Sockets 1: Intro & Addresses

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 */
}
```

Sockets 2: Basic Syscalls

Basic Network Programming Calls

There are a relatively large number of calls related to sockets and network programming.

The most important are:

- **socket** – create a socket
- **bind** – associate (bind) an *address* (name) to a socket
- **listen** – allow connection requests on a socket
- **accept** – accept a connection on a (listening) socket
- **connect** – initiate a connection via a socket
- **write/send/sendto/sendmsg** – send data through the socket
- **read/recv/recvfrom/recvmsg** – retrieve data from the socket

Client-Server System Calls

The standard calls for a stream-based *server* are:

1. `socket()`
2. `bind()`
3. `listen()`
4. `accept()`
5. `read()`'s/`write()`'s

The standard calls for a stream-based *client* are:

1. `socket()`
2. `connect()`
3. `write()`'s/`read()`'s

Client-Server System Calls (contd.)

The standard calls for a datagram-based *server* are:

1. `socket()`
2. `bind()`
3. `recvfrom()`'s/`sendto()`'s

The standard calls for a datagram-based *client* are:

1. `socket()`
2. `sendto()`'s/`recvfrom()`'s

socket

The `socket` call creates a socket:
`int socket(int domain, int type, int protocol)`

- `domain` specifies the *communication domain*, which determines the **address family** (address format) and the types of sockets/connections that are available
- `type` defines the *communication semantics*
- `protocol` is usually 0 (zero), meaning use the default protocol for the specified socket type
- returns a *file descriptor* **handle** for the new socket

socket (contd.)

`domain` will be one of:

- `AF_INET` – IPv4;
- `AF_INET6` – IPv6
- `AF_UNIX/AF_LOCAL` – UNIX sockets
- `AF_PACKET` – raw packets

`type` will be one of

- `SOCK_STREAM` – stream (e.g., TCP)
- `SOCK_DGRAM` – datagram (e.g., UDP)
- `SOCK_SEQPACKET` – stream-message (e.g., SCTP)
- `SOCK_RDM` – reliable datagram (e.g., DCCP)
- `SOCK_RAW` – raw packets

socket (contd.)

Standard call to setup an IPv4 TCP socket:

```
  int sockfd = socket(AF_INET, SOCK_STREAM, 0));
```

bind

`bind` associate an address with a (open) socket:
`int bind(int sockfd, const struct sockaddr *addr, socklen_t addrlen)`

- `sockfd` is the file descriptor for the socket
- `addr` is the address to associate
- `addrlen` is the length of the particular address struct
- returns 0 on success else -1

`bind()` is always used when initializing a server, but may also be used with clients.

If `bind()` is not called, the kernel assigns an address (e.g., sets the port number).

bind (contd.)

Standard calls to setup IPv4 TCP socket address for a *server*:

```
struct sockaddr_in servaddr;

memset(&servaddr, 0, sizeof(servaddr));
servaddr.sin_family = AF_INET;
servaddr.sin_port = htons(1234);
servaddr.sin_addr.s_addr = htonl(INADDR_ANY);

bind(sockfd, (struct sockaddr *) &servaddr, sizeof(servaddr));
```

One could set up the address directly in the struct declaration:

```
struct sockaddr_in servaddr =
    {AF_INET, htons(1234), htonl(INADDR_ANY)};
```

listen

`listen` tells the kernel to accept connections to a socket:
`int listen(int sockfd, int backlog)`

- `sockfd` is the file descriptor for the socket
- the socket type must be `SOCK_STREAM` or `SOCK_SEQPACKET`
- `backlog` is a suggestion about the maximum size for the **queue** of *completed by not yet accepted connections*
- returns 0 on success else -1

One of the standard calls to set up a *server* for a *connection-oriented protocol*.

The standard call is:

```
listen(sfd, 10);
```

accept

`accept` completes a connection with a *listening* socket:
`int accept(int sockfd, struct sockaddr *addr, socklen_t *addrlen)`

- `sockfd` is the file descriptor for the listening socket
- `addr` is the address of the connecting *client*
- `addrlen` is the length of the particular address struct
- returns a file descriptor for a socket for the connection, which will be different from the listening socket FD `sockfd`
- if client address is not required, `addr` and `addrlen` should be set to `NULL`

accept (contd.)

The standard call when not interested in the caller's address is:

```
accept(sockfd, (struct sockaddr *) NULL, NULL);
```

If one is interested in getting information about the caller:

```
struct sockaddr_in clientaddr;
socklen_t clientlen;
accept(sockfd, (struct sockaddr *) &clientaddr, &clientlen);
```

connect

`connect()` is used in a *client* program, to establish a connection to the server:

```
int connect(int sockfd, const struct sockaddr *serv_addr, socklen_t addrlen)
```

- `sockfd` is the file descriptor for the socket
- `serv_addr` is the address of the target server socket
- `addrlen` is the length of the particular address struct
- returns 0 on success else -1

`connect()` *must* be used with TCP connections, but can also be used to create a "connected UDP socket." (though this is less common).

connect (contd.)

The standard setup and call for an IPv4 TCP socket:
(assuming a socket has already been created)

```
struct sockaddr_in servaddr;

//Set up server address struct:
memset(&servaddr, 0, sizeof(servaddr));
servaddr.sin_family = AF_INET;
servaddr.sin_port = htons(1234);
inet_aton("131.230.133.20",&servaddr.sin_addr);

connect(sockfd, (struct sockaddr *)&servaddr, sizeof(servaddr))
```

Example: Server (receives a message)

```
char msg[201];
int listen_fd, connect_fd, nread;
struct sockaddr_in servaddr;

//Create main server socket:
listen_fd = socket(AF_INET, SOCK_STREAM, 0);

//Set up server address struct and give socket address:
servaddr.sin_family = AF_INET;
servaddr.sin_port = htons(1234);
servaddr.sin_addr.s_addr = htonl(INADDR_ANY);
bind(listen_fd, (struct sockaddr *) &servaddr, sizeof(servaddr));

//Start socket listening:
listen(listen_fd, 10);

//Main server loop:
while(1) {
  //Accept a new connection and get socket to use for client:
  connect_fd = accept(listen_fd, (struct sockaddr *) NULL, NULL);

  //Read client message and print it:
  nread = read(connect_fd,msg,200); msg[nread] = '\0';
  printf("Client message: %s\n",msg); }

  //Close connection to client:
  close(connect_fd); }
```

Example: Client (sends a message)

```
//Get server's IP address and message from arguments:
char *server_ip = argv[1];
char *msg = argv[2];

int sock_fd;
struct sockaddr_in servaddr;

//Create socket:
sock_fd = socket(AF_INET, SOCK_STREAM, 0);

//Set up server address struct:
servaddr.sin_family = AF_INET;
servaddr.sin_port = htons(1234);
inet_aton(server_ip,&servaddr.sin_addr);

//Connect to server:
connect(sock_fd, (struct sockaddr *)&servaddr, sizeof(servaddr));

//Send message to server:
write(sock_fd,msg,strlen(msg));

//Close connection:
close(sock_fd);
```

shutdown()

`shutdown()` is another syscall that can sometimes be useful for controlling socket connections:

```
int shutdown(int sockfd, int how)
```

`shutdown()` "causes all or part of a full-duplex connection on the socket associated with sockfd to be shut down."

- `sockfd` is the file descriptor for the socket
- `how` denotes the action to take:
  - `SHUT_RD` – further receptions disallowed
  - `SHUT_WR` – further transmissions disallowed
  - `SHUT_RDWR` – receptions and transmissions disallowed

Sockets 3: I/O Syscalls

write, send, sendto, sendmsg

`write()` can be used with *connected sockets*.

There are other calls that provide additional functionality or must be used with *unconnected sockets*:

- `send` – similar to `write()`
- `sendto` – includes target address, for unconnected sockets
- `sendmsg` – multiple buffers, etc.

TCP (`SOCK_STREAM`) sockets are necessarily connected.

UDP (`SOCK_DGREAM`) sockets are unconnected by default, but can become connected by using `connect()`.

send

Like `write()`, `send()` must be used with *connected sockets*.

Identical in function to `write()` with the addition of the `flags` (options) parameter:

```
ssize_t send(int s, const void *buf, size_t len, int flags)
```

Among the values for use in `flags` are:

- `MSG_DONTWAIT` – make operation **non-blocking**
- `MSG_OOB` – send **out-of-band** data
- `MSG_EOR` – terminates a record, for sockets of type `SOCK_SEQPACKET`
- `MSG_NOSIGNAL` – do *not* generate `SIGPIPE` signal on errors

sendto

`sendto()` is the primary call used with *unconnected sockets* (e.g., UDP).

It includes the address to send the message to:

```
ssize_t sendto(int s, const void *buf, size_t len, int flags,
               const struct sockaddr *to, socklen_t tolen)
```

read, recv, recvfrom, recvmsg

`read()` can be used with *connected* sockets.

There are other calls that provide additional functionality or must be used with *unconnected sockets*:

- `recv` – similar to `read()`
- `recvto` – includes source address, for unconnected sockets
- `recvmsg` – multiple buffers, etc.

recv

Like `read()`, `recv()` must be used with *connected sockets*.

Differs from `read()` only in the addition of the `flags` parameter:

```
ssize_t recv(int s, void *buf, size_t len, int flags)
```

Among the values for `flags` are:

- `MSG_DONTWAIT` – make operation **non-blocking**
- `MSG_OOB` – send **out-of-band** data
- `MSG_PEEK` – do not remove received data from kernel buffer
- `MSG_WAITALL` – block until *full* request can be satisfied

recvfrom

`recvfrom()` is the primary call for use with *unconnected sockets* (e.g., UDP).

It returns the address that sent the message:

```
ssize_t recvfrom(int s, void *buf, size_t len, int flags,
                 struct sockaddr *from, socklen_t *fromlen)
```

write/send and Closed Connections

The behavior of `write()/send()` when the connection has been closed, requires some additional explanation.

By default, a `write()/send()` to a closed (stream) connection causes a `SIGPIPE` signal to be generated and sent to the process.

The *default disposition* for `SIGPIPE` is *termination*.

This means that attempting to a `write()/send()` to a closed connection will cause the calling process to immediately terminate.

(Exactly the same behavior results with a `write()` to a *pipe* when all the pipe's read ends have been closed.)

write/send and Closed Connections (contd.)

There are two ways to change the behavior of `write()`/`send()` so that they return the error `EPIPE` without being terminated by a `SIGPIPE` signal:

1. change the disposition of `SIGPIPE` to *ignored*:
   e.g., `signal(SIGPIPE, SIG_IGN)`
2. use the `MSG_NOSIGNAL` flag with each `send()` call:
   e.g., `send(sockfd, buff, bytes, MSG_NOSIGNAL)`

The difference is that `MSG_NOSIGNAL` applies *per-call*, while ignoring `SIGPIPE` sets a process attribute (so affects all `write()`/`send()` calls in *all threads* in the process).

Sockets 4: IP Addresses & Hostnames

1. Introduction and Addresses
2. Basic System Calls
3. Socket I/O Syscalls
4. **IP Addresses and Hostnames**
   - **address format conversion**
   - **hostnames**
   - **address translation**
5. Endianess, Options, Servers

IP Address Format Conversion

Often an "IP address" will be available as a *string* in dotted quad (IPv4) or colon (IPv6) format.

This is typical if an address must be passed to a program as a command-line argument.

True IP addresses are *binary numbers* (e.g., 32-bit unsigned int for IPv4).

(Note that true IP addresses must also be stored in **network byte order** rather than **host byte order**.)

Thus, it is common to have to convert "IP address" *strings* into true IP addresses.

IP Address Format Conversion (contd.)

There are a number of conversion calls:

- **inet_aton** – convert dotted quad string to IPv4 address:
  ```
  int inet_aton(const char *cp, struct in_addr *inp)
  ```
- **inet_ntoa** – convert IPv4 address to dotted quad string:
  ```
  char *inet_ntoa(struct in_addr in)
  ```
- **inet_pton** – converts a string address into a network address:
  ```
  int inet_pton(int af, const char *src, void *dst)
  ```
- **inet_ntop** – converts network address into a string:
  ```
  const char *inet_ntop(int af, const void *src,
                        char *dst, socklen_t cnt)
  ```

IP Address Format Conversion (contd.)

An example call to `inet_aton` is:

```
struct sockaddr_in servaddr;
inet_aton("131.230.133.20", &servaddr.sin_addr);
```

An example call to `inet_pton` with an IPv4 address is:

```
struct sockaddr_in servaddr;
inet_pton(AF_INET, "131.230.133.20", &servaddr.sin_addr);
```

An example call to `inet_pton` with an IPv6 address is:

```
struct sockaddr_in6 servaddr;
inet_pton(AF_INET6, "::ffff:131.230.133.20",
                    &servaddr.sin6_addr);
```

Address Translation/DNS

Because humans prefer to use **hostnames** rather than IP addresses, network-related programs will often have to translate hostnames to IP addresses.

There are **address translation** functions that can translate from hostnames to IP addresses and the reverse.

These calls do not directly/specifically invoke the **DNS** system, but will do so as part of the standard lookup procedure specified for your OS (e.g., `hosts` file, then NIS, then DNS).

Unfortunately, while the older calls are relatively straightforward, the latest, IPv6-supporting calls are quite complex.

Address Translation/DNS (contd.)

`gethostbyname` is the now older, now deprecated call, though modern versions should work for both IPv4 and IPv6:
`struct hostent *gethostbyname(const char *name)`

The `hostent` struct contains the returned address(es):

```
struct hostent {
  char  *h_name;              /* Official name of host */
  char **h_aliases;           /* Array of aliases, NULL terminated */
  int    h_addrtype;          /* Host address type */
  int    h_length;            /* Length of each IP address */
  char **h_addr_list;         /* Array of IP addresses, NULL terminated */
}

#define h_addr h_addr_list[0] /* First address */
```

Returns `NULL` on error and sets `h_errno` global (use `herror()` or `hstrerror()` for error string).

Address Translation/DNS (contd.)

Here is an example showing how a client could lookup a server's IP address and use it with `connect()`:

```
  int sockfd = socket(AF_INET, SOCK_STREAM, 0));

  //Lookup server's IP address info using gethostbyname:
  struct hostent *hostinfo;
  hostinfo = gethostbyname("pc00.cs.siu.edu");

  //Set up server address:
  struct sockaddr_in servaddr;
  memset(&servaddr, 0, sizeof(servaddr));
  servaddr.sin_family = AF_INET;
  servaddr.sin_port = htons(1234);
  servaddr.sin_addr = *(struct in_addr *)hostinfo->h_addr;  //cast is required

  connect(sockfd, (struct sockaddr *)&servaddr, sizeof(servaddr));
```

Address Translation/DNS (contd.)

`getaddrinfo` is the newer call:

```
int getaddrinfo(const char *host, const char *service,
                const struct addrinfo *hints,
                struct addrinfo **results)
```

- `host` is typically the target hostname (as a C string), else `NULL` (must have `host` or `service` non-NULL)
- `service` is the target *service name* or port (as a C string), else `NULL`
- `hints` is an `addrinfo` struct with certain fields set to limit the addresses returned
- `results` is a *linked list* of `addrinfo` structs containing the result addresses
- returns 0 on success else an error code (use `gai_strerror()` to get error string)

Address Translation/DNS (contd.)

The `addrinfo` struct is defined as:

```
struct addrinfo {
  int              ai_flags;     /* Flags (for hints)*/
  int              ai_family;    /* Address family */
  int              ai_socktype;  /* socket() type*/
  int              ai_protocol;  /* socket() protocol */
  size_t           ai_addrlen;   /* Size of ai_addr struct */
  struct sockaddr *ai_addr;      /* sockaddr struct pointer*/
  char            *ai_canonname; /* Canonical hostname */
  struct addrinfo *ai_next;      /* Next struct addrinfo in list */
  }
```

Address Translation/DNS (contd.)

By default, `service` is to be the service name for the server, as can be found in `/etc/services`. The port number (as a string) can be supplied as well, but then the `ai_flags` field of `hints` must include the value: `AI_NUMERICSERV`.

The `hints` struct must have at least the following two fields set:

- `ai_family` – `AF_INET` or `AF_INET6`
- `ai_socktype` – `SOCK_STREAM` (TCP) or `SOCK_DGRAM` (UDP)

`results` is a linked list of addresses, because machines may have multiple IP addresses and there can be multiple results given the particular `hints`.

The memory for the list is dynamically allocated, and must be reclaimed when no longer needed, using `freeaddrinfo()`.

Address Translation/DNS (contd.)

Example showing how a client can get an IP address(es):
(uses only first address obtained)

```
  struct addrinfo hints;
  struct addrinfo *servinfo;

  //Setup hints and lookup server's IP address:
  memset(&hints, 0, sizeof(hints));
  hints.ai_family = AF_UNSPEC;     //Allow IPv4 or IPv6 address
  hints.ai_socktype = SOCK_STREAM; //TCP connection
  hints.ai_flags = AI_NUMERICSERV; //To allow numeric port rather than service name
  getaddrinfo("www.cs.siu.edu", "1234", &hints, &servinfo);

  //Create socket using results:
  int sockfd = socket(servinfo->ai_family, servinfo->ai_socktype, servinfo->ai_protocol);

  //Connect to server using first address found:
  connect(sock_fd, (struct sockaddr *)servinfo->ai_addr, servinfo->ai_addrlen);

  freeaddrinfo(servinfo);  //Free the address linked list from getaddrinfo
```

Sockets 5: Endianess, Options, Servers

1. Introduction and Addresses
2. Basic System Calls
3. Socket I/O Syscalls
4. IP Addresses and Hostnames
5. **Endianess, Options, Servers**
   - **endianess**
   - **network byte order**
   - **socket options**
   - **server designs**

Endianness

When data such as integers are stored in *multiple bytes*, there are *alternative orders* in which the bytes may be stored in memory.

This is referred to as **endianness** or **byte ordering**.

Byte order is important in network computing because machines using *different architectures* will need to exchange information such as IP addresses.

The two most common byte orderings are:

- **big endian** – most significant byte first (lowest address)
- **little endian** – least significant byte first

(The terms are from *Gulliver's Travels*.)

Endianness (contd.)

Note that because bits are not individually addressable typically, it is always assumed to be low bit "first" (at low end of byte).

The x86 architecture uses a little endian scheme, while big endian is used in Sun's SPARC and IBM's PowerPC.

While each machine will use some **host byte order**, information to be exchanged must be convereted to a *common* **network byte order**.

(Network byte order is big endian!)

Functions are provided to translate integers from host to network byte order and vice versa.

These must be used with IP addresses and port numbers not created via network programming calls.

Endianness (contd.)

Conversion functions:

- **htons** – convert 2 byte host order to network order:
  `uint16_t htons(uint16_t hostshort)`
- **htonl** – convert 4 byte host order to network order:
  `uint32_t htonl(uint32_t hostlong)`
- **ntohs** – convert 2 byte network order to host order:
  `uint16_t ntohs(uint16_t netshort)`
- **ntohl** – convert 4 byte network order to host order:
  `uint32_t ntohl(uint32_t netlong)`

getsockopt, setsockopt

There are various options that can be set on sockets, and two calls allow one to retrieve and set these options:

```
int getsockopt(int s, int level, int optname,
               void *optval, socklen_t *optlen);

int setsockopt(int s, int level, int optname,
               const void *optval, socklen_t optlen);
```

For example, to allow a port to be immediately reused:
(this should be done for servers to allow immediate restart)

```
  int i=1;
  setsockopt(sockfd, SOL_SOCKET, SO_REUSEADDR, &i, sizeof(i));
```

Server Designs

Servers are generally designed to run *indefinitely*, so will contain an *infinite loop!*

Connection-oriented (e.g., TCP) servers will have a separate file descriptor for each connection, and can potentially have multiple active connections (clients) at any time.

There are two basic classes of connection-oriented server:

- **sequential/iterative** – handles a single connection until done, then handles the next, etc.
- **parallel/concurrent** – handles multiple connections all concurrently (simultaneously)

Server Designs (contd.)

There are three main approaches to building parallel/concurrent connection-oriented servers:

1. `fork()` off a *subprocess* to handle each active connection (listening server runs in parent process)
2. use `pthread_create()` to create a new *thread* to handle each active connection (listening server runs in main thread)
3. handle all clients in a single process thread, using `epoll` (or older `poll()` or `select()`) to alternate among all active connections based on whether there is pending I/O

The first two approaches are relatively simple to implement, but cannot handle very large numbers of clients due to their resource requirements.

Modern high capacity servers (e.g., **nginx**) use the third approach.

Server Designs (contd.)

Parallel servers that handle connections in separate processes/threads, must deal with the need to collect these processes/threads when clients terminate.

However, since clients terminate *asynchronously*, it is problematic to collect the processes/threads in the main listening server loop.

It is therefore common to collect subprocess children by either:

1. setting `SIGCHLD` to be *ignored*
2. setting up a handler for `SIGCHLD` that calls `waitpid()`

With client threads, it is common to set the threads as *detached* by using `pthread_detach()`.