TCP over UDP

**Cheng-Han Lee**

Columbia University  
1214 Amsterdam Avenue  
New York, USA 10027

cl2804@columbia.edu

**Salman Abdul Baset**

Columbia University  
1214 Amsterdam Avenue  
New York, USA 10027

sa2086@columbia.edu

**Henning Schulzrinne**

Columbia University  
1214 Amsterdam Avenue  
New York, USA 10027

hgs@cs.columbia.edu

**ABSTRACT**

The intension of the project is to develop a building block to implement an extensive library which can be used in the scenario where users behind network address translators (NATs) are not able to establish TCP connections but are able to establish UDP connections. The main objective of the project is to implement a TCP-over-UDP (ToU) library in C/C++ language that provides socket-like APIs in which users can build connections that have almost the same congestion control, flow control, and connection control mechanism as offered by TCP. In this document, we present background concepts and architectures of implementing ToU library based on the IETF draft TCP-over-UDP [draft-baset-tsvwg-tcp-over-udp-01][1], and describe testing results for simple scenarios of a client and a server that are using ToU library. At the end, we also present our quantitative tests and analyses of ToU in comparison with real TCP Reno results in Ubuntu systems, which we believe indicates that the ToU library is similar to real TCP Reno in terms of congestion window pattern in general network environments.

**Categories and Subject Descriptors**

**D.3.3 [Programming Languages**]: C/C++

**General Terms**

Design, Performance, Reliability

**Keywords**

TCP, UDP, networks, library

# INTRODUCTION

It becomes a problem sometimes when establishing a direct transmission control protocol (TCP) connection between two hosts behind NATs. When both the client and the server are behind different NAT devices, the applications running on hosts may not be able to establish a direct TCP connection with one another. However, with certain NAT types, such as applying ICE-UDP [2], although applications cannot establish TCP connections directly, it is possible for these applications to exchange user datagram protocol (UDP) traffic. In this sense, using UDP is preferable for such applications; nevertheless, applications may require the underlying transport layer to provide reliability, connection control, congestion control, and flow control mechanisms. Unlike TCP, UDP fails to provide these semantics. Therefore, in [1], the TCP-over-UDP, a reliable, congestion control, and flow control transport protocol on top of UDP, is proposed, and in order to achieve these mechanisms, ToU almost uses the same header as TCP does which allows ToU to easily implement TCP's reliability, connection control mechanism, congestion control mechanism and congestion control mechanism.

In this project, we have implemented an efficient socket-like library in C/C++ for programmers to use. In order to make ToU library simple and easy, the APIs of ToU is just like C/C++ socket APIs [3]. We will describe briefly the APIs of ToU in Section 4.1.1. The implementation of ToU is in accordance with the draft of [1] and the algorithms described in RFC documents related to TCP specifications such as [4], [5], [6], [7], [8], and [9]. We will discuss details in Section 4.1.2.

ToU library is an user-space library which includes Boost C++ library [3] for managing threads and timers, and since it is on top of UDP, UDP-related function calls such as *sendto*() and *recvfrom*() are used to carry out the underlying sending and receiving. We will present the ToU transmission model in Section 4.2.1 and 4.2.2, and will also illustrate program internal operations and what each program does in a form of finite state machine (FSM) in Section 4.2.2.

The rest of the paper is organized as follows. In Section 2 and 3, we will show the related works of ToU and its background, respectively. Section 4 shows the architecture and implementation in detail. In Section 5, we will demonstrate that the patterns of congestion window size, average throughput, and performance of ToU are close to real TCP Reno by giving various experiments on well-configured real machines. Our analysis and conclusion are also summarized. The task list will be shown in Section 6.

# RELATED

ToU takes advantage of existing Almost TCP over UDP (atou) implementations [10].

# BACKGROUND

## UDP

UDP uses a simple transmission model without implicit hand-shaking dialogues for guaranteeing reliability, ordering, or data integrity. Thus, UDP provides an unreliable service and datagram may arrive out of order, appear duplicate, or go missing without awareness of hosts. UDP assumes that the error checking and correction is either not necessary or performed in the application, avoiding the overhead of such processing at the network transport level.

## TCP

TCP provides a connection-oriented, reliable, and byte stream service. Firstly, the term connection-oriented means the two applications using TCP must establish a TCP connection before they can exchange data. Secondly, for achieving reliability, TCP assigns a sequence number to each byte transmitted, and expects a positive acknowledgment (ACK) from the receiving TCP. If the ACK is not received within a timeout interval, the data is retransmitted. The receiving TCP uses the sequence numbers to rearrange the segments when they arrive out of order, and to eliminate duplicate segments.  Finally, TCP transfers a continuous stream of bytes. TCP does this by grouping the bytes in TCP segments, which are passed to IP for transmission to the destination. TCP itself decides how to segment the data and it may forward the data at its own convenience. In addition to the properties above, TCP is also a full duplex protocol, meaning that each TCP connection supports a pair of byte streams, one flowing in each direction.

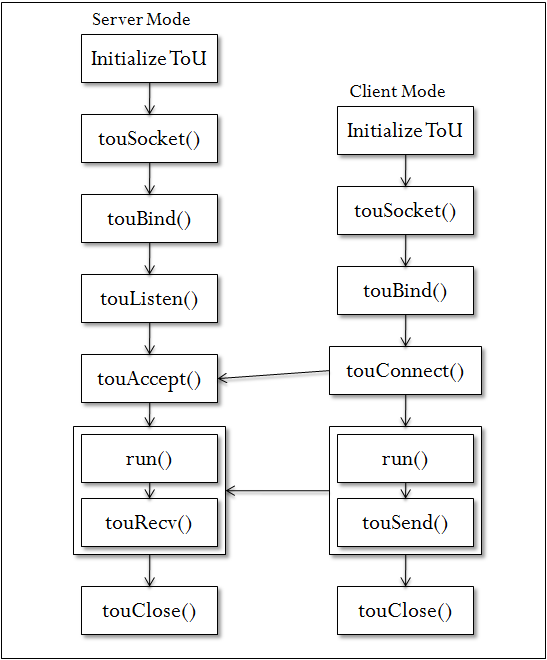
# ARCHITECTURE

In this section, we begin with introducing ToU APIs. We then present several mechanisms ToU used, such as flow control and congestion control. We also explain how ToU works as a user-space library and its relevant underlying network protocols. We then will explain how different types of modules and running threads are worked with ToU library. At the end, we detail how some of the essential data structures and internal operations work in ToU.

## Introduction to ToU

### ToU APIs

**Figure 1. ToU APIs for server/ client mode.**



ToU library is a user-space library that provides socket-like function calls to the applications, including *touSocket(), touAccept(), touConnect(), touBind(), touListen(), touSend(), touRecv(), run(),* and *touClose()*, as shown in Figure 1.

From the application point of view, applications can interact with the ToU library through these calls by including the header file "*tou.h*". The returning values and function parameters of ToU library are like C/C++ Socket APIs [3].

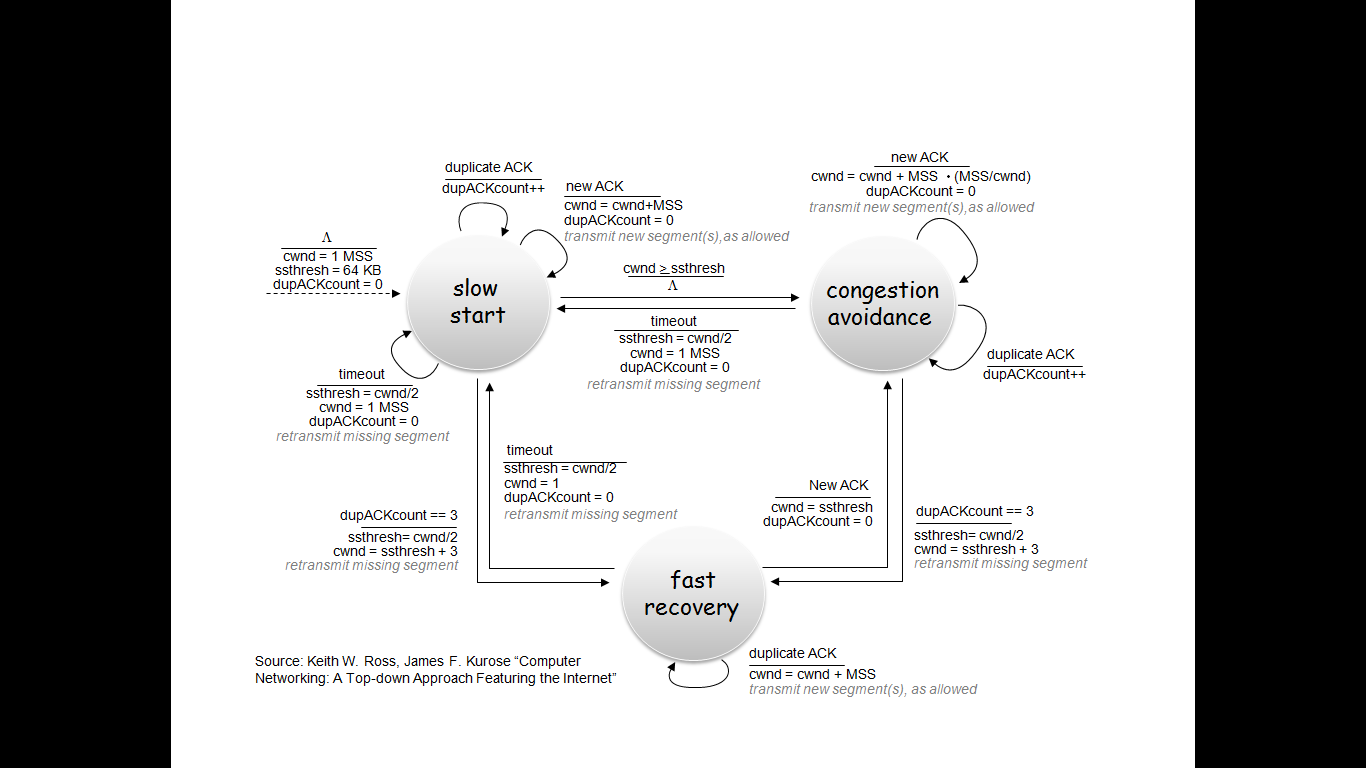
In order to use ToU library, applications must instantiate a ToU object at the beginning of program. It then call *touSocket* to get the file descriptor, *touBind* to bind the port, and use *touListen* to limit the number of connections allowed on the incoming queue or *touConnect* to connect to the peer. The server uses *touAccept* to pick up the connection requests and starts to wait for the data. The client uses *touSend* to send the data and server will receive it by calling *touRecv*. *Run* should be call after Select system call to handle the ToU processing. Finally, *touClose* will be called at the end of transmission to close the connection.

### Flow control and Congestion control

ToU follows the algorithms described in TCP [4], [5], [6], [7], [8] and [9], including flow control mechanism, connection control mechanism, and congestion control mechanism.

In flow control, while a ToU receiver sends an ACK back to sender, the ACK packet also indicates sender the number of bytes a receiver can receive beyond the last received ToU segment, without causing overrun and overflow in its internal buffers.

**Figure 2. ToU Congestion control FSM. [11]**



In congestion control, ToU adopts TCP congestion control [RFC 2581] [7]. That is, ToU will go through the slow start, congestion avoidance, and fast recovery phases as shown in Figure 2 [11], and we can find the implementation which correspondent to Figure 2 in *tou\_ss\_ca.cpp* and *tou\_congestion.h*.

During slow start, a ToU sender starts with a congestion window (cwnd) of two maximum segment size (MSS) bytes, and increments it by at most one MSS bytes for every ACK received that cumulatively acknowledges new data. Specifically, the congestion window size doubles in every round-trip time (RTT), which results in the cwnd glows exponentially. When the cwnd exceeds a threshold (ssthresh), the algorithm switches to congestion avoidance.

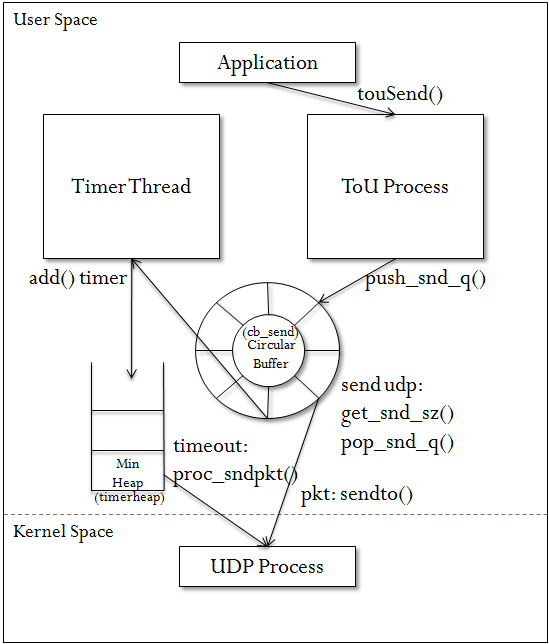
During the phase of congestion avoidance, the congestion window is additively increased by one MSS in every RTT, making cwnd glows polynomially. If timeout occurs, ToU will reduce the size of cwnd to one MSS and return to slow start phase. If receiving three duplicate acknowledgments, ToU will execute fast retransmit algorithm and halve the congestion window size, entering the fast recovery phase.

During fast recovery phase, ToU is following classic TCP Reno standard in contrast to the implementation in Linux systems. In ToU implementation, after the fast retransmit, which is retransmitting the lost packets that was indicated by three duplicate ACKs, ToU is to retransmit one segment for each RTT and transmit new segments while recovering from drops. To be more specific, ToU does this by first halving cwnd after fast retransmit and then incrementing cwnd for each duplicate ACK until it becomes large enough to allow new segments to be transmitted into the network, and ToU will keep doing so until it hits the time of the first drop has been acknowledged. Once receiving the new acknowledgment, ToU will return to congestion avoidance phase. In the meantime, if timeout occurs, it will go back to slow start phase, and reduce the size of cwnd to the value of one MSS.

Nevertheless, the real Linux implementation, such as Ubuntu as we will see in tests in Section 5, is similar to that of ToU used but does not inflate cwnd. Instead of setting cwnd to ssthresh+3, cwnd is gradually reduced by decrementing it by one for every two duplicate ACKs received. This implementation allows the sender to send a new packet for every two duplicate ACKs so as to evenly spread the transmissions and retransmissions over the phase of fast recovery.

Since ToU sticks with RFC documentation, there are slightly differences when we compare Linux TCP implementation results with ToU’s. But the overall trends are still close. We will see the differences and discuss more in Section 5.

Figure 3. ToU transmission model



## Working Modules and Running Threads

The ToU library should interact with application thread along with a supporting timer thread and a close thread. The close thread is instantiated at the near end of the program for connection closure, and the application and timer thread are running at the beginning. Thus, basically, the ToU is a two-thread model library.

The library is fully modularized, implementing the circular buffer for data input/output processing, ToU Process for handling packets that complied with Porcesstou FSM (see Figure 8), timer thread to deal with the retransmission of lost packets when the timers are fired, ToU control block for keeping track of internal states, ToU socket table for managing connections, and logging mechanisms for debugging and recording transit processes.

In the following sections, we are going to show the internal of the ToU implementation and to illustrate various components in terms of transmission model and receiving model, as shown in Figure 3 and 6, respectively. Then we will go through the operations of each component in detail, such as ToU Process (touprocess), Timer Thread (timer), and ToU data structures.

### ToU transmission model

Figure 3 shows the path taken by a single transmission of data from a user-space program using ToU library to a kernel-space UDP process. As has been noted in Section 4.1.1, user programs must instantiate a ToU object first, and then call *touSocket*, *touBind*, and *touConnect* in sequence to set up the connection. When *touConnect* returns successfully, user programs then begin to transmit data.

ToU uses touPkg, see Figure 4, data structure to represent each packet internally. The ToU header format, touHeader, is referring to [1], and we can find its implementation in Figure 4. When an user program writes data to network by calling *touSend()*, *touSend()* invokes *push\_snd\_q()*, trying to place data onto circular buffer (cb\_send) as many as possible and will return number of bytes successfully placed. At this point, data are still queued in the circular buffer and not yet transmitted. ToU Process will invoke *get\_snd\_sz()* to get an amount of size sender can send based on the minimal value of cwnd and advertised window (awnd). If there are data available to send, data then can be popped out from circular buffer by calling *pop\_snd\_q()* and propagated downward to UDP layer by calling the system call *sendto()*. After the transmission, a duplicate copy of touPkg will be wrapped in a format of timer node data structure, as shown in Figure 5, with a transmission timestamp recorded. The *add()* is called to place the timer node onto timer minimum heap (timerheap) where all of outstanding packets are stored. The transmission timestamp (xmitms) records the transmission time that can be used to calculate the round-trip time (RTT)[[1]](#footnote-1) once the sender receives the correspondent ACK from the peer. We will talk more about timer minimum heap in Section 4.2.3.2.

One of the major responsibilities of the timer thread on sender side is to keep track of the current time with the timestamp of the root in timerheap. If timer thread finds a fired timer node which is not yet acknowledged, the *proc\_sndpkt()* will be called to retransmit fired packet and reset the timer as well as update current congestion control states. ToU will enter slow start phase again with cwnd of one MSS, because timeout implies that the given packet may be lost and the current network is unstable.

**Figure 4. ToU packet**

|  |
| --- |
| class touPkg {  public:  touHeader t; // ToU's header  string \*buf; // Payload  int buflen; // Length of payload  };  class touHeader {  public:  u\_long seq; // Sequence Number: 32 bits  u\_long mag; // Magic Cookie: 32-bits  u\_long ack\_seq; // Acknowledgment Number: 32 bits    u\_short doff:4; // Data Offset: 4 bits  u\_short dres:4; // Reserved: 4 bits  u\_short res:1; // Reserved flag: 1 bit  u\_short cwr:1; // Congestion window reduced flag: 1 bit  u\_short ece:1; // ECN-Echo flag: 1 bit  u\_short ack:1; // Acknowledgement: 1 bit  u\_short psh:1; // Push Function: 1 bit  u\_short rst:1; // Reset the connection: 1 bit  u\_short syn:1; // Synchronize sequence numbers: 1 bit  u\_short fin:1; // No more data from sender: 1 bit  u\_short window; // Window size: 16 bits  }; |

**Figure 5. Timer node**

|  |
| --- |
| class node\_t{  public:  conn\_id c\_id; //socket fd(sock id)  time\_id t\_id; //timer id  seq\_id p\_id; //sequence number  sockTb \*st; //socket table pt  string \*payload;  long ms; //expected fired time (ms)  long xmitms; //xmit time (ms)  /\* used in ack \*/  long push\_ackms;//must-send-an-ack time (ms)  long nagle\_ackms;//delayed ack time (ms)  }; |

### ToU receiving model

On the receiver side, as shown in Figure 6, once data have arrived from peer, ToU Process gets data from kernel space by calling *recvfrom()*. Data will be de-encapsulated to the format of touPkg. ToU Process then looks into the ToU header of packet to check whether the sequence number is in-order. If it is, packet will be placed in the circular buffer (cb\_recv) so that once application program calls *touRecv()*, data can be popped out from cb\_recv and passed into application’s user buffer. If ToU Process finds the packet is out-of-order*, push\_hprecvbuf*() will be called to handle the packet by pushing the out-of-order packet onto out-of-order minimum heap (minhp\_recv) waiting for the in-order packet to come. Once receiver gets the expected in-order packet, it then can recover out-of-order packets from minhp\_recv and place the newly arrived in-order packet and recoverable out-of-order packets onto cb\_recv.

Based on the RFC 1122 [5], ToU implements delayed ACK mechanism. Timer thread on the receiver side plays an important role regarding when to send an ACK packet. In particular, ACK can only be delayed less than 500 milliseconds, and in a stream of full-sized segments there should be an ACK for at least 1000 milliseconds. In ToU, as shown in Figure 7, the delays transmission of ACKs is up to 200 milliseconds, since 200 is generally adopted by most TCP implementations in Linux.

Timer thread keeps track of the current time with the fired time of ACKs in a double-ended queue (ackdeque). The ackdeque stores outstanding ACK nodes for each connection. If there is a fired ACK, *proc\_delack()* will be invoked by timer thread to check whether to send an immediate ACK back with the calling of *sendto()*. The decision of sending ACK or delaying ACK is referring to the delayed ACK FSM. Note that the beginning state is initialized to Xmit.

**Figure 6. ToU receiving model**

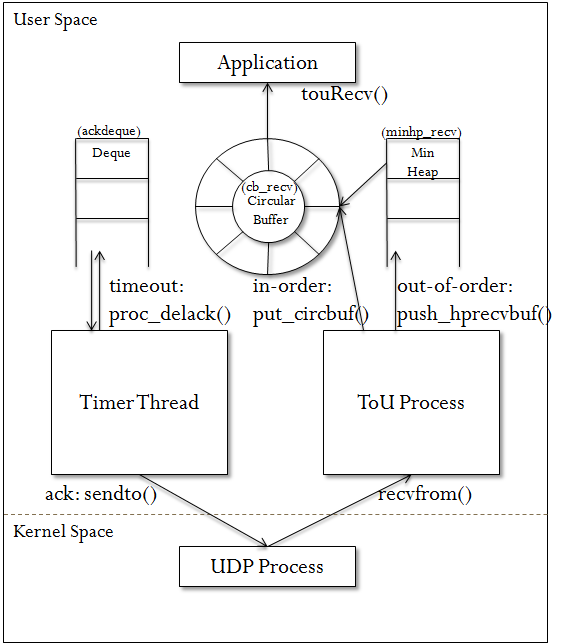
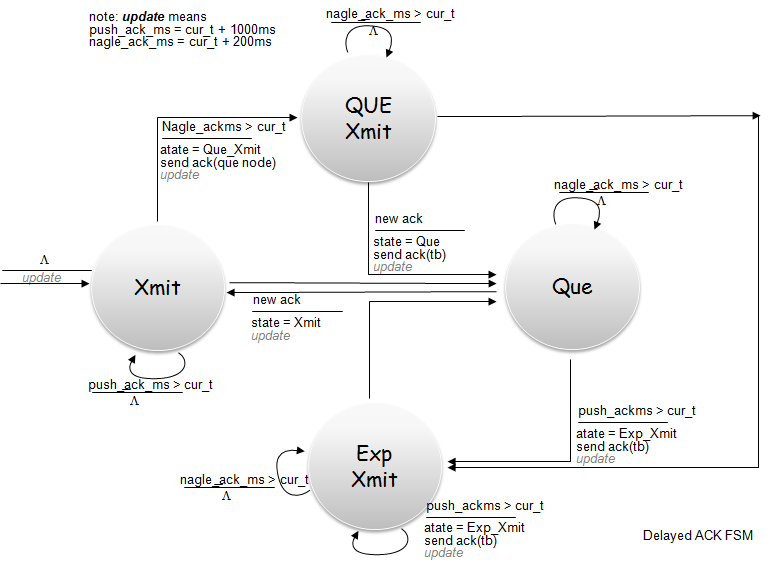


Figure 7. Delayed ACK FSM



### Programs and internal operations

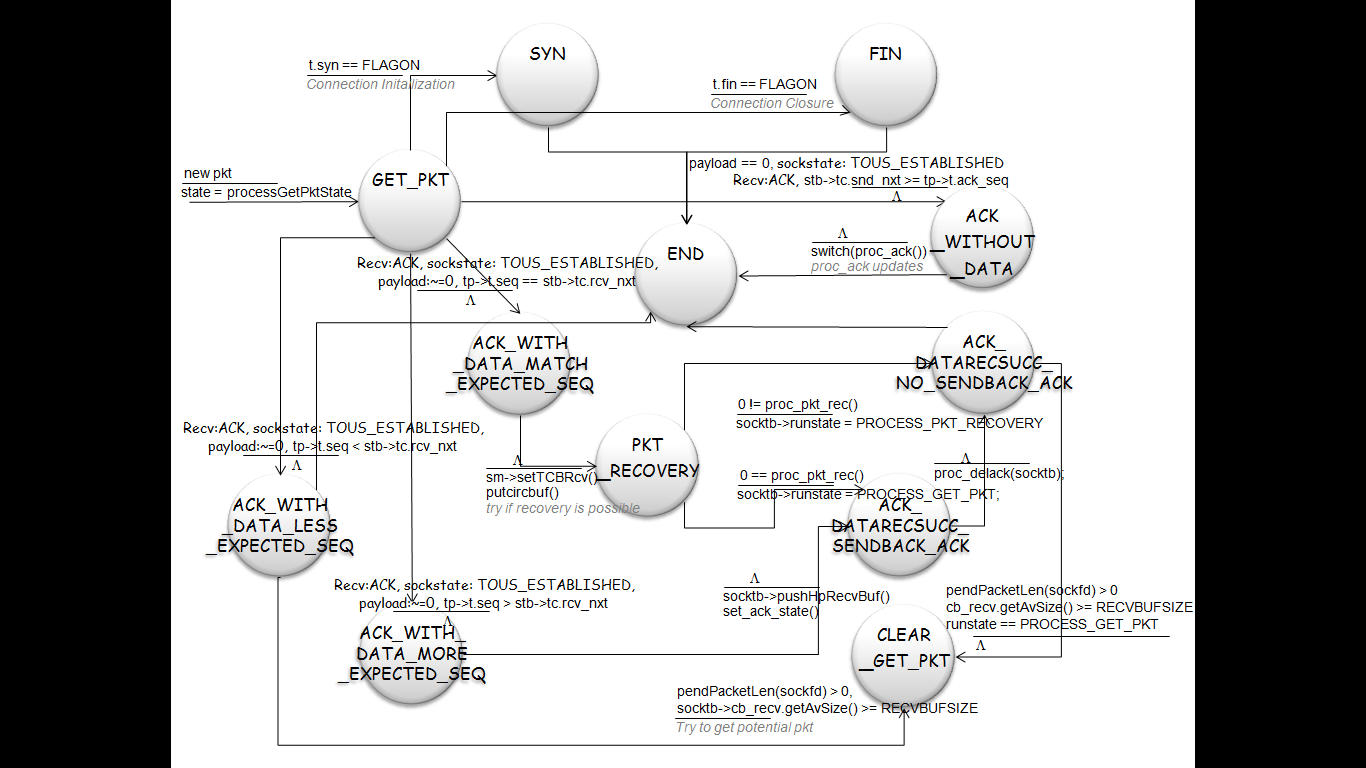
Table 1. Library files

|  |  |
| --- | --- |
| Program name | Function |
| circ\_buf.cpp  circ\_buf.h | Circular buffer for queuing incoming and outgoing data. |
| processtou.cpp  processtou.h | ToU process is handling the sending/receiving of packets. |
| timer.cpp  timer.h  timer\_mng.cpp  timer\_node.cpp  timestamp.cpp  timestamp.h | Timer management provides operations for other functions to call that can help manipulate timer nodes.  Timer thread will keep track of fired timer nodes and retransmit lost packets if necessary. |
| tou\_boost.h  tou\_comm.h | Include Boost library for threads and timers. |
| tou\_sock\_table.cpp  tou\_control\_block.h  tou\_sock\_table.h | Define data structures, such as ToU control block and ToU socket table. |
| tou\_ss\_ca.cpp  tou\_congestion.h | Define FSM for congestion control mechanism. |
| tou.cpp  tou.h | Provide APIs for application programs. |
| tou\_header.cpp  tou\_header.h | Define ToU packet structures. |
| tou\_logger.cpp  tou\_logger.h | Provide logging mechanisms. |

It is worth noting that ToU library consists of many files that make up different modular components, as shown in Table 1. Among these files, we will choose some of the core programs, such as ToU Process (processtou.cpp), Timer Thread, and ToU data structures, to briefly give an abstract description.

#### Processtou.h/Processtou.cpp

Figure 8. Processtou FSM



Processtou.cpp is one of the most important programs in ToU since it is responsible for the receiving and sending of all incoming and outgoing packets. While ToU receives or needs to transmit a packet, processtou will be called to handle the operation. To be more specific, the *run()* function of *processTou* class in *processtou.cpp* will carry out a processtou FSM, as shown in Figure 8, to handle the processing by looking into the ToU packet header and ToU control blocks.

#### Timer.h/Timer.cpp/Timer\_mng.cpp

Timer has its own thread of execution and will be instantiated at the start of the program along with the ToU object. The timer thread checks timerheap (in Figure 3) for outstanding packets and ackdeque (in Figure 6) for fired delayed ACKs in every hundred millisecond to see if any of the retransmission is necessary. This checking is implemented at *doit()* function of *timerCk* class in *timer.cpp*. Since the checking is so frequent, we use minimum heap to maintain the timer data structure, which results in a time complexity of O(1) in every checking iteration. This is because timer thread only needs to peek at the root of timerheap. If the retransmission is not necessary, timer thread will wait for another 100 millisecond. However, if retransmission happens, in *doit()*, timer thread will invoke *proc\_sndpkt()* to handle the packet retransmission. *Proc\_sndpkt()* handles it by firstly clearing acknowledged timer nodes with the calling of *proc\_clear\_ackedpkt()*, and then comparing root’s expected fired time with current time. If the expected fired time (ms) is lesser than the current time, a timeout occurs, *sendpkt()* will be invoked to handle the retransmission process. After retransmission, a new timer node will be generated with a new back-off fired time[[2]](#footnote-2) to replace the old timer. Finally, *proc\_sndpkt()* will call *settwnd()* to update the congestion control state. *Settwnd()* follows the FSM in Figure 2 to set up new states, such as sshrtesh and cwnd values.

#### Tou\_sock\_table.h/ Tou\_control\_block.h

Figure 9. ToU Socket Data Structures

#### Untitled.png

In order to hold ToU state information and internal variables for different connections, we refer to general Linux implementation and RFC 793 [4] where [transmission control](http://www.tcpipguide.com/free/t_TCPConnectionPreparationTransmissionControlBlocksT.htm) block (TCB) is introduced, and create our own ToU socket tables (sockTb) and ToU control block (touCb). SockTb and touCb store almost every important internal state and variables, as shown in Figure 9.

In addition to sockTb and touCb, we introduce socket management class (sockMng) to help manage sockTbs belonging to different connections by providing general functions that can operate on various data. For example, functions such as *getSocketTable()* returns a sockTb pointer of specific socket identifier, *setSocketState()* sets the socket states, and *setTCB()* sets the snd\_nxt and snd\_una values.

As we can see in Figure 9, we use a vector, which is a container, called stbq to store sockTbs for every connection. In this way, once we need to operate on specific connection, we can just fetch its sockTb from the stbq with socket ID and use general functions provided by sockMng to operate on it.

The touCb contains all the important information about the connection, such as the congestion state (cc\_state) that identifies the current congestion control state. The touCb is also used to implement the sliding window mechanism. [It holds variables](http://www.tcpipguide.com/free/t_TCPSlidingWindowDataTransferandAcknowledgementMech.htm) that keep track of the number of bytes received and acknowledged, bytes received and not yet acknowledged, current window size and so forth. Figure 10 shows the detail of the data structure.

**Figure 10. ToU Control Block (TCB)**

|  |
| --- |
| class tou\_cb {  public:  short t\_state; //11 connection states  short cc\_state; //3 congestion states  short t\_delack\_state; //2 delayed ack states  u\_long t\_rto; //current timeout(ms): RTO(i)  u\_short t\_rto\_beta; //RTO(i) beta (default std: 2)  u\_long t\_rtt; //round time trip  float t\_rtt\_alpha;//RTT alpha (default std: 0.125)  /\* WND & SEQ control. See RFC 783 \*/  u\_long iss; //initial send seq #  u\_long irs; //initial rcv seq #  u\_long snd\_una; //send # unacked  u\_long snd\_nxt; //send # next  u\_long rcv\_nxt; //rcv # next  /\* Additional Variables for implementation \*/  short dupackcount;//duplicate ack (count to three)  short TcpMaxConnectRetransmissions;  short TcpMaxDataRetransmissions;  u\_long snd\_wnd; //sender's window  u\_long rcv\_wnd; //rcv window  u\_long snd\_cwnd; //congestion-controlled wnd  u\_long snd\_awnd; //sender's advertised window from recver  u\_long snd\_ssthresh;//threshold for slow start  }; |

#### 

# MEASUREMENTS

The goal of measurements is to validate the correctness of congestion control mechanism and flow control mechanism of ToU by comparing the ToU testing results to the results of TCP Reno in Linux systems. In particular, we will compare the throughput of sending large files and the pattern of congestion window. In the following section, we introduce the benchmarks and testing topology, and then we illustrate the test plans and test results. At the end, we analyze the results of ToU comparing to TCP Reno as well as giving a brief summary.

## Environmental Setup

### Hardware, OS, and kernel tuning

All tests were conducted on an experimental test bed. High-end PCs were connected to a megabit router to form the client/server topology, see Figure 11. The network was isolated. Sender and receiver machines used in tests have identical operating systems, Ubuntu 9.04, and software configurations. The commands given to set up configurations are shown in Figure 12. Both PCs are connected to the router at 100Mb/sec.

Figure 11. Table Hardware setting of client, server, and router

Figure 12. Kernel TCP/UDP tuning

|  |
| --- |
| echo "**reno**" > /proc/sys/net/ipv4/tcp\_congestion\_control  echo "**3**" > /proc/sys/net/ipv4/tcp\_reordering  echo "0" > /proc/sys/net/ipv4/tcp\_abc  echo "0" > /proc/sys/net/ipv4/tcp\_dsack  echo "0" > /proc/sys/net/ipv4/tcp\_sack  echo "0" > /proc/sys/net/ipv4/tcp\_fack  echo "0" > /proc/sys/net/ipv4/tcp\_timestamps  echo "**1**" > /proc/sys/net/ipv4/tcp\_window\_scaling  echo "1" > /proc/sys/net/ipv4/tcp\_no\_metrics\_save  echo "0" > /proc/sys/net/ipv4/tcp\_slow\_start\_after\_idle  echo "5000" > /proc/sys/net/core/netdev\_max\_backlog  echo "72800" > /proc/sys/net/core/optmem\_max  # Socket buffer's size  echo "**72800**" > /proc/sys/net/core/rmem\_default  echo "4194304" > /proc/sys/net/core/rmem\_max  echo "**72800**" > /proc/sys/net/core/wmem\_default  echo "4194304" > /proc/sys/net/core/wmem\_max  #Number of pages allowed for queuing.  echo "65000 80000 94000" > /proc/sys/net/ipv4/tcp\_mem  echo "65000 80000 94000" > /proc/sys/net/ipv4/udp\_mem  #TCP buffer limits  echo "4096 **72800** 4194304" > /proc/sys/net/ipv4/tcp\_rmem  echo "4096 **72800** 4194304" > /proc/sys/net/ipv4/tcp\_wmem  #Minimal size buffers used by UDP sockets.  echo "8192" > /proc/sys/net/ipv4/udp\_rmem\_min  echo "8192" > /proc/sys/net/ipv4/udp\_wmem\_min |

In addition to the Kernel TCP/UDP tuning, the host CPU sometimes will hand over large chunks of data to the intelligent NIC in a single transmit request, letting NIC break down data into smaller segments with the adding of TCP, IP, and Ethernet headers. This technique is called TCP segmentation offload (TSO) [12] when applied to TCP, or generic segmentation offload (GSO). While comparing ToU with TCP performance, it affects the output of throughput and congestion window variation. We need to turn it off by commands shown in Figure 13.

Figure 13. setTSO.sh

|  |
| --- |
| #! /bin/bash  ethtool –K ethX tso off  ethtool –K ethX gso off  ethtool –k ethX |

### Connection monitor

In order to monitor TCP states and collect measurements of internal TCP state variables, we use a module called tcpprobe [13] which can record the states of TCP connections in response to incoming and outgoing packets, see Figure 14 for command details.

Figure 14. setTCPProbe.sh

|  |
| --- |
| #! /bin/bash  modprobe tcp\_probe port=$1  chmod 666 /prob/net/tcpprobe  cat /proc/net/tcpprobe > /tmp/tcpprobe.out & TCPPROB=$! |

### Delay and loss rate emulation

The server can be configured with various packet loss rates and round trip propagation delays to emulate wide range network conditions. We use a network emulation functionality called netem [14] to emulate variable delay, loss, duplication and re-ordering, see Figure 15.

Figure 15. Commands for emulating WAN

|  |
| --- |
| # Delay 50ms  tc qdisc add dev eth0 root netem delay 50ms  tc qdisc change dev eth0 root netem delay 100ms 10ms  # Loss rate set to 0.1%  tc qdisc change dev eth0 root netem loss 0.1% |

## Test Results and Analysis

### Test Plans

In order to collect TCP connection results, there are many tools available for generating traffic [15], such as iperf, [netperf](http://www.netperf.org/netperf/NetperfPage.html), [nttcp](http://hpux.cs.utah.edu/hppd/hpux/Networking/Admin/nttcp-1.47/), and ttcp. We used ttcp to generate traffic since it is light and easy to configure. We add *getsockopt()* function call to get cwnd values at the time when we send data, and execute the commands as following,

Server: ttcp -r -v -p Port > output\_file

Client: ttcp -t -v -l 1456 -p Port IP < test\_file

We seek to define a series of benchmark tests that can be consistently applied and that exercise congestion control functionality of TCP Reno so as to compare its results with ToU library. Since the possibility of packet drop is associated with bulk data transfer, we will perform tests by transmitting 30MB and 100MB file in a format of text and binary. We run it under the conditions such as nice network condition with loss rate of 0%, regular network condition with loss rate of 0.01%, and bad network condition with loss rate of 1%. We will show the results in terms of the trends of cwnd (number of packet) with corresponding sequence number or time (sec).

### Nice network environment

### : 100MB, loss rate **0%**, delay 0ms 100ms 300ms

### **Figure 16. cwnd(packets) vs. time(sec)**

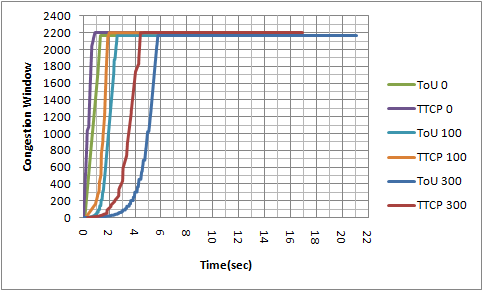
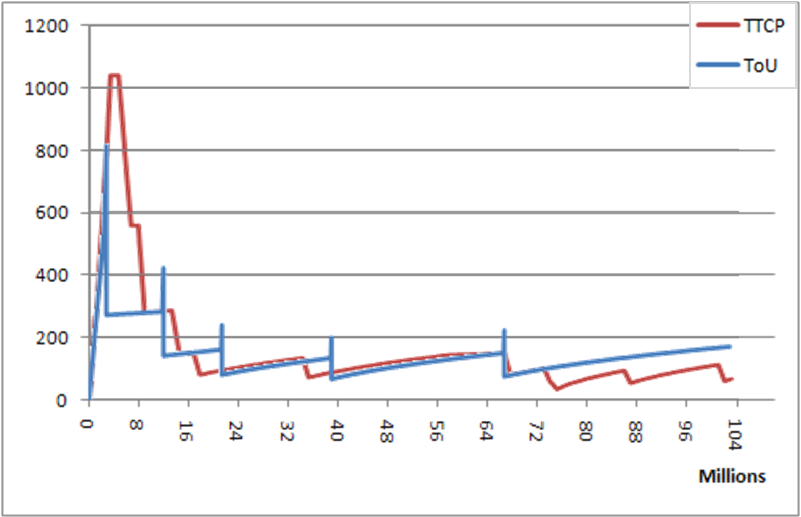


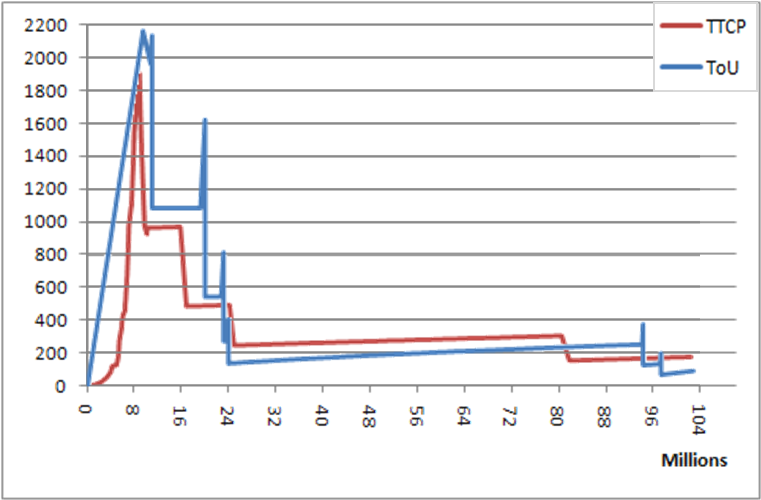
Figure 16 illustrates three different ToU transfers from NW120J to M5200A under the conditions of server delay 0ms, 100ms, and 300ms. Both TCP Reno and ToU are using slow-start throughout the transmission and climbing quickly to the limiting window size of around 2200 segments. The results reflected in Figure 16 show that while we give higher delay, TCP Reno and ToU need to spend more time to reach the limiting window size. In general, the trend of congestion window of ToU seems to be similar to the TCP Reno.

### Regular network environment

### : 100MB, loss rate **0.01%**, delay 0ms 100ms 300ms

### **Figure 17. cwnd(packets) vs. sequence number**





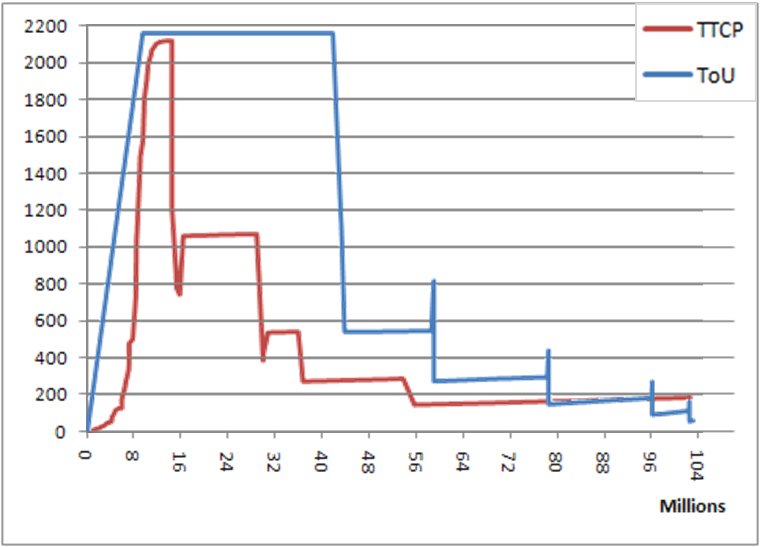


Figure 17 compares the sequence number (in millions) with congestion window size (segments) under different latencies.

Since we are sending 100MB file, and a packet is about 1.5 KB, we can calculate that the total number of transmission is about 70000. We also know that the loss rate is 0.01% so the transmission will encounter about 6~7 losses. This result is correspondent to what we have observed in graphs.

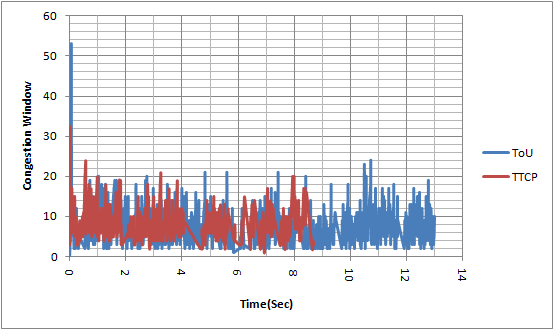
Generally speaking, the patterns of TCP Reno and ToU are similar. Take first graph for example (delay 0ms), the congestion window ramps up exponentially during the slow-start and suffers the first packet loss when it hits 820 (ToU) and 1020(TCP). It then enters the fast recovery phase. Once the lost packet is acknowledged, they enter the congestion avoidance phase and the congestion window grows polynomially since then.

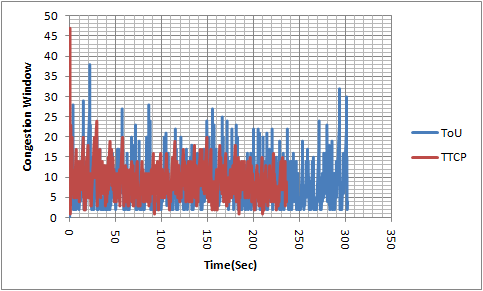
However, when we look into the fast recovery phase of TCP Reno and ToU, we can always find a spike-shaped growth of cwnd in ToU but not in TCP. It is because the specification of classic fast recovery in RFC indicates that the cwnd should firstly be halved after fast retransmit and then it start to inflate cwnd until cwnd becomes large enough to allow new packets to be sent from sender. The Linux implementation, as we mentioned in Section 4.1.2, is letting cwnd decremented by one for every two duplicate ACKs received. In this sense, we can see the growth of cwnd in TCP can be decreased below halved ssthresh and bounce back to ssthresh when it receives a new ACK.

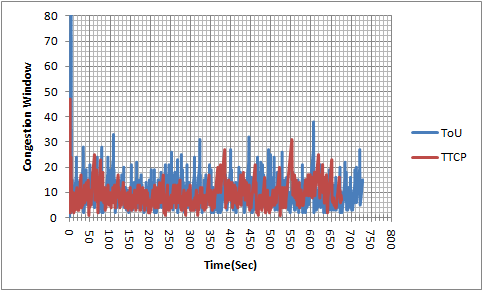
### Bad network environment

### : 30MB, loss rate **1%**, delay 0ms 100ms 300ms

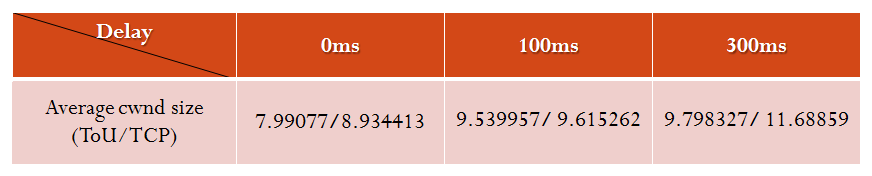
### **Figure 18. cwnd(packets) vs. time(sec)**







Since the packet loss rate is very high, cwnd dramatically reduces to around 10 or even lesser at the very beginning. As we can see in Figure 18, the congestion states are switching swiftly in both TCP and ToU so we think their patterns are similar. We also observe that the average congestion window sizes are close in these three tests. They are as following,



### ToU without congestion control mechanism

### : 100MB, loss rate **0%**, delay 0ms 100ms 300ms

We notice that ToU takes more time to complete the transmission, and its throughput is lower than TCP Reno, as summarized in Table 2 in Section 5.3.

In order to identify the issues, we will give hypotheses (a. and b.) and conduct an experimental test which removes modules of congestion control mechanism in ToU to verify that possible causes are due to the modeling issue.

1. Packet overhead

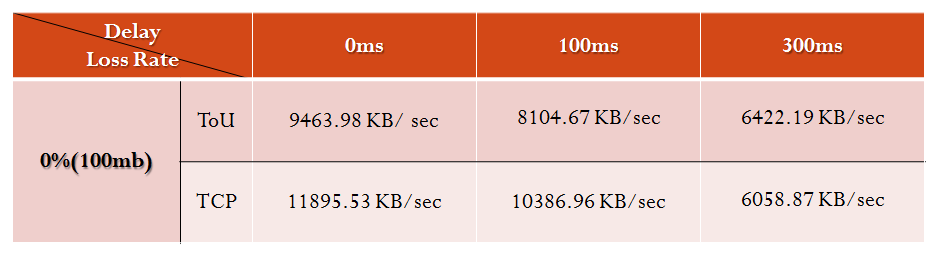
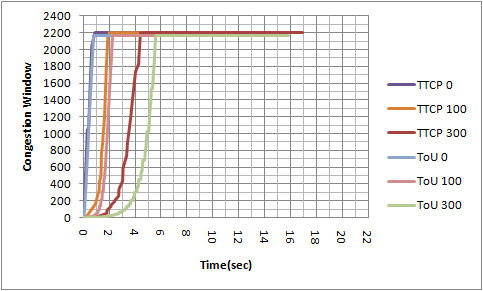
A normal TCP packet runs on Ethernet can carry at most 1460 bytes. However, since the size of ToU header is 16 bytes and size of UDP header is 8 bytes, the maximum payload size ToU can utilize on Ethernet is 1456 bytes. It makes a difference of 4 bytes in every transmission comparing to TCP. However, the overhead is pretty minor, and the effects are negligible.

1. ToU model

In Table 2, we notice that when the delay time comes larger the throughput differences become smaller. It is possibly because the delay gives ToU more time to handle the congestion control processing. In ToU, because of the congestion control mechanism, we need to instantiate a timer thread and many other related data structures. The pushing and popping of data are happening frequently, which slow down the overall throughputs.

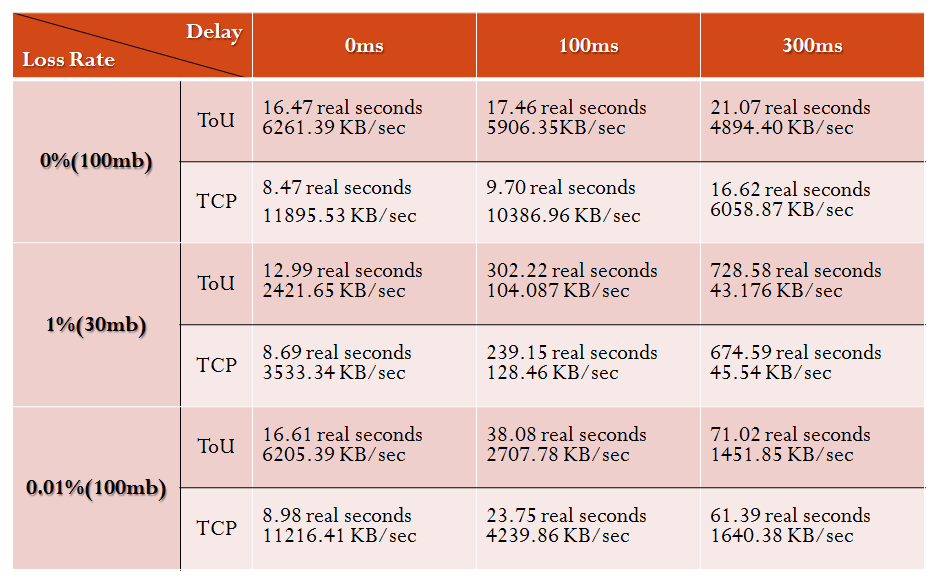
In the following section, we conduct an experiment to test the hypothesis (b.), see results in Figure 19.

### **Figure 19. cwnd(packets) vs. sequence number**



We remove the processing of timer thread, lost packets checking, and duplicate ACKs retransmission, leaving only circular buffers, socket tables, and related functions for basic transmission. In Figure 19, we find out that throughputs of ToU are improved greatly that the results are almost close to TCP Reno. When the delay time is 300ms, ToU even beats TCP Reno.

## Test Conclusion

**Table 2. Test Summary**

According to our test results, the patterns of cwnd versus time or sequence number are similar. Due to the modeling issue, throughputs are lower than TCP Reno, but when the network has higher transmission delay time, the throughput differences come small.

# TASK LIST

ToU consists of many parts, including circular buffer, timer, ToU data structure, processtou, congestion control, connection control, and logging mechanism. Among them, circular buffer is originally written by Salman A. Baset, and enhanced and featured with thread lock by Cheng-Han Lee. Chinmay Gaikwad is responsible for implementing connection control (3-way handshake and closure mechanism) and takes part in some of the other modules. Cheng-Han Lee mainly works on the implementation of timer, internal data structures, packet data structure, processtou, congestion control mechanism, and logging mechanism.

# REFERENCES

1. [draft-baset-tsvwg-tcp-over-udp-01] Salman A. Baset and Henning Schulzrinne, *"TCP-over-UDP"*, baset-tsvwg-tcp-over-udp-01 (work in progress), Dec., 2009.
2. [I-D.ietf-mmusic-ice]Rosenberg, J., "*Interactive Connectivity Establishment (ICE): A Protocol for Network Address Translator (NAT) Traversal for Offer/Answer Protocols*", draft-ietf-mmusic-ice-19 (work in progress), October 2007.
3. Beej Jorgensen, *"Beej's Guide to Network Programming"*, 2009, *<* <http://beej.us/guide/bgnet/> >.
4. [RFC0793] Postel, J., *"Transmission Control Protocol"*, STD 7, RFC 793, September 1981.
5. [RFC1122] Braden, R., Editor, *"Requirements for Internet Hosts -- Communication Layers"*, STD 3, [RFC 1122](http://www.faqs.org/rfcs/rfc1122.html), October 1989.
6. [RFC2525] Paxson, V., Allman, M., Dawson, S., Fenner, W., Griner, J., Heavens, I., Lahey, K., Semke, J. and B. Volz, *"Known TCP Implementation Problems"*, [RFC 2525](http://www.faqs.org/rfcs/rfc2525.html), March 1999.
7. [RFC2581] M. Allman, V. Paxson, W. Stevens, *“TCP Congestion Control”*, April, 1999
8. [RFC2861] Mark Handley, Jitendra Padhye, and Sally Floyd, *"TCP Congestion Window Validation"*, June, 2000.
9. [RFC3782] S. Floyd, T. Henderson, and A. Gurtov, *"The NewReno Modification to TCP's Fast Recovery Algorithm"*, April 2004
10. Dunigan, T. and F. Fowler, "*Almost TCP over UDP*", 2004, < <http://www.csm.ornl.gov/~dunigan/netperf/atou.html> >.
11. [James F. Kurose](http://www.amazon.com/James-F.-Kurose/e/B001IGQHKM/ref=ntt_athr_dp_pel_1), [Keith W. Ross](http://www.amazon.com/Keith-W.-Ross/e/B001IGQFWC/ref=ntt_athr_dp_pel_2), *"Computer Networking: A Top-Down Approach (5th Edition) "*, March 2009.
12. Large Receive Offload, Jonathan Corbet, Aug. 2007,  
    < <http://lwn.net/Articles/243949/>>
13. Tcpprobe, Linux Foundatio, <<http://www.linuxfoundation.org/collaborate/workgroups/networking/tcpprobe>>
14. Netem, Linux Foundatio, <<http://www.linuxfoundation.org/collaborate/workgroups/networking/netem>>
15. Generating TCP Traffic, Linux Foundatio, <<http://www.linuxfoundation.org/collaborate/workgroups/networking/performance_testing>>

Columns on Last Page Should Be Made As Close As Possible to Equal Length

1. ToU Round-Trip Time estimates is computed as:  
   SRTT(i+1) = α \* SRTT(i) + ( 1-α ) \* S(i)  
   RTO(i) = β \* SRTT(i), where α = 0.125, β = 2.  
   Source: <http://www.opalsoft.net/qos/TCP-10.htm> [↑](#footnote-ref-1)
2. Exponential backoff: when timeout occurs, the RTO is doubled.  
   RTOn+1 = max ( 2 RTOn, 64) seconds [↑](#footnote-ref-2)