CS 3251 - Networking

RxP Design Document

Hunter Brennick & Sherman Mathews

Introduction

Is your protocol non-pipelined (such as Stop-and-Wait) or pipelined (such as Selective Repeat)?

RxP is a Selective Repeat pipelined protocol. The connection will established via a 3-way handshake. Both the RxPClient and RxPServer have send and receive buffers to enable bi-directional pipelined Selective Repeat functionality. The RxPServer knows what the sequence number of the next packet it receives from the RxPClient should be, and vica-versa.

How does your protocol handle lost packets?

The receiver will respond with an ACK for every packet it receives. The timeout is set respective to the last ACK it has received. If a timeout occurs all unacknowledged packets in the send queue will be retransmitted.

How does your protocol handle corrupted packets?

A checksum will be utilized to ensure that bits within our layer weren't corrupted during transit. If the checksum does not match the receiver will trash the packet. The sender will resend the packet on a timeout. We are using the md5 algorithm compute our checksum.

How does your protocol handle duplicate packets?

We will take advantage of sequence numbers to detect and handle duplicate packets. Packets with a sequence number less than or equal to the current expected sequence number can be identified as duplicate packets. Duplicate packets will be discarded by the receiver.

How does your protocol handle out-of-order packets?

Once again, we will take advantage sequence numbers to detect out of order packets. If a packet is out-of-order it will be placed in the receive buffer, if there is space. Out-of-order packets will remain in the receive buffer until the preceding missing out-of-order packet(s) are received.

How does your protocol provide bi-directional data transfers?

Our protocol supports bi-directional data transfers by maintaining a send and a receive buffer on both the client and server. The agent that initiates the handshake to establish the connection starts as the sender, however as the two begin communicating, at any point in time either can be the sender or receiver. Both hosts need to be able to send and receive packets at the same time, hence

the need to manage two separate buffers on each host. The RxP window size header lets the sender know how much data (in bytes) it can send to the receiver.

Does your protocol use any non-trivial checksum algorithm (i.e., anything more sophisticated than the IP checksum)?

Yes, we are using the md5 checksum algorithm to check for corrupted packets.

RxP Design

High-Level Description

RxP is a reliable, connection-oriented transport layer protocol. It operates in a space above UDP. RxP is a simpler, lightweight version of TCP. RxP manages connections much the same way as TCP. It uses handshakes, maintains connections, and has states controlling the flow of communication. RxP will use Selective Repeat for window-based flow control. The application using RxP will open and establish a connection, then begin to stream packets, like TCP. On the receiver end, in-order packets are reconstructed and concatenated, while out-of-order packets are placed in a receive buffer until the missing packets arrive, and then sent to up to the application layer. Our protocol handles packet loss, packet corruption, packet reordering, and duplicate packets as described in the Introduction above.

The connection will established via a 3-way handshake. The RxPClient will initialize the handshake by sending a SYN packet to a specified RxPServer, the RxPServer will synchronize its sequence number for this respective RxPClient and respond with an ACK packet. After a certain period of time the RxPServer will send a SYN packet that will allow the client to synchronize with the server's sequence number. It will then inform the RxPServer it has received said SYN packet with response ACK packet. Both the RxPClient and RxPServer have send and receive buffers to enable bi-directional pipelined Selective Repeat functionality. The RxPClient and RxPServer are aware of each other's receive buffer capacity via each other's respective window size headers. The RxPServer knows what the sequence number of the next packet it receives from the RxPClient should be, and vica-versa.

Header Structure and Fields

RxP Header (X bits total)		
Source Port Number (16 bits)		Destination Port Number (16 bits)
Sequence Number (32 bits)		
Acknowledgement Number (32 bits)		
Window Size (32 bits)		
Frequency (5 bits)	Control Bits [ACK|SYN|FIN] (3 bits) + 8 bit offset	RxP Checksum (16 bits)

Source Port Number (16 bits)

Port number of sending host.

Destination Port Number (16 bits)

Port number of receiving host.

Sequence Number (32 bits)

The sequence number of the first byte of data in this RxP packet. If the SYN control bit is set, the sequence number provided is the initial sequence number.

Acknowledgement Number (32 bits)

If the ACK Control Bit is set, this header field contains the next sequence number the sender of the RxP packet expects to receive. Once a connection is established this is always sent in an RxP packet.

Length (32 bits)

The size of the data in bytes.

Window Size (16 bits)

The number of bytes of data that can be sent before an acknowledgement from the receiver is necessary. Is strongly correlated to the size available in the receive window of the sender.

Frequency (5 bits)

The number of times the packet has been sent.

Control Bits (5 bits, 1 ea)

In order as specified in RxP header table above. Each consumes 1 bit of space in the RxP header.

- **ACK**: Acknowledgment number valid flag.
- **SYN**: Synchronize sequence numbers flag.
- **FIN**: End of data flag.

8 bit offset (8 bits)

The number of 32 bit words the RxP implementer can add to the end of the header

RxP Checksum (16 bits)

Used to detect RxP packet corruption. It is computed using the Adler-33 checksum algorithm.

API Description

RxPSocket API:

- **rxp_socket RxPSocket(int window_size, bool debugger_enabled)**
  - Creates a new RxPSocket. When called (assuming valid parameters), the an RxPSocket is initialized. If the target RxPSocket takes too long to acknowledge a packet, the packet timeout will trigger and the packet will be retransmitted. The packet timeout is relative to the frequency header on the RxPPacket. If a packet times out a certain number of times the RxPClient will close its connection to the RxPServer and notify the connected client application of said closure. The following methods are the public API an initialized RxPSocket exposes.
- **void bind(short port_number)**
  - Associates a socket with a port number.
- **void accept()**
  - Accepts and establishes a connection client. Receives a SYN to initialize the handshake, then responds with a SYN/ACK. Will then receive an ACK in response to the SYN/ACK to begin a new session with a client. This function blocks until a connection is established.
- **void connect(int destinationIP, short destinationPort)**
  - Attempts to establish a connection with the specified destinationIP:destinationPort. First sends a SYN, receives a SYN/ACK, then responds with an ACK.
- **void send(string data)**

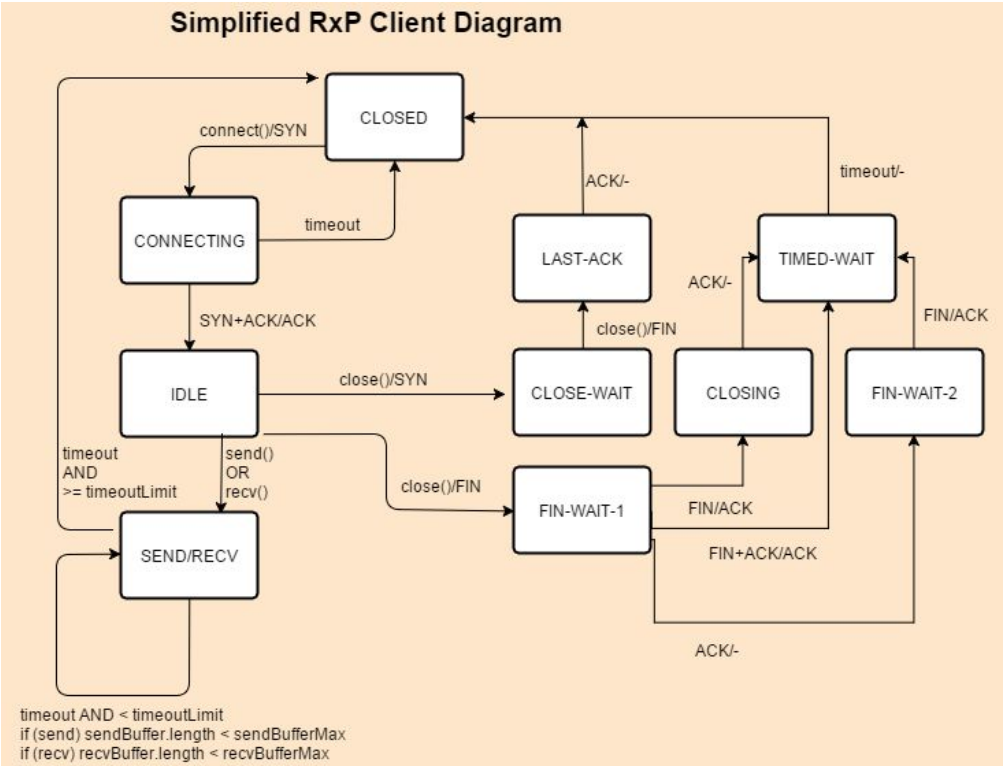
        - Called a sender to stream bytes to a receiver.
    - **string recv(string data)**
        - Called by a receiver to stream bytes from a sender.
    - **void close()**
        - This method will send a packet with the Control Bit FIN set to an RxPServer instance. Receives a FIN/ACK or FIN, then respond with an ACK. If no FIN/ACK or FIN received, a timeout occurs.

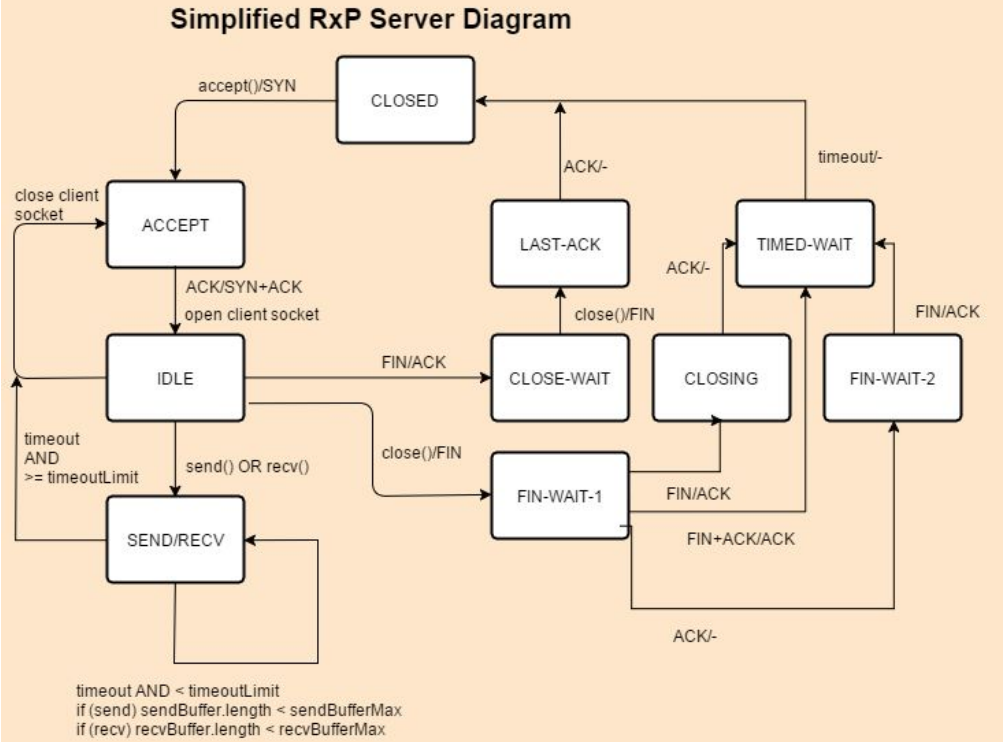
Algorithm Descriptions

md5 checksum

Checksum calculation for detecting corrupted packets. The md5 checksum is computed against all headers and body of the RxP packet.

Finite State Machines





References

md5 (Python). Retrieved November 20, 2015, from https://docs.python.org/2/library/md5.html