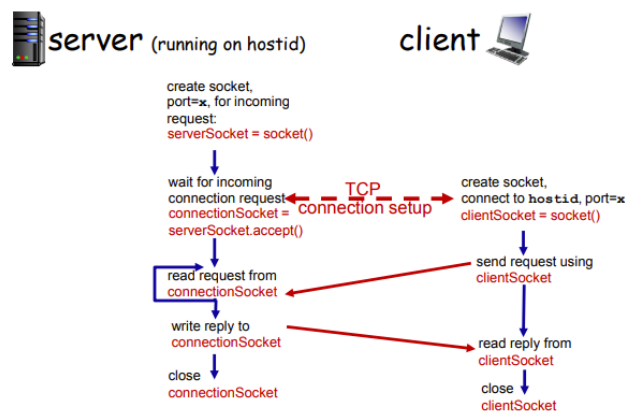


# [CYDF329-TP]

## <2022350227>-<장정호>

## 인트로

### Client/server socket interaction: TCP



class 17-18 pdf

Redis-Like 네트워크의 경우 tcp 프로토콜을 사용하므로, 위의 네트워크 구조를 참고해 구현했다. 다만, 서버가 다중 클라이언트의 접속을 허락해야 하므로, thread를 사용해 해당 부분을 추가적으로 구현했다.

## 목차

1. Redis-Like 네트워크 구현을 위한 socket 함수들
2. Client 구현 세부 사항
3. Server 구현 세부 사항
4. 빌드 방식 및 실행 결과

# Redis-Like 네트워크 구현을 위한 socket 함수들

```
// https://man7.org/linux/man-pages/man2/socket.2.html
#include <sys/socket.h>

int socket(int domain, int type, int protocol);
```

socket 함수는 세 개의 매개변수와 int형의 반환값을 가진다.

- **domain:** 통신할 주소 family.
- **type:** 소통 semantic(의미).
- **protocol:** 소켓에 사용할 프로토콜.
- **반환값:** 생성된 소켓의 디스크립터.

socket 함수는 세 개의 매개변수가 가리키는 특징을 지닌 소켓을 생성한다. 매개변수들의 특징은 **domain**(AF\_INET(IPv4), AF\_INET6(IPv6)...), **type**(SOCK\_STREAM(신뢰적 데이터 전송), SOCK\_DGRAM(비신뢰적 데이터 전송)...), **protocol**(IPPROTO\_TCP, IPPROTO\_UDP...)과 같다.

**반환값**은 소켓 생성에 성공하면 해당 소켓의 디스크립터, 실패하면 -1이다.

```
// https://man7.org/linux/man-pages/man2/connect.2.html
#include <sys/socket.h>

int connect(int sockfd, const struct sockaddr *addr, socklen_t addrlen);
```

connect 함수는 세 개의 매개변수와 int형의 반환값을 가진다.

- **sockfd:** 연결할 소켓 디스크립터.
- **addr:** 연결할 대상 정보가 담긴 구조체.
- **addrlen:** 두 번째 매개변수의 크기.

- **반환값:** 함수의 성공 여부.

connect 함수는 **sockfd**를 **addrlen** 크기의 **addr** 구조체 속에 정의된 대상에 연결한다.

**반환값**은 루틴이 성공적으로 수행되면 0, 그렇지 않으면 -1이다.

server-client 네트워크에서 주로 client가 server에 접속하기 위해 사용하는 함수다.

```
// https://man7.org/linux/man-pages/man3/sockaddr.3type.htm
1
#include <sys/socket.h>

struct sockaddr {
    sa_family_t    sa_family;    /* Address family */
    char           sa_data[];    /* Socket address */
};
```

sockaddr 구조체는 위와 같다. **sa\_family**는 통신할 주소 family를 의미하며 socket 함수의 매개변수 **domain**과 동일한 값을 사용한다. **sa\_data**는 IP address와 Port number 등의 소켓에서 사용할 주소를 의미한다.

```
// https://man7.org/linux/man-pages/man3/sockaddr.3type.htm
1
#include <netinet/in.h>

struct sockaddr_in {
    sa_family_t    sin_family;    /* AF_INET */
    in_port_t      sin_port;      /* Port number */
    struct in_addr  sin_addr;     /* IPv4 address */
};

struct in_addr {
    in_addr_t      s_addr;
};
```

```
typedef uint32_t in_addr_t;
typedef uint16_t in_port_t;
```

IPv4 통신의 경우, `sockaddr` 구조체의 `sa_data`를 더욱 자세하게 명시한 `sockaddr_in` 구조체를 사용할 수 있다. 앞선 `sockaddr` 구조체의 `sa_data`가 `sin_port`와 `sin_addr`로 구분되어 있다. 이 때, `sin_addr`은 `in_addr` 구조체 형태로, 4바이트의 `s_addr` 변수를 담고있다.

**sin\_port**에 Port number를 담고 **sin\_addr.s\_addr**에 IP address를 담을 수 있으므로, `sockaddr` 구조체 보다 가독성 좋게 코드를 작성할 수 있다. 따라서, 실습 코드 구현을 위해 `sockaddr_in` 구조체를 이용했다.

```
// https://man7.org/linux/man-pages/man2/bind.2.html
#include <sys/socket.h>

int bind(int sockfd, const struct sockaddr *addr, socklen_t
addrlen);
```

`bind` 함수는 세 개의 매개변수와 `int` 형태의 반환값을 가진다.

- **sockfd**: 특성을 지정할 소켓 디스크립터.
- **addr**: 특성 정보가 담긴 구조체.
- **addrlen**: 두 번째 매개변수의 크기.
- **반환값**: 함수의 성공 여부.

`bind` 함수는 **sockfd**에 **addrlen** 크기의 **addr** 구조체 속에 정의된 특징을 부여한다. 이 때 통신할 주소 family가 `AF_INET`인 경우, `connect` 함수와 마찬가지로 `sockaddr` 구조체 대신 `sockaddr_in` 구조체를 사용할 수 있다.

**반환값**은 루틴이 성공적으로 수행되면 0, 그렇지 않으면 -1이다.

server-client 네트워크에서 주로 server가 사용하며, Port Number를 직접 소켓에 부여하기 위해 사용된다.

```
// https://man7.org/linux/man-pages/man2/listen.2.html
#include <sys/socket.h>

int listen(int sockfd, int backlog);
```

listen 함수는 두 개의 매개변수와 int 형태의 반환값을 가진다.

- **sockfd**: 소켓 디스크립터.
- **backlog**: listen할 queue의 최대 길이.
- **반환값**: 함수의 성공 여부.

bind 함수는 **sockfd**를 accept 함수의 passive로 지정한다. 또한, **backlog** 만큼 accept 할 수 있도록 설정한다.

**반환값**은 루틴이 성공적으로 수행되면 0, 그렇지 않으면 -1이다.

후술할 accept 함수를 통해 받아들일 passive 소켓을 지정하는 역할을 한다. 이 또한, server-client 네트워크에서 주로 server가 사용한다.

```
//https://man7.org/linux/man-pages/man2/accept.2.html
#include <sys/socket.h>

int accept(int sockfd, struct sockaddr *_Nullable restrict
addr, socklen_t *_Nullable restrict addrlen);
```

accept 함수는 세 개의 매개변수와 int 형태의 반환값을 가진다.

- **sockfd**: 소켓 디스크립터.
- **addr**: 연결된 소켓의 특징을 담은 구조체.
- **addrlen**: 두 번째 매개변수의 크기.
- **반환값**: 연결된 소켓의 디스크립터.

accept 함수는 **sockfd**에 연결 요청이 온 소켓을 받아들이는 역할을 수행한다. 이 때, **addrlen** 크기의 **addr** 구조체를 통해 연결 요청이 온 소켓의 특징을 지정할 수 있으며, Null로써 비워 놓을 수도 있다.

**반환값**은 연결에 성공하면 연결 요청한 소켓의 디스크립터, 오류가 발생하면 -1이다.

server-client 네트워크에서 주로 server가 client의 connect 요청을 받아들이고 해당 소켓에 대한 디스크립터를 얻기 위해 사용된다.

```
// https://www.man7.org/linux/man-pages/man2/send.2.html
#include <sys/socket.h>
```

```
ssize_t send(int sockfd, const void buf[.len], size_t len,
int flags);
```

send 함수는 네 개의 매개변수와 ssize\_t 형태의 반환값을 가진다.

- **sockfd**: 소켓 디스크립터.
- **buf[.len]**: 전달할 데이터가 담긴 buf의 주소.
- **len**: 전달할 데이터의 크기.
- **flags**: 다양한 옵션들의 사용 유무.
- **반환값**: 전달된 데이터의 크기.

send 함수는 **sockfd**에 해당하는 소켓을 통해 **buf**에서 **len**만큼의 데이터를 전달한다. 이 때, **flags**를 설정해 다양한 옵션을 사용할 수 있다.

**반환값**의 경우 Error가 발생하면 -1, 그렇지 않으면 전달에 성공한 데이터의 크기를 의미한다.

```
// https://www.man7.org/linux/man-pages/man2/recv.2.html
#include <sys/socket.h>

ssize_t recv(int sockfd, void buf[.len], size_t len, int flags);
```

recv 함수는 네 개의 매개변수와 ssize\_t 형태의 반환값을 가진다.

- **sockfd**: 소켓 디스크립터.
- **buf[.len]**: 받을 데이터를 저장한 buf의 주소.
- **len**: 받을 데이터의 크기.
- **flags**: 다양한 옵션들의 사용 유무.
- **반환값**: 받아진 데이터의 크기.

recv 함수는 **sockfd**에 해당하는 소켓을 통해 **buf**에서 **len**만큼의 데이터를 받는다. 이 때, **flags**를 설정해 다양한 옵션을 사용할 수 있다.

**반환값**의 경우 Error가 발생하면 -1, 그렇지 않으면 성공적으로 받아진 데이터의 크기를 의미한다.

```
// https://man7.org/linux/man-pages/man2/close.2.html
#include <unistd.h>

int close(int fd);
```

close 함수는 한 개의 매개변수와 int 형태의 반환값을 가진다.

- **fd**: 파일 디스크립터.
- **반환값**: 함수의 성공 여부.

close 함수는 **fd**에 해당하는 파일 디스크립터를 닫아서 더 이상 사용하지 않는 형태로 만든다.

**반환값**의 경우 루틴이 성공적으로 수행되면 0, 그렇지 않으면 -1이다.

```
// https://www.man7.org/linux/man-pages/man2/shutdown.2.html
#include <sys/socket.h>

int shutdown(int sockfd, int how);
```

shutdown 함수는 두 개의 매개변수와 int 형태의 반환값을 가진다.

- **sockfd**: 소켓 디스크립터.
- **반환값**: 함수의 성공 여부.

shutdown 함수는 **sockfd**에 대한 소켓 연결을 끊는다. 이 때, **how**(SHUT\_RD(수용 차단), SHUT\_WR(전달 차단), SHUT\_RDWR(수용 및 전달 차단)) 매개 변수를 통해 소켓의 기능을 선택적으로 끊을 수 있다.

**반환값**은 루틴이 성공적으로 수행되면 0, 그렇지 않으면 -1이다.

## Client

1. client 소켓 생성 후 server에 connect하는 루틴.

2. 문자열을 RESP 데이터로 변환하거나 RESP 데이터를 문자열로 파싱하는 루틴.

3. 사용자에게 입력을 받고 server와 상호작용 하는 루틴.

client는 사용자의 입력을 RESP 데이터로 변환해 server에 전달하고, server로 받은 RESP 데이터를 ASCII 데이터로 파싱해 사용자에게 출력해야 한다. 해당 루틴을 수행하기 위해 위의 세 가지 루틴들이 필요하다. 각각의 루틴들에 대해 기술하겠다.

### 1. client 소켓 생성 후 server에 connect하는 루틴.

server와 통신하려면 소켓을 생성해 server에 연결해야 한다. socket 함수를 통해 소켓을 생성하고, connect 함수를 통해 해당 소켓을 server에 연결하도록 했다.

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

    // Process Socket
    int clientSocket = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);
    if(clientSocket < 0){
        perror("***socket API error***\n");
        return -1;
    }

    ...

}
```

socket 함수를 통해 clientSocket에 소켓을 생성하도록 했다. 과제를 위해선 tcp 프로토콜을 사용해야 하므로, **domain**은 임의로 IPv4를 의미하는 "AF\_INET", **type**은 신뢰적 데이터 전송을 지원하는 "SOCK\_STREAM", 그리고 **protocol**은 tcp를 의미하는 "IPPROTO\_TCP"를 선택했다. 추가로, 반환값의에 따라 예외 처리 하도록 코딩했다.

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

    ...

}
```



```

    struct sockaddr_in sockStruct;
    memset(&sockStruct, 0, sizeof(struct sockaddr_in));

    sockStruct.sin_family = AF_INET;
    sockStruct.sin_port = htons(atoi(argv[2]));

    if(inet_pton(AF_INET, argv[1], &sockStruct.sin_addr.s_a
ddr) != 1){
        perror("***inet_pton API error***\n");
        close(clientSocket);
        return -1;
    }

    ...

}

```

connect 함수에 사용할 구조체의 값을 결정해야 한다. 구현할 통신은 AF\_INET 통신 family를 사용하므로, sockaddr\_in 구조체를 사용하며, 구조체 이름은 sockStruct로 결정했다.

sockStruct의 sin\_family는 주소 family가 들어가야 하므로, IPv4를 의미하는 AF\_INET을 저장한다.

sockStruct의 sin\_port에는 포트 번호가 들어가야 한다. 포트 번호는 command-line의 두 번째 매개변수(argv[2])로 전달 받는다. argv[2]는 문자열이므로, atoi 함수를 통해 정수로 바꾸고 htons 함수를 통해 호스트 바이트 순서를 네트워크 바이트 순서로 바꿔 저장한다.

sockStruct의 sin\_addr 구조체 속 s\_addr에는 IP 주소가 들어가야 한다. IP 주소는 command-line의 첫 번째 매개변수(argv[1])로 전달 받는다. argv[1]도 문자열이므로, 정수로 바꾼 후 네트워크 바이트 순서로 바꿔 저장해야 한다. inet\_pton 함수에 주소 family와 문자열, 변경한 값을 저장할 주소를 전달하면, 주소 family에 맞게 문자열을 변경해 저장해 준다. 이를 통해 argv[1]을 변경해서 sockStruct.sin\_addr.s\_addr에 IP 주소를 저장한다. 이 때, inet\_pton 함수의 반환값에 따라 예외 처리 하도록 코딩했다.

```

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

```

...

```

        if(connect(clientSocket, (struct sockaddr *)&sockStruct,
        sizeof(sockStruct)) < 0){
            perror("***connect API error***\n");
            close(clientSocket);
            return -1;
        }

        ...

    }

```

connect 함수의 **sockfd**는 clientSocket, **addr**은 sockaddr로 타입 캐스팅한 sockStruct, 그리고 **addrlen**은 sockStruct 구조체의 크기로 설정해서 clientSocket이 server에 연결되도록 했다. 추가로, 반환값의 종류에 따라 예외 처리 하도록 코딩했다.

위의 과정을 통해 소켓을 생성하고 해당 소켓을 server에 연결함으로써 server와 통신 할 준비를 마쳤다.

## 2. 문자열을 RESP 데이터로 변환하거나 RESP 데이터를 문자열로 파싱하는 루틴.

server와 상호 작용하기 위해선 사용자에게 받은 ASCII 데이터를 RESP 데이터로, 서버에게서 받은 RESP 데이터를 ASCII 데이터로 파싱할 수 있어야 한다. 따라서 각각의 동작을 수행하는 transAscii(ASCII to RESP) 함수와 parseResp(RESP to ASCII) 함수를 구현했다.

### transAscii(ASCII to RESP)

transAscii 함수는 사용자에게 입력 받은 명령어를 RESP 프로토콜 형태로 변환하는 역할을 수행한다. 이 때, 허용되지 않은 명령어에 대한 예외 처리를 같이 수행한다.

추가로, **transAscii** 함수는 전달된 문자열이 **Null Termination** 됐다고 가정하며, 전달할 문자열을 **Null Termination** 한다. 자세한 구현 사항은 다음과 같다.

```

int transAscii(char* buf, char* comp){

```

```

...

// Process Set Command
if(strncmp(buf, "set", 3)==0){

    ...

}

// Process Get Command
if(strncmp(buf, "get", 3)==0){

    ...

}

// Process EXIT Command
else if(strncmp(buf, "EXIT\0", sizeof(5))==0){
    strncpy(comp, "*1\r\n$4\r\nQUIT\r\n\0", 15);
    return 0;
}

// Process Error Exception
else{
    printf("(error) ERR unknown command '%s'\n", buf);
    return -1;
}

}

```

transAscii 함수는 매개변수 중 buf를 통해 ASCII 데이터를 전달 받고, comp를 통해 RESP 프로토콜 형태로 변환한 데이터를 전달한다.

transAscii 함수는 문자열 비교를 통해 명령어의 종류를 파악한다. set, get, 그리고 EXIT 인지 여부를 살펴보고 모두 아니라면 Error로 간주해 예외 처리한다.

EXIT인 경우, 곧바로 RESP 형태의 종료 명령어를 comp에 저장해 전달하도록 한 후 0을 반환한다. Error인 경우, 에러 표시와 함께 전달 받은 ASCII 데이터를 출력한 후 -1을 반환한다.

```

unsigned int countSp(const char *str, unsigned int *ptr) {

    // var For check ptr's idx
    unsigned int countNum = 0;
    // var For check sp's place
    unsigned int countPtr = 0;

    while (*str != '\0') {

        if (*str == ' ') {

            if(countNum < 2) ptr[countNum] = countPtr;
            countNum++;
        }
        countPtr++;
        str++;
    }
    return countNum;
}

```

set과 get 명령어를 처리할 때는 전달 받은 문자열의 SP(띄어쓰기) 개수를 세는 countSp 함수가 사용된다. 구현 사항은 다음과 같다.

매개변수로 문자열 str과 SP의 위치를 정수로 기록할 ptr 배열을 전달 받는다. 루프를 통해 str이 Null을 가리킬 때까지 str을 1을 더한다.

루프를 돌 때, 만약 str이 SP를 가리키면 countNum을 1 증가 시켜 countNum을 통해 SP의 개수를 저장한다.

루프를 돌 때, str을 1 증가 시킬 때 countPtr도 1 증가 시킨다. 만약 str이 SP를 가리키고 countNum이 2보다 작으면 ptr의 countNum 인덱스에 countPtr을 저장해서 SP의 위치를 정수 형태로 ptr에 저장한다. 이 때 ptr에 SP의 위치를 최대 두 번 저장하는 이유는, 가능한 명령어 중 가장 SP가 많은 경우(set 명령어)에 SP가 두 개이기 때문이다.

모든 동작이 끝나면 SP의 개수인 countNum을 리턴한다.

모든 루틴이 종료되면, countNum에는 SP의 개수가, 그리고 ptr 배열에는 각각의 SP들에 대한 위치 정보가 저장되게 된다.

```

        unsigned int ptr[3];

        // Process Set Command
        if(strncmp(buf, "set", 3)==0){

            if(countSp(buf, ptr) != 2){
                printf("(error) ERR wrong number of arguments f
or 'set' command\n");
                return -1;
            }

            // Ex: set(ptr[0])key(ptr[1])value(ptr[2])
            ptr[2] = strlen(buf);

            // Slice Strings
            // Ex: set(\0)key(\0)value\0
            int i;
            for(i = 0; i < 2; i++) buf[ptr[i]] = '\0';

            strncpy(comp, "*3\r\n$3\r\nSET\r\n\0", 14);
            sprintf(comp+strlen(comp), "$%d\r\n%s\r\n\0", ptr
[1]-ptr[0]-1, buf+ptr[0]+1);
            sprintf(comp+strlen(comp), "$%d\r\n%s\r\n\0", ptr
[2]-ptr[1]-1, buf+ptr[1]+1);

            return 1;

        }

```

set 명령어일 경우 루틴은 위와 같다.

일단 첫 3바이트 값이 set인 게 확인됐으므로, countSp 함수를 통해 SP 개수가 두 개인지 확인한다. 그렇지 않다면 set 명령어를 잘못 사용한 것이므로, error 메시지를 띄우고 -1을 반환한다.

이후, strlen을 통해 ptr[2]가 문자열의 끝을 가리키도록 한다.(Ex: set(ptr[0])key(ptr[1])value(ptr[2])) 그리고 문자열의 모든 ptr 위치를 Null로 구분한다. (Ex: set(\0)key(\0)value\0)

마지막으로, set 명령어에 공통적으로 들어가는 RESP 구문을 comp에 작성하고 key와 value 문자열을 RESP 구문 형태로 바꿔 comp에 붙여 넣는다. 이 때, Null Termination에 의해 **comp+strlen(comp)**는 기존 comp의 널 위치를 가리킨다. 또한, **ptr[n]-ptr[n-1]-1**은 n번째 문자열의 길이를 의미하게 되고 **buf+ptr[n]+1**은 n번째 문자열의 시작 위치를 가리키게 된다. 따라서, strncpy와 sprintf를 통해 RESP 형태의 문자를 comp에 작성할 수 있다.

모든 동작이 끝나면 1을 반환한다.

```
unsigned int ptr[3];

// Process Get Command
if(strncmp(buf, "get", 3)==0){

    if(countSp(buf, ptr) != 1){
        printf("(error) ERR wrong number of arguments f
or 'get' command\n");
        return -1;
    }

    // Ex: get(ptr[0])key(ptr[1])
    ptr[1] = strlen(buf);

    // Slice Strings
    // Ex: get(\0)key\0
    buf[ptr[0]] = '\0';

    strncpy(comp, "*2\r\n$3\r\nGET\r\n\0", 14);
    sprintf(comp+strlen(comp), "%d\r\n%s\r\n\0", ptr
[1]-ptr[0]-1, buf+ptr[0]+1);

    return 1;

}
```

get 명령어일 경우 루틴은 위와 같다. 동작은 set 명령어일 경우와 동일 한다. 다만, get 명령어는 문자열 두 개가 아닌 한 개를 이용한다는 점을 고려해 SP의 개수가 1이 아닐 경우에

Error 예외 처리, 그리고 ptr은 두 개만 사용한다. set과 마찬가지로 모든 동작이 끝나면 1을 반환한다.

```
// return -1: Error
// return 0: EXIT
// return 1: GET || SET
```

반환값들을 고려했을 때, -1인 경우 Error임을, 0인 경우 EXIT 명령어임을, 그리고 1인 경우 GET 또는 SET 명령어임을 알 수 있다.

### parseResp(Resp to ASCII)

parseResp 함수는 server로부터 전달받은 RESP 데이터를 ASCII 형태로 파싱하는 역할을 수행한다. 이 때, server로부터 인식할 수 없는 데이터를 전달 받을 경우 예외 처리를 수행한다.

추가로, **parseResp** 함수는 전달된 문자열이 **Null Termination** 되지 않았다고 가정한다. 자세한 구현 사항은 다음과 같다.

```
int parseResp(char* buf){

    // Parse Bulk Strings
    if(*buf == '$'){

        ...

    }
    // Parse Simple Strings
    else if(strncmp(buf, "+OK\r\n", 5)==0){
        printf("OK\n");
        return 0;
    }
    // Process Error Exception
    else{
        printf("***Unintelligible Response from Server***
\n");
    }
}
```

```

        return -1;
    }

}

```

parseResp 함수는 파싱할 RESP 데이터가 담긴 포인터 buf를 매개변수로 전달받는다. server로부터 전달된 RESP 데이터는 총 세 가지 형태다. \$로 시작하는 Bulk Strings, +로 시작하는 Simple Strings, 그리고 Error다. 문자열 비교를 통해 문자열의 종류를 구분한다.

\$로 시작하는 경우, Bulk Strings와 관련된 루틴을 수행한다.

+로 시작하는 경우, OK response일 경우밖에 없으므로 전달 받은 문자열이 "+OK\r\n"인지 체크한다. OK response가 맞다면 OK를 출력하고 0을 반환한다.

Bulk Strings 또는 OK response가 아닐 경우, 예외 처리를 위해 에러 메시지를 출력하고 -1을 반환한다.

```

// Parse Bulk Strings
if(*buf == '$'){

    if(strncmp(buf, "$-1", 3) == 0){
        printf("(nil)\n");
        return 1;
    }
    else{

        // Ex: $4\r\ntest\r\n => test\r\n
        while(*buf != '\r') buf++;
        buf += 2;

        // backup To print out
        char* backup = buf;

        // Ex: test\r\n => \r\n
        while(*buf != '\r') buf++;

        // Ex: $4\r\ntest\r\n => $4\r\ntest\0\n
        *buf = '\0';
    }
}

```



```

        printf("\\\"%s\\\"\\n", backup);

        return 1;
    }

}

```

\$로 시작한 Bulk Strings의 가능한 경우의 수는 둘로, "\$-1"인 경우와 "\$-1"이 아닌 경우로 나뉜다.

"\$-1" 경우에는 Null 값을 의미하므로, (nil)을 출력하고 1을 반환한다.

"\$-1"이 아닌 경우에는 특정 문자열이 전달된 것이다. 따라서, buf를 "\r"을 찾을 때까지 증가시킨 후, 추가로 2를 더해 문자열을 가리키도록 한다.(Ex: \$4\r\ntest\r\n ⇒ test\r\n) 그리고 backup에 buf를 저장한 후 다시 "\r"을 찾을 때까지 buf를 증가시킨다. buf가 "\r"일 때, 해당 "\r"을 Null로 바꾼다.(Ex: (backup)test\r\n ⇒ (backup)test\0\n) 이 과정을 통해 문자열의 끝에 Null Termination이 이루어지므로, backup을 %s로 출력한다. 이후 1을 반환한다.

```

// return -1: ERROR
// return 0: Simple Strings
// return 1: Bulk Strings

```

반환값들을 고려했을 때, -1인 경우 Error임을, 0인 경우 Simple Strings(OK response)임을, 그리고 1인 경우 Bulk Strings임을 알 수 있다.

위의 두 과정을 통해 server와 상호 작용할 준비를 마쳤다.

### 3. 사용자에게 입력을 받고 server와 상호작용 하는 루틴.

앞서 구현한 함수들과 했던 루틴을 기반으로 server와 상호 작용할 수 있게 루틴을 구현했다.

```

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

```

```

...

// Server Interaction
for(;;){

    int res = 0;

    // flag For transAscii
    int flag1 = 0;
    // flag For parseResp
    int flag2 = 0;

    char buf[BUF_SIZE];
    char comp[BUF_SIZE];

    fgets(buf, BUF_SIZE, stdin);
    buf[strcspn(buf, "\n")] = '\0';

    ...

}

...

}

```

send와 recv 함수의 반환값을 저장할 res, transAscii와 parseResp의 반환값을 각각 저장할 flag1과 flag2, 그리고 사용자에게 입력 받은 문자열을 저장할 buf와 server에 보낼 RESP 데이터를 저장할 comp 배열을 선언한다.

이후, fgets 함수를 통해 사용자가 엔터를 입력할 때까지 buf에 데이터를 받고 “\n”를 Null로 바꿔 해당 ASCII 데이터를 Null Termination한다.

```

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

    ...

    for(;;){

```

```

        ...

        flag1 = transAscii(buf, comp);
        if(flag1 == -1) continue;

        res = send(clientSocket, comp, strlen(comp), 0);
        if(res <= 0){
            perror("***send API error***\n");
            close(clientSocket);
            return -1;
        }

        ...

    }

    ...

}

```

먼저, transAscii 함수를 통해 buf 속 문자열을 RESP 데이터로 변환해 comp에 저장한다. 그리고 transAscii 함수의 반환값을 flag1 변수에 저장한다. 이 때, transAscii 함수의 실행 중 오류가 발생한 경우 예외 처리를 수행하도록 한다.

이후, send 함수로 **clientSocket**을 통해 server에 **comp**의 데이터를 **strlen(comp)**만큼 전달하도록 한다. 이 때, **flags** 값은 설정하지 않는다. 그리고 send 함수를 통해 아무런 값도 전달되지 않거나 실행 중 오류가 발생할 경우 예외 처리를 수행하도록 한다.

```

// One byte for Null Termination
#define BUF_SIZE 1024+1

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

    ...

    for(;;){

```

```

...

    res = recv(clientSocket, buf, BUF_SIZE, 0);
    if(res <= 0){
        perror("***recv API error***\n");
        close(clientSocket);
        return -1;
    }
    buf[res] = '\0';

    flag2 = parseResp(buf);
    // ERROR or EXIT
    if(flag2 == -1 || (flag1 == 0 && flag2 == 0)){
        break;
    }

}

close(clientSocket);
return 0;

}

```

먼저, recv 함수로 **clientSocket**을 통해 server로 부터 전달된 데이터를 **BUF\_SIZE** 만큼 받아들이며 **buf** 배열에 저장한다. 이 때, **flags** 값은 설정하지 않는다. 그리고 recv 함수를 통해 아무런 값도 받아지지 않았거나 실행 중 오류가 발생할 경우 예외 처리를 수행하도록 한다. 데이터를 잘 전달 받았다면 전달 받은 데이터의 끝에 Null Termination을 수행 한다.

이후, parseResp 함수를 통해 buf를 파싱하여 사용자에게 출력을 한 후, 반환값을 flag2 변수에 저장한다. flag2가 만약 -1이거나(**parseResp 함수의 Error**), flag1과 flag2가 모두 0이면(**사용자가 EXIT 명령어 입력 및 server로부터의 OK response**) 무한 루프를 빠져나와 clientSocket을 닫고 main 함수를 종료한다.

앞서 언급한 루틴들을 모두 종합해 client의 루틴을 구현할 수 있다.

# Server

1. server 소켓 생성 후 client를 받아들일 준비를 하는 루틴.
2. client를 받아들인 후, client와 상호 작용하는 thread를 생성하는 루틴.
3. KVS(key-value stores) 관련 동작을 수행하는 루틴.
4. client와 상호 작용하는 thread 함수의 루틴.

server는 client로부터 받은 RESP 데이터를 파싱해 동작을 수행하고, 동작 결과를 RESP 데이터 형태로 client에게 전달해야 한다. 해당 루틴을 수행하기 위해 위의 네 가지 루틴들이 필요하다. 각각의 루틴들에 대해 기술하겠다.

## 1. server 소켓 생성 후 client를 받아들일 준비를 하는 루틴.

client와 통신하려면 소켓을 생성해 client의 연결 요청을 받아들여야 한다. socket 함수를 통해 소켓을 생성하고, bind와 listen 함수를 통해 client의 연결 요청을 받아들일 준비를 할 수 있다.

```
int main(int argc, char *argv[]){  
  
    ...  
  
    int serverSocket = socket(AF_INET, SOCK_STREAM, IPPROTO  
_TCP);  
    if(serverSocket < 0){  
        perror("***socket API error ***\n");  
        return -1;  
    }  
  
    ...  
  
}
```

socket 함수를 통해 serverSocket 소켓을 생성한다. client와 통신해야 하므로, client에서 소켓을 생성할 때와 동일하게 **domain**은 "AF\_INET(IPv4)", **type**는 "SOCK\_STREAM(신뢰적 데이터 전송)", 그리고 **protocol**은 "IPPROTO\_TCP"로 설정한다. 마찬가지로 예외 처리도 수행한다.

```
int main(int argc, char *argv[]){  
    ...  
  
    struct sockaddr_in sockStruct;  
    memset(&sockStruct, 0, sizeof(struct sockaddr_in));  
  
    sockStruct.sin_family = AF_INET;  
    sockStruct.sin_port = htons(atoi(argv[1]));  
    sockStruct.sin_addr.s_addr = htonl(INADDR_ANY);  
    ...  
}
```

bind 함수에 사용할 구조체의 값을 결정해야 한다. 구현할 통신은 AF\_INET 통신 family를 사용하므로, sockaddr\_in 구조체를 사용하며, 구조체 이름은 sockStruct로 결정했다.

sockStruct의 sin\_family는 주소 family가 들어가야 하므로, IPv4를 의미하는 AF\_INET을 저장한다.

sockStruct의 sin\_port에는 포트 번호가 들어가야 한다. 포트 번호는 command-line의 첫 번째 매개변수(argv[1])로 전달 받는다. argv[1]은 문자열이므로, atoi 함수를 통해 정수로 바꾸고 htons 함수를 통해 호스트 바이트 순서를 네트워크 바이트 순서로 바꿔 저장한다.

sockStruct의 sin\_addr 구조체 속 s\_addr에는 IP 주소가 들어가야 한다. IP 주소의 경우, 프로그램이 동작할 PC에 존재하는 사용가능한 랜카드의 IP 주소를 의미하는 INADDR\_ANY를 사용한다. INADDR\_ANY의 경우 정수이므로, htonl 함수를 통해 호스트 바이트 순서를 네트워크 바이트 순서로 바꿔 저장한다.

(이 때, htons 함수는 2바이트 정수의 바이트 오더를 변경하는 함수며, htonl은 4바이트 정수의 바이트 오더를 변경하는 함수다.)

```

// Define Max Client Num
#define MAX_CLIENT 10

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

    ...

    if(bind(serverSocket, (struct sockaddr *)&sockStruct,
sizeof(sockStruct)) < 0){
        perror("***bind API error***\n");
        close(serverSocket);
        return -1;
    }

    if(listen(serverSocket, MAX_CLIENT) < 0){
        perror("***listen API error***\n");
        close(serverSocket);
        return -1;
    }

    ...

}

```

bind 함수에 **serverSocket**과 **socktStruct**, 그리고 **sockStruct**의 크기를 전달해서 serverSocket의 IP 주소와 포트 번호를 설정한다. 반환값에 따라 예외 처리를 수행한다.

listen 함수에 **serverSocket**과 **MAX\_CLIENT**를 전달해서 serverSocket을 후술할 accept 함수의 passive 소켓으로 설정함과 동시에 최대로 받아들일 client의 수를 MAX\_CLIENT로 제한한다. 반환값에 따라 예외 처리를 수행한다.

위의 과정을 통해 client를 받아들일 준비 과정을 마쳤다.

## 2. client를 받아들인 후, client와 상호 작용하는 thread를 생성하는 루틴.

client를 받아들인 후 thread를 생성하는 과정을 무한 루프를 통해 수행해 다중 client를 지원한다. 이 때, 무한 루프를 끝낼 적당한 수단이 필요하다. 이를 위해 signal 함수와 select 함수를 이용했는데, 해당 함수들의 루틴을 먼저 기술한 후 무한 루프 속 루틴을 기술하겠다.

## signal 함수

```
// https://man7.org/linux/man-pages/man2/signal.2.html
#include <signal.h>

typedef void (*sighandler_t)(int);

sighandler_t signal(int signum, sighandler_t handler);

SIGINT(signum): ctrl+c
SIG_IGN(handler): 해당 시그널 무시
```

signal 함수는 프로그램 실행 중 발생할 수 있는 signal을 다루는 함수다. signum이라는 시그널이 발생했을 때, 기존에 정의된 동작이 아닌 handler로 정의된 루틴을 수행하도록 해 준다.

main 함수 속 무한 루프 루틴을 ctrl+c(프로그램 중단) 신호를 기준으로 처리할 것으로, signal 함수를 통해 기존의 동작(프로그램 종료)을 무시하고 후술할 select 함수의 예외처리 과정을 이용할 계획이다.

## select 함수

```
// https://man7.org/linux/man-pages/man3/FD_CLR.3.html
#include <sys/select.h>

typedef /* ... */ fd_set;

int select(int nfd, fd_set *_Nullable restrict readfds,
           fd_set *_Nullable restrict writefds,
           fd_set *_Nullable restrict exceptfds,
           struct timeval *_Nullable restrict timeout);

EINTR(errno): 인터럽트에 의해 프로그램이 중단됨을 의미.
```

select 함수는 다섯 개의 매개변수와 int 형 반환값을 가진다.

- **nfd**: fd\_set 속 파일 디스크립터의 최대 개수.



- **readfds:** read 행위를 관찰할 fd\_set.
- **writefds:** write 행위를 관찰할 fd\_set.
- **exceptfds:** 예외 처리를 관찰할 fd\_set.
- **timeout:** 시간 제한 정보를 담을 수 있는 timeval 구조체.
- **반환값:** 파일 디스크립터.

select 함수는 **fd\_set** 속 특정 동작을 모니터 해 특정 디스크립터에 특정 동작이 수행됐는지 확인할 수 있는 함수다. **반환값**으로 관찰한 동작이 수행된 파일 디스크립터, timeout이 발생하면 0, 에러가 발생하면 -1을 전달한다. 이 때, 자세한 에러 정보는 errno에 저장된다.

```
// https://man7.org/linux/man-pages/man3/FD_CLR.3.html
struct timeval {
    time_t      tv_sec;          /* seconds */
    suseconds_t tv_usec;        /* microseconds */
};
```

timval 구조체는 위와 같이 생겼다. 시간을 sec 단위와 microsec 단위로 저장할 수 있는 구조체다.

```
#define __FD_SETSIZE      1024
typedef long int __fd_mask;
#define __NFDBITS        (8 * (int) sizeof (__fd_mask))

typedef struct
{
    __fd_mask fds_bits[__FD_SETSIZE / __NFDBITS];
} fd_set;
```

<https://blog.naver.com/tipsware/220810795410>

fd\_set의 구조는 파일 디스크립터 정보를 담고 있다. 파일 디스크립터가 0부터 시작한다는 점을 통해 bit mask를 통해 메모리 공간을 최소화한 구조체다.

```
int  FD_ISSET(int fd, fd_set *set);
void FD_SET(int fd, fd_set *set);
void FD_ZERO(fd_set *set);
```

fd\_set과 관련된 매크로 함수는 위와 같다.

**FD\_ZERO**는 전달 받은 fd\_set을 초기화하는 동작을, **FD\_SET**은 전달 받은 fd\_set에 fd를 저장하는 동작을, 그리고 **FD\_ISSET**은 fd\_set 속 fd의 동작 여부(동작시 1을 반환)를 반환하는 동작을 수행한다.

select 함수를 통해 accept에 시간 제한을 걸고 ctrl+c 동작이 수행되면 예외 처리를 통해 무한 루프를 끝내도록 구현할 계획이다.

**client를 받아들인 후, client와 상호 작용하는 thread를 생성하는 루틴.**

```
void stopInt(int sig){
    signal(sig, SIG_IGN);
}

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

    int threadHandle;

    int res;
    fd_set reads, temps;
    struct timeval timeout;

    FD_ZERO(&reads);
    FD_SET(serverSocket, &reads);

    signal(SIGINT, stopInt);
    printf("Ctrl + C for Stop Program\n");

    ...
}
```

```
}
```

무한 루프에 진입하기 전, thread를 다루기 위한 threadHandle, select 함수의 반환값을 저장할 res, fd\_set reads와 temps, 그리고 timeval timeout을 선언한다. 이 때, fd\_set을 두 개 선언한 이유는 select 함수에 fd\_set을 전달하면 해당 fd\_set을 직접 수정하기 때문에 backup이 필요하다.

FD\_ZERO를 통해 reads를 초기화하고, FD\_SET을 통해 serverSocket을 reads에 넣는다. 이후, signal 함수를 실행해 SIGINT(ctrl+c) 동작이 발생하면 stopInt 함수를 실행하도록 설정한다. stopInt 함수는 전달 받은 SIGINT를 SIG\_IGN(시그널 무시)하도록 설정한다.

```
int main(int argc, char *argv[]){  
  
    ...  
  
    for(;;){  
  
        temps = reads;  
        timeout.tv_sec = 0;  
        timeout.tv_usec = 500;  
  
        res = select(serverSocket+1, &temps, 0, 0, &timeout);  
  
        ...  
    }  
  
    ...  
}
```

무한 루프에 진입하면, 먼저 backup fd\_set인 temps에 reads를 저장하고 timeout을 500 마이크로세크로 설정한다.(실험 결과 반응 속도가 제일 적당하다고 판단했습니다.) 이후, select 함수에 **serverSocket+1**을 전달해 serverSocket까지 관찰하게 하고, **readfds**에 temps를, **timeout**에 timeout 구조체를 전달해 serverSocket의 read(accept) 행위를 관찰하게 한다. select의 반환값은 res에 저장한다.

```

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

    ...

    for(;;){

        ...

        if(res < 0){

            // ctrl + c
            if(errno == EINTR){
                break;
            }
            // Error
            else{
                perror("***select API error***\n");
                close(serverSocket);
                return -1;
            }
        }

    }

    ...

}

```

만약 res가 0보다 작으면 에러가 발생했다는 의미다. 이 때, errno을 확인한다.

errno가 EINTR일 경우, ctrl+c 인터럽트가 실행된 것이므로 무한 루프를 그대로 종료한다.  
(앞서 signal 함수를 통해 SIGINT를 SIG\_IGN 했으므로, 실제로 프로그램이 종료되진 않는다.)

errno가 EINTR가 아닐 경우, select 함수 내부에서 에러가 발생한 것이므로 예외 처리를 진행한다.

```

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

    ...

    for(;;){

        ...

        // timeout
        else if(res == 0){
            continue;
        }
        else{
            if(FD_ISSET(serverSocket, &temps)){

                int clientSocket = accept(serverSocket, NULL, NULL);

                if(clientSocket < 0){
                    perror("***accept API error***\n");
                    close(serverSocket);
                    return -1;
                }

                pthread_t thread;
                if(pthread_create(&thread, NULL, cliInteract, (void*)&clientSocket) < 0){
                    perror("***pthread_create API error***\n");

                    close(serverSocket);
                    return -1;
                }

                if(pthread_detach(thread) != 0){
                    perror("***pthread_detach API error***\n");

                    close(serverSocket);
                    return -1;
                }
            }
        }
    }
}

```

```

    }
}

...

close(serverSocket);
return 0;

}

```

만약 res가 0이면 timeout이 발생했다는 뜻이다. 따라서, 무한 루프를 다시 수행하도록 한다.

만약 res가 0보다 크다면 temps fd\_set 속 파일 디스크립터에 reads가 발생했다는 의미다. 따라서, FD\_ISSET을 통해 동작이 발생한 파일 디스크립터가 serverSocket임을 확인한다. 반환값이 1이라면 serverSocket에 reads가 발생했다는 뜻이고, 이는 곧 client로 부터 connect 요청이 왔다는 의미다.

따라서 accept 함수에 **serverSocket**을 전달해 clientSocket에 연결된 client의 디스크립터를 저장하고 반환값에 대한 예외 처리를 한다. 그리고 pthread\_create 함수에 **thread**와 **clientSocket** 값을 넘겨 **cliInteract** 함수를 실행하는 thread를 생성하고 반환값에 대한 예외 처리를 한다. 이 후, pthread\_detach 함수에 **thread**를 넘겨 해당 thread가 main 함수와 개별 동작을 수행하도록 한 후 반환값에 대한 예외 처리를 한다.

사용자의 ctrl+c에 의해 무한 루프가 종료되면 serverSocket을 닫고 main 함수를 종료한다.

위의 루틴들을 통해 server가 client로 부터 connect 요청이 왔을 때, 해당 client를 accept하고 그 client를 대상으로 동작하는 thread를 생성해 상호 작용하며, 사용자로부터 ctrl+c가 발생하면 프로그램을 종료하는 것을 구현할 수 있다.

### 3. KVS(key-value stores) 관련 동작을 수행하는 루틴.

KVS 시스템은 값을 저장할 구조체를 전역 변수로 선언하고 다양한 함수들을 통해 thread들이 구조체에 접근할 수 있도록 구현했다.

**KVS 시스템에 문자열을 저장할 때는 동적 할당된 문자열의 포인터가 전달된다고 가정한다.**

```
// global variable
unsigned int kvs_cnt = 0;
struct kvs{
    char* key;
    char* value;
};
struct kvs KVS[10];
```

KVS 시스템에서 사용할 kvs 구조체의 형태는 위와 같다. 구조체 내부에는 key 문자열을 가리킬 key 포인터와 value 문자열을 가리킬 value 포인터가 존재한다. 과제의 구현 사항에서 최대 10개의 key, value를 저장해야 하므로, kvs 구조체를 10개 가진 kvs 구조체 배열인 KVS를 선언했다.

값이 채워진 kvs 구조체를 구분하기 위해 인덱스를 저장할 kvs\_cnt 변수도 선언했다.

```
void KVSinit(){

    int i;
    for(i = 0; i < 10; i++){
        memset(&KVS[i], 0, sizeof(struct kvs));
    }
}

void KVSfin(){

    int i;
    for(i = 0; i < kvs_cnt; i++){
        free(KVS[i].key);
        free(KVS[i].value);
    }
}
```

KVSinit 함수는 구조체를 사용하기 전 값들을 모두 초기화하는 함수다. 루프를 통해 모든 KVS 인덱스에 접근해 kvs 구조체의 크기만큼 0으로 초기화한다.

KVSfin 함수는 구조체 속 포인터를 모두 반환하는 함수다. 루프를 통해 kvs\_cnt의 크기까지 KVS[i] 구조체 속 key와 value의 값을 free해준다.

```

int KVSfind(char* key){

    int i;
    for(i = 0; i < kvs_cnt; i++){
        if(strncmp(KVS[i].key, key, strlen(key))==0){
            return i;
        }
    }
    return -1;
}

```

KVSfind 함수는 key 포인터를 매개변수로 전달 받는다. KVSfind 함수는 전달 받은 key를 지닌 구조체가 있는 지 확인하는 함수다.

루프를 통해 kvs\_cnt 인덱스까지 KVS 구조체 배열을 돌면서 전달 받은 key 값을 지닌 구조체가 존재하는 지 확인한다. 존재한다면 해당 구조체의 인덱스를, 그렇지 않다면 -1을 반환한다.

```

void KVSset(char *key, char *value){

    int idx = KVSfind(key);
    if(idx != -1){
        free(KVS[idx].value);
        KVS[idx].value = value;
    }
    else{
        KVS[kvs_cnt].key = key;
        KVS[kvs_cnt].value = value;
        kvs_cnt++;
    }
}

```

KVSset 함수는 매개변수로 **key**와 **value** 포인터를 받는다. KVSset 함수는 **key** 값을 가진 기존의 kvs 구조체의 value를 **value**로 업데이트하거나, **key** 값과 **value**값을 가지는 kvs 구조체를 새롭게 추가하는 함수다.



먼저, KVSfind 함수에 **key** 포인터를 전달해 해당 **key** 값을 가진 kvs 구조체가 존재하는지 확인해 반환값을 idx 변수에 저장한다.

idx가 -1이 아니라면 이미 **key** 값을 가진 kvs 구조체가 존재하는 것이므로, **key** 값을 가진 kvs 구조체의 기존의 value를 free하고 새롭게 전달받은 value 포인터를 저장해 업데이트 한다.

idx가 -1이라면 **key** 값을 가진 구조체가 존재하지 않는 것이므로, KVS[kvs\_cnt]에 전달받은 **key** 값과 **value** 값을 저장하고, kvs\_cnt를 1 증가시킨다.

```
char* KVSget(char* key){  
  
    int idx = KVSfind(key);  
    if(idx != -1){  
        char* tmp = KVS[idx].value;  
        return tmp;  
    }  
    else{  
        return NULL;  
    }  
}
```

KVSget 함수는 매개변수로 **key** 포인터를 받는다. KVSget 함수는 전달받은 **key** 값에 해당하는 **value** 값을 반환하는 함수다.

먼저, KVSfind 함수에 **key** 값을 전달 해 해당 **key** 값을 가진 kvs 구조체가 존재하는지 확인해 반환값을 idx 변수에 저장한다.

idx가 -1이 아니라면 이미 **key** 값을 가진 kvs 구조체가 존재하는 것이므로, **key** 값을 가진 kvs 구조체의 value 값을 tmp 변수에 넣고, tmp 변수의 값을 반환한다.

idx가 -1이라면 **key** 값을 가진 구조체가 존재하지 않는 것이므로, NULL 값을 반환한다.

위의 루틴들을 통해 KVS 관련 동작들을 수행할 수 있다.

#### 4. client와 상호 작용하는 thread 함수의 루틴.

pthread\_create 함수로 생성된 스레드는 개별 client 소켓들을 가지고 상호 작용하게 된다. 이 때 편의성을 위해 RESP 데이터의 속 문자열을 추출해 파싱하는 함수를 따로 구현했다. 따라서, 해당 함수를 기술한 후 전체적인 스레드로 돌아가는 함수를 기술하겠다.

```

char* parseResp(char* buf){

    // Extract NUM to num
    // Ex: $(((NUM)))\r\nStrings\r\n
    unsigned int num = atoi(buf+1);

    // Mov buf to Strings
    // Ex: $NUM\r\nStrings\r\n => Strings\r\n
    while(*buf != '\r') buf++;
    buf += 2;

    // Extract Strings to extr
    // Ex: (((Strings)))\r\n
    char* extr = (char*)malloc(num+1); // For Null Termination
    memset(extr, 0, num+1);           // For Null Termination
    strncpy(extr, buf, num);

    // return Strings' Pointer
    return extr;
}

```

RESP 데이터에서 문자열을 파싱하는 parseResp 함수는 위와 같이 RESP 데이터를 가리키는 포인터 buf를 매개변수로 전달받는다. 이 때, buf는 Bulk Strings의 시작 문자인 \$를 가리킨다고 가정한다.

먼저, 문자열의 길이를 파악하기 위해 atoi 함수를 통해 buf+1을 정수로 만든 후 num에 저장한다.(Ex: \$(buf+1)NUM\r\nStrings\r\n)

그리고, buf가 '\r'를 가리킬 때까지 증가시킨 후, '\r'를 가리킬 때 2를 증가시킨다.(Ex: \$NUM\r\nStrings\r\n ⇒ Strings\r\n)

이후, (num+1) 크기의 동적 할당을 받은 주소를 extr에 저장한 후 0으로 초기화한다. buf를 extr에 num 만큼 저장한다.(Ex: (buf)Strings\r\n) 이 때, (num+1) 만큼 동적 할당하는 이유는 Null Termination을 위한 1바이트 공간을 남기기 위해서다.

마지막으로 extr을 반환한다.

```

// One byte for Null Termination
#define BUF_SIZE 1024+1

void* cliInteract(void* clientSocket){

    int soc = *(int*)clientSocket;
    printf("%d Socket connected\n", soc);

    int res;
    char buf[BUF_SIZE];
    char* key;
    char* value;
    char* tmp;

    ...

}

```

쓰레드로써 client와 상호 작용하는 cliInteract 함수는 포인터 clientSocket을 매개변수로 전달 받는다. 이 때, clientSocket은 int 포인터다. 포인터가 가리키는 주소는 main 함수에서 재활용되므로, 지역변수 soc에 형변환을 통해 저장한다.

이후, recv 함수와 send 함수의 반환값을 저장할 res, 전달 받은 데이터와 보낼 데이터를 저장할 char 포인터 buf, key 문자열을 가리킬 key 포인터와 value 문자열을 가리킨 value 포인터, 그리고 임시로 buf의 주소를 저장할 tmp를 선언한다.

```

pthread_mutex_t mutex;

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

    KVSinit();

    ...

    pthread_mutex_init(&mutex, NULL);
    for(;;){

```

```

    ...

}
pthread_mutex_destroy(&mutex);

...

KVSfin();
return 0;

}

```

추가적으로, 공용 메모리인 KVS 시스템에 여러 스레드가 동시에 접근할 수 있으므로 뮤텝스를 사용해야 한다. 따라서, main 함수 시작 후 KVSinit으로 KVS 시스템 메모리 공간을 초기화하고 pthread\_mutex\_init 함수를 통해 뮤텝스를 활성화한다. 이 후, 무한 루프가 끝나면 pthread\_mutex\_destroy 함수를 통해 뮤텝스를 비활성화하고 KVSfin을 통해 KVS 시스템 메모리 공간을 정리한다.

```

void* cliInteract(void* clientSocket){

    ...

    for(;;){

        res = recv(soc, buf, BUF_SIZE, 0);
        if(res <= 0){
            perror("***recv API error***\n");
            close(soc);
            exit(1);
        }
        buf[res] = '\0';

        // Process SET Command
        if(strncmp(buf, "*3\r\n$3\r\nSET\r\n", 13)==0){

            ...

```

```

    }
    // Process GET Command
    else if(strncmp(buf, "*2\r\n$3\r\nGET\r\n", 13)==0)
{
    ...

}
// Process EXIT Command
else if(strncmp(buf, "*1\r\n$4\r\nQUIT\r\n", 14)==
0){
    ...

}
// Process Error
else{
    strncpy(buf, "-ERR\r\n\0", 7);
}

res = send(soc, buf, strlen(buf), 0);
if(res <= 0){
    perror("***send API error***\n");
    close(soc);
    exit(1);
}

}

close(soc);
printf("%d Socket disconnected\n", soc);

}

```

무한 루프를 통해 지속적으로 client와 상호 작용한다.

recv 함수에 **soc**, **buf**, 그리고 **BUF\_SIZE**를 전달해 client로 부터 BUF\_SIZE 크기의 데이터를 buf에 전달 받는다. 반환값에 따라 아무런 값도 전달받지 못하거나 에러면 예외 처리를 한다. 그렇지 않다면, buf[res]에 NULL을 넣어 Null Termination 한다.

이후, buf와 문자열 비교를 통해 set, get, exit, 그리고 에러 명령어를 구분한다. 이 때, 만약 에러가 발생하면 "-ERR\r\n\0"인 RESP 에러 코드를 buf에 저장한다. 나머지 명령어들의 경우에도 대부분 buf에 client에 전달할 값을 저장한다.

마지막으로, send 함수에 **soc**을 전달해 **strlen(buf)** 만큼 **buf**에서 데이터를 client에게 전달하도록 한다. 아무런 값도 전달하지 못하거나 에러면 예외 처리를 한다.

무한 루프를 빠져나왔을 경우, close 함수를 통해 soc 소켓을 닫고 함수를 종료한다.

```
void* cliInteract(void* clientSocket){  
    ...  
    for(;;){  
        ...  
        // Process SET Command  
        if(strncmp(buf, "*3\r\n$3\r\nSET\r\n", 13)==0){  
            // Mov Buf to First String  
            tmp = buf+13;  
  
            key = parseResp(tmp);  
  
            // Mov Buf to Second String  
            tmp++;  
            while(*tmp != '$') tmp++;  
  
            value = parseResp(tmp);  
  
            pthread_mutex_lock(&mutex);  
            KVSset(key, value);  
            pthread_mutex_unlock(&mutex);  
  
            strncpy(buf, "+OK\r\n\0", 6);  
        }  
    }  
}
```

```

    ...

}

    ...

}

```

set 명령어를 이용할 때 공통적으로 발생하는 13바이트 크기의 문자열을 buf와 비교해서 client가 요청한 명령이 set인지 확인한다.

set 명령이 맞다면, Bulk Strings 문자인 \$ 위치인 (buf+13)을 tmp에 저장하고 tmp를 parseResp에 전달해서 얻은 key 값 포인터를 key 포인터에 저장한다.

이후, tmp를 1 증가시켜 \$이 아닌 다른 값을 가리키게 한 후, tmp가 다시 \$을 가리킬 때까지 증가시킨다. 그리고 다시 tmp를 parseResp에 전달해서 얻은 value 값 포인터를 value 포인터에 저장한다. 이로써 key와 value 문자열의 포인터를 얻을 수 있다.

이제 공용 메모리인 KVS에 접근해야 하므로, pthread\_mutex\_lock 함수를 통해 뮤텁스를 걸고 KVSset 함수에 key와 value 포인터를 전달해 (key, value) 쌍을 KVS 시스템에 저장한다. 이후, pthread\_mutex\_unlock 함수를 통해 뮤텁스를 푼다.

마지막으로, set을 성공적으로 마쳤다는 "+OK\r\n0" OK response를 buf에 저장한다.

```

void* cliInteract(void* clientSocket){

    ...

    for(;;){

        ...

        // Process GET Command
        else if(strncmp(buf, "*2\r\n$3\r\nGET\r\n", 13)==0)
        {

            key = parseResp(buf+13);

            pthread_mutex_lock(&mutex);
            value = KVSget(key);

```

```

        pthread_mutex_unlock(&mutex);

        if(value != NULL){
            sprintf(buf, "$%ld\r\n%s\r\n\0", strlen(value), value);
        }
        else{
            strncpy(buf, "$-1\r\n\0", 5);
        }
    }

    ...

}

...

}

```

get 명령어를 이용할 때 공통적으로 발생하는 13바이트 크기의 문자열을 buf와 비교해서 client가 요청한 명령이 get인지 확인한다.

get 명령이 맞다면, Bulk Strings 문자인 \$ 위치인 (buf+13)을 parseResp에 전달해서 얻은 key 값 포인터를 key 포인터에 저장한다.

이제 공용 메모리인 KVS에 접근해야 하므로, pthread\_mutex\_lock 함수를 통해 뮤텁스를 걸고 KVSget 함수에 key 포인터를 전달해 반환값을 value 포인터에 저장한다. 이후, pthread\_mutex\_unlock 함수를 통해 뮤텁스를 푼다.

마지막으로, value가 NULL이 아니라면 사용자로부터 전달받은 key가 KVS 시스템에 존재한다는 뜻이므로 sprintf를 통해 해당 value와 value의 크기를 적절히 RESP 데이터화시켜 buf에 저장한다. 만약 value가 NULL이라면 사용자로부터 전달받은 key가 KVS 시스템에 존재하지 않는다는 뜻이므로, NULL을 의미하는 Bulk Strings인 "\$-1\r\n\0"를 buf에 저장한다.

```

void* cliInteract(void* clientSocket){

```

```

    ...

```



```

for(;;){

    ...

    // Process EXIT Command
    else if(strncmp(buf, "*1\r\n$4\r\nQUIT\r\n", 14)==
0){

        res = send(soc, "+OK\r\n", 5, 0);
        if(res <= 0){
            perror("***send API error***\n");
            close(soc);
            exit(1);
        }
        break;
    }

    ...

}

...

}

```

EXIT 명령어를 이용할 때 공통적으로 발생하는 14바이트 크기의 문자열을 buf와 비교해서 client가 요청한 명령이 EXIT인지 확인한다.

EXIT 명령이 맞다면, OK response를 바로 전달한 후 반환값에 따라 예외 처리한다. 그리고 무한 루프를 종료한다.

위의 코드들을 통해 server에서 쓰레드를 활용해 다중 client와 1:N 상호 작용을 할 수 있는 코드를 구현할 수 있다.

## 빌드 방식 및 실행 결과

```
// Makefile

all: client server

client: client.c
    gcc -o client client.c

server: server.c
    gcc -o server server.c

clean:
    rm server client
```

프로그램 빌드의 경우, 소스코드와 동봉된 Makefile을 통해 빌드할 수 있다.

```
user@redis: ~/Downloads/last/NetworkTerm-main$ make
gcc -o client client.c
client.c: In function 'transAscii':
client.c:60:50: warning: embedded '\0' in format [-Wformat-contai
ns-nul]
   60 |         sprintf(comp+strlen(comp), "%d\r\n%s\r\n\0", ptr[1]-
ptr[0]-1, buf+ptr[0]+1));
      |                                         ^~
client.c:61:50: warning: embedded '\0' in format [-Wformat-contai
ns-nul]
   61 |         sprintf(comp+strlen(comp), "%d\r\n%s\r\n\0", ptr[2]-
ptr[1]-1, buf+ptr[1]+1));
      |                                         ^~
client.c:82:50: warning: embedded '\0' in format [-Wformat-contai
ns-nul]
   82 |         sprintf(comp+strlen(comp), "%d\r\n%s\r\n\0", ptr[1]-
ptr[0]-1, buf+ptr[0]+1);
      |                                         ^~

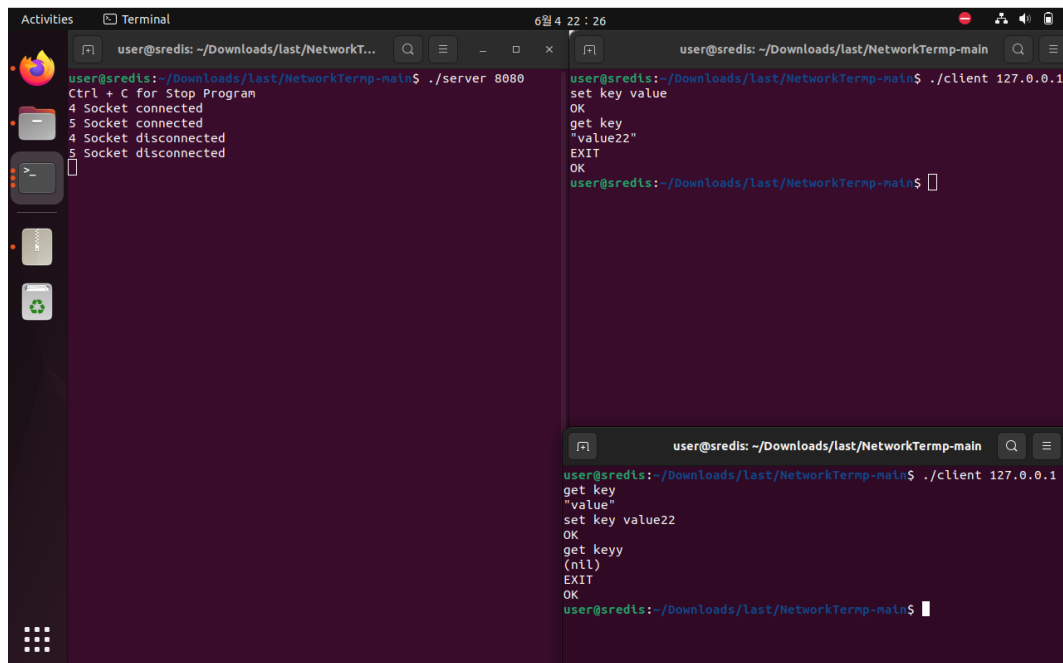
gcc -o server server.c
server.c: In function 'cliInteract':
server.c:133:13: warning: implicit declaration of function 'close
'; did you mean 'pclose'? [-Wimplicit-function-declaration]
   133 |         close(soc);
      |         ^~~~~~
server.c:168:45: warning: embedded '\0' in format [-Wformat-conta
ins-nul]
   168 |         sprintf(buf, "%d\r\n%s\r\n\0", strlen(v
alue), value);
      |                                         ^~

user@redis: ~/Downloads/last/NetworkTerm-main$ ls
client client.c Makefile README.md server server.c
user@redis: ~/Downloads/last/NetworkTerm-main$ ./server 8080
Ctrl + C for Stop Program
4 Socket connected
4 Socket disconnected

user@redis: ~/Downloads/last/NetworkTerm-main$ ./client 127.0.0.
1 8080
set testkey testvalue
OK
get testkey
"testvalue"
get testkeyy
(nil)
EXIT
OK
user@redis: ~/Downloads/last/NetworkTerm-main$
```

Example Scenario

Example Scenario 대로 server와 client를 연결시키고 명령어를 입력하면 위와 같이 잘 동작하는 모습을 확인할 수 있다.



```
user@sredis: ~/Downloads/last/NetworkTemp-main$ ./server 8080
Ctrl + C for Stop Program
4 Socket connected
5 Socket connected
4 Socket disconnected
3 Socket disconnected

user@sredis: ~/Downloads/last/NetworkTemp-main$ ./client 127.0.0.1
set key value
OK
get key
"value22"
EXIT
OK
user@sredis: ~/Downloads/last/NetworkTemp-main$

user@sredis: ~/Downloads/last/NetworkTemp-main$ ./client 127.0.0.1
get key
"value"
set key value22
OK
get key
(nil)
EXIT
OK
user@sredis: ~/Downloads/last/NetworkTemp-main$
```

multi client

Multi client 상황에서도 client끼리 KVS 시스템을 충돌 없이 잘 동작함을 확인할 수 있다.