Chapter 08 :
Elementary UDP Sockets

Index

- 8.1. Introduction
- 8.2. *recvfrom* and *sendto* Functions
- 8.3. UDP Echo Server : *main* Function
- 8.4. UDP Echo Server : *dg_echo* Function
- 8.5. UDP Echo Client : *main* Function
- 8.6. UDP Echo Client : *dg_cli* Function
- 8.7. Lost Datagrams
- 8.8. Verifying Received Responce
- 8.9. Server Not Running
- 8.10. Summary of UDP Example
- 8.11. *connect* Function with UDP
- 8.12. *dg_cli* Function (Revisited)
- 8.13. Lack of Flow Comtrol with UDP
- 8.14. Determining Outgoing Interface with UDP
- 8.15. TCP and UDP Echo Server Using *select*
- 8.16. Summary

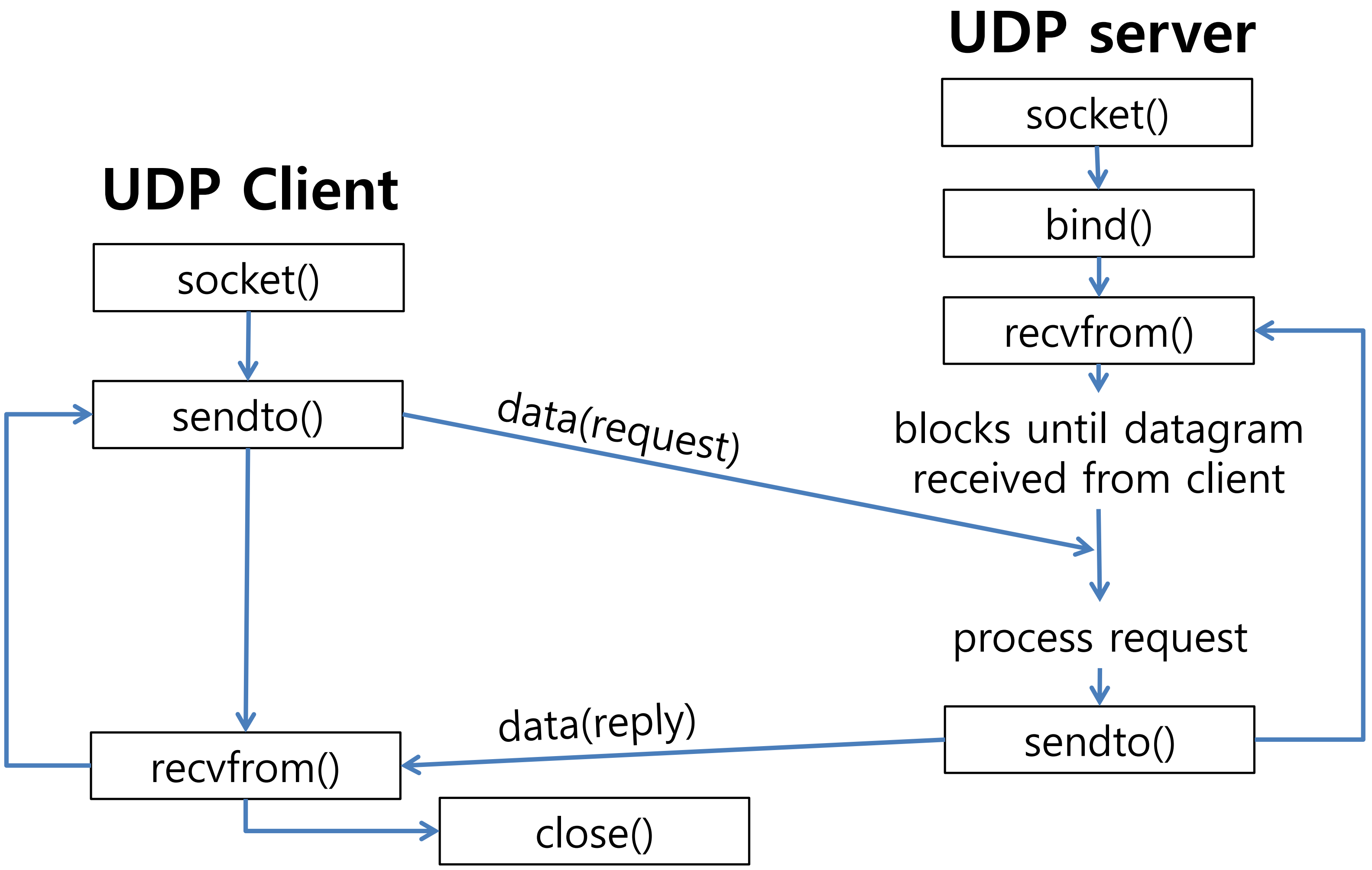
Introduction

- **Differences between TCP & UDP**

TCP	UDP
connection-oriented	connectionless
reliable	unreliable
byte stream	datagram protocol

- **Applications built with UDP : DNS, NFS, SNMP**

Introduction



recvfrom, sendto Functions

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

```
ssize_t recvfrom(int sockfd, void* buff, size_t nbytes,
                 int flags, struct sockaddr* from,
                 socklen_t* addrlen)
```

```
ssize_t sendto(int sockfd, void* buff, size_t nbytes,
               int flags, const struct sockaddr* to,
               socklen_t addrlen)
```

- **New arguments :** ***flags, from/to, addrlen***
- **Caution :** recvfrom()'s ***addrlen*** is a **pointer type**

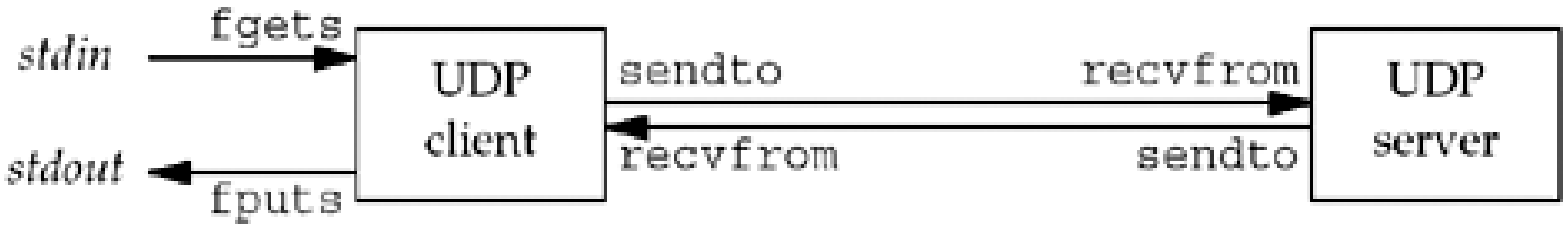
recvfrom, sendto Functions

- **int *flags***
  - Handle this argument at Chapter 14.
  - *recv, send, recvmsg, sendmsg*
  - For now, just set the *flags* to 0.

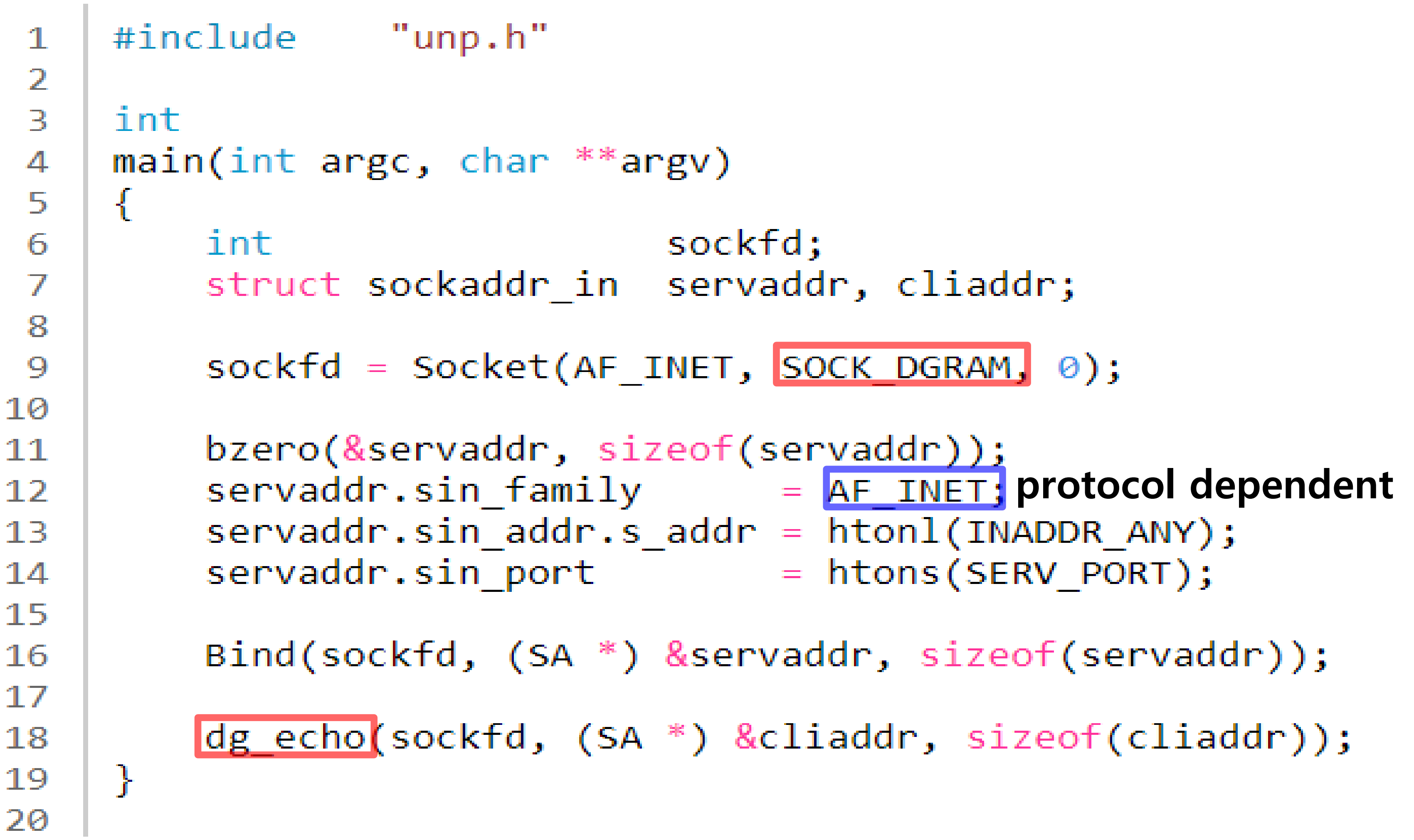
- **struct sockaddr* *from* (*to*)**
  - Contains protocol address (IP, port number)
  - Size is specified by *addrlen*

UDP Echo Server

- **UDP version of Echo Server in Chapter 5.**



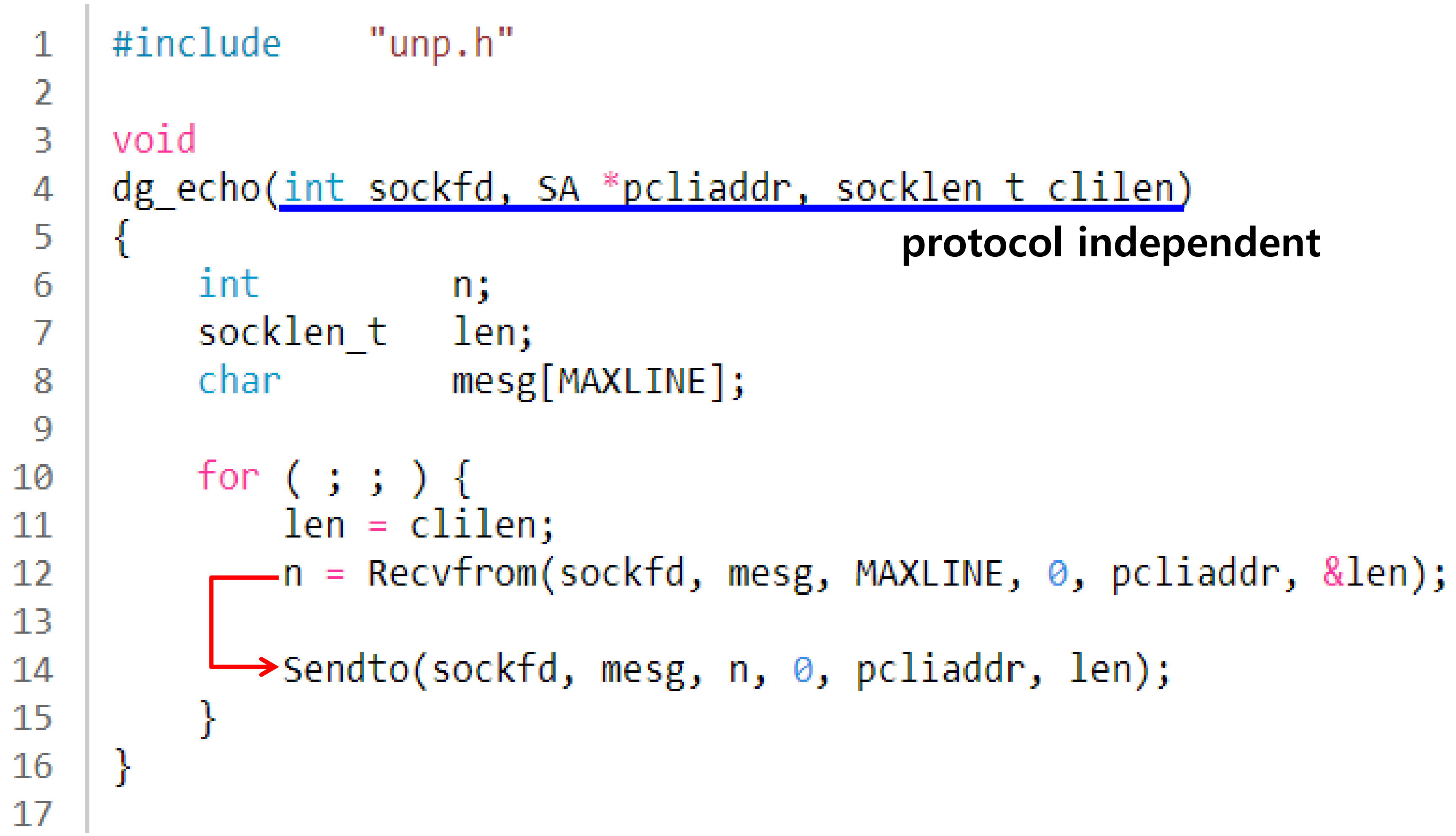
UDP Echo Server



```c
#include    "unp.h"

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

    sockfd = Socket(AF_INET, SOCK_DGRAM, 0);

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

    Bind(sockfd, (SA *) &servaddr, sizeof(servaddr));

    dg_echo(sockfd, (SA *) &cliaddr, sizeof(cliaddr));
}
```

protocol dependent

UDP Echo Server



```c
#include    "unp.h"

void
dg_echo(int sockfd, SA *pcliaddr, socklen_t clilen)
{
    int         n;
    socklen_t   len;
    char        mesg[MAXLINE];

    for ( ; ; ) {
        len = clilen;
        n = Recvfrom(sockfd, mesg, MAXLINE, 0, pcliaddr, &len);

        Sendto(sockfd, mesg, n, 0, pcliaddr, len);
    }
}
```

protocol independent

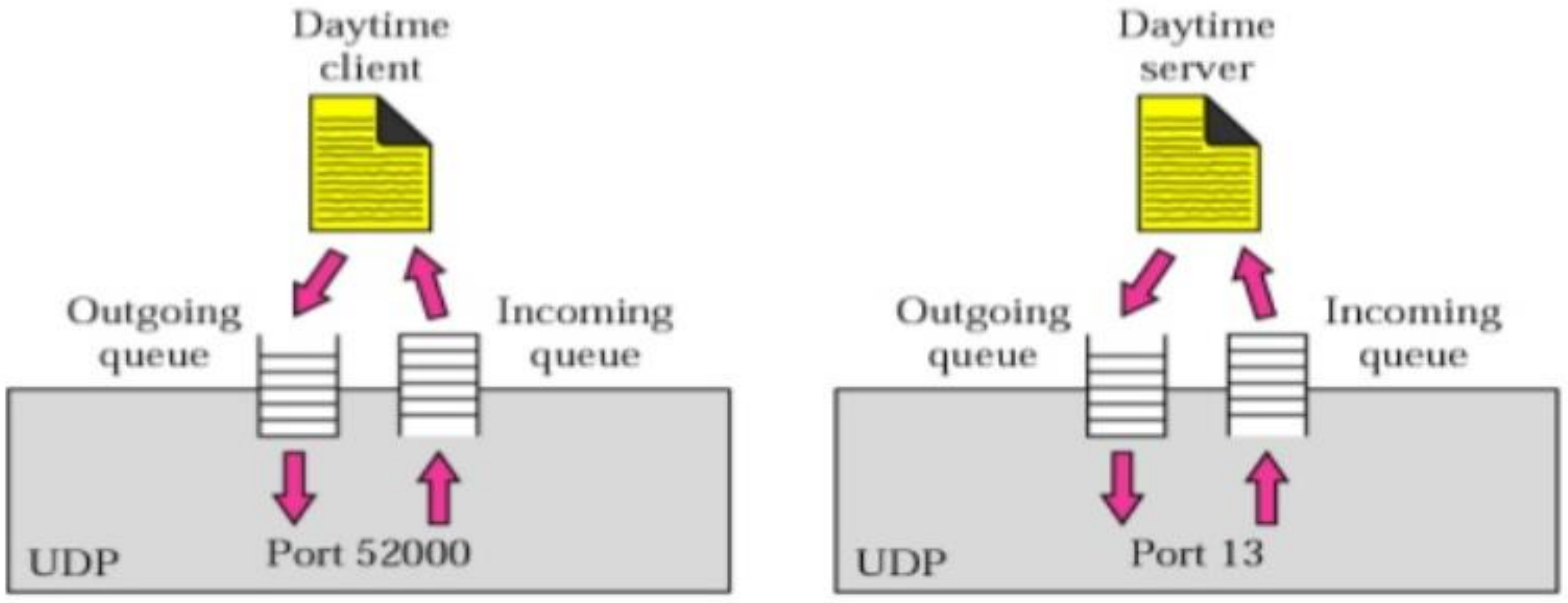
UDP Echo Server

- **Details of *dg_echo***
  1. This function never terminates
     - Since UDP is connectionless, there is nothing like an EOF as we have like TCP.

  2. This function provides an *iterative server.*
     - There is no call to *fork*, so a single serer process handles any and all clients.

UDP Echo Server

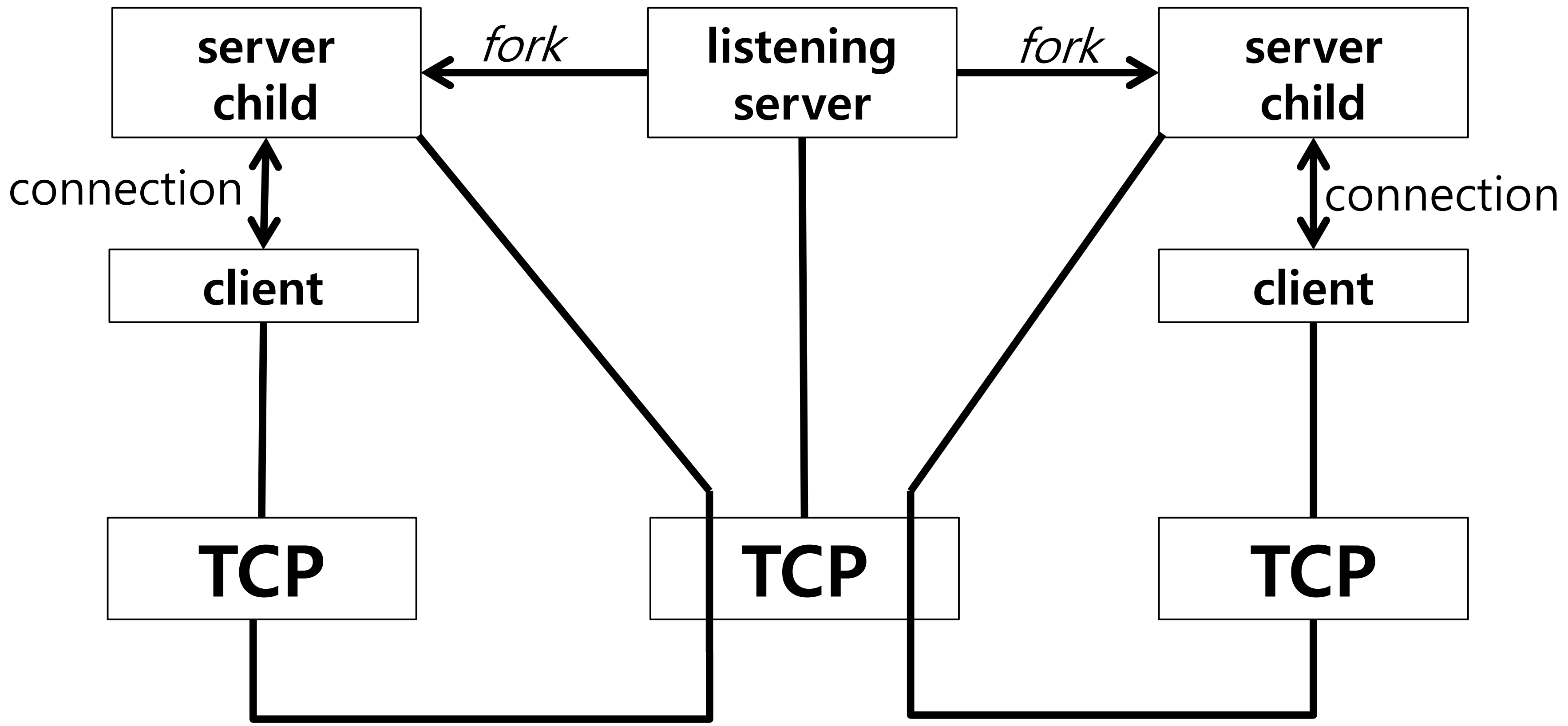
- **Queues in UDP**



- **Size of this queue can be modified by SO_RCVBUF socket option.** **(Chapter 7.)**

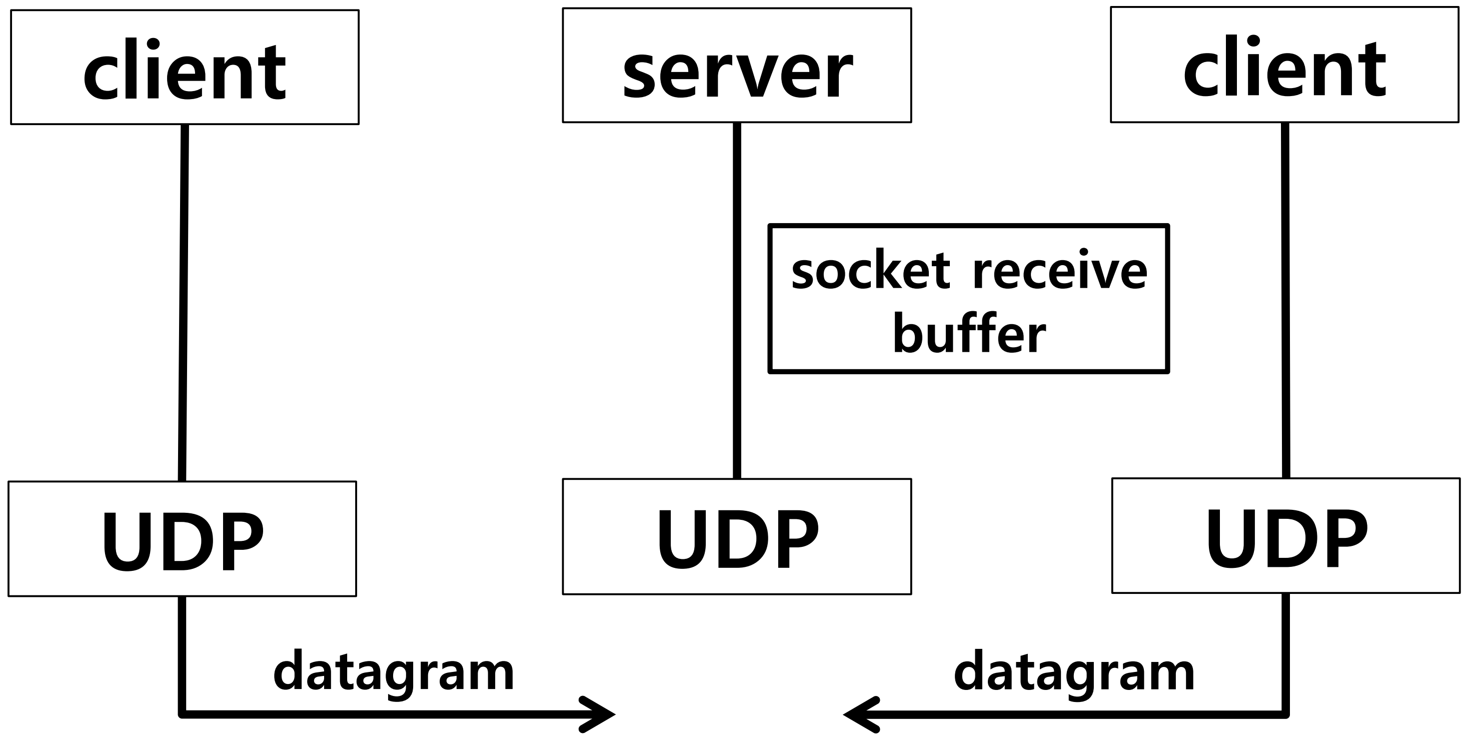
UDP Echo Server

- **TCP client/server with two clients**

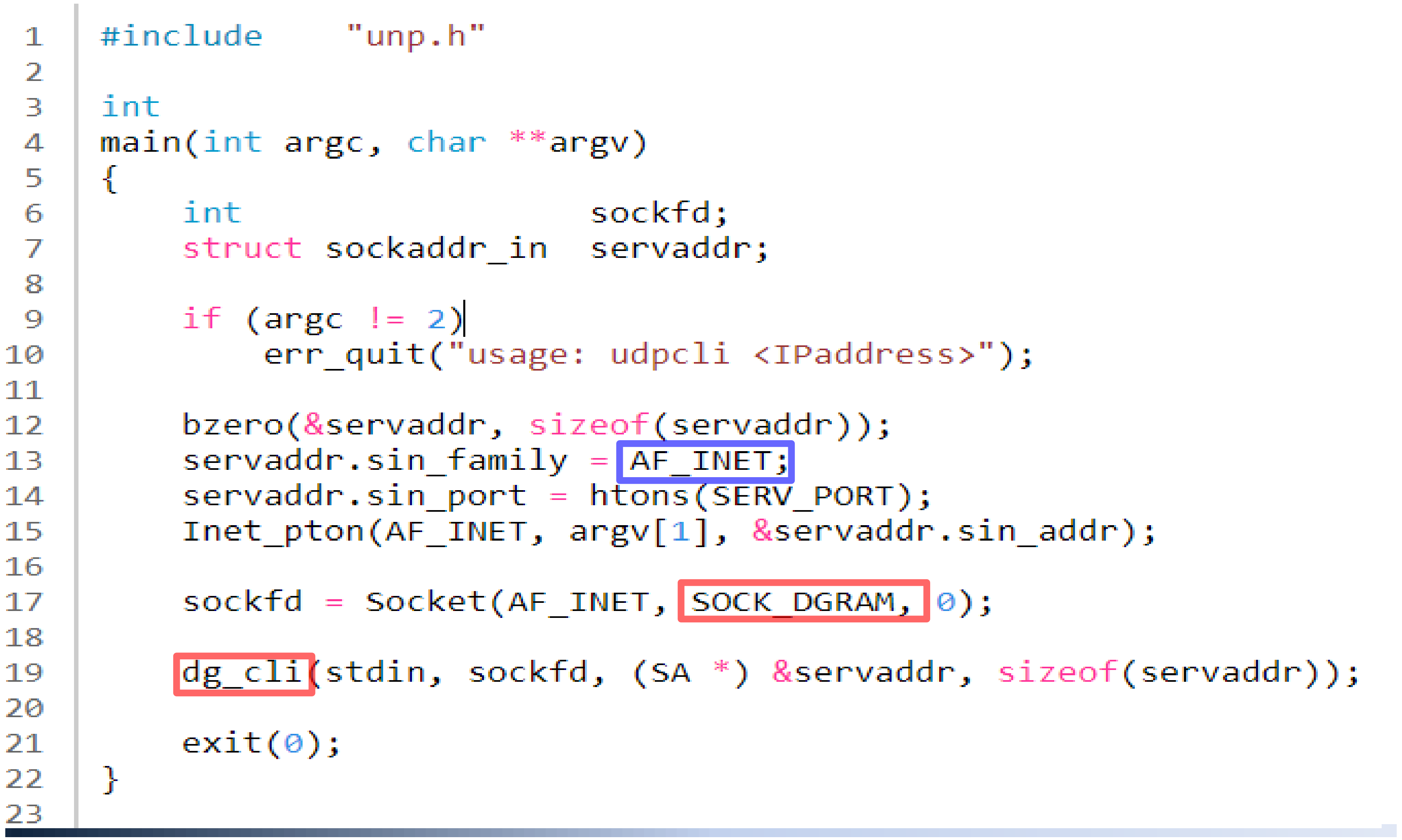


UDP Echo Server

- UDP client/server with two clients



UDP Echo Client



```c
#include    "unp.h"

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

    if (argc != 2)
        err_quit("usage: udpcli <IPaddress>");

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

    sockfd = Socket(AF_INET, SOCK_DGRAM, 0);

    dg_cli(stdin, sockfd, (SA *) &servaddr, sizeof(servaddr));

    exit(0);
}

```

UDP Echo Client

```
#include    "unp.h"

void
dg_cli(FILE *fp, int sockfd, const SA *pservaddr, socklen_t servlen)
{
    int n;
    char    sendline[MAXLINE], recvline[MAXLINE + 1];

    while (Fgets(sendline, MAXLINE, fp) != NULL) {

        Sendto(sockfd, sendline, strlen(sendline), 0, pservaddr, servlen);

        n = Recvfrom(sockfd, recvline, MAXLINE, 0, NULL, NULL);

        recvline[n] = 0;    /* null terminate */
        Fputs(recvline, stdout);
    }
}
```

Lost Datagrams

- **UDP client/server is not reliable.**
  - If a client datagram is lost, the client will block forever in its call to *recvfrom.*
  - Or, if the server's reply is lost, the client will block again anyway.

- **To prevent this problem, we can place a timeout on the call to *recvfrom.* (Chapter 14.)**
- **But, this is not a perfect solution. we can't certainly know which is missing : request or reply.**
- **Adding reliablity to UDP (Chapter 22.)**

Verifying Received Response

- **Change the client *main* function**

```
servaddr.sin_port = htons(SERV_PORT);
```

to

↓

```
servaddr.sin_port = htons(7);
```

echo protocol port

Verifying Received Response

```c
#include    "unp.h"

void
dg_cli(FILE *fp, int sockfd, const SA *pservaddr, socklen_t servlen)
{
    int             n;
    char            sendline[MAXLINE], recvline[MAXLINE + 1];
    socklen_t       len;
    struct sockaddr *preply_addr;

    preply_addr = Malloc(servlen);

    while (Fgets(sendline, MAXLINE, fp) != NULL) {

        Sendto(sockfd, sendline, strlen(sendline), 0, pservaddr, servlen);

        len = servlen;
        n = Recvfrom(sockfd, recvline, MAXLINE, 0, preply_addr, &len);
        if (len != servlen || memcmp(pservaddr, preply_addr, len) != 0) {
            printf("reply from %s (ignored)\n",
                    Sock_ntop(preply_addr, len));
            continue;
        }

        recvline[n] = 0;    /* null terminate */
        Fputs(recvline, stdout);
    }
}

```

Verifying Received Response

1. **Allocate another socket address structure**

2. **Compare returned address**
   - Coution : In section 3.2 , we never need to set or examine a length field(*sockaddr_in.sin_len*) of the socket address structure.

     But, *memcmp* compares every byte of data of two socket address structures, so we should set a length field when constructing the *sockaddr.*

     If we don't, *memcmp* compares 0(didn't set) with 16 (sockaddr_in) and will not match.

Verifying Received Response

- **This new program can fail if the server is multihomed.**

```
macosx % host freebsd4
freebsd4.unpbook.com has address 172.24.37.94
freebsd4.unpbook.com has address 135.197.17.100
macosx % udpcli02 135.197.17.100
hello
reply from 172.24.37.94:7 (ignored)
goodbye
reply from 172.24.37.94:7 (ignored)
```

- **We can manage this kind of situation by looking at host's domain name**(Chapter 11.)**, or by *select* function.** (example at Chapter 22.)

Server Not Running

1. Start *tcpdump*
2. Start the client on the same host, specify the server host as freebsd4

```
maxosx % udpcli01 172.24.37.94

hello, world
```

3. Output of *tcpdump*

address resolution protocol

```
0.0          arp who-has freebsd4 tell macosx
0.003576     arp reply freebsd4 is-at 0:40:5:42:d6:de
0.003601     macosx.51139 > freebsd4.9877: udp 13
0.009781     freebsd4 > macosx: icmp: freebsd4 udp port 9877 unreachable
```

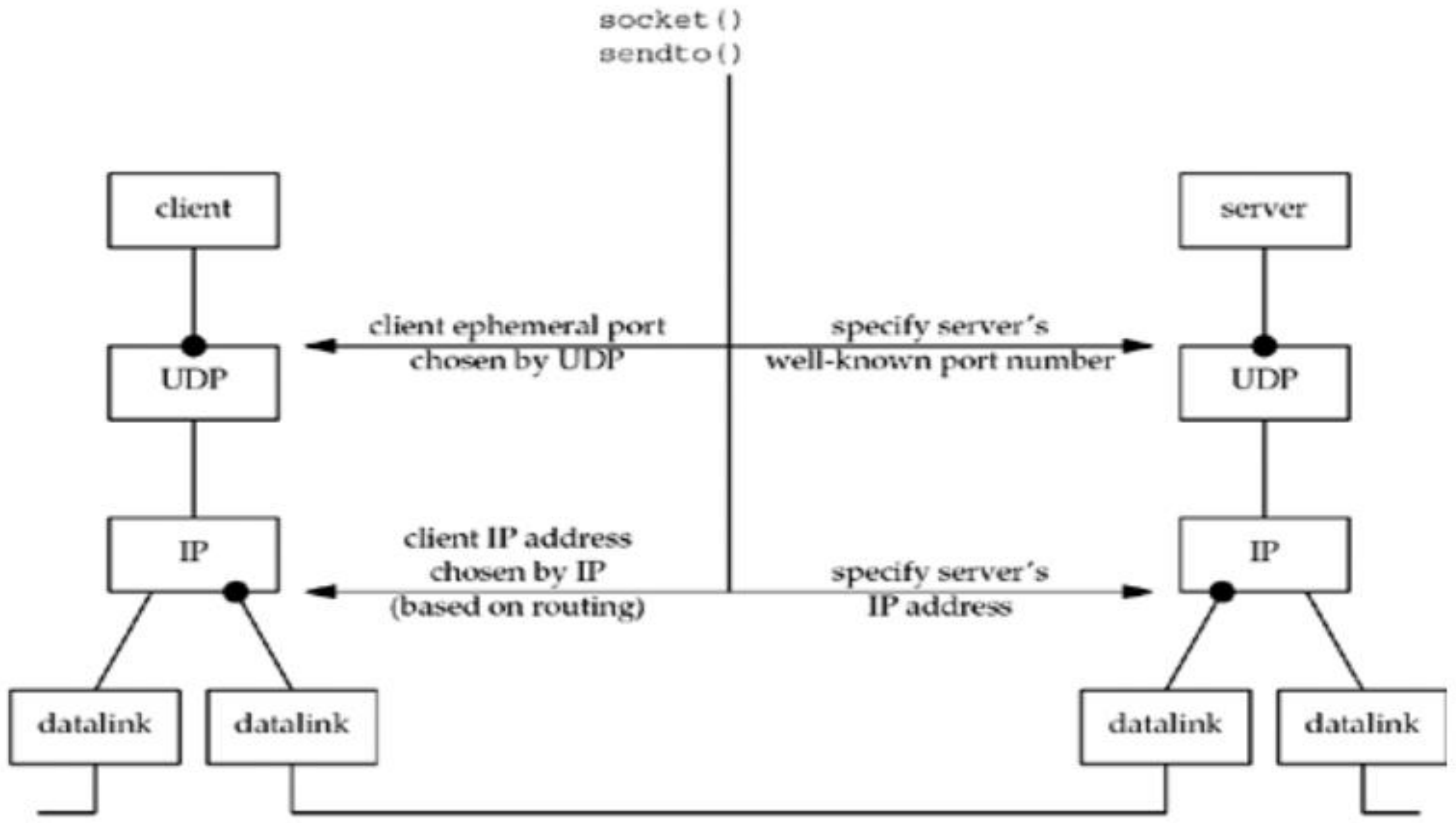
Server Not Running

- **This ICMP error is an *asynchoronous error,* and returned to the client process, not client socket.**

- **The basic rule : An asynchronous error is not returned for a UDP socket unless the socket has been connected.**(Section 8.11.)

Summary of UDP Example

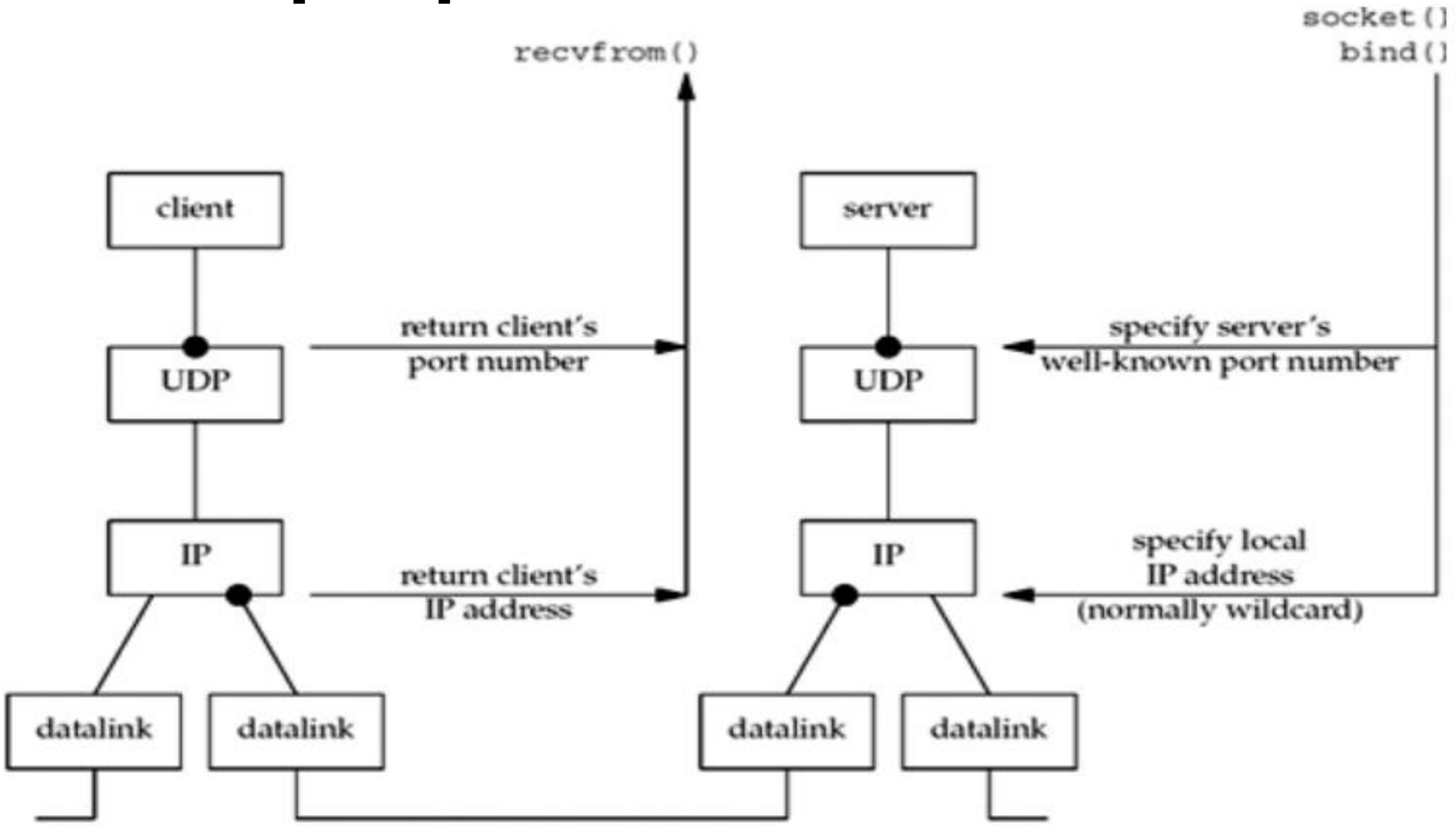
- **Client's perspective**



socket()
sendto()
client
server
client ephemeral port chosen by UDP
specify server's well-known port number
UDP
UDP
client IP address chosen by IP (based on routing)
specify server's IP address
IP
IP
datalink
datalink
datalink
datalink

Summary of UDP Example

- **Server's perspective**


recvfrom()
socket()
bind()
client
server
return client's port number
specify server's well-known port number
UDP
UDP
IP
IP
return client's IP address
specify local IP address (normally wildcard)
datalink
datalink
datalink
datalink

Summary of UDP Example

- **Available information from arriving IP datagram**

From client's IP datagram	TCP server	UDP server
Source IP address	*accept*	*recvfrom*
Source port number	*accept*	*recvfrom*
Destination IP address	*getsockname*	*recvmsg*
Destination port number	*getsockname*	*getsockname*

connect Function with UDP

- **We can call *connect* for a UDP socket.**

- **Difference between TCP *connect***
  1. No three-way handshake
  2. Kernel just checks for any immediate errors(like unreachable error)
  3. Records the IP address and port number of the peer
  4. Returns immediately to the calling process

connect Function with UDP

- **Naming can be confusing.**
  - If we say *sockname* as the local protocol address and *peername* as the foreign protocol address, maybe name as *setpeername* would be better.
  - similarly, a better name for the *bind* would be *setsockname.*

connect Function with UDP

- #define
  - ***unconnected UDP socket,*** the default when we create a UDP socket.
  - ***connected UDP socket,*** the result of calling *connect* on a UDP socket.

connect Function with UDP

- **Differences compared to *unconnected UDP***
  1. No longer specify the destination IP address and port number for an output operation. So we can use *write* or *send* instead of *sendto.*
  2. We do not use *recvfrom* to know the sender, but use *read, recv, recvmsg* instead.
  3. Asynchronous errors are retruned to the process for connected UDP sockets. Unconnected ones do not receive those errors.

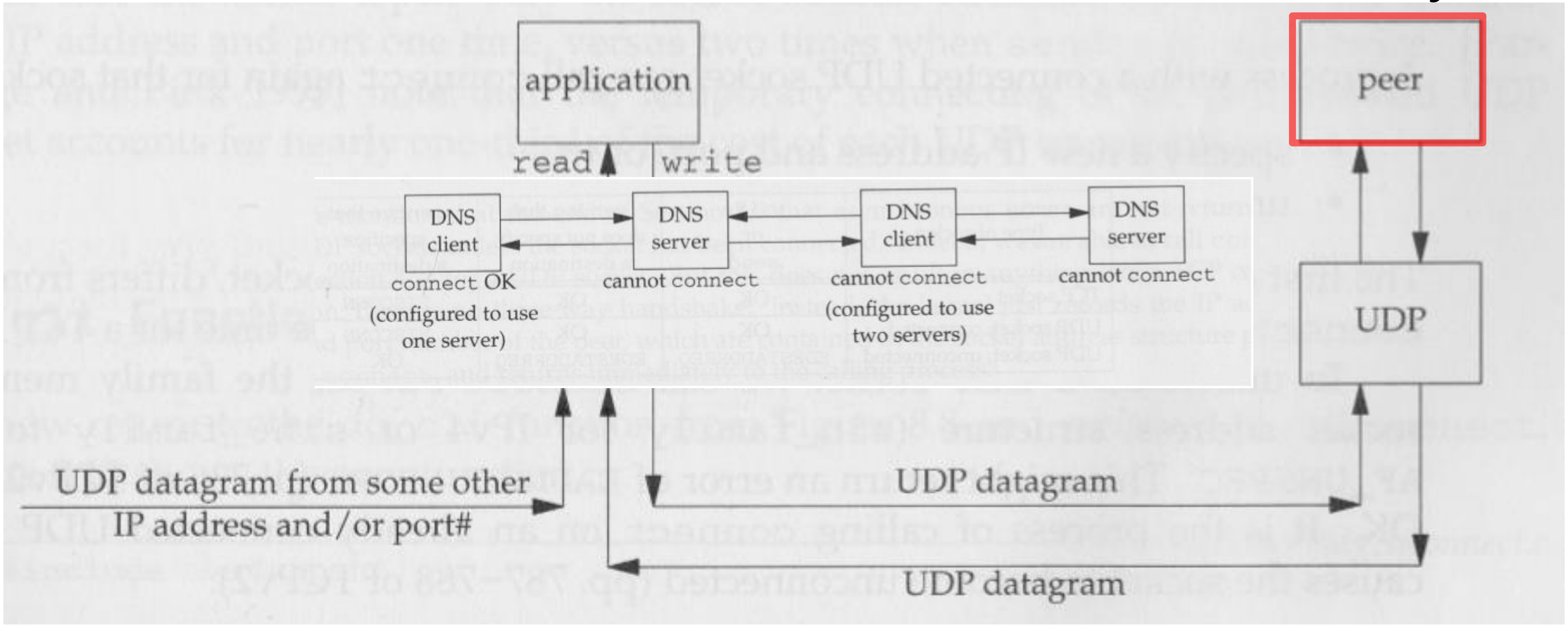
connect Function with UDP

Type of socket	*write* or *send*	*sendto* (destination not specified)	*sendto* (destination specified)
TCP socket	OK	OK	EISCONN
UDP socket, connected UDP socket, unconnected	OK EDESTADDRREQ	OK EDESTADDRREQ	EISCONN OK

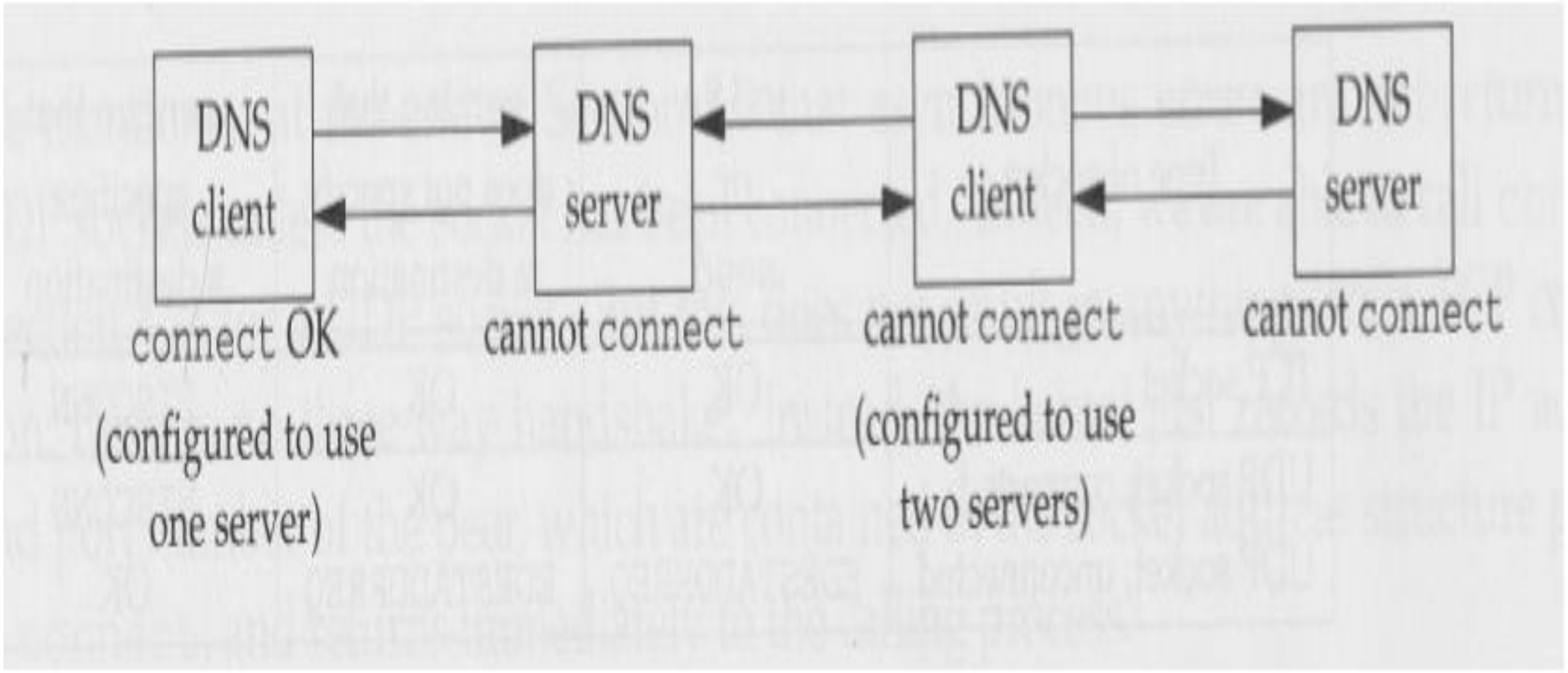
- EISCONN : the socket is already connected.
- EDESTADDRREQ : destination address required.

connect Function with UDP

One-and-Only Peer



connect Function with UDP



connect Function with UDP

- **Calling *connect* multiple times**
  - To specify a new IP address and port
    - Differs from the use with TCP : *connect* can be called only once for a TCP socket.
  - To unconnect the socket
    - Re-call *connect* by setting socket address structure's family member(sin_family) to AF_UNSPEC.
    - There are variant ways to unconnect sockets.
    - The most portable solution is to zero out an address structure, and set family memver to AF_UNSPEC.

connect Function with UDP

- **Performance**

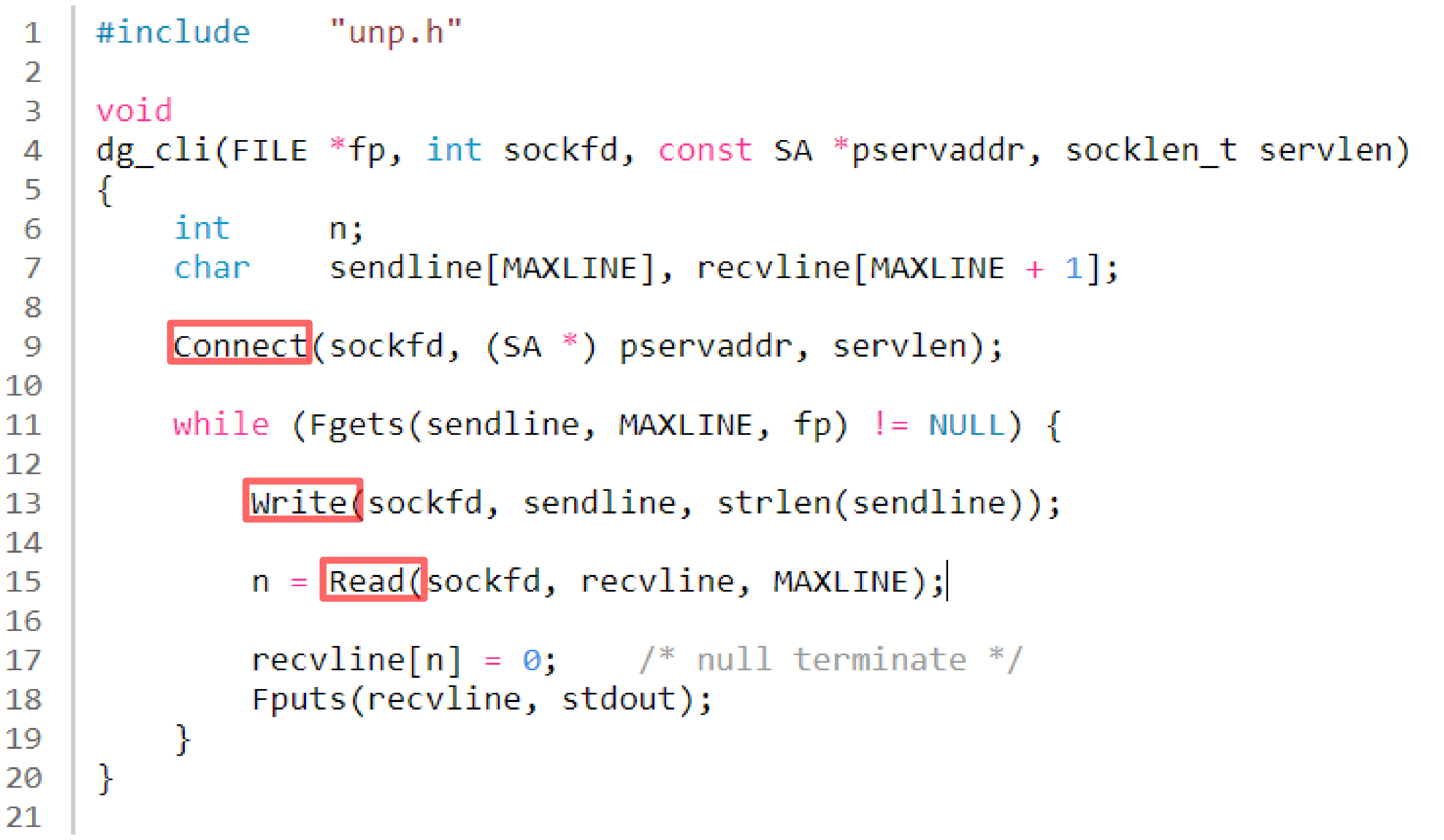
Use *sendto* on unconnected UDP

- Connect the socket
- Output the datastream 1
- Unconnect the socket
- Connect the socket
- Output the datastream 2
- Unconnect the socket

Use *write* twice on connected UDP

- Connect the socket
- Output the datagram 1
- Output the datagram 2

dg_cli Function (Revisited)



```c
#include    "unp.h"

void
dg_cli(FILE *fp, int sockfd, const SA *pservaddr, socklen_t servlen)
{
    int     n;
    char    sendline[MAXLINE], recvline[MAXLINE + 1];

    Connect(sockfd, (SA *) pservaddr, servlen);

    while (Fgets(sendline, MAXLINE, fp) != NULL) {

        Write(sockfd, sendline, strlen(sendline));

        n = Read(sockfd, recvline, MAXLINE);

        recvline[n] = 0;    /* null terminate */
        Fputs(recvline, stdout);
    }
}

```

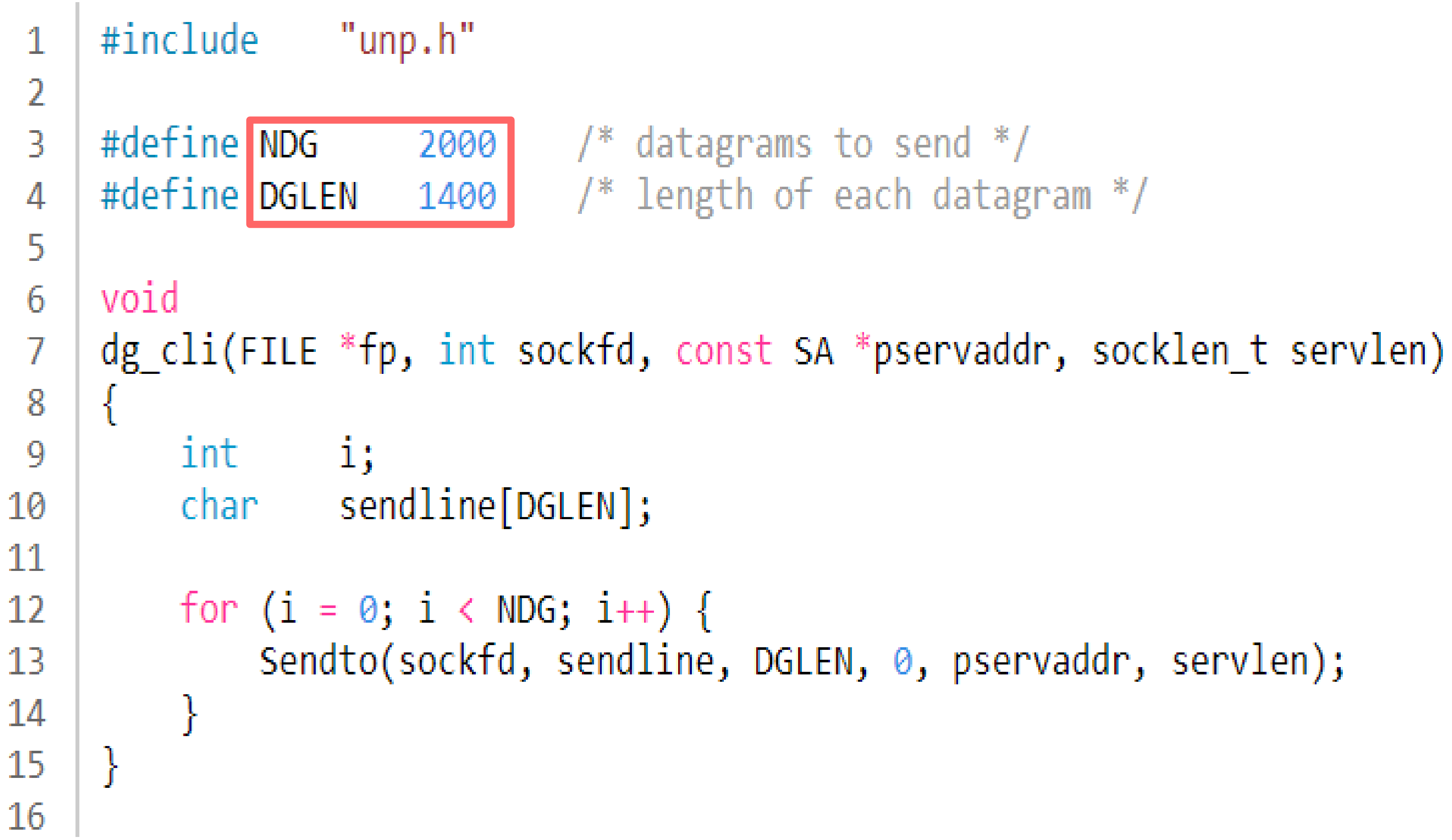
dg_cli Function (Revisited)

```
maxosx % udpcli04 172.24.37.94
hello, world
read error: Connection refused
```

- Although we started client without starting the server, error message appears after we send the first datagram to the server.

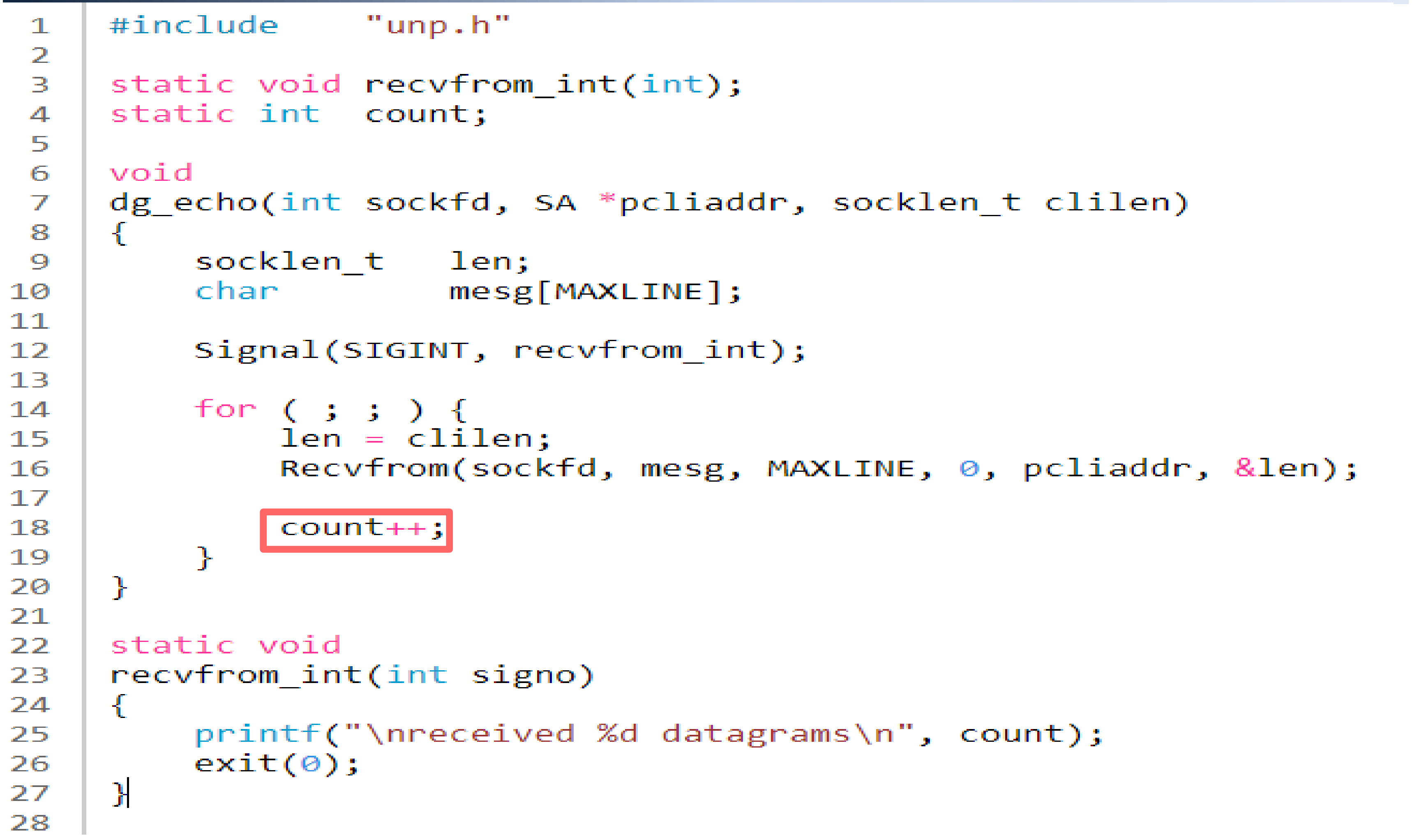
- WHY? ***there is no three-way handshake.***

Lack of Flow Control with UDP



```
#include    "unp.h"

#define NDG      2000    /* datagrams to send */
#define DGLEN    1400    /* length of each datagram */

void
dg_cli(FILE *fp, int sockfd, const SA *pservaddr, socklen_t servlen)
{
    int     i;
    char    sendline[DGLEN];

    for (i = 0; i < NDG; i++) {
        Sendto(sockfd, sendline, DGLEN, 0, pservaddr, servlen);
    }
}

```

Lack of Flow Control with UDP



```c
#include	"unp.h"

static void	recvfrom_int(int);
static int	count;

void
dg_echo(int sockfd, SA *pcliaddr, socklen_t clilen)
{
	socklen_t	len;
	char		mesg[MAXLINE];

	Signal(SIGINT, recvfrom_int);

	for ( ; ; ) {
		len = clilen;
		Recvfrom(sockfd, mesg, MAXLINE, 0, pcliaddr, &len);

		count++;
	}
}

static void
recvfrom_int(int signo)
{
	printf("\nreceived %d datagrams\n", count);
	exit(0);
}
```

Lack of Flow Control with UDP

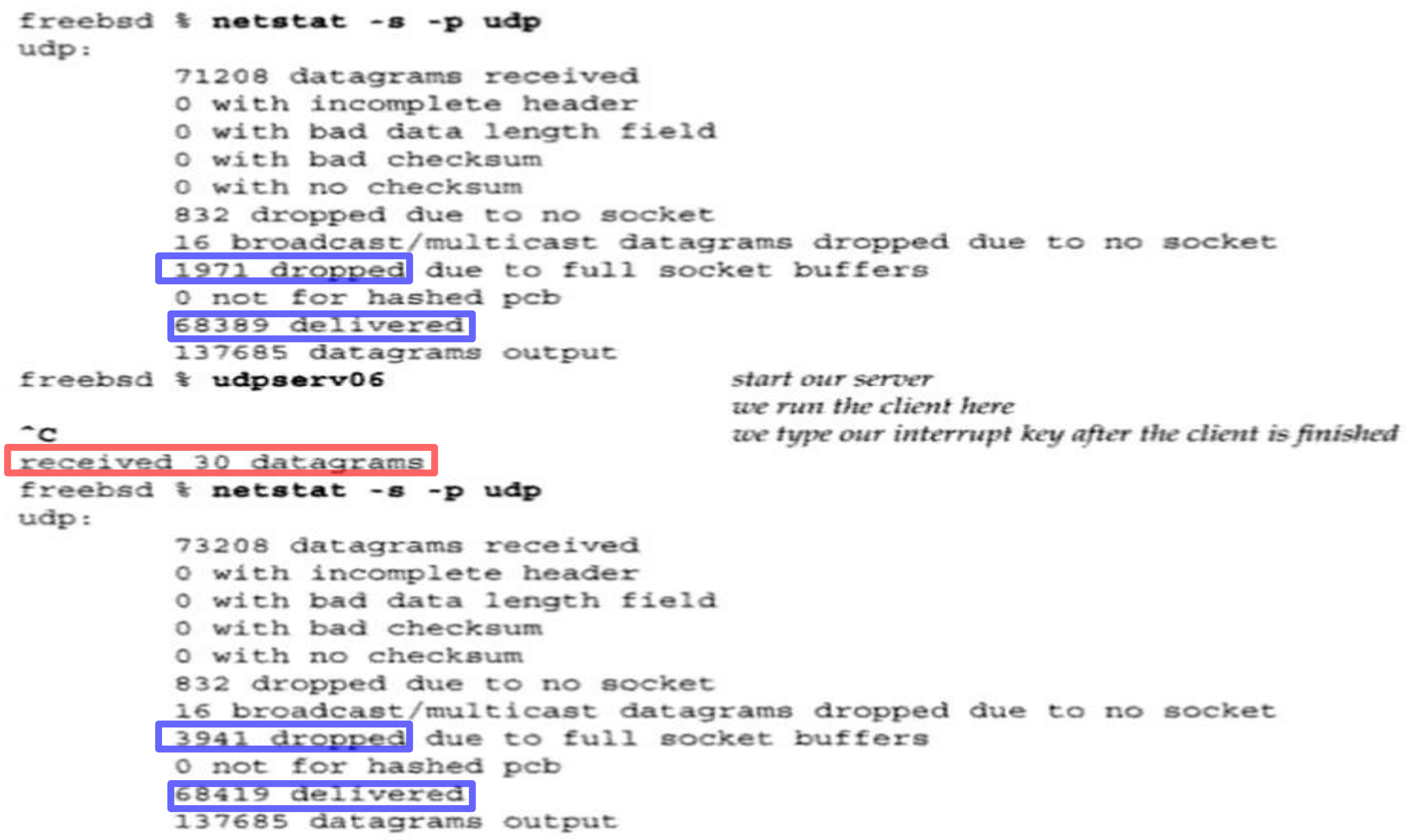
Server

SPARKStation

Client

RS/6000

Lack of Flow Control with UDP



```
freebsd % netstat -s -p udp
udp:
        71208 datagrams received
        0 with incomplete header
        0 with bad data length field
        0 with bad checksum
        0 with no checksum
        832 dropped due to no socket
        16 broadcast/multicast datagrams dropped due to no socket
        1971 dropped due to full socket buffers
        0 not for hashed pcb
        68389 delivered
        137685 datagrams output
freebsd % udpserv06                          start our server
                                             we run the client here
^C                                           we type our interrupt key after the client is finished
received 30 datagrams
freebsd % netstat -s -p udp
udp:
        73208 datagrams received
        0 with incomplete header
        0 with bad data length field
        0 with bad checksum
        0 with no checksum
        832 dropped due to no socket
        16 broadcast/multicast datagrams dropped due to no socket
        3941 dropped due to full socket buffers
        0 not for hashed pcb
        68419 delivered
        137685 datagrams output
```

Lack of Flow Control with UDP

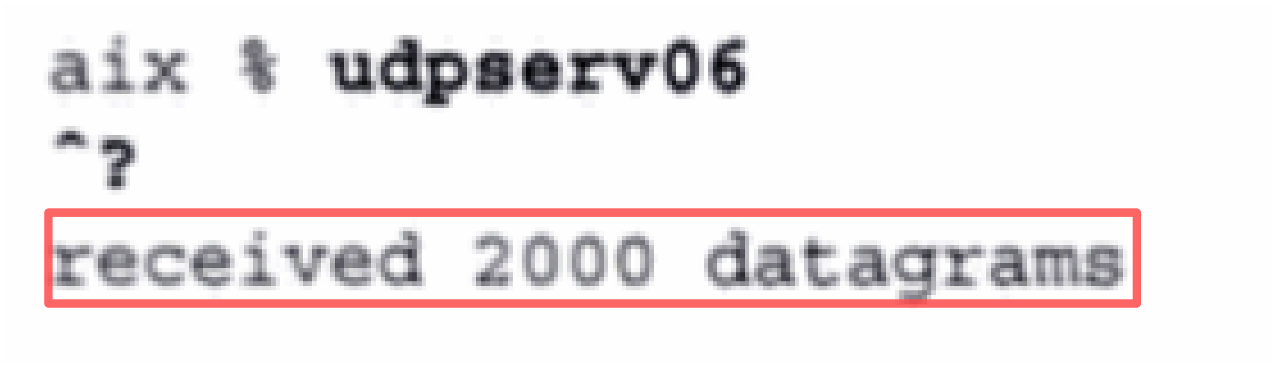
Server

Client

SPARKStation

RS/6000

Lack of Flow Control with UDP

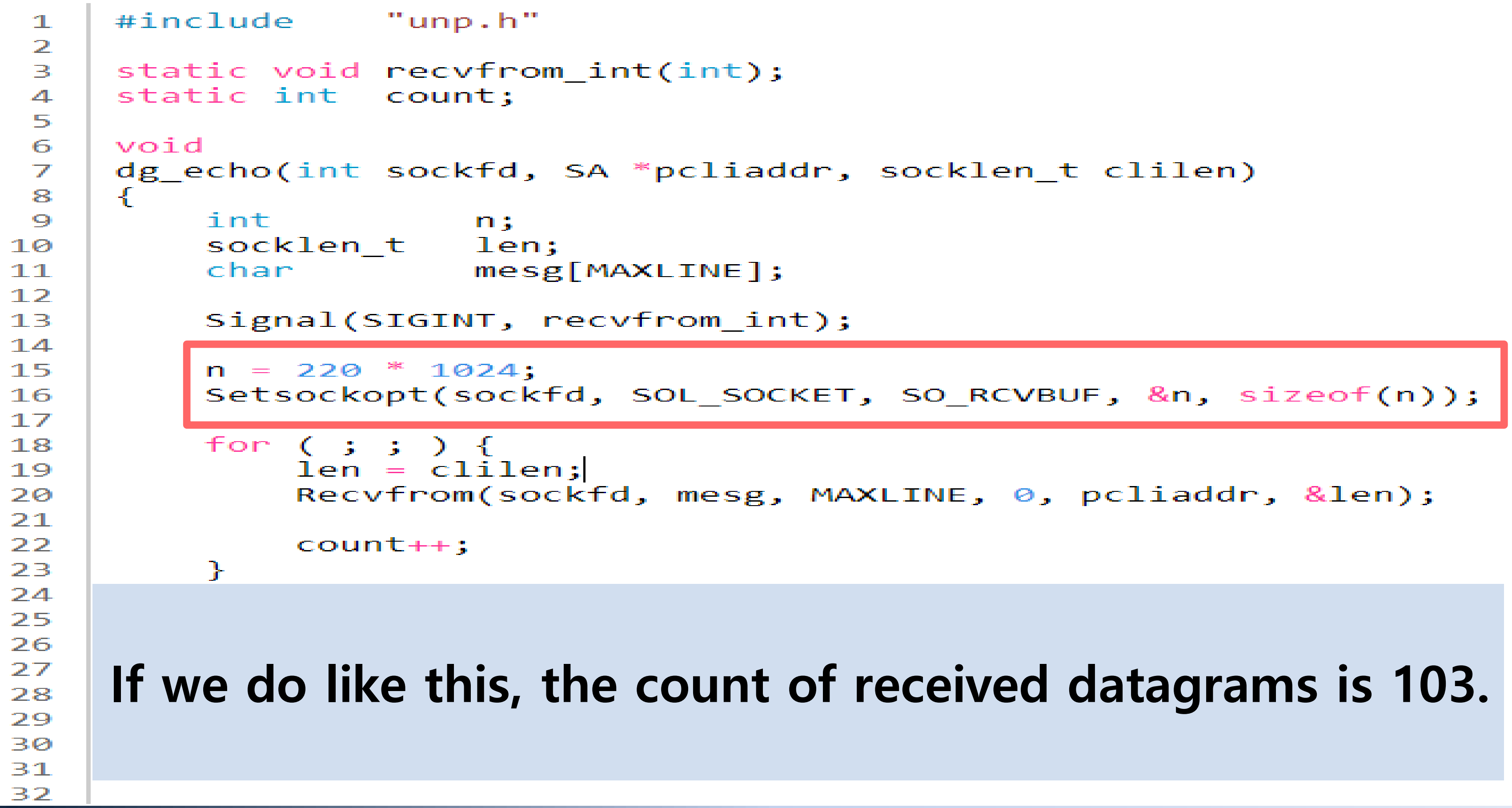


```
aix % udpserv06
^?
received 2000 datagrams
```

- All datagrams received successfully!

Lack of Flow Control with UDP

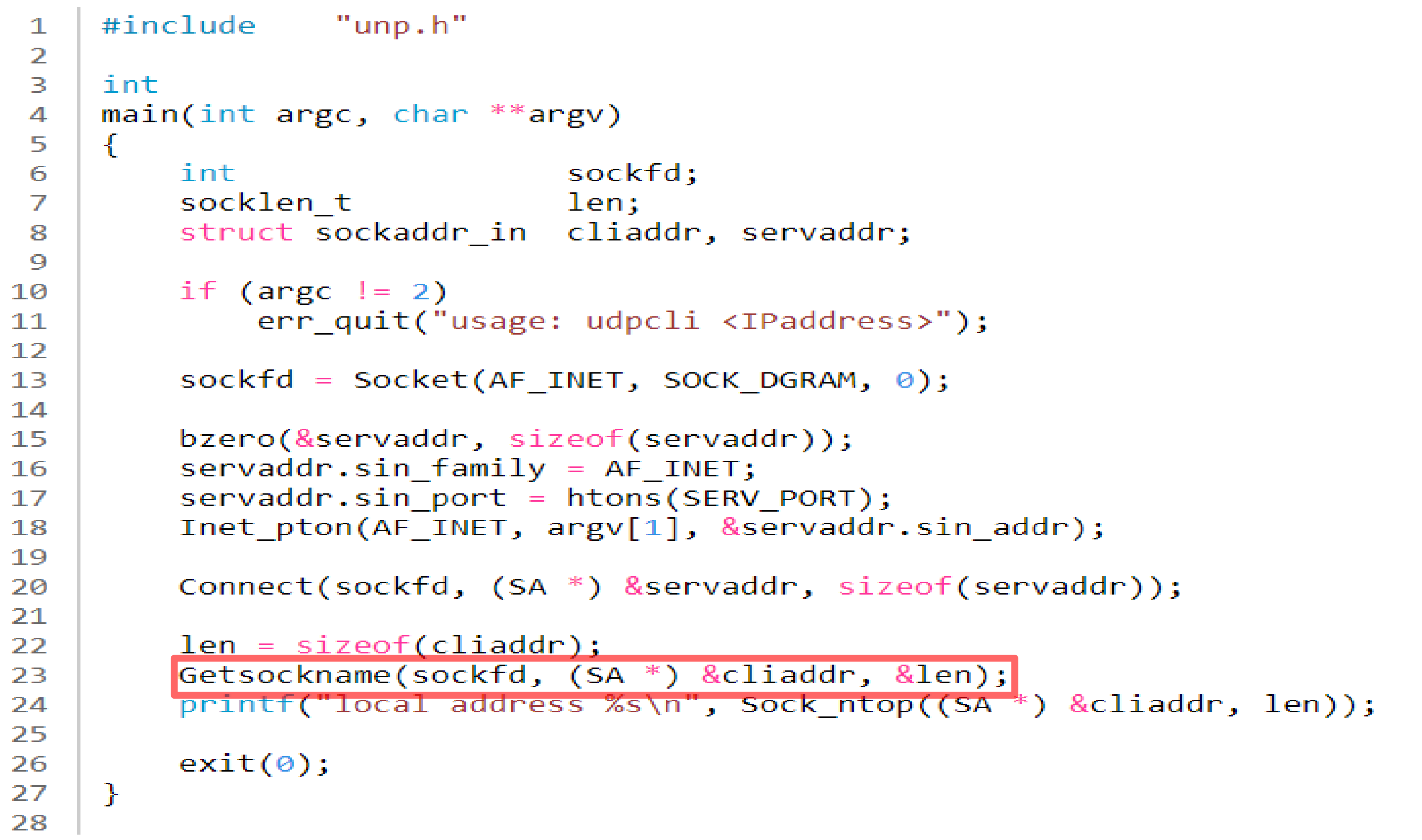
- **UDP socket receive buffer**



```
#include     "unp.h"

static void recvfrom_int(int);
static int   count;

void
dg_echo(int sockfd, SA *pcliaddr, socklen_t clilen)
{
        int           n;
        socklen_t     len;
        char          mesg[MAXLINE];

        Signal(SIGINT, recvfrom_int);

        n = 220 * 1024;
        Setsockopt(sockfd, SOL_SOCKET, SO_RCVBUF, &n, sizeof(n));

        for ( ; ; ) {
                len = clilen;
                Recvfrom(sockfd, mesg, MAXLINE, 0, pcliaddr, &len);

                count++;
        }
```

If we do like this, the count of received datagrams is 103.

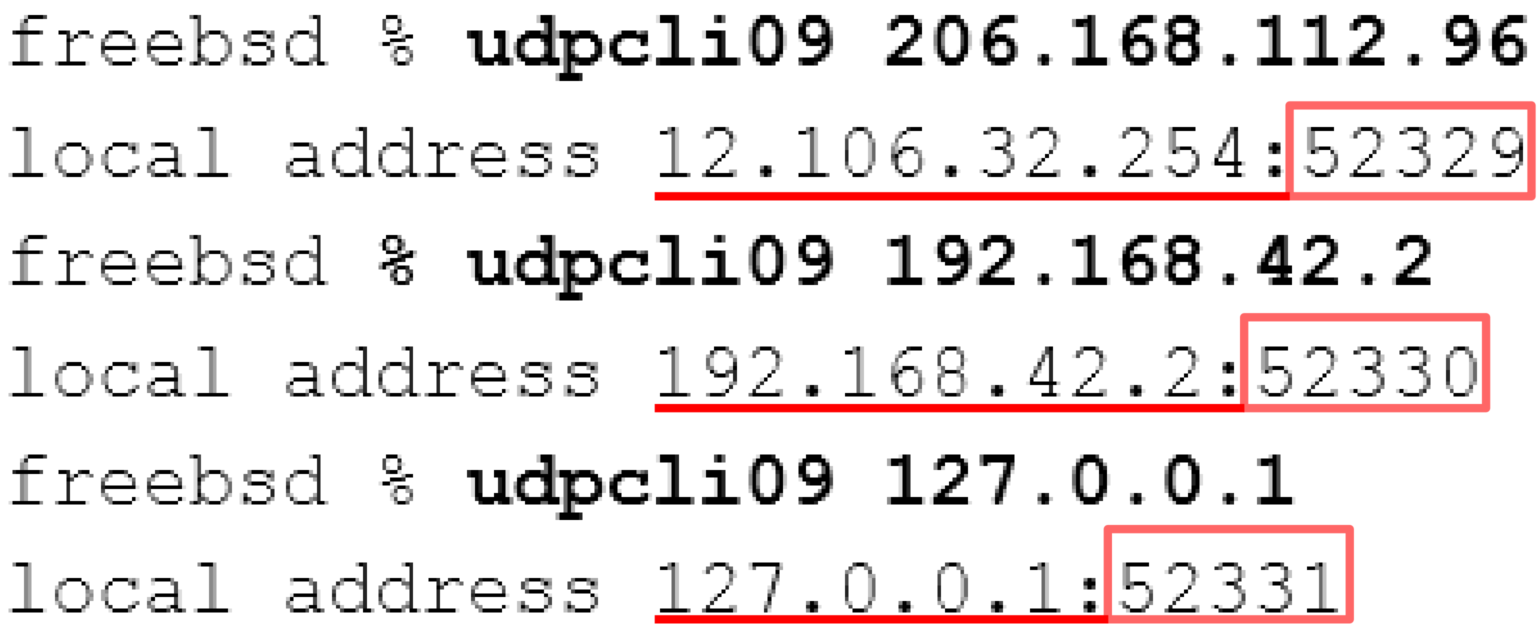
Determining Outgoing Interface



```c
#include	"unp.h"

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

	if (argc != 2)
		err_quit("usage: udpcli <IPaddress>");

	sockfd = Socket(AF_INET, SOCK_DGRAM, 0);

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

	Connect(sockfd, (SA *) &servaddr, sizeof(servaddr));

	len = sizeof(cliaddr);
	Getsockname(sockfd, (SA *) &cliaddr, &len);
	printf("local address %s\n", Sock_ntop((SA *) &cliaddr, len));

	exit(0);
}
```

Determining Outgoing Interface

- **On multihomed host**



```
freebsd % udpcli09 206.168.112.96
local address 12.106.32.254:52329
freebsd % udpcli09 192.168.42.2
local address 192.168.42.2:52330
freebsd % udpcli09 127.0.0.1
local address 127.0.0.1:52331
```

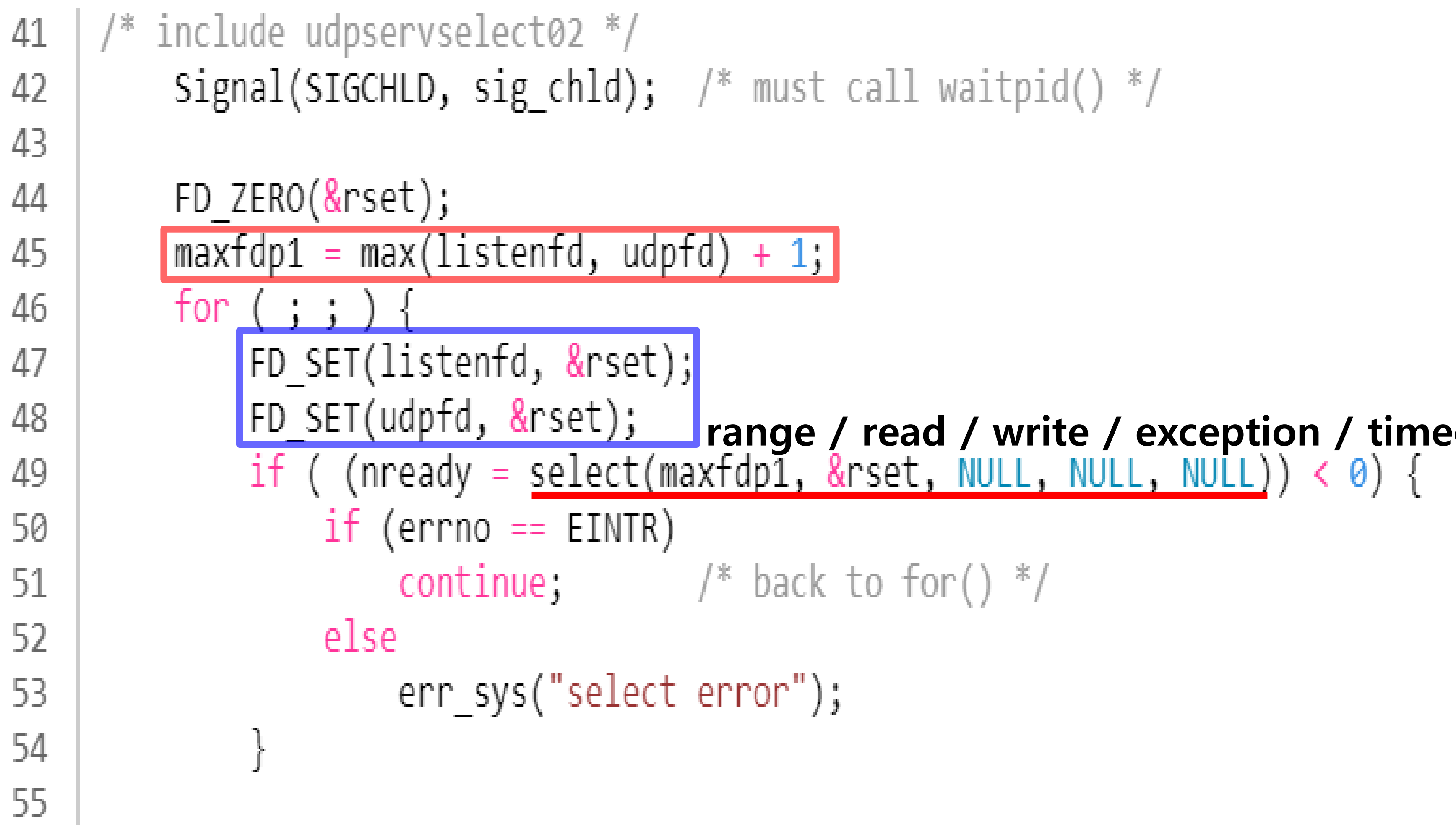
TCP and UDP Echo Server Using *select*

```c
/* include udpservselect01 */
#include    "unp.h"

int
main(int argc, char **argv)
{
    int                 listenfd, connfd, udpfd, nready, maxfdp1;
    char                mesg[MAXLINE];
    pid_t               childpid;
    fd_set              rset;
    ssize_t             n;
    socklen_t           len;
    const int           on = 1;
    struct sockaddr_in  cliaddr, servaddr;
    void                sig_chld(int);

```

TCP and UDP Echo Server Using *select*

```
        /* 4create listening TCP socket */
    listenfd = Socket(AF_INET, SOCK_STREAM, 0);

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

    Setsockopt(listenfd, SOL_SOCKET, SO_REUSEADDR, &on, sizeof(on));
    Bind(listenfd, (SA *) &servaddr, sizeof(servaddr));

    Listen(listenfd, LISTENQ);

        /* 4create UDP socket */
    udpfd = Socket(AF_INET, SOCK_DGRAM, 0);

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

    Bind(udpfd, (SA *) &servaddr, sizeof(servaddr));
/* end udpservselect01 */
```

TCP and UDP Echo Server Using *select*



```
/* include udpservselect02 */
    Signal(SIGCHLD, sig_chld);  /* must call waitpid() */

    FD_ZERO(&rset);
    maxfdp1 = max(listenfd, udpfd) + 1;
    for ( ; ; ) {
        FD_SET(listenfd, &rset);
        FD_SET(udpfd, &rset);
        if ( (nready = select(maxfdp1, &rset, NULL, NULL, NULL)) < 0) {
            if (errno == EINTR)
                continue;       /* back to for() */
            else
                err_sys("select error");
        }

```

range / read / write / exception / timeout

TCP and UDP Echo Server Using *select*

```
        if (FD_ISSET(listenfd, &rset)) {
            len = sizeof(cliaddr);
            connfd = Accept(listenfd, (SA *) &cliaddr, &len);

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

        if (FD_ISSET(udpfd, &rset)) {
            len = sizeof(cliaddr);
            n = Recvfrom(udpfd, mesg, MAXLINE, 0, (SA *) &cliaddr, &len);

            Sendto(udpfd, mesg, n, 0, (SA *) &cliaddr, len);
        }
    }
}
```

KWANGWOON UNIVERSITY

Thank You!