CS 2200 Systems and Networking

Prof. Forsyth

Project 5: Networking

Due: **December 1st 2024**

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 (including implementing your own).
- Understand and implement the stop-and-wait protocol with Positive (ACK) 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. You can see a list of those decrypted messages in the client.c file we provide you.

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, including your packetize() function, checksum() function, and send and receive threads
- `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.
- The code must protected shared resources among concurrent threads.

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:

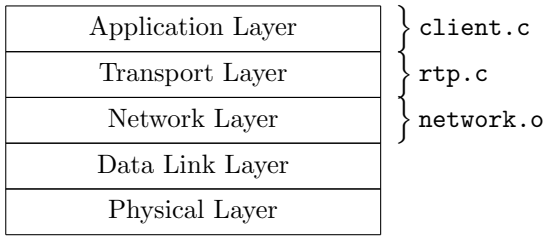
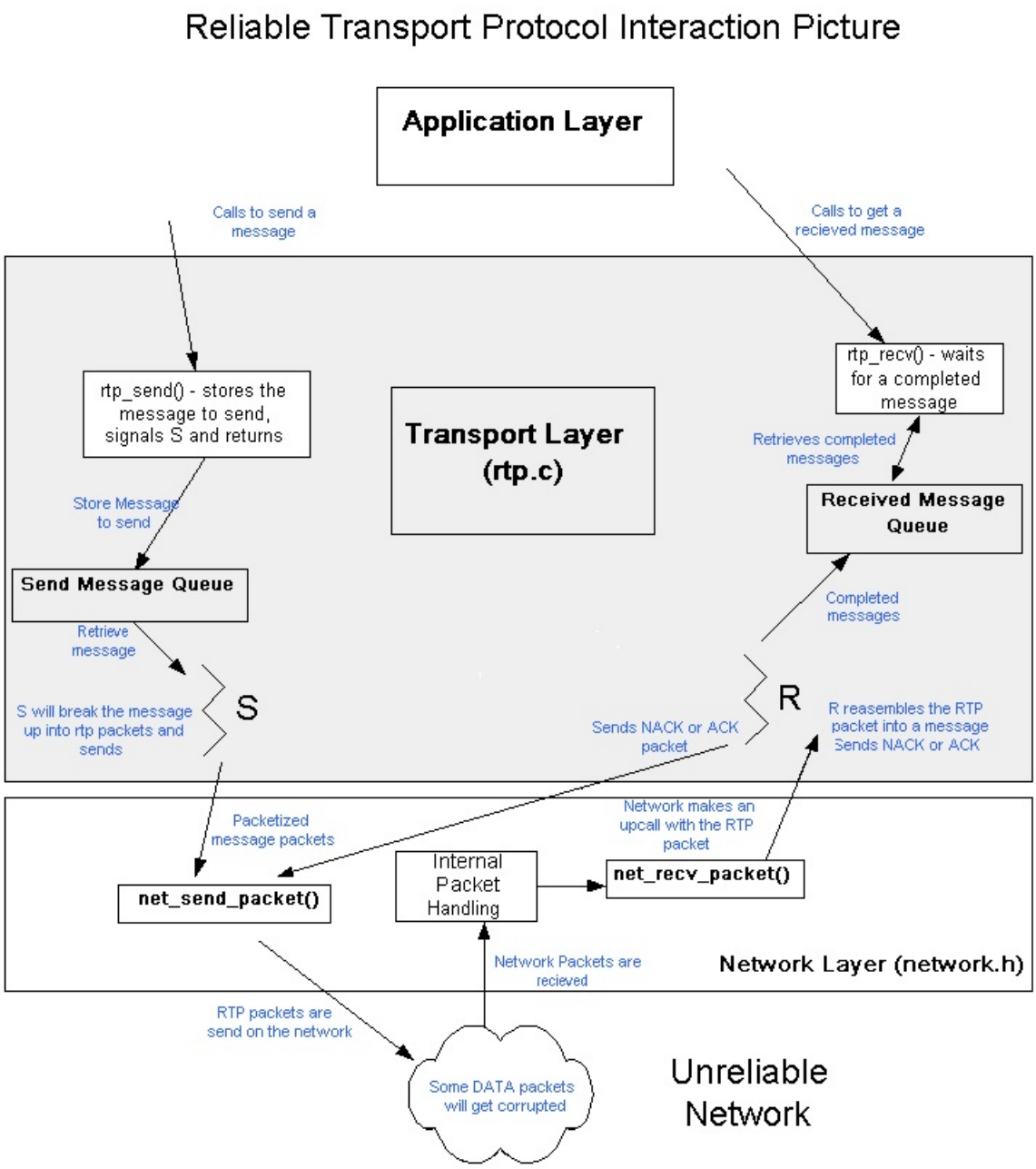


Figure 1: The Protocol Stack

- 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 will have to use the provided functions from these files to access the network layer**. (Hint: when we are ready to send a packet, what function must we call?)
- 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.** This includes implementing the various transport layer services for packetizing and queuing a buffer of arbitrary length.
- 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. In particular, you may want to familiarize yourself with functions available in **network.c**. 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
```

The specific server and port you run on is up to you. We suggest standardizing these arguments in your testing. However, the client and server will need to communicate via the same port, so whichever argument you use to run the server will also need to be used on the client. More debugging tips can be seen in the appendix.

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` (this is a struct). 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. (Don't be afraid to take advantage of the heap. 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.
   (a) 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.
   (b) What is the length of the payload? This might depend on whether we are looking at the last packet or not.
   (c) You can simply call the checksum() function for now (you will implement this as well).
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: Computing the Checksum

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.

How does the receiving thread know whether to send an ACK or a NACK? It utilizes a concept called the `checksum`, which determines whether the packet sent was corrupted by performing a particular weighted algorithm on its characters. Thus the receiving thread will:

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.

This way we can ensure not to send corrupted data up to the application layer.

(Part A) Open rtp.c and find the `checksum` function. In this project the checksum is computed as follows.

1. First, swap each pair of characters. Read the swaps from the left side. For example, if our string is of the form
   $$a_1a_2a_3a_4a_5a_6a_7$$
   we should manually reorder them such that we produce
   $$a_2a_1a_4a_3a_6a_5a_7$$
2. Following this, find the ASCII values of all of the characters. The checksum will be equal to the sum of the each character's ASCII value multiplied by its index value. We are using 0-based indexing.
3. Return the final sum calculated. This is how the server computes the checksum, thus the client must do the same.

As an example, suppose we are given the string "abcdefg". When we reorder it, we obtain "badcfeg". Then we have

$$(b * 0) + (a * 1) + (d * 2) + (c * 3) + (f * 4) + (e * 5) + (g * 6)$$

Note that if the string is of odd length, there is no need to swap the last character. Just continue with the ASCII calculations.

7 Part III: Receive and Send Implementations

Now that our checksum is computed, we can take a hold of the sending and receiving threads and properly communicate between the server and the client. Depending on the values returned by checksum, we will return either a Negative Acknowledgement (NACK) or Positive Acknowledgment (ACK).

(Part A) Open rtp.c and find the `rtp_recv_thread` function. Find the line that says FIX ME: Part III-A. Complete the following steps.

1. If the packet is a DATA packet, the payload is added to the current buffer. Note that we should only do this if the checksum of the data matches the checksum in the packet header.
   (a) Make sure to send the relevant packets to the server so it knows whether to continue sending you packets or resend the last packet.
   (b) Note that if this is the last packet in the sequence, and the packet was corrupted, we don't want the loop to terminate.
2. Next, implement the code that will signal the sending thread that a NACK or ACK has been received. This means producing a mechanism for the sending thread to see which type of acknowledgement it was (hint: it's okay to add fields to the `rtp_connection_t` data structure). Make sure you protect against concurrent modifications where necessary.

(Part B) Open rtp.c and find the `rtp_recv_thread` function again. Find the line that says FIX ME: Part III-B. At this point in the function, an entire message has been received.

1. 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.
2. 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 appropriate condition variable to let the client know a full message has been received (hint: are we accessing the sending queue or receiving queue?).

(Part C) Open rtp.c and find the `rtp_send_thread` function. Find the line that says FIX ME: Part III-C. At this point, you should wait to be signaled by the receiving thread that a NACK or ACK has been received.

1. If notified that it was an ACK, continue as normal.
2. If notified that it was a NACK, resend the last packet.

You should **NOT** call `net_receive_packet` in the send thread. The receiving thread is responsible for receiving packets.

8 Running the Project

Ensure that your files are all within your workspace folder for Docker. First, begin by starting your Docker Desktop and navigating to your project files folder. Make sure you are in the directory that contains the Makefile! To compile all of the code you wrote, use the following command:

```
$ make
```

Recall also that we can use

```
$ make clean
```

to empty out our executable if necessary.

Our project now consists of a Python file we provide you (rtp-server.py) which will run the server, and the executable you just made (rtp-client). We will run these in tandem to start running the project. First, run the server. Open a command prompt and run

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

with your own desired arguments for port number and corruption rate. For example, you might use 8080 for port number, and 0.80 for corruption rate (this has to be a value between 0 and 1). Then you would say

```
$ python rtp-server -p 8080 -c 0.8
```

Note that if the python command above does not work, you may need to replace `python` with `python3`. Next, run the client in a separate command window. You can do this simply by running

```
$ ./rtp-client [host] [port number]
```

With the example from above, we might run

```
$ ./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.

To open a second terminal within the Docker environment, use the `attach.sh` script from the Docker installation zip after using `run.sh` to launch the initial terminal. Managing multiple terminal sessions can sometimes be tricky, so you might be interested in using tmux to manage the terminals running your client and the server if you have trouble.

NOTE: You will need to run this command each time you open your Docker container instance to install Tmux.

```
$ apt-get update && apt-get upgrade && apt-get install tmux
```

From there, you can simply run `tmux` and the application will begin. Follow the rest of the instructions on the article for more details.

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.

9 Debugging Tips

1. In general, this project can be difficult to debug as the print statements in the server file we provide you only say when a packet is received and what the payload was, and not really anything about bugs happening the background. For example, you might run the project and see no output from the server or client. This could happen due to incorrect implementations of packetize (e.g. are you remembering to use the heap?) or checksum. We suggest using GDB for local debugging on client.c and rtp.c.

2. Memcpy is a valid library function for the payload.
3. Make sure you set the fields for the last packet in packetize correctly (in particular, its size).
4. What kind of field in rtp.h could help us know what kind of acknowledgement we have? It might be helpful not to overcomplicate this.
5. Don't be afraid to use the heap in other functions outside packetize as well.
6. In the sending thread, make sure you implement threading correctly (do we want an if or a while loop?).

10 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.