Resource Sharing

- There are many mechanisms for sharing resources (hardware, software, data)
  - Data migration — move the data to the computation
  - Computation migration — move the computation to the data
  - Job migration — move either all or part of the job(computation and data)
  - ➥The fundamental tradeoff is to complete the user's instructions as quickly and as cheaply as possible
- Computation vs. communication
  - If communication is fast and cheap, we can utilize all the resources in the distributed environment
  - If communication is slow and expensive, we should do most computation locally

Data Migration

- Process at site A needs to access file at site B
  1. Copy file to site A, access there
     - ✔All access are local (at site A)
     - ✘Data must be converted to site A's format
     - ✘Costly transfer if file is large
     - ✘Multiple copies of file can cause consistency problems
  2. Keep file on site B, access remotely
     - ✔Saves file transfer cost
     - ✔Single copy of file, so no consistency problems
     - ✘Data must be converted to site A's format
     - ✘Single "file server" for file at site B may be a performance bottleneck

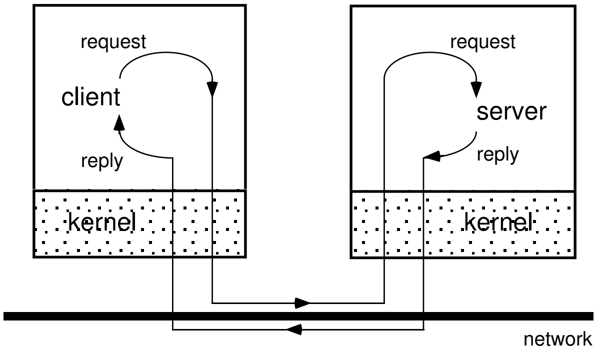
Job Migration

- When a job arrives at a given site for processing, it may be advantageous to execute all or part of the job at one or more remote sites by moving the data and the computation
  - Load balancing — evenly distribute workload across the distributed system
  - Computational speedup — concurrently (in parallel) execute parts of the job
  - Hardware preference — job may best match another piece of hardware
    - Example: vision application on a specialized parallel machine
  - Software preference — job may require software available only on a specific site
    - Example: math package licensed for use on only one computer
  - Transparency — may want to hide migration from the user

Computation Migration & Client-Server Model

- Sometimes it is more efficient to transfer the computation instead of the data
  - Example: retrieve all lines of a very large file that contain a specific substring
- Client / server model
  - *Server* = process (or collection of processes) that provides a *service*
    - Example: name service, file service
    - Multiple servers may provide that service
  - *Client* — process that uses the service
  - Request / reply protocol:
    - Client establishes connection with server
    - Client sends server a request to perform some action
    - Server sends back a response
  - Inter-process communication:
    - Message passing
    - Remote procedure call (RPC)

Message Passing



- Request / reply protocol
  - Client sends **request** message to server asking it to perform some service
  - Server performs service, sends **reply** message containing results or error code
- Requires two primitive operations:
  - send(*message*)
  - receive(*message*)

Direct vs. Indirect Communication

- Direct communication — explicitly name the process you're communicating with
  - send(*destination-process*, *message*)
  - receive(*source-process*, *message*)
  - Link is associated with exactly two processes
    - Between any two processes, there exists at most one link
    - The link may be unidirectional, but is usually bidirectional
- Indirect communication — communicate using mailboxes (owned by receiver)
  - send(*mailbox*, *message*)
  - receive(*mailbox*, *message*)
  - Link is associated with two or more processes that share a mailbox
    - Between any two processes, there may be a number of links
    - The link may be either unidirectional or bidirectional

Blocking Send & Receive

- Blocking send:
  - Sends, then *blocks* (i.e., gets suspended by OS) until either:
    - message is transmitted to receiver's kernel (unreliable)
    - message is received by receiver and acknowledged (reliable)
  - After resuming, can reuse message buffer for another message
- Blocking receive:
  - Blocks until a message is received (may be minutes, hours, …)
    - May automatically acknowledge (reliable)

Non-blocking Send & Receive

- Non-blocking send:
  - Sends, then goes on to next instruction
  - Advantage: sending process can execute in parallel with message transmission
  - Problem: must avoid modifying message buffer until message has been received (but how do you know?)
    1. Have kernel copy message to kernel, let process continue
    2. Keep message in user space, have kernel send interrupt when message has been received (difficult to program)
- Non-blocking receive:
  - Receive returns with buffer, but doesn't know if there's a message there or not
    - Polls or receives interrupt when message is ready (difficult to program)

Buffering

- Link may have some capacity that determines the number of message that can be temporarily queued in it
- Zero capacity: (queue of length 0)
  - No messages wait
  - Sender must wait until receiver receives the message — this synchronization to exchange data is called a *rendezvous*
- Bounded capacity: (queue of length *n*)
  - If receiver's queue is not full, new message is put on queue, and sender can continue executing immediately
  - If queue is full, sender must block until space is available in the queue
- Unbounded capacity: (infinite queue)
  - Sender can always continue