Michael Chen & George Dong

Mchen96@gatech.edu & gdong8@gatech.edu

CS 3251 – Computer Networking I – Project 2

27 November 2015

# Files Submitted

- README.pdf: a document containing a detailed description of the protocol as well as a description of how to run the bundled programs.
- NetEmu.py: the unmodified networking emulator, which is used to simulate an unreliable network.
- src/FxAClient.java: the client application. When executed on a system with the appropriate arguments, the application will allow the client to send and receive files from the server.
- src/FxAServer: the server application. When executed on a system with the appropriate command line arguments, the application will allow the server to accept requests from the client.
- src/RxpPacket.java: a class that represents a single packet transmitted over the network. Its main function is to divide the byte arrays into several fields, as per the packet structure discussed later in this report.
- src/RxpSocket.java: the main networking class, this class contains the finite state machine responsible for maintaining the socket state, sending packets, and handling acknowledgements.
- src/RxpServerSocket.java: a server socket whose role is to bind to a port on the server, receive the packets from the UDP socket, and multiplex them based on the client.

# Usage Instructions

## Compile Instructions

The application can be compiled as a standard java application. Note that for the client, the main method is located in the FxAClient class. For the server, the main method is located in the FxAServer class.

## Execution Instructions

First, start the network emulator with the desired operating constraints. The protocol can handle packet duplicates, reordering, losses, and corruptions. (For instance, NetEmu.py 5000 -l 10 -c 10 -d 10 -D 100 -r 25).

After starting the emulator, execute the server with the source UDP port the server should use, the IP address of the network emulator, and the UDP address of the network emulator (For instance, java RxpServer 8001 127.0.0.1 5000). Once the server is active, the terminate command can be used to close the port. The window <size> command may be use to set the size of the window, in segments. Note that the window can only be resized when the new size is larger than the size of the packets in the receive window.

Finally, the client can be started with the client's source UDP port, the emulator IP address and the emulator UDP port as arguments (For instance, java 8000 127.0.0.1 5000). Once the client application has been executed, the connect command should be used to establish a connection with the server. The command get <file> can be used to retrieve a file from the server. The file must be in the same location as the server's application. The command put <file> can be used to send a file to the server. The file must be located in the same directory as the client application.

The window <size> command can be used to set the size of the receive window, in segments. Finally, the disconnect command gracefully disconnects the client from the server.

## Introduction

A reliable transport layer protocol specification was designed in order to create a reliable end to end protocol over an unreliable network layer. The protocol is a go-back-n pipelined protocol that features both positive and negative acknowledgements.

The protocol uses a connection-oriented design, in which the client sends a request to the server in order to initiate a connection. Once the connection is established, no distinction is made between the client and the server; both sides are capable of either transmitting or receiving. Since the protocol makes no distinction between the server and the client once a connection has been established, the protocol is inherently bidirectional.

To handle lost packets, a timeout mechanism is used. If the sender of a packet does not receive a positive acknowledgement from the receiver within a preset timeout, the packet is assumed to be lost; to compensate, the lost packet should be resent. The exact packet lost is identified using packet sequence numbers; a more specific description of this functionality has been included in the next sections.

If a duplicate packet is received, the packet is discarded. However, a positive acknowledgement is also sent to the sender in order to confirm that the packet was successfully received. This scenario may occur when the acknowledgement packet is lost.

If an out of order packet is received, the packet is discarded. Furthermore, a negative acknowledgement must be sent back in order to notify the sender that one of the packets was either lost or delayed and that it should be retransmitted.

A simple addition checksum is used to determine if a packet has been corrupted. If a packet is determined as corrupted via the checksum, a negative acknowledgement is sent. The sender, upon receiving the negative acknowledgement, should retransmit the corrupted packet and every single packet that followed it.

## Packet Structure

The packet header was designed in order to provide the functionality required by the project constraints. Figure 1, which is shown below, shows the structure of the packet header.

<table>
<tr><td colspan="2">Source Port (16 bits)</td><td>Destination Port (16 bits)</td></tr>
<tr><td colspan="3">Sequence Number (32 bits)</td></tr>
<tr><td colspan="3">Acknowledgement Number (32 bits)</td></tr>
<tr><td colspan="2">Payload Length (16 bits)</td><td>Available Window Size (16 bits)</td></tr>
<tr><td>Reserved (12 bits)</td><td>Flags (4 bits)</td><td>Checksum (16 bits)</td></tr>
<tr><td colspan="3">Payload…</td></tr>
</table>

Figure 1: Packet Structure

Two 16 bit shorts contain the source and destination ports. These are used for multiplexing the packets received among different applications. Each port should only be bound to one application at a time.

A 32 bit integer contains the sequence number, which identifies the position of the first payload byte if a payload is present. The next 32 bit integer contains the acknowledgement number, which is the position of next expected byte when receiving a packet.

The following 16 bit short contains the effective length of the data payload only. The size of the header field does not count towards the payload length.

The following 16 bit short states the available size of the receiving window. The available window size is defined as the difference between the total window size and the space occupied by all the received bytes whose acknowledgements have been sent, from the receiver's perspective.

After 12 bits that are reserved for future expansion, the next 4 bits contains flags that are used to signal a packet's purpose. The possible values for these flags are, from least significant to most significant bit, ACK (positive acknowledgement), NACK (negative acknowledgement), SYN (synchronize sequence numbers), and FIN (Finalize connection). Figure 2 shows the order of these flags.

<table>
<tr><td>FIN</td><td>SYN</td><td>NACK</td><td>ACK</td></tr>
</table>

Figure 2: Packet Flags

## Checksum Implementation

The checksum algorithm selected consists of adding together all the bytes in both the header and the packet payload. Each pair of bytes in the header (excluding the checksum) and the payload will be read as a short and added to the results of the preceding addition. If the payload has an odd number of bytes, the last byte will be right padded with zeros before the addition is performed. The result of the addition, truncated to a short, is the value of the checksum.

On the received side, the client performs the exact same addition. If the sum obtained differs from the sum included in the packet headers, the packet must be discarded and a negative acknowledgement must be issued to notify the sender that the packet was corrupted.

## Sequence and Acknowledgement numbers

Sequence numbers are used in order to identify out of order or missing packets. In the proposed protocol, sequence numbers are byte addressed. In other words, a packet with a payload of n bytes increases the sequence number by n. SYN and FIN packets will be treated as one byte-payload packets when increasing the sequence numbers.

The initial sequence number when a connection is initiated should be selected at random; the SYN flag is used to notify the other endpoint which value was selected as the initial sequence number.

The acknowledgement number is used to acknowledge a packet. Every packet must be acknowledged when they are received; the acknowledgement number is always the sequence number of the next expected byte.

## Finite State Diagrams

In order to simplify the finite state diagrams, the diagrams have been divided into two separate diagrams. The first diagram indicates the packets used to initiate and finalize a connection. The second diagram indicates the packet order used to transmit the actual payload once a connection has been initiated.

Figure 3 shows the finite state diagram that is used in order to establish and close a connection. The protocol uses a modified version of the TCP state diagram. The notation used in Figure 3 is as follows: a text over a horizontal line specifies what event will cause a state change. A text under a horizontal line specifies what action should be taken as part of the state change. The character Ø is used to denote whenever no action needs to be taken during a state change.

The most significant change with respect to TCP is that a four way handshake is used instead of a three way handshake. When the connect() method is called by the parent application, the client will send an empty packet with the SYN flag set. Upon receiving this packet, the server should reply with a packet with the SYN and ACK flags set. In addition, this reply should have a four byte random challenge (denoted as NONCE in Figure 3). The client, in turn, should reply with the SHA-256 hash of the nonce (and with the acknowledgement flag set). Note that HASH and NONCE are not represented as flags, the labels in the figure simply indicate the payload of those specific packets.

The server should only allocate the receive buffer once this hash has been received and verified. Finally, the server should also reply an acknowledgement upon receiving the hash, as shown in the figure below. The idea of using a four way handshake that requires work on the client's part before allocating resources can help mitigate some attack vectors such as SYN floods.

The half-open state has also been eliminated. The close wait state no longer exists; a FIN+ACK is sent by an endpoint upon receiving a FIN packet from the opposite party.

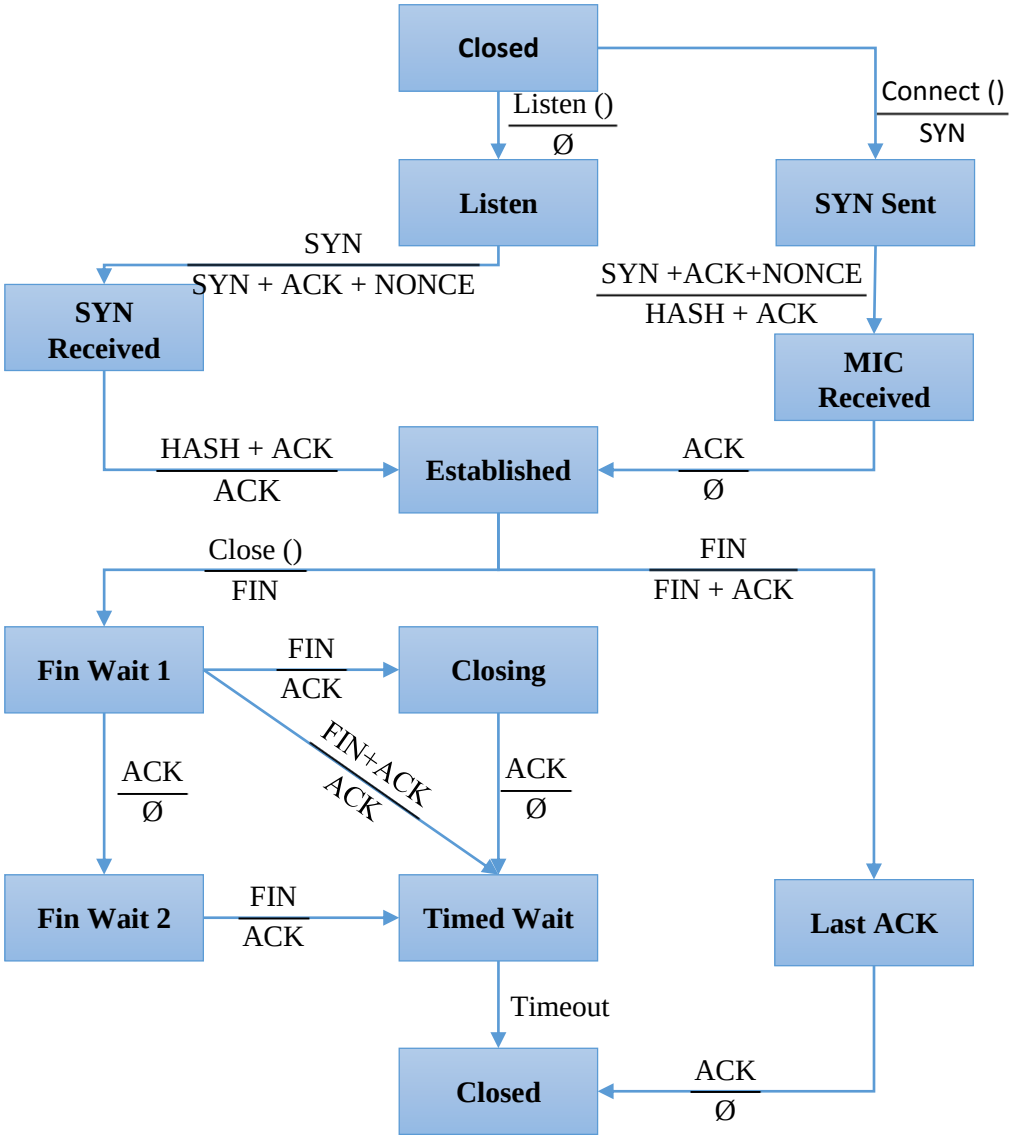
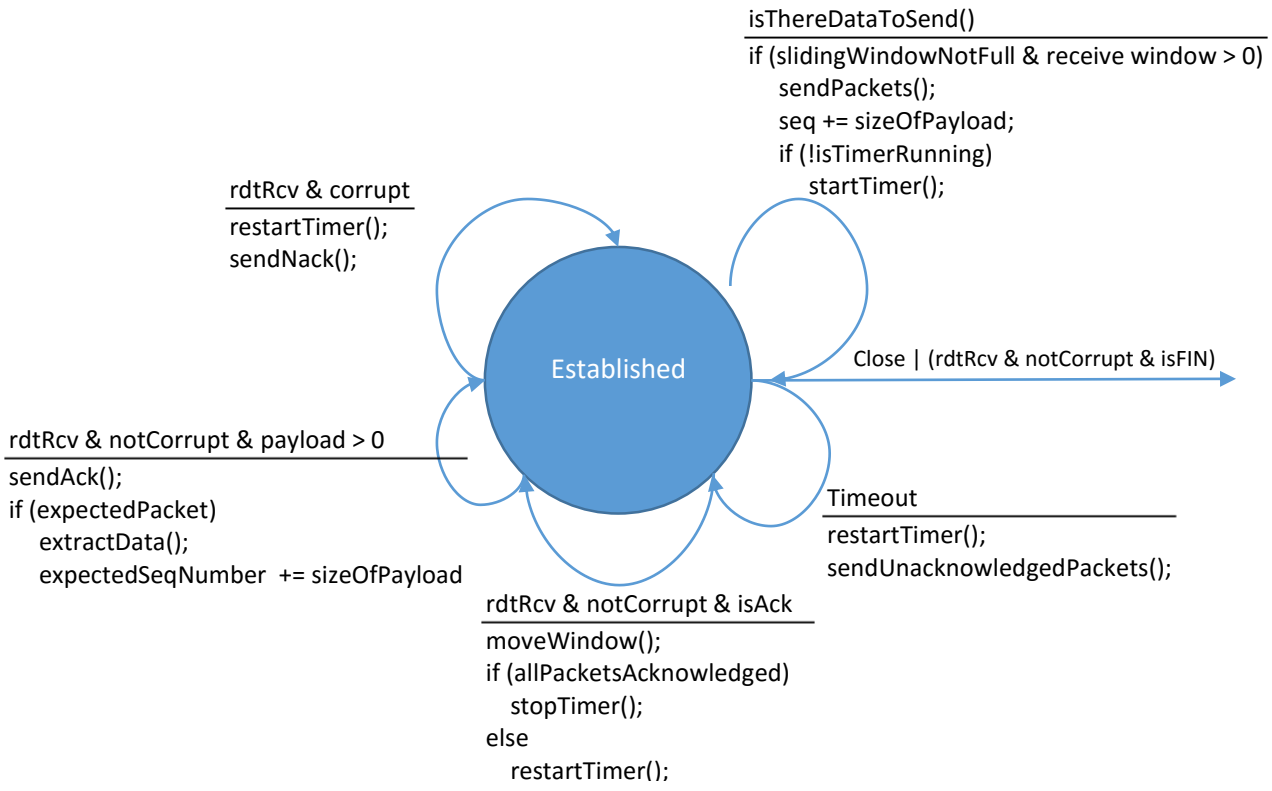



Figure 3: Connection/Disconnection State Diagram

While sending or receiving packets, the actions shown in Figure 4 should be followed. Note that this diagram only applies once the connection has been established. Furthermore, the diagram is not relevant once a close command or a FIN packet is received.

- If there is data to send according to the send buffer, the sliding window used for pipelining is not full, and there is space in the receiver's buffer, one (or more) packets should be sent. Each packet should have the maximum size that does not cause fragmentation at the IP layer if possible. The packet cannot exceed the available receive window length once "in-flight" unacknowledged packets are accounted for. A timer should be started at this point in order to check later that acknowledgements have been received for these packets. Note that if no data can be sent due to a full receiver window, a single packet must be. (not shown in Figure)
- If a timeout occurs due to unacknowledged packets, resend all unacknowledged packets and restart the timer.
- If a corrupt packet is received (regardless of type), send a negative acknowledgement and restart the timer. Note that this must be done regardless of the packet type, since it is unknown if the packet flags themselves are corrupted.
- If an acknowledgement is received, the sliding window should be moved as the acknowledged packets do not need to be kept in the send buffer.
- If a packet containing a payload is received, an acknowledgement should be sent regardless of whether the packet is the expected packet or a duplicate. If the packet was the expected packet, the data should be extracted and the next expected sequence number should be updated.



## Stream Mechanics

The data is sent and received from the socket through an OutputStream and an InputStream, respectively. The streams used in this implementation of the RxpProtocol described above support reading and writing, either one byte at a time or through byte arrays.

It is important to note that the message boundaries are not preserved; one write command will not necessarily correspond to one read command. This implementation attempts to send packets whose size are equal to the maximum segment size; each packet become available to the received as soon as it has been validated.

Note that the optional mark and reset methods in the InputStream are not supported in the current implementation.

## API

The API used to send data, receive data, and initiate connections is based on the RxpSocket class, which is design to operate similarly to the Socket class. A second class, RxpServerSocket, provides the methods required to bind and listen to a specific port. The different methods and constructors available to client applications can be seen below.

### RxpSocket

RxpSocket()

Creates a new, unconnected socket.

void connect (SocketAddress endpoint, int rxpPort, int udpPort)

Connects a port to a specified endpoint. This method is only used by the client in order to initiate a connection with the server.

@param endpoint: the target UDP address and UDP port of the server

@param rxpPort: the target Rxp port of the server

@param udpPort: the client's source udp port

@throws IOException if the server cannot be reached.

Void setWindowSize(int segments)

Sets the number of segments in the sliding window. Note that if the size of the sliding window equals one, the protocol behaves like a stop and wait protocol.

@param segments The number of segments in the sliding window.

Void setBufferSize(int size)

Sets the size of the receive buffer. The default size is 4096 bytes.

@param size The size of the receive buffer, in bytes.

InputStream getInputStream()

Gets an input stream that can be used to read the data from the socket.

OutputStream getOutputStream()

Gets an output stream that can be used to write data to the socket.

Void Close()

Closes the socket, the input stream, and the output stream

Boolean isClosed()

Returns true if the server socket is closed, false otherwise.

## RxpServerSocket

RxpServerSocket()

Creates a new, unbounded server socker

Void listen(int udpPort, int rxpPort)

Binds the server socket to a specific port and begins listening for incoming connections. If a client attempts to connect after this method is called, the connection will be successful and resources will be allocated,

@param udpPort The udp port this instance should be bound to.

@param rxpPort The target rxp port this instance should be bound to.

RxpSocket accept()

Blocks the current thread until a new connection is available. Once the connection has been established, an RxpSocket is returned. This RxpSocket should be used to send and receive data to and from the client.

Void setWindowSize(int windowSize)

Sets the window size of the receiver buffer of the server. This window size will be applied to all the connections the server has opened. The size is specified in segments.

@param segments The number of segments in the sliding window.

Boolean isClosed()

Returns true if the server socket is closed, false otherwise.

Void close()

Closes the current server socket and all associated RxpSockets associated with this port.

## Known Issues

The server currently uses one thread to sequentially handle all clients. While the socket implementation in the lower layer will negotiate connection attempts in a separate thread, the received data will only be processed sequentially.

If a client does not disconnect gracefully from the server (for instance, if the client is closed without writing the command disconnect first), the server will perpetually await data from this client. This occurs because the server does not initiate transactions with the client; therefore no errors that would signal an error are evident from the server's perspective.

A "keep alive" packet would be beneficial to this implementation, as it would allow the server to recognize connections that have been forcefully closed from the client's side. Having one thread per client would also increase the reliability and security of the server against ill-intentioned individuals.