08.Replication and Consistency

08We replicate data for increasing throughput, fault tolerance, availability(data crashes or tons of connections and DDoS attack), to improve latency(a copy closer to you will be accessible faster). We can also support disconnected operations. Desisgning for latency is something you need to do as it quickly becomes the constraint of the system.
Achieve fault tolerance

- Deal with failures / incorrect behaviors through redundancy
- We have already discussed consensus protocols for replicated state machines
  - Illusion of a single machine
  Increase availability
- Data may be available only intermittently in a mobile setting
  - A local replica in the mobile node can provide support for disconnected operations
- Data might become unavailable due to excessive load
  - For example, release of a new operating system
  Improve performance:
- Sharing of workload to increase the throughput of served requests and reduce the latency for individual requests
  - Example: replicate a Web server to sustain a higher number of users and reduce queuing effects for individual users
- Replicate data close to the users to reduce the latency for individual requests
  - Examples: cache in processors, local copy on mobile devices, content-delivery networks, geo-replicated datastores …

Dealing with latency

Latency does not improve over time as others performance metrics do. As other metrics improve, latency may easily become the limiting factor for performance.

	1983	2024	Improvement
CPU speed	1x10Mhz	8x3.2 Ghz	> 2000x
Memory size	<= 2MB	32 GB	> 16000x
Disk capacity	<= 30 MB	4 TB	> 100000x
Network bandwidth	3 Mbps	> 10 Gbps	> 3000x
Latency (RTT)	2.54 ms	0.1 ms	< 30x

We use 5 replicas to have a trade off between guarantees and loads as that is not a great number of nodes.
You also need to take in consideration the physics of the transportation of the information, latency as a limit of improvement dictated by the physical system you are using to distribute information.

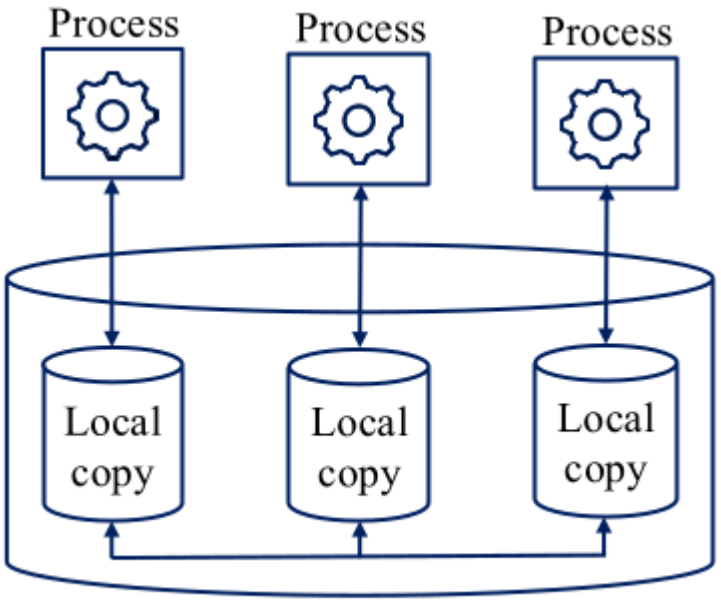
Replication: Challenges

You want the illusion of working with a single copy. Latency, communication and the protocol to communicate between nodes becomes more complicated the more consistency you want. The problem is the way to reach the level of consistency we want.
Main problem: consistency across replicas

- Changing a replica demands changes to all the others
- What happens if multiple replicas are updated concurrently?
  - Write-write conflicts / read-write conflicts
  - What is the behavior in the case of conflicts?
  Goal: provide consistency with limited communication overhead
  Scalability vs performance
- Replication may degrade performance!
- The cost to ensure that data is consistent across replicas can be very high
  - E.g., wait until a write has been successfully propagated to all replicas before making any progress
  Different consistency requirements depending on the application scenario
- Data-centric vs client-centric

Consistency models

We will see data store where they work on the data they store locally and update them consistently. This data have replication protocols that permit the data to remain consistent.



Distributed data store

A consistency model is a contract between the processes and the data store

- Stricter guarantees simplify the development but incur higher costs
- Weaker guarantees reduce the cost but make development difficult
- Tradeoffs: guarantees, performance, ease of use
  Several different models
- Guarantees on content: maximum “difference” on the versions stored at different replicas
- Guarantees on staleness: maximum time between a change and its propagation to all replicas(web browser mechanism)
- Guarantees on the order of updates: constrain the possible behaviours in the case of conflicts, data-centric vs client-centric(read the transactions from the local copy and do some assumptions on the order of operations)

Consistency protocols

Consistency protocols implement consistency models
We first overview the main implementation strategies used in consistency protocols then we discuss consistency models in detail and show how protocols can guarantee them

- Single leader

- Multi leader
- Leaderless

Single leader protocols

Writes goes through the leader. The read may go through the leader
One of the replicas is designated as the leader. When clients want to write to the data store, they must send the request to the leader, which first writes the new data to its local storage
The other replicas are known as followers. Whenever the leader writes new data to its local storage, it also sends the data to all its followers
When a client wants to read from the database inn some systems, it queries the leader(replicas only used as “backup”, like in Raft). In other systems, it can query any replica either the leader or a follower.
Type of protocol

- Synchronous: the write operation completes after the leader has received a reply
  from all the followers
- Asynchronous: the write operation completes when the new value is stored on the leader. Followers are updated asynchronously
- Semi-synchronous: the write operation completes when the leader has received a reply from at least k replicas. k is a configuration parameter in many replicated databases (e.g., Cassandra)
  Synchronous (or semi-synchronous with k followers) replication is safer
- Even if k-1 replicas fail, we still have a copy of the data
- Followers can recover from failures by asking updates to other replicas (catch-up recovery)
  What happens if the leader fails?
- We can elect a new leader (failover)
- Many tricky situations depending on the specific assumptions
  - E.g., followers may not be up-to-date, network may be partitioned
  To ensure safety, we need some consensus protocol (see previous lectures)
  Single leader protocols with synchronous or semi-synchronous replication widely adopted in distributed databases(PostgreSQL, MySQL, Oracle, SQL Server, MongoDB)
  Context: single organization / data center
- Low latency within the data center makes synchronous replication feasible

- Benefits in the case of frequent read accesses, which can be distributed across replicas
  Further optimizations used in practice(E.g., partition the data and assign a different leader to each partition,to better distribute the write load)
  No write-write conflicts possible:
- The leader receives all write operations and determines their order
  Read-write conflicts still possible
- Depending on the specific implementation
- E.g., a client reads from an asynchronous replica and does not see writes it previously performed on the leader

Multi leader protocols

When the geographical area is really large we use multiple leaders protocols. They guarantee less consistency. They also can be used when the transactions are distributed geographically
Writes are carried out at different replicas concurrently
No single leader means that there is no single entity that decides the order of writes
It is possible to have write-write conflicts in which two clients update the same value almost
concurrently

- How to solve conflicts depends on the specific consistency model
- We will discuss several of them later
  Multi leader protocols often adopted in geo-replicated settings
- Contacting a leader that is not physically co-located can introduce prohibitive costs
  In general, more difficult to handle …
- Simultaneous writes can create conflicts
  … but in practice, conflicts are rare and easy to solve in several application scenarios
- E.g., social network
  - Writes (posts, comments) are not frequent
  - Writes for one user / group of users often performed on the same replica
  - Conflicts are not critical: concurrent comments can be stored in different orders on different replicas
  Multi leader protocols natively supported in some database

- In addition to single leader protocols
  Single leader protocols within a data center. Multi leader protocols across data centers. Sometimes implemented in external tools(E.g., Tungsten replicator for MySQL, GoldenGate for Oracle)

Leaderless protocols

In single leader and in multi leader protocols, clients send writes to a single node. In leaderless replication, the client contacts multiple replicas to perform the writes/reads. In some implementations, a coordinator forwards operations to replicas on behalf of the client
Leaderless replication uses quorum-based protocols to avoid conflicts:

- Similar to a voting system
- We need a majority of replicas to agree on the write
- We need an agreement on the value to read
  Leaderless replication used in some modern key-value / columnar stores(E.g., Amazon Dynamo, Riak, Cassandra)
  We will distinguish models that can be implemented with highly available protocols and models that cannot. In this context, we say that a protocol is **highly available** if it does not require synchronous/blocking communication: if a node or a network link fails a client can still receive a reply from a correct (non-failed) replica. A model that is highly available cannot guarantee consistency as the node can continue to work with its local information. There are models that are intrisically highly available implementable and models that cannot be implemented with highly availability.

DATA-CENTRIC CONSISTENCY MODELS

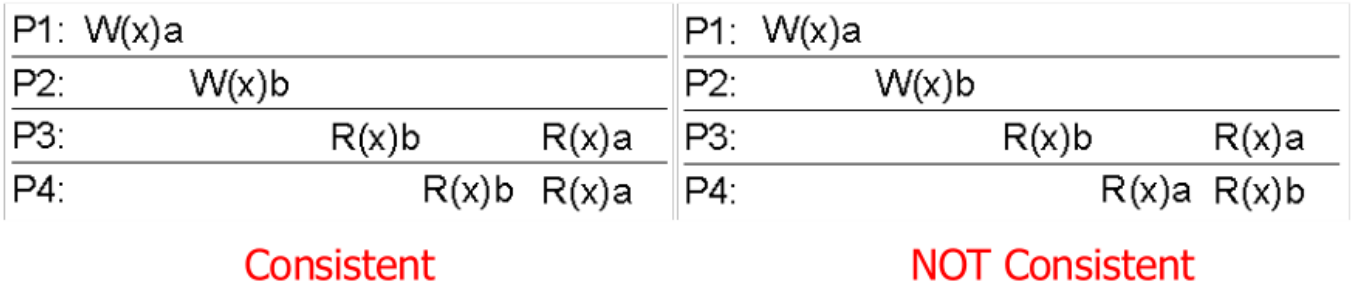
Predicate on the order in which the operations are perceived within the local copies.
Graphical convention

- One line for each process
- Operations of each process appear in temporal order
- W(x)a means that the value a is written on the data item x
- R(x)a means that the value a is read from the data item x
  Ideally, we would like all operations to:
- Take place instantaneously at some point in time

- Be globally ordered according to their time of occurrence
  This is not possible in a distributed system as there is no single clock available.
  The most we can get is similar to RAFT where we have the illusion that everything is happening in order, but we are not considering time.
  Even without considering time, we need to implement (expensive) coordination protocols to ensure global order.

Sequential consistency

The result is the same as if the operations by all processes were executed in some sequential order, and the operations by each process appear in this sequence in the order specified by its program. Operations within a process may not be re-ordered. All processes see the same interleaving. Does not rely on time



The sequence of operations must be consistent on each process to not violate the sequential consistency.
Originally developed as a cache/memory model for multi-processor computers.
Values can be read from the local cache and any old value is allowed unless the variable is declared as volatile, or there is a synchronized block. Synchronized blocks are sequentially consistent. In practice, synchronization is almost always controlled through synchronized blocks. Cases like the example in the previous slide should never occur.
Highly availability means no strong synchronization as it needs disconnected operations and not requires continuous communication between processes(still required to agree on order of operations).
Protocol assure that operations are executed in a sequential order, but we don't know which sequential order.

Consider the following program

Process P1	Process P2	Process P3
x = 1;	y = 1;	z = 1;
print (y, z);	print (x, z);	print (x, y);

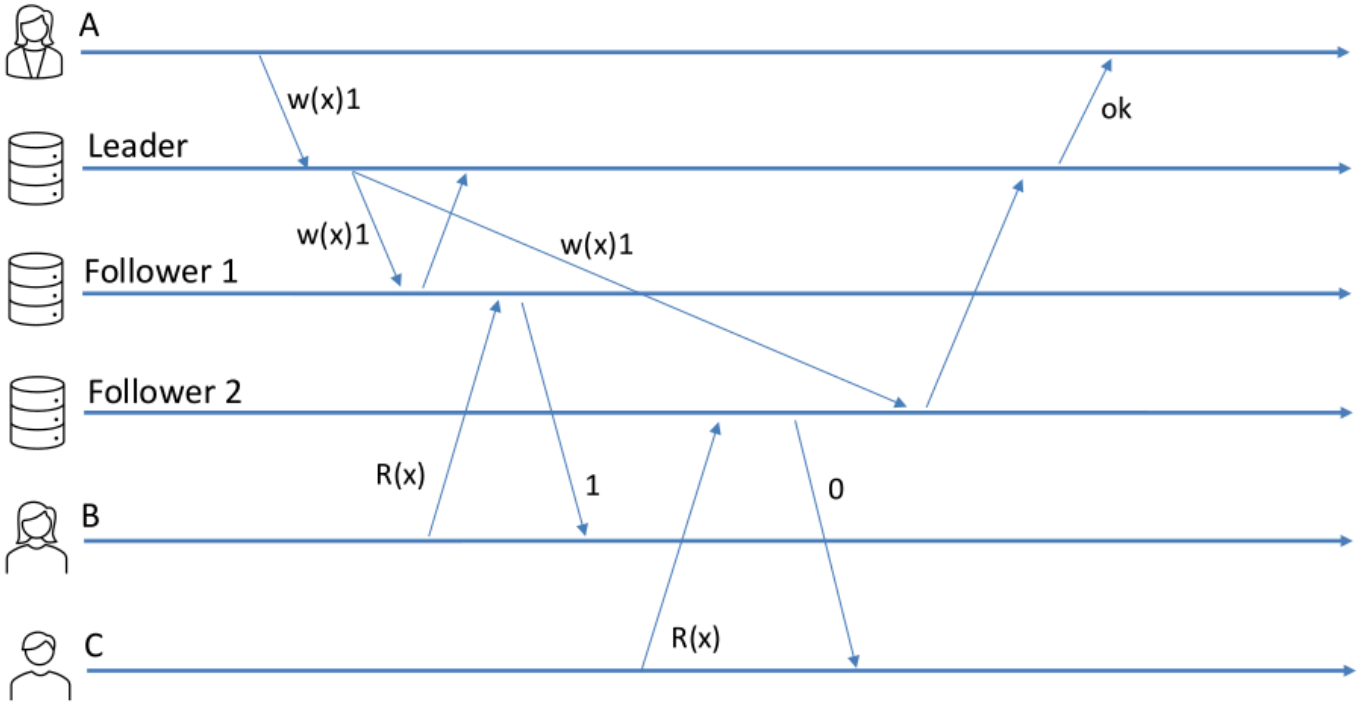
All the following (and many other) executions are sequentially consistent

x = 1;	x = 1;	y = 1;	y = 1;
print (y, z);	y = 1;	z = 1;	x = 1;
y = 1;	print (x, z);	print (x, y);	z = 1;
print (x, z);	print (y, z);	print (x, z);	print (x, z);
z = 1;	z = 1;	x = 1;	print (y, z);
print (x, y);	print (x, y);	print (y, z);	print (x, y);
Prints: 001011	Prints: 101011	Prints: 010111	Prints: 111111

All the replicas need to agree on a given order of operations.
Solutions:

- Single leader replication with synchronous replication
  - In practice, one of the reference implementations for sequential consistency
    - MySQL, PostgreSQL, MongoDB, …
- Failover (dealing with a leader failure) sometimes a manual procedure

Single Leader Implementation



Sequence: R(x)0 by C → W(x)1 by A → R(x)1 by B
(For simplicity, we are not considering possible failures of leader and followers)

35

The previous protocol works under the following assumptions

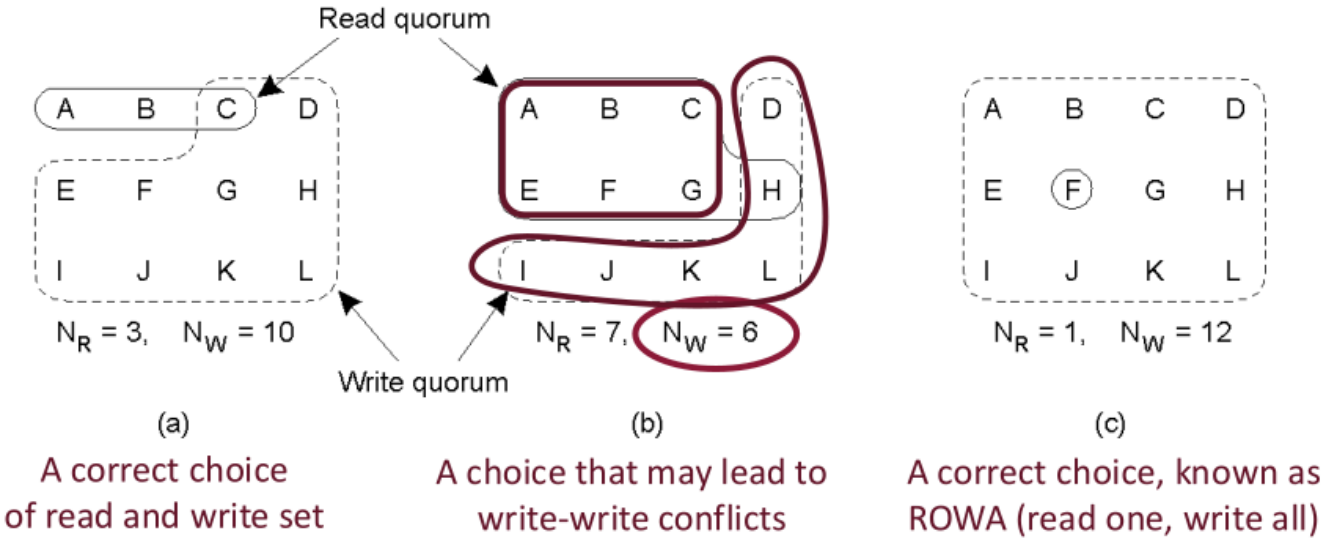
- Links are FIFO: update messages are received in the same order in which the leader sent them
- Clients are “sticky”: they always read from the same replica (either the leader or a follower)
  Simple protocol as if yo want to write something you have to communicate to the leader that will propagate the changes. The important part is that the leader decide the order of operations and propagate it. The reads can happens on multiple replica of the data, still consistent as everyone agree on order of operations.

Leaderless implementation

The idea is that a client propagate a request to multiple servers. Client contact a set of nodes.
Quorum-based:

- Clients contact multiple replicas to perform a read or a write operation
- An update occurs only if a quorum of the servers agrees on the version number to be assigned
- Reading requires a quorum to ensure the latest version is being read
  Typically:

- NR + NW > N Avoids read-write conflicts
- NW > N/2 Avoids write-write conflicts



The idea is that there will be at least a node that will receive both the operations and can answer. For writes I need at least a node in both partition of the set to know if there are concurrent writes and can serialize them. Depending on the workloads you can decide the number of nodes in the read and write partition.
How to update replicas in a leaderless implementation?

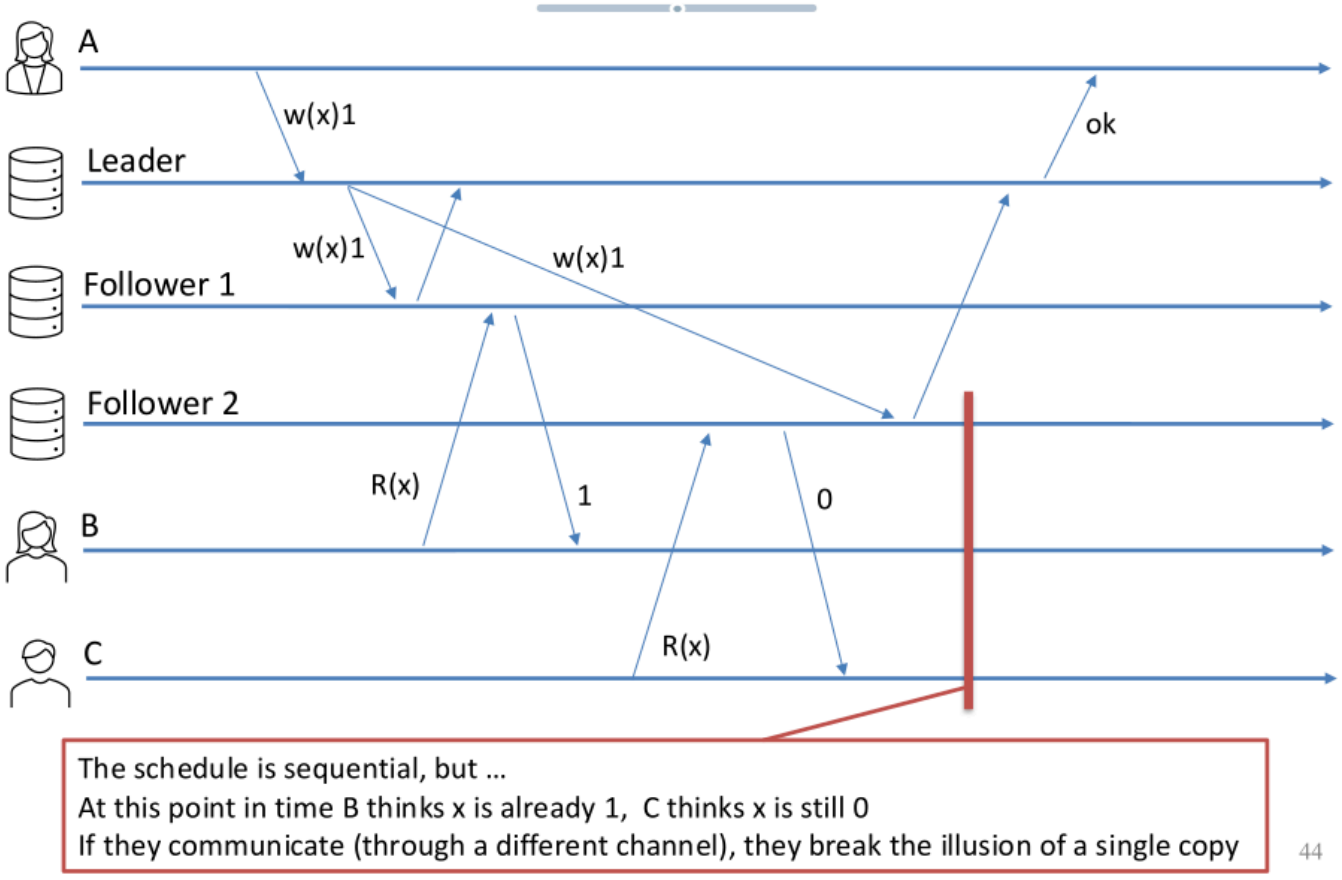
- Read-repair: when a client makes a read from several nodes in parallel, it can detect any stale responses. It sends the new value to the replicas that are not up-to-date
- Anti-entropy: in addition to read-repair, nodes periodically exchange data in background to remain up-to-date

Linearizability

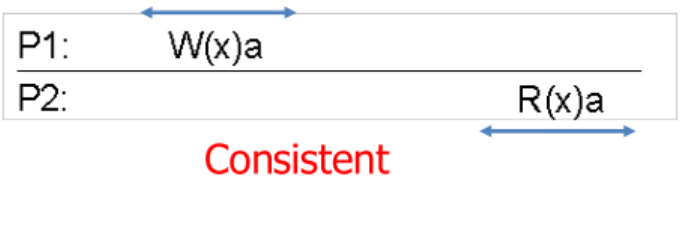
Each operation should appear to take effect instantaneously at some moment between its start and its completion.
Also known as strong / external / atomic consistency
Strongest possible consistency guarantee in presence of replication
Linearizability is a recency guarantee

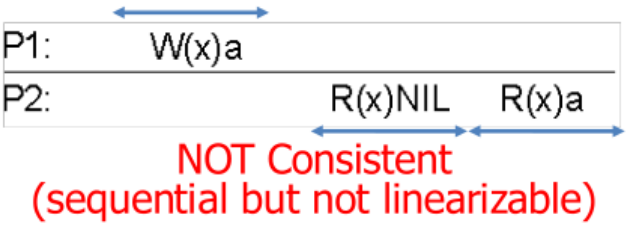
- When a client completes a write on the data-store, all clients need to see the effect of the write
- This gives the illusion of having a single copy of the data store
  It is also called external consistency as all the processes have the illusion of a

single clock. So if the processes can talk to each others on external channels the information need to be coherent



All writes become visible (as if they were executed) at some instant in time
Global order is maintained
Operations have a duration(E.g., from the time the clients submits an operation to the time it is durably stored in each replica)

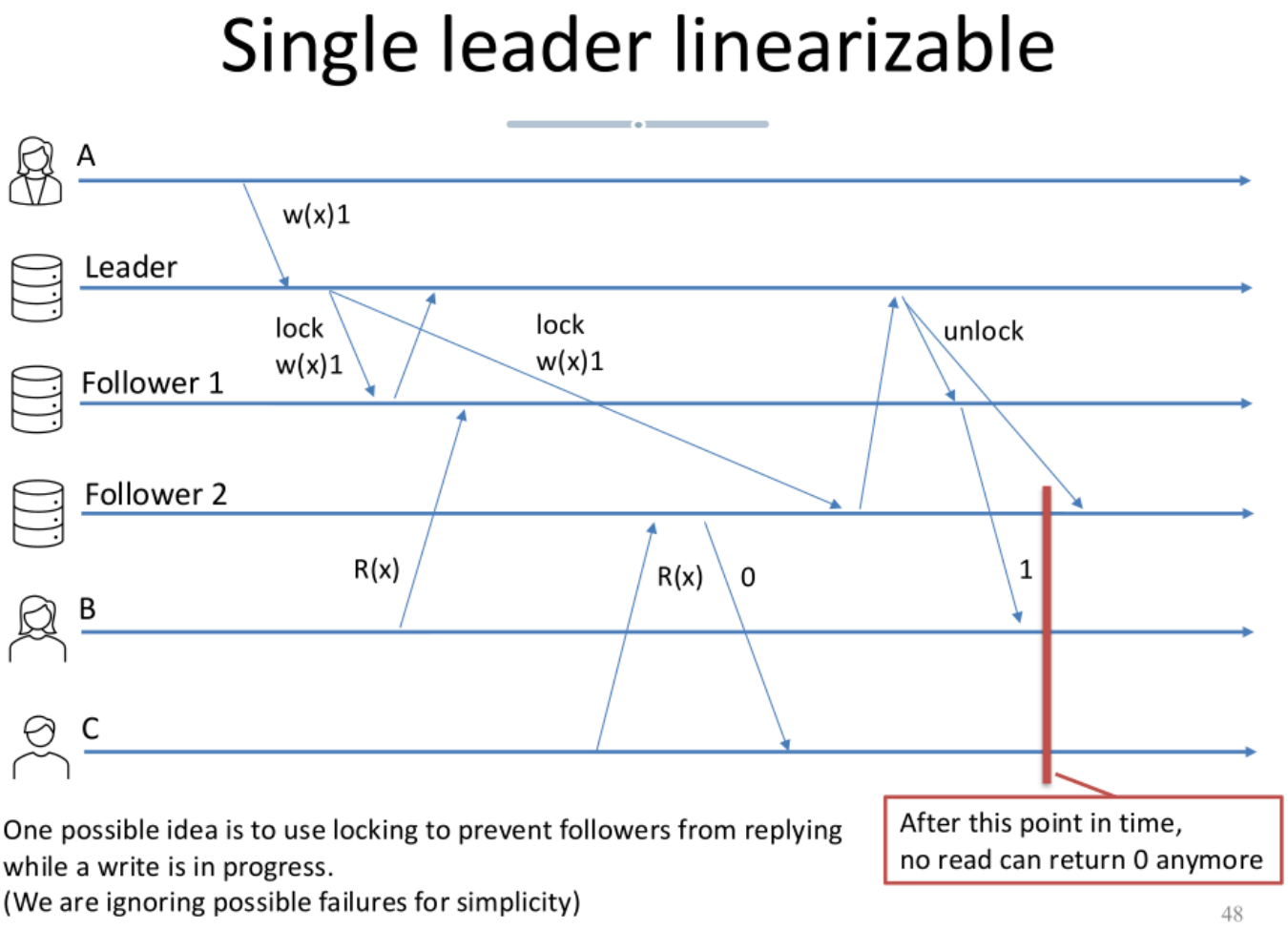




Linearizability is a composable property: if the operations on individual variables are linearizable the global schedule is also linearizable. When extended to multiple operations (transactions) linearizability is often called strict serializability
Single leader replication may implement linearizability

- The leader orders writes according to their timestamp
- Replicas are updated synchronously and atomically(E.g., locking protocol to avoid reading while a write is in progress)

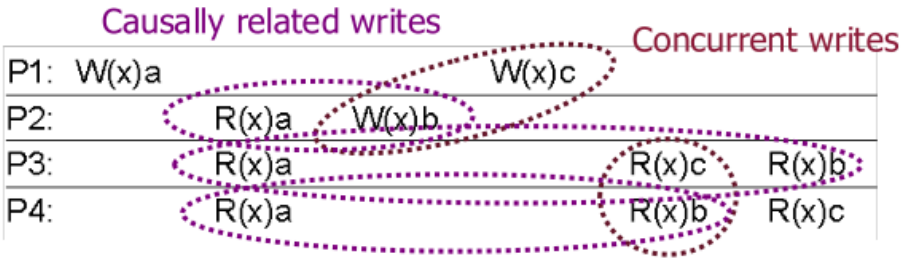
- Much more difficult to guarantee in the presence of failures
  It is an agreement/consensus problem to determine if and how the network is partitioned and who is the leader



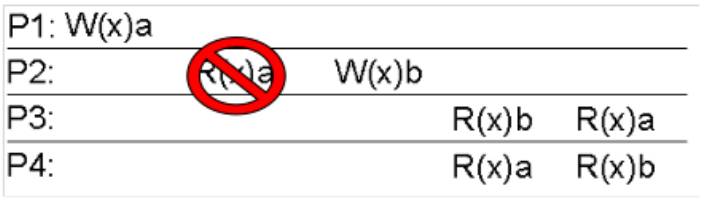
Causal consistency

Writes that are potentially(no way to tell if a message is replying to another one semantically) causally related must be seen by all processes in the same order.
Concurrent writes may be seen in any order at different machines
Giving up on total order bu we care on partial order: only messages that are related need to be in order(this is for a third process external from the exchange)
Weakens sequential consistency based on Lamport's notion of happened-before

- Lamport's model deals with message passing
- Here causality is between reads and writes



Consistent



NOT Consistent
(it becomes consistent without P2's R(x))

Causal consistency defines a causal order among operations, more precisely, causal order is defined as follows:

- A write operation W by a process P is causally ordered after every previous operation O by the same process even if W and O are performed on different variables
- A process P reads its own writes: a read operation by P on a variable x is causally ordered after a previous write by P on variable x
- Causal order is transitive
  It is not a total order: operations that are not causally ordered are said to be concurrent
  We use causal consistency as it is easier to guarantee within a distributed environment, smaller overhead, easier to implement

implementation

Multi leader implementations are possible (which enable concurrent updates)

- Writes are timestamped with vector clocks
- Vector clocks define what the process knew when it performed the write
  - The potential causes of the write
- An update U is applied to a replica only when all the write operations that are possible causes of U have been received and applied
  - Otherwise, a read always returns the previous value
  The above implementation is highly available:
- Clients can continue to interact with the store even if they are disconnected from other replicas
- The local replica will return an old value but it avoids violation of causality

- New writes can also be performed
  - The rest of the world will not be informed
  - The writes that occur in the rest of the world will be concurrent
  - This is clearly not possible under sequential consistency!

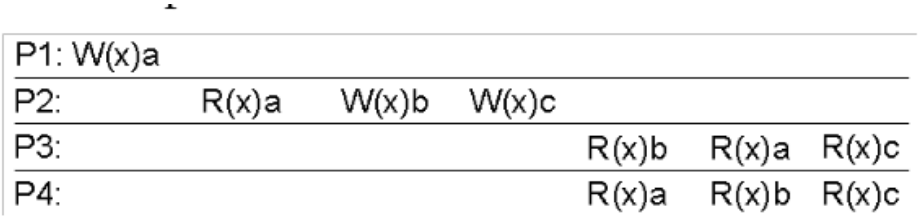
  Note: this implementation works only if clients cannot migrate between replicas (they are sticky)!
- If clients can migrate from one replica to another, no highly available implementations are possible

  A multi leader implementation is necessary as we need to work on multiple different replicas that can work separately.

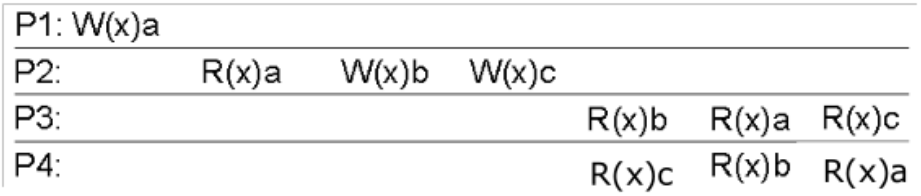
FIFO consistency

Writes done by a single process are seen by all others in the order in which they were issued; writes from different processes may be seen in any order at different machines

In other words, causality across processes is dropped • Also called PRAM consistency (Pipelined RAM). If writes are put onto a pipeline for completion, a process can fill the pipeline with writes, not waiting for early ones to complete. You only care on the order of operation on a single line. The only constraint is that you need to be consistent on the line.



Consistent



Not Consistent

This model is highly available. Even if you are disconnected when you reconnect and receive the messages you reorder them. Scalar clocks are sufficient as we don't care what the others are doing, care only about messages from processes in the same order.

Implementation

Very easy to implement: even with multi-leader solutions (concurrent updates)
The updates from a process P carry a sequence number: a replica performs an update U from P with sequence number S only after receiving all the updates from P with sequence number lower than S
FIFO consistency still requires all writes to be visible to all processes, even those that do not
care. Moreover, not all writes need be seen by all processes(E.g., those within a transaction/critical section).
Some consistency models introduce the notion of synchronization variables

- Writes become visible only when processes explicitly request so through the variable
- Appropriate constructs are provided (e.g., synchronize)
  It is up to the programmer to force consistency when it is really needed, typically
- At the end of a critical section, to distribute writes
- At the beginning of a “reading session” when writes need to become visible

Summary of data centric models

Consistency	Description
Linearizable	All processes must see all shared accesses in the same order. Operations behave as if they took place at some point in (wall-clock) time.
Sequential	All processes see all shared accesses in the same order. Accesses are not ordered in time.
Causal	All processes see causally-related shared accesses in the same order.
FIFO	All processes see writes from each other in the order they were used. Writes from different processes may not always be seen in that order.

Eventual consistency

In systems where we cannot have simultaneous updates or mostly reads eventual consistency is often sufficient as updates are guaranteed to eventually propagate to all replicas.
Very popular today for three reasons

1. Very easy to implement

2. Very few conflicts in practice(E.g., in social media applications, a user often accesses and updates the same replica.Today’s networks offer fast propagation of updates)
3. Dedicated data-types (conflict-free replicated data-types)
   Strong eventual consistency means that despite the fact the events are propagate in different order to different replicas they will be in the same order at some point in time.
   Conflict-free replicated data types (CRDTs) guarantee convergence even if updates are received in different orders(Commutative semantics). I may use timestamp to the operations to permit the reordering. It is possible that at some point in time the operations aren't in order but they will eventually be at some point in time later.
   The trick is that you propagate the difference and compute the correctness on the sum of that.
   Reasonable trade-off between performance and complexity: often used in geo-replicated data stores
   Can be difficult to reason about: in the case of concurrent updates, the value of a replica can temporarily store a “wrong” result. In practice, assumes that concurrent updates are rare

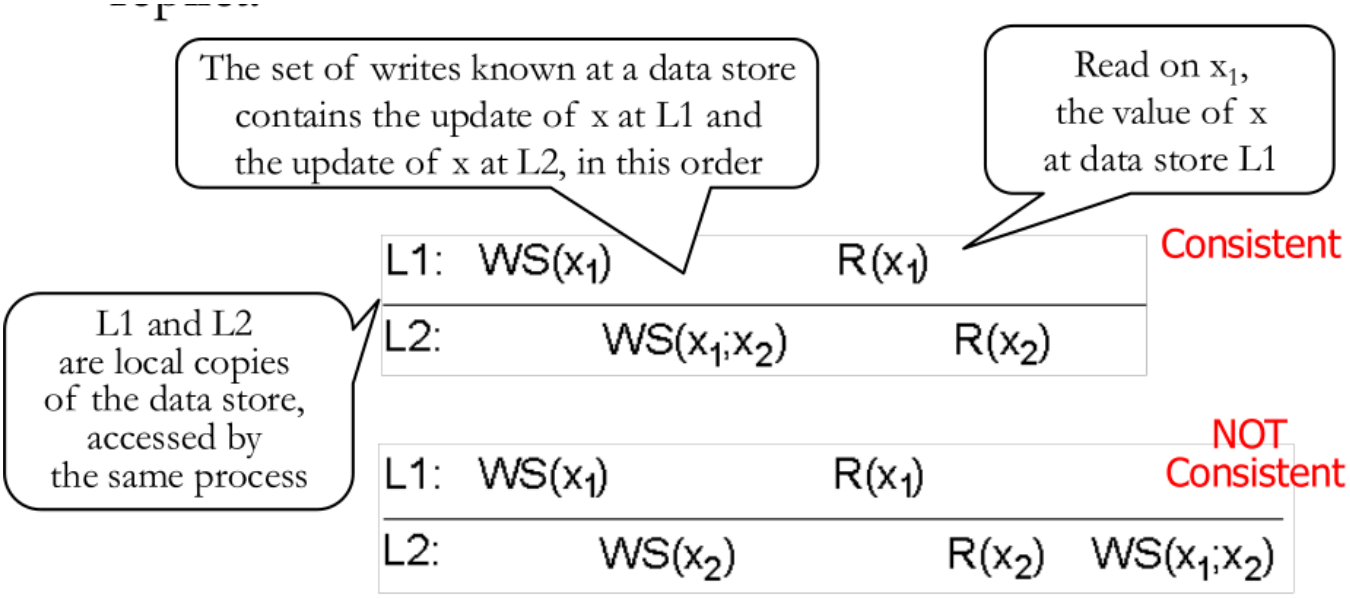
CLIENT-CENTRIC CONSISTENCY MODELS

What happens if a client dynamically changes the replica it connects to? Problem addressed by client-centric consistency models that provide guarantees about accesses to the data store from the perspective of a single client.
Four propetrties:

1. Order of reads in respect of other reads
2. Order of writes in respect of previous writes
3. Order of reads in respect of previous writes
4. Order of writes in respect of previous reads

Monotonic reads

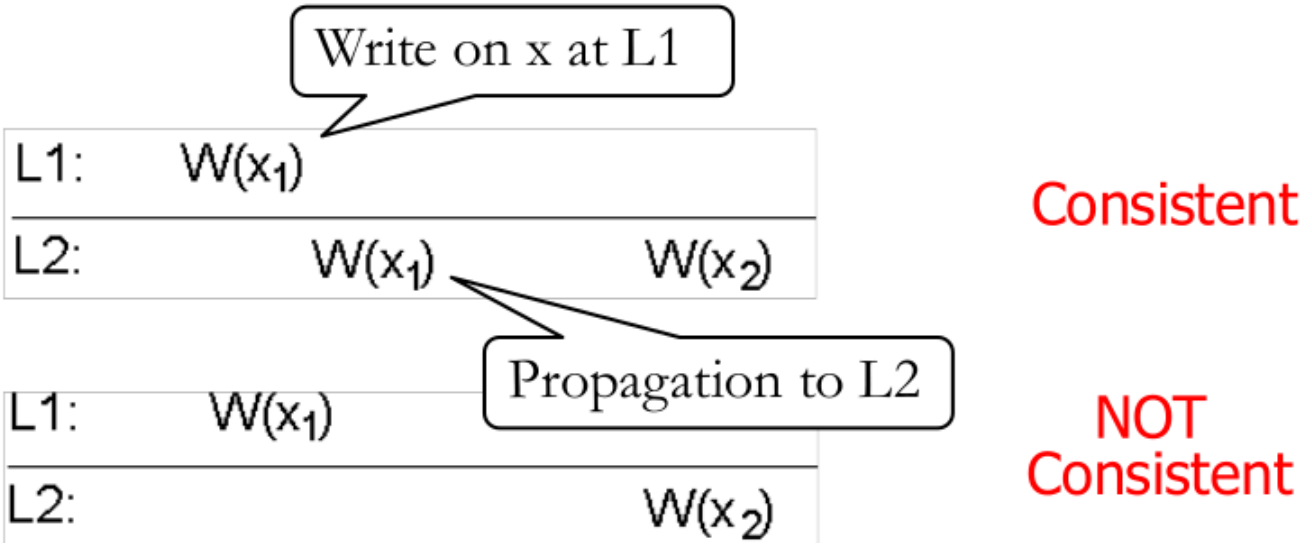
If a process reads the value of a data item x, any successive read operation on x by that process will always return that same value or a more recent value
Once a process reads a value from a replica, it will never see an older value from a read at a different replica.



The process is the same at two different locations and we have the clock of the client. X_1 is the value of X at location 1 at a given point in time.

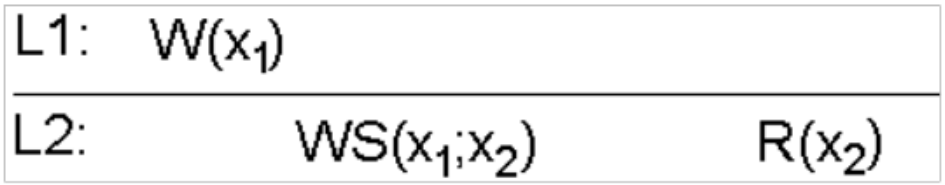
Monotonic writes

A write operation by a process on a data item x is completed before any successive write operation on x by the same process
Similar to FIFO consistency, although this time for a single process. A weaker notion where ordering does not matter is possible if writes are commutative. x can be a large part of the data store (e.g., a code library)

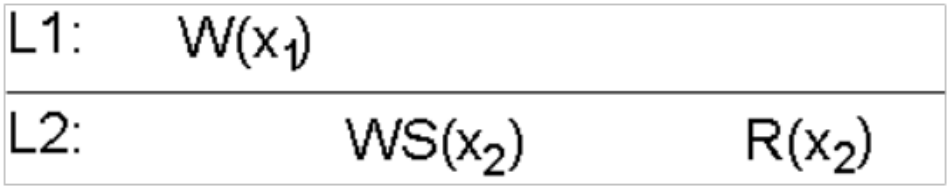


Read your writes

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



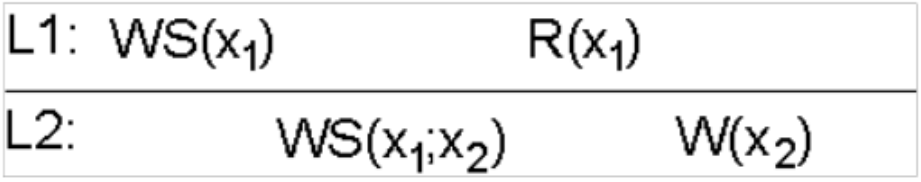
Consistent



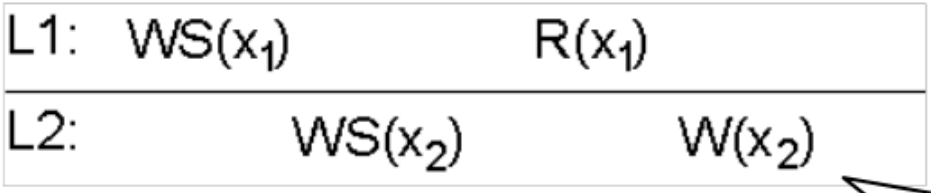
NOT Consistent

Writes follow reads

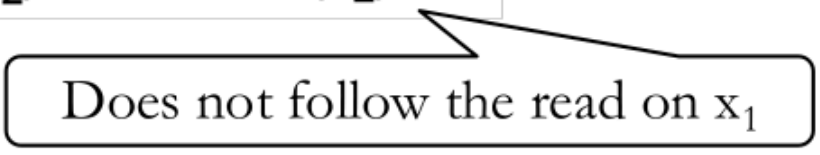
A write operation by a process on a data item x following a previous read operation on x by the same process is guaranteed to take place on the same or more recent value of x that was read



Consistent



NOT Consistent



Client-centric consistency:implementation

Each operation gets a unique identifier, for instance: replica id + sequence number
Two sets are defined for each client:

- Read-set: the write identifiers relevant for the read operations performed by the client
- Write-set: the identifiers of the write performed by the client
  Can be encoded as vector clocks: latest read/write identifier from each replica
  Clients reports to servers what they already seen in different replicas.

Client-centric consistency

Monotonic-reads: before reading on L2, the client checks that all the writes in the read-set have been performed on L2
Monotonic-writes: as monotonic-reads but with write-set in place of read-set
Read-your-writes: see monotonic-writes
Write-follow-reads: firstly, the state of the server is brought up-to-date using the read-set and then the write is added to the write-set

DESIGN STRATEGIES

Replica placement

- Permanent replicas: statically configured, take decision based on geographical regions
- Server-initiated replicas: you have facilities and you dynamically decide based on the load which replica you use
- Client initiated replicas: rely on a client cache, that can be shared among clients for enhanced performance

Update propagation

You can propagate the actual value, a notification(comunicate the update on the repèlica) or an operation to be performed. When you propagate the entire data written on a given replica when you have a lot of reads but few writes. If you have a lot of writes but few reads you propagate the notification(if someone read ask for the newer copy. called active replication). If propagating the new value is really expensive but the operation is small you can propagate the operation, if the operation is expensive but is done on few data you propagate the data.

Update Propagation

What to propagate?

- Perform the update and propagate only a notification
  - Used in conjunction with invalidation protocols avoids unnecessarily propagating subsequent writes
  - Small communication overhead
  - Works best if # reads << # writes
- Transfer the modified data to all copies

  - Works best is # reads >> # writes
- Propagate information to enable the update operation to occur at the other copies (active replication)
  - Very small communication overhead, but may require unnecessary processing power if the update operation is complex
  - Need to consider side effects
  How to propagate?
- Push-based approach
  - The update is propagated to all replicas, regardless of their needs
  - Typically used to preserve high degree of consistency
- Pull-based approach
  - An update is fetched on demand when needed
  - More convenient if # reads << # writes
  - Typically used to manage client caches
- Leases can be used to control the frequency of polling
  - They were developed to deal with replication …

Issue	Push-based	Pull-based
State of server	List of client replicas and caches	None
Messages sent	Update (and possibly fetch update later)	Poll and update
Response time at client	Immediate (or fetch-update time)	Fetch-update time

Comparison assuming one server and multiple clients, each with its own cache

Propagation strategies

Leader-based protocols

- Propagation can be synchronous, asynchronous, or semi-synchronous
  Leaderless protocols
- Read repair
  - When a client makes a read from several replicas in parallel, it can detect stale responses from some replica

    - The client (or a coordinator on its behalf) updates the stale replicas
- Anti-entropy process
    - Background process that constantly checks for differences among replicas and copies missing data
    - Interaction between replicas can be push or pull, and different strategies are possible

Case Studies

Distributed databases

Over the lectures, you have seen various topics related to managing shared state in distributed
systems:

- Concurrency control (isolation): locking protocols, timestamp based protocols
- Atomicity: atomic commit protocols
- Replication
  Let’s see how these aspects are considered in modern distributed databases
  Broadly speaking, modern systems can be classified based on the decision they take with respect to the CAP theorem
  As network partitions may always occur, systems may choose to offer either
- Strong guarantees (C) through blocking communication: higher latency, potentially not available in the presence of failures
- High availability (A) through non-blocking communication: lower latency, lower guarantees
  Various trade-offs have become popular over time

NoSQL (2000s)
- Key-value stores, wide-columns, document stores
  - Operations to access/modify individual items
- Isolation: no need, no multi-item transactions
- Atomicity: no need, no multi-item transactions
- Replication: asynchronous and/or multi-leader

Strong guarantees

High availability

NewSQL (2010s)
- Relational
- Isolation: serializable (locking and timestamp)
- Atomicity: (blocking) commit protocols
- Replication: sequential consistency / linearizable
- New techniques to limit costs
  - Case studies in the next slides

Case study: Spanner

Designed for very large databases: many partitions, each partition is replicated
Standard techniques:

- Single leader replication with Paxos for fault-tolerant agreement on followers and leader
- 2PC for atomic commit
- Timestamp protocols for concurrency control
  Made a synchronous system as they use expensive HW and need to know the time is real. If there Novelty: TrueTime
- Very precise clocks (atomic clocks + GPS)
- Offer an API that returns an uncertainty range
- The “real” time is certainly within the range
  Read-write transactions use TrueTime to decide when to commit
- The transaction coordinator asks a transaction timestamp to TrueTime
- It waits to release all locks and commit only when the uncertainty range is certainly passed the commit timestamp is certainly passed for every node
- Transactions are ordered based on time: linearizable!
  Read-only transactions also acquire a timestamp through TrueTime
- They need not lock, but simply read the latest value at that time
- Significant optimization when read-only operations are frequent
  Use real clock time as timestamp. This is good ONLY for read only transactions. You keep multiple copies of the data when you need to read you lock the data, get a timestamp and read the data only until the timestamp. Really fast and can

be distributed. Get the latest value available. Don't care that at time you have concurrent read or write as they are committed in the future.

Case study: Calvin

Simple transaction, high cost for 2PC and you need them as there could be failures. On the critical path you are keeping the lock on the database. SO we do the two thing on two different time. Complesive power decrease as transactions need to be deterministic. You preprocess the computation of transactions to permit determinism. Run a consensus process and then distribute the transactions and from that point in time you assume there is no failure as if a replica fails there will be another one that can vote and the failed replica can redo the transaction log when come back in time as the transactions are deterministic. So there is zero risk commitment.
Designed for the same settings as Spanner. Adopts a sequencing layer to order all incoming
requests (read and write), replicated for durability and essentially, a replicated log implemented using Paxos
Operations are required to be deterministic and they are executed everywhere in the same order
Guarantees: linearizability provided by the sequencing layer (essentially, a replicated log)

- The sequencing layer dictates the order of transactions
- Nodes read their local copy of the replicated log and process transactions in a way that guarantees equivalence to the order defined in the log
  Advantage:
- Agreement (order of execution) achieved before acquiring locks: lower lock contention
- No need for 2PC: the failure of a participant does not lead to an abort
  - As transactions are deterministic, the participant can return to the state before the failure by simply replaying all the operations in the log
  - 2PC is responsible for most of the execution time of simple transactions (frequently, the most common ones)

Case study: VoltDB

Construct smaller databases from a central one. Ask the developer how to build the partition. For single partition transactions can go directly to the partition and really execute sequentially

Developers specify how to partition database tables and transactions(E.g., hotel and flights tables both partitioned by city)

Single-partition transactions may execute sequentially on that partition without coordinating with other partitions. Standard protocols for other transactions.