SP - 3.3 Socket Interface - Echo Server

hyeok's Log · 2022년 4월 14일

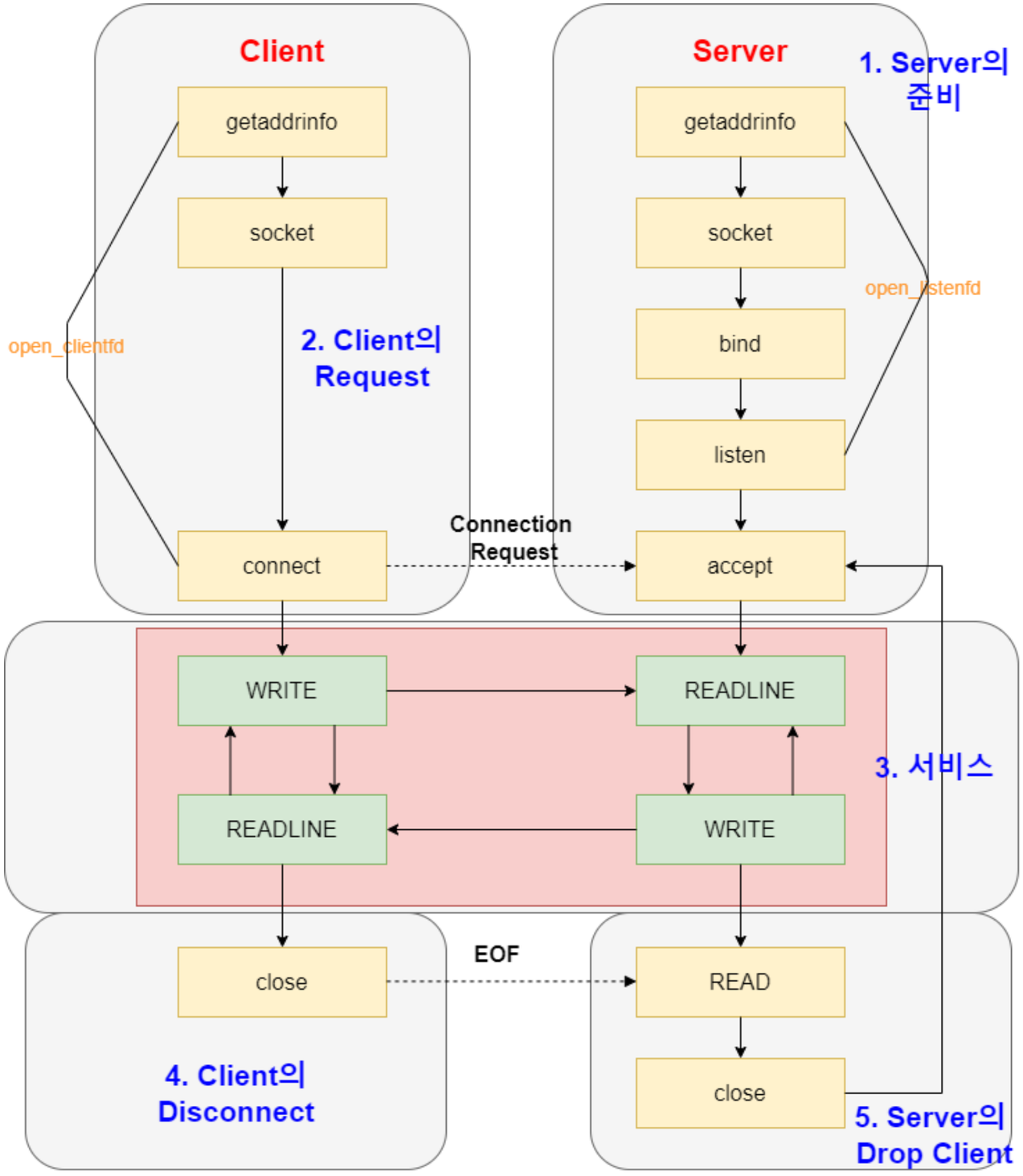
sp

본격적으로 Socket Interface를 사용해 네트워크 프로그래밍을 하는 방법에 대해 알아보겠다. 우리는 Socket Interface의 이해를 위해, Echo Server를 예시로 들 것이다. Linux Shell에서 echo 명령을 사용해본 경험이 있을 것이다. Echo Server는, 사용자(Client)로부터 받은 메세지를 반복한다. 동굴에서 소리를 치면 다시 소리가 돌아오는 것을 에코(Echo)라고 하는 것이 기억나는가? 바로 그 동작을 수행하는 서버이다.

Socket Interface의 이해 - Echo Server

Introduction

아래 그림을 보자.



- **서버에서는 getaddrinfo 함수를 호출해 IP Address와 Port Number에 대한 정보를 가져온다.**

    - *이 정보들을 이용해 Socket을 만든다.*
    - *만든 Socket을 Binding한다.*
    - *Binding된 소켓을 가지고 Listening을 한다.*
        - **Listen : Server가 Client의 Connection Request를 받겠다는 것!**

- **Client가 Connection Request를 하게 되면, Listening 중인 Server에서는 accept라는 함수를 통해 Connection을 정립한다.**
    - **accept를 하면, Client와 Server 사이에 Connection이 만들어져서 서로 데이터를 주고 받을 수 있게 된다.**

- 한편, **클라이언트를 보자. 똑같이 getaddrinfo를 통해 IP주소와 포트넘버를 가져와 소켓을 생성**한다.
- 이어서, **connect 함수와 Socket을 이용해 Server에게 Connection Request를 보낸다.**
- **서버에서는 이 Request를, OS Kernel의 Network Stack에 있는 TCP Manager를 이용해 큐잉한다. ★**
    - ***Dequeue하여 accept하는 것이다.***

- accept를 완성하면, Connection이 빌드된다. accept를 한 다음, 서버는 Buffered I/O, 예를 들어서, 지난번에 언급한 RIO Package의 rio_readlineb 함수를 사용해 데이터가 오기를 기다린다.
- 클라이언트는 컨넥션이 맺어졌으니, rio_writen을 이용해 메세지를 작성한다.
    - 메세지를 작성해서 Enter 키를 누르면, 메세지가 서버로 날아간다.
        - **클라이언트의 소켓 디스크립터에 쓴다! ★**
    - **서버에서는 소켓(디스크립터)으로부터 메세지를 Read한다. ★**
    - 서버에서는 읽은 메세지를 다시 Socket Descriptor에 반사한다.

    - 서버는 또 다시 메세지를 받기 위해 Loop를 돈다.

- 한편, 클라이언트는 앞서 Write 후, rio_readlineb를 호출해 서버의 Echo를 기다린다. 클라이언트 자신이 보냈던 메세지가 다시 돌아오게 되면, 그것을 읽고, 화면에 프린트한다.

- 이어서, 클라이언트도 루프를 돌면서 쓰기를 기다린다.

~> 이러한 과정이 반복되는 것이다. Server와 Client가 Connection을 맺고 나서, **각자 Iteration을 돌면서 Echo Service를 제공/소비**하는 것이다.

- **Client가 EOF를 날리면 Connection이 끊어지게 된다. close 함수를 통해 디스크립터를 닫음으로써 이것이 수행된다. ★**
    - ***서버에서는 EOF를 받으면, 마찬가지로 close한다.***

> 현재 이 서버에서는, Client와 Connection을 맺어서 통신하고 있을 때, 동시에 다른 Client에서 Request가 오면, 복수의 Connection을 동시에 처리하지 못하고, 다른 Client를 기다리게 한다. 즉, **Concurrent Server가 아니다.**
>
> > 우리는 Chapter3를 끝마친 후, Concurrent Server에 대해 다룰 것이다.

상기한 과정을 요약하면 다음과 같다.
1) 서버를 먼저 실행시킨 후, open_listenfd를 호출한다.
2) 클라이언트를 실행시킨 후, open_clientfd를 호출한다.
※ 상기한 두 openfd함수에서는 그림에 묘사된 Flow를 진행한다.
3) 서버가 Request를 Accept하면 Connection이 맺어지고, 서비스가 수행된다.
4) Client가 Disconnect한다.(close)
5) Server가 Drop Client를 하고, Connection이 끝나게 된다.

socket function (Client & Server)

> **Client와 Server는 Socket Descriptor를 만들기 위해 'socket' 함수를 호출**한다.

- 서버와 클라이언트는 각자 **socket 함수를 호출해 소켓 디스크립터를 생성**한다.
- **socket함수는 File Descriptor Table의 인덱스를 반환한다. ★**

```
int socket(int domain, int type, int protocol);
// Domain 이름, Type, 프로토콜을 파라미터로 받는다.

/* Example */
int clientfd = Socket(AF_INET, SOCK_STREAM, 0);     // Stevens Style Wrapper
// AF_INET : IPv4 기준 Internet Connection임을 나타내는 매크로
// SOCK_STREAM : 소켓이 컨넥션의 Endpoint임을 나타내는 매크로
// clientfd 변수에는 소켓 파일 디스크립터가 넘어온다.
```

- 일반적인 관습으로, socket 함수의 반환값을 **'sockfd'**라는 이름의 변수에 담는다.

bind function (Server)

> **Server는 socket 함수로 만든 소켓 디스크립터에 대해 bind 함수를 호출**한다.
>
> > ***OS Kernel에게, 생성한 Socket Address와 Descriptor, IP Address, Port Number 등을 전달하는 작업을 수행한다. ★***

```
int bind(int sockfd, SA *addr, socklen_t addrlen);
// Connection의 Endpoint 주소가 addr일때
// sockfd를 읽음으로써, Connection에서 오는 데이터(바이트)를 읽을 수 있다.
```

- 프로세스는 socket 함수의 반환값인 sockfd를 이용해 데이터를 읽거나 쓴다. Connection에서

주고받는 Byte Streams를 다룰 수 있는 것이다.

    - **Connection의 Endpoint인 Socket으로 데이터가 전달되고, 받는 것이다. ★**

- **서버가 sockfd 디스크립터를 통해서 Communication을 하겠다고 OS Kernel에게 알리는 과정이 bind인 것이다. ★**

listen function (Server)

> **Server는 bind 이후 listen을 호출**한다.

- **OS Kernel은 Default 설정으로 인해, socket 함수로 만들어지는 모든 반환 Descriptor를 Connection 관계에서 Client 쪽의 Endpoint에 위치한 Active Socket으로 인식한다. ★★★**

    - *서버에서는 **listen 함수를 통해 "이 sockfd는 Client쪽의 fd가 아니라, Server쪽의 fd야!"라는 정보를 OS에게 알린다. ★★★***

- *즉, 서버는 Listen을 통해 **"이 sockfd는 서버꺼구, 이제 나는 Connection Request를 받을(Listening)꺼야!"라는 말을 OS Kernel에게 알리는 것이다. ★★★***

> 클라이언트로부터 오는 모든 **Request를 받을 준비**가 된 것이다.

```
int listen(int sockfd, int backlog);
// "커널아, 이 sockfd를 이제 Listening Socket으로 사용할거야!"
// backlog : 이 sockfd를 위해 커널이 최대 몇개의 Request를
// 큐잉할지를 지정하는 값이다. 최대 1024개까지 가능하다고 알려진다.
```

- **커널의 Network Stack에 있는 TCP Manager가 Connection Request를 큐잉**한다고 했다.

    - **Connection Request를 Blocking하는 것이 아니라, Listening하는 것**이다.
        - **Connection이 맺어지면, 큐잉된 '그 Request'를 제거**한다.

- listen 함수의 반환값을 관습적으로 **listenfd**라는 변수명의 변수에 저장한다.

accept function (Server)

> **Server는 listen을 통해 Client로부터의 Connection Request를 기다리다가, Request가 오면 accept System Call을 이용해 Connection을 정립**한다.

```
int accept(int listenfd, SA *addr, int *addrlen);
```

- 서버는 Connection Request가 listenfd에 바운드된 Connection에 도착하면, **도착한 클리언트의 Socket Address를 addr에, 사이즈를 addrlen에 업데이트**한다. ★★★
    - 그리고나서, **Connected Descriptor를 반환한다. (accept의 반환값)**
- **Connected Descriptor : UNIX I/O 루틴을 이용해 Client와 Server가 소통할 수 있는 File Descriptor이다.**
    - **accept 함수의 반환값**이다.
- 관습적으로 accept 함수의 반환값은 **'connfd'**라는 변수명의 변수에 저장한다.

connect function (Client)

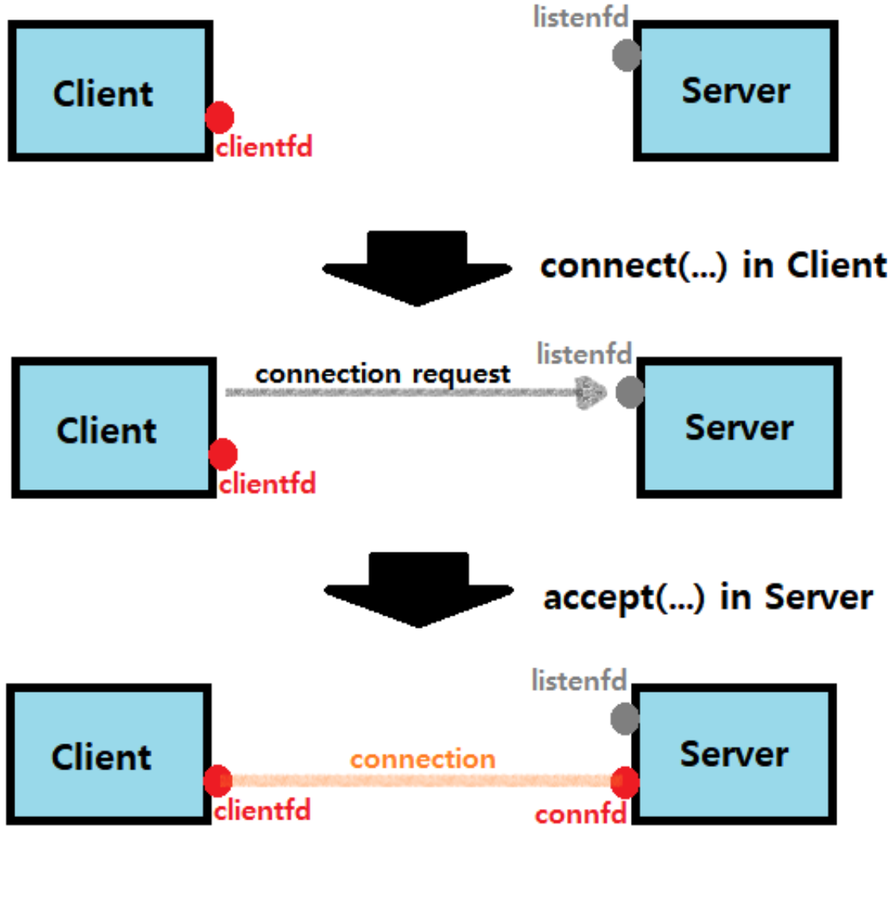
> 한편, **Client는 connect라는 함수를 통해 Server에 Connection Request를 보낸다.**
>
> > "서버야, 나 Connection 만들거야!"

```
int connect(int clientfd, SA *addr, socklen_t addrlen);
```

- 관습적으로 connect 함수의 반환값은 **'clientfd'**라는 변수명의 변수에 저장한다.
- **Server에서 Request를 accept해 Connection이 빌드되면, Client는 clientfd에 데이터를 읽고 쓸 수 있다. ★★★**

> 상기한 과정으로 만들어지는 Connection의 식별 Pair는 다음과 같다.
>
> > **(ClientAddress:EphemeralPortNum, addr.sin_addr:addr.sin_port)**

connect & accept routine



1) 현재 클라이언트는 clientfd, 서버는 listenfd를 만든 상태이다. listenfd 인덱스값이 3번이라 해보자.
(2번 STDERR 다음)
~> 서버는 listen 이후, accept를 호출해 Connection Request를 계속 기다린다.

2) 클라이언트의 connect 함수로 인한 Connection Request가 OS의 Queue에 도달한다.
~> 서버는 Dequeue하여 Connection을 빌드한다.

3) 빌드가 되면, 서버는 accept 함수로부터 connfd를 받아온다. listenfd가 3번이었으니, connfd는 4번일 것이다.
~> connfd와 clientfd 사이에 Channel이 형성된다.
=> 클라이언트는 clientfd를 통해서 읽고 쓰고, 서버는 connfd를 통해서 읽고 쓴다.

전체 과정을 다시 한 번 요약해보자.
1) 서버가 먼저 소켓을 만들고 **bind**한다. sockfd에 해당하는 Descriptor를 가지고 **Connection을 맺겠다고 OS 커널에게 알리는 것**이다.

2) 이어서 서버가 **listen과 accept를 통해 Connection을 정립**하고, Client와 Server가 서로 데이터 통신을 하게 된다. (**Client에선 소켓을 만들고 connect를 호출**해 Request를 보낸 것)

3) connect와 listen 이후 시점으로 생각해보자. 클라이언트에는 clientfd가, 서버에는 listenfd가 있다. **서로가 각자 만든 Socket File Descriptor**이다.

4) Client의 Request를 listen 상태의 서버가 accept를 호출해 받아들이고, connfd가 만들어져서 clienfd와 connfd가 Connection을 맺는다.

5) 서버 입장에서는 fd가 두 개인 것이다. Listening을 위한 listenfd, 그리고 Connection을 맺은 connfd가 있다.

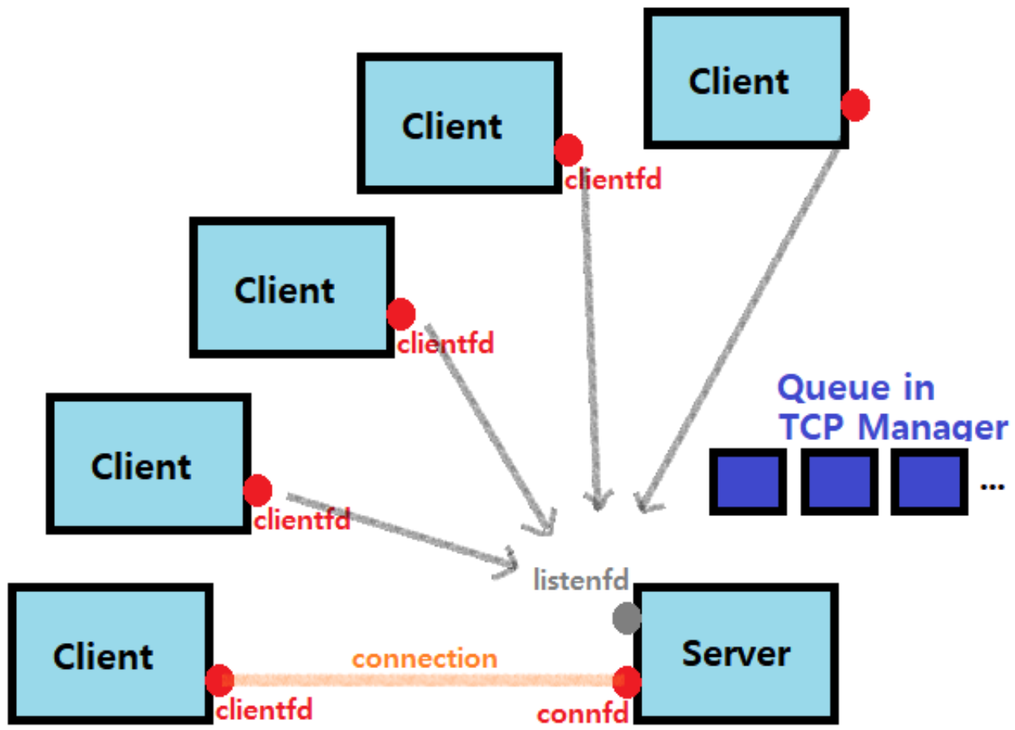
Why listenfd and connfd ?

왜 서버에서는 listenfd와 connfd를 따로 두는 것일까?

Answer) 서버는 기본적으로 여러 클라이언트와 소통해야한다. 즉, 현재 Connection이 맺어진 클라이언트 외에도 다른 클라이언트로부터 Request가 또 올 수 있다.

~> listenfd는 이 Request를 큐잉할 수 있게 한다. OS Kernel 내의 TCP Manager에 말이다.

~> accept 함수 호출은 현재 수행중인 Connection이 Disconnected 전까지는 호출되지 않는다. 따라서 listenfd를 이용해 Request를 큐잉(보관) 해놓는 것이다. ★



- **Listening Descriptor**
  - **Client의 Connection Request를 받기 위한 Endpoint**이다.
  - 서버 프로세스에 대해서 **한 번만 생성되고, 서버가 죽을 때까지 계속 살아있는다.**
- **Connected Descriptor**
  - **Client와의 Connection 관계에서의 Endpoint**이다.
  - 서버가 Connection **Request를 accept할 때마다 생성**된다.
  - **서비스가 종료되면 소멸**된다.

> Why Two fd for Server?
>
> > ***여러 클라이언트와의 Connection을 Concurrent하게 대응하기 위함이다. ★★★***

- c1이라는 클라이언트가 컨넥션을 요청하면, 서버가 accept를 하고, c1에 대한 connfd를 만든다.
    - 이때, 동시에 c2라는 클라이언트가 요청을 하면, 서버가 이를 따로 accept하여 c2에 대한 connfd가 만들어진다.
- **즉, connfd가 여러개 생길 수 있는 것이다. 하나의 listenfd만 두고 말이다.**
    - **Concurrent Server를 구축해서 여러 개의 Connection을 Simultaneous하게 처리할 수 있다.**
        - **각각의 (clientfd_i, connfd_i) Pair를 두고 동시 처리를 하는 것이다. ★**

(참고로, 현재 예시 상황의 Echo Server는 Concurrent Server가 아니다. Sequential하게 connfd를 만들어서 하나하나 Connection을 처리하고 있다. 왜냐? 이제부터 시작할 코드 레벨 분석을 보면 알 수 있다.)

Echo Server - Code Level Analysis

아래의 코드를 천천히 음미하면서 익히자. 위의 예시 에코 서버에 대한 코드이다. Stevens의 교재에서 추출 및 변형하였다. 자세한 설명은 주석으로 대체한다.

open_clientfd function (Client)

```
int open_clientfd(char *hostname, char *port) {
    struct addrinfo hints, *listp, *p;
    int clientfd;

    /* 이 함수를 호출하는 Client가 통신하게될 Server의 Address를 받는 과정 */
```

```
    memset(&hints, 0, sizeof(struct addrinfo));
    hints.ai_socktype = SOCK_STREAM;            // Connection Open
    hints.ai_flags = AI_NUMERICSERV;            // addrinfo 구조체의 hint 설정
    hints.ai_flags |= AI_ADDRCONFIG;            // (받아들이자)
    Getaddrinfo(hostname, port, &hints, &listp);
    // listp가 연결리스트를 가리키게 된다. 안에 다양한 IPv4 주소들이 들어있다.

    /* 성공적으로 Connect할 대상을 찾기 위해 연결리스트를 순회한다. */
    for (p = listp; p; p = p->ai_next) {        // 순회마다 소켓을 생성한다.
        if ((clientfd = socket(p->ai_family, p->ai_socktype, p->ai_protocol)) < 0)
            continue;           // 현재 조회 노드가 소켓 생성 실패하면, 다음 노드로!

        /* 서버와 소통할 소켓 생성에 성공하면 */
        if (connect(clientfd, p->ai_addr, p->ai_addrlen) != -1) // Request!
            break;              // Connection이 맺어지면, 순회를 종료!

        Close(clientfd);        // Connection이 맺어지지 않으면, Close fd!
    }

    /* connect가 되면, 필요없는 연결리스트 정보는 free해버림. */
    Freeaddrinfo(listp);
    if (!p)             // 순회를 다 돌았으면, open_clientfd 함수 자체가 실패한 것
        return -1;
    else                // Connection을 맺은 File Descriptor를 반환!
        return clientfd;
}
```

open_listenfd function (Server)

```
int open_listenfd(char *port) {
    struct addrinfo hints, *listp, *p;
    int listenfd, optval = 1;

    /* 이 함수를 호출하는 Server가 통신하게될 Client의 Address를 받는 과정 */
    memset(&hints, 0, sizeof(struct addrinfo));
    hints.ai_socktype = SOCK_STREAM;            // Connection Open
    hints.ai_flags = AI_PASSIVE | AI_ADDRCONFIG;// hint는 일단 그냥 받아들이자.
    hints.ai_flags |= AI_NUMERICSERV;
    Getaddrinfo(NULL, port, &hints, &listp);

    /* bind할 수 있는 대상을 찾기 위해 연결리스트를 순회한다. */
    for (p = listp; p; p = p->ai_next) {        // 순회마다 소켓을 생성한다.
        if ((listenfd = socket(p->ai_family, p->ai_socktype, p->ai_protocol)) < 0)
            continue;           // 현재 조회 노드가 소켓 생성 실패하면, 다음 노드로!

        /* bind로부터, '주소 사용중' 에러를 막기위한 처리 (받아들이자) */
```

```
        Setsockopt(listenfd, SOL_SOCKET, SO_REUSEADDR, (const void *)&optval , sizeof(int));

        /* 뽑아낸 주소에 Descriptor를 Bind시킨다. */
        if (bind(listenfd, p->ai_addr, p->ai_addrlen) == 0)
            break;              // bind에 성공하면, 순회를 멈춘다.
        Close(listenfd);        // bind에 실패하면, Close하고, 순회를 이어간다.
    }

    /* bind가 되면, 필요없는 연결리스트 정보는 free해버림. */
    Freeaddrinfo(listp);
    if (!p)                     // 순회를 다 돌았으면, open_listenfd 함수 자체가 실패한 것
        return -1;

    /* bind에 성공한 fd에 대해 listen 함수를 호출해 Request를 기다린다. */
    if (listen(listenfd, LISTENQ) < 0) {    // Connection Request가 온다.
        Close(listenfd);    // listen에 실패하면 open_listenfd 함수 자체가 실패한 것
        return -1;
    }
    return listenfd;        // listen에 성공한 fd를 반환한다.
}
```

~> open_listenfd 함수는 서버의 listen까지의 과정을 커버한다.

main function (Client)

```
int main(int argc, char **argv) {
    char *host, *port, buf[MAXLINE];
    int clientfd;
    rio_t rio;              // 네트워크 통신에 적합한 RIO Package를 사용한다.

    host = argv[1];
    port = argv[2];

    clientfd = Open_clientfd(host, port);   // connect 함수까지 수행!
    Rio_readinitb(&rio, clientfd);          // RIO Package 사용 준비

    while (Fgets(buf, MAXLINE, stdin) != NULL) {    // 계속 반복 루틴
        Rio_writen(clientfd, buf, strlen(buf));     // 쓴다.
        Rio_readlineb(&rio, buf, MAXLINE);          // 읽는다. (서버 반응 대기)
        Fputs(buf, stdout);                         // 응답이 오면 프린트!
    }

    Close(clientfd);
```

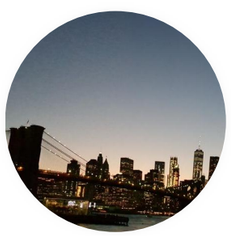
```
    return 0;
}
```

main function (Server)

```
void echo(int connfd) {
    char buf[MAXLINE];
    size_t n;
    rio_t rio;

    Rio_readinitb(&rio, connfd);
    while((n = Rio_readlineb(&rio, buf, MAXLINE)) != 0) {    // EOF 받기 전까지 ★
        printf("server received %d bytes\n", (int)n);        // EOF는 Client가 전달
        Rio_writen(connfd, buf, n);
        // Rio_writen이 수행되면 Client main의 readlineb 다음(Fputs)이 수행 ★★
    }
}

int main(int argc, char **argv) {
    int listenfd, connfd;
    socklen_t clientlen;
    struct sockaddr_storage clientaddr;
    char client_hostname[MAXLINE], client_port[MAXLINE];

    listenfd = Open_listenfd(argv[1]);      // listen까지의 작업을 수행
    while (1) {
        clientlen = sizeof(struct sockaddr_storage);    // 이 부분 중요(후술)
        connfd = Accept(listenfd, (SA *)&clientaddr, &clientlen); // Accept!

        Getnameinfo((SA *) &clientaddr, clientlen,
            client_hostname, MAXLINE, client_port, MAXLINE, 0);

        printf("Connected to (%s, %s)\n", client_hostname, client_port);
        echo(connfd);
        Close(connfd);      // 컨넥션 끝나면 계속 반복(accept~close를 계속 반복)
    }                       // Not Cuncurrent Server!!!! ★★★

    return 0;
}
```

~> 코드를 보면 알 수 있듯이, 이 에코 서버는 Concurrent하지 않다. accept 후, 해당 Connection을 위한 echo routine을 다 수행하고 나서, close하고, 다시 accept하고~ 이런 과정을 Sequential하게 반복하고 있으므로!

이렇게 해서, 네트워크 프로그래밍에 대한 기초 개념을 익혔다. 이제, 다음 포스팅부터는 본격적인 Concurrent Programming에 대해 다룰 것이다.



hyeok's Log

팔로우

이전 포스트
SP - 3.2 Network Programmin...

다음 포스트
SP - 4.1 Concurrent Programm...

0개의 댓글

댓글을 작성하세요

댓글 작성