Project 1 Report
by: Ankur Chaudhry and Chaitanya Kulkarni

Project Overview:

The server and client programs have not been changed since the last lab, since the ultimate goal is to use UDP protocol to transfer a file reliably using modifications to how we perform flow control, data control and so on it the tcpd daemon.

It all starts with first setting up timer program, then both the troll programs, then the tcpd on the client side, and tcpd on the server side. Then we setup the server on the server side, and once all of this is setup and ready, we run the client, which will initiate the transfer.

Within our tcpd we have implemented a sliding window protocol with timeouts, to maintain flow control. We have implemented CRC checks to maintain error control. We make use of the TCP packet header for all communications to and from tcpd for consistency.

We make use of packet types to distinguish different packets from each other, since we make use of only one thread in each of the tcpd (one on client, and one on server side, each program executed separately) we are responsible for the context switching. i.e., different packets are processed differently, and asynchronously, since they all arrive at the same socket.

The following components that are used in our project are described in detail below.

Wrap Around Buffer:

We make use of a circular queue to manage both send buffer and receive buffer. The send buffer is the buffer at the client side responsible for storing incoming packets from the client, and sending them to the troll to be ultimately send to the server. We maintain a buffer of max 64000 bytes. With a sliding window of 20000 bytes.

We went with the circular buffer because it has a very useful property of a circular buffer which is, it does not need to have its elements shuffled around when one is consumed. (If a non-circular buffer were used then it would be necessary to shift all elements when one is consumed.) In other words, the circular buffer is well-suited as a FIFO buffer while a standard, non-circular buffer is well suited as a LIFO buffer.

Circular buffering makes a good implementation strategy for a queue that has fixed maximum size. Should a maximum size be adopted for a queue, then a circular buffer is a completely ideal implementation; all queue operations are constant time. However, expanding a circular buffer requires shifting memory, which is comparatively costly. For arbitrarily expanding queues, a linked list approach may be preferred instead. But since we maintain a fixed size, circular queue is perfect for the job.

We also maintain 2 book keeping circular queues, to validate or invalidate the data. On the client side, if the data is received from the client, it validates the newly entered block. Once an ACK is received for sending that packet to the server, we invalidate the packet. This allows us to keep track of all ACK received. Same happens on the server side, except, validation for a particular packet happens upon receiving that packet from the client, and invalidation for that packet happens on sending that packet to the ftps program.

Timer:

Timer program that maintains the list of packets and a list of timeout values is implemented using a linked list. It

is present as a separate program and is also executed separately. It communicates to tcpd on the client side through TCP sockets. It makes use of select function to perform asynchronously. Tcpd can send two types of packets to the timer program. One of them is to add a new packet and timeout value, and another is to cancel the timeout for a packet. Timer has one type of outgoing message. On timeout, it sends a command to the tcpd that invokes it that a timeout occurred, on one of the packets. Within the same command it also sends the packet seq number that timed out.

To start, one would provide a handshake with the timer program using the tcpd. Once that handshake is established with the timer program, we can now start sending it start timer calls and cancel timer calls as per the sliding window protocol.

We will receive timeouts from the timer (asynchronously), which is processed correctly by tcpd by resending the timed-out packet, along with resending the start timer for that packet back to the timer.

CRC:

Anytime a message is transferred over a physical medium, the possibility exists that it may be corrupted by noise. Accordingly, since the earliest days of data communication, various mechanisms have been devised to detect when the data received was not the same as the data sent.

An efficient method of detecting errors is the checksum. A checksum is calculated by adding together the values of all of the data bytes in the message. Checksums can be 8, 16, or 32 bits wide (overflow from the addition is ignored). In a typical application, the checksum is appended to the end of the message. The receiver verifies the message by re-calculating the checksum on the data and comparing its result to the checksum that was sent.)

we make use of CRC to perform packet error checks. Each packet, before being sent by tcpd to the server, is processed to produce a crc value which is added in the TCP header of the packet. This same checksum is then used on the server side to check if the packet's data is intact, without any errors. Once this is verified, the packet is then further processed. Otherwise its discarded, and no ACK is sent.

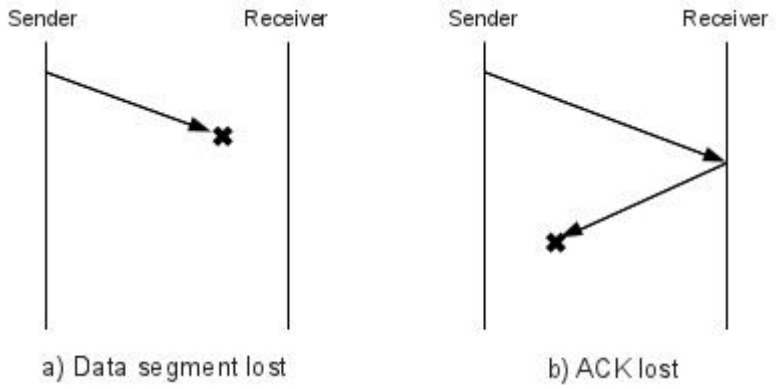
We make use of CRC – 16 to calculate our crc checksum. Each byte is processed. The idea behind CRC calculation is to look at the data as one large binary number. This number is divided by a certain value and the remainder of the calculation is called the CRC. Dividing in the CRC calculation at first looks to cost a lot of computing power, but it can be performed very quickly.

The processor overhead involved in calculating the checksum is not too bad when you consider the large number of errors that the algorithm can detect. The 16-bit CRC is itself sent least significant byte first.

The initial value of the CRC, known as the "preset," can be either 0 or 0xFFFF. Originally, implementers used a preset of zero. This preset, however, exposed a weakness in the algorithm. A message that started with an arbitrary number of zeros would have a CRC of zero until a 1 bit was detected. Today, the predominant preset is OxFFFF, which avoids the leading zero problem.

RTT/RTO:

RTO, or Retransmission *Timeout*, determines how long TCP waits for acknowledgment (ACK) of transmitted segment. If the acknowledgment isn't received within this time it is deemed lost. Actually, ACK could be lost too, but there is no way for sender to differentiate between those two cases, as illustrated by the following figure:



The important part of calculating RTO is to determine how long it takes for a segment to go to the receiver and for ACK to come back from receiver to sender. This is a *Round Trip Time*, or RTT. In some ideal world (and very static for that matter) this value would be constant and would never change. And RTO would be easy to determine, it is equal to RTT, maybe slightly slightly larger, but nevertheless the two would be almost equal.

But we are not living in an ideal word and this process is complicated by the fact that network conditions constantly change. Not only that, but receiver has also certain freedom to chose when it will return ACK.

In order to achieve that, two new variables are introduced, *smoothed RTT*, or short SRTT, and *RTT variance*, or RTTVAR. Those two variables are updated, whenever we have a new RTT measurement, like this (taken from the RFC6298):

RTTVAR <- (1 - beta) * RTTVAR + beta * |SRTT - R'|
SRTT <- (1 - alpha) * SRTT + alpha * R'

alpha and *beta* are parameters that determine how fast we forget the past. If this parameter is too small new measurements will have little influence on our current understanding of expected RTT and we will slowly react to changes. If, on the other hand, *alpha* approaches 1 then the past will not influence our current estimation of RTT and it might happen that a single RTT was huge for whatever reason and that suddenly we have wrong estimation. Not only that, but we could have erratic behavior of SRTT. So, *alpha* and *beta* parameters have to be carefully selected. The values recommended by RFC are *alpha*=1/8 and *beta*=1/4.

Finally, RTO is calculated like this:

RTO <- SRTT + max (G, K*RTTVAR)

Still, there is a question of initial values, i.e. what to do when first SYN segment is sent? In that case RFC specifies you have to set RTO to 1 second, which is actually lower than specified in the previous RFC that mandated minimum value of 3 seconds. When first acknowledgment returns its RTT value is stored into SRTT and variance is set to RTT/2. Then, RTO is calculated as usual.

Packet Formats:

We make use of TCP Header at all times. The following structure is used as header for all packets that that communicated in this program.

typedef struct TCP

{

```
  uint16_t src_port;
  uint16_t dst_port;
  uint32_t seq;
  uint32_t ack;
  uint8_t  data_offset;  // 4 bits
  uint8_t  flags;
  uint16_t window_size;
  uint16_t checksum;
  uint16_t urgent_p;
}tcp;


//Structure used by all
typedef struct Packet{
  struct sockaddr_in header;
  char packetType;
  tcp tcpHeader;
  char body[MSS];
} Packet;
```

We make use of the packetType to indicate what type of packet we are trying to process in tcpd, since the tcpd code for the client side and the server side is the same. This allows for use to perform the context switching between the various types of packets that arrive at the tcpd, since all packets go through the same socket.

Various Implementations of RECV, SEND, CONNECT, BIND etc:

The protocol uses that same recv and send functions as used for tcp protocols, since the idea of this project was to emulate a tcp connection.

The SEND function establishes a connection with the tcpd once a handshake has taken place. It only sends if it receives a response from the tcpd to send more. This allows the tcpd to maintain a floodGate concept. Tcpd controls the flow of data that comes from the client, making it so, such that the tcpd on the client side is not congested.

The RECV function is primarily used by the server. The RECV function, first establishes a handshake with the tcpd on the server side, to let it know its waiting for packets to be received. Once, that is done, any packets sent to the server go through this function. This function, just as the rest of these functions, have been implemented in such a way so as to emulate the tcp protocol. The interface for these functions are the same as that of the tcp protocol.

The CONNECT function is used to establish a handshake on the client side.

How to transfer a file:

1. The whole project is divided into 4 parts:
    a. Client (ftpc) - reads the file sends to TCPD (Client)
    b. TCPD (Client) - receives data from ftpc and forwards to troll
    c. Troll - receives data from TCPD (Client) or TCPD (Server) and forwards it to TCPD (Server) or TCPD (Client) respectively. Can also modify, delay or drop the packet
    d. TCPD (Server) - receives data from troll and sends to ftps.
    e. Server (ftps) - receives data from TCPD (Server) and stores file to disk
2. Client part and troll are in the directory Client and Server part and troll is in the directory Server. (For successful execution make sure Client part is on 'sl1' machine)
3. There is a Makefile in the default directory which compiles all the file required at once.
4. To compile all the files go to directory "Project" and type "make".
5. Steps to send a file from Client to Server:
    a. Start the TCPD (Server) by going to Server directory and starting "tcpd <local-port> <troll-port>". This will start the TCPD (server) at specified port number and also provide the port of troll you will be running at.
    b. Start Troll by starting "troll <port-number". This will start troll at the port number specified.
    c. Start the Server (ftps) by starting "ftps <local-port> <tcpd-port>", where local-port is the port at which server runs and <tcpd-port> is the port where TCPD (Server) is running. This will register the server with TCPD and TCPD will now forward all packets to the server.
    d. Go to Client directory and start Troll by starting "troll <port-number". This will start troll at the port number specified.
    e. Start the TCPD (Client) by starting "tcpd <local-port> <troll-port> <timer-port>", where local-port is the port at which TCPD runs and <troll-port> is the port where Troll is running and <timer-port> is the port where timer is running.
    f. Now, start the timer by starting "timer <local-port>", where local-port is the port where timer is running.
    g. Next, start the file transfer by starting the Client by starting "ftpc <remote-IP> <remote-port> <local-file> <tcpd-port>", where remote-IP is the IP of Server and remote-port is the port number of TCPD (Server) and local-file is the file you want to send and tcpd-port is the port of TCPD (Client).
6. Server will then accept the connection and receive the file and then exit after saving it in the Server directory.
7. You can confirm the file is transferred successfully by checking the "diff" of local file and file received inside the server directory.
8. For sending the next file, just kill TCPDs and do steps (a), (b) and (e) again.

Possible extensions to the program:

One way to improve performance of this program will be to make use of threads to handle every packet that arrives at the tcpd. This will allow for parallel processing of the packets. We make use of multiple sockets, each arriving at a different thread, and each thread processing the packet with arrives at their station, and nothing else.

This also takes care of some the errors we found due to garbling of packets ending up in the wrong processing pipeline. Like a packet sent from the client side is known to be of packetType 3. But when this packet gets garbled, there is a small chance the packet type gets garbled too. This packet type could be any of the other packet types, which means in our original code this packet will get processed as a completely different (and unknown) packet. Creating undesired side effects. (We ended up handling each bad packet separately anyway, to prevent such errors, but going for a Multi-threaded approach would have been much more cleaner and safer).

Multi-threading the tcpd will fix these problems.

Work distribution:

Ankur worked on the timer program, maintaining a linked list of packets along with their time out values including the 2 input communications for starting timers and canceling timers, and 1 output message for timeouts. He was also responsible for the RTO and RTT calculations, The SEND, RECV functions that are used by both client and server, and finally linking all of this together with the tcpd.

Chaitanya worked on the CRC checksum, test as well as generation functions. Also was responsible for the Wrap around buffer and book keeping circular buffers, context switching of various types of packets that need to be processed in tcpd and finally putting it all together, ensuring that all components work to emulate the tcp protocol in tcpd.