Sockets 1: Intro & Addresses

1. **Introduction and Addresses**
   - **sockets**
   - **types of sockets**
   - **socket addresses**
   - **address structs**
2. Basic System Calls
3. Socket I/O Syscalls
4. IP Addresses and Hostnames
5. Endianess, Options, Servers

Sockets

Sockets are an *interprocess communication* (IPC) mechanism that allows processes on different *networked machines* to exchange data.

They can also be used between processes on the *same machine*.

Sockets are frequently used in **client-server** applications (both network and local client-server applications).

While we use the term "sockets" to refer to the IPC mechanism in general, programs will create one or more socket *objects*.

In the latter sense, "a socket" is a kernel object that serves as a *communication endpoint* for a program.

(This is similar to how we speak of "pipes" as an IPC mechanism but then also of a specific pipe created with `pipe()`.)

Sockets (contd.)

Sockets use *file descriptor handles*, just as with regular files and pipes.

Thus, some calls that work for regular files or pipes also work with sockets, including `read()` and `write()`.

However, there are other I/O calls that can provide additional functionality in conjunction with sockets.

Sockets provide **bidirectional** communication between processes (unlike pipes and FIFOs).

Sockets can be used between **unrelated processes** because they are accesible via **socket addresses** (unlike pipes, whose FDs must be inherited).

Sockets (contd.)

Communication among *networked machines* uses the **Internet Protocol Suite**, typically referred to as TCP/IP or just IP.

An *address* for a **network socket** involves both an **IP address** and a **port**.

Socket-based communication within a machine primarily uses **UNIX domain sockets**.

UNIX domain sockets have addresses that are *filesystem pathnames*, like FIFOs.

Sockets are generally superior to FIFOs even for local client-server architectures:

- bidirectional
- separate file descriptor handles for each connection

Socket Types

It is common to speak of sockets as being of a certain *type* based on various characteristics of the protocols being used or how the socket has been configured.

It is common to speak about these socket types:

- **network** vs. **UNIX/local** (i.e., *address family*)
- **stream** vs. **datagram**
- `SOCK_STREAM` vs. `SOCK_DGRAM`
  (i.e., the **communication semantics** specified by `socket()`'s *type* argument)
- **connected** vs. **unconnected/connectionless**
- **passive/listening** vs. **active**

Socket Types: Stream vs. Datagram

A **stream socket** passes data as a continuous byte stream, in FIFO order.

It provides the same basic semantics as a pipe.

A **datagram socket** passes data in individual messages, called **datagrams**.

A "stream socket" might also be called a "`SOCK_STREAM` socket."

Similarly for "datagram socket" and "`SOCK_DGRAM` socket."

Socket Types: Connected vs. Unconnected

With **connected sockets**, the addresses of the two ends of a connection are set and remain in effect.

With **unconnected sockets**, the remote end address must be specified for each communication (message).

Some protocols are necessarily connected (e.g., TCP).

Some protocols may be used in a connected or unconnected manner (e.g., UDP).

Whether a socket is connected or not determines which I/O calls can be used to exchange data via the socket.

Being connected does not imply other properties such as being reliable; a connected UDP socket still does not provide reliable transport.

Socket Types: Passive vs. Active

Sockets may also classified as **passive** or **active**.

A passive socket is one that is waiting to receive connections, as a result of a `listen()` call.

Thus, passive sockets are also called **listening sockets**.

For a socket to be capable of being a listening socket, it must be using a *stream-based, connection-oriented* protocol.

A *server* process using such a protocol will set its socket to be listening.

Socket Address (sockaddr) Structs

Many socket-related calls require a *socket address* specification.

Addresses are mostly used to specify a socket to try to connect to, but they also sometimes specify how a local socket should be identified.

Socket addresses differ in their structure/components depending on the **communication domain** (e.g., IPv4, IPv6, UNIX).

As a result, different C **struct** types will be needed for different domains.

Most socket-related calls must be able to work with multiple address types, however, but C does not support *type hierarchies*.

The solution is to specify a *generic* address type struct in calls, and then **cast** specific address arguments to that type.

Generic: sockaddr

Generic type for socket addresses:

```
struct sockaddr {
  sa_family_t sa_family;
  char        sa_data[];
}
```

This is the socket address type that is specified in socket and network calls when a socket address is required, such as in `bind()`, `connect()`, etc.

This type effectively serves as the *supertype* of all other socket address types.

IPv4: sockaddr_in

Socket address type for IPv4:

```
struct sockaddr_in {
  sa_family_t    sin_family; /* address family: AF_INET */
  uint16_t       sin_port;   /* port in network byte order */
  struct in_addr sin_addr;   /* IPv4 address (a struct) */
}
```

The actual IPv4 address is stored in:

```
struct in_addr {
  uint32_t s_addr; /* IPv4 address in network byte order */
}
```

IPv6: sockaddr_in6

Socket address type for IPv6:

```
struct sockaddr_in6 {
  u_char          sin6_family;   /* AF_INET6 */
  u_int16m_t      sin6_port;     /* Transport layer port # */
  u_int32m_t      sin6_flowinfo; /* IPv6 flow information */
  struct in6_addr sin6_addr;     /* IPv6 address (a struct) */
  uint32_t        sin6_scope_id  /* Scope ID */
}
```

The actual IPv6 address is stored in:

```
struct in6_addr {
  u_int8_t s6_addr[16]; /* IPv6 address */
}
```

IPv6: sockaddr_in6 (contd.)

Notice that the IPv6 address is ultimately specified as an array of sixteen 8-bit elements.

Together the array elements make up a single 128-bit IPv6 address, but this way the individual bytes of the address can be accessed.

UNIX: sockaddr_un

Socket address type for UNIX sockets:

```
#define UNIX_PATH_MAX    108

struct sockaddr_un {
  sa_family_t sun_family;               /* AF_UNIX */
  char        sun_path[UNIX_PATH_MAX]; /* pathname */
}
```