2016

임베디드 소프트웨어

TEAM PROJECT #2

목차

목표

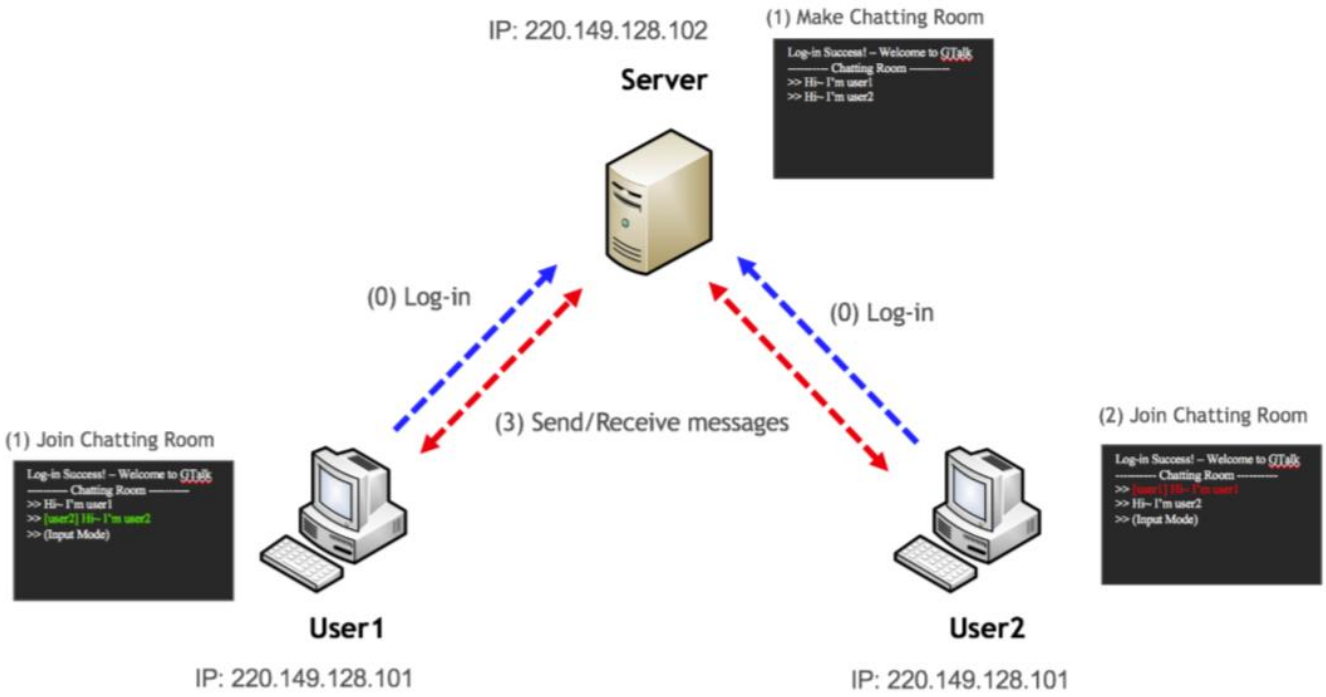


Figure 1 그룹 채팅

Project #2 는 그룹채팅 시스템을 구현하는 것이 목표이다. Figure 1 은 2 명의 사용자가 서버에 접속 하여 통신하는 그룹채팅 예제이다.

결과 코드

결과코드는 Project #2 와 Project #3 가 같은 코드를 사용한다.

저장소

본 프로젝트는 bitbucket.org 저장소를 이용해 버전관리를 하고 있으며, 주소는 다음과 같다.

- https://bitbucket.org/JunminSeo/kpu_eos_termproject

빌드

소스코드의 빌드는 터미널에서 'make' 명령을 이용해 빌드할 수 있다.

	소스 코드 빌드
1	make

Figure 2 소스 코드 빌드

기존 서버 모델의 한계

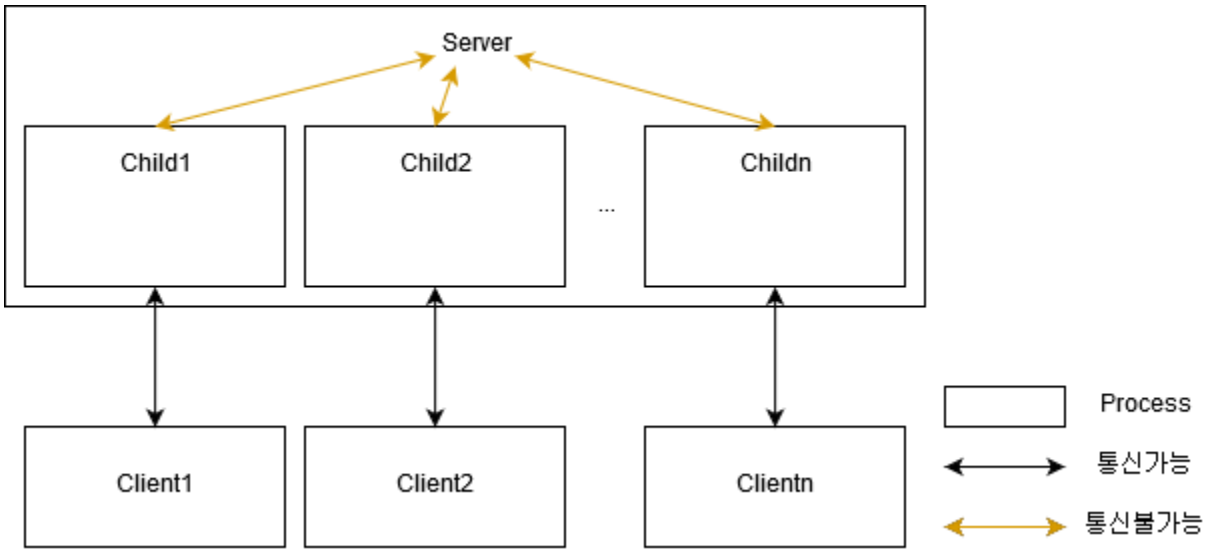


Figure 3 Multi-process Server Model

Project#1 에서 사용했던 모델은 'fork()'를 이용한 Multi-Process 서버 모델이다. 따라서 'Child'와 'Client'간 통신은 이루어 질 수 있어도 'Child'와 'Parent'간 통신은 프로세스가 다르기 때문에 이뤄질 수 없다. 또한, 클라이언트가 접속할 때마다 'fork()'로 새로운 프로세스가 만들어지기 때문에 비용이 많이 든다. 각 프로세스들 간의 통신은 'IPC(Inter Process Communication)'을 이용하면 할 수 있지만 코드의 복잡도가 늘어난다.

이에 따라 팀은 기존 서버 모델이 아닌 다른 대안을 찾기로 했고, 이와 관련된 사항을 다음 장부터 정리했다.

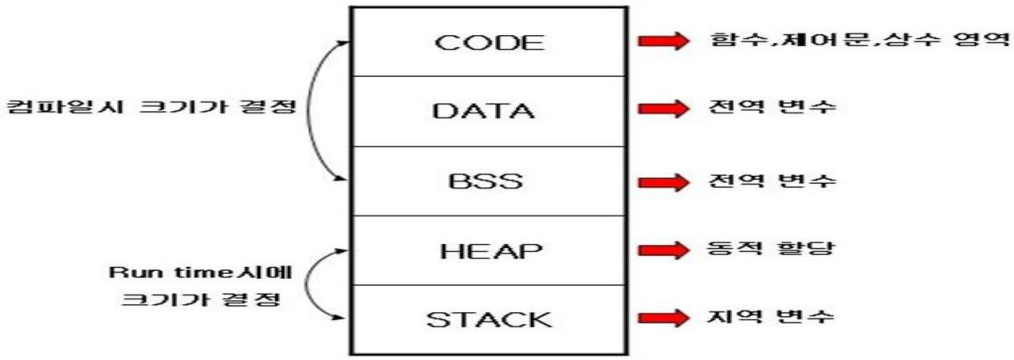
주요 개념

팀에서 고려했던 모델에 대해 살펴보기 전, 몇 가지 중요한 개념을 짚고 넘어가도록 하겠다.

Process / Thread (구현회, 2016)

프로세스는 완벽히 독립적이라 메모리 영역을 다른 프로세스와 공유하지 않는다. 쓰레드는 프로세스의 직접 실행 정보를 제외한 나머지 프로세스 관리 정보를 공유한다.

프로세는 Window 에서 예를 들면 여러 개의 독립된 프로그램을 실행하기 위해 Multi Process 방법을 이용하여 게임, 인터넷, 음악 등 다양한 프로그램을 동시 실행하는 것을 의미한다.



쓰레드는 프로세스 내에서 생성되는 하나의 실행 주체를 의미하며, Stack 을 제외한 메모리 영역을 말한다. Stack 을 모든 Thread 에 할당하여 독립적인 실행 흐름이 가능하다. CODE 영역을 공유하여 함수를 공유하고 HEAP 영역을 공유하여 변수를 공유한다.

IO 모델(IO Model) (Shichao, 2015)

블록킹 IO 모델(Blocking IO Model)

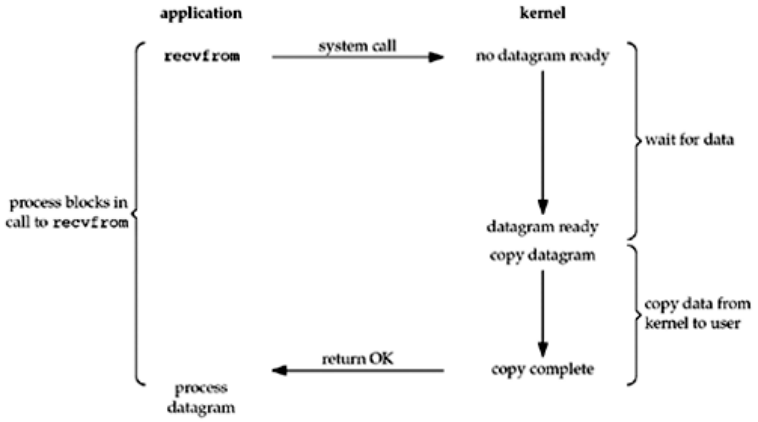


Figure 4 Blocking IO Model

우리가 C 언어에서 흔히 사용하는 'scanf()' 같은 함수를 통해 IO 를 수행하는 것을 Blocking IO 라고 한다. 즉, 사용자가 입력이 끝날 때까지 프로세스가 정지해 있는 것을 의미한다. 이를 조금 시스템의 관점에서 보면 application 은 IO 를 위해 시스템 콜을 보낸다. 그러면 커널은 데이터가 들어올 때까지 프로세스를 정지시킨다. 데이터가 다 들어오면 커널은 데이터를 복사하고 프로세스를 재개한다. IO 작업을 하는 동안 프로세스는 계속 정지해 있어야 하기 때문에 비효율적이다. 하지만 진행하는 흐름이 단순해서 구현하기 쉽고, 꽤 많은 경우 Blocking IO 만으로도 처리할 수 있기 때문에 빠르게 적용할 수 있는 장점이 있다.

넌블로킹 IO 모델(Non-blocking IO Model)

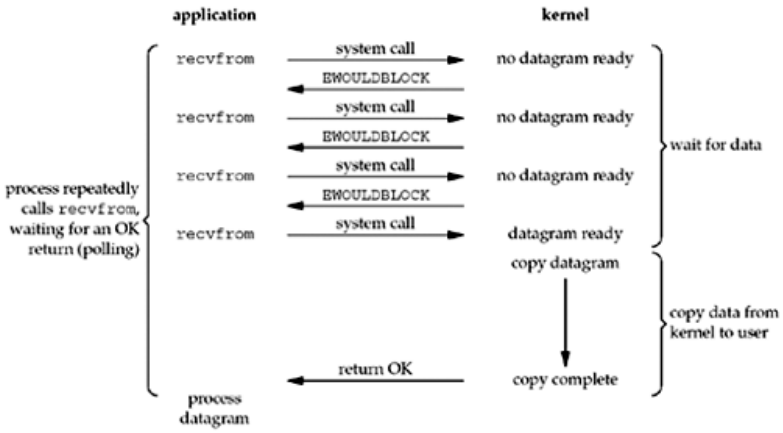


Figure 5 Non-blocking IO Model

반면, Non-blocking IO 모델은 어플리케이션이 IO 요청을 보내면 즉시 응답을 준다. 데이터가 없을 수도 있는데 이 역시 '없다'라는 응답을 프로세스에 전달한다. 그러면 프로세스는 다른 작업을 하다 또 IO 요청을

하게 되고 데이터가 있으면 커널은 데이터를 복사해서 프로세스에 전달한다. 일련의 작업 동안 프로세스는 정지하지 않기 때문에 다른 일을 수행할 수 있다. 하지만 프로세스는 IO 요청이 완료되었는지에 대한 여부를 지속적으로 검사해야 하기 때문에 프로그램의 흐름이 복잡해질 수 있다.

IO 다중화 모델(IO Multiplexing Model)

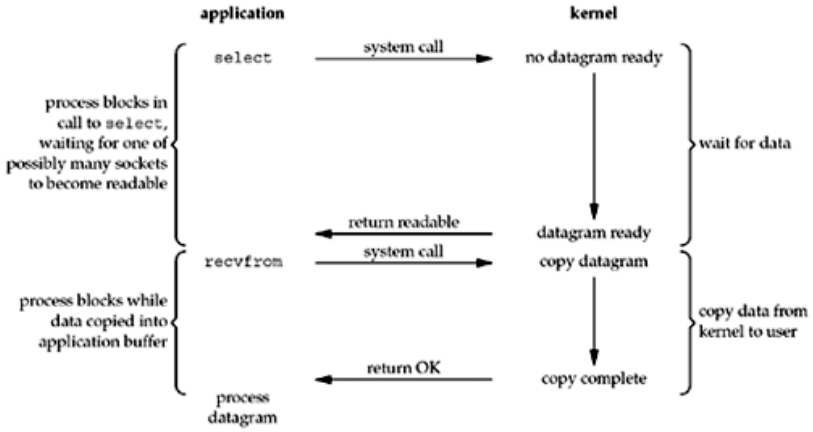


Figure 6 IO Multiplexing Model

IO Multiplexing 모델은 'select'와 'poll'라고 할 수 있는데, 여러 파일 디스크립터를 묶어 폴링하는 방식이다. 이 역시 Blocking IO 로 볼 수도 있으나 Blocking IO 의 경우 모든 IO 시스템 콜이 IO 작업을 위해 프로세스를 정지했던 것과 달리 IO Multiplexing 모델은 'select'로 대표되는 하나의 시스템 콜이 IO 작업을 위해 프로세스를 정지하고 있다가 'select'에 있는 파일 디스크립터의 IO 가 완료되면 바로 프로세스 정지를 풀고 IO 를 수행하는 방법이다. 이 방식을 이용하면 싱글 스레드로도 여러 IO 를 동시에 처리할 수 있다. 이 방식은 Timeout 이 있으면 Non-blocking IO 모델처럼 동작하고 Timeout 이 없으면 Blocking IO 모델처럼 동작한다.

Epoll

Epoll 은 2002 년 Linux 커널에 추가된 기능으로 종전에 쓰이던 'select', 'poll'보다 더 개선된 방식이다. 'select'의 경우 fd 가 비트마스크라 최대 1024 개까지 등록할 수 있고, 이벤트를 감지하는 방식도 모든 파일 디스크립터를 검사하는 방식이라 비효율적이다. 'poll' 역시 순차적으로 모든 디스크립터를 검사하는 방식이다. 'epoll'을 이용한 기법에서는 관찰해야 할 파일 디스크립터와 그 디스크립터로부터 관찰해야 할 event 의 종류를 'epoll 인스턴스'에 저장할 수 있다. 이 'epoll 인스턴스'에 등록된 파일 디스크립터들은 운영체제에 넘겨져 저장되며, 이후에 프로그램이 epoll_wait 함수를 호출해 대기하고 있으면 등록된 event 가 발생할 때 운영체제가 해당 파일 디스크립터를 프로세스로 넘겨준다. 'select' 기법에서 직접 파일 디스크립터 배열을 검사하며 event 를 찾아내야 했던 것과는 다르게, 'epoll'기법에서는 모든 계획을 운영체제에 미리 전달한 뒤 'epoll_wait()' 함수를 통해 기다리기만 하면 된다.

서버 모델

위의 개념들을 이용해 몇 가지 서버 모델을 디자인 했으며, 결과는 아래와 같다. 소켓 통신에 있어 TX, RX 는 모두 쓰레드나 프로세스가 아닌 IO 다중화로 처리를 하여 효율성을 높였다.

싱글 쓰레드 서버 모델(Single-thread server model)

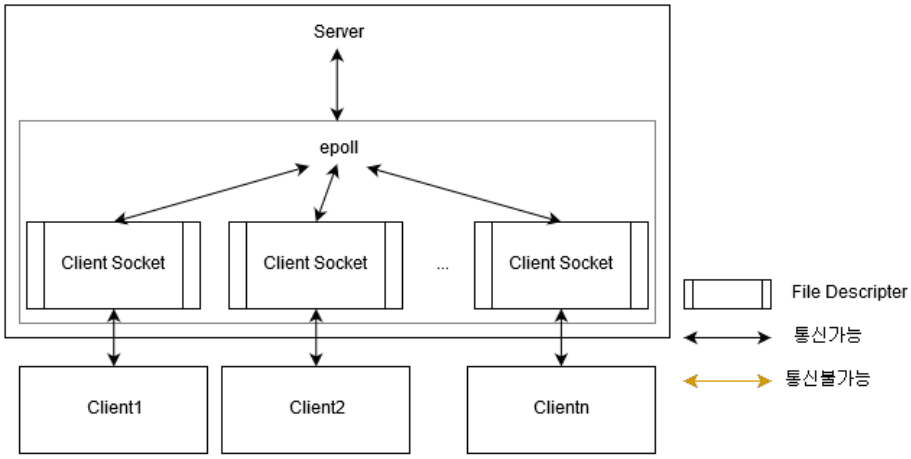


Figure 7 Single-thread Server Model

첫 번째로 고려했던 모델은 Single-thread server model 이다. 모든 IO 는 'epoll'을 통해 동기적으로 이뤄진다. 이러한 모델의 장점은 프로세스를 생성하거나 쓰레드를 생성하는데 드는 비용이 전혀 없다. 또한 데이터를 교환하기 위하여 추가적인 IPC 설비가 필요치 않고, 쓰레드를 사용했을 때 생길 수 있는 동기화 문제가 없다. 따라서 다른 모델보다 훨씬 좋은 퍼포먼스를 보인다. 단점으로는 클라이언트의 수가 늘어나면 늘어날 수록 'epoll'이 모니터링 해야 하는 이벤트의 수도 늘어나게 된다. 따라서 polling 방식의 'epoll'에서 병목 현상이 일어나 퍼포먼스에 심각한 문제가 생길 수 있다. 하지만 본 프로젝트에서 감당해야할 클라이언트의 수가 그리 많지 않기 때문에 이 모델이 다른 모델들 보다 퍼포먼스 면에서 우수하다.

다중 쓰레드 서버 모델(Muilti-thread server model)

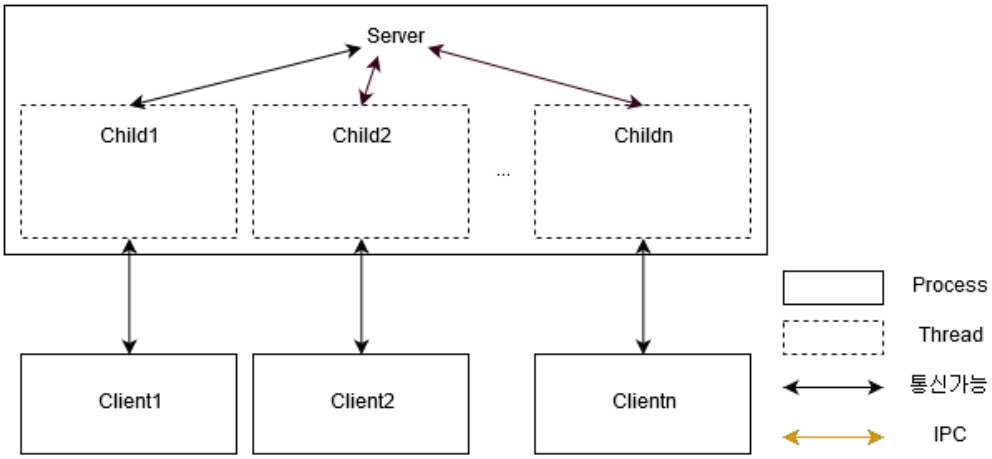


Figure 8 Multi-thread Server Model

다중 쓰레드 서버 모델의 경우에는 구성 자체는 다중 프로세스 서버 모델과 비슷하다. 차이점이 있다면 다중 프로세스가 아니라 다중 쓰레드로 동작한다는 것인데, 이는 프로세스가 나뉘는 것이 아니라 내부에 데이터 교환이 자유롭다는 장점이 있다. 추가적으로 쓰레드는 경량 프로세스 이므로 비용이 프로세스를 생성하는 것보다 적게 든다. 단점으로는 TLB 와 같은 하드웨어 리소스를 공유할 때, 동기화 문제가 생길 수 있어 주의를 요한다. 또한 단일 흐름이 아니기 때문에 디버깅이 복잡해 지고, 쓰레드에 문제가 생겼을 때 프로세스 전체에 영향을 미친다.

IPC 를 포함한 다중 프로세스 서버 모델(Multi-process server model with IPC)

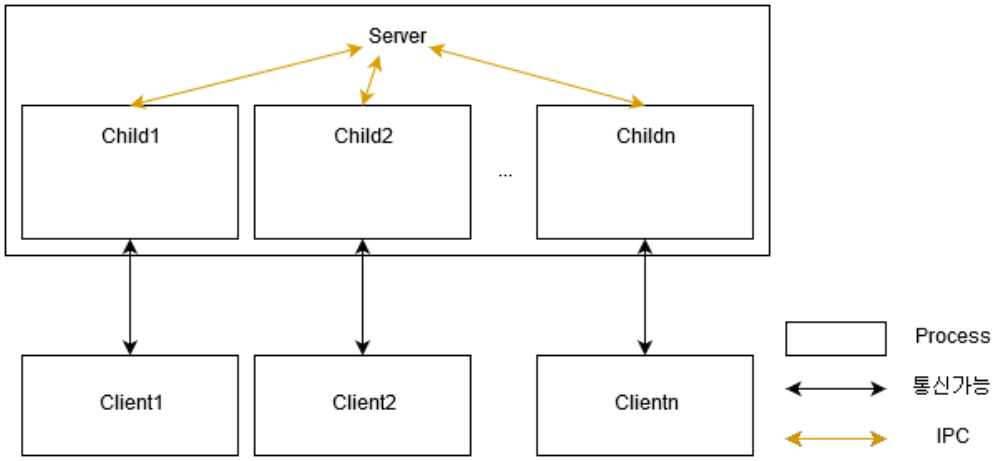


Figure 9 Multi-process Server Model with IPC

IPC 가 있는 다중 프로세스 서버 모델은 기존 'fork()'를 이용한 모델의 한계인 부모와 자식간 통신을 IPC 로 극복한 모델이다. 이 모델은 기존 Project #1 에서 구현한 프로그램에 IPC 설비만 추가하면 된다는 장점이

있으나, IPC 설비에 다중 리시버, 브로드 캐스트 지원이 복잡하다. 또한 클라이언트를 추가할 때마다 'fork()'로 새로운 프로세스를 생성해야 하므로 비용이 너무 커 본 프로젝트에서는 채택하지 않았다.

클라이언트 모델

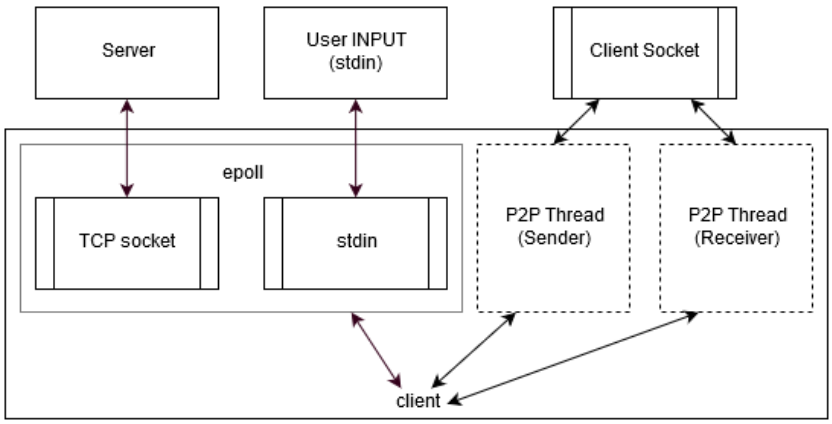


Figure 10 Client model

클라이언트 모델은 기본적으로 'epoll'을 이용한 IO 다중화를 적용했다. 따라서 사용자로부터 입력을 받는 프로세스가 TCP 소켓과 서버와 통신을 블록하지 않는다. 사용자는 이를 통해 메시지의 입력과 서버에서 메시지를 받는 것을 동시에 할 수 있다. 또한 P2P 방식으로 파일을 주고 받을 때, 싱글 쓰레드로 이를 구현하게 되면 너무 복잡해 지므로 쓰레드로 분리해 구현하였다.

클라이언트 정보 추적

서버에는 다수의 클라이언트가 접속해 있고, 이 클라이언트들의 상태는 모두 제각각 이므로(ex. 사용자 A 는 로그인 상태에 있고, 사용자 B 는 로그인 성공 상태에 있음) 이를 추적해야 할 필요성이 있다. 이에 클라이언트 정보를 위한 구조체를 선언하였고, 이를 조작하기 위한 메소드들을 정의하였다.

clicnth.h

```
1  #define MAX_CLIENT 5
2
3  struct _client_info {
4      cli_state_t state;
5      int sock_fd;
6      char username[255];
7      char ipaddr[255];
8  } _client_info;
9
10 typedef struct _client_t {
11     int idx;
12     struct _client_info cli[MAX_CLIENT];
13 } client_t;
14
15 extern int init_client_pool(client_t*);
16 extern int add_client(client_t*, int);
17 extern int del_client(client_t*, int);
18 extern int set_client_username(client_t*, int, char*, size_t);
19 extern int set_client_ipaddr(client_t*, int, char*, size_t);
20 extern int set_client_state(client_t*, int, cli_state_t);
21 extern int get_client_username(const client_t*, int, char*, size_t);
22 extern int get_num_client(const client_t*);
23 extern cli_state_t get_client_state(const client_t*, int);
24
25 static int query_client(const client_t*, int);
```

Figure 11 clicntl.h

'init_client_pool()'은 'client_t'로 대표되는 구조체를 초기화 하기 위한 메소드 이다.

'add_client()'는 'client_t' 구조체에 사용자를 추가하는 메소드로 'MAX_CLIENT'가 넘어가면 사용자 추가를 중단하고 -1 를 리턴한다.

'del_client()'는 'client_t'구조체에 있는 사용자를 제거하는데, 로그인 에러가 발생했을 시나 사용자가 접속을 종료했을 때와 같이 더 이상 사용자를 추적해야 할 필요가 없을 때 호출한다.

'set_client_username()' 메소드는 'client_t' 구조체에 사용자 이름란을 체운다. 서버에서 전체 클라이언트에게 브로드 캐스트를 할 때, 사용자 이름을 붙여 넣기 위함이다.

'set_client_ipaddr()' 메소드는 'client_t' 구조체에 접속한 클라이언트의 ip 주소를 기록한다.

'set_client_state()'는 현재 접속한 클라이언트의 상태를 설정하기 위한 메소드이다.

‘get_client_username()’는 호출한 소켓에 맞는 사용자 이름을 ‘char *’에 기록한다.

본 서버에 접속한 클라이언트의 상태를 추적하기 위해 enum 을 선언했다.

	clicntl.h
1	`typedef enum _cli_state_t {`
2	`    CLIENT_ERROR = 0, CLIENT_AUTH, CLIENT_GRANT, CLIENT_DENIED`
3	`} cli_state_t;`

Figure 12 Client state enum

‘CLIENT_ERROR’은 여러 이유로 인하여 사용자를 추적하는데 문제가 생겼을 때, 설정하기 위해 정의했다.

‘CLIENT_AUTH’는 현재 사용자가 인증을 받지 않은 상태임을 나타낸다. 따라서 이 상태의 사용자는 인증 과정을 거쳐야 하며, 인증이 불허되면 사용자 풀에서 지워지고 소켓은 닫히게 된다.

‘CLIENT_GRANT’는 사용자가 인증을 완료하고, 채팅 룸에 입장한 상태이다. 이 상태의 사용자는 상호 메시지를 교환할 수 있다.

‘CLIENT_DENIED’는 인증에 실패 했을 때 갖는 상태인데, ‘CLIENT_AUTH’ 상태에서 액세스가 거부되면 사용자 풀에서 사용자가 지워지고 소켓이 닫히므로 사용하지는 않는다.

통신 프로토콜 정의

본 서버에는 추후 파일 교환기능도 추가해야 하므로 일반 메시지와 파일 전송과 같은 메시지를 구분해야 할 필요가 있었다. 이에 따라 다음과 같은 프레임을 갖는 프로토콜을 정의하였다.

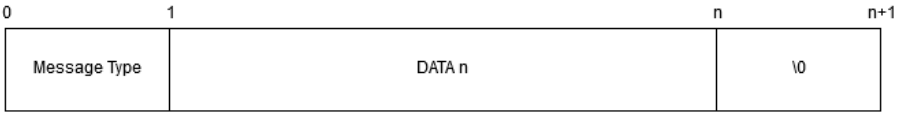


Figure 13 통신 프로토콜 프레임

첫 번째 옥텟(Octet)은 메시지 타입을 나타내며 이는 다음과 같은 enum 으로 정의된다.

Enum _msg_type

```
1  enum _msg_type {
2      MSG_CHAT = 0, MSG_CRED, MSG_FILE, MSG_CLOSE
3  };
```

Figure 14 메시지 타입

그 다음 필요한 n 개의 메시지를 넣고 마지막에 널 문자('\0')로 메시지의 끝을 나타냈다. 따라서 읽는 측에서는 널 문자('\0')가 들어올 때까지 메시지를 읽고 맨 앞의 1 옥텟(Octet)을 이용해 메시지의 종류를 판독하면 된다. 일반적인 통신 프로토콜이라면 메시지의 무결성을 확인하기 위하여 데이터의 개수와 CRC 등을 넣어야 하지만 구현의 편의성을 위해 생략하였다.

프로토콜에 맞는 메시지를 손 쉽게 보내기 위하여 몇 가지 메소드들을 정의했다.

msg.h

```
1  extern ssize_t sendMsg(int, const char*, size_t);
2  extern ssize_t sendCred(int, const char*, size_t, const char*, size_t);
3  extern ssize_t sendBye(int);
```

Figure 15 msg.h

먼저 'sendMsg()'는 일반 메시지를 보내기 위한 메소드이다. 여기에 소켓 파일 디스크립터와 보낼 메시지 그리고 메시지의 길이를 넣으면 소켓 파일 디스크립터에 일반 메시지를 보내게 된다.

그 다음 'sendCred()'는 로그인 정보를 보내기 위한 메소드이다.

'sendBye()'는 연결 종료를 보내기 위한 메소드이다.

모든 메소드는 성공 시, 보낸 메시지 전체 길이를 리턴하고 실패 시, -1 를 리턴한다. 'size_t'는 unsigned 형이라 -1 를 리턴할 수 없으므로 signed 형인 'ssize_t'를 리턴한다.

각 메시지 별 세부 규약은 다음과 같다.

일반 메시지

Figure 16 일반 메시지 프레임

일반 메시지는 1 옥텟(Octet)의 메시지 타입 다음에 바로 메시지가 온다. 그 후, 널문자('\0')로 메시지를 종료한다.

로그인 정보

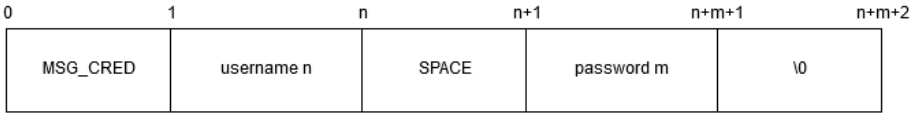


Figure 17 로그인 정보 프레임

로그인 정보를 알리는 메시지는 1 옥텟(Octet)의 메시지 타입 다음에 사용자 이름과 암호를 넣는데 이는 사이에 공백 문자를 둬 구분한다. 마지막으로 널 문자('\0')로 마무리 한다.

파일 전송

파일 전송은 Project #3 에서 구현할 내용으로 Draft 수준으로 남겨뒀다.

연결 종료

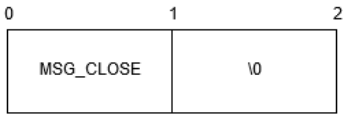


Figure 18 연결 종료 프레임

연결 종료 메시지는 1 옥텟(Octet)의 메시지 타입과 널 문자('\0')만을 보내는 간단한 메시지이다. 서버나 클라이언트가 이 메시지를 받으면 즉시 연결을 종료한다.

IO 다중화 제어

본 서버는 IO 다중화를 제어하기 위해 다음과 같은 메소드들을 정의했다.

epollcntl.h

```
1  extern int init_epoll(int);
2  extern int add_epoll_event(int, int);
3  extern int set_socket_nonblock(int);
```

Figure 19 epollcntl.h

'init_epoll()' 메소드는 epoll 을 초기화 하며, epoll 디스크립터를 반환한다.

'add_epoll_event()'는 epoll 에 소켓 이벤트를 등록하기 위한 메소드로 성공 시 0 을 반환하고, 실패 하면 -1 을 반환한다.

'set_socket_nonblock()' 는 기본적으로 Tcp 소켓이 블록킹 IO 모드로 동작하기 때문에 이를 넌 블록킹 IO 모드로 동작하게 해줄 필요가 있기에, Tcp 소켓을 넌블록킹 IO 모드로 동작하게 바꿔준다.

명령어

모든 명령은 '/'문자로 시작하여 작성하고, 명령과 파라미터를 구분하기 위하여 공백문자(' ')를 사용한다. 클라이언트에서 지원하는 명령은 다음과 같다. 파일 명령의 경우에는 Project #3 을 위해 미리 정의가 되어 있고 구현되어 있지는 않다.

Table 1 명령어 리스트

명령어	파라미터	설명
file	<파일 이름>	파일 전송 (구현되지 않음. Project #3 에서 구현예정)
bye	N/A	접속 종료

사용예

접속종료

```
1  /bye
```

Figure 20 접속 종료 명령어 사용 예

접속 종료를 할 때, 클라이언트에서 '/bye' 명령을 입력하면 안전하게 접속을 종료할 수 있다.

사용자로부터 입력 받은 명령어를 처리하기 위한 루틴은 다음과 같다.

client.c::main()::100

```
1  if (iscmd(buf, strlen(buf))) {
2      char *pbuf = strtok(buf, " ");
3
4      // Close command
```

```
5       if (strncmp(pbuf + 1, msg_cmd[MSG_CLOSE], strlen(msg_cmd[MSG_CLOSE])) == 0) {
6           sendBye(sock_fd);
7           close(sock_fd);
8           goto SUCCESS;
9       }
10      else if (strncmp(pbuf + 1, msg_cmd[MSG_FILE], strlen(msg_cmd[MSG_FILE])) == 0) {
11          printf("this is file command %s\n", pbuf + 1);
12      }
13      else {
14          printf("Unknown command %s\n", pbuf + 1);
15      }
16  }
17  else {
18      if (sendMsg(sock_fd, buf, strlen(buf) + 1) == -1) {
19          perror("Failed send()");
20          exit(EXIT_FAILURE);
21      }
22  }
```

Figure 21 명령어 처리 루틴

먼저 공백 문자가 명령어를 구분하는 인자이므로 'strtok()'를 이용해 띄어쓰기를 기준으로 입력받은 문장을 자른다. '/' 문자는 명령이라는 것을 구분하기 위한 구분자 역할만을 수행하므로 명령을 비교하기 위한 'strncmp()'함수에는 'pbuffer + 1'이 들어간다.

프로세스 흐름

서버

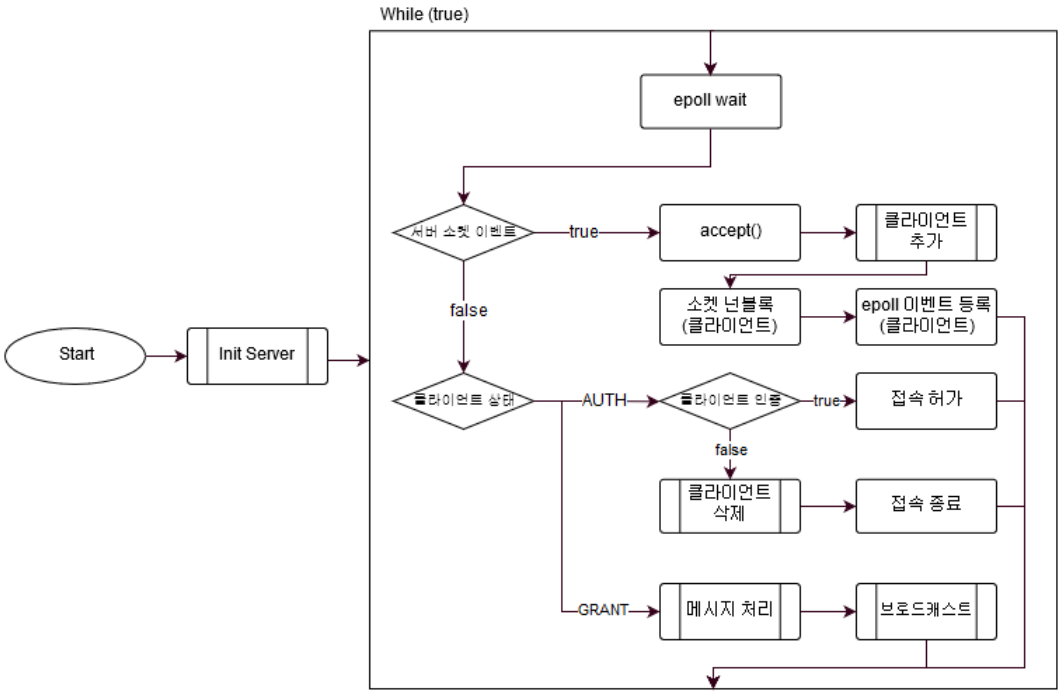


Figure 22 서버 흐름도

서버는 프로그램이 시작되면 초기화를 수행한다. 이 과정을 통해 서버 접속용 소켓도 'listen()'상태로 만들고 epoll 도 초기화 하며 서버 소켓을 epoll 의 이벤트로 등록한다. 그 다음 서버는 무한 루프를 통해 계속 'epoll_wait()'에서 대기하며 서버 소켓이나 클라이언트 소켓 이벤트가 발생하길 기다린다. 서버 소켓 이벤트가 발생하면 'accept()'를 수행해서 연결을 확립하고 'client_t' 구조체에 클라이언트 정보를 추가한다. 그리고 추가된 클라이언트 소켓을 epoll 이벤트에 등록하여 클라이언트로부터의 IO 가 epoll 에 의해 관리될 수 있도록 한다.

클라이언트 소켓 이벤트가 발생하면 클라이언트 상태를 확인한다. 클라이언트 상태가 인증이 필요한 상태인 AUTH 라면 인증을 수행하고, 접속 허가 혹은 불허를 알린다. 접속 허가일 경우에는 클라이언트 상태를 GRANT 로 세팅하고, 접속 불허 일 경우에는 클라이언트를 클라이언트 풀에서 지우고 소켓을 닫는다. 소켓이 닫히게 되면 epoll 이벤트에서도 자동으로 제거된다. 만약 클라이언트 상태가 접속 허가 상태라면 메시지를 처리하게 되는데, 메시지 종류에 따라 조금 다른 반응을 나타낸다. 메시지가 일반 메시지라면 브로드 캐스트를 하게 되고, 메시지가 접속 종료를 알리는 메시지라면 클라이언트 풀에서 사용자를 지우고 소켓을 닫는다. 서버는 이 과정을 프로그램이 종료될 때까지 반복한다.

클라이언트

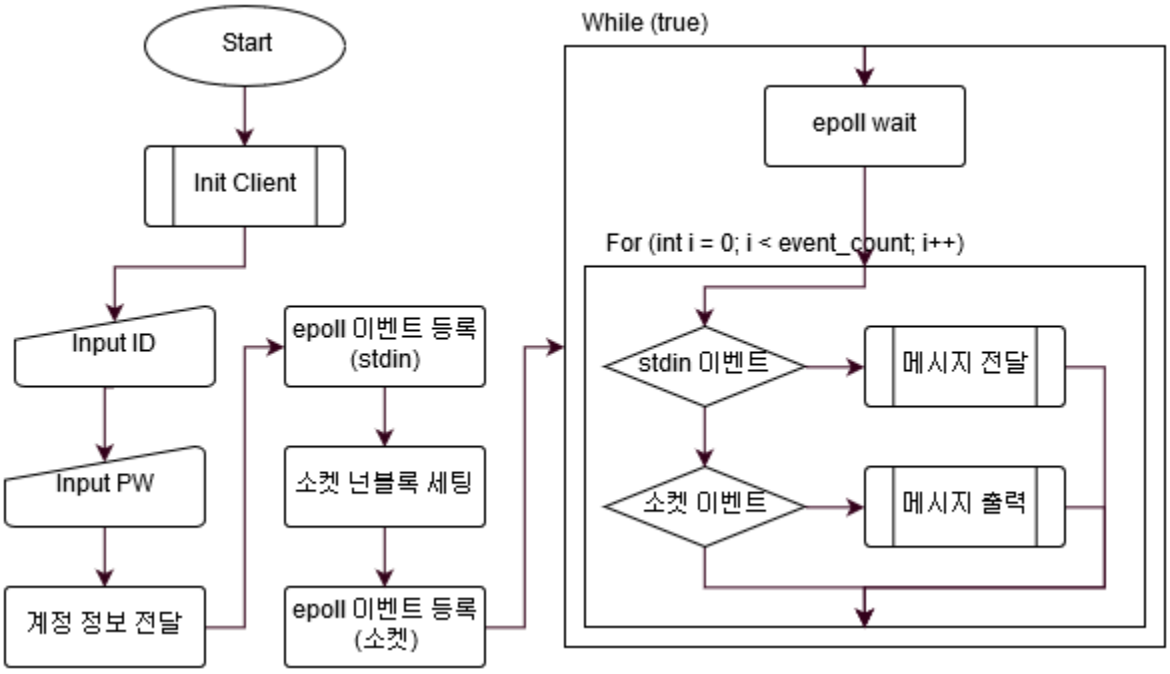


Figure 23 클라이언트 흐름도

클라이언트는 시작되면 소켓을 초기화 하고 접속을 시도한다. 접속이 완료되면, 사용자로부터 아이디와 패스워드를 입력 받는데 이 과정은 블록킹 IO 로 진행된다. 그 후, 계정 정보를 전달하고 소켓과 표준 입력을 epoll 이벤트에 등록한다. 이 때부터 모든 소켓은 넌 블록킹 IO 로 동작하고 모든 IO 는 epoll 의 이벤트를 검출하여 처리한다. 표준 입력으로부터 이벤트가 발생하면 각 명령에 맞는 메시지를 서버에게 보낸다. 소켓 이벤트가 발생하면 메시지를 사용자에게 출력한다.

동작 결과

서버에 접속하면 먼저 ID 와 PW 를 입력 받고, 로그인에 성공하면 서로 메시지를 주고 받을 수 있다.

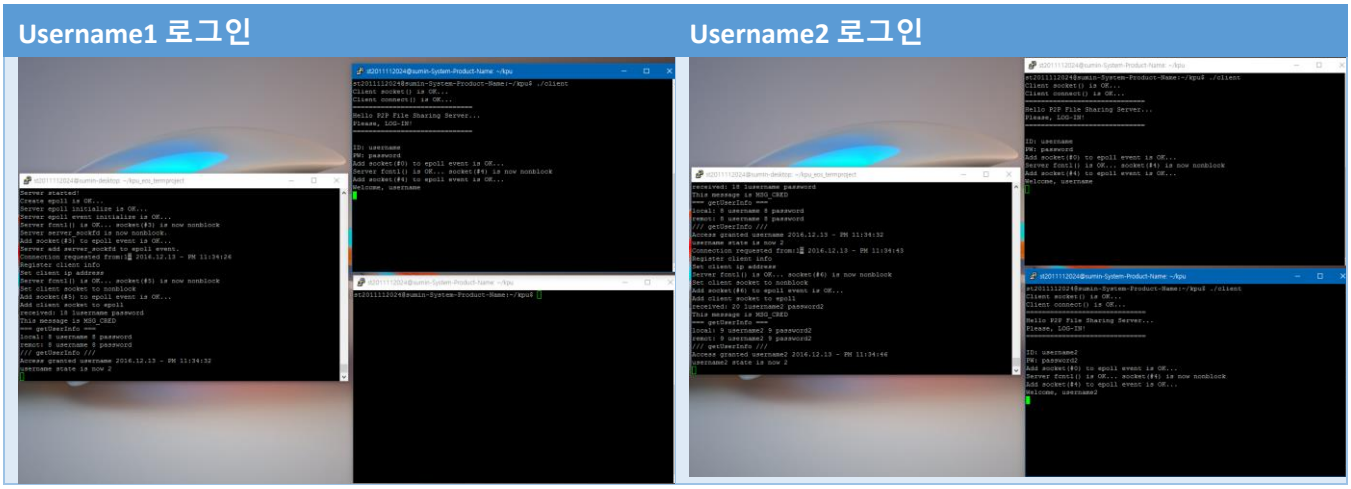


Figure 24 로그인 화면

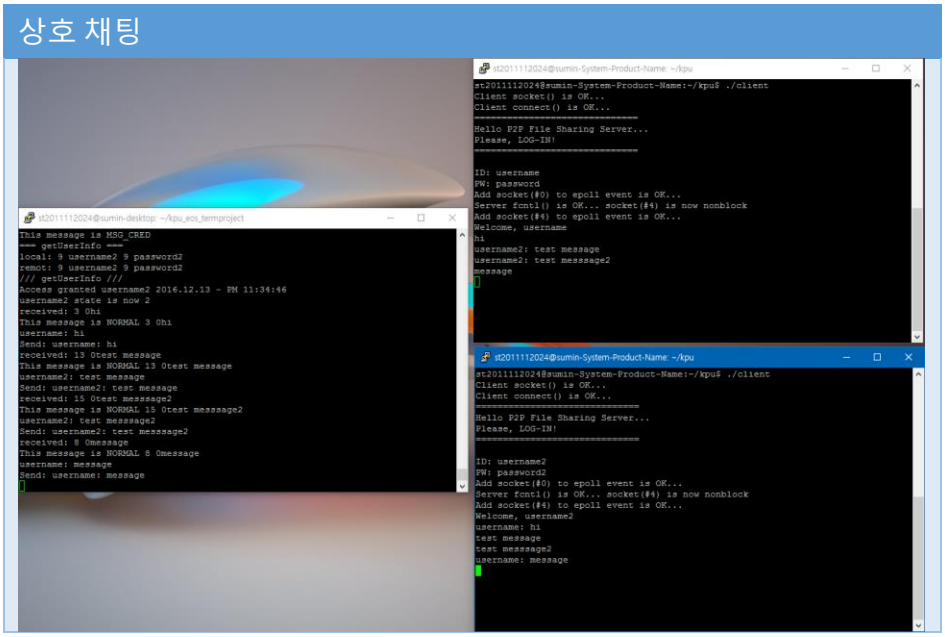


Figure 25 채팅 화면

인용 자료

Shichao. (2015 년 8 월 17 일). "I/O Multiplexing: The select and poll function." Shichao's Notes: https://notes.shichao.io/unp/ch6/에서 검색됨

yundream. (2011 년 4 월 13 일). "epoll." joinc: http://www.joinc.co.kr/w/Site/Network_Programing/AdvancedComm/epoll24#s-1 에서 검색됨

구현회. (2016). "운영체제." 한빛아카데미.

김승현. (2013 년 08 월 22 일). "epoll 설명." BlueHeart's Cabin: http://blueheartscabin.blogspot.kr/2013/08/c-epoll.html 에서 검색됨

2016

임베디드 소프트웨어

TEAM PROJECT #3

목차

목표

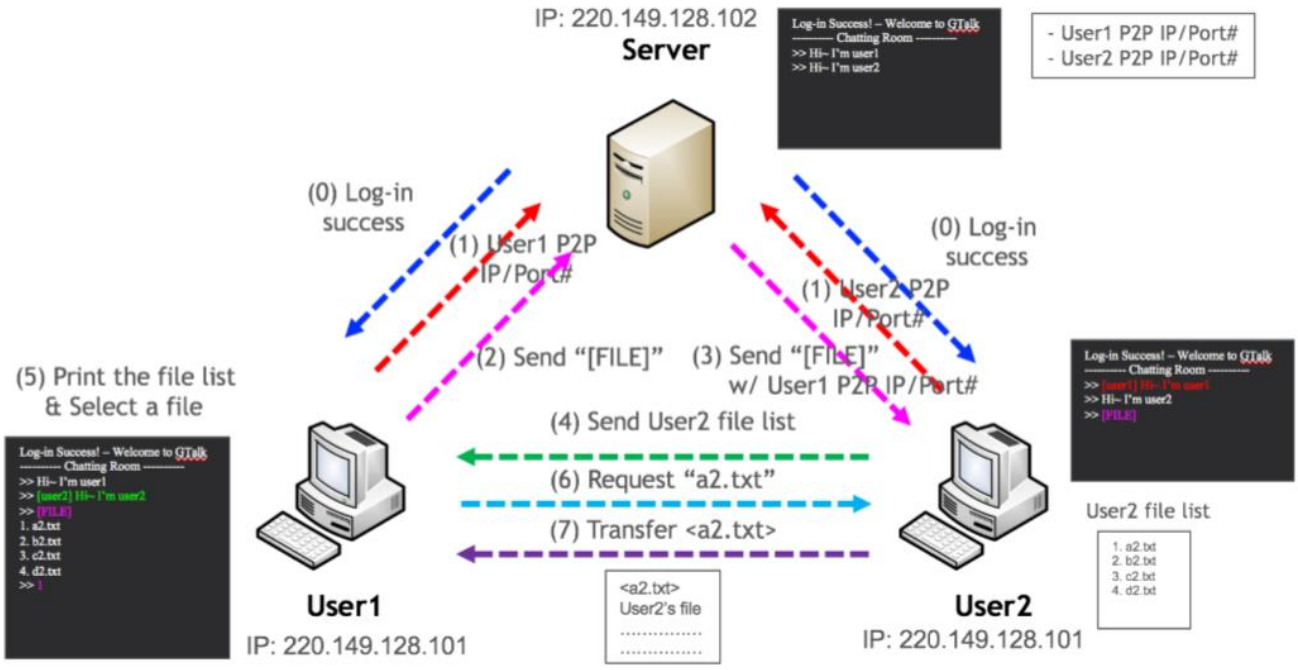


Figure 1 P2P 파일 전송

Project #3 는 서버가 중계하는 그룹채팅 시스템에 P2P 파일전송 기능을 추가하는 것이다.

결과 코드

결과코드는 Project #2 와 Project #3 가 같은 코드를 사용한다.

저장소

본 프로젝트는 bitbucket.org 저장소를 이용해 버전관리를 하고 있으며, 주소는 다음과 같다.

- https://bitbucket.org/JunminSeo/kpu_eos_termproject

빌드

소스코드의 빌드는 터미널에서 'make' 명령을 이용해 빌드할 수 있다.

	소스 코드 빌드
1	make

Figure 2 소스 코드 빌드

통신 프로토콜

기존 Project #2 에서 사용한 프로토콜에서 P2P 파일 전송 기능을 지원 하기 위하여 추가적인 프레임을 정의하였다.

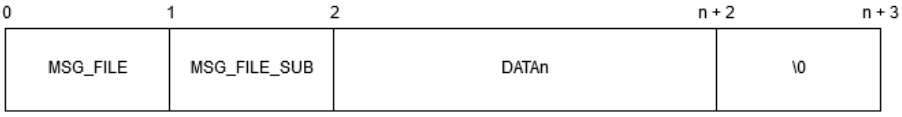


Figure 3 파일 통신 프로토콜 프레임

기존 프로토콜과 거의 흡사하지만 추가적인 상태를 구분하기 위해 1 옥텟(Octet)의 타입 구분자를 추가하였다. 따라서 첫 번째 옥텟(Octet)은 파일 메시지라는 것을 알리는 'MSG_FILE'이 들어가고 그 다음 옥텟(Octet)은 현재 수행되고 있는 파일 명령을 나타낸다. 지원하는 파일 명령은 다음과 같다.

	Enum _msg_type_file
1	`enum _msg_type_file {`
2	`    MSG_FILE_LS_REQ = 0, MSG_FILE_LS_START, MSG_FILE_LS_BODY, MSG_FILE_LS_END,`
3	`    MSG_FILE_DN_REQ, MSG_FILE_DN_ACK, MSG_FILE_DN_NACK`
4	`};`

Figure 4 파일 메시지 타입

파일 리스트

파일 리스트를 조회하기 위해 3 가지의 프레임을 정의하였다. 파일 리스트의 요청과 파일 리스트임을 알리는 시작, 실제적인 파일 리스트를 포함하는 본문, 파일 리스트의 끝을 알리는 끝 프레임이다.

파일 리스트 요청

Figure 5 파일 리스트 요청 프레임

파일 리스트를 요청하기 위한 프레임은 메시지 타입과 파일 메시지 타입 이외에 추가적으로 사용자 이름 정보가 들어간다. 이 정보는 클라이언트에서 서버로 전송 할 때는 수신자 이름이 들어가고 서버에서 클라이언트로 전송할 때는 발신자 이름이 들어간다. 프레임에 수/발신자 이름을 넣은 이유는 구현을 한 이유는 서버에서 추가적으로 수신자와 발신자의 상태를 추적하지 않고 메시지를 이용해 처리하기 위함이다.

파일 리스트 시작

Figure 6 파일 리스트 시작 프레임

파일 리스트 시작 프레임 역시 파일리스트 요청 프레임 파일 메시지 타입만 빼면 모두 동일하다.

파일 리스트 본문

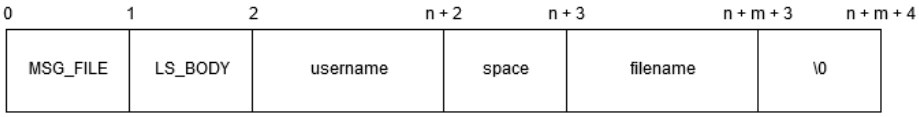


Figure 7 파일 리스트 본문 프레임

파일 리스트 본문은 파일 이름을 보내기 위한 프레임으로 메시지 타입, 파일 메시지 타입 이외에 송/수신자 이름이 들어간다. 앞의 데이터와 구분을 하기 위해 구분자로 공백문자를 사용하며, 그 뒤 데이터는 파일 이름으로 체워진다.

파일 리스트 끝

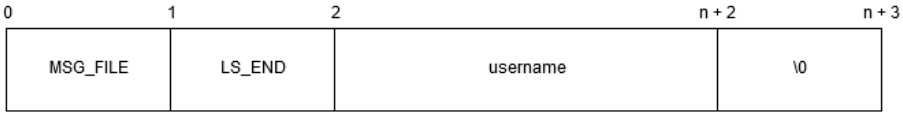


Figure 8 파일 리스트 끝 프레임

파일 리스트 끝을 나타내는 메시지는 파일리스트의 시작 메시지 파일 메시지 타입만 제외하면 모두 같다.

파일 다운로드

파일 다운로드와 관련된 프레임은 3 개를 정의하였다. 파일 다운로드의 요청과 다운로드를 승인하거나 거부하는 프레임이다.

파일 다운로드 요청

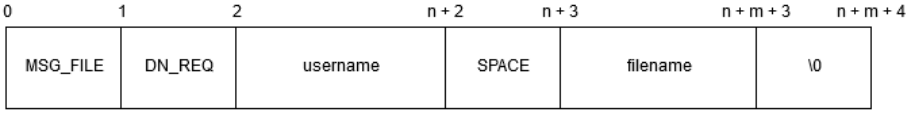


Figure 9 파일 다운로드 요청 프레임

파일 다운로드 요청 프레임에는 사용자 이름과 파일 이름이 들어간다.

파일 다운로드 승인

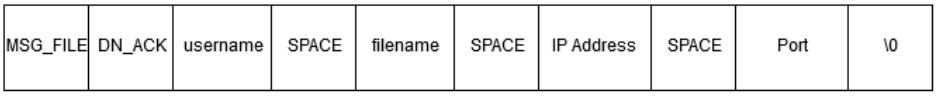


Figure 10 파일 다운로드 승인 프레임

파일 다운로드 승인 프레임에는 파일 다운로드 요청 프레임처럼 사용자 이름과 파일 이름이 들어가고 추가적으로 다른 클라이언트에서 접속할 수 있도록 IP 주소와 포트 번호가 들어간다. 이 프레임을 발송하는 사용자는 메시지를 보내기 전, 다른 사용자가 접속할 수 있도록 소켓을 열어 두어야 한다.

파일 다운로드 거부

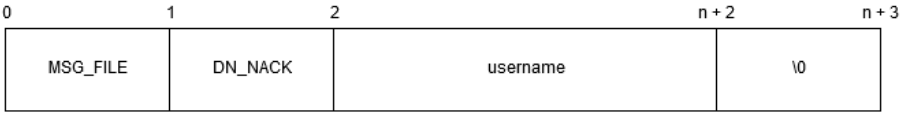


Figure 11 파일 다운로드 거부 프레임

파일이 존재하지 않거나 기타 모종의 이유로 ACK 프레임을 보낼 수 없을 경우에는 NACK 프레임을 보낸다. 에러코드를 포함하지 않기 때문에 서버에서는 NACK 메시지가 들어오면 무조건 파일이 없다고 간주하고 “File not found” 에러 메시지를 클라이언트에게 보낸다.

명령어

파일을 송수신 하기 위해 클라이언트에서 지원하는 명령은 다음과 같다.

Table 1 파일 명령어 리스트

명령어	파라미터	설명
/file	ls <사용자 이름>	<사용자 이름>이 공유한 파일의 리스트를 출력한다.
/file	dn <사용자 이름> <파일 이름>	<사용자 이름>에게서 <파일 이름>을 다운받는다.

사용 예

명령어를 사용하기 위한 예시는 다음과 같다.

	username 사용자가 username2 사용자의 파일 조회
1	/file ls username2

Figure 12 명령어 사용 예 1

	username 사용자가 username2 사용자의 “hello.txt” 파일 다운로드
1	/file dn username2 hello.txt

Figure 13 명령어 사용 예 2

프로세스 시퀀스

파일 리스트 요청

Client A 가 Client B 에게 공유 중인 파일 리스트를 요청하는 흐름은 다음과 같다.

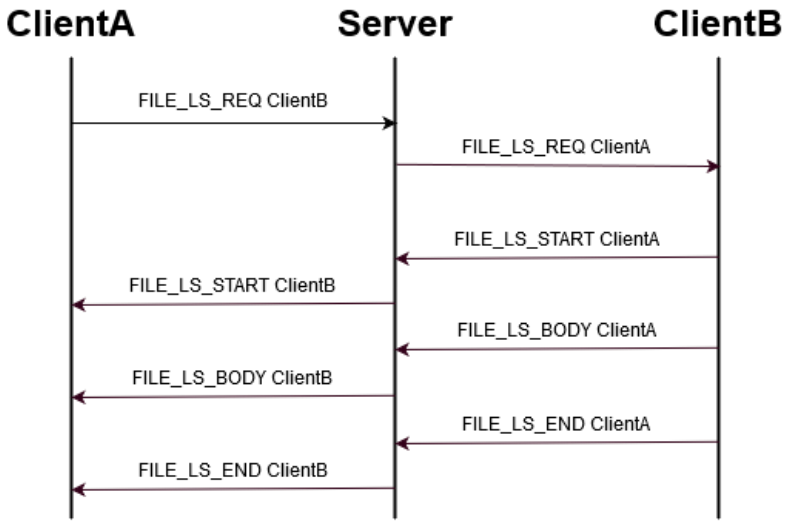


Figure 14 파일 리스트 요청 시퀀스 다이어그램

먼저 'Client A'는 서버에게 'Client B'의 파일 리스트를 요청하는 메시지를 보낸다. 그러기 위해 'FILE_LS_REQ' 메시지의 수신자에 'Client B'를 적는다. 이 메시지를 받은 서버는 그대로 전달하는 것이 아니라 수신자를 발신자인 'Client A'로 바꿔서 'Client B'에 전달한다. 이렇게 메시지에 지속적으로 발신자와 수신자가 기록되어 있으므로 서버는 파일 리스트 전송에 대한 상태를 추가적으로 추적할 필요가 없어진다. 'Client B'는 메시지를 받고 파일 리스트의 시작 메시지, 본문, 끝 메시지를 차례로 보낸다. 본문 메시지에는 각각의 파일 이름이 담겨 보내진다. 만약 디렉토리에 파일이 5 개 있다면 본문 메시지는 5 번 보내지게 된다. 하나의 메시지에 모든 파일 리스트를 담아 보내지 않는 이유는 한 번에 모든 파일리스트를 전송하면 파일의 개수가 많은 경우 프로토콜에서 지원하는 버퍼의 크기를 넘을 수 있기 때문이다. 이러한 정보를 받은 'Client A'는 받은 정보를 화면에 출력한다.

파일 공유

Client B 가 Client A 에게 공유중인 파일을 전송하는 흐름은 다음과 같다.

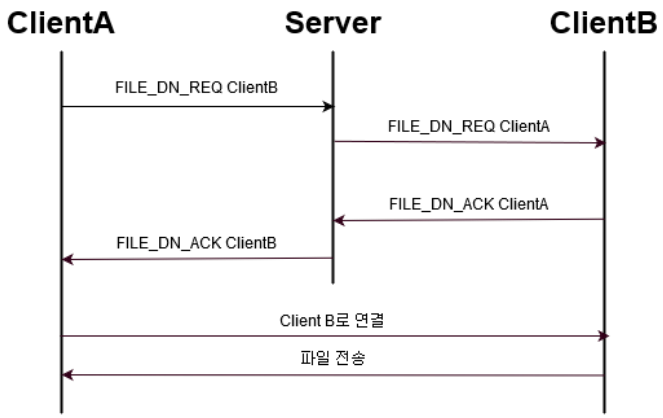


Figure 15 파일 전송 시퀀스 다이어그램

'Client A'는 서버로 파일 다운로드를 요청하는 메시지를 보낸다. 이 때, 'FILE_DN_REQ'의 수신자는 'Client B'였지만 서버에서 'Client B'로 메시지를 전달할 때는 발신자 이름인 'Client A'를 적어 보낸다. 그 후, 'Client B'에서 파일이 존재하면 'FILE_DN_ACK' 메시지를 전달하는데 이 때, P2P 파일 공유를 위해 파일 전송용 소켓을 열고 IP 주소와 포트 번호를 같이 적어 보낸다. 서버로부터 'FILE_DN_ACK'를 받은 'Client A'는 그 메시지 안에 있는 IP 주소와 포트 번호를 이용해 'Client B'로 바로 접속한다. 그 후, 'Client B'로부터 파일을 다운로드 받는다.

알려진 문제

메시지 처리

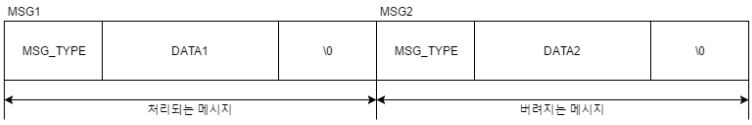


Figure 16 메시지 처리 문제

메시지 처리의 간편화를 위해 프로토콜을 1 옥텟(Octet)의 'MSG_TYPE'으로 메시지 종류를 구분하고 마지막에 널 문자('\0')를 이용해 메시지의 종료를 판단한다. 이렇게 함으로써 서버나 클라이언트는 메시지를 버퍼에서 한 번만 읽고 처리를 할 수 있다. 하지만 이 방식은 IO 가 충분히 느리고, 메시지를 처리하는 속도가 매우 빠르다는 전제하에 정상 작동을 보장한다. 즉, 메시지를 처리하는 과정에서 메시지가 연달아 두 개 이상 버퍼에 쌓이게 되면 첫 번째 메시지는 널 문자('\0')까지 읽고 처리가 되지만 두 번째 메시지는 널 문자 다음에 있으므로 그냥 버려진다. 이 때문에 서버와 클라이언트는 메시지를 주고 받을 때, 주의하여야 한다.

동작 결과

파일 리스트

파일 리스트 요청

파일을 다운로드 하기 전, 공유된 파일에 대한 리스트를 얻을 수 있다.

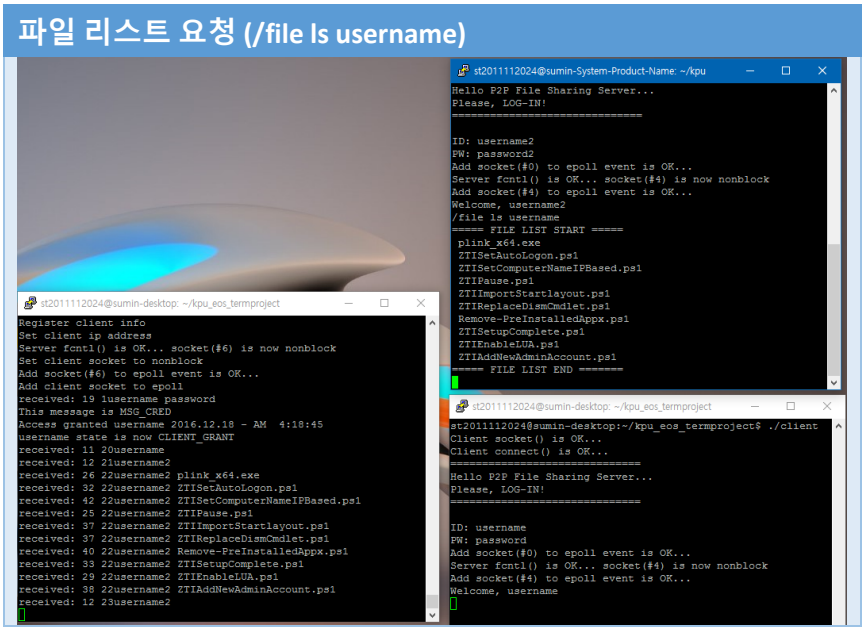


Figure 17 파일 리스트 요청

파일 리스트 요청 실패

파일 리스트를 요청했으나 서버에 접속한 사용자가 없을 경우, 서버는 사용자가 없음을 알리는 메시지를 클라이언트에게 보낸다.

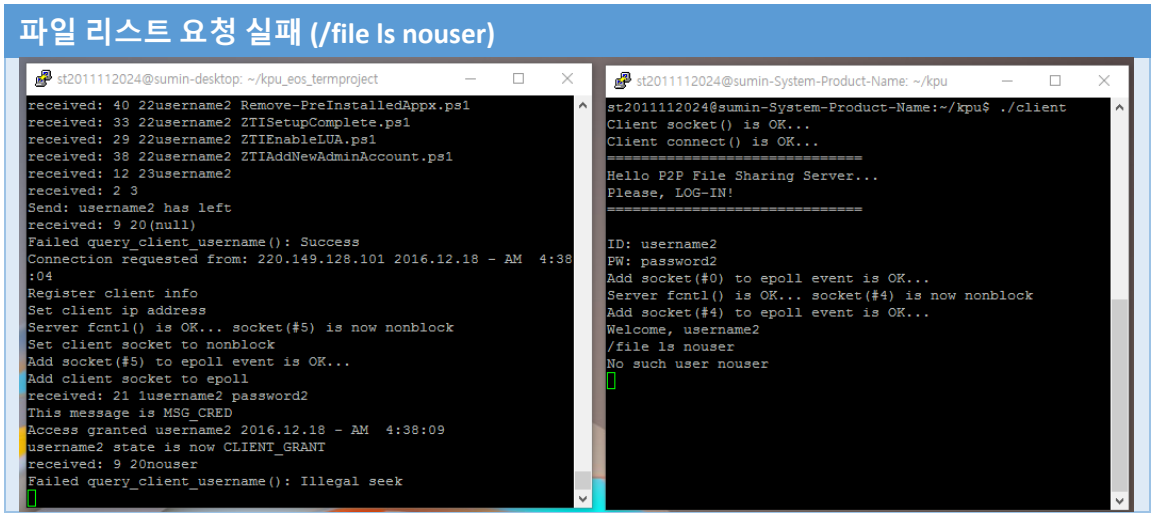


Figure 18 파일리스트 요청 실패

파일 다운로드

클라이언트 username 에서 username2 로부터 테스트 파일 test.txt 를 다운로드하는 것을 테스트 했으며 결과는 다음과 같다.

원본 test.txt

	원본 test.txt
1	DESCRIPTION
2	The epoll API performs a similar task to poll(2): monitoring multiple
3	file descriptors to see if I/O is possible on any of them. The epoll
4	API can be used either as an edge-triggered or a level-triggered
5	interface and scales well to large numbers of watched file
6	descriptors. The following system calls are provided to create and
7	manage an epoll instance:
8	
9	* epoll_create(2) creates an epoll instance and returns a file
10	descriptor referring to that instance. (The more recent
11	epoll_create1(2) extends the functionality of epoll_create(2).)
12	
13	* Interest in particular file descriptors is then registered via
14	epoll_ctl(2). The set of file descriptors currently registered on
15	an epoll instance is sometimes called an epoll set.
16	
17	* epoll_wait(2) waits for I/O events, blocking the calling thread if
18	no events are currently available.

Figure 19 원본 파일

파일 다운로드

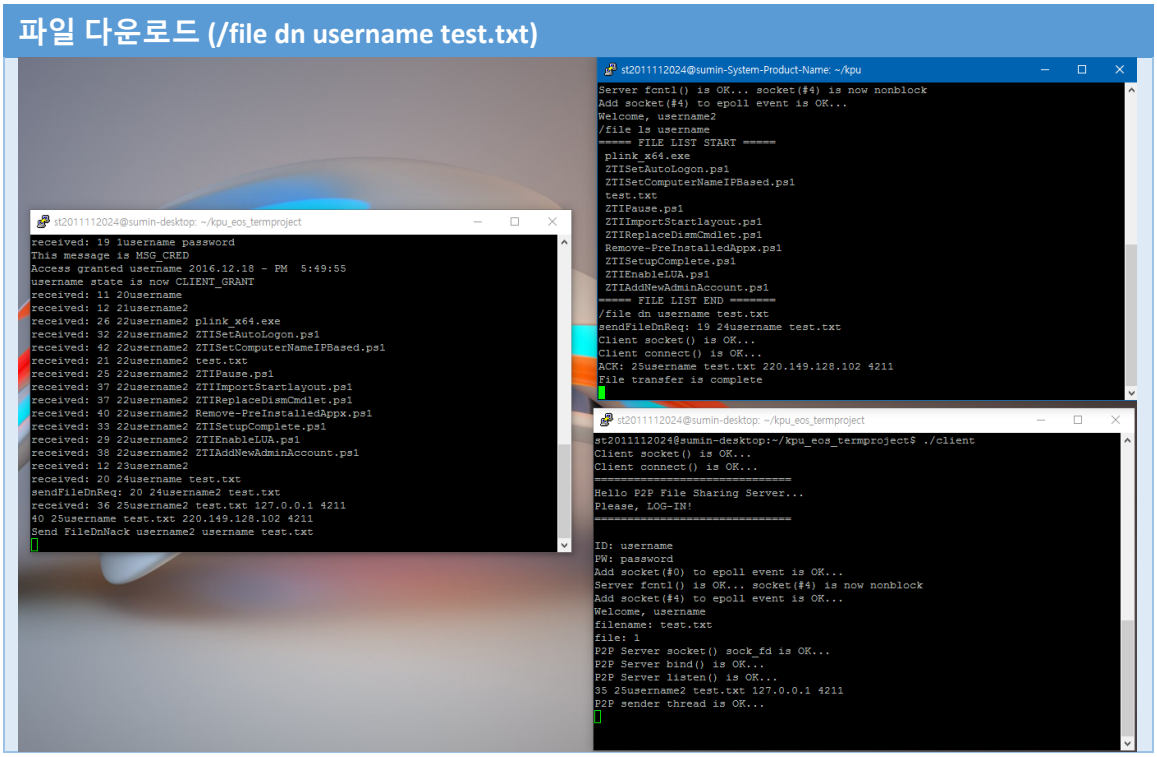


Figure 20 파일 다운로드

다운로드 후 test.txt

결과 test.txt

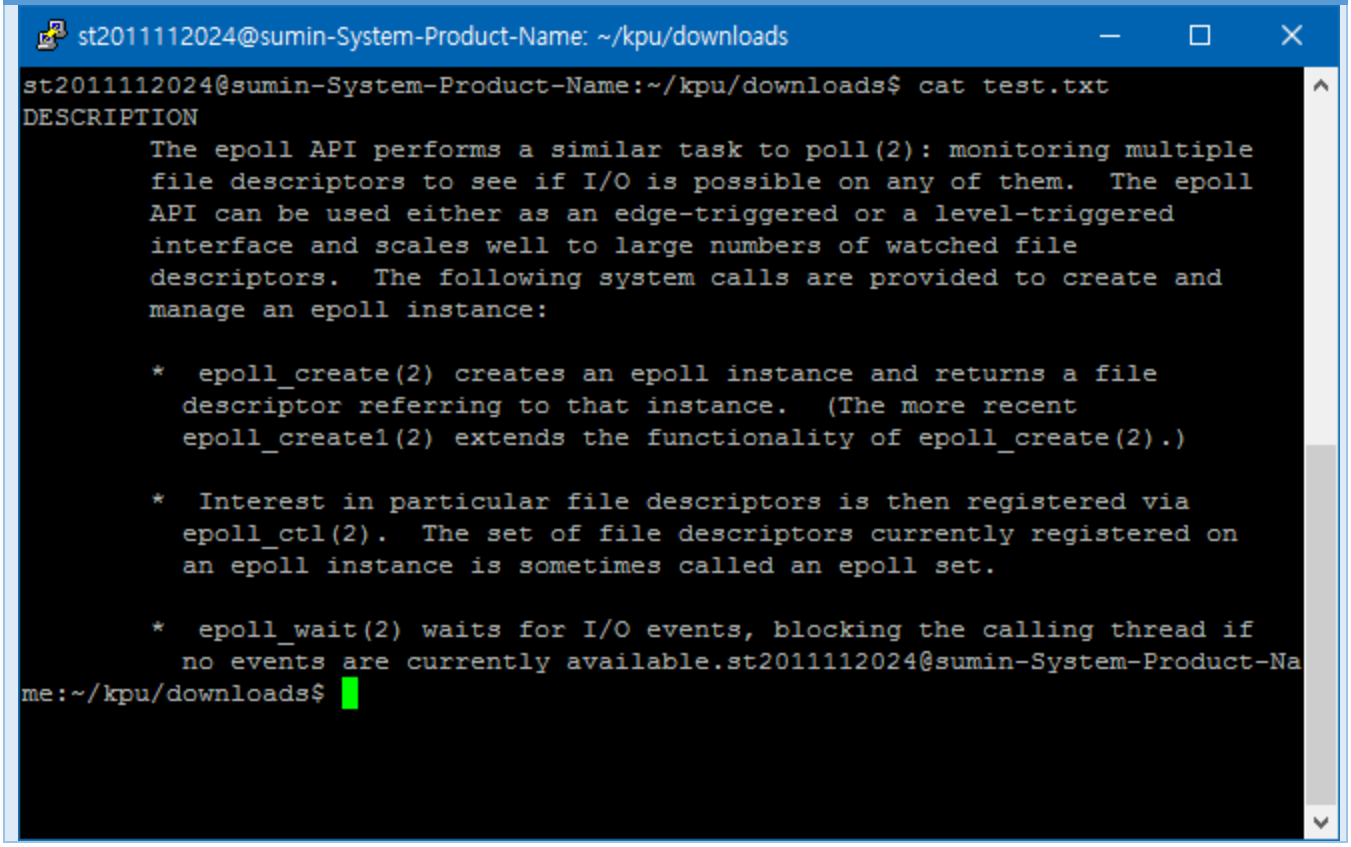


Figure 21 결과 test.txt