Distributed Computing
CS 111
Operating System Principles
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

Goals of Distributed Computing

- Better services
  - Scalability
    - Some applications require more resources than one computer has
    - Should be able to grow system capacity to meet growing demand
  - Availability
    - Disks, computers, and software fail, but services should be 24x7!
  - Improved ease of use, with reduced operating expenses
    - Ensuring correct configuration of all services on all systems
- New services
  - Applications that span multiple system boundaries
  - Global resource domains, services decoupled from systems
  - Complete location transparency

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?

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)

- A solution relying on special hardware support
  - Which has pluses and minuses
- Primarily for parallel processing
- Core parallelism problem for SMP:
  - The memory bandwidth problem

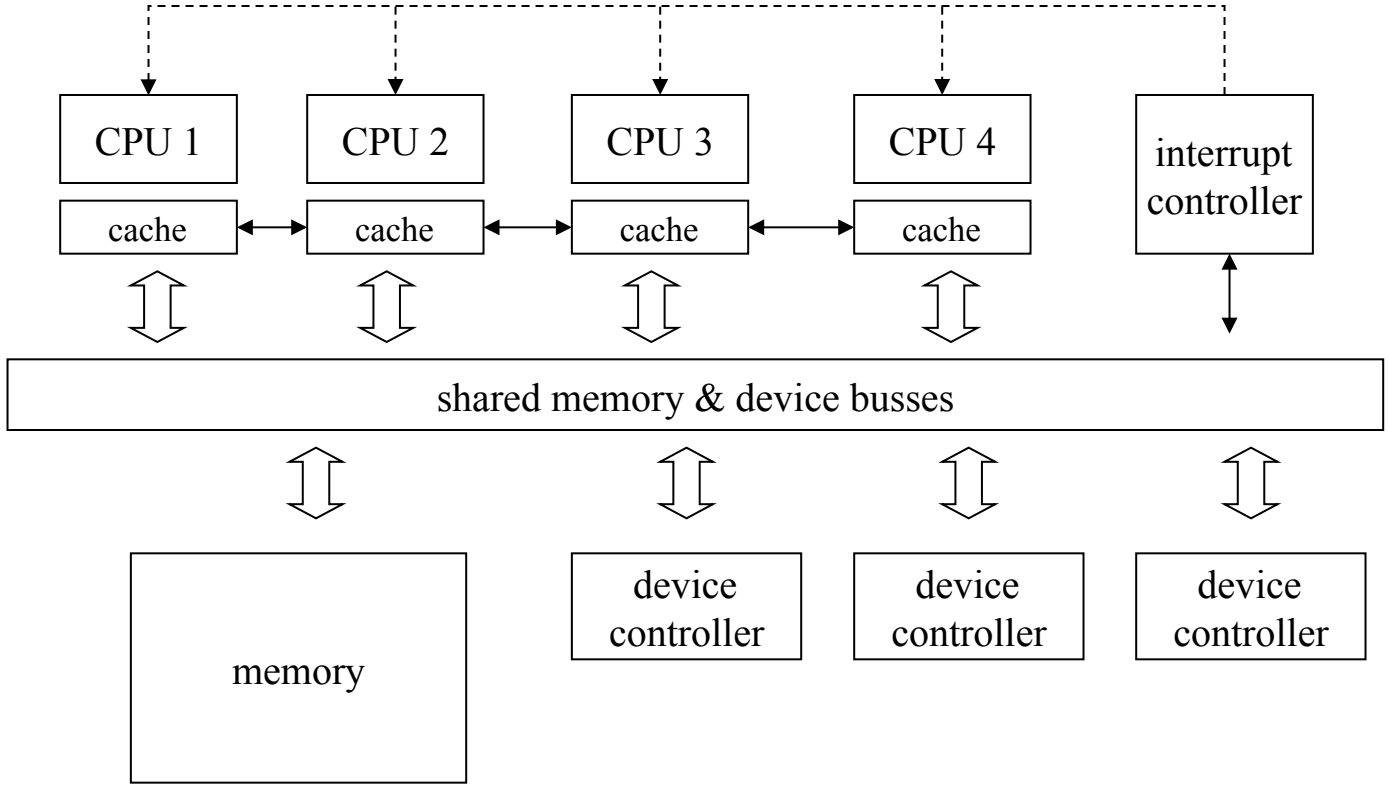
SMP Systems

- 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
  - Multi-way Intel servers
  - To some extent, modern multicore processors

SMP Goals

- Price performance
  - Lower price per MIP than single machine
- 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



The SMP Price/Performance Argument

- A computer is much more than a CPU
  - Mother-board, disks, controllers, power supplies, case
  - CPU might cost 10-15% of the cost of the computer
- Adding CPUs to a computer is very cost-effective
  - A second CPU yields cost of 1.1x, performance 1.9x
  - A third CPU yields cost of 1.2x, performance 2.7x
- Same argument also applies at the chip level
  - Making a machine twice as fast is ever more difficult
  - Adding more cores to the chip gets ever easier
- Massive multi-processors are an obvious direction

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

Handling Kernel Synchronization

- Multiple processors are sharing one OS copy
- What needs to be synchronized?
  - Every potentially sharable OS data structure
    - Process descriptors, file descriptors, data buffers, message queues, etc.
    - All of the devices
- Could we just lock the entire kernel, instead?
  - Yes, but it would be a bottleneck
  - Remember lock contention?
  - Avoidable by not using coarse-grained locking

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

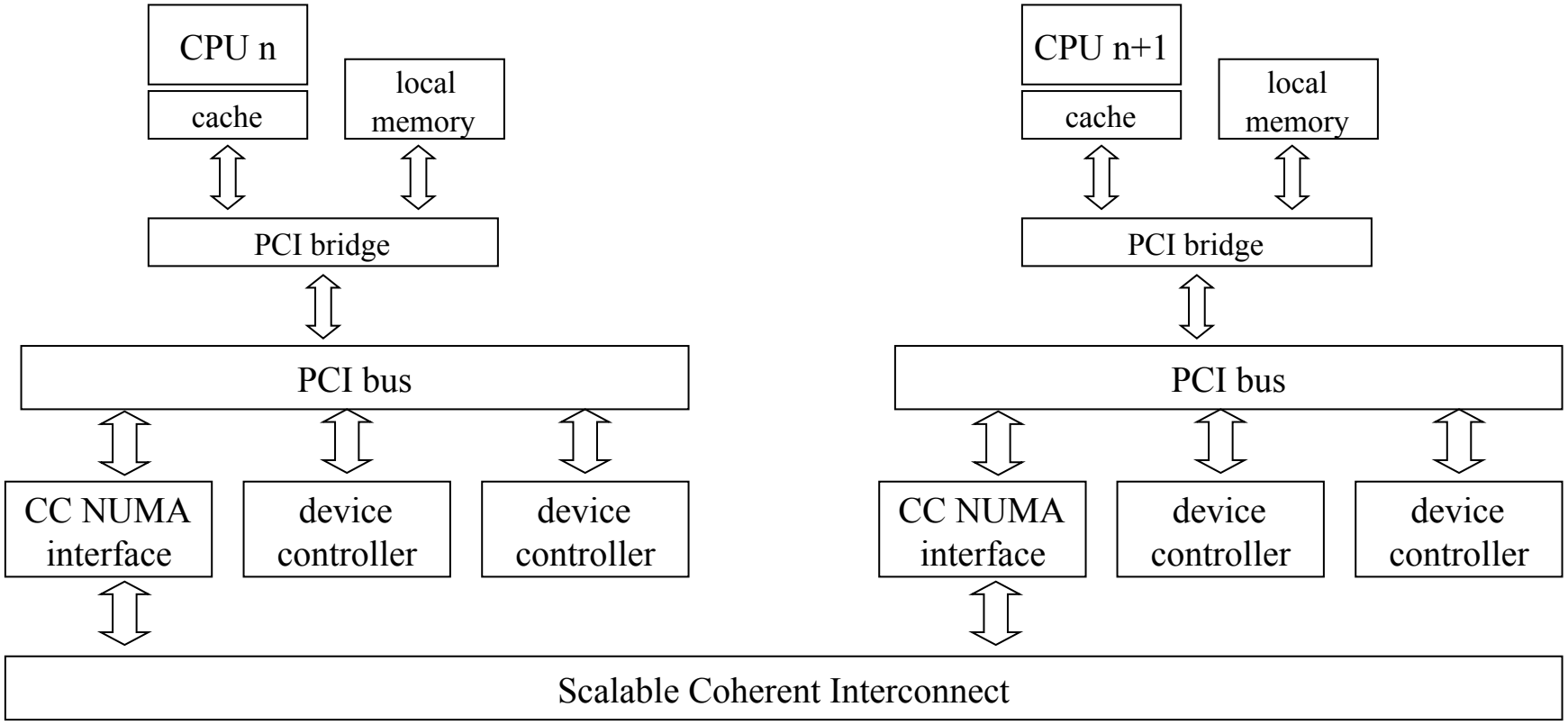
Managing Memory Contention

- 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 very expensive
  - Without them, memory contention taxes performance
  - Cost/complexity limits how many CPUs we can add

NUMA

- Non-Uniform Memory Architectures
- Another approach to handling memory in SMPs
- Each CPU gets its own memory, which is on the bus
  - Each CPU has fast path to its own memory
- Connected by a Scalable Coherent Interconnect
  - A very fast, very local network between memories
  - Accessing memory over the SCI may be 3-20x slower
- These interconnects can be highly scalable

A Sample NUMA SMP Architecture



OS Design for NUMA Systems

- All about local memory hit rates
  - Each processor must use local memory almost exclusively
  - Every outside reference costs us 3-20x performance
  - We need 75-95% hit rate just to break even
- How can the OS ensure high hit-rates?
  - Replicate shared code pages in each CPU's memory
  - Assign processes to CPUs, allocate all memory there
  - Migrate processes to achieve load balancing
  - Spread kernel resources among all the CPUs
  - Attempt to preferentially allocate local resources
  - Migrate resource ownership to CPU that is using it

The Key SMP Scaling Problem

- True shared memory is expensive for large numbers of processors
- NUMA systems require a high degree of system complexity to perform well
  - Otherwise, they're always accessing remote memory at very high costs
- So there is a limit to the technology for both approaches
- Which explains why SMP is not ubiquitous

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

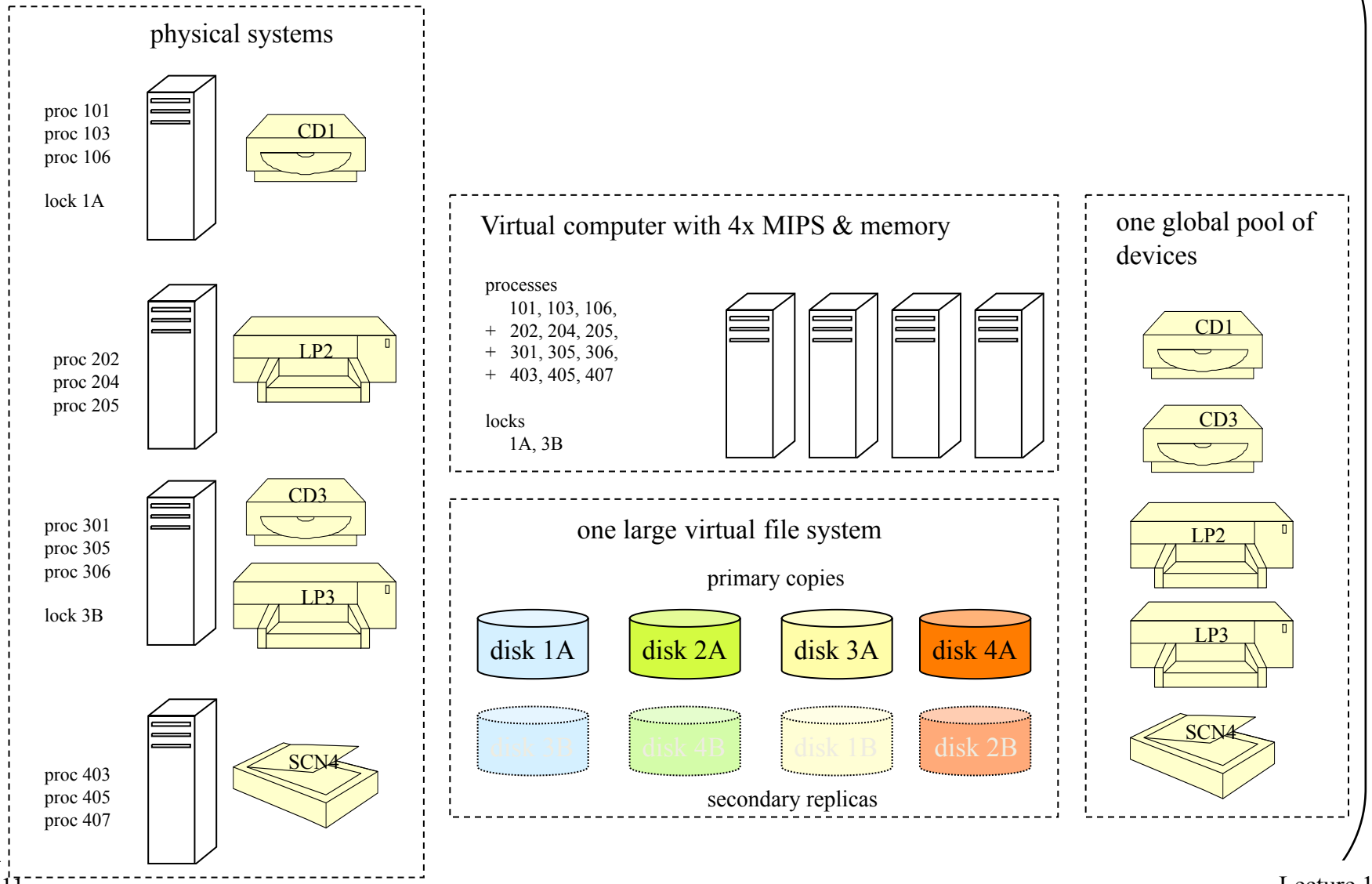
Motivations for Single System Image Computing

- 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

Why Did This Sound Like a Good Idea?

- Programs don't run on hardware, they run on top of an operating system
- All the resources that processes see are already virtualized
- Don't just virtualize a single system's resources, virtualize many systems' resources
- Applications that run in such a cluster are (automatically and transparently) distributed

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 minimize overhead
  - 10-20% overall is not uncommon, 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

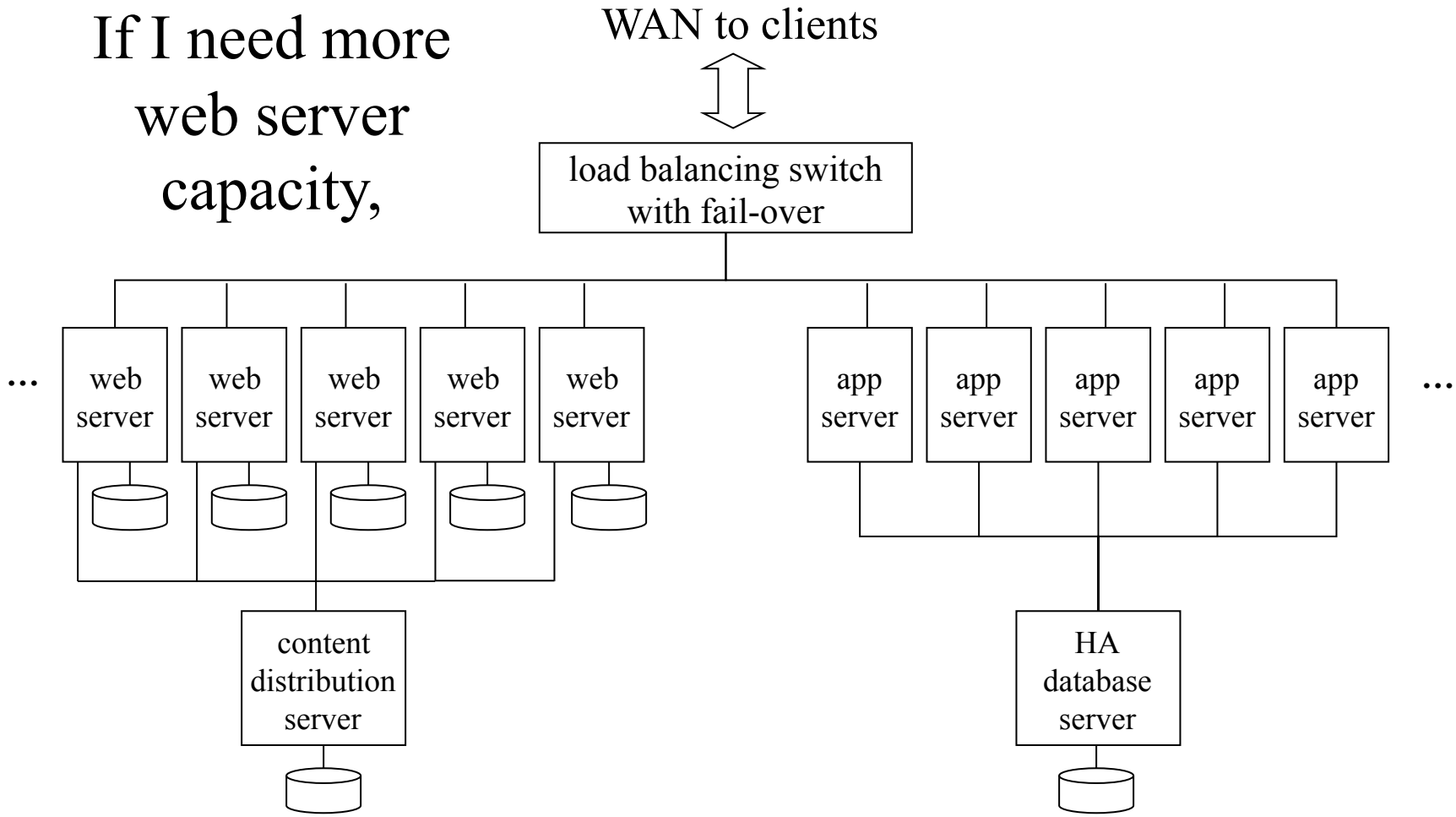
- Characterization:
  - A parallel group of independent computers
  - Serving similar but independent requests
  - Minimal coordination and cooperation required
- Motivation:
  - Scalability and price performance
  - Availability – if protocol permits stateless servers
  - Ease of management, reconfigurable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

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
  - 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

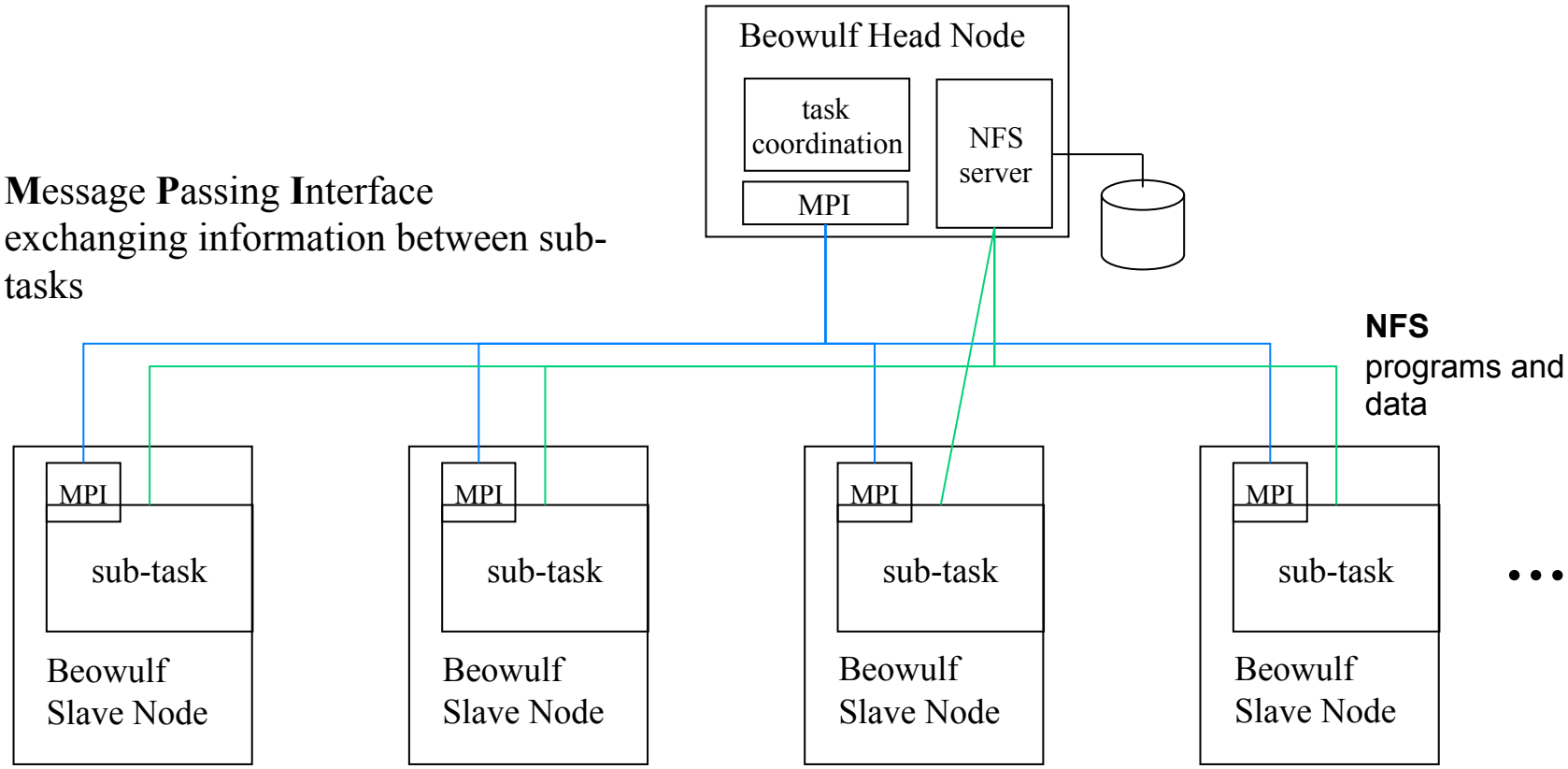
Limited Transparency Clusters

- Single System Image clusters had problems
  - All nodes had to agree on state of all objects
  - Lots of messages, lots of complexity, poor scalability
- What if they only had to agree on a few objects?
  - Like cluster membership and global locks
  - Fewer objects, fewer operations, much less traffic
  - Objects could be designed for distributed use
    - Leases, commitment transactions, dynamic server binding
- Simpler, better performance, better scalability
  - Combines best features of SSI and horizontally scaled loosely coupled systems

Example: Beowulf Clusters

- A technology for building high performance parallel machines out of commodity parts
- One server machine controlling things
- Lots of pretty dumb client machines handling processing
- A LAN technology connecting them
  - Standard message passing between machines
- Applications must be written for parallelization

Beowulf High Performance Computing Cluster



There is no effort at transparency here. Applications are specifically written for a parallel execution platform and use a Message Passing Interface to mediate exchanges between cooperating computations.

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 might or might not be distributed
- Many of the hard problems of the distributed ones 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
  - With no interactions in the midst of computation
- 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

Map-Reduce

- A computational technique for performing operations on large quantities of data
  - For not-quite embarrassingly parallel operations
- Basically:
  - Divide the data into pieces
  - Farm each piece out to a machine
  - Collect the results and combine them
- For example, searching a large data set for occurrences of a phrase
- Originally developed by Google

Map-Reduce in Cloud Computing

- A master node divides the problem among N cloud machines
- Each cloud machine performs the map operation on its data set
- When all complete, the master performs the reduce operation on each node's results
- Can be divided further
  - E.g., a node given a piece of a problem can divide it into smaller pieces and farm those out
  - Then it does a reduce before returning to its master

An Important Lesson From Map-Reduce

- Map-reduce is powerful, widely used, and successful
- It is not fully general
- BUT, by recognizing that many important computations don't require generality
- It allows efficient, correct distributed computations for wide range of applications
- The lesson: utility is usually more important than generality

Do-It-Yourself Distributed Computing in the Cloud

- Generally, you can submit any job you want to the cloud
- If you want to run a SSI or horizontally scaled loosely coupled system, be their guest
  - Assuming you pay, of course
- They'll offer basic system tools
- You'll do the distributed system heavy lifting
- Wouldn't it be nice if you had some middleware to help . . . ?

Distribution at the Application Level

- This course has focused on the OS as a "platform"
  - OS services have evolved to meet application needs
  - SMP creates a scalable distributed OS platform
  - SSI clusters are a robust distributed OS platform
- There are limitations to such a platform
  - Architectural limitations on scalability
  - A legacy of single-system semantics
  - Heterogeneity is a fundamental fact of life
- Who said "applications must be written to an OS?"
  - Perhaps there are other, more suitable, platforms

A Different Paradigm

- We tried to make remote services appear local
  - This failed for the reasons that Deutch laid out
- We don't want to distinguish local from remote
  - Doing so is awkward, constraining, and poor abstraction
- What's our other option?
- What if we made all services seem remote?

Embracing Remote Services

- Design interactions for remote services
- Provide:
  - Discovery
  - Rendezvous
  - Leases
  - Rebinding
  - And other features to deal with Deutsch's fallacies
- And then provide efficient local implementations
  - Minimizing performance penalty for local resources

Alternatives to Distributed Operating Systems

- Network aware applications
  - That register themselves with network name services
  - Exchange services by sending messages
  - Monitor the comings and goings of their partners
- Distributed middleware
  - To provide convenient, distributed objects and services
  - Examples:
    - Platforms: RPC, COM/.NET, Java Beans
    - Environments: Erlang, Rational Rose, Ruby on Rails
    - Services: TIBCO pub/sub messaging

RPC As an Underlying Paradigm

- Procedure calls are already a fundamental paradigm
  - Primary unit of computation in most languages
  - Unit of information hiding in most methodologies
  - Primary level of interface specification
- RPC is a natural boundary between client and server
  - Turn procedure calls into message send/receives
- A few limitations
  - No implicit parameters/returns (e.g., global variables)
  - No call-by-reference parameters
  - Much slower than procedure calls (TANSTAAFL)
  - Partial failure far more likely than local procedure calls

Key Features of RPC

- Client application links against local procedures
  - Calls local procedures, gets results
- All RPC implementation is inside those procedures
- Client application does not know about RPC details
  - Does not know about formats of messages
  - Does not worry about sends, timeouts, resents
  - Does not know about external data representation
- All of this is generated automatically by RPC tools
  - Canonical versions of converting calls to messages
- The key to the tools is the interface specification

Objects – Another Key Paradigm

- Not inherently distributed, but . . .
- A dominant application development paradigm
- Good interface/implementation separation
  - All we can know about object is through its methods
  - Implementation and private data opaquely encapsulated
- Powerful programming model
  - Polymorphism ... methods adapt themselves to clients
  - Inheritance ... build complex objects from simple ones
  - Instantiation ... trivial to create distinct object instances
- Objects are not intrinsically location sensitive
  - You don't reference them, you call them

Local Objects and Distributed Computing

- Local objects are supported by compilers, inside an address space
  - Compiler generates code to instantiate new objects
  - Compiler generates calls for method invocations
- This doesn't work in a distributed environment
  - All objects are no longer in a single address space
  - Different machines use different binary representations
  - You can’t make a call across machine boundaries

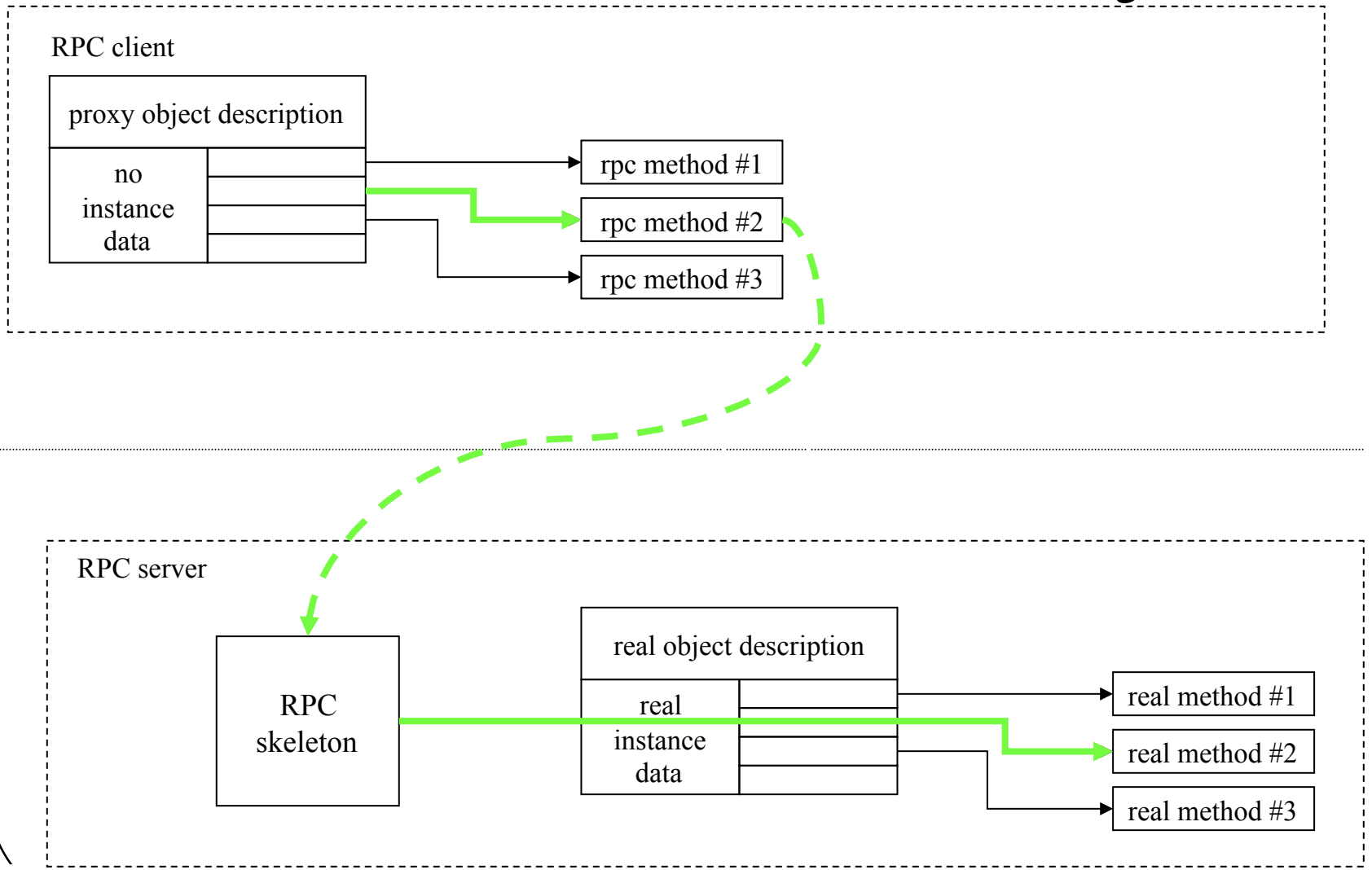
Merging the Paradigms

- Implement method calls with RPC, instead of local procedure calls
- The concept of an object hides what's inside, anyway
  - You shouldn't use global variables and calls by reference with them, anyway
- The mechanics are a bit more complicated than simply RPC, though

Invoking Remote Object Methods

- Compile OO program with proxy object implementation
  - Defines the same interface (methods and properties)
  - All method invocations go through the local proxy
- Local implementation is proxy for remote server
  - Translate parameters into a standard representation
  - Send request message to remote object server
  - Get response and translate it to local representation
  - Return result to caller
- Client cannot tell that object is not local

Proxies for Distributed Objects



Dynamic Object Binding

- How can we compile to a binary when some of the objects (and their implementations) are remote?
- Local objects are compiled into an application and are fully known at compile time
- Distributed objects must be bound at some later time
- These objects are provided by servers
  - The available servers change from minute to minute
  - New object classes can be created in real time
  - So the "later time" is run time
- We need a run-time object "match-maker"
  - Like DLLs on steroids

Object Request Brokers (ORBs)

- ORBs are the matchmakers
- A local portal to the domain of available objects
- A registry for available object implementations
  - Object implementers register with the broker
- Meeting place for object clients and implementers
  - Clients go to broker to obtain services of new objects
- A local interface to remote object components
  - Clients reference all remote objects through local ORB
- A router between local and remote requests
  - ORBs pass messages between clients and servers
- A repository for object interface definitions

But Still TANSTAAFL

- Moving distribution out of OS doesn't change the fact that distributed computing is complex
- It avoids having to ensure that everything local is invisibly distributed
- But those remote application-level objects still:
  - Need synchronization
  - Need to reach consensus
  - Need to handle partial failures
- Advantage is you can customize it to your needs

Evolution of System Services

- Operating systems started out on single computers
  - This biased the definition of system services
- Networking was added on afterwards
  - Some system services are still networking-naïve
  - New APIs were required to exploit networking
  - Many applications remained networking-impaired
- New programming paradigms embrace the network
  - Focus on services and interfaces, not implementations
  - Goal is to make distributed applications easier to write
- Increasingly, system services offered by the network

The Changing Role of Operating Systems

- Traditionally, operating systems:
  - Abstracted heterogeneous hardware into useful services
  - Managed system resources for user-mode processes
  - Ensured resource integrity and trusted resource sharing
  - Provided a powerful platform for application developers
- Now,
  - The notion of a self-contained system is fading
  - New programming platforms:
    - Are instruction set and operating system independent
    - Encompass and embrace distributed computing
    - Provide much higher level objects and services
- But they still depend on powerful underlying operating systems

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