|  |
| --- |
| Department of Computer Science and Engineering |
| 678 Final Project Report |
| Ankush Srivastava & Rachit Sood |
|  |
| Prof : Mr. Naeem Shareef |

Monday, March 26, 2012

[Overview: 2](#_Toc287642227)

[Components Implemented: 2](#_Toc287642228)

[Wrap-around Buffer: 2](#_Toc287642229)

[Timer Process: 3](#_Toc287642230)

[FTPS AND FTPC: 4](#_Toc287642231)

[Checksum calculation: 4](#_Toc287642232)

[RTT and RTO Computation: 4](#_Toc287642233)

[Packet Formats: 5](#_Toc287642234)

[Function Implementations: 6](#_Toc287642235)

[Connection Shutdown: 7](#_Toc287642236)

[How to compile and run the program: 7](#_Toc287642237)

[Possible Future Enhancements: 8](#_Toc287642238)

## Overview:

The objective of this project is to develop a transport layer protocol using a simple file transfer operation. This is used to demonstrate features similar to those of TCP using the unreliable service provided by UDP.

In other words we can say that a reliable UDP service has been emulated by doing this project.

The project simulates an actual network environment where events like packet losses, garbling, reordering, duplicates, delays etc. come into picture. As UDP service is unreliable service so we need to implement mechanisms to handle these situations of packet loss. The troll utility is used to control and simulate lossy links and packet garbling leading to lost packets requiring retransmission.

The TCP functionality has been implemented in the TCP Daemon (TCPD) processes at the client and server sides. The FTP client and server side function calls communicate with the local TCP Daemon which is present on server and client side both. The communication between the application process and the local TCPD process can be implemented with any inter-process communication mechanism such as UDP sockets. UDP communication within a machine can be assumed to be reliable. Any outside host call for instance from client host to server host which are different machines assumed to be manipulated using TCP.

## Components Implemented:

#### Wrap-around Buffer:

The wraparound buffer at each end handles the packets which are to be sent (at the client side) and those packets which have just come in (at the server side). It uses the sliding window protocol with selective repeat. A buffer size of 64 KB is used at both the sending and receiving ends and the MSS is 1000 bytes.

At the client side, when the Daemon receives a packet from the client application, it checks either if the sliding window head is less than the sum of tail and window size or if the sum of the head, tail and window size subtracted from 64(max size before wrapping around) is lesser than the size of the window. If either of these conditions is satisfied, the program enters a loop where packet is kept at the buffer and copied to the send buffer [head].MSS (in other words, the packet is placed at the head of the buffer if the window has not yet reached its capacity). The Ack value of the sendbuffer[head] is set to 0 to indicate that the MSS has not been acknowledged yet. The CRC (Cyclic Redundancy Check) is computed using a 16 bit polynomial and is stored in the header before sending the packet.

Before another packet can be sent, the head variable is incremented (closing the window on the left end). Head modulo 64 is computed to ensure that the buffer wraps around (head is set to 0 after reaching the max value of 64).

The sender then waits on select for Acks or for a timeout to occur. When an Ack is received , the file descriptor is set to indicate that the receiver has acknowledged the packet. The CRC value at the header of the Ack is compared with a newly computed CRC. Non-matching CRC values imply that the packet is garbled.

The sender then matches the Ack number to the sequence number of the sent packets stored in the current window of the send buffer. The variable ‘crossed’ is used to determine if the sliding window has passed a particular node in the buffer. It is set to 1 if that is the case.

If the Ack =1 and the node has been ‘crossed’ it means that the packet has been received at the receiver’s end. If Ack = 0 then the packet has not received an acknowledgement yet. If ack=1 and the window has not ‘crossed’ over yet, then it indicates that there is no hole in the buffer. So the tail is moved right by 1 and the window ‘slides’ over.

The above mentioned actions are carried out when either the tail is <=head (in which case the program loops from tail to head, the remainder of the buffer) or if tail > head (in which case we loop from tail to 64 and then 0 to head, wrapping around.)

At the receiver’s end, the packet which is received from the client is kept in the receiver’s buffer. The CRC is computed using the same 16 bit polynomial and compared with the CRC in the header of the received packet. Non matching values indicate packet corruption. Holes in the buffer are indicated by setting the variable breakdowns to 1.

The sequence number of the packet is extracted and it’s modulo 64 value is stored in a variable. The variable in window checks if the packet received is in the current window, if so, an Ack is sent to troll.

The received packet is stored in the buffer and sent to the server application (ftps). The variable recvbase is incremented by 20.

The buffer management system works in tandem with the timer process to handle events of retransmission due to packet loss.

#### Timer Process:

The timer process works with the buffer to handle retransmission. The timers are implemented using a delta-list. The delta-list is maintained in a separate process called the timer process. When a new timer starts, a message is sent to local timer process which contains the duration, the port number and the byte sequence number of the packet. The timer will notify TCPD when a timer has expired and TCPD will notify the timer whenever it receives an ACK or sends a packet so that the delta list can be updated.

The "time" field in the delta timer contains the time difference from the previous event (node). For instance, if three timers need to be started expiring at 6, 15 and 21 seconds from now, the head of the list will have the values 6, 9 and 6. In the implementation, the timeout of each subsequent packet can be determined by adding the values of all the timers in its path. The header contains the value of the first timeout, and the rest can be determined by just following through.

The timer sleeps for a time period equal to the time of the first node. It wakes up whenever it receives a request and places it in the appropriate position in the linked list.

At the client end, the timer starts at the local process when a packet is sent. The client waits on select until an Ack is received or a timeout occurs. When an acknowledgement is received, the entry for that particular Ack in the buffer is deleted. The timer is run on an infinite loop.

At the server end, the select function waits on the timeout variable. If a packet is lost and a timeout occurs, a message is sent to the client with the same sequence number as the acknowledgment that it expects to receive. The client matches this with the sequence number stored in its buffer window and retransmits the missing packet(s).

#### FTPS AND FTPC:

At the client side application, when the ftpc generates the first packet, the file size is put into the first 4 bytes of the packet. The next 20 bytes store the filename and the rest of the packet contains data which needs to be sent. Subsequent packets contain only data. The FIN packets do not contain any data, they just have the FIN bit set.

At the server side of the application the ftps reads the first 4 bytes of the first packet to represent the file size, the next 20 bytes represent the filename and the rest, data. Subsequent packets are taken to contain data alone. When ACKs are sent, the packets do not contain any data, they only have the ACK bit set.

#### Checksum calculation:

The CRC (Cyclic Redundancy Check) check sum technique is used for computing the checksum. The 16-bit CRC-16-CCITT polynomial we are going to use is x16+x12+x5+1.

The Client TCPD performs the computations and appends the bits. The Server TCPD performs modulo-2 of the received message. If there is a non-zero remainder, an error has occurred and the packet will be dropped.

#### RTT and RTO Computation:

The Round Trip Time (RTT) is the time elapsed since the sending of a packet till the receipt of ACK for that packet. RTT is computed for every packet and Retransmission Time Out (RTO) is computed for the next packet to be sent. These are central to computing the timeout in case of a lost or delayed packet.

The RTT was calculated using the system time. The start\_time stores the system time when the packet was first sent and end\_time has the system time when an Ack is received. The difference between the end\_time and the start\_time gives the RTT.

RTO can be computed from RTT by using the Jacobson’s algorithm. The mean deviation is a good approximation to the standard deviation, but easier to compute. This leads to the following equations that are applied to each RTT measurement M.

Err = M - A   
A <- A + gErr   
D <- D+ h(|Err| - D)   
RTO = A + 4D

A is the smoothed RTT (an estimator of the average) and D is the smoothed mean deviation. Err is the difference between the measured value just obtained and the current RTT estimator. Both A and D are used to calculate the next retransmission timeout (RTO). The gain g is for the average and is set to 1/8 (0.125). The gain for the deviation is h and is set to 0.25. The larger gain for the deviation makes the RTO go up faster when the RTT changes.

## Packet Formats:

The TCP header at the server is defined as follows:

typedef struct tcp\_header

{

int sequence\_num; // Sequence number

int ack; //Acknowledgement number

int ackseq; //Acknowledgement sequence number

int bytes\_in\_packet; //Number of bytes in a pcaket

int fin; //Finishing number

int finack;//Acknowledgement number of FIN

int stop; //Stop value

char data[900]; //Actual data

} tcp\_header;

The TCP header at the client is defined as follows:

typedef struct tcp\_header

{

int sequence\_num; // Sequence number

int ack; //Acknowledgement number

int ackseq; //Acknowledgement sequence number

int bytes\_in\_packet; //Number of bytes in a pcaket

int fin; //Finishing number

int finack;//Acknowledgement number of FIN

int stop; //Stop value

char data[900]; //Actual data

unsigned long checksum;//Checksum of the data

struct sockaddr\_in msg\_header;//Message header

} tcp\_header;

## Function Implementations:

* SOCKET() : Creates a new socket.

**Parameters:** family (AF\_INET), type (datagram or stream. SOCK\_STREAM is used since TCP is being used) and protocol.

int socket(int family,int type, int protocol)

* BIND(): Binds a port to a socket. It calls the bind() function.

**Parameters:** instance of the socket, the address of the port and length of the address.

int BIND(int sockfd, struct  sockaddr \*myaddr,int addrlen)

{

int b;

b=bind(sockfd,(struct sockaddr\*)&myaddr, addrlen);

return b;

}

* LISTEN() : Used by the server to listen to requests from the client.

**Parameters:** instance of the socket, backlog

* ACCEPT():

Used by the server to accept client request. This is not used here as UDP is being used.

**Parameters:** instance of socket, address of peer and length of the address.

* CONNECT():

Establishes a connection with the server. This is not used here as UDP is used for communication.

**Parameters:** instance of socket, server address, length of address.

* SEND() and RECV() are respectively used to send and receive data.

**Parameters:** instance of socket, message, length of message, flags

void  SEND(int sockfd, const void \*message, int len, int flags)

Recv parameters: instance of socket, message, length of message, flags

void  RECV(int sockfd, void \*message, int len, unsigned  int flags)

* CLOSE():

Closes the socket. Calls the close() function.

**Parameters:** instance of the socket.

void CLOSE(sockfd)

{

int sock=sockfd;

close(sock);

}

## Connection Shutdown:

The connection termination is initiated at the sender:

* Sender sends a packet with the FIN bit set. It also sends the RTT value in the packet. The sender sends a packet without data upon receiving the last acknowledgment packet.
* The receiver on receipt of FIN packet, sends an acknowledgment back to the sender.
* When it sends the ACK packet, it goes into a wait state, with a timer set to RTT. The only packet the receiver can now receive is a retransmitted FIN packet. It will send another acknowledgment if it receives FIN packet and resets the timer.
* If the receiver does not receive any packet before the timer expires, then it closes the connection and releases all the buffers and memory.
* The sender on receiving the acknowledgment will release all the memory.

## How to compile and run the program:

Server side is compiled and executed first. The makefile helps avoid keying in all flags while compiling. Seven terminals are used to execute the project.

7 steps are given below in order to run the whole program.

Initially open 7 terminals.

4 on the client side.

3 on the server side.

**Step 1:** make – creates the makefile in all terminals. This is done only while executing for the first time. Subsequent executions do not need a make unless changes have been made to the code.

**CLIENT SIDE**

**Step 2:** Run  ./troll -t 11000 on the client side on one terminal

**Step 3:** Run ./tcpdc <destination host> on the client side

**Step 4:** Run ./timer on the client side

**Step 5:** Run ./ftpc <file name to send>

**SERVER SIDE**

**Step 6:** Run ./troll 25000 on the server side

**Step 7:** Run ./tcpds <client host> on the server side

**Step 8:** Run ./ftps <file name to save by>

## Possible Future Enhancements:

The project could be further extended by giving the server the capability to send messages in addition to receiving messages. In other words, the project could be extended to support two-way traffic.

in the current implementation, By using a spoofed IP address and repeatedly sending purposely assembled SYN packets attackers can cause the server to consume large amounts of resources keeping track of the bogus connections. This is known as a SYN flood attack. SYN Cookies are the key element of a technique used to guard against SYN flood attacks. Daniel J. Bernstein, the technique's primary inventor, defines SYN Cookies as "particular choices of initial TCP sequence numbers by TCP servers." In particular, the use of SYN Cookies allows a server to avoid dropping connections when the SYN queue fills up. Instead, the server behaves as if the SYN queue had been enlarged. The server sends back the appropriate SYN+ACK response to the client but discards the SYN queue entry. If the server then receives a subsequent ACK response from the client, the server is able to reconstruct the SYN queue entry using information encoded in the TCP sequence number.

Other improvements could include implementing the push operation provided by TCP, implementing 3 way handshake and half-open connections and dynamically changing the window size based on network traffic.