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,

WAN to clients

load balancing switch with fail-over

web server

web server

web server

web server

content distribution server

web server

app server

app server

app server

app server

app server

HA database server

…
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
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.

**Message Passing Interface**

exchanging information between sub-tasks

---

**Beowulf High Performance Computing Cluster**

Beowulf Head Node

- **task coordination**
- **NFS server**

MPI

NFS programs and data

---

Beowulf Slave Node

- **MPI**
- **sub-task**

Beowulf Slave Node

- **MPI**
- **sub-task**

Beowulf Slave Node

- **MPI**
- **sub-task**

Beowulf Slave Node

- **MPI**
- **sub-task**

...
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
- 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

• 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
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
  – 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

RPC client

proxy object description

no instance data

rpc method #1
rpc method #2
rpc method #3

RPC server

RPC skeleton

real object description

real instance data

real method #1
real method #2
real method #3
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