9. 패킷 처리와 프로토콜 구현

9.1 C로 직접 구현하는 HTTP 서버 (GET/POST 지원)

📘 개요

HTTP는 텍스트 기반의 **애플리케이션 계층 프로토콜**로, 웹 브라우저와 서버 간에 요청과 응답을 주고받기 위해 사용된다. C 언어로 직접 HTTP 서버를 구현하면, **소켓 수준의 I/O 처리부터 프로토콜 파싱**까지 전체 통신 구조를 제어할 수 있다.

✅ 주요 목표

- TCP 기반 HTTP 서버 구축
- GET/POST 요청 구문 파싱
- 정적 파일 응답 또는 요청 본문 처리
- HTTP 1.0/1.1의 기본 구조 준수

🧩 HTTP 요청 메시지 예시

```
GET /index.html HTTP/1.1
Host: localhost:8080
User-Agent: curl/7.68.0
```

```
POST /submit HTTP/1.1
Host: localhost:8080
Content-Length: 11

name=Alice
```

🔧 기본 구현 흐름

1. `socket()`, `bind()`, `listen()`을 이용해 서버 소켓 열기
2. 클라이언트 접속 시 `accept()` 호출
3. `recv()`로 요청 수신
4. 요청 파싱: 메서드, URI, 헤더, 바디 구분
5. 응답 메시지 구성 후 `send()`
6. 연결 종료 (`close()`)

기본 HTTP 서버 코드 예제 (GET 대응)

```c
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <unistd.h>
#include <netinet/in.h>

#define PORT 8080

void handle_client(int client_fd) {
    char buffer[4096];
    int bytes = recv(client_fd, buffer, sizeof(buffer) - 1, 0);
    if (bytes <= 0) return;

    buffer[bytes] = '\0';
    printf("수신 요청:\n%s\n", buffer);

    // 간단한 GET 요청 처리
    if (strncmp(buffer, "GET / ", 6) == 0) {
        const char* body = "<html><body><h1>Hello, World!</h1></body></html>";
        char response[1024];
        sprintf(response,
            "HTTP/1.1 200 OK\r\n"
            "Content-Type: text/html\r\n"
            "Content-Length: %zu\r\n"
            "\r\n"
            "%s", strlen(body), body);
        send(client_fd, response, strlen(response), 0);
    }

    close(client_fd);
}

int main() {
    int server_fd = socket(AF_INET, SOCK_STREAM, 0);
    struct sockaddr_in addr = {0};
    addr.sin_family = AF_INET;
    addr.sin_port = htons(PORT);
    addr.sin_addr.s_addr = INADDR_ANY;

    bind(server_fd, (struct sockaddr*)&addr, sizeof(addr));
    listen(server_fd, 5);

    printf("HTTP 서버가 포트 %d에서 대기 중...\n", PORT);

    while (1) {
        int client_fd = accept(server_fd, NULL, NULL);
        handle_client(client_fd);
    }

    close(server_fd);
```

```
    return 0;
}
```

✅ POST 요청 처리 확장

1. 헤더에서 `Content-Length` 값 파싱
2. 그 길이만큼의 본문 데이터 `recv()`로 추가 수신
3. 본문 분석 및 응답 생성

```
// Content-Length: 값 찾기
char* content_length_header = strstr(buffer, "Content-Length:");
int length = atoi(content_length_header + 15);

// 바디 추출
char* body_start = strstr(buffer, "\r\n\r\n") + 4;
```

🧠 보완 가능한 기능

- 파일 시스템 기반 정적 파일 서비스 (`fopen`, `fread`)
- MIME 타입 자동 지정 (`Content-Type`)
- 404/500 등 상태 코드 대응
- HTTP 버전별 처리 (keep-alive, 헤더 추가 등)
- CGI 또는 FastCGI 방식 연동

🧩 활용 사례

- 네트워크 프로토콜 실습
- 경량 웹 서버 개발
- IoT 디바이스용 내장 HTTP 처리기
- 웹 API 모의 서버(mock server) 작성

🔐 보안 주의사항

- 사용자 입력 검증 필수 (디렉터리 트래버설 등 방지)
- 절대 경로 및 파일 접근 제어
- 헤더 오버플로 방지 (고정 길이 버퍼 조심)

9.2 FTP/SMTP와 같은 간단한 응용 계층 프로토콜 구현

📘 개요

FTP와 SMTP는 OSI 7계층 중 **응용 계층(Application Layer)**에 해당하는 전통적인 텍스트 기반 프로토콜이다. C 언어를 통해 이러한 프로토콜을 직접 구현해 보면, **상태 기반 처리 방식**, **명령 파싱**, **문자열 기반 통신 구조**에 대한 깊은 이해가 가능하다.

✅ FTP 프로토콜 간단 구현

▶️ 프로토콜 개요

- **포트**: 기본적으로 21번 (제어), 20번 (데이터)
- **전송 모드**: Active, Passive
- **명령어 구조**: `USER`, `PASS`, `LIST`, `RETR`, `STOR`, `QUIT`

```
클라이언트 → USER alice
서버      → 331 Password required
클라이언트 → PASS secret
서버      → 230 Login successful
```

▶️ 핵심 구현 포인트 (제어 채널만)

- 명령어를 파싱하여 문자열 비교 후 적절한 응답을 전송
- 파일 리스트 응답은 간단한 텍스트로 구성 가능
- 로그인 처리는 고정 계정으로 간단화 가능

```
if (strncmp(buffer, "USER", 4) == 0)
    send(client_fd, "331 Please specify the password.\r\n", ...);
else if (strncmp(buffer, "PASS", 4) == 0)
    send(client_fd, "230 Login successful.\r\n", ...);
else if (strncmp(buffer, "LIST", 4) == 0)
    send(client_fd, "150 Here comes the directory listing.\r\n", ...);
```

✅ SMTP 프로토콜 간단 구현

▶️ 프로토콜 개요

- **포트**: 기본 25번
- **기본 흐름**:
  - `HELO example.com`
  - `MAIL FROM:<alice@example.com>`
  - `RCPT TO:<bob@example.com>`
  - `DATA`
  - `QUIT`

```
HELO localhost
250 Hello
MAIL FROM:<alice@domain.com>
250 OK
RCPT TO:<bob@domain.com>
250 OK
DATA
354 End data with <CR><LF>.<CR><LF>
Subject: Test

Hello Bob!
.
250 OK
```

▶️ 핵심 구현 포인트

- 각 명령을 상태 기반으로 파싱 처리
- `DATA` 입력 이후 텍스트 수신 후 `.` 입력 시 전송 완료
- 실제 메일 전송은 생략하고 콘솔에 출력 가능

```
if (strncmp(buffer, "HELO", 4) == 0)
    send(client_fd, "250 Hello\r\n", ...);
else if (strncmp(buffer, "MAIL FROM:", 10) == 0)
    send(client_fd, "250 OK\r\n", ...);
else if (strncmp(buffer, "RCPT TO:", 8) == 0)
    send(client_fd, "250 OK\r\n", ...);
else if (strncmp(buffer, "DATA", 4) == 0)
    send(client_fd, "354 End data with <CR><LF>.<CR><LF>\r\n", ...);
```

💡 상태 머신 구성

상태	설명
INIT	클라이언트 접속 직후 대기 상태
AUTH	사용자 인증 처리 중
COMMAND	명령어 수신 대기 상태
DATA_INPUT	`DATA` 명령 이후 본문 입력 처리

상태 전이에 따라 유효한 명령어만 처리할 수 있도록 구현해야 한다.

🛠️ 활용 예시

- FTP 서버의 제어 채널만 구현해 디버깅용 서버 구축
- SMTP 명령 수신기 구현하여 **메일 수신 테스트 서버(mock)** 작성
- 프로토콜 fuzzing이나 침투 테스트 환경 구축용

⚠️ 보안 및 유의사항

- 텍스트 기반이므로 패킷 스니핑 시 민감 정보 노출 위험
- 인증 기능 없음 → 외부 노출 금지
- 데이터 채널 구현 시 포트 열림 주의 필요

📌 요약

항목	FTP	SMTP
용도	파일 업로드/다운로드	메일 발송
명령 구조	`USER`, `PASS`, `LIST`, `RETR`, `QUIT`	`HELO`, `MAIL`, `RCPT`, `DATA`, `QUIT`
전송 모드	제어 채널 + 데이터 채널	단일 채널에서 메일 데이터 송신
구현 난이도	중간 (명령/상태 다수)	낮음 (선형 상태 흐름)

9.3 RAW 소켓을 이용한 커스텀 패킷 생성

📘 개요

RAW 소켓은 운영체제의 일반적인 소켓 API보다 더 **저수준의 패킷 접근**을 제공한다. 이를 이용하면 TCP, UDP, ICMP 등 전송 계층을 우회하고, **IP 헤더 및 그 이하 계층의 필드를 직접 조작**하여 패킷을 생성하거나 분석할 수 있다.

✅ RAW 소켓의 특징

항목	설명
접근 권한	일반 사용자 제한, `root` 권한 필요 (`CAP_NET_RAW`)
헤더 구성 책임	커널이 자동 생성하지 않음, 사용자 직접 구성 (`IP_HDRINCL` 사용 시)
용도	보안 테스트, 프로토콜 구현 실험, ICMP 유틸리티 제작 등

✅ RAW 소켓 생성 및 설정

```
int sock = socket(AF_INET, SOCK_RAW, IPPROTO_RAW);
```

- `AF_INET`: IPv4
- `SOCK_RAW`: RAW 소켓
- `IPPROTO_RAW`: 사용자 정의 전송 계층 프로토콜 사용

헤더 직접 작성 시 다음 옵션 필요:

```
int one = 1;
setsockopt(sock, IPPROTO_IP, IP_HDRINCL, &one, sizeof(one));
```

✅ IP 헤더 수동 생성

```
struct iphdr {
    unsigned int ihl:4;
    unsigned int version:4;
    uint8_t tos;
    uint16_t tot_len;
    uint16_t id;
    uint16_t frag_off;
    uint8_t ttl;
    uint8_t protocol;
    uint16_t check;
    uint32_t saddr;
    uint32_t daddr;
};
```

- `ihl`: 헤더 길이 (기본값 5 = 20바이트)
- `version`: IPv4 → 4
- `protocol`: `IPPROTO_TCP`, `IPPROTO_ICMP` 등
- `check`: 체크섬은 직접 계산하거나 생략 시 커널이 처리

✅ 예제: ICMP Echo Request 생성 (ping 흉내)

```
int sock = socket(AF_INET, SOCK_RAW, IPPROTO_ICMP);

struct sockaddr_in target;
target.sin_family = AF_INET;
inet_pton(AF_INET, "8.8.8.8", &target.sin_addr);

char packet[sizeof(struct icmphdr)];
struct icmphdr *icmp = (struct icmphdr *) packet;
icmp->type = ICMP_ECHO;
icmp->code = 0;
```

```
icmp->un.echo.id = htons(1234);
icmp->un.echo.sequence = htons(1);
icmp->checksum = 0;
icmp->checksum = calculate_checksum((unsigned short *)icmp, sizeof(struct icmphdr));

sendto(sock, packet, sizeof(packet), 0, (struct sockaddr *)&target, sizeof(target));
```

※ `calculate_checksum()` 함수는 ICMP 표준 체크섬 계산 알고리즘 필요

✅ 체크섬 계산 함수 예시

```
unsigned short calculate_checksum(unsigned short *ptr, int nbytes) {
    unsigned long sum = 0;
    while (nbytes > 1) {
        sum += *ptr++;
        nbytes -= 2;
    }
    if (nbytes == 1)
        sum += *(unsigned char *)ptr;

    sum = (sum >> 16) + (sum & 0xffff);
    sum += (sum >> 16);
    return (unsigned short)(~sum);
}
```

📌 유의 사항

항목	설명
권한 문제	대부분 OS에서 RAW 소켓은 root 전용
방화벽/커널 차단 가능	일부 시스템에서는 보안상 제한됨
체크섬 오류 주의	TCP, UDP, ICMP 등은 체크섬 필수 계산 필요
헤더 정렬/패딩	바이트 순서, 패딩, 정렬 등 구조체 설계 주의

🔧 활용 사례

- **ping, traceroute** 구현
- **네트워크 취약성 분석 도구** 작성
- **IDS/IPS 시스템 개발용 트래픽 생성기**
- **커스텀 프로토콜 실험 (IP 옵션 필드 등)**

9.4 `libpcap`을 이용한 패킷 스니핑

📘 개요

`libpcap`은 패킷 캡처를 위한 **범용 C 라이브러리**로, 다양한 운영체제에서 **네트워크 트래픽을 가로채고 분석**하는 데 사용된다. Wireshark, tcpdump와 같은 툴이 이 라이브러리를 기반으로 동작한다.

✅ 주요 기능

기능	설명
패킷 캡처	NIC를 promiscuous mode로 설정하여 모든 패킷 수신
필터링 지원	BPF (Berkeley Packet Filter)를 통한 고속 필터링
프로토콜 독립성	이더넷, 무선, 루프백 등 다양한 인터페이스 지원

✅ 기본 사용 흐름

```
#include <pcap.h>

pcap_t *pcap_handle;
char errbuf[PCAP_ERRBUF_SIZE];

pcap_handle = pcap_open_live("eth0", BUFSIZ, 1, 1000, errbuf);
```

인자	설명
`"eth0"`	캡처할 인터페이스 이름
`BUFSIZ`	캡처 버퍼 크기
`1` (promiscuous)	모든 패킷 수신 여부
`1000`	읽기 타임아웃 (ms)
`errbuf`	오류 발생 시 메시지 저장용 버퍼

✅ 패킷 수신 콜백 등록

```
void packet_handler(u_char *args, const struct pcap_pkthdr *header, const u_char
*packet) {
    printf("수신한 패킷 길이: %d 바이트\n", header->len);
}

pcap_loop(pcap_handle, 10, packet_handler, NULL);
```

- `pcap_loop`: 지정한 횟수만큼 패킷을 수신하며 콜백 실행
- `header`: 캡처된 패킷의 메타정보 (길이, 타임스탬프 등)
- `packet`: 실제 이더넷/네트워크 프레임 데이터

✅ BPF 필터 설정 예시

```
struct bpf_program fp;
pcap_compile(pcap_handle, &fp, "tcp port 80", 0, PCAP_NETMASK_UNKNOWN);
pcap_setfilter(pcap_handle, &fp);
```

- `"tcp port 80"`: HTTP 트래픽만 캡처
- BPF 문법은 tcpdump와 동일 (`udp`, `icmp`, `host 192.168.0.1` 등)

✅ 캡처 종료 및 정리

```
pcap_freecode(&fp);
pcap_close(pcap_handle);
```

📦 패킷 파싱 예시 (이더넷/IP/TCP)

```
#include <netinet/ip.h>
#include <netinet/tcp.h>

void packet_handler(u_char *args, const struct pcap_pkthdr *header, const u_char
*packet) {
    struct iphdr *ip = (struct iphdr *)(packet + 14);       // Ethernet 헤더 14바이트
    struct tcphdr *tcp = (struct tcphdr *)(packet + 14 + ip->ihl * 4);

    printf("출발지 IP: %s\n", inet_ntoa(*(struct in_addr *)&ip->saddr));
    printf("목적지 포트: %d\n", ntohs(tcp->dest));
}
```

✅ 주의 사항

항목	설명
루트 권한 필요	대부분의 시스템에서 인터페이스 접근은 root만 가능
인터페이스 이름 확인	`pcap_findalldevs()` 로 사용 가능한 인터페이스 조회 가능
프리미티브 제한	일부 환경에서는 promiscuous 모드 비허용

🔧 활용 사례

- IDS/IPS 시스템
- 네트워크 포렌식 분석기
- 트래픽 시각화 도구 (flow 분석)
- 특정 프로토콜 감지기 또는 필터링

🧠 확장 학습 포인트

- `pcap_dump_open()`으로 패킷 로그 저장 (Wireshark와 호환되는 `.pcap` 포맷)
- `libpcap` → `tcpdump`, `Wireshark`로 연동
- 고성능 환경에서는 `PF_RING`, `DPDK` 등과 비교

9.5 ICMP 패킷 처리 (ping 프로그램 구현)

📘 개요

ICMP(Internet Control Message Protocol)는 IP 프로토콜의 일부로, **네트워크 상태를 진단**하거나 오류를 알리기 위해 사용된다.
`ping`은 대표적인 ICMP Echo Request/Reply를 이용한 유틸리티이며, **지연 시간(RTT)** 측정과 **호스트 접근성 테스트**에 사용된다.

✅ ICMP 프로토콜 구조

ICMP는 IP 계층 위에서 동작하며, 전송 계층(TCP/UDP)을 사용하지 않는다. 주요 메시지 타입:

Type	Code	설명
8	0	Echo Request (ping 요청)
0	0	Echo Reply (ping 응답)
3	여러	목적지 도달 불가
11	0/1	TTL 초과

✅ ICMP 헤더 구조 (`<netinet/ip_icmp.h>`)

```
struct icmphdr {
    uint8_t  type;
    uint8_t  code;
    uint16_t checksum;
    union {
        struct {
            uint16_t id;
            uint16_t sequence;
```

```
        } echo;
        uint32_t   gateway;
        struct {
            uint16_t __unused;
            uint16_t mtu;
        } frag;
    } un;
};
```

- `type`, `code`: 메시지 종류
- `checksum`: ICMP 헤더와 데이터에 대한 체크섬
- `id`, `sequence`: 식별자 및 시퀀스 번호 (매칭용)

✅ ping 프로그램 핵심 흐름

```
int sockfd = socket(AF_INET, SOCK_RAW, IPPROTO_ICMP);
```

- RAW 소켓을 통해 ICMP 직접 생성 및 수신

```
// 송신용 Echo Request 생성
struct icmphdr icmp;
icmp.type = ICMP_ECHO;
icmp.code = 0;
icmp.un.echo.id = htons(getpid() & 0xFFFF);
icmp.un.echo.sequence = htons(seq++);
icmp.checksum = 0;
icmp.checksum = calculate_checksum((unsigned short*)&icmp, sizeof(icmp));
```

```
// 수신 대기
recvfrom(sockfd, buffer, sizeof(buffer), 0, (struct sockaddr*)&addr, &addr_len);
```

✅ 체크섬 계산 함수 예시

```
unsigned short calculate_checksum(unsigned short *buf, int len) {
    unsigned long sum = 0;
    while (len > 1) {
        sum += *buf++;
        len -= 2;
    }
    if (len == 1) sum += *(unsigned char *)buf;
    sum = (sum >> 16) + (sum & 0xffff);
    sum += (sum >> 16);
    return ~sum;
}
```

✅ 예시: 간단한 ping 유사 구현

```c
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <unistd.h>
#include <arpa/inet.h>
#include <netinet/ip_icmp.h>
#include <sys/socket.h>
#include <sys/time.h>

int main() {
    int sockfd = socket(AF_INET, SOCK_RAW, IPPROTO_ICMP);
    if (sockfd < 0) perror("socket");

    struct sockaddr_in dest;
    dest.sin_family = AF_INET;
    inet_pton(AF_INET, "8.8.8.8", &dest.sin_addr);

    char sendbuf[64];
    struct icmphdr *icmp = (struct icmphdr *)sendbuf;
    memset(icmp, 0, sizeof(*icmp));

    icmp->type = ICMP_ECHO;
    icmp->code = 0;
    icmp->un.echo.id = htons(getpid() & 0xFFFF);
    icmp->un.echo.sequence = htons(1);
    icmp->checksum = calculate_checksum((unsigned short *)icmp, sizeof(*icmp));

    sendto(sockfd, sendbuf, sizeof(*icmp), 0, (struct sockaddr *)&dest, sizeof(dest));

    char recvbuf[1024];
    recv(sockfd, recvbuf, sizeof(recvbuf), 0);
    printf("Ping 응답 수신\n");

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

📌 주의 사항

항목	설명
루트 권한 필요	ICMP를 RAW로 생성하는 소켓은 일반 사용자 권한으로 제한됨
체크섬 정확성	필수. 오류 시 대상이 응답하지 않음
보안 이슈	방화벽에서 ICMP 차단되는 경우 있음

🔧 확장 포인트

- `gettimeofday()`를 사용해 RTT 계산
- 여러 번 전송 후 평균 지연 시간, packet loss율 출력
- `poll()` 또는 `select()`로 수신 시간 초과 처리
- ICMP TTL 응답 기반의 traceroute 구현도 가능