Motivation

- Developing network applications requires an Application Program Interface (API)
- TCP/IP protocol specifications do not include an API definition
- Many APIs defined by OS platform developers
  - Sockets (BSD Unix), standardized by IEEE (POSIX)
  - TLI/XTI (AT&T System V)
  - Winsock (Microsoft)
  - MacTCP (Apple)
**Typical Functions Needed**

- Specify the endpoints of a network connection (local and remote)
- Open a connection (active open)
- Wait for an incoming connection (passive open/listen)
- Terminate a connection
- Abort a connection because of errors

---

**Example Sequence of Socket Calls**

![Socket Call Diagram]

0-4
Unix File Descriptor Table

Socket Descriptor

Socket Descriptor Data Structure

<table>
<thead>
<tr>
<th>Descriptor Table</th>
<th>Family: PF_INET</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Service: SOCK_STREAM</td>
</tr>
<tr>
<td>1</td>
<td>Local IP: 111.22.3.4</td>
</tr>
<tr>
<td>2</td>
<td>Remote IP: 123.45.6.78</td>
</tr>
<tr>
<td>3</td>
<td>Local Port: 2249</td>
</tr>
<tr>
<td>4</td>
<td>Remote Port: 3726</td>
</tr>
<tr>
<td>...</td>
<td>Netprog: Sockets API</td>
</tr>
</tbody>
</table>
Creating a Socket

```c
int socket (int family, int type, int proto);
```

- **family**: Specifies the protocol family
  - PF_INET for TCP/IPv4, PF_INET6 for TCP/IPv6
- **type**: Type of service
  - SOCK_STREAM or SOCK_DGRAM
- **proto**: protocol to use (0 = default)
- Returns pointer to socket descriptor (-1 on error)
- Allocates socket data structure, but does not define endpoint addresses

POSIX Data Types

- `int8_t` signed 8-bit integer
- `uint8_t` unsigned 8-bit integer
- `int16_t` signed 16-bit integer
- `uint16_t` unsigned 16-bit integer
- `int32_t` signed 32-bit integer
- `uint32_t` unsigned 32-bit integer

Regular C data types
- `u_char`, `u_short`, `u_int`, `u_long`
Creating a Socket

int socket (int family, int type, int proto);

- **family**: Specifies the protocol family
  - PF_INET for TCP/IPv4, PF_INET6 for TCP/IPv6
- **type**: Type of service
  - SOCK_STREAM or SOCK_DGRAM
- **proto**: protocol to use (0 = default)
- Returns pointer to socket descriptor (-1 on error)
- Allocates socket data structure, but does not define endpoint addresses

Specifying Host Address

- Need to specify an IP address and TCP port number of the local endpoint
- The `bind()` system call is used to assign an address to an existing socket.
  ```c
  int bind(int sockfd,
           (struct sockaddr_in*) &myaddr,
           sizeof(myaddr));
  ```
- `bind` returns 0 if successful or -1 on error
Specifying Host Address

- Need to specify an IP address and TCP port number of the local endpoint
- The `bind()` system call is used to assign an address to an existing socket.
  ```c
  int bind(int sockfd, (struct sockaddr_in*) &myaddr, sizeof(myaddr));
  ```
- `bind` returns 0 if successful or -1 on error

Address Structure

- POSIX data types for endpoint address:
  - `sa_family_t` address family
  - `socklen_t` size of structure
  - `in_addr_t` IPv4 address
  - `in_port_t` TCP port number
TCP/IPv4 Socket Address Structure

- POSIX data types for endpoint address:

  ```c
  struct sockaddr_in {
    uint8_t sin_len;
    sa_family_t sin_family;
    in_port_t sin_port;
    struct in_addr sin_addr;
    char sin_zero[8];
  };

  struct in_addr {
    in_addr_t s_addr;
  }
  ```

TCP/IP Socket Address Structure

```
<table>
<thead>
<tr>
<th></th>
<th>sin_len</th>
<th>AF_INET</th>
<th>sin_port</th>
<th>sin_addr</th>
<th>sin_zero</th>
</tr>
</thead>
</table>
```
Network Byte Order

- Network byte order for Internet protocols is big-endian.
- IP address and TCP port number store in `sockaddr` structure must be in network byte order.
- Host byte order may be big- or little-endian
- Use library functions to convert between network and host byte order

Byte Order Conversion

- ‘h’ : host byte order
- ‘n’ : network byte order
- ‘s’ : short (16bit)
- ‘l’ : long (32bit)

```
uint16_t htons(uint16_t);
uint16_t ntohs(uint16_t);
uint32_t htonl(uint32_t);
uint32_t ntohl(uint32_t);
```
bind() Example

int mysock, err;
struct sockaddr_in myaddr;

mysock = socket(PF_INET, SOCK_STREAM, 0);
myaddr.sin_family = AF_INET;
myaddr.sin_port = htons(portnum);
myaddr.sin_addr = htonl(ipaddress);

err = bind(mysock, (sockaddr *) &myaddr, sizeof(myaddr));
**bind() Address Parameters**

- Typically used by servers to assign listening port number and IP address
  - Port number is the (well-known) port number assigned to application
- Can also be used by clients to bind to a specific port
  - Calling bind() with a port number of 0 results in the OS assigning an available port
- How to determine IP address?
  - No general way to determine the IP address to bind to, when the host has multiple network interfaces
  - Common practice is to specify IP address as INADDR_ANY to let OS assign the IP address

---

**Configuring a passive-mode socket**

- Server needs to initialize socket to receive incoming TCP connections
  - Also called “listening” socket
- Performed by `listen()`
  ```c
  int listen( int sockfd, int backlog);
  ```
  - `sockfd` is the TCP socket (already bound to an address)
  - `backlog` is the number of incoming connections the kernel should be able to queue for the application
  - Returns 0 normally, -1 on error
- Must be called once for each new connection the server wants to accept
Accepting a connection

- **listen()** must be followed by **accept()** to obtain parameters of accepted connection

```c
int accept(int sockfd,
           struct sockaddr_in* cliaddr,
           socklen_t *addrlen);
```

- `sockfd` is the passive mode TCP socket returned by the socket() call
- `cliaddr` is a pointer to allocated space for the socket address structure
- `addrlen` is a *value-result* argument
  - must be set to the size of `cliaddr` when calling
  - on return, will be set to be the number of used bytes in `cliaddr`

Accepting a connection (continued)

- **accept()** returns new socket descriptor (-1 on error)
  - Different from original listening socket descriptor
  - Must use new descriptor to read and write data
**Initiating a TCP connection**

- Clients initiate connection using `connect()` system call
  - Also sets up an endpoint address (IP address, TCP port number) for the client socket.
  - Clients don’t need to call bind first
- Results in the OS performing TCP connection setup to remote host via 3-way handshake
  ```c
  int connect(int sockfd, const struct sockaddr_in *server, socklen_t addrlen);
  ```
  - `sockfd` is socket returned by `socket()` call
  - `server` contains the IP address and TCP port number of the server
  - Returns 0 normally, -1 on error

**Closing a connection**

- Either side can initiate closing the connection with the `close()` system call
- Must be called by both sides to terminate cleanly
- If the other end has closed the connection, and there is no buffered data, reading from a TCP socket returns 0 to indicate EOF.
Receiving data from a socket

```c
int read( int fd, char *buf, int max);
```

*max* is the maximum number of bytes the read is willing to accept (size of buffer)

- By default, `read()` will block until data is available.
  - Non-blocking option available
- Returns number of bytes read into the buffer
  - Between 1 and *max* bytes when connection is open
  - 0 indicates EOF (other side closed connection)

---

Sending data to a socket

```c
int write( int fd, char *buf, int num);
```

*num* is the number of bytes to be transmitted

- Writes are non-blocking by default
  - Might not be able to write all *num* bytes
  - Returns actual number of bytes written (between 0 and num)