Data Management in the Cloud, Lecture 3

SCALABLE CONSISTENCY AND TRANSACTION MODELS

THANKS TO M. GROSSNIKLAUS

1

Sharding and Replication

- Sharding (Partitioning)
  - Breaking a database into several collections (*shards*)
  - Each data item (e.g., a document) goes in one shard
- Replication
  - Have multiple copies of a database
  - Each data item lives in several places

Can combine sharding and replication

Why do systems shard and replicate?

2

Issues with Sharing and Replication

Sharding: If an operation needs multiple data items, it might need to access several shards

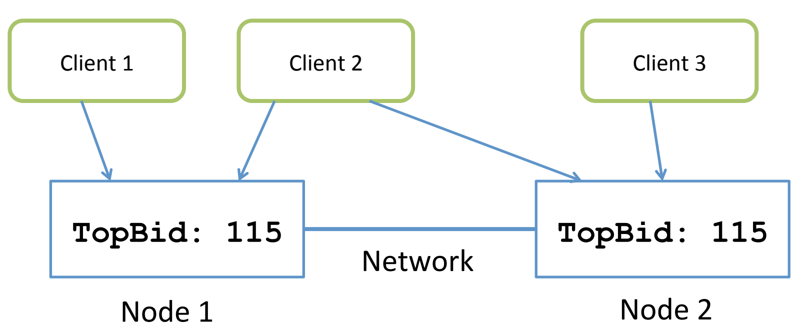
Replication: Trying to keep replicas in sync with each other.

Can you think of others?

Managing Replicas 1

Strategy 1: Write all

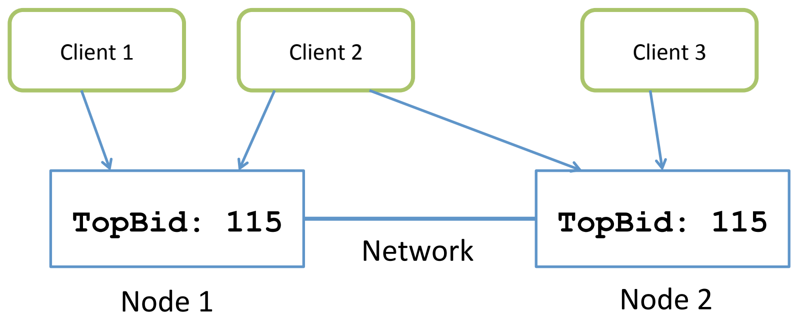
- Write: Update both, synchronously
- Read: Access either



Managing Replicas 2

Strategy 2: Write one

- Write: Update one, propagate update aynschronously
- Read: Access either



5

Brewer's Conjecture

- Three properties that are desirable and expected from real-world shared-data systems
  - **C:** data consistency – you get most recent write or an error
  - **A:** availability – every request gets a (non-error) response
  - **P:** tolerance of network partition
- At *PODC 2000* (Portland, OR), Eric Brewer made the conjecture that only two of these properties can be satisfied by a system at any given time
- Conjecture was formalized and confirmed by MIT researchers Seth Gilbert and Nancy Lynch in 2002
- Now known as the **CAP Theorem**

6

Data Consistency

- Database systems typically implement ACID transactions
  - **A**tomicity: "all or nothing"
  - **C**onsistency: transactions never observe or result in inconsistent data
  - **I**solation: transactions are not aware of concurrent transactions
  - **D**urability: once committed, the state of a transaction is permanent
- Useful in automated business applications
  - banking: at the end of a transaction the sum of money in both accounts is the same as before the transaction
  - online auctions: the last bidder wins the auction
- There are applications that can deal with looser consistency guarantees and periods of inconsistency

7

Availability

- Services are expected to be highly available
  - every request should receive a response
  - if you can read a data item, you can update it
  - it can create real-world problems when a service goes down
- Realistic goal
  - service should be as available as the network it runs on
  - if any instance of a service on the network is available, the service should be available

8

Partition-Tolerance

- A service should continue to perform as expected
  - if some nodes crash
  - if some communication links fail
- One desirable fault tolerance property is resilience to a network partitioning into multiple components
- In cloud computing, node and communication failures are not the exception but everyday events

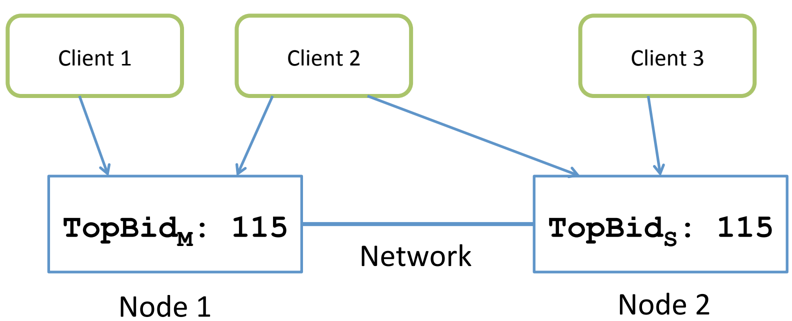
9

Managing Replicas: Activity

Strategy 3: Master-Slave

1. Write: Update master, propagate to slave asynchronously
2. Read Strong: Access master
3. Read Weak: Access either

For each op, does it give consistency or availability in a partition?



10

Problems with CAP (from D. Abadi)

- Asymmetry of CAP properties
  - **C** is a property of the system in general
  - **A** is a property of the system only when there is a partition
- There are not three different choices
  - in practice, **CA** and **CP** are indistinguishable, since **A** is only sacrificed when there is a partition
- Used as an excuse to not bother with consistency
  - "Availability is really important to me, so CAP says I have to get rid of consistency"

Source: Daniel Abadi, Yale University

11

Another Problem to Fix

- Apart from availability in the face of partitions, there are other costs to consistency
- Overhead of synchronization schemes
- Latency
  - if replicas are far apart, can be a long wait to update one
  - but for reliability, you might want replicas in different data centers

Source: Daniel Abadi, Yale University

12

A Cut at Fixing Both Problems

P[A|C]/E[L|C]

- In the case of a partition (**P**), does the system choose availability (**A**) or consistency (**C**)?
- Else (**E**), does the system choose latency (**L**) or consistency (**C**)?

- PA/EL
  - Dynamo, SimpleDB, Cassandra, Riptano, CouchDB, Cloudant
- PC/EC
  - ACID compliant database systems
- PA/EC
  - GenieDB
- PC/EL
  - Existence is debatable

Source: Daniel Abadi, Yale University

13

A Case for P*/EC

- Increased push for horizontally scalable transactional database systems
  - cloud computing
  - distributed applications
  - desire to deploy applications on cheap, commodity hardware
- Vast majority of currently available horizontally scalable systems are P*/EL
  - developed by engineers at Google, Facebook, Yahoo, Amazon, etc.
  - these engineers can handle reduced consistency, but it's really hard, and there needs to be an option for the rest of us
- Also
  - distributed concurrency control and commit protocols are expensive
  - once consistency is gone, atomicity usually goes next → NoSQL

Source: Daniel Abadi, Yale University

14

Key Problems to Overcome

- High availability is critical, replication must be a first-class citizen
- Today's systems generally act, then replicate
  - complicates semantics of sending read queries to replicas
  - need confirmation from replica before commit (increased latency) if you want durability and high availability
  - In-progress transactions must be aborted upon a master failure
- Want system that replicates then acts
- Distributed concurrency control and commit are expensive, want to get rid of them both

Source: Daniel Abadi, Yale University 15

Key Idea

- Instead of weakening ACID, strengthen it
- Challenges
  - guaranteeing equivalence to *some* serial order makes active replication difficult
  - running the same set of transactions on two different replicas might cause replicas to diverge
- Disallow any nondeterministic behavior
- Disallow aborts caused by DBMS
  - disallow deadlock (restrict locking order)
  - distributed commit much easier if there are no aborts

Source: Daniel Abadi, Yale University 16

Consequences of Determinism

- Replicas produce the same output, given the same input
  - facilitates active replication
- Only initial input needs to be logged, state at failure can be reconstructed from this input log (or from a replica)
- Active distributed transactions not aborted upon node failure
  - greatly reduces (or eliminates) cost of distributed commit
  - don't have to worry about nodes failing during commit protocol
  - don't have to worry about effects of transaction making it to disk before promising to commit transaction
  - any node that potentially can deterministically abort the transaction need only send one message
  - this message can be sent in the middle of the transaction, as soon as it knows it will commit

Source: Daniel Abadi, Yale University

17

Strong vs. Weak Consistency

- Strong consistency
  - after an update is committed, each subsequent access will return the updated value
- Weak consistency
  - the systems does not guarantee that subsequent accesses will return the updated value
  - a number of conditions might need to be met before the updated value is returned
  - **inconsistency window:** period between update and the point in time when every access is guaranteed to return the updated value

Based on: "Eventually Consistent" by W. Vogels, 2008

18

Eventual Consistency

- Specific form of weak consistency
- "If no new updates are made, eventually all accesses will return the last updated values"
- In the absence of failures, the maximum size of the inconsistency window can be determined based on
  - communication delays
  - system load
  - number of replicas
  - …
- Not a new esoteric idea!
  - Domain Name System (DNS) uses eventual consistency for updates
  - RDBMS use eventual consistency for asynchronous replication or backup (e.g. log shipping)

Based on: "Eventually Consistent" by W. Vogels, 2008

19

However …

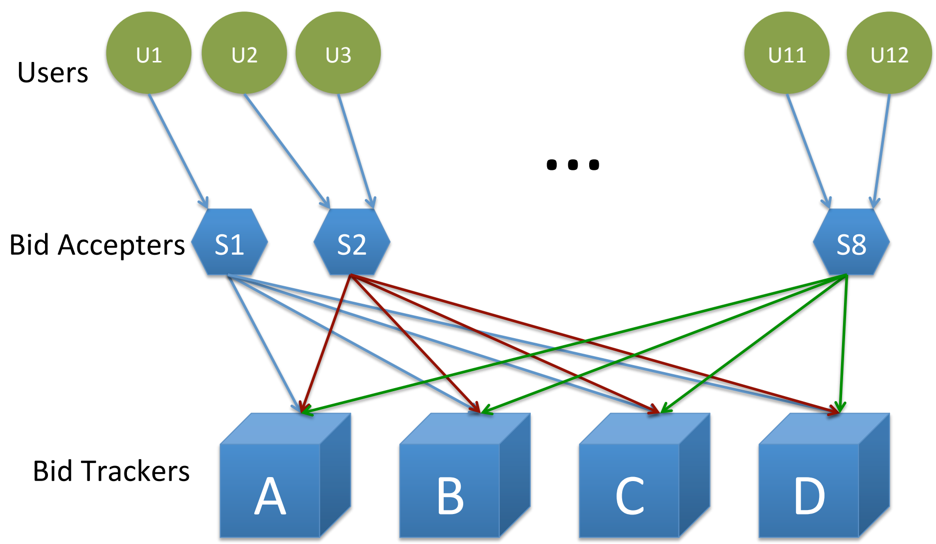
Eventual Consistency and Perpetual Inconsistency are not mutually exclusive!

See Doug Terry paper on consistency of baseball scores.

- Consistent prefix
- Monotonic reads

20

Eventual-Consistency Activity



21

Items for Auction

Boots

Tricycle

Zune

22

Models of Eventual Consistency

- Causal Consistency
  - if A communicated to B that it has updated a value, a subsequent access by B will return the updated value, and a write is guaranteed to supersede a causally earlier write
  - access by C that has no causal relationship to A is subject to normal eventual consistency rules
- Read-your-writes Consistency
  - special case of the causal consistency model
  - after updating a value, a process will always read the updated value and never see an older value
- Session Consistency
  - practical case of read-your-writes consistency
  - data is accessed in a session where read-your-writes is guaranteed
  - guarantees do not span over sessions

Based on: "Eventually Consistent" by W. Vogels, 2008

23

Models of Eventual Consistency

- Monotonic Read Consistency
  - if a process has seen a particular value, any subsequent access will never return any previous value
- Monotonic Write Consistency
  - system guarantees to serialize the writes of one process
  - systems that do not guarantee this level of consistency are hard to program
- Properties can be combined
  - e.g. monotonic reads plus session-level consistency
  - e.g. monotonic reads plus read-your-own-writes
  - quite a few different scenarios are possible
  - it depends on an application whether it can deal with the consequences

Based on: "Eventually Consistent" by W. Vogels, 2008

24

Configurations

- Definitions
  - **N**: number of nodes that store a replica
  - **W**: number of replicas that need to acknowledge a write operation
  - **R**: number of replicas that are accessed for a read operation
- W+R > N
  - e.g. **synchronous replication** (N=2, W=2, and R=1)
  - write set and read set always overlap
  - strong consistency can be guaranteed through **quorum protocols**
  - risk of reduced availability: in basic quorum protocols, operations fail if fewer than the required number of nodes respond, due to node failure
- W+R = N
  - e.g. **asynchronous replication** (N=2, W=1, and R=1)
  - strong consistency cannot be guaranteed

Based on: "Eventually Consistent" by W. Vogels, 2008

25

Configurations

- R=1, W=N
  - optimized for **read access**: single read will return a value
  - write operation involves all nodes and risks not succeeding
- R=N, W=1
  - optimized for **write access**: write operation involves only one node and relies on asynchronous updates to other replicas
  - read operation involves all nodes and returns "latest" value
  - durability is not guaranteed in presence of failures
- W < (N+1)/2
  - risk of conflicting writes
- W+R <= N
  - **weak/eventual consistency**

Based on: "Eventually Consistent" by W. Vogels, 2008

26

BASE

- **B**asically **A**vailable, **S**oft state, **E**ventually Consistent
- As consistency is achieved eventually, conflicts have to be resolved at some point
  - read repair
  - write repair
  - asynchronous repair
- Conflict resolution is typically based on a global (partial) ordering of operations that (deterministically) guarantees that all replicas resolve conflicts in the same way
  - client-specified timestamps
  - vector clocks

Vector Clocks

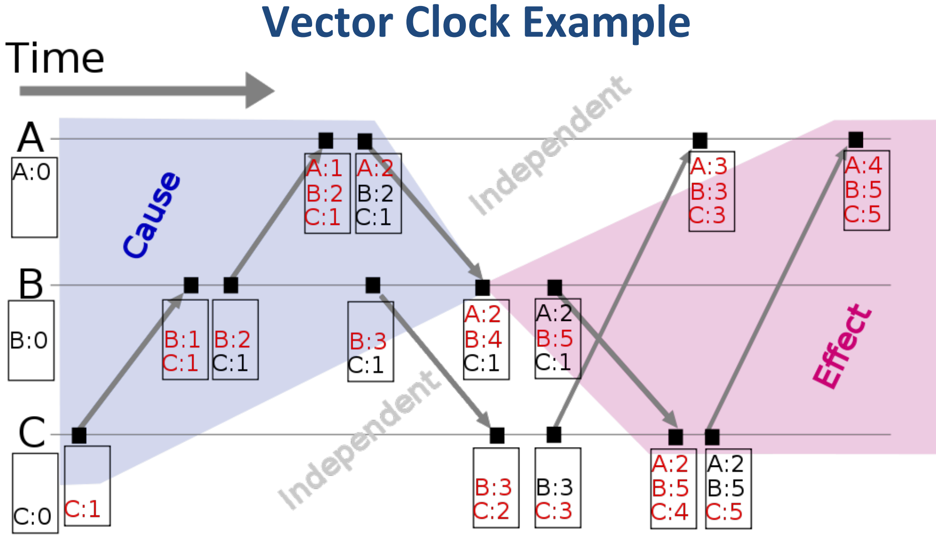
- Generate a partial ordering of events in a distributed system and detecting causality violations
- A **vector clock** of a system of *n* processes is an vector of *n* logical clocks (one clock per process)
  - messages contain the state of the sending process's logical clock
  - a local "smallest possible values" copy of the global vector clock is kept in each process
- Vector clocks algorithm was independently developed by Colin Fidge and Friedemann Mattern in 1988

Update Rules for Vector Clocks

- All clocks are initialized to zero
- A process increments its own logical clock in the vector by one
  - each time it experiences an internal event
  - each time a process prepares to send a message
  - each time a process receives a message
- Each time a process sends a message, it transmits the entire vector clock along with the message being sent
- Each time a process receives a message, it updates each element in its vector by taking the pair-wise maximum of the value in its own vector clock and the value in the vector in the received message

29

Vector Clock Example



Source: Wikipedia (http://www.wikipedia.org)

30

References

- S. Gilbert and N. Lynch: **Brewer's Conjecture and the Feasibility of Consistent, Available and Partition-Tolerant Web Services.** *SIGACT News 33(2), pp. 51-59, 2002.*
- D. J. Abadi: **Consistency Tradeoffs in Modern Distributed Database System Design: CAP is Only Part of the Story.** *Computer, vol. 45, no. 2, pp. 37-42, Feb. 2012*.
- A. Thomson and D. J. Abadi**. The Case for Determinism in Database Systems.** *Proc. VLDB Endow. 3, 1-2 (September 2010).*
- D. Terry. **Replicated Data Consistency Explained Through Baseball.** *Commun. ACM 56, 12, Dec. 2013.*
- W. Vogels: **Eventually Consistent.** *ACM Queue 6(6), pp. 14-19, 2008.*