Programming Assignment 4: Transmission Control Protocol (TCP)

Assigned: April 03, 2018 Due: April 30, 2018, 11:59pm

Preamble

This handout starts out by giving a brief introduction to TCP in section 1 before moving onto the problem statement. So, don't get scared if it seems very long at first. Also, note that unlike the previous assignments in this course, this assignments is relatively more open ended and as a result you have more freedom in your implementation details as long as you adhere to the protocol specified in section 2.1.

Learning Outcomes

After completing this assignment, students should be able to:

- Write code that builds reliable data transfer for application running atop unreliable UDP sockets
- Understand and implement retransmission timeouts
- Understand and implement one's complement checksum for data integrity

Section 1: Background

1.1 Introduction

The Transmission Control Protocol (TCP) standard is defined in the Request For Comment (RFC) standards document number 793 by the Internet Engineering Task Force (IETF). The original specification written in 1981 was based on earlier research and experimentation in the original ARPANET. The design of TCP was heavily influenced by what has come to be known as the "end-to-end argument."

As it applies to the Internet, the end-to-end argument says that by putting excessive intelligence in physical and link layers to handle error control, encryption or flow control you unnecessarily complicate the system. This is because these functions will usually need to be implemented at the endpoints anyways, so duplication of this functionality in the intermediate points can be a waste. The result of an end-to-end network then, is to provide minimal functionality on a hop-by-hop basis and maximal control between end- to-end communicating systems.

The end-to-end argument helped determine the design of various components of TCP's reliability, flow control, and congestion control algorithms. The following are a few important characteristics of TCP.

1.2 Byte Stream Delivery

TCP interfaces between the application layer above and the network layer below. When an application sends data to TCP, it does so in 8-bit byte streams. It is then up to the sending TCP to segment or delineate the byte stream in order to transmit data in manageable pieces to the receiver1. It is this lack of 'record boundaries" which give it the name" byte stream delivery service.

1.3 Connection-oriented Approach

Before two communicating TCP endpoints can exchange data, they must first agree upon the willingness to communicate. Analogous to a telephone call, a connection must first be made before two parties exchange information.

1.4 Reliability

A number of mechanisms help provide the reliability TCP guarantees. Each of these is described briefly below.

Checksums: All TCP segments carry a checksum, which is used by the receiver to detect errors with either the TCP header or data.

Duplicate data detection: It is possible for packets to be duplicated in packet switched network; therefore TCP keeps track of bytes received in order to discard duplicate copies of data that has already been received.

Retransmissions: In order to guarantee delivery of data, TCP must implement retransmission schemes for data that may be lost or damaged. The use of positive acknowledgements by the receiver to the sender confirms successful reception of data. The lack of positive acknowledgements, coupled with a timeout period (see timers below) calls for a retransmission.

Sequence numbers: In packet switched networks, it is possible for packets to be delivered out of order. It is TCP's job to properly sequence segments it receives so it can deliver the byte stream data to an application in order.

Timers: TCP maintains various static and dynamic timers on data sent. The sending TCP waits for the receiver to reply with an acknowledgement within a bounded length of time. If the timer expires before receiving an acknowledgement, the sender can retransmit the segment

1.5 Connection Establishment and Termination

TCP provides a connection-oriented service over packet switched networks. Connection-oriented implies that there is a virtual connection between two endpoints. There are three phases in any virtual connection. These are the connection establishment, data transfer and connection termination phases.

1.6 Three-way handshake

In order for two hosts to communicate using TCP they must first establish a connection by exchanging messages in what is known as the three-way handshake. The diagram below depicts the process of the three-way handshake.

From Figure 1, it can be seen that there are three TCP segments exchanged between two hosts, Host A and Host B. Reading down the diagram depicts events in time.

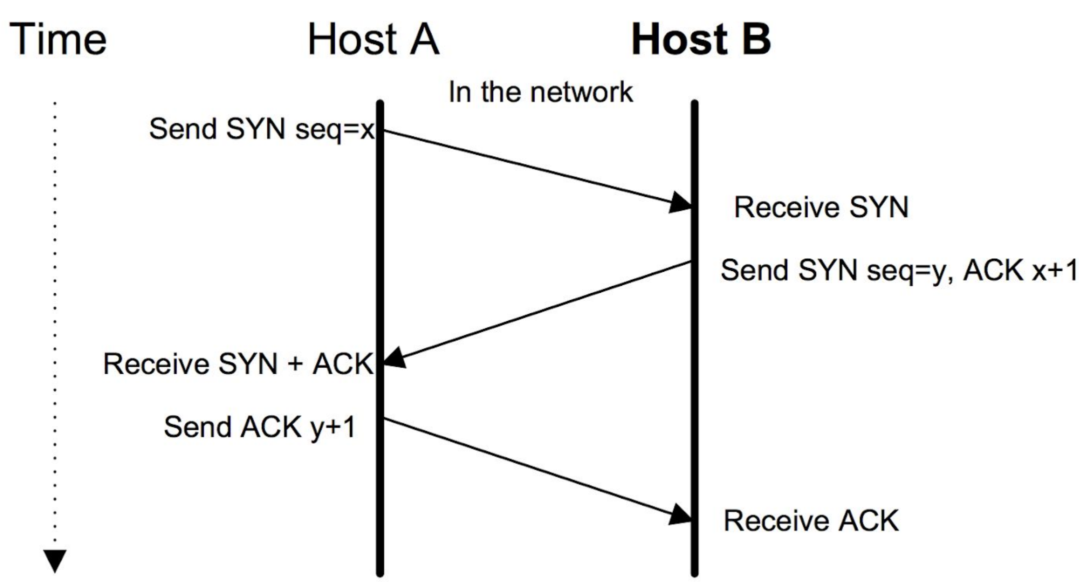


Figure 1. Three way handshake

To start, Host A initiates the connection by sending a TCP segment with the SYN control bit set and an initial sequence number (ISN) we represent as the variable x in the sequence number field.

At some moment later in time, Host B receives this SYN segment, processes it and responds with a TCP segment of its own. The response from Host B contains the SYN control bit set and its own ISN represented as variable y. Host B also sets the ACK control bit to indicate the next expected byte from Host A should contain data starting with sequence number x+1.

When Host A receives Host B's ISN and ACK, it finishes the connection establishment phase by sending a final acknowledgement segment to Host B. In this case, Host A sets the ACK control bit and indicates the next expected byte from Host B by placing acknowledgement number y+1 in the acknowledgement field.

In addition to the information shown in the diagram above, an exchange of source and destination ports to use for this connection are also included in each senders' segments.

1.7 Data transfer

Once ISNs have been exchanged, communicating applications can transmit data between each other. Most of the discussion surrounding data transfer requires us to look at flow control and congestion control techniques which we discuss later in this document. A few key ideas will be briefly made here, while leaving the technical details aside.

A simple TCP implementation will place segments into the network for a receiver as long as there is data to send and as long as the sender does not exceed the window advertised by the receiver. As the receiver accepts and processes TCP segments, it sends back positive cumulative acknowledgements, indicating the next expected byte in sequence. If data is duplicated or lost, a "hole" may exist in the byte stream. A receiver will continue to acknowledge the most current contiguous byte it has accepted.

If data queued by the sender reaches a point where data sent will exceed the receiver's advertised window size, the sender must halt transmission and wait for further acknowledgements and an advertised window size that is greater than zero before resuming.

Timers are used to avoid deadlock and unresponsive connections. Delayed transmissions are used to make more efficient use of network bandwidth by sending larger "chunks" of data at once rather than in smaller individual pieces.

1.8 Connection termination

In order for a connection to be terminated, four segments are required to completely close a connection. Four segments are necessary due to the fact that TCP is a full-duplex protocol, meaning that each end must shut down independently. The connection termination phase is shown in Figure 2 below.

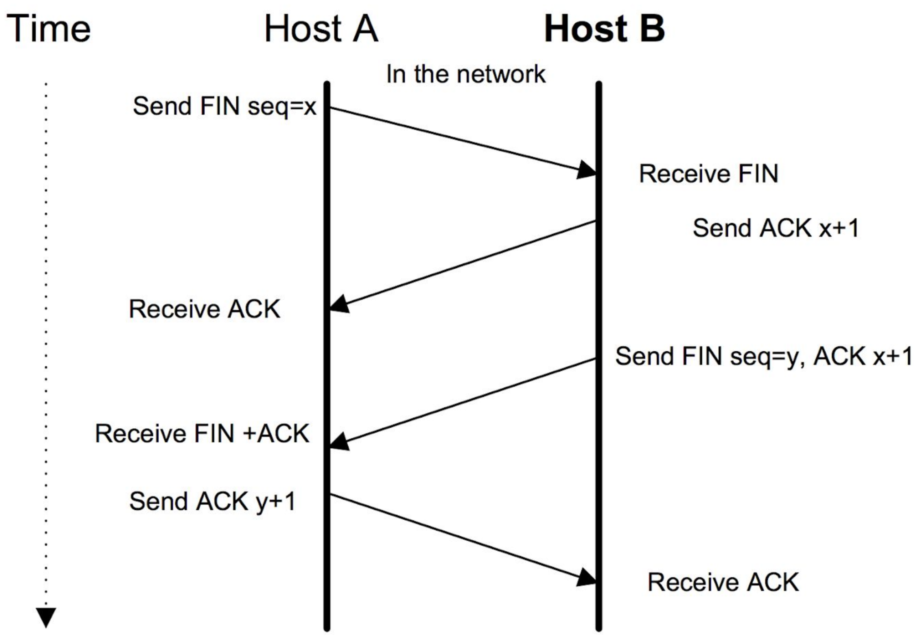


Figure 2: Connection Tear Down

Notice that instead of SYN control bit fields, the connection termination phase uses the FIN control bit fields to signal the close of a connection.

To terminate the connection in our example, the application running on Host A signals TCP to close the connection. This generates the first FIN segment from Host A to Host B. When Host B receives the initial FIN segment, it immediately acknowledges the segment and notifies its destination application of the termination request. Once the application on Host B also decides to shut down the connection, it then sends its own FIN segment, which Host A will process and respond with an acknowledgement.

1.9 Sliding Window and Flow Control

Flow control is a technique whose primary purpose is to properly match the transmission rate of sender to that of the receiver and the network. It is important for the transmission to be at a high enough rate to ensure good performance, but also to protect against overwhelming the network or receiving host.

TCP uses the window field, briefly described previously, as the primary means for flow control. During the data transfer phase, the window field is used to adjust the rate of flow of the byte stream between communicating TCPs.

Figure 3 below illustrates the concept of the sliding window.

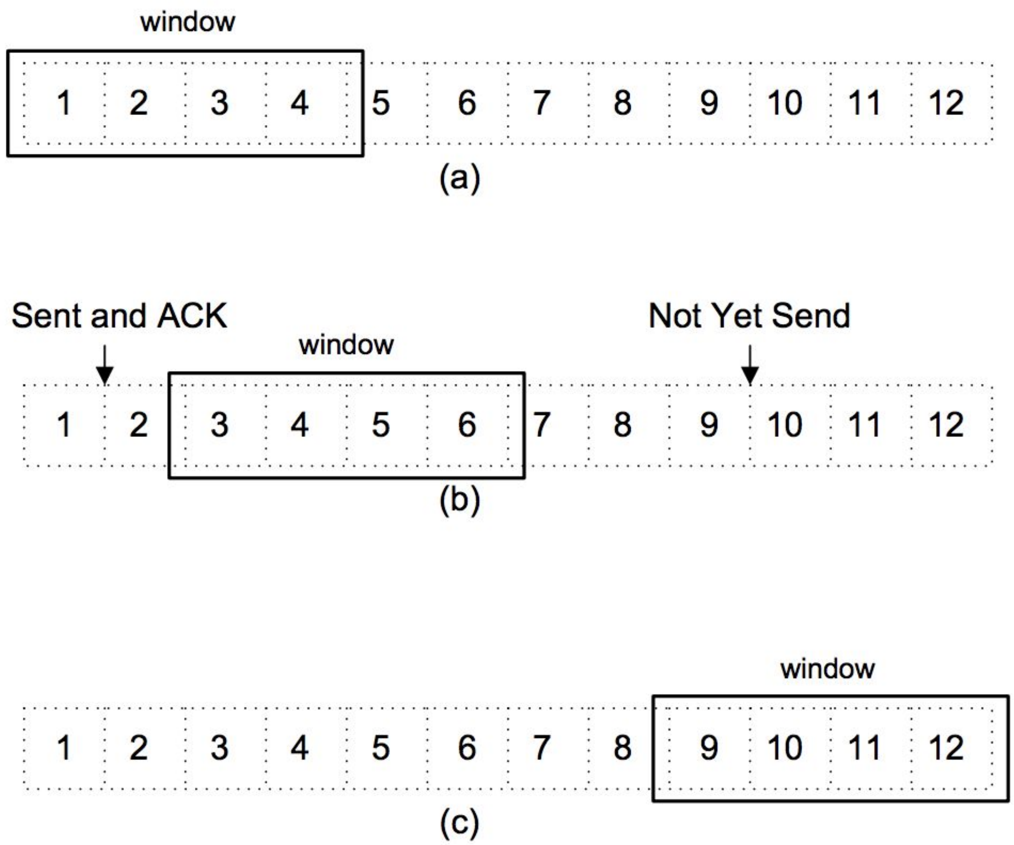


Figure 3: Sliding Window Protocol

In this simple example, there is a 4-byte sliding window. Moving from left to right, the window ”slides” as bytes in the stream are sent and acknowledged.

1.10 Congestion control

TCP congestion control and Internet traffic management issues in general is an active area of research and experimentation. This final section is a very brief summary of the standard congestion control algorithms widely used in TCP implementations today.

1.11 Slow Start

Slow Start, a requirement for TCP software implementations is a mechanism used by the sender to control the transmission rate, otherwise known as sender-based flow control. This is accomplished through the return rate of acknowledgements from the receiver. In other words, the rate of acknowledgements from the receiver determines the rate at which the sender can transmit data.

Initially, the Slow Start algorithm sets the congestion window to one segment, which is the maximum segment size (MSS) initialized by the receiver during the connection establishment phase. When acknowledgements are returned by the receiver, the congestion window increases by one segment for each acknowledgement returned. Thus, the sender can transmit the minimum of the congestion window and the advertised window of the receiver, which is simply called the transmission window.

Slow Start is actually not very slow when the network is not congested and network response time is good. For example, the first successful transmission and acknowledgement of a TCP segment increases the window to two segments. After successful transmission of these two segments and acknowledgements completes, the window is increased to four segments. Then eight segments, then sixteen segments and so on, doubling from there on out up to the maximum window size advertised by the receiver or until congestion finally does occur.

1.12 Congestion avoidance

During the initial data transfer phase of a TCP connection the Slow Start algorithm is used. However, there may be a point during Slow Start that the network is forced to drop one or more packets due to overload or congestion. If this happens, Congestion Avoidance is used to slow the transmission rate. However, Slow Start is used in conjunction with Congestion Avoidance as the means to get the data transfer going again so it doesn't slow down and stay slow.

In the Congestion Avoidance algorithm retransmission timer expiring or the reception of duplicate ACKs can implicitly signal the sender that a network congestion situation is occurring. The sender immediately sets its transmission window to one half of the current window size (the minimum of the congestion window and the receiver's advertised window size), but to at least two segments. If congestion was indicated by a timeout, the congestion window is reset to one segment, which automatically puts the sender into Slow Start mode. If congestion was indicated by duplicate ACKs, the Fast Retransmit and Fast Recovery algorithms are invoked (see below).

As data is received during Congestion Avoidance, the congestion window is increased. However, Slow Start is only used up to the halfway point where congestion originally occurred. This halfway point was recorded earlier as the new transmission window. After this halfway point, the congestion window is increased by one segment for all segments in the transmission window that are acknowledged. This mechanism will force the sender to more slowly grow its transmission rate, as it will approach the point where congestion had previously been detected.

1.13 Fast retransmit

When a duplicate ACK is received, the sender does not know if it is because a TCP segment was lost or simply that a segment was delayed and received out of order at the receiver. If the receiver can reorder segments, it should not be long before the receiver sends the latest expected acknowledgement. Typically no more than one or two duplicate ACKs should be received when simple out of order conditions exist. If however more than two duplicate ACKs are received by the sender, it is a strong indication that at least one segment has been lost. The TCP sender will assume enough time has lapsed for all segments to be properly re-ordered by the fact that the receiver had enough time to send three duplicate ACKs.

When three or more duplicate ACKs are received, the sender does not even wait for a retransmission timer to expire before retransmitting the segment (as indicated by the position of the duplicate ACK in the byte stream). This process is called the Fast Retransmit algorithm.

Section 2: Problem Statement

For this assignment, you have to implement a Transmission Control Protocol which should incorporate only the following features **on top of the unreliable UDP sockets:**

- Reliability (with appropriate re-transmissions)
- Data Integrity (with checksums)
- Connection Management (SYN and FIN)
- Optimizations (fast retransmit)

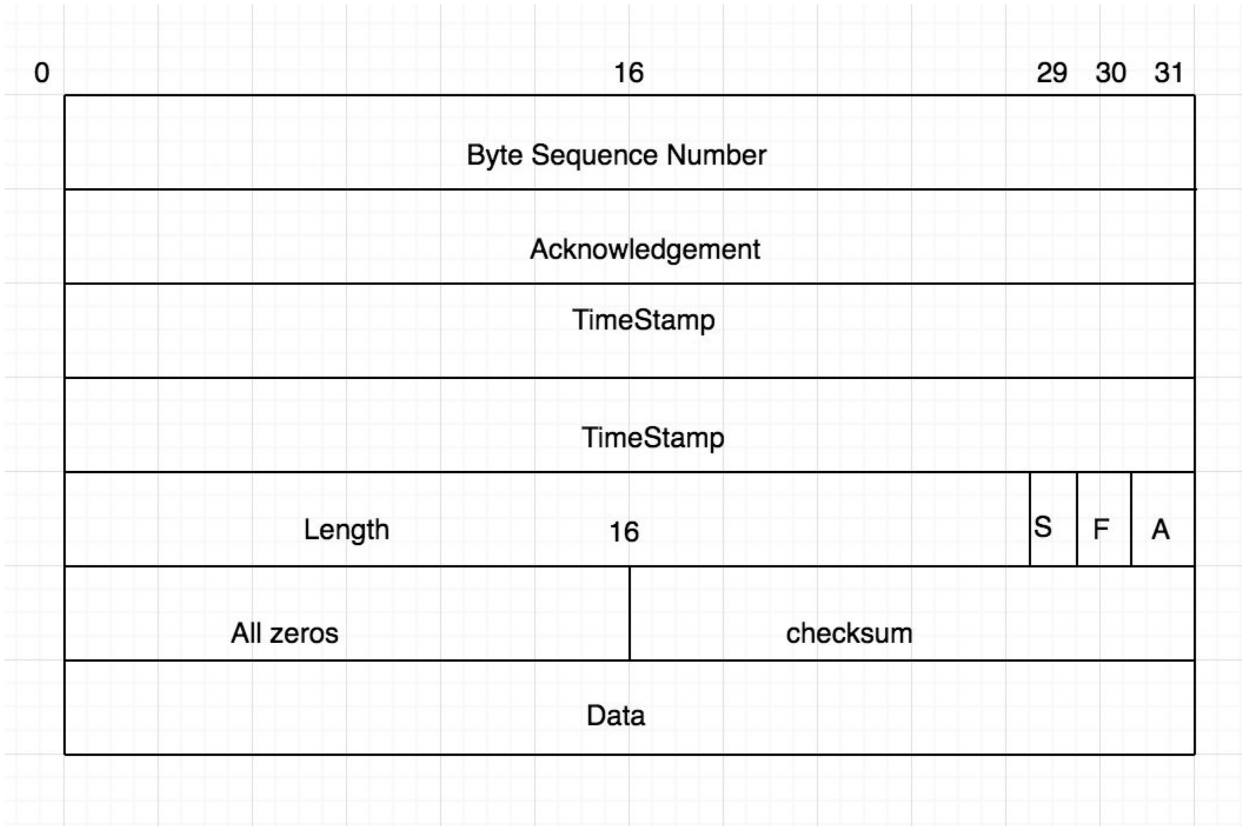
Then you are required to transfer a file from the client to the server using the new reliable transport protocol you just developed.

2.1 Protocol Specification

The various components of the protocol are explained step by step. Please strictly adhere to the specifications.

2.1.1 Message Format

All data are sent using UDP. You have to communicate all the transport layer information in the data portion of a UDP packet. To support reliability, hosts will implement a variant of the Go-Back-N protocol. The sender will tag each outgoing message with an increasing sequence number. The receiver will use the sequence numbers to ensure that all messages have been received in the correct order. If a message is received out of order it will be stored in the receiver’s buffer (but not delivered to an application) and an acknowledgment corresponding to the last successful contiguous byte received will be retransmitted. The sender will maintain an acknowledgment timeout based on the round trip time of the link between the end hosts. If a duplicate acknowledgment is received or if the acknowledgment timeout expires the sender will resend the unacknowledged segment destined for the receiver. A one's complement checksum (The detailed explanation is in later section) on the is used to enforce message integrity. The message segment will have the following format:



2.1.2 Byte Sequence Number is incremented according to the bytes sent. It indicates the position of the first byte of the data in this segment.

2.1.3 Acknowledgment indicates the next byte expected in the reverse direction.

2.1.4 Timestamp is derived from the System.nanoTime() function and is the time of data transmission (in nanoSeconds) which is 64 bits (or 8 bytes) long in size.

2.1.5 Length is the length of the data portion (in bytes) and please pay attention that the least three significant bits are used by flags. That means valid number of bits for the length field are only 27 bits (You will need to do bit manipulation to access and set these fields).

2.1.6 Three flags: S for SYN, A for ACK, and F for FIN.

2.1.7 Checksum is the one's complement checksum computed over the Sequence Number. See appendix A on how to calculate one's complement checksum.

2.1.8 Maximum Number of Retransmissions If unacknowledged messages remain in a host's send buffer and no response from the destination has been received after multiple retransmission attempts, the sending host will stop trying to send the messages and report an error. This maximum is set to 16 by default.

2.1.9 Maximum Transmission Unit

Maximum Transmission Unit(MTU) determines the maximum size of payload that can be transmitted in one packet. So in order to transfer a huge file , it needs to be divided into smaller chunks the maximum size of each chunk being MTU. It is passed as a command line argument to the client and server during startup.

2.1.10 Connection State and Peer Actions

Suppose host A wants to send a message to host B. Assume, for now, that A has never sent a message to B (as will be clear, it does not matter if A has rebooted or has sent a message a long time ago, etc.).

2.1.10.1 Data Send Actions:

A will send a data segment as governed by the size of the sliding congestion window. In this assignment, the congestion window is a configured parameter ~~.~~

A will include a monotonically increasing timestamp (See Section 2.2 on how to compute this timestamp) on the packet. If A thinks this is a new connection (because it has never communicated with B before, or because it has somehow lost the connection state to B), it will set the sequence number to 0. For each subsequent message, A will include a new timestamp on the packet and increment the sequence number by the number of bytes sent.

2.1.10.2 Receiver Actions:

Connection start and data transfer: If this is the first time B is communicating with A and has just received a segment with sequence number equal to zero, it creates a new connection state for A. For each segment received, B will send an acknowledgment to A. The acknowledgment packet has sequence number corresponding to the next expected byte. If no data is sent along with the acknowledgment, the length of this segment will be zero. Note data packets cannot have zero length. B also copies the timestamp field from the data segment into the corresponding ACK segment. Thus, A can use this timestamp field in the acknowledgment to calculate the round trip time for segments (refer to our discussion in class on calculating round trip times).

Retransmissions: For each packet sent, the sender must maintain a retransmission timer (the computation of the timer is described in the next section). Whenever a packet has not been ACKed before its retransmission timer goes off, it must be re-sent. There is a timer set for the re-transmissions as well — thus, if the re-transmission is not ACKed, the packet will be re-re-transmitted. Apart from timeout based retransmissions, the sender also uses three duplicate acknowledgments for the same sequence number as an indicator of loss and retransmits the corresponding lost segment. This is the "fast retransmit" approach.

2.2 Timeout Computation

The sender places the current time in the packet timestamp field of the message header. When a packet is acknowledged by the receiving client it will copy the packet timestamp into the acknowledged timestamp field. When the sender receives the acknowledgment it can subtract the acknowledgment timestamp from the current time to calculate the round trip time. If a client is sending a cumulative acknowledgment of several packets, the timestamp from the latest received packet which is causing this acknowledgment should be copied into the reply. We will use a simple exponentially weighted average to compute the timeout.

Before sending the first packet, the timeout value is, arbitrarily, set to 5 seconds. Assume the sender has just received an ack with sequence S and timestamp T. Let C be the current time at the sender and TO be the timeout time.

```
if (S = 0)
        ERTT := (C - T)
        EDEV := 0
        TO := 2*ERTT
Else
        SRTT := (C - T)
        SDEV := |SRTT - ERTT|
        ERTT := a*ERTT + (1-a)*SRTT
        EDEV := b*EDEV + (1-b)*SDEV
        TO := ERTT + 4*EDEV
```

The value of a is set to 0.875 and b is 0.75.

2.2.1 Computing the sender timestamp

In the 64-bit timestamp field, the sender includes a monotonically increasing timestamp with granularity of one nanosecond. This can be obtained from the System.nanoTime() function.

2.3 Host Commands/Output Format

You will implement a java executable called TCPend. Command line arguments will indicate which host is the initiator of a TCP transfer. The transfer initiator tcpend must support the following options at startup:

java TCPend -p <port> -s <remote-IP> -a <remote-port> –f <file name> -m <mtu> -c <sws>

- port: Port number at which the client will run
- remote-IP : IP address of the remote process (Receiver)
- remote-port: port at which the remote process/ transmission is running
- file name: name of the file to be transferred
- mtu : Maximum transmission unit
- sws: Sliding window size

The remote process (that is receiving the data) uses the following set of arguments:

java TCPend -p <port> -m <mtu> -c <sws>

Host output

Each host should output the information about each segment that it sends and receives in the following format. < snd/rcv> <time> <flag-list> <seq-number> <number-of-bytes> <ack-number> where, flag-list includes S (SYN), A (ACK), F (FIN), and D (Data). The following are valid output lines for a connection initiator that sends 112 bytes of data:

snd 34.335 S - - - 0 0 0

rcv 34.8 S A - - 0 0 1

snd 34.81 - A - - 0 0 1

snd 35.5 - - - D 1 56 1

snd 35.6 - - - D 57 56 1

rcv 36.2 - A - - 0 0 113

snd 36.65 - - F - 113 0 1

rcv 37.2 - A F - 1 0 114

snd 37.3 - A - - 113 0 2

At the end of the transfer, i.e., once the connection has been closed you should print the following statistics:

- Amount of Data Transferred/Received
- No of Packets Sent/Received
- No of Packets discarded (out of sequence)
- No of Packets discarded (wrong checksum)
- No of Retransmissions
- No of Duplicate Acknowledgements

Section 3: Testing

To test your implementation you could use a mininet virtual topology like you have been doing in the earlier assignments but modify the router implementation to drop packets with a given probability. E.g., drop 5% of the IP packets received. Then you could try sending the file from one mininet host to another (use xterm to open terminal windows on different mininet hosts) connected via the modified router and see if the file gets transferred correctly. [Here] is a sample router implementation of your assignment2 which drops roughly 5% of the packets. Look at the end of forwardIPPacket() method inside Router.java. You can modify the drop rates for your testing. You can use the similar commands from assignment 2 to run it.

To run it first start mininet using the following command:

```
sudo ./run_mininet.py topos/single_rt.topo -a
```

Then start pox:

```
./run_pox.sh
```

Then start your virtual router that drops 5% of all the IP packets.

```
java -jar VirtualNetwork.jar -v r1 -r rtable.r1 -a arp_cache
```

Now open terminal windows on hosts h1 and h2 by typing xterm h1 h2 in mininet console. Navigate to the directory where with your TCP implementation and and send the file from h1 to h2. If your implementation is correct, the file should be successfully transferred to h2 i.e., content should exactly match.

Note that this assignment is more open ended than your previous assignments in this course. So, you have more freedom for your implementation methodology as long as you correctly meet the protocol defined above. To test, the TAs would try to transfer files of different sizes (small, medium, large) with different packet drop rates and would check if data transfer was correct. I.e., file received by the receiver should be identical to the one present at the sender.

Section 4: Submission

- The source code for your TCP implementation. Include all the java files in a src directory
- A makefile that compiles the java source code
- A README file with the names and CS usernames of both group members

You must submit a single tar file containing the above. To create the tar file, run the following command, replacing username1 and username2 with the CS username of each group member:

tar czvf username1_username2.tgz src makefile README

Upload the tar file to the Assignment 4 on canvas. Please submit only one tar file per group.

Appendix A

calculating the checksum:

1. Add the 16-bit values up. Each time a carry-out (17th bit) is produced, swing that bit around and add it back into the LSb (one's digit). This is somewhat erroneously referred to as "one's complement addition."

2. Once all the values are added in this manner, invert all the bits in the result. A binary value that has all the bits of another binary value inverted is called its "one's complement," or simply its "complement."

For example, we have 4 parts each with 16 bits to be added up for checksum calculation:

```
1000 0110 0101 1110
1010 1100 0110 0000
0111 0001 0010 1010
1000 0001 1011 0101
```

First, we add the first two 16-bit values at a time:

```
1000 0110 0101 1110   First 16-bit value
+   1010 1100 0110 0000   Second 16-bit value
---------------------
1 0011 0010 1011 1110   Produced a carry-out, which gets added
+  \----------------> 1   back into LBb
```

```
--------------------
0011 0010 1011 1111
+   0111 0001 0010 1010   Third 16-bit value
--------------------
0 1010 0011 1110 1001   No carry to swing around (**)
+   1000 0001 1011 0101   Fourth 16-bit value
--------------------
1 0010 0101 1001 1110   Produced a carry-out, which gets added
+  \---------------> 1   back into LBb
--------------------
0010 0101 1001 1111   Our "one's complement sum"
```