Sockets

Overview Sockets are a method of *IPC* that allow data to be exchanged between applications, either on the same host or on different hosts connected by a network. In a typical *client-server* scenario, applications communicate using sockets as follows:

- both client and server applications create sockets
- the server binds it socket to a predefined address so that client can locate it

Socket address structures Most socket functions require a *pointer to a socket address structure*. Each supported protocol suite defines its own socket address structure and the names of these structures begin with `sockaddr_` and end with a unique suffix for each protocol suite.

IPv4 socket address strcuture is defined as follows:

```
#include <netinet/in.h>
struct in_addr {
  in_addr_t s_addr; /* 32-bit IPv4 address */
                    /* network byte ordered */
};
struct sockaddr_in {
  uint8_t sin_len;        /* structlen (16) */
  sa_family_t sin_family; /* AF_INET */
  in_port_t sin_port;     /* 16-bit portno */
  struct in-addr sin_addr;
  char sin_zero[8];       /* unused */
};
```

Four socket functions pass a socket address structure *from the process to the kernel*: `bind`, `connect`, `sendto`, and `sendmsg`. Five socket functions that pass a socket address structure *from the kernel to the process* are: `accept`, `recvfrom`, `recvmsg`, `getpeername`, and `getsockname`. For these five functions, the socket length argument is used as a **value-result** argument.

Socket system call: socket() Creates a new socket.

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

Communication domain, **domain**, specifies which protocol family we will use:

- `AF_UNIX`, `AF_LOCAL`: local communication
- `AF_INET`: IPv4 Internet protocol
- `AF_INET6`: IPv6 Internet protocol
- `AF_IPX`: Novell IPX protocol
- `AF_NETLINK`: kernel user interface device
- `AF_AX25`: amateur radio AX.25 protocol, etc.

The **type** parameter indicates the *socket type* which determines communication semantics:

- `SOCK_STREAM`: reliable, two-way connection-based
- `SOCK_DGRAM`: connectionless, unreliable fixed-length datagrams
- `SOCK_SEQPACKET`: sequenced, reliable connection-based path for fixed maximum length datagrams; a consumer is required to read an entire packet with each input system call
- `SOCK_RAW`: raw network protocol access
- `SOCK_RDM`: reliable datagram layer w/o ordering
- `SOCK_PACKET`: obsolete

From Linux 2.6.27, we can BITWISE-OR the following values with above:

- `SOCK_NONBLOCK`: set `O_NONBLOCK` file status (save calling extra call to `fcntl`
- `SOCK_CLOEXEC`: see `open(2)` for its use

The following is a brief comparison between two most common types:

	stream	**datagram**
reliable delivery	yes	no
message boundary preserved	no	yes
connection-oriented	yes	no

The **protocol** specifies specifies a particular protocol for the given protocol family. Normally a single protocol exists for a given protocol family and we can set this value as 0.

Socket system call: bind() Binds a socket to an address. A server employs this call to bind its socket to a well-known address so that clients can locate the socket.

```
int bind(int sockfd, struct sockaddr *addr,
         socklen_t addrlen)
```

The `struct sockaddr` is just for casting the actual socket address structure (e.g. `sockaddr_in` or `sockaddr_in6`).

```
struct sockaddr {
  sa_familiy_t sa_family;
  char sa_data[14];
};
```

Socket system call: listen() Allows a *passive stream* socket to accept incoming connections from other sockets. The `sockfd` is a descriptor to a socket of type `SOCK_STREAM` or `SOCKSEQPACKET`.

```
int listen(int sockfd, int backlog);
```

The `backlog` is the maximum length of the queue of pending connections. If the queue is full, the client may receive an `ECONNREFUSED` error or the request will be ignored if underlying protocol supports retransmission.

Socket system call: accept() Accepts a connection from a peer application on a *listening stream socket* (`SOCK_STREAM`, `SOCK_SEQPACKET`), and optionally returns the address of the peer socket.

```
int accept(int sockfd, struct sockaddr *addr,
           socklen_t *addrlen);
```

The `sockfd` must be a descriptor for a socket which was created with `socket()`, bound to a local address with `bind`, and is listening for connections after a `listen()`.

Socket system call: connect() Establishes a connection with another socket.

```
int connect(int sockfd, struct sockaddr *addr,
            socklen_t addrlen);
```

Socket I/O Socket I/O can be performed using `read()` and `write()` system calls, or using a range of socket-specific system calls (e.g. `send()`, `recv()`, `sendto()`, and `recvfrom()`).

By default, these system calls perform **blocking I/O**. **Non-blocking I/O** is also possible by using the `fcntrl() F_SETFL` operation to enable the `O_NONBLOCK` open file status flag.

Stream sockets Operation of stream socke1ts are analogous to a **telephone call**:

1. **install telephone**: create a socket using `socket()`
2. **make a phone call**: caller connect its socket to another application's socket for a communication; two sockets are connected as follows:
   (a) server calls `bind()` to a predefined address and then calls `listen()` to notify the kernel of its willingness to accept incoming connections.
   (b) client establishes the connection by calling `connect()` specifying the address of the socket to which the connection is to be made
   (c) server that called `listen()` then accepts the connection using `accept()`
3. **talk**: once the connection has been established, data can be transmitted (e.g. `read()` and `write()`) in both directions until one of the application calls `close()`

Active vs passive sockets By default a socket created using `socket()` is active. **Active sockets** can be used in a `connect()` call to establish a connection to a passive socket. This is called as performing an `active open`.

A **passive socket** (also called a **listening socket**) is one which has been marked to allow incoming connections by calling `listen()`. Accepting an incoming connection is referred to as performing a **passive open**.

Datagram sockets Operation of datagram sockets can be explained by analygo with the **postal system**:

1. **setup mailbox**: `socket()` system call is equivalent of setting up a mailbox
2. **setup address**: server binds its socket to a well-known address so client can initiate communication
3. **send a mail**: client calls `sendto()` with the address of the socket as its destination
4. **receive a mail**: server calls `recvfrom()` which may block if no datagram has not yet arrived
5. `close()` closes the communication