Elementary UDP Sockets

Elementary UDP Sockets

connectionless, unreliable, datagram

- *recvfrom* and *sendto* functions
- UDP echo server
- UDP echo client
- Verify received responses
- *connect* function with UDP
- Rewrite *dg_cli* function with *connect*
- Lack of flow control with UDP
- Determine outgoing interface with UDP
- TCP and UDP echo server using *select*

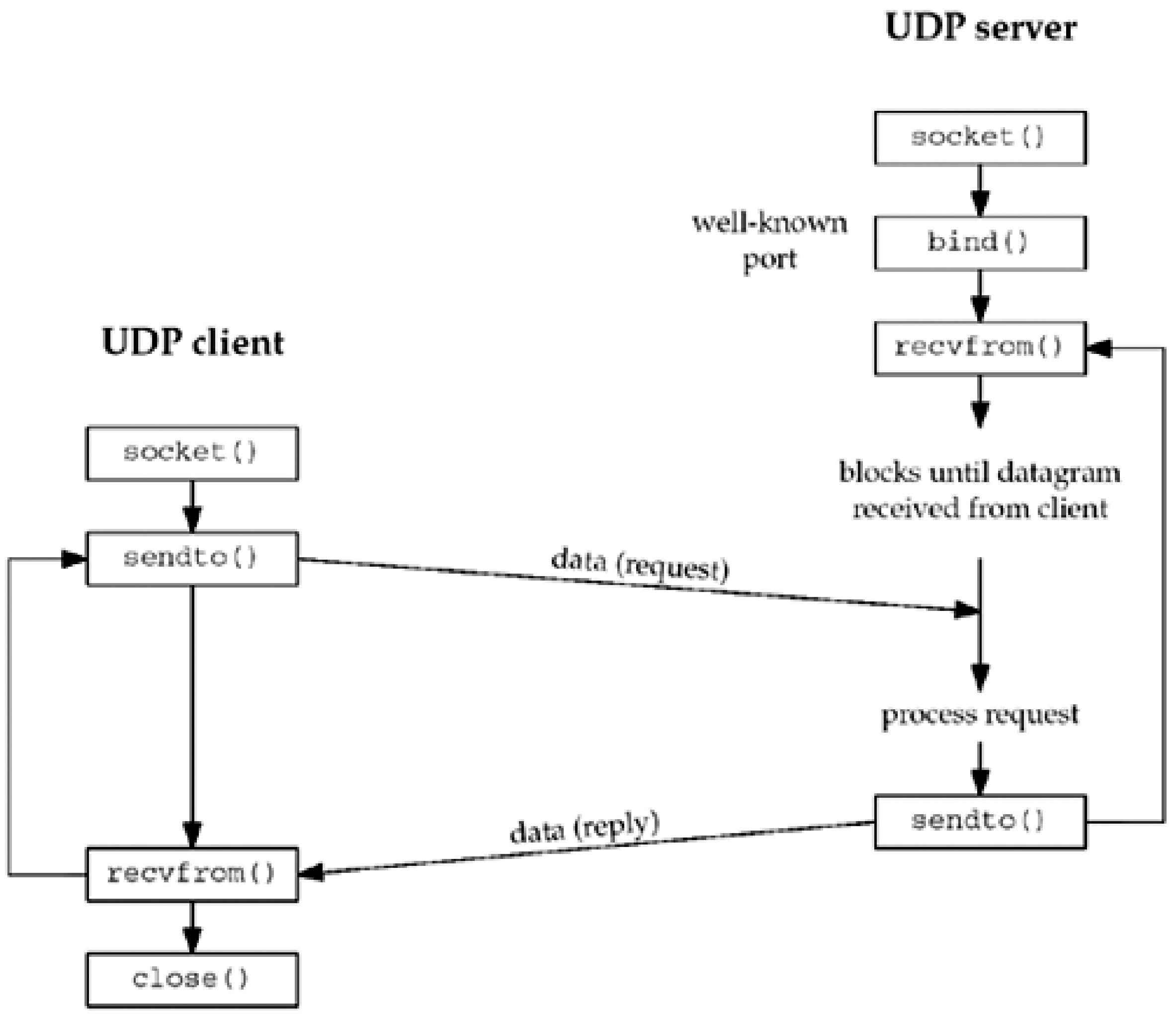
Introduction

- There are some fundamental differences between applications written using TCP versus those that use UDP.
- These are because of the differences in the two transport layers:
  - UDP is a connectionless, unreliable, datagram protocol, quite unlike the connection-oriented, reliable byte stream provided by TCP.
  - Nevertheless, there are instances when it makes sense to use UDP instead of TCP, some popular applications are built using UDP: DNS, NFS, and SNMP, for example.

Introduction

- In UDP, the client does not establish a connection with the server.
  - Instead, the client just sends a datagram to the server using the sendto function which requires the server’s address of the destination as a parameter.
- The server does not accept a connection from a client.
  - Instead, the server just calls the recvfrom function, which waits until data arrives from some client.
  - recvfrom returns the protocol address of the client, along with the datagram, so the server can send a response to the correct client.

Socket functions for UDP client/server



RECVFROM AND SENDTO FUNCTIONS

recvfrom and sendto Functions

- These two functions are similar to the standard read and write functions, but three additional arguments are required.

```
#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 and sendto Functions

- The first three arguments, sockfd, buff, and nbytes, are identical to the first three arguments for read and write: descriptor, pointer to buffer to read into or write from, and number of bytes to read or write.
- The flags argument, we will discuss the recv, send, recvmsg, and sendmsg functions in chapter 14. we do not need them with our simple UDP client/server example now. Thus, we will always set the flags to 0.

recvfrom and sendto Functions

- The sendto function:
  - The to argument for sendto function is a socket address structure containing the protocol address (e.g., IP address and port number) of where the data is to be sent.
  - The size of this socket address structure is specified by addrlen.
- The final two arguments to sendto are similar to the final two arguments to connect:
  - We fill in the socket address structure with the protocol address of where to send the datagram (in the case of UDP) or with whom to establish a connection (in the case of TCP).

recvfrom and sendto Functions

- The recvfrom function
  - The recvfrom function fills in the socket address structure pointed to by from with the protocol address of who sent the datagram.
  - The number of bytes stored in this socket address structure is also returned to the caller in the integer pointed to by addrlen.
- The final two arguments to recvfrom are similar to the final two arguments to accept:
  - The contents of the socket address structure upon return tell us who sent the datagram (in the case of UDP) or who initiated the connection (in the case of TCP).
- Note:
  - The final argument to sendto is an integer value, while the final argument to recvfrom is a pointer to an integer value (a value-result argument).

recvfrom and sendto Functions

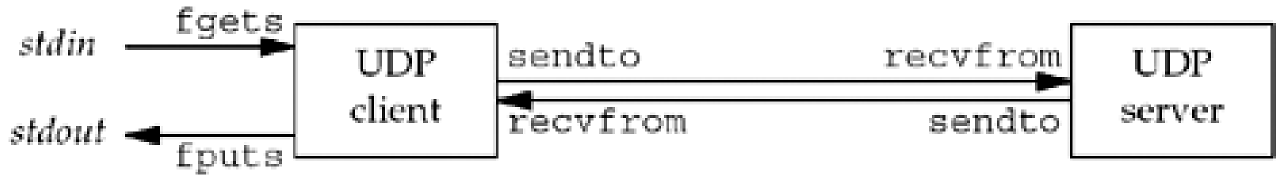
- Both functions return the length of the data that was read or written as the value of the function.
  - The recvfrom function, with a datagram protocol, the return value is the amount of user data in the datagram received.
  - Writing a datagram of length 0 is acceptable.
  - In the case of UDP, this results in an IP datagram containing an IP header (normally 20 bytes for IPv4 and 40 bytes for IPv6), an 8-byte UDP header, and no data.
  - This also means that a return value of 0 from recvfrom is acceptable for a datagram protocol: It does not mean that the peer has closed the connection, as does a return value of 0 from read on a TCP socket. Since UDP is connectionless, there is no such thing as closing a UDP connection.

recvfrom and sendto Functions

- If the from argument to recvfrom is a null pointer, then the corresponding length argument (addrlen) must also be a null pointer, and this indicates that we are not interested in knowing the protocol address of who sent us data.

UDP ECHO SERVER: MAIN FUNCTION

Simple echo client/server using UDP



UDP echo server
udpcliserv/udpserv01.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));
}
```

Simple echo client/server using UDP

- 7–12
  - We create a UDP socket by specifying the second argument to socket as SOCK_DGRAM (a datagram socket in the IPv4 protocol). As with the TCP server example, the IPv4 address for the bind is specified as INADDR_ANY and the server's well-known port is the constant SERV_PORT from the unp.h header.
- 13
  - The function dg_echo is called to perform server processing.

UDP ECHO SERVER: DG_ECHO FUNCTION

dg_echo Function

```
#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);
}
}
```

dg_echo function
lib/dg_echo.c

Read datagram, echo back to sender

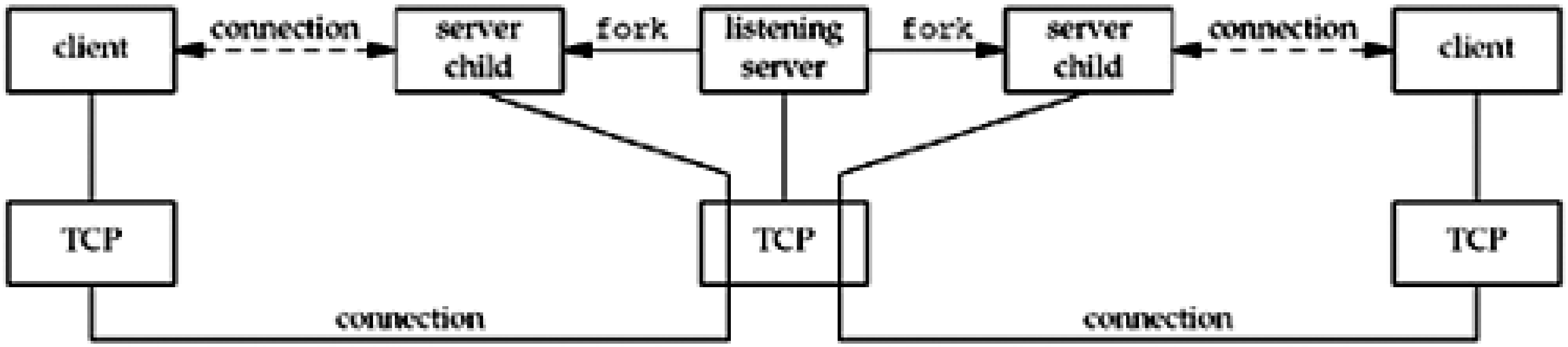
- 8–12
  - This function is a simple loop that reads the next datagram arriving at the server's port using recvfrom and sends it back using sendto.
- There are numerous details to consider.
  - First, this function never terminates. Since UDP is a connectionless protocol, there is nothing like an EOF as we have with TCP.
  - Next, this function provides an iterative server, not a concurrent server as we had with TCP. There is no call to fork, so a single server process handles any and all clients. In general, most TCP servers are concurrent and most UDP servers are iterative.

Read datagram, echo back to sender

- There is implied queuing taking place in the UDP layer for this socket. Indeed, each UDP socket has a receive buffer and each datagram that arrives for this socket is placed in that socket receive buffer.
  - the process calls recvfrom, the next datagram from the buffer is returned to the process in a first-in, first-out (FIFO) order.
  - if multiple datagrams arrive for the socket before the process can read what's already queued for the socket, the arriving datagrams are just added to the socket receive buffer. But, this buffer has a limited size.

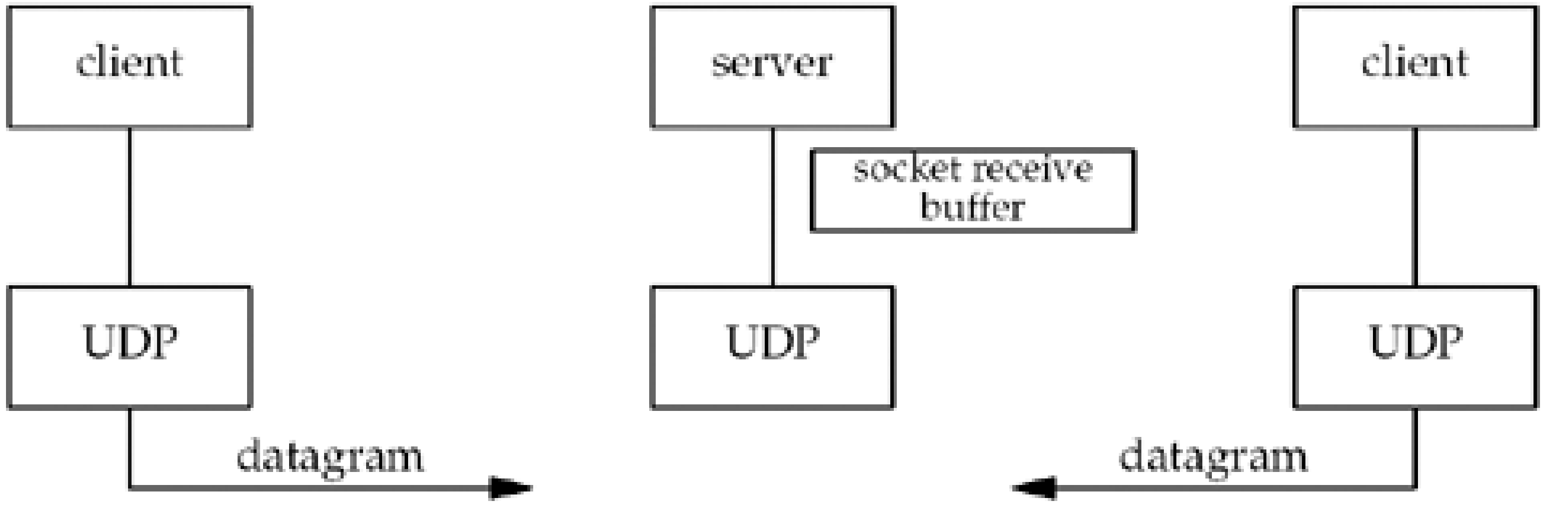
Summary of TCP client/server with two clients

- In the TCP client/server, there are two connected sockets and each of the two connected sockets on the server host has its own socket receive buffer.
- the server.



Summary of UDP client/server with two clients.

- There is only one server process and it has a single socket on which it receives all arriving datagrams and sends all responses. That socket has a receive buffer into which all arriving datagrams are placed.



The Protocol Independent

- The main function of ***udpserv01.c*** is protocol-dependent (it creates a socket of protocol AF_INET and allocates and initializes an IPv4 socket address structure), but the dg_echo function is protocol-independent.
- The reason dg_echo is protocol-independent is because the caller (the main function in our case) must allocate a socket address structure of the correct size, and a pointer to this structure, along with its size, are passed as arguments to dg_echo.
- The function dg_echo never looks inside this protocol-dependent structure:
  - It simply passes a pointer to the structure to recvfrom and sendto. recvfrom fills this structure with the IP address and port number of the client, and since the same pointer (pcliaddr) is then passed to sendto as the destination address, this is how the datagram is echoed back to the client that sent the datagram.

UDP ECHO CLIENT: MAIN FUNCTION

UDP echo client

```
#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
udpcliserv/udpcli01.c

UDP echo client

- 9–12
  - An IPv4 socket address structure is filled in with the IP address and port number of the server. This structure will be passed to dg_cli, specifying where to send datagrams.
- 13–14
  - A UDP socket is created and the function dg_cli is called.

UDP ECHO CLIENT: DG_CLI FUNCTION

dg_cli Function

dg_cli function
lib/dg_cli.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);
n = Recvfrom(sockfd, recvline, MAXLINE, 0, NULL, NULL);
recvline[n] = 0; /* null terminate */
Fputs(recvline, stdout);
}
}
```

dg_cli Function

- 7–12
- There are four steps in the client processing loop:
  - read a line from standard input using fgets, send the line to the server using sendto, read back the server's echo using recvfrom, and print the echoed line to standard output using fputs.
- Our client has not asked the kernel to assign an ephemeral port to its socket. (With a TCP client, we said the call to connect is where this takes place.)
- With a UDP socket, the first time the process calls sendto, if the socket has not yet had a local port bound to it, that is when an ephemeral port is chosen by the kernel for the socket. As with TCP, the client can call bind explicitly, but this is rarely done.

dg_cli Function

- As with the server function dg_echo, the client function dg_cli is protocol-independent, but the client main function is protocol-dependent. The main function allocates and initializes a socket address structure of some protocol type and then passes a pointer to this structure, along with its size, to dg_cli.

LOST DATAGRAMS

UDP Lost Datagram

- If a client datagram is lost (say it is discarded by some router between the client and server), the client will block forever in its call to recvfrom in the function dg_cli, waiting for a server reply that will never arrive.
- Similarly, if the client datagram arrives at the server but the server's reply is lost, the client will again block forever in its call to recvfrom.
- A typical way to prevent this is to place a timeout on the client's call to recvfrom.

UDP Lost Datagram

- Just placing a timeout on the recvfrom is not the entire solution.
- For example, if we do time out, we cannot tell whether our datagram never made it to the server, or if the server's reply never made it back. If the client's request was something like "transfer a certain amount of money from account A to account B" (instead of our simple echo server), it would make a big difference as to whether the request was lost or the reply was lost.

VERIFYING RECEIVED RESPONSE

VERIFYING RECEIVED RESPONSE

- What we can do is change the call to recvfrom to return the IP address and port of who sent the reply and ignore any received datagrams that are not from the server to whom we sent the datagram.
- First, we change the client main function in Figure 8.7 to use the standard echo server.
  - servaddr.sin_port = htons(SERV_PORT); with
  - servaddr.sin_port = htons(7);

udpcliserv/dgcliaddr.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);
    }
}
```

verifies returned socket address

- 12–18
  - In the call to recvfrom, we tell the kernel to return the address of the sender of the datagram. We first compare the length returned by recvfrom in the value-result argument and then compare the socket address structures themselves using memcmp.
- 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)

verifies returned socket address

- We specified the IP address that does not share the same subnet as the client.
- The IP address returned by recvfrom (the source IP address of the UDP datagram) is not the IP address to which we s

Solutions for Reply IP address

- One solution is for the client to verify the responding host's domain name instead of its IP address by looking up the server's name in the DNS, given the IP address returned by recvfrom.
- Another solution is for the UDP server to create one socket for every IP address that is configured on the host, bind that IP address to the socket, use select across all these sockets (waiting for any one to become readable), and then reply from the socket that is readable.

SERVER NOT RUNNING

Server Not Running

- If we do so and type in a single line to the client, nothing happens. The client blocks forever in its call to recvfrom, waiting for a server reply that will never appear.
- First we start tcpdump (wireshark) on the host macosx, and then we start the client on the same host, specifying the host freebsd4 as the server host. We then type a single line, but the line is not echoed.
- macosx % udpcli01 172.24.37.94
- hello, world

tcpdump output when server process not started

```
1 0.0                    arp who-has freebsd4 tell macosx
2 0.003576 ( 0.0036)    arp reply freebsd4 is-at 0:40:5:42:d6:de
3 0.003601 ( 0.0000)    macosx.51139 > freebsd4.9877: udp 13
4 0.009781 ( 0.0062)    freebsd 4 > macosx: icmp: freebsd4 udp port 9877 unreachable
```

1. an ARP request and reply are needed before the client host can s
2. In line 3, we see the client datagram sent but the server host responds in line 4 with an ICMP "port unreachable."end the UDP datagram to the server host.
3. Instead, the client blocks forever in the call to recvfrom.
4. We call this ICMP error an asynchronous error. The error was caused by sendto, but sendto returned successfully.

Solution for server process not started

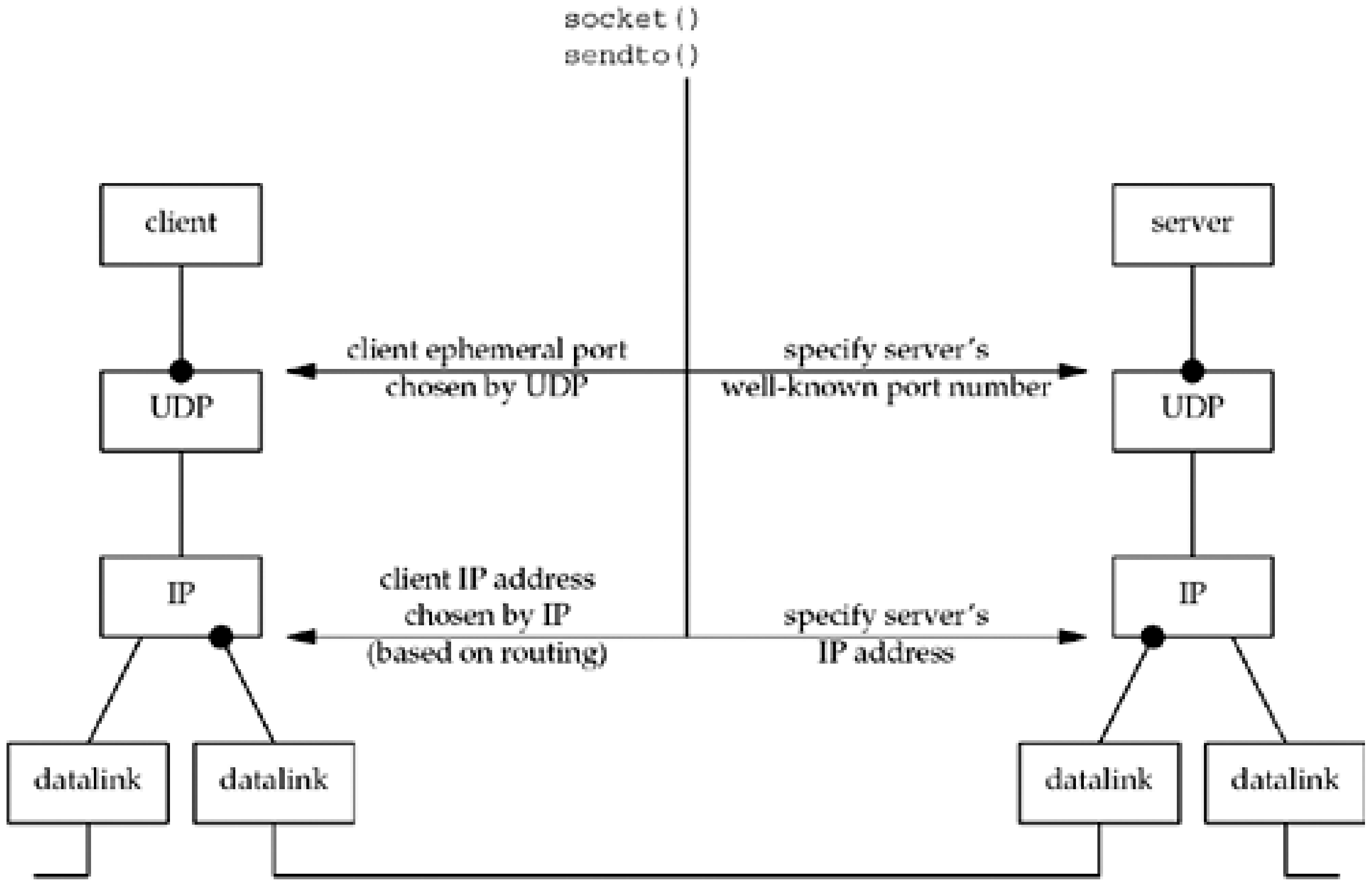
- Consider a UDP client that sends three datagrams in a row to three different servers (i.e., three different IP addresses) on a single UDP socket.
- The client then enters a loop that calls recvfrom to read the replies.
- Two of the datagrams are correctly delivered (that is, the server was running on two of the three hosts) but the third host was not running the server.
- This third host responds with an ICMP port unreachable. This ICMP error message contains the IP header and UDP header of the datagram that caused the error.

Solution for server process not started

- Linux returns most ICMP "destination unreachable" errors even for unconnected sockets, as long as the SO_BSDCOMPAT socket option is not enabled. All the ICMP "destination unreachable" errors are returned, except codes 0, 1, 4, 5, 11, and 12.

SUMMARY OF UDP EXAMPLE

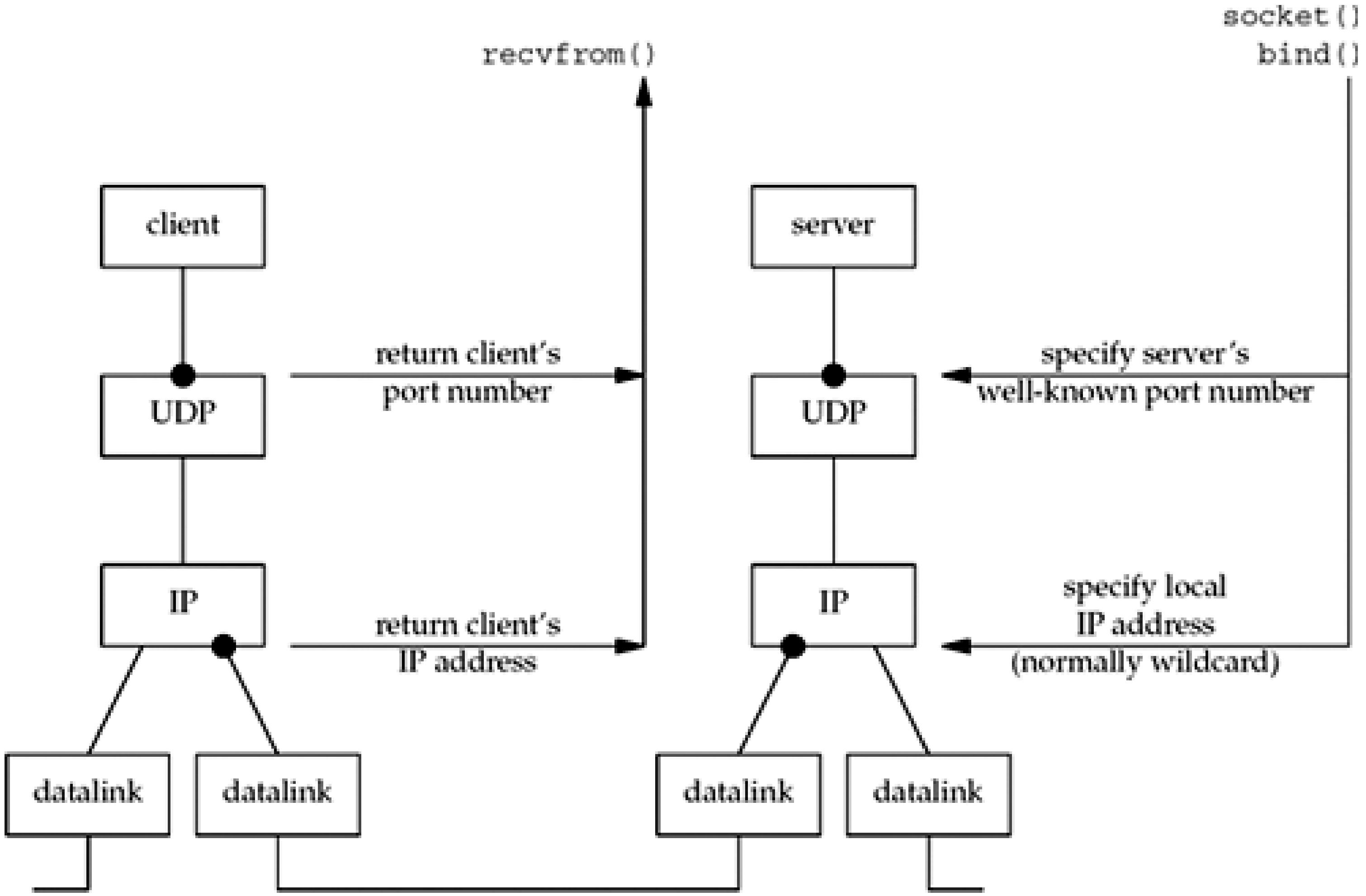
UDP client's perspective


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

UDP client's perspective

- The client must specify the server's IP address and port number for the call to sendto.
- Normally, the client's IP address and port are chosen automatically by the kernel
  - The client can call bind if it so chooses.
  - Two values are chosen by the kernel, the client's ephemeral port is chosen once, on the first sendto, and then it never changes.
- The client's IP address, however, can change for every UDP datagram that the client sends, assuming the client does not bind a specific IP address to the socket.

UDP server's perspective

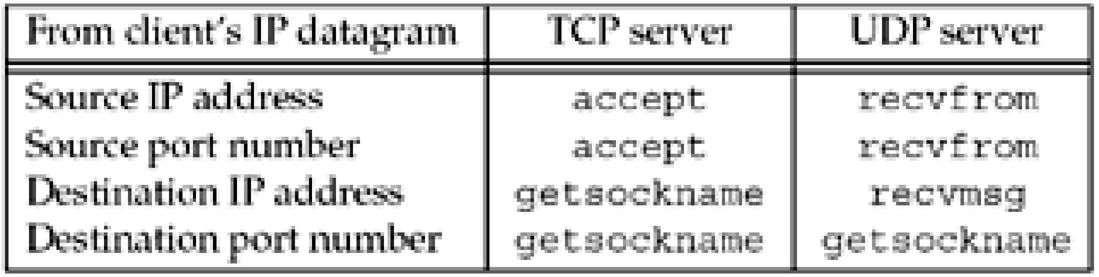

recvfrom()
socket()
bind()
client
server
return client's port number
specify server's well-known port number
UDP
UDP
IP
IP
return client's IP address
specify local IP address (normally wildcard)
datalink
datalink
datalink
datalink

UDP server's perspective

- There are at least four pieces of information that a server might want to know from an arriving IP datagram:
  - the source IP address,
  - destination IP address,
  - source port number, and
  - destination port number.

Information available to server from arriving IP datagram

- A TCP server always has easy access to all four pieces of information for a connected socket, and these four values remain constant for the lifetime of a connection.
- The destination IP address can only be obtained by
  - IPv4: setting the IP_RECVDSTADDR socket option
  - IPv6: the IPV6_PKTINFO socket option and then calling recvmsg instead of recvfrom.



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

Information available to server from arriving IP datagram

- UDP is connectionless, the destination IP address can change for each datagram that is sent to the server.
- A UDP server can also receive datagrams destined for one of the host's broadcast addresses or for a multicast address.

CONNECT FUNCTION WITH UDP

Connect Function in UDP

- Since an asynchronous error is not returned on a UDP socket unless the socket has been connected.
  - Call connect for a UDP socket. but this does not result in anything like a TCP connection: no three-way handshake in UDP.
- Instead, the kernel just checks for any immediate errors (e.g., an obviously unreachable destination), records the IP address and port number of the peer (from the socket address structure passed to connect), and returns immediately to the calling process.

Connect Function in UDP

- With this capability, we must now distinguish between
  - An unconnected UDP socket, the default when we create a UDP socket
  - A connected UDP socket, the result of calling connect on a UDP socket

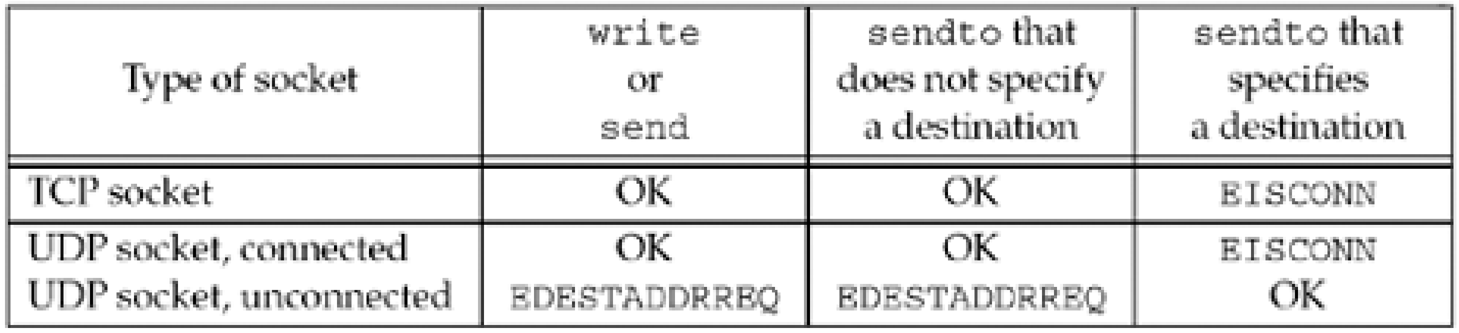
A Connected UDP Socket

- A connected UDP socket, three things change, compared to the default unconnected UDP socket:

1. We do not need to use recvfrom to learn the sender of a datagram, but read, recv, or recvmsg instead.
   - Similar to TCP, we can call sendto for a connected UDP socket, but we cannot specify a destination address.
2. We do not need to use recvfrom to learn the sender of a datagram, but read, recv, or recvmsg instead.
   - a connected UDP socket exchanges datagrams with only one IP address
3. Asynchronous errors are returned to the process for connected UDP sockets.
   - unconnected UDP sockets do not receive asynchronous errors

TCP and UDP sockets

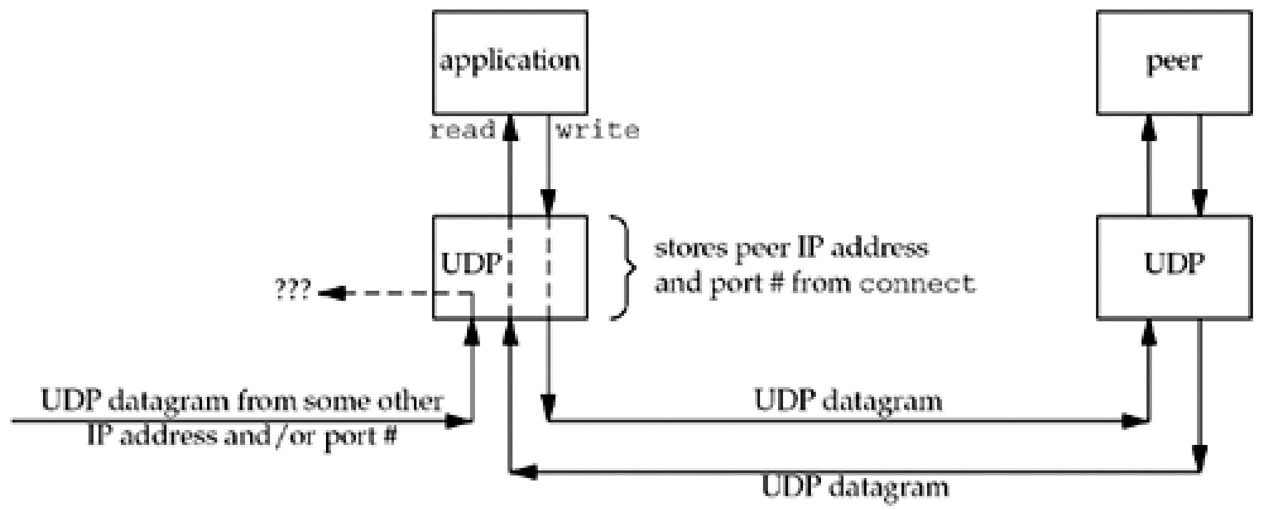
- The POSIX specification states that an output operation that does not specify a destination address on an unconnected UDP socket should return ENOTCONN, not EDESTADDRREQ.



Type of socket	`write` or `send`	`sendto` that does not specify a destination	`sendto` that specifies a destination
TCP socket	OK	OK	`EISCONN`
UDP socket, connected	OK	OK	`EISCONN`
UDP socket, unconnected	`EDESTADDRREQ`	`EDESTADDRREQ`	OK

Connected UDP socket

- The application calls connect,
  - specifying the IP address and port number of its peer.
  - uses read and write to exchange data with the peer.

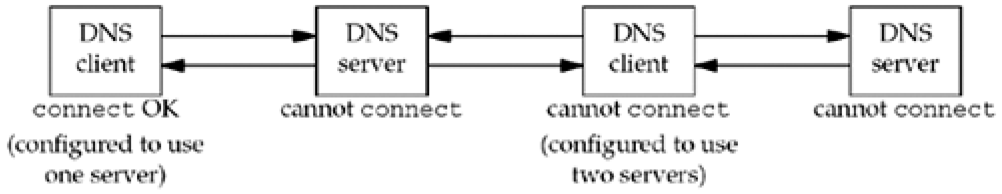

application
read
write
peer
UDP
stores peer IP address and port # from connect
UDP
???
UDP datagram from some other IP address and/or port #
UDP datagram
UDP datagram

Connected UDP socket

- A UDP client or server can call connect only if that process uses the UDP socket to communicate with exactly one peer.
- Normally, it is a UDP client that calls connect, but there are applications in which the UDP server communicates with a single client for a long duration (e.g., TFTP); in this case, both the client and server can call connect.

DNS clients and servers

- A DNS client can be configured to use one or more servers, normally by listing the IP addresses of the servers in the file /etc/resolv.conf.
  - If a single server is listed (the leftmost box in the figure), the client can call connect
  - if multiple servers are listed (the second box from the right in the figure), the client cannot call connect.
- Also, a DNS server normally handles any client request, so the servers cannot call connect.



Calling connect Multiple Times

- A process with a connected UDP socket can call connect again for that socket for one of two reasons:
  - To specify a new IP address and port
  - To unconnect the socket

Performance

- When an application calls sendto on an unconnected UDP socket, it involves the following six steps by the kernel:
  - Connect the socket
  - Output the first datagram
  - Unconnect the socket
  - Connect the socket
  - Output the second datagram
  - Unconnect the socket

Performance

- When an application knows it will be sending multiple datagrams to the same peer, it is more efficient to connect the socket explicitly.
- Calling connect and then calling write two times involves the following steps by the kernel:
  - Connect the socket
  - Output first datagram
  - Output second datagram

DG_CLI FUNCTION (REVISITED)

dg_cli Function That Calls *connect*

udpcliserv/dgcliconnect.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 That Calls *connect*

- The changes are the new call to connect and replacing the calls to sendto and recvfrom with calls to write and read.
- This function is still protocol-independent since it doesn't look inside the socket address structure that is passed to connect.

dg_cli Function That Calls *connect*

- Running the program on the host macosx, specifying the IP address of the host freebsd4 (which is not running our server on port 9877), we have the following output:
  - macosx % udpcli04 172.24.37.94
  - hello, world
  - read error: Connection refused
- The tcpdump output:
- macosx % tcpdump
  - 1 0.0 macosx.51139 > freebsd4.9877: udp 13
  - 2 0.006180 (0.0062) freebsd4 > macosx: icmp: freebsd4 udp port 9877 unreachable
- The ICMP error is mapped by the kernel into the error ECONNREFUSED, which corresponds to the message string output by our err_sys function: "Connection refused."

LACK OF FLOW CONTROL WITH UDP

Sending a Fixed Number of Datagrams

- UDP not having any flow control. First, we modify our dg_cli function to send a fixed number of datagrams.

udpcliserv/dgcliloop1.c

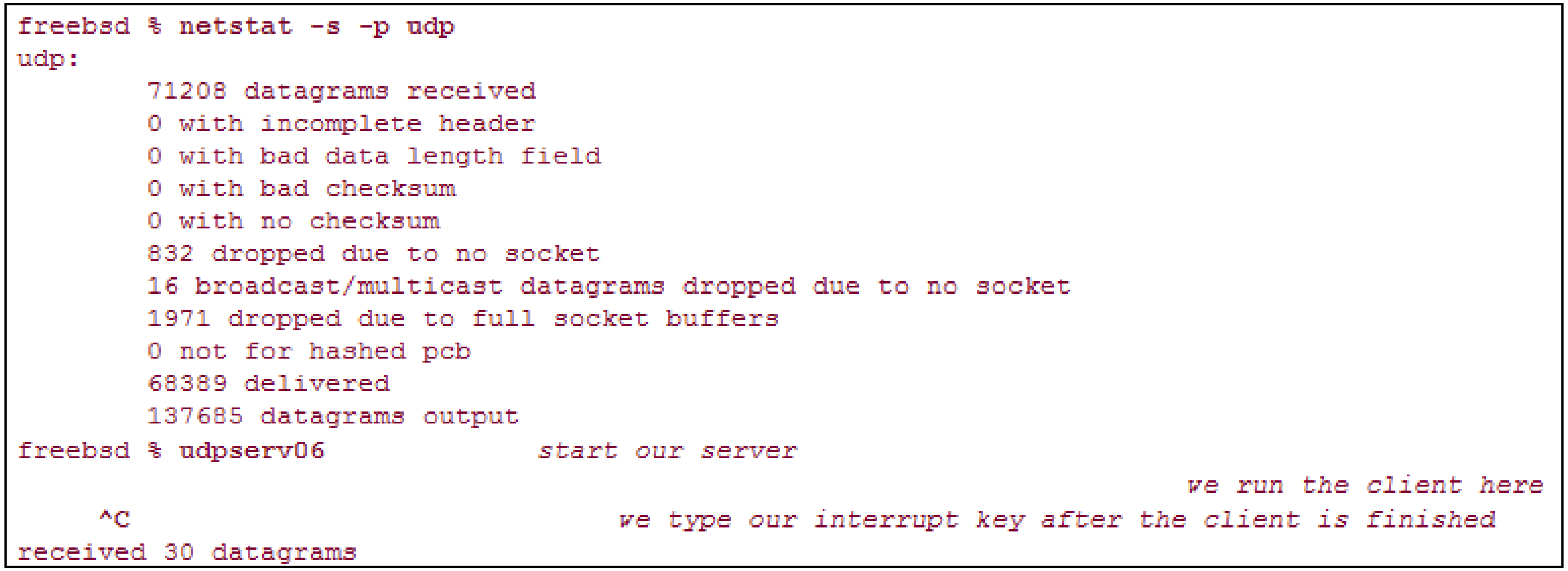
```
#include "unp.h"
#define NDG 2000 /* datagrams to send */
#define DGLEN 1400 /* length of each datagram */
void
dg_cli(FILE *fp, int sockfd, const SA *pservaddr, socklen_t servlen)
{
    int i;
    char sendline[DGLEN];
    for (i = 0; i < NDG; i++) {
        Sendto(sockfd, sendline, DGLEN, 0, pservaddr, servlen);
    }
}
```

udpcliserv/dgecholoop1.c

```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);
}
```

Output on server host

- The client sent 2,000 datagrams, but the server application received only 30 of these, for a 98% loss rate.



```
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
```

UDP Socket Receive Buffer

- The default size of the UDP socket receive buffer under FreeBSD is 42,080 bytes, which allows room for only 30 of our 1,400-byte datagrams.
- We increase the size of the socket receive buffer, expect the server to receive additional datagrams.
- The dg_echo function sets the socket receive buffer to 240 KB. We can change this with the SO_RCVBUF socket option.

udpcliserv/dgecholoop2.c

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

UDP Socket Receive Buffer

- If we run this server on the Sun and the client on the RS/6000, the count of received datagrams is now 103.
- While this is slightly better than the earlier example with the default socket receive buffer.

DETERMINING OUTGOING INTERFACE WITH UDP

Determine the Outgoing Interface

- A connected UDP socket can also be used to determine the outgoing interface that will be used to a particular destination.
- The kernel chooses the local IP address (assuming the process has not already called bind to explicitly assign this).
- This local IP address is chosen by searching the routing table for the destination IP address, and then using the primary IP address for the resulting interface.

udpcliserv/udpcli09.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);
}
```

Determine the Outgoing Interface

- We have the following output:
  - freebsd % udpcli09 206.168.112.96
  - local address 12.106.32.254:52329

  - freebsd % udpcli09 192.168.42.2
  - local address 192.168.42.1:52330

  - freebsd % udpcli09 127.0.0.1
  - local address 127.0.0.1:52331

TCP AND UDP ECHO SERVER USING SELECT

Combined TCP and UDP Echo Server Using *select*

- A single server using *select* to multiplex a TCP socket and a UDP socket:
  - create listening TCP socket
  - create UDP socket
  - establish signal handler for SIGCHLD
  - prepare for *select*
  - call *select*
  - handle new client connection
  - handle arrival of datagram

TCP and UDP Echo Server Using Select

```
#include      "unp.h"                          udpcliserv/udpservselect.c
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);
```

```
/* create 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);
           /* create 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 */
```

A listening TCP socket is created that is bound to the server's well-known port.

A UDP socket is also created and bound to the same port.

A signal handler is established for SIGCHLD because TCP connections will be handled by a child process.

```
/* 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");
                }
```

initialize a descriptor set for select and calculate the maximum of the two descriptors for which we will wait.

call select, waiting only for readability on the listening TCP socket or readability on the UDP socket. Since our sig_chld handler can interrupt our call to select, we handle an error of EINTR.

We accept a new client connection when the listening TCP socket is readable, fork a child, and call our str_echo function in the child.

```
        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 udpservselect02 */
```

If the UDP socket is readable, a datagram has arrived. We read it with recvfrom and send it back to the client with sendto.

Summary

- Lots of features in TCP are lost with UDP:
  - detecting lost packets, retransmitting, verifying responses as being from correct peer, flow control, etc.
- Some reliability can be added
- UDP sockets may generate asynchronous errors reported only to connected sockets
- Use a request-reply model

Assignment

1. What is the largest length that we can pass to sendto for a UDP/IPv4 socket, that is, what is the largest amount of data that can fit into a UDP/IPv4 datagram? Modify ~/src/lib/dg_cli.c to send maximum-size UDP datagrams, read it back, and print the number of bytes returned by recvfrom() to calculate the average throughput of UDP.

- Rewrite the udpservselect.c to a TCP and UDP daytime server using *select which* sends back to the same client the daytime message. This type of echoing is used by the network engineers to check the daytime information from a remote system.