Extreme computing
Infrastructure

Stratis D. Viglas

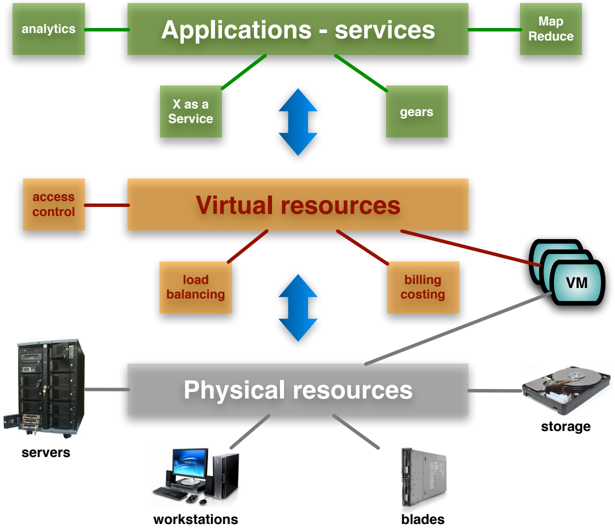
School of Informatics
University of Edinburgh

Outline

So, you want to build a cloud

- Slightly more complicated than hooking up a bunch of machines with an ethernet cable
  - Physical *vs.* virtual (or logical) resource management
  - Interface?
- A host of issues to be addressed
  - Connectivity, concurrency, replication, fault tolerance, file access, node access, capabilities, services, . . .
  - Tired already? (*Hint: you should be.*)
- We'll tackle the problems from the ground up
  - The problems are nothing new
  - Solutions have existed for a long time
  - However, it's the first time we have the capability of applying them all in a single massively accessible infrastructure

Typical cloud architecture



Outline

Distributed file systems

- The idea is quite straightforward
  - Separation of logical and physical storage
  - Not everything resides on a single physical disk
  - Or the same physical rack
  - Or the same geographical location
  - Or the same domain
- Obligatory buzzwords
  - NFS, AFS , CODA, GFS , HDFS
- When dinosaurs roamed the earth[1]...

[1] Either that cliche, or an obscure Tom Waits quote: “we have to go all the way back to the civil war” – immediate pass for anyone who can name the reference.

So, what's a (distributed) file system?

- Operating system service responsible for secondary storage I/O
- Kind of easy when we're talking about a single disk on the motherboard's controller
  - Format the disk, maintain bookkeeping structures, handle operating system's DMA traps by scheduling disk I/O and copying back to memory

So, what's a (distributed) file system?

- Operating system service responsible for secondary storage I/O
- Kind of easy when we're talking about a single disk on the motherboard's controller
    - `<bliss>`
    - Format the disk, maintain bookkeeping structures, handle operating system's DMA traps by scheduling disk I/O and copying back to memory
    - `</bliss>`

So, what's a (distributed) file system?

- Operating system service responsible for secondary storage I/O
- Kind of easy when we're talking about a single disk on the motherboard's controller
  - `<bliss>`
  - Format the disk, maintain bookkeeping structures, handle operating system's DMA traps by scheduling disk I/O and copying back to memory
  - `</bliss>`
- What if we have multiple disks, not necessarily on the same machine?
- Fundamental issues of distributed systems
  - File access, file services, sharing, sessions, design

Servers and clients revisited

- File directory — or file system tree
  - Mapping of file names to internal locations that can be used by the file service
  - *E.g.*, `/your/important/file.txt` $\rightarrow$ `(server, /dev/sd0, 0x42)`
- File service
  - Provides file access interface to clients
- Client module (or driver)
  - Client side interface for file and directory service
  - If done right, helps provide complete access transparency

File access

- Separation of responsibility and explicit access
  - Client/server architectures
  - User initiates a connection and accesses remote resources by name
  - Typical examples: `ftp`, `telnet`
    - Early days of UNIX – no need for anything special
  - Horribly inflexible, need something better
- Transparent access
  - User accesses remote resources just as local ones
  - Hence, a distributed file system

File service types

- Upload/download model
  - Multiple servers, but each server responsible for specific files
  - Read file: copy file from server to client
  - Write file: copy file from client to server

File service types

- Upload/download model
  - Multiple servers, but each server responsible for specific files
  - Read file: copy file from server to client
  - Write file: copy file from client to server

Advantages

- Doesn't get simpler than this
- Straightforward extension of explicit access

File service types

- Upload/download model
  - Multiple servers, but each server responsible for specific files
  - Read file: copy file from server to client
  - Write file: copy file from client to server

Advantages

- Doesn't get simpler than this
- Straightforward extension of explicit access

Disadvantages

- Wasteful: what if client needs small piece?
- Problematic: what if client doesn't have enough space?
- Inconsistent: what if others need to modify the same file?

File service types (cont.)

- Remote access model
    - File service provides functional interface
        - `create()`, `delete()`, `read()` bytes, `write()` bytes, *etc.*
- In fact, same interface one would have in a centralised file system

File service types (cont.)

- Remote access model
  - File service provides functional interface
    - `create()`, `delete()`, `read()` bytes, `write()` bytes, *etc.*
- In fact, same interface one would have in a centralised file system

Advantages

- Client gets only what's needed
- Server can manage coherent view of file system

File service types (cont.)

- Remote access model
  - File service provides functional interface
    - `create()`, `delete()`, `read()` bytes, `write()` bytes, *etc.*
- In fact, same interface one would have in a centralised file system

Advantages

- Client gets only what's needed
- Server can manage coherent view of file system

Disadvantages

- Possible server and network congestion
- Servers are accessed for duration of file access
- Same data may be requested repeatedly

What to share and where?

- Ideal situation: each `read()` returns the result of last `write()`
  - Trivial for a single server and without caching
  - Horrible performance and a single point of failure
  - Caching can help, but it creates more problems
    - Cache invalidation and data propagation
    - Requires state and generates traffic on small changes
- Sessions relax the rules
  - File changes are only visible to the process/machine modifying it
  - Last process to modify the file wins
  - Simple and efficient (but not transactional)
- Immutable files
  - Works wonders for replication and versioning, but potentially wasteful
  - What about concurrent modifications?
- File access as an atomic transaction
  - Either all modifications succeed, or they all fail
  - If multiple transactions start concurrently, they are converted to a serialisable schedule

More on transparency

- Goal is to access remote files as if they were local
- Remote file system name space should be syntactically consistent with local name space
    - Either redefine the way all files are named and provide a syntax for specifying remote files
        - *E.g.*, `//server/dir/file`
        - Sensitive as it can cause legacy applications to fail if naming conventions change
    - Or, use file system mounting
        - Overlay portions of remote name space over local name space
        - Makes the remote name space look like it's part of the local name space

Stateful or stateless?

- Stateful: server maintains client-specific state
  - Shorter requests
  - Better performance in processing requests
  - Cache coherence is possible since the server can know who's accessing what
  - File locking is possible
- Stateless: server maintains no information on client accesses
  - Each request identifies file and offsets
  - Server can crash and recover: no state to lose
  - Client can crash and recover (as usual)
  - No `open()`/`close()` needed as they only establish state
  - No server space used for state
    - Great for scalability: gimme more clients, I don't know them, I don't care!
  - But what if a file is deleted on server while client is working on it?
  - File locking (and, potentially, transactions) not possible

Caching

- Hide latency to improve performance for repeated accesses
- Possibilities: server's disk, server's memory, client's disk, client's memory
  - The last two create cache consistency problems (unfortunate, since they're the best performing options)
- Write-through caching: every change is propagated to master copy
  - What if another client reads its own (out-of-date) cached copy?
  - All accesses will require checking with server
  - Or, server maintains state and sends invalidations
- Write-behind caching: delay the writes and batch them
  - Data buffered locally (and others don't see updates!)
  - Remote files updated periodically
  - One bulk wire is more efficient than lots of little writes
  - Problem: ambiguous semantics

Caching (cont.)

- Read-ahead caching: be proactive and prefetch data
  - Request chunks of file (or the entire file) before it is needed
  - Minimize wait when it actually is needed
- Write-on-close caching: implement session semantics and be done with it
- Centralised control
  - Server is responsible for keeping track of who has what open and cached on each node
  - Stateful file system, excessive traffic

Case studies

- Obligatory reference: NFS
- AFS: the most widely deployed distributed file system
- GFS: Google (and Hadoop's) file system, with radically different design objectives

NFS: Network File System, Sun Microsystems, 1985

- Arguably the first distributed file system
- Machines on the same physical network
- Design goals
  - Any machine can be a client or a server
  - Workstations do not necessarily have a disk
  - Heterogeneity a first class citizen
    - Hardware and operating system are not an issue
  - Access transparency
    - Remotely stored files accessed as local files through standard system calls
    - Separation of physical and logical storage
  - Fault-tolerance and recovery
    - No (shared) state whatsoever
  - High performance
    - Network I/O approximately equal to (or faster than) disk I/O
    - Cache, read ahead, write behind

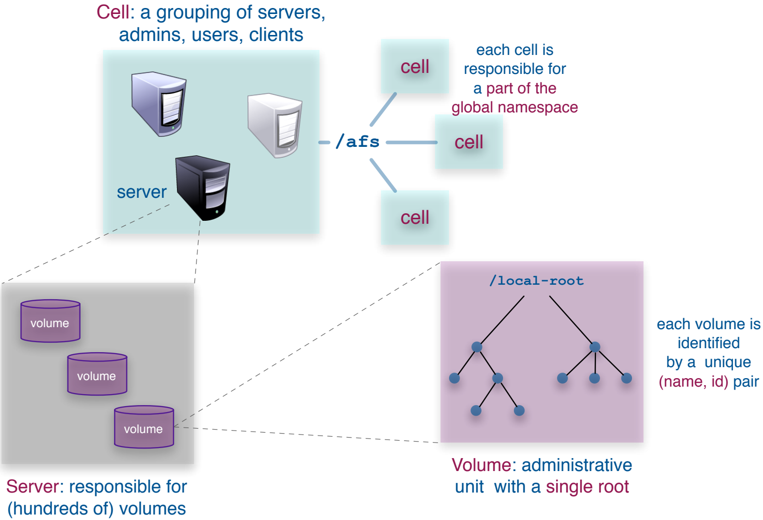
Case study: the Andrew File System (AFS), CMU 1986

- Developed at Carnegie-Mellon University
- Spin-off, acquired by IBM, and subsequently open-sourced
- Most common DFS deployment nowadays
- Goals: large-scale information sharing; scale-out to tens of thousands of nodes
- Design driven by certain key assumptions
  - Most files are small
  - Reads are more common than writes
  - Most files are accessed by one user at a time
  - Files are referenced in bursts and once referenced, a file is likely to be referenced again (spatial and temporal locality)

AFS design decisions

- Whole file serving: on `open()` send the entire file
- Whole file caching
  - Client caches entire file on local disk
  - Client writes the file back to server on `close()`
    - If modified
    - Keeps cached copy for future accesses
- Each client has an AFS disk cache
  - Part of disk devoted to AFS (*e.g.,* 100MB)
  - Client manages cache using LRU
- Clients communicate with set of trusted servers
- Each server presents one identical name space to clients
  - All clients access it in the same way
  - Location transparent

Architecture



File management and access

- Information service: the Volume Location Server (VLS) is a directory of cells and hosts the Volume Location Database (VLDB)
  - All the nodes of the system see the same name space in the form `/afs/cellname/path`
  - For example, `afs/inf.ed.ac.uk/home/derp/code/src/crash.c`
- Read-only volumes may be replicated on multiple servers
- To access a file

File management and access

- Information service: the Volume Location Server (VLS) is a directory of cells and hosts the Volume Location Database (VLDB)
    - All the nodes of the system see the same name space in the form `/afs/cellname/path`
    - For example, `afs/inf.ed.ac.uk/home/derp/code/src/crash.c`
- Read-only volumes may be replicated on multiple servers
- To access a file
    1. Traverse AFS mount point, *e.g.,* `/afs/inf.ed.ac.uk`
    2. AFS client contacts VLDB on VLS to look up the volume

File management and access

- Information service: the Volume Location Server (VLS) is a directory of cells and hosts the Volume Location Database (VLDB)
  - All the nodes of the system see the same name space in the form `/afs/cellname/path`
  - For example, `afs/inf.ed.ac.uk/home/derp/code/src/crash.c`
- Read-only volumes may be replicated on multiple servers
- To access a file
  1. Traverse AFS mount point, *e.g.,* `/afs/inf.ed.ac.uk`
  2. AFS client contacts VLDB on VLS to look up the volume
  3. VLDB returns volume id and list of machines ($\geq$ 1) maintaining file replicas
  4. Request root directory from any machine in the list

File management and access

- Information service: the Volume Location Server (VLS) is a directory of cells and hosts the Volume Location Database (VLDB)
    - All the nodes of the system see the same name space in the form `/afs/cellname/path`
    - For example, `afs/inf.ed.ac.uk/home/derp/code/src/crash.c`
- Read-only volumes may be replicated on multiple servers
- To access a file
    1. Traverse AFS mount point, *e.g.,* `/afs/inf.ed.ac.uk`
    2. AFS client contacts VLDB on VLS to look up the volume
    3. VLDB returns volume id and list of machines (≥ 1) maintaining file replicas
    4. Request root directory from any machine in the list
    5. Root directory contains files, subdirectories, and mount points

File management and access

- Information service: the Volume Location Server (VLS) is a directory of cells and hosts the Volume Location Database (VLDB)
  - All the nodes of the system see the same name space in the form `/afs/cellname/path`
  - For example, `afs/inf.ed.ac.uk/home/derp/code/src/crash.c`
- Read-only volumes may be replicated on multiple servers
- To access a file
  1. Traverse AFS mount point, *e.g.*, `/afs/inf.ed.ac.uk`
  2. AFS client contacts VLDB on VLS to look up the volume
  3. VLDB returns volume id and list of machines (≥ 1) maintaining file replicas
  4. Request root directory from any machine in the list
  5. Root directory contains files, subdirectories, and mount points
  6. Continue parsing the file name until another mount point (from previous step 5) is encountered; go to step 2 to resolve it

Caching in AFS

- On file `open()`
  - Server sends entire file to client and provides a callback promise
    - It will notify the client when any other process modifies the file (possible due to write-through caching)
- If a client modifies a file, contents are written to server on file `close()`
- When a server detects an update
  - Notifies all clients that have been issued the callback promise
  - Clients invalidate cached files
- If a client goes down, then it must recover
  - Contact server with timestamps of all cached files to decide whether to invalidate
- Session semantics: if a process has a file open, it continues accessing it even if it has been invalidated
  - Upon `close()`, contents will be propagated to server; last update wins

AFS pros and cons

Advantages

- Scales well
- Uniform name space
- Read-only replication
- Security model supports mutual authentication, data encryption (though we didn't talk about those)

AFS pros and cons

Advantages

- Scales well
- Uniform name space
- Read-only replication
- Security model supports mutual authentication, data encryption (though we didn't talk about those)

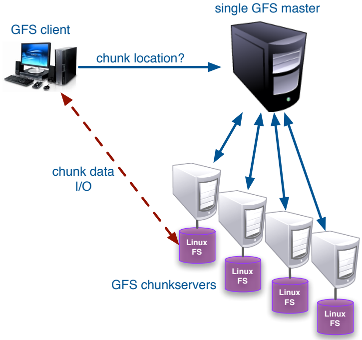
Disadvantages

- Session semantics
- Directory-based permissions
- Uniform name space

Case study: the Google File System (GFS)

- Custom-built file system from Google (bluebrint for HDFS)
- Radically different design objectives, tailored towards large-scale data-intensive analytics
- Basic assumption: things fail; deal with it
  - Thousands of nodes
  - Bugs and hardware failures are out of file system designer's control
  - Monitoring, error detection, fault tolerance, automatic recovery
- Files are much larger than traditional standards
  - Single file size in the order of multiple gigabytes
  - Billions of files constantly served
- Modifications are mainly appends
  - Random writes are practically nonexistent
  - Many files are written once, and read sequentially
- Two types of reads
  - Large streaming reads, or small random reads (but in the forward direction)
- Sustained bandwidth more important than latency

GFS architecture



- GFS cluster
  - A single master, with multiple chunkservers per master
  - Each chunkserver is running a commodity Linux OS and FS
- GFS file
  - Represented as fix-sized chunks
  - Each chunk with a 64-bit unique global ID
  - Stored mirrored across chunkservers (fault tolerance)

More on the design

- Master server maintains all metadata
  - Name space, access control, file-to-chunk mappings, garbage collection, chunk migration
  - Simple, flat design
    - No directories, no hierarchy, only a mapping from metadata to path name
  - Master answers queries only about chunk locations
  - A client typically asks for multiple chunk locations in a single request
  - The master also proactively provides chunk locations immediately following those requested (*a la* read-ahead, but only for metadata)
- GFS clients
  - Consult master for metadata
  - Access data directly from chunkservers
  - No caching at clients and chunkservers due to the frequent case of streaming

Files, chunks, and metadata

- Each file is split in 64MB chunks, thus minimising number of requests to master and overhead of chunk access
- Fewer metadata entries, all kept in master's memory
  - 64 bytes of metadata per 64MB of data
  - File and chunk name spaces
  - File-to-chunk mappings
  - Locations of a chunk's replicas
- No persistent state: midway between stateful and stateless design
  - Chunkservers are monitored through "heartbeat" messages; if a server is dead, use one of the other chunkservers to retrieve a chunk's replica

Consistency model

- Relaxed consistency: concurrent changes are consistent but their order is undefined (first to commit wins)
  - An append is atomically committed at least once
  - Then, all changes to a chunk are applied in the same order to all replicas
  - Primitive versioning to detect missed updates
- To update a chunk
  - The master grants a chunk lease to a replica, which determines the order of updates to all replicas
  - The lease has a timeout of 60s, but can be extended
  - If a lease times out, the master assumes the server is dead and grants a lease to different server
- Replication objectives
  - Maximize data reliability and availabity, and network bandwidth
  - Chunk replicas are spread across physical machines and racks
  - Each file has a replication factor (*i.e.,* how many times its chunks are replicated); low replication factor $\rightarrow$ higher priority

Fault tolerance and detection

- Fast recovery
  - Master and chunkservers are designed to restore their states and start in seconds regardless of termination conditions
- Chunk replication
- Master replication
  - Shadow masters provide read-only access when the primary master is down
- Data integrity
  - A chunk is divided into 64kB blocks, each with its checksum
  - Verified at read and write times
  - Background proactive scans for rarely used data

Outline

Why replicate?

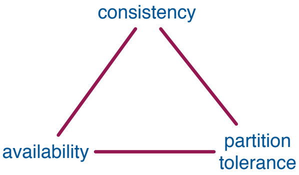
- If one's good, five is better
- The enemy of "good enough" is not "better", it is "$+1$"
- Enhance reliability
  - Correctness in the presence of faults or errors
  - For example, while at least one the AFS servers has not crashed, data is available
- Improve performance
  - Load sharing
  - Alternative locations to access data from
    - But how do we keep them consistent?

More on replication

- Relationship between growth and scalability
  - As the number of users/processes of a system grows, its performance degrades
  - And the probability of failure of any system node grows
- Replication caters for
  - Remote sites working in the presence of local failures
    - If some node fails, its process can be replicated elsewhere
  - Protection against data corruption
    - Probability of all replicas corrupted is lower
  - Data movement minimisation
    - Alternative locations for each piece of data; push processing to the data, not the other way around
- Replication requirements
  - Transparency: clients see logical objects, not physical ones, but each access returns a single result
  - Consistency: all replicas are consistent for some specified consistency criterion

Where's the catch? The CAP theorem

- CAP stands for Consistency, Availability, Partition tolerance
  - Consistency: all nodes see the same data at the same time
  - Availability: node failures do not prevent system operation
  - Partition tolerance: link failures do not prevent system operation
- Largely a conjecture attributed to Eric Brewer
- *A distributed system can satisfy any two of these guarantees at the same time, but not all three*
- You can't have a triangle; pick any one side



More on the requirements

- Transparency is handled at the file system level
- Consistency has different semantics depending on the file system and the application
- Data-centric consistency: a contract between file system and processes

No explicit synchronisation models

Strict	Absolute time ordering of all shared accesses matters
Linearisability	All processes must see all shared accesses in the same order; accesses are ordered according to a global timestamp
Sequential	All processes see all shared accesses in the same order; accesses are not ordered in time
Causal	All processes see causally-related shared accesses in the same order
FIFO	All processes see writes from each other in the order they were used; writes from different processes may not always be seen in that order

Excplicit synchronisation models

Weak	Shared data is consistent only after a synchronization is done
Release	Shared data is made consistent when a critical region is exited
Entry	Shared data pertaining to a critical region is made consistent when a critical region is entered

Sequential consistency example

Sequentially consistent

P1	W(x)1				
P2		W(x)2			
P3			R(x)2		R(x)1
P4				R(x)2	R(x)1
P5		R(x)2			R(x)1

Not sequentially consistent

P1	W(x)1				
P2		W(x)2			
P3			R(x)2		R(x)1
P4				R(x)1	R(x)2
P5			R(x)2		R(x)2

- Notation: $W(x)y$ read value y for resource x (resp. for reads)
- Definition: all processes see the same interleaving set of operations regardless of what that interleaving is
- Consistent case: the first write occurred after the second on all replicas
- Inconsistent case: writes have occurred in a non-sequential order; different replicas see different orders (P4 sees $R(x)1$ before $R(x)2$)

Weak consistency example

Weakly consistent

P1	W(x)1	W(x)2	S			
P2				R(x)2	R(x)1	S
P3				R(x)1	R(x)2	S

Not weakly consistent

P1	W(x)1	W(x)2	S			
P2				S	R(x)1	R(x)2
P3				R(x)1	R(x)2	S

- It all comes down to when processes synchronise
- Consistent case: P2 and P3 have not synchronised, so they cannot have any guarantees about the value of x
- Inconsistent case: P2 has synchronised; it must read the last written value since P1 has synchronised before it (P3 is still okay)

Client-centric and eventual consistency

- Looking at the problem from a different perspective
  - What if we sacrifice global consistency and only care about local consistency?[2]
- (Elective) Assumption: lack of simultaneous updates
  - Maintain a consistent data view for individual clients currently operating on said data
  - Remember sessions?
- Notion of consistency is dictated by the application
  - Read-only access: no problem!
  - Infrequent writes (*e.g.,*DNS): so long as every client updates the same replica, conflicts are minimised
    - In some cases even stale copies are acceptable (*e.g.,* caching web pages)
- Eventual consistency: update single replica, allow the update to propagate lazily
  - Eventually, all replicas will have the same view

[2] Effectively we sacrifice transactional semantics, which makes DB people cry.

Fault tolerance

- Closely related to dependability (a “jack of all trades” term)
    - Availability: system is ready to be used immediately
    - Reliability: system is always up
    - Safety: failures are never catastrophic
    - Maintainability: all failures can be repaired without users noticing (*e.g.,* hot swapping)
- Failure: whenever a resource cannot meet its promise (*e.g.,*CPU failure, link failure, disk failure, . . .)
    - The cause of failure is known as a fault
- A fault tolerant system can provide its services even in the presence of faults

Failure models

Type of failure	Description
Crash	A node halts, but is working correctly until it halts
Ommision (receive, send)	A node fails to respond to requests, either incoming (receive) or outgoing (send)
Timing	A node's response lies outside the specified time interval
Response (value, state)	A node's response is incorrect; the value is wrong, or the flow of control deviates from the correct one
Arbitrary (or, Byzantine)	A server produces arbitrary responses at arbitrary times

Welcome to History 101

- The Byzantine empire (330 – 1453AD)
- At its peak it encompassed the Balkans, Turkey, and the majority of the Mediterrenean
- Conspiracies and intrigue were common practice; typical for intentionally malicious activity among the members of a group
- Who do you trust?
- Analogy to distributed systems
  - Potential disagreement and conflicting state reports, so how can we converge to a common state?

The solution: redundancy and majority algorithms

- Hide the occurence of failure by reducing its probability of being repeatable
  - Information redundancy: error detection and recovery (mostly handled at the hardware layer)
  - Temporal redundancy: start operation and if it does not complete start it again; make sure operation is idempotent or atomic (think transactions)
  - Physical redundancy: add extra software and hardware and have multiple instances of data and processes

The problems of fault tolerance

- Process resilience
  - Process failure prevention by replicating processes into groups
    - Design issues
    - How to achieve agreement within a group
- Reliable client/server communications
  - Masking crash and omission failures
- Reliable group communication
  - What happens when processes join/leave the group during communication?
- Distributed commit
  - Transactional semantics and atomicity: operation should be performed by all group members, or none at all
- Recovery strategies
  - Recovery from an error is fundamental to fault tolerance

Consensus algorithms

- Objective: have all non-faulty processes reach consensus quickly

The two generals problem

Two armies, each led by a general, plan to attack a city. Each army is on a hill; the city is in the valley between the hills. Generals communicate through messengers. A messenger has to go through the valley (and potentially be intercepted by the city's defenders). The attack will be successful only if both generals attack at the same time. How can they coordinate?

- No sequence of communication steps ensures a consensus
- Generalisation: the Byzantine Generals problem
  - Same goal, but multiple generals/armies
  - There is a solution only if the number of exchanged messages is three times the number of lost messages

Byzantine fault tolerance

A consensus can be reached if we have $3m + 1$ processes and up to m of them are faulty ($2m + 1$ functioning properly); the system is then Byzantine fault tolerant

Reliable client/server communication

- In addition to process failure, there are communication failures (also known as link failures, or partitionings)
- Almost all failures can be reduced to crash or omission failures
  - Also known as missing clients/servers, or lost requests respectively
- Crashes: exception handling
- Ommisions/lost requests: message acknowledgments and timeouts
- Stateful and stateless servers (again)
  - Stateful servers: client/server communication cannot proceed until all client-specific information is recovered
  - Stateless servers: server continues working in the absence of or after losing client-specific information
    - Possibly less efficient, but functional nevertheless

Recovery

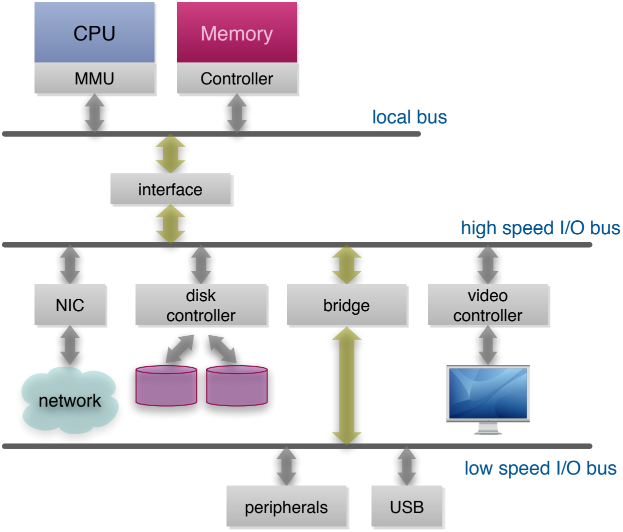
- Okay, we have failed; now what?
- Recovery: bringing a failing process to a correct state
- Backward recovery: return the system to some previous correct state and then continue
    - Take checkpoints: a consistent snapshot of the system
        - Expensive to take, need global coordination
        - When do we get rid of a checkpoint? In case of failure, how far back in time will be have to go?
    - Example: retransmission of lost packets
- Forward recovery: bring the system to a correct state and then continue
    - Account for all potential errors upfront
    - For every possible error, come up with a recovery strategy
    - Apply recovery strategies and bring the system to a correct state
    - Really tricky to implement and only for specific protocols
    - Example: self-correcting codes, reconstruction of damaged packets
- Backward recovery is implemented more often than forward recovery

Outline

Overview

- One of the most important techniques for the separation of hardware, operating system, and applications
- Various instances of virtualisation used every day without even knowing (hey, it's virtual after all!)
- Started back in 1964 with IBM's CP-40, a "*virtual machine/virtual memory time sharing operating system*"
- Key ideas: abstraction and well-defined interfaces
  - These interfaces can be implemented differently for different platforms (think Java)
  - Or emulated by the host platform (think VMWare)
- We will focus on three types of virtualisation
  - CPU, memory, and device (I/O)

CPU s and computer architecture



What's in a CPU and how can we virtualise it?

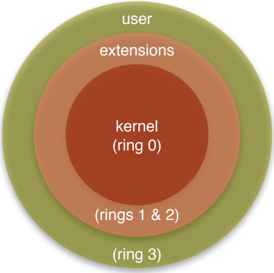
- It all comes down to one thing: the Instruction Set Architecture (ISA)
  - State visible to the programmer (registers, volatile memory)
  - Instructions that operate on the state
- Divided into two parts
  - User ISA used for developing/executing user programs (go wild, you can't break the system from here)
  - System ISA used mainly by the kernel for system resource management (careful)
- Most CPU virtualisation techniques focus on the ISA
  - System ISA virtualisation, instruction interpretation, trap and emulate, binary translation, hybrid models

User ISA: state and instructions

- State captures the various components of the system
  - Virtual memory (physical, swap)
  - Special purpose registers (program counter, conditions, interrupts)
  - General purpose registers (this is the actual data that is manipulated)
  - ALU floating point registers (mathematical operations)
- Instructions capture the current parameters of each stage in the processor's pipeline
  - Typically: fetch, decode, access registers, memory, write-back
  - One instruction per stage
  - Mutiple instructions in the pipeline, at different stages

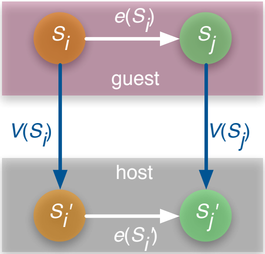
System ISA: where it all takes place

- Privilege levels (or rings)
- Control registers of the processor
- Processor and/or operating system traps and interrupts
  - Hardcoded vectors (non-maskable interrupts and standard handlers)
  - Dispatch table (extension interrupt handlers)
- System clock
- Memory management unit
  - Page table, translation lookaside buffer
- Device I/O



kernel + extensions = system

The CPU virtualisation isomorphism



Formal definition

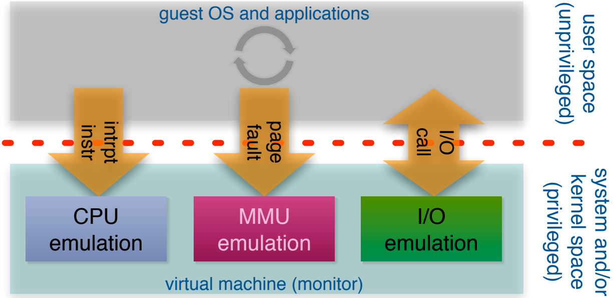
- Virtualisation is the construction of an isomorphism from guest state to host state
  - Guest state S_i is mapped onto host state S_i' through some function $V() : V(S_i) = S_i'$
  - For every transformation $e()$ between states S_i and S_j in the guest, there is a corresponding transformation $e'()$ in the host such that $e'(S_i') = S_j'$ and $V(S_j) = S_j'$
  - Virtualisation implements $V()$ and the translation of $e()$ to $e'()$

Virtualising the System ISA

- Key concept: the virtualisation monitor (or hypervisor)
  - This is the actual implementation of the virtual machine
  - The guest assumes complete control of the hardware
  - But that is not possible — in fact, it's a security breach
  - So the monitor supervises[3] the guest and virtualises calls to the guest's System ISA
  - Retargets them for the host
- Methodology is straightforward
  - Whenever the guest accesses the System ISA , the monitor takes over
  - Monitor maintains guest system state and transforms it whenever necessary
  - Guest system instructions are implemented as monitor functions affecting the host
  - Two-fold goal
    - Normal instructions are executed natively
    - Privileged instructions are isolated from the host

[3] It's called a hypervisor and not a supervisor because it might be monitoring more than one guests.

Trap and emulate



- Not all architectures support “trap and emulate” virtualisation
  - Most current CPU s have direct virtualisation hooks
- Trapping costs might be high (more calls than necessary)
- Virtual monitor runs at a higher privilege level
  - For instance, the Linux kernel only supports rings 0 (kernel) and 3 (user) though extensions like `kvm` solve the problem

Other types of CPU virtualisation

- Binary translation
  - Either compile programs to an intermediate representation and interpret it
    - Java (bytecode), `llvm` (virtual processor)
    - Implement the entire runtime multiple times for different platforms
  - Or, transform on-the-fly the natively compiled binary code
    - Very error-prone and hard to get right, especially when shifting between architectures
- Hybrid models
  - Solid parts of the system are binary translated (*e.g.,* kernel functionality)
  - User code is trapped and emulated

But where is the monitor?

- The virtual machine monitor is yet another process
  - Shares the same virtual address space with the address space it is virtualising (!)
- As with CPU virtualisation, it handles specific interrupts (page faults)
- If using trap-and-emulate CPU virtualisation the situation is somewhat easier
  - The monitor only needs to be protected from guest accesses
  - Easy; run in host kernel/extension level
  - Monitor specific ranges of the virtual address space to identify if a memory request needs to be resolved or not; offload others to host OS
- For binary translation need a memory model distinguishing between host (priviliged, non-translated) and guest (unprivileged, translated) accresses
- Hardware-support: segmentation on x86 architectures
  - Monitor in dedicated memory region
  - Guest cannot see monitor's region

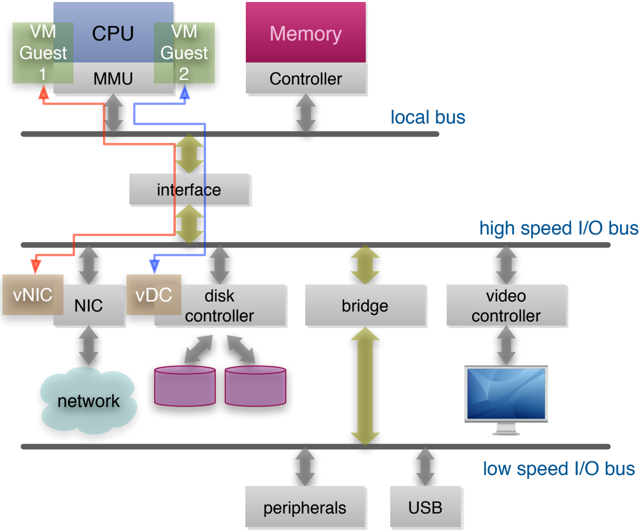
One step further out

- CPU virtualisation
  - Execute instructions developed for one CPU on another one
- Memory virtualisation
  - Allow multiple guest operating systems and their applications to see the same memory address space
  - Executed by a host operating system on a host CPU
- Both of them are a good start; but full-fledged systems access devices as well
  - A device is anything that can perform I/O
  - Hard-disk drives, displays, peripherals, you name it

Why virtualise I/O and how?

- Uniformity and isolation
  - A disk should behave like a single local disk regardless of whether it is remote or a RAID
  - Devices isolated from one another; they operate as if they were the only device around
- Performance and multiplexing
  - Let lower-level entities optimise the I/O path; they know how to do things better than explicit read/writes
  - Parallelise the process (*e.g.,* when replicating data)
- System evolution and reconfiguration
  - Taking the system offline to connect a new drive, or repair a damaged one is no longer an option
- Techniques: direct access, emulation, paravirtualisation

Direct access



Virtualisation through direct access

Advantages

- No changes to guest, same operation is what it was designed for
- Easy to deploy
- Simple monitor: only implement drivers for the virtual hardware

Virtualisation through direct access

Advantages

- No changes to guest, same operation is what it was designed for
- Easy to deploy
- Simple monitor: only implement drivers for the virtual hardware

Disadvantages

- Cannot happen without specialised hardware
- Need to make the hardware interface visible to the guest
  - We just lost extensibility
- Different hardware, different drivers
  - Guest needs to cater for all possible drivers (not only the real ones, but the virtual ones as well!)
- Too much reliance on the hardware for software-related operations (*e.g.*, scheduling, multiplexing, *etc.*)

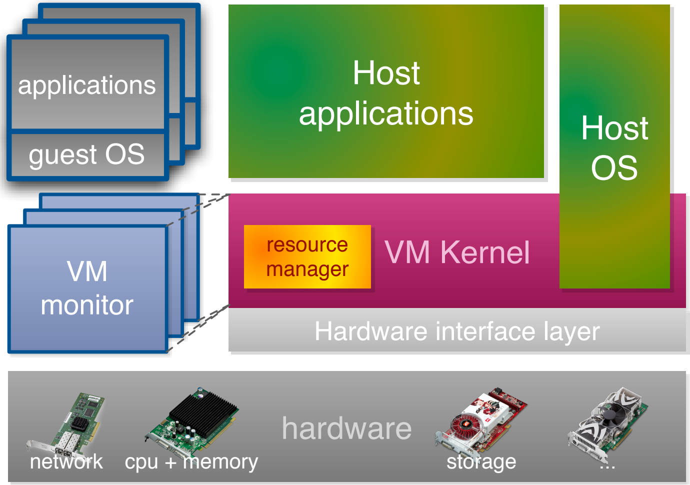
Device emulation

- Just as before, introduce an abstraction layer
  - Per class of device, *e.g.,* for all disk drives
  - Implement the abstraction for different instances of the device *e.g.,* drivers for disk interfaces, types of disk (HDD, solid state, . . .)
- Advantages
  - Device isolation
  - Stability: guest needs to operate just as before
  - Devices can be moved freely and/or reconfigured
  - No special hardware; all at the monitor level
- Disadvantages
  - The drivers need to be in the monitor or the host
  - Potentially slow: path from guest to device is longer
  - Possibility of duplicate effort: different drivers for the guest, different drivers for host

Paravirtualisation

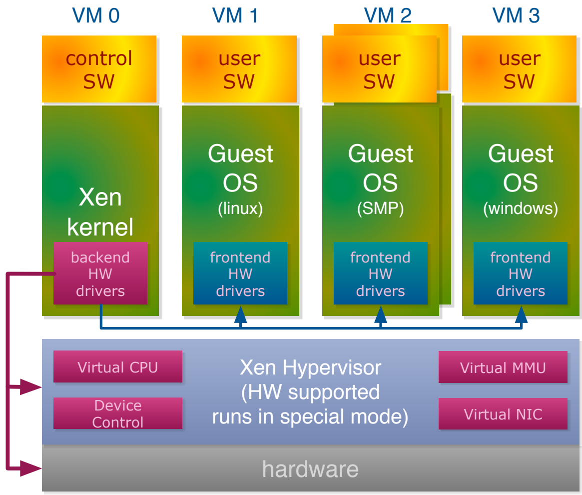
- The solution most contemporary virtual machine monitors use
- Effectively, reverse the direction of the communication
  - Instead of trapping guest calls and emulating them by translating them for the host
    - Expose the monitor and allow guest to make monitor calls
    - Implement guest-specific drivers
    - Implement the drivers once for each device at the monitor
- Advantages
  - Monitor now becomes simpler (and simple usually equals fast)
  - No duplication
- Disadvantages
  - We still need drivers, but now drivers for the guest
  - Bootstrapping becomes an issue: can't host a guest operating ststem until there are drivers available

The design of VMWare ESX 2.0[4]



[4] Adapted from `http://vmware.com/pdf/esx_2_performance_implications.pdf`

The hybrid design of the Xen hypervisor



- Pararvirtualisation for Linux guests
- Hardware-virtualisation for Windows
- Single implementation of device drivers, single access to hardware

Outline

The world is going parallel

- To be fair, the world was always parallel
  - We just didn't comprehend it, didn't pay attention to it, or we were not exposed to the parallelism
- Parallelism used to be elective
  - There were tools (both software and hardware) for parallelism and we could choose whether to use them or not
  - Or, special problems that were a better fit for parallel solutions
- Parallelism is now enforced
  - Massive potential for parallelism at the infrastructure level
  - Application programmers are forced to think in parallel terms
  - Multicore chips; parallel machines in your pocket

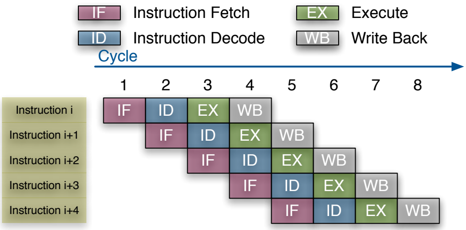
Implicit *vs.* explicit parallelism

- High-level classification of parallel architectures
- Implicit: there but you do not see it
  - Most of the work is carried out at the architecture level
  - Pipelined instruction execution is a typical example
- Explicit: there and you can (and had better) use it
  - Parallelism potential is exposed to the programmer
  - When applications are not implemented in parallel, we're not making best use of the hardware
  - Multicore chips and parallel programming frameworks (*e.g.,*MPI, MapReduce)

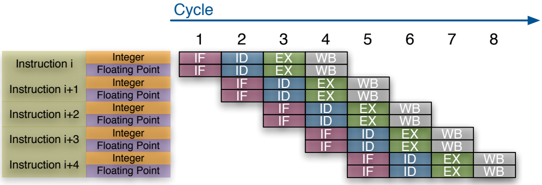
Implicit parallelism through superscalar execution

- Issue varying number of instructions per clock tick
- Static scheduling
  - Compiler techniques to identify potential
  - In-order execution (*i.e.,* instructions are not reordered)
- Dynamic scheduling
  - Instruction-level parallelism (ILP)
  - Let the CPU (and sometimes the compiler) examine the next few hunderds of instructions and identify parallelism potential
  - Schedule them in parallel as operands/resources become available
  - There are plenty of registers; use them to eliminate dependencies
  - Execute instructions out of order
    - Change the order of execution to one that is more inherently parallel
  - Speculative execution
    - Say there is a branch instruction but which branch will be taken is uknown at issue time
    - Maintain statistics and start executing one branch

Pipelining and superscalar execution



standard pipelining



2-issue superscalar architecture

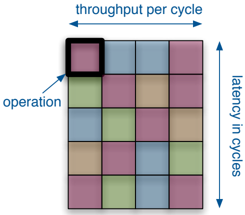
ILP, data dependencies, and hazards

- CPU and compiler must preserve program order: the order in which the instruction would be executed if the original program were executed sequentially
  - Study the dependencies among the program's instructions
- Data dependencies are important
  - Indicated the possibility of a hazard
  - Determines the order in which computations should take place
  - Most importantly: sets an upper bound on how much parallelism can possibly be exploited
- Goal: exploit parallelism by preserving program order only where it affects the outcome of the program

Speculative execution

- Greater ILP: overcome control dependencies by speculating in hardware the outcome of branches and executing the program as if the guess were correct
  - Dynamic scheduling: fetch and issue instructions
  - Speculation: fetch, issue, and execute a stream of instructions as if branch predictions were correct
- Different predictors
  - Branch predictor: outcome of branching instructions
  - Value predictor: outcome of certain computations
  - Prefetching: lock on to memory access patterns and fetch data/instructions
- But predictors make mistakes
  - Upon a mistake, we need to empty the pipeline(s) of all erroneously predicted data/instructions
  - Power consumption: more circuitry on the chip means more power, which means more heat

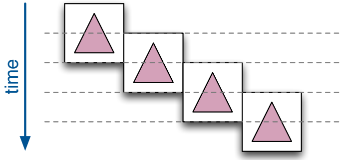
Explicit parallelism



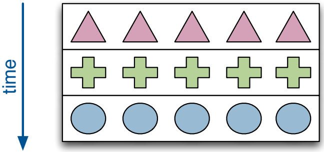
- Parallelism potential is exposed to software
  - Both to the compiler and the programmer
- Various different forms
  - From loosely coupled multiprocessors to tightly coupled very long instruction word architetures
- Little's law: parallelism = throughput $\times$ latency
  - To maintain a throughput of T operations per cycle when each operation has a latency of L cycles, we need to execute $T \times L$ independent operations in parallel
  - To maintain fixed parallelism
    - Decreased latency results in increased throughput
    - Decreased throughput allows increased latency

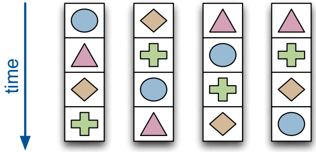
Types of parallelism

pipelined parallelism: different operations in different stages

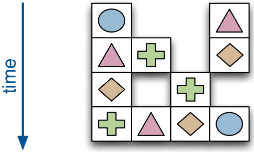


data-level parallelism: same computation over different data





thread-level parallelism: different threads performing different computations



instruction-level parallelism: no two threads performing the same computation at the same time

Enforced parallelism

- The era of programmers not caring what's under the hood is over
- To gain performance, we need to understand the infrastructure
- Different software design decisions based on the architecture
- But without getting too close to the processor
  - Problems with portability
- Common set of problems faced, regardless of the architecture
  - Concurrency, synchronisation, type of parallelism

Concurrent programming

- Sequential program: a single thread of control that executes one instruction and when it is finished it moves on to the next one
- Concurrent program: collection of autonomous sequential threads executing logically in parallel
  - The physical execution model is irrelevant
  - Multiprogramming: multiplexed threads on a uniprocessor
  - Multiprocessing: multiplexed threads on a multiprocessor system
  - Distributed processing: multiplexed processes on different machines
- Concurrency is not only parallelism
  - Difference between logical and physical models
  - Interleaved concurrency: logically simultaneous processing
  - Parallelism: physically simultaneous processing

Reasons for concurrent programming

- Natural application structure
  - World is not sequential; easier to program in terms of multiple and independant activities
- Increased throughput and responsiveness
  - No reason to block an entire application due to a single event
  - No reason to block the entire system due to a single application
- Enforced by hardware
  - Multicore chips are the standard
- Inherent distribution of contemporary large-scale systems
  - Single application running on multiple machines
  - Client/server, peer-to-peer, clusters
- Concurrent programming introduces problems
  - Need multiple threads for increased performance without compromising the application
  - Need to synchronise concurrent threads for correctness and safety

Synchronisation

- To increase throughput we interleave threads
- Not all interleavings of threads are acceptable and correct programs
  - Correctness: same end-effect as if all threads were executed sequentially
- Synchronisation serves two purposes
  - Ensure safety of shared state updates
  - Coordinate actions of threads
- So we need a way to restrict the interleavings explicitly at the software level

Thread safety

- Multiple threads access shared resource simultaneously
- Access is safe if and only if
  - Accesses have no effect on the state of the resource (for example, reading the same variable)
  - Accesses are idempotent (for example $y = x^2$)
  - Accesses are mutually exclusive
    - In other words, accesses are serialised

	You	**Your roomate**
3:05	Arrive home	
3:10	Look in fridge, no milk	
3:15	Go to supermarket	
3:20		Arrive home
3:25	Arrive supermarket, buy milk	Look in fridge, no milk
3:30		Go to supermarket
3:35	Arrive home, put milk in fridge	
3:40		Arrive supermarket, buy milk
3:45		
3:50		Arrive home, put milk in fridge

resources not protected: too much milk

Mutual exclusion and atomicity

- Prevent more than one thread from accessing a critical section at a given time
  - Once a thread is in the critical section, no other thread can enter the critical section until the first thread has left the critical section
  - No interleavings of threads within the critical section
  - Serialised access
- Critical sections are executed as atomic units
  - A single operation, executed to completion
  - Can be arbitrary in size (*e.g.,* a whole block of code, or an increment to a variable)
- Multiple ways to implement this
  - Semaphores, mutexes, spinlocks, copy-on-write, . . .
  - For instance, the `synchronized` keyword in Java ensures only one thread accesses the synchronised block
  - At the end of the day, it's all locks around a shared resource
    - Shared or no lock to read resource (multiple threads can have it)
    - Exclusive lock to update resource (there can be only one)

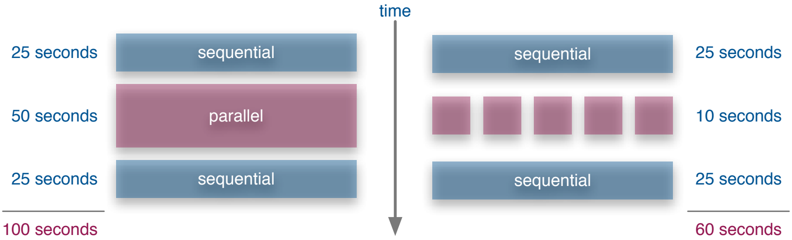
Potential concurrency problems

- Deadlock: two or more threads stop and wait for each other
  - Usually caused be a cycle in the lock graph
- Livelock: two or more threads continue to execute but they make no real progress toward their goal
  - Example, each thread in a pair of threads undoes what the other thread has done
  - Real world example: the awkward walking shuffle
    - Walking towards someone, you both try to get out of each other's way but end up choosing the same path
- Starvation: some thread gets deferred forever
  - Each thread should get a fair chance to make progress
- Race condition: possible interleaving of threads results in undesired computation result
  - Two threads access a variable simulatneously and one access is a write
  - A mutex usually solves the problem; contemporary hardware has the compare-and-swap atomic operation

Parallel performance analysis

- Extent of parallelism in an algorithm
  - A measure of how parallelisable an algorithm is and what we can expect in terms of its ideal performance
- Granularity of parallelism
  - A measure of how well the data/work of the computation is distributed across the processors of the parallel machine
  - The ratio of computation over communication
  - Computation stages are typically separated from communication by synchronization events
- Locality of computation
  - A measure of how much communication needs to take place

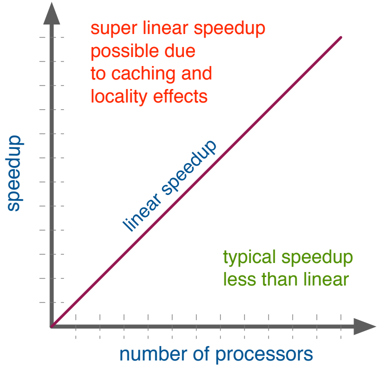
Parallelism extent: Amdahl's law

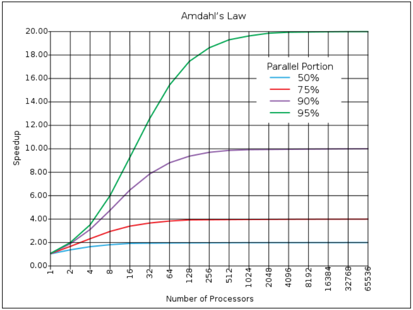


- Program speedup is defined by the fraction of code that can be parallelised
- In the example, speedup = sequential time / parallel time = 100 seconds / 67 seconds $= 1.67$
- speedup $= \frac{1}{(1-p)+\frac{p}{n}}$
  - p: fraction of work that is parallelisable
  - n: number of parallel processors

Implications of Amdahl's law

- Speedup tends to be $\frac{1}{1-p}$ as number of processors tends to infinity
- Parallel programming is only worthwhile when programs have a lot of work that is parallel in nature (so called embarrassingly parallel)





Fine *vs.* coarse granularity

Fine-grain parallelism

- Low computation to communication ratio
- Small amounts of computation between communication stages
- Less opportunity for performance enhancement
- High communication overhead

Coarse-grain parallelism

- High computation to communication ratio
- Large amounts of computation between communication stages
- More opportunity for performance enhancement
- Harder to balance efficiently

Load balancing

- Processors finishing early have to wait for the processor with the largest amount of work to complete; leads to idle time, lower utilisation
- Static load balancing: programmer makes decisions and fixes *a priori* the amount of work distributed to each processor
  - Works well for homogeneous parallel processing
    - All processors are the same
    - Each core has an equal amount of work
  - Not so well for heterogeneous parallel processing
    - Some processors are faster than others
    - Difficult to distrubute work evenly
- Dynamic load balancing: when one processors finishes its allocated work it takes work from processor with the heaviest workload
  - Also known as task stealing
  - Ideal for uneven workloads and heterogeneous systems

Outline

Infrastructure

- Overview
- Distributed file systems
- Replication and fault tolerance
- Virtualisation
- Parallelism and parallel/concurrent programming
- Services

What it is all about

- Service-oriented architecture (SOA)
  - Another name for large scale components wrapped behind a standard interface
  - Not Web Services; these are just a possible instantiation
  - Again, the ideas are not new; SOA is intended as a reference to application-building
    - Builds on previous ideas such as (software bus, enterprise bus, ...)
- Loosely-coupled: the services are independent of each other, heterogeneous, distributed
- Message-based: interaction through messages rather than through direct calls (unlike Web services, CORBA, RPC)
  - Raise your hand if you're thinking MPI

So where's the novelty?

- Blast from the past
  - Message-oriented middleware is not new
  - Message brokering is not new
  - Event-based developement is not new
- What is different is the context, need, and maturity of the approach
  - By definition, our needs are getting more complex
  - Emergence of standard interfaces
  - Development needs to be simplified; skip the middle-man
  - Use of complex underlying infrastructure
- Why is it interesting now?
  - Basic technology in place
  - We are only now starting to understand truly distributed applications
  - The key problem is integration not programming

Infrastructure as a service

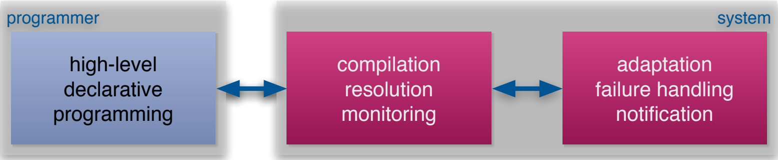
If you build it, they will come

SOA ≠ Web services

- Web services were the hype about
  - Interoperability
  - Standardization
  - Integration across heterogeneous, distributed systems
- Service-oriented architectures are more general
  - Large-scale software design and engineering
  - Grassroots architecture of distributed systems
- SOA is possible without Web services
  - Just a little more difficult
- The introduced changes and challenges
  - Language independence (care about the interface, not the implementation)
  - Event-based interaction (synchronous models are dead)
  - Message-based exchanges (RPC s dead too)
  - Composability, composition and orchestration

Plumbers are expensive

- The promise of SOA is to facilitate integration by
  - Letting the system automatically decide on integration decisions
    - Protocols to use
    - Intermediate processing needed
    - Routing and distribution
    - Data transformations
  - Enforcing standards for defining and operating with services
  - Enforcing the use of service interfaces
- Automatic software and plumbing generation
  - Contracts and service-level agreements
  - Agree on the what; let the infrastructure deal with the how



Infrastructure as a service

- Service contracts involve the interface, the service-level agreement and the quality-of-service guarantee
- Based on contracts we can develop, debug, optimise and maintain systems developed as a combination of services
- Service contracts are not the static, compile-time pre- and post-conditions of conventional programming languages
- Additional software layer in charge of the dynamic aspects of using services
- Run-time software engineering
  - Reconfigure the system to abide by the contract

The move to utility computing

Conventional computing

- Permanent residence
- Static setup
- Hardware changes rarely
- Software does not move
- Fixed network infrastructure

The move to utility computing

Conventional computing

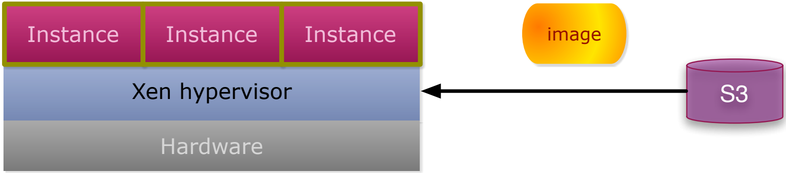
- Permanent residence
- Static setup
- Hardware changes rarely
- Software does not move
- Fixed network infrastructure

Utility computing

- Temporary residence
- Dynamic setup
- Hardware changes often
- Software can move
- Dynamic network infrastructure
- No special machines

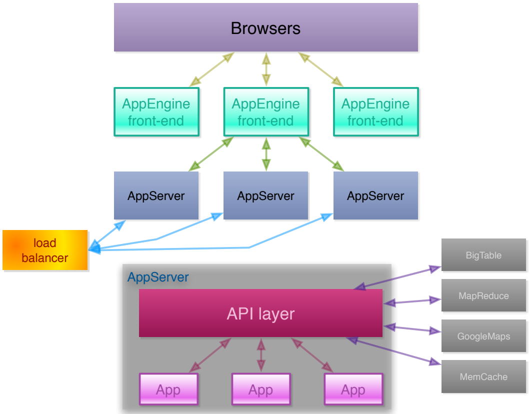
- Why the move?
  - Structured to support micro-billing (higher profit margins)
  - Utility is on-demand and lightweight
  - Minimal staff requirements from host's perspective (lower cost)

What does it look like? (Amazon EC2 + S3)



- EC2 : elastic cloud computing
- S3 : simple storage service
- System image (guest OS, libraries, applications) is transferred from S3
- Starts being executed be the hypervisor; becomes an instance of a virtual machine

Another example: Google AppEngine



- Python development and runtime
- Bindings for multiple other languages
- Storage based on BigTable (optimisation via MemCache)
- Plenty of API s
- Per-use billing, transparent scaling

Challenges

- Where's the code?
  - Where does the software actually run?
  - How does one software component locate another?
  - How do we deploy and monitor software components?
- Where is our data kept?
  - Not what we are used to
  - File- or resource-oriented storage
  - Databases are horribly difficult to deploy
    - The architecture of the standard DB server is not conducive
    - Persistent storage is often a remote service (and slow)
    - Storage at the instance level is often transient (and fast)
- How do we refer to system nodes?
  - DNS is useless because names and IP s change
- How do we configure our software?
  - Dynamic configuration: nothing machine-specific; nothing defined prior to runtime

Potential solutions through composition of services

- Naming service
  - Available to all instances
  - Can be dynamically discovered and accessed at runtime
  - Registry of service components
- Deployment service
  - Single base operating system image with minimal footprint
  - One per machine
  - Referred to and found via the naming service
  - Package deploy/undeploy
    - Contain mixture of low-level services and resources
- Software service
  - Has a lifecycle
  - Auto-redeployed on failure
    - Fully decentralised and fault-tolerant solution
  - Machine-level service to expose operational aspects of machines for monitoring

The name of the (current) game

- Storage
    - Bridge gap between fast and slow
    - Allow automatic migration in face of failure
    - Need to be able to “freeze” it to offline storage at shutdown
        - Distributed file systems (*e.g.*,HDFS)
    - Querying and processing concepts
        - Hadoop/MapReduce/Pig/Sawzall
- Scalable parallel computing
    - High concurrency potential
    - Antidote to variations in inter- node network latency
    - Need more processing power? Add more nodes!
- Infrastucture as a service
    - Cheaper: enterprises starting to pay attention
    - Simpler: dynamic infrastructure with less configuration

You got served[5]

David Wheeler

- All problems in computer science can be solved by another level of indirection

[5] Horrible 2004 film, but South Park kind of saved it (episode 115).

You got served[5]

David Wheeler

- All problems in computer science can be solved by another level of indirection
  - . . . except for the problem of too many levels of indirection

[5] Horrible 2004 film, but South Park kind of saved it (episode 115).