# Optimizing TCP Start-up Performance

Yin Zhang, Lili Qiu, and Srinivasan Keshav *

**Abstract**

The performance of many networking protocols is dependent on a handful of tuning parameters. However, it is not obvious how to set or adapt these parameters to optimize performance. We believe that this optimization task can benefit from passive monitoring of current network performance.

In this paper, we apply this methodology for initializing TCP parameters, such as initial congestion window size and slow start threshold for short connections. We analytically derive the optimal initial parameters, and use simulations to study its effectiveness. Our innovations include: (i) derivation of optimal TCP initial parameters as a function of link characteristics; (ii) abstract a communication path as a virtual link and model cross traffic as perturbation on it; (iii) an efficient architecture for network performance discovery; (iv) a new pacing algorithm that combines leaky bucket flow control with traditional window-based flow control. Our results show this approach leads to significant performance improvement.

**Key words**: Congestion control, analysis, TCP initial parameters, network performance discovery, pacing, simulation.

## 1 Introduction

The performance of many networking protocols is dependent on a handful of tuning parameters. However it is not obvious how to set or adapt these parameters to optimize performance. We believe that this optimization task can benefit from passive monitoring of current network performance. In this paper, we apply this methodology for initializing TCP parameters.

It is well known that TCP interacts poorly with the World Wide Web, the most popular application on the Internet. The major problem is that the TCP slow start procedure [11], which is initiated both at connection start up and upon restart after an idle period, may require several $RTT$s to probe the network capacity. This is usually too *slow* for Web transfers, which tend to be short and bursty [16]. There have been a lot of efforts trying to address the issue. However, none of them provides a satisfactory solution.

In this paper, we propose a new technique, which we call TCP/SPAND, to improve TCP start-up performance. We first theoretically analyze the problem and derive optimal TCP initial parameters for the simplest scenario in which there is only a single TCP connection and the network condition is given. Recent studies [18] [7] have shown nearby hosts experience similar or identical network performance within a time period measured in minutes. Such level of network stability allows us to abstract an end-to-end communication path as a *virtual link*, which is quite stable, and apply our optimality results to it. In order to accurately determine the virtual link characteristics, we

*Based on the contribution, the order of the first two authors is interchangeable.

have designed an efficient network performance discovery architecture based on the idea of shared, passive network monitoring recently proposed in Berkeley's SPAND project [22].

As in any scheme that can have a potentially large initial congestion window, we need a way of smoothly sending out the packets in the initial window. We propose a novel pacing scheme that solves the problem gracefully. Our scheme incorporates leaky bucket flow control into the traditional window based flow control used by TCP. It helps to reduce the burstiness of TCP, and is potentially useful in other contexts such as *differentiated service* and TCP over satellite links.

We have implemented TCP/SPAND in the **ns** simulator [17]. Our results show that in many situations TCP/SPAND significantly reduces latency for short transfers without degrading the performance of the connections using standard TCP [11]. Therefore deploying TCP/SPAND at Web servers can provide great performance benefit. Furthermore, since our scheme only involves modifications at the sender side, such deployment can be carried out easily, and all the client side applications can be left untouched.

The rest of the paper is organized as follows. Section 2 specifies the problem with TCP's initial parameters. Section 3 briefly overviews the previous work. In Section 4, we theoretically analyze the TCP startup dynamics and derive the optimal initial parameters. Section 5 introduces the virtual link abstraction for a communication path with cross traffic. Section 6 presents an efficient architecture for shared, passive network performance discovery. Section 7 describes our leaky bucket-based pacing scheme. Section 8 presents simulation results to evaluate the effectiveness of our approach. We end with concluding remarks and future work in Section 9.

## 2 Problem Specification

TCP is a closed-loop flow control scheme, in which a source dynamically adjusts its flow control window in response to implicit signals of network overload. Specifically, a source uses the slow start algorithm to probe the network capacity by gradually growing its congestion window until congestion is detected or its window size reaches the receiver's advertised window. The slow start is also terminated if the congestion window grows beyond a threshold. In this case, it uses the congestion avoidance to further open up the window until a loss occurs. It responds to the loss by adjusting its congestion window and other parameters.

The success of TCP relies on the feedback mechanism, where it uses a packet loss as the indication to adapt itself through adjusting a variety of parameters, such as the congestion window size (*cwnd*), the slow start threshold (*ssthresh*), and the smoothed round-trip time (*srtt*) and its variance (*rttvar*). However, for a feedback mechanism to work, a connection should last long enough. When transfers are small compared to the bandwidth-delay product of the link, such as file transfers of several hundred kilobytes over satellite links or typical-sized web pages over terrestrial links, it is very likely that we have no or few feedback – loss indications. In this case, the connection's performance is primarily governed by the initial parameters. In the original TCP [11], these parameters are set arbitrarily, which can lead to very poor performance for such short connections.

## 3 Previous Work

There are a number of proposals on improving the start-up performance of connections.

Two typical examples of application level approaches are launching multiple concurrent TCP connections and P-HTTP (persistent HTTP) [21]. However neither of them solves the problem completely. Using multiple concurrent connections makes TCP overly aggressive for many environments and can contribute to congestive collapse in shared networks [9] [8]. P-HTTP reuses a single TCP connection for multiple Web transfers, thereby amortizing the connection setup overhead, but still pays the full cost of TCP slow start.

T/TCP [5] bypasses the three-way handshaking by having the sender to begin transmitting data in the first segment sent (along with the SYN) [1]. In addition, T/TCP proposes temporal sharing of TCP control block (TCB) state, including maximum segment size (MSS), smoothed RTT, and RTT variance [5]. [6] also mentions the possibility of caching the "congestion avoidance threshold" without offering details.

[13] proposes to use the bandwidth-delay product to estimate the initial *ssthresh*. However estimating the available bandwidth by packet-pair like scheme does not work for the FIFO network. Furthermore, neither T/TCP nor [13] addresses the issue of initial congestion window, so the slow start can still be unnecessarily slow.

To address the issue of the initial congestion window, [3] proposes to increase the initial window to roughly 4K bytes. Since they use a fixed initial window for all connections, the value has to be very conservative, and the improvement is still inadequate in situations where the bandwidth delay product is much larger.

TCP control block interdependence [23] specifies temporal reusing of TCP state, including carrying over congestion window information from one connection to the next. Similar to [23], Fast start [20] reuses the congestion window size (*cwnd*), the slow start threshold (*ssthresh*), the smoothed round-trip time (*srtt*) and its variance (*rttvar*). Directly reusing previous TCP parameters, as proposed in both approaches, can be too aggressive when network condition changes. Fast start [20] addresses it by resorting to router support, which has deployment problems.

In summary, so far there is little analysis on how to set TCP initial parameters to optimize performance. Although intuitively the previous parameters may give good indication of current network state, they are still not optimal. Moreover, information sharing is very primitive in existing approaches. Spatially, information sharing is limited to within a single host; while temporally, only the most recent values are used, and all previous ones are simply discarded, though they can be valuable.

## 4 Optimal Choice of TCP Initial Parameters

To have a better understanding of the problem, we first analyze the TCP start-up dynamics, and derive the optimal initial parameters for a simple scenario, where there is only a single TCP connection and the network characteristics are given (Only congestion loss is considered.). In subsequent sections, we will show how we apply such optimality results to more general contexts.

We use the following notations in our derivations:

- Let $W_{opt} = propagation\ delay * bottleneck\ bandwidth$, which is the number of packets the link can hold.
- Let $W_c = W_{opt} + B$, where $B$ is the buffer size at the bottleneck link. $W_c$ is the total number of packets that the link and the buffer can hold.

## 4.1 Choosing Safe *ssthresh*

Choosing an appropriate initial *ssthresh* is crucial for TCP start-up performance. *ssthresh* determines how long the slow start phase lasts. An overly large *ssthresh* often introduces loss during the slow start phase, which can degrade performance drastically in the following two ways:

1. Multiple losses are likely to occur in slow start phase, which often result in timeouts;
2. Due to the burstiness of the slow start, loss may occur when *cwnd* is much smaller than $W_c$, causing the *ssthresh* to be much lower than $\frac{W_c}{2}$ after recovering from the loss. This can degrade performance drastically as shown in [15].

Therefore, it is important to choose an initial *ssthresh* that prevents loss in the slow start. We call such *ssthresh* a *safe ssthresh*, and derive the largest safe *ssthresh* (denoted as $SS_{max}$).

We use the following network model:

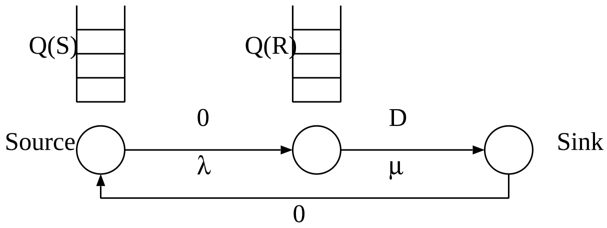


Figure 1: A simplified network model. The source sends data at a rate $\lambda$. The single-server has a rate $\mu$ equal to the slowest server along the path, and a buffer as large as the slowest server's buffer. The delay $D$ is the sum of the delays on the data and ACK path. As justified below, we only need to consider $\mu = 1$. Accordingly, $D$ becomes $W_{opt}$.

As shown in Figure 1, there are two buffers: the source's ready queue and the bottleneck buffer with size $B$. When the link from the source is not fast enough, the buffer at the source may have queue build-up. So the problem is essentially a two-bottleneck problem, and we need to consider the queue build-up in both buffers. This is more complicated than one bottleneck case, which is commonly used by most analyses. The buffer at the source differs from the bottleneck buffer in that it never overflows, because whenever the buffer is full, the sending process is put to sleep. Therefore, we only need to ensure that the bottleneck buffer does not overflow during the slow start.

The bottleneck link has service rate $\mu$, and the sender's sending rate is $\lambda$. We only need to consider $\mu = 1$, because we can always normalize $\mu$ to 1 by choosing the time unit to be $\frac{1}{x}$ seconds, if $\mu = x$ packets/second. Accordingly, the propagation delay becomes $W_{opt}$.

For simplicity, we ignore the transmission time of ACKs. We also ignore the delayed ACK for the moment. Now the slow start phase evolves as follows: starting from the initial window size (1 in standard TCP), the congestion window is increased by one for every ACK, causing two packets to be injected into the network. Since ACKs are separated by at least the bottleneck service time of a packet, the maximum sending rate from the source can not exceed 2, which means $\lambda > 2$ leads to exactly the same performance as $\lambda = 2$. Therefore, we only need to consider $1 \leq \lambda \leq 2$.

We divide the window evolution into *cycles*. *Cycle* $j$ ($j \geq 0$) starts right before the source sends the first of $2^j$ packets sent in the cycle, and ends right before the source receives the acknowledgment of the first packet sent in the cycle.

Let $Q(S_k)$ and $Q(R_k)$ denote the queue length in the source and bottleneck buffer respectively at the beginning of the $k$th cycle. The *critical cycle* is defined as the first cycle when the source's ready queue does not empty out at the beginning of the cycle. More formally, cycle $t$ is *critical* if it satisfies the following properties: (C1) $Q(S_t) = 0$; (C2) $Q(S_{t+1}) > 0$; and (C3) no loss occurs in cycles 0, 1, ... t. At this time, the link from the source becomes a bottleneck.

We start by making the following two observations:

First, the system behavior before the critical cycle is quite simple: The source's ready queue always empties out by the end of each cycle. Therefore, at the beginning of every cycle, all the outstanding packets are either on the link from the bottleneck buffer towards the destination (due to zero propagation delay on the other links) or in the bottleneck buffer. Since the link can only hold a certain number of packets, we can easily compute the queue length at the bottleneck buffer.

Second, after the critical cycle, labeled as $t$, both $Q(S)$ and $Q(R)$ increase monotonically as each new packet gets sent. Suppose the number of packets in the bottleneck buffer at the time $t$ is $Q(R_t)$. After cycle $t$, packets arrive at the bottleneck at rate $\lambda$ and get drained at rate 1. Each packet builds up the queue at the bottleneck buffer by $\frac{\lambda-1}{\lambda}$ packets. Suppose the bottleneck buffer has room for $x$ more packets at the beginning of the cycle $t+1$. Before the bottleneck buffer overflows, at most $x * \frac{\lambda}{\lambda-1}$ more packets can enter it. To ensure no overflow during the slow start, the slow start phase has to end before sending $x * \frac{\lambda}{\lambda-1}$ more packets.

Now it's clear that the key is to locate the critical cycle and find out the queue length at each buffer in the cycle immediately following the critical cycle if any. To do these, we introduce two variables $C$ and $M$.

$C$ satisfies: (C1) $Q(S_C) = 0$; (C2) $Q(S_{C+1}) > 0$ given no loss occurs in cycles 0, 1, ... , $C$. As derived in the Appendix, $C = max(\lfloor log_2(\lambda + \lambda W_{opt})\rfloor + 1, \lfloor log_2 \frac{2\lambda}{2-\lambda}\rfloor + 1)$. By definition, if critical cycle exists in slow start, it must be equal to $C$.

$M$ is defined as the largest integer $m$ satisfying: (M1) $Q(S_m) = 0$; and (M2) no loss occurs in cycles 0, 1, ..., $m$. As shown in the Appendix, we have $M = min(\lfloor log_2(VB)\rfloor, \lfloor log_2 \frac{2VW_c}{V+2}\rfloor, C)$, where $\frac{1}{V} = 1 - \frac{1}{\lambda}$.

Clearly, $M \le C$, so we only need to consider the following two cases:

1. $M < C$: In this case, we have (see Appendix for details of derivation):

$$SS_{max} = min(2^{M+1} - 1, \frac{V}{2} * min(B, W_c - 2^M) + 2^M), \text{ where } \frac{1}{V} = 1 - \frac{1}{\lambda}.$$

2. $M = C$: In this case (see Appendix for details of derivation), $SS_{max}$ is the largest integer satisfying

$$SS_{max} + 2^{\lceil log_2 SS_{max}\rceil} \le VB - VQ(R_{C+1}) - Q(S_{C+1}) + 2^{C+1}$$

where $\frac{1}{V} = 1 - \frac{1}{\lambda}$, $Q(S_{C+1}) = 2^C - \lambda - \lambda * max(W_{opt}, 2^{C-1})$, and $Q(R_{C+1}) = \lambda + \lambda * max(W_{opt}, 2^{C-1}) - W_{opt}$.

A few comments follow: When $\lambda = 2$, we have $SS_{max} = min(W_c, B + 2^{\lfloor log_2 B\rfloor + 1})$, which is consistent with the result given in [15]. However, our model is much more general. When delayed ACK is used, our analysis still applies. The only modification we need to make is to decrease the growth rate of the congestion window to $1 + \frac{1}{b}$ per cycle, where $b$ is the average number of packets acknowledged by an ACK. Consequently, the $SS_{max}$ will be larger than without delayed ACKs.

## 4.2 Choosing Optimal *cwnd*

In this subsection, we derive the optimal initial *cwnd* (denoted as $cwnd_{opt}$) that minimizes the completion time of a given file transfer.

As we know, the slow start algorithm is designed to probe the available network capacity. The throughput during slow start is very low. More specifically, in $k$ RTTs during slow start, no more than $2^k - 1$ packets are sent. Therefore, if the network condition is known and stable, we should try to avoid slow start, and do congestion avoidance directly. This can be achieved by setting the initial *ssthresh* to be no larger than the initial *cwnd*. To find $cwnd_{opt}$, we only need to consider the congestion avoidance phase. Figure 2 (obtained from [12]) depicts roughly how the congestion window and service rate evolve during congestion for TCP/Reno, going through periodical *epochs*. The sender has a loss at the end of each epoch, and recovers from it using fast retransmission.

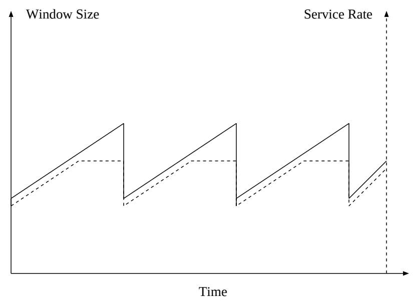


Figure 2: Evolution of service rate and congestion window during congestion avoidance for TCP/Reno

Choosing $cwnd_{opt}$ is essentially the problem of fitting a block in the service rate curve to minimize the transfer time, where the block size is equal to the file size. This is a geometric problem, and can be solved as follows:

First we compute the number of segments sent during each epoch. Let

$S(R)$: the total number of segments sent in region $R$

$T$: the region of an epoch

$F$: a transferred file size

$b$: the average number of packets acknowledged by an ACK

During congestion avoidance phase, in each epoch other than the first and last ones, the window size starts from $\lfloor \frac{W_c+1}{2} \rfloor$ and increases linearly in time, with a slope of $\frac{1}{b}$ packets per round trip time. When the window size reaches $W_c + 1$, a loss occurs. Before the loss is detected by duplicated ACKs, another $W_c$ packets are injected into the network. Then the window drops back to $\lfloor \frac{W_c+1}{2} \rfloor$ and a new epoch begins. Therefore, the total number of packets during an epoch can be computed as $S(T) = b * (\sum_{x=\lfloor \frac{W_c+1}{2} \rfloor}^{W_c} x) + 2 * W_c + 1$.

Next we give a geometric proof to show that we can minimize the transfer time by choosing $cwnd_{opt}$ to be the largest integer $n$ satisfying $b * \sum_{i=n}^{W_c} i \geq P$, where $P = F \ mod \ S(T)$.

Since the duration of an intermediate epoch (period other than the first and last ones) is constant, we can ignore all intermediate epochs and only consider the first and last epochs in order to minimize the completion time. Suppose the transmission regions span in the first and last epochs are $A$ and $B$ respectively. Altogether, there are following three cases:

1. The transfer spans only a single epoch:

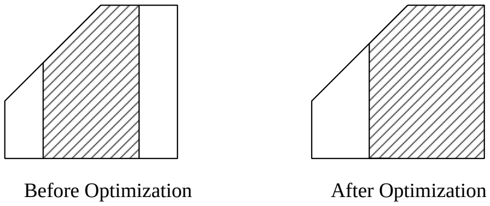


Figure 3: Case 1: Optimizing completion time for a transfer that spans a single epoch

Since the service rate increases monotonically within each epoch, we can minimize completion time by moving the shaded region to the right end as shown in Figure 3.

2. $S(A+B) \leq S(T)$:

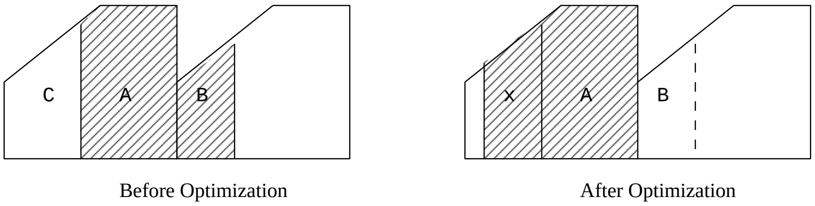


Figure 4: Case 2: Optimizing completion time when $S(A+B) \leq S(T)$

$S(A+B) \leq S(T) \;\Rightarrow\; S(C) \geq S(B)$. Therefore, we can find $x$ within $C$ such that $S(x) = S(B)$. Due to monotonicity, replacing $B$ with $C$ as shown in Figure 4 leads to minimum completion time.

3. $S(A+B) > S(T)$:

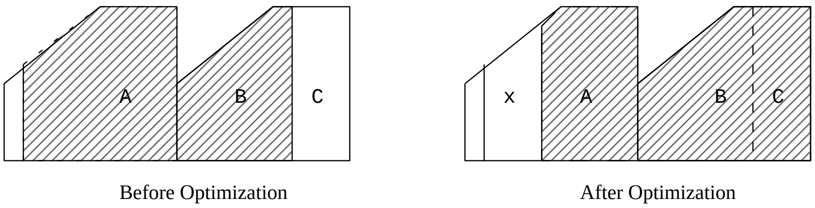


Figure 5: Case 3 : Optimizing completion time when $S(A+B) > S(T)$

$S(A+B) > S(T) \;\Rightarrow\; S(A) > S(C)$. Therefore, we can find $x$ within $A$ such that $S(x) = S(C)$. Again, due to monotonicity, by moving $x$ to $C$, as shown in Figure 5, we get the smallest completion time.

For ease of presentation, we ignore a minor technical detail in all three cases shown above: Moving a block to the end of a epoch, where the window size reaches $W_c + 1$, can result in an extra loss. We take care of this by ensuring that the target region of a block ends at $W_c$ instead of $W_c + 1$.

With some straightforward calculation, we immediately get that the minimum completion time can be achieved when the initial *cwnd* equals the largest integer $n$ satisfying $b * \sum_{i=n}^{W_c} i \geq P$ and $P = F \; mod \; S(T)$, where $b$, $F$, $T$ and $S()$ are defined earlier in this section.

A few comments follow: When $b * \sum_{i=n}^{W_c} i = P^* > P$, we can reduce the initial *cwnd* by $\lfloor \frac{P^* - P}{b*(W_c - cwnd_{opt} + 1)} \rfloor$ without increasing the number of $RTT$s required for a transfer. This allows us to use a smaller (usually much smaller) and *safer* initial *cwnd* to achieve almost the same (slightly larger) completion time. We call this a *shift optimization*.

Moreover, since the service rate becomes flat when $cwnd \geq W_{opt}$, it can be easily shown that if $W_{opt} < n$, we can choose $W_{opt}$ as the initial window size. This allows us to achieve the same minimum completion time and reduce the queuing delay at the bottleneck buffer as well.

To summarize, in this section, we have derived the safe *ssthresh* to prevent loss in the slow start. This can help to remove timeouts caused by multiple losses due to an overly large *ssthresh*, and prevent unnecessary low *ssthresh* in the next epoch caused by a loss occurring at a very small *cwnd*. On the other hand, the performance is optimized when we choose $cwnd_{opt}$ as the initial *cwnd*. In this case, we do congestion avoidance from the very beginning, so initial *ssthresh* can be set to any value no larger than the $cwnd_{opt}$.

# 5 A Virtual Link Model for Network Path and Cross Traffic

The analytical results given in the previous section are based on the assumption that there is a single connection using the network. In order to apply these results to more general contexts, we introduce the *virtual link model*.

For any network connection, the underlying communication path can be abstracted as a *virtual link* dedicated to the connection. The virtual link consists of a bottleneck link and a bottleneck buffer, as illustrated in Figure 6. The effect of cross traffic is modeled as perturbation on the virtual link.

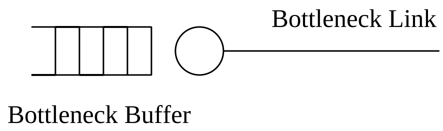


Figure 6: Virtual link model for a communication path. The effect of cross traffic is modeled as perturbation on the virtual link.

The virtual link model allows us to isolate the connection of interest. Once we know the virtual link's characteristics, we can apply our analytical results to choose optimal TCP initial parameters. Therefore, the key is to accurately extract the virtual link characteristics.

The only virtual link characteristic we need is the $W_c$. Intuitively, $W_c$ can be estimated as one segment smaller than the *cwnd* when a loss is detected during congestion avoidance. Our final estimation is the mean of $W_c$ over a connection. To verify this is actually an accurate estimation of $W_c$ even in presence of cross traffic, we run a set of simulations. The simulation topology is illustrated in Figure 7. We tried a number of random distributions for bottleneck link service time of a packet, including exponential distribution, normal distribution, Pareto distribution, etc., as shown in Table 1. We *physically* measure the $W_c$ by measuring the actual service time of the bottleneck link and then compare it with the estimated $W_c$. Our results, summarized in Figure 8, indicate that for a large number of distributions, the estimated $W_c$ is quite accurate (above 85% for all the distributions we use here, except the Pareto distribution, where $W_c$ varies widely and hence the mean $W_c$ is unreliable.) Moreover, the estimated $W_c$ is usually less that the actual $W_c$, which is conservative.

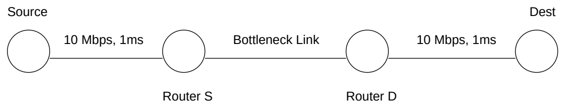


Figure 7: Simulation topology. Bottleneck buffer=20 packets; bottleneck link propagation delay=50 ms; Bottleneck service time for a packet is based on different random distributions.

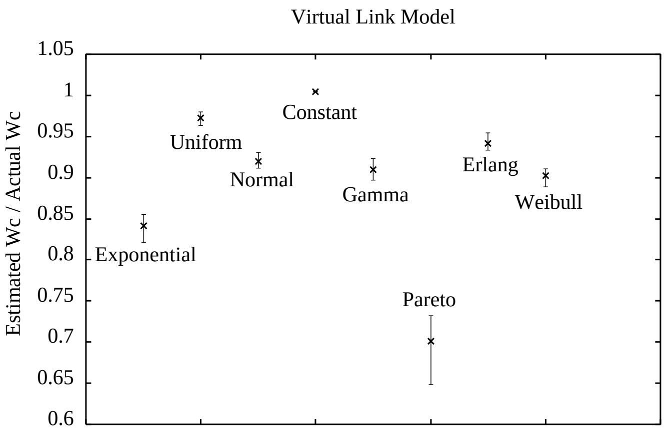


Figure 8: Estimated $W_c$ vs. actual $W_c$ with different distributions for the bottleneck's service time. The parameters for the distributions are given in Table 1. Bars show 90% confidence intervals.

| Model | Parameters | | Model | Parameters |
|---|---|---|---|---|
| Exponential | $mean = 0.005$ | | Gamma | $shape\ k = 2.5$, $mean = 0.005$ |
| Uniform | $min = 0.0025$, $max = 0.0075$ | | Erlang | $shape\ k = 5$, $mean = 0.005$ |
| Normal | $mean = 0.005$, $std = 0.005$ | | Pareto | $mean = 0.005$, $shape\ \alpha = 1.5$ |
| Constant | $value = 0.005$ | | Weibull | $scale\ a = 0.005$, $shape\ b = 1.5$ |

Table 1: Different random distributions used to generate bottleneck's service time for each packet (unit: second)

# 6 An Efficient Network Performance Discovery Architecture

As shown in the previous section, an end host can extract the characteristics of a virtual link by observing its performance. However measurements from a single host can sometimes be redundant, inaccurate, or out-of-date. The study of Internet traces shows "nearby" hosts experience similar or identical throughput performance within a time period measured in minutes [18] [7]. This suggests sharing performance information across hosts can help to determine the network performance more accurately. Based on the observation, [22] proposed an architecture called SPAND (Shared PAssive Network Discovery) that determines network characteristics by making shared, passive measurements from a collection of hosts. The main idea is as follows: Clients report network statistics to a performance server by submitting *Performance Reports*, and the performance server aggregates collected information to predict current network condition.

The idea of making *shared* and *passive* measurements from a collection of hosts, as introduced by SPAND, is very useful. It allows us to explore the temporal and spatial locality of the network characteristics without introducing additional wide-area traffic. However directly re-using SPAND architecture in our context imposes a number of problems. First, in the original SPAND architecture, the sender needs to generate a separate message for each performance report, which is very inefficient. Second, modifying the sender's TCP stack to generate these performance reports is not easy. Finally, to query information in the performance server, a client needs to send an explicit performance request, which incurs overhead and extra latency.

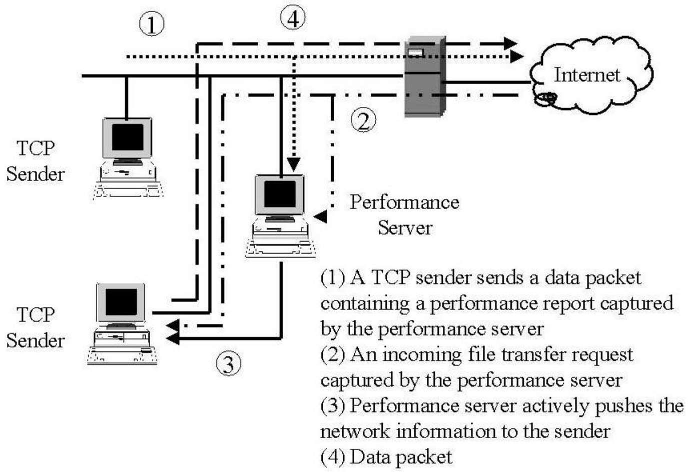


Figure 9: NewSPAND: an efficient network performance discovery architecture

To address these issues, we propose a new performance discovery architecture called NewSPAND. As illustrated in Figure 9, our architecture consists of TCP senders and a performance server that is able to capture packets. Compared to the original SPAND, our NewSPAND architecture has the following two major innovations:

- Rather than generating a separate packet for each performance report, a TCP sender piggybacks the report to its next data packet by using a new TCP option, which can be simply ignored by TCP receivers. After capturing a packet with the new TCP option turned on, the performance server extracts the performance information. This allows the TCP sender to report network statistics in an efficient and clean way.

- Unlike the original SPAND architecture, where the performance server passively receives inquiries from clients and generates a performance response, NewSPAND introduces the *active push* mechanism: Whenever the performance server sees an incoming file transfer request (e.g. an HTTP GET request), which indicates a TCP sender is going to need the network information, it actively pushes its estimated network performance to the TCP sender by sending a TCP packet with a special option flag. This reduces the latency and overhead incurred by the inquiry packets.

Thus NewSPAND removes the overhead in generating and accessing performance reports while retaining all the benefits of original SPAND. To effectively use it for sharing information among different connections, we need to address the following issues:

- *What to report:*
  The performance report includes $W_c$ and *srtt*, where $W_c$ is used to compute the optimal *cwnd*, and *srtt* is needed to smoothly send the packets in the initial window, which will be discussed in more details later.
- *When to report:*
  To timely update the information at the performance server, a TCP sender sends the performance report whenever *cwnd* decreases.
- *How to aggregate the reports:*
  The performance server does both spatial and temporal aggregation to predict the future network condition. Spatially, it aggregates reports that share the same source and destination subnet address. In order to do temporal aggregation, the performance server keeps a fixed number of bins (3 in our simulations), where the performance reports are stored based on the time at which they are received. Each bin contains 100 seconds of reports in our simulation. The server computes for each bin the median of all reports on some network characteristics. These medians are further averaged to predict the current network condition. Moreover, when there are not enough recent performance reports, the performance server informs the sender to fall back to its default restarting behavior.

To summarize, in this section we have designed an efficient network performance discovery architecture, which allows us to share network performance information across hosts in the same domain. With this architecture, we can accurately extract the characteristics of the virtual link between two Internet domains.

# 7 Leaky Bucket-based Pacing

A common criticism of using a large initial congestion window and avoiding slow start is that it can result in a large initial burst overflowing the bottleneck buffer. We deal with this with a pacing scheme that allows the sender to smoothly pace out all segments in its initial window. Our scheme combines leaky bucket flow control with traditional window-based flow control as used in TCP. More specifically, a TCP sender uses a leaky bucket to shape its outgoing traffic. When the sender has a packet to send, it needs to first check the token bucket. If there are sufficient tokens, the packet is sent immediately, otherwise, it is delayed until there are enough tokens.

The depth of the token bucket can be configured to limit the burstiness of the outgoing traffic. It is set to 4 segments in our simulations. (Note that even standard TCP with delayed ACKs can send out 3 segments back to back during slow start.)

The token filling rate limits the average sending rate of the sender. In our scheme, it is set to $\frac{ssthresh}{srtt}$ during slow start phase and $\frac{cwnd}{srtt}$ during congestion avoidance phase, and will be re-adjusted whenever there is a change in *cwnd* or *srtt*. Note that in slow start phase we set the rate differently from congestion avoidance phase. This is because in slow start phase, the congestion window is increased every time a new ACK is received. If we simply set the rate to $\frac{cwnd}{srtt}$, the sender may send out less than *cwnd* segments in one RTT due to the lower token filling rate in the beginning of the cycle. Setting the rate to $\frac{ssthresh}{srtt}$, the target rate of slow start phase, solves the problem. Note that this won't make our scheme more bursty than the standard TCP because we still use the ACK clocking to adjust the congestion window.

Neither leaky bucket nor pacing is a new idea. Leaky bucket has been widely studied in ATM context. Pacing has been proposed in fast start [20] (implemented in ns [17]) and rate-based pacing (RBP) [24] to solve the same problem we mentioned here. However, incorporating leaky bucket into TCP is new and can potentially solve TCP's well-known problem of being too bursty. Note that pacing schemes proposed in fast start and RBP can not be used to solve TCP's "bursty" problem. In fast start, the sender uses a parameter **maxburst** to limit the size of any burst it sends out. However, it doesn't work when there is ACK compression, because many closely spaced ACKs can trigger a row of closely spaced bursts, each of which is smaller than **maxburst**, but as a whole can form a large burst. In RBP, the only segments paced out are those in the first congestion window after a connection goes idle for a long time. Upon receiving the first new ACK, the *cwnd* is re-adjusted (usually decreased) to the actual number of packets in flight. Clearly, this can not be used solve TCP's "bursty" problem. In addition, since *cwnd* is re-adjusted (usually decreases) upon receiving the first ACK, it does not fit for our purpose of pacing out the $cwnd_{opt}$ segments.

Since the token filling rate is inversely proportional to *srtt*, we need a relatively accurate estimation of $RTT$, which can not be achieved by the 200 ms or 500 ms timer granularity in standard TCP. In our simulations, we use a 50 ms timer. If the *timestamp* option is available, it can be used to further improve the accuracy of RTT estimation. We also need a fine-grained timer to reschedule a segment for later transmission in case there is no sufficient tokens.

There is evidence to suggest that the overhead of software timers is not likely to be significant with modern processor technologies. ( [10] reports an overhead of the order of a few microseconds.) Moreover, the timer overhead is unlikely to be a significant addition to the cost of taking interrupts and processing ACKs that goes with ACK clocking [20].

Currently, we are using leaky bucket based pacing scheme only when there is a large burst to send. As part of our future work, we will further study the effectiveness of our pacing scheme in reducing the burstiness of TCP traffic and its potential applications in the context of *differentiated service*.

Now we have derived our final scheme, called **TCP/SPAND**, for initializing TCP parameters. In this scheme,

- TCP senders of the same domain share the network performance information using NewSPAND architecture (Section 6) to extract the characteristic (i.e. $W_c$) of the virtual link (Section 5) between two Internet domains.
- We initialize *cwnd* to $cwnd_{opt}$ computed by applying our theoretical results (Section 4) based on the virtual link property.
- We employ the leaky bucket based pacing scheme (Section 7) to smoothly send out the packets in the initial

| Scenario | Bandwidth | Link Delay | Descriptions |
|---|---|---|---|
| 1 | 1.6 Mbps | 50 ms | typical terrestrial WAN links with close to T1 speed |
| 2 | 1.6 Mbps | 200 ms | typical geostationary satellite links with close to T1 speed |
| 3 | 45 Mbps | 200 ms | typical geostationary satellite links with T3 speed |

Table 2: Different simulation scenarios

window.

# 8 Simulation Results

## 8.1 Methodology

We implement TCP/SPAND in the **ns** network simulator [17] to evaluate its effectiveness. Our implementation is based on TCP NewReno [13], a variant of TCP that uses partial new ACK information to recover from multiple packet losses in a window at the rate of one per RTT. We implement a variant of NewSPAND scheme described in Section 6 with the same expected performance.

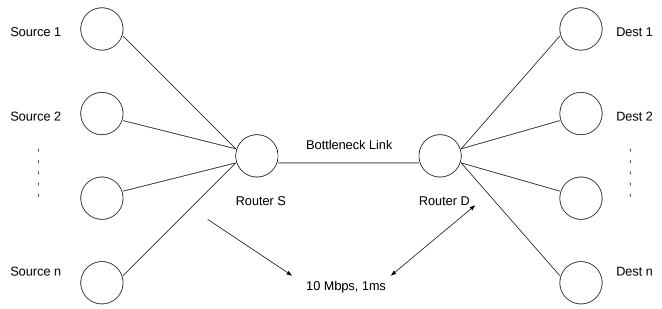


Figure 10: Simulation topology. Bottleneck buffer is 10 KB. The settings for bottleneck link are summarized in Table 2.

The topology used for the simulation is shown in Figure 10. One or more bursty connections are established between a subset of the sources on the left and sinks on the right.

The maximum window size of all the TCP connections (including ftp used as cross traffic) is set to 100 KB. The bottleneck buffer is 10 KB. The bottleneck link router uses FIFO scheduling and drop-tail buffer management. The TCP segment size is set to 1 KB. We consider three scenarios shown in Table 2.

We set up our experiments in the following way to mimic the behavior of Web transfers: Each experiment consists of 40 rounds. In each round every sender transfers one file with start time uniformly distributed around a central point by a time interval (denoted as *jitter*). Between each round there is 120 seconds of idle time.

We use the average completion time of file transfers as our performance metric. For each simulation configuration, we report the mean of 10 runs of an experiment.

## 8.2 Performance Evaluation

We compare the performance of TCP/SPAND with the following four variants of TCPs:

- Reno with slow start restart (reno-ssr): TCP/Reno which enforces slow start when restarting data flow after an idle period.
- Reno without slow start restart (reno-nssr): TCP/Reno which reuses the prior congestion window upon restarting after an idle period (This is the scheme used in SunOS.).
- NewReno with slow start restart (newreno-ssr): TCP/NewReno with restart behavior similar to reno-ssr.
- NewReno without slow start restart (newreno-nssr): TCP/NewReno with restart behavior similar to reno-nssr.

All the TCP protocols in our experiments assume T/TCP-style accelerated open that avoids 3-way handshaking at connection setup. Moreover, in order to remove the performance difference due to different timer granularities, all protocols use the same timer granularity of 50 ms.

#### 8.2.1 Varying the number of competing connections

First we compare performance as the number of competing connections varies from 1 to 30 while transfer size is kept at 40 KB, a typical Web transfer size. Five low-bandwidth telnet sessions compete with the main flows to help avoid deterministic behavior.

Figure 11 shows the results for two different scenarios where the bottleneck link is a T1 link (1.6 Mbps) with a latency of 50 ms or 200 ms (Scenario 1 and 2 in Table 2). We make two observations. First, TCP/SPAND leads to a significant improvement in completion time. In Scenario 1 (shown in Figure 11(a)), TCP/SPAND reduces average completion time by $35\% - 60\%$ compared to reno-ssr and newreno-srr; by $40\% - 50\%$ compared to reno-nssr; by $25\% - 50\%$ compared to newreno-nssr. The improvement is even larger in Scenario 2 (shown in Figure 11(b)) due to the much larger bandwidth delay product: 50% to 70% improvement over reno-ssr and newreno-ssr, and 40% to 60% improvement over reno-nssr and newreno-nssr.

Second, the performance improvement tends to decrease as the number of connections increases. This is expected because as connection number increases, each connection's share of $W_c$ decreases, so that the impact of slow start phase on TCP performance becomes less significant.

#### 8.2.2 Varying transfer size

Second, we compare performance as the transfer size varies. Figure 12 summarizes the results. In all cases, TCP/SPAND reduces the completion time over wide range of transfer size. In percentage terms, the improvement decreases as the transfer size increases. This is what we would expect, since the start-up performance is very crucial for small transfer, while it is no longer so significant for large transfer size. Moreover, the improvement increases with link delay due to the larger bandwidth delay product.

#### 8.2.3 Cross traffic

Finally, we compare performance under cross traffic in the following two ways: (i) use multiple ftp sessions as cross traffic; and (ii) model the effect of cross traffic on the bottleneck's service time by different distributions.

As before, we first evaluate performance as the number of competing connections varies, and the transfer size is kept at 40 KB. The simulation results are illustrated in Figure 13. In both cases, a number of ftp connections

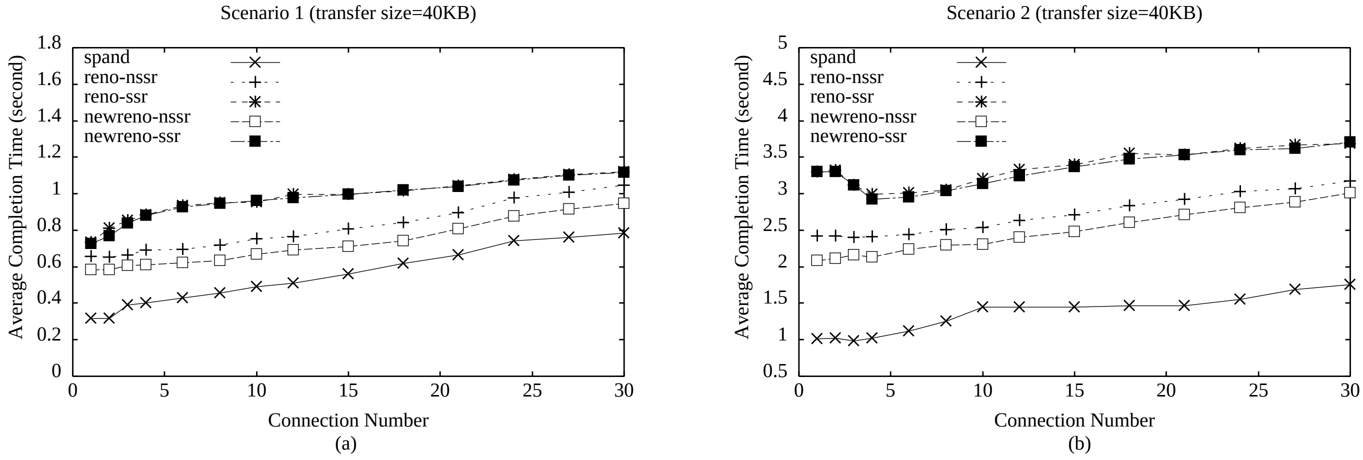


Figure 11: Performance comparison for different number of connections. Bottleneck link has setting of either Scenario 1 or Scenario 2 (defined in Table 2). In each round, file transfers start within 10 seconds (i.e. jitter=10 sec). 5 telnet sessions are used to avoid deterministic behavior.

make the bottleneck link heavily loaded. Figure 13(a) shows the results of using 2 ftp sessions as cross traffic to create contention for T1 link with 200 ms latency. TCP/SPAND reduces average completion time by 20% to 45% compared to reno-ssr and newreno-ssr. The improvement is smaller compared to reno-nssr and newreno-nssr. For large connection number (greater than 20), reno-nssr, newreno-nssr and TCP/SPAND perform similarly due to the small share of $W_c$. The performance improvement is more significant when the bottleneck link is T3 as shown in Figure 13(b).

Figure 14 show the results of varying transfer size and keep the connection number constant. As we can see, the performance benefit of TCP/SPAND is greater for large bandwidth.

We also model the effect of cross traffic on the bottleneck's service time by different random distributions. The results are summarized in Figure 15. In all six cases, TCP/SPAND consistently outperforms other four TCP schemes.

## 8.3 TCP Friendliness

In this section, we demonstrate the performance improvement of TCP/SPAND does not come at the expense of degrading the performance of the connections using standard TCP. In other words, TCP/SPAND is TCP friendly.

We show this by considering a mixture of TCPs. One half of the connections use TCP/SPAND, while an equal number use reno-ssr, one of the least aggressive TCP schemes. We then compare their performance with the case in which all connections use reno-ssr. The timer granularity used in all reno-ssr is 200 ms to ensure even the reno-ssr with coarse timer granularity is unaffected by TCP/SPAND, which employs fine-grained timers. The jitter for transfer start time during each round is set to 0.1 second to create big contention for the bottleneck bandwidth. Again 5 telnet sessions are introduced to avoid deterministic behavior.

Figure 16 summarizes the simulation results for four different settings, where the bottleneck link is T1 link with latency of either 50 ms (Scenario 1) or 200 ms (Scenario 2), and the transfer size is 40 KB or 100 KB.

As expected, in all four settings, TCP/SPAND outperforms reno-ssr. Meanwhile, reno-ssr experiences little or

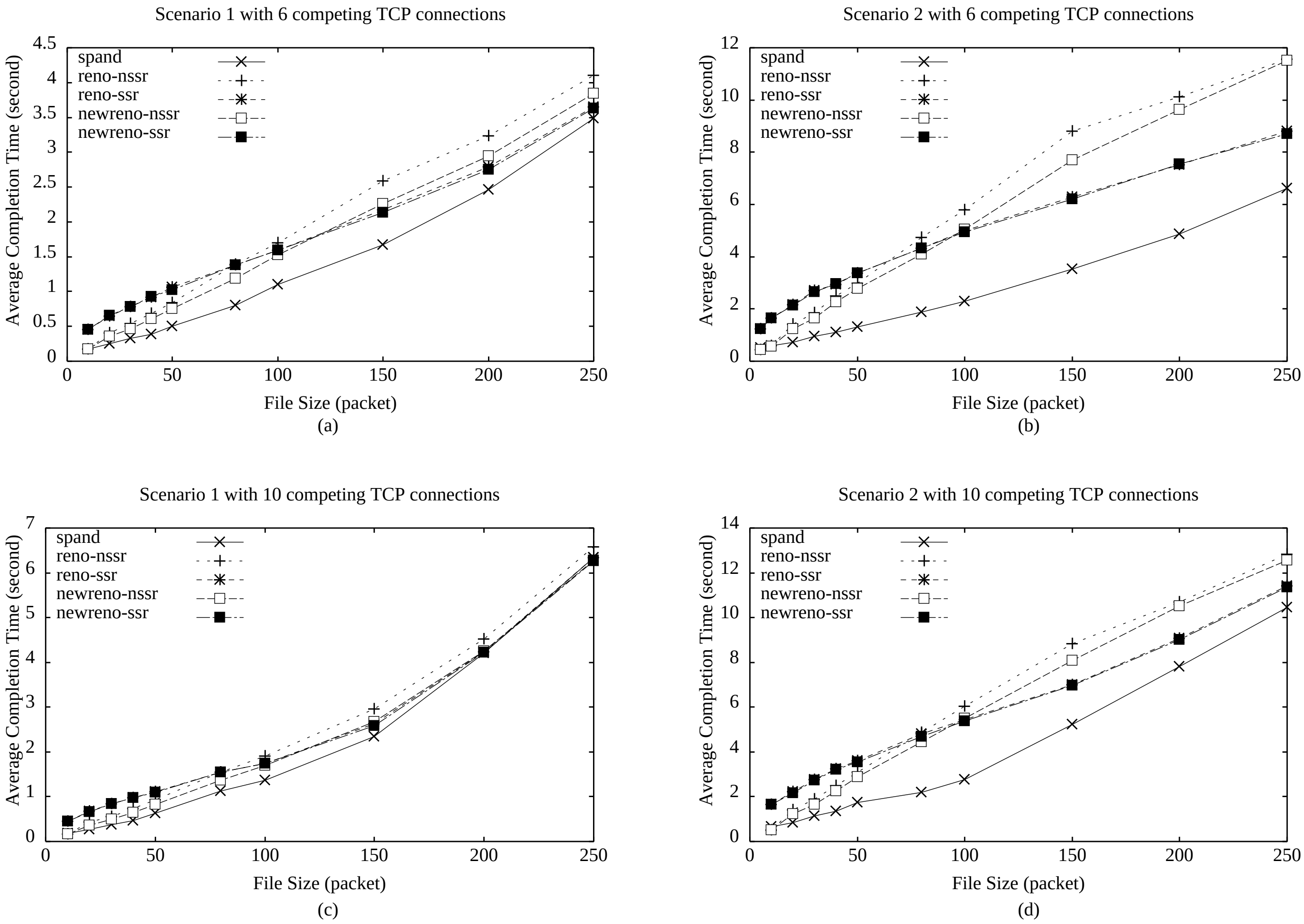


Figure 12: Performance comparison for different transfer sizes. Either 6 or 10 TCP connections competes for the bottleneck link with setting of either Scenario 1 or Scenario 2 (defined in Table 2). In each round, file transfers start within 10 seconds (i.e. jitter=10 sec). 5 telnet sessions are used to avoid deterministic behavior. jitter=10

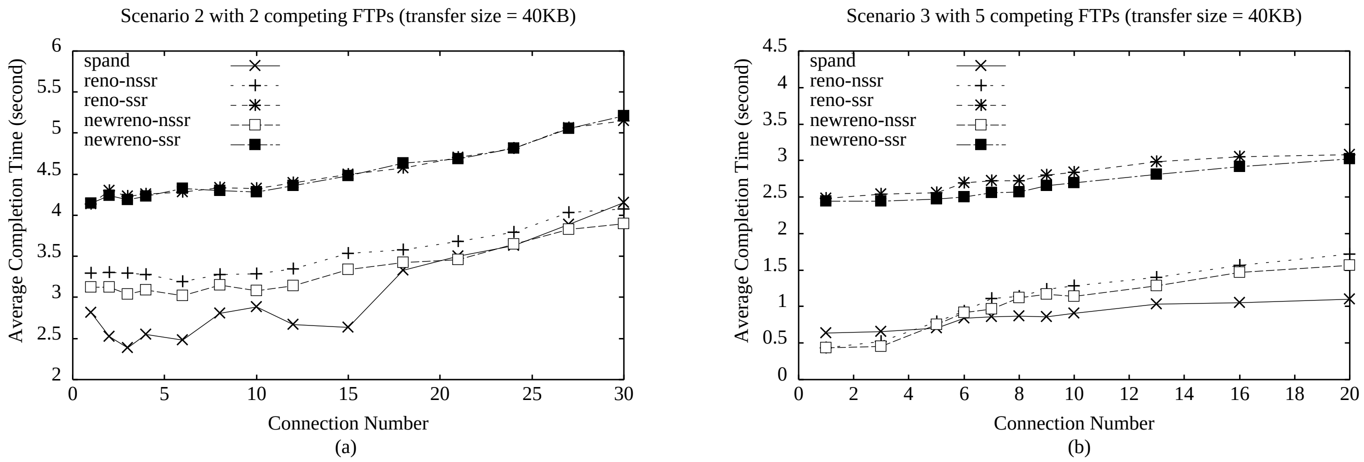


Figure 13: Performance comparison for different number of connections with FTPs and 5 telnet sessions as cross traffic. In (a), the jitter for transfer start time in each round is 10 sec; while in (b), the jitter is 0.1 sec.

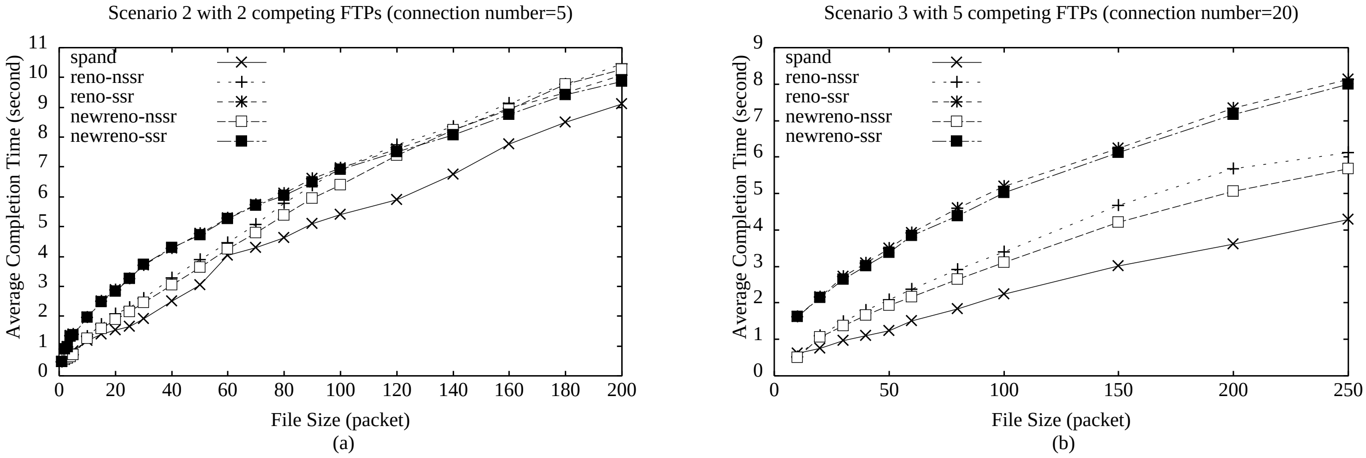


Figure 14: Performance comparison for different transfer sizes with FTP as cross traffic. Settings are the same as in Figure 13.

no performance degradation in presence of TCP/SPAND (consistently less than 5% in all cases we have studied.) This can be explained by the following three reasons. First, with NewSPAND architecture, the sender can have a quite accurate estimation of current network condition. Second, the $cwnd_{opt}$ computed based on our analysis is usually much smaller than the estimated $W_c$, especially when *shift optimization* (described in Section 4.2) is used. This is conservative. Finally, the leaky bucket-based pacing scheme effectively reduces the burstiness of TCP upon start-up and restart.

What we didn't expect is that in many cases, the performance of reno-ssr is improved in the presence of TCP/SPAND. This can be partly explained by the fact that TCP/SPAND can utilize available bandwidth more efficiently and thus can finish transmission very fast, leaving the whole capacity of the bottleneck link to those connections using reno-ssr.

To summarize, TCP/SPAND improves performance significantly without degrading the performance of unenhanced TCP.

# 9 Conclusions and Future Work

In this paper, we have demonstrated how we can improve TCP start-up performance by passive monitoring of current network performance. Our key contributions include:

- Analyze TCP start-up dynamics and derive optimal TCP initial parameters as a function of link characteristics;
- Abstract a communication path as a virtual link, and model the effect of cross traffic as perturbation on it;
- Propose an efficient architecture for network performance discovery based on the idea of shared, passive network monitoring;
- Design a new pacing scheme that can effectively reduce TCP's burstiness at connection start-up and restart or after a loss.

The effectiveness of TCP/SPAND demonstrates that introducing external information into the protocol can help to optimize protocol performance. As part of our future work, we will further explore how to apply this general

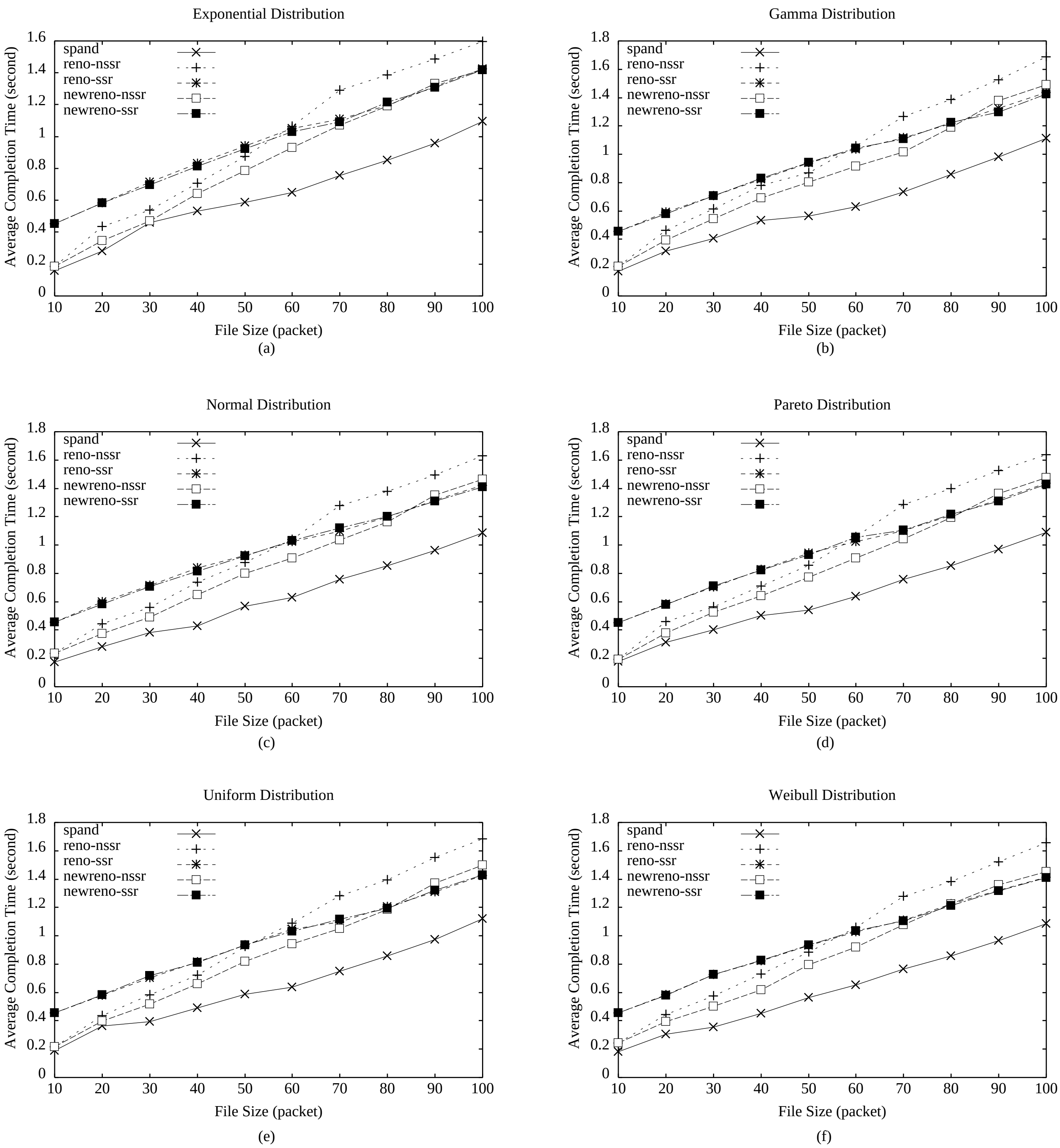


Figure 15: Model the effect of cross traffic on bottleneck link's service time with different distributions whose parameters are given in Table 1

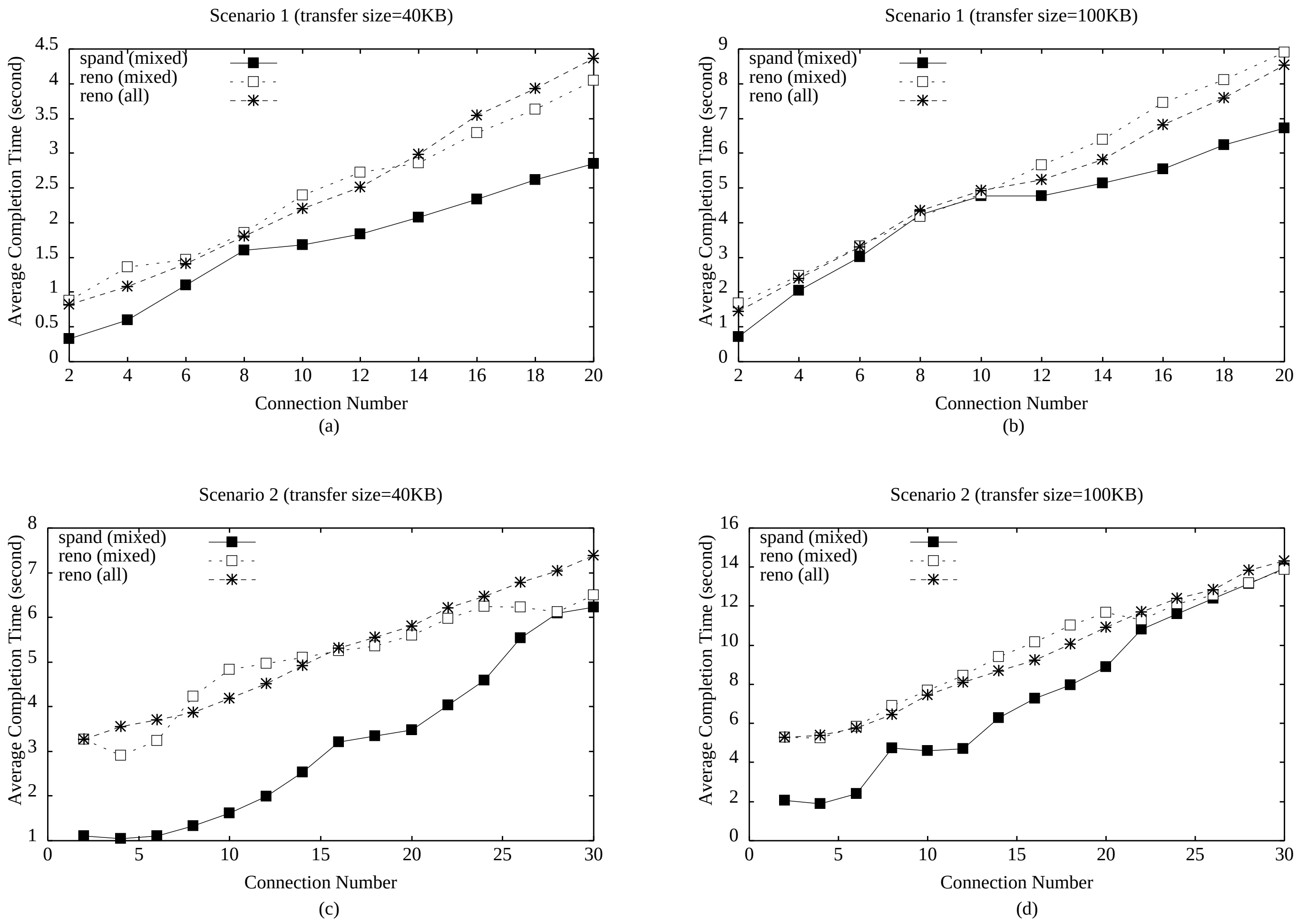


Figure 16: TCP friendliness. 5 telnet sessions are used as background traffic. Transfer start time in each round has a jitter of 0.1 sec. Transfer size is either 40 KB or 100 KB.

methodology in different network protocols. Another interesting direction we are looking at is how to incorporate our pacing scheme in integrated services network. As pointed out in [10], TCP does not cooperate well with the leaky-bucket typed traffic policer due to its burstiness. Our pacing scheme can potentially help to solve the problem in an effective and clean way. We will also investigate how to improve temporal and spatial aggregation at the performance server. Our final plan is to implement TCP/SPAND in real systems.

## 10 Acknowledgement

We would like to thank Cristian Estan for his helpful comments and inputs.

## References

[1] M. Allman, S. Dawkins, D. Glover, J. Griner, T. Henderson, J. Heidemann, H. Kruse, S. Ostermann, K. Scott, J. Semke, J. Touch, and D. Tran. Ongoing TCP Research Related to Satellites. Internet Draft draft-ietf-tcpsat-res-issues-05.txt, November 1998.

[2] M. Allman, D. R. Glover, and L. A. Sanchez. Enhancing TCP Over Satellite Channels using Standard Mechanisms. RFC-2488, September, 1998.

[3] M. Allman, S. Floyd, and C. Partridge. Increasing TCP's Initial Window. RFC-2414, September 1998.

[4] M. Allman and C. Hayes. An Evaluation of TCP with Larger Initial Windows. ACM Computer Communication Review, July 1998

[5] R.T. Braden. Extending TCP for Transactions - Concepts. RFC-1379, November 1992.

[6] R.T. Braden. T/TCP - TCP Extensions for Transactions Functional Specification. RFC-1644, July 1994.

[7] H. Balakrishnan, S. Seshan, M. Stemm, and R. H. Katz. Analyzing Stability in Wide-Area Network Performance. In *Proc. ACM SIGMETRICS '97*, 1997.

[8] H. Balakrishnan, V. N. Padmanabhan, S. Seshan, M. Stemm, and R. H. Katz. TCP Behavior of a Busy Internet Server: Analysis and Improvements. In *Proc. IEEE INFOCOM '98*, March 1998.

[9] S. Floyd and K. Fall. Promoting the Use of End-to-End Congestion Control in the Internet. Submitted to IEEE Transactions on Networking. (available from http://www-nrg.ee.lbl.gov/floyd/papers.html)

[10] W. Feng, D. D. Kandlur, D. Saha, and K. G. Shin. Understanding TCP Dynamics in an Integrated Services Internet. 7th International Workshop on Network and Operating Systems Support for Digital Audio and Video, May 1997.

[11] V. Jacobson and M. Karels. Congestion Avoidance and Control. In *Proc. ACM SIGCOMM 1988*, August 1988.

[12] D. P. Heyman, T.V. Lakshman, and A. L. Neidhardt. A New Method for Analyzing Feedback-Based Protocols with Applications to Engineering Web Traffic over the Internet. In *Proc. ACM SIGMETRICS '97*, 1997.

[13] J. C. Hoe. Improving the Start-up Behavior of a Congestion Control Scheme for TCP. In *Proc. ACM SIGCOMM '96*, August 1996.

[14] A. Hughes, J. Touch, and J. Heidemann. Issues in TCP Slow-Start Restart After Idle. Internet Draft draft-ietf-tcpimpl-restart-00.txt, March 1998.

[15] T.V. Lakshman and U. Madhow. Performance Analysis of Window-Based Flow Control using TCP/IP: the Effect of High Bandwidth-Delay Products and Random Loss. IFIP Transactions C-26, High Performance Networking V, pp. 135-150, North-Holland, 1994.

[16] B. A. Mah. An Empirical Model of HTTP Network Traffic. In *Proc. INFOCOM '97*, 1997.

[17] UCB/LBNL/VINT Network Simulator - ns (version 2). http://www-mash.cs.berkeley.edu/ns, 1997.

[18] V. Paxson. Measurements and Analysis of End-to-End Internet Dynamics. PhD thesis, U.C. Berkeley, May 1996.

[19] V.N. Padmanabhan. Addressing the Challenges of Web Data Transport. Ph.D. Thesis, UC Berkeley, 1998.

[20] V.N. Padmanabhan and R. Katz. TCP Fast Start: A Technique for Speeding Up Web Transfers. In *Proc. IEEE Globecom '98 Internet Mini-Conference*, Sydney, Australia, November 1998.

[21] V.N. Padmanabhan and J.C. Mogul. Improving HTTP Latency. In *Proc. Second International World Wide Web Conference*, October 1994.

[22] S. Seshan, M. Stemm, and R. H. Katz. SPAND: Shared Passive Network Performance Discovery. In *Proc 1st Usenix Symposium on Internet Technologies and Systems (USITS '97)*, Monterey, CA, December 1997

[23] J. Touch. TCP control block interdependence. RFC-2140, April 1997.

[24] V. Visweswaraiah and J. Heidemann. Improving Restart of Idle TCP Connections. Technical report 97-661, University of Southern California, November 1997.

# 11 Appendix

Here we fill in the details of the derivation of safe *ssthresh* in Section 4.1.

## 11.1 Derivation of $C$ (as defined in Section 4.1)

Given no loss occurs in cycles 0, 1, ..., C, derive $C$ such that (C1) $Q(S_C) = 0$; (C2) $Q(S_{C+1}) > 0$.

Suppose cycle $C$ lasts for time $T_C$. By definition, $T_C$ includes the transmission time of the first packet at the bottleneck link, the queuing delay at the time of the transmission, and the propagation delay. That is, $T_C = 1 + Q(S_C) + Q(R_C) + propagation\ delay$, where $Q(S_C) = 0$, $Q(R_C) = max(0, 2^{C-1} - W_{opt})$, and $propagation\ delay = W_{opt}$. So $T_C = 1 + max(W_{opt}, 2^{C-1})$.

During $T_C$, altogether $2^C$ packets enter the source's ready queue, among which $\lambda * T_C$ packets get served. Hence $Q(S_{C+1}) = max(0, 2^C - \lambda T_C)$. Therefore, we have:

$$Q(S_{C+1}) > 0 \Longleftrightarrow 2^C - \lambda * T_C > 0 \Longleftrightarrow 2^C - \lambda * (1 + max(W_{opt}, 2^{C-1})) > 0$$

This gives $C \geq max(\lfloor log_2(\lambda + \lambda W_{opt}) \rfloor + 1, \lfloor log_2 \frac{2\lambda}{2-\lambda} \rfloor + 1)$. By (C1) and (C2), $C$ is the smallest integer satisfying $Q(S_{C+1}) > 0$. Hence $C = max(\lfloor log_2(\lambda + \lambda W_{opt}) \rfloor + 1, \lfloor log_2 \frac{2\lambda}{2-\lambda} \rfloor + 1)$.

## 11.2 Derivation of $M$ (as defined in Section 4.1)

$M$ is defined as the largest integer $k$ satisfying:

- (M1) $Q(S_k) = 0$;
- (M2) no loss occurs in cycles 0, 1, ..., $k$.

From (M1), we immediately have $M \leq C$.

In cycle $k$, $2^k$ packets enter the bottleneck buffer. Each packet builds up the queue at the bottleneck buffer by $1 - \frac{1}{\lambda} \equiv \frac{1}{V}$. Altogether these $2^k$ packets reduce available buffer at the bottleneck by $2^k(1 - \frac{1}{\lambda})$ packets. To ensure no overflow, we need

$$2^k(1 - \frac{1}{\lambda}) \leq B - Q(R_k) = B - max(0, 2^{k-1} - W_{opt}) = min(B, W_c - 2^{k-1})$$

Hence $k \leq min(\lfloor log_2(VB) \rfloor, \lfloor log_2 \frac{2VW_c}{V+2} \rfloor)$. By definition, $M$ is the maximum number satisfying the above conditions. This, combined with $M \leq C$, gives $M = min(\lfloor log_2(VB) \rfloor, \lfloor log_2 \frac{2VW_c}{V+2} \rfloor, C)$.

## 11.3 Analysis of Safe $ssthresh$

Let's consider the following two cases:

### 11.3.1 Case 1: $M < C$

In this case, no critical cycle exists during slow start. Suppose by way of contradiction that the critical cycle does exist. By definition of $M$, either (M1) or (M2) does not hold for $M + 1$, otherwise $M$ can not be the *largest*. Since $M + 1 \leq C$, (M1) must hold. Therefore, (M2) can not hold for $M + 1$, i.e., loss must have occurred in cycle $M + 1$. This contradicts with (C3)!

To ensure zero loss during slow start, by induction, we only need to ensure zero loss in the last two cycles. In other words, a safe $ssthresh$ must satisfy

$$\begin{cases} \lfloor log_2 ssthresh \rfloor \leq M & \text{Condition 1} \\ 2 * (ssthresh - 2^{\lfloor log_2 ssthresh \rfloor}) * \frac{1}{V} \leq min(B, W_c - 2^{\lfloor log_2 ssthresh \rfloor}) & \text{Condition 2} \end{cases}$$

where $\frac{1}{V} = 1 - \frac{1}{\lambda}$. Condition 2 is based on the observation that during the last cycle in the slow start, each of the first $ssthresh - 2^{\lfloor log_2 ssthresh \rfloor}$ ACKs triggers the source to send out two new packets. These packets build up the queue at the bottleneck buffer by a total of $2 * (ssthresh - 2^{\lfloor log_2 ssthresh \rfloor}) * \frac{1}{V}$ packets, which must be less than the available buffer at the beginning of the cycle (i.e. $min(B, W_c - 2^{\lfloor log_2 ssthresh \rfloor})$). Afterwards, $cwnd \geq ssthresh$ and slow start phase ends.

Notice that $2^M$ is *safe*. Hence, by definition, $SS_{max} \geq 2^M$, that is, $\lfloor log_2 SS_{max} \rfloor \geq M$. This, combined with Condition 1, immediately gives $\lfloor log_2 SS_{max} \rfloor = M$. Replacing $\lfloor log_2 SS_{max} \rfloor$ with $M$ in Condition 2, we have

$$SS_{max} = min(2^{M+1} - 1, \frac{V}{2} * min(B, W_c - 2^M) + 2^M)$$

### 11.3.2 Case 2: $M = C$

Clearly, $2^C$ is safe, so $SS_{max} \geq 2^C$. If $SS_{max} > 2^C$, then critical cycle exists during slow start. Since $M = C$, there is no loss before and including critical cycle $C$. Therefore, we only need to ensure no loss occurs after the critical cycle $C$. After the cycle $C$, *cwnd* grows from $2^{C+1}$ to $SS_{max}$. The total number of packets sent during this period, including $Q(S_{C+1})$ packets in the source queue at the beginning of cycle $C+1$, is

$$Q(S_{C+1}) + 2^{C+1} + 2^{C+2} + ... + 2^{\lceil log_2 SS_{max} \rceil - 1} + SS_{max} = Q(S_{C+1}) + 2^{\lceil log_2 SS_{max} \rceil} + SS_{max} - 2^{C+1}.$$

Hence the maximum queue build-up in the bottleneck buffer is $\frac{1}{V} * (Q(S_{C+1}) + 2^{\lceil log_2 SS_{max} \rceil} + SS_{max} - 2^{C+1})$. We need $max\ build\_up \leq B - Q(R_{C+1})$, which gives us

$$SS_{max} + 2^{\lceil log_2 SS_{max} \rceil} \leq VB - VQ(R_{C+1}) - Q(S_{C+1}) + 2^{C+1}$$

where $\frac{1}{V} = 1 - \frac{1}{\lambda}$, $Q(S_{C+1}) = 2^C - \lambda - \lambda * max(W_{opt}, 2^{C-1})$, and $Q(R_{C+1}) = 2^C - Q(S_{C+1}) - W_{opt} = \lambda + \lambda * max(W_{opt}, 2^{C-1}) - W_{opt}$. So the final $SS_{max}$ is the largest integer satisfying this condition.