Socket Programming Tutorial

Networked Computing Lab (NXC Lab)

Department of Electrical and Computer Engineering

Seoul National University

https://nxc.snu.ac.kr

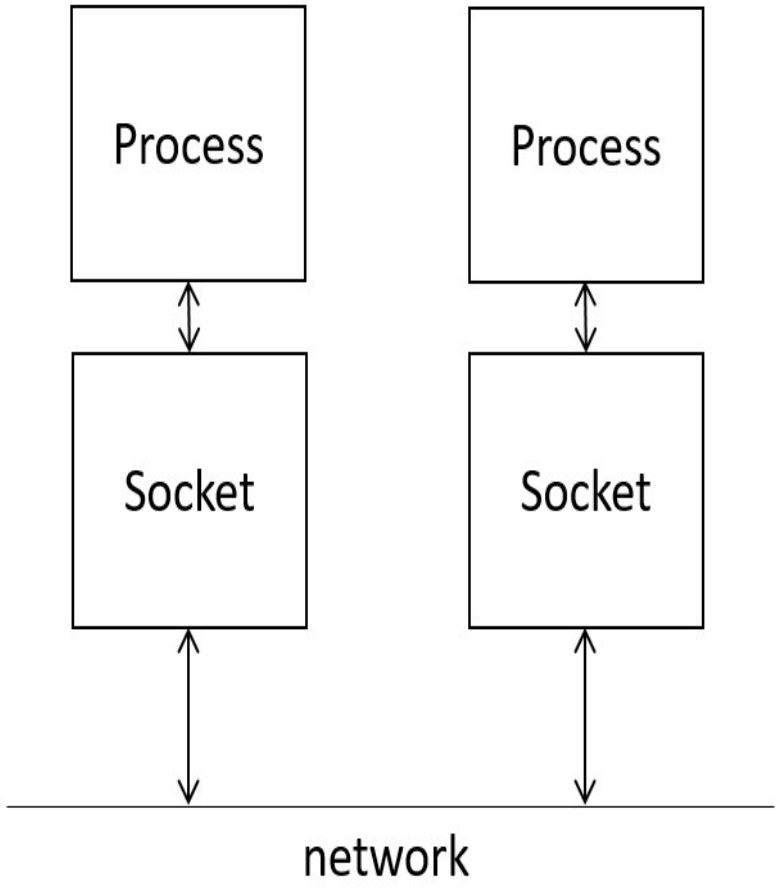
shinikpark@snu.ac.kr

Socket Programming Tutorial

SOCKET PROGRAMMING BASICS

What is Socket?

- With socket, two different processes can communicate each other.
- Socket is nothing but a ***file***.
- So, socket has ***a file descriptor***, which is just an integer to identify opened file.
- Two different processes have sockets and they ***read received data from socket*** and ***write to socket for sending data*** to network.



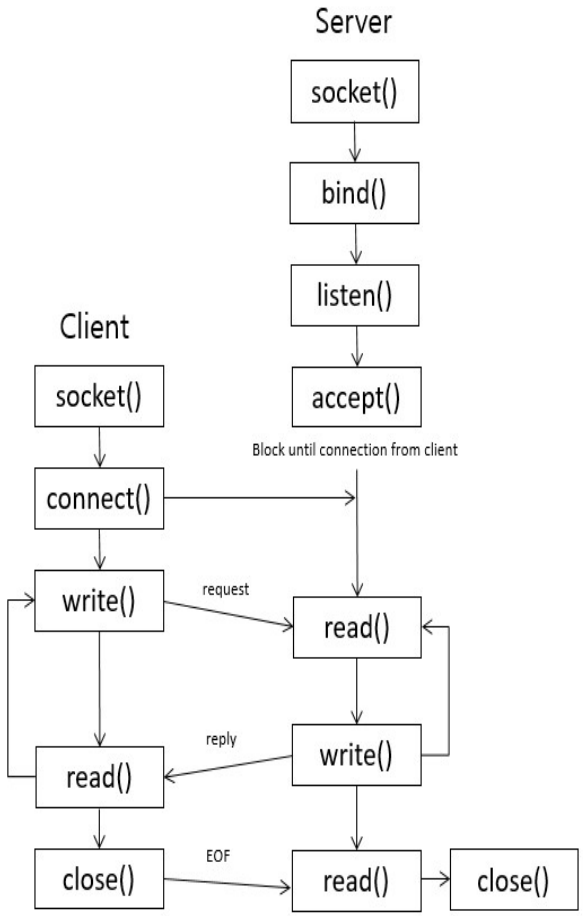
Socket Types

- Stream Sockets
  - Use TCP (Transmission Control Protocol) for data transmission.
  - Delivery in a networked environment is guaranteed.
  - Items arrive in the same order.

- Datagram Sockets
  - Use UDP (User Datagram Protocol) for data transmission.
  - Delivery in a networked environment is not guaranteed.
  - Connectionless (Build a packet with the destination information and send it out.)

Client & Server Overview

- ***Client: Request to server for information.***
  - Create a socket with the $\boldsymbol{socket}()$ system call.
  - Connect socket to the address of the server using the $\boldsymbol{connect}()$ system call.
  - Send and receive data with $\boldsymbol{read}()$ and $\boldsymbol{write}()$ system calls.
- ***Server: Takes request from the clients, perform required processing and send it to the client.***
  - Create a socket with the $\boldsymbol{socket}()$ system call.
  - Bind the socket to an address (IP + port) using the $\boldsymbol{bind}()$ system call.
  - Listen for connections with the $\boldsymbol{listen}()$ system call.
  - Accept a connection with the $\boldsymbol{accept}()$ system call. This call typically blocks the connection until a client connects with the server
  - Send and receive data with $\boldsymbol{read}()$ and $\boldsymbol{write}()$ system calls.



Socket Address Structure

- You will use socket functions, and most of the socket functions use socket address structures.
- ***sockaddr*** : generic socket address structure

```
struct sockaddr {
    // represents an address family, most of cases AF_INET)
    unsigned short     sa_family;

    // 14 bytes of protocol specific address, for the internet
    // family, port number IP address (sockaddr_in) is used
    char               sa_data[14];
}
```

Socket Address Structures

- ***sockaddr_in*** : one type of *sockaddr*, it represents port number IP address

```
struct sockaddr_in {
    short int          sin_family;  // AF_INET
    unsigned short int sin_port;    // 16-bit port number
    struct in_addr     sin_addr;    // 32-bit IP address
    unsigned char      sin_zero[8];
}
```

- ***in_addr*** : structure used in above *sockaddr_in*

```
struct in_addr {
    unsigned long s_addr;
}
```

Socket Address Structures

- ***hostnet*** : contains information related to host

```
struct hostent {
    char *h_name;        // e.g. unist.ac.kr
    char **h_aliases;    // list of host name alias
    int h_addrtype;      // AF_INET
    int h_length;        // length of ip address
    char **h_addr_list; // points to structure in_addr
    #define h_addr h_addr_list[0]
};
```

Network Byte Orders

- All computers doesn't store bytes in same order.
  - Little Endian: Low-order byte is stored on the starting addresses
  - Big Endian: High-order byte is stored on the starting addresses
- To make machines with different byte order communicate with each other, Internet protocol specify a canonical byte order convention for data transmitted over the network.
- **sin_*port*** and **sin_*addr*** of ***sockaddr_in*** should be set with this Network Byte Order.

```
htons() : Host to Network Short
htonl() : Host to Network Long
ntohl() : Network to Host Long
ntohs() : Network to Host Short
```

IP Address Function

- These functions convert Internet addresses between ASCII strings and network byte ordered binary values (values that are stored in socket address structures)

- int **inet_aton**(const char *strptr, struct in_addr *addrptr)

```
#include <arpa/inet.h>

int retval;
struct in_addr addrptr
memset(&addrptr, '\0', sizeof(addrptr));
retval = inet_aton("68.178.157.132", &addrptr);
```

IP Address Function

- In_addr_t **inet_addr**(const char *strptr)

```
#include <arpa/inet.h>

struct sockaddr_in dest;
memset(&dest, '\0', sizeof(dest));
dest.sin_addr.s_addr = inet_addr("68.178.157.132");
```

- char ***inet_ntoa**(struct in_addr inaddr)

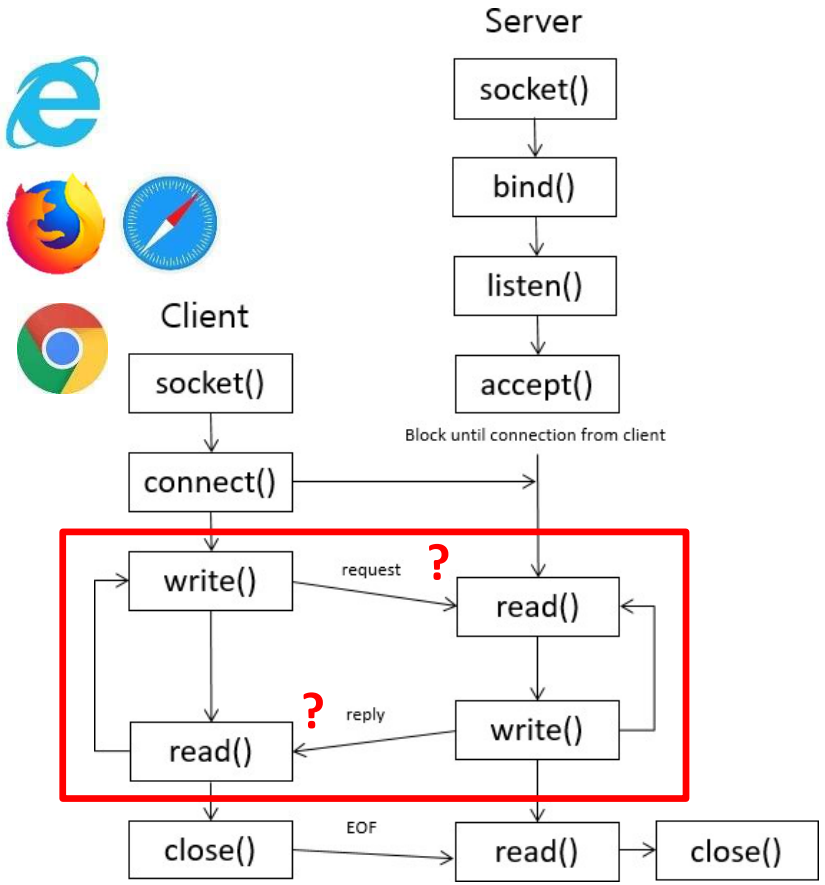
```
#include <arpa/inet.h>

char *ip;
ip = inet_ntoa(dest.sin_addr);
printf("IP Address is : %s\n", ip);
```

Socket Programming Tutorial

SIMPLE HTTP WEB SERVER

HTTP Web Server

<Interaction between Server and Client>



Protocol :
Promise between server and client

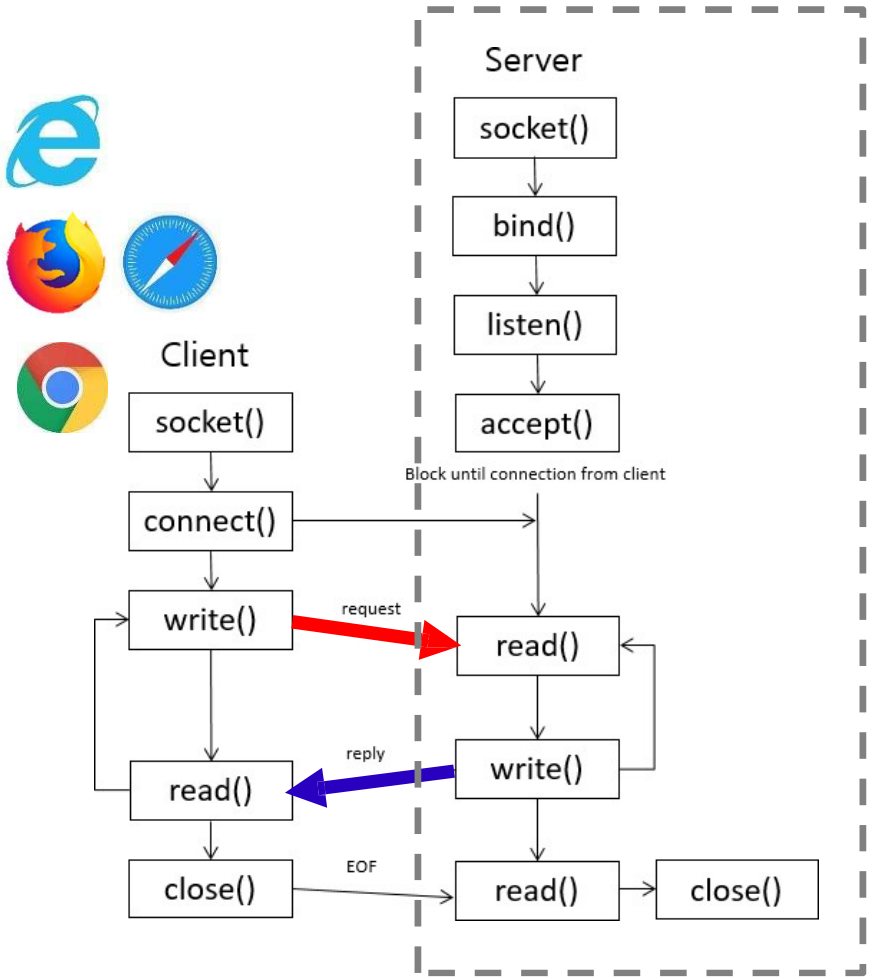
HTTP :
Hyper Text Transfer Protocol

Web browser and Web Server Communicate with HTTP Protocol

(HTTP Method, URL, HTTP Version, Success, Content)

HTTP Web Server

<Interaction between Server and Client>



Client Request

```
GET /restapi/v1.0 HTTP/1.1
Accept: application/json
Authorization: Bearer UExBMDFUMDRQV1Mw…
```

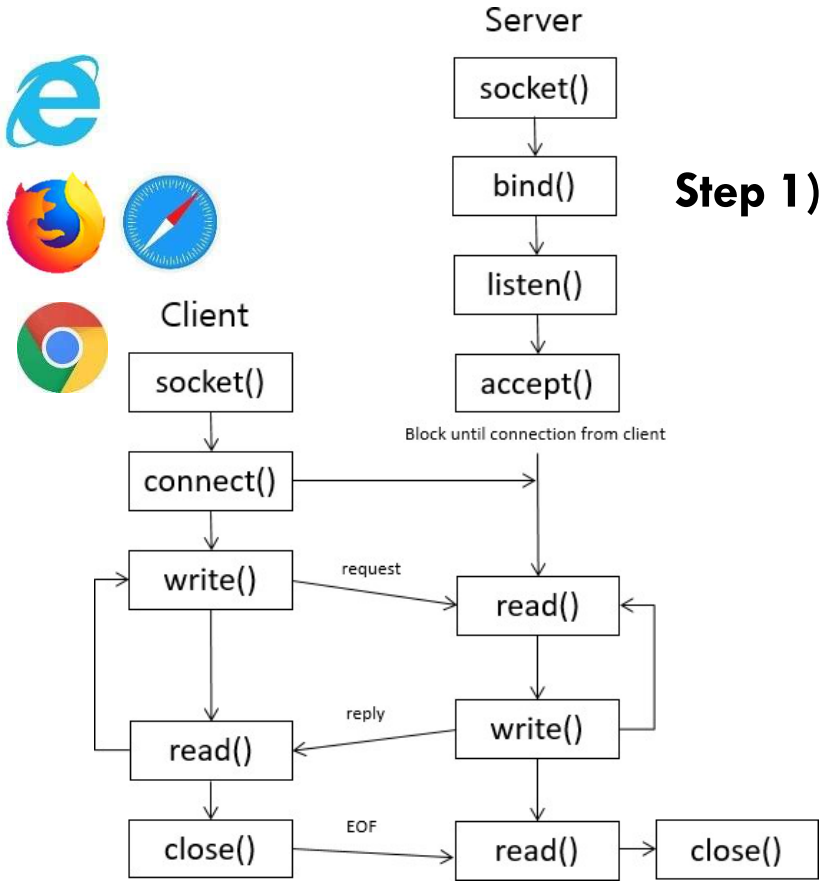
Server Response

```
HTTP/1.1 200 OK
Date: Mon, 23 May 2005 22:38:34 GMT
Content-Type: text/html; charset=UTF-8
Content-Encoding: UTF-8
Content-Length: 138
Last-Modified: Wed, 08 Jan 2003 23:11:55 GMT
Server: Apache/1.3.3.7 (Unix) (Red-Hat/Linux)
ETag: "3f80f-1b6-3e1cb03b"
Accept-Ranges: bytes
Connection: close

<html>
  <head>
    <title>An Example Page</title>
  </head>
  <body> Hello World
  </body>
</html>
```

We will implement this part

HTTP Web Server

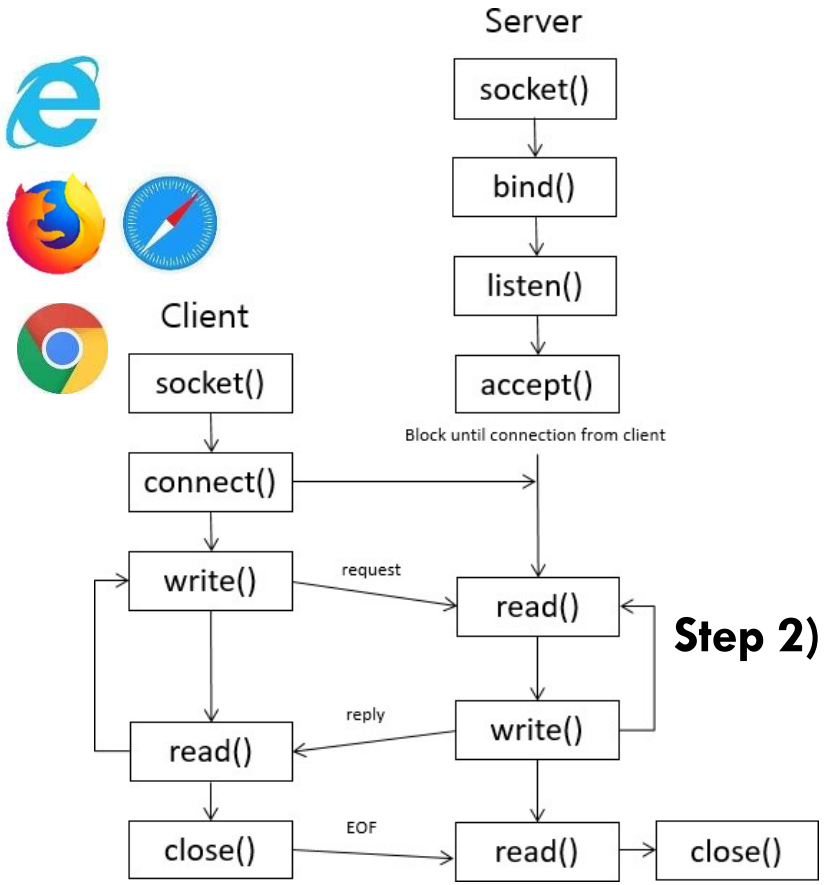
<Interaction between Server and Client>



Step 1) Prepare socket connection
And wait for client connection

HTTP Web Server

<Interaction between Server and Client>



Step 2)

Read client request → Interpret message

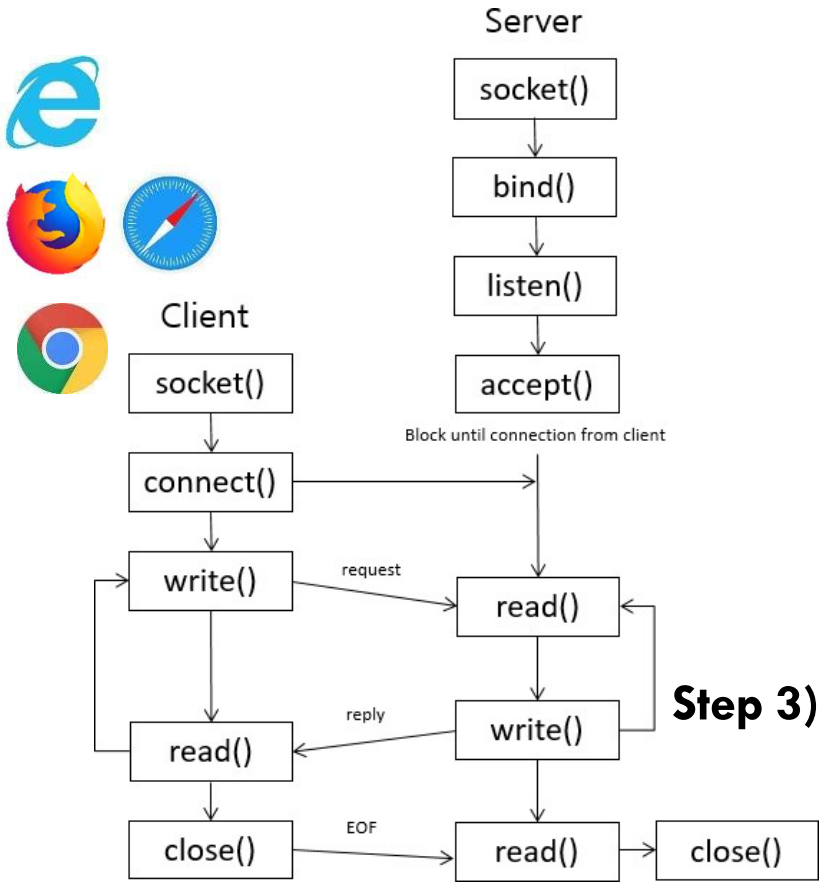
Method Path HTTP Version

```
GET /restapi/v1.0 HTTP/1.1
Accept: application/json
Authorization: Bearer UExBMDFUMDRQV1Mw…
```

Header (Meta Data)

HTTP Web Server

<Interaction between Server and Client>



Step 3)

Make server response → Write message

HTTP Version

Result (200: Success)

```
HTTP/1.1 200 OK
Date: Mon, 23 May 2005 22:38:34 GMT
Content-Type: text/html; charset=UTF-8
Content-Encoding: UTF-8
Content-Length: 138
Last-Modified: Wed, 08 Jan 2003 23:11:55 GMT
Server: Apache/1.3.3.7 (Unix) (Red-Hat/Linux)
ETag: "3f80f-1b6-3e1cb03b"
Accept-Ranges: bytes
Connection: close

<html>
  <head>
    <title>An Example Page</title>
  </head>
  <body> Hello World
  </body>
</html>
```

Header

Content

Let's start programming!

Step 1) Prepare socket connection
And wait for client connection

Download sample code from eTL

```
sockfd = socket(AF_INET, SOCK_STREAM, 0);
```

Make socket with TCP connection
(TCP : SOCK_STREAM)
(UDP : SOCK_DGRAM)

```
int tr = 1;
if (setsockopt(sockfd, SOL_SOCKET, SO_REUSEADDR, &tr, sizeof(int)) == -1) {
  perror("setsockopt");
  exit(1);
}
```

Socket option example)
Setting SO_REUSEADDR option

```
/* Initialize socket structure */
bzero((char *) &serv_addr, sizeof(serv_addr));

serv_addr.sin_family = AF_INET;
serv_addr.sin_addr.s_addr = INADDR_ANY;
serv_addr.sin_port = htons(portno);
```

Initialize socket structure)
Set
AF_INET, INADDR_ANY, Port

Let's start programming!

Step 1) Prepare socket connection
And wait for client connection

Download sample code from eTL

```
/* TODO : Now bind the host address using bind() call.*/
if ( bind(sockfd, (struct sockaddr *)&serv_addr, sizeof(serv_addr)) == -1 ){
  perror("bind error");
  exit(1);
}
```

Bind socket to the server address

```
if ( listen(sockfd, 10) == -1 ){
  perror("listen error");
  exit(1);
}
```

Make socket listen to clients)
2'nd parameter (10) :
10 clients waiting for concurrent connection

```
while (1) {
  newsockfd = accept(sockfd, (struct sockaddr *) &cli_addr, &clilen);
  if ( newsockfd == -1 ){
    perror("accept error");
    exit(1);
  }
  respond(newsockfd);
}
```

Accept : Server waits for Client connection)
You can read and write to connected client
with **newsockfd**

Let's start programming!

Step 2) Read client request
→ Interpret message

Download sample code from eTL

```
void respond(int sock) {
  int offset, bytes;
  char buffer[9000];
  bzero(buffer,9000);

  offset = 0;
  bytes = 0;
  do {
    // bytes < 0 : unexpected error
    // bytes == 0 : client closed connection
    bytes = recv(sock, buffer + offset, 1500, 0);
    offset += bytes;
    // this is end of http request
    if (strncmp(buffer + offset - 4, "\r\n\r\n", 4) == 0) break;
  } while (bytes > 0);
```

Read client request from socket
Until last 4 characters of request become "\r\n\r\n"
(refer to HTTP Protocol)

Result?

Let's start programming!

Step 3) Make server response
→ Write message

Download sample code from eTL

```
char message[] =
"HTTP/1.1 200 OK\r\n
Content-Type: text/html;\r\n\r\n
<html><body>Hello World!
</body></html>\r\n\r\n";

int length = strlen(message);
while(length > 0) {
  printf("send bytes : %d\n", bytes);
  bytes = send(sock, message, length, 0);
  length = length - bytes;
}
printf("close\n");
shutdown(sock, SHUT_RDWR);
close(sock);
```

Write response message to socket
Response should be end with
"\r\n\r\n" as well.

Result?