Socket Programming - Lecture Notes

Tushaar Gangarapu (tushaargvsg45@gmail.com)

March 8, 2018

1 Why and What of Sockets?

Sockets are used to build *networked applications.* Sockets are *end-point of communication.* A socket is associated with each end-host (user) of an application. Some examples of the same include *FTP*, *P2P* etc. They are identified by both *IP address* and *port number.*

1.1 Types of Sockets

Two types of sockets- Stream sockets, Datagram sockets

- **Stream sockets (TCP)**: SOCK_STREAM
  - Connection oriented (includes establishment termination)
  - Reliable, in order delivery
  - At-most-once delivery, no duplicates
  - File-like interface
  - *Eg*: ssh, http
- **Datagram sockets (UDP)**: SOCK_DGRAM
  - Connectionless (just data-transfer)
  - Best-effort delivery, possibly lower variance in delay
  - Packet-like interface
  - *Eg*: IP Telephony, streaming audio

1.2 How do Sockets Work?

IP Telephony example (between two entities, say Adam and Eve):

- Necessity of a *telephone*
- Necessity to identify a unique *phone number* associated with a telephone
- The ringing mechanism on Eve's phone must be on!
- Adam dials Eve's number
- Telephone rings, Eve answers
- Data Exchange happens
- Hang-up the phone

Extending the IP Telephony analogy:

- An end point for communication
- An address to distinguish the end-host
- The receiver waits (listens) for an active connection
- The sender initiates a connection to the receiver
- Data exchange is done once the establishment of connection is made
- Close the connection

In socket terminology:

- `socket()`: end-point
- `bind()`: unique address = IP address + port number
- `listen()`: wait for the sender
- `connect()`: sender initiates!
- `accept()`: receiver accepts the connection
- `send()`, `recv()`: data exchange
- `close()`: communication termination

1.3 Client-Server Paradigm

Client	Server
Sometimes on	Always on
Initiates request to the sender when interested	Serve services to many clients
Needs to know server's address	Needs a fixed address
Eg: Web browser	*Eg*: Web server

2 Pre-requisites for Programming with Sockets (Stream Sockets)

Client Process Activity	System Call	Server Process Activity	System Call
create a socket	`socket()`	create a socket	`socket()`
bind a socket address (optional)	`bind()`	bind a socket address	`bind()`
		listen for incoming connection requests	`listen()`
request a connection	`connect()`		
		accept connection	`accept()`
send data	`send()`		
		receive data	`recv()`
		send data	`send()`
receive data	`recv()`		
disconnect socket (optional)	`close()`	disconnect socket (optional)	`close()`

2.1 Datastructures Used in Socket Programming

Assumption: Single IP address for a host (also known as, single homed; not more than one interface ('*if*')) needed to bind IP address and port number.

```
// 'sockaddr': general and 'sockaddr_in': internet specific applications
struct sockaddr {
        unsigned short sa_family ;      // sa_family = AF_INET: networked applications
        char sa_data[14] ;              // IP address and port number to bind to
}

/* 'sockaddr_in' and 'sockaddr' are of the same size
 * we cast 'sockaddr_in' into 'sockaddr' while programming
 */
struct sockaddr_in {
        short sin_family ;              // sin_family is same as sa_family
        unsigned short sin_port ;       // Port number: 0 - 65535 (0 - 1024: reserved)
        struct in_addr sin_addr ;       // IP address
        char sin_zero[8] ;              // sin_zero is the padding
}

struct in_addr {
        unsigned long s_addr ;          // 4 bytes long integer IP (10.10.54.4)
}
/* sockaddr_in addr ;
 * addr.sin_family = AF_INET ;
 * addr.sin_port = htons(5576) ;
 * addr.sin_addr.s_addr = INADDR_ANY ;
 * bzero(&addr, 0)
 */
```

2.2 Network Byte Ordering (Big Endian)

Endianness: Storing of integers from left to right (or) right to left in the memory.
Eg: 0x0A0B0C0D (32-bit integer); How to store?
Big Endian: [0x0A][0x0B][0x0C][0x0D]
Little Endian: [0x0D][0x0C][0x0B][0x0A]

Usually, the microprocessor pre-processes to store integers as in little or big endian format on the host computer. Now, if one host stores and transfers data in big endian while other in little endian, then the discrepency arrising due to the decision of MSB and LSB, brings about the necessity to standardize the way of storing integers.

For network communication, **network byte ordering** or **big endian** format must be used, irrespective of the host byte ordering (Host Byte Order → Network Byte Order).

2.3 Necessary Functions

IP Address:

- `inet_aton()`: ASCII-to-Network (10.10.54.4 → 0000100100001...)
- `inet_ntoa()`: Network-to-ASCII (0000100100001... → 10.10.54.4)
- `inet_pton()`: Presentation-to-Network (`inet_pton(AF_INET, "127.0.0.1", &(sa.sin_addr))`)
- `inet_ntop()`: Network-to-Presentation (`inet_ntop(AF_INET, &(sa.sin_addr), str, INET_ADDRSTRLEN);`)
- `bzero(char *c, int n)`: binds 'n' zeros, starting from character 'c'

Byte Ordering:

- `htons()`, `htonl()`: Host-to-Network (short/long)
- `ntohs()`, `ntohl()`: Network-to-Host (short/long)

3 Programming with Sockets

3.1 System Calls

3.1.1. `socket()`: **end-point of communication**

- `int sockfd = socket(int domain, int type, int protocol)`
- domain = PF_INET (IPv4 Communication Protocols)
- type = SOCK_STREAM for TCP, SOCK_DGRAM for UDP
- protocol = 0 (usually) if a single protocol for communication is used, else use `getprotoent()`
- retuns integer socket descriptor (if proper) else returns -1 (if error)

AF_INET vs. PF_INET: Intially, it was thought that maybe an address family ("AF" in "AF_INET") might support several protocols that were referenced by their protocol family ("PF" in "PF_INET") which did not happen. So the correct thing to do, is to use AF_INET in `struct sockaddr_in` and PF_INET in your call to `socket()`. But practically speaking, you can use AF_INET everywhere.

3.1.2. `bind()`: **get unique address (IP address + port number)**

- `int bind(int sockfd, struct sockaddr *addr, socklen_t addrlen)`
- sockfd = socker file descriptor
- addr = (`struct sockaddr *`)&addr: casting `sockaddr_in` into `sockaddr`
- addrlen = length of `sockaddr_in`

3.1.3. `listen()`: **listen to incoming connections**

- `int listen(int sockfd, int backlog)`
- sockfd = socket file descriptor (from server `socket()`)
- backlog = maximum length of the queue of pending connections for 'sockfd' may grow

3.1.4. `accept()`: **accept a new client connection**

- `int accept(int sockfd, struct sockaddr *addr, socklen_t *addrlen)`
- sockfd = socket file descriptor (from server `socket()`)
- addr = client address (`sockaddr_in` cast into `sockaddr`) on return of `accept()`
- addrlen = length of client `sockaddr_in`

3.1.5. `connect()`: **connect to a server**

- `int connect(int sockfd, struct sockaddr *addr, socklen_t *addrlen)`
- sockfd = socket file descriptor (from client `socket()`)
- addr = remote server address (`sockaddr_in` cast into `sockaddr`)
- addrlen = length of remote server `sockaddr_in`

3.1.6. `send()`, `recv()`: **data exchange**

- `int send(int sockfd, const void *buf, size_t len, int flags)`
- `int recv(int sockfd, const void *buf, size_t len, int flags)`
- sockfd = socket file descriptor
- buf = pointer to the buffer (data)
- len = size of the data buffer
- flags = 0 (usually no flags used); MSG_CONFIRM, etc. can be used
- returns number of bytes actually sent or received

3.1.7. `close()`: **close the connection**

- `int close(int sockfd)`
- sockfd = socket file descriptor of the socket to be closed