Time-lined TCP: a Transport Protocol for Delivery of Streaming Media over the Internet

by

Biswaroop Mukherjee

A thesis
presented to the University of Waterloo
in fulfilment of the
thesis requirement for the degree of
Master of Mathematics
in
Computer Science

Waterloo, Ontario, Canada, 2000

Contents

List of Tables

List of Figures

Abstract

This thesis 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 research into the 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 [6] [26] [31] (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 uses window based congestion control of TCP with modifications to support time-lines. TLTCP sends data in a similar fashion to TCP until the deadline for a piece 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 use extensive simulations to examine the behavior of TLTCP and find that under a wide range of network conditions TLTCP flows share bandwidth equally with competing TCP flows and performs time-lined data delivery. 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.

We also describe an API for TLTCP that involves augmenting the `recvmsg` and `sendmsg` socket calls. The API allows the sender to associate data with deadlines and the receiver is not only handed the received data but also informed of the gaps in the data sequence.

Chapter 1

Introduction

1.1 Problem Statement

The goal of this thesis is to create a transport protocol for time-sensitive data streams that compete fairly with existing traffic in the Internet.

1.2 Motivation

The Internet has seen phenomenal growth since its creation. It is used for information, entertainment, business transactions and as means of communication by a large population of users. The World Wide Web (WWW) [3] in particular, has emerged as the single largest application on the Internet [5]. Initially the WWW was based on text and pictures but recently powerful desktop computers, better encoding technologies and faster networks have resulted in the emergence of streaming media as an important component of the WWW as well as the Internet.

1.2.1 Streaming Media

Streaming media is gaining popularity both in terms of the number of users as well as the variety of applications. For instance, streaming media is being deployed on an increasing number of web-pages for enhancing information presentation. Live streaming media is being used for creating collaborative work environments and distance education. Streaming media is also being used to provide audio and visual entertainment over the Internet.

To highlight the increasing popularity of streaming media among the users of the Internet we quote the statistics provided by the vendor of a popular suite of streaming media applications, Real Networks [24]. According to the data released on December 7, 1999, there are approximately 500,000 web-pages, more than 1,750 radio stations and 100 TV stations that provide streaming media content over the Internet. In addition, there are over 92 million unique users that play back this content over the Internet.

It is clear that streaming media has a growing user base and is being used for a variety of purposes; from business presentations to entertainment. Streaming media is a powerful technology and it is expected [12] that in the near future, new and more demanding applications will emerge and usage will increase many-fold. Consequently, we expect that in the future streaming media will become a vital part of the Internet.

Therefore, it is not surprising that streaming media systems have stimulated research as well as commercial interest [24]. There are a number of challenging issues that must be addressed in building such a system. One of the outstanding issues and the motivation for our work is the problem of data transport.

1.2.2 Data Transport

A vital function of data transport protocols that operate over best-effort networks like the Internet is congestion control. It is widely believed [4] [31] [10] 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 [7] which incorporates congestion control [16] [38]. 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. For instance, the popular streaming audio system (Real Audio) which is distributed by Real Networks [24] uses UDP [21].

Since UDP does not implement any form of 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 the existing TCP traffic in the Internet (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 [10] similar to the one that occurred in October, 1986 [16].

On one hand, most systems in the Internet do not provide mechanisms suitable for the delivery of time-sensitive data that a streaming media application can use. On the other hand, there is a valid concern that streaming media applications may perform improper congestion control, if any, and compete unfairly with the existing traffic on the Internet.

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.

1.3 Proposed Approach

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.

Although using TCP will ensure that an application is TCP-friendly, it is unsuitable for streaming media applications because it will potentially send obsolete data that would no longer be useful to the receiving application. TLTCP on the other hand, operates under the assumption that the application is loss-resilient and the data is time sensitive. In accordance with the requirement of media applications TLTCP trades reliability in favor of timely data delivery.

Conceptually the working of TLTCP may be described as follows (a detailed description can be found in Chapter 3). When using TLTCP, in addition to specifying the memory location of the data and its size an application includes a 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. TLTCP uses most of the congestion control mechanisms of TCP with modifications to support data with deadlines. Like TCP, data sends are regulated by a congestion window and ACKs from the receiver. But unlike TCP, as time progresses obsolete data in the window is replaced by current data. TLTCP also regulates its window size and reacts to congestion using mechanisms that are similar to TCP. As a result, under most situations a TLTCP sender sends data in a manner that is similar to a TCP sender.

Associating data with deadlines allows the transport protocol, TLTCP, to deliver time-sensitive data for streaming media applications. 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. Therefore, late data is no more useful than lost data. Conceptually as time progresses, the playback application requires new data as the old data becomes obsolete. In essence, what data is relevant depends upon the progression of time.

Figure 1.1 shows an example of how one might associate deadlines with data in order to create a time-line. Issues related to the implementation and interface are discussed in more detail in Chapter 3 and Section 3.3 respectively.

```
struct deadlines {
   void     *data;
   int       length;
   Deadline  expires;
} time_line[SIZE];
```

Figure 1.1: Time-lined data.

1.4 Contributions

The contributions of the thesis are as follows:

- 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 does not treat all data as a byte-stream. We have created 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. Note that 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.

1.5 Outline

The remainder of this thesis is organized as follows.

In Chapter 2, we present an overview of the issues related to streaming media and emphasize the problem of data transport. In this context we discuss TCP and its reliable data delivery mechanism as well as several schemes proposed in the literature for performing streaming media data transport over the Internet. We further highlight the drawbacks of these proposals and describe how we overcome them in TLTCP.

In Chapter 3 we first explain our design principles and then go on to describe the components of TLTCP in detail. Moreover, we describe how TLTCP operates under different scenarios using illustrative examples. Section 3.3 describes how our protocol would be used in conjunction with streaming media applications and Section 3.3.2 describes the proposed API.

We then use the *ns-2* network simulator [42] to determine the extent to which TLTCP is able to share bandwidth equally with competing TCP flows. The experiments conducted and our results are described in Chapter 4. This is followed by conclusions and a few directions for future work in Chapter 5.

Chapter 2

Background and Related Work

In this chapter, we provide the reader with a brief description of the background assumed in the rest of the thesis and outline some previous work that is relevant to the problem addressed. We begin by outlining the functioning of a typical streaming media system and the challenges involved in implementing one. We then discuss the aspect of a streaming media system that is the subject of our research — transport of continuous media data over Internet. We further elaborate on a crucial aspect of the streaming media transport protocol, end-to-end congestion control, and how it may impact the competing traffic on the Internet. We also describe the concept of TCP-friendliness [17] that attempts to quantify this impact. Window based congestion control has been widely used in TCP for reliable data delivery over the Internet. TLTCP adapts the window based congestion control mechanism to deliver streaming media data by removing its reliability requirement. As part of the background, we outline the working of window-based congestion control in a popular flavor of TCP, TCP-Reno. In Chapter 3 we describe the additional mechanisms that TLTCP incorporates so that window-based congestion control can be used for time-sensitive data transport. We finally describe other proposals for solving the problem of congestion control in streaming media applications and highlight their differences from our approach.

2.1 Streaming Media

Streaming media is the technology that allows a client to commence playback, as data is being delivered to it, before all of the media content has been received. An example architecture for creating, serving and presenting streaming media to the end user is shown in Figure 2.1.

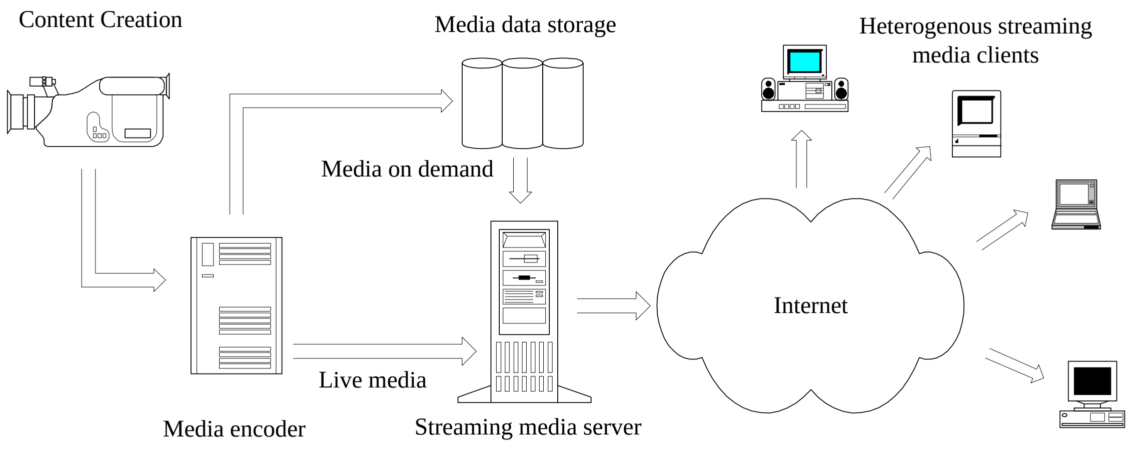


Figure 2.1: Example of a streaming media system.

Content from a microphone (audio samples) or a camera (video frames) [13] is encoded to create a digital representation [23] and since most streaming media systems are bandwidth constrained [12] the encoder typically compresses the data before transmission. The media data upon creation, may be sent directly to the client across the network for live playback (referred to as *live streaming media* [35]) or it may be stored in a media server [13] and be served to the clients upon demand (referred to as *streaming media on* *demand*). Data may be sent individually to all the clients (*unicast*), or it may be sent to an address that represents a group of clients (*multicast*) [35]. Since multicast is still an evolving technology in the Internet, we confine our work to the common case of unicast.

Upon receiving a request from a client, the server calculates a transmission schedule [33]. The *schedule* determines which section of the data (encoded video frames or audio samples) is to be sent at a given point in time, during playback. Note that this schedule may be adapted, according to the changing network conditions, during the playback. In order to deal with unexpected delays in the data delivery, the client buffers some data before commencing playback [13]. The playback application periodically selects the appropriate chunk of data (from the buffer), decodes it and presents the content to the user. Further details of the functioning of the streaming media server and the playback client are presented in Section 3.3. In the next section we outline some fundamental issues that are involved in creating such a system.

2.2 Challenges in Creating a Streaming Media System

There are several challenges involved in creating streaming media systems. For instance, a streaming media server needs to serve data at high rates. Specialized schemes for data storage and retrieval are thus needed to support uninterrupted media playback over fast networks. Several such disk storage schemes are described by Gemmell *et al.* [13]. The need for augmenting an operating systems and its subsystems in order to support data and computation intensive multimedia applications has been discussed by Steinmetz [37]. Operating systems for streaming media applications need process management functions that deal with soft and hard real-time restrictions in addition to resource reservations and guaranteed timing delays for memory access. An overview of these and other issues involved in building a streaming media application can be found in an early paper by Furht

[12]. However, the major issues in creating a transport protocol like TLTCP is designing suitable congestion control mechanisms and if possible exploit the error-resilience and time-sensitive nature of the media data. We elaborate on these two aspects of a streaming media system in the next section.

2.2.1 Continuous Media Data Encoding

Streaming media systems deal with audio and video data, which is often collectively referred to in the literature as continuous media data. *Continuous media* is so named because it is comprised of media sections that are useful to the user only when presented continuously in time [13]. A set of continuously recorded video frames or audio samples may be considered as media *sections* (also referred to in the literature [13] as media *quanta*). In order to be useful, each section needs to be presented to the user at a particular time relative to the start of the playback. We refer to such data as *time-lined* data because the presentation of sections follows a continuous time-line. Our proposed transport protocol is designed with this time-lined behavior of continuous media data in mind and hence the name, Time-lined TCP (TLTCP).

Digitization and encoding of continuous media content is an important aspect of streaming media. An encoding scheme should be designed to minimize the bandwidth required while the required codecs should be computationally inexpensive. Several popular encoding schemes for audio and video have been standardized, e.g., MPEG 1, 2, 4 and 7 [32], Motion JPEG, H.261 [1] and Dolby AC-3. Moreover, in order to utilize the present best-effort infrastructure of the Internet the encoded data should be error-resilient so that media applications are able to cope with the delays and losses in transmission over the Internet. Since the arrival times of various sections cannot be predicted with accuracy, it is important that playback continue (at a lower quality) even if some of the data is lost or late. This property of continuous media data is referred to as *error-resilience* or *loss-tolerance*.

For instance, an encoding technique designed for scalable transmission over the Internet is layered encoding [20], where a media stream consists of a base layer and several enhancement layers. To perform playback a client must receive at least the base layer and to progressively improve the quality of the media presentation it may scale up the number of enhancement layers it receives. The base-layer can be served uniformly to a set of heterogeneous clients and depending upon the network bandwidth available to a particular client or its decoding capability, it receives data from the enhancement layers to enhance the quality of its presentation. Examples of layered encoding schemes for audio and video are discussed by McCanne [20] and Turletti [41].

TLTCP exploits the property of error-resilience, since it does not guarantee reliable data delivery. It also attempts to adhere to the time-lined nature of the continuous media data by keeping track of and discarding obsolete data. All the encoding schemes described above are error-resilient and time-lined and can therefore be used in conjunction with TLTCP in a streaming media system. For instance consider an encoding scheme like MPEG1 video that has frames with different degrees of importance. When handing frames to the TLTCP sender for delivery, the frames are written in the order of priority with appropriate deadlines according to the transmission schedule. TLTCP then starts sending the most important frame for the earliest deadline and goes on to the less important frames, if the deadline has not expired. But if the deadline does expire before it can send the less important frames, TLTCP discards them and starts sending the most important frames associated with the earliest deadline that has not expired. In Section 3.3 we describe in detail how TLTCP delivers MPEG video and layered media data.

2.2.2 Transport Protocol

The component of a streaming media system that this work investigates is the data transport protocol. The problems that need to be addressed are as follows:

- The proposed protocol must be able to perform unreliable but timely delivery of media data from the server to the playback client.
- The congestion control mechanisms for the proposed protocol must be robust over a wide range of network conditions.
- The data flows should compete fairly with the existing traffic on the Internet.

The temporal characteristics of continuous media are such that in order to be useful for playback at the client data delivery may be unreliable but must take into account the time-lines specific to the application. This is the guiding principle for the time-lined delivery mechanisms of TLTCP, which are described in Chapter 3. Another crucial aspect of the data transport protocol is congestion control. In a network like the Internet, where no traffic policing is done, improper congestion control mechanisms may endanger the stability of the network itself [10] and in the worst case *congestion collapse* may occur. Furthermore, the media traffic generated should compete fairly with the existing traffic in the Internet. In a network like the Internet, where congestion control is performed end-to-end, the way a stream competes with others is determined by its behavior under congestion. The reason for this is as follows. All flows are expected to react to congestion by appropriately reducing their data rates. If a flow competes unfairly it may not reduce its data rate and as a result may obtain a greater share of the bottleneck bandwidth. TLTCP's congestion control mechanisms are robust and ensure that it competes fairly with the existing traffic on the Internet.

In the next section we examine the issues related to data transport over the Internet in greater detail. We begin by describing the congestion control principles of TCP and highlight the merits that prompt us to base the congestion control mechanisms of TLTCP on TCP. We then compare our approach with several other congestion control mechanisms proposed in the literature for streaming media systems.

2.3 Congestion Control in the Internet

2.3.1 TCP

TCP is a protocol that guarantees reliable data delivery over the Internet. It is also the most popular protocol used in the Internet today (90-95%)[5]. TCP was designed with the best-effort service offered by the IP layer in mind and performs end-to-end congestion control. In practice the congestion control mechanism of TCP has been widely used over the Internet and is regarded as the single most important reason for the stability of the Internet [4]. Our proposed protocol TLTCP, is based on TCP and uses the same mechanisms to discover bandwidth, send data in the steady state and react to congestion as TCP. As described by Jacobson *et al.* [16], not only does TCP use sound congestion control mechanisms but it also achieves excellent throughput. Hence, it is expected that TLTCP will also achieve good throughput and be robust under congestion. In Chapter 4 we show by means of simulation that TLTCP does indeed achieve throughput similar to competing TCP streams and performs appropriate congestion control. In addition, as a result of inheriting most of the data sending functionality of TCP, TLTCP's data sending characteristics are very similar to that of TCP. In other words, a TLTCP sender sends packets at similar times as a TCP sender would if it were operating under the same conditions. As a result, TLTCP interacts with competing TCP streams and reacts to congestion in a fashion that is similar to another TCP stream as just another TCP stream and reacts to congestion in a similar fashion. Therefore, it competes with the TCP traffic in a TCP-friendly fashion. We elaborate on TCP-friendliness in the next section. The following are the salient features of the congestion control mechanisms of TCP, that TLTCP adopts.

TCP's congestion control behavior is dictated primarily by how the size of its sliding window is controlled. The sender's window keeps track of all the unacknowledged data and is an efficient way of ensuring reliability. It ensures reliability, because the sending window keeps track of all the unacknowledged packets and retransmits the packet if it detects a loss; it is efficient because it allows a full window's worth of data to be in transit, thereby utilizing all the bandwidth available. Even though the sliding window is vital for ensuring reliability in TCP, more importantly for the purpose of this thesis, it can be used as a tool for congestion control.

The *size* of the sender's window represents the amount of data in transit. A sender that is maintaining a full window of packets releases a packet into the network only when it receives an acknowledgement for the receipt of a packet within the window, because the ACK signals the removal of a packet from the network. This rule, referred to as the principle conservation of packets [16] attempts to ensure that the number of outstanding packets in transit is kept constant. Using this principle, a robust and self-regulating congestion control mechanism is created by maintaining a window size that is approximately equal to the capacity of the network in terms of the amount of data in transit. This is the underlying principle of TCP's (and TLTCP 's) congestion control mechanism. In TLTCP data reliability is not essential as it is meant to serve loss-tolerant media applications. Upon determining that some data in the (logical) sending window is obsolete, the sender may replace it by newer data without altering the size of the (logical) window. In this way TLTCP is able to adhere to the principle of conservation of packets but it is still able to send time-sensitive error-tolerant data.

Other mechanisms that TLTCP adopts from TCP include slow-start, additive-increase multiplicative-decrease of the window size and fast-retransmit and fast-recovery. TCP's slow start mechanism allows it to begin utilizing all the available bandwidth quickly, by doubling its window size every round-trip time, thus providing good throughput, without a large wait while discovering bandwidth. After discovering the available bandwidth, TCP attempts to avoid congestion by additively increasing its congestion window when there are no packet losses, and by multiplicatively reducing its congestion window when a packet loss is detected. Fast-retransmit and fast-recovery allow TCP to recover from single packet losses by waiting for three duplicate ACKs, retransmitting the lost packet (fast-retransmit) and then resuming normal sends with half the window size (fast-recovery). Details of TCP's mechanisms can be found in the paper by Jacobson *et al.* [16] and a book by Stevens [39].

2.3.2 TCP-friendliness

In the previous section we saw the principles of congestion control that are employed by TCP streams. While the careful design affords TCP the stability and flexibility to effectively use the Internet, it leaves TCP vulnerable to bandwidth stealing by UDP based flows or even flows resulting from improper TCP implementations. To date, popular applications in the Internet like web browsers and file transfer programs use TCP flows to transfer data. But as pointed out in the previous chapter new applications that are gaining popularity, such as streaming media, use UDP. Potentially such applications can act in a *TCP-unfriendly* manner by using less stringent methods for congestion control, thereby unfairly acquiring more bandwidth than competing TCP flows.

There is a concern in the research community that several applications that use UDP for transferring large amounts of data across the Internet do not use appropriate mechanisms for congestion control [4]. In view of the damage that this can cause to both the Internet and the applications that use the Internet, there has been a concerted effort [17] in the research community to promote the use of *TCP-friendly* mechanisms for applications that use UDP. These applications are expected to perform appropriate congestion control and compete with TCP in a fair manner.

The basic idea behind TCP-friendliness is that if two flows are operating along the same path, a non-TCP flow should not adversely impact a competing TCP flow any more than another competing TCP flow would [17]. In several studies [6] [31] [26] this has been interpreted and measured by the ability of non TCP-based applications to equally share bandwidth with TCP-based applications. This is typically measured by observing the throughput obtained by several flows (both TCP and non-TCP) simultaneously operating over the same bottleneck, under the same conditions, and determining the bandwidth shares of the flows of the different protocols.

2.3.3 Congestion Control for Streaming Media

In the Section 2.3.1, we saw the congestion control mechanism of TCP. We now turn our attention to the proposals for congestion control in streaming media applications that do not require data reliability and therefore typically use UDP. A streaming media system that uses UDP is at liberty to choose any mechanism for congestion control that it desires (including none). As described in the previous section, in a best-effort shared network like the Internet, the congestion control mechanisms of such a stream would impact its fairness when competing with other streams. Therefore, it is imperative for UDP-based streams to function in a TCP-friendly manner. Previous work has recognized the importance of and created several schemes for, performing TCP-friendly congestion control for such UDP streams.

Previous work [6] [36] [26] [31] 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 in an attempt to achieve the same throughput as a TCP stream would, if it was operating under the same conditions.

These approaches are based on models that attempt to characterize TCP's congestion control mechanisms [17] [19] [25]. Models with varying degrees of complexity and accuracy have been proposed, from a simple steady state model of TCP [17] to models that take into account correlated losses and timeouts [25]. These models are represented as equations and are used at the sender to control the sending rate. As data is sent, the application measures or estimates values for the various parameters required by the model based on receiver reports such as packet loss rates, round-trip times and timeout values. Using these parameters and the model the sender periodically recomputes the appropriate sending rate. In general, if there are no packet losses the sending rates are increased. Otherwise, depending upon the loss rates the equation is used to recalculate a lower sending rate. The time between two recomputations is referred to as the *recomputation interval*.

In the following we discuss several drawbacks inherent in rate-based approaches that may reduce their practical significance.

- A fundamental problem in trying to model TCP is that its behavior differs significantly under different conditions. As a result an equation that accurately characterizes TCP under one set of conditions may fail under others. For instance, as pointed out by Padhye *et al.* [25], the model for TCP that they propose is not able to characterize TCP under certain conditions (e.g., TCP over modem connections with large dedicated buffers). Even if the rate equation is accurate for the particular operating conditions, rate-based schemes have several critical parameters that need tuning for different scenarios. For instance, the value of the recomputation interval needs to be tuned carefully. If the sending rate is recalculated too frequently, the loss rates observed between the short time intervals may not be meaningful. However, if the sending rate is not recalculated frequently enough the flow will not be responsive to changes in network conditions [26]. This means that parameters need to be tuned not only for every new connection setup, but potentially the parameters may also need to be adapted in each connection as the network conditions change during playback. The later may introduce further errors in the models because of lagged response. To date, this issue has been observed by previous work but no satisfactory solution has been proposed. In most cases the authors have tuned the parameters statically [26] for every scenario. We expect this to be a major problem in the practical use of TCP models.

- Because TCP is an ACK-clocked protocol we believe that it will be difficult for a rate-based scheme to mimic it. Indeed, as pointed out by Rejaie *et al.* [31] there seems to be a consensus in the Internet research community about this. This is because the robust conservation of packets mechanism cannot be easily recreated and used without a sending window. In the rate-based mechanisms a sending rate is calculated every recomputation interval without keeping track of the amount of data in transit (as there is no sending window). This may lead to oscillations in the case of rapidly varying network conditions. In TCP and TLTCP on the other hand, every packet sent is ACK-clocked and as a result the number of packets in transit is maintained close to the data capacity of flow path. For instance, if the bandwidth available to a flow decreases, new ACKs arrive at the sender less frequently thereby automatically adapting the data rate to the available bandwidth.

- Future changes to TCP or its operating environments may require reworking and reevaluating new models and resulting protocols. On the other hand, if there is a change in the popular version of TCP used in the Internet, we expect to be able to add the new functionality into TLTCP while retaining its mechanisms to keep track of data deadlines, thereby creating a version of TLTCP which is similar to and friendly with the new version of TCP.

- Rate-based protocols rely on the accuracy of operating system timing functions in order to accurately measure the parameters required by the model, such as data rate and loss rate. A rate-based protocol also relies on an accurate timer to inject packets into the network (since it does not use ACK-clocking) and to schedule re-computations at time intervals specified by the model. Most rate-based proposals use UDP and are implemented at the user level. As a result of the inaccuracy of timing functions at the user level such mechanisms will be disadvantaged. For instance consider the common case of a sender attempting to measure round-trip times at the user level by measuring the time between sending a message and receiving an acknowledgement for it. The measured time will include the time the message and its acknowledgement has spent in the kernel buffers, the overhead of system calls and the inaccuracy in the timing function (e.g., `gettimeofday()`). Even if we assume that the error introduced will be small, it has been shown that small errors can have a significant impact on rate-based congestion control because of their heavy dependence on accurate timing. Ramesh *et al.* [29] point out several factors that can result in inaccurate packet loss estimates in the model developed by Padhye *et al.* [25]. They further show that these inaccurate estimates can lead to under or over-allocation of bandwidth to non TCP flows.

Given the popularity of streaming media and the significance of its possible impact on the Internet, it is not surprising that several congestion control mechanisms have been proposed for them. In the following we first list the proposed IETF (Internet Engineering Task Force) standards that relate to streaming media. We then discuss the pros and cons of various other congestion control proposals for streaming media in the Internet.

The IETF has proposed protocols for real-time transport over the Internet — RTP/RTCP [34]. These protocols specify packet header fields including sending sequence numbers, timestamps and receiver reports that are meant for use by a rate-based congestion control mechanism at the application level. One such application level mechanism specifically designed to use the RTP/ RTCP reports is proposed by Sisalem *et al.* [36]. However as pointed out before, applications are not in the best position to perform congestion control and may use improper mechanisms, if any. TLTCP on the other hand proposes incorporating robust congestion control mechanisms at the transport level in the kernel, thereby freeing the application from that responsibility. The IETF has also proposed an application level protocol that is meant to be used by streaming media applications, called RTSP [35]. RTSP is designed to fulfill the control part of the client-server interaction such as, synchronization, connection setup and tear-down, and several remote-control like features, like pause and play.

Sisalem *et al.* [36] propose a rate-based mechanism for congestion control that is meant to be implemented by applications that use the RTP/ RTCP [34] specifications on top of UDP. Their scheme dynamically computes an additive increase rate and also performs backoff by multiplicatively reducing the data rate. Experiments conducted with RED gateways are reported and show that their scheme does not share bandwidth equally under situations with low loss rates. Our simulation experiments show that TLTCP is stable over a wide range of network conditions and shares bandwidth equitably under conditions of low loss rates.

RAP [31] is a rate-based protocol that employs a relatively simple additive-increase multiplicative-decrease (AIMD) model of TCP's congestion control mechanism and is able to obtain relatively TCP-friendly behavior when competing for bandwidth with TCP Sack flows [8]. While it is friendly when competing with TCP Sack flows over RED [11] switches, it is not able to share bandwidth fairly with the popular implementations of TCP in the Internet today [8], TCP Tahoe or TCP Reno [30]. A significant advantage of TLTCP is that it is based on, and therefore competes fairly with TCP Reno, which is the most widely used TCP implementation in the Internet today [27] [28]. Furthermore, as explained previously, it will be fairly simple to adapt TLTCP to the new flavors of TCP if the need arises as a result of the new flavors gaining popularity.

Padhye *et al.* [26] describe and evaluate a rate-control protocol based on a more detailed model of TCP throughput [25] than the proposals above. Although the simulation results reported show that their protocol is TCP-friendly under a variety of network conditions, the recomputation interval is chosen using a different method for the different experiments shown. Therefore, the method to calculate the best recomputation interval may vary across different network conditions. No algorithm for choosing the appropriate method automatically is mentioned. This may limit the benefits of this scheme when used in practice. Padhye *et al.* also point out that the proposed scheme does not ensure equitable sharing of bandwidth with TCP streams when bottleneck link delays are too small or too large since it makes accurately estimating loss rates difficult. On the other hand, our simulation results show that TLTCP is friendly across a wide range of bottleneck link delays.

Cen *et al.* [6] describe a streaming control protocol (SCP), that uses a congestion window along with rate equations for sending data. While their approach is similar to ours in many respects, they are not faithful to TCP in order to improve smoothness in streaming. SCP does not have time-lined data delivery mechanisms and as a result just relies on a no-retransmission policy to avoid delays. The experimental results reported using an implementation of SCP on top of UDP show that the packet rates of TCP flows competing with SCP flows are significantly lower under a variety of network configurations, indicating TCP-unfriendly behavior.

Another scheme reported by Jacobs *et al.* [15] 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 mimick the congestion window of TCP at the user level is likely to be inaccurate. As explained before, 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. Further details of the scheme are not provided and it is unclear how TCP-friendly such an approach would be.

In light of the drawbacks of the rate-based approaches and the advantages of the window-based congestion control enumerated above, we decided to create a window-based mechanism that allows for time-lined delivery of streaming media data over the Internet, but does not guarantee data reliability. Our goal is to retain the robust congestion control mechanisms of TCP and add time-lined data delivery mechanisms to it. This approach is suitable for providing service to streaming media applications that do not require reliability and are unwilling to tolerate unpredictable delays in data delivery and thus presently resort to rate-based approaches over UDP. When designing a streaming media system it is important to consider the pros and cons of all the proposals. We like to view the approaches in this area as a part of a spectrum of TCP-friendliness, which we describe next.

2.3.4 Spectrum of TCP-friendliness

One view of TCP-friendliness is that there is a spectrum of possible approaches to being TCP-friendly. At one extreme one might use aggressive UDP applications that completely ignore congestion in order to obtain larger portions of the available bandwidth. Clearly such an approach is not TCP-friendly and its widespread use could have serious consequences. TCP is located at the other end of the spectrum. Although it may not share bandwidth fairly under a variety of conditions (e.g., when flows have different round-trip times [14] or when there are a large number of flows [22]), TCP can be viewed as the reference point for (or definition of) TCP-friendliness. Streaming media applications that are implemented using TCP will be TCP-friendly (by definition) but their performance may suffer because TCP's data delivery is driven by the reliability requirement.

Between these extremes lie a number of different approaches which vary in their ability to be TCP-friendly. Techniques that are less TCP-friendly are likely to provide benefits to the application because they will be *stealing* bandwidth from other TCP applications. We believe that TLTCP provides an important point on this spectrum in that it is very close to TCP in terms of friendliness but offers significant performance advantages to streaming media applications when compared with TCP.

In the next chapter, we describe the principles behind the design and the operating of the proposed protocol, TLTCP.

Chapter 3

Time-lined TCP

In this chapter we describe the functioning of the proposed protocol, Time-lined TCP (TLTCP) and how it supports time-sensitive applications. We start by listing our design principles and discuss how these principles have influenced our approach. We then describe the details of the time-lined data delivery mechanisms of TLTCP and its operation. We also describe how TLTCP deals with lost data and its impact on its TCP-friendliness. We argue that TLTCP's approach of associating data with deadlines is natural to continuous media data. We then describe in brief, the working of a streaming media application that uses TLTCP. Finally we present augmentations to the present socket API, that we propose for use with TLTCP and streaming media applications.

3.1 Design Principles

3.1.1 Congestion Control

TLTCP is meant to function over the Internet, which is a network that depends upon end-to-end congestion control. As mentioned in Chapter 2, the absence of suitable mechanisms for congestion control may have a negative effect on existing traffic and in the worst case may cause congestion collapse. Hence a key design principle for the proposed protocol is that the congestion control mechanisms must ensure fairness and network stability, two of our primary goals.

For the purpose of congestion control, TLTCP uses the window-based, ACK-clocked mechanism of TCP, because it is robust, adapts quickly and competes fairly with TCP flows.

A window-based protocol is robust, as described in Chapter 1, because in its steady state it attempts to maintain a constant number of packets in transit, where the number of packets is an estimate of the capacity of the network. ACK-clocked data sending is a feedback scheme that adapts quickly with the traffic changes in the round-trip path. For instance when the bandwidth available to an ACK-clocked stream increases the data packets and ACKs can traverse the path quickly, resulting in a faster arrival rate for the ACKs and implicitly causing a faster rate of packet sending. Similarly, if there is a reduction in the bandwidth share available to the stream due to increased traffic at the bottleneck, the rate of sending data slows down quickly in response to a slowdown in the rate of arrival of ACKs. Thus, ACK-clocked data sending is an appropriate feedback mechanism for the Internet. Therefore, TLTCP uses a congestion window with ACK-clocking to regulate its data sends but augments its operation to make it suitable for time-lined data delivery.

3.1.2 Fairness

The Internet is a shared network where the users are expected to coexist with each other. It is possible for a flow that sends packets more aggressively than the others, to obtain more than its fair share of the bandwidth.

Most of the traffic in the Internet is due to TCP flows (90-95%) [7]. These flows in their steady state increase their data rate additively in the absence of packet losses and back off exponentially in the event of congestion. If a new protocol such as a streaming media transport protocol is introduced in the Internet that increases its sending rate more aggressively than TCP and does not reduces its data rate during congestion as quickly, it will obtain more bandwidth than competing TCP flows and is regarded as TCP-unfriendly. This is clearly an undesirable situation from the perspective of fairness. Thus, an important design consideration when creating a new protocol for the Internet is fairness when competing with TCP traffic. This property of a protocol is often called TCP-friendliness.

TLTCP retains TCP's congestion control mechanisms; it increases its sending rate and responds to congestion in the same way as TCP. Additionally, the mechanisms to support time-lines are designed such that, when possible, a TLTCP sender would send data in the same fashion as a TCP sender operating under similar conditions. As a result of these measures, TLTCP flows are TCP-friendly over a wide range of conditions and can be expected to compete fairly with existing traffic on the Internet.

3.1.3 Time-sensitive Data Delivery

The proposed protocol is designed to deliver time-sensitive and error-resilient data. This implies that not all of the data needs to be sent and the data that is sent must arrive in time to be useful for playback at the receiver.

With these requirements in mind we design TLTCP by excluding the strict reliability requirement of TCP. In TLTCP each section of time-sensitive data is associated with a deadline. TLTCP sends a section of data and performs retransmissions for it as TCP would but only until the deadline associated with that section has expired. In order to keep track of deadlines associated with data sections a timer is added to TLTCP to keep track of the current section. When the unsent data becomes obsolete due a deadline expiry, TLTCP replaces it with new data. Another way to view this mechanism is in terms of the progression of the sliding window. In addition to the ACK-based progression of the sliding window that is present in TCP the deadline expiry of a section of the data in TLTCP causes the window to move beyond the obsolete data, even if that data has not been sent or acknowledged.

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. Under a receiver relative scheme the TLTCP transport layer would use round-trip time (RTT) estimates in order to predict whether or not the current section of data to be sent would arrive at the receiver prior to the deadline. It would then discard data that would be unlikely to be delivered prior to the deadline.

3.2 Functioning of TLTCP

In this section we describe how TLTCP operates, highlighting the mechanisms that support the delivery of time-sensitive data. To clarify how TLTCP the function under different scenarios we use illustrative examples. 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 (for more information see Section 2.3.1) and we use TCP to refer to TCP-Reno.

3.2.1 The Sender

The TLTCP sender accepts time-sensitive data from the application via the TLTCP API (described in Section 3.3.2). 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. Figure 3.1 shows 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.

```
struct time_lined_data {
   int lowest_seqno; // lowest seq # for this section
   time deadline;    // deadline for this section

   time_lined_data *next;
};
```

Figure 3.1: The structure of a node in the linked list of data deadlines.

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.

3.2.2 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 3.2 shows the sequence of actions that are taken after expiry of the lifetime timer.

All of the data that is handed to a TLTCP sender is considered to be a continuous sequence of bytes divided into data sections with deadlines associated with each of them. 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. We call these discontinuities in the delivered sequences, *gaps*.

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 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

```
if ( Lifetime_timer has EXPIRED ) {
      // Remove obsolete data from the buffers and from
      // the timeline list
      remove_expired_data(timeline_list, &buf);
      if (!timeline_list_empty()) {
              // Get lowest seq of unexpired
              // data and move the window accordingly
              current_node = get_current_node(timeline_list);
              store_unacked_seqno();
              move_window(current_node.lowest_seqn);

              // Set the lifetime timer for the deadline
              // of the new data
              set_lifetime_timer(current_node.deadline);
      }
}
```

Figure 3.2: Pseudo code of the actions taken on the expiry of lifetime timer.

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 3.2.4).

3.2.3 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. Note that the sequence numbers that a receiver is willing to accept is determined by the next expected sequence number and the size of the 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 (this is explained in more detail in Section 3.3.2).

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`.

3.2.4 ACKs for Obsolete Data

Besides accepting time-lined data from the sending application the sender needs to keep track of acknowledgments for obsolete data. This allows TLTCP to ensure that the sender's send window is correctly sized and is permitted to advance as ACKs arrive for the obsolete data.

Reconsider the example described in Section 3.2.2, 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 never resends obsolete data. When there is a loss detected in obsolete data the TLTCP sender a sends a packet containing current data, instead of resending the lost data that is now obsolete. In the next section we describe in detail some packet loss scenarios and how TLTCP deals with them.

3.2.5 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 3.3. 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 .

However, as shown in Figure 3.4, 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

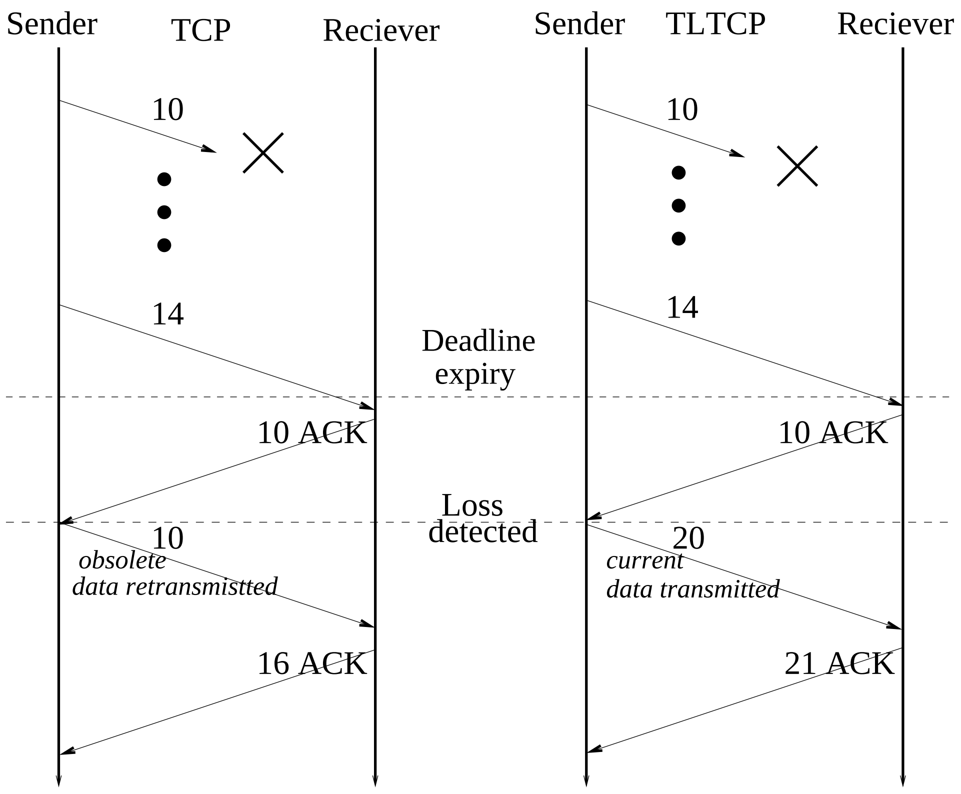


Figure 3.3: Example of a loss in obsolete data.

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.

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. A simplified depiction of such a scenario is shown in Figure 3.5 and described below.

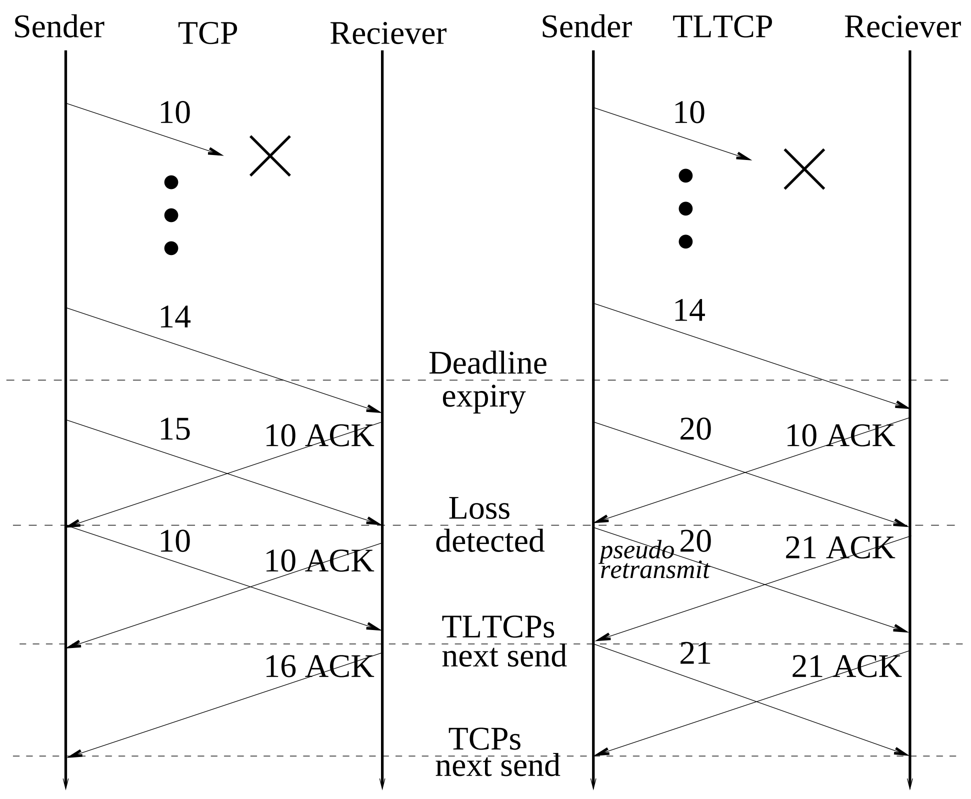


Figure 3.4: Example of a pseudo-retransmission.

Suppose in the original example of Section 3.2.2, 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` [38] 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 `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

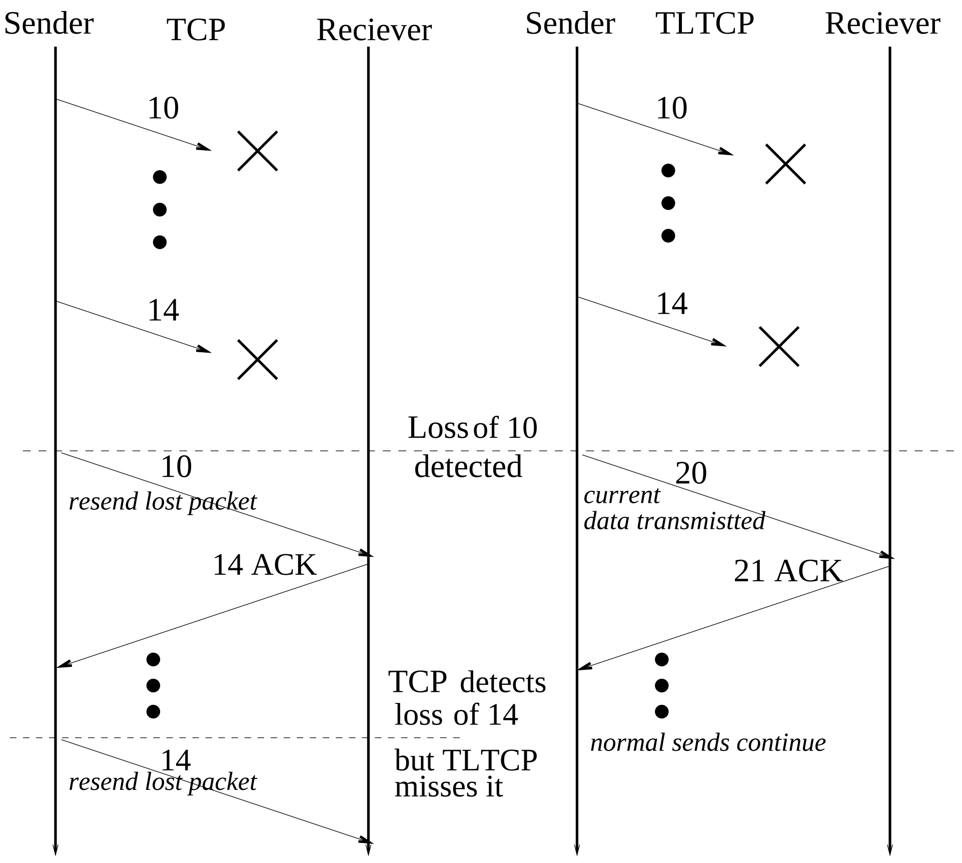


Figure 3.5: Example of a TLTCP missing a packet loss in the obsolete data.

second slowdown.

In our simulation experiments reported in Chapter 4 we test TLTCP under a variety of packet loss conditions and describe how the scenarios described here affect the behavior of TLTCP flows.

3.3 Applications

3.3.1 Streaming Media Using TLTCP

A class of applications that is gaining popularity on the Internet and thus significance are streaming media applications. TLTCP supports the delivery of time-sensitive data and can thus support such applications.

In Chapter 1 we have described the overall architecture of a streaming media system and the concept of time-lined data delivery, upon which TLTCP is based. In the context of this chapter the components of interest are the server and the client that are connected by the proposed transport protocol, TLTCP. In a simple scenario, upon receiving a request for a particular continuous media file from the client, the server calculates a transmission schedule depending upon the temporal distribution of media sections, the network bandwidth available and the buffering at the client. Since the network conditions and the amount of data buffered at the client may change over time, feedback from the client is used to adaptively change the transmission schedule. Note that feedback from the client is not used for congestion control as in the UDP-based approaches described in Chapter 2 but is used at the application level and thus does not need to be as frequent or precise. The application can rely on TLTCP to manage all the data transport functionality, including congestion control. TLTCP is suited for continuous media data because instead of treating all data as a byte stream, it exploits the temporal nature of continuous media data by distinguishing between individual sections of data and associating deadlines with them.

Recall that the server typically starts by creating a TLTCP data connection to the playback client and then calculates a schedule for the transmission of data. Each section of data to be sent (e.g., sequence of video frames, layers of video, or audio samples) are assigned a deadline that is determined by the schedule. Then the application passes a few sections of media data to the TLTCP sender, adapts the transmission schedule according to the feedback from receiver if necessary and then write the next few sections according to the new schedule. The TLTCP sender would use the deadlines specified for the given sections to send as much of a section as it can before the deadline expiry and then move on to the next section.

As pointed out before, the sending application receives periodic feedback from the playback application about the state of the playback, amount of data buffered and the network throughput and uses it to adapt the transmission schedule. For instance if the effective data rate reduces during the playback because of increased contention in the network, the server may modify the transmission schedule to send every alternate media section so that each of the media sections get more time to get delivered. TLTCP on its part, only attempts to deliver the sections of data that are not obsolete.

The client would begin playback after first receiving and buffering a few of the initial sections of the data. During playback appropriate sections of data are read from the buffer, decoded and presented to the user. If the sender is not able to send all the data in a section before the deadline associated with the segment expires, the receiver will continue with a lower quality playback. The quality of the playback depends upon the application's ability to tolerate lost data.

As an example, consider a hierarchical layered encoding scheme consisting of a base layer and several enhancement layers (for a more detailed description of layering see Chapter 2). A layered media stream can be represented as,

$$\{L_1^1, L_2^1, L_3^1, L_4^1, L_1^2, L_2^2, L_3^2, L_4^2, \ldots\}$$

where for *section* i, L_1^i is the base layer and L_2^i, L_3^i and L_4^i are the enhancement layers. The sending side of the application computes a schedule and associates a deadline, d^i, with each section ($\{L_1^i, L_2^i, L_3^i, L_4^i : d^i\}$). The sender writes several sections (e.g., $i, i+1, i+2, \ldots$) along with their corresponding deadlines, ($d^i, d^{i+1}, d^{i+2}, \ldots$) to TLTCP. TLTCP attempts to deliver all the data in section i in the order written (i.e, $\{L_1^i, L_2^i, L_3^i, L_4^i\}$). Once the deadline for section i expires the data is obsolete and is discarded. TLTCP then continues by attempting to deliver section $i+1$. What is presented to the user at the receiving end (i.e., the quality of the playback) will depend on what portion of section i $\{L_1^i, L_2^i, L_3^i, L_4^i\}$ arrived prior to the playback time of section i, the encoding scheme and the application's ability to tolerate loss.

Similarly, MPEG-1 video [32] has frames with varying degrees of importance for the playback application. The frames are of three types I, P and B in the respective order of importance. 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. Figure 3.6 shows an example of the dependencies.

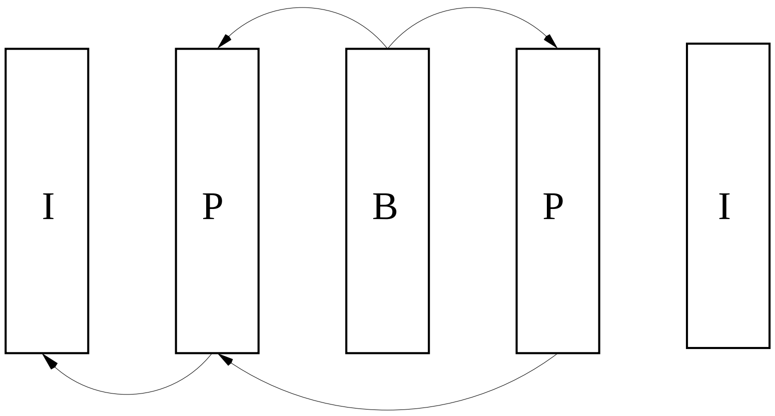


Figure 3.6: MPEG-1 frame dependencies.

Because of the bidirectional dependencies, the display order of frames differs from the order in which it is stored in a file or transported. For instance the display order of an MPEG-1 video may be. $\{I^1, P_1^1, B^1, P_2^1, I^2, P^2, B^2, I^3, \ldots\}$. However, the order in which this sequence is stored in an MPEG file will be $\{I^1, P_1^1, P_2^1, I^2, B^1, P^2, I^3, B^2, \ldots\}$.

TLTCP sections are created form an MPEG-1 file in the same order as it is stored, but the deadlines are assigned according to the order of display. Hence 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 of frames it is dependent upon, but sent after the later of the frames it is dependent upon. In this manner the order in which TLTCP is handed the frames is the same as it is stored but the deadlines depend upon the order in which the frames are to be displayed. Thus in the example above the deadline assignments would be as follows.

$$\{\{I^1, P_1^1, P_2^1 : d^1\}, \{I^2 : d^2\}, \{B^1 : d^1\}, \{P^2 : d^2\}, \{I^3 : d^3\}, \{B^2 : d^2\}, \ldots\}$$

The sending application can start by handing all the encoded frames in the way described above. TLTCP would try to deliver the sections in the order written. If however the available bandwidth is not sufficient to deliver all of $\{I^1, P^1, P_2^1\}$ TLTCP will discard P_2^1 at the expiry of d^1 and start sending the more important frame, I^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 more important data that still has a chance of reaching the receiver 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 above, the data sections handed to TLTCP in the reduced bandwidth cases would look like $\{\{I^1, P_1^1, P_2^1 : d^1\}, \{I^2 : d^2\}, \{P^2 : d^2\}, \{I^3 : d^3\}, \ldots\}$ and $\{\{I^1 : d^1\}, \{I^2 : d^2\}, \{I^3 : d^3\}, \ldots\}$ 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.

3.3.2 The TLTCP API

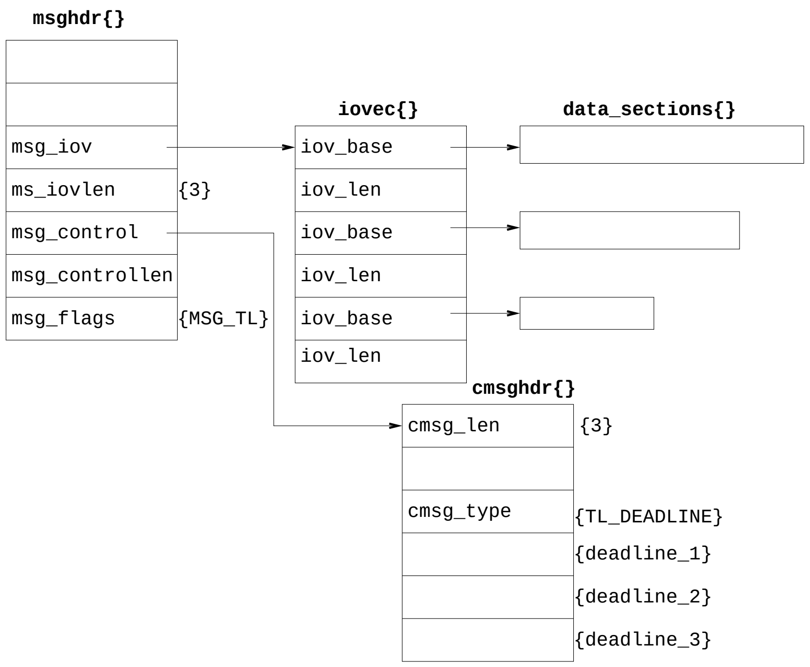
The API for TLTCP has two main functions. First, the sending application needs to be able to specify to TLTCP segments of data along with their associated deadlines. Second, the receiving end needs to be able to deliver to the client application the received data along with information about where gaps are located.

We propose augmenting the UNIX socket calls of `recvmsg` and `sendmsg` [40] for this purpose. A time-line data send is flagged by a new flag, `MSG_TL`. The scatter gather I/O vector `msg_iov` is used to indicate the data to be sent and the associated deadlines are provided in an ancillary data message of type `TL_DEADLINE`. Similarly, on the receiver side the delivery of time-lined data would be indicated by the flag, `MSG_TL`. The received data is placed by TLTCP into the `msg_iov` and ancillary data is used distinguish the data `TL_DATA` from the gaps `TL_GAP`. Note that these changes allow the socket calls to retain their semantics while incorporating the new functionality of TLTCP.

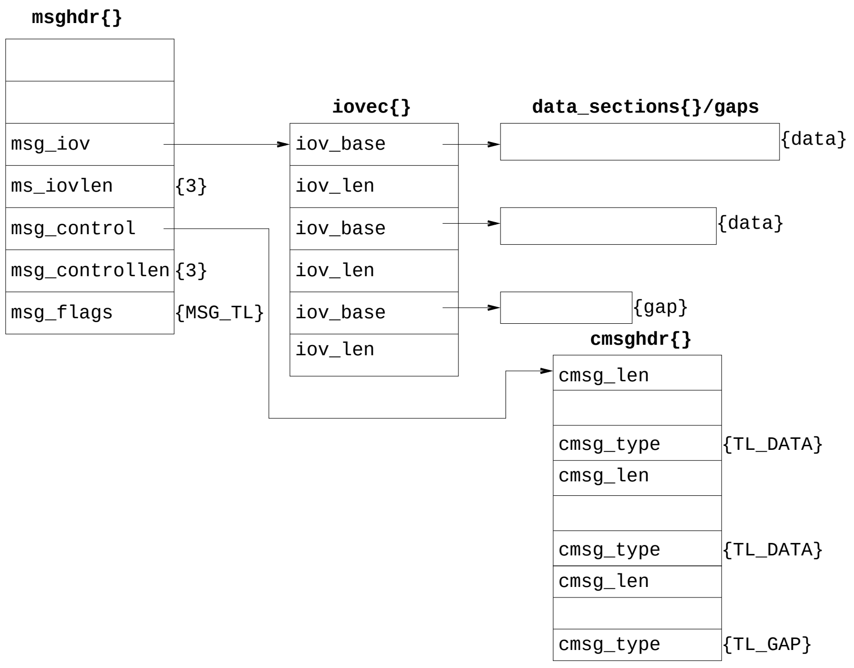
To see a more detailed example of how the API would be used consider the following example. The server process first creates a `SOCK_STREAM` socket and connects it to the receiver to establish the data connection. Then the various fields of the `msg_header` structure are filled in before calling `sendmsg` with a `MSG_TL` flag used to indicate time-lined data. Figure 3.7(a) depicts the fields of the `msg_header` data structure that is passed as an argument to the `sendmsg` call. 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. Figure 3.7(b) depicts the `msg_header` structure after the call to `recvmsg` has completed. 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.

In this chapter we have described how TLTCP operates and how applications would use it. In the next chapter we study the characteristics of TLTCP under a variety of simulated network conditions and see how the mechanisms described in this chapter shape TLTCP's behavior.



(a) sendmsg



(b) recvmsg

Figure 3.7: The `msg_header` structure at the sender and the receiver.

Chapter 4

Simulations

In this chapter we evaluate the behavior of TLTCP using simulations. In particular, according to our goal, we intend investigate if TLTCP is able to perform time-lined data sends and measure its TCP-friendliness. There are several reasons why simulation experiments are more suitable than live Internet experiments for our purposes.

- One of our primary goals was to create a transport protocol that shares bandwidth equitably with competing TCP traffic. In order to quantify TLTCP's performance on this count 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 add significant noise to the experimental results. On the other hand, with simulations impact due to the *other factors* can be eliminated or factored into the results. Furthermore, we need to compare the impact of TLTCP streams on competing TCP streams, with a *baseline* (control) case where only of TCP streams compete against each other. For the measurements obtained in the baseline case 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 [42] and have conducted several experiments. The goal of the experiments is to study TLTCP's time-lined data transport behavior and to quantify its TCP-friendliness. Our experimental results show that TLTCP is a protocol that performs time-lined data delivery in a robust, TCP-friendly fashion over a wide range of simulated Internet conditions, thereby satisfying our goal. In the following sections we first show that the data transfer in TLTCP is indeed time-lined and then quantify the TCP-friendliness of TLTCP over a wide range of network conditions.

4.1 Time-lined Data Transfer

Using a simulated network as shown in Figure 4.1 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 when operating under the same network conditions. In particular we want to determine, if the TLTCP sender stops sending sequences from the current section at the expiry of the specified deadline and if it then begins to send the sequences for the next section. The TCP sender on the other hand is expected to send a continuous stream of data sequences. For the sake of clarity in Figure 4.2, we use constant sized data sections of 700,000 bytes each associated with constant deadlines of 1 second, in order to ensure that the whole section cannot be delivered within the given deadline. The other parameters used in this simulation are shown in Table 4.1 and justification for the values is provided in the next section.

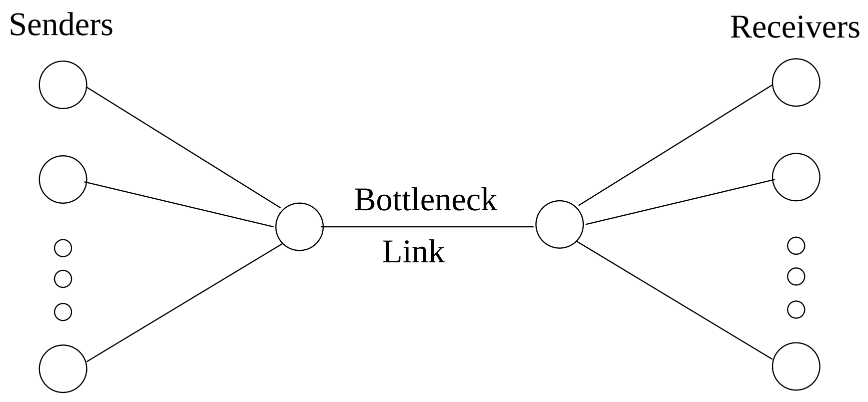


Figure 4.1: Topology used for simulations.

Shown in Figure 4.2 is a plot of sequence number verses time where each sequence number represents a 1,500 byte data packet. Let us make some observations before we provide explanations for them. It can be seen that the sequence numbers of the packets of the TLTCP session are discontinuous unlike that of the TCP session. It can also 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

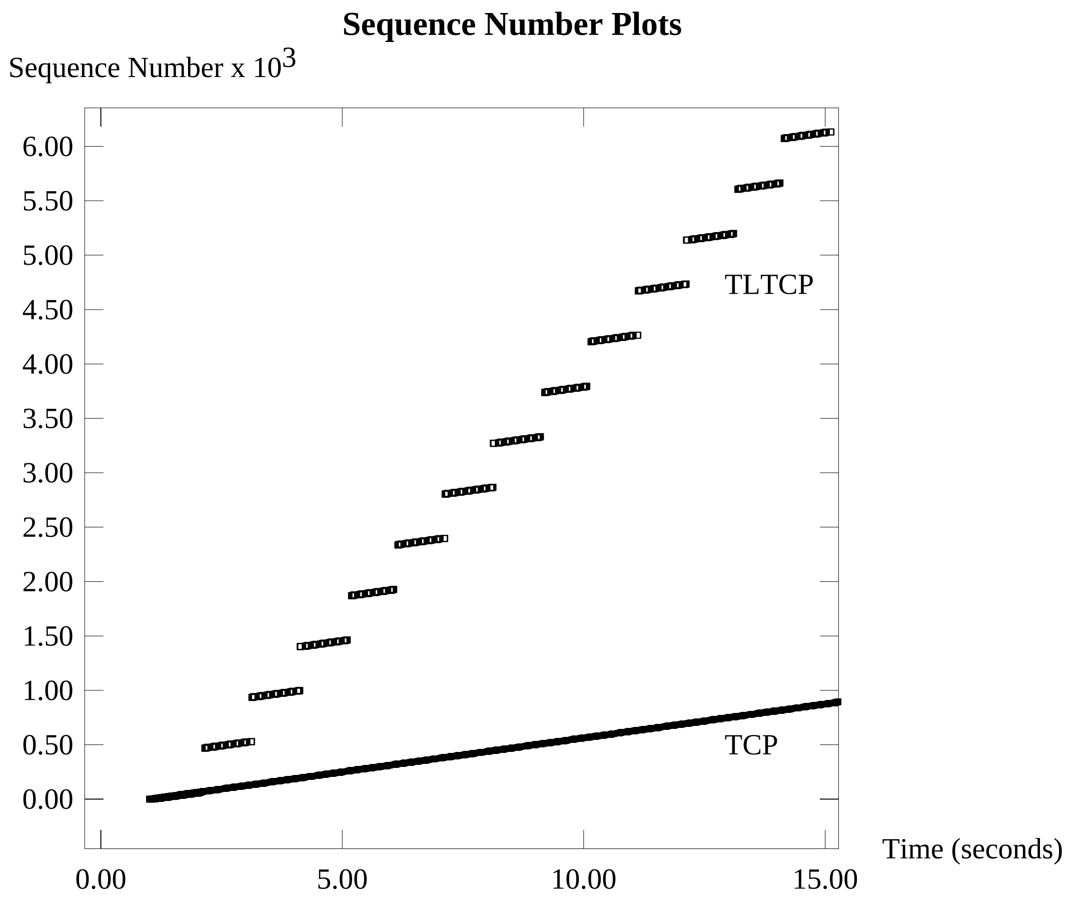


Figure 4.2: Data sending characteristics of TLTCP as compared to TCP operating under the same network conditions.

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 ($\lceil 700000 \div 1500 \rceil = 467$). That is why at the end of every second the next sequence sent is a multiple of 467, indicating that TLTCP is starting to send a new section. Throughout our experiments this pattern of data sending is observed in TLTCP; namely, at the end of the specified deadline the TLTCP sender stops sending data from the expired sections and starts sending data from newer sections. It can thus be inferred that TLTCP indeed performs data transfer in a time-lined manner. Note that to an external observer the TLTCP and TCP streams are similar if the discontinuities in the sequence numbers are ignored.

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). Fairness in sharing the available bandwidth is an important goal for TLTCP and in the next section, we investigate this further by measuring the TCP-friendliness of TLTCP.

4.2 TCP-friendliness

4.2.1 The Metrics

In several studies [6] [31] [26] 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* [31] [26], 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 [26]. 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 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 $min(\overline{oddflows}, \overline{evenflows})$.

$F_{best} = max(\overline{oddflows}, \overline{evenflows})/min(\overline{oddflows}, \overline{evenflows})$.

Note that in order to ensure that the partitions $oddflows$ and $evenflows$ and $max_{n/2}$

and $min_{n/2}$ are of equal sizes we keep the total number of flows, N, even in all our experiments.

4.2.2 The Methodology

In order to examine the extent to which TLTCP is TCP-friendly we conduct several experiments using the *ns-2* simulator [42]. Since the TCP flows themselves do not share bandwidth equally if their round-trip times are not equal [18] [9], we consider N sources configured symmetrically (as shown in Figure 4.1) 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, which is shown in Table 4.1, 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 [7]. 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 [39]. 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 [22] 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 heavily 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.

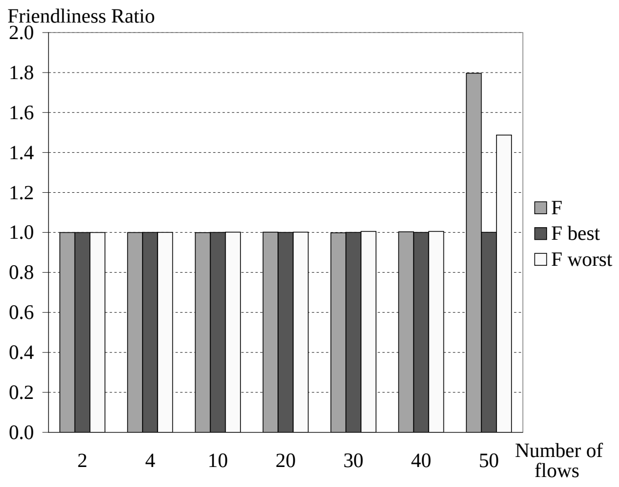
Parameter	Value
Packet size	1,500 bytes
ACK size	40 bytes
Bottleneck link BW	1.5 Mbps
Bottleneck link delay	20 ms
Router buffer size	400 pkts
Source/Dest link BW	10 Mbps
Source/Dest link delay	2 ms
Receiver max window size	10 pkts
Simulated time	500 sec
Size of TLTCP sections	700,000 bytes
Deadlines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

Table 4.1: Default simulation parameters.

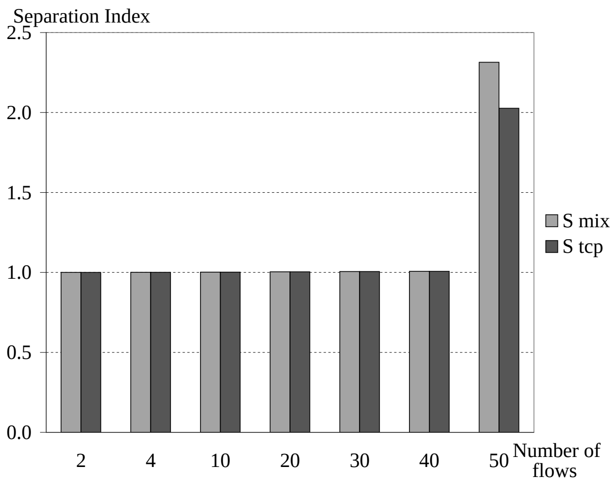
4.2.3 Varying the Number of Flows

In our first set of experiments we vary the total number of flows and examine our metrics for TCP-friendliness. As mentioned before, an important design goal is that the TLTCP flows should share bandwidth with TCP-flows over the bottleneck link in an equitable fashion. The object of this set of experiments is to progressively increase contention at the bottleneck by increasing the number of competing flows in order to study the resource sharing behavior of the TLTCP flows.

Figure 4.3(a) shows the variation in the friendliness ratio while the Figure 4.3(b) shows the variation in the separation indices when the total number of flows (N) are varied. Recall that in all our experiments, of the total N flows half ($N/2$) are TCP and the other half ($N/2$) are TLTCP. The X-axis in both the figures represents the number of flows. The height of the bars (Y-axis) in the Figure 4.3(a) represents the values for the parameters F, F_{best} and F_{worst} respectively. Similarly the height of the bars in Figure 4.3(b) represent the values of the separation indices S_{MIX} and S_{TCP} respectively. In both the figures, the closer the values of F, F_{best}, F_{worst}, S_{MIX} and S_{TCP} are to 1 the more equally the bandwidth is being shared. Section 4.2.1 gives a detailed description of the above metrics.



(a) Friendliness ratios.



(b) Separation indices.

Figure 4.3: Varying the number of competing flows.

As seen in Figures 4.3(a) and 4.3(b) 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. This can be seen in F_{worst} and F_{best} in Figure 4.3(a) and S_{TCP} in Figure 4.3(b), where the value are not close to 1. 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. By examining the traces for the experiment we see that in the case of 50 flows many of the TCP flows obtain half the throughput of the rest of the competing flows (both TCP and TLTCP).

The situation where a number of TCP streams compete over a single bottleneck router has been studied previously by Morris [22]. He has observed that if there are a large number of competing flows, TCP's congestion control mechanisms fail to ensure fair 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. This is also seen in our experiment with 50 flows and is illustrated by F_{worst} in Figure 4.3(a) and S_{TCP} in Figure 4.3(b). 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 amount of buffer space 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

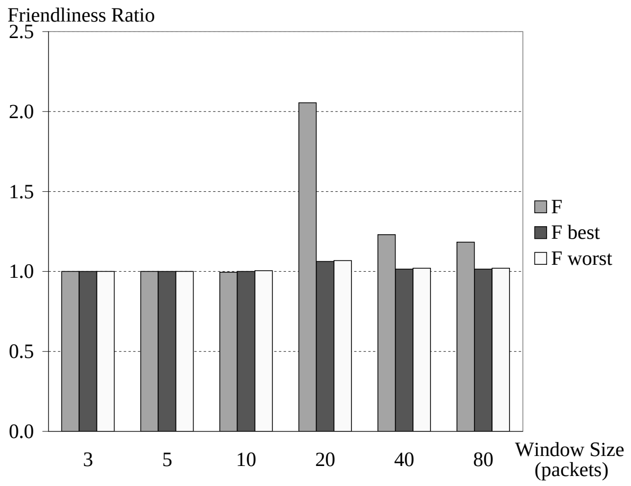
Figures 4.3(a) and 4.3(b) that in the experiments with a mix of TCP and TLTCP flows, higher values for the friendliness ratio and separation index are observed as compared to the baseline experiment with just TCP flows. This indicates that some TLTCP flows obtain larger throughput than the rest of the 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 3.2.5, 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 (this was also observed by Morris [22]). 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.

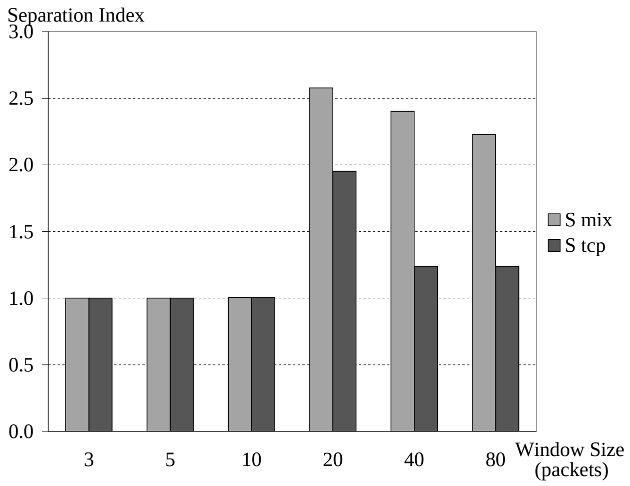
4.2.4 Varying the Maximum Window Size

In this section, we consider the impact of increasing the maximum receiver window sizes on the TCP-friendliness of TLTCP. A large receiver window size would allow the flows to potentially utilize a large send window size (up to the receiver window size). As noted in Section 3.2.5, 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, note that in both TCP and TLTCP large receiver windows increase the possibility of greater variations in send window sizes among competing flows. Therefore, we expect to see an inequitable distribution of bandwidth for larger window sizes.

In the Figures 4.4(a) and 4.4(b) the Y-axes represent the friendliness ratios and separation indices respectively while the X-axes show window sizes of 5, 10, 20, 40 and 80 packets. These represent windows 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 [39]. The remaining values were chosen to significantly exceed these commonly used sizes.



(a) Friendliness ratios.



(b) Separation indices.

Figure 4.4: Varying the maximum receiver window 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.

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. With a mix of TCP and TLTCP flows all with the maximum window size of 20, we see that the value of F is close to 2 and the value of S_{MIX} is close to 2.5. It can also seen from Figure 4.4(a) 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 (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. Note that this dynamic behavior is essential for the flows to adapt to the changing network conditions in the Internet (e.g., when the number of competing flows changes). 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, a flow with a larger sending window size is more likely to experience packet losses than a flow with a smaller window. Thus 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.

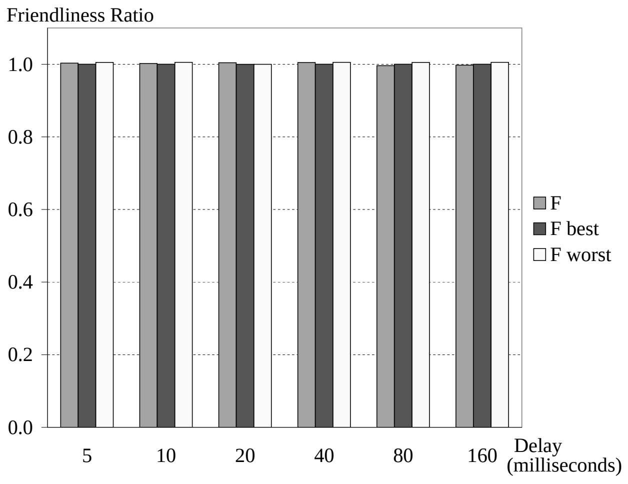
4.2.5 Varying the Propagation Delay

It is known that different flows between different pairs of hosts in the Internet would encounter a wide variety of round-trip delays; from the large delay of an intercontinental satellite connection to the small delay of a cross-campus connection. It is thus important that a transport protocol be able to function properly across a wide range of round-trip delays.

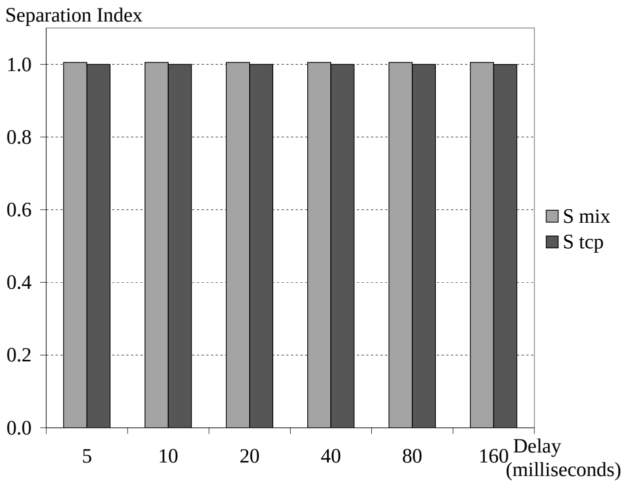
Dealing with a large range of round-trip delays has been reported as a problem with existing rate-based streaming media protocols. In their work on TFRCP, a rate-based protocol, Pahdye *et al.* [26] report that with *small* round-trip delays TFRCP behaves aggressively as compared to TCP, therefore obtaining a larger share of the bottleneck bandwidth than the competing TCP flows. They explain that this is likely because a rate-based protocol may not able to react to traffic fluctuations as quickly as TCP. 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 more quickly react 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. Figures 4.5(a) and 4.5(b) show the friendliness ratios and separation indices respectively, in their Y-axes while the X-axes represent the range bottleneck delays, form



(a) Friendliness ratios.



(b) Separation indices.

Figure 4.5: Varying delay.

5 milliseconds to 160 milliseconds. 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. This indicates that under the conditions used in this experiment TLTCP is able to operate in a TCP-friendly fashion for a wide range of delays. Moreover, 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.

4.2.6 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 obsolete data increases, as does the potential for handling larger amounts of obsolete data. Therefore, in the next set of experiments we examine the impact of a range of deadlines on the friendliness of TLTCP.

Figure 4.6 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. The amount of data that the application associates with each deadline is kept constant at 700,000 bytes. Note that for the same section size of 700,000 bytes, the experiments reported in Sections 4.2.3, 4.2.5 and 4.2.4 use the deadlines of 5 seconds while the experiment reported in Section 4.1 uses the deadline of 1 second. 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 deadlines. However, for very short deadlines TLTCP is not able to share bandwidth equitably. In this case the deadline interval is 0.5 seconds which corresponds to a data rate of 11.2 Mbps per stream (with 30 such streams) over a 1.5 Mbps link. 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.

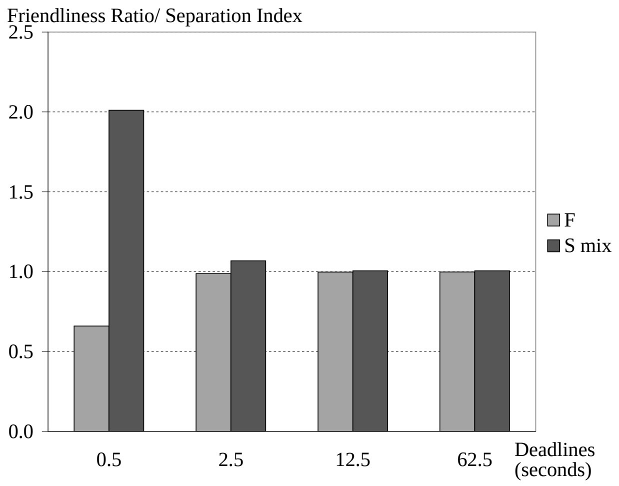


Figure 4.6: Varying data deadlines.

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.

The actual number of packets sent for a section of data depends on the bandwidth obtained by the stream. But with small deadlines of 0.5 seconds we observe that 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. This results in only a few instances of fast-retransmit and fast-recovery because at least three duplicate ACKs are needed to trigger fast-recovery. 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 (where the TLTCP flows obtain less throughout than the rest of the flows) than with deadlines of 2.5 seconds (where the average throughput are approximately the same). 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 4.6) is unlikely to occur when TLTCP is being used by a real application.

We conclude the chapter by noting that TLTCP not only transports data in a timelined 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.

Chapter 5

Conclusions and Future Work

We conclude this thesis by summarizing the problem studied and the proposed solution, TLTCP. As with most research, this thesis has led us to think about several related research problems which we outline in the future work section.

5.1 Conclusions

This thesis proposes a new transport protocol, called Time-lined TCP (TLTCP), that is designed for the transport of streaming media over the Internet. Remaining within the confines of TCP's window-based congestion control, TLTCP incorporates new mechanisms that allow it to deliver time-sensitive data without requiring data reliability.

There are two main issues to be addressed in designing such a protocol. Firstly, since the data being transported is time-sensitive, data that arrives late is no more useful than lost data. As a result, at any point in time during playback the protocol must attempt to deliver only the data that is still relevant for playback at the client and should avoid sending obsolete data. Secondly, since the protocol is meant to operate over the Internet, it should incorporate robust end-to-end congestion control mechanisms that ensure equitable sharing of the network bandwidth with the existing traffic. Since most of the current traffic in the Internet is TCP, the property of being fair to the existing traffic is called TCP-friendliness.

The problem of ensuring TCP-friendliness has been the focus of much recent research. Most of these approaches have proposed rate-based congestion control, a mechanism that has some inherent flaws. We choose to investigate an alternative approach that uses window-based congestion control for the delivery of streaming media data. Because streaming media data is error-resilient and time-sensitive, we have introduced new mechanisms in TLTCP that allow time-lined delivery of data but does not require reliability. Most of the mechanisms of TLTCP are based on TCP. However, instead of treating all data as a byte stream, TLTCP requires that each section of streaming media data be associated with a deadline. A TLTCP sender, upon expiry of the deadline for a section of data discards it and starts sending data from the next section whose deadline has not expired.

We have implemented TLTCP in the network simulator *ns-2*. In our first experiment we show that the TLTCP sender, upon expiry of a deadline, skips the sequences for obsolete data and starts sending data from newer sections. Thus we verify that TLTCP indeed performs data delivery in a time-lined manner. Furthermore, in order to accurately measure the impact that TLTCP flows have on competing TCP flows we perform extensive experiments simulating a wide range of network conditions. We use two metrics for quantifying TCP-friendliness that are based on the throughput obtained by the competing streams, the separation index and the friendliness ratio. Our simulation results show that TLTCP competes fairly with TCP over a wide range of network conditions and in most of the scenarios where it is not TCP-friendly, TCP itself does not share bandwidth equitably. We have used simulations because it allows us to discount external factors that may have introduced noise in our measurements in live Internet experiments. In addition, simulations facilitate the testing of TLTCP over a wide range of scenarios that are difficult to produce and control in live Internet experiments.

TLTCP is designed to serve streaming media applications. In a streaming media system the server creates a sending schedule for the media file requested. The schedule specifies parts of media data that are needed for playback before specified deadlines. TLTCP's API exploits the characteristics of streaming media data by requiring that the server associate deadlines with each section of media data as per the sending schedule. At the client end the TLTCP API, not only delivers data but also informs the playback application of the holes in the delivered sequences, facilitating reconstruction and playback of application data. The TLTCP API itself does not introduce new socket calls but involves augmentating to two existing socket calls, `recvmsg` and `sendmsg`.

5.2 Future Work

During the research described in this thesis we have come across a number of issues that would be interesting topics for future research. Some of these are natural extensions to our present work, while others pose new challenges. The following is a list of possible directions for future research.

- In this thesis, our investigation of the behavior of TLTCP is restricted to simulation experiments. While this allowed us to measure the friendliness metrics accurately and over a wide range of network conditions, live Internet experiments with a full kernel implementation are necessary in order to validate our simulation experiments. Not only would live Internet experiments allow the testing of TLTCP under realistic traffic conditions but operating from a real kernel implementation would illustrate the impact of non-network elements (e.g., scheduler latency [2] and variable packet sizes).

- The experiments reported in Chapter 4 are not driven by a real application. Instead we send dummy data at unrealistically large data rates, in order to stress test the time-line mechanisms of TLTCP. An important next step would be to create a real streaming media player that uses TLTCP. This could be done using a software codec for a particular streaming media format and the proposed TLTCP API. Using such a streaming media application, the impact of using TLTCP on a real application can be measured in terms of application performance and compared with the performance of other streaming media systems. Furthermore, by measuring the application's performance and friendliness with competing traffic at the same time one can better evaluate the tradeoff between application performance and TCP-friendliness discussed in Section 2.3.4.

- Some changes to the TLTCP protocol may facilitate its wide use and deployment on the Internet. For instance, if the protocol can be modified in such a way that only the server's network stack needs to be modified to deploy TLTCP then a TLTCP server would be able to provide streaming media over the Internet to all the unmodified TCP clients. With this in mind, TLTCP is designed to require minimal changes to the TCP receiver. We believe that by modifying the sequence numbers at the server in such a way that the discarded data is masked a TLTCP server can operate with an unmodified TCP receiver. The drawback of this approach is that the receiver would not be aware of the gaps in the data sequence. Thus it may require additional complexity in the playback application which would potentially decrease performance. However, most existing streaming media applications are not informed of data losses by the transport layer and incur this overhead.

- While the points above are in spirit close to the proposed protocol (TLTCP), we believe there are several other approaches to the general problem of streaming media data transport that require further investigation. Although there exists an abundance of literature on rate-based mechanisms for congestion control in streaming media systems [36] [26] [31], possibilities for using the more robust window-based mechanisms have not been adequately explored for unreliable data transport. Our work on TLTCP explores the use of TCP's window-based mechanisms for the transport of streaming media data. The next step would be to deviate from the window control mechanisms used by TCP in order to provide better support for streaming media. One possible avenue is to incorporate window management mechanisms that improve smoothness in data transport. This may require less buffering at the client and thus lead to an improvement in the performance of a streaming media application. The new mechanisms should however ensure that the fundamental principle of window-based congestion control, such as the conservation of packets, is not be violated.

- There are several emerging trends in the area of networking that can potentially change the structure and composition of the Internet as we know it today. In order to apply TLTCP in these new environments existing mechanisms of TLTCP may require modification. One of the emerging trends in the area of networking has been that of wireless networks. Although a lot of work has been done to improve reliable data transport over wireless networks, we believe that there is room for further work especially in the area of unreliable data delivery, such as streaming media over wireless networks. It would be interesting to explore the window-based mechanisms of TLTCP in this context.

- Another emerging trend is that of multicast networks. While there is currently little support for multicast data delivery in the Internet, more mechanisms are expected to be deployed in the near future. TLTCP in its present form is designed for unicast transport. To take advantage of the multicast mechanisms that may be deployed, TLTCP needs to be extended to support multicast. Congestion control in multicast streams is an involved problem in itself and is currently an active area of research. It is believed that the simple ACK based scheme of TCP may not scale for multicast. It would be a challenging problem to create a multicast equivalent of the robust congestion control mechanisms of TCP.

Bibliography

[1] H.261 recommendation – video codec for audiovisual services at p*64kbp/s. Technical report, International Telecommunication Union, March 1993.

[2] M. Allman and A. Falk. On the effective evaluation of tcp. *ACM Computer Communications Review*, 29(5), 1999.

[3] T. Berners-Lee, R. Cailliau, A. Luotonen, H.F. Nielsen, and A. Secret. The world-wide web. *Communications ACM*, 37(8):76–82, August 1994.

[4] 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.

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

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

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

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

[9] 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.

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

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

[12] B. Furht. Multimedia systems: An overview. *IEEE Multimedia*, 1(1):47–59, 1994.

[13] J. Gemmel, H. Vin, Kandlur, V. Rangan, and L.A. Rowe. Multimedia storage servers: A tutorial and survey. *IEEE Computer Magazine*, 28(5):40–49, 1995.

[14] T.R. Henderson, E. Sahouria, S. McCanne, and R.H. Katz. On improving the fairness of tcp congestion avoidance. In *Proceedings of Globecomm*, 1998.

[15] 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.

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

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

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

[19] 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.

[20] S. McCanne. *Scalable Compression and Transmission of Internet Multicast Video*. PhD thesis, University of California, Berkeley, December 1996.

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

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

[23] K. Nahrstedt and R. Steinmetz. Resource management in networked multimedia systems. *IEEE Computer Magazine*, 28(5):52–63, 1995.

[24] Real Networks. Real networks press release, December 7 1999.

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

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

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

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

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

[30] 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.

[31] 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.

[32] 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.

[33] J. Salehi, Z. Zhang, J. Kurose, and D. Towsley. Supporting stored video: Reducing rate variability and end-to-end resource requirements through optimal smoothing. In *ACM SIGMETRICS*, 1996.

[34] H. Schulzrinne, S. Casner, R. Frederick, and V. Jacobson. Request for comments: 1889, January 1996.

[35] H. Schulzrinne, A. Rao, and R. Lanphier. Request for comments: 2326, April 1998.

[36] 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.

[37] R. Steinmetz. Analyzing the multimedia operating system. In *IEEE Multimedia*, April 1995.

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

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

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

[41] T. Turletti, S. F. Parisis, and J-C. Bolot. Experiments with a layered transmission scheme over the internet. Technical Report 3296, INRIA, Sophia Antipolis, France, November 1997.

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