The University of Calgary

Department of Electrical & Computer Engineering

ENSF 462 Networked Systems

(Fall 2023)
Due: Nov. 07th, 2023

Lab 4 Reliable Data Transfer Protocol

Lab Section	Section Date	Location
B01	November 02nd, 2023	ENA 305
B02	November 07th, 2023	ENG 24
B03	November 01st, 2023	ICT 319

Objectives

In this lab, you will be writing the sending and receiving code for implementing the stop-and-wait RDT 3.0 protocol discussed in class and the textbook, building upon an existing RDT 1.0 implementation.

RDT 1.0 Implementation

The provided codes include the implementation of application layer (`Sender.py`, `Receiver.py`), transport layer (`RDT.py`), and network layer (`Network.py`) that cooperate to provide end-to-end communication. The provided code implements the RDT 1.0 protocol, where the sender sends messages to the receiver, and the receiver sends them back. The messages are sent through the transport layer provided by an RDT implementation using the `rdt_1_0_send` and `rdt_1_0_receive` functions. The starting code `rdt.py` provides only the RDT 1.0 version of the protocol, which does not handle packet corruption or loss. The RDT protocol uses `udt_send` and `udt_receive` functions from `network.py` to transfer bytes between the sender and receiver. `rdt.py` relies on the `Packet` class (in the same file) to form transport layer packets.

To run the starting code, you may run:

```
python Receiver.py 5678
```

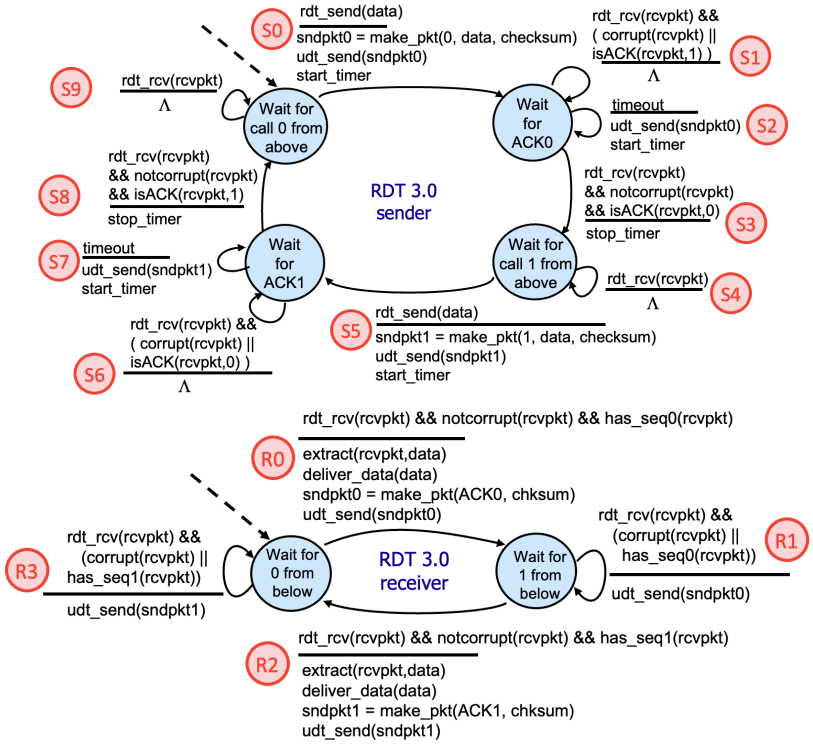
and

```
python Sender.py localhost 5678
```

in separate terminal windows. Be sure to start the receiver first, allowing it to start listening on a socket, and start the sender soon after, before the receiver times out.

RDT 3.0 Implementation Requirements

Your job will be to implement the RDT 3.0 protocol, with FSM specifications shown in the following figure.



Following features need to be implemented in your code:

Sender:

- The sender calls the `rdt_3_0_send` function to transmit multiple messages to the receiver. The provided code includes an example message, or you can customize it to send your preferred message.
- Sequence numbers should be used to identify different messages. While the FSM specification above illustrates the use of a one-bit sequence number, you have the flexibility to choose a multi-bit sequence number if desired.
- The sender transmits one message at a time. Following message transmission, the sender must await a reply from the receiver and take appropriate actions based on the FSM specification. Since packet reordering is not considered (with `prob_pkt_reorder=0` in `Network.py`), *the sender simply resends the previous message upon detecting corruption or receiving an ACK with an unexpected sequence number.*
- During the communication, informative statements should be printed, including:
  - Print “Send message `seq_num`” upon sending a message, where `seq_num` is the sequence number of that message.
  - Print “Timeout! Resend message `seq_num`" when timeout occurs.
  - Print "Corruption detected in ACK. Resend message `seq_num` " when detecting corruption.
  - Print "Receive ACK `seq_num`. Resend message `seq_num` " when receiving an ACK with unexpected sequence number

- Print "Receive ACK `seq_num`. Message successfully sent!" when receiving an ACK with expected sequence number.

Receiver

- The receiver calls the `rdt_3_0_receive` function to receive messages from the sender.
- Upon receiving a message, if the message is not corrupted and has the expected sequence number, receiver will send an ACK with the current message sequence number. Otherwise, it will send an ACK with the sequence number of the previous correctly received message.
- During the communication, informative statements should be printed, including:
  - Print “Corruption detected! Send ACK `ACK_seq_num`” when detecting corruption.
  - Print "Receive message `msg_seq_num`. Send ACK `ACK_seq_num` " Here `msg_seq_num` and `ACK_seq_num` are the message sequence number and ACK sequence number. Be careful that different `ACK_seq_num` should be used in different cases.

Coding instruction:

- You need to extend `rdt.py` to tolerate packet corruption and loss. Specifically, you need to write `rdt_3_0_send` and `rdt_3_0_receive` functions to implement the sending and receiving code of RDT 3.0.
- You may need to modify/extend the `Sender.py and Receiver.py` to implement the required features.
- You may need to modify/extend the `Packet` class to include the necessary information for these functions to work correctly.
- The provided implementation of `Network.py` is reliable, with zero probability for packet corruption and loss. But we will test your code with non-zero probability for packet corruption and loss by changing the values of `prob_pkt_loss` and `prob_byte_corr` of the `NetworkLayer` class. *You need to change those variables yourself to test your code.*
- `Network.py` offers student-callable functions to simulate network-layer behavior. *You do not need to modify the* `Network.py` *code except for adjusting the packet corruption and loss probabilities* `prob_pkt_loss` *and* `prob_byte_corr`.

Submit a lab report that includes the following:

- **Your name and UCID #**
- **Your version of `Sender.py`, `Receiver.py`, `RDT.py`, and `Network.py`**
- **Screenshots demonstrating the sender and receiver handling packet corruption and loss**