# Sockets

Ciro S. Costa

10, Jul, 2015

## Contents



## Sockets

We can think of sockets as a way to speak to other programs using standrd unix file descriptors.

When unix programs do any sort of IO they do it by reading or writing to a file descriptor, which is an integer associated with an open file ("everything in unix is a file!").

Calling `socket()` returns us a socket descriptor, which then allows us to communicate through it using specialized `send` (for sending) and `recv` (for receiving) socket calls.

### Stream Sockets

`SOCK_STREAM` are a reliable two-way connected communication streams with guaranteed order. Web browsers use the HTTP protocol, which uses stream sockets to get pages. Uses TCP and IP under the hood.

## Datagram Sockets

`SOCK_DGRAM` also uses IP for routing but UDP for transmission. They're called 'connectionless' as we don't need to really keep a connection open.

## Structs and Types

***Most significant Bit*** bit position in a binary number having the greatest value. it can also correspond to the sign bit of a signed binary number in one's or two's complement notation, 1 meaning negative and 0 positive. MSB and LSB are indications of the ordering of the sequence of bits in the bytes sent over the vire. MSB means that the MSB will arrive fist. LSB means that the LSB will arrive first.

***Two's Complement*** corresponds to a binary signed number representation based on a simple mathematical operation. It is defined as `the complement to 2^N` (result of subtracting the number from `2^N` - taking the one's complement and adding one). This corresponds to the negative version of the original number in a way that both can coexist in a natural way.

**`memset(void *str, int c, size_t n)`** copies the character `c` (`unsigned char`) to the first `n` characters of the string (block of memory) pointed to, by the argument str.

***Endiannes*** is the ordering or sequencing of bytes of a word of digital data in computer memory storage or during transmission. There are two of them: *big-endian* and *little-endian*. The first stores the most-significant byte of a word at the smallest mem address and the least significant byte at the largest. The other is the inverse. As an example, `x86` processors store in little-endian. **In data networking Big-endian is the most common convention**, then giving the name of *network byte order*.

An example of `big-endian` (and so, `network byte order`) is the daily representation of the number `123`, with the hundreds (which is the most significant - has the biggest value) at the first position in the memory and the least significant at the last positions.

Conversions: - `htons` : Host to Network Short - `htonl` : Host to Network Long - `ntohs` : Network to Host Short - `ntohl` : Network to Host Long

This is important because we want to be portable. **Put those bytes in network byte order before putting them on the network!**.

```
socket descriptor: int
```

```
/**
 * Holds socket address information for
```

```
 * many types of sockets
 */
struct sockaddr {
  // address family (commonly AF_INET)
  unsigned short  sa_family;
  // 14 bytes of protocol address
  // dest address and port number
  char            sa_data[14];
}

/**
 * just like sockaddr but specialized
 * for internet stuff. Can also be casted
 * to 'socketaddr' to pass to socket.
 * Note that sin_zero[8] should be set
 * all to zeros via memset to pad the
 * struct correctly to sockaddr.
 *
 * sin_port and sin_addr must be in NBO as they get
 * wrapped in the packet at the IP and UDP layers,
 * respectively. sin_family is kernel stuff. It does
 * not get onto the network.
 */
struct sockaddr_in {
  short int           sin_family;
  unsigned short int  sin_port; // NBO
  struct in_addr      sin_addr; // NBO
  unsigned char       sin_zero[8];
}

struct in_addr {
  // 32-bit long, 4 bytes
  // just use the proper conversion
  // function to make it NBO
  unsigned long s_addr;
};

/**
 * Struct that we'll get when querying for a given
 * hostname in a DNS lookup.
 *
 * Retrive this with gethostbyname()
 */
struct hostent {
  // official name of the host
  char  *h_name;
```

```
  // alternate names for the host
  char  **h_aliases;
  // the type of address being returned
  int   h_addrtype;
  // the length of the address in bytes
  int   h_length
  // zero-terminated array of network addresses for
  // the host (NBO)
  char  **h_addr_list;
};

// The first address in h_addr_list.
#define h_addr h_addr_list[0]
```

## Signals

> A limited form of inter-process communication. It consists of an asynchronous notification sent to a process or a thread within the same process in order to notify of an event. **When sent, the operating system interrupts the target process's normal flow of execution to deliver the signal (for any non-atomic instruction)**. If the process has previously registered a signal handler, that routine is executed (otherwise, executes the default). Note that signals are a light-weight form of IPC as the computational and memory footprint is small.

`raise` library function sends the specified signal to the current process. `kill` sends a specified signal to a specified process if permissions allows. With the `signal()` sys call we're able to install handlers (expect for SIGKILL and SIGSTOP which is not interceptable/handled).

Because we're dealing with an asynchronous operations that might suffer from race conditions (for example, receiving a signal while in the middle of the execution of the handler routine), `sigprocmask()` comes in hand to save us by letting the programmer block and unblock the delivery of signals.