Reliable Data Transfer Protocol

Pranav Shenoy
Ian Donn

CS 3521 - Computer Networking
November 4th

Reliable Data Transfer Protocol on UDP

This implementation of Reliable Transfer Protocol (RxP) provides reliable, bi-directional, connection-oriented, window-based, data transfer with byte-stream communication semantics.

Reliability

Each connection is initialized with a three-way handshake. This establishes synchronization and establishes the connection. The implementation of these three steps is as follows:

1. Host 1 sends a synchronization (SYN) packet to Host 2
2. Host 2 sends a SYN-ACK (acknowledgement) packet back to Host 1
3. Host 1 sends an ACK to Host 2

This establishes the connection and data transfer in either direction can begin. Because the data size is limited by a buffer size, we will implement a sliding-window protocol in order to ensure that all data is put in order and duplicate packets are discarded, but an ACK is still resent. Each packet is assigned a sequence number incremented based on bytes. Each packet also requires an ACK to be sent back confirming its reception. We use the Go-Back-N protocol for our timeouts. This is a pipelined protocol in which if the timer resets and only ACK n has been received, the sender goes back to that packet and retransmits the rest.

Corruption of a packet is fixed through the standard internet checksum. Each packet will contain a header which includes a checksum calculated at the transmitting host. The receiving host will also calculate the same checksum and verify with the existing checksum as to whether the packet is corrupted or intact. ACKs are only sent after validating a received packet using checksum. Thus, all corrupted packets will be received only after the timeout retransmission. The receiver is only waiting on the next sequence number which means that all duplicate packets are ignored, and lost or corrupted packets are each re-sent by the sender according to Go-Back-N. In particular, a lost or corrupted packet is automatically retransmitted when a timeout on the sender is reached.

The checksum algorithm is the standard IP checksum. with additions. It takes the one's complement sum of the payload byte per byte, and then adds all 16 bit words in the header and then taking the one's complement of that number.

Simultaneous bi-directional flow is handled by first checking for any available data that has been received, processing it (validating, moving it into buffers, etc.), and then sending out data from the application layer. Along with this data, the RxP protocol will attach the necessary ACK values and set the field bits accordingly to the correct values so that the other end can know it's data has reached the other side properly.

Flow control, while not implemented in our code, would be handled by including in the header a 2 byte window size field. This field is set equal to the remaining buffer size in the receiver, and would be updated on each ACK. The sender would then not send more data than is available in the receiver's buffer.

Byte-stream semantics take the form of maintaining SEQ, length, and ACK numbers in the header. The SEQ field indicates the location in the byte stream at which the data in this packet starts. The length is the number of bytes in the payload, so the packet transmits bytes SEQ through SEQ + length - 1. The receiver then sends back an ACK equal to SEQ + length to represent the next byte it expects from the byte stream. Out of order packets are buffered temporarily until the next packet in the stream (with the correct SEQ expected by the receiver), at which point the receiver pulls as many packets in byte-stream order from the buffer, and ACKs the final SEQ+length of the final packet. The buffer is of a limited size, so older packets are displaced and discarded when new packets arrive if the buffer is full.

When either endpoint decides to close out a connection, it sends an END message to the other endpoint. The other endpoint will acknowledge the closing signal, stop accepting new data into its buffer, send the remaining data to the first endpoint, and then send an END message to the first endpoint. Only once the endpoint has received an END message from the other endpoint will it stop accepting data. This ensures no data is lost due to a premature close.

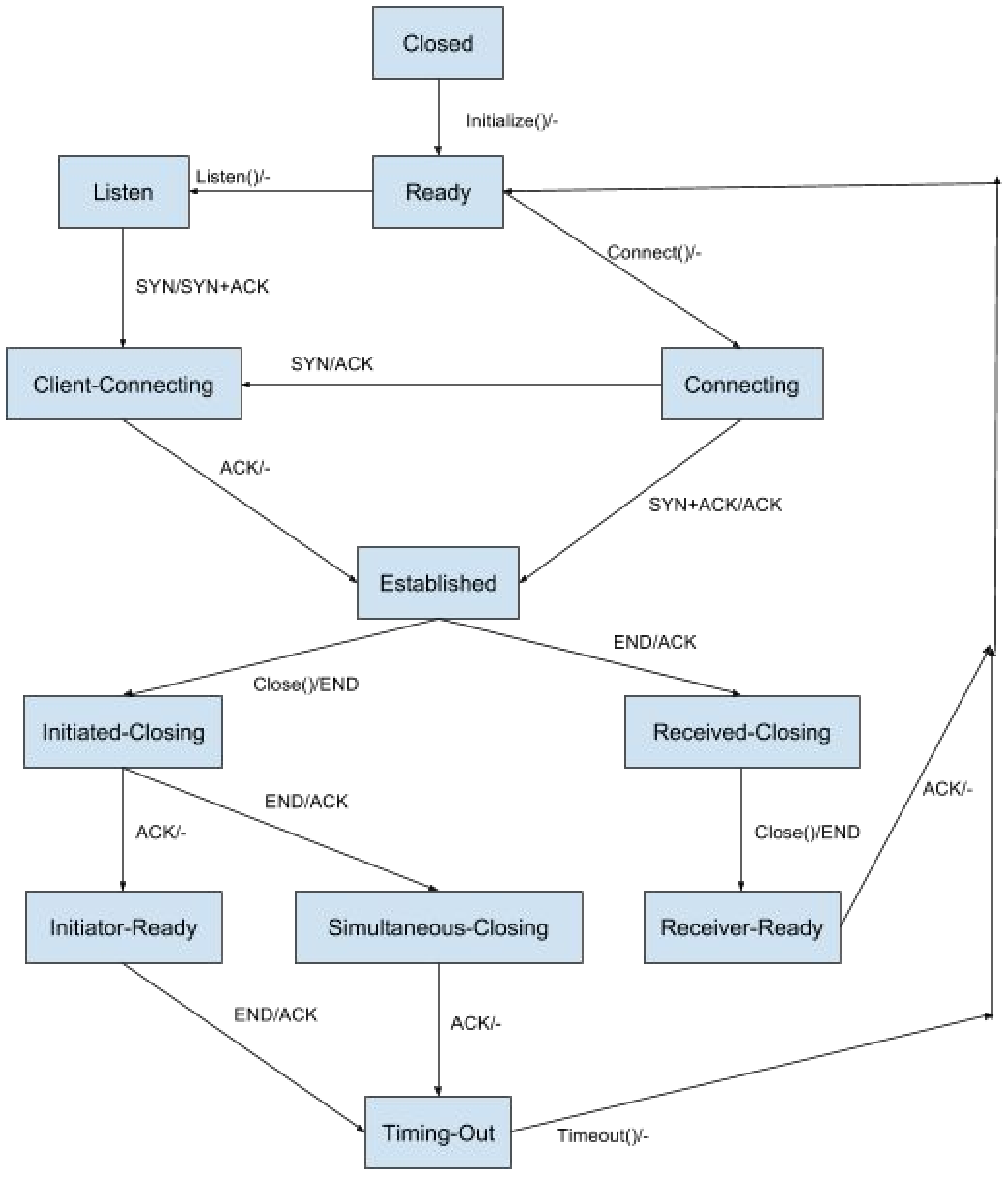
Header Info
2 bytes: source port
2 bytes: destination port
4 bytes: sequence number
4 bytes: acknowledgement number
2 bytes: length
(not implemented, but used for flow control) 2 bytes: window size
2 bytes: Internet checksum
1 byte: bit fields: 00000[SYN][ACK][END]
<length> bytes of data follow after this 17 (19 with flow control) byte header

API

Method	**Return value**	**Description**
Connect(ipaddr, portnum)	boolean	This sends the request to a server to start communication. It is step one of the three-way handshake. Returns whether connection was successful. (Must connect to a server that is already listening to connect reliably)
Listen(portnum)	new connection	This is for the server to listen to requests from clients. A request starts the three-way handshake. Returns a new connection between the server and connected client, and the original connection stays in listen to await more clients
Send(data)	void	This is the command for either server or client to send some amount of data. This command adds the data to a buffer to be transmitted.
Get(length)	data	This gets an amount of data from the buffer. If there is less than the parameter length in the buffer already, it will all be returned. Otherwise the specified amount will be.
Close()	void	This closes the connection. Only completes after both ends have finished sending their queued data.
Initialize(buffsize)	buffsize	Required before Connect and Listen. Initializes buffer and and sets up other required values. Returns buffer size.
SetBuffer(buffsize)	buffsize	If input meets requirements, the input is set as new buffer size. Buffer size that was chosen is returned.

GetBuffer()	buffsize	Returns the buffer size.

RxP State Machine Diagram



Closed
Initialize()/-
Listen
Listen()/-
Ready
Connect()/-
SYN/SYN+ACK
Client-Connecting
SYN/ACK
Connecting
ACK/-
SYN+ACK/ACK
Established
END/ACK
Close()/END
Initiated-Closing
Received-Closing
ACK/-
END/ACK
Close()/END
ACK/-
Initiator-Ready
Simultaneous-Closing
Receiver-Ready
END/ACK
ACK/-
Timing-Out
Timeout()/-