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

1

Agenda

- Introduction and overview
- Data replication and eventual consistency
- Bayou: beyond eventual consistency
- Practical BFT: fault tolerance

2

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

3

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)

4

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

5

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

6

ACID vs BASE

ACID

- Strong consistency for transactions highest priority
- Availability less important
- Pessimistic
- Rigorous analysis
- Complex mechanisms

BASE

- Availability and scaling highest priorities
- Weak consistency
- Optimistic
- Best effort
- Simple and fast

7

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

8

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

9

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

10

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

11

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

12

Data Replication and Eventual Consistency

13

Replication

- Why replication?
  - Volume of requests
  - Proximity
  - Availability
- Challenge of replication: consistency

14

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

15

Eventual Consistency

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

16

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

17

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

18

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

19

Bayou

20

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

21

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?

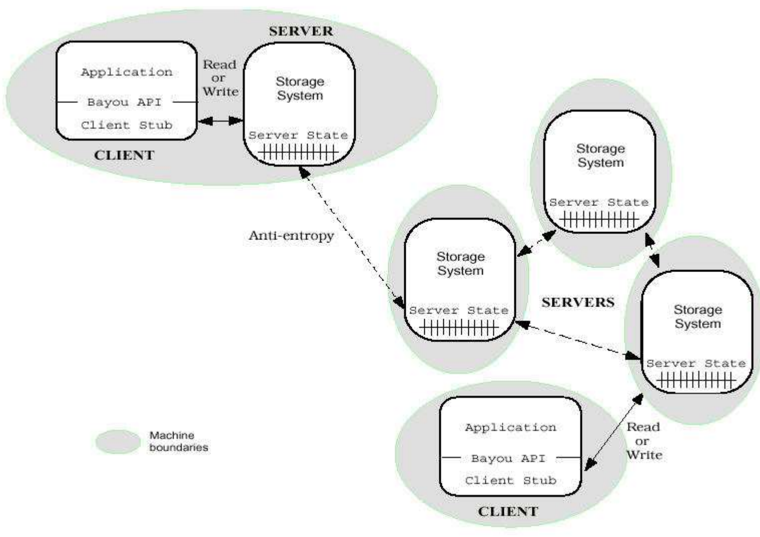
22

Bayou

- System developed at PARC in the mid-90's
- First coherent attempt to fully address the problem of disconnected operation
- Several different components

23

Bayou Architecture



24

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?

25

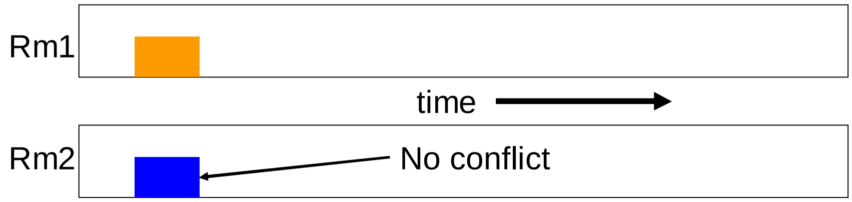
Conflict Resolution

- 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"

26

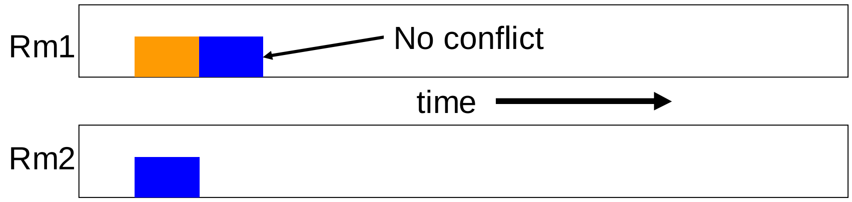
Meeting room scheduler

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!



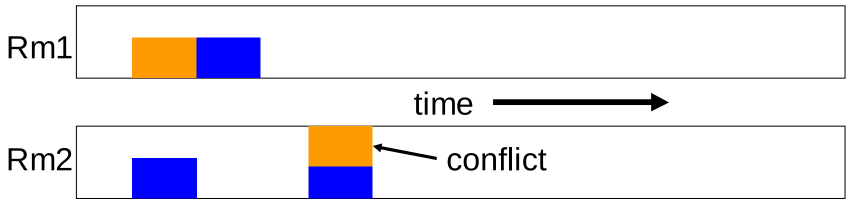
27

Meeting Room Scheduler



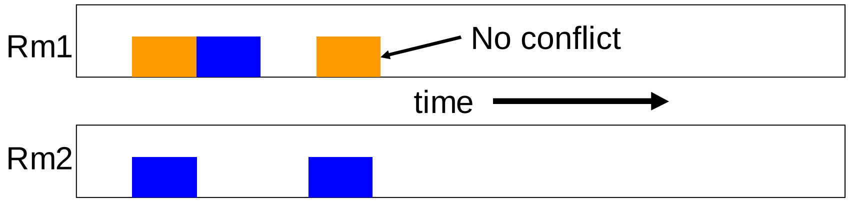
28

Meeting Room Scheduler



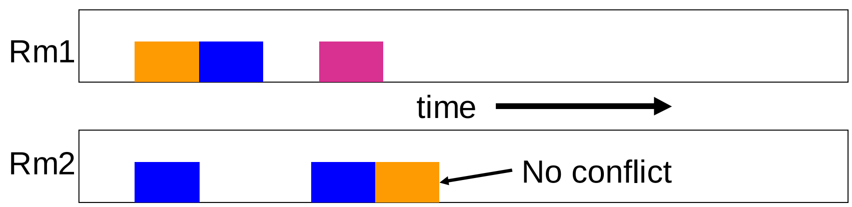
29

Meeting Room Scheduler



30

Meeting Room Scheduler



31

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

32

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

33

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

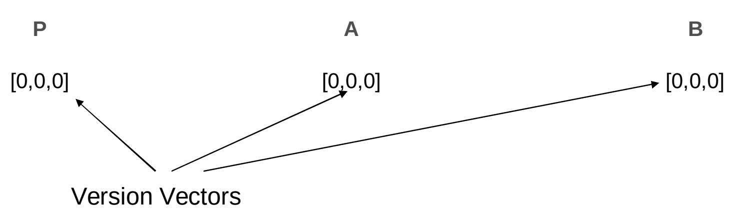
34

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.)

35

Example with Three Servers



36

All Servers Write Independently

P	A	B
<inf,1,P>	<inf,2,A>	<inf,1,B>
<inf,4,P>	<inf,3,A>	<inf,5,B>
<inf,8,P>	<inf,10,A>	<inf,9,B>
[8,0,0]	[0,10,0]	[0,0,9]

37

P and A Do Anti-Entropy Exchange

P	A	B
<inf,1,P>	<inf,1,P>	<inf,1,B>
<inf,2,A>	<inf,2,A>	<inf,5,B>
<inf,3,A>	<inf,3,A>	<inf,9,B>
<inf,4,P>	<inf,4,P>	
<inf,8,P>	<inf,8,P>	[0,0,9]
<inf,10,A>	<inf,10,A>	
[8,10,0]	[8,10,0]	
↑	↑	
<inf,1,P>	<inf,2,A>	
<inf,4,P>	<inf,3,A>	
<inf,8,P>	<inf,10,A>	
[8,0,0]	[0,10,0]	

38

P Commits Some Early Writes

P	A	B
<1,1,P>	<inf,1,P>	<inf,1,B>
<2,2,A>	<inf,2,A>	<inf,5,B>
<3,3,A>	<inf,3,A>	<inf,9,B>
<inf,4,P>	<inf,4,P>	
<inf,8,P>	<inf,8,P>	[0,0,9]
<inf,10,A>	<inf,10,A>	
[8,10,0]	[8,10,0]	
↑		
<inf,1,P>		
<inf,2,A>		
<inf,3,A>		
<inf,4,P>		
<inf,8,P>		
<inf,10,A>		
[8,10,0]		

39

P and B Do Anti-Entropy Exchange

P	A	B
<1,1,P>	<inf,1,P>	<1,1,P>
<2,2,A>	<inf,2,A>	<2,2,A>
<3,3,A>	<inf,3,A>	<3,3,A>
<inf,1,B>	<inf,4,P>	<inf,1,B>
<inf,4,P>	<inf,8,P>	<inf,4,P>
<inf,5,B>	<inf,10,A>	<inf,5,B>
<inf,8,P>		<inf,8,P>
<inf,9,B>	[8,10,0]	<inf,9,B>
<inf,10,A>		<inf,10,A>
[8,10,9]		[8,10,9]
↑		↑
<1,1,P>		<inf,1,B>
<2,2,A>		<inf,5,B>
<3,3,A>		<inf,9,B>
<inf,4,P>		
<inf,8,P>		[0,0,9]
<inf,10,A>		
[8,10,0]		

40

P Commits More Writes

P		P
<1,1,P>		<1,1,P>
<2,2,A>		<2,2,A>
<3,3,A>		<3,3,A>
<inf,1,B>		<4,1,B>
<inf,4,P>	→	<5,4,P>
<inf,5,B>		<6,5,B>
<inf,8,P>		<7,8,P>
<inf,9,B>		<inf,9,B>
<inf,10,A>		<inf,10,A>
[8,10,9]		[8,10,9]

41

Bayou Writes

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

42

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

43

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

44

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

45

Read Your Writes

- Every read in a session should see all previous writes in that session

46

Monotonic Reads and Writes

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

47

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

48

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

49

Practical Byzantine Fault Tolerance

Only a high-level summary

50

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

51

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

52

Algorithm Components

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

53

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

54

View Changes

- Changes primary
- Required when primary malfunctioning

55

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

56

Most Surprising Result

- Very little performance loss!

57