3. TCP 소켓 프로그래밍

3.1 TCP 서버 구현 (단일 접속)

✅ 서버의 기본 흐름 요약

```
1. socket()   → 소켓 생성
2. bind()     → IP와 포트에 바인딩
3. listen()   → 수신 대기 상태 진입
4. accept()   → 클라이언트 접속 수락 (새 소켓 반환)
5. recv()/send() → 데이터 송수신
6. close()    → 연결 종료
```

✅ 전체 예제 코드

```c
// tcp_server_single.c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <arpa/inet.h>

#define PORT 12345
#define BUFFER_SIZE 1024

int main() {
    int server_fd, client_fd;
    struct sockaddr_in server_addr, client_addr;
    socklen_t addr_len = sizeof(client_addr);
    char buffer[BUFFER_SIZE] = {0};

    // 1. 소켓 생성
    server_fd = socket(AF_INET, SOCK_STREAM, 0);
    if (server_fd < 0) {
        perror("socket failed");
        exit(EXIT_FAILURE);
    }

    // 2. 바인딩
    memset(&server_addr, 0, sizeof(server_addr));  // 구조체 초기화
    server_addr.sin_family = AF_INET;              // IPv4
    server_addr.sin_addr.s_addr = INADDR_ANY;      // 0.0.0.0 (모든 IP 허용)
    server_addr.sin_port = htons(PORT);            // 포트 번호 (네트워크 바이트 순서)

    if (bind(server_fd, (struct sockaddr *)&server_addr, sizeof(server_addr)) < 0) {
        perror("bind failed");
        close(server_fd);
        exit(EXIT_FAILURE);
    }
```

```
    // 3. 리스닝 상태 진입
    if (listen(server_fd, 1) < 0) {
        perror("listen failed");
        close(server_fd);
        exit(EXIT_FAILURE);
    }

    printf("Waiting for connection on port %d...\n", PORT);

    // 4. 클라이언트 접속 수락
    client_fd = accept(server_fd, (struct sockaddr *)&client_addr, &addr_len);
    if (client_fd < 0) {
        perror("accept failed");
        close(server_fd);
        exit(EXIT_FAILURE);
    }

    printf("Client connected: %s:%d\n",
           inet_ntoa(client_addr.sin_addr), ntohs(client_addr.sin_port));

    // 5. 데이터 수신 및 응답
    int bytes_received = recv(client_fd, buffer, BUFFER_SIZE - 1, 0);
    if (bytes_received < 0) {
        perror("recv failed");
    } else {
        buffer[bytes_received] = '\0';  // 널 종료
        printf("Received: %s\n", buffer);

        const char *reply = "Message received.\n";
        send(client_fd, reply, strlen(reply), 0);
    }

    // 6. 연결 종료
    close(client_fd);
    close(server_fd);
    printf("Connection closed.\n");

    return 0;
}
```

✅ 테스트 방법

1. 서버 실행

```
gcc tcp_server_single.c -o tcp_server_single
./tcp_server_single
```

2. 클라이언트 연결 (예: netcat)

```
nc localhost 12345
```

> 메시지를 입력하면 서버가 수신 후 응답하고 종료돼

✅ 핵심 체크포인트

구간	주의사항
`socket()`	실패 시 errno 확인 (리소스 부족 등)
`bind()`	포트 중복 방지 (`SO_REUSEADDR` 활용 가능)
`accept()`	실제 통신은 `client_fd` 로 수행
`recv()`	0을 반환하면 연결 종료 의미
`close()`	서버와 클라이언트 소켓 모두 닫기

✅ 확장 아이디어

- 로그에 `client_fd`, IP, 포트 출력
- `fork()` 혹은 `pthread` 기반 다중 접속 구조로 확장 예정
- `select()` 기반 멀티플렉싱 구조로도 진화 가능

3.2 TCP 클라이언트 구현

✅ 클라이언트 동작 순서

```
1. socket() → 소켓 생성
2. connect() → 서버에 연결
3. send()/recv() → 데이터 송수신
4. close() → 연결 종료
```

✅ 전체 예제 코드

```
// tcp_client.c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <arpa/inet.h>

#define SERVER_IP "127.0.0.1"
#define SERVER_PORT 12345
```

```c
#define BUFFER_SIZE 1024

int main() {
    int sock;
    struct sockaddr_in server_addr;
    char buffer[BUFFER_SIZE];

    // 1. 소켓 생성
    sock = socket(AF_INET, SOCK_STREAM, 0);
    if (sock < 0) {
        perror("socket failed");
        exit(EXIT_FAILURE);
    }

    // 2. 서버 주소 설정
    memset(&server_addr, 0, sizeof(server_addr));
    server_addr.sin_family = AF_INET;
    server_addr.sin_port = htons(SERVER_PORT);

    // IP 주소 문자열을 바이너리로 변환
    if (inet_pton(AF_INET, SERVER_IP, &server_addr.sin_addr) <= 0) {
        perror("inet_pton failed");
        close(sock);
        exit(EXIT_FAILURE);
    }

    // 3. 서버에 연결 요청
    if (connect(sock, (struct sockaddr *)&server_addr, sizeof(server_addr)) < 0) {
        perror("connect failed");
        close(sock);
        exit(EXIT_FAILURE);
    }

    printf("Connected to %s:%d\n", SERVER_IP, SERVER_PORT);

    // 4. 메시지 전송
    const char *msg = "Hello from client!";
    send(sock, msg, strlen(msg), 0);

    // 5. 서버로부터 응답 수신
    int bytes_received = recv(sock, buffer, BUFFER_SIZE - 1, 0);
    if (bytes_received > 0) {
        buffer[bytes_received] = '\0';
        printf("Server replied: %s\n", buffer);
    }

    // 6. 소켓 닫기
    close(sock);
    printf("Connection closed.\n");

    return 0;
}
```

✅ 테스트 방법

1. 서버 실행

먼저 앞서 만든 `tcp_server_single`을 실행해.

```
./tcp_server_single
```

2. 클라이언트 실행

```
gcc tcp_client.c -o tcp_client
./tcp_client
```

> 클라이언트가 "Hello from client!"를 보내고, 서버는 응답을 돌려준다.

✅ 주의할 점

단계	체크포인트
`connect()`	실패 시 서버가 켜져 있는지, IP/포트가 정확한지 확인
`send()`	반환값은 실제 전송된 바이트 수, `EPIPE`나 `SIGPIPE` 조심
`recv()`	반환값 0 → 서버가 종료한 것
`close()`	연결 종료 시 반드시 호출

✅ 확장 방향

- 클라이언트가 사용자 입력을 받아 서버에 전송하게 만들기
- 서버 주소를 명령행 인자로 받도록 확장
- 여러 번 송수신 가능한 구조로 변경

3.3 다중 클라이언트 처리: `fork` 기반

✅ 개요

- `fork()`를 통해 클라이언트마다 별도의 자식 프로세스를 생성함
- 각 프로세스는 독립된 주소 공간을 가지므로 안정적
- 단점: 프로세스 생성 비용이 높고, 자식 프로세스 관리가 필요함

✅ 동작 흐름

```
1. socket()
2. bind()
3. listen()
4. accept() → 클라이언트 연결 수락
5. fork() → 자식 프로세스 생성
   ├─ 부모: accept 루프로 돌아감
   └─ 자식: 클라이언트와 통신 후 종료
6. close()
```

✅ 예제 코드

```c
// tcp_server_fork.c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <arpa/inet.h>
#include <sys/wait.h>
#include <signal.h>

#define PORT 12345
#define BUFFER_SIZE 1024

// 자식 프로세스 종료 시 좀비 처리 방지용
void handle_sigchld(int sig) {
    (void)sig;
    while (waitpid(-1, NULL, WNOHANG) > 0);
}

int main() {
    int server_fd, client_fd;
    struct sockaddr_in server_addr, client_addr;
    socklen_t addrlen = sizeof(client_addr);
    char buffer[BUFFER_SIZE];

    // 시그널 핸들러 등록 (좀비 프로세스 방지)
    signal(SIGCHLD, handle_sigchld);

    // 1. socket 생성
    server_fd = socket(AF_INET, SOCK_STREAM, 0);
    if (server_fd < 0) {
        perror("socket");
        exit(EXIT_FAILURE);
    }

    // 2. bind
    memset(&server_addr, 0, sizeof(server_addr));
    server_addr.sin_family = AF_INET;
```

```
    server_addr.sin_port = htons(PORT);
    server_addr.sin_addr.s_addr = INADDR_ANY;

    if (bind(server_fd, (struct sockaddr *)&server_addr, sizeof(server_addr)) < 0) {
        perror("bind");
        close(server_fd);
        exit(EXIT_FAILURE);
    }

    // 3. listen
    if (listen(server_fd, 10) < 0) {
        perror("listen");
        close(server_fd);
        exit(EXIT_FAILURE);
    }

    printf("Server listening on port %d...\n", PORT);

    // 4. 루프를 돌며 클라이언트 처리
    while (1) {
        client_fd = accept(server_fd, (struct sockaddr *)&client_addr, &addrlen);
        if (client_fd < 0) {
            perror("accept");
            continue;
        }

        pid_t pid = fork();
        if (pid == 0) {
            // 자식 프로세스
            close(server_fd); // 자식은 서버 소켓 불필요

            int n = recv(client_fd, buffer, BUFFER_SIZE - 1, 0);
            if (n > 0) {
                buffer[n] = '\0';
                printf("Client: %s\n", buffer);
                send(client_fd, buffer, n, 0);
            }

            close(client_fd);
            exit(0); // 자식 종료
        } else if (pid > 0) {
            // 부모 프로세스
            close(client_fd); // 부모는 이 클라이언트 처리 안함
        } else {
            perror("fork");
        }
    }

    close(server_fd);
    return 0;
}
```

✅ 테스트 방법

1. 서버 실행:

```
gcc tcp_server_fork.c -o tcp_server_fork
./tcp_server_fork
```

1. 클라이언트를 여러 번 실행하거나 telnet으로 접속:

```
telnet localhost 12345
```

✅ 포인트 정리

포인트	설명
`fork()`	클라이언트마다 별도 프로세스 생성
`SIGCHLD` 핸들러	좀비 프로세스 제거
부모/자식	소켓 역할 분리 명확히 해야 함
`close()` 처리	각각의 소켓을 정확히 닫아야 함

✅ 확장 방향

- 자식 프로세스에서 클라이언트 요청 반복 처리
- 로깅 또는 파일 전송 기능 추가
- `waitpid()`를 통한 자식 상태 감시 개선

3.4 다중 클라이언트 처리: `pthread` 기반

📌 개요

- `fork()`와 달리 **스레드는 메모리 공간을 공유**하기 때문에 생성 비용이 낮고 성능이 유리함.
- 각 클라이언트 연결을 `pthread_create()`로 **새 스레드에 위임**하여 처리.
- 공유 자원에 대한 **동기화 이슈**는 직접 관리해야 함.

✅ 구조 요약

```
main() {
  socket();
  bind();
  listen();
  while (1) {
    accept();
```

```
    pthread_create(); // 클라이언트 처리를 새 스레드에 위임
  }
}
스레드 함수 {
  recv/send 반복;
  close();
  pthread_exit();
}
```

✅ 예제 코드

```c
// tcp_server_pthread.c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <pthread.h>
#include <arpa/inet.h>

#define PORT 12345
#define BUFFER_SIZE 1024

void *handle_client(void *arg) {
    int client_fd = *(int *)arg;
    free(arg);  // 동적으로 할당한 포인터 해제
    char buffer[BUFFER_SIZE];

    int n = recv(client_fd, buffer, BUFFER_SIZE - 1, 0);
    if (n > 0) {
        buffer[n] = '\0';
        printf("Client: %s\n", buffer);
        send(client_fd, buffer, n, 0); // Echo back
    }

    close(client_fd);
    pthread_exit(NULL);
}

int main() {
    int server_fd, *client_fd;
    struct sockaddr_in server_addr, client_addr;
    socklen_t addrlen = sizeof(client_addr);

    server_fd = socket(AF_INET, SOCK_STREAM, 0);
    if (server_fd < 0) {
        perror("socket");
        exit(EXIT_FAILURE);
    }

    memset(&server_addr, 0, sizeof(server_addr));
    server_addr.sin_family = AF_INET;
```

```
    server_addr.sin_port = htons(PORT);
    server_addr.sin_addr.s_addr = INADDR_ANY;

    if (bind(server_fd, (struct sockaddr *)&server_addr, sizeof(server_addr)) < 0) {
        perror("bind");
        close(server_fd);
        exit(EXIT_FAILURE);
    }

    if (listen(server_fd, 10) < 0) {
        perror("listen");
        close(server_fd);
        exit(EXIT_FAILURE);
    }

    printf("Server listening on port %d...\n", PORT);

    while (1) {
        client_fd = malloc(sizeof(int));
        if (!client_fd) {
            perror("malloc");
            continue;
        }

        *client_fd = accept(server_fd, (struct sockaddr *)&client_addr, &addrlen);
        if (*client_fd < 0) {
            perror("accept");
            free(client_fd);
            continue;
        }

        pthread_t tid;
        if (pthread_create(&tid, NULL, handle_client, client_fd) != 0) {
            perror("pthread_create");
            close(*client_fd);
            free(client_fd);
            continue;
        }

        pthread_detach(tid); // 스레드 리소스 자동 회수
    }

    close(server_fd);
    return 0;
}
```

✅ 주요 포인트 정리

항목	설명
`pthread_create()`	클라이언트 처리용 스레드 생성
`pthread_detach()`	스레드 종료 시 자동으로 자원 회수
`client_fd`	힙에 동적 할당해서 race condition 방지
`handle_client()`	클라이언트와 통신 처리 담당

✅ 테스트 방법

1. 서버 컴파일 및 실행:

```
gcc tcp_server_pthread.c -o tcp_server_pthread -lpthread
./tcp_server_pthread
```

1. 여러 개의 `telnet` 또는 클라이언트로 접속해보면, 각각의 연결이 별도의 스레드에서 처리됨.

✅ 확장 포인트

- `read`/`write` 반복 루프 → 지속적인 서비스로 확장
- `thread pool` 방식으로 변경 (성능 최적화)
- mutex나 semaphore를 이용한 공유자원 제어

3.5 접속 끊김, 오류 처리, 종료 처리

📌 1. 접속 끊김의 종류

구분	설명
정상 종료	클라이언트가 `close()` 또는 `shutdown()`을 호출
비정상 종료	프로세스 강제 종료 (`kill`, `crash`, 네트워크 장애 등)
타임아웃 종료	장시간 응답 없음 → 서버가 강제로 연결 종료

📌 2. 감지 방식: `recv()` 반환값으로 확인

```
int n = recv(sock, buf, sizeof(buf), 0);

if (n > 0) {
    // 정상 데이터 수신
} else if (n == 0) {
    // 상대방이 "정상적으로 연결 종료" (close)
} else {
    // 오류 발생
    perror("recv");
}
```

📌 3. `errno` 값에 따른 오류 식별

```
#include <errno.h>

if (recv(...) < 0) {
    if (errno == EINTR) {
        // 시그널에 의해 인터럽트됨 → 재시도 가능
    } else if (errno == EWOULDBLOCK || errno == EAGAIN) {
        // 논블로킹 소켓에서 데이터 없음 → 기다리기
    } else {
        // 기타 오류 → 연결 종료
    }
}
```

📌 4. 클라이언트 강제 종료의 감지

- 클라이언트가 강제로 종료되면 서버는 **recv() → -1**, 또는 **send() 시 SIGPIPE** 발생
- 방지 방법:

```
// send()에서 SIGPIPE 방지
send(sock, buf, len, MSG_NOSIGNAL);

// 또는 전역적으로 무시
signal(SIGPIPE, SIG_IGN);
```

📌 5. `SO_LINGER`로 종료 시도 제어

- `close()` 호출 시 **송신 버퍼가 비워지길 기다릴지 여부** 설정
- 강제로 RST 보내서 즉시 종료할 수도 있음

```c
struct linger sl;
sl.l_onoff = 1;
sl.l_linger = 0;
setsockopt(sock, SOL_SOCKET, SO_LINGER, &sl, sizeof(sl));
```

📌 6. 예외 상황 처리 전략

상황	처리 전략
클라이언트가 예고 없이 종료됨	`recv() == 0` 처리 및 리소스 해제
`send()` 중 SIGPIPE	`MSG_NOSIGNAL` 옵션 또는 `signal(SIGPIPE, SIG_IGN)`
과도한 클라이언트 접속	`listen()` 큐 확장 또는 접속 제한 로직 구현
스레드/프로세스 누수	`pthread_detach()`, `waitpid()` 로 회수

📌 7. 예제 코드 조각

```c
char buf[1024];
int n = recv(client_fd, buf, sizeof(buf), 0);
if (n == 0) {
    printf("클라이언트가 연결을 종료했습니다.\n");
    close(client_fd);
} else if (n < 0) {
    perror("recv 실패");
    close(client_fd);
}
```

📌 8. 종료 시 리소스 정리 체크리스트

- `close(socket_fd)` 수행
- 스레드라면 `pthread_exit()` 또는 함수 return
- `malloc()` 했던 자원 `free()`
- 상태 로그 저장 또는 클린업 핸들러 등록

✅ 정리 요약표

항목	주요 처리 방법
연결 종료 감지	`recv() == 0`, `send()` 오류, `SIGPIPE`
오류 복구	`errno` 에 따라 재시도 또는 연결 종료
자원 회수	`close()`, `free()`, `pthread_detach()`

항목	주요 처리 방법
강제 종료	`SO_LINGER` 설정 또는 `shutdown()`
보안적 종료	무응답 클라이언트에 `timeout` 정책 적용

3.6 TCP 연결 유지 (`keep-alive`, `SO_LINGER` 옵션)

📌 1. TCP 연결 상태와 타임아웃 개요

TCP 연결은 상태 기반(Stateful) 프로토콜이라,

- 연결이 유지되는 동안 시스템 자원을 점유함
- 클라이언트가 종료를 알리지 않으면 "유령 연결(zombie)"이 발생할 수 있음

이를 해결하려면:

- 비정상 연결을 **자동으로 감지하고 종료**
- 연결 종료 시 **데이터 손실 없이 마무리**

✅ 2. `SO_KEEPALIVE`: 유휴 연결 생존 감시

📍 목적

- 장시간 **유휴 상태인 TCP 연결**을 감지하고 강제 종료
- 상대방이 죽었는지 감지하는 역할 (ping과 유사)

📍 사용법

```
int optval = 1;
setsockopt(sock, SOL_SOCKET, SO_KEEPALIVE, &optval, sizeof(optval));
```

📍 동작 메커니즘 (리눅스 기본값 기준)

항목	설명	기본값
`tcp_keepalive_time`	유휴 시간 후 첫 probe 전송	7200초 (2시간)
`tcp_keepalive_intvl`	probe 간 간격	75초
`tcp_keepalive_probes`	실패 허용 횟수	9회

> 즉, 아무런 데이터도 오가지 않으면 2시간 후 probe 시작 → 75초 간격으로 9회까지 시도 → 실패 시 `recv()`가 0을 반환하고 연결 종료

📍 커널 파라미터 변경

```
echo 60 > /proc/sys/net/ipv4/tcp_keepalive_time
echo 10 > /proc/sys/net/ipv4/tcp_keepalive_intvl
echo 5  > /proc/sys/net/ipv4/tcp_keepalive_probes
```

또는 `/etc/sysctl.conf`에 설정 후 `sysctl -p`

✅ 3. `SO_LINGER`: 종료 시 데이터 처리 방식 제어

📍 기본 동작

```
close(sock);
```

- 내부적으로는 `FIN` 전송
- **전송 큐에 남은 데이터가 있으면** 전송 완료까지 블로킹됨

📍 `SO_LINGER` 옵션으로 커스터마이징

```
struct linger sl;
sl.l_onoff = 1;     // SO_LINGER 활성화
sl.l_linger = 0;    // 0초 대기: RST 전송 (즉시 종료)
setsockopt(sock, SOL_SOCKET, SO_LINGER, &sl, sizeof(sl));
```

설정	동작
`onoff=0`	기본값, graceful close (기다림)
`onoff=1`, `linger>0`	linger 초만큼 기다림 (타임아웃 안에 전송 안되면 강제 종료)
`onoff=1`, `linger=0`	즉시 `RST` 전송 (연결 즉시 종료)

📌 4. 실무 적용 전략

시나리오	추천 설정
서버가 클라이언트 비정상 종료 감지 필요	`SO_KEEPALIVE` 활성화
대용량 전송 후 빠른 종료가 필요	`SO_LINGER` with linger > 0
에러나 공격 상황에서 즉시 연결 제거	`SO_LINGER` with linger = 0
일반 웹 서버	`SO_KEEPALIVE` 비활성화 (HTTP는 단기 연결)

🧪 예제 코드

```
// keep-alive 활성화
int optval = 1;
setsockopt(sock, SOL_SOCKET, SO_KEEPALIVE, &optval, sizeof(optval));

// linger 설정
struct linger so_linger = { .l_onoff = 1, .l_linger = 5 };
setsockopt(sock, SOL_SOCKET, SO_LINGER, &so_linger, sizeof(so_linger));
```

✅ 요약 정리

옵션	목적	비고
`SO_KEEPALIVE`	유휴 연결 자동 감지 및 정리	시스템 파라미터 조정 가능
`SO_LINGER`	close 시점에 데이터 처리 방식 제어	즉시 종료 가능 (`RST`)