Contents Delivery Programming

2009년 2학기

숙명여자대학교 정보과학부
멀티미디어과학전공

윤용익
yiyoon@sookmyung.ac.kr

강의 목차

1주 : 강의 소개 및 Part 1- Contents Delivery Programming 개요

2주 ~ 4주 : Part 2- Internal IPC Programming (Unix 위주: C)

5주 ~ 7주 : Part 3 - Socket Programming (Unix 위주: C)

8주 : 중간 프로젝트 발표

9주 ~ 10주: Part 4 - Socket Programming (Java)

11주 : Part 5 - RPC Programming (Unix 위주)

12주 ~ 14주 : Part 6 - RMI Programming (Java)

15 주 : 기말고사 (Term 프로젝트 발표)

12월 8일 : Term 프로젝트 발표

12월 10일 : 기말고사

강의 교재

- 주 및 부교재
  - W. R. Stevens, Unix Network Programming, Prentice-Hall, 1999년 이전판
    - Unix IPC (Ch 3), RPC (Ch 18), Socket programming (Ch 6)
  - W. R. Stevens, Unix Network Programming Vol. 1, Prentice-Hall, 1999.
    - Socket Programming (Part 2 And Part 3)
  - W. R. Stevens, Unix Network Programming Vol. 2, Prentice-Hall, 1999.
    - Unix IPC and RPC programming
  - 최재영 외 2인, 프로그래머를 위한 Java 2, 홍릉과학출판사
    - Java Socket (10장), RMI(12장)

Part 1

Overview of Contents Delivery Programming

1. 본 과목의 이해

- Contents Delivery System?
  - Contents Delivery Network ?
  - Client / Server Model?

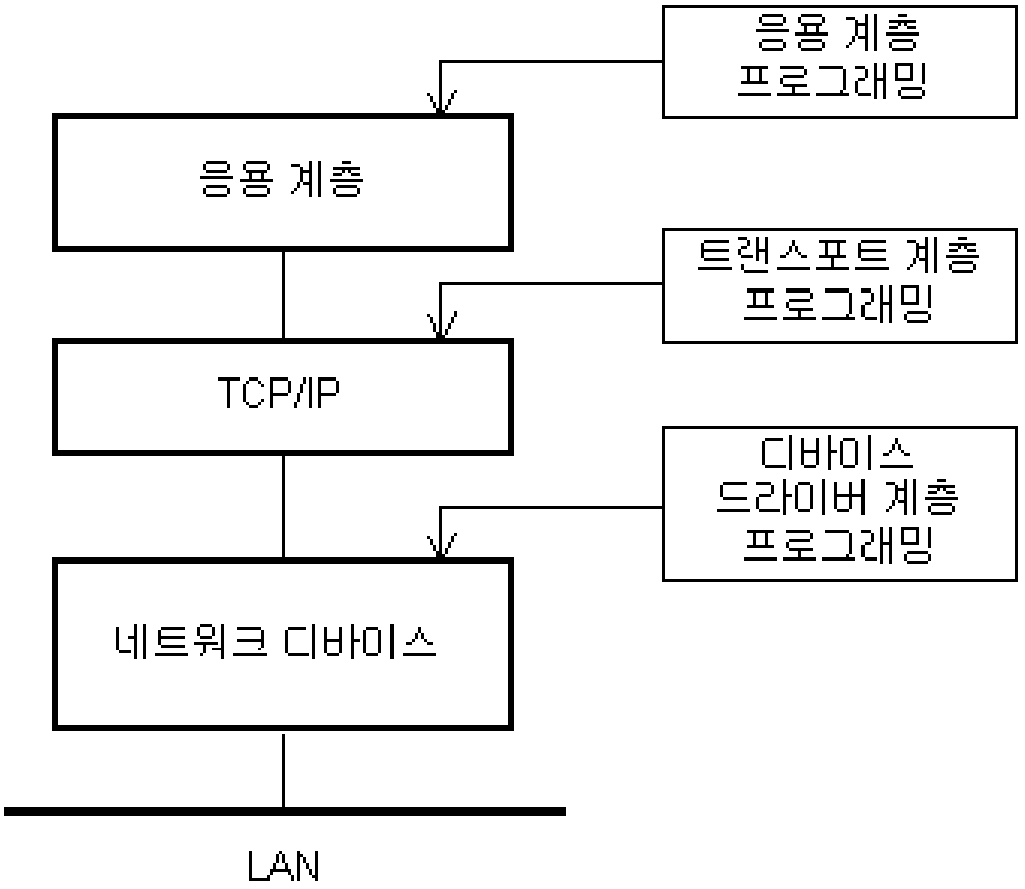
- Information Infrastructure
  - Internet ?
    - TCP/IP

- Network Programming ?
  - TCP/IP Reference Model

→ **Contents Delivery Programming?**

1.1 TCP/IP Reference Model

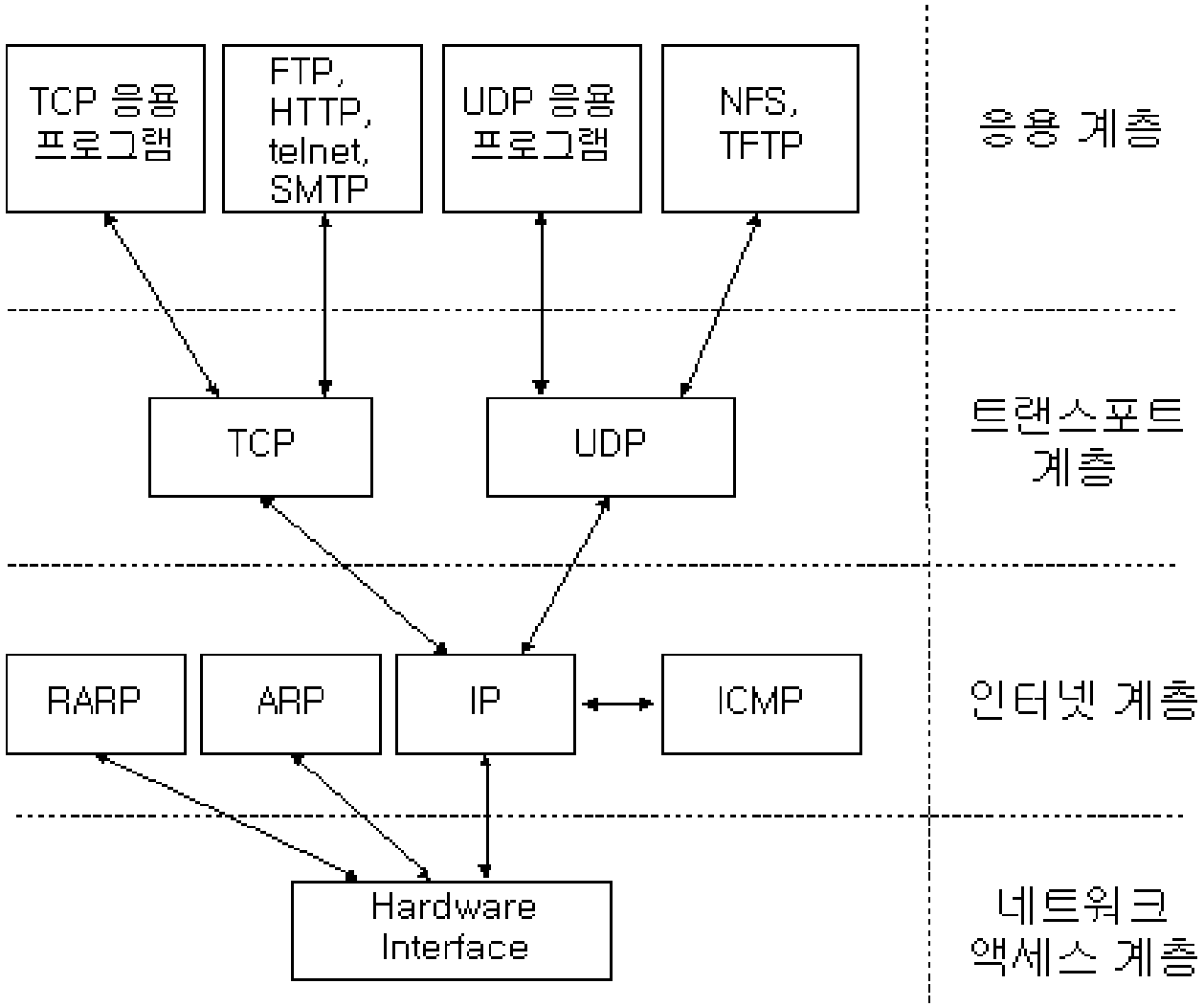
- 네트워크 프로그래밍의 계층별 분류



1.1.1 계층별 분류와 특징

계층	예	특 징
응용 계층	http CORBA RPC	•이미 작성된 유틸리티 활용 •개발, 변경, 운영의 편리
트랜스포트 계층	Socket Winsock	•패킷 단위의 송수신 처리 •인터넷 프로그램의 기초
디바이스 드라이버 계층	Packet Driver NDIS, ODI	•MAC 프레임 단위의 송수신 •흐름제어, 오류제어가 없음

1.1.2 TCP/IP 내부 구조

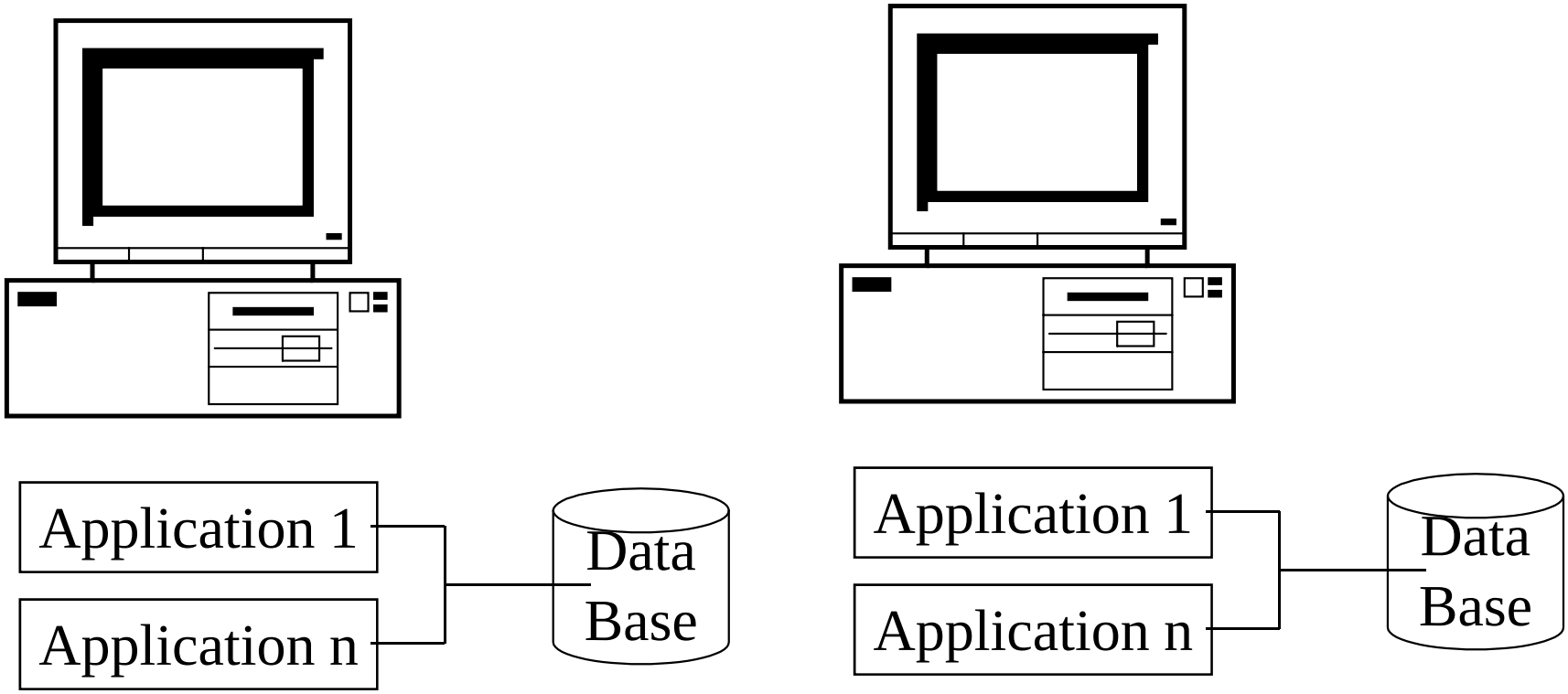


1.2 클라이언트-서버 통신 모델

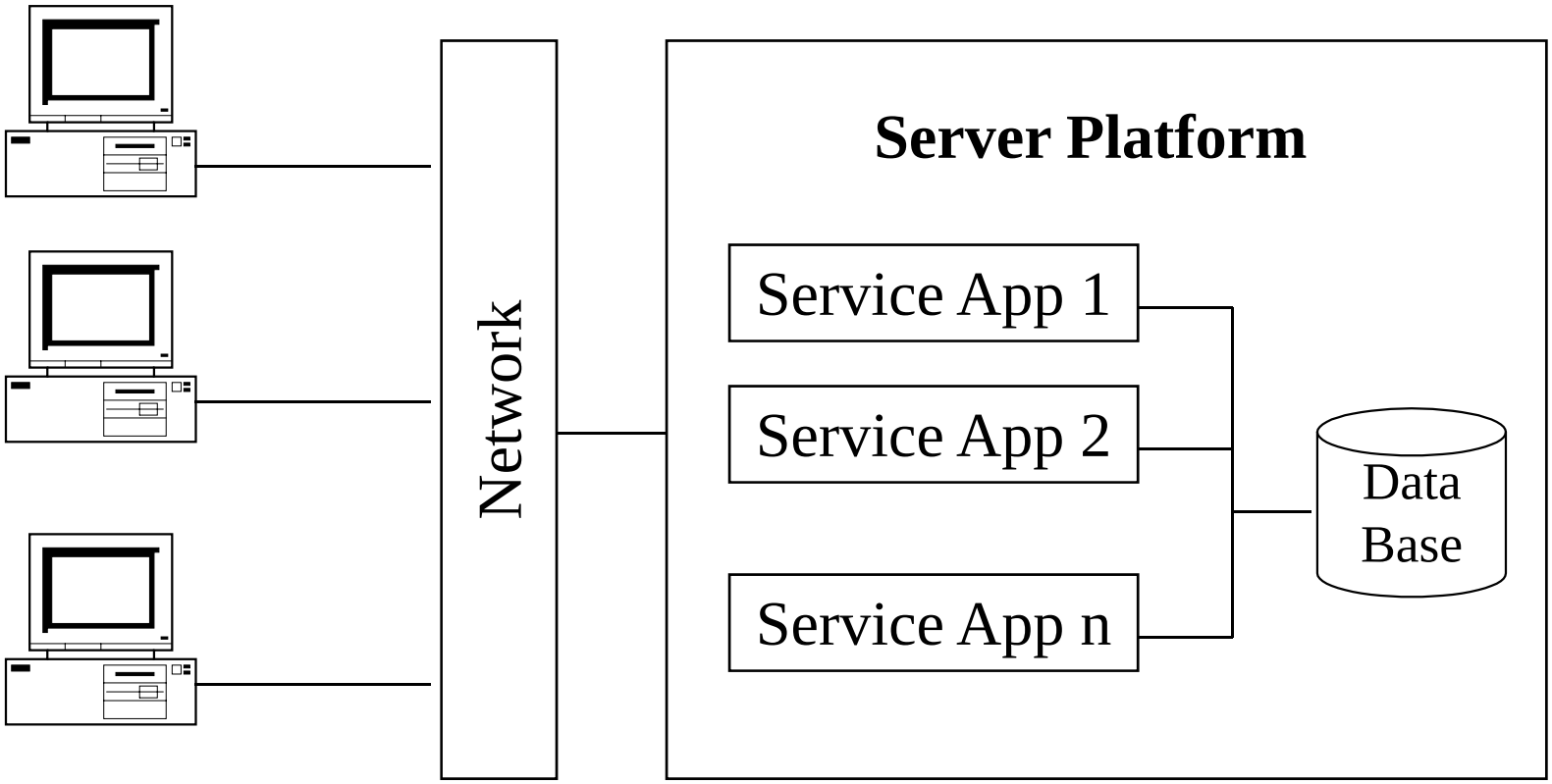
- 서버 : 서비스 제공
- 클라이언트 : 서버가 제공하는 서비스를 이용하는 장비와 이에 필요한 프로그램

- 서버가 먼저 실행
- 클라이언트가 요구(request)를 보내면 서버가 이에 대한 응답(response)를 보내준다

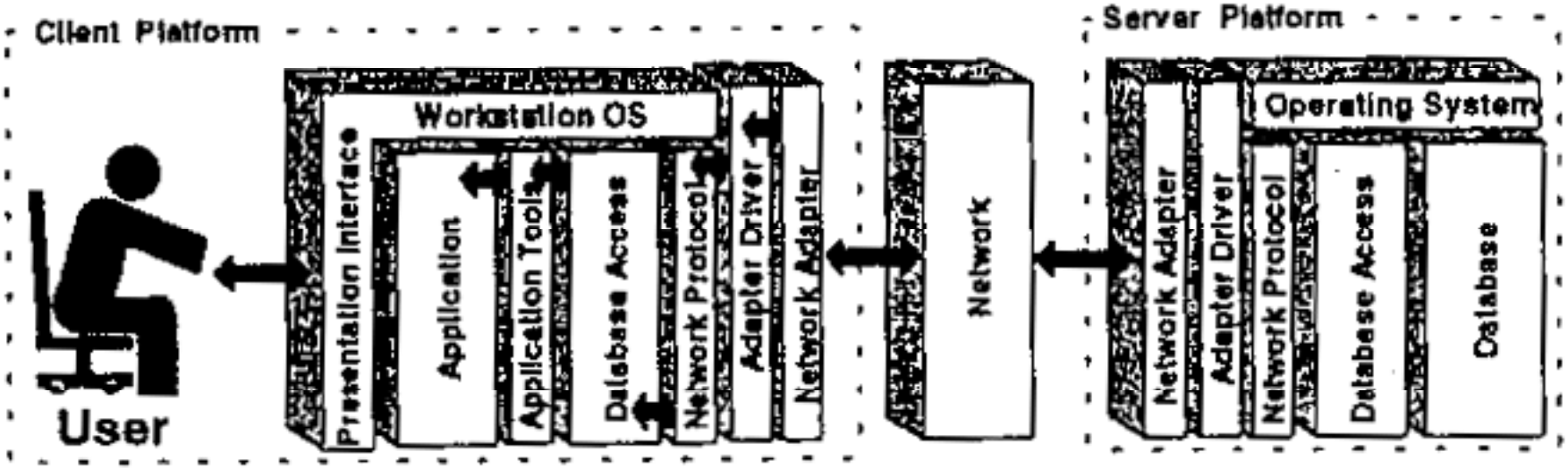
Isolated Single-user Architecture



Client/Server Architecture (1)

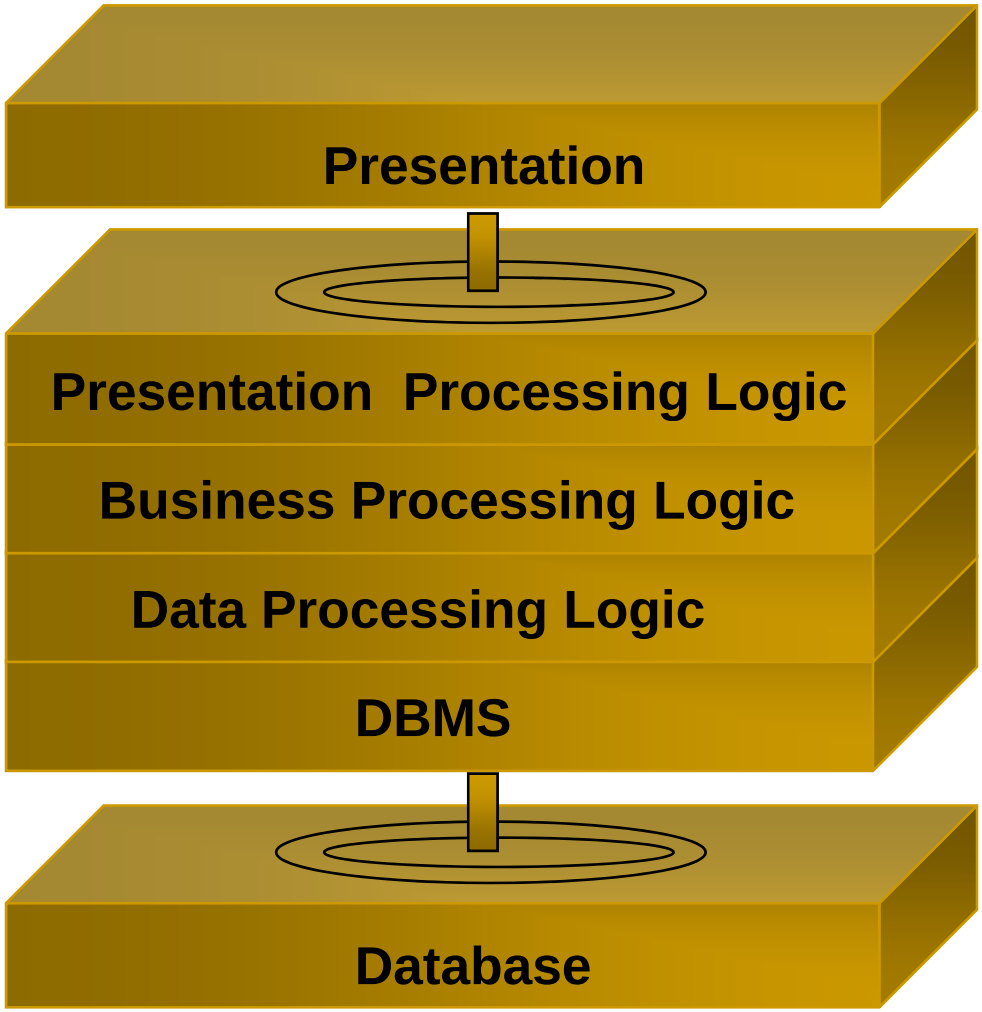


Client/Server Architecture (2)



Spectrum of Client/Server Definitions

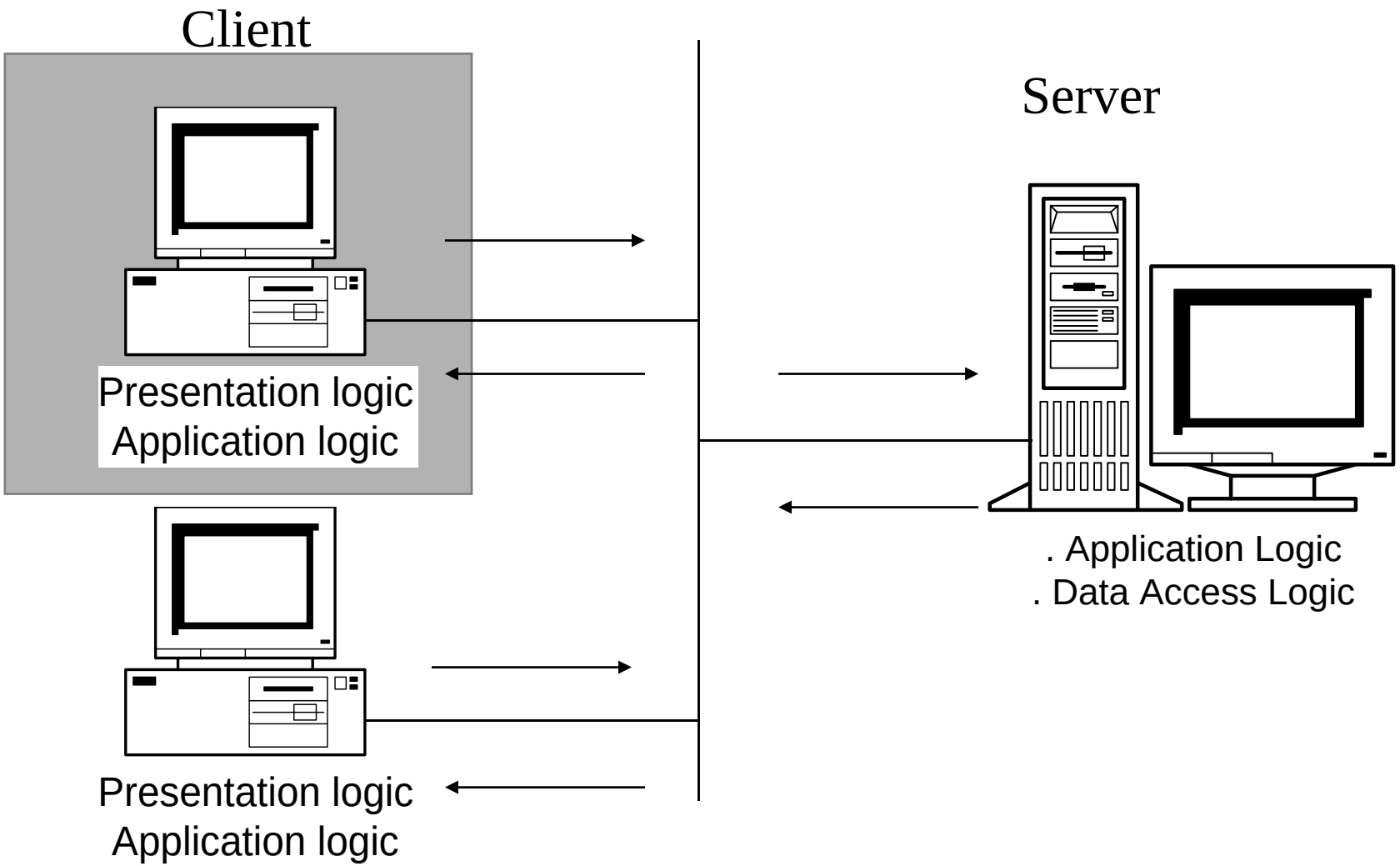
- Presentation
- Application
- Data management
- Network



Application Tasks in C/S

- **User Interface**
  - what the user actually sees
- **Presentation logic**
  - what happens when the user interacts with the form on the screen
- **Application logic**
- **Data requests and results acceptance**
- **Data integrity**
  - such as validation, security, completeness
- **Physical data management**
  - such as update m retrieval, deletion, and addition

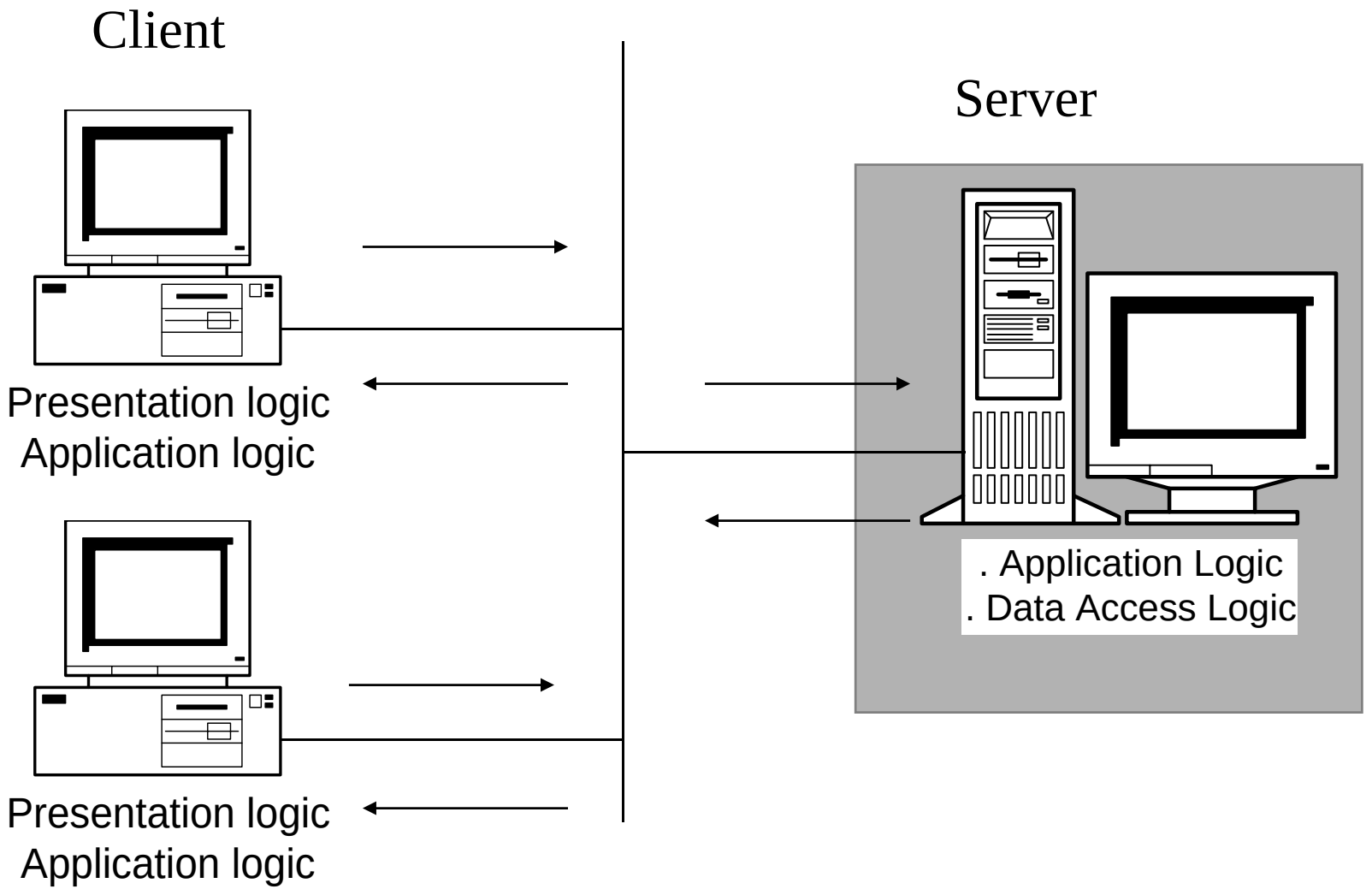
Client/Server Components: Role of Client (1)



Client/Server Components: Role of Client (2)

- **Definition:** The client is any application/function/launcher/requester process
- **The client extends the user's workplace**
  - Access to remote functions and data
  - Communication with other peer processes
  - Resource sharing as required
- **The client allows the user to personalize the workplace**
  - Custom interface design
  - Hardware/software "rightsized" to requirements
  - Sense of ownership
- **The client hides complexity of underlying infrastructure**
  - Network, operating system, and data access are hidden
  - User has a common syntax and lexicon across all application s
- **A client may also be a server concurrently**

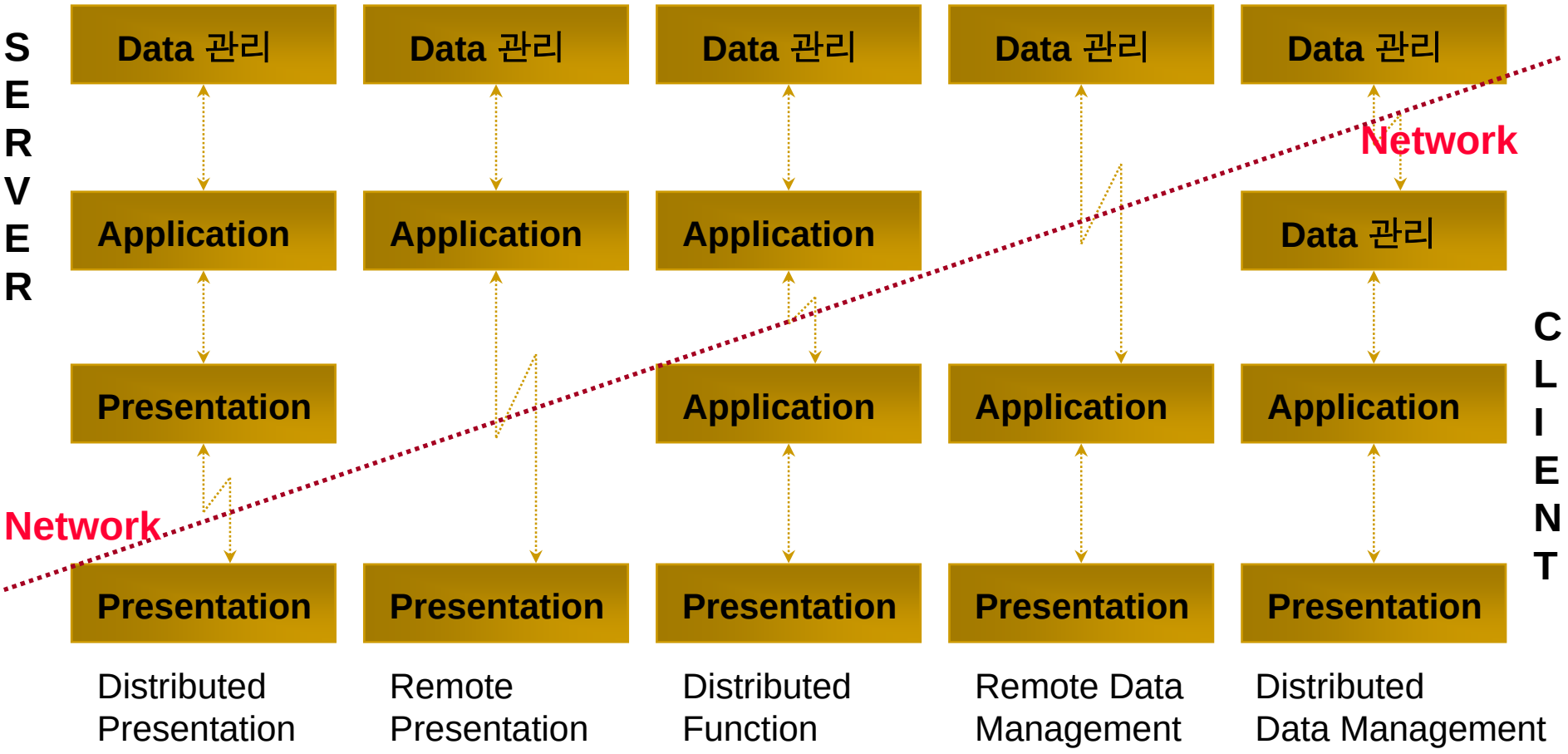
Client/Server Components: Role of Server (1)



Client/Server Components: Role of Server (2)

- **Definition:** The server is a specialized responder to requests
  - That is, the server is a process
- **The server implements specific functions**
  - Access to data, services, custom applications
  - May provide some aspects of presentation
- **Servers can be arranged in a variety of configurations**
  - Tiered, star, network
  - New servers can be implemented with minimal impact
  - Multiple servers may exist on the same hardware platform
- **Servers may exist on a variety of hardware/software platforms**
  - Not restricted to microcomputers
- **A server may also be a client concurrently**

Client/Server Types



Part 2

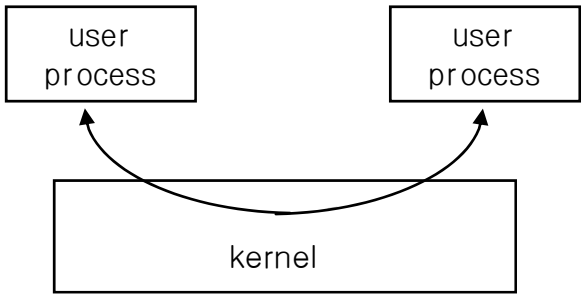
Interprocess Communication (Internal IPC in UNIX)

Concurrent Processing Environment

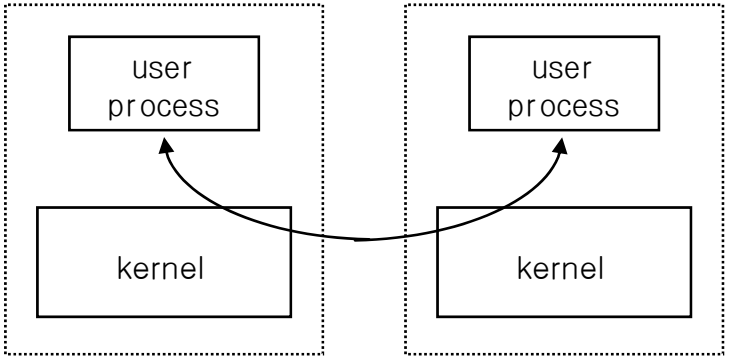
- Objective of IPC
  - Synchronization between processes
    - Semaphore: allowable waiting queue
    - wait and wakeup function: P & V operation
  - Mutual Exclusion between processes
    - Monitor
    - Region : critical section <-- semaphore
  - Communication between processes
    - Buffer : Mailbox, message queue
    - Signal : Socket

Interprocess Communication의 개념

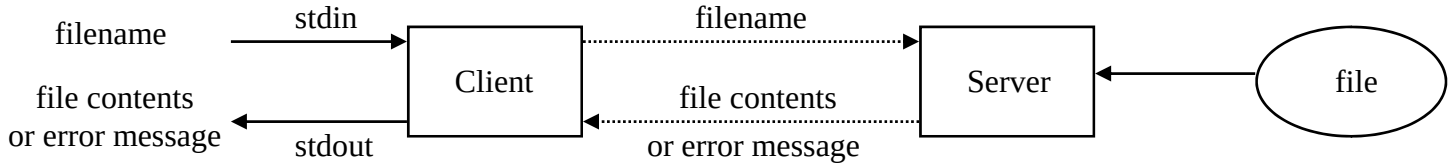
☐ 단일 컴퓨터 시스템상에서의
두 프로세스간 IPC



☐ 서로 다른 컴퓨터 시스템상에서의
두 프로세스간 IPC



☐ Client-Server Model의 예



Unix Internal IPC

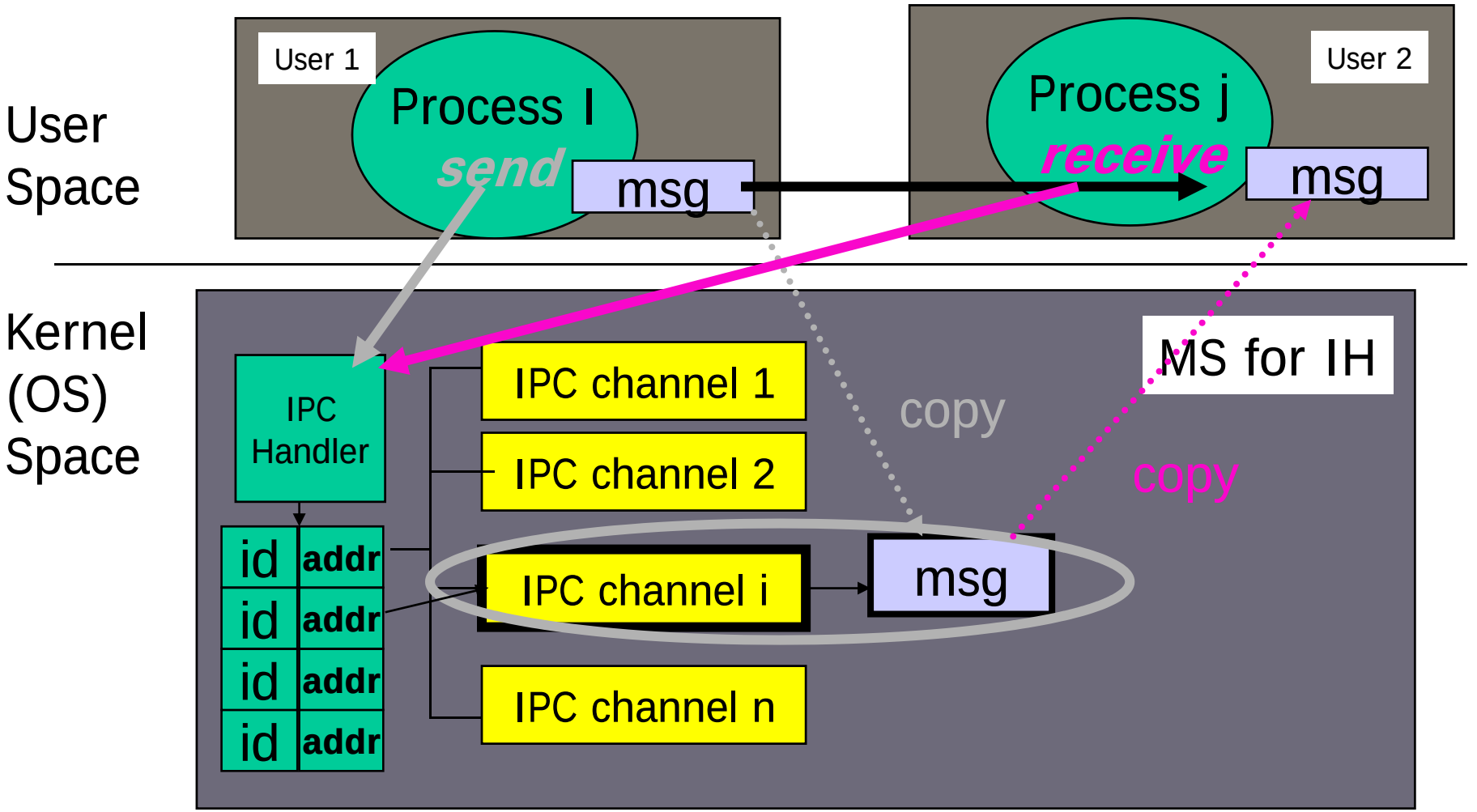
☐ Summary of Unix IPC system calls

	Message queue	Semaphore	Shared memory
include file	<sys/msg.h>	<sys/sem.h>	<sys/shm.h>
system call to create or open	msgget	semget	shmget
system call for control operations	msgctl	semctl	shmctl
system calls for IPC operations	msgsnd	semop	shmat
	msgrcv		shmdt

☐ kernel은 각 IPC channel에 대해서도 file을 다루기 위한 정보와 비슷한 정보의 구조를 갖음

- <sys/ipc.h>에 정의
- 이 구조를 사용하고 변경하기 위해서는 세가지의 ctl system call을 사용
- 하나의 IPC channel을 만들거나 열기위해 사용

```
struct ipc_perm {
   ushort uid;      /*owner's user id */
   ushort gid;      /* owner's group id */
   ushort cuid;     /* creator's user id */
   ushort cgid;     /* creator's group id */
   ushort mode;     /* access modes */
   ushort seq;      /* slot usage sequence number */
   key_t  key;      /* key */
};
```



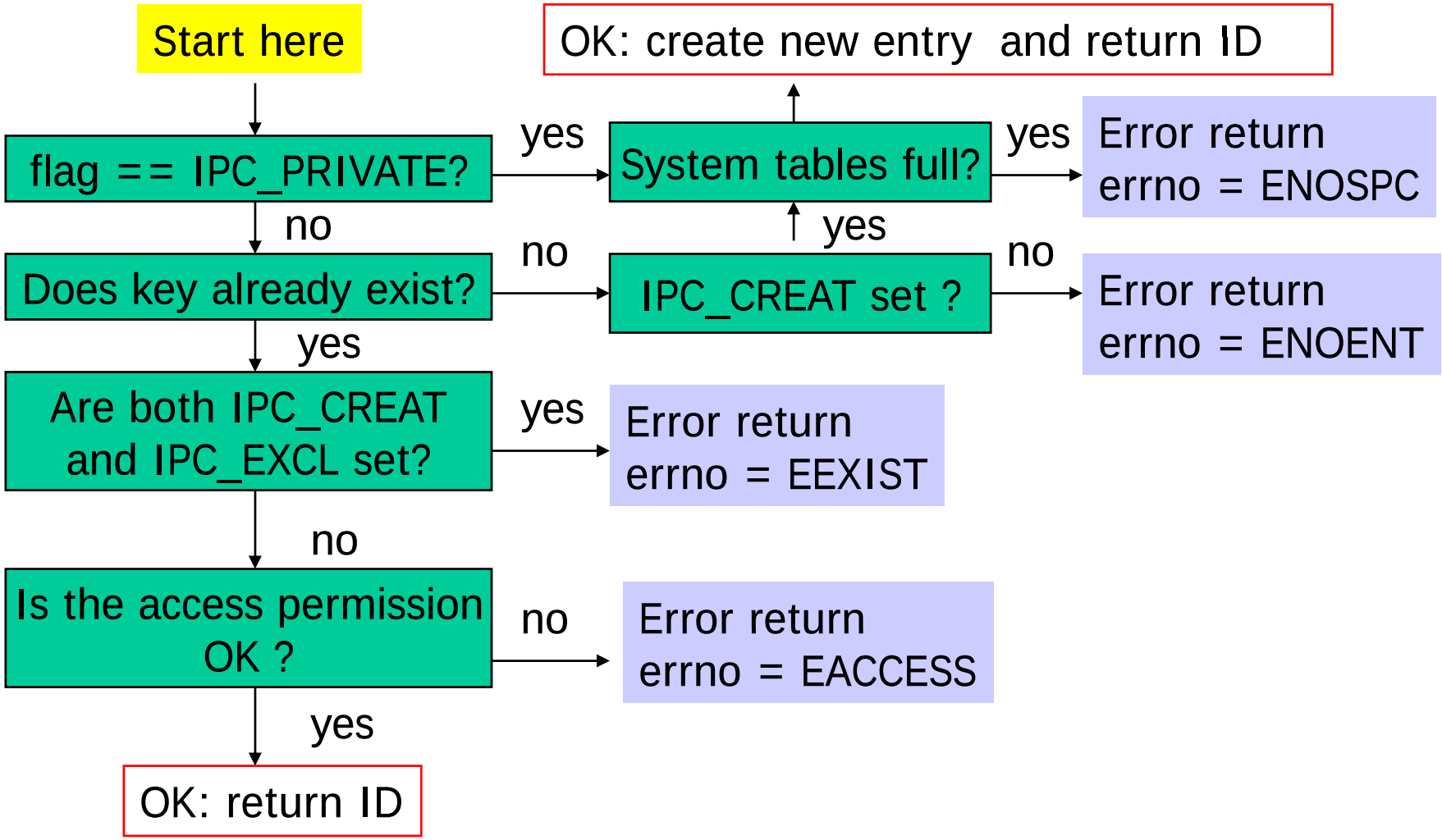
IPC channel : means of MQ, SM, Sema, Port

- Three get system calls
  - take a key value
    - return integer key value

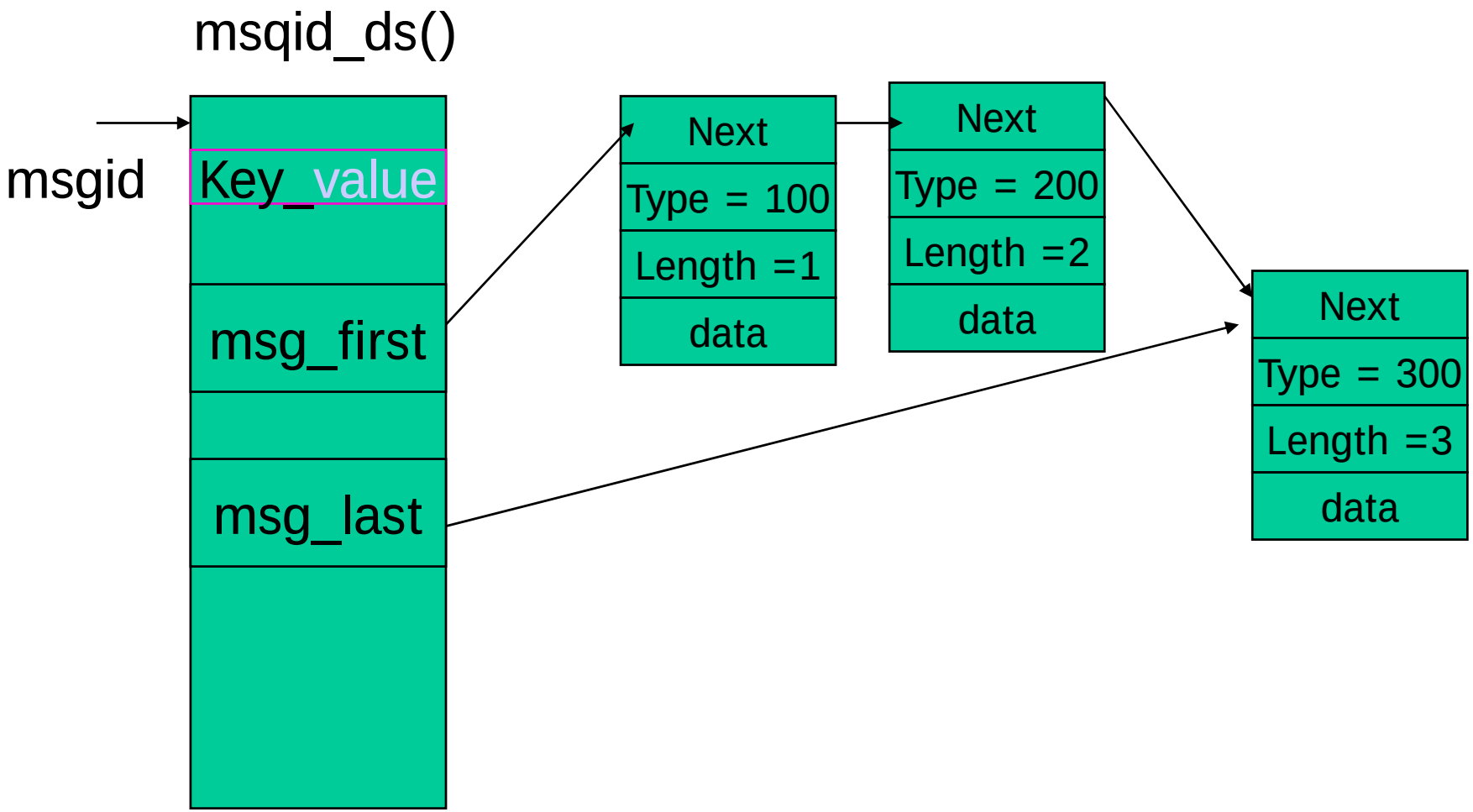
  - msgget
  - semget
  - shmget

  - IPC_CREAT
  - IPC_EXCL
  - IPC_PRIVATE
  - IPC_PERM

Logic for opening or creating an IPC channel



Message queue structure in kernel



Message queues(1)

□ system내의 각 message queue에 대해서 kernel은 다음과 같은 정보 구조를 유지

```
#include <sys/types.h>
#include <sys/ipc.h>          /* defines the ipc_perm structure */

struct msqid_ds {
   struct ipc_perm msg_perm;   /* operation permission struct */
   struct msg      *msg_first; /* ptr to first message on q */
   struct msg      *msg_last;  /* ptr to last message on q */
   ushort          msg_cbytes; /* current # bytes on q */
   ushort          msg_qnum;   /* current # of messages on q */
   ushort          msg_qbytes; /* max # of bytes allowed on q */
   ushort          msg_lspid;  /* pid of last msgsnd */
   ushort          msg_lrpid;  /* pid of last msg rcv */
   time_t          msg_stime;  /* time of last msgsnd */
   time_t          msg_rtime;  /* time of last msgrcv */
   time_t          msg_ctime;  /* time of last msgctl (that changed the above) */
};
```

Message queues(2)

☐ msgget system call
- 새로운 message queue을 만들거나 기존의 message queue에 접근

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>

int msgget(key_t key, int msgflag);
```

- msgflag : 다음 상수들의 조합

Numeric	Symbolic	Description
0400	MSG_R	Read by owner
0200	MSG_W	Write by owner
0040	MSG_R >> 3	Read by group
0020	MSG_W >> 3	Write by group
0004	MSG_R >> 6	Read by world
0002	MSG_W >> 6	Write by world
	IPC_CREAT	
	IPC_EXCL	

Message queues(3)

☐ msgsnd system call
- message queue에 message 하나를 넣음

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>

int msgsnd(int msqid, struct msgbuf *ptr, int length, int flag);
```

- *ptr* :

```
struct msgbuf {
   long mtype;          /* message type, must be > 0 */
   char mtext[1];       /* message data */
};
```

- *length* : message의 길이(byte단위)
- *flag* : IPC_NOWAIT나 0으로 지정
  * IPC_NOWAIT : message queue에 새로운 message를 위한 여유 공간이 없을 경우에 즉시 system call에서 돌아오도록 함

Message queues(4)

☐ msgrcv system call
- message queue에서 message를 읽음

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>

int msgrcv(int msqid, struct msgbuf *ptr, int length, long msgtype, int flag);
```

- *ptr* : msgsnd와에서와 비슷하며 받은 message를 저장할 장소를 명시
- *length* : ptr이 지시하는 구조의 data부분의 크기를 명시
- msgtype : queue로부터 어떤 message를 요구하는지를 명시
  - * msgtype = 0, queue의 첫번째 message를 받음
  - * msgtype > 0, msgtype과 동일한 type을 갖는 첫번째 message를 받음
  - * msgtype < 0, msgtype의 절대치보다 같거나 작은 type중에서 가장 작은 type을 갖는 첫번째 message를 받음
- *flag* : 요구된 type의 message가 queue에 없을 경우에 어떻게 할 것인지를 명시
  - * MSG_NOERROR 설정시, 받은 message의 data 부분이 length보다 크다면 즉시 data 부분을 자르고 error없이 복귀함

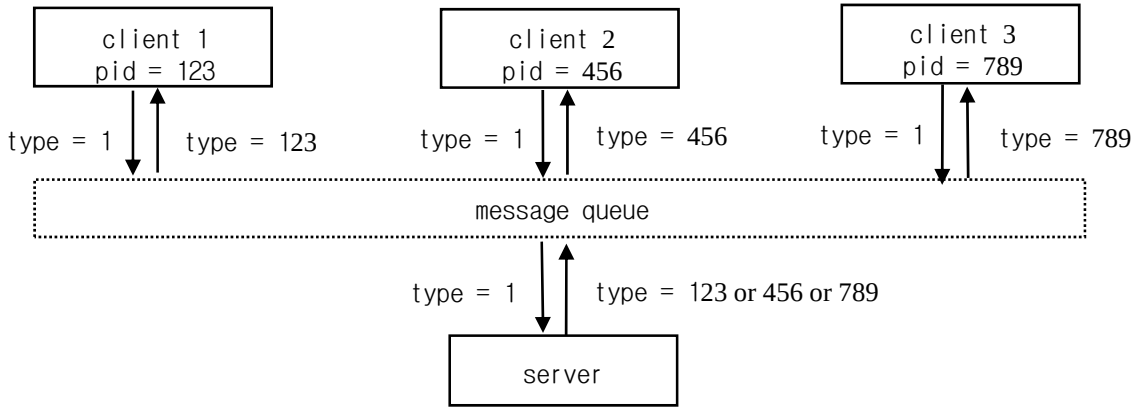
Message queues(5)

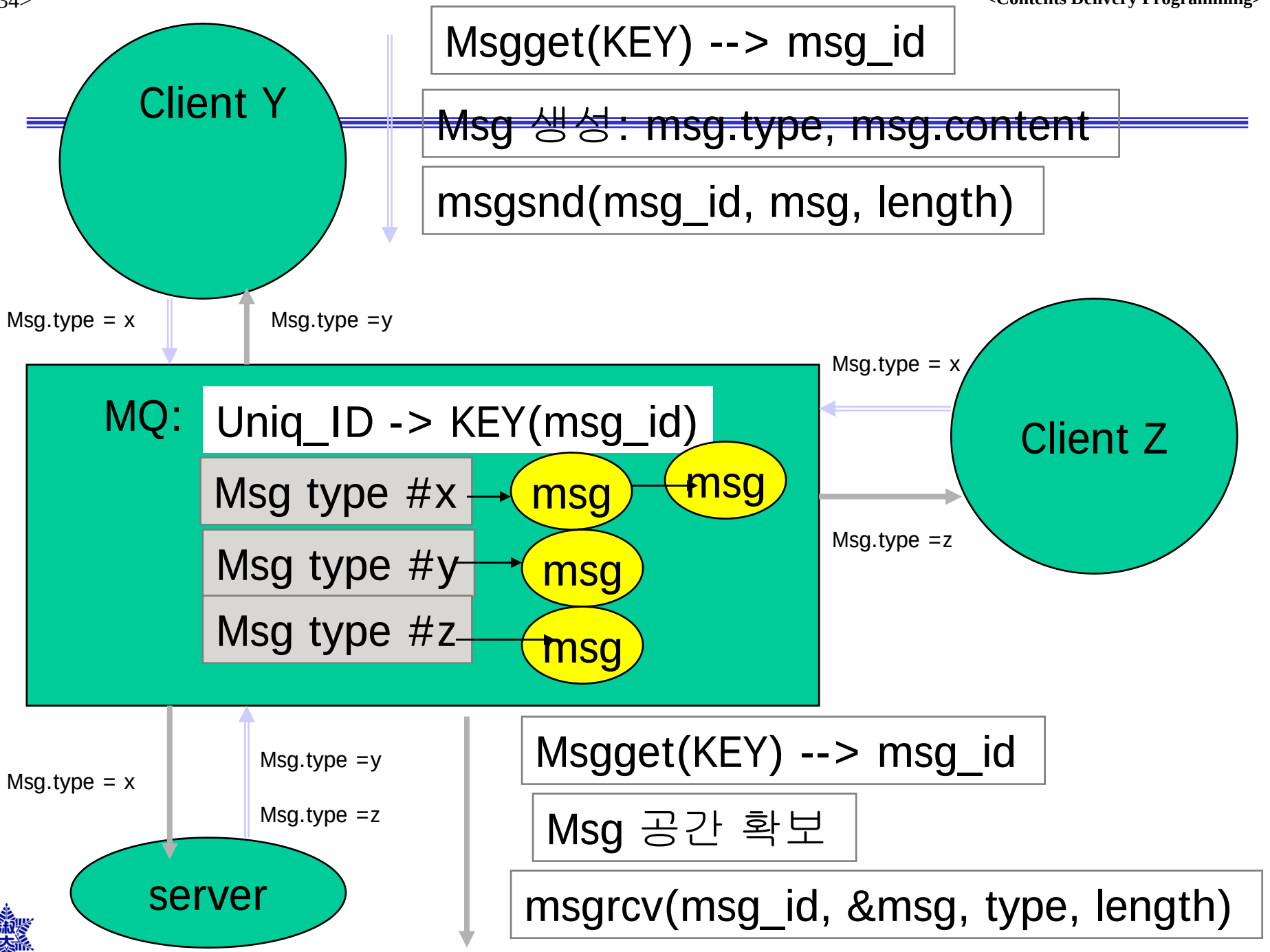
☐ msgctl system call
- message queue에 대한 다양한 제어 기능을 제공

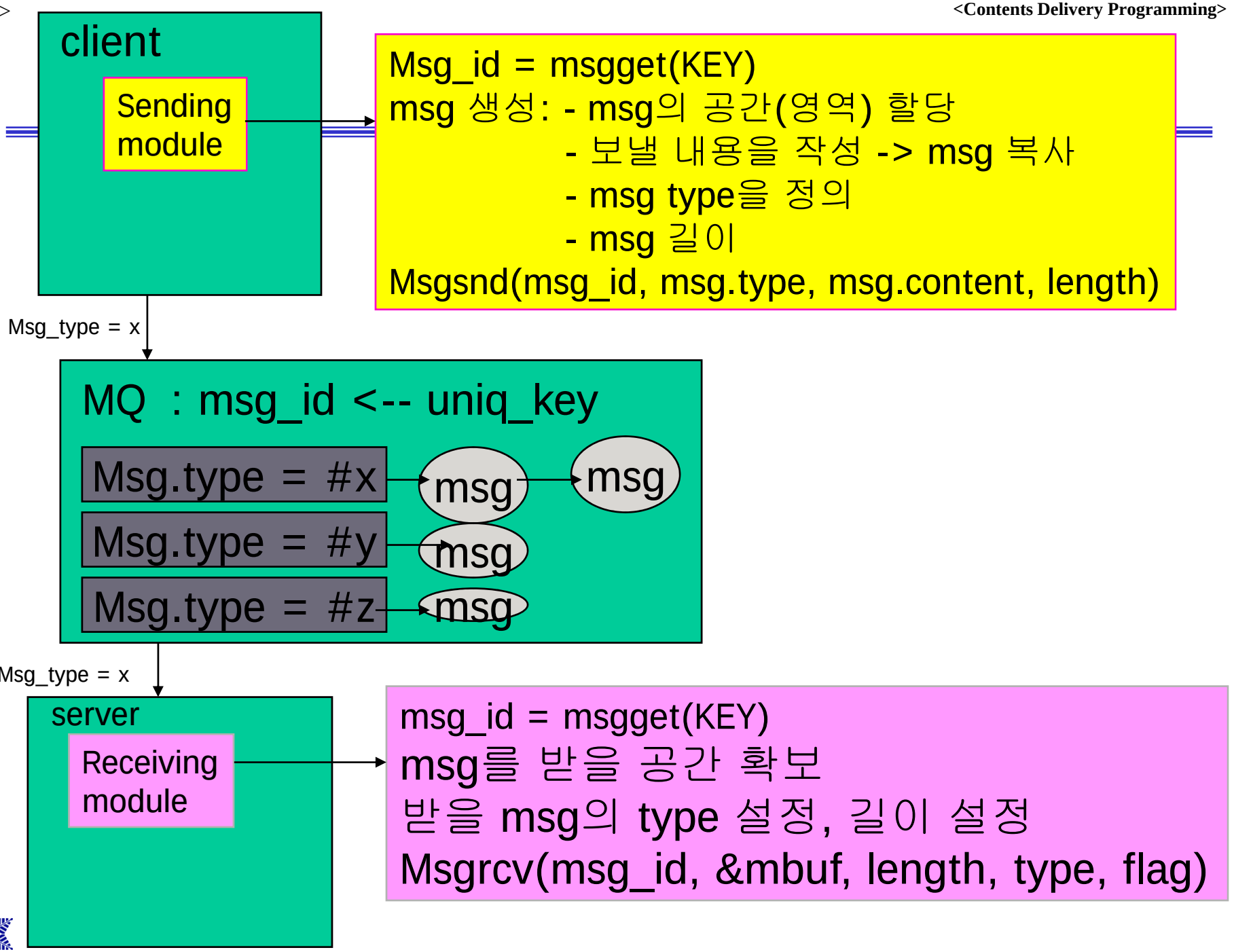
```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>

int msgctl(int msqid, int cmd, struct msqid_ds *buff);
```

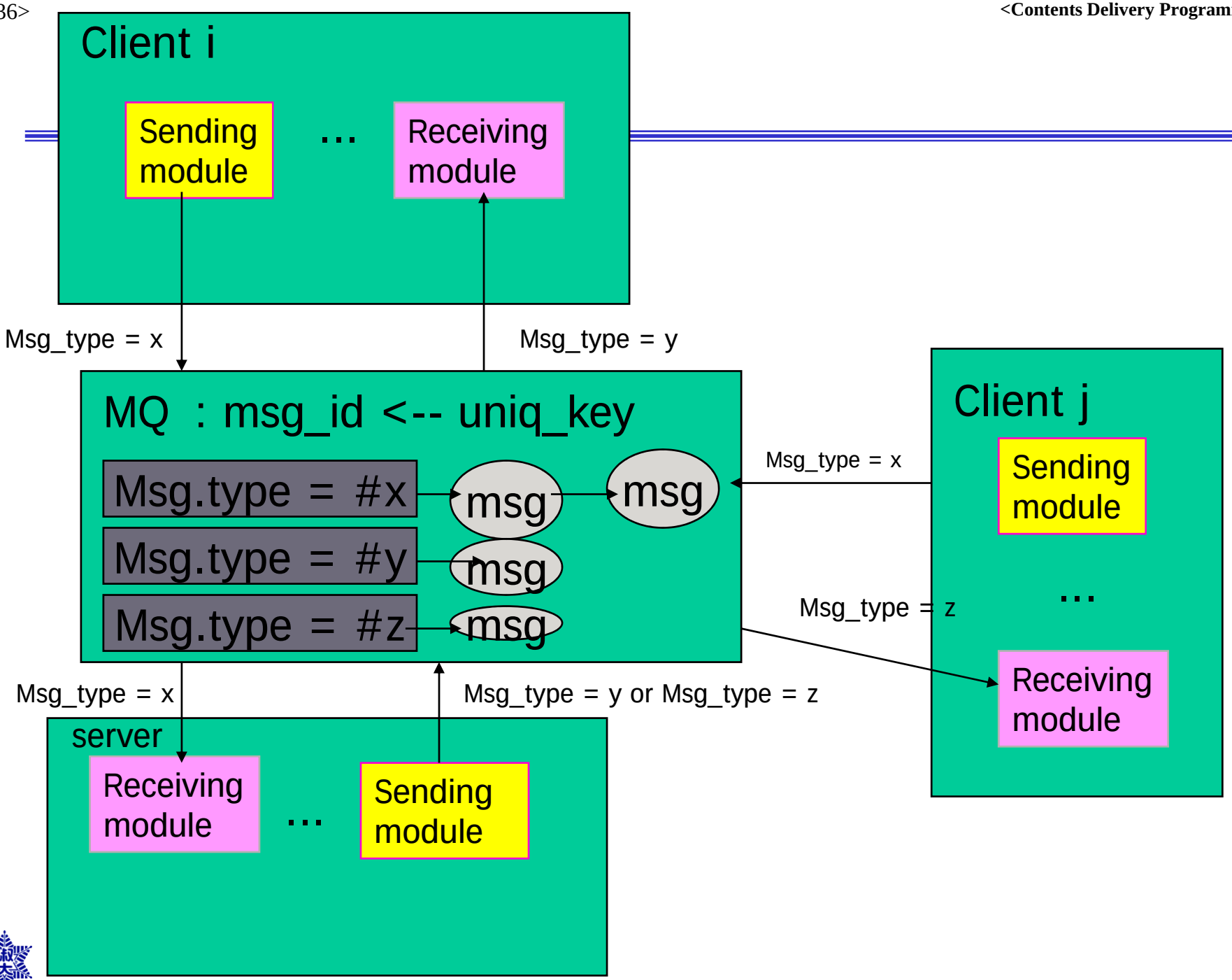
* *cmd* : IPC_RMID는 message queue를 제거하는 것으로 여기서는 이것만 사용

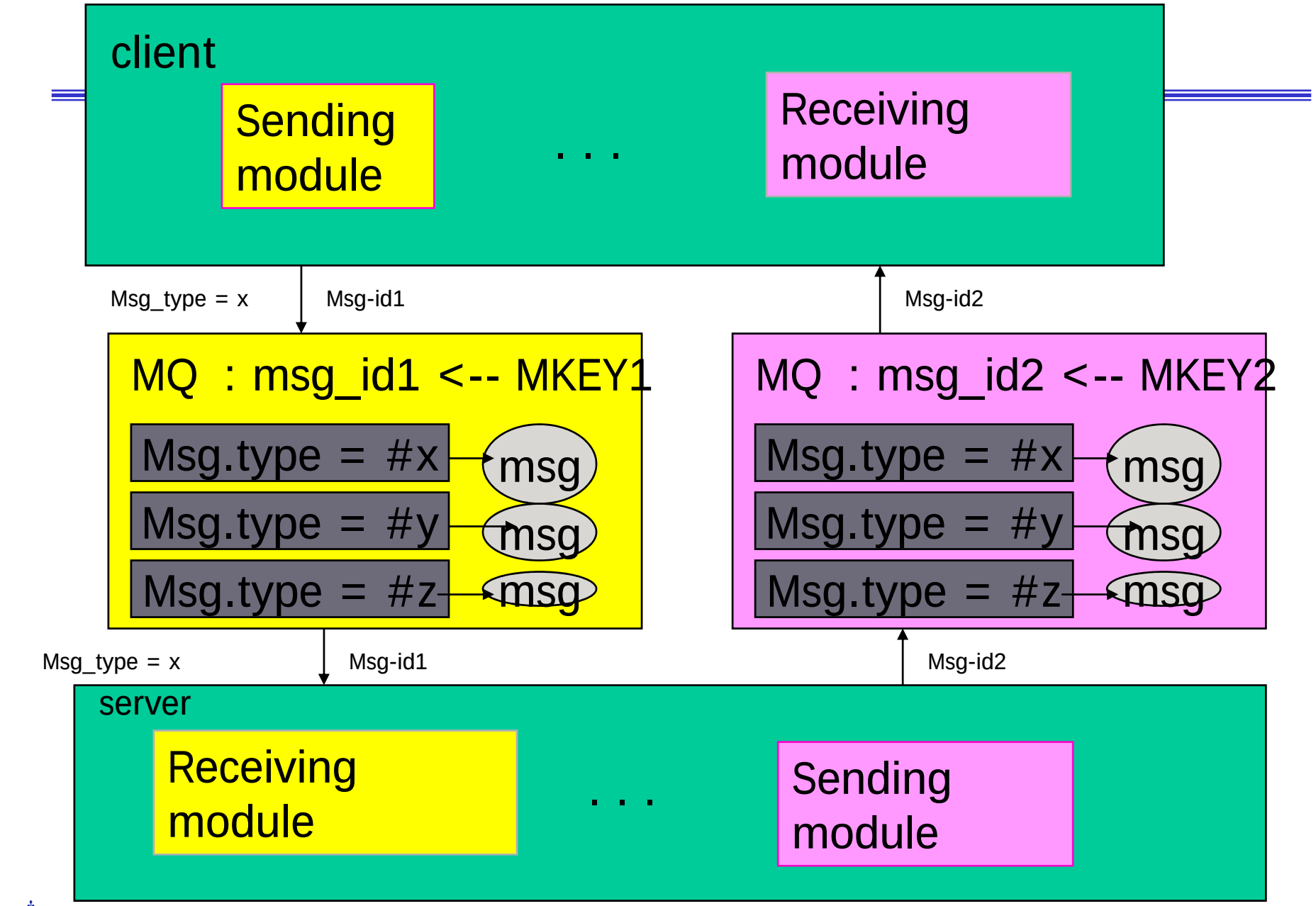
☐ Multiplexing Messages
- 각 message마다 관련된 type을 갖는 목적
  : 다수의 process들이 하나의 queue에서 message들을 다중화할 수 있도록 하기 위해서

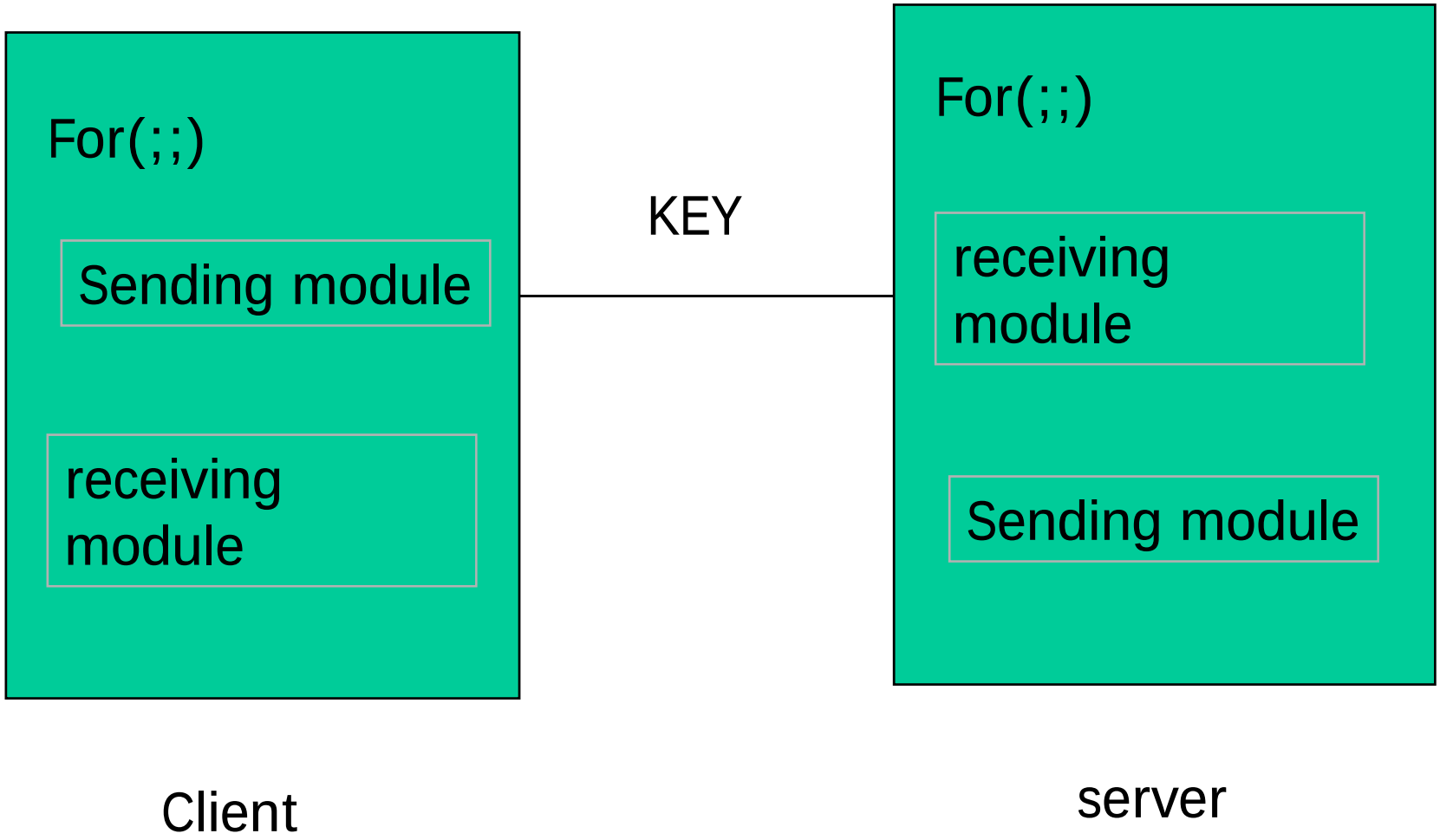


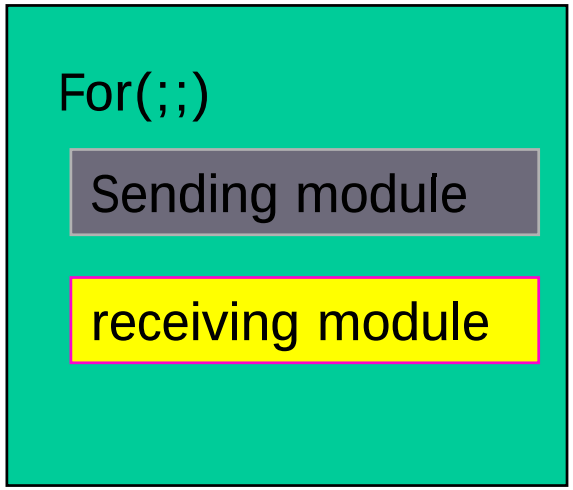

Client Y
Msgget(KEY) --> msg_id
Msg 생성: msg.type, msg.content
msgsnd(msg_id, msg, length)
Msg.type = x
Msg.type =y
MQ:
Uniq_ID -> KEY(msg_id)
Msg type #x
Msg type #y
Msg type #z
msg
msg
msg
msg
Msg.type = x
Client Z
Msg.type =z
Msg.type = x
Msg.type =y
Msg.type =z
server
Msgget(KEY) --> msg_id
Msg 공간 확보
msgrcv(msg_id, &msg, type, length)

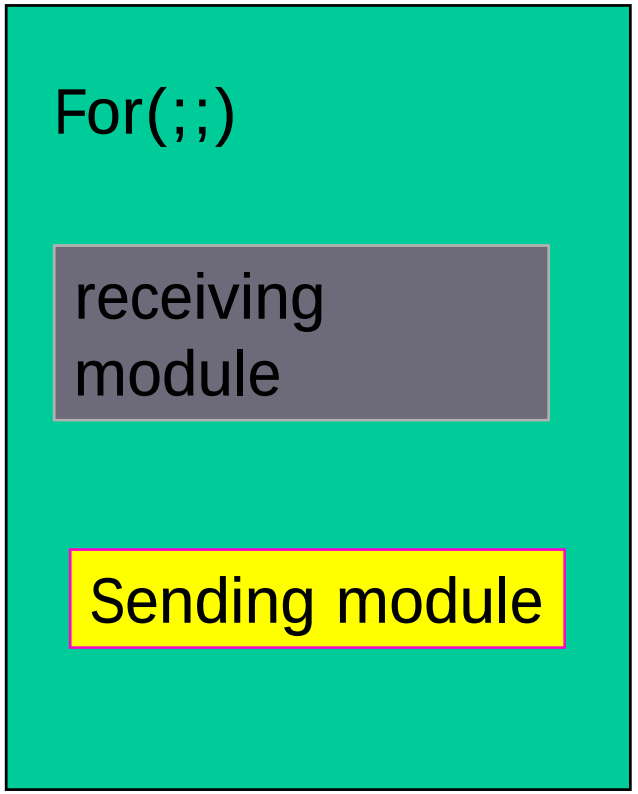

client
Sending module
Msg_id = msgget(KEY)
msg 생성: - msg의 공간(영역) 할당
- 보낼 내용을 작성 -> msg 복사
- msg type을 정의
- msg 길이
Msgsnd(msg_id, msg.type, msg.content, length)
Msg_type = x
MQ : msg_id <-- uniq_key
Msg.type = #x
Msg.type = #y
Msg.type = #z
msg
msg
msg
msg
Msg_type = x
server
Receiving module
msg_id = msgget(KEY)
msg를 받을 공간 확보
받을 msg의 type 설정, 길이 설정
Msgrcv(msg_id, &mbuf, length, type, flag)


Client i
Sending module
...
Receiving module
Msg_type = x
Msg_type = y
MQ : msg_id <-- uniq_key
Msg.type = #x
Msg.type = #y
Msg.type = #z
msg
msg
msg
msg
Msg_type = x
Client j
Sending module
...
Msg_type = z
Receiving module
Msg_type = x
Msg_type = y or Msg_type = z
server
Receiving module
...
Sending module


client
Sending module
. . .
Receiving module
Msg_type = x
Msg-id1
Msg-id2
MQ : msg_id1 <-- MKEY1
Msg.type = #x
Msg.type = #y
Msg.type = #z
msg
msg
msg
MQ : msg_id2 <-- MKEY2
Msg.type = #x
Msg.type = #y
Msg.type = #z
msg
msg
msg
Msg_type = x
Msg-id1
Msg-id2
server
Receiving module
. . .
Sending module


For(;;)
Sending module
receiving module
KEY
For(;;)
receiving module
Sending module
Client
server


For(;;)
Sending module
receiving module


For(;;)
receiving module
Sending module

두개의 Message queue를 사용하는 client-server의 예제(1)

☐ msgq.h

```
#include    <sys/types.h>
#include    <sys/ipc.h>
#include    <sys/msg.h>

#include    <sys/errno.h>
extern int  errno;

#define     MKEY1     1234L
#define     MKEY2     2345L

#define     PERMS     0666
```

☐ Server를 위한 main 함수

```
#include    "msgq.h"

main()
{
            int         readid, writeid;

            /*
             * Create the message queues, if required.
             */

            if ( (readid = msgget(MKEY1, PERMS | IPC_CREAT)) < 0)
                        err_sys("server: can't get message queue 1");
            if ( (writeid = msgget(MKEY2, PERMS | IPC_CREAT)) < 0)
                        err_sys("server: can't get message queue 2");

            server(readid, writeid);

            exit(0);
}
```

두개의 Message queue를 사용하는 client-server의 예제(2)

□ Client를 위한 main 함수

```
#include    "msgq.h"

main()
{
            int         readid, writeid;

            /*
             * Open the message queues.  The server must have
             * already created them.
             */
            if ( (writeid = msgget(MKEY1, 0)) < 0)
                        err_sys("client: can't msgget message queue 1");
            if ( (readid = msgget(MKEY2, 0)) < 0)
                        err_sys("client: can't msgget message queue 2");
            client(readid, writeid);

            /*
             * Now we can delete the message queues.
             */
            if (msgctl(readid, IPC_RMID, (struct msqid_ds *) 0) < 0)
                        err_sys("client: can't RMID message queue 1");
            if (msgctl(writeid, IPC_RMID, (struct msqid_ds *) 0) < 0)
                        err_sys("client: can't RMID message queue 2");
            exit(0);
}
```

두개의 Message queue를 사용하는 client-server의 예제(3)

☐ mesg.h

```
/*
 * Definition of "our" message.
 *
 * You may have to change the 4096 to a smaller value, if message queues
 * on your system were configured with "msgmax" less than 4096.
 */

#define     MAXMESGDATA (4096-16)
                                                /* we don't want sizeof(Mesg) > 4096 */

#define     MESGHDRSIZE (sizeof(Mesg) - MAXMESGDATA)
                                                /* length of mesg_len and mesg_type */

typedef struct {
  int       mesg_len;   /* #bytes in mesg_data, can be 0 or > 0 */
  long      mesg_type;  /* message type, must be > 0 */
  char      mesg_data[MAXMESGDATA];
} Mesg;
```

두개의 Message queue를 사용하는 client-server의 예제(4)

□ mesg_send 함수

```
#include    "mesg.h"

/*
 * Send a message using the System V message queues.
 * The mesg_len, mesg_type and mesg_data fields must be filled
 * in by the caller.
 */

mesg_send(id, mesgptr)
int         id;                     /* really an msqid from msgget() */
Mesg        *mesgptr;
{
            /*
             * Send the message - the type followed by the optional data.
             */

            if (msgsnd(id, (char *) &(mesgptr->mesg_type), mesgptr->mesg_len, 0) != 0)
                        err_sys("msgsnd error");
}
```

두개의 Message queue를 사용하는 client-server의 예제(5)

□ mesg_rcv 함수

```
#include    "mesg.h"

/*
 * Receive a message from a System V message queue.
 * The caller must fill in the mesg_type field with the desired type.
 * Return the number of bytes in the data portion of the message.
 * A 0-length data message implies end-of-file.
 */

int mesg_recv(id, mesgptr)
int         id;                     /* really an msqid from msgget() */
Mesg        *mesgptr;
{
            int         n;

            /*
             * Read the first message on the queue of the specified type.
             */
            n = msgrcv(id, (char *) &(mesgptr->mesg_type), MAXMESGDATA,mesgptr->mesg_type, 0);
            if ( (mesgptr->mesg_len = n) < 0)
                        err_dump("msgrcv error");
            return(n);              /* n will be 0 at end of file */
}
```

하나의 Message queue를 사용하는 client-server의 예제(1)

☐ Server program

```
#include    <stdio.h>
#include    "mesg.h"
#include    "msgq.h"

Mesg        mesg;

main()
{
            int         id;

            /*
             * Create the message queue, if required.
             */
            if ( (id = msgget(MKEY1, PERMS | IPC_CREAT)) < 0)
                        err_sys("server: can't get message queue 1");
            server(id);
            exit(0);
}
```

하나의 Message queue를 사용하는 client-server의 예제(2)

```
server(id)
int         id;
{
            int         n, filefd;
            char        errmesg[256], *sys_err_str();

            /*
             * Read the filename message from the IPC descriptor.
             */
            mesg.mesg_type = 1L;                /* receive messages of this type */
            if ( (n = mesg_recv(id, &mesg)) <= 0)
                        err_sys("server: filename read error");
            mesg.mesg_data[n] = '₩0';           /* null terminate filename */

            mesg.mesg_type = 2L;                /* send messages of this type */
            if ( (filefd = open(mesg.mesg_data, 0)) < 0) {
                        /*
                         * Error.  Format an error message and send it back
                         * to the client.
                         */
                        sprintf(errmesg, ": can't open, %s₩n", sys_err_str());
                        strcat(mesg.mesg_data, errmesg);
                        mesg.mesg_len = strlen(mesg.mesg_data);
                        mesg_send(id, &mesg);
```

하나의 Message queue를 사용하는 client-server의 예제(3)

```
        } else {
                /*
                 * Read the data from the file and send a message to
                 * the IPC descriptor.
                 */
                while ( (n = read(filefd, mesg.mesg_data, MAXMESGDATA)) > 0) {
                        mesg.mesg_len = n;
                        mesg_send(id, &mesg);
                }
                close(filefd);
                if (n < 0)
                        err_sys("server: read error");
        }

        /*
         * Send a message with a length of 0 to signify the end.
         */
        mesg.mesg_len = 0;
        mesg_send(id, &mesg);
}
```

하나의 Message queue를 사용하는 client-server의 예제(4)

☐ Client program

```
#include    <stdio.h>
#include    "mesg.h"
#include    "msgq.h"

Mesg        mesg;

main()
{
            int         id;

            /*
             * Open the single message queue.  The server must have
             * already created it.
             */
            if ( (id = msgget(MKEY1, 0)) < 0)
                        err_sys("client: can't msgget message queue 1");
            client(id);

            /*
             * Now we can delete the message queue.
             */
            if (msgctl(id, IPC_RMID, (struct msqid_ds *) 0) < 0)
                        err_sys("client: can't RMID message queue 1");
            exit(0);
}
```

하나의 Message queue를 사용하는 client-server의 예제(5)

```
client(id)
int         id;
{
            int         n;
            /*
             * Read the filename from standard input, write it as
             * a message to the IPC descriptor.
             */
            if (fgets(mesg.mesg_data, MAXMESGDATA, stdin) == NULL)
                        err_sys("filename read error");
            n = strlen(mesg.mesg_data);
            if (mesg.mesg_data[n-1] == '₩n')    n--;     /* ignore the newline from fgets() */
            mesg.mesg_data[n] = '₩0';                    /* overwrite newline at end */
            mesg.mesg_len = n;
            mesg.mesg_type = 1L;                         /* send messages of this type */
            mesg_send(id, &mesg);
            /*
             * Receive the message from the IPC descriptor and write
             * the data to the standard output.
             */
            mesg.mesg_type = 2L;    /* receive messages of this type */
            while( (n = mesg_recv(id, &mesg)) > 0)
                        if (write(1, mesg.mesg_data, n) != n)
                                    err_sys("data write error");
            if (n < 0)  err_sys("data read error");
}
```

Semaphores(1)

□ semaphore
- 동기화의 기본
- 다수의 process들의 작업을 동기화하기위해서 사용
- 주로 사용하는곳은 shared memory segment에의 접근을 동기화하기 위함

□ 시스템내의 모든 semaphore의 집합에 대해 kernel은 다음과 같은 구조를 유지

```
#include <sys/types.h>
#include <sys/ipc.h>    /* defines the ipc_perm structure */

struct semid_ds {
   struct ipc_perm sem_perm;        /* operation permission struct */
   struct sem      *sem_base;       /* ptr to first semaphore in set */
   ushort             sem_nsems;    /* # of semaphores in semop */
   time_t             sem_otime;    /* time of last semop */
   time_t             sem_ctime;    /* time of last change */
};
```

*
```
struct sem {
    ushort semval;      /* semaphore value, nonnegative */
    short   sempid;     /* pid of last operation */
    ushort semncnt;     /* # awaiting semval > cval */
    ushort semzcnt;     /* # awaiting semval = 0 */
};
```

Semaphores(2)

☐ semget system call
- semaphore를 만들거나 기존의 semaphore에 접근

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>

int semget(key_t key, int nsems, int semflag);
```

- nsems : semaphore의 수
- semflag : 다음 상수들의 조합

Numeric	Symbolic	Description
0400	SEM_R	Read by owner
0200	SEM_W	Write by owner
0040	SEM_R >> 3	Read by group
0020	SEM_W >> 3	Write by group
0004	SEM_R >> 6	Read by world
0002	SEM_W >> 6	Write by world
	IPC_CREAT	
	IPC_EXCL	

Semaphores(3)

☐ semop system call

- 집합내의 하나 또는 그 이상의 semaphore 값들에 대해 연산을 수행

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>

int semop(int semid, struct sembuf **opsptr, unsigned int nops);
```

- *opsptr* :

```
struct sembuf {
    ushort sem_num;      /* semapore # */
    short  sem_op;       /* semaphore operation */
    short  sem_flg;      /* operation flags */
};
```

- sem_op :
  - ① sem_op > 0 : sem_val의 값이 semaphore의 현재 값에 더해짐
    이것은 semaphore가 제어하는 자원들의 해제에 해당
  - ② sem_op = 0 : 호출한 process는 semaphore의 값이 0이 될때까지 기다리기를 원함
  - ③ sem_op < 0 : 호출한 process는 semaphore의 값이 sem_op의 절대값보다 크거나 같아질 때까지 기다린다.
    이것은 자원의 할당에 해당

Semaphores(4)

☐ semctl system call
- semaphore에 대한 다양한 제어 연산들을 제공

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>

int semctl(int semid, int semnum, int cmd, union semun arg);

  union semun {
      int               val;        /* used for SETVAL only */
      struct semid_ds   *buff;      /* used for IPC_STAT and IPC_SET */
      ushort            *array;     /* used for IPC_GETALL & IPC_SETALL */
  } arg;
```

- *cmd* : IPC_RMID는 semaphore를 제거하기 위해 사용
  GETVAL은 semaphore 값을 가져오기 위해 사용
  SETVAL은 명시된 semaphore 값으로 바꾸기 위해 사용

Semaphores를 사용한 예제

```
#include    <sys/types.h>
#include    <sys/ipc.h>
#include    <sys/sem.h>
#define SEMKEY          123456L     /* key value for semget() */
#define PERMS           0666
static struct sembuf    op_lock[2] = {
            0, 0, 0,    /* wait for sem#0 to become 0 */
            0, 1, 0     /* then increment sem#0 by 1 */ };
static struct sembuf    op_unlock[1] = {
            0, -1, IPC_NOWAIT       /* decrement sem#0 by 1 (sets it to 0) */ };
int         semid = -1; /* semaphore */

my_lock(fd)
int         fd;
{           if (semid < 0)
                        if ( (semid = semget(SEMKEY, 1, IPC_CREAT | PERMS)) < 0)
                                    err_sys("semget error");
            if (semop(semid, &op_lock[0], 2) < 0)
                        err_sys("semop lock error");
}

my_unlock(fd)
int         fd;
{           if (semop(semid, &op_unlock[0], 1)< 0)
                        err_sys("semop unlock error);
}
```

Shared Memory(1)

☐ Client-Server 화일 복제 프로그램에 사용되는 통상적인 몇가지 단계를 고려해보자.

- server가 입력 화일에서 읽는다. 이러한 동작은 보통 kernel이 데이타를 자신의 내부 block buffer중의 하나로 읽어들이고, 이것을 server의 buffer로 복제하는 두가지 동작으로 이루어진다.

- server는 pipe, FIFO, message queue 등의 기법중에 하나를 사용하여 이러한 data를 message안에 쓸 수 있다. 이러한 세가지 IPC 형태의 어느 것이나 데이타가 user의 buffer로 복제하여야 한다.

- client는 IPC channel에서 데이타를 읽는데 이때, 다시 데이타를 kernel의 IPC buffer 에서 client의 buffer로 복제하여야 한다.

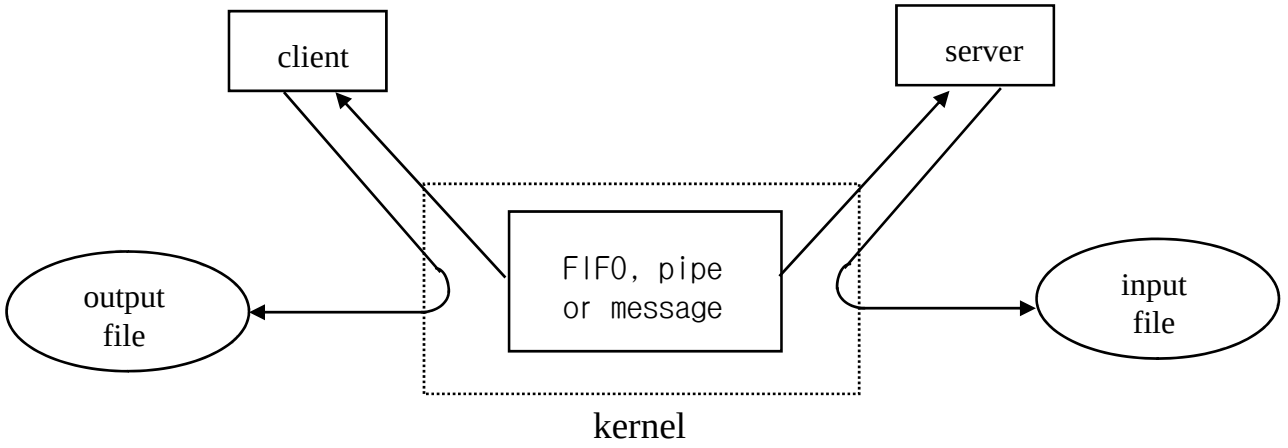
- 마지막으로 데이타는 system call write의 두번째 독립 변수인 client의 buffer에서 출력 화일로 복제된다.

☐ 전체적으로 데이타를 네번 복제하는 것이 필요

- 이러한 네번의 복제는 kernel과 사용자 process간에서 행해진다.
  - 이러한 복제를 intercontext copy 라고 한다.

Shared Memory(2)

☐ client와 server사이에서의 전형적인 데이타 이동



☐ 이러한 IPC 형태(pipe, FIFO, message queue)와 관련된 문제
- 두 프로세스가 정보를 교환하기 위해서 정보가 kernel을 거쳐가야 한다.

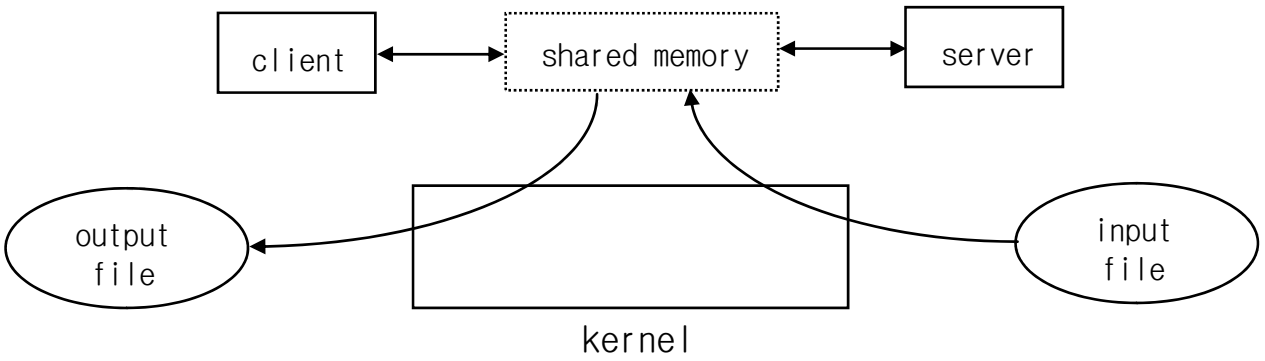
☐ shared memory
- 두개 이상의 프로세스가 기억장소의 일부를 공유하도록 해서 이러한 문제를 해결

Shared Memory(3)

☐ shared memory 를 이용한 Client-Server 화일 복제 프로그램에서의 몇가지 단계

- server는 semaphore를 사용하여 shared memory segment로 접근한다.
- server는 입력화일을 shared memory segment로 읽어들인다.
  읽어들일 주소는 shared memory를 가리킨다.
- 읽기가 완료되면 일꾼은 다시 semaphore를 사용하여 client에게 알린다.
- client는 shared memory segment에서 출력화일로 데이타를 쓴다.

☐ shared memory를 이용한 client와 server간의 데이타 이동



Shared Memory(4)

□ 모든 shared memory segment에 대해서 kernel은 다음과 같은 정보 구조를 유지

```
#include <sys/types.h>
#include <sys/ipc.h>                    /* defines the ipc_perm structure */

struct shmid_ds {
   struct ipc_perm shm_perm;            /* operation permission struct */
   int             shm_segsz;           /* segment size */
   struct XXX      shm_YYY;             /* implementation dependent info */
   ushort          shm_lpid;            /* pid of last operation */
   ushort          shm_cpid;            /* creator pid */
   ushort          shm_nattch;          /* current # attached */
   ushort          shm_cnattch;         /* in-core # attached */
   time_t          shm_atime;           /* last attach time */
   time_t          shm_dtime;           /* last detach time */
   time_t          shm_ctime;           /* last change time */
};
```

Shared Memory(5)

☐ shmget system call

- shared memory segment를 생성하거나 기존의 것에 접근

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>

int shmget(key_t key, int size, int shmflag);
```

- return value : shared memory 식별자 *shmid*
- nsems : semaphore의 수
- semflag : 다음 상수들의 조합

Numeric	Symbolic	Description
0400	SHM_R	Read by owner
0200	SHM_W	Write by owner
0040	SHM_R >> 3	Read by group
0020	SHM_W >> 3	Write by group
0004	SHM_R >> 6	Read by world
0002	SHM_W >> 6	Write by world
	IPC_CREAT	
	IPC_EXCL	

Shared Memory(6)

☐ shmat system call
- shared memory segment를 attach시킴

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>

int shmat(int shmid, char *shmaddr, int shmflag);
```

- return value :shared memory segment의 address
- shared memory의 address를 결정하는 규칙 :
  - ① *shmaddr* = 0 : system은 호출한 프로세스를 위한 주소를 선정
  - ② *shmaddr* <> 0 : 돌려주는 주소는 호출 프로세스가 *shmflag* 독립변수로 SHM_RND값을 명시했는지 여부에 따라 다르다.

☐ shmdt system call
- shared memory segment를 detach시킴

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>

int shmdt(char *shmaddr);
```

Shared Memory(7)

☐ shmctl system call
- shared memory segment를 제거시킴(*cmd*값으로 IPC_RMID를 명시)

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>

int shmctl(int shmid, int cmd, struct shmid_ds *buf);
```

☐ shm.h

```
#include        "mesg.h"

#define NBUFF               4               /* number of buffers in shared memory */
                                            /* (for multiple buffer version */
#define SHMKEY              ((key_t) 7890) /* base value for shmem key */

#define SEMKEY1             ((key_t) 7891) /* client semaphore key */
#define SEMKEY2             ((key_t) 7892) /* server semaphore key */

#define PERMS               0666
```

Part 3

Socket Programming
(Unix 기반)

유닉스 소켓 시스템 콜

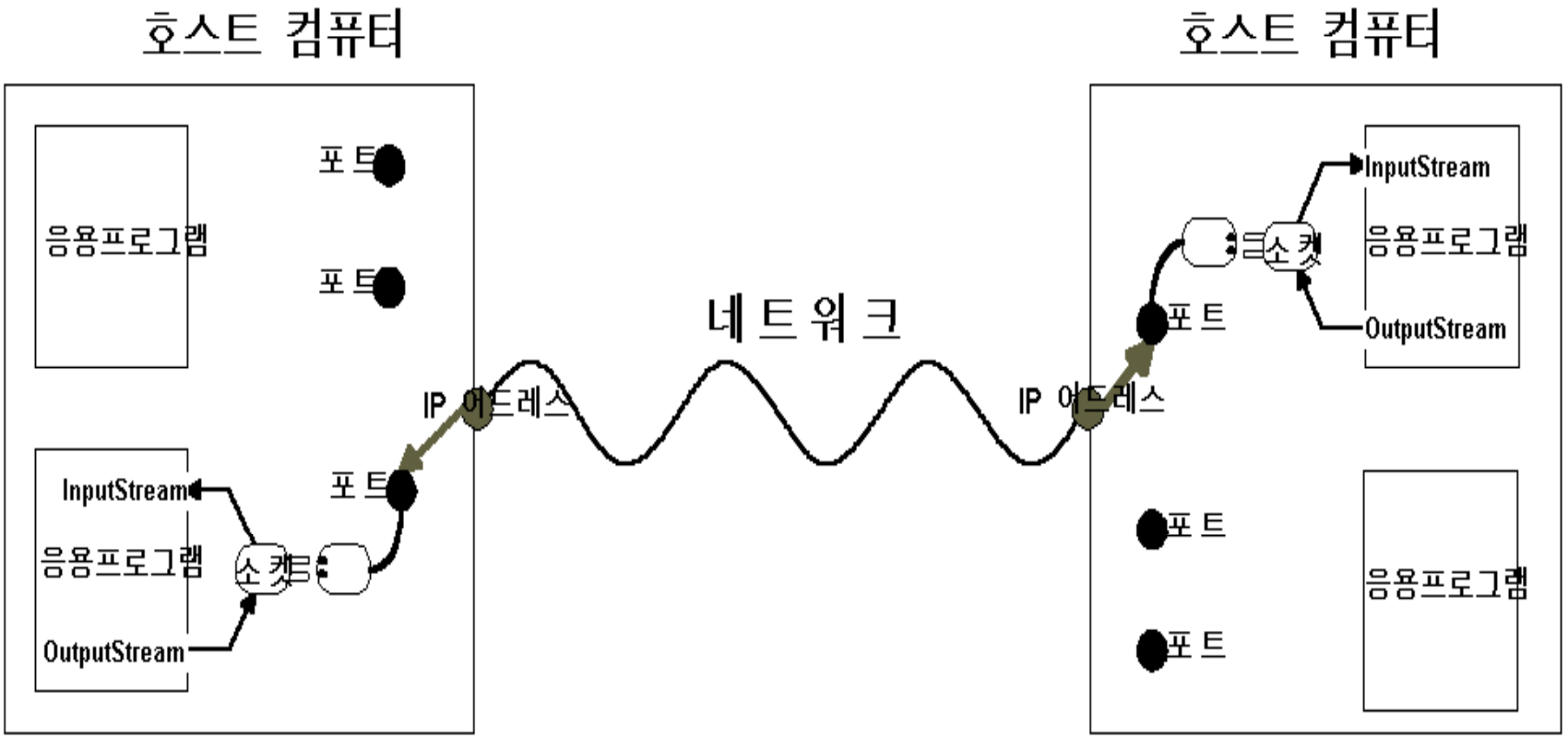
- BSD 소켓 API의 소개
- IP 주소변환 설명
- 소켓을 이용한 클라이언트 및 서버 프로그램 작성 방법 소개
- 유닉스 시스템 콜 signal()과 fork() 소개
- 토크 프로그램 작성

소켓의 정의

- TCP/IP를 이용하는 API
- 1982년 BSD(Berkeley Software Distribution) 유닉스 4.1
- 윈도우 소켓(윈속), 자바 소켓 등도 TCP/IP를 이용하기 위한 API를 제공

소켓

- 소켓 소개
  - IP 어드레스와 포트, 소켓의 관계



Berkeley socket

□ UNIX system을 위하여 가장 많이 보급되고 있는 두가지 통신 API

- Berkeley socket
- System V TLI(Transport Layer Interface)
- C programming language를 위하여 개발

□ Network I/O는 File I/O보다 더욱 상세하고 많은 선택사항들을 필요로 한다.

- 망 연결을 초기화하기 위해서 프로그램은 자신의 역할이 어떤 것 인지 알아야 한다.
- 망 연결은 연결 지향형 또는 비연결형이 될 수 있다.
- 이름은 화일 운용에 대해서보다 망에 있어서 더욱 중요하다.
- 망 연결에 필요한 매개변수들은 file I/O보다 많다.

  { 규약, 지역주소, 지역프로세스, 상대주소, 상대 프로세스 }

-몇몇 통신규약에서 레코드 경계는 중요하다.

- 망 접속은 다수의 통신규약을 지원해야 한다.

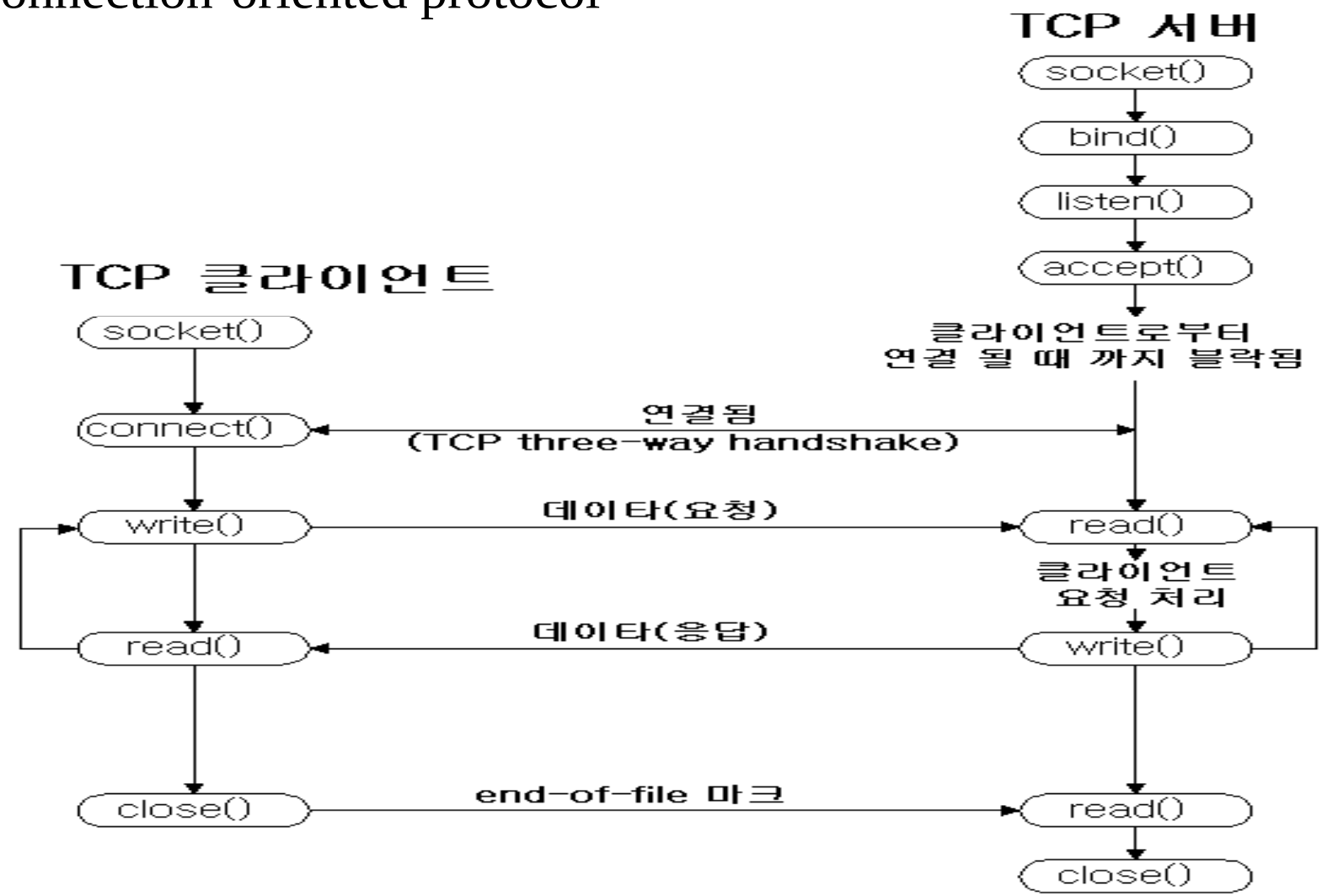
Berkeley Socket(계속)

☐ socket, TLI, Messages, FIFOs

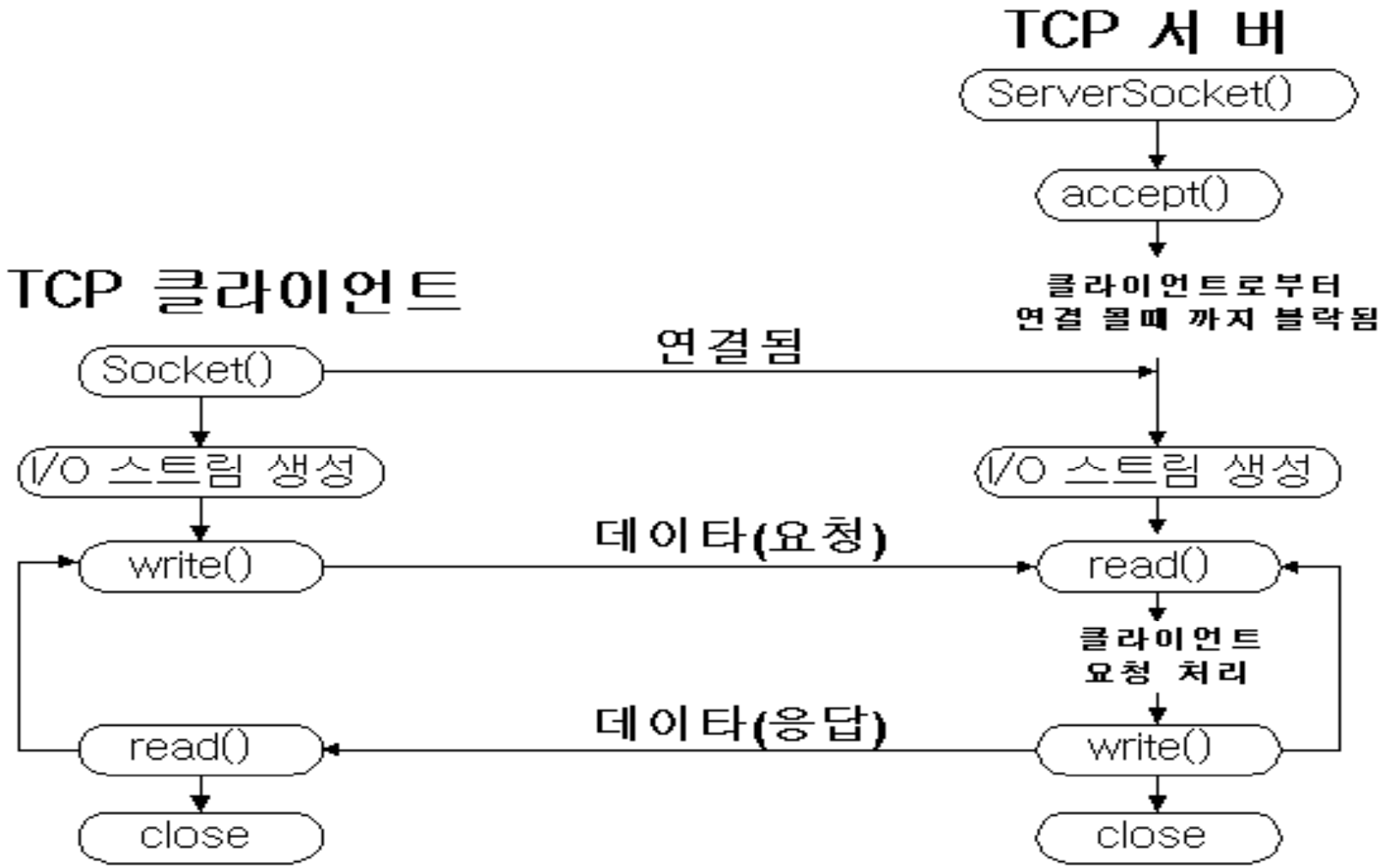
		Sockets	TLI	Messages	FIFOs
Server:	allocate space		t_alloc()		
	create endpoint	socket()	t_open()	msgget()	mknod() open()
	bind address	bind()	t_bind()		
	specify queue	listen()			
	wait for connection	accept()	t_listen()		
	get new fd		t_open()		
Client:	allocate space		t_alloc()		
	create endpoint	socket()	t_open()	msgget()	open()
	bind address	bind()	t_bind()		
	connect to server	connect()	t_connect()		
	transfer data	read() write() recv() send()	read() write() t_rcv() t_snd()	msgrcv() msgsnd()	read() write()
	datagrams	recvfrom() sendto()	t_rcvudata() t_sndudata()		
	terminate	close() shutdown()	t_close() t_sndrel() t_snddis()	msgctl()	close() unlink()

C 언어에서 TCP 소켓 프로그램 작성

connection-oriented protocol

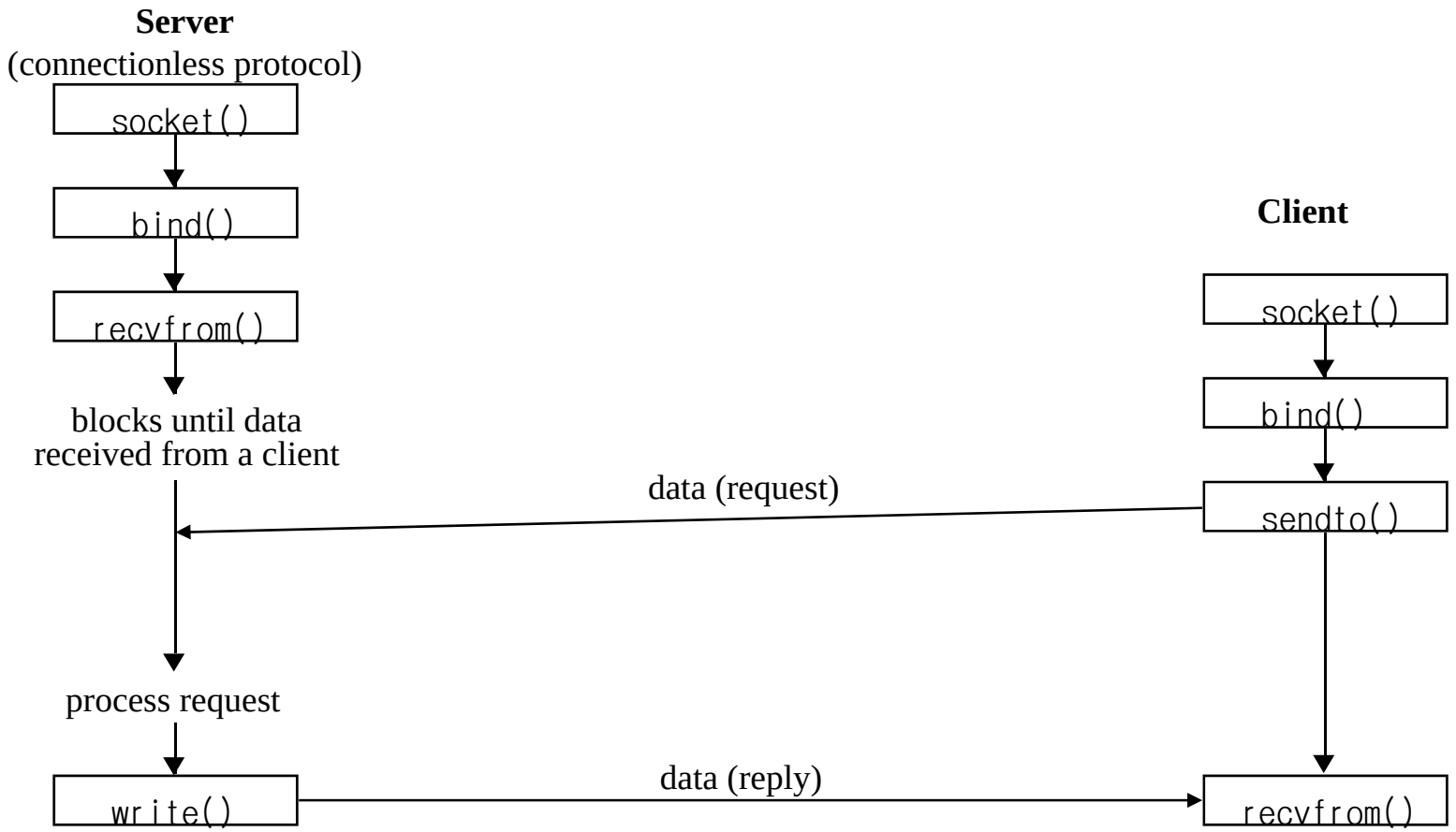


Java 에서 TCP 소켓 프로그램 작성



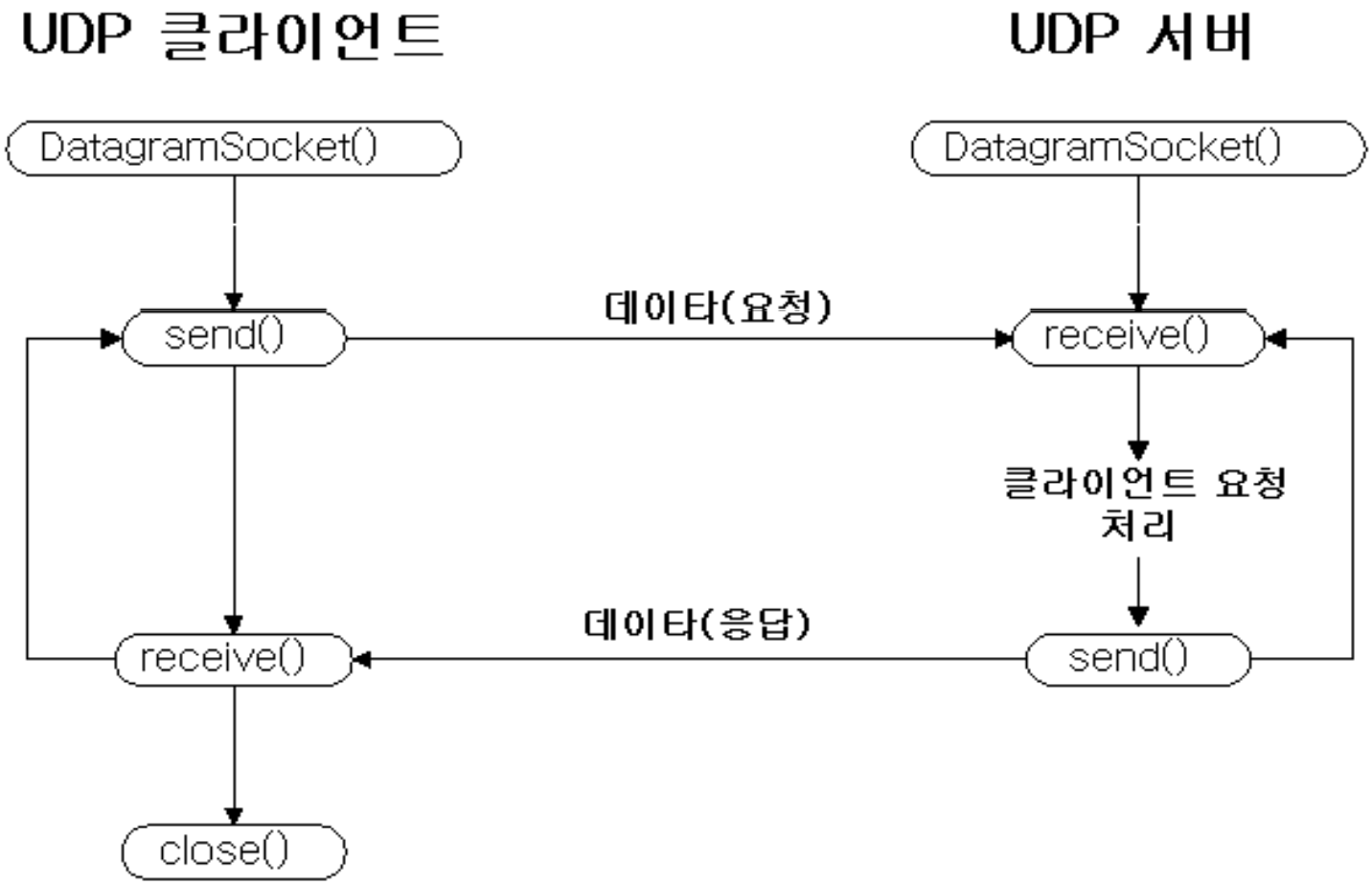
Berkeley Socket(계속)

☐ connectionless protocol에 대한 socket system calls



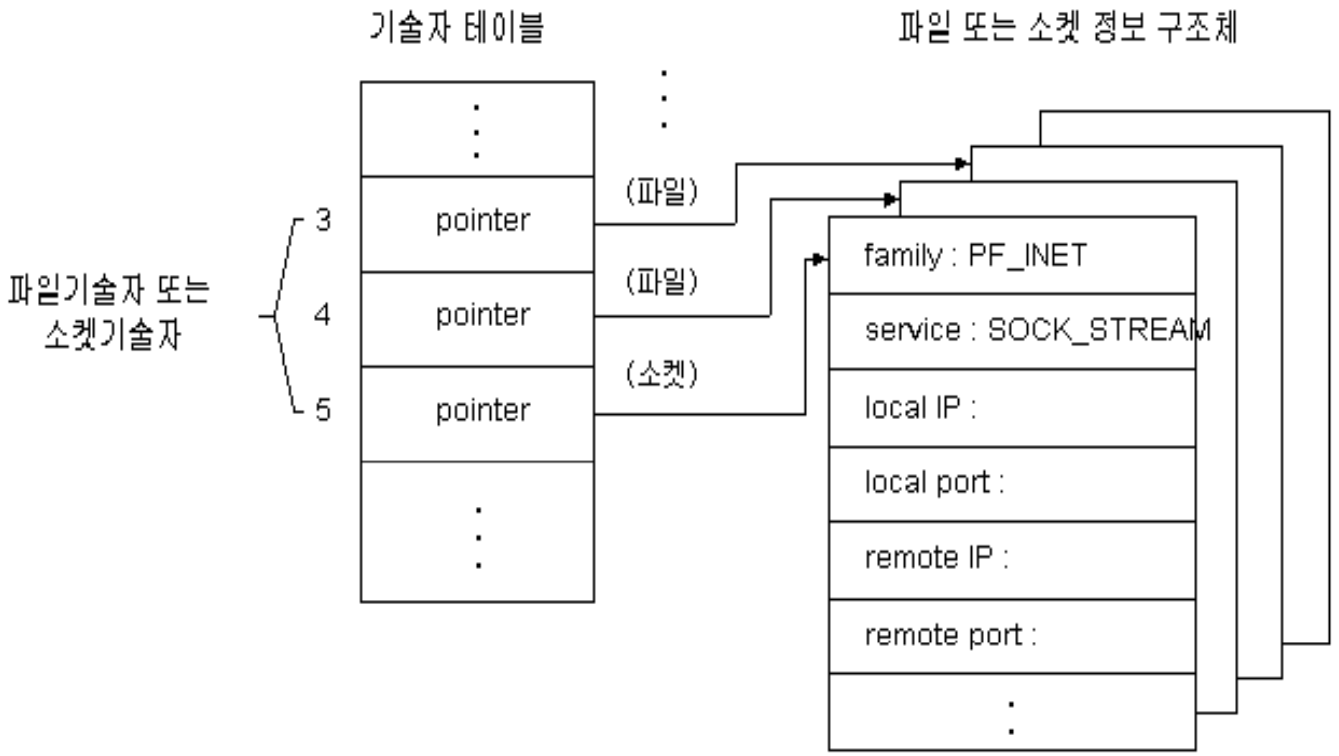
Java에서 DatagramSocket

- 자바 UTP 프로그램 작성

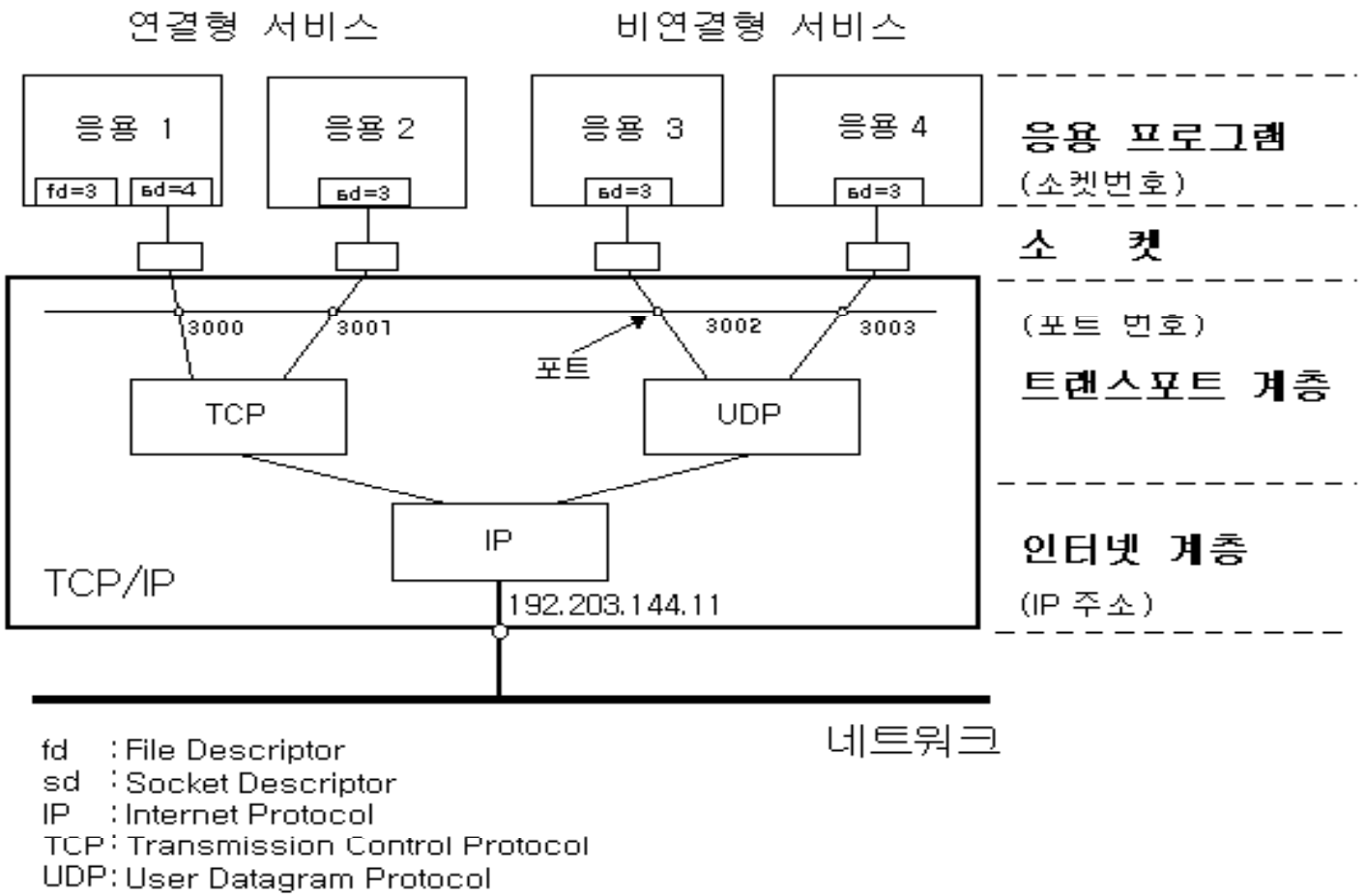


소켓 번호

- 소켓 개설시 리턴되는 값
- 기술자 테이블(descriptor table)에 위치



소켓 번호 (Cont'd)


연결형 서비스
비연결형 서비스
응용 1
응용 2
응용 3
응용 4
fd=3
fd=4
fd=3
fd=3
fd=3
응용 프로그램
(소켓번호)
소 켓
3000
3001
3002
3003
포트
(포트 번호)
TCP
UDP
트랜스포트 계층
IP
인터넷 계층
TCP/IP
192.203.144.11
(IP 주소)
네트워크
fd : File Descriptor
sd : Socket Descriptor
IP : Internet Protocol
TCP : Transmission Control Protocol
UDP : User Datagram Protocol

소켓의 개설

- 소켓을 이용한 데이터 송수신시 필요한 정보

1. 프로토콜 (TCP or UDP)
2. 자신의 IP 주소
3. 자신의 포트 번호
4. 상대방의 IP 주소
5. 상대방의 포트 번호

소켓의 개설 (Cont'd)

- 소켓 시스템 콜

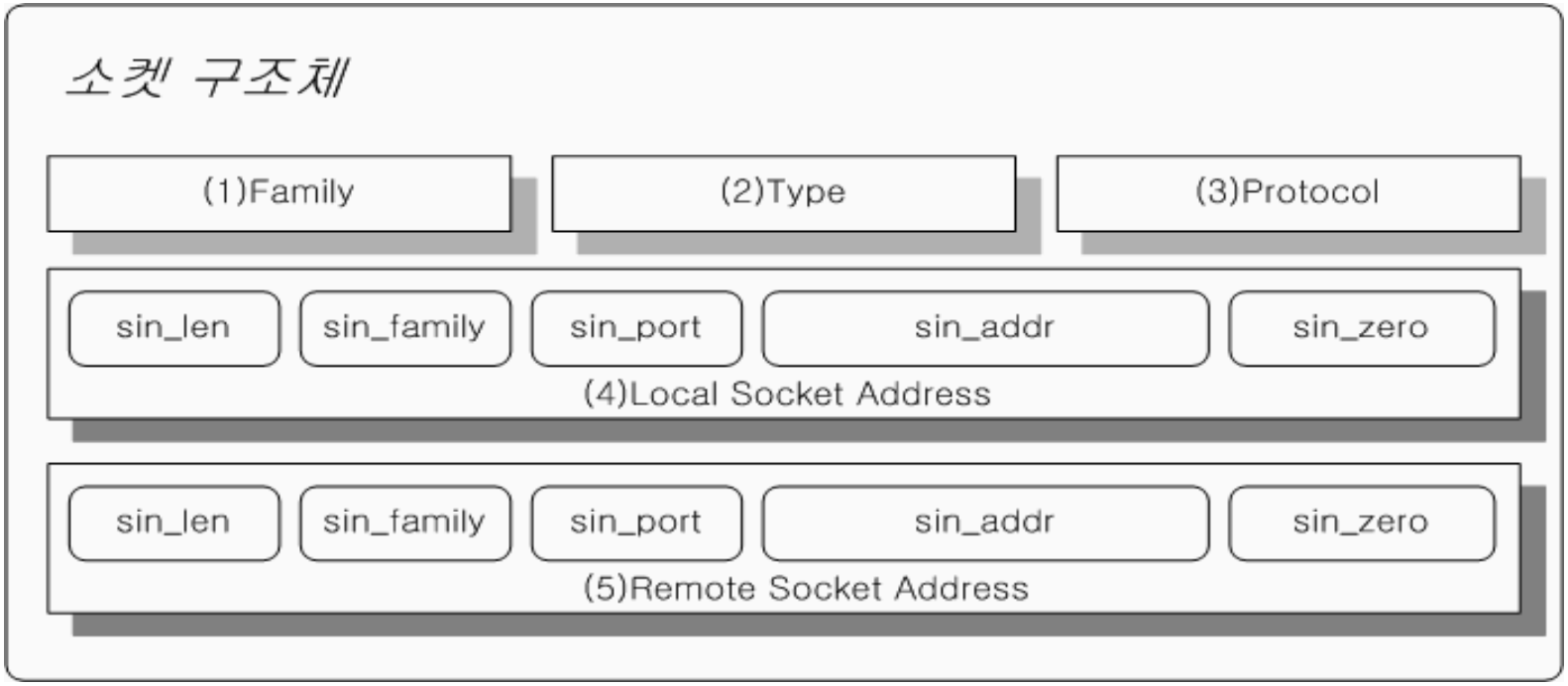
```
#include <sys/socket.h>
Int socket (
    int domain,          /* 프로토콜 체계 */
    int type,            /* 서비스 타입 */
    int protocol );      /* 프로토콜 */
```

소켓의 개설 (Cont'd)

- Domain
  - PF_INET / AF_INET
  - PF_INET6 / AF_INET6
  - PF_UNIX / AF_UNIX
    - PF : Protocol Family, AF: Address Family
    - Sockaddr / sockaddr_in의 구조체
- Type
  - SOCK_STREAM
  - SOCK_DGRAM
  - SOCK_RAW
- Protocol
  - 0 (자동적으로 프로토콜이 선택되어 진다)

소켓의 구조

1. 소켓 구조체



* 네트워크를 통하여 다른 컴퓨터와 통신을 하기 위해서는 위와 같은 다양한 속성의 설정이 필요하기 때문에, 이 후 등장할 여러 가지 속성 간의 특징 파악이 필요

소켓의 구조

2. Family

- 지원하는 프로토콜 그룹 중에서 사용하려는 소켓이 적용될 프로토콜을 선택

Family Protocol	프로토콜 상수 설명
AF_UNIX	유닉스 기본 파일 체계 소켓
AF_INET	IPV4의 인터넷 프로토콜
AF_INET6	IPV6의 인터넷 프로토콜
AF_NS	Xerox 네트워크 시스템 프로토콜
AF_ISO	ISO 프로토콜
AF_IPX	Novell IPX 프로토콜
AF_APPLETALK	Appletakl DDS

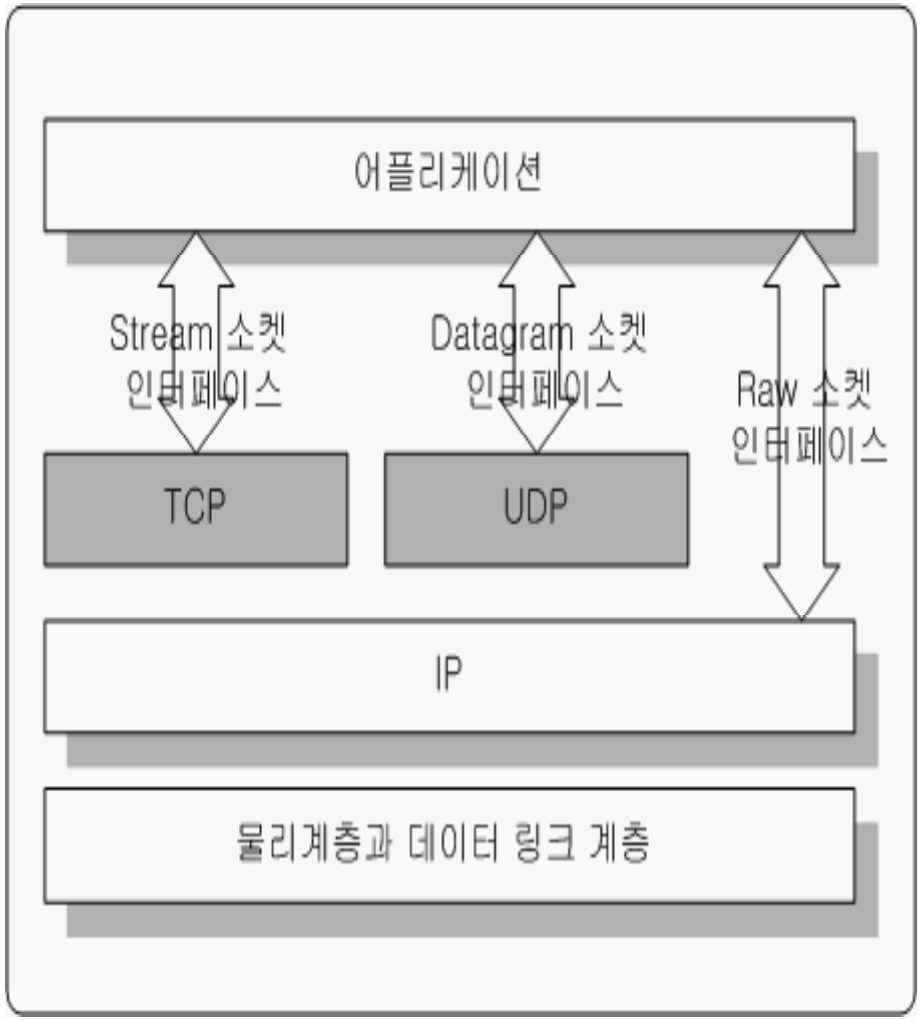
소켓의 구조

3. Type
   - 연결 지향형과 비 연결형, 저수준 프로토콜 제어형
   - 저 수준 프로토콜
     ○ **IP** 프로토콜과 같은 레벨에 있는 프로토콜을 사용 시 필요
     ex) ICMP (Internet Control Message Protocol)
     ○ **TCP/UDP** 보다 하위 계층으로 사용이 까다로운 반면
     직접적인 제어가 가능

Type	각 Type 상수에 대한 설명
SOCK_STREAM	TCP(스트림 소켓)
SOCK_DGRAM	UDP(데이터그램 소켓)
SOCK_RAW	Raw소켓

4. Protocol – 여러 소켓 형태를 제공하는 경우 사용, 기본 값은 **0**

소켓의 타입



1. Stream Socket

- 연결 지향형 **(TCP** 기반**)**
- 소켓 간의 연결 후 데이터 전송
- 일상 생활의 전화 개념과 유사

2. Datagram Socket

- 비 연결형 **(UDP** 기반**)**
- 송수신 시 도착지 주소 필수
- 일상 생활의 편지 개념과 유사

3. Raw Socket

- 저 수준 프로토콜 액세스
- **ICMP, OSPF** 등이 사용
- **IP** 계층 이용

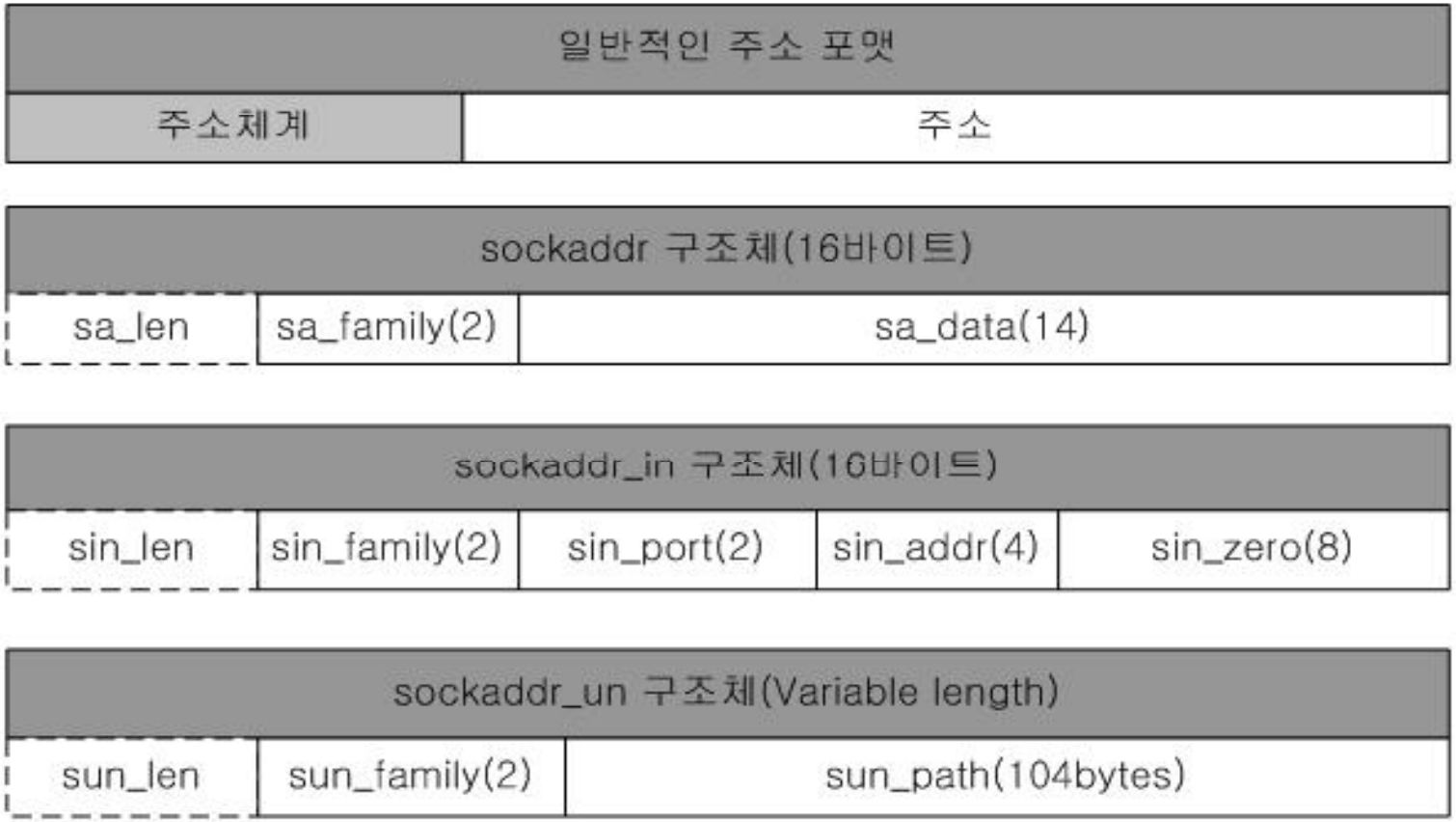
예제: 소켓 생성

- open_socket.c 예제

```
…
int fd1, sd1;
fd1 = open(“/etc/passwd”, O_RDONLY, 0);
sd1 = socket(PF_INET, SOCK_STREAM, 0);
printf(“File Descriptor %d\n”, fd1);
printf(“Socket Descriptor %d\n”, sd1);
close(fd1);
close(sd1);
```

  - 실행 결과
    - File Descriptor = 3
    - Socket Descriptor = 4

기본 자료 구조(1)



* 길이원소(len)는 4.3BSD-Reno에서 추가. 모든제작사가 소켓 주소 길이원소를 지원하지 않으며 Posix1.g에서도 이원소를 꼭 필요로 하는것은 아니다.

기본 자료 구조(2)

1. sockaddr 구조체
   - 기본 적인 소켓 주소 구조체
   - 호환성을 위해 존재
   - 변수 : 구조체의 크기, Family Type, 소켓의 주소 데이터

2. sockaddr_in 구조체
   - IPv4 소켓 주소 구조체
   - 주소를 담기 위해 in_addr 구조체 사용
   - 변수 : 구조체의 크기, Family Type, 소켓의 주소 및 포트 번호

3. sockaddr_un 구조체
   - 유닉스 소켓 주소 구조체
   - 동일 호스트에서의 통신이 일반 TCP통신보다 두 배 빠름
   - 변수 : 소켓 상수, 호스트 경로

소켓 주소 구조체

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

struct in_addr {
    u_long s_addr;              /* 32bit IP 주소 */
}

struct sockaddr_in {
    short  sin_family;              /* 주소 체계 */
    u_short sin_port;               /* 16bit 포트 번호 */
    struct    in_addr sin_addr;         /* 32bit IP 주소 */
    char      sin_zero[8];              /* dummy */
}
```

소켓 주소 구조체 (Cont'd)

- sockaddr_in 구조체를 사용
- sockaddr과의 호환성을 위해 dummy 삽입
- sin_family
  - AF_INET 인터넷 주소 체계
  - AF_UNIX 유닉스 파일 주소 체계
  - AF_NS XEROX 주소 체계

Socket Address

☐ 많은 BSD network system call은 독립변수로서 socket address 구조의 지시자를 필요로 한다.
- 이 구조의 정의는 <sys/socket.h>에 있다.

```
struct sockaddr {
        u_short     sa_family;          /* address family: AF_xxx value */
        char        sa_data[14];        /* up to 14 bytes of protocol-specific address */
};
```

☐ 14 byte의 protocol-specific address의 내용은 address의 type에 따라서 해석
☐ Internet family에 대하여 다음 구조가 <netinet/in.h>에 정의

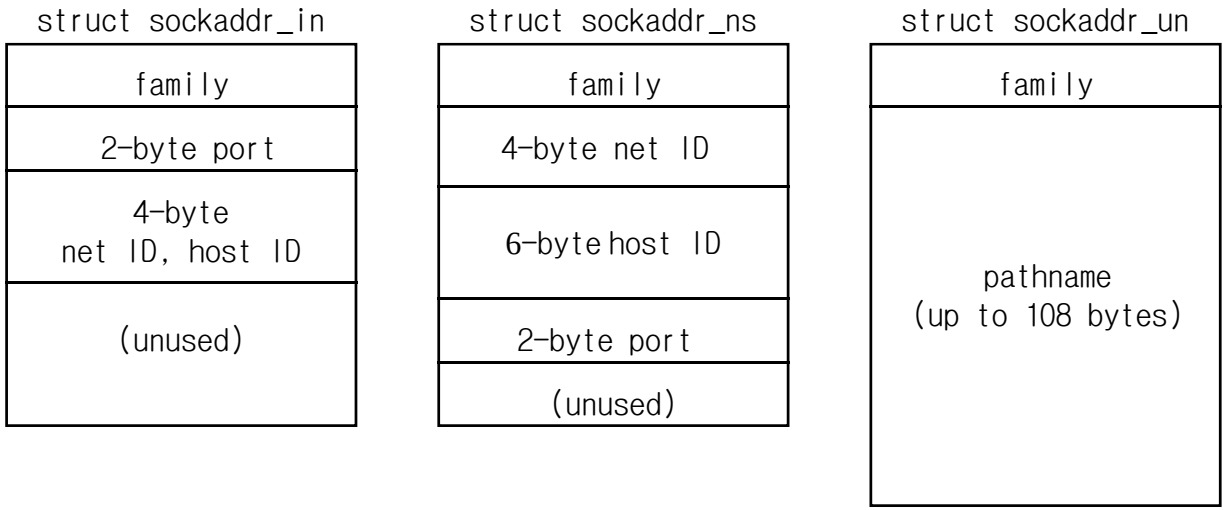
```
struct  in_addr {
        u_short                 s_addr;                 /* 32-bit netid/hostid */
};
struct sockaddr_in {
        short                   sin_family;  /* AF_INET */
        u_short                 sin_port;    /* 16-bit port number */
                                             /* network byte ordered */
        struct in_addr          sin_addr;    /* 32-bit netid/hostid */
                                             /* netowrk byte ordered */
        char                    sin_zero[8]; /* unused */
};
```

Socket Address(계속)

☐ <sys/types.h>에 정의된 unsigned data type

C Data type	4.3 BSD	System V
unsigned char	u_char	unchar
unsigned short	u_short	ushort
unsigned int	u_int	uint
unsigned long	u_long	ulong

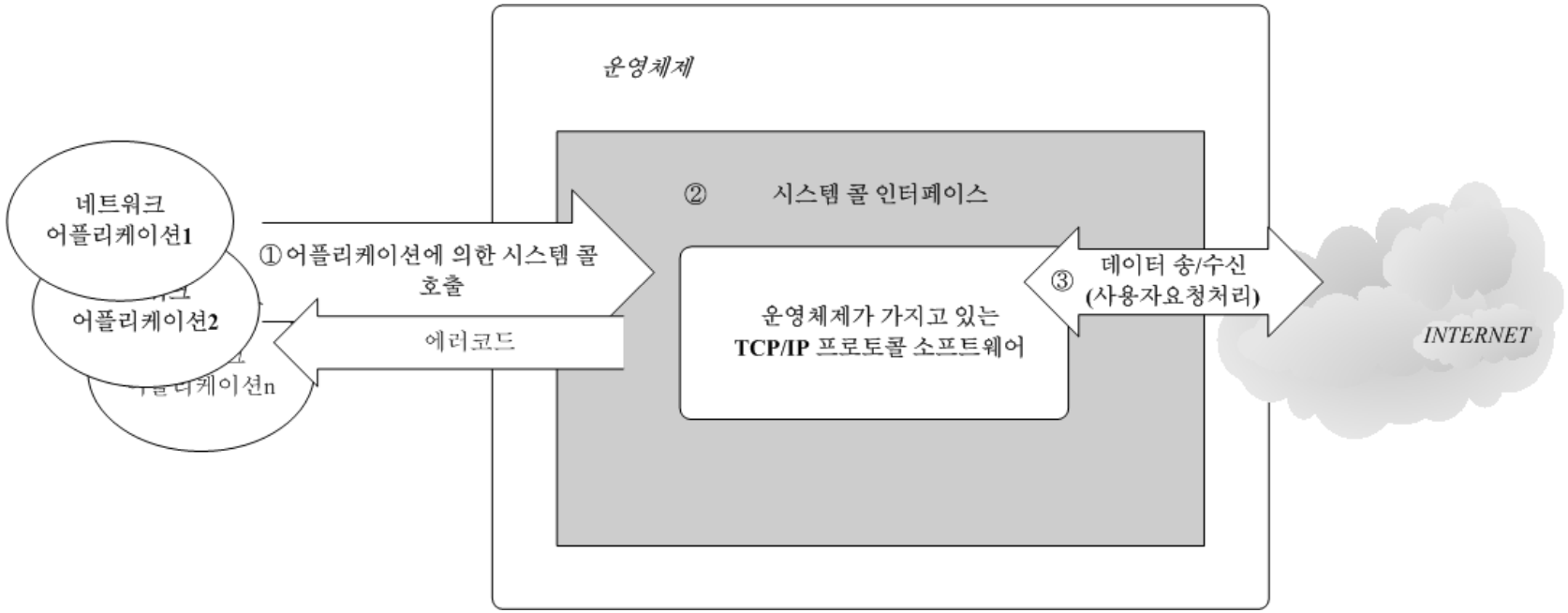
☐ Internet, XNS, Unix family에 대한 socket address 구조



네트워크 프로그래밍을 위한 함수

1. 시스템 호출 함수의 개요
2. 소켓 관련 함수
3. 소켓 옵션 관련 함수
4. 동시 처리를 위한 함수

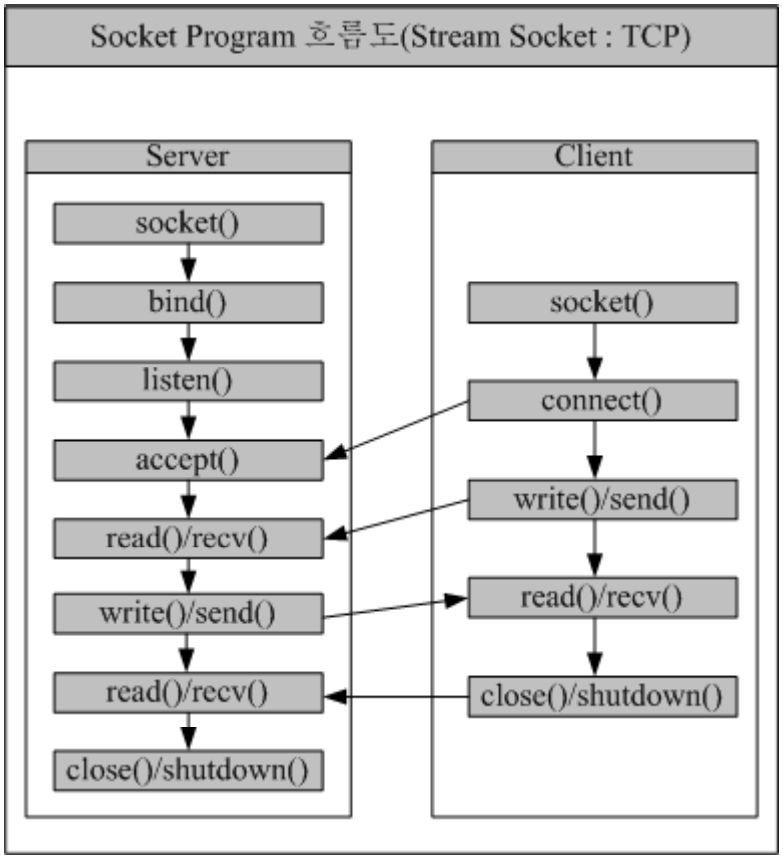
1. 시스템 호출 함수의 개요

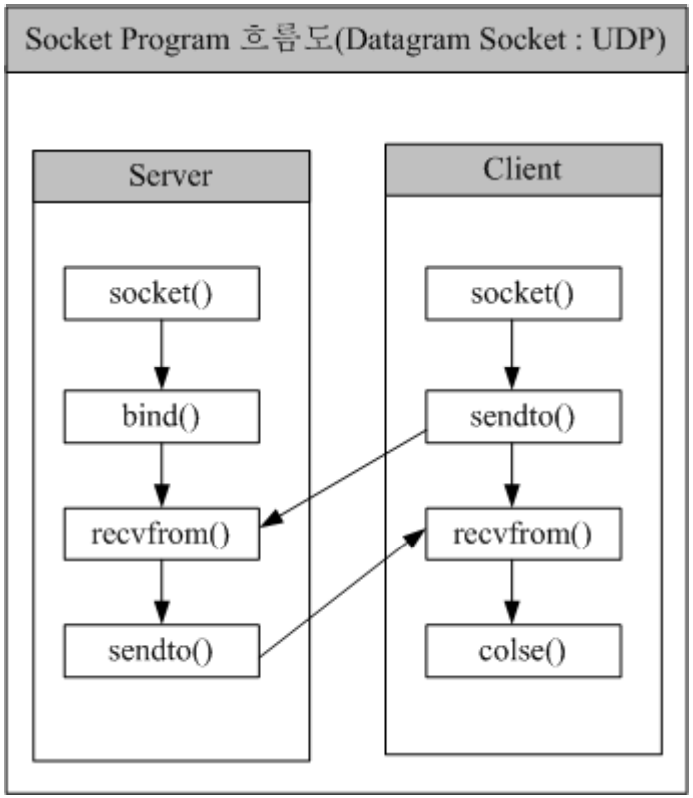


- 시스템 콜이란 개발의 편의를 위해 운영체제의 고유 기능을 호출하는 것
- 동작 매커니즘
  1. 필요에 의해 시스템 함수 호출
  2. 제어권한 이동(어플리케이션 > 시스템 호출 인터페이스 > 운영체제)
  3. 운영체제 내부 모듈이 호출에 대한 처리를 진행
  4. 제어권한 이동(운영체제 > 시스템 호출 인터페이스 > 어플리케이션)

2. 소켓 관련 함수

1. 소켓 함수를 이용한 클라이언트/서버의 구조





2. 소켓 관련 함수

☐ socket system call

- 원하는 통신 protocol(Internet TCP, Internet UDP, XNS SPP 등)의 type을 지정

☐ Syntax

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

```
#include <sys/types.h>
#include <sys/socket.h>

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

☐ socket system call

1. 네트워크 통신을 위해 사용될 새로운 소켓 생성
2. 파일 기술자 테이블에 새로 생성된 소켓의 인덱스를 파일 기술자 번호로 변환
   - return value : sock 지정번호, sockfd
3. 입력 인자
   - family : 프로토콜의 부류 입력

AF_UNIX	UNIX internal protocols
AF_INET	Internet protocols
AF_NS	Xerox NS protocols
AF_IMPLINK	IMP link layer

   - type : 소켓과 함께 사용할 통신 형태 입력

☐ socket type

-SOCK_STREAM	stream socket
-SOCK_DGRAM	datagram socket
-SOCK_RAW	raw socket
-SOCK_SEQPACKET	sequenced packet socket
-SOCK_RDM	reliably delivered message socket (not implemented yet)

   - protocol : TCP/UDP/RAW 입력, 일반적으로 '0'

Elementary Socket System call

☐ socket family와 type에 대응하는 protocol

	AF_UNIX	AF_INET	AF_NS
SOCK_STREAM	Yes	TCP	SPP
SOCK_DGRAM	Yes	UDP	IDP
SOCK_RAW		IP	Yes
SOCK_SEQPACKET			SPP

☐ family, type 그리고 protocol간의 조합

family	*type*	*protocol*	Actual protocol
AF_INET	SOCK_DGRAM	IPPROTO_UDP	UDP
AF_INET	SOCK_STREAM	IPPROTO_TCP	TCP
AF_INET	SOCK_RAW	IPPROTO_ICMP	ICMP
AF_INET	SOCK_RAW	IPPROTO_RAW	(raw)
AF_NS	SOCK_STREAM	NSPROTO_SPP	SPP
AF_NS	SOCK_SEQPACKET	NSPROTO_SPP	SPP
AF_NS	SOCK_RAW	NSPROTO_ERROR	Error protocol
AF_NS	SOCK_RAW	NSPROTO_RAW	(raw)

- IPPROTO_xxx
  - <netinet/in.h>
- NSPROTO_xxx
  - <netns/ns.h>

Elementary Socket System call(계속)

☐ socket system calls과 연계요소들

	protocol	*local-addr, local-process*	*foreign-addr, foreign-process*
connection-oriented server	socket()	bind()	listen(), accept()
connection-oriented client	socket()	connect()	
connectionless server	scoket()	bind()	recvfrom()
connectionless client	socket()	bind()	sendto()

☐ socketpair system call

- 이름없이 연결되어 있는 sockvec[0]과 sockvec[1]이라는 두가지 socket 지정 번호를 돌려줌

```
#include <sys/types.h>
#include <sys/socket.h>

int socketpair (int family, int type, int protocol, int sockvec[2]);
```

2. 소켓 관련 함수

bind system call

- 이름없는 socket에 이름을 부여
- 로컬 인터페이스 어드레스와 포트번호를 소켓과 서로 연관되게 묶는 함수입니다

Syntax

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

```
#include <sys/types.h>
#include <sys/socket.h>

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

bind의 용도

1. 서버 측에서 처음 생성된 소켓이 어떤 주소 값도 갖고 있지 않음
2. server는 주지된 주소를 시스템에 기록한다.
   - 소켓에 지역 주소 및 포트(Well-Known Port)를 할당 하여 활성화
3. 입력 인자
   - sockfd : socket() 함수 호출로 얻은 소켓의 기술자 입력
   - myaddr : 로컬 IP 및 포트에 대한 정보가 담긴 'sockaddr_in' 주소 입력
   - addrlen : 'sockaddr_in'의 크기 입력
4. client는 자신을 위한 특정 주소를 기록할 수 있다.
5. connectionless client는 시스템이 자신에게 유일한 주소를 부여하였음을 확인할 필요가 있으며, 이로써 상대측 server가 응답을 보낼 유효한 반송주소를 갖게 되는 것이다.

Bind

- 처음 만들어 질 때 소켓은 그와 관련된 로컬 어드레스를 가지지 않습니다. 따라서 **bind** 함수는 connect 나 listen 함수를 뒤이어 호출하기 전에, 접속되지 않은 소켓에 사용됩니다. 이 함수는 접속지향형소켓(stream socket : TCP) 이나 비접속(datagram : UDP) 소켓 이나 모두 사용합니다. 소켓이 socket 함수를 호출해서 생성될 때, 소켓은 이름공간(name space)에 존재합니다. 하지만, 할당된 이름을 가지지는 않습니다. 그러므로, 이름 지여지지 않은 소켓과 로컬 어드레스를 연결지우기 위해서 이 함수는 사용됩니다.
- 이 함수에 의해서 할당되어 지는 이름은 세가지 부분으로 구성됩니다. (인터넷 주소 체계를 사용할 때 : 주소체계(address family), 호스트주소(host address), 어플리케이션에 사용되는 포트번호(port number)) WinSock2 에서는 *name* 매개변수는 SOCKADDR 구조체의 포인터로 반드시 넘겨야 한다라는 그런 규정은 없습니다. 그저 윈도우즈 소켓 1.1 과의 호환성 때문에 남았는 산물정도라고 생각해 두세요. 서비스 프로바이더는 *namelen* 의 메모리 사이즈 블록 포인터에 대해 연관을 받지 않습니다. 메모리 블록에서 첫번째 2Byte (SOCKADDR 구조체의 sa_family 에 해당되는 부분입니다) 는 소켓을 생성하는데 사용되는 어드레스 체계(address family)를 가지고 있어야 합니다. 그렇지 않다면, WSAEFAULT 에러가 발생 할 것입니다.
- 만약 어플리케이션이 어떤 로컬 어드레스를 할당 할 것인지 고려하지 않고 사용된다면, *name* 매개변수의 sa_data 멤버를 위해 INADDR_ANY 라는 상수값을 명시해야 합니다. 이 상수값을 사용하면, 적절한 네트웍 주소를 사용하기 위해 기본이되는 서비스 프로바이더를 적용하게 됩니다. 이 상수값은 한 개 이상의 네트웍 인터페이스나 주소가 있는 호스트일 경우 어플리케이션 프로그래밍을 간략히 하기 위해서 사용됩니다. 대부분의 프로그램은 인터페이스 어드레스에 대해 INADDR_ANY를 사용합니다.
- TCP/IP에서 포트번호가 0으로 명시된 경우에는 서비스 프로바이더가 어플리케이션을 위한 고유한 포트번호 (1024~5000 사이의 값)를 할당합니다. 어플리케이션은 주소와 포트를 알기위해 bind 된 후에 getsockname 함수를 사용할 수 있습니다. 만약 인터넷 주소가 INADDR_ANY 로 사용되어 있다면, getsockname 함수는 접속되기 전까지는 사용하지 못합니다. (몇몇 주소들은 호스트가 multihome 일 경우 올바를 수도 있기 때문입니다) 0번 포트가 아닌 특정한 포트 번호를 바인딩하는 것은 클라이언트 어플리케이션을 무용지물이 되게 합니다. 왜냐하면, 다른 소켓에 이미 그 포트 번호를 사용하고 있을 경우 충돌 할 수 있는 위험이 있기 때문입니다. 따라서, 클라이언트 프로그램이 포트 0을 지정하는 반면, 서버 프로그램은 대개 자신의 포트 번호를 지정합니다.
- connect 함수를 호출하기 전에는 **bind** 함수와 같이 소켓과 어드레스를 묶는 함수가 사실상 필요치 않습니다. 소켓이 묶이기 전에 **connect** 함수가 호출될 경우 bind 함수가 INADDR_ANY 의 어드레스와 포트 번호 0을 이용하여 소켓에 대해 호출된 것처럼 소켓은 자동으로 묶이게 됩니다. 대부분의 클라이언트 어플리케이션은 로컬 인터페이스 어드레스나 특별한 포트 값을 지정할 필요가 없습니다. 따라서 **bind**함수를 호출하기 전에 connect 함수를 호출함으로써 이들 어플리케이션에 대한 단계를 줄일 수 있습니다. 일반적으로 서버 어플리케이션은 Well-known 포트와 묶인 소켓을 생성해야 합니다. 이를 위해 **bind** 함수는 대부분 서버에 대해 명확하게 호출되어야 합니다.

2. 소켓 관련 함수

☐ listen system call

- connection을 받아들이겠다는 의지를 나타내기 위하여 connection-oriented server가 사용
- socket과 bind system call 후와 accept system call전에 수행

☐ Syntax

- int **listen** (int sockfd, int backlog);

```
#include <sys/types.h>
#include <sys/socket.h>

int listen (int sockfd, int backlog);
```

1. 서버 측의 접속 대기 큐의 최대 연결 가능 수 설정
2. 입력 인자
   1. sockfd : socket()함수 호출로 획득한 소켓의 소켓 기술자 입력
   2. backlog : 접속 대기 큐의 최대 연결 가능 수를 지정(TCP 서버에서만 사용)

2. 소켓 관련 함수

- accept system call
  - connection-oriented server가 listen system call을 수행한 후, server가 accept system call을 수행함으로써 어떤 client process로 부터 실제적 connection을 기다린다.

- Syntax

int **accept** (int sockfd, struct sockaddr * cliaddr, socklen_t * addrlen);

```
#include <sys/types.h>
#include <sys/socket.h>

int accept (int sockfd, struct sockaddr *peer, int *addrlen);
```

1. TCP서버에서 호출되고 연결이 완료된 큐의 최초에 위치한 데이터 반환
2. 큐가 비어있는 상태에는 클라이언트의 접속이 이루어질 때 까지 블록 됨
3. 정상적으로 함수가 종료되면 새로 연결된 파일 기술자를 반환
4. 입력 인자
   - sockfd : listen() 함수 호출로 설정한 소켓의 소켓 기술자 입력
   - cliaddr : 클라이언트의 IP 주소와 로컬 포트 정도를 설정한 구조체 입력
   - addrlen : ‘sockaddr’ 구조체의 크기 입력

2. 소켓 관련 함수

connect system call

- local system과 외부 system사이의 실제적인 connection을 설정

Syntax

int **connect** (int sockfd, const struct sockaddr * servaddr, socklen_t addrlen);

```
#include <sys/types.h>
#include <sys/socket.h>

int connect (int sockfd, struct sockaddr *servaddr, int addrlen);
```

1. 클라이언트에서 서버로의 연결을 담당
2. 소켓 기술자와 상대 서버에 대한 주소 정보를 바탕으로 서버로 접속 시도
3. 클라이언트에서 connect()호출 전에 bind()를 호출하지 않았다고 해도 임의의 포트와 로컬 IP를 자동으로 설정하여 함수 호출
4. 3-way handshaking으로 서버와 연결을 설정함
5. 입력 인자
   - sockfd : socket() 함수 호출로 저장된 소켓의 소켓 기술자 입력
   - servaddr : 접속할 서버의 주소와 포트에 관한 정보를 담은 구조체 입력
   - addrlen : ‘sockaddr’ 구조체의 크기 입력

2. 소켓 관련 함수

read / write / recv / send

1. ssize_t **read** (int sockfd, void * buffer, size_t len);
   - 원하는 소켓(sockfd)에서 특정 길이(len)만큼을 사용자 버퍼(buffer)로 읽어 들임

2. ssize_t **write** (int sockfd, void * buffer, size_t len);
   - 원하는 소켓(sockfd)에서 특정 길이(len)만큼을 사용자 버퍼(buffer)로 보냄

3. ssize_t **recv** (int sockfd, void * buffer, size_t len, int flags);
   - read 함수와 동일하나 flags 변수의 값에 따라 데이터 수신 방법이 다름
   - flags 옵션
     - MSG_PEEK : 네트워크 버퍼에서 데이터 제거를 하지 않고 버퍼에 복사만 함
     - MSG_OOB : 긴급 메시지 전달에 사용하며 TCP 헤더의 URG bit을 1로 변경
     - MSG_WAITALL : 완전한 양의 데이터를 수신 할 때만 recv 호출에서 복귀

4. ssize_t **send** (int sockfd, void * buffer, size_t len, int flags);
   - Write 함수와 동일하나 flags 변수의 값에 따라 데이터 송신 방법이 다름
   - flags 옵션
     - MSG_OOB : OOB Data 송신
     - MSG_DONTROUTE : 데이터를 라우팅 하지 않는다. 즉 라우팅 조건을 무시

Elementary Socket System call

□ send, sendto, recv, recvfrom system calls

```
#include <sys/types.h>
#include <sys/socket.h>

int send(int sockfd, char *buff, int nbytes, int flags);

int sendto(int sockfd, char *buff, int nbytes, int flags,
                struct sockaddr *to, int addrlen);

int recv(int sockfd, char *buff, int nbytes, int flags);

int recvfrom(int sockfd, char *buff, int nbytes, int flags,
                struct sockaddr *from, int *addrlen);
```

- 표준 read와 write 시스템 호출과 유사
- flags 독립변수는 0이거나 다음을 OR연산으로 형성한 값

MSG_OOB	send or recieve out-of-band data
MSG_PEEK	peek at incoming message(recv or recvfrom)
MSG_DONTROUTE	bypass routing (send or sendto)

2. 소켓 관련 함수

sendto / recvfrom

ssize_t **recvfrom** (int sockfd, void * message, size_t len, int flags,
struct sockaddr * send_addr, size_t * add_len);

ssize_t **sendto** (int sockfd, void * message, size_t len, int flags,
const struct sockaddr * dest_addr, size_t dest_len);

- 두 함수도 데이터 송수신을 담당하지만 recvfrom(수신)과 sendto(송신)함수는 UDP 통신에서 사용
- recv/send와의 차이점은 목적지에 대한 정보가 추가 된다는 것으로 이는 UDP의 특징인 비 연결성을 생각하면 송수신에 반드시 필요한 정보임을 알 수 있음

close / shutdown

int **close** (int sockfd);

- 소켓 종료 후에는 이전에 연결된 상대방에게 데이터 송신 불가
- 대기 열의 데이터에 대해서 해당 데이터 수신 작업 완료 후 소켓 종료

int **shutdown** (int sockfd, int howto);

- TCP 연결에 대한 종료를 담당하고 옵션에 따라 종료 방법 조절 가능
- close()함수와 다르게 별 다른 작업 없이 바로 종료 시킨다는 차이
- howto 인자에 따른 옵션
  - SHUT_RD : Read 불가, Write 가능
  - SHUT_WR : Read 가능, Write 불가
  - SHUT RDWR : Read / Write 모두 불가

3. 소켓 옵션 관련 함수

- int **getsockopt** (int sockfd, int level, int optname, void * optval, socklen_t * optlen);
  - 소켓(sockfd)의 지정된 옵션(optname)의 값(optval)과 바이트 수(optlen)을 반환

- int **setsockopt** (int sockfd, int level, int optname, void * optval, socklen_t optlen);
  - 소켓(sockfd)에 옵션(optname)을 지정하고 그에 따른 값(optval)과 바이트 수 (optlen)을 설정

- 소켓 및 프로토콜 종류와 그에 따른 옵션, 데이터 값으로 구성
  - level : 소켓 및 프로토콜의 종류 (IP, ICMP, IPv6, TCP 등)
  - optname : 세부 옵션
  - optval : 선택한 옵션에 대한 데이터 값
  - optlen : optval의 크기 입력

3. 소켓 옵션 관련 함수

2. 옵션의 주요 내용

Level	Option Name	Description	Type
SOL_SOCKET	SO_BROADCAST	브로드캐스트 메시지의 전송여부 설정	int
	SO_DONTROUTE	패킷이 프로토콜의 경로 배정을 피하게 설정	int
	SO_ERROR	에러에 대한 획득 설정	int
	SO_KEEPALIVE	주기적으로 연결이 유효한지의 테스트설정	int
	SO_RCVBUF	수신 버퍼 사이즈 설정	int
	SO_SNDBUF	송신 버퍼 사이즈 설정	int
	SO_RCVTIMEO	소켓 수신에 대한 타임아웃 설정	timeval
	SO_SNDTIMEO	소켓 송신에 대한 타임아웃 설정	timeval
IPPROTO_IP	IP_TTL	시스템이 사용하는 기본 TTL설정 획득	int
	IP_MULTICAST_TTL	출발TTL 명세	u_char
	IP_MULTICAST_LOOP	루프백에 대한 명세	u_char
	IP_ADD_MEMBERSHIP	멀티캐스트 그룹 참가	ip_mreq
	IP_DROP_MEMBERSHIP	멀티캐스트 그룹 탈퇴	ip_mreq
IPPROTO_TCP	TCP_KEEPALIVE	최초 질의전의 연결에서의 휴면시간규정	int
	TCP_MAXSEG	TCP 최대 세그먼트크기 획득 및 설정	int
	TCP_STDURG	Urgent 포인터에 대한 해석	int

4. 동시 처리를 위한 함수

1. signal() 함수

1. 시스템에 정의된 이벤트 발생 시 사용자가 정의한 처리를 하는 함수
2. 형태
   - void **(*signal)**(int sig, void (*func)(int))(int);
3. 인자
   - func : 시그널이 전달 될 때 호출되는 함수의 주소 값 입력
   - sig : 처리하거나 무시할 시그널을 설정(요약)

Signal	#	*Description*	
SIGHUP	1	연결 끊기	
SIGINT	2	터미널 인터럽트(Ctrl + C)	
SIGQUIT	3	터미널 종료	
SIGILL	4	잘못된 명령	
SIGKILL	9	프로세스 죽이기, 이 시그널은 잡히지 않음	
SIGALRM	14	경고 클럭	
SIGTERM	15	Kill 시그널 발생 전에 전송 되므로 종료상황을 추적 할 수 있다.	
SIGSTOP	19	정지 시그널, 이 시그널은 잡히지 않음	
SIGURG	23	소켓에서 URG 플래그 세팅 시 발생	
SIGIO	29	기술자에서 입출력이 가능함	fcntl() 함수 참고
SIGPWR	30	전원 실패	
SIGSYS	31	사용 안함	

signal() 사용 방법

- 유닉스에서 어떤 이벤트(event)가 발생하면 이것을 프로세스에게 알리는 수단으로 사용

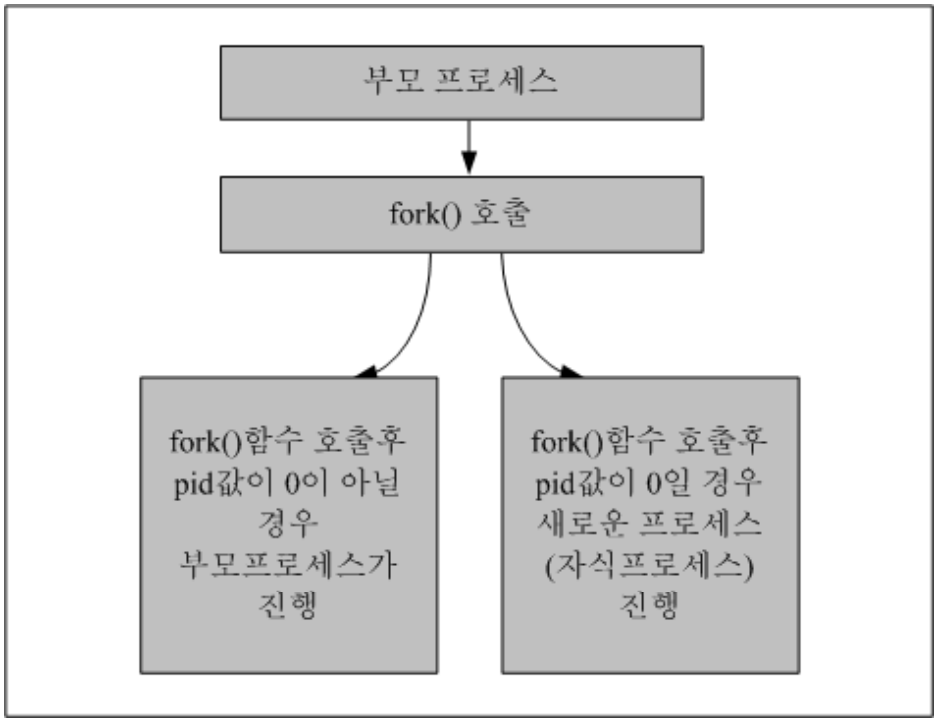
```
#include <signal.h>
main() {
   ….
   signal(SIGINT, sigint_func); /* 시그널 처리 함수 지정 */
   ….
}

int sigint_func() {
   /* SIGINT 시그널 발생시 처리 내용 */
}
```

4. 동시 처리를 위한 함수

2. fork() 함수

1. 새로운 프로세스를 생성하고 복제된 프로세스는 현재 프로세스와 같은 속성과 코드를 소유
2. 부모 프로세스는 자식 프로세스에 대한 개수에 대한 제한이 있으므로 에러 발생에 대한 제어 필요
3. 형태
   - pid_t **fork** (void);
4. 동작
5. 특징

- 코드, 스택, ,파일기술자, 소켓 번호 등을 공유
- PID와 변수는 공유하지 않음



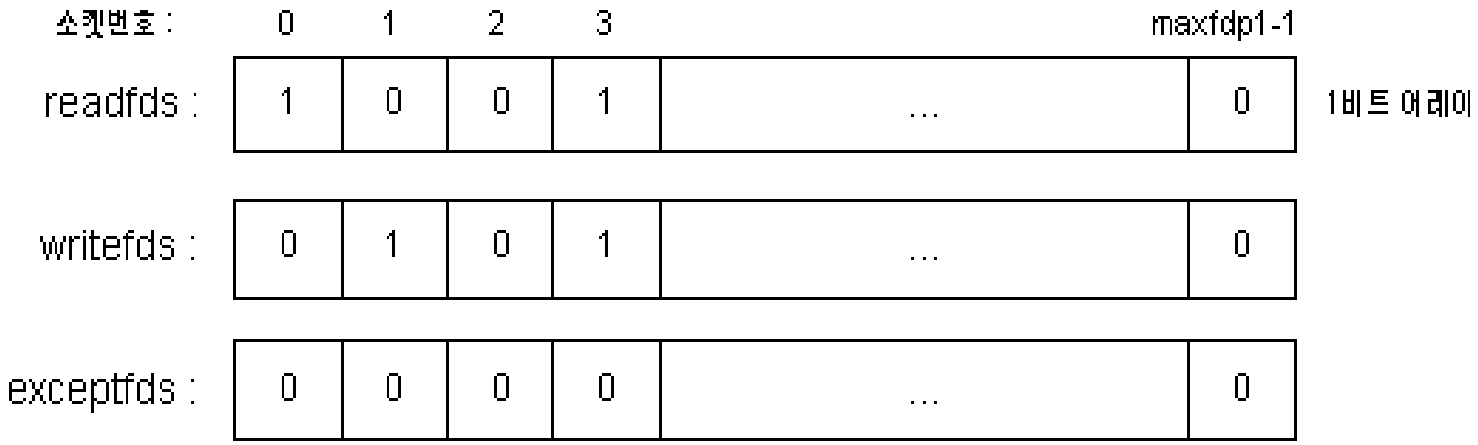
4. 동시 처리를 위한 함수

3. select () 함수
   1. 파일 기술자의 상태 변화 감지 후 해당 상태에 대한 처리
   2. 상태 변화는 읽기/쓰기/예외처리 의 3가지
   3. 형태
      - int **select** (int n, fd_set * readfds, fd_set * writefds, fd_set * exceptfds, const struct timeval * timeout);
   4. 입력 인자
      - n : 상태를 살펴 볼 최대 기술자 크기
      - timeout : 상태를 살펴 볼 주기 설정(0인 경우 무한 반복)
      - readfds / writefds / exceptfds : 읽기/쓰기/예외처리 관련 기술자
   5. 상태 기술자 관리를 위한 매크로들

void	FD_ZERO	*(fd_set * fdset);*	파일 기술자 집합을 초기화 한다.
void	FD_CLR	*(int fd, fd_set * fdset);*	원하는 파일 기술자만 초기화 한다.
void	FD_SET	*(int fd, fd_set * fdset);*	원하는 파일 기술자만 세팅한다.
void	FD_ISSET	*(int fd, fd_set * fdset);*	지정한 파일 기술자가 세팅되어 있는지 확인.

select() 시스템 콜

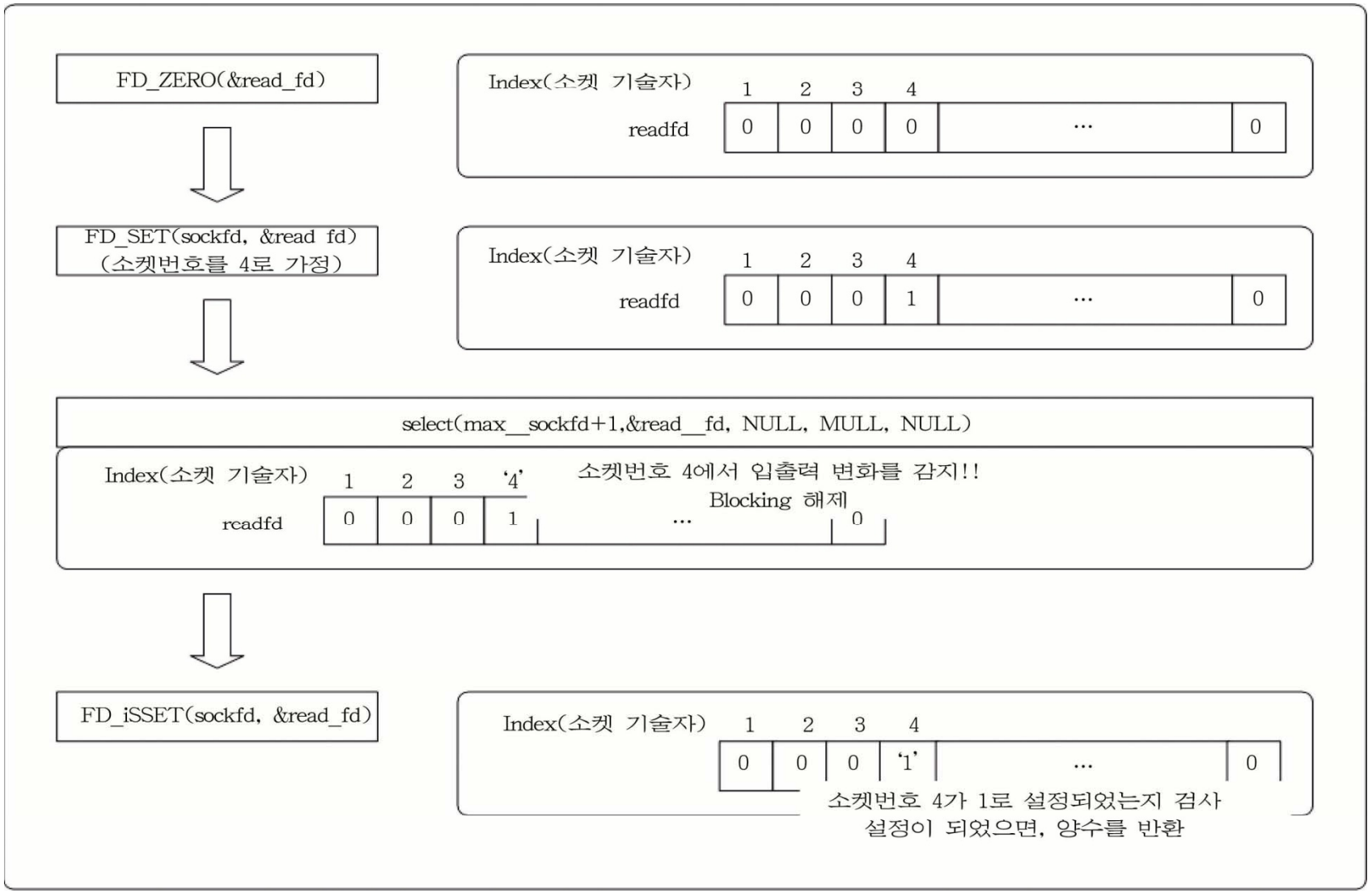
```
int select (
  int maxfdp1,        /* 최대 소켓 번호 크기 + 1 */
  fd_set *readfds,    /* 읽기 변화를 감지할 소켓번호*/
  fd_set *writefds,   /* 쓰기 변화를 감지할 소켓번호*/
  fd_set *exceptfds, /* 예외 변화를 감지할 소켓번호*/
  struct timeval *tvptr);
                /* select 시스템 콜이 기다리는 시간 */
```



select() 시스템 콜 (Cont'd)

- 매크로
  - FD_ZERO(fd_set *fdset)
    - fdset의 모든 비트를 지운다
  - FD_SET(int fd, fd_set *fdset)
    - fdset 중 소켓 fd에 해당하는 비트를 1로 한다
  - FD_CLR(int fd, fd_set *fdset)
    - fdset 중 소켓 fd에 해당하는 비트를 0으로 한다
  - FD_ISSET(int fd, fd_set *fdset)
    - fdset중 소켓 fd에 해당하는 비트가 세트되어 있으면 양수값을 리턴

Select를 이용한 예



4. 동시 처리를 위한 함수

4. poll() 함수
   1. select 함수와 같은 기술자 관리 방식

   2. select 에 비하여 관리할 기술자를 선택 할 수 있음

   3. select 함수 보다 빠른 응답 속도를 보이나 사용이 불편한 단점

   4. 형태
      - Int **poll** (struct pollfd *ufds, unsigned int nfds, int timeout);

   5. 입력 인자

Argument	*Description*
*ufds	파일 기술자와 감시하고자 하는 이벤트를 설정한 배열의 주소이다.
nfds	감시를 원하는 기술자의 개수를 입력한다.
timeout	밀리 초 단위의 대기 시간으로 음수 값은 무한정 대기의 의미를 지닌다.

4. 동시 처리를 위한 함수

5. Thread
   1. 스레드는 프로세스와는 다르게 메모리 공유 가능
   2. 메모리 공유로 인하여 동기화 문제 발생 (관련 도서 참고)
   3. 사용 함수
      - Int **pthread_create** (pthread_t * thread, pthread_attr_t * attr, void * (* start_routine)(void *), void * arg);

Argument	*Description*
thread	pthread_t에 대한 포인터. 쓰레드가 생성되면 쓰레드의 식별자가 입력된다.
attr	쓰레드의 속성을 입력한다. 정의 하지 않을 경우 NULL 값을 사용하며, 기본설정대로 새로운 쓰레드가 첨가 가능(Joinable)하고 비실시간(non Real-time) 스케줄 정책으로 생성된다.
(*start_routine) (void*)	쓰레드의 생성은 새로운 작업의 생성을 의미한다. 원하는 작업을 하는 함수를 가리키는 함수 포인터를 입력한다. 뒤의 'void *' 인자 값은 생성 함수의 4번째 인자 값에서 설정 가능하다.
arg	쓰레드가 작업 할 함수에서 사용할 인자로 본 함수 호출 시 넘겨주는 값이다.

      - void **pthread_join** (pthread_t th, void ** thread_return);
        - fork 에서 자식 프로세스를 기다리는 wait 함수와 같은 기능
      - void **pthread_exit** (void *retval);
        - 스레드를 종료 시킬 때 호출

Elementary Socket System call(*예제*)

- concurrent server로 가정할 경우의 전형적인 시나리오

```
int sockfd, newsockfd;

if ( (sockfd = socket(...)) < 0)  err_sys("socket error");
if (bind(sockfd, ...) < 0)        err_sys("bind error");
if (listen(sockfd, 5) < <0)       err_sys("listen error");

for ( ; ; ) {
        newsockfd = accept(sockfd, ...);        /* blocks */
        if (newsockfd < 0)        err_sys("accept error");

        if (fork() == 0) {
                close(sockfd);  /* child */
                doit(newsockfd);        /* process the request */
                exit(0);
        }

        close(newsockfd);               /* parent */
}
```

Elementary Socket System call(예제)

- iterative server로 가정할 경우의 전형적인 시나리오

```
int sockfd, newsockfd;

if ( (sockfd = soket(...)) < 0) err_sys("socket error");
if (bind(sockfd, ...) < 0)      err_sys("bind error");
if (listen(sockfd, 5) < <0)     err_sys("listen error");

for ( ; ; ) {
        newsockfd = accept(sockfd, ...);    /* blocks */

        if (newsockfd < 0)
                err_sys("accept error");

        doit(newsockfd);        /* process the request */
        close(newsockfd);
}
```

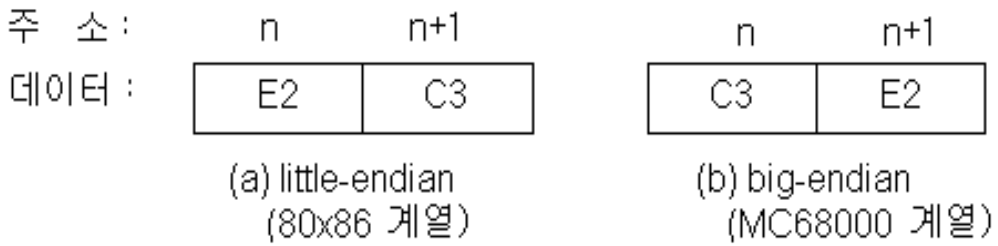
유닉스 소켓 프로그래밍을 위한 유틸리티 함수들

1. 바이트 순서 변환 함수
2. 주소 변환 구조체
3. 인터넷 주소 변환 함수
4. 원격지 호스트 정보를 얻는 함수

1. 바이트 순서 변환 함수

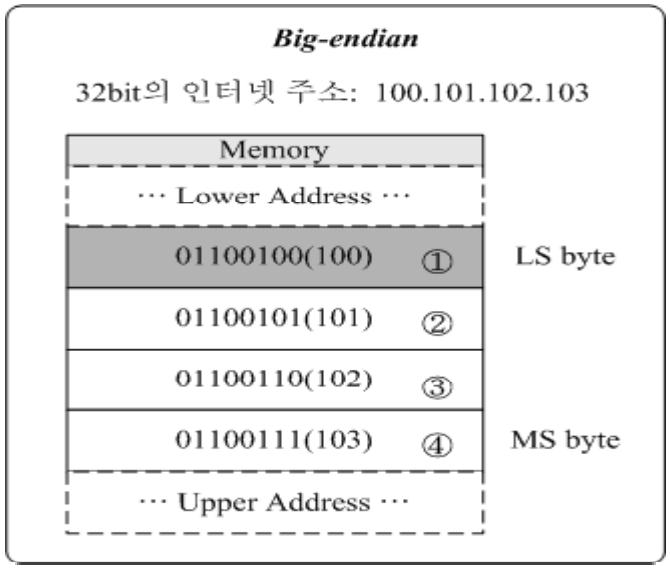
- 호스트 바이트 순서
  - CPU에 따라 메모리에 저장하는 순서가 틀림
- 네트워크 바이트 순서
  - CPU에 관계 없이 High-order 순서로 전송
  - Internet protocol을 위하여 설계
- htonl(), htons(), ntohl(), ntohs()
  - htonl convert host-to-network, long integer
  - htons convert host-to-network, short integer
  - ntohl convert network-to-host, long integer
  - ntohs convert network-to-host, short integer

```
#include <sys/types.h>
#include <netinet/in.h>

u_long      htol(u_long hostlong);
u_short     htons(u_short hostshort);
u_long      ntohl(u_long netlong);
u_short     ntohs(u_short netshort);
```



바이트 순서 (예)

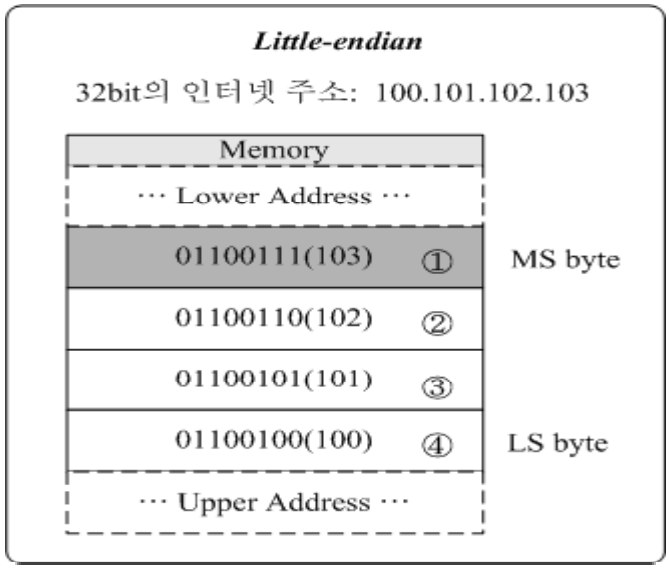
Big-Endian



- 데이터의 최상위 바이트의 내용을 저장될 메모리의 시작주소에 저장

- 모토로라 마이크로프로세서 및 **IBM** 메인 프레임 기반

Little-Endian



- 데이터의 최하위 바이트의 내용을 저장될 메모리의 시작주소에 저장

- DEC VAX 컴퓨터 및 인텔 마이크로프로세서

바이트 순서 (예제 프로그램)

- byte_order.c 예제

```
…
pmyservent = getservbyname("echo", "udp");
printf("Port number of 'echo', 'udp' without
    ntohs() : %d \n", pmyservent->s_port);
printf("Port number of 'echo', 'udp' with
    ntohs() : %d \n", ntohs(pmyservent->s_port));
…
```

- 실행 결과 (x86 Linux System)

```
Port number of 'echo', 'udp' without ntohs() : 1792
Port number of 'echo', 'udp' with ntohs() : 7
```

- 실행 결과 (mc68000계열)

```
Port number of 'echo', 'udp' without ntohs() : 7
Port number of 'echo', 'udp' with ntohs() : 7
```

2. 주소 변환 구조체

- 기본적인 IP 주소 체계를 도메인 주소로 변환 시 사용

A. hostent 구조체

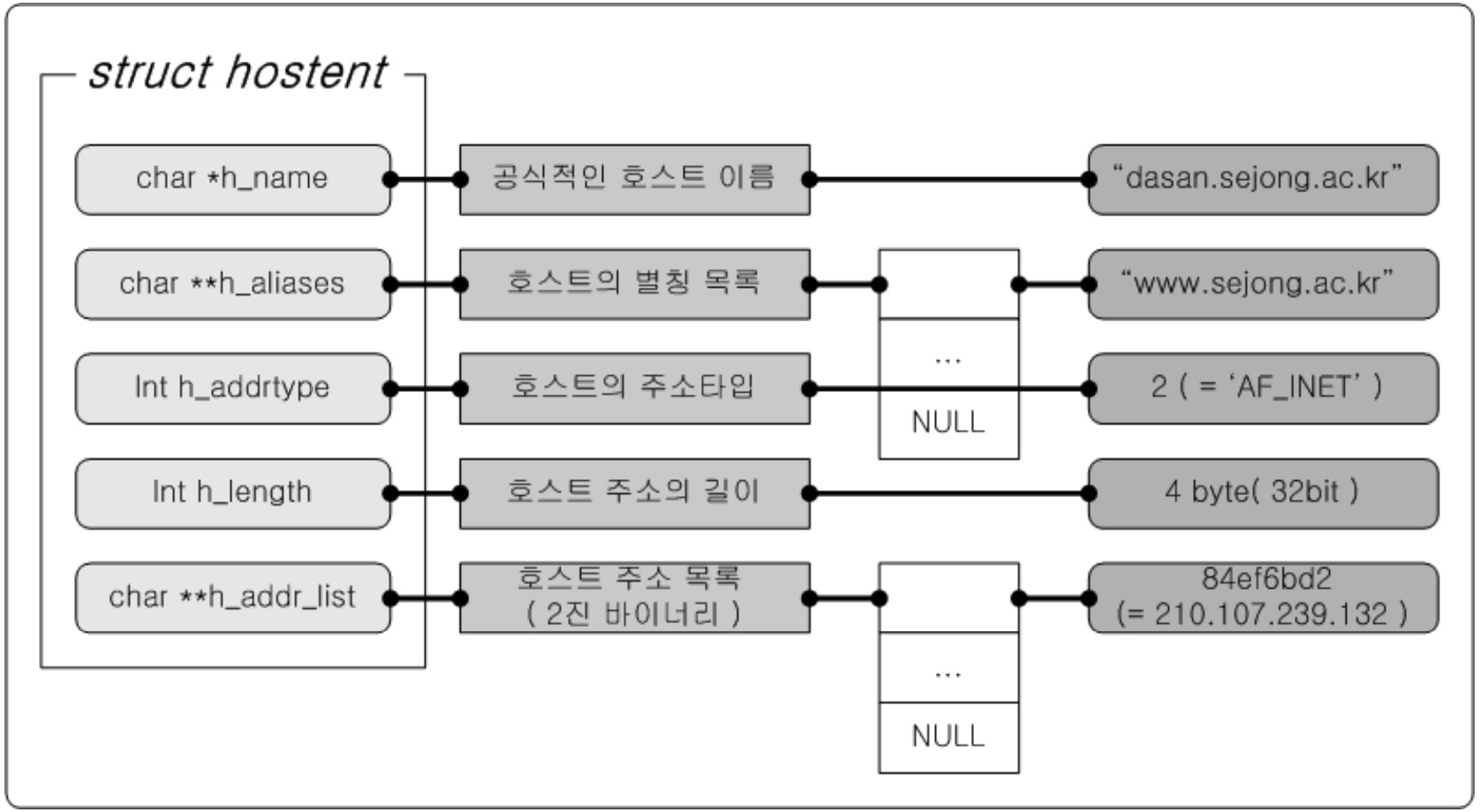
- **gethostbyaddr(), gethostbyname()** 함수에 사용
- 변수 : 호스트의 공식 명칭, 별칭, 호스트 주소의 종류, 주소 체계의 바이트 수, 도메인에 포함된 **IP list**

B. servent 구조체

- **getservbyname(), getservbyport()** 함수에 사용
- 변수 : 서버의 공식 명칭, 별칭, 서버가 제공하는 포트 번호, 서버가 사용하는 프로토콜 타입의 문자열

원격지 호스트 정보를 위한 구조체 (1)

* hostent 구조체

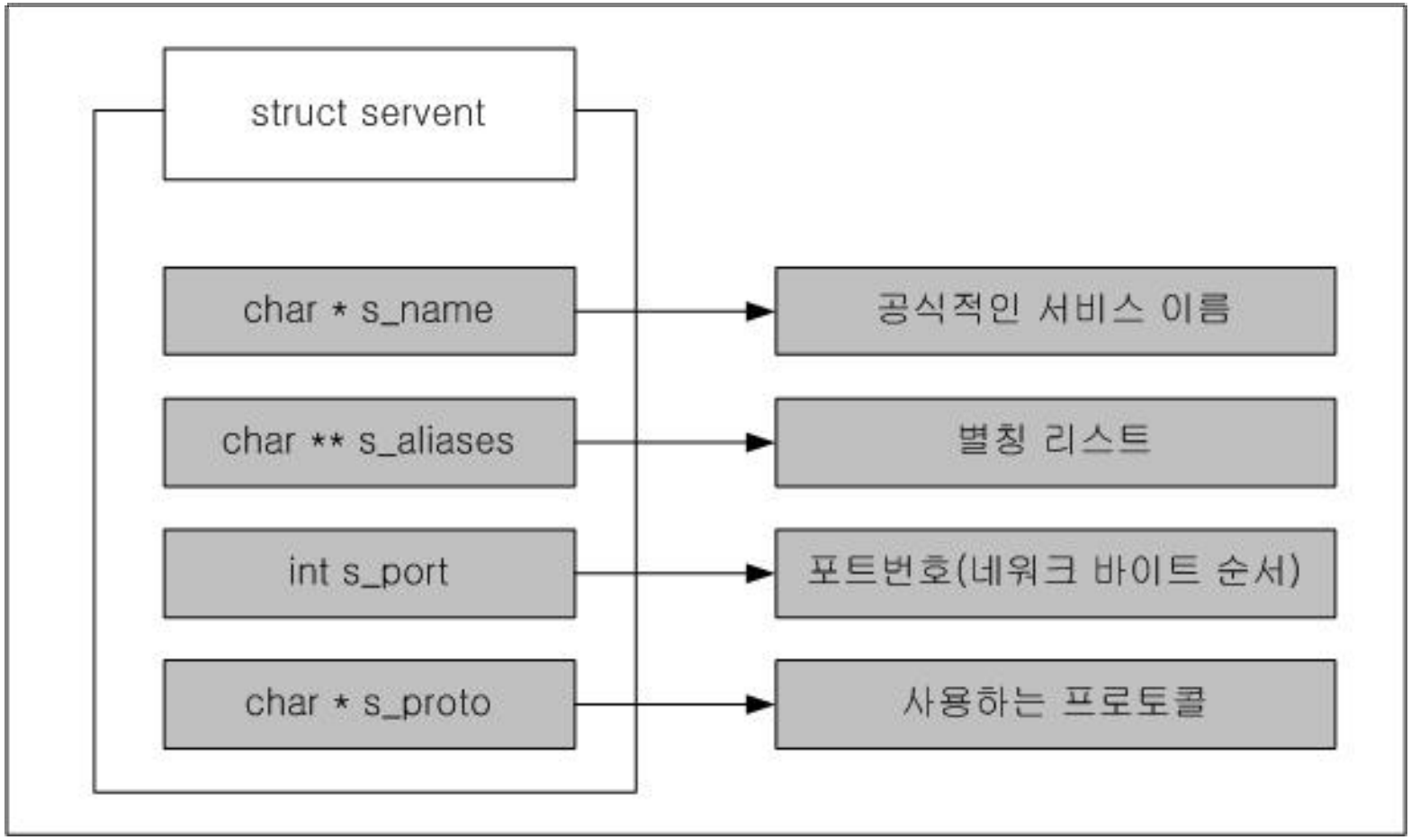


- hostent 구조체 정의

```
struct hostent {
    char  *h_name;              /* 호스트 이름 */
    char  **h_aliases;          /* 호스트 별명들 */
    int    h_addrtype;          /* 호스트 주소의 종류 */
    int    h_length;            /* 주소의 크기 */
    char  **h_addr_list;        /* IP 주소 리스트 */
};
#define h_addr haddr_list[0]  /* 첫번째 주소 */
```

원격지 호스트 정보를 위한 구조체 (2)

* servtent 구조체



getservbyname()

- 시스템이 현재 지원하는 TCP/IP 응용 프로그램의 정보를 알아내는 함수로, 서비스 이름과 프로토콜을 인자로 주어 호출하면 서비스 관련 각종 정보를 포함하고 있는 servent라는 구조체의 포인터를 리턴한다.

  - pmyservent = getservbyname("echo", "udp");
  - servent 구조체의 정의(netdb.h 파일 참조).

```
struct  servent {
        char *s_name;       /* 서비스 이름 */
        char **s_aliases;     /* 별명 목록 */
        int   s_port;         /* 포트 번호 */
        char *s_proto;       /* 사용하는 프로토콜 */
};
```

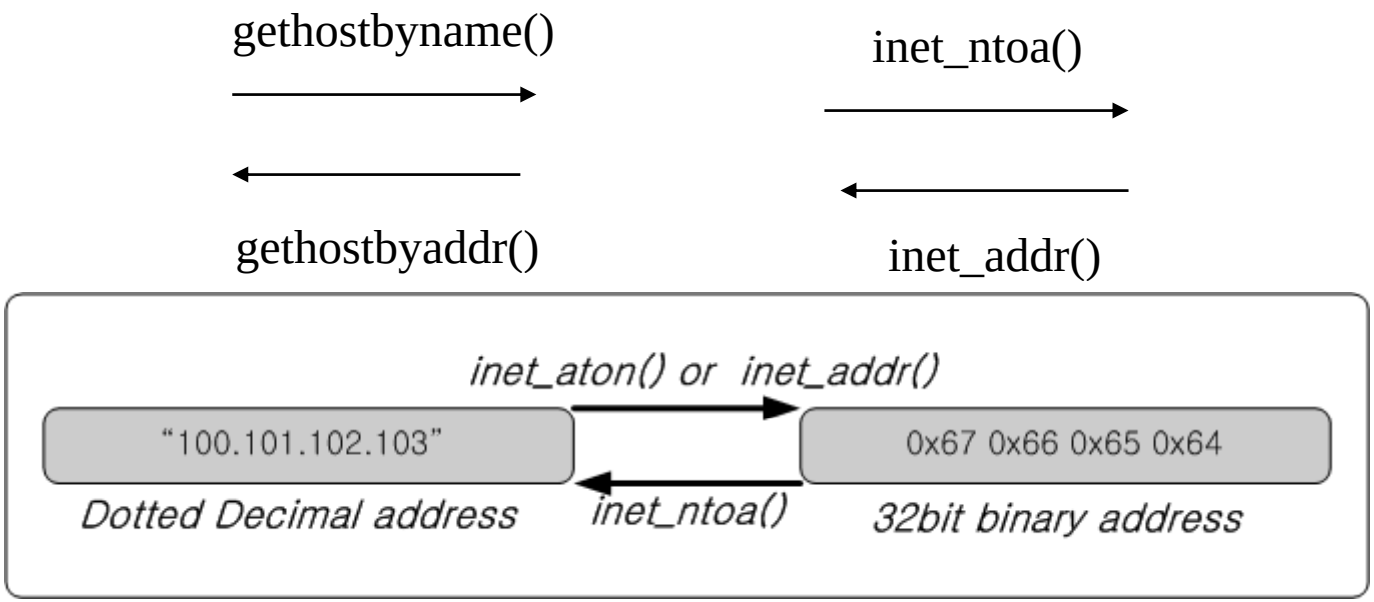
- servent 구조체는 네트워크로부터 얻은 정보이므로 그 내용이 네트워크 바이트 순서로 되어 있고 이를 화면에 출력해 보려면 호스트 바이트 순서로 바꾸어야 한다.

3. 인터넷 주소 (IP주소) 변환

- 인터넷 주소를 표현하는 방식
  - 도메인 네임
  - 32bit IP 주소
  - 십진수 표시법 (dotted decimal)

- IP 주소 변환 함수 예
  - mm.sookmyung.ac.kr : : 203.252.201.16

Domain Name : IP 주소 (binary) : dotted decimal



주소변환 함수(Address Conversion Routines)-1

unsigned long int **inet_addr** (const char * strptr);

- ‘Dotted Decimal' 문자열 형태의 주소를 네트워크 바이트 순서를 갖는 이진 바이너리로 바꾸는 함수
- 점으로 분리된 십진수 개념의 문자열을 32-비트 Internet 주소로 변환
- 입력 값이 적절치 못할 경우 INADDR_NONE('-1')을 반환
- ’-1‘은 ’Dotted Decimal’ 로 표현 시 ‘255.255.255.255’ 이므로 유효성 검사가 반드시 필요

Address Conversion Routines (2)

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

- 'Dotted Decimal' 형태의 문자열을 바이너리 형태로 바꾸는 함수
- ‘strptr’ 로 들어온 ‘Dotted Decimal' 형태의 문자열은 변환되어 'in_addr' 타입의 ’addrptr‘ 에 저장됨
- 주소가 올바르다면 ‘0’ 이 아닌 값, 그렇지 않다면 ‘0’ 을 반환

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

- 'in_addr' 형의 바이너리 주소 ‘inaddr’을 ‘Dotted Decimal' 형태의 문자열로 변경
- 반환되는 문자열은 정적으로 할당된 버퍼에 저장
- 함수의 연속적인 호출은 버퍼를 중복하여 덮어 쓰므로 주의

```
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>

unsigned long inet_addr(char *ptr);
char *inet_ntoa(struct in_addr inaddr);
```

-inet_addr <-> inet_ntoa : 역변환

IP 주소 변환 (예제)

- ascii_ip.c 예제

```
…
struct in_addr host_ip;
haddr = argv[1];              /* dotted decimal 주소 */
host_ip.s_addr = inet_addr(haddr);

/* IP 주소 (hexa 4byte) 출력 */
printf("IP Address (hexa) 0x%x\n", host_ip.s_addr);

/* dotted decimal 로 다시 변환하여 출력 */
printf("%s\n", inet_ntoa(host_ip));
…
```

IP 주소 변환 (예제)

- get_hostent.c 예제

```
…
struct in_addr host_ip;
haddr = argv[1];  /* dotted decimal 주소 */

host_ip.s_addr = inet_addr(haddr);

/* IP 주소 (hexa 4byte) 출력 */
printf("IP Address (hexa) 0x%x\n", host_ip.s_addr);

/* dotted decimal 로 다시 변환하여 출력 */
printf("%s\n", inet_ntoa(host_ip));
```

- get_hostent.c 실행 결과

```
#> get_hostent mm.sookmyung.ac.kr
official host name : mm.sookmyung.ac.kr
host address type :          2
length of host address :     4
IP address :                 203.252.201.16
```

- dotted decimal => 도메인 네임
  - gethostbyaddr() 이용

4. 원격지 호스트 정보를 얻는 함수

1. struct hostent * **gethostbyname** (const char * hostname);
   - 전달 값으로 호스트의 이름(도메인)을 받아서 hostent 구조체에 결과 값을 돌려주는 함수

2. struct hostent * **gethostbyaddr** (const char * addr, size_t len, int family);
   - 호스트의 IP주소(바이너리 형태의 주소)를 이용하여 해당 호스트에 대한 정보를 저장
   - addr은 호스트 IP 주소이고 len 은 IP 주소의 크기
     (IPv4 = 4, IPv6 = 16)

3. int **gethostname** (char * name, size_t namelen);
   - 현재의 호스트 이름을 반환
   - 전달 인자 name은 호스트의 이름을 저장할 곳의 주소
   - namelen 은 name의 바이트 길이
   - 성공한 경우 반환 값 0, 실패인 경우 반환 값 -1

4. 원격지 호스트 정보를 얻는 함수

4. struct servent * **getservbyname** (const char * servname,
   const char * protoname);

   - 해당 호스트에서 진행되고 있는 서비스에 대한 각 정보를 서비스에 대한 이름과 해당 프로토콜로 얻을 수 있게 해주는 함수

   - 수행 중 에러 발생 시에는 결과 값으로 NULL 반환

5. struct servent * **getservbyport** (int port, const char * protoname);

   - 해당 호스트에서 진행되고 있는 서비스에 대한 각 정보를 포트 번호로 얻을 수 있게 해주는 함수

   - 수행 중 에러 발생 시에는 결과 값으로 NULL 반환

클라이언트/서버의 작성 방법

1. 서버의 유형
2. 연결형 클라이언트/서버의 작성
3. 비연결형 클라이언트/서버의 작성

서버 프로그램 작성 절차

- 서버 프로그램이 먼저 수행되고 있어야 한다.
- 서버는 socket()을 호출하여 통신에 사용할 소켓을 개설한다.
- 소켓번호와 자신의 소켓주소를 bind()로 서로 연결한다.
  - *소켓주소(자신의 IP 주소 자신의 포트번호)*

- 소켓번호는 응용 프로그램이 알고 있는 통신 창구 번호이고, 소켓 주소는 네트워크 시스템(즉, TCP/IP)이 알고 있는 주소이므로 이들의 관계를 묶어 두어야 한다.
- listen()을 호출하여 클라이언트로부터의 연결요청을 기다리는 수동 대기모드로 들어간다.
- 클라이언트로부터 연결요청이 왔을 때 이를 처리하기 위하여 accept()를 호출한다.
- 클라이언트가 connect()를 호출하여 연결요청을 해오면 이를 처리한다.
- 메시지를 송수신한다.

클라이언트 프로그램 작성 절차

- socket()을 호출하여 소켓을 만든다.
- 서버에게 연결요청을 보내기 위하여 connect()를 호출한다. 이 때 서버의 소켓주소(④+⑤) 구조체를 만들어 인자로 준다.
  - *소켓주소(상대방의 IP 주소 상대방의 포트번호)*
  - *상대방 == 서버의 IP주소 및 포트번호*

- 대부분의 클라이언트는 bind()를 사용하여 포트 번호를 특정한 값으로 지정할 필요가 없다(클라이언트는 보통 시스템이 자동으로 배정하는 포트 번호를 사용한다).
- connect()를 호출하여 연결요청을 한다.
- 메시지를 송수신한다.

소켓의 동작 모드

- blocking 모드
  - 함수를 호출했을 경우 동작 완료때 까지 멈춤
  - listen(), connect(), accept(), read(), write() 등

- non-blocking 모드
  - 시스템 콜 호출 후 바로 값이 리턴
  - 계속 루프(loop)를 돌면서 확인(폴링)

- 비동기(asynchronous) 모드
  - IO 변화를 감지
  - select() 함수를 사용

1. 서버의 유형

1. 서버 유형의 결정
   1. 클라이언트 접속 빈도
      - 얼마 만큼의 클라이언트가 어느 정도의 빈도로 접속하는지에 따름

   2. 필요한 규모와 처리량
      - 주고 받는 데이터의 크기가 어느 정도이며 필요한 대역폭에 따름

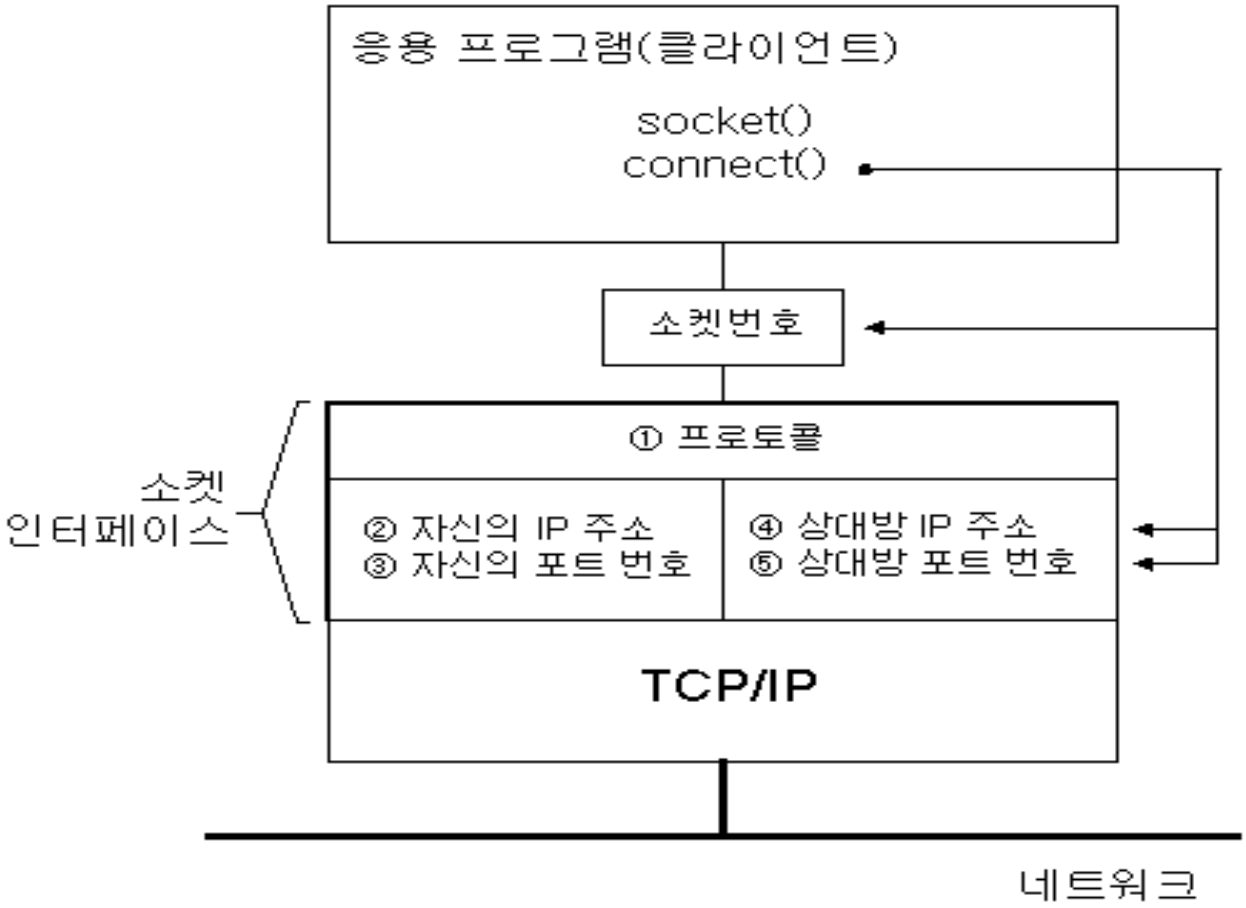
   3. 데이터의 안정성
      - 클라이언트 간 혹은 서버와 클라이언트 간의 데이터 신뢰도
      - 요즘의 네트워크 망은 설비가 잘 되어 있어 안정성에 대한 문제는 줄어 듬

2. 기본적인 서버의 유형

서버 타입	선택사항	
클라이언트의 요청에 대한 처리 방식	동시처리형	단순형
서버로의 접근 방식	연결형(TCP)	비연결형(UDP)

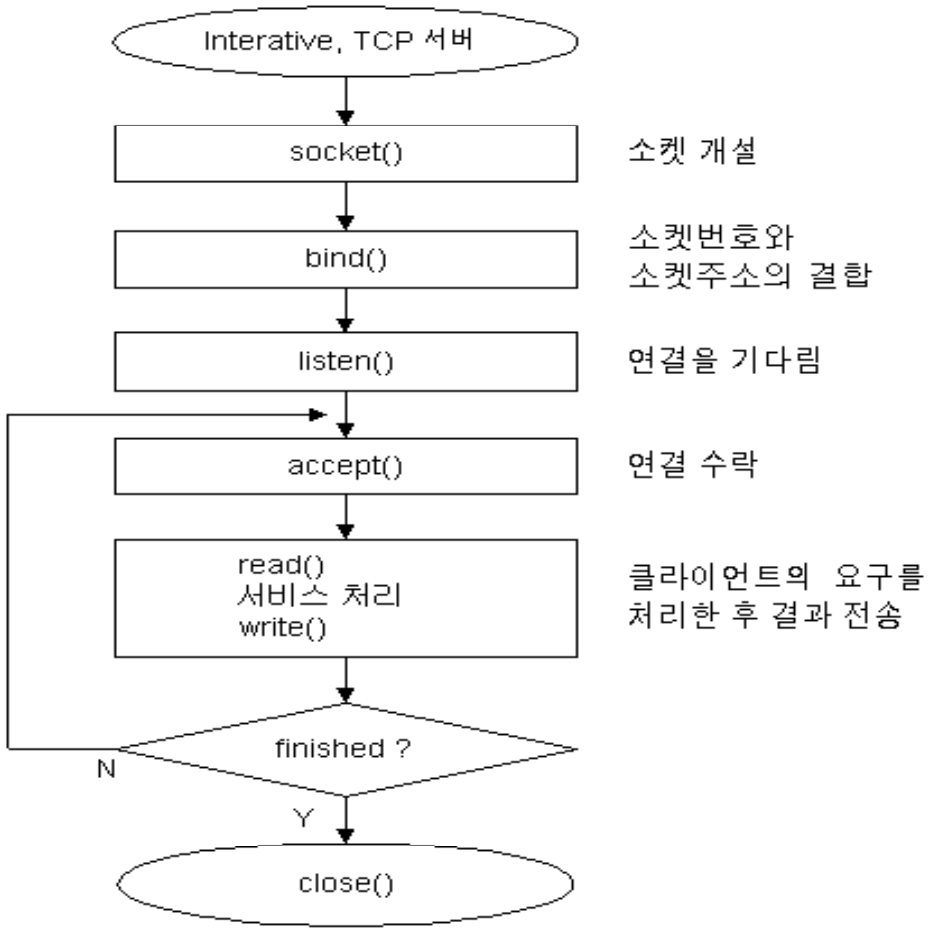
연결형 클라이언트

- socket() 호출 시 소켓 번호와 소켓 인터페이스의 관계



연결형 서버 P/G

- Iterative 모델의 TCP 서버 프로그램 작성절차



비연결형 클라이언트

- SOCK_DGRAM으로 소켓 생성
- connect()없이 바로 메시지 송수신

- 각 패킷마다 IP 주소와 포트 번호를 입력

```
sendto(int s, char* buf, int length,
        int flag, sockaddr* to, int tolen)
recvfrom(int s, char* buf, int length,
        int flags, sockaddr* from, int fromlen)
```

포트 번호 배정

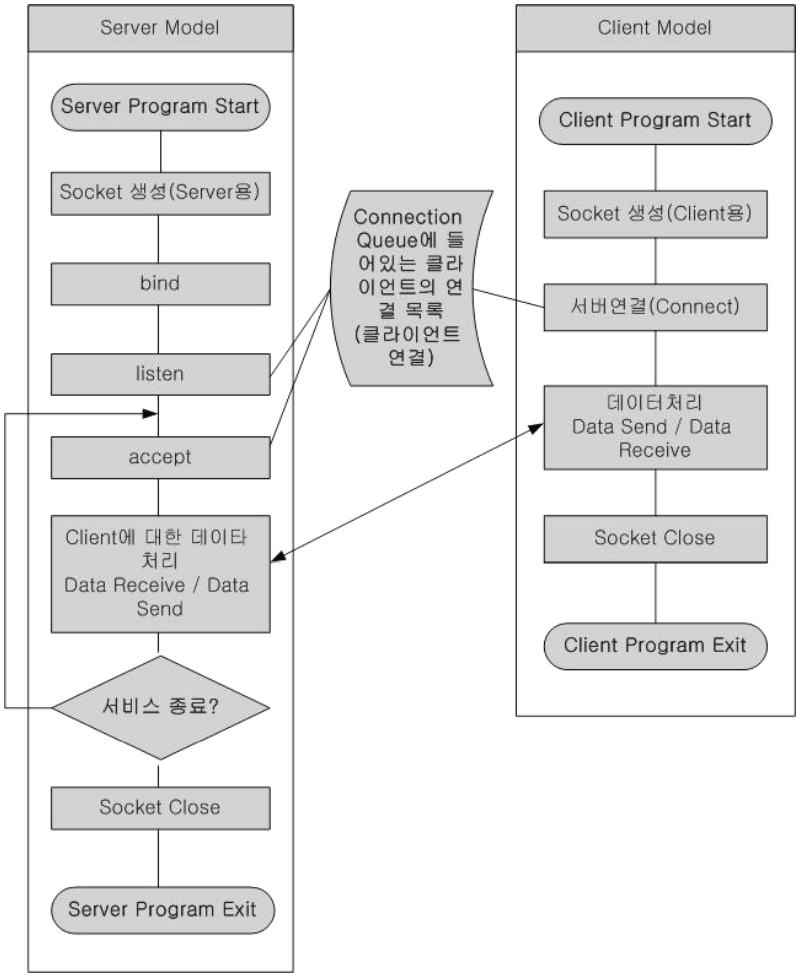
- 클라이언트 포트 번호 배정 시기
  - TCP 소켓의 경우 connect() 호출 이후에
  - UDP 소켓의 경우 sendto() 호출 이후에

- getsockname()을 이용 확인

```
getsockname(s, (struct sockaddr *)&clinet_addr,
                &addr_len);
port = clinet_addr.sin_port;
printf("client port number : %d\n", port);
```

2. 연결형 클라이언트/서버의 작성

1. 단순 연결형 클라이언트/서버



1) 서버와 클라이언트의 소켓 생성

2) 서버는 BIND 작업 후 대기 큐 설정(LISTEN)

3) 서버는 클라이언트의 접속 대기(ACCEPT)

4) 클라이언트가 서버로 접속 시도(CONNECT)

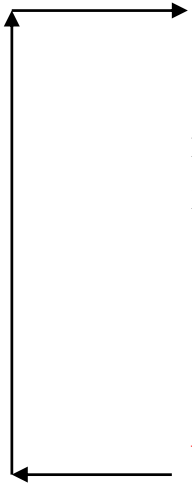
5) 접속이 완료 되면 서버와 클라이언트 간의 데이터 송수신 작업

6) 서버 종료 작업

7) 서버와 클라이언트 소켓 닫음(CLOSE)

(예) 단순 연결형 서버 socket 생성 및 수행

```
/*서버소켓생성*/
server_socket = socket(AF_INET, SOCK_STREAM, 0);
/*서버 주소 및 포트 설정*/
server_addr.sin_family = AF_INET;
server_addr.sin_port = htons(SERVER_PORT);
server_addr.sin_addr.s_addr = htonl(INADDR_ANY);
memset(&(server_addr.sin_zero), 0, 8);

/*해당 주소 및 포트로 바인딩작업*/
bind(server_socket, (struct sockaddr*)&server_addr, sizeof(struct sockaddr));
/*클라이언트의 대기열수 설정*/
listen(server_socket, 10);

client_socket = accept(server_socket, (struct sockaddr*)&server_addr, &addrsize);

msgsize = read(client_socket, BUFF, sizeof(BUFF));
if (msgsize <= 0)
{
        printf("message receive error\n");
        continue;
}
write(client_socket, BUFF, strlen(BUFF) );
```

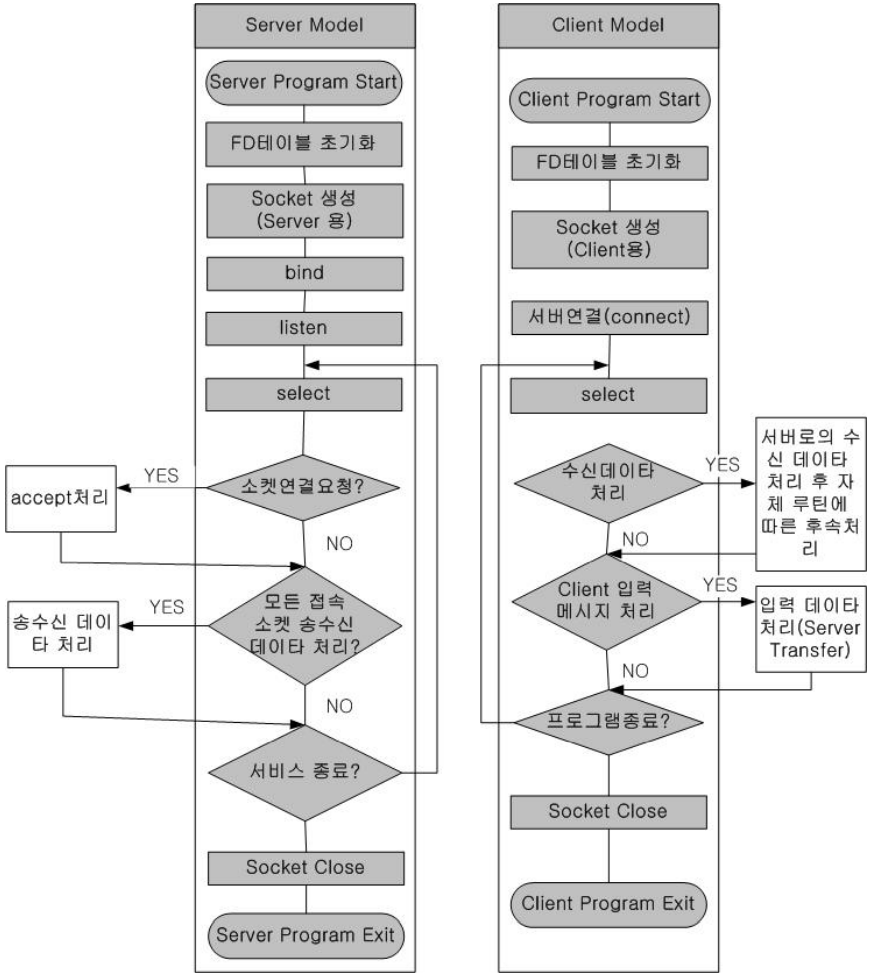


(예) 단순 Client socket 생성 및 수행

```
/*소켓 생성 TCP이므로 SOCK_STREAM*/
connect_fd = socket(AF_INET, SOCK_STREAM, 0);

/*사용자의 서버 IP및 포트에 대한 입력을 받아서 sockaddr_in구조체를 구성*/
connect_addr.sin_family = AF_INET;
connect_addr.sin_port = htons(atoi(argv[2]));
connect_addr.sin_addr.s_addr = inet_addr(argv[1]);
memset(&(connect_addr.sin_zero), 0, 8);

/*서버로의 소켓 접속*/
connect(connect_fd, (struct sockaddr*)&connect_addr, sizeof(connect_addr));

msgsize = read(connect_fd, msg, sizeof(msg));
                        if (msgsize <= 0)
                                continue;
                        printf("Server Message>>%s", msg);
memset(msg, '\0', sizeof(msg));
fgets(msg, 1024, stdin);
msgsize = strlen(msg);
write(connect_fd, msg, msgsize);
```

2. 연결형 클라이언트/서버의 작성

2. 다중 연결 연결형 클라이언트/서버



1) 서버와 클라이언트의 통신 초기화 (SELECT 관련)

2) 서버와 클라이언트 소켓의 생성

3) 서버는 BIND 작업 후 대기 큐 설정 (LISTEN)

4) 서버는 클라이언트의 접속 대기(ACCEPT)

5) 클라이언트가 서버로 접속 시도 (CONNECT)

6) 서버 측의 기술자에 이벤트가 발생

7) 접속 처리 후 클라이언트와 데이터 송수신

8) 서버 종료 작업

9) 서버와 클라이언트 소켓 닫음(CLOSE)

(예) 다중 연결형 서버 socket 생성 및 수행

```
/*서버소켓생성*/
server_socket = Socket(AF_INET, SOCK_STREAM, 0);

/*서버 주소 및 포트 설정*/
.....
/*해당 주소 및 포트로 바인딩작업*/
Bind(server_socket, (struct sockaddr*)&server_addr, sizeof(struct sockaddr));

/*클라이언트의 대기열수 설정*/
Listen(server_socket, 10);

/*fd_set 데이타 필드 초기화*/
FD_ZERO(&readfd);
maxfd = server_socket;
client_index = 0;

while(1)
{
        /*fd_set를 해당 소켓 기술자로 설정*/
        FD_SET(server_socket, &readfd);


        /*접속된 클라이언트의 소켓 기술자를 fd_set에 설정*/
        for (start_index = 0; start_index < client_index; start_index++)
        {
                FD_SET(client_fd[start_index], &readfd);
                if (client_fd[start_index] > maxfd)
                        maxfd = client_fd[start_index];
        }
        maxfd = maxfd + 1;

                return 0;
```

→ 다음 페이지 계속

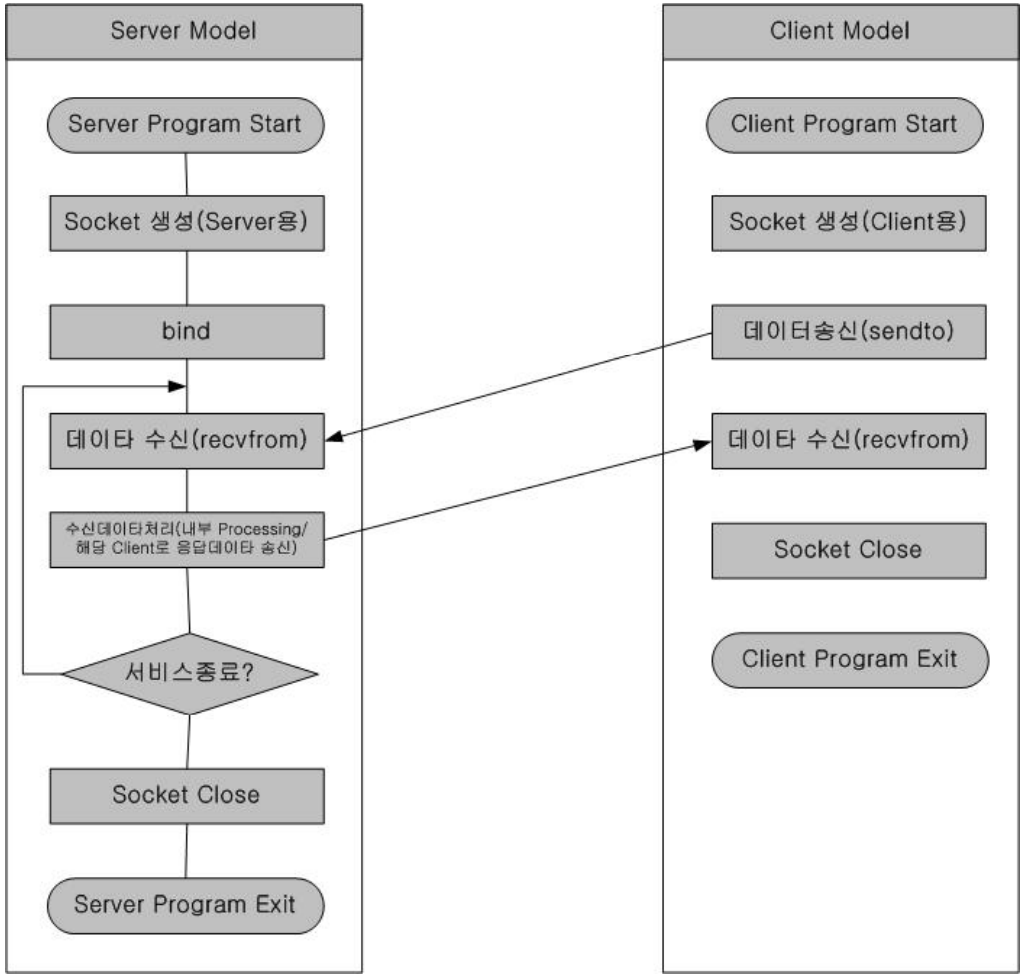
```
select(maxfd, &readfd, NULL, NULL,NULL);

/*해당 소켓기술자에 변화가 생겼는지 검사*/
if (FD_ISSET(server_socket, &readfd))
{
        addrsize = sizeof(struct sockaddr_in);
        client_socket = Accept(server_socket, (struct sockaddr*)&server_addr, &addrsize);
        FD_SET(client_socket, &readfd);

        client_fd[client_index] = client_socket;
        client_index++;

        if (client_index == 5)    break;
}
/* 해당 소켓 기술자에 변화가 생겼는지 검사
  서버에 접속된 모든 클라이언트의 소켓 기술자 검사 */
for (start_index = 0; start_index < client_index; start_index++)
{
        if (FD_ISSET(client_fd[start_index], &readfd))
        {
                msgsize = read(client_fd[start_index], BUFF, sizeof(BUFF));
                if (msgsize <= 0)  continue;
                printf("Client Message>>%s", BUFF);
                msgsize = strlen(BUFF);
                write(client_fd[start_index], BUFF, msgsize);
        }
}
```

(예) 다중 Client socket 생성 및 수행

```
/*서버로의 소켓 접속*/
Connect(connect_fd, (struct sockaddr*)&connect_addr, sizeof(connect_addr));

/*fd_set 데이타 필드 초기화*/
FD_ZERO(&readfd);

while(1)
{
        /*fd_set를 해당 소켓 기술자로 설정*/
        FD_SET(0, &readfd);
        FD_SET(connect_fd, &readfd);

        select (connect_fd+1, &readfd, NULL, NULL, NULL);
        /*해당 소켓 기술자에 변화가 생겼는지 검사*/
        if (FD_ISSET(connect_fd, &readfd))
        {
                msgsize = read(connect_fd, msg, sizeof(msg));
                if (msgsize <= 0) continue;
                printf("Server Message>>%s", msg);
        }
        /*해당 입력 기술자에 변화가 생겼는지 검사*/
        if (FD_ISSET(0, &readfd))
        {
                memset(msg, '\0', sizeof(msg));
                fgets(msg, 1024, stdin);
                msgsize = strlen(msg);
                write(connect_fd, msg, msgsize);
        }
}
```

3. 비연결형 클라이언트/서버의 작성

1. 단순 비 연결형 클라이언트/서버



1) 서버와 클라이언트의 소켓 생성

2) 서버는 BIND 작업 후 대기

3) 서버와 클라이언트 간의 데이터 송수신

4) 서버는 단순히 클라이언트로 데이터 전송

5) 서버 종료 작업

6) 서버 소켓 닫음

6) 클라이언트 소켓 닫음

* 양쪽 모두 상대방의 주소를 알고 있어야 지속적인 데이터 송수신이 가능함을 기억

(예) 단순 비연결형 서버

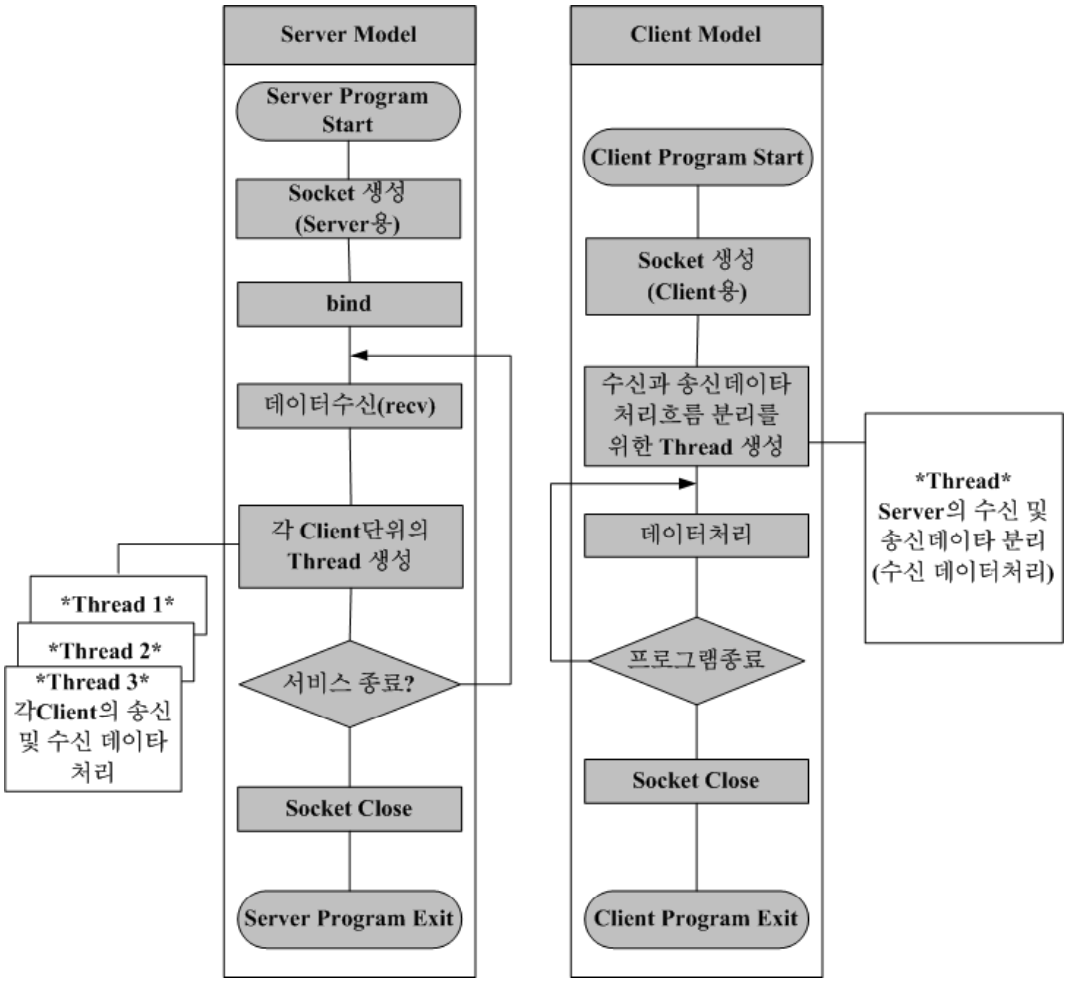
```
/*1. 서버소켓생성*/
server_socket = Socket(AF_INET, SOCK_DGRAM, 0);

server_addr.sin_family = AF_INET;
server_addr.sin_port = htons(SERVER_PORT);
server_addr.sin_addr.s_addr = htonl(INADDR_ANY);
memset(&(server_addr.sin_zero), 0, 8);

bind(server_socket, (struct sockaddr*)&server_addr, sizeof(struct sockaddr));

addrsize = sizeof(struct sockaddr_in);
while(1)
{
        memset(BUFF, '\0', sizeof(BUFF));

        msgsize = recvfrom(server_socket, BUFF, sizeof(BUFF), 0,
                        (struct sockaddr*)&client_addr, &addrsize);
        if (msgsize <= 0) continue;

        printf("Client Message>>%s", BUFF);
        msgsize = strlen(BUFF);
        sendto(server_socket, BUFF, msgsize, 0, (struct sockaddr*)&client_addr, addrsize);
}
```

(예) 단순 비연결형 클라이언트

```
memset(msg, '\0', sizeof(msg));
fgets(msg, 1024, stdin);
msgsize = strlen(msg);

/*사용자 입력 데이타에 대한 처리. 서버 전송* sendto함수 이용*/
msgsize = sendto(connect_fd, msg, msgsize, 0, (struct sockaddr*)&connect_addr, addrsize);
printf("Message Send>>%s", msg);

memset(temp_BUFF, '\0', sizeof(temp_BUFF));
/*서버에서의 송신 데이타에 대한 처리 recvfrom함수 이용*/
recvfrom((int)sockfd, temp_BUFF, sizeof(temp_BUFF), 0,
                        (struct sockaddr*)&server_addr, &temp_addrsize);
printf("Message Receive>>%s", temp_BUFF);
```

3. 비연결형 클라이언트/서버의 작성

2. 동시 처리 비 연결형 클라이언트/서버



1) 서버와 클라이언트의 소켓 생성

2) 서버는 BIND 작업 후 대기

3) 클라이언트가 서버로 접속 데이터 송신

4) 서버는 클라이언트의 접속 시마다 클라이언트 처리용 스레드 생성

5) 이 후의 클라이언트와의 통신은 각각의 스레드가 담당

6) 서버 종료 작업

7) 종료 시 스레드를 통한 클라이언트 종료

8) 서버와 클라이언트 소켓 닫음 (CLOSE)

(예) 다중 처리용 비연결형 서버

```
while(1)
{           memset(BUFF, '\0', sizeof(BUFF));
            /*접속 클라이언트에 대한 수신 데이타 처리*/
            msgsize = recvfrom(server_socket, BUFF, sizeof(BUFF), 0,
                                    (struct sockaddr*)&client_addr, &addrsize);
            if (msgsize <= 0)       continue;
            memcpy(&curr_addr, &client_addr, sizeof(struct sockaddr_in));

            memset(CURR_BUFF, '\0', sizeof(CURR_BUFF));
            memcpy(CURR_BUFF, BUFF, strlen(BUFF));
            /* 수신에 대한 데이타 처리는 Thread로 처리. 현재의 예제에서는 실용성이 많지 않으나
              실제에서는 수신 데이타에 대한 처리는 많은 작업이 있을수 있음 */
            pthread_create(&thread, NULL, thread_process, (void *)server_socket);
}
void* thread_process(void * server_socket)
{
            memset(temp_BUFF, '\0', sizeof(temp_BUFF));
            memcpy(temp_BUFF, CURR_BUFF, strlen(CURR_BUFF));
            printf("Client Message>>%s", temp_BUFF);
            /*Thread로 처리되는 클라이언트 데이터 송신*/
            sendto((int)server_socket, temp_BUFF, strlen(temp_BUFF), 0,
                        (struct sockaddr*)&temp_addr, sizeof(struct sockaddr));
}
```

(예) 다중 처리용 비연결형 클라이언트

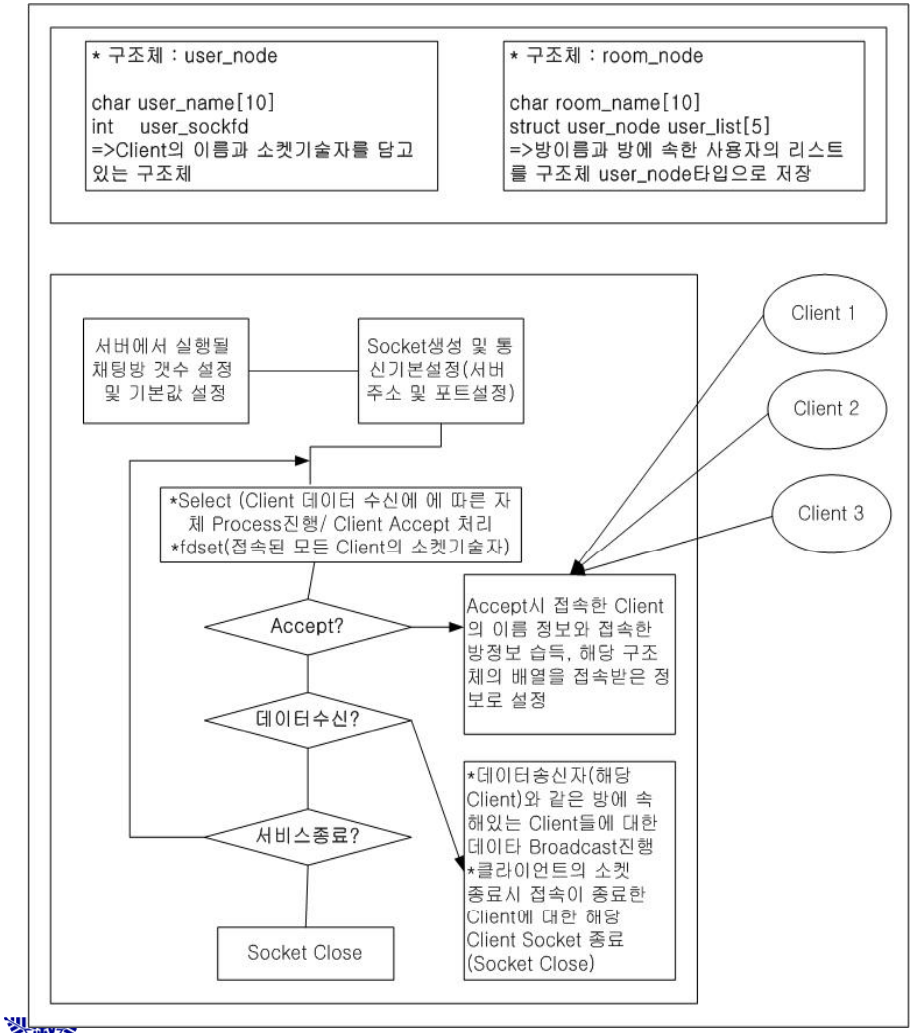
```
/*데이타 수신에 대한 Thread생성*/
pthread_create(&thread, NULL, thread_process, (void*)connect_fd);
while(1)
{
        memset(msg, '\0', sizeof(msg));
        fgets(msg, 1024, stdin);
        msgsize = strlen(msg);
        /*사용자 입력 데이타에 대한 처리. 서버 전송* sendto함수 이용*/
        msgsize = sendto(connect_fd, msg, msgsize, 0, (struct sockaddr*)&connect_addr, addrsize);
        printf("Message Send>>%s", msg);
}

void* thread_process(void * sockfd)
{
        while(1)
        {
                memset(temp_BUFF, '\0', sizeof(temp_BUFF));
                /*서버에서의 송신 데이타에 대한 처리 recvfrom함수 이용*/
                recvfrom((int)sockfd, temp_BUFF, sizeof(temp_BUFF), 0,
                        (struct sockaddr*)&server_addr, &temp_addrsize);
                printf("Message Receive>>%s", temp_BUFF);
        }
}
```

채팅을 위한 클라이언트/서버 프로그램 작성

1. 프로그램 개요 및 구조
2. 채팅 프로그램

1. 프로그램 개요 및 구조 (1)

- 서버 프로그램 구조



1) 서버에서 사용하는 구조체 초기화

2) 서버 소켓 생성 및 클라이언트 대기

3) select를 이용한 접속 상황 관리

- 클라이언트가 접속하면 소켓 기술자의 상태가 변화하여 접속 여부 확인

4) 접속 처리

- 접속 한 클라이언트가 넘겨주는 정보로 서버의 각 구조체를 업데이트함

5) 데이터 송수신

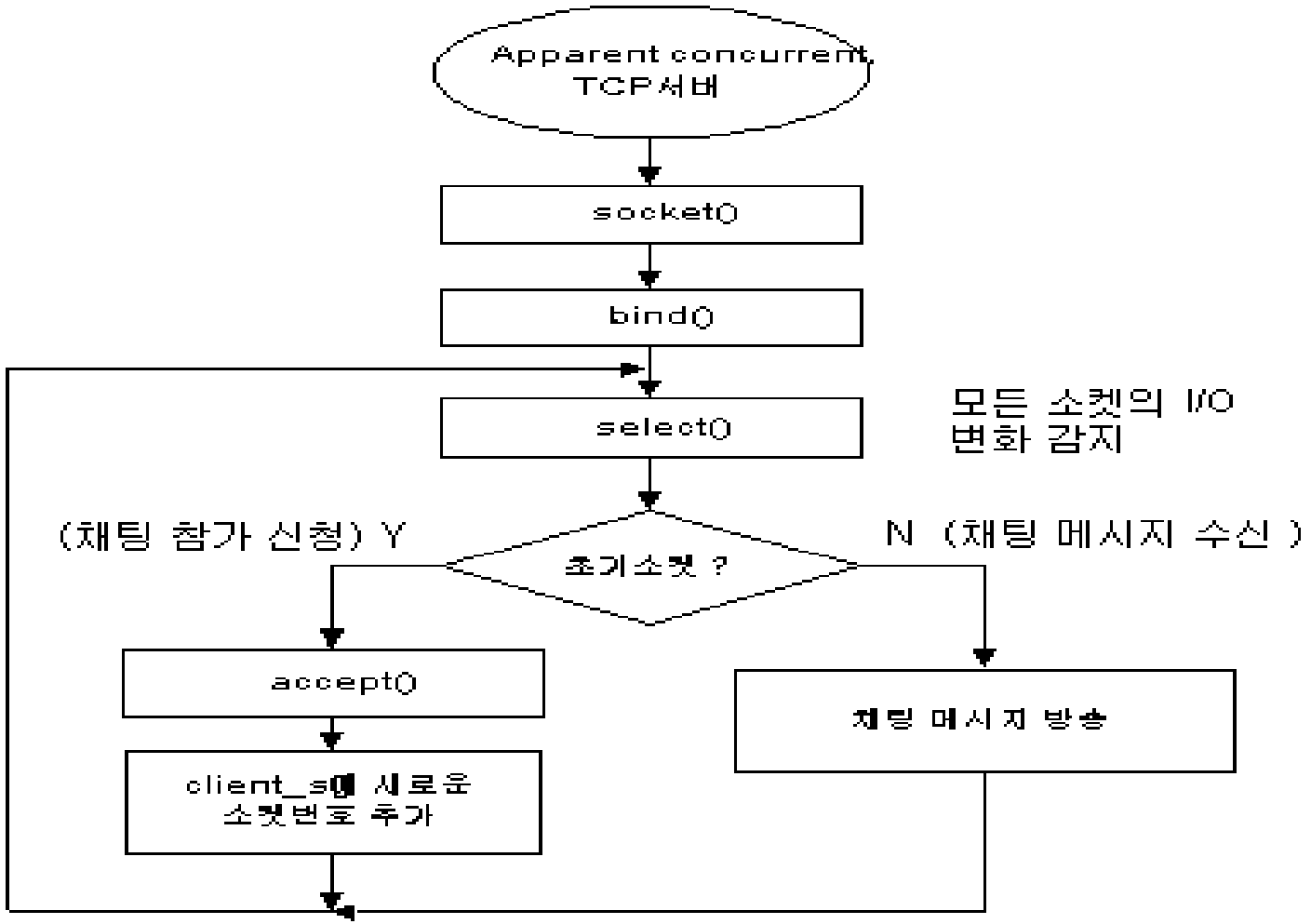
- 클라이언트로 받은 데이터는 검색하여 같은 채팅 방 안에 접속한 각각의 다른 클라이언트에게 전송
- 데이터 수정 작업 필요(이름 첨가 등)

6) 서버 소켓 닫기

- 종료 하지 않는 다면 **3)**으로 복귀

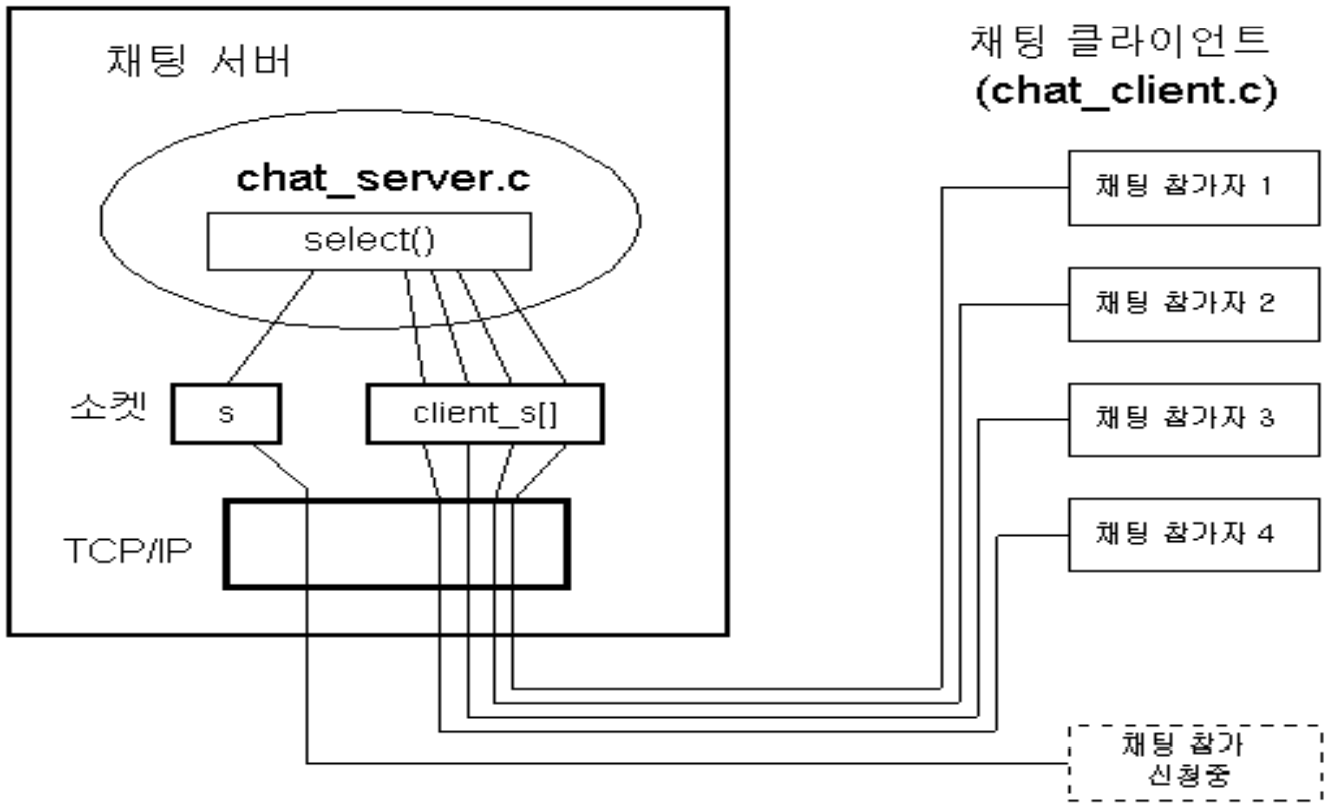
7) 프로그램 종료

채팅 서버 구조


Apparent concurrent TCP서버
socket()
bind()
select()
모든 소켓의 I/O 변화 감지
(채팅 참가 신청) Y
초기소켓 ?
N (채팅 메시지 수신)
accept()
client_s에 새로운 소켓번호 추가
채팅 메시지 방송

채팅 서버 구조 (Cont'd)

- 채팅 서버와 클라이언트와의 연결 관계



서버: 채팅 방 설정

```
/*채팅에서의 방을 3개로 설정*/
struct room_node roomlist[3];

/*각 방에 대한  방이름 초기화*/
strcpy(roomlist[0].room_name,"room1");     strcpy(roomlist[1].room_name,"room2");
strcpy(roomlist[2].room_name,"room3");

/*각 방의 접속 사용사 수를 초기화*/
roomlist[0].user_count = 0;         roomlist[1].user_count = 0;         roomlist[2].user_count = 0;

/* 서버소켓생성*/
server_socket = socket(AF_INET, SOCK_STREAM, 0);

/*서버 IP및 포트 설정*/
server_addr.sin_family = AF_INET;
server_addr.sin_port = htons(SERVER_PORT);
server_addr.sin_addr.s_addr = htonl(INADDR_ANY);
memset(&(server_addr.sin_zero), 0, 8);

bind(server_socket, (struct sockaddr*)&server_addr, sizeof(struct sockaddr));
listen(server_socket, 0);

FD_ZERO(&readfd);
```

서버: 서버에 접속한 클라이언트 소켓 기술자 확인

```
/*서버에 접속한 클라이언트 소켓 기술자[각 방의 유저구조체포함]를 fd_set에 설정*/

FD_SET(server_socket, &readfd);
for (room_index = 0; room_index < 3; room_index++)
{
        for (user_index = 0; user_index < roomlist[room_index].user_count; user_index++)
        {
                tempsockfd = roomlist[room_index].user_list[user_index].user_sockfd;
                FD_SET(tempsockfd, &readfd);

                if (tempsockfd > maxfd)
                        maxfd = tempsockfd;
        }
}
maxfd = maxfd + 1;
```

서버: 클라이언트 접속 처리

```
select(maxfd, &readfd, NULL, NULL,NULL);
 /*서버로의 접속이 있는 클라이언트에 대한 처리*/

 if (FD_ISSET(server_socket, &readfd))
 {
        addrsize = sizeof(struct sockaddr_in);

        client_socket = accept(server_socket, (struct sockaddr*)&server_addr, &addrsize);
        memset(BUFF, '\0', sizeof(BUFF));

        msgsize = read(client_socket, BUFF, sizeof(BUFF));

        if (msgsize <=0)
        {
                printf("Enter user Error\n");
                continue;
        }
}
```

서버: 각 방 별로 참가자 확인 및 확약

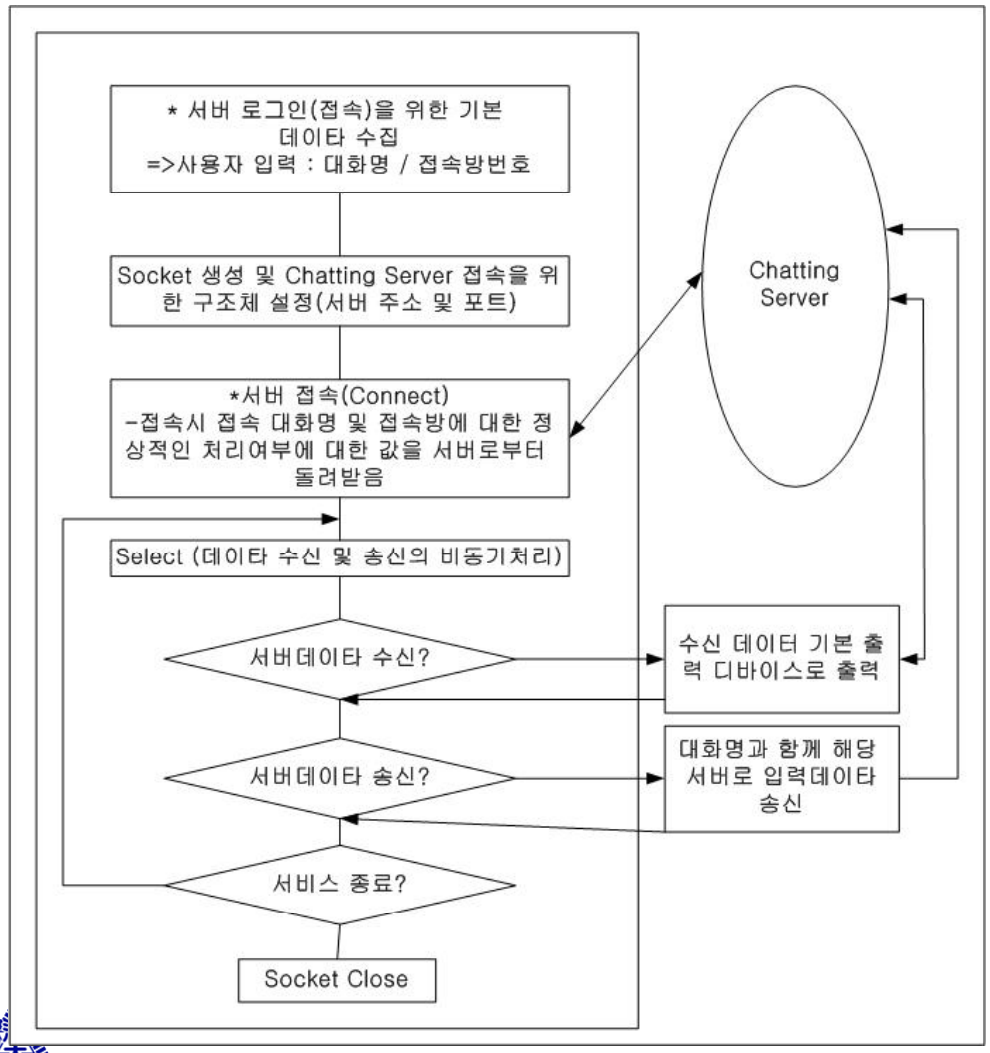
```
printf("Receive Message:%s\n", BUFF);

/*각 방이 5명으로 제한되어있으므로 해당 유저인원 체크*/
if (BUFF[0] == '1')
{
        printf("Login Room1 Count:%d\n", roomlist[0].user_count);
        if (roomlist[0].user_count == 5)
        {
                strcpy(BUFF, "User Count Overflow Error");
                write(client_socket, BUFF, strlen(BUFF));
                close(client_socket);
                continue;
                /*인원초과*/
        }
        roomlist[0].user_list[roomlist[0].user_count].user_sockfd = client_socket;
        roomlist[0].user_count++;
        strcpy(BUFF, "ConnectOK");
        write(client_socket, BUFF, strlen(BUFF));
}
```

서버: 참가자에게 수신 정보 전달

```
for (room_index = 0; room_index < 3; room_index++) {
  for (user_index = 0; user_index < roomlist[room_index].user_count; user_index++) {
    if (FD_ISSET(roomlist[room_index].user_list[user_index].user_sockfd,&readfd)){
                memset(BUFF, '\0', sizeof(BUFF));
                tempsockfd = roomlist[room_index].user_list[user_index].user_sockfd;
                msgsize = read(tempsockfd, BUFF,sizeof(BUFF));
                if (msgsize <= 0) {
                  if (user_index == roomlist[room_index].user_count) {
                              close(roomlist[room_index].user_list[user_index].user_sockfd);
                              roomlist[room_index].user_count--;    }
                  else if (user_index < roomlist[room_index].user_count)  {
                              close(roomlist[room_index].user_list[user_index].user_sockfd);
                              for (temp_user_count = user_index; temp_user_count < roomlist[room_index].user_count;
                                                                  temp_user_count++)
                                {           roomlist[room_index].user_list[temp_user_count] =
                                                      roomlist[room_index].user_list[temp_user_count+1];
                                            roomlist[room_index].user_count--;
                                }
                      }
                  else        {
                              printf("Receive Message=>%s\n", BUFF, msgsize);
                              for (temp_user_count = 0; temp_user_count < roomlist[room_index].user_count;
                                          temp_user_count++)      {
                                          msgsize = strlen(BUFF);
                                          write(roomlist[room_index].user_list[temp_user_count].user_sockfd,
BUFF,msgsize); } }
  } } }
```

1. 프로그램 개요 및 구조 (2)

- 클라이언트 프로그램 구조



1) 서버에 전달 할 기본 데이터 입력
- 사용자의 대화 명 및 방 번호

2) 서버 접속을 위한 구조체 설정
- 서버 주소 및 포트 정보 입력

3) 소켓 생성

4) 서버로 접속 시도 및 확인
- **Connect** 함수 이용하여 접속
- 접속 처리 결과를 서버로부터 받음

5) 데이터 수신
- 수신 데이터는 수정하지 않고 바로 화면에 출력함
- 서버에서 분류되어 전송되었으므로 클라이언트는 따로 처리 할 필요 없음

6) 데이터 송신
- 자신의 대화 명 추가 후 서버로 송신

7) 클라이언트 프로그램 소켓 닫기
- 종료 하지 않는 다면 **5)**으로 복귀

8) 프로그램 종료

클라이언트: 채팅 방 요구 준비

```
/*사용자 대화명 입력 처리*/
        fgets(name,30, stdin);
        name[strlen(name)-1] = '\0';

        while(1)
        {       /*접속 방번호 입력*/
                printf("#Enter Room Number(ex=>1 or 2 or 3)=>");
                room_number = (char)fgetc(stdin);
                fgetc(stdin);

                if (room_number != '1' && room_number !='2' && room_number != '3')
                {       printf("Incorrect Room Number\n");
                        continue;   }
                else    break;      }

        memset(msg, '\0', sizeof(msg));
        msg[0] = room_number;

        for (temp_index = 0; temp_index < strlen(name); temp_index++)
        {
                msg[temp_index+1] = name[temp_index];
        }
```

클라이언트: 채팅방 요구

```
connect_fd = socket(AF_INET, SOCK_STREAM, 0);
connect_addr.sin_family = AF_INET;
connect_addr.sin_port = htons(atoi(argv[2]));
connect_addr.sin_addr.s_addr = inet_addr(argv[1]);
memset(&(connect_addr.sin_zero), 0, 8);

/*서버 접속*/
connect(connect_fd, (struct sockaddr*)&connect_addr, sizeof(connect_addr));
msgsize = strlen(msg);
write(connect_fd, msg, msgsize);
memset(msg, '\0', sizeof(msg));

/*채팅 서버에 대한 접속 결과 확인*/
read(connect_fd, msg, sizeof(msg));
if (!strcmp(msg, "ConnectOK"))
{               printf("*******Server Connection Success********\n");
                printf("*******Start  Chatting Program  ********\n");
}
if (!strcmp(msg, "User Count Overflow Error"))
{               printf("%s\n", msg);
                exit(1);
}
```

클라이언트: 채팅

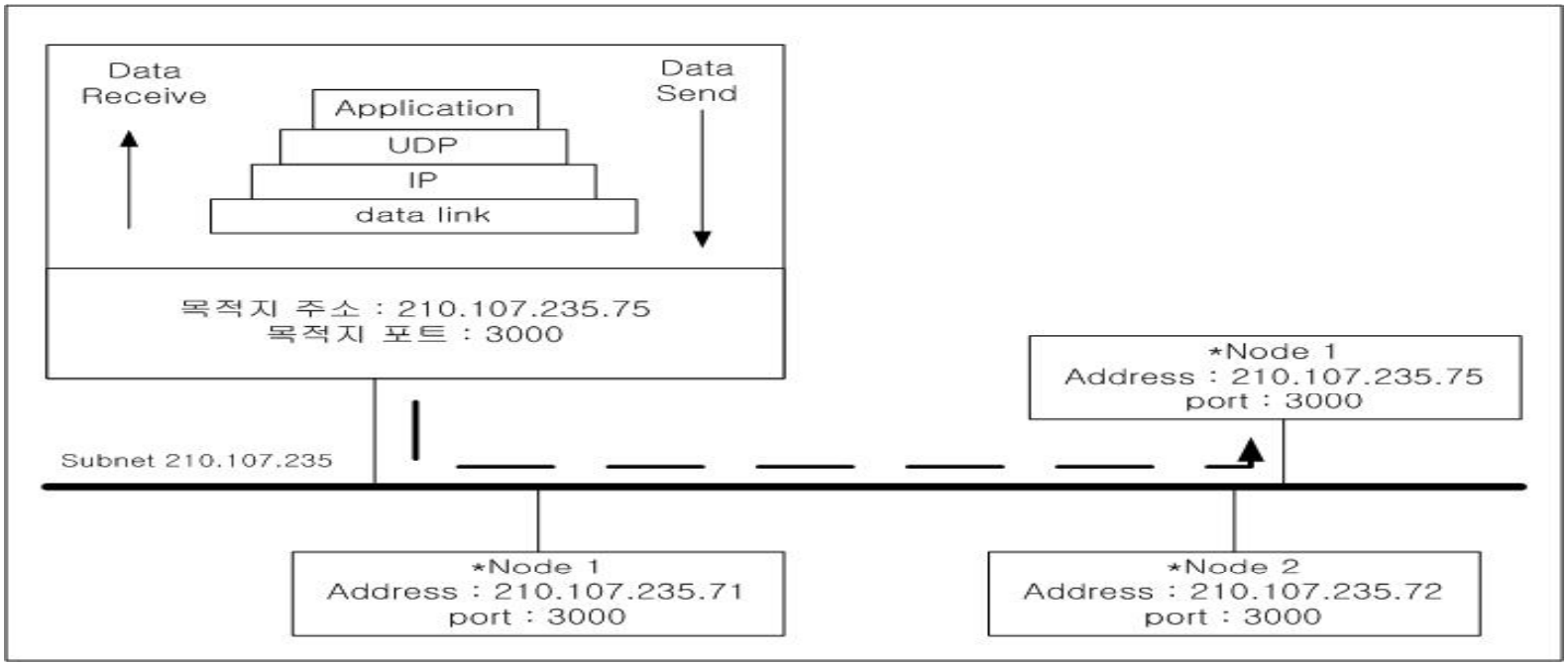
```
FD_ZERO(&readfd);
while(1)
{           FD_SET(0, &readfd);
            FD_SET(connect_fd, &readfd);
            select (connect_fd+1, &readfd, NULL, NULL, NULL);

            if (FD_ISSET(connect_fd, &readfd))
            {           memset(msg, '\0', sizeof(msg));
                        msgsize = read(connect_fd, msg, sizeof(msg));
                        if (msgsize <= 0)       continue;
                        printf("*From=>%s\n", msg);
            }
            if (FD_ISSET(0, &readfd))
            {           memset(msg, '\0', sizeof(msg));
                        fgets(msg, 1024, stdin);
                        msg[strlen(msg)-1] = '[';
                        strcat(msg, name);
                        msg[strlen(msg)] = ']';
                        msgsize = strlen(msg);
                        write(connect_fd, msg, msgsize);
            }
}
```

멀티캐스트 채팅을 위한 프로그램 작성

1. 멀티캐스트 방식의 특징
2. 멀티캐스트 채팅 프로그램

1. 멀티캐스트 방식의 특징

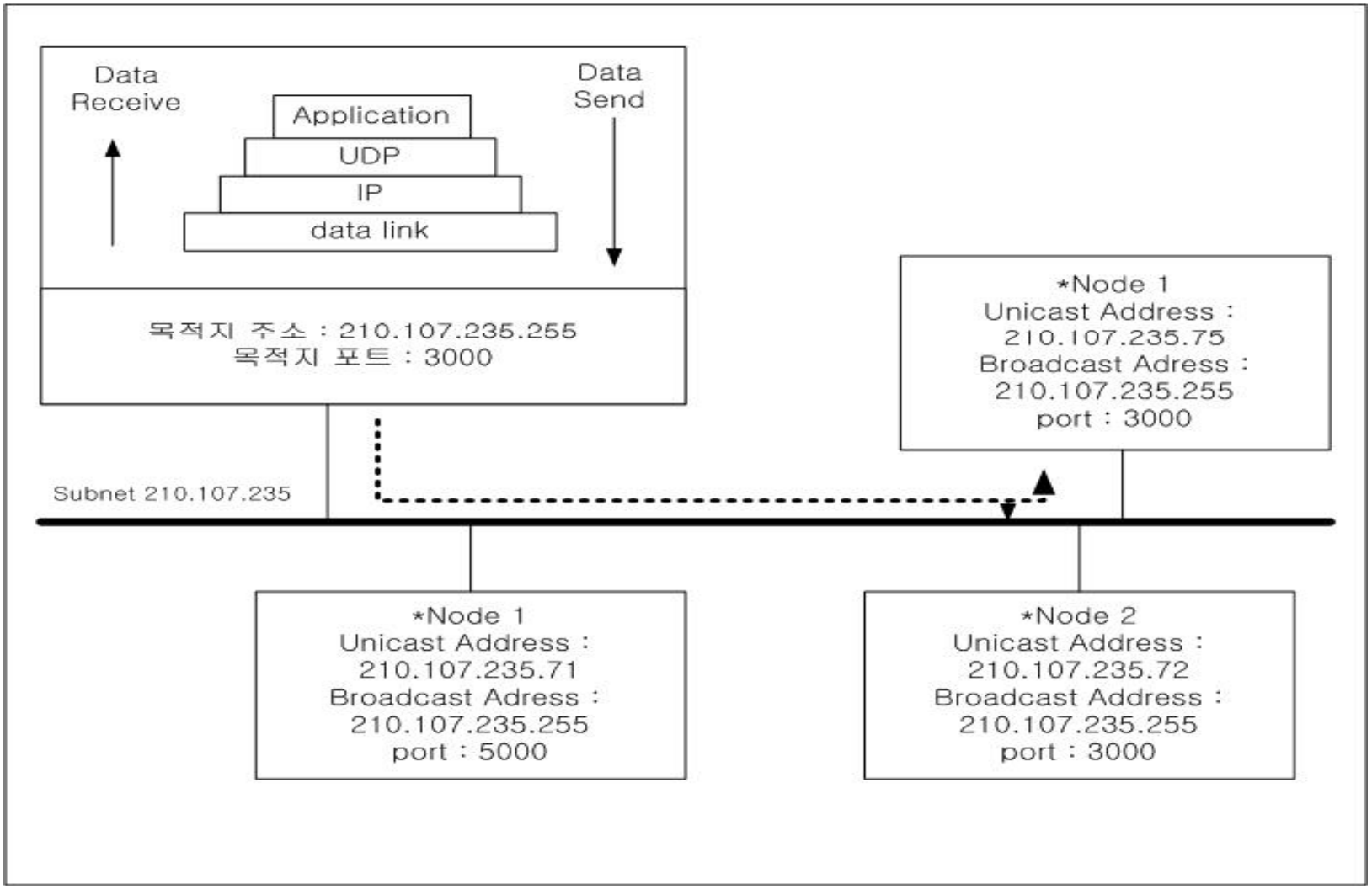
1. Unicast



1. 송수신자가 일대일로 대화하는 네트워크의 가장 일반적인 통신 방식
2. 송신 측 순서 : UDP 계층 > IP 계층 > Data Link 계층 > 전송
3. 수신 측 순서 : Data Link 계층에서 주소 비교 후 목적지라면 송신의 반대순서로 읽어 들임
4. 데이터를 보내는 곳에서 목표로 한 하나의 주소로만 전달됨

1. 멀티캐스트 방식의 특징

2. Broadcast



1. 멀티캐스트 방식의 특징

3. Multicast
   1. 필요성
      - 멀티미디어 데이터의 등장으로 일대다의 전송 방식이 필요하게 됨
      - 유니캐스트 방식은 많은 클라이언트들의 연결을 감당하기 힘듬
      - 브로드캐스트 방식은 같은 종속망의 모든 클라이언트에게 전송하는 단점
      - 그런 이유로 LAN이나 WAN사이에서 이루어지는 멀티캐스트가 필요해짐
   2. 가용 주소

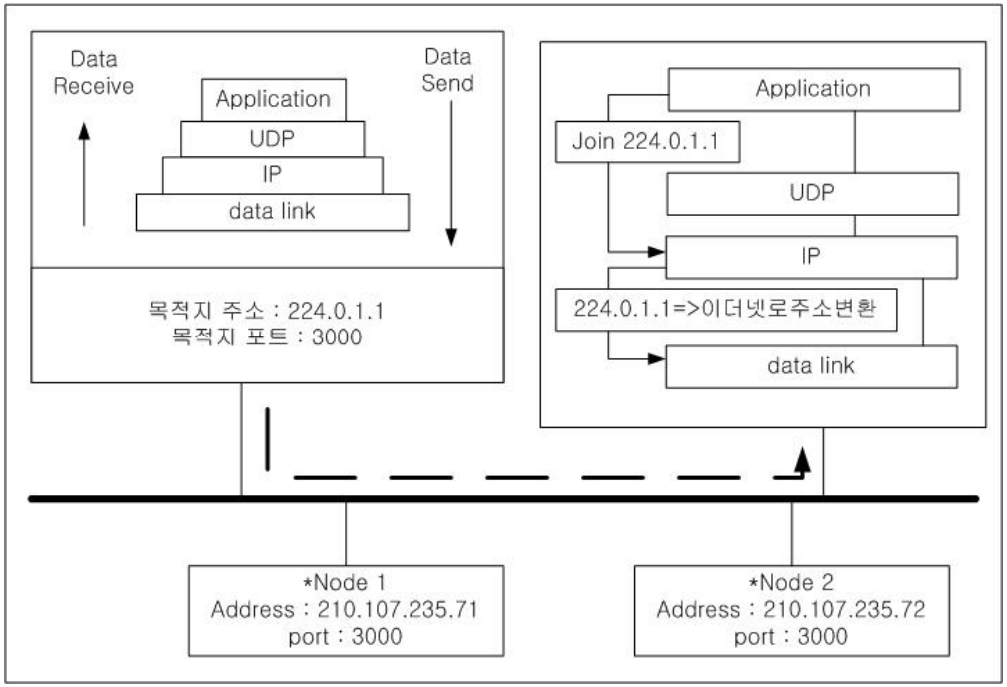
0	Class A 주소	0.0.0.0 ~ 127.255.255.255
10	Class B 주소	128.0.0.0 ~ 191.255.255.255
110	Class C 주소	192.0.0.0 ~ 223.255.255.255
1110	Class D 주소(멀티캐스트주소)	224.0.0.0 ~ 239.255.255.255
11110	예약된 주소	240.0.0.0 ~ 247.255.255.255

- 224.0.0.0 ~ 239.255.255.255 에 걸쳐 사용되는 Class D 가 멀티캐스트
- Class D 의 하위 28비트는 멀티 캐스트 그룹 ID, 32비트는 그룹 주소

1. 멀티캐스트 방식의 특징

3. 멀티캐스트의 흐름

- LAN

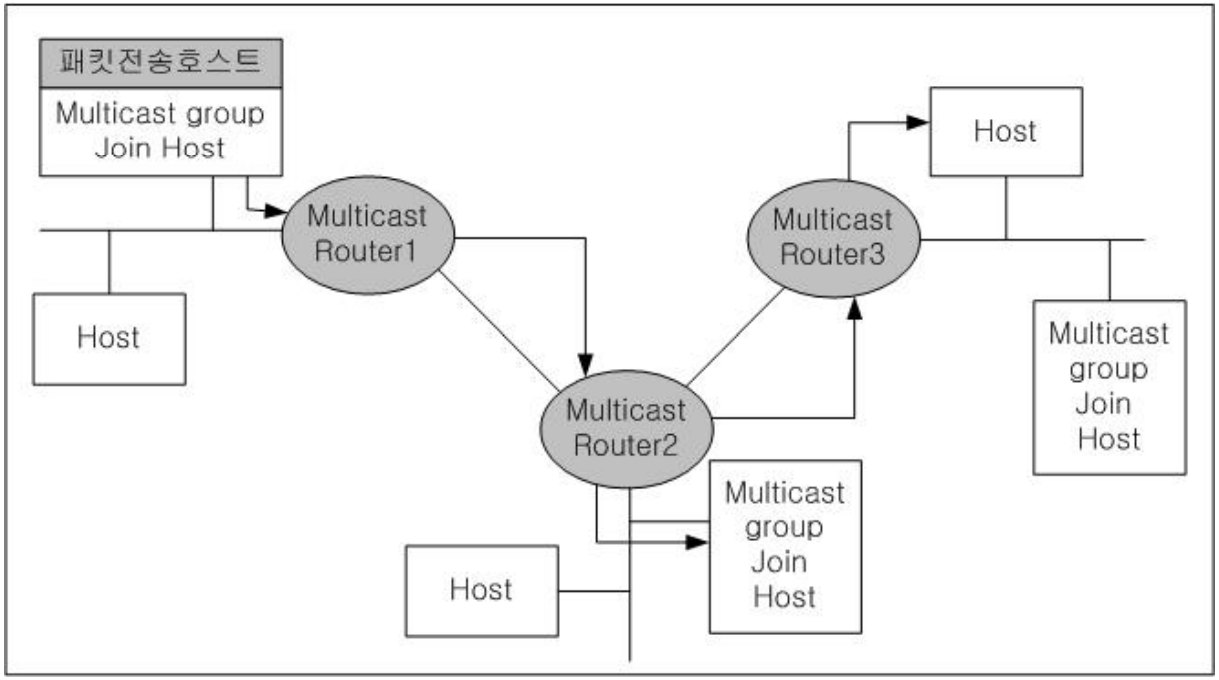


- ➢ 수신 측은 자신의 주소로 대기하지 않고 멀티캐스트 그룹주소로 대기함
- ➢ 해당 멀티캐스트 그룹 주소에 포함된 주소로만 데이터 송신 작업
- ➢ 송신 측은 목적지 주소와 포트를 가지고 해당 호스트로 데이터 전송

1. 멀티캐스트 방식의 특징

3. 멀티캐스트의 흐름

- WAN



- ➢ 송신 측이 속한 LAN에 대하여 우선적으로 처리
- ➢ 멀티캐스트 라우터 사이는 MRP를 이용하여 데이터 전송 처리
- ➢ 외부 LAN으로 전송 시 전송 데이터의 복사본을 만들어 라우터로 전송

* MRP = Multicast Routing Protocol

멀티캐스트 P/G

- 멀티캐스트 (Multicast)
  - 하나의 패킷을 다수의 수신자에게 전달
  - class D 주소를 사용
  - 중간의 라우터들이 멀티캐스트를 지원해야 함
  - UDP 소켓 사용
  - IGMP를 통해 그룹에 참여 또는 탈퇴
  - TTL을 통한 범위 설정
    - TTL(Time-to-Live) in the DNS context defines the duration in seconds that the record may be cached

- 멀티캐스트 패킷 송신
  - UDP 소켓 개설 후 전송
  - 굳이 멀티캐스트 그룹에 가입할 필요가 없다

- 멀티캐스트 패킷의 TTL 설정
  - 패킷이 전체 네트워크로 전송되지 않도록 제한

```
int ttl = 32;
setsockopt(setd_s, IPPROTO_IP, IP_MULTICAST_TTL, &ttl, sizeof(ttl));
```

멀티캐스트 P/G 작성

- 멀티캐스트 구조체

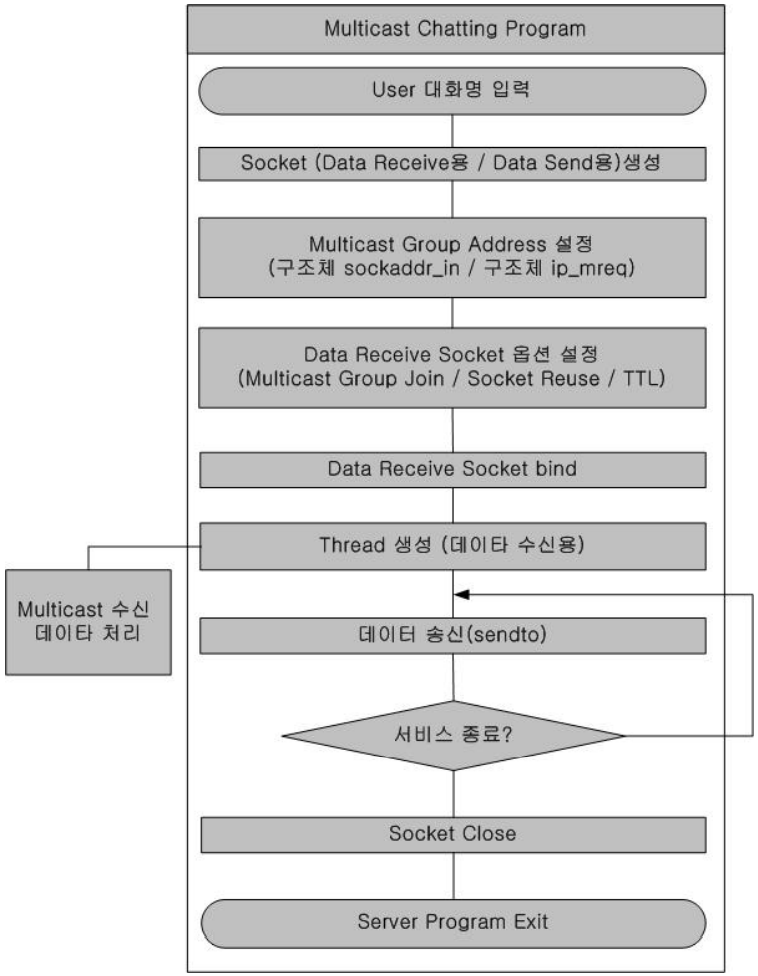
```
struct ip_mreq {
  struct in_addr imr_multiaddr;/* 멀티캐스트 그룹 주소*/
  struct in_addr imr_interface;/* 자신의 IP 주소 */
}
```

- 멀티캐스트 패킷 수신

```
setsockopt(recv_s, IPPROTO_IP, IP_ADD_MEMBERSHIP,
            &mreq, sizeof(mreq));
setsockopt(recv_s, SOL_SOCKET, SO_REUSEADDR,
            &set, sizeof(set));
bind(recv_s, (struct sockaddr*)&mcst_group,
     sizeof(mcast_group));
```

2. 멀티캐스트 채팅 프로그램

1. 구조



(0) 멀티캐스트는 서버/클라이언트가 동일

(1) 멀티 캐스트 그룹 주소 설정 작업

```
    struct ip_mreq {
struct in_addr imr_multiaddr;
                /*멀티캐스트 그룹 주소*/
                struct in_addr imr_interface;
                /*자신의 IP주소*/
                }
```

(2) 데이터 송수신 소켓 생성**(UDP)**

(3) 생성한 소켓의 옵션 설정

IP_ADD_MEMBERSHIP : 멀티캐스트 그룹 가입

IP_DROP_MEMBERSHIP : 멀티캐스트 그룹 탈퇴

IP_MULTICAST_LOOP : 멀티캐스트패킷 루프백 여부

IP_MULTICAST_TTL : 멀티캐스트패킷 **TTL** 값

(4) 설정한 소켓 연동**(BIND)**

(5) 데이터 송수신

(6) 소켓 종료**(CLOSE)**

(7) 프로그램 종료

(예) 멀티캐스트 프로그램

```
/*멀티 캐스트 수신 및 송신 소켓 생성*/
multicast_recv_sockfd = socket(AF_INET, SOCK_DGRAM, 0);
multicast_send_sockfd = socket(AF_INET, SOCK_DGRAM, 0);

memset(&multicast_group_addr, 0, sizeof(multicast_group_addr));

/*멀티 캐스트 주소 및 포트 설정*/
multicast_group_addr.sin_family = AF_INET;
multicast_group_addr.sin_port = htons(atoi(argv[2]));
multicast_group_addr.sin_addr.s_addr = inet_addr(argv[1]);

/*멀티 캐스트 주소와 수신 IP주소 설정*/
mreq.imr_multiaddr = multicast_group_addr.sin_addr;
mreq.imr_interface.s_addr = htonl(INADDR_ANY);

multicast_recv_sockfd = socket(AF_INET, SOCK_DGRAM, 0);

/*소켓 옵션 설정 멀티캐스트 그룹 추가*/
setsockopt(multicast_recv_sockfd, IPPROTO_IP, IP_ADD_MEMBERSHIP, &mreq, sizeof(mreq));
/*소켓 재사용*/
setsockopt(multicast_recv_sockfd, SOL_SOCKET, SO_REUSEADDR, &multicast_index, sizeof(multicast_index));
/*TTL설정*/
setsockopt(multicast_recv_sockfd, IPPROTO_IP, IP_MULTICAST_TTL, &ttl, sizeof(ttl));

send_result = bind(multicast_recv_sockfd, (struct sockaddr*)&multicast_group_addr, sizeof(multicast_group_addr));
```

Part 4

자바 네트워크 프로그래밍 환경 소개

1. 입출력 스트림 클래스의 소개
2. 스레드(Thread) 클래스의 소개

(

- OutputStream 클래스 메소드

함 수	반환형	설 명
close()	void	스트림을 닫아서 이 스트림이 사용하는 모든 시스템 자원을 해제한다.
flush()	void	버퍼에 들어가 있는 출력 바이트를 강제적으로 기입한다.
write(int b)	void	이 출력 스트림으로 지정된 바이트를 기입한다.
write(byte[] b)	void	지정된 바이트 배열로부터 출력스트림에 b.length 바이트를 기입한다.
write(byte[] off, int len)	void	off 로부터 시작되는 바이트 배열로부터 출력스트림에 len 바이트를 기입한다.

[표10.2] 바이트 스트림 - OutputStream 클래스 메소드

- OutputStream 클래스 메소드의 사용
  - 다음은 OutputStream 클래스의 메소드를 이용하여 파일에 문자를 삽입하는 프로그램이다.
  - 절대경로 "i:/" 에 Example.java 파일을 생성하고, 그 파일에 문자 'a'를 100번 기입한다.

```
1: import java.io.*;
2:
3: public class OutputStreamTest{
4:     public static void main(String[] args) {
5:         try{
6:             OutputStream out = new FileOutputStream("i:/Example.java");
7:             int i;
8:             for(i=0; i<100 ; i++)
9:             {
10:                out.write('a');
11:            }
12:            out.close();
13:        } catch(IOException e) {
14:            System.out.println(e);
15:        }
16:    }
17: }
```

- OutputStream 클래스 타입의 객체참조 out을 선언
  - 절대경로 “i:/Example.java" 파일에 데이터를 쓰기 위해 FileOutputStream을 이용하고, 읽어온 데이터를 OutputStream 객체참조에 연결함.

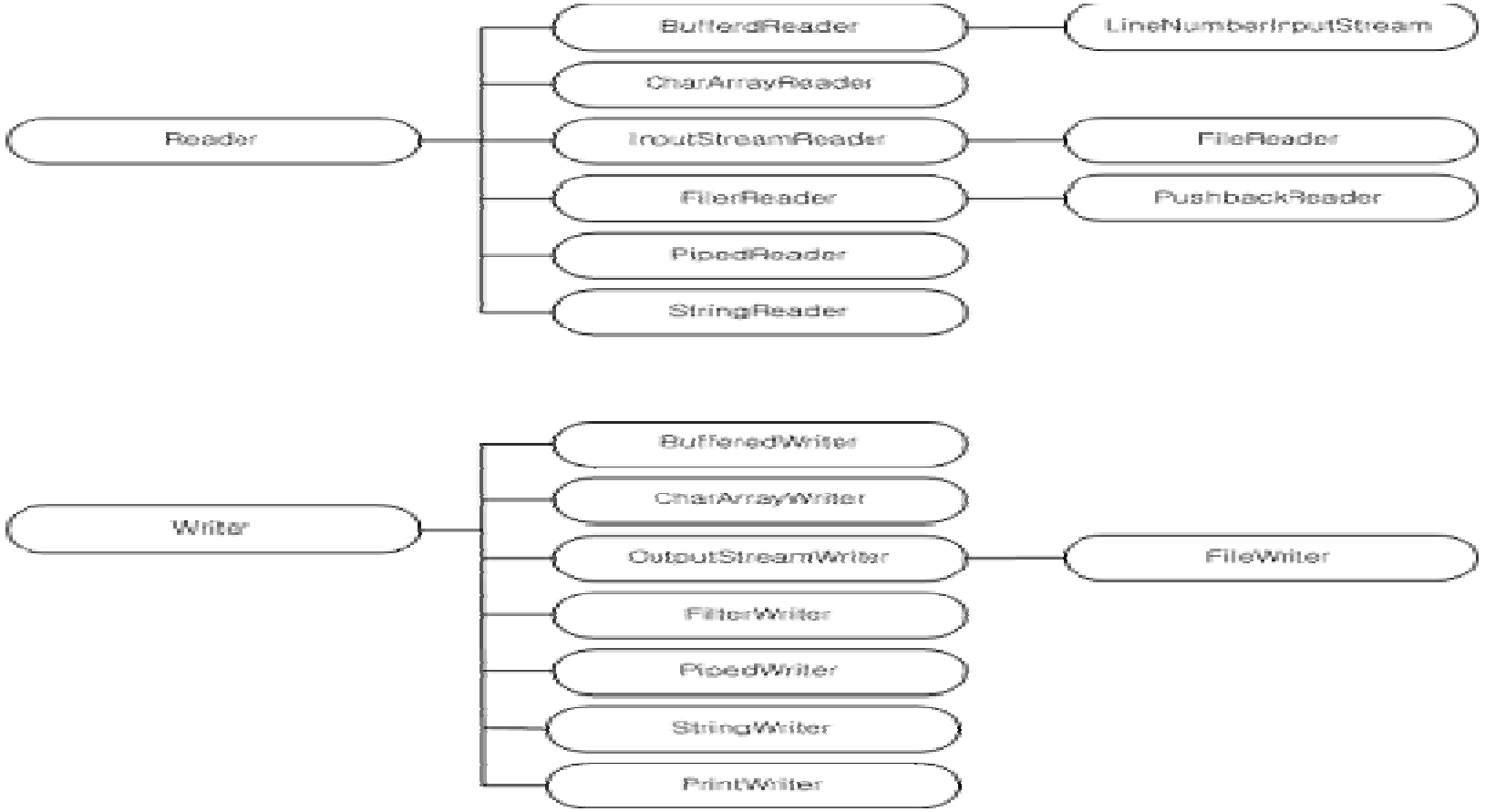
```
OutputStream out = new FileOutputStream("i:/Example.java");
```

- 문자 ‘a’를 파일에 100 번 출력하도록 함.

```
for(i=0; i<100 ; i++)
  {
        out.write('a');
  }
```

문자 스트림 클래스

- 문자 스트림 클래스
  - 2바이트의 문자를 다룬다는 것을 제외하고 바이트 스트림과 거의 유사함.
  - 메소드 역시 유사한데, 매개변수로 사용되는 데이터의 타입이 byte가 아니라 char라는 특징



- Reader, Writer 클래스에서도 InputStream, OutputStream 클래스와 마찬가지로 기본적인 메소드를 지원.
- Reader 클래스의 read 메소드들은 InputStream 클래스의 read() 메소드와 유사함.

  – 차이점

    - InputStream 에서는 read() 메소드가 추상 메소드이고, 다른 read() 메소드들은 이 메소드를 바탕으로 쓰여짐.
    - Reader 클래스에서는 반대로 read(char[] buf, int off, int len) 메소드가 추상 메소드임.
    - 파일 등 대부분의 스트림에서 한 두 바이트 단위로 입출력하는 것보다는 더 큰 단위로 입출력하는 것이 훨씬 효율적이기 때문에 이런 방식으로 바꿈.

- Reader/Writer 클래스 메소드

함 수	반환형	설 명
close()	void	Stream을 닫고, 모든 시스템 자원을 반환한다.
mark(int limit)	void	Stream의 현재의 위치에 마크를 설정한다.
read()	int	하나의 문자를 읽어들인다.
read(char[] buf)	int	배열에 문자를 읽어들인다.
read(char[] buf, int off, int len)	int	배열의 일부에 문자를 읽어들인다.
skip(long n)	long	문자를 스킵한다.

Reader 클래스 메소드

함 수	반환형	설 명
close()	void	Stream을 닫고, 모든 시스템 자원을 반환한다.
write(int c)	void	하나의 문자를 기입한다.
write(String str)	void	스트링을 기입한다.
write(char[] buf)	void	문자의 배열을 기입한다.
write(char[] buf, int off, int len)	void	문자 배열의 일부를 기입한다.
write(String str, int off, int len)	void	스트링의 일부를 기입한다.

Writer 클래스 메소드

- 바이트 스트림에서의 문자처리
  - 바이트단위로 읽어온 문자를 char형으로 변환
- 문자 스트림에서의 문자처리
  - 16bit 유니코드를 사용하는 자바에서 한글이 깨지는 문제를 해결

- 문자 스트림 클래스의 사용
  - 사용자입력을 무한정 받아들여 파일에 기술하는 프로그램

```
1: import java.io.*;
2:
3: public class StringInput {
4:       public static void main(String [] args) throws Exception {
5:
6:     if (args.length < 1 ) {
7:               System.out.println("Usage : java StringInput file");
8:               return;
9:       }
10:        String inputString;
11:        InputStreamReader isr = new InputStreamReader(System.in);
12:        BufferedReader br = new BufferedReader(isr);
13:
14:        FileOutputStream fos = new FileOutputStream(args[0]);
15:        OutputStreamWriter osr = new OutputStreamWriter(fos);
16:        BufferedWriter bw = new BufferedWriter(osr);
17:
18:        while((inputString = br.readLine()) != null) {
19:               bw.write(inputString + "₩n");
20:        }
21:        bw.close();
22:        br.close();
23:    }
24: }
```

- InputStreamReader 객체는 사용자의 입력을 위해 준비

```
InputStreamReader isr = new InputStreamReader(System.in);
```

- 시스템의 완충작용 즉, 버퍼링을 지원하기 위해서 InputStreamReader 객체를 BufferedReader 객체의 인자로 줌.

```
BufferedReader br = new BufferedReader(isr);
```

- 사용자 입력을 파일로 생성하기 위해 FileOutputStream을 생성했는데, 이때 인자로 들어오는 값은 처음 자바를 실행할 때, 넣어주었던 arg[0] 값이됨.

```
FileOutputStream fos = new FileOutputStream(args[0]);
```

- 실제 사용자 입력 즉, BufferedReader 스트림을 null이 아닐 때 까지 한 라인씩 읽어서 BufferedWriter 객체를 이용해 파일에 기술함.

```
while((inputString = br.readLine()) != null) {
    bw.write(inputString + "\n");
   }
```

스레드(Thread) 클래스의 소개

- 스레드
  - 한 프로그램 내에서 독립적으로 수행되는 하나의 제어흐름
  - 스레드는 어떤 변수나 파일을 각 각의 스레드들 간에 공유하기에 편리한 방법.

- 프로세스
  - 완전히 독립적으로 수행되지만, 시스템 자원을 사용

- 자바에서는 C언어와는 다르게 스레드를 자체적으로 지원함으로 C에서 사용하는 fork()와 같은 시스템 자원을 사용하지 않는다.

스레드의 상태 변이

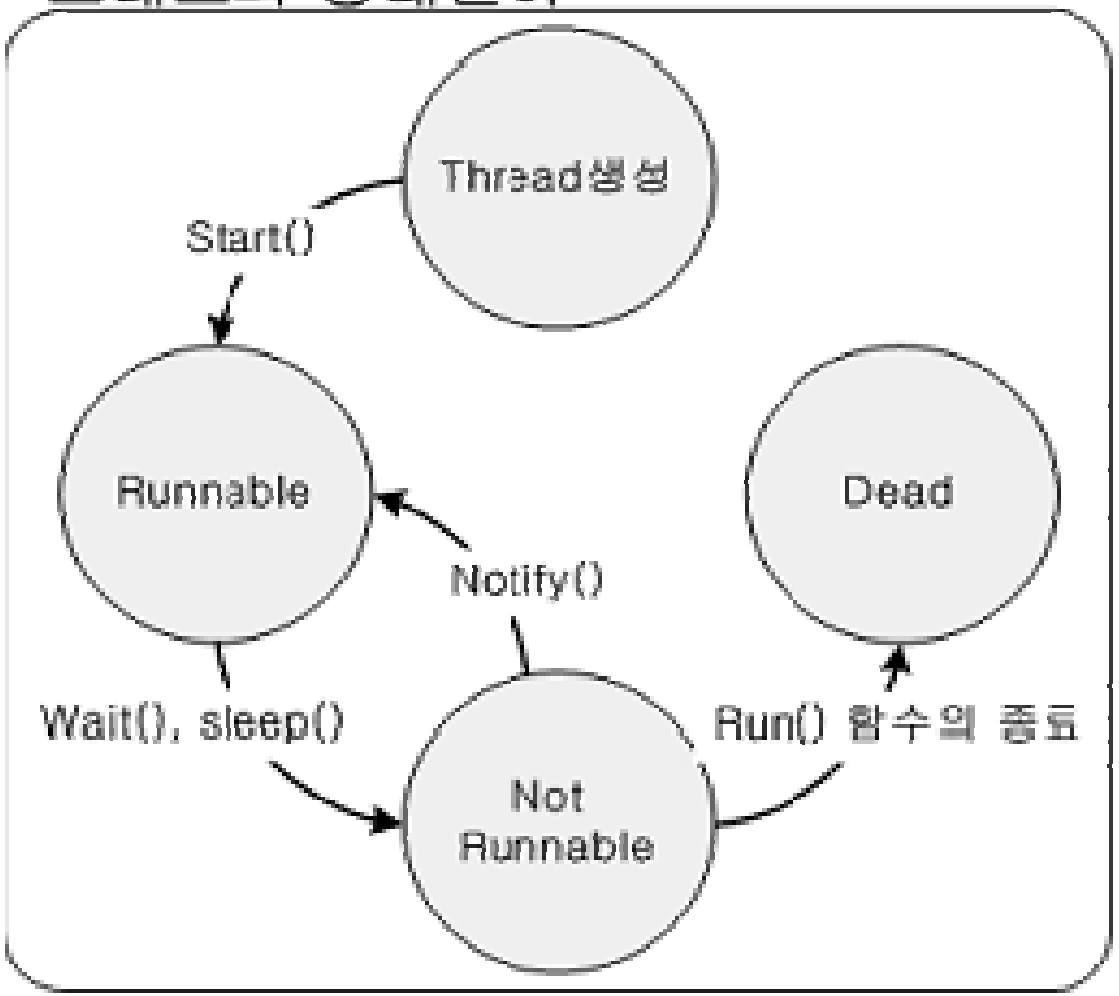
- Thread생성
  - 스레드를 디자인하고 new 연산자를 이용하여 스레드를 생성한 것을 의미.

- Runnable
  - 새로운 스레드가 생성되어 그 스레드의 start() 메소드를 호출하면, 스레드는 실행 가능한 Runnable 상태가 됨.
  - 실행권한을 가진 후보스레드
    - 실행 큐라고 하는 것에 스레드를 넣어두고 자바 런타임 스케줄러에 의해서 처리.
    - 실행큐란 Runnable상태에 있는 스레드들을 넣어두는 자료구조.
    - CPU는 한순간에 단 하나의 스레드만을 실행함.
      - 실행큐에 있는 스레드들 중에서 하나만을 골라서 실행

- Not Runnable
  - 실행중인 스레드의 wait(), sleep() 메소드를 호출하게 되면 스레드는 Runnable 상태에서 Not Runnable상태로 전환하게 된다. 전에 말한 실행큐에서 제거되고 대기큐라는 곳으로 자리를 옮기는 것이다.

- Dead
  - 스레드가 할 일을 모두 마치면 스레드는 Dead 상태가 됨.
  - 즉, 스레드의 run 메소드가 끝나거나 리턴 된 경우 스레드는 Dead상태가 된다.

스레드의 상태변이


Thread생성
Start()
Runnable
Notify()
Dead
Wait(), sleep()
Run() 함수의 종료
Not
Runnable

스레드 클래스의 사용

- 자바에서 스레드를 사용하기 위한 두 가지 방법
  - Thread 클래스를 상속
  - interface 클래스인 Runnable 클래스를 구현

- 스레드 클래스의 사용
  - 다중 상속을 피하기 위해 두 가지 방법을 지원

Thread 클래스의 상속

- 다른 클래스를 상속받지 않고, 단지 java.lang.Thread 클래스만을 상속받을 때 사용가능.

```
public class Thread_Test extends Thread
```

- 스레드 클래스를 상속받았다면, 실제 스레드 객체를 생성하여 스레드를 시작하게 하여야함.
  - 스레드의 시작을 알리는 메소드는 start() 임.

```
Thread thread = new Thread_Test();
thread.start();
```

프로그램의 구현

• 본 프로그램은 스레드를 이용하여 **0.5**초 간격으로 "**Thread** 번호"를 출력하는 프로그램이다.

```
3: public class ThreadTest extends Thread
4: {
5:         int test;
6:         public ThreadTest()
7:         {
8:         }
9:
10:        public void run()
11:        {
12:                    try
13:                    {
14:                              int i=0;
15:                              for(i=0; i<100; i++)
16:                              {
17:                                        System.out.println("Thread" + i");
18:                                        System.out.println("");
19:                                        sleep(500);
20:                              }
21:                    } catch(InterruptedException ie)
22:                    {
23:                    }
24:        }
25:
26:        public static void main(String[] args)
27:        {
28:                    Thread thread = new ThreadTest();
29:                    thread.start();
30:        }
31: }
```

- ThreadTest 클래스는 Thread 클래스를 상속받음.

```
public class ThreadTest extends Thread
```

- Thread 클래스의 객체를 생성하여 스레드를 시작함.
  - run() 메소드를 실행함

```
Thread thread = new ThreadTest();
   thread.start();
```

- 0 ~ 99 까지 루프를 돌면서 sleep(500) 기간동안, 즉 0.5초 간격으로 Thread + 번호 를 출력함.

```
for(i=0; i<100; i++)
  {
    System.out.println("Thread“ + i");
    System.out.println("");
    sleep(500);
  }
```

Runnable 클래스의 구현

- Runnable 클래스는 interface 클래스로 스레드 자원을 사용할 모든 객체가 준수해야할 기본적인 형을 정의함.

- 스레드를 생성하기 위한 두 번째 방법
  - java.lang.Runnable 클래스를 구현.
  - Runnable 인터페이스의 정의
    - 인터페이스를 구현할 때에는 정의된 메소드를 반드시 구현하여야 함.

```
public interface Runnable {
public abstract void run(); // 꼭 구현해야하는 메소드
}
```

- Thread 클래스의 생성자와 Runnable 인터페이스를 이용하면, Thread 클래스를 상속받는 것과 마찬가지로 Thread를 이용할 수 있다.

```
class RunnableTest extends A implement Runnable {
public void run() {
  // 실행될 코드 삽입.....
}
}
```

- 다음은 Runnable 인터페이스를 구현하는 클래스를 정의하고 Thread 생성자를 이용하여 Thread 자원을 사용하는 방법이다.
  - 우선 RunnableTest 클래스의 객체를 만들고, 그 객체를 Thread 클래스 객체의 인자로 넣어준다.
  - 마지막으로 Thread 클래스 객체 t를 start() 해주면 Runnable 인터페이스를 이용한 스레드를 생성할 수 있다.

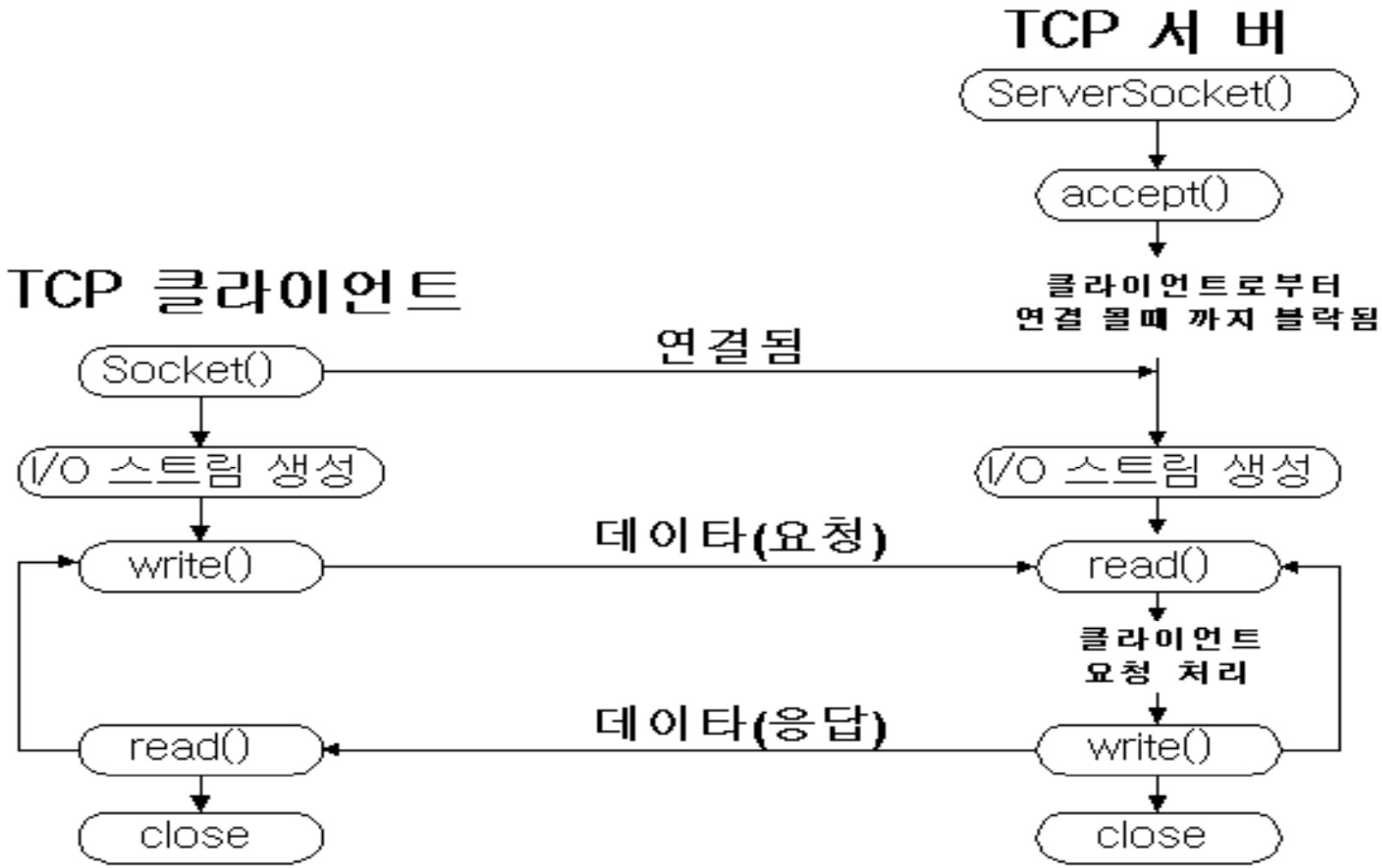
```
RunnableTest runnable = new RunnableTest();
Thread t = new Thread(runnable);
t.start();
```

프로그램의 구현

- 프로그램의 내용은 ThreadTest.java와 마찬가지로 "Thread i"를 0에서부터 99까지 0.5초 간격으로 출력하는 프로그램이다.

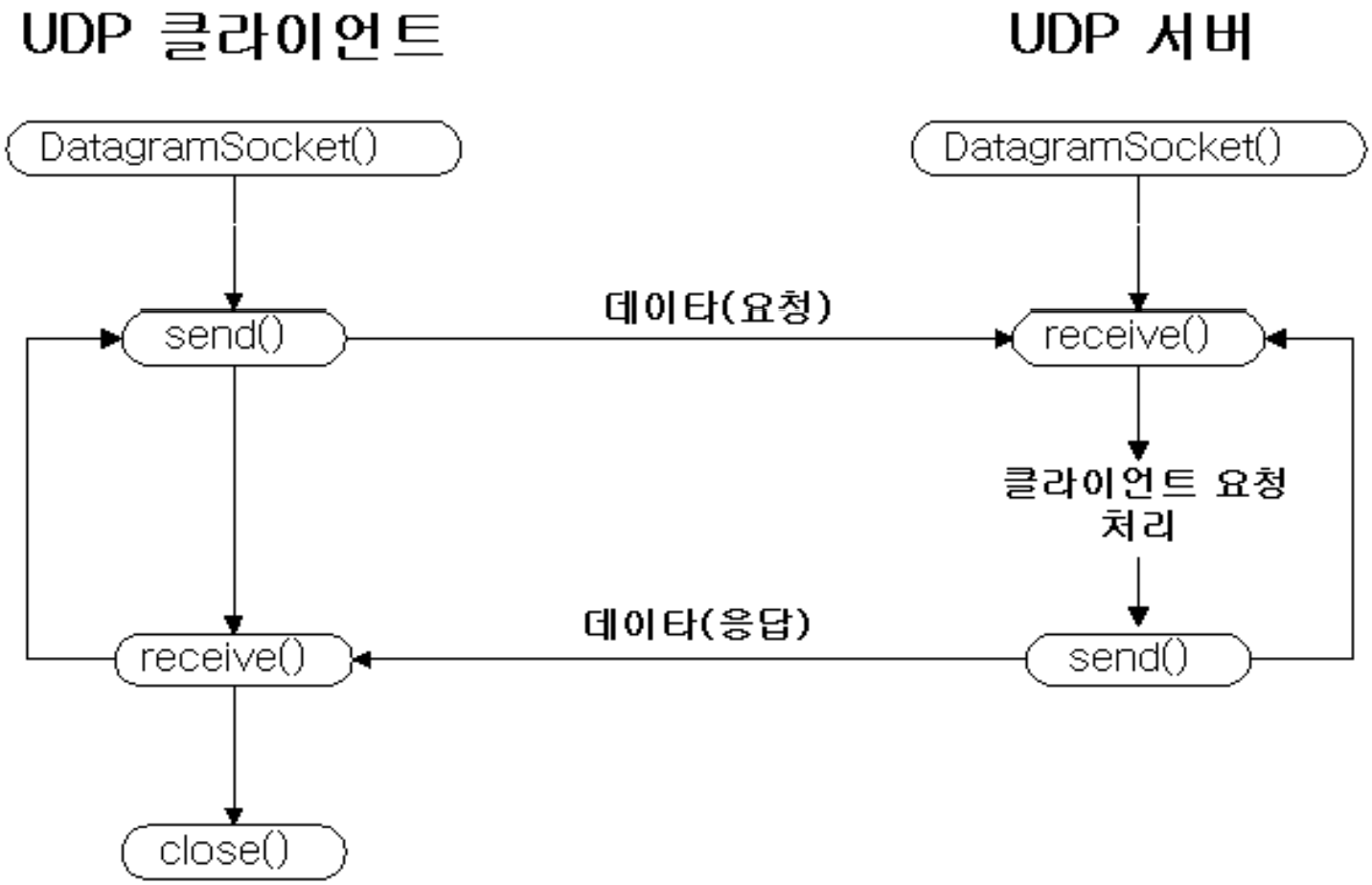
```
public class RunnableTest extends A implements Runnable
{
        int test;

        public RunnableTest()
        {
        }

        public void run()
        {
                try
                {
                        int i=0;
                        for(i=0; i<100; i++)
                        {
                                System.out.println("Thread" + i);
                                System.out.println("");
                                Thread.sleep(500);
                        }

                } catch(InterruptedException ie)
                {
                        System.out.println(ie);
                }
        }
        public static void main(String[] args)
        {
                RunnableTest runnable = new RunnableTest();
                Thread t = new Thread(runnable);
                t.start();
        }
}
```

Java 에서 TCP 소켓 프로그램 작성



Java에서 DatagramSocket

- 자바 UTP 프로그램 작성



JAVA Socket (1)

- 자바에서 네트워크 프로그램(서버)
  - java.net 패키지
  - 서버 소켓(ServerSocket)이라는 클래스를 서버측에서 사용
    - 생성자

      - ServerSocket(int port)
      - ServerSocket(int port, int backlog)
      - ServerSocket(int port, int backlog, InetAddress bindAddr)

    - 생성자의 port 부분은 서버 소켓이 사용할 포트 번호를 표시
      - 포트 번호는 2 바이트로 표현, 0~2^{16}-1 사이의 값을 가짐
      - 0~1023번까지는 보통 시스템에서 사용할 목적으로 예약, 사용자는 1024번 이후를 사용하는 것이 바람직

JAVA Socket (2)

- backlog 아규먼트
  - 동시에 컨넥션 요청이 오는 경우에, 큐에 클라이언트의 요청을 몇 개까지 기록할 것인가를 기술
  - backlog는 큐의 최대 길이
  - backlog보다 많은 클라이언트가 컨넥션을 기다리고 있으면 나머지 클라이언트들은 컨넥션되지 않는다.
- bindAddr은 서버 프로그램이 여러 개의 컴퓨터에서 작동되는 경우에 각 컴퓨터에서 컨넥션을 받는 것이 아니라, 특정한 컴퓨터의 포트만 이용해서 컨넥션이 이루어지도록 할 때 사용

– 프로그래머는 서버 소켓을 만들어서 accept() 메소드를 호출
– accept() 메소드는 클라이언트로부터 컨넥션이 요청될 때까지 서버를 블락시키고 있다가, 클라이언트로부터 요청이 들어오면 클라이언트와 통신할 수 있는 Socket 클래스를 리턴한다.

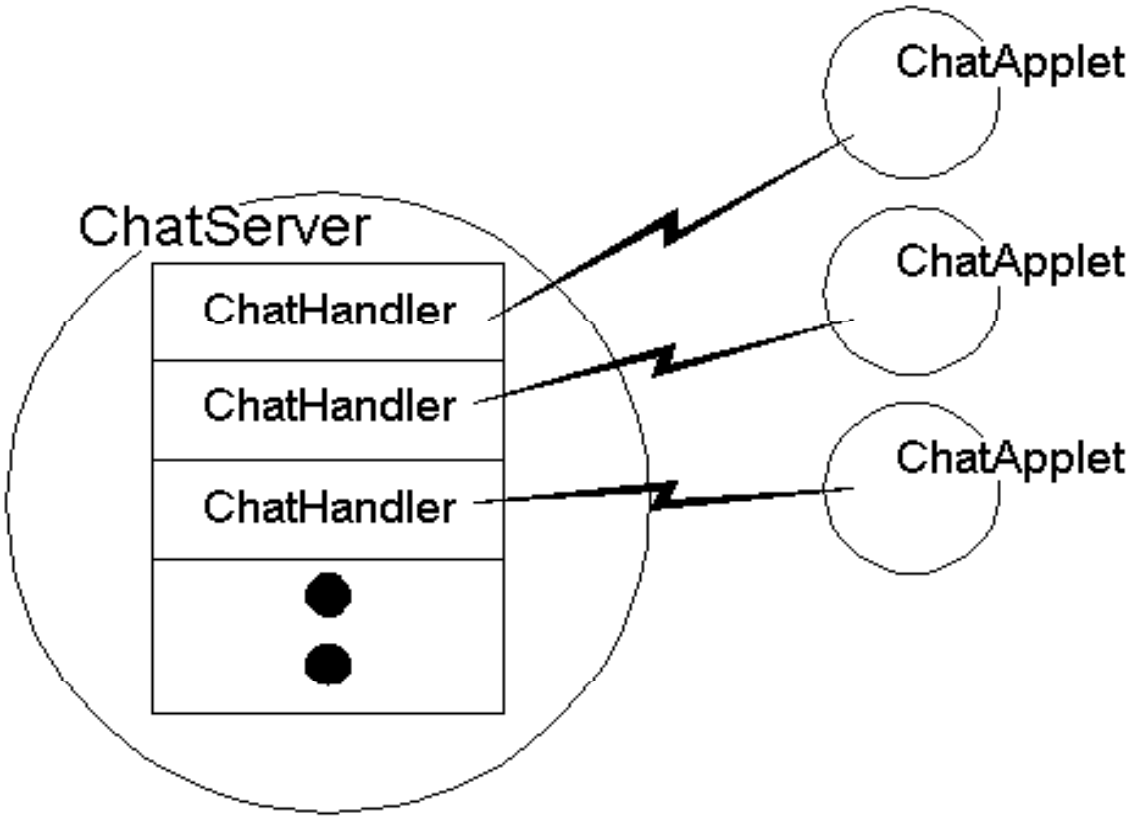
JAVA Socket (3)

- 자바에서 네트워크 프로그램(클라이언트)
  - 소켓(Socket)이라는 클래스를 이용한다.
    - 생성자
      - Socket(String host, int port)
      - Socket(InetAddress address, int port)

    - host 아규먼트는 컴퓨터의 이름을 기술하는 문자열이고, port는 포트 번호를 의미
    - address 아규먼트는 IP 어드레스를 나타낸다. 자바에서는 IP 어드레스를 위해 InterAddress 클래스를 만들었다.
  - Socket 클래스를 만든 후에는 클라이언트와 서버 사이에 데이터를 주고받을 수 있는 I/O 스트림을 만들어야 한다.
  - 소켓으로부터 데이터를 받아들이기 위해서는 InputStream이 필요하고, 데이터를 전송하기 위해서는 OutputStream이 필요
  - 소켓에서 InputStream을 얻기 위해서는 getInputStream() 메소드를, OutputStream을 얻기 위해서는 getOutputStream() 메소드를 이용

JAVA Socket (4)

채팅 프로그램



JAVA Socket (5)

예제 : ChatServer.java

```
 1  import java.net.*;
 2  import java.io.*;
 3  import java.util.*;
 4
 5  public class ChatServer {
 6      protected Vector handlers;
 7
 8      public ChatServer (int port) throws IOException {
 9          ServerSocket server = new ServerSocket (port);
10          handlers = new Vector(2, 5);
11          while (true) {
12              Socket client = server.accept ();
13              System.out.println ("Accepted from " + client.getInetAd
14              ChatHandler c = new ChatHandler (this, client);
15              handlers.addElement(c);
16              c.start ();
17          }
18      }
.....
```

JAVA Socket (6)

예제 : ChatHandler.java

```
……..
 5  public class ChatHandler extends Thread {
 6      protected Socket                s;
 7      protected DataInputStream       i;
 8      protected DataOutputStream      o;
 9      protected ChatServer            server;
10      protected boolean               stop;
11
12      public ChatHandler (ChatServer server, Socket s) throws IOEx
13          this.s = s;
14          this.server = server;
15          i = new DataInputStream (
                    new BufferedInputStream (s.getInputStream()));
16          o = new DataOutputStream (
                    new BufferedOutputStream (s.getOutputStream()));
……...
```

JAVA Socket (7)

```
19      public void run () {
20          String name = s.getInetAddress ().toString ();
21          try {
22              name = i.readUTF ();
23              broadcast (name + " has joined.");
24              while (!stop) {
25                  String msg = i.readUTF ();
26                  broadcast (name + " - " + msg);
27              }
28          } catch (IOException ex)   {}
29          finally {
30              server.handlers.removeElement (this);
31              broadcast (name + " has left.");
32              try {
……….
```

JAVA Socket (8)

```
38      protected void broadcast (String message) {
39          synchronized (server.handlers) {
40              Enumeration e = server.handlers.elements ();
41              while (e.hasMoreElements ()){
42                  ChatHandler c = (ChatHandler)
e.nextElement ();
43                  try {
44                      synchronized (c.o) {
45                          c.o.writeUTF (message);
46                      }
47                      c.o.flush ();
48                  } catch (IOException ex) {
49                      stop = true;
50                  }
……
```

JAVA Socket (9)

예제 : ChatApplet.java

```
 ……..
 7  public class ChatApplet extends Applet implements
Runnable, ActionListener {
  ……..
18      public void init () {
19          card = new CardLayout();
20          setLayout (card);
21          Panel login = new Panel(new BorderLayout());
  ……..
29          Panel chat = new Panel(new BorderLayout());
30          chat.add ("Center", output = new TextArea ());
31          output.setEditable (false);
32          chat.add ("South", input = new TextField ());
33          input.setEditable (false);
34          input.addActionListener(this);
35          lineCount = 0;
36
37          add(login, "login");
38          add(chat, "chat");
39          card.show(this, "login");
40      }
```

JAVA Socket (10)

```
42      public void start () {
43          listener = new Thread (this); stop = false;
44          listener.start ();
45      }
46
47      public void stop () {
.....
52
53      public void run () {
54          try {
.....
62              Socket s = new Socket (host, Integer.parseInt (port));
63              i = new DataInputStream (
                    new BufferedInputStream (s.getInputStream ()));
64              o = new DataOutputStream (
                    new BufferedOutputStream (s.getOutputStream ()));
65              output.append(" connected.");
66              input.setEditable (true);
67              input.requestFocus ();
68              execute ();
69          } catch (IOException ex) {
70              ex.printStackTrace (System.out);
71          }
72      }
```

JAVA Socket (11)

```
74      public void execute () {
75          try {
76              while (!stop) {
77                  String line = i.readUTF ();
78                  if(lineCount > 50) {
79                      output.setText (lastLine + "");
80                      lineCount = 0;
81                  }
82                  output.append (line + "");
83                  lastLine = line;
84                  lineCount++;
85              }
86          } catch (IOException ex) {
87              ex.printStackTrace (System.out);
88          } finally {
.........
```

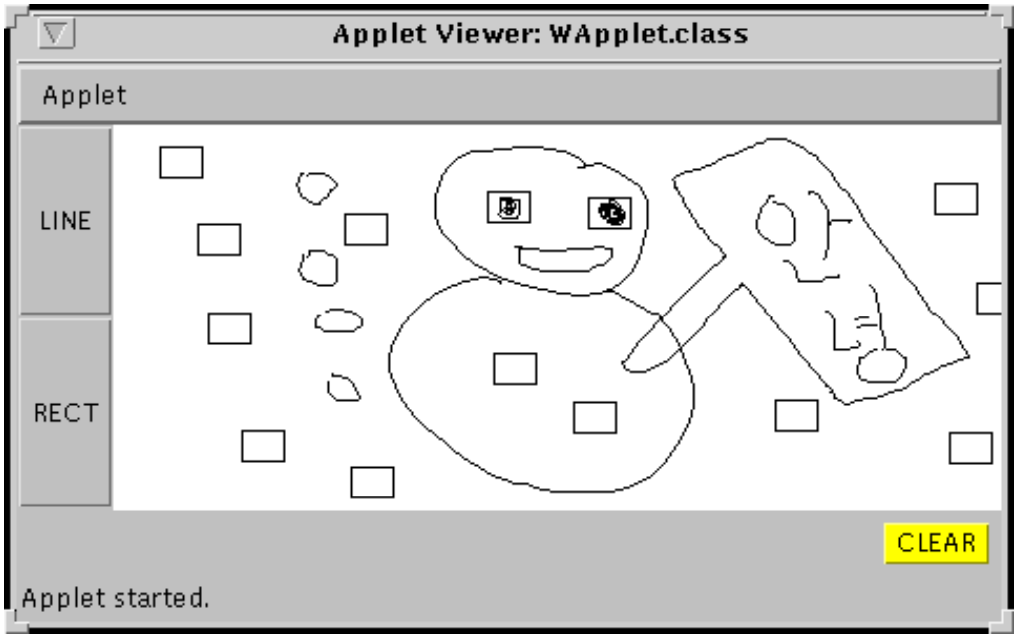
JAVA Socket (12)

```
 98      public void actionPerformed (ActionEvent e) {
 99          Component c = (Component) e.getSource();
100          if(c == logtext) {
101              String loginname = logtext.getText();
102              loginname = loginname.trim();
.........
106              try {
107                  o.writeUTF(loginname);
108                  o.flush();
109                  card.show(this, "chat");
110              } catch(IOException ex) {
111                  ex.printStackTrace(System.out);
112              }
113          } else if(c == input) {
114              try {
115                  o.writeUTF (input.getText());
116                  o.flush ();
117              } catch (IOException ex) {
.....
```

JAVA Socket (13)

화이트 보드 작성

- 채팅 프로그램을 조금 변경
- 인터넷상에서 공동으로 그림을 그릴 수 있는 화이트 보드 프로그램
  - 채팅프로그램에서는 사용자가 키입력을 하고 엔터키를 치는 경우에 메시지가 서버에 전달
  - 화이트보드에서는 마우스를 클릭하거나 드래그하는 경우에 마우스의 위치 정보를 서버측에 전달



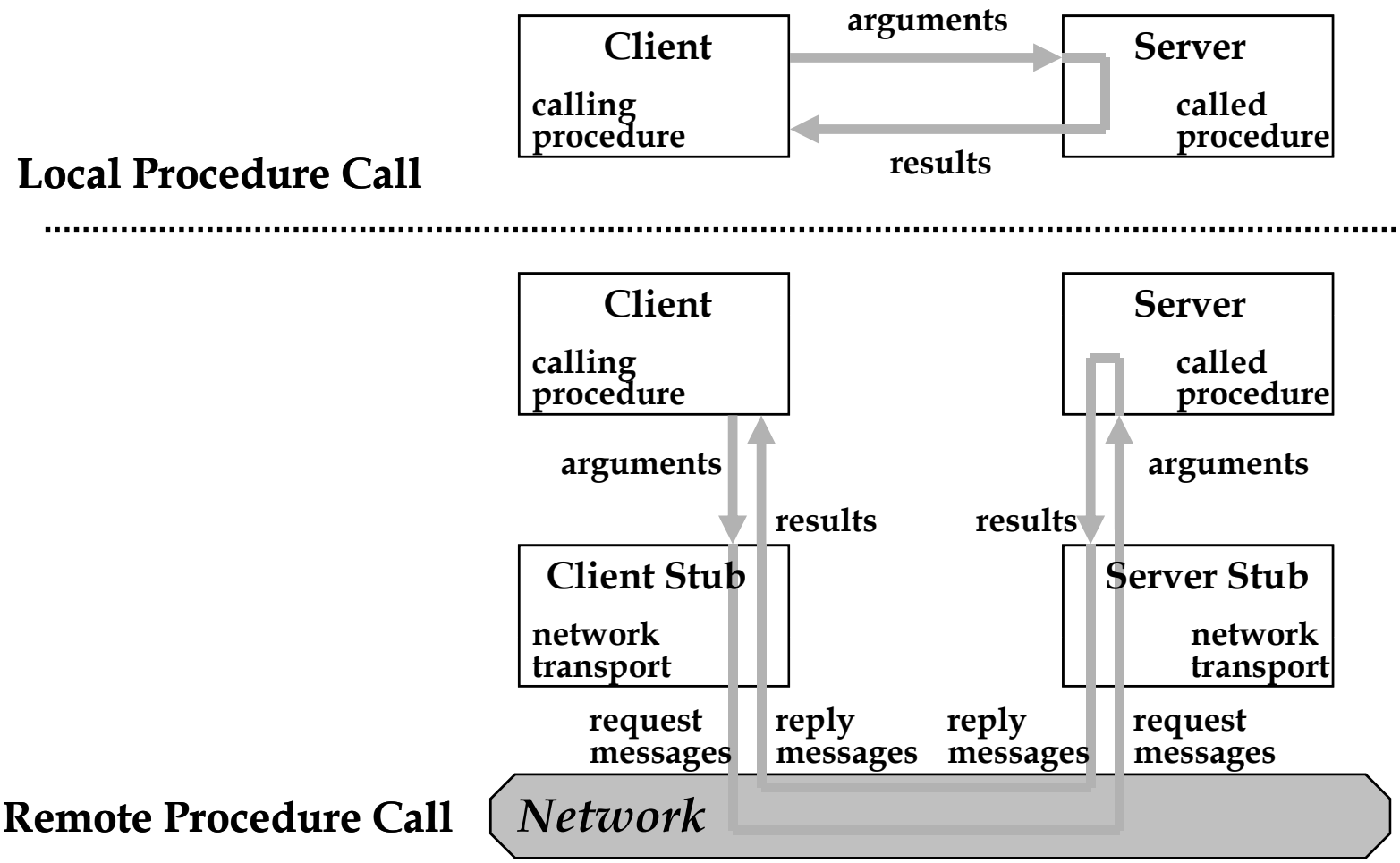
Part 5

Remote Procedure Call

RPC 란 ?

- 하나의 client가 네트워크 상의 다른 컴퓨터나 server의 procedure를 실행하는것
- 사용자 프로그래밍 툴
- socket interface보다 프로그래밍이 용이
- protocol compiler를 사용
  - 통신 프로토콜 정의
- Distributed Computing Environment 하에서 분산 응용을 개발하기 위한 toolkit

Remote & Local Procedure Call



Remote Procedure Call(1)

- 개념
  - Client-Server Model
  - stub
  - XDR(External Data Representation)

Remote Procedure Call(2)

- Client-Server Model
  - request and reply 통신모델 이용
  - client와 server는 두개의 stub에 의하여 통신

- stub
  - RPC 프로토콜을 구현하고, 메세지를 구성 및 교환하는 방법을 지정한 통신 interface
  - stub를 생성하기 위해 protocol compiler 사용

- XDR(External Data Representation)
  - machine-independent data representation
  - client & server stub에서 데이타 전송시 사용

RPC 응용을 개발하기 위한 단계

- 1 단계
  - 프로토콜 정의
    - protocol definition 규칙에 따라 client & server 사이의 interface를 정의
      - service procedure, data type of parameter, return argument
    - 정의된 프로토콜은 RPCGEN으로 compile
      - client와 server의 stub가 생성
- 2 단계
  - client & server 프로그래밍
    - client의 main 프로그램 작성
    - server의 응용 프로그램 작성

RPC & IPC 특징

- IPC(InterProcess Communication)
  - 풍부한 library
  - 개발 및 디버깅이 어려움
  - High-Performance
  - machine-dependent data representation
    - byte-ordering 문제
- RPC
  - 개발 용이
  - Low-Performance
  - machine-independent data representation

RPC Products

- Sun ONC(Open Network Computing) RPC
  - RPCGEN, RPC Runtime, XDR(External Data Representation)
- Apollo NCS(Network Computing System) RPC
  - Object Oriented model
  - IDL(Interface Definition Language), RPC runtime, NDR(Network Data Representation), ASN.1 BER
- OSF DCE RPC
  - based Apollo NCS RPC
  - added Security, Directory, Thread

High-level RPC 응용 개발

- high-level function을 이용
  - UDP transport protocol 사용
- client & server의 interface를 직접 구현
- protocol compiler를 사용하지 않고 프로그램 가능
- TCP를 사용하기 위해서는 low-level function을 사용
- 개발 과정
  - 프로토콜 정의, client & server 프로그램 작성

프로토콜 정의

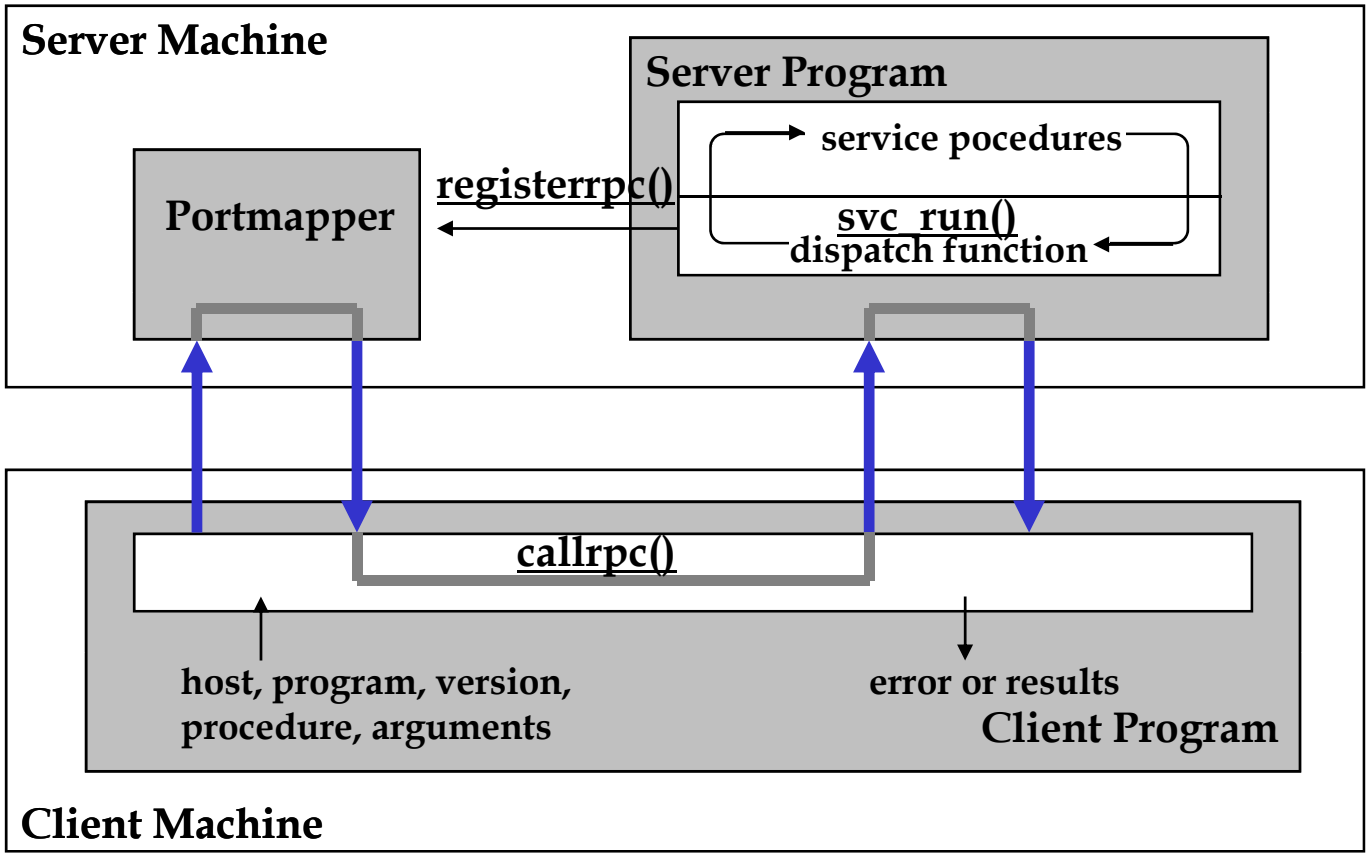
- Data Types
  - ex) char, int, double etc.
- Program, Procedure, Version Numbers
  - ex)

```
#define DIR_SIZE 8192
#define DIRPROG ((u_long) 0x20000000) /* server
  program # */
#define DIRVERS ((u_long) 1) /* program version
  number */
#define READDIR ((u_long) 1) /* procedure num for
  look-up */
```

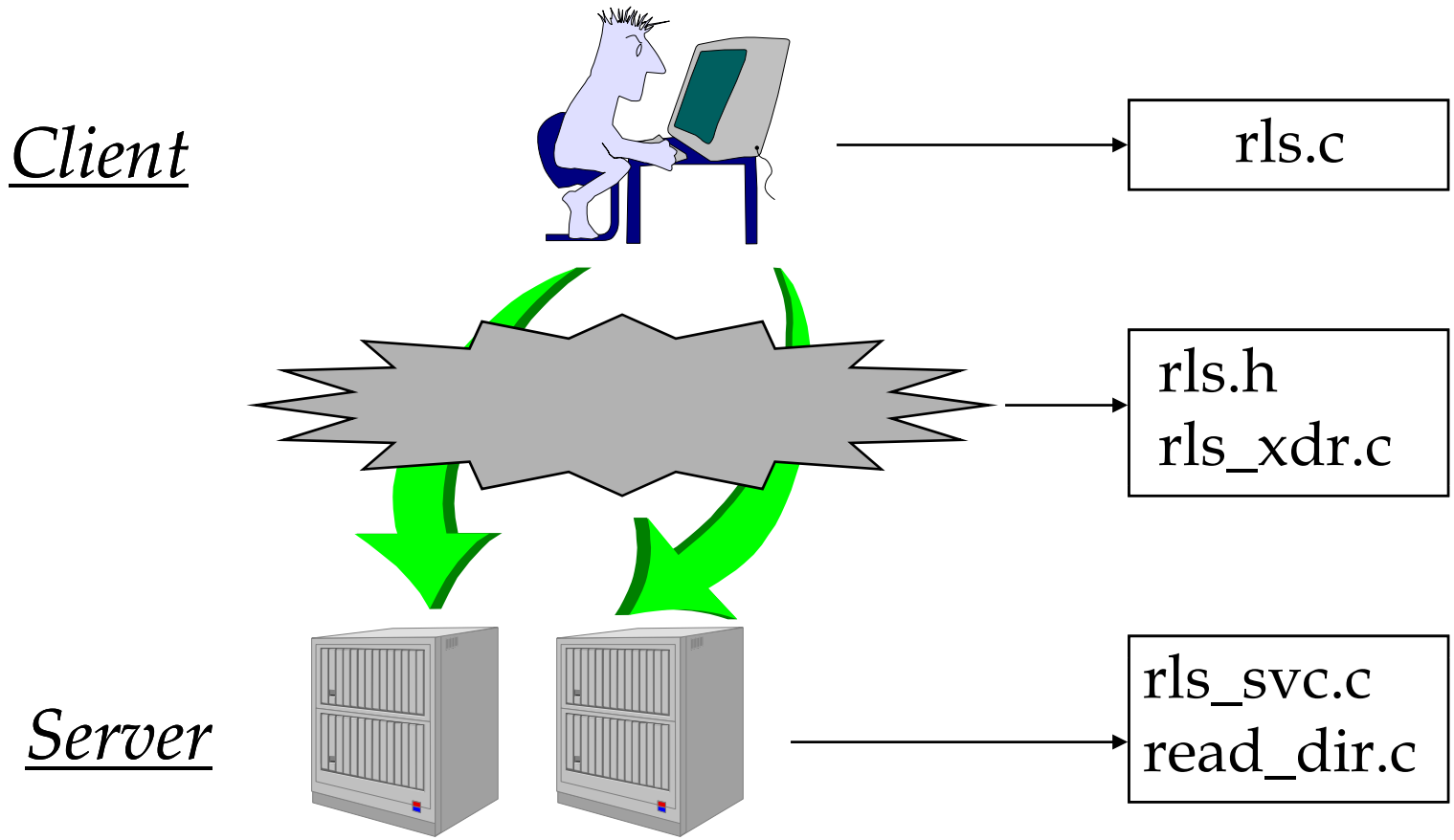
High-level functions

- client
  - callrpc()
    - int callproc(host, prognum, versnum, procnum, inproc, in, outproc, out)
- server
  - registerpc()
    - #include <rpc/rpc.h>
    - int registerrpc(prognum, versum, procnum, proname, inproc, outproc)
  - svc_run()
    - void svc_run()

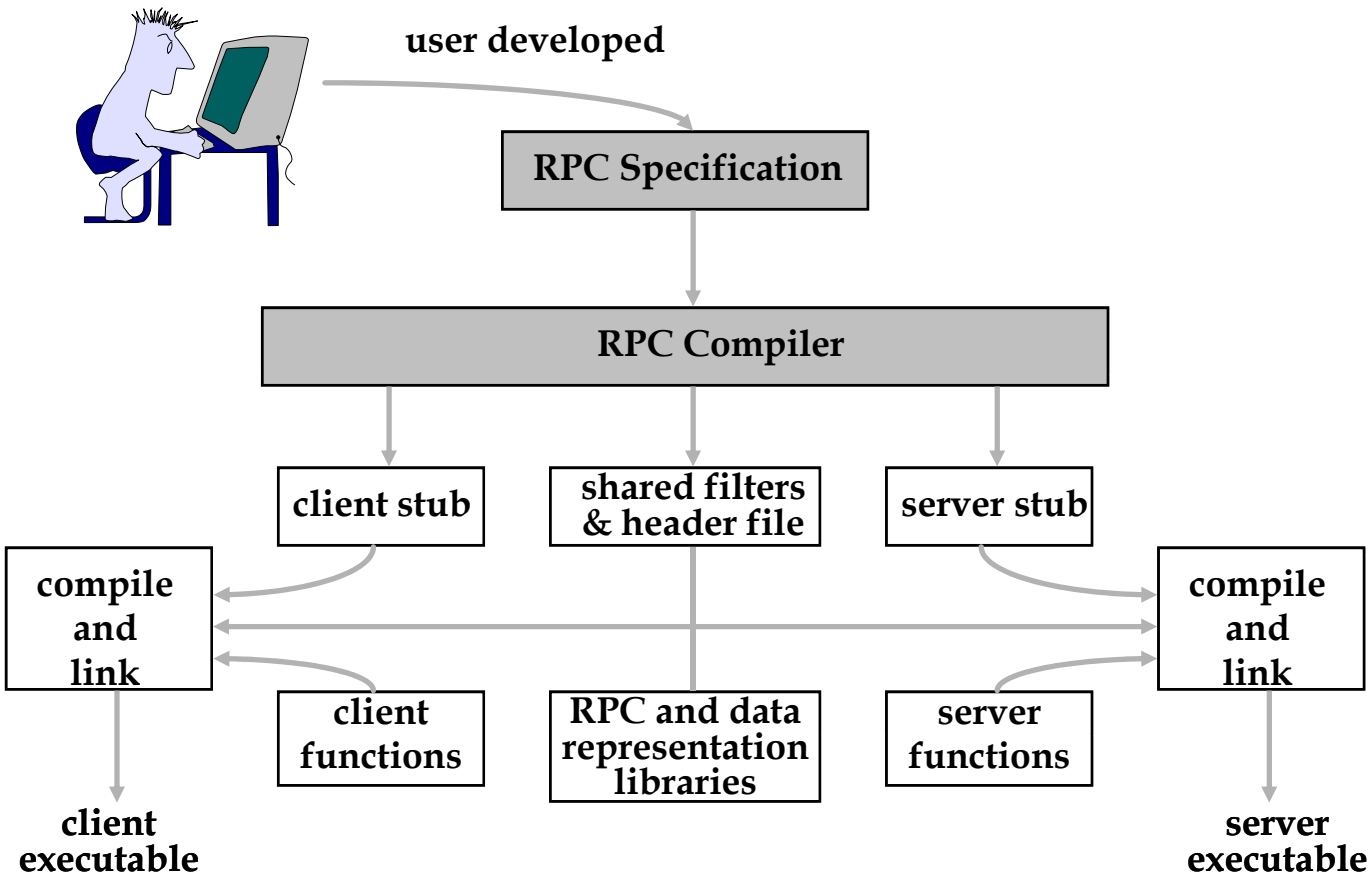
High-level RPC



Example Program



Protocol Compiler & Lower-level RPC Programming



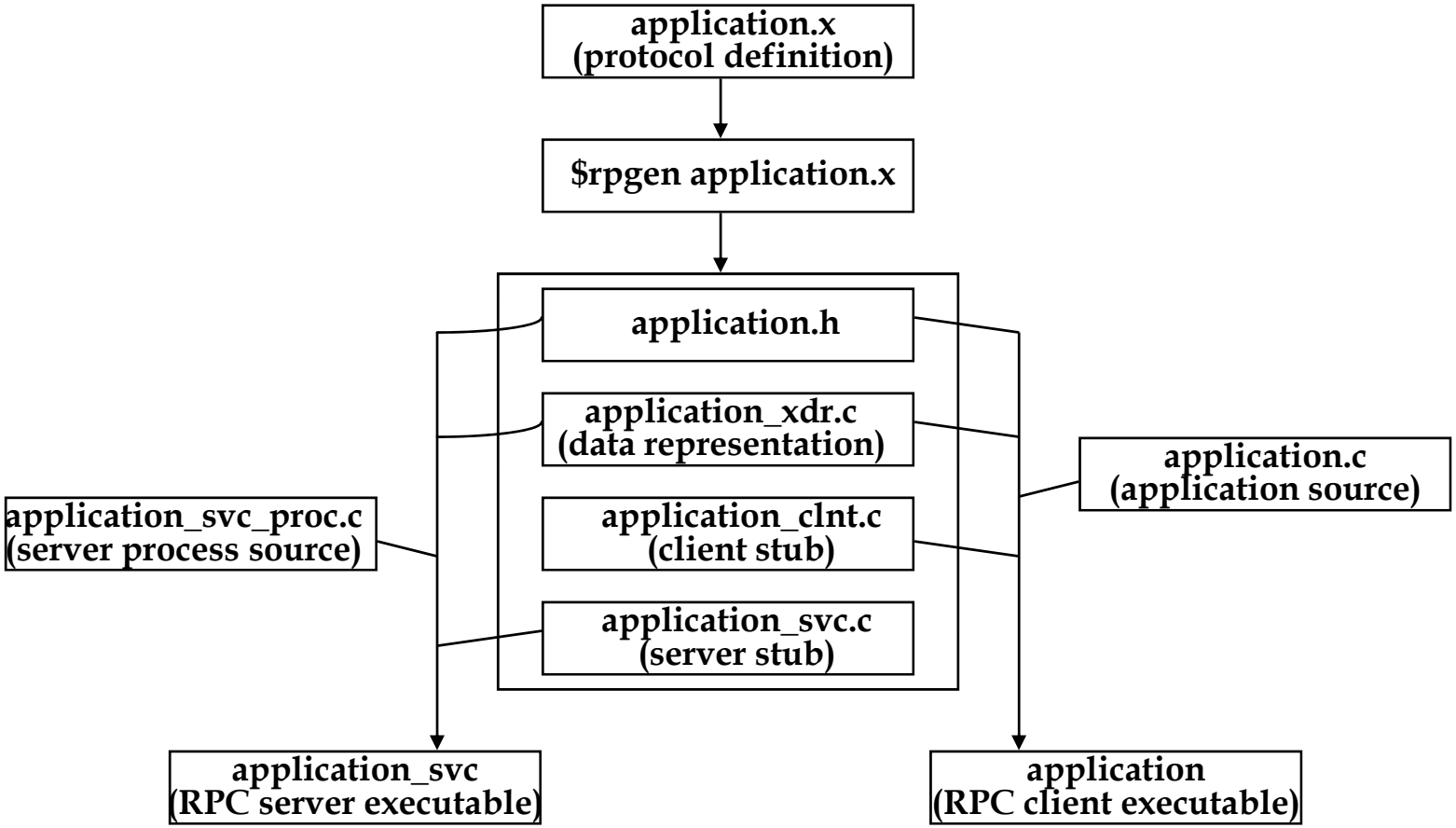
Low-level RPC 응용 프로그램

- 프로토콜 정의 프로그램(application.x)
  - RPCGEN으로 프로토콜 정의 프로그램 compile
    - XDR routine 생성(application_xdr.c)
    - 프로토콜 정의에 대한 include file 생성 (application.h)
    - client stub 생성(application_clnt.c)
    - server stub 생성(application_svc.c)

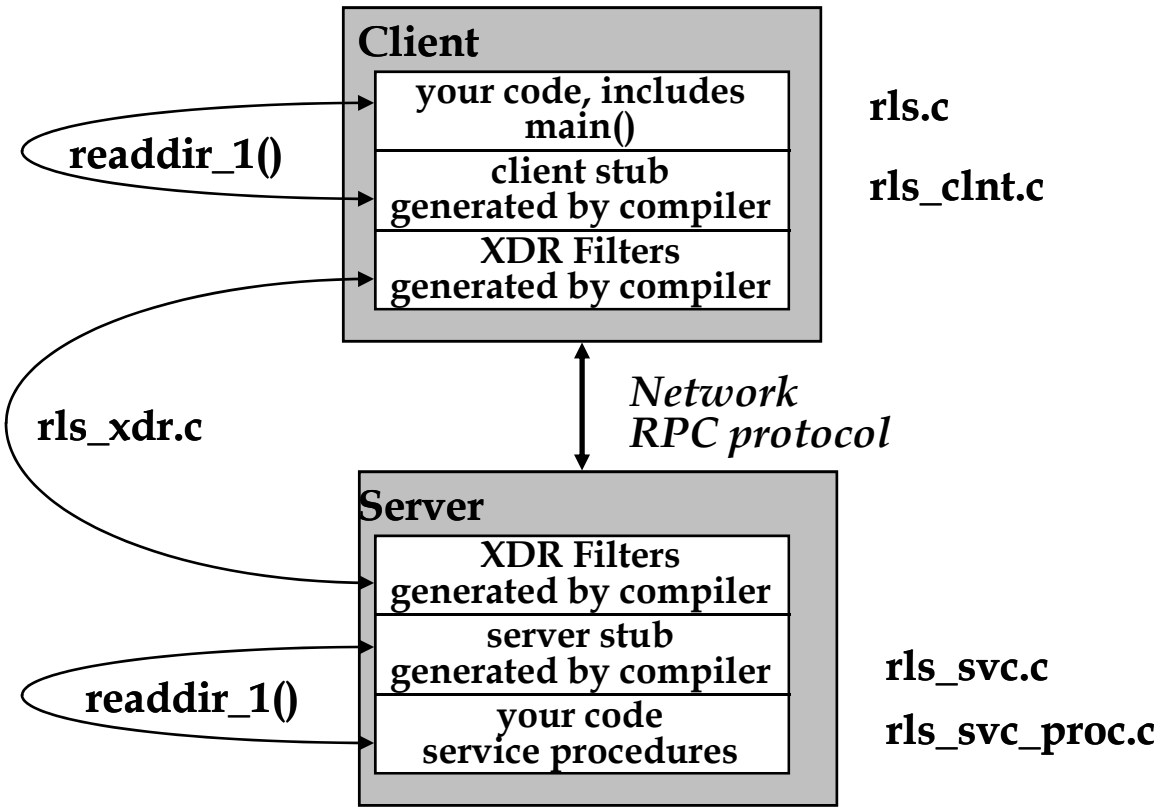
- 서버 서비스 프로그램(application_svc_proc.c)
  - server stub의 dispatch 루틴에 의해서 호출될 함수

- 클라이언트 프로그램(application.c)

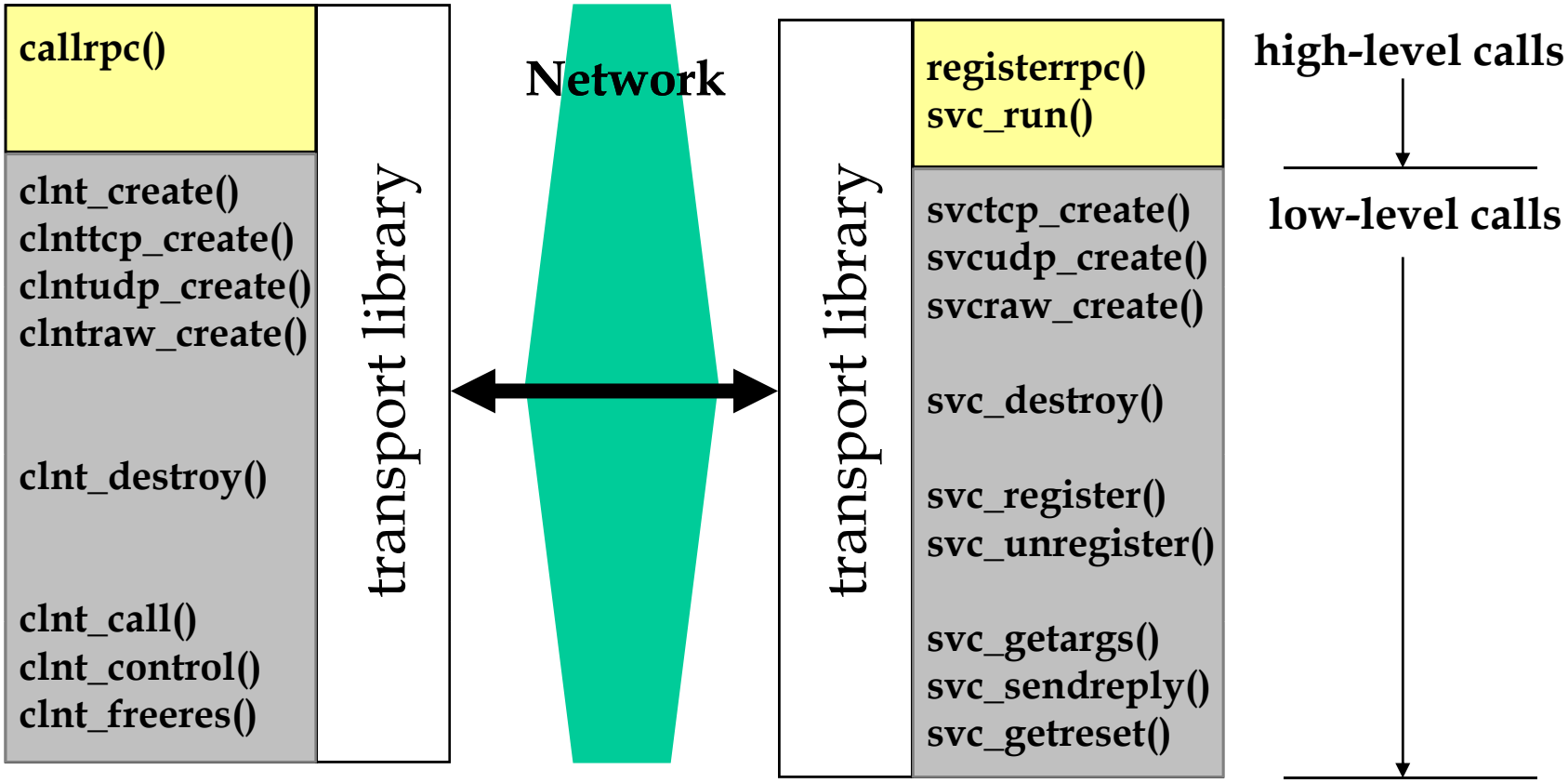
RPCGEN을 이용한 응용 개발



Example Program



High-level & Low-level calls

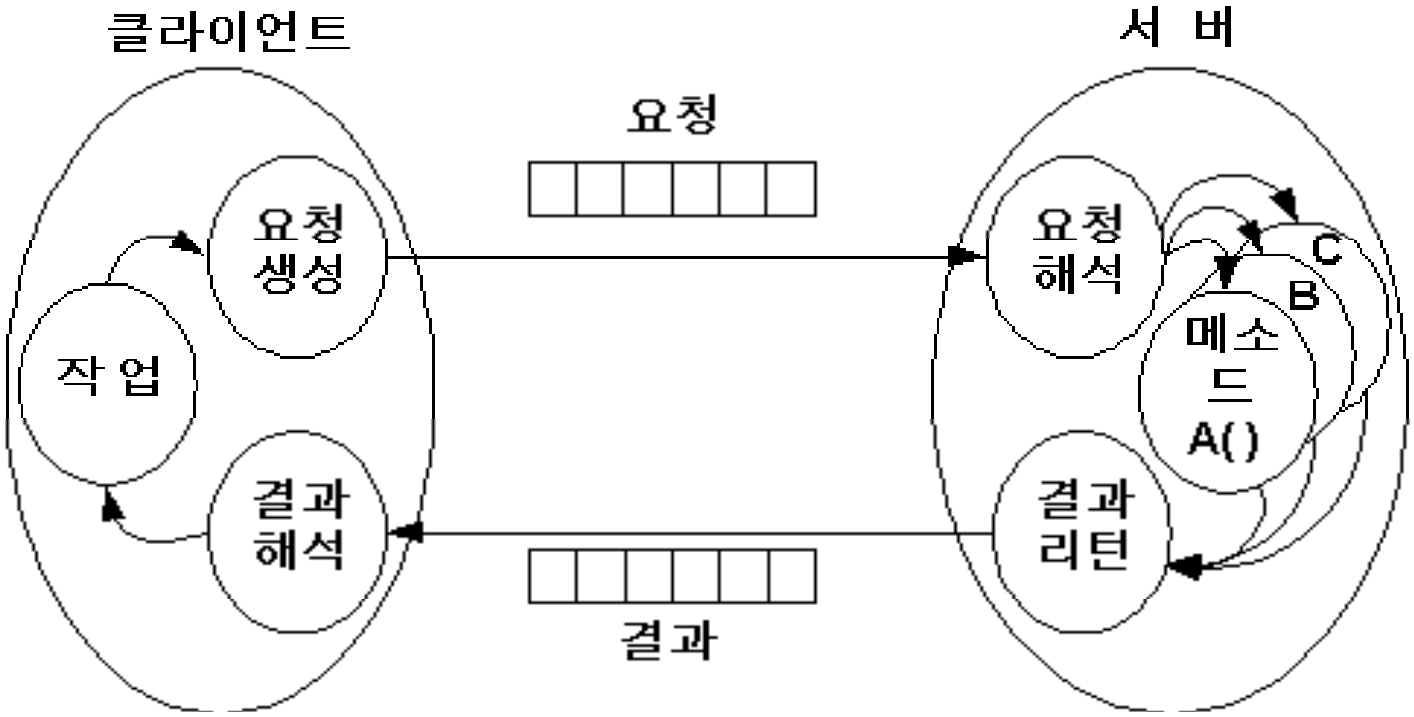


Part 6

Java RMI

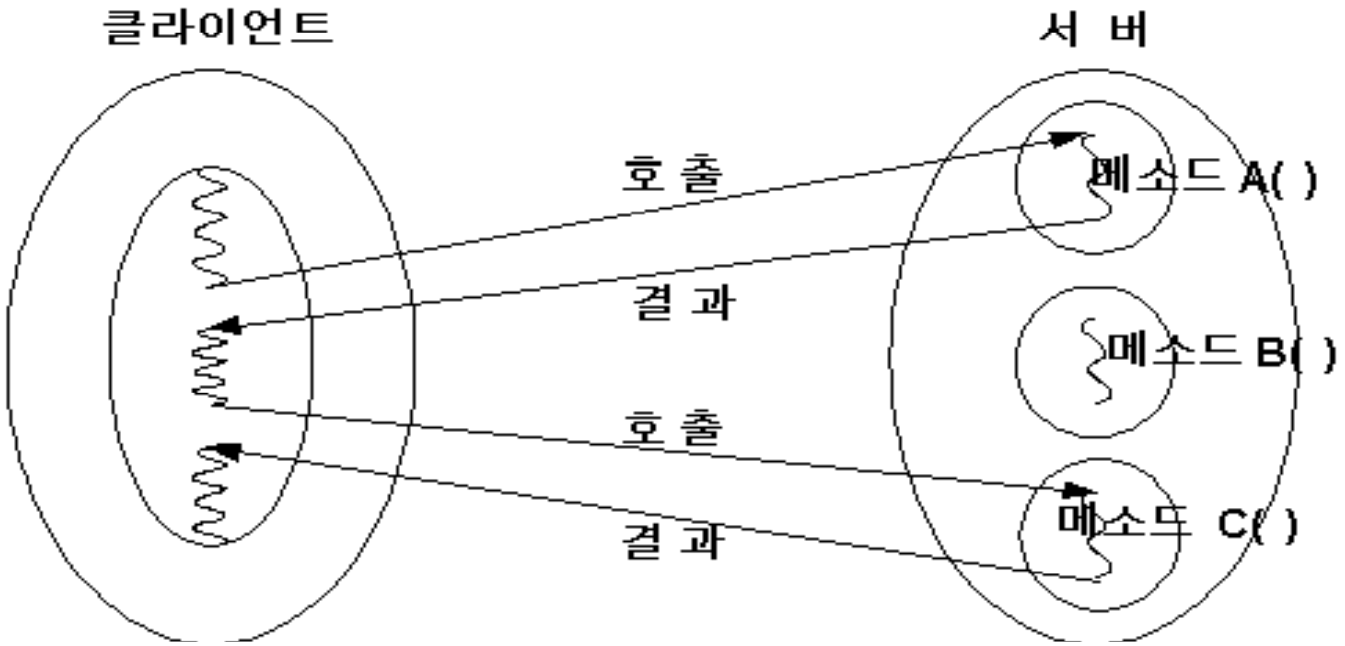
1. RMI 기초(1)

- 분산 컴퓨팅
  - 소켓 사용
    - 저수준 프로그래밍
    - 사용하기 어려움



1. RMI 기초(2)

- 분산 컴퓨팅
  - 분산 객체 사용 – RMI, CORBA
    - 고수준 프로그래밍
    - 사용이 편리



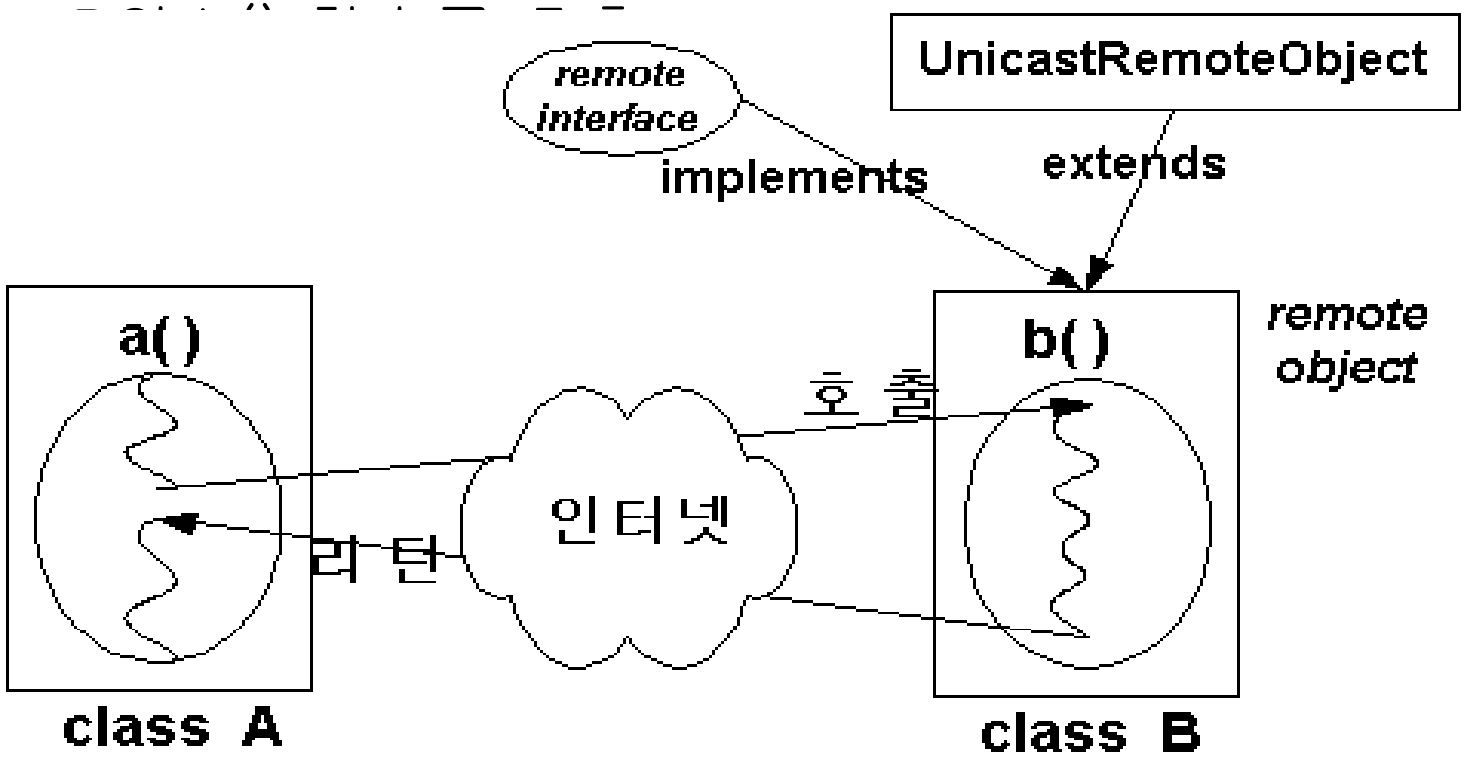
1. RMI 기초(3)

- RMI의 장점
  - 객체지향적.
  - 프로그램 작성 및 사용의 용이성.
  - 보안성.
  - 기존 시스템과 통합.

- RMI의 기본적인 목적
  - 원격 객체(remote object)의 메소드를 호출할 수 있는 방법 제공
  - **원격 객체** – 다른 자바 가상 머신(일반적으로 다른 컴퓨터에 있는)에서 호출할 수 있는 메소드를 가지는 객체
  - 원격 객체는 UnicastRemoteObject 클래스로부터 상속받고, 원격 인터페이스를 임플리멘츠 한다.

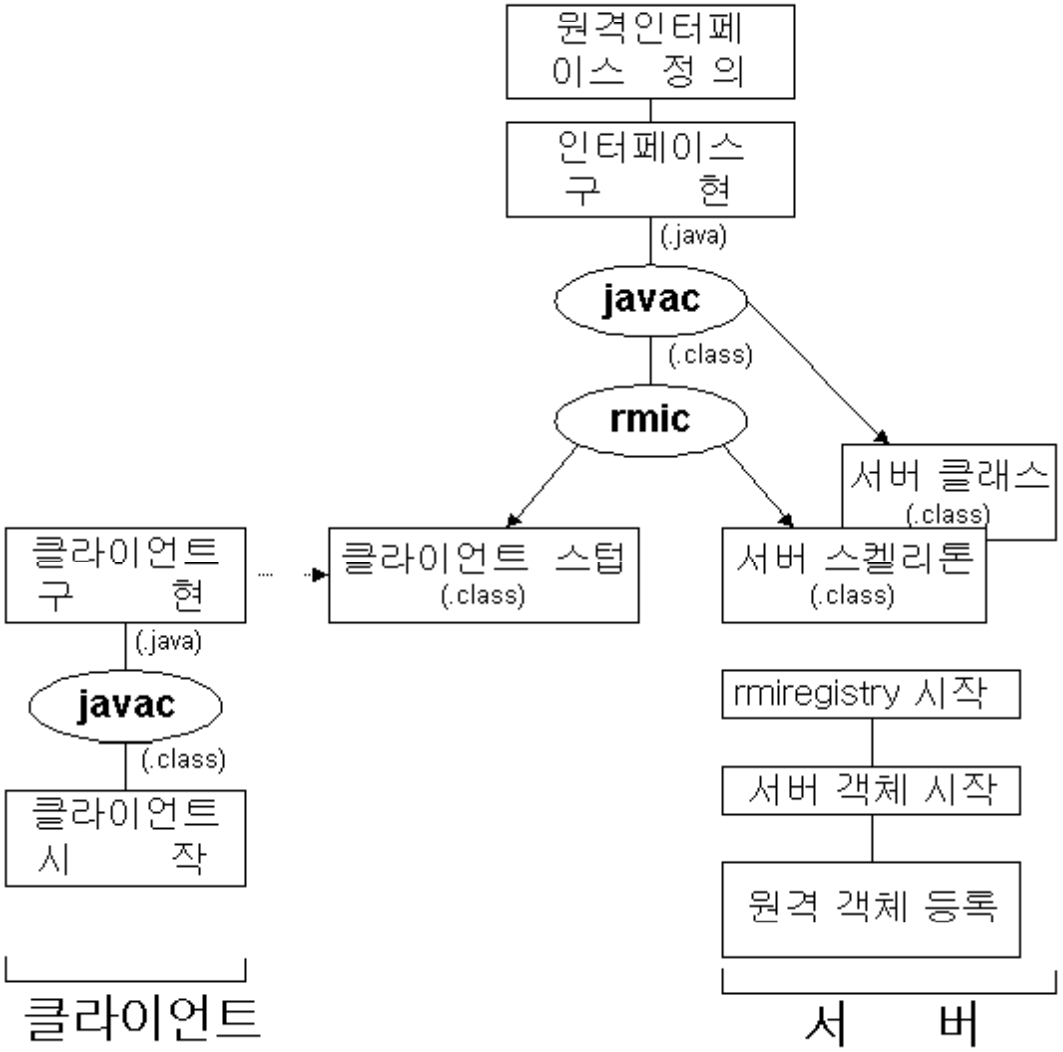
1. RMI 기초(4)

- 클라이언트와 원격 객체의 관계
  - class A – 클라이언트
  - class B – 원격객체(서버)
  - 클라이언트 class A의 함수 a()는 원격 객체 class



1. RMI 기초(5)

- 자바 RMI를 이용한 클라이언트/서버 프로그램의 작성 단계



1. RMI 기초(6)

- RMI 프로그래밍 절차
  1. 원격 인터페이스를 정의한다.
  2. 원격 인터페이스를 임플리멘츠 하는 원격 객체(서버)를 작성한다.
  3. 원격 객체를 이용하는 프로그램(클라이언트)을 작성한다.
  4. stub과 skeleton 클래스를 생성한다.
  5. registry를 실행시킨다.
  6. 서버와 클라이언트를 실행시킨다.

- 원격 인터페이스
  - 사용되는 패키지 - java.rmi.remote, java.rmi.server
  - 모든 원격 인터페이스는 Remote 인터페이스로부터 상속 받아야 한다.
  - Remote 인터페이스는 메소드가 선언되지 않은 인터페이스이다.
    - public interface Remote { }
  - RemoteObject – RMI 환경에서 가장 상위 클래스

1. RMI 기초(7)

- RMI 구조 이해
  - TCP/IP 바탕
  - 3 계층 구조
    - Stub/Skeleton 계층 - 마샬 스트림
    - Remote Reference 계층 - 객체의 특성 파악
    - Transport 계층 - 네트워 전송

  - RMI에서 원격 객체의 메소드를 호출할 때 메소드의 아규먼트와 리턴 값이 네트워크를 통해 전달된다.
  - 네트워크를 통해 전송할 때 메소드의 아규먼트와 리턴 값을 바이트 스트림으로 변경해서 전달하게 되는데, 이것을 마샬 스트림(marshal stream)이라고 한다.
  - 마샬 스트림을 만드는 과정을 마샬링(marshaling)이라고 반대 과정은 언마샬링(unmarshaling)이라고 한다.

2. 자바 RMI 예제 (1)

1. sayHello()라는 메소드를 갖는 Hello라는 원격 인터페이스를 정의한다.

예제 : Hello.java

```
1  package hello;
2
3  import java.rmi.*;
4
5  public interface Hello extends Remote {
6
7                    public String sayHello() throws
   java.rmi.RemoteException;
8
9 }
```

- 참고
  1) 원격 인터페이스는 java.rmi.Remote 인터페이스로부터 상속받는다.
  2) 원격 인터페이스는 public으로 선언되어야 한다.

2. 자바 RMI 예제 (2)

2. Hello 인터페이스를 임플리멘츠한 HelloImpl이라는 원격 객체를 만든다.

- 예제 : HelloImpl.java

```
 1  package hello;
 2
 3  import java.rmi.*;
 4  import java.rmi.server.*;
 5  import java.net.*;
 6
 7  public class HelloImpl extends UnicastRemoteObject implements Hello {
 8
 9      public HelloImpl() throws RemoteException {         super();  }
12
13      public String sayHello() throws RemoteException {
                return "Hello World";
16      }
17
18      public static void main(String[] args) {
19          System.setSecurityManager(new RMISecurityManager());
20
21          try {
22              HelloImpl h = new HelloImpl();
23              Naming.rebind("rmi:/203.252.201.16:9999/hello", h);
24              System.out.println("Hello Server ready");
25          } catch (RemoteException re) {
26              System.out.println("Exception in HelloImpl.main: " + re);
27          } catch (MalformedURLException mfe) {
28              System.out.println("MalformedURLException in HelloImpl.main" + mfe);
29          }
30      }
31 }
```

2. 자바 RMI 예제 (3)

- bind와 rebind
  - rebind() 메소드와 bind() 메소드는 원격 객체를 등록하는데 사용되지만, 차이점은 bind() 메소드는 동일한 이름이 이미 등록되어 있으면 java.rmi.AlreadyBoundException을 발생시키지만, rebind()는 이미 등록된 객체 대신에 새로운 객체로 대치한다. 두 메소드 모두 아규먼트로 RMI URL와 인스턴스 이름이 사용된다. RMI URL은 다음과 같은 형태로 되어있다.
    - 형 태

      rmi://host:port/object_name

    - 따라서 예제 프로그램처럼 간단히 hello라고 이름만 기술한 경우는
      - 예 :

        Naming.rebind("rmi://localhost:1099/hello", h);

        와 동일하다. RMI는 기본적으로 1099 포트를 사용한다.

- 참고
  - 1. 원격 객체는 원격 인터페이스에 선언되지 않은 메소드들도 정의할 수 있지만 원격 인터페이스에 선언되지 않은 메소드는 클라이언트에서 호출할 수 없다.
  - 2. 원격 객체의 생성자와 메소드들은 RemoteException을 throws해야 한다.

2. 자바 RMI 예제 (4)

- 컴파일하기
  % javac -d . Hello.java HelloImpl.java

- 3. HelloImpl 원격 객체를 이용하는 프로그램(클라이언트)을 작성한다.
- 예제 : HelloClient.java

```
1  package hello;
2
3  import java.rmi.*;
4
5  public class HelloClient {
6
7      public static void main(String[] args) {
8
9                              System.setSecurityManager(new
RMISecurityManager());
10
11          try {
```

2. 자바 RMI 예제 (5)

```
12            Hello h = (Hello)
Naming.lookup("rmi://203.252.201.16/hello");
13
14            String message = h.sayHello();
15            System.out.println("HelloClient: " +message);
16        } catch(Exception e) {
17            System.out.println("Exception in main: "+ e);
18        }
19    }
20 }
```

- 4. rmic를 이용해서 stub과 skeleton 클래스를 만들고, jar 파일을 만든다.

```
% rmic -d . hello.HelloImpl
% jar cvf hello.jar hello/*.class
```

참고

1) rmic는 stub과 skel을 생성하는 컴파일러이다.

12.2 자바 RMI 예제

- 5. rmiregistry를 등록한다.
  % unsetenv CLASSPATH
  % rmiregistry &
  [1] 7382

- 6. 보안 설정 파일을 생성한다.
- 예제 : java.policy

```
1  grant{
2      permission java.net.SocketPermission "*:1024-
65535",
3          "connect,accept";
4      permission java.net.SocketPermission "*:80",
"connect";
5  };
```

- 7. rmi 서버를 가동시킨다.
  % java -Djava.security.policy=java.policy hello.HelloImpl

2. 자바 RMI 예제 (6)

- 7. 클라이언트 프로그램을 실행시킨다.

  % java -Djava.security.policy=java.policy hello.HelloClient
  HelloClient: Hello World

- RMI 클라이언트를 애플릿으로 작성한 예제
- 예제 : HelloApplet.java

```
1  package hello;
2
3  import java.awt.*;
4  import java.applet.*;
5  import java.rmi.*;
6
7  public class HelloApplet extends Applet {
8      String message = "";
9
10     public void init() {
```

2. 자바 RMI 예제 (7)

```
 11          try {
 12              Hello obj =
(Hello)Naming.lookup("//"+getCodeBase().
getHost() + "/hello");
 13              message = obj.sayHello();
 14          } catch(Exception e) {
 15              System.out.println("HelloApplet
exception: " + e.getMessage());
 16          }
 17      }
 18
 19      public void paint(Graphics g) {
 20          g.drawString(message, 25, 50);
 21      }
 22 }
```

2. 자바 RMI 예제 (8)

- 예제 : Hello.html

```
1  <html><title>Hello World</title>
2  <applet code=hello.HelloApplet archive=hello.jar
   width=400 height=400>
3  </applet>
4  </html>
```

- 컴파일하기

```
% javac -d . HelloApplet.java
% jar cvf hello.jar hello/*.class
```

- 결과

```
% appletviewer  Hello.html
```

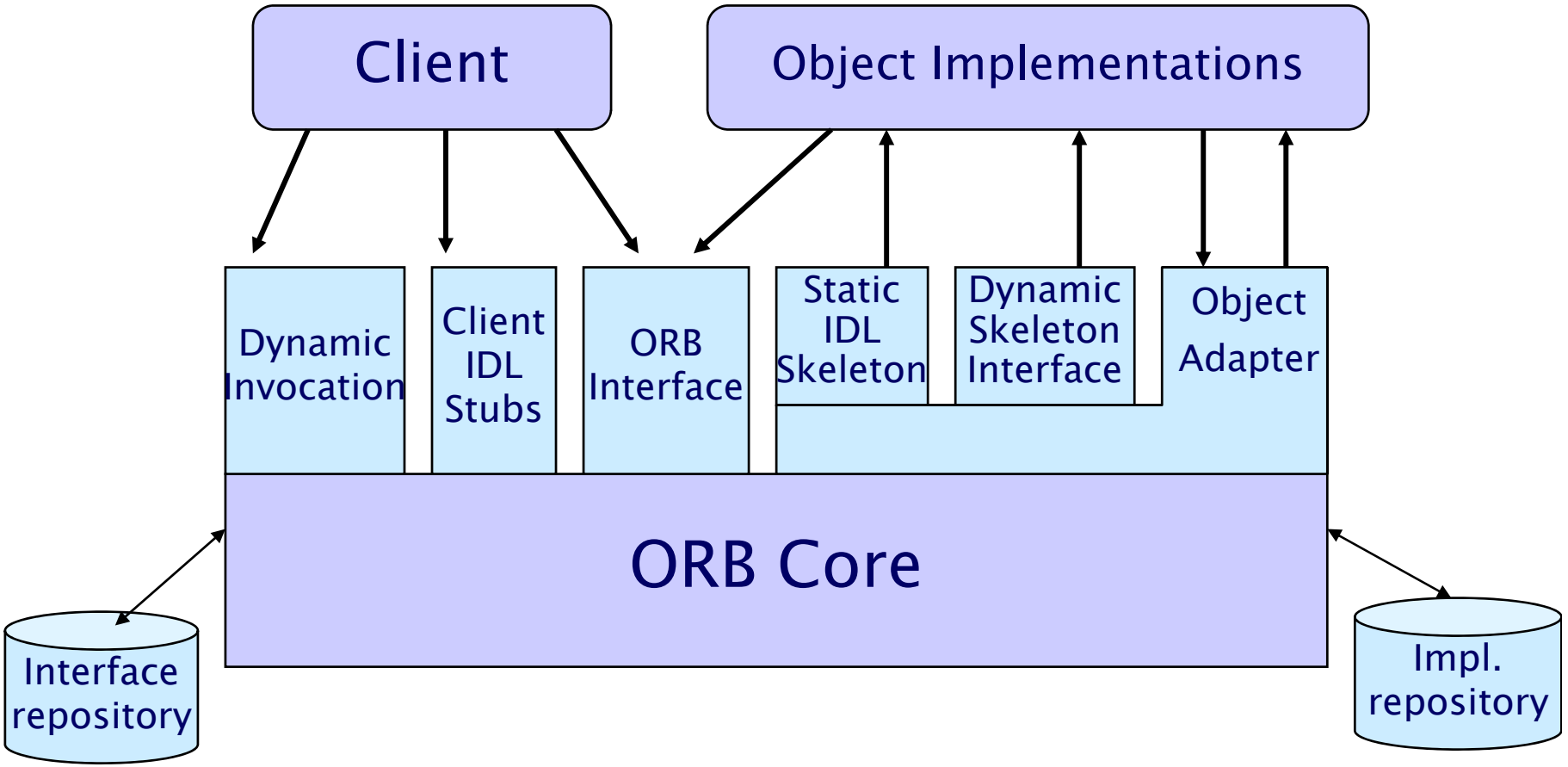
Part 7

CORBA 개요 및 구현환경

CORBA 개요

- 객체 관리 그룹 (OMG : Object Management Group)
  – 객체지향기술과 이기종 분산환경에서 애플리케이션 개발에 필요한 하부구조의 기술규격 규정을 목적으로 하는 단체
- 객체 관리 아키텍처(OMA : Object Management Architecture)
- CORBA (Common Object Request Broker Architecture)
  : ORB의 표준화

CORBA 구조


Client
Object Implementations
Dynamic Invocation
Client IDL Stubs
ORB Interface
Static IDL Skeleton
Dynamic Skeleton Interface
Object Adapter
ORB Core
Interface repository
Impl. repository

CORBA 구조

- ORB Core
  - 클라이언트/구현객체 통신의 기반이 되는 컴포넌트
- Interface Repository
  - 구현객체의 함수 파라미터와 복귀값 등의 인터페이스 정보를 관리
- Implementation Repository
  - 구현객체가 있는 위치와 저장방법을 관리

CORBA IDL

- IDL(Interface Definition Language)

: OMG에서 규정한 인터페이스 정의 언어 표준

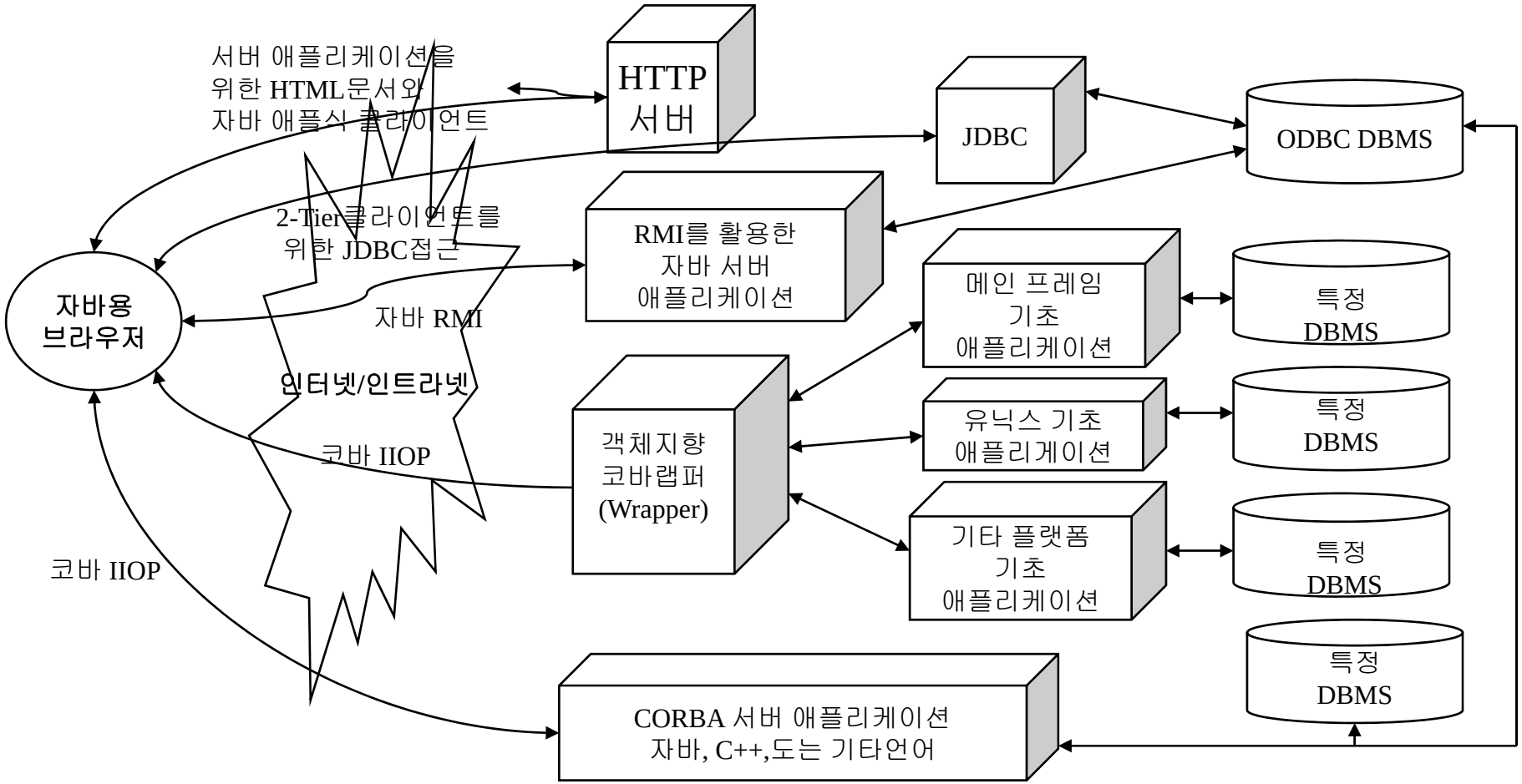
Client와 Server 사이에 필요한 객체를 추출하고

인터페이스 정의

-사용되는 객체에 대한 인터페이스 정의

-속성값, 연산, 각 연산의 매개변수 값을 정의

Java & CORBA 모델



=> RMI : 순수 자바 애플리케이션을 위한 뛰어난 분산 컴퓨팅 모델

=> Java ——————————— 기존 애플리케이션
Interface(CORBA, OMA가 매카니즘 제공)

CORBA를 이용한 Java application 의 아키텍처