Eventual Consistency Today: Limitations, Extensions and Beyond

Peter Bailis and Ali Ghodsi, UC Berkeley

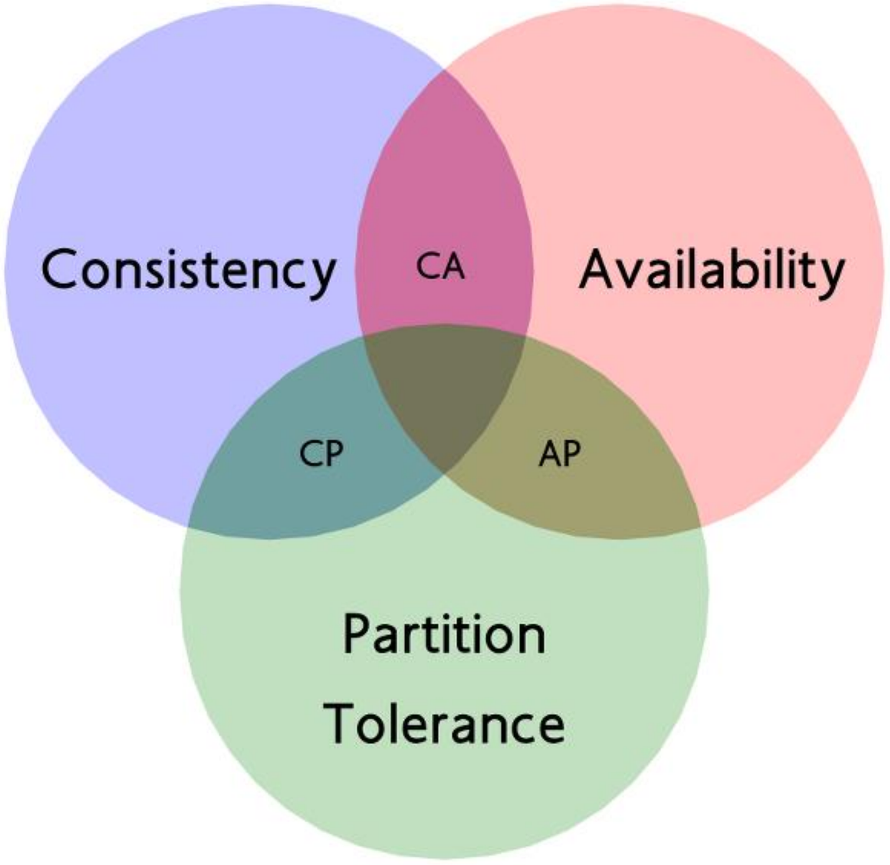
- Nomchin Banga

Outline

- Eventual Consistency: History and Concepts
- How eventual is eventual consistency?
- Programming eventual consistency
- Stronger guarantees than eventual consistency
- Conclusion

Brewer’s CAP Theorem

- Cost of maintaining a single-system image
- Cannot “sacrifice” partition tolerance
- Consistency-Availability trade-off
- Consistency-Latency trade-off



Eventual Consistency

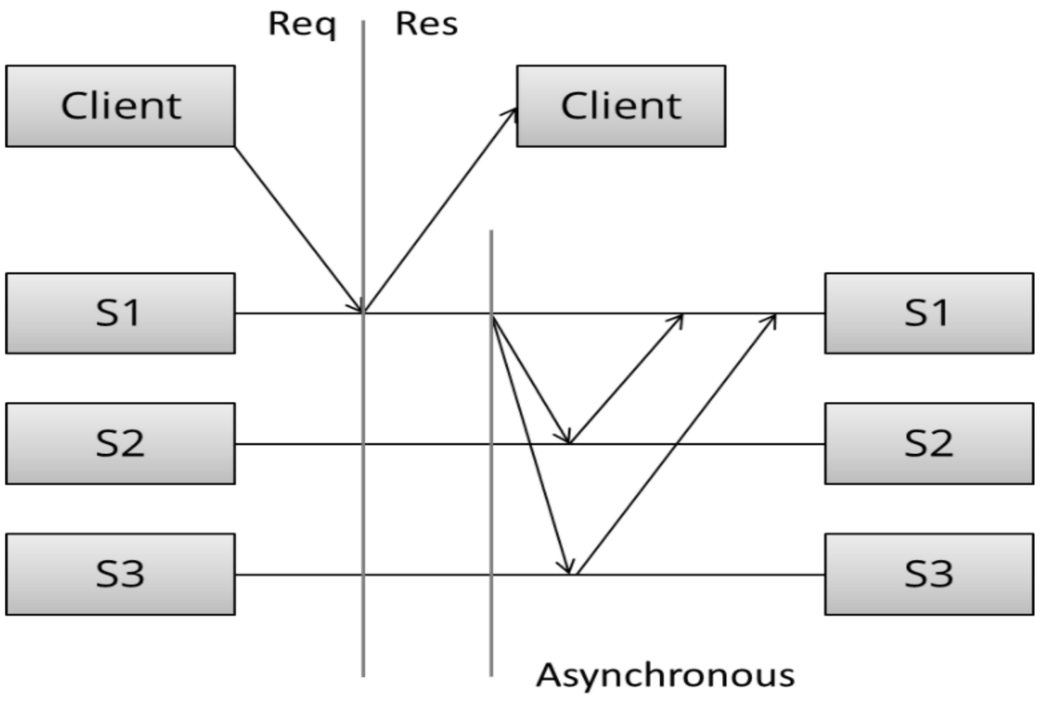
"…changes made to one copy eventually migrate to all. If all update activity stops, after a period of time all replicas of the database will converge to be logically equivalent: each copy of the database will contain, in a predictable order, the same documents; replicas of each document will contain the same fields."

Eventual v/s Strong Consistency

EVENTUAL	STRONG
System can return any data	System will always return correct, consistent and last updated data
Does not specify which value is eventually chosen	Consistency is immediate
"Predictable order" of execution may differ from that of a single system image database	Fixed set of rules for determining order of executions
Window of inconsistency	Single system image

Implementing Eventual Consistency

- *Anti-entropy* – To ensure convergence, replicas must exchange information about which write they have seen



Asynchronous all-to-all broadcast

Implicit Assumptions:

- system partitions eventually heal and converge, OR
- partitioned nodes eventually die

Quantifying Eventual Consistency

- Metrics
  - Time : how long will it take for writes to become visible for reads
  - Version : how many versions old will a given read be

- Mechanisms
  - Measurement : how consistent is my store under current workload
  - Prediction : how consistent will my store be under a given workload and configuration

Benefits of Eventual Consistency

- Easy to implement – no difficult corner cases to handle failed replicas and network partitions
- All operations complete locally – low latency
- Data durability might be at risk – write to multiple nodes
- Rate of anti-entropy determined by system

Safety and Liveness

- *Safety* – nothing bad happens
  - every value that is read was, at some point in time, written to the database
- *Liveness* – all requests eventually receive a response
- Eventual Consistency is purely a liveness property.
  - Replicas agree but there are no guarantees with respect to what happens

Probabilistic Bounded Staleness

- *Expectation* of recency for reads of data items
  - 100ms after a write completes, 99.9% of reads will return the most recent version
  - 85% of reads will return a version that is within two of the most recent
- Degree of inconsistency determined by

Inconsistency Window of Major DDBS

LinkedIn: **13.6 ms**

cassandra: **200 ms**

SimpleDB: **500 ms**

yammer: **202 ms**

amazon web services S3: **12 s**

Eventual Consistency is "good enough"

Designing Eventually Consistent System

- Compensation – way to achieve safety retroactively
- Choosing Eventually Consistent system
  - Benefit of weak consistency
  - Cost of each inconsistency anomaly
  - Rate of anomalies

  Maximize B-CR
- Design for compensation
  - Need for compensation
  - Possible anomalies and the correct “apologies”

Compensation by Design

- Programming for Compensation – error prone

- State-of-the-art : “compensation-free” programming
  - CALM/ACID 2.0 – *Consistency As Logical Monotonicity*
  - CRDTs – Commutative, Replicative Data Types

CALM/ACID 2.0

- **Monotonicity** - programs compute an ever-growing set of facts and do not ever retract the facts they emit
- Monotonic programs provide safety guarantees
- Examples of operations
  - Monotonic : Initializing variables, accumulating set members
  - Non-monotonic : Variable overwrites, set deletion, counter resets

CALM/ACID 2.0

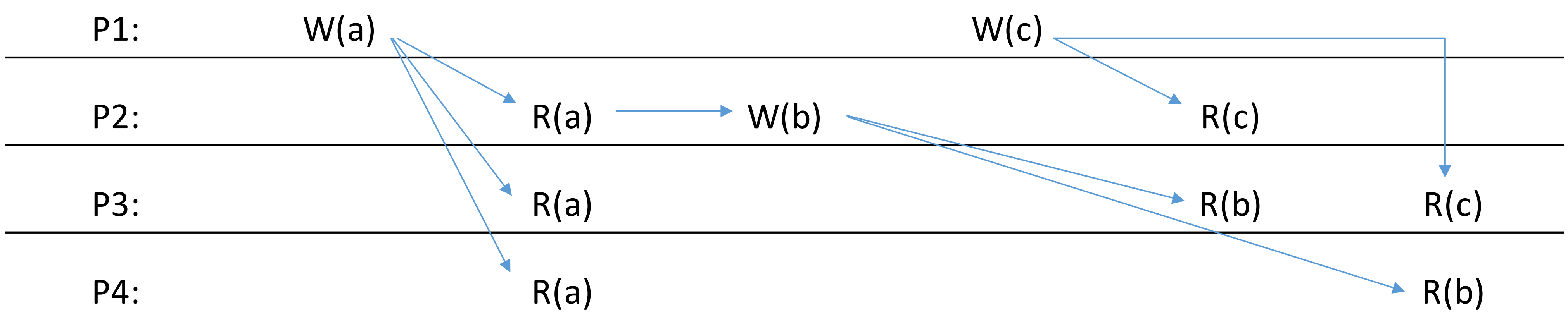
- Programmers can use ACID 2.0 for achieving logical monotonicity
- ACID 2.0 – Associativity, Commutativity, Idempotence, Distributed
- Associativity and *Commutativity* can tolerate message re-ordering in eventual consistency
- *Idempotence* allows at-least-once message delivery, instead of at-most-once

Commutative, Replicated Data Types (CRDT)

- Use CALM and ACID 2.0 within standard data types like graphs
  - Example : increment-only counter replicated on two servers
- Separate data store and application-level consistency
  - “weak” distributed read/write consistency
  - “strong” application consistency – semantic guarantee
- Existing systems that use CRDTs – Statebox, Riak, Bloom language

Stronger than Eventual

- Causal Consistency – guarantees each process's write are seen in order, transitive data dependencies hold



Stronger than Eventual

- Causal consistency
    - Not possible to have a stronger model without violating high availability or high convergence
    - Causality bolted-on top of eventual consistency (safety and liveness decoupled)
    - COPS, Eiger systems – less than 7% overhead for one of Facebook's workload
- Re-architecting distributed databases using ACID properties
    - Transactional atomicity
    - SQL Read Committed and Repeatable Read

Recognizing the Limits

- Inherent cost for choosing high availability and low latency
- Cannot maintain global correctness constraints
  - Ex: Uniqueness requirements
- Cannot guarantee correctness constraints on individual data items
  - Ex: Bank balance should be non-negative

Research Scope

- Re-thinking distributed transaction algorithms to incorporate stronger consistency models like Repeatable Reads
- Rule-based concurrency model for transactions in Cassandra that places a deterministic bound on “predictable order” of transactions
- Use CRDTs as a client-side enhancement in Spark to provide stronger safety guarantees

Thank You!