Course Snapshot

We have covered all the fundamental OS components:

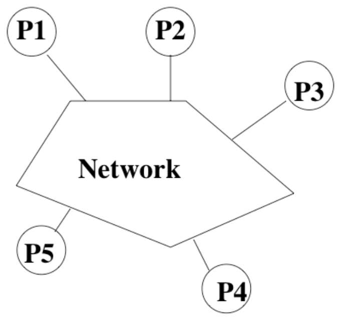
- Architecture and OS interactions
- Processes and threads
- Synchronization and deadlock
- Process scheduling
- Memory management
- File systems and I/O

The Next Few Classes

- Distributed Systems
  - Networking Basics
  - Distributed services (email, www, telnet)
  - Distributed Operating Systems
  - Distributed File Systems
- Guest lectures and special topics
  - Linux (already done ✅)

Distributed Systems

- **Distributed system:** a set of physically separate processors connected by one or more communication links



- Nearly all systems today are distributed in some way.
  - Email, file servers, network printers, remote backup, world wide web

Parallel versus Distributed Systems

- **Tightly-coupled systems:** "parallel processing"
  - Processors share clock, memory, and run one OS
  - Frequent communication
- **Loosely-coupled systems:** "distributed computing"
  - Each processor has its own memory
  - Each processor runs an independent OS
  - Communication should be less frequent

Advantages of Distributed Systems

- **Resource sharing:**
  - Resources need not be replicated at each processor (for example, shared files)
  - Expensive (scarce) resources can be shared (for example, printers)
  - Each processor can present the same environment to the user (for example, by keeping files on a file server)
- **Computational speedup:**
  - *n* processors potentially gives you n times the computational power
  - Problems must be decomposable into subproblems
  - Coordination and communication between cooperating processes (synchronization, exchange of results) is needed.

Advantages of Distributed Systems

- **Reliability:**
  - Replication of resources yields fault tolerance.
  - For example, if one node crashes, the user can work on another.
  - Performance will degrade, but system remains operational.
  - However, if some component of the system is centralized, a single point of failure may result
  - **Example:** If an Edlab workstation crashes, you can use another workstation. If the file server crashes, none of the workstations are useful.
- **Communication:**
  - Users/processes on different systems can communicate.
  - For example, mail, transaction processing systems like airlines, and banks, WWW.

Distributed Systems

- Modern work environments are distributed => operating systems need to be distributed
- What do we need to consider when building these systems?
  - Communication and networks
  - Transparency (how visible is the distribution?)
  - Security
  - Reliability
  - Performance and scalability
  - Programming models

Distributed System Design

What gets harder when we move from a stand alone system to a distributed environment?

- resource sharing
- timing (e.g., synchronization)
- critical sections
- deadlock detection and recovery
- failure recovery

Networks

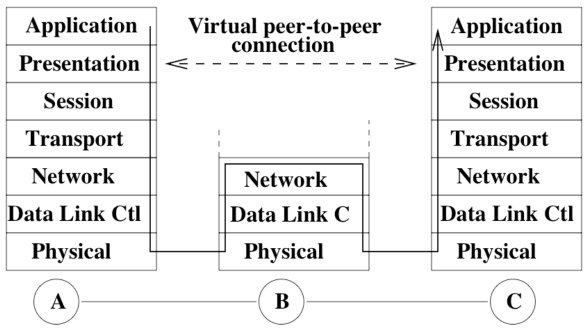
- Networks are usually concerned with providing efficient, correct, and robust message passing between two separate nodes.
- **Local Area Network (LAN)** usually connects nodes in a single building and needs to be fast and reliable (for example, Ethernet).
  - **Media:** twisted-pair, coaxial cable, fiber optics
  - **Typical bandwidth:** 10-100-1000 Mb/s (10Gb/s now available)
- **Wide Area Network (WAN)** connects nodes across the state, country, or planet.
  - WANs are typically slower and less reliable than LAN (for example, Internet).
  - **Media:** telephone lines (T1 service), microwave links, satellite channels
  - **Typical bandwidth:** 1.544 Mb/s (T1), 45 Mb/s (T3)

Principles of Network Communication

- Data sent into the network is chopped into "packets", the network's basic transmission unit.
- Packets are sent through the network.
- Computers at the switching points control the packet flow.
- **Analogy:** cars/road/police - packets/network/computer
- Shared resources can lead to contention (traffic jams).
- **Analogy:**
  - **Shared node -** Mullins Center
  - **Shared link** - bridge

Communication Protocols

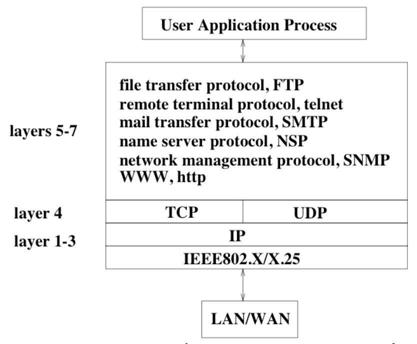
- Protocol: a set of rules for communication that are agreed to by all parties
- Protocol stack : networking software is structured into layers
  - Each layer N, provides a service to layer N+1, by using its own layer N procedures and the interface to the N-1 layer.
  - Example: International Standards Organization/ Open Systems Interconnect (ISO/OSI)



ISO Network Protocol Stack

- **Application layer:** applications that use the net, e.g., mail, netscape, X-services, ftp, telnet, provide a UI
- **Presentation layer:** data format conversion, e.g., big/little endian integer format)
- **Session layer:** implements the communication strategy, such as RPC. Provided by libraries.
- **Transport layer:** reliable end-to-end communication between any set of nodes. Provided by OS.
- **Network layer:** routing and congestion control. Usually implemented in OS.
- **Data Link Control layer:** reliable point-to-point communication of packets over an unreliable channel. Sometimes implemented in hardware, sometimes in software (PPP).
- **Physical layer:** electrical/optical signaling across a “wire”. Deals with timing issues. Implemented in hardware.

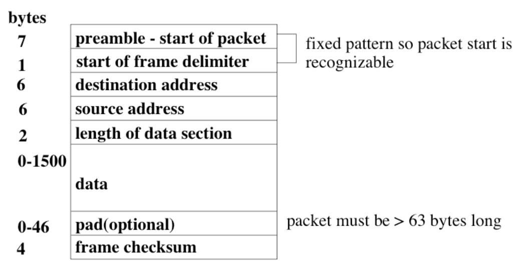
TCP/IP Protocol Stack


User Application Process
layers 5-7
file transfer protocol, FTP
remote terminal protocol, telnet
mail transfer protocol, SMTP
name server protocol, NSP
network management protocol, SNMP
WWW, http
layer 4
TCP
UDP
layer 1-3
IP
IEEE802.X/X.25
LAN/WAN

- Most Internet sites use TCP/IP - Transmission Control Protocol/ Internet Protocol.
  - It has fewer layers than ISO to increase efficiency.
  - Consists of a suite of protocols: UDP, TCP, IP...
  - TCP is a **reliable** protocol -- packets are received in the order they are sent
  - UDP (user datagram protocol) an **unreliable** protocol (no guarantee of delivery).

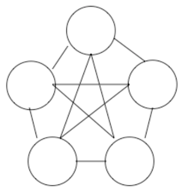
Packet

- Each message is chopped into packets.
  - Each packet contains all the information needed to recreate the original message.
  - For example, packets may arrive out of order and the destination node must be able to put them back into order.
  - Ethernet Packet Contents



  - The data segment of the packet contains headers for higher protocol layers and actual application data

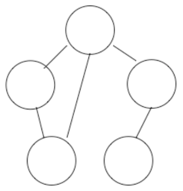
Point-to-Point Network Topologies



Fully Connected

- **Fully connected:** all nodes connected to all other nodes
  - Each message takes only a single "hop", i.e., goes directly to the destination without going through any other node
  - Failure of any one node does not affect communication between other nodes
  - Expensive, especially with lots of nodes, not practical for WANs

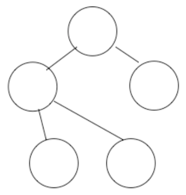
Point-to-Point Network Topologies



Partially Connected

- **Partially connected:** links between some, but not all nodes
  - Less expensive, but less tolerant to failures. A single failure can partition the network.
  - Sending a message to a node may have to go through several other nodes => need routing algorithms.
  - WANs typically use this structure.

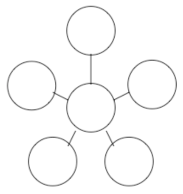
Point-to-Point Networks Topologies



Tree Structured

- **Tree structure:** network hierarchy
  - All messages between direct descendants are fast, but messages between "cousins" must go up to a common ancestor and then back down.
  - Some corporate networks use this topology, since it matches a hierarchical world view...
  - Not tolerant of failures. If any interior node fails, the network is partitioned.

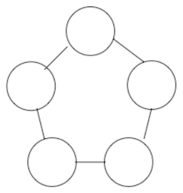
Point-to-Point Networks Topologies



Star

- **Star:** - all nodes connect to a single centralized node
  - The central site is generally dedicated to network traffic.
  - Each message takes only two hops.
  - If one piece of hardware fails, that disconnects the entire network.
  - Inexpensive, and sometimes used for LAN

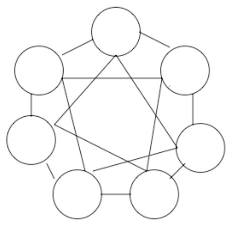
Ring Networks Topologies



Ring

- **One directional ring -** nodes can only send in one direction.
  - Given *n* nodes, message may need to go *n-1* hops.
  - Inexpensive, but one failure partitions the network.
- **Bi-directional ring -** nodes can send in either direction.
  - With *n* nodes, a message needs to go at at most *n*/2 hops.
  - Inexpensive, tolerates a single failure by increasing message hops. Two failures partition the network.

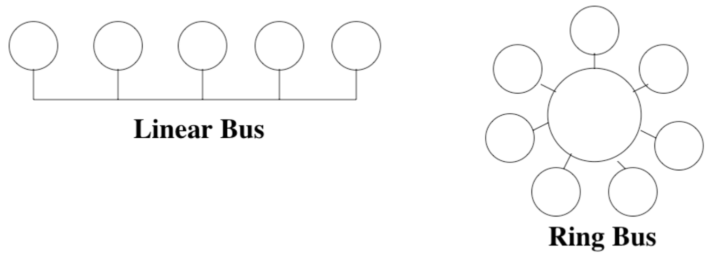
Ring Networks Topologies



Doubly Linked Ring

- **Doubly connected ring** nodes connected to neighbors and one away neighbors
  - A message takes at most *n*/4 hops.
  - More expensive, but more tolerant of failures.

Bus Network Topologies



- **Bus** nodes connect to a common network
- **Linear bus** - single shared link
  - Nodes connect directly to each other using multiaccess bus technology.
  - Inexpensive (linear in the number of nodes) and tolerant of node failures.
  - Ethernet LAN use this structure.
- **Ring bus** - single shared circular link
  - Same technology and tradeoffs as a linear bus.

Resource Sharing

There are many mechanisms for sharing (hardware, software, data) resources.

- **Data Migration:** moving the data around
- **Computation Migration:** move the computation to the data
- **Job Migration**: moving the job (computation and data) or part of the job

=> The fundamental tradeoff in resource sharing is to complete user instructions as fast and as cheaply as possible. (Fast and cheap are usually incompatible.)

If communication is cheap: use all resources
If computation is slow/expensive: local processing
Reality is somewhere in between

Client/Server Model

- One of the most common models for structuring distributed computation is by using the *client/server* paradigm.
  - A *server* is a process or collection of processes that provide a service, e.g., name service, file service, database service, etc.
  - The server may exist on one or more nodes.
  - A *client* is a program that uses the service.
  - A client first binds to the server, i.e., locates it in the network and establishes a connection.
  - The client then sends the server a request to perform some action. The server sends back a response.
  - RPC is one common way this structure is implemented.

Remote Procedure Call

Basic idea:

- Servers export procedures for some set of clients to call.
- To use the server, the client does a procedure call.
- OS manages the communication.

Remote Procedure Call: Implementation Issues

For each procedure on which we want to support RPC:

- The RPC mechanism uses the procedure *signature* (number and type of arguments and return value)
  - to generate a client stub that bundles up the RPC arguments and sends it off to the server, and
  - to generate the server stub that unpacks the message, and makes the procedure call.

Remote Procedure Call: Implementation Issues

Client Stub:

```
build message
send message
wait for response
unpack reply
return result
```

Server Stub:

```
create threads
loop
    wait for a command
    unpack request parameters
    call procedure with thread
    build reply with result(s)
    send reply
end loop
```

Comparison between RPC and a regular procedure call

- Name of procedure
- Parameters
- Result
- Return address

Remote Procedure Call

- How does the client know the right port?
  - The binding can be static - fixed at compile time.
  - Or the binding can be dynamic - fixed at runtime.
- In most RPC systems, dynamic binding is performed using a name service.
  - When the server starts up, it exports its interface and identifies itself to a network name server
  - The client, before issuing any calls, asks the name service for the location of a server whose name it knows and then establishes a connection with the server.

Example: Remote Method Invocation (RMI) in Java

- Java provides the following classes/interfaces:
  - **Naming:** class that provides the calls to communicate with the remote object registry
  - public static void bind(String name, Remote obj) - Binds a server to a name.
  - public static Remote lookup(String name) - Returns the server object that corresponds to a name.
- **UnicastRemoteObject:** supports references to non-replicated remote objects using TCP, exports the interface automatically when the server object is constructed
- Java provides the following tools:
  - **rmiregistry** server-side name server
  - **rmic:** given the server interface, generates client and server stubs that create and interpret packets

Example: Server in Java

- **Server**
  - Defines an interface listing the signatures of methods the server will satisfy
  - Implements each of the methods in the interface
  - Main program for server:
    - Creates one or more server objects - normal constructor call where the object being constructed is a subclass of RemoteObject
    - Registers the objects with the remote object registry
- **Client**
  - Looks up the server in the remote object registry
  - Uses normal method call syntax for remote methods
  - Should handle RemoteException

Example: Hello World Server Interface

Declare the methods that the server provides:

```
package examples.hello;

// All servers must extend the Remote interface.
public interface Hello extends java.rmi.Remote {

    // Any remote method might throw RemoteException.
    // Indicates network failure.
    String sayHello() throws java.rmi.RemoteException;
}
```

Example: Hello World Server

```
package examples.hello;
import java.rmi.*;
import java.rmi.server.UnicastRemoteObject;

public class HelloImpl extends UnicastRemoteObject implements Hello
{
    public HelloImpl() throws RemoteException {
     // The superclass constructor exports the interface and gets a port
      super();
  }

  public String sayHello() throws RemoteException {
      // This is the "service" provided.
      return  "Hello World!";
  }
```

Example: Hello World Server (contd)

```
public static void main(String args[])
  {
      // Create and install a security manager
      System.setSecurityManager(new RMISecurityManager());

      // Construct the server object.
      HelloImpl obj = new HelloImpl();

      // Register the server with the name server.
      Naming.rebind("//myhost/HelloServer", obj);
  }
}
```

Example: Hello World Client

```
package examples.hello;

import java.awt.*;
import java.rmi.*;

public class HelloApplet extends java.applet.Applet {
     String message = "";

     // The init method begins the execution of the applet on the client
     // machine that is viewing the Web page containing the reference
     // to the applet.
     public void init() {
       try {
            // Looks up the server using the name server on the host that
            // the applet came from.
            Hello obj = (Hello)Naming.lookup(
                              "//" + getCodeBase().getHost() + "/HelloServer");
```

Example: Hello World Client (contd)

```
             // Calls the sayHello method on the remote object.
         message = obj.sayHello();
     } catch (RemoteException e) {
       System.out.println("HelloApplet RemoteException caught");
     }
   }

   public void paint(Graphics g) {
     // The applet will write the string returned by the remote method
     // call on the display.
     g.drawString(message, 25, 50);
   }
}
```

Summary

- Virtually all computer systems contain distributed components
- Networks hook them together
- Networks make tradeoffs between speed, reliability, and expense