Berkeley Socket API

Byte Order

A host could use little-endian or big-endian.

The Berkely Socket API uses network byte order, which is big-endian. Hosts must convert from host byte order (little-endian or big-endian) to network byte order (big-endian).

There are conversion functions provided to convert between host and network byte order.

If the host is big-endian, then these conversion functions do nothing.

Always call the conversion functions to ensure your code is portable, even if you are currently on a big-endian machine.

htons

uint16_t htons(uint16_t *hostshort*)

Convert a 16-bit value from host byte order to network byte order.

htonl

uint32_t htonl(uint32_t *hostlong*)

Convert a 32-bit value from host byte order to network byte order.

ntohs

uint16_t ntohs(uint16_t *netshort*)

Convert a 16-bit value from network byte order to host byte order.

ntohl

uint32_t ntohl(uint32_t *netlong*)

Convert a 32-bit value from network byte order to host byte order.

Addresses

IP

```
struct sockaddr
{
     sa_family_t ss_family
     char        sa_data[]
}
```

The ss_family is at least

Family	Purpose
AF_INET	IPv4
AF_INET6	IPv6
AF_UNIX	UNIX Domain socket, local to the machine

The sa_data is a memory trick in C. It makes the struct large enough to hold one of the structs below.

IPv4

```
struct sockaddr_in
{
     sa_family_t    sin_family        // AF_INET
     in_port_t      sin_port          // network byte order
     struct in_addr sin_addr
}

struct in_addr
{
     uint32_t s_addr                  // network byte order
}
```

IPv6

```
struct sockaddr_in6
{
     sa_family_t     sin6_family           // AF_INET6
     in_port_t       sin6_port        // network byte order
     uint32_t        sin6_flowinfo
     struct in6_addr sin_addr
     uint32_t        sin6_scope_id
}

struct in6_addr
{
     uint8_t s6_addr[16]              // network byte order
}
```

Unix Domain

```
struct sockaddr_un
{
     sa_family_t  sun_family          // AF_UNIX
     char         sun_path[]
}
```

Address conversion

in_addr_t inet_addr(const char *cp)

Converts a string (e.g. “192.168.0.7”) to a inetaddr_t.

Interfaces

There is no way to get a list of network interfaces in POSIX. However, most Unix-like systems provide the getifaddrs function.

`int getifaddrs(struct ifaddrs **ifap)`

The ifap argument is a pointer to a linked list that will be filled in. Each interface is an entry in the linked list. Interfaces can appear multiple times for each family they have.

A network interface can handle IPv4 and IPv6 simultaneously.

Functions

Client	Server
socket	socket
	bind
	listen
connect	accept
read/write/recv/send	read/write/recv/send
close	close

socket

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

Create an unbound socket.
Both the client and server must create sockets.
A socket must be bound to a port.

The domains are, at least:

Domain	Purpose
AF_INET	IPv4
AF_INET6	IPv6
AF_UNIX	UNIX Domain socket, local to the machine

The types are, at least:

Type	Purpose
SOCK_STREAM	Stream socket
SOCK_DGRAM	Datagram socket
SOCK_RAW	Raw socket

The following combinations are valid:

	AF_INET	**AF_INET6**	**AF_UNIX**
SOCK_STREAM	TCP	TCP	Yes
SOCK_DGRAM	UDP	UDP	Yes
SOCK_RAW	IPv4	IPv6	

The protocol should be set to 0. Non-zero options are supported but are not needed for anything we will be doing (for now).

bind

int bind(int socket, const struct sockaddr *address, socklen_t address_len)

Assign a local address to a socket so a server can listen for client connections.

listen

int listen(int socket, int backlog)

Used by a server to listen for clients initiating a 3-way handshake.

The backlog is the number of connections that can be queued.

connect

int connect(int socket, const struct sockaddr *address, socklen_t address_len)

Typically used to establish a TCP connection from a client to a server.
Starts the 3-way handshake by sending the server a SYN packet.

accept

int accept(int socket, struct sockaddr *restrict address, socklen_t *restrict address_len)

The server calls accept, which blocks, waiting for a client to connect to it. Accepts sends a SYN/ACK packet. Once the client receives the SYN/ACK it responds with a SYN packet and the 3-way handshake is complete.

read

ssize_t read(int fildes, void *buf, size_t nbyte)

The read function reads from a file descriptor. The accept and connection functions return a file descriptor where the file entry is a socket. Reading from the file descriptor reads data from the network connection.

The client and server can read in any order, based on the protocol.

write

```
ssize_t write(int fildes, const void *buf, size_t nbyte)
```

The write function writes to a file descriptor. The accept and connection functions return a file descriptor where the file entry is a socket. Writing to the file descriptor writes data to the network connection.

The client and server can write in any order based on the protocol.

close

```
int close(int fildes)
```

Close shuts down the network connection with the FIN and ACK sequence.

The client or the server can close first, based on the protocol.

recv

```
ssize_t recv(int socket, void *buffer, size_t length, int flags)
```

The recv function is the same as read, but it only works with sockets, and you can pass the flags to provide some additional control.

send

```
ssize_t send(int socket, const void *buffer, size_t length, int flags)
```

The send function is the same as write, but it only works with sockets, and you can pass the flags to provide some additional control.