TCP Startup Performance in Large Bandwidth Delay Networks

Ren Wang, Giovanni Pau, Kenshin Yamada, M.Y. Sanadidi, and Mario Gerla
Computer Science Department
University of California, Los Angeles
Los Angeles, CA 90095, USA
{renwang,gpau,kenshin,medy,gerla}@cs.ucla.edu

Abstract — **Next generation networks with large bandwidth and long delay pose a major challenge to TCP performance, especially during the startup period. In this paper we evaluate the performance of TCP Reno/Newreno, Vegas and Hoe's modification in large bandwidth delay networks. We propose a modified Slow-start mechanism, called Adaptive Start (Astart), to improve the startup performance in such networks. When a connection initially begins or re-starts after a coarse timeout, Astart adaptively and repeatedly resets the Slow-start Threshold (*ssthresh*) based on an eligible sending rate estimation mechanism proposed in TCP Westwood. By adapting to network conditions during the startup phase, a sender is able to grow the congestion window (*cwnd*) fast without incurring risk of buffer overflow and multiple losses. Simulation experiments show that Astart can significantly improve the link utilization under various bandwidth, buffer size and round-trip propagation times. The method avoids both under-utilization due to premature Slow-start termination, as well as multiple losses due to initially setting *ssthresh* too high, or increasing *cwnd* too fast. Experiments also show that Astart achieves good fairness and friendliness toward TCP NewReno. Lab measurements using a FreeBSD Astart implementation are also reported in this paper, providing further evidence of the gains achievable via Astart.**

Keywords-congestion control; slow-start; rate estimation, large bandwidth delay networks

I. INTRODUCTION

TCP is a reliable data transfer protocol [15] used widely over the Internet for numerous applications, from FTP to HTTP. The current implementation of TCP Reno/NewReno mainly includes two phases: Slow-start and Congestion-avoidance. In the Slow-start phase, a sender opens the congestion window (*cwnd*) exponentially, doubling *cwnd* every Round-Trip Time (RTT) until it reaches the Slow-start Threshold (*ssthresh*). The connection switches then to Congestion-avoidance, where *cwnd* grows more conservatively, by only 1 packet every RTT (or linearly). The initial *ssthresh* is set to an arbitrary default value, ranging from 4K to 64K Bytes, depending on the operating system implementation.

This research was supported by NSF under grant ANI-0221528.

By setting the initial *ssthresh* to an arbitrary value, TCP performance may suffer from two potential problems: (a) if *ssthresh* is set too high relative to the network Bandwidth Delay Product (BDP), the exponential increase of *cwnd* generates too many packets too fast, causing multiple losses at the bottleneck router and coarse timeouts, with significant reduction of the connection throughput; (b) if the initial *ssthresh* is set low relative to BDP, the connection exits Slow-start and switches to linear *cwnd* increase prematurely, resulting in poor startup utilization especially when BDP is large.

Recent studies [10] reveal that a majority of the TCP connections are short-lived (mice), while a smaller number of long-lived connections carry most Internet traffic (elephants). A short-lived connection usually terminates even before it reaches "steady state". That is, before *cwnd* grows to make good utilization of the path bandwidth. Thus, the startup stage can significantly affect the performance of the mice. In a large BDP network, with the current Slow-start scheme, it takes many RTTs for a TCP connection to reach the ideal window (equal to BDP). For example, in current Reno/NewReno implementation with initial *ssthresh* set to 32 Kbytes, a TCP connection takes about 100 sec to reach the ideal window over a path with a bottleneck bandwidth of 100 Mbps and RTT of 100ms. The utilization in the first 10 sec is a meager 5.97%. With the rapid development of the Internet and ever-growing BDP, a more efficient Slow-start mechanism is required to achieve good link-utilization.

In this paper, we evaluate the performance in large bandwidth delay networks of three current TCP Slow-start implementations: (1) Reno/NewReno, (2) New-Reno with Hoe's modification [12] and (3) Vegas [3]. We then propose a sender-side only modification, called Adaptive Start (Astart), to improve TCP startup performance. Astart takes advantage of the Eligible Rate Estimation (ERE) mechanism proposed in TCP Westwood (TCPW) [20], adaptively and repeatedly resetting *ssthresh* during the slow-start phase. When ERE indicates that there is more available capacity, the connection opens its *cwnd* faster, enduring better utilization. On the other hand, when ERE indicates that the connection is close to steady state, it switches to Congestion-avoidance, limiting the risk of buffer overflow and multiple losses. Ns-2 simulation experiments show that Astart significantly enhances performance of TCP connections, and show that the enhancement increases as BDP increases. When BDP reaches around 750 packets, the throughput improvement is an order of magnitude higher than that of TCP Reno/NewReno for short-lived connections. We also conduct experiments to compare Astart with the method using a large initial window of 64 Kbytes [8] in commercial satellite works, and evaluate Astart fairness, friendliness and performance under dynamic loading. Lab measurements are also carried out using a FreeBSD implementation.

The rest of the paper is organized as follows. In Section II we review background work, and give a brief overview of TCPW and the eligible rate estimation. In Section III we evaluate startup performance of several TCP variants, including Reno/NewReno, Hoe's modification and Vegas. Section IV presents Adaptive Start, our proposed modification of TCP slow-start, and illustrates its basic behavior. In Section V, we conduct simulation experiment to evaluate Astart throughput performance, adaptivity against congestion and multiple congestion, fairness and friendliness, performance under dynamic load. We also compare Astart with the use of large initial window method in this section. Lab experiments using FreeBSD implementation are provided in Section VI. Finally, Section VII discusses future work and concludes the paper.

II. Background

A. *Related Work on TCP Slow-start Mechanism*

TCP congestion control consists mainly of two phases: Slow Start and Congestion avoidance [15]. A new connection begins in Slow-start, setting its initial *cwnd* to 1 packet, and increasing it by 1 for every received Acknowledgement (ACK). After *cwnd* reaches *ssthresh*, the connection switches to congestion-avoidance where *cwnd* grows linearly.

A variety of methods have been suggested in the literature recently aiming to avoid multiple losses and achieve higher utilization during the startup phase. A larger initial *cwnd*, roughly 4K bytes, is proposed in [1]. This could greatly speed up transfers with only a few packets. However, the improvement is still inadequate when BDP is very large, and the file to transfer is bigger than just a few packets [22]. Fast start [19] uses cached *cwnd* and *ssthresh* in recent connections to reduce the transfer latency. The cached parameters may be too aggressive or too conservative when network conditions change.

Smooth start [21] has been proposed to slow down *cwnd* increase when it is close to *ssthresh*. The assumption here is that default value of *ssthresh* is often larger than the BDP, which is no longer true in large bandwidth delay networks. [12] proposes to set the initial *ssthresh* to the BDP estimated using packet pair measurements. This method can be too aggressive, as we will show in Section III. In [22], SPAND (Shared Passive Network Discovery) has been proposed to derive optimal TCP initial parameters. SPAND needs leaky bucket pacing for outgoing packets, which can be costly and problematic in practice [2].

TCP Vegas [3] detects congestion by comparing the achieved throughput over a cycle of length equal to RTT, to the expected throughput implied by *cwnd* and *baseRTT* (minimum RTT) at the beginning of a cycle. This method is applied in both Slow-start and Congestion-avoidance phases. During Slow-start phase, a Vegas sender doubles its *cwnd* only every other RTT, in contrast with Reno's doubling every RTT. A Vegas connection exits slow-start when the difference between achieved and expected throughput exceeds a certain threshold. However, Vegas is not able to achieve high utilization in large bandwidth delay networks as we will show in Section 3, due to its over-estimation of RTT.

We believe that estimating the eligible sending rate and properly using such estimate are critical to improving bandwidth utilization during Slow-start.

B. *TCP Westwood and Eligilbe Rate Estimation Overview*

In TCP Westwood (TCPW) [4], the sender continuously monitors ACKs from the receiver and computes its current Eligible Rate Estimate (ERE) [20]. ERE relies on an adaptive estimation technique applied to ACK stream. The goal of ERE is to estimate the connection eligible sending rate with the goal of achieving high utilization, without starving other connections. We emphasize that what a connection is eligible for is not the residual bandwidth on the path. The connection is often eligible more than that. For example, if a connection joins two similar connections, already in progress and fully utilizing the path capacity, then the new connection is eligible for a third of the capacity.

Research on active network estimation [5] reveals that samples obtained by "packet pair" is more likely to reflect link capacity, while samples obtained by "packet train" give short-time throughput. In TCPW, the sender adaptively computes T_k, an interval over which the ERE sample is calculated. An ERE sample is computed by the amount of data in bytes that were successfully delivered in T_k. T_k depends on the congestion level, the latter measured by the difference between 'expected rate' and 'achieved rate' as in TCP Vegas. That is T_k depends on the network congestion level as follows:

$$T_k = RTT \times \frac{cwin/RTT_{\min} - RE}{cwin/RTT_{\min}},$$

where $\mathrm{RTT}_{\min}$ is the minimum RTT value of all acknowledged packets in a connection, and RTT is the smoothed RTT measurement. The expected rate of the connection when there is no congestion is given by *cwnd*/$\mathrm{RTT}_{\min}$, while RE is the achieved rate computed based on the amount of data acknowledged during the latest RTT, and exponentially averaged over time using a low-pass filter. When there is no congestion, and therefore no queuing time, *cwnd*/$\mathrm{RTT}_{\min}$ is almost the same as RE, producing small T_k. In this case, ERE becomes close to a packet pair measurement. On the other hand, under congestion conditions, RE will be much smaller than *cwnd*/$\mathrm{RTT}_{\min}$ due to longer queuing delays. As a result, T_k will be larger and ERE closer to a packet train measurement. After computing the ERE samples, a discrete version of a continuous first order low-pass filter using the Tustin approximation [23] is applied to obtain smoothed ERE.

In current TCPW implementation, upon packet loss (indicated by 3 DUPACKs or a timeout) the sender sets *cwnd* and *ssthresh* based on the current ERE. TCPW uses the following algorithm to set *cwnd* and *ssthresh*. (We will describe our proposed Adaptive Start in Section IV, which applies to both initial start-up phase and Slow-start after coarse timeouts.)

```
if (3 DUPACKS are received)
    ssthresh = (ERE*RTTmin)/seg_size;
    if (cwnd >ssthresh)  /*congestion avoid*/
        cwnd=ssthresh;
    endif
endif

if (coarse timeout expires)
    cwnd = 1;
    ssthresh =(ERE *RTTmin)/seg_size;
    if(ssthresh < 2)
        ssthresh = 2;
    endif
endif
```

III. TCP SLOW START PERFORMANCE

In this section we state briefly the current TCP Slow-start mechanisms, and evaluate their startup performance in large bandwidth delay networks by simulation. We illustrate the inadequacy of the current schemes when facing networks with large BDP, and reveal the reason behind it.

A. *Simulation Setup*

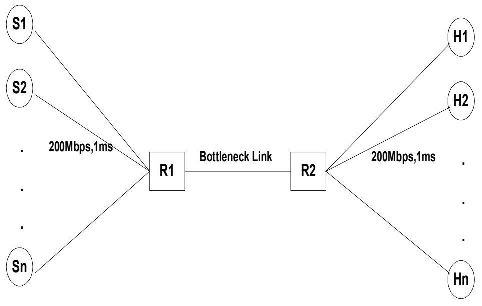


Figure 1. Network topology for simulations

All results in this paper are obtained using ns-2 [13]. The network topology is shown in Figure 1, where S_i represents a TCP sender and H_i a TCP receiver. R_1 and R_2 are two routers with finite buffer capacity, each set equal to the Bandwidth Delay Product (BDP) unless otherwise specified. Results are obtained for varying propagation time and bottleneck bandwidth. FTP is the simulated application. The receiver issues an ACK for every data packet received. We assume the receiver's advertised window is always large so that the actual sending window is always equal to *cwnd*. For the convenience, the window size is measured in number of packets, and the packet size is 1000 bytes. The initial *ssthresh* for Reno/Newreno is set to be 32 packets, equal to 32 Kbytes.

B. *TCP Reno/NewReno*

In TCP Reno/NewReno, a sender starts in Slow-start, *cwnd* < *ssthresh*, and every ACK received results in an increase of *cwnd* by 1 packet. Thus, the sender exponentially increases *cwnd*. When *cwnd* hits *ssthresh*, the sender switches to congestion avoidance phase, increasing *cwnd* linearly, considerably slower than in slow start.

In this Subsection, we evaluate Reno/NewReno startup performance in large BDP networks. If the initial ssthresh is too low [1], a connection exits Slow-start and switches to Congestion-avoidance prematurely, resulting in poor utilization. Figure 2 shows the Reno *cwnd* dynamics in the startup stage. The results are obtained for a bottleneck bandwidth of 40Mbps, and RTT values of 40, 100 and 200ms. The bottleneck buffer size is set equal to BDP in each case.

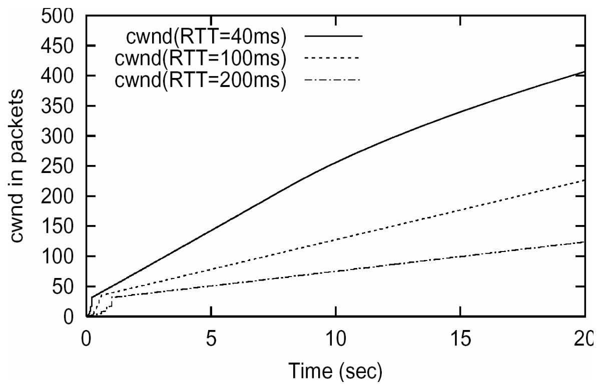


Figure 2. cwnd dynamic during the start-up phase

From Figure 2, we see that when RTT=100ms, Reno stops exponentially growing *cwnd* long before it reaches the ideal value (BDP=500). After that, *cwnd* increases slowly, and has not reach 500 by 20sec. As a result, the achieved throughput is only 12.90 Mbps, much lower than the desired 40 Mbps. Another observation concerns how RTT affects performance. When RTT increases, the ideal window grows too. On the other hand, because *cwnd* increases 1 packet per RTT during Congestion-avoidance, longer RTT means slower *cwnd* growth, resulting in even lower utilization. The results in Table 1 show the drastic reduction in utilization as RTT increases.

[1] In a network with small BDP, the initial *ssthresh* might be set too high. As a result, at some cycle in slow start, a Reno sender often overshoots the BDP, causing multiple losses and a coarse timeout. This is also a problem resulting from an inappropriate setting of ssthresh.

TABLE 1. NEWRENO UTILIZATION DURING FIRTST 20 SEC (BANDWIDTH =40MBPS)

RTT(ms)	20	50	100	150	200
Utilization(%)	95.6	71.8	23.2	11.9	7.2

Consider now the impact of bottleneck bandwidth on utilization during startup stage. With the increase of bottleneck bandwidth, the packet transmission speeds up. But the sender still has to wait for ACKs to increase *cwnd*. Thus, after prematurely exiting slow-start, *cwnd* grows with almost the same rate (1 packet per RTT) regardless of bandwidth. With larger bottleneck capacity, more bandwidth is left unused, which leads to lower utilization. Table 2 shows the relation between utilization and bottleneck bandwidth during the startup stage. The utilization drops to 4.7% with 200 Mbps bottleneck bandwidth.

TABLE 2. NEWRENO UTILIZATION DURING FIRTST 20 SEC (RTT =100MS)

Bandwidth(Mbps)	10	20	40	100	200
Utilization(%)	77.1	45.9	23.2	9.3	4.7

C. TCP Reno/NewReno with Hoe's Slow Start Modification

In [12], Hoe proposes a method for setting the initial *ssthresh* to the product of delay and estimated bandwidth. The bandwidth estimation is calculated by applying the least squares estimation on three closely-spaced ACKs (similar to the concept of packet pair [17]). RTT is obtained by measuring the round trip time of the first segment transmitted.

Hoe's modification enables the sender to get an estimation of the BDP at an early stage and set the *ssthresh* accordingly, thus avoiding switching to congestion avoidance prematurely. As illustrated in Figure 3 with large buffer space (buffer size=BDP=500), Reno with Hoe's modification increases *cwnd* exponentially and exits properly.

However, Hoe's modification may encounter multiple-loss problems when the bottleneck buffer is not big enough compared to the BDP, which could easily happen in large bandwidth delay networks. In Figure 3 when the buffer size is 125 packets (1/4 BDP), the connection encounters multiple losses and runs into a long recovery time (from 0.9 sec to 14.8 sec). The achieved throughput during the first 20 sec is only 3.61 Mbps, translating into 9% utilization.

The reason for the multiple losses is as follows. During Slow-start, for every ACK received, the sender increases *cwnd* by 1 and sends out 2 new packets. If the receiver acknowledges every packet, then after n RTT, *cwnd* will be 2^n. Suppose the access link capacity is at least twice as large as the bottleneck capacity, these 2^n packets will arrive at the bottleneck back to back at a speed twice that of the bottleneck link. Thus, to avoid losses, at least a buffer of 2^{n-1} packets is needed to hold off the temporarily bursting packets. Hoe's modification sets *ssthresh* to the estimated BDP, thus, a buffer size of BDP/2 is required to prevent multiple losses for single connection.

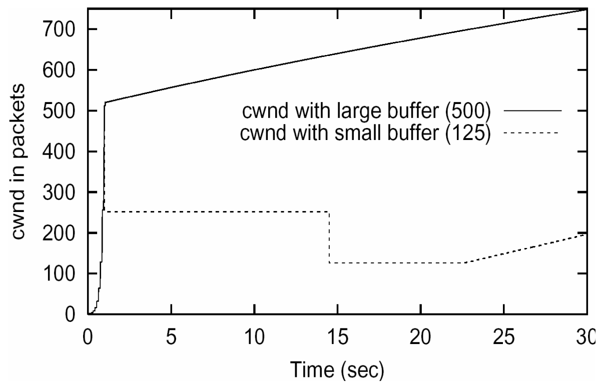


Figure 3. *cwnd* dynamics in NewReno with Hoe's modification

More importantly, Hoe's modification does not adjust to changing path load. If there are multiple connections starting up at approximately the same time, or other large volume traffic (for example, video transferring) joins in when a connection is in Slow-start, the Hoe's modification will have set the initial *ssthresh* too high, resulting in multiple losses and coarse timeout.

D. TCP Vegas

Unlike TCP Reno/Newreno that uses packet loss as congestion indication, TCP Vegas [3] detects incipient congestion by comparing the achieved throughput to the expected throughput at the beginning of a cycle (RTT). The difference between these two values reflects the queue length of the connection in the bottleneck router.

This Vegas method is applied to both Slow-start and Congestion-avoidance phases. In congestion-avoidance, *cwnd* increases by 1 per RTT if the difference is small, meaning that there is enough network capacity. Vegas reduces *cwnd* in the same fashion (by 1 packet) when the achieved throughput is considerably lower than the expected throughput.

During Slow-start, Vegas doubles its congestion window only every other RTT (compared to Reno's every RTT). When the difference between actual and expected throughput exceeds a threshold, Vegas stops its window doubling and switches to Congestion-avoidance (See Figure 4).

By growing *cwnd* slower and monitoring every RTT for incipient congestion, Vegas avoids multiple losses and the coarse timeout that would result [11]. However, when the BDP is large, Vegas may under-utilize the available bandwidth by switching to congestion avoidance too early [18]. The premature slow-start termination is caused by RTT over-estimation in the Vegas algorithm. In Vegas, the sender checks the difference between expected and actual throughput:

$diff = \frac{cwnd}{baseRTT} - \frac{cwnd}{RTT_n}$ only at the beginning of the RTT where *cwnd* is doubled[2].

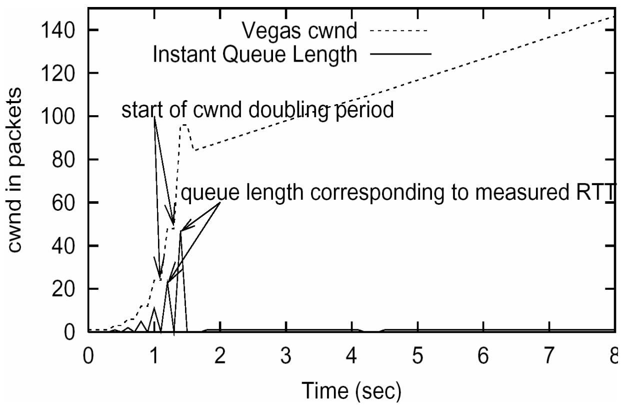


Figure 4. Vegas cwnd dynamic and queue length during startup phase (Bottleneck bandwidth =40 Mbps, baseRTT =100ms)

At this point, RTT_n is over-estimated because of the temporary queue buildup at the router during the previous cycle (the last two RTTs). Figure 4 shows the instantaneous queue length pattern. As a result of RTT over-estimation, *diff* is over-estimated too, and Vegas exits Slow-start prematurely. A more detailed analysis of this problem can be found in [18]. Figure 4 also shows Vegas *cwnd* dynamic over a path with BDP equal to 500 Packets. Vegas exits slow start at *cwnd*=96, while the ideal window is 500 packets

The startup under-utilization of Vegas is aggravated as BDP grows. Table 3 shows the ratio of the slow start termination *cwnd* to the ideal window value for different bottleneck bandwidth. The ratio is reduced to about 0.1 with a bottleneck of 100 Mbps.

TABLE 3. THE RATIO OF SLOW START TERMUNATION WINDOW TO THE IDEAL WINDOW (BDP) IN VEGAS (ROUND-TRIP TIME =100MS)

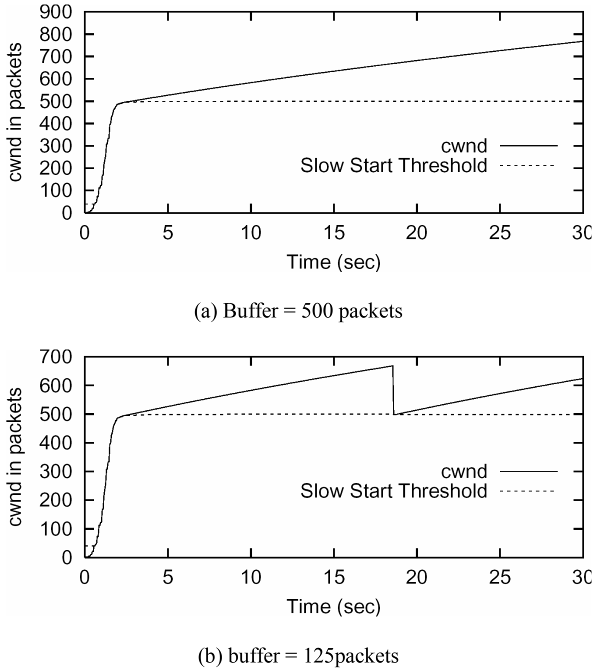
Bandwidth(Mbps)	10	20	40	80	150
Ratio	0.384	0.192	0.192	0.096	0.101

IV. MODIFIED TCP SLOW START FOR LARGE BANDWIDTH DELAY NETWORKS

In this Section, we propose a simple sender-side only modification, which we call Adaptive Start (Astart), to the traditional Reno/NewReno slow start algorithm. We take advantage of the TCPW eligible rate estimate, using it to adaptively and repeatedly reset *ssthresh* during the startup phase, both connection startup, and after every coarse timeout. The pseudo code of the algorithm is as follows. When an ACK arrives:

```
if ( 3 DUPACKS are received)
   switch to congestion avoidance phase;
else (ACK is received)
   if (ssthresh < (ERE*RTTmin)/seg_size)
         ssthresh =(ERE*RTTmin)/seg_size;
   endif
   if (cwnd >=ssthresh)  /*mini linear increase phase*/
      increase cwnd by 1/cwnd;
   else if cwnd <ssthresh)  /*mini exponential increae phase*/
      increase cwnd by 1;
   endif
endif
```

In TCPW, an eligible rate estimate is determined after every ACK reception. In Astart, when the current *ssthresh* is much lower than ERE*RTT_{min}, the sender resets *ssthresh* higher accordingly, and increases *cwnd* in slow-start fashion. Otherwise, *cwnd* increases linearly to avoid overflow. In this way, Astart probes the available network bandwidth for this connection, and allows the connection to eventually exit Slow-start close to the ideal window (See Figure 5). Compared to Vegas, TCPW avoids premature exit of slow start since it relies on both RTT and ACK intervals, while Vegas only relies on RTT estimates.



(a) Buffer = 500 packets

(b) buffer = 125packets

Figure 5. Astart cwnd dynamic during startup phase (Bottleneck bandwidth =40 Mbps, RTT=100ms, BDP =500 packets,)

Figure 5(b) illustrates the *cwnd* dynamic in the case of small buffer (equal to BDP/4). By applying Astart, the sender does not overflow the bottleneck buffer and thus multiple losses are avoided. Figure 6 gives a closer look at the *cwnd* dynamic. In effect, Astart consists of multiple mini-linear-increase and mini-exponential-increase phases. Thus, *cwnd* does not increase as fast as in Hoe's method, especially as *cwnd* approaches BDP. This prevents the temporary queue from building up too fast, and thus, prevents a sender from

[2] Provided that the difference indicates no congestion

overflowing a small buffer. Comparing the *cwnd* evolution in Figure 5 and Figure 6 to those in Figure 3, it is clear that *cwnd* increase in Astart follows a smoother curve when it is close to BDP.

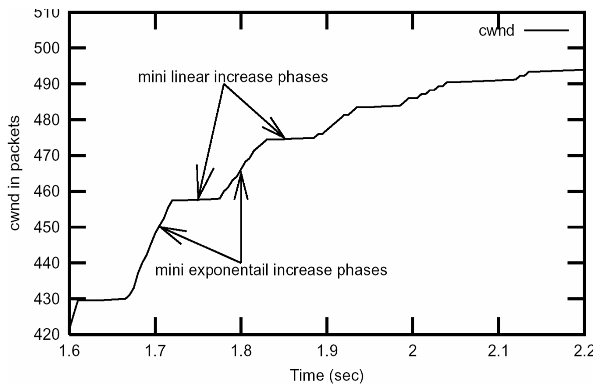


Figure 6. a closer look at Astart cwnd dynamic during startup phase

V. SIMULATION RESULTS AND DISCUSSION

In this Section, we evaluate the performance of Astart, comparing the throughput performance of the proposed Astart algorithm to other mechanisms we described and evaluated in the previous section. We also compare Astart with commercial satellite transport protocol where very large initial window is used. Finally we will evaluate how well Astart co-exists with TCP NewReno, the de facto Internet data transport protocol.

A. *Astart Behavior with Multiple Connections*

We ran simulation with 5 connections starting at the same time (the network parameter is the same as in Figure 5(a)). The results in Figure 7 show that each connection is able to estimate its share of bandwidth and switch to Congestion-avoidance at the appropriate time.

We drew graph with 5 connections for the conve-nience of presentation, simulations with more connec-tions show that Astart can promptly pump up its *cwnd* and then switch to Congestion-avoidance properly.

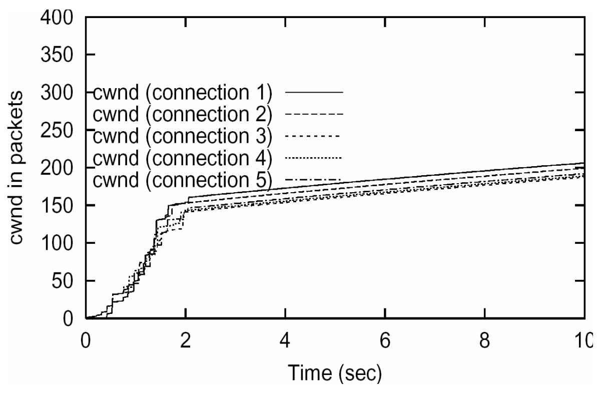


Figure 7. Astart cwnd Dynamic when 5 connections start at the same time (Bottleneck Bandwidth=40Mbps, RTT =100ms, BDP =500 packets)

B. *Startup in a Congested Network*

To evaluate the adaptivity of Astart when the network becomes congested, we also tested the startup behavior when another high-volume UDP connection joins the TCP connection during the slow start phase. We ran simulations with one TCP connection starting at time 0 over a link with capacity 40 Mbps. A UDP flow with intensity of 20 Mbps starts at 0.5 sec. Figure 8 shows that Hoe's method runs into multiple losses and finally times out. The reason is the setting of the initial *ssthresh* to 500 (BDP) at the very beginning of the connection, and the lack of adjustment to the change in network load later. In contrast, Astart has a more appropriate (lower) slow-start exit *cwnd*, thanks to the continuous estimation mechanism, which reacts to the new traffic and determines an eligible sending rate that is no longer the entire bottleneck link capacity.

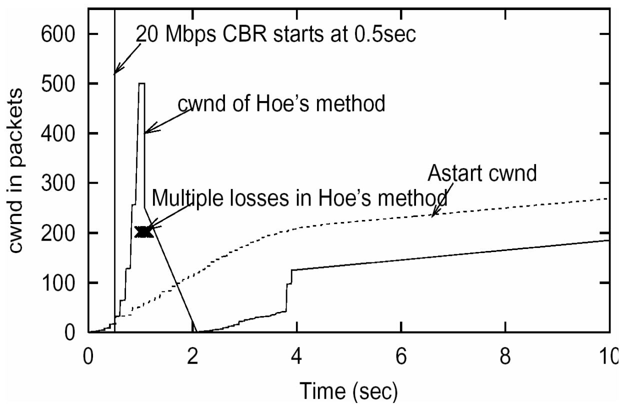


Figure 8. cwnd Dynamic with UDP traffic joins in during srartup(Bottleneck capacity=40 Mbps, RTT=100ms, BDP =500 packets)

C. *Throughput Comparison*

The summary of this sub-section is that Astart significantly improves TCP startup performance with regards to various bottleneck bandwidth, buffer size and round-trip time. To focus on the start-up performance of different schemes, we only calculate the throughput during the first 20 seconds.

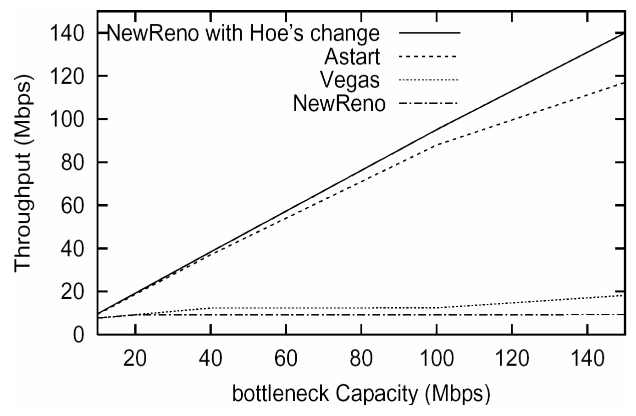


Figure 9. Throughput vs. bottleneck capacity (first 20 seconds)

The throughput of Astart, NewReno, NewReno with Hoe's modification and Vegas are examined under bottleneck bandwidth varying from 10 to 150 Mbps (while fixing the round-trip time at 100ms). The results in Figure 9 show that Astart and Hoe's modification achieve higher throughput, and scale with bandwidth. NewReno and Vegas performance lags in this scenario. Another observation is that Newreno with Hoe's modification slightly outperforms Astart. In Hoe's method, the initial *ssthresh* is immediately set to the bandwidth after 3 closely spaced ACKs returned, so *cwnd* increases by one for every ACK received. On the other hand, Astart gradually probes for bandwidth and slows down when the estimate is closer to the connection bandwidth share. We believe that the slightly lower throughput achieved by Astart above is more than compensated for by its avoidance of buffer overflow and multiple losses in other cases.

To assess the robustness of the different schemes to buffer size, we ran simulations with bottleneck buffer size varying from 100 (BDP/5) to 250 (BDP/2) packets. The bandwidth is 40 Mbps and RTT is 100 msec. The results in Figure 10 show that Astart is robust to buffer size reductions, while NewReno with Hoe's modification suffers when the buffer size is smaller than BDP/2. The reduction in buffer size has no meaningful impact on NewReno and Vegas. They still exit Slow-start prematurely as explained in Section III.

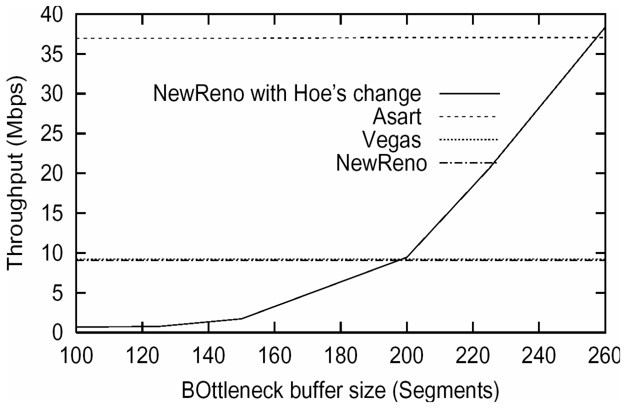


Figure 10. Throughput vs. bottleneck buffer size (first 20 seconds)

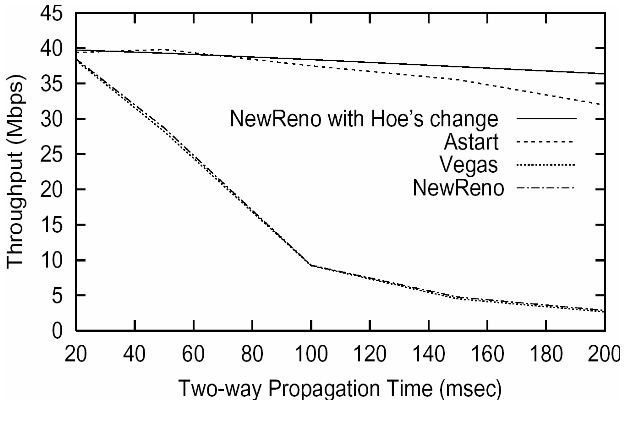


Figure 11. Throughput vs. two-way propagation time (first 20 seconds)

RTT can considerably affect the startup performance. Figure 11 shows the throughput of Astart, NewReno, Hoe's modification and Vegas with RTT varying from 20 to 200 msec. The Bottleneck bandwidth is fixed here at 40 Mbps and buffer size is set equal to BDP. Figure 11 shows that Astart and Hoe's method both scale well with RTT with Hoe's modification slightly better for the same reason previous stated (Hoe's method set the *ssthresh* immediately to the BDP where Astart probes and slows down when *cwnd* is close to the BDP). The performance of NewReno and Vegas deteriorate significantly as RTT increases.

The studies in the last two Sections focused on the performance a TCP connection during its initial startup phase. But Asart can also be used after any coarse timeout. This is of particular value to TCPW since after a timeout, ERE is small relative to the connection actual bandwidth share. This is because during a coarse timeout, the sender transmits so few packets, and therefore the share estimate is very low. Astart helps in this case by gradually probing for bandwidth share and switching to congestion avoidance at a more appropriate time.

D. *Comparing Astart to the Use of Large Initial Window (LIW) over satellite links*

In a connection that incorporates a satellite link, the main bottleneck in TCP performance is due to the large delay-bandwidth product nature of the satellite link. As we mentioned in Section II, a larger initial *cwnd*, roughly 4K bytes, is proposed in [1]. This could greatly speed up transfers with only a few packets. However, the improvement is still inadequate when BDP is very large, and the file to transfer is bigger than just a few packets.

More aggressively, commercial satellite data communication providers typically use a very large initial window (LIW) over satellite links, e.g., 64 Kbytes, and thus bypass the slow start stage of the normal TCP evolution [8]. This method effectively increases the utilization during the startup. However, it cannot single-handedly solve the problem of poor startup utilization over satellite links. Below we will show the reason and also compare the performance of Astart with LIW method.

A commercial satellite system using a geo-stationary (GEO) could have bandwidth up to 24 Mbps. Which results in a BDP of about 3000 with one-way propagation delay of 500 ms. Under this situation, even with an initial window of 64 Kbytes, it would take a very long time for TCP to fully utilize the link.

Figure 12 compares the startup behavior of Astart and LIW method. The bottleneck capacity is 10 Mbps and one-way propagation delay is 250 ms. The graph shows that although LIW method comes up strong at the very beginning, it fades quickly comparing to Astart due to bypassing the slow-start stage. As a result, the throughput of LIW method during this period is only 2.80 Mbps comparing to Astart's 9.33 Mbps.

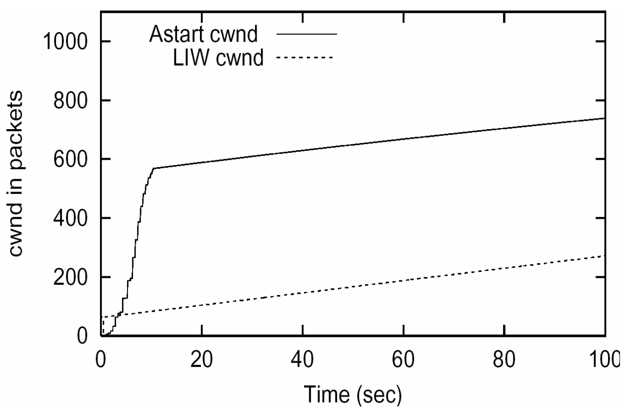


Figure 12. Congestion window dynamics of Asart and LIW method. (bottleneck=10Mbps, RTT=500ms, BDP =600)

Another challenge LIW method faces is caused by its inability to adapt to different network conditions. By setting the initial congestion to a large value, if the network is highly congested or many connections simultaneously join in, it is possible that using LIW overflows the buffers and causes multiple losses.

Moreover, a connection using a satellite link may also has a terrestrial part, thus using LIW end-to-end could affect the performance and fairness of the terrestrial part of the connection.

E. Fairness and Friendliness to TCP NewReno

Fairness relates to the relative performance of a set of connections of the same TCP variant. Friendliness relates to how sets of connections running different TCP flavors affect the performance of each other. The simulation topology consists of a single bottleneck link with a capacity of 50 Mbps, and one-way propagation delay of 35ms. The buffer size at the bottleneck router is equal to the pipe size. The link is loss free except where otherwise stated.

A set of simulations with 10 simultaneous flows was run to investigate fairness of Astart. To provide a single numerical measure reflecting the fair share distribution across the various connections we use the Jain's Fairness Index defined as [16]:

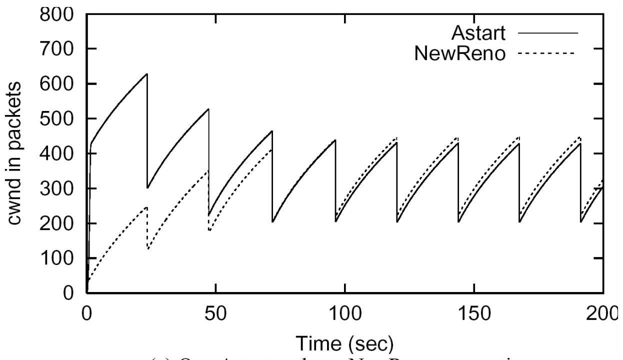
$$\text{Fairness Index} = \frac{(\sum_{i=1}^{n} b_i)^2}{n\sum_{i=1}^{n} b_i^2},$$

where b_i is the throughput of the i^{th} flow and n is the total number of flows. The fairness index always lies between 0 and 1. A value of 1 indicates that all flows got exactly the same throughput.

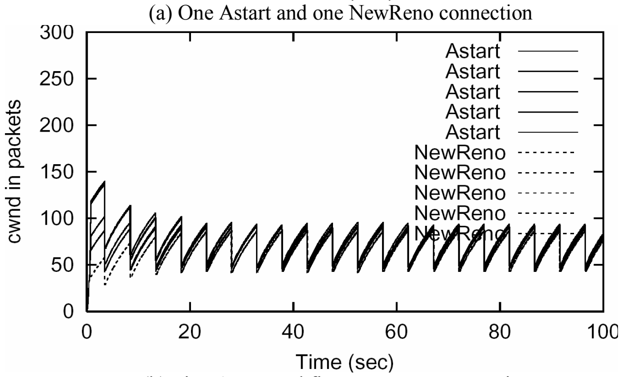
We calculate the fairness index for both Reno and TCPW. The Jain's fairness index of Astart reached 0.9949, and that of NewReno is 0.9944. Therefore, fairness of Astart is comparable to that of NewReno.

Since Astart invokes faster probing during startup, the evaluation of Astart friendliness toward NewReno is important. Thanks to good friendliness characteristics of TCPW, Astart connections can effectively coexist with NewReno connections over the same path. Figure 13 shows *cwnd* dynamics for Astart and NewReno connections.

The bottleneck link bandwidth is 50Mbps and a two way propagation delay is 70msec. In Figure 13(a), one Atart and one NewReno connection start running at the same time. The Astart connection benefits initially by quickly reaching cruising speed. Astart and NewReno connections both reach the same *cwnd* after a few congestion episodes. In Figure 13 (b), five Astart and five NewReno connections start simultaneously. The first started Atart connection gets more bandwidth initially, but again all connections, regardless of Astart or NewReno, reach fair share rate after a few congestion episodes.



(a) One Astart and one NewReno connection



(b) Five Astart and five NewReno connections
Figure 13. *cwnd* dynamics between Astart and NewReno (Bottlneck band width is 50Mbps, and RTT is 70msec)

F. Astart Performance under dynamic load

We evaluated the performance of Astart under highly dynamic load conditions. In 20 minutes simulation time, we ran 100 connections. Connections starting times are uniformly distributed over the simulation time. The lifetime of a connection is fixed at 30 seconds. We compare the results among NewReno with Astart.

Figure 14 shows Total throughput vs bottleneck bandwidth. The total throughput is computed as the sum of throughputs of all connections. Propagation delays are 70ms, and the bottleneck buffer size is set equal to the pipe size (BDP). In

10Mbps, Astart does not get much benefit, because NewReno does not have any difficulty filling the smaller pipe. As the bottleneck link capacity increases, the difference between Astart and NewReno becomes more obvious. At 200 Mbps, Astart achieves about 60% more throughput than NewReno.

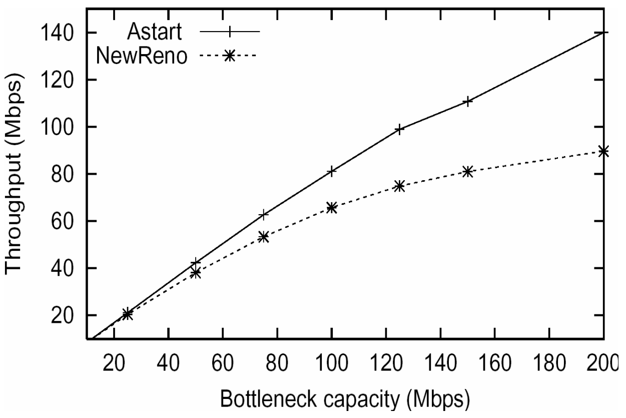


Figure 14. Throughput vs. bottleneckcapacity of Astart and NewReno under dynamic load (RTT = 70msec)

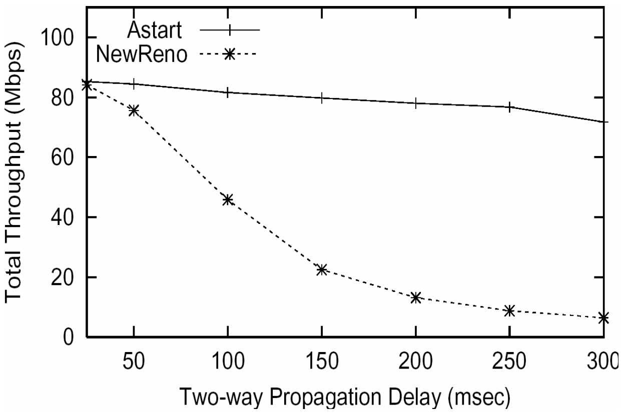


Figure 15. Throughput vs. RTT of Astart and NewReno under dynamic load (Bottleneck capacity = 100 Mbps)

To assess the relation of the efficiency to the E2E propagation time under dynamic load, we ran simulations with a bottleneck capacity of 100 Mbps and two-way propagation ranging from 20 to 300 msec. The results in Figure 15 show a significant gain for Astart up to 1016% over NewReno. When the RTT increases, the performance of NewReno degrades severely, while Astart is able to maintain respectable utilization.

VI. FREEBSD IMPLEMENTATION

To evaluate Astart performance in actual systems, we have implemented Astart algorithms on FreeBSD systems. Lab measurements confirmed our simulation results, showing that Astart behaves quite well in actual systems.

A. *Measurement setup*

Figure 16 shows the measurement configuration in our experiments. All PCs are running on FreeBSD [9] Release 4.5. CPU clock tick is 10msec (default). We use Dummynet [6][7] to emulate the bottleneck router. The bottleneck link (from PC router to TCP receiver) speed is set to 10Mbps. The propagation time is 400msec, and the router buffer is set equal to BDP (500Kbytes). Active Queue Management is not used, that is tail-drop is adopted to simplify the experiments. Further, no random packet loss was induced at this time. We use Iperf [14] as a traffic generator. The receiver's advertised window is set large enough at 4 Mbytes. We fixed the initial *ssthresh* for NewReno equal to 32 Kbytes in our measurements.

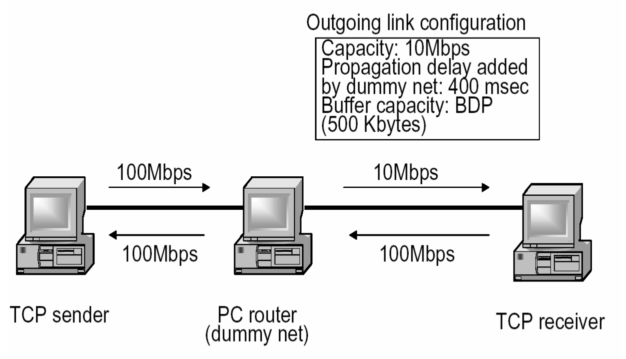

Outgoing link configuration
Capacity: 10Mbps
Propagation delay added by dummy net: 400 msec
Buffer capacity: BDP (500 Kbytes)
100Mbps
10Mbps
100Mbps
100Mbps
TCP sender
PC router (dummy net)
TCP receiver

Figure 16. Measurement topology and configuration

B. *Astart Behavior*

Figure 17 shows measured *cwnd* dynamics in Astart and NewReno connections at Slow Start. NewReno enters Congestion Avoidance phase after *cwnd* reaches the initial *ssthresh* (32k), and *cwnd* increases linearly by one packet per RTT. *cwnd* reaches only 2Mbps, 20 % of the link capacity, for the first 25 seconds. On the other hand, in the Astart connection, *cwnd* quickly converges to the link capacity due to the adaptive *ssthresh* resetting. We can also confirm that *cwnd* growth is not exponential throughout startup. Initially *cwnd* increases rapidly for the first 3 seconds, and then increases more slowly as the connection approaches the link capacity.

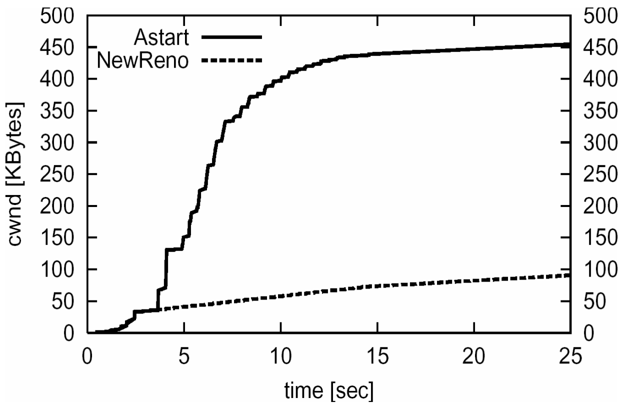


Figure 17. *cwnd* dynamics in Astart and NewReno at Start-up (Lab Measurements Results)

VII. Conclusions and Future Work

In this paper, we evaluated the startup performance of TCP Reno/Newreno, Vegas and Hoe's modification in large bandwidth delay networks. We proposed a modified Slow-start mechanism, called Adaptive Start (Astart), to improve the startup performance in such networks. Astart adaptively and repeatedly resets the Slow-start threshold (*ssthresh*), based on the eligible rate estimation mechanism proposed in TCP Westwood. By adapting to network conditions in the startup phase, a sender is able to grow the congestion window (*cwnd*) efficiently without overflowing the bottleneck buffer. Simulations and lab measurement experiments have shown that Astart can significantly improve the link utilization especially for large BDP. Compared to previous proposals, Astart is more robust to small buffer sizes, avoiding both premature termination of Slow-start, as well as multiple losses and the resultant coarse timeout. Experiments also have shown that Astart achieves good fairness and friendliness toward NewReno.

Work in progress includes considering environments where random loss is also possible in the very initial phase of a connection (when using wireless links) or when the round trip time is extremely large (beyond 0.5 sec). Extensive measurement experiments on wired/wireless and satellite networks are also planned.

Acknowledgment

We would like to thank Anders Persson for conducting the lab measurement experiments, and Hideyuki Shimonishi for helpful discussions.

References

[1] M. Allman, S. Floyd and C. Patridge, “Increasing TCP's initial Window”, INTERNET DRAFT, April 1998.

[2] A. Aggarwal, S. Savage, T.E. Anderson, “Understanding the Performance of TCP Pacing,” In Proceedings IEEE INFOCOM 2000, Tel Aviv, Israel, March 2000.

[3] L.S. Brakmo and L.L. Perterson. TCP Vegas: End-to-End Congestion Avoidance on a Global Internet. IEEE Journal on Selected Areas in Communication, Vol. 13, Nov. 8, October 1995.

[4] C. Casetti, M. Gerla, S. Mascolo, M. Y. Sanadidi, and R. Wang, “TCP Westwood: bandwidth estimation for enhanced transport over wireless links,” In Proceedings of Mobicom 2001, Rome, Italy, Jul. 2001.

[5] C. Dovrolis, P.Ramanathan and D. Moore, “What Do Packet Dispersion Techniques Measure?,” In Proceedings of Infocom 2001, Anchorage AK, April 2001.

[6] Luigi Rizzo, “Dummynet: a simple approach to the evaluation of network protocols”, ACM Computer Communication Review, 1997.

[7] IP Dummynet URL: http://info.iet.unipi.it/~luigi/ip_ dummynet/

[8] N. Ehsan, M. Liu and R. Ragland, "Measurement Based Performance Analysis of Internet over Satellite", 2002 International Symposium on Performance Evaluation of Computer and Telecommunication Systems (SPECTS 2002), July 2002, San Diego.

[9] FreeBSD Project, URL: http:://www.freebsd.org/

[10] L. Guo and I. Matta. The War between Mice and Elephants. In Proceedings of ICNP'2001: The 9th IEEE International Conference on Network Protocols*, Riverside*, CA, November 2001.

[11] G. Hengartner, J. Bolliger, and T. Gross, “TCP Vegas revisited,” In Proc. of IEEE Infocom 2000, March 2000, pp. 1546-1555

[12] J. C. Hoe, Improving the Start-up Behavior of A Congestion Control Scheme for TCP”, Proc. ACM SIGCOMM ’96, pp. 270-280.

[13] NS-2 Network Simularor (ver.2.) LBL, URL: http://www.mash.cs.berkley.edu/ns/.

[14] Iperf Version 1.7.0, URL: http://dast.nlanr.net/Projects/Iperf/

[15] V. Jacobson, “Congestion avoidance and control,” ACM Computer Communications Review, 18(4) : 314 - 329, Aug. 1988.

[16] R. Jain, “The art of computer systems performance analysis,” John Wiley and sons, QA76.9.E94J32, 1991.

[17] S. Keshav A Control-Theoretic Approach to Flow Control. In Proceeding of ACM SIGCOMM’ 1991, Pages 3-15, Sept. 1991.

[18] Soo-hyeong Lee, Byung G. Kim, and Yanghee Choi, "Improving the Fairness and the Response Time of TCP-Vegas," In Lecture Notes in Computer Science, Springer Verlag.

[19] V.N. Padmamabhan and R.H. Katz, “TCP Fast Start: A Technique for Speeding Up Web Transfers”, Proceedings of IEEE globecom’98, Sydney, Australia, Nov. 1998.

[20] R. Wang, M. Valla, M.Y. Sanadidi and M. Gerla, “Using Adaptive Bandwidth Estimation to provide enhanced and robust transport over heterogeneous networks”, 10th IEEE International Conference on Network Protocols (ICNP 2002), Paris, France, Nov. 2002.

[21] H. Wang, H. Xin, D.S. Reeves and K.G. Shin "A Simple Refinement of Slow Start of TCP Congestion Control", In proceedings of ISCC’00, Antibes, France, 2000

[22] Y. Zhang, L. Qiu and S. Keshav, “Optimizing TCP Start-up Performance”, Cornell CSD Technical Report, February, 1999.

[23] K. J. Astrom, and B. Wittenmark, “Computer controlled systems,” Prentice Hall, Englewood Cliffs, N. J., 1997.