CS 268: Lecture 20
Classic Distributed Systems: Bayou and BFT

Ion Stoica
Computer Science Division
Department of Electrical Engineering and Computer Sciences
University of California, Berkeley
Berkeley, CA 94720-1776

Agenda

- Introduction and overview
- Data replication and eventual consistency
- Bayou: beyond eventual consistency
- Practical BFT: fault tolerance
Why Distributed Systems in 268?

- You won’t learn it any other place....
- Networking research is drifting more towards DS
- DS research is drifting more towards the Internet

Two Views of Distributed Systems

- **Optimist**: A distributed system is a collection of independent computers that appears to its users as a single coherent system

- **Pessimist**: “You know you have one when the crash of a computer you’ve never heard of stops you from getting any work done.” (Lamport)
Recurring Theme

- Academics like:
  - Clean abstractions
  - Strong semantics
  - Things that prove they are smart

- Users like:
  - Systems that work (most of the time)
  - Systems that scale
  - Consistency *per se* isn’t important

- Eric Brewer had the following observations

A Clash of Cultures

- Classic distributed systems: focused on ACID semantics (transaction semantics)
  - Atomicity: either the operation (e.g., write) is performed on all replicas or is not performed on any of them
  - Consistency: after each operation all replicas reach the same state
  - Isolation: no operation (e.g., read) can see the data from another operation (e.g., write) in an intermediate state
  - Durability: once a write has been successful, that write will persist indefinitely

- 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 priority</td>
<td>highest priorities</td>
</tr>
<tr>
<td>Availability less important</td>
<td>Weak consistency</td>
</tr>
<tr>
<td>Pessimistic</td>
<td>Optimistic</td>
</tr>
<tr>
<td>Rigorous analysis</td>
<td>Best effort</td>
</tr>
<tr>
<td>Complex mechanisms</td>
<td>Simple and fast</td>
</tr>
</tbody>
</table>

## Why Not ACID+BASE?

- **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 Theorem [Brewer]

- 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 functioning nodes to be in contact with each other (no partition)

- **Examples:**
  - Single-site and clustered databases
  - Other cluster-based designs

- **Typical Features:**
  - Two-phase commit
  - Cache invalidation protocols
  - Classic DS style
Partition-Tolerance and Availability

- Comment:
  - Once consistency is sacrificed, life is easy….

- Examples:
  - DNS
  - Web caches
  - Practical distributed systems for mobile environments: Coda, Bayou

- Typical Features:
  - Optimistic updating with conflict resolution
  - This is the “Internet design style”
  - TTLs and lease cache management

Voting with their Clicks

- In terms of large-scale systems, the world has voted with their clicks:
  - Consistency less important than availability and partition-tolerance
Data Replication and Eventual Consistency

Replication

- Why replication?
  - Volume of requests
  - Proximity
  - Availability

- Challenge of replication: consistency
Many Kinds of Consistency

- **Strict**: updates happen instantly everywhere
  - A read has to return the result of the latest write which occurred on that data item
  - Assume instantaneous propagation; not realistic
- **Linearizable**: updates appear to happen instantaneously at some point in time
  - Like “Sequential” but operations are ordered using a global clock
  - Primarily used for formal verification of concurrent programs
- **Sequential**: all updates occur in the same order everywhere
  - Every client sees the writes in the same order
    - Order of writes from the same client is preserved
    - Order of writes from different clients may not be preserved
  - Equivalent to Atomicity + Consistency + Isolation
- **Eventual consistency**: if all updating stops then eventually all replicas will converge to the identical values

---

Eventual Consistency

- If all updating stops then eventually all replicas will converge to the identical values
Implementing Eventual Consistency

Can be implemented with two steps:

1. All writes eventually propagate to all replicas

2. Writes, when they arrive, are written to a log and applied in the same order at all replicas
   • Easily done with timestamps and “undo-ing” optimistic writes

Update Propagation

• Rumor or epidemic stage:
  - Attempt to spread an update quickly
  - Willing to tolerate incompletely coverage in return for reduced traffic overhead

• Correcting omissions:
  - Making sure that replicas that weren’t updated during the rumor stage get the update
Anti-Entropy

- Every so often, two servers compare complete datasets
- Use various techniques to make this cheap
- If any data item is discovered to not have been fully replicated, it is considered a new rumor and spread again

Bayou
System Assumptions

- Early days: nodes always on when not crashed
  - Bandwidth always plentiful (often LANs)
  - Never needed to work on a disconnected node
  - Nodes never moved
  - Protocols were “chatty”

- Now: nodes detach then reconnect elsewhere
  - Even when attached, bandwidth is variable
  - Reconnection elsewhere means often talking to different replica
  - Work done on detached nodes

Disconnected Operation

- Challenge to old paradigm
  - Standard techniques disallowed any operations while disconnected
  - Or disallowed operations by others

- But eventual consistency not enough
  - Reconnecting to another replica could result in strange results
    - E.g., not seeing your own recent writes
  - Merely letting latest write prevail may not be appropriate
  - No detection of read-dependencies

- What do we do?
Bayou

- System developed at PARC in the mid-90’s
- First coherent attempt to fully address the problem of disconnected operation
- Several different components
Motivating Scenario: Shared Calendar

- Calendar updates made by several people
  - e.g., meeting room scheduling, or exec+admin

- Want to allow updates offline

- But conflicts can’t be prevented

- Two possibilities:
  - Disallow offline updates?
  - Conflict resolution?

Conflict Resolution

- Replication not transparent to application
  - Only the application knows how to resolve conflicts
  - Application can do record-level conflict detection, not just file-level conflict detection
  - Calendar example: record-level, and easy resolution

- Split of responsibility:
  - Replication system: propagates updates
  - Application: resolves conflict

- Optimistic application of writes requires that writes be “undo-able”
### Meeting room scheduler

- Reserve same room at same time: conflict
- Reserve different rooms at same time: no conflict
- Reserve same room at different times: no conflict
- Only the application would know this!

<table>
<thead>
<tr>
<th>Room</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rm1</td>
<td></td>
</tr>
<tr>
<td>Rm2</td>
<td></td>
</tr>
</tbody>
</table>

No conflict

---

### Meeting Room Scheduler

<table>
<thead>
<tr>
<th>Room</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rm1</td>
<td></td>
</tr>
<tr>
<td>Rm2</td>
<td></td>
</tr>
</tbody>
</table>

No conflict
Meeting Room Scheduler

Rm1

Rm2

conflict

time

Meeting Room Scheduler

Rm1

Rm2

No conflict

time

Page 15
Meeting Room Scheduler

Other Resolution Strategies

- Classes take priority over meetings
- Faculty reservations are bumped by admin reservations
- Move meetings to bigger room, if available
- Point:
  - Conflicts are detected at very fine granularity
  - Resolution can be policy-driven
Updates

- Client sends update to a server

- 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:
  - It sets the commit-stamp
  - Commit-stamp different from time-stamp

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

<table>
<thead>
<tr>
<th>P</th>
<th>A</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;inf,1,P&gt;</td>
<td>&lt;inf,2,A&gt;</td>
<td>&lt;inf,1,B&gt;</td>
</tr>
<tr>
<td>&lt;inf,4,P&gt;</td>
<td>&lt;inf,3,A&gt;</td>
<td>&lt;inf,5,B&gt;</td>
</tr>
<tr>
<td>&lt;inf,8,P&gt;</td>
<td>&lt;inf,10,A&gt;</td>
<td>&lt;inf,9,B&gt;</td>
</tr>
<tr>
<td>[8,0,0]</td>
<td>[0,10,0]</td>
<td>[0,0,9]</td>
</tr>
</tbody>
</table>

### P and A Do Anti-Entropy Exchange

<table>
<thead>
<tr>
<th>P</th>
<th>A</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;inf,1,P&gt;</td>
<td>&lt;inf,2,A&gt;</td>
<td>&lt;inf,1,B&gt;</td>
</tr>
<tr>
<td>&lt;inf,2,A&gt;</td>
<td>&lt;inf,3,A&gt;</td>
<td>&lt;inf,5,B&gt;</td>
</tr>
<tr>
<td>&lt;inf,4,P&gt;</td>
<td>&lt;inf,4,P&gt;</td>
<td>&lt;inf,9,B&gt;</td>
</tr>
<tr>
<td>&lt;inf,8,P&gt;</td>
<td>&lt;inf,8,P&gt;</td>
<td>[0,0,9]</td>
</tr>
<tr>
<td>&lt;inf,10,A&gt;</td>
<td>&lt;inf,10,A&gt;</td>
<td></td>
</tr>
<tr>
<td>[8,10,0]</td>
<td>[8,10,0]</td>
<td></td>
</tr>
</tbody>
</table>

\[\uparrow\]

<table>
<thead>
<tr>
<th>P</th>
<th>A</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;inf,1,P&gt;</td>
<td>&lt;inf,1,P&gt;</td>
<td>&lt;inf,2,A&gt;</td>
</tr>
<tr>
<td>&lt;inf,4,P&gt;</td>
<td>&lt;inf,3,A&gt;</td>
<td>&lt;inf,10,A&gt;</td>
</tr>
<tr>
<td>&lt;inf,8,P&gt;</td>
<td>[0,10,0]</td>
<td></td>
</tr>
</tbody>
</table>

\[\uparrow\]
### P Commits Some Early Writes

<table>
<thead>
<tr>
<th>P</th>
<th>A</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;1,1,P&gt;</td>
<td>&lt;inf,1,P&gt;</td>
<td>&lt;inf,1,B&gt;</td>
</tr>
<tr>
<td>&lt;2,2,A&gt;</td>
<td>&lt;inf,2,A&gt;</td>
<td>&lt;inf,5,B&gt;</td>
</tr>
<tr>
<td>&lt;3,3,A&gt;</td>
<td>&lt;inf,3,A&gt;</td>
<td>&lt;inf,9,B&gt;</td>
</tr>
<tr>
<td>&lt;inf,4,P&gt;</td>
<td>&lt;inf,4,P&gt;</td>
<td></td>
</tr>
<tr>
<td>&lt;inf,8,P&gt;</td>
<td>&lt;inf,8,P&gt;</td>
<td>[0,0,9]</td>
</tr>
<tr>
<td>&lt;inf,10,A&gt;</td>
<td>&lt;inf,10,A&gt;</td>
<td></td>
</tr>
</tbody>
</table>

\[8,10,0] \quad \quad [8,10,0]

### P and B Do Anti-Entropy Exchange

<table>
<thead>
<tr>
<th>P</th>
<th>A</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;1,1,P&gt;</td>
<td>&lt;inf,1,P&gt;</td>
<td>&lt;1,1,P&gt;</td>
</tr>
<tr>
<td>&lt;2,2,A&gt;</td>
<td>&lt;inf,2,A&gt;</td>
<td>&lt;2,2,A&gt;</td>
</tr>
<tr>
<td>&lt;3,3,A&gt;</td>
<td>&lt;inf,3,A&gt;</td>
<td>&lt;3,3,A&gt;</td>
</tr>
<tr>
<td>&lt;inf,1,B&gt;</td>
<td>&lt;inf,1,B&gt;</td>
<td>&lt;inf,1,B&gt;</td>
</tr>
<tr>
<td>&lt;inf,4,P&gt;</td>
<td>&lt;inf,8,P&gt;</td>
<td>&lt;inf,4,P&gt;</td>
</tr>
<tr>
<td>&lt;inf,5,B&gt;</td>
<td>&lt;inf,10,A&gt;</td>
<td>&lt;inf,5,B&gt;</td>
</tr>
<tr>
<td>&lt;inf,8,P&gt;</td>
<td></td>
<td>&lt;inf,8,P&gt;</td>
</tr>
<tr>
<td>&lt;inf,9,B&gt;</td>
<td>[8,10,0]</td>
<td>&lt;inf,9,B&gt;</td>
</tr>
<tr>
<td>&lt;inf,10,A&gt;</td>
<td></td>
<td>&lt;inf,10,A&gt;</td>
</tr>
</tbody>
</table>

\[8,10,9] \quad \quad [8,10,9]

\[8,10,9] \quad [0,0,9]
### P Commits More Writes

<table>
<thead>
<tr>
<th>P</th>
<th>P</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;1,1,P&gt;</td>
<td>&lt;1,1,P&gt;</td>
</tr>
<tr>
<td>&lt;2,2,A&gt;</td>
<td>&lt;2,2,A&gt;</td>
</tr>
<tr>
<td>&lt;3,3,A&gt;</td>
<td>&lt;3,3,A&gt;</td>
</tr>
<tr>
<td>&lt;inf,1,B&gt;</td>
<td>&lt;4,1,B&gt;</td>
</tr>
<tr>
<td>&lt;inf,4,P&gt;</td>
<td>&lt;5,4,P&gt;</td>
</tr>
<tr>
<td>&lt;inf,5,B&gt;</td>
<td>&lt;6,5,B&gt;</td>
</tr>
<tr>
<td>&lt;inf,8,P&gt;</td>
<td>&lt;7,8,P&gt;</td>
</tr>
<tr>
<td>&lt;inf,9,B&gt;</td>
<td>&lt;inf,9,B&gt;</td>
</tr>
<tr>
<td>&lt;inf,10,A&gt;</td>
<td>&lt;inf,10,A&gt;</td>
</tr>
<tr>
<td>[8,10,9]</td>
<td>[8,10,9]</td>
</tr>
</tbody>
</table>

### 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

• These write dependencies are crucial in eliminating unnecessary conflicts
  - If file-level detection was used, all updates would conflict with each other

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

- When client move around and connects to different replicas, strange things can happen
  - Updates you just made are missing
  - Database goes back in time
  - Etc.

- Design choice:
  - Insist on stricter consistency
  - Enforce some “session” guarantees

- SGs ensured by client, not by distribution mechanism

Read Your Writes

- Every read in a session should see all previous writes in that session
Monotonic Reads and Writes

- A later read should never be missing an update present in an earlier read
- Same for writes

 Writes Follow Reads

- If a write $W$ followed a read $R$ at a server $X$, then at all other servers
  - If $W$ is in $Y$'s database then any writes relevant to $R$ are also there
Supporting Session Guarantees

- Responsibility of “session manager”, not servers!

- Two sets:
  - Read-set: set of writes that are relevant to session reads
  - Write-set: set of writes performed in session

- Causal ordering of writes
  - Use Lamport clocks

Practical Byzantine Fault Tolerance

Only a high-level summary
### The Problem

- Ensure correct operation of a state machine in the face of arbitrary failures

- Limitations:
  - no more than $f$ failures, where $n > 3f$
  - messages can’t be indefinitely delayed

### Basic Approach

- Client sends request to primary

- Primary multicasts request to all backups

- Replicas execute request and send reply to client

- Client waits for $f+1$ replies that agree

*Challenge: make sure replicas see requests in order*
Algorithm Components

- Normal case operation
- View changes
- Garbage collection
- Recovery

Normal Case

- When primary receives request, it starts 3-phase protocol
- **pre-prepare**: accepts request only if valid
- **prepare**: multicasts *prepare* message and, if 2f *prepare* messages from other replicas agree, multicasts *commit* message
- **commit**: commit if 2f+1 agree on commit
### View Changes

- Changes primary
- Required when primary malfunctioning

### Communication Optimizations

- Send only one full reply: rest send digests
- Optimistic execution: execute prepared requests
- Read-only operations: multicast from client, and executed in current state
<table>
<thead>
<tr>
<th>Most Surprising Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>• Very little performance loss!</td>
</tr>
</tbody>
</table>