CS162
Operating Systems and Systems Programming
Lecture 27

Peer-to-peer Systems, ManyCore OSes and Other Topics

December 6th, 2010

Prof. John Kubiatowicz

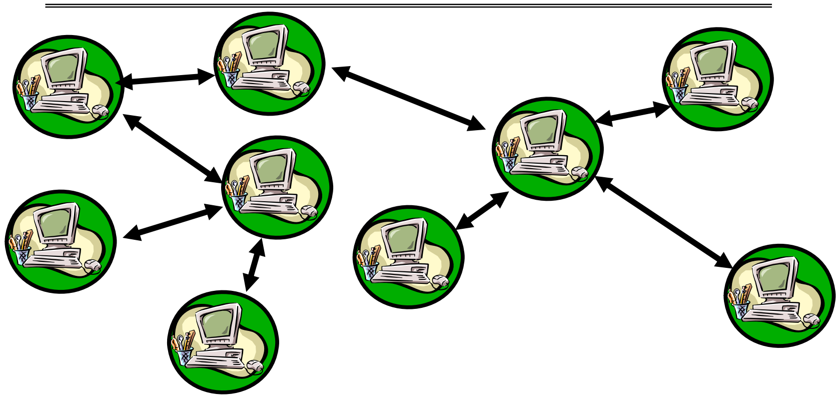
http://inst.eecs.berkeley.edu/~cs162

Goals for Today

- A couple of requested topics
  - Peer-to-Peer Systems
  - ManyCore OSes
  - Realtime OSs
  - Trusted Computing

Note: Some slides and/or pictures in the following are adapted from slides ©2005 Silberschatz, Galvin, and Gagne

Peer-to-Peer: Fully equivalent components



- Peer-to-Peer has many interacting components
  - View system as a set of equivalent nodes
    - » "All nodes are created equal"
  - Any structure on system must be self-organizing
    - » Not based on physical characteristics, location, or ownership

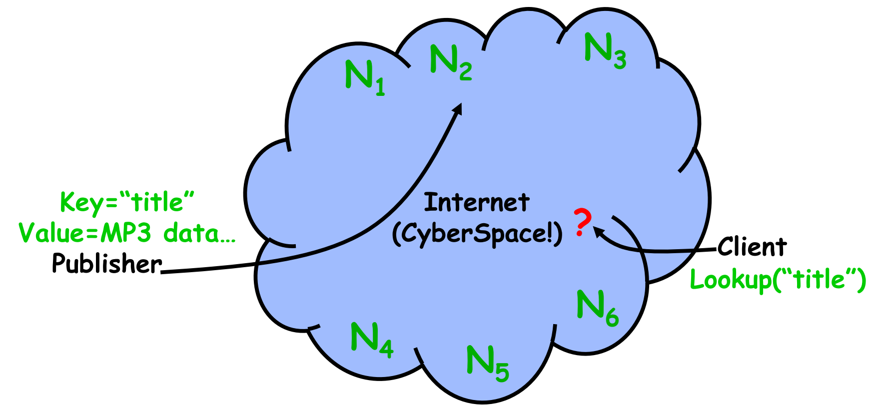
Research Community View of Peer-to-Peer

- Old View:
  - A bunch of flakey high-school students stealing music
- New View:
  - A philosophy of systems design at extreme scale
  - Probabilistic design when it is appropriate
  - New techniques aimed at unreliable components
  - A rethinking (and recasting) of distributed algorithms
  - Use of Physical, Biological, and Game-Theoretic techniques to achieve guarantees

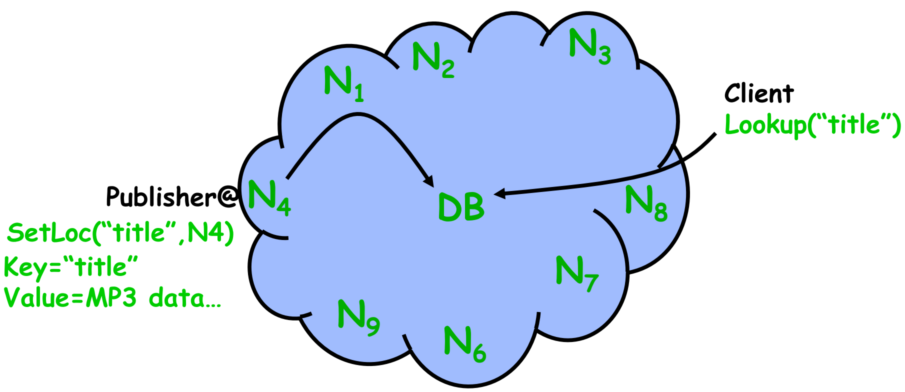
Why the hype???

- File Sharing: Napster (+Gnutella, KaZaa, etc)
  - Is this peer-to-peer? Hard to say.
  - Suddenly people could contribute to active global network
    - » High coolness factor
  - Served a high-demand niche: online jukebox
- Anonymity/Privacy/Anarchy: FreeNet, Publis, etc
  - Libertarian dream of freedom from the man
    - » (ISPs? Other 3-letter agencies)
  - Extremely valid concern of Censorship/Privacy
  - In search of copyright violators, RIAA challenging rights to privacy
- Computing: The Grid
  - Scavenge numerous free cycles of the world to do work
  - Seti@Home most visible version of this
- Management: Businesses
  - Businesses have discovered extreme distributed computing
  - Does P2P mean "self-configuring" from equivalent resources?
  - Bound up in "Autonomic Computing Initiative"?

The lookup problem

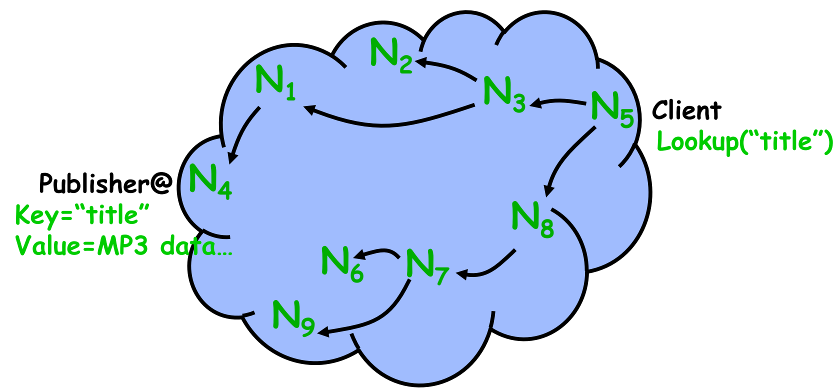


Centralized lookup (Napster)



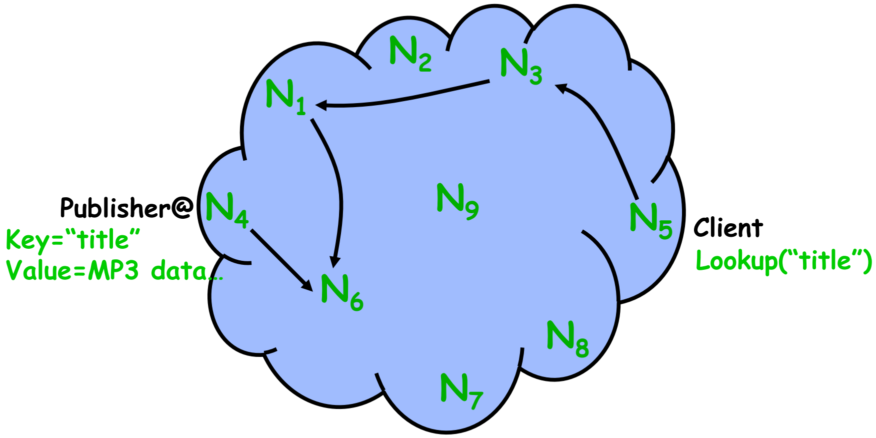
Simple, but $O(N)$ state and a single point of failure

Flooded queries (Gnutella)



Robust, but worst case $O(N)$ messages per lookup

Routed queries (Freenet, Chord, etc.)



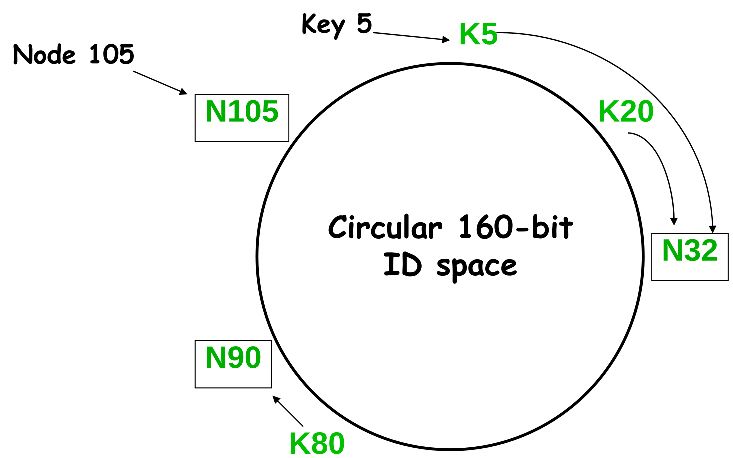
Can be O(log N) messages per lookup (or even O(1))
Potentially complex routing state and maintenance.

Chord IDs

- Key identifier = SHA-1(key)
- Node identifier = SHA-1(IP address)
- Both are uniformly distributed
- Both exist in the same ID space

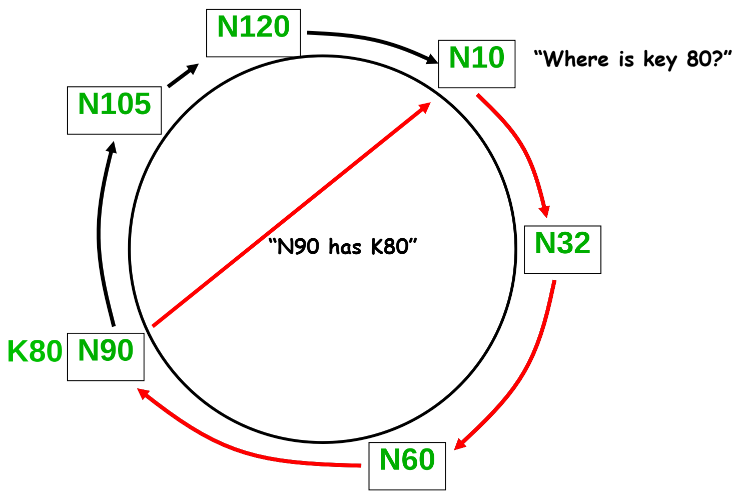
- How to map key IDs to node IDs?

Consistent hashing [Karger 97]



A key is stored at its successor: node with next higher ID

Basic lookup

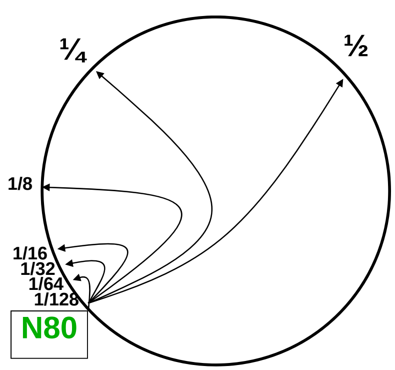


Simple lookup algorithm

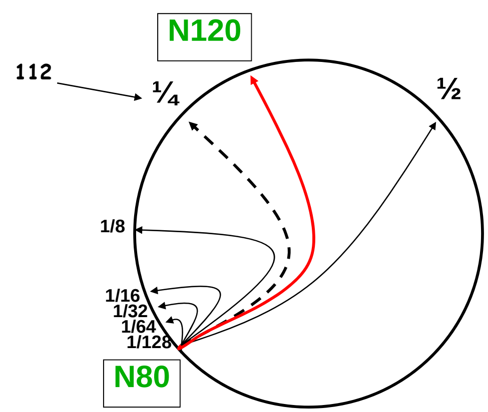
```
Lookup(my-id, key-id)
  n = my successor
  if my-id < n < key-id
     call Lookup(id) on node n     // next hop
  else
     return my successor           // done
```

- Correctness depends only on successors

"Finger table" allows log(N)-time lookups



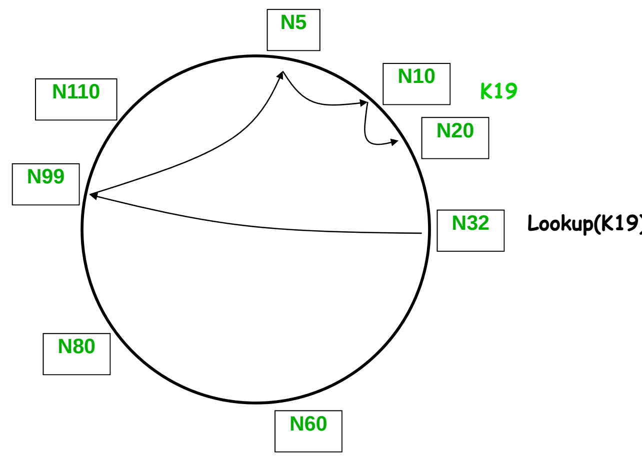
Finger *i* points to successor of $n+2^i$



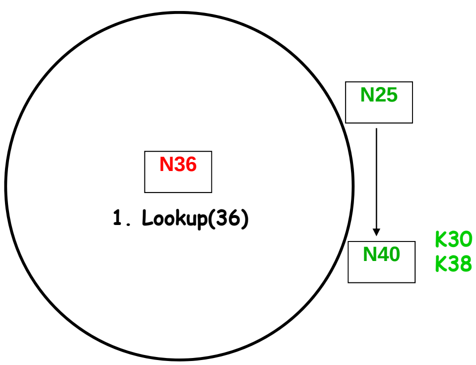
Lookup with fingers

```
Lookup(my-id, key-id)
  look in local finger table for
      highest node n s.t. my-id < n < key-id
  if n exists
      call Lookup(id) on node n   // next hop
  else
      return my successor         // done
```

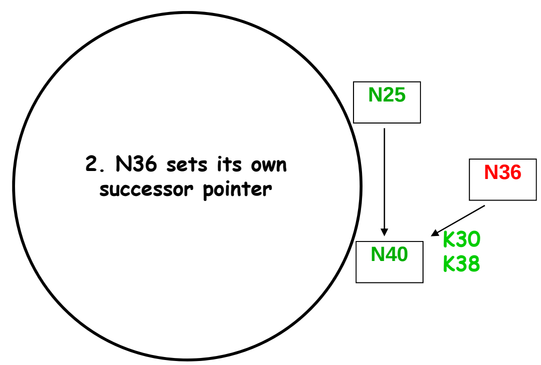

Lookups take O(*log(N)*) hops



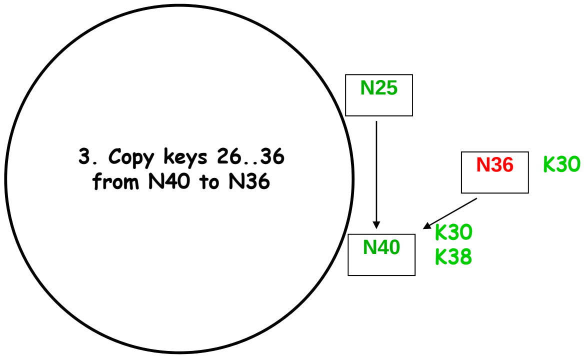
Joining: linked list insert



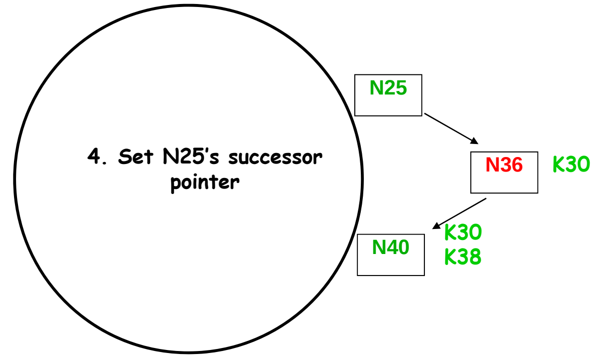
Join (2)



Join (3)

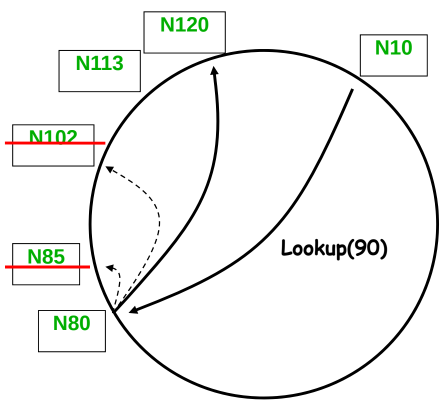


Join (4)



Update finger pointers in the background
Correct successors produce correct lookups

Failures might cause incorrect lookup



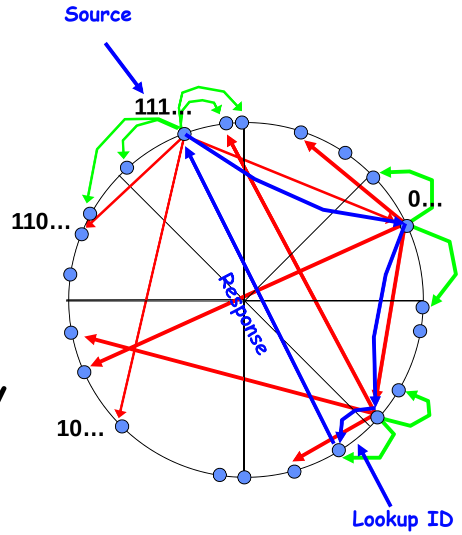
N80 doesn't know correct successor, so incorrect lookup

Solution: successor lists

- Each node knows *r* immediate successors
  - After failure, will know first live successor
  - Correct successors guarantee correct lookups
  - Guarantee is with some probability
- For many systems, talk about "leaf set"
  - The leaf set is a set of nodes around the "root" node that can handle all of the data/queries that the root nodes might handle
- When node fails:
  - Leaf set can handle queries for dead node
  - Leaf set queried to retreat missing data
  - Leaf set used to reconstruct new leaf set

Lookup with Leaf Set

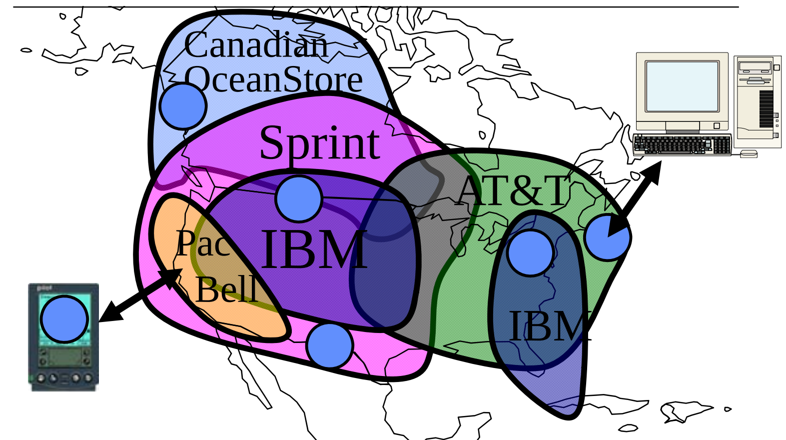
- Assign IDs to nodes
  - Map hash values to node with closest ID
- Leaf set is successors and predecessors
  - All that's needed for correctness
- Routing table matches successively longer prefixes
  - Allows efficient lookups



Administrivia

- Final Exam
  - Thursday 12/16, 8:00AM-11:00AM, 10 Evans Hall
  - All material from the course
    - » With slightly more focus on second half
  - Two sheets of notes, both sides
  - Will need dumb calculator
- Should be working on Project 4
  - Final Code due tomorrow (Tuesday 12/7)
  - Final Report due on next day
  - MAKE SURE TO FILL OUT YOUR GROUP EVALS!!
- I will have office hours this week at normal time
  - M/W 2:30-3:30
  - Feel free to come by to talk about whatever
- Need to get any regrade requests in by this Friday
  - i.e. Projects 1-3
  - Will consider Project 4 issues up until final

Utility-based Infrastructure



- Data service provided by storage federation
- Cross-administrative domain
- Contractual Quality of Service ("someone to sue")

OceanStore: Everyone's Data, One Big Utility

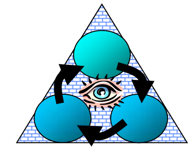
"The data is just out there"

- How many files in the OceanStore?
  - Assume 10^{10} people in world
  - Say 10,000 files/person (very conservative?)
  - So 10^{14} files in OceanStore!
  - If 1 gig files (ok, a stretch), get 1 mole of bytes! (or a Yotta-Byte if you are a computer person)

Truly impressive number of elements…
… but small relative to physical constants

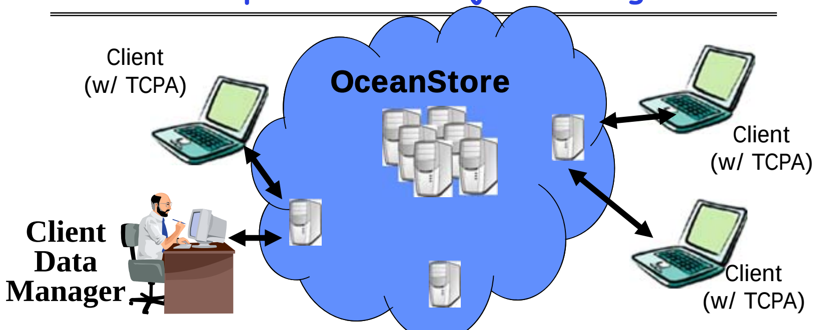
Key Observation: Want Automatic Maintenance

- Can't possibly manage billions of servers by hand!
- System should automatically:
  - Adapt to failure
  - Exclude malicious elements
  - Repair itself
  - Incorporate new elements



- System should be secure and private
  - Encryption, authentication
- System should preserve data over the long term (*accessible* for 1000 years):
  - Geographic distribution of information
  - New servers added from time to time
  - Old servers removed from time to time
  - Everything just works

Example: Secure Object Storage



- Security: Access and Content controlled by client
  - Privacy through data encryption
  - Optional use of cryptographic hardware for revocation
  - Authenticity through hashing and active integrity checking
- Flexible self-management and optimization:
  - Performance and durability
  - Efficient sharing

OceanStore Assumptions

Peer-to-peer

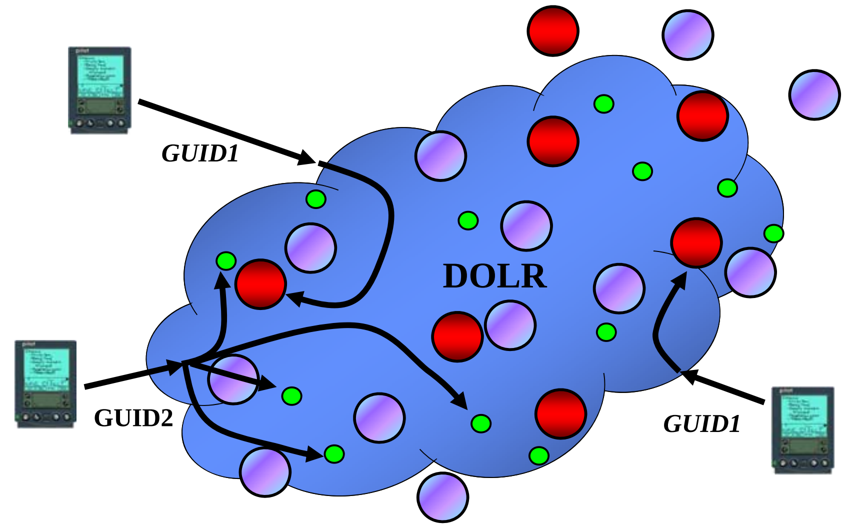
- Untrusted Infrastructure:
  - The OceanStore is comprised of untrusted components
  - Individual hardware has finite lifetimes
  - All data encrypted within the infrastructure
- Mostly Well-Connected:
  - Data producers and consumers are connected to a high-bandwidth network most of the time
  - Exploit multicast for quicker consistency when possible
- Promiscuous Caching:
  - Data may be cached anywhere, anytime

Quality-of-Service

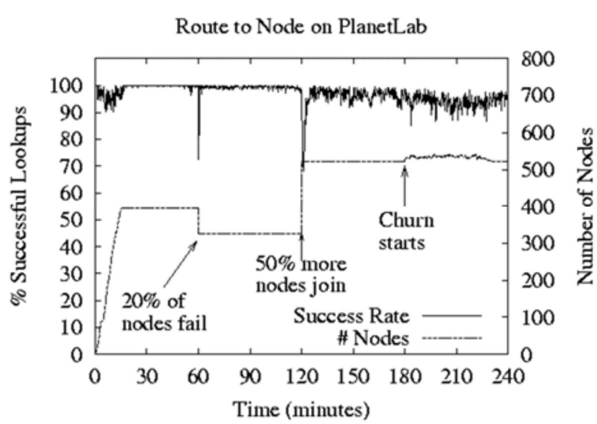
- Responsible Party:
  - Some organization (*i.e. service provider)* guarantees that your data is consistent and durable
  - Not trusted with *content* of data, merely its *integrity*

Peer-to-Peer for Data Location

Peer-to-Peer in OceanStore: DOLR
(Decentralized Object Location and Routing)

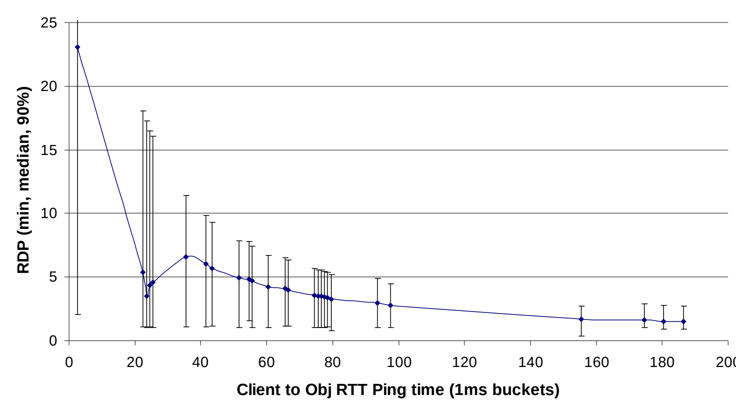


Stability under extreme circumstances



(May 2003: 1.5 TB over 4 hours)
DOLR Model generalizes to many simultaneous apps

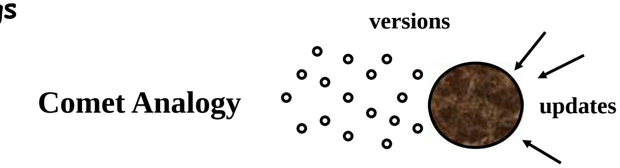
Object Location with Tapestry DOLR



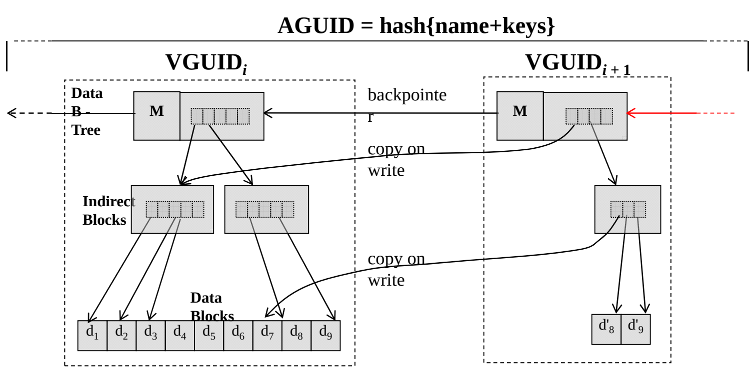
Peek at OceanStore Mechanisms

OceanStore Data Model

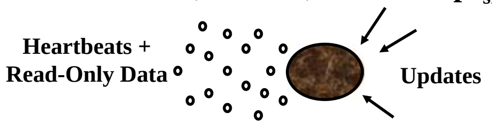
- Versioned Objects
  - Every update generates a new version
  - Can always go back in time (Time Travel)
- Each Version is Read-Only
  - Can have permanent name
  - Much easier to repair
- An Object is a signed mapping between permanent name and latest version
  - Write access control/integrity involves managing these mappings



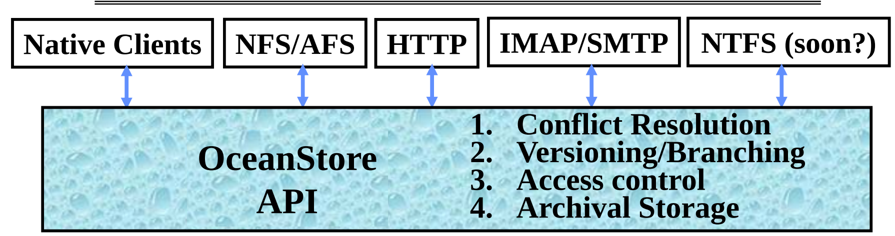
Self-Verifying Objects



♥Heartbeat: {AGUID,VGUID, Timestamp}$_{\text{signed}}$



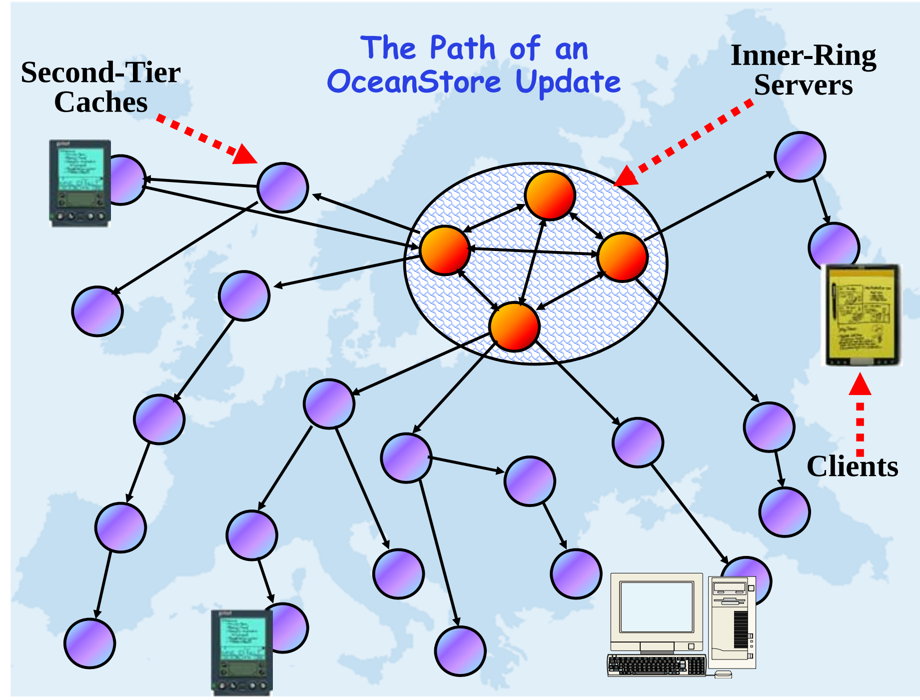
OceanStore API: Universal Conflict Resolution



- Consistency is form of optimistic concurrency
  - Updates contain predicate-action pairs
  - Each predicate tried in turn:
    - » If none match, the update is aborted
    - » Otherwise, action of first true predicate is applied
- Role of Responsible Party (RP):
  - Updates submitted to RP which chooses total order
- This is powerful enough to synthesize:
  - ACID database semantics
  - release consistency (build and use MCS-style locks)
  - Extremely loose (weak) consistency

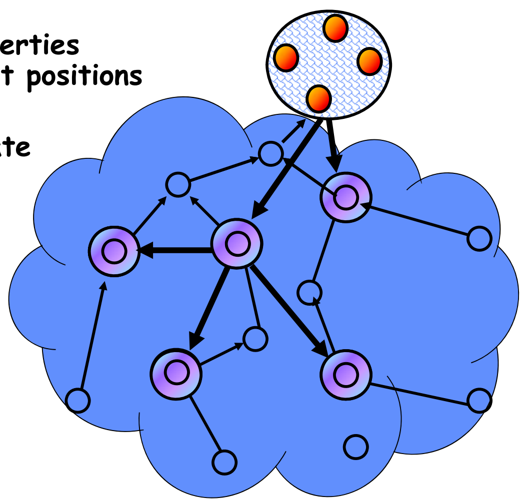
Two Types of OceanStore Data

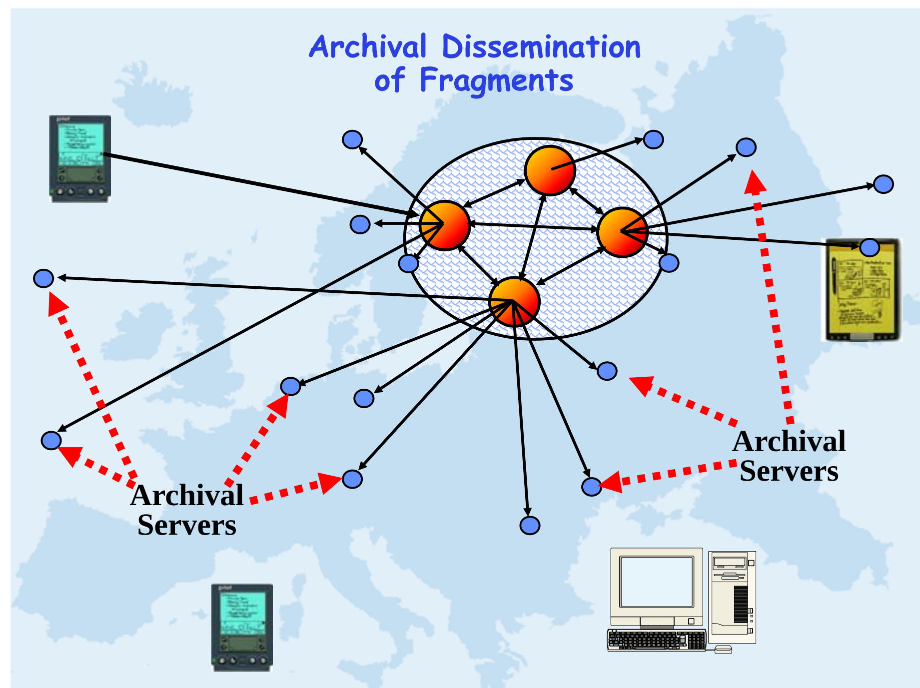
- *Active Data:* "Floating Replicas"
  - Per object virtual server
  - Interaction with other replicas for consistency
  - May appear and disappear like bubbles
- *Archival Data:* OceanStore's Stable Store
  - m-of-n coding: Like hologram
    - » Data coded into *n* fragments, any *m* of which are sufficient to reconstruct (e.g m=16, n=64)
    - » Coding overhead is proportional to n÷m (e.g 4)
    - » Other parameter, *rate*, is 1/overhead
  - Fragments are cryptographically self-verifying
- Most data in the OceanStore is archival!



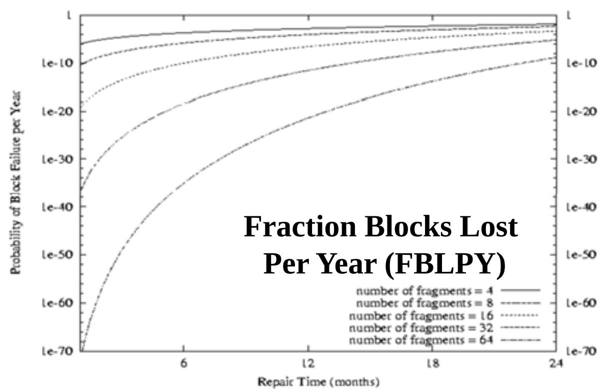
Self-Organizing Soft-State Replication

- Simple algorithms for placing replicas on nodes in the interior
  - Intuition: locality properties of Tapestry help select positions for replicas
  - Tapestry helps associate parents and children to build multicast tree
- Preliminary results encouraging
- Current Investigations:
  - Game Theory
  - Thermodynamics





Aside: Why erasure coding? High Durability/overhead ratio!



- Exploit law of large numbers for durability!
- 6 month repair, FBLPY:
  - Replication: 0.03
  - Fragmentation: 10^{-35}

Extreme Durability?

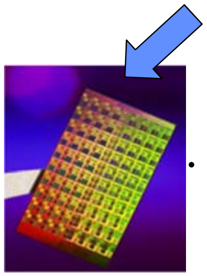
- Exploiting Infrastructure for Repair
  - DOLR permits efficient heartbeat mechanism to notice:
    - » Servers going away for a while
    - » Or, going away forever!
  - Continuous sweep through data also possible
  - Erasure Code provides Flexibility in Timing
- Data transferred from physical medium to physical medium
  - No "tapes decaying in basement"
  - Information becomes fully Virtualized
- Thermodynamic Analogy: Use of Energy (supplied by servers) to Suppress Entropy

Differing Degrees of Responsibility

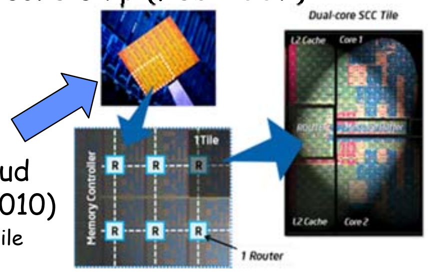
- Inner-ring provides quality of service
  - Handles of live data and write access control
  - Focus utility resources on this vital service
  - Compromised servers must be detected quickly
- Caching service can be provided by anyone
  - Data encrypted and self-verifying
  - Pay for service "Caching Kiosks"?
- Archival Storage and Repair
  - Read-only data: easier to authenticate and repair
  - Tradeoff redundancy for responsiveness
- Could be provided by different companies!

ManyCore
Operating Systems

ManyCore Chips: The future is here



- Intel 80-core multicore chip (Feb 2007)
  - 80 simple cores
  - Two FP-engines / core
  - Mesh-like network
  - 100 million transistors
  - 65nm feature size
- Intel Single-Chip Cloud Computer (August 2010)
  - 24 "tiles" with two cores/tile
  - 24-router mesh network
  - 4 DDR3 memory controllers
  - Hardware support for message-passing

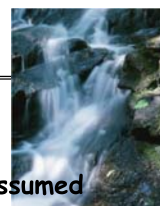


- "ManyCore" refers to many processors/chip
  - 64? 128? Hard to say exact boundary
- How to program these?
  - Use 2 CPUs for video/audio
  - Use 1 for word processor, 1 for browser
  - 76 for virus checking???
- **Parallelism must be exploited at all levels**

Services Support for Applications

- What systems support do we need for new ManyCore applications?
  - Should we just port parallel Linux or Windows 7 and be done with it?
  - A lot of functionality, hard to experiment with, possibly fragile, …
- Clearly, these new applications will contain:
  - Explicitly parallel components
    - » However, parallelism may be "hard won" (not embarrassingly parallel)
    - » Must not interfere with this parallelism
  - Direct interaction with Internet and "Cloud" services
    - » Potentially extensive use of remote services
    - » Serious security/data vulnerability concerns
  - Real Time requirements
    - » Sophisticated multimedia interactions
    - » Control of/interaction with health-related devices
  - Responsiveness Requirements
    - » Provide a good interactive experience to users

PARLab OS Goals: *RAPPidS*

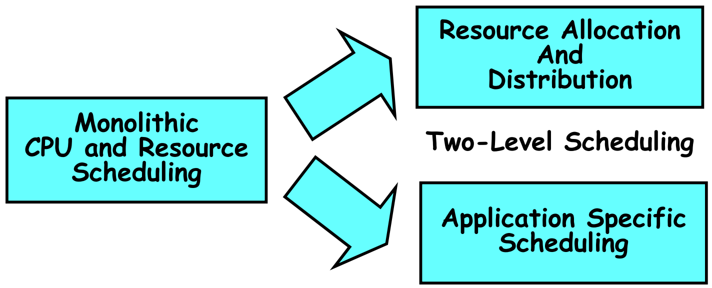


- **R**esponsiveness: Meets real-time guarantees
  - Good user experience with UI expected
  - Illusion of Rapid I/O while still providing guarantees
  - Real-Time applications (speech, music, video) will be assumed
- **A**gility: Can deal with rapidly changing environment
  - Programs not completely assembled until runtime
  - User may request complex mix of services at moment's notice
  - Resources change rapidly (bandwidth, power, etc)
- **P**ower-Efficiency: Efficient power-performance tradeoffs
  - Application-Specific parallel scheduling on Bare Metal partitions
  - Explicitly parallel, power-aware OS service architecture
- **P**ersistence: User experience persists across device failures
  - Fully integrated with persistent storage infrastructures
  - Customizations not be lost on "reboot"
- **S**ecurity and Correctness: Must be hard to compromise
  - Untrusted and/or buggy components handled gracefully
  - Combination of *verification* and *isolation* at many levels
  - Privacy, Integrity, Authenticity of information asserted

The Problem with Current OSs

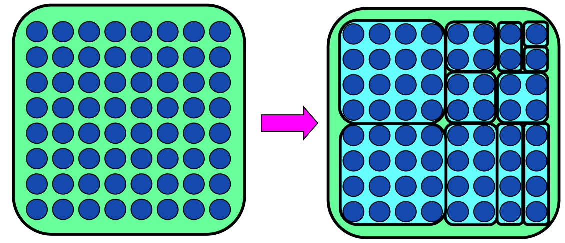
- What is wrong with current Operating Systems?
  - They (often?) do not allow expression of application requirements
    - » Minimal Frame Rate, Minimal Memory Bandwidth, Minimal QoS from system Services, Real Time Constraints, …
    - » No clean interfaces for reflecting these requirements
  - They (often?) do not provide guarantees that applications can use
    - » They do not provide performance isolation
    - » Resources can be removed or decreased without permission
    - » Maximum response time to events cannot be characterized
  - They (often?) do not provide fully custom scheduling
    - » In a parallel programming environment, ideal scheduling can depend crucially on the programming model
  - They (often?) do not provide sufficient Security or Correctness
    - » Monolithic Kernels get compromised all the time
    - » Applications cannot express domains of trust within themselves without using a heavyweight process model
- **The advent of ManyCore both:**
  - **Exacerbates the above with a greater number of shared resources**
  - **Provides an opportunity to change the fundamental model**

A First Step: Two Level Scheduling



- Split monolithic scheduling into two pieces:
  - Course-Grained Resource Allocation and Distribution
    - » Chunks of resources (CPUs, Memory Bandwidth, QoS to Services) distributed to application (system) components
    - » Option to simply turn off unused resources (Important for Power)
  - Fine-Grained Application-Specific Scheduling
    - » Applications are allowed to utilize their resources in any way they see fit
    - » Other components cannot interfere with their use of resources

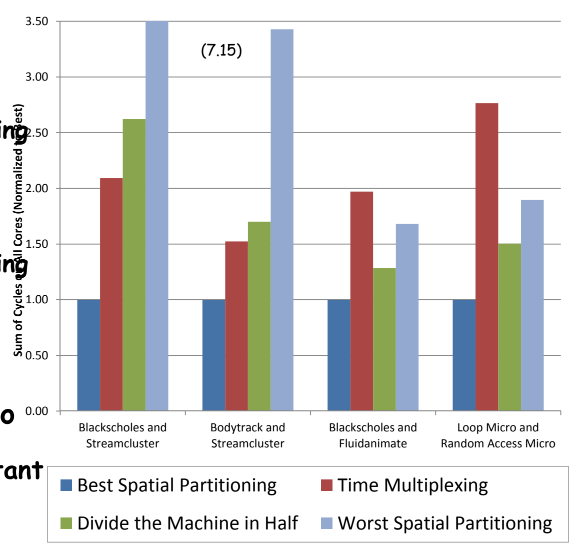
Important Idea: Spatial Partitioning



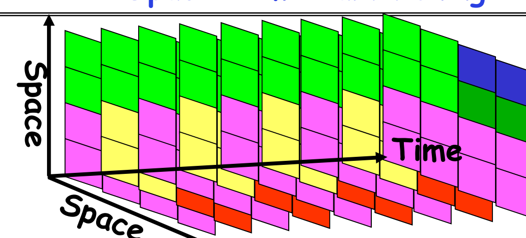
- Spatial Partition: group of processors within hardware boundary
  - Boundaries are "hard", communication between partitions controlled
  - Anything goes within partition
- Key Idea: Performance and Security Isolation
- Each Partition receives a vector of resources
  - Some number of dedicated processors
  - Some set of dedicated resources (exclusive access)
    - » Complete access to certain hardware devices
    - » Dedicated raw storage partition
  - Some guaranteed fraction of other resources (QoS guarantee):
    - » Memory bandwidth, Network bandwidth
    - » fractional services from other partitions

Performance w/ Spatial Partitioning

- RAMP Gold: FPGA-Based Emulator
  - 64 single-issue in-order cores
    - » Up to 8 slices using page coloring
  - Private L1 Inst and Data Caches
  - Shared L2 Cache
    - » Up to 8 slices using page coloring
  - Memory bandwidth partitionable into 3.4 GB/s units
- Spatial partitioning shows the potential to do quite well
  - However it is important to pick the right points.



Space-Time Partitioning

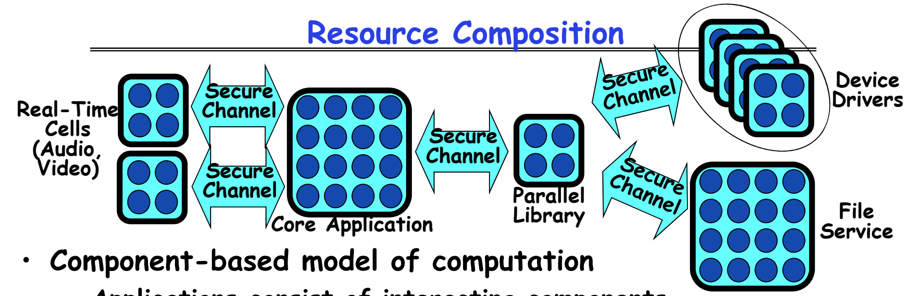


- Spatial Partitioning Varies over Time
  - Partitioning adapts to needs of the system
  - Some partitions persist, others change with time
  - Further, Partititions can be Time Multiplexed
    - » Services (i.e. file system), device drivers, hard realtime partitions
    - » Some user-level schedulers will time-multiplex threads within a partition
- Controlled Multiplexing, *not* uncontrolled virtualization
  - Multiplexing at coarser grain (100ms?)
  - Schedule planned several slices in advance
  - Resources gang-scheduled, use of affinity or hardware partitioning to avoid cross-partition interference

Defining the Partitioned Environment

- Our new abstraction: Cell
  - A user-level software component, with guaranteed resources
  - Is it a process? Is it a Virtual Private Machine? Neither, Both
  - Different from Typical Virtual Machine Environment which duplicates many Systems components in each VM
- Properties of a Cell
  - Has full control over resources it owns ("Bare Metal")
  - Contains at least one address space (memory protection domain), but could contain more than one
  - Contains a set of secured channel endpoints to other Cells
  - Contains a security context which may protect and decrypt information
  - Interacts with trusted layers of Tessellation (e.g. the "NanoVisor") via a heavily Paravirtualized Interface
    - » E.g. Manipulate address mappings without knowing format of page tables
- When mapped to the hardware, a Cell gets:
  - Gang-schedule hardware thread resources ("Harts")
  - Guaranteed fractions of other physical resources
    - » Physical Pages (DRAM), Cache partitions, memory bandwidth, power
  - Guaranteed fractions of system services

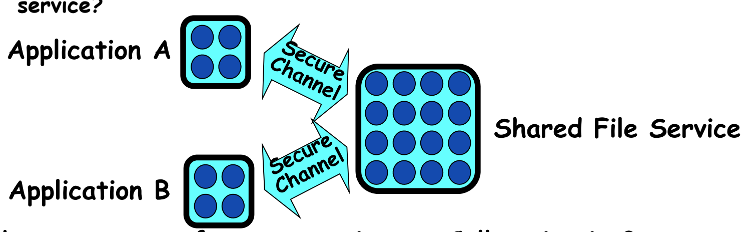
Resource Composition



- Component-based model of computation
  - Applications consist of interacting components
  - Produces composable: Performance, Interfaces, Security
- CoResident Cells ⇒ fast inter-domain communication
  - Could use hardware acceleration for fast secure messaging
  - Applications could be split into mutually distrusting partitions w/ controlled communication (echoes of μKernels)
- Fast Parallel Computation within Cells
  - Protection of computing resources not required within partition
    - » High walls between partitions ⇒ anything goes within partition
    - » Shared Memory/Message Passing/whatever within partition

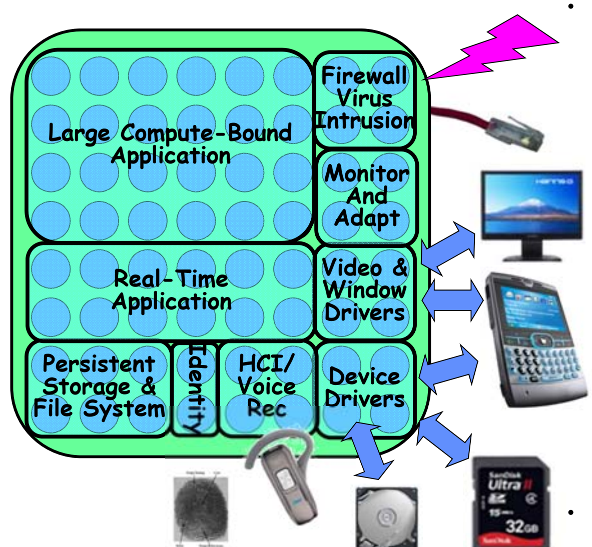
It's all about the communication

- We are interested in communication for many reasons:
  - Communication crosses resource and security boundaries
  - Efficiency of communication impacts (de)composability
- Shared components complicate resource isolation:
  - Need distributed mechanism for tracking and accounting of resources
    - » E.g.: How guarantee that each partition gets guaranteed fraction of service?



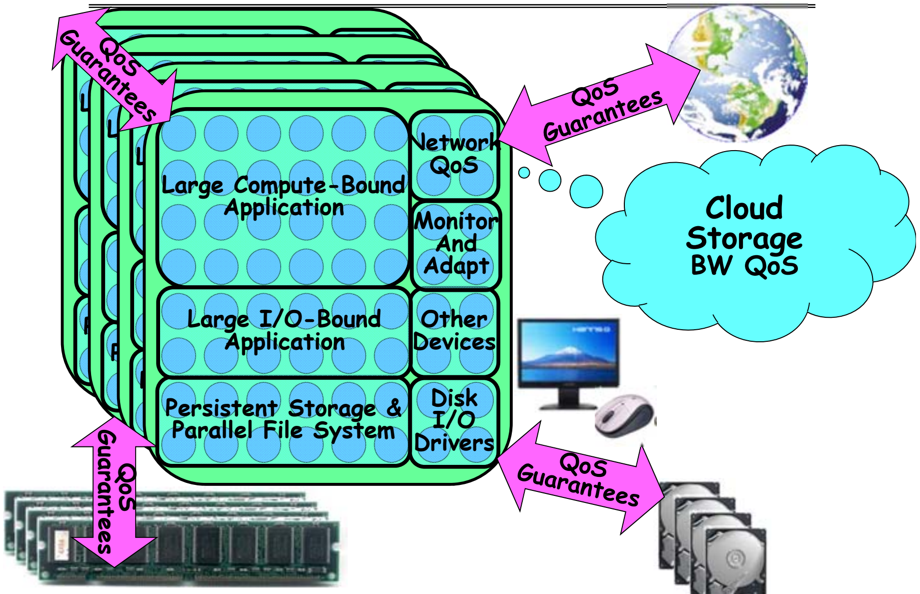
- How does presence of a message impact Cell activation?
  - Not at all (regular activation) or immediate change (interrupt-like)
- Communication defines Security Model
  - Mandatory Access Control Tagging (levels of information confidentiality)
  - Ring-based security (enforce call-gate structure with channels)

Tessellation: The Exploded OS



- Normal Components split into pieces
  - Device drivers (Security/Reliability)
  - Network Services (Performance)
    - » TCP/IP stack
    - » Firewall
    - » Virus Checking
    - » Intrusion Detection
  - Persistent Storage (Performance, Security, Reliability)
  - Monitoring services
    - » Performance counters
    - » Introspection
  - Identity/Environment services (Security)
    - » Biometric, GPS, Possession Tracking
- Applications Given Larger Partitions
  - Freedom to use resources arbitrarily

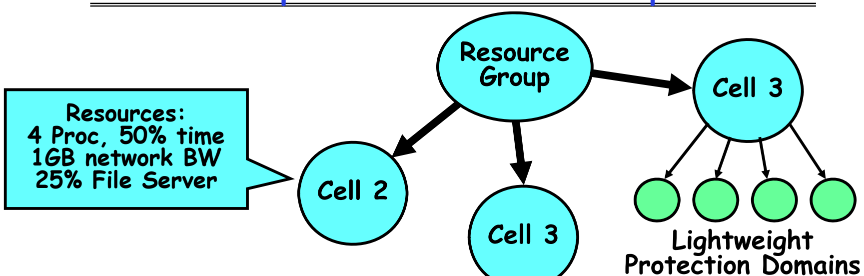
Tessellation in Server Environment



Another Look: Two-Level Scheduling

- First Level: Global partitioning of resources
  - Goals: Power Budget, Overall Responsiveness/QoS, Security
    - » Adjust resources to meet system level goals
  - Partitioning of CPUs, Memory, Interrupts, Devices, other resources
  - Constant for sufficient period of time to:
    - » Amortize cost of global decision making
    - » Allow time for partition-level scheduling to be effective
  - Hard boundaries ⇒ interference-free use of resources for quanta
    - » Allows AutoTuning of code to work well in partition
- Second Level: Application-Specific Scheduling
  - Goals: Performance, Real-time Behavior, Responsiveness, Predictability
    - » Fine-grained, rapid switching
  - CPU scheduling tuned to specific applications
  - Resources distributed in application-specific fashion
  - External events (I/O, active messages, etc) deferrable as appropriate

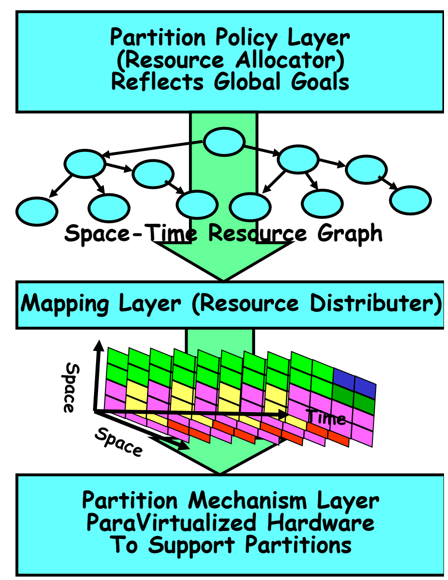
Space-Time Resource Graph



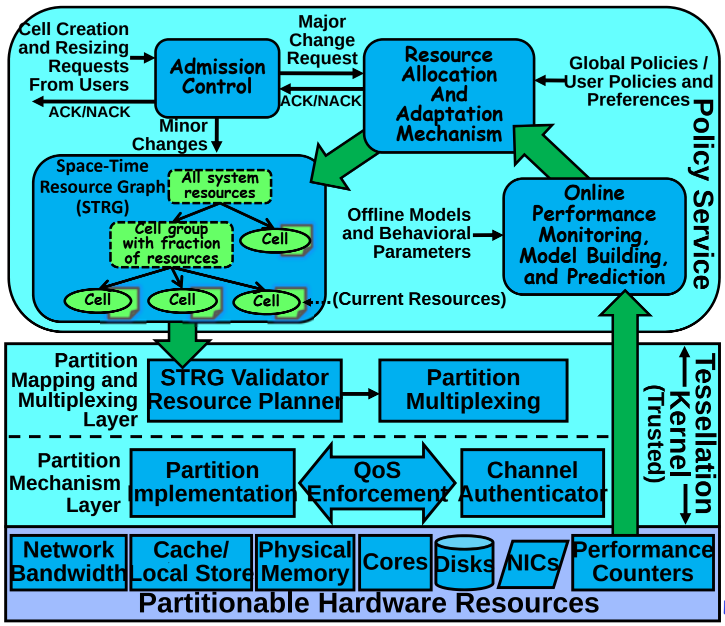
- Space-Time Resource Graph (STRG)
  - the explicit instantiation of resource assignments and relationships
- Leaves of graph hold Cells
  - All resources have a Space/Time component
    - » E.g. X Processors/fraction of time, or Y Bytes/Sec
  - Resources cannot be taken away except via explicit APIs
  - Resources include fractions of OS services
- Interior Nodes
  - Resource Groups can hold resources to be shared by children
  - "Pre-Allocated" resources can be shared as excess until needed
  - Some Similarity to Resource Containers

Implementing the Space-Time Graph

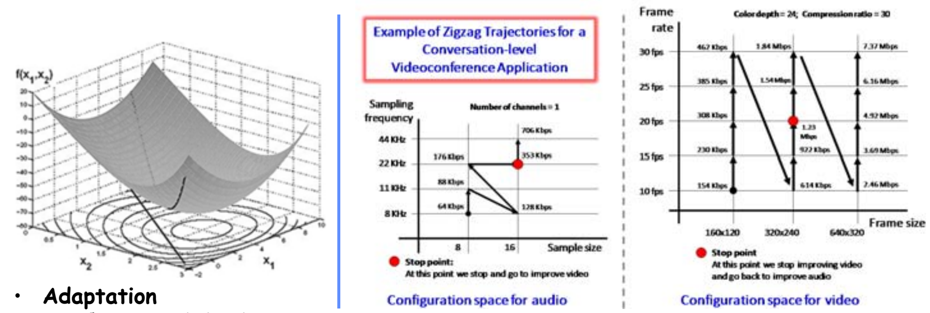
- Partition Policy Service (allocation)
  - Allocates Resources to Cells based on Global policies
  - Produces only implementable space-time resource graphs
  - May deny resources to a cell that requests them (admission control)
- Mapping Layer (distribution)
  - Makes no decisions
  - Time-Slices at a course granularity (when time-slicing necessary)
  - performs bin-packing like operation to implement space-time graph
  - In limit of *many* processors, no time multiplexing of processors, merely distributing of resources
- Partition Mechanism Layer
  - Implements hardware partitions and secure channels
  - Device Dependent: Makes use of more or less hardware support for QoS and Partitions



Resource Allocation Architecture



Modeling and Adaptation Policies



- Adaptation
  - Convex optimization
    - » Relative importance of different Cells expressed via scaling functions ("Urgency")
  - Walk through Configuration space
    - » Meet minimum QoS properties first, enhancement with excess resources
- User-Level Policies
  - Declarative language for describing application preferences and adaptive desires
- Modeling of Applications
  - Static Profiling: may be useful with Cell guarantees
  - Multi-variable model building
    - » Get performance as function of resources
    - » Or – tangent plane of performance as function of resources

Discussion

- How to divide application into Cell?
  - Cells probably best for coarser-grained components
    - » Fine-grained switching between Cells antithetical to stable resource guarantees
  - Division between Application components and shared OS services natural (obvious?)
    - » Both for security reasons and for functional reasons
  - Division between types of scheduling
    - » Real-time (both deadline-driven and rate-based), pre-scheduled
    - » GUI components (responsiveness most important)
    - » High-throughput (As many resources as can get)
    - » Stream-based (Parallelism through decomposition into pipeline stages)
- What granularity of Application component is best for Policy Service?
  - Fewer Cells in system leads to simpler optimization problem
- Language-support for Cell model?
  - Task-based, not thread based
  - Cells produced by annotating Software Frameworks with QoS needs?
  - Cells produced automatically by just-in-time optimization?
    - » i.e. Selective Just In Time Specialization or SEJITS

Scheduling inside a cell

- Cell Scheduler can rely on:
  - Coarse-grained time quanta allows efficient fine-grained use of resources
  - Gang-Scheduling of processors within a cell
  - No unexpected removal of resources
  - Full Control over arrival of events
    - » Can disable events, poll for events, etc.
- Pure environment of a Cell ⇒ Autotuning will return same performance at runtime as during training phase
- Application-specific scheduling for performance
  - Lithe Scheduler Framework (for constructing schedulers)
    - » Will be able to handle premptive scheduling/cross-address-space scheduling
  - Systematic mechanism for building composable schedulers
    - » Parallel libraries with different parallelism models can be easily composed
  - Of course: preconstructed thread schedulers/models (Silk, pthreads…) as libraries for application programmers
- Application-specific scheduling for Real-Time
  - Label Cell with Time-Based Labels. Examples:
    - » Run every 1s for 100ms synchronized to ± 5ms of a global time base
    - » Pin a cell to 100% of some set of processors
  - Then, maintain own deadline scheduler

What we might like from Hardware

- A good parallel computing platform (Obviously!)
  - Good synchronization, communication (Shared memory would be nice)
  - Vector, GPU, SIMD (Can exploit data parallel modes of computation)
  - Measurement: performance counters
- Partitioning Support
  - Caches: Give exclusive chunks of cache to partitions
  - High-performance barrier mechanisms partitioned properly
  - System Bandwidth
  - Power (Ability to put partitions to sleep, wake them up quickly)
- QoS Enforcement Mechanisms
  - Ability to give restricted fractions of bandwidth (memory, on-chip network)
  - Message Interface: Tracking of message rates with source-suppression for QoS
  - Examples: Globally Synchronized Frames (ISCA 2008, Lee and Asanovic)
- Fast messaging support (for channels and possible intra-cell)
  - Virtualized endpoints (direct to destination Cell when mapped, into memory FIFO when not)
  - User-level construction and disposition of messages
  - DMA, user-level notification mechanisms
  - Trusted Computing Platform (automatic decryption/encryption of channel data)

Realtime OS/Embedded Applications

- Embedded applications:
  - Limited Hardware
  - Dedicated to some particular task
  - Examples: 50-100 CPUs in modern car!
- What does it mean to be "Realtime"?
  - Meeting time-related goals in the real world
    - » For instance: to show video, need to display X frames/sec
  - Hard real-time task:
    - » one which we must meet its deadline
    - » otherwise, fatal damage or error will occur.
  - Soft real-time task:
    - » one which we should meet its deadline, but not mandatory.
    - » We should schedule it even if the deadline
  - Firm real time
    - » Result has no utility outside deadline window, but system can withstand a few missed results
- Determinism:
  - Sometimes, deterministic behavior is more important than high performance

Type of Real-Time Scheduling

- Dynamic vs. Static
  - Dynamic schedule computed at run-time based on tasks really executing
  - Static schedule done at compile time for all *possible* tasks
- Preemptive permits one task to preempt another one of lower priority
- Schedulability:
  - NP-hard if there are any resources dependencies
  - Options:
    - » Prove it definitely cannot be scheduled
    - » Find a schedule if it is easy to do
    - » Stuck in the middle somewhere

Scheduling Parameters

- Assume N CPUs available for execution of a single task set
- Set of tasks {Ti}
  - Periods pi
  - Deadline di (completion deadline after task is queued)
  - Execution time ci (amount of CPU time to complete)
- Handy values:
  - Laxity li = di – ci (amount of slack time before Ti *must* begin execution)
  - Utilization factor ui = ci/pi(portion of CPU used)

Static Schedule

- Assume non-preemptive system with 5 Restrictions:
  1. Tasks {Ti} are periodic, with hard deadlines and no jitter
  2. Tasks are completely independent
  3. Deadline = period pi = di
  4. Computation time ci is known and constant
  5. Context switching is free (zero cost) INCLUDING network messages to send context to another CPU(!)

Static Schedule

- Consider least common multiple of periods pi
  - This considers all possible cases of period phase differences
  - Worst case is time that is product of all periods; usually not that bad
  - If you can figure out (somehow) how to schedule this, you win
- Performance
  - Optimal if all tasks always run; can get up to 100% utilization
  - If it runs once, it will always work

EDF: Earliest Deadline First

- Assume a *preemptive* system with *dynamic* priorities, and (same 5 restrictions)
- Scheduling policy:
  - Always execute the task with the nearest deadline
- Performance
  - Optimal for uniprocessor (supports up to 100% of CPU usage in all situations)
  - If you're overloaded, ensures that a lot of tasks don't complete
    - » Everyone gets a chance to fail at expense of later tasks
- Variation: Constant Bandwidth Service (CBS)
  - Allows one or more of the EDF-scheduled tasks to be scheduled as "servers" with a guaranteed (minimum) fraction of the CPU
  - When deadline is "up", simply go on to next task and refresh the total fraction of CPU time for later use
    - » Set new deadline in future and new maximum CPU time

Least Laxity

- Assume a *preemptive* system with *dynamic* priorities, and (same 5 restrictions)
- Scheduling policy:
  - Always execute the task with the smallest laxity
- Performance:
  - Optimal for uniprocessor (supports up to 100% of CPU usage in all situations)
    - » Similar in properties to EDF
  - A little more general than EDF for multiprocessors
    - » Takes into account that slack time is more meaningful than deadline for tasks of mixed computing sizes
  - Probably more graceful degradations
    - » Laxity measure can dump tasks that are hopeless causes

EDF/Least Laxity Tradeoffs

- Pro:
  - If it works, it can get 100% efficiency (on a uniprocessor)
- Con:
  - It is not always feasible to prove that it will work in all cases
    - » And having it work for a while doesn't mean it will always work
  - Requires dynamic prioritization
  - The laxity time hack for global priority has limits
    - » May take too many bits to achieve fine-grain temporal ordering
    - » May take too many bits to achieve a long enough time horizon

Rate Monotonic

- Assume a *preemptive* system with *static* priorities, and (same 5 restrictions) plus
- Scheduling policy:
  - Highest static priority goes to shortest period; always execute highest priority

$$\mu = \sum \mu_i = \sum \frac{C_i}{p_i} \le N(2^{\frac{1}{N}} - 1) \qquad ; \mu \approx 0.7 \text{ for large N}$$

- Performance:
  - Provides a *guarantee* for schedulability with CPU load of ~70%
    - » Even with arbitrarily selected task periods
    - » Can do better if you know about periods & offsets
  - If all periods are multiple of shortest period, works for CPU load of 100%

Trusted Computing

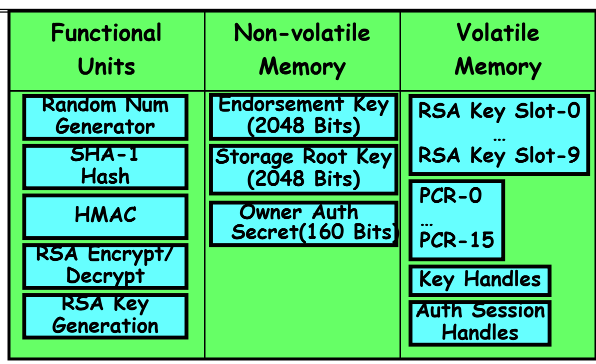
- Problem: Can't trust that software is correct
  - Viruses/Worms install themselves into kernel or system without users knowledge
  - **Rootkit:** software tools to conceal running processes, files or system data, which helps an intruder maintain access to a system without the user's knowledge
  - How do you know that software won't leak private information or further compromise user's access?
- A solution: What if there were a secure way to validate all software running on system?
  - Idea: Compute a cryptographic hash of BIOS, Kernel, crucial programs, etc.
  - Then, if hashes don't match, know have problem
- Further extension:
  - **Secure attestation:** ability to *prove* to a remote party that local machine is running correct software
  - Reason: allow remote user to avoid interacting with compromised system
- Challenge: How to do this in an unhackable way
  - Must have hardware components somewhere

TCPA: Trusted Computing Platform Alliance

- Idea: Add a Trusted Platform Module (TPM)
- Founded in 1999: Compaq, HP, IBM, Intel, Microsoft
- Currently more than 200 members
- Changes to platform
  - Extra: Trusted Platform Module (TPM)
  - Software changes: BIOS + OS
- Main properties
  - Secure bootstrap
  - Platform attestation
  - Protected storage
- Microsoft version:
  - Palladium
  - Note quite same: More extensive hardware/software system

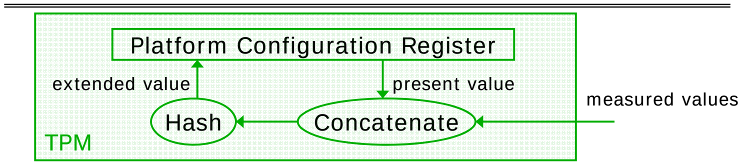
ATMEL TPM Chip
(Used in IBM equipment)

Trusted Platform Module

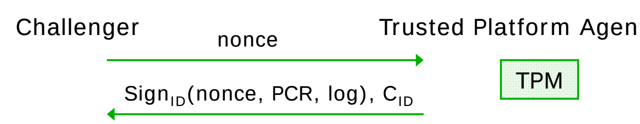


- Cryptographic operations
  - Hashing: SHA-1, HMAC
  - Random number generator
  - Asymmetric key generation: RSA (512, 1024, 2048)
  - Asymmetric encryption/ decryption: RSA
  - *Symmetric encryption/ decryption: DES, 3DES (AES)*
- Tamper resistant (hash and key) storage

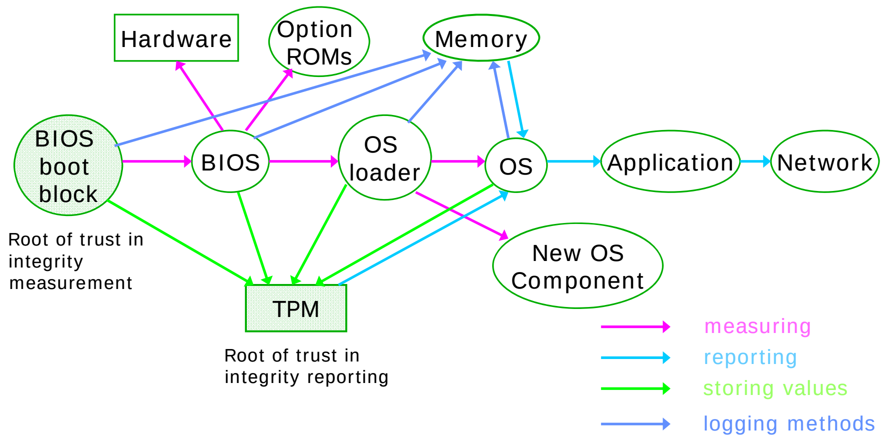
TCPA: PCR Reporting Value



- Platform Configuration Registers (PCR0-16)
  - Reset at boot time to well defined value
  - Only thing that software can do is give new measured value to TPM
    - » TPM takes new value, concatenates with old value, then hashes result together for new PCR
- Measuring involves hashing components of software
- Integrity reporting: report the value of the PCR
  - Challenge-response protocol:



TCPA: Secure bootstrap



Implications of TPM Philosophy?

- Could have great benefits
  - Prevent use of malicious software
  - Parts of OceanStore would benefit
- What does "trusted computing" really mean?
  - You are forced to trust hardware to be correct!
  - Could also mean that user is not trusted to install their own software
- Many in the security community have talked about potential abuses
  - These are only theoretical, but very possible
  - Software fixing
    - » What if companies prevent user from accessing their websites with non-Microsoft browser?
    - » Possible to encrypt data and only decrypt if software still matches ⇒ Could prevent display of .doc files except on Microsoft versions of software
  - Digital Rights Management (DRM):
    - » Prevent playing of music/video except on accepted players
    - » Selling of CDs that only play 3 times?

Conclusion

- Peer to Peer
  - A philosophy of systems design at extreme scale
  - Probabilistic design when it is appropriate
  - New techniques aimed at unreliable components
  - A rethinking (and recasting) of distributed algorithms
- Space-Time Partitioning: grouping processors & resources behind hardware boundary
  - Two-level scheduling
    - » Global Distribution of resources
    - » Application-Specific scheduling of resources
  - Cells: Basic Unit of Resource and Security
    - » User-Level Software Component with Guaranteed Resources
    - » Secure Channels to other Cells
- Tessellation OS
  - Exploded OS: spatially partitioned, interacting services
  - Check out: http://parlab.eecs.berkeley.edu

Conclusion (Con't)

- Realtime OS
  - Provide Guaranteed behavior to applications
  - Guarantees of:
    - » Meeting deadlines (time)
    - » Meeting throughput requirements (rate)
  - Tessellation provides better support for Realtime
    - » By providing resource-isolated Cells, applications get better chance to meet realtime scheduling guaranees
- Trusted Hardware
  - A secure layer of hardware that can:
    - » Generate proofs about software running on the machine
    - » Allow secure access to information without revealing keys to (potentially) compromised layers of software
  - Cannonical example: TPM