



Distributed Systems

Winter Term 2025/26

Roland Wismüller
Universität Siegen
roland.wismueller@uni-siegen.de
Tel.: 0271/740-4050, Büro: H-B 8404

Stand: January 8, 2026



Distributed Systems

Winter Term 2025/26

9 Replication and Consistency



Contents

- ➔ Introduction, motivation
- ➔ Distribution protocols
- ➔ Data centric consistency
- ➔ (Strong) eventual consistency
- ➔ Client centric consistency

Literature

- ➔ Tanenbaum, van Steen: Kap. 6



9.1 Introduction and Motivation

- ➔ **Replication**: several (identical) copies of data objects are stored in the distributed system
 - ➔ processes can access an arbitrary copy
- ➔ Reasons for the replication:
 - ➔ increase in availability and reliability
 - ➔ if a replica is not available, use another one
 - ➔ reading multiple replicas with majority vote
 - ➔ increase in read performance
 - ➔ for large systems: concurrent read access can be serviced by different replicas
 - ➔ with systems spread over a large area: access request is sent to a replica in the vicinity



Central Problem of Replication: Consistency

- ➔ When data is changed, **all** replicas must be kept consistent
- ➔ e.g.: send all updates to all replicas via totally ordered atomic multicast
 - ➔ high overhead when frequent updates occur
 - ➔ in some replicas these may actually never be read
 - ➔ totally ordered atomic multicast is very expensive with many / widely dispersed replicas
- ➔ Strict consistency maintenance of replicas always deteriorates performance and scalability
- ➔ Solution: weakened consistency requirements
 - ➔ often only very weak demands, e.g. News, Web, ...



Consistency Models

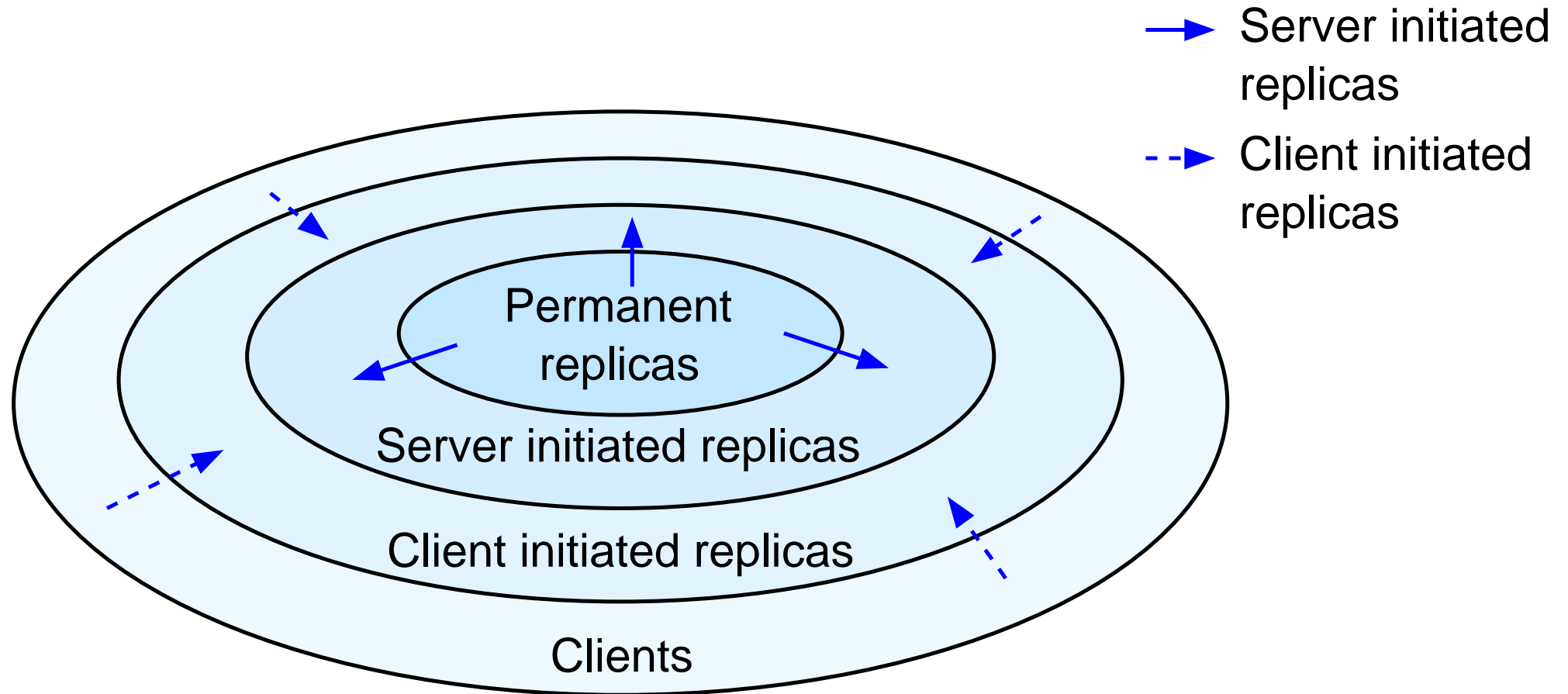
- ➔ A consistency model determines the order in which the write operations (updates) of the processes are “seen” by the other processes
- ➔ Intuitive expectation: a read operation always returns the result of the last write operation (**strict consistency**)
 - ➔ problem: there is no global time
 - ➔ pointless to speak of the “last” write operation
 - ➔ therefore: other consistency models necessary
- ➔ **Data-centric consistency models**: view of the data storage
- ➔ **Client-centric consistency models**: view of **one** process
 - ➔ assumption: (essentially) no update by multiple processes



9.2 Distribution Protocols

- ➔ Question: where, when and by whom are replicas placed?
 - ➔ permanent replicas
 - ➔ server initiated replicas
 - ➔ client initiated replicas
- ➔ Question: how are updates distributed (regardless of consistency model)
 - ➔ sending invalidations, status or operations
 - ➔ pull or push protocols
 - ➔ unicast or multicast

Placing the Replicas



→ All three types can occur simultaneously



Permanent Replicas

- ➔ Initial set of replicas, static, mostly small
- ➔ Examples:
 - ➔ replicated web site (transparent to client)
 - ➔ mirroring (client deliberately chooses a replica)

Server Initiated Replicas

- ➔ Server creates additional replicas on demand (*Push-Cache*)
 - ➔ e.g., for web hosting services
- ➔ Difficult: deciding when and where replicas will be created
 - ➔ usually access counter for each file, additional information about the origin of the requests (→ nearest server)



Client initiated Replicas

- ➔ Other term: *Client Cache*
- ➔ Client cache locally stores (frequently) used data
- ➔ Goal: improving access time
- ➔ Management of the cache is completely left to the client
 - ➔ server doesn't care about consistency
- ➔ Data is usually kept in the cache for a limited time only
 - ➔ prevents use of extremely obsolete data
- ➔ Cache usually placed on client machines, or shared cache for multiple clients in their proximity
 - ➔ e.g., web proxy caches



Forwarding Updates: What's Being Sent?

- ➔ The new value of the data object
 - ➔ good with high read/update ratio
- ➔ The update operation (active replication)
 - ➔ saves bandwidth (operation with parameters is usually small)
 - ➔ but more computing power required
- ➔ Just a notification (invalidation protocols)
 - ➔ notification makes the copy of the data object invalid
 - ➔ on next access a new copy will be requested
 - ➔ requires very little network bandwidth
 - ➔ good at low read/update ratio



Pull and Push Protocols

- ➔ **Push**: updates are distributed on the initiative of the server that made the change
 - ➔ replicas don't have to request updates
 - ➔ common in permanent and server-initiated replicas
 - ➔ when a relatively high degree of consistency is required
 - ➔ at high read/update ratio
 - ➔ problem: server must know all replicas
- ➔ **Pull**: replicas actively request data updates
 - ➔ common with client caches
 - ➔ at low read/update ratio
 - ➔ disadvantage: higher response time for cache access
- ➔ **Leases**: mixed form: first push for some time, then pull later



Unicast vs. Multicast

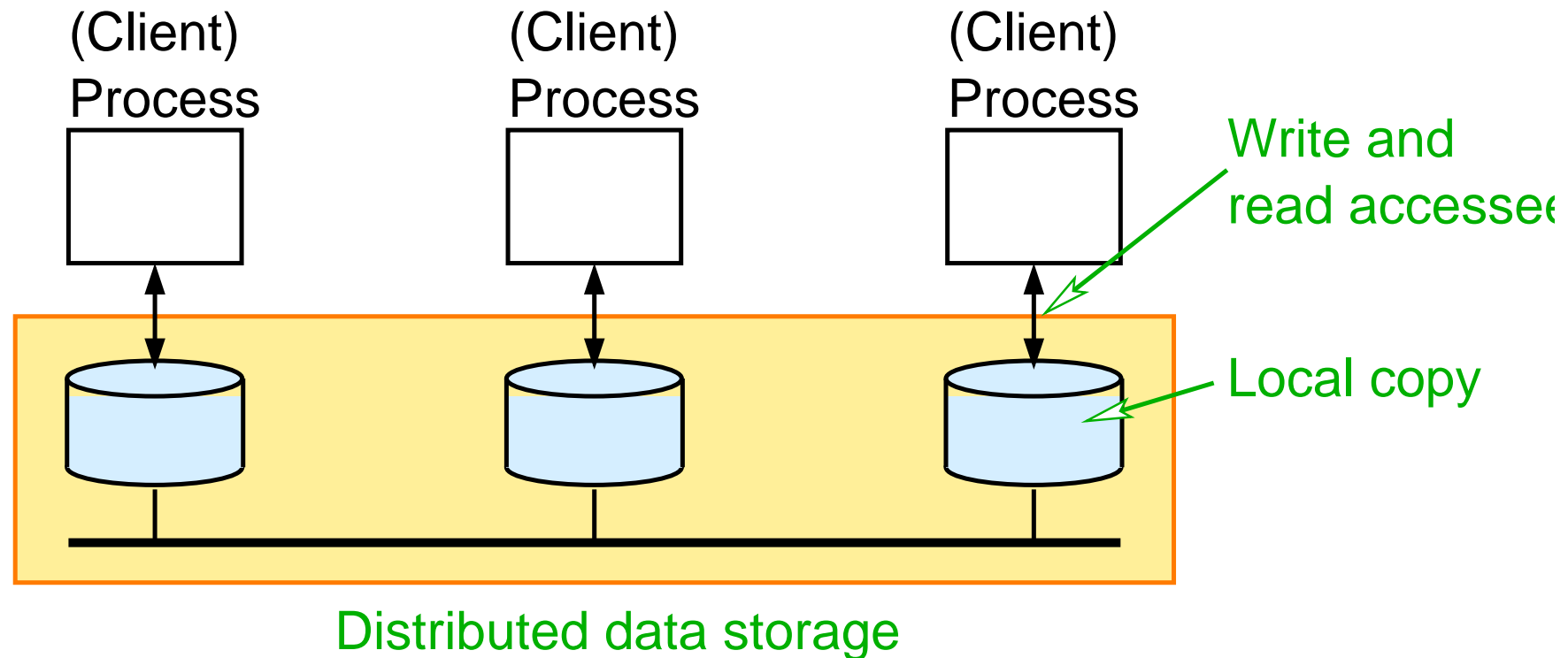
- ➔ Unicast: send update individually to each replica server
- ➔ Multicast: send one message and leave the distribution to the network (e.g. IP multicast)
 - ➔ often much more efficient
 - ➔ especially in LANs: hardware broadcast possible
- ➔ Multicast is useful for push protocols
- ➔ Unicast is better with pull protocols
 - ➔ only a single client/server requests an update

9.3 Data Centric Consistency



9.3.1 Consistency Models

Basis: model of a distributed data store



- ➔ logical, shared data memory
- ➔ physically distributed and replicated across multiple nodes

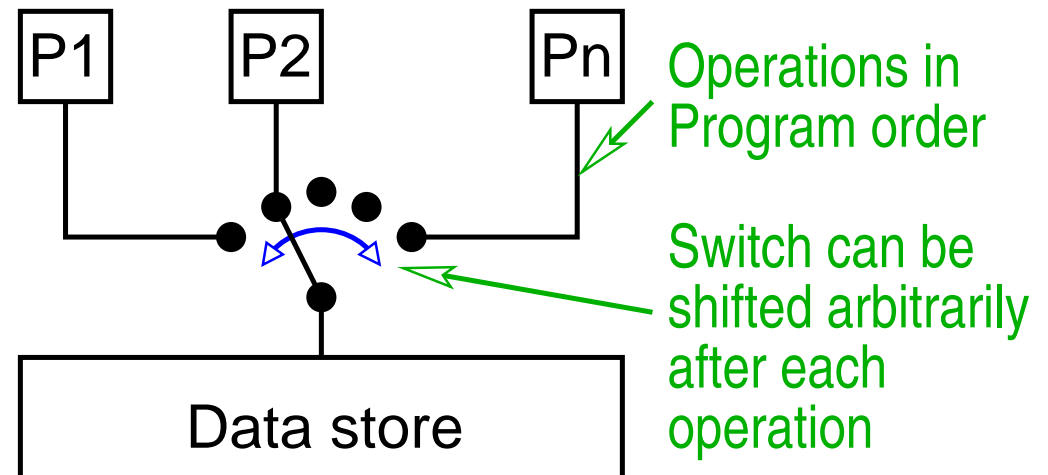
Sequential Consistency

- ➔ A data store is **sequentially consistent** if the result of each program execution is as if:
 - ➔ the (read/write) operations of all processes are executed in a (random) sequential order,
 - ➔ in which the operations of each individual process appear in the order specified by the program.

➔ I.e. the execution of the operations of the individual processes can be interleaved arbitrarily

➔ Independent of time or clocks W

➔ All processes see the (write) accesses in the same order





Sequential Consistency: Examples

Allowed sequence:

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

Forbidden Sequence:

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

➔ Notation:

➔ $W(x)a$: the value 'a' is written into the variable 'x'

➔ $R(x)a$: variable 'x' will be read, result is 'a'

➔ A possible sequential order of the left sequence:

➔ $W_2(x)b, R_3(x)b, R_4(x)b, W_1(x)a, R_3(x)a, R_4(x)a$

Sequential Consistency: Examples

Allowed sequence:

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

Forbidden Sequence:

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

➔ Notation:

➔ $W(x)a$: the value 'a' is written into the variable 'x'

➔ $R(x)a$: variable 'x' will be read, result is 'a'

➔ A possible sequential order of the left sequence:

➔ $W_2(x)b, R_3(x)b, R_4(x)b, W_1(x)a, R_3(x)a, R_4(x)a$



Linearizability

- ➔ Stronger than sequential consistency
- ➔ Assumption: the nodes (processes) have synchronized clocks
 - ➔ i.e. an approximation of a global time
- ➔ Operations have time stamps based on these clocks
- ➔ In comparison with sequential consistency additionally required:
 - ➔ the sequential order of operations is consistent with their timestamps
- ➔ Complex implementation
- ➔ Used for formal verification of concurrent algorithms



Causal Consistency

- ➔ Weakening of sequential consistency
- ➔ (Only) write operations that are potentially causally dependent must be visible to all processes in the same order

Causally, but not seq. consistent:

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

Not causally consistent:

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



Weak Consistency

- ➔ In practice: access to shared resources is coordinated via synchronization variables (SV)
- ➔ Then: weaker consistency requirements are sufficient:
 - ➔ accesses to SVs are sequentially consistent
 - ➔ an operation on a SV is not allowed until all previous write accesses to data have been completed everywhere
 - ➔ no operation on data is allowed before all previous operations on SVs have been completed

Allowed event sequence:

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

Invalid event sequence:

P1:	W(x)a	W(x)b	S
P2:		S	R(x)a



Release Consistency (*Freigabe-Konsistenz*)

- ➔ Idea as with weak consistency, but distinction between *acquire* and *release* operations (mutual exclusion!)
- ➔ before an operation on the data is performed all *acquire*-operations of the process must be completed
- ➔ before the end of a *release* operation all operations of the process on the data must be completed
- ➔ *acquire* / *release* operations of a process are seen everywhere in the same order

Allowed event sequence:

P1: *acq*(L) *W*(x)_a *W*(x)_b *rel*(L)

P2: *acq*(L) *R*(x)_b *rel*(L)

P3: *R*(x)_a



Comparison of models

Strict	Absolute time sequence of all shared accesses (physically not useful!)
Linearization	All processes see all (write) accesses in the same order. Accesses are sorted by a (non-unique) global timestamp.
Sequential	All processes see all (write) accesses in the same order. Accesses are not sorted by time.
Causal	All processes see causally linked (write) accesses in the same order.
Weak	Data is only reliably consistent after a synchronization has been performed.
Release	Data is made consistent when leaving the critical region.



9.3.2 Consistency Protocols

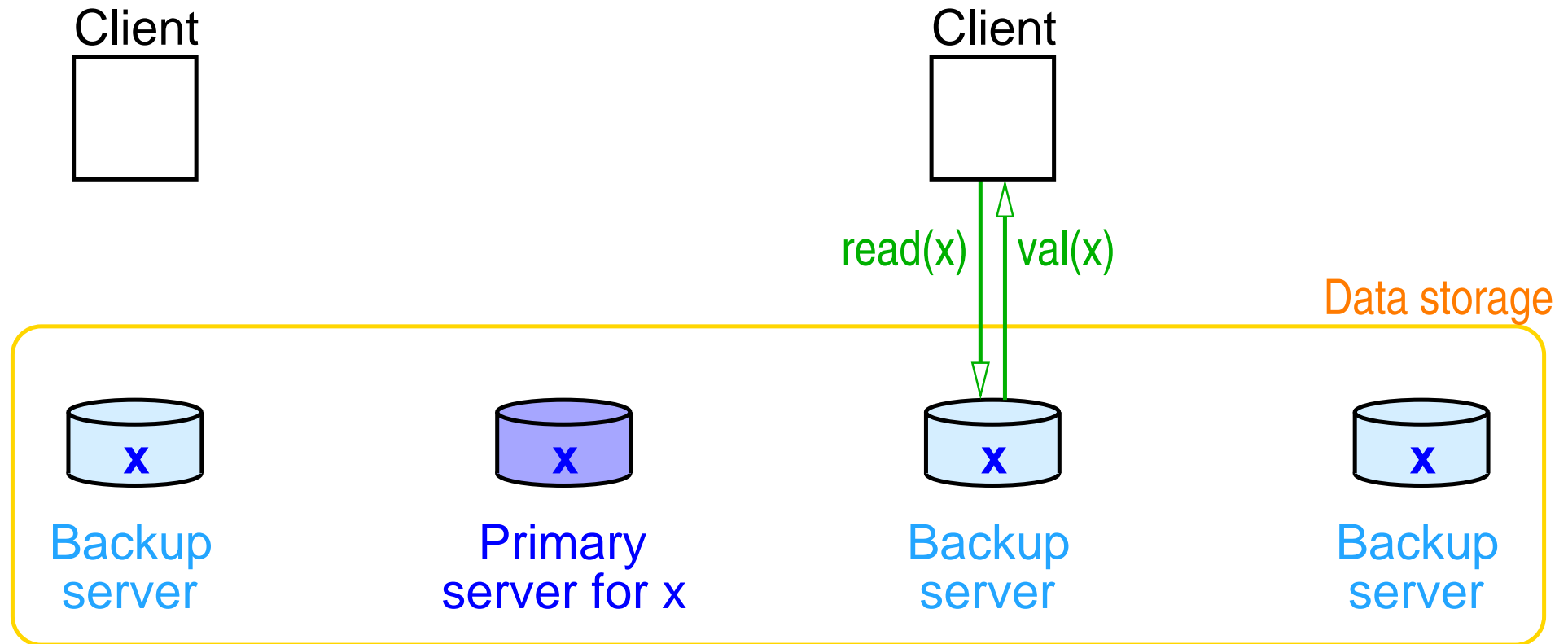
- ➔ Describe how replica servers coordinate with each other to implement a specific consistency model
- ➔ Focus in the following:
 - ➔ consistency models that globally serialize operations
 - ➔ e.g., sequential, weak and release consistency
- ➔ Two basic approaches:
 - ➔ **primary-based** (*primärbasierte*) **protocols**
 - ➔ write operations are always coordinated by a special copy (**primary copy**)
 - ➔ **replicated-write protocols**
 - ➔ write operations go to multiple copies



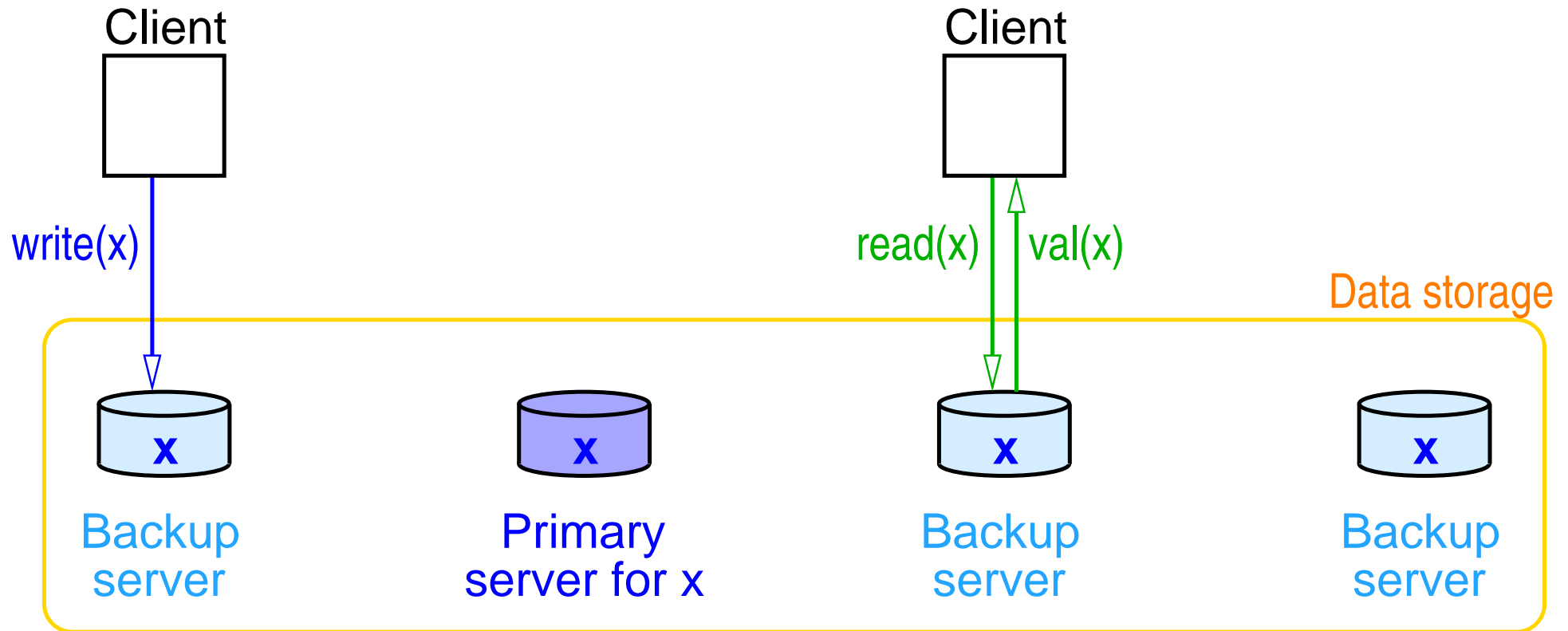
Primary-Based Protocols

- ➔ Read operations are possible on arbitrary (local) copies
- ➔ Write operations must be handled by the primary copy
 - ➔ e.g., to realize a sequential consistency:
 - ➔ the primary copy updates all other copies and waits for acknowledgements, only then it replies to the client
 - ➔ problem: performance
- ➔ **Remote-write protocols**
 - ➔ the writer forwards the operation to a fixed primary copy
- ➔ **Local-write protocols**
 - ➔ writer must become primary copy before it can do the update
 - ➔ i.e., the primary copy is migrated between servers
 - ➔ good model also for mobile users

Remote Write Protocol: Workflow (Sequential Consistency)

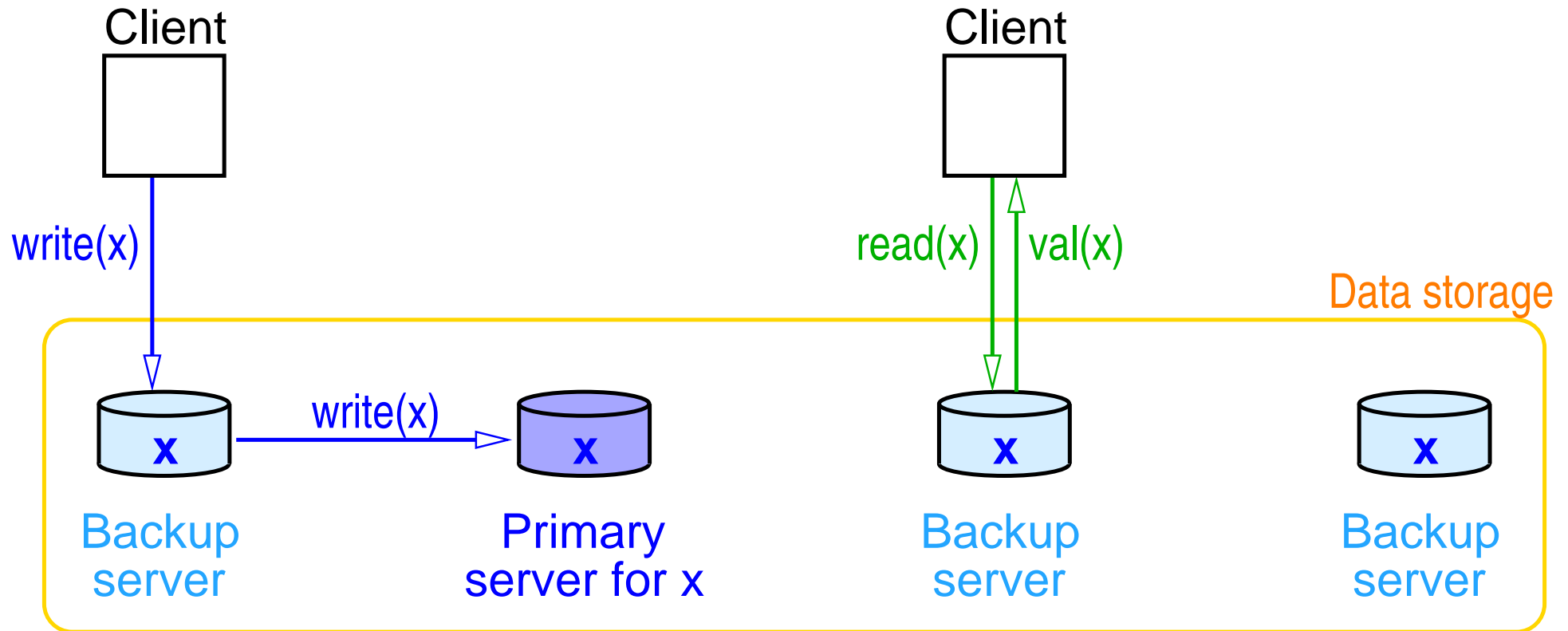


Remote Write Protocol: Workflow (Sequential Consistency)



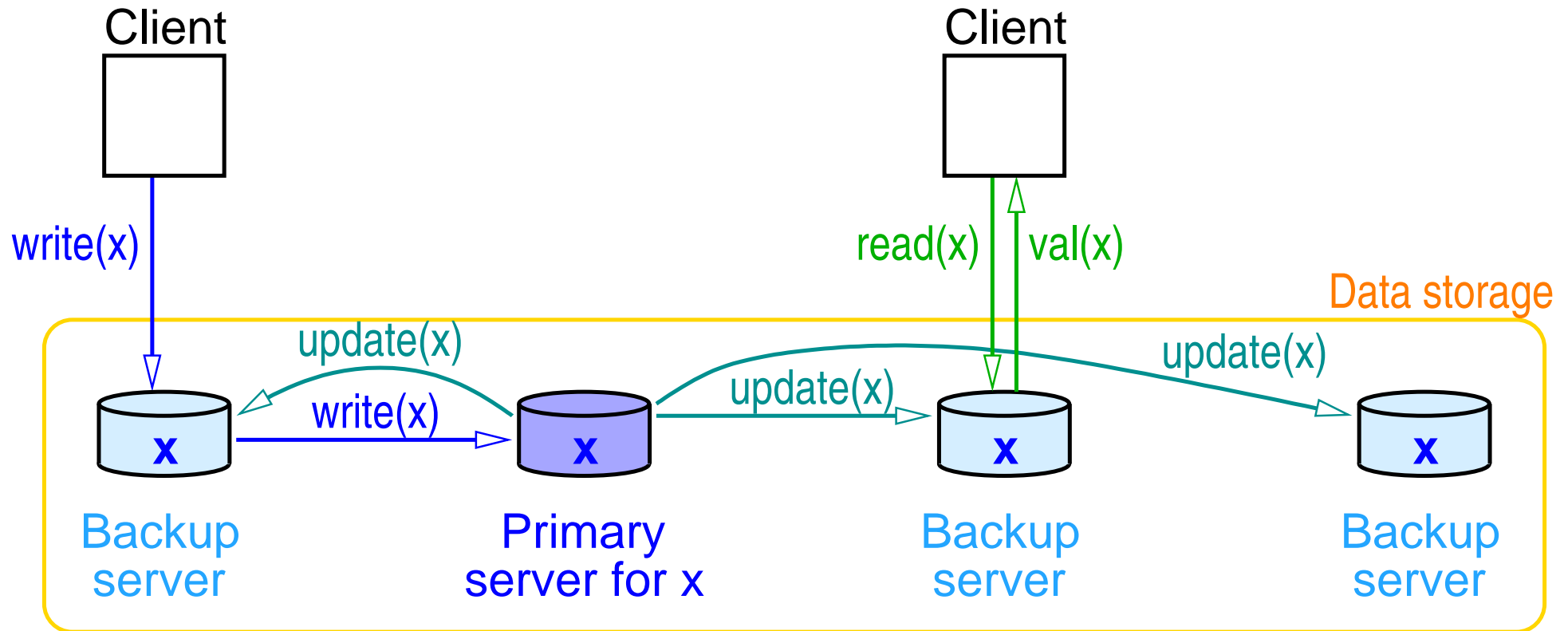
(1) Write request

Remote Write Protocol: Workflow (Sequential Consistency)



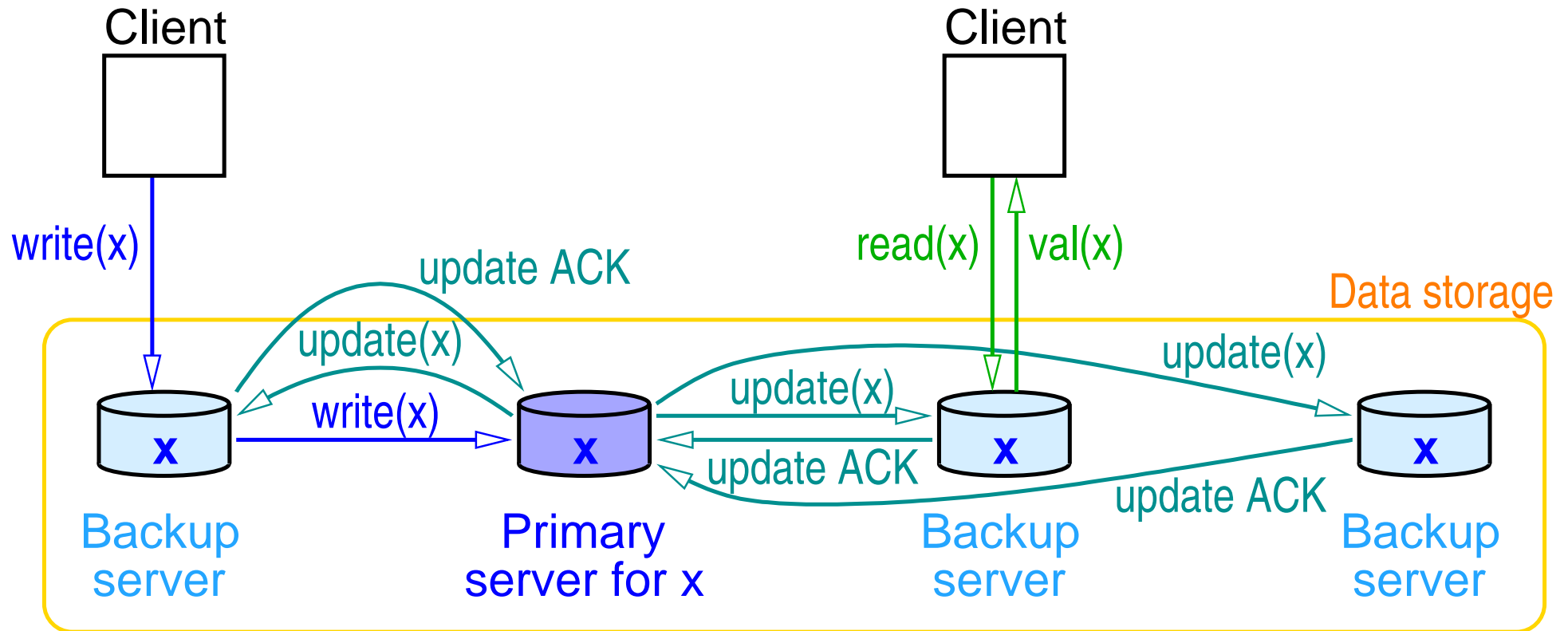
(1) Write request is forwarded to primary server

Remote Write Protocol: Workflow (Sequential Consistency)



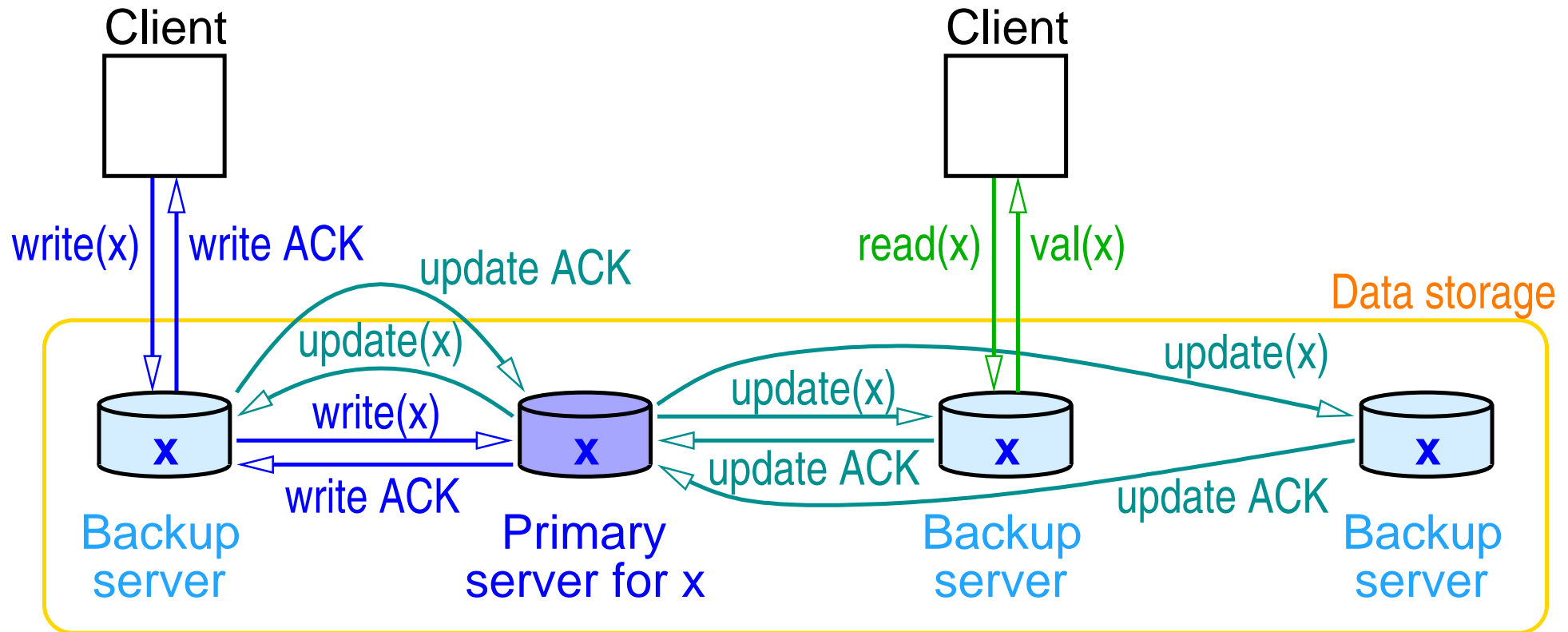
- (1) Write request is forwarded to primary server
- (2) Primary server updates all backups

Remote Write Protocol: Workflow (Sequential Consistency)



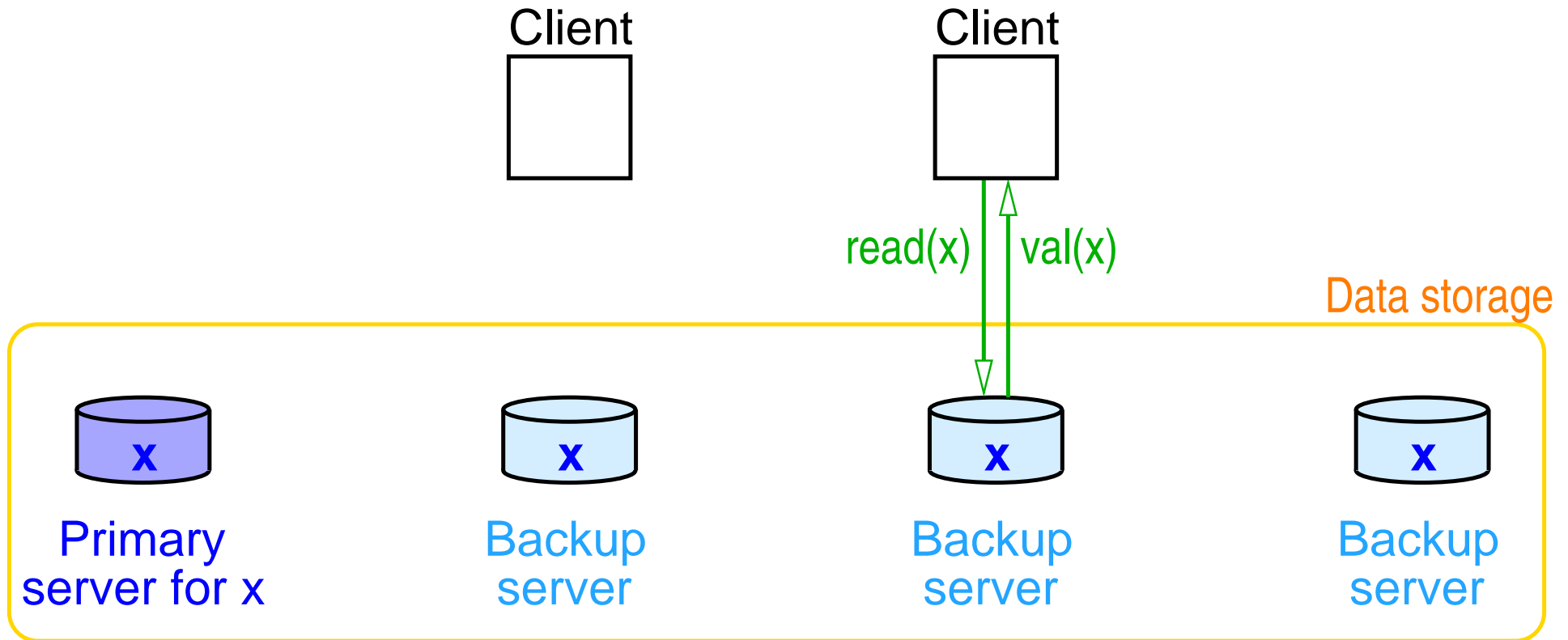
- (1) Write request is forwarded to primary server
- (2) Primary server updates all backups and waits for acknowledgements

Remote Write Protocol: Workflow (Sequential Consistency)

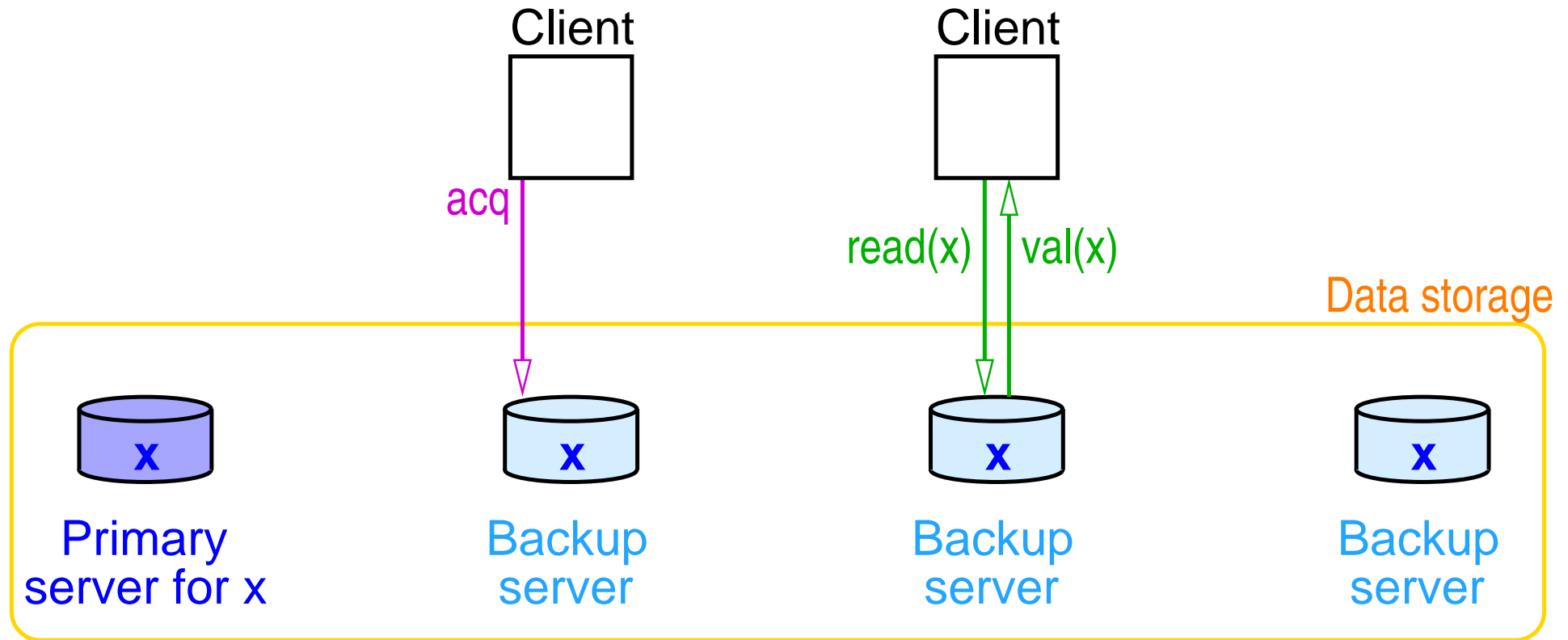


- (1) Write request is forwarded to primary server
- (2) Primary server updates all backups and waits for acknowledgements
- (3) Acknowledge the end of the write operation

Local Write Protocol: Workflow (Release Consistency)

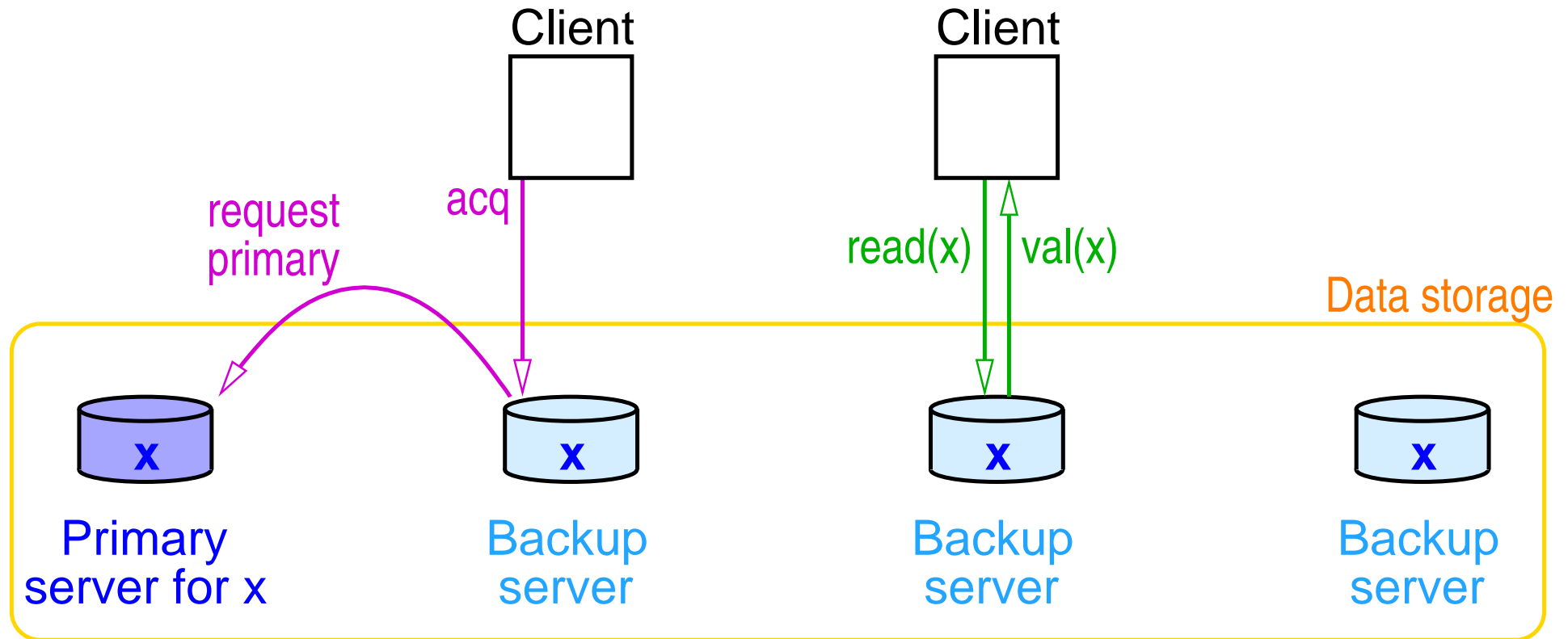


Local Write Protocol: Workflow (Release Consistency)



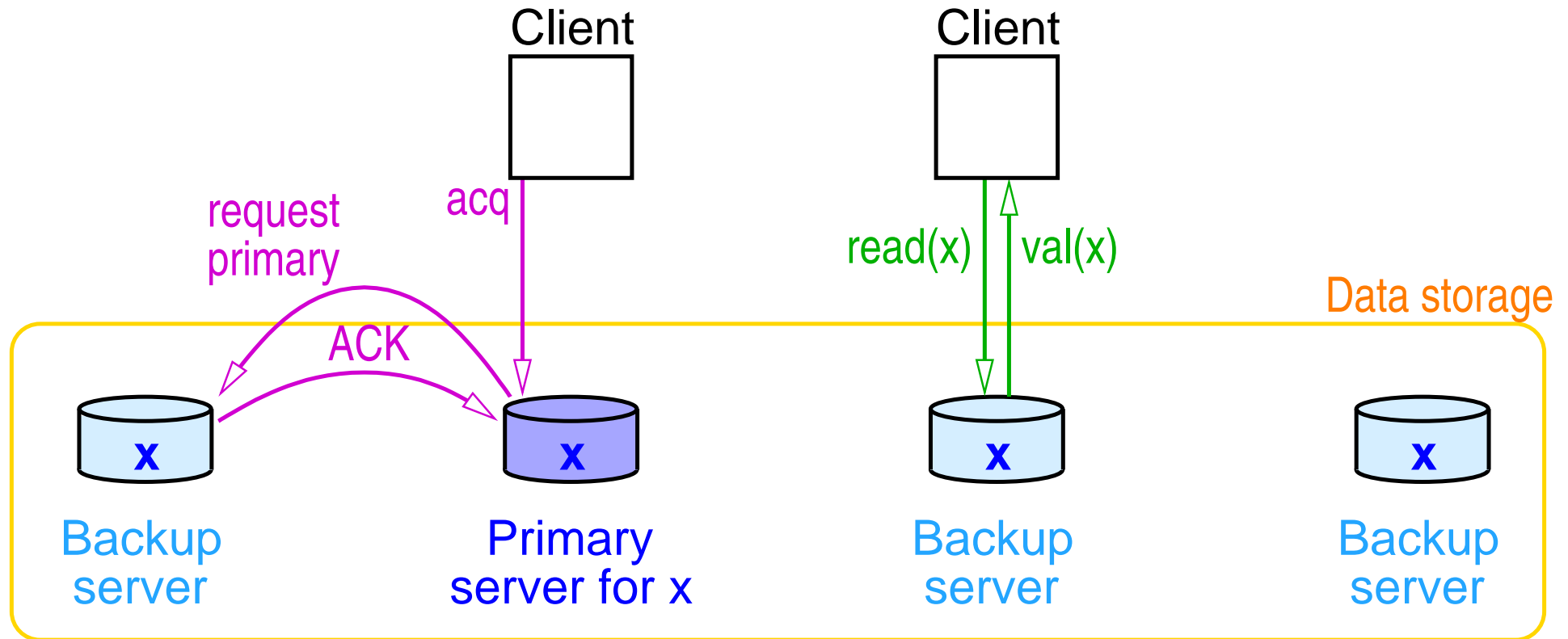
(1) Acquire lock;

Local Write Protocol: Workflow (Release Consistency)



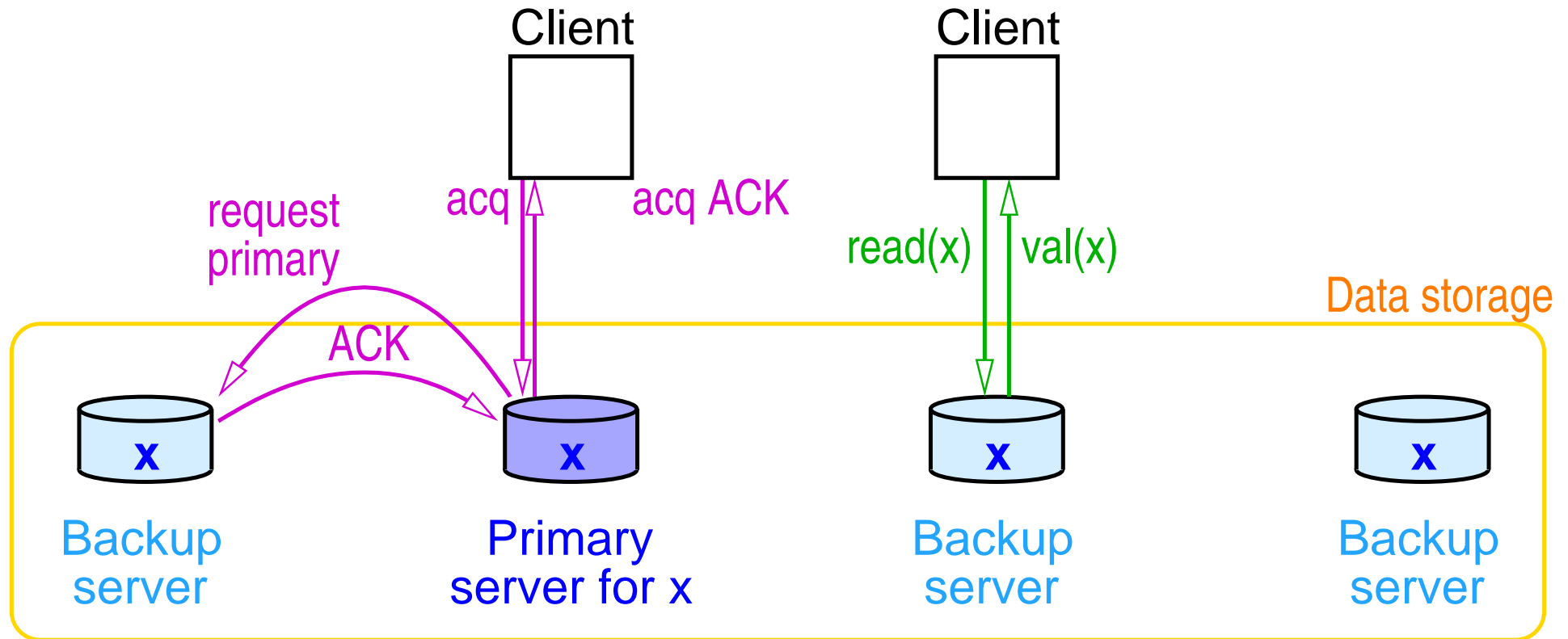
(1) Acquire lock; Move primary copy to new server

Local Write Protocol: Workflow (Release Consistency)



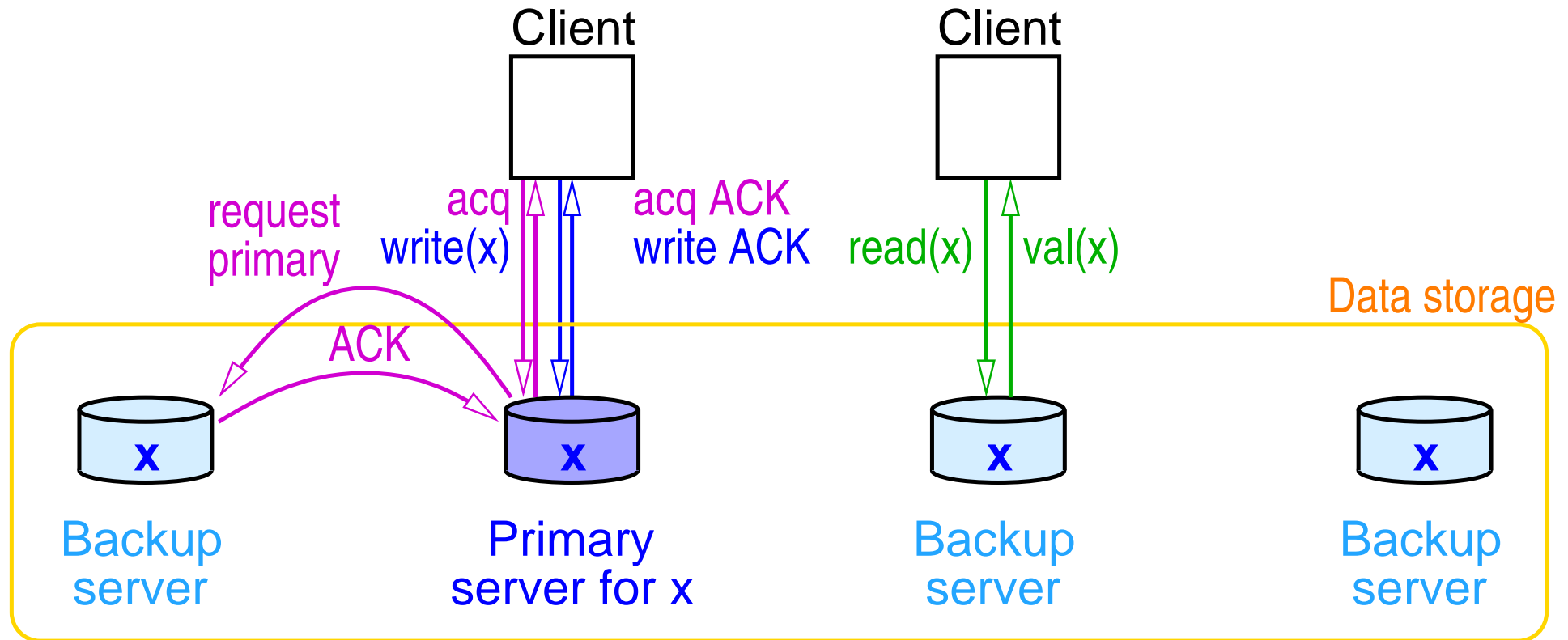
(1) Acquire lock; Move primary copy to new server

Local Write Protocol: Workflow (Release Consistency)



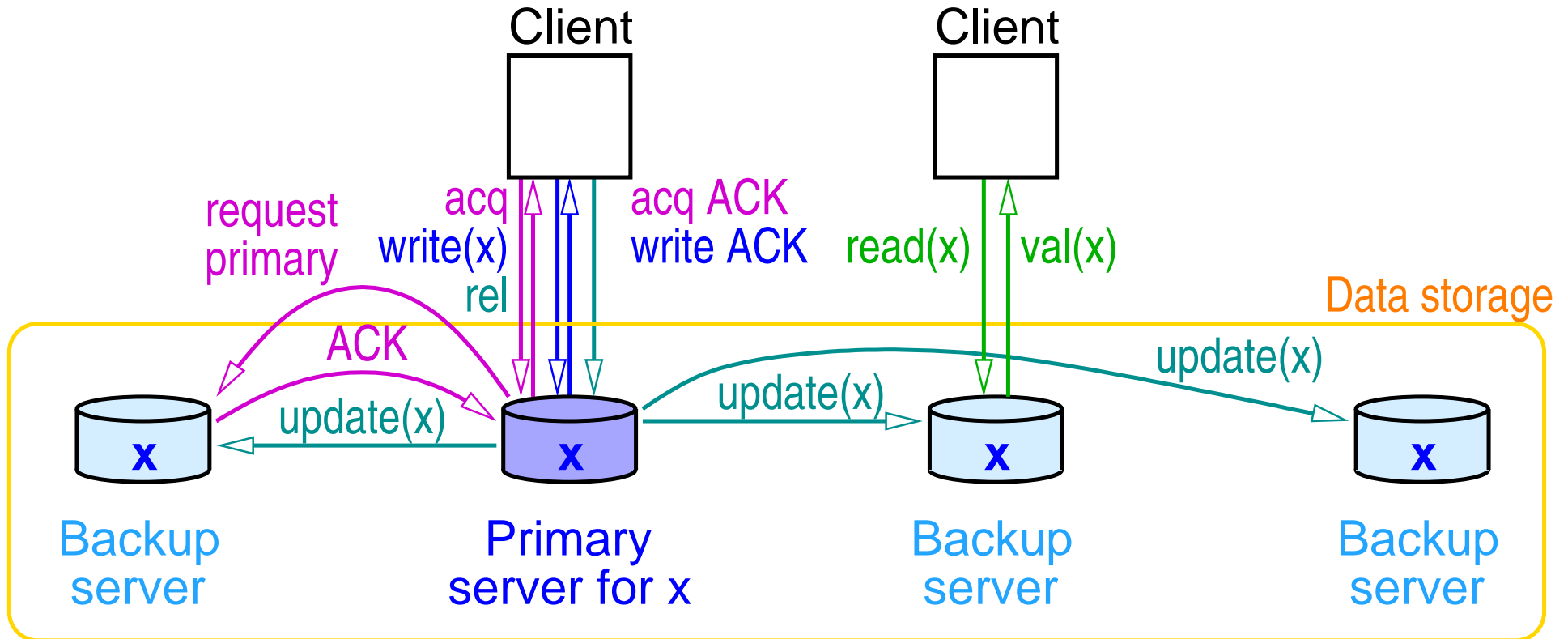
- (1) Acquire lock; Move primary copy to new server
- (2) Acknowledge the end of the write operation

Local Write Protocol: Workflow (Release Consistency)



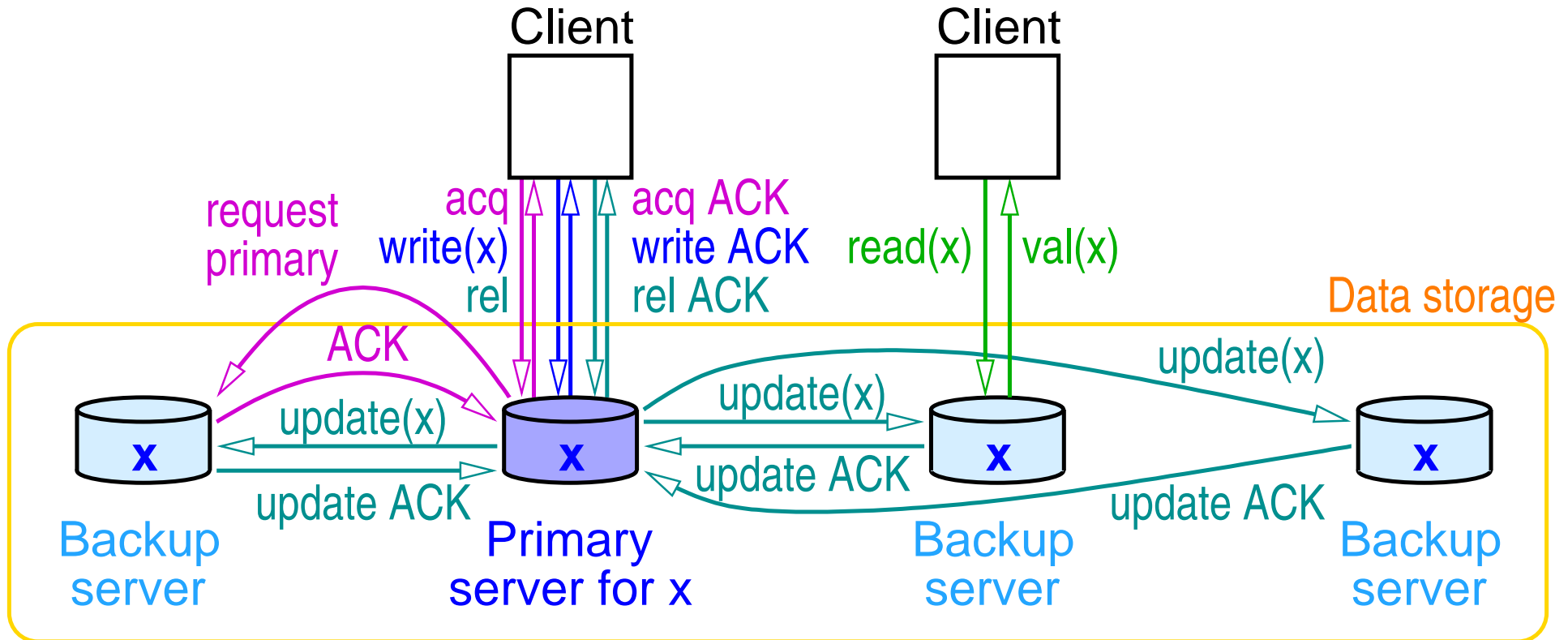
- (1) Acquire lock; Move primary copy to new server
- (2) Acknowledge the end of the write operation
- (3) Write operations are executed (only) on the local server

Local Write Protocol: Workflow (Release Consistency)



- (1) Acquire lock; Move primary copy to new server
- (2) Acknowledge the end of the write operation
- (3) Write operations are executed (only) on the local server
- (4) New primary server updates backups

Local Write Protocol: Workflow (Release Consistency)



- (1) Acquire lock; Move primary copy to new server
- (2) Acknowledge the end of the write operation
- (3) Write operations are executed (only) on the local server
- (4) New primary server updates backups and waits for acknowledgements

Distributed Systems

Winter Term 2025/26

18.12.2025

Roland Wismüller
Universität Siegen
roland.wismueller@uni-siegen.de
Tel.: 0271/740-4050, Büro: H-B 8404

Stand: January 8, 2026

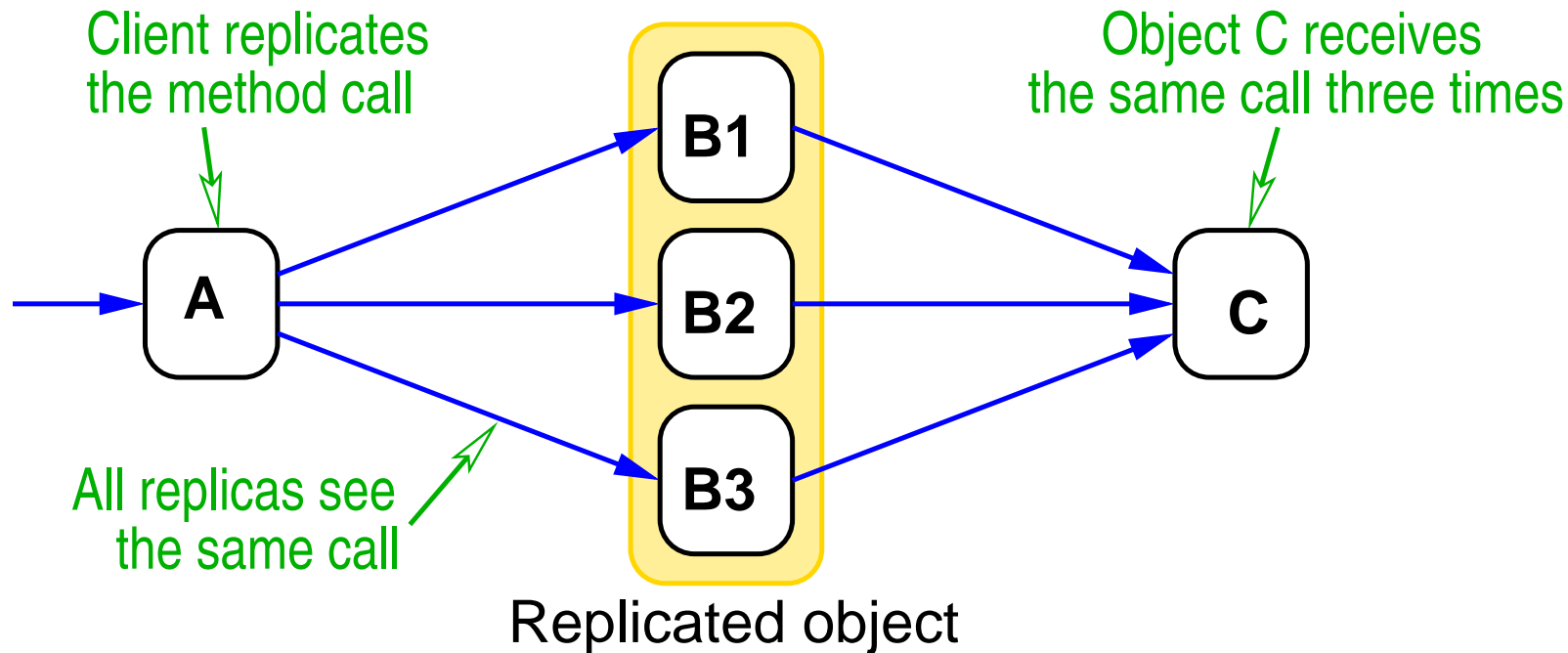


Replicated Write Protocols

- ➔ Allow execution of write operations on (multiple) arbitrary replicas
- ➔ In the following, two approaches (for sequential consistency):
 - ➔ active replication
 - ➔ update operations are passed on to all copies
 - ➔ requirement: globally unique sequence of operations
 - ➔ using totally ordered multicast
 - ➔ or via central sequencer process
 - ➔ quorum-based protocols
 - ➔ only a portion of the replicas needs to be modified
 - ➔ however, also multiple copies need to be read

Problem With Replicated Object Calls

➔ What happens when a replicated object calls another?



➔ Solution: middleware that is aware of replication

➔ coordinator of B makes sure that only one call is sent to C and its result is distributed to all replicas of B

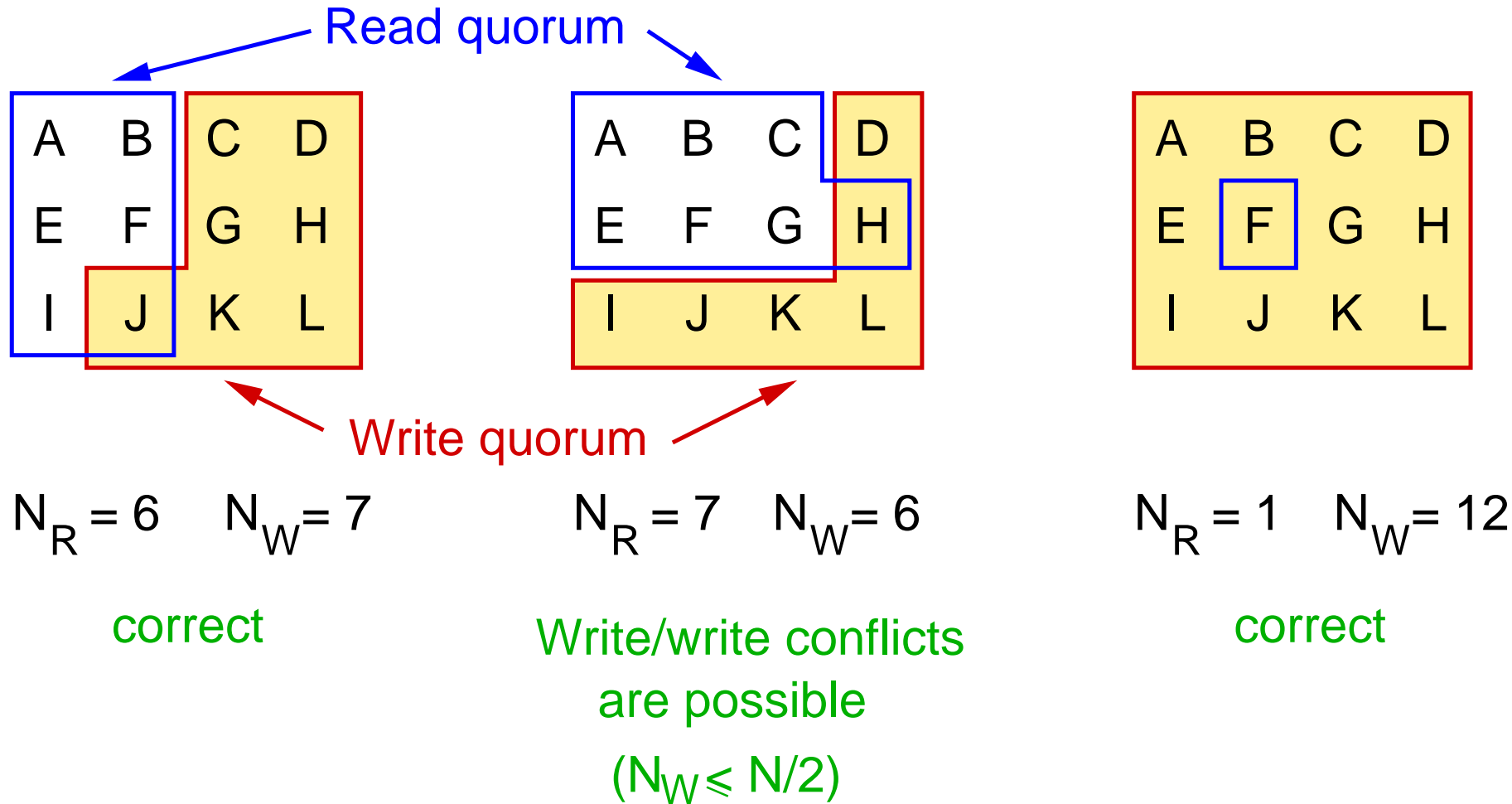


Quorum-based Protocols

- ➔ Clients need to communicate with multiple servers for writing **and** for reading
- ➔ When writing: send the request to (at least) N_W copies
 - ➔ their servers must agree to the change
 - ➔ data gets a new version number when changed
 - ➔ condition: $N_W > N/2$ (N = total number of copies)
 - ➔ prevents write/write conflicts
- ➔ When reading: send the request to (at least) N_R copies
 - ➔ client selects the latest version (highest version number)
 - ➔ condition: $N_R + N_W > N$
 - ➔ ensures that in any case the latest version is read



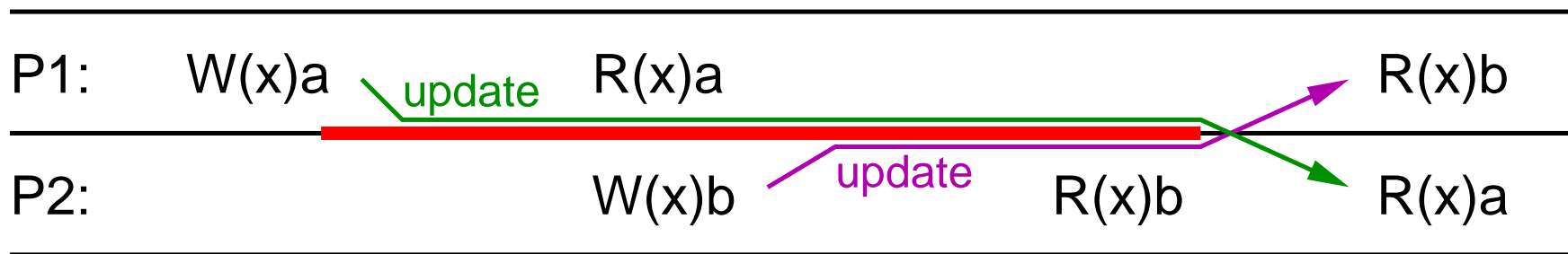
Quorum-based Protocols: Examples



9.4 (Strong) Eventual Consistency



- ➔ Problem with protocols that globally serialize operations: fault tolerance
 - ➔ esp. loss of communication / network partitioning
- ➔ Example (assuming sequential consistency):



- ➔ while P1 and P2 cannot communicate, write operations are not allowed to become visible, not even locally
- ➔ i.e. while network is partitioned, at least one partition can no longer perform read/write operations!
 - ➔ c.f. quorum based protocols!



Common situation in practice:

- ➔ Conflicting (concurrent) updates to the data are relatively rare
 - ➔ shared data store contains independent data elements
- ➔ Clients are usually independent from each other
 - ➔ i.e., they don't compare their results

Eventual Consistency

- ➔ If no new updates are made, **eventually** all correct replicas will have the same value
 - ➔ a replica is correct, if it did not (permanently) crash
- ➔ In case of conflicting updates, conflict resolution is needed
 - ➔ i.e., updates must be 'merged'
 - ➔ may require consensus and/or rollback



Strong Eventual Consistency

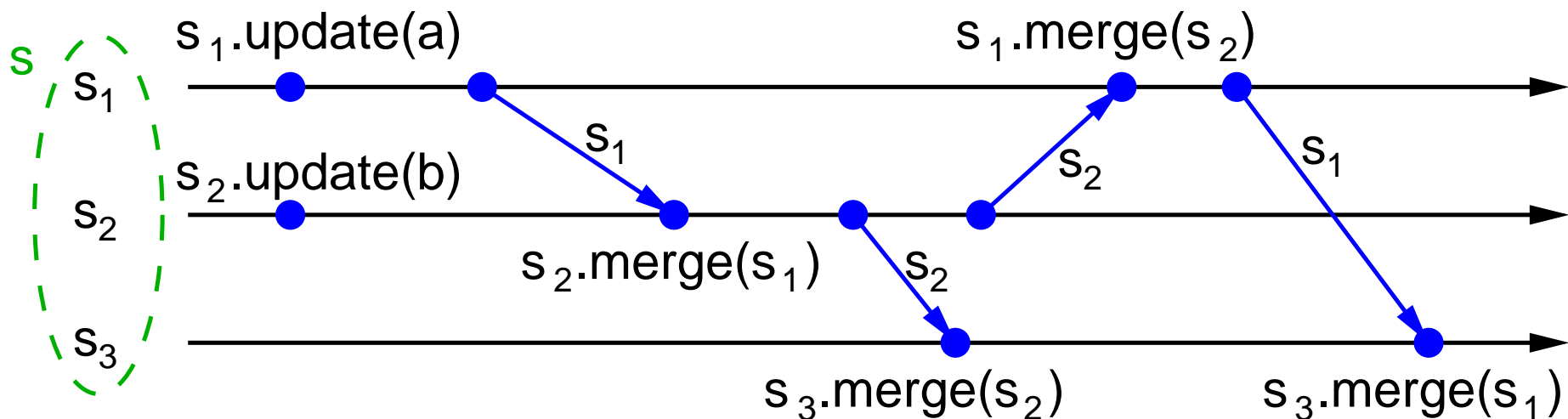
- ➔ Adds safety guarantee: if two replicas have seen the same set of updates, they are identical
- ➔ Does not require conflict resolution or rollbacks
- ➔ Ensures safety and liveness despite any number of non-byzantine failures

Conflict-Free Replicated Datatypes (CRDTs)

- ➔ Pattern for distributed data structures that guarantees strong eventual consistency by design
- ➔ No synchronisation required
- ➔ Examples: counters, lists, sets, editable text documents, ...
- ➔ Important application: collaborative editing of documents

State-based (Convergent) Replicated Datatypes

- ➔ Object with the following methods (executing on a single replica):
 - ➔ *init* (constructor): sets initial object state
 - ➔ *value*: returns the value of the object
 - ➔ *update*: modifies the object
 - ➔ *merge*: merges state from a remote replica into the local state
- ➔ Each replica occasionally send its state to the other replicas





State-based (Convergent) Replicated Datatypes ...

- ➔ Sufficient conditions for strong eventual consistency:
 - ➔ all methods executions terminate
 - ➔ all updates are eventually delivered (directly or indirectly) to all correct replicas
 - ➔ the set S of possible object values is a join semilattice
 - ➔ i.e., there is a partial order \leq on S , such that any two elements $x, y \in S$ have a least upper bound $x \sqcup y$
 - ➔ $x.merge(y) = x \sqcup y$
 - ➔ state is monotonically non-decreasing across updates, i.e., $s \leq s.update(a)$



Example: State-based Grow-only Counter

```
class GCounter:  
    def __init__(self, n):  
        self.val = [0] * n # Vector: one element per replica  
    def value(self):  
        return sum(self.val)  
    def increment(self):  
        self.val[myReplicaID()] += 1  
    def merge(self, other):  
        # component-wise maximum  
        self.val = [max(x,y) for x,y in zip(self.val, other.val)]
```

➔ The (partial) order on vectors is defined component-wise:

$$a \leq b \Leftrightarrow \forall i \in [0, n - 1] : a_i \leq b_i$$

➔ Very similar to a vector timestamp (👉 6.3)



Example: State-based Positive-Negative Counter

- ➔ Problem: we cannot decrement the local counter, as this violates the monotonicity requirement
- ➔ Solution: compose two grow-only counters

```
class PNCounter:
    def __init__(self, n):
        self.inc, self.dec = GCounter(n), GCounter(n)
    def value(self):
        return self.inc.value() - self.dec.value()
    def increment(self):
        self.inc.increment()
    def decrement(self):
        self.dec.increment()
    def merge(self, other):
        self.inc, self.dec = self.inc.merge(other.inc), \
            self.dec.merge(other.dec)
```



Operation-based (Commutative) Replicated Datatypes

➔ Object with the following methods:

➔ *init* (constructor): sets initial object state

➔ *value*: returns the value of the object

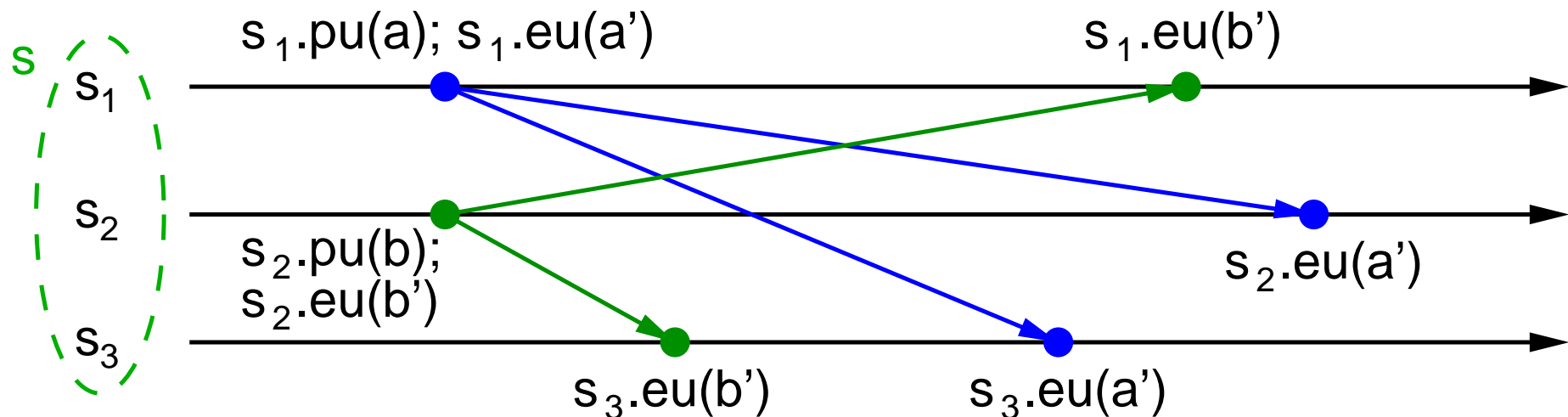
➔ *prepare_update*: prepare an update

➔ is immediately followed by *effect_update*

➔ *effect_update*: performs the update

execute on
single replica

executes on all replicas





Operation-based (Commutative) Replicated Datatypes ...

- ➔ Sufficient conditions for strong eventual consistency:
 - ➔ all methods executions terminate
 - ➔ communication uses a reliable, causally ordered multicast
 - ➔ allows concurrent updates to be delivered in any order
 - ➔ concurrent updates commute
 - ➔ i.e., the order of execution doesn't matter

- ➔ Note: if *all* updates commute, a reliable, unordered multicast is sufficient



Example: Operation-based Positive-Negative Counter

```
class Counter:
    def __init__():
        counter = 0
    def value():
        return counter
    def effect_increment():      # no prepare method necessary
        counter += 1
    def effect_decrement():    # no prepare method necessary
        counter -= 1
```

- ➔ Easier than state-based counter, but requires a multicast whenever counter is incremented / decremented
- ➔ Reliable, unordered multicast is sufficient, as updates always commute



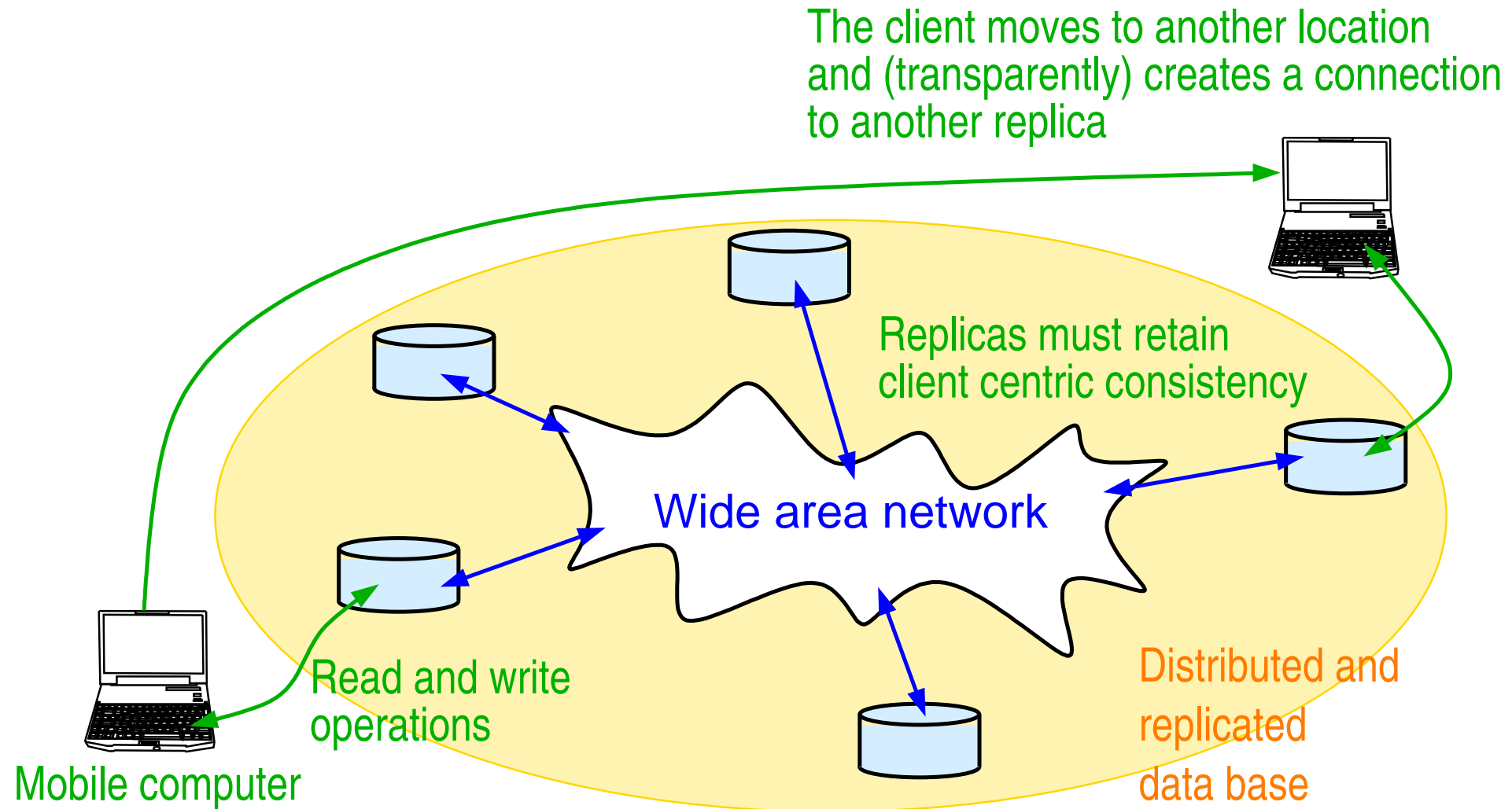
Some more CRDTs

- ➔ Grow-only set
 - ➔ similar to grow-only counter
- ➔ Two-phase set: supports both add and remove operations
 - ➔ similar to positive-negative counter: removed objects are collected in a *tombstone* set
- ➔ Sequence CRDTs
 - ➔ allow distributed, collaborative real-time editing of documents
 - ➔ basic idea:
 - ➔ each element has a (globally) unique identifier
 - ➔ identifier space is dense and totally ordered
 - ➔ order of identifiers determines element sequence
 - ➔ ensures that concurrent insert / remove operations commute



- ➔ Eventual consistency may create problems, when a client changes the replica it is accessing
 - ➔ updates may not have arrived there yet
 - ➔ client experiences inconsistent behavior
- ➔ Solution: client-centric consistency models
 - ➔ guarantee consistency for an **individual** client
 - ➔ but not for concurrent accesses by multiple clients
- ➔ Models (see Tanenbaum / van Steen, Ch. 6.3):
 - ➔ Monotonic Read
 - ➔ when a client reads a variable x , every subsequent read of x returns the same or a more recent value
 - ➔ Monotonic Write
 - ➔ Read Your Writes
 - ➔ Writes Follow Reads

Illustration of the problem





9.6 Summary

- ➔ Replication due to availability and performance
- ➔ Problem: consistency of copies
 - ➔ strictest model: sequential consistency
 - ➔ weakenings: causal consistency, weak \sim , release \sim
 - ➔ (strong) eventual consistency
 - ➔ client-centric consistency models
- ➔ Implementation of replication and consistency:
 - ➔ replication scheme: static, server initiated, client initiated
 - ➔ distribution protocols
 - ➔ type of update, push / pull, unicast / multicast
 - ➔ consistency protocols
 - ➔ primary based / replicated write / CRDTs