Socket programming with TCP/UDP

Sockets

- a way to speak to other programs using standard Unix file descriptors (like a pipe)
- low level communication endpoint used for processing info across a network

Types of sockets:

1. Stream Sockets (*SOCK_STREAM*)
    - reliable 2 way connected communication streams
    - uses TCP (Transmission Control Protocol)
2. Datagram Sockets (*SOCK_DGRAM*)
    - unreliable, connectionless (although they can be *connect()*'d)
    - uses UDP (User Datagram Protocol)

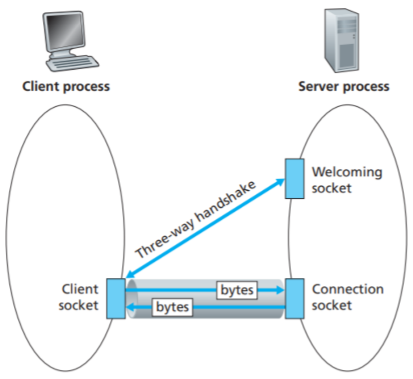
TCP API and client-server paradigm

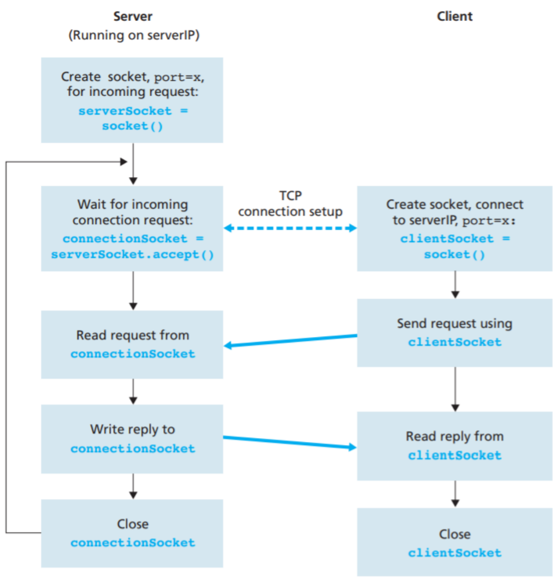
TCP is a connection-oriented protocol → before the exchange of data a TCP connection must be established, one end to the client socket and the other to the server socket.

The connection is associated to the client and server socket addresses (IP + port number). Now, the data exchange happens by droping the data into the TCP connection via its socket.

Client must contact server so:

- the server process must be running
- the server must have a special door (socket) that welcomes the client's contact
- client creates a local socket specifying the IP address and port number for the server
- the server accepts the contact and creates a new socket for that client's demand




Server
(Running on serverIP)
Client
Create socket, port=x, for incoming request: serverSocket = socket()
Wait for incoming connection request: connectionSocket = serverSocket.accept()
TCP connection setup
Create socket, connect to serverIP, port=x: clientSocket = socket()
Read request from connectionSocket
Send request using clientSocket
Write reply to connectionSocket
Read reply from clientSocket
Close connectionSocket
Close clientSocket

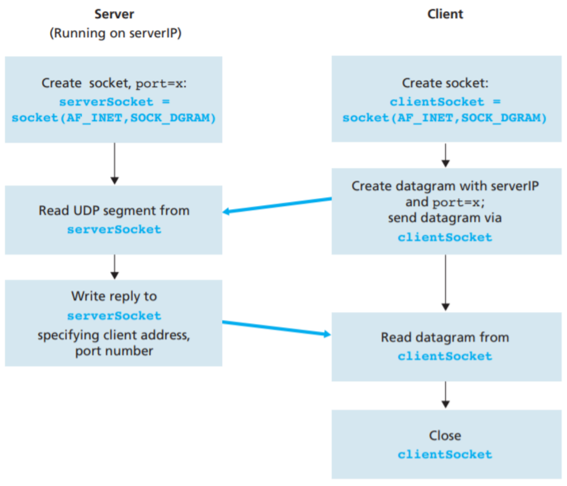
UDP API and client-server paradigm

UDP is a connection-less protocol → datagram oriented (individual messages).

We have no guarantee for datagram delivery and ordering.

Client must contact server so:

- the server process must be running
- client attaches to the packet a destination address (destination host's IP address and the destination socket's port number)
- the client's source address is also attached to the packet, but this process is done by the underlying operating system



Advantages: -less latency, higher bandwidth, broadcast (1 sender multiple receivers).

fdf

Programming

1. *Structs and data handling*

a) Socket descriptor → regular int

b) struct sockaddr → holds socket address info

```
struct sockaddr{
  unsigned short sa_family; //address family, AF_xxx
  char sa_data[14]; //14 bytes of protocol address
};
```

- sa_family will be AF_INET
   - ▼ What is AF_INET?

     Describes the type of addresses that our socket can communicate with, in this case Internet Protocol v4 addresses (IPv4)
- sa_data contains a destination address and port number for the socket

c) struct sockaddr_in → basically sockaddr but easier to manage

```
struct sockaddr_in {
  short int sin_family; //address family
  unsigned short int sin_port; //port nr
  struct in_addr sin_addr; //internet address
  unsigned char sin_zero[8]; //same size as struct sockaddr
}
```

- sin_zero should be set to all zeros with the function memset()
- sockaddr_in ↔ sockaddr so even tho socket() wants a sockaddr* you can give a sockaddr_in (cast it)

2. ***Conversions***

Two types that you can convert: short (two bytes) and long (four bytes). Functions:

- htons() -- "Host to Network Short"
- htonl() -- "Host to Network Long"
- ntohs() -- "Network to Host Short"
- ntohl() -- "Network to Host Long"

!!! put your bytes in Network Byte Order before you put them on the network.

The sin_addr and sin_port need to be in Network Byte Order because they get encapsulated. So sin_family doesn't need to because it does not get sent out on the network.

3. ***IP addresses & how to deal with them***

- inet_addr() → converts an IP address in numbers-and-dots notation into an unsigned long
- inet_ntoa() → reverse for inet_addr()

```
struct sockaddr_in ina;
ina.sin_addr.s_addr = inet_addr("10.12.110.57");
```

4. ***System calls & functions***

▼ socket()

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

int socket(int domain, int type, int protocol);
```

- domain → AF_INET
- type → what kind of socket it is (SOCK_STREAM/SOCK_DGRAM)
- protocol → set to '0' so the function will choose the correct protocol based on the type
- returns -1 on error

Socket options

```
int setsockopt(int sockfd, int level, int optname,
             const void *optval, socklen_t optlen);
```

Optname:

- SO_REUSEADDR – reuse local addresses – use it to get rid of the address already in use after you terminate your TCP/IP app abruptly
- SO_BROADCAST – enables broadcast = one sender – Universal(all) receiver(s)!

▼ bind()

= associate that socket with a port on your local machine

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

int bind(int sockfd, struct sockaddr *my_addr, int addrlen);
```

- sockfd → the socket file descriptor
- my_addr → pointer to a struct sockaddr that contains info
- addrlen → set to sizeof(struct sockaddr)
- returns -1 on error
- ports below 1024 are reserved, so choose any port right up to 65535

Some of the process of getting your own IP address and/or port can can be automated:

```
my_addr.sin_port = 0; //choose an unused port at random
my_addr.sin_addr.s_addr = INADDR_ANY; //use my IP address
//no need to put INADDR_ANY in network byte order bcs is actually 0
```

▼ connect()

= connect to a remote host

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

int connect(int sockfd, struct sockaddr *serv_addr, int addrlen);
```

▼ listen()

= you wait for incoming connections → 2 steps: listen() and accept()

```
int listen(int sockfd, int backlog);
```

- backlog → nr of connections allowed on the incoming queue

If you're gonna listen, you'll make the next sequence of system calls: socket(), bind(), listen(), accept();

▼ accept()

- returns a brand new socket file descriptor

```
int accept(int sockfd, void *addr, int *addrlen);
```

▼ send() and recv()

```
int send(int sockfd, const void *msg, int len, int flags);
```

- returns the number of bytes actually sent out

```
int recv(int sockfd, void *buf, int len, unsigned int flags);
```

- returns the number of bytes actually read into the buffer

▼ sendto() and recvfrom()

```
ssize_t sendto(int sockfd, const void *buf, size_t len, int flags,
               const struct sockaddr *dest_addr, socklen_t addrlen);
```

- returns nr. of bytes transmitted or -1 on error
- data size must fit < 64Kb
- 1 sendto must be consumed by exactly 1 recvfrom on the receiver

```
ssize_t recvfrom(int sockfd, void *buf, size_t len, int flags,
                 struct sockaddr *src_addr, socklen_t *addrlen);
```

- returns nr. of bytes received or -1 on error
- if not all data is read in one call the remaining is discarded
- cannot receive more than max 64Kb

▼ close() and shutdown()

- close the connection on your socket descriptor → close(sockfd);
- shutdown() → like close but more control; ret 0 on succes, -1 otherwise

```
int shutdown(int sockfd, int how);
//how can be : 0 - no more receives; 1 - no more sends; 2 - no more
```

▼ getpeername() — Who are you?

- The function getpeername() will tell you who is at the other end of a connected stream socket. Once you have their address, you can use inet_ntoa() or gethostbyaddr() to print or get more information. The synopsis:

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

int getpeername(int sockfd, struct sockaddr *addr, int *addrlen);
```

▼ gethostname() — Who am I?

- It returns the name of the computer that your program is running on. The name can then be used by gethostbyname(), below, to determine the IP address of your local machine.

```
#include <unistd.h>

int gethostname(char *hostname, size_t size);
```

▼ DNS

- DNS - Domain Name Service
- you tell it what the human-readable address is for a site, and it'll give you the IP address

```
#include <netdb.h>

struct hostent *gethostbyname(const char *name);
struct hostent {
        char    *h_name;
        char    **h_aliases;
        int     h_addrtype;
        int     h_length;
        char    **h_addr_list;
    };
    #define h_addr h_addr_list[0]
```

- h_name -- Official name of the host.
- h_aliases -- A NULL-terminated array of alternate names for the host.
- h_addrtype -- The type of address being returned; usually AF_INET.
- h_length -- The length of the address in bytes.
- h_addr_list -- A zero-terminated array of network addresses for the host. Host addresses are in Network Byte Order.
- h_addr -- The first address in h_addr_list.

▼ Example

```
/*
    ** getip.c -- a hostname lookup demo
    */

    #include <stdio.h>
    #include <stdlib.h>
    #include <errno.h>
    #include <netdb.h>
    #include <sys/types.h>
    #include <sys/socket.h>
    #include <netinet/in.h>
    #include <arpa/inet.h>

    int main(int argc, char *argv[])
    {
        struct hostent *h;

        if (argc != 2) {  // error check the command line
            fprintf(stderr,"usage: getip address\n");
            exit(1);
        }

        if ((h=gethostbyname(argv[1])) == NULL) {  // get the host info
            herror("gethostbyname");
            exit(1);
        }

        printf("Host name  : %s\n", h->h_name);
        printf("IP Address : %s\n", inet_ntoa(*((struct in_addr *)h->h_addr)));

       return 0;
    }
```

5. ***Example***

▼ client.py ← TCP

```
from socket import *
serverName = ’servername’
serverPort = 12000
clientSocket = socket(AF_INET, SOCK_STREAM)
clientSocket.connect((serverName,serverPort))
sentence = raw_input(‘Input lowercase sentence:’)
clientSocket.send(sentence)
modifiedSentence = clientSocket.recv(1024)
print ‘From Server:’, modifiedSentence
clientSocket.close()
```

▼ server.py ← TCP

```
from socket import *
serverPort = 12000
serverSocket = socket(AF_INET,SOCK_STREAM)
serverSocket.bind((‘’,serverPort))
serverSocket.listen(1)
print ‘The server is ready to receive’
while 1:
connectionSocket, addr = serverSocket.accept()
sentence = connectionSocket.recv(1024)
capitalizedSentence = sentence.upper()
connectionSocket.send(capitalizedSentence)
connectionSocket.close()
```

▼ client.py ← UDP

```
from socket import *
serverName = ‘hostname’  #IP address or name of the server
serverPort = 12000
clientSocket = socket(socket.AF_INET, socket.SOCK_DGRAM)
message = raw_input(’Input lowercase sentence:’)
clientSocket.sendto(message,(serverName, serverPort)) #we attach the dest address to our message
modifiedMessage, serverAddress = clientSocket.recvfrom(2048) #sv address contains the IP address and the port
print modifiedMessage
clientSocket.close()
```

▼ server.py ← UDP

```
from socket import *
serverPort = 12000
serverSocket = socket(AF_INET, SOCK_DGRAM)
serverSocket.bind((’’, serverPort))
print ”The server is ready to receive”
while 1:
  message, clientAddress = serverSocket.recvfrom(2048)
#we make use of client address because we have to know where to send our reply
  modifiedMessage = message.upper()
  serverSocket.sendto(modifiedMessage, clientAddress)
```

▼ client.c ← UDP

```
//UDP client in the internet domain

#include <stdio.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <netdb.h>
#include <stdlib.h>
#include <string.h>
#include <sys/unistd.h>

void error(char* msg){
    perror(msg);
    exit(1);
}

int main(int argc, char **argv) {
    int sockfd, length, n;
    struct sockaddr_in server, from;  //ip addressing (ip, port, type)
    struct hostent *hp; //DNS
    char buffer[1024];

    if(argc < 2){
        printf("Usage: %s <port> \n", argv[0]);
        exit(1);
    }

    sockfd = socket(AF_INET, SOCK_DGRAM, 0);

    if(sockfd < 0)
        error("Error on opening client socket");

    server.sin_family = AF_INET;
    server.sin_addr.s_addr = inet_addr("192.168.100.40");
    server.sin_port = htons(atoi(argv[1]));

    length = sizeof(struct sockaddr_in);

    printf("Please enter the message: ");
    bzero(buffer, 1024);
    fgets(buffer, 1023, stdin);

    n = sendto(sockfd, buffer, strlen(buffer), 0 , (struct sockaddr*)&server, length);

    if(n<0)
        error("Error on sending information to server");

    n = recvfrom(sockfd, buffer, 1023, 0, (struct sockaddr*)&from, &length);

    if(n<0)
        error("Error on receiving information from server");

    write(1, "Got an ack: ", 12);
    write(1, buffer, n);


    return 0;
}
```

▼ server.c ← UDP

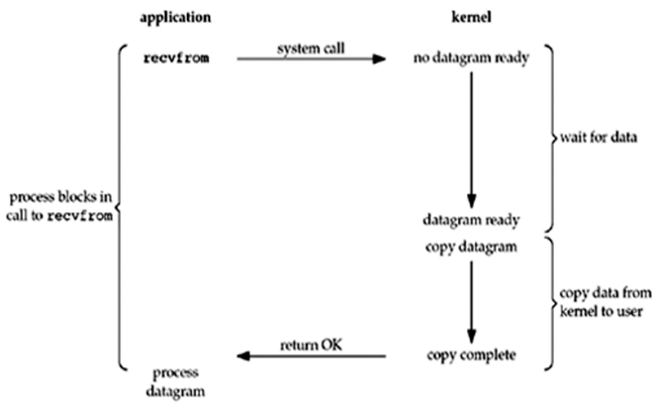
```
/*
 * Create a datagram server, the port is passed
 * as an argument. The server runs forever.
 */

#include <stdio.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>
#include <unistd.h>
#include <stdlib.h>
#include <strings.h>
#include <string.h>

void error(char* msg){
    perror(msg);
    exit(1);
}

int main(int argc, char** argv) {
    int sockfd, length, fromlen, n;
    struct sockaddr_in server;
    struct sockaddr_in from;
    char buf[1024];

    if(argc < 2){
        fprintf(stderr, "Error, no port provided\n");
        exit(1);
    }

    sockfd = socket(AF_INET, SOCK_DGRAM, 0);

    if(sockfd<0)
        error("Error on opening server socket");

    length = sizeof(server);
    bzero(&server, length);
    server.sin_addr.s_addr = INADDR_ANY;
    server.sin_family = AF_INET;
    server.sin_port = htons(atoi(argv[1]));

    if(bind(sockfd, (struct sockaddr* )&server, length) < 0)
        error("Error on binding");

    fromlen = sizeof(struct sockaddr_in);

    while(1){
        n = recvfrom(sockfd, buf, 1024, 0, (struct sockaddr*)&from, &fromlen);
        if(n<0)
            error("Error on receiving from client");
        write(1,"Received a datagram: ", 21);
        write(1, buf, n);
        n = sendto(sockfd, "Message received\n", strlen("Message received\n"), 0, (struct sockaddr*) &from, fromlen);

        if(n<0)
            error("Error on sending to the client");
    }



    return 0;
}
```

Advanced TCP/IP – I/O Modes

▼ Network operation steps - READ

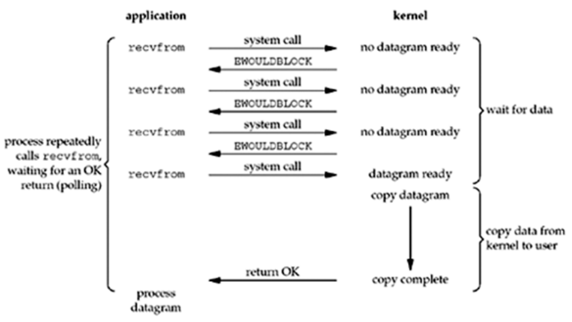
1. Wait for data to arrive from network.
2. Copying the data from the kernel to the process – i.e. copying the data from the kernel buffer into the application space.

1. ***Blocking I/O Model***



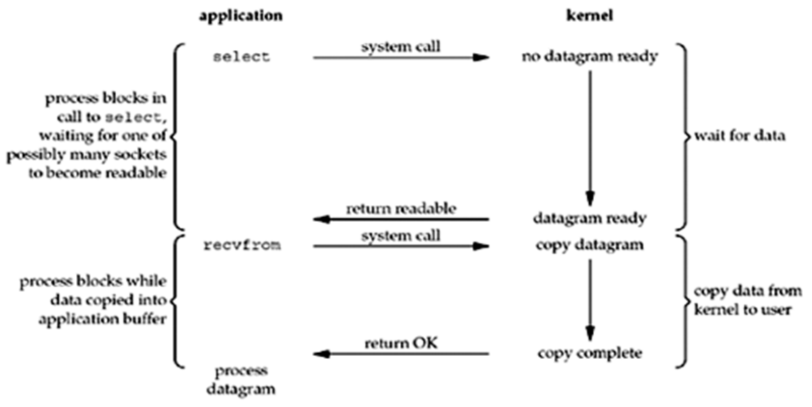
- process blocks = proces is sleeping
- process is sleeping while it waits for data
- when data arrives, the process wakes up and continues with its work

2. ***Non-Blocking I/O Model***



- we can use setsockopt to set the socket to non blocking
- we are able to process other things between calls

3. ***Multiplexed I/O Model***


application
kernel
select
system call
no datagram ready
process blocks in call to select, waiting for one of possibly many sockets to become readable
wait for data
return readable
datagram ready
recvfrom
system call
copy datagram
process blocks while data copied into application buffer
copy data from kernel to user
return OK
copy complete
process datagram

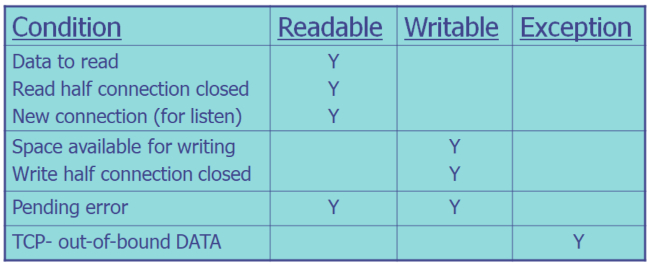
- involves using select and poll

▼ select()

```
#include <sys/select.h>
#include <sys/time.h>

int select(int maxfd+1, fd_set* readset, fd_set* writeset, fd_set *exceptset,
const struct timeval *timeout)
//readset - watches sockets  for read
//writeset - watches sockets for write
//timeout - how long?

void FD_ZERO(fd_set, *fdset); //clear all bits in fdset
void FD_SET(int fd, fd_set* fdset); //turn on the bit for fd in fdset
void FD_CLR(int fd, fd_set* fdset); //turn off the bit for fd in fdset
void FD_ISSET(int fd, fd_set* fdset); //is the fd ready?
```

- returns positive count of ready descriptors, 0 if timeout and -1 on error

Conditions for a socket to be ready



Condition	Readable	Writable	Exception
Data to read	Y		
Read half connection closed	Y		
New connection (for listen)	Y		
Space available for writing		Y	
Write half connection closed		Y	
Pending error	Y	Y	
TCP- out-of-bound DATA			Y