Sockets

Sockets are another way for two processes to communicate.

Characteristics (esp. compared to pipes):

- One side considered "server", has publishable "address".
  (Pipe ends unpublishable, only sharable by fork.)
- The other side considered "client", contacts server by published address.
- So unrelated processes (even on different computers) can still make contact.
  (Pipes: Processes must come from the same fork tree.)
- File descriptor is two-way street.
  (Pipe is one-way street.)

Socket Varieties: Axis 1: Scope

3 scopes ("domain", "address family"):

- Unix domain: Local to computer. Address is a filename.
- IPv4: Over the network. (Has special "loopback" address for local.) 32-bit address plus 16-bit "port number".
- IPv6: Like IPv4 but 128-bit address.

I will cover IPv4.

Socket Varieties: Axis 2: Abstraction Level

A lower level (datagram) and a higher level (stream):

- Datagram: By packets (chunks). One write syscall $\Rightarrow$ one chunk $\Rightarrow$ one read syscall.

  Packet loss possible. Neither side notified if it happens.

  Packets can be out of order too.

  Think “telegram”. Thus “datagram”.

- Stream: Network stack works hard to confirm, timeout, resend, restore data order. You have almost no worry, just use as byte stream.

  Network stack can re-chunk for efficiency. Receiver may not see sender’s original chunking.

I will cover stream.

Stream Socket: Client Workflow

1. Call `socket`: create socket fd.
2. Fill in address struct.
   Call `connect`: Use fd to connect to server at address.
3. Use fd to talk to server.
4. Close fd when done with client.

Stream Socket: Server Workflow

Server workflow complicated because multiple FDs to juggle:

- One FD per client (*cfd* below).
- One FD to wait for new clients (*sfd* below).

1. Call `socket`: create socket *sfd*.
2. Fill in address struct.
   Call `bind`: Bind *sfd* to address.
3. Call `listen`: Declare "*sfd* is for waiting for clients to connect". (Bad name, does not actually listen/wait.)
4. Loop over:
   4.1 Call `accept(sfd)`: Actually wait for a client to connect. Get yet another socket *cfd*.
   4.2 Use *cfd* to talk to client. Close when done with client.
5. Close *sfd* if no longer waiting for new clients.

Socket Creation

`int socket(int family, int type, int protocol);` Returns positive socket FD, or -1 if error.

`family`: `AF_UNIX`, `AF_INET` (IPv4), `AF_INET6`

`type`: `SOCK_DGRAM`, `SOCK_STREAM`, advanced low-level types.

`protocol`: 0. (Other values for advanced low-level types.)

I will focus on IPv4 stream sockets (TCP/IP).

connect

```
int connect(int fd,
            const struct sockaddr *server_addr,
            socklen_t addrlen);
```

Real address struct is never `sockaddr`:

Unix domain: `sockaddr_un`

IPv4: `sockaddr_in`

IPv6: `sockaddr_in6`

Always have to cast pointer type and provide size, e.g.,

```
connect(int myfd,
        (struct sockaddr *) &myaddr,
        sizeof(struct sockaddr_in));
```

Also, address structs may contain padding/reserved bytes, best to set 0 before filling in fields (e.g., `memset`).

If success, may simply use `read`, `write`, `close` on `fd`.
Also `recv`, `send`, `shutdown` for socket-specific features.

IP (IPv4) Addresses

IPv4 address: 32-bit, 4-byte number. Identifies computers. (Actually identifies network interfaces.)

Human-friendly dot-notation as string: Each byte in decimal, separated by dots.
Examples:
Mathlab: 142.1.96.164
uoft.me: 104.236.216.17
loopback address: 127.0.0.1

Use 'dig' program to look up IP addresses from domain names. There are also C library functions. They work by asking DNS—domain name servers.

It is possible for many domain names to map to one IP address. (Small research exercise: What are some use cases?)

It is possible for one domain name to map to many IP addresses. (Small research exercise: What are some use cases?)

IPv4 Address+Port Struct

```
struct sockaddr_in {
  sa_family_t     sin_family;   // AF_INET
  in_port_t       sin_port;     // port
  struct in_addr sin_addr;     // IPv4 address
};

struct in_addr {
  uint32_t        s_addr;
};
```

Port and IPv4 address need to be in “network byte order” (next slide).

Two special addresses:

`htonl(INADDR_LOOPBACK)`: loopback, 127.0.0.1

`INADDR_ANY`: 0.0.0.0, request binding to all network interfaces

Big/Little Endian, Network Byte Order

16-bit number 772 = hex 0304. 2 bytes. Which byte order?

- Big endian, network byte order: 03, 04.
- Little endian (e.g., Intel CPUs): 04, 03.

Similarly for 32-bit numbers.

For portability:

Library functions to convert from machine (host) order to network order: ‘htonl’ (32-bit), ‘htons’ (16-bit).

The other direction: ‘ntohl’, ‘ntohs’.

(I don’t know why the system calls don’t auto-convert for us.)

Between dot-notation string and 32-bit network byte order: ‘inet_pton’, ‘inet_ntop’. (“p” = presentation, dot-notation)

bind

```
int bind(int fd,
         const struct sockaddr *addr,
         socklen_t addrlen);
```

Real address struct is never `sockaddr`:
Unix domain: `sockaddr_un`
IPv4: `sockaddr_in`
IPv6: `sockaddr_in6`
Always have to cast pointer type and provide size, e.g.,

```
bind(int myfd,
     (struct sockaddr *) &myaddr,
     sizeof(struct sockaddr_in));
```

Also, address structs may contain padding/reserved bytes, best to set 0 before filling in fields (e.g., `memset`).

listen, accept

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

`backlog` specifies max queue length in network stack.
Queue grows when clients call `connect` but you don't call `accept`.

```
int accept(int fd,
           struct sockaddr *client_addr,
           socklen_t *addrlen);
```

If success, returns new socket *cfd* for talking to client.
`client_addr` receives client address.
(Again, never really `sockaddr`, depends on address familiy.)

May simply use `read`, `write`, `close` on *cfd*.

Also `recv`, `send`, `shutdown` for socket-specific features.

Stream Socket: No Packet Boundary

"No packet boundary" means: Suppose sender goes:

```
write(fd, n1, chunk1);
write(fd, n2, chunk2);
```

and receiver tries ($n \geq n1 + n2$):

```
r = read(fd, myspace, n);
```

then all splitting and merging are possible:

- only first part of chunk1 ($1 \leq r < n1$)
- only all of chunk1 ($r = n1$)
- chunk1 + first part of chunk2
- all of chunk1 + chunk2

If local, usually 2nd case. Don't let that fool you, all 4 cases when non-local. Check r, call `read` again to get more.

Broken Pipe

When you write to pipe/socket but the other end has closed: “broken pipe”. Your process gets `SIGPIPE`.

Default action: Process killed. Very undesirable for socket programs.

To override: Set action to `SIG_IGN` (ignore). Then process not killed, `write` returns -1, `errno` is `EPIPE`, you can check and react.