Distributed Computing

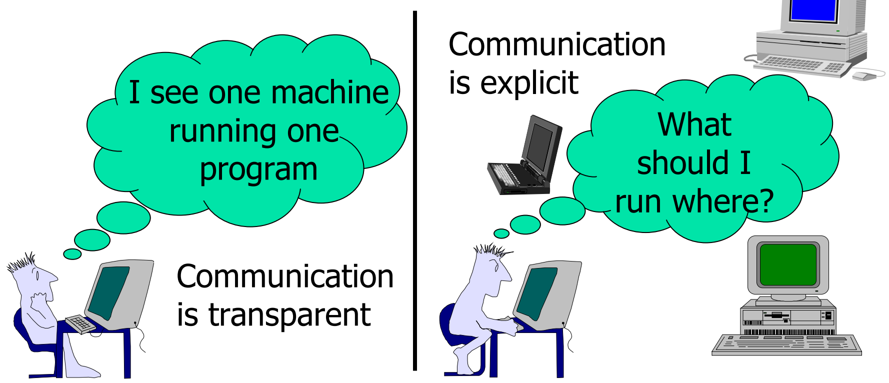
A Programmer's Perspective

Advent of Distributed Computing

- Early programming models were designed for a single computer – there was no notion of communication between multiple machines
- With the advent of computer networks, an important question arose – how should communication be presented to the programmer? what communication abstractions make more sense?
- Different paradigms presented different communication abstractions

Main Issue in Designing Communication Abstractions

- Should the programmer write communication *explicitly* or should communication be hidden from the programmer?

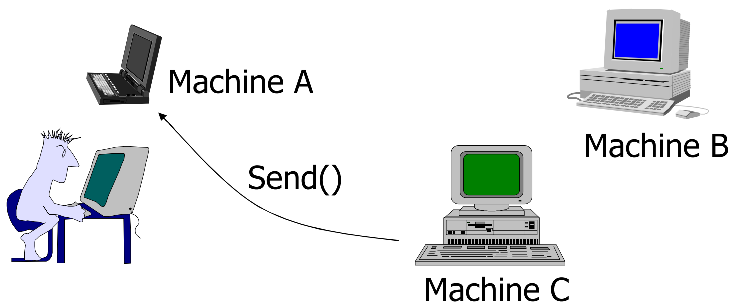


Paradigms for Distributed Computing

- UNIX IPC (Inter-Process Communication)
- RPC (Remote Procedure Calls)
- Distributed Shared Memory
- Object-Oriented Model

Inter-Process Communication (UNIX Socket IPC)

- Simplest approach – programmer does it all.
  - World consists of multiple machines and data
  - Data is sent from one machine to another using **`send(machine,port,`*`data`*`)`**

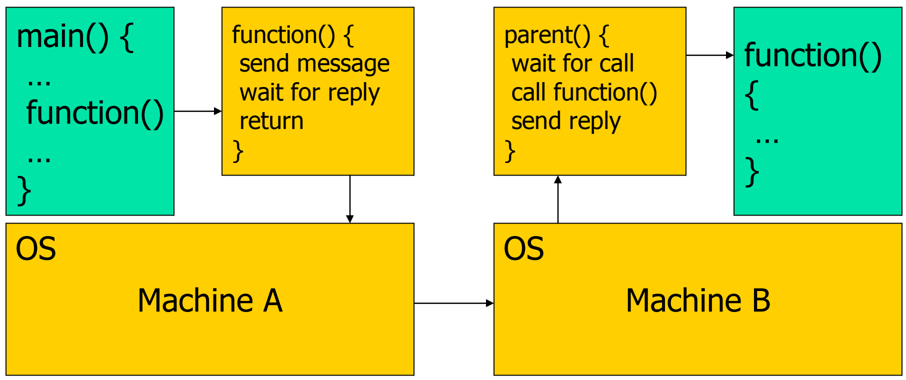


Remote Procedure Calls

- Communication is completely hidden
  - World is a single machine running a single sequential program
  - The compiler distributes different program functions to different machines to improve performance
  - Programmer does not have to know that multiple machines are involved

Implementing RPC

- Making remote function calls look like local ones

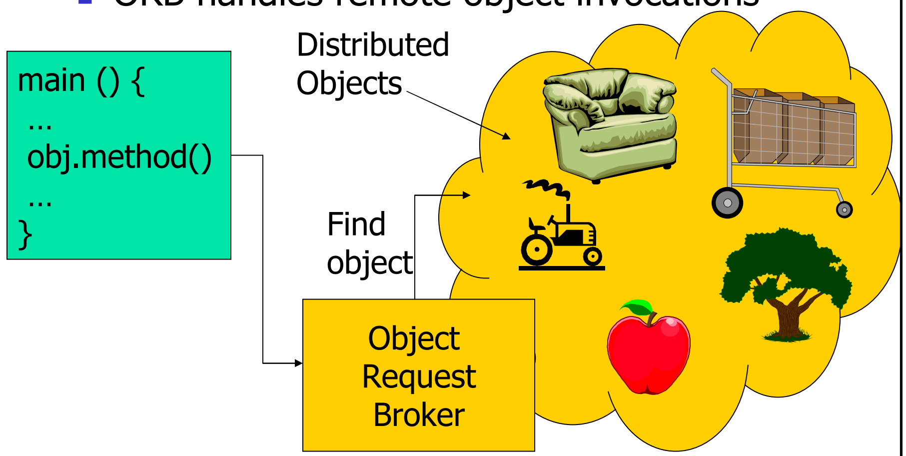


Distributed Objects (CORBA)

- World is made of objects
- Programmer invokes methods on objects without having to know where the objects are
- Objects reside on multiple machines. System delivers method calls to objects across network (CORBA)

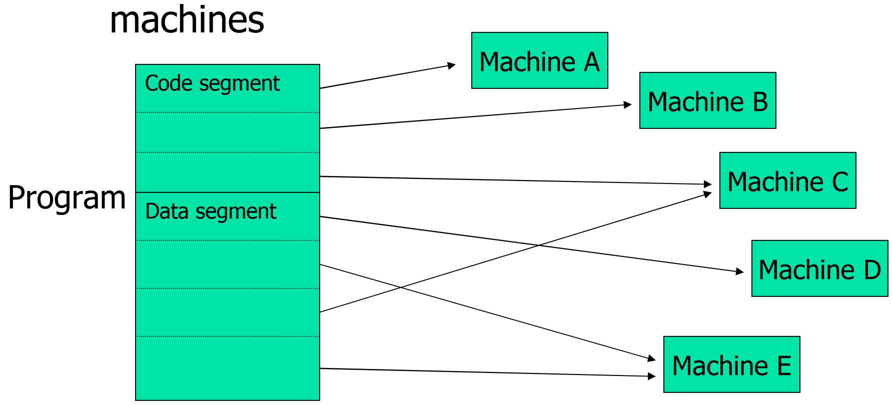
Distributed Objects (CORBA)

- ORB handles remote object invocations



Distributed Shared Memory (Not widely used)

- Program has big virtual address space
- Address space is distributed among multiple machines



Group Communication

- World consists of multicast groups
- Each group may span multiple machines
- Programmer knows about groups but does not have to worry about their physical locations
- A message sent to a group is received by all members

Important Observation

- All distributed communication paradigms can be built on top of IPC.
- Models of distributed computing using IPC:
  - Client-server
  - Peer-to-peer

Client-Server Model

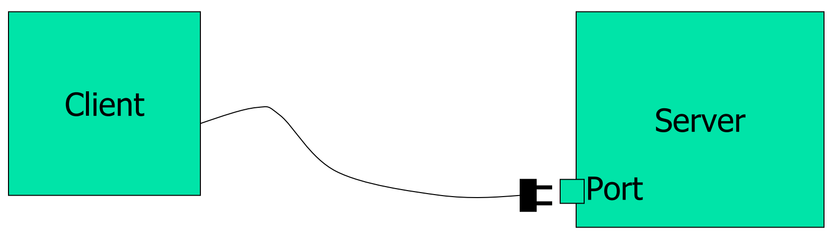
- World is made of:
  - Servers: machine who provide services to a population of clients (e.g., web servers, pop mail servers, audio servers, etc)
  - Clients: those who connect to servers (e.g., browsers, mail clients, etc)
- Servers are “well known”

Peer-to-Peer

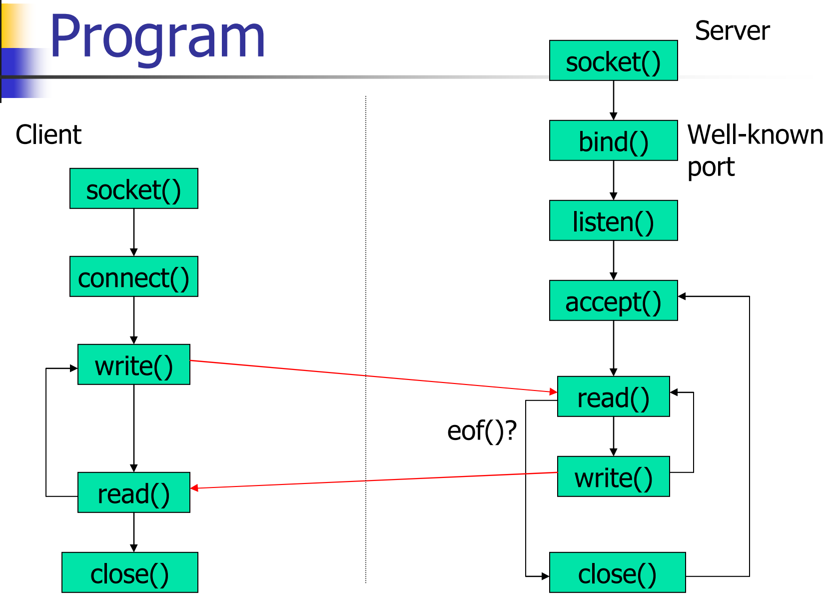
- All machines are equal – there is no separation into servers and clients
- Machines collectively perform a service to their peers
- Advantages:
  - No central point of failure
  - Potentially more scalable
- Disadvantages:
  - More difficult to program

The Socket Abstraction

- Client plugs into a server port
- Connection creates a bi-directional pipe



A Simple Client-Server Program



The `socket()` Call

- Creates a socket of a particular type
- int socket (int *family*, int *type*, int *protocol*)
  - Family
    - AF_INET: IPv4 protocols
    - AF_INET6: IPv6 protocols
    - AF_LOCAL: UNIX socket
    - AF_ROUTE: Routing socket
  - Type
    - SOCK_STREAM: Stream (TCP) socket
    - SOCK_DGARM: Datagram (UDP) socket
    - SOCK_RAW: Raw (IP) socket

The `bind()` Call

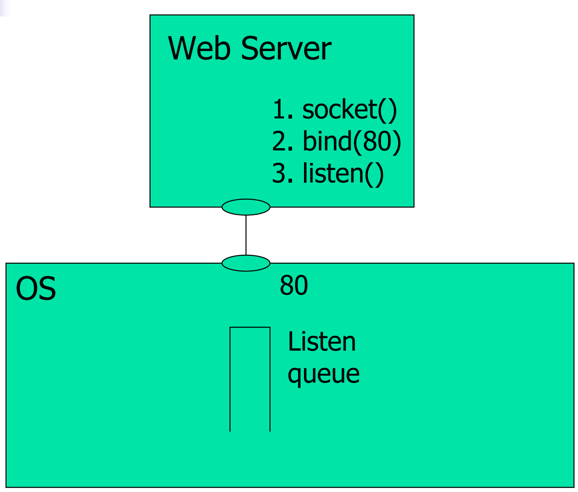
- Executed on the server to assign a (well-known) port address to the socket
- int bind (int *sockfd*, const struct sockaddr **myaddr*, socklen_t *addrlen*)

IP Address
Port Address

The `listen()` call

- Moves the socket from the CLOSED to the LISTEN state in the TCP state diagram – socket is now ready to accept connections
- int listen (int *sockfd*, int *backlog*)

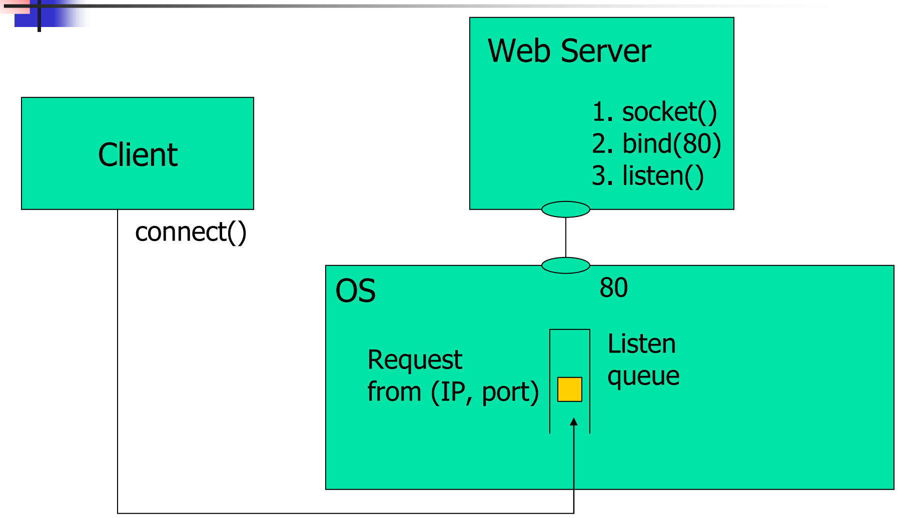
Server Initialization



The `connect()` Call

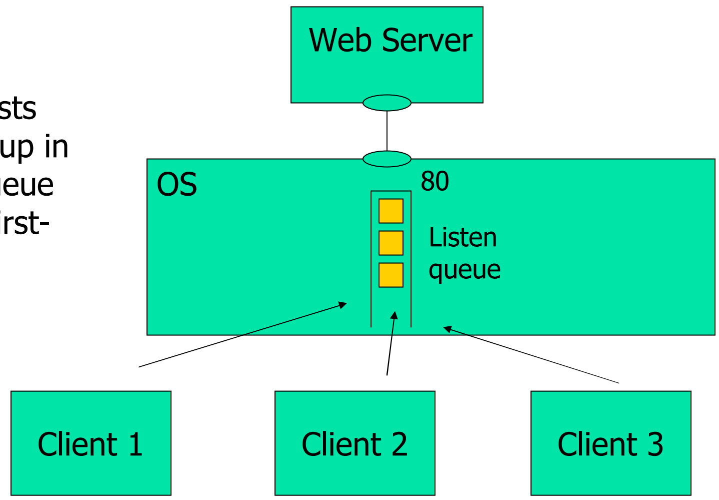
- Establishes a connection with a server
- int connect (int *sockfd*, const struct sockaddr **servaddr*, socklen_t *addrlen*)

Connecting to the Server



Busy Server Operation

Client requests get queued-up in the listen queue
First-come first-served



The `accept()` Call

Client requests get queued-up in the listen queue
First-come first-served