Networks :

Make M/cs communicate

TCP, UDP

TCP - FIFO, reliable communication two "2-endpoints"

↑ ~~bit~~ byte-oriented

client - server

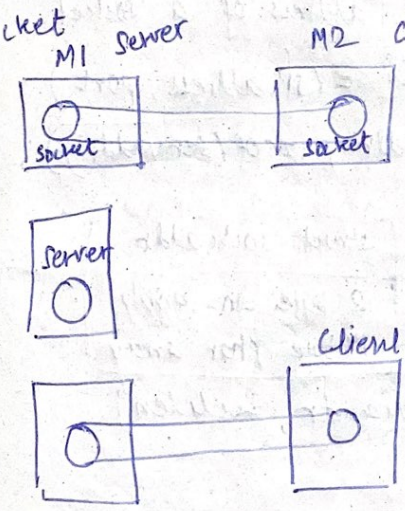
↑ ↑

Takes service. Provides service

Server :

TCP comm.

Socket



Server:

Socket ( ) — opens a socket
bind ( ) — gives an address to the sockets
listen ( ) -
accept ( ) — waits for a client to connect
send ( ) / recieve( ). — once a client connects
close( ) — closes the socket

Client:

socket() - opens a socket
bind() ?? not needed ??
connect() - connect to server

send() / recv() data

close()

```
int sockfd;
sockfd = socket(family, type, protocol)
                  |      |       |
              AF_INET  SOCK_STREAM  0

struct sockaddr_in serveraddr, cliaddr;
serveraddr.sin_family = AF_INET;
serveraddr.sin_port = htons(20000);  //network byte order
servaddr.sin_addr
inet_aton("10.5.17.1", &servaddr.sin_addr);
```

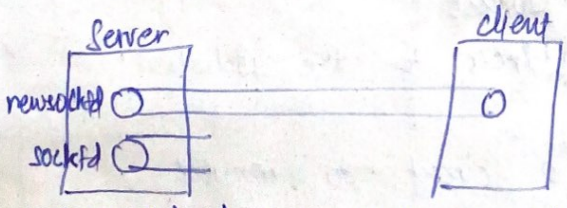
address of a socket = (IP address, port)

```
bind(sockfd, (struct sockaddr *)&servaddr, sizeof(servaddr));
listen(sockfd, 5)
clilen = sizeof(cliaddr)
newsocketfd = accept(sockfd,
                     (struct sockaddr *)&cliaddr, &clilen)
```

blocked

struct sockaddr
2 byte sin_family
14 byte char array

```
strcpy(buff, "abcd");
x= send(sockfd, buff, strlen(buff)+1, 0);
        newsockfd
```




```
y= recv(newsockfd, buff, buff_size, 0);
```

blocked

Networks

UDP

Correctedness, unreliable, no FIFO

message-oriented

① What you send in one send() will be recieved in ~~another~~ one recv() if recieved.

② If the buffer size in recv() call < what is sent, remaining data will be lost

UDP server

socket(—, SOCK_DGRAM, —)

bind()

sendto/recvfrom

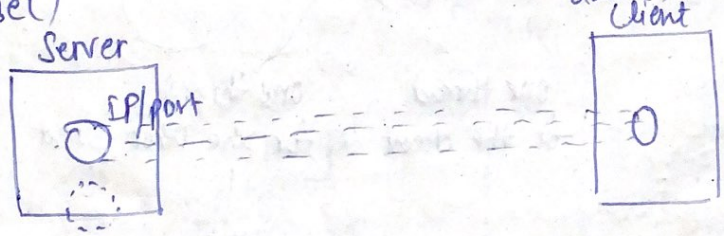
close()

UDP client

socket(—, SOCK_DGRAM, —)

send/recv ≡ sendto/recvfrom

close()



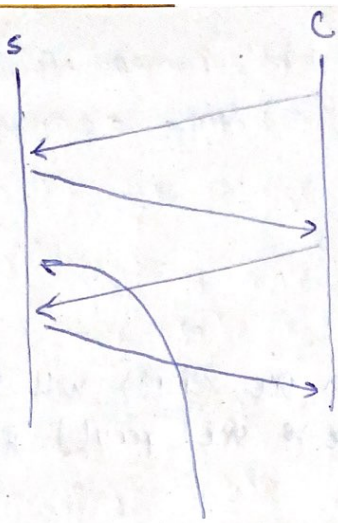
same as send

~~send~~ sendto(—, —, —, —);

(struct sock_addr *) &servaddr, sizeof(servaddr));

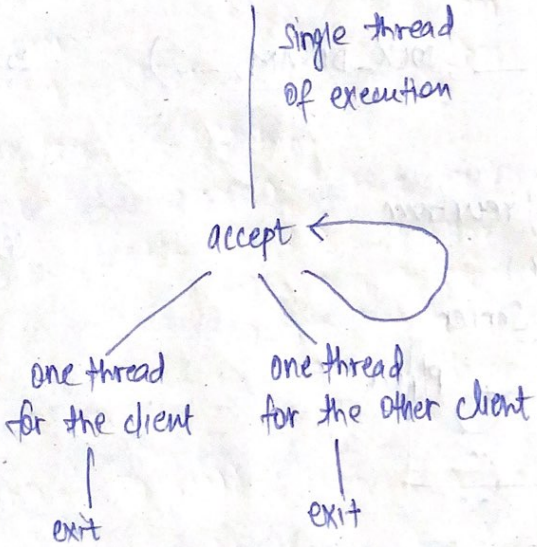
send ≡ sendto(—, —, —, —, NULL, 0);

recvfrom(—, —, —, —, (struct sock_addr *) &cliaddr, sizeof(cliaddr));



some other client can connect in b/w the previous client-server comm.

```
newsockfd = accept(sockfd, ... );
{
   file transfer
}
```




```
int x=5;
printf("%d\n", x);
if (fork() == 0)
{
   x = x+5;
   printf("%d\n", x);
}
else
{
  sleep(10);
  printf("%d\n", x);
}
```

O/P =>
5
10
5

If there is no sleep() then its upto the CPU which process can be scheduled first so the last 2 nos. can be interchanged.

x = fork()

parent process

fork()

0 → child process

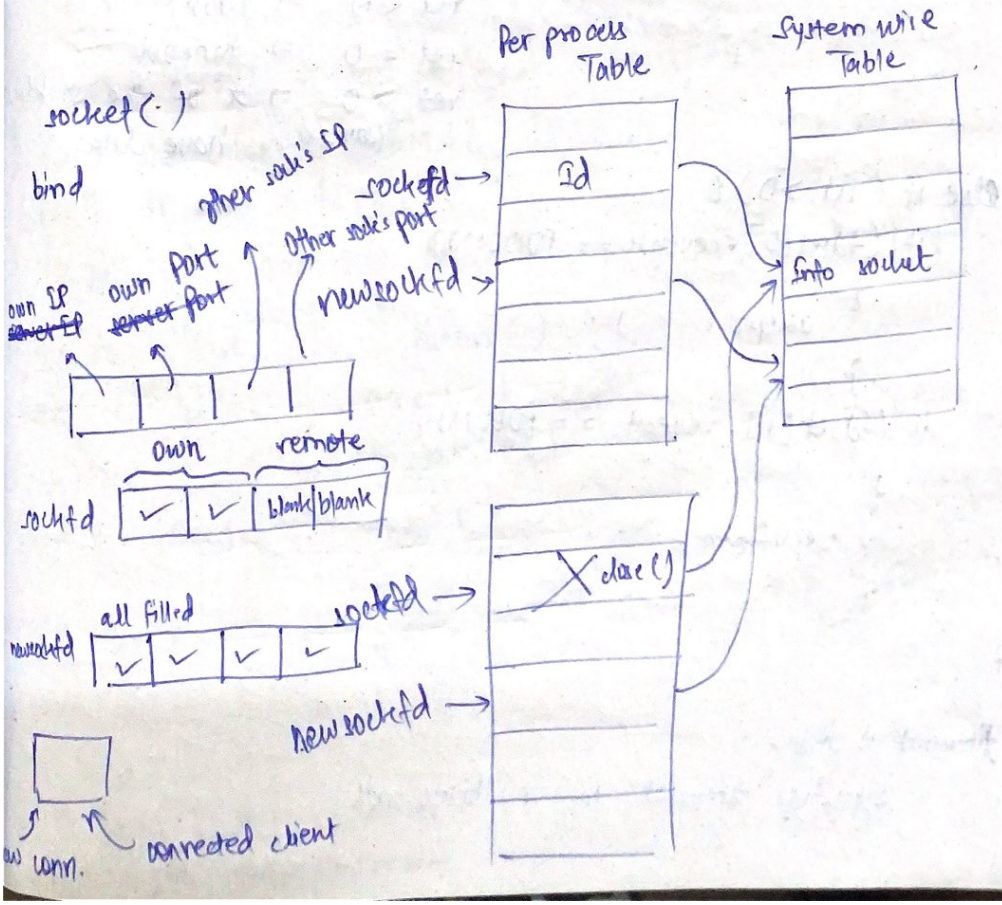
id of the child → parent process (same one)

The memory gets copied in both the processes.

x_c = 10 (was 5)

x_p = 5

```
while(1){
    newsockfd = accept(sockfd, —, —, —);
    if(fork() == 0){
        close(sockfd);
        comm. with client
        exit(0);
    }
    close(newsockfd);
}
```



Ports

0 - 1023 — cannot use these ports (well known ports)

1024 - $\frac{3}{4}$th of 65536 — registered ports

If we want to make server to accept both TCP & UDP client whichever comes first

accept recvfrom

we have a function for this

poll / select

```
struct pollfd fdset[2];
fdset[0].fd = tcpsockfd;
fdset[0].events = POLLIN;
fdset[1].fd = udpsockfd;
fdset[1].events = POLLIN;
poll(fdset, 2, timeout);
ret = poll (fdset, 2, timeout); //blocking
```

```
pollfd
{
  int fd;
  short events;
  short revents;
}
```

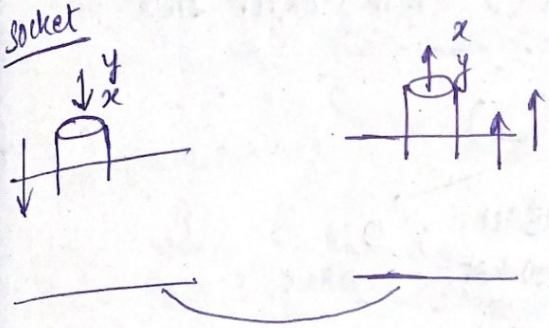
ret < 0 ⇒ error
ret = 0 ⇒ timeout
ret > 0 ⇒ x of the sock
(say x) have data

```
else if (ret > 0) {
    if (fdset[0].revents == POLLIN)
        {
          accept ( );
        }
    if (fdset[1].revents == POLLIN)
        {
          recvfrom ( );
        }
}
```

timeout ⇒ int
signifies timeout in milliseconds

Networks:

Socket



TCP socket	UDP socket
Reliable	Unreliable
In order delivery	May be out of order
Byte oriented	Message oriented

TCP server	TCP client
waits for client	asks for service

Client - server programming

TCP server

Create socket - socket()

Bind the socket - bind()
(specify IP address & port)

One m/c has one IP addr only difference is in port

Configure queue - listen()

Wait for client - accept()

Recv for client - recv()

Send to client - send()

TCP Client

Create socket - socket()

No bind -

BUT every socket has an IP, port

Because no one externally connects client IP, port is

assigned dynamically

connect() → accept() from server side

socket (— , — , —)

- AF_INET OR PF_INET (using the internet family of sockets)
- SOCK_STREAM (for TCP socket)
- 0

sockfd = socket(...)

↑ Returns socket descriptor

err = bind (sockfd, (struct sockaddr *) &servaddr, sizeof (struct sockaddr));

struct sockaddr_in serv_addr;
serv_addr.sin_family = AF_INET;
serv_addr.sin_port = 20000; (take anything > 10000)
serv_addr.sin_addr.s_addr = INADDR_ANY (accept packets from any IP address)

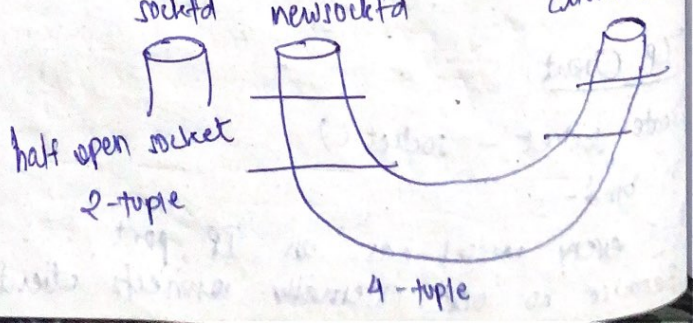
Port = 16 bits
< 65536

listen(sockfd, 5) ← while one client is being serviced by the server, 5 more clients can wait

TCP server ← Blocking call

newsockfd = accept (sockfd, (struct sockaddr *) & ----)

(Server waits for client's connection)



```
while(1){
    newsockfd = accept(---)
    // communicate with client using newsockfd
    ;
    close(newsockfd); // newsockfd is gone, but sockfd
                      is still there
}
```

send(newsockfd, buffer, len, 0);
- newsockfd: new
- buffer: from where?
- len: how many bytes
- 0

recv(newsockfd, buffer, len, 0);
- newsockfd: new
- buffer: where to put?
- len: max no. of bytes to recieve
- 0

Block if no data in socket buffer

returns no. of bytes send or recv (-1 on error)

TCP ~~gu~~ guarrantees that if I send 100 bytes on one side they will arrive on the other side in the same order but we cannot say that if they arrive altogether.

How to know what I actually recieved?

Return value of recv call will tell how many bytes we recieve.

If recv call comes out to be retun value 0 the it means that the other side has closed the connection This can be used to detect that the other end has sent all the data or not.

TCP client

sockfd = socket() - same

connect(sockfd, (struct sockaddr *) ----)

recv/send

Any application in the world has <IP addr, port>

close(sockfd);

servaddr.sin_family = AF_INET;
servaddr.sin_port = 20000;
servaddr.sin_addr.s_addr = 10.5.18.84; ✗
(cannot specify this way)

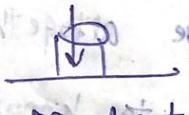
IP address = 32 bit integer

instead there are functions
inet_aton ("10.5.18.84", &serv_addr.sin_addr.s_addr);
↑
(ASCII to network)
OR
inet_pton (...)

Two applications cannot use the same port on the same m/c. Therefore system looks a port for the client ~~and then~~ and assigns it (assignment is done by the local m/c). Server always binds & client has an option to bind (may not bind).

UDP server
= socket() ← SOCK_DGRAM (for UDP socket)
bind()


UDP server


UDP client

No kind of pipes, no reliability
so no kind of accept or connect

recvfrom
(—, —, —, —, —, —)
same
client who sent

UDP client

socket()

sendto (—, —, —, —, —, —)
same
who to send

send() → where I am specifying (where this packet is supposed to go)

Here client can send to any server since we are now specifying each time which server to send

In the recvfrom & accept function we have last parameters which specify which client we have recieved from (In case of UDP) & which client we have connected to (In case of TCP) when the function comes out.

So, In case of UDP a server first need to make a recvfrom call to get the address of client and then we come to know the client's IP, port. Now, we can use it to send data to client.

All the above calls if they fail the return -1;

perror() func:

There is a number to every error.
When there is an error, the error no. value is set to some particular value.
perror() is a func where we can add our own msg then it tell the error after a colon (:)

serv_addr.sin_port = htons(20000);

htons() is used to automatically adapt to the byte orientation used on a particular m/c (Little Endian OR Big Endian)

There is prescribles standard of network byte order, it says whatever your integer format is, when you put on wire you put it in this order.

htons → host to network short
↑
because port is 16 bits

When the connect call will bind to the accept call

the cli_addr will be filled with the address of the client and the cliten will be filled up with the size of struct sockaddr. So we know which is the client we connected to.
If we give null-pointers to this still it will connect. It just add up an extra info for the client

127.0.0.1 is a special address for the local interface

Iterative server - A server which handles client one-by-one
It handles one client in one accept call., then does every thing for that client only then it goes to the other accept call again to allow the next client

---

Networks:

Now we have a case that we need our server to accepts data or send calls from a TCP or a UDP socket whichever comes first.
We know that an accept (in TCP) & recvfrom (in UDP) is a blocking call and we don't come out of them until we get a connect (in TCP) & sendto (in UDP). Therefore, we need some other mechanism to block on both TCP & UDP and come out wrt whichever arrives first. This can be done using select/poll.

Also we can have a case of a concurrent server which can process multiple clients at same time. we can handle each request by spawning a new process using fork() call.

int fork()

Program is run in a process. For each program we create a process.

1 process
|
function call
|
1 process
1 return value

1 process
|
fork()
parent process — one return value of fork
child process — a different return value of fork (0)

```
                    ← 1 process
x = fork();
if (x==0)           ← 2 process
    printf ("This \n");
else
    printf ("That \n");
```

O/P:

This OR That
That This

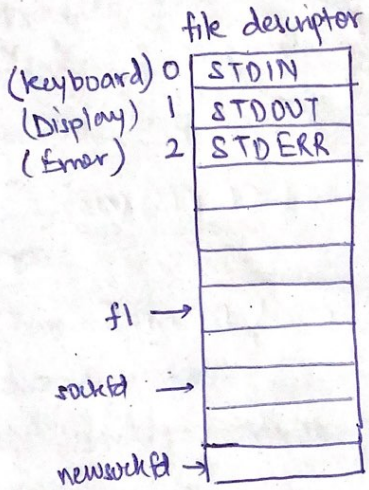
```
int d=10;
x= fork ();
if (x==0)
{
```
gets a copy of d | code for child process |
```
}
else {
```
gets a separate copy of d | code for parent process |
```
}
```

Each process gets a ~~copy of all~~ separate copy of all the variables

```
int d=10; x=fork()
if(x==0)
  {
    d=15;
  }
else
  {
    sleep(20);
    printf("%d\n", d);
  }
```

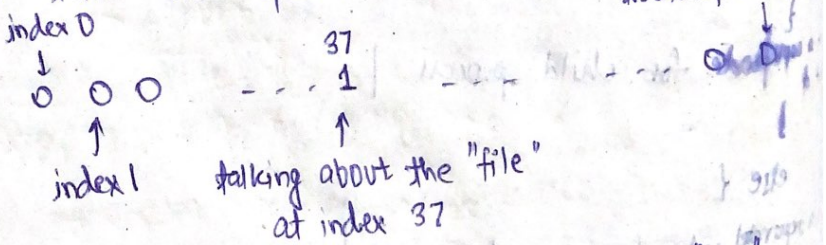
File descriptor

fl = open(...)



sockfd = socket(...)

Each of open call of whatever you open returns an index to the file descriptor table and the table contains info about the file opened



FD_SET - bit pattern with 1 bit for every "file"

```
FD_SET fd;
```

Macros to manipulate FD_SETs

```
FD_SET waitfd;
FD_ZERO(&waitfd);   // zero out all the bits
```

```
FD_SET(&waitfd, sockfd); // set the bit at index sockfd
FD_CLR(&waitfd, sockfd); // clears the bit at index sockfd
FD_ISSET(&waitfd, sockfd); // Returns 0 if sockfd is not set
                           // 1/>0 otherwise

                                        &time
select (maxfd, &waitfd, 0, 0, &t);
        (int)  (FD_SET*)              time out
```

timeout is defined as:

```
struct timeval {
    int tv_sec;   // seconds
    int tv_usec;  // microseconds
};
```

maxfd = max (all sockfd) + 1

it is telling that we only think about bits till maxfd rest are unimportant (more for optimisation purpose)

sockfd1, sockfd2

wait on sockfd1 & sockfd2

come out if data in any one or 3 sec have passed

```
struct timeval t;
FD_SET fd;
FD_ZERO(&fd);
FD_SET(&fd, sockfd1);
FD_SET(&fd, sockfd2);
t.tv_sec = 3;
t.tv_usec = 0;
err = select(max(sockfd1, sockfd2)+1, &fd, 0, 0, &t);
```

↓ Blocks until either data in sockfd1, or data in sockfd2, or 3 sec over

Suppose data comes only in sockfd1, select will come out But we don't know where in which socket we have data Here system comes to rescue. The system sets only the bit which has data the remaining ones will be set to zero.

```
if (FD_ISSET (&fd, sockfd1))
    /* data in sockfd1 */
{
    recv (sockfd1);
}
if (FD_ISSET (&fd, sockfd2))
    /* data in sockfd2 */
{
    recv (sockfd2);
}
/* timeout */ if (FD_ISSET(&fd, sockfd1) == 0 &&
                  FD_ISSET (&fd, sockfd2) == 0)
{
}
```

How do you send a series of strings?
(How do you ~~send~~ detect when a string of strings end?)

client ⟶ server

⟵ string of strings

"10.14.3.7 \0", "12.5 ---" --- .

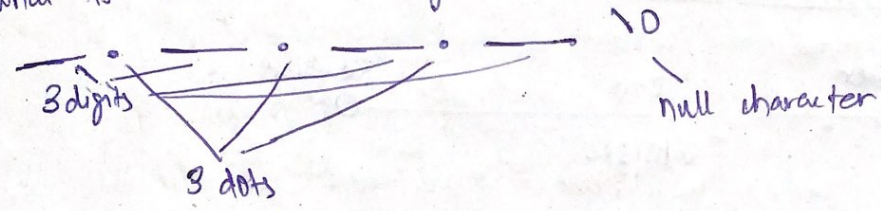
We can do this by first sending the number of strings and then the other side can just accept that no. of strings.

—— 10 —— 10 —— \0\0

indicator of end of string of strings

Here we have UDP protocol so how can a client know what is the length of ~~te~~ string coming from the server, if the server is sending 300 bytes and client has 100 bytes buffer the 200 bytes will just gets dropped.

So you can't do it in a single PC. But if you are sending IPs (DNS server) then you ~~kn~~ know ~~the~~ what is the max length of an IP address



max_size = 16 bytes

TCP socket (sockfd1)
block on accept()

UDP socket
block on recvfrom()

```
select (— , - - -)
if (FD_ISSET (&fd_set, sockfd1)
 {  newsockfd = accept (- - -)  // does not block as some dummy packets are recv as connection request
      fork()
        :
      recv()
 }
 if (FD_ISSET (&fd_set, sockfd2))
  {
    recvfrom()
  }
```

To see ip addresses of DNS name

nslookup www.google.com

HTTP - Text Based Protocol

Command
Header

fieldname: value

→ GET ——— 1.1
Connection: close
ACCEPT_LANGUAGE:
ACCEPT_TYPE: application/pdf

Command
Request
Response
Entity

→ PUT ——— 1.1

How to make send/recv non blocking?

recv(—, —, —, flag)
0

|| MSG_DONTWAIT — for non-blocking
If there is nothing in the buffer, it will fail
return value < 0
ERR_NO set to EWOULDBLOCK OR EAGAIN
check ERR_NO in if and proceed accordingly.

|| MSG_PEEK
Return data from the beginning of the recv queue.
A subsequent call will return the same data.

|| MSG_WAITALL
This flag indicates that it will block until the full data request specified in parameters are not recved.

How to make all blocking calls non-blocking?

fcntl(sockfd, F_SETVAL, O_NONBLOCK)
command
F_GETVAL, ———

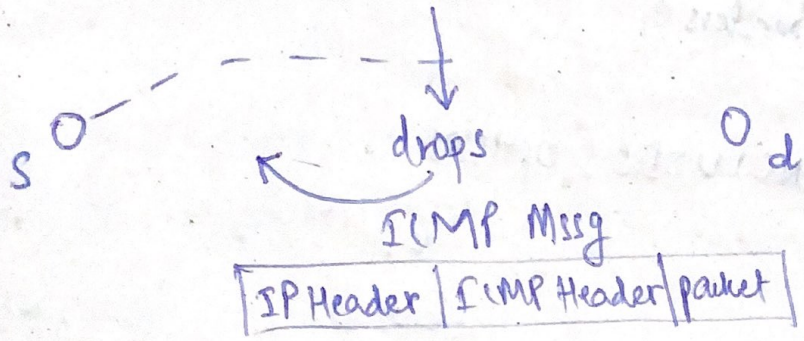
This overwrites other flag bits

2 bit flags
00 ← Block
10
overwrites → 01 ← Non block

man page F_SETFL
F_GETFL

```
val = fcntl(sockfd, F_SETVAL, 0);
fcntl(sockfd, F_SETVAL, VAL | O_NONBLOCK);
```

Internet Control Message Protocol (ICMP):



Destination Unreachable has IP header + 64 bits of IP pkt

To know which ~~data~~ pkt of IP was dropped

To know about the header & type used in higher layer protocol

IP Header	Higher Layer Header	Data

Networks Lab:

Gethostbyname ← DNS call

getsockopt

setsockopt

Raw sockets:

socket(___, SOCK_RAW, IPPROTO_ICMP)

Raw sockets — no ports

sendto / recvfrom

Suppose 5 processes opened 5 ~~raw~~ ~~to~~ raw sockets on IPPROTO_ICMP.
When OS gets an ICMP pkt it gives to <u>all</u> 5 raw sockets.

recvfrom on this will give the entire IP pkt, including IP header & ICMP header

IP header	ICMP Header	ICMP data

<u>How to send?</u>

If you don't say anything to the OS, the OS will generate its own IP header.

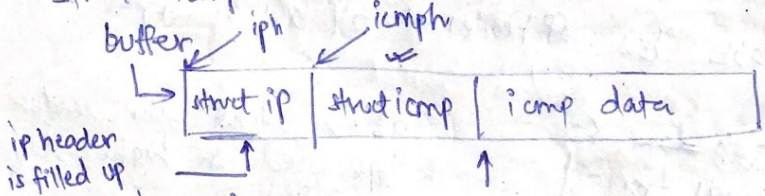
① allow setting IP header int one=1

setsockopt(sockfd, <u>IPPROTO_IP</u>, IP_HDRINCL, &one, sizeof(one))
level

Now how to set these headers?

~~struct~~ ip
struct ipheader — INCLUDES options

Similarly,
struct icmp



ip header is filled up

```
struct ip *iph = ;
" icmp *icmph;
char buff[1000];
iph buff iph = (struct ip *) buff;
iph → hlen =
iph → ver =
icmph = buff (struct icmp *) (buff + sizeof(struct ip))
icmph → type =
icmph → code =
char * data = buff + sizeof(struct ip) + sizeof(struct icmp)
```

sendto(—, buff, len, — —)

There are some ~~file~~ ~~fd~~ fields which are set ~~automati~~ automatically like checksum & for some we have options like for identity we can put our own, but if we do not put our own IP will put his own identity.

✓ Lookup alarm() for signal SIGALRM
This is for making the reciever wake up when a message is in the pipe ready to recv.

✓ setsockopt(—, SOL_SOCKET, SNDBUFFSIZE, —)
RCVBUFFSIZE

✓ funtl(, O_ASYNC)

sender

ID, seqno, sending_time
To keep track of lost or dropped pkts.