

Network Adaptive TCP Slow Start

Yogesh Bhumralkar, Jeng Lung, and Pravin Varaiya

University of California, Berkeley
{ykb, jlung, [varaiya](mailto:varaiya@eecs.berkeley)}@eecs.berkeley

Abstract—Web browsing generally involves multiple short file transfers. The reliable transport protocol used by the World Wide Web is the Transmission Control Protocol (TCP), but several recent studies have shown that TCP is extremely inefficient for the transfer of small files. TCP uses the slow start phase to gradually ramp up its congestion window in order to probe for available bandwidth. The problem with this algorithm is that it does not efficiently utilize the available bandwidth when the file sizes are small because the majority (if not all) of the file finishes transferring while still in this slow start phase. Thus, data is never really transferred at the peak available rate. We propose to explore various other algorithms as alternatives to the TCP slow start algorithm. The goal is to provide a mechanism for performing the transfer of small files much more efficiently than is currently done with TCP. The new algorithms considered in this paper rely on previous network performance history as well as the size of the data in an effort to speed up the file transfer.

1 INTRODUCTION

The increasing popularity of the World Wide Web (WWW) over the course of the past several years has resulted in an exponential growth in the amount of Web traffic. Although the composition of the data keeps changing over time, and while the demand for certain kinds of data tends to fluctuate depending on the popularity of the data, one thing that has remained consistent is the nature of the flows. The nature of the flows that we refer to is the fact that in general, Web data download involves the transfer of multiple short files followed by periods of idle time – the reason for this will be explained shortly. Studies indicate that almost 70% of gateway traffic consists of Web transfers (most of which are short transfers of about 10-20 packets on average) as opposed to long-lived FTP-type flows [26]. As the use of the WWW, and therefore, the amount of Web traffic, constantly increases, it is imperative to ensure that we have the best possible resources to deal with the growing demand for precisely this type of traffic. These resources include not only the deployment of better technology as far as the hardware and software concerned, but even more importantly, include the design infrastructure that the Web is based on.

At present, clients use the HyperText Transfer Protocol (HTTP) to request and retrieve information from Web servers. HTTP transfers Web objects by setting up multiple concurrent connections between the server and

the client, with each connection transferring one of the requested objects. The following simplified description illustrates what happens during a typical Web transfer session. When a user sitting at a machine accesses a Web page, HTTP relays that request to the server. The server, in turn, responds by transferring each of the objects on the page (such as the HTML document itself, the images etc.) via a separate connection. Then, there is a brief period of inactivity while the user browses through this information (which is the idle time mentioned earlier) until the next time that the user decides to access either a link on the same page or a new Web page altogether.

HTTP is an application layer protocol that requires a reliable transport mechanism at the lower levels in order to be successful. Since the Transmission Control Protocol (TCP) has been in wide use, and has been proven to work in a variety of environments, it was selected as the *de facto* reliable transport protocol of choice. TCP conducts a three-way handshake for each connection setup by HTTP to transfer the server objects, and it utilizes a congestion window (*cwnd*) for each connection in order to keep track of the number of outstanding packets within the network. TCP enters the slow start phase either at connection startup or after the connection has been idle for too long (usually more than a round trip time), where this *cwnd* is initialized to 1 [9]. TCP then probes for the available bandwidth by exponentially increasing its congestion window per round trip time (*RTT*). This is accomplished by increasing the congestion window by 1 for every successful acknowledgement (ACK) received by the sender before the expiration of the retransmission timer [7], [15], [16]. The congestion window is thus increased exponentially either until it equals the receiver's advertised window, or until a loss occurs in the network. If the latter holds true, then TCP assumes that there is congestion in the network inducing losses and therefore, it reduces the *cwnd* to half its value and transitions to the more conservative congestion avoidance phase. TCP maintains another variable, the slow start threshold (or *ssthresh*) in order to determine whether it should be in the slow start or congestion avoidance phase [2], [16]. If $cwnd < ssthresh$, then the flow is still in slow start; otherwise its in congestion avoidance. The bandwidth probing mechanism during slow start, although relying on an exponential window increase mechanism, actually takes

several *RTT*'s. Discovering the bandwidth in this manner is therefore, extremely inefficient. If the network has sufficient bandwidth (i.e., the transfer never moves to the congestion avoidance phase and just finishes in slow start without multiple packet losses), then in this case the duration of the transfer is directly related only to the *RTT* of the connection and not to the amount of available bandwidth along the path. So, the main problem with TCP's algorithms is that the pace of this algorithm is normally too slow for most web transfers since they are usually completed in only a few *RTT*'s. Recent measurements from a busy Internet server show that 85% of the packets are transferred by the server while the flows are still in their slow start phase [5]. This indicates that the bulk of the file (if not all of it) is actually transferred while the flow is still in the slow start phase probing for bandwidth. These statistics imply that a majority of the file transfers in the Internet occur sub-optimally due to the nature of the slow start algorithm.

Although it is very slow in discovering bandwidth, however, slow start is at the same time overly aggressive in probing for bandwidth because of the way it handles congestion window increase. The problem is that due to the exponential increase in the window size, it is possible to overshoot the congestion window by twice the amount of available bandwidth. Therefore, slow start is both, inefficient and aggressive, all at once. Short flows, which transfer most of their data during slow start, really suffer due to these inefficiencies, indicating that it might be possible to design other algorithms as alternatives to slow start that are much better suited for handling small file transfers.

This paper presents a simulation study involving alternatives to the TCP slow start algorithm. The goal is to provide a mechanism for performing the transfer of small files much more efficiently than is currently done with TCP. The new algorithms considered in this paper rely on previous network performance history as well as the *size* of the data in an effort to speed up the file transfer. Our methodology takes advantage of the work done on the TCP/SPAND project in that it uses some of their concepts and ideas. Although many of the concepts presented in this paper are similar to those in earlier works, the novelty of this algorithm involves looking at the size of the data to be transferred and then making a decision on the best possible way to transfer the file. The file is either transferred at a constant rate (based on the previous values of *cwnd*), or it is transferred by starting at a higher initial window, or it is transferred using just normal TCP (if the file is too large). We try to evaluate our algorithm on the basis of its efficiency (judged by the file transfer time) and its TCP Friendliness across a wide

number of connections and file sizes. The basic idea is that TCP is actually a very good protocol when it comes to long data transfers. It is only for short flows, when the slow start phase comprises a majority of the life of the flow, that TCP becomes inefficient.

The rest of this paper is organized as follows. Section 2 informs the reader on previous work done in addressing this problem of TCP slow start. Section 3 proposes improvements to slow start and establishes the conditions under which these new methods should be used. Section 4 provides analysis on the efficiency of our algorithm and the savings that can be achieved over TCP. Section 5 describes the simulation setup and methodology. The results of the simulation experiments are presented in Section 6. We conclude with our observations and acknowledgements in Sections 7 and 8 respectively.

2 RELATED WORK

There have been numerous proposals to improve upon the TCP slow start problem both, at the transport level and at the application level. In the transport protocol itself, one proposal involves increasing the initial window size to 2-4 segments, depending on the maximum segment size [1]. The upper bound for the initial window here is:

$$IW = \min(4 * MSS, \max(2 * MSS, 4380))$$

As mentioned in [1], this is a recommended value for the initial window that can be used at the beginning of the transfer (upon connection setup) or after a prolonged idle period. The window still gets initialized to 1 upon expiration of the retransmission timer. The problem with this proposal is that it would modify the TCP stack so that this algorithm gets utilized for all TCP flows. It does not take into consideration network conditions or the properties of the bottleneck link in order to decide how to transfer the file. The algorithm we're proposing has the added benefit that it utilizes the higher initial windows by first checking the network measurement history and the file size of the transfer. *Transaction TCP* (T/TCP) proposes to cache previous connection count history in order to get rid of the three-way handshake in certain situations to speed up connection establishment [3], [4]. T/TCP uses caches to maintain TCP control block information, e.g., smoothed RTT (*srtt*), RTT variance (*rttvar*), congestion avoidance threshold (*ssthresh*), and the maximum segment size (MSS) [3]. Although it does not provide details, this work also mentions the possibility of caching "congestion avoidance threshold." The problem is that many browsers and servers open multiple concurrent connections to a server anyway, and many servers don't support persistent connections [23]. *TCP control block interdependence* emphasizes temporal

sharing of TCP state, including the reuse of the congestion window of the previous connection [11]. Similar to this scheme, TCP Fast Start proposes reutilizing the congestion window size ($cwnd$), the slow start threshold ($ssthresh$), the smoothed round-trip time ($srtt$) and its variance ($rttvar$) [9]. Although all of these algorithms try to aggregate and share information to some extent, they do not take advantage of information regarding the file to be transferred to speed up the transfer.

There are several application-level approaches to tackle the inefficiencies of TCP slow start as well. Using multiple concurrent TCP connections can cause problems for TCP congestion control since these connections do not share any congestion information [9]. This also increases congestion because the group of flows as a whole is much more aggressive in probing for bandwidth than any single flow [23]. In addition, only a subset of these connections usually backs off upon experiencing congestion [23]. Several other solutions multiplex logically distinct streams onto a single TCP connection at the application level. These include Persistent-connection HTTP (P-HTTP) [24] and the MUX protocol [25]. These solutions have their own drawbacks as well. Since they are application-specific, each type of application would need to re-implement the same functionality [23]. Furthermore, they result in unnecessary coupling of the different streams of data: if packets from a particular flow get lost, the other flows might still stall needlessly because of the TCP semantics of guaranteed, in-order delivery [23]. Therefore, the flows are no longer processed independently.

In summary, there is very little information on how to manipulate the various TCP parameters in order to get the most optimal performance. Also, we do agree with some of the approaches that mention ways to share information and use previous network measurement information. However, we believe that the size of the data is an important parameter that is missing from all of these algorithms since it determines the network conditions that the transfer actually “experiences.” For instance, depending on the time scales of the congestion periods, a short flow might have enough bandwidth available to conduct its transfer whereas a long-lived FTP flow might see varying network conditions over the course of its transfer. Bandwidth between hosts can vary from kilobits to hundreds of megabits per second and, in general, networks exhibit a great deal of heterogeneity [12]. Therefore, a particular algorithm for data transfer might be optimal in one case but not in the other. Hence, there is a need for more *adaptive* algorithms that take into consideration various factors in making their decisions.

We now propose our set of algorithms that combine the concepts from some of the previous works with our idea of using the size of the transfer as one of the parameters in deciding how to transfer the data.

3 NETWORK ADAPTIVE SLOW START

The adaptive TCP slow start algorithm consults previous history regarding the flows to determine the amount of network resources available. Based on the estimate of the available capacity and the size of the transfer, it decides on one of three methods for performing the transfer.

The network measurement methodology is identical to the one used by TCP/SPAND [13]. We implement their concept of shared, passive measurements in our algorithm. The reason for sharing measurement information is that in the local sub-domain, there is plenty of bandwidth available and therefore, the bottleneck links are usually much further along the path. In such a scenario, any two hosts in the sub-domain would experience similar performance to distant hosts and so sharing of performance information is very useful [12]. Passive measurements provide the added benefit that no additional traffic is introduced into the network in the process of discovering the resources available. Earlier methods, such as Packet Pair and Packet Bunch Mode, require the generation of a lot of traffic in order to determine the measurement information and are extremely inefficient in using the existing bandwidth. Also, these techniques do not always produce accurate results. So, for our case, the traffic that the applications generate is itself used in order to determine the necessary network characteristics.

In our algorithm, we maintain a global state variable that keeps a history of the ending congestion windows ($cwnd$) and the smoothed round-trip times ($srtt$) of previous connections. Since the $srtt$ is an average estimate of the round-trip time over the course of the entire connection, we use only the value obtained from the most recent connection as the estimate for future connections. The congestion window, however, only represents a single instant in time (ie, the end-point) of the previous connection. Therefore, we pool information of ending congestion windows across all of the hosts in the local network, and aggregate this information using a low pass filter. The weights of the low pass filter are set (ie, they’re not dynamic), and the values of the congestion window and the smoothed round-trip time are updated upon completion of any given file transfer. We do not weight the previous history very heavily because we want to be able to adapt to changes in network conditions.

Besides maintaining history of previous network performance, we also try to utilize the size of the file being transferred in choosing our particular method of transfer. Previous studies have shown that network conditions that determine the amount of available bandwidth tend to remain pretty stable over time scales that are orders of magnitude higher than the usual time that a flow spends in slow start [9]. This implies that we can essentially assume that the available bandwidth stays constant over the course of the slow start phase. In our algorithm, we first calculate the time required for a flow to complete its slow start phase assuming that we know the available bandwidth. We use previous network history (namely the low pass filtered value of the congestion window) as an estimate of this value. In this calculation, we assume that the receiver ACKs each incoming segment and that there are no ACK losses in the network. With these assumptions, the slow start time (ss) is calculated as [2]:

$$ss = R (\log_2 W)$$

Here, R is the round-trip time and W is the size of the congestion window in terms of number of segments. We then calculate the number of packets that we can send in this time if we transmit at a rate equivalent to the stored congestion window value (call this value *maxpossible*).

$$maxpossible = (W \times ss)/R$$

We compare this value of the number of packets that can be sent with the actual size of the file to be sent and then choose one of the following methods:

In the first method, we first determine whether the actual file size is smaller than *maxpossible*. If it is, then we check to see whether the previous connection finished in the slow start phase or the congestion avoidance phase (by comparing the value of the *cwnd* from the previous connection with the *ssthresh* value of the same transfer). If in congestion avoidance, then we know that we've gone through the bandwidth discovery phase, and that our congestion window falls in the region of $[0.5 * maxwnd, maxwnd]$, where *maxwnd* is the maximum possible window that can be maintained without inducing losses in the network (essentially, this is the available bandwidth). With this being the case, we just perform the transfer at a constant rate equivalent to the current averaged value of the congestion window. It is important to note that this constant rate transfer is in some sense a TCP constant rate transfer and not a UDP transfer. What this means is that unlike UDP, which does no congestion control or backoff upon loss, our transfer follows the semantics of regular TCP when it experiences packet loss.

The second method takes care of the case when the previous congestion window falls in the slow start regime of that transfer. In this case, we know that the previous

flow was still in the bandwidth discovery phase and we can't be overly aggressive in conducting our transfer. Therefore, we initialize the starting window of our new transfer to half of the congestion window value as a conservative estimate to the amount of available bandwidth. If this estimate falls below 1 then we just initialize the window to 1. This way, we know that even when the congestion window is doubled in the next round, it won't induce losses right away if the original estimate hasn't overshot the amount of available bandwidth.

The final method is used when the file size of the new transfer is greater than the *maxpossible* value. The previous two methods are based on the assumption that the bandwidth is stable on the time scale of the entire slow start phase. However, we don't make the same assumption beyond this phase. Since the *maxpossible* value represents the number of packets that can be transferred during slow start, anything greater than this would spill over into congestion avoidance. Since we don't make the same bandwidth stability assumption for congestion avoidance, we decide that we can't optimize the transfer in the same way as the other two methods. Essentially, we are saying that the file is too big since we're only trying to optimize small file transfers. At this point, we just decide to use normal TCP to transfer the file.

Thus, we choose the most optimal algorithm based on the size of the file – this ensures that for any given set of network conditions and a given file size, we're making the right decision on how to conduct the transfer.

4 THEORETICAL ANALYSIS

To gain better understanding of the efficiency of this algorithm, we analyzed its performance in comparison to that of normal TCP Reno. The analysis applies to TCP without delayed ACKs. The performance measure that we have used in our simulations is the transfer time of the file, so we use the same metric in order to perform our analysis.

We make the following assumptions in evaluating the performance of our algorithm. First, we assume that the flow does not go into congestion avoidance and only stays in slow start. This assumption makes sense since our algorithm is geared towards providing maximum improvement in the transfer time of short flows that usually finish transfer while mostly still in the slow start phase. The second assumption we make is that there is enough buffer space and bandwidth so that there are no packet drops. This assumption is used to compare the gains in the best case scenario for both, our algorithm and for TCP Reno.

We compare the time required to transfer a flow of x packets. We know from [7] that in TCP slow start, the time of transfer is:

$$t_1(x) = (\text{ceil}(\log_2 x) + 1) * R$$

Here, R represents the round trip time of the connection. The amount of time required to transfer a flow of the same size (i.e., x packets) with our algorithm for packets of size P and an available bandwidth of B is:

$$t_2(x) = (P/B) * x + R$$

We can then calculate the minimum available bandwidth required so that the transfer time using TCP is greater than the time using our adaptive TCP algorithm. Essentially, this is the region of operation for our algorithm because we're only using our constant rate transfer if there is enough bandwidth available to transfer the entire file faster than with slow start. Let us use the notation $c \log_2(x)$ to denote $\text{ceil}(\log_2 x)$.

$$t_1(x) \geq t_2(x)$$

$$(c \log_2(x) + 1) * R \geq \frac{P}{B} * x + R$$

$$c \log_2(x) + 1 \geq \frac{P * x}{B * R} + 1$$

$$c \log_2(x) \geq \frac{P * x}{B * R}$$

$$B \geq \frac{P * x}{R} \left(\frac{1}{c \log_2(x)} \right)$$

Essentially, this indicates the minimum bandwidth-delay product required so that the time of transfer with our algorithm is lower than with TCP slow start. We see that this product is directly proportional to the length of the flow (which is $P * x$) and inversely proportional to the log of the number of packets being sent.

We then compare the two time functions.

$$T(x) = \frac{t_1(x)}{t_2(x)} = \frac{(c \log_2 x + 1) * R}{Ax + R}$$

Here $A = P/B$ (i.e., A is the time required to transmit a packet at an available bandwidth of B). We see that the time function for our algorithm is linear with a slope of A . Looking at the function $T(x)$ more closely we see that there is a range of x values (determined by the slope of the function $t_2(x)$) over which $T(x) > 1$ (i.e., our algorithm requires less time than slow start). If $A \gg 0$ and $A < 0.4$ then the two functions intersect fairly quickly and the range is small (spanning only a few packets). If $A > 0.7$ then the two functions don't intersect at all, indicating that there isn't sufficient bandwidth available in order to use the adaptive TCP algorithm. For reasonable values

of A (i.e., packet size and bandwidth), the range of packets falls approximately in $0 < x < 50$. This means that our algorithm is much better than TCP slow start when the amount of data to be sent is very small.

We also evaluated the percentage savings that can be achieved in the transfer time by using our algorithm.

$$\begin{aligned} \frac{t_1(x) - t_2(x)}{t_2(x)} &= \frac{(c \log_2 x + 1) * R - (A * x + R)}{(c \log_2 x + 1) * R} \\ &= 1 - \frac{\left(\frac{A}{R}\right) * x + 1}{c \log_2 x + 1} \end{aligned}$$

This function essentially reflects the same behavior as $T(x)$ except here, we can calculate the number of packets to send for maximal savings in time for a given topology and network conditions (i.e., available bandwidth, packet size and delay). As an example, consider the following sample values: $P = 8\text{kbits}$, $B = 0.5\text{Mbps}$, and $R = 200\text{ms}$. Performing the calculations, we see that although the range in which we gain some savings is $0 < x < 75$, we attain savings of 40% or greater only when the number of packets sent falls in $0 < x < 35$. Peak savings in transfer time are achieved by using our algorithm when the number of packets sent for this particular scenario is $x=8$. Therefore, we see that for reasonable values of bandwidth and delay, we achieve maximum savings by using our algorithm when the number of packets sent is very small.

We decided to run simulations to verify this behavior of our algorithm.

5 SIMULATION METHODOLOGY

We implemented our new algorithm in the **ns** network simulator [8]. The topology of interest to us is one where we're doing data transfer from one local area network to another across some wide area network. We consider the advantages of sharing information amongst hosts on the same LAN. This is motivated by the SPAND architecture, in which it is assumed that hosts in well-connected domains communicate over WANs (or networks in general) with unknown characteristics [10], [12]. These hosts reside within uncongested, high-speed, low-latency domains. Therefore, information for hosts in any given sub-domain is aggregated within that sub-domain. The simulation topology is depicted in Figure 1 [13].

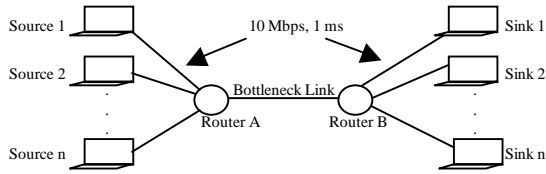


Figure 1: Simulation Topology

A connection per host is established between a portion of the sources on the left with their counterpart sinks on the right. The TCP packet/segment size is set to 1 KB. The size of the bottleneck buffer is 20 KB. The bottleneck router uses FIFO scheduling with drop-tail buffer management. The scenarios shown in Table 1 are considered (exactly the same as the ones used in the paper by Zhang et. al) [13].

Scenario	Bandwidth	Link Delay	Description
1	1.6 Mbps	50 ms	T1 speed terrestrial WAN link
2	1.6 Mbps	200 ms	T1 speed geo-stationary satellite link
3	45 Mbps	200 ms	T3 speed geo-stationary satellite link

Table 1: Scenarios (topologies) used for the simulations

The simulations are carried out in a manner similar to the simulations in Keshav’s TCP/SPAND paper. Each end-to-end flow sends 10 files to its corresponding sink, with a 10 s idle time in between each transfer. There is a *jitter* variable used to control the exact start time beyond this 10 s interval. The performance metric used in our case is the average completion time of the flows [13].

6 SIMULATION RESULTS

The performance of TCP with adaptive slow start is compared with the performance of TCP/NewReno. The granularity of the timer is changed to 10 ms, and this value is used for both, adaptive TCP and for TCP/NewReno. The reason for this is that we rely heavily on the previous values of *srtt*, which need to be fairly precise. The original TCP resolution of 200 ms is too coarse-grained for our purposes, and therefore, we decided to use the much finer timer resolution.

6.1 Varying the number of competing connections

In the first experiment, we try to see how the new algorithm compares to TCP with increasing number of connections. As mentioned above, the metric used is the

average time of transfer. The total number of connections is changed from an initial value of 1 to its final value of 30. Each connection transfers 10 files of size 40 KB (the size of a typical web page). The transfer time plotted on the graphs is the time obtained by averaging all of the existing flows. We first do a simulation run in which all of the connections are TCP/NewReno, and then we do another run in which all flows use our adaptive TCP. The graphs for these simulations are presented below in Figure 2. The graphs depict the average transfer time of the 40 KB file for each of the two scenarios. In comparing TCP/NewReno and our adaptive TCP, we see that our algorithm is much faster in transferring the file in both topologies. We see that the gains are especially more pronounced in the high latency scenario (i.e., Scenario 2).

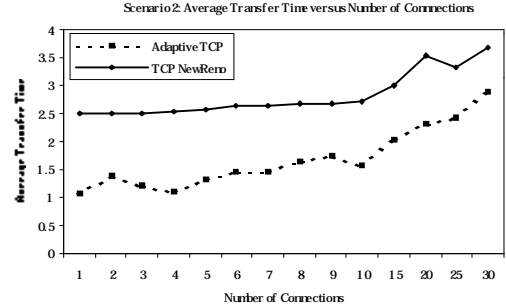
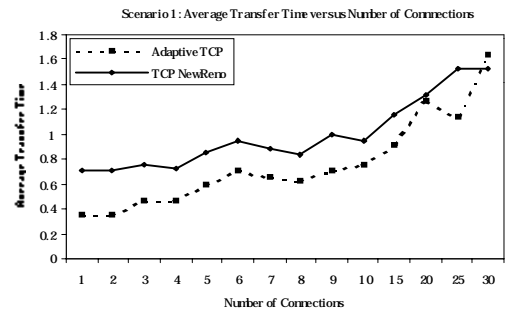


Figure 2: Performance with different number of connections

The gains obtained in Scenario 1 fall approximately in the range of 4 – 50 %, while for Scenario 2 the range is about 22 – 57 %. This does not include the one case in Scenario 1 when TCP/NewReno does better than our algorithm (the case with 30 connections) by about 8 %. From these graphs we see that using our adaptive algorithm instead of TCP slow start results in much better completion times.

6.2 Varying the transfer file size

In this part of the experiment, we try to measure the performance of our algorithm as we vary the size of the

file being transferred. The number of connections, in this case, is held constant at 10, and each of the connections transfers 10 files of the given size. Each point on the graphs is obtained by taking the average of the transfer times of all 100 of these transfers. The experiment is run for the first two scenarios listed in the table above. The graphs for this part of the experiment are presented below.

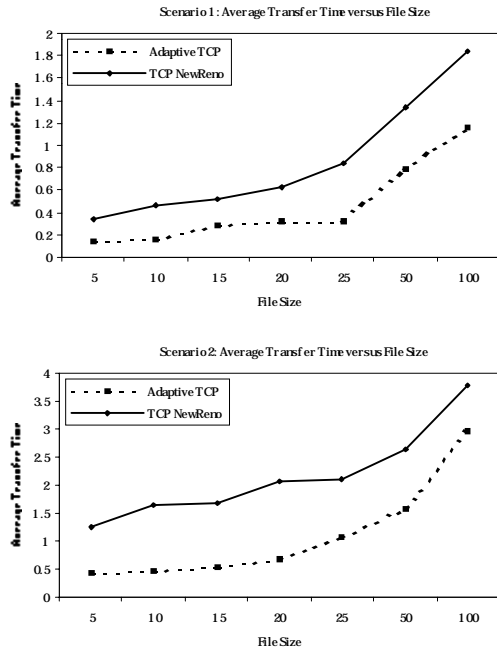


Figure 3: Performance with different file sizes in terms of number of packets

We can see from these graphs that our adaptive TCP algorithm performs better in all cases than regular TCP/NewReno – the average transfer time is always lower with our algorithm than with TCP/NewReno. We also see that the gains are bigger in the topology in Scenario 2, which has a higher bandwidth-delay product.

6.3 Effects of cross traffic

We wanted to observe the effects of adding cross traffic to our simulations. We use multiple concurrent long-lasting FTP sessions to model the cross traffic in the network. First, we fix the transfer size to 40 KB and notice the effects of varying the number of connections from 1 to 30. We use Scenario 2 for this simulation with 2 competing FTP flows. Next, we fix the number of connections at 20 and consider the effects of varying the file size from 5 packets to 100 packets. This simulation is performed with the topology in Scenario 3 with 5

competing long-lasting FTP flows. The graphs of these two simulations are shown below.

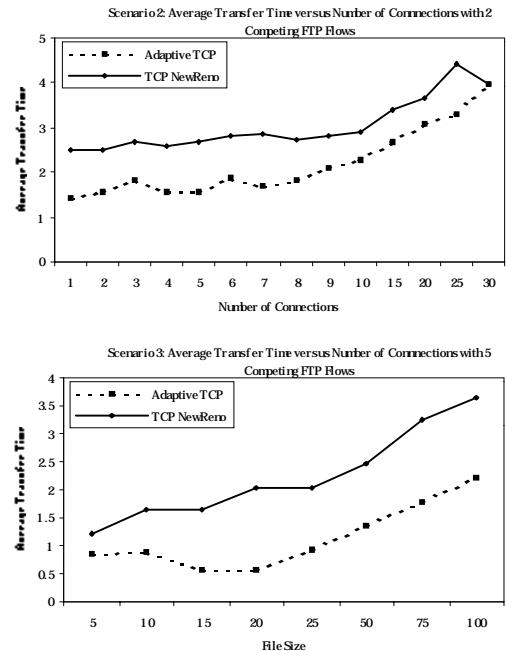


Figure 4: Performance with cross traffic

We observe once again that our algorithm performs much better than TCP, even when there is cross traffic present. The gains in the average transfer time when varying the number of connections range from 15 % to about 42 %. We also see that the gains are not as high as the number of connections is increased, probably because each connection's share is reduced and the scope for improvement is much lower [13]. When we vary the file size instead of the number of connections, we see that the average transfer time for our adaptive TCP algorithm is considerably lower than TCP/NewReno in all cases. The gains range from 29% (when the file size is 5 packets) to 71 % (for a file size of 20 packets). The gains are lowest for really small packets, mainly because the number of packets being transferred is too small to allow any significant improvements in transfer time. When the file size reaches 20 packets however, we realize maximum savings in transfer time. From that point, the percentage gain gets progressively worse as the file size is increased, although our algorithm still outperforms TCP/NewReno by a wide margin.

6.4 TCP Friendliness

An important consideration in making any changes to TCP is whether the new proposed algorithm is TCP friendly or not. The key idea here is that the new

algorithm should work in conjunction with TCP without adversely affecting the performance of regular TCP.

We show TCP friendliness by observing the transfer times of flows when there is a mixture of TCP flows. We run three separate simulation experiments: in the first, all connections use the adaptive TCP algorithm; in the second, half of the connections use adaptive TCP, the other half use TCP/NewReno; and in the final experiment, all connections use TCP/NewReno. In each case, the total number of connections is set to 20 and the file transfer size is varied. There is no cross traffic present in this setup. The topologies from scenarios 1 and 2 are explored. The following two graphs demonstrate the results of these experiments. For the most part, we see that the adaptive TCP algorithm transfers the files much faster than TCP/NewReno. Other than when the file sizes are very large (in comparison to the bandwidth-delay product of the network), using our adaptive TCP algorithm results in sharp reductions in the average file transfer times. More importantly, however, we see that these reductions do not come at the expense of other TCP flows.

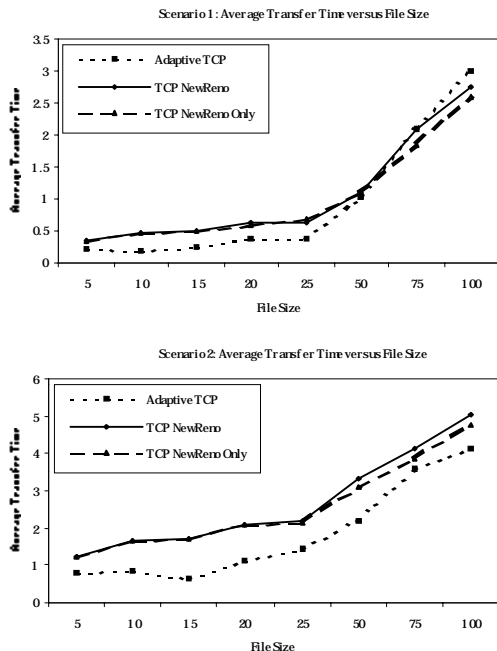


Figure 5: TCP Friendliness study

We observe that the transfer times for TCP/NewReno are almost equivalent regardless of whether flows using the adaptive TCP algorithm are present or not. This means that our algorithm only chooses the more aggressive modes of transfer when it knows that there is plenty of bandwidth present in the network and the file to be transferred is quite small. Otherwise, the algorithm chooses TCP/NewReno, which is why the discrepancy

between the transfer times of the adaptive TCP flows and the TCP/NewReno flows is small for larger file transfers.

The data, therefore, indicates that the adaptive TCP algorithm is TCP friendly and that it does not result in any kind of degradation in the performance of TCP/NewReno. In addition, we actually wanted to quantitatively measure the impact of mixing connections of both types of TCP's. For this purpose, we ran more simulations where the setup was as follows: we initiated 20 connections, each with 10 files to transfer with a fixed file size of 40 KB. There were three sets of simulations for each of the first two scenarios: the first set had only adaptive TCP flows, the second contained half adaptive TCP and half TCP/NewReno flows, and the last set included only TCP/NewReno flows. We measured the average transfer times for each of these three sets of simulations. The results are presented in the tables below.

	Scenario 1: 20 Modified TCP flows	Scenario 1: 10 Modified TCP/10 NewReno flows	Scenario 1: 20 TCP/NewReno flows
Modified TCP Delay (s)	0.982	0.723	-
TCP/NewReno Delay (s)	-	1.07	1.34
Average Delay (s)	0.982	0.897	1.34

Table 2: TCP Friendliness Study Average Delay Comparison (Scenario 1)

	Scenario 2: 20 Modified TCP flows	Scenario 2: 10 Modified TCP/10 NewReno flows	Scenario 2: 20 TCP/NewReno flows
Modified TCP Delay (s)	1.90	1.77	-
TCP/NewReno Delay (s)	-	3.02	3.10
Average Delay (s)	1.90	2.39	3.10

Table 3: TCP Friendliness Study Average Delay Comparison (Scenario 2)

We notice that while our adaptive TCP algorithm is the fastest in transferring the files, this speed does not come at the expense of the other TCP flows; in fact, it is actually beneficial to the TCP/NewReno flows that are present at the same time. For instance, the overall average file transfer time of the NewReno flows in the presence of adaptive TCP flows is slightly lower than when there are no adaptive TCP flows present at the same time. The reason for this is that the adaptive TCP flows are more efficient in utilizing their available bandwidth, and they finish their transfers faster, thus leaving the rest of the bandwidth for the TCP/NewReno flows. When all the flows are TCP/NewReno, however, they compete against each other throughout the course of the transfers for their share of the bandwidth. Another thing we realize from these tables is that the average transfer time for our adaptive TCP flows is lower in the

presence of TCP/NewReno flows than when there are no TCP/NewReno flows. This phenomenon also occurs for the same reasons as stated above. When there are only adaptive TCP flows present, they compete against each other for the available bandwidth. However, TCP/NewReno flows, when present, are more conservative in terms of their window increase algorithm, and so our adaptive TCP flows can use the available bandwidth more efficiently and finish in a shorter amount of time. It is important to note that in either case, the adaptive TCP flows always finish faster than the TCP/NewReno flows, implying that there are benefits to using this new algorithm.

7 CONCLUSIONS AND FUTURE WORK

In this paper, we have demonstrated how we can improve the performance of TCP when dealing with short flow transfers that are on the order of the bandwidth-delay product of the network. Since most short flows finish their transfers while they are still in the slow start phase, the inefficiencies of this phase seriously hurt these data transfers. Most improvements to TCP that have been proposed in the past are aimed at completely changing the behavior of the protocol in all cases, without realizing that what is optimal for short flows is not necessarily the optimal thing to do for longer flows. We believe that our approach is better than previous proposals since it picks the algorithms to be used based on the size of the file to be transferred. We do not just blindly apply the same algorithms to all cases. We also make use of previous network performance history to get a better estimate of the available bandwidth. We use these estimates to determine how much data we can pump through the network more aggressively.

There are a number of things that can still be improved upon with this work. For instance, a lot of the parameters for how we do averaging of congestion windows and round-trip times are still untested. There is some work involved in figuring out whether we can obtain parameters that will work optimally in general or if they will actually be topology dependent. Also, for the purposes of the simulation, we have assumed that we are going from one well-connected local domain to another across a WAN of unknown characteristics. However, we have not considered the case when connections are being made to different hosts that reside on different domains. The immediate open question that relates to this case is trying to figure out how to maintain previous performance history across these different domains. Clearly, we cannot just maintain a single value of the congestion window and round-trip time as we have done

with our simulations because the network conditions going to different domains across different LAN's will be different.

8 ACKNOWLEDGEMENTS

We would like to thank Professor Anthony Joseph and Tina Wong for their many helpful suggestions and comments.

References

- [1] M. Allman, S. Floyd, and C. Partridge. Increasing TCP's Initial Window. RFC-2414, September 1998.
- [2] M. Allman, C. Hayes and S. Ostermann. An evaluation of TCP with Larger Initial Windows. ACM Computer Communication Review, July 1998.
- [3] R.T. Braden. Extending TCP for Transactions – Concepts. RFC-1379, November 1992.
- [4] R.T. Braden. T/TCP – TCP Extensions for Transactions Functional Specification. RFC-1644, July 1994.
- [5] H. Balakrishnan, 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.
- [6] 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.aciri.org/floyd/papers.html>)
- [7] V. Jacobson and M. Karels. Congestion Avoidance and Control. In *Proc. ACM SIGCOMM 1988*, August 1988.
- [8] UCB/LBNL/VINT Network Simulator – ns (version 2). <http://www-mash.cs.berkeley.edu/ns>, 1997
- [9] V.N. Padmanabhan and R.H. Katz. TCP Fast Start: A Technique for Speeding Up Web Transfers. In *Proc. IEEE Globecom '98 Internet Mini-Conference*, Sydney, Australia, November 1998.
- [10] 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)*, Monterrey, CA, December 1997.
- [11] J. Touch. TCP Control Block Interdependence. RFC-2140. April 1997.
- [12] M. Stemm, R.H. Katz, and S. Seshan. A Network Measurement Architecture for Adaptive Applications.
- [13] Y. Zhang, L. Qiu, and S. Keshav. Optimizing TCP Start-up Performance. 1999.
- [14] K. Poduri and K. Nichols. Simulation Studies of Increased Initial TCP Window Size. RFC-2415. September 1998.
- [15] M. Allman, V. Paxson, and W. Stevens. TCP Congestion Control. RFC-2581. April 1999.
- [16] A.S. Tannenbaum. Computer Networks. pp521-545. Prentice-Hall Publishing.
- [17] S.Floyd, M.Handley, J.Padhye, and J.Widmer. "Equation-Based Congestion Control for Unicast Applications", February 2000.
- [18] J.C.Hoe. "Improving the Start-up Behavior of a Congestion Control Scheme for TCP". In *Proc. ACM SIGCOMM 96*, August 1996.
- [19] G.Miller, K.Thompson, and R.Wilder. Wide Area Internet Traffic Patterns and Characteristics. November 1997.
- [20] G.Miller and K.Thompson. The Nature of the Beast: Recent Traffic Measurements from an Internet Backbone. 1998.
- [21] J.Touch, J.Heidemann, and K.Obraczka. Analysis of HTTP Performance. August 1996.
- [22] J.Padhye, V.Firoui, D.Towsley, and J.Kurose. Modeling TCP Throughput: A Simple Model and its Empirical Validation. In *Proc. ACM SIGCOMM '98*, August 1998.
- [23] D. Andersen, D. Bansal, D. Curtis, S. Seshan, and H. Balakrishnan. System Support for Bandwidth Management and Content Adaptation in Internet Applications.
- [24] V.N. Padmanabhan, and J.C. Mogul. Improving HTTP Latency. In *Proc. Second International WWW Conference* (Oct. 1994).

- [25] J. Gettys. MUX protocol specification. WD-MUX-961023.
<http://www.w3.org/pub/WWW/Protocols/MUX/WD-mux-961023.html>,
1996.
- [26] G. Miller and K. Thompson. “The Nature of the Beast: Recent Traffic
Measurements from an Internet Backbone”, 1998.