1 Learning Objective

During this project, you will implement the simplest of Transport Layer Protocols, the Stop-and-Wait Protocol for the reliable transfer of data through an artificial network. Specifically you will:

- Demonstrate how messages are segmented into packets and how they are reassembled.
- Understand why a checksum is needed and when it is used.
- Understand and implement the Stop-and-Wait Protocol with ACK (Acknowledgments), NACK (Negative Acknowledgments), and re-transmissions.

For a description of the Stop-and-Wait Protocol, read Section 13.6.1 in your textbook.

2 The Protocol Stack

Here is a diagram that shows where the code in this project fits into the protocol stack:

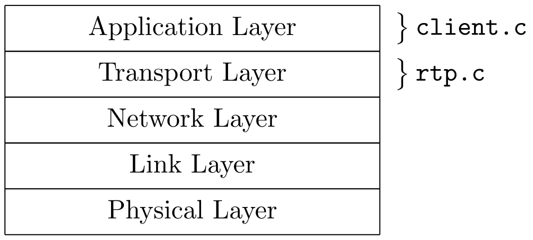

Figure 1: The Protocol Stack

Note that the Network Layer, Link Layer, and Physical Layers are all implemented by your operating system or networking hardware. For the sake of this project, you will be using the UDP transport protocol to emulate an unreliable network layer. The application program, `client.c`, simply makes the appropriate calls to connect to the remote server, send and receive messages, and disconnect from the remote server. You will be providing the send and receive features for the client to use in `rtp.h`.

3 Code Walkthrough

Here, we will briefly describe the code provided for this project. It is important that you study and understand the code given to you. In the past, we have asked students to write a large portion of this code, but we have found this to be unreasonable during the last few weeks of school. As a compromise, we have ginve you this code and simply ask you to complete it. However, YOU ARE RESPONSIBLE FOR UNDERSTANDING ALL OF THE CLIENT'S CODE.

The client program takes two arguments. The first argument is the IP address of the server it should connect to (such as 127.0.0.1, the localhost), and the second argument is the port number it should connect to. You may call the client from the terminal like so:

```
$ ./prj5−client 127.0.0.1 8080
```

The `client.c` program represents the application layer. It uses the services provided by the Transport Layer (`rpt.c`). It begins by connecting to the remote server. Look at the `socket_setup()` in `rpt.c` - it creates the UDP socket that is used to emulate an unreliable network. The `socket_setup()` function also initializes and returns a `CONN_INFO` structure. Next, the client program sends a message to the remote server using the `rtp_send_message()` function. The client program also receives messages from the network. If the message isn't available or if the entire message has not been received, then the `rtp_receive_message()` function should block until the message has been received from the server. The client program continues to send and receive messages until it is finished. Lastly, the client program calls `rtp_disconnect()` in order to terminate the connection with the remote server. Note that there are four distinct types of Transport Layer packets:

1. `DATA` is a data packet that contains part of a message in its payload.
2. `LAST_DATA` is just like a data packet, but it also says that it is the last packet for the message.
3. `ACK` is an acknowledgment that a packet was properly received.
4. `NACK` is a negative acknowledgment that the packet was not properly received.
5. `TERM` is used to tell the server to close the connection early (this is not specifically needed for the project).

4 Implementation

4.1 Segmentation of Data

When data is sent over a network, the data is chopped up into one or more parts and each part is sent inside a packet. A packet contains information that describes the message; it contains information such as the source and destination of the packet, the type of data, and the data itself. The data being sent over the network is referred to as the "payload". Look in `rtp.h` - what other fields does our network packet carry? Think about why each field is needed. How much payload can we fit into each packet? Note that the packet structure in this project is greatly simplified.

Open up `rtp.c` and find the `packetize()` function. Complete this function. Its purpose is to turn a message into an array of packets. It should do the following:

1. Allocate an array of packets big enough to carry all of the data.
2. Populate all the fields of the packet including the payload. Remember, the last packet should be a `LAST_DATA` packet. All other packets should be `DATA` packets. This is very important! The server does check for this, and it will disconnect if these fields are not filled in properly. If you do not mark the last packet is `LAST_DATA`, the server will hang forever!
3. The "count" variable *points* to an integer. Update this integer, setting it equal to the length of the array you are returning.
4. Return the array of packets.

Hint: Remember that when you divide the length of the message by the length of the packet you are performing integer division. It may not always be the case that the message is a multiple of the packet size. Think of a way to handle this special case.

4.2 Transport Protocol

Bad things happen in an unreliable network. In the Stop-and-Wait protocol, sending a message requires the following individual steps:

1. Send one packet at a time.
2. After each packet, wait for an `ACK` or `NACK` to be received.

3. If a `NACK` is received, resend the last packet. Otherwise, send the next packet.

Receiving a message requires the following additional steps:

1. Compute the checksum for each packet payload upon arrival.
2. If the checksum does not match the checksum reported in the packet header, send a `NACK`. If it does match, send an `ACK`.

Note that most Stop-and-Wait Protocols also implement a sequence number to handle lost packets. We will ignore this issue in order to keep the project simple. You should think about why it may be necessary to keep track of sequence numbers.

Complete the following:

1. Open `rtp.c` and find the `checksum()` function. Complete this function. Simply sum up the ASCII values of each character in the buffer and return the sum. This is how the server computes the checksum. The server and the client use this checksum.
2. Open `rtp.c` and find the `rtp_receive _message()` function. This message is for receiving messages from the data connection. To do this, you will use `CONN_INFO` and the `recvfrom()` function. If the packet is a `DATA` packet, the payload will be added to the current message buffer. The packet's payload should only be added to the buffer if the checksum of the data matches the checksum in the packet header. if the checksum matches, you should send an `ACK` packet using `sendto()`. If the checksum does not match, then you should send a `NACK` packet in the same manner.
3. Open `rtp.c` and find the `rtp_send_message()` function. This function should handle packetizing the message and sending the packets one by one using `sendto()`. You should then accept the server's response using `recvfrom()`. If the response is an `ACK`, you should send the next packet. Otherwise, you should resend the packet. Continue to do this until all of the packets have been sent.

5 Short Answer

Give your answers to the following questions in the provided `answers.txt` file:

A. How does the protocol implemented in this project ensure that the entirety of a sent message is received?

B. What is the benefit of dividing a message over multiple packets rather than sending it as a single packet?

C. How might you improve the protocol implemented in this project in order to finish sending messages more quickly?

6 Running the Project

To compile all of the code, use the following command:

```
$ make
```

To run the server on linux, use the following command to run the server:

```
$ ./prj5-server -p [port number] [-c corruption_rate]
```

To run the server on mac, use the following command to run the client:

```
$ python prj5−server −p [port number] [−c corruption_rate]
```

Regardless of operating system, make sure that `python` is bound to python 2. The server will not run on python 3! Your client should work with an UNMODIFIED version of the server. For example, if you wanted to run a server on port 8080 with a corruption rate of 99%, you would execute the following command:

```
$ ./prj5−server −p 8080 −c .99
```

If you wanted to run a client that would send messages to this server, you would then execute the following command (in a different terminal):

```
$ ./prj5−client 127.0.0.1 8080
```

The server will take the client's messages and then convert them into Pig Latin. The server will be printing out debug statements in order for you to understand what it is doing.

To make a `.tar.gz` simply execute the following command:

```
$ make submit
```

Make sure to submit this archive on Canvas.