Operating Systems Principles

Advanced Architectures

Mark Kampe  
(markk@cs.ucla.edu)

Advanced Architectures

15A. Distributed Computing
15B. Multi-Processor (and NUMA) Systems
15C. Tightly Coupled (SSI) Clusters
15D. Loosely Coupled (Horizontally Scalable)
15E. Cloud Models
3F. Virtual Machines

Goals of Distributed Computing
• better services
  — scalability
    • apps too big to run on a single computer
    • grow system capacity to meet growing demand
    — improved reliability and availability
    — improved ease of use, reduced CapEx/OpEx
  • new services
    — applications that span multiple system boundaries
    — global resource domains, services (vs. systems)
    — complete location transparency

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
• application level distributed computing
  — peer-to-peer, application level protocols
  — distributed middle-ware platforms

Evaluating 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
  — on how many things do distinct systems agree
  — how is that agreement achieved

SMP systems and goals
• Characterization:
  — multiple CPUs sharing memory and devices
• Motivations:
  — price performance (lower price per MIP)
  — scalability (economical way to build huge systems)
  — perfect application transparency
• Example:
  — single socket, multi-core Intel CPUs
Symmetric Multi-Processors

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
- 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 obvious direction

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 in 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 references satisfied from per-core 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
- contention wastes cycles, reduces throughput
  - 2 CPUs might deliver only 1.9x performance
  - 3 CPUs might deliver only 2.7x performance

Managing Memory Contention

- Fast n-way memory is very expensive
  - without it, memory contention taxes performance
  - cost/complexity limits how many CPUs we can add
- Non-Uniform Memory Architectures (NUMA)
  - each CPU has its own memory
    - 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
Non-Uniform Memory Architecture
Symmetric Multi-Processors

OS design for NUMA systems
• it is all about local memory hit rates
  – 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

Single System Image (SSI) Clusters
• Characterization:
  – a group of seemingly independent computers
    collaborating to provide SMP-like transparency
• Motivation:
  – higher reliability, availability than SMP/NUMA
  – more scalable than SMP/NUMA
  – excellent application transparency
• Examples:
  – Locus, Sun Clusters, MicroSoft Wolf-Pack, OpenSSI
  – enterprise database servers

Modern Clustered Architecture

Structure of a Modern OS
OS design for SSI clustering

• 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
    • request execution of node-specific requests
    • node-specific requests are forwarded to owning node
• implementation is large, complex, difficult
• the exchange of messages can be very expensive

SSI Clustered 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
• nice for application developers and customers
  – but they are complex, and not particularly scalable

Lessons Learned

• 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 greatly complicated operating system
  – distributed objects are more complex to manage
  – complex optimizations to reduce the added overheads
  – new modes of failure w/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, Google search farm, Hadoop

Horizontal Scalability w/HA

• farm of independent servers
  – servers run same software, serve different requests
  – may share a common back-end database
• front-ending 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

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

Limited Location 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 create, destroy, status, authorization
  - Accessing devices (e.g. tape drives)

Loosely Coupled Scalability

- Many servers, continuous change
  - Dramatic fluctuations in load volume and types
  - Continuous node additions for increased load
  - Nodes and devices are failing continuously
  - Continuous and progressive s/w updates
- Most services delivered via switched HTTP
  - Clients/server communication is over WAN links
  - Large (whole file) transfers to optimize throughput
  - Switches route requests to appropriate servers
  - Heavy reliance on edge caching

Distributed Systems – Summary

- 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
- Performance vs. independence vs. robustness
  - Cooperating redundant nodes offer higher availability
  - Communication and coordination are expensive
  - Mutual-dependency creates more modes of failure

Clouds: Applied Horizontal Scalability

- Many servers, continuous change
  - Dramatic fluctuations in load volume and types
  - Continuous node additions for increased load
  - Nodes and devices are failing continuously
  - Continuous and progressive s/w updates
- Most services delivered via switched HTTP
  - Clients/server communication is over WAN links
  - Large (whole file) transfers to optimize throughput
  - Switches route requests to appropriate servers
  - Heavy reliance on edge caching
Geographic Disaster Recovery

• Cloud reliability/availability are key
  — one data center serves many ($10^3$-$10^7$) clients
• Local redundancy can only provide 4-5 nines
  — fires, power and communications disruptions
  — regional scale (e.g. flood, earthquake) disasters
• Data Centers in distant Availability Zones
  — may be running active/active or active/stand-by
  — key data is replicated to multiple data centers
  — traffic can be redirected if a primary site fails

WAN-Scale Replication

• WAN-scale mirroring is slow and expensive
  — much slower than local RAID or network mirroring
• Synchronous Mirroring
  — each write must be ACKed by remote servers
• Asynchronous Mirroring
  — write locally, queue for remote replication
• Mirrored Snapshots
  — writes are local, snapshots are mirrored
• Fundamental tradeoff: reliability vs. latency

WAN-Scale Consistency

• CAP theorem - it is not possible to assure:
  — Consistency (all readers see the same result)
  — Availability (bounded response time)
  — Partition Tolerance (survive node failures)
• ACID databases sacrifice partition tolerance
• BASE semantics make a different trade-off
  — Basic Availability (most services most of the time)
  — Soft state (there is no global consistent state)
  — Eventual consistency (changes propagate, slowly)

Dealing with Eventual Consistency

• distributed system has no single, global state
  — state updates are not globally serialized events
  — different nodes may have different opinions
• expose the inconsistencies to the applications
  — ask the cloud, receive multiple answers
  — let each application reconcile the inconsistencies
• BASE semantics are neither simple nor pretty
  — they embrace parallelism and independence
  — they reflect the complexity of distributed systems

Distributed Computing Reformation

• systems must be more loosely coupled
  — tight coupling is complex, slow, and error-prone
  — move towards coordinated independent systems
• move away from old single system APIs
  — local objects and services don’t generalize
  — services are obtained through messages (or RPCs)
  — in-memory objects, local calls are a special case
• embrace the brave new (distributed) world
  — topology and partnerships are ever-changing
  — failure-aware services (commits, leases, rebinds)
  — accept distributed (e.g. BASE) semantics

Changing Architectural Paradigms

• a “System” is a collection of services
  — interacting via stable and standardized protocols
  — implemented by app software deployed on nodes
• Operating Systems
  — manage the hardware on which the apps run
  — implement the services/ABIs the apps need
• The operating system is a platform
  — upon which higher level software can be built
  — goodness is measured by how well it does that job
What Operating Systems Do

• Originally (and at the start of this course)
  – abstract heterogeneous hardware into useful services
  – manage system resources for user-mode processes
  – ensure resource integrity and trusted resource sharing
  – provide a powerful platform for developers
• None of this has changed, but ...
  – notion of a self-contained system becoming obsolete
  – hardware and OS heterogeneity is a given
  – most important interfaces are higher level protocols
• Operating Systems continue to evolve as
  – new applications demand new services
  – new hardware must be integrated and exploited

Final Exam (Mon 6/6)

• Location: Humanities A51
• Part 1: 11:30-13:00
  – 10 questions, similar to mid-term
  – covering weeks 6-10
• Part 2: 13:00-14:30
  – 6 hard questions, choose any 3 to answer
    • real problems: analyze, explain, propose approach
    • questions not answered in reading or lecture
  – covering the entire course

Control and Data Planes

Scalability: Cluster Protocols

• Consensus protocols do not scale well
  – they only work for small numbers of nodes
• Minimize number of consensus operations
  – elect a single master who makes decisions
  – partitioned and delegated responsibility
• Avoid large-consensus/transaction groups
  – partition work among numerous small groups
• Avoid high communications fan-in/fan-out
  – hierarchical information gathering/distribution
Hierarchical Communication Structure

Paradigm – Objects
• 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

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 is 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 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
**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 a run-time object "match-maker"
  - Tracks object servers and classes as they come and go
  - Matches clients' object requests with available servers
    (like DLLs on steroids)

**Object Request Brokers (ORBs)**
- 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 router between local and remote requests
  - ORBs reference all remote objects through local ORB
- A repository for object interface definitions

**ORBs and Distributed Objects**

**Distributed Applications**
- Operating Systems started 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 network
  - Focus on services and interfaces, not implementations
  - Goal is to make distributed applications easier to write

**SMP Device I/O**
- All processors can access all 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
    • Servicing processor need not be one that initiated I/O
- 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

**Global Resource View**
(Heterogeneous systems & resources)
Single Point of Management
(for heterogeneous systems)

Loosely Coupled Availability
(Kimberlite HA Linux platforms)

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/services
  - machine, language, operating system independent