## Agenda

- Review of Bayou
- Channeling Eric Brewer (CAP theorem)
- A peek at fault tolerance

## Why Bayou?

- Eventual consistency: strongest scalable consistency model
- But not strong enough for mobile clients
  - Accessing different replicas can lead to strange results
  - Application-independent conflict detection misses some conflicts and creates others falsely
- Bayou was designed to move beyond eventual consistency
  - Session guarantees
  - Application-specific conflict detection and resolution

## Review of Bayou

*With examples!*

## Bayou System Assumptions

- Variable degrees of connectivity:
  - Connected, disconnected, and weakly connected
- Variable end-node capabilities:
  - Workstations, laptops, PDAs, etc.
- Availability crucial

## Resulting Design Choices

- Variable connectivity ⇒ Flexible update propagation
  - Incremental progress, pairwise communication
- Variable end-nodes ⇒ Flexible notion of clients and servers
  - Some nodes keep state (servers), some don't (clients)
  - Laptops could have both, PDAs probably just clients
- Availability crucial ⇒ Must allow disconnected operation
  - Conflicts inevitable
  - Use application-specific conflict detection and resolution
Components of Design

- Update propagation
- Conflict detection
- Conflict resolution
- Session guarantees

Updates

- Identified by a triple:
  - Commit-stamp
  - Time-stamp
  - Server-ID of accepting server
- Updates are either committed or tentative
  - Commit stamps increase monotonically
  - Tentative updates have commit-stamp=inf
- Primary server does all commits: (why?)
  - It sets the commit-stamp
  - Commit-stamp different from time-stamp

Update Log

- Update log in order:
  - Committed updates (in commit-stamp order)
  - Tentative updates (in time-stamp order)
- Can truncate committed updates, and only keep db state
  - Why?
- Clients can request two views: (or other app-specific views)
  - Committed view
  - Tentative view

Tentative vs Committed Views

- Committed view:
  - Updates will never be reordered
  - But may be substantially out-of-date
- Tentative view:
  - Much more current
  - But updates might be reordered
- Tradeoff is application-dependent:
  - Calendars: avoid tentative commitments, but don’t count on them
  - Weather: being current more important than permanence

Anti-Entropy Exchange

- Each server keeps a version vector:
  - R.V[X] is the latest timestamp from server X that server R has seen
- When two servers connect, exchanging the version vectors allows them to identify the missing updates
- These updates are exchanged in the order of the logs, so that if the connection is dropped the crucial monotonicity property still holds
  - If a server X has an update accepted by server Y, server X has all previous updates accepted by that server

Requirements for Eventual Consistency

- Universal propagation: anti-entropy
- Globally agreed ordering: commit-stamps
- Determinism: writes do not involve information not contained in the log (no time-of-day, process-ID, etc.)
Example with Three Servers

All Servers Write Independently

P and A Do Anti-Entropy Exchange

P Commits Some Early Writes

P and B Do Anti-Entropy Exchange

P Commits More Writes
Bayou Writes

- Identifier (commit-stamp, time-stamp, server-ID)
- Nominal value
- Write dependencies
- Merge procedure

Conflict Detection

- Write specifies the data the write depends on:
  - Set X=8 if Y=5 and Z=3
  - Set Cal(11:00-12:00)=dentist if Cal(11:00-12:00) is null

Conflict Resolution

- Specified by merge procedure (mergeproc)
- When conflict is detected, mergeproc is called
  - Move appointments to open spot on calendar
  - Move meetings to open room

Session Guarantees

- Ensured by client, not by distribution mechanism
- Needed to ensure user sees sensible results

To implement, client records:
  - All writes during that session (write-set)
  - The writes relevant to each read read-set
    - Must be supplied by server
    - Can be approximated by version vector

The Four Session Guarantees

<table>
<thead>
<tr>
<th>Guarantee</th>
<th>State updated</th>
<th>State checked</th>
</tr>
</thead>
<tbody>
<tr>
<td>Read your writes</td>
<td>Write</td>
<td>Read</td>
</tr>
<tr>
<td>Monotonic reads</td>
<td>Read</td>
<td>Read</td>
</tr>
<tr>
<td>Writes follow reads</td>
<td>Read</td>
<td>Write</td>
</tr>
<tr>
<td>Monotonic writes</td>
<td>Write</td>
<td>Write</td>
</tr>
</tbody>
</table>

Example

- Return to example with servers P, A, and B
- Client attaches to server P with vector [8,3,5]
- Client reads, with read-set {P6,A1,A2,B5}
- Client writes, with timestamp P9
- Client then detaches and reattaches to another server
- For which of these vectors can client read or write?
What Reads/Writes are Allowed?

Read-set \{P6,A1,A2,B5\}, Write-set P9

- \[7,1,6\] Read Your Writes: No
  Monotonic Reads: No
  Monotonic Writes: No
  No R, No W

- \[7,4,6\] Read Your Writes: No
  Monotonic Reads: Yes
  Monotonic Writes: No
  Write Following Reads: Yes
  No R, No W

What Reads/Writes are Allowed?

Read-set \{P6,A1,A2,B5\}, Write-set P9

- \[9,3,4\] Read Your Writes: Yes
  Monotonic Reads: No
  Monotonic Writes: Yes
  No R, No W

- \[10,3,8\] Read Your Writes: Yes
  Monotonic Reads: Yes
  Monotonic Writes: Yes
  Write Following Reads: Yes
  R, W

Channeling Eric Brewer

Slightly more hair, much less wisdom

A Clash of Cultures

- Classic distributed systems: focused on ACID semantics
  - A: Atomic
  - C: Consistent
  - I: Isolated
  - D: Durable

- Modern Internet systems: focused on BASE
  - Basically Available
  - Soft-state (or scalable)
  - Eventually consistent

ACID vs BASE

<table>
<thead>
<tr>
<th>ACID</th>
<th>BASE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Strong consistency for</td>
<td>Availability and scaling</td>
</tr>
<tr>
<td>transactions highest</td>
<td>highest priorities</td>
</tr>
<tr>
<td>priority</td>
<td>Weak consistency</td>
</tr>
<tr>
<td>Availability less</td>
<td>Optimistic</td>
</tr>
<tr>
<td>important</td>
<td>Best effort</td>
</tr>
<tr>
<td>Pessimistic</td>
<td>Simple and fast</td>
</tr>
<tr>
<td>Rigorous analysis</td>
<td></td>
</tr>
<tr>
<td>Complex mechanisms</td>
<td></td>
</tr>
</tbody>
</table>

Why the Divide?

- What goals might you want from a shared-date system?
  - C, A, P

- Strong Consistency: all clients see the same view, even in the presence of updates

- High Availability: all clients can find some replica of the data, even in the presence of failures

- Partition-tolerance: the system properties hold even when the system is partitioned
**CAP Conjecture (later theorem)**

- You can only have two out of these three properties
- The choice of which feature to discard determines the nature of your system

**Consistency and Availability**

- Comment:
  - Providing transactional semantics requires all nodes to be in contact with each other
- Examples:
  - Single-site and clustered databases
  - Other cluster-based designs
- Typical Features:
  - Two-phase commit
  - Cache invalidation protocols
  - Classic DS style

**Consistency and Partition-Tolerance**

- Comment:
  - If one is willing to tolerate system-wide blocking, then can provide consistency even when there are temporary partitions
- Examples:
  - Distributed databases
  - Distributed locking
  - Quorum (majority) protocols
- Typical Features:
  - Pessimistic locking
  - Minority partitions unavailable
  - Also common DS style
  - Voting vs primary replicas

**Partition-Tolerance and Availability**

- Comment:
  - Once consistency is sacrificed, life is easy…
- Examples:
  - DNS
  - Web caches
  - Coda
  - Bayou
- Typical Features:
  - TTLs and lease cache management
  - Optimistic updating with conflict resolution
  - This is the "Internet design style"

**Techniques**

- Expiration-based caching: AP
- Quorum/majority algorithms: PC
- Two-phase commit: AC

**Byzantine**
### Failures

- So far, have assume nodes are either up or down
- But nodes are far more interesting than that!

### Failure Models

<table>
<thead>
<tr>
<th>Type of Failure</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Crash failure</td>
<td>A server fails, but is working correctly until it fails</td>
</tr>
<tr>
<td>Value failure</td>
<td>A server fails to respond to incoming requests</td>
</tr>
<tr>
<td>Deadlock</td>
<td>A server fails to receive incoming messages</td>
</tr>
<tr>
<td>Timing failure</td>
<td>A server's response lies outside the specified time interval</td>
</tr>
<tr>
<td>Response failure</td>
<td>The server's response is incorrect</td>
</tr>
<tr>
<td>Value error</td>
<td>The value of the response is wrong</td>
</tr>
<tr>
<td>State transition failure</td>
<td>The server deviates from the correct flow of control</td>
</tr>
<tr>
<td>Arbitrary failure</td>
<td>A server may produce arbitrary responses at arbitrary times</td>
</tr>
</tbody>
</table>

### Previous Algorithms

- Only cope with crash-failure
- What happens if some other failure occurs?
- Bayou as an example:
  - If server lies about updates, algorithm gets hopelessly confused
- Generally, most other distributed protocols fail when faced with anything other than crash failures
- Next: how to deal with a wider variety of failures

### Same Dichotomy Exists

- Classic Distributed Systems:
  - Byzantine Algorithms
  - Two-phase Commit
- Internet style:
  - Checkable or “self-verifying” protocols
  - Very new field in Internet research
  - You now know as much as we do about it.....

---

Page 7