Implementation of the Multi-Threading Support Framework of A User-level Tranport Mechanism

Yong Lai
Computer Science Department
University of North Dakota
Grand Forks, ND 58202-9015
yong.lai@und.nodak.edu

Suhas Lande
Computer Science Department
University of North Dakota
Grand Forks, ND 58202-9015
suhaslande@yahoo.com

Jun Liu
Computer Science Department
University of North Dakota
Grand Forks, ND 58202-9015
jliu@cs.und.edu

Abstract

Application-controlled congestion control and feedback serve to support latency-sensitive streaming media applications. Moving parts of the protocol processing into user space is one of the promising techniques for providing applications with the flexibility of network control. This paper presents a user-level implementation of the transport functionality to support real-time streaming media applications. This transport functionality consists of a supportive framework and a selective retransmission method. The supportive framework facilitates the interaction between a real-time streaming media application and the user-level implementation of a transport protocol stack. The selective retransmission method allows an application to control the transmission of packets. The prominent feature of this user-level transport functionality is its ability of enabling real-time streaming media applications to control the transmission of data packets.

1 Introduction

The implementation of protocol stacks has been traditionally structured into a monolithic software which either executes in the kernel of an operation system or in a single trusted user-level server. In this monolithic structure, all protocol stacks execute in a single address space. This organization primarily favors security and performance concerns because most internal per-connection information is hidden from the view of applications [1]. The downside of a monolithic design is the inflexibility and the lack of controls by user-level applications. The monolithic implementation of protocol stacks can be viewed as an autonomous system which only provides very limited control capabilities to the applications. In particular, once a data segment is submitted to the protocol stack for transmission, an application generally loses the control over this data segment.

It has become increasingly important to make the protocol stacks to possess two abilities: multiplicity of protocols, and offering applications with better control ability [2]. Multiplicity of protocols is that multiple implementations of protocol stacks co-exist in a system to cope with the diversified transmission needs. For example, latency-critical applications favor low latencies over high throughput. In systems that support both throughput-intensive and latency-critical applications, it is realistic to expect both types of implementations to co-exist. Allowing applications to integrate their transmission needs into protocol processing provides applications with higher efficiency and greater flexibility in protocol cost management. For example, real-time streaming media applications can greatly benefit from customized retransmission protocols which reflect the real-time demands [3]. Accommodating application-specific transmission needs serves to efficiently couple the transmission and the application [1].

Moving parts of the protocol processing into user space is one of the promising techniques for accommodating applications' transmission needs. In a user-level networking system, application-specific customization of a networking stack enables application-controlled flow control and feedback [4]. In contrast to the kernel implementations of legacy protocols, user-level networking does not need to rely on general-purpose kernel components. The kernel can be completely or partially removed from the critical path of the user-level networking system. Moreover, a user-level networking system eases prototyping, debugging, and maintaining new applications.

This paper presents the user-level implementation of a transport functionality which supports real-time streaming media applications. This transport functionality consists of a supportive framework and a selective retransmission method. The supportive framework provides the basic support for enabling the application's control on packet transmission. The selective retransmission method is used together with the supportive framework to allow a real-time streaming media application to dynamically decide the on-demand retransmission of packets. The supportive framework consists of the user-level implementation of a transport protocol, a real-time streaming media application emulator, and an interface mechanism between the former two components. The real-time streaming media application emulator consists of two key components: a stream emulator and a stream controller. The stream emulator generates real-time streaming media flows, and the stream controller performs controls on the transmission of packets. The transport protocol stack

performs unreliable transmission of data packets. The interface mechanism integrates the application-specific transmission preference to the transport protocol stack. The selective retransmission method is adopted by the stream controller to control the transmissions of packets based on the application's preference.

The prominent feature of this user-level transport functionality is enabling real-time streaming applications to control the transmission of data packets. In real-time streaming applications, data packets are typically valid for a short time period. Continued transmission of a data packet beyond its deadline does not benefit the functioning of a real-time streaming application. Actually, transmitting expired data packets unnecessarily consumes network bandwidth and the processing power of a protocol stack. Meanwhile, real-time streaming applications can also prioritize the transmission of data packets. Hence, it is useful to make a real-time streaming application to dynamically determine the transmission of data packets based on deadlines and/or priorities. Based on the status of transmission, a real-time streaming application emulator can dynamically cancel the transmission of a packet, or update the content of a packet that has not been acknowledged.

Selective reliability is another feature of this user-level transport functionality. Although both reliability and low latency are desirable for applications relying on real-time streaming media, they are contradicting to each other. Reliability results in unstable latency of delivery because some packets have to be retransmitted several times before they are successfully delivered. Maintaining a low latency makes it difficult to reliably transmit all packets. In order to achieve a low latency, retransmission of packets has to be repressed. Selective retransmission allows a portion of the data packets to be reliably transmitted. The selection of packets can be performed dynamically to reflect the need of the real-time streaming media application.

In the rest of this paper, the related work is presented in Section 2. The supportive framework is described in Section 3. The selective retransmission method is described in Section 4. Our work is summarized in Section 5.

2 Related Work

The real-time transport protocol (RTP) [5] provides end-to-end delivery services for data with real-time characteristics, such as interactive audio and video. RTP is an application-level protocol which defines a standardized packet format for delivering audio and video over the Internet. RTP itself does not provide any mechanism to ensure timely delivery or provide other quality-of-service guarantees, but relies on lower-layer services to do so. Namely, RTP only enables a real-time streaming application to form and control the packet streams at the application level. However, RTP can not match the transport properties with the intent of an application. One solution to address the need of matching is to use a configurable transport protocol (such as CTP [6]). A configurable transport protocol (in kernel space) supports the construction of customized protocols from collections of software modules, each of which implements a given transport property.

User-level networking infrastructure serves to improve network performance and to provide flexibility to applications, especially for the latency-sensitive streaming media applications.

A number of user-level designs have been proposed in previous work.

`icTCP` [7] is an "information and control" TCP implementation that exposes key pieces of internal TCP state and allows certain TCP variables to be set in a safe fashion. The primary benefit of `icTCP` is that it enables a variety of TCP extensions to be implemented at user-level while ensuring that extensions are TCP-friendly. By exposing information and safe control of the TCP congestion window, TCP protocols can be implemented at the user level.

Alpine [8] is a user-level networking infrastructure for network protocol development. Alpine supports a FreeBSD networking stack on top of a Unix operating system. Alpine virtualizes the FreeBSD networking stack in user space without compromising the compatibility among kernel, networking stack, and applications.

The Network Protocol Server (NPS) [9] is a real-time network system which is implemented on Real-Time Mach for supporting real-time communication. NPS provides a framework for implementing real-time network protocols which require to bound protocol processing time. The key feature of NPS is that the processing time of the network system is predictable because the priority of the network system is under good control in NPS.

The U-Net communication architecture [4] provides parallel and distributed applications with a virtual view of a network interface to enable user-level access to high-speed communication devices. The U-Net architecture removes the kernel from the communication path and allows protocols to be constructed at the user level.

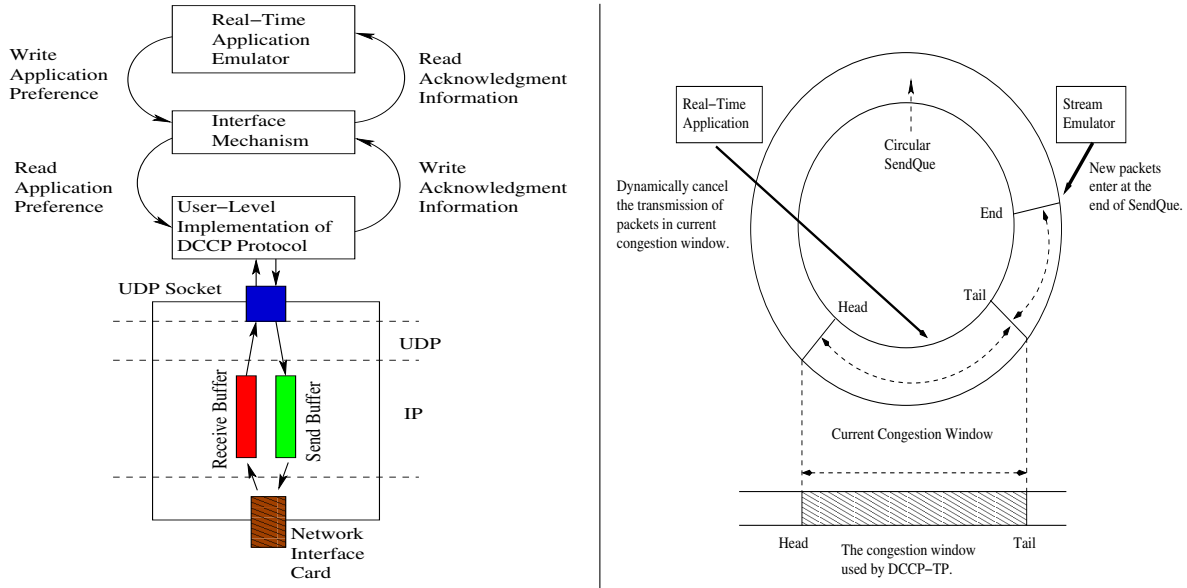
An experimental user-level implementation of TCP [10] is TCP/IP stack with TCP in user space and IP in kernel space. Placing TCP in user space enables a flexible transport protocol which allows integrated communication subsystems.

Jetstream Gbit/s LAN [11] is a set of user-space protocols which provide applications with a flexible low-level network interface to delivers delay-sensitive traffic. Jetstream frames contain a channel identifier so that the network driver can immediately associate an incoming frame with its application. Applications are enabled to delegate the processing of their data to the network interface without the need to first move the data into the application's address space.

Selective reliability provides an application with the flexibility of providing reliability to selected packets while constraining the transmission latency. The Image Transport Protocol (ITP) [12] is designed for image transmission over loss-prone congested or wireless networks. ITP improves user-perceived latency and achieves better interactive performance at the receiver. ITP runs over UDP and incorporates receiver-driven selective reliability to adapt to network congestion. ITP is a generic selectively reliable unicast transport protocol with congestion control that can be customized for specific applications and formats. Compared to the traditional but overly restrictive approach of transporting images using TCP, ITP allows a receiver application to improve the interactivity and responsiveness of reconstructed images.

3 The Supportive Framework

This supportive framework aims to construct a user-level network buffering in order to allow an application to achieve a controllable tradeoff between reliability and transmission latency. User-level network buffering is one of the important factors leading to the improvement of the performance of I/O-intensive applications [13]. However, this supportive framework does not provide an application with the direct access to the network I/O device. The supportive framework follows a client/server architecture by making a stream source and a sink to act as a client and a server, respectively. The framework consists of a user-level implementation of the DCCP protocol, an interface mechanism, and a real-time streaming application emulator. The real-time streaming application emulator has two key components: a stream emulator which handles data streams, and a controller which controls the transmission of the unacknowledged packets. The interface mechanism is shared between the real-time streaming media application emulator and the user-level implementation of the DCCP transport protocol for enabling the interaction between the application and the transport protocol stack. A schematic diagram of the supportive framework is shown in Figure 1 (a).



(a) Architecture of the transport functionality. (b) The interface mechanism.

Figure 1: The schematic diagram of the user-level transport functionality.

3.1 The Client/Server Architecture

The real-time streaming media application emulator runs on both the source and the sink of a real-time streaming media flow. Data packets are generated in real-time at the source of a flow and are sent to the sink for processing. `Iperf` [14] is used to establish a client/server platform for prototyping the real-time streaming media application emulator running on both the source and the sink sides. `Iperf` was designed to deliver a sequence of packets

from a source to a sink primarily for network measurement purposes, especially measuring the link bandwidth. We have customized `Iperf` to make it to rely on the transport service provided by the user-level transport protocol stack. Based on the existing structure of `Iperf`, the source of of a real-time streaming media flow is treated as the client, and the sink which consumes the flow is treated as the server.

3.2 The User-Level Implementation of the DCCP Transport Protocol

The transport functionality adopts the Datagram Congestion Control Protocol (DCCP)) [15] as the transport protocol. DCCP facilitates a real-time streaming media application to control the trade-offs between maintaining a low delay and the reliable in-order delivery. The user-level implementation of DCCP, the DCCP/TP [16] package, is used toward the user-level transport protocol stack in the transport functionality. DCCP/TP is a fresh-start implementation of DCCP and is optimized for portability. DCCP/TP supports many DCCP features and implements reliable connection setup, tear down, and feature negotiation.

DCCP/TP emulates the socket support which is traditionally implemented in the kernel of an operating system. DCCP/TP opens a regular UDP socket through which all DCCP packets are sent to/received from the kernel. DCCP/TP relies on the UDP/IP protocol stack that is implemented in the kernel. DCCP/TP generates and maintains sockets for DCCP connections. No matter how many DCCP connections are opened, the single UDP socket delivers all packets for the DCCP connections.

3.3 The Stream Emulator

The stream emulator simulates the arrivals of data segments and feeds the data segments to the DCCP/TP stack for transmission. The stream emulator can either play back stored stream media or generate on-the-fly streams. When playing back a stored stream media, the stream emulator feeds the data segments according to the arrival times that are specified in the trace file of the stream. A real-time clock is used by the stream emulator to generate data packets at the specified times. The trace of a stored stream is required to specify the arrival times of data segments in a strictly ascending order. Since precise timing may require special hardware support, the emulator approximates the precise timestamps in order to enhance portability. The low-cost real-time enabled operating system kernel often generates interrupt signals at constant frequency, *e.g.*, the real-time kernel patched from Linux [17]. Hence, the precise time spacing between two consecutive packets is approximated by the closest time span of a number of consecutive interrupt signals. When emulating an on-the-fly stream, the stream emulator generates data segments according to a prescribed algorithm.

3.4 The Transmission Controller

The transmission controller is another key component of a real-time streaming media application emulator, which enables the emulator to have control on the transmission of packets.

The emulator can perform two types of controls: cancelling the transmission of an unacknowledged data packet, or updating the content of an unacknowledged data packet. The transmission controller can stop the continued transmission of an unacknowledged data segment. The controller put a cancellation mark in the temporary buffering that is used by the user-level transport protocol stack for holding the unacknowledged data segments. The transport protocol stack can then stop the continued transmission of the corresponding unacknowledged data segment and removes it from the buffering. The controller can also update the content of an unacknowledged data segment when a real-time streaming media application emulator would like dynamically decide to make new content to be transmitted.

3.5 The Interface Mechanism

The real-time streaming media application emulator accesses the interface mechanism, called `SendQue`, through a set of APIs that are provided by the `SendQue` library. The interface mechanism is deployed between the real-time streaming media application emulator and DCCP/TP, which allows the emulator to have limited control on the transmission of packets. The interface mechanism also serves as the user-level network buffering which is similar to the kernel-space packet buffering used in TCP. In a monolithic network protocol stack, packets are first queued in the kernel-space packet buffer at the transport layer after they are passed into the kernel. Afterwards, packets are transmitted under the control of the transport protocol in use. Due to this reason, an application loses the control on transmitting packets after submitting packets to the kernel through a BSD socket.

In our user-level transport functionality, the interface mechanism holds the not-yet-acknowledged packets in the user space. The key component of this mechanism is a circular packet queue, called `SendQue`, which is the packet buffer in the transport protocol stack in a monolithic kernel. Even though the packet buffer is maintained in user space by DCCP/TP, a real-time streaming media application still can not access this packet buffer because DCCP/TP and the real-time streaming media application are encapsulated in different processes or threads. As shown in Figure 1 (b), the interface mechanism acts as a mirror of the packets that are under the control of DCCP/TP. The interface mechanism is maintained by an independent process/thread and allows the access by both DCCP/TP and the real-time streaming media application emulator. This interface mechanism can facilitate the implementation of a new control algorithm without the need of configuration. Configurations are typically needed in previous designs of user-level congestion control mechanisms, *e.g.*, in [18]. The interface mechanism has the following features.

3.5.1 Exchanging the status of packet transmission

A circular packet queue is used in the interface mechanism and serves as the shared medium between DCCP/TP and the real-time streaming media application emulator. Through this circular queue, DCCP/TP can pass the status of the transmission of packets to the emulator. The stream emulator also buffer the newly generated packets at the end of the circular queue. The real-time streaming media application emulator also passes its control decisions to DCCP/TP through expressing the control decisions on the records of data segments that

are buffered in the circular queue. The record of a data segment in the circular queue includes the following descriptive information.

- *An application-specific identifier*: It is used to label a data segment in the circular queue. The transmission controller distinguishes data segments by their application-specific identifiers.
- *A timestamp when a data segment is generated by the stream emulator and the deadline of the data segment*: They are used to keep track of the expiration of a data segment.
- *The status of the transmission of a data segment and a time-stamp when the segment was last attempted to transmit by DCCP/TP*: They are used by DCCP/TP to express to the application emulator about the status of transmission. The transmission status of a data segment is one of *not yet transmitted*, *transmitted but not acknowledged*, and *successfully acknowledged*.
- *The size of a data segment and the pointer to the memory holding the content of the data segment*: They are used for allowing the application emulator to dynamically update the content of a data segment whose transmission is still not completed. DCCP/TP generates a DCCP packet structure with the data body portion of the packet being left empty. The size information of a data segment is used by DCCP/TP to reserve a memory space for filling the content of a data segment immediately before the data segment is physically transmitted. *Late binding* the content of a data segment facilitates the application emulator to send the up-to-date content of a data segment.

Depending on the size of the current congestion window, DCCP/TP gradually copies the descriptive information of data segments from `SendQue` into its own congestion window. Once a data segment's information is copied by DCCP/TP, it is eligible to be transmitted by DCCP/TP. Unlike passing a data segment into the kernel space in a monolithic network stack, the content of a data segment is not passed to DCCP/TP. DCCP/TP will read the data content of a data segment (using the pointer to the content of a data segment) immediately before the data segment is transmitted, *i.e.*, when the packet is passed to the regular UDP socket. Late binding allows a real-time streaming application to update the content of a data segment at any time before the packet is physically transmitted. The only restriction on the application is that the size of a data body should not be increased when updating the data content.

Whenever DCCP/TP updates its congestion window upon receiving an acknowledgment (ACK) packet, it also forwards the updates to `SendQue`. Since DCCP/TP has the ability of performing selective acknowledgment (SACK), an acknowledgment vector can be included in an ACK packet. One ACK package can trigger the update to the transmission status of a number of data segments. Both DCCP/TP's congestion window and `SendQue` move across those packets that have been acknowledged and stop at the next unacknowledged packet.

3.5.2 Accepting the transmission control from an application

A real-time streaming media application emulator can dynamically perform controls on the transmission of packets that have not been successfully acknowledged. The control can be one of the followings.

First, an application can update the content of a data segment as long as this data segment has not been successfully acknowledged. Since DCCP/TP only reads the content of a data segment immediately before transmitting the segment, a data segment is always transmitted with its most recent content. Correspondingly, the sink of a steam can receive multiple non-identical copies of the same data segment. Hence, a new convention has to be agreed between a pair of data source and sink, that is, the sink of a stream always replace the old data content with the most recently received content of a data segment. In the monolithic network stacks, the commonly adopted convention is that only the content of the first receipt of a data segment is kept and all later received copies are discarded.

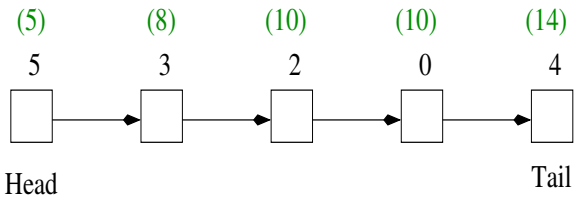
Second, a real-time streaming media application emulator can choose to reliably transmit a set of selective packets by relaxing their deadlines of transmission. Data segments with relaxed deadlines are eligible for more retransmissions than the data segments with regular deadlines, thus, they are more likely to be reliably delivered. By allowing only a set of selective packets to be eligible for more retransmissions, an application can both control the congestion and provide reliability to selective packets.

Third, a real-time streaming media application emulator can choose to cancel the transmission of a packet by artificially mark the state of this packet as *acknowledged*. When DCCP/TP scans and updates the circular queue the next time, DCCP/TP can thus mark this data segment in its own congestion window as acknowledged.

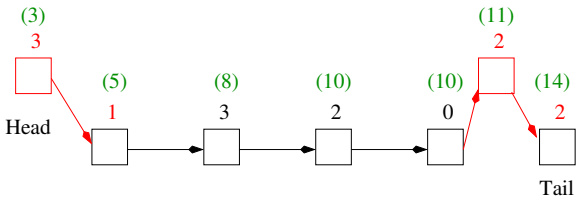
3.6 The Real-Time Clock Timing Subsystem

Since only one real-time clock is available in most of the low-cost real-time operating systems, a real-time clock (RTC) timing subsystem is needed to support the generation of multiple flows of data segments in real-time. The RTC subsystem performs real-time timing for stream emulators. A stream emulator only handles the generation of one real-time stream. A stream emulator submits a timing request to the RTC subsystem and waits for the activation signal from the RTC subsystem upon the requested amount of time has elapsed. Thus, a stream emulator can not submit a next timing request before it receives the activation signal of its current timing request.

The RTC subsystem maintains a linked list which holds the timing requests submitted by the stream emulators. At any time, each stream emulator can only have at most one timing request in the list. Each timing request is inserted into the list based on the waiting time before a stream emulator generates the next data segment. Since the timing services in the low-cost real-time operating systems mostly rely on interrupt signals at a fixed frequency, the time period specified in a timing request is converted into a number of interrupt signals whose time span closely matches the time period requested. The head of the list contains the earliest expiring request among the requests that are currently in the list. The rest timing requests are sorted in the list according to their expiration times.



(a) The list formed from a sequence of timing requests of 10, 8, 5, 10, and 14 interrupts worthy of waiting time.



(b) The list formed after two new timing requests of 3 and 11 interrupts worthy of waiting time are submitted 1 time unit after the arrival of the sequence of requests in (a).

Figure 2: Illustration of the list of timing requests maintained in the RTC subsystem. The integer in parenthesis denotes the number of time units that are requested by a stream emulator. The integer without parenthesis indicates the remaining number of time units before the expiration of a timing request.

In order to reduce the overhead of maintaining the list, each item in the list is only specified with a number of interrupt signals which corresponds to the remaining waiting time after the expiration of the timing request immediately before it. The maintenance operation is illustrated in Fig. 2. When a set of timing requests with their waiting time being specified in number of interrupt signals are assumed to be submitted to the RTC subsystem at the same time, the resulting list is shown in Fig. 2 (a) when the list is assumed empty beforehand. Each request in the list is labeled with the remaining number of interrupt signals after its previous request is expired. Namely, the overall remaining number of interrupt signals before the expiration of a timing request is the sum of the incremental numbers of interrupt signals from the head to the request itself. Upon each clock interrupt signal, the RTC subsystem decrements the number of remaining interrupt signals specified for the current head of the list. When this number becomes 0, the timing request specified in the head is expired. The head of the list can be changed when new timing requests are submitted. Fig. 2 (b) shows the resulting list after the insertions are made for two new timing requests that are submitted in one interrupt worthy of time after the arrival of the set of requests as shown in Fig. 2 (a).

3.7 The Threading System

The above described mechanisms are maintained by a number of threads as shown in Figure 3. The DCCP/TP thread is never blocked (except waiting for network I/O) and drives the entire transport functionality. The `SendQue` thread can block itself when either there is no space in the circular queue to hold new packets or the stream emulator does not have new data packets available. The two major components of a real-time streaming application are encapsulated into separate threads. Theoretically speaking, these two threads always run without being blocked. However, a particular application may decide to make either of the two threads to be blocked whenever it is needed to do so. The `SendQue` thread can be woken up by any of the other three threads whenever any of the wake-up conditions is satisfied.

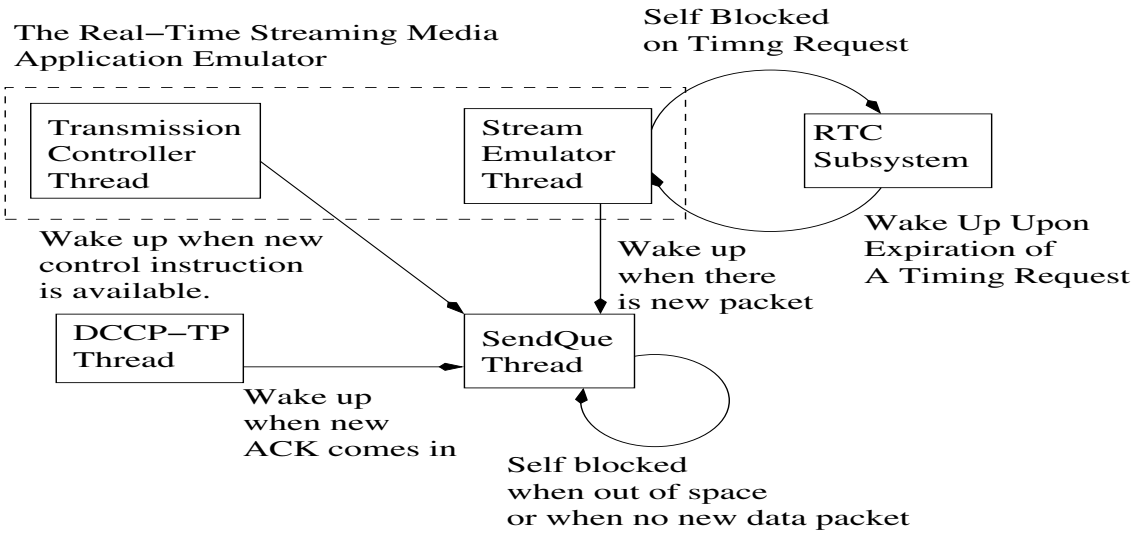


Figure 3: Illustration of the signaling between threads.

4 The Selective Retransmission Support

Sequence numbers are commonly adopted in the reliable transport protocols for keeping track of the transmission status of packets. In a monolithic networking stack, a fixed sequence number is assigned to each packet. The receiver acknowledges the transmission status of packets using the sequence numbers of packets. Sequence numbers are assigned to packets in the same order as the packets are submitted to the network stack. In our user-level transport functionality, a data segment is assigned with a dynamic sequence number. Flexible transmission arrangement is enabled by breaking the fixed association of a sequence to a data segment. Moreover, a packet obtains different sequence numbers in consecutive transmission attempts. The need of assigning sequence numbers on a dynamic basis can be directly benefited from the design of the DCCP protocol.

The DCCP protocol also uses sequence numbers to identify data segments. In our user-level transport functionality, a data segment can be made to bind to different sequence numbers. The packets queued in the congestion window in DCCP/TP are just a sequence of containers with labels (such as the application-specific packet identifiers), and the content of a data packet is only loaded immediately before the actual transmission of the packet. The content of a data packet is looked up using the application-specific identifier. The ability of making a data segment to bind to different sequence numbers on demand enables the selective retransmission of data segments.

A selective retransmission method can be constructed based on different criteria. We describe two different selective retransmission methods: the deadline based method and the priority based method.

4.1 The Deadline-based Selective Retransmission

It is assumed that every packet is associated with a deadline. Continued transmission of a packet beyond its deadline does not contribute to the functioning of a real-time streaming application. DCCP/TP removes the unacknowledged data segments that have past their deadlines from DCCP/TP's congestion window. The sequence numbers that were previously assigned to the expired data segments are assigned to other data segments. Meanwhile, DCCP/TP also reports to `SendQue` that the expired data segments have been given up their transmissions. If a data segment is to be reliably transmitted, then it is granted with a relaxed deadline.

4.2 The Priority-based Selective Retransmission

It is assumed that every packet is associated with a priority. Transmission of packets is prioritized upon the occurrence of congestion. The congestion window reduces its size when a packet loss signal is detected. Fewer packets are eligible to be transmitted under a congestion window with a reduced size. Packets with higher priorities are selected to be covered by the reduced congestion window. Hence, the sequence numbers are dynamically bound to data segments whenever the order of transmission of the highly prioritized data segments is rearranged based on their priorities. In this case, each packet also has to be associated with a grace period during which its transmission can be attempted. When the grace period expires, a packet is given up its transmission. Hence, reliability is selectively given to the packets with higher priorities.

5 Conclusions

This paper described a user-level transport functionality which facilitates real-time streaming media applications to control the tradeoff between reliability and transmission latency. The basic idea of this transport functionality is to allow a real-time streaming media application to gain the ability of controlling the transmission of packets based on the application's preference and the state of network congestion. The key component of the supportive framework is an interface mechanism which enables the interaction between a real-time streaming media application and the user-level implementation of DCCP protocol. By only providing reliable transmission to a selective set of packets, a real-time streaming media application can well control the transmission latency.

References

[1] Mogul, Jeffrey, Brakmo, Lawrence, Lowell, David E., Subhraveti, Dinesh, and Moore, Justin. Unveiling the transport. In *CCR*, 34(1):99–106, New York, NY, USA, January 2004.

[2] Thekkath, Chandramohan A., Nguyen, Thu D., Moy, Evelyn, and Lazowska, Edward D. Implementing network protocols at user level. In *CCR*, 23(4):64–73, 1993.

[3] Watson, Richard W. and Mamrak, Sandy A. Gaining efficiency in transport services by appropriate design and implementation choices. In *ACM Transactions on Computer Systems*, 5(2):97–120, May 1987.

[4] von Eicken, Thorsten, Basu, Anindya, Buch, Vineet, and Vogels, Werner. U-Net: a user-level network interface for parallel and distributed computing. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*, pages 40–53, Copper Mountain, CO, United States, December 1995.

[5] Schulzrinne, Henning, Casner, Stephen, Frederick, Ron, and Jacobson, Von. RTP: A Transport Protocol for Real-Time Applications. http://www.rfc-editor.org/rfc/rfc3550.txt, July 2003. Internet proposed standard RFC 3550.

[6] Bridges, Patrick G., Wong, Gary T., Hiltunen, Matti, Schlichting, Richard D., and Barrick, Matthew J. A configurable and extensible transport protocol. In *IEEE/ACM Transactions on Networking*, 15(6):1254–1265, December 2007.

[7] Gunawi, Haryadi S., Arpaci-Dusseau, Andrea C., and Arpaci-Dusseau, Remzi H. Deploying safe user-level network services with icTCP. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 317–332, San Francisco, CA, United States, December 6-8 2004.

[8] Ely, David, Savage, Stefan, and Wetherall, David. Alpine: A User-Level Infrastructure for Network Protocol Development. In *Proceedings of the USENIX Symposium on Internet Technologies and Systems (USITS)*, pages 171–184, San Francisco, CA, United States, March 26-28 2001.

[9] Nakajima, Tatsuo and Tokuda, Hideyuki. User-level Real-Time Network System on Microkernel-based Operating Systems. In *Real-Time Systems*, 14(1):45–60, Kluwer Academic Publishers, Norwell, MA, USA, January 1998.

[10] Braun, Torsten, Diot, Christophe, Hoglander, Anna, and Roca, Vincent. An Experimental User Level Implementation of TCP. INRIA, Technical Report RR-2650, Sophia Antipolis, France, September 1995.

[11] Edwards, Aled, Watson, Greg, Lumley, John, Banks, David, Calamvokis, Costas, and Dalton, C. User-space protocols deliver high performance to applications on a low-cost Gb/s LAN. In *Proceedings of the ACM International Conference on Communications Architecture and Protocols (SIGCOMM)*, pages 14–23, London, United Kingdom, August 31-sep 2 1994.

[12] Raman, Suchitra, Balakrishnan, Hari, and Srinivasan, Murari. An Image Transport Protocol for the Internet. In *Proceedings of the International Conference on Network Protocols (ICNP)*, pages 209–219, IEEE Computer Society, Osaka, Japan, November 14-17 2000.

[13] Magoutis, Kostas, Seltzer, Margo, and Gabber, Eran. The Case Against User-level Networking. In *Proceedings of the Annual Workshop on System-Area Networks (SAN)*, Madrid, Spain, February 14 2004.

[14] NLANR. Iperf Bandwidth Measurement Toolkit. The Distributed Application Support Team (DAST), May 2005.

[15] Kohler, Eddie, Handley, Mark, and Floyd, Sally. Datagram Congestion Control Protocol (DCCP). http://www.ietf.org/rfc/rfc4340.txt, March 2006. IETF RFC 4340.

[16] Phelan, Tom. DCCP-TP—a fresh-start implementation of the Datagram Congestion Control Protocol (DCCP). http://www.phelan-4.com/dccp-tp/tiki-index.php.

[17] Molnar, Ingo. The Real-Time Linux Wiki and Kernel Patch. http://rt.wiki.kernel.org/index.php/Main_Page.

[18] Gu, Yunhong and Grossman, Robert L. Supporting Configurable Congestion Control in Data Transport Services. In *Proceedings of the ACM/IEEE conference on Supercomputing (SC)*, pages 31–41, Seattle, WA, United States, November 12-18 2005.