socket(): Creating a Socket

```c
int socket(int family, int type, int proto);
```

- **family** specifies the protocol family (PF_INET for TCP/IP).
- **type** specifies the type of service (SOCK_STREAM, SOCK_DGRAM).
- **protocol** for the specific protocol (usually 0).
- The `socket()` system call returns a socket descriptor (small integer) or a -1 on error.
- `socket()` allocates resources needed for a communication endpoint - but it does not deal with endpoint addressing.
- Passive sockets and active sockets

Specifying an Endpoint Address

- Remember that the sockets API is generic.
- There must be a generic way to specify endpoint addresses.
- TCP/IP requires an IP address and a port number for each endpoint address.
- Other protocol suites (families) may use other schemes.
Generic socket addresses

```c
struct sockaddr {
    u_char   sa_len;
    u_short  sa_family;
    char     sa_data[14];
};
```

- `sa_len` specifies the total length.
- `sa_family` specifies the address type.
- `sa_data` specifies the address value.

sockaddr

- An address that will allow me to use sockets to communicate with kids.
- Address type AF_KIDS
- Address values:
  - Andrea 1
  - Jeff 2
  - Abrar 3
  - Nancy 4

AF_KIDS

- Initializing a sockaddr structure to point to Abrar:
  ```c
  struct sockaddr Abrar;
  Abrar.sa_family = AF_KIDS;
  Abrar.sa_data[0] = 3;
  ```
AF_INET

- For AF_KIDS we only needed 1 byte to specify the address.
- For AF_INET we need:
  - 16 bit port number
  - 32 bit IP address

```c
struct sockaddr_in {
    u_char sin_len;
    u_short sin_family;
    u_short sin_port;
    struct in_addr sin_addr;
    char sin_zero[8];
};
```

struct in_addr

```c
struct in_addr {
    u_long s_addr; /* IP ADDRESS */
};
```

- in_addr just provides a name for the ‘C’ type associated with IP addresses.
- Network byte Order: all values stored in a sockaddr_in must be in network byte order:
  - sin_port a TCP/IP port number.
  - sin_addr an IP address.

TCP/IP Addresses

- We don’t need to deal with sockaddr structures since we will only deal with one protocol family.
- We can always use sockaddr_in structures.
- The C functions that make up the sockets API expect structures of type sockaddr.
Assigning an address to a socket

The `bind()` system call is used to assign an address to an existing socket.

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

- `bind` returns 0 if successful or -1 on error.

bind()

- Calling `bind()` assigns the address specified by the `sockaddr` structure to the socket descriptor.
- You can give `bind()` a `sockaddr_in` structure:

```c
bind( mysock, 
     (struct sockaddr*) &myaddr, 
     sizeof(myaddr) );
```

bind() Example

```c
int mysock; 
struct sockaddr_in myaddr;  

mysock = socket(PF_INET,SOCK_STREAM,0);  
myaddr.sin_family = AF_INET;  
myaddr.sin_port = htons( portnum );  
myaddr.sin_addr = htonl( ipaddress);  
bind(mysock, (struct sockaddr*)&myaddr, 
     sizeof(myaddr));
```
Uses for `bind()`

- There are uses for `bind()`:
  - Server would like to bind to a well known address (port number).
  - Client can bind to a specific port.
  - Client can ask the O.S. to assign any available port number.

What Port - who cares?

- Clients typically don’t care what port they are assigned.
- When you call `bind` you can tell it to assign you any available port:
  ```c
  myaddr.port = htons(0);
  ```

What is my IP address?

- How can you find out what your IP address is so you can tell `bind()`?
- There is no realistic way for you to know the right IP address to give `bind()` - what if the computer has multiple network interfaces?
- Specify the IP address as: `INADDR_ANY`, this tells the OS to take care of things.
Other socket system calls
- General Use
  - `read()`
  - `write()`
  - `close()`
- Connection-oriented (TCP)
  - `connect()`
  - `listen()`
  - `accept()`
- Connectionless (UDP)
  - `send()`
  - `recv()`

Client Software Design
- Clients are usually simpler than server
- Locating A Server Address
  - Hardwired
  - User Input or file input (manual)
  - Special protocol/service
    - Broadcast (not scalable)
    - Multicast (may cause an overhead)
    - Server Locator/directory (e.g. 411)
      - centralized, distributed/duplicated
- Host Name/address and port number as arguments (`argv`, `argc`)

Looking up Domain Name
```c
struct hostent {  
    char *h_name; /* official name of host */  
    char **h_aliases; /* alias list */  
    int h_addrlen; /* length of address */
    char **h_addr_list; /* list of addresses */
    #define h_addr h_addr_list[0] /* for compatibility */
};
struct hostent *hostptr;
char myhost="condor.depaul.edu"  
if (hostptr = gethostbyname(myhost))  
    printf("Host: %s\n", inet_ntoa(hostptr->h_addr));
else {  
    /* error in "myhost" name */
}
```
Looking up Well-known Port by name

```c
struct servent {
    char *s_name; /* official service name */
    char **s_aliases; /* alias list */
    int s_port; /* port # */
    char *s_proto; /* protocol to use */
};
```

```c
struct servent *servptr;
if (servptr = getservbyname("echo","tcp"))
    printf("Server Port: %d\n", ntohs(servptr->s_port));
else {
    /* error in service or protocols name */
}
```

TCP Client Algorithm

1. Find remote server end-point address (IP+port)
2. socket (PF_INET, SOCK_STREAM, 0)
3. Choosing local IP addr and port number is automatic
   • IP for multi-homed hosts: problem and solution
4. connect(s, servaddr, servaddrlen)
   • test socket
   • fills in the socket structure from servaddr
   • chooses the local endpoint
   • initiates TCP connection (3 way handshaking)
5. Write() and Read() -- stream-oriented
6. Shutdown(s,direction) and close(s)

TCP Client Algorithm

```c
#define BUFLEN 120
char buf[BUFLEN];
char *bufptr, *req="this is my request";
int buflen, n;
buflen = BUFLEN;
bufptr = buf;
write(s, req, strlen(req));
/* Read the response */
while((n= read(s,bufptr, buflen)) >0 ) {
    bufptr +=n;
    buflen -=n;
    if (n == 0)
        break; // End-of-file
}
```
UDP Client Algorithm

- Same as TCP Client except in the following:
  - step (4):
    - connected and unconnected UDP
    - no handshaking or testing is performed
  - step (5):
    - Message-oriented (instead of stream-oriented)
      means single read is enough
  - step (6):
    - close() and shutdown() do not send signals to remote

Network Programming Style

- API Wrappers
  - Increase re-use
  - Improve reliability
  - Improve design quality (focus on design issues)
  - Better portability
- Examples
  - CORBA, SmartSockets, ACE, ..
  - Comer’s routines: connectTCP(mac, service), connectUDP(mac, service)

Client Examples

- DAYTIME
- TIME
- TCP ECHO
- UDP ECHO