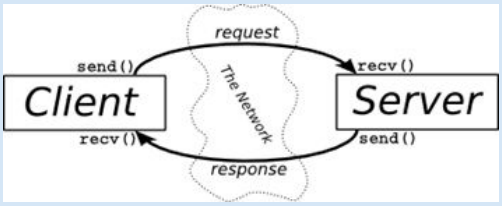
**Client-Server Model**

- Usually 2 computers, one provides a service (server), one issues service requests (clients)
- Form of interaction:
  - a *server* starts first and awaits contact
  - a *client* starts second and initiates the connection




- Some examples of client-server model: file transfer service; web service; email service; video on demand service; content sharing service; real-time search service; domain name service; address allocation service..

- Key characters of a **client**
  - client often runs locally on a user's personal computer and is invoked directly by a user
  - client actively initiates contact with the server
  - client can access multiple services as needed (browsers), but usually contacts one remote server at a time
  - client typically does not require powerful computer hardware

- Key characters of a **server**
  - it is dedicated to provide one service that can handle multiple remote clients at the same time
  - it is often invoked automatically when a system boots, and continues to execute through many sessions
  - it waits passively for contact from remote clients
  - it accepts contact from arbitrary clients, but offers a single service
  - it may require powerful hardware and a sophisticated OS

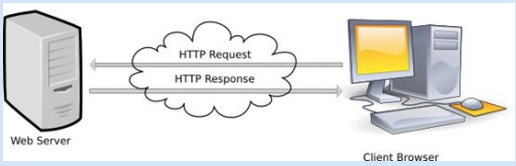
- Server
  - refers to a *program* that waits passively for communication, not the *computer* on which it executes
  - when a computer is dedicated to running one or more service programs, the computer itself is sometimes called a server (ok...)

- Server vs. Client

| | Server | Client |
|---|---|---|
| Start | Starts first | Starts second |
| Destination | Does not need to know which client will contact | Must know which server to contact |
| Connection | Waits passively and arbitrarily long for connection from clients | Initiate a connection whenever a communication is needed |
| Communication | Communicate with a client by both sending and receiving data | Communicate with a server by both sending and receiving data |
| Termination | Stays running after serving one client and waits for another | May terminate after interaction with a server |

**Requests and Responses**

- Once contact has been established, two-way communication is possible (i.e., data can flow from a client to a server or from a server to a client)
- In some cases, a client sends a series of requests and the server issues a series of responses (e.g., a database client might allow a user to look up more than one item per session)




**Multiple servers**

- Allowing a given computer to operate multiple services is helpful because…
  - hardware can be shared
  - single computer has lower system administration overhead than multiple computer systems
  - experience has shown that the demand for a service is often sporadic
  - If demand for a service is low, consolidating services on a single computer can dramatically reduce cost

**Multiple clients**

- A single computer can run…
  - a single client
  - multiple copies of a client that contact a server (ssh command window * 10 lol)
  - multiple clients that each contact a particular server
- Allowing multiple clients is useful because…
  - servers (services) can be accessed simultaneously
- For example, a user can have three 3 windows open simultaneously running three 3 applications
  - one that retrieves and displays email
  - another that connects to a chat service
  - and a third running a web browser

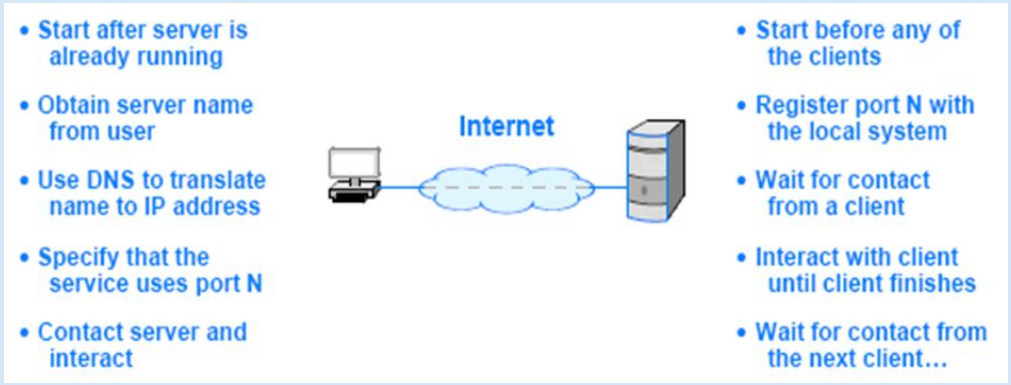
So we have clients and servers, and we need a formal way to exactly specify the communication between servers and clients; let's consider the **TCP/IP protocal suite**.

TCP/IP protocol suite is a group of protocol and application programming interfaces which are used to communicate over the Internet.

**IP**: the Internet Protocol - handles **addressing** used to identify specific hosts, and delivery of messages between hosts
**TCP**: the Transmission Control Protocol - is a **heavyweight** communication protocol which provides a *reliable* communication channel across an IP network at the expense of performance
**UDP** : the User Datagram Protocol (aka the Unreliable Datagram Protocol) – provides a **lightweight** communication channel across an IP network at the expense of program complexity (i.e., extra effort is needed if reliable channel is needed).
**ICMP**: the Internet Control Message Protocol – provides a method for handling error conditions, and network control.

**Server Identification**

- The Internet protocols divide server/service identification into two pieces:
  - an identifier for the *computer* on which a service runs (**IP**)
  - an identifier for a *service* on the computer (**Port Number**)
- Identifier for the computer
  - Unique 32 (or 128) bit identifier known as an Internet Protocol address (IP address)
  - Client must specify the server's IP address
    - Each computer is also assigned a name, and the **Domain Name System** (DNS) is used to translate names into addresses
    - Thus, a user can specify a name such as www.cse.nd.edu rather than an integer address
- Identifier for a service
  - Assigned a unique 16-bit identifier known as a protocol port number (or port number)
    - Email = port number 25, and the Web = port number 80, SSH = port number 22…
  - When a service begins execution…
    - it registers with its local OS by specifying the port number it will use
  - When a client contacts a remote server to request service…
    - the request contains a port number
  - When a request arrives at a server
    - software on the server uses the port number in the request to determine which should handle the request (demultiplexing)
- Summary (with order)




**Communication Channels**

- The TCP/IP protocol suite provides two communication protocols: **TCP and UDP**
  - TCP is a *high-overhead* protocol. The network code takes care of many error conditions, and provides a reliable communication channel. But due to this overhead, the throughput suffers.
  - UDP is a *low-overhead* protocol. The programmer has to provide the error handling code. The channel is fast, but reliability suffers
- **Message** Paradigm (UDP)
  - network accepts and delivers messages
  - messages are self-contained
    - protocol simply forwards based on the self-contained address
    - N bytes in a msg, N bytes received on the other side
  - spray and pray - no guarantees
    - may be lost, deliver out of order, or duplicated. creates more burden on programmer

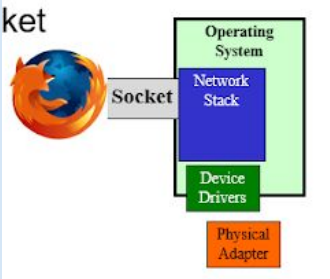
- **Stream** Paradigm (TCP)
    - stream: sequence of bytes flow from one application program to another
    - bi-directional: client to server, server to client
    - no boundaries or size limits
    - reliable data delivery
- Comparison

| Stream Paradigm (TCP) | Message Paradigm (UDP) |
|---|---|
| Connection oriented | Connectionless |
| 1-to-1 communication | Many-to-many communication |
| Sequence of individual bytes | Sequence of individual messages |
| Arbitrary length transfer | Fixed message size |
| Used by most applications | Used for multimedia applications |
| Built on TCP protocol | Built on UDP protocol |

Ok, since we have the architecture, now what?
Gotta start developing code to implement the service functions!

We need to understand what TCP/IP provides in respect to an application programming interface (API).
- The underlying feature of TCP/IP programming is the idea of a communication socket.
- The OS provides a bunch of functions and libraries you can call to develop your service code.

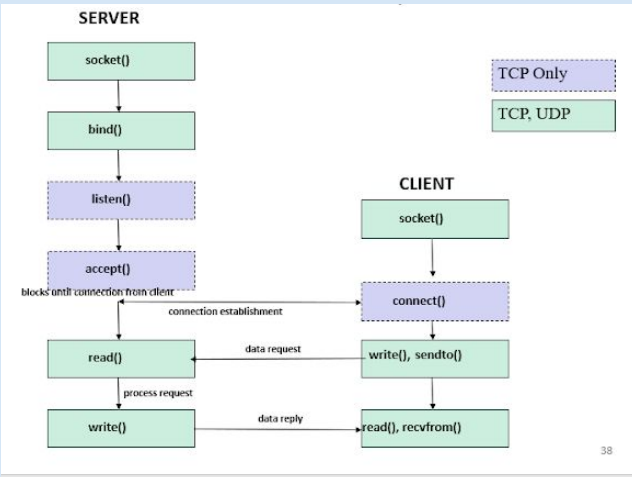
**Sockets**
- Socket programming
    - Adapter that gets an app onto the network
    - API
    - Depend on the type of socket




- Socket operations
    - Specify local/remote communication endpoints
    - Client: Initiate a connection; Server: Wait for a connection
    - Send/receive data; handle incoming data; terminate connection gracefully; handle error conditions; allocate and release local resources; etc.

- Socket communication
    - kinda similar to **file** I/O… you have open, close, read, write, seek, etc.

- think of socket as the **file descriptor** for network communications, and we can create, read, write and close the socket
- the order of these operations:



(wow tbt old systems days...)

- a nice guide to network programming: Beej's guide to network programming http://beej.us/guide/bgnet/