Intermediate Project: Implement a Reliable Data Transfer Protocol
CPSC 3600 - Network Systems
Summer 2023

In this project you will be implementing a simple reliable data transfer (RDT) protocol. This is **NOT** a real protocol used on the internet, but it does illustrate some ideas that show up in real protocols (like TCP).

All network communication in this project will be simulated, so as to allow us complete control over when packets are corrupted, lost, or misordered. This is important for testing and proving that your solution works, but isn't something we can do with real network traffic.

1 Assignment Instructions

You will be implementing a full-duplex GBN host. Full-duplex means that the program is capable of both sending and receiving data. The autograder will create two instances of your GBN host and have them talk with each other through a simulated network.

The project template includes three files. You will only be editing one of these files; the others will be used to test your projects. The important functionality of each of these files is lightly documented below; **please refer to the code itself for the complete documentation.**

1. **network_simulator.py**: **You do not need to implement any code in *network_simulator.py***. You'll be using functions defined in it to manage sending and receiving through the simulated network, as well as starting and stopping timers.

   (a) ***EventEntity enumeration*** network_simulator.py defines an enumeration used to identify which of the two hosts are sending or receiving a given packet. The entity assigned to a given instance of gbn_host is passed in to the constructor and is stored in self.entity.

   (b) ***pass_to_network_layer(entity, packet)*** Passes packed messages to the simulated network layer, where it is communicated to the host on the other end of the simulated socket. Your code should call this once it has packed a message intended for the client on the other side of the simulated socket. The two expected parameters are 1) the entity that is sending the message (stored in self.entity), 2) the packed bytes of the packet.

   (c) ***pass_to_application_layer(entity, data)*** Passes decoded data up to the application layer. Your code should call this once it has received a packet containing data. The two expected parameters are 1) the entity that is passing a packet's payload to an application, and 2) the unpacked data to be delivered to an application.

   (d) ***start_timer(entity, timer_interval)*** Starts a timer event for a given host. Two parameters are expected, the entity for which a timer should be started and the interval of the timer (this is defined for you in __init__).

   (e) ***stop_timer(entity)*** Stops any current timer events associated with a given host. A single parameter is expected, the entity for which a timer should be stopped.

2. **gbn_host.py**:

   (a) ***receive_from_application_layer(payload)*** The autograder will call this function when a given entity receives a piece of data from a simulated application to transmit across the simulated network. The behavior of this function is specified in the Sender FSM shown in Figure 2.

   (b) ***receive_from_network_layer(bytes)*** The autograder will call this function when a packet addressed to a given entity is received from the simulated network. The behavior of this function is specified in the Sender and Receiver FSMs shown in Figures 2 and 3. Note that this function needs to handle behavior that occurs in both the sender and the receiver FSM. When you receive data from the network layer, you'll need to determine which functionality is appropriate.

   (c) ***timer_interrupt()*** The autograder will call this function when a timer set by your program expires.

   (d) ***create_data_pkt(seq_num, payload)*** should create a bytes object representing a data packet with this seq_num and payload.

   (e) ***create_ack_pkt(seq_num)*** should create a bytes object representing an ack packet with this seq_num.

(f) ***create_checksum(packet)*** should produce a checksum using the bytes provided for the packet.

(g) ***unpack_pkt(packet)*** should unpack a packet object and return a dictionary with the different values. This function will be called by the network simulator when printing messages about the packets traveling through the simulated network.

(h) ***is_corrupt(bytes)*** should determine whether the received data is corrupted, based on the checksum that is included with the packet. This will be used by your code, and will also be tested by the autograder.

3. **rdt_tester.py**: This is the file you should run to test your code on your local machine. You can control which test cases are run by uncommenting the different test files listed in the *tests* list in the main method. Only the first test is uncommented in the file you will download with your template. Once you have this one working, begin uncommenting the other tests.

   You do not have to use this file at all, but it will be helpful for debugging. Alternatively, you can just submit your files to Gradescope once you are ready to test them and base your development on the output log shown there.

2 Reading a Finite State Machine

We will use finite state machines to describe the logical functionality of this RDT protocol. Finite state machines show how a system changes state in response to specific events. An example FSM is shown below in Figure 1. This particular example illustrates the behavior of a simple vending machine.

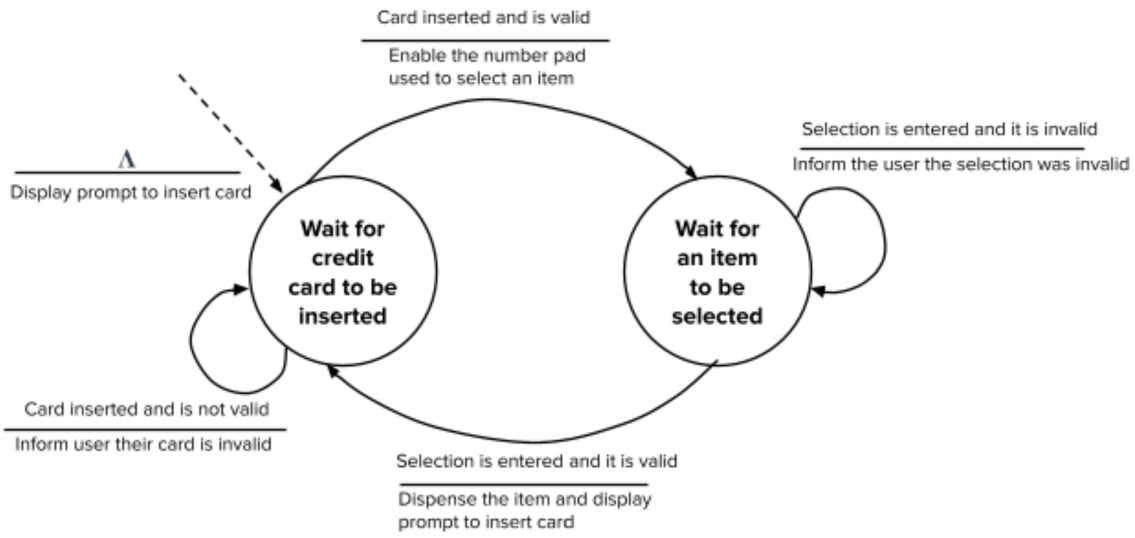

Figure 1: This finite state machine illustrates the behavior of a simple vending machine.

Circles represent different states that the system can occupy. Transitions between states are show using solid arrows. Each transition is annotated with information about what event caused the transition and what is done in response to it. Events are indicated above the line in the annotation, and responses are indicated below the line in the annotation. A symbol Λ below the line indicates that nothing happens in response to this event. Transitions can go back to the same state; this indicates that an event has occurred but that logical state of the system has not changed. A dashed arrow pointing at a state indicates that the program starts in that state. Initial values of variables can be indicated via a transition annotation linked to this dashed line.

3 Sender Functionality

The sender's functionality is specified in the FSM shown in Figure 2.

4 Receiver Functionality

The receiver's functionality is specified in the FSM shown in Figure 3.

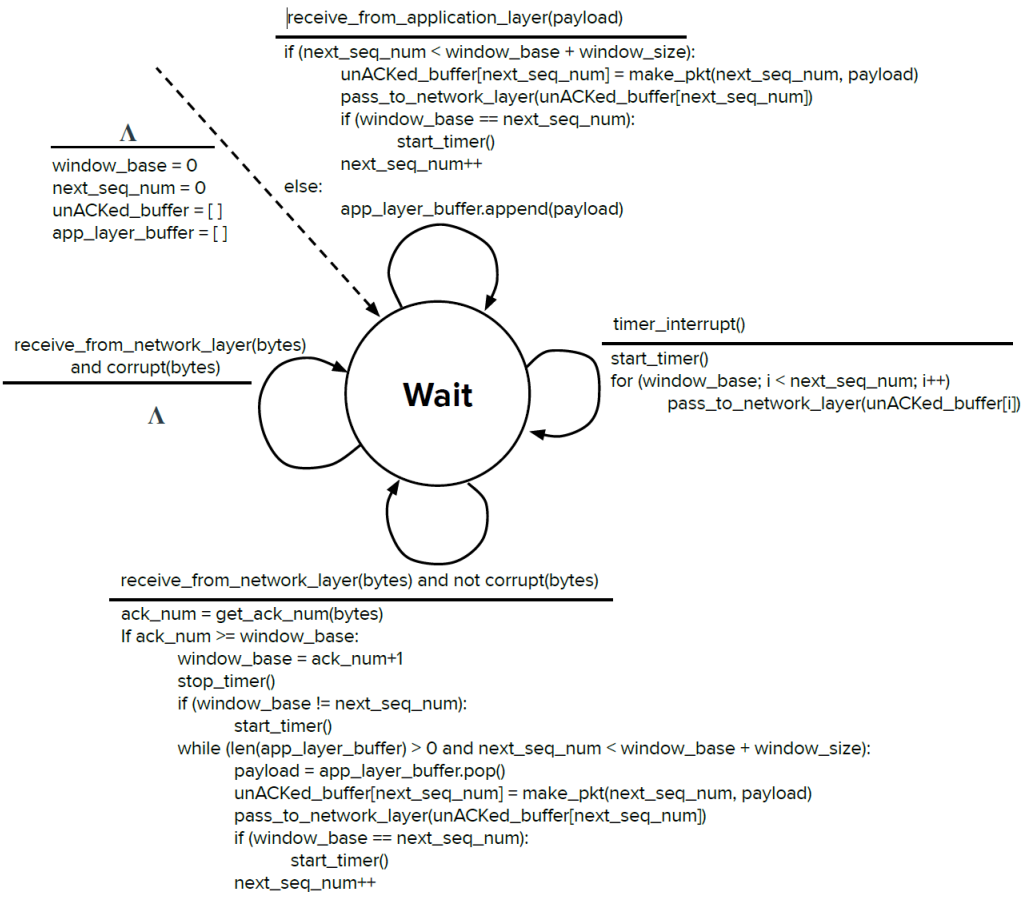

Figure 2: GBN Sender: All code shown here is pseudocode intended to describe the logical functionality of this RDT protocol. Your code will be similar to this, however it's implementation may differ at certain points.

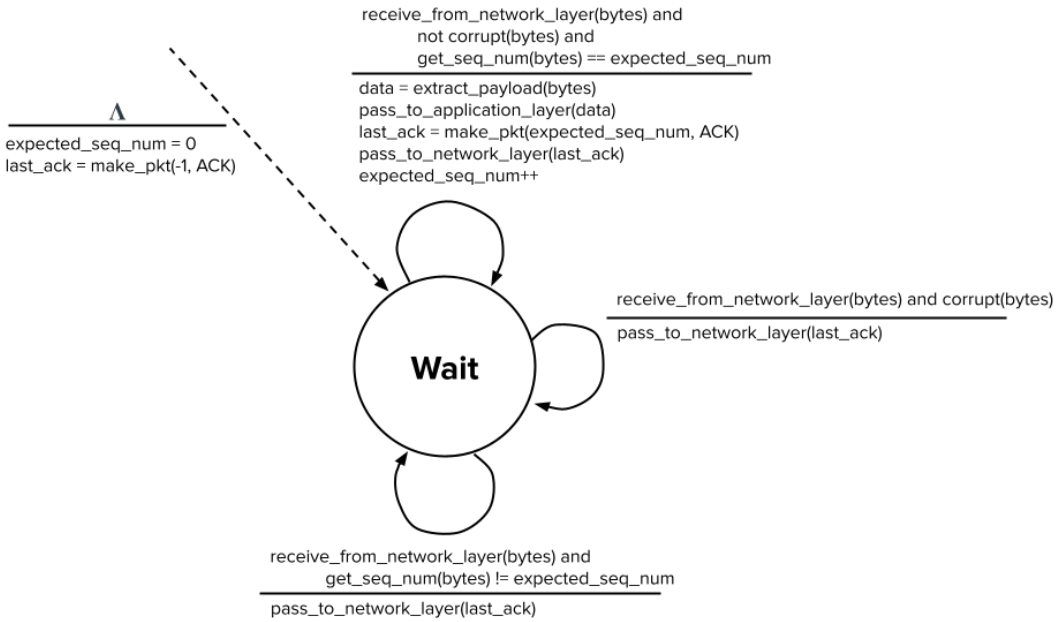

Figure 3: GBN Receiver: All code shown here is pseudocode intended to describe the logical functionality of this RDT protocol. Your code will be similar to this, however it's implementation may differ at certain points.

5 Packet Format

Your packets will contain the following header fields:

1. Packet Type (unsigned half) - a numeric value representing the type of a packet. 0x0 equals a data packet and 0x1 equals an ACK packet. We're using an unsigned half instead of a byte for this to make the checksum process simpler (this way the checksum will be aligned at the byte boundaries).
2. Packet Number (unsigned int) - the relevant tracking number associated with this packet. For data packets, this is the sequence number. For acknowledgement packets, this is the sequence number of the last successfully received data packet.
3. Checksum (unsigned half) - this should contain the the Internet checksum computed for your packet, as discussed in Section 6.
4. Payload Length (unsigned int) - the length of the included *payload*. This field is only included in data packets.
5. Payload (varchar string) - a variable length payload included in data packets.

6 Computing an Internet Checksum

The algorithm to compute an Internet checksum is fairly straightforward:

1. Convert your packet into a byte array using pack. One complication here is that you don't yet know the checksum, but the checksum should be part of the packed data. We can get around this by substituting 0 for where the checksum should go when packing the data, and then repacking the data once we have computed the checksum.

```
# We're packing an arbitrary set of data in this example. The checksum will eventually
# be stored in the H value. We're going to put a 0 there for now.
message = pack("!IH2s", 378, 0, "ab".encode())

# Get the checksum using a function we've written
checksum = compute_checksum(message)

# Now re-pack the data with the checksum
message = pack("!IH2s", 378, checksum, "ab".encode())
```

2. Divide your packet up into 16-bit words. If your packet contains an odd number of bytes, you'll need to pad the end of the packet with a 0-byte (0x0000) in order to get the final 16-bit word.

```
# Check to see if the packet contains an odd number of bits. If so, append a 0-byte to the
# end of the packet data. bytes(1) creates a byte array of length 1, initialized with 0's.
if len(packet) % 2 == 1:
    packet = packet + bytes(1)

# Next, divide your byte array into a series of 16-bit words. Every entry in a bytearray
# is a single byte, which means it only uses the lowest 8 bits. To convert individual bytes
# into a 16-bit word we need to perform a bitwise shift operation, which can move the lowest
# 8 bits into the position of the next highest 8 bits. We do this with the bitwise left shift
# operator <<.
# Once this has been done, we can OR the result with the next byte. This results in a
# combination of the two bytes, where the first byte is placed in the 8-15 bit positions,
# and the second byte is placed in the 0-7 bit positions.
for i in range(0, len(pkt), 2):
    word = pkt[i] << 8 | pkt[i+1]
```

3. Sum each of the words together, carrying any overflow bits.

```
# Compute the two words you want to sum, as discussed above
word1 = ...
word2 = ...

# Add the words together like you would add any other numbers
summed_words = word1 + word2

# Carry any overflow bits. When adding 2 16-bit words, it's possible that a 1 will result in
# the 17th bit position. If this occurs, we want to remove this bit and add it back to the
# lowest 16 bits of the computed number. The line of code below does both of these.
# (summed_wrods & 0xffff) ensures that the upper 16 bits will always be 0, and
# (summed_words >> 16) right shifts the sum by 16 bits, ensuring that it will equal either
# a 0 or a 1, depending on what value was in the 17th position.
result = (summed_words & 0xffff) + (summed_words >> 16)
```

4. Perform the 1's complement on the result.

```
# The one's complement is computed using the ~ operator. We specifically want a 16-bit value,
# and Python is internally representing all of the numbers we're working with as 32-bit
# integers, so we need to zero out the upper 16 bits by ANDing the result with 0xffff.
checksum = ~result & 0xffff
```

You now have the checksum and can repack the data with the checksum as shown in step 1.

6.1 Handling When the Packet Length is Corrupted

Any part of your packet can be corrupted. The above algorithm can be executed without issue when any part of a packet is corrupted **except** for when the packet length is corrupted. You can only check for corruption once the entire packet has been received, but you can't receive the entire packet unless you first know the packet length. If the packet length is corrupted then you won't be able to fetch the entire packet.

In the test cases provided, corruption of the packet length results in a value much larger than the actual packet. This will cause an exception to be thrown when you attempt to fetch the packet payload when using a corrupted packet length. This exception will contain a message like "*unpack requires a buffer of 134217728 bytes*". **This is expected behavior**.

When you receive this exception (once you have your pack and unpack functions working correctly), you should treat the packet you've received as corrupt and handle it appropriately. This will require you to wrap calls to unpack in a *try...except...* block in order to catch the exception. This will allow you to catch this particular instance of corruption prior to calling the *is_corrupt()* method you will be implementing. All other instances of corruption should be detected using that function.

7 Reading the program's debug output

Two log files are produced for each test that is run. These log files report what occurs when A is sending data to B (including B's acknowledgments to A) and when B is sending data to A (including A's acknowledgments to B). Each entry contains the following information: [the entity where this event occurred] @ [the simulated time it occurred at]: [the specific event].

```
B @ 109.3122: Rcvd from Application Layer: aaaa
B @ 109.3122: Passing to Network Layer: [TYPE: DATA, SEQ: 0, CKSUM: 15673, LEN: 4, PAYLOAD: aaaa]
B @ 109.3122: Starting Timer
A @ 109.7030: Rcvd from Network Layer: [TYPE: DATA, SEQ: 0, CKSUM: 15673, LEN: 4, PAYLOAD: aaaa]
A @ 109.7030: Passing to Application Layer: aaaa
A @ 109.7030: Passing to Network Layer: [TYPE: ACK, SEQ: 0, CKSUM: 65534]
B @ 110.2940: Rcvd from Network Layer: [TYPE: ACK, SEQ: 0, CKSUM: 65534]
B @ 110.2940: Stopping Timer
A @ 189.3456: Passing to Network Layer: [TYPE: ACK, SEQ: 0, CKSUM: 65534]
B @ 190.1969: Rcvd from Network Layer: [TYPE: ACK, SEQ: 0, CKSUM: 65534]
A @ 192.7117: Passing to Network Layer: [TYPE: ACK, SEQ: 0, CKSUM: 65534]
A @ 192.7117: CORRUPTING PACKET!: [TYPE: ACK, SEQ: 0, CKSUM: 65534]
B @ 302.8746: Rcvd from Application Layer: dd
B @ 302.8746: Passing to Network Layer: [TYPE: DATA, SEQ: 1, CKSUM: 39832, LEN: 2, PAYLOAD: dd]
B @ 302.8746: LOSING PACKET!: [TYPE: DATA, SEQ: 1, CKSUM: 39832, LEN: 2, PAYLOAD: dd]
B @ 302.8746: Starting Timer
B @ 305.8746: Timer Interrupt
B @ 305.8746: Starting Timer
B @ 305.8746: Passing to Network Layer: [TYPE: DATA, SEQ: 1, CKSUM: 39832, LEN: 2, PAYLOAD: dd]
A @ 305.9913: Rcvd from Network Layer: [TYPE: DATA, SEQ: 1, CKSUM: 39832, LEN: 2, PAYLOAD: dd]
A @ 305.9913: Passing to Application Layer: dd
A @ 305.9913: Passing to Network Layer: [TYPE: ACK, SEQ: 1, CKSUM: 65533]
A @ 305.9913: LOSING PACKET!: [TYPE: ACK, SEQ: 1, CKSUM: 65533]
B @ 308.8746: Timer Interrupt
B @ 308.8746: Starting Timer
B @ 308.8746: Passing to Network Layer: [TYPE: DATA, SEQ: 1, CKSUM: 39832, LEN: 2, PAYLOAD: dd]
A @ 309.1529: Rcvd from Network Layer: [TYPE: DATA, SEQ: 1, CKSUM: 39832, LEN: 2, PAYLOAD: dd]
A @ 309.1529: Passing to Network Layer: [TYPE: ACK, SEQ: 1, CKSUM: 65533]
A @ 309.1529: LOSING PACKET!: [TYPE: ACK, SEQ: 1, CKSUM: 65533]
B @ 311.8746: Timer Interrupt
B @ 311.8746: Starting Timer
B @ 311.8746: Passing to Network Layer: [TYPE: DATA, SEQ: 1, CKSUM: 39832, LEN: 2, PAYLOAD: dd]
A @ 312.4941: Rcvd from Network Layer: [TYPE: DATA, SEQ: 1, CKSUM: 39832, LEN: 2, PAYLOAD: dd]
A @ 312.4941: Passing to Network Layer: [TYPE: ACK, SEQ: 1, CKSUM: 65533]
B @ 312.7364: Rcvd from Network Layer: [TYPE: ACK, SEQ: 1, CKSUM: 65533]
B @ 312.7364: Stopping Timer
B @ 421.3759: Rcvd from Application Layer: eeeee
B @ 421.3759: Passing to Network Layer: [TYPE: DATA, SEQ: 2, CKSUM: 53293, LEN: 5, PAYLOAD: eeeee]
B @ 421.3759: CORRUPTING PACKET!: [TYPE: DATA, SEQ: 2, CKSUM: 53293, LEN: 5, PAYLOAD: eeeee]
B @ 421.3759: Starting Timer
A @ 421.8057: Rcvd from Network Layer: [TYPE: DATA, SEQ: 3, CKSUM: 53293, LEN: 5, PAYLOAD: eeeee]
A @ 421.8057: Passing to Network Layer: [TYPE: ACK, SEQ: 1, CKSUM: 65533]
A @ 421.8057: CORRUPTING PACKET!: [TYPE: ACK, SEQ: 1, CKSUM: 65533]
B @ 424.3759: Timer Interrupt
B @ 424.3759: Starting Timer
B @ 424.3759: Passing to Network Layer: [TYPE: DATA, SEQ: 2, CKSUM: 53293, LEN: 5, PAYLOAD: eeeee]
```

```
A @ 424.6422: Rcvd from Network Layer: [TYPE: DATA, SEQ: 2, CKSUM: 53293, LEN: 5, PAYLOAD: eeeee]
A @ 424.6422: Passing to Application Layer: eeeee
A @ 424.6422: Passing to Network Layer: [TYPE: ACK, SEQ: 2, CKSUM: 65532]
B @ 425.0137: Rcvd from Network Layer: [TYPE: ACK, SEQ: 2, CKSUM: 65532]
B @ 425.0137: Stopping Timer
A @ 577.4670: Passing to Network Layer: [TYPE: ACK, SEQ: 2, CKSUM: 65532]
A @ 577.4670: LOSING PACKET!: [TYPE: ACK, SEQ: 2, CKSUM: 65532]
A @ 686.3049: Rcvd from Network Layer
A @ 686.3049: Passing to Network Layer: [TYPE: ACK, SEQ: 2, CKSUM: 65532]
A @ 686.3049: LOSING PACKET!: [TYPE: ACK, SEQ: 2, CKSUM: 65532]
```

Each of the possible messages that may appear in the logs are explained below.

1. **Rcvd from Application Layer** indicates that the simulated application has called send() and given your transport layer protocol data to send across the network. This maps on to the simulator calling your program's *receive_from_application_layer()* function.
2. **Rcvd from Network Layer** indicates the simulated network layer has received a packet from the other end of the connection, which could be a data message or an acknowledgment message. This maps on to the simulator calling your program's *receive_from_network_layer()* function.
3. **Passing to Network Layer** indicates that your transport layer protocol is passing a prepared packet to the simulated network layer for transmission to the other end of the connection. This maps on to your program calling *simulator.pass_to_network_layer()*. The contents of the packet sent are displayed in the log, assuming you've packed your packet correctly.
4. **Passing to Application Layer:** indicates that your transport layer protocol has received a valid piece of data in the correct order and has given that to the application it was addressed to. This maps on to your program calling *simulator.pass_to_application_layer()*. The data you give to the application layer is displayed in the log.
5. **Starting Timer** indicates that a timer has been started. This maps on to your program calling *simulator.start_timer()*.
6. **Stopping Timer** indicates that a timer has been stopped. This maps on to your program calling *simulator.stop_timer()*.
7. **Timer Interrupt** indicates that a timer your application had previously set has expired. This maps on to the simulator calling your program's *timer_interrupt()* function.
8. **LOSING PACKET** indicates that a packet loss event has occurred in the simulator. The packet that was lost will never be received by the other end of the connection. The packet that was lost is visible in the line above the LOSING PACKET message.
9. **CORRUPTING PACKET** indicates that a packet corruption event has occurred in the simulator. The packet will arrive, but the checksum should indicate that the packet has been corrupted. The packet that was corrupted is visible in the line above the LOSING PACKET message.
10. **WARNING: ATTEMPTED TO START TIMER WHILE ONE IS ALREADY RUNNING** (not shown) indicates that you have attempted to start a timer when another timer was already running. The GBN protocol never has more than one timer running at a time on a specific host.
11. **WARNING: ATTEMPTED TO STOP A TIMER BUT NONE WERE RUNNING** (not shown) indicates that you have attempted to stop a timer but no timer was actually running.

Additional output is shown in the console if a test fails to pass. This output reports the expected state of important variables and the actual state. The differences between these state values may help you to identify what to look for in the log files while determining what is incorrect in your code.

8 Testing your project

Your program will be graded against 14 test cases, including 2 focused on your checksum and 12 simulating a range of different network conditions. The 2 test cases checking your checksum can only be run on Gradescope. The others can be run locally using *rdtll_tester.py*.

The remaining 12 test conditions are based on three factors: packet corruption rate, packet loss rate, and data arrival rate. Logs from my reference implementation for each of the final 12 test cases are included in the

template files. The autograder checks the validity of your code by comparing your program's behavior against the behavior you will see in the logs. In other words, a correct implementation will produce the same behavior as my reference implementation.

1. **Testing your is_corrupt() function**
   (a) Uncorrupt packet
   (b) Corrupt packet
2. **Slow arrival rate (1 packet every 20 seconds)**
   (a) No loss, no corruption
   (b) 25% loss, no corruption
   (c) No loss, 25% corruption
   (d) 25% loss, 25% corruption
3. **Medium arrival rate (3 packets every second)**
   (a) No loss, no corruption
   (b) 10% loss, no corruption
   (c) No loss, 10% corruption
   (d) 10% loss, 10% corruption
4. **Fast arrival rate (100 packets every second)**
   (a) No loss, no corruption
   (b) 10% loss, no corruption
   (c) No loss, 10% corruption
   (d) 10% loss, 10% corruption

9 Submitting Your Project

You can submit your project through Gradescope (which can be accessed via Canvas). You'll see the Reliable Data Transfer Protocol assignment listed on your dashboard. Click on it and a window will appear where you can drag your code file. You should only submit gbn_host.py.

A batch of tests will begin running once you submit your code. These should complete fairly quickly, after which you'll see what tests you passed and failed.