Lab 4
UDP Programming and Concurrent Server Programming

Guideline

- Read text materials and practice TCP/UDP Socket Programming
- Use your notebook PC to examine context and produce the source code when you finish the successful practice.
- **Objectives**
  - Understand the UDP server/client programming mechanism
  - Learn the usage of multi-process, multi-thread and I/O multiplexing
- **Practice**

  1. UDP Programming Test
  2. Multi-process server programming
  3. I/O multiplexing (Select / Poll) server programming

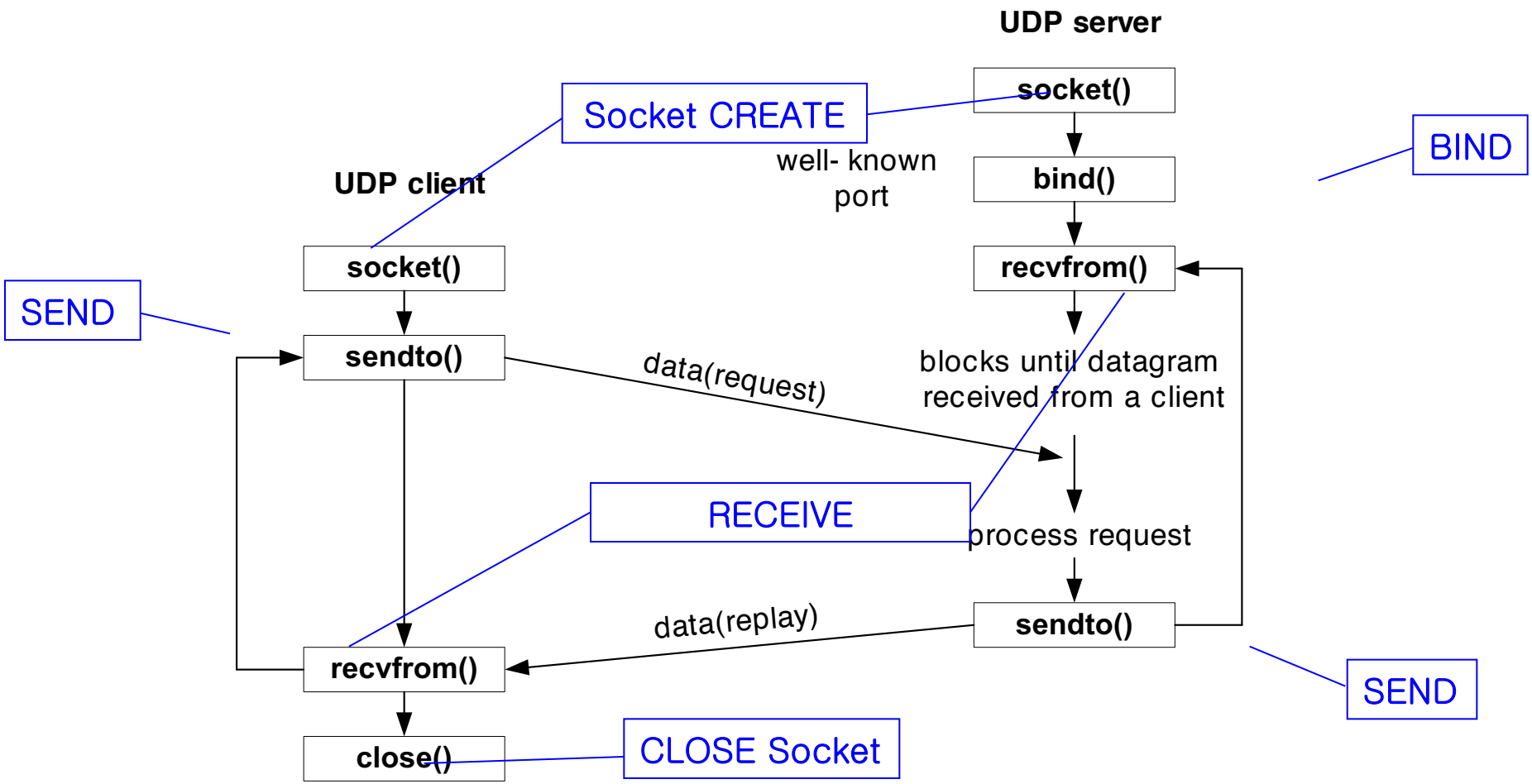
LAB for UDP Testing

Guideline

- **Objectives**
  - Understand the UDP server/client programming mechanism
  - Test simple UDP application – Echo, File Transfer
  - Understand the causes and consequences of failure occurrence in UDP communication and the need for additional logic to address them.
- **Practice**

  1. UDP Echo Server/Client
  2. UDP File Transfer Server/Client
  3. UDP Failures

Socket functions for UDP client-server (Basic model)



Ref.lecture5. slide16

Preparation for Echo Server - Client

- Download source code site: https://github.com/wody34/ee614/
- Requirement file: UDP_Echo_Client.c, UDP_Echo_Server.c
- Compile both c files using gcc
  - gcc UDP_Echo_Client.c -o UDP_Echo_Client
  - gcc UDP_Echo_Server.c -o UDP_Echo_Server
- Execute UDP_Echo_Server and then execute UDP_Echo_Client
  - ./UDP_Echo_Server 10000 (#10000 is port number)
  - ./UDP_Echo_Client 127.0.0.1 10000 (#127.0.0.1 is server ip address)
- In this example, client program try to typing any message. And then, client program send message to server program using sendto() function. Server program receive message using recvfrom() function.

UDP Echo Server/Client

Execution

Server

Test — UDP_Echo_Server 10000 — 82×11

```
SeongHwanui-MacBook-Pro:Test jihwankim$ gcc UDP_Echo_Server.c -o UDP_Echo_Server
SeongHwanui-MacBook-Pro:Test jihwankim$ ./UDP_Echo_Server 10000
Received packet from 127.0.0.1:51119
Data: ANCL


Received packet from 127.0.0.1:51119
Data: LAB4
```

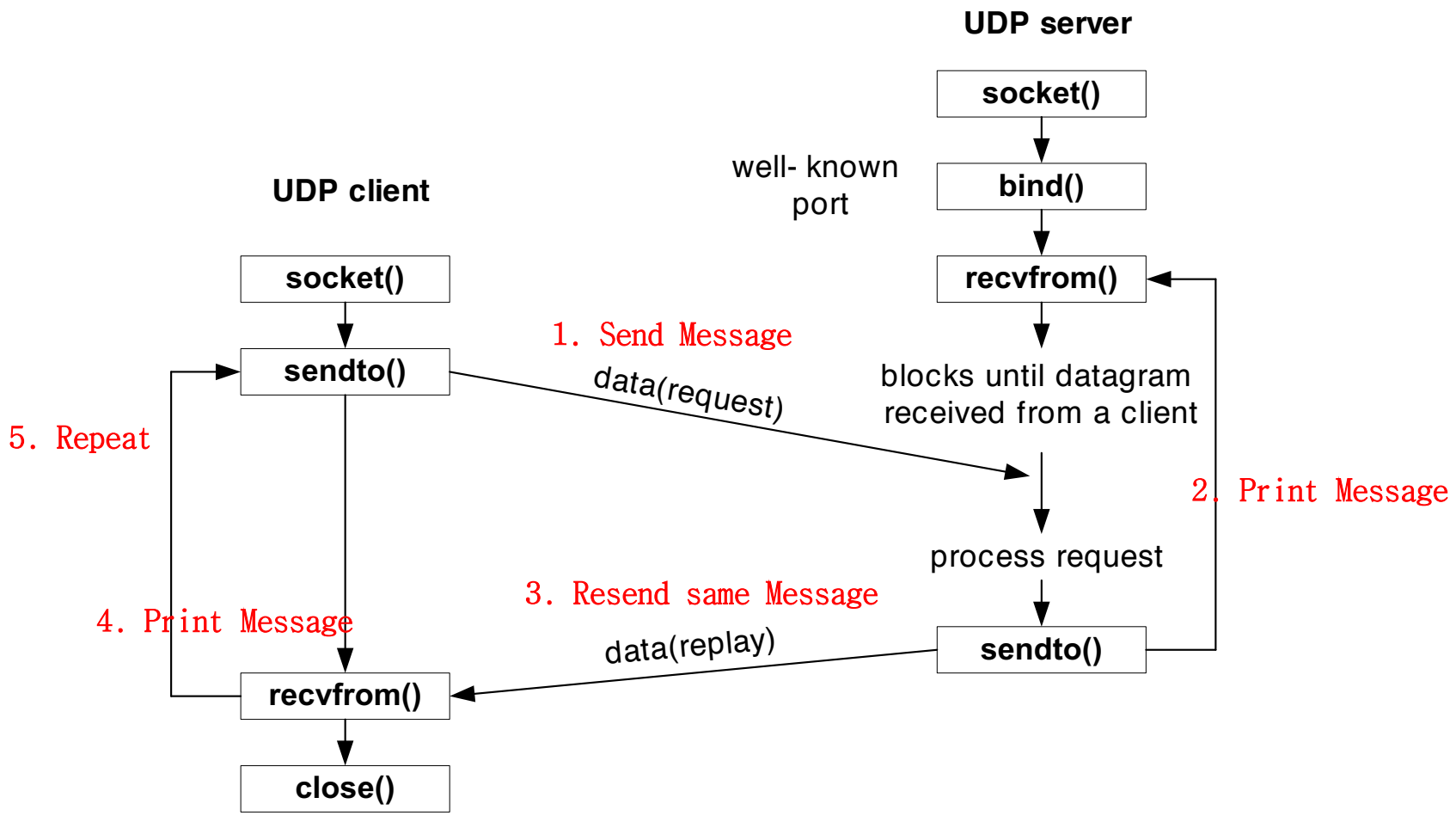
Client

Test — UDP_Echo_Client 127.0.0.1 10000 — 82×9

```
SeongHwanui-MacBook-Pro:Test jihwankim$ gcc UDP_Echo_Client.c -o UDP_Echo_Client
SeongHwanui-MacBook-Pro:Test jihwankim$ ./UDP_Echo_Client 127.0.0.1 10000
ANCL
Received a message from the server : ANCL

LAB4
Received a message from the server : LAB4
```

Make sure that the string you entered is returned properly through the server.

Socket functions for UDP client-server (Echo Server - Client)



UDP Echo Server/Client

UDP Echo Server(1)

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

#define BUFFERSIZE 256

void error_handling(char* msg) {
   perror(msg);
   exit(1);
}

int main(int argc, char **argv)
{
   int sockfd;
   char message[BUFFERSIZE];
   int str_len;
   struct sockaddr_in serv_addr;
   struct sockaddr_in client_addr;
   socklen_t clientlen;

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

   sockfd = socket(AF_INET, SOCK_DGRAM, 0);
   if(sockfd == -1)
      error_handling("socket() error");          1

   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(sockfd, (struct sockaddr*)&serv_addr,
sizeof(serv_addr)) == -1)
      error_handling("bind() error");            2

   for( ; ; ){
      clientlen = sizeof(client_addr);

      str_len = recvfrom(sockfd, message, BUFFERSIZE-
1, 0, (struct sockaddr*)&client_addr, &clientlen);

      message[str_len] = '\0';

      sendto(sockfd, message, str_len, 0, (struct
sockaddr*)&client_addr, clientlen);

      printf("Received packet from %s:%d\nData: %s\n\n",
inet_ntoa(client_addr.sin_addr),
ntohs(client_addr.sin_port), message);
   }
   return 0;
}
```

```
int socket(int family, int type, int protocol);

family:= AF_INET (IPv4 protocol)
type := (SOCK_DGRAM or SOCK_STREAM )
protocol := 0 (IPPROTO_UDP or IPPROTO_TCP)
```

family	Description
AF_INET	IPv4 Protocols
AF_INET6	IPv6 Protocols
AF_LOCAL	Unix Domain Protocols
AF_ROUTE	Routing sockets
AF_KEY	Key socket

type	Description
SOCK_STREAM	Stream socket
SOCK_DGRAM	Datagram socket
SOCK_SEQPACKET	Sequenced packet socket
SOCK_RAW	Raw socket

protocol	Description
IPPROTO_TCP	TCP transport protocol
IPPROTO_UDP	UDP transport protocol
IPPROTO_SCTP	SCTP transport protocol

```
int bind(int sockfd, struct sockaddr *my_addr,
int addrlen);

sockfd := socket descriptor (returned from socket())
my_addr := socket address, struct sockaddr_in is used
addrlen := sizeof(struct sockaddr)
```

Ref.lecture5. slide17

UDP Echo Server/Client

UDP Echo Server(2)

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

#define BUFFERSIZE 256

void error_handling(char* msg) {
   perror(msg);
   exit(1);
}

int main(int argc, char **argv)
{
   int sockfd;
   char message[BUFFERSIZE];
   int str_len;
   struct sockaddr_in serv_addr;
   struct sockaddr_in client_addr;
   socklen_t clientlen;

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

   sockfd = socket(AF_INET, SOCK_DGRAM, 0);
   if(sockfd == -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(sockfd, (struct sockaddr*)&serv_addr,
sizeof(serv_addr)) == -1)
      error_handling("bind() error");

   for( ; ; ){
      clientlen = sizeof(client_addr);

      str_len = recvfrom(sockfd, message, BUFFERSIZE-
1, 0, (struct sockaddr*)&client_addr, &clientlen);   3

      message[str_len] = '\0';

      sendto(sockfd, message, str_len, 0, (struct
sockaddr*)&client_addr, clientlen);   4

      printf("Received packet from %s:%d\nData: %s\n\n",
inet_ntoa(client_addr.sin_addr),
ntohs(client_addr.sin_port), message);
   }
   return 0;
}
```

```
int recvfrom(int sockfd, void *buf, int len,
unsigned int flags, struct sockaddr *from, int
*fromlen);

buf := buffer to receive the message
len := length of the buffer ( "don' t give me more!" )
from := socket address of the process that sent the data
fromlen := sizeof(struct sockaddr)
flags := 0

returned := the number of bytes received
```

```
int sendto(int sockfd, const void *msg, int len,
int flags, const struct sockaddr *to, int tolen);

msg := message you want to send
len := length of the message
flags := 0
to := socket address of the remote process
tolen := sizeof(struct sockaddr)
returned := the number of bytes actually sent
```


Ref.lecture5. slide19

UDP Echo Server/Client

UDP Echo Server(3)

1 sockfd = socket(AF_INET, SOCK_DGRAM, 0);

This code is create socket part. You should insert second factor as SOCK_DGRAM for using UDP.

2 bind(sockfd, (struct sockaddr*)&serv_addr, sizeof(serv_addr));

This code is try bind socket part.

3 str_len = recvfrom(sockfd, message, BUFFERSIZE-1, 0, (struct sockaddr*)&client_addr, &clientlen);

This code is to receive message what is user input some message in client program. Second factor "message" is variable to receive message, third factor is variable's length. Commonly last factor is struct sockaddr's size. However, unlike sendto function, in recvfrom function you should insert last factor as address value of struct sockaddr's size.
recvfrom function's return value save "str_len" as the number of bytes received.

4 sendto(sockfd, message, str_len, 0, (struct sockaddr*)&client_addr, clientlen);

This code is to send message to client program. Second factor "message" is variable with message, third factor is variable's length. Last factor is struct sockaddr's size. sendto function's return value save as the number of bytes actually sent.

UDP Echo Server/Client

UDP Echo Client(1)

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

#define BUFFERSIZE 256

void error_handling(char* msg) {
   perror(msg);
   exit(1);
}

int main(int argc, char **argv)
{
   int sockfd;
   char message[BUFFERSIZE];
   char message2[BUFFERSIZE];
   int str_len;
   struct sockaddr_in serv_addr;
   socklen_t serverlen;

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

   sockfd = socket(AF_INET, SOCK_DGRAM, 0);
   if(sockfd == -1)
      error_handling("socket() error");
```

1

```
   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]));

   while(1){
      serverlen = sizeof(serv_addr);
      fgets(message,BUFFERSIZE,stdin);
      sendto(sockfd, message, strlen(message), 0, (struct
sockaddr*)&serv_addr, serverlen);
```

2

```
      str_len = recvfrom(sockfd, message2,
BUFFERSIZE-1, 0, (struct sockaddr*)&serv_addr,
&serverlen);
```

3

```
      message2[str_len] = '\0';

      printf("Received a message from the server :
%s\n",message2);
   }
   close(sockfd);
```

4

```
   return 0;
}
```

```
int socket(int family, int type, int protocol);
```

family := AF_INET (IPv4 protocol)
type := (SOCK_DGRAM *or* SOCK_STREAM)
protocol := 0 (IPPROTO_UDP *or* IPPROTO_TCP)

```
int sendto(int sockfd, const void *msg, int len,
int flags, const struct sockaddr *to, int tolen);
```

msg := message you want to send
len := length of the message
flags := 0
to := socket address of the remote process
tolen := sizeof(struct sockaddr)
returned := the number of bytes actually sent

```
int recvfrom(int sockfd, void *buf, int len,
unsigned int flags, struct sockaddr *from, int
*fromlen);
```

buf := buffer to receive the message
len := length of the buffer ("don't give me more!")
from := socket address of the process that sent the data
fromlen := sizeof(struct sockaddr)
flags := 0

returned := the number of bytes received

```
close (socketfd);
```

UDP Echo Server/Client

UDP Echo Client(2)

1 sockfd = socket(AF_INET, SOCK_DGRAM, 0);

This code is create socket part. You should insert second factor as SOCK_DGRAM for using UDP.

2 sendto(sockfd, message, strlen(message), 0, (struct sockaddr*)&serv_addr, serverlen);

This code is to send message to server program. Second factor "message" is variable including message, third factor is variable's length. Last factor is struct sockaddr's size. sendto function's return value save as the number of bytes actually sent.

3 str_len = recvfrom(sockfd, message2, BUFFERSIZE-1, 0, (struct sockaddr*)&serv_addr, &serverlen);

This code is to receive message from server program. Second factor "message" is variable to receive message, third factor is variable's length. Commonly last factor is struct sockaddr's size. However, unlike sendto function, in recvfrom function you should insert last factor as address value of struct sockaddr's size. recvfrom function's return value save "str_len" as the number of bytes received.

4 close(sockfd);

This code is close socket part.

Preparation for File Transfer Server - Client

- Download source code site: https://github.com/wody34/ee614/
- Requirement file: UDP_FTCli.c, UDP_FTServ.c, student_info.json
- Compile both c files using gcc
  - gcc UDP_FTCli.c -o UDP_FTCli
  - gcc UDP_FTServ.c -o UDP_FTServ
- Execute UDP_FTServ and then execute UDP_FTServ
  - ./UDP_FTServ (#port number is given source code)
  - ./UDP_FTCli 127.0.0.1 /(student_info.json file path)/student_info.json /(want to copy file path)/(copy file name)
- In this example, client program send the copy file path and origin file's pieces split by 1024 byte until the end of origin file. Server program receive copy file path and then create the file at that path. And server program receive origin file's pieces and write copy file from the beginning to the next.

UDP File Transfer Server/Client

Execution

UDP_FT_Server

Compile & Execution

Copy the file & print buf

```
server — -bash — 73×23
SeongHwanui-MacBook-Pro:server jihwankim$ gcc UDP_FTServ.c -o UDP_FTServ
SeongHwanui-MacBook-Pro:server jihwankim$ ./UDP_FTServ
Received from client: [/Users/jihwankim/Downloads/Test/student_info.json]
{
  "students": [
    {
      "name": "jack",
      "age": "29",
      "major": "computing"
    },
    {
      "name": "peter",
      "age": "26",
      "major": "math"
    },
    {
      "name": "sam",
      "age": "27",
      "major": "electronic"
    }
  ]
}
SeongHwanui-MacBook-Pro:server jihwankim$
```

UDP_FT_Client

Compile & Execution

Copy done!

```
Test — -bash — 84×5
SeongHwanui-MacBook-Pro:Test jihwankim$ gcc UDP_FTCli.c -o UDP_FTCli
SeongHwanui-MacBook-Pro:Test jihwankim$ ./UDP_FTCli 127.0.0.1 /Users/jihwankim/Downl
oads/Test/server/student_info.json /Users/jihwankim/Downloads/Test/student_info.json
Filename sent.
SeongHwanui-MacBook-Pro:Test jihwankim$
```

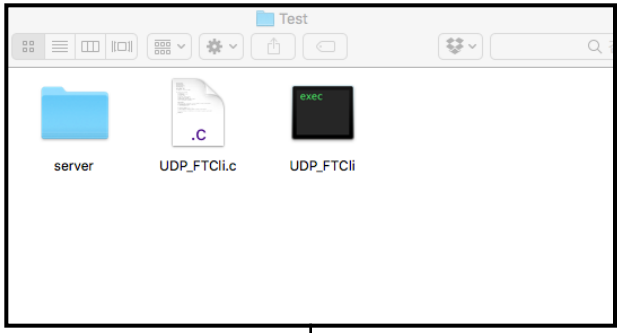
Origin File Path

Copy File Path

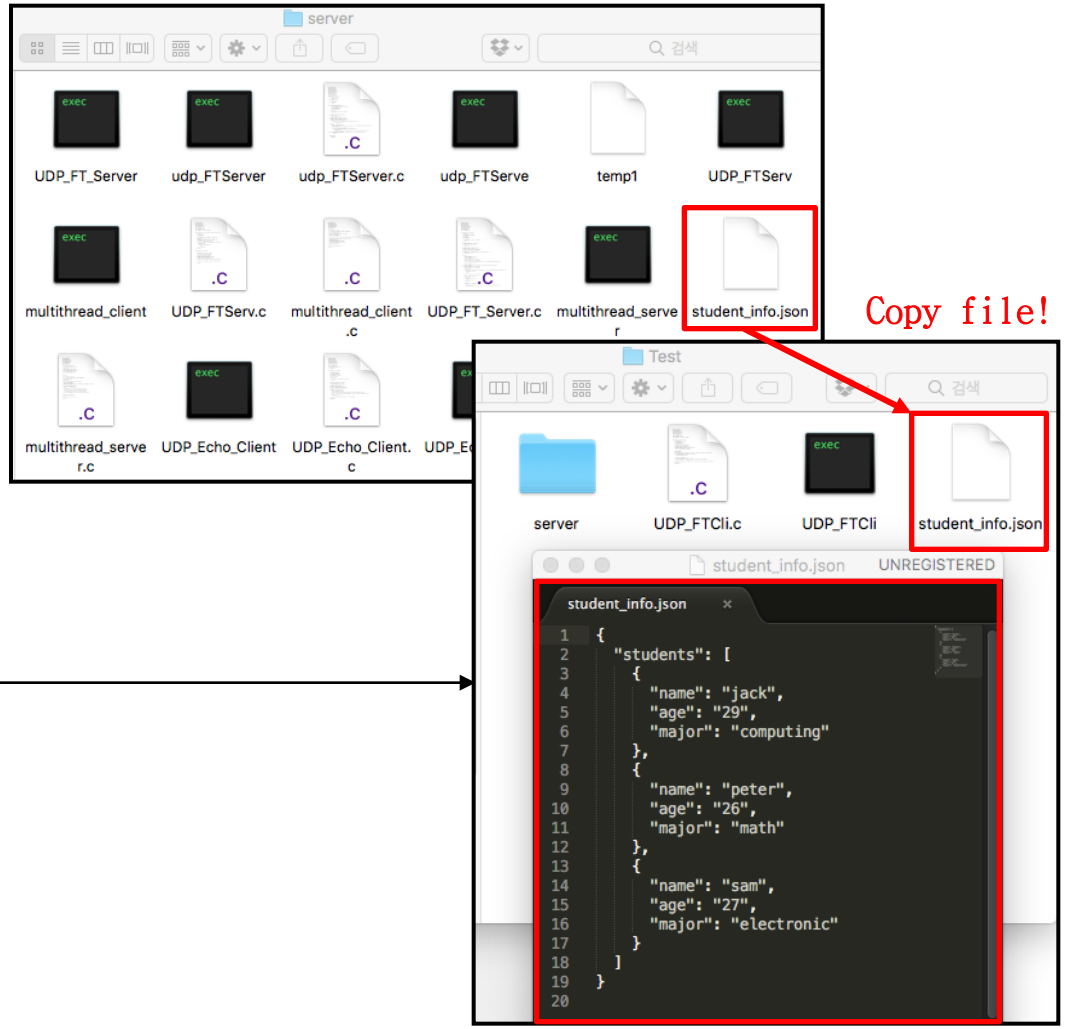
UDP File Transfer Server/Client

Execution

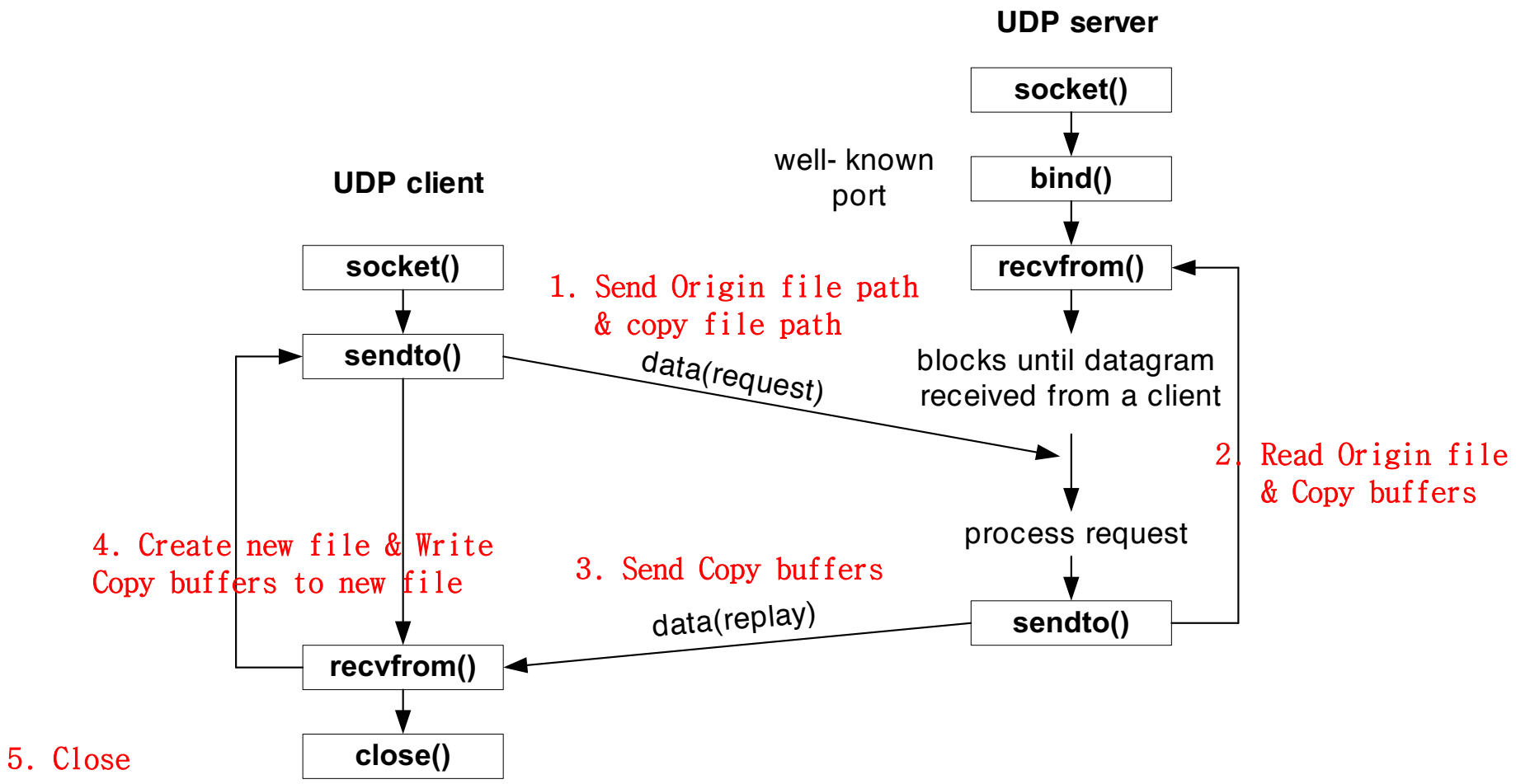
Before execution...



After execution...



Socket functions for UDP client-server (File Transfer Server - Client)



UDP File Transfer Server/Client

UDP File Transfer Server(1)

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

#define SERV_PORT 49312
#define MAXLINE   1024

char *END_FLAG = "================END";

void run(int sockfd, struct sockaddr *cliaddr, socklen_t
clilen)
{
   int n, fd;
   socklen_t len;
   char buf[MAXLINE];

   len = clilen;
   n = recvfrom(sockfd, buf, MAXLINE, 0, cliaddr,
&len);                                                    3
   buf[n] = 0;
   printf("Received from client: [%s]\n", buf);
   sendto(sockfd, "ok", strlen("ok"), 0, cliaddr, len);   4
   fd = open(buf, O_RDWR | O_CREAT, 0666);                5

   while ((n = recvfrom(sockfd, buf, MAXLINE, 0,
cliaddr, &len))) {                                        6
      buf[n] = 0;
      printf("%s", buf);
      if (!(strcmp(buf, END_FLAG))) {
         break;
      }
      write(fd, buf, n);                                  7
   }
   close(fd);
}

int main(int argc, char **argv)
{
   int sockfd;
   struct sockaddr_in servaddr, cliaddr;

   sockfd = socket(AF_INET, SOCK_DGRAM, 0);               1
   bzero(&servaddr, sizeof(servaddr));
   servaddr.sin_family = AF_INET;
   servaddr.sin_addr.s_addr = htonl(INADDR_ANY);
   servaddr.sin_port = htons(SERV_PORT);

   bind(sockfd, (struct sockaddr *) &servaddr,
sizeof(servaddr));                                        2

   run(sockfd, (struct sockaddr *) &cliaddr,
sizeof(cliaddr));

   return 0;
}
```

int socket(int *family*, int *type*, int *protocol*);

family := AF_INET (IPv4 protocol)
type := (SOCK_DGRAM *or* SOCK_STREAM)
protocol := 0 (IPPROTO_UDP *or* IPPROTO_TCP)

int bind(int *sockfd*, struct sockaddr **my_addr*, int *addrlen*);

sockfd := socket descriptor (returned from socket())
my_addr := socket address, struct sockaddr_in is used
addrlen := sizeof(struct sockaddr)

int sendto(int *sockfd*, const void **msg*, int *len*, int *flags*, const struct sockaddr **to*, int *tolen*);

msg := message you want to send
len := length of the message
flags := 0
to := socket address of the remote process
tolen := sizeof(struct sockaddr)
returned := the number of bytes actually sent

int recvfrom(int *sockfd*, void **buf*, int *len*, unsigned int *flags*, struct sockaddr **from*, int **fromlen*);

buf := buffer to receive the message
len := length of the buffer ("don't give me more!")
from := socket address of the process that sent the data
fromlen := sizeof(struct sockaddr)
flags := 0
returned := the number of bytes received

UDP Echo Server/Client

UDP File Transfer Server(2)

1
```
sockfd = socket(AF_INET, SOCK_DGRAM, 0);
```

This code is create socket part. You should insert second factor as SOCK_DGRAM for using UDP.

2
```
bind(sockfd, (struct sockaddr *) &servaddr, sizeof(servaddr));
```

This code is try bind socket part.

3
```
n = recvfrom(sockfd, buf, MAXLINE, 0, cliaddr, &len);
```

Using recvfrom function, server program receive "buf(copy file path)" from client program.

4
```
sendto(sockfd, "ok", strlen("ok"), 0, cliaddr, len);
```

When server program well receive copy file path, using sendto function server program send string "ok" to client program.

5
```
fd = open(buf, O_RDWR | O_CREAT, 0666);
```

Server program create file name as " buf(copy file path)" , when that file name not exist.

UDP Echo Server/Client

UDP File Transfer Server(3)

6

```
n = recvfrom(sockfd, buf, MAXLINE, 0, cliaddr, &len);
```

Using recvfrom function, server program repeatedly receive "buf(origin file's piece)" from client program until end of the origin file's piece.

7

```
write(fd, buf, n);
```

Server program write "buf(origin file's piece)" to copy file in order.

UDP File Transfer Server/Client

UDP File Transfer Client(1)

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

#define SERV_PORT 49312
#define MAXLINE   1024

char *END_FLAG = "================END";

int main(int argc, char **argv)
{
   int sockfd, n, fd;
   struct sockaddr_in servaddr;
   char buf[MAXLINE];
   char *target, *path;

   bzero(&servaddr, sizeof(servaddr));
   servaddr.sin_family = AF_INET;
   servaddr.sin_port = htons(SERV_PORT);
   inet_pton(AF_INET, argv[1], &servaddr.sin_addr);

   sockfd = socket(AF_INET, SOCK_DGRAM, 0);      1

   path = argv[2];
   target = argv[3];
   sendto(sockfd, target, strlen(target), 0, (struct sockaddr
*) &servaddr, sizeof(servaddr));                  2
   n = recvfrom(sockfd, buf, MAXLINE, 0, NULL,
NULL);                                            3
   if (!strncmp(buf, "ok", 2)) {
      printf("Filename sent.\n");
   }
   fd = open(path, O_RDONLY);                     4
   while ((n = read(fd, buf, MAXLINE)) > 0) {
      sendto(sockfd, buf, n, 0, (struct sockaddr *)
&servaddr, sizeof(servaddr));
   }                                              5
   sendto(sockfd, END_FLAG, strlen(END_FLAG), 0,
(struct sockaddr *) &servaddr, sizeof(servaddr));

   close(sockfd);                                 6
   return 0;
}
```

```
int socket(int family, int type, int protocol);

family:= AF_INET (IPv4 protocol)
type := (SOCK_DGRAM or SOCK_STREAM )
protocol := 0 (IPPROTO_UDP or IPPROTO_TCP)
```

```
int sendto(int sockfd, const void *msg, int len,
int flags, const struct sockaddr *to, int tolen);

msg := message you want to send
len := length of the message
flags := 0
to := socket address of the remote process
tolen := sizeof(struct sockaddr)
returned := the number of bytes actually sent
```

```
int recvfrom(int sockfd, void *buf, int len,
unsigned int flags, struct sockaddr *from, int
*fromlen);

buf := buffer to receive the message
len := length of the buffer ( "don't give me more!" )
from := socket address of the process that sent the data
fromlen := sizeof(struct sockaddr)
flags := 0

returned := the number of bytes received
```

```
close (socketfd);
```

UDP Echo Server/Client

UDP File Transfer Client(2)

1

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

This code is create socket part. You should insert second factor as SOCK_DGRAM for using UDP.

2

```
sendto(sockfd, target, strlen(target), 0, (struct sockaddr *) &servaddr, sizeof(servaddr));
```

Using sendto function, client program send "target(argv[3] = copy file path)" to server program.

3

```
n = recvfrom(sockfd, buf, MAXLINE, 0, NULL, NULL);
```

Using recvfrom function, client program receive "buf(ok = process well prior sendto function)" from server program.

4

```
fd = open(path, O_RDONLY);
```

Client program open file name "path(argv[2] = origin file's path)".

UDP Echo Server/Client

UDP File Transfer Client(3)

5

```
while ((n = read(fd, buf, MAXLINE)) > 0) {
        sendto(sockfd, buf, n, 0, (struct sockaddr *) &servaddr, sizeof(servaddr));
    }
```

Client program read the file amount 1024 bytes in order to end of file and then each piece save the "buf" variable and using sendto function, client program send "buf" variable to server program.

6

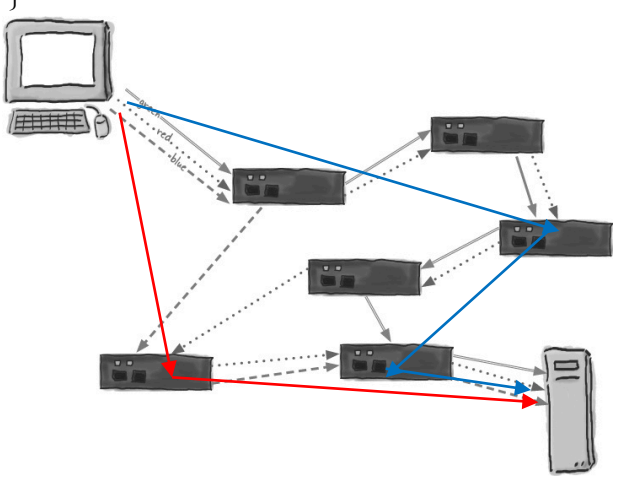
```
close(sockfd);
```

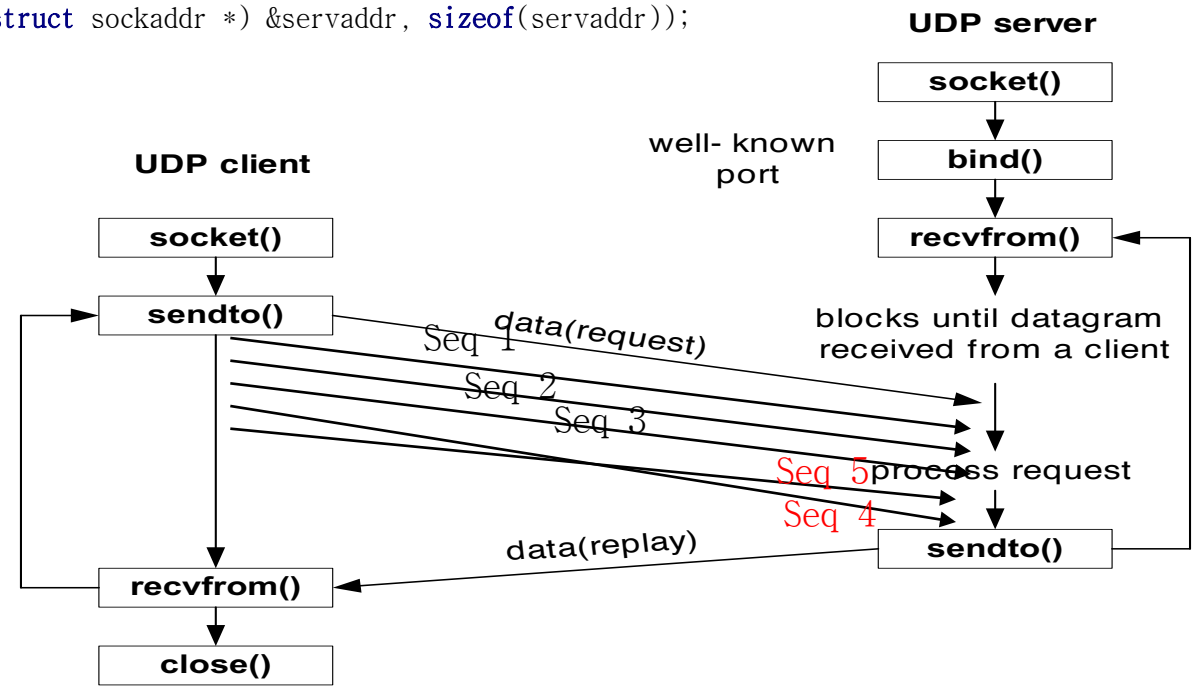
This code is close socket part.

Failure occurrence in UDP communication (1)

- Send bulk packets continuously with sequence number ascending order
- The order of sending and receiving packets may not match

```
for(i = 0; ; ++i) {
      b.sequence = i;
      sendto(sockfd, &b, sizeof(b), 0, (struct sockaddr *) &servaddr, sizeof(servaddr));
  }

struct Bulk {
   int sequence;
   char pad[MAX];
}
```





Failure occurrence in UDP communication (1)

- In lab source repository, download UDP_FailServ.c and UDP_FailCli.c
- Compile source codes
  : gcc –o UDP_FailServ UDP_FailServ.c
  : gcc –o UDP_FailCli UDP_FailCli.c
- Launch server and client
  $ ./UDP_FailServ
  $ ./UDP_FailCli 127.0.0.1
- Figure out that packet is transferred in ordered.

```
Sequence num: 1333381
Sequence num: 1333382
Sequence num: 1333383
Sequence num: 1333384
Sequence num: 1333385
Sequence num: 1333386
Sequence num: 1333387
```

Failure occurrence in UDP communication (1)

Client

```
for(i = 0; ; ++i) {
    b.sequence = i;
    sendto(sockfd, &b, sizeof(struct Bulk), 0, (struct sockaddr *) &servaddr,
     sizeof(servaddr));
}
```

- Client continuously send packet that is 1024 bytes in size with ascending sequence number

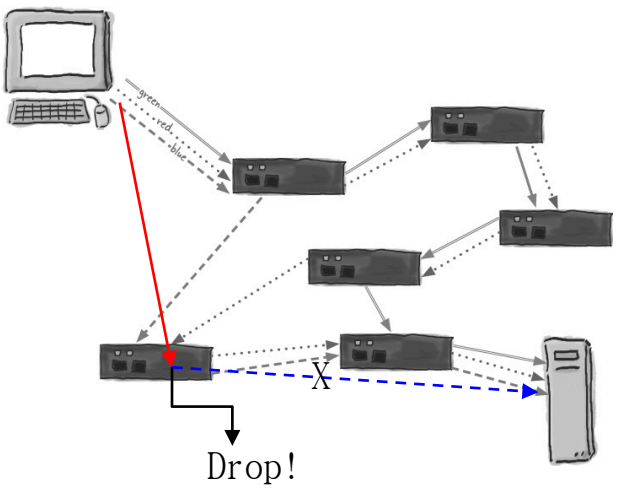
Server

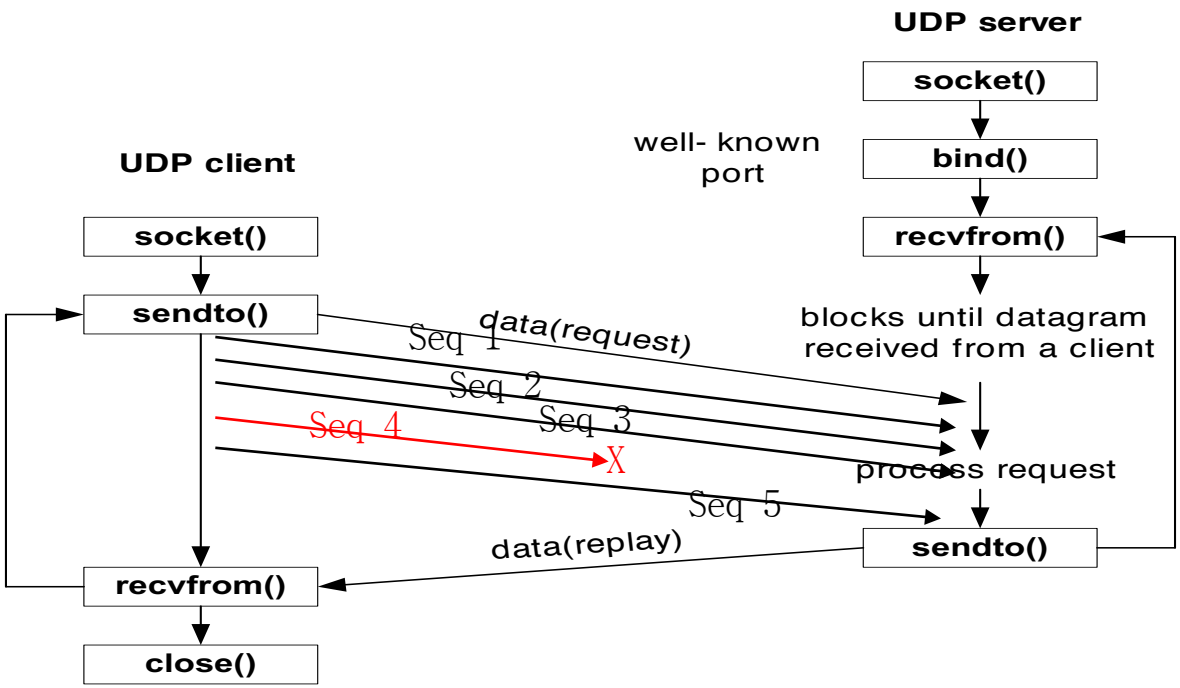
```
while ((n = recvfrom(sockfd, &b, sizeof(struct Bulk), 0, cliaddr, &len))) {
    printf("Sequence num: %d₩n", b.sequence);
    if(b.sequence > max)
        max = b.sequence;
    else
        printf("Error!₩n");
}
```

- Server receive packet and print out sequence number of the packet.
- When an unordered packet arrives, an error message is printed.

Failure occurrence in UDP communication (2)

- Turn off the network (e.g., eth0) for a while during packet transmission
- Then restart the network.
- Can we see packet loss due to network failure?





Failure occurrence in UDP communication (2)

- In this practice we will reuse example code, UDP_FailServ.c and UDP_FailCli.c
- Launch server and client $ ./UDP_FailServ $ ./UDP_FailCli 127.0.0.1
- Turn off the loop back network (lo0) for a while during packet transmission
- See what happens

```
Sequence num: 1077905
Sequence num: 1077906
Sequence num: 1077907
Sequence num: 1077908
Sequence num: 1077909
```

```
[SeongHwanKimui-MacBook-Air:lab4 wody34$ sudo ifconfig lo0 down
[Password:
 SeongHwanKimui-MacBook-Air:lab4 wody34$
```

- Then restart the network.
- Find out where the sequence number starts from.

```
Sequence num: 16511682
Sequence num: 16511683
Sequence num: 16511684
```

```
[SeongHwanKimui-MacBook-Air:lab4 wody34$ sudo ifconfig lo0 up
 SeongHwanKimui-MacBook-Air:lab4 wody34$
```

Lesson in UDP practice (2)

- In an unstable network, packets can be dropped.
- Since UDP does not check whether the packet transmission / reception is successful, it is impossible to check whether there is a dropped packet or not.

Conclusion - Failure occurrence in UDP communication

- UDP is a lightweight protocol that by design doesn't handle things like packet sequencing.
- TCP is a better choice if you want robust packet delivery and sequencing.
- UDP is generally designed for applications where packet loss is acceptable or preferable to the delay which TCP incurs when it has to re-request packets.
- UDP is therefore commonly used for media streaming.

LAB for Concurrent Server Programming

Test Concurrent Server

- TA demonstrate non-concurrent server (modified version of unp/tcpservcli/tcpserv01.c)
  - Simple echo server
- Please connect to the TA's server with unp/tcpservcli/tcpcli01 and test the echo function (IP Address: **143.248.x.x** to be announced in class)
  - $ ./tcpcli01 143.248.xxx.xxx

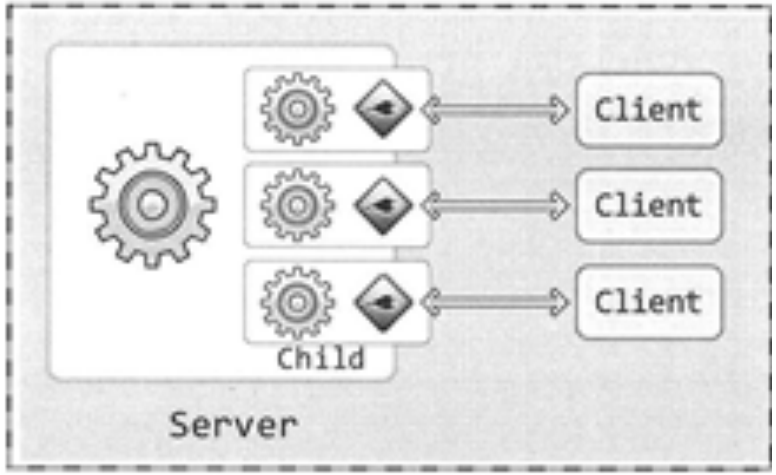
```
#include    "unp.h"

int
main(int argc, char **argv)
{
    int                 listenfd, connfd;
    pid_t             childpid;
    socklen_t          clilen;
    struct sockaddr_in cliaddr, servaddr;

    listenfd = Socket(AF_INET, SOCK_STREAM, 0);

    bzero(&servaddr, sizeof(servaddr));
    servaddr.sin_family      = AF_INET;
    servaddr.sin_addr.s_addr = htonl(INADDR_ANY);
    servaddr.sin_port        = htons(SERV_PORT);

    Bind(listenfd, (SA *) &servaddr, sizeof(servaddr));
    Listen(listenfd, LISTENQ);

    for ( ; ; ) {
        clilen = sizeof(cliaddr);
        connfd = Accept(listenfd, (SA *) &cliaddr, &clilen);
        str_echo(connfd);
        Close(connfd);
    }
}
```

Why non-concurrency happens?

- Process resource (CPU, especially program counter) occupied by blocking functions

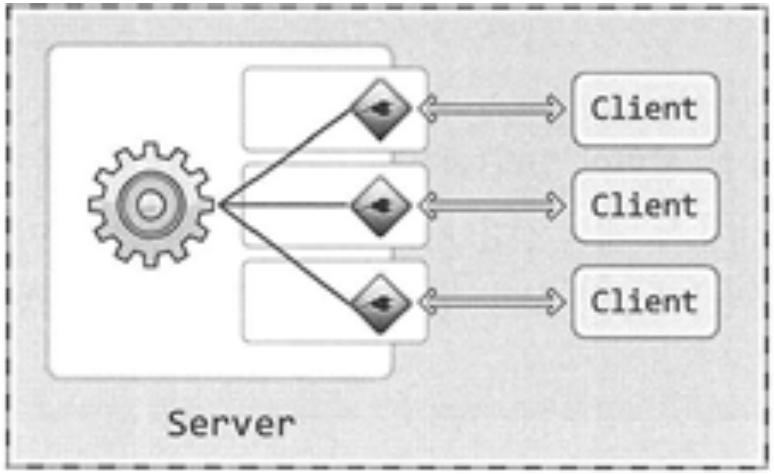
```
                    for ( ; ; ) {
                        clilen = sizeof(cliaddr);
No longer available ──→ connfd = Accept(listenfd, (SA *) &cliaddr, &clilen);
Stucked ──→ str_echo(connfd);
                        Close(connfd);
                    }
```

Two ways to overcome problem



Method 1. Multi Process(Thread)

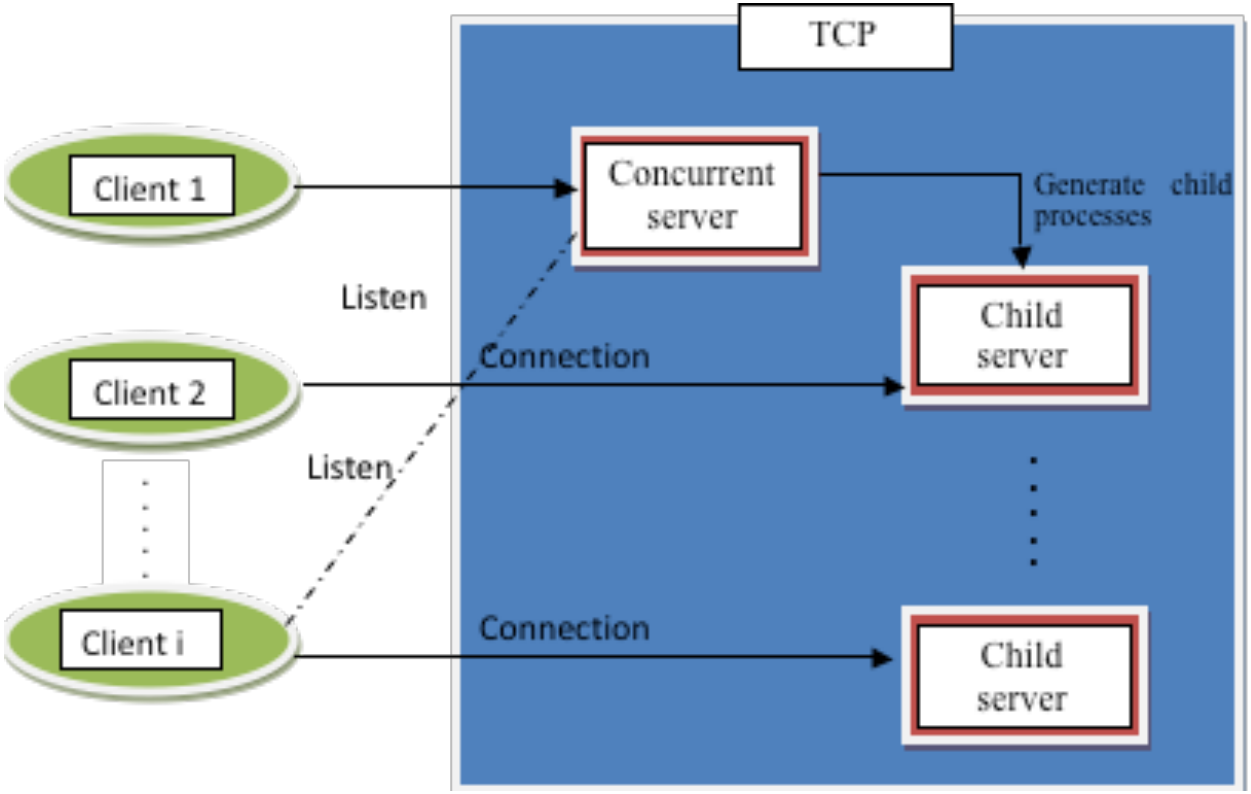
Create a processing objects (Process, Thread) that can run simultaneously



Method 2. I/O Multiplexing
- Select, Poll

The process manages the state of the socket and takes the appropriate action immediately when there is a socket event (new client, send/recv data)

Concept of Multi-process based concurrent server



Multi-process can be operated simultaneously by multicore CPU and CPU time sharing mechanism

Multi-process based Concurrent Server

- Download tcpserv01_non.c and move the file to /unp/tcpservcli/ directory
- Add make rule to ./Makefile

Line 3
```
PROGS = tcpcli01 tcpcli04 tcpcli05 tcpcli06 ₩
        tcpcli07 tcpcli08 tcpcli09 tcpcli10 ₩
        tcpserv01_non tcpserv01 …
```

Add to Line 39
```
tcpserv01_non:  tcpserv01_non.o
        ${CC} ${CFLAGS} -o $@ tcpserv01_non.o ${LIBS}
```

- Please modify for loop in tcpserv01_non.c to concurrent server with fork function and complete sig_chld

```
for ( ; ; ) {
   clilen = sizeof(cliaddr);
   connfd = Accept(listenfd, (SA *) &cliaddr, &clilen);

   if (/*create child process*/ ) {
      Close(listenfd);   /* close listening socket */
      str_echo(connfd);  /* process the request */
      exit(0);
   }
   Close(connfd);       /* parent closes connected socket */
}
```

```
void sig_chld(int) {
   /* complete code */
}
```

Select based Server Programming

- TA provide three different typed select based tcp server.
  - Donwload tcpservselect(02, 03, 04).c
  - Move the file into /unp/tcpservcli/ directory
  - Add make rule to Makefile

```
        PROGS = tcpcli01 tcpcli04 tcpcli05 tcpcli06 ₩
Line 3          tcpcli07 tcpcli08 tcpcli09 tcpcli10 ₩
                tcpserv01_non tcpserv01 tcpserv02 tcpserv03 tcpserv04 ₩
                tcpserv08 tcpserv09 tcpservselect01 tcpservselect02 tcpservselect03 tcpservselect04

        tcpservselect02:    tcpservselect02.o
                ${CC} ${CFLAGS} -o $@ tcpservselect02.o ${LIBS}
Add to
Line 63 tcpservselect03:   tcpservselect03.o
                ${CC} ${CFLAGS} -o $@ tcpservselect03.o ${LIBS}

        tcpservselect04:   tcpservselect04.o
                ${CC} ${CFLAGS} -o $@ tcpservselect04.o ${LIBS}
```

Select() Function Primitives and timeout argument

- This function allows the process to instruct the kernel
  - To wait for any one of multiple events to occur
  - To wake up the process only when one or more of these events occur or when a specified amount of time has passed
- select() system call function

```
#include <sys/select.h>

#include <sys/time.h>

int select(int maxfdp1, fd_set *readset, fd_set *writeset, fd_set
*exceptset, const struct timeval *timeout);

          Returns: positive count of ready descriptors, 0 on timeout, -1 on error
```

- Structure of timeval

```
struct timeval  {
  long   tv_sec;          /* seconds */
  long   tv_usec;         /* microseconds */
};
```

Select based Server Programming

- tcpservselect02.c
  - **Wait forever**: Return only when one of the specified descriptors is ready for I/O: timeout argument specify as a **null pointer**

```
nready = Select(maxfd+1, &rset, NULL, NULL, NULL);
```

- tcpservselect03.c
  - **Wait up to a fixed amount of time** – Return when one of the specified descriptors is ready for I/O

```
struct timeval timeout = {5, 0};
nready = Select(maxfd+1, &rset, NULL, NULL, &timeout);
```

- tcpservselect04.c
  - **Do not wait at all** – Return immediately after checking the descriptors (polling): **tv_sec and tv_usec all set to 0**

```
struct timeval timeout = {0, 0};
```

select() : wait forever (timeval is NULL)

❖ client connection

```
[miracle@localhost io_multiplexing]$ gcc -o selectServer1 selectServer1.c
[miracle@localhost io_multiplexing]$ ./selectServer1
listen fd : 3
Wait forever...
select positive count : 1 - FD_ISSET fd : 3
connect clientfd : 4
Wait forever...
select positive count : 1 - FD_ISSET fd : 3
connect clientfd : 5
Wait forever...
```

```
[miracle@localhost io_multiplexing]$ gcc -o selectClient1 selectClient1.c
[miracle@localhost io_multiplexing]$ ./selectClient1
```

```
[miracle@localhost io_multiplexing]$ ./selectClient1
```

- tcpservselect02.c / tcpcli01.c

select() : wait forever (timeval is NULL)

❖ client connection & read & write

```
[miracle@localhost io_multiplexing]$ ./selectServer1
listen fd : 3
Wait forever...
select positive count : 1 - FD_ISSET fd : 3
connect clientfd : 4
Wait forever...
select positive count : 1 - FD_ISSET fd : 4
read data : hi server.
Wait forever...
select positive count : 1 - FD_ISSET fd : 3
connect clientfd : 5
Wait forever...
select positive count : 1 - FD_ISSET fd : 5
read data : hi server.
Wait forever...
```

```
[miracle@localhost io_multiplexing]$ gcc -o selectClient2 selectClient2.c
[miracle@localhost io_multiplexing]$ ./selectClient2
write message : hi server.
read message : hi server.
```

```
[miracle@localhost io_multiplexing]$ ./selectClient2
write message : hi server.
read message : hi server.
```

- tcpservselect02.c / tcpcli01.c

select() : Wait up to a fixed amount of time (timeval is to set 5sec)

❖ client connection & read & write

```
[miracle@localhost io_multiplexing]$ gcc -o selectServer2 selectServer2.c
[miracle@localhost io_multiplexing]$ ./selectServer2
listen fd : 3
Wait 5 sec...
select positive count : 0 - select() timeout : 5 sec
Wait 5 sec...
select positive count : 0 - select() timeout : 5 sec
Wait 5 sec...
select positive count : 1 - FD_ISSET fd : 3
connect clientfd : 4
Wait 5 sec...
select positive count : 1 - FD_ISSET fd : 4
read data : hi server.
Wait 5 sec...
select positive count : 1 - FD_ISSET fd : 4
disconnect client
Wait 5 sec...
select positive count : 0 - select() timeout : 5 sec
Wait 5 sec...
select positive count : 0 - select() timeout : 5 sec
Wait 5 sec...
select positive count : 0 - select() timeout : 5 sec
Wait 5 sec...
select positive count : 0 - select() timeout : 5 sec
Wait 5 sec...
```

```
[miracle@localhost io_multiplexing]$ ./selectClient
write message : hi server.
read message : hi server.
[miracle@localhost io_multiplexing]$
```

```
[miracle@localhost io_multiplexing]$ ./selectClient
```

- tcpservselect03.c / tcpcli01.c

select() : Do not wait at all (timeval is 0)

❖ client connection & read & write

```
[miracle@localhost io_multiplexing]$ gcc -o selectServer3 selectServer3.c
[miracle@localhost io_multiplexing]$ ./selectServer3
listen fd : 3
Do not wait at all...
select positive count : 0
Do not wait at all...
select positive count : 0
Do not wait at all...
select positive count : 0
Do not wait at all...
select positive count : 0
Do not wait at all...
select positive count : 1 - FD_ISSET fd : 3
connect clientfd : 4
Do not wait at all...
select positive count : 1 - FD_ISSET fd : 4
read data : hi server.
Do not wait at all...
select positive count : 0
Do not wait at all...
select positive count : 1 - FD_ISSET fd : 4
disconnect client
Do not wait at all...
select positive count : 0
Do not wait at all...
```

```
[miracle@localhost io_multiplexing]$ ./selectClient
write message : hi server.
read message : hi server.
[miracle@localhost io_multiplexing]$ ^C
[miracle@localhost io_multiplexing]$
```

```
[miracle@localhost io_multiplexing]$ ./selectClient
```

- tcpservselect04.c / tcpcli01.c

Poll based Server Programming

- TA provide three different typed poll based tcp server.
  - Donwload tcpservpoll(02, 03, 04).c
  - Move the file into /unp/tcpservcli/ directory
  - Add make rule to Makefile

Line 3

```
PROGS = tcpcli01 tcpcli04 tcpcli05 tcpcli06 ₩
        tcpcli07 tcpcli08 tcpcli09 tcpcli10 ₩
        tcpserv01_non tcpserv01 tcpserv02 tcpserv03 tcpserv04 ₩
        tcpserv08 tcpserv09 tcpservselect01 tcpservselect02 tcpservselect03 tcpservselect04 ₩
        tcpservpoll01 tcpservpoll02 tcpservpoll03 tcpservpoll04 tsigpipe
```

Add to Line 75

```
tcpservpoll02:  tcpservpoll02.o
        ${CC} ${CFLAGS} -o $@ tcpservpoll02.o ${LIBS}

tcpservpoll03: tcpservpoll03.o
        ${CC} ${CFLAGS} -o $@ tcpservpoll03.o ${LIBS}

tcpservpoll04: tcpservpoll04.o
        ${CC} ${CFLAGS} -o $@ tcpservpoll04.o ${LIBS}
```

Poll() Function Primitives and timeout argument

- **poll**() performs a similar task to select(2): it waits for one of a set of file descriptors to become ready to perform I/O.
- poll() system call function

```
int  poll(struct pollfd *fdarray, unsigne long nfds, int timeout);
```

timeout value	Description
INFTIM	Wait forever
0	Return immediately, do not block
> 0	Wait specified number of milliseconds

- Structure of pollfd

```
struct pollfd {
    int     fd;      /* descriptor to check */
    short   events;  /* events of interest on fd */
    short   revents; /* events that occurred on fd */
};
```

Available Events is described in table

Constant	Input to *events* ?	Result from *revents* ?	Description
POLLIN	•	•	Normal or priority band data can be read
POLLRDNORM	•	•	Normal data can be read
POLLRDBAND	•	•	Priority band data can be read
POLLPRI	•	•	High-priority data can be read
POLLOUT	•	•	Normal data can be written
POLLWRNORM	•	•	Normal data can be written
POLLWRBAND	•	•	Priority band data can be written
POLLERR		•	Error has occurred
POLLHUP		•	Hangup has occurred
POLLNVAL		•	Descriptor is not an open file

Poll based Server Programming

- tcpservpoll02.c
  - **Wait forever**: Return only when one of the specified descriptors is ready for I/O: timeout argument specify as a **INFTIM (-1)**

```
nready = Poll(client, maxi+1, INFTIM);
```

- tcpservpoll03.c
  - **Do wait at all** – Return immediately after checking the descriptors

```
nready = Poll(client, maxi+1, 0);
```

- tcpservpoll04.c
  - **Wait up to a fixed amount of time** – Return when one of the specified descriptors is ready for I/O

```
nready = Poll(client, maxi+1, 1000);
```

poll() : wait forever (timeout value : INFTIM, -1)

❖ client connection & read & write

```
[miracle@localhost io_multiplexing]$ gcc -o pollServer1 pollServer1.c
[miracle@localhost io_multiplexing]$ ./pollServer1
listen fd : 3
Wait forever...
poll positive count : 1 - connect clientfd : 4
Wait forever...
poll positive count : 1 - read data [clientfd = 4] : hi server.
Wait forever...
poll positive count : 1 - close clientfd : 4
Wait forever...
```

```
[miracle@localhost io_multiplexing]$ ./pollClient
write message : hi server.
read message : hi server.
[miracle@localhost io_multiplexing]$
```

```
[miracle@localhost io_multiplexing]$ ./selectClient
```

- tcpservpoll02.c / tcpcli01.c

poll() : Return immediately, do not block

(timeout value : 0)

❖ client connection & read & write

```
[miracle@localhost io_multiplexing]$ gcc -o pollServer2 pollServer2.c
[miracle@localhost io_multiplexing]$ ./pollServer2
listen fd : 3
Do not wait at all...
poll positive count : 0
Do not wait at all...
poll positive count : 0
Do not wait at all...
poll positive count : 0
Do not wait at all...
poll positive count : 0
Do not wait at all...
poll positive count : 0
Do not wait at all...
poll positive count : 1 - connect clientfd : 4
Do not wait at all...
poll positive count : 1 - read data [clientfd = 4] : hi server.
Do not wait at all...
poll positive count : 0
Do not wait at all...
poll positive count : 1 - close clientfd : 4
Do not wait at all...
poll positive count : 0
Do not wait at all...
poll positive count : 0
```

```
read message : hi server.
[miracle@localhost io_multiplexing]$ ./pollClient
write message : hi server.
read message : hi server.
[miracle@localhost io_multiplexing]$
```

```
[miracle@localhost io_multiplexing]$ ./selectClient
```

- tcpservpoll03.c / tcpcli01.c

poll() : Wait specified number of milliseconds

(timeout value > 0)

❖ client connection & read & write

```
[miracle@localhost io_multiplexing]$ gcc -o pollServer3 pollServer3.c
[miracle@localhost io_multiplexing]$ ./pollServer3
listen fd : 3
Wait 1000 millisec...
poll positive count : 0 - timeout : 1000 millisec
Wait 1000 millisec...
poll positive count : 0 - timeout : 1000 millisec
Wait 1000 millisec...
poll positive count : 0 - timeout : 1000 millisec
Wait 1000 millisec...
poll positive count : 0 - timeout : 1000 millisec
Wait 1000 millisec...
poll positive count : 0 - timeout : 1000 millisec
Wait 1000 millisec...
poll positive count : 0 - timeout : 1000 millisec
Wait 1000 millisec...
poll positive count : 0 - timeout : 1000 millisec
Wait 1000 millisec...
poll positive count : 1 - connect clientfd : 4
Wait 1000 millisec...
poll positive count : 1 - read data [clientfd = 4] : hi server.
Wait 1000 millisec...
poll positive count : 1 - close clientfd : 4
Wait 1000 millisec...
poll positive count : 0 - timeout : 1000 millisec
```

```
[miracle@localhost io_multiplexing]$ ./pollClient
write message : hi server.
read message : hi server.
[miracle@localhost io_multiplexing]$
```

```
[miracle@localhost io_multiplexing]$ ./selectClient
```

- tcpservpoll04.c / tcpcli01.c

How can we build file transfer server with I/O Multiplexing?

- There might needs some context manager and data structure
- Code will be very complex!
- How about write APIs that functions frequently used in a standardized form?
  - In the following lectures, we will learn about event-driven web server framework.