Time-lined TCP for the TCP-friendly Delivery of Streaming Media

Biswaroop Mukherjee and Tim Brecht
Department of Computer Science, University of Waterloo, Waterloo, Ontario, Canada N2L 3G1.
bmukherj@shoshin.uwaterloo.ca,brecht@cs.uwaterloo.ca

Abstract

This paper introduces Time-lined TCP (TLTCP). TLTCP is a protocol designed to provide TCP-friendly delivery of time-sensitive data to applications that are loss-tolerant, such as streaming media players. Previous work on unicast delivery of streaming media over the Internet proposes using UDP and performs congestion control at the user level by regulating the application's sending rate (attempting to mimic the behavior of TCP in order to be TCP-friendly). TLTCP, on the other hand, is intended to be implemented at the transport level, and is based on TCP with modifications to support time-lines. Instead of treating all data as a byte stream TLTCP allows the application to associate data with deadlines. TLTCP sends data in a similar fashion to TCP until the deadline for a section of data has elapsed; at which point the now obsolete data is discarded in favor of new data. As a result, TLTCP supports TCP-friendly delivery of streaming media by retaining much of TCP's congestion control functionality. We describe an API for TLTCP that involves augmenting the `recvmsg` *and* `sendmsg` *socket calls. We also describe how streaming media applications that use various encoding schemes like MPEG-1 can associate data with deadlines and use TLTCP's API. We use simulations to examine the behavior of TLTCP under a wide range of networks and workloads. We find that it indeed performs time-lined data delivery and under most circumstances bandwidth is shared equally among competing TLTCP and TCP flows. Moreover, those scenarios under which TLTCP appears to be unfriendly are those under which TCP flows competing only with other TCP flows do not share bandwidth equitably.*

1. Introduction

It is widely believed [1] [23] [8] that congestion control mechanisms are critical to the stable functioning of the Internet. Presently, the vast majority (90-95%) of Internet traffic uses the TCP protocol [3] which incorporates congestion control [11] [26]. However, due to the growing popularity of streaming media applications and because TCP is not suitable for the delivery of time-sensitive data, a growing number of applications are being implemented using UDP [15].

Appears in the Proceedings of the International Conference on Network Protocols (ICNP), Osaka Japan, pp. 165-176, November 2000

Since UDP does not implement congestion control, protocols or applications that are implemented using UDP should detect and react to congestion in the network. Ideally, they should do so in a fashion that ensures fairness when competing with existing Internet traffic (i.e., they should be TCP-friendly). Otherwise such applications may obtain larger portions of the available bandwidth than TCP-based applications. Moreover, the wide-spread use of protocols that do not implement congestion control or avoidance mechanisms could result in a congestive collapse of the Internet [8] similar to the collapse that occurred in October, 1986 [11].

The work described here is motivated by these concerns. From the perspective of the application there is a need for a protocol that is designed for transporting data with deadlines over a network that provides no quality of service (QoS) guarantees. From the perspective of the network there is a need for a protocol that generates streams that compete fairly with the existing traffic and performs congestion control using robust mechanisms. To this end we have created a new protocol, called time-lined TCP (TLTCP) designed to support the TCP-friendly delivery of time-sensitive data over the Internet.

Contributions

- We have created a new transport protocol, called time-lined TCP (TLTCP), for delivering time-sensitive data over the Internet. We have devised a way for TLTCP to use the robust window-based congestion control of TCP without requiring that the data be delivered reliably. As a result, TLTCP competes fairly with TCP flows (and is TCP-friendly) over a wide range of network conditions. TLTCP associates each section of data with a deadline and uses a novel time-lined data delivery mechanism in TLTCP that uses these deadlines to keep track of the sections of data that are obsolete and ensures that no obsolete data is sent.

- TLTCP provides an interface that is more suited to continuous media applications than a simple end-to-end byte stream. We propose augmenting the present socket API that allows a sending application to specify a deadline when handing a section of data to TLTCP. The API also allows TLTCP to inform the receiving application of the gaps in the data being delivered. The proposed changes do not alter but extend the semantics of the present socket API.

- We have performed extensive simulation experiments to evaluate TLTCP. The experiments show that TLTCP indeed performs data delivery in a time-lined fashion. Furthermore, using data from our simulations we have quantified the effect of TLTCP flows on competing TCP flows. Our simulation results show that TLTCP is indeed TCP-friendly over a wide range of network conditions. In addition, the circumstances where TLTCP seems to be TCP-unfriendly are those under which TCP is unable to share bandwidth equitably.

The remainder of this paper is organized as follows. In Section 2 we describe related work and in Section 3 we explain our approach to the problem. Section 4 describes how our protocol would be used in conjunction with streaming media applications. How the TLTCP protocol operates is described in Section 5. We report the results from our simulation experiments using the *ns-2* network simulator [29] in Section 6. This is followed by conclusions in Section 7.

2. Related work

Previous work [4] [25] [18] [23] has examined rate-based algorithms for implementing TCP-friendly congestion control. In each case the sender throttles the rate at which it injects packets into the network in order to perform congestion control. To compete fairly with TCP, the sending rate is regulated, thus attempting to achieve the same throughput as a TCP-stream would if operating under the same conditions. These approaches are based on models that attempt to characterize TCP congestion control mechanisms [12] [14] [17]. As data is sent, the application measures or estimates values for the parameters of the model; such as packet loss rates, round-trip times, and timeout values. Using these parameters and the model the application periodically recomputes the appropriate sending rate. The proposed schemes differ primarily in the complexity and accuracy of the model used.

RAP [23] employs a relatively simple additive-increase, multiplicative-decrease (AIMD) model of TCP's congestion control mechanisms and is able to obtain relatively TCP-friendly behavior when competing for bandwidth with TCP Sack flows [6]. While it is targeted towards future Internet scenarios in which TCP Sack and RED [9] are widely deployed, it is not able to share bandwidth fairly with common implementations of TCP [6], TCP Tahoe or TCP Reno [22]. A significant advantage of TLTCP is that it is based on and is TCP-friendly with TCP Reno implementations, which is the most widely used TCP implementation in the Internet today [19] [20].

Sisalem *et al.* [25] propose a rate based equation that is designed to be used with RTP/RTCP. Their scheme dynamically computes an additive increase rate and also performs backoff. Experiments conducted with RED gateways are reported and show that their scheme does not share bandwidth equally under situations with low loss rates. We expect TLTCP to be more stable and share bandwidth equally under conditions with low loss rates.

Padhye *et al.* [18] describe and evaluate a rate control protocol based on a more detailed model of TCP throughput [17]. Although they are able show that their protocol is TCP-friendly under a variety of network configurations and conditions, the recomputation interval (the time between rate adjustments) must be chosen carefully. As can be seen from their simulation results the best recomputation interval may vary across different network conditions, making it difficult to use one recomputation strategy under a variety of circumstances. They also point out that they do not share bandwidth fairly with TCP streams when bottleneck link delays are small or large because it makes it difficult to obtain accurate estimates of loss rates.

Ramesh *et al.* [21] describe a number of potential drawbacks of model based approaches. In particular, they point out that several factors can result in inaccurate packet loss estimates in the model developed by Padhye *et al.* [17]. These inaccurate estimates can lead to under or over-allocation of bandwidth to non TCP flows. TLTCP is not model based but ACK-clocked and thus is not impacted by these drawbacks.

Cen *et al.* describe a streaming control protocol (SCP) [4], that uses a congestion window based policy for congestion avoidance. While their approach is similar to TCP they they do not perform retransmission and are not faithful to TCP in order to improve smoothness in streaming. The experimental results reported using an implementation of SCP on top of UDP show that the packet rates of competing SCP and TCP sessions differ significantly under a variety of network configurations.

Another scheme reported by Jacobs *et al.* [10] attempts to mimic TCP's congestion window in user space. The window size is used to estimate bandwidth which is then used to drive a *media pump* at the sender that uses UDP to send data to the receiver. Attempting to mimic the congestion window of TCP at the user level is likely to be inaccurate. This is because, the fact a message is written to the UDP socket does not mean that the packet has been released into the network. A mechanism in the user space would have no means of knowing if the message or its acknowledgement is waiting in the kernel buffers or traversing a link. TLTCP does not use a media pump to regulate its data sends but instead it uses a sliding window protocol like TCP. TLTCP also does not use UDP and is meant to be implemented in the kernel by making changes to the TCP stack. Furthermore unlike the schemes proposed in past, TLTCP uses the time-lined nature of continuous media to drive its data sends. Details of the scheme proposed by Jacobs *et al.* [10] are not provided and it is unclear how TCP-friendly such an approach would be.

3. Proposed approach

Unlike previous work, our approach is not based on models of TCP. Instead we propose a new protocol, called time-lined TCP (TLTCP), that is intended to be implemented at the transport level, and is based on TCP with modifications to support time-lines. Time-lines are used for the delivery of time-sensitive data to loss-tolerant applications such as streaming media players. Such applications are time-sensitive because data that arrives after the deadline by which it was meant to be played is not useful and will simply be ignored. Although using TCP will ensure that an application is TCP-friendly, TCP is unsuitable for such data transfer because it will potentially send obsolete data that would no longer be useful to the receiving application.

When using TLTCP, in addition to specifying the data and its size, an application includes the deadline after which the transport protocol should stop trying to send that data. TLTCP attempts to send the data until the deadline has expired, at which point it is presumed that the data would be obsolete by the time it would reach the receiver. Once a deadline has expired TLTCP abandons the obsolete data in favor of new data that is associated with later deadlines. Note that deadlines are defined to be relative to the sender. For best-effort service, the present scheme could be easily extended to make the deadlines relative to the receiver by factoring in the RTT estimates to the deadlines. TLTCP is intended to be implemented in the transport level of the kernel. Since TLTCP is ACK-clocked, it is able to mimic the behavior of TCP over a wide range of conditions. As TCP continues to evolve [11] [26] [2] [6] we believe that it would be relatively easy to implement a time-lined version of the protocol. However, we expect that it will be relatively difficult to produce accurate models and develop TCP-friendly protocols for each future variation of or modification to TCP.

4. Applications

The continuous media application that uses TLTCP is expected to handle the encoding scheme specific functions, while relying on TLTCP to perform congestion control and best effort data delivery. The sending application would typically calculate a schedule for the transmission of its data. Each *section* of data being sent (e.g., sequence of video frames, layers of video, or audio samples) would be assigned a deadline that is determined by the schedule (which would account for buffering and delay characteristics of the encoding and decoding schemes). The receiver application would begin playback after first receiving and buffering some portion of the data. During playback portions of data are decoded and presented to the user. If the sender is not be able to send all or even portions of a section before the deadline associated with the section expires, the receiver may be able to continue with a lower quality playback, depending on the application's ability to tolerate lost data.

For example MPEG-1 video [24] that has frames with varying degrees of importance for the playback application, I, P and B respectively. Roughly speaking, the I frames can be displayed *independently* while the P frames can only be displayed if the *previous* I or P frames has arrived. The B frames are *bidirectionally* encoded and cannot be displayed unless the previous non-bidirectionally encoded (I or P) frame as well as the next non-bidirectionally encoded (I or P) frame are delivered. Because of the bidirectional dependencies, the display order of frames differs from the order in which they are stored in a file or transported. For instance the display order of an MPEG-1 video may be, $\{I_1^1, P_1^1, B_1^1, P_2^1, I_1^2, P_1^2, B_1^2, I_1^3, \ldots\}$. However, the order in which this sequence is stored in an MPEG file will be $\{I_1^1, P_1^1, P_2^1, I_1^2, B_1^1, P_1^2, I_1^3, B_1^2, \ldots\}$.

TLTCP sections are created from an MPEG-1 file in the same order as they are stored but the deadlines are assigned according to the order of display. The same deadline is assigned to an I frame, the P frames directly dependent on the I frame, the P frames that are dependent on the P frames that depend on the I frame an so on. The B frames are assigned the same deadlines as the earlier frames they depend upon, but they are sent after the frames they depend upon. Thus in the example above the deadline assignments would be as follows. $\{\{I_1^1, P_1^1, P_2^1 : d^1\}, \{I_1^2 : d^2\}, \{B_1^1 : d^1\}, \{P_1^2 : d^2\}, \{I_1^3 : d^3\}, \{B_1^2 : d^2\}, \ldots\}$. The sending application would start by writing the encoded frames to the socket as described above and TLTCP would try to deliver the sections in the order they were written. However, if the available bandwidth is insufficient to deliver all of the section $\{I_1^1, P_1^1, P_2^1\}$ TLTCP may discard P_2^1 at the expiry of d^1 and start sending the more important frame, I_1^2 since it is associated with the later deadline d^2. In other words, if the bandwidth is insufficient TLTCP will discard the less important data and instead attempt to deliver the more important data that still has a chance of reaching the receiver in time for playback. Note that, if the available bandwidth decreases further (due to congestion), the sending application upon receiving feedback from the playback application may decide to change its transmission schedule and just send the I and P frames or even just the I frames so that the important frames have more time to get delivered. Reusing the example, the data sections handed to TLTCP in these two reduced bandwidth cases described above would look like $\{\{I_1^1, P_1^1, P_2^1 : d^1\}, \{I_1^2 : d^2\}, \{P_1^2 : d^2\}, \{I_1^3 : d^3\}, \ldots\}$ and $\{\{I_1^1 : d^1\}, \{I_1^2 : d^2\}, \{I_1^3 : d^3\}, \ldots\}$ respectively. The MPEG receiver on the other hand, will be able to continue playback but the quality of playback would worsen as more frames are skipped.

The API

The API for TLTCP has two main functions. First, the sending application needs to be able to specify to TLTCP segments of data along with their associated deadlines. Second, the receiving end needs to be able to deliver to the client application the received data along with information about where gaps are located. We propose augmenting the UNIX socket calls of `recvmsg` and `sendmsg` [28] for this purpose.

To see how the API would be used consider the following example. The server process first creates a `SOCK_STREAM` socket and connects it to the receiver to establish the data connection. Then the various fields of the `msg_header` structure are filled in before calling `sendmsg` with a `MSG_TL` flag used to indicate time-lined data. Pointers for each of the data sections to be sent by TLTCP are stored in an array of `msg_iov` structures. These are made up of a pointer to the data, `iov_base` and the size of the data, `iov_len`. The size of the `msg_iov` array is equal to the number of sections being written and is stored in the `msg_iovlen` field of the `msg_header`. Deadlines corresponding to the data sections are provided using an ancillary data message. The value of the deadlines are stored in `msg_control` field of `msg_header`, with the message type (`cmsg_type`) specified as, `TL_DEADLINE`. The length `cmsg_len`, is again equal to the number of data sections.

At the receiver end when `recvmsg` is called the `MSG_TL` flag indicates that the data received is time-lined. The receiver can then read the ancillary data pointed to by `msg_control`, in order to distinguish between the data and gaps. If a field in the ancillary data contains `TL_DATA` then the corresponding field of the msg_iov structure points to valid data and the application can store the pointer in order to retrieve the data later. On the other hand the ancillary data contains TL_GAP then the application needs to make a note of the size and location of the gap and take this into account during playback.

5. Functioning of TLTCP

As discussed previously, except for the additional mechanisms to support time-lines, the functionality and thus the data sending characteristics of TLTCP are similar to TCP. The following description of TLTCP is based on TCP-Reno. We assume that the reader is familiar with TCP-Reno and we use TCP to refer to TCP-Reno.

5.1 The Sender

The TLTCP sender accepts time-sensitive data from the application via the TLTCP API. Each section of data is associated with a deadline by which it should be sent. The sender maintains a linked list, called *time-line list*, that stores the deadlines for the time-lined data. A node in this list that stores the deadline and starting sequence number for the associated section of data. Note that the data itself is stored in the kernel buffers as TCP and the lowest_seqno field of the list node points to the first data byte of a section in the buffer.

The sender performs data sends as a normal TCP sender would until the expiry of the lifetime timer which indicates that the deadline for the current section of data has expired. It then selects the next section of data to be sent from the list and sets the lifetime timer to the deadline for this section. All of the data up to the lowest sequence number of the new section of data is discarded.

5.2 Lifetime Timer

In addition to the TCP timers, TLTCP has a timer called the lifetime timer. This new timer keeps track of the deadlines associated with the oldest data in the sending window (the minimum of the receiver's advertised window and the congestion window). The lifetime timer counts down in the same fashion as the TCP timers. When a lifetime timer expires any data associated with that deadline that has not already been sent is considered obsolete and is discarded from the sending window. In other words, in response to a deadline expiry the sending window is moved forward to sequence numbers that are not obsolete. TLTCP then attempts to send the data associated with the next deadline and the lifetime timer is set to that deadline. Furthermore, upon expiry of the lifetime timer the time-line list is updated to contain only entries for the data sections that are not obsolete. Figure 1 shows the sequence of actions that are taken after expiry of the lifetime timer. Due to expiry of the deadlines some data sections may not be delivered completely leaving *gaps* in the sequence of bytes that is delivered to the receiver.

Let us consider an example that illustrates how a TLTCP sender transports continuous media data to a receiver. Suppose that the sender has a send window size of 10 bytes. For simplicity assume single byte payload for all packets. The sender can then send 10

```
if ( Lifetime_tmr has EXPIRED ) {
 rem_expired_data(timeline_list, &buf);
 if (!timeline_list_empty()) {
   cur_node=get_cur_node(timeline_list);
   store_unacked_seq();
   move_window(cur_node.lowest_seq);

   set_lifetime_tmr(cur_node.deadline);
 }
}
```

Figure 1. Pseudo code of the actions taken on the expiry of lifetime timer.

consecutive packets. Further assume that an application has specified the deadlines for sequence numbers 10 to 19 and 20 to 29, as d_1 and d_2 respectively, where $d_2 > d_1$ (i.e., the deadline for packets 10 to 19 will expire before the deadline for packets 20 to 29). TLTCP sets the lifetime timer to the deadline d_1 and commences sending. Now suppose that when deadline d_1 expires only packets 10 to 14 have been sent. At this point TLTCP will abandon the sending of all the sequences from 10 to 19 and 20 will be the next packet to send. It will also set the lifetime timer to d_2 and continue to keep track of the unacknowledged packets from the obsolete data. This is done in order to preserve the semantics of the congestion window mechanism (for a detailed explanation see Section 5.4).

5.3 The Receiver

Upon expiration of the lifetime timer the sender discards all data associated with the current deadline that has not yet been sent. However, if the receiver is not informed of this it would consider the discarded data to be lost and reject packets from the new section because they are beyond its receive window. The receiver would continue to acknowledge the last received sequence number, which is now obsolete. On the other hand, since the sender has already discarded the obsolete data it would continue to send the current data and a deadlock would result. In order to prevent this deadlock, when data is discarded the TLTCP sender explicitly notifies the receiver of the change in its next expected sequence number. The expected sequence number update notifications also allow the receiver to keep track of the gaps in the stream. Information about where the gaps are located (along with the data) will eventually be passed to the application when it attempts to read the data.

Expected sequence number notifications are included with every packet by using 32-bits of the available TCP-options. We call this 32-bit field, seq_update. The receiver knows that it needs to skip sequence numbers whenever it receives a packet containing a seq_update value that is greater than its next expected sequence number and adjusts its next expected sequence number to the sequence number contained in the field seq_update.

5.4 ACKs for Obsolete Data

The sender needs to keep track of acknowledgments for obsolete data, in order to ensure that the send window is correctly sized and is permitted to advance as ACKs arrive for the obsolete data.

Reconsider the example described in Section 5.2, when the deadline d_1 expires, packets 10 to 14 have already been sent. At this point TLTCP keeps track of the fact that it might receive ACKs for packets 10 to 14 and removes packets 10 to 19 from its buffer. The sender then continues by sending data associated with the next deadline d_2. Packets 20, 21, 22, 23 and 24 are sent and the send window is full. Once the window is full, no more data can be sent until outstanding ACKs arrive. One way to *logically* view the current situation is to imagine the obsolete data occupying slots in the current send window. Thus the send window could be thought of as $\{10, 11, 12, 13, 14, 20, 21, 22, 23, 24\}$. When ACKs for obsolete data arrive, the sender's window is moved by the amount of data that is ACKed, thus allowing new sends. For example, if an ACK is received for sequence number 12 the window will move ahead by 3 sequence numbers (since ACKs are cumulative) and the sender may send three new packets 25, 26, 27. Thus keeping track of ACKs for obsolete data is necessary because these ACKs allow the window to move forward. In the example above, the logical window moves forward upon the receipt of the ACK for sequence number 12.

In order to recognize ACKs for obsolete data, TLTCP uses a vector to store the highest sequence sent and the last ACK received for each obsolete section that has unacknowledged data. The size of the vector is bounded by the window size. As the ACKs for obsolete data arrive the entries in the vector are freed and as more unacknowledged data becomes obsolete, new entries are added. Note that even though TLTCP keeps track of the sequence numbers of the unacknowledged data that is obsolete, it sends data from new sections instead of retransmitting obsolete data.

5.5 Handling Lost Packets

If a lost packet is detected prior to the deadline expiry for that data TLTCP will retransmit the lost packet. Thus, TLTCP attempts to reliably deliver data prior to the expiry of the deadline associated with the data. On the other hand, if the lost packet is obsolete, TLTCP sends the lowest unacknowledged packet that is current. This is similar to the actions that would be taken by TCP, except that TLTCP would transmit current data rather than retransmit (possibly) obsolete data as in the case of TCP.

To clarify how this works reconsider the above example but now suppose that the window size is 5. Assume that packets 10 to 14 have been sent and then due to a deadline expiry packets 10 to 19 are deemed obsolete. Now imagine that packet 10 is lost and this is detected by the sender either because of three duplicate ACKs or a retransmit timeout. The TLTCP sender would then send the next unacknowledged packet, in this case 20. This may result in behavior that is close to but not identical to TCP. In order to further illustrate this scenario we now compare the actions that TLTCP would take with those of TCP under the same conditions. The scenario is depicted in Figure 2. If this is the first time that packet 20 is sent then TLTCP behaves the same as TCP. When we say that TLTCP behaves the same as TCP, we mean that it sends a packet when TCP does. However, the sequence number of the data being sent may be different in each case. If in the case of TLTCP, the packet sent and ACK for the sequence number 20 are not lost and if in the case of TCP, the packet that TCP resends and its ACK are not lost then TLTCP's ACK for 20 would arrive at the same time as TCP's ACK for 10. These ACKs would clock the subsequent sends at the same time for both TCP and TLTCP .

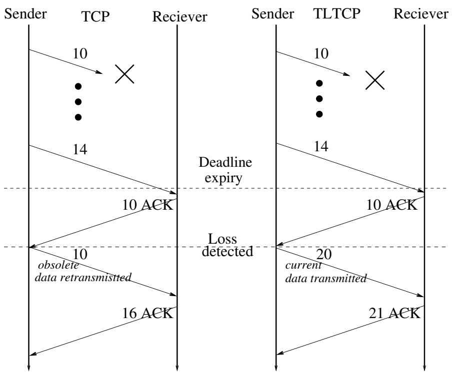

Figure 2. Example of a loss in obsolete data.

However, as shown in Figure 3, if packet 20 has already been sent (because of a window size greater than 5) and the ACK for it has not been received, TLTCP sends it again. We refer to this as a *pseudo-retransmission* since TLTCP is retransmitting data that may not require retransmission in order to ensure that a packet is sent when TCP would send a packet. If the ACK for the original send of packet 20 arrives prior to an ACK for the pseudo-retransmission then that ACK will clock TLTCP's subsequent send sooner than it would be clocked with TCP.

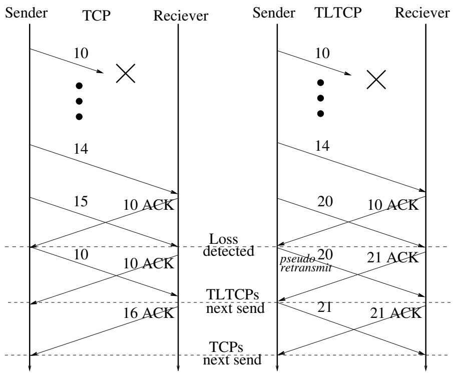

Figure 3. Example of a pseudo-retransmission.

Deviation from the behavior of TCP may also occur because of pseudo-retransmissions and a `seq_update` message. The loss of an obsolete packet, besides triggering a pseudo-retransmission, could cause subsequent losses of obsolete packets to be ignored as shown in Figure 4. Suppose in the original example of Section 5.2, packet 14 is lost in addition to packet 10. Under this scenario TCP would retransmit the lost packet and reduce its rate of sending by halving `ssthresh` [26] as a result of three duplicate ACKs or by reducing its congestion window due to a timeout. However, TLTCP's pseudo-retransmission would include a

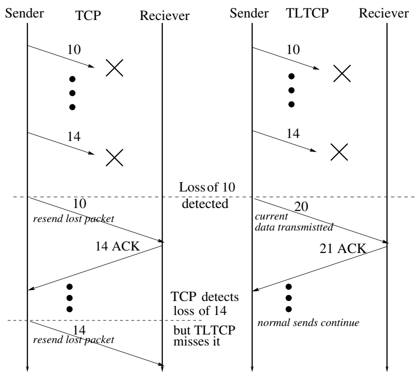

Figure 4. Example of a TLTCP missing a packet loss in the obsolete data.

`seq_update` that would cause the receiver to move its receive window beyond packets 10 to 19 and request packets 20 and beyond, therefore missing the fact that packet 14 is lost. In general, if before a packet loss is detected a new `seq_update` is received at the receiver, the receiver will ignore the missing data and request for data `seq_update` onwards. As a consequence, as shown in the example, TLTCP would be unable to detect the loss of packets subsequent to a pseudo-retransmission and would not experience the second slowdown.

6. Simulations

In this section we evaluate the behavior of TLTCP using simulations. There are several reasons why simulation experiments are more suitable than live Internet experiments for our purposes. In order to quantify TLTCP's TCP-friendliness we need to measure the effect of TLTCP traffic on TCP streams, discounting the impact of all *other factors* such as background traffic. In a live Internet scenario these factors are beyond our control and in most cases would add significant noise to the experimental results, while with simulations impact due to the *other factors* can be eliminated or factored into the results. Furthermore, for the measurements obtained in the baseline case (control experiment) to be meaningful the experiments must be run under the same conditions as the original experiment. Because the conditions of a simulation are reproducible, the baseline experiments can be run and valid measurements for comparison can be easily obtained. TLTCP is a new protocol and in order to test it thoroughly we need to vary several network parameters in a controlled fashion. Using simulations we are able to study the effect of varying several parameters over a wide range, one at a time, in order to quantify the effect of each one of them. In a live Internet experiment most of the network parameters, such as the number of flows competing at the bottleneck, are beyond our control while others like link delays and bottleneck bandwidth are difficult to vary. We have implemented TLTCP in the *ns-2* simulator [29] and have conducted several experiments to study TLTCP's time-lined data transport behavior and to quantify its TCP-friendliness.

6.1 Time-lined Data Transfer

Using a simulated network as shown in Figure 6, we begin two simultaneous data transfer sessions between a TCP sender and receiver and a TLTCP sender and receiver. We keep track of packet arrivals of both the streams in order to compare their data sending characteristics. For the sake of clarity in Figure 5, we use constant sized data sections of 700,000 bytes each associated with constant deadlines of 1 second to ensure that the whole section cannot be delivered within the given deadline. The other parameters used in this simulation are shown in Table 1 and justification for the values is provided in the next section.

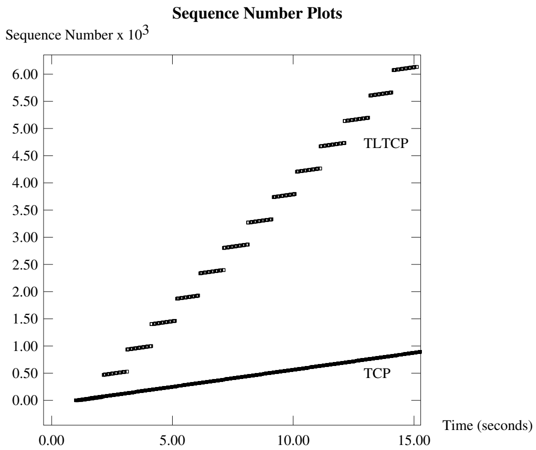

Figure 5. Data sending characteristics of TLTCP as compared to TCP.

Shown in Figure 5 is a plot of sequence number verses time where each sequence number represents a 1,500 byte data packet. It can be seen that in the case of the TLTCP flow, data is sent sequentially for the duration of one second (which is the deadline set for all sections of the data). At the end of the deadline there is a visible jump in the sequence number (to the next multiple of 467) and sequential sending resumes again for another second. Also note that the slopes of the continuous sections of the TLTCP plot and the TCP plot are the same. In fact, the lines are almost coincident if the discontinuities of the TLTCP trace are masked.

The observed discontinuities in the sequence number of the TLTCP stream stems from the fact that at the expiry of the deadlines TLTCP stops sending data from the expired section and starts sending a new section of data. New sections of data in this experiment begin with sequence numbers that are multiples of 467 ($\lceil 700000 \div 1500 \rceil = 467$), indicating that TLTCP is starting to send a new section. Throughout our experiments this pattern of data sending is observed in TLTCP. It can thus be inferred that TLTCP indeed performs data transfer in a time-lined manner. The fact that the slopes of the continuous sections of TLTCP's packet trace and that of TCP are the same implies that they consume equal bandwidth. It can be seen from the graphs that each of the streams consume approximately half of the 1.5 Mbps bandwidth ($900 \times 1500 \times 8 \div 14 = 771428.57$, where approximately 900 packets of 1500 bytes are delivered in 14 seconds by each stream).

6.2 TCP-friendliness

In several studies [4] [23] [18] TCP-friendliness has been interpreted and measured by the ability of non-TCP flows to equally share bandwidth with TCP flows. This is typically measured by observing the throughput obtained by several flows (both TCP and non-TCP) simultaneously operating over the same bottleneck link and determining the bandwidth shares of each flow.

We consider two main metrics for examining the extent to which the flows share bandwidth equally. The *friendliness ratio* [23] [18], F, is the ratio of the mean throughput observed by non-TCP flows (TLTCP flows in our case), $\overline{T_{TLTCP}}$, to the mean throughput obtained by TCP flows, $\overline{T_{TCP}}$, $F = \overline{T_{TLTCP}}/\overline{T_{TCP}}$. Since the friendliness ratio does not expose variations in observed bandwidth in individual flows we also consider the ratio of the maximum observed bandwidth to the minimum observed bandwidth [18]. We call this the separation index, S. We examine the separation index across all flows in an experiment. In the experiments with both TCP and non-TCP flows we call this measure S_{MIX}, whereas in the experiments where only TCP flows are present we call it S_{TCP}.

In all of our experiments we use a total of N flows (where N is even) with an equal number of competing TLTCP and TCP flows $(N/2)$. As a *baseline* for comparison we also run experiments under the same conditions with all N flows being TCP flows. In order to produce a metric similar to F when only TCP flows are considered we compute the ratio of the mean throughput of one half of the TCP flows to the mean throughput of the other half. The value of F will vary depending upon which of the $N/2$ flows are chosen for each half. Therefore, we compute and consider two extremes for F, F_{worst} and F_{best}. F_{worst} computes the worst possible value of F as the ratio of the mean bandwidth of the $N/2$ highest bandwidth flows to the mean bandwidth of the $N/2$ lowest bandwidth flows, $F_{worst} = \overline{max_{n/2}(allflows)}/\overline{min_{n/2}(allflows)}$.

F_{best} on the other hand, computes the best possible F. This is done by sorting the flows by bandwidth and dividing the flows into two groups, odd ranked ($oddflows$) and even ranked flows ($evenflows$). Then we compute the ratio of the maximum of the mean of the odd and even flows $max(\overline{oddflows}, \overline{evenflows})$, to the minimum of the mean of the odd and even flows $min(\overline{oddflows}, \overline{evenflows})$. $F_{best} = max(\overline{oddflows}, \overline{evenflows})/min(\overline{oddflows}, \overline{evenflows})$
Since the TCP flows themselves do not share bandwidth equally if their round-trip times are not equal [13] [7], we consider N sources configured symmetrically (as shown in Figure 6) such that the end-to-end delays of all the streams are equal. In all of our experiments each sender is continuously sending data to the corresponding receiver. We choose our initial set of simulation parameters, shown in Table 1, to be representative of Internet traffic. Later experiments consider the impact that changes to some of these parameters have on the TCP-friendliness of TLTCP.

The bottleneck link has a bandwidth of 1.5 Mbps, which is representative of a T1 link. We use a 1,500 byte packet size, which is a common size of packets seen in the Internet [5]. A maximum receiver window of 10 packets (15,000 bytes) is used which is near the higher end of the default values used for typical TCP implementations [27]. We assume that all the data transfers are unidirectional and therefore set the ACK size to 40 bytes, which is the size of a TCP ACK with no payload. The source and destination hosts connect to the bottleneck link with a 10 Mbps link which represents a local area network. Previous simulation results [16] suggest that for TCP to share bandwidth evenly among a large number of flows a bottleneck router queue needs to be provisioned to hold 10 times as many packets as the number of flows. Therefore, in order to ensure that TCP shares bandwidth equally we provision the queue at the bottleneck router to hold 400 packets. All the experiments are run for a simulated time of 500 seconds and data collection begins after the first 50 seconds to avoid the transient effects of startup. The TLTCP flows are given sections of 700,000 bytes each and the deadlines for these sections are set at 5 seconds. This corresponds to a maximum data rate of 1.12 Mbps. This is intentionally chosen to be high in order to thoroughly exercise the time-line specific mechanisms of TLTCP.

Parameter	Value
Packet size	1,500 bytes
ACK size	40 bytes
Bottleneck link BW	1.5 Mbps
Bottleneck link delay	20 ms
Router buffer size	400 pkts
Source/Dest link BW	10 Mbps
Source/Dest link delay	2 ms
Receiver max window size	10 pkts
Simulated time	500 sec
Size of TLTCP sections	700,000 bytes
Deadlines for TLTCP sections	5 sec
Total number of flows	30

Table 1. Default simulation parameters.

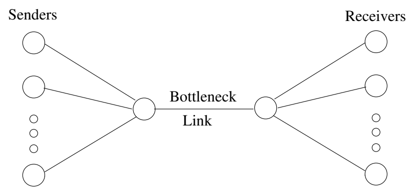

Figure 6. Topology used for simulations.

6.2.1 *Varying the Number of Flows*

In our first set of experiments we increase contention at the bottleneck by increasing the number of competing flows in order to study the resource sharing behavior of the TLTCP flows. As seen in Figures 7 and 8 across the range of flows used in the experiments TLTCP obtains good friendliness ratios and separation indices except when a total of 50 flows is reached. While TLTCP does not share bandwidth fairly at this point, it is important to notice that in the baseline case, 50 TCP flows competing amongst themselves under the same conditions do not share bandwidth fairly. This can be seen in F_{worst} in Figure 7 and S_{TCP} in Figure 8, where the values are not close to 1.

The situation where a number of TCP streams compete over a single bottleneck router has been studied previously by Morris [16]. He has observed that if there are a large number of competing flows, TCP's congestion control mechanisms fail to ensure fair

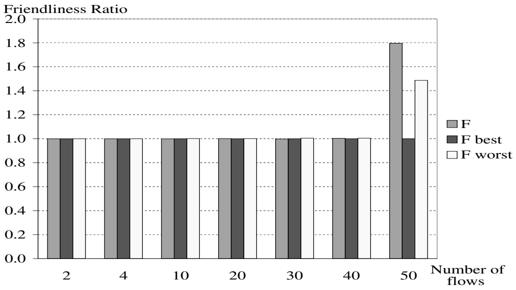

Figure 7. Varying flows: friendliness ratios

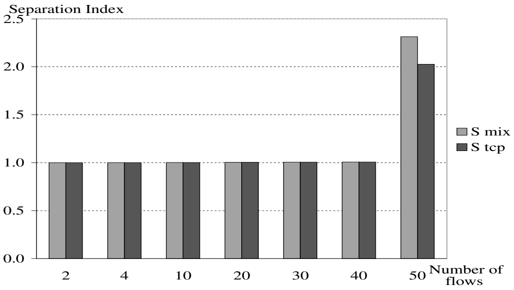

Figure 8. Varying flows: Separation indices

sharing of the bottleneck bandwidth. As a result of the high packet loss rates that occur in this situation and subsequent timeouts, the bandwidth obtained by competing flows is highly variable. Morris suggests that when the number of flows exceeds 10 times the queue size of the bottleneck router TCP does not share bandwidth equally. In our experiments, with a bottleneck buffer queue of size 400, the fairness ratios and separation indices are close to the ideal value of 1 for up to 40 TCP and TLTCP flows $(400/10)$. However, with a total of 50 flows the queue size is less than 10 packets per flow $(50 > 400/10)$ and thus the flows (TLTCP and TCP) do not share the bandwidth equitably. For still larger number of flows TCP's fairness detoriates further and thus the notion of TCP-friendliness looses its meaning. We therefore do not consider larger number of flows.

TLTCP's congestion control mechanisms are based on TCP. It is thus expected that the sharing behavior of TLTCP would be no better than that of TCP. It can be seen from Figures 7 and 8 that in the experiments with a mix of TCP and TLTCP flows, higher values for the friendliness ratio and separation index are observed as compared to the baseline experiment with just TCP flows. This is because in the experiments above TLTCP flows do not reduce their data rates as much as the competing TCP flows during congestion. As described in Section 5.5, TLTCP performs a pseudo-retransmission in response to a loss of obsolete data and cannot keep track of subsequent losses in the obsolete data. In the case of 50 competing flows, due to heavy contention at the bottleneck, the packet loss rates are high and the data rates are low As a result, there is a greater likelihood of multiple losses for obsolete data in some TLTCP flows. Since these TLTCP flows are unable to detect some of these losses they do not reduce their sending rates during congestion as much as the competing TCP flows, thereby obtaining a larger share of the bandwidth. By examining the individual flows we observe that during the simulation run there are fewer retransmissions for most of the TLTCP flows than the competing TCP flows, confirming that the TLTCP flows indeed miss some of the packet losses and as a consequence do not reduce their data rate as often as the competing TCP streams. Unless otherwise stated we use a total of 30 flows for our remaining experiments. This ensures that the bottleneck router has sufficient buffer space and therefore decreases the likelihood that TCP flows will not share bandwidth equally.

6.2.2 Varying the Maximum Window Size

In this section, we consider the impact of increasing the maximum receiver window sizes on the TCP-friendliness of TLTCP. As noted in Section 5.5, the scenarios that cause the behavior of TLTCP to deviate from that of TCP occur when there are multiple packet losses in the obsolete data. There is a greater likelihood of this occurring with larger window sizes, since there is a possibility of more unacknowledged obsolete data in this case. Moreover in both TCP and TLTCP large receiver windows increase the possibility of greater variations in send window sizes among competing flows.

In the Figures 9 and 10 we show the friendliness ratios and separation indices respectively for window sizes of 7,500, 15,000, 30,000, 60,000 and 120,000 bytes respectively. The sizes 7,500 and 15,000 were chosen to loosely correspond to default window sizes commonly used in TCP implementations [27]. The remaining values were chosen to significantly exceed these commonly used sizes. The results of these experiments demonstrate that under the conditions used for these simulations TLTCP and TCP share bandwidth fairly when the receiver window size is within the ranges typically used as defaults in current TCP implementations.

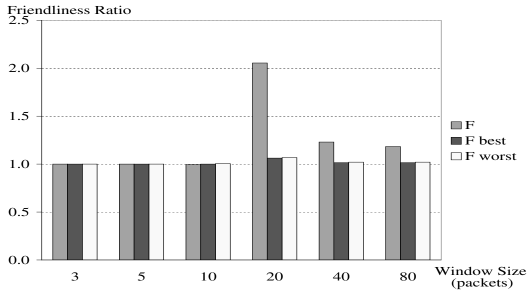

Figure 9. Window: TCP-friendliness ratios

However when the maximum window size is 20 packets, there is unequal sharing of the bandwidth. In the experiment with only TCP flows and a window size of 20, the friendliness ratio is seen to be close to 1 but the separation index is close to 2. This is an instance where the separation index is a valuable metric in uncovering unfriendliness. It can also seen from Figure 9 that the friendliness ratios in both of these cases (i.e., with just TCP flows and the with a mix of TLTCP and TCP flows) improve considerably when the receiver's window size is further increased to 40. Again, even though the friendliness ratios for the TCP only cases

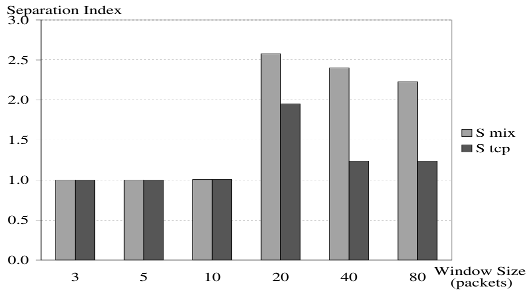

Figure 10. Window: Separation indices

(with maximum window sizes of 40 and 80) are close to 1 the separation indices indicate that there are disparities in the throughput of the individual streams. It is also observed that the results for the window size of 80 are fairly similar to those for 40.

It is unclear to us why the flows are less fair with a window size of 20 than with larger window size of 40 and 80. We speculate that the increase in unfriendliness when the window size is 20 is because a larger window size may cause the bandwidth sharing to be unequal.The sender's congestion window in all the flows varies from a minimum of 1 segment to a maximum of the receiver's advertised window. Ideally when all the flows are in equilibrium they would have equal window sizes and would thus achieve the same throughput. But this equilibrium is not reached because the congestion control mechanisms of both TCP and TLTCP keep changing the size of the congestion window by additively incrementing it when there are no losses and multiplicatively decrementing it when a loss is inferred. Additionally, since packets are forwarded in routers using a FIFO discipline (instead of per flow forwarding) some flows may experience bursty losses while others may experience no losses at all. The flows that experience the losses reduce their congestion window while others keep incrementing it, thus resulting in the disparity in observed throughput. A large receiver window (such the ones used in these experiments) increases the disparity among the flows as it allows the flows without losses to increase their window size to a larger extent (up to the large receiver window limit).

We also believe the reason that the results for the window sizes of 40 and 80 are similar and indicate increased friendliness is that the trend towards unfairness is likely to be self-limiting. That is, after a point increasing the receiver window size is not likely to result in an appreciable difference in the friendliness metrics observed for both TCP and TLTCP. The reason for this is that in most cases a flow will be able to increase its window to a limited size before experiencing a packet loss and consequently reducing it. As a result, most flows would not be able to significantly increase their sending windows to sizes much larger than the average as they would experience packet losses before reaching the limit. By examining the traces from our experiments we find out that in spite of doubling the maximum possible window size in each step, the average acquired window sizes across the flows in each of the experiments are indeed similar.

In the experiments with a mix of TLTCP and TCP flows, by examining the traces we observe that the TLTCP flows are the ones that obtain greater bandwidths. This is because of the fact that TLTCP cannot infer multiple packet losses in obsolete data. If a TLTCP flow has a large window size as in this experiment, it is likely to have more obsolete data in the sending window. This in turn means that there is a larger likelihood of multiple packet losses in obsolete data. Thus, with a larger receiver window such a TLTCP stream is likely to increment its window more than a TCP stream and would continue to do so until a loss is detected. Therefore on an average TLTCP streams obtain greater throughput with large maximum receiver window sizes. But note that the extent of the disparities in the throughput is not expected to get much worse for still larger windows because of self-limiting nature of the unfairness described above.

6.2.3 Varying the Propagation Delay

It is known that flows between different pairs of hosts in the Internet would encounter a wide variety of round-trip delays. It is thus important that a transport protocol be able to function properly across a wide range of round-trip delays. Dealing with a large range of round-trip delays has been reported as a problem with existing rate-based streaming media protocols. In their work on TFRCP, a rate-based protocol, Pahdye *et al.* [18] report that with *small* round-trip delays TFRCP behaves aggressively as compared to TCP, therefore obtaining a larger share of the bottleneck bandwidth than the competing TCP flows. They also point out that with *large* round-trip delays and comparatively small rate recomputation intervals, TFRCP is unable to accurately estimate loss rates and as a result its performance is highly variable.

An advantage of TLTCP when compared with rate-based protocols is that it is ACK-clocked and it uses the ACK-based round-trip timing mechanisms of TCP. Therefore, we expect that TLTCP will be able to react more quickly to traffic fluctuations and provide stable behavior over a wider range of operating conditions than rate-based protocols. In the next set of experiments we study TLTCP's behavior over a large range of bottleneck delays.

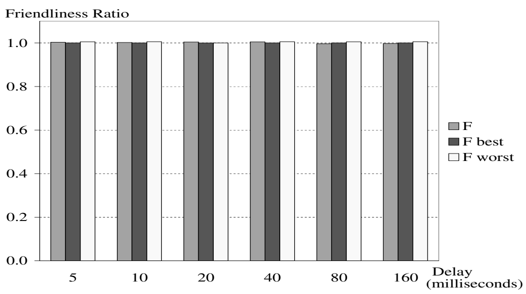

Figure 11. Varying delay: friendliness ratios

It can be seen from the figures that the friendliness ratios and separation indices obtained for TLTCP are very close to 1, as are those obtained for TCP. The observation that TLTCP and TCP flows are able to share the bandwidth equitably over a wide range of bottleneck delays indicates that the lifetime timer expiry events in the TLTCP flows do not significantly affect the accuracy of round-trip timing mechanisms. In addition, by examining the traces of our experiments we saw that, on average, the round-trip estimates of the TLTCP flows are close to that of the TCP flows.

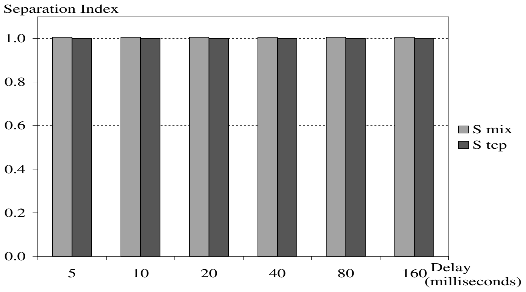

Figure 12. Varying delay: Separation indices

6.2.4 *Varying the Deadlines*

In the same way that large window sizes increase the likelihood that the behavior of TLTCP deviates from that of TCP the time-line chosen can also impact TLTCP. Clearly with large enough deadlines it will be possible to send all the packets of a section prior to its deadline and TLTCP will operate in a manner that is identical to TCP. However, as deadlines become smaller the likelihood of having to deal with losses in obsolete data increases, Therefore, in the next set of experiments we examine the impact of a range of deadlines on the friendliness of TLTCP.

Figure 13 shows fairness ratios and separation indices for a variety of deadline intervals, from 0.5 seconds (which corresponds to the resolution of timers in common implementations of TCP) to 62.5 seconds. Since there is no notion of time-lines in TCP, we compare the results of an experiment with 15 TLTCP flows and 15 TCP flows to another experiment where all the 30 flows are TCP. The results show that TLTCP operates fairly over a large range of

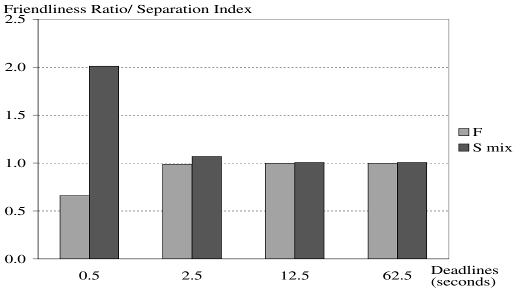

Figure 13. Varying data deadlines.

deadlines. However, for very short deadlines TLTCP is not able to share bandwidth equitably. In this case the deadline interval is 0.5 seconds. This corresponds to a data rate of 11.2 Mbps per stream (with 30 such streams) over a 1.5 Mbps link as the size of the section remains 700,000 bytes. It is interesting to note that in this instance TLTCP streams obtain *lower* throughput than the competing TCP streams, unlike the other experiments where the TLTCP streams obtain higher throughput.

The reason for this is the twofold impact of short deadlines on TLTCP's data sends. First, very few packets are sent in sequence before the deadline expires and data from the next section needs to be sent. Second, `seq_update` messages are sent frequently because the deadlines expire frequently. Note that the `seq_update` messages are a part of the new data packets that are sent. With small deadlines of 0.5 seconds we observe that in these experiments a TLTCP sender is able to send very few packets before the next jump in data sequence is indicated by a `seq_update` message. Since there are just a few packets being sent in each section, if a loss occurs, there is a high likelihood that before three subsequent packets are received at the receiver a `seq_update` message will reach the receiver. If less than three packets reach the receiver after a loss and before a `seq_update`, the fast-recovery mechanisms will not be triggered. Therefore, due to the small number of packets in each section that reach the receiver TLTCP streams are not able to reduce their sending rate by using fast recovery. This is confirmed by examining the trace files where we found that far fewer instances of fast-retransmit and fast-recovery are observed in the experiment with the deadlines of 0.5 seconds than with deadlines of 2.5 seconds. So instead of reducing their congestion window by half using fast recovery, the TLTCP flows experience timeouts that abruptly reduce their sending window to one segment. This is why with very small deadlines TLTCP streams obtain a smaller portion of the available bandwidth.

Note that in the scenario described above (with a deadline of 0.5 seconds) only a small number of packets are actually sent while most of the packets are discarded at the sender because of the expiry of the corresponding deadline. This is clearly undesirable for a real application and indicates that the deadlines are not set properly. An application using TLTCP would attempt to maximize the amount of data that reaches the receiver and reduce the dropping of packets. In a situation where there is a lot of data being dropped the application is expected to set larger deadlines or reduce the size of the section. Therefore, the unfairness observed in the experiment with the deadlines of 0.5 seconds (Figure 13) is unlikely to occur when TLTCP is being used by a real application.

We conclude this section by noting that TLTCP not only transports data in a time-lined fashion but does so in a TCP-friendly manner over a wide range of window sizes, deadlines, round-trip times and competing traffic. Furthermore, most of the conditions under which TLTCP flows appear to be unfair to TCP flows are the conditions under which TCP itself is unable to share bandwidth equitably.

7. Conclusions and Future work

The paper proposes a new protocol time-lined TCP , for the delivery of time-sensitive data over the Internet. Remaining within the confines of TCPs window based congestion control (designed for reliable data transfer), TLTCP attempts to deliver non-contiguous, time-sensitive data and is thus, suitable for continuous media players and other applications that send time-sensitive data. It is designed to compete fairly with the existing traffic in the Internet. Augmentations to the present socket calls are proposed to allow TLTCP to accept data with deadlines from the sending application, as well as, deliver the data received and indicate gaps to the receiving application. Finally we present the results of our simulations that show that TLTCP is likely to compete fairly with the existing traffic in the Internet.

In future we intend to integrate TLTCP into the kernel, and test it in the Internet. In order to facilitate deployment we intend to modify a TLTCP sender, such that a TLTCP sender can interoperate with TCP acting as the receiver. This will allow us to leverage the installed base of TCP for streaming media servers that use TLTCP . We also intend to further explore the possibility of *backing-off* on the strict friendliness requirements in order to provide better performance to the applications.

8. REFERENCES

[1] B. Braden, D. Clark, J. Crowcroft, B. Davie, S. Deering, D. Estrin, S. Floyd, V. Jacobson, G. Minshall, C. Partridge, L. Peterson, K. Ramakrishnan, S. Shenker, J. Wroclawski, and L. Zhang. Request for comments: 2309, April 1998.

[2] L. Brakmo and L. Peterson. TCP Vegas: End to end congestion avoidance on a global Internet. *IEEE Journal on Selected Areas in Communications*, 13(8), October 1995.

[3] CAIDA. Traffic workload overview, http://www.caida.org/Learn/Flow/tcpudp.html.

[4] S. Cen, C. Pu, and J. Walpole. Flow and congestion control for Internet streaming applications. In *Multimedia Computing and Networking*, 1998.

[5] K.C. Claffy. Internet measurement and data analysis: topology, workload, performance and routing statistics. In *NAE Workshop*, 1999. http://www.caida.org/Papers/Nae/.

[6] K. Fall and S. Floyd. Simulation-based comparisons of Tahoe, Reno, and SACK TCP. *Computer Communication Review*, 26(3):5–21, July 1996.

[7] S. Floyd. Connections with multiple congested gateways in packet-switched networks, Part 1: One-way traffic. *ACM Computer Communications Review*, 20(5):30–47, 1991.

[8] S. Floyd and K. Fall. Promoting the use of end-to-end congestion control in the Internet. *IEEE/ACM Transactions on Networking*, to appear.

[9] S. Floyd and V. Jacobson. Random early detection gateways for congestion avoidance. *IEEE/ACM Transactions on Networking*, 1(4):397–413, August 1993.

[10] S. Jacobs and A. Eleftheriadis. Streaming video using dynamic rate shaping and tcp congestion control. *Journal of Visual Communication and Image Representation*, 9(3):211–222, September 1998.

[11] Van Jacobson. Congestion avoidance and control. In *ACM SIGCOMM*, pages 314–329. ACM Press, August 1988.

[12] J. Mahdavi and S. Floyd. TCP-friendly unicast rate based flow control, January 1997. http://www.psc.edu/network-ing/papers/tcp_friendly.html.

[13] A. Mankin. Random drop congestion control. In *ACM SIGCOMM*, pages 1–7, 1990.

[14] M. Matthis, J. Semke, J. Mahdavi, and T. Ott. The macroscopic behavior of the TCP congestion avoidance algorithm. *Computer Communication Review*, 27(3), July 1997.

[15] A. Mena and J. Heidemann. An empirical study of real audio traffic. In *Proceedings of IEEE Infocom*, Tel-Aviv, Israel, to appear. IEEE.

[16] R. Morris. TCP behavior with many flows. In *IEEE International Conference on Network Protocols (ICNP)*, October 1997.

[17] J. Padhye, V. Firoiu, D. Towsley, and J. Kurose. Modeling TCP throughput: A simple model and its emperical valiations. In *ACM SIGCOMM*, 1998.

[18] J. Padhye, J. Kurose, D. Towsley, and R. Koodli. A model based TCP-friendly rate control protocol. In *IEEE NOSSDAV*, June 1999.

[19] V. Paxson. Automated packet trace analysis of TCP implemenations. In *Proceedings of SIGCOMM*, 1997.

[20] V. Paxson. End-to-end Internet packet dynamics. In *Proceedings of SIGCOMM*, 1997.

[21] Sridhar Ramesh and Injong Rhee. Issues in TCP model-based flow control. Technical Report TR-99-15, North Carolina State University, 1999.

[22] R. Rejaie, M. Handley, and D. Estrin. RAP: An end-to-end rate based congestion control mechanism for real-time streams in the Internet. Technical Report 98–681, University of Southern California, 1998.

[23] R. Rejaie, M. Handley, and D. Estrin. RAP: An end-to-end rate based congestion control mechanism for real-time streams in the Internet. In *IEEE Infocomm*, March 1999.

[24] A. Rowe, K. Patel, B.C. Smith, and K. Liu. Mpeg video in software: Representation, transmission and playback. In *Proceedings of IST/SPIE 1994 International Symposium on Electrical Imaging: Science and Technology, San Jose, CA*, February 1994.

[25] D. Sisalem and H. Schulzrinne. The loss-delay adjustment algorithm: A TCP-friendly adaptation scheme. In *International Workshop on Network and Operating System Support for Digital Audio and Video (NOSSDAV)*, July 1998.

[26] W. Stevens. Request for comments: 2001, January 1997.

[27] W.R. Stevens. *TCP/IP Illustrated, Volume 1*. Addison Wesley, 1994.

[28] W.R. Stevens. *UNIX Network Programming, Volume 1*. Prentice Hall, 2nd edition, 1998.

[29] UCB/LBNL/VINT. Network simulator – ns (version 2), http://www-mash.CS.Berkeley.EDU/ns/.