Fault Tolerant Distributed Systems

Dependable Systems 2014

Lena Herscheid, Dr. Peter Tröger

Distributed Systems – Motivation

RECAP

"Distributed programming is the art of solving the same problem that you can solve on a single computer using multiple computers."

- For Scalability
- For Performance / Throughput
- For **Availability**
  - Build a highly-available or reliable system using unreliable components
- Divide and conquer approach
  - **Partition** data over multiple nodes
  - **Replicate** data for fault tolerance or shorter response time

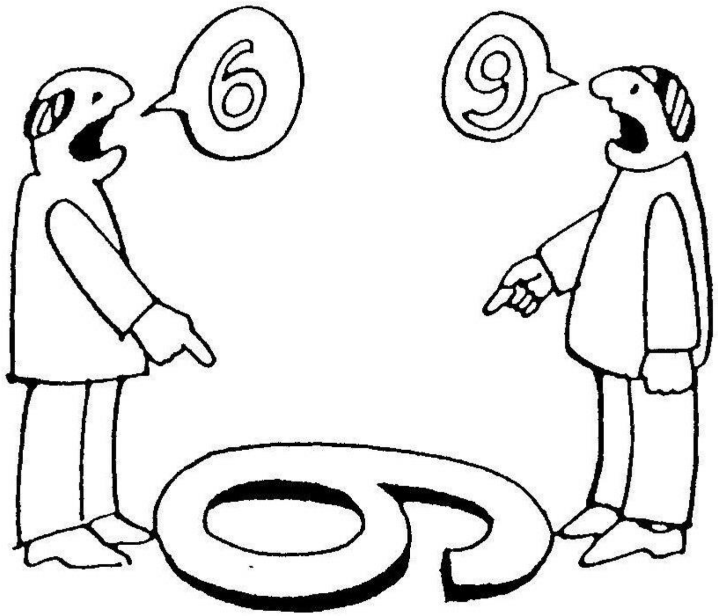
Distributed Systems – Abstractions

RECAP

"You can't tolerate faults you haven't considered."

- **Timing** assumptions
  - Synchronous
    - known upper bound on message delay
    - each processor has an accurate clock
  - Asynchronous
    - processes execute at independent rates
    - message delay can be unbounded
- **Failure model**: Crash, Byzantine, …
- **Failure detectors**: strong/weakly accurate, strong/weakly complete
- **Consistency model**
  - strong: the visibility of updates is equivalent to a non-replicated system
  - weak: client-centric, causal, eventual…

Consistency

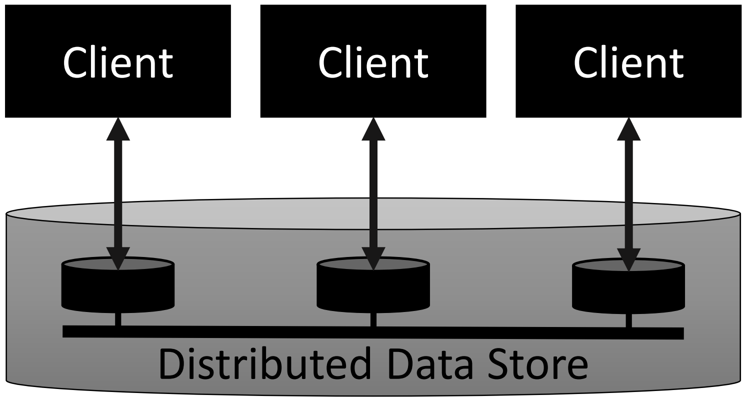


Vogels, Werner. "Eventually consistent." *Communications of the ACM* 52.1 (2009): 40-44.

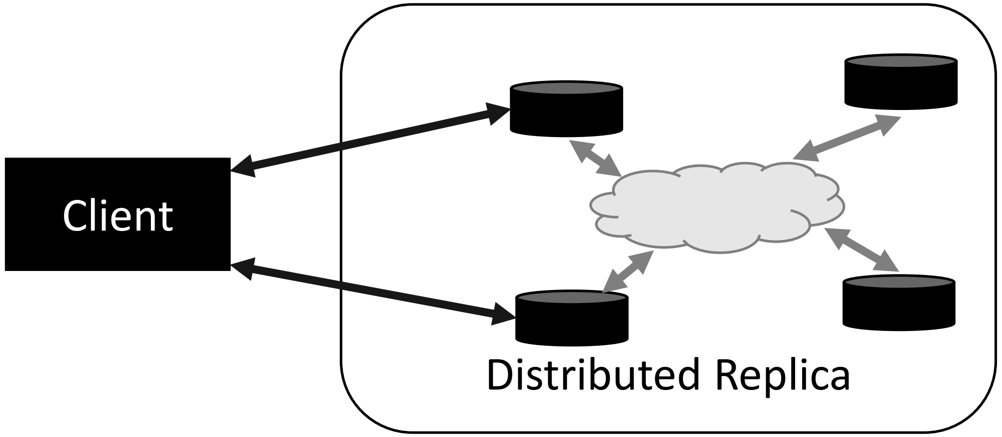
Terry, Doug. "Replicated data consistency explained through baseball.“ *Communications of the ACM* 56.12 (2013): 82-89.

Consistency Models

- Set of guarantees that describes **constraints** on the outcome of sequences of interleaved and simultaneous operations
  - *"A system is in a consistent state, if all replicas are identical and the ordering guarantees of the specific consistency model are not violated."*
- "Contract" between the programmer and the storage system
  - If the programmer obeys certain rules, the results will be consistent
  - An *abstraction*: details are application-specific and require inside knowledge

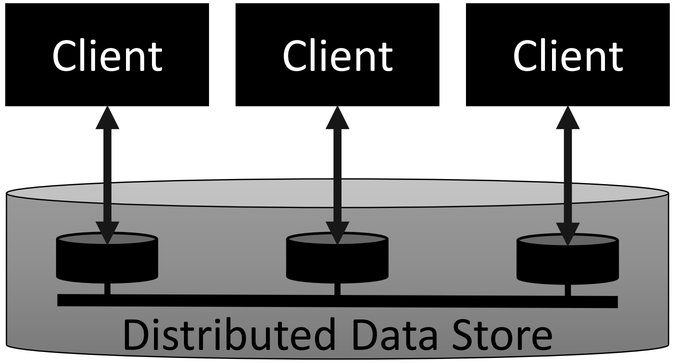


Data-Centric



Client-Centric

Data-Centric Consistency



- Assume no explicit synchronization operations
- Consistency needs depend on client and data

more constrained

- **Strict**: absolute global time
- **Linearizable / Atomic**: all shared accesses seen in the same order + nonunique global timestamp
- **Sequential**: all shared accesses seen in the same order

strong

- **Causal**: preserve causal order
- **FIFO**: preserve per-process order

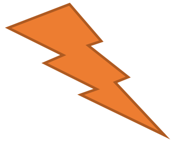
weak

Strict Consistency

- Any read on a data item returns a value corresponding to the most recent write
- Needs global clock → theoretically impossible in asynchronous systems
  - (not even guaranteed by programming languages!)

P1	W(x) → a		
P2			R(x) → a

P1	W(x) → a		
P2		R(x) → NIL	R(x) → a



Sequential Consistency

- The result of any execution is the same as some sequential order
  - Implementation: Use Lamport clocks → all concurrent ops can be re-ordered
- The operations of each individual process appear in this sequence in the order specified by its program

P1	W(x) → a				
P2		W(x) → b			
P3			R(x) → b		R(x) → a
P4				R(x) → b	R(x) → a

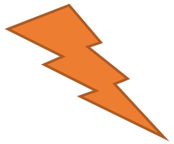
P2	**P3**	**P4**	**P1**	**P3**	**P4**
W(x) → b	R(x) → b	R(x) → b	W(x) → a	R(x) → a	R(x) → a

Sequential order

Linearizable / Atomic Consistency

- Sequentially consistent, and
- Ordered timestamps: if timestamp(x) < timestamp(y), x must occur before y in the sequence

P1	W(x) → a				
P2		W(x) → b			
P3			R(x) → b		R(x) → a
P4				R(x) → b	R(x) → a



P2	**P3**	**P4**	**P1**	**P3**	**P4**
W(x) → b	R(x) → b	R(x) → b	W(x) → a	R(x) → a	R(x) → a

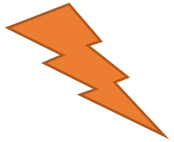
Invalid sequential order!

Causal Consistency

- **Causally related** statements need to execute in the same order for all processors
  - The result of one statement affects the other: W(x) is causally related with R(x)

P1	W(x) → b		
P2	W(x) → a		
P3		R(x) → a	R(x) → b
P4		R(x) → b	R(x) → a

P1		R(x) → a	W(x) → b		
P2	W(x) → a				
P3				R(x) → a	R(x) → b
P4				R(x) → b	R(x) → a



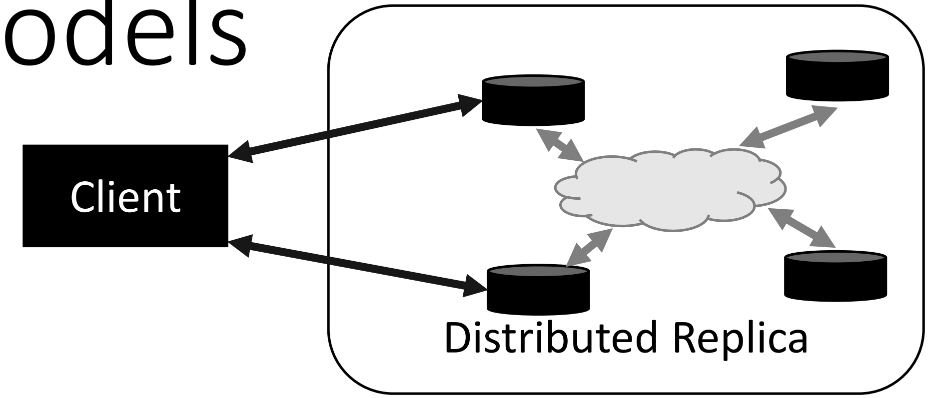
FIFO Consistency

- Writes done by a single process are seen by all other processes in the order in which they were issued

P1		R(x) → a	W(x) → b		
P2	W(x) → a				
P3				R(x) → a	R(x) → b
P4				R(x) → b	R(x) → a

P1		W(x) → a	W(x) → b		
P2	W(x) → a				
P3				R(x) → a	R(x) → b
P4				R(x) → b	R(x) → a

Client-Centric Consistency Models



- System-wide consistency is hard
  - Focus on specific clients' view on the system
- Stronger variants of eventual consistency
  - **Monotonic Reads**: If a process has seen a value of x at time t, it will never see an older version of x at t'>t
  - **Monotonic Writes**: If a process updates a data item, all preceding updates by the same process will be performed first
  - **Read Your Writes (RYW) / Immediate Consistency**: Once a data item has been updated, any read will return the updated value
  - **Writes Follow Reads**: A write following a read by the same process is guaranteed to take place on the same or a newer value
  - **Bounded Staleness**: Reads return data with maximum age t
  - **Consistent Prefix / Snapshot Isolation**: Only "snapshot" data which existed at the master at some point in time is returned

Tuning Consistency

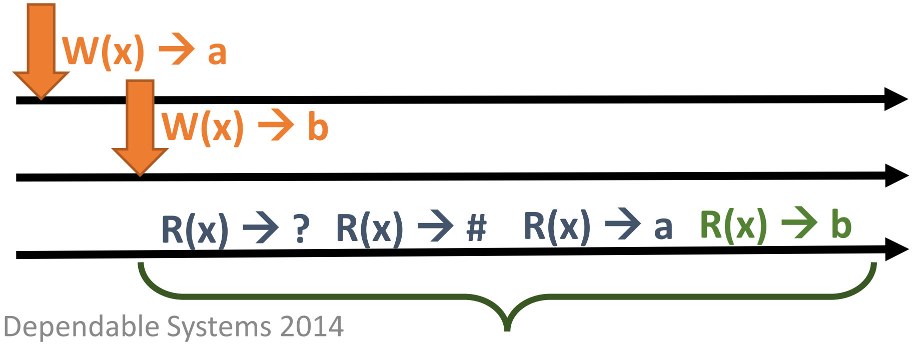
N: number of replicas
W: number of nodes that must acknowledge a write (W <= N)
R: number of nodes that must agree for a read (R <= N)

- Number of tolerated faults grows with N-R and N-W
- **W << N** → High write availability
- **R << N** → High read availability
- **W = N** → Fully consistent reads and writes
- **R+W > N** → Immediately (RYW) consistent
  - E.g., N = 3, R = 2, W = 2

Eventual Consistency

Why was the Amazon engineer's wedding cancelled?
When the priest asked the question, his answer was eventually consistent.

- State of the art in NoSQL databases
- Update one replica → *eventually* all replicas will be updated
  - Only liveness, no safety
- **Optimistic replication**: dealing with eventual consistency
  - Eventual consistency comes at the cost of additional client logic
  - Allow replicas to drift apart
  - Tentative scheduling: sites may repeatedly change and re-order operations to eventually arrive at the same result
  - Conflict resolution



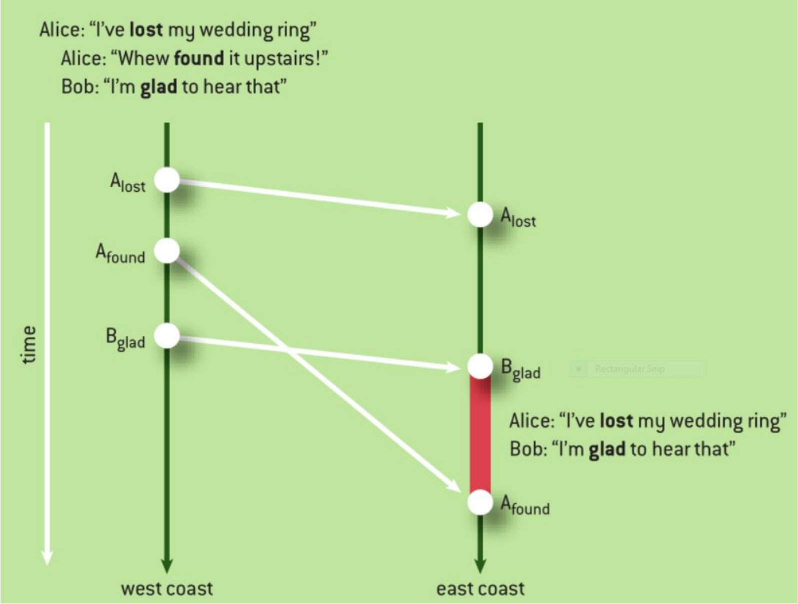
Optimistic Replication

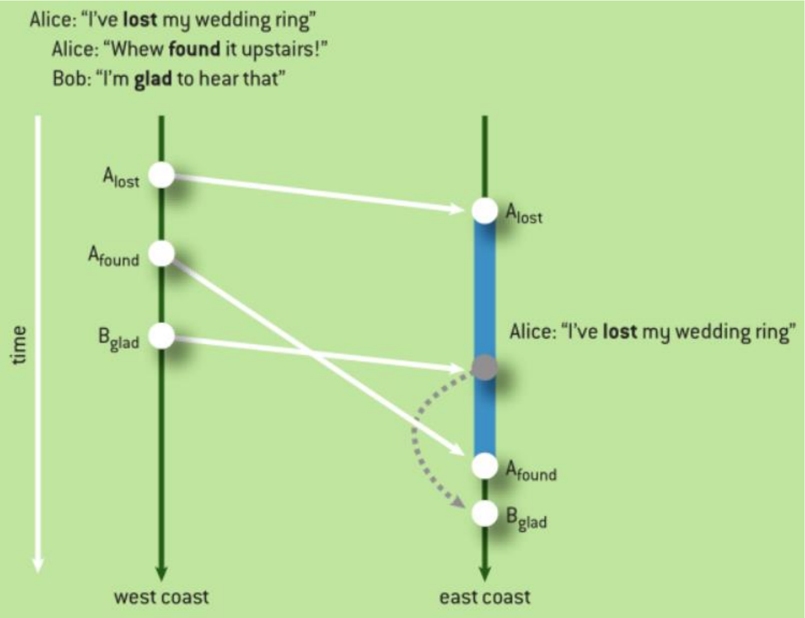
Optimistic Replication (Yasushi Saito and Marc Shapiro)
http://research.microsoft.com/apps/pubs/default.aspx?id=66979

Choice	Description	Effects
Number of masters	Which replicas can submit updates?	Defines the system's basic complexity, availability and efficiency.
Operation definition	What kinds of operations are supported, and to what degree is a system aware of operation semantics?	
Scheduling	How does a system order operations?	Defines the system's ability to handle concurrent operations.
Conflict management	How does a system define and handle conflicts?	
Operation propagation strategy	How are operations exchanged between sites?	Defines networking efficency and the speed of replica convergence
Consistency guarantees	What does a system guarantee about the divergence of replica state?	Defines the transient quality of replica state.

Causal Consistency

- Stronger than eventual consistency
- Rules for causality (similar to happened before)
  1. Same **thread of execution**
  2. **Reads-from** (W(x) → R(x))
  3. **Transitivity**
     (a → b and b → c imply a → c)
- Causality information attached to each write
- Client-side library tracks causality rules
- In geo-replicated systems: delay remote writes, until all causally previous writes have occurred





Quiz

1. Are there systems in which sequential consistency implies strict consistency?
2. How many faults can be tolerated in a distributed storage system with N = R = W?
3. What consistency guarantees can you make in a system with N=3 nodes, where R=2 nodes are required for reads, and R=2 nodes are required for writes?

Trade-Offs

Fox, Armando, and Eric A. Brewer. "Harvest, yield, and scalable tolerant systems." *Hot Topics in Operating Systems, 1999. Proceedings of the Seventh Workshop on*. IEEE, 1999.

Brewer, Eric A. "Lessons from giant-scale services." *Internet Computing, IEEE*5.4 (2001): 46-55.

Pritchett, Dan. "Base: An acid alternative." *Queue* 6.3 (2008): 48-55.

Brewer, Eric. "CAP twelve years later: How the" rules" have changed."*Computer* 45.2 (2012): 23-29.

Abadi, Daniel J. "Consistency tradeoffs in modern distributed database system design." *Computer-IEEE Computer Magazine* 45.2 (2012): 37.

Harvest and Yield

- **Yield**: probability of completing, *queries completed / queries offered*
  - Can be measured as availability (“4 nines”)
  - Similar to uptime, but closer to user experience
- **Harvest**: completeness of the answer, *data available / complete data*
  - Sometimes, imperfect query results or answers can be tolerated
- Trading Harvest for Yield in the presence of faults
  - Stop answering requests or reply incompletely?
  - Replicate high-priority data → reduced probability of losing it
  - Trading response time for harvest: intelligent degradation

DQ Analysis (Brewer. Lessons from giant-scale services. 2001)

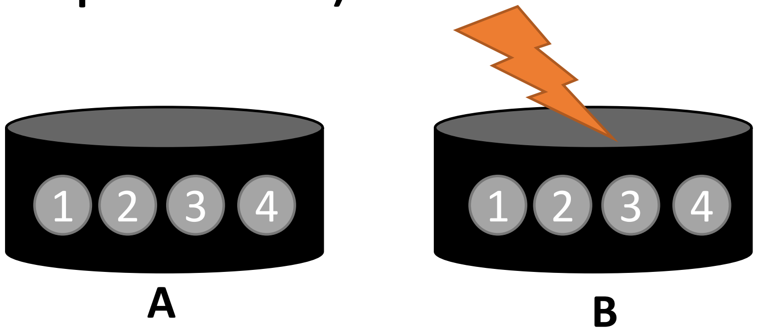
- **DQ value:** *data per query (D) × queries per second (Q) → constant*
- Intuition: There's always a physical bottleneck tied to data movement
  - Bandwidth
  - I/O
- DQ decreases linearly with the number of failed nodes
- Controlled **graceful degradation** to avoid overload in case of faults
  - Limit capacity / Admission control → decrease Q to maintain D
  - Decision: how should saturation influence availability (yield)/ QoS (harvest)?

Replication vs Partition

Harvest: data available / complete data
Yield: queries completed / queries offered
DQ: **D**ata per query * **Q**ueries per second

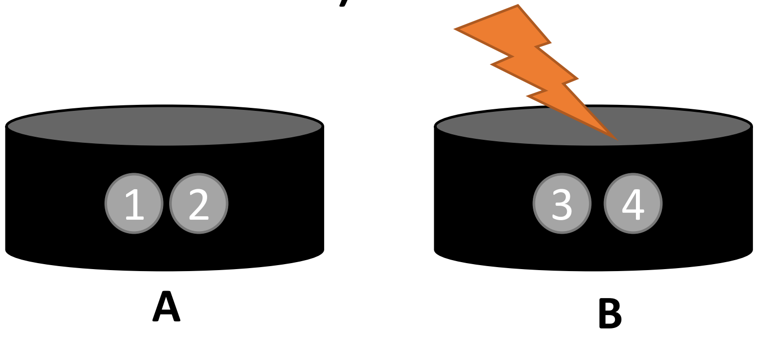
Replication

- Reduced Q, constant D
- Reduced Yield
- Can A handle the additional workload? (Load redirection problem)



Partition

- Reduced D, constant Q
- Reduced Harvest
- Saved storage space does not affect DQ bottleneck (DQ still constant)



Brewer. Lessons from giant-scale services.

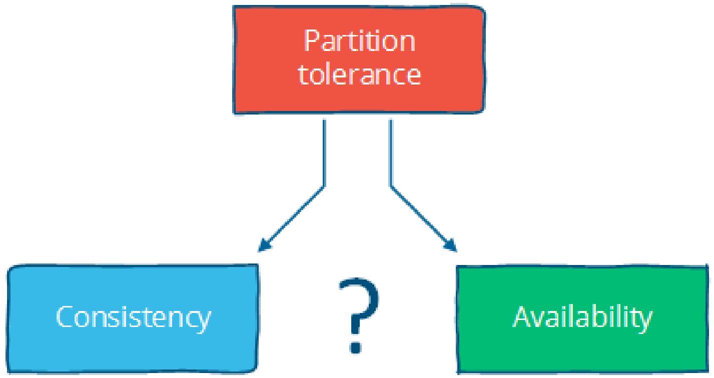
- Use **symmetry** to simplify analysis and management.
- **Harvest** and **yield** are more useful metrics than just uptime.
- Focus on **MTTR** at least as much as on MTBF.
- Data replication is insufficient for preserving uptime under faults; you also need **excess DQ**.
- **Graceful degradation** is a critical part of a high-availability strategy.

CAP Theorem (strong)

- **C**onsistency, **A**vailability, **P**artition tolerance – pick 2
  - Consistency := *strong* consistency
  - Availability := a client can always reach some replica
  - Partition Tolerance := survive network partitions between data replica
- CA: cluster databases
  - Network partitions not part of the fault model
- CP: some distributed databases
  - Majority protocols → unavailable once the majority fails
- PA: web caching
  - Caching → stale copies might be returned

CAP Theorem (weak)

- In practice, you want to lose neither C nor A completely
- Weak CAP Principle:
  “You can have both C and A, except when there is a partition”
  - Partitions are a property of the underlying system (unreliable communication)
  - Distributed storage systems offer configuration options for the C-A trade-off
- Relaxed consistency models (eventual or causal)
- Probabilistic availability
  - Replicate high-priority data



PACELC Model

- If there is a partition (**P**) →
  how does the system trade off availability and consistency (**A**, **C**)?
- Else (**E**) →
  how does the system trade off latency and consistency (**L**, **C**)?
- Network partitions are probably going to happen
  https://github.com/aphyr/partitions-post

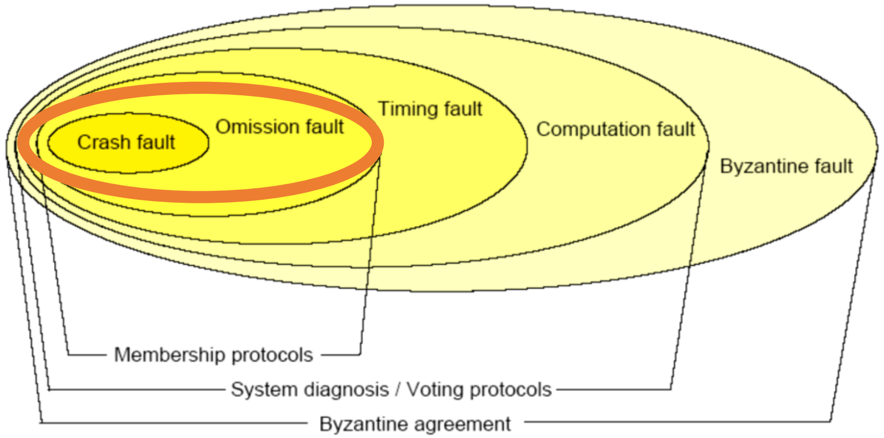
Distributed storage systems classified in terms of PACELC:

- **PA/EL**: Dynamo, Cassandra, Riak
- **PC/EC**: ACID systems, BigTable, Hbase
- **PA/EC**: MongoDB
- **PC/EL**: PNUTS

Consistency vs Availability

- CAP Theorem → in the presence of network partitions, pick Consistency or Availability
- **ACID** usually achieved by 2PC
- *The availability of any system is the product of the availability of the components required for operation.*
  - The more databases involved in 2PC, the lower the availability
- **BASE** (**B**asically **A**vailable **S**oft-state **E**ventual **C**onsistency)
  - ACID's C and I can be traded for availability and performance

Fault Tolerance vs Real Time



Fault Tolerance

- *Reordering* for data consistency
- *Determinism*: coherent data state
- *Concurrency* → harder to ensure consistency

Real Time

- *Reordering* to meet deadlines
- *Determinism*: bounded temporal behaviour
- *Concurrency* → increased efficiency

http://www.ece.cmu.edu/~mead/2003-03-06.pdf

Quiz

1. How does the DQ value scale with
   a) The number of nodes in the system?
   b) The number of tolerated faults?
2. Why is it wise to use a combination of both replication and partition?
3. Why is there a trade-off between strong consistency and availability?

Replication

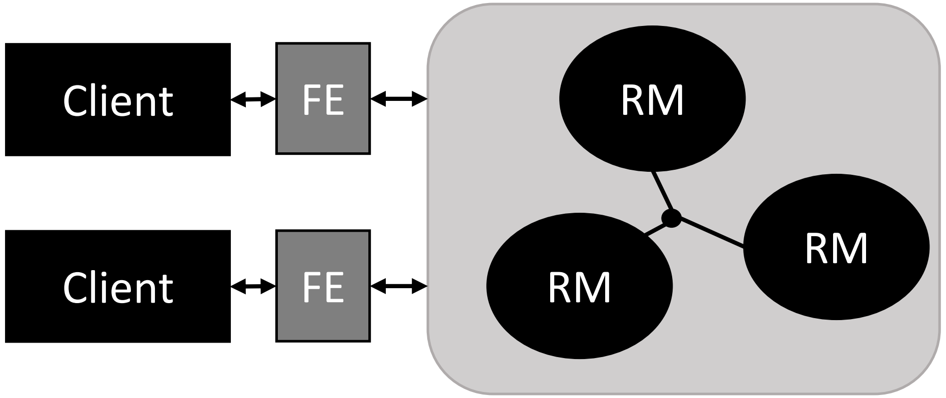
Wiesmann, Matthias, et al. "Understanding replication in databases and distributed systems." *Distributed Computing Systems, 2000. Proceedings. 20th International Conference on*. IEEE, 2000.

Wiesmann, Matthias, et al. "Database replication techniques: A three parameter classification." *Reliable Distributed Systems, 2000. SRDS-2000. Proceedings The 19th IEEE Symposium on*. IEEE, 2000.

http://docs.mongodb.org/manual/replication/

Replication

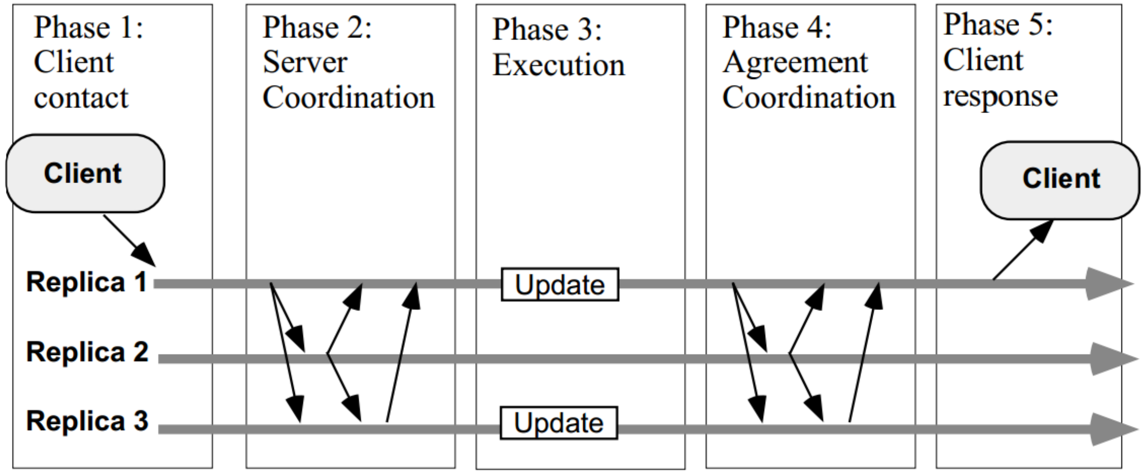
- *Logical objects* implemented by multiple physical copies: ***replicas***
- Clients do operations on replicas, preserving consistency properties
- **Replication transparency**
  - Clients unaware of the existence of individual objects
  - Operations are sent to one copy only
- **Replica managers**
  - Maintain replication transparency
  - Maintain a level of consistency



Replication Protocols

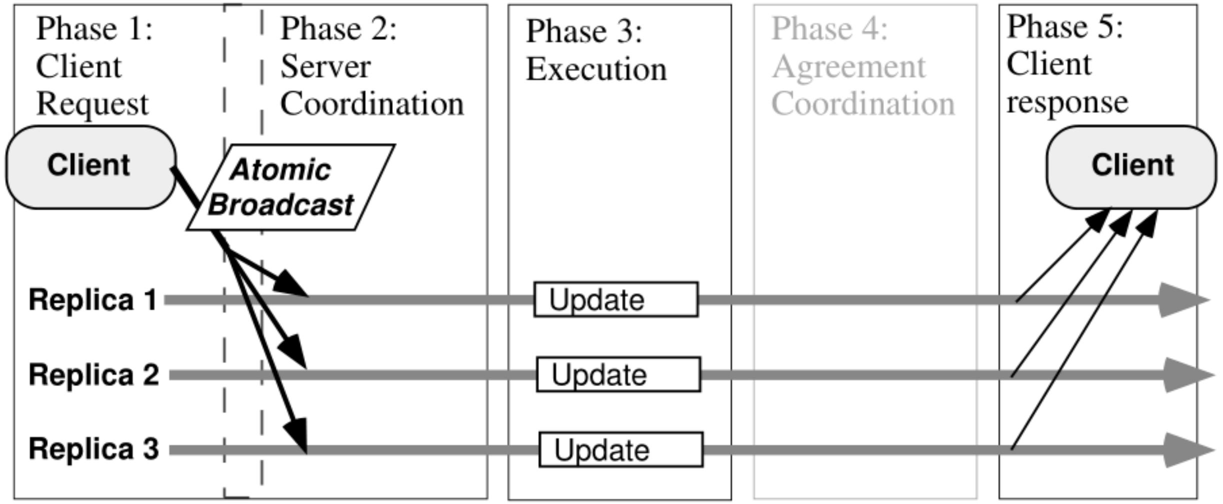
- Abstract replication protocol (Wiesmann et al., 2000)

1. **Request**
   - sent to one (passive replication) or to all replica (active replication)
2. **Server Coordination**
   - find an ordering of operations (sequential consistency)
3. **Execution**
4. **Agreement Coordination**
   - commit or abort?
5. **Response**
   - synchronous vs asynchronous



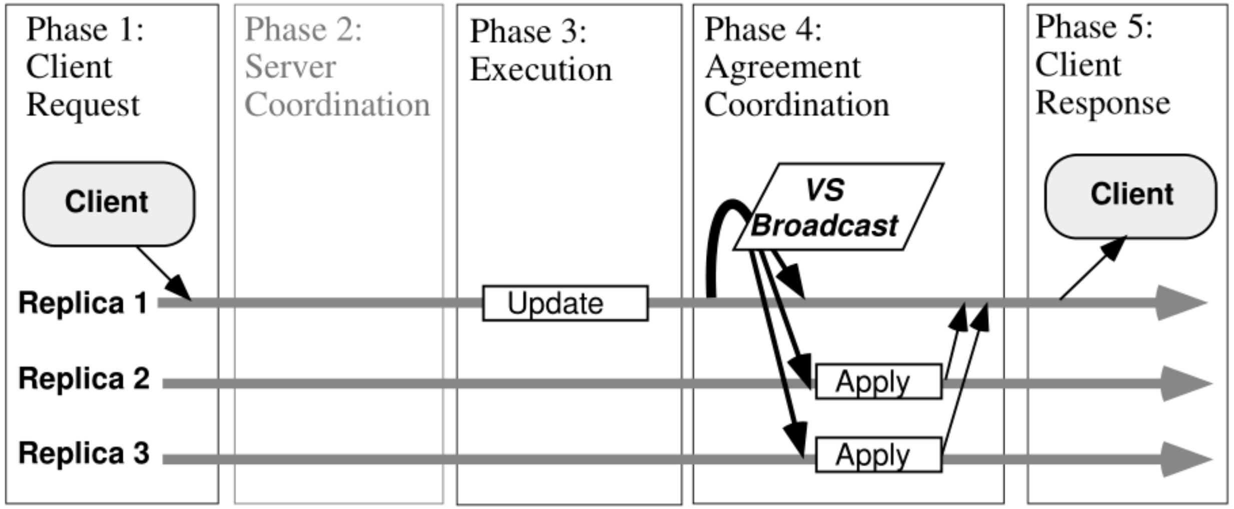
State Machine Replication (Active)

- Non-centralized, all replicas process the same sequence of requests
- Replicas need to work *deterministically*
  - Same ordered input → same result
- Needs *atomic broadcast*
  - All processors receive messages in the same order
  - Either all processors receive the message, or none of them



Primary/Backup Replication (Passive)

- Clients send requests to primary replica
  - Primary sends update requests to backups
  - Updates != original client invocation → non-determinism possible
- Needs view synchronous broadcast
- Asynchronous update

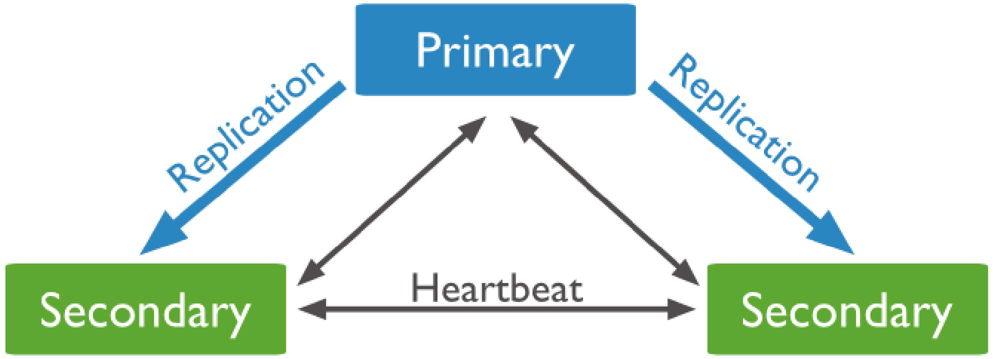


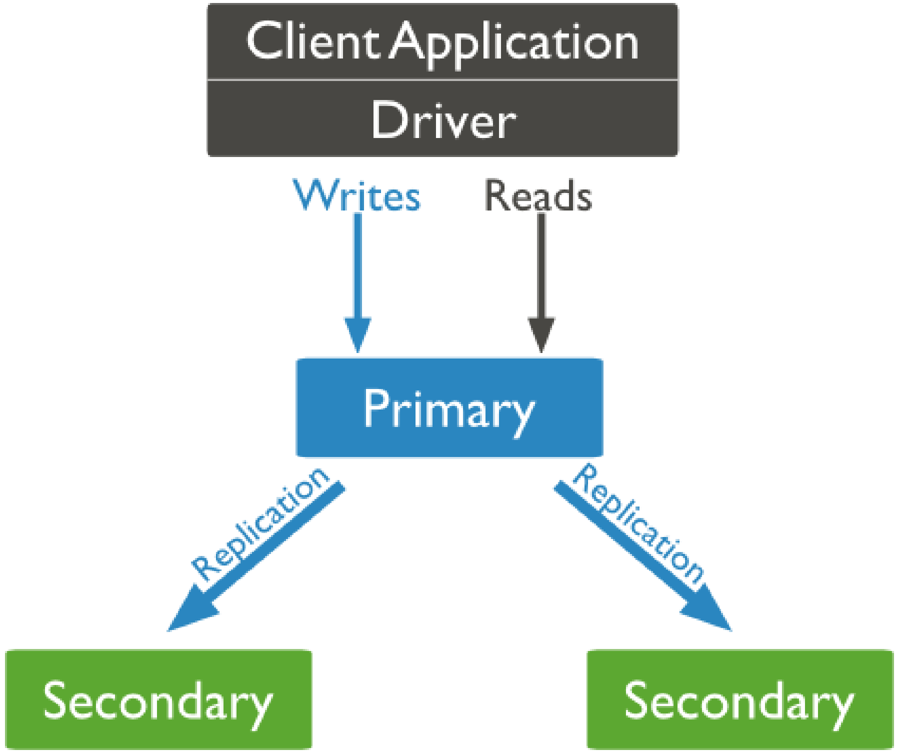
Eager vs Lazy Replication

- Another dimension to the replication problem: *When* are updates processed?
- **Eager replication**: updates propagated within the boundaries of a transaction
  - Before the response is sent to the client
  - E.g., using 2PC
- **Lazy replication**: local update, later propagation
  - Asynchronous → eventually consistent
  - Higher performance

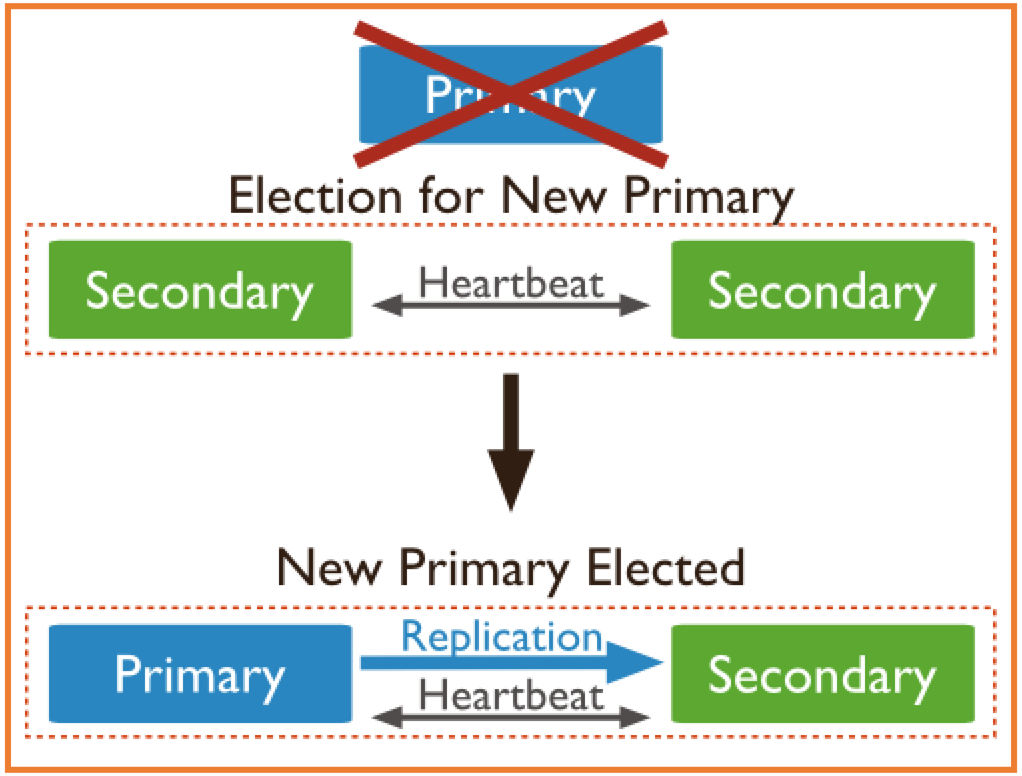
Case Study: Replication in MongoDB

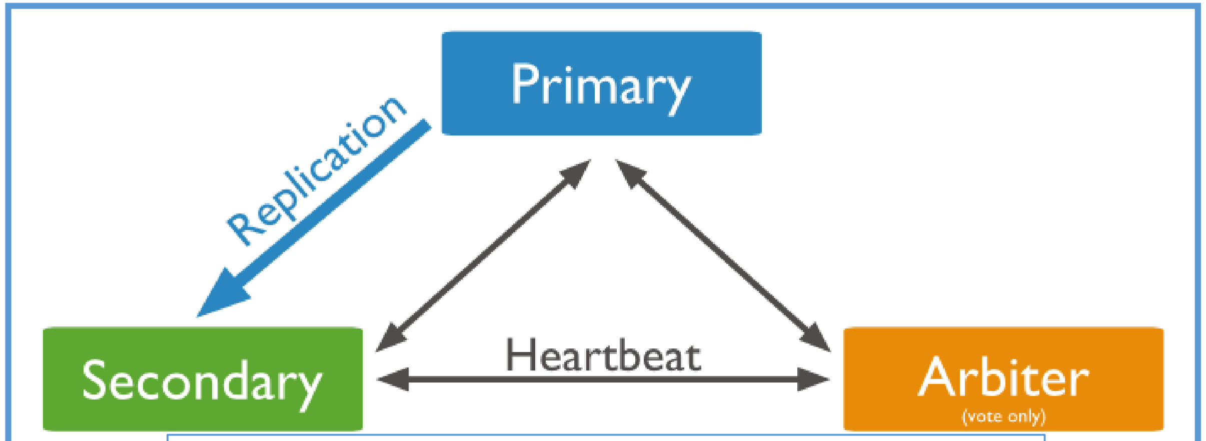
- Primary/Backup replication
  - Writes are always routed to primary
  - Reads can also be routed to secondaries
    - → Increase read availability
- Supports data centre locality awareness



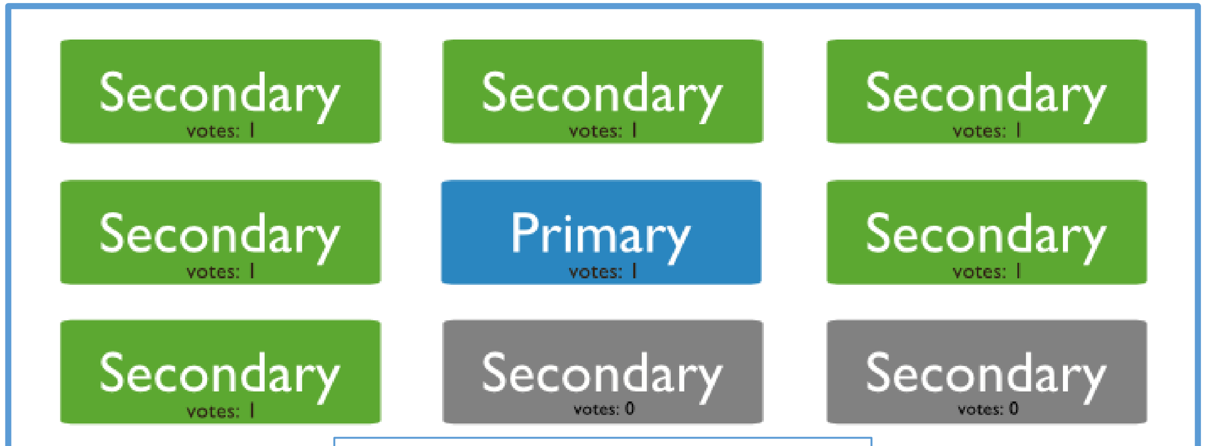


Replication in MongoDB – Primary Failover





Arbiters (for even-numbered replica sets)



Non-voting secondaries