Socket Programming

Yunmin Go

School of CSEE

Handong Global University

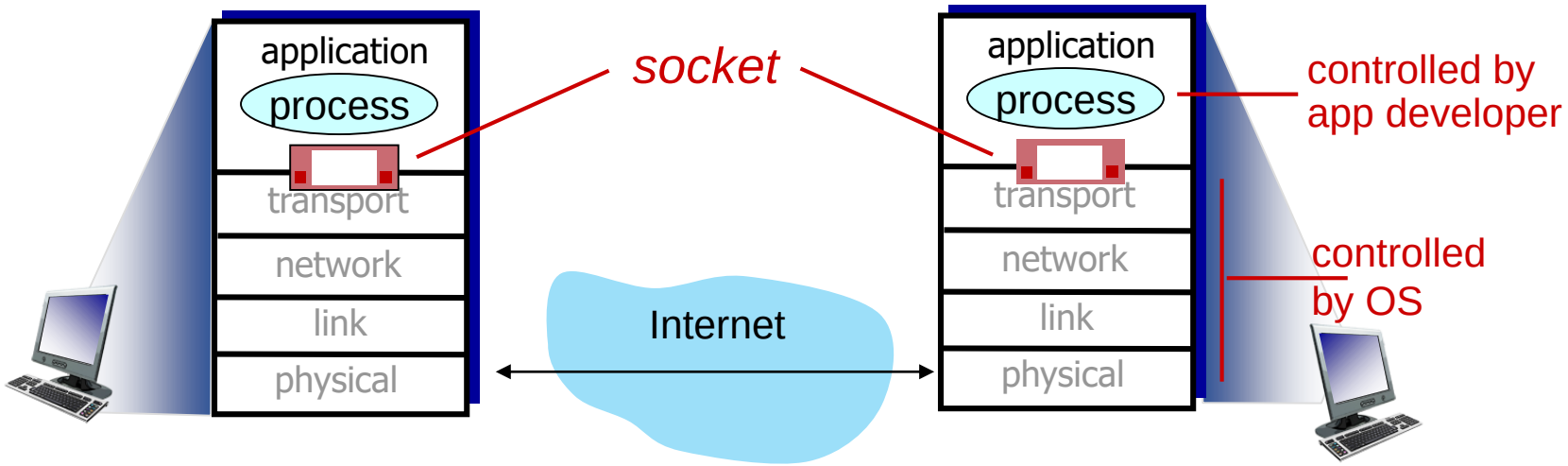
Socket Programming: Roadmap

- Introduction
- TCP Socket programming #1
- TCP Socket programming #2
- UDP Socket programming
- Web Server Example

Socket programming

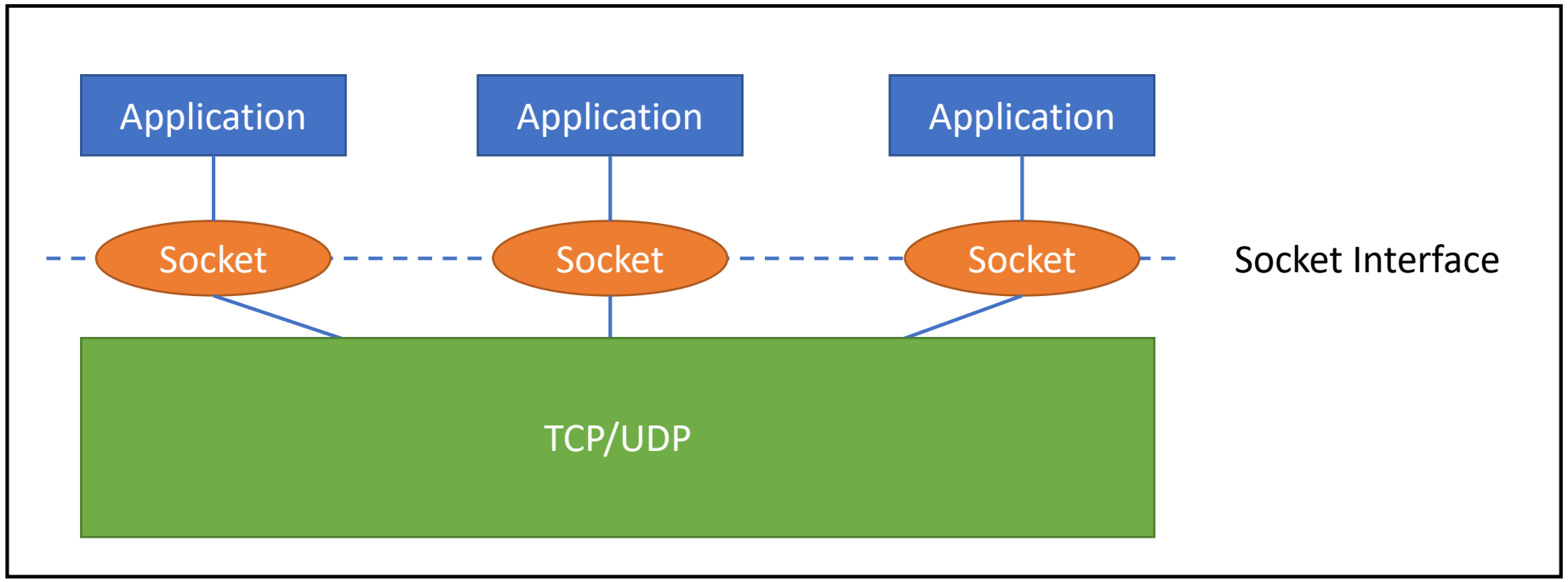
goal: learn how to build client/server applications that communicate using sockets

socket: door between application process and end-end-transport protocol



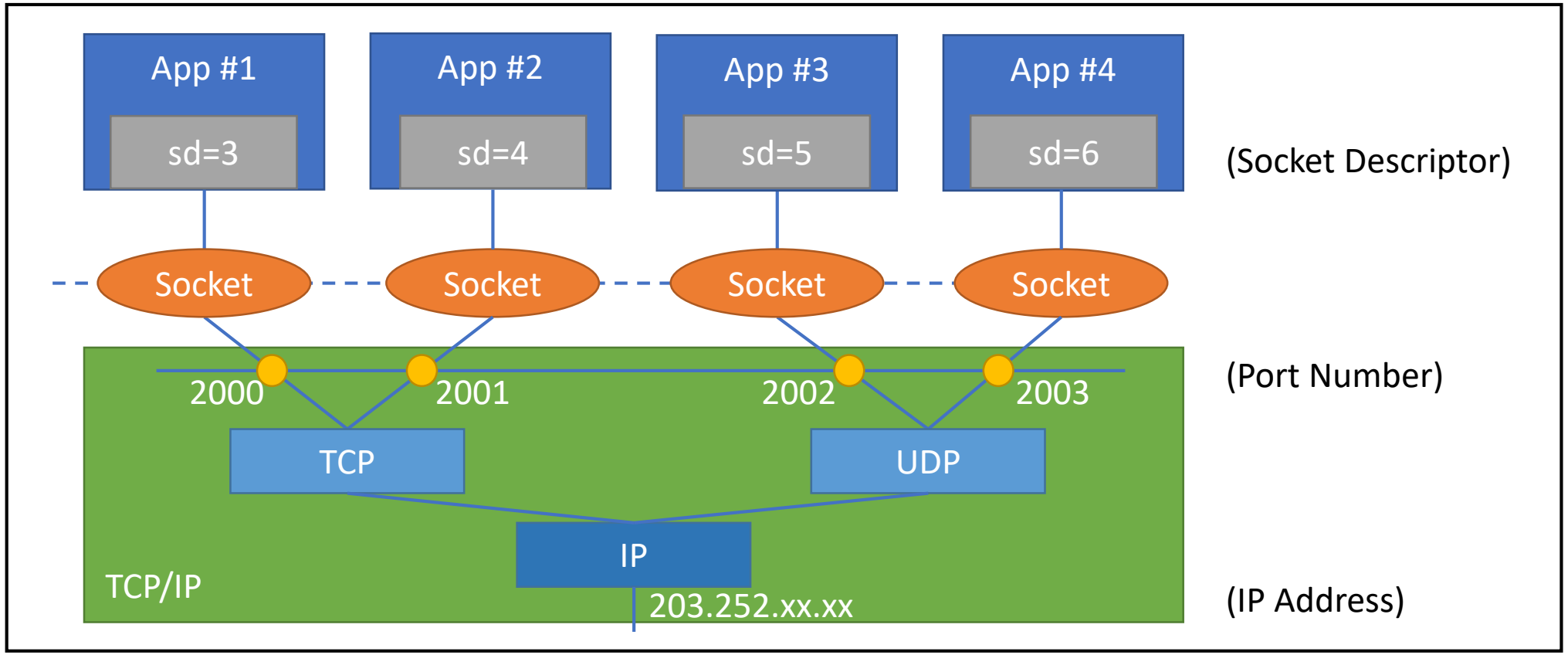
Socket

- Socket interface
  - Located between application and TCP, UDP



Socket

- Socket and TCP/UDP relationship



Socket

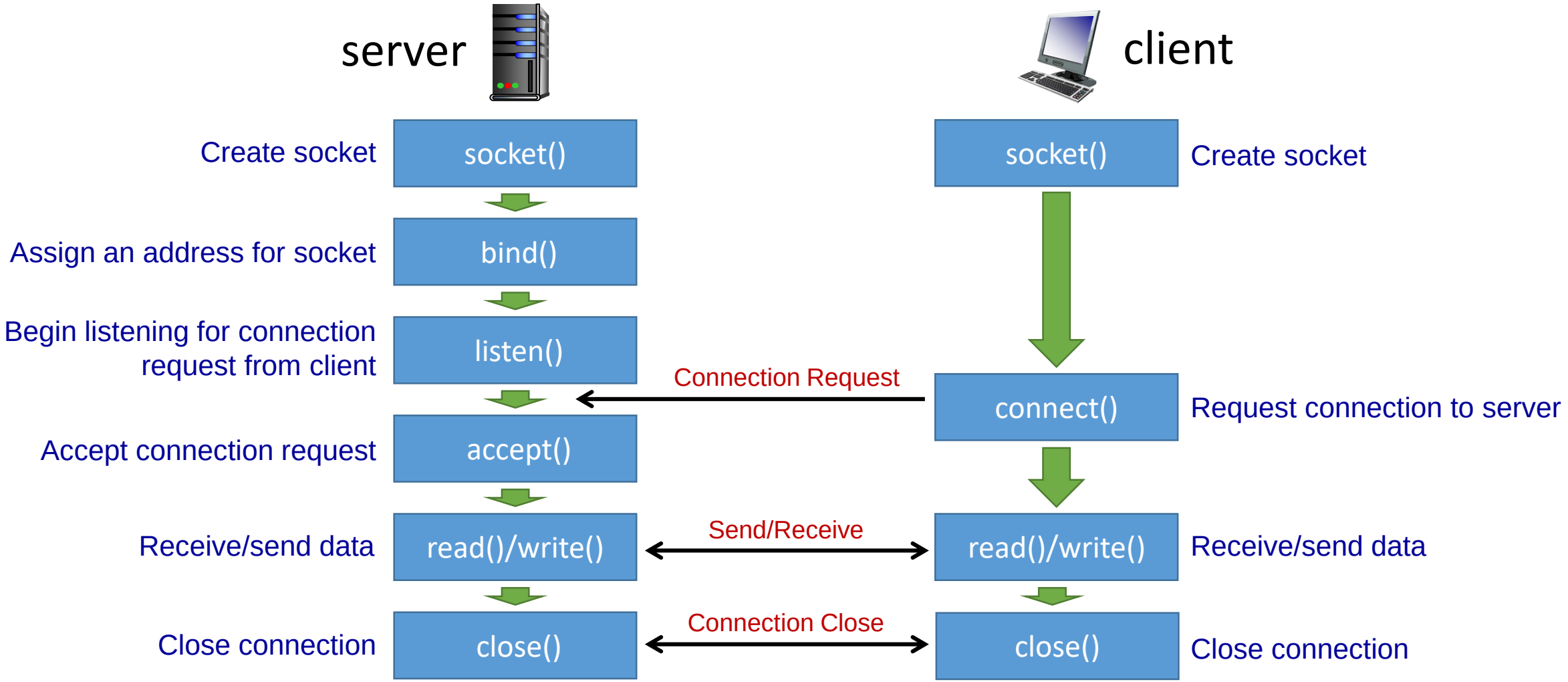
- Each socket is associated with five components
  - Protocol
    - Protocol family and protocol
  - Source address, source port
  - Destination address, destination port

- Where to define components
  - Protocol: socket()
  - Source address and port: bind()
  - Destination address and port: connect(), sendto()

Socket Programming: Roadmap

- Introduction
- TCP Socket programming #1
- TCP Socket programming #2
- UDP Socket programming
- Web Server Example

TCP Server/Client Function Call



TCP Server Example: tcp_server.c

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

void error_handling(char *message);

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

   struct sockaddr_in serv_addr;
   struct sockaddr_in clnt_addr;
   socklen_t clnt_addr_size;

   char message[]="Hello World!";

   if (argc != 2){
      printf("Usage : %s <port>\n", argv[0]);
      exit(1);
   }

   serv_sock = socket(PF_INET, SOCK_STREAM, 0);
   if (serv_sock == -1)
      error_handling("socket() error");

   memset(&serv_addr, 0, sizeof(serv_addr));
   serv_addr.sin_family = AF_INET;
   serv_addr.sin_addr.s_addr = htonl(INADDR_ANY);
   serv_addr.sin_port = htons(atoi(argv[1]));

   if (bind(serv_sock, (struct sockaddr*) &serv_addr, sizeof(serv_addr)) == -1)
      error_handling("bind() error");

   if (listen(serv_sock, 5) == -1)
      error_handling("listen() error");

   clnt_addr_size = sizeof(clnt_addr);
   clnt_sock = accept(serv_sock, (struct sockaddr*)&clnt_addr, &clnt_addr_size);
   if (clnt_sock == -1)
      error_handling("accept() error");

   write(clnt_sock, message, sizeof(message));

   close(clnt_sock);
   close(serv_sock);
   return 0;
}
```

socket()

bind()

listen()

accept()

write()

close()

TCP Client Example: tcp_client.c

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

void error_handling(char *message);

int main(int argc, char* argv[])
{
   int sock;
   struct sockaddr_in serv_addr;
   char message[30];
   int str_len = 0;
   int idx = 0, read_len = 0;

   if (argc != 3) {
      printf("Usage : %s <IP> <port>\n", argv[0]);
      exit(1);
   }
```

```c
   sock = socket(PF_INET, SOCK_STREAM, 0);
   if (sock == -1)
      error_handling("socket() error");

   memset(&serv_addr, 0, sizeof(serv_addr));
   serv_addr.sin_family = AF_INET;
   serv_addr.sin_addr.s_addr = inet_addr(argv[1]);
   serv_addr.sin_port = htons(atoi(argv[2]));

   if (connect(sock, (struct sockaddr*)&serv_addr, sizeof(serv_addr)) == -1)
      error_handling("connect() error!");

   while (read_len = read(sock, &message[idx++], 1))
   {
      if (read_len == -1)
      {
         error_handling("read() error!");
         break;
      }
      str_len += read_len;
   }
   printf("Message from server: %s \n", message);
   printf("Function read call count: %d \n", str_len);

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

socket()

connect()

read()

close()

socket()

```
#include <sys/socket.h>
int socket(int domain, int type, int protocol)
```

Create an endpoint for communication

- *domain*: protocol family
  - PF_INET: IPv4
  - PF_INET6: IPv6
- *type*: type of service
  - SOCK_STREAM: TCP
  - SOCK_DGRAM: UDP
  - SOCK_RAW: raw IP
- *protocol*: specifies the specific protocol
  - Usually 0 which means the default
  - IPPROTO_TCP
  - IPPROTO_UDP

- Return value
  - Success: a file descriptor for the new socket
  - Error: -1
- Example

```
sock = socket(PF_INET, SOCK_STREAM, 0);
if (sock == -1)
   error_handling("socket() error");
```

TCP Socket

```
sock = socket(PF_INET, SOCK_DGRAM, 0);
if (sock == -1)
   error_handling("socket() error");
```

UDP Socket

bind()

```
#include <sys/socket.h>
int bind(int sockfd, const struct sockaddr *addr, socklen_t addrlen);
```

Assign an address to a socket

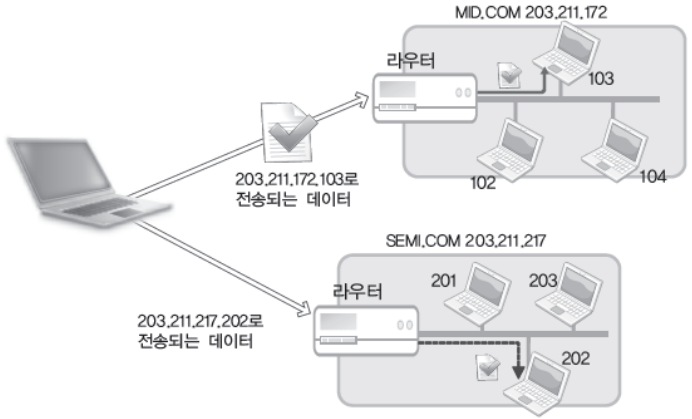
- *sockfd*: file descriptor for the socket
- *addr*: address (IP address and port number) to assign a socket
- *addrlen*: the size (bytes) of the address structure pointed to by *addr*

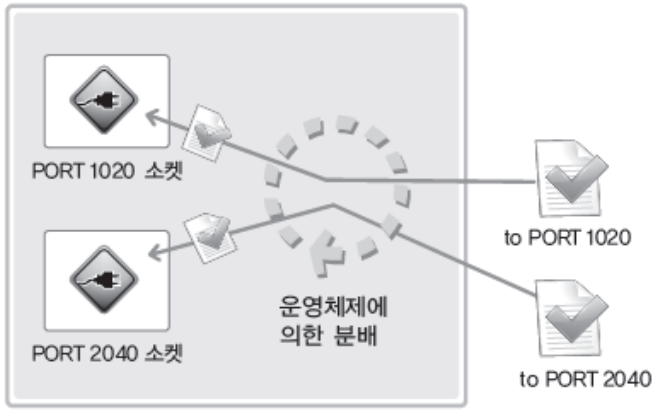
- Return value
  - Success: 0
  - Error: -1
- Example

```
memset(&serv_addr, 0, sizeof(serv_addr));
serv_addr.sin_family = AF_INET;
serv_addr.sin_addr.s_addr = htonl(INADDR_ANY);
serv_addr.sin_port = htons(atoi(argv[1]));

if (bind(serv_sock, (struct sockaddr*) &serv_addr, sizeof(serv_addr)) == -1)
   error_handling("bind() error");
```

Internet Address

- IP Address
  - Used to identify computers on the Internet
  - There are IPv4 (4-byte address) and IPv6 (16bytes address)
  - When creating a socket, we must specify a basic protocol
- Port
  - Used to identify sockets in the computer
  - Port number is represented in 16bits
    → Total range 0 ~ 65535
  - Well-known port: 0 ~ 1023
    - its use has already been decided
    - e.g., 22 for ssh, 80 for web, etc
  - Ephemeral port: a short-lived transport protocol port
    - allocated automatically from a predefined range by the IP stack software
    - e.g., Linux: 32768 to 60999





Address Structure

- Defined in <netinet/in.h>

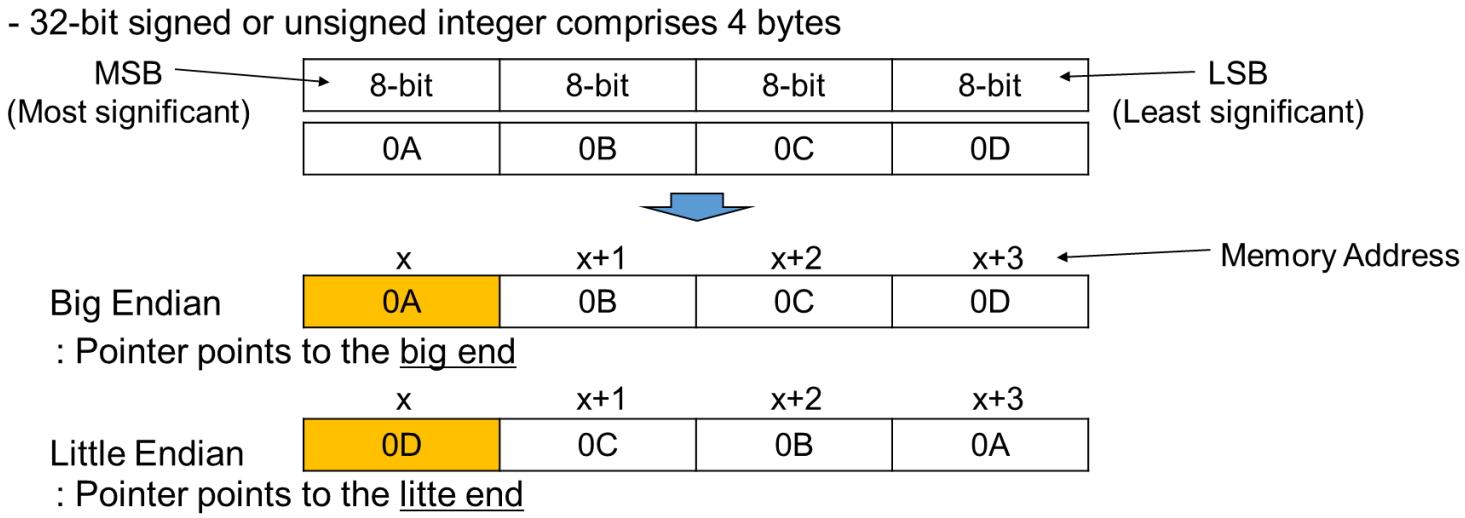
```
struct sockaddr {
    u_short         sa_family;    // Address family
    char            sa_data[14];  // address
};

struct sockaddr_in {
    sa_family_t     sin_family;   // Address family: AF_INET
    uint16_t        sin_port;     // Port number (16bits), Network byte order
    struct in_addr  sin_addr;     // IP address (32bits), Network byte order
    char            sin_zero[8];  // Unused
};

struct in_addr {
    in_addr_t       s_addr;       // IPv4 Internet address (32bits)
};
```

```
memset(&serv_addr, 0, sizeof(serv_addr));
serv_addr.sin_family = AF_INET;
serv_addr.sin_addr.s_addr = htonl(INADDR_ANY);
serv_addr.sin_port = htons(atoi(argv[1]));

if (bind(serv_sock, (struct sockaddr*) &serv_addr, sizeof(serv_addr)) == -1)
    error_handling("bind() error");
```

※ INADDR_ANY: the socket will be bound to all local interfaces

Byte Order Conversion

- Byte ordering
  - Little endian: least significant byte first
  - Big endian: most significant byte first
- Network byte order = big endian
  - Most-significant byte at least address of a word
  - Host byte order depends on host's CPU



htons(), ntohs(), htonl(), ntohl()

```
#include <arpa/inet.h>
uint16_t htons(uint16_t hostshort); // convert unsigned short integer from host byte order to network byte order
uint16_t ntohs(uint16_t netshort);  // convert unsigned short integer from network byte order to host byte order
uint32_t htonl(uint32_t hostlong);  // convert unsigned integer from host byte to network byte order
uint32_t ntohl(uint32_t netlong);   // convert unsigned integer from network byte order to host byte order
```

Convert values between host and network byte order

- Example

```
unsigned short host_port = 0x1234;
unsigned short net_port;
unsigned long host_addr = 0x12345678;
unsigned long net_addr;

net_port = htons(host_port);
net_addr = htonl(host_addr);

printf("Host ordered port: %#x \n", host_port);
printf("Network ordered port: %#x \n", net_port);
printf("Host ordered address: %#lx \n", host_addr);
printf("Network ordered address: %#lx \n", net_addr);
```

```
>> Host ordered port: 0x1234
>> Network ordered port: 0x3412
>> Host ordered address: 0x12345678
>> Network ordered address: 0x78563412
```

inet_addr(), inet_aton ()

```
#include <arpa/inet.h>
in_addr_t inet_addr(const char *string);
int inet_aton(const char *string, struct in_addr *addr)
```

IPv4 address manipulation

- Convert dotted decimal IP address (String) to 32bit big-endian integer
- inet_addr()
  - Return value
    - Success: Internet address (32bit Integer)
    - Error: -1
- inet_aton()
  - Similar with inet_addr(), but the result value is returned using argument
  - Return value
    - Success: 1 (true)
    - Error: 0 (false)

- Example

```
char *addr1 = “1.2.3.4”;
char *addr2 = “127.232.124.79”;
struct sockaddr_in addr_inet;

unsigned long conv_addr = inet_addr(addr1);
if (conv_addr == INADDR_NONE)
   printf("Error occured! \n");
else
   printf("Network ordered integer addr#1: %#lx \n", conv_addr);

if(!inet_aton(addr, &addr_inet.sin_addr))
   printf("Conversion error\n”);
else
   printf(" Network ordered integer addr#2: %#x \n“, addr_inet.sin_addr.s_addr);

>> Network ordered integer addr: 0x4030201
>> Network ordered integer addr: 0x4f7ce87f
```

inet_ntoa()

```
#include <arpa/inet.h>
char *inet_ntoa(struct in_addr adr);
```

IPv4 address manipulation

- Convert 32bit big-endian integer to dotted decimal IP address (String)
- inet_ntoa()
  - Convert string IP address to dotted decimal IP address
  - Return value
    - Success: converted dotted decimal IP address (String)
    - Error: -1
  - Return value

- Example

```
struct sockaddr_in addr1, addr2;
char *str_ptr;
char str_arr[20];

addr1.sin_addr.s_addr = htonl(0x1020304);
addr2.sin_addr.s_addr = htonl(0x1010101);

str_ptr = inet_ntoa(addr1.sin_addr);
strcpy(str_arr, str_ptr);
printf("Dotted-Decimal notation1: %s \n", str_ptr);

>> Dotted-Decimal notation1: 1.2.3.4
```

listen()

```
#include <sys/socket.h>
int listen(int sockfd, int backlog);
```

Listen for connections on a socket

- Tell OS to receive and queue SYN packets
- TCP Server only
- *sockfd*: file descriptor for the socket (socket type should be SOCK_STREAM)
- backlog: the maximum number of connection requests that system can queue while it waits for the server to accept them

- Return value
  - Success: 0
  - Error: -1
- Example

```
if (listen(serv_sock, 5) == -1)
    error_handling("listen() error");
```

accept()

```
#include <sys/socket.h>
int accept(int sockfd, struct sockaddr *addr, socklen_t *addrlen);
```

Accept a connection on a socket

- TCP Server only
- Block until a connection request arrives
- *sockfd*: file descriptor for the accepted socket (socket type should be SOCK_STREAM)
- *addr*: pointer to a sockaddr structure to be filled in with the address of the client socket
- *addrlen*: it will contain the actual size of the client address

- Return value
  - Success: file descriptor for the accepted sock (>0)
  - Error: -1
- Example

```
clnt_addr_size = sizeof(clnt_addr);
clnt_sock = accept(serv_sock, (struct sockaddr*)&clnt_addr, &clnt_addr_size);
if (clnt_sock == -1)
   error_handling("accept() error");
```

connect()

```
#include <sys/socket.h>
int connect(int sockfd, const struct sockaddr *addr, socklen_t addrlen);
```

Initiate a connection on a socket

- For a TCP socket, it establishes a connection to the server
- For a UDP socket, it simply stores the server's address so that the client can use a socket description
- *sockfd*: file descriptor for the socket
- *addr*: address to connect
- *addrlen*: the size (bytes) of the address structure pointed to by *addr*

- Return value
  - Success: 0
  - Failure: -1
- Example

```
memset(&serv_addr, 0, sizeof(serv_addr));
serv_addr.sin_family = AF_INET;
serv_addr.sin_addr.s_addr = inet_addr(argv[1]);
serv_addr.sin_port = htons(atoi(argv[2]));

if (connect(sock, (struct sockaddr*)&serv_addr, sizeof(serv_addr)) == -1)
    error_handling("connect() error!");
```

read()

```
#include <sys/socket.h>
ssize_t read(int fd, void *buf, size_t count);
```

Read from a file descriptor

- Block until data received
- Attempts to read up to *count* bytes from file descriptor *fd* (socket) into the buffer starting at *buf*
- Return value
  - Success: the number of bytes read is returned, and the file position is advanced by this number
    - Zero indicates end of file = connection close
  - Error: -1

- Example

```
while (read_len = read(sock, &message[idx++], 1))
{
   if (read_len == -1)
   {
      error_handling("read() error!");
      break;
   }
   str_len += read_len;
}
```

write()

```
#include <sys/socket.h>
ssize_t write(int fd, const void *buf, size_t count);
```

Write to a file descriptor

- writes up to *count* bytes from the buffer starting at *buf* to the file (socket) referred to by the file descriptor *fd*
- Return value
  - Success: the number of bytes written is returned
  - Error: -1

- Example

```
char message[]="Hello World!";
write(clnt_sock, message, sizeof(message));
```

recv()

```
#include <sys/socket.h>
ssize_t recv(int socket, void *buffer, size_t length, int flags);
```

Receive a message from a connected socket

- Block until data received
- *socket*: socket file descriptor
- *buffer*: points to a buffer where the message should be stored
- *length*: the length in bytes of the buffer pointed to by the *buffer* argument (maximum length of the *buffer*)
- *flags*: type of message reception
  - 0 for regular data
- Return value
  - Success: the number of bytes received
    - Zero indicates the connection close
  - Error: -1
- Example

```
while((str_len = recv(recv_sock, buf, sizeof(buf), 0)) != 0)
{
   if (str_len == -1)
      continue;
   buf[str_len]=0;
    puts(buf);
}
```

send()

```
#include <sys/socket.h>
ssize_t send(int socket, const void *buffer, size_t length, int flags);
```

Send a message on a socket

- Transmit the data in *buffer* upto *length* bytes
- *socket*: socket file descriptor
- *buffer*: buffer containing the message to send
- *length*: the length of the message in bytes.
- *flags*: type of message transmission
  - 0 for regular data
- Return value
  - Success: the number of bytes transmitted
  - Error: -1
- Example

```
write(sock, "123", strlen("123"));
send(sock, "4", strlen("4"), MSG_OOB);
```

close()

```
#include <unistd.h>
int close(int fd);
```

Close a file descriptor

- Prevent any more read and writes to the socket
- If the remote side calls recv(), it will return 0
- If the remote side calls send(), it will receive a signal SIGPIPE and send() will return -1 and errno will be set to EPIPE.
- Return value
  - Success: 0
  - Error: -1
- Example

```
close(clnt_sock);
close(serv_sock);
```

Review: tcp_server.c & tcp_client.c

```
// Server
serv_sock = socket(PF_INET, SOCK_STREAM, 0);
if (serv_sock == -1)
   error_handling("socket() error");

memset(&serv_addr, 0, sizeof(serv_addr));
serv_addr.sin_family = AF_INET;
serv_addr.sin_addr.s_addr = htonl(INADDR_ANY);
serv_addr.sin_port = htons(atoi(argv[1]));

if (bind(serv_sock, (struct sockaddr*) &serv_addr, sizeof(serv_addr)) == -1)
   error_handling("bind() error");

if (listen(serv_sock, 5) == -1)
   error_handling("listen() error");

clnt_addr_size = sizeof(clnt_addr);
clnt_sock = accept(serv_sock, (struct sockaddr*)&clnt_addr, &clnt_addr_size);
if (clnt_sock==-1)
   error_handling("accept() error");

write(clnt_sock, message, sizeof(message));

close(clnt_sock);
close(serv_sock);
```

socket()

bind()

listen()

accept()

write()

close()

```
// Client
sock = socket(PF_INET, SOCK_STREAM, 0);
if (sock == -1)
   error_handling("socket() error");

memset(&serv_addr, 0, sizeof(serv_addr));
serv_addr.sin_family = AF_INET;
serv_addr.sin_addr.s_addr = inet_addr(argv[1]);
serv_addr.sin_port = htons(atoi(argv[2]));

if (connect(sock, (struct sockaddr*)&serv_addr, sizeof(serv_addr)) == -1)
   error_handling("connect() error!");

while (read_len = read(sock, &message[idx++], 1))
{
   if (read_len == -1)
   {
      error_handling("read() error!");
      break;
   }
   str_len += read_len;
}
printf("Message from server: %s \n", message);
printf("Function read call count: %d \n", str_len);

close(sock);
```

socket()

connect()

read()

close()

Socket Programming: Roadmap

- Introduction
- TCP Socket programming #1
- TCP Socket programming #2
- UDP Socket programming
- Web Server Example

TCP: overview RFCs: 793,1122, 2018, 5681, 7323

- point-to-point:
  - one sender, one receiver
- reliable, in-order *byte stream:*
  - no “message boundaries"
- full duplex data:
  - bi-directional data flow in same connection
  - MSS: maximum segment size
- cumulative ACKs
- pipelining:
  - TCP congestion and flow control set window size
- connection-oriented:
  - handshaking (exchange of control messages) initializes sender, receiver state before data exchange
- flow controlled:
  - sender will not overwhelm receiver

No Message boundary in TCP

- A "message boundary" is the separation between two messages being sent over a protocol.
- UDP preserves message boundaries
  - E.g., if the server calls write() twice, the client needs to call read() twice.
- TCP does not preserve message boundaries
  - E.g., even if the server calls write() twice, the client can read all data at once.

```
// Server (from tcp_server2.c)
char message1[] = “Hello World!111”;
char message2[] = “Hello World!222”;
…
write(clnt_sock, message1, sizeof(message1));
write(clnt_sock, message2, sizeof(message2));
```

```
// Client (from tcp_client2.c)
sleep(1);
read_len = read(sock, message, sizeof(message));
printf("Message from server: ");
for (i = 0; I < sizeof(message); i++)
   printf(“%c”, message[i]);
printf(“Read_len: %d \n", read_len );
```

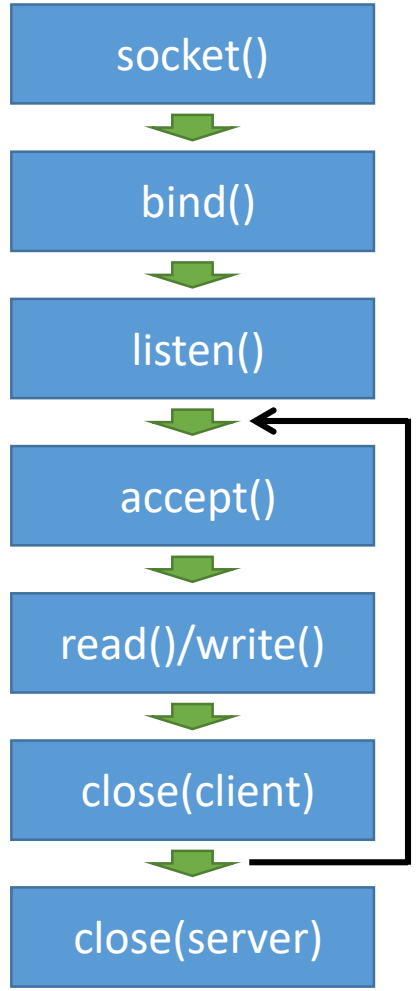
Results

```
yunmin@ym-ubnutu:~/Workspace$ ./hclient2 127.0.0.1 9191
Message from server: Hello World!111Hello World!222◆◆p^◆
Read_len: 32
```

Server Type

- Iterative server
  - A single server process receives and handles incoming requests on a “well-known” port
  - Most UDP servers are iterative

- Concurrent server
  - A separate process to handle each client
  - The main server process creates a new service process to handle each client
  - Most TCP servers are concurrent: for ease of connection management



Iterative vs. Concurrent servers

Iterative server skeleton

```
int sockfd, newsockfd;
if ((sockfd = socket(…)) < 0)
   err_sys(“socket error”);
if ((bind(sockfd, …) < 0)
   err_sys(“bind error”);
if (listen(sockfd, 5))
   err_sys(“listen error”);
for (;;) {
   newsockfd = accept(sockfd, …);
   if (newsockfd < 0)
      err_sys(“accept error”);
   doit(newsockfd);
   close(newsockfd);
}
```

Concurrent server skeleton

```
int sockfd, newsockfd;
if ((sockfd = socket(…)) < 0)
   err_sys(“socket error”);
if ((bind(sockfd, …) < 0)
   err_sys(“bind error”);
if (listen(sockfd, 5))
   err_sys(“listen error”);
for (;;) {
   newsockfd = accept(sockfd, …);
   if (newsockfd < 0)
      err_sys(“accept error”);
   if (fork() == 0) {
      close(sockfd);
      doit(newsockfd);
      exit(0);
   }
   else {
      close(newsockfd);
   }
}
```

Echo Server Example: tcp_echo_server.c

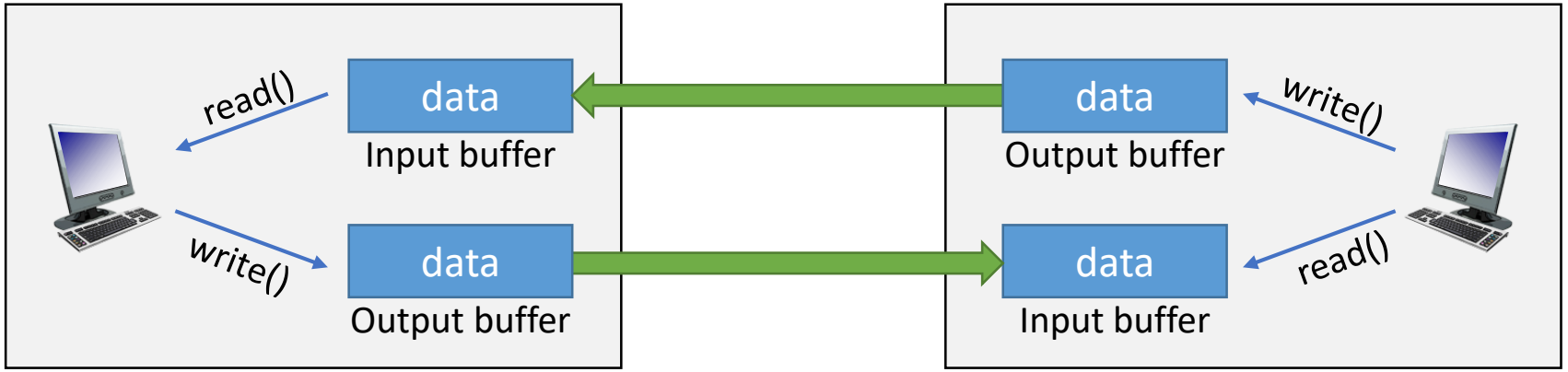
```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <arpa/inet.h>
#include <sys/socket.h>

#define BUF_SIZE 1024
void error_handling(char *message);

int main(int argc, char *argv[])
{
   int serv_sock, clnt_sock;
   char message[BUF_SIZE];
   int str_len, i;

   struct sockaddr_in serv_adr;
   struct sockaddr_in clnt_adr;
   socklen_t clnt_adr_sz;

   if (argc != 2) {
      printf("Usage : %s <port>\n", argv[0]);
      exit(1);
   }

   serv_sock=socket(PF_INET, SOCK_STREAM, 0);
   if (serv_sock == -1)
      error_handling("socket() error");
   memset(&serv_adr, 0, sizeof(serv_adr));
   serv_adr.sin_family = AF_INET;
   serv_adr.sin_addr.s_addr = htonl(INADDR_ANY);
   serv_adr.sin_port = htons(atoi(argv[1]));

   if (bind(serv_sock, (struct sockaddr*)&serv_adr, sizeof(serv_adr)) == -1)
      error_handling("bind() error");

   if (listen(serv_sock, 5) == -1)
      error_handling("listen() error");

   clnt_adr_sz=sizeof(clnt_adr);
   for (i = 0; i < 5; i++)
   {
      clnt_sock = accept(serv_sock, (struct sockaddr*)&clnt_adr, &clnt_adr_sz);
      if (clnt_sock == -1)
         error_handling("accept() error");
      else
         printf("Connected client %d \n", i+1);

      while ((str_len = read(clnt_sock, message, BUF_SIZE)) != 0)
         write(clnt_sock, message, str_len);

      close(clnt_sock);
   }

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

Echo Client Example: tcp_echo_client.c

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

#define BUF_SIZE 1024
void error_handling(char *message);

int main(int argc, char *argv[])
{
   int sock;
   char message[BUF_SIZE];
   int str_len;
   struct sockaddr_in serv_adr;

   if (argc != 3) {
      printf("Usage : %s <IP> <port>\n", argv[0]);
      exit(1);
   }

   sock=socket(PF_INET, SOCK_STREAM, 0);
   if (sock == -1)
      error_handling("socket() error");

   memset(&serv_adr, 0, sizeof(serv_adr));
   serv_adr.sin_family = AF_INET;
   serv_adr.sin_addr.s_addr = inet_addr(argv[1]);
   serv_adr.sin_port = htons(atoi(argv[2]));
   if (connect(sock, (struct sockaddr*)&serv_adr, sizeof(serv_adr)) == -1)
      error_handling("connect() error!");
   else
      puts("Connected.....");

   while(1)
   {
      fputs("Input message(Q to quit): ", stdout);
      fgets(message, BUF_SIZE, stdin);

      if (!strcmp(message,"q\n") || !strcmp(message,"Q\n"))
         break;

      str_len = write(sock, message, strlen(message));

      recv_len = 0;
      while (recv_len < str_len)
      {
         recv_cnt = read(sock, &message[recv_len], BUF_SIZE-1);
         if (recv_cnt == -1)
            error_handling("read() error!");
         recv_len+=recv_cnt;
      }
      message[recv_len]=0;
      printf("Message from server: %s", message);
   }
   close(sock);
   return 0;
}
```

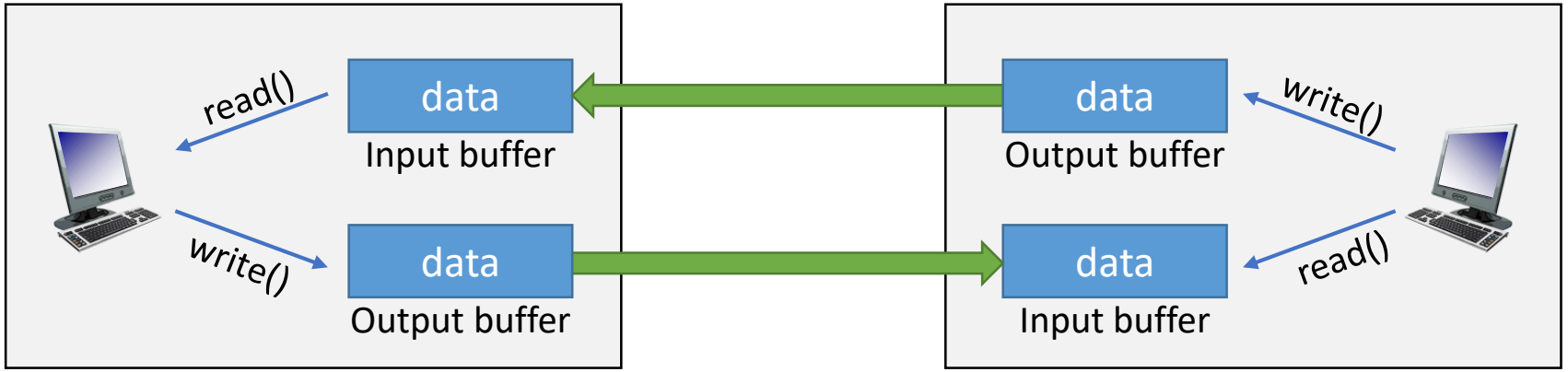
Input & Output Buffer for TCP Socket

- Input buffer and output buffer exists on the TCP socket
- Input buffer and output buffer are created when socket is created
- Even if the socket is closed, data remaining in the output buffer will continue to be transmitted
- When the socket is closed, the data remaining in the input buffer is destroyed



Half-close

- Meaning of close()
  - It means complete destruction of the socket
  - No more I/O is possible
  - If the data transmitting and receiving have not been completed yet, it causes problems
- Half-close
  - Close only one of the input or output streams



shutdown()

```
#include <unistd.h>
int shutdown(int socket, int how);
```

Shut down socket send and receive operations

- This function shall cause all or part of a full-duplex connection on the socket
- *socket*: socket file descriptor
- *how*: type of shutdown
  - SHUT_RD (0): disables further receive operations
  - SHUT_WR (1): disables further send operations
  - SHUT_RDWR (2): disables further send and receive operations
- The shutdown() does not actually free up the socket descriptor. To free the descriptor, use close()

- Return value
  - Success: 0
  - Error: -1
- Example

```
shutdown(clnt_sd, SHUT_WR);
read(clnt_sd, buf, BUF_SIZE);
printf("Message from client: %s \n", buf);

close(clnt_sd); close(serv_sd);
```

File Server Example: tcp_file_server.c

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

#define BUF_SIZE 30
void error_handling(char *message);

int main(int argc, char *argv[])
{
   int serv_sd, clnt_sd;
   FILE * fp;
   char buf[BUF_SIZE];
   int read_cnt;

   struct sockaddr_in serv_adr, clnt_adr;
   socklen_t clnt_adr_sz;

   if (argc != 2) {
      printf("Usage: %s <port>\n", argv[0]);
      exit(1);
   }

   fp = fopen("file_server.c", "rb");
   serv_sd = socket(PF_INET, SOCK_STREAM, 0);
```

```c
   memset(&serv_adr, 0, sizeof(serv_adr));
   serv_adr.sin_family = AF_INET;
   serv_adr.sin_addr.s_addr = htonl(INADDR_ANY);
   serv_adr.sin_port = htons(atoi(argv[1]));

   bind(serv_sd, (struct sockaddr*)&serv_adr, sizeof(serv_adr));
   listen(serv_sd, 5);

   clnt_adr_sz = sizeof(clnt_adr);
   clnt_sd = accept(serv_sd, (struct sockaddr*)&clnt_adr, &clnt_adr_sz);

   while(1)
   {
      read_cnt = fread((void*)buf, 1, BUF_SIZE, fp);
      if (read_cnt < BUF_SIZE)
      {
         write(clnt_sd, buf, read_cnt);
         break;
      }
      write(clnt_sd, buf, BUF_SIZE);
   }

   shutdown(clnt_sd, SHUT_WR);
   read(clnt_sd, buf, BUF_SIZE);
   printf("Message from client: %s \n", buf);

   fclose(fp);
   close(clnt_sd); close(serv_sd);
   return 0;
}
```

File Client Example: tcp_file_client.c

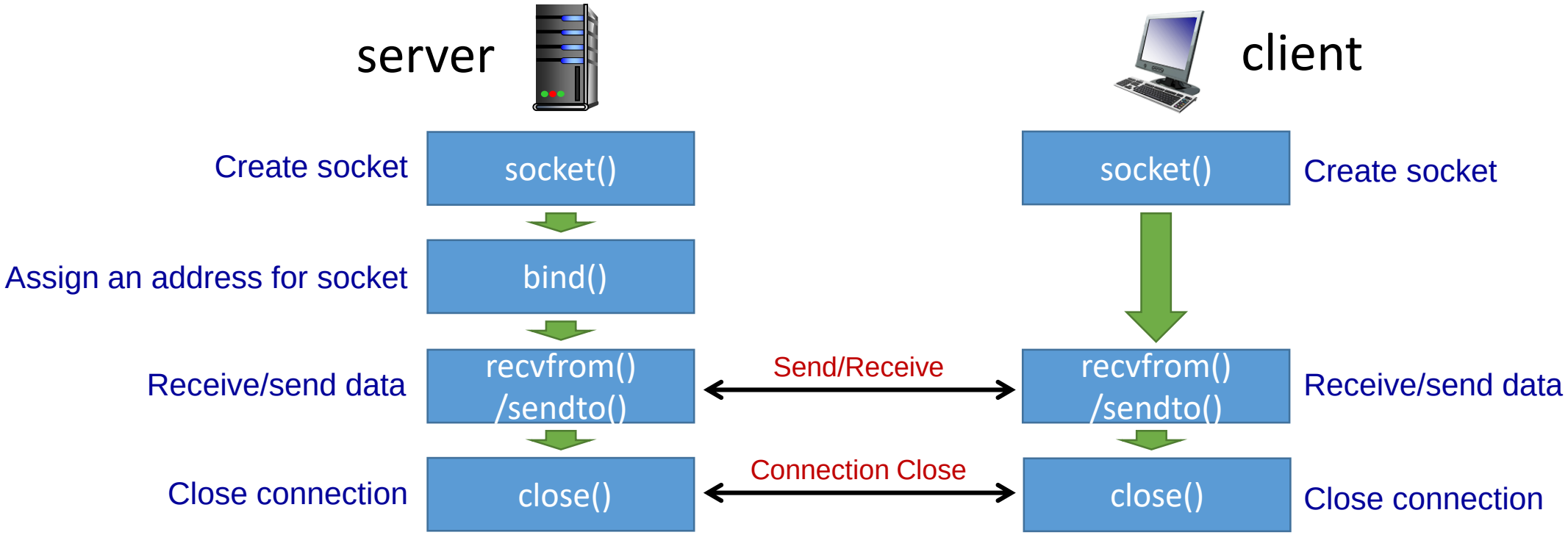
```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <arpa/inet.h>
#include <sys/socket.h>

#define BUF_SIZE 30
void error_handling(char *message);

int main(int argc, char *argv[])
{
   int sd;
   FILE *fp;

   char buf[BUF_SIZE];
   int read_cnt;
   struct sockaddr_in serv_adr;
   if (argc != 3) {
      printf("Usage: %s <IP> <port>\n", argv[0]);
      exit(1);
   }
```

```
   fp = fopen("receive.dat", "wb");
   sd = socket(PF_INET, SOCK_STREAM, 0);

   memset(&serv_adr, 0, sizeof(serv_adr));
   serv_adr.sin_family = AF_INET;
   serv_adr.sin_addr.s_addr = inet_addr(argv[1]);
   serv_adr.sin_port = htons(atoi(argv[2]));

   connect(sd, (struct sockaddr*)&serv_adr, sizeof(serv_adr));

   while((read_cnt = read(sd, buf, BUF_SIZE )) != 0)
      fwrite((void*)buf, 1, read_cnt, fp);

   puts("Received file data");
   write(sd, "Thank you", 10);
   fclose(fp);
   close(sd);
   return 0;
}
```

Socket Programming: Roadmap

- Introduction
- TCP Socket programming #1
- TCP Socket programming #2
- UDP Socket programming
- Web Server Example

UDP Server/Client Function Call



recvfrom()

```
#include <sys/socket.h>
ssize_t recvfrom(int sockfd, void *buffer, size_t length, int flags,
                 struct sockaddr *src_addr, socklen_t *addrlen);
```

Receive a message from a socket

- Block until data received
- *sockfd*: socket file descriptor
- *buffer*: points to a buffer where the message should be stored
- *length*: receives the data up to *length* bytes into *buffer*
- *flags*: type of message reception
  - 0 for regular
- *src_addr*: pointer to a sockaddr structure to be filled in with the address of the peer socket
- *addrlen*: it will contain the actual size of the peer address

- Return value
  - Success: the number of bytes received
    - Zero for EOF
  - Error: -1
- Example

```
str_len = recvfrom(serv_sock, message, BUF_SIZE, 0,
                   (struct sockaddr*)&clnt_adr, &clnt_adr_sz);
```

sendto()

```
#include <sys/socket.h>
ssize_t sendto(int sockfd, const void *buffer, size_t length, int flags,
                        const struct sockaddr *dest_addr, socklen_t *addrlen);
```

Send a message on a socket

- *sockfd*: socket file descriptor
- *buffer*: buffer containing the message to send
- *length*: transmits the data in *buffer* upto *length* bytes
- *flags*: type of message transmission
  - 0 for regular
- *dest_addr*: address to transmit data
- *addrlen*: the size (bytes) of the address structure pointed to by *addr*

- Return value
  - Success: the number of bytes sent
  - Error: -1
- Example

```
sendto(serv_sock, message, str_len, 0,
        (struct sockaddr*)&clnt_adr, clnt_adr_sz);
```

UDP Server Example: udp_echo_server.c

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

#define BUF_SIZE 30
void error_handling(char *message);

int main(int argc, char *argv[])
{
   int serv_sock;
   char message[BUF_SIZE];
   int str_len;
   socklen_t clnt_adr_sz;

   struct sockaddr_in serv_adr, clnt_adr;
   if (argc != 2) {
      printf("Usage : %s <port>\n", argv[0]);
      exit(1);
   }
```

```c
   serv_sock = socket(PF_INET, SOCK_DGRAM, 0);
   if (serv_sock == -1)
      error_handling("UDP socket creation error");

   memset(&serv_adr, 0, sizeof(serv_adr));
   serv_adr.sin_family = AF_INET;
   serv_adr.sin_addr.s_addr = htonl(INADDR_ANY);
   serv_adr.sin_port = htons(atoi(argv[1]));

   if (bind(serv_sock, (struct sockaddr*)&serv_adr, sizeof(serv_adr)) == -1)
      error_handling("bind() error");

   while(1)
   {
      clnt_adr_sz = sizeof(clnt_adr);
      str_len = recvfrom(serv_sock, message, BUF_SIZE, 0,
                         (struct sockaddr*)&clnt_adr, &clnt_adr_sz);
      sendto(serv_sock, message, str_len, 0,
                         (struct sockaddr*)&clnt_adr, clnt_adr_sz);
   }

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

socket()

bind()

recvfrom()/
sendto()

close()

UDP Client Example: udp_echo_client.c

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

#define BUF_SIZE 30
void error_handling(char *message);

int main(int argc, char *argv[])
{
   int sock;
   char message[BUF_SIZE];
   int str_len;
   socklen_t adr_sz;

   struct sockaddr_in serv_adr, from_adr;
   if (argc != 3) {
      printf("Usage : %s <IP> <port>\n", argv[0]);
      exit(1);
   }

   sock = socket(PF_INET, SOCK_DGRAM, 0);
   if (sock == -1)
      error_handling("socket() error");
```

socket()

```c
   memset(&serv_adr, 0, sizeof(serv_adr));
   serv_adr.sin_family=AF_INET;
   serv_adr.sin_addr.s_addr=inet_addr(argv[1]);
   serv_adr.sin_port=htons(atoi(argv[2]));

   while(1)
   {
      fputs("Insert message(q to quit): ", stdout);
      fgets(message, sizeof(message), stdin);
      if (!strcmp(message,"q\n") || !strcmp(message,"Q\n"))
         break;

      sendto(sock, message, strlen(message), 0,
               (struct sockaddr*)&serv_adr, sizeof(serv_adr));
      adr_sz = sizeof(from_adr);
      str_len = recvfrom(sock, message, BUF_SIZE, 0,
                  (struct sockaddr*)&from_adr, &adr_sz);

      message[str_len]=0;
      printf("Message from server: %s", message);
   }

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

sendto()/
recvfrom()

close()

Connected UDP

- Internal procedures in sendto()
  - 1) Allocate a destination IP and port for UDP socket
  - 2) Send data to allocated destination IP and port
  - 3) Deallocate destination IP and port from UDP socket
- Connected UDP
  - We can use the socket that has destination address already assigned
    - Skip address allocate and deallocate procedure
  - We can use write() and receive() instead of sendto() and recvfrom()
  - It does not mean connection-oriented socket like TCP

How to create connected UDP

```
sock = socket(PF_INET, SOCK_DGRAM, 0);
if (sock == -1)
   error_handling("socket() error");

memset(&serv_adr, 0, sizeof(serv_adr));
serv_adr.sin_family = AF_INET;
serv_adr.sin_addr.s_addr = inet_addr(argv[1]);
serv_adr.sin_port = htons(atoi(argv[2]));

connect(sock, (struct sockaddr*)&serv_adr, sizeof(serv_adr));
```

UDP Client Example: udp_echo_con_client.c

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

#define BUF_SIZE 30
void error_handling(char *message);

int main(int argc, char *argv[])
{
   int sock;
   char message[BUF_SIZE];
   int str_len;
   socklen_t adr_sz;

   struct sockaddr_in serv_adr, from_adr;
   if (argc != 3) {
      printf("Usage : %s <IP> <port>\n", argv[0]);
      exit(1);
   }

   sock = socket(PF_INET, SOCK_DGRAM, 0);
   if (sock == -1)
      error_handling("socket() error");
   memset(&serv_adr, 0, sizeof(serv_adr));
   serv_adr.sin_family = AF_INET;
   serv_adr.sin_addr.s_addr = inet_addr(argv[1]);
   serv_adr.sin_port = htons(atoi(argv[2]));

   connect(sock, (struct sockaddr*)&serv_adr, sizeof(serv_adr));

   while(1)
   {
      fputs("Insert message(q to quit): ", stdout);
      fgets(message, sizeof(message), stdin);
      if (!strcmp(message,"q\n") || !strcmp(message,"Q\n"))
         break;

      write(sock, message, strlen(message));

      str_len = read(sock, message, sizeof(message)-1);
      message[str_len] = 0;

      printf("Message from server: %s", message);
   }
   close(sock);
   return 0;
}
```

connect()

write()

read()

Socket Programming: Roadmap

- Introduction
- TCP Socket programming #1
- TCP Socket programming #2
- UDP Socket Programming
- Web Server Example

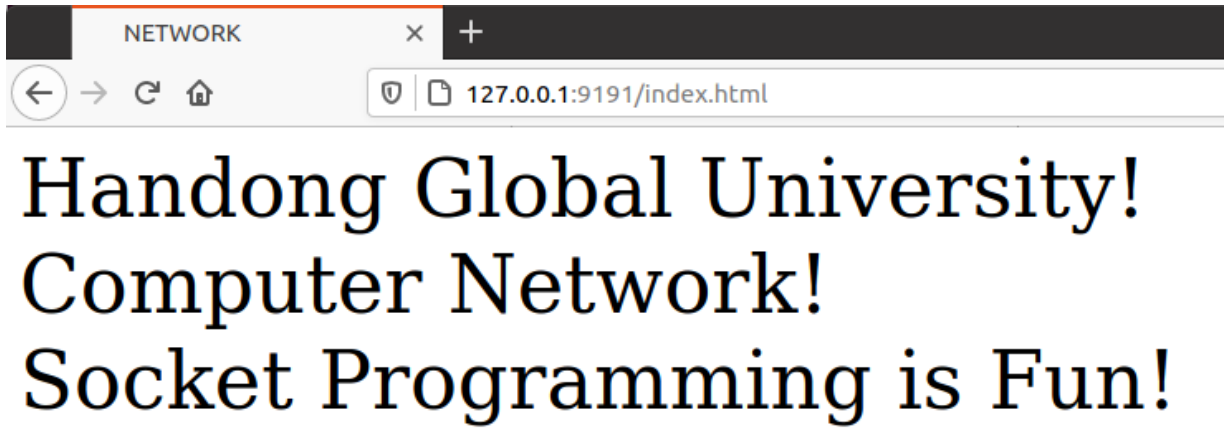
Simple Web Server

- Let's implement a simple web server using TCP socket
- Our web server can provide simple web page by client request

Server

```
yunmin@ym-ubnutu:~/Workspace$ ./webserv_linux 9191
Connection Request : 127.0.0.1:52208
Connection Request : 127.0.0.1:52210
```

Client



- We will implement iterative server, not concurrent server
- For the client, we will use web browser instead of implementing client program

Function Calls for Simple Web Server

server

client

socket()

bind()

listen()

accept()

request_handler()

send_data()

close(client)

close(server)

Open web browser

Type URL

HTTP Request

HTTP Response

Close connection

Connection Request

Request connection to server

GET /index.html HTTP/1.1

Transmit HTTP Request

Receive HTTP Request

HTTP/1.0 200 OK
Index.html

Receive HTTP Response and object file

Send HTTP Response & Requested file

Display web page

Connection Close

Close connection

Close connection

Source Codes: main()

```
int main(int argc, char *argv[])
{
   int serv_sock, clnt_sock;
   struct sockaddr_in serv_adr, clnt_adr;
   int clnt_adr_size;
   char buf[BUF_SIZE];
   pthread_t t_id;
   if (argc != 2) {
      printf("Usage : %s <port>\n", argv[0]);
      exit(1);
   }

   serv_sock=socket(PF_INET, SOCK_STREAM, 0);
   memset(&serv_adr, 0, sizeof(serv_adr));
   serv_adr.sin_family = AF_INET;
   serv_adr.sin_addr.s_addr=htonl(INADDR_ANY);
   serv_adr.sin_port = htons(atoi(argv[1]));
   if (bind(serv_sock, (struct sockaddr*)&serv_adr, sizeof(serv_adr)) == -1)
      error_handling("bind() error");
   if (listen(serv_sock, 20) == -1)
      error_handling("listen() error");
```

```
   while(1)
   {
      clnt_adr_size = sizeof(clnt_adr);
      clnt_sock = accept(serv_sock, (struct sockaddr*)&clnt_adr, &clnt_adr_size);
      printf("Connection Request : %s:%d\n",
      inet_ntoa(clnt_adr.sin_addr), ntohs(clnt_adr.sin_port));

      request_handler(&clnt_sock);

      // pthread_create(&t_id, NULL, request_handler, &clnt_sock);
      // pthread_detach(t_id);
   }
   close(serv_sock);
   return 0;
}
```

Source Codes: request_handler()

```
// void* request_handler(void *arg)
void request_handler(void *arg)
{
   int clnt_sock = *((int*)arg);
   char req_line[SMALL_BUF];
   FILE* clnt_read;
   FILE* clnt_write;

   char method[10];
   char ct[15];
   char file_name[30];

   clnt_read = fdopen(clnt_sock, "r");
   clnt_write = fdopen(dup(clnt_sock), "w");
   fgets(req_line, SMALL_BUF, clnt_read);
   if (strstr(req_line, "HTTP/") == NULL)
   {
      send_error(clnt_write);
      fclose(clnt_read);
      fclose(clnt_write);
      return;
   }
   strcpy(method, strtok(req_line, " /"));
   strcpy(file_name, strtok(NULL, " /"));
   strcpy(ct, content_type(file_name));
   if (strcmp(method, "GET")!=0)
   {
      send_error(clnt_write);
      fclose(clnt_read);
      fclose(clnt_write);
      return;
   }

   fclose(clnt_read);
   send_data(clnt_write, ct, file_name);
}
```

Source Codes: send_data()

```
void send_data(FILE* fp, char* ct, char* file_name)
{
   char protocol[] = "HTTP/1.0 200 OK\r\n";
   char server[] = "Server:Linux Web Server \r\n";
   char cnt_len[] = "Content-length:2048\r\n";
   char cnt_type[SMALL_BUF];
   char buf[BUF_SIZE];
   FILE* send_file;

   sprintf(cnt_type, "Content-type:%s\r\n\r\n", ct);
   send_file = fopen(file_name, "r");
   if (send_file == NULL)
   {
      send_error(fp);
      return;
   }
   fputs(protocol, fp);
   fputs(server, fp);
   fputs(cnt_len, fp);
   fputs(cnt_type, fp);

   while (fgets(buf, BUF_SIZE, send_file) != NULL)
   {
      fputs(buf, fp);
      fflush(fp);
   }
   fflush(fp);
   fclose(fp);
}
```

Source Codes: content_type(), send_error()

```
char* content_type(char* file)
{
   char extension[SMALL_BUF];
   char file_name[SMALL_BUF];
   strcpy(file_name, file);
   strtok(file_name, ".");
   strcpy(extension, strtok(NULL, "."));

   if (!strcmp(extension, "html")||!strcmp(extension, "htm"))
      return "text/html";
   else
      return "text/plain";
}
```

```
void send_error(FILE* fp)
{
   char protocol[] = "HTTP/1.0 400 Bad Request\r\n";
   char server[] = "Server:Linux Web Server \r\n";
   char cnt_len[]  ="Content-length:2048\r\n";
   char cnt_type[] = "Content-type:text/html\r\n\r\n";
   char content[] = "<html><head><title>NETWORK</title></head>"
                    "<body><font size=+5><br>Error! Please re-check the file name
and requet method"
                    "</font></body></html>";
   fputs(protocol, fp);
   fputs(server, fp);
   fputs(cnt_len, fp);
   fputs(cnt_type, fp);
   fflush(fp);
}
```

Appendix

Name-to-Address Conversion

gethostbyname()

```
#include <netdb.h>
struct hostent *gethostbyname(const char *hostname);
```

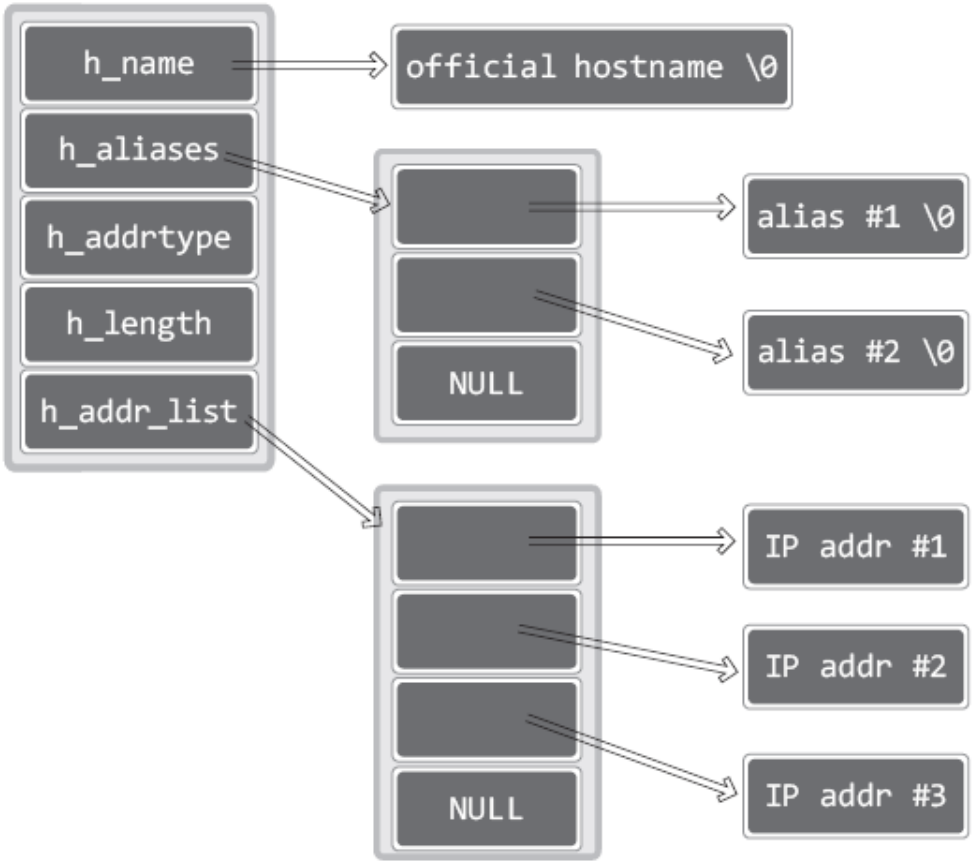
Get host information by taking host name

- We can obtain the host's official name, aliases, and IP addresses by *hostname*
- *hostname*: a hostname or an IPv4 address in standard dot notation
- Return value
  - Success: structure type of *hostent* for given host name
  - Error: NULL

struct hostent

- Defined in netdb.h

```
#define  h_addr haddr_list[0]
struct hostent {
    char    *h_name;        // official name of host
    char    **h_alias;      // alias list
    int     h_addrtype;     // host address type
    int     h_length;       // length of address
    char    **h_addr_list;  // list of addresses
};
```



gethostbyaddr()

```
#include <netdb.h>
struct hostent *gethostbyaddr(const char *addr, socklen_t len, int family);
```

Get host information by taking network byte order address

- We can obtain host's official name, aliases, and IP addresses by *addr* (network byte or address)
- *addr*: address of host (type = struct in_addr)
- len: length of *addr* (IPv4=4, IPv6=16)
- *familiy*: address family (AF_INET=IPv4, AF_INET6=IPv6)
- Return value
  - Success: structure type of *hostent* for given host name
  - Error: NULL