



**POLITECNICO**  
MILANO 1863

# Replication and Consistency

Alessandro Margara

[alessandro.margara@polimi.it](mailto:alessandro.margara@polimi.it)

<https://margara.faculty.polimi.it>

# Replication: why?

---

- 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

# Replication: why?

---

- 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

# Don't forget the speed of light ...

	OR	VA	TO	IR	SY	SP	SI
CA	22.5	84.5	143.7	169.8	179.1	185.9	186.9
OR		82.9	135.1	170.6	200.6	207.8	234.4
VA			202.4	107.9	265.6	163.4	253.5
TO				278.3	144.2	301.4	90.6
IR					346.2	239.8	234.1
SY						333.6	243.1
SP							362.8

(c) Cross-region (CA: California, OR: Oregon, VA: Virginia, TO: Tokyo, IR: Ireland, SY: Sydney, SP: São Paulo, SI: Singapore)

Table 1: Mean RTT times on EC2 (min and max highlighted)

# Replication: examples

---

- Collaboration platforms (e.g., Google Doc, Microsoft 365, DropBox)
  - Documents / files may be replicated in users' devices
  - Support for disconnected operations
  - Documents / files are reconciled upon reconnection
  - Assumption: conflicts (simultaneous changes from multiple users) are infrequent
- Distributed file systems
  - May also support simultaneous changes
- Content delivery networks (CDN, e.g., used by Netflix)
  - Geographically distributed networks that replicate the content to better serve end users

# Replication: challenges

---

- 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

# Replication: challenges

---

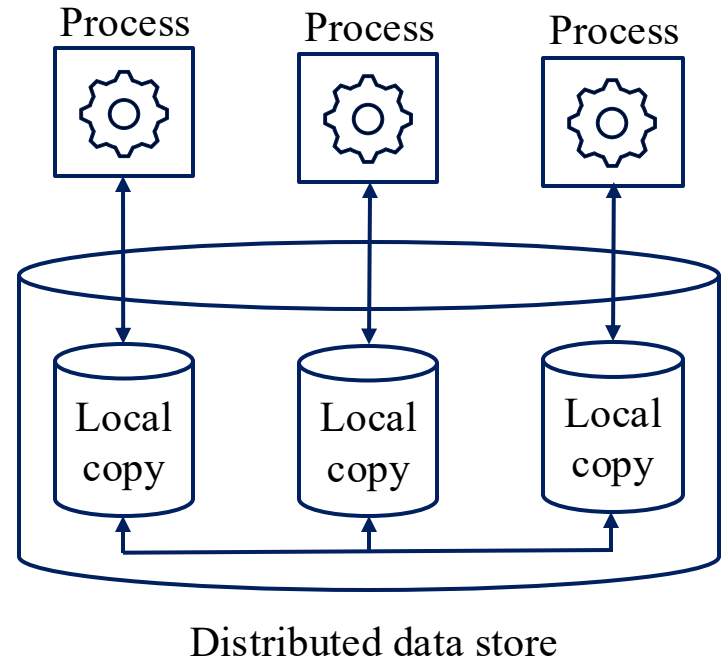
- Scalability vs performance
  - Replication may degrade performance!
  - The cost to ensure that data is consistent across replicas can be very high
    - E.g., wait until a write has been successfully propagated to all replicas before making any progress
- Different consistency requirements depending on the application scenario
  - Data-centric vs client-centric



# Consistency models

---

- Focus on a (distributed) data store
  - Shared memory, filesystem, database, ...
  - The store consists of multiple items
    - Files, variables, ...
- Ideally, a read should show the result of the last write
  - What does last mean?
  - Impossible to determine without a global clock



# Consistency models

---

- 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

# Consistency models

---

- 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
  - Guarantees on the order of updates
    - Constrain the possible behaviors in the case of conflicts
    - Data-centric vs client-centric

# Consistency protocols

---

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

# Single leader protocols

---

- One of the replicas is designated as the leader
  - When clients want to write to the datastore, 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
  - In 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
    - Here, we focus on these systems

# Single leader protocols

---

- 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)

# Single leader protocols

---

- 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

---

- 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



# Single leader protocols

---

- 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

---

- 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

---

- 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

---

- 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

# Outline

---

- In the following, we review the main consistency models
  - Data-centric
  - Client-centric
- We show how they can be implemented within the classes of protocols identified before
  - Single leader
  - Multi leader
  - Leaderless

# Outline

---

- 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

# Outline

---

- High availability is related to the CAP theorem
  - C = consistency
  - A = availability
  - P = network partition (failures)
- In the presence of network failures (P), you can have availability (A) or consistency (C), but not both
  - We investigate the topic further and we show that *some* (*weak*) consistency models can be achieved with high availability



# **DATA-CENTRIC CONSISTENCY MODELS**

# Data-centric consistency models

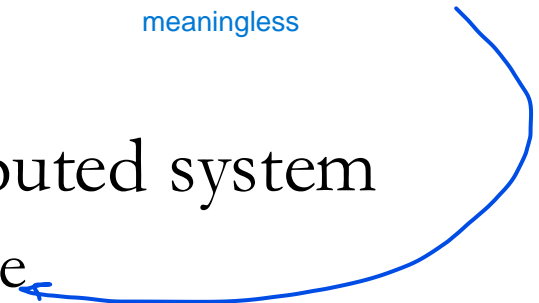
Ideally: we would like to have the illusion that data are in the same place

---

- We now review the most widely used data-centric consistency models
- 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  $x$

# Data-centric consistency models

---

- 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
    - There is no single clock available
  - 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”*

Every process must perceive the same order of operations.

# Sequential consistency

- Operations within a process may not be re-ordered
- All processes see the same interleaving
- Does not rely on time

Picture represent a picture from outside.

Each line represents the view of a process, which is reading or writing from his local storage

P1:	W(x)a				
P2:	<sup>4</sup> W(x)b				
P3:		<sup>1</sup> R(x)b		<sup>5</sup> R(x)a	
P4:			<sup>2</sup> R(x)b		
			<sup>3</sup>	R(x)a	

**Consistent**

I can find a sequence for the operations to be executed that is consistent

Since we don't have a world clock, we don't know the actual order

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

**NOT Consistent**

P3 and P4 received update from P1 and P2 in different order

We assume that at the beginning is a null value associated to every object, so before a read of an object there must be a write

# Sequential consistency

---

- Originally developed as a cache/memory model for multi-processor computers
- C++, Java memory models
  - Are they sequential?

L. Lamport. “How to make a multiprocessor computer that correctly executes multiprocess programs” ACM Transactions on Programming Languages and Systems. 1979.

# Sequential consistency (Java)

## Thread 1

```
int x = 0;
```

```
int y = 0;
```

```
...
```

```
...
```

```
...
```

```
...
```

```
x = 1;
```

```
y = 1;
```

## Thread 2

```
...
```

```
...
```

```
...
```

```
...
```

```
...
```

```
...
```

```
...
```

```
...
```

```
...
```

```
read y = 1;
```

```
read x: which values are allowed?
```

one can expect that x must be 1 before its assigning happened before the y assignment

**0 is allowed!**

Sequential consistency (between threads of a process) is too difficult to implement that also memory models of programming languages don't implement it

# Sequential consistency (Java)

---

- Values can be read from the local cache ...
- ... and any old value is allowed In this case different threads can see different orders of operations
- 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

Sequential consistency is difficult!



# Sequential consistency

- Consider the following program

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

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

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

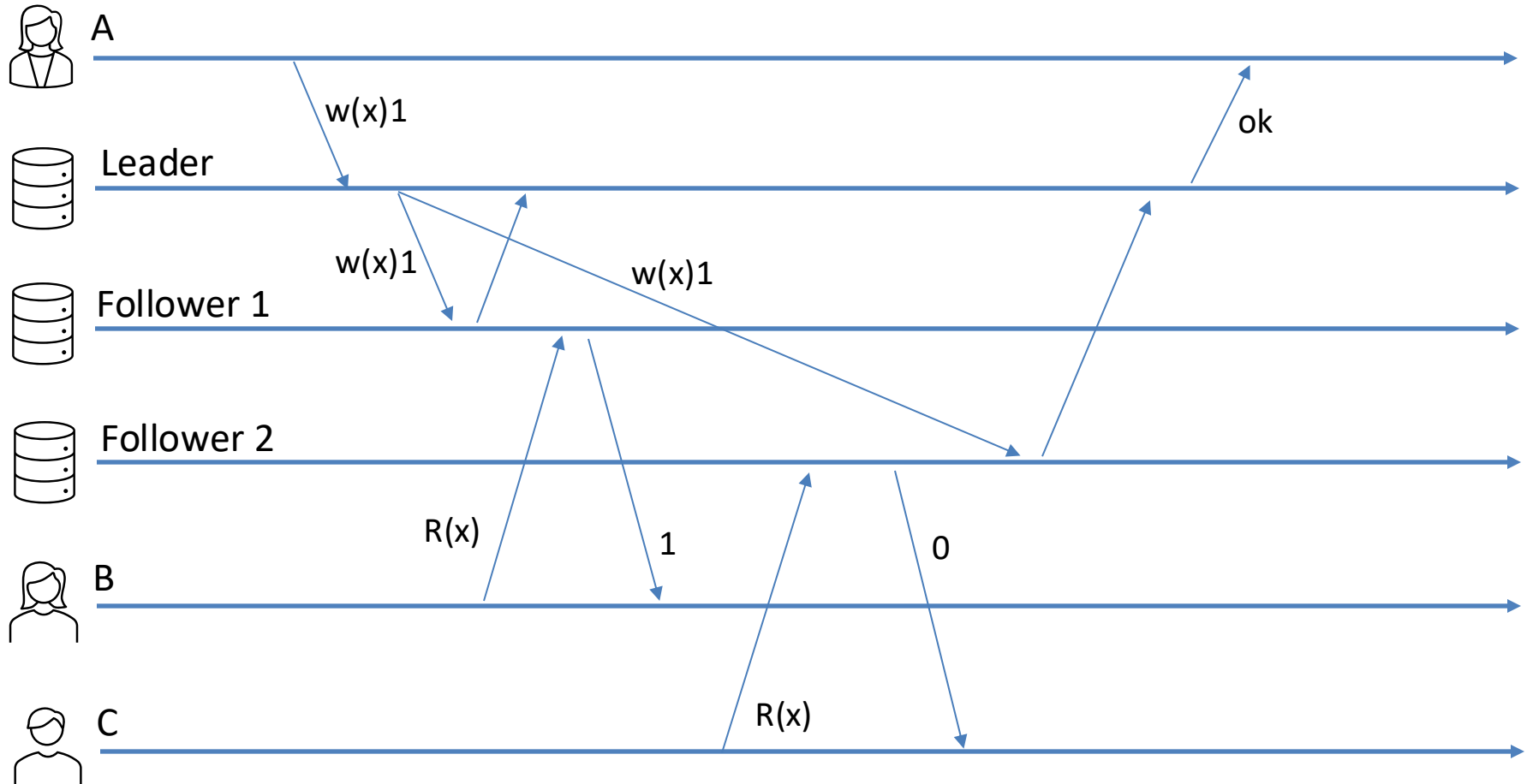
# Sequential consistency implementation

---

- 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

We centralize the decision of the sequence



That is the sequence everybody agrees on

Sequence:  $R(x)0$  by C  $\rightarrow W(x)1$  by A  $\rightarrow R(x)1$  by B

(For simplicity, we are not considering possible failures of leader and followers)

# Single leader implementation

---

- 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)

# Leaderless implementation

---

- 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

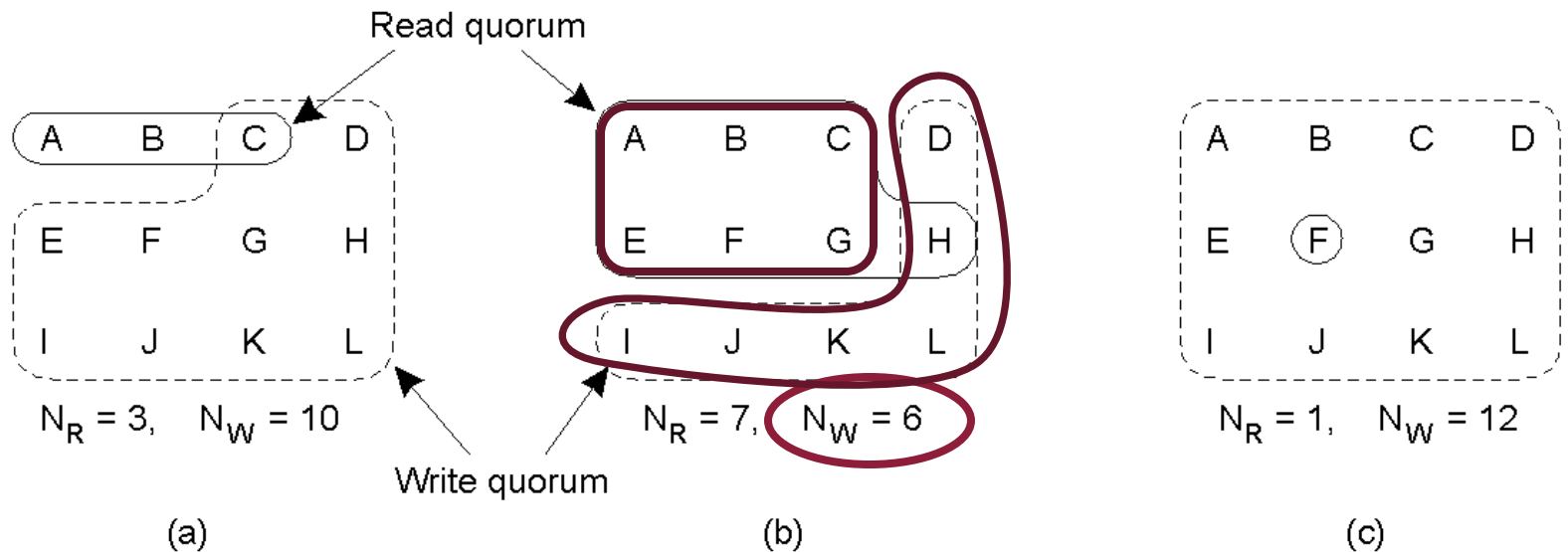
number of writing  
clients for a certain  
write

- $NW > N/2$

Avoids  
write-write conflicts

There is at least one node that learns  
about conflict of two contemporary clients wanting to write and decides about the one who can do it

# Leaderless implementation



A correct choice  
of read and write set

A choice that may lead to  
write-write conflicts

A correct choice, known as  
ROWA (read one, write all)

In this configuration, the chosen read quorum ( $N_R$ ) and write quorum ( $N_W$ ) overlap enough to ensure that conflicts are avoided.

# Leaderless implementation

---

- 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

# Sequential consistency implementation

---

- Sequential consistency limits availability
  - No highly available implementations are possible
- Single leader protocols
  - Client needs to contact the leader
  - Leader must propagate the update to the replicas
    - In a synchronous way if we want to guarantee sequential consistency!
- Leaderless
  - Client needs to contact a quorum of servers
- Two main problems related to availability
  - High latency due to synchronous interactions
  - Clients are blocked in the case of network partitions



# Linearizability

---

*“Each operation should appear to take effect instantaneously at some moment between its start and its completion”*

# Linearizability

---

- 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

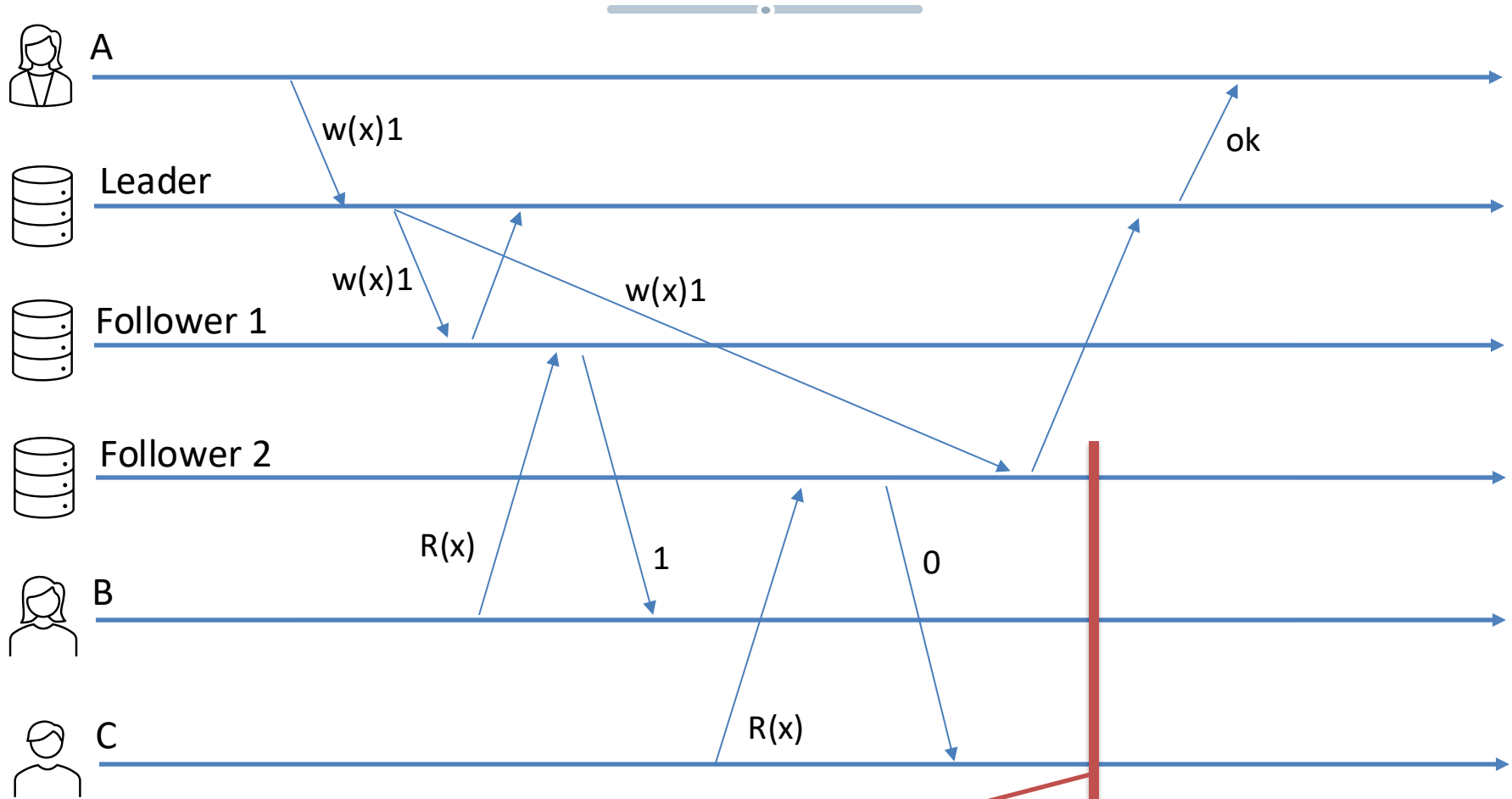
# Linearizability

---

- Why is sequential consistency not enough?
  - Sequential consistency does not include a notion of time
  - Linearizability includes a notion of time: operations behave as if they took place at a single point of (wall clock) time
  - It may require time to propagate an operation to all replicas, so the operation may not be instantaneously visible
  - Let's see our previous example again

All the processes need to have the illusion that there's a single clock.

# Linearizability



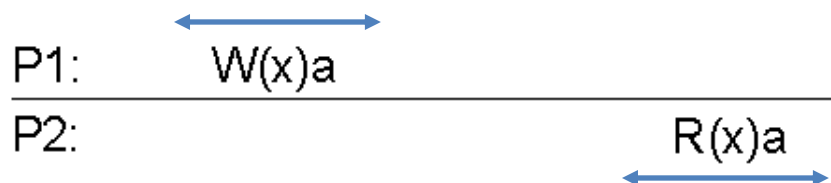
The schedule is sequential, but ...

At this point in time B thinks x is already 1, C thinks x is still 0

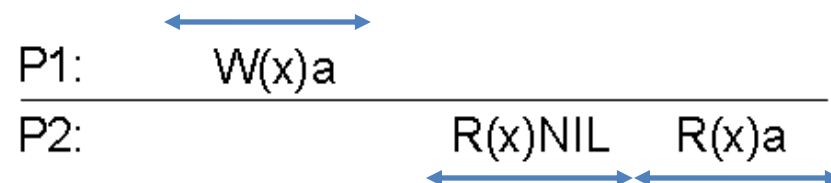
If they communicate (through a different channel), they break the illusion of a single copy

# Linearizability

- 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



Consistent



NOT Consistent  
(sequential but not linearizable)

# Linearizability

---

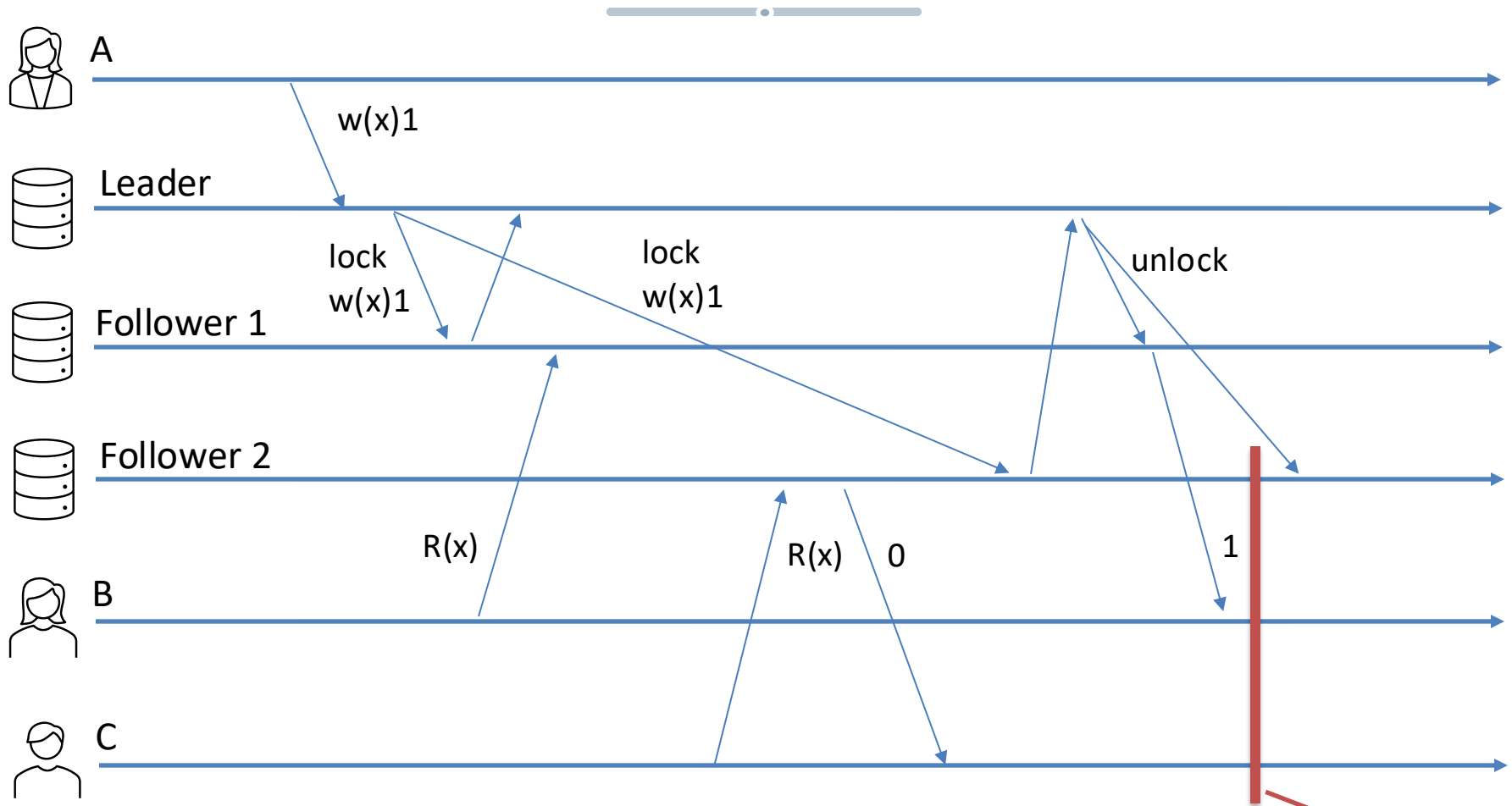
- Linearizability is a *composable* property
  - If the operations on individual variables are linearizable ...
  - ... the global schedule is also linearizable
- When extended to multiple operations (transactions) linearizability is often called strict serializability

# Linearizability

---

- 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

# Single leader linearizable



One possible idea is to use locking to prevent followers from replying while a write is in progress.

(We are ignoring possible failures for simplicity)

After this point in time,  
no read can return 0 anymore



# Causal consistency

---

- Consider a group chat discussion
  - A says “Distributed systems are the best!”
  - B says “No way! They are too complex ...”
- If C first sees B and then A, she cannot understand what is going on
  - Therefore, everybody else must see A’s message before B’s one

# Causal consistency

---

- Consider other two messages
  - A says, “Apple released a new MacBook”
  - B says, “I’m having a lot of fun learning about replication and consistency!”
- The two messages are not related to each other
  - The order in which they are seen from other members does not matter

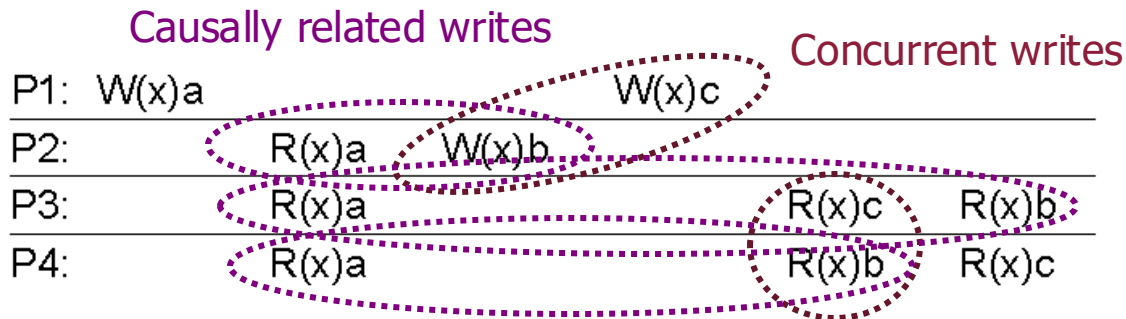
# Causal consistency

---

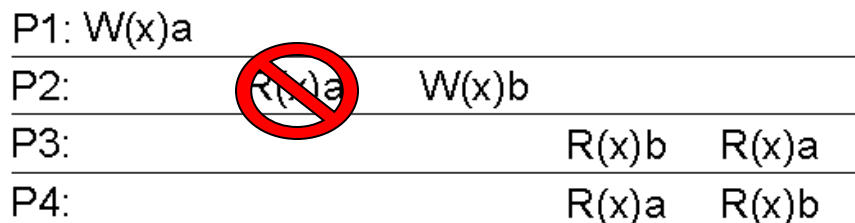
*“Writes that are potentially causally related must be seen by all processes in the same order.  
Concurrent writes may be seen in any order at different machines”*

# Causal consistency

- 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

---

- 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

# Causal consistency

---

- Why causal consistency?
- Easier to guarantee within a distributed environment
  - Smaller overhead
- Easier to implement

# Causal consistency: 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

# Causal consistency: implementation

---

- 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



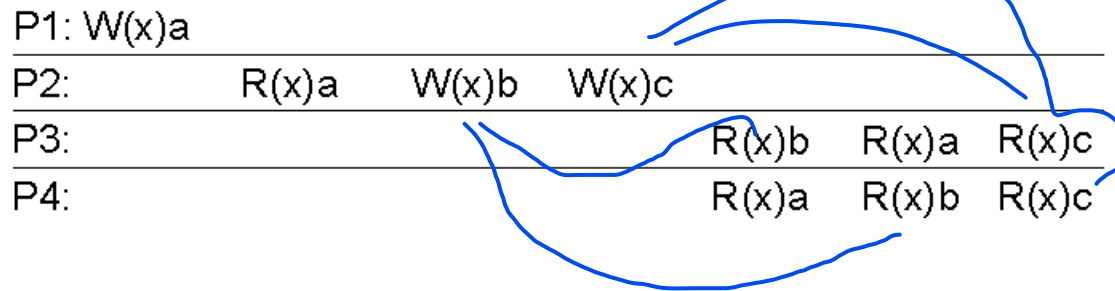
# FIFO consistency

---

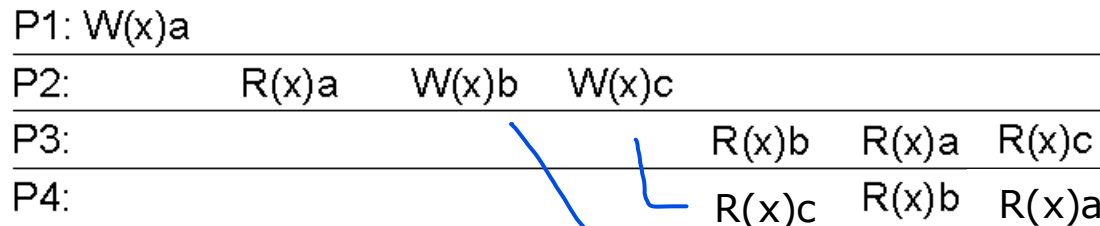
*“Writes done by a single process are seen by all others in the order in which they were issued; writes from different processes may be seen in any order at different machines”*

# FIFO consistency

- 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



Consistent



Not Consistent

Le due letture fatte rispetto a scritture fatte dallo stesso processo P2 sono in ordine inverso

# FIFO consistency: implementation

---

- Very easy to implement
  - Even with multi-leader solutions (concurrent updates)
- The updates from a process  $P$  carry a sequence number (scalar clock, no need for vector clock)
  - 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  $S$

# Consistency models and synchronization

- 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

# Consistency models and synchronization

- 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

ordered from the strongest to the weakest, notice that every consistency model adds some requirements to the one below it.

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

best effort: i retry until i reach all replicas. there are no guarantees on order of operations.

There may be moments in which replicas states diverge, but eventually they will reach the same state. For example two data center might be inconsistency for some time, then when they are merged they reach the same state but there might be conflicts.

There might be some deterministic way to resolve conflicts (es. attaching timestamp to messages)

- The models considered so far are data-centric
  - Provide a system-wide consistent data view in the presence of simultaneous, concurrent updates
- However, there are situations where there are
  - No simultaneous updates (or can be easily resolved)
  - Mostly reads
- Examples: Web caches, DNS, social media (geo-distributed data stores)

# Eventual consistency

---

- In these systems, eventual consistency is often sufficient
  - Updates are guaranteed to eventually propagate to all replicas
- Very popular today for three reasons
  - Very easy to implement
  - 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
  - Dedicated data-types (conflict-free replicated data-types)



# Eventual consistency

---

- Conflict-free replicated data types (CRDTs) guarantee convergence even if updates are received in different orders
  - Commutative semantics
- Example: integer counter
  - Replicas do not store only the last value ...
  - ... but the set of increase/decrease operations performed
  - These operations can be applied in any order in different replicas, and yield the same result

# Eventual consistency

---

- Another example
  - Append-only data structure
    - E.g., list of comments for a given post in a forum
  - Last-write-wins approach possible
    - Clients attach an identifier (e.g., a timestamp) to each append
    - In case of conflicts, the write with the larger identifier is considered as the last one
    - Eventually, all the replicas converge to the same order of elements in the data structure

# Eventual consistency

---

- 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**

# Client-centric consistency

---

- 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

# 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”*

# Monotonic reads

- Once a process reads a value from a replica, it will never see an older value from a read at a different replica

The set of writes known at a data store contains the update of  $x$  at L1 and the update of  $x$  at L2, in this order

Read on  $x_1$ , the value of  $x$  at data store L1

the value of variable  $x$  in location 1 in this point in time

L1:  $WS(x_1)$

$R(x_1)$

Consistent

L2:  $WS(x_1; x_2)$

$R(x_2)$

WS: write set (writes done by any processes)  
W: write are operations of our process

L1:  $WS(x_1)$

$R(x_1)$

NOT  
Consistent

L2:  $WS(x_2)$

$R(x_2)$   $WS(x_1; x_2)$

L1 and L2 are local copies of the data store, accessed by the same process

Rule: reads need to be monotonic: everything i read should be there

because the process has read the update  $x_1$  in the replica L1 and then after that reads the

# Monotonic writes

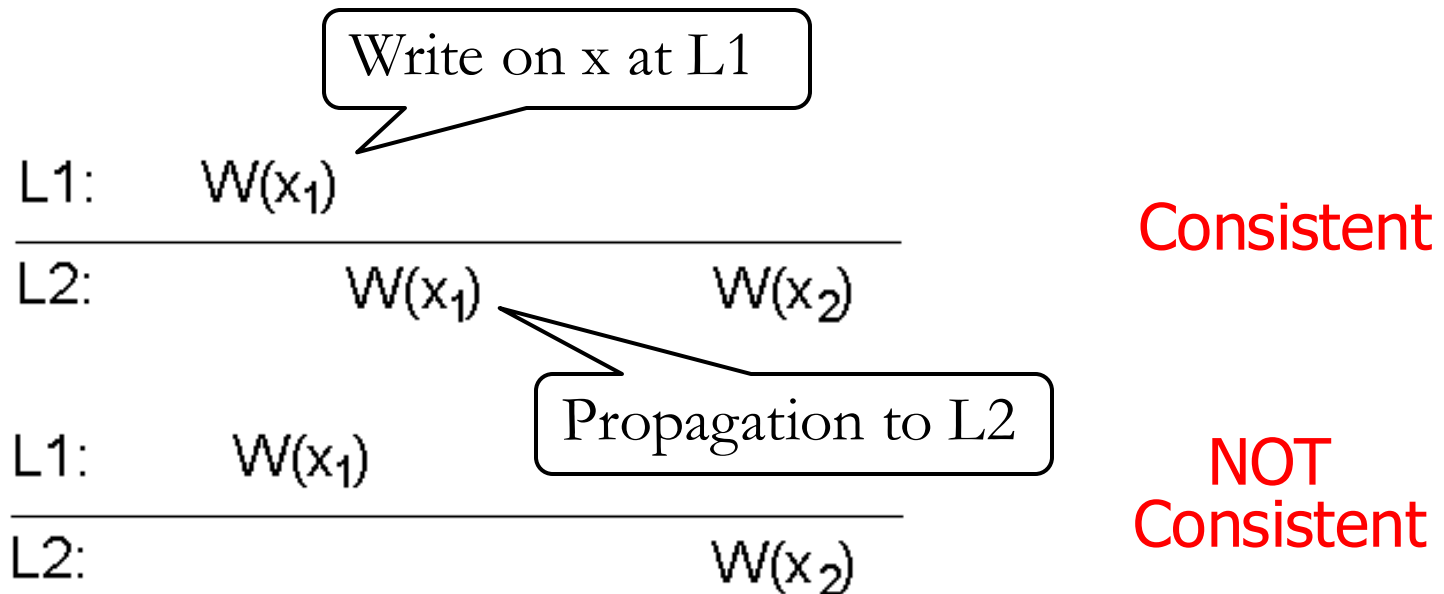
---

*“A write operation by a process on a data item  $x$  is completed before any successive write operation on  $x$  by the same process”*



# Monotonic writes

- 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”*

# Read your writes

---

- Examples: updating a Web page, or a password

L1:  $W(x_1)$

---

L2:  $WS(x_1; x_2)$   $R(x_2)$

Consistent

L1:  $W(x_1)$

---

L2:  $WS(x_2)$   $R(x_2)$

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”*

# Writes follow reads

- Example: guarantee that users of a newsgroup see the posting of a reply only after seeing the original article

L1:	WS( $x_1$ )		R( $x_1$ )
<hr/>			
L2:		WS( $x_1; x_2$ )	W( $x_2$ )

Consistent

L1:	WS( $x_1$ )		R( $x_1$ )
<hr/>			
L2:		WS( $x_2$ )	W( $x_2$ )

NOT Consistent

Does not follow the read on  $x_1$

# 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

# 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**



# Implementing replication

---

- We have seen some protocols to keep replicas consistent with respect to some consistency model
- There are further issues in designing a replicated datastore, including
  - How to place replicas?
  - What to propagate?
  - How to propagate updates between them?

# Replica placement

---

- Permanent replicas
  - Statically configured
  - E.g., DNS, CDNs, ...
- Server-initiated replicas
  - Created dynamically, e.g., to cope with access load
  - Move data closer to clients
  - Often require topological knowledge
- Client-initiated replicas
  - Rely on a client cache, that can be shared among clients for enhanced performance

# 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  $\#reads \ll \#writes$
  - Transfer the modified data to all copies
    - Works best is  $\#reads \gg \#writes$
  - 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

# Update propagation

---

- 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  $\#reads \ll \#writes$
    - Typically used to manage client caches
  - Leases can be used to control the frequency of polling
    - They were developed to deal with replication ...

# Update propagation

---

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

Replication in distributed databases

# **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



# 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

# Distributed databases

---

- 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

They use real wall-clock time to timestamp messages, they can do it because they have very expensive hardware. There's a central clock and they know what are the maximum boundaries between that clock and the data centers. So they have a synchronous distributed system.

Corbett et al. “Spanner: Google’s Globally-Distributed Database” OSDI 2012

# Case study: Spanner

---

- 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

# Case study: Calvin

---

- Designed for the same settings as Spanner
- Adopts a sequencing layer to order all incoming requests (read and write)
  - Replicated for durability
  - Essentially, a replicated log implemented using Paxos
- Operations are required to be deterministic ...
- ... and they are executed everywhere in the same order

# Case study: Calvin

---

- 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

---

- Developers specify how to partition database tables and transactions
  - E.g., hotel and flights tables both partitioned by city  
guarantee that tuples regarding the same city will be in the same partition
- Single-partition transactions may execute sequentially on that partition without coordinating with other partitions
  - Standard protocols for other transactions

# Bibliography

---

- M. Van Steen, A. S. Tanenbaum “Distributed Systems” 3<sup>rd</sup> edition, 2017
- M. Kleppmann “Designing Data-Intensive Applications: the Big Ideas Behind Reliable, Scalable, and Maintainable Systems”, 2017
- A. Margara et al. “A model and survey of distributed data-intensive systems”, ACM Computing Surveys, 2023
- P. Viotti, M. Vukolic “Consistency in Non-Transactional Distributed Storage Systems”, ACM Computing Surveys, 2016
- P. Bailis et al. “Highly Available Transactions: Virtues and Limitations”, VLDB, 2014