CLOUD COMPUTING

REPLICATION & CONSISTENCY

Lorenzo Leonini
lorenzo.leonini@unine.ch

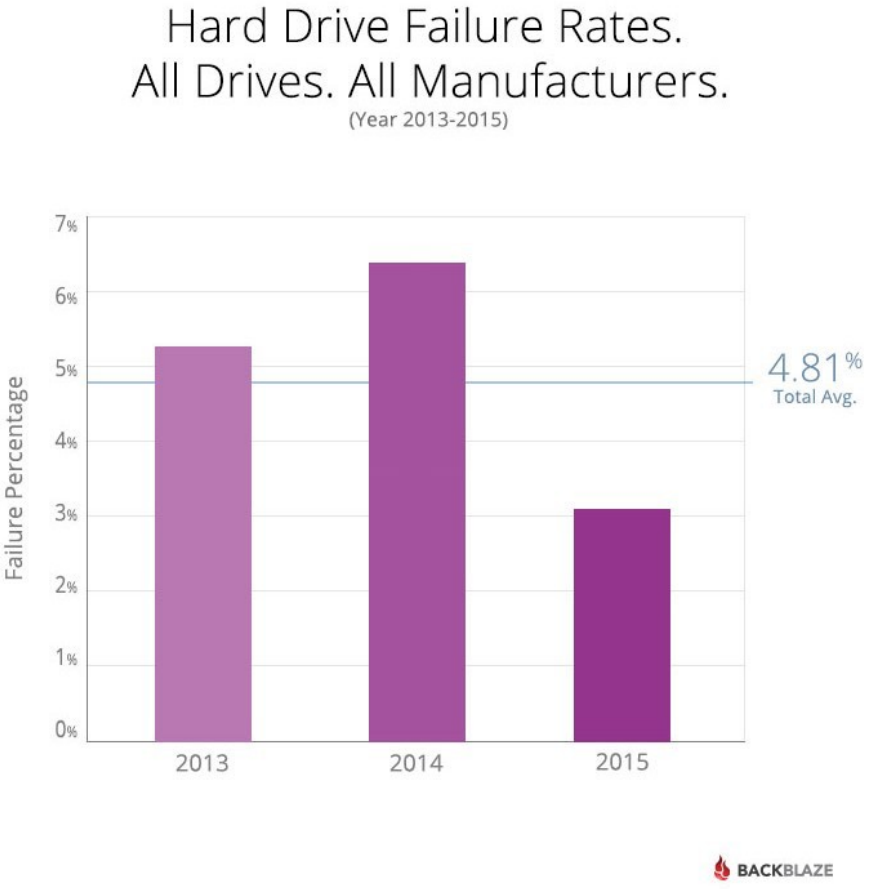
Course objectives

- Discuss fault tolerance and replication
- Understand that replication requires defining a semantic for concurrent data access
- Understand the concepts of consistency models, the CAP impossibility result and resulting engineering choices
- Discover the Raft consensus algorithm

Replication and consistency

Need for fault tolerance

- All parts used in datacenters can fail
  - Servers fail:
    - hardware problems, software problems
    - Disks fail
      - Study by Backblaze, Inc. on near to 50,000 disks
  - Humans fail
    - Switching off a server without notice



Need for fault tolerance

- A data center as a whole can fail
  - Problems with the power supply (outage longer than UPS units can deliver)
  - Intern would like to know what the little red button on the power system does
- How can we preserve the availability of stored data in the face of failures?
- Store multiple copies of the same data
  - In the same data center: survive fault/unavailability of one or more server(s)

OVH - 2017-11-09

The problems would have started at the datacenter of Strasbourg (SBG) with a simultaneous cut of the two electrical arrivals of 20kV, certain chains of generators supposed to be prepared for this eventuality having not started. Two rooms are particularly affected (SBG1 and SBG4), while generators feeding SGB2 work. The two routing chambers of the Strasbourg site (respectively in SGB1 and SGB2) are however inoperative, effectively isolating the datacenter from the rest of the web.

Murphy was decidedly in shape this morning, then OVH's entire European optical network joined the party: all the 100Gbps interconnections linking the data centers of Roubaix and Graveline to the various POPs (points of presence) fell at the same time, an almost improbable case. In practice, this means, for example, that the Roubaix datacenter is totally isolated since all its interconnections with TH2, London, Brussels, Frankfurt or Amsterdam are down.

Panne géante chez OVH, vous êtes au... courant ?

22 COMMENTAIRES

Quelqu'un a le mail de Murphy ?

par Yannick Guerrini 9 novembre 2017 09:42

Réveil difficile pour OVH : une panne géante touche l'hébergeur/FAI depuis tôt ce matin (7h14), avec comme conséquence plusieurs milliers de sites - petits ou gros, voire gouvernementaux - inaccessibles, mais également de nombreux autres services (téléphonie, Cloud, VPS, mails, serveurs de jeux...) impactés.

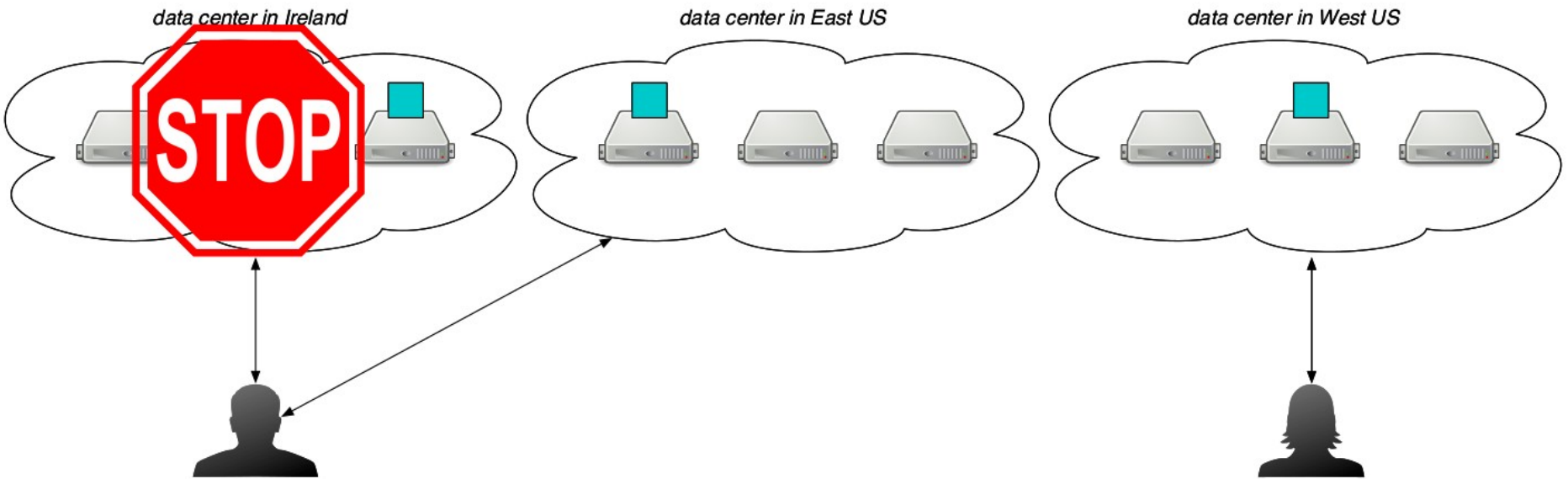
Grosse panique sur l'Internet européen

Les problèmes auraient débuté au datacenter de Strasbourg (SBG) avec **une coupure simultanée des deux arrivées électriques de 20kV**, certaines chaines de groupes électrogènes censés parer à cette éventualité ne s'étant pas mis en marche. Deux salles sont plus particulièrement touchées (SBG1 et SBG4), tandis que les générateurs alimentant SGB2 fonctionnent. Les deux chambres de routage du site de Strasbourg (respectivement dans SGB1 et SGB2) sont en revanche inopérantes, isolant de fait le datacenter du reste de la toile.

Murphy étant décidément en forme ce matin, c'est ensuite **tout le réseau optique européen d'OVH** qui s'est joint à la fête : toutes les interconnexions 100Gbps reliant les datacenters de Roubaix et Graveline aux différents POP (points de présence) sont tombées en panne simultanément, un cas de figure quasiment improbable. En pratique, cela signifie par exemple que le datacenter de Roubaix est totalement isolé puisque toutes ses interconnexions avec TH2, Londres, Bruxelles, Francfort ou Amsterdam sont en panne.

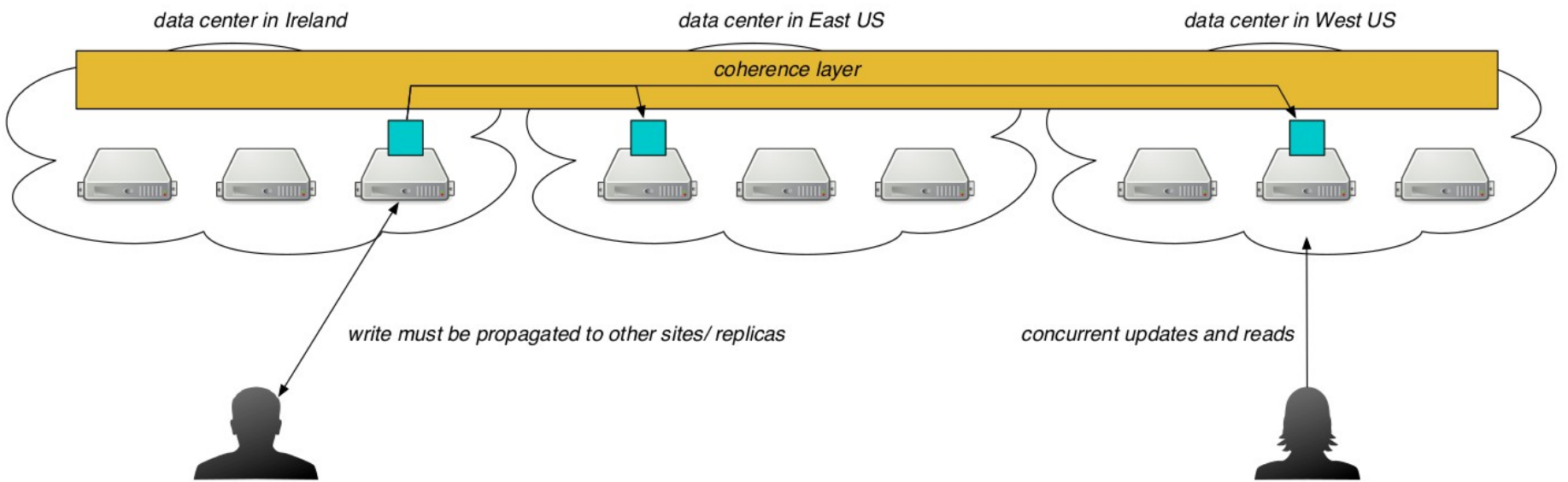
Georeplication

- Replicate across data centers: survive faults/unavailability of entire data center
  - Bonus: serve local users with local copy of data



Maintaining coherent copies

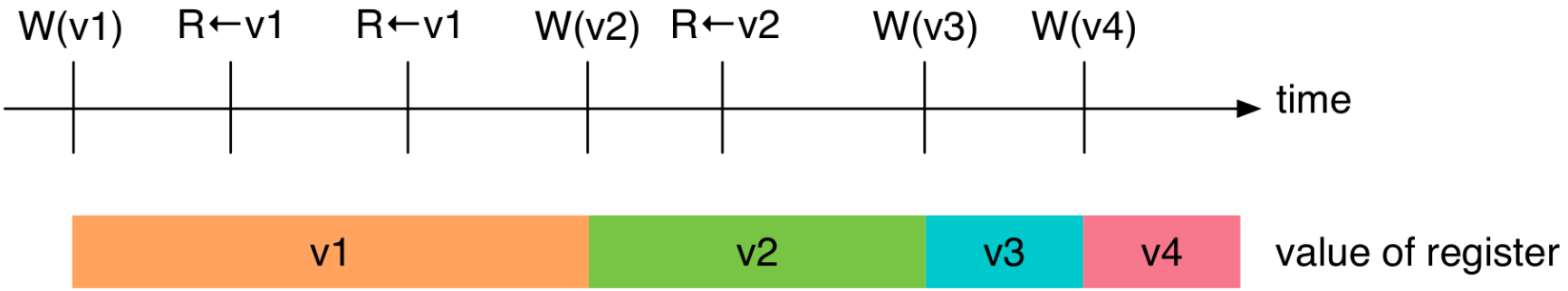
- Updates to one copy must be propagated to other copies
  - This is the role of a coherence mechanism
- But latencies between data centers can be important



Consistency

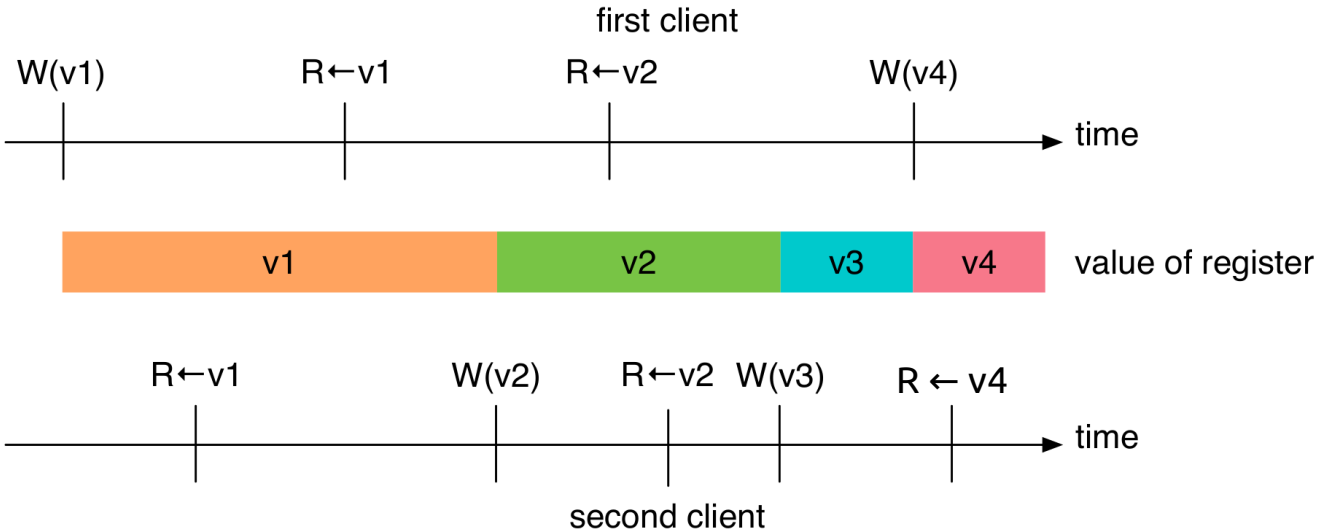
- Multiple copies of the same data means we have to deal with concurrent read and writes
  - How do we maintain coherent copies?
  - How do multiple clients see modifications to a data item?
    - Formalized as consistency properties
  - **A coherence protocol enforces a consistency property**
- Stronger consistency properties require more costly coherence protocols
  - Let’s start by defining the notion of consistency model

Single-process history



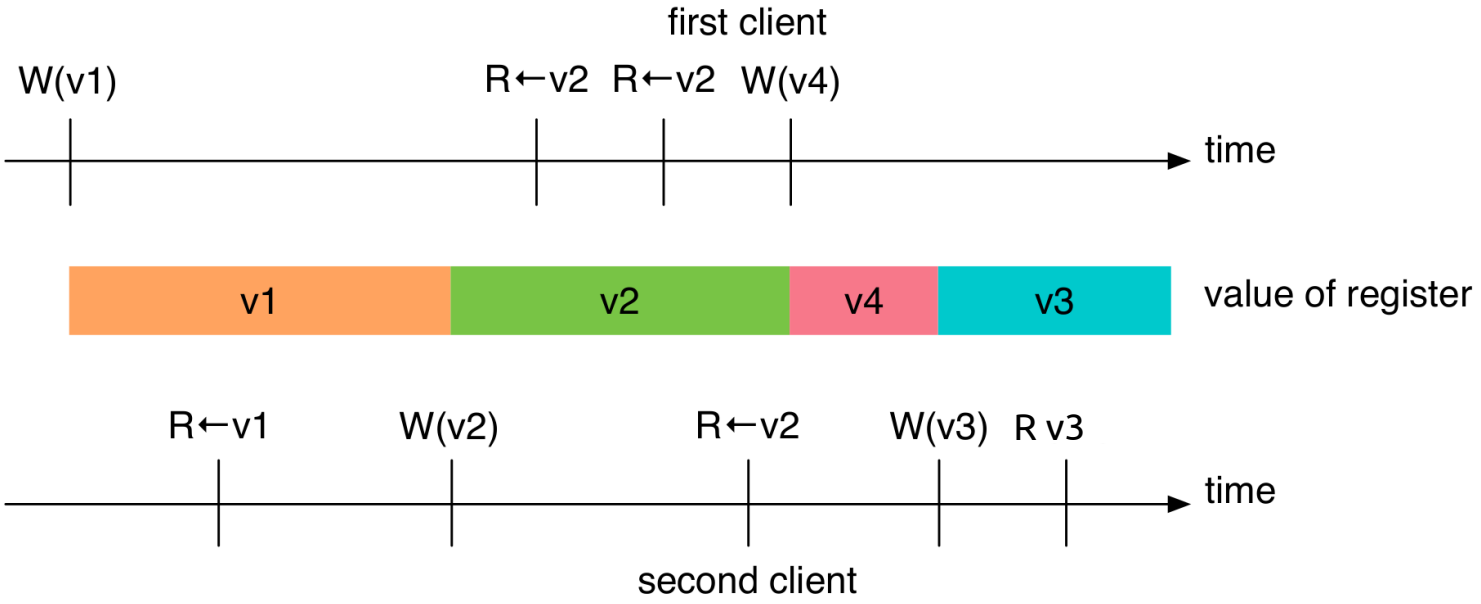
- Implicit rules expected by sequential programmer
  - Read returns the last value written
  - Single thread means single order of operations
- This is a consistency model
  - It defines the set of valid histories of operations and responses

Concurrent history (with instantaneous operations)



- Change: read does not return last value written by same client (1st client's 2nd read returns v2, not value v1)
- Relax: read operation returns last value written by any process at the moment it is called
  - History still respects the ordering of operations by each client

Concurrent history (with instantaneous operations)

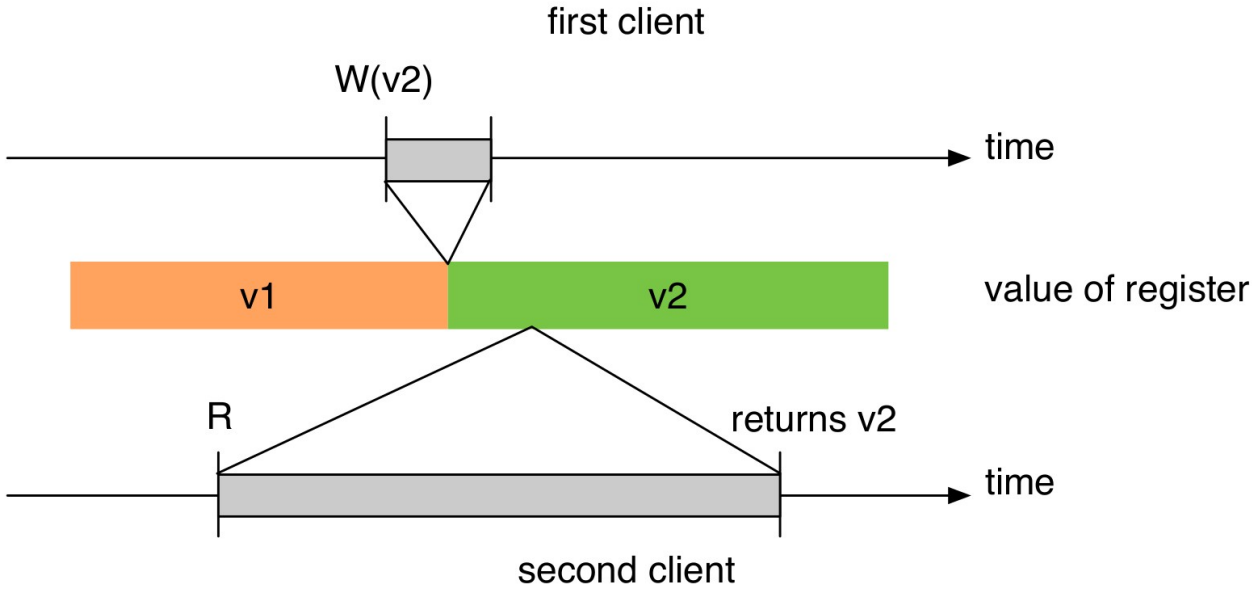


- This is another valid global history for the same local operations from the two clients
  - Accesses can just be scheduled differently (speed of processes, scheduler actions, etc.)

But operations are not instantaneous in practice!

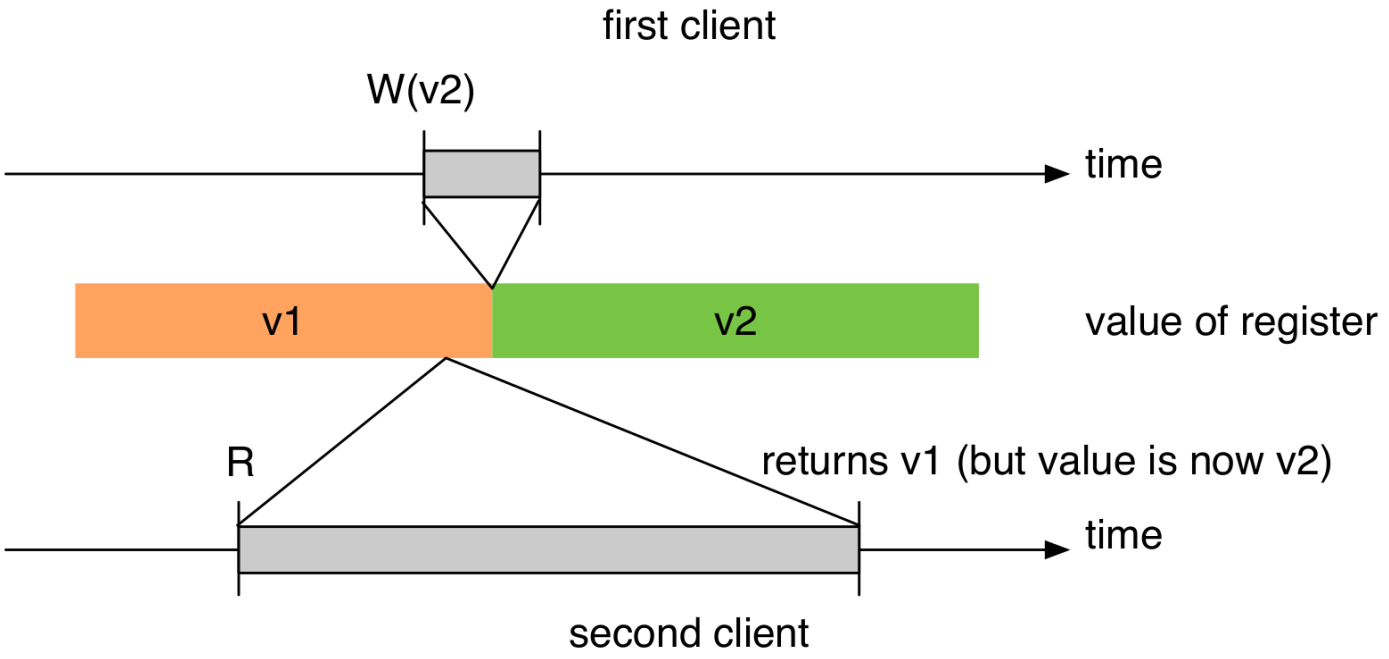
- Processes are distant to each other
  - Replicas in different clouds with georeplication
- Operations take time to be processed
  - Delay between sending the read/write operation and seeing the result
  - WAN links between clouds
- As a result, operations are interleaved in time

A non-coherent history (according to our previous consistency criteria)



- 2nd client's read does not return the value present when its read command has been initiated
  - Violates our consistency criteria

The symmetry

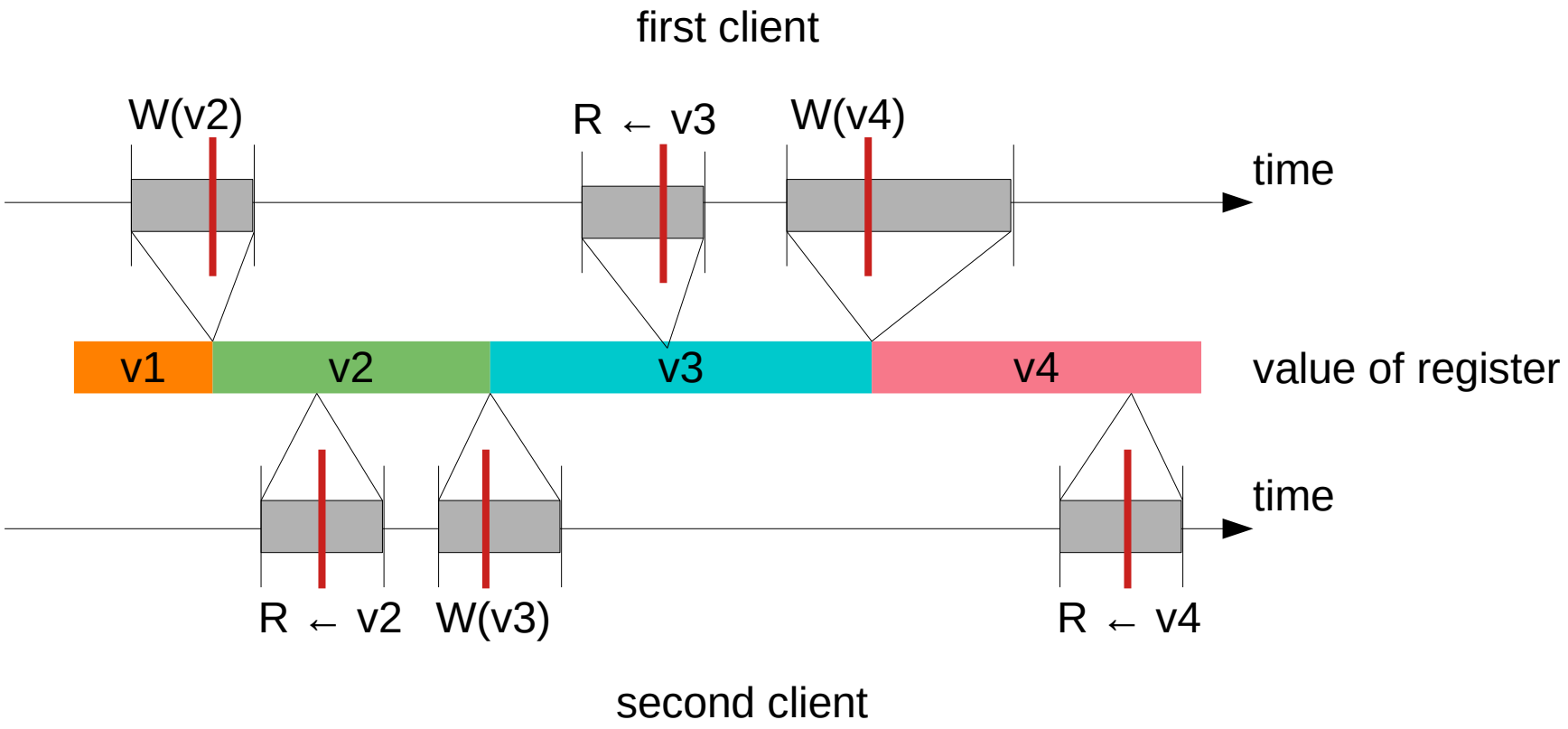


- We need a consistency criteria that is relaxed and allow reasoning about concurrent operations that take time to complete

Linearizability

- Assume each operation happens atomically at some time point between its invocation and its return
- Allows deciding on a total, linear order
  - **Every read starting later than a write returns the value of that write, or that of a later write**
  - Prohibits stale reads and non-monotonic reads
- A strong consistency model, and easy to reason about
  - Strongest memory model for distributed systems !

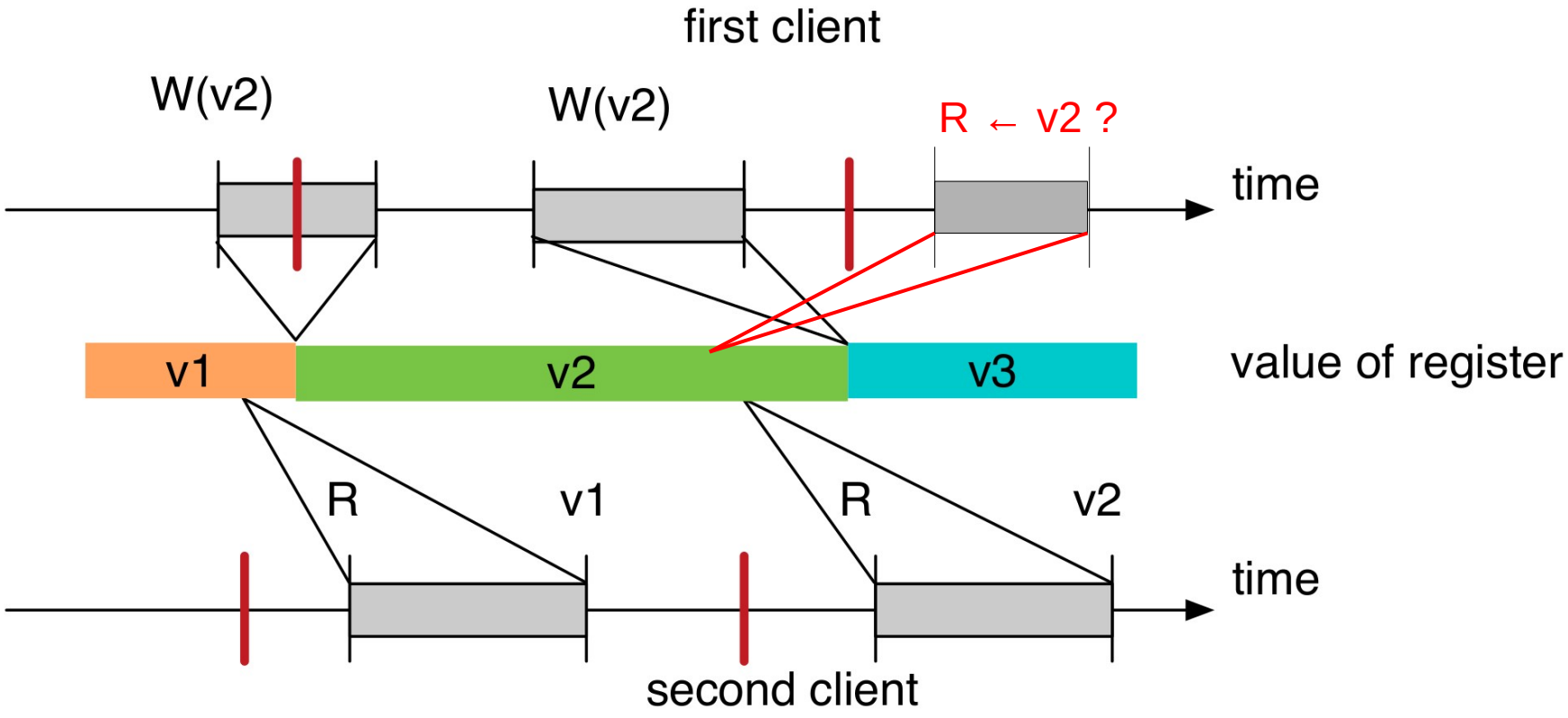
Linearizable history



Sequential consistency

- Allow operations to appear as if they took place atomically, but maybe before their invocation (reads) or after their completion (writes)
- Preserves the order of operations for each client
  - “Read your writes” semantics → reads/writes of a client cannot cross !
- All clients see the writes as appearing in the same order but does not require this order to respect the “real-time” order of invocations
  - Still forms a global, total ordering of operations
- Weaker than linearizability: allow more valid histories

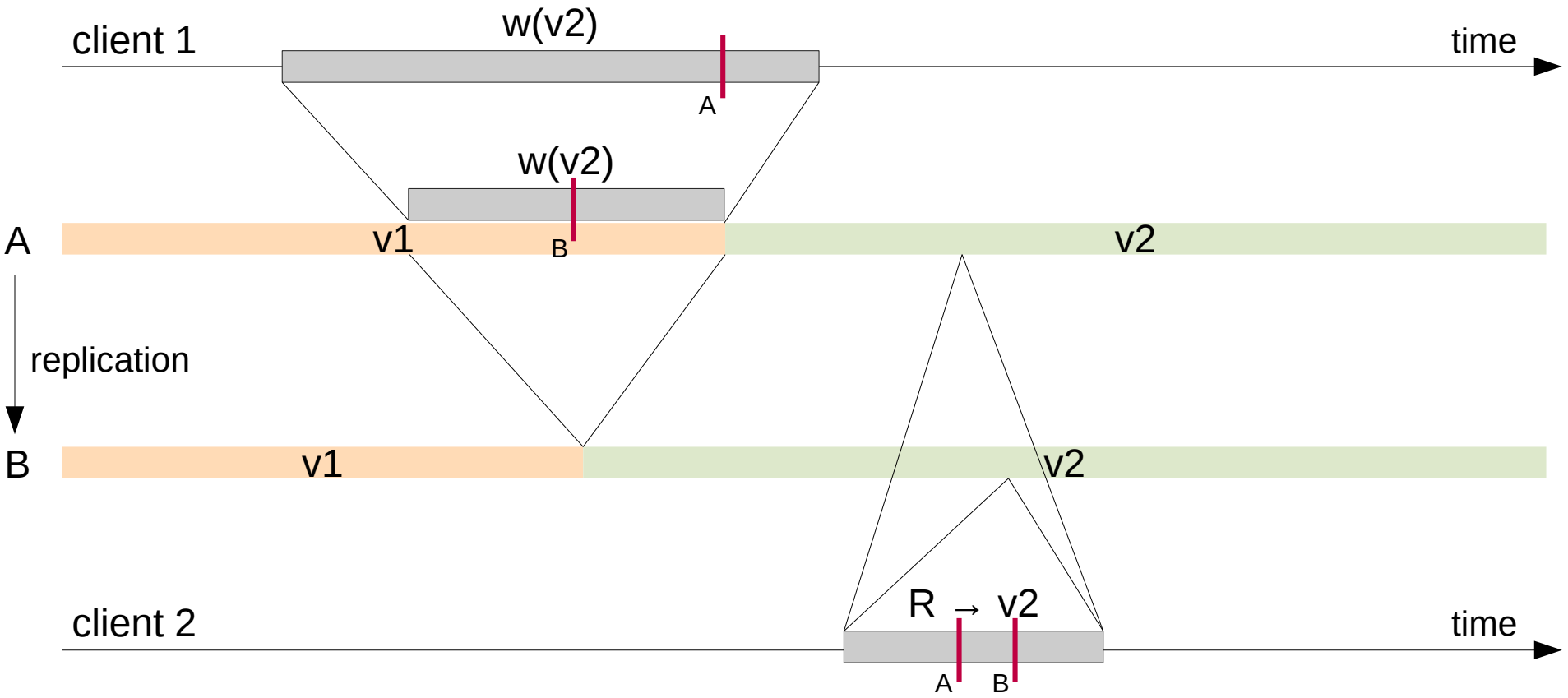
Sequentially consistent history



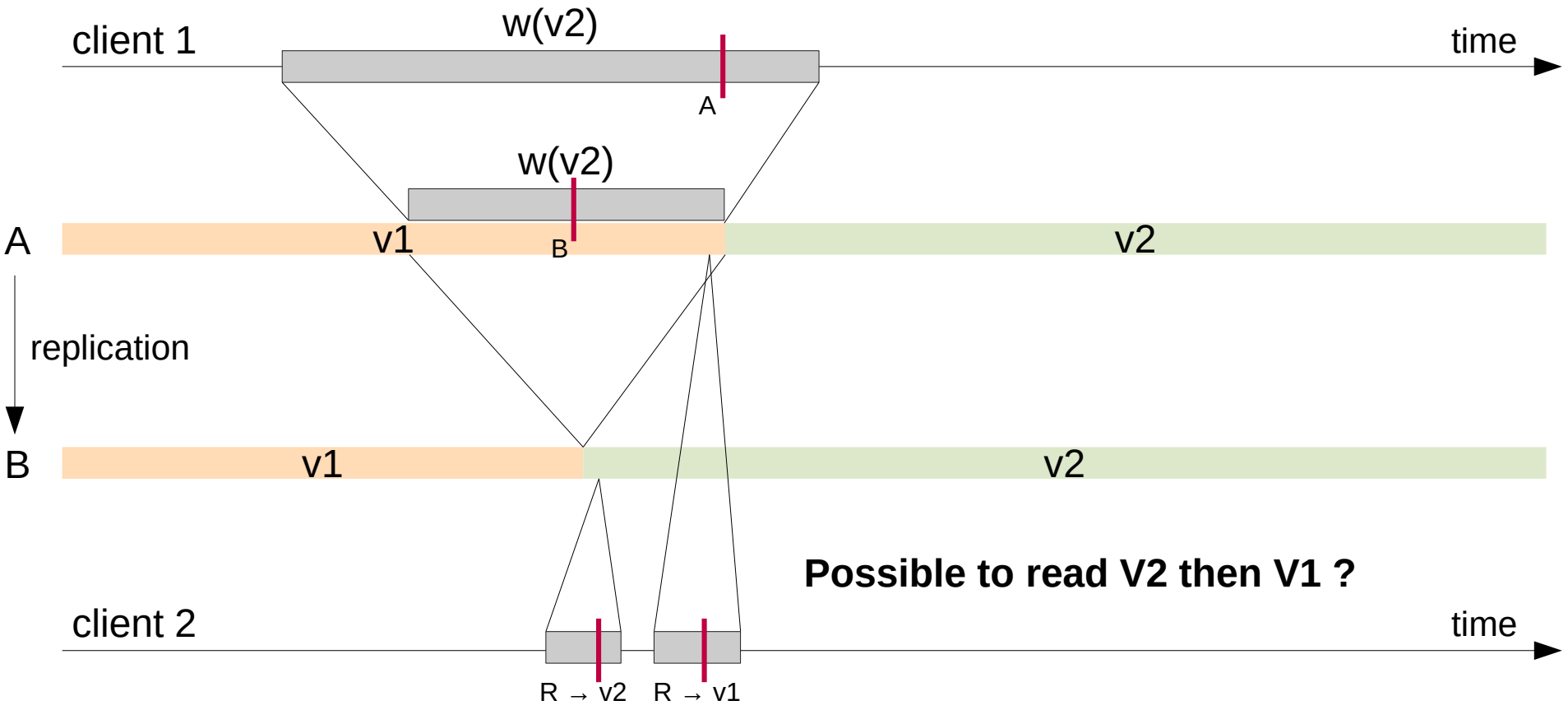
Composability

- Ability to combine individual operations and keep safety properties
  - Combining = accessing different objects
  - Safety = keeping a unique visible order of updates
- Sequential consistency is not *composable*
  - The combination of sequentially-consistent history is not necessarily sequentially consistent
  - Two processes reading a register from multiple replicas may not see the same order of writes
  - Order remains guaranteed if reading from same replica
- Linearizability is *composable*

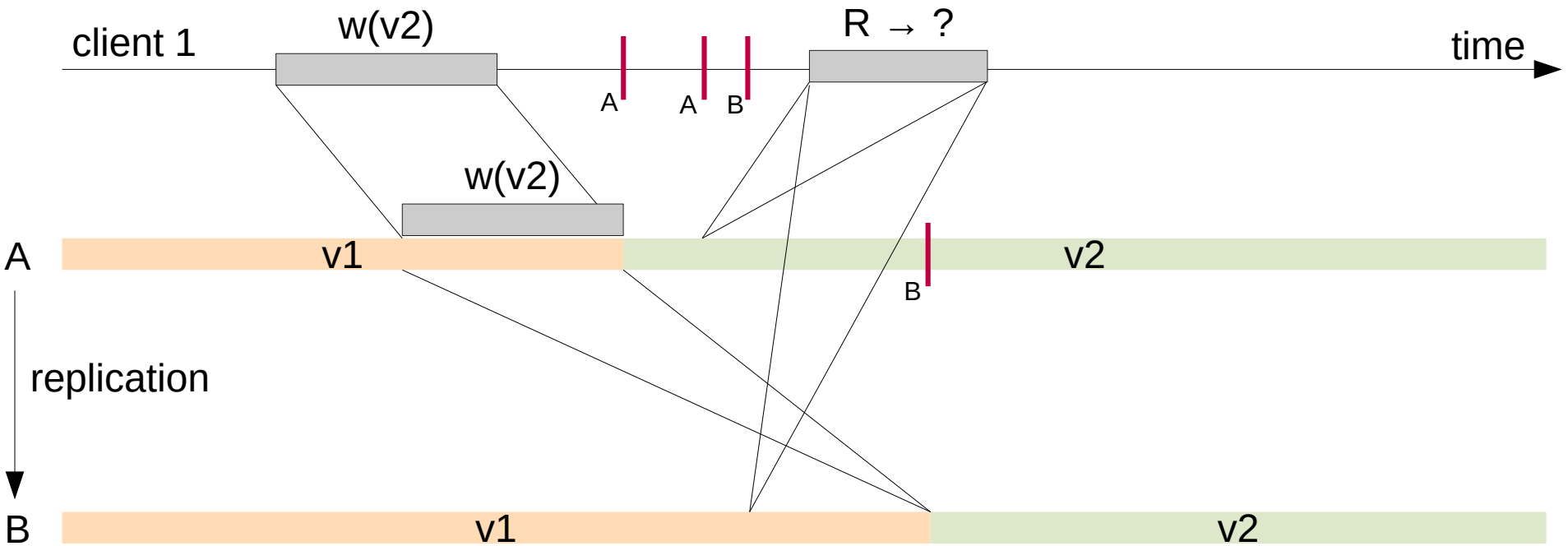
Linearizable composition (1)



Linearizable composition (2)



Sequential composition



Easy to read V2 then V1 (non-monotonic read) → not composable

Further relax consistency: Causal consistency

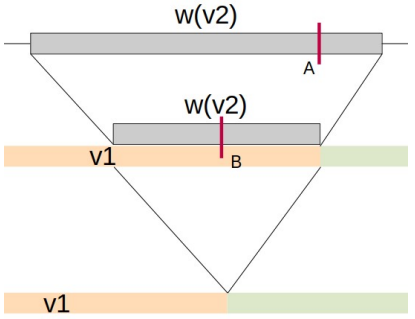
- Not all operations from one client must be ordered (seen by other clients) in the order they were issued ...
- ... only causally related operations should be ordered according to this client's order
  - A write *b* is causally related to a read *a* when *a* was performed by the client in the same thread (memory space) and this relation has been exposed by the client
  - Example: a comment to a blog post is causally dependent to the previous comments seen when reading the blog post page
- Writes that are not causally related might be seen in different orders by different clients
  - Unlike linearizability and sequential consistency that impose that all writes are seen in same order by all

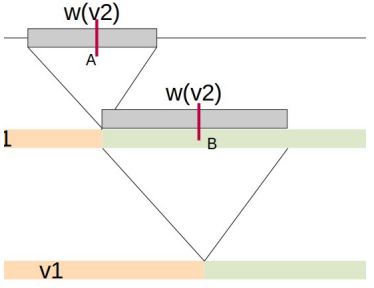
Implementing coherence

- Imposing order requires coordination
  - The more constraints on the order (the more histories we exclude), the more complex and costly is the coordination protocol
- Let’s discuss some implementation options
  - This is a large subject with a vast array of techniques and a vast literature
  - Approaches for other consistency models can often be seen as variants of these algorithms

Options for implementing coherence

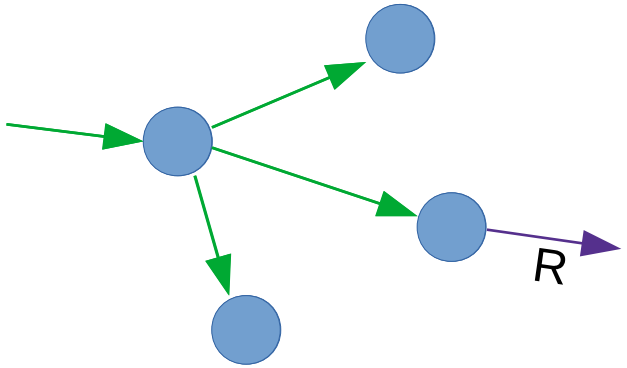
- When are replicas updated?
  - Synchronous replication
    - Immediately
    - Reply to client only after operation completes
  - Asynchronous replication
    - In the background, later
    - Reply to client immediately
- Algorithm and performance tradeoffs
  - Consistency model
  - Favor read performance vs write performance (or ⟷)
  - Performance under faults

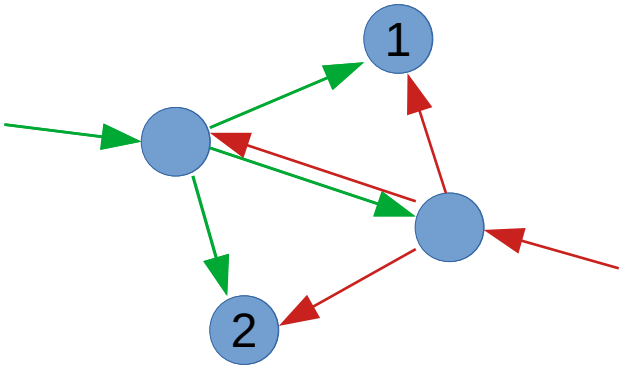




Linearizability: Synchronous, write-all (1)

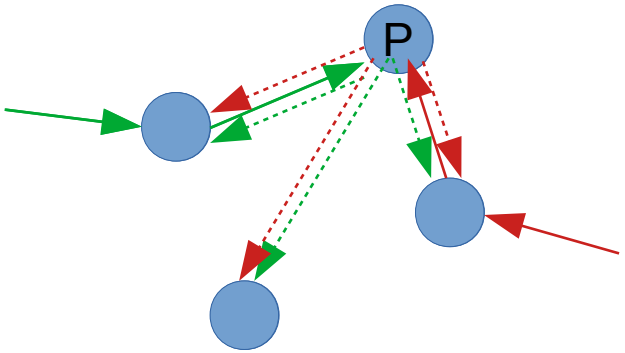
- Write goes to all replicas (slow), blocks if a single replica is faulty
- Reads served from local replica (fast)
- Must still order concurrent writes
  - or replicas will diverge, which will break consistency
  - example, concurrent W(v1) and W(v2)
    - replica 1 receives W(v1) then W(v2) — ends with v2
    - replica 2 receives W(v2) then W(v1) — ends with v1



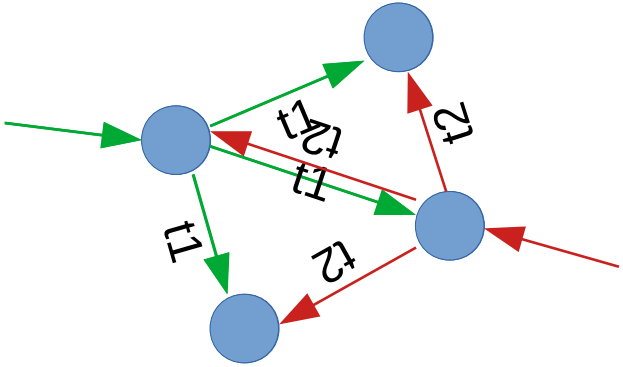


Linearizability: Synchronous, write-all (2)

- Solution 1: Primary-copy
  - Designate a primary replica, send it all writes. The primary broadcast all writes in the order it decides

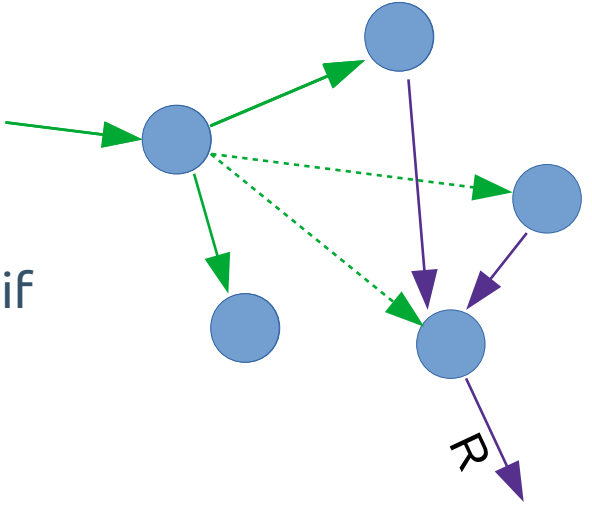


- Solution 2: Broadcast and use timestamps to order writes
  - W(v1) timestamp 1 and W(v2) timestamp 2
    - replica X receives W(v1,1) then W(v2,2) — ends with v2
    - replica Y receives W(v2,2), ignores W(v1,1) — ends with v2



Linearizability: Synchronous, write-quorum

- Write to a majority of replicas (slow)
- Reads from majority of replicas (slow)
  - Guaranteed to read one copy of “latest” written value
- Non blocking under failures of minority of replicas
- Must still order concurrent writes
- ~~Solution 1~~: designate a primary replica…
  - Not a good idea here: loose non-blocking advantage if the primary fails
- Solution 2: use timestamps to order writes

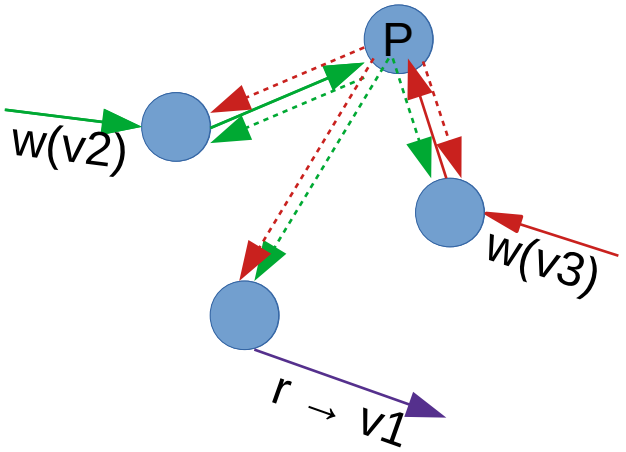


Asynchronous replication

- Operations first execute on local replica and reply sent immediately to client
- Then, updates sent to remaining replicas in the background
  - May loose updates if local replica fails before doing this
- How to deal with conflicting updates that are sent asynchronously to different replicas?

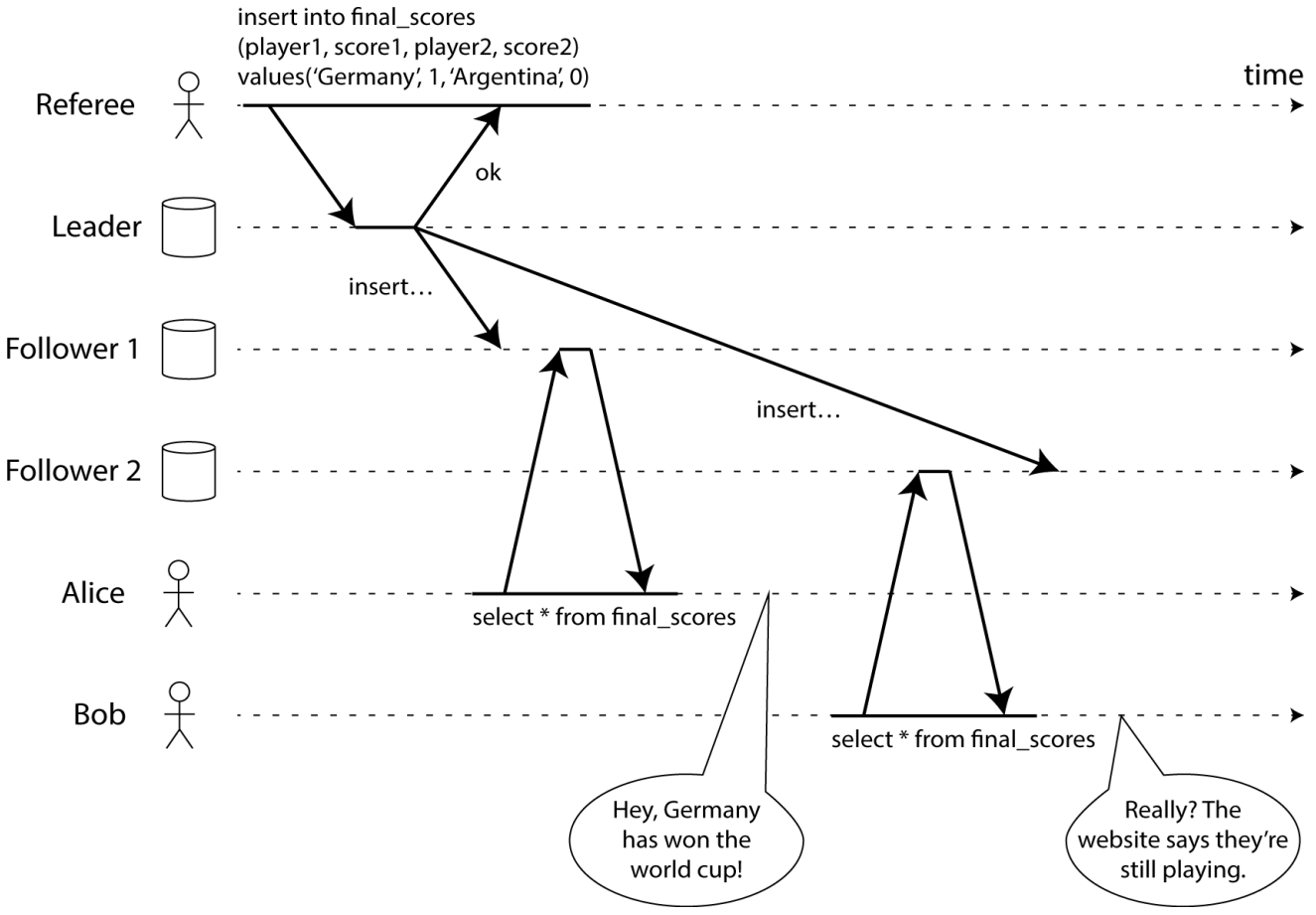
Sequential consistency: Asynchronous primary-copy

- One replica is the primary, prohibit updates at other replicas
  - Writes forwarded to primary, who broadcasts them in some order to all other replica
  - But this happens after replying to the client
- Reads done on local replica
  - Can return stale data
    - Update not yet received from primary
  - As a result, can only provide sequential consistency
  - Still OK for many applications!



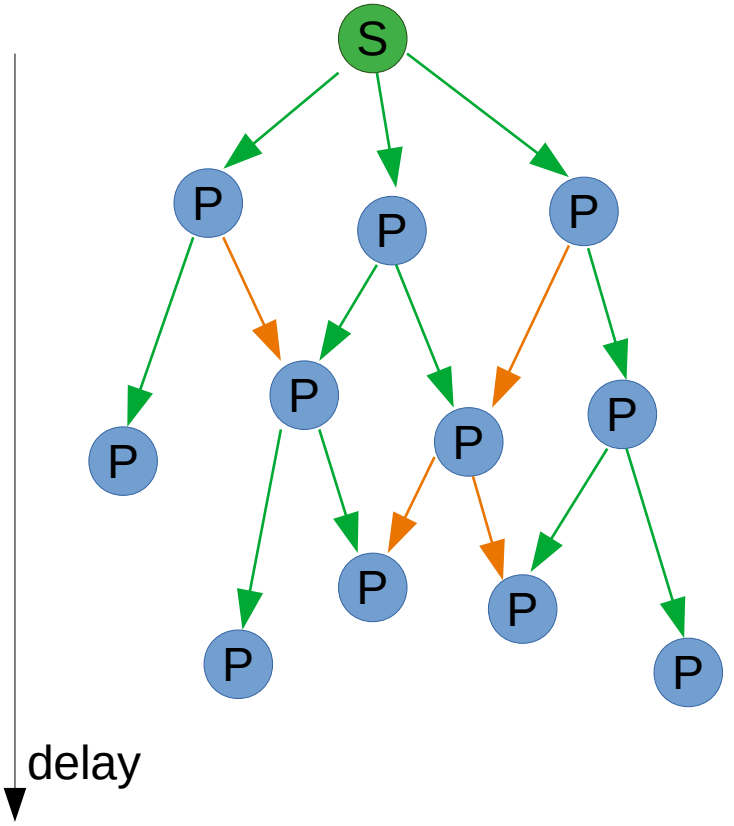
Sequential consistency - Example 1

- Master-Slave read replicas



Sequential consistency - Example 2

- P2P Video streaming
  - Some nodes send the stream to other nodes
  - Sequentiality: all nodes receive the frames in the same order
  - If nodes are receiving streams from multiple nodes sequentiality will be broken
    - However the nodes still will be able to re-order the frames (they are numbered) before playing

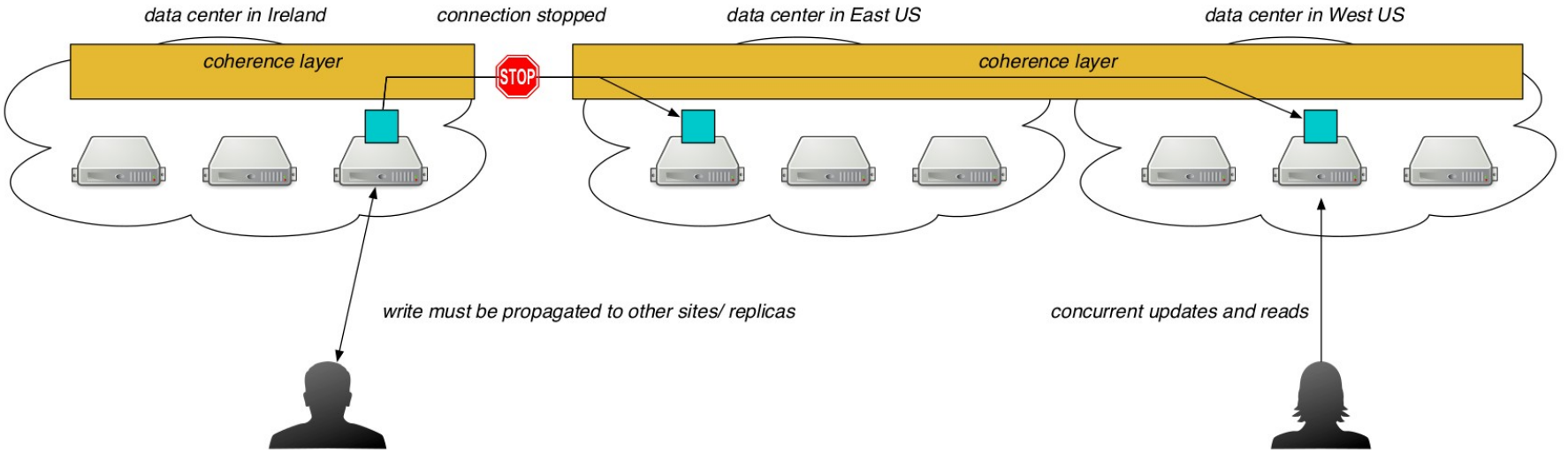


Availability and partitions

Availability and partitions

- Strong consistency models require updating replicas of the data to ensure order guarantees
- Two other criteria are important for a replicated cloud storage system
  - Availability
    - Is the system responsive within a bounded time to operations sent by clients?
  - Partition-tolerance
    - Can the system continue its operation despite arbitrary partitions due to network failures or disconnections?

Partition

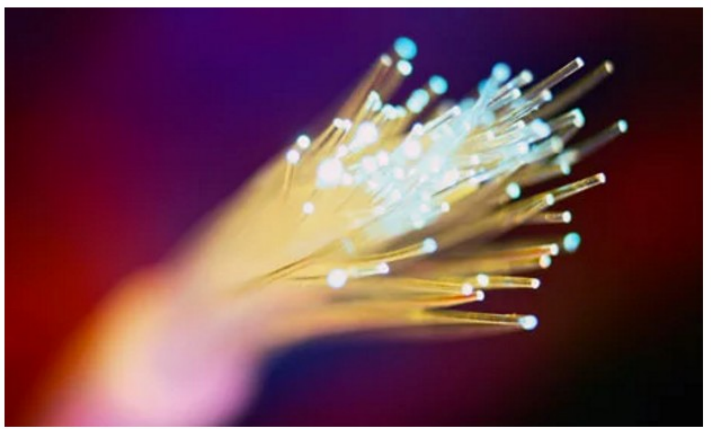


- Partitions between data centers happen in practice
  - Inside a data center, proper network fabric and link redundancy make them less stringent, but this is less the case between DC
  - Any situation where a part of the replicas are unreachable is a partition

A real example of partition

Georgian woman cuts off web access to whole of Armenia

Entire country loses internet for five hours after woman, 75, slices through cable while scavenging for copper



The woman damaged a fibre-optic cable with her spade

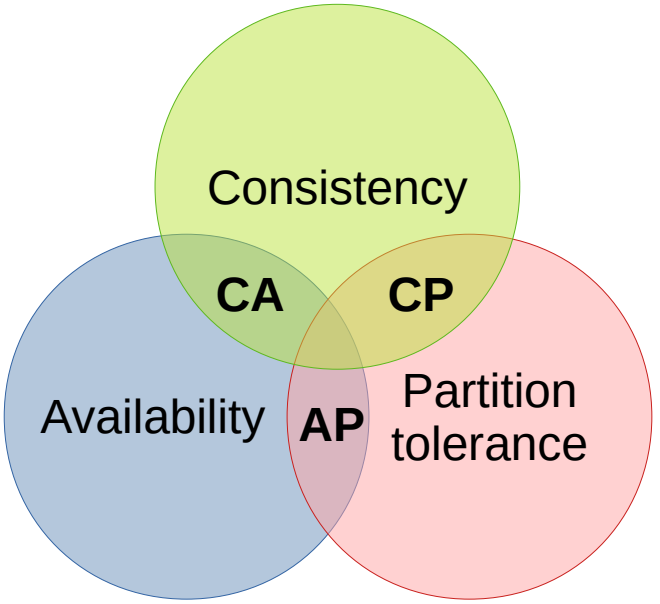
An elderly Georgian woman was scavenging for copper to sell as scrap when she accidentally sliced through an underground cable and cut off internet services to all of neighbouring Armenia, it emerged on Wednesday.

The woman, 75, had been digging for the metal not far from the capital Tbilisi when her spade damaged the fibre-optic cable on 28 March.

- More serious examples in http://www.bailis.org/papers/partitions-queue2014.pdf

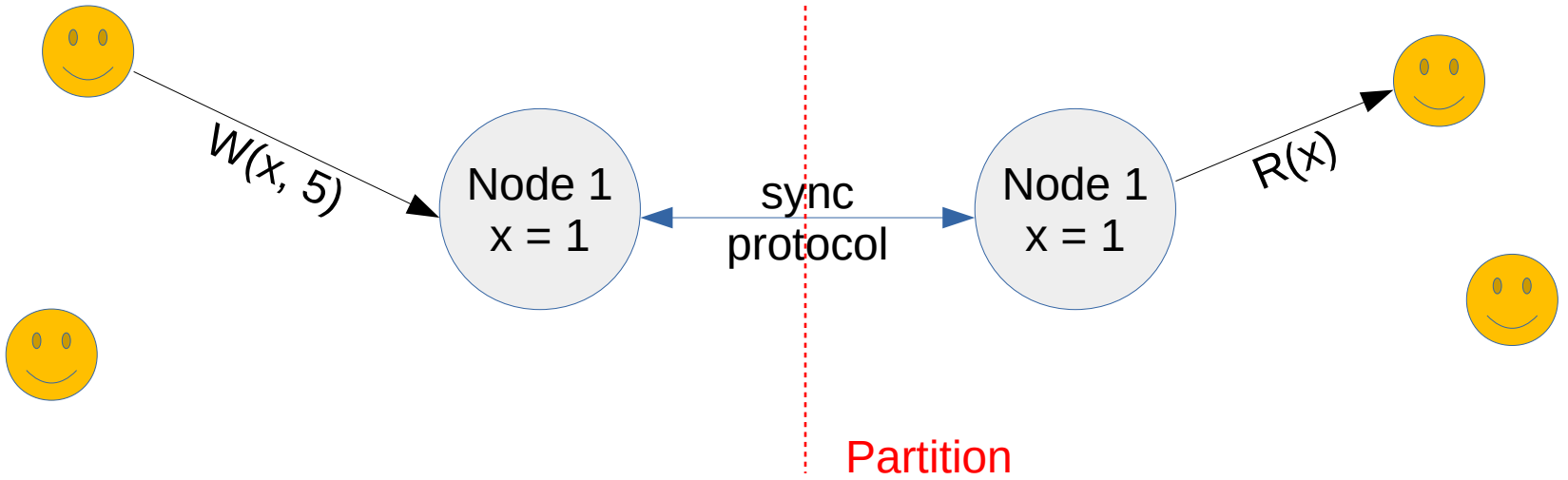
CAP Theorem

- **Properties**
  - **Consistent**
    - Every read receives the most recent write or an error
  - **Available**
    - Every request receives a response that is not an error
  - **Partition** tolerance



- **It's not possible to guarantee all 3 properties <u>at the same time</u> !**

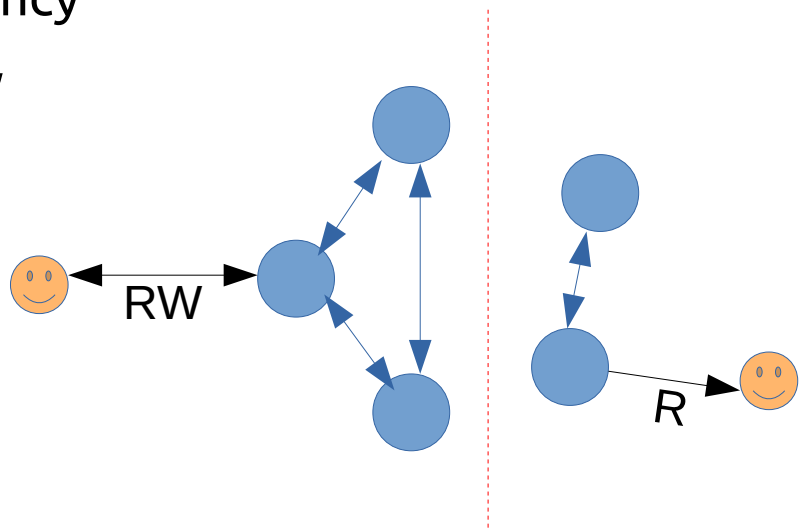
CAP Theorem



- The system is **available**
  - You can write => potentially inconsistent
- The system is **consistent**
  - You cannot write => not available

CAP Theorem - Critisism

- Partition tolerance is not a choice
  - Network are always unreliable even local networks
- Consistency means only linearized consistency
- Does not describe when network gets slow
- Does not describe nodes failures
- Many intermediate solutions
  - Quorum
    - One side is fully available
  - Partially available for read
    - During a certain delay
- Classifying systems as CP or AP is too simplistic



Eventual consistency

- Accept to send approximative answers
  - Eventually all access will return the latest value
    - Convergence
- Conflict resolution
  - Exchange of possible values between nodes
  - Reconciliation (choose the final value between available values)
    - No universal approach
    - Can be done asynchronously or before read/write if inconsistency is found

Time based reconciliation

- Typically used for "last-write-wins"
  - How to distinguish two writers writing "at the same time" ?
    - If not synchronized, later write might be overwritten by an earlier one
  - Rely on synchronized clocks
    - NTP - Network Time Protocol
      - Precision limit ~ 5-10ms
  - Don't capture causality of changes

Vector clocks (1)

- Vector associated to a data, that contains one tuple for each nodes having modified that data
  - D([S_1, v_1], [S_2, v_2], ..[S_n , v_n])
    - S: server/node
    - v: incremental version of data for a server
- At each change on node *n*, if the pair [S_n, v_n] is existing, V_n will be increased, if not [S_n, 1] will be created

Vector clocks (2)

- Writes handled by any of the replicas
  - Version vectors allow tracking (but not enforcing) causality between versions of an object
  - Version vector encodes amount of writes seen by each replica prior to current write
- Tolerates concurrent conflicting writes
  - Breaks strict consistency
  - No agreed-upon order between replicas
  - But works under partitions!
- Reads can return multiple conflicting versions

Vector clocks - Example

- Example
  - Client A writes D1 on Sx
  - Client B reads D1 and writes D2 on Sx
  - Clients C and D read D2 and independently write D3 on Sy and D4 on Sz
  - A client ask all the versions and detect a conflict
    - Once the conflict is solved, D5 is written on Sx

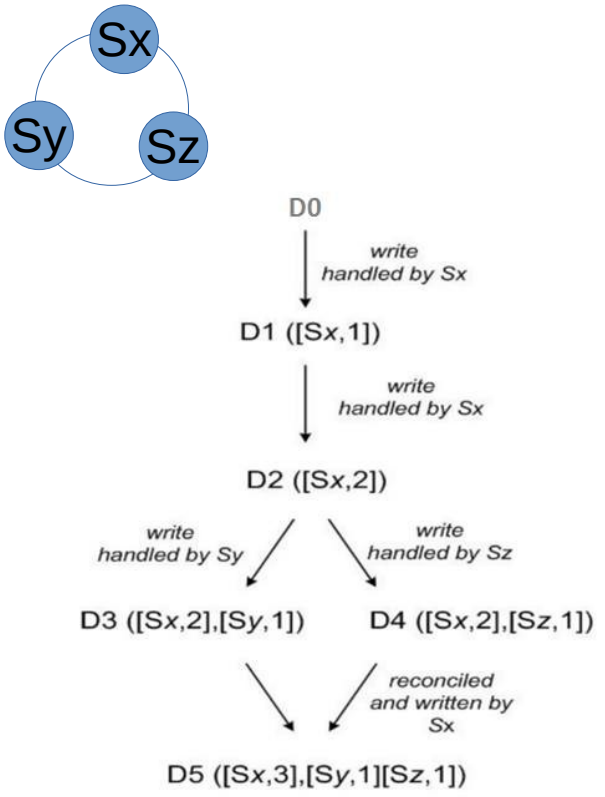


Image from original Dynamo paper

Vector clocks - Conflicts

- Is a value of D an ancestor of D' ?
  - $\forall [S, v] \in D, \exists [S, v'] \in D'$ where $v \leq v'$
    - D and D' are on the same branch, there is no conflict
  - D and D' are on parallel branches, they must be reconciliated
    - Sementically: by the client
    - Last-write-wins: use timestamps for v

D	D'	Conflict ?
([Sx, 3])	([Sx, 4], [Sy, 2])	no
([Sx, 2], [Sz, 1])	([Sy, 1], [Sz, 1])	yes
([Sx, 5], [Sy, 6])	([Sx, 8], [Sy, 6], [Sz, 2])	no

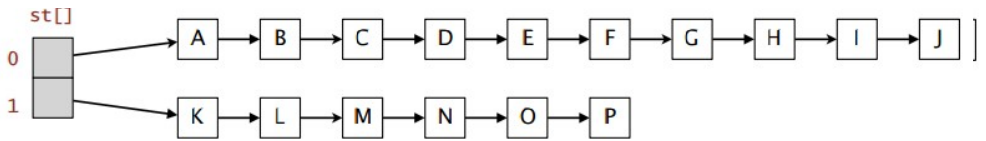
Cassandra

- Tunable consistency
  - Write
    - ALL: to all replicas
    - QUORUM: to a majority of replicas
    - ONE: at least one replica
  - Read
    - ALL: get the values of all replicas
    - QUORUM: get from a majority of replicas
    - ONE: from one replica (eventual consistency)
  - Many more levels (for example to control replication among data centers)
    - Sloppy quorum: "best effort" quorum in case not all nodes are available

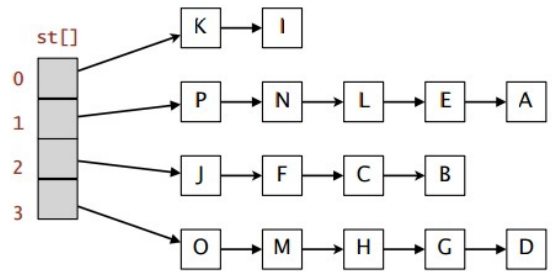
- Quorum
  - R: Number of reads
  - W: Number of writes
  - N: number of replicas
  - If R + W > N
    - Reads/writes overlap => "strong" consistency

Consistent hashing (1)

- Using hash table to distribute keys between servers
- server = hash(key) % count(servers)
  - 2 servers



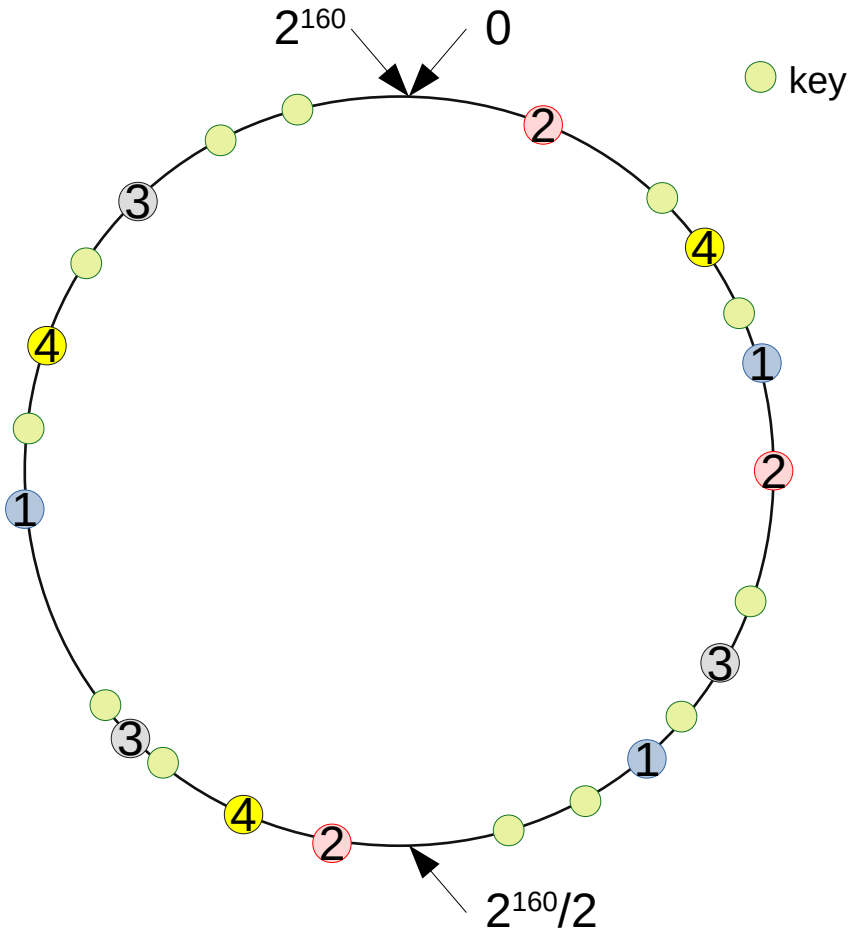
  - 4 servers



- When resizing servers, many keys have to be moved !
  - We have seen that problem with DB sharding

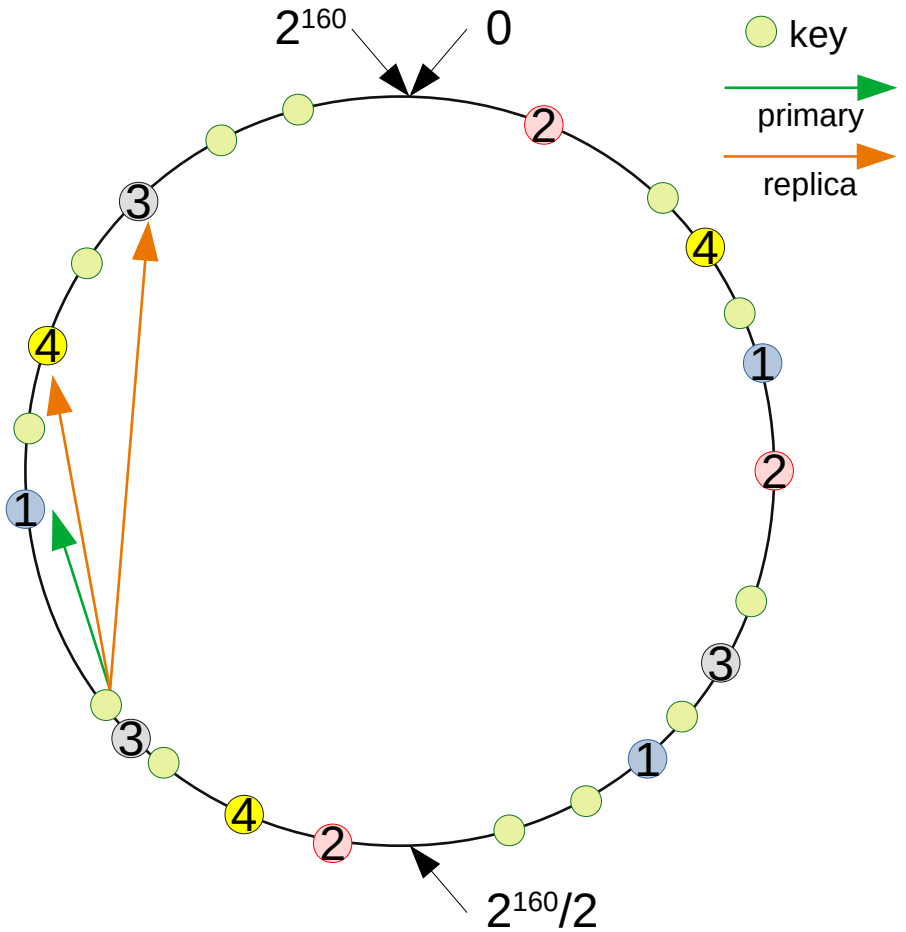
Consistent hashing (2)

- Keys have a consistent hash value
  - Typically SHA1(key) → 160 bits
- Each server pick a random position in this value space
  - For better global repartition, each server pick X random positions → virtual nodes
- Each server is responsible of the keys before its position (clockwise)
  - Now adding/removing a (virtual) node is only moving K/n keys
    - K: number of keys
    - n: number of slots
- Also called Distributed Hash Table (DHT)



Consistent hashing - Cassandra

- In Cassandra, the node after the key value is the **primary** node
  - But the data is also replicated X times (configurable) on the following nodes (clockwise)
- When using read/write operations like ONE or QUORUM (that don't need all the replicas), the nodes are not chosen at random. It is a priority list with the primary node first and then the successors
  - So, even using ONE you will have a good chance of being consistent



Advantages of eventual consistency

- Performance
  - Reads and writes can be served very fast by addressing a single replica
  - Amazon declares that any ms of additional latency translates into lost revenue in practice

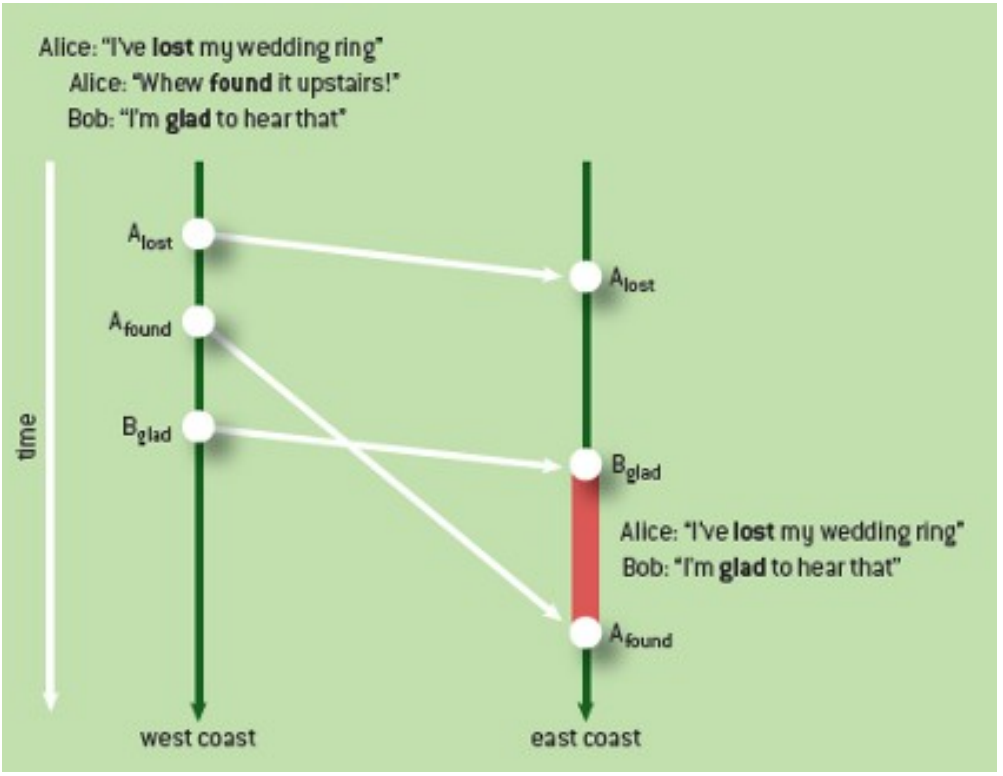
Advantages of eventual consistency

- Supports continuity of service under partition
  - **Original usage**: Amazon shopping cart
    - If an item gets added to the cart on one replica; and removed from the cart on another replica, but not from the former;
    - Business choice is an inclusive merge procedure: keep potentially deleted items but do not remove items
    - Apparently less costly for Amazon to offer items shipped by mistake to customer!
  - Conflicting versions happen rarely in practice in Amazon's data centers

Disadvantages of eventual consistency

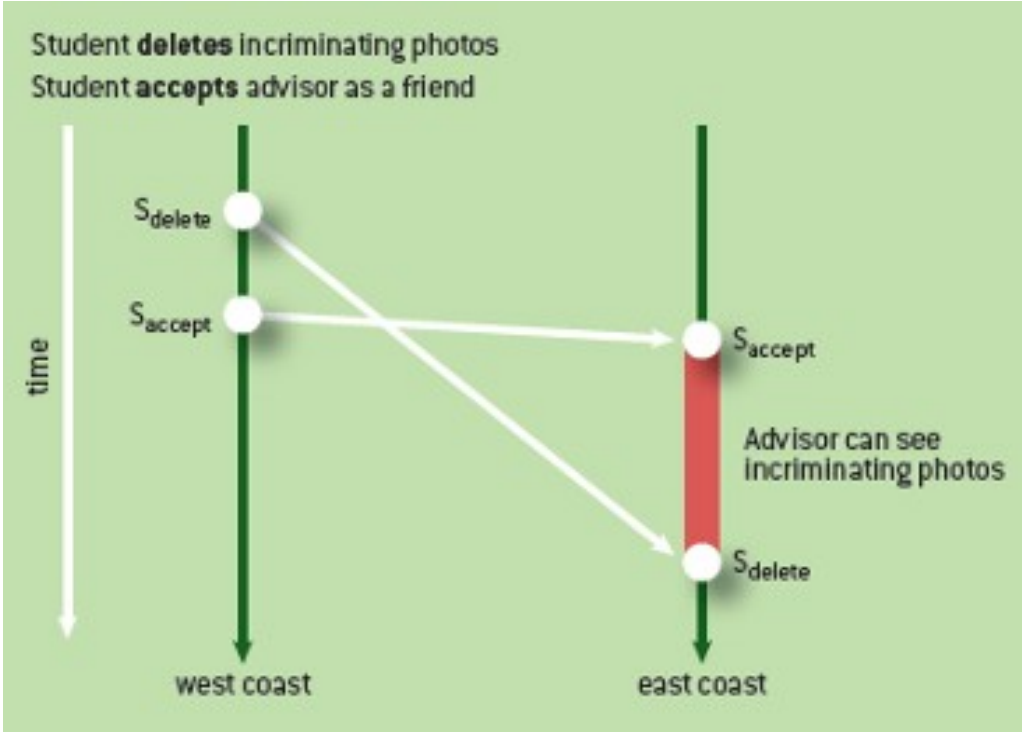
- More complex for the programmer
  - Last-writer-wins resolution might loose important updates
  - Complex to predict and handle all possible conflicts
- No guarantees on ordering of writes seen in each data center can lead to unwanted behaviors at the application level
- Might need to expose conflicts to the user
  - Dropbox: choose the latest file revision

Anomaly I: Comment reordering



https://queue.acm.org/detail.cfm?id=2610533

Anomaly II:
Photo privacy violation



https://queue.acm.org/detail.cfm?id=2610533

Consensus

The Agreement Problem

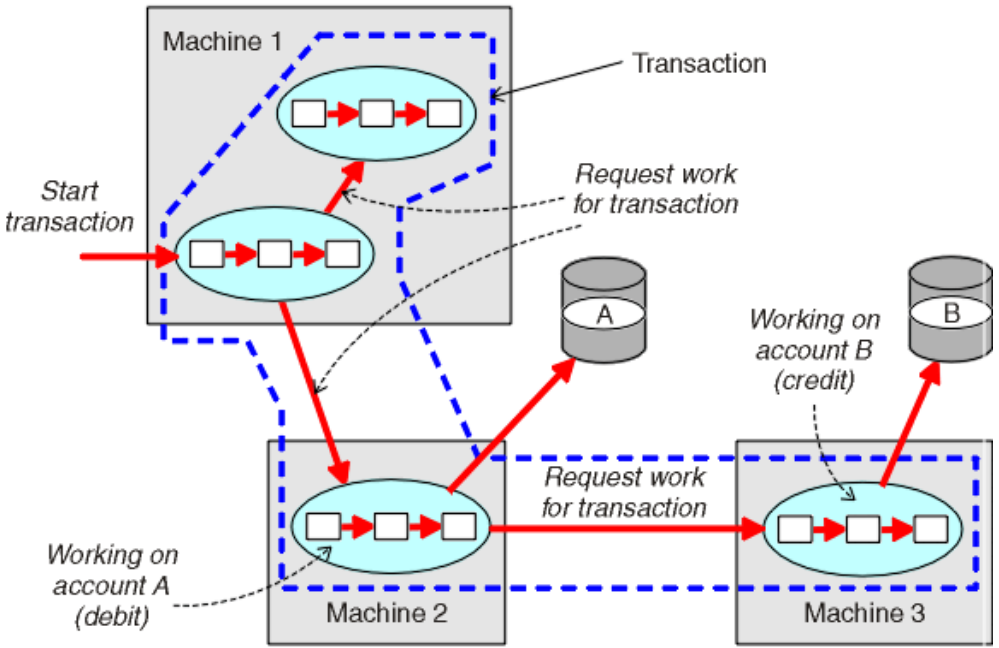
- In a distributed system
  - Some modes propose values (or actions) and send them to the others
  - All nodes have to agree on the next one
    - Same ordering
  - But in presence of
    - Failure/recovery of communication channels and machines
    - Concurrent processes (uncertainity of timings and order of events)

Consensus properties

- Agreement
  - All processes that decide do so on the same value
- Validity
  - If a process decides a value *v*, then *v* must have been proposed by some correct process
- Integrity
  - A node chooses at most once
- Termination
  - Every non-faulty process decides on a value

Two Phases Commit (2PC)

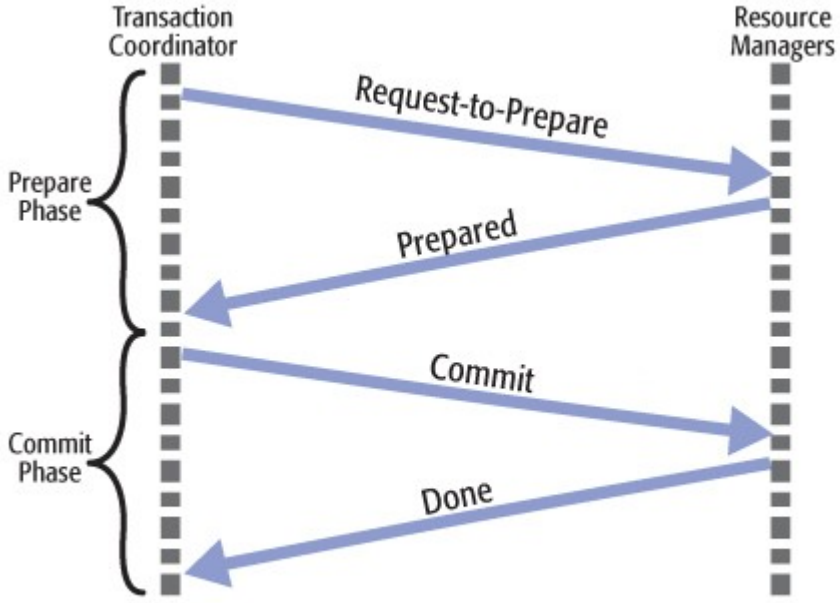
- Used for distributed ACID database transactions
  - Runs in multiple processes on several machines
  - All nodes must do their relative work
- Two features
  - Recoverable processes
    - Each process can restore an earlier state in case of failure
  - Commit protocol
    - Allows multiple processes to coordinate the commiting or aborting of a transaction
- For each transaction a process acts as a coordinator (Transaction Manager)
  - It oversees the activity of all participants (Ressource Manager) to ensure a consistent outcome



https://www.ibm.com/support/knowledgecenter/en/SSAL2T_9.1.0/com.ibm.cics.tx.doc/concepts/c_two_phz_commit_process.html

Two Phases Commit - Phases

- Prepare phase
  - Coordinator ask processes to prepare to commit
    - A process must prepare a permanent record of his work (durability)
    - If a process agree to commit, it must ensure that it will be able to do it
- Commit phase
  - The coordinator records the responses
  - If all responses are positive, it will send a commit message, if not it will send an abort message to each process



- This should guarantee that a successful transaction will be permanent
- But, if the coordinator fails after phase 1, all nodes are stuck !

https://docs.particular.net/nservicebus/azure/understanding-transactionality-in-azure

Consensus

- Achieve system reliability in presence of a number of faulty processes
  - Recover from failures automatically
    - Minority of processes fail: system makes progress
    - Majority of processes fail: lose availability but retain consistency
- FLP result (paper in references)
  - Consensus cannot be guaranteed in bounded time in a purely asynchronous network
    - Cannot distinguish between network and node failure
  - Assumptions of concensus protocols:
    - Most network operations will be bounded in time
    - Majority of participants will be available

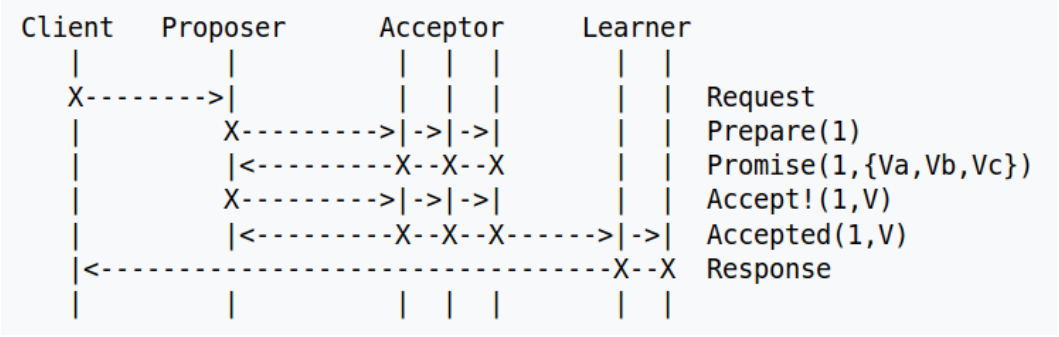
State Machine Replication (SMR)

- Programs are modeled as a state machine (more general model than read-write registers)
  - States: data
  - Transition: deterministic execution of the program input
- Can model tables or documents in a document store, with high-level operations, examples:
  - Increase reader count in metadata
  - Append a record to document
- Correctness is ensured by having all the replicas starting at the same state and applying the same sequence of transitions
  - A consensus algorithm is needed for that

Paxos protocol

Lamport - 1989

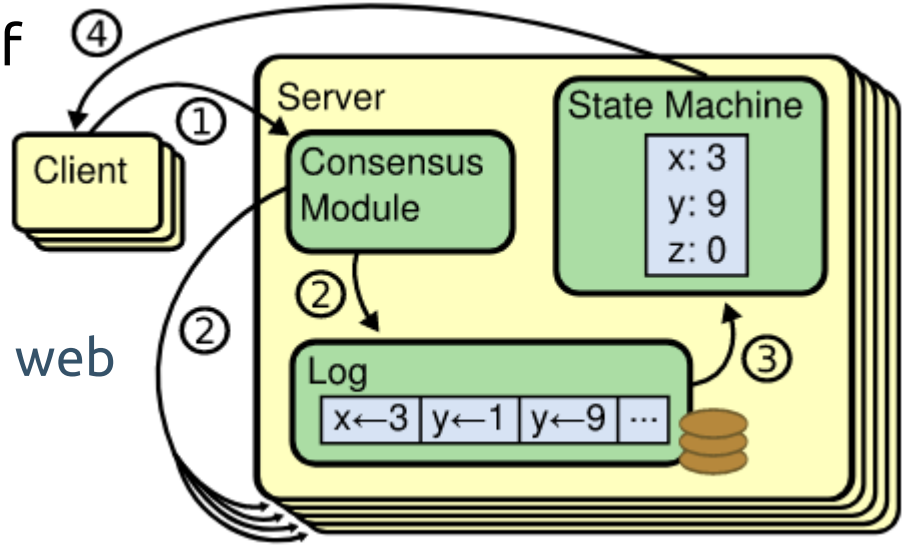
- Name comes from Island of Paxos, a fictive history that describe the protocol
  - The Part-Time Parliament (see references)
- Historical algorithm that has been proven to be correct
  - Dozen of papers discussing on optimizations, applications and usages
- The algorithm is complex to understand (not intuitive) and even more difficult to implement correctly



http://lamport.azurewebsites.net/pubs/pubs.html#lamport-paxos

Raft

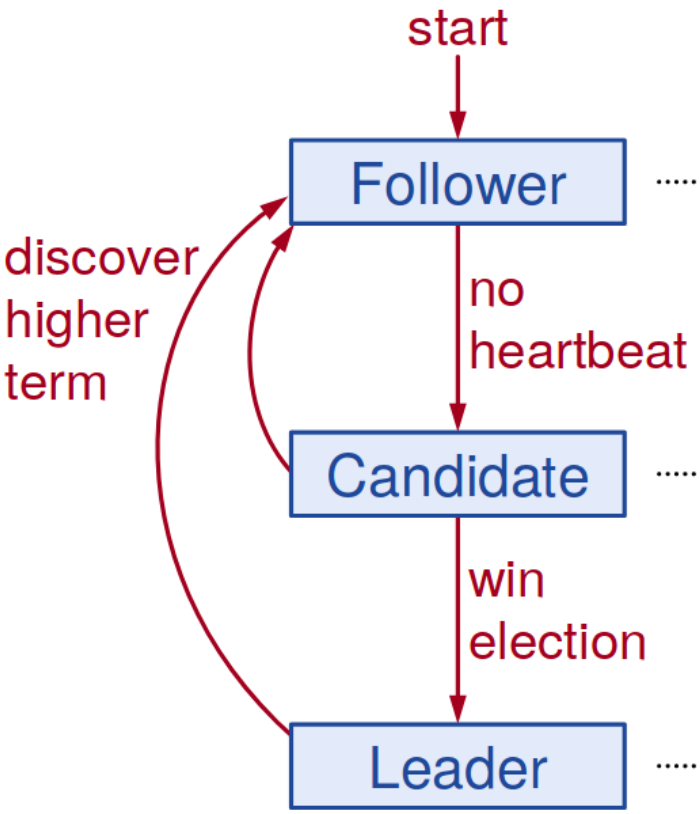
- Motivation
  - Having a concensus algorithm usable to build real systems
  - The algorithm must be understandable
- High popularity and growing number of implementations
  - Docker Swarm
  - Etcd (used by kubernetes)
  - 99 implementations referenced on the web site
- https://raft.github.io/



Raft - Overview

- Leader election
  - Select which node will act as a leader during the term
    - Require majority of votes
  - Randomized timeouts to avoid split votes
  - Hearbeat and timeouts to detect crashes and choose new leader
- Log replication (normal operation)
  - The leader accept commands from clients and append them to its log
  - The leader replicates its log to other nodes
- Safety
  - Only a node with up-to-date logs can become a leader

Raft - States



Passive (but expects regular heartbeats)

Issues **RequestVote** RPCs to get elected as leader

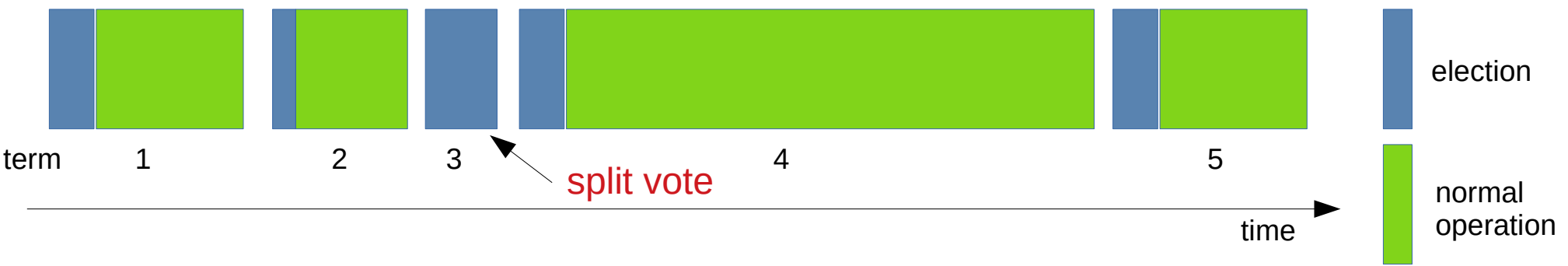
Issues **AppendEntries** RPCs:

- Replicate its log
- Heartbeats to maintain leadership

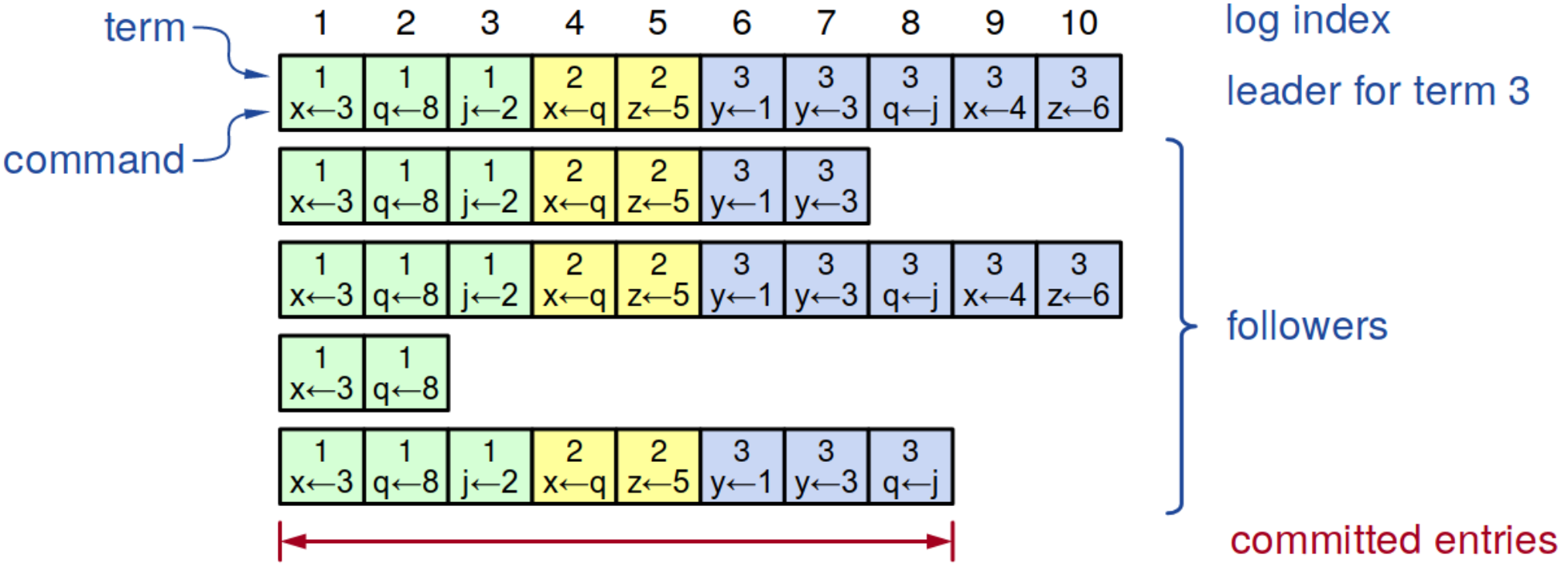
From slides of Diego Ongaro (author of Raft)

Raft - Operation timeline

- Operation divided in terms
  - Each term start with a leader election
    - The leader will be the leader for all the term
  - Once a leader is elected, sytem enter normal operation mode

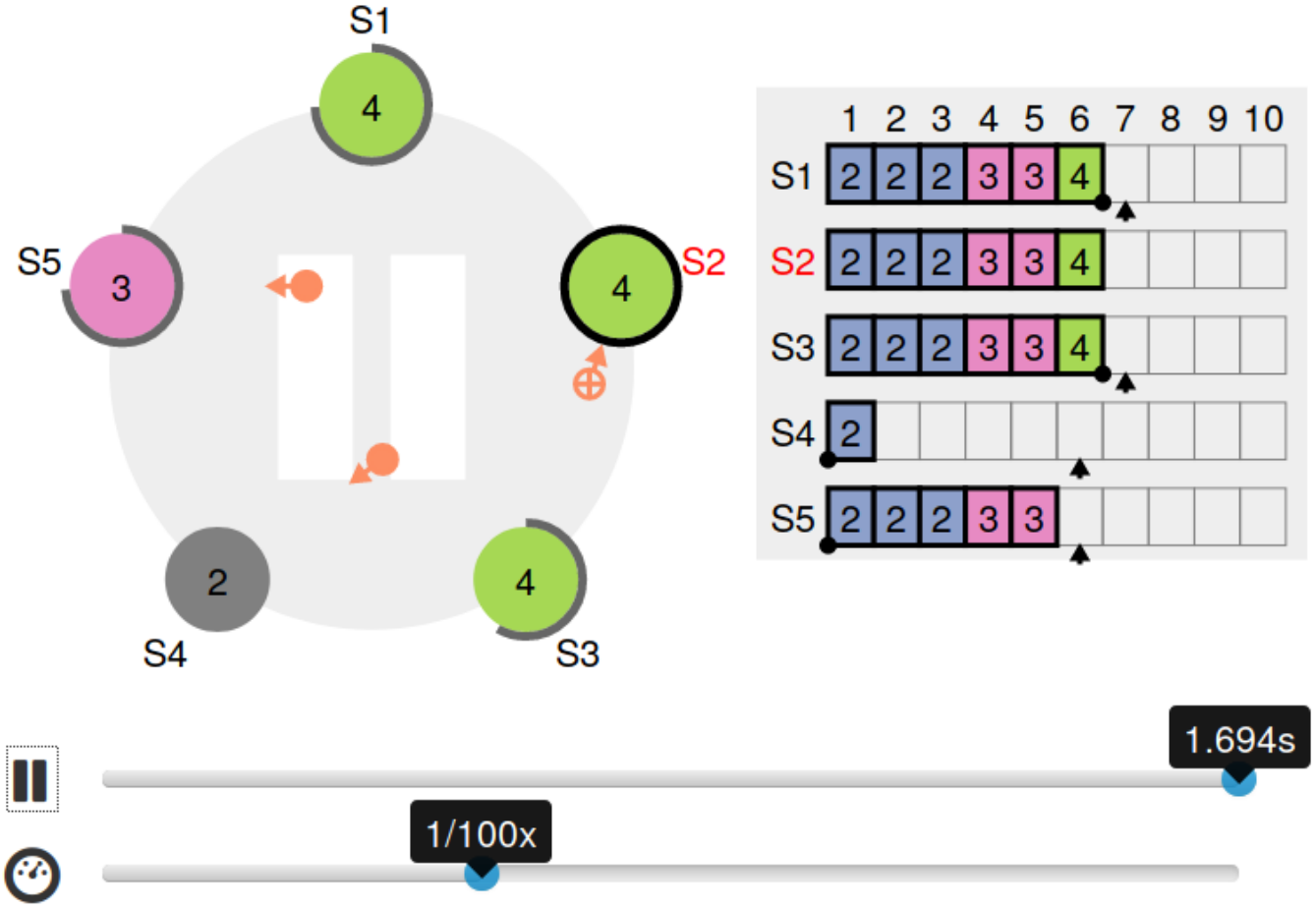


Raft - Log Structure



From slides of Diego Ongaro (author of Raft)

RaftScope


S1
4
S5
3
4
S2
2
S4
4
S3
1 2 3 4 5 6 7 8 9 10
S1 2 2 2 3 3 4
S2 2 2 2 3 3 4
S3 2 2 2 3 3 4
S4 2
S5 2 2 2 3 3
1.694s
1/100x

Conclusion

- Cloud computing offer a large variety of options for storage
  - Within a single application, different types of data can use different data stores due to different requirements
- (geo)Replication is necessary to handle faults and to support local access to data
  - Must define a consistency model applying to concurrent accesses
    - Strong vs. eventual consistency flavors
    - CAP theorem: with partitions, can support (strong) consistency, or availability, but not both
  - Impact on performance and programmability
- Consensus is a fundamental problem in fault-tolerant distributed systems
  - Some algorithms guarantee progresses when most participants are available

References

- Impossibility of Distributed Consensus with One Faulty Process, Fischer, Lynch, Paterson - 1985
- The part-time parliament, Lamport - 1998
- Paxos Made Simple, Lamport - 2001
- Consensus on Transaction Commit, Gray, Lamport - 2005
- In Search of an Understandable Consensus Algorithm, Ongaro, Ousterhout - 2014