الاسم / محمود ابراهيم جاد ابوالوفا 19016532

الاسم / عبدالرحمن أحمد يسري النعناعي 19015889

# Networks Assigment2 report

## The overall organization of the program :

- The program is divided in to two subprograms
  - A program simulating the server
  - A program simulating the client
- The server program is organized through running a main function which executes a while loop which
    - First : listen for new connections
    - Accept new connection for incoming client
    - Check for the file requested
    - send the file in form of packets
    - the packet are sent in groups equal to the size to cwin
    - if stop and wait
      - then the server wait to receive the acknowledgment of each packet before sending the next one in the window
    - if selective repeat
      - then the server send the group as a whole and receive the acknowledgment of all of them assigning each ack to the right packet
      - then after all acks are received the next group is then sent
    - if timeout the cwin is then reduced to 1
    - wait for new connections
    - close if time out

  the time out is set to 3 secs if the ack is not received after that then the server will consider it a lost packet

- **The client program is organized through running a main function which executes a while loop which :**
    - **Request the desired file from server**
    - **Receive the file in form of packets**
    - **Send the acknowledgement for each packet received**
    - **Organize the packet and order them in the right order**
    - **Write the file**

## Major functions :

### client program :

```
void read_input_file(char *path, char args[][1024]) {
   FILE *filePointer;
   int bufferLength = 255;
   char buffer[bufferLength]; /* not ISO 90 compatible */
   filePointer = fopen(path, "r");
   int i = 0;
   while (fgets(buffer, bufferLength, filePointer)) {
      strcpy(args[i], buffer);
      args[i][strlen(buffer) - 2] = '\0';
      printf("%s\n", args[i]);
      i++;
   }

   fclose(filePointer);
}
```

**Used to get input from input file**

```
packet create_packet(char *data) {
   struct packet pack;
   strcpy(pack.data, data);
   pack.len = strlen(data) + 8;
   return pack;
}
```

**Used to create a packet**

```
void receive_file(char *file) {
   printf("Reading Data\n");
   int size = 500;
   int numbytes;
   char p_array[size];
   struct packet received_data;
   int expected_seqno = 1;
   ack_packet ackPacket;
   if ((numbytes = recvfrom(sockfd, &received_data, MAXBUFLEN, 0,
         (struct sockaddr*) &g_their_addr, &g_addr_len)) == -1) {
      perror("recvfrom");
      exit(1);
   }

   // This array to hold packets in place according to packet seqno
   //                  Important
   // client will receive the last package as zero to stop
   // this will be the last package in received_data[]
   // write received_data[]  except the last one.
   cout << "Num of packets will be received " << received_data.seqno << "\n";
   int n = received_data.seqno;
   int ack = 0, acks_number = 0;
   int length_of_last_packet;
   while (1) {
      if (acks_number == n - 1)
         break;
      if ((numbytes = recvfrom(sockfd, &received_data, MAXBUFLEN, 0,
            (struct sockaddr*) &g_their_addr, &g_addr_len)) == -1) {
         perror("recvfrom");
         exit(1);
      }
      memcpy(received_packets[received_data.seqno], received_data.data,
            received_data.len);
      cout << "recieved packet no : " << received_data.seqno << "\n";
      ack = received_data.seqno;
      ackPacket = create_Ack_packet(ack);
      send_ack(ackPacket);
      acks_number++;
      length_of_last_packet = received_data.len;
      cout << "sent Ack packet no : " << ackPacket.ackno << "\n";
   }
   /*here you should write received_packets to file*/
   FILE *recievedFile = fopen(file, "wb");
   cout << length_of_last_packet << " \n";
   for (int i = 0; i < n - 1; i++) {
      if (i == n - 2)
         size = length_of_last_packet - 8;
      cout << fwrite(&received_packets[i], sizeof(char), size, recievedFile);
   }
   fclose(recievedFile);
   printf("Finished reading\n");
   fflush(stdout);
}
```

**used to receive and order the packets**

```
void send_ack(ack_packet ackPacket) {
   int numbytes;
   ackPacket.len = 8;
   ackPacket.check_sum = 1;
   if ((numbytes = sendto(sockfd, &ackPacket,
sizeof(ackPacket), 0,
         (struct sockaddr*) &g_their_addr, g_addr_len)) == -1)
{
      perror("talker: sendto");
      exit(1);
   }
}
```

**used to send acknowledgment back to server**

## server program :

```
void read_input_file(char *path, char args[][1024]) {
   FILE *filePointer;
   int bufferLength = 255;
   char buffer[bufferLength]; /* not ISO 90 compatible */
   filePointer = fopen(path, "r");
   int i = 0;
   while (fgets(buffer, bufferLength, filePointer)) {
      strcpy(args[i], buffer);
      args[i][strlen(buffer) - 2] = '\0';
      printf("%s\n", args[i]);
      i++;
   }

   fclose(filePointer);
}
```

**Used to get input from input file**

```
void create_file_packets(char *path) {
   FILE *fileptr;
   long filelen;
   int numbytes;
   struct packet file_packet;
   fileptr = fopen(path, "rb");  // Open the file in binary mode
   cout << "iam here and fileptr is : " << fileptr << endl;
   fseek(fileptr, 0, SEEK_END);          // Jump to the end of the
file
   filelen = ftell(fileptr);         // Get the current byte offset
in the file
   rewind(fileptr);
   char send_buffer[500]; // no link between BUFSIZE and the file
size
   while (!feof(fileptr)) {
      int nb = fread(send_buffer, 1, 500, fileptr);
      file_packet = create_packet(send_buffer, nb);
      filePackets.push_back(file_packet);
   }
   file_packet = { 0, 0, 0, "" };
   filePackets.push_back(file_packet);
   return;
}
```

Used to convert file into packet

```
void congestion_control_selective() {
   uniform_real_distribution<> dis(0, 1);
   mt19937 gen(seed);
   char empty[] = "";
   bool finished = false;
   // This packet contains number of packets will the
   // Client receive
   packet p = create_packet(empty, 0);
   p.seqno = filePackets.size();
   send_packet(p);
   int packetCounter = 0, sent_packets_num, ack_counter;
   int total_acks = 0;
   state s = slow_start;
   while (!finished) {
      cout << cwnSize << "\n";
      // Send packets to client and set timer for each one.
      int remained_size = (filePackets.size() - 1 - packetCounter);
      sent_packets_num = min(cwnSize, remained_size);
      int begin = packetCounter;
      for (int i = 0; i < sent_packets_num; i++) {
         struct p_timer t = { (uint32_t) packetCounter, 0,
               chrono::system_clock::now() };
         packetsTimer.push_back(t);
         filePackets[packetCounter].seqno = packetCounter;
         float propToSend = dis(gen);
         if (propToSend > plp) {
            cout << "packet sent : " << filePackets[packetCounter].seqno
                  << endl;
            send_packet(filePackets[packetCounter]);
         } else {
            cout << "packet dropped seq no = "
                  << filePackets[packetCounter].seqno << endl;
         }
         packetCounter++;
      }

      // receive Acks from client
      ack_counter = begin;
      int received_acks = 0;
      while (ack_counter < packetCounter) {
         /*cout << received_acks << "\n";*/

         if (received_acks == sent_packets_num)
            break;
         ack_packet ack = recieve_ack_packet();
         if (ack.len != 0) {
            packetsTimer[ack.ackno].numberOfAcks++;
            if (packetsTimer[ack.ackno].numberOfAcks == 1) {
               received_acks++;
               if (s == fast_recovery) {
                  s = congestion_avoidance;
               }
               cwnSize += 1;
            } else if (packetsTimer[ack.ackno].numberOfAcks == 2) {
               received_acks--;
            } else {
               received_acks--;
               if (s == slow_start) {
                  ssthreshold = cwnSize / 2;
                  cwnSize = ssthreshold + 3;
                  s = congestion_avoidance;
               } else if (s == congestion_avoidance) {
                  ssthreshold = cwnSize / 2;
                  cwnSize = ssthreshold + 3;
                  s = fast_recovery;
               }
               packetsTimer[ack.ackno].numberOfAcks = 0;
               send_packet(filePackets[packetsTimer[ack_counter].seqno]);
```

**used to handle congestion control and state transitions**

```
ack_packet recieve_ack_packet() {
   struct ack_packet received_ack;
   struct pollfd pfd = { .fd = clientsock, .events = POLLIN };
   int state = poll(&pfd, 1, 1);
   recvfrom(clientsock, &received_ack, MAXBUFLEN, 0,
         (struct sockaddr*) &g_their_addr, &g_addr_len);
   if (state != 0) {
      cout << "Ack recived : " << received_ack.ackno << endl;
      return received_ack;
   }
   return empty_ack;
}
```

**Used to receive acknowledgment packet**

```
void get_file_name(char *path, char *file_name) {
   char parsed[100][1024];
   char *token;
   char *rest = path;
   int i = 0;

   while ((token = strtok_r(rest, "\\", &rest))) {
      strcpy(parsed[i], token);
      i++;
   }
   cout << "last element is : " << parsed[i - 1] << endl;
   strcpy(file_name, parsed[i - 1]);
}
```

**Used to get file name**

```
ack_packet create_Ack_packet(int ack_no)
{
   struct ack_packet pack;
   pack.len = 8;
   pack.ackno = ack_no;
   return pack;
}
```

**Used to create acknowledgment packet**

## Data structures :

**Struct packet**

Containing :

Checksum field

Length field

Seqno field

Data field (array)

**Used to simulate packet**

**Struct ack_packet**

Containing :

Checksum field

Length field

Ack_no field

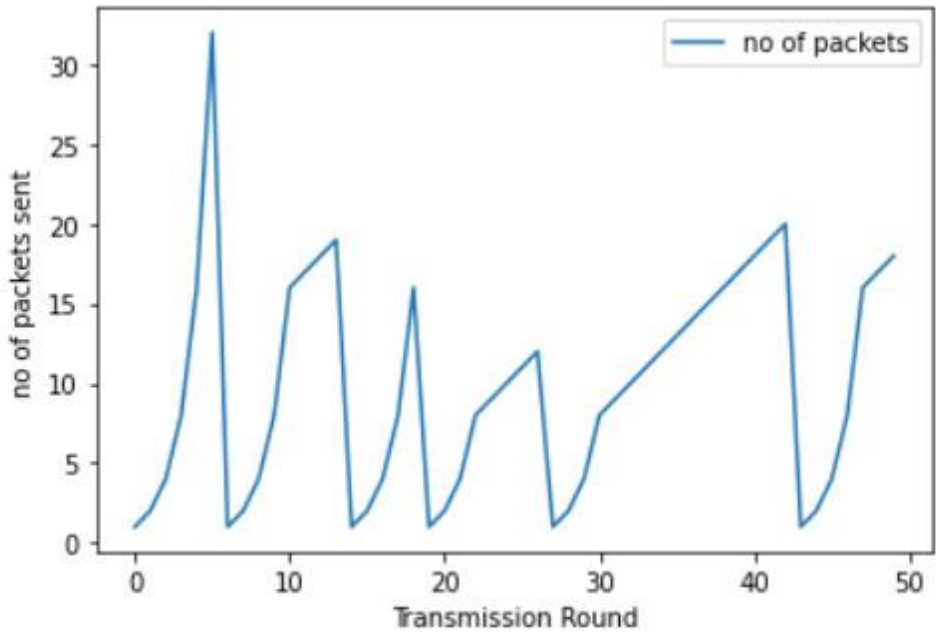
**Used to simulate acknowledgment packet**

**Array :** to store data

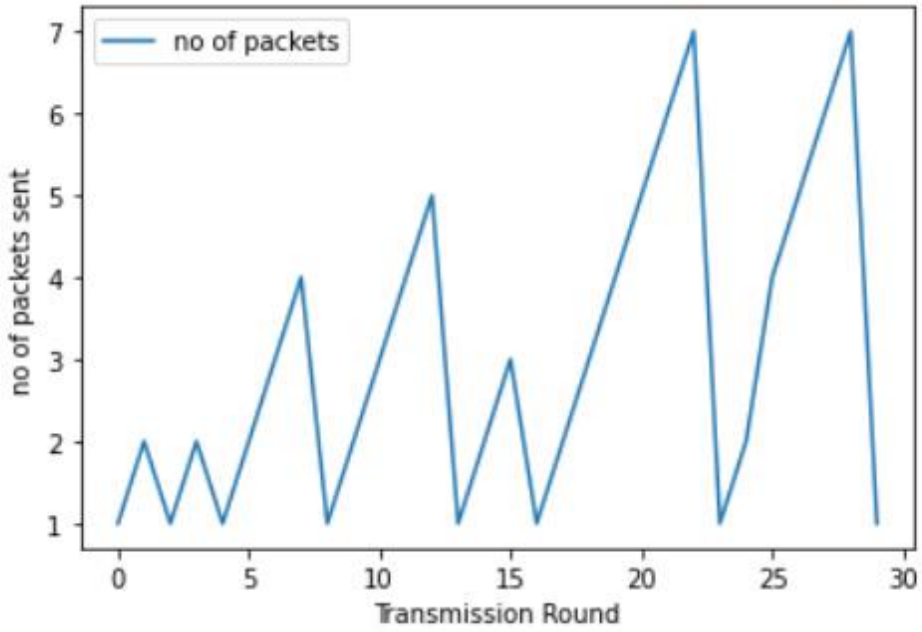
**Vector of packet :** to store converted file to packets

# Analysis :

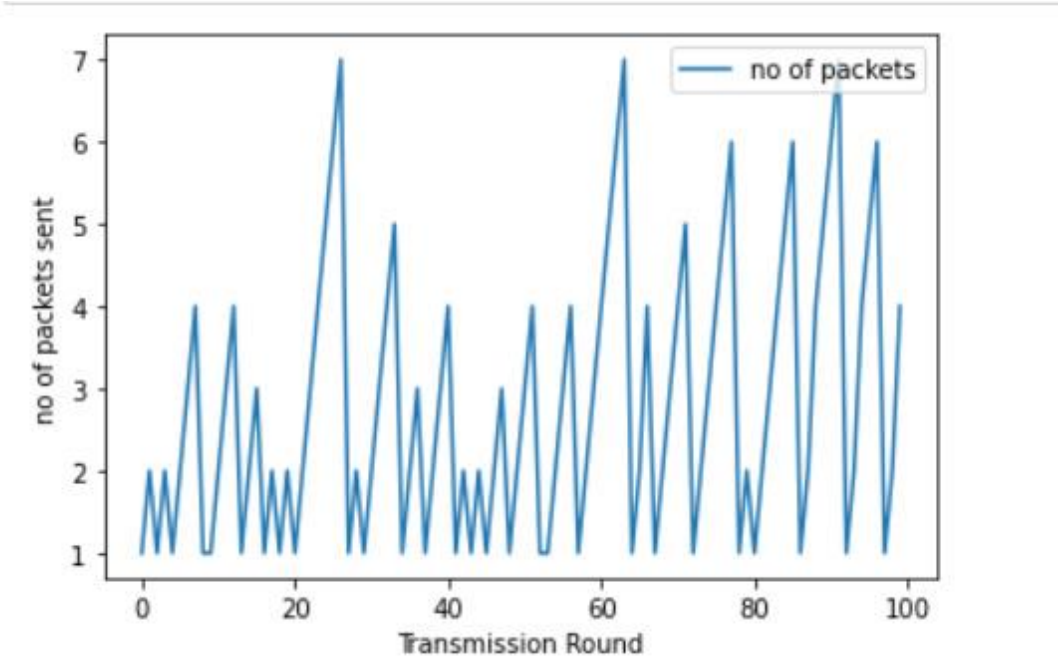
## For plp = 0.01




## For plp = 0.05



## For plp = 0.1




## Plp = 0.3