**Design Document**

Title and Authors

Phase 4: Implementation of RDT 3.0 over an unreliable UDP channel with bit-errors and loss

Team Member(s): Michael Burton, Jocelyn Frechette, Jesse Hayes Lewis

Introduction:

In this project phase, the purpose was to implement the reliable data transfer 3.0 (RDT3.0) protocol to reliably transfer a jpg image over an unreliable UDP channel that has both bit errors and data loss in the channel. The implementation is split between two files, one sender (UDPClient.py) and one receiver (UDPServer.py). For the implementation of RDT3.0, the sender was modified from phase 3 of this project to account for the loss of data. To account for loss in the channel, timers are introduced. The introduction of timers allow for any lost packets to be resent over the channel once a timeout is reached. The timeout window is defined by the user with a reasonable timeout window between 10 and 50 milliseconds. To introduce loss into the channel, packets are simply not sent, packets are chosen in randomized fashion that is similar to randomization of bit errors that were introduced in phase 3 of this project. The receiver remains largely unchanged from the previous phase, the only modification made on the receiver is the introduction of the loss of ACK packets. The logic remains similar to RDT2.2 protocol receiver.

Flowchart for sender:

A diagram of a flowchart

Description automatically generated

Code Description:

The description of the code will be split by the sending (UDPClient.py) and the receiving side (UDPServer.py) each broken down into their respective code blocks.

UDPClient.py

A screenshot of a computer

Description automatically generated

In this code block, the python modules necessary to execute the code are imported and all constants are defined including, the states and overall packet structure.

A screen shot of a computer program

Description automatically generated

The MakePacket() function takes in filename and a packet size. The file is open using the open() function. We begin assembling the packet by appending the ACK field, the sequence number, and the data for the desired packet size. Then, the checksum is calculated using the checksum() function which is then appended into the individual packet. This initial step is for the first packet of data to be sent out. In the while() loop, this process is repeated until the end of the file is reached.

A screenshot of a computer screen

Description automatically generated

This block of code is a block functions used to implement the state machine. The split\_ack\_packet() function chops up the received packet and returns the packet in its respective parts which includes the checksum, the ACK response, and the sequence number. The is\_ACK() function takes the split packet and confirms that ACK sequence is what we are expecting. The is\_corrupt() function checks to if the received checksum and the calculated checksum on are not the same. If they are not same, then the packet is corrupt.

A computer screen with text

Description automatically generated

The corruptor() function takes in a byte object, which in the case can either be data packet or an ACK packet, performs a bit-wise NOT which is AND with 0xFF (255 in decimal) bit mask. The corrupted packet is returned to the main script.

A computer screen with white text

Description automatically generated

The code block above shows the rand\_indices() function which is new from phase 3. Given that loss and corruption is introduced. A more generalized function is introduced. The function takes in the packet list and percent of packets to either be corrupted or lost. The number of packets are calculated based on the percentage. Then the number of indices to be corrupted are generated by the numpy random.choice generator which follows a uniform distribution without replacement. The indices in the packet list to be lost or corrupted are then sorted and returned back to the main part of the script.

A screen shot of a computer program

Description automatically generated

The checksum() function takes the data portion of the packet and calculates a custom checksum. The custom checksum takes a sum of all the bytes plus the second byte, and then takes 2 bytes from the end, and returns that as a checksum.

A computer screen shot of a black screen

Description automatically generated

The code block above shows the initialization portion of the UDPClient class, which sender portion of the code. In this code block, the initial state is initialized as well as the port and socket is defined.

A computer screen shot of a program

Description automatically generated

The code block above are the send() and receive() methods within the UDP Client class. The send() method uses the socket.sendto() function to send packets from the sender to the defined receiver port and IP address. The receive() method is where the timer for the RDT 3.0 protocol is defined. The socket.settimeout() function takes in the timeout window that is set by the user upon initialization of the UDPClient code. If there is no timeout present, the message will fall under the “try” case which will send out over the UDP channel. In the case of error/loss in the channel, the recvfrom() has a blocking call which not allow the code to move forward without the timeout present. So when the timer timeouts, a “Timeout Error” will be returned by the “except” case. Further explanation of this “Timeout Error” is explained in the state machine code block which is to follow.

A screen shot of a computer program

Description automatically generated

In this code block is the first portion of the RDT3.0 state machine implementation. The state machine implementation is defined by the next\_state() method under the UDPClient class. The next\_state() method takes in instantiation of the UDPClient, the size of the packet, and a boolean for packet loss. The first if statement, is the wait for class from above state, if no loss is present, the packet will be sent and the state is advanced to waiting for an ACK from the receiver. The elif statement calls the receive method to receive an incoming ACKs, if a timeout error has occurred a packet is resent and the state is returned to the wait for ACK state. If a packet is received, the received packet is split into its respective parts and checksum is calculated. The packet is then checked for corruption and if it received the proper ACK from the receiver. If all is correct, the state is advanced to Wait for call from above for the sequence number.

A computer screen shot of a program code

Description automatically generated

The code block above is the second half of the state machine under the next\_state() method. First the sender will attempt to receive a message from the receiver, if the receiver gets a timeout, the packet will be resent and the state is returned to waiting for an incoming ACK. If the packet is received from the receiver, the packet is split into its respective parts and the checksum is calculated. Error checking is then executed checking for bit errors and if the ACK was for the proper sequence number. If all is correct, the state is then returned to wait for call from above for the next sequence number. Else if its returned for waiting ACK of the previously sent packet. Lastly, error state if there any errors present should there be any errors.

A screenshot of a computer

Description automatically generatedA screenshot of a computer program

Description automatically generated

The code block is the where the UDPClient is run. To initialize the sender, the IP address is hardcoded as the special loopback, then the user prompted for the port number for the receiver, the corruption percentage, the loss percentage, and the timeout window. The user input for the timeout window is then converted into milliseconds and then sender is then initialized. Once the client is initialized, the jpg is then split up into packets with a data size of 1024 bytes per packet. Once the packets are defined, the indices for corruption and dropped packets are calculated by the rand\_indices() function. With everything defined, the main while loop is initiated, which will start the state machine. In this loop, the indices for corruption are checked and those respective packets will be corrupted. Additionally, the lost packets will be checked and not sent to simulate packet loss in the UDP channel. The state is advanced to the next state, if the state is in waiting for the call from above state the packet index is advanced to the next packet for sending. Lastly, performance timers are implemented for this while loop to create performance plots which are shown in a following section.

UDPServer.py

A screenshot of a computer program

Description automatically generated

The first block of code is definition of constants on the server side, this where the states are defined as certain numbers in order to implement logic in the code. Additionally, ACKS and sequence numbers are defined here for proper implementation in the code.

A computer screen shot of a program code

Description automatically generated

Deliver\_data writes the packet\_array (data at this point) to a file\_name in the local directory.

Corruptor – see UDPClient.py explanation

A screen shot of a computer program

Description automatically generated

checksum-see UDPClient.py explanation

A screen shot of a computer program

Description automatically generated

split\_packet-works similarly to split\_ack\_packet from UDPClient.py but gets the data as well and returns it.

A screen shot of a computer

Description automatically generated

is\_ack-see UDPClient.py explanation

is\_corrupt-see UDPClient.py explanation

A computer screen shot of a program

Description automatically generated

UDPServer class, two functions are defined in this class. Once is a static function that defines the name and port of the receiver and sets up the socket for the client. This init function also sets the state of the receiver and creates the data buffer that the application layer reads from. In addition, there is a parameter that keeps track of the number of corrupt ack messages sent. The function called send() is another static function that sends the data to the designated location with a tuple of the IP address and port number. The receive function receives data from the sender.

A screenshot of a computer program

Description automatically generated

This code block is the next\_state() method which is the implementation of the RDT2.2 implementation, with the addition of dropping ACK packets which will be described in the following code blocks. In this code block, the first if statement the received is in the wait for 0 below state, the receiver will receive a packet, split the packet into its respective parts, calculate the checksum of the received message besides the ACK field which does not technically exist for data packets. Error checking is executed to check for any bit errors in the data packet and if it is the sequence number the receiver is expecting. If everything is correct, append the data and deliver to the application layer and then send a positive ACK to the receiver.

A computer screen shot of a computer code

Description automatically generatedA computer code on a black background

Description automatically generated

This code block is the continuation of the state machine, where after the positive ACK is assembled, a random number generator determines whether the ACK packet will be corrupted based on the corruption percentage set by the user. If it determined to be corrupted the ACK will pass through the corruptor. Additionally, a similar approach is taken for dropped ACK packets, a random number generated determines whether the packet will be lost, if the packet will not be lost then the ACK is then sent either corrupted or uncorrupted. The state machine is then advanced to the next state to wait for the next data packet from below. The else statement follows similar logic as what was previously described for this code block . The state is then advanced to the waiting for the next packet from below for the next sequence number.

A computer screen shot of a program code

Description automatically generatedA screen shot of a computer

Description automatically generated

This code block is the rest of the state machine and follows the same logic as previously described. The last else statement returns some number in case there if there are other errors.

A screenshot of a computer program

Description automatically generatedA screenshot of a computer screen

Description automatically generated

The code block above is the UDPServer, the receiving side, is executed. The receiver IP address is first defined and the user is prompted for the port number, the corrupted percentage, and loss percentage. The loss and corruption percentage are then converted into a decimal number. The UDPServer class is then initialized and the server is then bound to the socket. Once the class is initialized, the receiver state is then initialized, the first packet must be received from the sender indicated the number of packets to expect before moving into the main state machine. Once the first packet is received, the main while loop is executed which advanced the states of the state machine. Once the server state is not equal to the receiver state the packet index is incremented by one and then set equal again. Additionally, performance timers are implemented for performance graphs that are shown in following section. The data is then delivered to the application layer and put into a copy.jpg file, which if successful will be a copy of the original jpg image.

Execution Screenshots:

The code execution screenshots will show the following scenarios: No loss or bit error over the channel, ACK packet bit error, data packet bit error, ACK packet loss, data packet loss.

Scenario 1: No loss/bit error

Screens screenshot of a computer

Description automatically generated

Initialization step, with the Client on the left and the server on the right, the timeout on the client side is set to 10 ms. The corruption and loss percentages are set to 0 to introduce no bit error or loss into the UDP channel.

A screenshot of a computer screen

Description automatically generated

Post execution of scenario 1, as it can be shown no loss or bit error was introduced.

Scenario 2: ACK Packet Bit Error

A screenshot of a computer

Description automatically generated

One the server side, ACK packet corruption is introduced at a rate of 15% corruption. No error or loss is introduced on the client side in this scenario.

Screens screenshot of a computer screen

Description automatically generated

Post execution of scenario 2.

Scenario 3 Data Packet Bit Error

Screens screenshot of a computer

Description automatically generated

Initialization of the code for scenario 3, as seen above, a 15% corruption is introduced with a timeout window of 10 ms, and a 0 % packet loss on the sender side. On the receiving side, no bit errors or packet loss is introduced in this scenario.

Screens screenshot of a computer

Description automatically generated

Post execution of scenario 3, showing successful data transfer across the channel.

Scenario 4 ACK Packet loss:

Screens screenshot of a computer

Description automatically generated

Initialization step of scenario 4, as seen above a packet loss percentage on the receiving side at 15% and corruption level of 0%. On the sending side, the corruption and loss percentages are kept at 0% for this scenario.

Screens screenshot of a computer

Description automatically generated

Post Execution of the code for scenario 4. As seen above, there was successful data transfer across the lossy channel.

Screens screenshot of a computer

Description automatically generatedScenario 5 Data packet loss:

Initialization step of scenario 5, the loss percentage is set as 15% and corruption level is set to 0% for this scenario on the sending side. One receiving side, the corruption and loss percentages are kept 0% for this scenario.

A screenshot of a computer

Description automatically generated

Post execution step of the data packet loss scenario. The screenshot above shows successful data transfer across the lossy channel.

Performance Plots:

**Client-Side Packet Corruption:**

The following data points were captured by sweeping the random-corruption rate setting on the Client-Side from a value of 0% to 60%, in steps of +5%. At each Client-Side corruption rate, the Elapsed Time shown is the average across 10 separate runs at that specific Client-Side Corruption setting.

Constants for Client-Side Corruption vs Completion Time:

* Port 4005 used for all tests.
* Server-Side Corruption set to 0%.
* Client-Side Loss set to 0%
* Server-Side Loss set to 0%
* Timeout = 40ms

From the collected data we can see both the client completion time and server completion time increase linearly together as the Client-side corruption percentage is increased, as seen by the extrapolated trend line for each respective completion time.

**Client-Side Packet Loss:**

The following data points were captured by sweeping the random-packet loss rate setting on the Client-Side from a value of 0% to 60%, in steps of +5%. At each Client-Side loss rate, the Elapsed Time shown is the average across 10 separate runs at that specific Client-Side loss setting.

Constants for Client-Side Corruption vs Completion Time:

* Port 4005 used for all tests.
* Server-Side Corruption set to 0%.
* Client-Side Corruption set to 0%
* Server-Side Loss set to 0%
* Timeout = 40ms

From the collected data we can see both the client completion time and server completion time increase linearly together as the Client-side loss percentage is increased, as seen by the extrapolated trend line for each respective completion time.

**Server-Side Packet Corruption:**

The following data points were captured by sweeping the random-corruption rate setting on the Server-Side from a value of 0% to 60%, in steps of +5%. At each Server-Side corruption rate, the Elapsed Time shown is the average across 10 separate runs at that specific Server-Side Corruption setting.

Constants for Server-Side Corruption vs Completion Time:

* Port 4005 used for all tests.
* Client-Side Corruption set to 0%
* Client-Side Loss set to 0%
* Server-Side Loss set to 0%
* Timeout = 40ms

From the collected data we see both the Client and Server completion times increase relatively close to one another as the Server-side corruption increases. Once again, we see behavioral differences between sweeping the client-side corruption setting and the server-side corruption setting. Instead of seeing a linear increase in total completion time we see an **exponential** increase as the server-side corruption is increased. This is most likely due to the way our client-side code handles invalid or missing Ack messages from the server.

**Server-Side Packet Loss:**

The following data points were captured by sweeping the random-Packet loss rate setting on the Server-Side from a value of 0% to 60%, in steps of +5%. At each Server-Side loss rate, the Elapsed Time shown is the average across 10 separate runs at that specific Server-Side loss setting.

Constants for Server-Side Corruption vs Completion Time:

* Port 4005 used for all tests.
* Client-Side Corruption set to 0%
* Client-Side Loss set to 0%
* Server-Side Corruption set to 0%
* Timeout = 40ms

From the collected data we see both the Client and Server completion times increase relatively close to one another as the Server-side loss increases. Once again, we see behavioral differences between sweeping the client-side corruption setting and the server-side loss setting. Instead of seeing a linear increase in total completion time we see an **exponential** increase as the server-side loss is increased. This is most likely due to the way our client-side code handles invalid or missing Ack messages from the server. Another item of note when comparing phase 3 results to phase 4 results, we see a much larger value for Elapsed Time as the corruption/loss parameters are increased towards 60%. We see delays of up to 70 seconds to transfer the file using only timeout’s and no invalid Ack ID’s like we did in Phase 3.