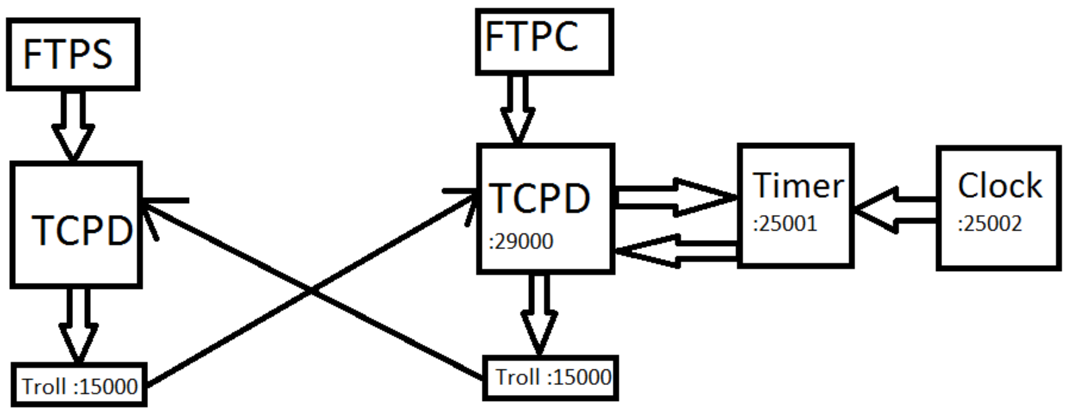
CSE 678 Project report
Winter 2012

Dhinesh Dharman, Vikram Gajanan

Project Overview:

The aim of the project is to implement a TCP like reliable communication using the unreliable UDP sockets. UDP is a connectionless protocol in which the sender sends the data to the destination. The data could be lost before reaching the destination or could be corrupted by the time it reaches the destination. The destination doesn't send any feedback to the sender making the communication unreliable. We implement a system such that a sender sends data to the destination for which the destination responds with an Acknowledgement in case the data is received without any errors. By this way the sender receives feedback for the packets it sends and knows if the packets are being received by the destination.

Our implementation is illustrated in the figure below. The numbers inside the boxes is the port number used by the processes.



The client application process (FTPC) communicates with the TCPDC process. It tells the TCPDC on which port it needs to communicate with the remote server. The TCPDC receives data from the FTPC process. It converts the raw data into a packet format and sends it to the TROLL which in turn sends the packets to the remote server (TCPDS) process. The TCPDS process receives the packets from TROLL on the sender side. It retrieves data from the packets it receives from TROLL and sends the data to the FTPS application process. The FTPS process writes the received data in a file.

In our implementation we have a single TCPD process which has both the send and receive buffers. This single process takes care of the implementation of TCPDS and TCPDC processes.

FTPC process:

The FTPC process opens a file in the sender's machine. It creates a socket and binds it to a port and sends data through this socket to the TCPD process, a maximum of 1000 bytes at a time. Initially it sends the name of the File and its size before sending the actual data in the file.

It includes sub - routines CONNECT and SEND. These sub-routines take the format of typical TCP socket routines Connect and Send. Before the sub-routine is called a TCP socket is created and passed as an argument to CONNECT. The sub-routine CONNECT extracts the parameters associated with the socket (like the fields associated with the struct sockaddr_in) and acts like a wrapper of the UDP socket function sendto. The CONNECT sub-routine sends information to the TCPDC process such as the IP address of the remote machine and the port number it needs to communicate with.

The SEND sub-routine is a wrapper function to the UDP socket call sendto. While CONNECT sends the remote IP and port number, the SEND function sends the actual data from the file.

<u>FTPS process</u>:

The FTPS process receives data from the TCPD process and writes it to a file. Initially it receives the file name and file size before receiving the actual contents of the file. The FTPS process includes sub-routines like ACCEPT and RECV. ACCEPT accepts a connection in the specified socket. It is implemented to receive 1 byte of data from the TCPD process to denote data reception from the client. The RECV sub-routine is a wrapper function to the recvfrom udp socket function.

<u>TCPD process</u>:

The TCPD process is the part of our implementation which makes the communication reliable.

-It receives data from the FTPC client process

-It converts the received data into a TCP segment by adding a 20 byte header to it

```
typedef struct{
  TCP_header  hdr;
  char        data[TCP_MSS];
}TCP_segment;
```

-It stores the TCP segments in a transmit buffer. The transmit buffer is of size 64KB. It can store many TCP segments before they are transmitted to the destination.

```
struct _TCPBUF_ {
  TCP_segment ts;
  int len;
  int txCount;
  double lastTxTime;
  struct _TCPBUF_* next;
};
typedef struct _TCPBUF_ TCP_buffer;
```

-It also implements a sliding window for the transmit buffer.

```
typedef struct _TCPWIN_ {
  int   low;
  int   high;
  char  window[WINDOW_SIZE*TCP_MSS];
} TCP_window;
```

-It computes the CRC for a packet and stores the value in the header. It also computes the CRC for an incoming packet from the TROLL on both the sender and the receiver sides.

-The TCPD process implements a receive buffer similar to the transmit buffer where it stores the TCP segments received from the sender.

-The TCPD process also communicates with a Timer process that implements a timer in case of retransmission time out

-The TCPD process has a socket bound to a port. The FTPC process, the Timer process and the TROLL from the server side send packets to the same port. When a packet is received by the TCPD process it inspects the incoming packet to determine which process has sent the packet.
We don't use the URG pointer or the URG flag in TCP header. Instead we use it to identify the process from which the packets were sent. If the URG flag contains LS – the TCP segment is a local segment sent by either the FTPC process, if the URG flag contains TS – the TCP segment is a Timer segment sent by the Timer process, if it contains RS – the TCP segment is a Remote Segment sent by the Remote Machine.

<u>Module Description</u>:

<u>CRC algorithm</u>:
-We implement a 16 bit CRC.
-CRC is calculated by the following steps:

> Writing out the number (or data in hex) for which the CRC is to be calculated and appending a 16 bit remainder (all zeros) to its end.
> Iteratively exor the data with a 17 bit polynomial function and left shift the data if the MSB of the data is a 1 or just left shift the data if the MSB of the polynomial is 0.
> Repeat the above step till the end of data. The remainder that we get is the CRC

-CRC computation is implemented in two sub-routines – one that performs the EXOR operation and the other that performs the left shift operation

-The first byte of the data is extracted and EXORed with the first byte of the CRC polynomial (if the MSB of the data is 1); the first byte is left shifted; the first bit of the next byte of data is extracted by performing an AND operation with 0x80. If the result is a 1 the first byte of (left shifted data) is ORed with a 0x01.

- The above step is repeated till the end of data is reached

```
u_short computeChecksum(char *buf, int length, u_short *crc ){

    int i,cnt;
    unsigned char buff[length+2],buff_1[1],Poly[1];
    unsigned short Crc,ans;
    unsigned char poly1 = 0xC0;
    unsigned char poly2 = 0X02;
    unsigned char poly3 = 0x80;
    bzero(buff,sizeof(buff));
    memcpy(buff,buf,length);
    memcpy(&Crc,crc,2);

    memcpy((buff+length),&Crc,2);

    for ( i = 0; i < length*8; i++){
      if(buff[0] & 0x80){
          buff[0] = poly1^buff[0];
          buff[1] = poly2^buff[1];
          buff[2] = poly3^buff[2];
      }
      leftshift_16(buff,length);
    }
    memcpy(&ans,buff,2);
    return ans;
    /*printf("crc: %d\n",ans);*/ }
void leftshift_3(unsigned char *buffer,int size){
    unsigned char temp;
    int j;
    for(j = 0; j<size; j++)
    {
        buffer[j] = buffer[j]<<1;
        temp = buffer[j+1] & 0x80;
        if(temp)
            buffer[j] = buffer[j] | 0x01;
        else
            buffer[j] = buffer[j] | 0x00;
        if (j == size-1) buffer[j+1] = buffer[j+1]<<1;
    }
}
```

Wraparound Buffer:

The send and receive buffers are wrap around buffers. These are implemented using an array. The wrap around buffer is of size 64kb. When 1000 bytes of data is sent from the send buffer, the first 1000 bytes of the send buffer becomes vacant. The content of the buffer is then left shifted by 1000 bytes. The incoming TCP segments to the send buffer are added to the end of data in the array. Similarly in the receive buffer once the first 1000 bytes of the receive buffer is sent to the FTPS process the contents of the receive buffer is left shifted by 1000 bytes and the incoming entries to the receive buffer are added to the end of data in the array.

Timer:

When the sender sends a segment to the destination it sends a message to the Timer process to start the timer for T seconds for the TCP segment TC. The timer process accepts the request and starts the timer. When the timer terminates the Timer process sends a message to the TCPD process indicating that the timer has terminated for the segment TC. The TCPD process searches for the segment TC in its buffer. Had the ACK for the segment TC arrived before the Timer times out, the segment TC would have been removed from the send buffer. If the segment TC exists in the send buffer then the ACK for the segment hasn't been received yet and the segment has to be retransmitted. The segment TC is retransmitted and the new request is sent to the timer process to start a timer for the retransmitted segment.

There are two functionalities implemented by the Timer process. One function is that it ticks every second irrespective of whether it receives a time out request. The other function is that it maintains a list of when each of transmitted segments need to time out.

A child process is created inside the timer process by forking it. The child process sends a tick every second to the parent process. This implements the Timer processes' first functionality.

```
if (fork() == 0){
    int sock = socket (AF_INET, SOCK_DGRAM, 0);
    int a = 0;
    while (1){
      if   (sendto   (sock,   &a,   4,   0,   (struct   sockaddr*)&clock,
sizeof(clock)) < 0){;
        perror("error sending tick from clock to timer");
        exit(1);
      }
      /*
      printf("Clock sending tick...\n");
      */
      sleep(1);
    }
    exit(0);
  }
```

The Timer's second functionality is implemented as a Delta List. The Delta list is a data structure like a priority queue and is well suited to manage time outs. If a linked list were to be used the key in every node of the linked list needs to be decremented by 1 for every clock tick, which becomes impossible when the linked list becomes larger and larger. In case of a delta list the objects are ordered in such a way that their key is relative to the key of the objects preceding it. If the first 4 elements in a priority queue are 10, 12, 15 and 20 the corresponding entries in a delta list would be 10, 2, 3 and 5.

The delta list is implemented as a singly linked list. Each node in the list contains 4 keys – the actual RTO, RTO relative to the preceding node, System Time and the address of the segment in the send buffer when a time out request is received. These values are sent as part of the Timeout request to the Timer process. When new requests for retransmission are received by the timer process it inserts a new node in the delta list. The position at which the new node is inserted is determined by its RTO value.

RTT/RTO:
Retransmission Timeout (RTO) is computed using the Jacobson's algorithm. According to Jacobson's algorithm, the RTT is measured each time RTO is to be computed. This is designated as measured RTT (M).

```
  /* Update M for RTO */
        if (p->txCount == 1){
```

```
      gettimeofday(&tim, NULL);
      t1 = tim.tv_sec + (tim.tv_usec/1000000.0);
      M = t1 - p->lastTxTime;     }
```

The measured RTT is smoothed and estimated each time (A) RTO is to be computed.

```
int computeRTO (){
  /* M - latest RTT measurement */
  if (ackCount == 1){
    A = M + 1;
    D = A / 2;
  }
  else if (ackCount > 1){
    Err = M - A;
    A = A + g * Err;
    D = D + h * (abs(Err)-D);
  }
  RTO = (int)(A + 4 * D);
  RTO = (RTO == 0)?1: RTO;


  printf("Current RTO = %d M = %.2f\n",RTO, M);
  return 0;
}
```

Packet Formats:

Packets are exchanged between the TCPD process and FTPC process, FTPS process and Timer Process and also between the Timer Process and the Clock Process. The TCP packet format is used for packet exchange between the TCPD process and all other processes. We don't use the URG pointer or the URG flag in TCP header to indicate urgent data. Instead we use it to identify the process from which the packets were sent. If the URG flag contains LS – the TCP segment is a local segment sent by either the FTPC process, if the URG flag contains TS – the TCP segment is a Timer segment sent by the Timer process, if it contains RS – the TCP segment is a Remote Segment sent by the Remote Machine.

The Clock process sends 4 bytes of data set to 0 to the Timer Process on each tick.

Implementation of SOCKET, BIND, SEND, RECV, CLOSE:

These functions take input arguments values as taken by the TCP socket functions Socket, Send, Bind, Recv and Close. These are just wrapper buffers which make use of UDP socket functions – sendto, recvfrom and close in their respective implementations.

```
int RECV (int sockfd, void *buf, int len, int flags){

  struct sockaddr_in TCPD;
  struct sockaddr_in tcpd;
  int TCPD_len;
  int ret;
  char b[1050];
  sockfd = socket (AF_INET, SOCK_DGRAM, 0);

  tcpd.sin_family = AF_INET;
  tcpd.sin_port = src.sin_port;
  tcpd.sin_addr.s_addr = src.sin_addr.s_addr;

  bind (sockfd, (struct sockaddr *)&tcpd, sizeof(struct sockaddr_in));
  /* Receive data via TCPD process */
  ret    =    recvfrom    (sockfd,    buf,    TCP_MSS,    flags,    (struct
sockaddr*)&TCPD, &TCPD_len);
  if (ret < 0){
    perror("error receiving data via TCPD process");
    exit(1);
  }
```

In the above function RECV src is a global variable of type struct sockaddr_in. It uses the functions socket, bind and recvfrom. The SEND, CLOSE, BIND and SOCKET functions are implemented in a similar way.

```
int SOCKET (int domain, int type, int protocol){
  return socket (AF_INET, SOCK_DGRAM, 0);
}

int CLOSE (int sockfd){
  return close(sockfd);
}
```

Compiling and Running the Program:

Execute the script startServer.sh <port no> on the server side.
Execute the script startClient.sh <Server name> <Port no> <Filename> on the client side.

Future extensions:

The current implementation doesn’t acknowledge packets cumulatively and doesn’t implement congestion control. It might not be difficult to cumulatively acknowledge packets but it is difficult to implement congestion control. We currently implement the Timer process to tick every one second. The time period of ticks can be reduced to be lesser than 1 second. This will make congestion control more efficient.