Distributed Systems
CS 111
Operating Systems
Peter Reiher

Outline

- Goals and vision of distributed computing
- Basic architectures
  - Symmetric multiprocessors
  - Single system image distributed systems
  - Cloud computing systems
  - User-level distributed computing
- Distributed file systems

Important Characteristics of Distributed Systems

- Performance
  - Overhead, scalability, availability
- Functionality
  - Adequacy and abstraction for target applications
- Transparency
  - Compatibility with previous platforms
  - Scope and degree of location independence
- Degree of coupling
  - How many things do distinct systems agree on?
  - How is that agreement achieved?

Types of Transparency

- Network transparency
  - Is the user aware he's going across a network?
- Name transparency
  - Does remote use require a different name/kind of name for a file than a local user?
- Location transparency
  - Does the name change if the file location changes?
- Performance transparency
  - Is remote access as quick as local access?

Loosely and Tightly Coupled Systems

- Tightly coupled systems
  - Share a global pool of resources
  - Agree on their state, coordinate their actions
- Loosely coupled systems
  - Have independent resources
  - Only coordinate actions in special circumstances
- Degree of coupling
  - Tight coupling: global coherent view, seamless fail-over
    - But very difficult to do right
  - Loose coupling: simple and highly scalable
    - But a less pleasant system model

Globally Coherent Views

- Everyone sees the same thing
- Usually the case on single machines
- Harder to achieve in distributed systems
- How to achieve it?
  - Have only one copy of things that need single view
    - Limits the benefits of the distributed system
    - And exaggerates some of their costs
  - Ensure multiple copies are consistent
    - Requiring complex and expensive consensus protocols
- Not much of a choice

Major Classes of Distributed Systems

- Symmetric Multi-Processors (SMP)
  – Multiple CPUs, sharing memory and I/O devices
- Single-System Image (SSI) & Cluster Computing
  – A group of computers, acting like a single computer
- Loosely coupled, horizontally scalable systems
  – Coordinated, but relatively independent systems
  – Cloud computing is the most widely used version
- Application level distributed computing
  – Application level protocols
  – Distributed middle-ware platforms

Symmetric Multiprocessors (SMP)

- What are they and what are their goals?
- OS design for SMP systems
- SMP parallelism
  - The memory bandwidth problem

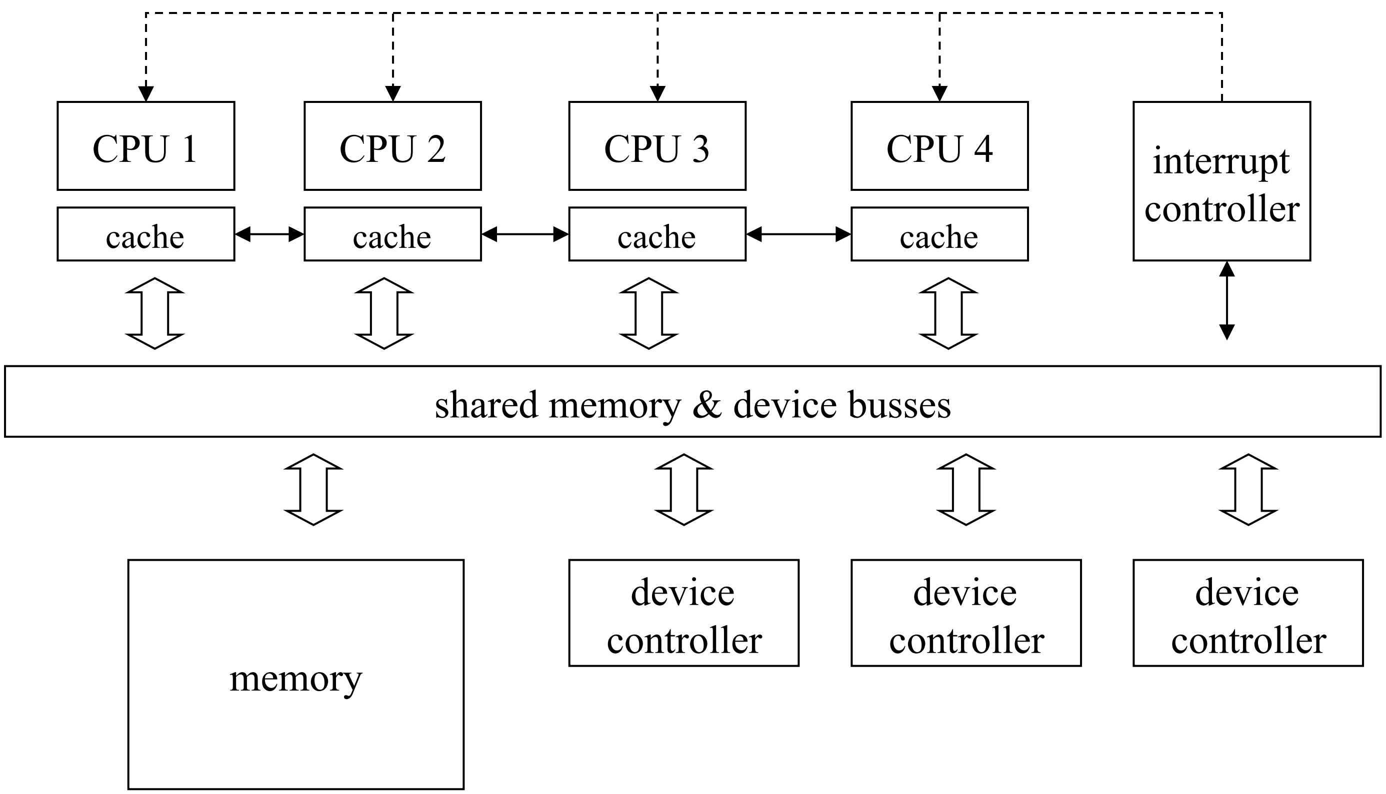
SMP Systems

- Computers composed of multiple identical compute engines
  - Each computer in SMP system usually called a node
- Sharing memories and devices
- Could run same or different code on all nodes
  - Each node runs at its own pace
  - Though resource contention can cause nodes to block
- Examples:
  - BBN Butterfly parallel processor
  - More recently, multi-way Intel servers

SMP Goals

- Price performance
  - Lower price per MIP than single machine
  - Since much of machine is shared
- Scalability
  - Economical way to build huge systems
  - Possibility of increasing machine's power just by adding more nodes
- Perfect application transparency
  - Runs the same on 16 nodes as on one
  - Except faster

A Typical SMP Architecture



SMP Operating Systems

- One processor boots with power on
  - It controls the starting of all other processors
- Same OS code runs in all processors
  - One physical copy in memory, shared by all CPUs
- Each CPU has its own registers, cache, MMU
  - They cooperatively share memory and devices
- ALL kernel operations must be Multi-Thread-Safe
  - Protected by appropriate locks/semaphores
  - Very fine grained locking to avoid contention

SMP Parallelism

- Scheduling and load sharing
  - Each CPU can be running a different process
  - Just take the next ready process off the run-queue
  - Processes run in parallel
  - Most processes don't interact (other than inside kernel)
    - If they do, poor performance caused by excessive synchronization
- Serialization
  - Mutual exclusion achieved by locks in shared memory
  - Locks can be maintained with atomic instructions
  - Spin locks acceptable for VERY short critical sections
  - If a process blocks, that CPU finds next ready process

The Challenge of SMP Performance

- Scalability depends on memory contention
  - Memory bandwidth is limited, can't handle all CPUs
  - Most references better be satisfied from per-CPU cache
  - If too many requests go to memory, CPUs slow down
- Scalability depends on lock contention
  - Waiting for spin-locks wastes time
  - Context switches waiting for kernel locks waste time
- This contention wastes cycles, reduces throughput
  - 2 CPUs might deliver only 1.9x performance
  - 3 CPUs might deliver only 2.7x performance

Managing Memory Contention

- Each processor has its own cache
  - Cache reads don’t cause memory contention
  - Writes are more problematic
- Locality of reference often solves the problems
  - Different processes write to different places
- Keeping everything coherent still requires a smart memory controller
- Fast n-way memory controllers are <u>very</u> expensive
  - Without them, memory contention taxes performance
  - Cost/complexity limits how many CPUs we can add

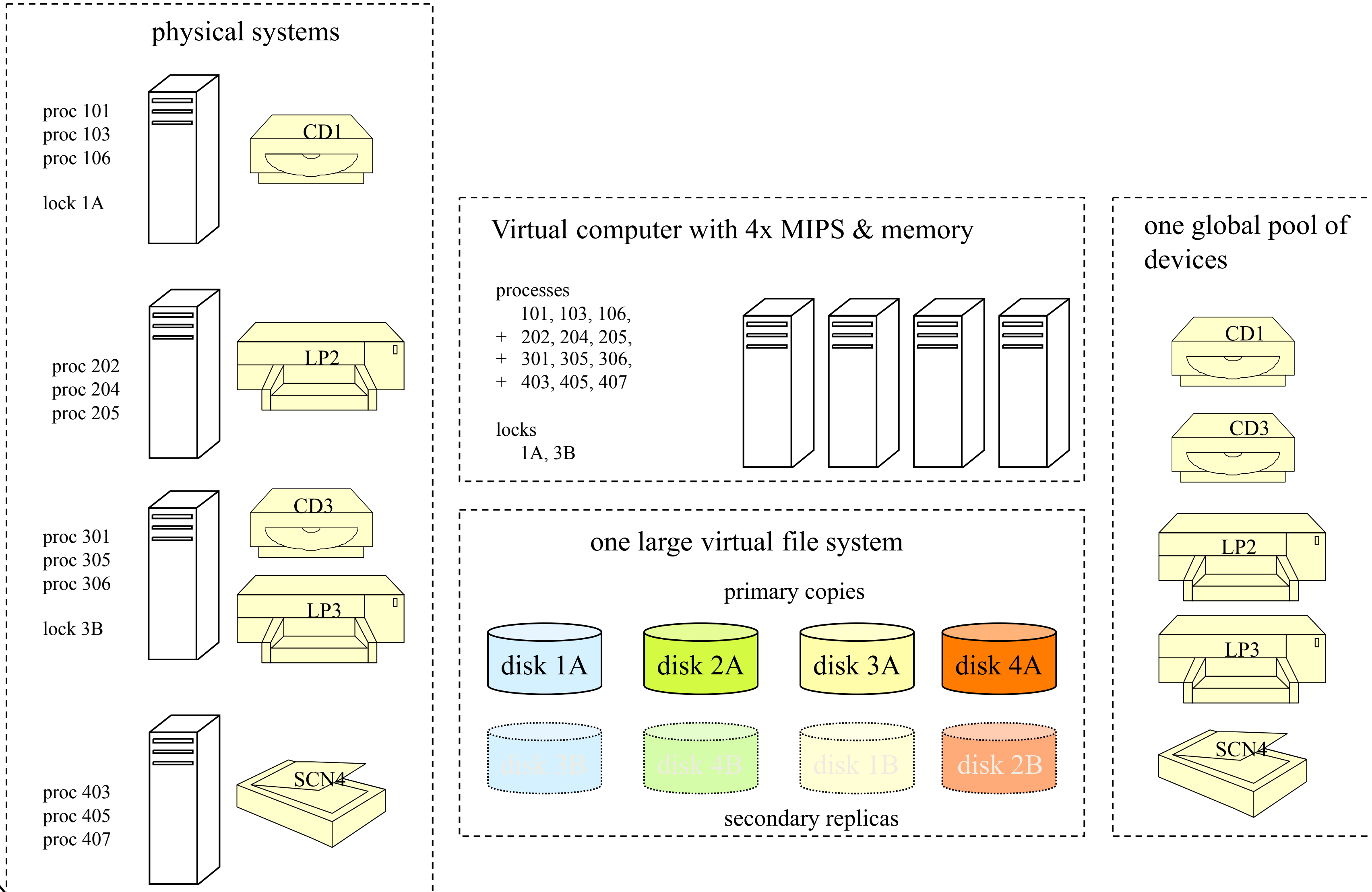
Single System Image Approaches

- Built a distributed system out of many more-or-less traditional computers
  - Each with typical independent resources
  - Each running its own copy of the same OS
  - Usually a fixed, known pool of machines
- Connect them with a good local area network
- Use software techniques to allow them to work cooperatively
  - Often while still offering many benefits of independent machines to the local users

Motivations for Single System Image Computing

- High availability, service survives node/link failures
- Scalable capacity (overcome SMP contention problems)
  - You're connecting with a LAN, not a special hardware switch
  - LANs can host hundreds of nodes
- Good application transparency
- Examples:
  - Locus, Sun Clusters, MicroSoft Wolf-Pack, OpenSSI
  - Enterprise database servers

The SSI Vision



OS Design for SSI Clusters

- All nodes agree on the state of all OS resources
  - File systems, processes, devices, locks, IPC ports
  - Any process can operate on any object, transparently
- They achieve this by exchanging messages
  - Advising one another of all changes to resources
    - Each OS's internal state mirrors the global state
  - To execute node-specific requests
    - Node-specific requests automatically forwarded to right node
- The implementation is large, complex, and difficult
- The exchange of messages can be very expensive

SSI Performance

- Clever implementation can reduce overhead
  - But 10-20% overhead is common, can be much worse
- Complete transparency
  - Even very complex applications "just work"
  - They do not have to be made "network aware"
- Good robustness
  - When one node fails, others notice and take-over
  - Often, applications won't even notice the failure
  - Each node hardware-independent
    - Failures of one node don't affect others, unlike some SMP failures
- Very nice for application developers and customers
  - But they are complex, and not particularly scalable

An Example of SSI Complexity

- Keeping track of which nodes are up
- Done in the Locus Operating System through "topology change"
- Need to ensure that all nodes know of the identity of all nodes that are up
- By running a process to figure it out
- Complications:
  - Who runs the process? What if he's down himself?
  - Who do they tell the results to?
  - What happens if things change while you're running it?
  - What if the system is partitioned?

Is It Really That Bad?

- Nodes fail and recovery rarely
- So something like topology change doesn't run that often
- But consider a more common situation
- Two processes have the same file open
  - What if they're on different machines?
  - What if they are parent and child, and share a file pointer?
- Basic read operations require distributed agreement
  - Or, alternately, we compromise the single image
  - Which was the whole point of the architecture

Scaling and SSI

- Scaling limits proved not to be hardware driven
  - Unlike SMP machines
- Instead, driven by algorithm complexity
  - Consensus algorithms, for example
- Design philosophy essentially requires distributed cooperation
  - So this factor limits scalability

Lessons Learned From SSI

- Consensus protocols are expensive
  - They converge slowly and scale poorly
- Systems have a great many resources
  - Resource change notifications are expensive
- Location transparency encouraged non-locality
  - Remote resource use is much more expensive
- A very complicated operating system design
  - Distributed objects are much more complex to manage
  - Complex optimizations to reduce the added overheads
  - New modes of failure with complex recovery procedures

Loosely Coupled Systems

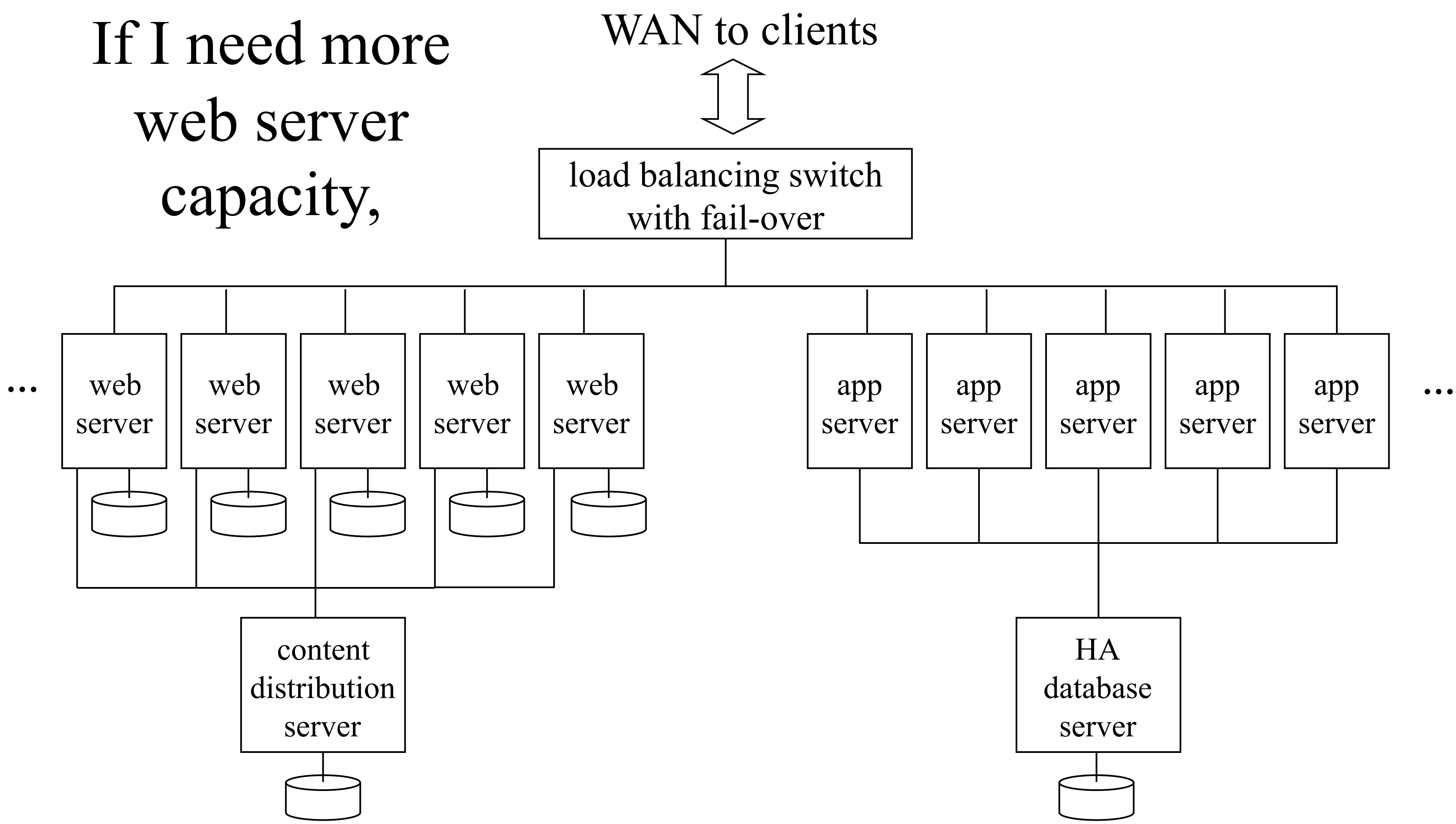
- Characterization:
  - A parallel group of independent computers
  - Serving similar but independent requests
  - Minimal coordination and cooperation required
- Motivation:
  - Scalability and price performance
  - Availability – if protocol permits stateless servers
  - Ease of management, reconfigurable capacity
- Examples:
  - Web servers, app servers, cloud computing

Horizontal Scalability

- Each node largely independent
- So you can add capacity just by adding a node "on the side"
- Scalability can be limited by network, instead of hardware or algorithms
  - Or, perhaps, by a load balancer
- Reliability is high
  - Failure of one of N nodes just reduces capacity

Horizontal Scalability Architecture

If I need more web server capacity,



Elements of Loosely Coupled Architecture

- Farm of independent servers
  - Servers run same software, serve different requests
  - May share a common back-end database
- Front-end switch
  - Distributes incoming requests among available servers
  - Can do both load balancing and fail-over
- Service protocol
  - Stateless servers and idempotent operations
  - Successive requests may be sent to different servers

Horizontally Scaled Performance

- Individual servers are very inexpensive
  - Blade servers may be only $100-$200 each
- Scalability is excellent
  - 100 servers deliver approximately 100x performance
- Service availability is excellent
  - Front-end automatically bypasses failed servers
  - Stateless servers and client retries fail-over easily
- The challenge is managing thousands of servers
  - Automated installation, global configuration services
  - Self monitoring, self-healing systems
  - Scaling limited by management, not HW or algorithms

What About the Centralized Resources?

- The load balancer appears to be centralized
- And what about the back-end databases?
- Are these single points of failure for this architecture?
- And also limits on performance?
- Yes, but . . .

Handling the Limiting Factors

- The centralized pieces can be special hardware
  - There are very few of them
  - So they can use aggressive hardware redundancy
    - Expensive, but only for a limited set of machines
  - They can also be high performance machines
- Some of them have very simple functionality
  - Like the load balancer
- With proper design, their roles can be minimized, decreasing performance problems

Cloud Computing

- The most recent twist on distributed computing
- Set up a large number of machines all identically configured
- Connect them to a high speed LAN
  - And to the Internet
- Accept arbitrary jobs from remote users
- Run each job on one or more nodes
- Entire facility probably running mix of single machine and distributed jobs, simultaneously

Distributed Computing and Cloud Computing

- In one sense, these are orthogonal
- Each job submitted to a cloud might or might not be distributed
- Many of the hard problems of the distributed jobs are the user's problem, not the system's
  - E.g., proper synchronization and locking
- But the cloud facility must make communications easy

What Runs in a Cloud?

- In principle, anything
- But general distributed computing is hard
- So much of the work is run using special tools
- These tools support particular kinds of parallel/ distributed processing
- Either embarrassingly parallel jobs
- Or those using a method like map-reduce
- Things where the user need not be a distributed systems expert

Embarrassingly Parallel Jobs

- Problems where it's really, really easy to parallelize them
- Probably because the data sets are easily divisible
- And exactly the same things are done on each piece
- So you just parcel them out among the nodes and let each go independently
- Everyone finishes at more or less same time

The Most Embarrassing of Embarrassingly Parallel Jobs

- Say you have a large computation
- You need to perform it N times, with slightly different inputs each time
- Each iteration is expected to take the same time
- If you have N cloud machines, write a script to send one of the N jobs to each
- You get something like N times speedup

MapReduce

- Perhaps the most common cloud computing software tool/technique
- A method of dividing large problems into compartmentalized pieces
- Each of which can be performed on a separate node
- With an eventual combined set of results

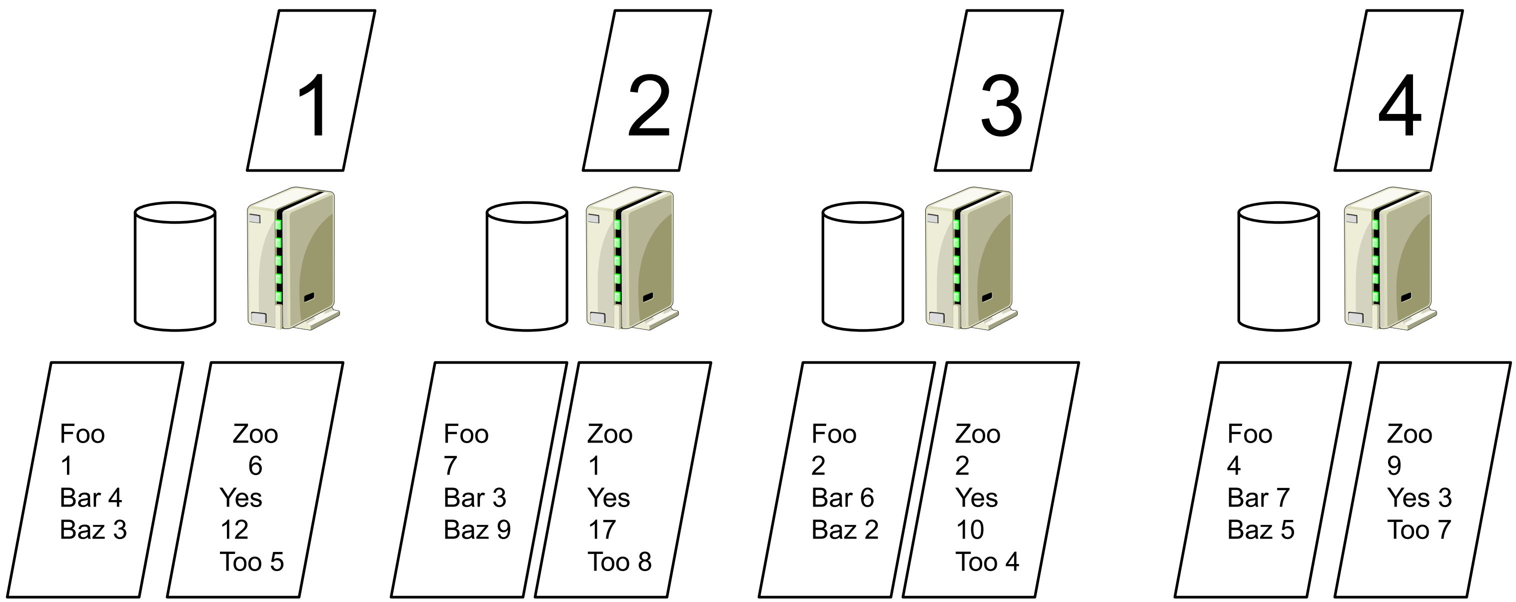
The Idea Behind MapReduce

- There is a single function you want to perform on a lot of data
  - Such as searching it for a string
- Divide the data into disjoint pieces
- Perform the function on each piece on a separate node (*map*)
- Combine the results to obtain output (*reduce*)

An Example

- We have 64 megabytes of text data
- We want to count how many times each word occurs in the text
- Divide it into 4 chunks of 16 Mbytes
- Assign each chunk to one processor
- Perform the map function of "count words" on each

The Example Continued

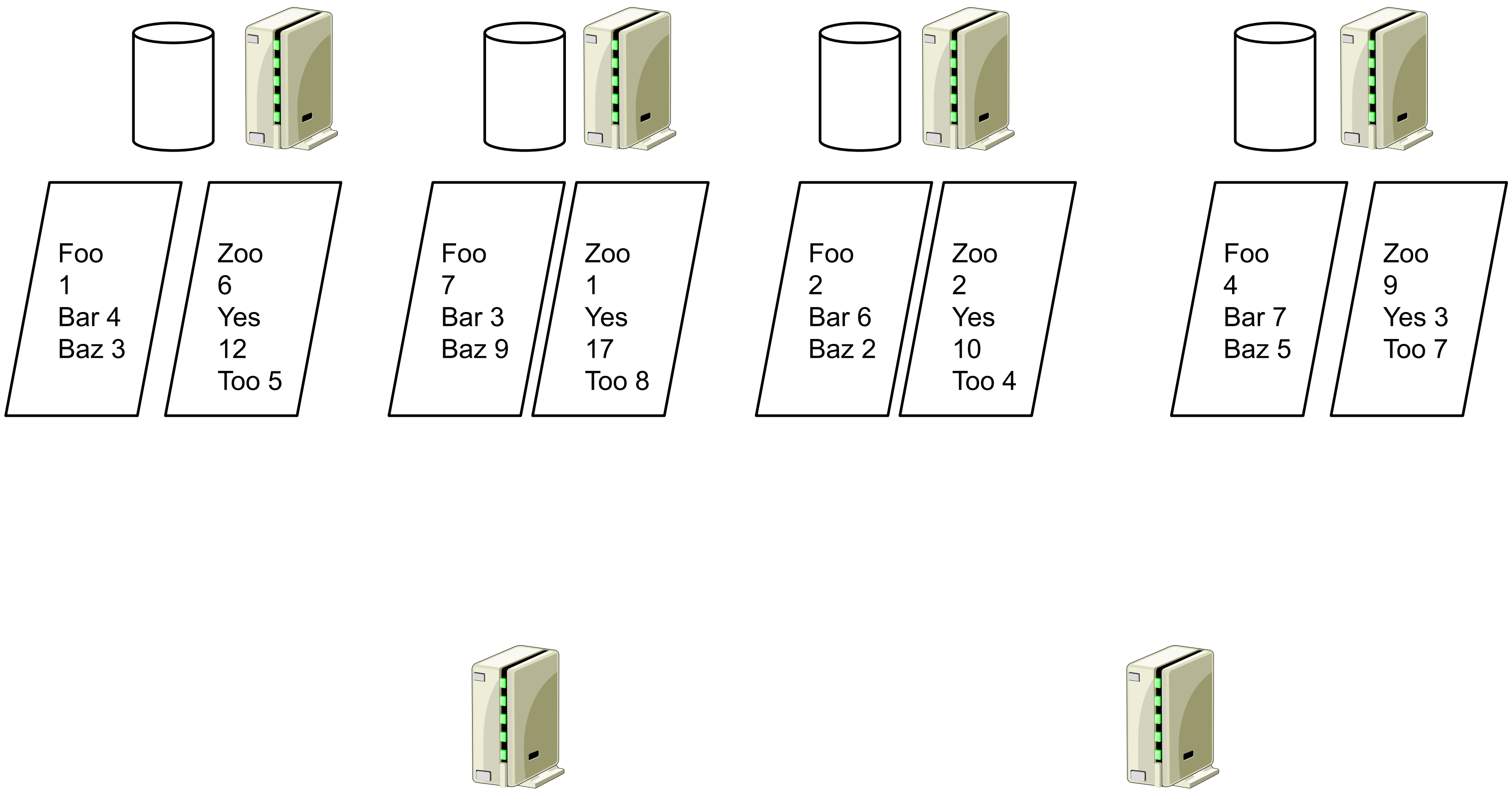


That's the map stage

On To Reduce

- We might have two more nodes assigned to doing the reduce operation
- They will each receive a share of data from a map node
- The reduce node performs a reduce operation to “combine” the shares
- Outputting its own result

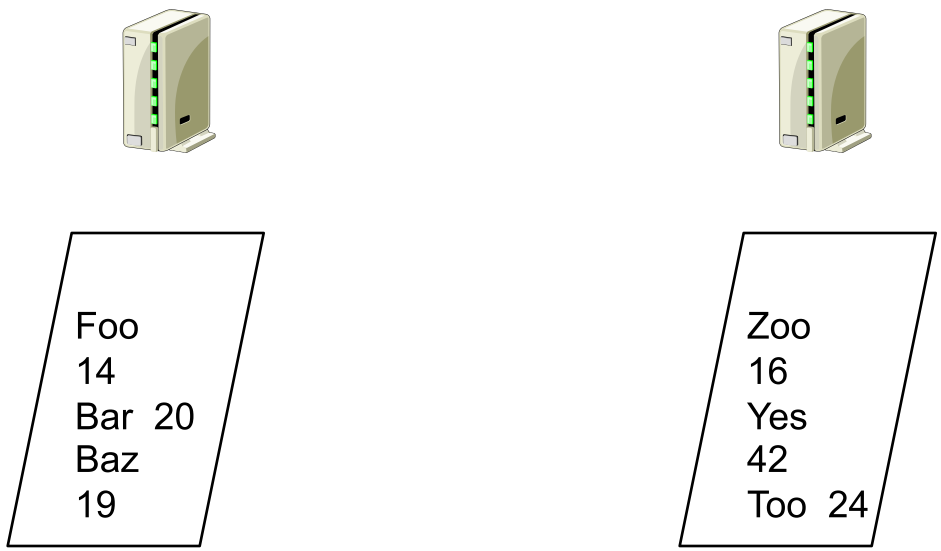
Continuing the Example



The Reduce Nodes Do Their Job

Write out the results to files

And MapReduce is done!



But I Wanted A Combined List

- No problem
- Run another (slightly different) MapReduce on the outputs
- Have one reduce node that combines everything

Synchronization in MapReduce

- Each map node produces an output file for each reduce node
- It is produced atomically
- The reduce node can’t work on this data until the whole file is written
- Forcing a synchronization point between the map and reduce phases

Distributed File Systems: Goals and Challenges

- Sometimes the files we want aren't on our machine
- We'd like to be able to access them anyway
- How do we provide access to remote files?

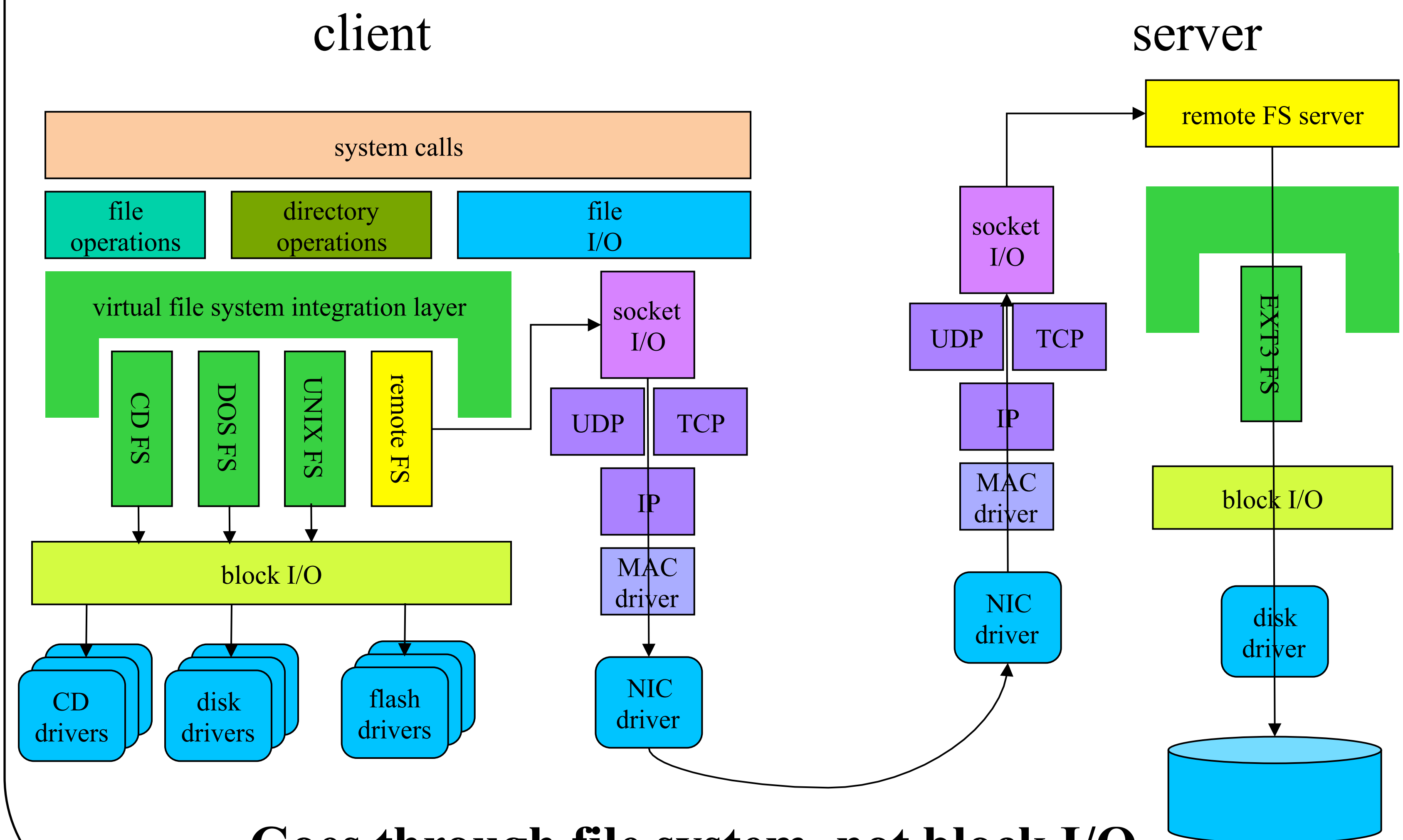
Key Characteristics of Network File System Solutions

- APIs and transparency
  - How do users and processes access remote files?
  - How closely do remote files mimic local files?
- Performance and robustness
  - Are remote files as fast and reliable as local ones?
- Architecture
  - How is solution integrated into clients and servers?
- Protocol and work partitioning
  - How do client and server cooperate?

Remote File Access Protocols

- Goal: complete transparency
  - Normal file system calls work on remote files
  - Support file sharing by multiple clients
  - High performance, availability, reliability, scalability
- Typical Architecture
  - Uses plug-in file system architecture
  - Client-side file system is merely a local proxy
  - Translates file operations into network requests
  - Server-side daemon receives/process requests
  - Translates them into real file system operations

Remote File Access Architecture



Goes through file system, not block I/O

The Client Side

- On Unix/Linux, makes use of VFS interface
- Allows plug-in of file system implementations
  - Each implements a set of basic methods
    - create, delete, open, close, link, unlink, etc.
  - Translates logical operations into disk operations
- Remote file systems can also be implemented
  - Translate each standard method into messages
  - Forward those requests to a remote file server
  - RFS client only knows the RFS protocol
    - Need not know the underlying on-disk implementation

Server Side Implementation

- Remote file system server daemon
  - Receives and decodes messages
  - Does requested operations on local file system
- Can be implemented in user- or kernel-mode
  - Kernel daemon may offer better performance
  - User-mode is much easier to implement
- One daemon may serve all incoming requests
  - Higher performance, fewer context switches
- Or could be many per-user-session daemons
  - Simpler, and probably more secure

Remote File Access: Problems and Solutions

- Authentication and authorization
- Synchronization
- Performance
- Robustness

Performance Issues

- Performance of the remote file system now dependent on many more factors
  - Not just the local CPU, bus, memory, and disk
- Also on the same hardware on the server that stores the files
  - Which often is servicing many clients
- And on the network in between
  - Which can have wide or narrow bandwidth

Some Performance Solutions

- Appropriate transport and session protocols
  - Minimize messages, maximize throughput
- Partition the work
  - Minimize number of remote requests
  - Spread load over more processors and disks
- Client-side pre-fetching and caching
  - Fetching whole file at a once is more efficient
  - Block caching for read-ahead and deferred writes
  - Reduces disk and network I/O (vs. server cache)
  - Cache consistency can be a problem

Robustness Issues

- Three major components in remote file system operations
  - The client machine
  - The server machine
  - The network in between
- All can fail
  - Leading to potential problems for the remote file system's data and users

Robustness Solution Approaches

- Network errors – support client retries
  - Have file system protocol uses *idempotent* requests
  - Have protocol support all-or-none transactions
- Client failures – support server-side recovery
  - Automatic back-out of uncommitted transactions
  - Automatic expiration of timed out lock leases
- Server failures – support server fail-over
  - Replicated (parallel or back-up) servers
  - Stateless remote file system protocols
  - Automatic client-server rebinding

The Network File System (NFS)

- Transparent, heterogeneous file system sharing
  - Local and remote files are indistinguishable
- Peer-to-peer and client-server sharing
  - Disk-full clients can export file systems to others
  - Able to support diskless (or dataless) clients
  - Minimal client-side administration
- High efficiency and high availability
  - Read performance competitive with local disks
  - Scalable to huge numbers of clients
  - Seamless fail-over for all readers and some writers

The NFS Protocol

- Relies on idempotent operations and stateless server
  - Built on top of a remote procedure call protocol
  - With eXternal Data Representation, server binding
  - Versions of RPC over both TCP or UDP
  - Optional encryption (may be provided at lower level)
- Scope – basic file operations only
  - Lookup (open), read, write, read-directory, stat
  - Supports client or server-side authentication
  - Supports client-side caching of file contents
  - Locking and auto-mounting done with another protocol

NFS and Updates

- An NFS server does not prevent conflicting updates
  - As with local file systems, this is the applications' job
- Auxiliary server/protocol for file and record locking
  - All leases are maintained on the lock server
  - All lock/unlock operations handed by lock server
- Client/network failure handling
  - Server can break locks if client dies or times out
  - "Stale-handle" errors inform client of broken lock
  - Client response to these errors are application specific
- Lock server failure handling is very complex

Distributed Systems - Summary

- Different distributed system models support:
  - Different degrees of transparency
    - Do applications see a network or single system image?
  - Different degrees of coupling
    - Making multiple computers cooperate is difficult
    - Doing it without shared memory is even worse
- Distributed systems always face a trade-off between performance, independence, and robustness
  - Cooperating redundant nodes offer higher availability
  - Communication and coordination are expensive
  - Mutual dependency creates more modes of failure