Introduction to Computer Networks

Lab 02: Single-client Server Implementation

1. 실습의 목표

socket 통신의 server 예제를 따라 해보고, 코드를 이해한다. 본 실습에서 구현하는 서버는 하나의 클라이언트만 받아서 통신하고, 그 클라이언트의 접속이 끝나면 종료되는 서버이다. 다음 실습에서는 여러 개의 클라이언트를 동시에 서비스할 수 있는 서버를 구현할 것이다.

2. 작성해 볼 프로그램

server: socket 통신을 위한 서버 쪽 프로그램

3. 프로그램

아래의 예제를 따라 server.cc 프로그램을 작성한다. 절대 copy & paste 하지 말고 직접 손으로 타이핑할 것! 그래야 자기 것으로 만들 수 있다.

```
// server.cc -- a simple socket server – serves only a single client

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <errno.h>
#include <string.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>
#include <arpa/inet.h>

#define MAXDATASIZE 1000
#define BACKLOG 10

int main(int argc, char *argv[]) {

    int sockfd;
    int new_fd;
    struct addrinfo hints, *servinfo;
    struct sockaddr_storage their_addr;
    socklen_t sin_size;
    char s[INET_ADDRSTRLEN];
    int rv;

    char buf[MAXDATASIZE];
    int numbytes;

    if(argc != 2) {
        printf("usage: server portnum\n");
        exit(1);
    }
```

```
    memset(&hints, 0, sizeof hints);
    hints.ai_family = AF_UNSPEC;
    hints.ai_socktype = SOCK_STREAM;
    hints.ai_flags = AI_PASSIVE; // use my IP

    if ((rv = getaddrinfo(NULL, argv[1], &hints, &servinfo)) != 0) {
        fprintf(stderr, "getaddrinfo: %s\n", gai_strerror(rv));
        exit(1);
    }

    if((sockfd = socket(servinfo->ai_family, servinfo->ai_socktype, servinfo-
>ai_protocol)) == -1) {
        perror("server: socket");
        exit(1);
    }

    if (bind(sockfd, servinfo->ai_addr, servinfo->ai_addrlen) == -1) {
        close(sockfd);
        perror("server: bind");
        exit(1);
    }

    freeaddrinfo(servinfo);

    if (listen(sockfd, BACKLOG) == -1) {
        perror("listen");
        exit(1);
    }

    printf("server: waiting for connections on port %s...\n", argv[1]);

    sin_size = sizeof their_addr;
    new_fd = accept(sockfd, (struct sockaddr*)&their_addr, &sin_size);
    if(new_fd == -1) {
        perror("accept");
        exit(1);
    }

    inet_ntop(their_addr.ss_family, &((struct sockaddr_in*)&their_addr)->sin_addr,
s, sizeof s);
    printf("server: got connection from %s\n", s);

    close(sockfd);

    while(1) {

        if((numbytes = recv(new_fd, buf, sizeof buf, 0)) == -1) {
            perror("recv");
            close(new_fd);
            exit(0);
        }

        if(numbytes == 0) {
            close(new_fd);
            break;
        }

        buf[numbytes] = '\0';
```

```
        printf("server: received '%s'\n", buf);

        if (send(new_fd, buf, strlen(buf), 0) == -1) {
            perror("send");
            close(new_fd);
            exit(0);
        }
    }
    close(new_fd);

    return 0;
}
```

4. 프로그램 설명

아래에는 코딩한 내용을 라인별로 설명하였다.

```
// server.cc -- a simple socket server -- serves only a single client

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <errno.h>
#include <string.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>
#include <arpa/inet.h>
```

- Server 프로그램을 만들 때 사용되는 헤더 파일들. Socket programming을 위해서 필요한 헤더들이므로 늘 모두 include 시켜준다.

```
#define MAXDATASIZE 1000
#define BACKLOG 1
```

- MAXDATASIZE: 데이터를 송신, 수신하는데 쓰이는 버퍼의 크기
- BACKLOG: accept 되기 전 대기 클라이언트의 최대 개수 (뒤에서 설명)

```
int main(int argc, char *argv[]) {

    int sockfd;
    int new_fd;
    struct addrinfo hints, *servinfo;
    struct sockaddr_storage their_addr;
    socklen_t sin_size;
    char s[INET_ADDRSTRLEN];
    int rv;
    char buf[MAXDATASIZE];
```

```
    int numbytes;
```

- main 함수의 시작과 사용될 변수들.
- sockfd, new_fd: 소켓 ID. 클라이언트는 하나의 소켓 ID만 사용하는 반면, 서버는 여러 개의 소켓 ID를 사용한다. 먼저 클라이언트를 기다리기 위한 소켓이 하나 필요하다. 클라이언트가 접속을 하면, 접속을 받아들이면서 새로운 소켓을 만들어 그 클라이언트를 담당하게 한다. 이는 여러 개의 클라이언트를 서비스하기 위한 구조이다.

다음과 같이 생각하면 된다. 호텔 프론트에서 한 명의 서버가 늘 손님을 기다린다. 손님이 올 때마다 새로운 벨보이를 지정해서 그 손님을 담당하게 하고 자신은 또 다른 손님을 기다린다. 소켓 서버는 이런 방식으로 구현된다.

```
    if(argc != 2) {
        printf("usage: server portnum\n");
        exit(1);
    }

    memset(&hints, 0, sizeof hints);
    hints.ai_family = AF_UNSPEC;
    hints.ai_socktype = SOCK_STREAM;
    hints.ai_flags = AI_PASSIVE; // use my IP

    if ((rv = getaddrinfo(NULL, argv[1], &hints, &servinfo)) != 0) {
        fprintf(stderr, "getaddrinfo: %s\n", gai_strerror(rv));
        exit(1);
    }
```

- 이 부분은 client와 유사하지만, 두 가지 차이가 있다. 먼저 hints.ai_flags = AI_PASSIVE이고, getaddrinfo의 첫 번째 인자는 NULL이다. 서버는 목적지로 접속하는 것이 아니고 접속을 기다리는 것이기 때문에 목적지 address가 없다.

```
    if((sockfd = socket(servinfo->ai_family, servinfo->ai_socktype, servinfo-
>ai_protocol)) == -1) {
        perror("server: socket");
        exit(1);
    }
```

- socket을 만든다.

```
    if (bind(sockfd, servinfo->ai_addr, servinfo->ai_addrlen) == -1) {
        close(sockfd);
        perror("server: bind");
        exit(1);
    }
```

- 클라이언트에서는 소켓을 만든 이후 서버로 접속하기 위해 connect 함수를 사용하였다. 서버에서는 클라이언트의 접속을 받기 위해 소켓과 자신의 특정 포트를 연결시키는 bind 함수를 호출해

야 한다.

```
    freeaddrinfo(servinfo);
```

- bind까지 하면 더 이상 servinfo 구조체의 정보는 필요가 없다. free 시켜준다.

```
    if (listen(sockfd, BACKLOG) == -1) {
        perror("listen");
        exit(1);
    }
```

- listen 함수를 호출하면 이제부터 클라이언트의 접속을 받아들일 수 있다. 여기서 두 번째 인자인 BACKLOG는 대기자의 수이다. 서버에 동시에 연결하고자 요청하는 클라이언트의 수가 많은 경우에, 연결이 확립되지 않은 채로 대기하게 된다. 이 대기열의 최대 크기가 BACKLOG이다. 이 BACKLOG의 개수보다 넘어서서 클라이언트가 오면 접속 요청을 거절한다.

```
    printf("server: waiting for connections on port %s...\n", argv[1]);

    sin_size = sizeof their_addr;
    new_fd = accept(sockfd, (struct sockaddr*)&their_addr, &sin_size);
    if(new_fd == -1) {
        perror("accept");
        exit(1);
    }

    inet_ntop(their_addr.ss_family, &((struct sockaddr_in*)&their_addr)->sin_addr,
s, sizeof s);
    printf("server: got connection from %s\n", s);
```

- 클라이언트의 요청이 오면 요청을 받아들인다. accept는 blocking 함수로 클라이언트로부터 연결 요청이 들어올 때까지 리턴 되지 않고 기다린다. accept 함수는 클라이언트의 연결 요청을 받아들인 후에 이 클라이언트에 새로운 소켓을 부여하는데, 이 새로운 소켓의 ID가 리턴 값인 new_fd이다. 또한, 두 번째 인자인 their_addr에는 클라이언트의 주소 정보가 들어간다.

inet_atop 함수를 이용하여 클라이언트의 IP address를 스트링으로 만들고, 출력한다.

```
    close(sockfd);
```

- 이제부터는 새로운 소켓이 클라이언트를 담당하므로, 원래 클라이언트를 기다리던 sockfd는 피요 없어지므로 닫아준다. 지금 이렇게 하는 이유는 이 서버가 single-client 서버이기 때문이다! 당연히 여러 개의 client를 받아들이는 서버에서는 프론트 서버(sockfd)를 닫으면 안 된다.

```
while(1) {

        if((numbytes = recv(new_fd, buf, sizeof buf, 0)) == -1) {
            perror("recv");
            close(new_fd);
            exit(0);
        }

        if(numbytes == 0) {
            close(new_fd);
            break;
        }

        buf[numbytes] = '\0';
        printf("server: received '%s'\n", buf);

        if (send(new_fd, buf, strlen(buf), 0) == -1) {
            perror("send");
            close(new_fd);
            exit(0);
        }
    }
    close(new_fd);

    return 0;
}
```

- 클라이언트와 연결된 이후의 서버의 동작이다. 일단 무한루프로 쌓여 있어서 특정 조건을 만족하기 전에는 계속 반복된다. 아래에서는 이 부분을 더 세부적으로 살펴본다.

```
        if((numbytes = recv(new_fd, buf, sizeof buf, 0)) == -1) {
            perror("recv");
            close(new_fd);
            exit(0);
        }
```

- 먼저 클라이언트의 메시지를 기다려서 수신한다. recv 함수는 blocking 함수이므로 메시지가 수신되기 전에는 리턴되지 않는다.

```
        if(numbytes == 0) {
            close(new_fd);
            break;
        }
```

- 서버에서 중요한 부분이다. recv 함수의 리턴 값은 수신된 메시지의 바이트 수이다. 만약 에러가 발생하면 -1이 리턴된다. 그런데 0이 리턴되는 경우는, 클라이언트가 연결을 종료한 경우이다. 클라이언트가 연결을 종료하면, 연결종료 메시지를 서버로 보내고, 서버의 recv 함수는 리턴 값을 0

으로 해서 리턴한다. 이런 경우에는 서버도 종료시켜주면 되기 때문에 new_fd를 닫고, 무한루프에서 빠져나가면 된다.

```
        buf[numbytes] = '\0';
        printf("server: received '%s'\n", buf);

        if (send(new_fd, buf, strlen(buf), 0) == -1) {
            perror("send");
            close(new_fd);
            exit(0);
        }
```

- recv의 리턴 값이 0도 아니고 -1도 아닌 경우에는 어떤 메시지가 수신되었다는 뜻이다. 이 경우에는 수신된 메시지의 끝에 NULL을 붙여서 스트링을 만들고, 출력해준다.

그런 다음, send 함수를 호출하여 수신된 메시지를 그대로 다시 클라이언트로 보내준다. (buf에 있는 내용을 바꾸지 않았으므로 그대로 클라이언트로 전송됨. 만약 다른 내용을 클라이언트로 보내고 싶으면 buf의 내용을 바꿔주면 된다.)

5. 컴파일 및 실행

- 프로그램을 컴파일 하여 실행파일을 얻는다.

```
$ g++ -o server server.cc
```

- 두 개의 ssh 창을 띄운다. 이번 실습에서 작성한 server를 먼저 띄우고, 지난 실습에서 만들었던 client를 띄워서 접속이 되는지 확인한다.

```
$ ./server 55555
```

- 클라이언트를 실행시킨다.

```
$ ./client 127.0.0.1 55555
```

주의!) 포트번호는 각자 다른 번호로 해야 한다. 포트번호는 20000-60000 사이 중에서 아무 번호나 선택한다. 만약 server 프로그램을 실행시켰을 때 Address already in use 에러가 난다면 다른 사람의 포트번호와 충돌한 것이므로 다른 번호를 선택한다.

제대로 작동하는지 확인한다.

6. 과제

server.cc를 다음과 같이 수정하여 lab2_server.cc 파일로 만든다.

- 현재 서버는 받은 메시지를 그대로 돌려주는 echo server이다. 하지만 iphone Siri와 같이 몇 가

지 단어에는 반응하도록 수정한다.

- 만약, 클라이언트가 “hello”를 보내는 경우 서버는 “how are you?”라고 클라이언트로 보낸다.

- 만약, 클라이언트가 “thank you”를 보내는 경우 서버는 “you are welcome!”이라고 보낸다.

- 만약, 클라이언트가 “/random 100”과 같이 보내면, 서버는 1부터 100까지의 숫자 중 랜덤으로 하나를 선택하여 클라이언트로 보내준다.

- 이 세가지 이외의 메시지가 오면 못 알아들었으므로 “what?” 이라고 클라이언트로 보낸다.

- 아래와 같이 컴파일 하여 서버를 켜 놓는다.

```
$ g++ -o lab2_server lab2_server.cc
$ ./lab2_server 55555
```

- 클라이언트 프로그램을 이용하여 “hello”, “thank you”, ”/random 100”, ”/random 1000”, ”who are you?” 와 같은 문장을 보내 봄으로써 테스트한다.