Network Programming
Many network applications are based on the client-server model:

- A server process and one or more client processes
- Server manages some resource
- Server provides service by manipulating resource for clients
- Server activated by request from client

Note: clients and servers are processes running on hosts (can be the same or different hosts)
A network is a group of connected systems that are able to communicate in order to exchange data.

There are many kinds of networks, some examples:

- **LAN** (Local Area Network) spans a building or campus
  - *Ethernet* is most prominent example
- **WAN** (Wide Area Network) spans country or world
  - Typically high-speed *point-to-point telecom lines*

An internetwork (internet) is an interconnected set of networks

- The Global IP Internet (uppercase ‘I’) is the most famous example of an internet (lowercase ‘i’)}
Network devices that originate, route and terminate data are called 'nodes'.
- Nodes or 'hosts' can be personal computers, phones, servers as well as special networking hardware.

Two such devices can be said to be 'networked' together when one device is able to exchange information with the other device.

A 'router' is an networking device that forwards data between networks in an internet.

A 'link' is the means of connecting one location to another for the purpose of transmitting and receiving data across networks.
- phone lines, fiberoptic cables, bluetooth, ethernet, wireless, etc...
Logical Structure of an internet

- Ad hoc interconnection of networks
  - No particular topology
  - Vastly different router & link capacities

- Send packets from source to destination by hopping through networks
  - Router forms bridge from one network to another
  - Different packets may take different routes
The Notion of an internet Protocol

- How is it possible to send bits across incompatible LANs and WANs?

- **Solution:** protocol software running on each host and router
  - A protocol is a set of rules that governs how hosts and routers should cooperate when they transfer data from network to network.
  - The rules define the syntax, semantics and synchronization of communication and possible error recovery methods.
  - Protocols may be implemented by hardware, software, or a combination of both.
What Does an internet Protocol Do?

- **Provides a naming scheme**
  - An internet protocol defines a uniform format for *host addresses*
  - Each host (and router) is assigned at least one of these internet addresses that uniquely identifies it

- **Provides a delivery mechanism**
  - An internet protocol defines a standard transfer unit (*packet*)
  - Packet consists of *header* and *payload*
    - *Header*: contains info such as packet size, source and destination addresses
    - *Payload*: contains data bits sent from source host

- **Lots of other things…**
  - Example, what order do the bytes go on the wire? Little-endian or Big-endian?
Global IP Internet (upper case)

- Most famous example of an internet

- Based on the TCP/IP protocol family
  - **IP (Internet Protocol)**:
    - Provides basic *naming scheme* and unreliable *delivery capability* of packets (*datagrams*) from *host-to-host*
  - **UDP (Unreliable Datagram Protocol)**
    - Uses IP to provide *unreliable* datagram delivery from *process-to-process*
  - **TCP (Transmission Control Protocol)**
    - Uses IP to provide *reliable* byte streams from *process-to-process*

- Accessed via a mix of Unix file I/O and functions from the *sockets interface*
Organization of an Internet Application

**Internet client host**

- **Client** (User code)
- **TCP/IP** (Kernel code)
- **Network adapter** (Hardware and firmware)
- **Sockets interface** (system calls)

**Internet server host**

- **Server**
- **TCP/IP**
- **Network adapter**

**Global IP Internet**

- **Network adapter**
- **Hardware and firmware**
- **Sockets interface** (system calls)
A Programmer’s View of the Internet

- Hosts are mapped to a set of 32-bit *IP addresses*
  - 128.122.49.30

- The set of IP addresses is mapped to a set of identifiers called Internet *domain names*
  - 128.122.49.30 is mapped to `cs.nyu.edu`

- A process on one Internet host can communicate with a process on another Internet host over a *connection*
The Internet maintains a mapping between IP addresses and domain names in a huge worldwide distributed database called DNS.

Conceptually, programmers can view the DNS database as a collection of millions of host entries. Each host entry defines the mapping between a set of domain names and IP addresses.

These names can be resolved on the command line using `nslookup` command.

Many names can be mapped to the same IP or multiple IPs mapped to same name. Why? Geolocation.
Internet Connections

- Clients and servers communicate by sending streams of bytes over connections. Each connection is:
  - Point-to-point: connects a pair of processes.
  - Full-duplex: data can flow in both directions at the same time,
  - Reliable: stream of bytes sent by the source is eventually received by the destination in the same order it was sent.

- A port is a 16-bit integer that identifies a process:
  - Ephemeral port: Assigned automatically by client kernel when client makes a connection request. (Also used by server in some cases)
  - Well-known port: Associated with some service provided by a server (e.g., port 80 is associated with Web servers)

- A socket is an endpoint of a connection
  - Socket address is an IP address: port pair
Popular services have permanently assigned well-known ports and corresponding well-known service names:

- Examples
  - echo server: 7/echo
  - ssh servers: 22/ssh
  - email server: 25/smtp
  - web servers: 80/http

- 0-1023 have special semantics

Mappings between well-known ports and service names is contained in the file /etc/services on each Linux machine.
Anatomy of a Connection

- A connection is uniquely identified by the socket addresses of its endpoints (socket pair)
  - \((cliaddr:cliport, servaddr:servport)\)

*Client socket address*
128.2.194.242:51213

*Server socket address*
208.216.181.15:80

Connection socket pair
(128.2.194.242:51213, 208.216.181.15:80)

*Client host address*
128.2.194.242

*Server host address*
208.216.181.15

51213 is an ephemeral port allocated by the kernel

80 is a well-known port associated with Web servers
Sockets

- **What is a socket?**
  - To the kernel, a socket is an endpoint of communication
  - To an application, a socket is a file descriptor that lets the application read/write from/to the network
    - All Unix I/O devices, including networks, are modeled as files

- **Clients and servers communicate with each other by reading from and writing to socket descriptors**

- **The main distinction between regular file I/O and socket I/O is how the application “opens” the socket descriptors**
Socket Address Structures

- **Generic socket address:**
  - For address arguments to `connect`, `bind`, and `accept`
  - Necessary only because C did not have generic (void*) pointers when the sockets interface was designed

```c
struct sockaddr {
    uint16_t sa_family;  /* Protocol family */
    char    sa_data[14]; /* Address data. */
};
```
- **Internet-specific socket address:**
  - Must cast `struct sockaddr_in*` to `struct sockaddr*` for functions that take socket address arguments.

```c
struct sockaddr_in  {
    uint16_t    sin_family; /* Protocol family (always AF_INET) */
    uint16_t    sin_port;  /* Port num in network byte order */
    struct in_addr sin_addr; /* IP addr in network byte order */
    unsigned char sin_zero[8]; /* Pad to sizeof(struct sockaddr) */
};
```

![Diagram showing the structure of `struct sockaddr_in` and its fields](image)
**Client/Server**

1. **Start server**
   - Socket
   - Bind
   - Listen
   - Accept
   - Connect

2. **Start client**
   - Socket
   - Connect
   - Write
   - Read

3. **Exchange data**
   - Write
   - Read
   - Close

4. **Disconnect client**
   - Read
   - Close

5. **Drop client**
   - Read
   - Close
Sockets Interface: `socket`

- Clients and servers use the `socket` function to create a socket descriptor:

```c
int socket(int domain, int type, int protocol)
```

- Example:

```c
int clientfd = Socket(AF_INET, SOCK_STREAM, 0);
```

  Indicates that we are using 32-bit IPv4 addresses
  Indicates that the socket will be the end point of a TCP connection
Sockets Interface: **bind**

- A server uses `bind` to ask the kernel to associate the server’s socket address with a socket descriptor:

```c
int bind(int sockfd, sockaddr* addr, socklen_t addrlen);
```

- The process can read bytes that arrive on the connection whose endpoint is `addr` by reading from descriptor `sockfd`.

- Similarly, writes to `sockfd` are transferred along connection whose endpoint is `addr`.
Sockets Interface: `listen`

- By default, kernel assumes that descriptor from socket function is an active socket that will be on the client end of a connection.

- A server calls `listen` to tell the kernel that a descriptor will be used by a server rather than a client:

  ```c
  int listen(int sockfd, int backlog);
  ```

- Converts `sockfd` from an active socket to a *listening* socket that can accept connection requests from clients.

- `backlog` is a hint about the number of outstanding connection requests that the kernel should queue up before starting to refuse requests.
**Sockets Interface: accept**

- Servers wait for connection requests from clients by calling `accept`:

  ```c
  int accept(int listenfd, sockaddr* addr, int* addrlen);
  ```

- Waits for connection request to arrive on the connection bound to `listenfd`, then fills in client’s socket address in `addr` and size of the socket address in `addrlen`.

- Returns a connected descriptor that can be used to communicate with the client via Unix I/O routines.
Connected vs. Listening Descriptors

- **Listening descriptor**
  - End point for client connection requests
  - Created once and exists for lifetime of the server

- **Connected descriptor**
  - End point of the connection between client and server
  - A new descriptor is created each time the server accepts a connection request from a client
  - Exists only as long as it takes to service client

- **Why the distinction?**
  - Allows for concurrent servers that can communicate over many client connections simultaneously
    - E.g., Each time we receive a new request, we fork a child to handle the request
A client establishes a connection with a server by calling `connect`:

```c
int connect(int clientfd, sockaddr* addr, socklen_t addrlen);
```

Attempts to establish a connection with server at socket address `addr`
- If successful, then `clientfd` is now ready for reading and writing.
- `addrlen` is `sizeof(sockaddr_in)`
1. Server blocks in \texttt{accept}, waiting for connection request on listening descriptor \texttt{listenfd}

2. Client makes connection request by calling and blocking in \texttt{connect}

3. Server returns \texttt{connfd} from \texttt{accept}. Client returns from \texttt{connect}. Connection is now established between \texttt{clientfd} and \texttt{connfd}
Socket Client & Server

- Lets take a look at some code…
- See lecture24/client.c and lecture24/server.c