Advanced Systems Topics

Steven Hand

Lent Term 2010

Course Aims

- This course aims to help students develop and understand complex systems and interactions, and to prepare them for emerging systems architectures.
- It will cover a selection of topics including:
  - internet routing protocols,
  - operating systems,
  - database systems,
  - distributed storage systems,
  - mobile and ad-hoc systems, and
  - architecture and applications of sensor networks
- On completing the course, students should be able to
  - describe similarities and differences between current Internet routing protocols
  - describe three techniques supporting extensibility
  - argue for or against distributed virtual memory
  - discuss the challenges of sensor networking

Course Outline

- Part I: Internet Routing Protocols [TGG, 6L]
  - Internet as a Distributed System
  - Intra-domain routing (RIP, OSPF, ISIS)
  - BGP: path vectors and live-lock.
  - Convergence, scalability and stability.
- **Part II: Advanced Operating Systems [SMH, 6L]**
  - **Distributed & Persistent Virtual Memory**
  - **Microkernels & Extensible Operating Systems**
  - **Virtual Machine Monitors**
  - **Distributed Storage [3L]**
- Part III: Mobile and Sensor Systems [CM, 4L]
  - Introduction
  - Mobile & Ad Hoc Systems
  - Sensors: Challenges and Applications

Recommended Reading

- Singhal & Shivaratri, **Advanced Concepts in Operating Systems**, McGraw-Hill, 1994
- Stonebraker & Shivaratri, **Readings in Database Systems**, Morgan Kaufmann (3rd ed.), 1998
- Bacon and Harris, **Operating Systems**, Addison Wesley, 2003
- Hennessy & Patterson, **Computer Architecture: a Quantitative Approach**, Morgan Kaufmann, 2003

- Additional links and papers (via course web page)
  - **www.cl.cam.ac.uk/Teaching/current/AdvSysTop/**

Process Communication Models

- Two primary models for communication in concurrent / parallel programs:
- 1. **Shared memory model**:
  - collection of "threads" sharing address space
  - reads/writes on memory locations implicitly and immediately globally visible
  - e.g. x := x + 1
- 2. **Message passing model:**
  - collection of "processes" (private address spaces)
  - explicit coordination through messages, e.g

Process1		*Process 2*
send_msg(FETCH, "x", P2);	→	receive(&msg);
	←	send_msg(VALUE, "x", x, P1);
temp := receive_val(&msg);		
temp := temp +1;		
send_msg(VALUE, "x", temp, P2);	→	x = receive_val(&msg);

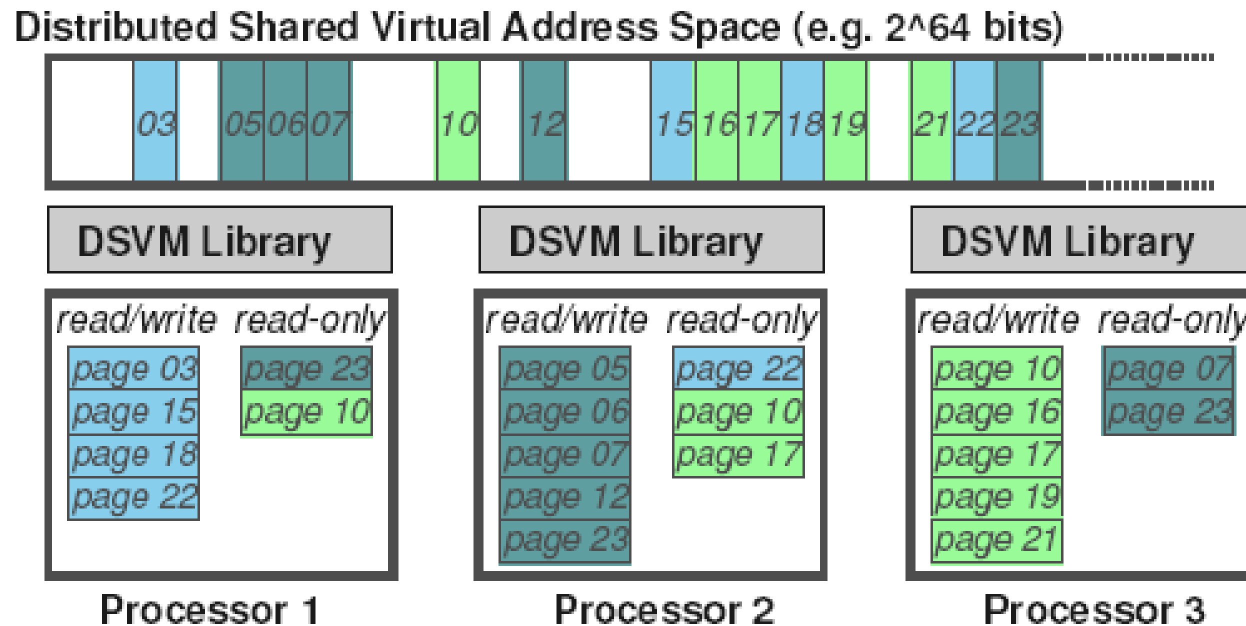
Process Communication Models

- Both have advantages and disadvantages...
- **Message passing**:
  - control & protection: separate address spaces means sharing only happens when required
  - performance: communicate min amount of data
  - **BUT**: verbose, complicated, ugly ;-)
- **Shared memory**:
  - ease of use: just read and write variables
  - transparent scalability: just add processes
  - **BUT**: race conditions, synchronisation, cost

Distributed Shared Virtual Memory

- Memory model typically dictated by hardware:
  - shared memory on tightly-coupled systems,
  - message passing on loosely-coupled systems
- Radical idea: provide shared memory on clusters!
  - each page has a “home” processor
  - can be mapped into remote address spaces
  - on read access, page in across network
  - on write access, sort out ownership...
- OS/DSVM library responsible for:
  - tracking current ownership
  - copying data across network
  - setting access bits to ensure coherence

DSVM Model



- **All processors share a single virtual address space**
  - this can be extremely large, e.g. 64-bits
- Physical address space is the aggregate of all local memory
- Mapping from virtual to physical managed by DSVM layer
  - ensure at most 1 read/write copy of any page, but can safely have multiple read-only copies
- (basically same as multi-processor cache coherence protocols)

Implementing DSVM

- **Simplest case: centralized page manager**
  - single processor maintains 2 per-page data structures
  1. **owner(p)** = the processor P that created – or which last wrote to – the page p
  2. **copyset(p)** = all processors with a copy of p
     - can store copyset as a bitmap to save space
- Then on a **read fault** need four messages:
  - (1) contact manager; (2) manager forwards to owner; (3) owner sends page; (4) requester acks to manager
  - If successful, manager updates copyset(p)
- On a **write fault**, need a bit more work:
  - (1) contact manager; (2) manager invalidates copyset; (3) manager contacts owner; (4) owner relinquishes page; (5) requester acks to manager, who updates owner(p)
  - Note that (2) may require many messages (or b'cast?)

DSVM Optimizations

- **Static distributed management**:
  - Aim to load-balance: manager(p) is Hash(p)
  - Use centralized algorithm for each manager
- **Dynamic distributed management**:
  - Aim to reduce messages: manager(p) = owner(p)
  - Can broadcast to find manager(p) but needs care:
    - E.g. consider concurrent write faults on P1 & P2
    - P3 owns page: gets message from P1 & replies
    - P3 ignores message from P2 – no longer owner!
    - P1 also ignores message from P2 – not yet owner!
    - To fix need "atomic broadcast" (c/f distributed systems)

DSVM Optimizations

- A better solution is to keep per-processor hint:
  - Each processor P maintains **probOwner(p)** = the processor which P believes to be the owner
    - Initialized to a default value, e.g. Hash(p)
  - If P1 takes a fault on p, contacts P2=probOwner(p)
    - If correct, P2 sends P1 the page; otherwise P2 forwards the request to P3 = his probOwner(p)
    - In either case, if P1 requested write access, P2 updates his probOwner(p) := P1
    - Also update probOwner() if see a broadcast invalidate
  - Can also allow non-owner to reply to a read request if he has an up-to-date copy
    - Updates local copyset => need multi-stage invalidate

Weaker Consistency

- Even with optimizations, can be expensive, e.g. false-sharing:
  - P1 owns p, P2 just has read-access
  - P1 writes p -> copies to P2...
  - ... but P2 doesn't care about this change
- Can reduce traffic by using **weaker memory consistency**:
  - so far assumed sequential consistency: every read sees latest write
  - easy to use, but expensive
- Instead can do e.g. **release consistency**:
  - reads and writes occur locally
  - explicit acquire & release for synchronization
  - analogy with memory barriers in MP
- Best performance by doing **type-specific coherence**:
  - private memory -> ignore
  - write-once -> just service read faults
  - read-mostly -> owner broadcasts updates
  - producer-consumer -> live at P, ship to C
  - write-many -> release consistency & buffering
  - synchronization -> strong consistency

DSVM: Evolution & Conclusions

- **mid 1980's**: IVY at Princeton (Li)
  - sequential consistency (used probOwner(), etc)
  - some nice results for parallel algorithms with large data sets
  - overall: too costly
- **early 1990's**: Munin at Rice (Carter)
  - type-specific coherence
  - release consistency (when appropriate)
  - allows optimistic multiple writers
  - almost as fast as hand-coded message passing
- **mid 1990's**: Treadmarks at Rice (Keleher)
  - introduced “lazy release consistency”
  - update not on release, but on next acquire
  - reduced messages, but higher complexity
- **On clusters**:
  - **can always do better with explicit messages**
  - complexity argument fails with complex DSVM
- **On non-ccNUMA multiprocessors: sounds good!**

Persistence

Why is virtual memory volatile?

- Virtual memory means memory is (or at least may be) backed by non-volatile storage.
- Why not make this the default case?
  - no more distinction between files and memory
  - easier programmatic access to file system / DB
  - can benefit from type system
- **Orthogonal Persistence** => manner in which data is accessed is independent of how long it persists
- Two main options for implementation:
  1. Functional/interpreted languages: fake out in runtime.
  2. Imperative/compiled languages:
  - prescribe way to access data (e.g. pure OO), or
  - use the power of virtual memory...

Persistent Virtual Memory

- Actually a very old idea: e.g. Multics:
  - developed 1964- by MIT, GE and AT&T Bell Labs
  - no filesystem; user saw a number of **segments** = orthogonal regions of virtual address space:
    - backed by non-volatile secondary store
    - created and named by users
    - remained available until explicitly deleted
  - tree of **directories** and non-directories (c/f Unix)
    - directories contain set of **branches** (~= inodes)
    - branches contain ACL plus ring bracket (b1 <= b2)
    - branches also contain limit (l >= b2) and list of **gates**
    - process running within limit can 'jump' through a gate

Persistent Virtual Memory

- Multics was not successful (for a number of reasons!), but persistent VM idea lived on
- As size of secondary storage grew, it became impossible to directly name (refer to) all data
- One possible solution was **pointer swizzling**:
  - e.g. the Texas portable C++ library
  - can allocate objects on a special **persistent heap**
  - data in persistent pages canonically addressed by special 64-bit **persistent pointers** (PPtrs)
  - ensure PPtrs are never directly accessed:
    - mark any resident persistent page as invalid
    - trap on access and for every PPtr **p**
      - allocate a new page P and mark it invalid
      - swizzle (rewrite) **p** to refer to P
      - unprotect original page and resume

Recoverable Virtual Memory

- RVM provides a (subset of) transactional semantics to regions of virtual memory
  - building block for filesystems, DBs, applications.
  - best known work is lightweight RVM (SOSP '93)
- LRVM just considers **atomicity** and **durability**
- Processes map [32-bit] regions of persistent [64-bit] segments into their virtual address space, and then:
  - Start with t = begin_transaction(rmode)
  - Invoke one or more set_range(t, base_addr, nbytes)
  - Finally end_transaction(t, cmode)
- Unless rmode is 'norestore' , LRVM copies the contents of the range(s) to an **undo log** => restore on abort
- On commit, write changes to **redo log**
  - synchronous write unless cmode is 'noflush'.
- LRVM looks-aside into the redo log => can lazily flush changes to segments on disk, and truncate log.

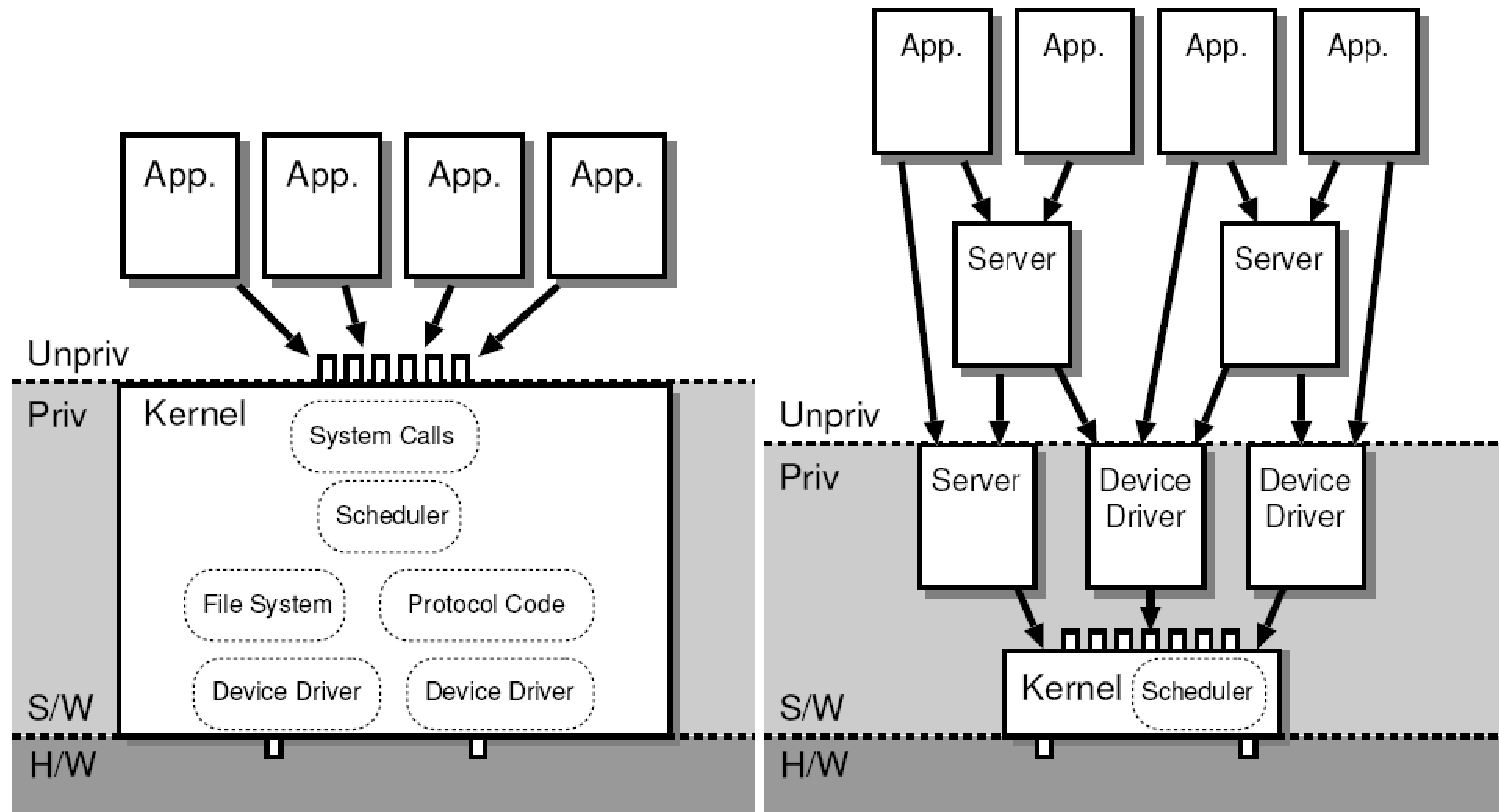
Making LRVM Faster…

- LRVM remarkably successful:
  - Less than 20% the size of previous systems, and about 2x-3x faster
- But still has (up to) 3 copies: undo, redo, trunc
- Rio Vista (SOSP'97) used NVRAM to optimize:
  - mapping a region is just mmap() on NVRAM
  - set_range() makes copy to NVRAM
  - no copy for redo (NVRAM) or truncate (NVRAM)
    - on reboot, flush NVRAM contents to disk
  - claimed 2000x speed up over LRVM
- Authors required asbestos trousers ;-)

OS Structures

- Earliest operating systems were “monolithic”
- 1970s: Unix pioneered notion of the **kernel**
  - Just the essentials in privileged mode
  - Everything else (e.g. shell, login, etc) a process
- Number of structures since then:
  - **microkernels**: put just the bare essentials in kernel, everything else in (priv or unpriv) servers
  - **extensible OSes**: let unpriv code run within kernel
  - **vertically structured OSes**: put just the bare essentials in kernel, everything else in unpriv libraries
  - **virtual machine monitors**: run entire OS unprivileged on top of privileged hypervisor

Kernels (lhs) & Microkernels (rhs)

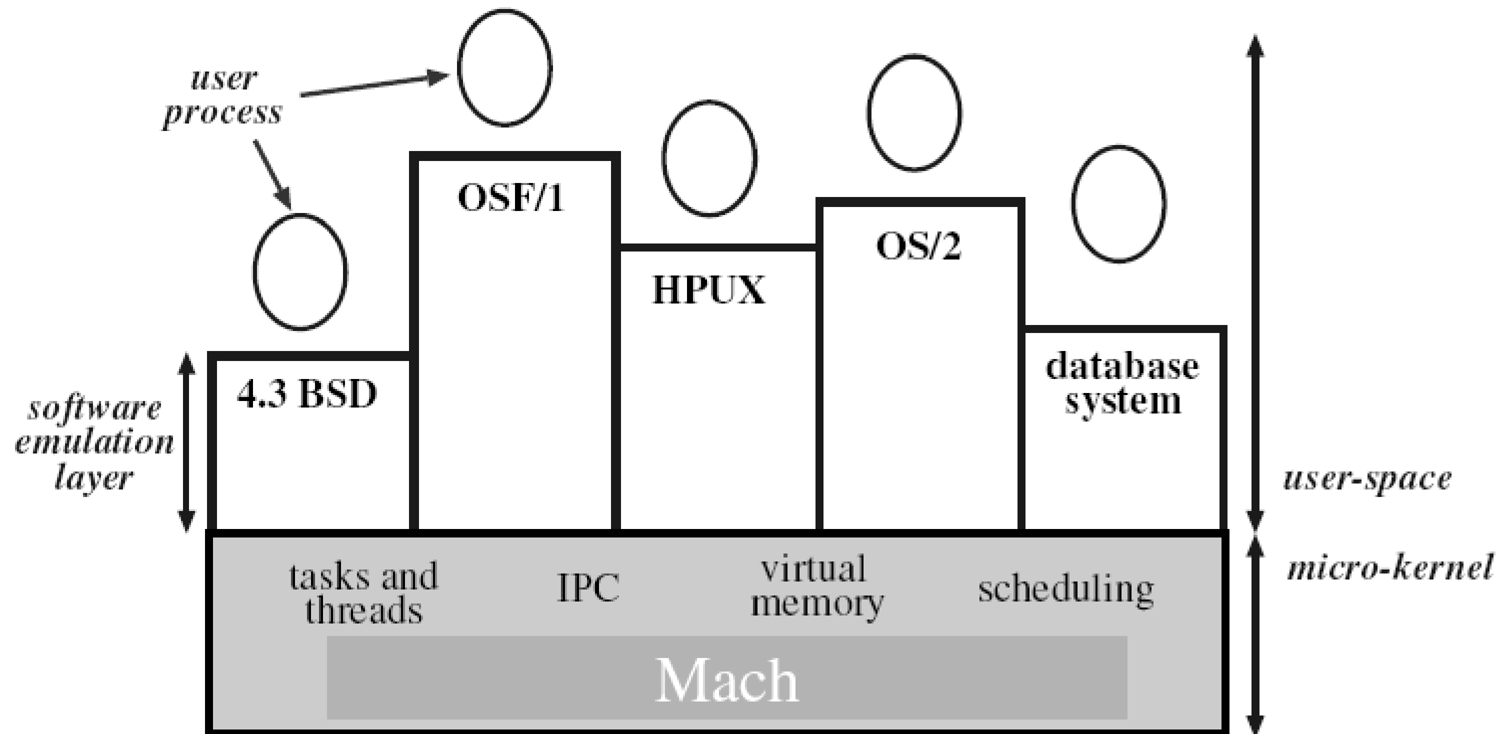


- New concept in early 1980‘s: simplify kernel
  - modularity: support multiprocessors, distributed computing
  - move functions to user-space servers, access servers via (IPC)

Microkernel Benefits

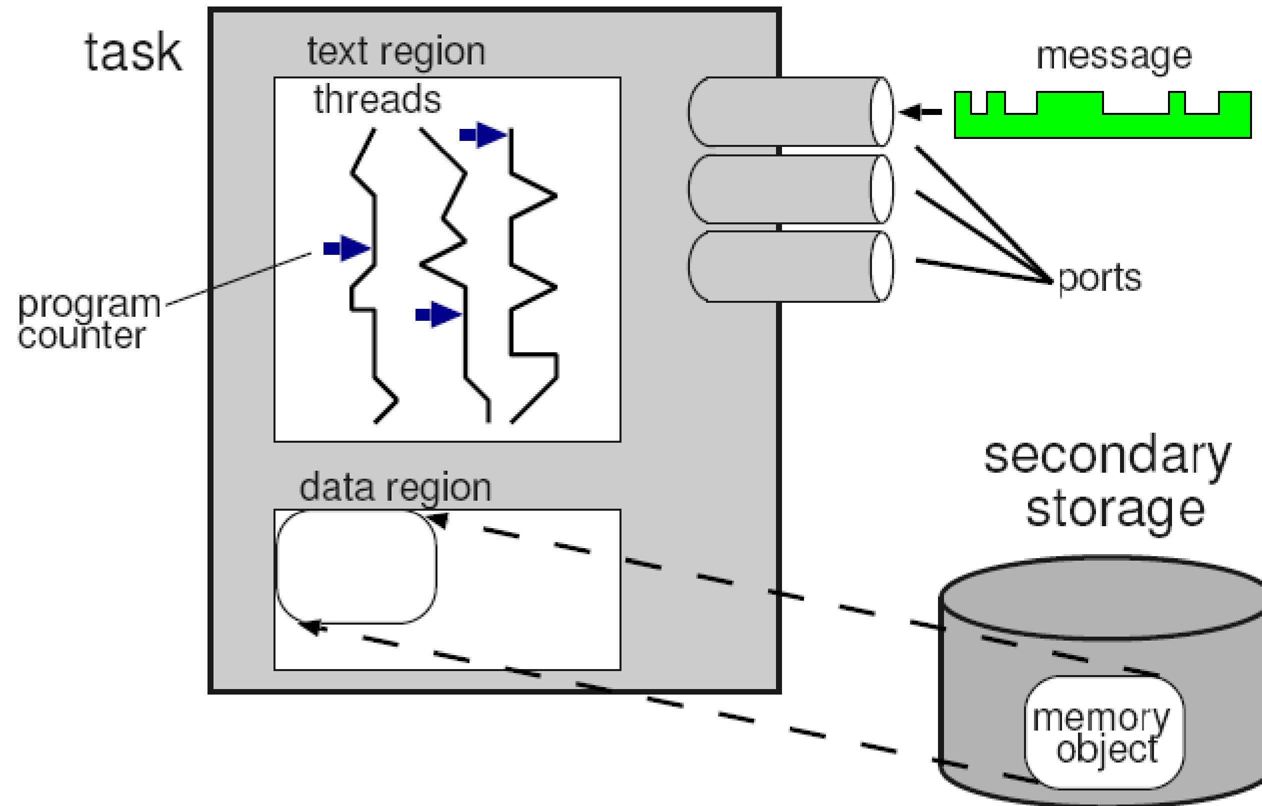
- **Multiprocessor support**:
  - Servers can be scheduled anywhere
  - Only require spin-locks in small microkernel
- **Real-time support**:
  - Small kernel allows predictable performance
  - Use RM / EDF (aperiodic tasks for best effort)
- **Modularity**:
  - Easy to replace – or evolve – functionality
  - (and get **extensibility** too via user-space servers)
- **Portability**:
  - Machine dependent code limited to kernel
- **Security**:
  - Small kernel easier to verify

The Mach Microkernel



- Developed at CMU 1985- (Rashid, Bershad, …)
- Aimed to support diverse architectures:
  - including multiprocessors & heterogeneous clusters!
  - hence used **message-based IPC** (as per RIG, Accent)
- Also targeted compatibility with 4.3BSD, OS/2, …

Mach Abstractions



- **Tasks** (unit of protection) & **Threads** (unit of scheduling)
- **IPC** based on **ports** and **messages**:
  - Port is a generic reference to a resource
  - Implemented as a buffered communications channel
  - IPC is **asynchronous**: message passing between threads

Mach Implementation

- All resources accessed via IPC
  - Protection achieved via port capabilities
  - send right, send-once right, receive right
    - send of a receive right “teleports” endpoint
  - Messages can be in-band if small, or passed by reference if larger (i.e. via virtual address region)
- (mostly) machine independent memory management via memory objects
  - Send IPC to memory object to satisfy page fault
  - Since message-based, can even be over network!
- Compatibility layers in unprivileged servers
  - Mach ‘reflects’ system calls via IPC upcalls

Microkernel Reality

- **Looks good on paper, but in practice performance was very poor**:
  - many user-kernel crossings => expensive
  - flexible asynchronous IPC introduces latency
  - machine-independent parts lack optimization
  - e.g. Chen (SOSP'93) compared Mach to Ultrix:
    - worse locality affects caches and TLBs
    - large block copies thrash memory system
- Other benefits all a bit “proof-by assertion”
  - e.g. ‘small kernel’ => simplicity, security, modularity, etc…
- Basic dilemma:
  - if too much in kernel, lose benefits
  - if too little in kernel, too costly
- By mid 90’s, most people had given up...

L3/L4: Making Microkernels Perform

- Liedtke (SOSP'95) claims that problems were a failure in **implementation**, not in concept
- To fix, you simply have to:
  1. **minimise what should be in the kernel; and**
  2. **make those primitives really fast**.
- The L3 (and L4, SOSP'97) systems provided just:
  - recursive construction of address spaces
  - threads and basic thread scheduling
  - synchronous local IPC
  - unique identifier support
- Hand-coded in i486 assembly code!

L3/L4 Design & Implementation

- Address spaces support by three primitives:
  1. **Grant**: give pages to another address space
  2. **Map**: share pages with another address space
  3. **Flush**: take back mapped or granted pages
- Threads execute with address space:
  - characterised by set of registers
  - micro-kernel manages thread -> address space binding
- IPC is synchronous message passing between threads:
  - highly optimised for i486 (3us vs Mach's 18us)
  - interrupts handled as messages too
- Does it work? '97 paper getpid() comparison:

System	Time	Cycles
Linux	1.68s	223
L4Linux	3.95s	526
MkLinux (Kernel)	15.41s	2050
MkLinux (User)	110.60s	14710

- Q: are these micro-benchmarks useful? what about portability?

Extensible Operating Systems

- **Extensibility** is about building an OS which can be "extended" (customized) at run time.
- Why do we care?
  - **Fixing mistakes**.
  - **Supporting new features** (or hardware).
  - **Efficiency**, e.g.
    - packet filters
    - run-time specialisation
  - **Individualism**, e.g.
    - per-process thread scheduling algorithms.
    - customizing replacement schemes.
    - avoiding "shadow paging" (DBMS)
- One of the major OS research themes of the 90's
  - (Presupposes microkernels are The Wrong Way™ ;-)

Kernel-Level Extensibility

- Most things can be handled just by allowed bits of code to be “downloaded” into kernel
  - e.g. Linux kernel modules
    - requires dynamic relocation and linking
    - support for [un]loading on demand
  - e.g. NT (XP, Win7) services and device drivers
    - well-defined entry / exit routines
    - can control load time & behaviour
- Main problem is that we don’t know what the hell this code will do!
  - A bad extension can crash/corrupt/subvert OS
  - Above OSes ignore this issue (root, administrator)
- Plus issues with specificity; and interface stability

Guaranteeing Safety

- General problem is extremely hard
  - viz. ensure extension code has no bugs (i.e. conforms to some [formal] specification)
  - also want to prove termination = tricky
- Some solutions include:
  - **Trusted Compiler & digital signatures** (+ CA)
    - Run extension code iff digital signature is kosher
    - Kinda sidesteps the real issues, but still useful
  - **Proof-Carrying Code**
  - **Sandboxing**
  - Using **Language-Based Safety**

Proof-Carrying Code

- **Take code, check it, and run iff checker says it's ok.**
  - "Ok" means cannot read, write or execute outside some logical fault domain (subset of kernel virtual address space)
  - (note that this is a quite weak notion of safety)
- **Problem: how do we check the code?**
  - generating proof on fly tricky + time-consuming...
  - ... so **expect proof supplied and just check proof**
- Overall can get very complex, e.g. need:
  - formal specification language for safety policy
  - formal semantics of language for untrusted code
  - language for expressing proofs (e.g. LF)
  - algorithm for validating proofs
  - method for generating safety proofs
- Possible though, see e.g.
  - Necula & Lee, *Safe Kernel Extensions without Run-time Checking*, OSDI 1996; Necula, *Proof Carrying Code*, PPOPL 1997

Sandboxing

- PCC needs a lot of theory and a lot of work
- Sandboxing takes a more direct approach:
  - take untrusted [binary] code as input
  - transform it to make it safe
  - run transformed code
- First work was **Software-Fault Isolation (SFI)**
  - Scan code, and rewrite stores to make them safe; “safe” means limit target to subset of kernel VAS, e.g.

```
sw $t1, 0($t0)->    bge    $t0, top, fault
                    blt    $t0, bottom, fault
                    sw     $t1, 0($t0)
```

  - In practice, code could access one of a number of “segments” (each a contiguous region of kernel VAS)
  - SFI could also check for dangerous instructions…

Issues with SFI

- 1. **Code expansion**
  - Each store becomes at least 4 instructions
    - in the best case of a single aligned segment
  - Spills registers, and can break some code (e.g. CAS)
  - Certain static analysis techniques (e.g. DFA) can help optimize, but still take a performance hit
- 2. **Doesn't handle reads**
  - "Safe" extension can still leak sensitive information
- 3. **Doesn't handle control flow**
  - Either within extension, or to/from rest of kernel
- 4. **Limited / more difficult for non RISC machines**
  - Variable length instructions, ret, call gates (?)
- Lots of subsequent work to improve this
  - BGI (Cambridge) is most recent – see SOSP'09 paper

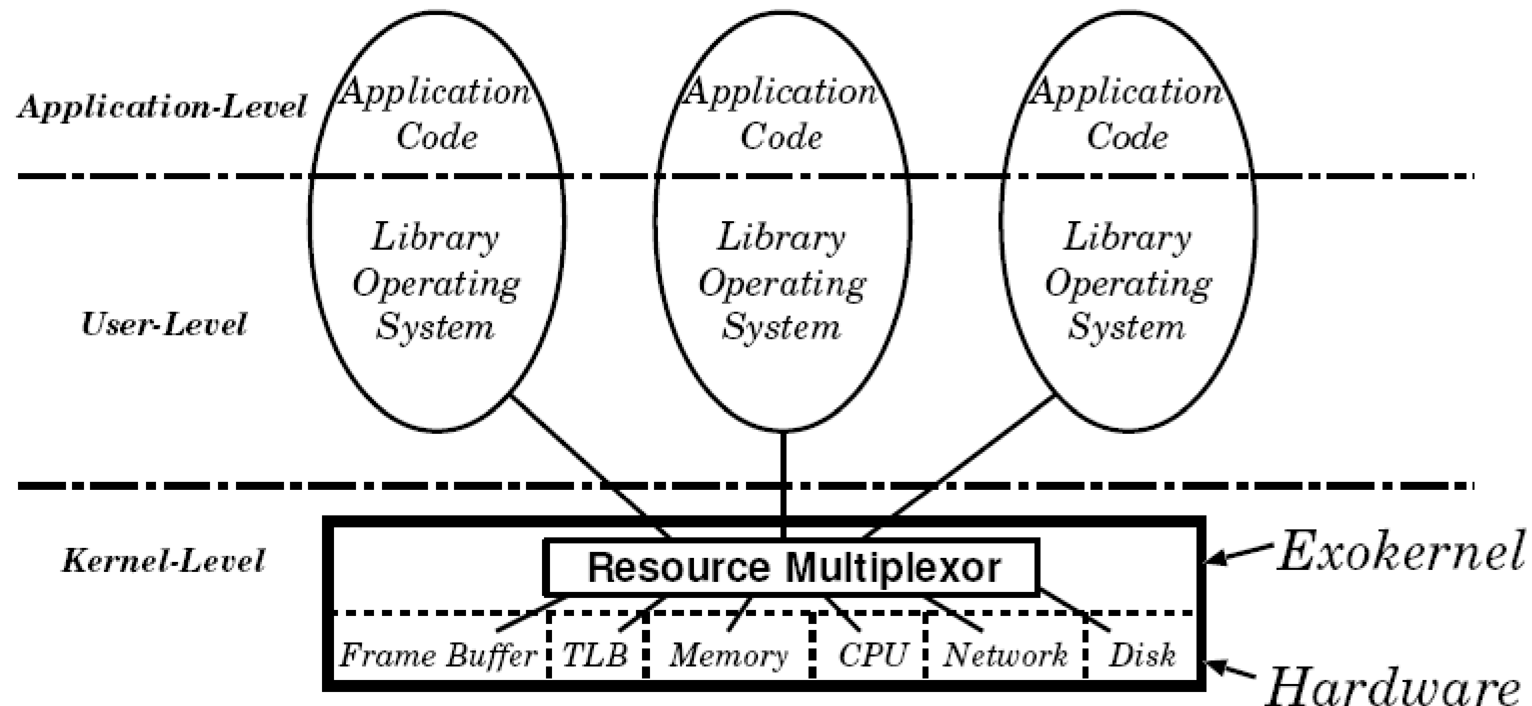
The SPIN Operating System

- **SPIN: a research OS designed for extensibility**
  - Aim to let extensions run in kernel with performance comparable with procedure call => **use language level (compiler checked) safety**
- SPIN kernel written (mostly) in Modula-3
  - Type-safe, with strong interfaces & automatic memory mgt
  - (some low-level kernel stuff in C/assembly)
- Kernel resources referenced by **capabilities**
  - A capability is an unforgeable reference to a resource
- **In SPIN, capabilities are Modula-3 pointers**
  - protection domain is enforced by language name space
  - (not via regions of virtual address space as in previous systems)
- Extensions based on defined interfaces, but somewhat ungeneral:
  - define events and handlers
  - applications register handlers for specific events
  - e.g. handler for "select a runnable thread"
  - what about unforseen needs?
- Problems: trusted compiler, locks, termination. . .

The Vino Operating System

- Set out to overcome perceived problems with SPIN
- Download **grafts** written in C/C++ into kernel.
  - free access to most kernel interfaces
  - safety achieved by SFI (sandboxing)
  - (must use trusted compiler, and trusted kernel linker)
- Prevent quantitative resource abuse (e.g. memory hogging) by **resource quotas** and **accounting**
- Prevent resource starvation by **timeouts**
  - grafts must be preemptible => run in kernel threads
  - decide "experimentally" how long graft can hold certain resources (locks, ipl (?), cpu (?))
  - if graft exceeds limits, terminate.
- Safe graft termination "assured" by **transactions**:
  - wrapper functions around grafts
  - all access to kernel data via accessors
  - two-phase locking + in-memory undo stack

The Exokernel (MIT, 1995-)



- "Exterminate All OS Abstractions!" since they:
  - deny application-specific optimization
  - discourage innovation
  - impose mandatory costs
- Leads to **Exokernel**: minimal resource multiplexor on top of which applications "choose" their own OS abstractions…

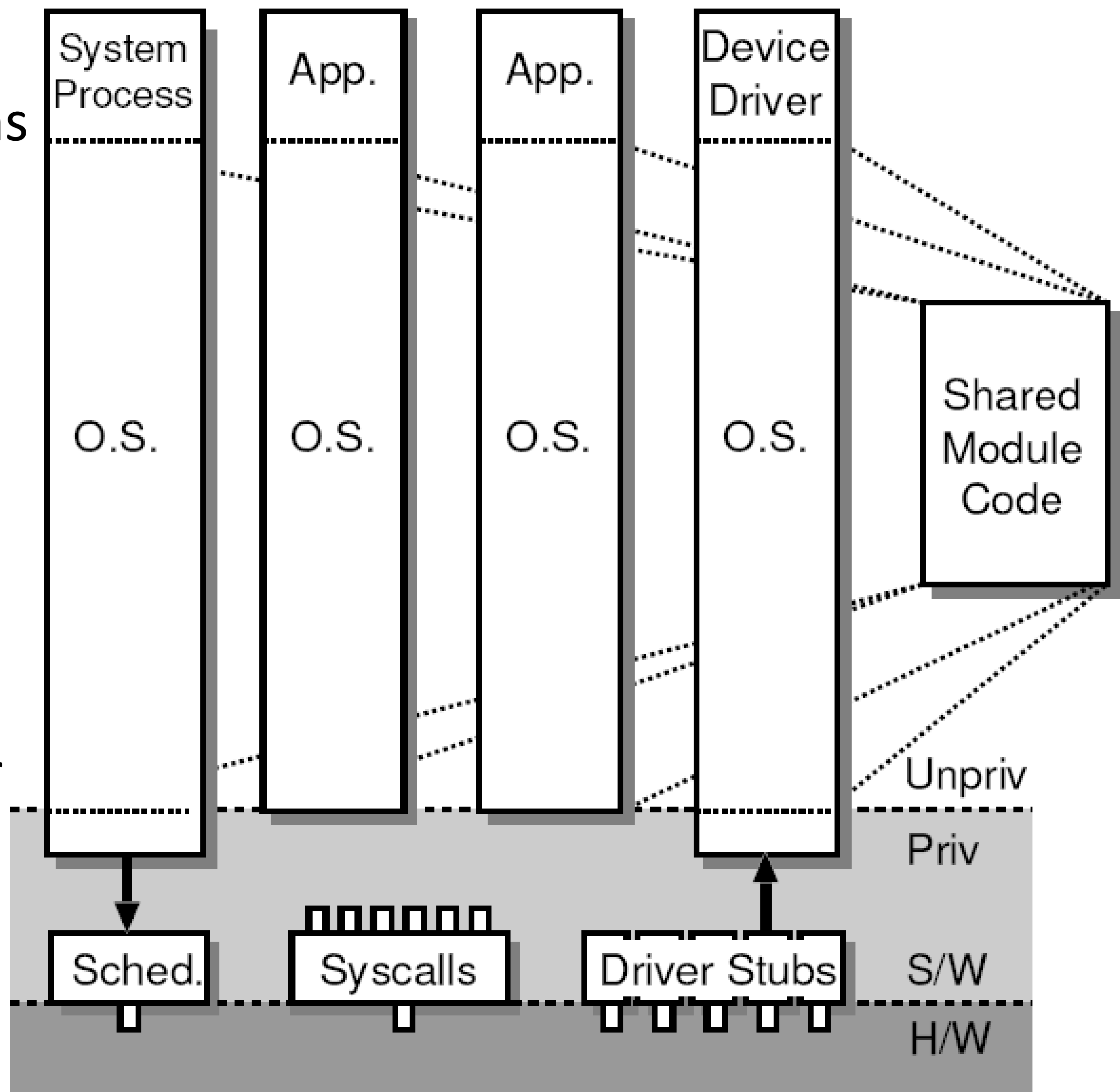
Building an Exokernel

- Key idea: separate concepts of **protection** and **abstraction**
  - Protection is required for safety
  - Abstraction is about convenience
  - Typically conflated in existing OSes, e.g. filesystem instead of block-device access; sockets instead of raw network
- If protect at lowest level, can let applications choose their own abstractions (file system or DB or neither; netw proto)
- Exokernel itself just multiplexes "raw" hardware resources; applications link against **library OS** to provide abstractions

  => **get extensibility, accountability & performance**.
- Still need some "downloading":
  - Describe packets you wish to receive using **DPF**; exokernel compiles to fast, unsafe, machine code
  - **Untrusted Deterministic Functions** (UDFs) allow exokernel to sanity check block allocations.
- Lots of cheezy performance hacks (e.g. Cheetah)

Nemesis (Cambridge, 1993-)

- OS designed for soft real-time applications
- Three principles:
  - Isolation: explicit guarantees to apps
  - Exposure: multiplex real resources
  - Responsibility: apps must do data path
- NTSC is minimal resource multiplexor
  - Just does real-time scheduling & events
  - No device drivers
- Alpha, x86, ARM

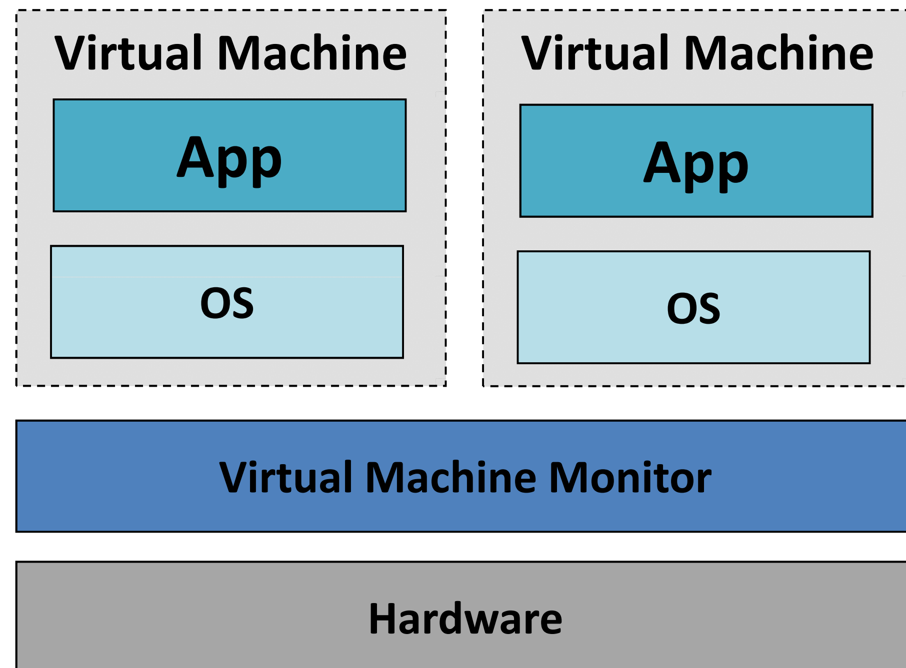


Nemesis versus Exokernel

- Both are **vertically-structured** OSes:
  - Small bit at bottom does secure multiplexing
  - Applications choose functionality to suit them
- **Differences in motivation**:
  - Exokernel: extensibility & performance
  - Nemesis: real-time (accountability) & minimality
- **Differences in virtual addressing**:
  - Exokernel: Unix-style address space per application
  - Nemesis: Single address space (64 bits) shared by entire system … but protection domains per application
- **Differences in linkage**:
  - Exokernel: standard C library
  - Nemesis: strongly-typed IDL, module name space
- And of course, differences in marketing ;-)

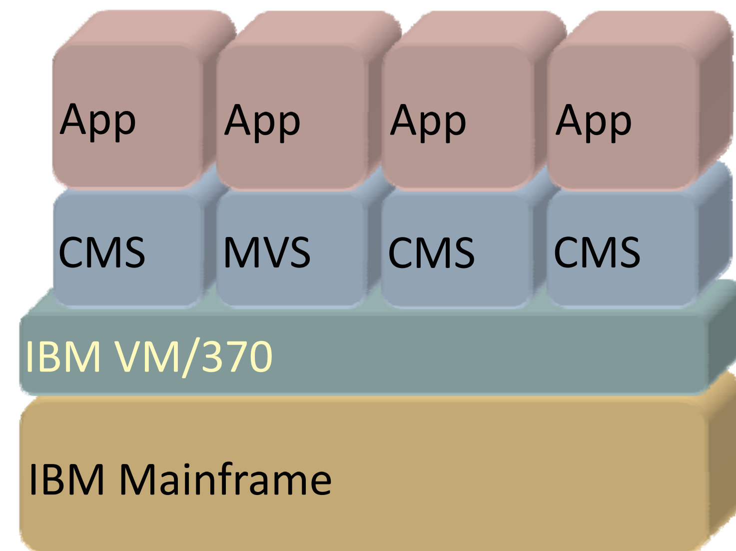
Virtual Machine Monitors

- An alternative system software structure
- Use **virtual machine monitor** (or **hypervisor**) to share h/w between multiple OSes
- Why virtualize?
  - H/W too powerful
  - OSes too bloated
  - Compatibility
  - Better isolation
  - Better manageability
  - Cool and froody



IBMs VM/CMS

- VMM idea pioneered by IBM
  - 1960's: IBM researchers propose VM for OS 360
  - 1970's: implemented on System/370
  - 1990's: VM/ESA for ES9000
  - 2000's: z/VM for System Z
- VMM provides OS with:
  - Virtual console
  - Virtual processor
  - Virtual (physical) memory
  - Virtual I/O devices



IBMs VM/CMS

- Key technique is **trap-and-emulate**:
  - Run guest OS (in virtual machine) in user mode
  - Most instructions run directly on real hardware
  - Privileged instructions trap to VMM, which emulates the appropriate behaviour
- Need some additional software to emulate memory management h/w, I/O devices, etc
- IBM’s VM provides complete virtualization
  - can even run another VMM
  - (and people do – up to 4-levels deep!!)
- Success ascribed to extreme flexibility

And then...?

- VMMs incredibly successful in industry, but mostly **ignored** by academic researchers
  - Instead did microkernels (& extensibility, etc)
- Can be ascribed to a difference in focus
- **VMMS**:
  - Focus on pragmatism, i.e. make it work!
  - Little design freedom (hardware is the spec)
  - Few chances to write research papers
- **Microkernels**:
  - Focus on 'architectural purity'
  - Design whatever you want
  - Write a paper about each design decision ;-)

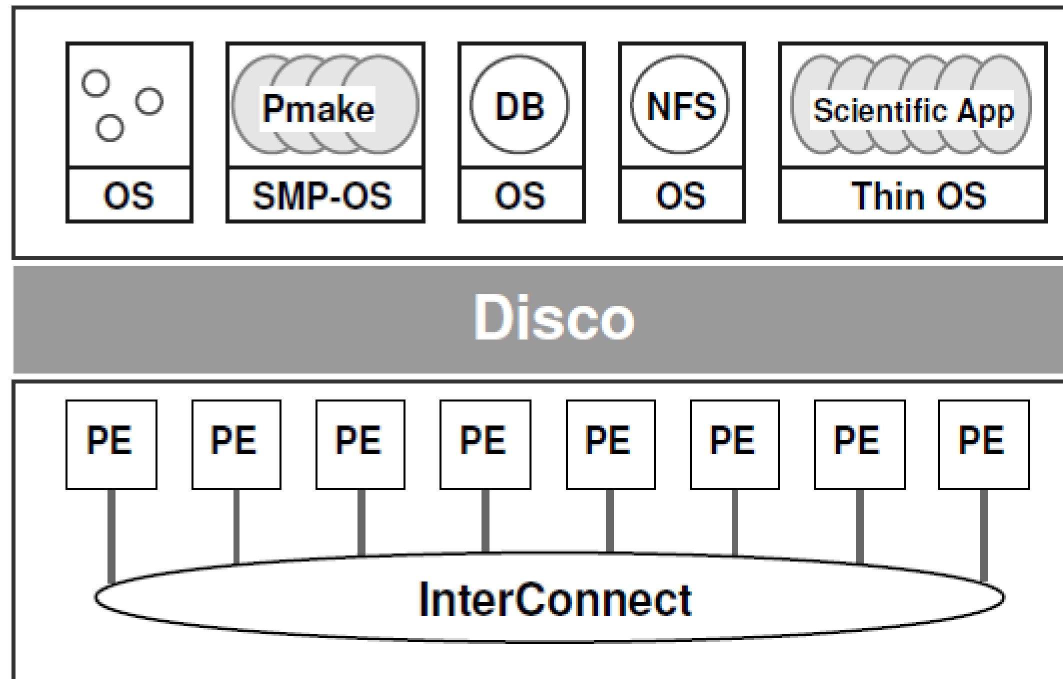
Similarities and Differences

- Both approaches advocate:
  - A small lowest layer with simple well-defined interfaces,
  - Strong isolation/security between components; and
  - Claim benefits of modularity, extensibility, robustness, security
- **But: Different entity multiplexed by lowest layer**:
  - **VMMs**: operating systems (few, big)
  - **uKerns**: tasks or threads (many, small)
- **But: Different basic abstractions provided**:
  - **VMMs**: closely aligned with hardware so have explicit CPU upcalls, maskable asynchronous events, etc
  - **uKerns**: somewhat higher level, so get threads, transparent preemption, capabilities, synchronous IPC
- **But: Different support for address spaces**:
  - **VMMs**: >1 address space per scheduled entity
  - **uKerns**: >1 scheduled entity per address space

Disco (Stanford, 1995)

- VMM idea regained popularity in mid 90’s
- Disco was designed to support cc-NUMA:
  - Commodity OSes didn’t do well on NUMA
  - Tricky to modify them successfully…
  - … but even trickier to write a new OS from scratch
- Instead use Disco hypervisor (VMM):
  - Disco manages inter-processor interconnect, remote memory access, allocation, etc
  - Fakes out UP or SMP virtual machine on top
  - (Mostly) unmodified commodity OS runs in VM

Disco Architecture



- Virtual CPU looks like a real MIPS 10000
  - Trap-and-emulate for privileged instruction (including TLB fill: ‘physical’->‘machine’ mapping)
  - Some changes to OS (buffer cache, NIC) for sharing
- Also enables use of special-purpose OSes…

VMware

- Startup founded 1998 by Stanford Disco dudes
- Basic idea: virtual machines for x86
- One problem: **x86 not classically virtualizable**!
  - **Visibility of privileged state**.
    - e.g. guest can observe its privilege level via *%cs*.
  - **Not all sensitive instructions trap**.
    - e.g. privileged execution of *popf* (pop flags) modifies on-chip privileged state… but doesn't trap if executed in user mode!
    - => cannot just use trap-and-emulate
- To address this, use **dynamic binary rewriting**
  - only of kernel; user-mode code executes unmodified
- Also need to manage guest copy of hardware page-tables (and other gory x86 stuff)
  - Use **shadow page tables**

VMware Implementation

- **DBR: translate at run-time, and on-demand**
  - Use trap-and-emulate to track execution mode
  - Engage translation when enter kernel mode
- Works on **translation units** (TUs): up to 12 x86 instructions (or less if hit control flow insn)
  - Translate from x86 to { safe-subset of x86 }
  - Sensitive instructions replaced with either explicit traps, or user-space emulation code
  - Each TU turns into a **compiled code fragment**: CCF
  - CCF's linked together / optimized over time
    - e.g. to amortize cost of traps
- Can be fragile (imprecise) => not 100% compat

VMware Implementation

- Shadow page tables used to track the “physical” to “machine” address mapping:
  - Need to translate guest page tables into usable ones
  - Much more difficult than Disco (MIPS) since x86 has hardware defined page tables
    - e.g. accessed and dirty bits; 32/PAE/64; superpages; etc.
  - Similar tricks needed for segmentation
- DBR and shadowing lead to a performance hit...
- Emulating I/O devices hurts even more
  - VMware address this by writing special device drivers for display, NIC, etc
- Modern CPUs have hardware support to help with various aspects (VT/SVM, EPT/NPT)
  - But a good s/w VMM can sometimes outperform

Denali (U. Washington, 2001)

- **Motivation**: new application domains
  - pushing dynamic content code to caches, CDNs
  - application layer routing (or peer-to-peer)
  - deploying measurement infrastructures
- Use VMM as an isolation kernel
  - security isolation: no sharing across VMs
  - performance isolation: VMM supports fairness mechanisms (e.g. fair queuing and LRP on network path), static memory allocation
- Aim for decent overall performance by using **paravirtualization**
  - full x86 virtualization needs gory tricks
  - instead invent "new" x86-like ISA
  - write/rewrite OS to deal with this
- Only a proof of concept implementation:
  - Isolation kernel based on Flux OSKit
  - Can only run copies of one specially-constructed single-user guest OS with user-space TCP/IP stack plus user-level threads package
  - (cannot run commodity operating systems)
  - No SMP, no protection, no disk, no QoS

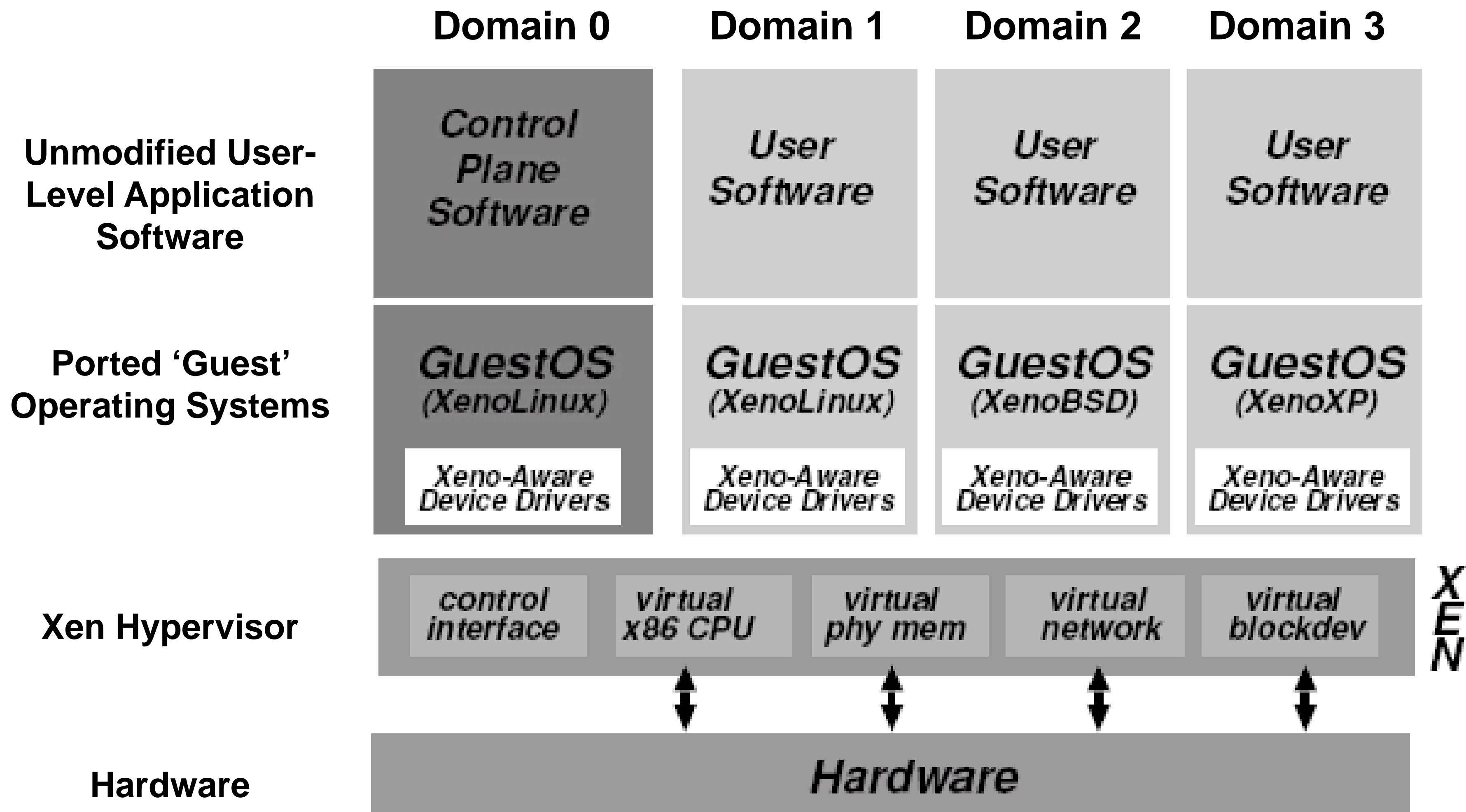
XenoServers (Cambridge, 1999-)

- **Vision**: XenoServers scattered across globe, usable by anyone to host services, applications, ...
  - Location is key, not cycles (so not quite 'the cloud')
- Use **Xen hypervisor** to allow the running of arbitrary untrusted code (including OSes)
  - No requirement for particular language or framework
- Crucial insight:
  - **use SRT techniques** to guarantee resources in time and space, **and then charge for them**.
  - share and protect CPU, memory, network, disks
- Sidestep Denial of Service
- Use paravirtualization, but real operating systems

Xen 1.0

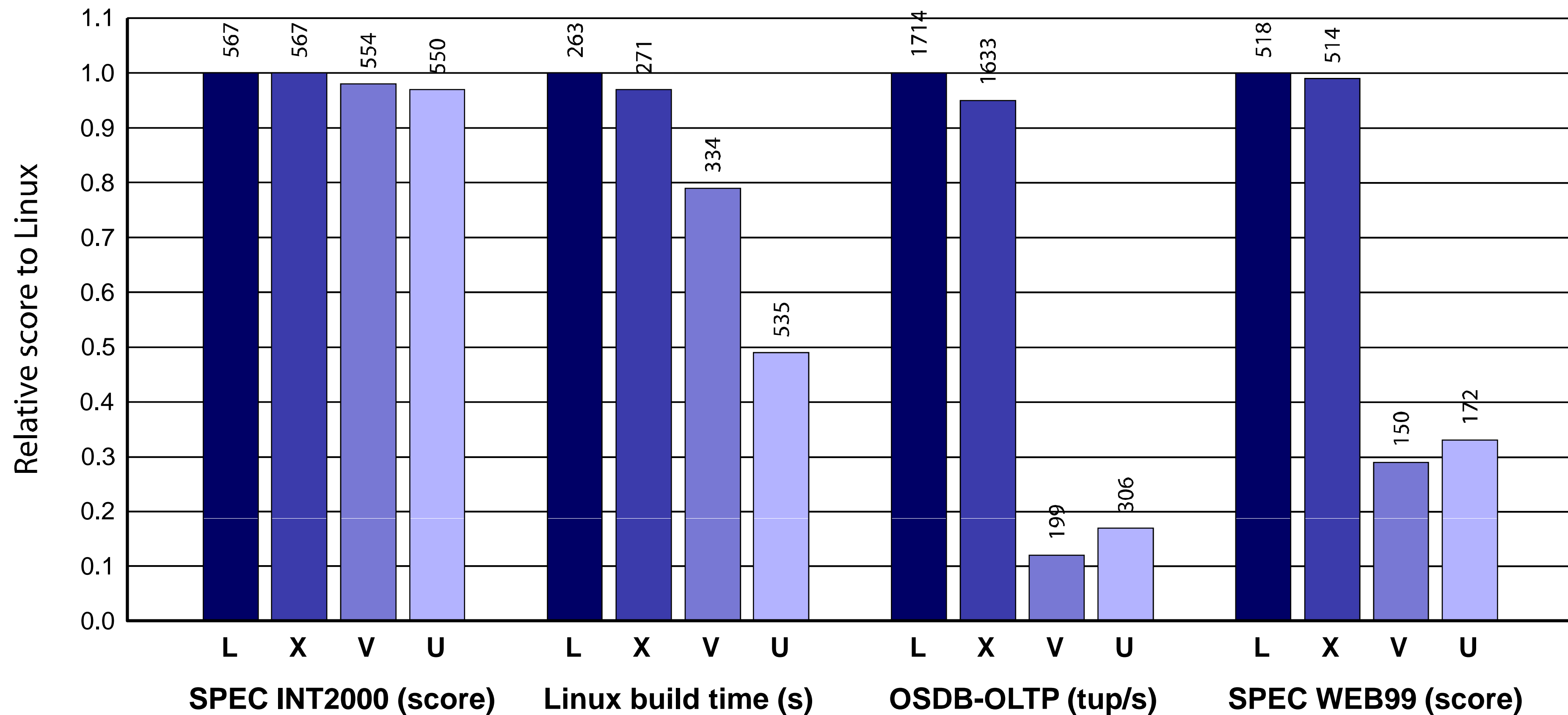
- Work on Xen started in late 2001 / early 2002
- **Xen 1.0 was small** (<50K LOC):
  - 16-bit start-up code (re-used from linux)
  - **SRT scheduler** (BVT), **scheduler activations** and **events**
  - device drivers for timers, NICs, IDE, SCSI.
- Special guest OS (**Domain 0**) started at boot time:
  - privileged interface to Xen to manage other domains
- Physical memory allocated at start-of-day:
  - guest uses **buffered updates** to manipulate page-tables
  - aware of "real" addresses => bit awkward
- Interrupts converted into events:
  - write to event queue in domain
  - domain "sees" events only when activated
- GuestOSes run own scheduler off **virtual or real-time timer**
- **Asynchronous queues** used for network and disk

Xen 1.0 Architecture



- (Xen 1.0 Figure from SOSP 2003 Paper)

Xen 1.0: System Performance

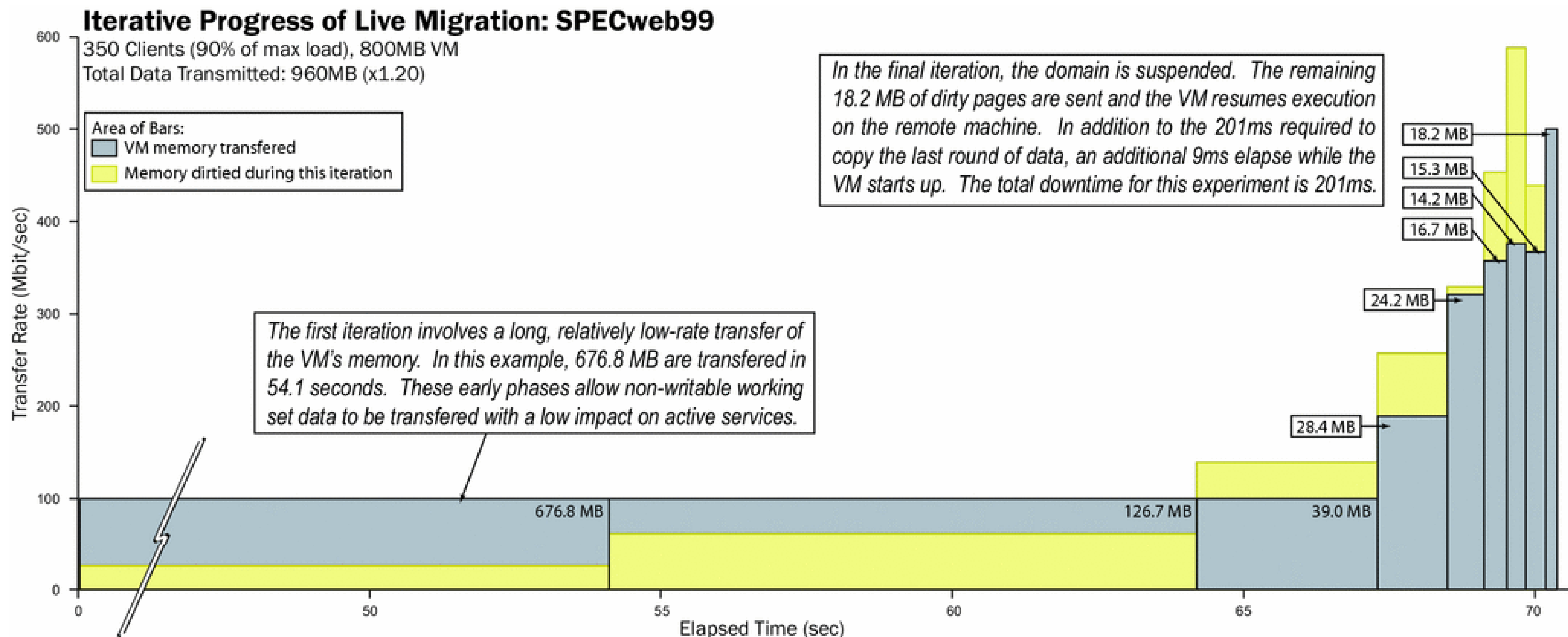


Benchmark suite running on Linux (L), Xen (X), VMware Workstation (V), and UML (U)

- Aim to compare **real** workloads

The Evolution of Xen

- Xen 2 (Nov 04) included many changes, e.g
  - Moved device drivers into **driver domains**
  - Support for **live migration** of virtual machines



The Evolution of Xen

- Xen 3 (Dec 05) included:
  - SMP guests
  - H/W-assisted full virtualization (VT, SVM)
  - 32/36/64-bit support
- Many enhancements since:
  - 32-on-64, COW storage, XSM, VTD, instruction emulation, shadow2, HAP, NUMA, page sharing, …
  - Releases for client (XCI) and cloud (XCP)
- Latest stable release: Xen 3.4 (May 09)
- Development (-unstable) approaching 4.0…
- More info (and code!) from

  http://www.xen.org/

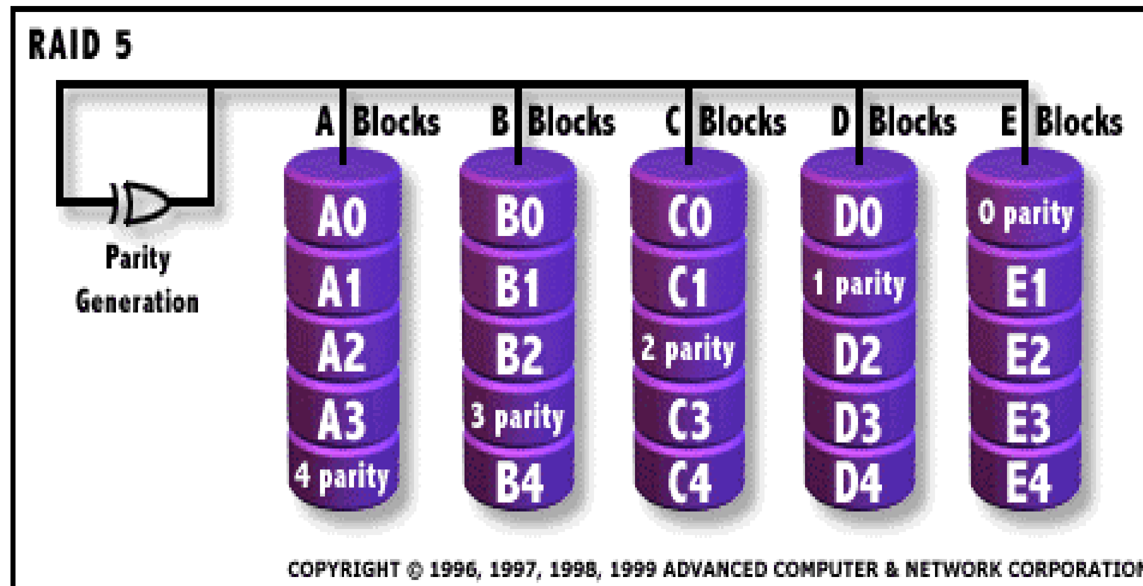
VMMs: Conclusions

- **Old technique having recent resurgence:**
  - really just 1 VMM between 1970 and 1995
  - now at least 10 under development
- Why popular today?
  - OS static size small compared to memory
  - (sharing can reduce this anyhow)
  - security at OS level perceived to be weak
  - flexibility (and "extensibility") as desirable as ever
- Emerging applications:
  - **Internet suspend-and-resume**:
    - run all applications in virtual machine
    - at end of day, suspend VM to disk, copy to other site, & resume
  - **Multi-level secure systems**:
    - many people run VPN from home to work, but machine shared for personal use => risk of viruses, information leakage, etc
    - instead run VM with only VPN access
  - **Data-center management & The Cloud**™

Persistent Storage

- File-systems and databases (and users!) want big, fast, reliable persistent storage
- Disks are cheap, so can scale amount of storage by just using a bunch of disks (JBOD)
  - But: reduced reliability if any disk fails
- RAID = Redundant Array of Inexpensive Disks
  - Set of techniques for building better volumes
  - Increase performance through *striping*
  - More reliable via *redundancy*
    - Simple mirroring (replication)
    - Generalized parity (Reed-Solomon)

Example: RAID-5 Storage

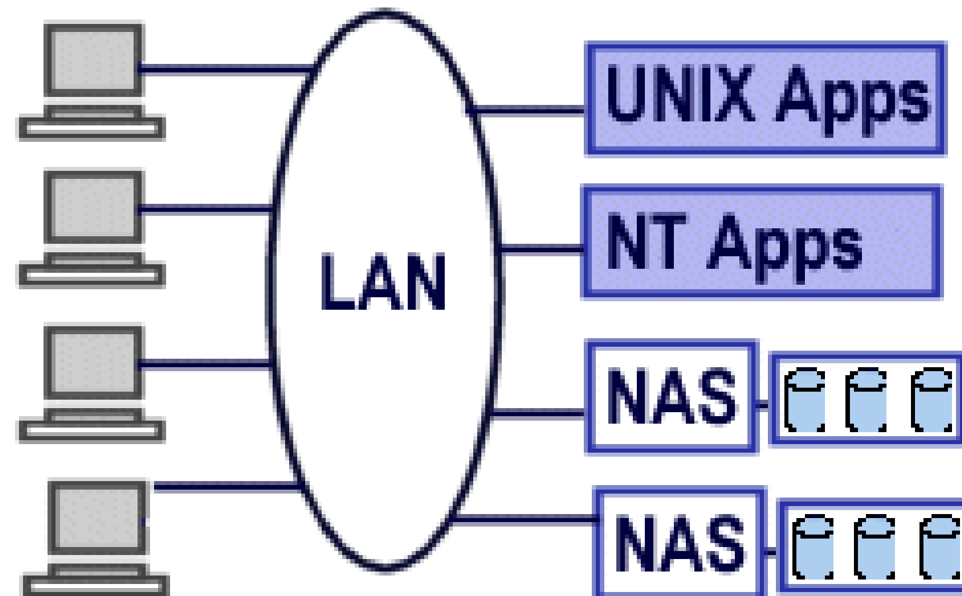


- Generate parity via XOR on writes, check on reads
- Faster reads (five spindles active at a time), and *maybe* faster writes – depends on read/modify/write issues
- Additional reliability (tolerate failure of one disk)
- Overall: provides scalable high quality storage

Distributed Storage

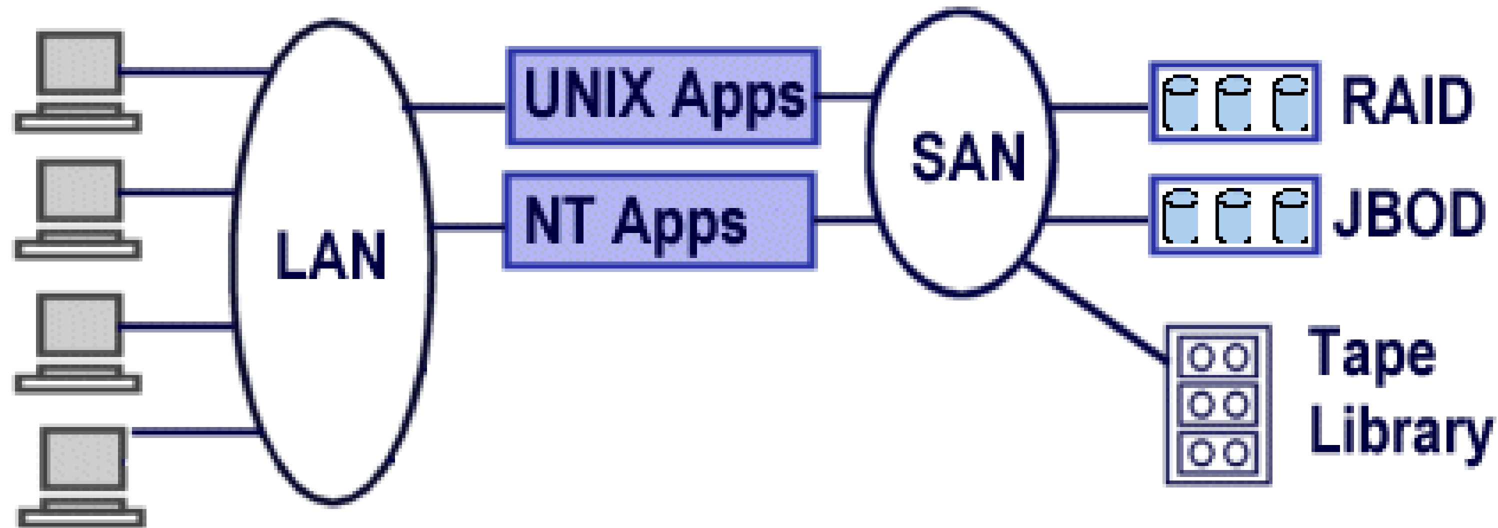
- Even better if make storage **distributed**
- Separate data management from applications
- Why is this a good idea?
  - Centralized data management
    - Provisioning, Security, Backup, etc
  - Even more scalability
  - Location fault tolerance
  - Client mobility (remote access)
- Two may options here: **NAS** and **SAN**

NAS: Network Attached Storage



- Distributes storage at the FS/DBMS level
- Runs over regular TCP/IP (or NetBIOS)
- Server (regular PC or specialized box) provides access via NFS, CIFS, SQL, ...

SAN: Storage Area Network



- Distributes storage at the ***block*** level, accessed via encapsulated SCSI commands
- Runs over specialized fiber channel network (or, more recently: iSCSI, ATAoE, FCoE)
- File-systems / DBMS run directly on hosts

NAS versus SAN

- NAS is the most commonly used
  - e.g. NFS server, CIFS server, NetApp filer, etc
  - Fairly simple RPC-based client software
    - To read "/etc/passwd" just issue two RPCs to server
- SAN more high-end ($$$ and performance)
  - Performance due to:
    - Custom (lossless, non-blocking) network, RDMA
    - High-end storage arrays with lots of NVRAM cache
    - And partly architectural (bottleneck avoidence)
  - Cost from dedicated HBAs + switches (and admins) can be reduced with gigE.. but also lower perf
  - Storage arrays still mostly *extremely* expensive
  - To read "/etc/passwd" issue a whole bunch of SCSI commands…

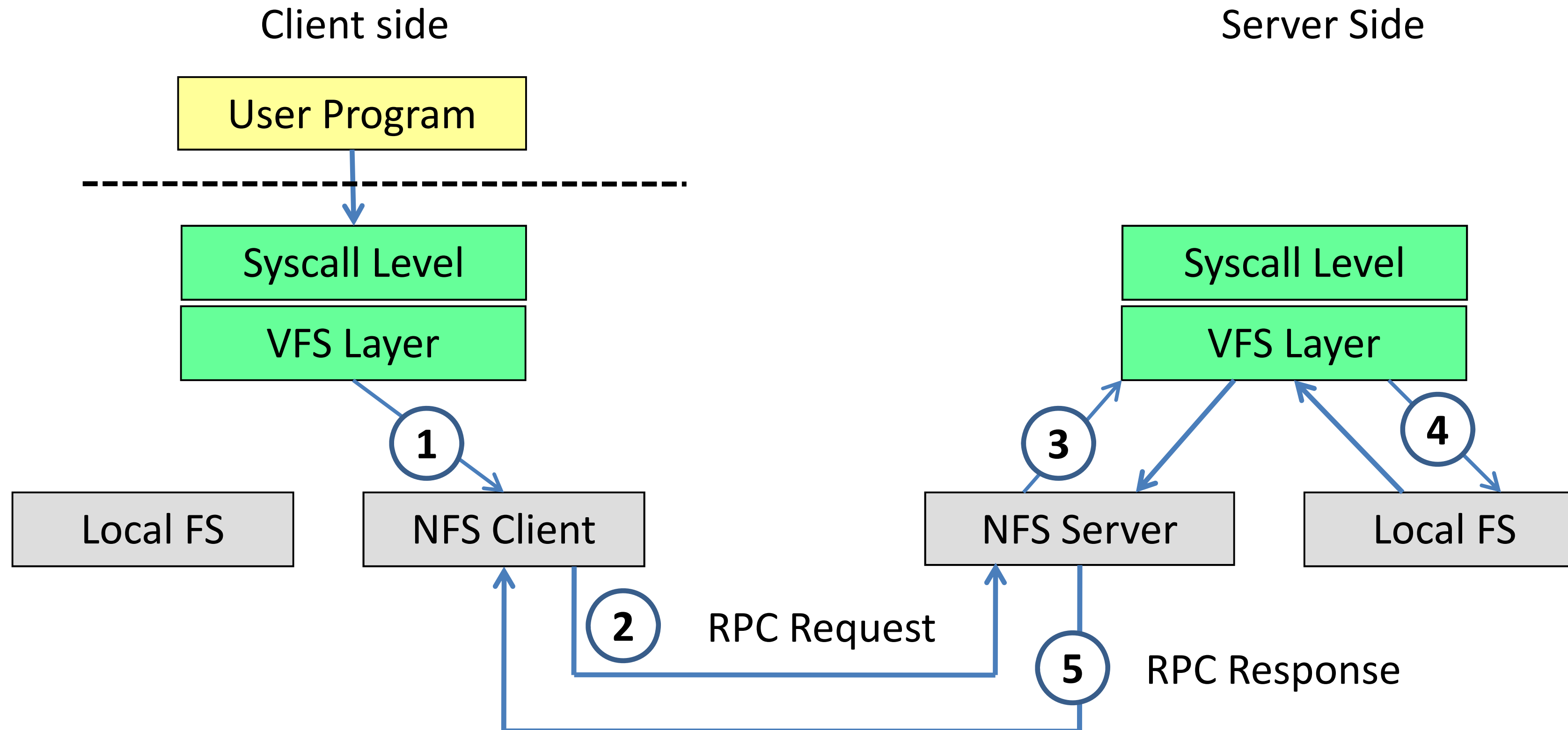
Distributed Storage: Topics

- We'll look at a number of systems and issues for both NAS and SAN systems
- Challenges include:
  - Handling failures (node, network, disk);
  - Providing strong (or reasonable) consistency;
  - Availability of data under various circumstances;
  - Security (confidentiality, integrity, deniability); and
  - Performance (local- or wide-area or both)
- First up: classic client-server systems

NFS: Networked File System

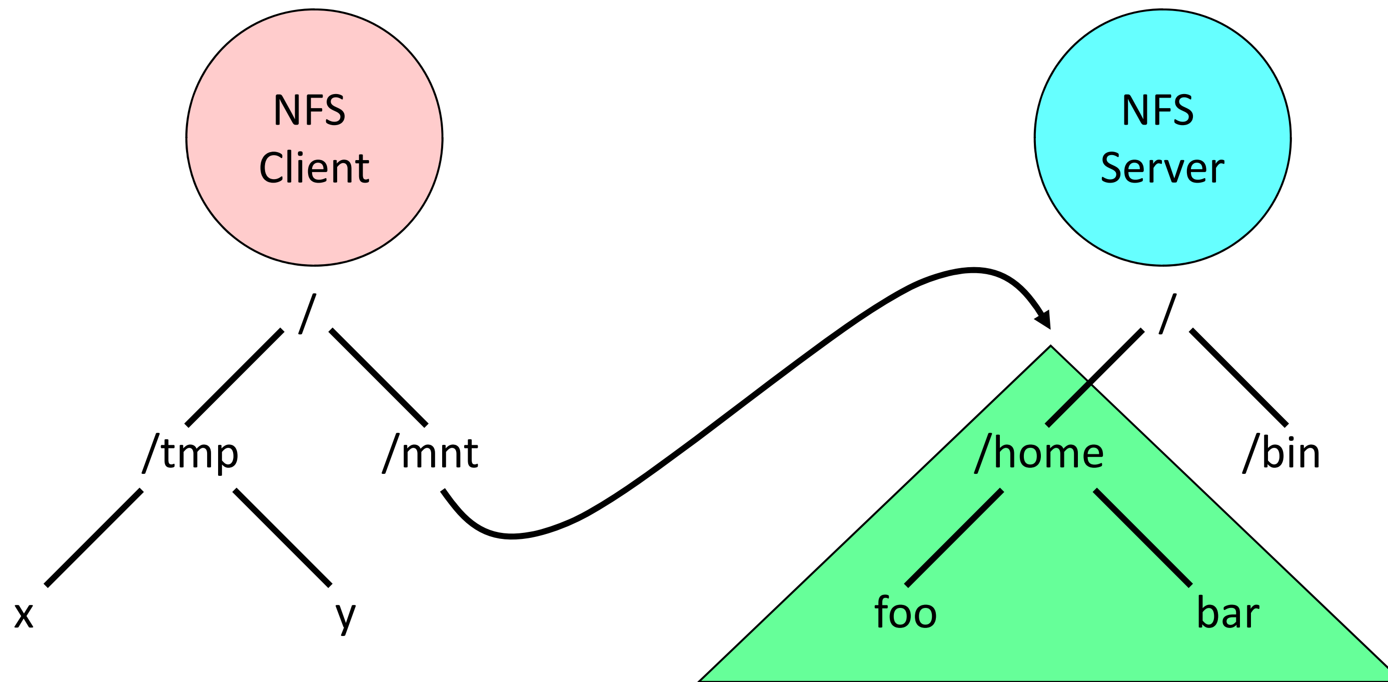
- NFS, developed by Sun, aimed to provide distributed filing by remote access (RPC)
- Key design decisions:
  - High degree of transparency
  - Tolerant of node crashes or network failure
- First public version, NFS v2 (1989), did this by:
  - Unix file system semantics (or almost)
  - Integration into kernel (including mount)
  - Simple stateless client/server architecture

NFS: Simple Client Server



- Client uses opaque **file handles** to refer to files
- Server translates these to local inode numbers
- SunRPC with XDR running over UDP (originally)

NFS: Mounting



- Dedicated mount RPC protocol which:
  - Performs authentication (if any);
  - Negotiates any optional session parameters; and
  - Returns root filehandle

NFS is *Stateless*

- Key NFS design decision to make fault recovery easier
- Stateless means:
  - Doesn't keep any record of current clients
  - Doesn't keep any record of current file accesses
- Hence server can crash + reboot, and clients shouldn't have to do anything (except wait ;-)
- Clients can crash, and server doesn't need to do anything (no cleanup etc)

Implications of Stateless-ness

- No "open" or "close" operations
  - use **lookup(<pathname>)**
- No implicit arguments
  - e.g. cannot support **read(fd, buf, 2048)**
  - Instead use **read(fh, buf, offset, 2048)**
- Note this also makes operations **idempotent**
  - Can tolerate message duplication in network / RPC
- Challenges in providing Unix FS semantics…

Semantic Tricks

- File deletion tricky – what if you discard pages of a file that a client has "open"?
  - NFS changes an unlink() to a rename()
  - Only works for same client (not local delete, or concurrent clients – "stale filehandle")
- Stateless file ***locking*** seems impossible
  - Add two other daemons: **rpc.lockd** and **rpc.statd**
  - Server reboot => **rpc.lockd** contacts clients
  - Client reboot => server's **rpc.statd** tries contact

Performance Problems

- Neither side knows if other is alive or dead
  - All writes must be synchronously committed on server before it returns success
- Very limited client caching…
  - Risk of inconsistent updates if multiple clients have file open for writing at the same time
- These two facts alone meant that NFS v2 had truly ***dreadful*** performance

NFS Evolution

- NFS v3 (1995): mostly minor enhancements
  - Scalability
    - Remove limits on path- and file-name lengths
    - Allow 64-bit offsets for large files
    - Allow large (>8KB) transfer size negotiation
  - Explicit asynchrony
    - Server can do asynchronous writes (write-back)
    - Client sends explicit **commit** after some #writes
  - Optimized operations (**readdirplus**, **symlink**)
- But had *major* impact on performance

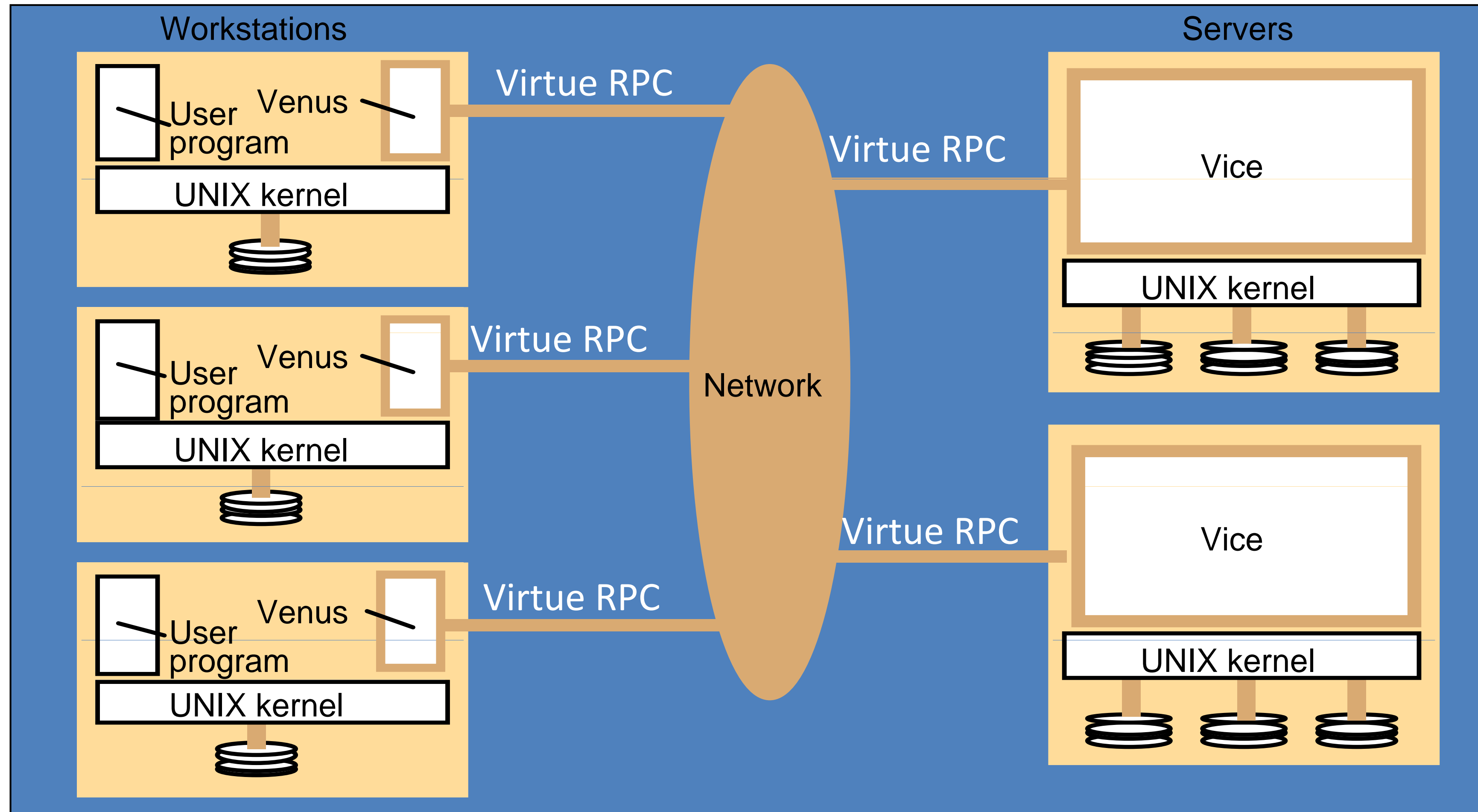
NFS Evolution (2)

- NFS v4 (2003): major rethink
  - **Single *stateful* protocol** (including mount, lock)
  - TCP (or at least reliable transport) only
  - Explicit **open** and **close** operations
  - Share reservations
  - Delegation
  - Arbitrary compound operations
- Actual success yet to be seen…

The Andrew File System (1983)

- A different approach to remote file access
- Meant to service a large organization
  - Scaling is a major goal
- Basic AFS model:
  - Files are stored permanently at file server machines
  - Users work from workstation machines
    - With their own private namespace
  - Andrew provides mechanisms to cache user’s files from shared namespace
- Even “local” accesses go via client

Vice, Virtue and Venus...



Basic Idea: Whole File Caching

- **Andrew caches *entire files* from the system**.
  - On open Venus caches files from Vice
  - On close, [modified] copies are written back
- Reading and writing bytes of a file are done on the cached copy by the local kernel
- Venus also caches contents of directories and symbolic links, for path-name translation
  - Exceptions for modifications to made directly on the server responsible for that directory

Why do Whole-File Caching?

- Minimizes communications with server
  - Less network traffic
  - Better performance
- Most files used in entirety anyway (prefetch)
- Simpler cache management
- However does requires substantial free disk space on workstations
  - Can be an issue for huge files
  - Later versions allow caching part of a file

Andrew Shared Namespace

- An AFS installation provides a single, globally shared file-system namespace
- A **fid** identifies a Vice file or directory
- A fid is 96 bits long; three 32-bit components:
  - **volume number** (a unit holding files of a single client)
  - **vnode number** (~= an inode for a single volume)
  - **uniquifier** (generation number for vnode numbers, thereby keeping certain data structures compact)
- High degree of name and location transparency
  - Fids do not embed any notion of location
  - Every server stores volume->server mapping

AFS Consistency

- Aiming to provide "local" semantics
- Implemented by callbacks:
  - On open, Venus checks if client already has copy
  - If not, then requests from Vice server that is custodian of that particular file
  - Server returns contents along with a **callback promise** (and logs this to durable storage)
- Whenever a client sends back an updated copy (e.g. on close), invoke all callbacks
- Same scheme used for volume map

AFS Pros and Cons (1)

- **Performance**
  - Most file operations are done locally (and most files typically have one writer in a time window)
  - Little load on servers beyond open/close timescales
- **Location transparency**
  - Indirection via volume map makes it easy to move volumes
  - Also can do limited replication (read-only files)
- **Scalability**
  - Initial design aimed for 200:1 client-server ratio
  - Indirection and caching makes this easily achievable
- "**Single System Image**"
  - Clients (workstations) essentially interchangeable

AFS Pros and Cons (2)

- **Good Security**
  - Client machines untrusted
    - only Vice servers trusted
  - Strong initial authentication via Kerberos
  - Can use encryption used to protect transmissions
- But:
  - **Complex and invasive** (“take over the world”)
  - **Usability issues**, e.g. ticket expiration, weird “last close wins” semantics for concurrent update
- **Ultimately AFS popular only in niche domains**

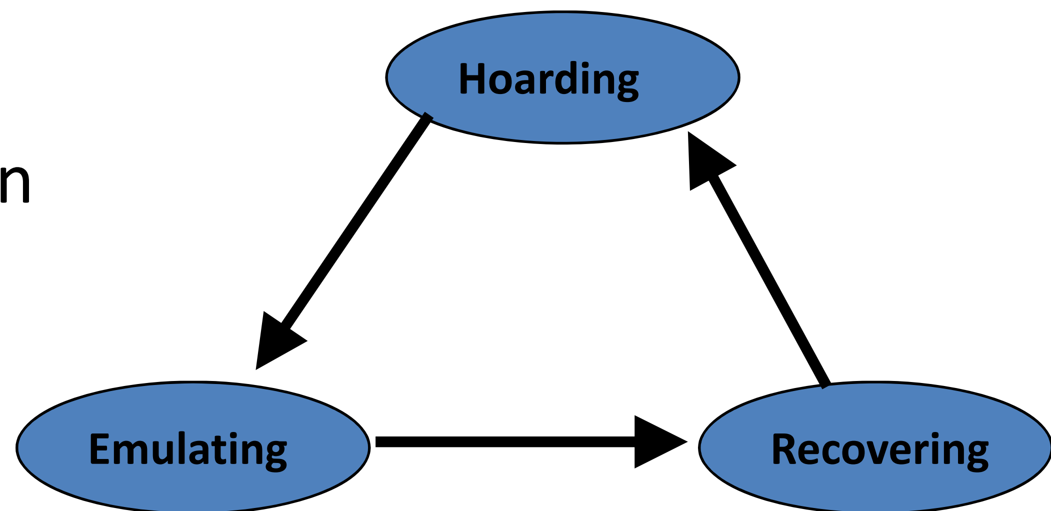
Coda (CMU, 1987+)

- A system supporting **optimistic replication**
  - Allow copies of data which may not be up-to-date
  - Essentially client/server (developed from AFS)
- Motivated by the emergence of laptops
  - When connected to network, laptop operated just like any other andrew workstation
  - When disconnected, however, AFS allowed no file updates once the leases expired
  - This was fine for temporary outages in AFS (e.g. reboot or network glitch), but not for mobile use

Coda Operation

- Change the Venus cache manager to operate in three different modes:

1. **Hoarding**
   – "Normal" operation
2. **Emulating**
   – Disconnected
3. **Reintegrating**
   – Reconciling changes back to the server



- Few changes required to Vice or Virtue

Coda: Hoarding

- “Normal” operation a little different than AFS
  - Aggressively cache copies of files on local disk
- Add a **Hoard Database** (HDB) to Coda clients
  - Specifies files to be cached on local disk
  - User can tweak HDB, and add priorities
    - Laptop disks were small back in the day
  - Files actually cached a function of hoard priority and actual usage – can pickup dependencies
- Do ***hoard walk*** periodically (or on request)
  - ensure disk has only highest priority files

Coda: Emulating

- When disconnected, attempts to access files not in the cache appear as failures to apps
- All changes made to anything are written in a persistent log (the client modification log)
  - In implementation was managed by using lightweight recoverable virtual memory (LRVM)
  - Simplifies Venus itself
- Venus purges unnecessary entries from the CML (e.g. updates to files later deleted)

Coda: Reintegrating

- Once a coda client is reconnected, it initiates a reintegration process
  - Performed one volume at a time
  - Venue ships replay to each volume
  - Volumes execute a log replay algorithm
  - Basic conflict detection and 'resolution'
- Lessons learned:
  - Reintegration can take a long time (need to have a fast network)
  - Conflicts rare in practice (0.75% chance of update of same file by two users within 24 hours)

Coda: Summary

- Generally better than AFS
  - Inherits most AFS advantages, but adds more
  - e.g. replicated Vice servers with writable replicas
  - e.g. CML can end up coalescing updates (or removing them entirely) => less traffic, server load
- Much simpler than earlier schemes (e.g. Ficus)
  - Client only needs to reconcile with "its" server
  - Servers themselves strongly connected + robust
  - Garbage collection straightforward

File Systems for SANs

- Recall that SAN has a bunch of "disks" (volumes) accessible via encapsulated SCSI
- But most file-systems don't expect multiple independent clients => need coordination
- Two main ways to build a shared-disk file-system: **asymmetric** or **symmetric**
- Asymmetric simplest:
  - Have dedicated metadata server (or servers) with exclusive access to metadata disk (or disks)
  - Clients do directory / inode lookups and allocation requests via the metadata server
  - Once have information, can directly read/write disks

Symmetric SDFS

- Unfortunately asymmetric systems can suffer from performance bottlenecks / failures
- A symmetric shared disk file system instead manages coordination between set of clients
  - Requires distributed lock manager, etc
  - Care needed to avoid deadlock
- Becoming more mature; examples include:
  - RedHat Global File System [GFS]; and
  - IBMs General Parallel File System [GPFS]
- Also get ***hybrid*** systems using object storage…

Redhat GFS

- Splits problem into two layers:
- Bottom layer: network-accessible logical volumes
  - **Assume basic LUN** (or volume or disk) is accessible by all nodes; range of solutions for price points
    - GNBD: software to export partions/disks of linux boxes
    - iSCSI: encapsulated SCSI to unix server or low-end array
    - Fiber channel: encapsulated SCSI to high perf array
  - **Can add multipath** software/hardware for greater fault tolerance (and potentially throughput)
  - **Build cluster LVM on top of this**:
    - Allows creation of logical volumes which span many LUNs
    - Software layer supports striping, resize, snapshot, etc

Redhat GFS

- File system layer sits on top of logical volumes
- Almost standard Unix semantics
- GLUM (or other) distributed lock management required for metadata updates and allocation
  - Uses “linux-ha” (heartbeat2) for transparent failover between replicas of lock managers
- Performance can be good, but metadata managers are complex and slow

Network Attached Secure Disks (CMU)

- **NASD: basic idea is a less stupid SAN**
- Still have shared disks, but:
  - Disks export a variable length *object* interface
  - Disk does create, read, write (including allocation)
  - Can use this to build DBMS, file system, etc
- "Secure"?
  - confidentiality and integrity for transfers
  - Disks know (a little) about access control
    - File manager issues clients capabilities
    - Updates NASD[s] to enable direct access
    - Revocation possible too ("password capabilities")

NASD Summary

- Basically gives a “half way house” between symmetric and asymmetric SAN systems
- Data path is fast and secure
- Off loads work from file-manager
  - E.g. NFS on NASD requires 10x less mgr cycles
- Can get parallelism (multiple NASDs can be accessed in parallel)
- Commercially available via Panasus (FAST’08)

Serverless Network File Systems

- New network technologies are much faster, with much higher bandwidth
  - Going over the net can be quicker than local disk!
- Serverless network file systems exploit this:
  - Peer machines providing file service for each other
  - High degree of location independence
  - Make use of all machine's caches
  - Provide reliability in case of failures

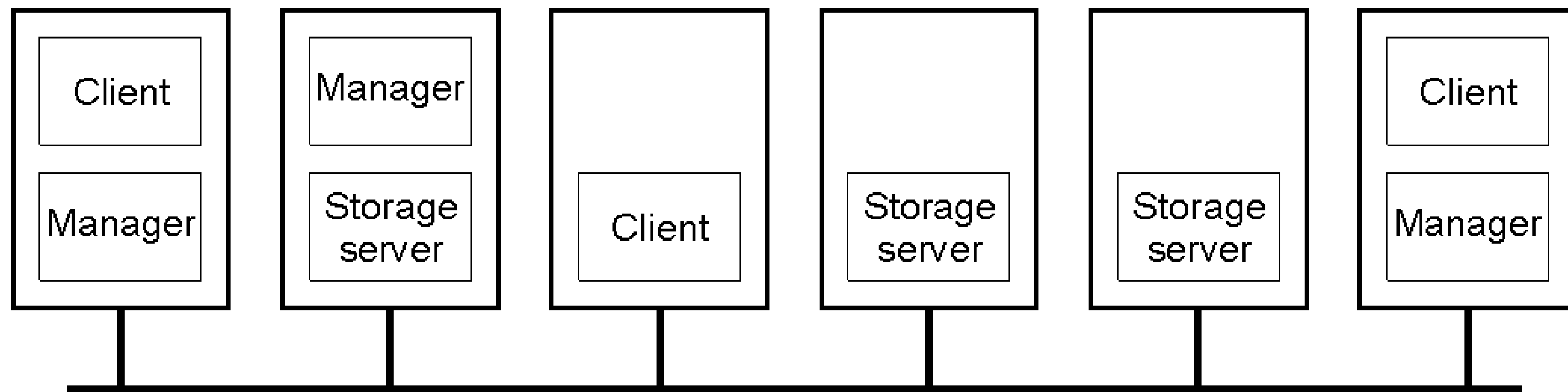
Example: xFS (Berkely)

- Part of the NOW project
  - Designed for high-speed LANs
  - Fully distributed file system – no single server
  - (Think P2P in the local area)
- Inherits ideas from several sources
  - Log-structured file systems
  - Zebra (network raid)
  - Multiprocessor cache coherence

How can we distribute a file server?

- Well what does a file server actually do?
  - Stores file data blocks on its disks
  - Maintains file location information
  - Maintains cache of data blocks
  - Returns data to clients on request
  - Manages cache consistency for its clients
- xFS nodes work in collaboration to provide the above functions:
  - Any data/metadata can be located at any machine
  - Each machine takes on one or more roles

Example: xFS (Berkely)



- Roles include **client**, **manager** & **storage server**
  - Client is just the client of the system as usual
  - Manger handles metadata + cache coherence
  - Storage server stores blocks
- All machines can also **cache** and **clean** (see later)

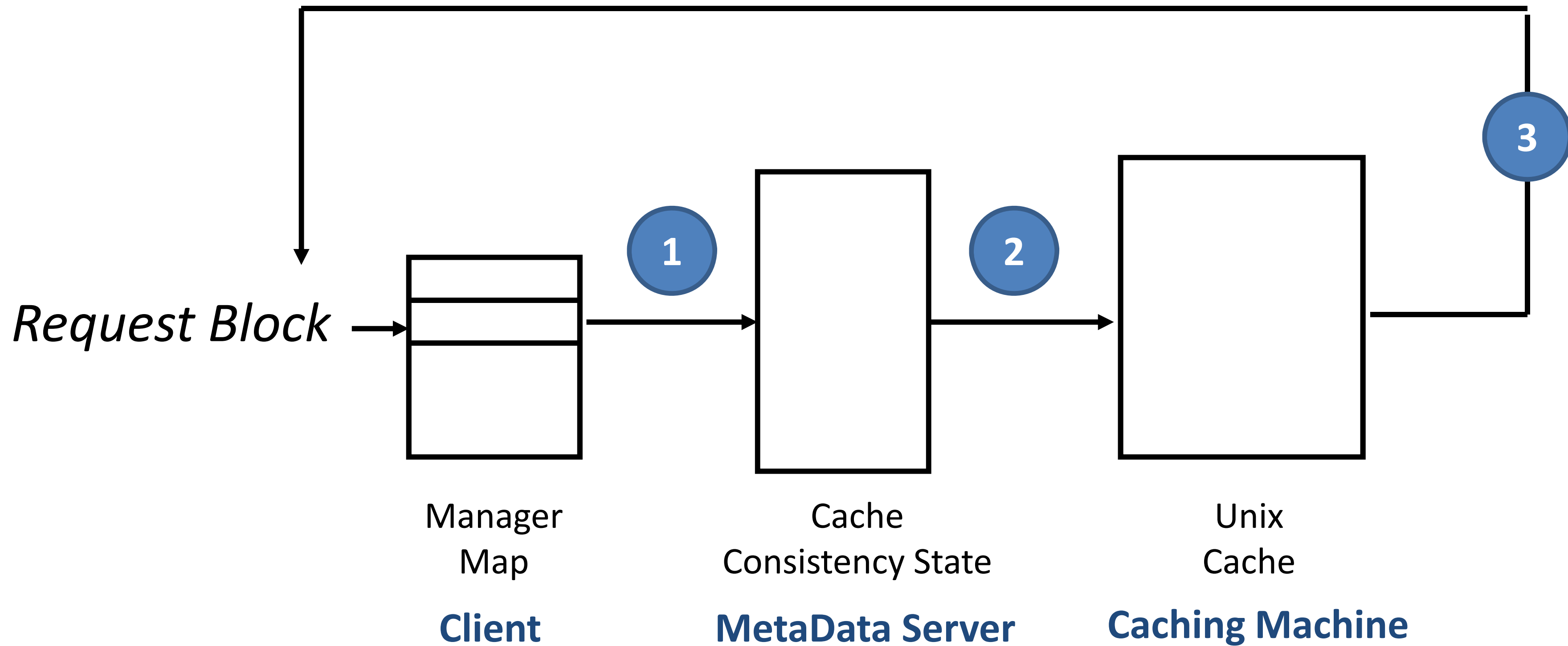
So how does it work?

- Assume client has looked up file in a directory
  - Results in index number (equivalent to inode #)
- Client uses **manager map** to find the metadata manager for this file/directory:
  - Manager map is globally replicated
  - Maps clusters of index numbers to managers; so << than #files (and can avoid updating)
- Client can now send request (e.g. read N bytes from offset O) to manager
  - Assuming it's not locally cached already

What does manager do?

- Manager maintains two data structures
  - **IMAP** translates index numbers to log addresses
  - **Cache map** tracks who has copy of data
- If manager receives a request, it first checks the cache map – if someone already has data cached, then redirect request to them
  - Always prefer cached to disk, even over network

Getting a Block from a Remote Cache



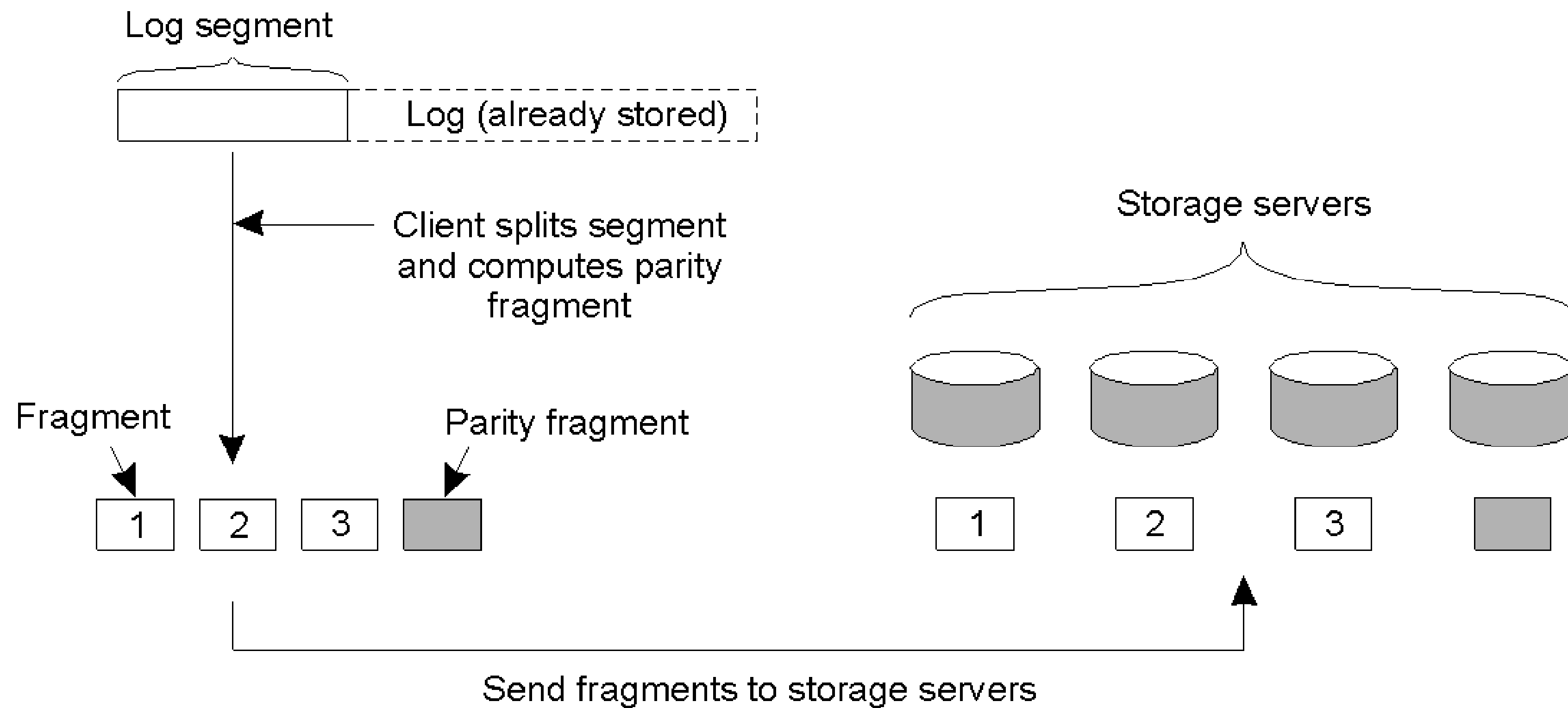
What if we need to go to disk?

- xFS combines two previous techniques to handle persistent storage:
  - Network RAID via **stripe groups**
  - **Log-structured storage**
- Stripe groups pretty simple:
  - Take e.g. 4 storage servers as a stripe group
  - Write blocks 0, 1, 2 on servers 0, 1, 2; parity on 3
  - A stripe group defines a logical volume
    - Set of inodes, etc
  - Track stripe groups in globally replicated **stripe map**
    - Limited in size, and with a fixed stripe size
    - (However can recursively mount volumes as per Unix)

Log-Structured Storage

- Based on LFS (Selzer)
  - Argues that file-system performance is poor mostly due to seeks (reads mostly cached)
  - Hence **consider disk as infinite append-only log**
  - All writes (data and metadata appended to log)
  - Need some extra structures to locate inodes, most recent version of data blocks, etc
- xFS uses this technique for stripe groups
  - Chiefly to avoid read/modify/write issues

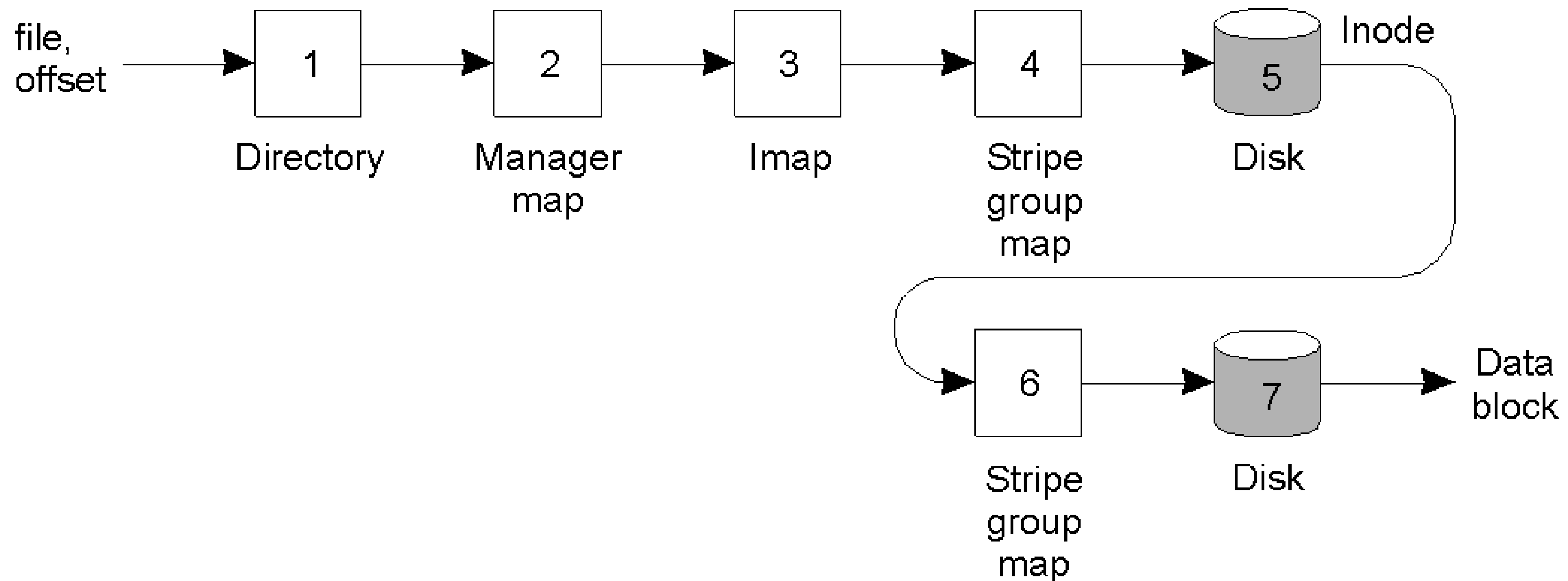
Batched updates with xFS



Back to example: what if cache misses?

- Manager for this file must consult its **IMAP**
  - IMAP translates from index# to address of inode
- "Address" is a log address, which is a triple:
  - <stripe group id, segment id, segment offset>
  - Segments are chunks of log managed separately
  - (log overall is chained list of segments)
- After reading the inode, have log address of relevant blocks for read…

Reading a block from Disk



- Writes operate in a similar fashion:
  - Simply append updated data block(s) if overwriting;
  - Or write new inode & data block(s) otherwise

Other Issues

- **Cache coherence**
  - xFS manages this on per [logical] block basis
  - To write a block, client requests token from the relevant metadata server
  - Metadata server retrieves token from whoever currently has it, and invalidates other caches
- **Log cleaning**
  - Append-only storage will eventually fill up
  - xFS relies on responsible distributed cleaners

xFS Summary

- Novel system aspects include:
  - Serverless design
  - Aggressive cooperative caching
  - Combined log-based / striped storage
- Performance results on early prototype showed up to 10x improvement over NFS
  - Though only compared against single NFS server
- But only partial story on fault tolerance
- And relies on trusting quite a lot of machines…

Farsite (MSR, OSDI 2002)

- A more recent serverless file system
  - Designed to exploit large number of desktop machines in universities / corporations
  - Less trusting than xFS ;-)
- Basic model puts machines into one of 3 roles
  - **Client machines**: access data
  - **Directory group members**: handle lookups
  - **File (storage) hosts**: store data
- Usually a machine has at most two roles

Farsite Operation

- Client machines interact with the user
- Directory groups collectively manage file information using PBFT:
  - **BFT = Byzantine Fault Tolerance**:
  - Every machine has own copy of file metadata
  - 3f+1 machines can tolerate up to f faults
- Conceptually clients issue read requests to directory groups who reply with data; when client updates a file, send update to directory group
- However high replication cost for BFT (e.g. 7 or 10) would mean high storage costs

Farsite Enhancements

- To solve this, Farsite introduces **file hosts**
  - Directory groups just maintain cryptographic hash of file contents => can detect byzantine file host
  - Hence can tolerate N-1 failures of file hosts, and so can replicated these to a smaller degree
- Farsite also allows whole file caching at clients
  - Directory group can issue lease to client (c/f AFS)
  - Client can then delay pushing updates => big win
  - Get both exclusive and read-only variants

Scaling a Farsite System

- Design aimed for 10^5 hosts
  - Must avoid directory group performance bottleneck
  - (Availability can also be an issue)
- Farsite solves this by allowing DG quorum to delegate part of its name space to a new DG
  - Due to hierarchical namespace nature, clients cache ***hints*** about which DG owns which prefix
  - If contact DG for path /a/b/c/…/z.txt, it replies with delegate info for longest prefix it knows
  - Subsequently use this DG directly for similar accesses
  - If stale, DG will inform client, who invalidates hint

Farsite: Summary

- Security/trust focused serverless system
- In addition to BFT, makes extensive use of PKI
  - Every user and machine has key pair
  - Used to validate machine roles and user accesses
- Did implementation on NT
  - Stacked approach with “switch” kernel module choosing between local and remote (CIFS)
  - Performance eval with 4 node DG and 1 client
- Anecdotal evidence that it didn’t work well
  - Desktop machines turned off a lot (green!)
  - Could be “fixed” by having dedicated file hosts

LBFS (MIT, SOSP 2001)

- Motivation:
  - More an more people have devices with wireless access (laptops & PDAs, cellular modems or WiFi)
  - However NFS/CIFS suck over high latency / low bw
- LBFS is file system for such networks
  - Avoids optimistic caching to avoid conflict issues
  - Eschews remote login since very limited use cases
- Key idea:
  - Extremely aggressive compression in the air

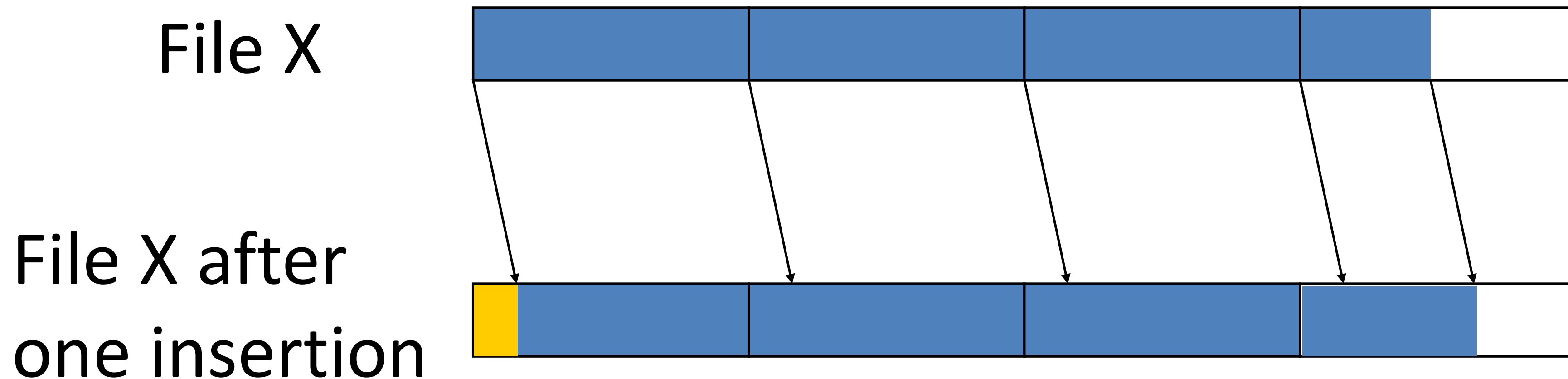
Aggressive Compression?

- Need to compress an order of magnitude better than traditional techniques
- How can we do this?
- Key idea:
  - Exploit similarities between server file-system and client persistent cache
  - Essentially have massive “codebook” we can use
- LBFS identifies ‘chunks’ of files by hash value
  - Instead of transferring chunk, just send hash

In More Detail

- Client maintains persistent cache on disk
- Server maintains full file system
- Both divide files into **chunks**, and for each chunk computes secure SHA-1 hash
- Both have index from hashes to file chunks
- Then when a client wants to e.g. write a portion of a file, just transfers hashes
  - If server already has those chunks can use them; replies to client with any misses, and client sends data
  - Can do similar thing on read (server sends hashes…)

This is going to work but…



➡ **The two files do not have a single block in common!**

- Also affects fetch of files never seen before…

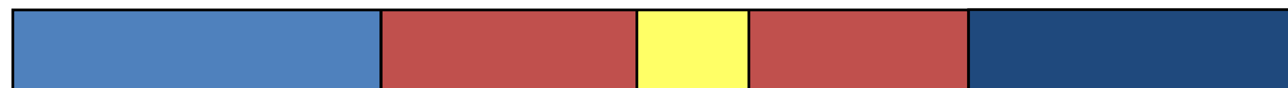
So here's the clever bit

- Rather than split files into fixed-size chunks, LBFS divides them based on ***content***
- Compute **Rabin fingerprint** over every overlapping 48-byte region of a file
  - Incremental "stream hash" so computing fast
- Define chunk boundary when low-order 13 bits of hash hits a magic (system wide) value
  - If hash uniform, expect 8K chunks on average
  - Use min (2K) and max (64K) for pathological cases

How it works

A file X partitioned into three chunks…

Same file X after one insertion inside middle chunk



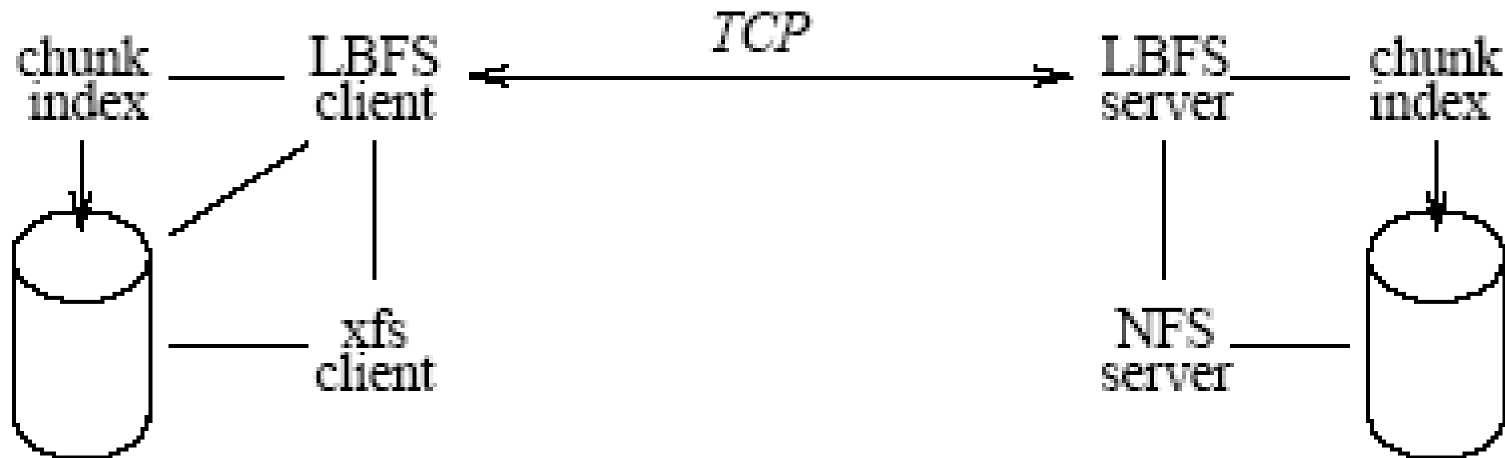
New Chunk

Chunk boundaries are *arbitrary* and identified by the content of their boundary regions

LBFS Protocol

- NFS with some changes:
  - Uses **leases** for close-to-open consistency
    - callbacks with limited lifetime; a few seconds
  - Practices **aggressive pipelining** of RPC calls
  - **Compresses** (gzip) all RPC traffic
- Reads/writes use additional calls not in NFS
  - GETHASH for reads
  - MKTMPFILE, and three others for writes
- Server ensures **atomicity of updates** by writing them first into a temporary file

Implementation

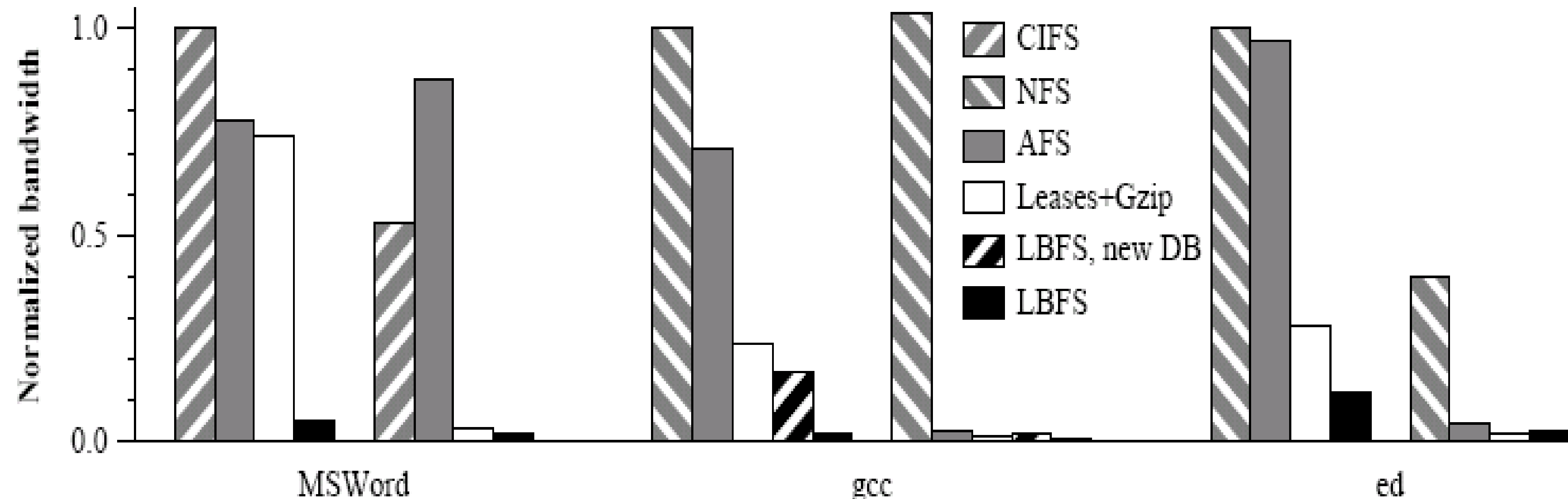


- Stateful transport for leases/callbacks etc
  - need modified/proxy TCP for ok perf on wireless
- Chunk database consistency requires some effort

Evaluation (I)

- Compared upstream and downstream bandwidth of LBFS with those of
  - CIFS (Common Internet File System)
  - NFS
  - AFS
  - LBFS with leases and gzip but w/o chunking
- Used a variety of “productivity” workloads
- Downstream benefits most from chunking

Evaluation (II)



- In each case:
  - first four bars show **upstream bandwidth**;
  - second four show **downstream bandwidth**

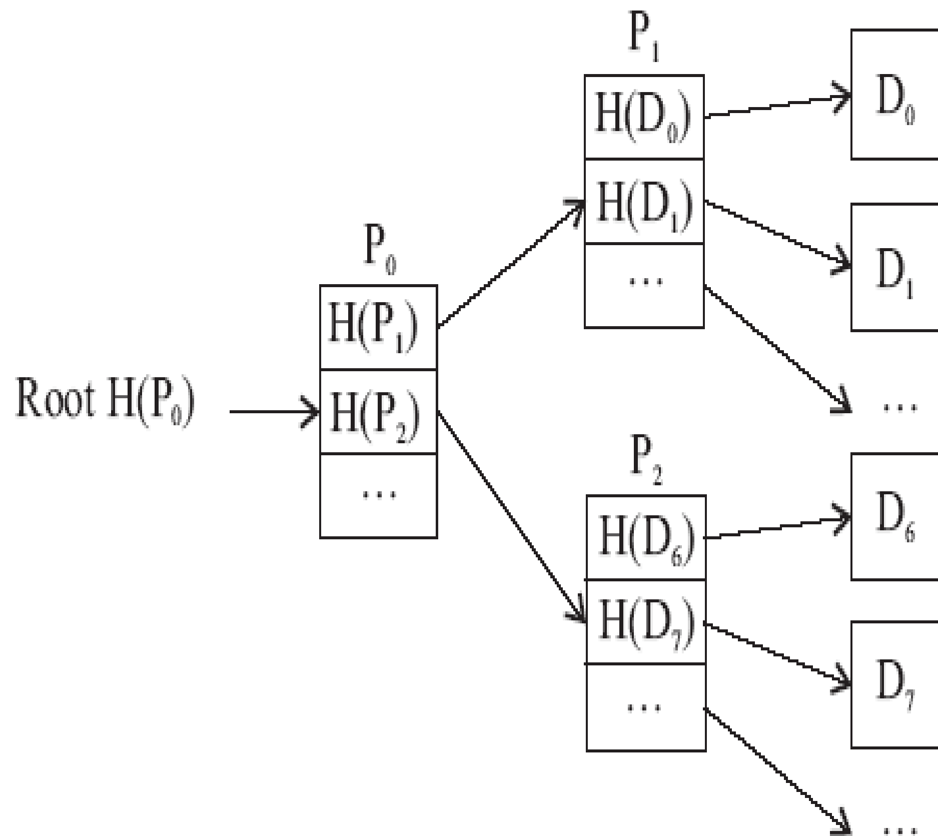
LBFS Summary

- Designed for low bandwidth (and possibly high latency) networks
- Achieves one order of magnitude less bandwidth than conventional file systems
  - Beats Shannon unless you're careful ;-)
- Related work includes:
  - rsync (rabin fingerprints);
  - and some ideas from web caching (e.g. transmit the address of data in the cache of the receiver)
- Still pretty cool though!

Venti (FAST 2002)

- **Motivation: Archival Storage** (viz. backup)
  - Traditionally uses tapes or optical drives, but these are slow (and weird to access)
  - Why not use magnetic disks? Cheap & fast
    - But subject to overwrite
- Venti idea: use magnetic disks to produce an ***immutable*** **(write-once) store**
  - Every file update is a new ‘version’
  - (c/f Elephant, Plan 9, etc)
- But what about directories?

Key Idea: Content Hash Addressing



- File contents are just a set of blocks
- Inode is a table/tree of block content hashes
- Directories map names to inode content hash
- And so on recursively…

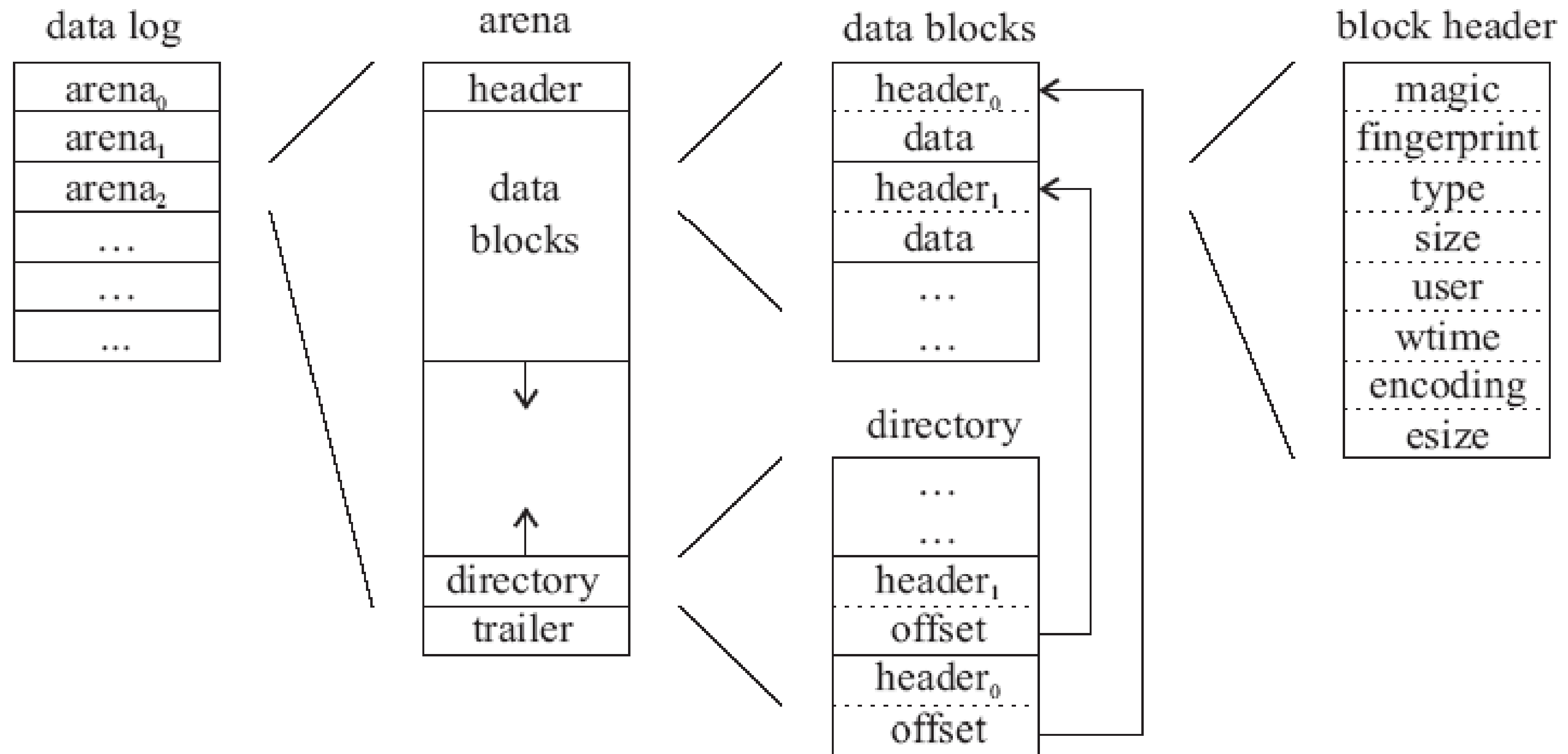
Some nice properties

- If a file changes (say one block), then that blocks content hash changes
  - Hence inode changes; and directory changes;
  - And so on right up the tree
- Hence fingerprint (= hash) of root directory uniquely captures entire file system!
- Same applies for any arbitrary subtree
  - Venti used this to implement 'vac' archival tool
  - Just creates 45 byte file including fingerprint ;-)

Implementation Challenges

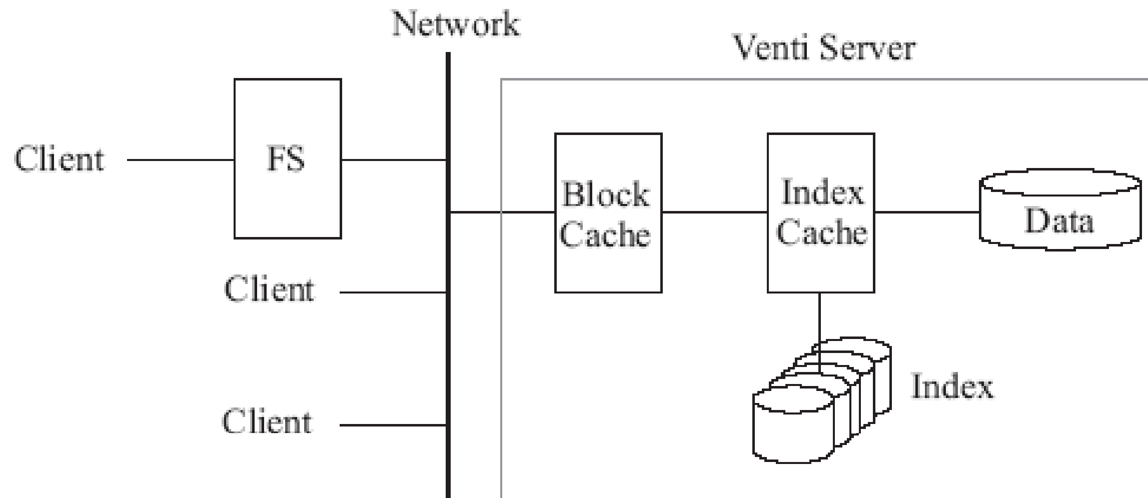
- If all blocks are addressed by content hashes, how do we actually ***find*** them on the disk?
- Venti uses log-based (append only) storage
  - Storage is arranged as an log of **arenas**
  - Each arena has a header and trailer each with some metadata (and also used for recovery)
  - Arena stores data blocks from the front (after header), and "directory" blocks from the back
    - "directory" here is just used to locate particular data blocks in this arena based on hash value

Storage Layout



- Ok, but still need to ***scan*** to locate blocks?

Caches and Indexes

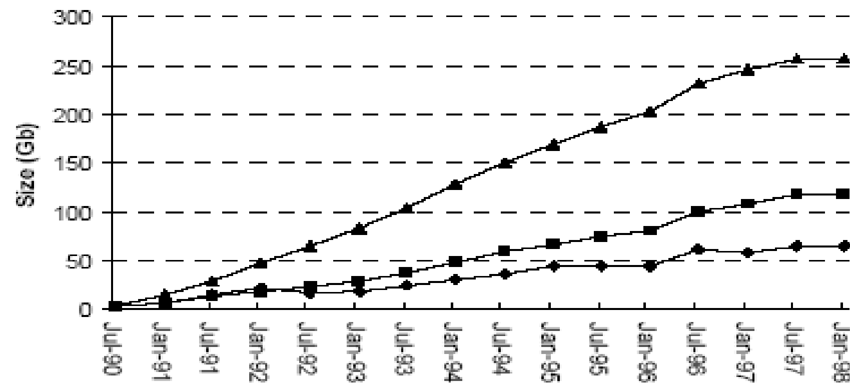


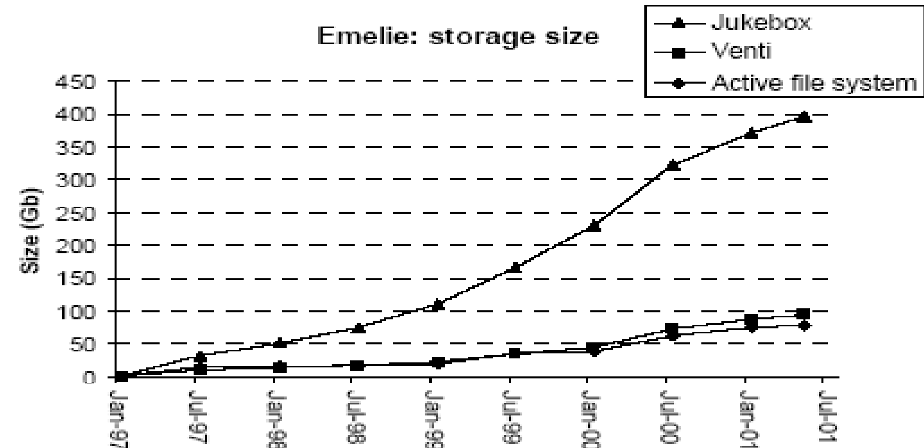
- To improve performance, Venti adds:
  - **On disk index** (hash->log location)
  - **Index cache** (in memory cache of index)
  - **Block cache** (direct hash->block function)

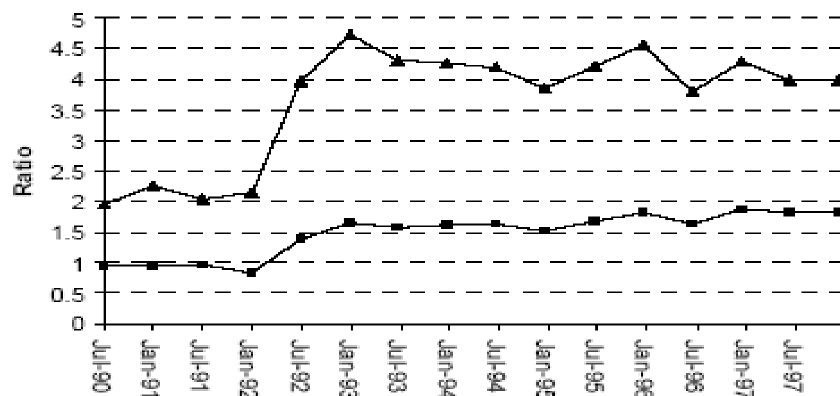
Evaluation

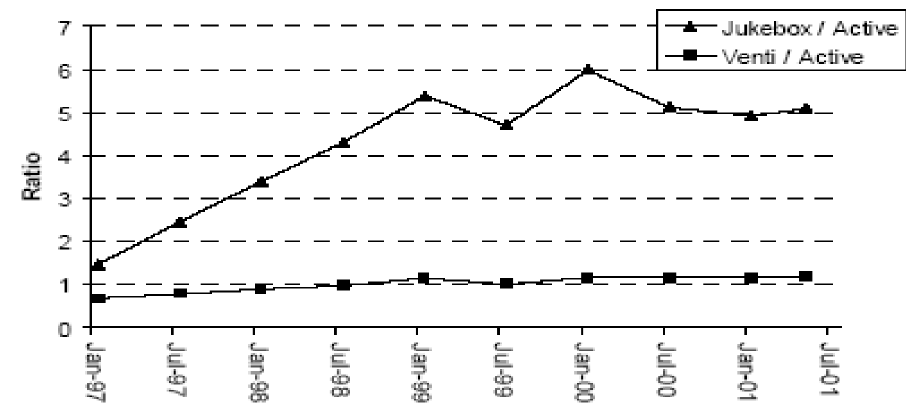
- Applied Venti to historical log of two plan-9 file servers, bootes and emelie
  - Spanning 1990 to 2001
  - Used plan-9 since does write-once storage
- 522 user accounts, 50-100 active all the time
- Numerous development projects hosted
- Several large data sets in used
  - Astronomical data, satellite imagery, multimedia files

Venti versus Plan-9 & Active FS









Venti: Summary

- **Addresses blocks by SHA-1 hash of contents**
  – Entire directory sub-tree defined by 20 byte hash!
- **Write once model**
  – Reduces accidental or malicious data loss
  – Simplifies administration
  – Simplifies caching
- **Magnetic disks as storage technology**
  – Large capacity at low price; fast random access
- **Reduces archive storage size by 50-75%**
  – Due to de-duplication, defrag and compression.

Summary & Outlook

We've seen a selection of systems topics:

- **Distributed and persistent virtual memory**
- **Microkernels** (Mach, L3/L4])
- **Extensible operating systems** (SPIN, Vino)
- **Vertically-structured operating systems** (Exokernel, Nemesis)
- **Virtual machine monitors** (VM/CVS, Disco, VMWare, Denali, Xen)
- **Distributed storage and filesystems** (NAS, SANs, NFS, AFS, Coda, NASD, xFS, Farsite, LBFS, Venti)

Lots more research ongoing in most of the above!