# Chapter 9
## Iterative Connectionless Server

`passiveUDP(const char *service);`
set up a UDP server socket
parameter: name/port number of service
`return passivesock(service,"udp",5);`

```
int passivesock(const char* service,
       const char* transport, int qlen)
```

Can do TCP or UDP (transport)

- server waits for a connection (TCP) or message (UDP) on this socket.
- this will be the "well-known port" to which the client tries to connect (TCP) or send a message (UDP).

parameters:
name or port number of the service
tcp/udp indicator
`qlen` ignored on UDP sockets

This socket can be used by iterative or concurrent servers

# passivesock
# Detailed Comments

see Section 9.2 for code

```
struct servent *pse;
struct protent *ppe;
struct sockaddr_in sin;
int s, type;
```

`pse` and `ppe` are used in setting up the `sin` structure.
The `sin` structure will be used for the `bind`.
`s` will be our socket
`type` will be `IPPROTO_UDP` (in our case)

```
memset(&sin,0,sizeof(sin));
sin.sin_family=AF_INET;
sin.sin_addr.s_addr=INADDR_ANY;
```

Initialize the internet address structure.
Our protocol (family) is always `AF_INET`.

The `INADDR_ANY` says we are not doing any firewall stuff.
Running on a host that is a gateway (a machine with several internet interfaces).
Each interface will have its own internet address.
This server will accept messages/connections on any internet address that belongs to the host.
This also allows us to move the server software to another host with no changes.

```
if (pse=getservbyname(service,transport))
```
We will take two guesses, this is guess 1:
it is the name of a service (like `"telnet"`).
if guess one is successful we go to the "then"
if it fails we go to the "else"

```
sin.sin_port =
      htons(ntohs((u_short)pse->s_port)+portbase);
```

Guess 1 worked, extract the port number from the service entry structure

With `portbase==0` the math does nothing.

Ports below 1000 are reserved (root access only)—
this math allows us to add an offset (`portbase`) so non-systems types can build client/server programs that use names such as `telnet`.

If `portbase==2000` then `telnet` would evaluate to 2023.

Math: convert to an integer, add the base, convert back to network standard

Guess 2: it is a port number (like `"2345"`).

```
else if
((sin.sin_port=htons((u_short)atoi(service)))==0)
  errexit("....");
```

This is the same format we saw in the client for guessing that it was a port number.

Try to convert the string to an integer `atoi`
put that integer into network standard order.
On failure `atoi` returns 0, which takes you to the error exit.

Note that gethostbyname is not needed, the server receives messages and replies to whoever sent them. It doesn't initiate interaction with another machine

```
if ((ppe=getprotobyname(transport))==0)
  errexit("...");
if (strcmp(transport,"udp")==0)
  type = SOCK_DGRAM;
else
  type = SOCK_STREAM;
```

This is the same as in the client.

`transport` should be either `"tcp"` or `"udp"`
The error exit will not occur on either of these two, it will only happen if it is called with some third and unexpected value.

Now that we know the request is valid we allocate the socket

```
s=socket(PF_INET,type,ppe->p_proto);
```

If this is a TCP request, this will be the master socket to be used by the server.

If this is a UDP request, this will be the (only) socket to be used by the server.

```
if (bind(s,(struct sockaddr *)&sin,sizeof(sin))<0)
  errexit("can't bind...");
```

Remember, clients need to know where to find the server, it's port must be known by the clients.
Terminology: the server is at a *well-known-address*

`bind` attachs the socket to the well-known-address

The port and address (`INADDR_ANY`) set up in the address structure previously are used.

Emphasis:
The bind on the server uses the server's address(es) and port.
The connect on the client uses the server's address and port.

A `bind` failure indicates wrong priviledge (not root) or the port is already in use.
Debugging: before testing your new version of a server, be sure to kill the old version (or bind will fail). When a server is killed, the port is not released immediately.

Two processes may be bound to the same port (using fork), if the socket is set up before the fork.
In this case one process (randomly) will get the connection/message.

```
if (type==SOCK_STREAM && listen(s,qlen)<0)
  errexit("can't listen...");
```

`listen` is only done for TCP connections
if the listen request for a tcp connection fails we error exit.
after listen call, connections will be queued for acceptance.
before listen call, the server is not available (not responding)

`qlen`: the maximum number of unaccepted clients that are allowed to be waiting. (Usually $\leq 5$ required by system)

```
return s;
```

The socket is set up; it now may be used.
Return the descriptor (number).

Review: either the socket correctly sets up or `passivesock` causes the program to exit with an error message.

# Server vs. Client (Summary)

| *Server* | *Client* |
|---|---|
| Bind/listen | Connect |
| Specify server's address | Specify server's address |
| Specifies own address | Specifies others address |
| Uses the specified port | Uses any port |
| Accepts connect or message from any port | Connects or sends to a specific port |
| passivesock | connectsock |

# The Connectionless Time Server (Details)

see Section 9.4 for code

```
int main(int argc, char *argv[])
{
  struct sockaddr_in fsin;
  char *service="time";
  char buf[1];
  int sock;
  time_t now;
  unsigned int alen;
```

This is a UDP time server, it could be used by your client.

In general, the main program will be similar for all of Comer's servers and the difference will be inside the procedure that the main program calls.
In this case the server is simple and he places the code (and variables) inside the main program.

```
switch (argc){
  case 1:
    break;
  case 2:
    service = argv[1];
    break
  default:
    errexit(...);
}
```

This switch is similar to the client switch.

Normally this program will attach to the `"time"` port; however, you can put an argument on the command line that overrides that.

This allows testing on alternate ports before you do the real install.

Normally the switch would be followed by a procedure call;
but, as noted before, the code of this server is embedded in the main program

`sock=passiveUDP(service);`
get a server socket (UDP type)

`while(1)`
infinite loop (the server never exits)

`alen=sizeof(fsin)`

`alen` starts with the size of `fsin`,

if the actual size of the sender's address is bigger than this, it won't fit in `fsin` and an error condition will occur.

Ends with the actual size of the senders address.

All internet address sturctures have the same size, so if this is not equal to `sizeof(fsin)` something is wrong; notice the program never checks.

```
if (recvfrom(sock,buf,sizeof(buf),0,
          (struct sockaddr *)&fsin,&alen)<0)
    errexit("...");
```

Receive a message. If there is an error, `recvfrom` returns a negative number and the server exits.

`buf`: the variable that the message is to be placed into.

`sizeof(buf)`: the message better be shorter than this

Actually this server ignores the contents of the incoming message.

`fsin`: (return address) the variable that gets the address of sender of the message.
(Highly useful for sending a reply!)

```
time(&now);
now=htonl((unsigned long)(now+UNIXEPOCH));
```

Get the information the client requested, in this case the time and prepare for network shipment.

`UNIXEPOCH`: convert the number to conform to the protocol. Here we are the system's network time server, so we must conform to the measure of time specified by the RFC.

In a client-server pair what matters is that both the client and server agree on the format of the time. They could agree either on UNIX or Internet, or some other; as long as they agree.

`htonl`: by convention all numbers are converted to network standard order for shipment.

```
sendto(sock,&now,sizeof(now),0,
        (struct sockaddr *)&fsin,sizeof(fsin))
```

Send the reply

`sock`: use the same socket

`now`: send the number as a string of bytes.

`fsin`: send to the return address.

receive–the address is the source (return) address
send–the address is the destination address
(It's always the other machines address,
because you should know your own.)

Note:

An attempt to `connectTCP` to this particular server would fail (`connectsock` would get an error).
This server does only UDP.

# Summary

1) Set up a UDP socket.

(Repeat)

2) Get an incoming request (save the reply address).

3) Perform the service or get the information.

4) Prepare the information for network transfer.

5) Send the reply.

# Homework Help

The `get_port` function is designed to drop into the Comer code.

Example of using a function to return the service, where the function returns a string.

```
char *service = get_port();
```