Optimistic (or Lazy) Replication

Alysson Bessani
DI-FCUL

Y. Saito & M. Shapiro. **Optimistic Replication**. ACM Computing Surveys. 37(1). 2005.

Outline

- Basic concepts
- Design choices
- Vector clocks and applications

Pessimistic Replication (PR)

- In most of this course, we discuss pessimistic/strongly consistent replication techniques
  - It tries to maintain **single-copy consistency**, i.e., **linearizability**, which give users an illusion of having a single highly-available copy of the data
  - On the downside, these replication techniques require blocking for disseminating information
  - This leads to bad performance as
    - The internet is slow and unreliable
    - This on-demand communication scales poorly in wide-area
    - Some applications require autonomy and quick feedback

Optimistic Replication (OR)

- A group of techniques for sharing data efficiently in wide-area or mobile environments
- The key feature that separates OR algorithms from PR is their approach to concurrency control
  - Pessimistic algorithms synchronously coordinate replicas during accesses and block other users during an update
  - Optimistic algorithms let data be accessed without a priori control based on assumptions that problems will occur rarely

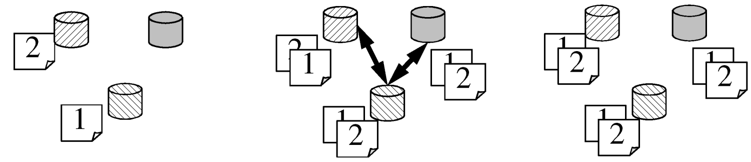
Optimistic Replication (OR)

- Advantages
  - Improved availability
  - Flexible w.r.t. networking
  - Scalable in number of replicas
  - Increased autonomy
- Disadvantages
  - Have to deal with diverging replicas and conflicts between concurrent operations
  - Less guarantees about the durability of an operation

OR Fundamentals

- Basic concepts
  - Object: minimal unit of replication
  - Replica: copy of an object stored in a site
  - Site: host that store copies of multiple objects
- Elements of optimistic replication (next slide)
  - Operation = precondition (for detecting conflicts) plus an update
    - An operations that fails its precondition is aborted
    - User submits operations locally to a site
    - The site updates its replica and exchange updates with others in background
  - Other elements:
    - propagation, tentative execution and scheduling, detecting and resolving conflicts, and commitment

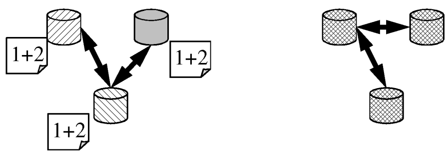
OR Fundamentals



(a) *Operation submission:* Users at different sites submit operations independently.

(b) *Propagation:* Sites communicate and exchange operations.

(c) *Scheduling:* Sites compute the ordering of operations.



(d) *Conflict resolution:* Sites detect conflicts and transform offending operations to produce results intended by users.

(e) *Commitment:* Sites agree on the final ordering and reconciliation result. Their changes become permanent.

Fig. 1. Elements of optimistic replication and their roles. Disks represent replicas, memo sheets represent operations, and arrows represent communications between replicas.

OR Fundamentals

- **Eventual consistency** is (intuitively) a weak consistency model in which replicas exchange updates in background and eventually reach the same state
- Formally, four conditions need to be satisfied:
  1. At any moment, for each replica, there is a (**committed**) **prefix** (CP) of the schedule that is equivalent to a prefix of the schedule of every other replica
  2. The CP of each replica grows monotonically over time
  3. All non-aborted operations in the CP satisfy their preconditions
  4. For every submitted operation *op*, either *op* or *abort(op)* will eventually be included in the CP

OR Design Choices and its Effects

Choice	Description	Effects
Number of writers	Which replicas can submit updates?	Defines the system's basic complexity, availability and efficiency.
Definition of operations	What kinds of operations are supported, and to what degree is a system aware of their semantics?	
Scheduling	How does a system order operations?	Defines the system's ability to handle concurrent operations.
Conflict management	How does a system define and handle conflicts?	
Operation propagation strategy	How are operations exchanged between sites?	Defines networking efficiency and the speed of replica convergence
Consistency guarantees	What does a system guarantee about the divergence of replica state?	Defines the transient quality of replica state.

Number of Writers

DNS, NIS, WWW/FTP mirroring

Active directory, Refdbms, Bayou, IceCube, Operational transformation

Usenet, Coda, Clearinghouse, Roam, Ficus, Palm

Single master ⟷ *Multi master*

Definition of Operations & Scheduling

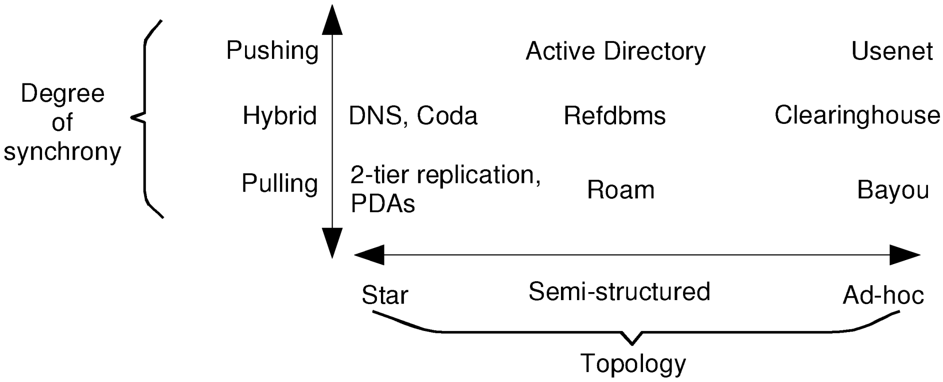
State transfer	Operation transfer: Syntactic	Operation transfer: Semantic
Usenet, DNS, Coda, Clearinghouse, Roam, Palm	Refdbms, Bayou, ESDS	IceCube, Operational transformation

- Scheduling: how operations are ordered?
  - **Syntactic**: based only on information about when, where, and by whom operations were submitted
  - **Semantic**: exploits information about the operation itself, such as commutativity or idempotency

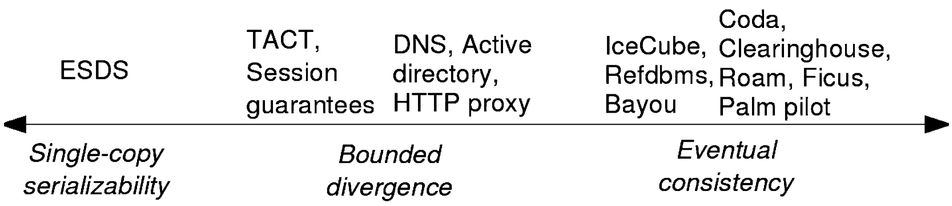
Conflict Handling

Prohibit	Ignore	Reduce	Detect&repair: Syntactic	Detect&repair: Semantic
Single master	Thomas write rule	Shrink objects Quick propagation App-specific ordering Divergence bounding	Two timestamps Vector timestamp	App-specific preconditions Canonical ordering Commuting updates

Propagation Strategies & Topologies



Consistency Guarantees



Comparing Main Strategies

	Single master, state transfer	Single master, op transfer	Multi master, state transfer	Multi master, op transfer
Availability	low: master single point of failure		high	
Conflict resolution flexibility	N/A		inflexible	flexible: semantic operation scheduling
Algorithmic complexity	very low		low	high: scheduling and commitment.
Space overhead	low: Tombstones	high: log	low: Tombstones	high: log
Network overhead	O(object-size)	O(#operations)	O(object-size)	O(#operations)

State-transfer System

- State-transfer systems restrict each operation to overwrite the entire object (blind write)
- These systems can converge simply by replicas receiving the newest content, skipping any intermediate operations
- Common update methods
  - Thomas write rule: only update if version more recent
  - **Vector clocks**: extension to Lamport's clock
- **Tombstones** (lists of deleted objects) need to be maintained to support object destruction
  - Otherwise, how to differentiate a deleted from a yet-to-be-created object?

Vector Clocks

- A data structure that accurately captures the *"happens before"* relation (see causal broadcast in previous lectures)
- A vector clock VC_i at site i is an M-element array (M = # master replicas) containing the last timestamp site i saw on other site j ($1 \le j \le M$)
- To submit a new operation *op*, site i increments $VC_i[i]$ and attach it to op (it will be the VC_{op}). A site j that receives *op* and VC_{op} knows that site i saw all its operations with timestamps up to $VC_{op}[j]$
- Let *a* and *b* be two operations. We say that **VC_b dominates VC_a** if $VC_a \neq VC_b$ and for all $1 \le k \le M$, $VC_a[k] \le VC_b[k]$.
- Operation *a* happens before *b* if and only if VC_b dominates VC_a
- If neither VC dominates the other, the operations are concurrent

- Why this is better than the Lamport's clock?
  - Hint: How to detect concurrency with Lamport clock?

Detecting Conflicts with Vector Clocks

- Two sites i and j maintain vector clocks (also called **Version Vectors**) for a every object replica they store
- At some point i and j exchange their VCs for a given object
- Conflicts are detected as follows:
  - If $VC_i = VC_j$, the replicas have not been modified
  - If VC_i dominates VC_j, i has a newer version than j (site i applied all updates of j, plus others). Site j copies the updated replicas and the VC from i (same in the symmetric case)
  - If neither vector dominates the other, the operations are concurrent and the system marks them as conflicts

- Vector clocks can also be used to decrease the amount of information exchanged between sites (propagation)

Propagation with Vector Clocks

```
—— Per-site data structures ——
type Operation = record
   issuer: SiteID  // The site that submitted the operation.
   ts: Timestamp  // The timestamp at the moment of issuance.
   op: Operation // Actual operation contents
var vc: array [1 .. M] of Timestamp  // The site's vector clock.
    log: set of Operation // The set of operation the site has received.
—— Called when submitting an operation ——
proc SubmitOperation(update)
   vc[myself] := vc[myself] + 1
   log := log ∪ { new Operation(issuer=myself, ts=vc[myself], op=update)}
—— Sender side: Send operations from this site to site dest ——
proc Send(dest)
   destVC := Receive dest's vector clock.
   upd := { u ∈ log | u.ts > destVC[u.issuer]}
   Send upd to dest.
—— Receiver side: Called via Send() ——
proc Receive(upd)
   for u ∈ upd
      Apply u.
      vc[u.issuer] := max(vc[u.issuer], u.ts)
   log := log ∪ upd
```

Fig. 15. Operation propagation using vector clocks. The receiving site first calls the sender's "Send" procedure and passes its vector clock. The sending site sends updates to the receiver which processes them in "Receive" procedure.

Propagation with Vector Clocks

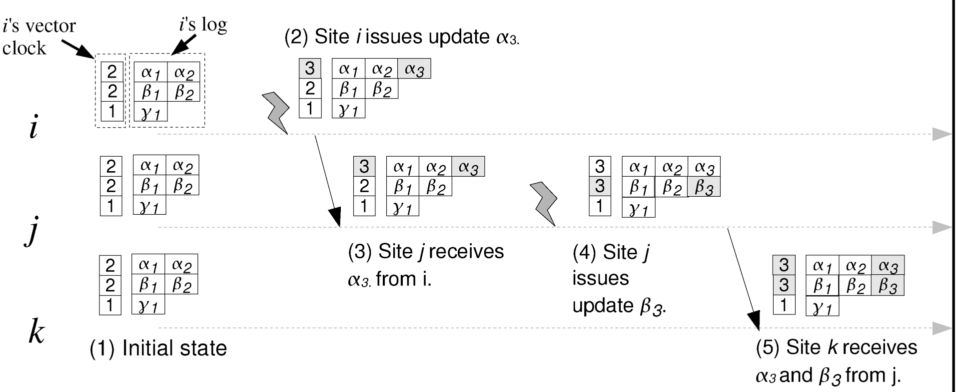


Fig. 16. Example of operation propagation using vector clocks. Symbols α, β and γ show updates submitted at i, j, and k, respectively. Shaded rectangles show changes at each step.