**UNIVERSITY OF DELAWARE**
**DEPARTMENT OF COMPUTER & INFORMATION SCIENCES**

**CISC 450 / CPEG 419: Computer Networks I**

Fall Semester, 2018 Professor: Adarsh Sethi

# Programming Project 2
# Some Helpful Hints

All the hints you had for Project 1 will apply to Project 2 as well, so you should first read through that document. Below are some additional issues to keep in mind:

1. Your programs for Project 2 must run on the host *cisc450.cis.udel.edu*. This is important because you will be asked to do a demo of your project and the demo will be conducted on this host.

2. Each message in Project 2 (either a data packet or an ACK, as described in the project specification) should be transmitted as a single UDP segment using a single invocation of the *sendto* command. Similarly, use a single invocation of the *recvfrom* command to receive a message. Both the header and the data characters are part of the packet and should be transmitted and received together in the single UDP segment. It is not permissible to send each field of the packet in separate *sendto* commands.

3. The project specification explicitly states that the data read from one line of the input file, including the newline character, should be transmitted in a single data packet. Different lines of the input file may have different number of characters, so different data packets may have different sizes. It is also not permissible to merge multiple lines of the input file into a single packet.

4. Recall once again that C strings are null-terminated. The project explicitly requires that no null character should be transmitted in a data packet. Note again that a null character is not the same as a newline character. Some of you may wish to use the C string library functions to read and write the data from the input file into your messages and from the messages to the output file. Use of these functions is not necessarily wrong, but you must be careful that *the null character is not transmitted in a data packet or written into the output file.* In Project 2, we will be looking carefully at whether or not you are transmitting the null character in the packets, and points will be deducted if you are doing so.

5. You should read one line from the input file and transmit it before reading the next line. It is not permissible to read in the all the lines from the input file into one giant array or other structure and then transmit them from there.

## Implementing Reliable Data Transfer

The remainder of this document will focus on how to implement the Stop-and-Wait Protocol. Note that, because we are using the Stop-and-Wait Protocol, the Sender must transmit one data packet and then wait for an ACK to come from the Receiver before sending the next data packet.

Your programs should follow the FSMs for *rdt3.1* for the ABP Protocol, as described in the lecture slides for Topic 3, Part 3, page 5. However, there are a few significant departures from the above protocol in your implementation:

- How you handle the "Call from above" represented by the *rdt_send(data)* function: Instead of waiting for the user above to call your protocol machine, you should call a function that reads the next line from the input file and hands it to you as the data to be transmitted in the next packet.
- Checksum: Your packet format does not have a checksum, and your data packets and ACKs are not going to carry a checksum. Thus you will not be doing any checksum computation, nor checking the received packets to see if there are any errors. Thus, the function *make_pkt* will not have a checksum parameter, and you will omit the calls to the functions *corrupt* and *notcorrupt* in your code.
- Timeout: The Sender must implement a timeout to recover from lost packets and/or ACKs. The class slides on Socket Programming, Topic 2 Part 3, describe how to implement a timeout (see last page, page 15). The value of the timeout that you specify has two components, seconds and microseconds. For instance, a timeout of 1.5 seconds would be specified with a seconds component of 1 and a microseconds component of 500000. This timer takes effect when you call the *recfrom* function to wait for an ACK and so there are no separate actions for starting and stopping the timer.

**Sender Actions:**

The FSM for the *rdt3.1* Sender has two states, "Wait for call from above" and "Wait for ACK". The general actions of your Sender can be implemented in two nested loops as described below:

- Before the loop, initialize your variables.
- Start outer loop to implement the state "Wait for call from above".
- You can ignore the self-loop transition on this state.
- Implement the transition to the state "Wait for ACK": Get a line from the input file (if any), make a packet, and transmit it (you will not start the timer at this point). If there are no more lines in the file, then quit outer loop.
- Start inner loop to implement the state "Wait for ACK".
- In this loop, call the function *recvfrom* to wait for an ACK along with a timeout.
- When you get to this next step after the *recvfrom*, either an ACK was received, or a timeout occurred.
- Check if timeout has occurred. If yes, then (self-loop transition for timeout) retransmit the packet and go back to start of inner loop.
- If you reach here, there was no timeout and an ACK was received.
- If the ACK sequence number is not what you were expecting (top self-loop transition in the FSM), then go back to start of inner loop (note: null action).
- If the ACK has the correct sequence number, (transition to the state "Wait for call from above"), then carry out the actions of this transition and quit inner loop (note: there is no need to stop the timer, as that automatically happened when ACK was received).
- End of inner loop.
- End of outer loop.

- At this point, transmit the “End of Transmission” packet; no ACK will be received or expected for this packet. You are finished.

**Receiver Actions:**

The FSM for the *rdt3.1* Receiver has only one state, “Wait for call from below”. The general actions of your Receiver can thus be implemented in a single loop as described below:

- Before the loop, initialize your variables.
- Start Receiver loop (to implement the state “Wait for call from below”).
- Call the *recvfrom* function to wait for a packet to arrive from the Sender. Note that the Receiver does not use a timeout.
- When a packet is received, check if Count field in the packet is 0. If it is 0, this signifies an EOT packet, so quit loop.
- If the Count field is not 0, this is a regular data packet. In this case, first call the *SimulateLoss* function described in the Project 2 specification to simulate packet loss.
- If *SimulateLoss* returns 0, then packet is lost; return to start of loop.
- If *SimulateLoss* returns 1, then packet is correctly received. Process the packet according to the FSM.
- Top self-loop transition: Check the packet sequence number; if this is a duplicate, do not deliver to the user, and generate the appropriate ACK.
- Bottom self-loop transition: Check the packet sequence number; if it is a new packet, deliver to the user (i.e., store in the output file), and generate the appropriate ACK.
- After ACK is generated, call the function *SimulateACKLoss* to simulate ACK loss (again this is described in the Project 2 specification).
- If the function *SimulateACKLoss* returns 0, the ACK is lost, and is not transmitted. If the function returns 1, then transmit the ACK.
- End of loop.