Reliable Transport Protocol (RxP) Design Report (Revised)

Philip Bale
Allen Chen

CS 3251
2015

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

A: The protocol is non-pipelined and uses Stop-and-Wait.

Q: How does your protocol handle lost packets?

A: The protocol handles lost packets by using a client-side retransmit timer.

Q: How does your protocol handle corrupted packets?

A: The protocol uses a checksum to handle corrupted packets. If the pre-transmit and post-transmit checksums do not match, the packet is dropped. Acknowledgements are used to recognize the last in-order packet that was received.

Q: How does your protocol handle duplicate packets?

A: The protocol uses sequence numbers attached to packets to handle duplicate packets. Received packets with duplicate sequence numbers are dropped. Acknowledgements are used to recognize the last in-order packet that was received.

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

A: The protocol uses sequence numbers attached to packets to handle out-of-order packets. Sequence numbers will be sequential and if a packet is received out-of-order it is dropped. Acknowledgements are used to recognize the last in-order packet that was received.

Q: How does your protocol provide bi-directional data transfers?

A: Our protocol does not allow for bi-directional data transfer

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

A: Yes, we use an MD5 checksum (https://en.wikipedia.org/wiki/MD5) to ensure data integrity.

High-level Description

Our RxP is a connection-oriented transfer protocol that improves upon the functionality of TCP; it will in essence hide the issues of a best-effort network (packet losses, reordered packets, and duplicate packets) and will provide the abstraction of a reliable packet stream. Our goal is to improve accuracy, packet size, security, and general management in terms of performance.

Packets are sent in a non-pipelined, stop-and-wait manner. A 3-way connection handshake is used for security and resource management. This will defer denial-of-service attacks that use SYN flooding. There will be no difference between packets sent from node A to node B and node B to node A. Rather than summing the bits in the packet as is done in TCP, our protocol uses a MD5 hash checksum of the packet to improve corruption detection.

A retransmission timer will be used to detect failed packet sending. If the retransmission timer expires before an acknowledgement is received, the packet is promptly retransmitted. Acknowledgements are only sent if received packets are valid and in-order; otherwise we will simply drop erroneous packets--whether they are corrupted or duplicated.

Any node using RxP may maintain multiple communication streams at any given time. In such, either side of the connection--node A or node B-- may terminate a connection at any time. When connected, there is essentially no differentiation between server and client.

Description of RxP header structure and its fields

Source Port (16 bits): Port of sending node. Port is bound to one application and is used for multiplexing packets

Destination Port (16 bits): Port of receiving node. Port is bound to one application and is used for multiplexing packets

Sequence Number (32 bits): If first packet (denoted by SYN flag), then this is marks the first packet in our message. Otherwise, this number denotes the ordering of the packet relative to the first packet.

Acknowledgement Number (32 bits): This number marks the last received packet and (if ASK is set), this value is the next expected sequence number to be received.

Control bits: 5?

- **SYN Flag (1 bit):** Initiates connection establishment
- **ACK Flag (1 bit):** Marks that the acknowledgment number is valid and should be used
- **FIN Flag (1 bit):** Marks that a node wants to terminate the connection
- **LST Flag (1 bit):** Denotes last packet in message
- **RST Flag (1 bit):** Reset connection

Receive Window (16 bits): Size of the buffer the receiving host has made available for this temporary storage of packets

Payload Length (16 bits): Size of the data

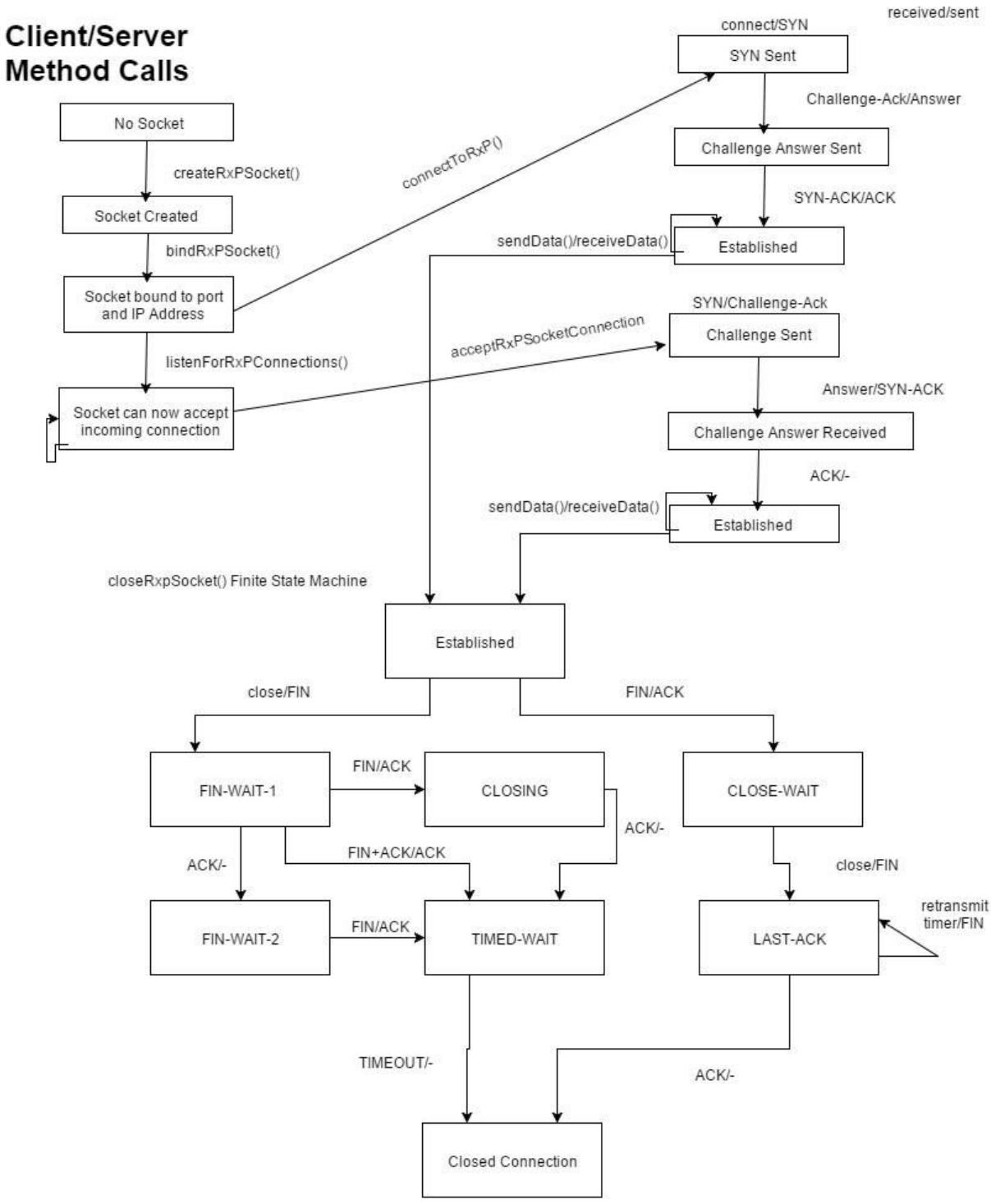
Checksum: 128 bits (MD5)
Payload: Variable length

Header Packet Diagram

Source Port (16 bits)	Destination Port (16 bits)
Sequence Number (32 bits)	
Acknowledgement Number (32 bits)	
Payload Length (16 bits)	Receive Window (16 bits)
SYN | ACK | FIN | LST | RST (5 bits)	Reserved (27 bits)
Checksum (32 bits)	
Payload (variable length)	

Finite State Machine Diagram for RxP Nodes

Client/Server Method Calls



API Description

```
/**
 * Creates and Binds an RxPSocket to an address
 * @param ipAddress          The IP address the socket is bound to
 * @param portNumber         The port number the socket is bound to
 */
RxPSocket createRxPSocket(ipAddress, portNumber))

 *Closes the connection between the two hosts
 *@param rxpSocket          The RxPSocket to close
 */
void closeRxpSocket(rxpSocket)


/** Tells the RxP socket to listen for incoming connections. Also specifies the length of the
*  listen queue.
* @rxpSocket                The socket that will begin listening for connections
* @backlog                  Limits the length of queued connections for this socket
*/
void listenForRxPConnections(rxpSocket, backlog)

/** The specified socket accepts a connection. The socket must be bound to a listening
*address and listening for connections.
*@param rxpSocket           The socket that will accept a connection
*@return RxPSocket          Returns a new socket used to send and receive data on the
*connection. It is bound to the socket on the other end of the connection.
*/
RxPSocket acceptRxPSocketConnection(rxpSocket)

// Initiation sender
/** Connects the passed in rxpSocket to a remote socket specified by the passed in IP address
*and port number.
* @param rxpSocket          The local socket used to connect
*@param ipAddress          The IP Address of the remote socket being connected to
*@param portNumber         The port number specified of the remote socket
void connectToRxP(rxpSocket, ipAddress, portNumber)

/** Sets the buffer window length of a socket to a certain length. The window length limit
*represents how much data the socket can handle from its peer at one time before it is
*passed to the application process.
*@param rxpSocket          The specified socket
*@param windowLength       The new buffer window length for the specified socket
```

```
*/
void setWindowSize(rxpSocket, windowLength)

/** Sends data to the specified socket. The socket must be connected to a remote socket.
*@param rxpSocket        The socket to which the data is sent to
*@param data             The data being sent to the socket
*/
void sendData(rxpSocket, data)

/** Receive data from the socket up to the maximum length specified.
*@param rxpSocket        The socket from which the data is being received from
*@param maxLength        The maximum length of data that will be received
string receiveData(rxpSocket, maxLength)
```

Non-trivial Algorithmic Descriptions

Corrupted Packets

RxP uses MD5 to compute the checksum of our packet and its payload. Each of the fields in the packet are concatenated together in the order that they appear in the header structure. The only part of the packet that is not included in the checksum hashing is the checksum itself. Since MD5 is 128 bits, we take the first 16 and last 16 bits and combine them to form our 32 bit checksum. We recompute this checksum upon receipt of a packet. If the two checksums do not match, then the packet is considered to be corrupted and is dropped.

Lost Packets

We use a transmission timer for *each* packet. If an acknowledgement is not received in that timeframe, the packet is considered to be lost and is retransmitted. SInce the receiver only acknowledges packets that are received in proper order, the retransmission timer works effectively for proper transmission.

Out of Order Packets & Duplicate Packets

If a packet is received but its sequence number does not match the next expected packet, then it is immediately dropped. Similarly, if a sequence number is received more than once, it is immediately dropped.

Window Based Flow Control

In order to prevent the rate of transmission to exceed appropriate bounds, the receiving window-size length that is indicated in our packets is used. The sender must limit itself to this upper bound or else the data will not be transmitted.