COMP3331/9331 Computer Networks and Applications

Assignment for Term 1, 2023

Version 1.0

Due: 11:59am (noon) Friday, 21 April 2023 (Week 10)

Updates to the assignment, including any corrections and clarifications, will be posted on the subject website. Please make sure that you check the subject website regularly for updates.

1. Change Log

Version 1.0 released on 9th March 2023.

2. Goal and learning objectives

For this assignment, you are to implement a reliable transport protocol over the UDP protocol. We will refer to the reliable transport protocol that you will be implementing in this assignment as Simple Transport Protocol (STP). STP will include most (but not all) of the features that are described in Sections 3.5.4 - 3.5.6 of the text Computer Networking by Kurose and Ross (7th or 8th ed.) or equivalent parts from the Week 4/5 lecture notes. Examples of these features include timeout, ACK, sequence numbers, sliding window, etc. Note that these features are commonly found in many transport protocols. Therefore, this assignment will give you an opportunity to implement some of these basic features of a transport protocol. In addition, you may have wondered why the designer of the TCP/IP protocol stack includes such a feature-less transport protocol as UDP. You will find in this assignment that you can design your own transport protocol and run it over UDP. This is the case for some multimedia delivery services on the Internet, where they have implemented their own proprietary transport protocol over UDP. QUIC, a newly proposed transport protocol also runs over UDP and implements additional functionalities such as reliability.

Recall that UDP provides point-to-point, unreliable datagram service between a pair of hosts. In this programming assignment, you will develop a more structured protocol, STP, which ensures reliable, end-to-end delivery of data in the face of packet loss. STP provides a byte-stream abstraction like TCP and sends pipelined data segments using a sliding window. However, STP does not implement congestion control or flow control. Finally, whereas TCP allows fully bidirectional communication, your implementation of STP will be asymmetric. There will two distinct STP endpoints, "sender" and "receiver" respectively. Data packets will only flow in the "forward" direction from the sender to the receiver, while acknowledgments will only flow in the "reverse" direction from the receiver back to the sender. To support reliability in a protocol like STP, state must be maintained at both endpoints. Thus, as in TCP, connection set-up and connection teardown phases will be an integral part of the protocol. STP should implement a sliding window protocol wherein multiple segments can be sent by the sender in a pipelined manner. Like TCP, STP will include some elements of both Go-Back-N (GBN) and Selective Repeat (SR). You will use your STP protocol to transfer a text file (examples provided on the assignment webpage) from the sender to the receiver.

The receiver program must also emulate the behaviour of an unreliable communication channel between the sender and receiver. Even though UDP segments can get lost, the likelihood of such losses is virtually zero in our test environment, where the sender and receiver will be executed on the same machine. Further, to properly test the implementation of your sender program, we would like to control the unreliable behaviour of the underlying channel. The provided receiver program emulates loss of STP segments in both directions – (i) data, SYN and FIN segments in the forward

direction and (ii) ACK segments in the reverse direction. You may assume that the underlying channel will never reorder or corrupt STP segments (in both directions).

Note that it is mandatory that you implement STP over UDP. Do not use TCP sockets. You will not receive any mark for this assignment if you use TCP sockets.

2.1 Learning Objectives

On completing this assignment, you will gain sufficient expertise in the following skills:

1. Detailed understanding of how reliable transport protocols such as TCP function.
2. Socket programming for UDP transport protocol.
3. Protocol and message design.

Non-CSE Student Version: The rationale for this option is that students enrolled in a program that does not include a computer science component have had very limited exposure to programming and in particular working on complex programming assignments. A Non-CSE student is a student who is not enrolled in a CSE program (single or double degree). Examples would include students enrolled exclusively in a **single degree program** such as Mechatronics or Aerospace or Actuarial Studies or Law. **Students enrolled in dual degree programs that include a CSE program as one of the degrees do not qualify**. Any student who meets this criterion and wishes to avail of this option **MUST** email cs3331@cse.unsw.edu.au to seek approval before **5pm, 17th March (Friday, Week 5)**. If approved, we will send you the specification for the non-CSE version of the assignment. We will assume by default that all students are attempting the CSE version of the assignment unless they have sought explicit permission. No exceptions.

3. Assignment Specification

STP should be implemented as two separate programs: Sender and Receiver. You should implement **unidirectional** transfer of data from the sender to the receiver. As illustrated in Figure 1, data segments will flow from Sender to Receiver while ACK segments will flow from receiver to sender. The sender and receiver programs will be run from different terminals on the same machine, so you can use localhost, i.e., 127.0.0.1 as the IP address for the sender and receiver in your program. Let us reiterate this, **STP must be implemented on top of UDP**. Do not use TCP sockets. If you use TCP, you will not receive any marks for your assignment.

You will find it useful to review Sections 3.5.4 - 3.5.6 of the text (or the relevant parts from the Week 5 lecture notes). It may also be useful to review the basic concepts of reliable data transfer from Section 3.4 (or relevant parts from the Week 4 lecture notes). Section 3.5 of the textbook which covers the bulk of the discussion on TCP is available to download on the assignment page.

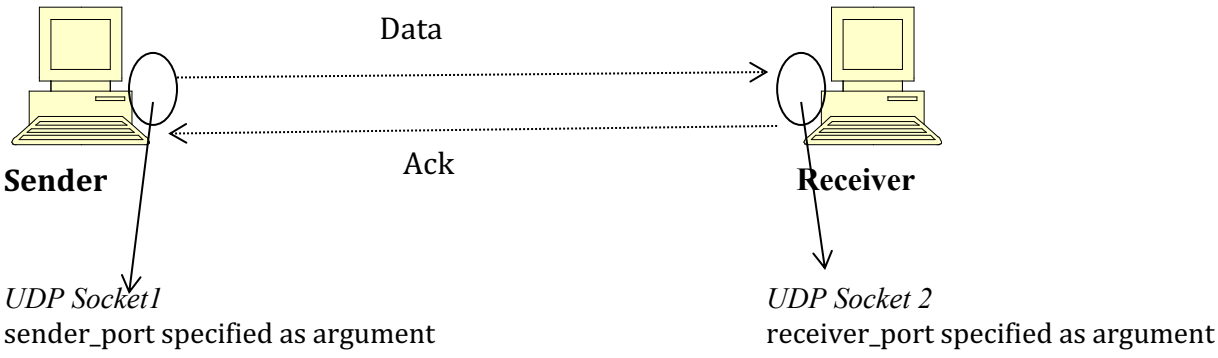


Figure 1: This depicts the assignment setup. A file is to be transferred from the Sender to the Receiver, both running on the same machine. Data segments will flow from the sender to receiver, while ACK segments will flow from the receiver to sender.

3.1 File Names

The main code for the sender should be contained in the following files: `sender.c`, or `Sender.java` or `sender.py`. You may create additional files such as header files or other class files and name them as you wish.

The sender should accept the following four arguments:

1. `sender_port`: the UDP port number to be used by the sender to send STP segments to the receiver. The sender will receive ACK segments from the receiver through this port. We recommend using a random port number between 49152 to 65535 (dynamic port number range) for the sender and receiver ports.
2. `receiver_port`: the UDP port number on which receiver is expecting to receive STP segments from the sender. The receiver should send ACK segments through this port to the sender. We recommend using a random port number in the same range noted above.
3. `FileToSend.txt`: the name of the text file that must be transferred from sender to receiver using your reliable transport protocol. You may assume that the file included in the argument will be available in the current working directory of the sender with the "read" access permissions set (execute "`chmod +r FileToSend.txt`" at the terminal in the directory containing the file).
4. `max_win`: the maximum window size in bytes for the sender window. This should be an unsigned integer. Effectively, this is the maximum number of data bytes that the sender can transmit in a pipelined manner and for which ACKs are outstanding. `max_win` must be greater than or equal to 1000 bytes (MSS) and does not include STP headers. When `max_win` is set to 1000 bytes, STP will effectively behave as a stop-and-wait protocol, wherein the sender transmits one data segment at any given time and waits for the corresponding ACK segment. While testing, we will ensure that `max_win` is a multiple of 1000 bytes (e.g., 5000 bytes).
5. `rto`: the value of the retransmission timer in milliseconds. This should be an unsigned integer.

The sender should be initiated as follows:

If you use Java:

```
java Sender sender_port receiver_port FileToSend.txt max_win rto
```

If you use C:

```
./sender sender_port receiver_port FileToSend.txt max_win rto
```

If you use Python 3:

```
python3 sender.py sender_port receiver_port FileToSend.txt max_win rto
```

During testing, we will ensure that the 5 arguments provided are in the correct format. We will not test for erroneous arguments, missing arguments, etc. That said, it is good programming practice to check for such input errors.

The main code for the receiver should be contained in the following files: `receiver.c`, or `Receiver.java` or `receiver.py`. You may create additional files such as header files or other class files and name them as you wish.

The receiver should accept the following five arguments:

1. `receiver_port`: the UDP port number to be used by the receiver to receive STP segments

from the sender. This argument should match the second argument for the sender.

2. `sender_port`: the UDP port number to be used by the sender to send STP segments to the receiver. This argument should match the first argument for the sender.
3. `FileReceived.txt`: the name of the text file into which the text sent by the sender should be stored (this is the file that is being transferred from sender to receiver). You may assume that the receiver program will have permission to create files in its working directory (execute "`chmod +w .`" at the terminal to allow the creation of files in the working directory) and that a file with this name does not exist in the working directory.
4. `flp`: forward loss probability, which is the probability that any segment in the forward direction (Data, FIN, SYN) is lost. This should be a float value between 0 and 1 (inclusive). If `flp` is 0.1, then the receiver will drop about 10% of the segments that it receives from the sender.
5. `rlp`: reverse loss probability, which is the probability of a segment in the reverse direction (i.e., ACKs) being lost. This should be a float value between 0 and 1 (inclusive). If `rlp` is 0.05, then the receiver will drop about 5% of the ACK segments generated.

The receiver should be initiated as follows:

If you use Java:

```
java Receiver receiver_port sender_port FileReceived.txt flp rlp
```

If you use C:

```
./ receiver receiver_port sender_port FileReceived.txt flp rlp
```

If you use Python 3:

```
python3 receiver.py receiver_port sender_port FileReceived.txt flp rlp
```

During testing, we will ensure that the 5 arguments provided are in the correct format. We will not test for erroneous arguments, missing arguments, etc. That said, it is good programming practice to check for such input errors.

The receiver must be initiated before initiating the sender. The two programs will be executed on the same machine. Pay attention to the order of the port numbers to be specified in the arguments for the two programs as they are in reverse order (sender port is first for the sender while receiver port is first for the receiver). If you receive an error that one or both port numbers are in use, then choose different values from the dynamic port number range (49152 to 65535) and try again.

The sender and receiver should exit after the file transfer is complete and the required information as stated in the subsequent sections of this document is written to the sender and receiver log files.

3.2 Segment Format

STP segments must have 2 *two*-byte fields: "type" and "seqno" headers. Each of these store unsigned integer values.

The "type" field takes on 5 possible values. DATA = 0, ACK = 1, SYN = 2, FIN = 3, RESET = 4.

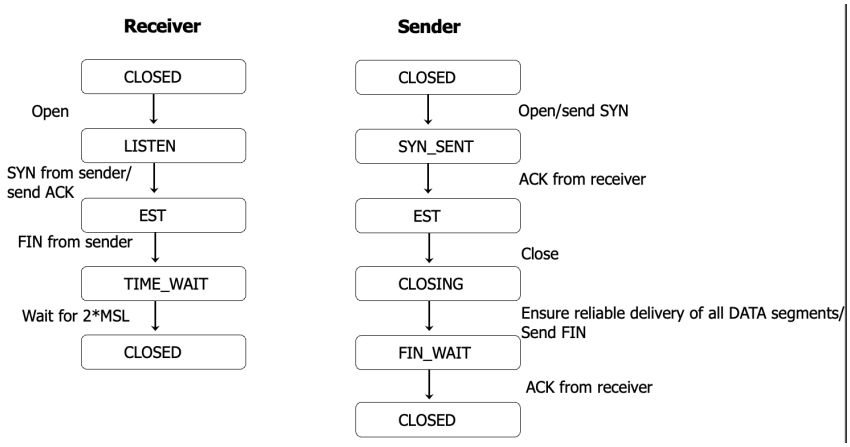
Unlike TCP, in which multiple types can be set simultaneously, STP segments must be of exactly one of the types specified above.

The "seqno" field indicates the sequence number of the segment. This field is used in all segments except RESET segment when it is set to zero. For DATA segments, the sequence number increases by the size (in bytes) of each segment. For ACK segments, the sequence number acts as a cumulative acknowledgment, and indicates the number of the next byte expected by the receiver. For SYN segments, the sequence number is the initial sequence number (ISN), which should be a randomly chosen integer between 0 to 2^16 - 1, which is the maximum sequence number. The sequence number of the first DATA segment of the connection should thus be ISN+1. For FIN packets, the sequence number is one larger than the sequence number of the last byte of the last data segment of the connection. The Maximum Segment Size (MSS) (excluding headers) for a STP segment is 1000 bytes. A DATA segment can thus be up to 1004 bytes long. The last DATA segment for the file being transferred may contain less than 1000 bytes as the file size may not be a multiple of 1000 bytes. All segments excluding DATA segments should only contain the headers and must thus be 4 bytes long.

The logic for determining the sequence number and ack number in STP is like TCP. However, STP does not use a separate ack number header field. Rather, the “seqno” field contains the ack number for the ACK segments.

3.3 State Diagram

The asymmetry between sender and receiver leads to somewhat different state diagrams for the two endpoints. The state diagram for STP is shown below, which depicts the normal behaviour for both end points.



The receiver can be in four possible states: CLOSED, LISTEN, ESTABLISHED and TIME_WAIT. Initially, it is in the CLOSED state. Upon issuing a passive open, it enters the LISTEN state. Note that the receiver is the passive host in our protocol and is initiated first, while the sender is initiated next and actively opens the connection. While in the LISTEN state, the receiver waits for a SYN packet to arrive on the correct port number. When it does, it responds with an ACK, and moves to the ESTABLISHED state. The ACKs sent by the receiver are cumulative (like TCP). After the sender has reliably transmitted all data (and received acknowledgments), it will send a FIN segment to the receiver. Upon receipt of the FIN, the receiver moves to the TIME_WAIT state. As in TCP, it remains in TIME_WAIT for two maximum segment lifetimes (MSLs) before re-entering the CLOSED state. This is to ensure that the receiver can respond to potentially retransmitted FIN segments from the sender. You may assume that the MSL is 1 seconds. In other words, the receiver should remain in TIME_WAIT for 2 seconds and then transition to CLOSED.

The sender can be in five possible states: CLOSED, SYN_SENT, ESTABLISHED, CLOSING and FIN_WAIT. Like the receiver, the sender starts in the CLOSED state. It then issues an active open by sending a SYN segment (to the receiver's port), thus entering the SYN_SENT state. This SYN transmission also includes the initial sequence number (ISN) of the conversation. The ISN should

be chosen at random from the valid range of possible sequence numbers (0 to 2^16 – 1). If a corresponding ACK is not received within `rto` msec, the sender should retransmit the SYN segment. If the SYN segment is not acknowledged after three retransmission attempts, a RESET segment must be sent to the destination port and the sender moves to the CLOSED state. In the common case in which the SYN is acknowledged correctly (the ACK must have the correct sequence number = ISN + 1), the sender enters the ESTABLISHED state and starts transmitting DATA segments. The sender maintains a single timer (for `rto` msec) for the oldest unacknowledged packet and only retransmits this packet if the timer expires. When the sending application (sitting above STP) is finished generating data, it issues a "close" operation to STP. This causes the sender to enter the CLOSING state. At this point, the sender must still ensure that any buffered data arrives at the receiver reliably. Upon verification of successful transmission, the sender sends a FIN segment with the appropriate sequence number (1 greater than the sequence number of the last data byte) and enters the FIN_WAIT state. Once the FIN segment is acknowledged, the sender re-enters the CLOSED state. If an ACK is not received before the timer (`rto` msec) expires, the sender should retransmit the FIN segment. If the FIN segment is not acknowledged after three retransmission attempts, the sender should send a RESET segment and return to the CLOSED state.

Strictly speaking, you don't have to implement the CLOSED state at the start for the sender. Your sender program when executed can immediately send the SYN segment and enter the SYN_SENT state. Also, when the sender is in the FIN_WAIT state and receives the ACK for the FIN segment, the program can simply exit. This is because the sender only transmits a single file in one execution and quits following the reliable file transfer.

Unlike TCP which follows a three-way handshake (SYN, SYN/ACK, ACK) for connection setup and independent connection closures (FIN, ACK) in each direction, STP follows a two-way connection setup (SYN, ACK) and one directional connection closure (FIN, ACK) process. The setup and closure are always initiated by the sender.

If one end point detects behaviour that is unexpected, it should reset the connection (i.e., close the connection) by sending a RESET segment. For example, if the receiver receives a data segment while it is in the SYN state (where it is expecting a SYN segment). A message should be printed to the terminal indicating that the connection is being reset. The state transition diagram on the previous page does not capture such erroneous scenarios. **Note that, we will NOT be rigorously testing your code for such unexpected behaviour.**

3.4 List of features to be implemented by the sender

You are required to implement the following features in the sender (and equivalent functionality in the receiver).

1. The sender should first open a UDP socket on `sender_port` and initiate a two-way handshake (SYN, ACK) for the connection establishment. The sender sends a SYN segment, and the receiver responds with an ACK. This is different to the three-way handshake implemented by TCP. If the ACK is not received before a timeout (`rto` msec), the sender should retransmit the SYN. If the SYN segment is not acknowledged after three retransmission attempts, a RESET segment must be sent to the receiver and the sender moves to the CLOSED state.

2. The sender must choose a random initial sequence number (ISN) between 0 and 2^{16}-1. Remember to perform sequence number arithmetic modulo 2^{16}. The sequence numbers should cycle back to zero after reaching $2^{16} – 1$.

3. A one-directional (forward) connection termination (FIN, ACK). The sender will initiate the connection close once the entire file has been reliably transmitted by sending the FIN segment and the receiver will respond with an ACK. This is different to the bi-directional close implemented by

TCP. If the ACK is not received before a timeout (`rto` msec), the sender should retransmit the FIN. The sender should terminate after connection closure. If the FIN segment is not acknowledged after three retransmission attempts, a RESET segment must be sent to `receiver_port` and the sender moves to the CLOSED state.

4. STP implements a sliding window protocol like TCP, whereby multiple segments can be transmitted by the sender in a pipelined manner. The sender should maintain a buffer to store all unacknowledged segments. The total amount of data that the sender can transmit in a pipelined manner and for which acknowledgments are pending is limited by `max_win`. Similar to TCP, as the sender receives ACK segments, the left edge of the window can slide forward, and the sender can transmit the next data segments (if there is pending data to be sent).

5. Each STP segment transmitted by the sender (including Data, SYN, FIN) must be encapsulated in a UDP segment and transmitted through the UDP socket.

6. The sender must maintain a single timer for retransmission of data segments (Section 3.5.4 of the text). The value of the timeout will be supplied to as an input argument to the sender program (`rto` msec). This timer is for the oldest unacknowledged data segment. In the event of a timeout, **only** the oldest unacknowledged data segment should be retransmitted (like TCP). The sender should not retransmit all unacknowledged segments. Remember that you are NOT implementing Go-Back-N.

7. The sender should implement all the features mentioned in Section 3.5.4 of the text, except for doubling the timeout. You are expected to implement the functionality of the simplified TCP sender (Figure 3.33 of the text) and fast retransmit (i.e., the sender should retransmit the oldest unacknowledged data segment on three duplicate ACKs) (pages 247-248).

8. The use of the "seqno" field was outlined in Section 3.2. For data segments, the sequence number increases by the size (in bytes) of each segment. For ACK segments, the sequence number acts as a cumulative acknowledgment, and indicates the number of the next byte expected by the receiver. The logic is thus like TCP, except that STP does not use a separate ACK header field. The ACK segments use the seqno header field to indicate the ACK numbers.

9. The sender will receive ACK segment from the receiver through the same socket, which the sender uses to transmit data. The ACK segment will be encapsulated in a UDP segment. The sender must first extract the ACK segment from the UDP segment and then process it as per the operation of the STP protocol. ACK segments have the same format as data segments but do not contain any data.

9. The sender should maintain a log file titled `Sender_log.txt` where it records the information about each segment that it sends and receives. You may assume that the sender program will have permission to create files in its current working directory. Start each entry on a new line. The format should be as follows:

```
<snd/rcv> <time> <type of packet> <seq-number> <number-of-bytes>
```

where <type of packet> could be SYN, ACK, FIN, DATA and RESET and the fields should be tab separated. Time should be in milliseconds and relative to when the SYN segment was sent – i.e., the SYN segment will always be sent at time 0. The number of bytes should be zero for all segments other than data segments. The receive window should be zero for all segments other than ACK segments.

For example, the following shows the log file for a sender that transmits 3000 bytes of data and the `rto` is 100 msec. The ISN chosen is 4521 and `max_win` is 3000 bytes. Notice that the third data packet is dropped and is hence retransmitted after a timeout interval of 100 msec.

```
snd  0          SYN  4521 0
```

```
rcv  10.34     ACK  4522 0
snd  10.45     DATA 4522 1000
snd  10.55     DATA 5522 1000
snd  10.67     DATA 6522 1000
rcv  36.76     ACK  5522 0
rcv  37.87     ACK  6522 0
snd  110.67    DATA 6522 1000
rcv  140.23    ACK  7522 0
snd  141.11    FIN  7522 0
rcv  176.34    ACK  7523 0
```

Once the entire file has been transmitted reliably and the connection is closed, the sender should also print the following statistics at the end of the log file (i.e., `Sender_log.txt`):

- Amount of (original) Data Transferred (in bytes) (excluding retransmissions)
- Number of Data Segments Sent (excluding retransmissions)
- Number of Retransmitted Data Segments
- Number of Duplicate Acknowledgements received

NOTE: Generation of this log file is very important. It will help your tutors in understanding the flow of your implementation and marking. So, if your code does not generate any log files, you will only be graded out of 25% of the marks.

The sender should finish execution after the file transfer is complete.

The sender should not print any output to the terminal. If you are printing output to the terminal for debugging purposes, make sure you disable it prior to submission.

3.5 Specific details about the receiver

1. The receiver should first open a UDP socket on `receiver_port` and then wait for segments to arrive from the sender. The first segment to be sent by the sender is a SYN segment and the receiver will reply with an ACK segment.

2. The receiver should next create a new text file called `FileReceived.txt`. You may assume that the receiver program will have permission to create files in its current working directory (execute "`chmod +w .`" at the terminal to allow the creation of files in the working directory). The received data will be written to this file in the correct order.

3. The receiver should initialise a receive window to store all received data. You may initialise it to be a large value (e.g., 16KB). It should be large enough to hold all the data that the sender can send in a pipelined manner (i.e., `max_win`). We will not use very large values for `max_win` in our tests.

4. The receiver should generate an ACK immediately after receiving any segment from the sender. The receiver should not follow Table 3.2 of the textbook and does not implement delayed ACKs. The format of the ACK segment is exactly like the STP data segment. It should however not contain any data. The ack number should be included in the "seqno" field of the STP segment. There is no explicit ACK field in the STP header.

5. The receiver should buffer all out-of-order data in the receive buffer. This is because STP implements reliable in-order delivery of data.

6. The receiver should write data (in correct order) from the receive buffer to the file, `FileReceived.txt`. At the end of the transfer, the receiver should have a duplicate of the text file sent by the sender. You can verify this by using the diff command on a Linux machine (`diff`

`FileReceived.txt FileToSend.txt`). When testing your program, if you have the sender and receiver executing in the same working directory then make sure that the file name provided as the argument to the receiver is different from the file name used by the sender.

7. The receiver program should emulate the behaviour of an unreliable communication channel between the sender and receiver. UDP segments can occasionally experience loss in a network, but the likelihood is very low when the sender and receiver and executed on the same machine. Moreover, to properly test the implementation of your sender program, we would like to control the unreliable behaviour of the underlying channel. The receiver emulates loss of STP segments in both directions which can be controlled through two command line argument: (i) `flp`: determines the probability of a segment (data, SYN, and FIN) in the forward direction from the sender being dropped. In other words, each segment arriving at the receiver socket will be dropped with a probability `flp`. If the packet is not dropped, then it will be processed as per the STP protocol. (ii) `rlp`: determines the probability of an ACK packet created by the receiver being dropped. In other words, each ACK segment created by the receiver will be dropped with a probability `rlp`. If the ACK segment is not dropped, then it will be transmitted through the socket to the sender.

Note about Random Number Generation

You will need to generate random numbers to implement segment loss. If you have not learnt about the principles behind random number generators, you need to know that random numbers are in fact generated by a deterministic formula by a computer program. Therefore, strictly speaking, random number generators are called pseudo-random number generators because the numbers are not truly random. The deterministic formula for random number generation in Python, Java and C uses an input parameter called a *seed*. If a fixed seed is used, then the same sequence of random numbers will be produced, each time the program is executed. This will thus likely generate the same sequence of segment loss in each execution of the receiver. While this may be useful for debugging purposes, it is not a realistic representation of an unreliable channel. Thus, you must ensure that you do not use a fixed seed in your submitted program. A simple way to use a different seed for each execution is to base the seed on the system time.

The following code fragment in Python, Java and C generate random numbers between 0 and 1 with a different seed in each execution.

- In Python, you initialise a random number generator by using `random.seed();`. By default, the random number generator uses the current system time. After that you can generate a random floating point number between (0,1) by using `random.random();`
- In Java, you initalise a random number generator by using `Random random = new Random();`. This constructor sets the seed of the random number generator to a value very likely to be distinct from any other invocation of this constructor. After that, you can generate a random floating point number between (0,1) by using `float x = random.nextFloat();`
- In C, you initalise a random number generator by using `srand(time(NULL));`. After that, you can generate a random floating point number between (0,1) by using `float x = rand()/((float)(RAND_MAX)+1);` Note that, `RAND_MAX` is the maximum value returned by the `rand()` function.

8. Once the file transfer is complete, the receiver should follow the state transition process as outlined in Section 3.3 while implementing connection closer. Pay particular attention to the transition from the TIME_WAIT state to the CLOSED state.

9. The receiver should also maintain a log file titled `Receiver_log.txt` where it records the information about each segment that it sends and receives. Information about dropped segments should also be included. The format should be exactly similar to the sender log file as outlined in

the sender specification (Section 3.4) with tab separated fields. Time should be in milliseconds and relative to when the SYN segment is received – i.e., the SYN segment will always be received at time 0. One difference from the sender is that the receiver log will also record any dropped segments as `drp`.

For example, the following shows the log file for a receiver that matches the scenario outlined in the sender specification (Section 3.4). To recall, the sender, transmits 3000 bytes of data and the `rto` is 100 msec. The ISN is 4521 and `max_win` is 3000 bytes. Recall that the third data packet is dropped.

```
rcv  0          SYN  4521 0
snd  0.34       ACK  4522 0
rcv  10.65      DATA 4522 1000
snd  10.75      ACK  5522 0
rcv  10.95      DATA 5522 1000
snd  11.03      ACK  6522 0
drp  11.06      DATA 6522 1000
rcv  115.4      DATA 6522 1000
snd  117.5      ACK  7522 0
rcv  143.4      FIN  7522 0
snd  144.5      ACK  7523 0
```

Once the entire file has been transmitted reliably and the connection is closed (remember to follow the state machine for the receiver), the receiver will also print the following statistics at the end of the log file (i.e., `Receiver_log.txt`):

- Amount of (original) Data Received (in bytes) – does not include retransmitted data
- Number of (original) Data Segments Received
- Number of duplicate Data segments received (if any)
- Number of Data segments dropped
- Number of ACK segments dropped

The receiver should finish execution after the file transfer is complete.

The receiver will only print a message altering of a closure of the connection due to a RESET packet to the terminal. No other output will be displayed.

3.6 Features excluded

There are several transport layer features adopted by TCP that are excluded from this assignment:

1. You do not need to implement timeout estimation. The timer value is provided as a command line argument (`rto` msec).
2. You do not need to double timeout interval.
3. You do not need to implement flow control and congestion control.
4. STP does not have to deal with corrupted or reordered segments. Segments will rarely be corrupted and reordered when the sender and receiver are executing on the same machine. In short, it is safe for you to assume that packets are only lost. Note however, that segments can be dropped by the unreliable channel implemented in the receiver program and thus segments may arrive out of order at the receiver.

3.7 Implementation Details

The picture below provides a high-level and simplified view of the assignment. The STP protocol logic implements the state maintained at the sender and receiver which includes all the state

variables and buffers. Note that each STP segment (in each direction) must be encapsulated in a UDP segment and transmitted through the UDP socket at each end point.

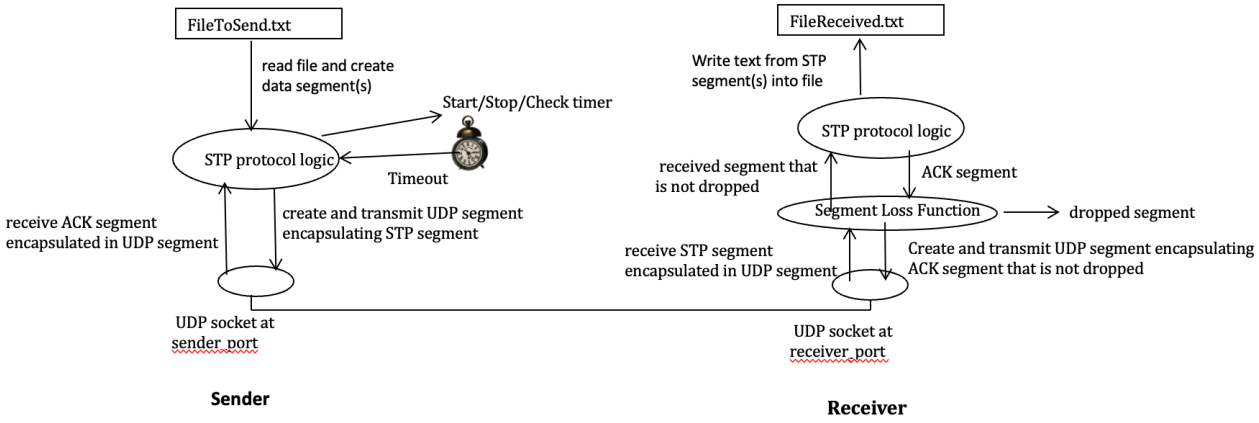


Figure 2: The overall structure of your assignment

Sender Design

The Sender must first execute connection setup, followed by data transmission and finally correction teardown as outlined in the state diagram description (Section 3.3). During connection setup, the sender transmits a SYN segment, starts a timer (`rto`) and waits for an ACK. During data transmission, the sender may transmit several STP data segments (determined by `max_win`), all of which need to be buffered (in case of retransmissions) and wait for the corresponding ACKs. A timer (`rto`) should be started for the oldest unacknowledged segment. During connection teardown, the sender must transmit a FIN segment, start a timer (`rto`) and wait for an ACK. Each STP segment (of any kind) that is to be transmitted must be encapsulated in a UDP segment and sent through the sender socket to the receiver. The sender should also process incoming ACK segments from the receiver. In the case of a timeout, the sender should retransmit the SYN or FIN if in the connection setup and teardown process (or send a RESET after three failed retransmission attempts) or retransmit the oldest unacknowledged segment if in the data transmission process. **As the sender needs to deal with multiple events, we recommend using multi-threading**.

Receiver Design

Recall that the receiver needs to implement the functionality of an unreliable channel that can drop segments in either direction. Upon receipt of a UDP segment through the socket, the receiver should extract the STP segment which is encapsulated within the UDP segment. The receiver should next call the function that simulates segment loss with a probability `flp`. If the segment is dropped, then nothing else needs to be done (other than updating the log). Remember that we are emulating loss of the incoming segment on the channel, thus, effectively this segment never arrived at the receiver and thus no action is necessary. If the segment is not dropped, then the receiver should execute the STP protocol logic (as outlined in Sections 3.5 and 3.3). For a SYN or FIN segment, the corresponding ACK should be sent, and other actions should be taken as per the state diagram shown in Section 3.3. For a Data segment, the data should be written to the receive buffer. If the data is received in order, then it can be written to the file, else if out of order, then it will remain in the buffer until the missing data is received. An appropriate ACK segment should be generated. The receiver next calls the function that simulates ACK segment loss with a probability `rlp`. If the ACK is to be dropped, then it should not be transmitted and nothing else needs to be done. Here we are emulating a scenario where the ACK segment is transmitted but dropped by the underlying channel and thus never reaches the sender. If the ACK is not dropped, then it should be encapsulated in a UDP segment and sent through the receiver socket to the sender.

All the above functionalities can be implemented in a single thread. However, the receiver must implement a timer during connection closer for transitioning from the TIME_WAIT state to the CLOSED state. **The suggested way to implement this is using multi-threading where a separate thread is used to manage the timer**. It may be possible to implement this using a single thread and using non-blocking or asynchronous I/O by using polling, i.e., select().

Data Structures

To manage your protocol, you will need a collection of state variables to help you keep track of things like state transitions, sliding windows and buffers. In TCP, this data structure is referred to as a control block: it's probably a good idea to create a control block class of your own and have a member variable of this type in your primary class. While we do not mandate the specifics, it is critical that you invest some time into thinking about the design of your data structures. You should be particularly careful about how multiple threads will interact with the various data structures.

6. Additional Notes

- This is NOT group assignment. You are expected to work on this individually.
- **Starter Code**: We have provided starter code in all 3 languages on the assignment page. You are welcome to use that to get started. Sample text files are also provided. We will use different files for our tests.
- **Assignment Help Sessions:** We will organise help sessions for all 3 programming languages to help you get started and provide you an opportunity to ask specific questions about the assignment. The details will be announced soon on the assignment page. Note that, this is not a forum for tutors to debug your code.
- **Tips on getting started**: The best way to tackle a complex implementation task is to do it in stages. A good starting point is to implement the functionality required for a stop-and-wait protocol (version rdt3.0 from the textbook and lectures), which sends one segment at a time. If you set the `max_win` argument to 1000 bytes (equal to the MSS) for the sender, then it will effectively operate as a stop-and-wait receiver as the sender window can only hold 1 data segment. You can first test with the loss probabilities (`flp`, `rlp`) set to zero to simulate a reliable channel. Once you verify that your protocol works correctly for this setting, you can increase the values for the loss probabilities to test that the sender can work as expected over a channel that losses packets (you may do this progressively, i.e., first only allow for packet loss in the forward direction, then only allow for packet loss in the reverse direction and finally test with packet loss in both directions). Test comprehensively with different loss probabilities to ensure that your sender works correctly.
- You can next progress to implement the full functionality of STP, wherein the sender should be able to transmit multiple packets in a pipelined manner (i.e., sliding window). First consider the case where the underlying channel is reliable (`flp` and `rlp` are set to 0). Set `max_win` to be a small multiple of the MSS (e.g., 4000 bytes). Once you verify that your protocol works correctly for this setting, you can increase the values for the loss probabilities to test that the sender can work as expected over a channel that losses packets (you may do this progressively, i.e., first only allow for packet loss in the forward direction, then only allow for packet loss in the reverse direction and finally test with packet loss in both directions). Test comprehensively with different loss probabilities to ensure that your sender works correctly.
- You can refer to the following resources for multi-threading. Note that you won't need to implement very complex aspects of multi-threading for this assignment.
    - Python: https://www.tutorialspoint.com/python3/python_multithreading.htm
    - Java: https://www.javatpoint.com/how-to-create-a-thread-in-java
    - C: https://www.geeksforgeeks.org/multithreading-in-c/

- It is imperative that you rigorously test your code to ensure that all possible (and logical) interactions can be correctly executed. **Test, test, and test**.
- **Debugging**: When implementing a complex assignment such as this, there are bound to be errors in your code. We strongly encourage that you follow a systematic approach to debugging. If you are using an IDE for development, then it is bound to have debugging functionalities. Alternately you could use a command line debugger such as pbd (python), jdb (java) or gdb (c). Use one of these tools to step through your code, create break points, observe the values of relevant variables and messages exchanged, etc. Proceed step by step, check and eliminate the possible causes until you find the underlying issue. Note that, we won't be able to debug your code on the forum or even in the help sessions.
- **Backup and Versioning:** We strongly recommend you to back-up your programs frequently. CSE backups all user accounts nightly. If you are developing code on your personal machine, it is strongly recommended that you undertake daily backups. We also recommend using a good versioning system so that you can roll back and recover from any inadvertent changes. There are many services available for this which are easy to use. If you are using an online versioning system, such as GitHub then you MUST ensure that your repository is private. We will NOT entertain any requests for special consideration due to issues related to computer failure, lost files, etc.
- **Language and Platform**: You are free to use C, Java, or Python to implement this assignment. Please choose a language that you are comfortable with. The programs will be tested on CSE Linux machines. So please make sure that your entire application runs correctly in VLAB. This is especially important if you plan to develop and test the programs on your personal computers (which may possibly use a different OS or version or IDE). Note that CSE machines support the following: **gcc version 10.2, Java 11, Python 2.7 and 3.9. If you are using Python, please clearly mention in your report which version of Python we should use to test your code.** You may only use the basic socket programming APIs providing in your programming language of choice. You may not use any special ready-to-use libraries or APIs that implement certain functions of the spec for you. If you are unsure, it is best you check with the course staff on the forum.
- You are encouraged to use the course discussion forum to ask questions and to discuss different approaches to solve the problem. However, you should **not** post your solution or any code fragments on the forum.
- We will arrange for additional consultations in Weeks 7-10 to assist you with assignment related questions. Information about the consults will be announced via the website.

7. Assignment Submission

Please ensure that you use the mandated file name. You may of course have additional header files and/or helper files. If you are using C, then you MUST submit a makefile/script along with your code (not necessary with Java or Python). This is because we need to know how to resolve the dependencies among all the files that you have provided. After running your makefile we should have the following executable files: `sender` and `receiver`.

In addition, you should submit a small report, `report.pdf` (no more than 2 pages). Provide details of which language you have used (e.g., Python 3) and the organisation of your code (makefiles, directories if any, etc.). Your python must contain a brief discussion of how you have implemented the STP protocol. This should include the overall program design, data structure design and a brief description of the operation of the sender and receiver. Also discuss any design trade-offs considered and made. If your program does not work under any circumstances, report this

here. Also indicate any segments of code that you have borrowed from the Web or other books.

You are required to submit your source code and report.pdf. You can submit your assignment using the give command through VLAB. Make sure you are in the same directory as your code and report, and then do the following:

1. Type tar -cvf assign.tar filenames
e.g., `tar -cvf assign.tar *.java report.pdf`

2. When you are ready to submit, at the bash prompt type `3331`

3. Next, type: `give cs3331 assign assign.tar` (You should receive a message stating the result of your submission). The same command should be used for 3331 and 9331.

Alternately, you can also submit the tar file via the WebCMS3 interface on the assignment page.

Important notes

- The system will only accept `assign.tar` submission name. All other names will be rejected.
- **Ensure that your program/s are tested in the VLAB environment before submission. In the past, there were cases where tutors were unable to compile and run students' programs while marking. To avoid any disruption, please ensure that you test your program in the VLAB environment before submitting the assignment. Note that, we will be unable to award any significant marks if the submitted code does not run during marking.**
- You may submit as many times as you wish before the deadline. A later submission will override the earlier submission, so make sure you submit the correct file. Do not leave until the last moment to submit, as there may be technical, or network errors and you will not have time to rectify it.

Late Submission Penalty: Late penalty will be applied as follows:

- Up to 24 hours after deadline: 5% reduction
- More than 24 hours but less than 48 hours after deadline: 10% reduction
- More than 48 hours but less than 72 hours after deadline: 15% reduction
- More than 72 hours but less than 96 hours after deadline: 20% reduction
- More than 96 hours after deadline: NOT accepted.

NOTE: The penalty is applied to your final total. For example, if you submit your assignment 1 day late and your total marks are 10, then your final mark will be 10 – 0.5 (5% penalty) = 9.5.

8. Plagiarism

You are to write all of the code for this assignment yourself. All source codes are subject to strict checks for plagiarism, via highly sophisticated plagiarism detection software. These checks may include comparison with available code from Internet sites and assignments from previous semesters. In addition, each submission will be checked against all other submissions of the current semester. Do not post this assignment on forums where you can pay programmers to write code for you. We will be monitoring such forums. Please note that we take this matter quite seriously. The LIC will decide on appropriate penalty for detected cases of plagiarism. The most likely penalty would be to reduce the assignment mark to ZERO. We are aware that a lot of learning takes place in student conversations, and don't wish to discourage those. However, it is important, for both those helping others and those being helped, not to provide/accept any programming language code in writing, as this is apt to be used exactly as is, and lead to plagiarism penalties for both the supplier and the copier of the codes. Write something on a piece of paper, by all means, but tear it up/take it

away when the discussion is over. It is OK to borrow bits and pieces of code from sample socket code out on the Web and in books. You MUST however acknowledge the source of any borrowed code. This means providing a reference to a book or a URL when the code appears (as comments). Also indicate in your report the portions of your code that were borrowed. Explain any modifications you have made (if any) to the borrowed code.

Generative AI Tools: It is prohibited to use any software or service to search for or generate information or answers. If its use is detected, it will be regarded as serious academic misconduct and subject to the standard penalties, which may include 00FL, suspension and exclusion.

9. Marking Policy

You should test your program rigorously before submitting your code. Your code will be marked using the following criteria:

Test 1 - Stop and Wait over a Reliable Channel: 2 marks

We will test your STP implementation when executed as a stop and wait protocol and when the underlying channel is reliable.

We show the instantiation of the two programs assuming the implementation is in Python 3. The arguments will be similar for C and Java.

```
python3 receiver.py 56007 59606 FileToReceive.txt 0 0
```

```
python3 sender.py 59606 56007 test1.txt 1000 rto
```

We will test for different values of `rto` and with different text files. We will compare the received file with the sent file, check the sender and receiver logs and other checks to ensure that the STP protocol is correctly implemented at both end points.

Test 2 - Stop and Wait over an Unreliable Channel: 4 marks

Next, we will test your STP implementation while operating as a stop and wait protocol but where the underlying channel is unreliable.

In the first instance, we will only induce packet loss in the forward direction. The receiver will be instantiated as follows (sender will be instantiated as above):

```
python3 receiver.py 56007 59606 FileToReceive.txt flp 0
```

We will test for different values of `flp`, `rto` and with different text files. Checks will be undertaken as noted above. (**1 mark**)

In the second instance, we will only induce packet loss in the reverse direction. The receiver will be instantiated as follows (sender will be instantiated as above):

```
python3 receiver.py 56007 59606 FileToReceive.txt 0 rlp
```

We will test for different values of `rlp`, `rto` and with different text files. Checks will be undertaken as noted above. (**1 mark**)

In the final instance, we will induce packet loss in both directions. The receiver will be instantiated as follows (sender will be instantiated as above):

```
python3 receiver.py 56007 59606 FileToReceive.txt flp rlp
```

We will test for different values of `flp`, `rlp`, `rto` and with different text files. Checks will be undertaken as noted above. (**2 marks**)

Test 3 - Sliding Window over a Reliable Channel: 4 marks

We will test your STP implementation when executed as a sliding window protocol and when the underlying channel is reliable.

We show the instantiation of the two programs assuming the implementation is in Python. The arguments will be similar for C and Java.

```
python3 receiver.py 56007 59606 FileToReceive.txt 0 0
```

```
python3 sender.py 59606 56007 test1.txt max_win rto
```

We will test for different values of `max_win` (always a multiple of 1000), `rto` and with different text files. We will compare the received file with the sent file, check the sender and receiver logs and other checks to ensure that the STP protocol is correctly implemented at both end points.

Test 4 - Sliding Window over an Unreliable Channel: 8 marks

Next, we will test your STP implementation when executed as a sliding window protocol but where the underlying channel is unreliable.

In the first instance, we will only induce packet loss in the forward direction. The receiver will be instantiated as follows (sender will be instantiated as above):

```
python3 receiver.py 56007 59606 FileToReceive.txt flp 0
```

We will test for different values of `max_win`, `flp`, `rto` and with different text files. Checks will be undertaken as noted above. (**2 marks**)

In the second instance, we will only induce packet loss in the reverse direction. The receiver will be instantiated as follows (sender will be instantiated as above):

```
python3 receiver.py 56007 59606 FileToReceive.txt 0 rlp
```

We will test for different values of `max_win`, `rlp`, `rto` and with different text files. Checks will be undertaken as noted above. (**2 mark**)

In the final instance, we will induce packet loss in both directions. The receiver will be instantiated as follows (sender will be instantiated as above):

```
python3 receiver.py 56007 59606 FileToReceive.txt flp rlp
```

We will test for different values of `max_win`, `flp`, `rlp`, `rto` and with different text files. Checks will be undertaken as noted above. (**4 marks**)

Test 5 – Report: 1 mark

The report should not be longer than 2 pages. Provide details of which language you have used (e.g., Python 3) and the organisation of your code (makefiles, directories if any, etc.). Your report must contain a brief discussion of how you have implemented the STP protocol. This should include the overall program design, data structure design and a brief description of the operation of the sender and receiver. Also discuss any design trade-offs considered and made. If your program does not work under any circumstances, report this here. Also indicate any segments of code that you have borrowed from the Web or other books. We will verify that the description in your report confirms with the actual implementations in the programs.

Test 6 – Properly documented and commented code: 1 mark

We recommend following well-known style guides such as:

Java: https://google.github.io/styleguide/javaguide.html

Python: https://peps.python.org/pep-0008/

IMPORTANT NOTE: If your sender and receiver do not generate log files as indicated in the specification, you will only be graded out of 25% of the total marks (i.e., a 75% penalty will be assessed).