Distributed Operating Systems

- How are computer systems related?
  - » Network operating systems
  - » Distributed operating systems
- How does timing work?
- Accessing remote services
- Surviving failures
  - » Machines that break
  - » Network connections that fail
- Design issues for distributed systems

Network Operating Systems

- Computers have “individual” operating systems
  - » Systems cooperate but are distinct
  - » Systems know about other computers
  - » Users can tell which machine they’re using
- Information & users have to switch machines explicitly
  - » Users use rlogin or ssh to connect to other machines
  - » Files transferred via ftp
- Systems share some information
  - » User names & access matrix
  - » Network information

Distributed Operating Systems

- Many computers act together
  - » Users can’t tell which computer they’re using
  - » Computers cooperate to make environment identical on every system
- Two kinds of transparency
  - » Data migration: data moves to where the user is running
    - – Move entire file (or more) or
    - – Move only the data needed immediately by the user
  - » Computation migration: processes migrate to machines on which data is stored
    - – May be more efficient than moving data around
    - – Harder to coordinate, and may result in poor load balancing

Process Migration

- Processes can be moved from one processor to another
  - » Suspend process on original machine or direct request to new machine
  - » Copy entire process address space to a new machine
  - » Restart it in new home
- Reasons for migrating a process are:
  - » Load balancing: try to even the load among processors
  - » Computation speedup: use faster processors for compute-intensive tasks
  - » Hardware or software preferences: run processes on a computer with the necessary hardware or software
  - » Data access: data used by the process is elsewhere, and it’s faster to move the process than the data
- Process migration is usually transparent to the user

Parallel (Multiprocessor) OS

- Multiprocessors have several CPUs in a single box
  - » Processors share memory, I/O devices
  - » Single copy of the OS in memory
  - » Example: `umbc8.umbc.edu`
- Similar to distributed operating system in many ways
  - » Scheduler may run processes on any available CPU
  - » Process migration is much cheaper: no data needs to be moved
  - » All processors share devices easily, but more synchronization is necessary (possible conflicts)
  - » Deadlock may be more likely: several things *can* happen at once

Real Distributed Systems

- Most systems have features of both distributed & network systems
- Example: Unix
  - » Users can tell which machine they're logged into
  - » User information is shared among all machines
  - » Processes aren't automatically migrated
  - » Logins can be load-balanced (as with `gl.umbc.edu`)
  - » Data access can be made transparent - all file systems available in the same way from all machines (distributed file system)
- Example: Windows NT
  - » Users know which machine they're using
  - » Files are shared among systems
  - » Processes don't move from one system to another, but can be sent to another CPU on the same system

Accessing Remote Services

- Processes on one system want to access services on another computer
  - » Use files stored on another computer
  - » Send print requests to a printer on another computer
- Remote services can be accessed by
  - » Exchanging messages using an Internet protocol
    - – Fetching files via FTP
    - – Getting Web pages via HTTP
  - » Using the *remote procedure call* (RPC) paradigm
    - – Local process makes what looks like a procedure call
    - – OS sends the information to a remote machine (if necessary) and waits for a reply
    - – OS gets the reply and returns the answer to the local process

Remote Procedure Calls

- Operating system determines whether the service is available locally
  - » If so, procedure is done locally
  - » If not, OS packages up parameters and sends them to a known *port* (port == mailbox) on a remote machine
    - – Message includes “return address” so remote machine can send the reply back
    - – Message includes both explicit parameters (from the procedure call) and implicit parameters (user name, authentication info, etc.)
- Remote server receives message and processes it
  - » Server performs same security checks as for a local procedure call
  - » Issue: how can server trust that incoming messages come from another (presumably trusted) OS?

RPC in Distributed File Systems

- Distributed file systems such as NFS use RPCs extensively
  - » Process makes a local procedure call such as `read` or `write`
  - » Operating system decides whether request can be handled locally or or must be sent to a remote server
- If file is remote, OS does the following (for a read):
  - » Send the relevant information in a message to the server
    - – File ID, offset, read size
    - – User name & authentication info
  - » Wait for a reply with the file data
  - » When the reply arrives, return the data to the user process

Handling RPCs on the Server

- Start a new process for each RPC message received
  - » Slow: new process for each remote request
  - » Simple to program
- Start a new thread in an existing task for each RPC message
  - » Faster: no need to create a new address space each time
  - » Threads for a given type of RPC can share memory & resources
  - » Can be less stable & secure: error in one thread can affect others
- Pick a thread from a pool of available threads for this RPC
  - » Even faster than starting a new thread
  - » Wasted resources: idle threads
  - » Threads may not be available during times of heavy load
  - » Same security & stability issues as other thread methods

When Distributed Systems Fail

- Distributed systems can be more vulnerable to failure than individual computers
  - » More components that can fail
  - » Increased complexity leads to higher likelihood of failure
- Failures come in two types
  - » Link between two computers fails (computers are fine)
  - » Computer in the distributed system fails (hardware or software)
- When failure occurs
  - » *Reconfigure* the system to allow work to continue
  - » *Recover* from failure when a failed component returns to service

Detecting Failures

- Exchange “are you alive” messages at fixed intervals
  - » If site A doesn’t get a message within the specified interval, it assumes that either
    - – The message was lost (retry it)
    - – Site B is down
    - – The link between A and B is down
  - » If site B doesn’t get a reply to the message it sent to A, it assumes
    - – Situation similar to the first situation
    - – Alternate paths may also be down
- Detect failures in other ways
  - » Regular messages (RPC, etc.) go unanswered
  - » Components report that they’ve failed

Reconfiguration

- Rearrange the components of the system so work can continue without the failed component
  - » Failed link can be avoided by rerouting messages
  - » Failed link can divide the system
    - – Each half of the system operates normally
    - – Operations that can only be done in the other half aren’t done (fail immediately)
  - » Failed computer can be avoided: run processes elsewhere
- Notify all computers in the distributed system of the failure
  - » Avoid delays waiting for resources that are known to have failed
  - » Send new requests somewhere that can handle them
  - » Update OS so it can make more intelligent decisions (revise scheduler, use alternate versions of files, etc.)

Recovery

- When a failed component is fixed, the system has to find out about it
  - » Active components (CPUs, etc.) can send out messages to the rest of the system
  - » Passive components (links, etc.) are discovered by
    - – Probing the component to see when it’s fixed
    - – Having a human notify the system that it’s fixed
- Recovery procedure
  - » Notify the rest of the system that the component is available
  - » Allow other components to use the fixed component again
  - » Complete any jobs that may have been waiting for the component to be fixed

Designing Distributed Systems

- Provide transparency to users
  - » Distributed system should look and feel to users no different than a single centralized computer
  - » Local & remote resources should work in the same ways
  - » Workload should be distributed evenly to all relevant resources
- User mobility
  - » Users should be able to log in anywhere and have it look the same
  - » Users' data should migrate to where they're logged in to improve performance
- Fault tolerance
  - » System should survive failures of any component
  - » Performance may degrade if components fail
  - » Users should be unaware of any failures (except for slower performance)

Designing Distributed Systems, continued

- Scalability
  - » Distributed systems should perform better when more components are added
  - » Distributed systems shouldn't have any central bottlenecks (single computer that does X for all users)
  - » A single component's demand must be bounded, regardless of the number of nodes in the system
    - – Possibly done by replicating the component and allowing requests to be spread out
- Grace under pressure
  - » System must handle high loads efficiently
    - – Reject new requests & retry them later
    - – Handle new requests, albeit slowly
  - » System must not deadlock (can be difficult in large distributed systems…)