C언어를 이용한 웹서버 구현

2020.06.05

소프트웨어학부 컴퓨터전공

2016003727

윤주경

목차

# 서론

## C언어를 이용한 Socket 통신

### 서버측 코드

### 클라이언트 측 코드

## HTTP request와 HTTP response

## 사용 함수 및 설명

# Part1 Client의 request message를 browser에서 출력

## 서버 디자인

## 코드 구현 과정

## 동작 예시

# Web server와 browser간 request와 response

## 서버 디자인

## 코드 구현 과정

## 동작 예시

제 1장

이번 프로젝트의 목표는 C언어만을 이용해서 서버와 클라이언트의 동작 방식에 대해 이해하고 실제로 구현해보는 것이다. 그 동안 몇차례 서버와 클라이언트에 대해 공부하고 실제로 결과물도 만들었지만 이미 존재하는 API나 프레임워크를 이용해 만들었기 때문에 완벽하게 이해하고 만들었다고 보기에는 무리가 있다. php와 java script를 이용해 웹서버를 만들어 보았지만 C언어로는 처음이기 때문에 기본적인 개념에 대해 알아가는 것이 중요하다.

하나하나 공부하면서 웹서버의 기능을 확장시키는 방식으로 프로젝트를 진행하였기 때문에 github를 이용해 프로젝트를 업로드 하였다. 과제이기 때문에 카피문제를 피하기 위해 최근에 무료가 된 github 비공개 저장소를 이용했다.

제 1절 C언어를 이용한 Socket통신

프로젝트 예제 파일로 서버와 클라이언트를 C언어로 구현한 코드가 주어졌다. 이 코드를 바탕으로 서버와 클라이언트가 어떤 순서와 방식으로 통신하는지 알아본다. 처음 공부하는 입장에서 코드를 다시 작성하였기 때문에 좋은 코드라고 할 수는 없지만 순서대로 진행하기 위해 코드를 따로 정리하지는 않았다.

**제 1항 서버측 코드**

연결을 위한 서버의 코드는 크게 5단계로 나눠진다. 소켓 생성, 소켓에 주소 할당, 클라이언트 요청 대기, 통신, 종료.

서버 쪽 코드를 먼저 살펴보면 우선 맨 처음으로 main함수의 명령인수의 숫자를 검사한다. 명령인수로 포트번호를 알려주기 때문에 명령인수가 적절치 않으면 프로그램을 종료해야 한다.

1. if (argc < 2) { // 명령 인수 숫자가 적다
2. fprintf(stderr, "Argument is not valid.\n"); // 에러문 출력
3. exit(0); // 서버 종료
4. }

 위 조건문을 지나면 받은 인자를 atoi 함수를 이용해 입력받은 Port번호를 정수형 변수로 변환해 준다.

1. int port\_number = atoi(argv[1]);
2. if( port\_number < 0 ){ error\_print("Not valid Port"); }

  그리고 <sys/socket.h> 헤더파일에 명시되어 있는 socket함수를 이용해서 소켓을 생성한다. 이 때 생성한 소켓은 소켓 디스크립터로 접근 할 수 있다. socket함수는 3개의 인자를 받는데 순서대로 통신 영역, 사용 할 전송 방식, 사용되는 프로토콜을 의미한다. 통신 영역은 IPv4를 사용하기 위해 PF\_INET, 웹 서버이기 때문에 TCP프로토콜을 사용하기 위해 SOCK\_STREAM, 마지막으로는 IPPROTO\_TCP을 인자로 넣어 소켓을 생성한다.

1. int client\_socket\_fd = socket(PF\_INET, SOCK\_STREAM, IPPROTO\_TCP);
2. if( client\_socket\_fd < 0 ){ // 소켓을 만들기 실패하면 종료
3. error\_print("Fail to open socket.\n");
4. }

만들어진 소켓에 주소를 할당하기 위해 bind 함수를 이용해준다. 그 전에 소켓 주소를 따로 정의해주기 위해 sockaddr\_in 구조체를 사용해주었다. 클라이언트의 요청을 받기 위해 클라이언트 주소도 필요하다.

1. struct sockaddr\_in server\_address, client\_addr;
2. // 소켓 주소를 담는 구조체, 서버와 클라이언트 2개 필요하다.
3. memset (&server\_address, 0, sizeof(server\_address));
4. // 구조체를 0으로 초기화
5. server\_address.sin\_family = AF\_INET;  // IPv4로 설정
6. server\_address.sin\_port = htons(port\_number); // htons함수를 통해 포트번호를 네트워크 바이트 순서로 변환해준다.
7. server\_address.sin\_addr.s\_addr = htonl(INADDR\_ANY);
8. // 주소를 자동으로 대입해주는 INADDR\_ANY 사용, long형이라 htonl사용

 bind함수를 통해 서버측 소켓에 서버의 주소를 배정해준다.

1. if ( bind( server\_socket\_fd, (struct sockaddr \*)&server\_address, sizeof(server\_address) ) < 0)
2. {
3. error\_print("error to bind");
4. }
5. // 첫번쨰 인자 : 서버 소켓 디스크립터
6. // 두번째 인자 : 서버의 주소 정보
7. // 세번째 인자 : 서버의 주소 정보의 크기

 bind까지 마치고 나면 클라이언트의 요청을 받아줘야 한다. listen 함수를 통해 서버측 소켓의 연결 대기열을 만들어서 서버는 대기상태가 된다.

1. listen(server\_socket\_fd, 7);

 client의 요청이 들어오면 클라이언트의 주소를 이용해서 연결을 허용해준다. 이 때 연결된 서버와 클라이언트는 accept함수의 성공 시 생성된 새로운 소켓을 이용해 통신한다.

1. socklen\_t req\_client = sizeof(client\_addr);

 이 때 서버와 클라이언트간 메시지를 주고 받기 위해 두개의 문자열을 생성하였다.

1. char \*request\_msg; // 클라이언트에서 받은 메시지
2. request\_msg = malloc( (size\_t)MSG\_SIZE );
3. char \*response\_msg; // 클라이언트로 보낼 메시지
4. response\_msg = malloc( (size\_t)MSG\_SIZE );

 accept(), write(), read()를 이용해서 클라이언트와 통신한다.

1. while(1){
2. int socket\_fd = accept( server\_socket\_fd, (struct sockaddr \*)&client\_addr, &req\_client);
3. if( socket\_fd < 0){
4. error\_print("Fail to accpet");
5. }
6. if( read(socket\_fd, request\_msg, MSG\_SIZE) < 0 ){
7. // client에서 request를 받는다.
8. error\_print("Fail to read");
9. }
10. printf("to server from client : %s", request\_msg);
11. printf("enter the msg : ");
12. fgets(request\_msg, MSG\_SIZE, stdin ); // 메시지 입력을 받는다.
13. if( write(socket\_fd, response\_msg, strlen(response\_msg))< 0 ) {
14. // write함수를 이용해 소켓에 메시지를 작성한다.
15. error\_print("Fail to writing to socket.");
16. }
17. close(socket\_fd);
18. }

 통신이 끝나면 할당된 메모리를 해제하고 열려있는 디스크립터를 닫아준다.

1. free(response\_msg);
2. free(request\_msg);
3. close(server\_socket\_fd);
4. return 0;

\* 에러문 출력 함수

1. void error\_print(char \*error\_msg){
2. perror(error\_msg);
3. exit(1);
4. }

**제 2항 클라이언트측 코드**

연결을 위해 서버에 요청을 보내는 클라이언트측 코드는 크게 4가지 단계로 나눠진다. 소켓 생성, 소켓 연결, 통신, 종료.

서버측과 마찬가지로 맨 처음에는 main함수의 명령인수를 검사한다. 서버와 달리 machine name과 포트번호 2가지 인수를 받는다. 이 때 편의를 위해 machine name은 localhost로 정의한다. 포트번호는 서버와 동일한 번호를 사용한다.

1. if (argc < 3) { // 인자가 충분치 않다.
2. fprintf(stderr, "Argument is out of quntity.\n");
3. // 에러문 출력후 클라이언트 종료
4. exit(0);
5. }

 위 조건문을 지나면 받은 인자를 atoi 함수를 이용해 입력받은 포트 번호를 정수형 변수로 변환해 준다.

1. int port\_number = atoi(argv[2]);
2. if( port\_number < 0 ){ error\_print("Not valid Port"); }

서버측과 동일한 프로토콜을 사용하는 소켓을 생성한다.

1. int client\_socket\_fd = socket(PF\_INET, SOCK\_STREAM, IPPROTO\_TCP);
2. if( client\_socket\_fd < 0 ){
3. error\_print("Fail to open socket.\n");
4. }

 그 다음에는 서버 접속을 위해 hostent 구조체를 정의해 클라이언트의 정보를 저장한다. 도메인 주소로 IP주소를 얻기 위해 gethostbyname()을 사용하였다.

1. struct hostent \*server;
2. if ( (server = gethostbyname(argv[1])) == NULL) {
3. fprintf(stderr, "Host error\n");
4. exit(0);
5. }

 서버와 마찬가지로 socketaddr\_in 구조체를 사용해서 소켓 주소를 저장한다. 이 때 bind() 는 따로 사용하지 않는다. 왜냐하면 서버는 어떤 클라이언트가 접근해 올 지 몰라 클라이언트의 정보를 미리 알 필요가 없다. 그래서 클라이언트에서는 고정된 포트 번호를 배정하는 bind 과정을 따로 할 필요가 없다. 이와 달리 서버는 항상 고정된 포트 번호가 필요하기 때문에 bind가 필요하다.

1. struct sockaddr\_in server\_address; // 소켓 주소를 담는 구조체
2. memset (&server\_address, 0, sizeof(server\_address));
3. // 구조체를 0으로 초기화
4. server\_address.sin\_family = AF\_INET; // IPv4로 설정
5. server\_address.sin\_port = htons(port\_number);
6. // htons함수를 통해 포트번호를 네트워크 바이트 순서로 변환해준다.
7. server\_address.sin\_addr.s\_addr = htonl(INADDR\_ANY);
8. // 서버주소에 IP 저장

위 과정이 끝나면 클라이언트는 서버와 통신할 준비가 되었다. Connect 함수를 통해 서버에 연결 요청을 보낸다. 이 때 connect함수의 인자는 순서대로 클라이언트 소켓 디스크립터, 서버의 주소 정보, 서버 주소 정보의 크기이다.

1. if( connect( client\_socket\_fd, (struct sockaddr \*)&server\_address, sizeof(server\_address)) < 0){
2. error\_print("Fail to connect");
3. }

 연결이 되었다면 통신하기 위한 두 개의 문자열을 생성한다.

1. char \*request\_msg; // 서버로 보낼 메시지
2. request\_msg = malloc( (size\_t)MSG\_SIZE );
3. char \*response\_msg; // 서버에서 받은 메시지
4. response\_msg = malloc( (size\_t)MSG\_SIZE );

write()와 read()를 이용해 서버와 통신 할 수 있다.

1. while(1){
2. printf("enter the msg : ");
3. fgets(request\_msg, MSG\_SIZE, stdin ); // 메시지 입력을 받는다.
4. if ( write(client\_socket\_fd, request\_msg, strlen(request\_msg)) < 0 ) {
5. // write함수를 이용해 소켓에 메시지를 작성한다.
6. error\_print("Fail to writing to socket.");
7. }


11. if ( read( client\_socket\_fd, response\_msg, MSG\_SIZE) < 0 ){//소켓 디스크립터에서 읽기
12. error\_print("Fail to reading from socket.");
13. }
14. printf("to client from server : %s", response\_msg);
15. }

통신이 끝나면 메모리 해제와 디스크립터를 닫아 프로그램을 종료한다.

1. free(response\_msg);
2. free(request\_msg);
3. close(client\_socket\_fd);
4. return 0;

제 3절 사용 도구

위 코드를 작성하면서 여러 개의 구조체와 함수, 헤더파일을 사용하였다. 해당 내용에 대해 간략하게 요약해본다.

#헤더파일

#include <stdio.h>

#include <stdlib.h>

#include <sys/socket.h>

#include <netinet/in.h>

#include <sys/types.h>

#include <netdb.h>

#include <string.h>

#include <unistd.h>

#hostent 구조체

struct hostent {

char \*h\_name; //도메인 이름

char \*\*h\_aliases; //다른 도메인 이름들

int h\_addrtype; //주소정보 체계(IPv4: AF\_INET, IPv6: AF\_INET6)

int h\_length; // 주소의 길이

char \*\*h\_addr\_list; //도메인 이름에 대한 IP주소가 정수 형태로 반환될 때 이 멤버 변수를 이용

};

#sockaddr\_in 구조체

struct sockaddr\_in {

short sin\_family; // 주소 체계: 항상 AF\_INET으로 설정 (IPv4)

AF\_INET u\_short sin\_port; // 16 비트 포트 번호, network byte order

struct in\_addr sin\_addr; // 32 비트 IP 주소

char sin\_zero[8]; // 전체 크기를 16 비트로 맞추기 위한 dummy

};

struct in\_addr {

u\_long s\_addr; // 32비트 IP 주소를 저장 할 구조체, network byte order

};

# CPU에서는 데이터를 저장하고 해석하는 방식이 두 가지가 있다. 그래서 네트워크에 대한 데이터를 저장하거나 가져올 때는 해당 형식에 맞춰야 한다. 네트워크에서는 Big endian 형식을 사용한다. 형식을 맞추기 위해 short, long 형에 맞춰 4가지 함수가 있다.

unsigned short htons(unsigned short);

unsigned short ntohs(unsigned short);

unsigned long htonl(unsigned long);

unsigned long ntohl(unsigned long);

# INADDR\_ANY

# IPv4 IPv6

#브라우저

해당 프로젝트에 사용되는 브라우저는 Google Chrome이다.

제 3절 HTTP response와 HTTP request

프로젝트의 요구사항을 만족시키기 위해 클라이언트를 브라우저로 변경하였다. Shell에서 이루어졌던 통신과 달리 브라우저와 서버간 통신이기 때문에 브라우저를 통해 서버를 접속하기만 해도 HTTP request를 보낸다. 이 request는 브라우저에서 보내오는 것이기 때문에 위에서 작성한 코드를 바꾸지 않고도 메시지를 볼 수 있다. 내용은 아래와 같다.

GET / HTTP/1.1

Host: localhost:8080

Connection: keep-alive

Upgrade-Insecure-Requests: 1

User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/83.0.4103.61 Safari/537.36

Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,image/apng,\*/\*;q=0.8,application/signed-exchange;v=b3;q=0.9

Sec-Fetch-Site: none

Sec-Fetch-Mode: navigate

Sec-Fetch-User: ?1

Sec-Fetch-Dest: document

Accept-Encoding: gzip, deflate, br

Accept-Language: ko-KR,ko;q=0.9,en-US;q=0.8,en;q=0.7

HTTP/1.1은 HTTP/1.0보다 많은 필드가 추가되었다.

첫번째 줄은 request-LINE으로 서버에 HTTP/1.1로 Get메소드를 이용하여 / 주소에 요청을 보낸다는 뜻이다. Method, request-line으로 이루어져 있다.

다음줄부터는 리퀘스트의 헤더의 내용이다.

HOST : 요청하는 호스트에 대한 호스트 명 및 아이피 주소이다. 프로젝트에서는 Localhost를 이용해 8080 포트로 접속하였다. 전체 URI가 필요하여 localhost:8080 이 적혀있다.

Connection : 클라이언트와 서버간 연결의 정보이다. HTTP메시지를 보내고 다시 받기위해 연결을 유지해야 하기 때문에 keep-alive로 되어 있다.

User-agent는 리퀘스트를 보낸 유저의 정보를 담고있다. 즉. 브라우저의 정보가 담겨져 있다. Mozilla/5.0 기반으로 KTHML을 사용한 Chrome이다. 뒤의 Safari는 비슷한 버전을 의미한다.

Accept는 클라이언트가 어떤 Content를 이해 할 수 있는지 나타낸다. 해당 줄은 MIME type으로 표현되어 있다. 파일종류/확장자로 표현되어 있는데 몇몇은 그렇지 않다. \*/\*는 어떤 미디어 타입도 가능하다는 뜻이다

Accept-encoding과 Accept-Language는 클라이언트가 원하는 인코딩 방식과 원하는 가능한 언어이다.

q는 품질을 의미한다 q=0.8의 의미는 품질이 80%이상인 경우에만 해당 내용을 선호한다는 뜻이다.

v는 버전을 의미한다.

Sec-fetch-site는 자원의 origin과 요청 initiator의 origin의 관계를 나타낸다

Sec-fetch-mode는 요청 모드를 의미한다.

Sec-fetch-user는 탐색 요청이 클라이언트에 의해 활성화 되었는지 알려준다. ?1인 경우 탐색 요청이 유저 이외의 것에 의해 트리거 되었다,

Sec-fetch-dest는 요청의 목적지를 의미한다. 이 데이터가 document형태로 사용될 거라는 것을 의미한다.

제 2장

1장의 코드를 통해 C언어를 이용한 소켓 프로그래밍에 대해 이해하였다. 2장에서는 프로젝트의 요구사항을 만족시키기 위해 위 코드를 확장 시켰다. 클라이언트는 브라우저가 대신하기 때문에 서버 측 코드만 수정해주면 되었다. Shell에서 이루어졌던 통신과 달리 브라우저로 통신하기 때문에 서버는 단순히 데이터만 보내면 안된다. 그 데이터에 헤더파일을 붙여 HTTP 통신을 해야한다.

제 1절 서버 디자인

브라우저가 http response를 보냈으니 서버는 http response를 보내야 한다. Response에 대한 헤더는 브라우저의 개발자 콘솔에서 볼 수 있다. 우선 화면에 출력해야 하므로 Content-type은 text/html이다. 그리고 content-length를 지정해 줘야 하는데, C언어 내장 함수로 간단히 구할 수 있다. 단순히 response message만 출력하는 것이지만 처음에는 메시지의 내용을 어떻게 구성할 지 막막하였다. 검색을 통해 헤더는 작성하였지만 메시지 내용이 문제였다. 다행히 개발자 콘솔에서 html요청에 대한 response의 내용을 볼 수 있었다. 그래서 해결하였다. 특히 헤더의 내용이 잘못되거나 빠지면 에러가 나와서 어떤 헤더가 필요한지 구하는 과정이 어려웠다.

제 2절 코드 구현 과정

코드 어떻게 짠지 간단한 설명과 함께 보여주자

제 3절 동작 예시

직접 어떻게 실행되는지 보여주자

제 3장

2장에서 성공적으로 클라이언트가 요청하면 HTML형식으로 HTTP response를 보내주었다. 이 다음에는 어떤 파일을 요청하면 그에 해당되는 형식에 맞게 response해 주어야 한다. 이 프로젝트에서는 5개의 파일 확장자만 인식하도록 했지만, 오류에 대한 처리도 필요하다.

제 1절 서버 디자인

HTML파일은 text형식으로 이루어져 있다. 그래서 사람이 직접 읽어서 어떤 의미인지 쉽게 확인할 수 있다. 그러나 사진, 오디오 등은 다르다. 이 파일들은 헥스 에디터를 이용해 바이너리 형식으로 읽을 수 있다.

우선 첫번째로 오류 처리부터 하였다. 존재하지 않는 파일을 찾거나 인식하지 않는 파일 확장자나 이상한 확장자를 쓰면 http 404를 보내도록 하였다.

그리고 HTML파일을 보내기 위해서 해당 파일을 열어서 그 내용을 버퍼에 넣었다. 그리고 그 버퍼를 헤더와 합쳐 클라이언트에 보내주었다.

그 다음으로 사진 파일을 보내기 위해서 파일을 열었다. 직접 확인이 가능했던 HTML파일과 다르게 제대로 파일을 읽는지 확인이 불가능 하였기 때문에 많은 시간이 걸렸다. 세그멘테이션 오류부터 시작해 다양한 오류를 겪었다.

제대로 파일을 읽는지 확인하기 위해 헥스 에디터를 이용해 버퍼가 바이너리 데이터가 제대로 들어가는지 확인했다. 가장 오래 힘들었던 부분은 바이너리 데이터도 잘 들어갔는데 content-length-mismatch에러가 일어나는 부분이었다. 버퍼의 데이터를 하나하나 검사하고, 엔터 하나 널 문자 하나가 잘못 되었는지 확인해보았다. 해결하고 나니 단순한 문제였지만 어떤 부분에서 길이 오류가 났는지 알수가 없는게 힘들었다. 해결법은 개발자 콘솔을 이용했다. 콘솔의 네트워크 항목에서 request에 대한 response가 나와있었다. 그 중 SIZE항에 마우스르 갖다 대면 xB transferred over network, resource size : yB라고 나와있었다. 서버에서 파일 크기를 하나하나 다 확인하고 있었기 때문에 a가 헤더와 메시지크기, b가 메시지 크기를 의미하는 것을 알게되었다. 문제점은 strlen에 있었다. Strlen은 널 문자를 기준으로 길이를 측정하는데 바이너리 데이터는 중간에 널 문자가 들어갈 수가 있다. 그래서 파일을 열었을 때 파일 크기를 저장해서 헤더 크기와 더해준다.

오디오 파일은 세그멘테이션 오류를 많이 겪었다. 그래서 gdb를 이용해서 어느 부분에서 일어나는지 확인해서 메모리 할당을 잘 해주었다.

그래서 디자인은 아래와 같이 되었다…. (사진 첨부하자)

제 2절 코드 구현 과정

코드 짠거 보여주기

제 3절 동작 예시

실행 화면 보여주자