# UDP Sockets

Chap 8, 14, 22

# Elementary UDP Sockets

Chap 8

# TCP versus UDP

- TCP
  - connection-oriented
  - reliable
  - byte stream
- Application: typically concurrent server
  - SMTP(Simple Mail Transfer Protocol)
  - Telnet
  - FTP
  - HTTP
  - NNTP(Network News TP)

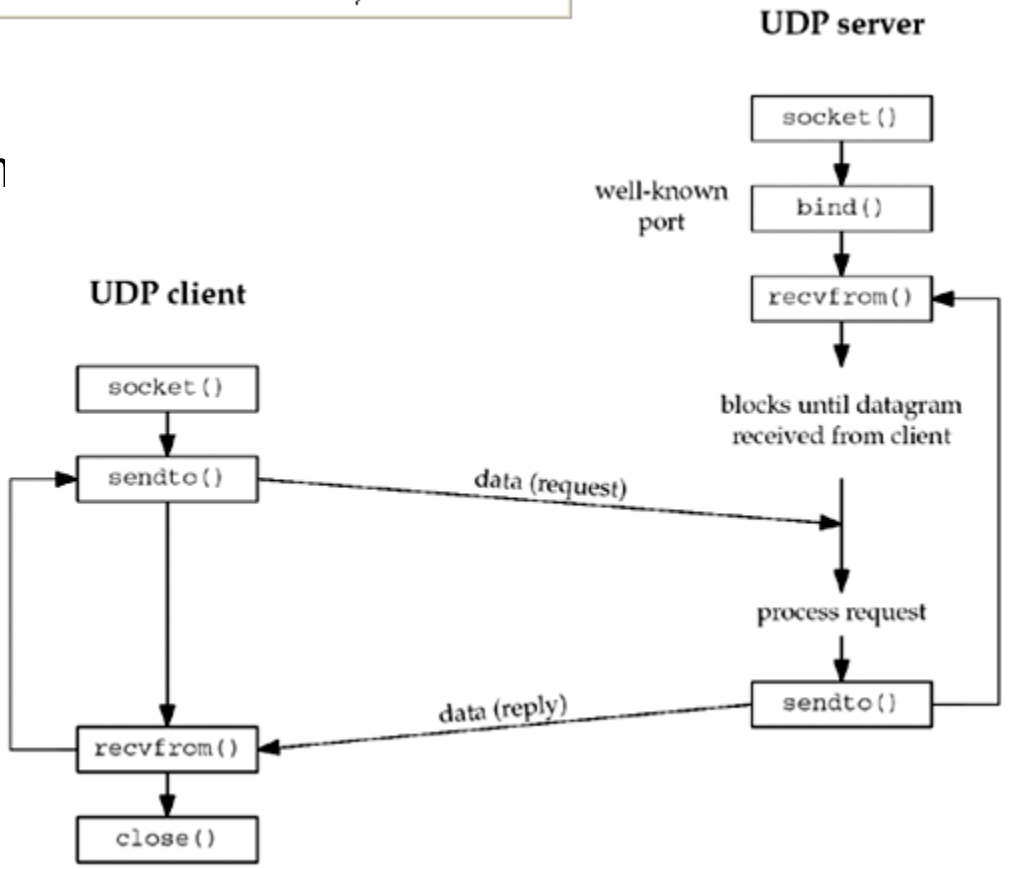
- UDP
  - connectionless
  - unreliable
  - datagram
- Applications: typically iterative server
  - SNMP(Simple Network Management Protocol)
  - TFTP(Trivial FTP)
  - BOOTP(Bootstrap Protocol)
  - DHCP(Bootstrap Protocol)

# Socket Functions for UDP Client/Server

```
#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, const void *buff, size_t nbytes, int flags,
const struct sockaddr *to, socklen_t addrlen);
```

Both return: number of bytes read or written if OK, −1 on error

- recvfrom
  - Return value 0 : datagram length 0
  - If no interest in senders address
    - from : NULL, addrlen : NULL



# UDP Echo Server : `main` Function

```
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));
}
```

```
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);
    }
}
```

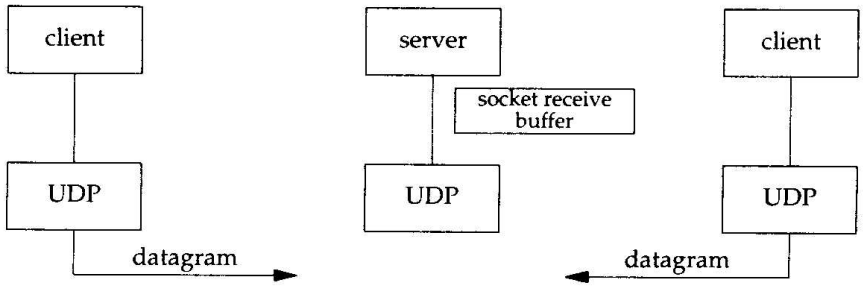



**Figure 8.6** Summary of UDP client–server with two clients.

- Never terminate
  - TCP에서 처럼 EOF를 알 수 없음
- Lost datagram: UDP has no flow control
  - Socket receive buffer가 full이면 datagram은 discard됨
- Protocol-dependent

# UDP Echo Client: main Function

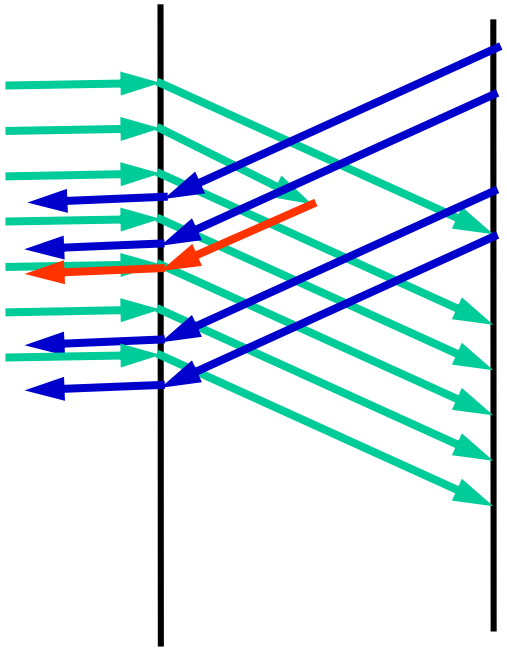
```c
int
main(int argc, char **argv)
{
    int                     sockfd;
    struct sockaddr_in servaddr;

    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);
} ? end main ?
```

```c
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 datagram ➔ recvfrom에서 blocked
- 수신된 response가 server가 보낸 것이지 verification 필요.
  - Response의 IP address와 server address를 비교: IP address를 2개이상 가질 수 있음
  - IP address를 domain name으로 conversion하여 domain name으로 비교
  - Server는 각각의 client에 대하여 socket을 create
- Server가 running되고 있지 않으면 recvfrom에서 blocked

# If Server is not Running ??

- Asynchronous error 발생
  - 보낸 datagram이 전달되지 못해서, 만일 ICMP error message(port unreachable)가 reporting되었을 경우 client process에게 error return해 줄 방법이 없음
  - recvfrom으로 기다리고 있는 client에게 kernel이 error를 발생시킨 datagram의 destination IP address와 destination port #를 알려 줄 방법 없음
  - Solution: use a connected UDP socket
    - 단, UDP socket이 1개의 peer와 connect하면 asynchronous error가 return 가능



- Sendto()가 successful return했다고 해서 datagram이 destination에 전달된 것은 아님.
  - 다만, interface output queue에 성공적으로 write했음을 의미함

# UDP Client_Server from Client's Perspective

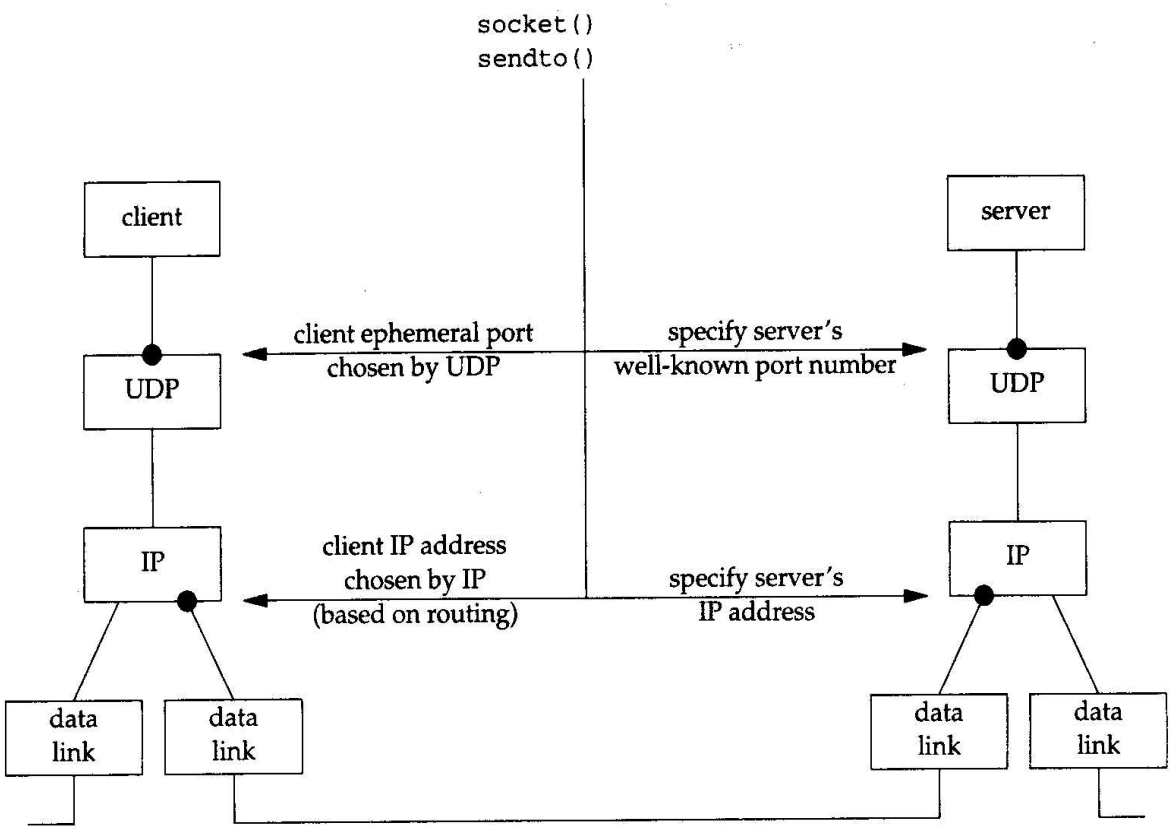



**Figure 8.11** Summary of UDP client–server from client's perspective.

- Client ephemeral port is chosen once (on the first `sendto`), and then never changes.
- Client IP address may change for every datagram, if client does not `bind` a specific address (in case of multihomed host)

# UDP Client-Server from Server's Perspectives

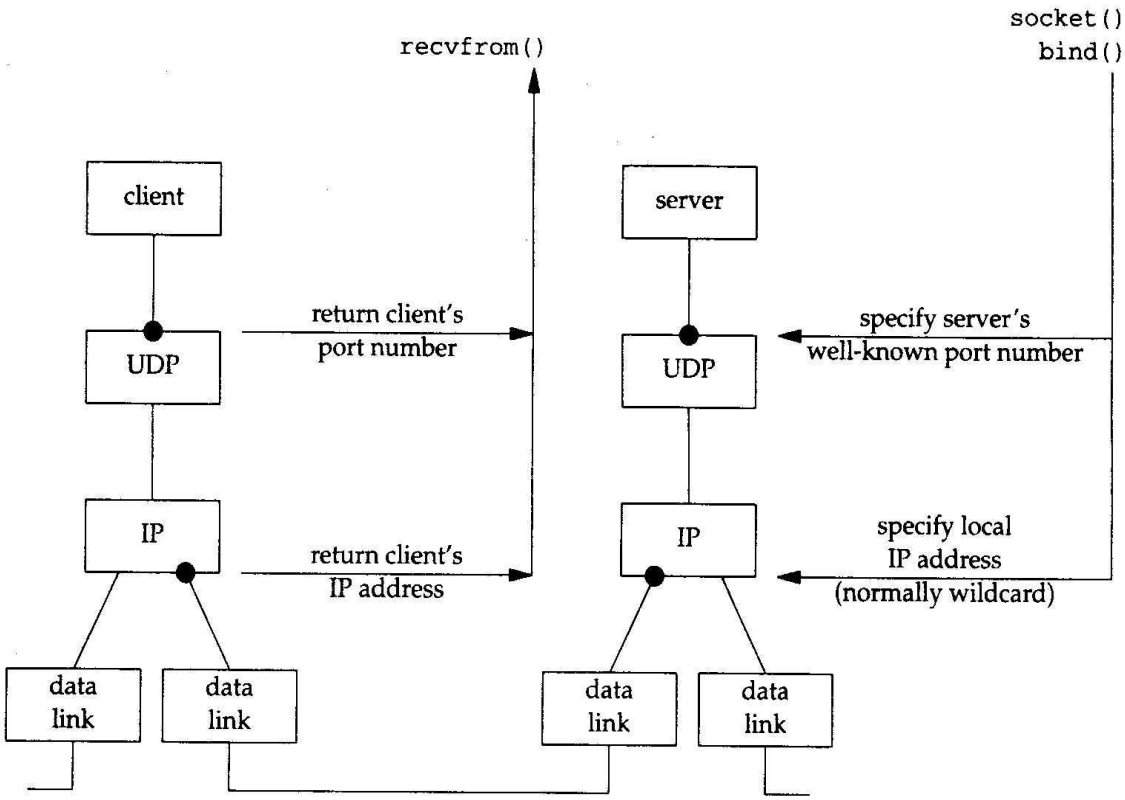



**Figure 8.12** Summary of UDP client–server from server's perspective.

| 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` |

**Figure 8.13** Information available to server from arriving IP datagram.

# Connected UDP Socket

- Call connect only to communication with exactly one peer
  - Kernel just records IP address and port # of the peer
- Connected UDP socket
  - No need to specify the destination IP addr and port # for output operation
    - write, send instead of sendto
  - No need to verify received response
    - read, recv instead of recvfrom
  - Asynchronous errors are returned
- Connected UDP socket provides better performance
  - Unconnected UDP socket: make a temporary connection(1/3 overhead)
- May connect multiple times for a UDP socket by specifying a new IP addr and port #

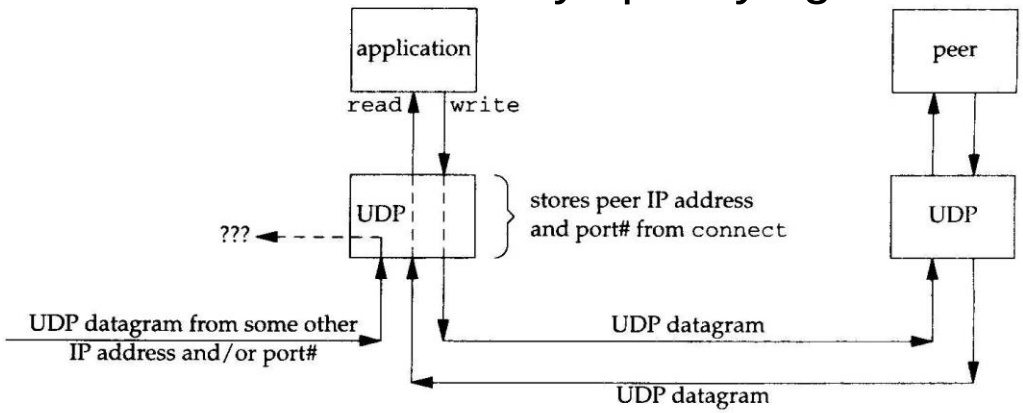



**Figure 8.15** Connected UDP socket.

# UDP Echo Client: Connected UDP socket

```c
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);
} ? end main ?
```

```c
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);
    }
}
```

- Lost datagram due to
  - lost in network
  - socket receive buffer overflow
    - UDP has no flow control
- Connected UDP socket can also be used to determine the outgoing interface to the particular destination

# TCP and UDP Echo Server using select

```
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);

        /* 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 */

/* 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");
        }

        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);
        }
    } ? end for ;; ?
} ? end main ?
/* end udpservselect02 */
```

# Advanced I/O Functions

Chap 14

# How to Place Timeouts on Sockets (1)

- Using `SIGALRM` signal

## Connection timeout 기간의 축소

lib/connect_timeo.c

```
#include "unp.h"

static void      connect_alarm(int);

int
connect_timeo(int sockfd, const SA *saptr, socklen_t salen, int nsec)
{
     Sigfunc  *sigfunc;
     int       n;

     sigfunc = Signal(SIGALRM, connect_alarm);
     if (alarm(nsec) != 0)
          err_msg("connect_timeo: alarm was already set");

     if ( (n = connect(sockfd, saptr, salen)) < 0) {
          close(sockfd);
          if (errno == EINTR)
               errno = ETIMEDOUT;
     }
     alarm(0);                       /* turn off the alarm */
     Signal(SIGALRM, sigfunc);   /* restore previous signal handler */

     return(n);
} ? end connect_timeo ?

static void
connect_alarm(int signo)
{
     return;       /* just interrupt the connect() */
}
```

## Response timeout

advio/dgclitimeo3.c

```
#include "unp.h"

static void      sig_alrm(int);

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

     Signal(SIGALRM, sig_alrm);

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

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

          alarm(5);
          if ( (n = recvfrom(sockfd, recvline, MAXLINE, 0, NULL, NULL)) < 0) {
               if (errno == EINTR)
                    fprintf(stderr, "socket timeout\n");
               else
                    err_sys("recvfrom error");
          } else {
               alarm(0);
               recvline[n] = 0;     /* null terminate */
               Fputs(recvline, stdout);
          }
     }
} ? end dg_cli ?

static void
sig_alrm(int signo)
{
     return;             /* just interrupt the recvfrom() */
}
```

alarm()은 초 단위
setitimer()는 micro sec 단위 설정 가능(실제는 msec 단위로 동작)

# How to Place Timeouts on Sockets (2)

- Using `select` with timeout

lib/readable_timeo.c

```c
#include "unp.h"

int
readable_timeo(int fd, int sec)
{
    fd_set          rset;
    struct timeval  tv;

    FD_ZERO(&rset);
    FD_SET(fd, &rset);

    tv.tv_sec = sec;
    tv.tv_usec = 0;

    return(select(fd+1, &rset, NULL, NULL, &tv));
        /* 4> 0 if descriptor is readable */
}
```

advio/dgclitimeo1.c

```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];

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

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

        if (Readable_timeo(sockfd, 5) == 0) {
            fprintf(stderr, "socket timeout\n");
        } else {
            n = Recvfrom(sockfd, recvline, MAXLINE, 0, NULL, NULL);
            recvline[n] = 0;    /* null terminate */
            Fputs(recvline, stdout);
        }
    }
}
```

# How to Place Timeouts on Sockets (3)

- Using `SO_RCVTIMEO` and `SO_SNDTIMEO` socket options
  - Caution: timeout applies to all I/O operations for the socket descriptor

advio/dgclitimeo2.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];
    struct timeval      tv;

    tv.tv_sec = 5;
    tv.tv_usec = 0;
    Setsockopt(sockfd, SOL_SOCKET, SO_RCVTIMEO, &tv, sizeof(tv));

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

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

        n = recvfrom(sockfd, recvline, MAXLINE, 0, NULL, NULL);
        if (n < 0) {
            if (errno == EWOULDBLOCK) {
                fprintf(stderr, "socket timeout\n");
                continue;
            } else
                err_sys("recvfrom error");
        }

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

# More on Socket I/O Functions

- recv and send (only for sockets)

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

ssize_t recv(int sockfd, void *buff, size_t nbytes, int flags);

ssize_t send(int sockfd, const void *buff, size_t nbytes, int flags);

                Both return: number of bytes read or written if OK, -1 on error
```

| *flags* | Description | recv | send |
|---|---|---|---|
| MSG_DONTROUTE | Bypass routing table lookup | | • |
| MSG_DONTWAIT | Only this operation is nonblocking | • | • |
| MSG_OOB | Send or receive out-of-band data | • | • |
| MSG_PEEK | Peek at incoming message | • | |
| MSG_WAITALL | Wait for all the data | • | |

- Scatter read and gather write

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

ssize_t readv(int filedes, const struct iovec *iov, int iovcnt);

ssize_t writev(int filedes, const struct iovec *iov, int iovcnt);

                Both return: number of bytes read or written, -1 on error
```

```
struct iovec {
  void   *iov_base;    /* starting address of buffer */
  size_t  iov_len;     /* size of buffer */
};
```

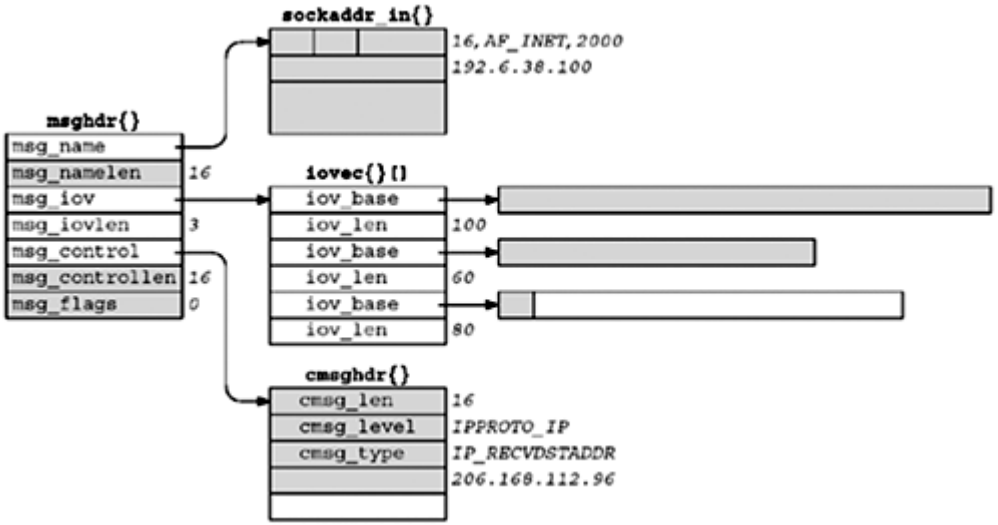
# More Advanced Socket I/O Functions

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

ssize_t recvmsg(int sockfd, struct msghdr *msg, int flags);

ssize_t sendmsg(int sockfd, struct msghdr *msg, int flags);
```

Both return: number of bytes read or written if OK, −1 on error

```
struct msghdr {
  void          *msg_name;        /* protocol address */
  socklen_t      msg_namelen;     /* size of protocol address */
  struct iovec *msg_iov;          /* scatter/gather array */
  int            msg_iovlen;      /* # elements in msg_iov */
  void          *msg_control;     /* ancillary data (cmsghdr struct) */
  socklen_t      msg_controllen;  /* length of ancillary data */
  int            msg_flags;       /* flags returned by recvmsg() */
};
```

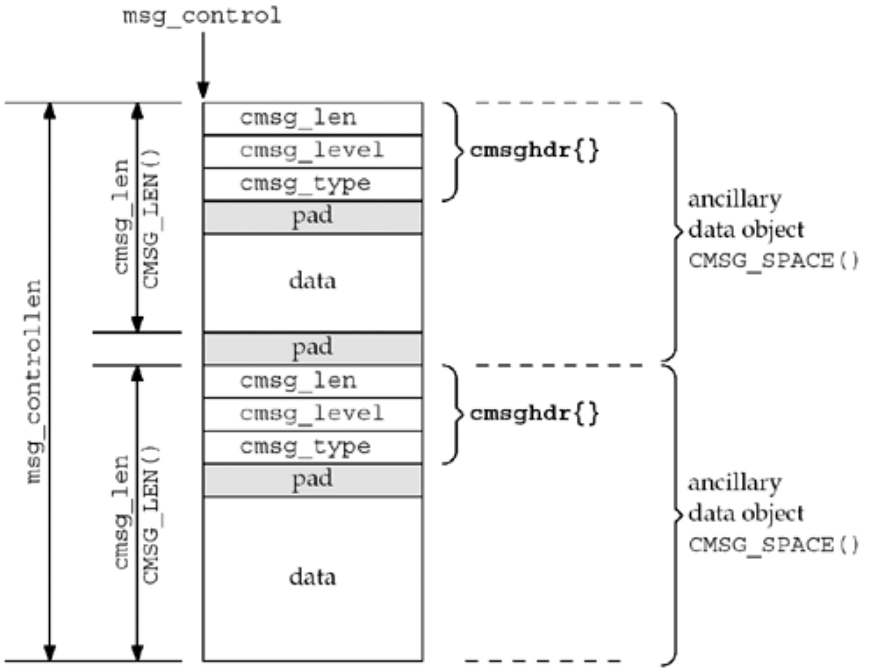
| Flag | Examined by: send *flags* sendto *flags* sendmsg *flags* | Examined by: recv *flags* recvfrom *flags* recvmsg *flags* | Returned by: recvmsg msg_flags |
|---|---|---|---|
| MSG_DONTROUTE | • | | |
| MSG_DONTWAIT | • | • | |
| MSG_PEEK | | • | |
| MSG_WAITALL | | • | |
| MSG_EOR | • | | • |
| MSG_OOB | • | • | • |
| MSG_BCAST | | | • |
| MSG_MCAST | | | • |
| MSG_TRUNC | | | • |
| MSG_CTRUNC | | | • |
| MSG_NOTIFICATION | | | • |



# Ancillary data - `cmsghdr` Structure

```
struct cmsghdr {
  socklen_t  cmsg_len;   /* length in bytes, including this structure */
  int        cmsg_level; /* originating protocol */
  int        cmsg_type;  /* protocol-specific type */
      /* followed by unsigned char cmsg_data[] */
};
```

| Protocol | cmsg_level | cmsg_type | Description |
|---|---|---|---|
| IPv4 | IPPROTO_IP | IP_RECVDSTADDR | Receive destination address with UDP datagram |
| | | IP_RECVIF | Receive interface index with UDP datagram |
| IPv6 | IPPROTO_IPV6 | IPV6_DSTOPTS | Specify destination options |
| | | IPV6_HOPLIMIT | Specify hop limit |
| | | IPV6_HOPOPTS | Specify hop-by-hop options |
| | | IPV6_NEXTHOP | Specify next-hop address |
| | | IPV6_PKTINFO | Specify packet information |
| | | IPV6_RTHDR | Specify routing header |
| | | IPV6_TCLASS | Specify traffic class |
| Unix domain | SOL_SOCKET | SCM_RIGHTS | Send/receive descriptors |
| | | SCM_CREDS | Send/receive user credentials |



# Socket I/O Summary

| Function | Any descriptor | Only socket descriptor | Single read/write buffer | Scatter/ gather read/write | Optional flags | Optional peer address | Optional control information |
|---|---|---|---|---|---|---|---|
| `read, write` | • | | • | | | | |
| `readv, writev` | • | | | • | | | |
| `recv, send` | | • | • | | • | | |
| `recvfrom, sendto` | | • | • | | • | • | |
| `recvmsg, sendmsg` | | • | | • | • | • | • |

# Socket and Standard I/O

- Buffering in Standard I/O library
  - fully buffered: all stream except for terminal devices
  - line buffered : terminal devices
  - unbuffered: stderr
- Caution
  - Socket에 standard I/O functions(fgets, fputs)를 쓰면 fully buffered됨

# Advanced UDP Sockets

Chap 22

# More on UDP

- Determining destination address of a UDP datagram
  - Wild card address can receive unicast, broadcast, and multicast datagrams on any interface
- Need some features for reliabilility
  - timeout and retransmission
  - handle lost datagrams
  - support sequence number

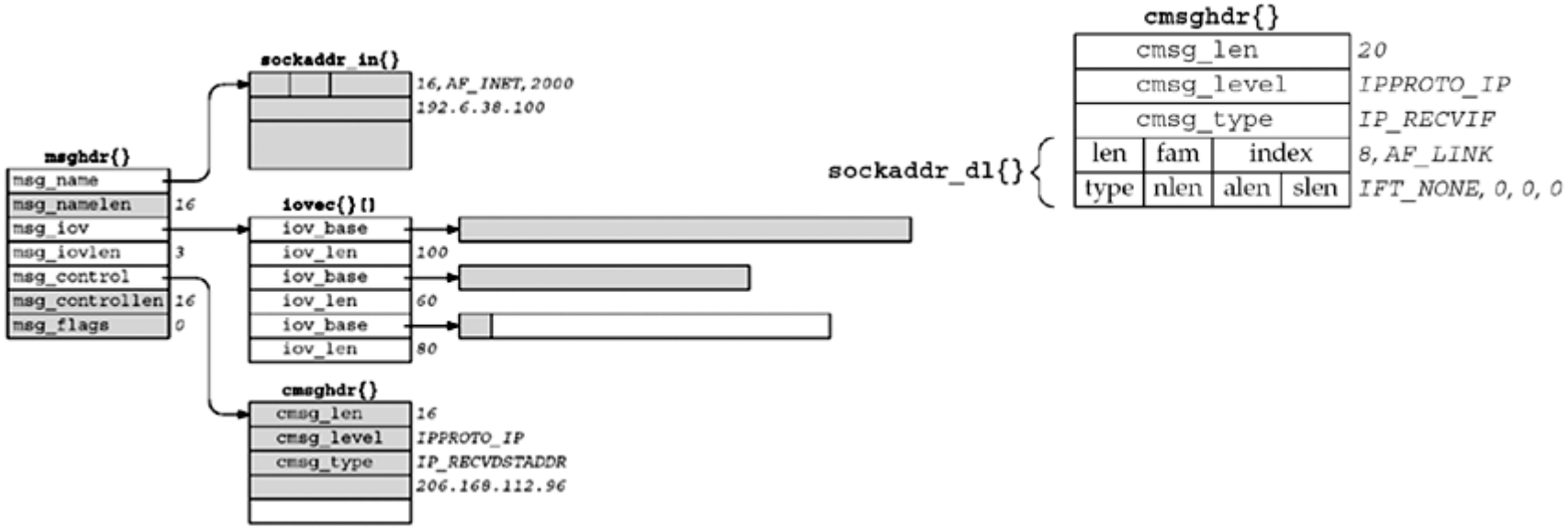
# Receiving Flags, Destination IP addr, and Interface Index

- Use `recvmsg`
  - returns `msg_flags` value
    - MSG_BCAST: broadcast address
    - MSG_MCAST: multicast address
    - MSG_TRUNC: datagram truncated
    - MAG_CTRUNC: control info truncated
  - returns destination addr of the received datagram

```
setsockopt(sockfd, IPPROTO_IP, IP_RECVDSTADDR, &on, size(on));
```

  - return index of the interface on which the datagram was received

```
setsockopt(sockfd, IPPROTO_IP, IP_RECVIF, &on, size(on));
```



# Datagram Truncation

- If received UDP datagram > application buffer, the datagram will be truncated
- Three possible scenarios on datagram truncation (depending upon implementation)
  - Discard the excess bytes and return MSG_TRUNC flag (Berkeley-derived implementation, Posix1.g)
  - Discard the excess bytes but do not tell the application (Solaris 2.5)
  - Keep the excess bytes and return them in subsequent read operatation
- Allocate application buffer > largest datagram
  - If equal, error

# When to Use UDP instead of TCP

- Adv. Of UDP
  - supports broadcasting and multicasting
  - no overhead for connection setup or teardown
    - UDP requires 2 packets to exchange a request and a reply
    - TCP requires about 10 packets to exchange assuming new TCP connection is established for each request-reply exchange
- Features of TCP that are not provided by UDP
  - positive ACK, reTx of lost packet, duplicate packet detection, sequencing of packets
  - windowed flow control
  - slow start and congestion avoidance
- Recommendation of UDP Usage
  - must be used for broadcast or multicast applications
    - desired level of error control must be added
  - can be used for simple request-reply applications
    - error detection must be needed
  - should not be used for bulk data transfer

# Adding Reliability to a UDP Application

- Add sequence numbers to detect lost, duplicated, or out-of-ordered packets
- Add timeout and retransmission
  - How to estimate retransmission timeout (RTO) - Jacobson

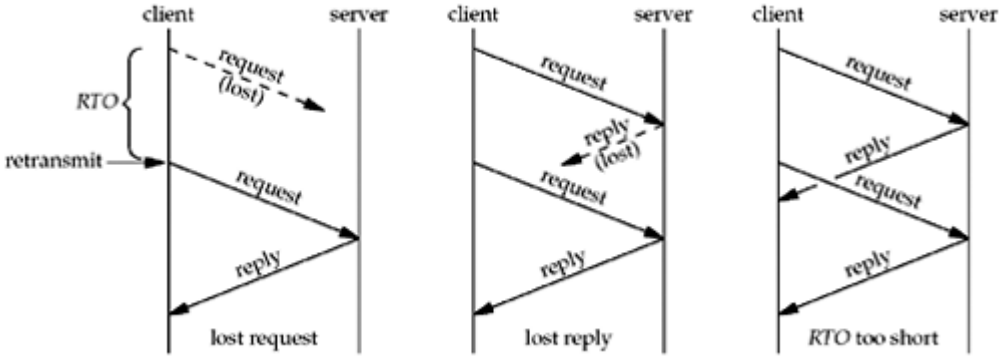
$$delta = measuredRTT - srtt$$

$$srtt \leftarrow srtt + g \times delta$$

$$rttvar \leftarrow rttvar + h(|delta| - rttvar)$$

$$RTO = srtt + 4 \times rttvar$$

  - When estimate RTO ? - Karn
    - Only when we receive a reply to a request that is not retransmitted, update the RTT estimators
    - Retransmission ambiguity problem



# Example

```
static sigjmp_buf jmpbuf;
{
    .  .  .
    form request

    signal(SIGALRM, sig_alrm); /* establish signal handler */
    rtt_newpack();             /* initialize rexmt counter to 0 */
sendagain:
    sendto();

    alarm(rtt_start());        /* set alarm for RTO seconds */
    if (sigsetjmp(jmpbuf, 1) != 0) {
        if (rtt_timeout())     /* double RTO, retransmitted enough? */
            give up
        goto sendagain;        /* retransmit */
    }
    do {
        recvfrom();
    } while (wrong sequence#);

    alarm(0);                  /* turn off alarm */
    rtt_stop();                /* calculate RTT and update estimators */

    process reply
    .  .  .
}

void
sig_alrm(int signo)
{
    siglongjmp(jmpbuf, 1);
}
```

# Concurrent UDP Servers

- Most UDP servers are iterative, but use concurrent server where the processing of client request takes long time