소켓 프로그래밍

21000247 박고언
21000494 이성경

HW1. TCP를 이용한 파일 전송 – 박고언

■ Question & Answer

1. 프로그램 작성 시 특히 주의해야 할 사항

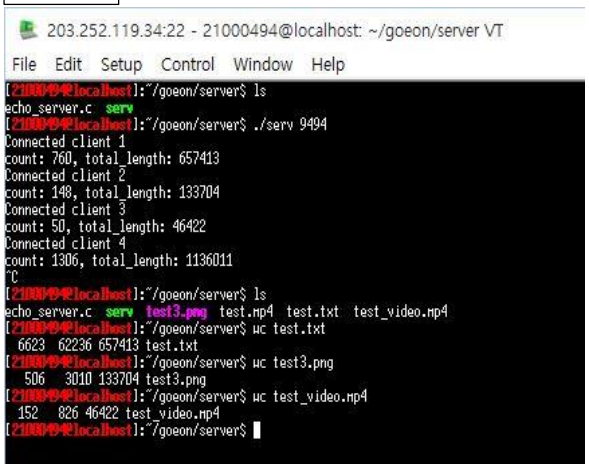
1) TCP는 byte stream protocol로서, 메시지의 경계를 유지하기 않기 때문에 처음에 보낸 것을 file name으로, 그 다음부터는 데이터로 받아들이게 하면 파일 이름에 데이터가 섞여서 저장될 수 있다.

=> 클라이언트에서 strlen(file_name)을 통해 얻은 file_name_size를 서버에게 전송한다. File name size를 전송할 때의 버퍼사이즈는 클라이언트와 서버 둘 다 sizeof(int) 즉, 4바이트이다. 그리고 두 번째로 file name을 전송할 때 버퍼 사이즈를 file_name_size로 하여 클라이언트에서 send하고 서버에서 recv하도록 한다. 파일 이름을 전송할 때 클라이언트의 send 버퍼 사이즈와 서버의 recv 버퍼 사이즈를 동일하게 함으로써 파일 이름과 다른 데이터가 섞이지 않도록 하였다.

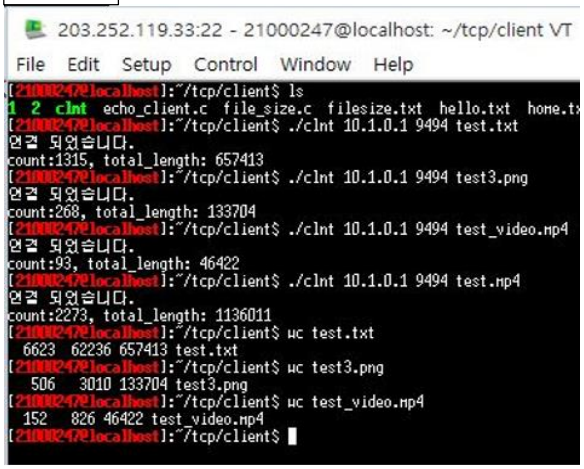
2. TCP가 byte-stream protocol이란 의미가 socket programming을 할 때 어떠한 점을 고려하여야 한다는 의미인가? 그리고 UDP와 비교하였을 때 어떠한 점에서 차이가 있는가? (programming HW2와 비교 설명)

=> TCP는 byte stream protocol이기 때문에 메시지 간의 경계를 구분하지 않는다. 따라서 client가 처음에 파일 이름을 보내고 그 다음부터 데이터를 전송한다 해도 server에서 맨 처음 읽은 데이터를 파일 이름이라고 할 수 없다. 파일 이름과 데이터가 섞일 가능성이 있기 때문이다. TCP가 동작하는 것을 보면 아래의 그림과 같다.

server



client



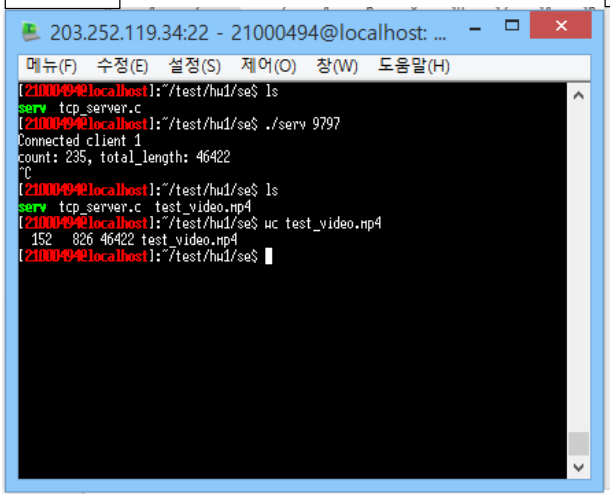
위의 그림은 server에서는 recv를 콜 하는 수를, client에서는 send를 콜 하는 수를 count를 통해 측정하였다. Client의 버퍼 사이즈는 500이고 server는 1000이다. Test.txt 파일을 보낼 때 client에서 send를 콜 한 수는 1315번인 반면, server에서 recv를 콜 한 수는 760번이다. 즉, Byte-stream protocol이므로 메시지의 경계를 구분하지 않기 때문에 server에서는 버퍼의 용량이 초과하지 않는 한 데이터를 버퍼에 채우고 읽어 들일 수 있다. 주의할 사항은 file_name과 데이터가 겹치지 않도록 file_name의 버퍼 사이즈를 고정 시켜 놓는 것이다.

반면, UDP는 메시지 간의 경계를 유지한다. 따라서 client가 파일 이름을 전송하면 그 파일 이름이 client가 보낸 다른 패킷들과 구분되므로 client가 보낸 파일 그대로 저장할 수 있다. 하지만 tcp의 경우 파일 이름과 다른 데이터가 섞일 가능성이 있기 때문에 서버에서 파일 이름을 읽을 때 파일 이름의 길이만큼만 데이터를 읽도록 하였다.

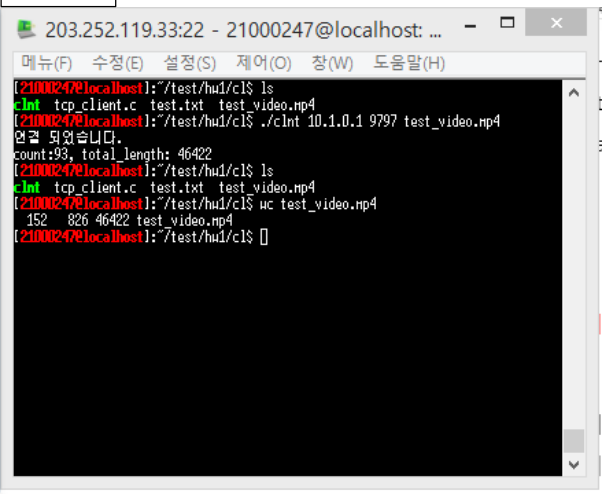
3. read buffer size가 write의 buffer size 보다 작을 때 어떤 현상이 발생하는가?

Server의 버퍼사이즈를 200, client의 버퍼 사이즈를 500으로 하여 실험 하였다.

server



client



: Tcp는 byte stream protocol이므로 서버의 버퍼가 초과하지 않는 만큼의 바이트를 수용하는 것이 가능하다. 따라서 클라이언트의 버퍼사이즈가 더 큰 경우, 서버는 여러 번의 recv를 통해 데이터를 읽어 들인다. 따라서 데이터의 손실이 발생하지 않는다.

■ 주요 부분 코드 및 설명

- TCP_Server
  - client와의 연결을 위한 소켓을 생성하고 소켓 생성 후, 소켓에 server의 IP 주소와 port 번호를 할당한다. 그리고 client가 요청할 때까지 대기.
  - 클라이언트의 연결 요청에 대해 accept를 콜 함으로써 연결요청을 수락하고 데이터 입출력에 사용할 소켓(clnt_sock)을 생성한다.

```
if((serv_sock = socket(PF_INET, SOCK_STREAM, 0)) == -1){
  error_handling("socket() error");
}

memset(&serv_addr, 0, sizeof(serv_addr)); // serv_addr의 사이즈만큼 serv_addr
  을 0으로 클리어
serv_addr.sin_family = AF_INET;
serv_addr.sin_addr.s_addr = htonl(INADDR_ANY);
serv_addr.sin_port = htons(atoi(argv[1]));

if(bind(serv_sock, (struct sockaddr*)&serv_addr, sizeof(serv_addr)) == -1)
  error_handling("bind() error");

if((listen(serv_sock, 5)) == -1)
  error_handling("listen() error");
```

서버에서 소켓을 생성하고 serv_sock에 소켓 파일 디스크립터 전달

서버의 주소 정보를 소켓에 할당

연결 요청 대기 상태에 두고자 하는 소켓의 파일 디스크립터와 연결요청 대기 큐 크기 정보

Iterative 한 형태를 지니기 위해 while문 안에 accept를 넣음

```
while(1){
  if((clnt_sock = accept(serv_sock, (struct sockaddr*)&clnt_addr, &clnt_adr_sz
  )) == -1)
    error_handling("accept() error");

  printf("Connected client %d\n", i++);
  str_len = recv(clnt_sock, &file_name_size, sizeof(file_name_size), 0);

  // 제목을 읽어서 파일 이름으로 저장하기
  str_len = recv(clnt_sock, file_name, file_name_size, 0); // file name size
  만큼만 읽음
  file_name[str_len] = '\0';

  if(str_len == 0)
    error_handling("read() error");

  fp = fopen(file_name, "ab");        // 받은
  if(fp == NULL) error_handling("fopen() error");

  while((str_len = recv(clnt_sock, message, BUF_SIZE, 0)) != 0){
    total_str_len += str_len;
    ret = furite(message, 1, str_len, fp);
    memset(message, 0, sizeof(message));
    count++;
  }
  printf("count: %d, total_length: %d\n", count, total_str_len);
  total_str_len = 0;
  count = 0;
  fclose(fp);
  close(clnt_sock);
}
```

Integer 변수 file_name_size에 client가 전송한 파일 이름의 길이를 저장함

위에서 저장한 file_name_size 값 만큼만 file_name을 읽어 들임

File_name의 마지막에 EOF를 나타내도록 '₩0'을 넣어줌

읽은 file_name으로 file open

BUF_SIZE는 1000이다. 버퍼를 초과하지 않는 크기로 데이터 수신함. 수신한 데이터는 fwrite를 통해 파일에 씀.

- TCP_Client

연결 요청

```
if(connect(sock, (struct sockaddr*)&serv_addr, sizeof(serv_addr)) == -1
  error_handling("connect() error");
else
  printf("연결 되었습니다.\n");

// file open
FILE *fp = fopen(file_name, "rb");
if(fp == NULL) error_handling("fopen() error");

// file size check
file_name_size = strlen(file_name);

// file name size 전송
str_len = send(sock, &file_name_size, sizeof(file_name_size), 0);
// file name 전송
str_len = send(sock, file_name, strlen(file_name), 0);

// 파일에서 내용을 읽어서 서버로 보낼 것
while( !feof(fp) ){
  file_read_size = fread(message, 1, sizeof(message), fp);
  str_len = send(sock, message, file_read_size, 0);
  total_write += str_len;
  count++;
}
```

파일 사이즈를 서버에 전송하여 file_name을 보냈을 때 서버가 file_name_size 만큼만 file name을 수신하도록 한다.

파일로부터 읽은 데이터를 send를 통해 클라이언트에 전송

■ 코드 작성 중 발생한 문제점과 해결방안

1. 데이터의 전송이 끝나고 서버에서 수신한 파일의 사이즈가 클라이언트에서 전송한 파일 사이즈보다 더 큰 문제가 있었다.

=> 처음에 클라이언트에서 파일을 send할 때 버퍼 사이즈를 sizeof(message)로 하였었다. 따라서 마지막에 데이터가 버퍼의 사이즈보다 적게 남았다 해도 버퍼 사이즈만큼 데이터를 보내기 때문에 server에서의 파일 사이즈가 더 컸었다. 따라서 fread의 리턴값을 file_read_size에 저장하여 파일로부터 읽은 값을 저장하여 send의 버퍼 사이즈로 지정함으로써 파일 사이즈가 더 크지 않고 똑같게 할 수 있었다.

2. client가 파일 이름을 보내고 다음부터 파일 내용을 전송하는데 서버에서 파일이름을 그 다음에 들어오는 데이터와 함께 읽어서 파일 이름이 이상하게 저장되는 현상이 발생했다.

=> 클라이언트가 처음에 파일 이름의 길이를 서버에 보낸 뒤 서버는 파일의 길이만큼 파일의 이름을 수신한다. 따라서 서버에서는 파일 이름의 길이만큼만 파일의 이름을 수신하여 파일 이름으로 정하고 그 다음은 모두 파일에 쓴다.

HW2. UDP를 이용한 파일 전송 – 이성경

■ Question & Answer

1. 프로그램 작성 시 특히 주의해야 할 사항

1) UDP를 사용하면 에러 복구를 하지 않기 때문에 client에서 전송한 패킷이 server에 도착한다는 보장이 없다. 문제의 조건에 의하면 client가 전송한 file name으로 server에서 file name을 지정하여야 하는데 그 패킷이 도착한다는 보장이 없다.

=> iterative 서버를 만들기 위해 client에 따라 다른 임의의 파일 이름이 필요 했다. 따라서 time stamp 값을 파일 이름으로 설정 하였다. 처음에 파일을 열 때, unix time stamp로 파일 이름을 설정하여 연다. 그리고 들어 오는 패킷을 fwrite를 통해 쓴다. client에서 파일의 데이터를 다 보내면 file name을 전송 한다. server에서 받으면 받은 file name으로 파일의 이름을 rename하고 그렇지 않으면 초기에 설정한 파일 이름을 바꾸지 않는다.

2) UDP를 사용하면 client가 보낸 패킷의 순서대로 server가 받는 것이 아니다. 따라서 server에서 파일 이름과 데이터를 구분해야 하는데 순서가 섞이므로 client에서 전송한 파일 이름이 server에서는 데이터로 저장 될 수도 있고, client에서 보낸 데이터가 server에서 파일 이름으로 저장할 수도 있다.

=> client에서 파일을 보낼 때 파일 이름에는 "FILE:"을, 데이터에는 "DATA:"를 패킷의 첫 부분에 붙여 줌으로써 구분자 역할을 하도록 한다. server에서는 패킷을 받을 때마다 앞의 5바이트를 검사하여 "DATA:" 일 경우 데이터에, "FILE:"일 경우 파일 이름으로 사용하도록 하여, 파일 이름과 데이터가 섞이는 일이 없도록 하였다.

3) Client는 파일을 다 보내고 close 하면 되지만 Server의 경우, client에서 보낸 패킷이 손실될 수 있기 때문에 언제 close 해야 하는 지 모호하다.

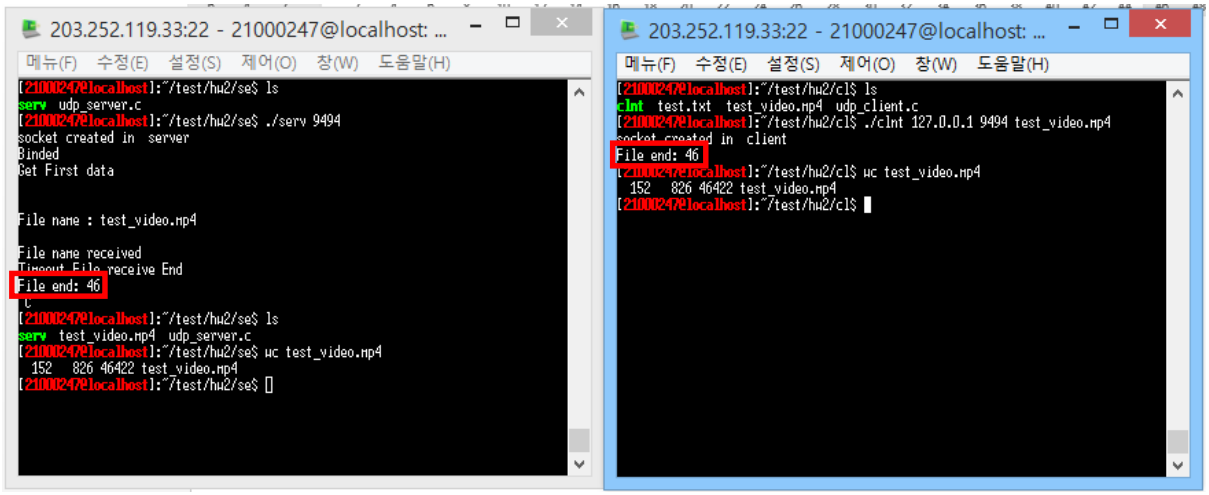
=> 패킷이 들어올 때마다 timer가 동작한다. 만약 Time out이 발생하면 더 이상 패킷이 들어오지 않기 때문에 client가 close 되었다고 판단하고 다른 클라이언트의 파일을 전송 받기 위해 새로운 파일을 연다

2. UDP에서는 message boundary를 유지한다는 의미를 보이기 위한 실험 방법을 디자인하고 이에 대한 결과를 서술하시오.

네트워크 상에서의 패킷 loss가 없는 상황에서 실험하기 위해 루프백 주소를 이용하였다. 서버와 클라이언트의 버퍼 사이즈를 다르게 하고 클라이언트에서 서버에게 파일을 전송 한다. UDP는 message의 boundary를 유지하기 때문에 서버가 충분히 더 많은 패킷을 버퍼에 저장할 수 있어도 클라이언트가 sendto를 콜 하는 수와 서버가 recvfrom을 콜 하는 수가 같을 것이다. 서버의 버퍼는 2048, 클라이언트는 1024로 하여 test_video.mp4 파일을 전송한다.

server

client



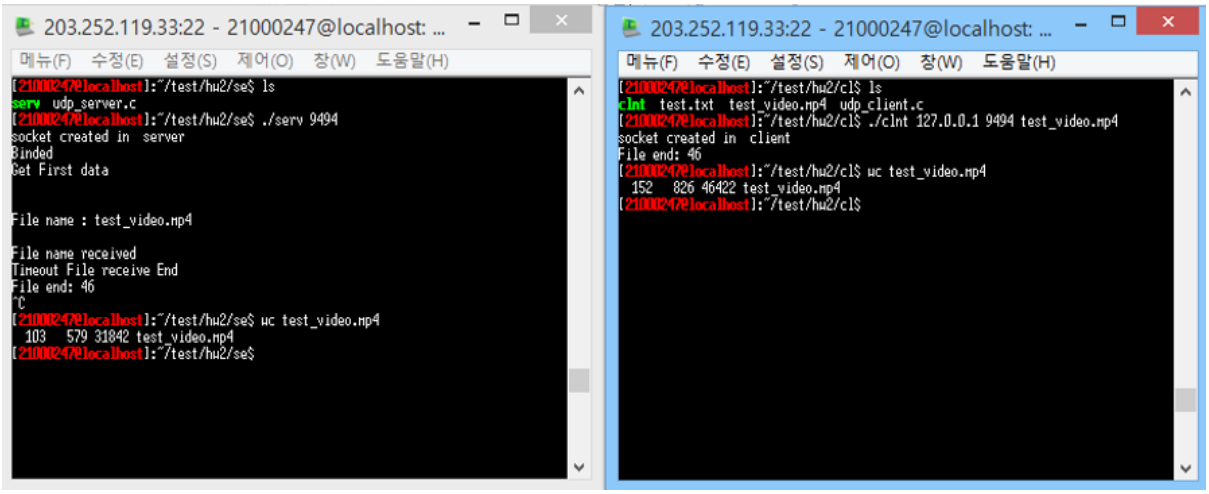
결과를 통해 알 수 있듯이 클라이언트가 46번 쓰고 서버가 46번 읽는다. 이것을 통해 클라이언트가 한 번 보내면 서버도 그 패킷을 바로 읽어 들인다는 것을 알 수 있다. 클라이언트가 한 번 1024바이트의 패킷을 보낼 때 서버는 2048 – 1024 = 1024바이트의 여유 공간이 있음에도 메시지의 경계를 유지 하기 때문에 바로 읽어 들인다. 따라서 클라이언트와 서버가 sendto, recvfrom한 횟수가 각각 같다.

3. read buffer size가 write의 buffer size 보다 작을 때 어떤 현상이 발생하는가?

서버의 버퍼는 700, 클라이언트의 버퍼는 1024로 하였다. 네트워크 상의 패킷 손실이 없을 때의 환경을 위해 루프백 주소를 사용하여 실험하였다. client에서 46422 바이트의 test_video.mp4 파일을 서버에 전송한다.

server

client



원래 46422 바이트였던 파일이 파일 전송이 완료된 후, 서버가 받은 파일은 31842 바이트 밖에 되지 않음을 알 수 있다. 즉, tcp와 달리 udp에는 메시지의 경계를 구분하기 때문에 서버의 버퍼 사이즈보다 큰 메시지가 들어오면 서버의 버퍼 사이즈만큼만 받아들이고 나머지는 버리게 된다.

따라서 데이터의 손실이 발생한다.

■ 주요 부분 코드 및 설명

- UDP_Server

```
while(1){
  // First data must be non block
  memset(buffer, 0, sizeof(buffer));
  file_recv_size = recvfrom( sd, buffer, BUFF_SIZE, 0, (struct sockaddr *)&cliaddr, &len );
```

Iterative 서버를 만들기 위해 첫 번째 패킷은 timeout이 없고, 두 번째 패킷부터 timer를 세팅한다.

```
static const char PRESET_NAME[] = "FILE:";
static const char PRESET_DATA[] = "DATA:";
```

클라이언트에서 구분자를 넣어서 패킷을 전송하므로 구분자를 선언한다.

```
// Timeout value setting
struct timeval tv;
```

Time out을 설정하기 위한 구조체 선언

```
// Open file mode write
timestamp_sec = (int)time(NULL);
sprintf(temp_name, "%d", timestamp_sec);
fp=fopen( temp_name, "wb" );
if( fp==NULL ){
  printf("Error\n");
}
```

그 프로그램이 동작할 때의 시간을 timestamp를 통해 파일 이름을 설정한다. 처음에는 시간 값을 파일 이름으로 설정하여 데이터를 그 속에 저장하고, 파일 이름이 올 경우 rename 한다.

```
// Set timeout
tv.tv_sec = TIMEOUT;
tv.tv_usec = 0;
if (setsockopt(sd, SOL_SOCKET, SO_RCVTIMEO,&tv,sizeof(tv)) < 0) {
  perror("Error");
}
```

파일이 들어올 때마다 타임 아웃 value를 세팅한다. TIMEOUT은 3으로 define되어 있다. 파일이 들어올 때마다 timeout값을 설정해 줌으로써 time out이 발생하면 서버가 종료되도록 하였다.

```
while(1){
  if( strncmp(buffer, PRESET_NAME, PRESET_SIZE) == 0 ){
    // Rename File name and Break cause it is last data
    memmove(&buffer[0], &buffer[PRESET_SIZE], strlen(buffer));
    printf("\n\nFile name : %s\n\n",buffer);
    if( rename( temp_name, buffer ) == -1 ){
      printf("Error file rename\n");
    }
    printf("File name received\n");
  } else {
    memmove(&buffer[0], &buffer[sizeof(PRESET_DATA)-1], sizeof(buffer));
    // wirte file
    fwrite( buffer, sizeof(char), file_recv_size-strlen(PRESET_DATA), fp);
  }

  i++;
  // If Client file end but last send data is loss, then timeout and break;
  memset(buffer, 0, sizeof(buffer));
  if( (file_recv_size = recvfrom( sd, buffer, BUFF_SIZE, 0, (struct sockaddr *)&cliaddr, &len )) < 0){
    printf("Timeout File receive End\n");
    break;
  }

}
```

수신한 데이터는 "DATA:" 혹은 "FILE:"을 맨 데이터의 처음에 구분자로 가진다. 따라서 FILE: 일 경우 파일 이름을 뜻 한다. 파일 이름이 들어올 경우 time stamp값으로 저장되어 있는 파일의 이름을 rename 한다.

FILE:이 아닐 경우, FILE:을 제외한 나머지 데이터를 fwrite하여 파일에 쓴다.

Time out이 걸릴 경우 recvfrom의 리턴값이 -1이다. 타임 아웃이 생기면 client의 데이터의 전송이 끝난 것으로 간주하고 서버도 종료한다.

```
// reset socket timeout opt
tv.tv_sec = 0;
tv.tv_usec = 0;
if (setsockopt(sd, SOL_SOCKET, SO_RCVTIMEO, &tv,sizeof(tv)) < 0) {
  perror("Error");
}
```

Iterative 하게 동작하므로 time out을 해제함

- UDP_Client

```
static const char PRESET_NAME[] = "FILE:";
static const char PRESET_DATA[] = "DATA:";
```

구분자 설정. 파일 이름 앞에는 "FILE:"을, 데이터 앞에는 "DATA:"를 붙인다.

```
while( !feof(fp) ){
  file_size = fread(file_temp_buffer, sizeof(char), sizeof(file_temp_buffer)-strlen(PRESET_DATA), fp);
  memcpy( file_buffer, PRESET_DATA, sizeof(PRESET_DATA) );
  memmove( &file_buffer[sizeof(PRESET_DATA)-1], &file_temp_buffer[0], sizeof(file_temp_buffer)-strlen(PRESET_DATA) );
  sendto( sockfd, file_buffer, file_size+strlen(PRESET_DATA), 0, (struct sockaddr *)&servaddr, sizeof(struct sockaddr) );
  memset(file_buffer, 0, sizeof(file_buffer));
  i++;
}
```

Fread 할 때 주의 할 점은 버퍼에서 구분자가 들어갈 자리는 빼고 파일을 읽어야 하는 것이다. 일단 파일을 읽으면 file_temp_buffer의 맨 앞에 "DATA:"를 붙여서 Server에서 DATA로 인식하도록 한다.

Sendto를 할 때에도 fread를 통해 읽어 들인 바이트(1바이트씩 여러 번이므로 리턴값은 읽은 바이트의 수)+구분자의 길이만큼 전송한다.

i는 총 몇 번 sendto를 실행하였는 지 count 한다.

```
// Send file name to Server
memset(name_buffer, 0, sizeof(name_buffer));
snprintf( name_buffer, sizeof(name_buffer), "%s%s", PRESET_NAME, argv[3] );
sendto( sockfd, name_buffer, strlen(name_buffer), 0, (struct sockaddr *)&servaddr, sizeof(struct sockaddr) );
```

파일의 데이터 전송이 끝나면 Argv[3](입력한 파일 이름)의 맨 앞에 "FILE:"을 붙여서 마지막으로 전송한다. 서버가 이 패킷을 받으면 파일 이름을 바꿀 것이고 못 받으면 처음에 저장한 그대로 파일 이름은 timestamp 값일 것이다.

■ 코드 작성 중 발생한 문제점과 해결방안

1. client는 데이터를 다 전송하면 끝을 내면 되지만 server에서는 client에서의 전송이 끝났음을 알 수 있는 방법이 없었다.

=> 에러를 복구하는 것이 아니므로 서버가 데이터를 수신할 때마다 timer가 동작한다. 3초로 세팅 되어 있다. 타이머가 없다면 서버는 계속해서 대기 하겠지만 타이머를 설정하여 3초간 패킷이 들어오지 않으면 client에서의 전송이 끝난 것으로 간주하고 서버를 종료하도록 하였다.

2. UDP는 client에서 보낸 순서대로 server에서 수신하지 않는다. 파일 이름을 받아서 서버에서도 그 이름으로 파일 이름을 설정하여야 하는데 무슨 패킷이 파일 이름인지, 파일 내용인지 구분이 필요했다.

=> 구분자를 넣어 해결하였다. 파일 이름의 경우 맨 앞에 "FILE:"을, 파일 내용일 경우 맨 앞에 "DATA:"를 붙여서 전송함으로써 서버에서 수신하면 "FILE:"이 붙어 있을 경우 파일 이름으로 rename하고 "DATA:"가 붙어 있을 경우 fwrite를 통해 파일에 쓴다.

HW3. UDP를 이용한 Reliable한 파일 전송

■ 디자인한 프로토콜 기술(Stop & Wait)

- **Server**
  - 버퍼 사이즈를 2048로 설정
  - i는 0부터 시작한다. i가 0일 때 last_i값과 curpacket.seq이 일치하고 curpacket.length <= 0이라면 이미 종료된 클라이언트가 종료되기 전에 전송했던 패킷으로 인식한다. 그리하여 ack을 보낸다.

- 들어온 패킷의 seq 넘버와 i가 일치할 때, i+1을 ack의 seq 넘버로 하여 ack 패킷을 전송한다. i가 0이라면 들어온 패킷의 데이터로 파일 이름을 설정한다.
- 들어온 패킷의 seq 넘버와 i가 일치할 때, i+1을 ack의 seq 넘버로 하여 ack 패킷을 전송한다. i가 0이 아니라면 데이터를 뜻하므로 fwrite를 통해 파일에 쓴다.
- 들어온 패킷의 seq 넘버와 i가 일치하지 않으면 기대하는 seq 넘버 값으로 다시 ack을 보낸다.
-

- **Client**
  - 파일 이름 버퍼 256. 버퍼 사이즈 1024, time out value는 0.4초로 설정
  - 파일 이름의 크기는 128을 초과해서는 안 된다.
  - i는 0부터 시작한다. i가 0일 때는 파일 이름을 전송하고 있음을 의미 한다. Packet을 보내고 그에 대한 ack을 받으면 i를 1 증가 시킨다.
  - 패킷을 보낼 때 i의 값을 sequence 넘버로 하여 전송하고, 패킷을 보내면 ack이 들어올 때까지 block 상태에 놓인다.
  - Ack의 seq 넘버가 i+1(다음으로 전송해야 할 패킷의 seq 넘버)와 일치하면 i를 1 증가시키고 다음 패킷을 전송 한다.
  - Ack의 seq 넘버가 i+1과 일치하지 않을 경우, 다른 ack이 들어올 때까지 다시 block 상태에 놓이며, 원하는 ack이 들어와야 그 다음 패킷을 전송할 수 있다.

➢ File을 다 전송하면 client를 종료한다.

■ 주요 부분 코드와 설명

● Server

```
int i=0, last_i=0;
```

Server에서는 packet을 읽을 때마다 i 값을 1 증가 시킨다. 그래서 timeout이 걸리면 last_i에 마지막 i값을 저장한 뒤 서버를 종료한다. => udp에서의 reliable한 종료를 위해 구현한 변수

```
struct spacket
{
  int seq;
  size_t length;
  char data[BUFF_SIZE];
};

struct spacket curpacket, ackpacket;
```

Seq: sequence number
Length:
Data: 전송하고자 하는 데이터
이 값들을 지닌 구조체 변수를 선언.
Curpacket은 보내는 packet, ackpacket은 잘 받았는 지 검사하기 위한 구조체이다.

```
while(1){
  memset(&curpacket, 0, sizeof(curpacket));
  printf("Wait data\n");
  recv_size = recvfrom( sd, &curpacket, sizeof(curpacket), 0, (struct sockaddr *)&cliaddr, &len );

  curpacket.seq = ntohl(curpacket.seq);
  curpacket.length = ntohl(curpacket.length);

  // If process end but send last data, then just send ack packet
  if( i == 0 && curpacket.length <= 0 && curpacket.seq == last_i ){
    memset(&ackpacket, 0, sizeof(ackpacket));
    // Set next expected seq and send
    ackpacket.seq = htonl(last_i+1);
    sendto( sd, &ackpacket, sizeof(ackpacket), 0, (struct sockaddr *)&cliaddr, sizeof(cliaddr) );
    continue;
  }
```

패킷을 받아서 그 패킷의 seq 넘버와 length를 저장한다.
그 패킷이 length가 0보다 작고 seq 넘버가 last_i와 같다는 것은, 프로세스를 종료한 이전 클라이언트가 보냈던 마지막 패킷이 도착했음을 의미한다. Iterative하므로 다른 client와의 통신 중에 그 이전 client의 마지막 패킷이 도착할 수가 있다. 이 마지막 패킷에 대해 단지 ackpacket만 보내고 server는 다시 패킷을 받기 위해 continue;를 한다.

```
// If expected seq number
if( curpacket.seq == i ){
  memset(&ackpacket, 0, sizeof(ackpacket));
  // Set next expected seq and send
  ackpacket.seq = htonl(i+1);
  sendto( sd, &ackpacket, sizeof(ackpacket), 0, (struct sockaddr *)&cliaddr, sizeof(cliaddr) );

  // First packet is include File name
  if( i == 0 ){
    fp=fopen( curpacket.data, "wb" );
    if( fp==NULL ){
      printf("Error\n");
      break;
    }
    // file open success last_i init
    last_i = 0;
  }
  // else furite data
  else {
    if( fp==NULL ){
      printf("No file pointer\n");
      continue;
    }
    if( curpacket.length > 0 ){
      furite( curpacket.data, sizeof(char), curpacket.length, fp);
    } else {
      // If get last packet, init
      last_i = i;
      i = -1;
      fclose(fp);
      printf("flocse\n");
    }
  }
  // Increase i
  i++;
} else {
  // Resend ack packet
  sendto( sd, &ackpacket, sizeof(ackpacket), 0, (struct sockaddr *)&cliaddr, sizeof(cliaddr) );
}
```

i 값은 받기를 기대하는 sequence 값이다. 따라서 i 값과 현재 seq 값이 같다면 server가 기대하는 패킷과 일치하는 값이므로 i를 1 증가시킨 seq넘버로 ackpacket을 보낸다.

i가 0이라는 것은 file_name을 보낸 것을 의미한다.(client에서 seq 넘버를 0으로 처음 보낼 때 file_name을 보내므로) 따라서 i 가 0일 경우 그 패킷의 data로 파일 이름을 설정한다.

i 가 0이 아니라면 받은 패킷은 파일 이름이 아닌 데이터를 뜻한다. 따라서 fwrite로 파일에 쓴다. Curpacket.length가 0보다 작을 때는 클라이언트가 보낸 마지막 패킷을 의미한다. 이럴 경우, 서버를 종료해야 하는데 그때의 i 값을 기억하기 위해 last_i에 저장한다.

Current 패킷(curpacket.seq)이 기대하는 패킷(i)과 일치하지 않을 때 기대하는 패킷(i)로 ackpacket을 전송한다.

- **Client**

```
// Set timeout
tv.tv_sec = 0;
tv.tv_usec = TIMEOUT;
if (setsockopt(sd, SOL_SOCKET, SO_RCVTIMEO,&tv,sizeof(tv)) < 0) {
  perror("Error");
}
```

타이머를 0.4초로 set 한다.

```c
// Send data
while(1){
  // Reset pakcet buffer
  memset(&curpacket, 0, sizeof(curpacket));

  // First Send file name to Server
  if( i == 0 ){
    curpacket.seq = htonl(0);
    curpacket.length = htonl(strlen(argv[3]));
    strcpy(curpacket.data, argv[3]);
  } else {
    memset(curpacket.data, 0, sizeof(curpacket.data));
    file_size = fread(curpacket.data, sizeof(char), sizeof(curpacket.data), fp);
    // Start at seq num 0
    curpacket.seq = htonl(i);
    curpacket.length = htonl(file_size);
  }

  sendto( sd, &curpacket, sizeof(curpacket), 0, (struct sockaddr *)&servaddr, sizeof(struct sockaddr) );
```

i == 0일 경우는 파일 이름을 전송할 때를 의미한다. 이럴 경우 seq 넘버를 0으로, 파일 제목을 데이터로 해서 전송하고, i != 0일 경우 i 값을 seq 넘버로 하여 패킷을 전송한다.

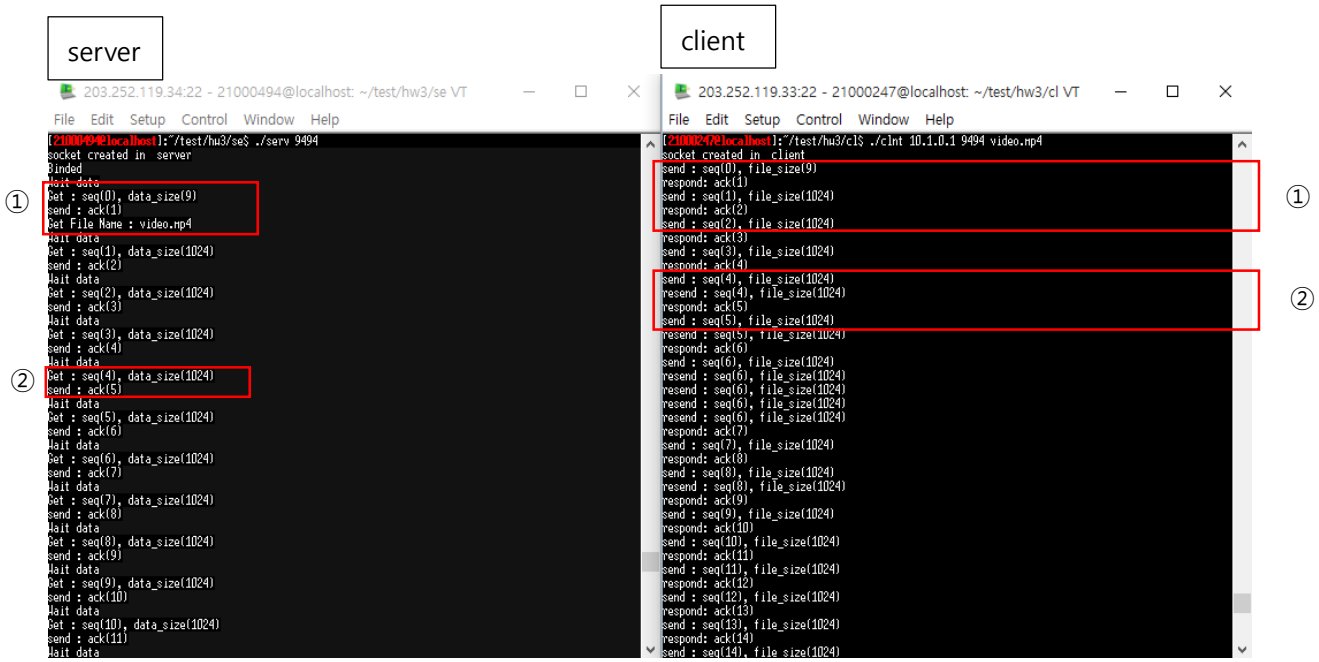
```c
  // Wait Ack Packet
  while(1){
    memset(&ackpacket, 0, sizeof(ackpacket));
    if( recvfrom( sd, &ackpacket, sizeof(ackpacket), 0, (struct sockaddr*)&servaddr, &len) < 0 ){
      // Timeout Retransmit data
      sendto( sd, &curpacket, sizeof(curpacket), 0, (struct sockaddr *)&servaddr, sizeof(struct sockaddr) );
      continue;
    }
    ackpacket.seq = ntohl(ackpacket.seq);

    // If expected ack same with next seq number, break
    if( (i+1) == ackpacket.seq ){
      break;
    }
  }

  // End of file
  if( i != 0 && file_size <= 0 ){
    break;
  }

  // Increase i
  i++;
}

//close client side connection
fclose(fp);
close(sd);
```

- 위에서 패킷을 보냈으므로 ack이 올 때까지 recvfrom함수에서 block 상태에 놓인다. Time_out이 생기면 보낸 패킷에 대한 ack을 받지 못 했으므로 curpacket을 retransmit 한다.
- Ack을 받았다면 seq 넘버와 i+1 값을 비교하여 같으면 그 다음 패킷을 보내도록 하고 다르면 다시 반복문 앞으로 가서 ack을 받는다.
- ack을 정상적으로 받았다면 i값을 1 증가 시키고 반복문의 처음으로 간다.
- 파일의 마지막이라면 반복문을 빠져 나와 종료한다.

■ Tcpdump를 하여 캡처한 내용을 통하여 자신의 프로그램이 정상적으로 동작함을 설명.

⇨ Tcpdump는 권한 때문에 사용할 수가 없어서 seq 넘버와 ack 넘버를 프린트 하여 프로그램이 정상적으로 동작하는 지 알아 보았다.

⇨ 아래의 그림은 client에서 server로 video.mp4 파일을 전송하는 과정의 일부이다.

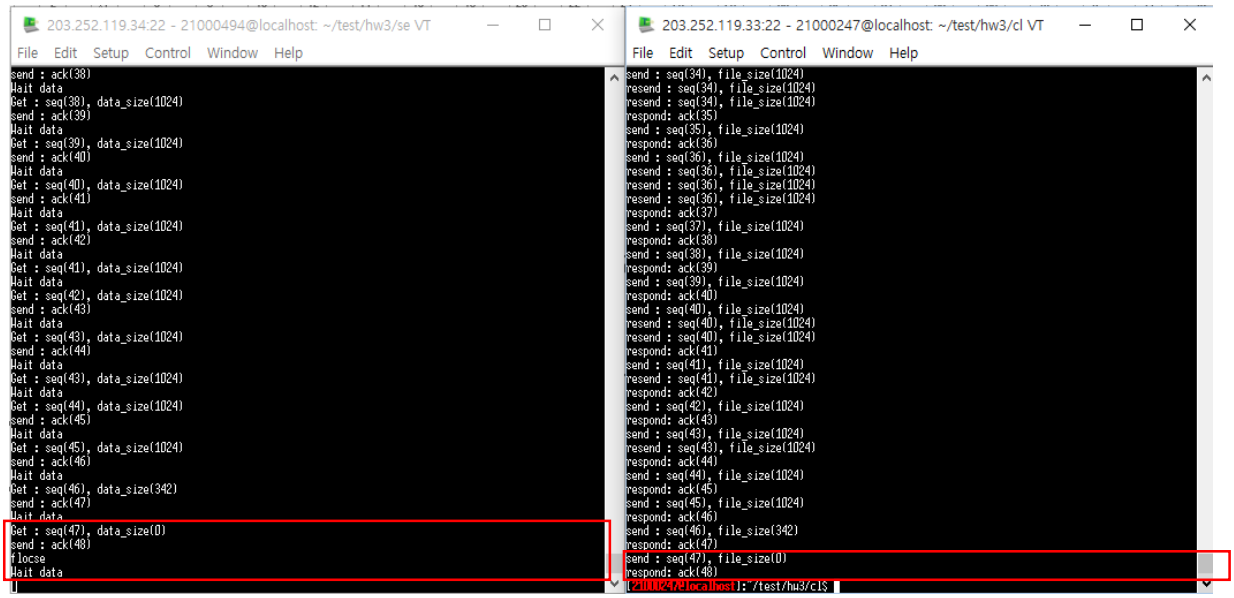


① 클라이언트 - send: seq(0), file_size(9) : 파일의 이름을 서버로 보낸다.
서버 – Get: seq(0), data_size(9) : 클라이언트에서 보낸 파일을 잘 받음
서버 – send: ack(1) : seq(1)인 데이터가 오기를 기대한다고 클라이언트에게 보냄
클라이언트 – respond: ack(1) : 서버가 보낸 ack을 받음
클라이언트 – send: seq(1), file_size(1024) : seq 넘버를 1 증가시키고 데이터와 함께 전송
⇨ 파일의 이름을 전송 받는다. 또한 패킷의 손실이 없을 때 seq넘버와 ack 넘버가 적절하게 잘 변함을 알 수 있다.

② 클라이언트 - send: seq(4), file_size(1024) : 파일 사이즈 1024바이트, seq넘버를 4로 하여 전송
클라이언트 – resend: seq(4), file_size(1024) : 정해진 시간(0.4초) 내에 ack이 오지 않아서 다시 전송
서버 – Get: seq(4), file_size(1024) : seq(4)의 패킷을 받음.
서버 – send: ack(5) : seq(4)를 잘 받았으므로 ack(5)를 보냄
클라이언트 – respond: ack(5) : 서버가 보낸 ack(5)를 받음
클라이언트 – send: seq(5), file_size(1024) : seq(5)인 데이터를 보냄
⇨ 클라이언트가 보낸 패킷에 대한 ack이 정해진 시간 내에 오지 않으면 다시 그 패킷을 보낸다. 패킷 손실에 대한 복구 처리가 가능하다.

server

client



③ ③

③ 클라이언트 - send: seq(47), file_size(0) : 파일에서 데이터를 다 읽어서 파일 사이즈가 0인 패킷을 전송한다.

서버 – resend: seq(47), file_size(0) : file_size가 0이므로 클라이언트에서의 파일 전송이 끝났음을 안다.

서버 – send: ack(48) : ack을 보냄으로써 client가 종료 될 수 있도록 한다.

서버 – fclose, wait data : 파일을 닫고 새로운 클라이언트를 기다린다.

클라이언트 – respond: ack(48) : 마지막 데이터에 대한 ack을 받았으므로 종료 한다.

⇨ 마지막에 보낸 데이터가 패킷 손실이 일어나지 않았을 때 클라이언트가 잘 종료 되며, 서버의 경우 iterative로서 잘 동작한다.