Eventual consistency:
Alleviating scalability problems of replication

Konrad Iwanicki
University of Warsaw

Supplement for Topic 07: Consistency & Replication
Distributed Systems Course
University of Warsaw

Based on multiple sources.

Reminder

Reasons for replication in distributed systems:

- Robustness

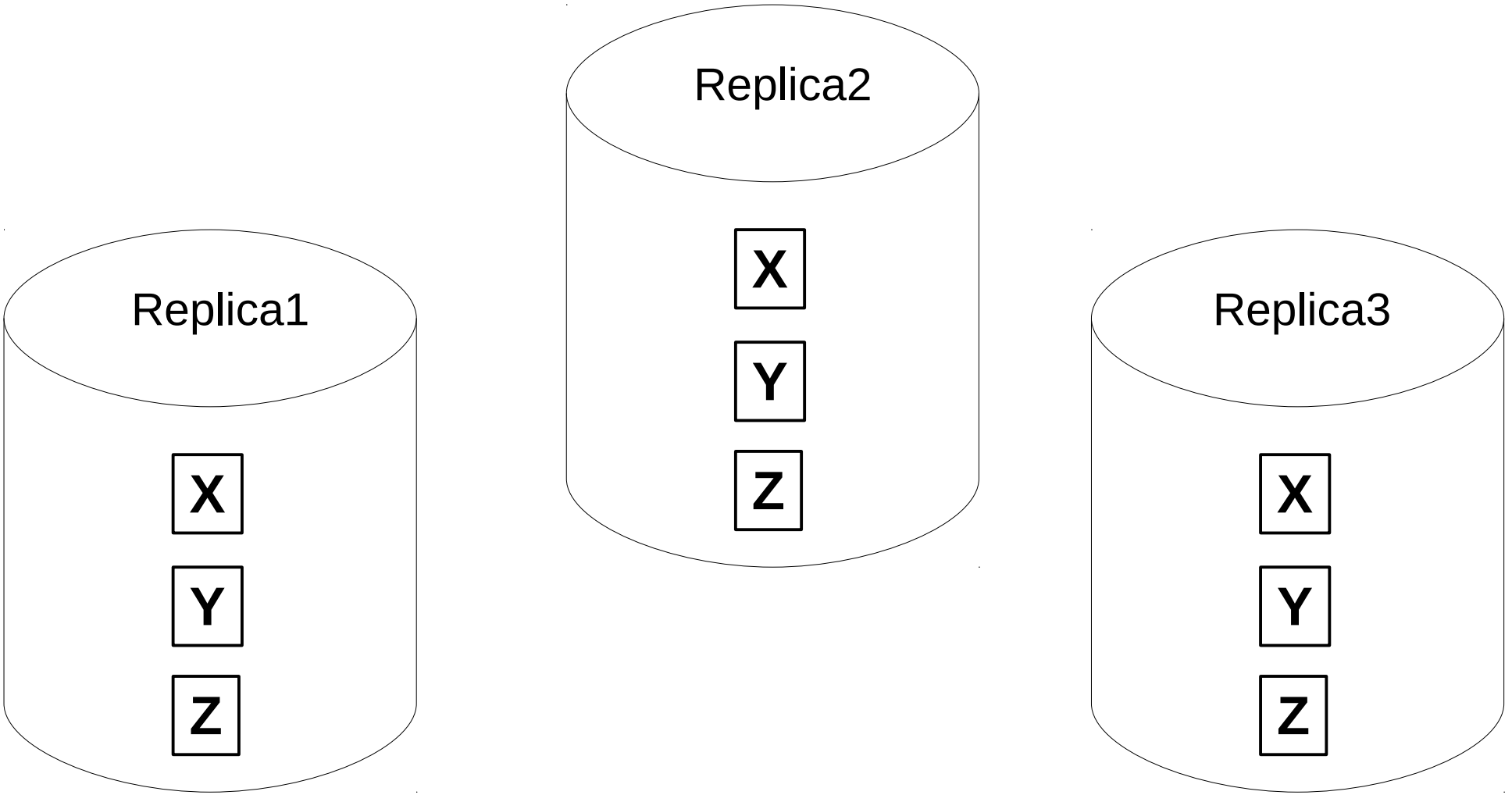
- Performance

Reminder

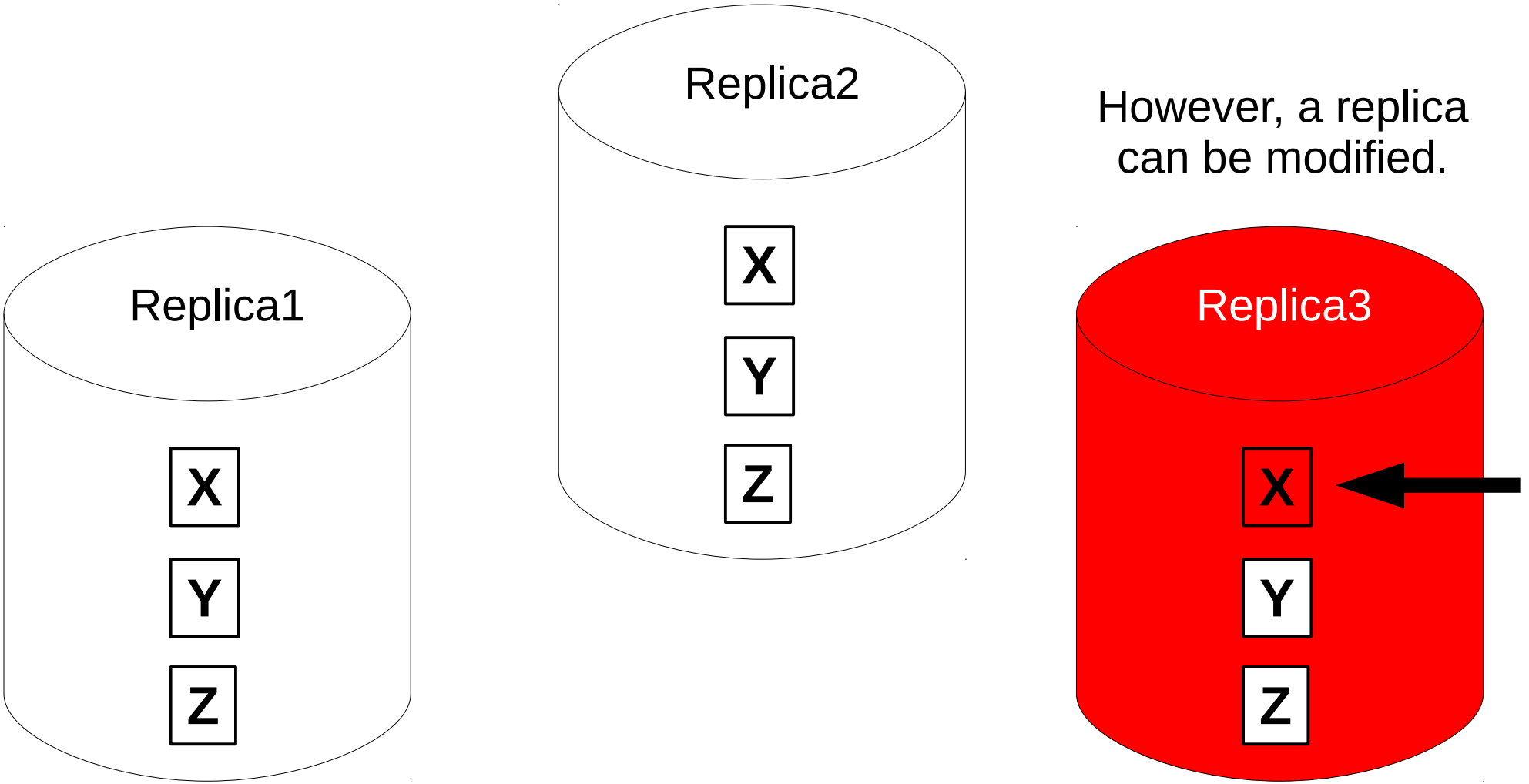
Reasons for replication in distributed systems:

- Robustness
  - Mask machine failures.
  - Tolerate faulty hardware.
- Performance
  - Increase the overall throughput.
  - Minimize client-perceived latency.

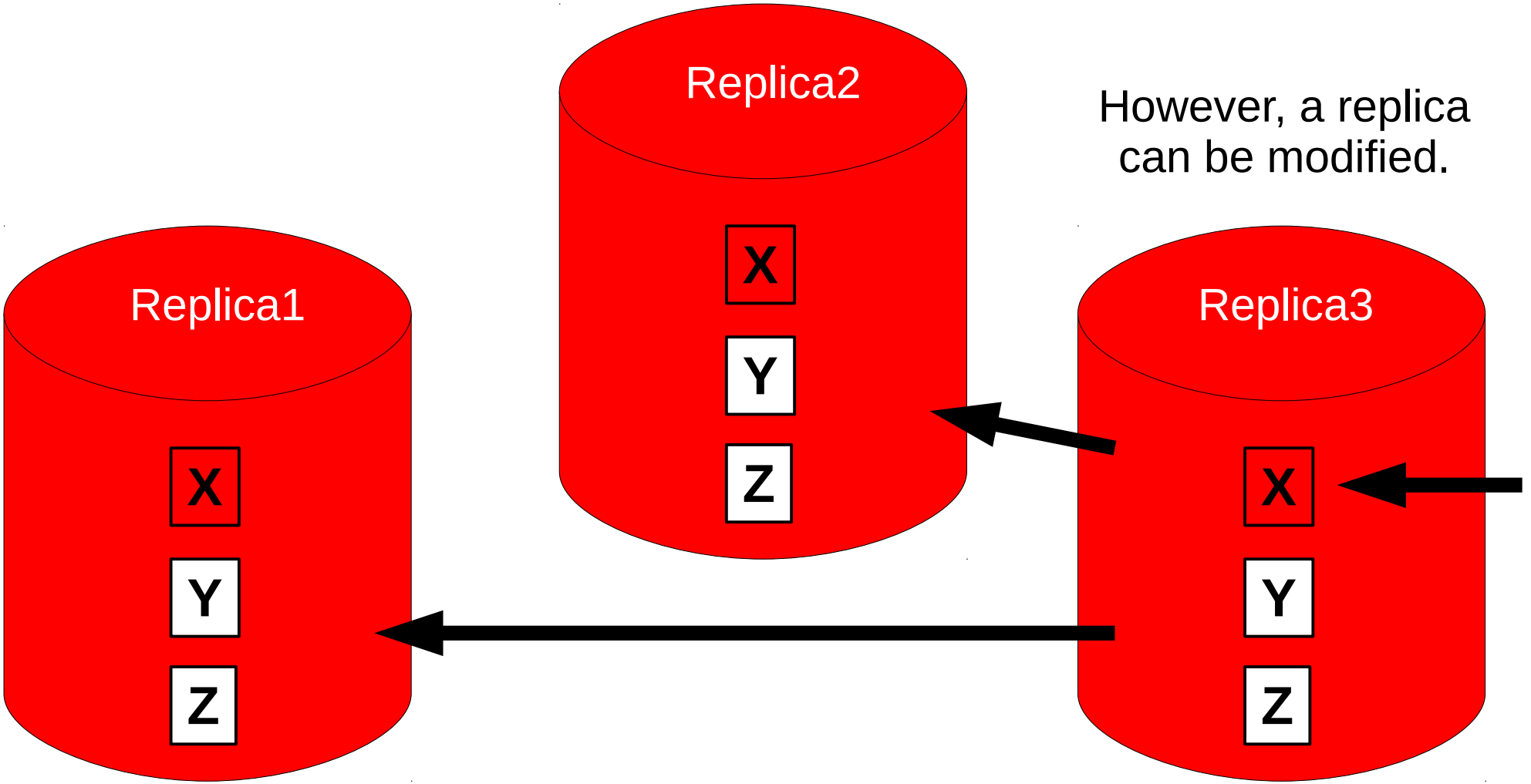
Reminder



Reminder

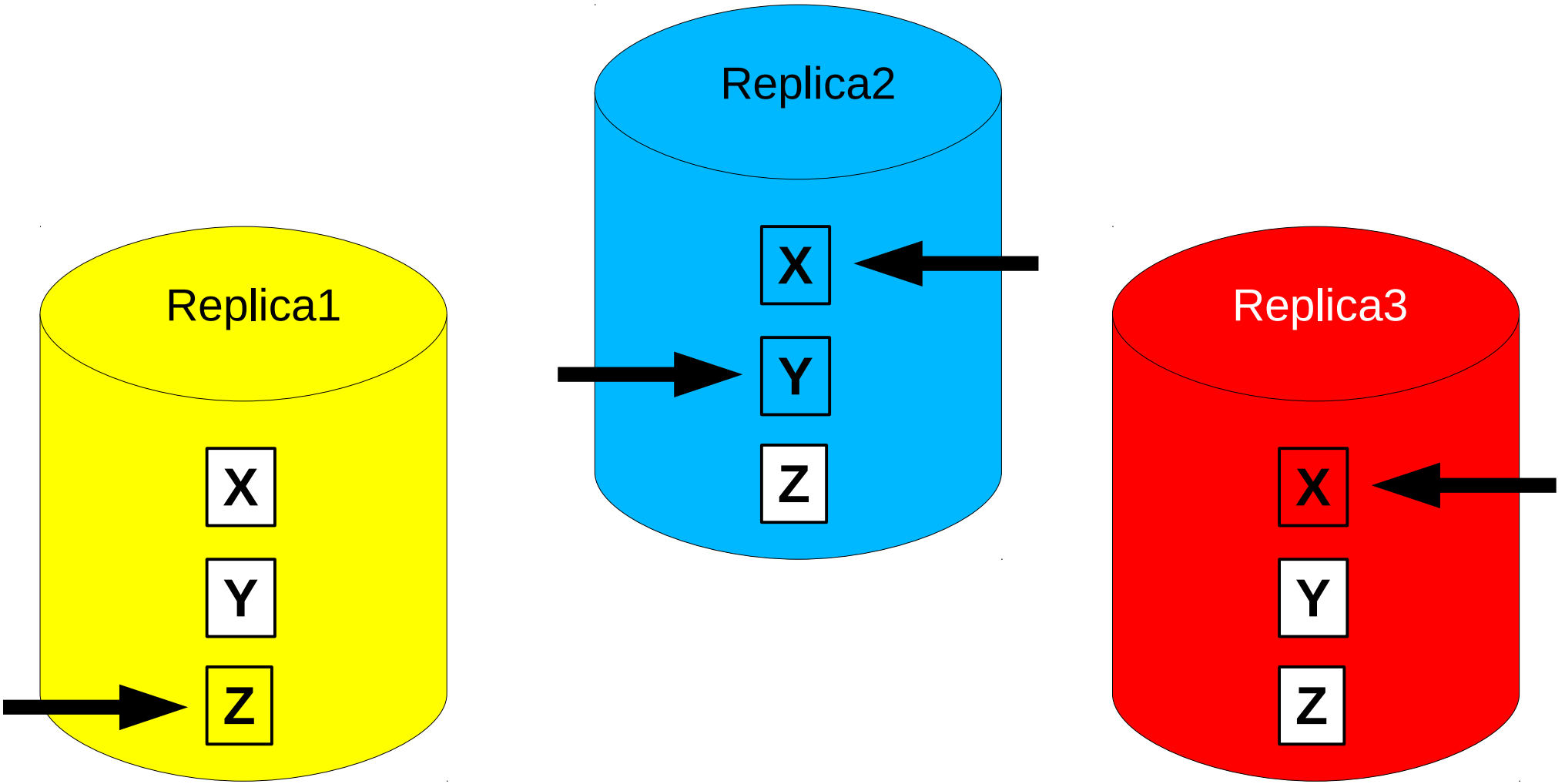

Replica2
X
Y
Z
However, a replica can be modified.
Replica1
X
Y
Z
Replica3
X
Y
Z

Reminder



When one replica is modified, other replicas must be modified as well, so that they remain **consistent**.

Reminder



Updates may be concurrent.

Reminder

Consistency Model

=

A contract between the replica system and its users:
If the users promise to obey certain rules,
the system promises to guarantee certain behavior
in the presence of concurrent operations.

Reminder

Strong consistency

Sequential consistency

All replicas see the same interleaving of operations; the operations initiated at a single replica appear in the Interleaving in the order they were issued at the replica.

Observation

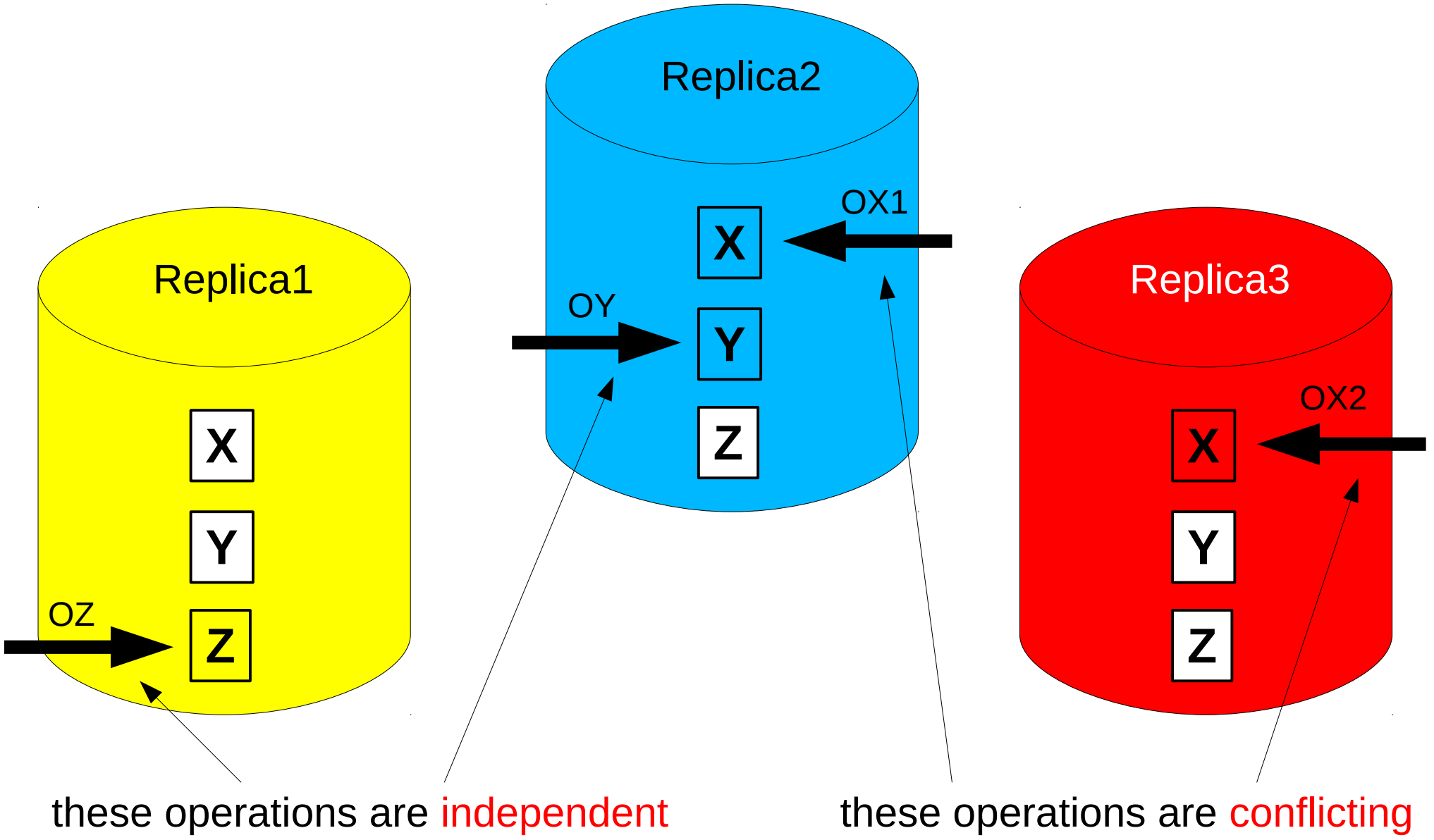
Strong consistency

Sequential consistency

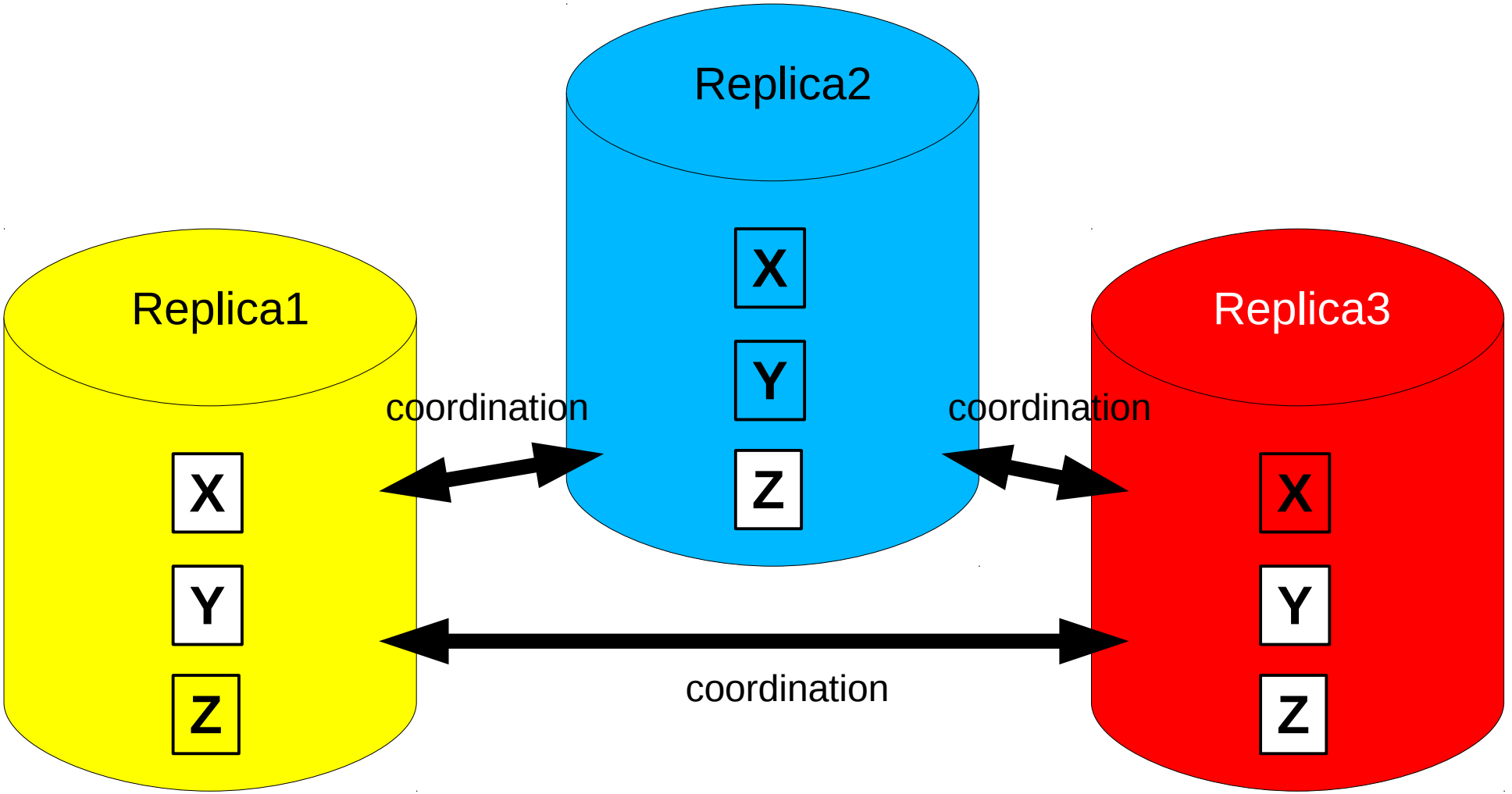
All replicas see the same interleaving of operations; the operations initiated at a single replica appear in the Interleaving in the order they were issued at the replica.

Implies that replicas agree on the ordering of operations.

Reminder

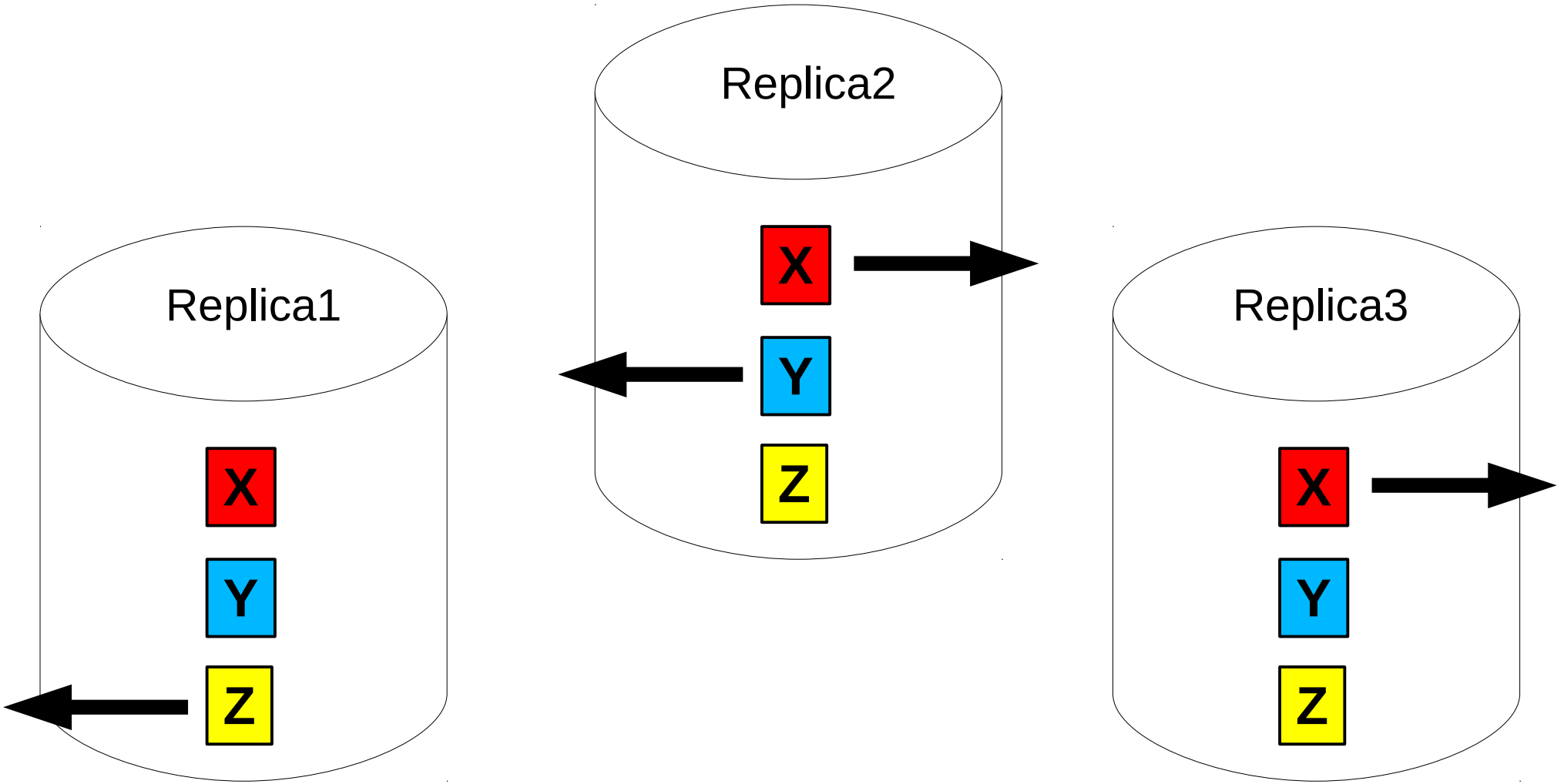

Replica2
OX1
X
OY
Y
Z
Replica1
X
Y
OZ
Z
Replica3
OX2
X
Y
Z
these operations are independent
these operations are conflicting

Reminder



Replicas establish the ordering of the operations.

Reminder

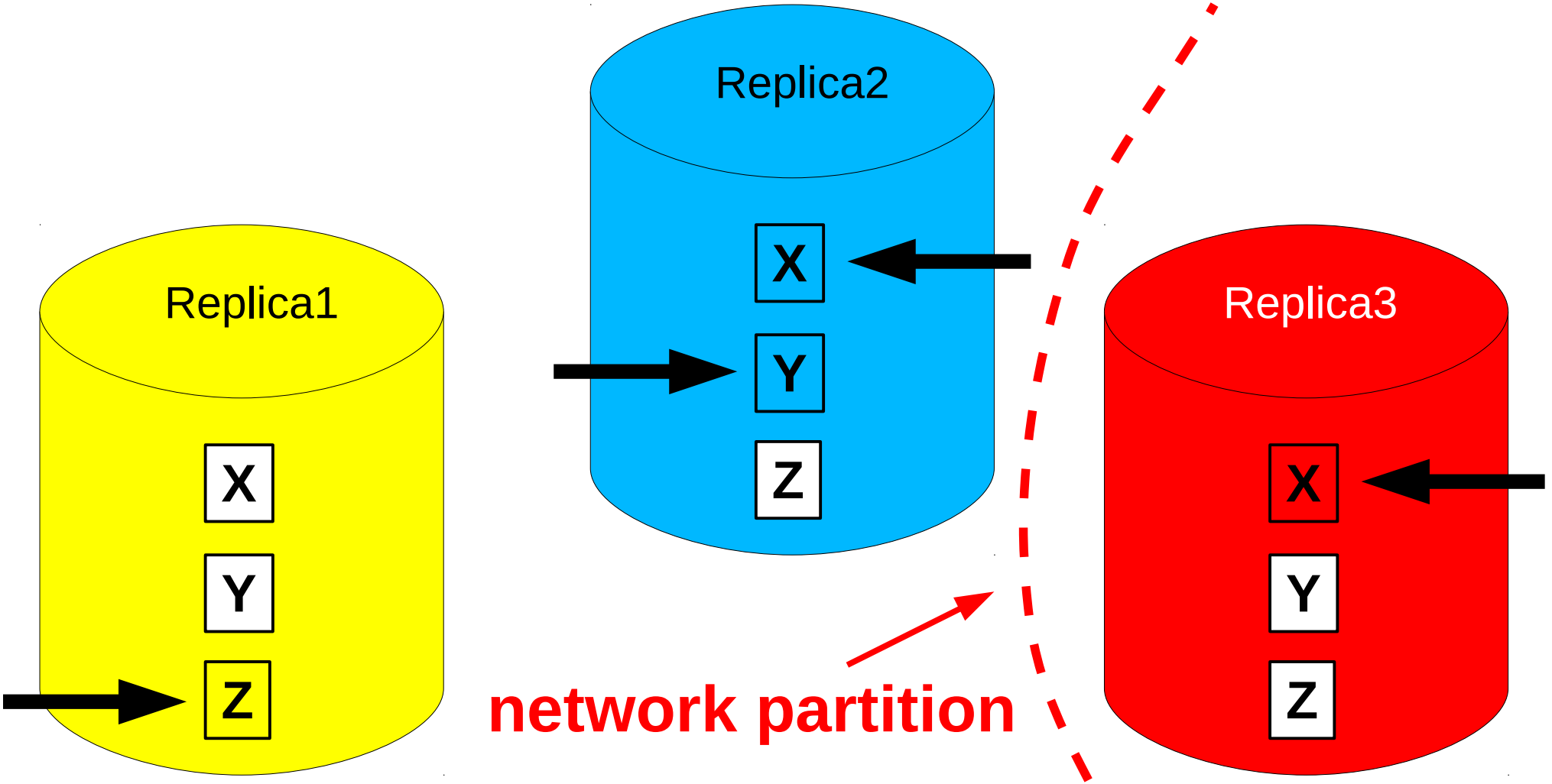


The replicas have converged on ordering OY, OZ, OX1, OX2.
A reply can be returned only after the requested operation has been ordered.

Observation

- Even when we relax consistency (e.g., only potentially conflicting/dependent operations, operation grouping), there still must exist a global agreement on the order of some operations.
- Establishing this order is
  - costly and
  - not always possible.

Observation



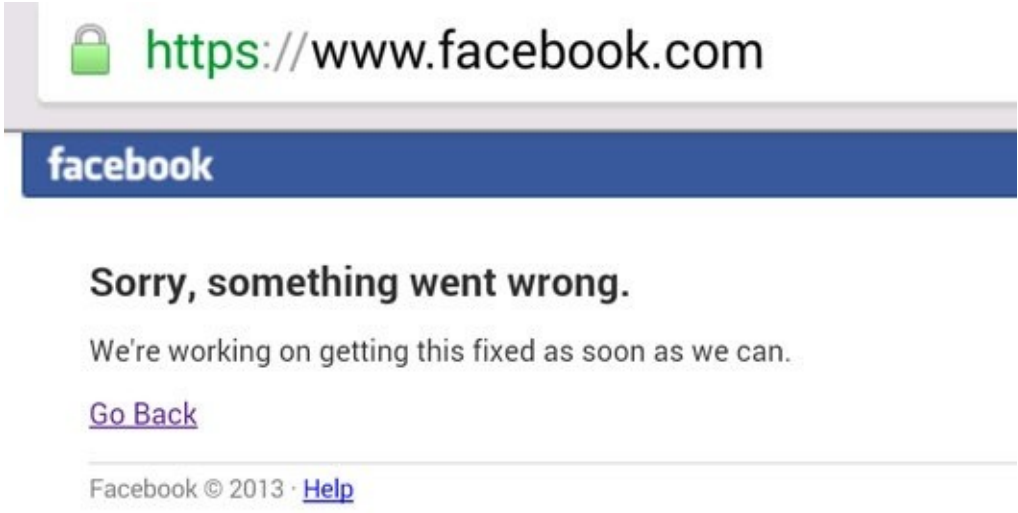
At least one of the partitions must not make any decision as to the ordering of operations; otherwise the system cannot guarantee strong consistency.

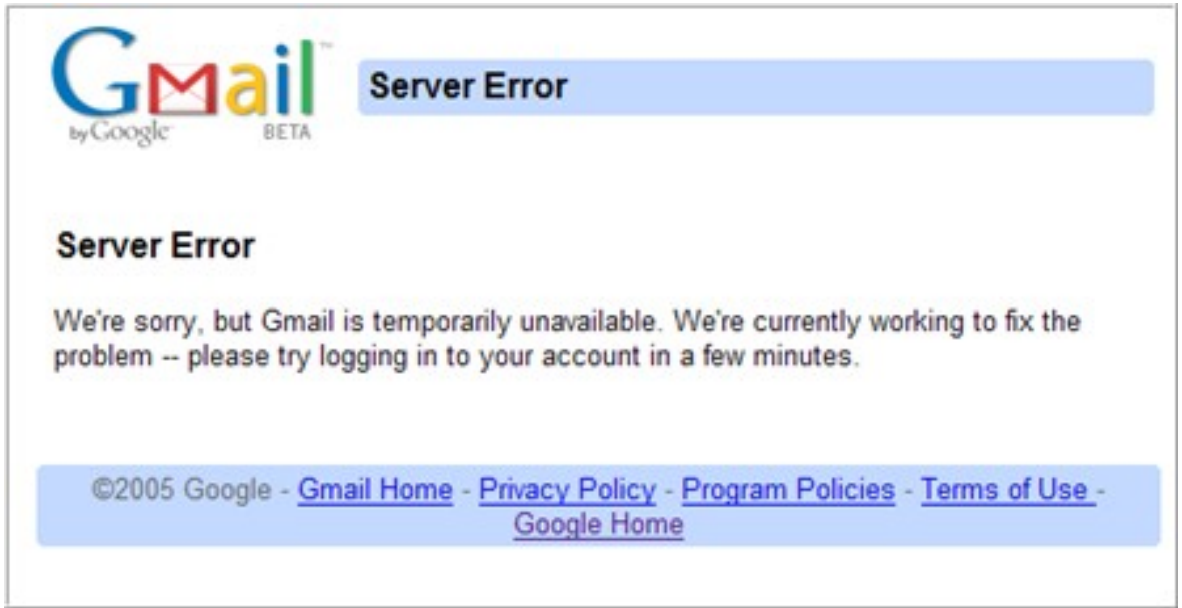
Observation

But many system SHOULD NEVER fail!

Observation

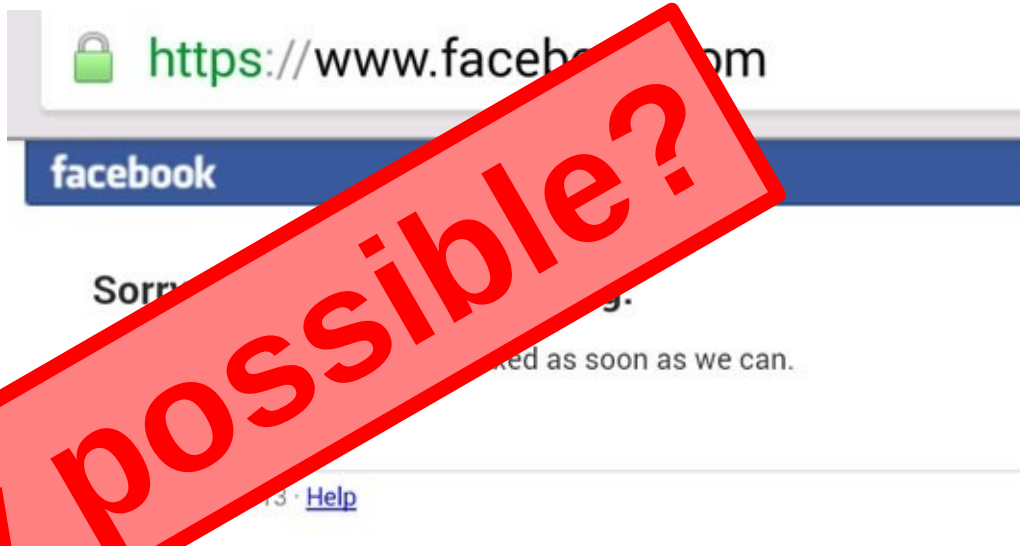
But many system SHOULD NEVER fail!

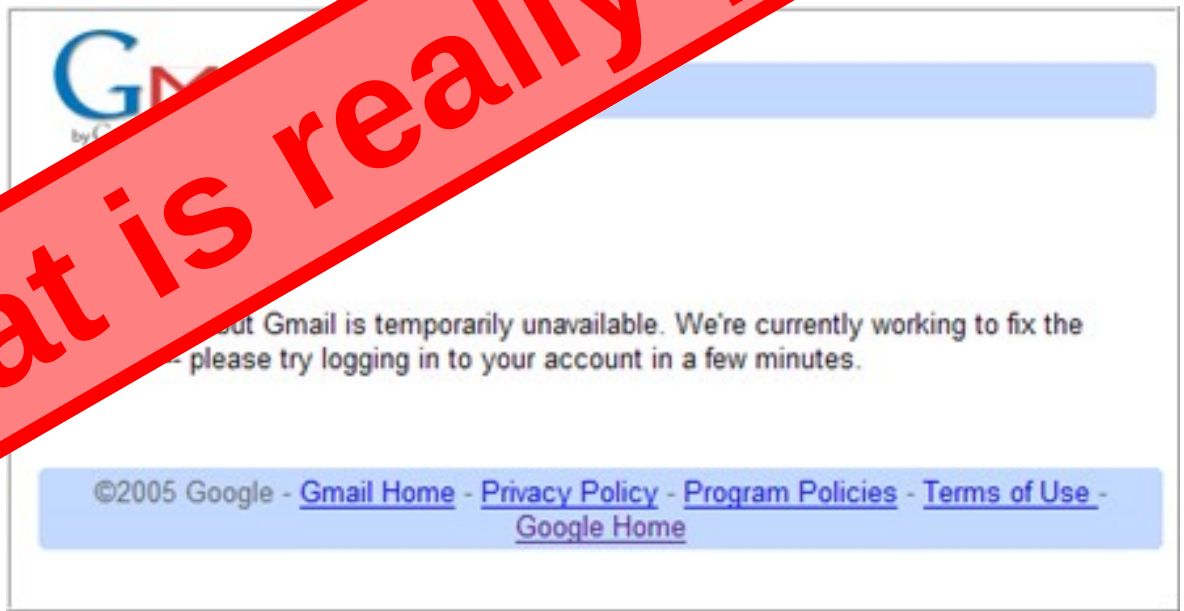




Observation

But many system SHOULD NEVER fail!





What is really possible?

CAP Theorem

(C)onsistency

(A)vailability

(P)artition tolerance

CAP Theorem

(C)onsistency

The property that (some of) replicas must be globally coordinated to make progress with operations submitted concurrently without violating system invariants.

(A)vailability

(P)artition tolerance

CAP Theorem

(C)onsistency

The property that (some of) replicas must be globally coordinated to make progress with operations submitted concurrently without violating system invariants.

(A)vailability

The probability that the system can carry out submitted operations at any given moment in time.

(P)artition tolerance

CAP Theorem

(C)onsistency

The property that (some of) replicas must be globally coordinated to make progress with operations submitted concurrently without violating system invariants.

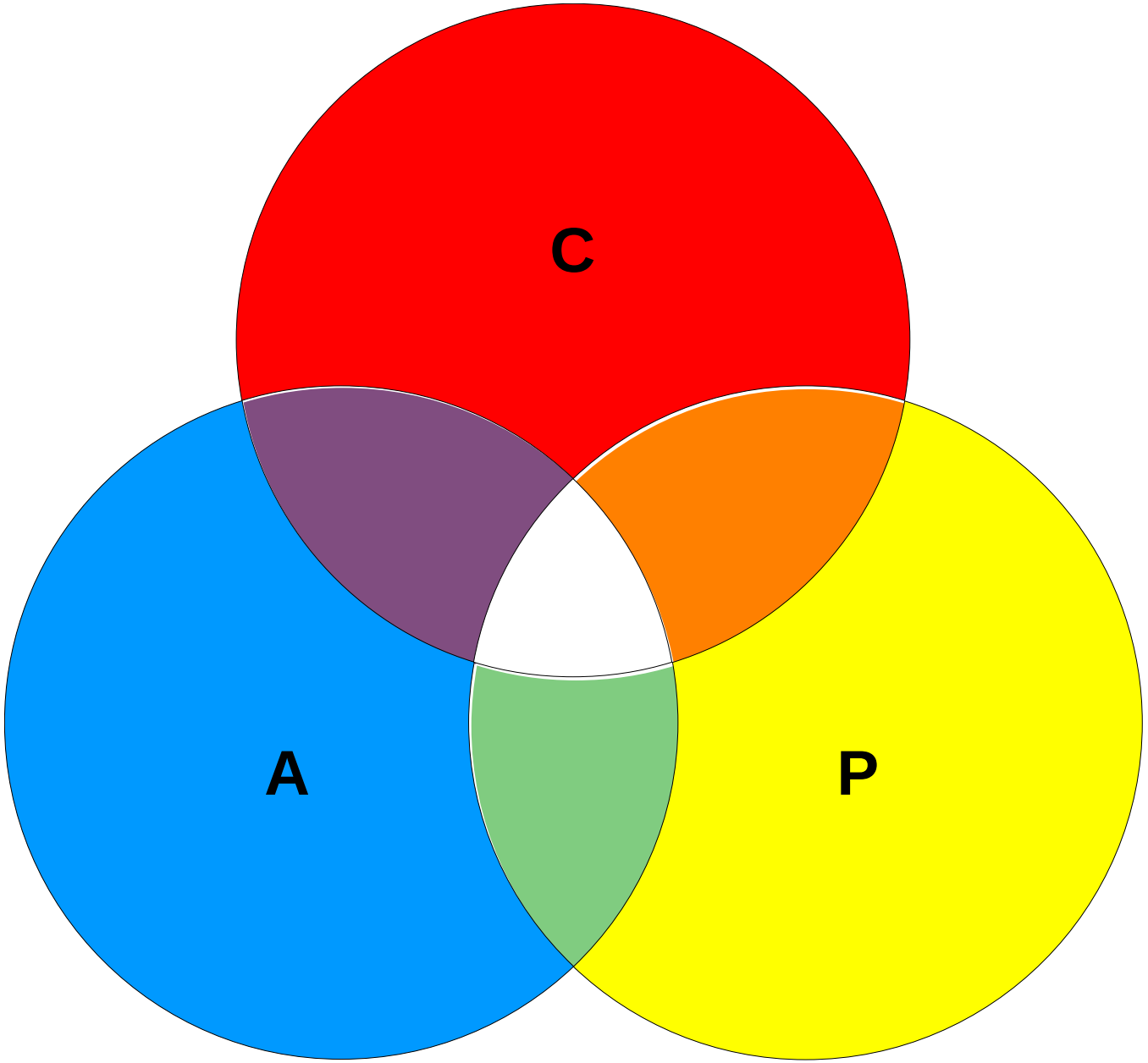
(A)vailability

The probability that the system can carry out submitted operations at any given moment in time.

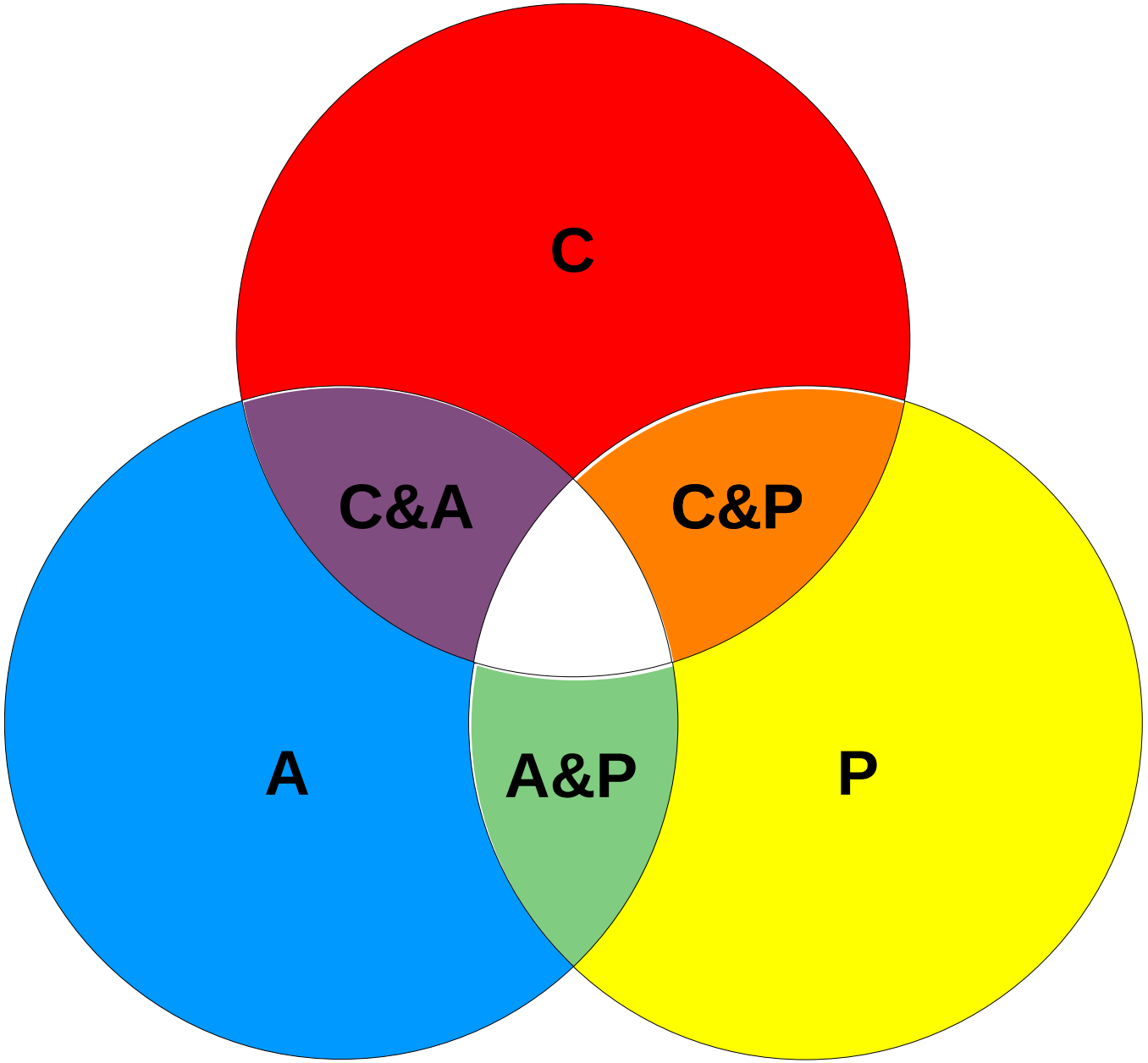
(P)artition tolerance

The capability of the system to operate even when replicas are partitioned into clusters between which communication is not possible.

CAP Theorem



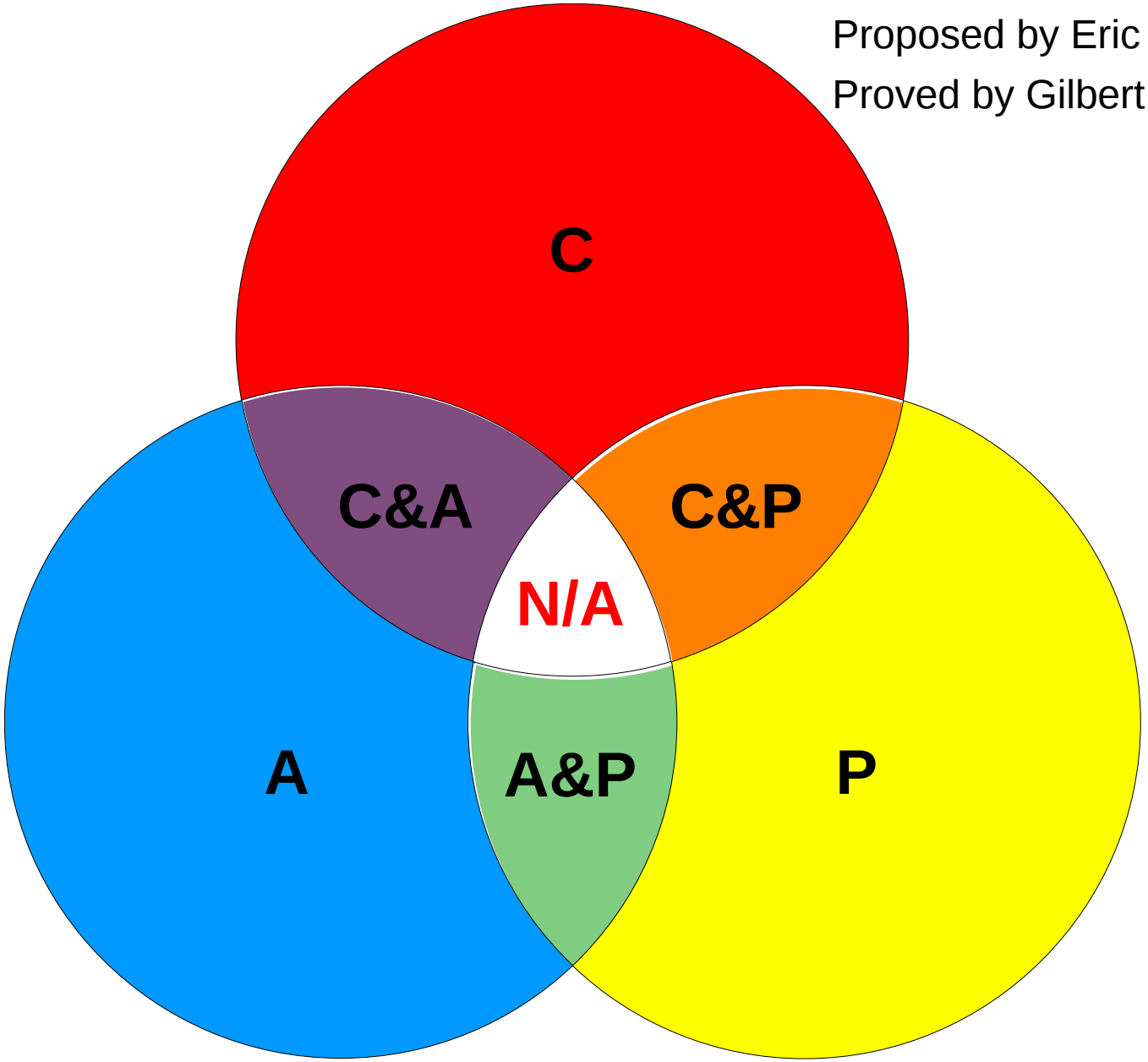
CAP Theorem



CAP Theorem

Proposed by Eric Brewer in 2000.

Proved by Gilbert & Lynch in 2002.



Alternative formulation

- This formulation of the CAP Theorem is slightly misleading.
  - Are partitions that common to design a system specifically for them?
  - In scalable systems, can you actually forfeit partition tolerance?

Alternative formulation

- Werner Vogels, Amazon CTO:
  - “An important observation is that in larger distributed-scale systems, network **partitions are a given**; therefore, consistency and availability cannot be achieved at the same time.”

http://www.allthingsdistributed.com/2008/12/eventually_consistent.html

Alternative formulation

- Coda Hale, Yammer software engineer:
  - “Of the CAP theorem’s Consistency, Availability, and Partition Tolerance, **Partition Tolerance is mandatory in distributed systems**. You cannot not choose it.”

http://codahale.com/you-cant-sacrifice-partition-tolerance/

Alternative formulation

What is partition tolerance is really about?

Alternative formulation

Consider the following scenario:

- A client submits an operation to a replica.
- Should the replica coordinate with other replicas before replying to the client or should it carry out the operation optimistically to minimize the client-perceived latency?
- If it does coordinate, suppose that it does not get a reply from the required number of replicas within a given time limit.
- Should it optimistically carry out the operation or should it report an error to the client?

Alternative formulation

PACELC (pronounced “pass-elk”):

- If there is a partition (P), how does the system trade off availability (A) and consistency (C);
- Else (E) when the system is running normally in the absence of partitions, how does it trade off latency (L) and consistency (C)?

Brewer 2012

Alternative formulation

System examples:

- **PA/EL**: Give up both Cs for availability and lower latency
  - Dynamo, Cassandra, Riak
- **PC/EC**: Refuse to give up consistency and pay the cost of availability and latency
  - BigTable, Hbase, VoltDB/H-Store
- **PA/EC**: Give up consistency when a partition happens and keep consistency in normal operations
  - MongoDB
- **PC/EL**: Keep consistency if a partition occurs but gives up consistency for latency in normal operations
  - Yahoo! PNUTS

Alternative formulation

System examples:

- **PA/EL**: Give up both Cs for availab[illegible]nd lower latency
  - Dynamo, Cassandra, Riak
- **PC/EC**: Refuse to give up co[illegible]ency and pay the cost of availability and latency
  - BigTable, Hbase, VoltDB[illegible]re
- **PA/EC**: Give up con[illegible]icy when a partition happens and keep consistency[illegible]mal operations
  - MongoDB
- **PC/EL**: Kee[illegible]sistency if a partition occurs but gives up consisten[illegible] latency in normal operations
  - Yaho[illegible]UTS

How can we trade off consistency?

Eventual consistency

Weakest sensible consistency

Eventual consistency

After operations in the system have stopped being issued, eventually all replicas will converge to the same state.

Eventual consistency

Weakest sensible consistency

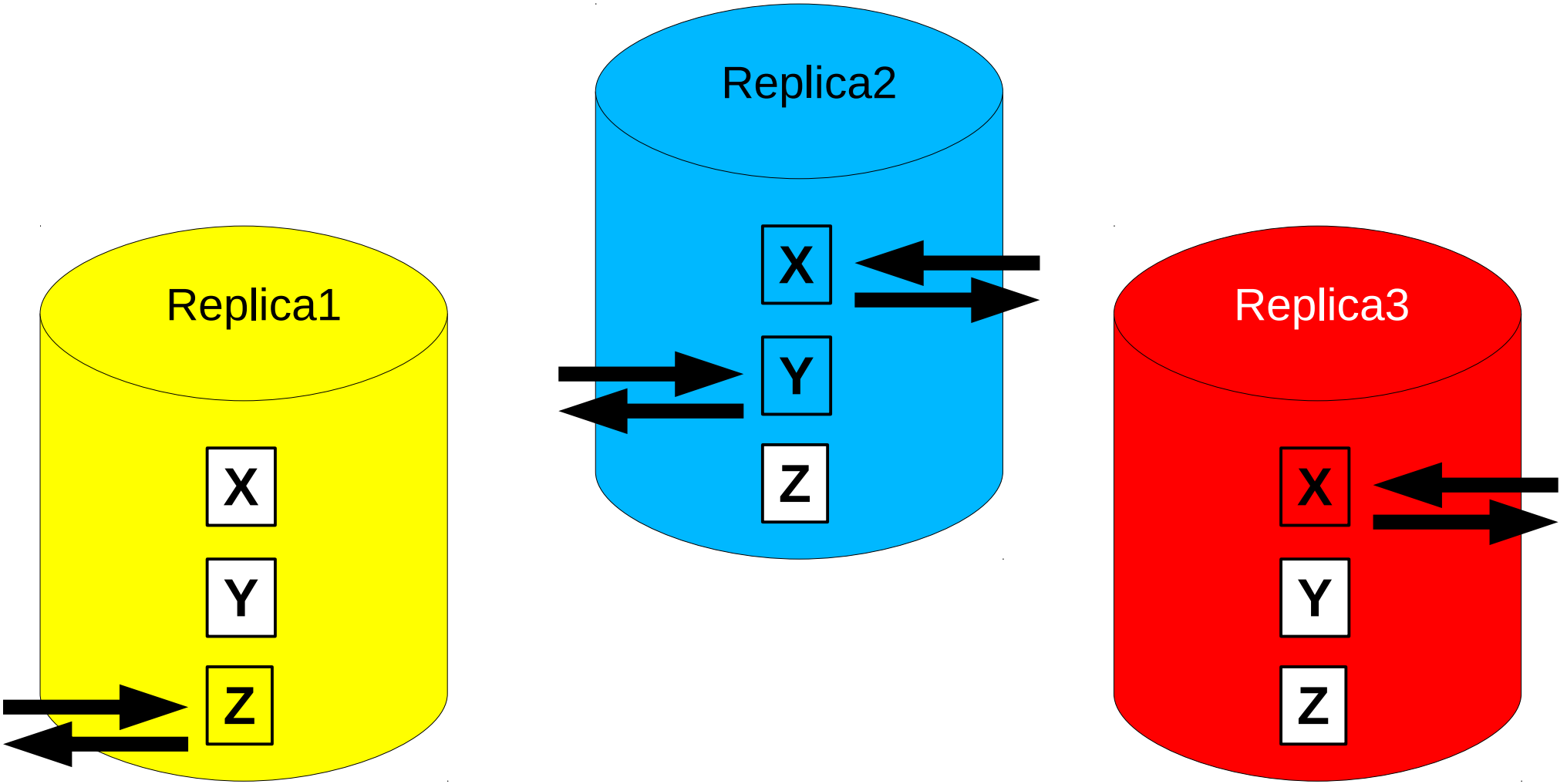
Eventual consistency

After operations in the system have stopped being issued, eventually all replicas will converge to the same state.

Sample application scenarios:

- Delay-tolerant systems (e.g., DNS).
- Mobile opportunistic systems (e.g., Bayou).
- Group-editing software (e.g., GIT).
- Cloud computing (e.g., Dynamo).

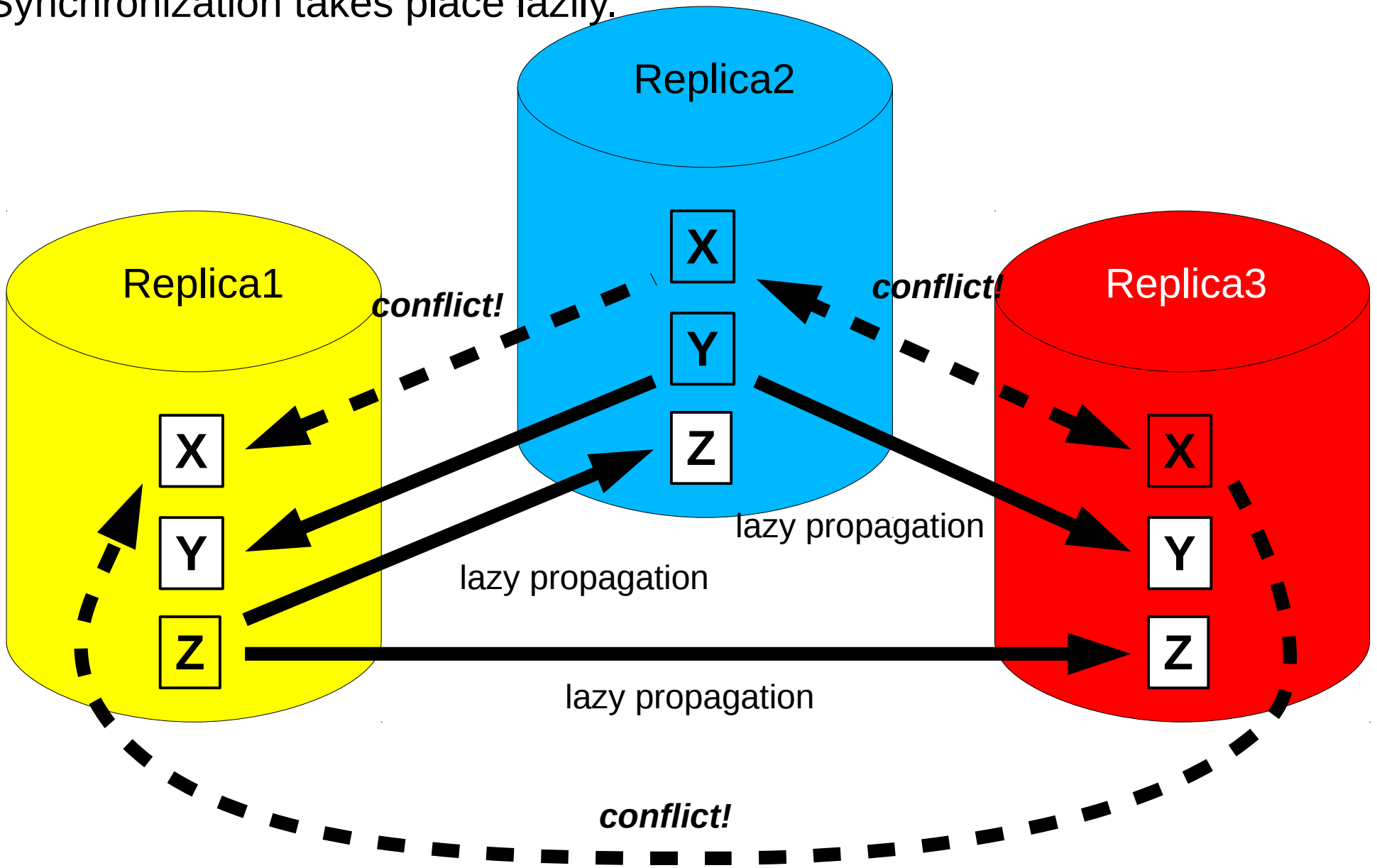
Reminder



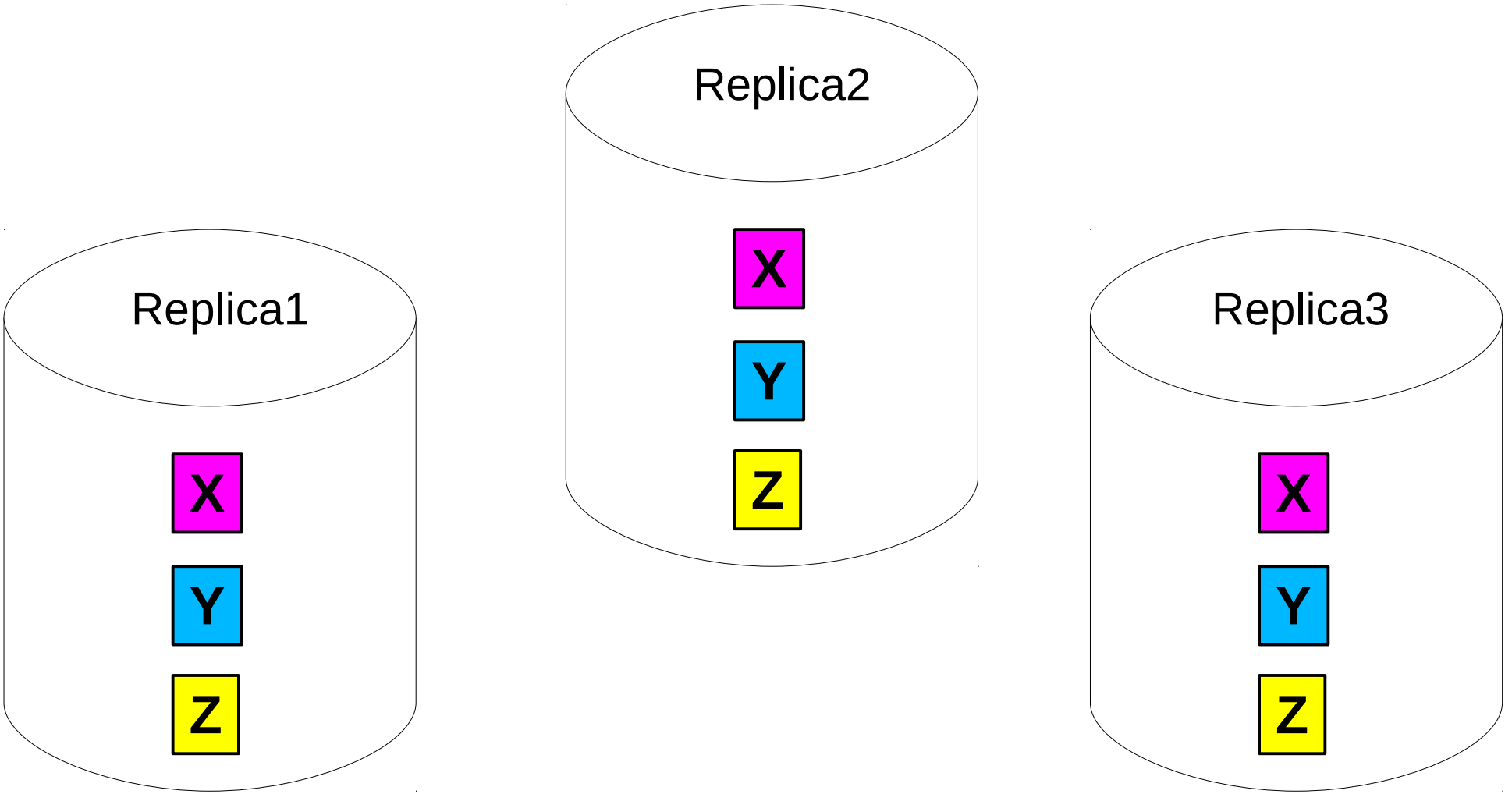
Each replica proceeds with local updates asynchronously.

Reminder

Synchronization takes place lazily.



Reminder



How to deal with conflicts?

Avoiding conflicts

- Allow for reading from any replica, but...
- ...designate a single (master) replica to perform updates.
  - Eliminates write-write conflicts.
  - Read-write conflicts are still there (will address them later).
- Must give up availability or partition tolerance.
- Surprisingly, not so uncommon.

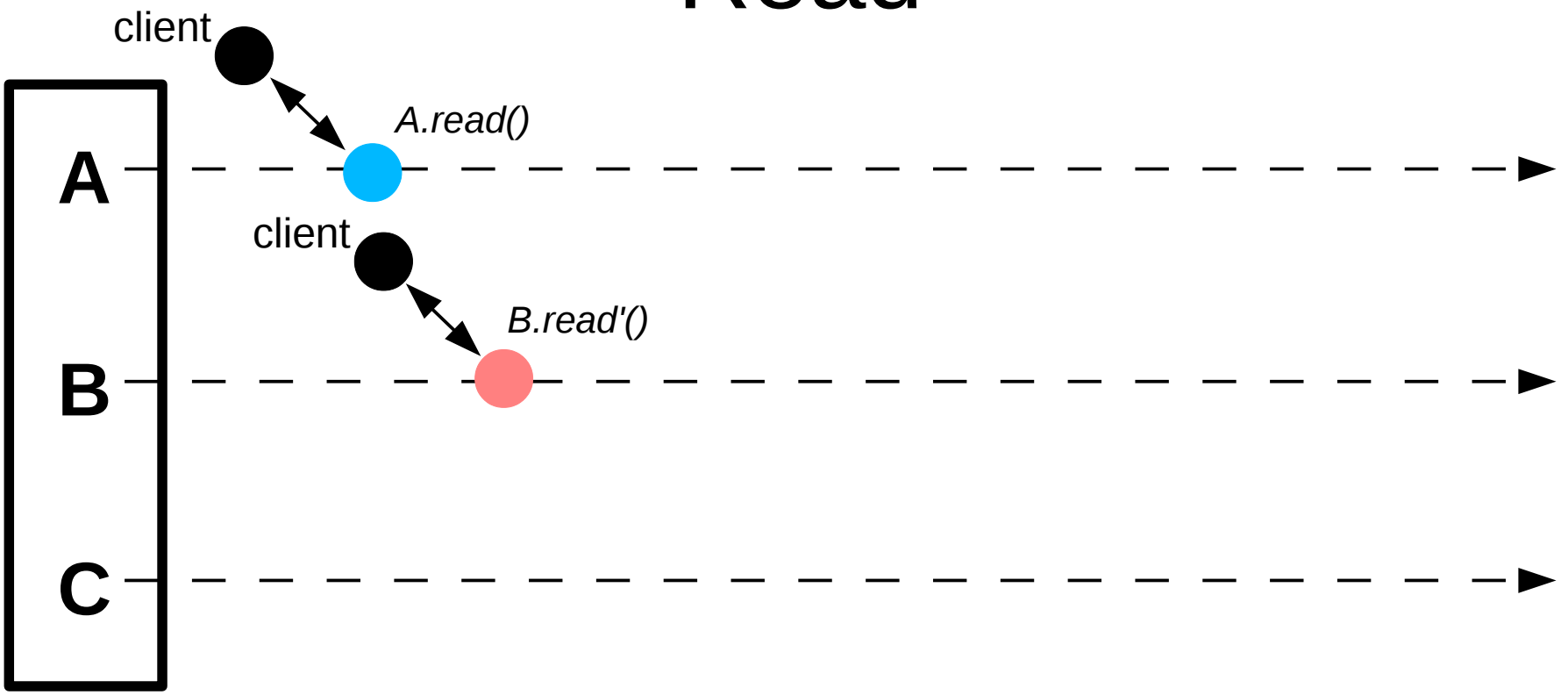
Example: Facebook

- Read from closest server.
- Write to California.
- Other servers synchronize every 15 minutes.
- After a write, read from California for 15 minutes.

Dealing with conflicts

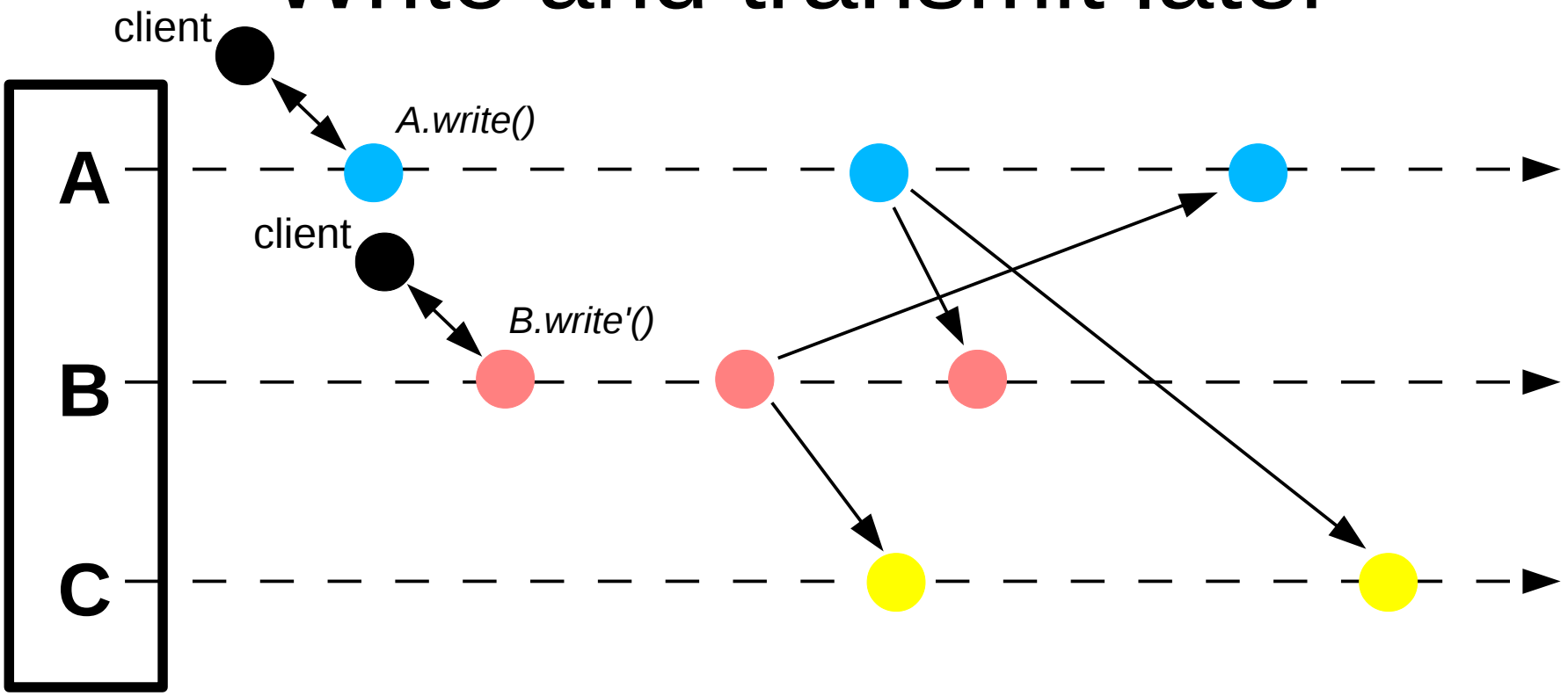
- Read and update replica (multiple masters).
- Transmit later.
- Detect conflict.
- Reconcile to a common state.
- Garbage-collect metadata.

Read



Read from a local replica.

Write and transmit later



Update a local (source) replica.
Transmit to other (downstream) replicas later.

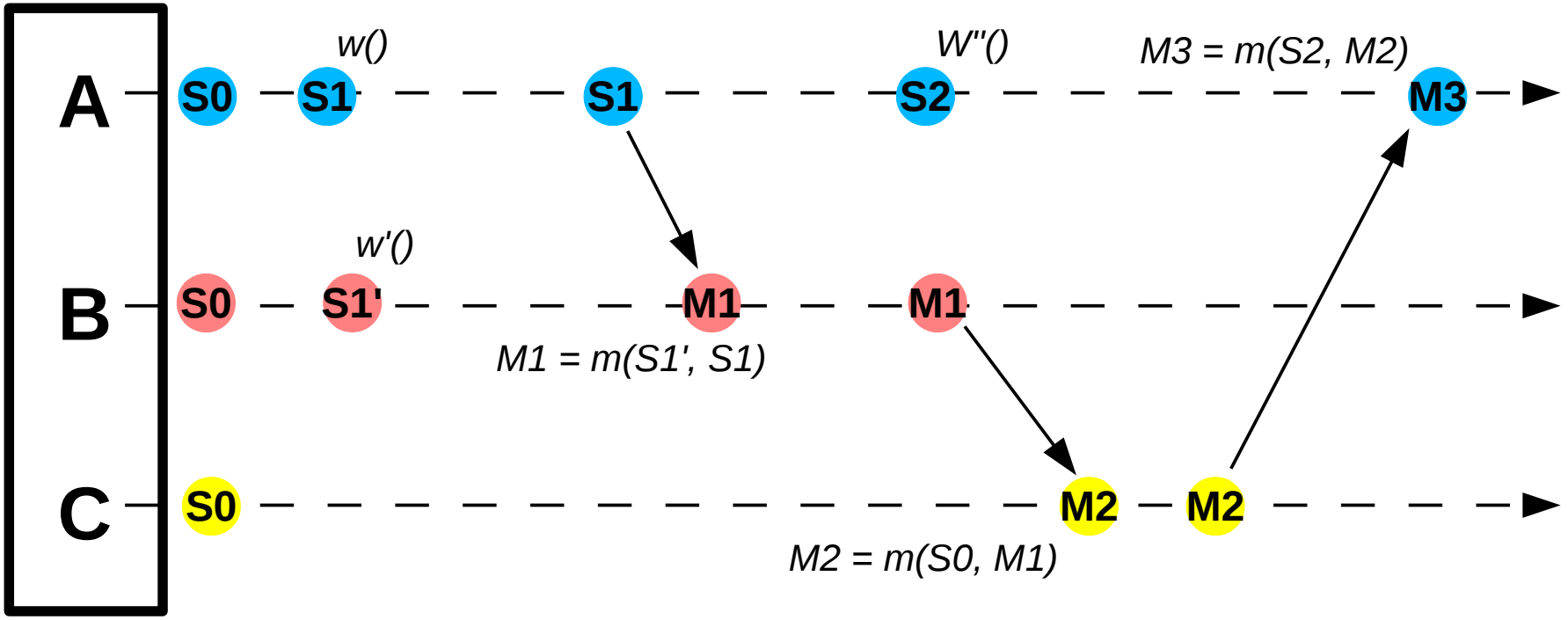
- Lazily / in background.
- Flooding, gossiping, anti-entropy, …

Receiving replicas apply the updates.

Write and transmit later

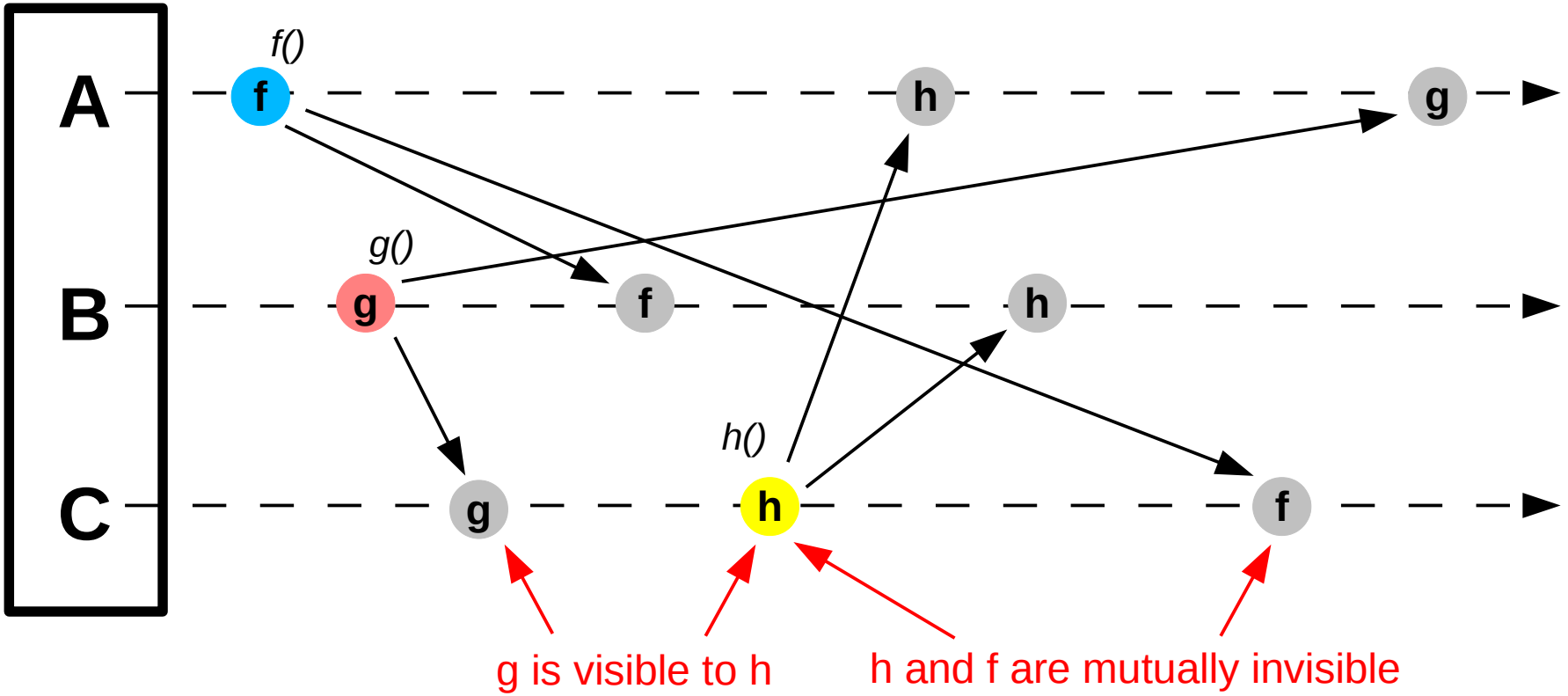
- What to transmit as updates?
- Which updates to transmit?

Data shipping (state-based)



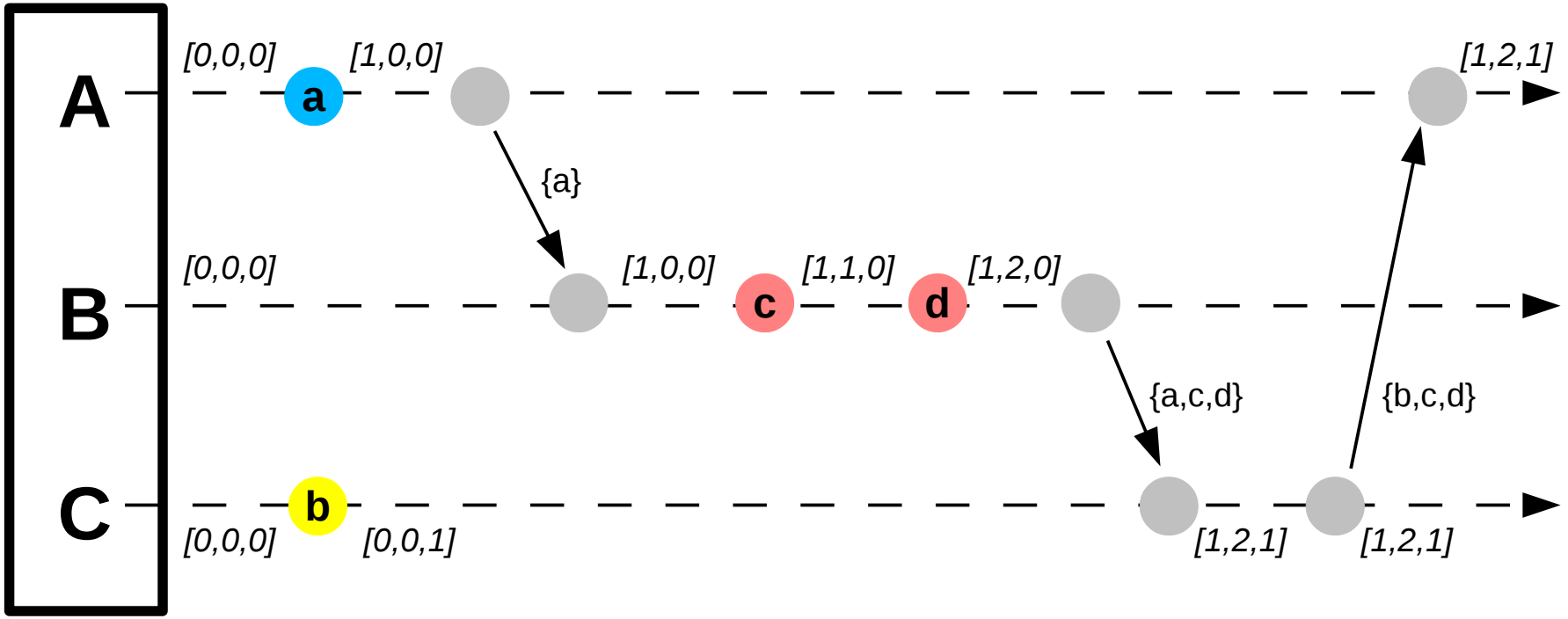
Transmit a replica state.
Merge the local state with the received one into a new local state.
State exchange may be arbitrary.
Example: GIT, SVN, dropbox.

Function shipping (op-based)



Disseminate an operation.
Apply a received operation locally.
Reliable broadcast for dissemination.
Example: Bayou, online text editing.

Vector clocks and anti-entropy



$V_j[i]$ = #updates at i of which j knows

$\max(0, V_j[i] - V_k[i])$ = #updates at i that j should transmit to k

Detecting concurrent updates

Two updates, a and b:

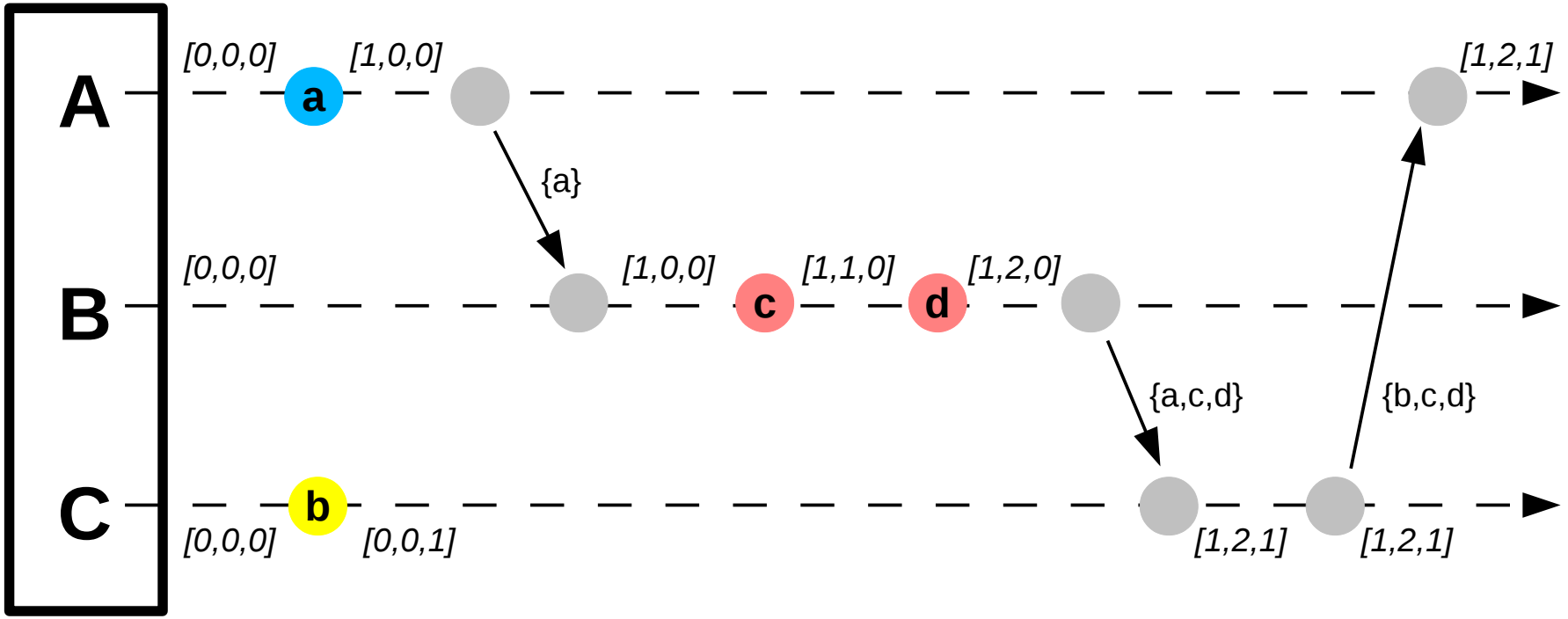
- $O(a) < O(b)$ => a is before b (they are not concurrent)
- $O(a) \not< O(b)$ and $O(b) \not< O(a)$ => a and b are concurrent

Approaches to detecting concurrency:

- Explicit history/dependence check
- Vector clocks
- Programmed checks

Metadata are required.

Vector clocks and concurrency



$V_j[i]$ = #updates at i of which j knows

$O(a) = V_a = V_j$ just after a was executed at replica j (the original replica at which a was performed)

Detecting concurrency:

- $V_a < V_b \equiv V_a[k] < V_b[k]$ for all k ≡ a happened before b
- $V_a \not< V_b$ and $V_b \not< V_a \equiv$ a is concurrent with b

Resolving concurrent updates

Resolution methods:

Resolving concurrent updates

Resolution methods:

- Concurrent updates do not conflict

Resolving concurrent updates

Resolution methods:

- Concurrent updates do not conflict
- Dynamic total order (consensus)

Resolving concurrent updates

Resolution methods:

- Concurrent updates do not conflict
- Dynamic total order (consensus)
- Static total order (arbitration)

Resolving concurrent updates

Resolution methods:

- Concurrent updates do not conflict
- Dynamic total order (consensus)
- Static total order (arbitration)
- Resolver algorithm

Resolving concurrent updates

Resolution methods:

- Concurrent updates do not conflict
- Dynamic total order (consensus)
- Static total order (arbitration)
- Resolver algorithm
- User input

Metadata garbage collection

- At least once delivery:
  - An update must be available for forwarding until received by ***all*** replicas.
- Sample synchronous approach (PODC'84):
  - Receiver acknowledges an update to all replicas.
  - If a replica receives an acknowledgment for the update from all other replicas, it can discard metadata for the update.
  - Is not live in the presence of partitions.
- Sample asynchronous approach (Bayou):
  - Truncate update log arbitrarily.
  - If necessary, recover with full state transfer.
  - May lose updates.

More on conflict resolution

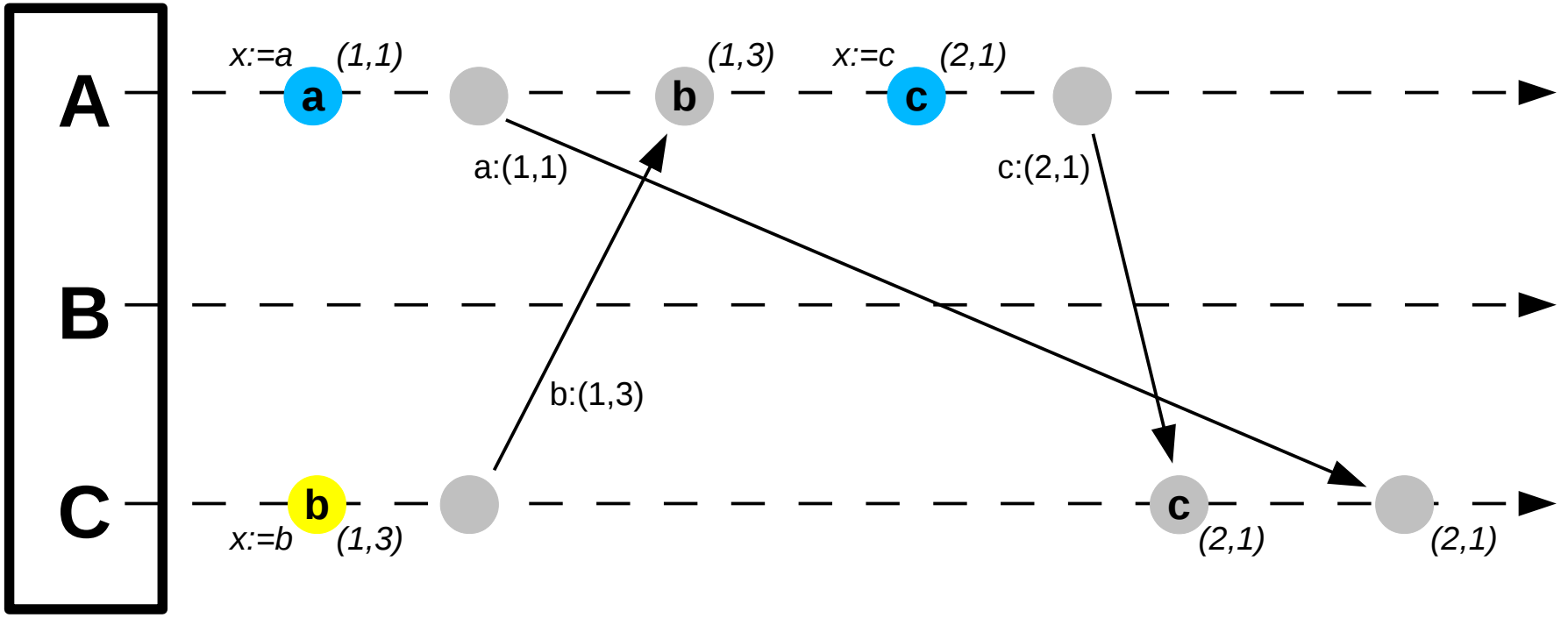
Dynamic total order:

- Requires synchronization (consensus), but
- … it is off the critical path (availability OK).
- Consensus may lead to rollbacks (expensive, confusing).

Decentralized (local) conflict resolution:

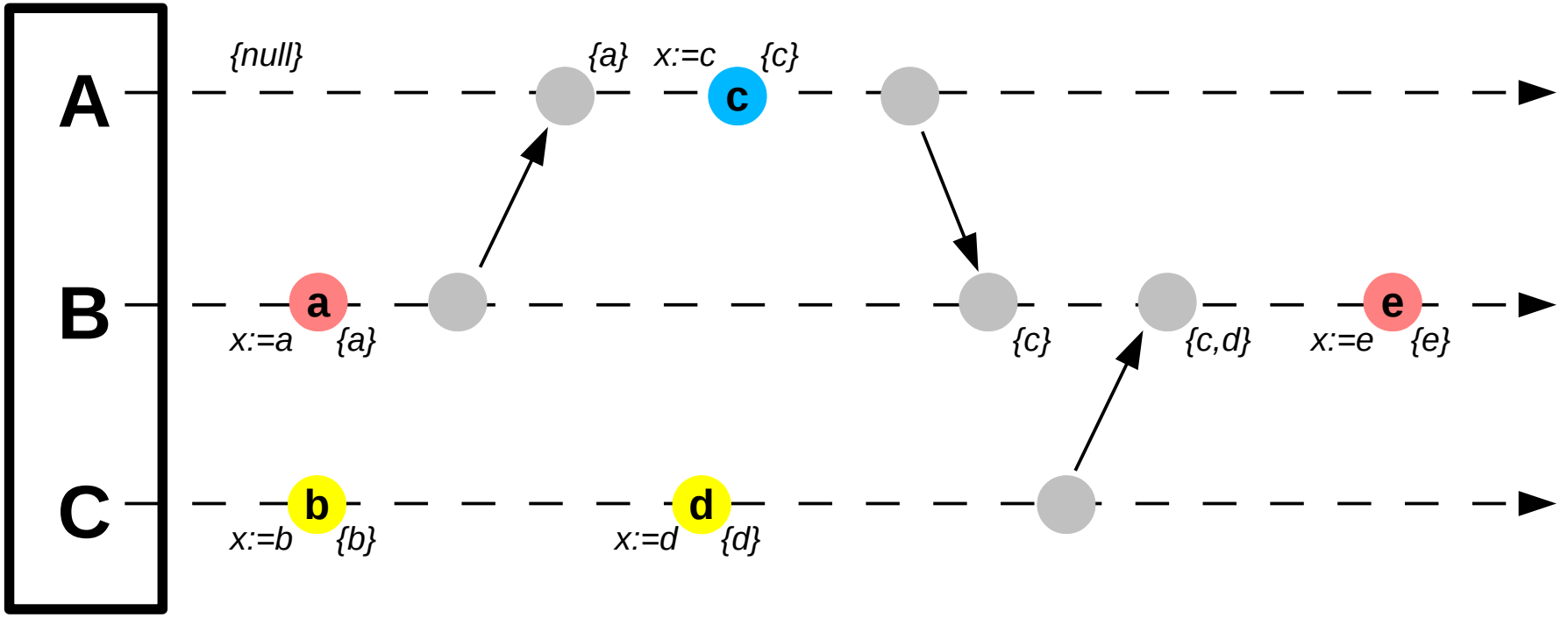
- No synchronization.
- Convergence conditions:
  - Deterministic
  - Depends on delivery set and not on
  - Delivery order nor local info, etc.

Example: Last Writer Wins (LWW)



Update: overwrite and stamp (unique, monotonic).
Transmit: data and timestamp.
Merge: the highest timestamp wins.

Example: Multi-value register



Sequential: normal register semantics.
Concurrent: set multiple values.
Read returns a set of values.

Example: Two-phase set (2P-Set)

Representation:

- A = a set of added items
- T = a set of tombstones (removed items)

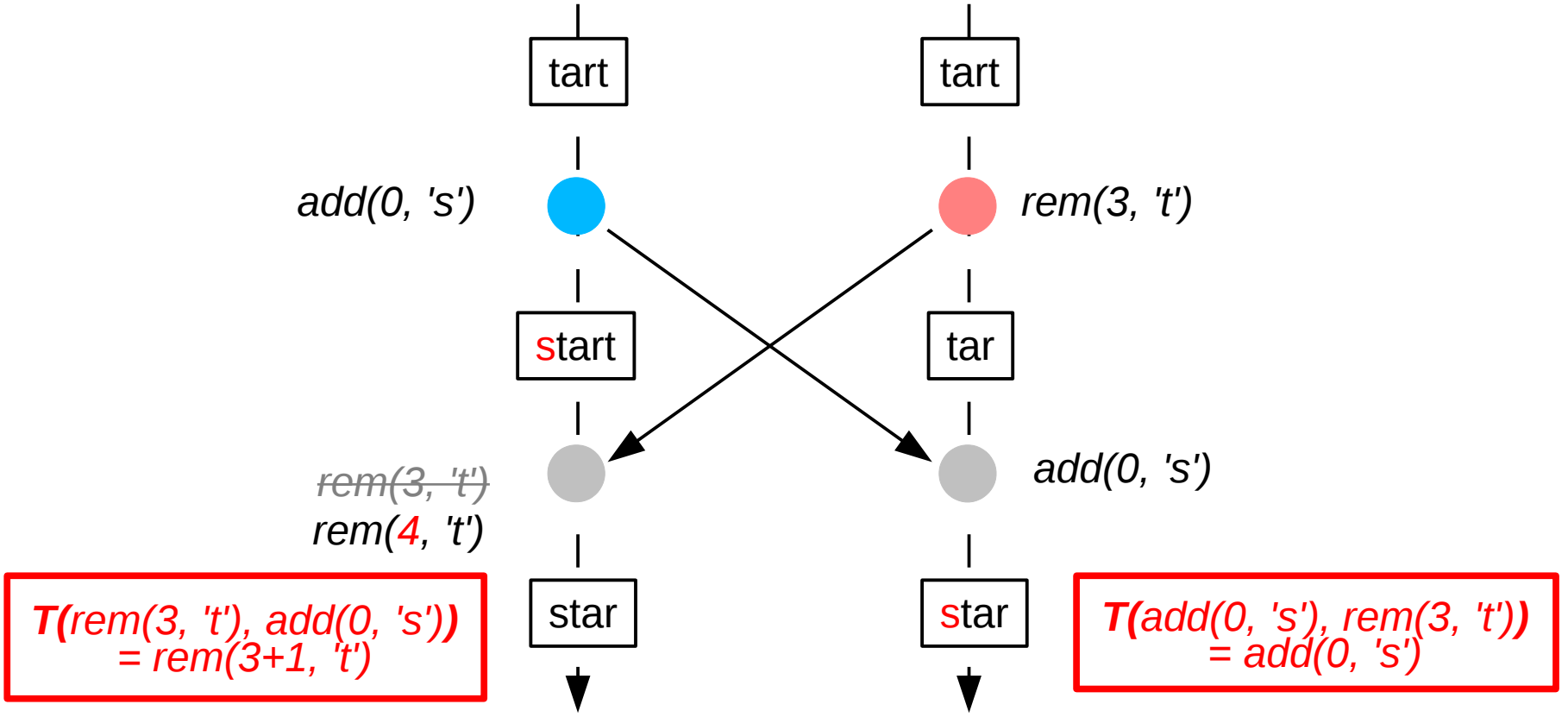
User operations:

- add(x) = $A := A \cup \{x\}$
- rem(x) = $T := T \cup \{x\}$
- has(x) = $x \in A \setminus T$?

Merge:

- $A := A_1 \cup A_2$
- $T := T_1 \cup T_2$

Example: Operational transformation (OT)



Transform received operations according to previously applied operations.

Ensures convergence in any order.

General requirements

General requirements

- Every update eventually reaches every replica at least once

General requirements

- Every update eventually reaches every replica at least once
- An update has effect on a replica at most once

General requirements

- Every update eventually reaches every replica at least once
- An update has effect on a replica at most once
- At all times, the state of the replica satisfies invariants

General requirements

- Every update eventually reaches every replica at least once
- An update has effect on a replica at most once
- At all times, the state of the replica satisfies invariants
- Two replicas that have received the same set of updates eventually reach the same state

State-based convergence criteria

Safety:

- Never go backwards => partial order, monotonic
- Apply each update once => m() idempotent
- Merge in any order => m() commutative
- Merge can involve many updates => m() associative

State-based convergence criteria

Safety:

- Never go backwards => partial order, monotonic
- Apply each update once => m() idempotent
- Merge in any order => m() commutative
- Merge can involve many updates => m() associative

Formally:

- m(s1,s1)=s1
- m(s1,s2)=m(s2,s1)
- m(s1,m(s2,s3)) = m(m(s1,s2),s3)

State-based convergence criteria

Safety:

- Never go backwards => partial order, monotonic
- Apply each update once => m() idempotent
- Merge in any order => m() commutative
- Merge can involve many updates => m() associative

Formally:

- m(s1,s1)=s1
- m(s1,s2)=m(s2,s1)
- m(s1,m(s2,s3)) = m(m(s1,s2),s3)

Semi-lattice (merge = smallest upper bound)

State-based convergence criteria

Example: 2P-Set

- $(A_1,T_1) \leq (A_2,T_2)$ iff $A_1 \subset A_2$ and $T_1 \subset T_2$.
- $m((A_1,T_1), (A_2,T_2)) = (A_1 \cup A_2, T_1 \cup T_2)$

Op-based convergence criteria

Safety:

- Operations must commute and be idempotent

Liveness:

- Each operation must be eventually delivered to each replica

Op-based convergence criteria

Example: OT

- $op_1 \bullet T(op_2, op_1) \equiv op_2 \bullet T(op_1, op_2)$
- $T(op_3, op_1 \bullet T(op_2, op_1)) \equiv T(op_3, op_2 \bullet T(op_1, op_2))$

Sample transformation:

- T(*add(p_1,c_1,r_1)*, *add(p_2,c_2,r_2)*) :-
  - If $p_1 < p_2$ then *add(p_1,c_1,r_1)*
  - Else if $p_1 = p_2$ and $r_1 < r_2$ then *add(p_1,c_1,r_1)*
  - Else *add(p_1+1,c_1,r_1)*

Op-based convergence criteria

Safety:

- Operations must commute and be idempotent

Liveness:

- Each operation must be eventually delivered to each replica

Op-based convergence criteria

Safety:

- Operations must commute and be idempotent

Liveness:

- Each operation must be eventually delivered to each replica

What if we have exactly once delivery?

Op-based convergence criteria

Safety:

- Operations must commute ~~and be idempotent~~

Liveness:

- Each operation must be eventually delivered to each replica exactly once

Op-based convergence criteria

Safety:

- Operations must commute ~~and be idempotent~~

Liveness:

- Each operation must be eventually delivered to each replica exactly once

What if we have causal delivery?

Op-based convergence criteria

Safety:

- Concurrent operations must commute ~~and be idempotent~~

Liveness:

- Each operation must be eventually delivered to each replica exactly once preserving causality

Conflict-free replicated data type (CRDT)

- Replicated at multiple machines
- Conflict-free:
  - Update without coordination
  - Decentralized resolution
- Data type:
  - Encapsulation
  - Well-defined interface

Eventual convergence by construction.

Conflict-free replicated data type (CRDT)

Register

- LWW
- Multi-value

Set:

- Grow-only
- Two phase
- Observed remove

Map

File system tree

Counter

- Unlimited
- Non-negative

Graph

- Directed
- Monotonic DAG
- Edit graph

Sequence

Conflict-free replicated data type (CRDT)

They are already being supported, for example, in NoSQL databases.

Client perspective

Sample scenario:

- A client is bound to replica A doing reads and updates.
- Replica A crashes or the client changes its location.
- The client continues, but bound to replica B.

Possible strange behavior observable by the client:

Client perspective

Sample scenario:

- A client is bound to replica A doing reads and updates.
- Replica A crashes or the client changes its location.
- The client continues, but bound to replica B.

Possible strange behavior observable by the client:

- The client's updates at A may not have yet been propagated to B.

Client perspective

Sample scenario:

- A client is bound to replica A doing reads and updates.
- Replica A crashes or the client changes its location.
- The client continues, but bound to replica B.

Possible strange behavior observable by the client:

- The client's updates at A may not have yet been propagated to B.
- The client may be operating at B on fresher/different entries than those at A.

Client perspective

Sample scenario:

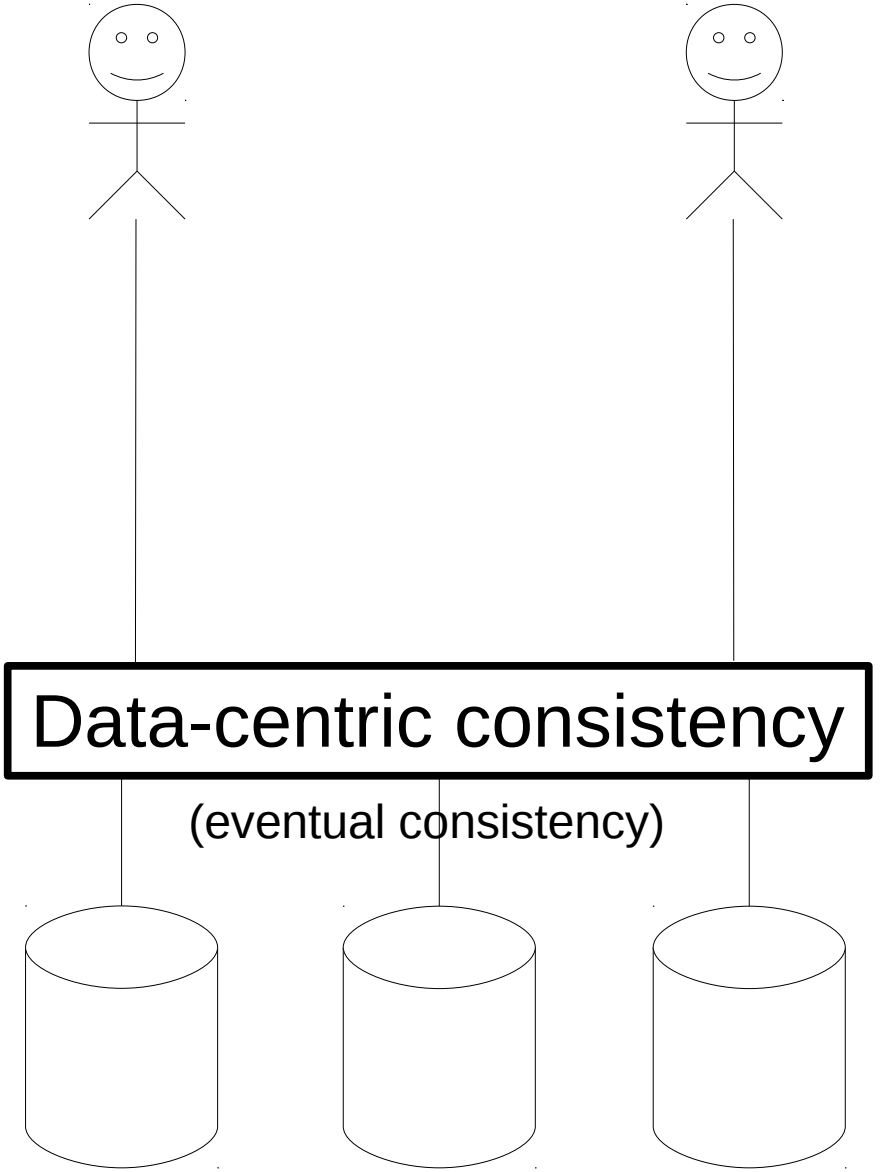
- A client is bound to replica A doing reads and updates.
- Replica A crashes or the client changes its location.
- The client continues, but bound to replica B.

Possible strange behavior observable by the client:

- The client's updates at A may not have yet been propagated to B.
- The client may be operating at B on fresher/different entries than those at A.
- The client's updates at A conflict with these at B.

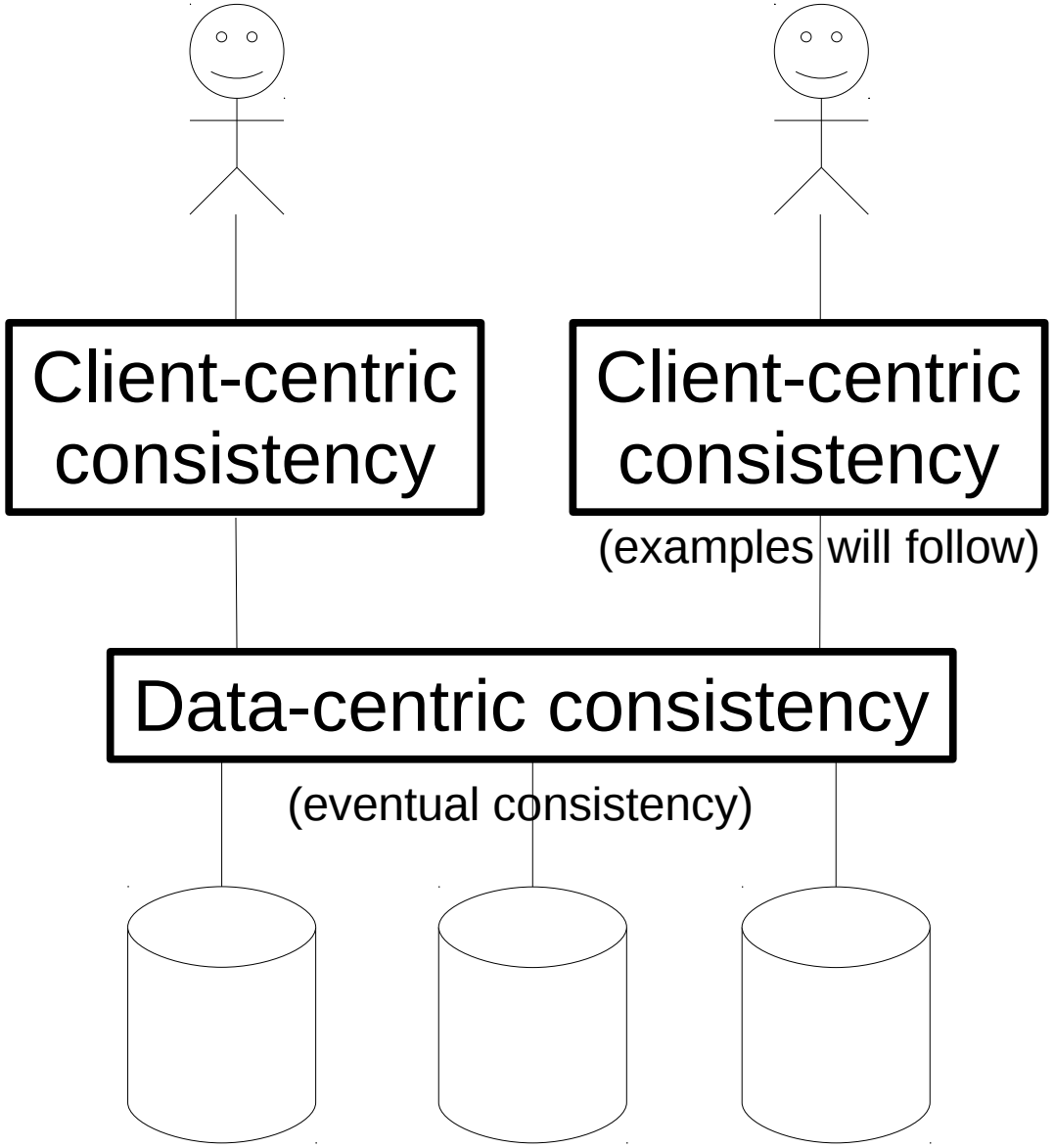
Client perspective

Describes how
the **entire system**
appears to any client.



Client perspective

Describes how
a **single client**
perceives the system.

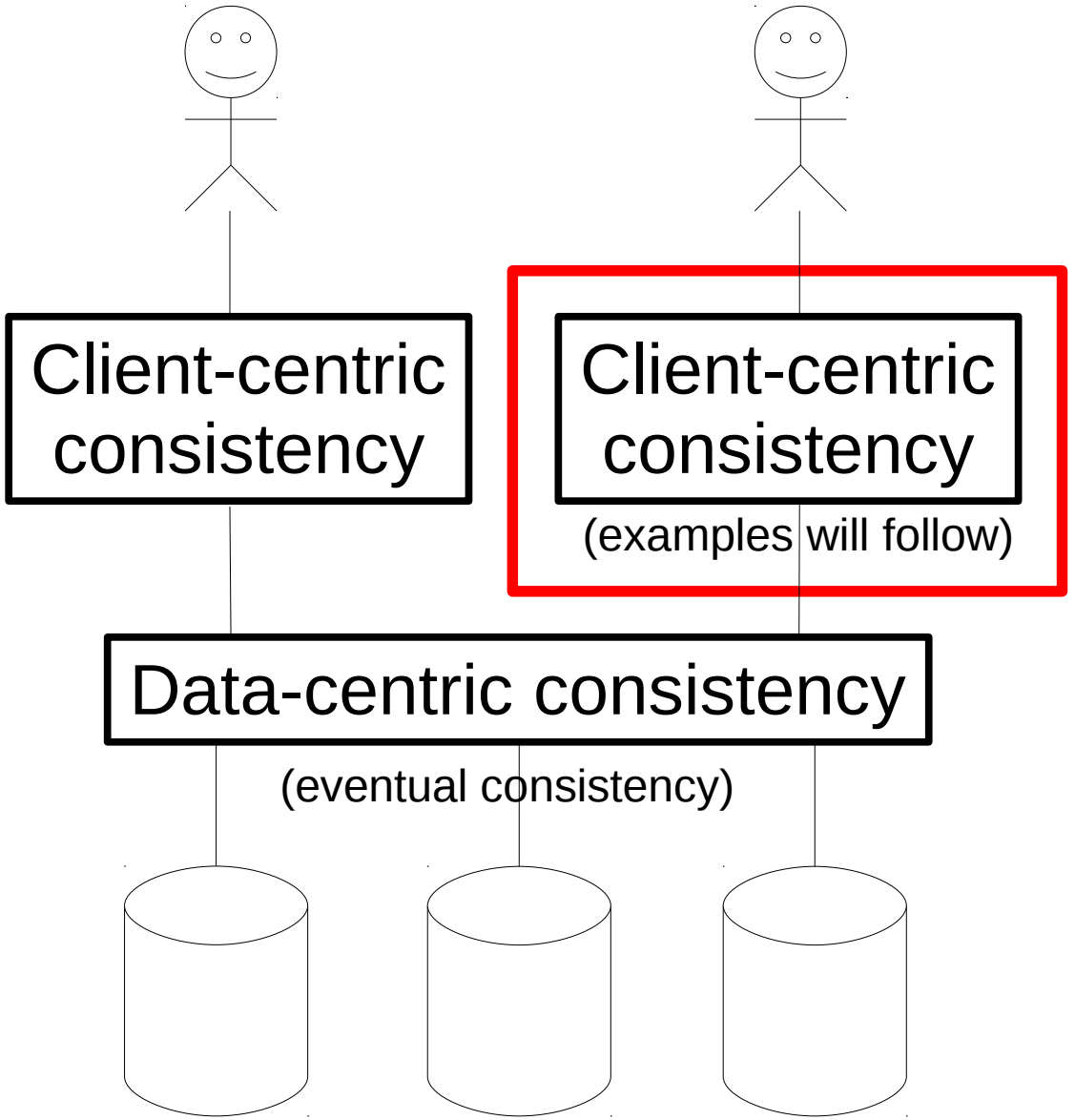


Describes how
the **entire system**
appears to any client.

Client perspective

Describes how
a **single client**
perceives the system.

Describes how
the **entire system**
appears to any client.



Client-centric consistency

Monotonic reads

If a process reads the value of a data item x, any successive read operation on x by that process will always return that same or a more recent value.

Client-centric consistency

Monotonic reads

If a process reads the value of a data item x, any successive read operation on x by that process will always return that same or a more recent value.

Client-centric consistency

Monotonic reads

If a process reads the value of a data item x, any successive read operation on x by that process will always return that same or a more recent value.

Examples:

- Reading a news website.
- Reading (but not commenting) somebody's blog.

Client-centric consistency

Monotonic writes

A write operation by a process on a data item x is completed before any successive write operation on x by the same process.

Client-centric consistency

Monotonic writes

A write operation by a process on a data item x is completed before any successive write operation on x by the same process.

Client-centric consistency

Monotonic writes

A write operation by a process on a data item x is completed before any successive write operation on x by the same process.

Examples:

- Pushing code commits from a local repository to a remote one.
- Editing a private text document online.

Client-centric consistency

Read your writes

The effect of a write operation by a process on data item x will always be seen by a successive read operation on x by the same process.

Client-centric consistency

Read your writes

The effect of a write operation by a process on data item x will always be seen by a successive read operation on x by the same process.

Client-centric consistency

Read your writes

The effect of a write operation by a process on data item x will always be seen by a successive read operation on x by the same process.

Examples:

- Changing a password.
- In an web shop, proceeding to the checkout after putting items into an cart.

Client-centric consistency

Writes follow reads

A write operation on data item x following a previous read operation on x by the same process, is guaranteed to take place on the same or a more recent value of x that was read.

Client-centric consistency

Writes follow reads

A write operation on data item x following a previous read operation on x by the same process, is guaranteed to take place on the same or a more recent value of x that was read.

Client-centric consistency

Writes follow reads

A write operation on data item x following a previous read operation on x by the same process, is guaranteed to take place on the same or a more recent value of x that was read.

Examples:

- Replying to posts on your FB Wall.
- Marking and tagging received e-mails.

Implementing CC consistency

- Implementation requires active participation from a client.
- Each write is given a unique identifier:
  - e.g., a combination of the origin replica identifier and a sequence number.
- A client maintains two sets:
  - A read set: A set of writes relevant for the read operations performed by the client.
  - A write set: Like above but relevant for writes.

Implementing CC consistency

- Implementation requires active participation from a client.
- Each write is given a unique identifier:
  - e.g., a combination of the origin replica identifier and a sequence number.
- A client maintains two sets:
  - A read set: A set of writes relevant for the read operations performed by the client.
  - A write set: Like above but relevant for writes.

Is keeping only the identifiers sufficient?

Implementing CC consistency

Example: writes follow reads:

- Before performing any write on a replica, make sure that all writes in your read set have been performed by the replica.

Implementing CC consistency

Example: writes follow reads:

- Before performing any write on a replica, make sure that all writes in your read set have been performed by the replica.

Problems?

Implementing CC consistency

- Sessions.
- Compressing the sets.
- Garbage collecting the sets.

Conclusions

Strong consistency

- A
- C
- I
- D

Weak consistency

- B
- A
- S
- E

Conclusions

Strong consistency	Weak consistency
Atomicity	B
Consistency	A
Isolation	S
Durability	E

Conclusions

Strong consistency

- Atomicity
- Consistency
- Isolation
- Durability

Weak consistency

- Basic
- Availability
- Soft state
- Eventually consistent

Conclusions

Strong consistency

- Atomicity
- Consistency
- Isolation
- Durability

Weak consistency

- Basic
- Availability
- Soft state
- Eventually consistent

There is a lot of room for research
on eventual consistency.

Conclusions

Weak consistency

Pros:

- Superior performance over strong consistency.
- Far better scalability.

Conclusions

Weak consistency

Pros:

- Superior performance over strong consistency.
- Far better scalability.

Cons:

- Difficult to program.
- May be difficult to understand for users.

Based on

- Nuno Preguiça and Marc Shapiro, “From strong to eventual consistency: getting it right,” A tutorial at OPODIS 2013.
- Dong Wang, “CAP Theorem,” Lecture slides for CSE 40822-Cloud Computing-Fall 2014.
- IEEE Computer, Volume 45, Issue 2, February 2012.
- Andrew Tanenbaum and Maarten van Steen, “Distributed Systems: Principles and Paradigms,” Second Edition, Prentice Hall, 2007.