Networking for Operating Systems
CS 111
Operating Systems
Peter Reiher

Outline

- Networking implications for operating systems
- Networking and distributed systems

Networking Implications for the Operating System

- Networking requires serious operating system support
- Changes in the clients
- Changes in protocol implementations
- Changes to IPC and inter-module plumbing
- Changes to object implementations and semantics
- Challenges of distributed computing

Changing Paradigms

- Network connectivity becomes “a given”
  – New applications assume/exploit connectivity
  – New distributed programming paradigms emerge
  – New functionality depends on network services
- Thus, applications demand new services from the OS:
  – Location independent operations
  – Rendezvous between cooperating processes
  – WAN scale communication, synchronization
  – Support for splitting and migrating computations
  – Better virtualization services to safely share resources
  – Network performance becomes critical

The Old Networking Clients

- Most clients were basic networking applications
  - Implementations of higher level remote access protocols
    - telnet, FTP, SMTP, POP/IMAP, network printing
  - Occasionally run, to explicitly access remote systems
  - Applications specifically written to network services
- OS provided transport level services
  - TCP or UDP, IP, NIC drivers
- Little impact on OS APIs
  - OS objects were not expected to have network semantics
  - Network apps provided services, did not implement objects

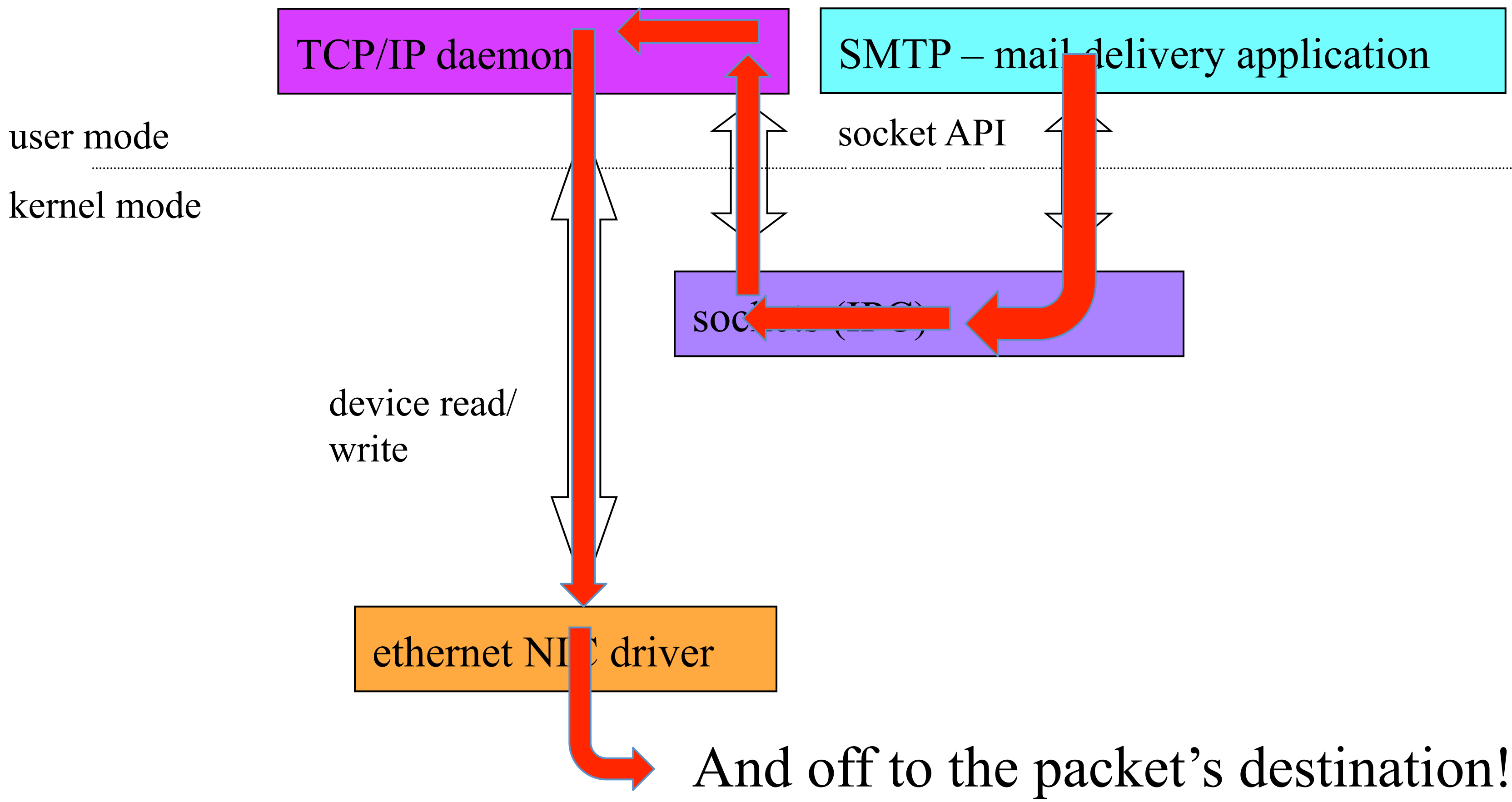
The New Networking Clients

- The OS itself is a client for network services
  - OS may depend on network services
    - netboot, DHCP, LDAP, Kerberos, etc.
  - OS-supported objects may be remote
    - Files may reside on remote file servers
    - Console device may be a remote X11 client
    - A cooperating process might be on another machine
- Implementations must become part of the OS
  - For both performance and security reasons
- Local resources may acquire new semantics
  - Remote objects may behave differently than local

The Old Implementations

- Network protocol implemented in user-mode daemon
  - Daemon talks to network through device driver
- Client requests
  - Sent to daemon through IPC port
  - Daemon formats messages, sends them to driver
- Incoming packets
  - Daemon reads from driver and interprets them
  - Unpacks data, forward to client through IPC port
- Advantages – user mode code is easily changed
- Disadvantages – lack of generality, poor performance, weak security

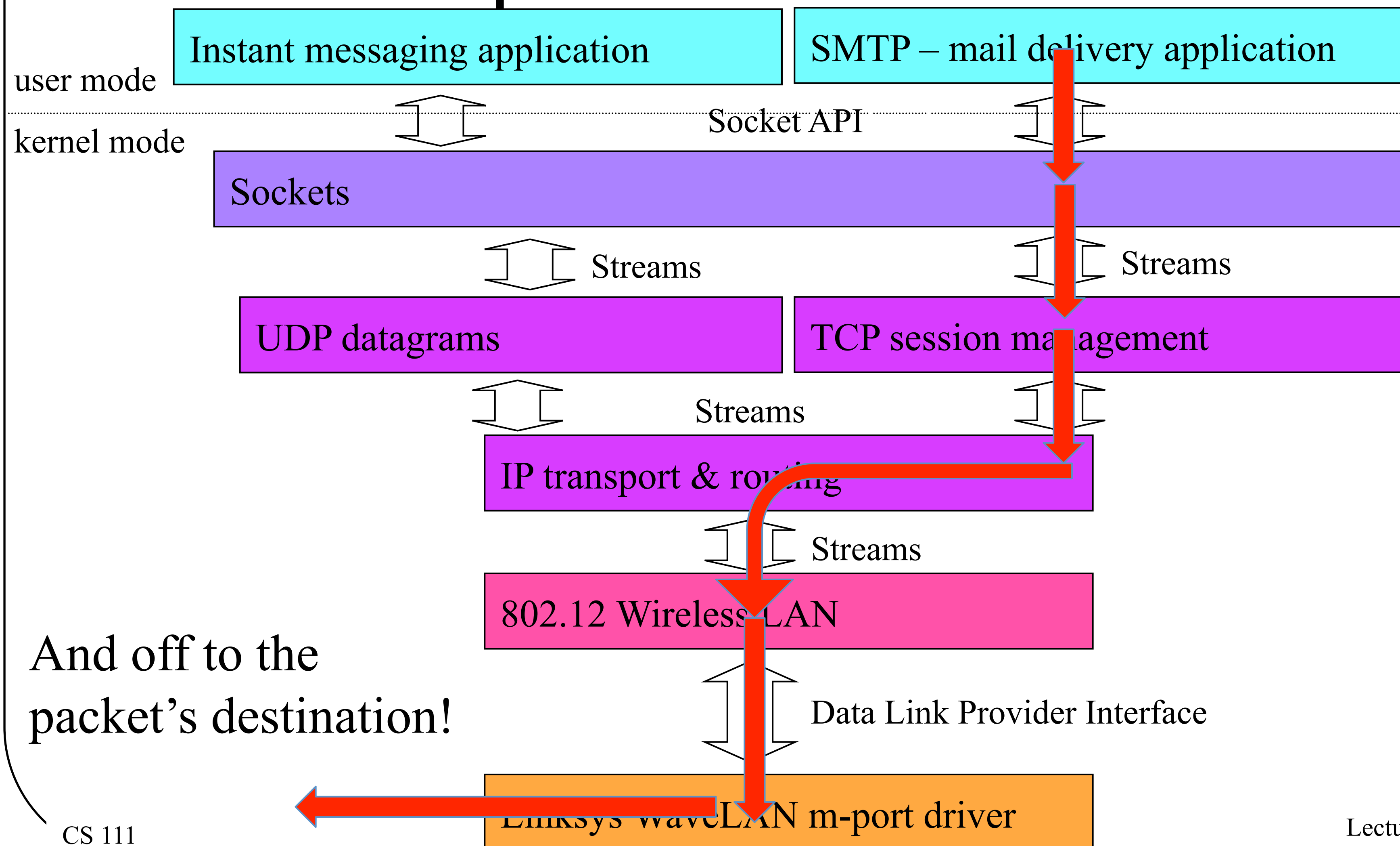
User-Mode Protocol Implementations



The New Implementations

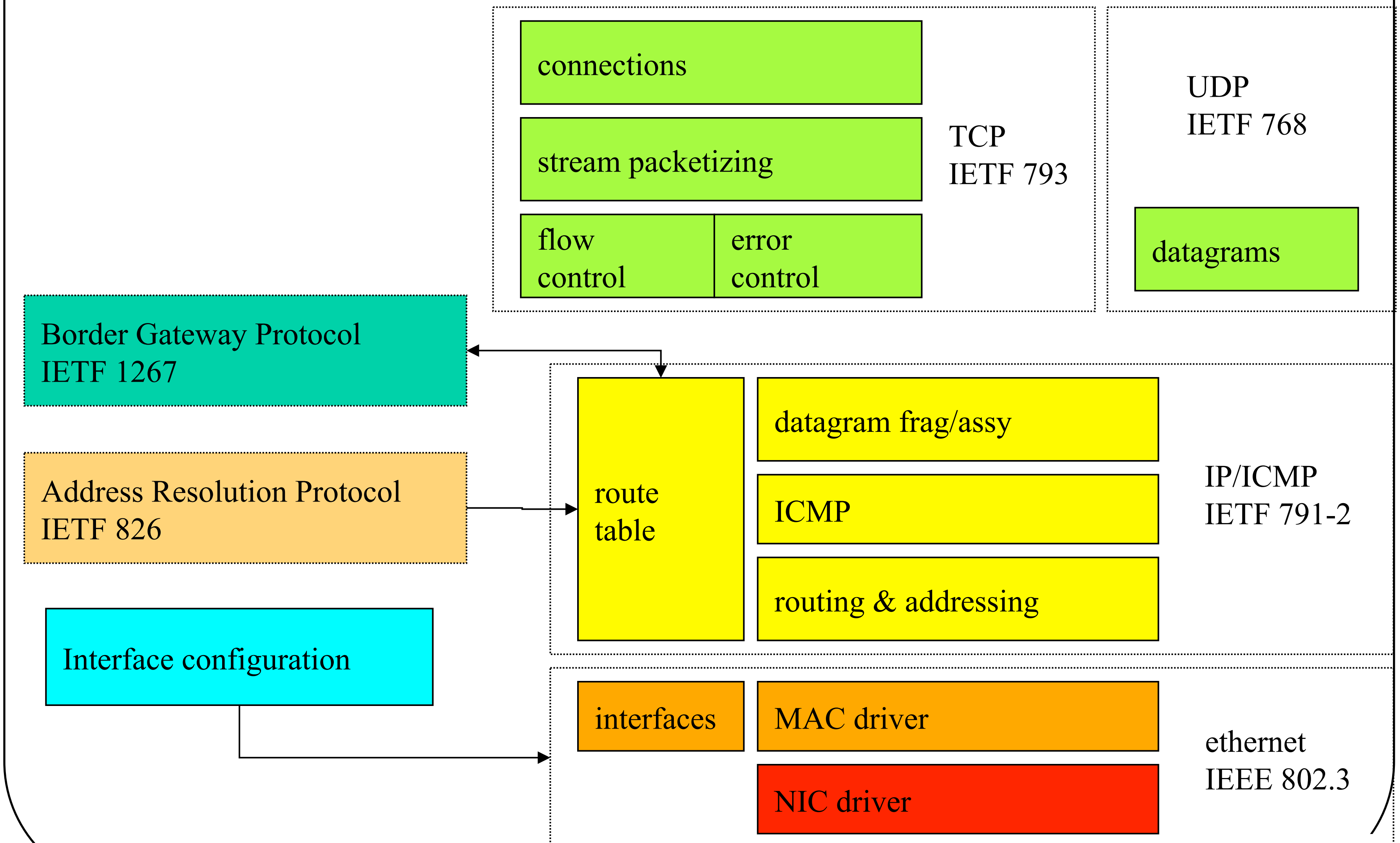
- Basic protocols implemented as OS modules
  - Each protocol implemented in its own module
  - Protocol layering implemented with module plumbing
  - Layering and interconnections are configurable
- User-mode clients attach via IPC-ports
  - Which may map directly to internal networking plumbing
- Advantages
  - Modularity (enables more general layering)
  - Performance (less overhead from entering/leaving kernel)
  - Security (most networking functionality inside the kernel)
- A disadvantage – larger, more complex OS

In-Kernel Protocol Implementations



And off to the packet's destination!

A Basic Ethernet Stack



IPC Implications

- IPC used to be occasionally used for pipes
  - Now it is used for all types of services
    - Demanding richer semantics, and better performance
- Previously connected local processes
  - Now it interconnects agents all over the world
    - Need naming service to register & find partners
    - Must interoperate with other OSes IPC mechanisms
- Used to be simple and fast inside the OS
  - We can no longer depend on shared memory
  - We must be prepared for new modes of failure

Improving Our OS Plumbing

- Protocol stack performance becomes critical
  - To support file access, network servers
- High performance plumbing: UNIX Streams
  - General bi-directional in-kernel communications
    - Can interconnect any two modules in kernel
    - Can be created automatically or manually
  - Message based communication
    - Put (to stream head) and service (queued messages)
    - Accessible via read/write/putmsg/getmsg system calls

Network Protocol Performance

- Layered implementation is flexible and modular
  - But all those layers add overhead
    - Calls, context switches and queuing between layers
    - Potential data recopy at boundary of each layer
  - Protocol stack plumbing must also be high performance
    - High bandwidth, low overhead
- Copies can be avoided by clever data structures
  - Messages can be assembled from multiple buffers
    - Pass buffer pointers rather than copying messages
    - Network adaptor drivers support scatter/gather
- Increasingly more of the protocol stack is in the NIC

Implications of Networking for Operating Systems

- Centralized system management
- Centralized services and servers
- The end of “self-contained” systems
- A new view of architecture
- Performance, scalability, and availability
- The rise of middleware

Centralized System Management

- For all computers in one local network, manage them as a single type of resource
  - Ensure consistent service configuration
  - Eliminate problems with mis-configured clients
- Have all management done across the network
  - To a large extent, in an automated fashion
  - E.g., automatically apply software upgrades to all machines at one time
- Possibly from one central machine
  - For high scale, maybe more distributed

Centralized System Management – Pros and Cons

+ No client-side administration eases management
+ Uniform, ubiquitous services
+ Easier security problems
- Loss of local autonomy
- Screw-ups become ubiquitous
- Increases sysadmin power
- Harder security problems

Centralized Services and Servers

- Networking encourages tendency to move services from all machines to one machine
  - E.g. file servers, web servers, authentication servers
- Other machines can access and use the services remotely
  - So they don't need local versions
  - Or perhaps only simplified local versions
- Includes services that store lots of data

Centralized Services – Pros and Cons

+ Easier to ensure reliability
+ Price/performance advantages
+ Ease of use
- Forces reliance on network
- Potential for huge security and privacy breaches

The End of Self Contained Systems

- Years ago, each computer was nearly totally self-sufficient
- Maybe you got some data or used specialized hardware on some other machine
- But your computer could do almost all of what you wanted to do, on its own
- Now vital services provided over the network
  - Authentication, configuration and control, data storage, remote devices, remote boot, etc.

Non-Self Contained Systems – Pros and Cons

+ Specialized machines may do work better
+ You don't burn local resources on offloaded tasks
+ Getting rid of sysadmin burdens
- Again, forces reliance on network
- Your privacy and security are not entirely under your own control
- Less customization possible

Achieving Performance, Availability, and Scalability

- There used to be an easy answer for these:
  - Moore's law (and its friends)
- The CPUs (and everything else) got faster and cheaper
  - So performance got better
  - More people could afford machines that did particular things
  - Problems too big to solve today fell down when speeds got fast enough

The Old Way Vs. The New Way

- The old way – better components (4-40%/year)
  - Find and optimize all avoidable overhead
  - Get the OS to be as reliable as possible
  - Run on the fastest and newest hardware
- The new way – better systems (1000x)
  - Add more $150 blades and a bigger switch
  - Spreading the work over many nodes is a huge win
    - Performance – may be linear with the number of blades
    - Availability – service continues despite node failures

The New Performance Approach – Pros and Cons

+ Adding independent HW easier than squeezing new improvements out
+ Generally cheaper
- Swaps hard HW design problems for hard SW design problems
- Performance improvements less predictable
- Systems built this way not very well understood

The Rise of Middleware

- Traditionally, there was the OS and your application
  - With little or nothing between them
- Since your application was "obviously" written to run on your OS
- Now, the same application must run on many machines, with different OSes
- Enabled by powerful middleware
  - Which offer execution abstractions at higher levels than the OS
  - Essentially, powerful virtual machines that hide grubby physical machines and their OSes

The OS and Middleware

- Old model – the OS was the platform
  - Applications are written for an operating system
  - OS implements resources to enable applications
- New model – the OS enables the platform
  - Applications are written to a middleware layer
    - E.g., Enterprise Java Beans, Component Object Model, etc.
  - Object management is user-mode and distributed
    - E.g., CORBA, SOAP
  - OS APIs less relevant to applications developers
    - The network is the computer

The Middleware Approach – Pros and Cons

\+ Easy portability

\+ Allows programmers to work with higher level abstractions

– Not always as portable and transparent as one would hope

– Those higher level abstractions impact performance

Networking and Distributed Systems

- Challenges of distributed computing
- Distributed synchronization
- Distributed consensus

What Is Distributed Computing?

- Having more than one computer work cooperatively on some task
- Implies the use of some form of communication
  – Usually networking
- Adding the second computer immensely complicates all problems
  – And adding a third makes it worse

The Big Goal for Distributed Computing

- Total transparency
- Entirely hide the fact that the computation/ service is being offered by a distributed system
- Make it look as if it is running entirely on a single machine
  - Usually the user's own local machine
- Make the remote and distributed appear local and centralized

Challenges of Distributed Computing

- Heterogeneity
  - Different CPUs have different data representation
  - Different OSes have different object semantics and operations
- Intermittent Connectivity
  - Remote resources will not always be available
  - We must recover from failures in mid-computation
  - We must be prepared for conflicts when we reconnect
- Distributed Object Coherence
  - Object management is easy with one in-memory copy
  - How do we ensure multiple hosts agree on state of object?

Deutsch's "Seven Fallacies of Network Computing"

1. The network is reliable
2. There is no latency (instant response time)
3. The available bandwidth is infinite
4. The network is secure
5. The topology of the network does not change
6. There is one administrator for the whole network
7. The cost of transporting additional data is zero

Bottom Line: true transparency is not achievable

Distributed Synchronization

- As we've already seen, synchronization is crucial in proper computer system behavior
- When things don't happen in the required order, we get bad results
- Distributed computing has all the synchronization problems of single machines
- Plus genuinely independent interpreters and memories

Why Is Distributed Synchronization Harder?

- Spatial separation
  - Different processes run on different systems
  - No shared memory for (atomic instruction) locks
  - They are controlled by different operating systems
- Temporal separation
  - Can't "totally order" spatially separated events
  - "Before/simultaneous/after" become fuzzy
- Independent modes of failure
  - One partner can die, while others continue

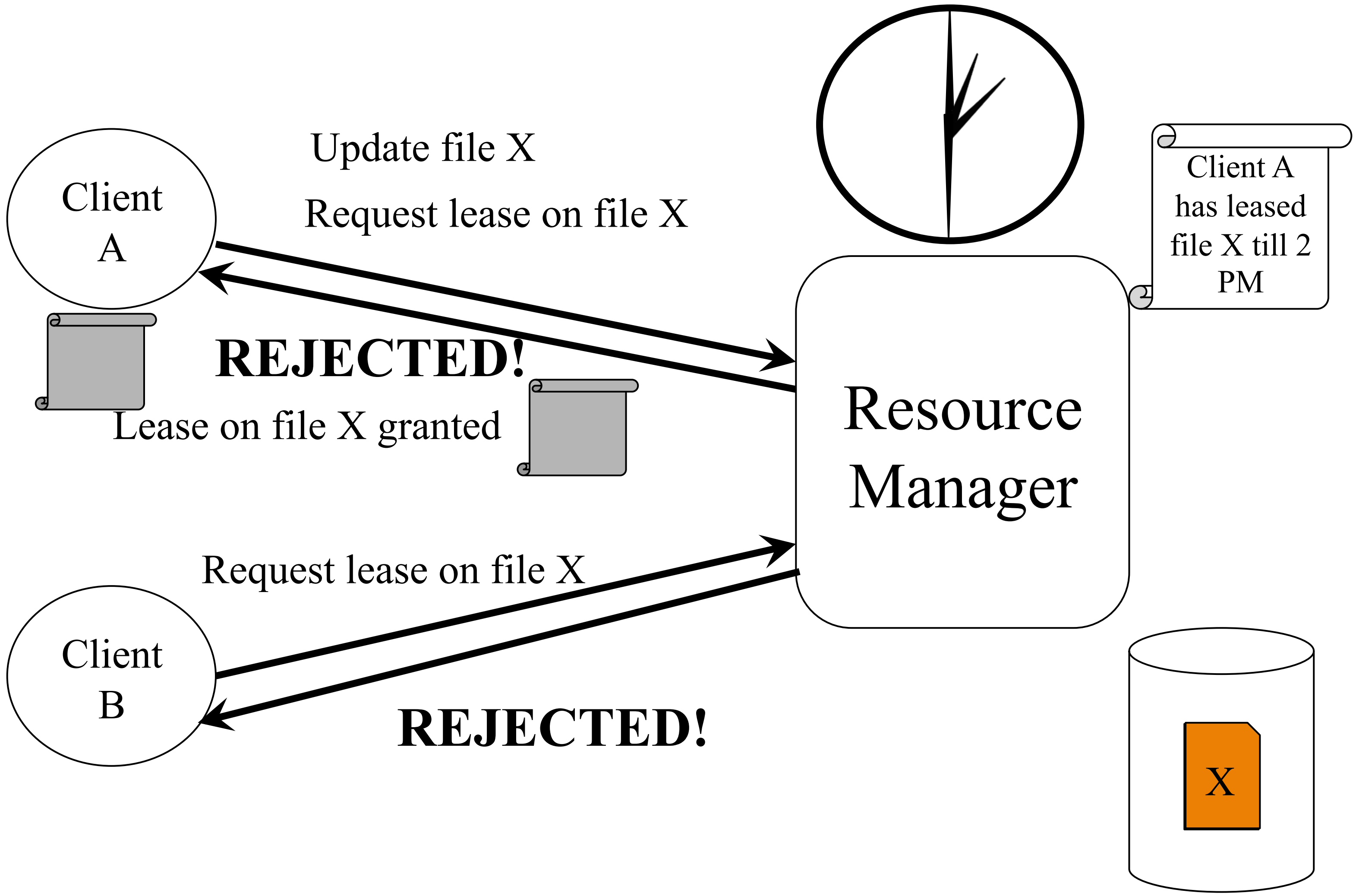
How Do We Manage Distributed Synchronization?

- Distributed analogs to what we do in a single machine
- But they are constrained by the fundamental differences of distributed environments
- They tend to be:
  – Less efficient
  – More fragile and error prone
  – More complex
  – Often all three

Leases

- A relative of locks
- Obtained from an entity that manages a resource
  - Gives client exclusive right to update the file
  - The lease "cookie" must be passed to server with an update
  - Lease can be released at end of critical section
- Only valid for a limited period of time
  - After which the lease cookie expires
    - Updates with stale cookies are not permitted
  - After which new leases can be granted
- Handles a wide range of failures
  - Process, node, network

A Lease Example



What Is This Lease?

- It's essentially a ticket that allows the leasee to do something
  - In our example, update file X
- In other words, it's a bunch of bits
- But proper synchronization requires that only the manager create one
- So it can't be forgeable
- How do we create an unforgeable bunch of bits?

What's Good About Leases?

- The resource manager controls access centrally
  - So we don't need to keep multiple copies of a lock up to date
  - Remember, easiest to synchronize updates to data if only one party can write it
- The manager uses his own clock for leases
  - So we don't need to synchronize clocks
- What if a lease holder dies, losing its lease?
  - No big deal, the lease would expire eventually

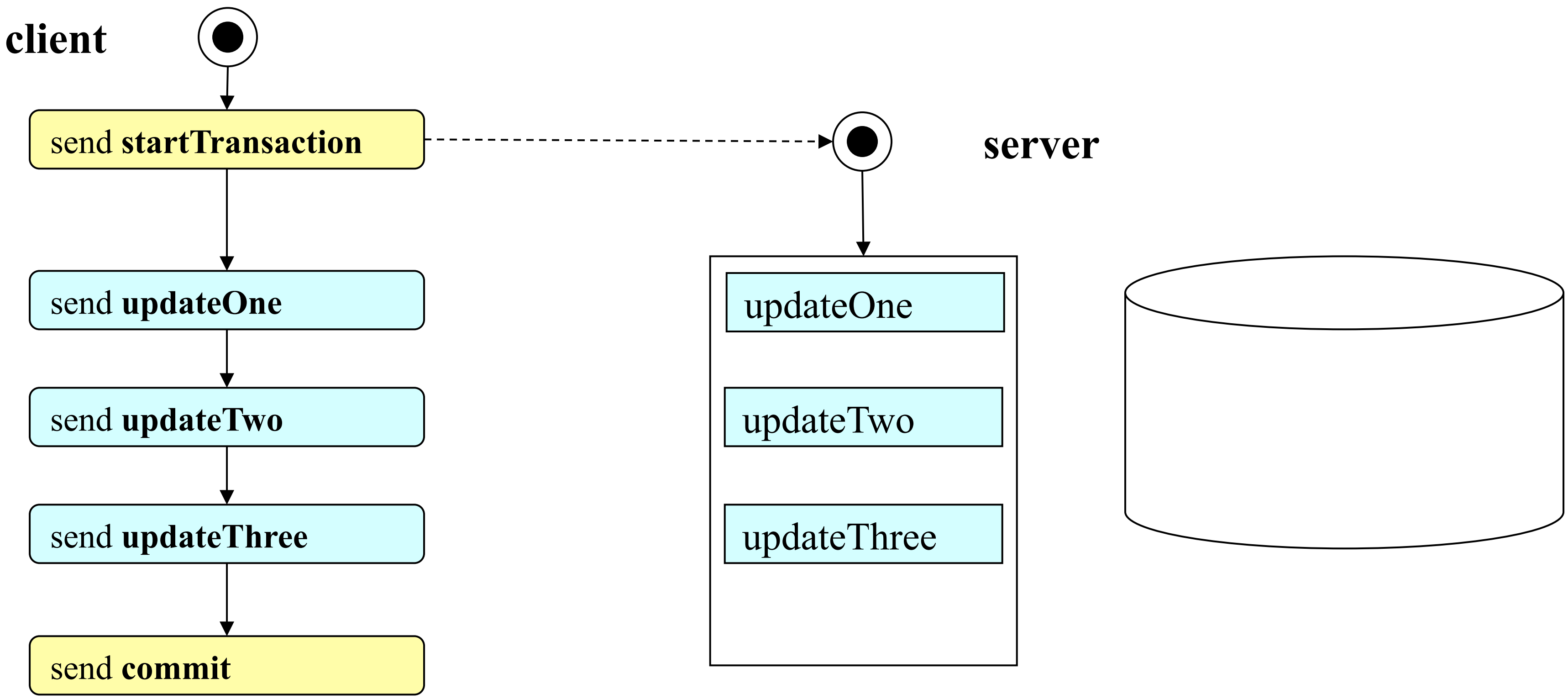
Lock Breaking and Recovery With Leases

- The resource manager can "break" the lock by refusing to honor the lease
  - Could cause bad results for lease holder, so it's undesirable
- Lock is automatically broken when lease expires
- What if lease holder left the resource in a bad state?
- In this case, the resource must be restored to last "good" state
  - Roll back to state prior to the aborted lease
  - Implement all-or-none transactions
  - Implies resource manager must be able to tell if lease holder was "done" with the resource

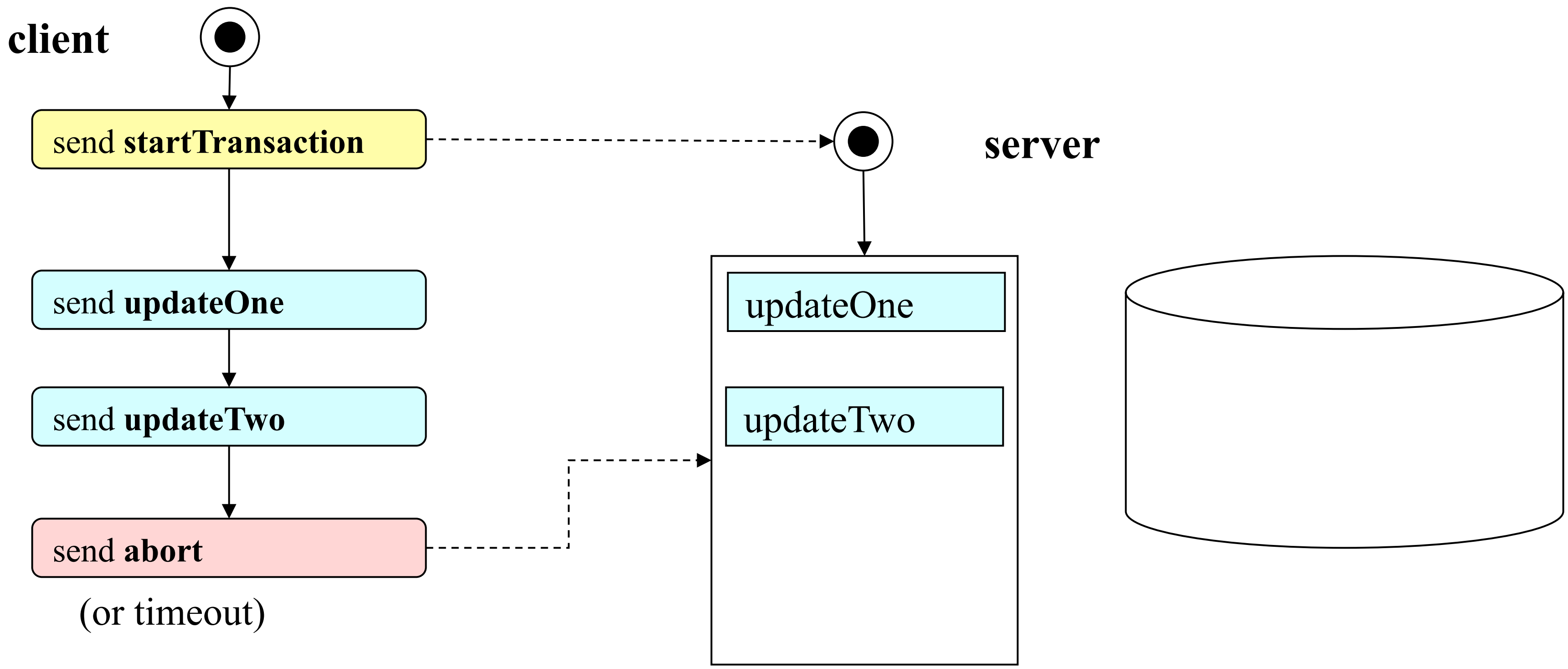
Atomic Transactions

- What if we want guaranteed uninterrupted, all-or-none execution?
- That requires true atomic transactions
- Solves multiple-update race conditions
  - All updates are made part of a transaction
    - Updates are accumulated, but not actually made
  - After all updates are made, transaction is committed
  - Otherwise the transaction is aborted
    - E.g., if client, server, or network fails before the commit
- Resource manager guarantees "all-or-none"
  - Even if it crashes in the middle of the updates

Atomic Transaction Example



What If There’s a Failure?



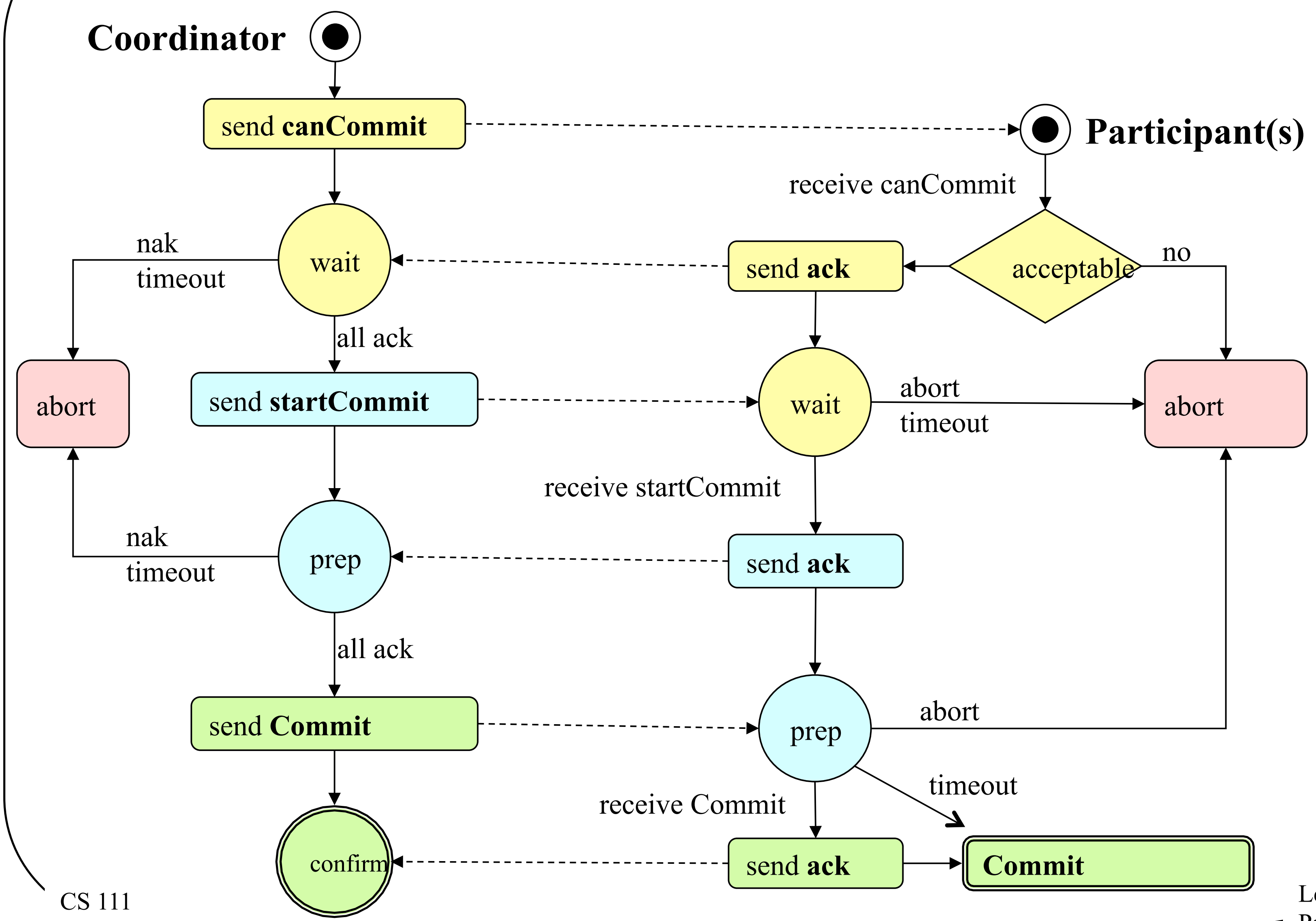
Transactions Spanning Multiple Machines

- That's fine if the data is all on one resource manager
  - Its failure in the middle can be handled by journaling methods
- What if we need to atomically update data on multiple machines?
- How do we achieve the all-or-nothing effect when each machine acts asynchronously?
  - And can fail at any moment?

Commitment Protocols

- Used to implement distributed commitment
  - Provide for atomic all-or-none transactions
  - Simultaneous commitment on multiple hosts
- Challenges
  - Asynchronous conflicts from other hosts
  - Nodes fail in the middle of the commitment process
- Multi-phase commitment protocol:
  - Confirm no conflicts from any participating host
  - All participating hosts are told to prepare for commit
  - All participating hosts are told to "make it so"

Three Phase Commit



Why Three Phases?

- There's a two phase commit protocol, too
- Why two phases to prepare to commit?
  - The first phase asks whether there are conflicts or other problems that would prevent a commitment
  - If problems exist, we won't even attempt commit
  - The second phase is only entered if all nodes agree that commitment is possible
  - It is the actual "prepare to commit"
  - Acknowledgement of which means that all nodes are really ready to commit

Distributed Consensus

- Achieving simultaneous, unanimous agreement
  - Even in the presence of node & network failures
  - Requires agreement, termination, validity, integrity
  - Desired: bounded time
- Consensus algorithms tend to be complex
  - And may take a long time to converge
- So they tend to be used sparingly
  - E.g., use consensus to elect a leader
  - Who makes all subsequent decisions by fiat

A Typical Election Algorithm

1. Each interested member broadcasts his nomination
2. All parties evaluate the received proposals according to a fixed and well known rule
   - E.g., largest ID number wins
3. After a reasonable time for proposals, each voter acknowledges the best proposal it has seen
4. If a proposal has a majority of the votes, the proposing member broadcasts a resolution claim
5. Each party that agrees with the winner's claim acknowledges the announced resolution
6. Election is over when a quorum acknowledges the result

Cluster Membership

- A Cluster is a group of nodes …
  - All of whom are in communication with one another
  - All of whom agree on an elected cluster master
  - All of whom abide by the cluster master's decisions
    - He may (centrally) arbitrate all issues directly
    - He may designate other nodes to make some decisions
- Useful idea because it formalizes set of parties who are working together
- Highly available service clusters
  - Cluster master assigns work to all of the other nodes
  - If a node falls out of the cluster, its work is reassigned

Maintaining Cluster Membership

- Primarily through *heartbeats*
- "I'm still alive" messages, exchanged in cluster
- Cluster master monitors the other nodes
  - Regularly confirm each node is working properly
  - Promptly detect any node falling out of the cluster
  - Promptly reassign work to surviving nodes
- Some nodes must monitor the cluster master
  - To detect the failure of the cluster master
  - To trigger the election of a new cluster master

The Split Brain Problem

- What if the participating nodes are partitioned?
- One set can talk to each other, and another set can also
  - But the two sets can't exchange messages
- We then have two separate clusters providing the same service
  - Which can lead to big problems, depending on the situation

Quorums

- The simplest solution to the split-brain problem is to require a *quorum*
  - In a cluster that has been provisioned for N nodes, becoming the cluster master requires (N/2)+1 votes
  - This completely prevents split-brain
    - It also prevents recovering from the loss of N/2 nodes
- Some systems use a "quorum device"
  - E.g., a shared (multi-ported) disk
    - Cluster master must be able to reserve/lock this device
    - Device won't allow simultaneous locking by two different nodes
  - Failure of this device takes down whole system
- Some systems use special election hardware

Conclusion

- Networking has become a vital service for most machines
- The operating system is increasingly involved in networking
  - From providing mere access to a network device
  - To supporting sophisticated distributed systems
- An increasing trend
- Future OSes might be primarily all about networking