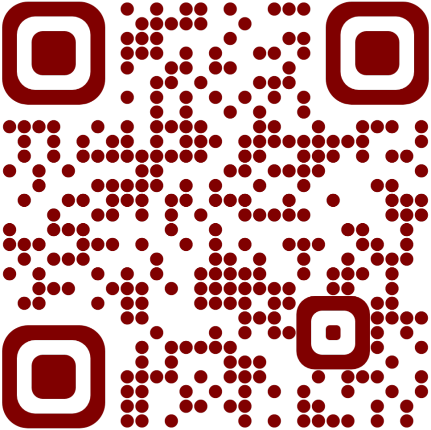
Socket Programming

Rathindra Nath Dutta

Senior Research Fellow
Advanced Computing & Microelectronics Unit
Indian Statistical Institute, Kolkata

October 21, 2022

https://ratcoinc.github.io/Networks/



WEB PAGE

What is a Socket?

- “A network **socket** is a software structure within a network node of a computer network that serves as an endpoint for sending and receiving data across the network . . . Sockets are created only during the lifetime of a process of an application running in the node” – WIKI[1]
- “A **socket** is a communications connection point (endpoint) that you can name and address in a network. Socket programming shows how to use socket APIs to establish communication links between remote and local processes” – IBM[2]
- “A **socket** is one endpoint of a two-way communication link between two programs running on the network” – ORACLE[3]

[1] https://en.wikipedia.org/wiki/Network_socket

[2] https://www.ibm.com/docs/en/i/7.5?topic=communications-socket-programming

[3] https://docs.oracle.com/javase/tutorial/networking/sockets/definition.html

How are Sockets uniquely identified?

- A *Socket* is identified to other hosts by its **Socket Address**

How are Sockets uniquely identified?

- A *Socket* is identified to other hosts by its **Socket Address**
- A *Socket Address* consists of a **Transport Protocol** (TCP, UDP, etc.), an **IP Address** (IPv4 or IPv6) and a **Port Number**

How are Sockets uniquely identified?

- A *Socket* is identified to other hosts by its **Socket Address**
- A *Socket Address* consists of a **Transport Protocol** (TCP, UDP, etc.), an **IP Address** (IPv4 or IPv6) and a **Port Number**
- A *Port Number* is a (unique) 16-bit unsigned integer assigned to one endpoint

How are Sockets uniquely identified?

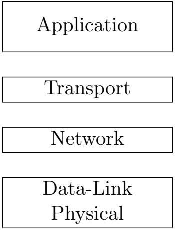
- A *Socket* is identified to other hosts by its **Socket Address**
- A *Socket Address* consists of a **Transport Protocol** (TCP, UDP, etc.), an **IP Address** (IPv4 or IPv6) and a **Port Number**
- A *Port Number* is a (unique) 16-bit unsigned integer assigned to one endpoint
- ports 0 through 1023 are well-known ports (aka system ports)
- ports 1024 through 49151 are registered ports[4]
- ports from 49152 through 65535 are dynamic or private ports; commonly known as ephemeral ports[5]

[4] IANA maintains the official list https://en.wikipedia.org/wiki/List_of_TCP_and_UDP_port_numbers

[5] ephemeral (adj.): lasting for a very short time or having a very short life cycle

Why another address?

TCP/IP Layer



Application

Transport

Network

Data-Link
Physical

Why another address?

TCP/IP Layer	Common Protocols
Application	TELNET, HTTP, DHCP, PING, FTP, . . .
Transport	TCP, UDP, . . .
Network	IP, ARP, TCMP, . . .
Data-Link Physical	Ethernet, WiFi, . . .

Why another address?

TCP/IP Layer	**Common Protocols**	**Data Packet**
Application	TELNET, HTTP, DHCP, PING, FTP, ...	Message
Transport	TCP, UDP, ...	Segment/Datagram
Network	IP, ARP, TCMP, ...	Datagram
Data-Link Physical	Ethernet, WiFi, ...	Frame Bits

Why another address?

TCP/IP Layer	Common Protocols	Data Packet	Address
Application	TELNET, HTTP, DHCP, PING, FTP, . . .	Message	Application Specific
Transport	TCP, UDP, . . .	Segment/Datagram	Port
Network	IP, ARP, TCMP, . . .	Datagram	Logical (IP)
Data-Link Physical	Ethernet, WiFi, . . .	Frame Bits	Physical(MAC)

Why another address?

TCP/IP Layer	Common Protocols	Data Packet	Address	Objective
Application	TELNET, HTTP, DHCP, PING, FTP, . . .	Message	Application Specific	
Transport	TCP, UDP, . . .	Segment/Datagram	Port	
Network	IP, ARP, TCMP, . . .	Datagram	Logical (IP)	
Data-Link Physical	Ethernet, WiFi, . . .	Frame Bits	Physical(MAC)	identification

Why another address?

TCP/IP Layer	Common Protocols	Data Packet	Address	Objective
Application	TELNET, HTTP, DHCP, PING, FTP, ...	Message	Application Specific	
Transport	TCP, UDP, ...	Segment/Datagram	Port	
Network	IP, ARP, TCMP, ...	Datagram	Logical (IP)	logical organization[1]
Data-Link Physical	Ethernet, WiFi, ...	Frame Bits	Physical(MAC)	identification

[1] better management, efficient routing

Why another address?

TCP/IP Layer	Common Protocols	Data Packet	Address	Objective
Application	TELNET, HTTP, DHCP, PING, FTP, ...	Message	Application Specific	
Transport	TCP, UDP, ...	Segment/Datagram	Port	host-to-host delivery[2]
Network	IP, ARP, TCMP, ...	Datagram	Logical (IP)	logical organization[1]
Data-Link Physical	Ethernet, WiFi, ...	Frame Bits	Physical(MAC)	identification

[2] process-to-process, a node can run multiple processes each talking via different protocol

[1] better management, efficient routing

Why another address?

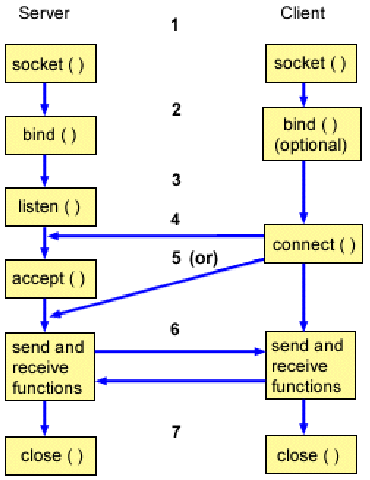
TCP/IP Layer	Common Protocols	Data Packet	Address	Objective
Application	TELNET, HTTP, DHCP, PING, FTP, ...	Message	Application Specific	end-to-end delivery[3]
Transport	TCP, UDP, ...	Segment/Datagram	Port	host-to-host delivery[2]
Network	IP, ARP, TCMP, ...	Datagram	Logical (IP)	logical organization[1]
Data-Link Physical	Ethernet, WiFi, ...	Frame Bits	Physical(MAC)	identification

[3] session control, a process(browser) may have multiple active session(tabs) to same client(google search), uses application specific URIs

[2] process-to-process, a node can run multiple processes each talking via different protocol

[1] better management, efficient routing

Socket Programming Using C



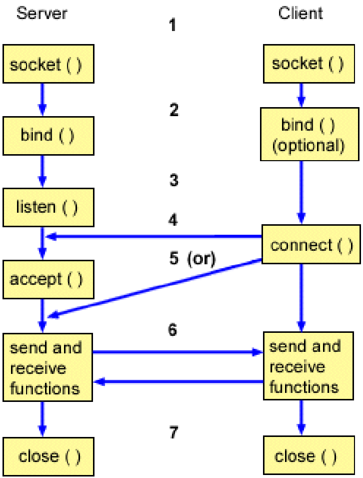
`socket()` creates and returns a socket descriptor representing an end-point for communications

Servers must bind a unique name to a socket descriptor using `bind()` to make it accessible from the network

`listen()` call shows willingness to accept client connection requests
NB: a socket cannot actively initiate any connection requests after a `listen()` call

image src: https://www.ibm.com/docs/en/i/7.5?topic=programming-how-sockets-work

Socket Programming Using C

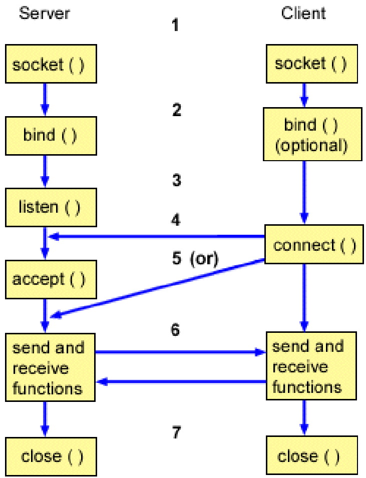


The client invokes `connect()` on a stream socket to establish a connection to the server

The server uses `accept()` to accept a client connection request
NB: The server must issue `bind()` and `listen()` calls successfully before `accept()`

image src: https://www.ibm.com/docs/en/i/7.5?topic=programming-how-sockets-work

Socket Programming Using C



When a connection is established between stream sockets (between client and server), we can use any of the data transfer methods of socket APIs such as `send()`, `recv()`, `read()`, `write()`, . . .

Finally, when a server or client wants to stop operations, it issues a `close()` call to release any system resources acquired by the socket

image src: https://www.ibm.com/docs/en/i/7.5?topic=programming-how-sockets-work

The socket() API

```
#include <sys/socket.h>
int socket(int domain, int type, int protocol);
```

Return value: On success, a file descriptor for the new socket is returned, lowest-numbered file descriptor not currently open for the process. On error, -1 is returned

Parameters:

`domain`: specifies a communication domain; selects the protocol family which will be used for communication

common domain values: `AF_UNIX` (Local communication), `AF_INET` (IPv4 Internet protocols), `AF_INET6` (IPv6 Internet protocols)

`type`: specifies the communication semantics commonly used types are: `SOCK_STREAM` (sequenced, reliable,two-way, connection-oriented byte streams, TCP), `SOCK_DGRAM` (connectionless, unreliable messages of a fixed maximum length, UDP)

`protocol`: specifies a particular protocol to be used with the socket usually a 0 is specified to denote the default protocol for the corresponding socket type

The bind() API

```
#include <sys/socket.h>
int bind(int sockfd, const struct sockaddr *addr,
         socklen_t addrlen);
```

assigns the address specified by `addr` to the socket referred to by the file descriptor `sockfd`

Return value: On success, zero is returned. On error, -1 is returned

Parameters:
`sockfd`: a socket file descriptor created with `socket()`
`addr`: a pointer to an address structure, actual structure depends on the socket address family
`addrlen`: specifies the size, in bytes, of the address structure pointed to by `addr`

Address Structure for AF_INET

```
#include <sys/socket.h>
#include <netinet/in.h>
struct sockaddr_in {
    sa_family_t    sin_family; /* address family: AF_INET */
    in_port_t      sin_port;   /* port in network byte order */
    struct in_addr sin_addr;   /* internet address */
};
/* Internet address */
struct in_addr {
    uint32_t       s_addr;     /* address in network byte orde
};
```

[1] https://man7.org/linux/man-pages/man7/ip.7.html

Binding a Socket to an Address

```
struct sockaddr_in sock_addr;
bzero((char *)&sock_addr, sizeof(sock_addr)); //clear
sockfd = socket(AF_INET, SOCK_STREAM, 0);
int portno = 54321;

sock_addr.sin_family = AF_INET;
sock_addr.sin_port = htons(portno);
sock_addr.sin_addr.s_addr = INADDR_ANY;
//sock_addr.sin_addr.s_addr = INADDR_LOOPBACK
//sock_addr.sin_addr.s_addr = inet_addr("127.0.0.1");
bind(sockfd, (struct sockaddr*)&sock_addr, sizeof(sock_addr))
```

[1] `htons()`: converts an unsigned short integer from host byte order to network byte order

[2] special addresses: `INADDR_LOOPBACK` (127.0.0.1) always refers to the *localhost* via the loopback device; `INADDR_ANY` (0.0.0.0) means any address for binding; `INADDR_BROADCAST` (255.255.255.255) means any host

[3] `inet_addr()`: converts a IPv4 host address string written in dotted decimal notation, into binary data in network byte order; require `#include <arpa/inet.h>`

The listen() API

```
#include <sys/socket.h>
int listen(int sockfd, int backlog);
```

marks the socket referred to by `sockfd` as a passive socket, i.e, a socket to be used to accept incoming connection requests using `accept()`

Return value: On success, zero is returned. On error, -1 is returned

Parameters:
`sockfd`: file descriptor that refers to a socket of type, e.g. `SOCK_STREAM`
`backlog`: defines the maximum length to which the queue of pending connections for `sockfd` may grow. If a connection request arrives when the queue is full, the client may receive an error with an indication of `ECONNREFUSED` or, if the underlying protocol supports retransmission, the request may be ignored so that a later reattempt succeeds.

[1] If the `backlog` value is greater than the value in `/proc/sys/net/core/somaxconn`, then it is silently capped to that value. Since Linux 5.4, the default in this file is 4096; in earlier kernels, the default value is 128

The accept() API

```
#include <sys/socket.h>
int accept(int sockfd, struct sockaddr *addr,
           socklen_t *addrlen);
```

used with connection-oriented socket types (e.g. SOCK_STREAM). It extracts the first connection request on the queue of pending connections for the listening socket **sockfd**, creates a new connected socket, and returns a new file descriptor for it.

Return value: On success, returns a file descriptor for the accepted socket (a nonnegative integer). On error, -1 is returned

[1] The newly created socket is not in the listening state. The original socket sockfd is unaffected

The accept() API

```
#include <sys/socket.h>
int accept(int sockfd, struct sockaddr *addr,
           socklen_t *addrlen);
```

Parameters:

`sockfd`: a socket created with `socket()`, bound to a local address with `bind()`, and is listening for connections using `listen()`

`addr`: a pointer to an address structure of the peer, actual structure depends on the socket address family

`addrlen`: a value-result argument; initialized to contain the size (in bytes) of the structure pointed to by `addr`; on return it will contain the actual size of the peer address

The close() API

```
#include <unistd.h>
int close(int fd);
```

closes a file descriptor, so that it no longer refers to any file and may be reused

Return value: returns zero on success. On error, -1 is returned

Parameters:
fd: closes the socket identified by the file descriptor

[1] https://man7.org/linux/man-pages/man2/close.2.html

The `read()` API

```
#include <unistd.h>
ssize_t read(int fd, void *buf, size_t count);
```

attempts to read up to `count` bytes from file descriptor `fd` into the buffer starting at `buf`

Return value: On success, the number of bytes read is returned (zero indicates end of file), and the file position is advanced by this number. On error, -1 is returned

Parameters:
`fd`: a file descriptor to read from
`buf`: pointer to a buffer area (array) to read into
`count`: maximum number of bytes to read

[1] https://man7.org/linux/man-pages/man2/read.2.html

[2] https://man7.org/linux/man-pages/man2/recv.2.html

The write() API

```
#include <unistd.h>
ssize_t write(int fd, void *buf, size_t count);
```

writes up to `count` bytes from buffer starting at `buf` into the file descriptor `fd`

Return value: On success, the number of bytes written is returned. On error, -1 is returned

Parameters:
`fd`: a file descriptor to write into
`buf`: pointer to a buffer area (array) to write from
`count`: maximum number of bytes to write

[1] https://man7.org/linux/man-pages/man2/write.2.html

[2] https://man7.org/linux/man-pages/man2/send.2.html

Creating an Echo Server

`server1.c`

Testing the Echo Server

compile and run the server in a terminal; and leave it be

```
gcc server1.c -o server1 && ./server1
```

run a `telnet`[1] client in another terminal

```
telnet <server ip> <server port>
telnet localhost 54321
```

write anything in telnet
type "quit" (or send `ctrl+]`) to close the connection

[1] Telnet is an application protocol used on the Internet or local area network to provide a bidirectional interactive text-oriented communication facility - WIKI