COMP3331/9331 Computer Networks and Applications

Assignment for Term 3, 2025

Version 1.3

Due: 4:59pm Monday, 17 November 2025 (Week 10)

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

1. Change Log

Version 1.0 released on 7th October 2025.
Version 1.1 released on 9th October 2025 — Marking Policy (Section 7) updated.
Version 1.2 released on 15th October 2025 — URP Monitor Program (Section 8) added.
Version 1.3 released on 5th November 2025 — Corrected "Total data received" in example Receiver log.

2. Goal and Learning Objectives

UDP provides a minimal, unreliable datagram service between hosts. When it was first introduced (RFC 768, 1980), UDP was intended as a lightweight alternative to TCP for applications that did not require reliable delivery, connection management, or flow/congestion control. Its design allowed applications to send datagrams directly with low overhead.

Over time, this simplicity has also made UDP a useful foundation for specialised transport protocols. For example, some multimedia services implement proprietary protocols on top of UDP, and QUIC—a modern transport protocol standardised by the IETF in 2021—also runs over UDP. QUIC incorporates features such as reliability, congestion control, and multiplexing, and today underpins HTTP/3, which is widely deployed across the Internet.

For this assignment, you will implement a reliable transport protocol over UDP, which we will refer to as the **UDP-based Reliable Protocol (URP)**. URP will include most (but not all) of the features described in Sections 3.5.4–3.5.6 of Computer Networking: A Top-Down Approach by Kurose and Ross (7th or 8th ed.), or the equivalent material from the Week 4/5 lecture notes. Examples include timeouts, acknowledgements (ACKs), sequence numbers, and sliding windows—features that are fundamental to many transport protocols. Implementing URP will give you hands-on experience with these building blocks.

URP ensures reliable, end-to-end delivery of data in the face of packet loss and corruption. Like TCP, it provides a byte-stream abstraction and supports pipelined data transfer using a sliding window with elements of both Go-Back-N (GBN) and Selective Repeat (SR). Unlike TCP, URP will not implement congestion control or flow control. URP will also be asymmetric: there will be two distinct protocol endpoints, **Sender** and **Receiver**. Data packets flow only in the forward direction (Sender → Receiver), while acknowledgements flow in the reverse direction (Receiver → Sender). Throughout this specification, "Sender" and "Receiver" (capitalised) refer to the protocol endpoints, while "sender" and "receiver" (lowercase) refer to the actions of sending and receiving.

Because URP must provide reliability over an unreliable channel, state must be maintained at both endpoints, and connection setup and teardown will be integral to the protocol. You will use URP to transfer a text file (examples provided on the assignment webpage) from the Sender to the Receiver.

To properly test your implementation, the Sender program must also emulate the behaviour of an unreliable channel. UDP segments rarely get lost or corrupted in our test environment (where both endpoints run on the same machine), so you will implement a Packet Loss and Corruption (PLC) module that deliberately introduces loss and corruption in both directions: (i) DATA, SYN, and FIN segments in the forward direction, and (ii) ACK segments in the reverse direction. This PLC is to be implemented on the Sender side only. You may assume that the channel will not reorder segments.

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

2.1 Learning Objectives

By 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 October (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

URP 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 programs. Let us reiterate this, **URP 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.

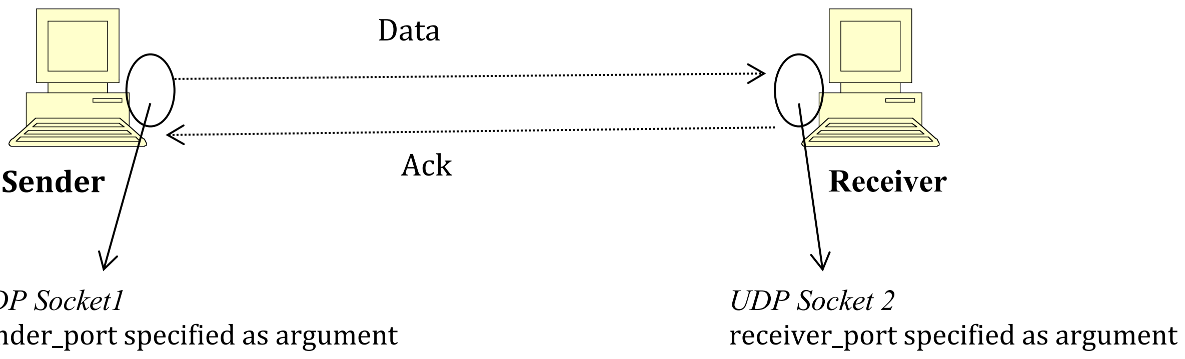
Data
Ack
Sender
Receiver
UDP Socket1
sender_port specified as argument
UDP Socket 2
receiver_port specified as argument

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.

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.

3.1 File Names

The main code for the Sender must be contained in one of 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 must accept the following nine arguments:

1. `sender_port`: the UDP port number to be used by the Sender to send URP segments to the Receiver. The Sender will receive ACK segments from the Receiver through this same port. During your testing, 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 through which the Receiver is expecting to receive URP segments from the Sender. The Receiver should send ACK segments to the Sender through this same port. During your testing we recommend using a random port number in the same range noted above.
3. `txt_file_to_send`: 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 txt_file_to_send`" at the terminal in the directory containing the file, where `txt_file_to_send` is set to the actual name of the file).
4. `max_win`: the maximum window size in bytes for the send 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 URP headers. When `max_win` is set to 1000 bytes, URP 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.
5. `rto`: the value of the retransmission timer in milliseconds. This should be an unsigned integer. While testing, we will ensure that `rto` is set to a reasonable value (e.g., at least 50 msec)

The remaining 4 arguments are used exclusively by the Packet Loss and Corruption (PLC) module:

6. `flp`: forward loss probability – the probability that any segment in the forward direction (DATA, FIN, SYN) is lost. This value must be a float between 0 and 1 (inclusive). For example, if `flp = 0.1`, then about 10% of the segments the Sender sends will be dropped by the PLC module.
7. `rlp`: reverse loss probability – the probability that a segment in the reverse direction (i.e., an ACK) is lost. This value must be a float between 0 and 1 (inclusive). For example, if `rlp = 0.05`, then the PLC module will drop approximately 5% of the ACK segments received.
8. `fcp`: forward corruption probability – the probability that any segment in the forward

direction (DATA, FIN, SYN), if not lost, is instead corrupted. This value must be a float between 0 and 1 (inclusive). For example, if `fcp = 0.1`, then about 10% of the non-dropped segments that the Sender transmits to the Receiver will be corrupted by the PLC module.

9. `rcp`: reverse corruption probability – the probability that a segment in the reverse direction (i.e., an ACK), if not lost, is instead corrupted. This value must be a float between 0 and 1 (inclusive). For example, if `rcp = 0.05`, then the PLC module will corrupt approximately 5% of the non-dropped ACK segments received.

The Sender should be initiated as follows:

If you use Java:

```
java Sender sender_port receiver_port txt_file_to_send max_win rto flp rlp fcp rcp
```

If you use C:

```
./sender sender_port receiver_port txt_file_to_send max_win rto flp rlp fcp rcp
```

If you use Python:

```
python3 sender.py sender_port receiver_port txt_file_to_send max_win rto flp rlp fcp rcp
```

During testing, we will ensure that the 9 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 one of 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 four arguments:

1. `receiver_port`: the UDP port number to be used by the Receiver to receive URP 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 URP segments to the Receiver. This argument should match the first argument for the Sender.
3. `txt_file_received`: the name of the text file into which the data 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 or may be overwritten.
4. `max_win`: the receive window size. This argument should match the fourth argument for the Sender. It is provided to the Receiver to ensure that the Receiver can initialise a buffer sufficient to hold all the data that the Sender can transmit in a pipelined manner.

The Receiver should be initiated as follows:

If you use Java:

```
java Receiver receiver_port sender_port txt_file_received max_win
```

If you use C:

```
./receiver receiver_port sender_port txt_file_received max_win
```

If you use Python:

```
python3 receiver.py receiver_port sender_port txt_file_received max_win
```

During testing, we will ensure that the 4 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. Also pay attention to the PLC parameters as both loss probabilities (forward and reverse) come before both corruption probabilities (forward and reverse).

The Sender and Receiver should exit after the file transfer is complete and their log files, as stated in the subsequent sections of this document, have been finalised.

3.2 Segment Format

A URP segment consists of a segment *header* and a *data* section. The segment header has a fixed length of 6 bytes and all fields must be set appropriately. The data section follows the header and is the payload data carried for the application. The length of the data section is not specified in the segment header; it can be calculated by subtracting the length of the segment header from the total length of the segment. All multi-byte fields in the segment header are represented in network byte order (big-endian), with the most significant byte first.

Offset	**Octet**	**0**								**1**							
Octet	**Bit**	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**
0	**0**	*Sequence Number*															
2	**16**	*Reserved*													*ACK*	*SYN*	*FIN*
4	**32**	*Error Detection*															
6	**48**	*Data*															
8	**64**	*(0 to MSS bytes)*															
10	**80**																
⋮	⋮																

Sequence Number (16 bits)

This field is used in every segment. Its meaning depends on the segment type:

DATA: The sequence number indicates the byte number of the first data byte in the segment relative to the connection's byte stream. For each segment carrying data, the sequence number increases by the number of payload bytes. Note that SYN and FIN control segments also consume one sequence number even though no data is sent.

ACK: The sequence number serves as a cumulative acknowledgment, indicating the next byte expected by the Receiver. (URP does not have a separate acknowledgment field.)

SYN: The sequence number is the initial sequence number (ISN), chosen randomly in the range 0 to $2^{16} - 1$, inclusive. The first DATA segment in the connection therefore has sequence number ISN + 1.

FIN: The sequence number is one greater than the sequence number of the last byte of the final DATA segment.

In URP, the logic for determining sequence and acknowledgment numbers is similar to TCP. However, because URP is unidirectional, each segment requires only one of these values. To avoid unnecessary overhead, URP reuses the single sequence number field to serve both roles.

Reserved (13 bits)

Must be set to zero.

Control Bits (3 bits, "flags")

Only one control bit may be set at a time. Unlike TCP, URP segments are always exactly one type: DATA, ACK, SYN, or FIN.

ACK (1 bit) — If set, the segment is an ACK. The sequence number field is interpreted as an acknowledgment number.

SYN (1 bit) — If set, the segment is a SYN. The sequence number is the initial sequence number (ISN).

FIN (1 bit) — If set, the segment is a FIN. The sequence number is one greater than the last byte number of the final DATA segment.

DATA — If no control bit is set, the segment is a DATA segment. The sequence number indicates the first byte of the payload.

Error Detection (16 bits)

This field provides protection against transmission errors in both the URP header and data. Its purpose is to allow the receiver of the segment to detect at least single-bit errors.

- You must make use of this field to implement an error-detection scheme of your choice.
- At minimum, the scheme should reliably detect single-bit errors in any segment.
- Possible approaches include a simple checksum (e.g., the 16-bit ones' complement sum used in TCP/UDP) or a parity-based scheme.
- The sender of the segment is responsible for calculating and setting the value of this field before transmission. The receiver of the segment must recompute and verify the value on arrival. If the check fails, the segment must be discarded without further processing.

Data (0 to MSS bytes)

The actual payload data carried in a DATA segment. The maximum segment size (MSS) for a DATA segment is 1000 bytes (payload only, excluding headers).

- With a 6-byte header, a DATA segment can therefore be up to 1006 bytes in total.
- The final DATA segment may be smaller if the file size is not a multiple of 1000 bytes.
- All non-DATA segments (SYN, ACK, FIN) consist only of the 6-byte header.

NOTE: A key requirement of this assignment is to implement the specified segment format. Submissions that deviate from this format will be capped at 50% of the available marks.

Aside: Network Byte Order

Computers store numbers in memory as a sequence of bytes. The catch is: not all computers agree on which end of the number goes first.

- **Big-endian** machines put the big end first — the most significant byte comes before the others.
- **Little-endian** machines do the opposite — the least significant byte comes first.

For example, the number `0x12345678` would be stored as:

- **Big-endian**: `12 34 56 78`
- **Little-endian**: `78 56 34 12`

This is a bit like writing dates: some countries write *day/month/year*, others write *month/day/year*. If two people don't agree, confusion follows.

To avoid this on the Internet, network protocols agree to use **big-endian order**, which is called *network byte order*.

In network applications you'll often see functions like `htonl()` (host-to-network-long) and `ntohl()` (network-to-host-long) used to do the conversion automatically. This ensures multi-byte values (such as 16-bit port numbers or 32-bit sequence numbers) are represented consistently, regardless of the CPU architecture

For this assignment, both the Sender and Receiver will be running on a big-endian system during marking, so you don't have to worry about conversions. But it's useful to know why those functions exist — they make it possible for computers to understand each other, big or little.

3.3 State Diagram

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

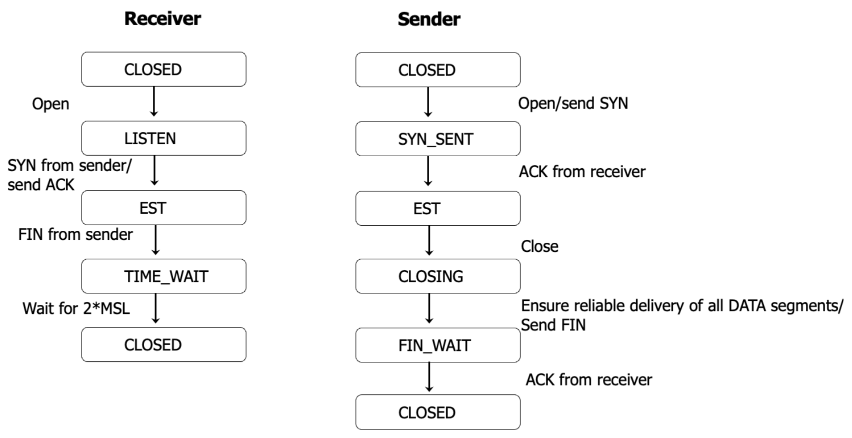

Figure 2: State diagram depicting the normal behaviour of URP.

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 segment to arrive on the correct port number. When it does, it responds with an ACK, and moves to the ESTABLISHED state. Should a duplicate SYN arrive while the

Receiver is in the ESTABLISHED state, it reissues the ACK and remains in 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 second. 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. In the common case in which the SYN is acknowledged correctly (the ACK must be error-free and 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. Once all DATA segments have been transmitted the Sender enters the CLOSING state. At this point, the Sender must still ensure that any buffered data arrives at the Receiver reliably. Upon acknowledgement 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 error-free ACK is not received before the timer (`rto` msec) expires, the Sender should retransmit the FIN segment.

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, URP 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 simply print a message to the terminal indicating that the connection is being reset and then terminate. For example, if the Receiver receives a DATA segment while it is in the LISTEN state (where it is expecting a SYN segment). The state transition diagram on the previous page does not capture such erroneous scenarios. You are free to specify the format of the message that is printed. **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 an error-free ACK is not received before a timeout (`rto` msec), the Sender should retransmit the SYN.

2. The Sender must choose a random initial sequence number (ISN) between 0 and $2^{16} - 1$, inclusive. 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 an error-free ACK is not received before a timeout (`rto` msec), the Sender should retransmit the FIN. The Sender should terminate after connection closure.

4. URP 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`. Like TCP, as the Sender receives error-free 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. The Sender should not assume that the entire file to be transferred will fit into available memory and therefore should not attempt to pre-construct all segments. Instead, it should read data incrementally from disk and construct segments dynamically as window space becomes available.

6. Each URP segment to be transmitted by the Sender (including DATA, SYN, FIN) must be encapsulated in a UDP datagram and sent through the UDP socket.

7. Before transmitting any segment (DATA, SYN, or FIN), the Sender must compute and set the error-detection field in the segment header.

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

9. The Sender should implement all the features mentioned in Section 3.5.4 of the text, except for doubling the timeout interval. 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 immediately retransmit the oldest unacknowledged data segment on three duplicate ACKs) (pages 247-248).

10. The use of the sequence number field was outlined in Section 3.2. For DATA segments, the sequence number increases by the size (in bytes) of each segment (excluding headers). 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 URP does not use a separate ACK header field. The ACK segments use the sequence number header field to indicate the ACK numbers.

11. The Sender will receive each 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 datagram. The Sender must first extract the ACK segment from the UDP segment and then process it as per the operation of the URP protocol. ACK segments have the same format as DATA segments but do not contain any data.

12. Upon receiving an ACK, the Sender must first validate the error-detection field in the segment header. If the ACK is invalid, it must be discarded and no further processing performed. Any unacknowledged segments (SYN, FIN, or DATA) remain unacknowledged, and the retransmission timer and duplicate ACK counter remain unchanged.

13. The Sender should emulate the behaviour of an unreliable communication channel between the Sender and Receiver, as described in Section 3.4.1.

14. The Sender must maintain a log file, as described in Section 3.4.2.

3.4.1 Packet Loss and Corruption (PLC) Module

UDP segments can occasionally be lost or corrupted in a network. However, when the Sender and Receiver run on the same machine, the likelihood of such errors is exceedingly low. To properly test the reliability of your URP implementation, we therefore need to introduce controlled unreliability into the channel. The PLC module must be implemented as part of your Sender program to emulate loss and corruption of URP segments in both directions, using the four command-line arguments — `flp`, `rlp`, `fcp`, and `rcp` — as described in Section 3.1.

The URP protocol must remain unaware of any losses or corruptions introduced by the PLC. When transmitting, URP assumes that any segment it passes to the PLC has entered the network. When receiving, URP assumes that only the segments delivered by the PLC have arrived; dropped segments are treated as if they were never received. Loss or corruption must be detected and handled solely by URP's reliability mechanisms, with no reliance on PLC behaviour.

For example, if the PLC drops a DATA segment produced by the Sender, URP assumes it was sent successfully and must later infer the loss. Conversely, if the PLC drops an ACK sent by the Receiver, URP believes that no ACK was ever received.

When the PLC module processes a fully constructed URP segment generated by the Sender, it should behave as follows:

1. **Drop:** With probability `flp`, the segment is dropped — that is, it is not encapsulated in a UDP datagram or sent through the socket. To decide whether to drop a segment, generate a random number in the range [0, 1). If the number is less than `flp`, drop the segment.
2. **Corrupt:** If the segment is not dropped, then with probability `fcp` it should be corrupted. To do this, select a random byte in the segment (excluding the first four header bytes, which simplifies logging) and flip a single bit within that byte.
3. **Transmit:** If the segment was not dropped, encapsulate the (possibly corrupted) URP segment in a UDP datagram and send it through the socket.

When the PLC module processes a UDP segment received from the socket, it should behave as follows:

1. **Drop:** With probability `rlp`, the segment is dropped — that is, it is not delivered to the URP protocol within the Sender.
2. **Corrupt:** If the segment is not dropped, then with probability `rcp` it should be corrupted. To do this, select a random byte in the segment (again excluding the first four header bytes) and flip a single bit within that byte.
3. **Deliver:** If the segment was not dropped, deliver the (possibly corrupted) URP segment to the URP protocol for processing.

Note about Random Number Generation

You will need to generate random numbers to establish an initial sequence number (ISN) and implement the Packet Loss and Corruption (PLC) module. 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 use 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 ISN and the same sequence of segment loss in each execution of the Sender. 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 fragments 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 in the range [0,1) by using `random.random()`.

- In Java, you initialise 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 in the range [0,1) by using `float x = random.nextFloat()`.

- In C, you initialise a random number generator by using `srand(time(NULL))`. After that, you can generate a random floating-point number in the range [0,1) by using `float x = rand()/((float)(RAND_MAX)+1)`. Note that, `RAND_MAX` is the maximum value returned by the `rand()` function.

3.4.2 Sender Log File

The Sender must maintain a log file in its current working directory named `sender_log.txt`, recording information about every segment it sends and receives, including segments that are dropped or corrupted. You may assume the Sender has permission to create files in its current working directory. If the file already exists, it should be overwritten. Each log entry must start on a new line. The format is as follows:

```
<direction> <status> <time> <segment-type> <seq-number> <payload-length>
```

There must be no whitespace within each field, and all fields must be separated by whitespace. You may choose the exact formatting (padding, spaces, or tabs), but the log file must be easily human-readable.

The six fields are defined as follows:

1. **Direction**: From the perspective of the Sender endpoint. The Sender will `snd` DATA, FIN, and SYN segments, and `rcv` ACK segments.
2. **Status**: The action taken by the PLC. Segments forwarded unchanged are marked `ok`, dropped segments `drp`, and corrupted segments `cor`.
3. **Time**: Time in milliseconds relative to when the first SYN segment was sent (even if it was dropped). The first entry will therefore have a time of 0.
4. **Segment type**: One of `ACK`, `DATA`, `SYN`, or `FIN`.
5. **Sequence number**: The value in the segment's sequence number header field.
6. **Payload length**: Number of bytes of payload data in the segment. This should be zero for all segments except DATA.

Log entries must be recorded in chronological order.

The following is an example log file for a Sender transmitting a 3500-byte file. The retransmission timeout (`rto`) is 100 msec, the maximum window size (`max_win`) is 3000 bytes, and the randomly generated initial sequence number (ISN) is 63999. Notice that the ACK for the initial SYN is dropped, and the SYN is retransmitted after approximately 103 msec. Small deviations of retransmission times from the `rto` are expected due to system latency and are considered acceptable.

```
snd  ok     0.00  SYN   63999     0
rcv  drp    0.37  ACK   64000     0
snd  ok   102.93  SYN   63999     0
rcv  ok   103.21  ACK   64000     0
snd  cor  104.52  DATA  64000  1000
snd  ok   104.53  DATA  65000  1000
snd  ok   104.56  DATA    464  1000
rcv  ok   104.78  ACK   64000     0
rcv  drp  104.79  ACK   64000     0
snd  drp  205.65  DATA  64000  1000
snd  ok   310.16  DATA  64000  1000
rcv  ok   310.58  ACK    1464     0
snd  ok   310.64  DATA   1464   500
rcv  ok   310.92  ACK    1964     0
snd  ok   310.95  FIN    1964     0
rcv  cor  311.20  ACK    1965     0
snd  cor  415.59  FIN    1964     0
snd  ok   517.98  FIN    1964     0
rcv  ok   518.35  ACK    1965     0
```

You are encouraged to write to the log file in real time. If the Sender buffers log output and waits until the transfer is complete to write the file, any failure during the Sender's operation could prevent crucial information from being recorded. Real-time logging can provide valuable insights even if an error occurs.

Once the entire file has been reliably transmitted and the connection is closed, the Sender should also write the following statistics at the end of the log file (i.e., `sender_log.txt`). Please note the amount of data is strictly payload data. You're encouraged to track these statistics in real-time at their logical code points within the URP protocol or PLC module, rather than try to pre- or post-compute them.

These statistics are from the perspective of the URP protocol, without direct knowledge of any actions by the PLC:

1. **Original data sent** (in bytes, only payload data, does not count any retransmissions)
2. **Total data sent** (in bytes, only payload data, also includes retransmissions)
3. **Original segments sent** (does not count any retransmissions)
4. **Total segments sent** (also includes retransmissions)
5. **Timeout retransmissions**
6. **Fast retransmissions**
7. **Duplicate acks received** (does not count corrupted acks)
8. **Corrupted acks discarded**

These statistics are from the perspective of the PLC module:

9. **PLC forward segments dropped**
10. **PLC forward segments corrupted**
11. **PLC reverse segments dropped**
12. **PLC reverse segments corrupted**

For the example scenario, the Sender log would be appended with:

```
Original data sent:           3500
Total data sent:              5500
Original segments sent:          6
Total segments sent:            11
Timeout retransmissions:         5
Fast retransmissions:            0
Duplicate acks received:         1
Corrupted acks discarded:        1
PLC forward segments dropped:    1
PLC forward segments corrupted:  2
PLC reverse segments dropped:    2
PLC reverse segments corrupted:  1
```

You must follow the above format, but you may choose how to separate each statistical description with its value, e.g. with padding or a fixed number of spaces/tabs, provided the summary statistics are easily human-readable.

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.

Once the file transfer is complete the Sender should finalise the log file and terminate.

The Sender must only print a message to the terminal if it is terminating prematurely, due to unexpected behaviour by the Receiver. No other output should be displayed. 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 named as per the `txt_file_received` argument. 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 of `max_win` to hold all data that the Sender can send in a pipelined manner.

4. The Receiver should generate an ACK immediately after receiving any error-free 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 URP DATA segment. It should however not contain any data. The acknowledgement number should be included in the sequence number field of the URP segment. There is no explicit acknowledgement number field in the URP header.

5. When a segment is received (SYN, FIN, or DATA), the Receiver must first check the segment for errors using the error detection field in the segment header. If the segment is in error, it must be discarded (including any data payload), and no ACK is sent.

6. Before transmitting an ACK, the Receiver must compute and set the error-detection field in the ACK's segment header.

7. The Receiver should buffer all out-of-order data in the receive buffer. This is because URP

implements reliable in-order delivery of data.

8. The Receiver should write data (in correct order) from the receive buffer to the `txt_file_received` file. To ensure the receive buffer does not overflow, the file should be written as correctly ordered data is available. As with the Sender, the Receiver should not assume that the entire file will fit within available memory. 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 (e.g., "`diff txt_file_to_send txt_file_received`"). 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.

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

10. The Receiver must maintain a log file, as described in Section 3.5.1.

3.5.1 Receiver Log File

The Receiver must maintain a log file in its current working directory named `receiver_log.txt`, recording information in chronological order about every segment it sends and receives. If the log file already exists, then it should be overwritten. The format is the same as the Sender log file, as outlined in the Sender specification (Section 3.4.2). The first field is from the perspective of the Receiver endpoint, i.e., the Receiver will `rcv` DATA, FIN, and SYN segments, and `snd` ACK segments. Time should be in milliseconds and relative to when the first (possibly corrupt) SYN segment is received, thus the first entry will be a SYN segment at time 0. One obvious difference from the Sender log is that the Receiver has no PLC module. So the log will not record any segments as `drp`, and the status in the second column relies purely on URP error detection, so all sent segments will be `ok`, while received segments may be `ok` or `cor`.

For example, the following shows the corresponding Receiver log file for the scenario outlined in the Sender specification (Section 3.4.2). To recall, the Sender transmits 3500 bytes of data and the `rto` is 100 msec. The ISN is 63999, the `max_win` is 3000 bytes.

```
rcv  ok     0.00  SYN   63999     0
snd  ok     0.06  ACK   64000     0
rcv  ok   102.91  SYN   63999     0
snd  ok   102.91  ACK   64000     0
rcv  cor  104.45  DATA  64000  1000
rcv  ok   104.46  DATA  65000  1000
snd  ok   104.46  ACK   64000     0
rcv  ok   104.48  DATA    464  1000
snd  ok   104.49  ACK   64000     0
rcv  ok   310.22  DATA  64000  1000
snd  ok   310.24  ACK    1464     0
rcv  ok   310.57  DATA   1464   500
snd  ok   310.57  ACK    1964     0
rcv  ok   310.85  FIN    1964     0
snd  ok   310.86  ACK    1965     0
rcv  cor  415.61  FIN    1964     0
rcv  ok   517.98  FIN    1964     0
snd  ok   517.99  ACK    1965     0
```

As with the Sender, you are encouraged to write to the log file in real time. If the Receiver buffers log output and waits until the transfer is complete to write the file, any failure during the Receiver's operation could prevent crucial information from being recorded. Real-time logging can provide valuable insights even if an error occurs.

Once the entire file has been reliably transferred and the connection is closed (remember to follow the state machine for the Receiver), the Receiver will also write the following statistics at the end of the log file (i.e., `receiver_log.txt`). Please note, the amount of data received is strictly payload data.

Once again, you're encouraged to track these statistics in real-time at their logical code points, rather than try to pre- or post-compute them.

1. **Original data received** (in bytes, only payload data, does not count data in duplicate or corrupt segments)
2. **Total data received** (in bytes, only payload data, includes data in duplicate but not corrupt segments)
3. **Original segments received** (does not count duplicate or corrupt segments)
4. **Total segments received** (includes duplicate and corrupt segments)
5. **Corrupted segments discarded**
6. **Duplicate segments received** (does not include corrupt segments)
7. **Total acks sent** (includes duplicate acks)
8. **Duplicate acks sent**

For the example scenario, the Receiver log would be appended with:

```
Original data received:        3500
Total data received:           3500
Original segments received:       6
Total segments received:         10
Corrupted segments discarded:     2
Duplicate segments received:      2
Total acks sent:                  8
Duplicate acks sent:              4
```

You must follow the above format, but you may choose how to separate each statistical description with its value, e.g. with padding or a fixed number of spaces/tabs, provided the summary statistics are easily human readable.

REMINDER: 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.

Once the file transfer is complete the Receiver should finalise the log file and terminate.

The Receiver must only print a message to the terminal if it is terminating prematurely, due to unexpected behaviour by the Sender. No other output should be displayed. If you are printing output to the terminal for debugging purposes, make sure you disable it prior to submission.

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 the timeout interval.
3. You do not need to implement flow control or congestion control.
4. URP does not have to deal with reordered segments. Segments will rarely be 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 or corrupted. Note however, that segments can be dropped or corrupted by the PLC module, and thus segments may effectively arrive out of order at the receiver.

3.7 Implementation Details

The figure below provides a high-level and simplified view of the assignment. The URP protocol logic implements the state maintained at the Sender and Receiver, which includes all the state variables and buffers.

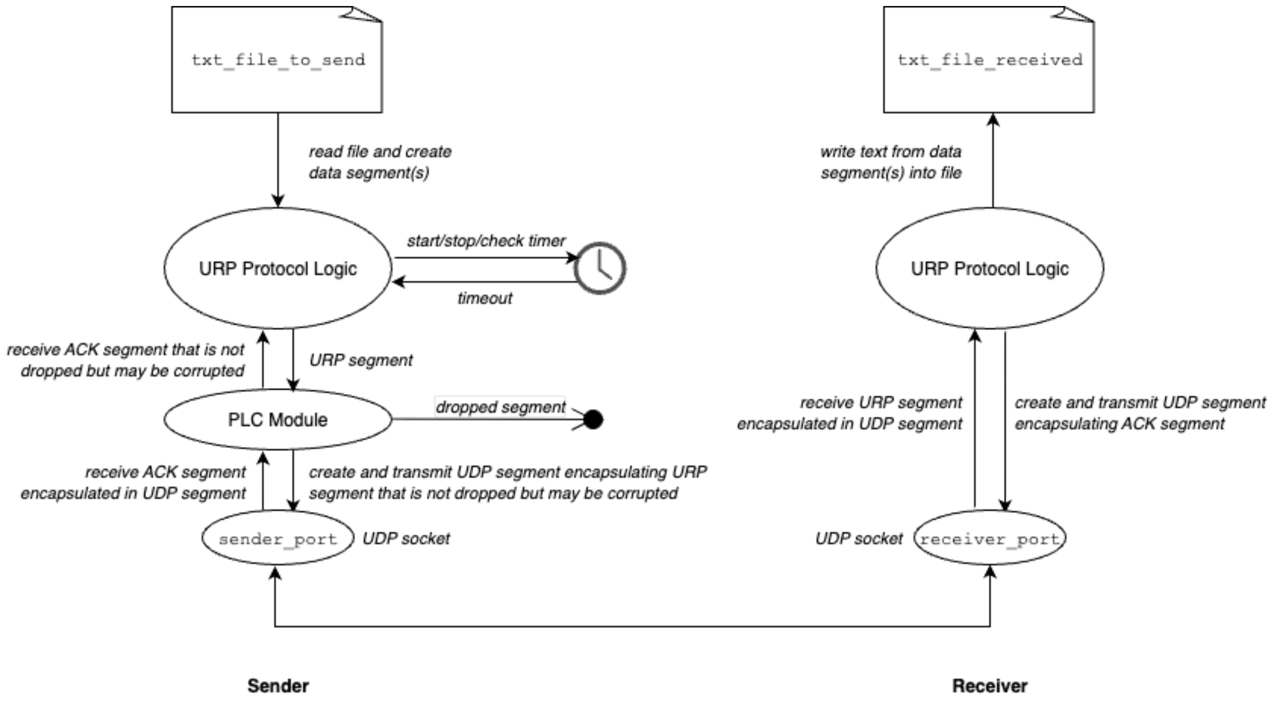
txt_file_to_send
read file and create data segment(s)
URP Protocol Logic
start/stop/check timer
timeout
receive ACK segment that is not dropped but may be corrupted
URP segment
PLC Module
dropped segment
receive ACK segment encapsulated in UDP segment
create and transmit UDP segment encapsulating URP segment that is not dropped but may be corrupted
sender_port
UDP socket
Sender
txt_file_received
write text from data segment(s) into file
URP Protocol Logic
receive URP segment encapsulated in UDP segment
create and transmit UDP segment encapsulating ACK segment
UDP socket
receiver_port
Receiver

Figure 3: The overall structure of your assignment.

Sender Design

The Sender must first execute connection setup, followed by data transmission, and finally connection 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 URP 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 URP segment (of any kind) must go through the PLC module, and if it is to be transmitted, must be encapsulated in a UDP datagram and sent through the Sender socket to the Receiver. The Sender should also process incoming ACK segments from the Receiver, after they have gone through the PLC module. Upon receipt of three duplicate ACKs, the Sender should retransmit the oldest unacknowledged DATA segment. In the case of a timeout, the Sender should retransmit the SYN or FIN, if in the connection setup or teardown process, or retransmit the oldest unacknowledged DATA segment, if in the data transmission phase. **As the Sender needs to deal with multiple events, we recommend using multi-threading**.

To clarify the logic behind fast retransmission and the use of a single timer, a simplified view of the Sender is shown below. This view focuses on the protocol layer (above the PLC that emulates an unreliable channel) and does not include connection setup or teardown.

```
event: window space available and unread file data remains
• read min(MSS, remaining file data) bytes of data from the file
• create URP segment where seqno is byte-stream number of first data byte in segment
• transmit segment
• start timer if not already running
    • think of timer as for oldest unACKed segment
    • expiration interval: rto msec

event: timeout
• retransmit oldest unACKed segment
• restart timer
• reset dupACKcount

event: ACK received
• if error detected in ACK
    • discard ACK
• otherwise if ACK acknowledges previously unACKed segment(s)
    • update what is known to be ACKed
    • if there are currently any unACKed segments
        • restart timer
    • otherwise (there are no unACKed segments)
        • stop timer
    • reset dupACKcount
• otherwise (no error detected in ACK and ACK acknowledges previously ACKed segment(s))
    • increment dupACKcount
    • if dupACKcount = 3
        • retransmit oldest unACKed segment
        • reset dupACKcount
```

Note that the use of a single timer does **not** require the Sender to maintain any additional state for each segment such as when it was sent.

Recall also that the Sender needs to implement the functionality of an unreliable channel that can drop and corrupt segments in either direction. Before sending any segment through the socket it must pass through the PLC to determine whether it may be dropped or corrupted. If the segment is to be dropped, then it should not be transmitted and nothing else needs to be done (other than updating the log). Only if the segment is not dropped should it be transmitted through the socket.

Upon receipt of a segment through the socket, it should first pass through the PLC to determine whether it may be dropped or corrupted. 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, so effectively this segment never arrived at the Sender and thus no action is necessary. If the segment is not dropped, only then should the Sender execute the URP protocol logic (as outlined above and in Sections 3.3, 3.4).

Receiver Design

Upon receipt of a UDP datagram through the socket, the Receiver should extract the URP segment which is encapsulated within the UDP datagram. The Receiver should then execute the URP protocol logic (as outlined in Sections 3.3 and 3.5). The Receiver should first check the segment for errors. If an error is detected, then the segment should be discarded. Otherwise, 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 all buffered in-order data should be written to the file and the buffer space made available. 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 ACK should then be encapsulated in a UDP datagram 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 closure 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**. However, 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.

Note about Artificial Delay
You may be tempted to add artificial delay to your Sender and/or Receiver using system calls such as `sleep()`. For example, to implement your retransmission timer, or to try and resolve synchronisation issues. You are generally cautioned against doing this. At the very least, it will introduce unnecessary inefficiency to your protocol. Of perhaps greater concern, if you are doing it in response to some issue, then most likely the bug will remain. It will just be more transient and difficult to find. There will almost certainly be a more logical solution that will improve the efficiency and robustness of your programs. In such scenarios, you are highly encouraged to take a systematic approach to debugging your code and properly resolve any issues.

4. Additional Notes

- This is NOT a group assignment. You are expected to work on this individually.
- **Sample Code**: We will provide sample code in all 3 languages on the assignment page that you may find useful. You are not required to use it, but you are welcome to adapt it as you see fit to help get started. Sample text files are also provided. We will use different files for our tests.
- **Programming Tutorial**: We will run online programming tutorials in Week 7 to help get students started with programming some of the building blocks of the assignment. A schedule for these sessions will be announced no later than Week 5.
- **Assignment Help Sessions:** We will run additional consultations in Weeks 5 and 7-9, for all 3 programming languages, to assist you with assignment related questions. A schedule will be posted on the assignment page of the course website. Please note, these sessions are 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`, `fcp`, `rcp`) 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 loses and/or corrupts 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 so on, until you test with packet loss and corruption in both directions). Test comprehensively with different probabilities to ensure that your Sender works correctly.

- You can next progress to implement the full functionality of URP, 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`, `rlp`, `fcp`, `rcp` 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 probabilities to test that the Sender can work as expected over a channel that loses and corrupts 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 so on, until you test with packet loss and corruption in both directions). Test comprehensively with different probabilities and window sizes 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.freecodecamp.org/news/what-are-threads-in-java-how-to-create-one/
  - 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. Alternatively, 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 course forum or even in the help sessions.
- **Backup and Versioning:** We strongly recommend you to back-up your programs frequently. CSE backs up 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 12.2, OpenJDK 17, and Python 3.11.** You may only use the basic socket programming APIs provided 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.

5. Assignment Submission

Please ensure that you use the mandated file names. You may of course have additional header files and/or helper files. If you are using C, then you MUST submit a Makefile along with your code. 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 3 pages). Provide details of which language you have used (e.g., C) and the organisation of your code (Makefile, directories if any, etc.). Your report must contain a brief discussion of how you have implemented the URP protocol. This should include the overall program design, data structure design, a brief description of the operation of the Sender and Receiver, and a description of how you've implemented error detection. 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. Next, type: `give cs3331 assign assign.tar`

You should receive a message stating the result of your submission, **ensure** it says it's accepted. 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 a file named `assign.tar`. 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 wait 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: The UNSW standard late penalty will apply, which is 5% per day of the maximum available mark, for up to 5 days. This assignment is worth 20 marks, therefore the penalty equates to a 1-mark deduction per day late. Submissions after 5 days will not be accepted.

6. Plagiarism

You are to write all the code for this assignment yourself. All source code is 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 or 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.

7. Marking Policy

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

Test 1 - Stop and Wait over a Reliable Channel: 4 marks

We will test your URP implementation when executed as a stop and wait protocol (`max_win=1000`) and when the underlying channel is reliable (`flp=rlp=fcp=rcp=0`).

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 1000

python3 sender.py 59606 56007 test1.txt 1000 rto 0 0 0 0
```

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 URP protocol is correctly implemented at both end points.

Test 2 - Stop and Wait over an Unreliable Channel: 6 marks

Next, we will test your URP implementation while operating as a stop and wait protocol (`max_win=1000`) but where the underlying channel is unreliable.

In the first instance, we will only induce packet loss in the forward and reverse direction (`fcp=rcp=0`). The Sender will be instantiated as follows (Receiver will be instantiated as above):

```
python3 sender.py 59606 56007 test1.txt 1000 rto flp rlp 0 0
```

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

In the second instance, we will only induce packet corruption in the forward and reverse direction (`flp=rlp=0`). The Sender will be instantiated as follows (Receiver will be instantiated as above):

```
python3 sender.py 59606 56007 test1.txt 1000 rto 0 0 fcp rcp
```

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

In the third instance, we will induce packet loss and corruption in both directions. The Sender will be instantiated as follows (Receiver will be instantiated as above):

```
python3 sender.py 59606 56007 test1.txt 1000 rto flp rlp fcp rcp
```

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

Test 3 - Sliding Window over a Reliable Channel: 2 marks

We will test your URP implementation when executed as a sliding window protocol and when the underlying channel is reliable (`flp=rlp=fcp=rcp=0`).

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 max_win

python3 sender.py 59606 56007 test1.txt max_win rto 0 0 0 0
```

We will test for different values of `max_win` (always a multiple of 1000, with the same value provided to both programs), `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 URP protocol is correctly implemented at both end points.

Test 4 - Sliding Window over an Unreliable Channel: 6 marks

Next, we will test your URP 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 and reverse direction (`fcp=rcp=0`). The Sender will be instantiated as follows (Receiver will be instantiated as above):

```
python3 sender.py 59606 56007 test1.txt max_win rto flp rlp 0 0
```

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

In the second instance, we will only induce packet corruption in the forward and reverse direction (`flp=rlp=0`). The Sender will be instantiated as follows (Receiver will be instantiated as above):

```
python3 sender.py 59606 56007 test1.txt max_win rto 0 0 fcp rcp
```

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

In the final instance, we will induce packet loss and corruption in both directions. The Sender will be instantiated as follows (Receiver will be instantiated as above):

```
python3 sender.py 59606 56007 test1.txt max_win rto flp rlp fcp rcp
```

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

Test 5 – Report: 1 mark

The report should not be longer than 3 pages. Provide details of which language you have used (e.g., Python) and the organisation of your code (e.g. Makefiles, directories if any, etc.). Your report must contain a brief discussion of how you have implemented the URP protocol. This should include the overall program design, error detection mechanism implemented, 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 NOTES:

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).

If your Sender and Receiver do not conform to the specified segment format as indicated in the specification, you will only be graded out of 50% of the total marks (i.e., a 50% penalty will be assessed).

8. URP Monitor Program

A monitoring program is available in the CSE environment that sits between the Sender and Receiver, forwarding and logging all segments. This can be a useful tool for verifying that your URP segment structure is implemented correctly.

To use the Monitor, select three distinct ports — `sender_port`, `monitor_port`, and `receiver_port` — one for each application. The Monitor and Receiver may be started in any order, but both must be running before you launch the Sender.

We assume that the Sender and Receiver are implemented in Python, but the arguments are identical for C and Java versions. Note that the Sender sends to the Monitor, and the Receiver receives from the Monitor. All three applications must be executed on the same host.

Run the Receiver, replacing the `sender_port` with the `monitor_port`:

```
python3 receiver.py receiver_port monitor_port [other arguments ...]
```

Run the Monitor, providing all three ports:

```
3331 urpmon sender_port monitor_port receiver_port
```

Run the Sender, replacing the `receiver_port` with the `monitor_port`:

```
python3 sender.py sender_port monitor_port [other arguments ...]
```