Michael Chen & George Dong

[Mchen96@gatech.edu](mailto:Mchen96@gatech.edu) & [gdong8@gatech.edu](mailto:gdong8@gatech.edu)

CS 3251 – Homework 4

4 November 2015

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

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

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

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

# Package Structure

The package 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 package header.

|  |  |  |
| --- | --- | --- |
| Source Port (16 bits) | | Destination Port (16 bits) |
| Sequence Number (32 bits) | | |
| Acknowledgement Number (32 bits) | | |
| Payload Length (16 bits) | | Available Window Size (16 bits) |
| Reserved (12 bits) | Flags (4 bits) | Checksum (16 bits) |
| Payload… | | |

Figure 1: Package 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 package.

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 package’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.

|  |  |  |  |
| --- | --- | --- | --- |
| FIN | SYN | NACK | ACK |

Figure 2: Packet Flags

# Checksum Implementation

The checksum algorithm selected consists of adding together all the bytes in both the header and the package 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 package headers, the package must be discarded and a negative acknowledgement must be issued to notify the sender that the package was corrupted.

# Sequence and Acknowledgement numbers

Sequence numbers are used in order to identify out of order or missing packages. In the proposed protocol, sequence numbers are byte addressed. In other words, a package with a payload of n bytes increases the sequence number by n. SYN and FIN packages will be treated as one byte-payload packages 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 package. Every package 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 packages used to initiate and finalize a connection. The second diagram indicates the package 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 package with the SYN flag set. Upon receiving this package, the server should reply with a package 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 packages.

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.

**Closed**

**Listen**

**SYN Received**

**SYN Sent**

**Established**

**Fin Wait 1**

**Fin Wait 2**

**Closing**

**Last ACK**

**Close Wait**

**Timed Wait**

**Closed**

Connect ()

SYN

SYN +ACK+NONCE

HASH + ACK

HASH + ACK

ACK

SYN

SYN + ACK + NONCE

Close ()

FIN

FIN

ACK

FIN

ACK

Close ()

FIN

FIN

ACK

Timeout

Listen ()

Ø

ACK

Ø

ACK

Ø

ACK

Ø

**MIC Received**

ACK

Ø

Figure 3: Connection/Disconnection State Diagram

While sending or receiving packages, 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) packages should be sent. Each package should have the maximum size that does not cause fragmentation at the IP layer if possible. The package cannot exceed the available receive window length once “in-flight” unacknowledged packages are accounted for. A timer should be started at this point in order to check later that acknowledgements have been received for these packages. 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 packages, resend all unacknowledged packages and restart the timer.
* If a corrupt package is received (regardless of type), send a negative acknowledgement and restart the timer. Note that this must be done regardless of the package type, since it is unknown if the package flags themselves are corrupted.
* If an acknowledgement is received, the sliding window should be moved as the acknowledged packages do not need to be kept in the send buffer.
* If a package containing a payload is received, an acknowledgement should be sent regardless of whether the package is the expected package or a duplicate. If the package was the expected package, the data should be extracted and the next expected sequence number should be updated.

isThereDataToSend()

if (slidingWindowNotFull & receive window > 0)

sendPackets();

seq += sizeOfPayload;

if (!isTimerRunning)

startTimer();

Timeout

restartTimer();

sendUnacknowledgedPackets();

rdtRcv & corrupt

restartTimer();

sendNack();

rdtRcv & notCorrupt & isAck

moveWindow();

if (allPacketsAcknowledged)

stopTimer();

else

restartTimer();

rdtRcv & notCorrupt & payload > 0

sendAck();

if (expectedPackage)

extractData();

expectedSeqNumber += sizeOfPayload

Close | (rdtRcv & notCorrupt & isFIN)

# 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 port)

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 port: the target Rxp port of the server

@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

## 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 close()

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