Network API

- API - Application Programming Interface
  - API is a set of functionality/services delivered by a programming system.

- Network API
  - The services (often provided by the operating system) that provide the interface between application and protocol software.

Network API wish list

- Generic Programming Interface.
  - Support multiple communication protocol suites (families).
  - Address (endpoint) representation independence.
  - Provide special services for Client and Server?

- Support for message oriented and connection oriented communication.
- Work with existing I/O services (when this makes sense).
- Operating System independence

TCP/IP

- TCP/IP does not include an API definition.

- There are a variety of APIs for use with TCP/IP:
  - Sockets by Berkeley
  - XTI (X/Open Transport Interface) by AT&T
  - Winsock - Windows Sockets API by Microsoft
  - MacTCP / Open Transport by Apple
Client-Server Model

- One side of communication is client, and the other side is server
- Server waits for a client request to arrive
- Server processes the client request and sends the response back to the client
- Iterative or concurrent

Functions needed:
- Specify local and remote communication endpoints
- Initiate a connection
- Wait for incoming connection
- Send and receive data
- Terminate a connection gracefully
- Error handling

Berkeley Sockets

- A socket is an abstract representation of a communication endpoint.
- Generic:
  - support for multiple protocol families.
  - address representation independence
- Sockets (obviously) have special needs:
  - establishing a connection
  - specifying communication endpoint addresses
- Sockets work with Unix I/O services just like files, pipes & FIFOs

Elements of a Socket

- Each socket can be uniquely identified by
  - Source IP address
  - Source port number
  - Destination IP address
  - Destination port number
  - An end-to-end protocol (TCP or UDP)

Types of Sockets

- Two different types of sockets
  - Stream sockets
  - Datagram sockets

Stream Sockets

- Also known as connection-oriented socket
- Use TCP
- Provide reliable, connected networking service
- Error free; no out-of-order packets
- Applications: telnet, ssh, http
Datagram Sockets
- Also known as connectionless socket
- Use UDP
- Provide unreliable, best-effort networking service
- Packets may be lost; may arrive out of order
- Applications: streaming audio/video

Unix Descriptor Table

Socket Descriptor Data Structure

Client-Server Model
- **Server**
  - Create a socket with the `socket()` system call
  - Bind the socket to an address using the `bind()` system call. For a server socket on the Internet, an address consists of a port number on the host machine.
  - Listen for connections with the `listen()` system call
  - Accept a connection with the `accept()` system call. This call typically blocks until a client connects with the server.
  - Send and receive data

Client-Server Model
- **Client**
  - Create a socket with the `socket()` system call
  - Connect the socket to the address of the server using the `connect()` system call
  - Send and receive data. There are a number of ways to do this, but the simplest is to use the `read()` and `write()` system calls.

Creating a Socket
```c
int socket(int family, int type, int protocol);
```
- **family** specifies the protocol family
  - `AF_INET`: IPv4 protocols
  - `AF_INET6`: IPv6 protocols
  - `AF_ROUTE`: Routing sockets
- **type** specifies the type of service
  - `SOCK_STREAM`
  - `SOCK_DGRAM`
  - `SOCK_RAW`
- **protocol** specifies the specific protocol (usually 0, which means the default)
  - `IPPROTO_TCP`: TCP transport protocol
  - `IPPROTO_UDP`: UDP transport protocol
socket()  
- The `socket()` system call returns a socket descriptor (small integer) or -1 on error.
- `socket()` allocates resources needed for a communication endpoint - but it does not deal with endpoint addressing.

Specifying an Endpoint Address  
- Remember that the sockets API is *generic*
- There must be a *generic* way to specify endpoint addresses.
- TCP/IP requires an IP address and a port number for each endpoint address.

bind()  
- calling `bind()` assigns the address specified by the `sockaddr` structure to the socket descriptor.

```c
bind( mysock,  
     (struct sockaddr*) &myaddr,  
     sizeof(myaddr) );
```

Necessary Background Information:  
**POSIX data types**
- `int8_t` signed 8bit int
- `uint8_t` unsigned 8bit int
- `int16_t` signed 16bit int
- `uint16_t` unsigned 16bit int
- `int32_t` signed 32bit int
- `uint32_t` unsigned 32bit int

**Generic socket addresses**
```c
struct sockaddr {  
    uint8_t     sa_len;  
    sa_family_t sa_family;  
    in_addr_t   sa_data[14];  
}  
```

- `sa_family` specifies the address type.
- `sa_data` specifies the address value.
sockaddr

- An address that will allow me to use sockets to communicate with you.
- address type AF_CSCE515
- address values:
  - Dean 1 Sayan 6
  - Devon 2 Yuliya 7
  - Samuel 3 Razvan 8
  - Shamik 4 Mythri 9
  - Henry 5 Femitolu 10

AF_CSCE515

- Initializing a sockaddr structure to point to Henry:

  ```
  struct sockaddr henry;
  henry.sa_family = AF_CSCE515;
  henry.sa_data[0] = 5;
  ```

AF_INET

- For AF_CSCE515 we only needed 1 byte to specify the address.
- For AF_INET we need:
  - 16 bit port number
  - 32 bit IP address

struct sockaddr_in (IPv4)

```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];
};
```

A special kind of sockaddr structure

struct in_addr

```c
struct in_addr {
    in_addr_t s_addr;
};
```

in_addr just provides a name for the ‘C’ type associated with IP addresses.

Byte Ordering

- Different computer architectures use different byte ordering to represent multibyte values.
- 16 bit integer:

  ![Byte Ordering Diagram](image-url)
**Byte Ordering**

- **Little-Endian**
  - High Byte
  - Low Byte
  - Addr A
  - Addr A+1
  - IBM 80x86
  - DEC VAX
  - DEC PDP-11

- **Big-Endian**
  - High Byte
  - Low Byte
  - Addr A
  - Addr A+1
  - IBM 370
  - Motorola 68000
  - Sun

**Byte Order and Networking**

- Suppose a Big Endian machine sends a 16 bit integer with the value 2:
  
<table>
<thead>
<tr>
<th>Little Endian</th>
<th>Big Endian</th>
</tr>
</thead>
<tbody>
<tr>
<td>Addr A</td>
<td>Addr A+1</td>
</tr>
<tr>
<td>Addr A+1</td>
<td>Addr A</td>
</tr>
<tr>
<td>0000000000000010</td>
<td>000000100000000</td>
</tr>
</tbody>
</table>

- A Little Endian machine will think it got the number 512:

<table>
<thead>
<tr>
<th>Little Endian</th>
<th>Big Endian</th>
</tr>
</thead>
<tbody>
<tr>
<td>Addr A</td>
<td>Addr A+1</td>
</tr>
<tr>
<td>Addr A+1</td>
<td>Addr A</td>
</tr>
<tr>
<td>0000001000000000</td>
<td>0000000000000010</td>
</tr>
</tbody>
</table>

**Network Byte Order**

- Conversion of application-level data is left up to the presentation layer.
- But hold on !!! How do lower level layers communicate if they all represent values differently ? (data length fields in headers)
- A fixed byte order is used (called *network byte order*) for all control data.

**Network Byte Order Functions**

- `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);
```

**TCP/IP Addresses**

- We don’t need to deal with `sockaddr` structures since we will only deal with a real protocol family.

- We can use `sockaddr_in` structures.

**Common Mistake:**

Ignoring Network Byte Order!

BUT: The C functions that make up the sockets API expect structures of type `sockaddr`.

```
int bind(int sockfd, struct sockaddr *my_addr, int addrlen);
int connect(int sockfd, struct sockaddr *serv_addr, int addrlen);
```
Assigning an address to a socket

- The `bind()` system call is used to assign an address to an existing socket.

```c
int bind( int sockfd, const struct sockaddr *myaddr, int addrlen);
```

- `bind` returns 0 if successful or -1 on error.

**Example**

```c
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));
```

- Why no htons/htons?
- Uses for `bind()`
  - There are a number of uses for `bind()`:
    - Server would like to bind to a well known address (port number).
    - Client can bind to a specific port.
    - Client can ask the OS to assign any available port number.

**Port schmo - who cares?**

- Clients typically don’t care what port they are assigned.

- When you call `bind` you can tell it to assign you any available port:

  ```c
  myaddr.port = htons(0);
  ```

- 1-1024: reserved port (assigned by privileged processes)

**What is my IP address?**

- How can you find out what your IP address is so you can tell `bind()`?

- There is no realistic way for you to know the right IP address to give `bind()` - what if the computer has multiple network interfaces?

- Specify the IP address as: `INADDR_ANY`, this tells the OS to take care of things.

  ```c
  myaddr.sin_addr.s_addr = htonl(INADDR_ANY);
  ```
IPv4 Address Conversion

```c
int inet_aton( char *, struct in_addr *);
```

Convert ASCII dotted-decimal IP address to network byte order 32 bit value. Returns 1 on success, 0 on failure.

```c
char *inet_ntoa(struct in_addr);
```

Convert network byte ordered value to ASCII dotted-decimal (a string).

Client-Server Communication (TCP)

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>socket()</code></td>
<td>Create a connection.</td>
</tr>
<tr>
<td><code>bind()</code></td>
<td>Bind a socket's IP and port.</td>
</tr>
<tr>
<td><code>listen()</code></td>
<td>Listen for incoming connection requests.</td>
</tr>
<tr>
<td><code>accept()</code></td>
<td>Accept a connection request.</td>
</tr>
<tr>
<td><code>read()</code></td>
<td>Read data from a socket.</td>
</tr>
<tr>
<td><code>write()</code></td>
<td>Write data to a socket.</td>
</tr>
<tr>
<td><code>connect()</code></td>
<td>Connect to a server.</td>
</tr>
<tr>
<td><code>close()</code></td>
<td>Close a connection.</td>
</tr>
</tbody>
</table>

TCP Client

TCP Server

Other socket system calls

- **General Use**
  - `read()`
  - `write()`
  - `close()`

- **Connection-oriented (TCP)**
  - `connect()`
  - `listen()`
  - `accept()`

- **Connectionless (UDP)**
  - `send()`
  - `recv()`

Assignment & Next time

- **Reading:**
  - UNP1, 3**
  - Socket Programming FAQ

- **Next Lecture:**
  - TCP Details