

# 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.

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

# 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 `printfs` 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:

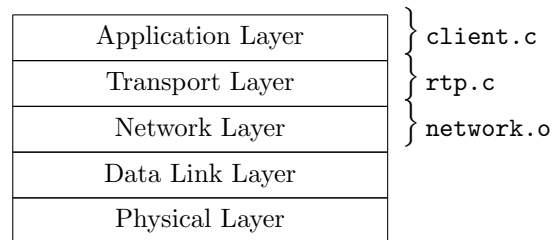


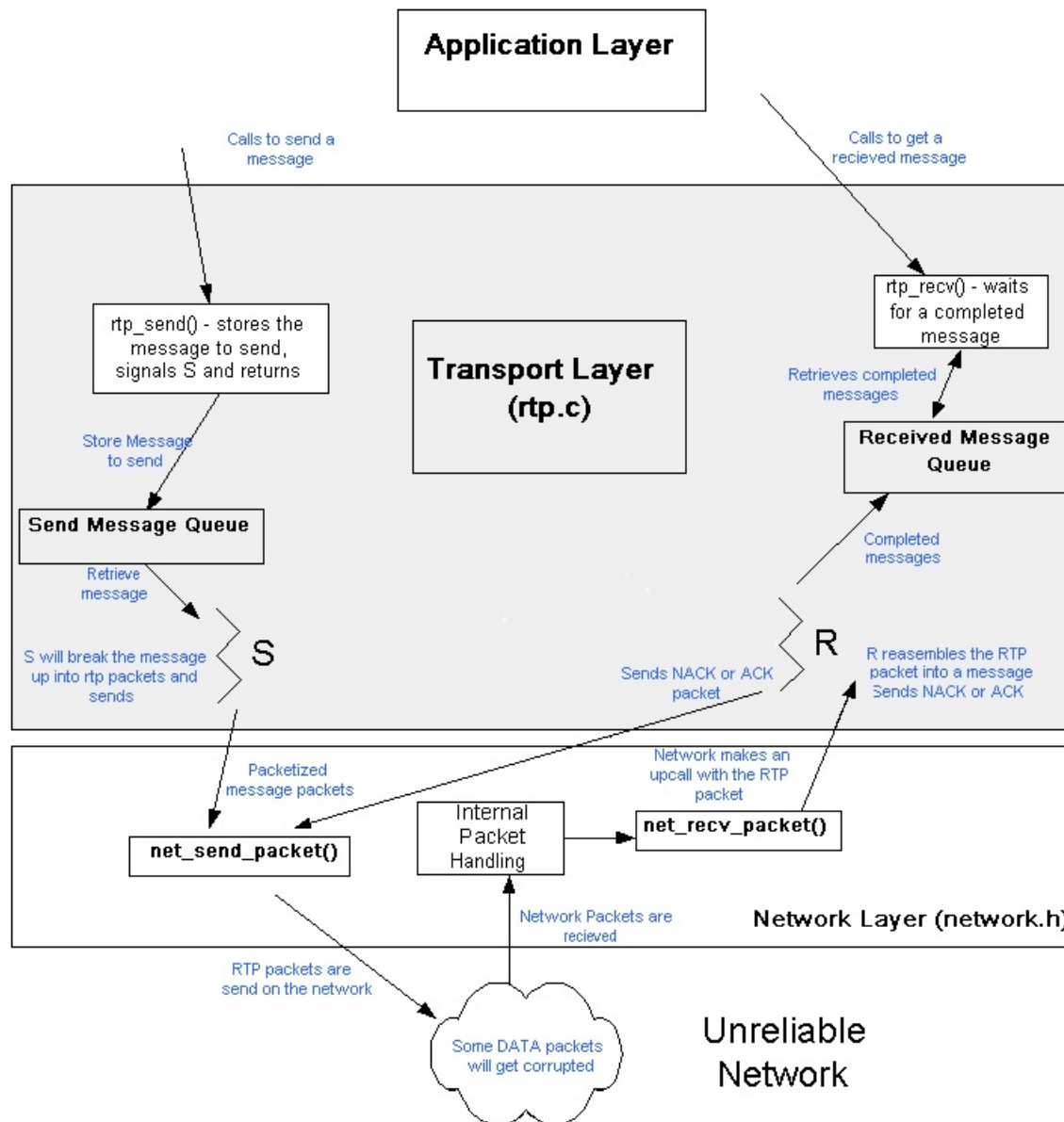
Figure 1: The Protocol Stack

- 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.

### 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:

## Reliable Transport Protocol Interaction Picture



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 to the remote host using `rtp_send_message()`. Sending the message could take quite some time if the network connection is slow (imagine sending a 5MB file over a 56k modem). Thus, 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 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_rcv_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 four 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.

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.
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. Simply sum the ASCII values of each character in the buffer and return the sum. 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_rcv_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_snd_thread` function. Find the line that says `FIX ME`. 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

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:

```
$ python rtp-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:

```
$ 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
```

The server will take the client's messages and reverse the order of the characters. The server will be printing out debug statements in order for you to understand what it is doing.

## 8 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 message is received by the other end, even if packets are lost?
- 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? (Note: You implemented a transport layer protocol in this project. You cannot suggest an improvement to a different layer, such as improving the bitrate.)

## 9 Deliverables

Type `make submit` to generate a tarball containing all the files needed for submission.

This tarball should contain:

- `src/client.c` - Code for the client program
- `src/network.c` - Code for the network layer
- `src/network.h` - Header file for the network layer
- `src/queue.c` - Code for the queue
- `src/queue.h` - Header file for the queue
- `src/rtp.c` - Code for the transport layer
- `src/rtp.h` - Header file for the transport layer
- `Makefile` - Working one provided for you; don't break it
- `answers.txt` - Your answers to the short answer questions

**Always re-download your assignment from Canvas after submitting to ensure that all necessary files were properly uploaded. If what we download does not work, you will get a 0 regardless of what is on your machine.**