EECS 262a
Advanced Topics in Computer Systems
Lecture 23

BigTable/Pond
April 18th, 2016

John Kubiatowicz
Electrical Engineering and Computer Sciences
University of California, Berkeley

http://www.eecs.berkeley.edu/~kubitron/cs262

Today's Papers

- Bigtable: a distributed storage system for structured data. Appears in *Proceedings of the 7th Conference on USENIX Symposium on Operating Systems Design and Implementation* (OSDI), 2006
- Pond: the OceanStore Prototype. Appears in *Proceedings of the 2nd USENIX Conference on File and Storage Technologies* (FAST), 2003

- Thoughts?

BigTable

- Distributed storage system for managing structured data
- Designed to scale to a very large size
  - Petabytes of data across thousands of servers
- Hugely successful within Google – used for many Google projects
  - Web indexing, Personalized Search, Google Earth, Google Analytics, Google Finance, …
- Highly-available, reliable, flexible, high-performance solution for all of Google's products
- Offshoots/followons:
  - Spanner: Time-based consistency
  - LevelDB: Open source incorporating aspects of Big Table

Motivation

- Lots of (semi-)structured data at Google
  - URLs:
    - » Contents, crawl metadata, links, anchors, pagerank, …
  - Per-user data:
    - » User preference settings, recent queries/search results, …
  - Geographic locations:
    - » Physical entities (shops, restaurants, etc.), roads, satellite image data, user annotations, …
- Big Data scale
  - Billions of URLs, many versions/page (~20K/version)
  - Hundreds of millions of users, thousands or q/sec
  - 100TB+ of satellite image data

What about a Parallel DBMS?

- Data is too large scale!
- Using a commercial approach would be too expensive
  - Building internally means system can be applied across many projects for low incremental cost
- Low-level storage optimizations significantly improve performance
  - Difficult to do when running on a DBMS

Goals

- A general-purpose data-center storage system
- Asynchronous processes continuously updating different pieces of data
  - Access most current data at any time
  - Examine changing data (e.g., multiple web page crawls)
- Need to support:
  - Durability, high availability, and very large scale
  - Big or little objects
  - Very high read/write rates (millions of ops per second)
  - Ordered keys and notion of locality
    - » Efficient scans over all or interesting subsets of data
    - » Efficient joins of large one-to-one and one-to-many datasets

BigTable

- Distributed multi-level map
- Fault-tolerant, persistent
- Scalable
  - Thousands of servers
  - Terabytes of in-memory data
  - Petabyte of disk-based data
  - Millions of reads/writes per second, efficient scans
- Self-managing
  - Servers can be added/removed dynamically
  - Servers adjust to load imbalance

Building Blocks

- Building blocks:
  - Google File System (GFS): Raw storage
  - Scheduler: schedules jobs onto machines
  - Lock service: distributed lock manager
  - MapReduce: simplified large-scale data processing
- BigTable uses of building blocks:
  - GFS: stores persistent data (SSTable file format for storage of data)
  - Scheduler: schedules jobs involved in BigTable serving
  - Lock service: master election, location bootstrapping
  - Map Reduce: often used to read/write BigTable data

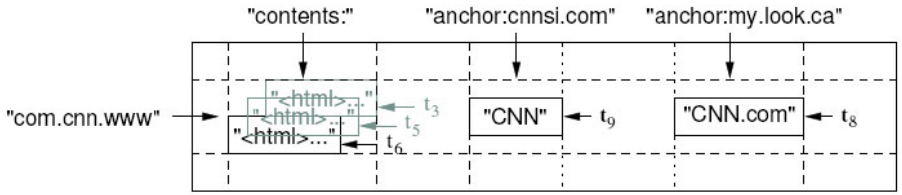
BigTable Data Model

- A big sparse sparse, distributed persistent multi-dimensional sorted map
  - Rows are sort order
  - Atomic operations on single rows
  - Scan rows in order
  - Locality by rows first
- Columns: properties of the row
  - Variable schema: easily create new columns
  - Column families: groups of columns
    - » For access control (e.g. private data)
    - » For locality (read these columns together, with nothing else)
    - » Harder to create new families
- Multiple entries per cell using timestamps
  - Enables multi-version concurrency control across rows

Basic Data Model

- Multiple entries per cell using timestamps
  - Enables multi-version concurrency control across rows

(row, column, timestamp) ➔ cell contents



- Good match for most Google applications:
  - Large collection of web pages and related information
  - Use URLs as row keys
  - Various aspects of web page as column names
  - Store contents of web pages in the `contents:` column under the timestamps when they were fetched

Rows

- Row creation is implicit upon storing data
  - Rows ordered lexicographically
  - Rows close together lexicographically usually on one or a small number of machines
- Reads of short row ranges are efficient and typically require communication with a small number of machines
- Can exploit this property by selecting row keys so they get good locality for data access
  - Example:

```
math.gatech.edu, math.uga.edu, phys.gatech.edu, phys.uga.edu
VS
edu.gatech.math, edu.gatech.phys, edu.uga.math, edu.uga.phys
```


Columns

- Columns have two-level name structure:
  - » family:optional_qualifier
- Column family
  - Unit of access control
  - Has associated type information
- Qualifier gives unbounded columns
  - Additional levels of indexing, if desired

Timestamps

- Used to store different versions of data in a cell
  - New writes default to current time, but timestamps for writes can also be set explicitly by clients
- Lookup options:
  - *"Return most recent K values"*
  - *"Return all values in timestamp range (or all values)"*
- Column families can be marked w/ attributes:
  - *"Only retain most recent K values in a cell"*
  - *"Keep values until they are older than K seconds"*

Basic Implementation

- Writes go to log then to in-memory table "memtable" (key, value)
- Periodically: move in memory table to disk => SSTable(s)
  - "*Minor compaction*"
    - » Frees up memory
    - » Reduces recovery time (less log to scan)
  - SSTable = immutable ordered subset of table: range of keys and subset of their columns
    - » One locality group per SSTable (for columns)
  - Tablet = all of the SSTables for one key range + the memtable
    - » Tablets get split when they get too big
    - » SSTables can be shared after the split (immutable)
  - Some values may be stale (due to new writes to those keys)

Basic Implementation

- Reads: maintain in-memory map of keys to {SSTables, memtable}
  - Current version is in exactly one SSTable or memtable
  - Reading based on timestamp requires multiple reads
  - May also have to read many SSTables to get all of the columns
- Scan = merge-sort like merge of SSTables in order
  - "*Merging compaction*" – reduce number of SSTables
  - Easy since they are in sorted order
- Compaction
  - SSTables similar to segments in LFS
  - Need to "clean" old SSTables to reclaim space
    - » Also to *actually* delete private data
  - Clean by merging multiple SSTables into one new one
    - » "*Major compaction*" => merge all tables

Locality Groups

- Group column families together into an SSTable
  - Avoid mingling data, ie page contents and page metadata
  - Can keep some groups all in memory
- Can compress locality groups
- Bloom Filters on locality groups – avoid searching SSTable
  - Efficient test for set membership: member(key) ➔ true/false
    - » False => definitely not in the set, no need for lookup
    - » True => probably is in the set (do lookup to make sure and get value)
  - Generally supports adding elements, but not removing them
    - » … but some tricks to fix this (counting)
    - » … or just create a new set once in a while

Bloom Filters

- Basic version:
  - m bit positions
  - k hash functions
  - for insert: compute k bit locations, set them to 1
  - for lookup: compute k bit locations
    - » all = 1 => return true (may be wrong)
    - » any = 0 => return false
  - 1% error rate ~ 10 bits/element
    - » Good to have some a priori idea of the target set size
- Use in BigTable
  - Avoid reading all SSTables for elements that are not present (at least mostly avoid it)
    - » Saves many seeks

Three Part Implementation

- Client library with the API (like DDS)
- Tablet servers that serve parts of several tables
- Master that tracks tables and tablet servers
  - Assigns tablets to tablet servers
  - Merges tablets
  - Tracks active servers and learns about splits
  - Clients only deal with master to create/delete tables and column family changes
  - Clients get data directly from servers
- All Tables Are Part of One Big System
  - Root table points to metadata tables
    - » Never splits => always three levels of tablets
  - These point to user tables

Many Tricky Bits

- SSTables work in 64k blocks
  - Pro: caching a block avoid seeks for reads with locality
  - Con: small random reads have high overhead and waste memory
    - » Solutions?
- Compression: compress 64k blocks
  - Big enough for some gain
  - Encoding based on many blocks => better than gzip
  - Second compression within a block
- Each server handles many tablets
  - Merges logs into one giant log
    - » Pro: fast and sequential
    - » Con: complex recovery
      - Recover tablets independently, but their logs are mixed…
        - Solution in paper: sort the log first, then recover...
      - Long time source of bugs
  - Could we keep the logs separate?

Lessons learned

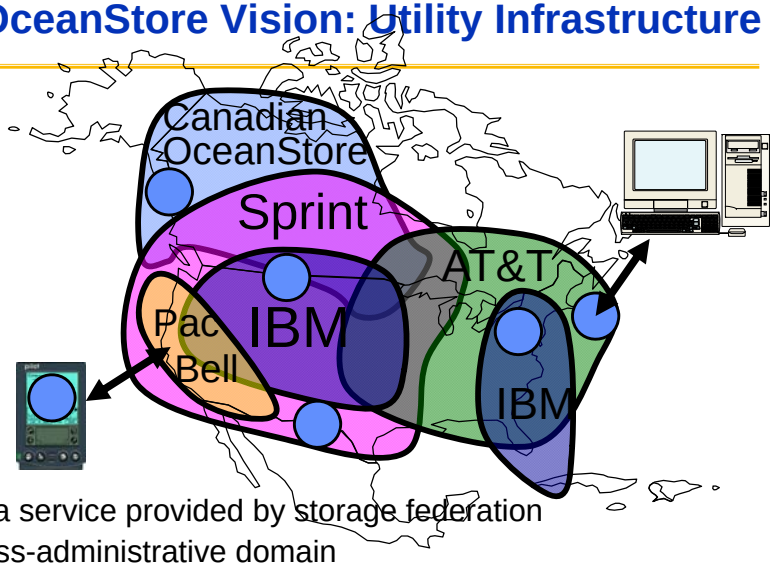
- Interesting point – only implement some of the requirements, since the last is probably not needed
- Many types of failure possible
- Big systems need proper systems-level monitoring
  - Detailed RPC trace of specific requests
  - Active monitoring of all servers
- Value simple designs

Is this a good paper?

- What were the authors' goals?
- What about the evaluation/metrics?
- Did they convince you that this was a good system/approach?
- Were there any red-flags?
- What mistakes did they make?
- Does the system/approach meet the "Test of Time" challenge?
- How would you review this paper today?

BREAK

OceanStore Vision: Utility Infrastructure



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

What are the advantages of a utility?

- For Clients:
  - Outsourcing of Responsibility
    - » Someone else worries about quality of service
  - Better Reliability
    - » Utility can muster greater resources toward durability
    - » System not disabled by local outages
    - » Utility can focus resources (manpower) at security-vulnerable aspects of system
  - Better data mobility
    - » Starting with secure network model⇒sharing
- For Utility Provider:
  - Economies of scale
    - » Dynamically redistribute resources between clients
    - » Focused manpower can serve many clients simultaneously

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 100s of years):
  - Geographic distribution of information
  - New servers added/Old servers removed
  - Continuous Repair ⇒ Data survives for long term

OceanStore Assumptions

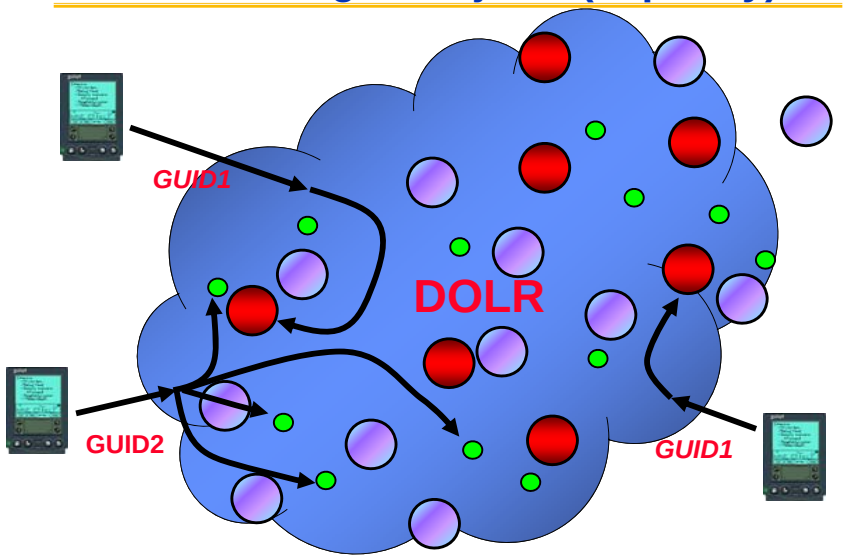
Peer-to-peer

- 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*

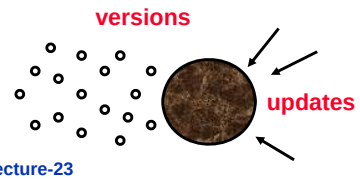
Recall: Routing to Objects (Tapestry)



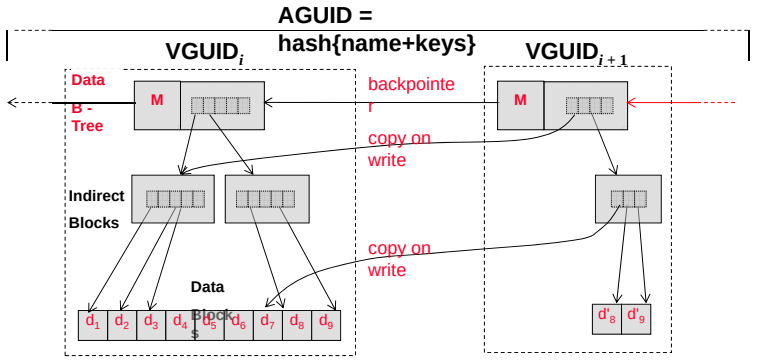
OceanStore Data Model

- 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

Comet Analogy



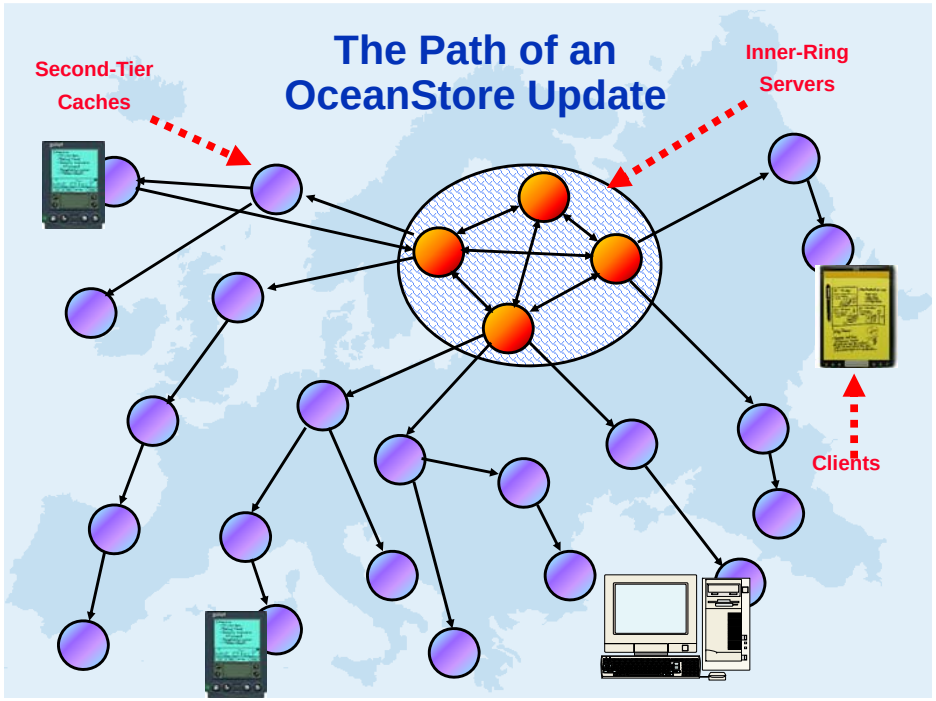
Self-Verifying Objects



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

Two Types of OceanStore Data

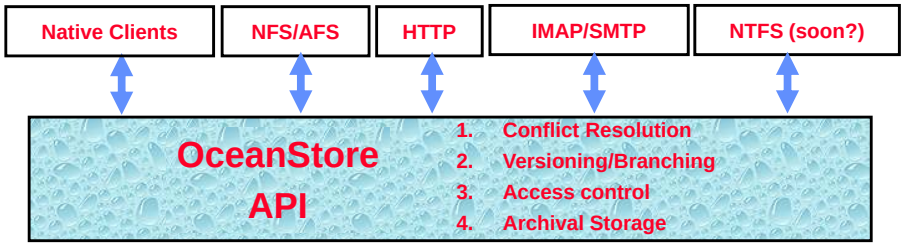
- *Active Data:* "Floating Replicas"
  - Per object virtual server
  - Interaction with other replicas for consistency
  - May appear and disappear like bubbles
- *Archival Data:* OceanStore's Stable Store
  - m-of-n coding: Like hologram
    - » Data coded into *n* fragments, any *m* of which are sufficient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



Byzantine Agreement

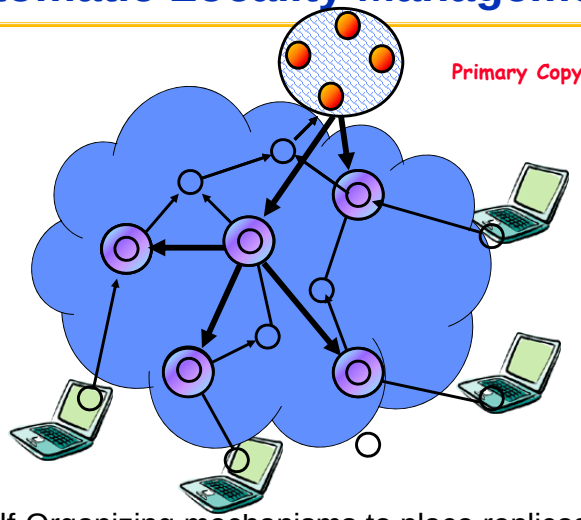
- Guarantees all non-faulty replicas agree
  - Given $N = 3f + 1$ replicas, up to f may be faulty/corrupt
- Expensive
  - Requires $O(N^2)$ communication
- Combine with primary-copy replication
  - Small number participate in Byzantine agreement
  - Multicast results of decisions to remainder
- Threshold Signatures
  - Need at least f signature shares to generate a complete signature

OceanStore API: Universal Conflict Resolution



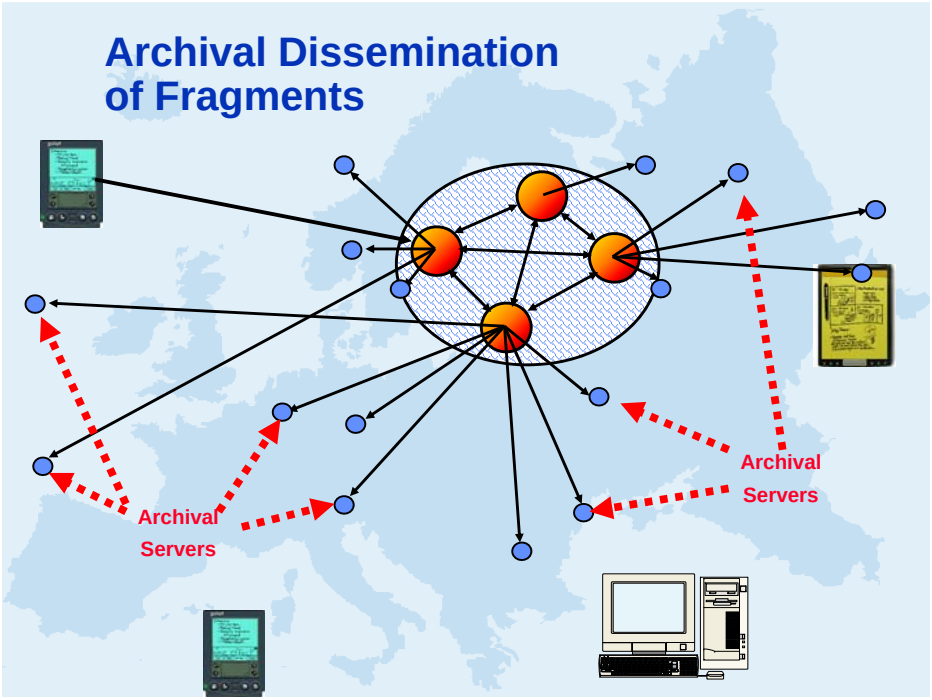
- Consistency is form of optimistic concurrency
  - Updates contain *predicate-action* pairs
  - Each predicate tried in turn:
    - » If none match, the update is *aborted*
    - » Otherwise, action of first true predicate is *applied*
- Role of Responsible Party (RP):
  - Updates submitted to RP which chooses total order

Peer-to-Peer Caching: Automatic Locality Management

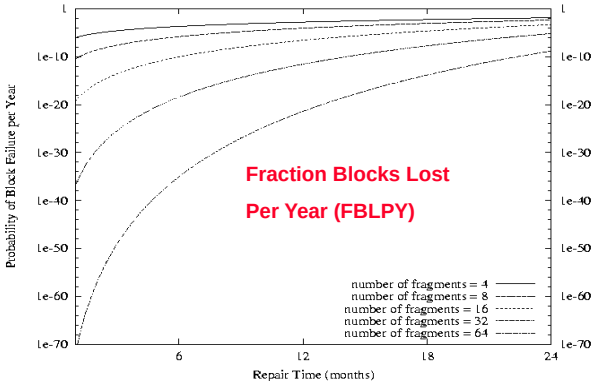


- Self-Organizing mechanisms to place replicas
- Automatic Construction of Update Multicast

Archival Dissemination of Fragments

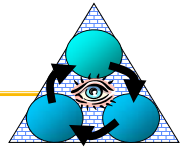


Aside: Why erasure coding? High Durability/overhead ratio!



- 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

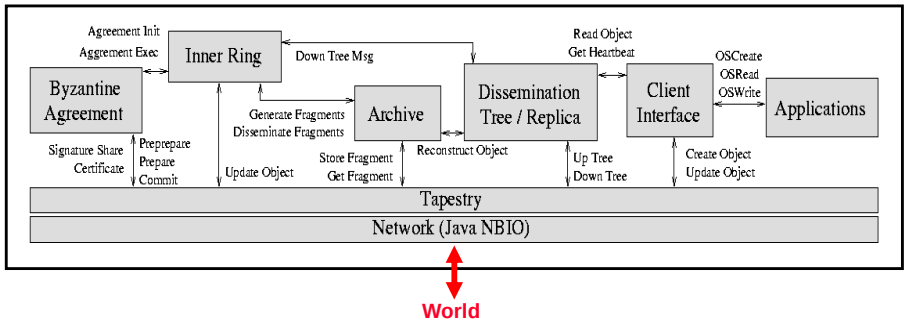
Differing Degrees of Responsibility

- 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!

OceanStore Prototype (Pond)

- All major subsystems operational
  - Self-organizing Tapestry base
  - Primary replicas use Byzantine agreement
  - Secondary replicas self-organize into multicast tree
  - Erasure-coding archive
  - Application interfaces: NFS, IMAP/SMTP, HTTP
- 280K lines of Java (J2SE v1.3)
  - JNI libraries for cryptography, erasure coding
- PlanetLab Deployment (FAST 2003, “Pond” paper)
  - 220 machines at 100 sites in North America, Europe, Australia, Asia, etc.
  - 1.26Ghz PIII (1GB RAM), 1.8Ghz PIV (2GB RAM)
  - OceanStore code running with 1000 virtual-node emulations

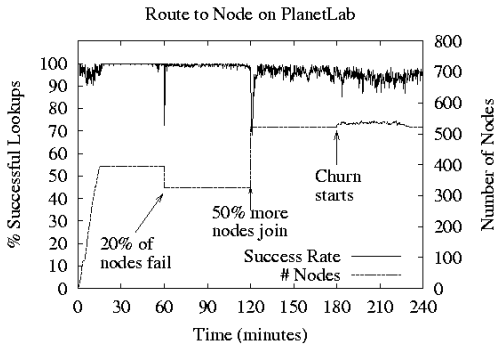
Event-Driven Architecture



- Data-flow style
  - Arrows Indicate flow of messages
- Potential to exploit small multiprocessors at each physical node

Problem #1: DOLR is Great Enabler— but only if it is stable

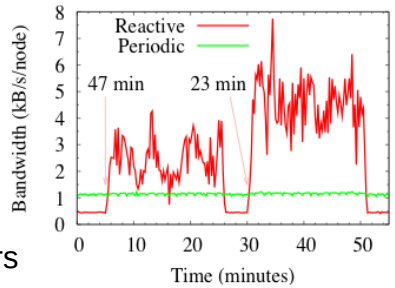
- Had Reasonable Stability:
  - In simulation
  - Or with small error rate
- But trouble in wide area:
  - Nodes might be lost and never reintegrate
  - Routing state might become stale or be lost
- Why?
  - Complexity of algorithms
  - Wrong design paradigm: strict rather than loose state
  - Immediate repair of faults



- Ultimately, Tapestry Routing Framework succumbed to:
  - Creeping Featurism (designed by several people)
  - Fragilility under churn
  - Code Bloat

Answer: Bamboo!

- Simple, Stable, Targeting Failure
- Rethinking of design of Tapestry:
  - Separation of correctness from performance
  - Periodic recovery instead of reactive recovery
  - Network understanding (e.g. timeout calculation)
  - Simpler Node Integration (smaller amount of state)
- Extensive testing under Churn and partition
- Bamboo is so stable that it is part of the OpenHash public DHT infrastructure.
- In wide use by many researchers



Problem #2: Pond Write Latency

- Byzantine algorithm adapted from Castro & Liskov
  - Gives fault tolerance, security against compromise
  - Fast version uses symmetric cryptography
- Pond uses threshold signatures instead
  - Signature proves that *f* +1 primary replicas agreed
  - Can be shared among secondary replicas
  - Can also change primaries w/o changing public key
- Big plus for maintenance costs
  - Results good for all time once signed
  - Replace faulty/compromised servers transparently

Closer Look: Write Cost

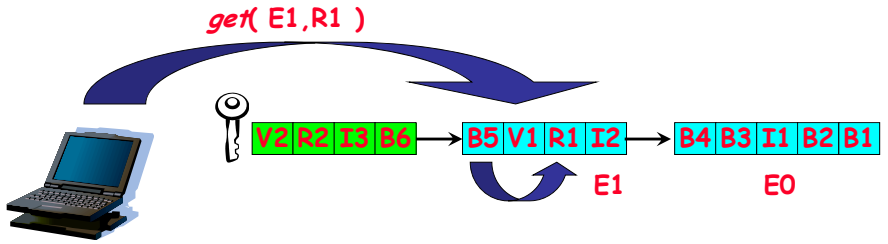
- Small writes
  - Signature dominates
  - Threshold sigs. slow!
  - Takes 70+ ms to sign
  - Compare to 5 ms for regular sigs.
- Large writes
  - Encoding dominates
  - Archive cost per byte
  - Signature cost per write
- Answer: Reduction in overheads
  - More Powerful Hardware at Core
  - Cryptographic Hardware
    - » Would greatly reduce write cost
    - » Possible use of ECC or other signature method
  - Offloading of Archival Encoding

Phase	4 kB write	2 MB write
Validate	0.3	0.4
Serialize	6.1	26.6
Apply	1.5	113.0
Archive	4.5	566.9
Sign Result	77.8	75.8

(times in milliseconds)

Problem #3: Efficiency

- No resource aggregation
  - Small blocks spread widely
  - Every block of every file on different set of servers
  - Not uniquely OceanStore issue!
- Answer: Two-Level Naming
  - Place data in larger chunks ('extents')
  - Individual access of blocks by name within extents



  - Bonus: Secure Log good interface for secure archive
  - Antiquity: New Prototype for archival storage
  - *Similarity to SSTable use in BigTable?*

Problem #4: Complexity

- Several of the mechanisms were complex
  - Ideas were simple, but implementation was complex
  - Data format combination of live and archival features
  - Byzantine Agreement hard to get right
- Ideal layering not obvious at beginning of project:
  - Many Applications Features placed into Tapestry
  - Components not autonomous, i.e. able to be tied in at any moment and restored at any moment
- Top-down design lost during thinking and experimentation
- Everywhere: reactive recovery of state
  - Original Philosophy: Get it *right* once, then repair
  - Much Better: keep working toward ideal (but assume never make it)

Other Issues/Ongoing Work at Time:

- Archival Repair Expensive if done incorrectly:
  - Small blocks consume excessive storage and network bandwidth
  - Transient failures consume unnecessary repair bandwidth
  - Solutions: collect blocks into *extents* and use *threshold* repair
- Resource Management Issues
  - Denial of Service/Over Utilization of Storage serious threat
  - Solution: Exciting new work on fair allocation
- Inner Ring provides incomplete solution:
  - Complexity with Byzantine agreement algorithm is a problem
  - Working on better Distributed key generation
  - Better Access control + secure hardware + simpler Byzantine Algorithm?
- Handling of low-bandwidth links and Partial Disconnection
  - Improved efficiency of data storage
  - Scheduling of links
  - Resources are *never* unbounded
- Better Replica placement through game theory?

Bamboo ⇒ OpenDHT

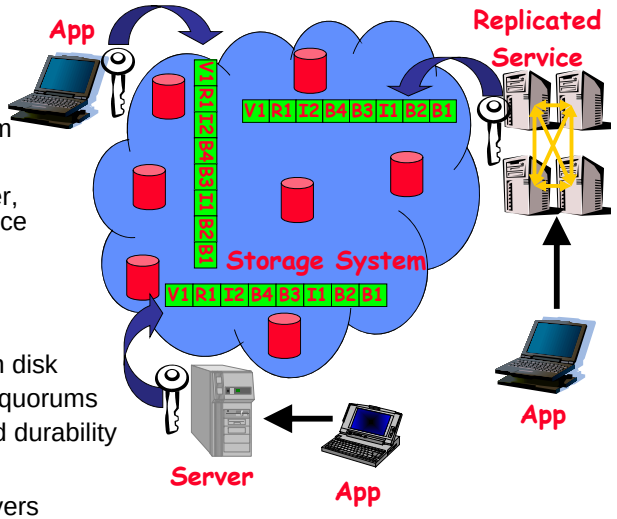
- PL deployment running for several months
- Put/get via RPC over TCP

OceanStore Archive ⇒ Antiquity

- Secure Log:
  - Can only modify at one point – log head.
    - » Makes consistency easier
  - Self-verifying
    - » Every entry securely points to previous forming *Merkle* chain
    - » Prevents substitution attacks
  - Random read access – can still read efficiently
- Simple and secure primitive for storage
  - Log identified by cryptographic key pair
  - Only owner of private key can modify log
  - Thin interface, only *append*()
- Amenable to secure, durable implementation
  - Byzantine quorum of storage servers
    - » Can survive failures at O(n) cost instead of O(n^2) cost
  - Efficiency through aggregation
    - » Use of Extents and Two-Level naming

Antiquity Architecture: Universal Secure Middleware

- Data Source
  - Creator of data
- Client
  - Direct user of system
    - » "Middleware"
    - » End-user, Server, Replicated service
  - *append*()'s to log
  - Signs requests
- Storage Servers
  - Store log replicas on disk
  - Dynamic Byzantine quorums
    - » Consistency and durability
- Administrator
  - Selects storage servers
- Prototype operational on PlanetLab



Is the Pond paper a good paper?

- What were the authors' goals?
- What about the evaluation/metrics?
- Did they convince you that this was a good system/approach?
- Were there any red-flags?
- What mistakes did they make?
- Does the system/approach meet the "Test of Time" challenge?
- How would you review this paper today?