C} \\ A_C \leftarrow \frac{cwnd}{SRTT_C} \\ D_C \leftarrow (E_C - A_C) \times minRTT_C \\ if\ \ CD_C = \mathbf{true} \\ \quad \left| \begin{array}{l} dwnd \leftarrow \mathbf{min}\left[\mathbf{0}, cwnd \times (\mathbf{1} - \beta_C) - \frac{cwnd}{\mathbf{2}}\right] \\ CD_C \leftarrow \mathbf{false} \end{array} \right. \\ dwnd \leftarrow dwnd + \mathbf{min}\left(\mathbf{0}, \alpha_C \times cwnd^{k_C} - \mathbf{1}\right) \quad if\ \ CD_C = \mathbf{false} \wedge D_C < \gamma_C \\ dwnd \leftarrow \mathbf{min}\left[\mathbf{0}, dwnd - (\zeta_C \times D_C)\right] \quad otherwise \\ cwnd \leftarrow \mathbf{max}(\mathbf{int\_max}, cwnd + dwnd) \end{array} \right. \tag{17}$$

5.2.3 FAST

FAST TCP adopts a fundamentally different approach from the other congestion-control mechanisms considered in this study. First, FAST aims to achieve an equilibrium *cwnd* that does not change during the life of a flow, while other congestion-control mechanisms lead to an oscillating *cwnd*. Second, FAST updates the *cwnd* based mainly on measured changes in queuing delay, using loss signals only when congestion prevents reaching a lossless equilibrium. Third, FAST does not resort to standard TCP congestion-avoidance procedures; instead, FAST uses its own procedures at all times during congestion avoidance. FAST adopts these approaches based on the idea that queuing delay can be measured quite frequently and thus accurately, while packet losses are rare events that provide insufficient information on which to estimate loss probability on a given flow.

Explaining FAST congestion-avoidance procedures requires numerous parameters and variables, as listed and defined in Table 5-5. In addition to procedures associated with *cwnd* increase on ACKs and decrease on losses and timeouts, FAST requires a periodic procedure to determine a target *cwnd* ($Tcwnd_F$). FAST also defines optional, periodic procedures for tuning a parameter (α_F), which determines how many packets a flow attempts to keep queued between a source and receiver. These optional, α-tuning procedures require two periodic processes: one to estimate flow throughput and one to adjust α_F based on changes in flow throughput.

Table 5-5. Symbols and Definitions Used to Model FAST Congestion-Avoidance Procedures

Symbol	Definition
$acksRTT_F$	Count of ACKs received on FAST flow during the most recent $SRTT_F$
α_F	Current α parameter
AD_F	Default α parameter setting (AD_F = 200) when α-tuning disabled
AT_F	Boolean indicating whether α-tuning is enabled (**true**) or disabled (**false**)
$A1_F$	First α parameter setting ($A1_F$ = 8)
$A2_F$	Second α parameter setting ($A2_F$ = 20)
$A3_F$	Third α parameter setting ($A3_F$ = 200)
Bk_F	Current average throughput on FAST flow
γ_F	Weight (γ_F = 0.5) of recent information when updating $Tcwnd_F$
$minRTT_F$	Minimum round-trip time experienced on FAST flow
$M0M1_F$	Set α = $A2_F$ when α = $A1_F$ and throughput passes $M0M1_F$ (= 1500 ppms)
$M1M0_F$	Set α = $A1_F$ when α = $A2_F$ and throughput passes $M1M0_F$ (= 1250 ppms)
$M1M2_F$	Set α = $A3_F$ when α = $A2_F$ and throughput passes $M1M2_F$ (= 15,000 ppms)
$M2M1_F$	Set α = $A2_F$ when α = $A3_F$ and throughput passes $M2M1_F$ (= 12,500 ppms)
$SRTT_F$	Average Smoothed Round-Trip Time experienced on FAST flow
T_F	Weight (T_F = 0.5) to assign to most recent throughput sample when computing Bk_F
$Tcwnd_F$	Current target congestion window
UA_F	Periodicity (UA_H = 200 s) for updating α parameter when α-tuning enabled
UW_F	Periodicity (UW_H = 20 ms) for updating $Tcwnd_F$

5.2.3.1 Increase Procedures. FAST uses periodic procedures (explained below in Sec. 5.2.3.4) to determine a target congestion window ($Tcwnd_F$) and then increases or decreases *cwnd* as needed to reach $Tcwnd_F$. FAST does not move *cwnd* to $Tcwnd_F$ in one step, but instead paces the rate of increase to reflect that expected number of ACKs arriving on a given flow within each round-trip time. Our model uses the following procedures for adjusting *cwnd* with each arriving ACK.

$$\mathbf{ACK} \equiv \left| \begin{array}{l} cwnd \leftarrow \dfrac{Tcwnd_F - cwnd}{acksRTT_F} \quad if \quad Tcwnd_F > cwnd \wedge acksRTT_F > \mathbf{0} \\ cwnd \leftarrow Tcwnd_F \quad if \quad Tcwnd_F < cwnd \end{array} \right. \tag{18}$$

5.2.3.2 Decrease Procedures. Upon an explicit loss for FAST, we reduce (19) *cwnd* by half and assign the reduced value to both the $Tcwnd_F$ and the *sst*. These actions provide a new, lower basis from which FAST can begin increasing the *cwnd* and also ensure that the flow remains in congestion avoidance.

$$\mathbf{Loss} \equiv \left| \begin{array}{l} cwnd \leftarrow \dfrac{cwnd}{\mathbf{2}} \\ Tcwnd_F \leftarrow cwnd \\ sst \leftarrow cwnd \end{array} \right. \tag{19}$$

5.2.3.3 Timeout Procedures. For a FAST timeout, we adopt procedures (20) analogous to those used with other congestion-control mechanisms. We set the *sst* to half the *cwnd* and then set *cwnd* and $Tcwnd_F$ to the initial congestion-window ($cwnd_{INT}$) and recommence slow start.

$$\mathbf{Timeout} \equiv \left| \begin{array}{l} sst \leftarrow \mathbf{max}\left(\dfrac{cwnd}{\mathbf{2}}, cwnd_{INT}\right) \\ cwnd \leftarrow cwnd_{INT} \\ Tcwnd_F \leftarrow cwnd \end{array} \right. \tag{20}$$

5.2.3.4 Periodic Procedures. FAST defines one mandatory periodic process (21) to update the target congestion window ($Tcwnd_F$) for the flow every UW_F (= 20 ms, here). A parameter (γ_F) determines how much weight is placed on the previous *cwnd* and how much weight is given to recent information. FAST procedures prevent the new target *cwnd* from being more than twice the current *cwnd*.

$$\mathbf{every}(UW_F) \equiv Tcwnd_F \leftarrow \mathbf{min}\left[\mathbf{2} \times cwnd, (\mathbf{1} - \gamma_F) \times cwnd + \gamma_F \times \left(\frac{minRTT_F}{SRTT_F} \times cwnd + \alpha_F\right)\right] \tag{21}$$

The α_F parameter may be fixed or tuned. If α-tuning is enabled, the following procedures (22) are executed every UA_F (= 200 s, here).

$$\mathbf{every}(UA_F) \quad if \quad AT_F = \mathbf{true} \equiv \left| \begin{array}{l} \alpha_F \leftarrow A2_F \quad if \quad Bk_F \geq M0M1_F \wedge \alpha_F = A1_F \\ \alpha_F \leftarrow A1_F \quad if \quad Bk_F \leq M1M0_F \wedge \alpha_F = A2_F \\ \alpha_F \leftarrow A3_F \quad if \quad Bk_F \geq M1M2_F \wedge \alpha_F = A2_F \\ \alpha_F \leftarrow A2_F \quad if \quad Bk_F \leq M2M1_F \wedge \alpha_F = A3_F \end{array} \right. \tag{22}$$

The various parameters associated with α-tuning are defined in Table 5-5. Estimated flow throughput (Bk_F) is a variable used in the α-tuning procedures. To estimate Bk_F, the following procedures (23) are used each round-trip time.

$$\textbf{every}(SRTT_F) \equiv \left| \begin{array}{l} Bk_F \leftarrow T_F \times Bk_F + (\mathbf{1} - T_F) \times \dfrac{acksRTT_F}{SRTT_F} \\ acksRTT_F \leftarrow \mathbf{0} \end{array} \right. \tag{23}$$

5.2.4 HSTCP

High Speed TCP (HSTCP) modifies standard TCP congestion-control procedures in order to achieve high transmission rates (e.g., 10 Gbps) when network conditions permit, while maintaining comparable performance to standard TCP when a network path exhibits moderate to heavy congestion. HSTCP retains the fundamental additive-increase and multiplicative-decrease (AIMD) strategy adopted by standard TCP; however, HSTCP alters the AIMD parameters to become a function of congestion-window size. The altered AIMD functions result in more aggressive increases and less aggressive decreases at larger window sizes. Below a low-window threshold (LW_{HS}) HSTCP adopts standard TCP congestion-avoidance procedures.

$$\textbf{SelectProcedures} \equiv \left| \begin{array}{ll} \textbf{TCPcongestionAvoidance} & \textit{if } cwnd < LW_{HS} \\ \textbf{HSTCPcongestionAvoidance} & \textit{otherwise} \end{array} \right. \tag{24}$$

Table 5-6 identifies and defines symbols used below when explaining HSTCP congestion-avoidance procedures.

Table 5-6. Symbols and Definitions Used to Model HSTCP Congestion-Avoidance Procedures

Symbol	Definition
HW_{HS}	High-window threshold (HW_{HS} = 83,000) for HSTCP procedures
LW_{HS}	Low-window threshold (LW_{HS} = 31) for applying HSTCP procedures
R_{HS}	Decrease congestion window by this percentage (R_{HS} = 0.1) after loss above LW_{HS}

5.2.4.1 Increase Procedures. HSTCP increases the *cwnd* additively upon receiving each ACK in a round-trip time until a loss is detected. The increase procedures (25) appear quite similar to standard TCP increase procedures, except that the numerator for the increase is a function of *cwnd* size.

$$\textbf{ACK} \equiv cwnd \leftarrow cwnd + \frac{f_\alpha(cwnd)}{cwnd} \tag{25}$$

Function $f_\alpha(c)$, defined below (26), returns the increase numerator that will yield the desired packet drop rate for a given window c. Function $f_\alpha(c)$ uses a subsidiary function, $g_\beta(c)$, defined below (27). Function $g_\beta(c)$ is also used to determine the multiplicative-decrease parameter applied on losses and timeouts.

$$f_\alpha(c) \equiv c^2 \times \frac{0.078}{c^{1.2}} \times 2 \times \frac{g_\beta(c)}{2 - g_\beta(c)} + 0.5 \tag{26}$$

$$g_\beta(c) \equiv \left(R_{HS} - 0.5\right) \times \frac{\log_{10}(c) - \log_{10}\left(LW_{HS}\right)}{\log_{10}\left(HW_{HS}\right) - \log_{10}\left(LW_{HS}\right)} + 0.50 \tag{27}$$

5.2.4.2 Decrease Procedures. Upon detecting an explicit loss, HSTCP reduces the *cwnd* by a multiplicative factor that is a function of the *cwnd*. The specific procedures, which use function $g_\beta(c)$, are given below.

$$\textbf{Loss} \equiv \left|\begin{array}{l} cwnd \leftarrow \left[\mathbf{1} - g_\beta(cwnd)\right] \times cwnd \\ sst \leftarrow cwnd \end{array}\right. \tag{28}$$

5.2.4.3 Timeout Procedures. For a timeout (29), HSTCP sets *sst* to the reduced *cwnd* and then resets the *cwnd* to its initial value. This enables slow-start procedures up to the new *sst*, after which congestion avoidance resumes.

$$\textbf{Timeout} \equiv \left|\begin{array}{l} cwnd \leftarrow \left[\mathbf{1} - g_\beta(cwnd)\right] \times cwnd \\ sst \leftarrow \mathbf{max}\left(cwnd, cwnd_{INT}\right) \\ cwnd \leftarrow cwnd_{INT} \end{array}\right. \tag{29}$$

5.2.5 H-TCP

H-TCP differs from other congestion-avoidance procedures in two main aspects. First, H-TCP determines the numerator of the *cwnd* increase as a function of elapsed time since the most recent packet loss. In order to improve fairness, the increase is scaled by the round-trip time experienced on a path. The motive is to give larger increases in *cwnd* during periods of low network congestion; thus, a flow could reach higher transmission rates more quickly on uncongested, high-bandwidth, long-delay paths. H-TCP adopts standard TCP increase procedures for a specified time after each loss. Second, H-TCP implements an adaptive back-off procedure to determine the multiplicative decrease in *cwnd* after a loss. The back-off factor is varied based on estimating the queuing delay on a path. The motive is to prevent senders from backing off too much after packet losses. H-TCP adopts standard TCP decrease procedures when flow throughput has changed by more than a specified amount since the most recent loss. To monitor changes in flow throughput, H-TCP requires a periodic process to measure average throughput. Table 5-7 identifies and defines parameters and variables used below to explain H-TCP congestion-avoidance procedures.

5.2.5.1 Increase Procedures. For each ACK received without a loss in a round-trip time, H-TCP increases *cwnd* by a fraction of the *cwnd*. The numerator of the increase fraction is a function

(30) of the most recently computed multiplicative-decrease parameter (β_H) and the elapsed time (Δ_H) since the most recent loss.

$$\text{ACK} \equiv cwnd \leftarrow cwnd + \frac{2 \times (1 - \beta_H) \times f_\alpha(\Delta_H)}{cwnd} \tag{30}$$

Table 5-7. Symbols and Definitions Used to Model H-TCP

Symbol	Definition
$acksRTT_H$	Count of ACKs received on H-TCP flow during the most recent U_H
β_H	Most recent computed percentage (initially $\beta_H = 0.5$) *cwnd* residual on a loss
$Bk1_H$	Average throughput on H-TCP flow at time of most recent loss
Bk_H	Current average throughput on H-TCP flow
Δ_H	Time elapsed since the most recent loss
ΔB_H	Percentage throughput increase ($\Delta B_H = 0.2$) for selecting
ΔL_H	Use normal TCP procedures until $\Delta_H > \Delta L_{H,}$ where $\Delta L_H = 1$ s
G_H	Maximum percentage ($G_H = 0.8$) *cwnd* reduction on a loss
$maxRTT_H$	Maximum round-trip time experienced on H-TCP flow
$minRTT_H$	Minimum round-trip time experienced on H-TCP flow
T_H	Weight ($T_H = 0.5$) to assign to most recent throughput sample when computing Bk_H
U_H	Periodicity ($U_H = 250$ ms) for updating throughput estimate for H-TCP flow

Function $f_\alpha(\Delta_H)$ returns (31) one if standard TCP increase procedures are in effect; otherwise returns a value exhibiting a quadratic increase with increasing Δ_H. The quadratic increase (32) is scaled by the minimum round-trip time measured on the flow.

$$f_\alpha(\Delta_H) \equiv \left| \begin{array}{ll} 1 & if \quad \Delta_H \le \Delta L_H \\ \max\left[h_\alpha(\Delta_H) \cdot minRTT_H, 1\right] & otherwise \end{array} \right. \tag{31}$$

$$h_\alpha(\Delta_H) \equiv 1 + 10 \times (\Delta_H - \Delta L_H) + 0.25 \times (\Delta_H - \Delta L_H)^2 \tag{32}$$

5.2.5.2 Decrease Procedures. Upon an explicit loss, H-TCP reduces (33) the *cwnd* by a fraction, computed as a function (34) of changing throughput. In addition, H-TCP records the average flow throughput at the time of the loss.

$$\textbf{Loss} \equiv \left| \begin{array}{l} \beta_H \leftarrow g_\beta\left(\left|\dfrac{Bk_H - Bk1_H}{Bk1_H}\right|\right) \\ cwnd \leftarrow cwnd \times \beta_H \\ Bk1_H \leftarrow Bk_H \end{array} \right. \tag{33}$$

Given default parameters, the H-TCP algorithm varies the back-off fraction between 0.5 and 0.8. The lower value (larger reduction) is adopted whenever measured throughput (Bk_H) has changed by a significant percentage (ΔB_H) since the most recent loss, which suggests that the flow is undergoing some disturbance or transition. Less significant change in throughput indicates that the flow is nearer to stability. Stable flows are reduced by a fraction reflecting the estimated queuing delay as a proportion of estimated propagation delay. The residual *cwnd* is capped by parameter G_H (= 0.8).

$$g_\beta(B) \equiv \left| \begin{array}{l} \mathbf{0.5} \quad \textit{if} \quad B > \Delta B_H \\ \mathbf{min}\left(\dfrac{minRTT_H}{maxRTT_H}, G_H\right) \quad \textit{otherwise} \end{array} \right. \tag{34}$$

5.2.5.3 Timeout Procedures. For a timeout, we adopt procedures (35) that mirror the rules H-TCP uses for an explicit loss with significant change in flow throughput. This amounts to reducing the *sst* to half the *cwnd*, recording the flow's average throughput and setting β_H = 0.5. We also reset the *cwnd* to its initial value, which reinitiates slow start.

$$\textbf{Timeout} \equiv \left| \begin{array}{l} \beta_H \leftarrow \mathbf{0.5} \\ Bk1_H \leftarrow Bk_H \\ sst \leftarrow \mathbf{max}\left(\dfrac{cwnd}{\mathbf{2}}, cwnd_{INT}\right) \\ cwnd \leftarrow cwnd_{INT} \end{array} \right. \tag{35}$$

5.2.5.4 Periodic Procedures. To support recording and monitoring of flow throughput, H-TCP requires a periodic process (36) to estimate average throughput. In our model, estimated throughput is updated every U_H (= 250) ms.

$$\textbf{every}(U_H) \equiv \left| \begin{array}{l} Bk_H \leftarrow T_H \times \dfrac{acksRTT_H}{U_H} + (\mathbf{1} - T_H) \times Bk_H \\ acksRTT_H \end{array} \right. \tag{36}$$

5.2.6 SCALABLE TCP

Scalable TCP adopts a simple, fixed-increase rule aimed at allowing a flow to increase its congestion window more quickly than would be the case with standard TCP. In addition, Scalable TCP defines a decrease rule that limits a flow to a fixed multiplicative decrease that is recommended to be much less than the 50% decrease used by standard TCP. The Scalable TCP rules are defined in an additive-increase, multiplicative-decrease (AIMD) form; however, the rules actually amount to a multiplicative-increase, multiplicative-decrease (MIMD) regime. Researchers have found [1] that MIMD algorithms are not guaranteed to converge to fair bandwidth sharing in drop-tail networks, such as the Internet. Empirical measurements [67] have also shown that failure to converge is a property of Scalable TCP. Below, we describe Scalable TCP procedures for increase on ACK, decrease on explicit loss and decrease on timeout. Our description uses the symbols and definitions shown in Table 5-8.

Table 5-8. Symbols and Definitions Used to Model Scalable TCP

Symbol	Definition
α_S	Increase (α_S = 0.01) applied by Scalable TCP on each ACK
β_S	Percentage residual *cwnd* (β_S = 0.875) applied by Scalable TCP on each loss
LW_S	Low-window threshold (LW_S = 16) for applying Scalable TCP procedures

Scalable TCP includes a low-window threshold (LW_S) that ensures standard TCP procedures for congestion avoidance are followed when the *cwnd* is small. Scalable TCP congestion-avoidance procedures are used only when the *cwnd* reaches the threshold.

$$\mathbf{SelectProcedures} \equiv \left| \begin{array}{ll} \mathbf{TCPcongestionAvoidance} & \textit{if} \;\; cwnd < LW_S \\ \mathbf{ScalableTCPcongestionAvoidance} & \textit{otherwise} \end{array} \right. \quad (37)$$

5.2.6.1 Increase Procedures. Upon receiving each ACK within a round-trip time without a congestion signal Scalable TCP increases (38) the *cwnd* by a fixed value α_S (= 0.01).

$$\mathbf{ACK} \equiv cwnd \leftarrow cwnd + \alpha_S \quad (38)$$

5.2.6.2 Decrease Procedures. Upon receiving an explicit-loss notification Scalable TCP reduces (39) the *cwnd* by a fixed percentage. Here, the reduction amounts to (1 - β_S =) 0.125. We also set *sst* to the new, lower *cwnd* to ensure the flow remains in congestion avoidance.

$$\mathbf{Loss} \equiv \left| \begin{array}{l} cwnd \leftarrow cwnd \times \beta_S \\ sst \leftarrow cwnd \end{array} \right. \quad (39)$$

5.2.6.3 Timeout Procedures. For a timeout we define procedures (40) that require Scalable TCP to set the *sst* to the reduced *cwnd* and then reset the *cwnd* to its initial value. This means that, like the other congestion-control mechanisms we model, Scalable TCP will reenter slow start until the *cwnd* passes the *sst* and then return to congestion avoidance.

$$\mathbf{Timeout} \equiv \left| \begin{array}{l} cwnd \leftarrow cwnd \times \beta_S \\ sst \leftarrow \mathbf{max}(cwnd, cwnd_{INT}) \\ cwnd \leftarrow cwnd_{INT} \end{array} \right. \quad (40)$$

5.3 Modeling the Transfer Phase in MesoNetHS

The section explains the key ideas underlying our model for the transfer phase of a flow. We begin by explaining how our model simulates data-transfer procedures in general and then concentrate separately on slow start and congestion avoidance. Previously in Sec. 5.2, we explained the detailed slow-start and congestion-avoidance procedures for individual congestion-control mechanisms. Here, we focus on the common approach used by MesoNetHS to model the transfer phase across all congestion-control mechanisms.

5.3.1 General Data-Transfer Procedures

We adopt a simplified model of data-transfer procedures in order to simulate fundamental aspects of congestion control without incurring the detailed complexity of TCP implementations. A simplified model permits simulating reasonably large, fast networks for suitable time durations on standard computing hardware without incurring excessive costs in processing time and memory use. Our simplified model retains key properties that enable us to compare and contrast various congestion-control mechanisms under a wide range of network conditions.

During the data-transfer phase for each flow, a simulated source transfers a randomly selected number (*flowDTs*) of DT segments. Each DT segment is assigned a sequence number; the first segment is number one and the sequence number increases by one for each subsequent segment. A flow's receiver, then, expects to receive DT segments in sequence, where each segment has a sequence number one greater than the most recently received segment. When the sequence number is as expected, the receiver sends an ACK back to the source. When the sequence number is higher than expected, the receiver sends a NAK back to the source. The ACK or NAK is numbered with the next expected sequence number. This simplification, which ignores the possibility for reordered segments, is feasible because MesoNetHS allows packets to be discarded but does not permit packet reordering. Absent reordering, our model of data-transfer procedures may omit features such as duplicate ACKs and selective ACKs.

A source in our model expects to receive a stream of ACKs and NAKs from a flow's receiver. Since a receiver sends an ACK or NAK only upon receiving a DT segment, a source can simply count each received ACK and NAK as evidence that one DT segment has been delivered successfully to the receiver. When the source has received one ACK or NAK for each segment comprising the flow, then the data transfer is finished and the source can terminate the flow. Of course, each NAK received by a source also causes the number of DT segments sent on the flow to increase by one (a retransmission) because the NAK indicates that one DT segment was lost and, thus, was not counted as successfully delivered. Receiving a NAK also stimulates a source to take remedial action. In our model, receipt of a NAK activates a source's loss procedures on a flow.

ACK and NAK segments may also be lost on a flow. This means that each lost ACK or NAK will not be counted by the source. For this reason, each lost ACK and NAK will also increase by one (a retransmission) the number of DT segments that must be sent by the source in order to receive a sufficient number of ACKs and NAKs. Further, if no ACKs or NAKs are received by a source for a retransmission-timeout (RTO) period, then a source must also take remedial action. In our model, expiration of a RTO activates a source's timeout procedures on a flow.

Whenever a NAK is received or a timeout occurs, a source notes the next sequence number that it intends to send on the flow. This enables the source to ignore window increase and decrease procedures for all subsequent ACKs and NAKs that arrive with lower sequence numbers. This technique ensures that window-increase procedures are abandoned in a round-trip time after a loss or timeout. The technique also ensures that window-decrease procedures are activated only once within a round-trip time.

The remaining elements of our general data-transfer model concern controlling the ability of a source to transmit a DT. A source maintains a flow *cwnd* using the procedures described earlier (Sec. 5.2). A source also knows the sequence number (*nextSeq*) for the next DT segment it intends to send and the highest sequence number (*highSeq*) received in an ACK or NAK. With this information, a source can compute (41) the number of unacknowledged DT segments (*unAckedDTs*) and thus the number of DT segments it is permitted to send (*unsentDTs*).

$$\left| \begin{array}{l} unAckedDTs \leftarrow nextSeq - highSeq \\ unsentDTs \leftarrow cwnd - unAckedDTs \end{array} \right. \tag{41}$$

Equation 41 reveals the self-clocking nature of TCP flows. Two conditions enable a source to send a DT segment: arrival of an ACK or NAK or increase in the congestion window. In our model, a timeout causes *highSeq* to be set to *nextSeq*, which means that *unsentDTs* will equal *cwnd*. Recall that we also reset *cwnd* to its initial value upon a timeout.

One last detail must be explained. The procedures in equation 41 can cause extra DTs to be sent at the end of a flow. To prevent this, our model computes (42) the difference (*residualDTs*) between the number of DTs (*flowDTs*) comprising a flow and unacknowledged DTs (*unAckedDTs*). The number of DTs that can be sent (*unsentDTs*) is then set to the minimum of the segments allowed by the congestion window or the segments required to complete the flow.

$$\left| \begin{array}{l} residualDTs \leftarrow flowDTs - unAckedDTs \\ unsentDTs \leftarrow \mathbf{min}(unsentDTs, residualDTs) \end{array} \right. \tag{42}$$

Data-transfer procedures are distributed across elements of MesoNetHS. Sources manage sending of DT segments and also react to flow timeouts. Access routers process incoming ACKs and NAKs on behalf of sources and also process incoming DTs on behalf of receivers. When processing incoming ACKs and NAKs an access router updates the *cwnd* as required by the congestion-control mechanism in use on the related flow. When processing incoming DTs an access router determines whether a receiver needs to send an ACK or NAK and then queues that assignment for the receiver.

5.3.2 Slow Start

Slow-start procedures are invoked within a simulated access router upon receipt of an ACK whenever the *cwnd* is below the *sst*. If the *cwnd* is below sst_{MAX}, then standard slow-start procedures are used to increase the *cwnd* at an exponential rate; otherwise, limited slow-start procedures are used to increase the *cwnd* at a logarithmic pace.

5.3.3 Congestion Avoidance

Congestion-avoidance procedures are invoked within a simulated access router upon receipt of any qualified NAK, and for qualified ACKs where the *cwnd* equals or exceeds the *sst*. Selected congestion-control mechanisms also require periodic procedures, which our model implements within simulated access routers. We implement timeout procedures within simulated sources.

5.3.3.1 Acknowledgement Procedures. MesoNetHS assigns one congestion-control mechanism to each simulated source, which represents a computer attached to the network and running a particular version of TCP. This means that the particular congestion-control mechanism in operation on a simulated flow will be determined by the congestion-control mechanism used by the flow's source. Upon receipt of a qualified ACK a simulated access router selects the appropriate window-increase procedures for the flow as a function of the congestion-control mechanism (*tcpType*) used by the simulated source. Qualified ACKs include all ACKs received within a round-trip time prior to a congestion signal.

5.3.3.2 Negative Acknowledgement Procedures. Upon receipt of a qualified NAK a simulated access router selects the appropriate window-decrease procedures for the flow as a function of the *tcpType* used by the simulated source. Qualified NAKs include the first NAK received within any given round-trip time for a flow.

5.3.3.3 Periodic Procedures. In general, MesoNetHS activates periodic procedures only after a flow passes initial slow start. Periodic procedures that estimate throughput are always active during a flow's transfer phase. MesoNetHS implements periodic procedures in a somewhat approximate form. Specifically, periodic procedures are invoked within a simulated access router only when an ACK or NAK has been received and provided that sufficient time has elapsed. Further, the timer is reset only after invoking the related procedures. Thus, MesoNetHS does not invoke periodic procedures on a precisely rigid schedule, as might be stimulated by a timer. Periodic procedures can be invoked regardless of whether an ACK or NAK is qualified to stimulate increase or decrease procedures for a flow.

5.3.3.4 Timeout Procedures. MesoNetHS invokes timeout procedures within a simulated source when a flow's RTO expires. A source's RTO is reset within a simulated access router whenever any ACK or NAK arrives for the source. Upon expiration of the RTO, a source selects the appropriate timeout procedures for the flow as a function of the source's *tcpType*.

5.4 Verifying Simulated Congestion-Control Mechanisms

To verify the behavior of congestion-control mechanisms simulated within MesoNetHS, we defined a test configuration similar to that used in an empirical study [67] of congestion-control mechanisms implemented in Linux. We also adopted parameters used in the empirical study and then simulated similar scenarios and recorded the evolution of the congestion window. Below, we give our simulated *cwnd* graphs and compare the behavior of our simulated congestion-control mechanisms to findings reported in the empirical study. First, we describe the test configuration adopted to produce the reported *cwnd* graphs.

We defined a dumbbell topology, shown in Fig. 5-6, similar to the dumbbell topology used in the empirical study. The topology in Fig. 5-6 is annotated with key parameter values used to generate the results presented below. The topology consists of two sources that attach to the same access router. Each source can transmit DTs to one of a pair of receivers that attach to the second access router in the topology. The empirical study places a *dummynet* router between the sources and receivers and uses that router to control propagation delay, bottleneck speed and buffer provisioning on the network path between the sources and receivers. Our simulations use MesoNetHS facilities to control path characteristics.

Access Router #1, highlighted in red in Fig. 5-6, simulates the bottleneck bandwidth for the path. Here, the bottleneck speed is set to 21 p/ms (packets/millisecond), which amounts to 21 x 1000 x 12000 = 252 Mbps, assuming 1500-byte packets. This approximates a 250 Mbps bottleneck link used in the empirical study. Note that the sources, receivers and backbone routers are configured with speeds exceeding the bottleneck access routers. The sources and receivers are capable of transmitting at close to 1 Gbps (960 Mbps), as are the backbone routers (984 Mbps).

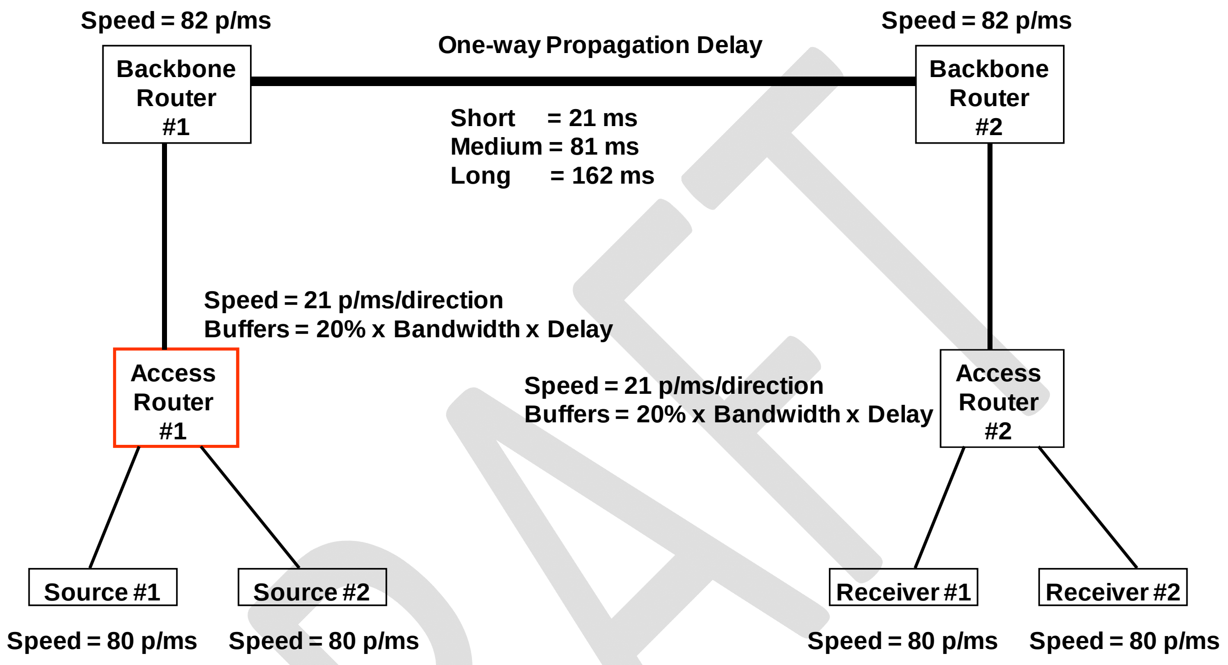

Figure 5-6. Simulated Dumbbell Topology for MesoNetHS Verification Experiments

The propagation delay of the network path in Fig. 5-6 is controlled by the one-way propagation delay of the backbone line. Round-trip propagation delay will be twice the one-way propagation delay. For our simulations with the dumbbell topology we used three different one-way delays (21 ms, 81 ms and 162 ms) to match the short (42 ms), medium (162 ms) and long (324 ms) round-trip propagation delays used in the empirical study.

We configured MesoNetHs to provision buffers for each router sufficient to accommodate the bandwidth-delay product. MesoNetHS also includes a parameter that can adjust the number of provisioned buffers. Here, we reduce the buffers to be 20% of the number required by the bandwidth-delay product. This matches the buffer provisioning used for several scenarios reported in the empirical study.

Given the topology and parameters from Fig. 5-6, we simulate congestion-control mechanisms under a scenario lasting 1000 s, where one source begins sending data immediately and the second source delays (250 s) and then starts to send data. For each congestion-control mechanism we use limited slow-start, with sst_{MAX} = 100 and sst_{INT} = $2^{32}/2$.We repeat the scenario three times for each congestion-control mechanism, varying the round-trip propagation delay (*rtt*) from short, to medium, to long with each repetition. We record and graph the time-varying *cwnd* for each scenario. The abscissa in all graphs is denominated in 100 ms units. All graphs also include the average overall *cwnd* when one flow is transmitting and when both flows are transmitting. When both flows are transmitting, each graph also displays the average *cwnd* for each flow.

5.4.1 Standard TCP Congestion-Control Model

Fig. 5-7 graphs *cwnd* evolution for standard TCP congestion-control under a short propagation delay. The graph shows the expected characterization of the TCP *cwnd*, which (after initial slow start ends with a loss just after the *cwnd* passes 1100) oscillates in a saw-tooth pattern between a *cwnd* of 550 and 1050 (average about 800). The bandwidth-delay product is (42 ms x 21p/ms =) 882 packets; thus, an average *cwnd* of 800 seems appropriate. After the second flow begins (at time 2500) it takes between 50 and 100 s for the *cwnd* of the two flows to converge to an equal value (average around 400). Convergence to a similar *cwnd* means the two flows will receive fairly equal average throughputs. This property of convergence to fairness is a hallmark trait of TCP congestion control.

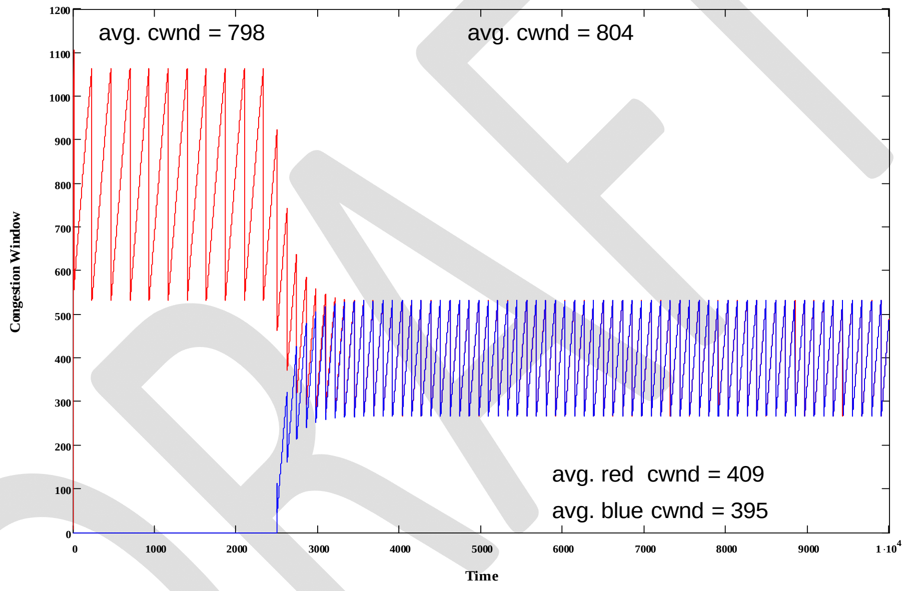

Figure 5-7. Evolution of *cwnd* for Two TCP Flows (*rtt* = 42 ms)

The next scenario, displayed in Fig. 5-8, begins to show why many researchers believe standard TCP congestion-control procedures are ill-suited to high-speed, long-delay environments. Here, the 162 ms round-trip propagation delay (*rtt*) suggests a *cwnd* of (162 ms x 21p/ms =) 3402 packets. The first flow reaches (and then exceeds) that value during slow start, which ends with a loss (at *cwnd* = 4200) early in the flow. After the loss, TCP reduces the *cwnd* in half (to 2100) and then TCP enters its congestion-avoidance regime. Increasing the *cwnd* with standard TCP congestion-avoidance procedures requires about 150 s for the flow to reach its peak window. Thus, absent other activities, the single flow would oscillate in throughput over periods of about 150 s.

Once the second flow commences, the *cwnd* of the two flows begin to converge; however, the lengthy propagation delay slows the increase in *cwnd* and the rate of convergence. In fact, the two flows in Fig. 5-8 have not fully converged even after 750 s. The situation becomes worse when *rtt* becomes even longer, as shown in Fig. 5-9. Further, increasing the network speed would increase the bandwidth-delay product and worsen the delay in recovering from packet losses and converging to fair throughputs.

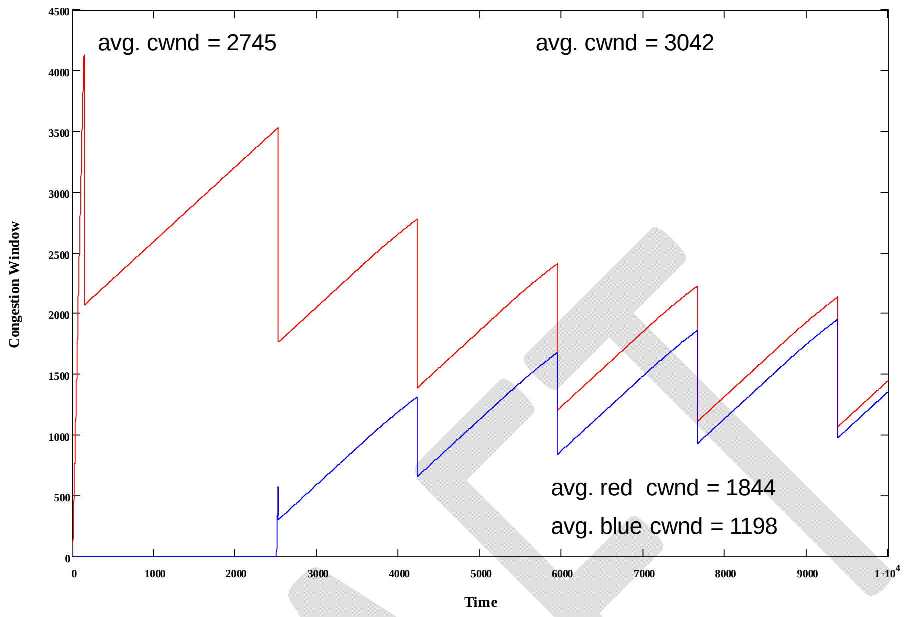

Figure 5-8. Evolution of *cwnd* for Two TCP Flows (*rtt* = 162 ms)

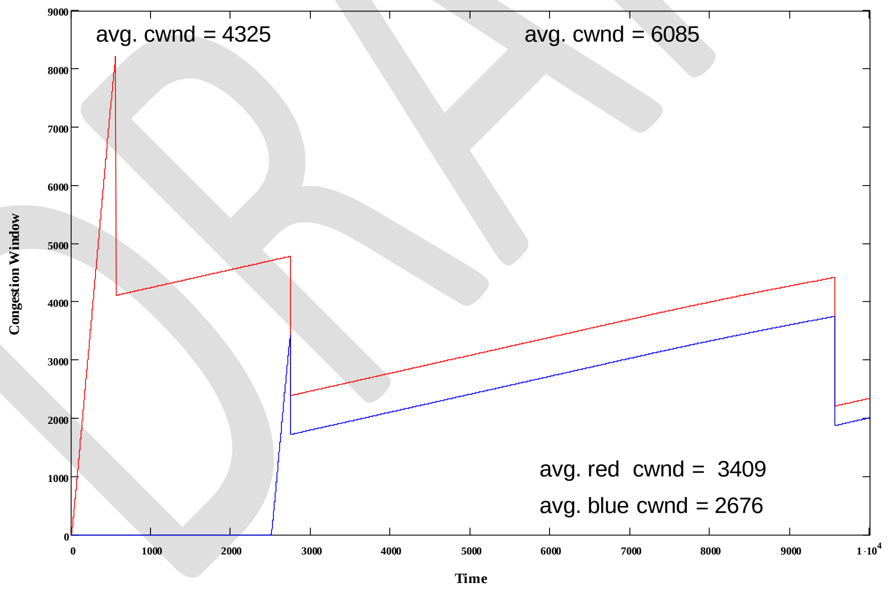

Figure 5-9. Evolution of *cwnd* for Two TCP Flows (*rtt* = 324 ms)

5.4.2 Behavior of BIC Congestion-Control Model

Next, we subject BIC congestion-control to the same three scenarios under which we simulated standard TCP. The resulting *cwnd* evolutions are shown in Figs. 5-10 through 5-12. The graphs display the heartbeat-like pattern of BIC *cwnd* evolution, as seen in the empirical study. Note that BIC congestion avoidance shows small improvement in convergence time for scenarios with short and medium propagation delays. At the long propagation delay, BIC exhibits significantly less fairness in bandwidth allocation than standard TCP. These findings are consistent with findings in the empirical study.

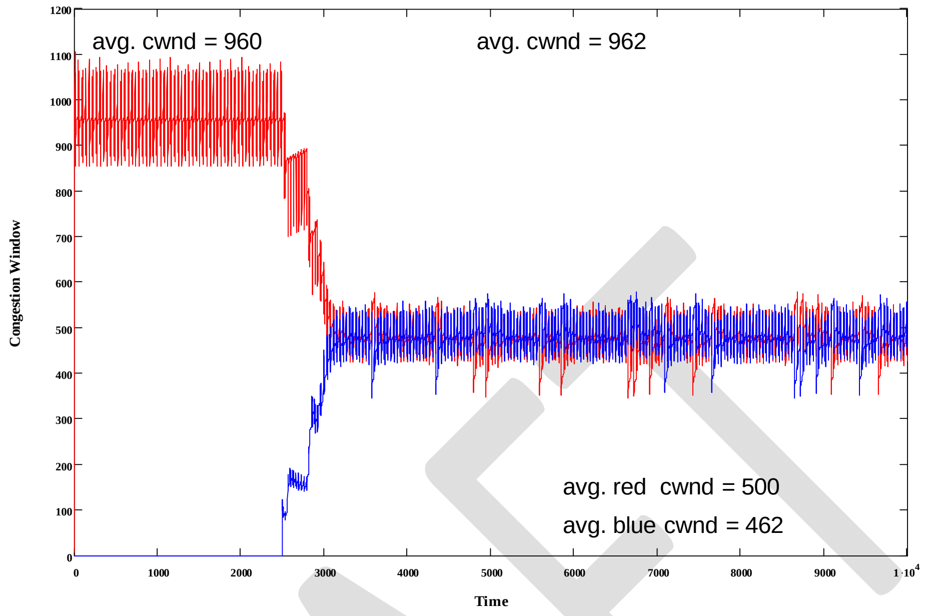

Figure 5-10. Evolution of *cwnd* for Two BIC Flows (*rtt* = 42 ms)

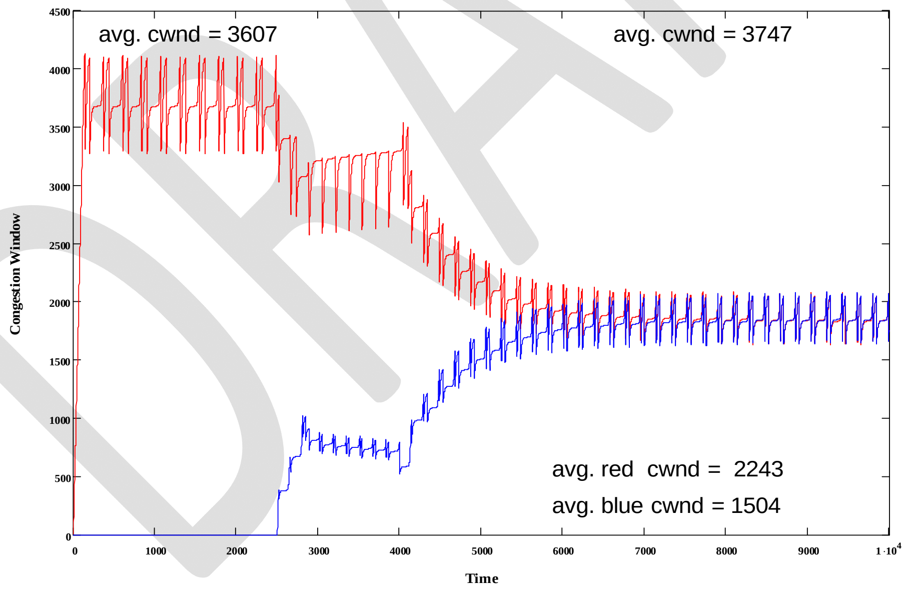

Figure 5-11. Evolution of *cwnd* for Two BIC Flows (*rtt* = 162 ms)

5.4.3 Behavior of CTCP Congestion-Control Model

The empirical study against which we compare our simulations for the other congestion-control mechanisms did not include CTCP; thus, in verifying the behavior of CTCP we must compare our simulations to results from a later empirical study [66]. Unfortunately, the later study did not adopt the same parameters and scenarios used in the first study. For that reason, comparing our

CTCP simulation results to the empirical study is not quite as direct as for the other congestion-control mechanisms. We can compare the pattern of *cwnd* evolutions between the simulations and the empirical results and we can compare the CTCP-related findings from the empirical study against the findings from our simulation.

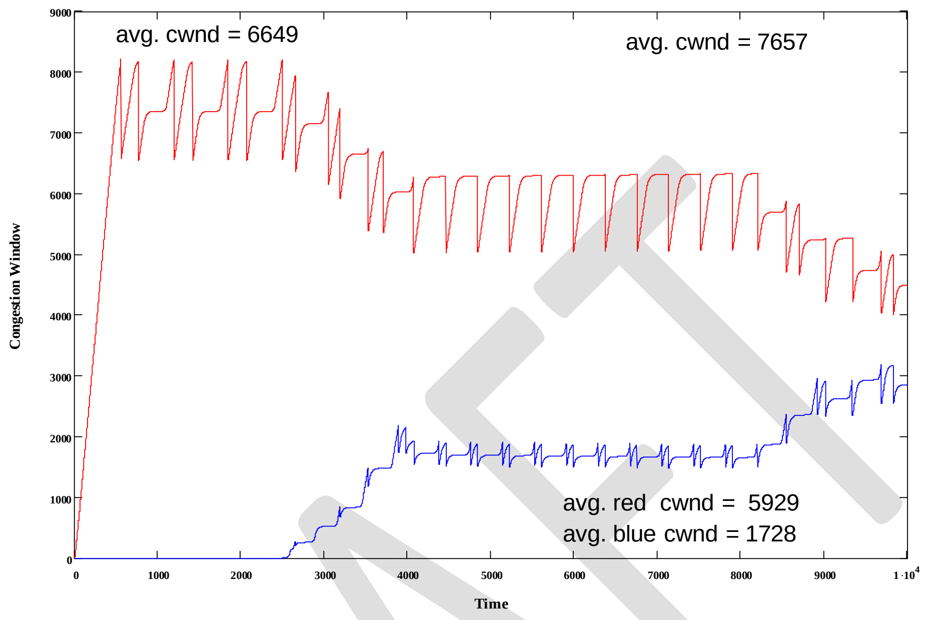

Figure 5-12. Evolution of *cwnd* for Two BIC Flows (*rtt* = 324 ms)

Figs. 5-13 through 5-15 show *cwnd* evolution for CTCP under our scenario with short, medium and long *rtt*. CTCP exhibits a distinctive pattern of *cwnd* evolution, which becomes evident once the second flow starts in Fig. 5-13 and 5-14. This pattern is also evident in one of the CTCP *cwnd* graphs in the empirical study. The empirical study reports that the time taken by CTCP to recover from a loss, as well as the convergence time when a second flow begins, is similar to standard TCP. Further, the empirical study finds that convergence time scales linearly with bandwidth-delay product. The MesoNetHS simulation of CTCP exhibits the same properties, as shown in the *cwnd* graphs.

The empirical study reports that CTCP exhibits similar *rtt* fairness to TCP Reno, but when buffers are smaller CTCP has slightly better *rtt* fairness. MesoNetHS simulations also show a similar fairness between CTCP and standard TCP; however, CTCP had a slight edge in *rtt* fairness for the case of medium propagation delay (*rtt* = 162 ms). The empirical study finds that link utilizations can be low and network responsiveness can be sluggish for CTCP. The potential sluggishness of CTCP responsiveness is also evident in Figs. 5-14 and 5-15. In Sec. 5.4.8, we report more about fairness, as well as link and buffer utilization, among all congestion-control mechanisms that we simulated.

5.4.4 Behavior of FAST Congestion-Control Model

The FAST congestion-control algorithm includes α–tuning as an option, which complicates the verification of the FAST simulation within MesoNetHS. The empirical study [67] reports results for FAST with α–tuning enabled. The designers of FAST indicate [61] that α–tuning is no longer used routinely within FAST implementations. Instead, the designers suggest fixing α_F to a value suitable for expected network conditions. Of course, the designers recognize that fixing α_F is not a general solution and list α–tuning as an open issue. In some empirical studies, the designers of FAST set α_F = 200. We report simulation results for FAST under three different configurations: α–tuning enabled (Figs. 5-16 through 5-18), α_F = 80 (Figs. 5-19 through 5-21) and α_F = 200 (Figs. 5-22 through 5-24).

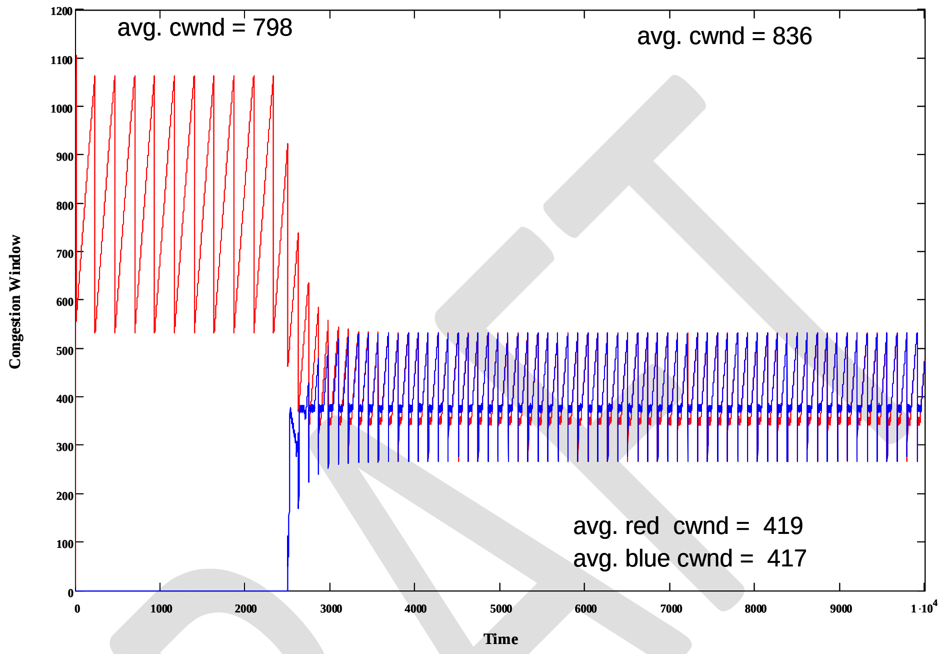

Figure 5-13. Evolution of *cwnd* for Two CTCP Flows (*rtt* = 42 ms)

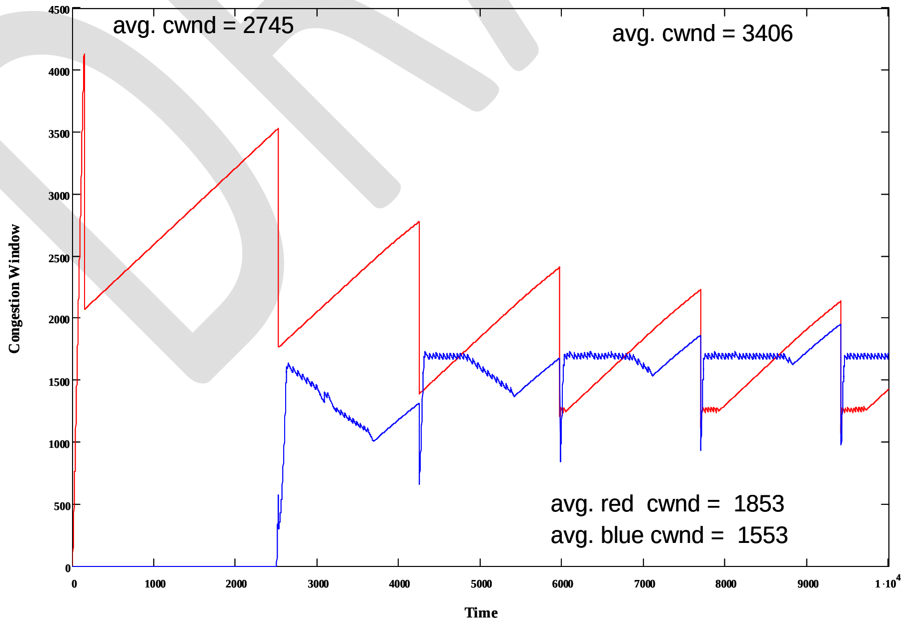

Figure 5-14. Evolution of *cwnd* for Two CTCP Flows (*rtt* = 162 ms)

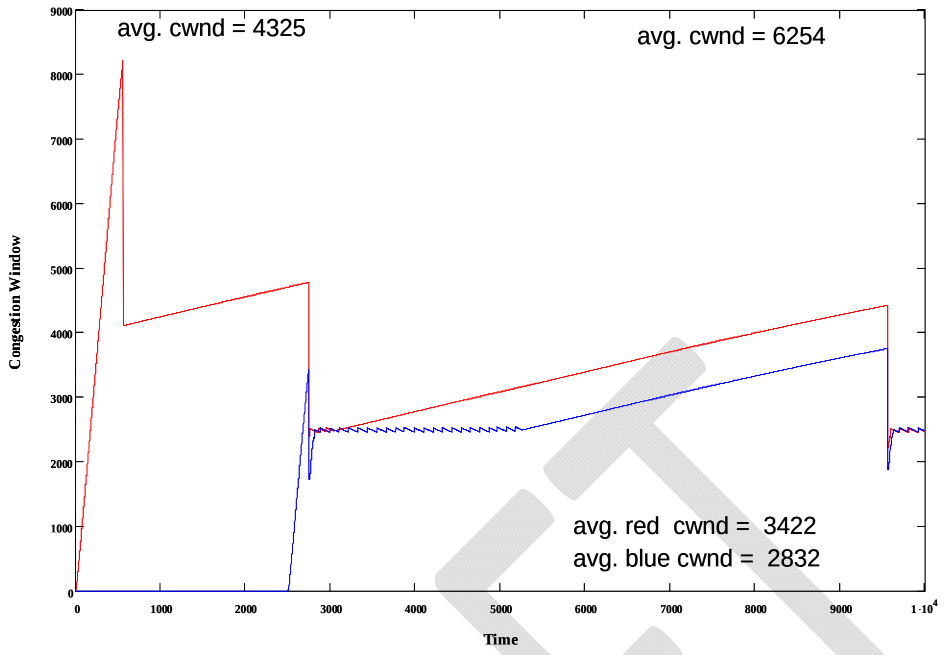

Figure 5-15. Evolution of *cwnd* for Two CTCP Flows (*rtt* = 324 ms)

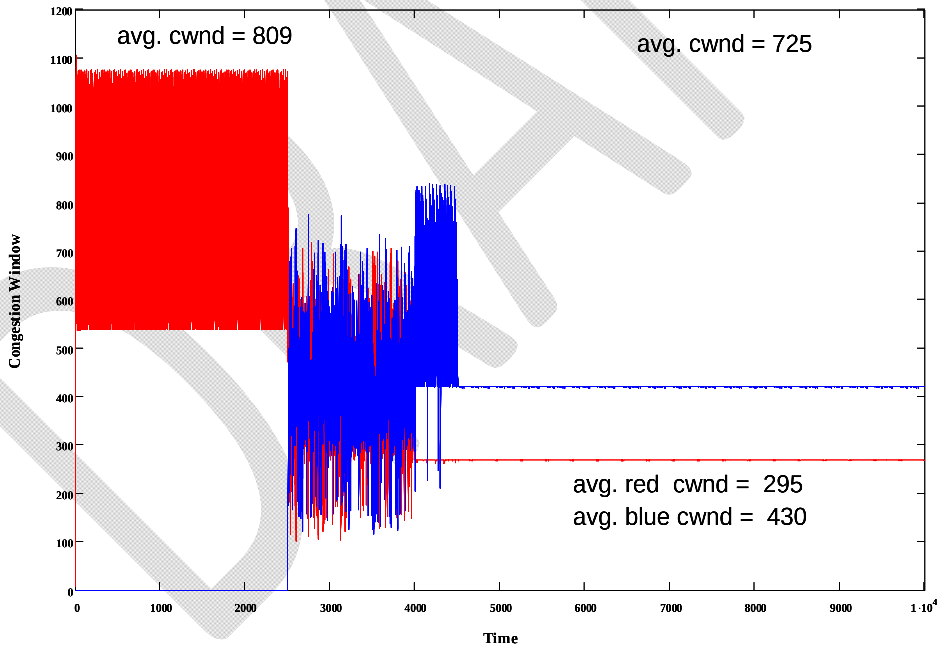

Figure 5-16. Evolution of *cwnd* for Two FAST Flows (*α*–tuning enabled, *rtt* = 42 ms)

The empirical study [67] comparing selected congestion-control mechanisms reports two main findings regarding FAST with *α*–tuning enabled. First, FAST can converge to fair bandwidth allocation and then diverge to unfair allocation. The MesoNetHS simulation shows this trait in Figs. 5-16 to 5-18. Second, when the network path has insufficient buffers to sustain $\alpha_F/2$ queued packets per flow, then *cwnd* oscillates as FAST floods the buffers with too many packets, leading to substantial packet losses. The FAST designers report this same potential to oscillate when buffers are insufficient. The MesoNetHS simulation shows this oscillatory behavior in Fig. 5-16, for each flow prior to reaching equilibrium, which becomes possible once α–tuning reduces α_F from its initial value (200) to 20. Fig. 5-22 also shows this oscillatory behavior for α_F = 200, which prevents either flow from ever achieving equilibrium.

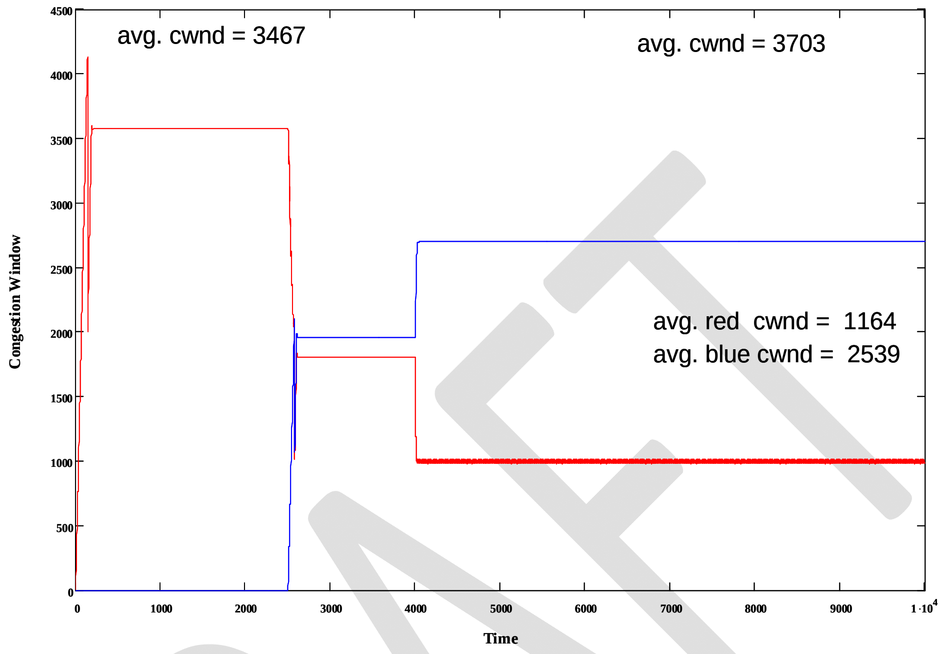

Figure 5-17. Evolution of *cwnd* for Two FAST Flows (α–tuning enabled, *rtt* = 162 ms)

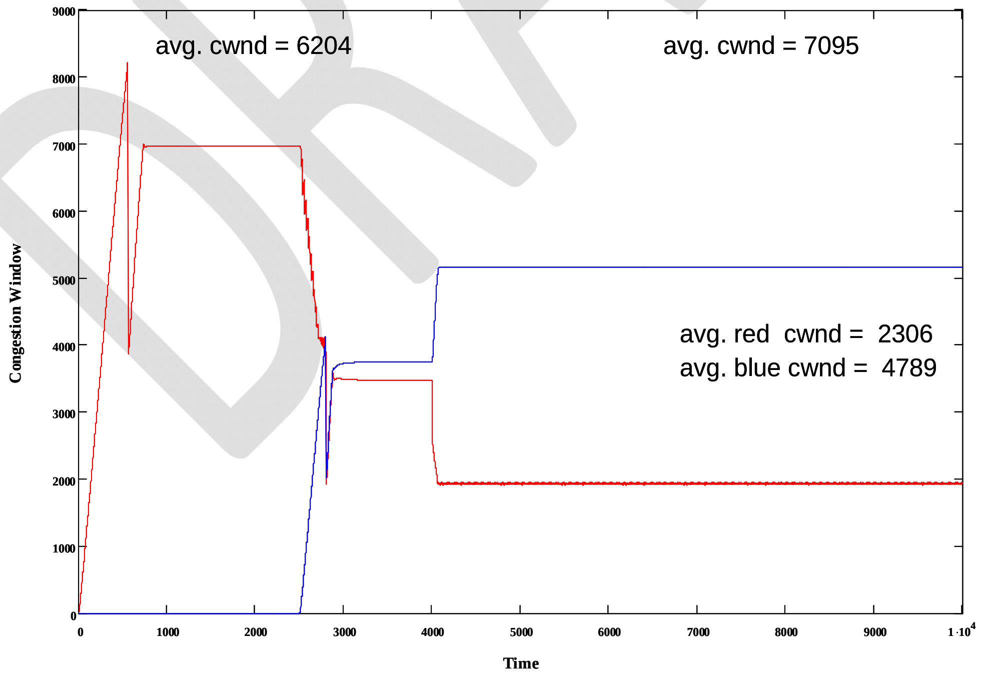

Figure 5-18. Evolution of *cwnd* for Two FAST Flows (α–tuning enabled, *rtt* = 324 ms)

The empirical study also reports that FAST has the fastest convergence time among the congestion-control mechanisms compared. Of course, the study notes the issue of divergence must be taken into account. For all MesoNetHS simulations where FAST converges to equilibrium the convergence time is very fast.

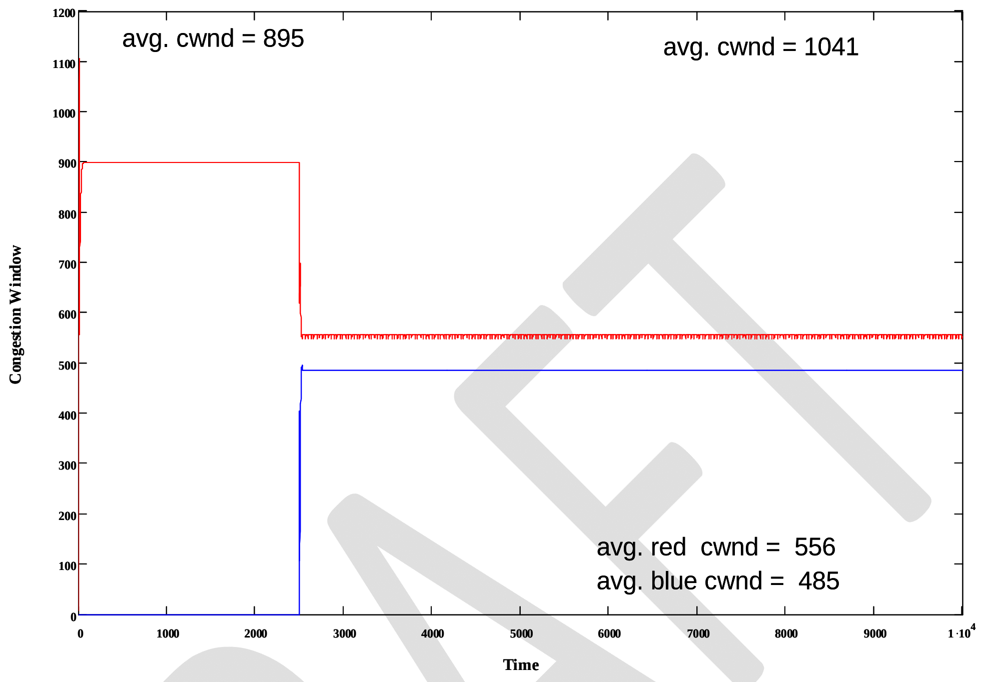

Figure 5-19. Evolution of *cwnd* for Two FAST Flows (α_F = 80, *rtt* = 42 ms)

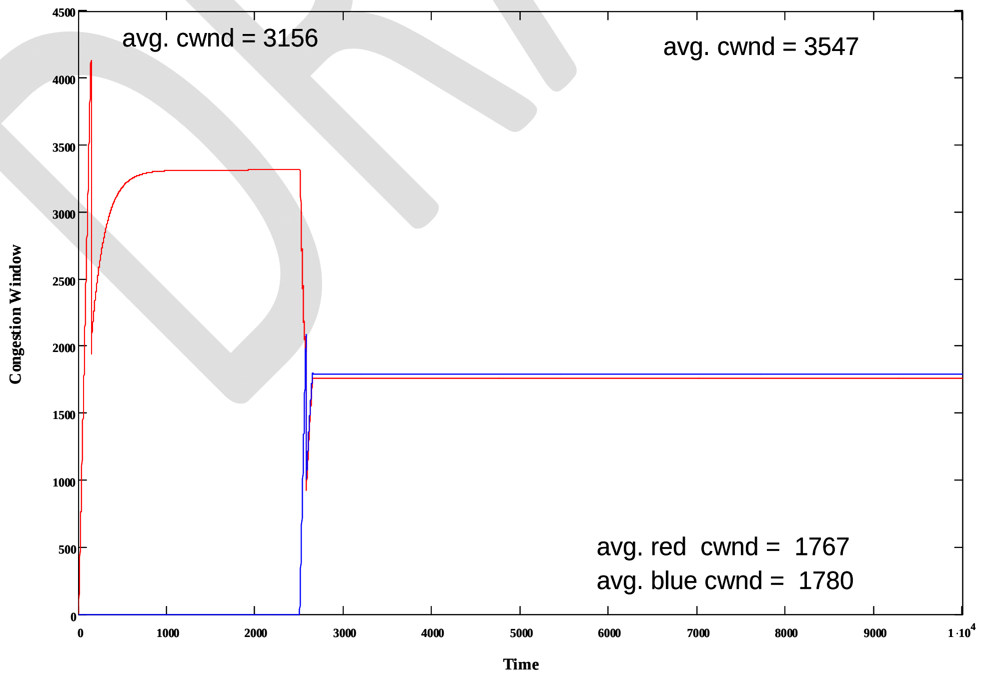

Figure 5-20. Evolution of *cwnd* for Two FAST Flows (α_F = 80, *rtt* = 162 ms)

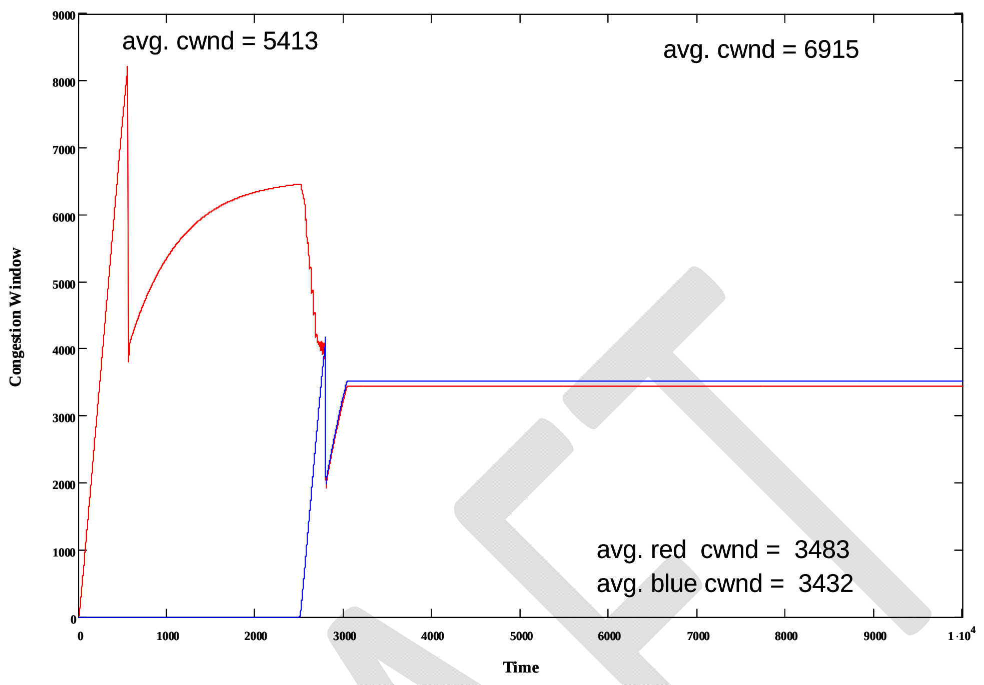

Figure 5-21. Evolution of *cwnd* for Two FAST Flows (α_F = 80, *rtt* = 324 ms)

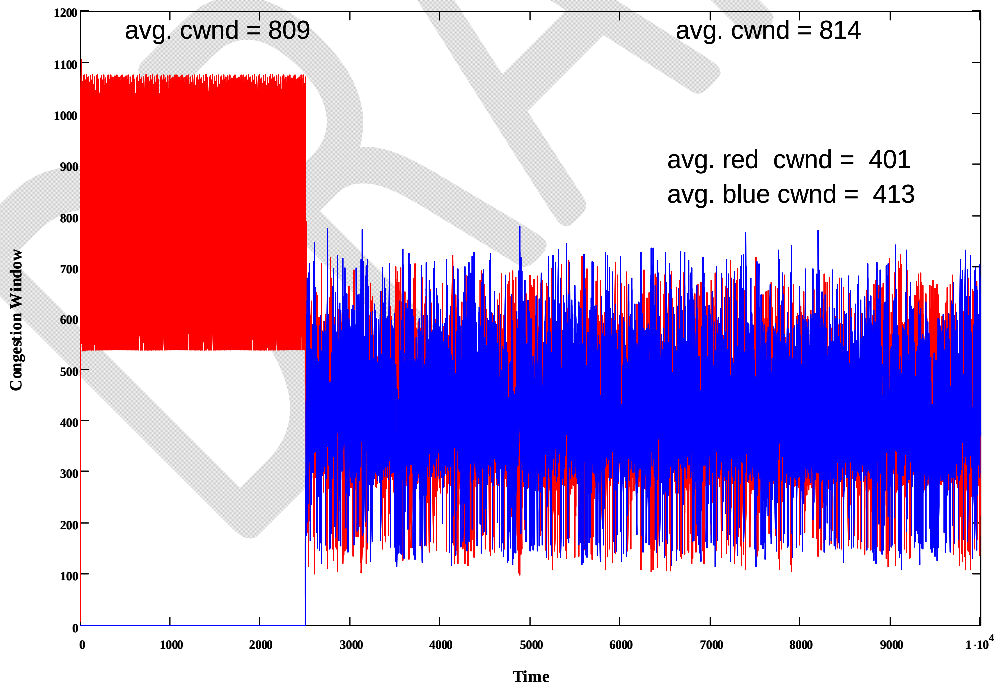

Figure 5-22. Evolution of *cwnd* for Two FAST Flows (α_F = 200, *rtt* = 42 ms)

MesoNetHS simulations achieved closest convergence among *cwnd* for competing flows with α_F = 80, which was the value we determined as best for the simulated network conditions.

We simulated with α_F = 200 to match values used by the designers of FAST in some empirical studies. Where buffers were sufficient, MesoNetHS simulations also achieved close convergence with this larger α_F.

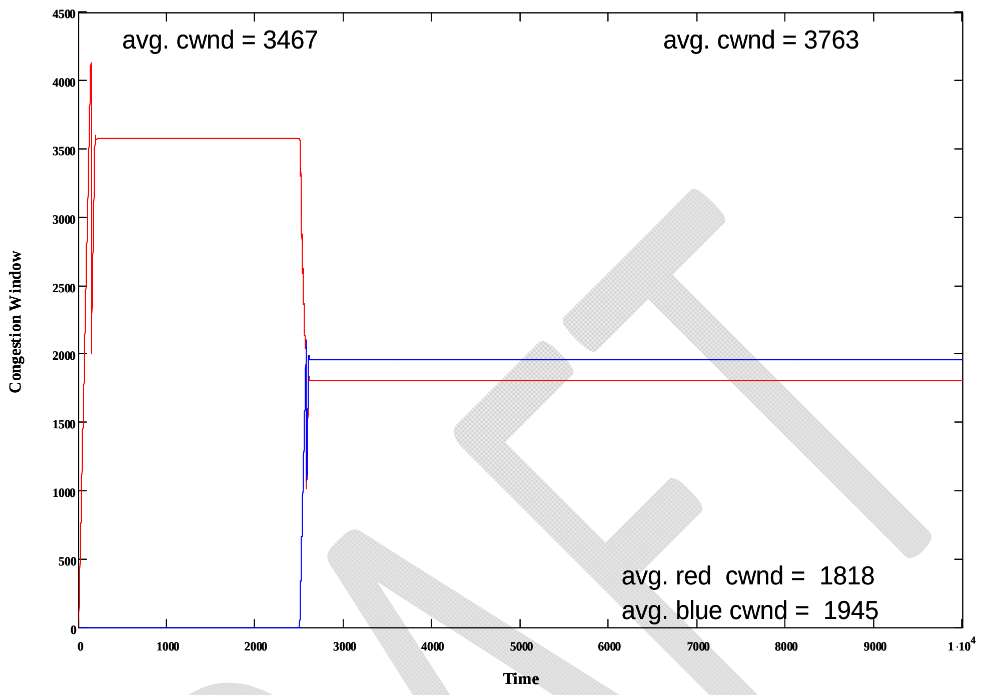

Figure 5-23. Evolution of *cwnd* for Two FAST Flows (α_F = 200, *rtt* = 162 ms)

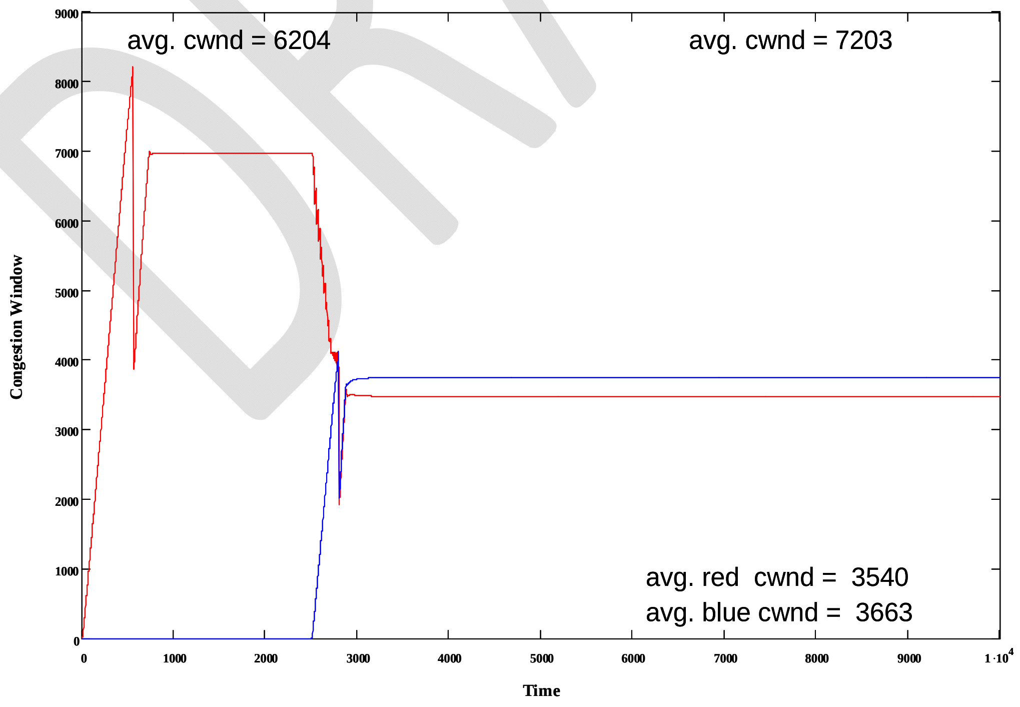

Figure 5-24. Evolution of *cwnd* for Two FAST Flows (α_F = 200, *rtt* = 324 ms)

5.4.5 Behavior of HSTCP Congestion-Control Model

The MesoNetHS simulation results for HSTCP, shown in Figs. 5-25 to 5-27, agree with results from the empirical study. HSTCP flows converge to fairness; however, this requires significant time, which increases with increasing *rtt*.

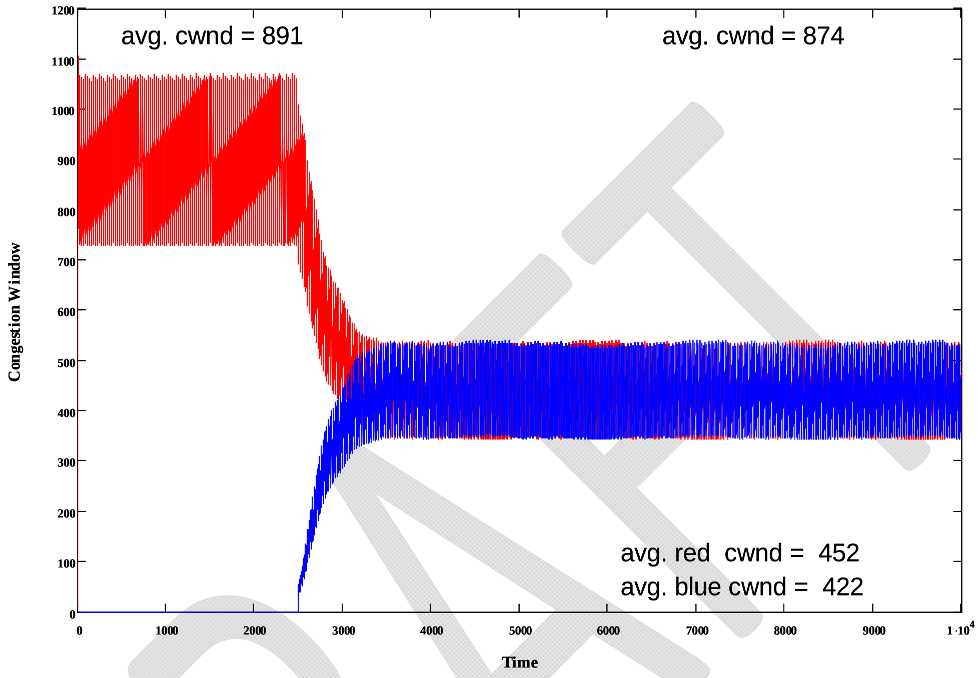

Figure 5-25. Evolution of *cwnd* for Two HSTCP Flows (*rtt* = 42 ms)

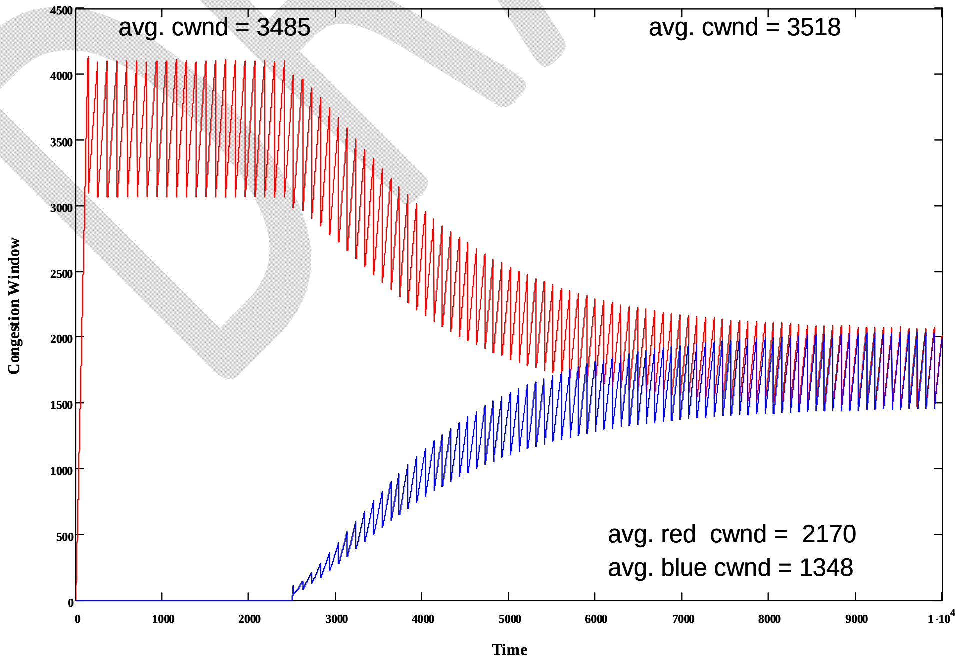

Figure 5-26. Evolution of *cwnd* for Two HSTCP Flows (*rtt* = 162 ms)

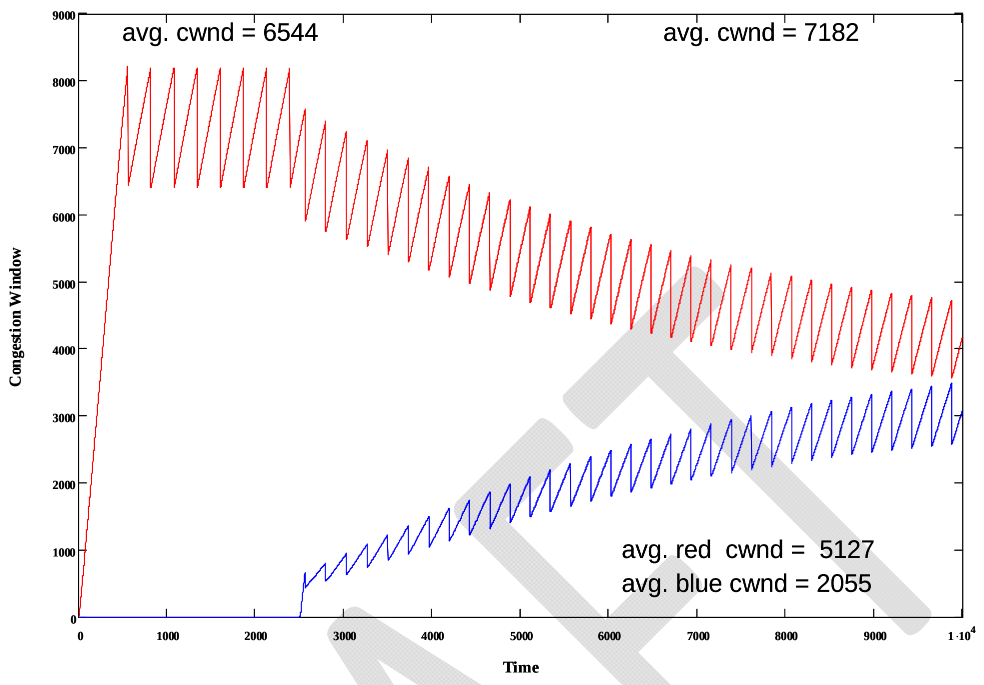

Figure 5-27. Evolution of *cwnd* for Two HSTCP Flows (*rtt* = 324 ms)

5.4.6 Behavior of H-TCP Congestion-Control Model

Figs. 5-28 to 5-30 display the *cwnd* evolutions produced by the MesoNetHS simulation of H-TCP, which appear quite similar in shape to those reported in the empirical study. H-TCP flows in the simulation appear to converge slightly slower than those reported in the empirical study. Convergence times for simulated H-TCP are second fastest among the congestion-control mechanisms simulated. This agrees with results from the empirical study.

5.4.7 Behavior of Scalable TCP Congestion-Control Model

MesoNetHS simulation results for Scalable TCP are shown in Figs. 5-31 to 5-33. For the three *rtt* values simulated, Scalable TCP did not converge to a fair allocation of bandwidth. Scalable TCP implements what amounts to a multiplicative-increase, multiplicative-decrease (MIMD) algorithm, which previous theoretical analysis [1] shows cannot guarantee convergence. The empirical study also found that Scalable TCP either does not converge or converges very slowly. Scalable TCP flows did not converge to fair bandwidth allocation over the 10-minute duration of the tests used in the empirical study. This agrees with the MesoNetHS simulation results.

5.4.8 Summary of Behavior of MesoNetHS Congestion-Control Models

In this section, we provide a summary of the comparative behavior of MesoNetHS simulations across all seven congestion-control mechanisms, as well as the two extra FAST configurations. We consider three aspects of performance: link utilization, buffer utilization and fairness. In making our comparisons, we use average *cwnd* as a surrogate for average throughput. We limit our numerical analyses to two (rounded) decimal places; thus, we do not discuss smaller differences in performance among the congestion-control mechanisms.

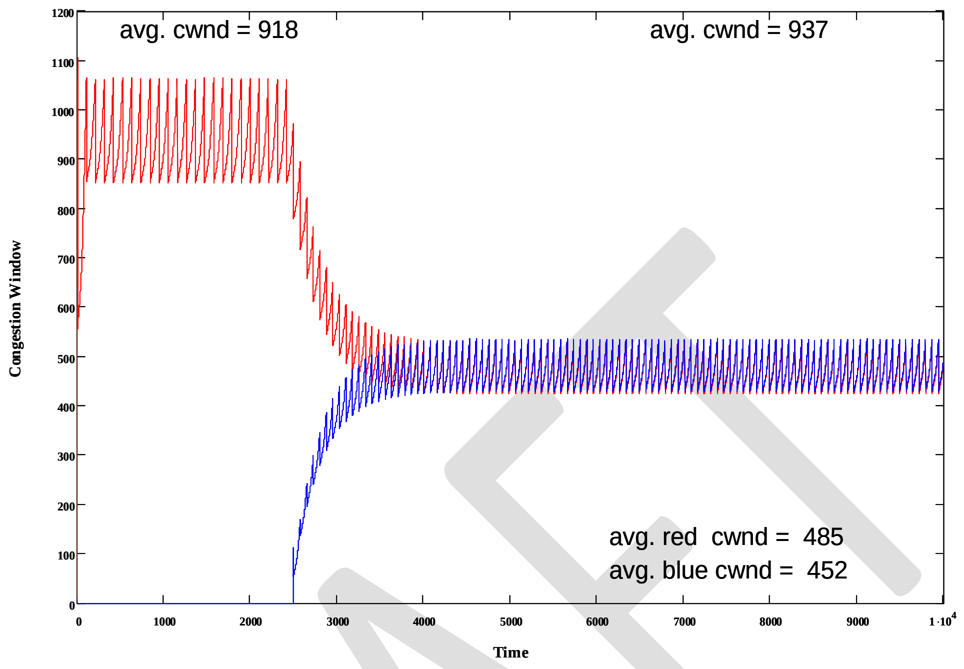

Figure 5-28. Evolution of *cwnd* for Two H-TCP Flows (*rtt* = 42 ms)

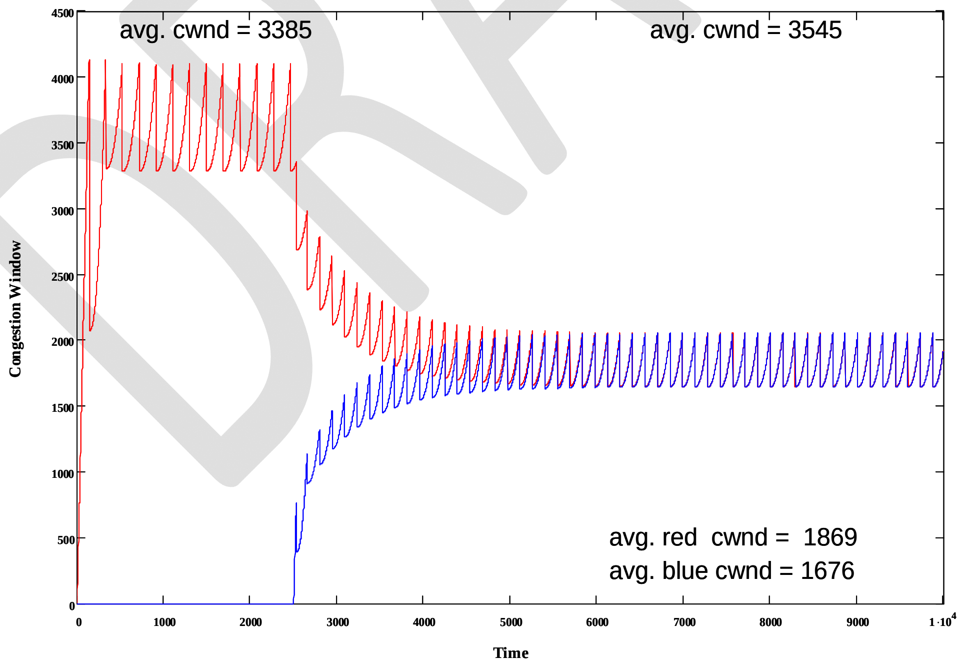

Figure 5-29. Evolution of *cwnd* for Two H-TCP Flows (*rtt* = 162 ms)

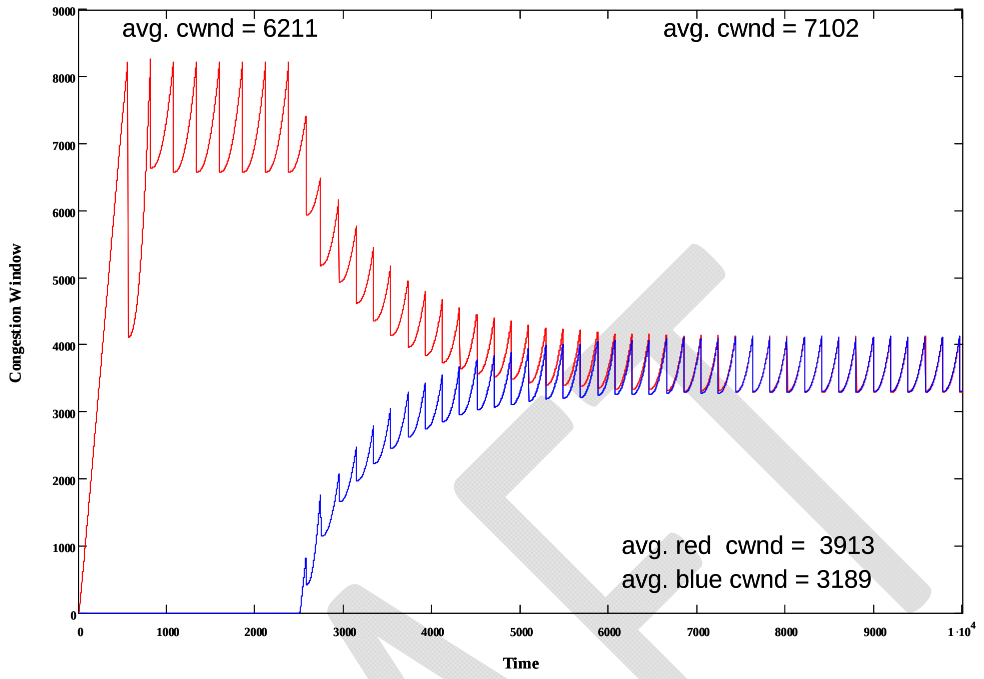

Figure 5-30. Evolution of *cwnd* for Two H-TCP Flows (*rtt* = 324 ms)

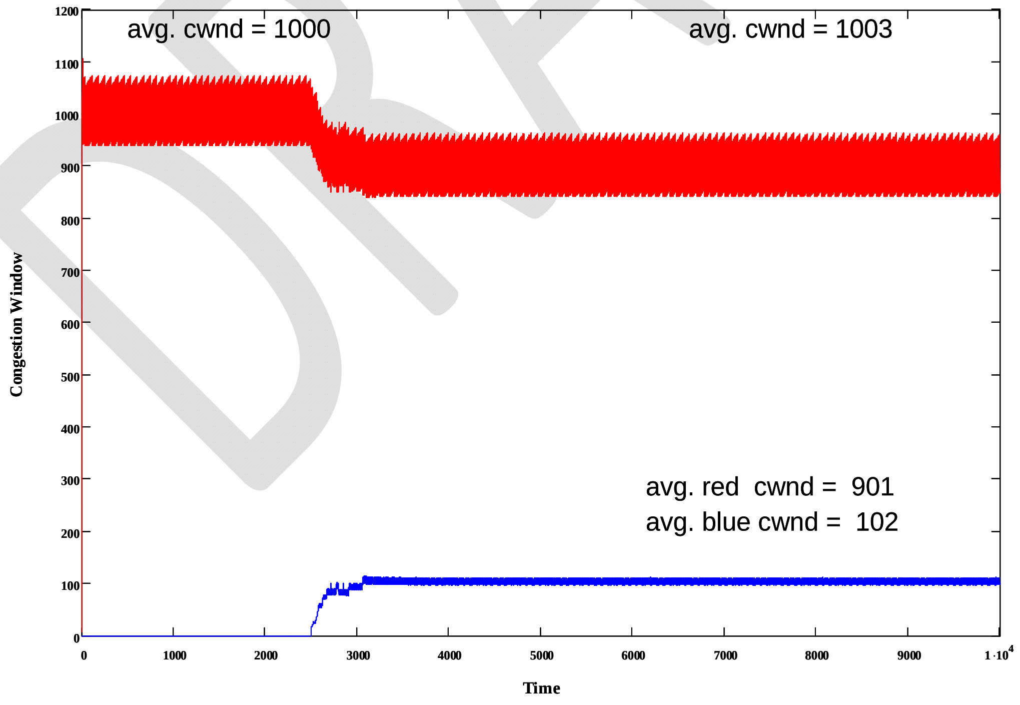

Figure 5-31. Evolution of *cwnd* for Two Scalable TCP Flows (*rtt* = 42 ms)

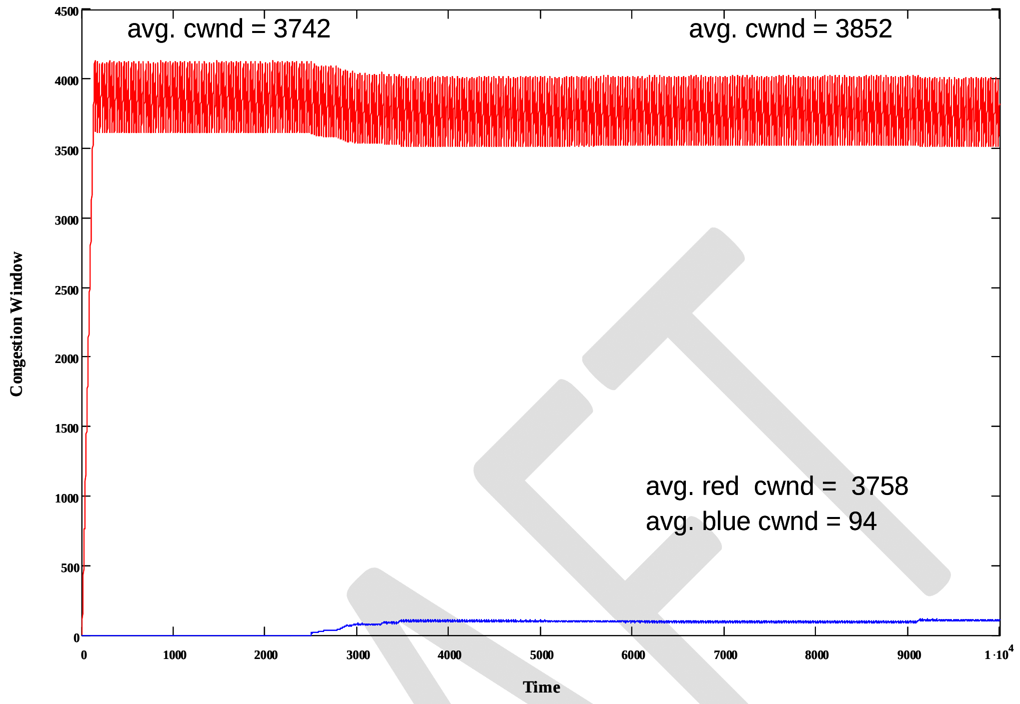

Figure 5-32. Evolution of *cwnd* for Two Scalable TCP Flows (*rtt* = 162 ms)

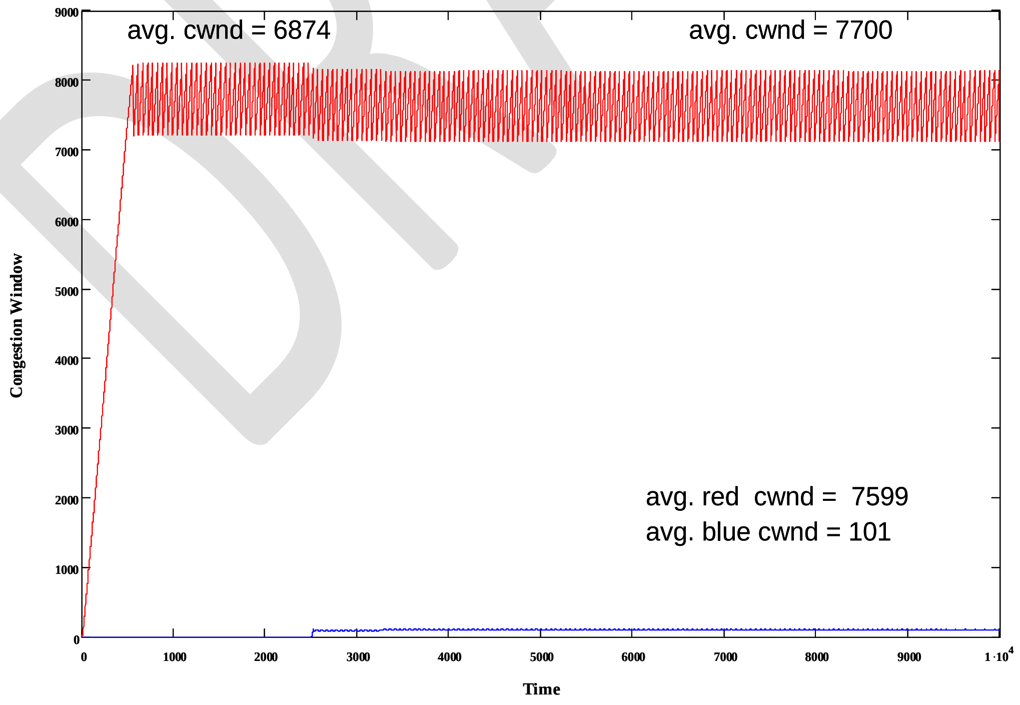

Figure 5-33. Evolution of *cwnd* for Two Scalable TCP Flows (*rtt* = 324 ms)

Table 5-9. Capacity (in Packets) of the Dumbbell Topology with Various Round-Trip Times

	rtt = 42 ms	*rtt* = 162 ms	*rtt* = 324 ms
Bandwidth-Delay Product (packets)	882	3402	6804
Buffers (packets)	176	680	1360
Buffers + Bandwidth-Delay Product	1058	4082	8164

Given a single path with a set of long-lived flows, an ideal congestion-control mechanism would yield a situation where each flow has the same average *cwnd* and the sum of the average *cwnd* over all flows equals the bandwidth-delay product (BDP). In such a situation the link is fully utilized, buffers are empty and each flow receives fair (i.e., the same) bandwidth. While congestion-control mechanisms are unlikely to be ideal, we can compare congestion-control mechanisms by examining relative link and buffer utilizations and fairness.

Table 5-9 (first row) displays the capacity (in packets) of the network path modeled by the dumbbell topology (Fig. 5-6) as a function of *rtt*. These figures define the throughput limits on a path, which caps the maximum link utilization. Once a path contains a sufficient number of packets, then some source will always be able to transmit. As an example, given *rtt* = 42, the path will hold 882 packets in aggregate. Average link utilization can be determined by summing the average *cwnd* over all flows on the path and dividing by the BDP. For example, from Fig. 5-7 we see two TCP flows with average *cwnd* of 409 and 395 packets, respectively. The average link utilization can then be computed as (409 + 395)/882 = 0.91.

In cases where the aggregate average *cwnd* exceeds the BDP, then the excess packets must be sitting in buffers on the path. Table 5-9 (second row) shows the buffer sizes (20% of BDP) as a function of *rtt*. We can estimate the buffer utilization on a path by subtracting the BDP from the aggregate average *cwnd* and then dividing the residual by the number of buffers on the path. (When the residual is $\leq$ 0, buffers are empty.) For example, Fig. 5-31 (*rtt* = 42 ms) shows that Scalable TCP leads to an aggregate average *cwnd* of (901 + 102 =) 1003 packets, giving a residual of (1003 – 882 =) 121 packets in buffers on the path. Thus, buffer utilization is (121/176 =) 0.69. In general, given several congestion-control mechanisms that yield 100% link utilization, we might prefer the one that leads to lowest buffer utilization.

Among several congestion-control mechanisms with high link utilization, we might also prefer the one allocating bandwidth most fairly. To measure fairness, we use Jain's fairness index [64] but applied to *cwnd* rather than throughput. We use the following formulation.

$$\frac{\left(\sum_{i=1}^{n} cwnd_i\right)^2}{n \times \sum_{i=1}^{n} (cwnd_i)^2} \tag{41}$$

Jain's fairness index ranges between 0 and 1, with a higher value denoting better fairness.

Table 5-10 gives link and buffer utilizations for each simulated congestion-control mechanism as a function of *rtt*. Even at the shortest *rtt* (= 42 ms), several of the congestion-control mechanisms fail to achieve full link utilization. For TCP and CTCP this results from slow recovery from packet losses. For FAST with α–tuning low utilization arises from two factors: prior to reaching equilibrium α_F is too high, which leads to substantial packet losses, and after reaching equilibrium α_F is too low to fully utilize the link. α_F is too high for FAST with α_F = 200, which leads to packet losses and an oscillating *cwnd*.

Table 5-10. Link and Buffer Utilizations for Simulated Congestion-Control Mechanisms

	rtt = 42 ms		*rtt* = 162 ms		*rtt* = 324 ms	
	Link Util.	Buffer Util.	Link Util.	Buffer Util.	Link Util.	Buffer Util.
TCP	0.91	0.00	0.89	0.00	0.89	0.00
BIC	1.00	0.45	1.00	0.51	1.00	0.63
CTCP	0.95	0.00	1.00	0.01	0.92	0.00
FAST α Tuning	0.82	0.00	1.00	0.44	1.00	0.21
FAST α_F = 80	1.00	0.90	1.00	0.21	1.00	0.08
FAST α_F = 200	0.92	0.00	1.00	0.53	1.00	0.29
HSTCP	0.99	0.00	1.00	0.17	1.00	0.28
H-TCP	1.00	0.31	1.00	0.21	1.00	0.22
Scalable TCP	1.00	0.69	1.00	0.66	1.00	0.66

As *rtt* increases, all congestion-control mechanisms except CTCP and standard TCP achieve full link utilization. (CTCP does achieve 100% at *rtt* = 162 ms, while maintaining an average of four buffered packets.) Among the congestion-control mechanisms achieving full utilization, H-TCP, HSTCP and FAST (α_F = 80) lead to relatively low buffer utilizations. BIC and Scalable TCP exhibit relatively high buffer utilizations.

Table 5-11 shows Jain's fairness index for the simulated congestion-control mechanisms as a function of *rtt*. As expected, Scalable TCP shows substantial unfairness. The unfairness of BIC and HSTCP increases with *rtt*. Also as expected, FAST with α–tuning leads to unfairness. Several congestion-control mechanisms (CTCP, FAST with fixed α_F, and H-TCP) yield fairness across all values of *rtt*.

As evident from our simulations, several of the proposed congestion-control mechanisms approach ideal performance under the limited cases reported here. H-TCP, FAST and HSTCP give full link utilizations. H-TCP and HSTCP also tend to limit buffer utilization at full link utilization. FAST limits buffer utilization under some circumstances. H-TCP and FAST with fixed α_F also show good fairness across values of *rtt*. How will the various congestion-control mechanisms compare in a larger topology with varying network conditions? We explore this question in the next four sections (Sec. 6-9).

Table 5-11. Bandwidth Fairness for Simulated Congestion-Control Mechanisms

	rtt = 42 ms	*rtt* = 162 ms	*rtt* = 324 ms
TCP	1.00	0.96	1.00
BIC	1.00	0.96	0.77
CTCP	1.00	1.00	1.00
FAST α Tuning	0.97	0.88	0.89
FAST α_F = 80	1.00	1.00	1.00
FAST α_F = 200	1.00	1.00	1.00
HSTCP	1.00	0.95	0.85
H-TCP	1.00	1.00	0.99
Scalable TCP	0.61	0.52	0.51