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Problems: replicas may be out of date, may accept conflicting writes, and may receive updates in different orders.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

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

Partial Solution: Allow writes to any \( N \) replicas (a quorum of size \( N \)). To be safe, reads must also request data from a quorum of replicas.

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

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).
Grapevine and Clearinghouse (Xerox)

Weakly consistent replication was used in earlier work at Xerox PARC:

- Grapevine and Clearinghouse name services
  Updates were propagated by unreliable multicast (“direct mail”).
- Periodic *anti-entropy exchanges* among replicas ensure that they eventually converge, even if updates are lost.
  Arbitrary pairs of replicas periodically establish contact and resolve all differences between their databases.
  Various mechanisms (e.g., MD5 digests and update logs) reduce the volume of data exchanged in the common case.
  Deletions handled as a special case via “death certificates” recording the delete operation as an update.
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 “touched” 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 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

1. Using any form of epidemic (randomized) anti-entropy, all updates will (eventually) be known to all replicas.

2. Imposing a global order on updates guarantees that all sites (eventually) apply the same updates in the same order.

3. Assuming conflict detection is deterministic, all sites will detect the same conflicts.

   Write conflicts cannot (generally) be detected when a site accepts a write; they appear when updates are applied.

3. Assuming conflict resolution is deterministic, all sites will resolve all conflicts in exactly the same way.
Issues and Techniques for Weak Replication

1. How should replicas choose partners for anti-entropy exchanges?
   Topology-aware choices minimize bandwidth demand by “flooding”,
   but randomized choices survive transient link failures.

2. How to impose a global ordering on updates?
   *logical clocks* and delayed delivery (or delayed commitment) of updates.

3. 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*.

4. How to determine which updates to propagate to a peer on each anti-entropy exchange?
   *vector clocks* or *vector timestamps*

5. When can a site safely *commit* or *stabilize* received updates?
   receiver acknowledgement by vector clocks (TSAE protocol)
Bayou Basics

1. Highly available, weak replication for mobile clients.
   
   *Beware*: every device is a “server”... let’s call ‘em sites.

2. Update conflicts are detected/resolved by rules specified by the application and transmitted with the update.
   
   interpreted dependency checks and merge procedures

3. Stale or tentative data may be observed by the client, but may mutate later.
   
   The client is aware that some updates have not yet been *confirmed*.
   
   “An inconsistent database is marginally less useful than a consistent one.”
Clocks

1. physical clocks

   Protocols to control drift exist, but physical clock timestamps cannot assign an ordering to “nearly concurrent” events.

2. logical clocks

   Simple timestamps guaranteed to respect causality: “A’s current time is later than the timestamp of any event A knows about, no matter where it happened or who told A about it.”

3. vector clocks

   Order(N) timestamps that say exactly what A knows about events on B, even if A heard it from C.

4. matrix clocks

   Order(N^2) timestamps that say what A knows about what B knows about events on C.

Acknowledgement vectors: an O(N) approximation to matrix clocks.
Update Ordering

**Problem:** how to ensure that all sites recognize a fixed order on updates, even if updates are delivered out of order?

**Solution:** Assign timestamps to updates at their accepting server, and order them by source timestamp at the receiver.

Assign nodes unique IDs: break ties with the origin node ID.

- What (if any) ordering exists between updates accepted by different servers?

  Comparing physical timestamps is arbitrary: physical clocks drift.

  Even a protocol to maintain loosely synchronized physical clocks cannot assign a meaningful ordering to events that occurred at “almost exactly the same time”.

- In Bayou, received updates may affect generation of future updates, since they are immediately visible to the user.
Causality

Constraint: The update ordering must respect potential causality.

- Communication patterns establish a \textit{happened-before} order on events, which tells us when ordering \textit{might} matter.

- Event \( e_1 \) \textit{happened-before} \( e_2 \) iff \( e_1 \) could possibly have affected the generation of \( e_2 \): we say that \( e_1 < e_2 \).

  \[
  e_1 < e_2 \text{ iff } e_1 \text{ was “known” when } e_2 \text{ occurred.}
  \]

  Events \( e_1 \) and \( e_2 \) are \textit{potentially causally related}.

- In Bayou, users or applications may perceive inconsistencies if causal ordering of updates is not respected at all replicas.

  An update \( u \) should be ordered after all updates \( w \) known to the accepting server at the time \( u \) was accepted.

  e.g., the newsgroup example in the text.
Causality: Example

A1 < B2 < C2
B3 < A3
C2 < A4
Logical Clocks

Solution: timestamp updates with logical clocks [Lamport]

Timestamping updates with the originating node’s logical clock $LC$ induces a partial order that respects potential causality.

Clock condition: $e_1 < e_2$ implies that $LC(e_1) < LC(e_2)$

1. Each node maintains a monotonically increasing clock value $LC$.
2. Globally visible events (e.g., updates) are timestamped with the current $LC$ value at the generating node.
   
   Increment local $LC$ on each new event: $LC = LC + 1$
3. Piggyback current clock value on all messages.
   
   Receiver resets local $LC$: if $LC_s > LC_r$ then $LC_r = LC_s + 1$
Logical Clocks: Example

A6-A10: receiver’s clock is unaffected because it is “running fast” relative to sender.

C5: LC update advances receiver’s clock if it is “running slow” relative to sender.
Which Updates to Propagate?

In an anti-entropy exchange, $A$ must send $B$ all updates known to $A$ that are not yet known to $B$.

Problem: which updates are those?

one-way “push” anti-entropy exchange (Bayou reconciliation)

“Here’s what I know that you don’t know.”

“A

“What do you know?”

“Here’s what I know.”

B

Systems & Architecture
Flooding and the Prefix Property

In Bayou, each replica’s knowledge of updates is determined by its pattern of communication with other nodes.

Loosely, a site knows everything that it could know from its contacts with other nodes.

• Anti-entropy transfers *flood* updates.
  Tag each update originating from site *i* with *accept stamp* \((i, LC_i)\).
  Updates from each site are bulk-transmitted *cumulatively* in an order consistent with their source accept stamps.

• Flooding guarantees the *prefix property* of received updates.
  If a site knows an update \(u\) originating at site \(i\) with accept stamp \(LC_u\), then it also knows all preceding updates \(w\) originating at site \(i\): those with accept stamps \(LC_w < LC_u\).
Causality and Reconciliation

In general, a transfer from $A$ must send $B$ all updates that did not happen-before any update known to $B$.

Can we determine which updates to propagate by comparing logical clocks $LC(A)$ and $LC(B)$? NO.
Causality and Updates: Example

A1 < B2 < C3
B3 < A4
C3 < A5
Motivation for Vector Clocks

*Logical* clocks induce an order consistent with causality, but it is actually stronger than causality.

- The converse of the *clock condition* does not hold: it may be that $LC(e_1) < LC(e_2)$ even if $e_1$ and $e_2$ are concurrent.

  If $A$ could have known anything $B$ knows, then $LC_A > LC_B$, but if $LC_A > LC_B$ then that doesn’t make it so.

  Concurrent updates may be ordered unnecessarily.

- In Bayou, logical clocks tell us that $A$ has *not* seen any update $u$ with $LC(u) > LC_A$, but what has $A$ seen?

  $LC_A$ does not say if $A$ saw a given update $w$ with $LC(w) < LC_A$.

  We need a clock mechanism that is not so sloppy about capturing causality.
Vector Clocks

*Vector clocks* (aka *vector timestamps*) are a more detailed representation of what a node might know.

1. In a system with $N$ nodes, each site keeps a vector timestamp $TS[N]$ as well as a logical clock $LC$.
   
   $TS[j]$ at site $i$ is the most recent value of site $j$’s logical clock that site $i$ “heard about”.
   
   $TS_i[i] = LC_i$: each site $i$ keeps its own $LC$ in $TS[i]$.

2. When site $i$ generates a new event, it increments its logical clock.
   
   $TS_i[i] = TS_i[i] + 1$

3. A site $r$ observing an event (e.g., receiving a message) from site $s$ sets its $TS_r$ to the pairwise maximum of $TS_s$ and $TS_r$.

   For each site $i$, $TS_r[i] = \text{MAX}(TS_r[i], TS_s[i])$
Vector Clocks: Example

A1 (1, 0, 0) → A2 (2, 0, 0) → A4 (4, 3, 0) → A5 (5, 3, 3)
B1 (0, 1, 0) → B2 (1, 2, 0) → B3 (1, 3, 0) → B4 (1, 4, 0)
C1 (0, 0, 1) → C3 (1, 2, 3) → C4 (1, 2, 4)

Question: what if I have two updates to the same data item, and neither timestamp dominates the other?
Vector Clocks and Causality

Vector clocks induce an order that exactly reflects causality.

- Tag each event $e$ with current $TS$ vector at originating site.
  
  vector timestamp $TS(e)$

- $e_1$ happened-before $e_2$ if and only if $TS(e_2)$ dominates $TS(e_1)$
  
  $e_1 < e_2$ iff $TS(e_1)[i] < TS(e_2)[i]$ for each site $i$

  “Everything potentially known when $e_1$ was generated must also potentially have been known when $e_2$ was generated.”

- Vector timestamps allow us to ask if two events are concurrent, or if one happened-before the other.

  If $e_1 < e_2$ then $LC(e_1) < LC(e_2)$ and $TS(e_2)$ dominates $TS(e_1)$.

  If $TS(e_2)$ does not dominate $TS(e_1)$ then it is not true that $e_1 < e_2$. 
Reconciliation with Vector Clocks

A can determine which updates to pass to B by comparing their current vector clocks.

Tag each update originating from site i with accept stamp \((i, LC_i)\).

If \(TS_A[i] > TS_B[i]\), then A has updates from i that B has not seen, i.e., they did not happen-before any updates known to B.

A sends B all updates from site i tagged with accept stamps \(LC\) such that \(TS_B[i] < LC \leq TS_A[i]\).

"What do you know?"

A

"Here’s what I know: TS_B"

"Here’s what I know that you don’t know: TS_A - TS_B"

B
The Prefix Property and Reconciliation

Vector clocks work for anti-entropy transfers because they precisely encapsulate which updates a site has seen.

- The *prefix property* must hold for this to work.
  
  If a site knows an update $u$ originating at site $i$ with accept stamp $LC_u$, then it also knows all preceding updates $w$ originating at site $i$: those with accept stamps $LC_w < LC_u$.

- $TS_B[i]$ is the origin timestamp ($LC$) of the *latest* update generated by $i$ and received at $B$.

- Updates $w$ with origin timestamps $LC(w) > TS_B[i]$ are exactly those updates that did not *happen-before* $TS_B$.
  
  If $LC(w) > TS_B[i]$, then $TS_B$ cannot dominate $TS(w)$, so $w$ cannot be known to $B$.

(Converse left as an exercise.)
When to Discard or Stabilize Updates?

Problem 1: when can a site discard its pending updates?

When can A know that every other site has seen some update u?

Problem 2: when to commit (stabilize) pending updates?

A committed update w is stable: no other update u can arrive with u < w; w will never need to be rolled back.

These two questions are equivalent if we also propagate “knowledge about knowledge” with anti-entropy transfers.

Suppose we know that each peer site s has received this update w.
We had transitive contact with s since it received w.
We have received all updates known by s when it received w.
We have received all updates generated by s before it received w.
We have received all updates generated before w: w is stable.
The Need for Propagating Acknowledgements

Vector clocks tell us what $B$ knows about $C$, but they do not reflect what $A$ knows about what $B$ knows about $C$.

Nodes need this information to determine when it is safe to discard/stabilize updates.

- $A$ can always tell if $B$ has seen an update $u$ by asking $B$ for its vector clock and looking at it.
  
  If $u$ originated at site $i$, then $B$ knows about $u$ if and only if $TS_B$ covers its accept stamp $LC_u$: $TS_B[i] >= LC_u$.

- $A$ can only know that every site has seen $u$ by looking at the vector clocks for every site.
  
  Even if $B$ recently received updates from $C$, $A$ cannot tell (from looking at $B$’s vector clock) if $B$ got $u$ from $C$ or if $B$ was already aware of $u$ when $C$ contacted it.
Matrix Clocks

*Matrix clocks* extend vector clocks to capture “what A knows about what B knows about C”.

- Each site $i$ maintains a matrix $MC_i(N,N)$.
  
  Row $j$ of $i$’s matrix clock $MC_i$ is the most recent value of $j$’s vector clock $TS_j$ that $i$ has heard about.

  $MC_i[i, i] = LC_i$ and $MC_i[i, \ast] = TS_i$

  $MC_i[j,k] = $ what $i$ knows about what $j$ knows about what happened at $k$.

- If $A$ sends a message to $B$, then $MC_B$ is set to the pairwise maximum of $MC_A$ and $MC_B$.
  
  If $A$ knows that $B$ knows $u$, then after $A$ talks to $C$, $C$ knows that $B$ knows $u$ too.
Propagating Acknowledgment Stamps

Matrix clocks require $N^3$ state for a system with $N$ nodes.

Propagate $N^2$ state on each exchange.

For anti-entropy, we can conservatively approximate the matrix clock using only $N^2$ state. [Golding]

- After $A$ passes updates to $B$, compute $B$’s **ack stamp** as the lowest $LC$ entry in $TS_B$.

- Each node keeps an **acknowledgement summary vector** $AS(N)$ of ack stamps for every other node “last it heard”.

- In an anti-entropy exchange from $A$ to $B$, compute $AS_B$ as a pairwise maximum of $AS_A$ and $AS_B$.

$AS_i[j]$ does not tell $i$ what $j$ knows about $k$, but it does say that $j$ knows about every event at $k$ prior to $AS_i[j]$, for every $k$. 
Golding’s TSAE Protocol

Golding defined a Timestamped Anti-Entropy (TSAE) protocol that predates Bayou.

- designed for replicated Internet services (e.g., refdbms)
- reconciliation by two-way pairwise anti-entropy exchanges flooded updates
- studied role of network topology in partner selection
- uses logical clocks for accept stamps
total commit ordering defined by logical clocks
- propagates knowledge of peer replica state by ack stamps and acknowledgment vectors
At the end of this example, everyone knows that everyone has seen all updates with accept stamps of 1, regardless of where they originated.

What else would be known if we used matrix clocks instead?
Matrix clocks and acknowledgement vectors can impede forward progress in the presence of failures.

- If a replica $A$ fails or becomes disconnected, other nodes recognize that it is “getting behind”.
  
  No node’s acknowledgment stamp can advance beyond the accept stamp $LC(w)$ of the last update $w$ received by $A$.

- If a replica gets behind, other nodes cannot retire received updates $u$ with $LC(u) > LC(w)$.

- One solution is to forcibly remove the disconnected node $A$ from the replica set.
  
  How to bring $A$ up to date if it later rejoins the replica set? How to order updates generated by $A$ while disconnected?
Committing Updates in Bayou

Bayou commits updates more aggressively using a primary-commit protocol.

- A single site is designated as the primary.
- The primary commits updates as it receives them.
  
  Primary assigns a commit sequence number (CSN) to each committed update.

  The final total update order is defined by CSN order.

- Sites learn of commitment through anti-entropy transfers.

  A site may learn of an update before learning that the update has been committed by the primary.

  Sites learn of commitment in CSN order. Updates known to be committed are stable: their order will never change.
Reconciliation with CSNs

Each site also maintains a knownCSN counter, the CSN of the latest committed update the site knows has committed.

In an anti-entropy transfer, A looks at B’s knownCSN.

If A knows update w has committed, and CSN(w) > knownCSN_B, then A notifies B that w has committed.

B updates knownCSN_B as it learns of committed updates.

(This assumes that sites learn of commitment in CSN order.)

A

“Here’s what I know that you don’t know: (knownCSN_A - knownCSN_B), (TS_A - TS_B)”

B

“Here’s what I know: (knownCSN_B, TS_B)”

“What do you know?”
Bayou Update Logs: Example

ordering properties
1. Each site sees/logs its own updates in accept order.
2. Each site sees/logs updates from other sites in accept order.
3. Each site sends updates in its update log order.
4. Each site sees all updates in a causal order.
5. Each site reorders received updates to causal timestamp order.
Suppose C is the primary
1. C commits updates in its update-log order.
2. New constraint: all sites order known-committed updates before all tentative updates.
3. Sites propagate updates/knowledge in update log order: known-committed updates propagate before tentative updates, and commit knowledge propagates in CSN order.
4. Can CSN order violate causality? Can it violate a total order based on accept stamps?
Discarding Updates in Bayou

Any site $A$ may truncate any prefix of the stable (committed) portion of its update log to reclaim space.

- $A$ needs no record of known-committed updates for itself.
  Committed updates are never rolled back, since commits are received in CSN order, the same as the final order.
- $A$ keeps stable updates in its log only so it can tell other sites about those updates.
- $A$ may discard stable updates if there is some other to reconcile with peer replicas that need those updates.

  Easy: send entire database state before sending updates.

  *Truncation is a tradeoff:* it reclaims local storage, but may make later reconciliations more expensive.
Reconciliation with Update Log Truncation

Each site maintains an *omitCSN* stamp to characterize the omitted log prefix.

*omitCSN*$_A$ is the latest CSN omitted from A’s log.

An incremental update transfer is insufficient if *omitCSN*$_A$ > *knownCSN*$_B$.

“*What do you know?*”

“Here’s what I know: (*knownCSN*$_B$, *TS*$_B$)”

“A haven’t forgotten anything I once knew that you don’t know yet. Here’s what I know that you don’t know: (*knownCSN*$_A$ - *knownCSN*$_B$), (*TS*$_A$ - *TS*$_B$).”
Reconciling a Lagging Server

What if a server is too far out of date for an incremental transfer?

- **A** can’t just send its entire database state.
  
  **B** must know about **A**’s pending tentative updates in case they must be rolled back.
  
  **B** may possess updates not known to **A**: these cannot be discarded.

- Instead, **A** must send a “committed view” of its database with all tentative updates rolled back.

  ...then complete the protocol as before by sending logged updates and commit records unknown to **B**.

  As described in the paper, Bayou rolls back *all* logged updates including committed updates, but I don’t see why this is necessary.

  **B** then rolls its log forward in the usual way, including any new updates from **A**, possibly interleaved with updates **B** already had.
That Pesky “O-Vector”

Problem: what if $B$ has a logged update $w$ that was already committed and discarded by $A$?

- $B$ saw $w$, but did not know that it was committed.
- $B$ cannot tell from $\text{omitCSN}_A$ that $A$’s committed view reflects $w$.
- $B$ must be prevented from reapplying $w$.

- **Solution**: Each site also keeps an $\text{omitTS}$ timestamp vector to characterize the omitted prefix of its update log.

  $\text{omitTS}_A[i]$ is the highest accept stamp of any update originating at site $i$ that was omitted from $A$’s log.

  On a full database transfer, roll back $A$’s state to $\text{omitTS}_A$, send that state with $\text{omitTS}_A$, then set $\text{omitTS}_B = \text{omitTS}_A$. This is safe because $\text{omitTS}_A$ must dominate the “old” $\text{omitTS}_B$.

- $B$ strikes from its log any updates $w$ covered by $\text{omitTS}_B$. 
Reconciliation Using Transportable Media

1. “Sender” dumps its replica state on media (e.g., a floppy).
   
   E.g., A dumps its update log to the disk, prefixed by \((\text{omitCSN}_A, \text{omitTS}_A)\) and \((\text{CSN}_A, \text{TS}_A)\).

2. “Receiver” must ask two questions:
   - Am I advanced enough to accept the updates on this disk?
     Is \(\text{CSN}_I > \text{omitCSN}_A\) ?
   - Will I learn anything from the updates on this disk?
     Is \(\text{CSN}_A > \text{CSN}_I\)? Or is \(\text{TS}_A[k] > \text{TS}_I[k]\), for any \(k\)?

3. This is exactly like network anti-entropy transfer...
   ...except the receiver has access to the sender’s state, and executes both sides of the protocol.
Questions About Bayou

1. What is the effect of a failed primary?

2. What will happen when you reconnect your laptop after you return from a long vacation on a tropical island?

3. If it was a working vacation with some buddies, can you assume that your replicas will converge in the absence of the primary to define the commit order?

4. Can you assume that the interleaving of updates observed during your vacation will be preserved when you reconnect?

5. What if one person goes home and reconnects before the others?

6. How to create/retire replicas and notify peers that they exist?