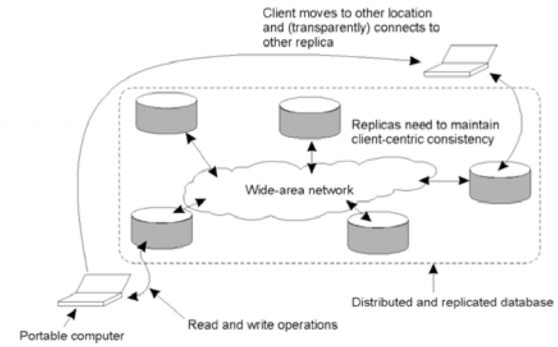
CIS 505: Software Systems Lecture Note on Consistency and Replication (2)

Instructor: Insup Lee
Department of Computer and Information Science
University of Pennsylvania

CIS 505, Spring 2007

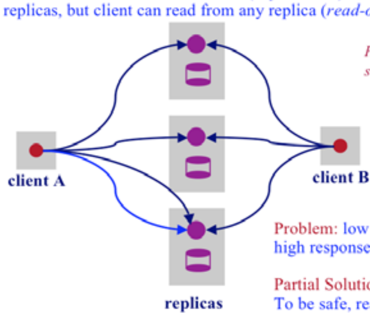
Client-Centric View



- The principle of a mobile user accessing different replicas of a distributed database.

Synchronous Replication

Basic scheme: connect each client (or *front-end*) with every replica: writes go to all replicas, but client can read from any replica (*read-one-write-all replication*).



How to ensure that each replica sees updates in the "right" order?

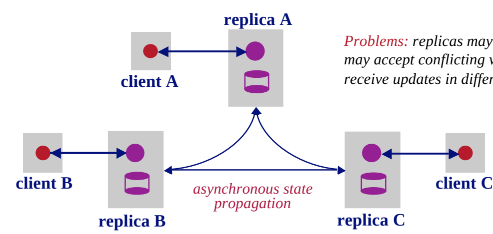
Problem: low concurrency, low availability, and high response times.

Partial Solution: Allow writes to any *N* replicas. To be safe, reads must also request data from the set of replicas.

Asynchronous Replication

Idea: build available/scalable information services with *read-any-write-any* replication and a weak consistency model.

- no denial of service during transient network partitions
- supports massive replication without massive overhead
- "ideal for the Internet and mobile computing" [Golding92]

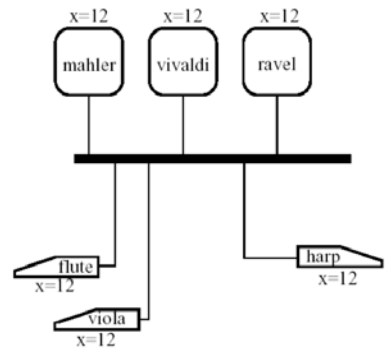


Problems: replicas may be out of date, may accept conflicting writes, and may receive updates in different orders.

Disconnected Operation

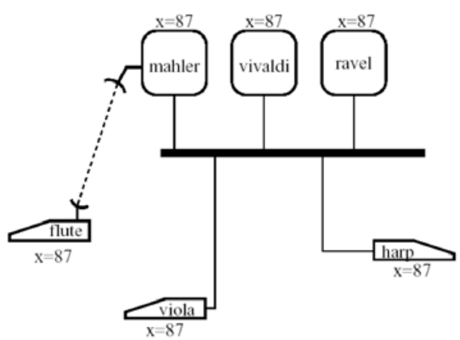
- Continue critical work when that repository is inaccessible.
- Key idea: caching data.
  - o Performance
  - o Availability
- Server Replication

An Example



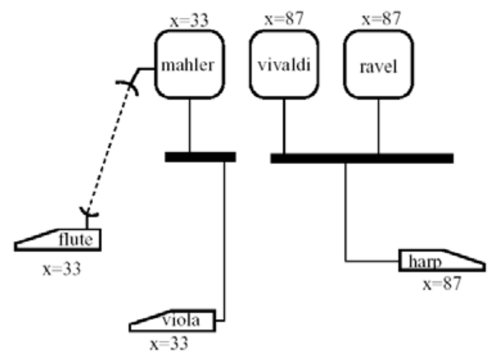
(a)

An Example



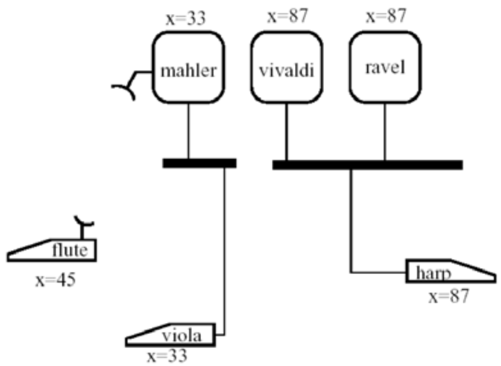
(b)

An Example



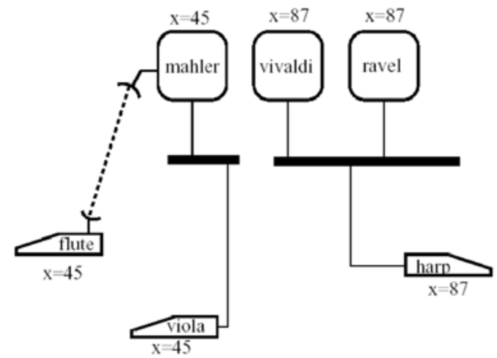
(c)

An Example



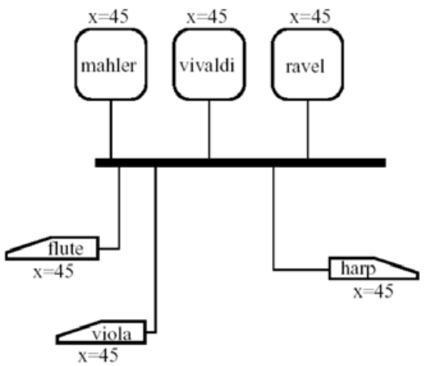
(d)

An Example



(e)

An Example



(f)

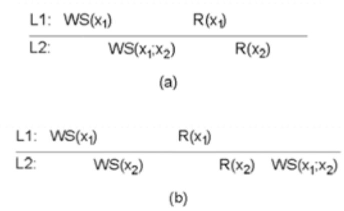
Four notions of Client-centric consistency

- Monotonic-read consistency
  - if a process reads x, any future reads on x by the process will returns the same or a more recent value
- Monotonic-write consistency
  - A write by a process on x is completed before any future write operations on x by the same process
- Read your write
  - A write by a process on x will be seen by a future read operation on x by the same process
- Writes follow reads
  - A write by a process on x after a read on x takes place on the same or more recent value of x that was read

Notation

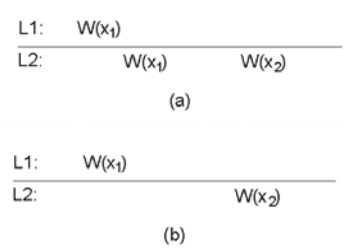
- Let $X_i[t]$ denote the version of data x at local copy L_i at time t.
- Version $X_i[t]$ is the result of a series of write operations at L_i since initialization.
- Use $WS(X_i[t])$ to denote this set of the series of writes at L_i.
- If operations in $WS(X_i[t])$ has also been performed at local copy L_j at a later time t_2, we write $WS(X_i[t_1]; X_J[t_2])$.
- Omit t if timing is clear.

Monotonic Reads



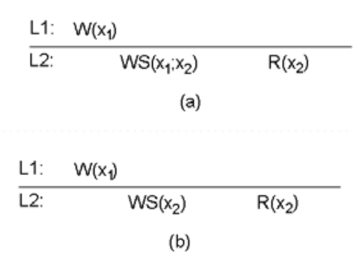
- Def: if a process reads x, any future reads on x by the process will returns the same or a more recent value
- Fig: The read operations performed by a single process *P* at two different local copies of the same data store.
  a) A monotonic-read consistent data store
  b) A data store that does not provide monotonic reads.
- Example: reading mail from different places

Monotonic Writes



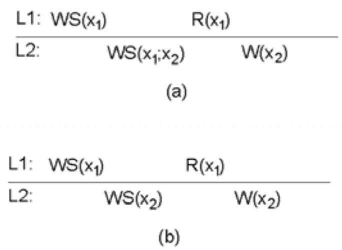
- Def: A write by a process on x is completed before any future write operations on x by the same process
- Fig: The write operations performed by a single process *P* at two different local copies of the same data store
  a) A monotonic-write consistent data store.
  b) A data store that does not provide monotonic-write consistency
- Update to part of the library

Read Your Writes



- Def: A write by a process on x will be seen by a future read operation on x by the same process
- Fig:
  a) A data store that provides read-your-writes consistency.
  b) A data store that does not.
- Example: update on web that is locally cached, update on password file

Writes Follow Reads

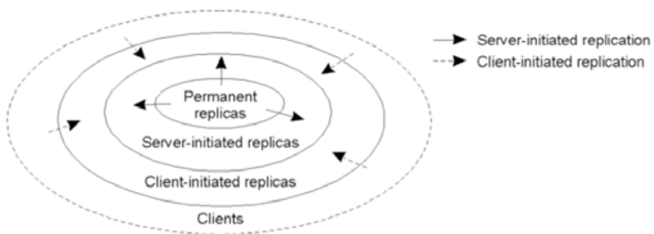


- Def: A write by a process on x after a read on x takes place on the same or more recent value of x that was read
- Fig:
  a) A writes-follow-reads consistent data store
  b) A data store that does not provide writes-follow-reads consistency
- Example: reading netnews and posting of a reaction

Implementation

- Each operation is assigned a unique global id
- For each client, keep two sets of write ids:
  - Read set: write ids relevant for reads by the client
  - Write set: write ids of writes by the client
- For monotonic-read consistency, use the read set
- For monotonic-write consistency, use the write set
- For read-your-write consistency, use both
- For writes-follow-reads consistency,..

Replica Placement



- The logical organization of different kinds of copies of a data store into three concentric rings.

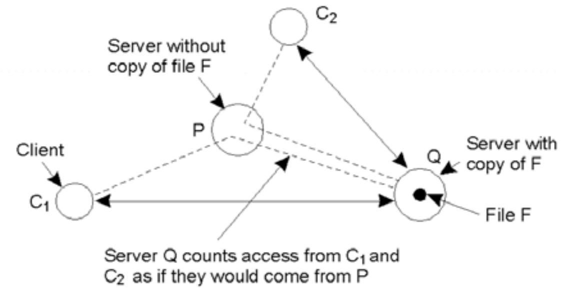
Permanent Replicas

- Two approaches for distributed date stores, like web sites
  1. Replicate files across a limited number of servers on a single LAN; Forward a request to one of the servers
  2. Mirror sites; Users select one of the mirror sites

Server-initiated replicas

- A server install temporary replicas to handle increased requires.
- Known as push caches.
- Issues
  - Where and when replicas should be added or deleted
  - Dynamic replication algorithm
    - Replicate to reduce the load on a server
    - Place in the proximity of clients
- Increasing used in Web hosting services

Server-Initiated Replicas



- Counting access requests from different clients.

Client-initiated replicas

- Known as (client) caches.
- To improve access times to data.
- How long data should be kept in a cache?
  - May become stale
  - Need to be deleted to make room for other data (LRU, FIFO, etc.)
- To improve cache hit, caches can be shared between clients.
- Prefetching

Update propagation

- What to propagate
  - A notification of an update
  - Actual data
  - Update operation
- Invalidation protocols
  - Use little network bandwidth
  - Work best when many updates compared to reads (i.e., read-to-write ratio is small)
- Transfer of modified data
  - Work best when read-to-write ratio is high
- Active replication
  - Transfer update operations with arguments
  - Trade-off communication with computation

Pull versus Push Protocols

- Update can be pushed or pulled.
- In the case of multiple client, single server systems:
  - A push-based approach uses server-based protocols
  - A pull-based approach uses client-based protocols
- Hybrid approach using lease

Issue	Push-based	Pull-based
State of server	List of client replicas and caches	None
Messages sent	Update (and possibly fetch update later)	Poll and update
Response time at client	Immediate (or fetch-update time)	Fetch-update time

Lease-based Approach

- A lease is a promise by a server that it will push updates to the client for a specified time.
- When a lease expires, the client needs to poll the server for updates and pull the modified data.
- Leases introduced by Gray and Cheriton (1989)
- Can be used to dynamically switch between push-base and pull-base approaches
- Questions: How long should be a lease
  - for frequently updated data?
  - for specified data that a client asks very infrequently?

Epidemic protocols

- Update propagation in eventual-consistent data stores
- A server that is part of a distributed data store is called
  - Infective: holds an update that it wants to spread.
  - Susceptible: has not yet been updated.
  - Removed: is not willing to spread its update.
- A server P picks another server Q at random to exchange updates with Q. Three approaches:
  1. P only pushes its own update to Q
  2. P only pulls in new updates from Q
  3. P and Q send updates to each other (i.e., pull-push)

Epidemic algorithms

- PARC developed a family of weak update protocols based on a disease metaphor (*epidemic algorithms* [Demers et. al. OSR 1/88]):
- Each replica periodically "touches" a selected "susceptible" peer site and "infects" it with updates.
  - Transfer every update known to the carrier but not the victim.
  - Partner selection is randomized using a variety of heuristics.
  - Theory shows that the epidemic will eventually infest the entire population (assuming it is connected).
    - Probability that replicas that have not yet converged decreases exponentially with time.
    - Heuristics (e.g., push vs. pull) affect traffic load and the expected time-to-convergence.

How to Ensure That Replicas Converge

- Using any form of epidemic (randomized) anti-entropy, all updates will (eventually) be known to all replicas.
- Imposing a global order on updates guarantees th at all sites (eventually) apply the same updates in the same order.
- Assuming conflict *resolution* is deterministic, all sites will resolve all conflicts in exactly the same way.

Issues and Techniques for Weak Replication

- How should replicas choose partners for anti-entropy exchanges?
  - Topology-aware choices minimize bandwidth demand by “flooding”, but randomized choices survive transient link failures.
- How to impose a global ordering on updates?
  - *logical clocks* and delayed delivery (or delayed commitment) of updates
- How to integrate new updates with existing database state?
  - Propagate updates rather than state, but how to detect and reconcile conflicting updates? Bayou: user-defined checks and *merge rules*.

Issues and Techniques for Weak Replication

- How to determine which updates to propagate to a peer on each anti-entropy exchange?
  - *vector timestamps*
- When can a site safely *commit* or *stabilize* received updates?
  - receiver acknowledgement by vector clocks