Project Report
(CSE 5462)

Gopi Krishna Tummala and Rupam Kundu
Computer Science and Engineering Department
Graduate Student
The Ohio State University

April 24, 2015

Contents

1 Project Overview

Problem Statement: In this Project we implemented and tested TCP-like reliable transport layer protocol by using any inter-process communication mechanism such as UDP. Additionally, we also implemented test programs on top of this transport layer, to check the functionality of individual modules. Then the functionality of this transport layer protocol is contrasted against TCP, by designing simple file-transfer application. File transfers are monitored, tested using *troll*, a network emulator, which emulates real world network by using user adjustable parameters such as packet garbling, packet drop etc.

Objective: The major focus of the project implementation is file-transfer application. This file-transfer application and its integration with the transport layer code is shown in the Figure 1. From the figure the top to *ftps* and *ftpc* are file-transfer server and file transfer client respectively. These applications run on two machines (beta.cse.ohio-state.edu and gamma.cse.ohio-state.edu in our case.) and referred as M2 and M1 respectively, in the following discussions. *ftps* and *ftpc* communicate with the M1 and M2 respectively using *TCPD* program on both the machines. These communications run on UDP sockets. We observed that UDP sockets inside machine are reliable and so we assumed no-packet loss in our further design calibrations. Also TCPD is supposed to communicated with local TCPD process of operating system. This functionality is show in the figure. Additionally these TCPD processes communicate with *troll* application installed on the M1 and M2 and TCPD on M2 communicates with *timer process* installed on M2.

However, the packet loss in the system is introduced by the communication between communication between *TCPD-M1* and *troll-M2* & *TCPD-M2* and *troll-M1*. Here comes the functionality of troll in emulating packet garbling and dropping. In accordance with packet drops and garbled packets the time process monitors the "ACK" messages and trigger *TCPD-M1* to retransmit the packet incase of a packet drop. Most of the TCP functionalities like handshake and retransmission are implemented inTCDP process on M1.

ftpc client transfers the files using this TCP implementation. First the size of the file is send using 4-bytes allocated for size, then the name of file is send using next 20 bytes and subsequently the data of the file is send. The real network scenarios are emulated using troll. Note all these communications happen using UDP and assumed to be reliable as these communication links are inside the machine. This section briefly outlined the functionality of individual blocks, however implementation details and design parameters are discussed clearly in the subsequent sections.

2 Main Components of Design: Explanation and Functionality

2.1 Client process

The client process mainly transfers a file (by breaking down into multiple packets of 1000 bytes each) to TCPD-M2 process via UDP sockets. The process initiation involves function calls like SOCKET(), BIND()and CONNECT()(discussed in more detail in Section) for establishing the connection. After the connection has been established, bytes from the file are read sequentially followed by calling the SEND() function to transfer the data to FTPS. SEND() is a blocking call and does not return until all bytes are written to the TCPD buffer.

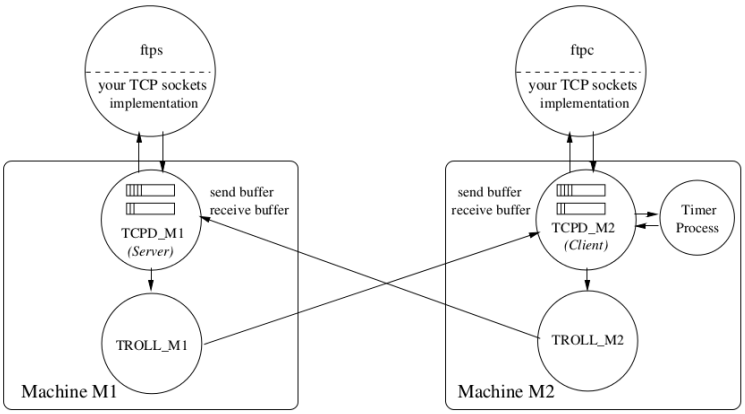


Figure 1: *Architecture of the system*

2.2 Client Daemon

TCPD in client's side maintains(TCPD_M2) and manages a wrap around Circular Buffer for receiving messages from *ftpc*. TCPD calculates the checksum(CRC) of each packets. TCPD also keeps track of the RTT and RTO values and updates them when necessary. Moreover the acknowledgements for the packets are also tracked by TCPD and takes decision for retransmit by communicating with the TimerProcess. Finally TCPD close the connections with the TIMER and ftpc process.

2.3 Server Daemon

The packets sent by Client Daemon(TCPD_M2) via TROLL is received by the TCPD daemon on the server side(TCPD_M1) and stored in a buffer. The checksum(CRC) for each packet is verified here for detecting packet drop and packet garbling and accordingly the acknowledgements are sent back on successful reception. After receiving all the packets those are transferred to *ftps*. Finally TCPD_M1 closes the connection with TROLL and *ftps*.

2.4 Server process

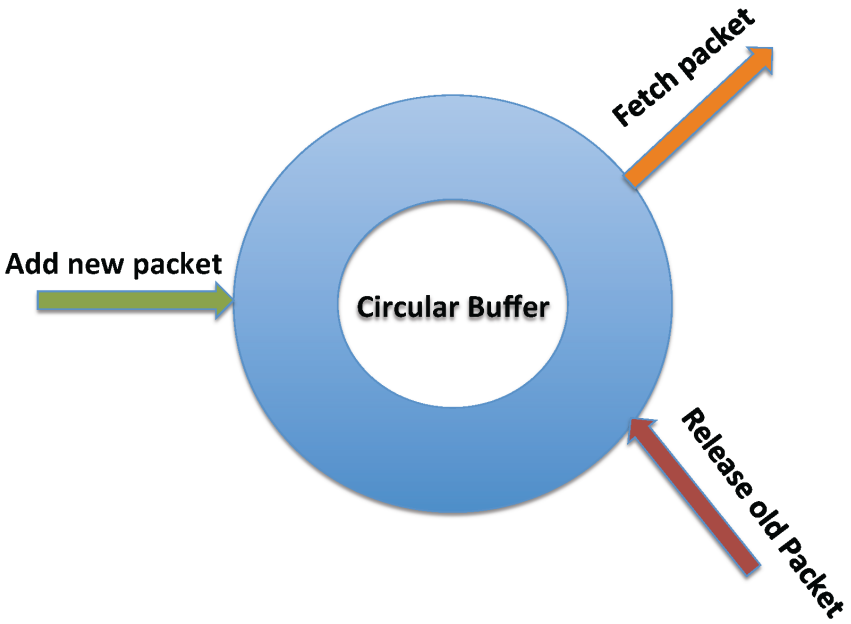
The server side receives the packets transferred from TCPD_M1 and store it in a folder/directory (different from the original one on client's side) by calling functions SOCKET(), BIND() and ACCEPT(). It usually blocks until a connection is made to it from the client.

2.5 TROLL

Troll is the program that is used in our implementation to emulate the real-world network environment. Troll can provide the emulated functionalities like lossy-links, packet garbling, delay and packet duplication etc. This emulation helps us to test and debug our application in different environmental conditions. Troll communicates with our implementation of TCDP directly. All packets will go through a TROLL process running on the same machine and undergo delay, duplication, garbling and/or drops before forwarding to desired address.

2.6 Circular Buffer

As stated in the project statement we implemented using circular buffers which helps to track the packets delivered to server. Circular buffer will have one pointer pointing to location where the data can be added that gives the start of the buffer. There will be another pointer pointing toward the end of the buffer location in memory. There will be another pointer to insert data into the buffer which moves along the with the data inserted into the buffer. Additionally a variable is maintained to keep track of the count of the values inserted into the buffer.



2.7 Timer Process

A timer is attached to each sent packet so that if timer expires, the packet can be re-transmitted in case the packet is not delivered. The delta list mentioned in the project web-page can be utilized to estimate when each timer expires. The process requires only one timer to track one process at a time while the timers for rest of the processes are tuned relative to the first one. Hence it is not computationally hard and saves lot of processing cycles. It is important to mention here that each process keeps track of the time for its respective packet.

The implementation comprised of a list where each entry in the list will have a delay, processID, next process entries. The processes are ordered in terms of increase in delay time. Following the delta list for timer implementation, Nth process's delay is relative to N-1th process's delay. Suppose process X and Y have timers of 5 and 9 respectively, the delay on A is 5. The delay on B is 4(9 minus 5). The queue_id will point to the 1st item in the list. After an item has been awakened, the process examines if it has received the ACK for that packet or not. If it has, the process is evicted from queue else a new process for that packet is appended at the end of the queue.

Implementation: The Timer is implemented using a doubly-linked list which will interact with TCPD_M2 using the following packet format:

Variable	Integer	Initial Value	Comments
sequence#	Integer	0	Contains the sequence number of a particular packet
time	Integer	0	Contains the RTT value of the packet
flag	Integer	0	Indicates one out of 2 operations: Insert, Delete

2.8 Jacobsons Algorithm and RTT & RTO

RTT is the acronym for Round Trip Time, its the time to reach the destination and come back to the source. Timer will be started when the data is sent for the destination and timer gets stopped when ACK of the data is received. In case ACK for data is not received before time-out period the time is reset to current time-stamp and the packet is retransmitted.

RTO is the acronym for Recover Time Objective. It quantifies the maximum tolerable time period that a system can be down after failure. Based on the RTO, the system can determine the necessary steps that are needed to ensure proper recovery.

Implementation: Jacobsons Algorithm We implemented Jacobsons algorithm to calibrated RTT/RTO as specified in the project statement. This implementation has following steps:

- Step1: Compute *srtt* (Smooth RTT) by using equations:

$$Err = M - srtt_{old}, \qquad M \; is \; current \; RTT$$

$$srtt = srtt_{old} + g * Err, \qquad g \; is \; the \; gain \; for \; the \; average \; and \; is \; set \; to \; 0.125$$

- Step:2 Compute D (Smoothed Deviation) by using the equation:

$$D = \alpha D_{old} + h * (|Err| - D_{old}), \qquad h \; is \; set \; to \; 0.25$$

- Step3: Compute RTO by using the equation:

$$RTO = SRTT + 4D$$

By the end of this algorithm RTT is calculated from step-1 and RTO is obtained from step-3. Firstly the time between sending a packet and receiving its acknowledgement is used to compute RTT. Upon receiving the first RTT sample, the variables are initialized as follows:

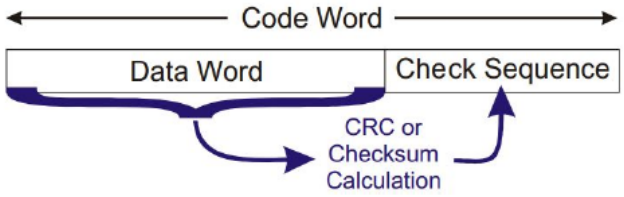
$$srtt \longleftarrow M$$

$$D \longleftarrow M/2$$

After the computation of RTT and RTO these values are used as a method of congestion avoidance.

2.9 Checksumming Algorithm:

Cyclic Redundancy Check (CRC) is employed to ensure successful transmission by detecting accidental errors in raw data transmission. The main idea is to append a certain number of check bits, usually referred to as the checksum, to the message being transmitted. The combined data and the checksum is known as the code word. Figure below represents a code word. This assists receiver to estimate errors during transmission. A negative acknowledgement is generally delivered in case of an error.



Checksum is created by a binary division with the generator polynomial as the divisor and the data word as a dividend. The remainder thus generated acts as a check sequence that is appended to the data word. Here we consider the most significant bit first representation. The sender and receiver agree on a generator polynomial. A 16-bit long CRC is used and the generator polynomial will be x1021(in hexadecimal). The most significant bit is always ignored by the CRC standards because it is always one. Two approaches can be used to check that the file received contains no errors. The first approach is to repeat the process of the sender and verify the check sequence. The other approach(which we will follow) is dividing the whole code word by the generator polynomial and check whether the remainder is zero.

Implementation: The checksumming algorithm is implemented using the standard CRC-16 implementation with following parameters:

- Width= 16bits
- Polynomial=0x8005
- Initial Remainder=0x0000
- Final XOR Value=0x0000
- Reflect data?=yes
- Reflect Remainder?=Yes
- Check value=0xBB3D

2.10 Packet Formats:

According to the Figure 1, there are 3 different communications:

- between *ftpc/ftps* and *tcpd(M2/M1)* daemon respectively
- between *tcpd(M2)* and *tcpd(M1)*
- between *timer process* and *tcpd(M2)*

Implementation Details:

- **Packet format for communications between *ftpc* and TCPD_M2, TCPD_M2 and TCPD_M1, TCPD_M1 and *ftps*:**

  typedef struct timer_packet
  float time ; //Associated Time
  int sequence_number ; //Sequence Number of Packet
  int type ; // 0 - Insert; 1 - Delete
  timer_packet;

- **Packet format for communications between TCPD_M2 and Timer:**

  typedef struct data_packet
  char payload[1000];
  int sequence_number;
  int size;
  char FYN;
  crc checksum;
  data_packet;

2.11 Function Calls to be implemented

The functions: SOCKET(), BIND(), ACCEPT(), CONNECT(), SEND(), RECV(), and CLOSE() are used to implement the project explained above. A brief overview of their functionalities are discussed below:

2.11.1 SOCKET()

Syntax Used to create socket connection.

$$int\ s = socket(domain, type, protocol)$$

where,
s: socket descriptor, an integer (like a file-handle)

domain: integer, communication domain
e.g., PF_INET (IPv4 protocol) generally used
type: communication type

- **SOCK_STREAM**: reliable, two-way, connection-based service
- **SOCK_DGRAM**: unreliable, connectionless,

protocol: specifies protocol (see file /etc/protocols for a list of options) - generally set to 0
NOTE: To implement this using UDP is to modify the "*type*" parameter as "*SOCK_DGRAM*"

Use: This will generate an endpoint that enables datagrams and will force us to handle unreliable and connectionless state.

2.11.2 BIND()

The function BIND() binds a socket with an address and reserves a port to be used by the socket.

Syntax

$$int\ status = bind(sockid, \&addrport, size);$$

status: error status, = -1 if bind failed
sockid: integer, socket descriptor
addrport: struct sockaddr, the (IP) address and port of the machine (address usually set to INADDR_ANY chooses a local address)
size: the size (in bytes) of the addrport structure.
NOTE:UDP bind() performs similarly as the TCP version. The local tcpd bind() should be called with the same parameters.

Use: Binds the created socket to the address specified. Uses standard bind().

2.11.3 ACCEPT()

Accept() accepts a connection in socket. It will extract the first connection off the queue of a listening socket. Accept is blocking: waits for connection before returning.

Syntax

$$int\ s = accept(sock, \&name, \&namelen);$$

s: integer, the new socket (used for data-transfer)
sock: integer, the orig. socket (being listened on)
name: struct sockaddr, address of the active participant
namelen: sizeof(name): value/result parameter

- must be set appropriately before call
- adjusted by OS upon return

Use: *ftps* makes use of this function while waiting to receive data from TCPD_M1.

2.11.4 CONNECT()

CONNECT() establishes connection with server. Connect is blocking.

Syntax

$$int\ status = connect(sock, \&name, namelen);$$

status: 0 if successful connect, -1 otherwise
sock: integer, socket to be used in connection
name: struct sockaddr: address of passive participant
namelen: integer, sizeof(name)

Use: CONNECT() is used to send the destination address details of *ftps* to TCPD_M2.

2.11.5 SEND()

SEND() used to send a message to a different socket. Since the sender needs to identify the desired recipient, the function operates only in a connected state.

Syntax

$$int\ count = sendto(sock, \&buf, len, flags, \&addr, addrlen);$$

count: Number of bytes transmitted (-1 if error)
buf: char[], buffer to be transmitted
len: integer, length of buffer (in bytes) to transmit
flags: integer, special options, usually just 0
addr: struct sockaddr, address of the destination
addrlen: sizeof(addr)

Use: *ftpc* uses SEND() to transfer data to TCPD_M2.

2.11.6 RECV()

RECV() is used to receive a message from a socket.

Syntax

$$int\ count = recvfrom(sock, \&buf, len, flags, \&addr, \&addrlen);$$

count: Number of bytes received (-1 if error)
buf: void[], stores received bytes
len: Length of bytes received
flags: integer, special options, usually just 0
name: struct sockaddr, address of the source
namelen: sizeof(name): value/result parameter

Use: *ftps* uses RECV() to treceive data from TCPD_M1.

2.11.7 CLOSE()

After using the socket, it should be closed. Closing a socket frees up the port used by the socket.

Syntax

$$status = close(s);$$

status: 0 if successful, -1 if error
s: the file descriptor (socket being closed)

Use: *ftpc* at the end of the message appends an "end-of-file" message. This is transferred to TCPD_M1 via TCPD_M2 which represents that the connection can be closed now.

2.12 Connection shutdown

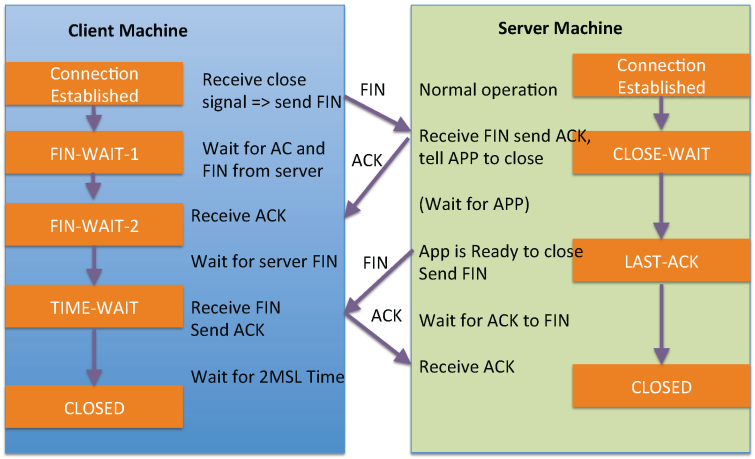
The connection termination is implemented using 4-step handshake as shown in the Figure below. The 4-steps are as follows.

- Client requests to terminate connection by sending a FIN packet to the server.
- Server receives and acknowledges the FIN packet.
- The server will then send a FIN to the client.
- The client will send an ACK and wait to time out. Server will close after receiving the ACK.

Implementation Details: Executation of CLOSE() enables *ftpc* to send end-of-file message to TCPD_M2 receiving which TCPD_M2 replies back FIN to TCPD_M1 and enters the FIN_WAIT1 state thus waiting further to hear back from TCPD_M1.

TCPD_M1 acknowledges by sending the ACK for the FIN packet received from TCPD_M2 and enters CLOSE_WAIT. However TCPD_M1 avoids transmitting a FIN back until it has received all data packets. On receiving all packets the buffer becomes empty and then TCPD_M1 reply back with a FIN and enters into the LAST_ACK state. In case the ACK from TCPD_M2 does not reach TCPD_M1 in time, it maintains a timer so that it can retransmit the FIN back to TCPD_M2. After the ACK comes back from TCPD_M2, TCPD_M1 closes its connection and enters CLOSED state.

Receiving FIN from TCPD_M1, TCPD_M2 enters the FIN_WAIT2 and sends an ACK when its buffer is empty and enters the TIME_WAIT state where it waits for a short period to ensure that last ACK sent is not lost. After that period it closes the connection.



3 Work Distribution:

- Delta Timer - Gopi Krishna Tummala
- CheckSumming - Rupam Kundu
- RTT computation, Shutdown - Gopi Krishna Tummala
- Circular Buffer Management - Rupam Kundu
- Project Overview and details - Gopi Krishna Tummala

- Detailed Description - Rupam Kundu
- Future Work - Rupam Kundu

4 Future Work

- In the current project RTO is calculated using standard Jacobson's algorithm. However problem arises when RTT decreases suddenly. Evidently Rttvar (mdev in Linux) increases as a result of which srtt goes down. Ideally, RTO should go down. But, RTO increases as it depends more on rttvar. This problem can be addressed using LINUX METHOD which can be incorporated in our project to get better performance like:

  if $(m < (srttrttvar))$

  $rttvar = (31/32) * rttvar + (1/32) * |srtt - m|$

  else

  $rttvar = (3/4) * rttvar + (1/4) * |srtt - m|$

- Communication to other TCPD processes in the system can be implemented.
- Multiple TCP connections can be implemented.
- Reliable File transfer application can be implemented on top of these.

5 References

[1] http://www1.cs.columbia.edu/ danr/courses/6761/Fall00/materials.html
[2] http://en.wikipedia.org/wiki/Circular_buffer
[3] http://en.wikipedia.org/wiki/Transmission_Control_Protocol
[4] http://en.wikipedia.org/wiki/Checksum
[4] TCP/IP Illustrated Volume 1 - The Protocols by W. Richard Stevens.