CS 3251 Homework 4 Design Report
Team: Francisco Zampieri, Vignesh Prasad
Date: November 4th, 2015

FxV Protocol

The FxV protocol is a protocol that can be used to transfer data reliably without packet loss, data corruption or wrongly ordered packets. The protocol is connection oriented and establishes a connection through a 4-way handshake. The 4-way handshake includes the same parts as the TCP three way handshake and also an authentication system that is used to ensure that server resources are not allocated to malicious users who do not have real IPs. The answers to the questions below can be used to get a high level description of the protocol:

1. Is your protocol non-pipelined (such as Stop-and-Wait) or pipelined (such as Selective Repeat)?
   FxV is a pipelined protocol, meaning it implements a Selective Repeat strategy. Packets sent with sequence number *s* are acknowledged by packets with acknowledgement number *s + 1*. For example received packets with sequence numbers 1, 3, and 4 are acknowledged with acknowledgement numbers 2, 5, and 6. Each host maintains two buffers that store the packets to be sent and the packets that have been received. The buffer is at least two times the size of the sliding window to remove ambiguity caused by modular arithmetic on sequence numbers. The sliding window is moved as and when packets are sent and received.

2. How does your protocol handle lost packets?
   Lost packets are handled by resending them. The sender maintains a logical timer for each packet. If the sender does not receive an ACK before the timer expires the packet is retransmitted. The sender may also perform a fast retransmit if it receives three ACKS for a different sequence number. For example if the packets from 1 through 5 are sent and packet 2 drops; the ACKS received are 2, 4, 5, 6. The three out of order ACKS 4, 5, 6 indicate that the packet was lost and the sender can retransmit the packet without waiting for the timeout.

3. How does your protocol handle corrupted packets?
   The protocol implements a checksum similar to the UDP checksum. If a packet is corrupted it is treated the same way as a lost packet.

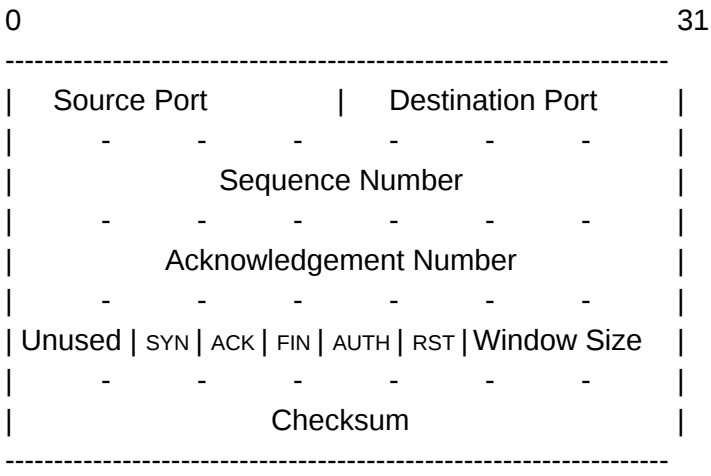
4. How does your protocol handle duplicate packets?
   The receiver re-sends the ACK in the event of a duplicate. This is to ensure that in case the ACK was corrupted or not received then the sender knows that the packet has been received.

5. How does your protocol handle out-of-order packets?
   Assuming the receiving buffer is not full packets that are placed in the buffer by sequence number modulo the buffer size, ensuring that the order of the packets is maintained. If a packet placed in the buffer is at the start of the window, then the window moves and the data is sent to the application. The buffer size is at least twice the window size so that there is no chance of data being replaced in the buffer before it is read by the application.

6. How does your protocol provide bi-directional data transfers?
   Both connected hosts may send and receive data. They both have a send and receive buffer that allow them to do this in a pipelined manner.

7. Does your protocol use any non-trivial checksum algorithm (i.e., anything more sophisticated than the IP checksum)?
   See the last page which describes all non trivial algorithms.

Header Structure and Fields



The RxP header will consist of 20 bytes. We dedicate 2 bytes to the source port number and 2 bytes to the destination port number. Sequence number and acknowledgement number receive 4 bytes each.

The Unused portion consists of 3 unused bits.

SYN, ACK, FIN, AUTH, and RST are all 1-bit flags. A client sends a packet with the SYN flag high when it wishes to connect. The server sends an AUTH+ACK packet back, indicating it wishes for the client to authenticate. The client returns an AUTH+ACK packet of its own, containing the authentication payload and acknowledging the server's request to authenticate. When the client authenticates successfully, the server sends a SYN+ACK to acknowledge the client's authentication and indicate the connection is open. The client sends an ACK packet to acknowledge the open connection.

This leaves 3 bytes for Window Size. Window size specifies the number of bytes the buffer is capable of receiving.

The header contains a 4-byte checksum field. It will be used in the calculation of a UDP-style checksum.

Formal API Description

The API provided below details functions as they would be used in Java.

1. FxVSocket():
   Creates a socket object which can be used to call the other methods.
2. bind(SocketAddress address):
   Binds the FxVSocket created to a port number and IP address which are specified as a parameter using the SocketAddress abstract class.
3. SocketAddress getSocketAddress():
   Gets the SocketAddress that the Socket is bound to. Can be useful for debugging purposes.
4. FxVSocket accept():
   Accepts a connection from another host and returns a FxVSocket object if the authentication is successful. Blocking call that passively waits for a connection to be made.
5. FxVSocket connect(SocketAddress address):
   Initiates a connection to the host specified in the address parameter and returns an FxVSocket if authentication is successful.
6. Byte[] read(int numBytes)
   The read method takes in the number of bytes the application wants and returns a byte array of that size to the application. Note that this could be a blocking call if the number of bytes specified have not been specified.
7. write(data)
   The write method takes in the data with types including boolean, char[], char, double, float, int, long, Object, String. The data is converted to bytes and packetized before being sent to the receiving host. The method will throw an Exception if the sending buffer is full.
8. close()
   Closes this socket.

Non-Trivial Algorithms

Our checksum algorithm is essentially a clone of UDP's checksum algorithm. It differs in that we dedicate 4 bytes to the checksum field rather than 2.

The procedure to calculate a checksum is as follows:

1. Sum together all 4-byte words in the packet, including the header and the payload and excluding the checksum field.
    a. Whenever there is a carryout, add 1 to the result.
2. Take the one's complement of the result (negate it). This is the value of the checksum field.

To verify a checksum value, step 1 is performed on the received packet. Upon obtaining the checksum result, the checksum value of the received packet is XOR'd against the result of the checksum calculation procedure. The result should be all 1's, or -0 in one's complement binary.

Selective Repeat is implemented through the following possible events:

The receiver is sent a sequence of packets in order: each packet is acknowledged with the sequence number of the packet + 1.

The receiver does not receive a packet of sequence number s*, but receives three packets of sequence number greater than s (fast retransmit):* the sender implicitly recognizes that the receiver did not receive packet *s*, and retransmits it.

The receiver is sent a packet with sequence number at the base of its sliding window: the sliding window moves up by 1 and data is sent upwards in the protocol stack to the application layer.

In addition, both hosts maintains two buffers: one for storing packets to be sent, and another for storing packets received. Both hosts maintain the indices of the sliding windows of each buffer.