1 Introduction

This project will expose you to each layer in the network stack and its respective purpose. In particular, you will "upgrade" the transport layer of a simulated network to make it more reliable.

As part of completing this project, you will:

- further explore the use of threads in an operating system, especially the network implementation.
- 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 ACKnowledgements, Negative (NACK) Acknowledgements, and retransmissions.
- create your own new protocol called Reliable Transport Protocol (RTP).

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

Our system consists of a client and a server. The client will be sending encrypted messages to the server. The server will then decrypt the messages and reply with the decrypted message. Then, the client will print out the full, decrypted, message that it received from the server.

2 Requirements

As you work through this project, you will be completing various portions of code in C. There are two files you will need to modify:

- `rtp.c`: the main RTP protocol implementation
- `rtp.h`: to add any necessary fields to the `rtp_connection_t` struct

As you should strive for any programming assignment, we expect quality code. In particular, your code must meet the following requirements:

- The code must not generate any compiler warnings
- The code must not print extraneous output by default (i.e. any debug `printf`s must be disabled by default)
- The code must be reasonably robust and free of memory leaks

Code that does not meet these requirements may lose points.

3 The Protocol Stack

We have provided you with code that implements the network protocol:

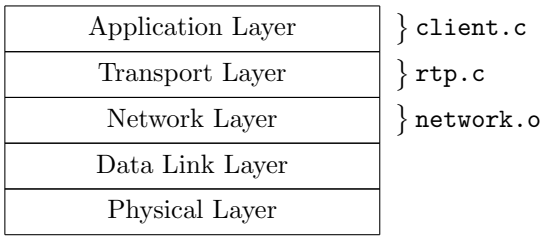
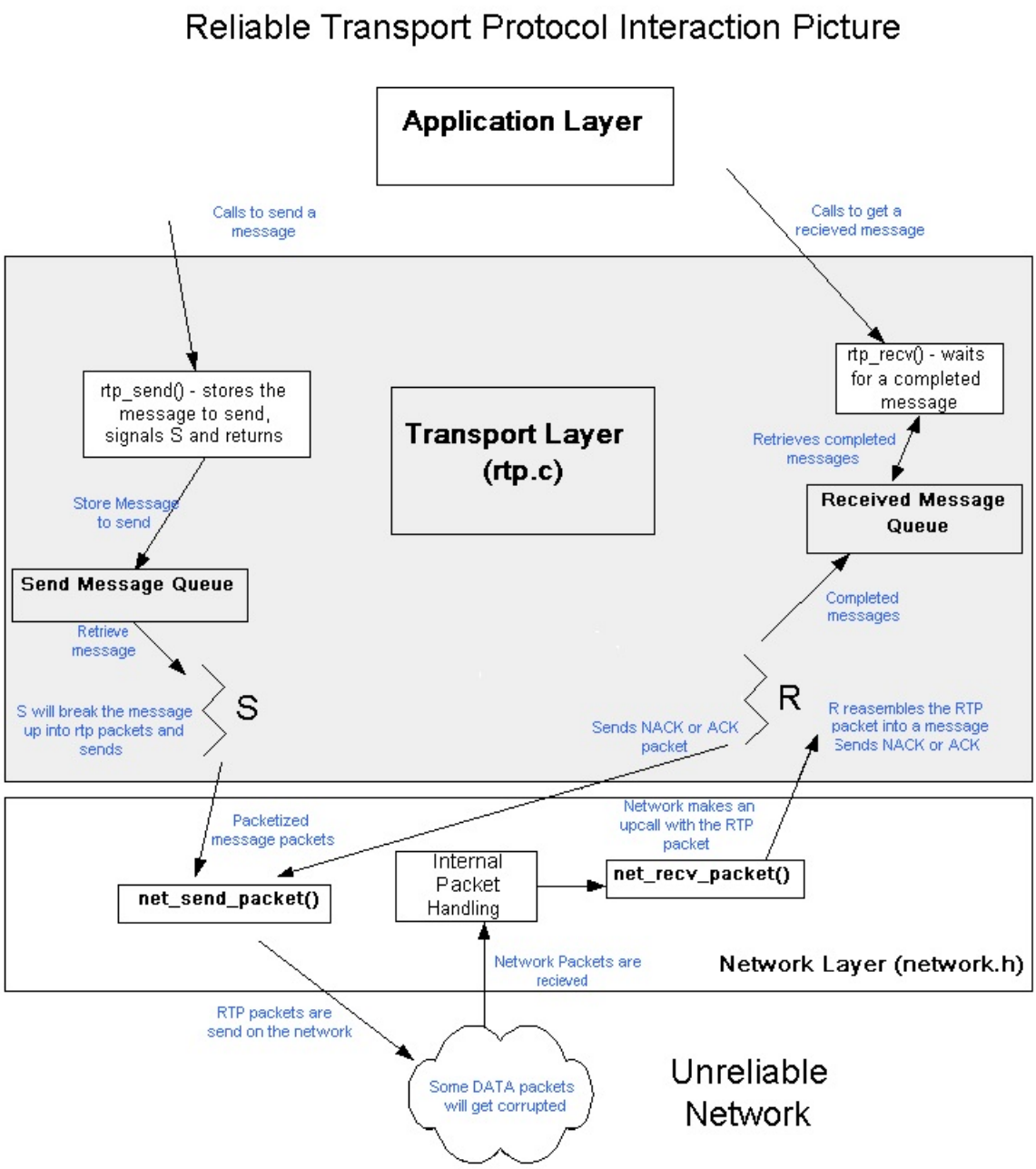


Figure 1: The Protocol Stack

- For the purpose of this project, the data link layer and the physical layer are both implemented by the operating system and the underlying network hardware.
- We have implemented our own network layer and provided it to you through the files `network.h` and `network.c`. You should use the provided functions from those files to access the network layer.
- The transport layer uses the services of the network layer to provide a specialized protocol to the application. The transport layer typically provides TCP or UDP services to the application using the IP services provided by the network layer. **For this project, you will be writing your own transport layer.**
- The application layer represents the end user application. The application simply makes the appropriate API calls to connect to remote hosts, send and receive messages, and disconnect from remote hosts. The message will be sent in encrypted with ROT13, and decrypted when receiving the actual message.

4 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. The following diagram displays the interactions between various parts of the code:



The client program takes two arguments. The first argument is the server it should connect to (such as localhost), and the second argument is the port it should connect to (such as 4000). Thus, the client can be run as follows:

```
$ ./rtp-client localhost 4000
```

4.1 High-level Logic

The `client.c` program represents the application layer. It uses the services provided by the transport layer (`rtp.c`). It begins by connecting to the remote host. Look at the rtp_connect connection in `rtp.c`. It simply uses the services provided by the network layer to connect to the remote host. Next, the `rtp_connect`

function initializes its `rtp_connection` structure, initializes its send and receive queue, initializes its mutexes, starts its threads, and returns the `rtp_connection` structure.

Next, the client program sends a message encode using ROT13 cryptography (the letters of the alphabet are offset 13 places) to the remote host using `rtp_send_message()`. The `rtp_send_message()` message makes a copy of the information to send, places the message into a send queue, and returns so that the application can continue to do other things. A separate thread, the `rtp_send_thread` actually sends the data across the network. It waits for a message to be placed into the send queue, then extracts that message from the queue and sends it.

Next, the client program receives a decrypted message from the network. What happens if a message isn't available or the entire message has not yet been received? The `rtp_receive_message()` function blocks until a message can be pulled from the receive queue. The `rtp_recv_thread` actually receives packets from the network and reassembles the packets into messages. Once it receives a message, it places the message into the receive queue so that `rtp_receive_message` can extract it and return it to the application layer.

The client program continues to send and receive messages until it is finished. Last, the client program calls `rtp_disconnect()` to terminate the connection with the remote host. This function changes the state of the connection so that other threads will know that this connection is dead. The `rtp_disconnect()` function then calls `net_disconnect()`, signals the other threads, waits for the threads to finish, empties the queues, frees allocated space, and returns.

4.2 Packets and Types

For the purposes of this project, there are five packet types:

- `DATA` - a data packet that contains part of a message in its payload.
- `LAST_DATA` - just like a data packet, but also signifies that it is the last packet in a message.
- `ACK` - acknowledges the receipt of the last packet
- `NACK` - a negative acknowledgement stating that the last packet received was corrupted.
- `TERM` - tells the server to shut down (you don't need to worry about this one as it's only used in the provided code).

The packet format is defined in `network.h`. Each packet has a `payload`, which can be up to `MAX_PAYLOAD_LENGTH` bytes, a `payload_length` indicator, `type` field, and a `checksum`.

5 Part I: Segmentation of Data

When data is sent over a network, the data is chopped up into one or more parts and sent inside packets. A packet contains information that describes the message such as the destination of the data, the source of the data, and the data itself! The data being sent over the network is referred to as the 'payload'. Look in network.h; what other fields does our network packet carry? Think about why each field is needed. How much payload data can we fit into each packet? (Note: as with many things in this project, the packet data structure is simplified).

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

1. Allocate an array of packets big enough to carry all of the data. (The provided code in the send thread function will free this array once it is done).
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 IMPORTANT. The server checks for this, and it will disconnect you if they are not filled in correctly. If you neglect the `LAST_DATA`

packet, your program will hang forever waiting for a response from the server, because it is waiting on you forever to send a terminating packet.

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 this is integer division. If `length % MAX_PAYLOAD_LENGTH = 0` this is a special case that should be handled.

There are several other parts of the source code that say FIX ME. The code to be inserted in these parts of the program will simply provide additional functionality but are not necessary at this time. We will return to these parts of the code in Part II.

6 Part II: When Things Go Wrong

In the stop-and-wait protocol, the sending thread does the following things:

1. Sends one packet at a time.
2. After each packet, wait for an ACK or a NACK to be received.
3. If a NACK is received, resend the last packet. Otherwise, send the next packet.

The receiving thread should:

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.

(Part A) Open rtp.c and find the `checksum` function. Complete this function. If the index of the character is odd, find the ASCII values of the character multiplied by its index. If the index of the character is even, simply find the ASCII values of the character. Finally, return the sum of all the values calculated. This is how the server computes the checksum and the server and client must compute the checksum the same way.

(Part B) Open rtp.c and find the `rtp_recv_thread` function. If the packet is a DATA packet, the payload is added to the current buffer. Modify the implementation so that the data is only added to the buffer if the checksum of the data matches the checksum in the packet header. Next, implement the code that will signal the sending thread that a NACK or ACK has been received. You will also need to determine a way to tell the sending thread whether a negative or positive acknowledgement was received. (Hint: it's ok to add fields to the `rtp_connection_t` data structure).

(Part C) Open rtp.c and find the `rtp_recv_thread` function. Find the line that says FIX ME: Part II-C. At this point in the function, an entire message has been received. Implement the code such that a new message variable is allocated, and it's respective fields have been assigned with the buffer's contents and length. Then, add that message variable to the rtp client's queue using the provided `queue_add` function. Do this in a thread-safe manner, and signal the `recv_cond` condition variable to let the client know a full message has been received.

(Part D) Open rtp.c and find the `rtp_send_thread` function. Find the line that says FIX ME: Part II-D. At this point, you should wait to be signaled by the receiving thread that a NACK or ACK has been received. Once notified, take the appropriate action. You should **NOT** call `net_receive_packet` in the send thread. The receiving thread is responsible for receiving packets.

7 Running the Project

First, you will need to make sure that python 3 is installed on your system to run the server. If you do not have python installed, you can install it with the following commands:

```
$ sudo apt-get update
$ sudo apt-get install python3.6
```

Next, to compile all of the code you wrote, use the following command:

```
$ make
```

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

```
$ python rtp-server -p [port number] [-c corruption_rate]
```

Note: if the python command above does not work, you may need to replace `python` with `python3`.

Your client should work with an UNMODIFIED version of the server in python 3. For example, if you wanted to run a server on port 8080 with a corruption rate of 99%, you would execute the following command:

```
$ python rtp-server.py -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):

```
$ ./rtp-client 127.0.0.1 8080
```

You'll want to start an instance of the server first, then run the client.

Warning: You may run into a "Port in Use" error that may look like this:

```
"OSError: [Errno 98] Address already in use".
```

This generally occurs when a server is running in the background. There are many methods to solve this, but we'd recommend using the following commands:

```
$ ps -ef |grep -v 'grep' |  grep 'python3.6 rtp-server.py'
burdell 7302 7261 0 4:20 pts/1 00:00:00 python3.6 rtp-server.py -p 8080 -c 0
$ sudo kill -9 7302
```

Note: The first number is the process ID, or PID, used in the kill command.

Managing multiple terminal sessions can be a pain, so I would recommend using tmux to manage the terminals running your client and the server.

The server will take the client's messages and decode them using ROT13. The server will be printing out debug statements in order for you to understand what it is doing.

8 Deliverables

You need to upload `src/rtp.c` and `src/rtp.h` to Gradescope, and an autograder will run to check if your project is correct. The autograder may take a couple of minutes to run.

Some of the points for the project will be given by the autograder but some will need to be manually graded by the TAs after the deadline. Make sure that you submit the correct version to Gradescope as there will not be time for regrades on this project since it is due at the end of the semester. Additionally, there will be no demos for this project.