

Operating Systems: Distributed File Systems

Topic 5

Ubiquitous Computing means Ubiquitous Data

So what do users *REALLY* care about?

- Their Data!

Memory is highly distributed

Users are mobile

So where should the data be?

- If storage and computation is cheap - owner computes
- If communication is cheap - use remote compute servers (Grid Based Computing?)

Parallel File Systems have a few highly shared large files

Distributed File Systems have many seldom shared small files

What if mobile user loses connection with owner of the data?

How can we find information stored on remote machines?

Power Awareness still an issue!

Intro to DFS

Assumption - Users have many small seldom shared files

- Use Replication across servers
- Clients cache local copies
- What happens if a client writes to a file?

How to handle naming and directory structures

- NFS Approach - Mount remote directories on local file systems
- AFS/CODA Approach - Impose a globally consistent naming scheme

Use of distributed local caches causes *semantic* challenges

- Cache Coherence - All users should see the same value
- Cache Consistency - When remote writes should become visible

Satyanaryanan's CODA - A DFS with Mobility Support

Coda [3] Based on CMU's Andrew File System (AFS)

Goals include

- Availability
- Scalability
- Use COTS hardware
- Transparency

Disconnection makes it harder (used to lock up AFS)

- Voluntary vs. Involuntary disconnects quite different
- So replicate whole files (since they are small) across servers
- Let clients hoard local copies in the event of disconnection
- Support reintegration of changes when reconnected
 - ▷ *Conflicts tend to be rare due to seldom shared nature of files*
- For scalability and efficiency
 - ▷ *Push the work back on the clients (there are more of them!)*
 - ▷ *Get guidance from the user when needed*

CODA Concepts

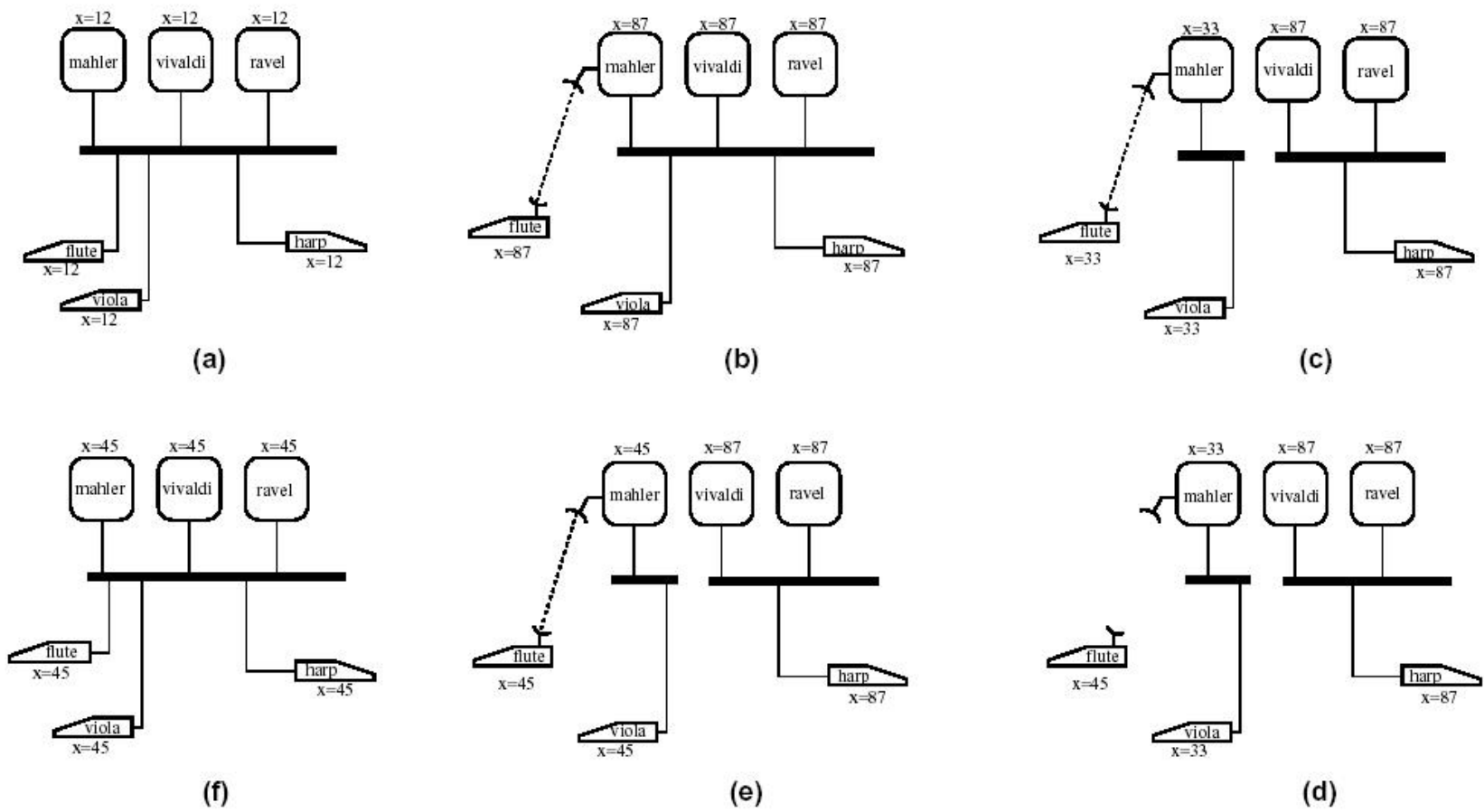
Some Replication Notation:

- A *Volume* is a directory subtree
- A *Volume Storage Group (VSG)* is the set of replication sites for a volume
- An *Available Volume Storage Group (AVSG)* is the subset the the contents of a VSG that a client can currently access.

VENUS is the cache manager

- Uses Callbacks (event driven interfaces)
- Returns the newest version in its AVSG in response to an open request

Disconnected Operation in Coda, An Example



CODA Replication

Which replica represents the true value?

- First Class Replicas (on Servers) are preferred, better quality
- Second Class Replicas (on Clients) are cached copies, faster to access

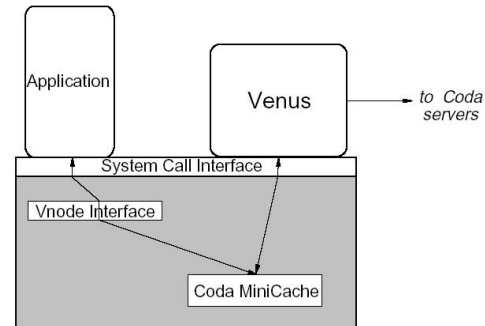
Want speed of second class replicas with quality of first class replicas

- Server Replication Supports Quality
- Disconnected operation trades-off availability for quality

Pessimistic vs. Optimistic Replica Control

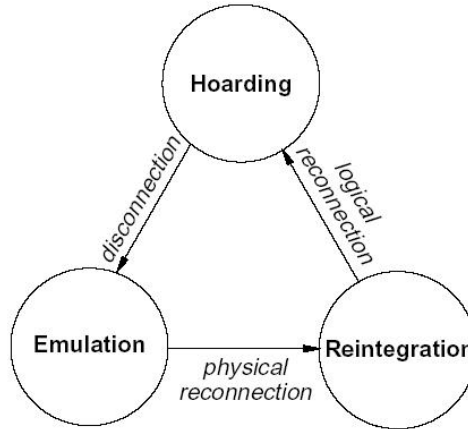
- Pessimistic replica control requires shared/exclusive control of accessed entities prior to disconnection
- Optimistic replica control permits speculative execution with fixup steps when guessing wrong
 - ▶ *Hence, conflict detection tools are needed*

CODA Client Architecture



Application accesses file system using normal Unix system calls
File system VNODE layer, instrumented to support CODA
Minicache used in kernel to satisfy high volume requests
Minicache misses satisfied by Venus
Venus may instigate changes in Minicache state (e.g. due to server reconnect or break)

Venus States



Venus operates in 3 states

- Hoarding - caching copies of files during connected operation
- Emulation = operating in disconnected mode (on cache only)
- Reintegration - reconciling local cache with AVSG upon reconnect

Venus Hoarding

Want to cache file in anticipation of disconnection

So which files to hoard?

- Want *equilibrium* where cache meets user expectations
 - ▷ *Has files user wants cached*
 - ▷ *Doesn't have (an excessive amount of) unused files*
- Use a prioritized algorithm and hints
 - ▷ *Explicit Hints - File names (paths) and attributes tracked using Hoard DataBase (HDB)*
 - ▷ *Implicit Hints - Use recent user file access patterns to generate implicit hints*
- Use *Hoarde Walking*
 - ▷ *Periodically traverse cache*
 - ▷ *Remove low priority entries*
 - ▷ *Heirarchical*

Consistency and Efficiency

Unix consistency semantics - writes immediately visible to readers

- Intuitive
- Efficient for shared memory/single processor systems
- However, expensive and complex for distributed systems

Session based consistency - A relaxed solution

- Instead of a Unix like model uses an approximation session level semantics
- Spatial and Temporal locality assumed - following updates to a file are most likely on the same node
- trades off accuracy for performance

Venus Emulation 1 of 2

Coda trusts only servers, not clients

Venus behaves a pseudo-server during emulation

- Does many server actions during periods of disconnection
- Generates temporary file ids for new objects pending reintegration, when permanent tfids are created
- To warrant trust, Venus must be faithful in emulation

What if a cache miss happens during emulation?

- Default is to return an error
- Other approach is to block process until cache miss can be serviced

Planning for reintegration means logging transactions

- Write/update transactions are preserved in a replay log for each volume
- Preserves system call arguments, version and state of all referenced objects
- All writes between open and close are aggregated into a single large write transaction.

- Inverse and overwrites are detected and can eliminate or reduce transaction count/size

Venus Emulation 2 of 2

Persistence of data and meta data during disconnection is important

- If a user shuts his laptop off before reconnecting what to do?
- Use Recoverable virtual memory to store local non-nested transactions
- Instead of using write-through, write back with periodic flushing is used.

Resource Management is critical

- Could run out of non-volatile storage
 - ▷ *File cache*
 - ▷ *RVM (transaction logs)*
- Try compression, selective backout of updates and removable media

Venus Reintegration

When reconnected must reconcile logged transactions by replaying transaction log against AVSG

- Get permanent fids for each temporary fid in the replay log
 - ▷ *Coda by default gets a small number of fids when connected, sometimes this step can be avoided*
- The replay log is shipped to the AVSG and is reconciled in parallel on each member server aborting if an error is detected, using the following replay algorithm:
 - ▷ *The log is parsed, a transaction is started and objects in the log are locked.*
 - ▷ *Each operation in the log is validated and executed (conflict detection and integrity tests)*
 - ▷ *Performing data transfers (called back-fetching)*
 - ▷ *Commits transactions and releases locks*

Upon success, can reclaim resources, on failure what to do?

- Save the log using an extension of Unix tar format
- Notify the user
- Offer tools for selective replay/log browsing

Venus Reintegration Conflict Handling (1 of 2)

Disconnected servers may have conflicts

- With servers and connected clients
- Or with other disconnected clients

What is a conflict, what kinds can occur?

- Corresponds to a hazard (hardware) or violation of Bernstein's conditions

$$\text{Conflict Set} = (R(P_i) \cap W(P_j)) \cup (R(P_j) \cap W(P_i)) \cup (w(P_i) \cap W(P_j))$$

- Only Write/Write conflicts matter
 - ▶ *Since Unix has atomic read/write system call semantics*

Venus Reintegration Conflict Handling (2 of 2)

How can conflicts be detected?

- Each replica has a *storeid* indicating the last update done to it.
- During log replay stage 2 (Validation and Execution) the storeid is checked
 - ▷ *If the storeids match the reintegration succeeds*
 - ▷ *If the storeids do not match*
 - ▷ *If the object type is a file, reintegration fails*
 - ▷ *If the object type is a directory, the names of the updated entries are checked, and if they were not updated recently, then the reintegration succeeds, otherwise it fails.*

In practice, reintegration was fast (on the order of 10s of seconds for 2-3 MB of data).

In practice conflicts are rare (due to seldom shared nature of files)

- 0.75 % of files write accessed by multiple users per day (including system files)
- 0.4 % of files write accessed by multiple users per week (not including system files)

Bayou: Support for data sharing among mobile users

Demers et al. [2] at Xerox PARC

A highly replicated distributed data management system

Bayou assumes wireless (so connectivity may be sporadic)

Design Motivations include

- Users want to share data
 - ▷ *Calendars, appointments, news, work materials, etc.*
- Mobile clients subject to voluntary and involuntary disconnection
- Must support reading and writing of data
- Must “feel” like a centralized highly available database service

Bayou Design Goals (1 of 4)

Each Design Goal Motivated an Approach

- Support for mobile computers with limited resources - Flexible Client/Server Architecture
 - ▷ *Anticipated many PDA style clients*
 - ▷ *Uses Servers to store data - may be portable devices*
 - ▷ *Clients access data, interface with users*
 - ▷ *Bayou has less distinct client/server model than Coda*
- High Availability for Reads and Writes - Read-any, Write-any weakly consistent replication
 - ▷ *Allow client to read or write any copy of a database*
 - ▷ *Atomically updating all available copies is inefficient and can't handle disconnection*
 - ▷ *Quorum based schemes can't handle disconnection for small groups/individuals*
 - ▷ *Pessimistic locking too expensive and limit availability*
- Reach Eventual Consistency with minimal assumptions about data communication characteristics - Use peer-to-peer anti-entropy to propagate updates
 - ▷ *Anti-entropy protocols a form of distributed reconciliation*
 - ▷ *Servers must get all writes and writes must be ordered*

- ▷ *In the absence of updates they converge to a new state*
- ▷ *Periodically each server randomly picks another server to exchange writes with.*
- ▷ *After the exchange the server pair has matching databases*

Bayou Design Goals (2 of 4)

- System Support for detecting update conflicts - Dependency checks on every write
 - ▷ *System level support techniques for write-write using update vectors and read-write using read-sets don't address application semantics*
 - ▷ *e.g. in a PIM a person is scheduled for conflicting appointments*
 - ▷ *So, define write-write conflicts as when the state of the database differs in an application-relevant way from what the write operation expects.*
 - ▷ *Each write includes the data written and a dependency set of application specified queries with their expected results.*
- Application-specific update conflict resolution - Merge procedures passed with each write for automated conflict resolution
 - ▷ *Let the application developer tell the O/S how to resolve conflicts*
 - ▷ *The Merge Procedure (mergeproc) is invoked when write conflict is detected*

Bayou Design Goals (3 of 4)

- Commit data to a stable value asap - Include a primary server for committing data and setting the commit order
 - ▷ *Requiring a server to get enough information to ensure that conflicting writes exist/will be accepted is inefficient*
 - ▷ *Bayou supports explicit write commits*
 - ▷ *Uncommitted writes are called tentative*
 - ▷ *Noncommitted writes cannot be applied before a committed write*
 - ▷ *Primary server does commits*
 - ▷ *Secondary servers tentatively accept writes and relay them to the primary server*
 - ▷ *Primary server notifies secondary servers of commits*
- Let disconnected clients/groups see their own updates - Permit clients to read tentative data
 - ▷ *Clients can expect tentative writes will be committed if possible*
 - ▷ *Support 2 views - Showing only committed data and showing tentative updates*
 - ▷ *Secondary servers agree on tentative update order after data exchange*

Bayou Design Goals (4 of 4)

- Give a client a replica consistent with its own actions - Session Guarantees
 - ▷ *Read Your writes*
 - ▷ *Monotonic Reads - Reads reflect monotonically nondecreasing set of updates (by timestamp)*
 - ▷ *Writes Follow Reads (on which they depend)*
 - ▷ *Monotonic Writes - Writes are propagated after writes that precede them.*
- Permit Applications to choose consistency/availability trade-off
 - ▷ *Support application selectable session guarantees*
 - ▷ *Allow choice of committed or tentative data*
 - ▷ *Support age parameters on reads*
- Give users control over placement and use of databases - Fluid replication
 - ▷ *Fluid replication allows copies of data to flow around in the system*
 - ▷ *Clients can specify number and location of replicas*
 - ▷ *Both primary servers and secondary servers can be changed*

Pond: The Oceanstore Prototype

Oceanstore's goal is to modernize DFS, goals include:

- Persistent storage
- Incremental Scalability
- Secure Sharing
- Universal Availability
- Long Term Durability
- Cooperatively operate at the internet scale

Employs a two tier design

- Top tier is powerful well connected hosts
 - ▷ *Serialize Changes*
 - ▷ *Archive Results*
- Lower tier: work stations
 - ▷ *Where users actually perform updates*
 - ▷ *Provide storage resources to the system*

Oceanstore Architectural Features

The unit of storage is the *data object*

- Data objects management should support following requirements
 - ▷ *Universal accessibility of information (access anywhere, any time)*
 - ▷ *Find the right balance between sharing and privacy of information*
 - ▷ *Have an easily understood and usable consistency model*
 - ▷ *Guarantee data integrity.*
- Flexible, can support e-mail or a Unix File system

Design assumptions are

- Infrastructure untrusted except in aggregation
- Infrastructure is constantly changing
 - ▷ *Variable congestion*
 - ▷ *Changing connectivity*
 - ▷ *So system must be self organizing and self-repairing*

Challenge: To design a system that:

- Provides an expressive storage interface to users
- While guaranteeing high durability

- Atop an untrusted and constantly changing infrastructure

Oceanstore Data Model

A data object is similar to a traditional file

Data objects are an ordered sequence of read only versions

Versions are persistent

- Simplifies caching and replication
- Permits *time travel* (rollback and version comparison)
- What about Hippocratic databases?

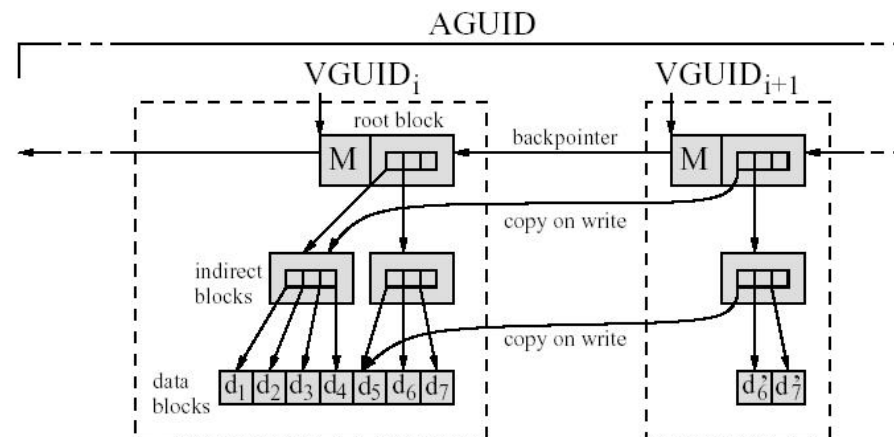
Oceanstore Version Support Internals

Copy on write used to reduce storage overhead

- Files stored in trees with pointers to leafs which contain data
- Versions made by tracking only updated blocks

GUID - Globally-Unique Identifier

- AGUID - Active GUID - handle for a sequence
- BGUID - Block GUID
- VGUID - Version GUID - BGUID of top of structure
- GUID's Hierarchically cryptographically hashed for security reasons



Oceanstore Application-Specific Consistency

Updates add a new head to the version stream of one or more data objects

- Updates are atomic
- Each action guarded by a predicate (much like Bayou)
 - ▷ *Example Actions: Append, replacing data, truncation*
 - ▷ *Example Predicates: checking latest version number*
- Supports application defined consistency semantics.

Oceanstore System Architecture

Changes to a single object must be coordinated via shared resources

Changes to different objects are independent (and can be parallelized)

Virtualization through Tapestry

- Virtual resources are mapped to physical resources
- The mapping is dynamic
- Applications access virtual resources
- Virtual resources have a GUID to get the state information needed to access that resource
- Tapestry supports decentralized object location and routing (DOLR)
 - ▷ *Built on a scalable overlay network (e.g. TCP/IP)*
 - ▷ *Messages sent via Tapestry routed using GUID, not IP address*
 - ▷ *Tapestry is locality aware (picks the nearest instance with high probability)*
 - ▷ *Physical hosts join tapestry by supplying Resource GUIDs to register themselves*

- ▷ *Hosts publish GUIDs of their resources in Tapestry*
- ▷ *Hosts may unpublish or leave the network at any time.*

Oceanstore Replication and Consistency

Replication and Consistency

- Each data object's AGUID mapping to its current state changes over time
- Each object has a primary replica designated to
 - ▷ *Serialize and Apply updates*
 - ▷ *Enforce Access Control*
- a *heartbeat* digital certificate is created to map AGUID to the VGUID of the most recent version.
 - ▷ *a tuple < AGUID, VGUID, Time Stamp, Version Number >*
- To securely identify the heartbeat a client may include a nonce in its request
 - ▷ *A nonce is a special one time message (avoids replay and guessing)*
 - ▷ *Response contains client id and nonce and is signed by the parent*
- Primary replica is implemented on a small inner ring of servers
 - ▷ *Inner ring servers run Byzantine Generals Algorithm to agree on updates and sign result*
 - ▷ *Primary replica is a virtual resource, not bound to a particular server.*
- Metadata and secondary replicas also need to be maintained

Oceanstore Ensuring Archival Integrity

How can we prevent data loss in the event of the failure of a small number of nodes?

- For disks we might use replication (mirroring)
- Replication is expensive (100 % overhead per replica)
- Alternatively we can use an erasure code (a superset of RAID)
 - ▷ *Parity Codes*
 - ▷ *Hamming Codes*
 - ▷ *Pond uses Cauchy Reed-Solomon Codes (beyond lecture scope)*
- Erasure Codes partitions a block into m equal sized fragments
- The m fragments are encoded into $n, n > m$ fragments
- $r = \frac{m}{n} < 1$ is the *rate of encoding*
 - ▷ *Storage overhead increases by a factor of $\frac{1}{r}$.*
 - ▷ *The original object can be reconstructed from any m fragments*
 - ▷ *This is much more fault tolerant than replication (if bad fragments are detected and discarded)*
 - ▷ *Pond uses a Cauchy Reed-Solomon Code with $m = 16, n = 32$.*
- Updates to primary replicas are erasure coded and fragments are distributed across servers.

Oceanstore Data Caching

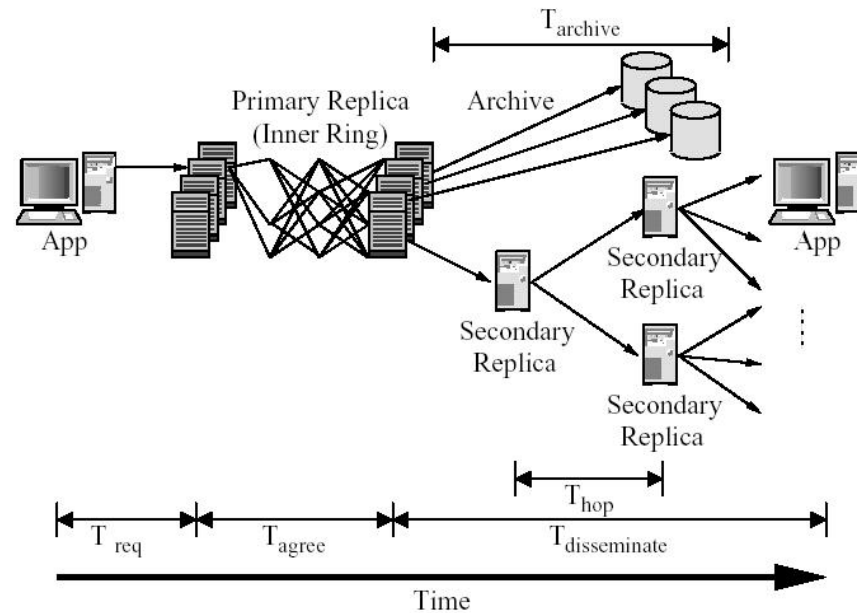
Assembling blocks from erasure code fragments is expensive

- m distinct blocks needed stored on m machines mean m messages

For frequently read blocks, we will use block caching

- When a host requests a block
 - ▷ *If the host does not have the block cached, request fragments and reassemble from m fragments*
 - ▷ *If the host has the block cached, use the local copy*
- Caching host publishes ownership to Tapestry
- Subsequent reads satisfied from the cached block.
 - ▷ *Amortizes the cost of reassembly over all the readers*
- Pond uses a soft state approach (i.e. old state is discarded)
- What if the most recent copy is needed
 - ▷ *Block cache keeps the heartbeat used to construct the block*
 - ▷ *Host needing the block can query Tapestry for just the heartbeat*
 - ▷ *If heartbeats match use the block cache, otherwise reassemble the block*
- Write-Invalidate (Push) based approach uses a *dissemination tree* for each object

Oceanstore Update Path



When a host initiates an update in Oceanstore

- Update propagates from client to target object's primary replica
- The update is serialized with other updates and applied
- The following are done concurrently
 - ▷ *A heartbeat is generated and propagated out to secondary storage and is multicast along with update to the dissemination tree*

- ▶ *A new version is erasure encoded and sent to the archival storage servers*

Oceanstore Archive Security and Fault Tolerance (1 of 3)

Employs a Byzantine Generals approach (Liskov-Castro Based Approach)

- Recall Byzantine failures can model nodes with compromised security.
- If we have $N = 3f + 1$ servers we can tolerate f faults
- Employs signed messages (hence $O(N^2)$ messages need to be sent).
- Hence the inner ring is kept small to reduce overhead.
- Liskov-Castro approach has 2 key management approaches
 - ▷ *Public Key - Permits 3rd party authentication but slow.*
 - ▷ *Symmetric Key (MAC)- Authentication possible only at end points, but several orders of magnitude faster*

Oceanstore Archive Security and Fault Tolerance (2 of 3)

Departures from Castro-Liskov Approach include

- Pond uses a mixed cryptographic approach
 - ▷ *MAC used in the inner ring*
 - ▷ *Public key used elsewhere (supports aggressive replication, since verification can be done without contacting inner ring).*
 - ▷ *Public key signature cost can be amortized over number of copies distributed.*
- Traditional Byzantine Generals Problem Solution tolerate f faults during the life of the system
 - ▷ *For long term usage, that is too restrictive*
 - ▷ *So Liskov-Castro reboot nodes periodically using a Trusted O/S*
 - ▷ *Liskov-Castro Approach assumes hardware support for key management and fixed membership*
 - ▷ *Want to vary inner ring membership AND not alter public keys*
 - ▷ *Use Rabin's Proactive Threshold Signatures - Secret Sharing*
 - ▷ *Partition a key into l (overlapping) shares, with k shares needed to reconstruct the key*
 - ▷ *Pond sets $l = 3f + 1$ and $k = f$.*

- ▷ To change the inner ring membership, the key is repartitioned into a different l shares (having less than k old shares doesn't help).
- ▷ Assuming no more than $k - 1$ faults (by Byzantine Generals Assumption), the inner ring nodes will delete the old keys in favor of the new keys.

Oceanstore Archive Security and Fault Tolerance (3 of 3)

So Who gets to be in the Inner Ring?

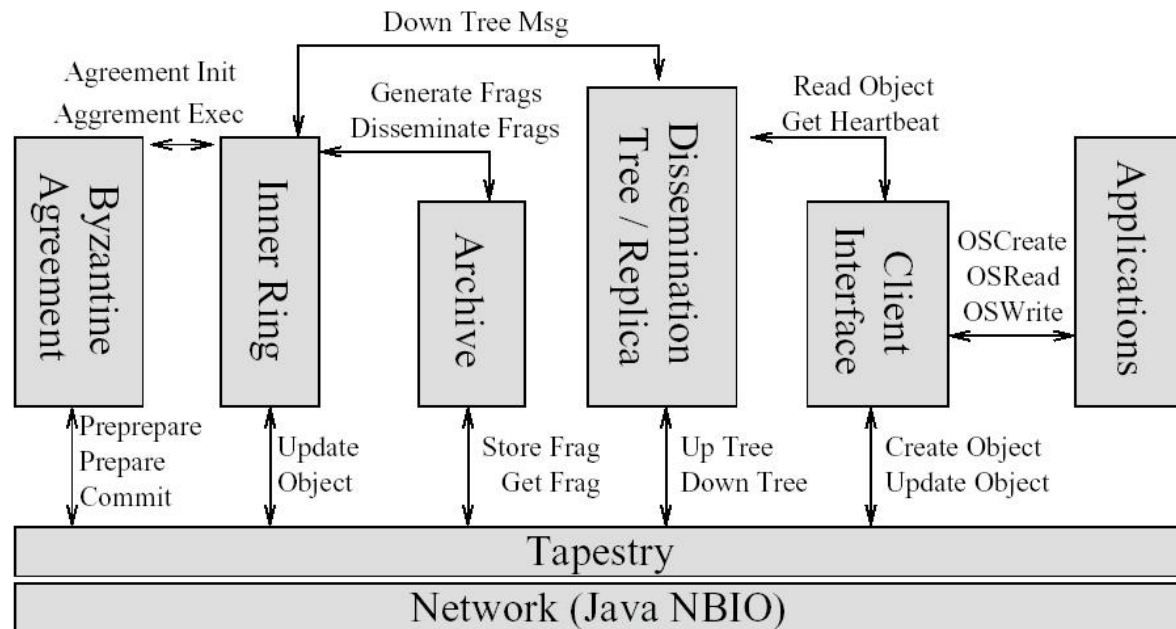
- Pond designates an arbiter called the *responsible party*
 - ▷ *Responsible party a single node, perhaps susceptible to compromise*
 - ▷ *If compromised, the makeup of the inner circle might change*
 - ▷ *But compromising data security requires breaking f nodes in the inner circle, not the responsible party*
 - ▷ *However (not mentioned in the paper) changing inner circle involves recomputing shares (expensive). Could this expose Pond to risk of denial of service attacks?*
- Traditional Byzantine Generals Problem Solution tolerate f faults during the life of the system
 - ▷ *For long term usage, that is too restrictive*
 - ▷ *So Liskov-Castro reboot nodes periodically using a Trusted O/S*
 - ▷ *Liskov-Castro Approach assumes hardware support for key management and fixed membership*
 - ▷ *Want to vary inner ring membership AND not alter public keys*
 - ▷ *Use Rabin's Proactive Threshold Signatures - Secret Sharing*
 - ▷ *Partition a key into l (overlapping) shares, with k shares needed to reconstruct the key*

- ▷ Pond sets $l = 3f + 1$ and $k = f$.
- ▷ To change the inner ring membership, the key is repartitioned into a different l shares (having less than k old shares doesn't help).
- ▷ Assuming no more than $k - 1$ faults (by Byzantine Generals Assumption), the inner ring nodes will delete the old keys in favor of the new keys.

Pond's Software Architecture

Pond is event driven, not threaded

- Event Driven faster under load (as per Culler's SEDA paper)
- Node process a subset of event types (e.g. Only Inner ring does Byzantine Generals)
- Culler's SEDA project was available to them
- Target architecture was Debian Linux for first implementation



Pond's Current Status

Pond is implemented using Java (50 KLOC)

- Strongly typed
- Has Garbage Collection implemented in JVM
 - ▷ *Current JVMs have "Stop the World" collectors (threads halted during collection)*
 - ▷ *This causes large and uncontrollable variations in delay*

The Liskov-Castro Approach not fully implemented

- View changes (like database views?) not implemented
- Checkpointing and Roll back

In practice, the system seemed to have modest increases in delay as wide area nodes were added (e.g. UCB to UCSD)

Barbar's Mobile Computing and Database Survey

Barbará [1] describes a mobile environment as having

- Mobile Units (wireless)
- Fixed Hosts - communicate over wired infrastructure
 - ▷ *Mobile Support Stations - Some fixed hosts have wireless to support mobile devices*

Data communications partitions requires mobiles to be in a cell

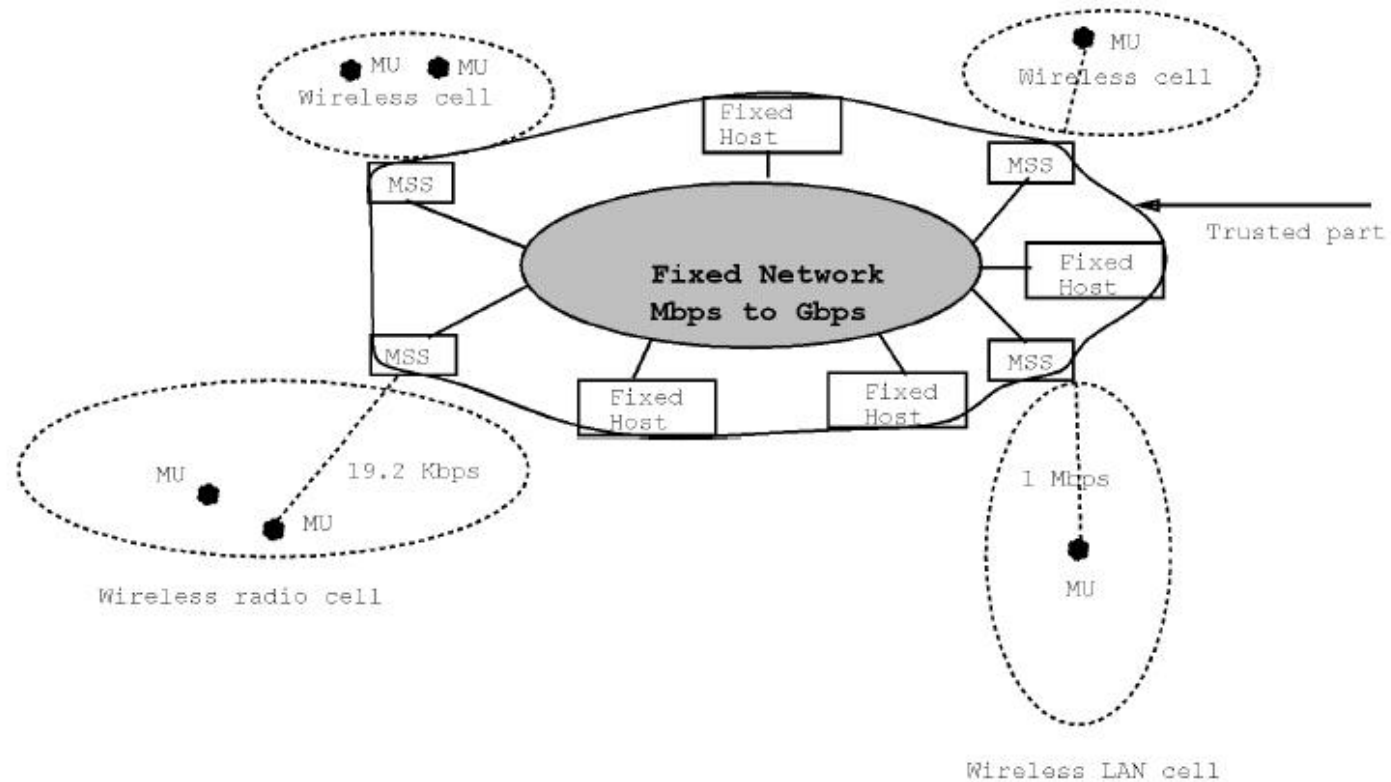
- A cell is a region of radio coverage (e.g. cellular telephones or wireless lan).

How does this differ from traditional distributed systems?

- Asymmetry in communication - Downstream is often faster than upstream
- Frequent Disconnections - Due to power down or roaming
- Power Limitations
- Screen Size

Barbará discusses topics not covered here (e.g. consistency and interfaces)

Barbar's Network Architecture



- MU **Mobile unit** (can be either dumb terminals or workstations)
- MSS **Mobile Support Station (has a wireless interface)**
- Fixed Host **(no wireless interface)**

Barbar - Data Dissemination (1 of 4)

Data Dissemination - Delivery of data from a set of producers to a set of consumers.

- Assumes data channels may be assymmetric (downstream faster than upstream).
- Single senders may have many receivers
- How to initiate updates
 - ▷ *Pull - Client initiated*
 - ▷ *Push - Server initiated (write invalidate) - Preferred, since server knows when data changes*
- The set of receivers may vary over time
 - ▷ *Due to disconnection (voluntary or involuntary).*
- Publish Subscribe model supports this
 - ▷ *E.g. Web radio/video on demand*
 - ▷ *Surely web TV has got to be coming!*
- So When to Push?
 - ▷ *Aperiodic - Only when new data needs to be sent (more bandwidth efficient)*
 - ▷ *Periodic - Handles disconnection better, since upon reconnection, updates come within a known time window.*

Barbar - Data Dissemination (2 of 4)

Acharya et al.'s Balanced Disks - Novel for 1995 time frame

- Uses a multi-level approach supporting nonuniform (e.g. prioritized) bandwidth allocation
- Provides mechanisms to support client-side cache management and prefetching for efficient multi-level broadcast

What are characteristics of a good broadcast program solution?

- Delivers data with approximately fixed interarrival times (avoids burstiness)
- Allocates bandwidth according to access probabilities

Where is the system's knowledge regarding data?

- Servers know values and when updates occur - so use push
- Clients know about data availability and actual needs for access - so use pull
- Acharya uses Integrated Push and Pull (IPP) to handle both cases, for scalability

IPP:

- ▶ *Allows Pull bandwidth to be adjusted (possibly at the expense of Push bandwidth)*

- ▷ *Imposes a Pull Threshold such that cache misses must have a sufficiently large penalty for being unfulfilled before a Pull request is sent to the server.*
- ▷ *Allowing the algorithm to discard incrementally larger parts of the schedule (choose the slowest parts) at the risk of starvation for some Pull requests.*

Barbar - Data Dissemination (2 of 4)

Invalidation reports are used to compress write invalidate traffic

- Trade off accuracy for size
- May cause false negatives on client side
 - ▷ *i.e. Clients may be told that correctly cached data is invalid*
- Introduce *Quasi-copies* - replicas allowed to deviate from the true value in controlled ways

What to do when a disconnected client node (sleeper) reattaches?

- Have client ask about cached items to see if they are still valid.
- However uplink bandwidth suffers if there are many items
- So instead use a hierarchical approach (Wu et al.)
 - ▷ *Aggregate data objects into groups*
 - ▷ *Ask the server about the last time the aggregate was updated*
 - ▷ *If the server gives a newer time, decompose the group and ask about its members*

Barbar - Data Dissemination (3 of 4)

What if a client misses an update (due to failure/congestion?)

- Normally not a big deal in best effort systems, since client will get next update (push).
- In real time systems, this may mean missing deadlines (bad!)
- Bestavros describes a real time approach using 3 strategies
 - ▷ *Flat - Server takes the union of the sets of items needed by the client and periodically updates the client*
 - ▷ *Need to worry about worst case latency on items with strictest deadlines*
 - ▷ *Rate Monotonic - Each object is periodically broadcast at a rate proportional to $\frac{1}{\text{deadline}}$.*
 - ▷ *Optimizes bandwidth, but complex and hard to schedule/manage*
 - ▷ *Slotted rate monotonic - Coalesces groups of data together on the same broadcast disk (a hybrid flat/rate monotonic approach).*
 - ▷ *Reduces scheduling overhead via aggregation*
 - ▷ *Provides better bandwidth utilization than flat structure.*

Efficiency vs. Integrity Tradeoff (Bestavros's Approach)

- Using highly redundant erasure codes gives good integrity
- But redundancy is expensive, as it encodes m blocks into $n, n > m$ blocks.
- However, you only need m blocks to recover the data
- So pick a code with larger n and broadcast $k, m \leq k \leq n$ of the encoded blocks.

Barbar - Data Dissemination (4 of 4)

Using Directory information lets clients sleep (Imielinski et al.)

- When clients wait to receive data, they consume substantial power
- Server can transmit a schedule telling each client when they will get data
 - ▷ *Kind of like statistical multiplexing*
 - ▷ *However, clients must wait for the schedule*
 - ▷ *But the schedule isn't big, so repeat it a few extra times so that clients with smaller windows to receive the schedule can still synchronize.*

Barbar - Location Dependent Queries

Introduced by Imielinski et al.

- e.g. Where is the nearest hospital?
- Optimal Solutions NP-Complete
- Naïve solutions have excessive numbers of messages
- suggest using greedy heuristics based on a decision tree based approach (ID3)
 - ▷ *For multivariate decision problems*
 - ▷ *Decisions yielding the most information gain done nearest the root*

Imielinski and Navas propose merging IP and GPS

Voelker and Bershad's Mobisaic suggests replacing URLs with Dynamic URLs and Active Documents

- Server's response tailored to client's location
- Documents served using HTML (dynamically generated or selected)

Spreitzer and Theimer propose using clients to track their location and a decentralized database

- ▷ *Clients use active badges, IR, etc. to compute location*

Barbar - Open Issues

Prototyping (of data dissemination protocols)

Bandwidth Utilization

Formalizing the Transactional Properties Needed in UbiComp

Optimization of Location Dependent Querying

Data visualization

Bibliography

References

- [1] Daniel Barbara. Mobile computing and databases - a survey. *Knowledge and Data Engineering*, 11(1):108--117, 1999.
- [2] A. J. Demers, K. Petersen, M. J. Spreitzer, D. B. Terry, M. M. Theimer, and B. B. Welch. The bayou architecture: Support for data sharing among mobile users. In *Proceedings IEEE Workshop on Mobile Computing Systems & Applications*, pages 2--7, Santa Cruz, California, 8-9 1994.
- [3] J. J. Kistler and M. Satyanarayanan. Disconnected operation in the coda file system. In *Thirteenth ACM Symposium on Operating Systems Principles*, volume 25, pages 213--225, Asilomar Conference Center, Pacific Grove, U.S., 1991. ACM Press.