Overview of Presentation

classes of distributed system
- loosely coupled, SMP, Single-system-image Clusters
- what they are, why we care, how they work

applications in distributed systems
- Peter Deutches 7 Falacies of network computing
- RPC, distributed objects, ORBs, CORBA, IDL

topics not to be covered (well covered in text)
- remote virtual memory, details of RPC

Major Classes of Distributed Systems

loosely coupled systems
- independent systems in constant communication

Symmetric Multi-Processors (SMP)
- multiple CPUs, sharing memory and I/O devices

Single-System Image – Cluster Computing
- a group of computers, networked together
- behaving as though they were one huge computer

Loosely Coupled Systems

Characterization:
- communicating but independent systems

Motivation:
- resource sharing, independence

Examples:
- client/server computing (e.g. file and printer sharing)
- work-group computing (e.g. peer file system sharing)
- network services (e.g. e-mail)
- network aware applications (e.g. browsers, X11 apps)

Loosely Coupled – Transparency

what things look the same as local?
- remote file systems
- remote terminal sessions, X sessions
- remote procedure calls

what things don't look the same as local?
- primitive synchronization (e.g. mutexes)
- basic Inter-Process Communication (e.g. signals)
- process creation, destruction, status, authorization
- accessing devices (e.g. tape drives)

Symmetric Multi-Processors

Characterization:

- multiple CPUs sharing memory and devices

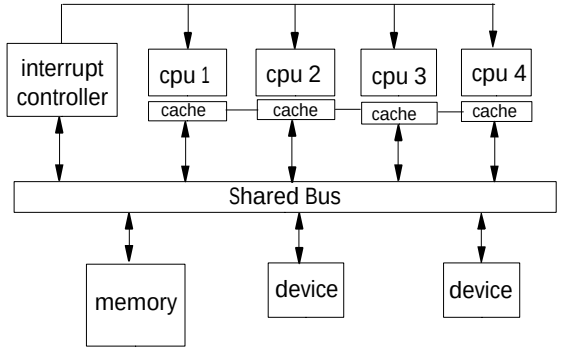
Motivation:

- price performance (lower price per MIP)
- scalability (economical way to build huge systems)
- perfect application transparency

Examples:

- 4-way Intel servers, 64-way SPARC servers

SMP Architecture



SMP Price/Performance

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

massive multi-processors are a huge win

- if you can manage memory and lock contention

SMP Operating System Design

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 must 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)

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 memory references are satisfied by per-CPU cache
- if too many requests go to memory, CPUs slow down

scalability depends on lock contention

- waiting for spin-locks waste 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

Cluster Computing

Characterization:

- a group of seemingly independent computers collaborating to provide SMP-like transparency

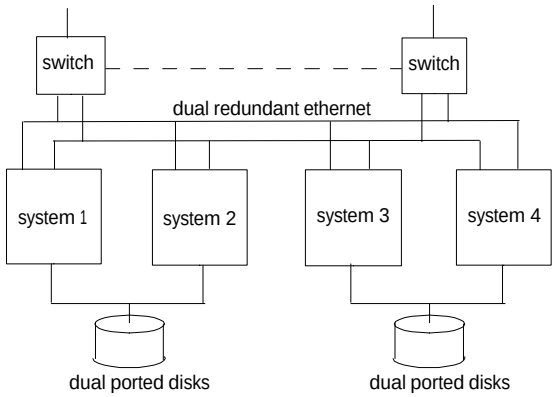
Motivation:

- high availability, service survives node/link failures
- scalable capacity (overcome SMP contention problems)
- good application transparency

Examples:

- Locus, Sun Clusters, MicroSoft Wolf-Pack
- enterprise storage, telecommunications control, ...

Cluster Computing Architecture



Clusters – complete transparency

Single System Image: file systems
- all processes see same files, wherever they run

Single System Image: processes
- all nodes know about all processes

Single System Image: devices
- all nodes see all the devices in the network

Single System Image: all system services
- it doesn't matter where requestor and resources are
- it is as if all processes were on a single computer

Tightly Coupled Systems

in SMP systems, all CPUs see same kernel resources

in clustered system, OS's must exchange messages
- advising one-another of all changes to resources
  - each OS's internal state mirrors the global state
- request execution of node-specific requests
  - node-specific requests automatically forwarded to right node

the implementation is large and complex

the exchange of messages can be very expensive

getting it right takes many staff-centuries

Clusters – what's so hard?

all nodes must agree on state of all resources
- requires many elaborate protocols
- complex error handling when someone fails

coordination requires a central coordinator
- the nodes must elect one of them to be the master
- distributed concensus algorithms can be very complex

when the master fails, everything stops

it may not always be possible to elect a master
- partially connected networks, (or worse) "split-brain"

Clustered Performance

clever implementation can minimize overhead
- 10-20% overall is not uncommon

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

very nice for application developers and customers

Distributed Systems – Summary

different degrees of transparency

- do applications see a network or a single system image

different degrees of coupling

- making multiple computers cooperate is difficult
- doing it without shared memory is even worse

performance, independence, robustness trade-off

- cooperating redundant nodes offer higher availability
- communication and coordination are expensive
- mutual dependency creates more modes of failure

Peter Deutsch's "Seven Falacies 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

True transparency is not achievable

Distributed Applications

assume we don't want to go with SMP or Cluster

- expense: extra hardware, extra protocols, overhead, ...
- inappropriate: clients already have their own computers
- pragmatic: Peter Deutsch is probably right

but we do want to run distributed applications

- client/server computing really is a good model
- we still like the scalability and availability advantages
- networking can embrace heterogenous systems

new applications must be “network aware”

Network-aware Applications

applications can be written to be network-aware

- register themselves with network name services
- exchange services by sending messages
- monitor the comings and goings of systems

this is very different from non-distributed applications

- obtain services by making procedure calls
- don't worry about other systems going up and down

writing such applications would be very different

- requires much more work from applications developers

Moving from ABIs to Protocols

Operating Systems and ABIs

- stable binary interfaces between applications and the OS
- the basis for version interoperability
- but ABIs are platform-specific, tied to an instruction set

Network Protocols

- are also binary interfaces for client/server interaction
- they are platform neutral (or at least they should be so)
- they provide a basis for version interoperabilty
- they also provide heterogenous platform interoperability

Location Independent Services

network-aware application interaction services

- an inter-process service request mechanism
- as easy to use as procedure calls
- works identically whether clients are local or remote

automatically handle complex details

- automatic client server rendezvous
- hiding of host and link failures

more than just network aware ... easy to use

- even local applications would want these services

Embracing a new Paradigm

we tried to make remote services appear local

- this failed for the reasons that Deutch layed out

we don't want to distinguish local from remote

- this is both awkward and inflexible

perhaps we should make all services seem remote

- design our interaction paradigms for remote services
- including discovery, rendezvous, leases, rebinding, and other features to deal w/Deutsch's falacies

perhaps we can have efficient local implementations

General Paradigm – RPC

procedure calls – a fundamental paradigm

- primary unit of computation in most languages
- primary unit of information hiding in most methodologies
- primary level of interface specification

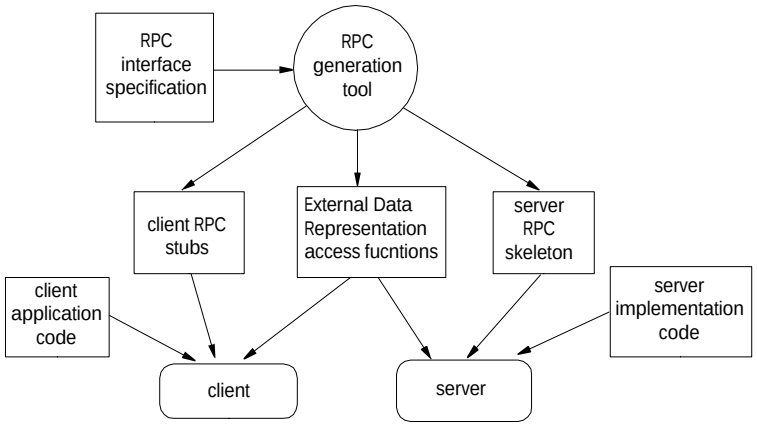
clients/servers can be split at procedure boundaries

- turn procedure calls into message send/receives

a few limitations

- no implicit parameters/returns (e.g. global variables)
- no call-by-reference parameters

Remote Procedure Calls – Tool Chain



(RPC – Key Features)

client application links against local procedures

client application 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 the RPC tools

the key to the tools is the interface specification

Paradigm – Objects

today's dominant application development paradigm

good interface/implementation separation

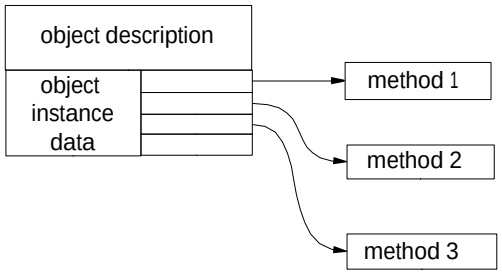
- all we can know about object is through its methods
- implementation and private data opquely encapsulated

powerful programming model

- polymorphism ... methods can adapt to requests
- inheritance ... build complex objects from simple ones
- instantiation ... trivial to create new object instances

objects are not intrinsically location sensitive

Simple Local Objects



Objects – Local vs. Distributed

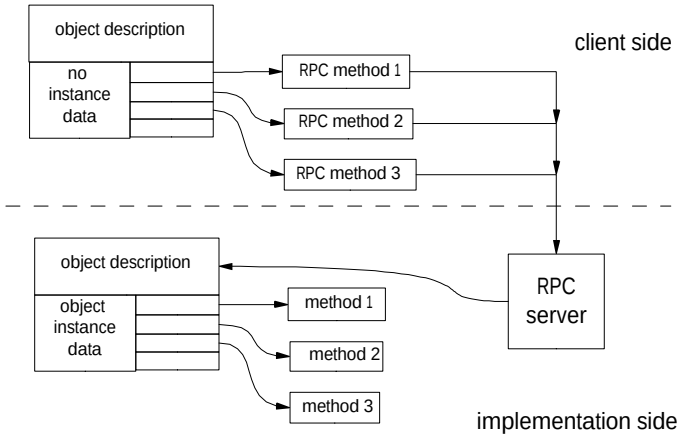
local objects

- 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
- method invocation must be done via message exchange

Proxies for Distributed Objects



(Invoking Remote Object Methods)

program compiles with proxy object implementation

- defines the same interface (methods and properties)
- all method invocations go through the local proxy

local implemenation acts as 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

Dynamic Object Binding

local objects are compiled into an application

- the compiler “knows” about all available objects
- there is no need to "discover" their implementations

distributed objects are provided by servers

- the available servers change from minute to minute
- new object classes can be created in real time

we need an object "match-maker"

- tracks object servers and classes as they come and go
- matches clients' object requests with available servers

Object Request Brokers (ORBs)

a registry for available object implementations
- object implementers register with the broker

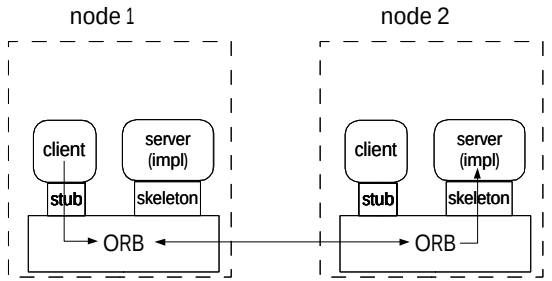
a meeting place for object clients and implementers
- clients go to broker to obtain services of new objects

a local interface to remote object components
- clients reference all remote objects through local ORB

a router between local and remote requests
- ORB forwards messages between clients and servers

a repository for object interface definitions

ORBs and distributed objects



Interface Definition Language

a language neutral interface definition language

describes all methods, parameters and returns

supports inheritance and polymorphism

used to generate client stubs and server skeletons
- C, C++, Smalltalk, COBOL, ADA, Lisp, ...
- conceptually similar to RPC specification and rpcgen tool

also used to generate standard request messages
- message language is portable and very well specified
- any client can talk to any server (full interoperability)

Internet Inter-ORB Protocol

different ORBs may have very different goals
- hard real time, small footprint, very fast local IPC
- huge numbers of clients, high-availability

Common Object Request Broker Architecture
- define standard model for objects and services

IIOP
- the common inter-ORB language
- enable different ORBs to exchange objects and services
- machine, language, and operating system independent

Distributed Applications

Operating Systems started out on single computer
- this biased the definition of system services

Networking was added on afterwards
- some system services are still networking-naive
- new APIs were required to exploit networking
- many applications remained networking-impaired

New programming paradigms embrace networking
- focus on services and interfaces, not implementations
- goal is to make it easier to write distributed applications

What do Operating Systems do?

efficient management of system resources
- so that they can serve the application developers

enable effective exploitation of hardware capabilities
- so that they can serve the application developers

provide trusted resource sharing
- to protect the application users

replace complexity with powerful abstractions
- facilitate development of more sophisticated applications

Key Concepts

classes of distributed systems
- loosely coupled, SMP, SSI clusters
- goals, characteristics and architecture of each
- coordination, performance, robustness issues

distributed computing paradigms
- Deutsch's 7 falacies, network aware applications
- Remote Procedure Calls, client/server architecture
- distributed objects (v.s. local objects)
- ORB model, Interface Definition Language, IIOP

SMP device I/O

all processors have equal access to memory/devices
- any processor can initiate an I/O operation
  initiating processor need not be one that requested the I/O
- any processor can service an I/O interrupt
  serviceing processor need not be one that initiated I/O

SMP interrupt controller picks which CPU to interrupt
- dynamic priorities, always interrupt lowest priority CPU
- fixed binding of some or all interrupts to one CPU
- automatic round-robin delivery

Managing Memory Contention

Fast n-way memory controllers are very expensive

- without them, memory contention taxes performance
- cost/complexity limits how many CPUs we can add

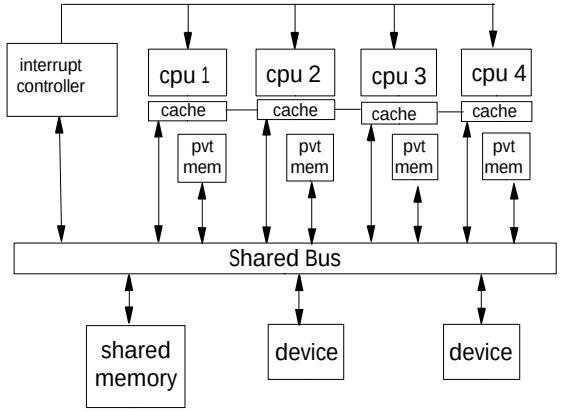
Non-Uniform Memory Architectures (NUMA)

- each CPU gets its own memory, which is on the bus
- CPU has fast path to its own memory
- using other CPU's memory (on bus) is much slower

NUMA memory controllers much easier to build

- cheaper controllers with faster access to local CPU
- easily support huge numbers of CPUs

NUMA Architecture



NUMA System Performance

it is all about local memory hit rates

- every outside reference costs us 3-10x performance
- we need 70-95% hit rate 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 ballancing
- spread kernel resources among all the CPUs
- attempt to preferentially allocate local resources