Available at
www.ElsevierComputerScience.com
POWERED BY SCIENCE DIRECT®

Computer Communications 27 (2004) 262–272

computer communications

www.elsevier.com/locate/comcom

Designing and implementing a new type of transport-layer socket: the UDTCP socket case

S.Y. Wang*

Department of Computer Science and Information Engineering, National Chiao Tung University, 1001 Ta-Hseuh road, Hsinchu 30050, Taiwan

Received 5 March 2003; revised 1 July 2003; accepted 11 July 2003

Abstract

On most operating systems, the UDP and TCP sockets are the two main types of sockets used to provide transport-layer networking services. However, for several reasons, UDP and TCP sockets are unsuitable for transporting delay-sensitive but error-tolerant streaming data such as the data generated by multimedia streaming applications. In this paper, we create a new type of socket that is suitable for transporting such data and propose a novel and simple implementation for it.

We have implemented this new type of socket in Free BSD 4.8 and call it the 'UDTCP socket.' It has both the UDP and TCP socket properties suitable for delay-sensitive but error-tolerant streaming data but not those UDP and TCP socket properties unsuitable for such data. When transporting a stream of such data whose sending rate needs to be regulated by TCP congestion control, our simulation results show that the UDTCP socket can provide a much better delay and delay-jitter performance than the TCP socket.

Keywords: Protocol design; Protocol implementation; Transport-layersocket; Operating system; Multimedia streaming

1. Introduction

Multimedia streaming applications are becoming more popular on the Internet [14]. These applications are usually delay and delay jitter-sensitive. However, because human perception can tolerate some errors, usually some degree of packet loss and packet reordering can be tolerated by these applications. By using dynamic variable rate encoding or layering techniques, these applications can be made adaptive [3,12,13,19,23,24,31]. The bandwidth requirement for such an adaptive application can be varied over time to dynamically match the current available bandwidth. This often results in lower packet loss rates and smaller delays.

Most of these delay-sensitive, but error-tolerant multimedia streaming applications use UDP rather than TCP to transport an audio or video stream on the Internet [14]. An example is Microsoft Inc's 'Media Player' product. These applications choose to use UDP rather than TCP because otherwise some undesirable delay problems will result as follows. First, when TCP packets are lost, TCP at the sending host will automatically retransmit them until they are received by the receiver. However, retransmitted packets may arrive too late to be useful for these delay-sensitive applications. This will only waste network bandwidth. Second, because TCP is designed to provide a reliable and in-sequence delivery service, when some TCP packets are lost, TCP at the receiving host will put other received out-of-order packets into its reassembly queue for re-sequencing. (Note: As long as one TCP packet is lost, all following TCP packets are viewed as out-of-order even though they are in sequence.) This design unnecessarily delays the delivery of arrived packets (actually, the data carried in these packets) to delay-sensitive multimedia streaming applications.

Although using UDP for multimedia streaming applications avoids the TCP delay problems, excessively using UDP for such applications can disrupt Internet congestion control. This problem has been pointed out and discussed in Refs. [8,4]. Motivated by this problem, many researchers have been working on this problem to design 'TCP-friendly' or 'TCP-like' protocols for such applications. This is evidenced by the many references cited in Section 2.

* Corresponding author. Tel.: +886-3-5131550; fax: +886-3-5724176.
E-mail address: shieyuan@csie.nctu.edu.tw (S.Y. Wang).

0140-3664/$ - see front matter © 2003 Elsevier B.V. All rights reserved.
doi:10.1016/j.comcom.2003.07.002

Unlike TCP, UDP performs no congestion control. Today many multimedia streaming applications use a fixed rate (e.g. 28, 33, 56, 112 Kbps or 1.5 Mbps) to transport their audio or/and video stream data. If these applications become popular and their traffic constitutes a large portion of the Internet traffic, the use of UDP as their transport protocol will disrupt Internet congestion control and cause problems. The first problem is congestion collapse. When a larger portion of the Internet traffic is carried by UDP, network congestion control will become less effective. The packet drop rates in routers will increase and the effective throughput of a network will decrease. The second problem involves fairness with TCP. Since TCP performs congestion control while UDP does not, when sharing network bandwidth, UDP-based applications will have unfair advantages over TCP-based applications. Therefore, during network congestion, these UDP-based applications may cause TCP-based applications such as web, ftp and email to stop their data transfers. More concerns about this problem have been discussed in Ref. [8].

Recently, to solve the above problems, many approaches have been proposed to apply TCP congestion control to a UDP packet stream. In these approaches, the multimedia streaming application uses a UDP socket to send and receive packets and a 'TCP-friendly' or 'TCP-like' congestion control protocol to control when to send data into the UDP socket. Because it is difficult to prove that these approaches implement true TCP congestion control under any given network condition, these approaches are often called 'TCP-friendly' or 'TCP-like'.

It is desirable if we can provide true TCP congestion control for a UDP packet stream under any network condition. Although a 'TCP-friendly' or 'TCP-like' congestion control scheme can demonstrate that in the 'long' run, under certain conditions, and in some studied network configurations, the proposed scheme is 'TCP-friendly', it is difficult to guarantee that a proposed scheme will not act too aggressively when the regulated flow is short-lived, under realistic network conditions, or in a unstudied network configuration. As such, a 'TCP-friendly' or 'TCP-like' congestion control scheme may behave unexpectedly and be harmful to network congestion control in some untested conditions and configurations. For example, in Ref. [34], the authors analytically and experimentally demonstrated that three TCP-friendly approaches [9,27,35] exhibit quite different fairness, smoothness, responsiveness, and aggressiveness properties than those of TCP under various network conditions.

To provide true TCP congestion control but no TCP error control for a stream of delay-sensitive but error-tolerant packets, we design and implement the UDTCP socket in FreeBSD 4.8. The UDTCP socket is a new type of socket. It has the properties of UDP and TCP sockets that are suitable for multimedia streaming applications but not those UDP and TCP properties that are unsuitable for such applications. The UDTCP socket properties include (1) datagram (i.e. unlike TCP's byte-stream service, the message boundary of application data is preserved), (2) true TCP congestion control (i.e. packets are sent under true TCP congestion control), (3) no mandatory TCP error control (i.e. lost/corrupted packets need not be forcibly retransmitted by the sending host), and (4) no mandatory TCP out-of-order packet re-sequencing (i.e. out-of-order packets need not be forcibly delayed in the TCP reassembly queue). Because of these properties, the UDTCP socket is very suitable for multimedia streaming applications.

The contributions of this paper are as follows. The first contribution is about protocol design. We create a new type of socket (UDTCP) that uses TCP congestion control but no TCP error control to transport a stream of delay-sensitive but error-tolerant data. The transported stream is 100% TCP friendly and causes no harm to network congestion control because the default in-kernel TCP congestion control is used. The second contribution is about protocol implementation. We propose a novel and simple implementation for the UDTCP socket. On FreeBSD 4.8, we can convert the TCP socket implementation into the UDTCP socket implementation by adding, modifying, or deleting only 43 lines of C statements of the TCP socket implementation.

The rest of this paper is organized as follows. Section 2 surveys related work. Section 3 presents our design and implementation of the UDTCP socket. Section 4 describes the design features that reduce the queuing delay of packets waiting in a UDTCP socket send buffer. Section 5 presents an analysis showing that to reduce the delay, re-sequencing packets in the TCP reassembly queue should be eliminated. Section 6 presents simulation results showing that the UDTCP socket provides a much better delay and delay-jitter performance than the TCP socket. Section 7 presents two extensions of the UDTCP socket. Section 8 discusses future work. Finally, Section 9 concludes this paper.

2. Related work

Many approaches have been proposed to apply TCP congestion control to a UDP packet stream. In these approaches, a multimedia streaming application program uses the UDP socket to send application data, and implements a 'TCP-like' or 'TCP-friendly' congestion control mechanism at the user-level to control the timing of sending application data. If desired, RTP [26] headers can be used to carry useful control information (e.g. sequence number) with data, and RTCP [26] control packets can be used to exchange control information between the sending and receiving application programs. Most of these proposed approaches can be classified into two categories. The first category is TCP simulation and the second is model or equation-based.

In the first category, to regulate the sending rate of a UDP packet stream, an approach can be either window-based or rate-based. A window-based approach, like TCP, maintains

a congestion window size variable (similar to cwnd in BSD TCP implementation) to control the current sending rate of a UDP packet stream. The goal is to simulate TCP's congestion window update algorithms [11,28] as closely as possible. A rate-based approach maintains a rate variable and uses TCP's additive increase and multiplicative decrease (AIMD) [7,11] principle to update it. To demonstrate that a proposed approach is TCP-friendly, extensive simulations or experiments are run to show that in the long run and in the studied network configurations, the proposed scheme can make a UDP packet stream achieve only the bandwidth that a TCP connection would achieve under the same environment. Examples of these approaches are presented in Refs. [2,3,5,6,22,25,29,32].

In the second category, a proposed approach tries to model the steady state behavior of TCP congestion control and uses the model to derive the relationship between the achieved throughput, packet loss rate, and round-trip time (RTT) of a TCP connection. To make a UDP packet stream TCP-friendly, such an approach would constantly measure the current packet loss rates and RTTs of the UDP packet stream, use the derived relationship to calculate the current fair share of available bandwidth for the UDP packet stream, and then use a simple rate-based scheme to send out UDP packets. Extensive simulations or experiments are run to demonstrate TCP-friendliness. Examples of these approaches are presented in Refs. [1,9,10,15–18,20,21].

There is a big implementation difference between these existing approaches and our UDTCP socket approach. In these existing approaches, the data generated by multimedia streaming applications are transmitted in UDP packets (i.e. using the UDP headers). The timing for sending out these UDP packets is controlled by a user-level controller that tries to simulate TCP's behavior. For the sending host to get RTT estimates and ACK sequence numbers (both of which are required to simulate TCP's behavior), the receiving host needs to constantly send back extra UDP packets, which are used to play the same role as TCP ACK packets. Since UDP does not automatically retransmit lost data, data carried in a lost UDP packet need not be forcibly retransmitted. Besides, since UDP does not re-sequence out-of-order data at the receiving host, out-of-order data need not be forcibly put into a reassembly queue and suffer unnecessary delays.

In contrast, in our UDTCP socket approach, data are transmitted in TCP packets (i.e. using TCP headers). These TCP packets are sent out under the true TCP congestion control that is already implemented in the kernel. Since the TCP protocol is used, to avoid forcibly retransmitting lost data, data are removed from the socket send buffer immediately after they are sent. To avoid suffering unnecessary delays at the receiving host when data arrive out of order, in our UDTCP socket approach, data will bypass the reassembly queue when they arrive.

3. Design and implementation of the UDTCP socket

In order to use TCP congestion control to transmit a stream of delay-sensitive but error-tolerant data and eliminate the need for implementing a complicated 'TCP-like' or 'TCP-friendly' protocol scheme in streaming application programs, we propose a novel and simple implementation to easily create a new type of socket that is suitable for such applications. This new type of socket is called the 'UDTCP socket' and is implemented in the kernel. It can be used by multimedia streaming application programs as follows.

A multimedia streaming application program can simply pump its data into a UDTCP socket as long as the UDTCP socket can accept more data. Data pumped into the UDTCP socket are transmitted as TCP packets under true TCP congestion control. Although the transmitted packets are TCP packets (i.e. using the TCP header), because we preserve the message boundary of application data and do not automatically retransmit lost data, they have the properties of UDP packets.

The design and implementation of the UDTCP socket are simple and almost the same as the TCP socket. In fact, we reused the in-kernel TCP socket implementation and made only four minor changes to convert a TCP socket into a UDTCP socket. Only 43 lines of C statements of the original TCP socket implementation need to be modified, added, or deleted. This number is small compared to 9708, which is the number of C statements in the TCP socket implementation of the FreeBSD 4.8 operating system.

Fig. 1 shows the four changes that are made to the TCP socket implementation. Three of these changes (Change 1, 2, and 3) are made to the control path of sending TCP packets from the sending host to the receiving host. One change (Change 4) is on the control path of receiving TCP acknowledgment packets at the sending host. These four changes are explained in the following: description

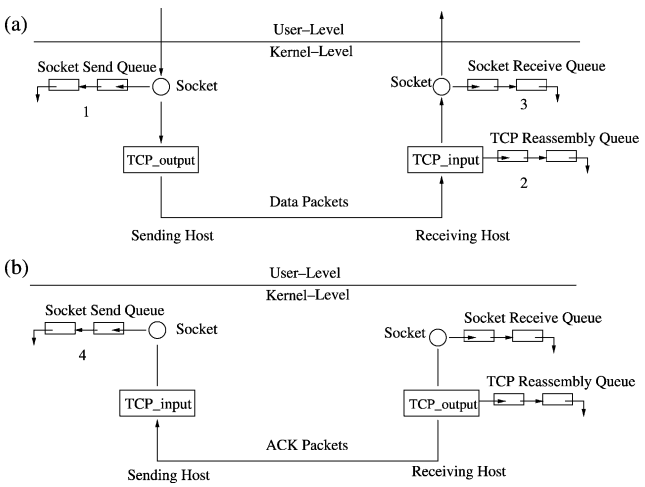

Fig. 1. The four minor changes made to the TCP protocol implementation to implement the UDTCP socket.

Change 1 is about the socket send buffer at the sending host. The purpose of this change is two fold. First, we want to preserve the message boundary of application data. The data generated by a streaming application normally is an audio or/and video stream. Such streams normally are composed of many frames. For example, an MPEG video stream may be composed of many I, P, and B frames. To facilitate frame format decoding at the receiving side, normally these frames are transmitted as separate UDP messages. The UDTCP socket thus also wants to be able to preserve message boundary.

However, traditionally because TCP provides a byte-stream service, application data separately written into a TCP socket is concatenated together and can be partially transmitted in any manner. To preserve the message boundary, application data separately written into a UDTCP socket should be kept separate in the socket send buffer. Because BSD UNIX's packet buffer structure (mbuf) allows the message boundary to be preserved and application data separately written into a UDP socket can already be kept separate in the socket send buffer, this change is easily done.

Second, we want to eliminate TCP's mandatory packet retransmission mechanism, which is used for recovering lost packets. Traditionally, because TCP provides a reliable service, a lost packet must be retransmitted until it is received by the receiving host. This retransmission attempt is canceled after 12 unsuccessful retransmissions, whose waiting time for retransmission is exponentially doubled after each unsuccessful retransmission. (The waiting times are 1, 2, 4, 8, 16, 32,…,128, 128, 128 s.)

It is clear that automatic retransmission is bad for a delay-sensitive but error-tolerant streaming application when the RTT between the sender and receiver is large. During a data transfer, to disable TCP's packet retransmission mechanism while still keeping TCP congestion control going, after a message (in this paper, a message means a piece of application data written atomically into a UDTCP socket) is transmitted, we immediately dequeue it from the socket send buffer. If the message is lost, which causes the TCP retransmission mechanism to be triggered, the message 'retransmitted' will be the next message to be transmitted (new), not the previously transmitted message (old). In such a design, a lost delay-sensitive message therefore will not be forcibly retransmitted. The decision to retransmit a lost message or not in the UDTCP socket approach is left to the multimedia streaming application program. Such a design is good for these programs because only the application-layer program knows whether a delay-sensitive message is still useful and worth retransmitting.

Implementation complexity: (1) A 10 C statement addition for using the mbuf's message boundary option, (2) A 15 C statement addition for forcing tcp_output() function to transmit a complete message, not a concatenated or partial message.

Change 2 is about the TCP reassembly queue at the receiving host. The reason for making this change is that we want an arrived out-of-order packet not to be delayed in the TCP reassembly queue. Rather, as soon as it arrives, we want it (actually, the message carried by it) to be available to the multimedia streaming application program immediately. Traditionally, because TCP provides a reliable and in-sequence delivery service, when a packet is lost, an incoming out-of-order packet must be put into the reassembly queue to wait for the lost packet to be retransmitted and arrive. However, for delay-sensitive but error-tolerant streaming applications, delaying already arrived packets will unnecessarily increases the chance that they will miss their playback times.

To avoid the delay problem caused by packet re-sequencing at the receiving host while keeping TCP congestion control going, we propose the following design for the TCP receiving host. This design can be described in three steps. Only one minor change needs to be made to the original TCP socket implementation in the second and third steps.

- First, to keep TCP congestion control going, as normal, an incoming out-of-order TCP packet still undergoes the default TCP receiving process. That is, it is put into the TCP reassembly queue and waits for the hole to be filled (i.e. waits for the lost packet to be retransmitted and arrive).
- Second, to allow the multimedia streaming application program to quickly access an already arrived packet, when a TCP packet arrives, we make a copy of the message carried by it and immediately enqueue the copy into the socket receive buffer.
- Third, when a lost and retransmitted packet finally arrives, which causes the hole in the TCP reassembly queue to be filled, now we dequeue a sequence of now-in-order TCP packets from the TCP reassembly queue but do not enqueue the messages carried by these TCP packets into the socket receive buffer. Traditionally in TCP design, these messages should be enqueued into the socket receive buffer when they become in order. However, since in the second step these messages have already been enqueued into the socket receive buffer, in the third step we should just discard them.

As previously described, in the UDTCP socket approach, the data payload carried in each retransmitted TCP packet is always a new message. This change to the original TCP socket design does not affect the operation and behavior of TCP congestion control. The reason is that in implementing TCP congestion control, only the control information carried in the TCP headers of transmitted TCP packets is important. The data payload carried in these TCP packets is irrelevant. This explains why in the above design, TCP

congestion control can still operate normally in the presence of packet losses.

Implementation complexity: (1) Six additional C statements to perform the copy and enqueue operations in the second step, (2) One C statement deleted to eliminate the enqueue operation in the third step.

Change 3 is about the socket receive buffer at the receiving host. This change preserves the message boundary of received application data in the UDTCP socket's receive buffer. As described previously, the BSD UNIX's packet buffer structure (mbuf) supports the message boundary option. It is easy to make this change.

Implementation complexity: (1) Add 10 C statements for using the mbuf message boundary option in the UDTCP socket's receive buffer.

Change 4 is about the socket send buffer at the sending host and deals with receiving acknowledgment packets. (It is in Fig. 1b.)

We need to make a change here so that when an acknowledgment packet is received, the messages in the UDTCP socket send buffer will not be dequeued. Traditionally, because TCP provides a reliable service, the application data in a TCP socket send buffer that has been transmitted may need to be retransmitted later. Therefore, application data queued in a TCP socket send buffer cannot be dequeued immediately after it is transmitted. Instead, it must wait until its corresponding acknowledgment packet is received. In this traditional design, a received acknowledgment packet may trigger the operation of dequeuing some application data from the TCP socket send buffer.

For the UDTCP socket send buffer, because messages are dequeued immediately after they are transmitted (described in Change 1), we should not let a received acknowledgment packet trigger the operation of dequeuing messages from the UDTCP socket send buffer.

Implementation complexity: (1) Delete one C statement to eliminate the dequeue operation. description

4. Reducing queuing delay

Usually a multimedia streaming application program is delay-sensitive. Therefore, the messages it generates should not experience long queuing delays in the UDTCP socket send buffer. However, because the UDTCP socket uses TCP congestion control to regulate the transmission of application messages, a long queuing delay problem may occur if there are many messages queued in the socket send buffer waiting to be sent and the network's available bandwidth suddenly decreases. To deal with this problem, we use the following method.

In this method, we limit the size of the UDTCP socket send buffer to only one message. This guarantees that as long as TCP congestion control does not time-out, a message queued in the UDTCP socket send buffer will be transmitted within one RTT of the multimedia stream. A multimedia streaming application program can greedily try to write its generated message into the UDTCP socket. If the socket send buffer is not full, the write operation will succeed and the application program can be assured that the message just written will be sent in one RTT if TCP does not time-out. On the other hand, if the UDTCP socket send buffer is full (indicated by the -1 returned from the write operation), the application program can use the select ( ) system call to wait until the socket send buffer becomes not full. At that time, the application program can write the current message to the UDTCP socket. If the current message has become too old to be useful, the application program can discard the current message, generate a more recent one, and write the new message to the UDTCP socket.

Fig. 2 depicts an example usage of the UDTCP socket. Inside the multimedia streaming application program, there are two tasks. Task 1 is responsible for generating the audio/video frames and task 2 is responsible for writing these frames as messages into the UDTCP socket. There is a FIFO frame buffer between task 1 and task 2. A generated frame is enqueued into the buffer by task 1 and later dequeued from the FIFO by task 2 for transmission. By maintaining the buffer occupancy around a threshold (e.g. one half of the buffer size), task 1 can use a variable-rate encoding or layering technique to dynamically change its bandwidth usage. If a frame in the FIFO becomes too old to be useful due to a sudden reduction in available bandwidth, task 1 can delete that frame when it needs to enqueue more recent frames to the FIFO but finds that the buffer occupancy is already above the threshold. Actually, since in the UDTCP socket approach frames are buffered in the application program rather than in the in-kernel socket send buffer, the application program can easily use more advanced or application-specific frame management schemes to manipulate its frames.

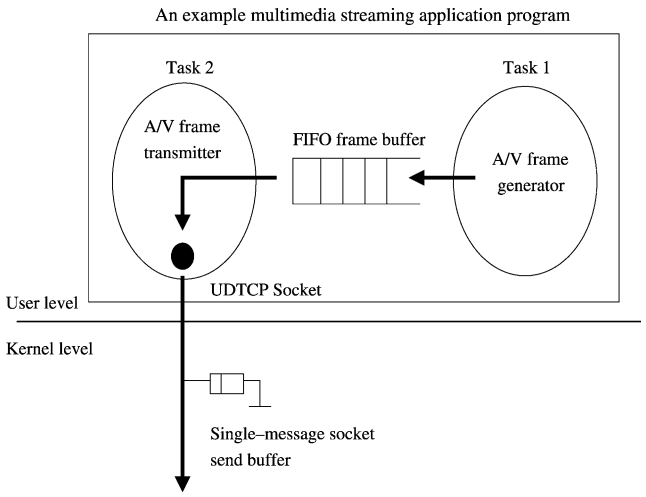

Fig. 2. An example usage of the UDTCP socket.

Because the messages in the UDTCP socket send buffer are dequeued immediately after they are transmitted, limiting the size of the UDTCP socket send buffer to only one message does not reduce the maximum throughput that a UDTCP connection can achieve. In contrast, this property cannot be achieved by the TCP socket. Traditionally, because TCP provides a reliable service, transmitted application data may need to be retransmitted and thus must remain in the TCP socket send buffer until a corresponding acknowledgment packet arrives. Since application data can be transmitted only after they are written into and stay in the socket send buffer, the size of a socket send buffer will limit the amount of application data that can be transmitted per RTT (i.e. in-flight). As a result, if the size of the TCP socket send buffer is less than the product of the available network bandwidth and RTT, it will become a throughput-limiting factor. In contrast, because the UDTCP socket does not intend to provide a reliable service, it does not suffer from the throughput-limiting problem even though we reduce its socket send buffer size to only one message.

5. Analysis

In this section, we perform a simple analysis to numerically show that re-sequencing out-of-order packets in a TCP reassembly queue should be eliminated; otherwise a large percentage of transmitted packets will wait unnecessarily in the reassembly queue for at least one RTT. (Note: Because one RTT is needed to recover a lost packet, if multiple packets are lost in a TCP window, out-of-order packets may need to wait more than one RTTs in the TCP reassembly queue.)

Fig. 3 shows the typical saw-tooth window growing and shrinking behavior of a greedy TCP connection. To simplify the analysis, we make two assumptions. First, packet losses are uniformly distributed over transmitted packets. That is, if the packet loss rate is $1/N$, then one packet loss will occur after N packets are successfully transmitted and received. Second, for each packet loss, the TCP fast retransmit and recovery mechanisms can always recover the lost packet without incurring a TCP time-out. Based on these assumptions, when a packet is lost, the TCP congestion control will cut the current congestion window size W to $W/2$ and then increase it by one packet every RTT until one packet is lost again. Because we assume that packet losses are uniformly distributed, the above cycle will repeat.

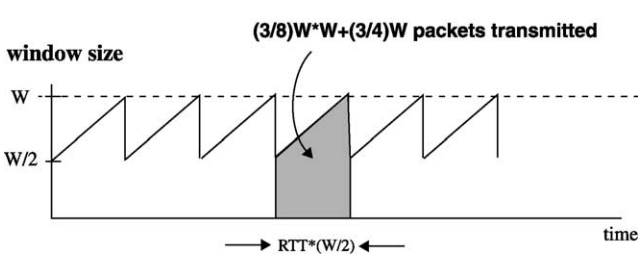

Fig. 3. TCP's typical saw-tooth window growing and shrinking behavior.

From Fig. 3, we see that in each cycle $W/2 + (W/2 + 1)+, \ldots, +W = (3/8)W \times W + 3W/4$ packets are sent and one packet is lost. When a packet loss occurs, the current congestion window size is W. This means that when a packet loss occurs, there are W packets following the lost packet in the network and all of them now become out-of-order packets. These W out-of-order packets need to wait in the TCP reassembly queue at least one RTT until the lost packet is retransmitted and finally arrives. It is clear that if the RTT is large, these packets will likely miss their playback deadline due to the extra one RTT delay.

Delaying arrived out-of-order packets in the TCP reassembly queue can cause a large percentage of transmitted packets to miss their deadline. In the above analysis, we see that in each cycle, W out of $(3/8)W \times W + 3W/4$ transmitted packets will need to unnecessarily wait at least one RTT delay. This out-of-order ratio $W/((3/8)W \times W + 3W/4)$ is large. For example, it is 22% and 12% when W is 10 and 20, respectively.

Another case in which packets may need to wait at least one RTT delay is when they are lost and retransmitted. However, this retransmission ratio is smaller and thus less important than the out-of-order ratio discussed above. In each cycle, since only one packet is lost out of $(3/8)W \times W + 3W/4$ transmitted packets, the retransmission ratio is equal to the packet loss rate, which is $1/((3/8)W \times W + 3W/4)$. Compared with the out-of-order ratio, we see that the retransmission ratio is only $1/W$ times of the out-of-order ratio. For example, the retransmission ratio is only 2.2% and 0.6% when W is 10 and 20, respectively.

Although the above analysis shows that the retransmission ratio is smaller than the out-of-order ratio, it does not mean that Change 1 (which eliminates retransmissions) is unimportant and thus can be eliminated. Actually, Change 1 is as important as Change 2 (which eliminates the delay spent waiting in the reassembly queue). The real benefit of Change 1 is that packets can be dequeued immediately after they are transmitted. Therefore, unlike TCP, the maximum throughput that a UDTCP connection can achieve is not limited by the size of its socket send buffer. It is this property that enables us to reduce the queuing delay spent in the socket send buffer while not limiting the maximum throughput that can be achieved by a UDTCP connection.

6. Simulation results

In this section, we present simulation results to demonstrate that, when used to transport the data of a multimedia streaming application program, the UDTCP socket provides a much better delay and delay-jitter performance than the TCP socket. Although we have implemented the UDTCP socket in FreeBSD 4.8 and can

generate experimental results, in order to compare the delay performance of the UDTCP and TCP sockets under various RTT conditions (which is difficult to achieve using real links) and avoid the task of precisely synchronizing many PCs' clocks (for measuring the delay experienced by transmitted messages), we used a network simulator to do the study. The simulator used is the NCTUns 1.0 network simulator [33], which is a high-fidelity and extensible network simulator and is available for download at http://NSL.csie.nctu.edu.tw/nctuns.html. We used this simulator because it can directly use our in-kernel UDTCP socket implementation to transport a multimedia streaming application program's data in a simulated network.

To compare the delay performance of the TCP and UDTCP sockets when they are used in different RTT conditions, we tested two different network and traffic configurations. The first configuration has a small RTT for multimedia streaming application programs while the second one has a large RTT. In each configuration, the TCP and UDTCP cases were tested to study the delay performance of the TCP and UDTCP sockets, respectively. Both of these two configurations used the simulation testbed depicted in Fig. 4. In this testbed, the link propagation delays of the links $(S1, C), (S2, C), \ldots, (S10, C)$ were set to 10 ms and the bandwidth of all of these links was set to 10 Mbps. The differences between these two configurations are summarized below. description

Configuration 1. The delay of the link (C, R) is 25 ms, the bandwidth of the bottleneck link (C, R) is 10 Mbps, and the maximum queue length of the FIFO in the router C is 50 packets. Under this configuration, the one-way link propagation delay from $S1, S2, \ldots,$ or $S10$, to R is 35 $(10 + 25)$ ms, and the maximum queuing delay in the bottleneck router C can be as large as 60 ms. (This is because the transmission time of a 1500-byte packet on a 10 Mbps link is 1.2 ms and up to 50 packets may be queued in the bottleneck router's FIFO.) In the TCP case, five TCP connections are contending for the bottleneck link's bandwidth. In the UDTCP case, five UDTCP connections are contending for the bottleneck link's bandwidth. The sending application programs of these five TCP/UDTCP connections are on $S1, S2, \ldots, S5$, respectively, and the receiving application programs are all on R. The socket send buffer size of these TCP connections is set to 32 Kb so that each TCP connection can fully utilize its share of available bandwidth.

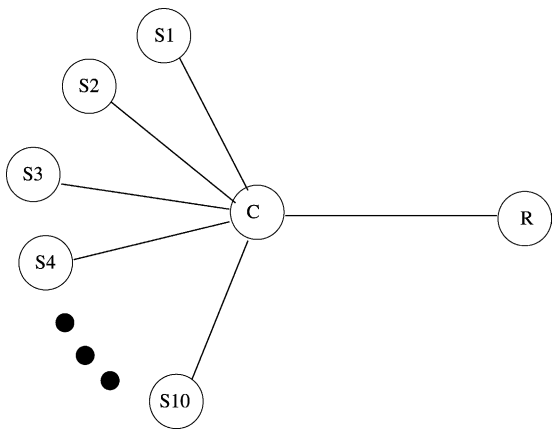

Fig. 4. The testbed network used in simulations.

Configuration 2. The delay of the link (C, R) is 50 ms, the bandwidth of the bottleneck link (C, R) is 20 Mbps, and the maximum queue length of the FIFO in the router C is 100 packets. Under this configuration, the one-way link propagation delay from $S1, S2, \ldots,$ or $S10$, to R is 60 $(10 + 50)$ ms and the maximum queuing delay in the bottleneck router C can be as large as 60 ms. (This is because the transmission time of a 1500-byte packet on a 20 Mbps link is 0.6 ms and up to 100 packets may be queued in the bottleneck router's FIFO.) In the TCP case, ten TCP connections are contending for the bottleneck link's bandwidth. In the UDTCP case, ten UDTCP connections are contending for the bottleneck link's bandwidth. The sending application programs of these ten TCP/UDTCP connections are on $S1, S2, \ldots, S10$, respectively, and the receiving application programs are all on R. The socket send buffer size of these TCP connections is set to 64 Kb so that each TCP connection can fully utilize its share of available bandwidth. description

To easily compare the results gathered on configuration 1 with those gathered on configuration 2, the above settings for configuration 1 and 2 are particularly chosen such that each TCP and UDTCP connection has a fair share of 2 Mbps bandwidth in both configurations.

Because in either the TCP or UDTCP case all connections are homogeneous, we chose to study only the connection between S1 and R. The performance studied is the distribution of the delays of the transmitted messages. The delay of a transmitted message is defined as the elapsed time between two events. The first event occurs when the sending application program successfully writes the message into the socket send buffer, and the second event occurs when the receiving application program reads the message out of the socket receive buffer. To calculate the elapsed time, each time when the sending application program tries to write a message into the socket send buffer, it puts the current timestamp into the message. When the receiving application program reads a message out of the socket receive buffer, it retrieves the timestamp carried in the message and deducts it from the current time to obtain the elapsed time. This elapsed time represents the application layer-to-application layer delay experienced by the message just read out.

The results of configuration 1 are depicted in Figs. 5–7. Fig. 5 shows the delay distribution of the transmitted messages when they are transported on a TCP connection. Fig. 6 shows the delay distribution when a UDTCP connection is used. Fig. 7 shows the percentage of

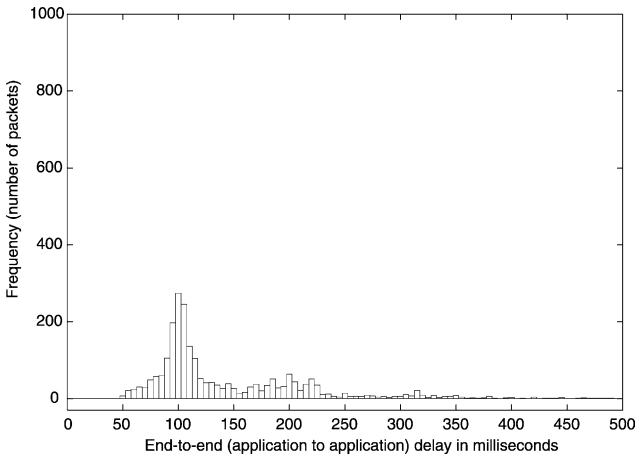

Fig. 5. Case 1: The delay distribution of the messages transported on a TCP connection.

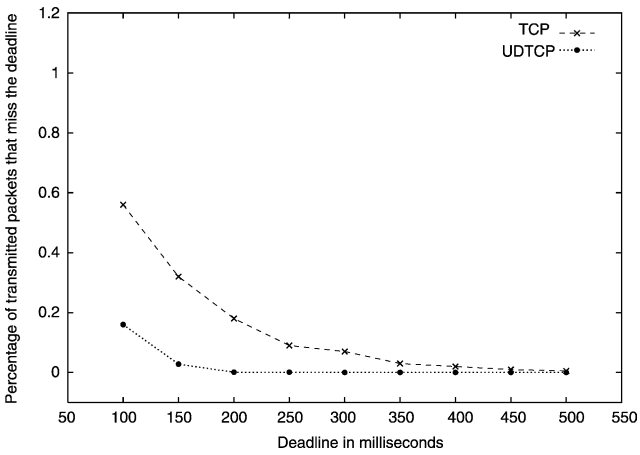

Fig. 7. Case 1: The percentage of transmitted messages that miss the deadline when transported on a TCP and UDTCP connection, respectively.

transmitted messages that miss a specified deadline when the deadline is varied from 100 to 500 ms. From these results, we see that the UDTCP socket provides a much better delay and delay-jitter performance than the TCP socket.

The results of configuration 2 are depicted in Figs. 8–10. In configuration 2, the round-trip link propagation delay ($60 \times 2 = 120$ ms) is larger than that ($35 \times 2 = 70$ ms) in configuration 1. We used this configuration to study the delay performance of the TCP and UDTCP sockets in networks with larger end-to-end link propagation delays (e.g. the Internet). Fig. 8 shows the delay distribution of transmitted messages when they are transported on a TCP connection. Fig. 9 shows the delay distribution when a UDTCP connection is used. Fig. 10 shows the percentage of transmitted messages that miss a specified deadline when the deadline is varied from 100 to 500 ms. Again, we see that the UDTCP socket provides a much better delay and delay-jitter performance than the TCP socket.

By comparing Fig. 7 with Fig. 10, we see that as the RTT of a multimedia packet stream increases, TCP becomes less and less suitable for transporting multimedia streaming application data while UDTCP is still very suitable. This phenomenon is reasonable and can be explained as follows. When the RTT becomes larger, arrived out-of-order messages need to wait longer (at least one RTT) in the TCP reassembly queue before the lost message can be retransmitted and finally arrive. This will increase the chance that these messages miss the specified deadline. In contrast, if messages are transported on the UDTCP connection, because arrived out-of-order messages need not wait at least one RTT in the reassembly queue, increasing RTT causes little impact on the delay performance.

Note that the fact that UDTCP intends not to provide a mandatory reliable service does not mean that it cannot provide a reliable service. Actually, it just leaves the retransmission decisions up to the streaming application, which is the best place to make such decisions. If

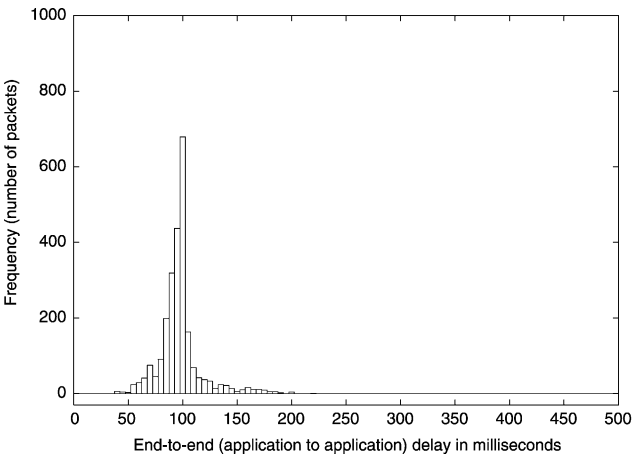

Fig. 6. Case 1: The delay distribution of the messages transported on a UDTCP connection.

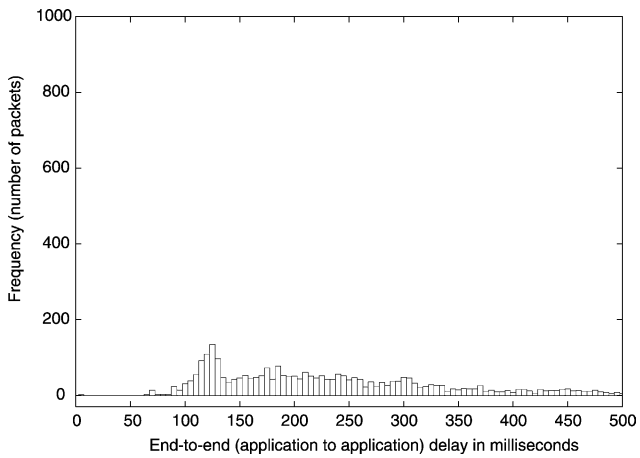

Fig. 8. Case 2: The delay distribution of the messages transported on a TCP connection.

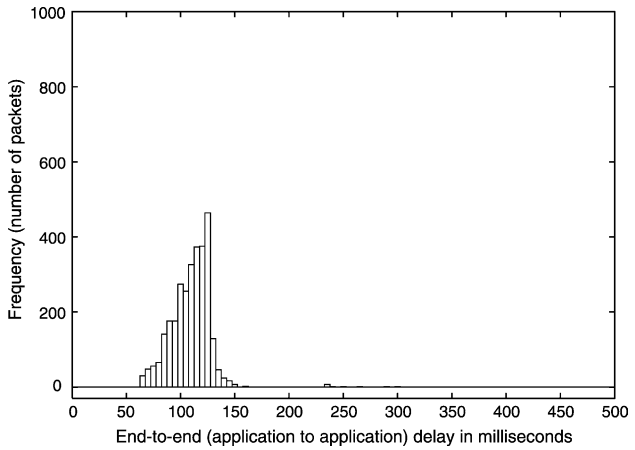

Fig. 9. Case 2: The delay distribution of the messages transported on a UDTCP connection.

the streaming application at the receiving side considers that some missing messages are very important and deserve retransmission, it can send a request packet back to the streaming application running at the sending side. Upon receiving such a request, the sending streaming application can write the requested message into the UDTCP socket again to retransmit it.

In our simulation studies, the sending streaming application does not retransmit lost packets. From the application's perspectives, these lost packets can be considered as packets that miss their playback deadlines. For this reason, to make fair performance comparisons between UDTCP and TCP, we conducted simulation runs to obtain the packet loss rates in the UDTCP cases.

For each configuration, we conducted five simulation runs and averaged their packet loss rates. The averaged packet loss rates for configuration 1 is found to be about 0.005 while that for configuration 2 is found to be about 0.002. We can see that these numbers are insignificant

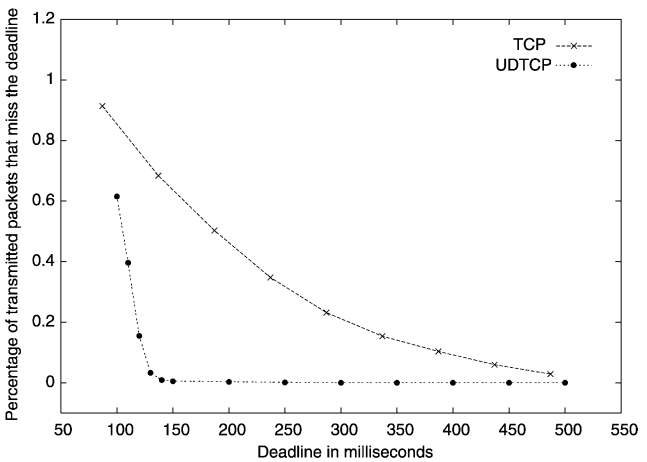

Fig. 10. Case 2: The percentage of transmitted messages that miss the deadline when transported on a TCP and UDTCP connection, respectively.

compared to the percentages of packets that miss their playback deadlines reported in Figs. 7 and 10. These results mean that the performance differences between UDTCP and TCP, from application's perspectives, are still almost the same as those reported in Figs. 7 and 10.

7. Extensions

7.1. Handling TCP timeout

When excessive packets are dropped in a TCP sending window, TCP congestion control may time-out for more than one second without transmitting any packet. This long delay may cause the message waiting in the single-message UDTCP socket send buffer to become useless. To solve this problem, we provide a system call that can be used to remove the message waiting in the UDTCP socket send buffer. A multimedia streaming application program can execute this system call if it cannot successfully write a message into the UDTCP socket for a certain period of time (e.g. 200 ms). This mechanism ensures that every message transmitted into the network is fresh and useful.

7.2. Guaranteeing a minimum rate

Some multimedia streaming applications may want a guarantee of a minimum rate for their audio/video streams while using TCP congestion control to fairly share excess available bandwidth with other traffic flows. This transport service would allow the application to maintain a reasonable performance when the network's bandwidth is insufficient for all traffic flows, while enabling the application to achieve a better performance when the network's bandwidth is abundant.

Although this transport service is no longer 100% TCP friendly, if multimedia streaming applications prefer to use it, this transport service can be easily supported in the UDTCP socket. In the kernel of the sending machine, we can set up a leaky bucket [30] to transmit messages at a desired rate. The leaky bucket is simply a timer which, when triggered, dequeues a message from the UDTCP socket send buffer and transmits it into the network. When the leaky bucket is not triggered, the message in the socket send buffer is transmitted into the network under TCP congestion control.

8. Future work

In the study, we did not compare the performances of an application that uses a UDTCP socket to send out its data with the performances of an application that uses a UDP socket and a fixed rate to send out its data (e.g. Microsoft Inc's 'Media Player' product). We feel that it does not make much sense to make such a comparison because the UDTCP

socket approach is congestion-responsive while the UDP socket and fixed-rate approach is not. For example, in the tested configurations, if the fixed-rate applications are each given a rate greater than its fair share of 2 Mbps, an excessive amount of their packets will be dropped in the bottleneck router C due to excessive FIFO overflows. On the other hand, if they are each given a rate less than 2 Mbps, none of their packets will be dropped due to FIFO overflow and these packets will not experience any queueing delay in the FIFO (because the FIFO is always empty).

A more reasonable comparison would be to compare the performances of an application that uses a UDTCP socket to send out its data with the performances of an application that uses a 'TCP-friendly' approach to send out its data. Although such a comparison would be more desirable, choosing which proposed 'TCP-friendly' approach to compare is not clear (e.g., which is most typical? which is most 'TCP-friendly'?). Also, porting the implementation of a complicated 'TCP-friendly' approach to make it work in our simulator is a large amount of work. For these reasons, we left such a comparison as our future work.

9. Conclusions

We create a new type of socket, called the 'UDTCP socket,' in the BSD UNIX kernel for transporting delay-sensitive but error-tolerant streaming data. The UDTCP socket uses the default in-kernel TCP congestion control implementation to transport a stream of data. Therefore, the transported stream of data is 100% TCP-friendly under any network condition and causes no harm to the stability of a network. Because the UDTCP socket is implemented in the kernel, an application program can readily use it without implementing a complicated 'TCP-like' or 'TCP-friendly' protocol scheme inside its own program.

We propose a novel and simple implementation for the UDTCP socket. We successfully identify only four places in the TCP socket implementation that need to be changed to convert a TCP socket to a UDTCP socket. In addition, all of these four changes are minor. Only 43 lines of C statements need to be added, modified, or deleted to convert the TCP socket implementation into the UDTCP socket implementation.

The UDTCP socket is suitable for delay-sensitive but error-tolerant streaming applications. It has both the UDP and TCP socket properties that are suitable for these applications but not those UDP and TCP socket properties that are unsuitable for these applications.

Acknowledgements

We would like to thank the anonymous reviewers for their valuable comments. This research was supported in part by MOE Program for promoting Academic Excellence of Universities under the grant number 89-E-FA04-1-4 and 91-E-FA06-4-4, the Lee and MTI Center for Networking Research, NCTU, and the Institute of Applied Science and Engineering Research, Academia Sinica, Taiwan.

References

[1] E. Altman, K. Avrachenkov, C. Barakat, Stochastic model of tcp/ip with stationary random losses, ACM SIGCOMM' 2000, Sweden (2000).
[2] J.-C. Bolot, V. Garcia, Control mechanismsfor packet audio in the internet, IEEE INFOCOM'96, San Francisco, CA (1996).
[3] J-C. Bolot, T. Turletti, Experience with control mechanisms for packet video in the internet, ACM Computer Communication Review 8 (1) (1998).
[4] B. Braden, D. Clark, J. Crowcroft, B. Davie, S. Deering, D. Estrin, S. Floyd, V. Jacobson, G. Minshall, C. Partridge, L. Peterson, K.K. Ramakrishnan, S. Shenker, J. Wroclawski, L. Zhang, Recommendations on queue management and congestion avoidance in the internet, RFC 2309, April (1998).
[5] D. Bansal, H. Balakrishnan, TCP-friendly congestioncontrol for streaming internet applications, INFOCOM'2001, Anchorage (2001).
[6] H. Balakrishnan, H. Rahul, An integrated congestion management architecture for internet hosts, ACM SIGCOMM'99 (1999).
[7] D.M. Chiu, R. Jain, Analysis of the increase and decrease algorihms for congestion avoidance in computer networks, Computer Networks and ISDN Systems 17 (1989) 1–14.
[8] S. Floyd, K. Fall, Promoting the Use of End-to-End Congestion Control in the Internet, IEEE/ACM Transactions on Networking, August (1999).
[9] S. Floyd, M. Handley, J. Padhye, J. Widmer, Equation-Based Congestion Control for Unicast Applications, ACM SIGCOMM'2000, Sweden (2000).
[10] G. Hasegawa, M. Murata, H. Miyahara, Fairness and Stability of Congestion Control Mechanism of TCP, IEEE INFOCOM'99 (1999).
[11] V. Jacobson, Congestion Avoidance and Control, ACMSIGCOMM'88 (1988) 314–329.
[12] K. Jeffay, Adaptive, best-effort delivery of digital audio and video across packet-switched networks, Network and Operating System Support for Digital Audio and Video (NOSSDAV) (1992).
[13] H. Kanakia, P. Mishra, A. Reibman, An adaptive congestive control scheme for real-time packet video transport, ACM SIGCOMM'93, San Francisco, CA, September (1993).
[14] G. Lawton, Video streams into the mainstream, IEEE Computer Magazine, July (2000).
[15] J. Mahdavi, S. Floyd, TCP-Friendly Unicast Rate-Based Flow Control, the end2end-interest mailing list, January 8, 1997.
[16] M. Mathis, J. Semke, J. Mahdavi, T. Ott, The macroscopic behavior of the tcp congestion avoidance algorithm, Computer Communication Review 27 (3) (1997).
[17] V. Misra, W. Gong, D. Towsley, stochastic differential equation modeling and analysis of TCP window size behavior, Proceedings Performance'99 (Istanbul, Turkey, October) (1999).
[18] T. Ott, J. Kemperman, M. Mathis, The stationary behavior of ideal TCP congestion avoidance, available at ftp://ftp.bellcore.com/pub/tjo/TCPwindow.ps
[19] A. Ortega, M. Khansari, Rate Control forVideo Coding over Variable Bit Rate Channels with Application to Wireless Transmission, ICIP'95, Washington DC, October (1995).
[20] J. Padhye, V. Firoiu, D. Towsley, J. Kurose, Modeling tcp throughput: a simple model and its empirical validation, ACM SIGCOMM'98 (1998).

[21] J. Padhye, J. Kurose, D. Towsley, R. Koodli, A Model Based TCP-Friendly Rate Control Protocol, Network and Operating System Support for Digital Audio and Video (NOSSDAV), June (1999).
[22] L. Rizzo, Pgmcc: a TCP-friendly single-rate multicast congestion control scheme, ACM SIGCOMM'2000 (2000).
[23] R. Rejaie, M. Handley, D. Estrin, Quality adaption for congestion controlled video playback overthe internet, ACM SIGCOMM'99 (1999).
[24] R. Rejaie, M. Handley, D. Estrin, Architecture considerations for playback of quality adaptive videoover the internet, IEEE ICON'2000 (2000).
[25] R. Rejaie, M. Handley, D. Estrin, RAP: an end-to-end rate-based congestion control mechanism for realtime streams in the internet, IEEE INFOCOM'99 (1999).
[26] H. Schulzrinne, S. Casner, R. Frederick, V. Jacobson, RTP: a Transport Protocol for Real-Time Applications, RFC 1889, 1889.
[27] I. Rhee, V. Ozdemir, Y. Yi, TEAR: TCP emulation at receivers-flow control for multimedai streaming, Technical Report, North Carolina State University, April (2000).
[28] W.R. Stevens, TCP slow start, congestion avoidance, fast retransmission, and fast recovery algorithms, RFC, 2001, January (1997).
[29] D. Sisalem, H. Schulzrinne, The loss-delay adjustment algorithm: a TCP-friendly adaptation scheme, Network and Operating System Support for Digital Audio and Video (NOSSDAV),Cambridge, UK, July 8–10 (1998).
[30] J.S. Turner, New directions in communications, IEEE Communications 24 (10) (1986) 8–15.
[31] T. Turletti, C. Huitema, IVS videoconferencing in the internet, IEEE/ACM Transaction on Networking 4 (3) (1996).
[32] L. Vicisano, L. Rizzo, J. Crowcroft, TCP-like congestion control for layered multicast data transfer, IEEE INFOCOM'98 (1998).
[33] S.Y. Wang, C.L. Chou, C.H. Huang, C.C. Hwang, Z.M. Yang,C, C. Chiou, C.C. Lin, The design and Implementation of the NCTUns1.0 network simulator, Computer Networks 42 (2) (2003) 175–197. (available at http://NSL.csie.nctu.edu.tw/nctuns.html).
[34] Y. Yang, M. Kim, S. Lam, Transient behaviors of TCP-friendly congestion control protocols, INFOCOM'2001, Anchorage, April 22–26 (2001).
[35] Y. Yang, S. Lam, General AIMD Congestion Control, ICNP'2000, Osaka, Japan, November 14–17 (2000).