# Algorithms and Distributed Systems 2023/2024 (Lecture Nine)

**MIEI - Integrated Master in Computer Science and Informatics**

**MEI – Master in Computer Science and Informatics**

Specialization block

**Nuno Preguiça** (nmp@fct.unl.pt)

Alex Davidson (a.davidson@fct.unl.pt)

NOVA SCHOOL OF SCIENCE & TECHNOLOGY

**Based on slides from João Leitão**

# Lecture structure:

- Recap on pitfalls of eventual consistency
- CAP Theorem
- Causal Consistency
- ChainReplication

# Quiz!!1!

**What do you remember?**

# Question 1

Serializability guarantees:

- Strong consistency: via a time-based ordering of operations
- Eventual consistency
- Strong consistency: via a total (sequential) ordering of operations

# Answer 1

Serializability guarantees:

- Strong consistency: via a time-based ordering of operations
- Eventual consistency
- ***Strong consistency: via a total (sequential) ordering of operations***

# Question 2

In a system with high-latency communication between **replicas**, performance is more impacted in:

- Strongly consistent systems
- Eventually consistent systems

# Answer 2

In a system with high-latency communication between **replicas**, performance is more impacted in:

- ***Strongly consistent systems: strongly consistent systems typically need to perform replication (somewhat) synchronously***

- Eventually consistent systems

# Summary: Eventual Consistency

- Imposes no restrictions to the observable state of the system by clients.

- Benefits:
  - Allows for faster response times for clients (by avoiding server replicas to coordinate with each client operation)
  - Improves availability (broader conditions in which the system is able to process client requests)

# Negatives of Eventual Consistency

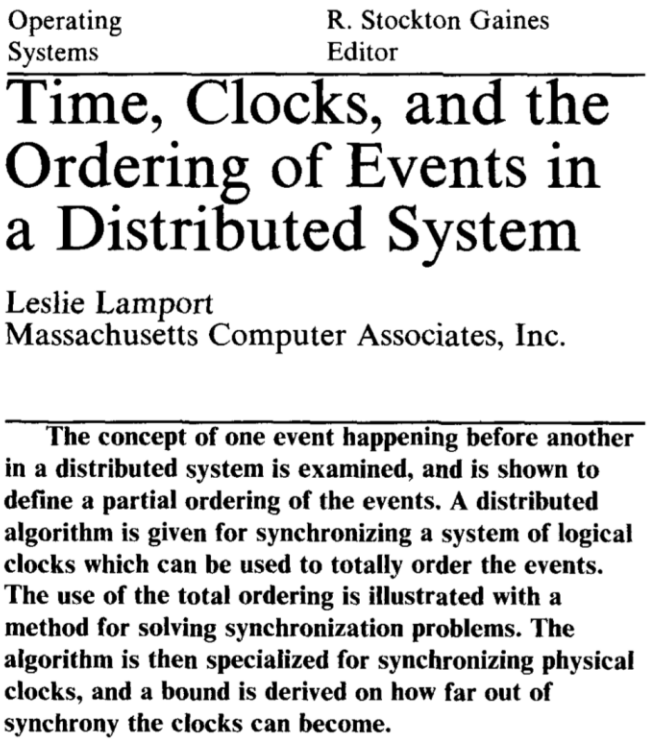
The behaviour of the system might be hard to grasp for users/developers.

- In particular, a sequence of reads executed by a client might exhibit state that is not consistent with the evolution of time perceived by the user.

- Why is this bad?
  - Non-commutative operations
  - Usability of system

# The happens before relationship… (in systems)

Lamport, 1978



Operating Systems — R. Stockton Gaines Editor

**Time, Clocks, and the Ordering of Events in a Distributed System**

Leslie Lamport
Massachusetts Computer Associates, Inc.

**The concept of one event happening before another in a distributed system is examined, and is shown to define a partial ordering of the events. A distributed algorithm is given for synchronizing a system of logical clocks which can be used to totally order the events. The use of the total ordering is illustrated with a method for solving synchronization problems. The algorithm is then specialized for synchronizing physical clocks, and a bound is derived on how far out of synchrony the clocks can become.**

- A common restriction of many consistency systems is related with ensuring that clients observe the state of the system evolving according to some order that respects the *"happens-before"* relationship.
- Unfortunately, this is not a requirement in eventual consistency.

# In practice why can this be bad?

# In practice why can this be bad?

- An example close to your daily life: Social Networks

- Imagine a system like Facebook or <insert_social_media_platform_young_people_use>.

# In practice why can this be bad?

- An example close to your daily life: Social Networks

- Now imagine that there are two relevant data objects: a) your posts; b) access list to your posts.

Posts: <Lots of innocent stuff>

Access list: <Lots of people>

# In practice why can this be bad?

- An example close to your daily life: Social Networks

- Further imagine that your professor is currently allowed to see your posts (i.e., is in the access list).

Posts: <Lots of innocent stuff>

Access list: <Lots of people> <Professor>

# In practice why can this be bad?

You went to a party in the night before the class, where imagine, you had to deliver a homework…

You are in no way ready to go to the class, but since you don’t want to tell the truth about why you cannot go, you send an e-mail claiming that you are sick... However, you also want to post that “sick” picture of the party last night in your social media profile...

After sending the e-mail what do you do?

# In practice why can this be bad?

- An example close to your daily life: Social Networks
- First you remove the professor from the access list.

Posts: <Lots of innocent stuff>

Access list: <Lots of people> ~~<Professor>~~

# In practice why can this be bad?

- An example close to your daily life: Social Networks

- Then you post that amazing picture from last night.

Posts: <Lots of innocent stuff>

Access list: <Lots of people> ~~<Professor>~~

# In practice why can this be bad?

- An example close to your daily life: Social Networks
- And you are safe right?

Posts: <Lots of innocent stuff>

Access list: <Lots of people> ~~<Professor>~~

# In practice why can this be bad?

- An example close to your daily life: Social Networks
- Well not exactly… if Facebook is providing only eventual consistency...

Posts: <Lots of innocent stuff>

Access list: <Lots of people> ~~<Professor>~~

# In practice why can this be bad?

- An example close to your daily life: Social Networks
- How does the (application server) create the Wall of a user?

Posts: <Lots of innocent stuff>

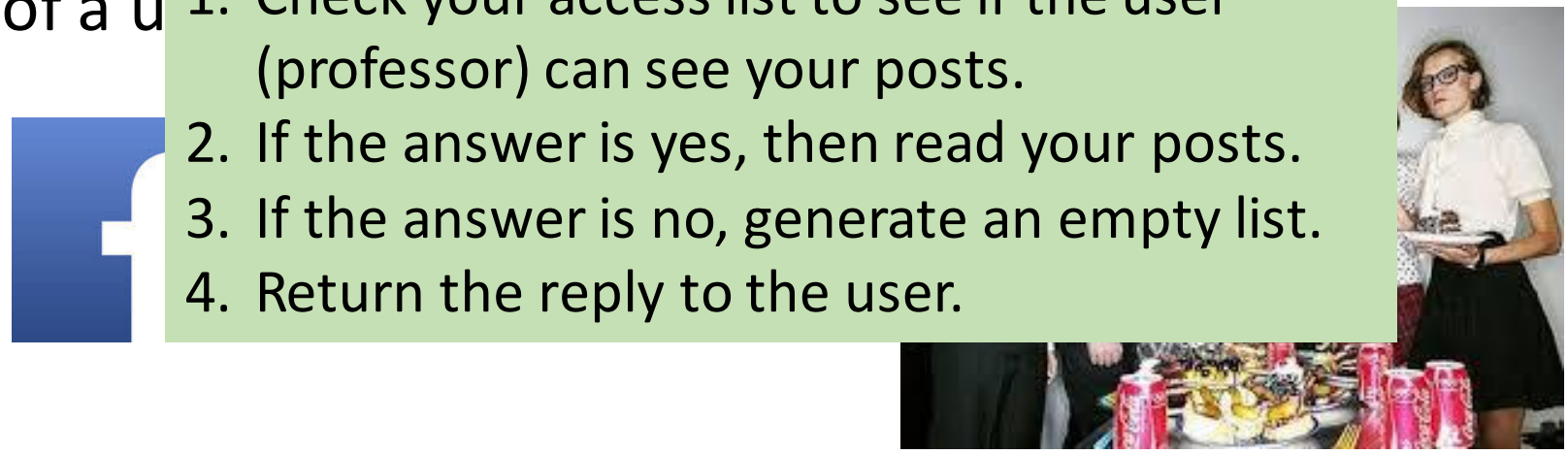
Access list: <Lots of people> ~~<Professor>~~

# In practice why can this be bad?

- An ex… …orks
- How … …Wall of a u…

When the (professor) client application issues a read for your posts list, the application server will execute the following steps:

1. Check your access list to see if the user (professor) can see your posts.
2. If the answer is yes, then read your posts.
3. If the answer is no, generate an empty list.
4. Return the reply to the user.



Access list: <Lots of people> <Professor>

# In practice why can this be bad?

- An ex… orks
- How … Wall of a u…

Unfortunately, this can go wrong…

Upon checking the access list the application server is allowed to see an outdated version of the list that still has ”target” user in the list.

The following read over the post list however, can return the most recent version containing ”that” picture.

Your intentions were defeated by eventual consistency.

# So, what can we do to overcome this sort of scenario...

- Simple strategy is to enforce the "happens before" relationship in the state, that is observed by two read operations.

- (Even with that you need to be careful on how you access the different data objects… more on that later on)

# A simple way to do this:

- Scrap this notion of using weak consistency and rely instead in some form of strong consistency.
- Strong consistency is the best ☺
- Wait, why did we leave strong consistency?

# A simple way to do this:

- Scrap this notion of using weak consistency and rely instead in some form of strong consistency.
- Strong consistency is the best ☺
- Wait, why did we leave strong consistency?
  - Strong consistency depends on total ordering of all client operations:
    - Increased response times (high latency)
    - Operations cannot happen if a majority of replicas is not reachable.

# A simple way to do this:

- Scrap this notion of using weak consistency and rely instead in some form of strong consistency.

Clients can wait… (actually they don't...)

Majority of replicas will always be available...

What are the odds really?

operations:

- Increased response times (high latency)
- Operations cannot happen if a majority of replicas is not reachable.

# A simple way to do this:

- Scrap this notion of using weak consistency and rely instead in some form of strong consistency.

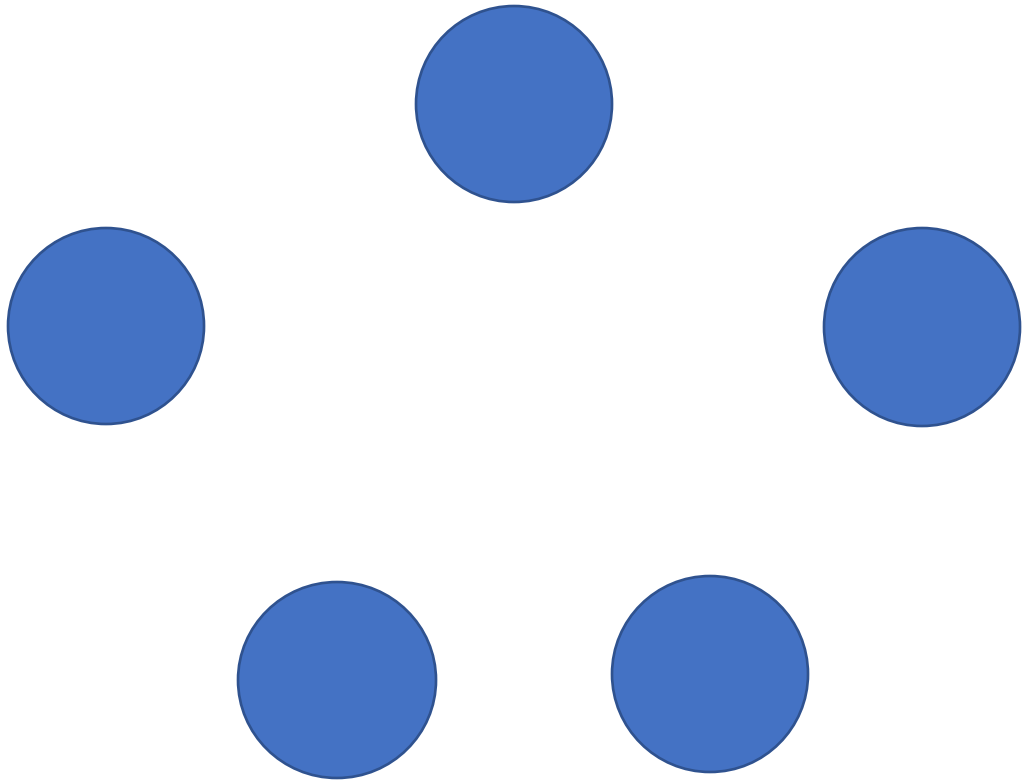
Clients can wait?

Majority of replicas will always be available...
What are the odds really?

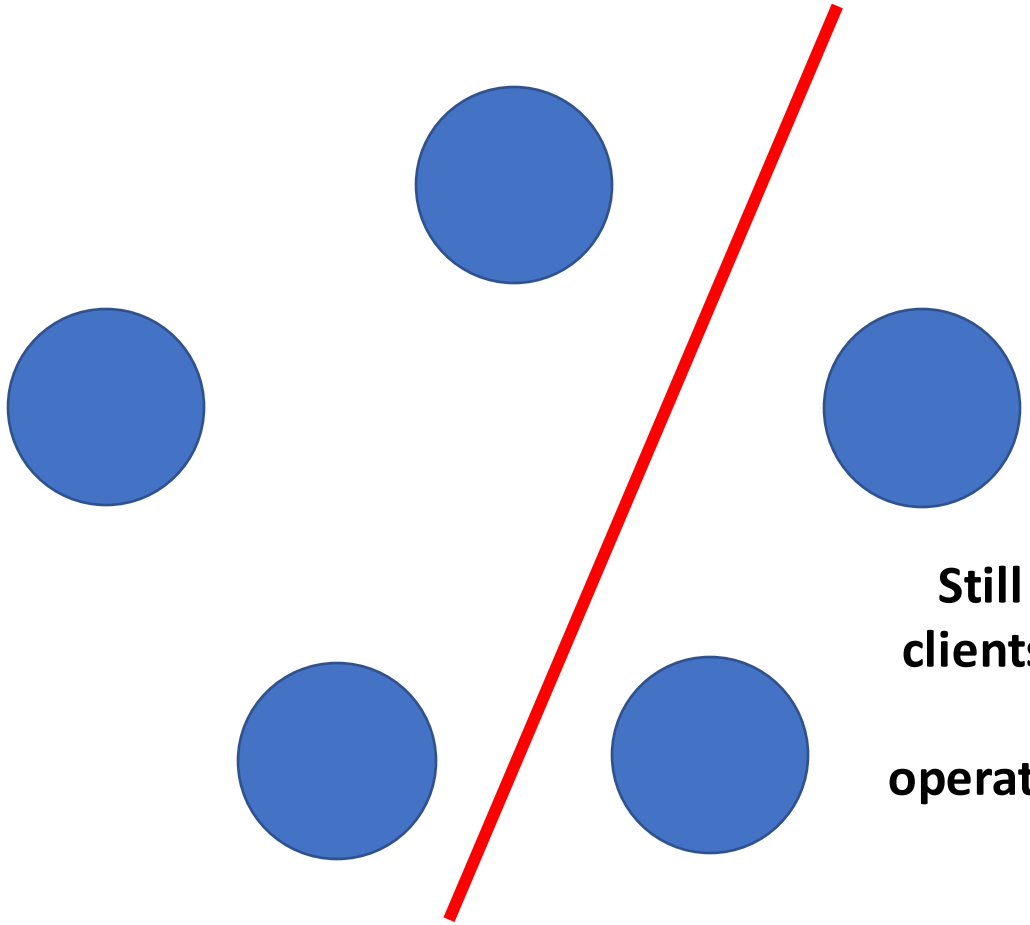
operations:

- Increased response times (high latency)
- Operations cannot happen if a majority of replicas is not reachable.

# The network is not your friend:
## Network Partitions

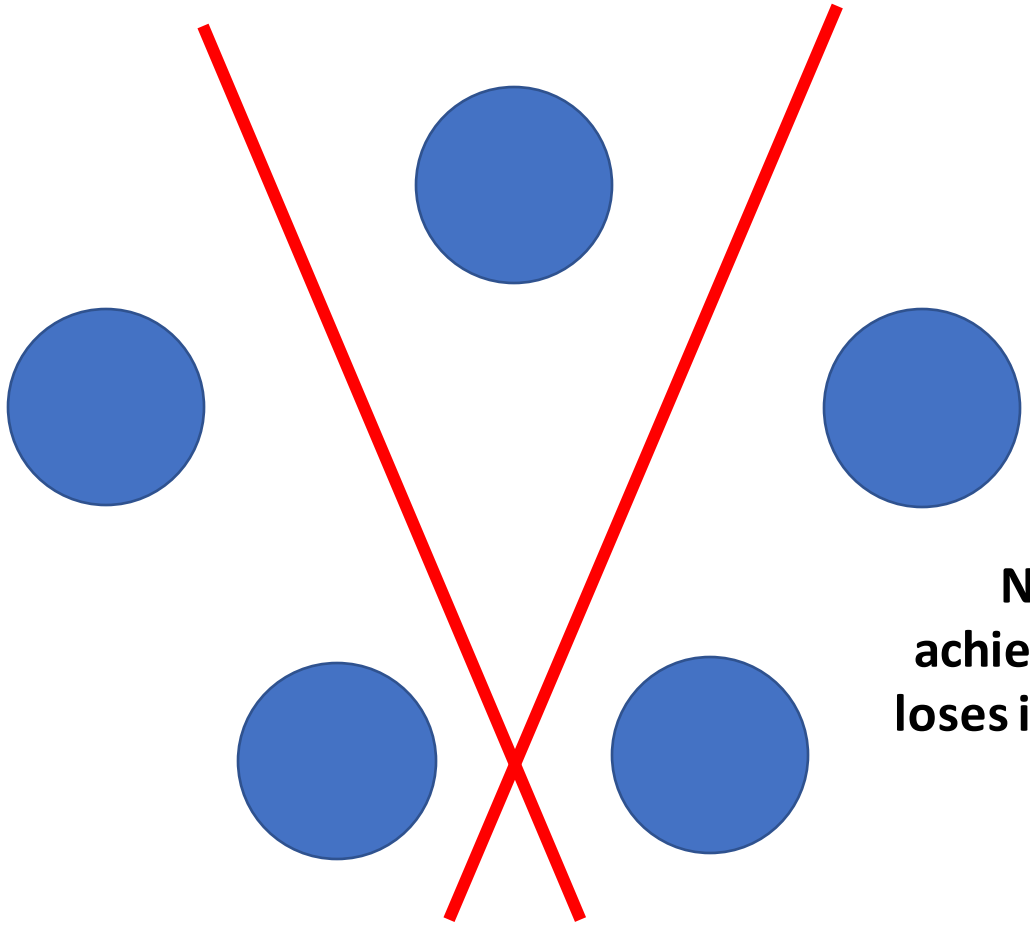


# The network is not your friend:
## Network Partitions



**Still have a majority… clients on the right side cannot complete operation that require a majority**

# The network is not your friend:
## Network Partitions



**No majority can be achieved... The system loses its availability (for every client)**

# The network is not your friend: Network Partitions

- Network partitions can happen at any time due to multiple effects:
  - Hardware failure (router/switches)
  - IP Routing instability (stabilization times of algorithms such as BGP and OSPF, see APRC course)
  - Poorly configured Firewalls.
  - Political decisions (blocking traffic from given sources, i.e. great firewall of China)
- Network partitions are more likely in scenarios where replicas are scattered across long distances (E.g., Geo-Replication).

# What can we do?

- We are looking for a solution that:
  - Is available: Allows clients to complete operations despite failures and network partitions.
  - Is strongly consistent: Creates a total order among, at least, the operations of clients that modify the state of the system.

- Any Ideas?

# What can we do?

- We are looking for a solution that:
  - Is available: Allows clients to complete operations despite failures and network partitions.
  - Is strongly consistent: Creates a total order among, at least, the operations of clients that modify the state of the system.

- Any Ideas?
- Well… such solution technically does not exist!

# CAP Theorem

- Introduced by Eric Brewer in 1999 as a principle.
- In a distributed system it is impossible to simultaneously provide:
  - Strong **C**onsistency (as in atomicity, every client will observe the effects of the most recent write).
  - **A**vailability (all operations finish eventually).
  - **P**artition-Tolerance (network anomalies that fraction the system into multiple components, where only the communication links inside each component are reliable).

# CAP Formal Proof

# CAP Formal Proof

- By contradiction, assume there is a distributed algorithm that provides C, A, P.
- Assume there is a read/write register, with initial value $v_0$, and two processes p1 and p2 that are partitioned (all messages between p1 and p2 are lost)

# CAP Formal Proof

- By contradiction, assume there is a distributed algorithm that provides C, A, P.
- Assume there is a read/write register, with initial value $v_0$, and two processes p1 and p2 that are partitioned (all messages between p1 and p2 are lost)
- Execution e1:
  - Process p1 executes write(v), to ensure A this operation concludes at some point in time t1.
  - Process p2 executes read() at time t2=t1+k. To ensure A, p2 must return a value, and due to the C property, p2 must return value v.

# CAP Formal Proof

- By contradiction, assume there is a distributed algorithm that provides C, A, P.
- Assume there is a read/write register, with initial value $v_0$, and two processes p1 and p2 that are partitioned (all messages between p1 and p2 are lost)
- Execution e1:
  - Process p1 executes write(v), to ensure A this operation concludes at some point in time t1.
  - Process p2 executes read() at time t2=t1+k. To ensure A, p2 must return a value, and due to the C property, p2 must return value v.
- Execution e2:
  - Process p1 does not execute any write.
  - Process p2 executes a read() operation at time t2=t1+k. By property A it must return some value, and by property C that value must be $v_0$.

# CAP Formal Proof

- By contradiction, assume there is a distributed algorithm that provides C, A, P.
- Assume there is a read/write register, with initial value $v_0$, and two processes p1 and p2 that are partitioned (all messages between p1 and p2 are lost)
- Execution e1:
  - Process p1 executes write(v), to ensure A this operation concludes at some point in time t1.
  - Process p2 executes read() at time t2=t1+k. To ensure A, p2 must return a value, and due to the C property, p2 must return value v.
- Execution e2:
  - Process p1 does not executes any write.
  - Process p2 executes a read() operation at time t2=t1+k. By property A it must return some value, and by property C that value must be $v_0$.
- **From the standpoint of p2, executions e1 and e2 are indistinguishable, and hence should return the same value.**

# Can we do better than eventual consistency?

- While being within the bounds set by the CAP theorem.

# Can we do better than eventual consistency?

- While being within the bounds set by the CAP theorem.

- Yes, we can: We can rely on Causal Consistency.

# Causal Consistency

- Each client always observes a system state that respects the cause -> effect relationships between write operations.
- Typically described by the combination of four fundamental properties (session guarantees):
  - Read your writes
  - Monotonic Reads
  - Monotonic Writes
  - Writes Follows Reads

# Monotonic Reads

- Subsequent reads issued by the same client should observe either the same state or an inflation of the system state (i.e., the state observed previously modified by new write operations).

- This property ensures that the system state does not "evolve into the past".
  - Assume that a client C writes 1 on object X and **then** 2 on object Y (both objects initially had a value 0).
  - If client C' observes Y = 0 and then X = 1, this does not violate Monotonic Reads.
  - If client C' then observes Y = 2 and then X = 0, this violates Monotonic Reads.

# Read your Writes

- A client must always be able to observe the effects of all previous writes issued by itself (for which it got a reply from the system).

- Observing the effects does not imply that the same value must be returned. It implies that the returned value is at least as up-to-date as the value written by the client itself.
  - Think ABD

# Monotonic Writes

- The effects of multiple write operations issued by a given client, must be observed respecting that order by every other client.

- E.g.:
  - Assume that a client C writes 1 on object X and **then** 2 on object Y (both objects initially had a value 0).
  - If a client C' observes Y = 0 and then X = 1, this does not violate Monotonic Writes.
  - If another client C'' then observes Y = 2 and then X = 0, this violates Monotonic Writes.

# Writes Follows Reads

- If a client observes the effects of a write operation in its session (through a read operation), then any subsequent write operation issued by that client must be ordered after the previously observed write.

- This is also known as the **transitivity** rule.
  - Previous properties only refer to a single client session.
  - This property establishes a connection between sessions of different clients.

# Causality

- Causality by itself does not guarantee that the state of multiple replicas converges to a single value (even if at some point in time, no more write operations occur).

- Divergence can last forever without breaking any causality property (**provided clients are always connected to the same replica**).

# Causal+ Consistency Model

- It’s a more recently (10+ years) formalized consistency model that simply combines the properties of:
  - Causal consistency.
  - Eventual consistency.

- Ensures that the state of replicas eventually converges to a single state, if no more write operations happen.

# Causal+ and CAP

- Causal+ consistency model is the strongest of the weak consistency models that is compatible with the CAP theorem.

- Caveats:
  - Clients must stick with a replica (aka *sticky clients*).
  - If a replica fails (permanently), then those clients might be forced to restart their sessions (which is a nice way to say sacrifice causal consistency guarantees).

# Implementing Replication Protocols offering Causal+ Consistency

- (Intuition) There is a partial order among write operations defined by a combination of:
  - A) Order of write operations executed by a client:

    W(x,1) W(y,2) => W(x,1) $\prec$ W(y,2)
  - B) Effects of write operations observed by a client (in their read operations) before they issue a write.

    R(x,2) W(y,1) => W(x,2) $\prec$ W(y,1)
- If $W_1 \prec W_2$ : We say $W_1$ is a causal dependency of $W_2$

# Implementing Replication Protocols offering Causal+ Consistency

**P1 : W(x)1 W(x)3**

**P2 : R(x)1 W(x)2**

**P3 : R(x)1 R(x)3 R(x)2**

**P4 : R(x)1 R(x)2 R(x)3**

- W(x)1 $\prec$ W(x)2, because P2 reads R(x)1 and then Writes W(x)2
- W(x)1 $\prec$ W(x)3, because P1 writes W8
- R(x)1 has to be observed *before* R(x)2 or R(x)3
- R(x)2 & R(x)2 do not imply a causal relationship between W(x)2 & W(x)3, so can be observed in different orders

# Implementing Replication Protocols offering Causal+ Consistency

- Track causal dependencies of operations (potentially associated to data objects written by each write operation).
- Ensure, **for each client**, that if a value generated by $W_a$ is observed, no value can be observed that has been overwritten by any $W_d$ such that $W_d \prec W_a$.
- i.e. Only allow the effects of $W_a$ to become visible after the effects of any operations that are overwritten by any $W_d$ becoming impossible to observe.

# Implementing Replication Protocols offering Causal+ Consistency

- This requires clients to memorize all write operations whose effects have been observed in their session
  - This includes not only the write operations directly observed but all of their causal dependencies): The **client causal history**.
- When performing a write operation, the causal history of the client is sent alongside the write and recorded in the data storage system.
  - Similar to Dynamo
- Dependencies of operations might grow quite significantly.

# Famous Systems providing Causal+ Consistency

To appear in *Proceedings of the 23rd ACM Symposium on Operating Systems Principles (SOSP'11)*

**Don't Settle for Eventual:**
**Scalable Causal Consistency for Wide-Area Storage with COPS**

Wyatt Lloyd*, Michael J. Freedman*, Michael Kaminsky†, and David G. Andersen‡
*Princeton University, †Intel Labs, ‡Carnegie Mellon University

- Coined the term "Causal+ Consistency" (but did not invent it).
- Tracks dependencies of operations (using a graph).
- Sometimes operations must be retried.
- Supports special read-only transactions.

# Famous Systems providing Causal+ Consistency

**ChainReaction: a Causal+ Consistent Datastore based on Chain Replication**


Sérgio Almeida
INESC-ID, Instituto Superior Técnico, U. Técnica de Lisboa
sergiogarrau@gsd.inesc-id.pt

João Leitão
CITI / DI-FCT-Universidade Nova de Lisboa *and* INESC-ID, IST, UTL
jc.leitao@fct.unl.pt

Luís Rodrigues
INESC-ID, Instituto Superior Técnico, U. Técnica de Lisboa
ler@ist.utl.pt


- Delays write operations to minimize metadata overhead (i.e., dependency tracking).
- Operations never need to be retried.
- Also supports special read-only transactions.

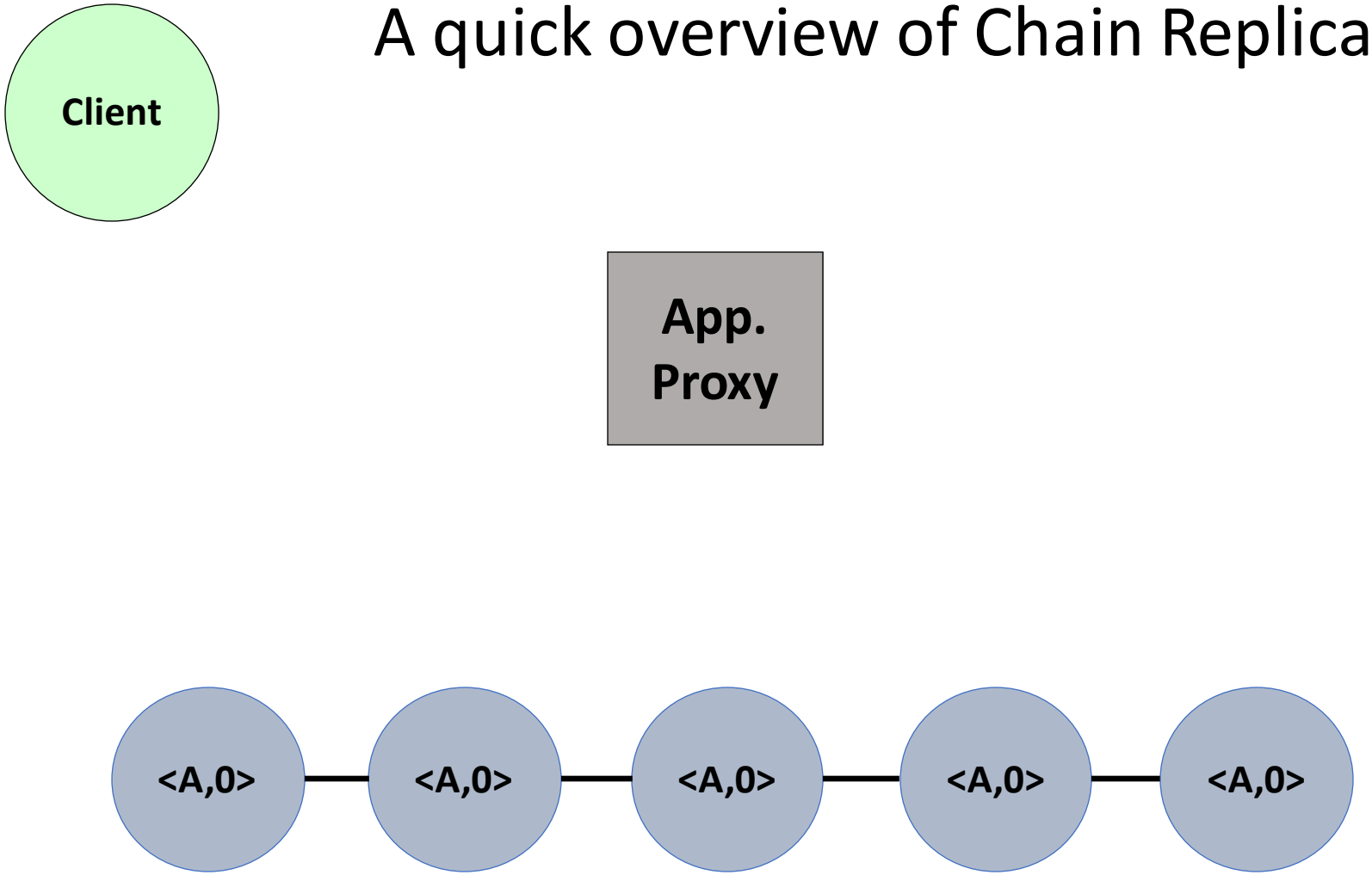
# Preliminaries:

- ChainReaction takes advantage of a specialized variant of **Chain Replication**.

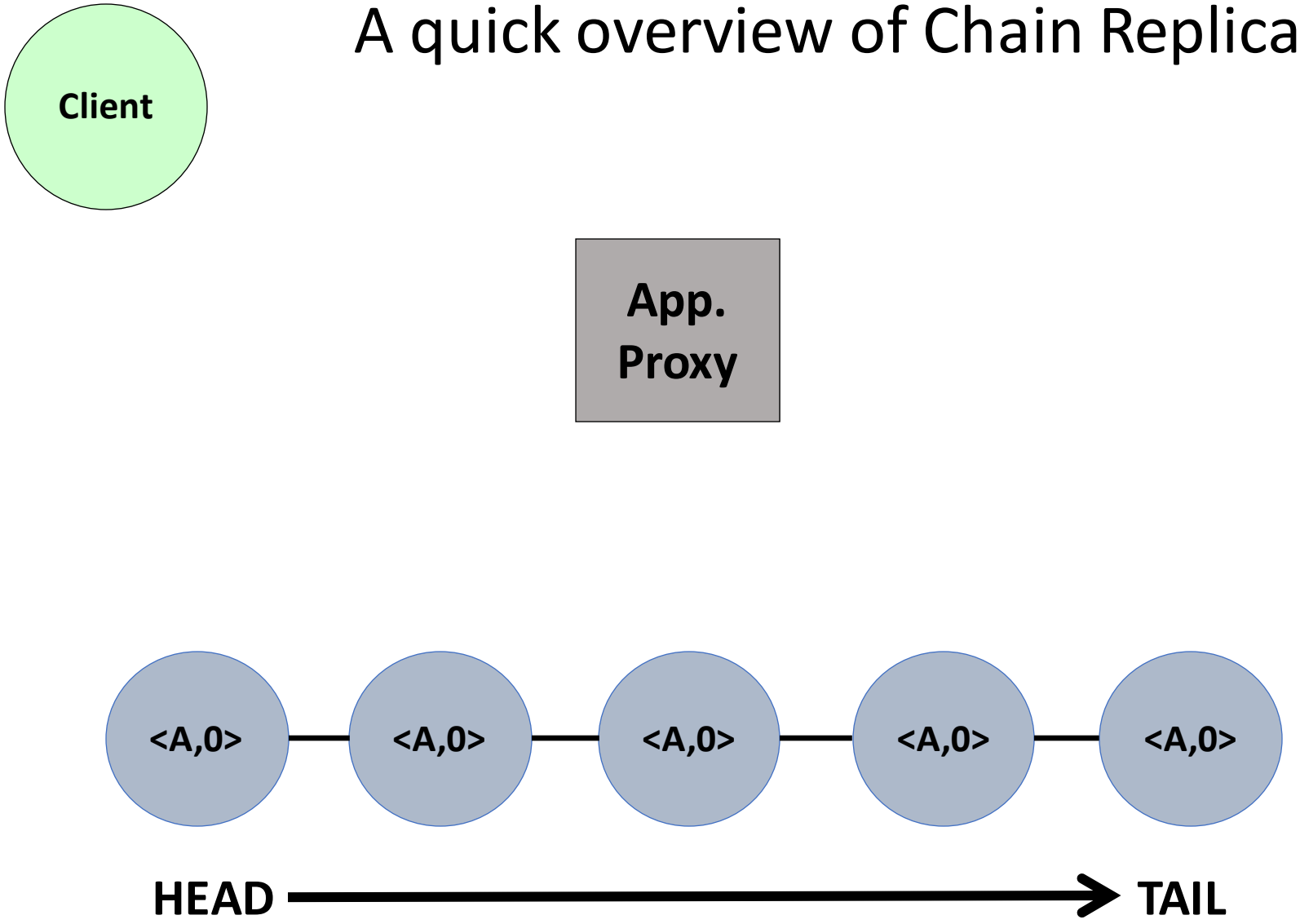
- Chain Replication is a simple replication strategy that provides linearizability (per chain), originally designed for a synchronous system under the crash-fault model.

- Nodes are organized in a chain that is maintained by an external membership service (e.g., Zookeeper).

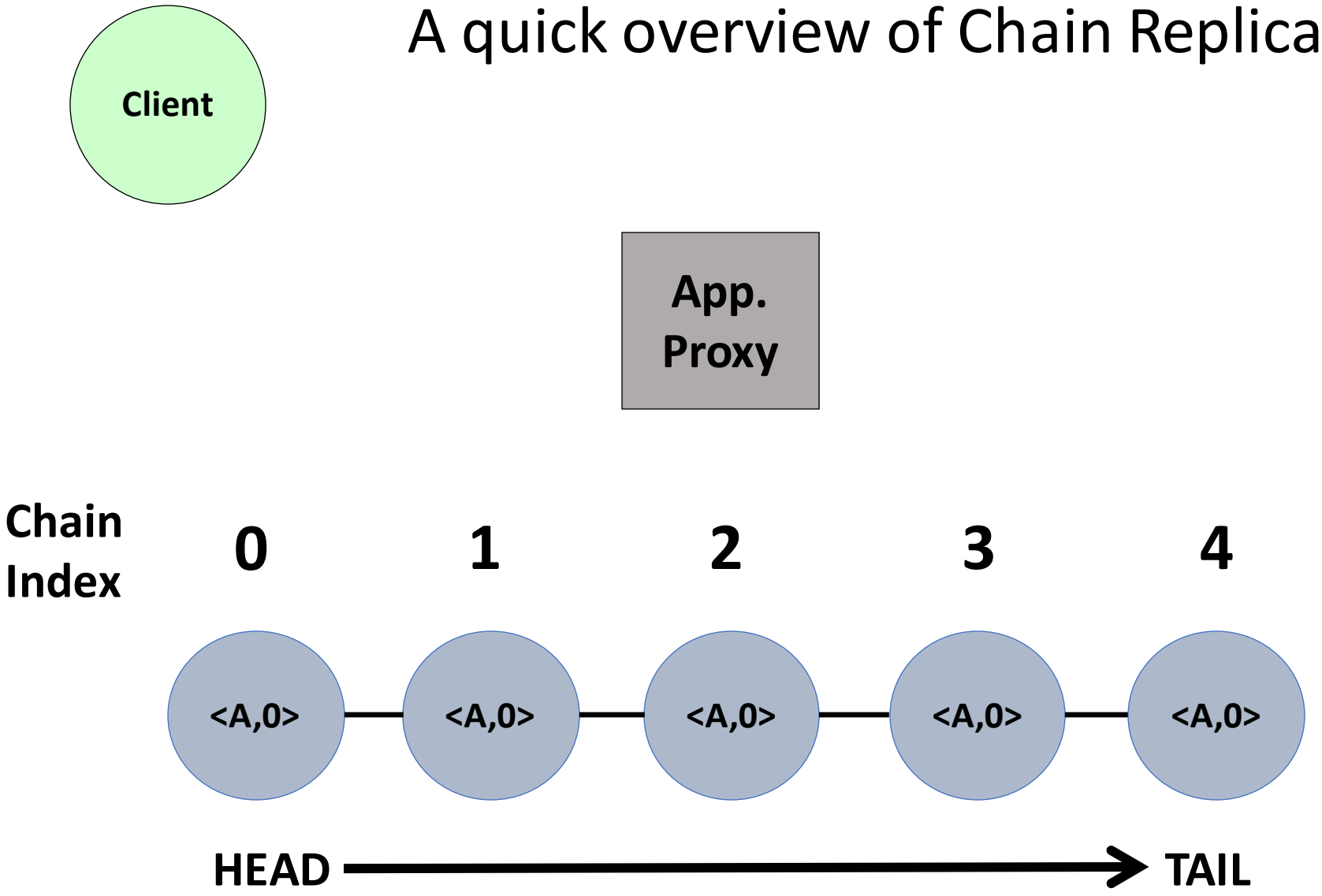
# A quick overview of Chain Replication



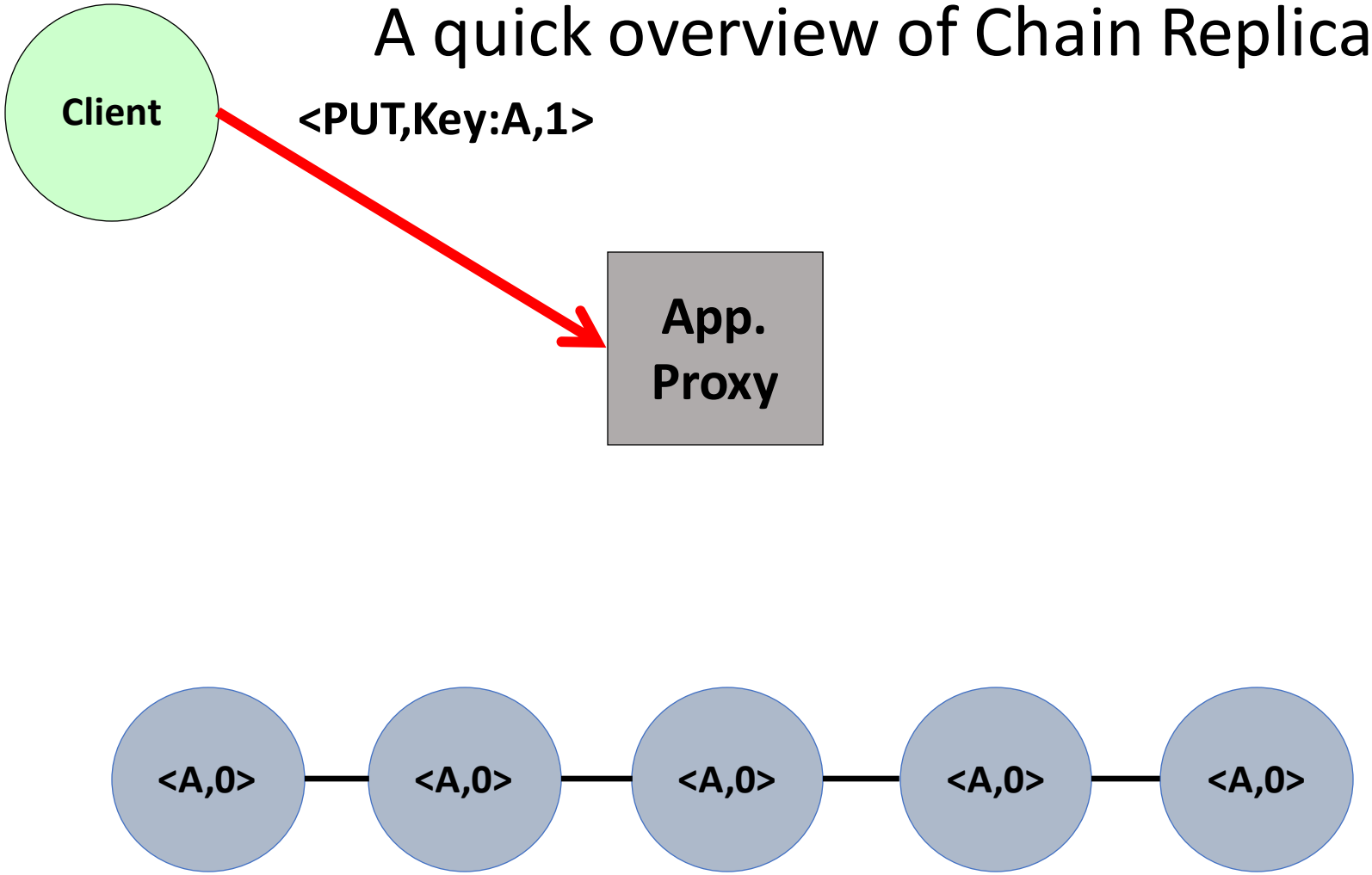
# A quick overview of Chain Replication



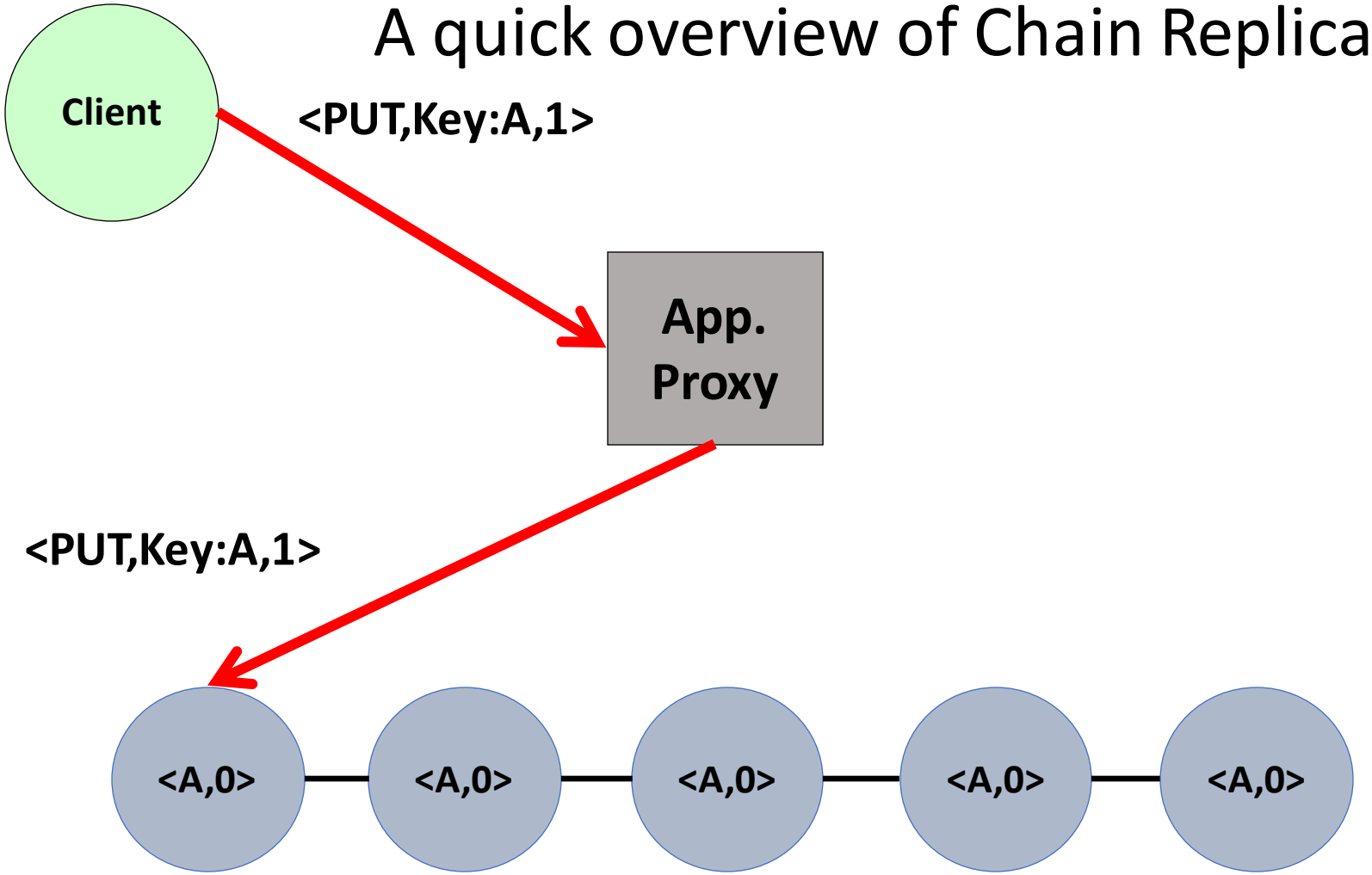
# A quick overview of Chain Replication



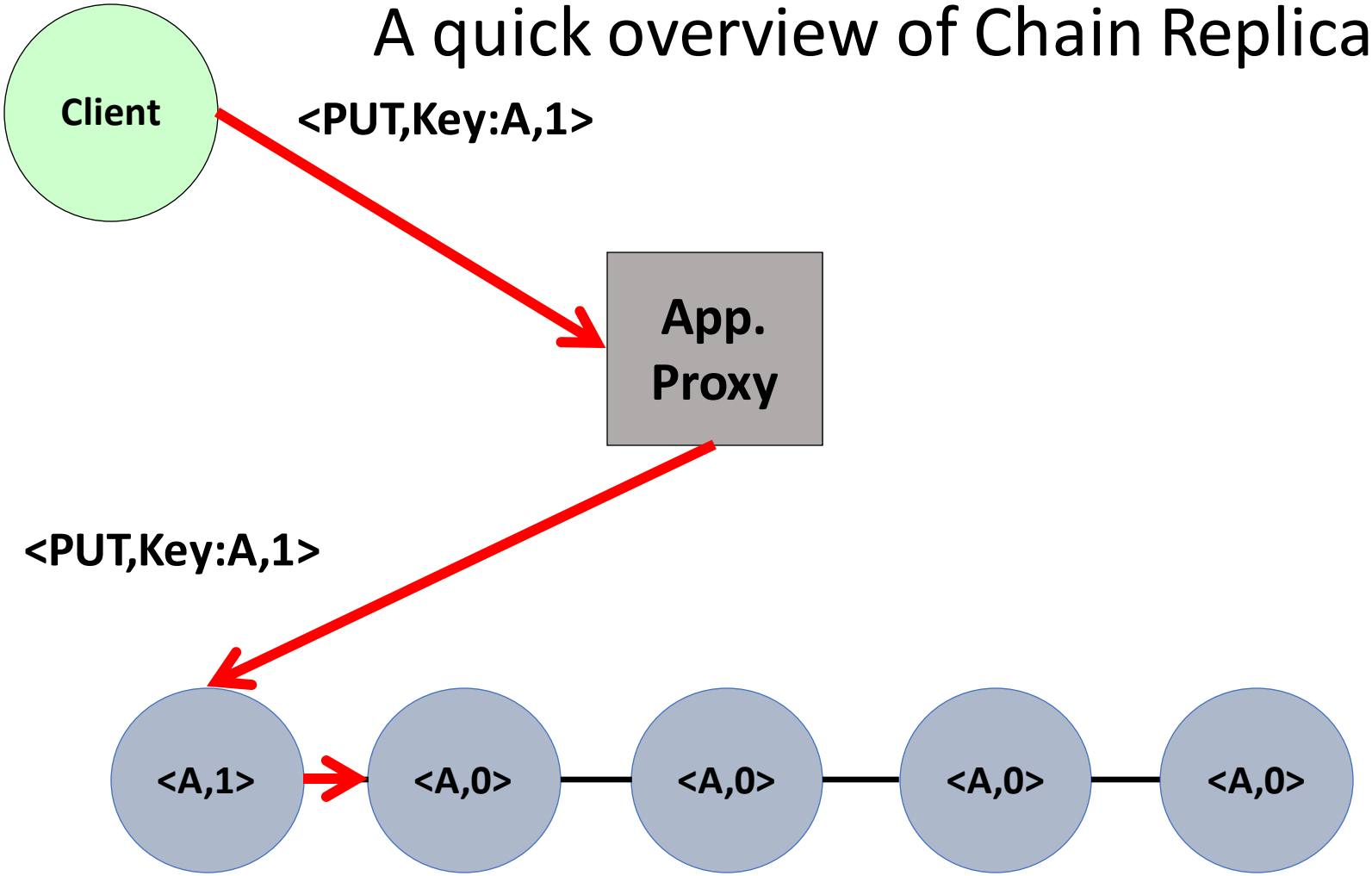
# A quick overview of Chain Replication



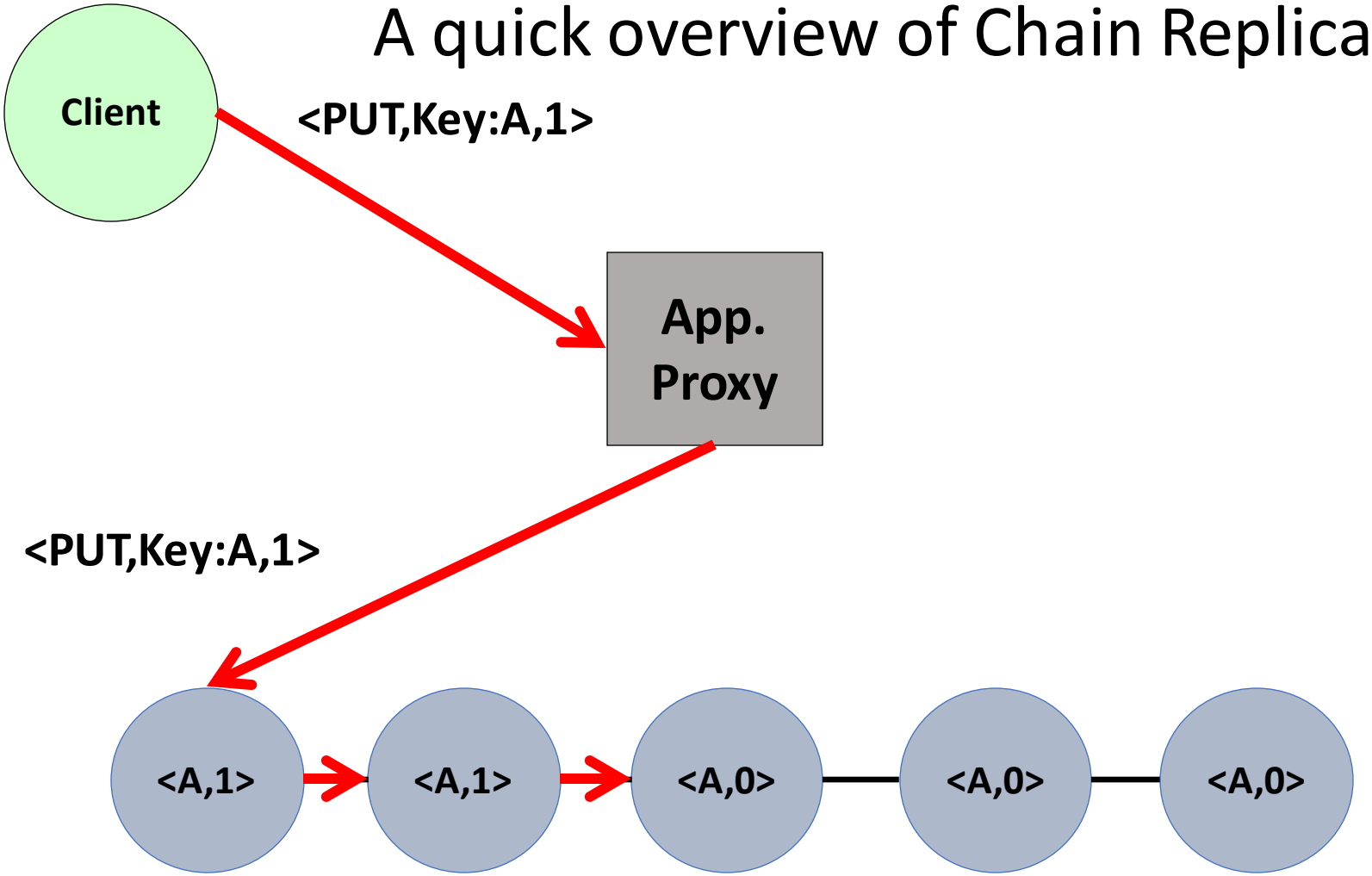
# A quick overview of Chain Replication



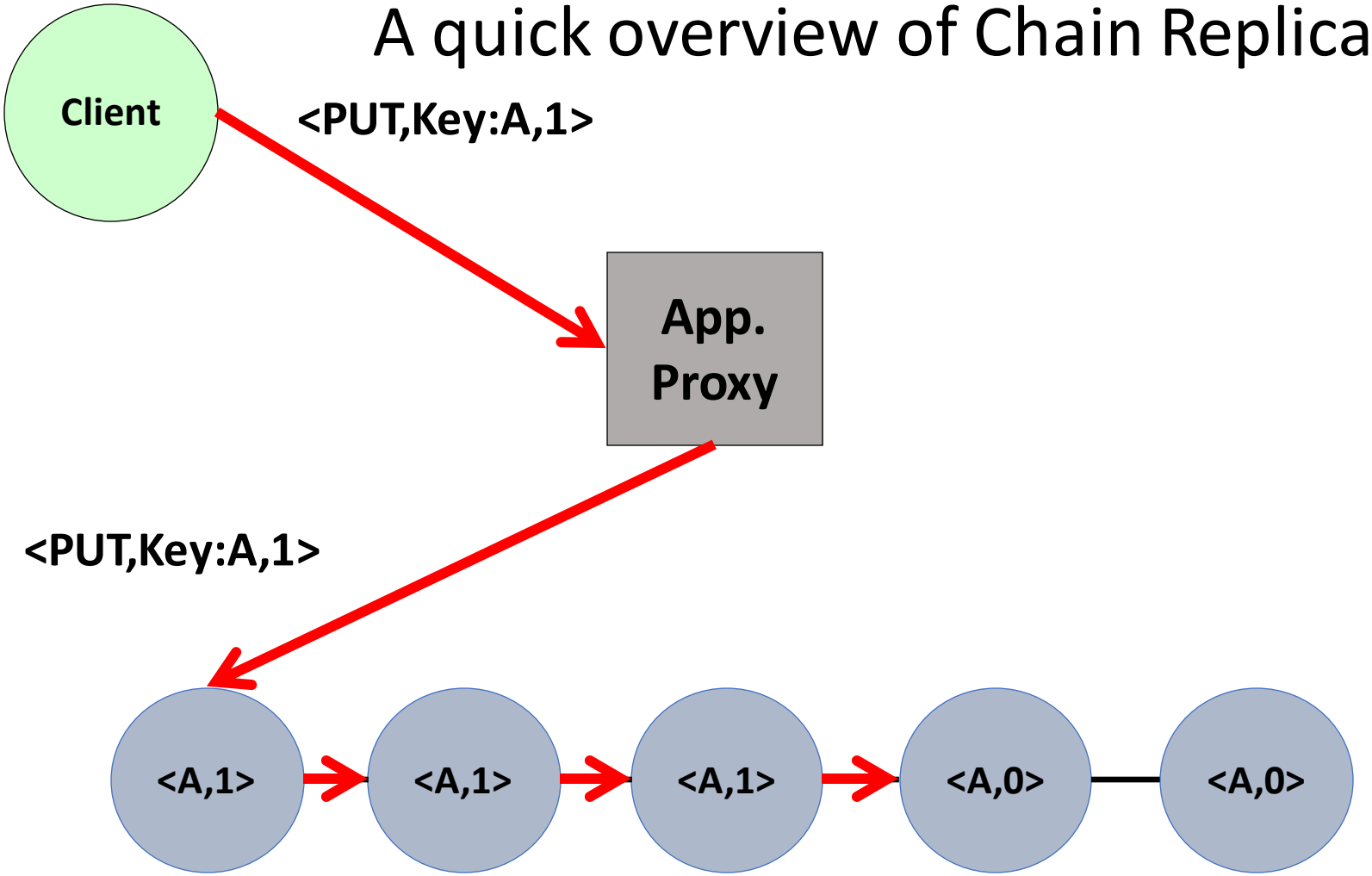
# A quick overview of Chain Replication



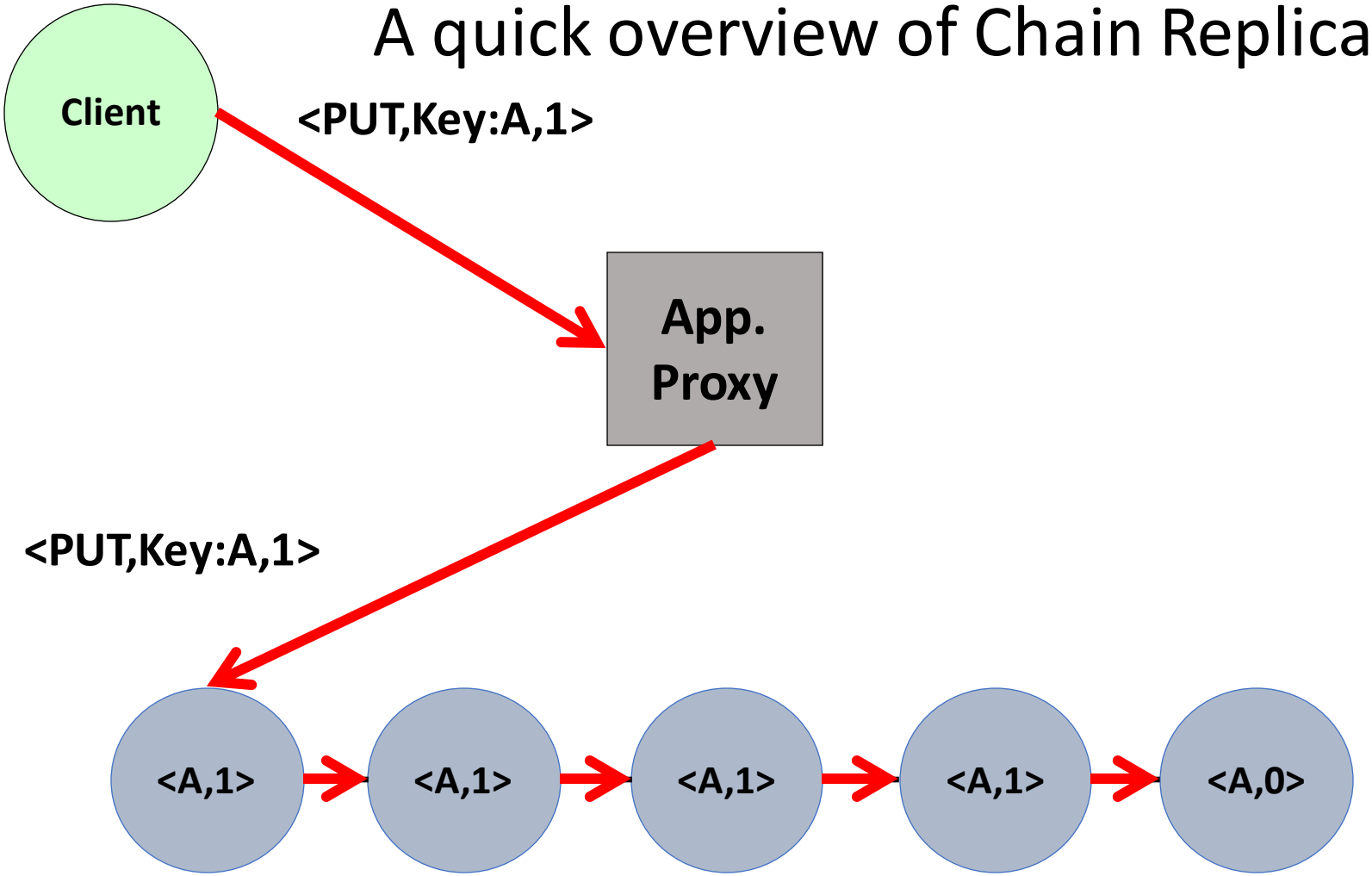
# A quick overview of Chain Replication



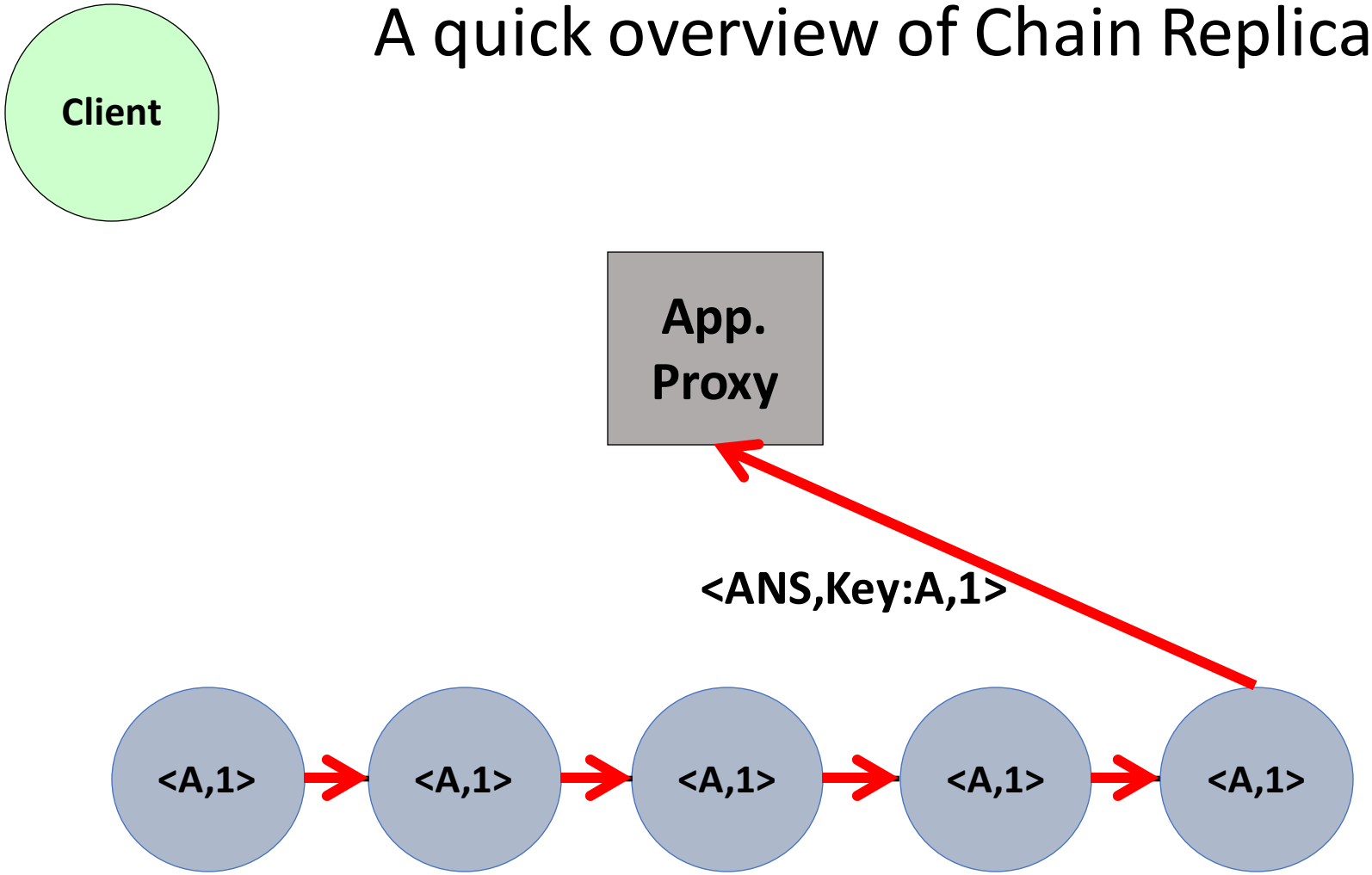
# A quick overview of Chain Replication



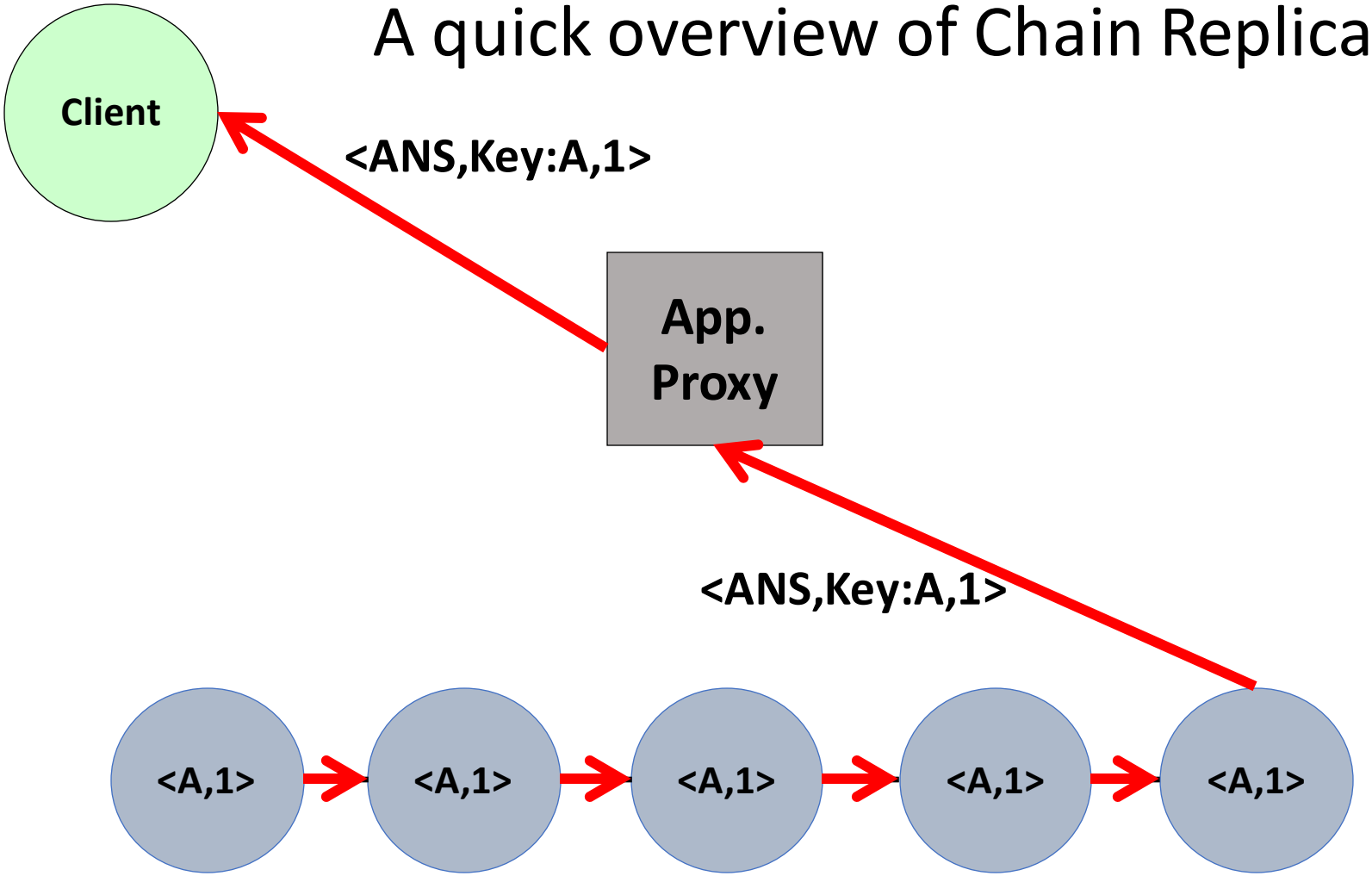
# A quick overview of Chain Replication



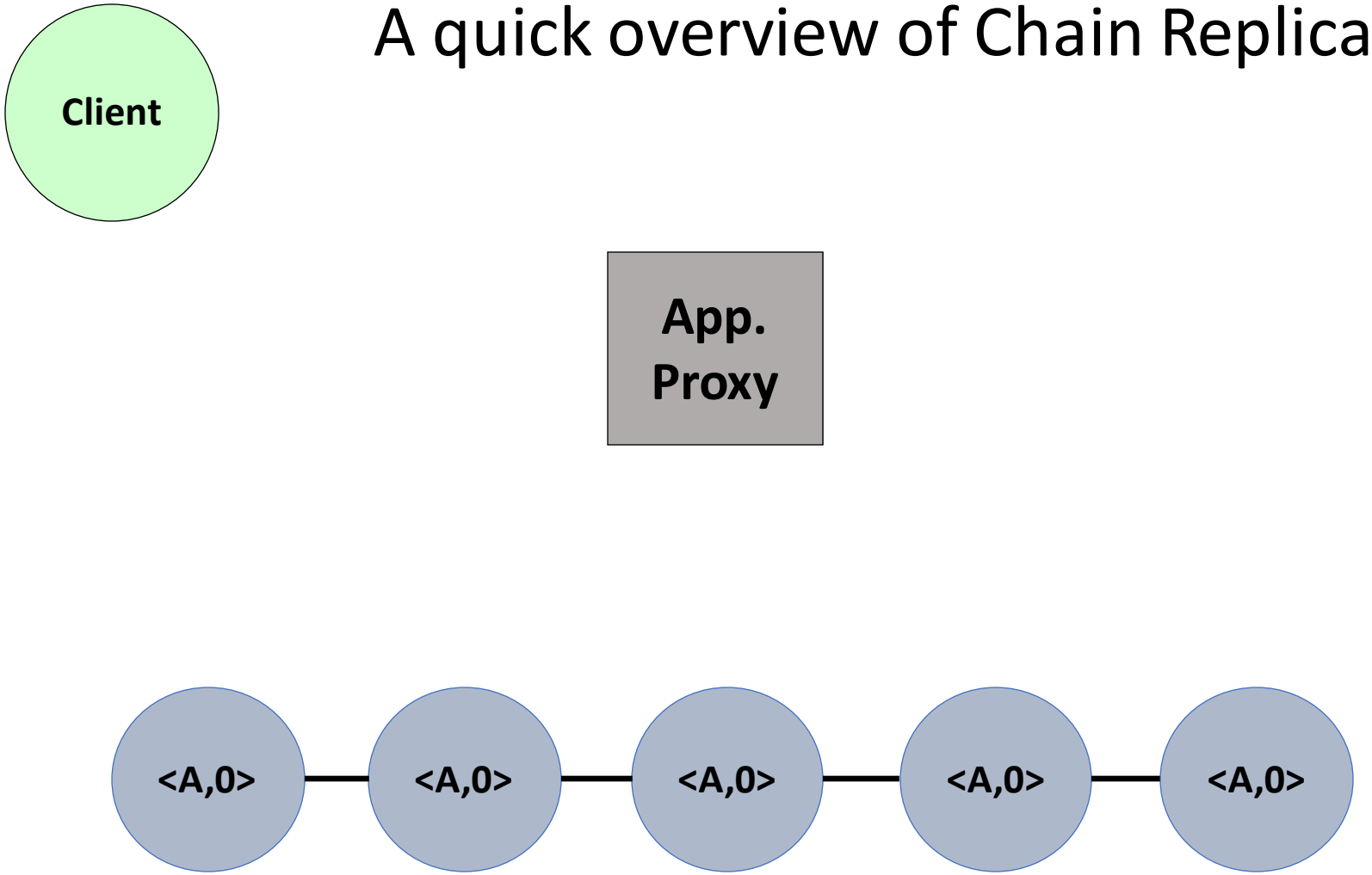
# A quick overview of Chain Replication



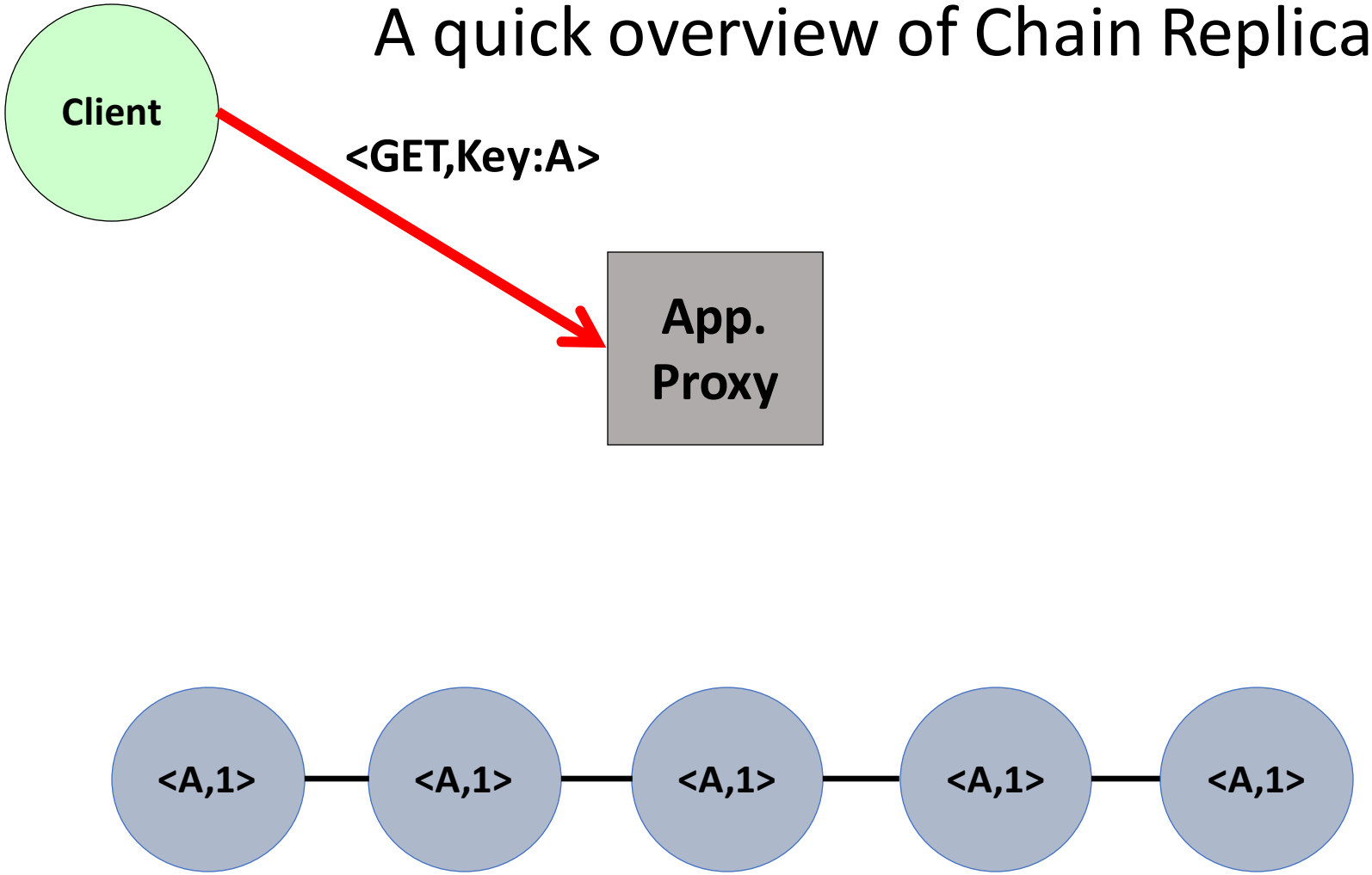
# A quick overview of Chain Replication



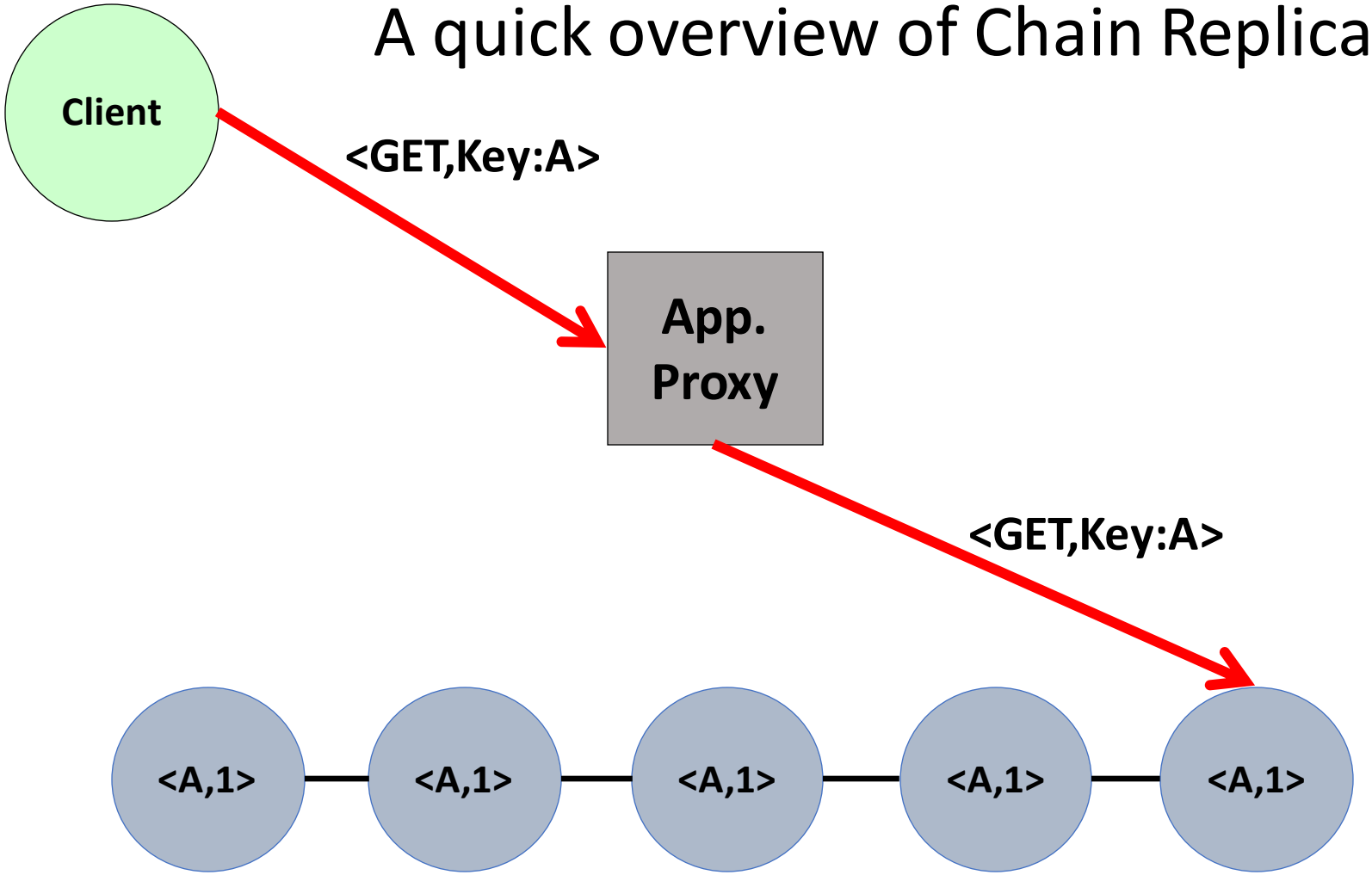
# A quick overview of Chain Replication



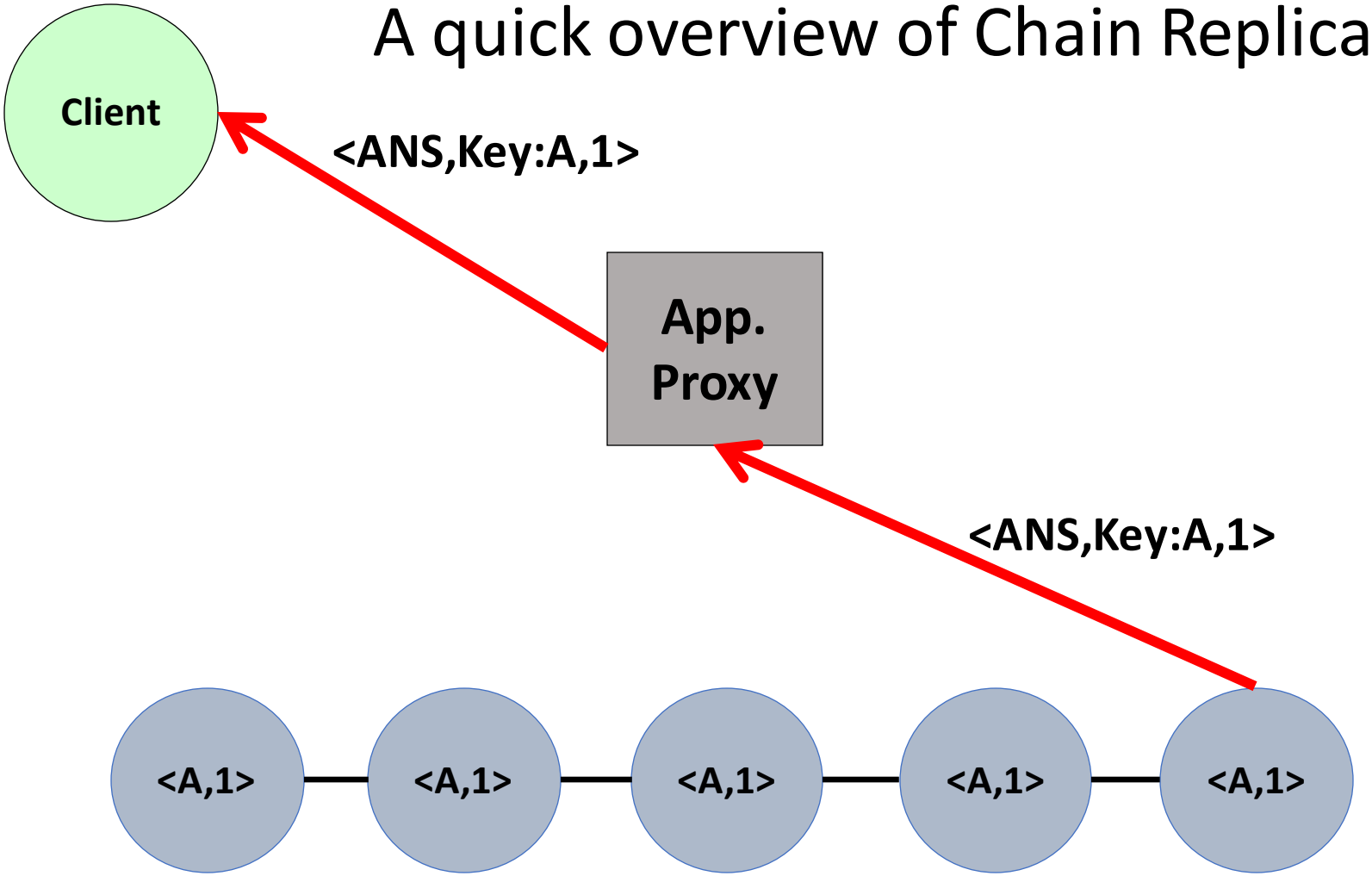
# A quick overview of Chain Replication



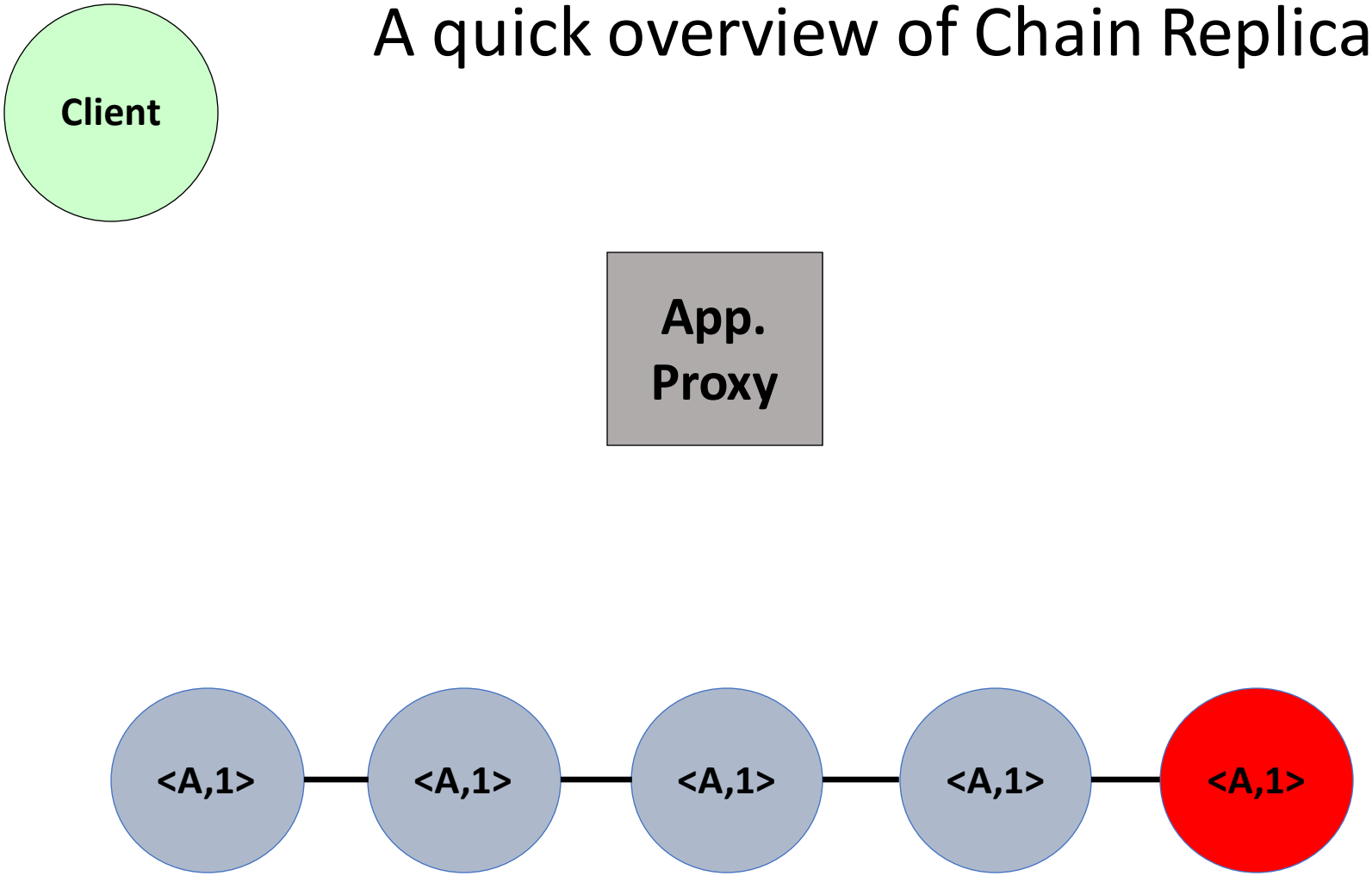
# A quick overview of Chain Replication



# A quick overview of Chain Replication



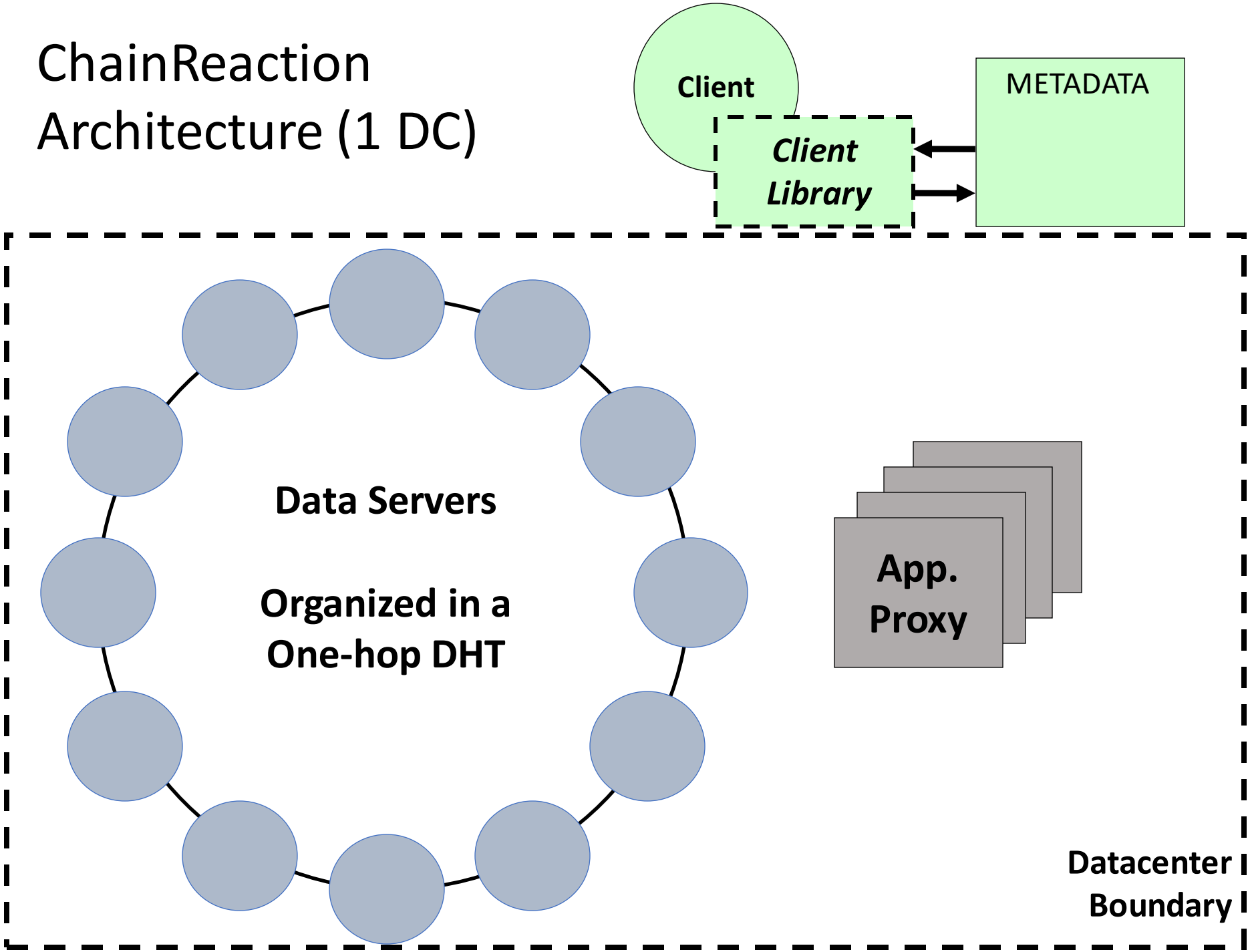
# A quick overview of Chain Replication



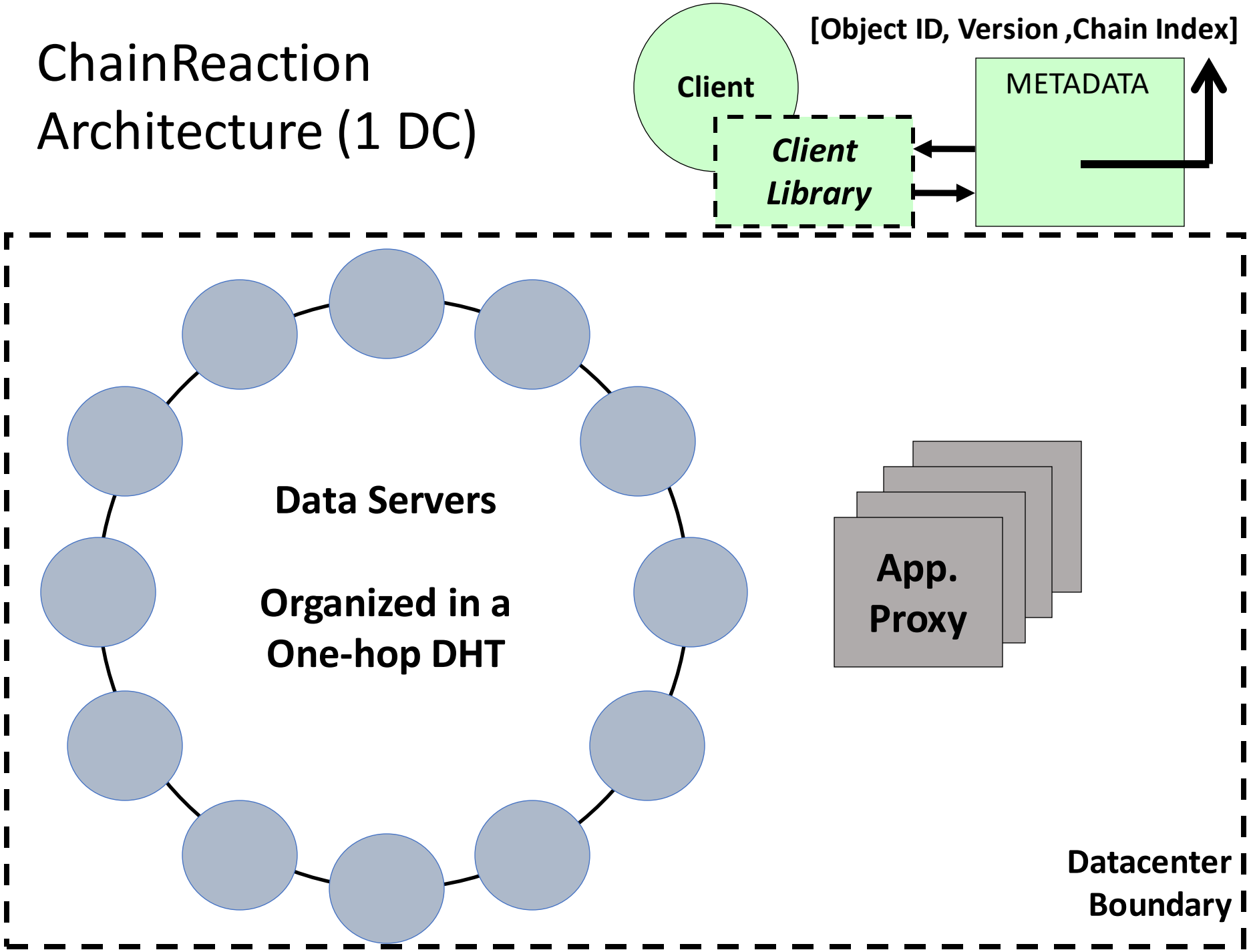
# ChainReaction

- A key-value distributed datastore
- Provides causal+ consistency (both within a single datacenter and across multiple datacenters)
- Takes advantage of a specialized version of Chain Replication:
  - Removes the tail-bottleneck by distributing reads.
  - Relaxes the replication-level associated with write operations.
  - Relaxes the consistency offered by chain replication.
- Offers high performance (optimized for reads).

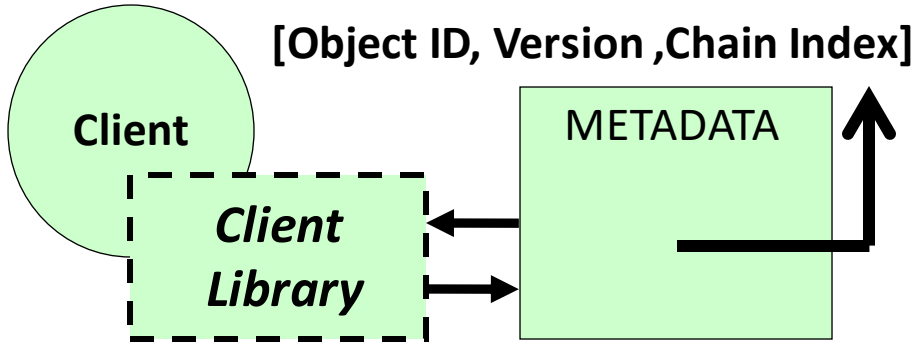
# ChainReaction Architecture (1 DC)



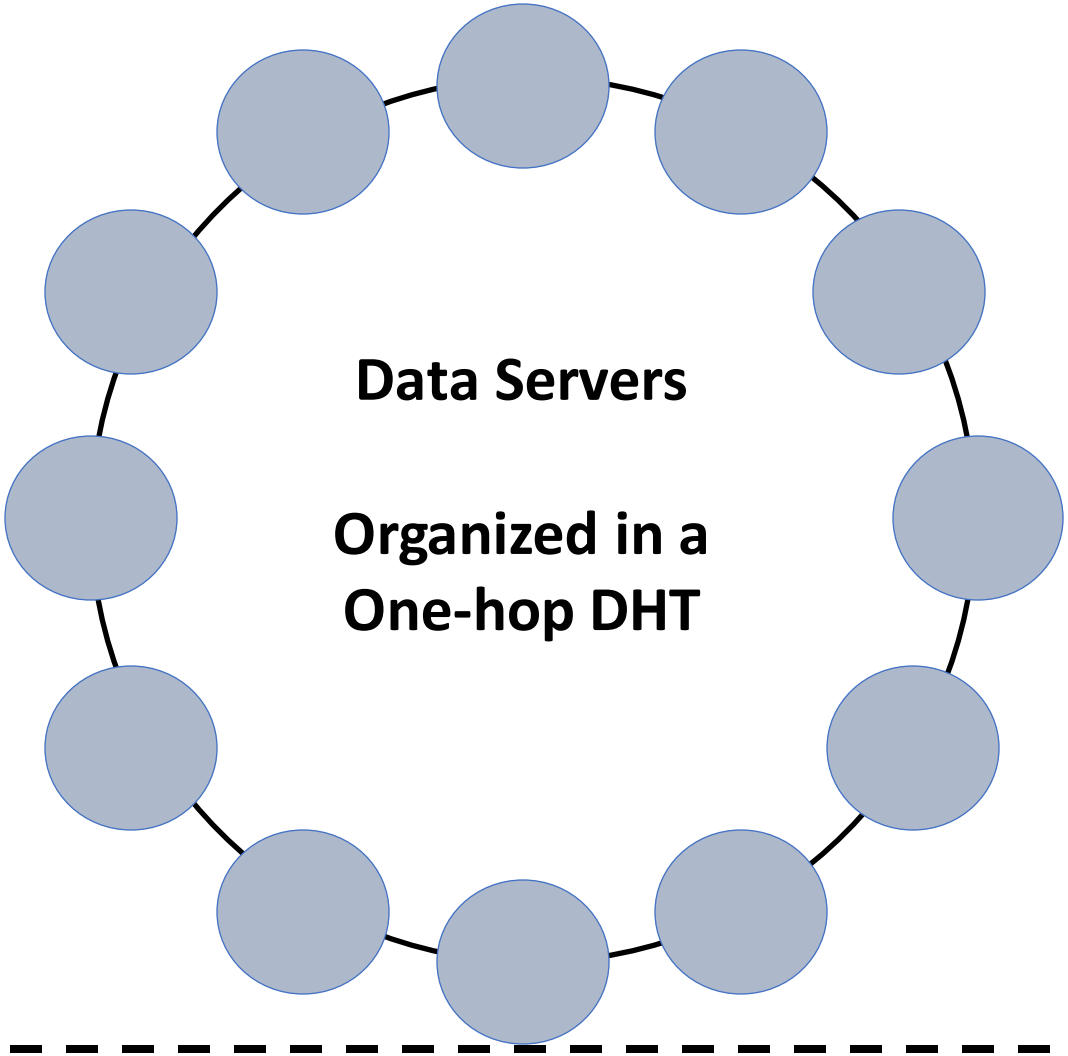
# ChainReaction Architecture (1 DC)

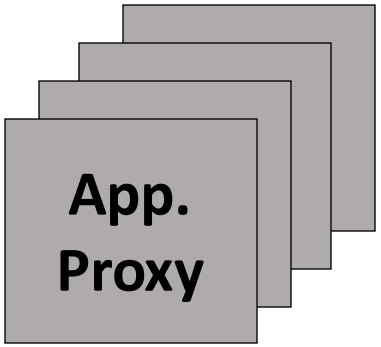


# ChainReaction Architecture (1 DC)



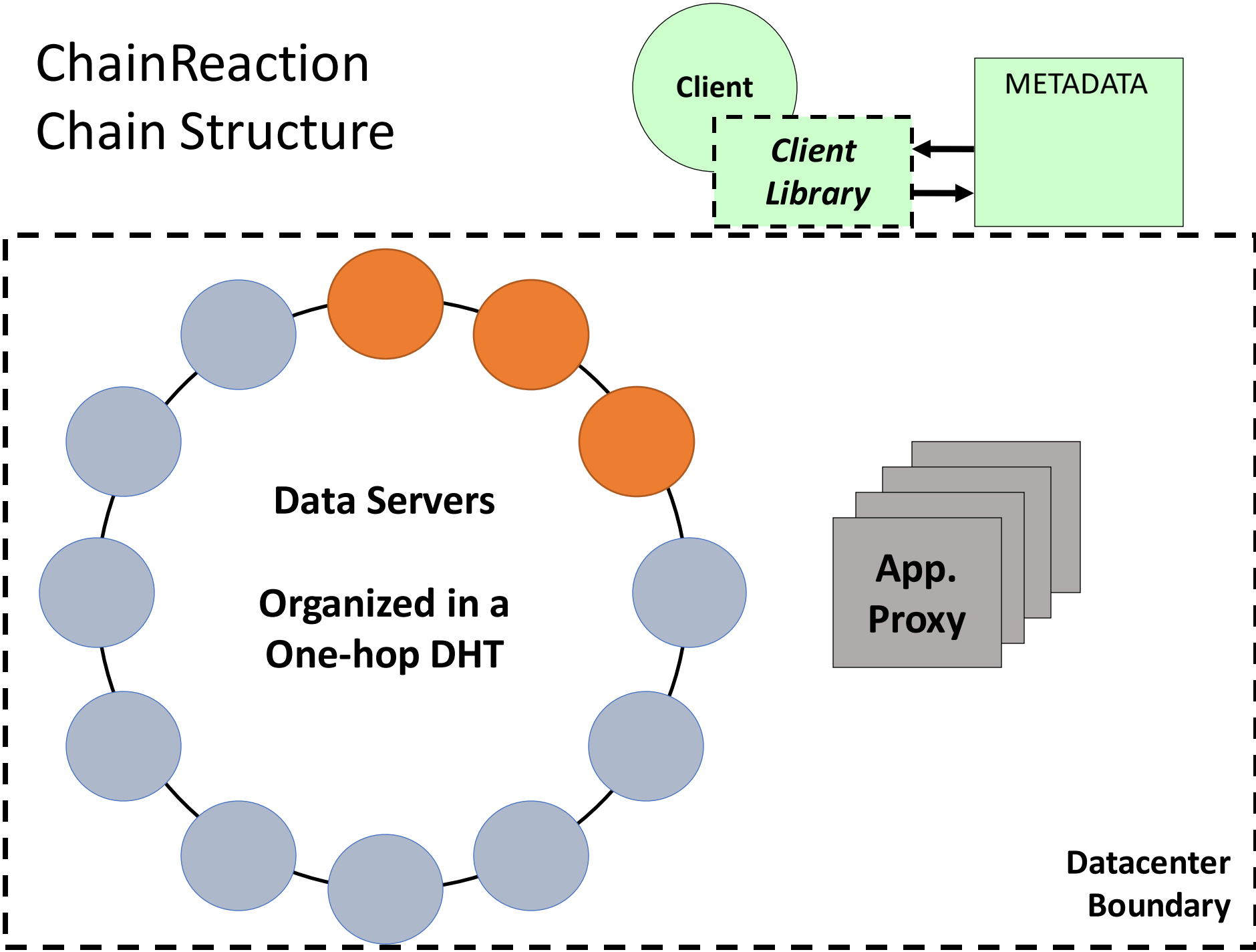

Implicitly this METADATA encodes the client causal history.



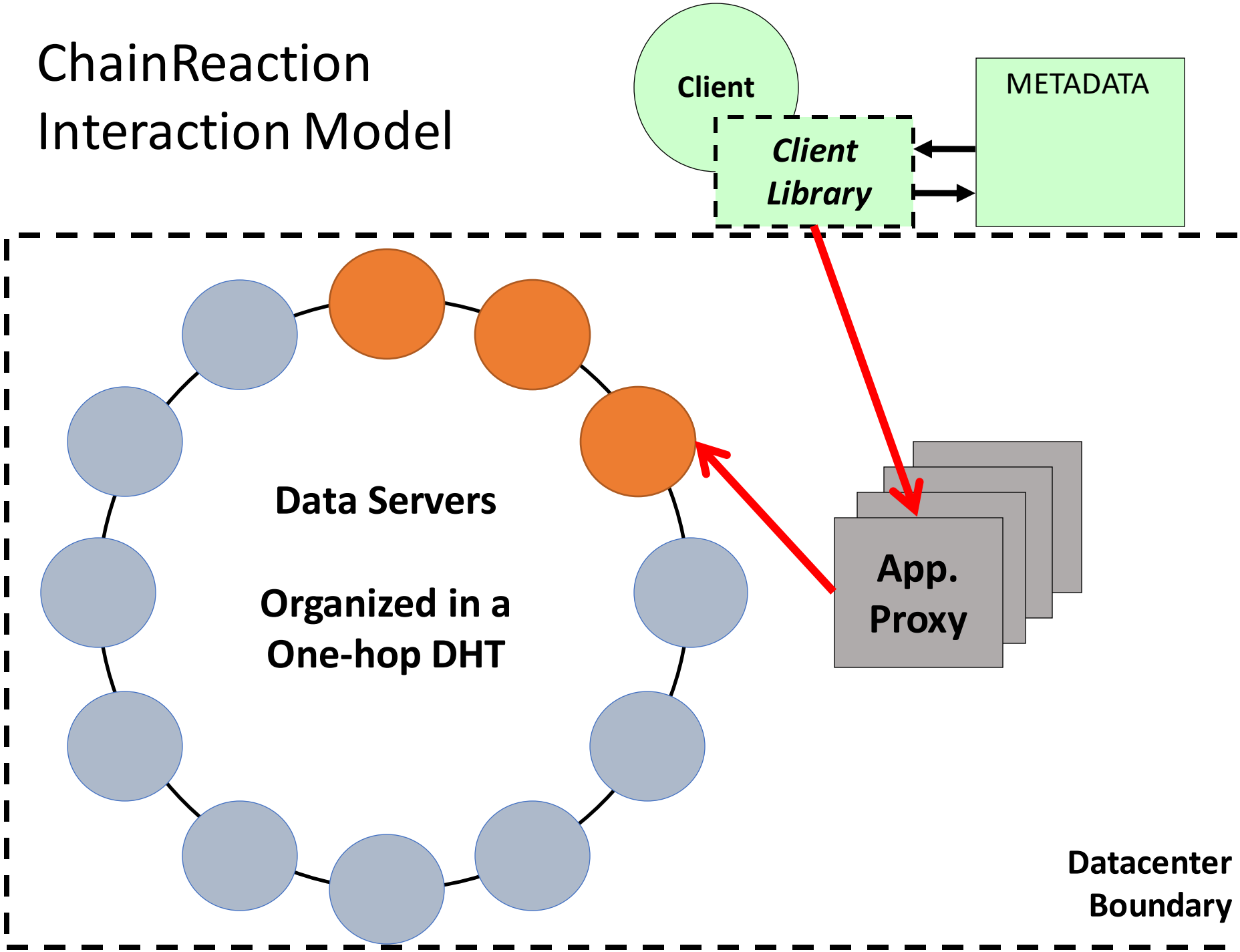




Datacenter Boundary

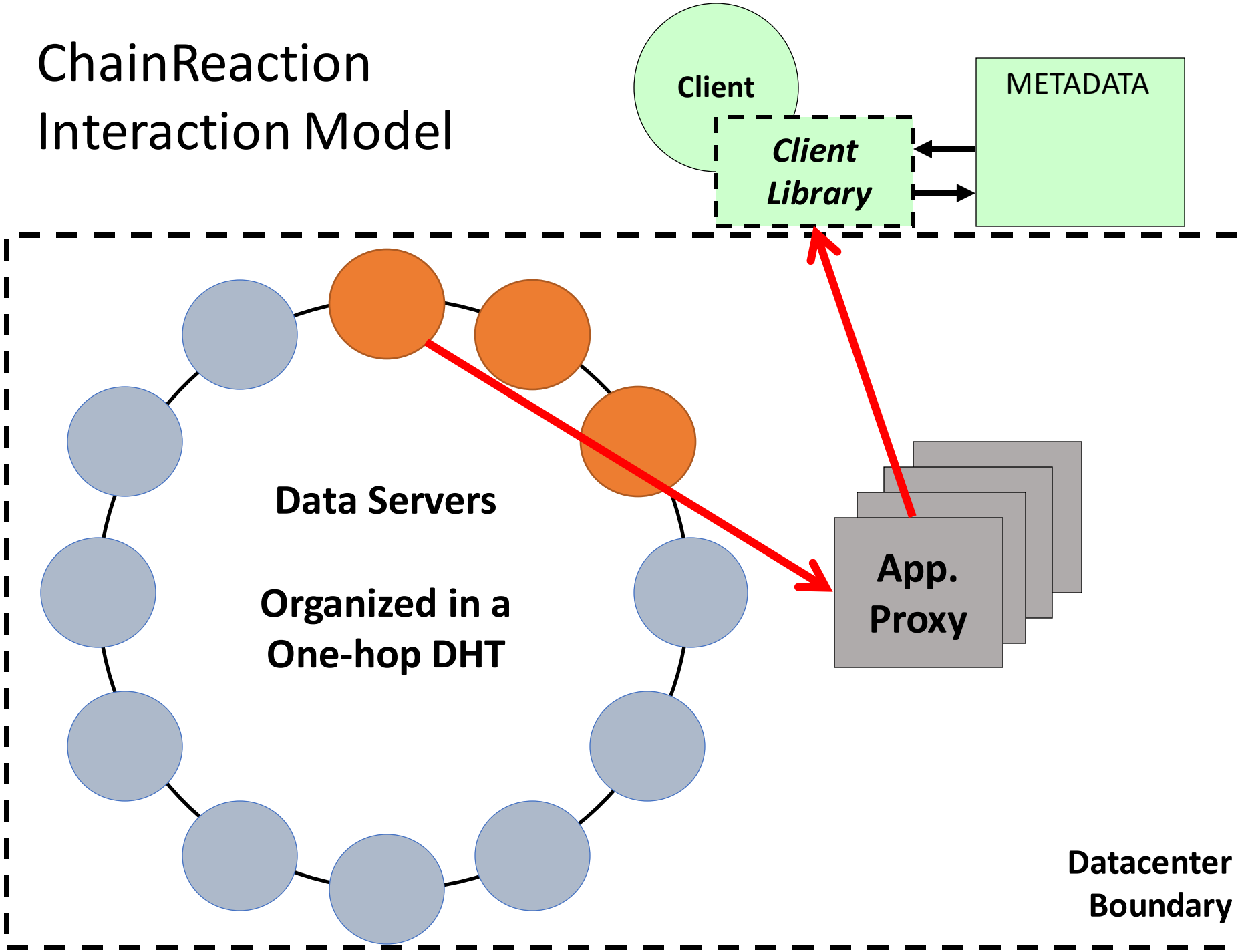
# ChainReaction
# Chain Structure



# ChainReaction Interaction Model



# ChainReaction Interaction Model



# ChainReaction: Replication Mechanism

Relies on a specialized version of Chain Replication where:

- Allow writes to return before reaching the tail.
- Support reads on all nodes of the chain.
- Trades write efficiency for metadata efficiency.

# ChainReaction: Replication Mechanism

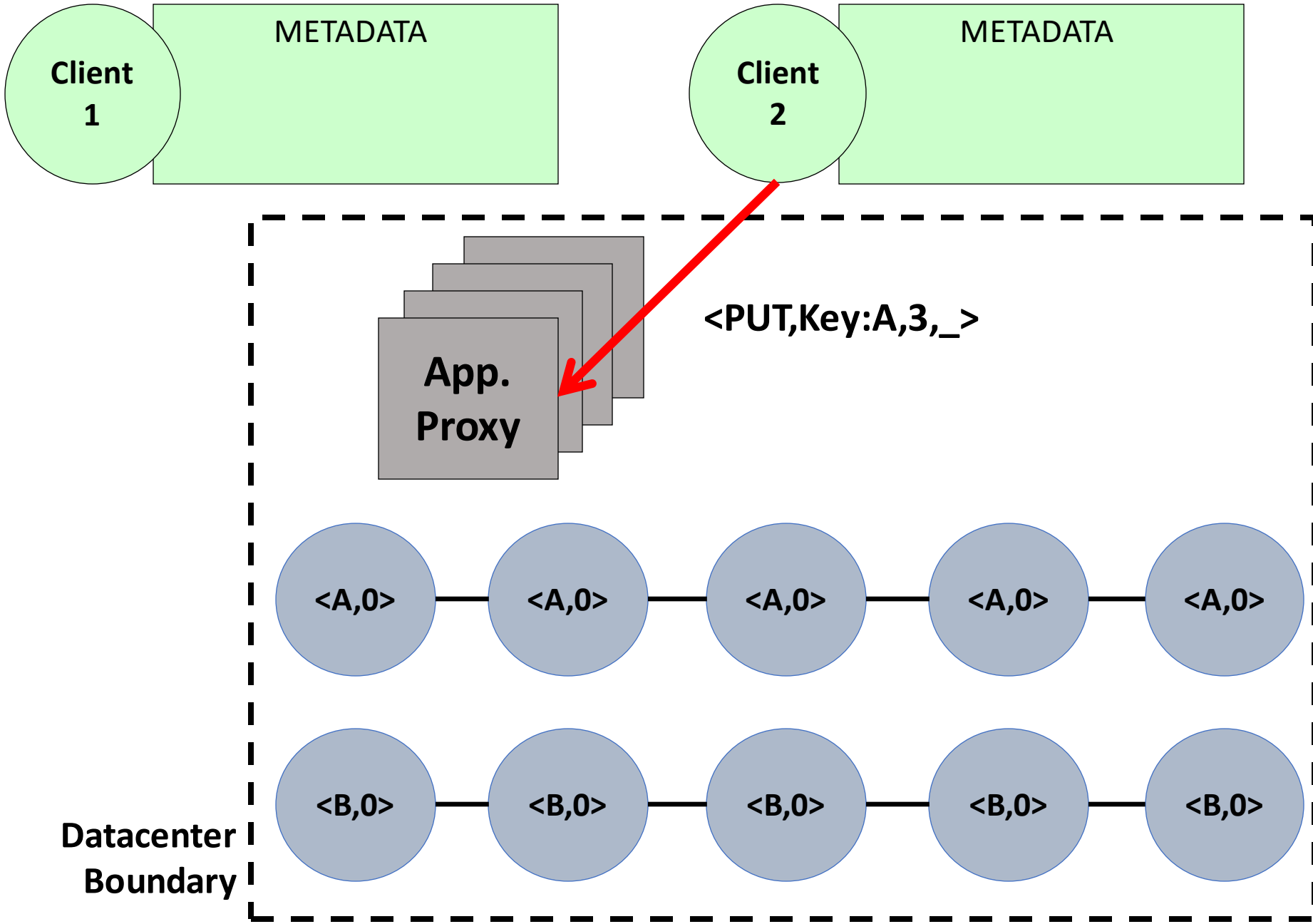
Relies on a specialized version of Chain Replication where:

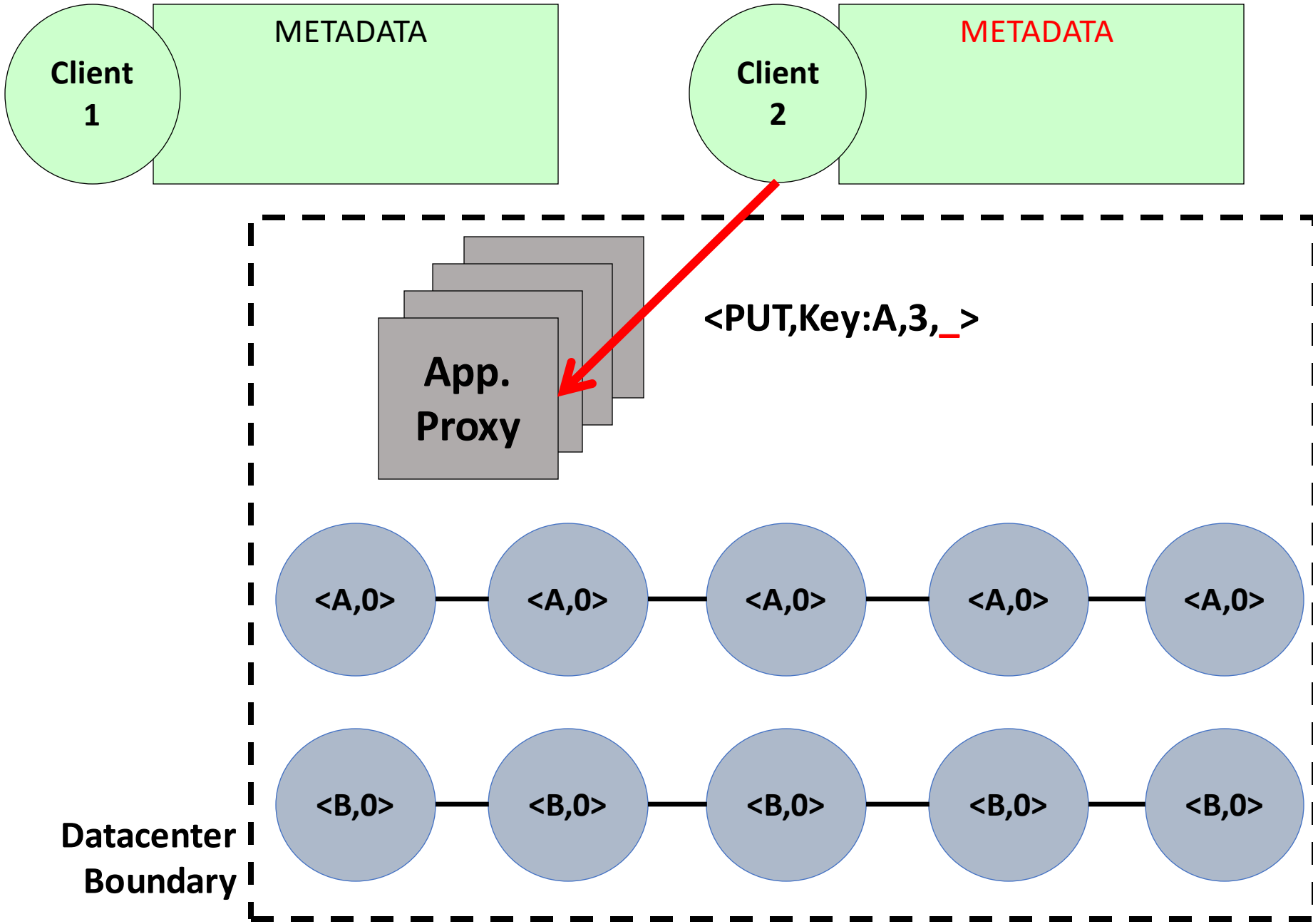
- Allow writes to return before reaching the tail.
- Support reads on all nodes of the chain.
- Trades write efficiency for metadata efficiency.

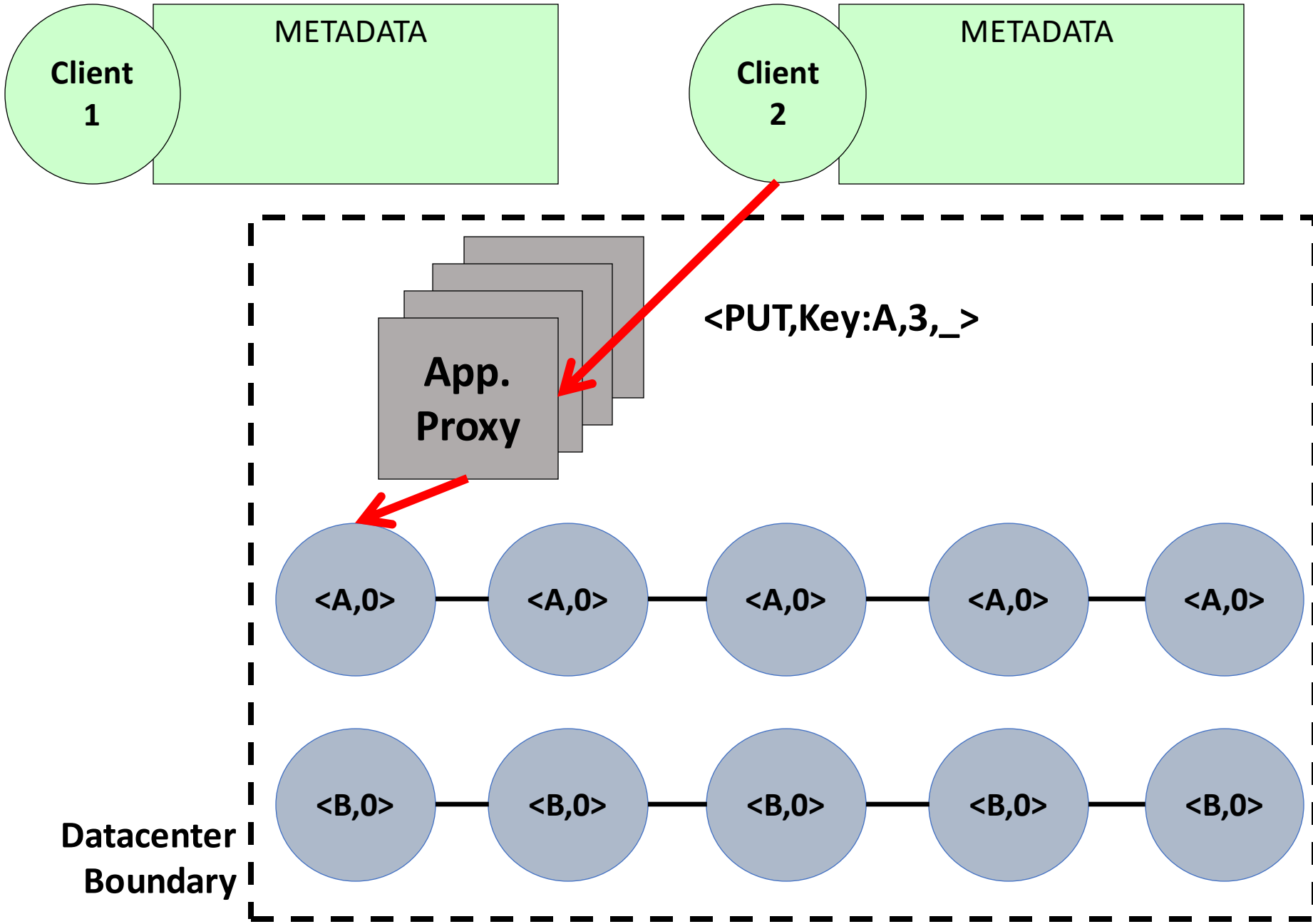
# ChainReaction: Replication Mechanism

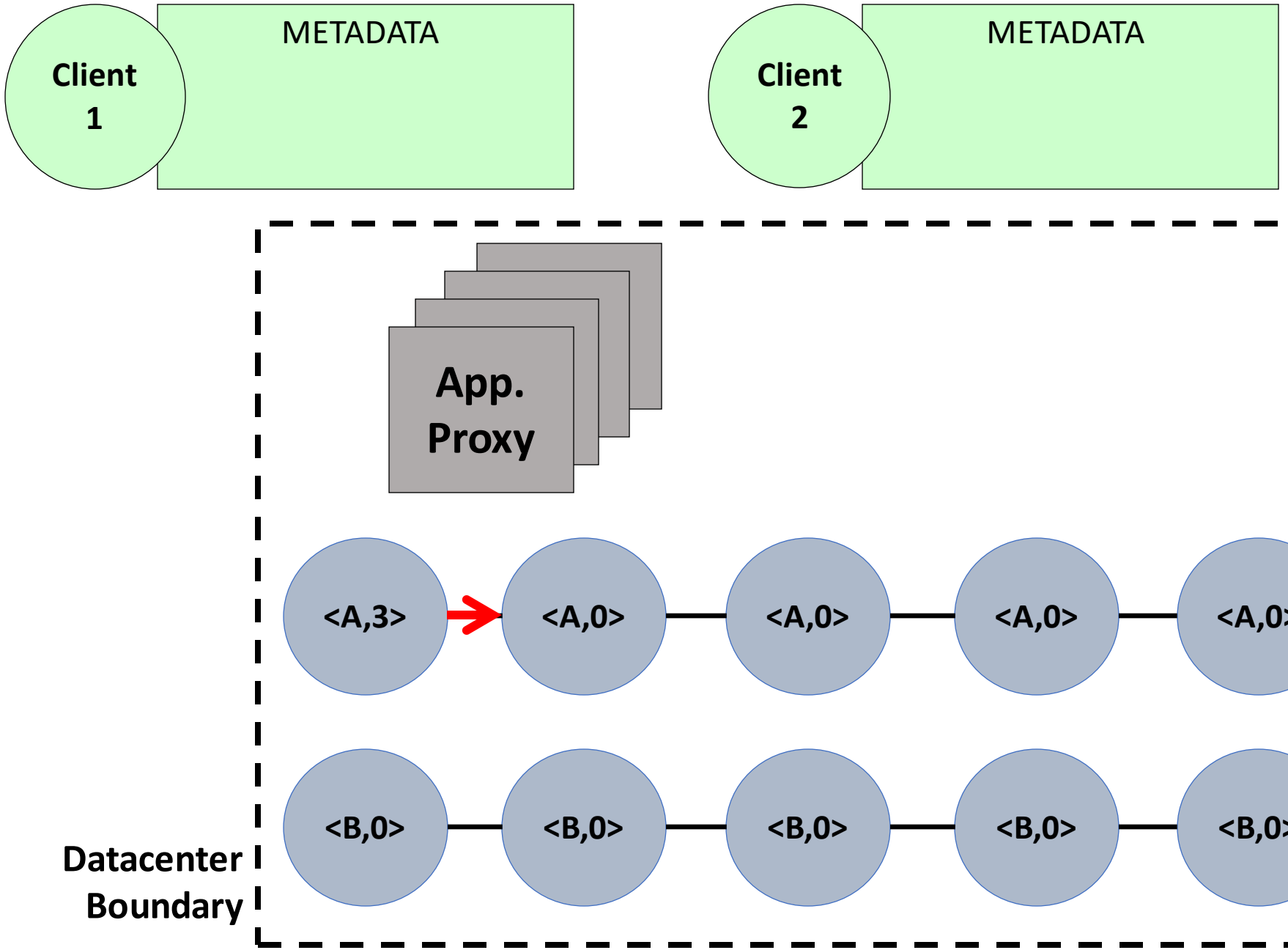
Allowing writes to return before reaching the tail.

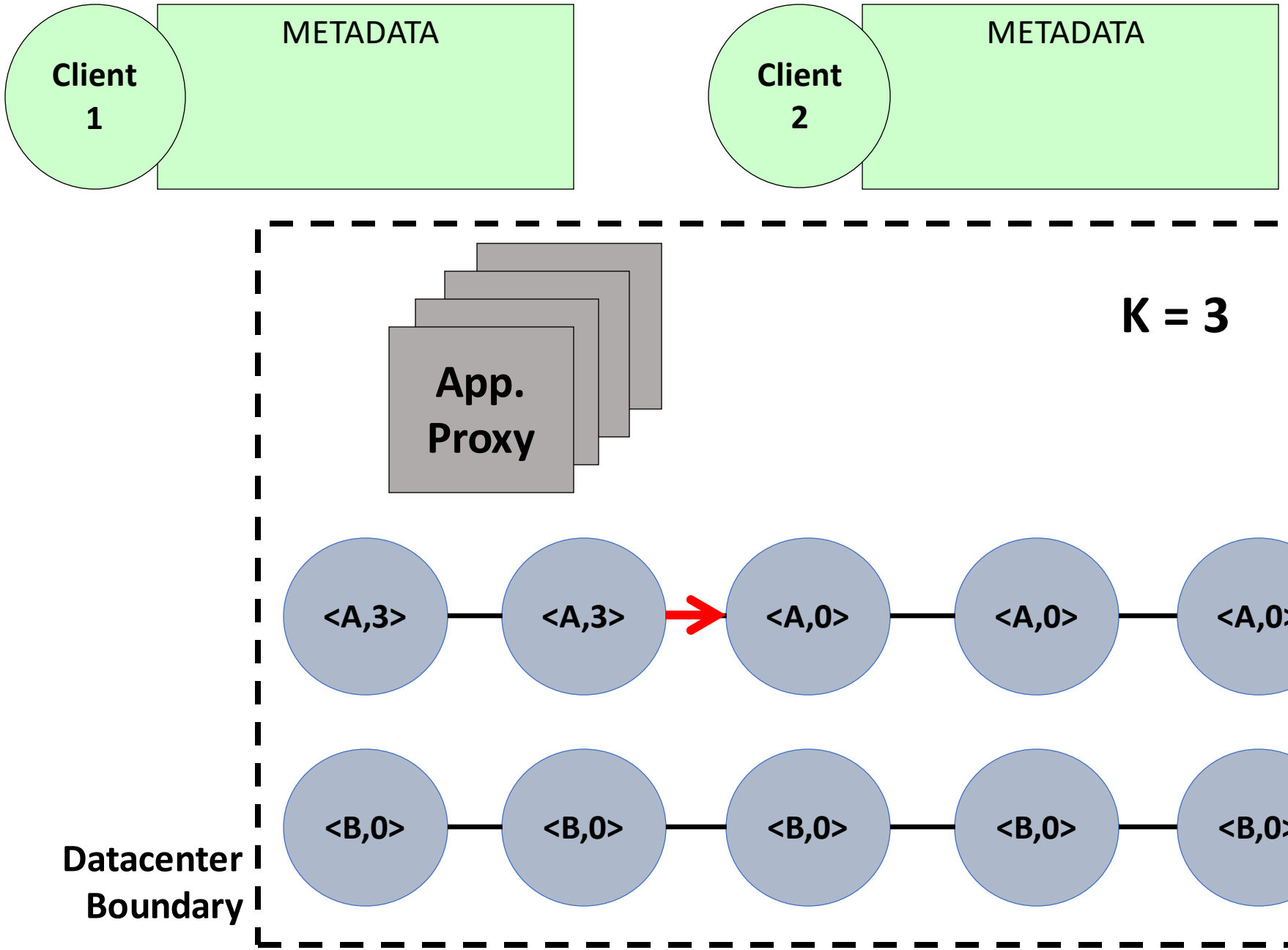
- Relax the number of nodes that have to process a write operation before returning the reply to the client:
  - This is controlled by a parameter K that can be set individually at each datacenter.
  - Allows for additional replicas to be added to the chain, that can share the load of reads without a significant penalization to the performance of write operations.

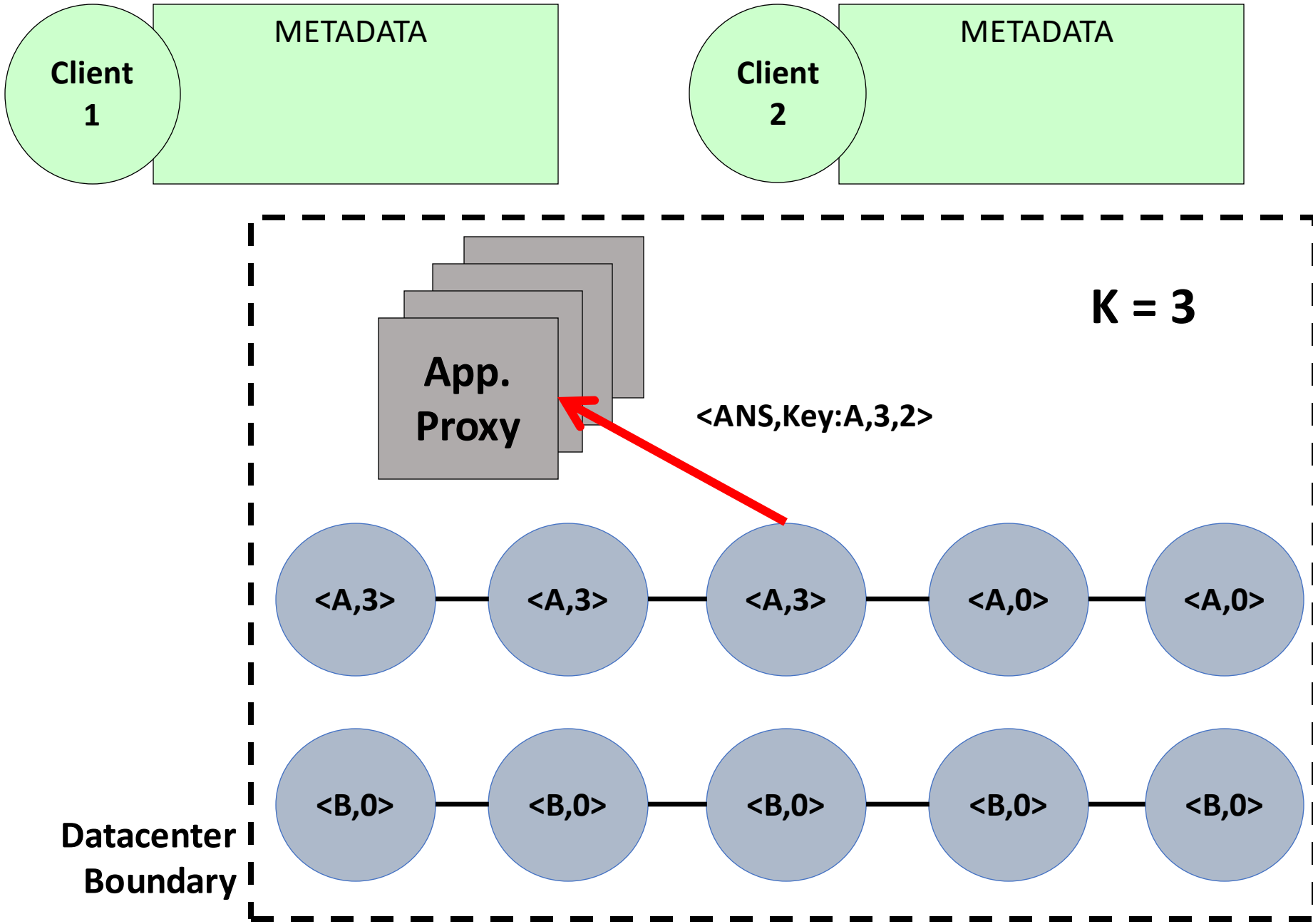

Client 1
METADATA
Client 2
METADATA
App. Proxy
<PUT,Key:A,3,_>
<A,0>
<A,0>
<A,0>
<A,0>
<A,0>
<B,0>
<B,0>
<B,0>
<B,0>
<B,0>
Datacenter Boundary

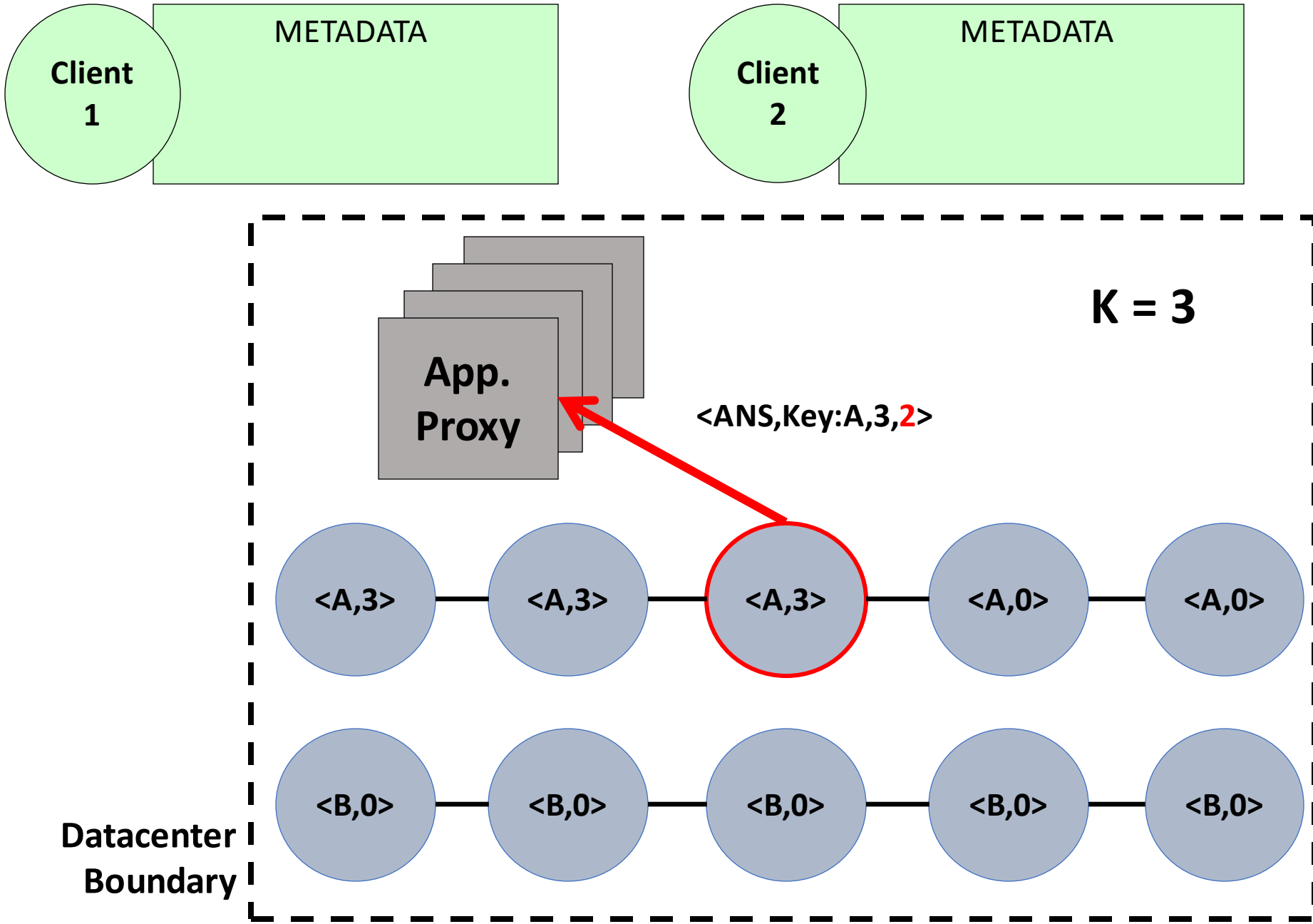

Client 1
METADATA
Client 2
METADATA
App. Proxy
<PUT,Key:A,3,_>
<A,0>
<A,0>
<A,0>
<A,0>
<A,0>
<B,0>
<B,0>
<B,0>
<B,0>
<B,0>
Datacenter Boundary

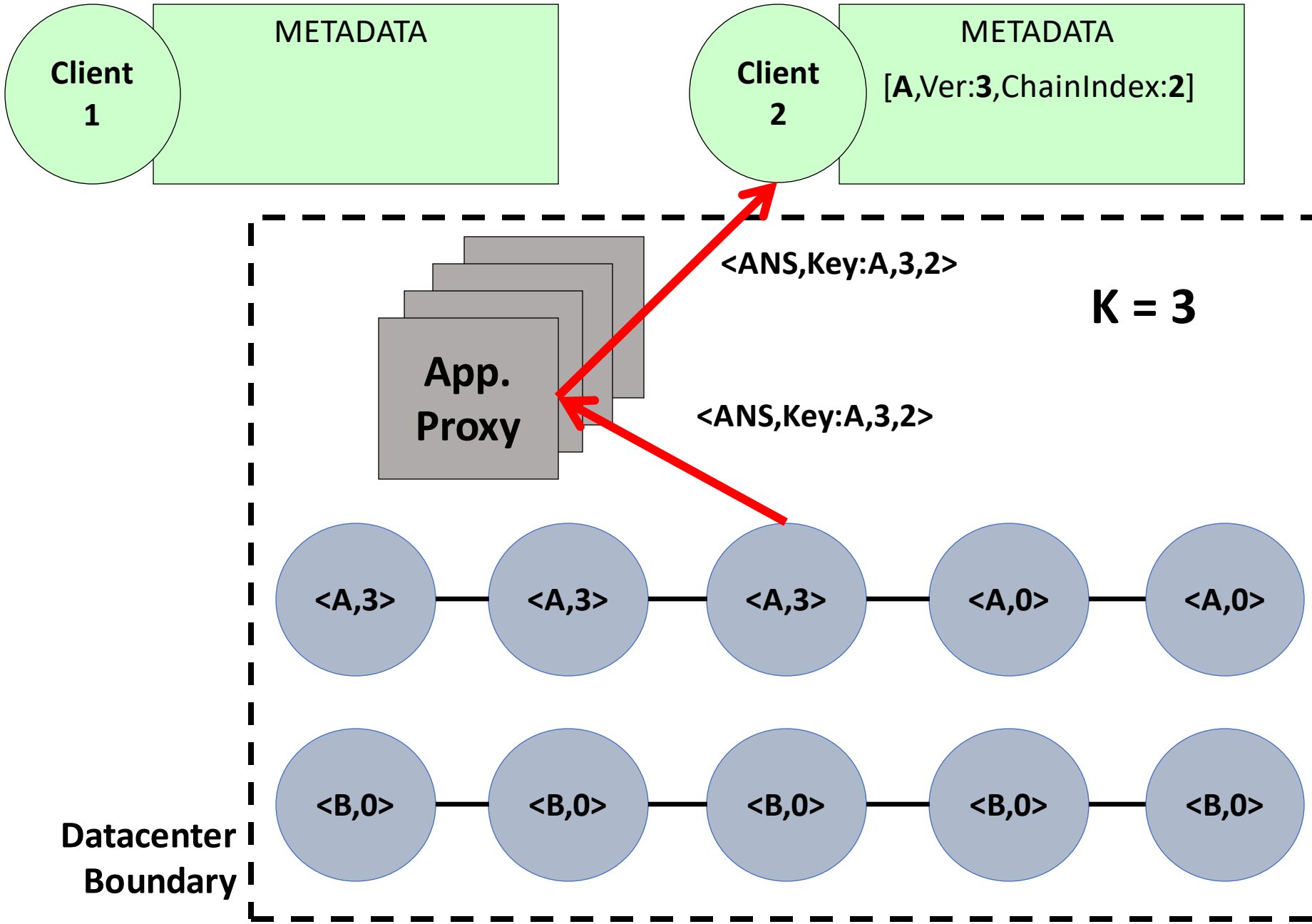

Client 1
METADATA
Client 2
METADATA
App. Proxy
<PUT,Key:A,3,_>
<A,0>
<A,0>
<A,0>
<A,0>
<A,0>
<B,0>
<B,0>
<B,0>
<B,0>
<B,0>
Datacenter Boundary

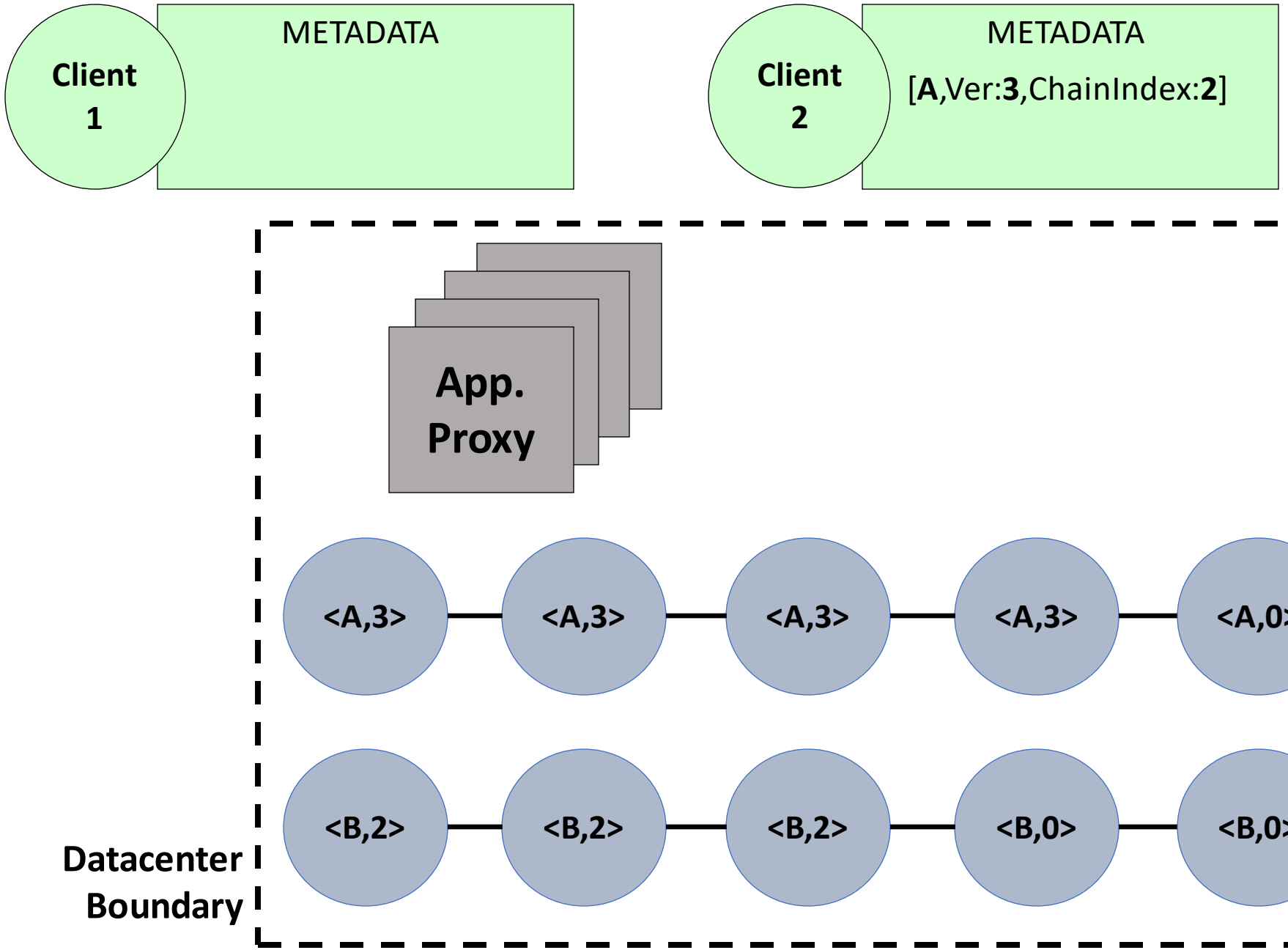

Client 1
METADATA
Client 2
METADATA
App. Proxy
<A,3>
<A,0>
<A,0>
<A,0>
<A,0>
<B,0>
<B,0>
<B,0>
<B,0>
<B,0>
Datacenter Boundary


Client 1
METADATA
Client 2
METADATA
App. Proxy
K = 3
<A,3>
<A,3>
<A,0>
<A,0>
<A,0>
<B,0>
<B,0>
<B,0>
<B,0>
<B,0>
Datacenter Boundary


Client 1
METADATA
Client 2
METADATA
K = 3
App. Proxy
<ANS,Key:A,3,2>
<A,3>
<A,3>
<A,3>
<A,0>
<A,0>
<B,0>
<B,0>
<B,0>
<B,0>
<B,0>
Datacenter Boundary


Client 1
METADATA
Client 2
METADATA
K = 3
App. Proxy
<ANS,Key:A,3,2>
<A,3>
<A,3>
<A,3>
<A,0>
<A,0>
<B,0>
<B,0>
<B,0>
<B,0>
<B,0>
Datacenter Boundary


Client 1
METADATA
Client 2
METADATA
[A,Ver:3,ChainIndex:2]
App. Proxy
<ANS,Key:A,3,2>
<ANS,Key:A,3,2>
K = 3
<A,3>
<A,3>
<A,3>
<A,0>
<A,0>
<B,0>
<B,0>
<B,0>
<B,0>
<B,0>
Datacenter Boundary


Client 1
METADATA
Client 2
METADATA
[A,Ver:3,ChainIndex:2]
App. Proxy
<A,3>
<A,3>
<A,3>
<A,3>
<A,0>
<B,2>
<B,2>
<B,2>
<B,0>
<B,0>
Datacenter Boundary

# ChainReaction: Replication Mechanism

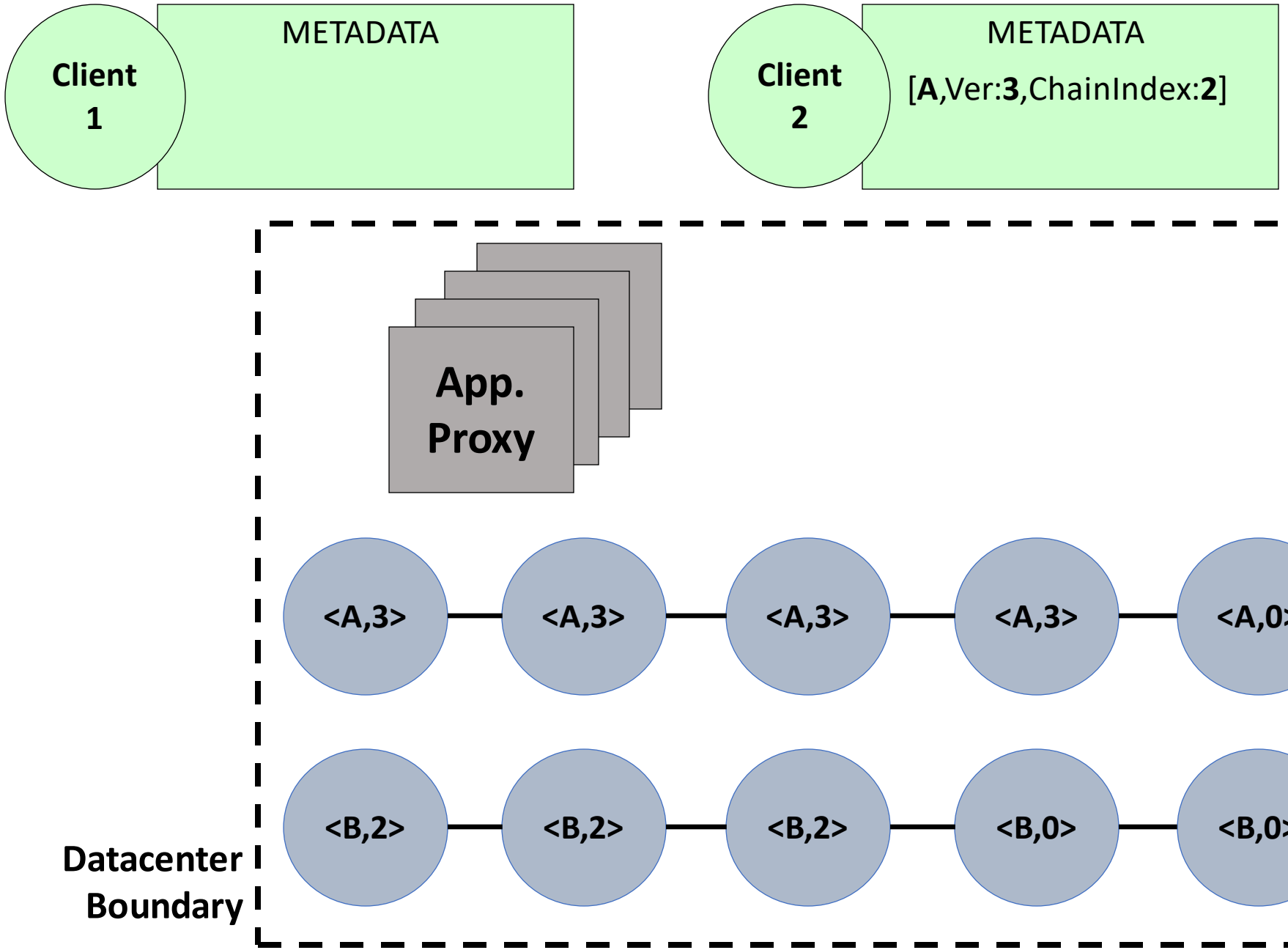
Relies on a specialized version of Chain Replication where:

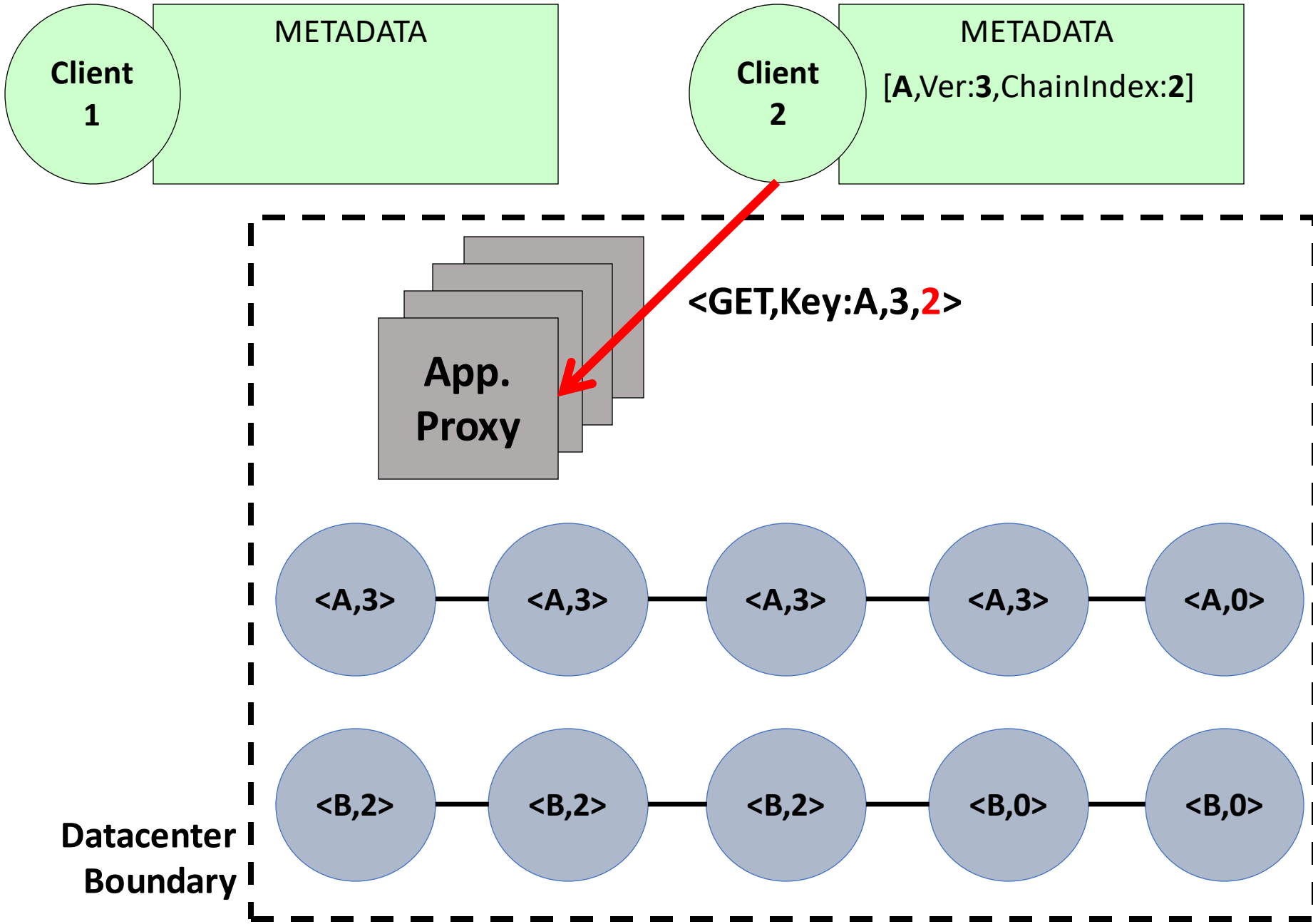
- Allow writes to return before reaching the tail.
- Support reads on all nodes of the chain.
- Trades write efficiency for metadata efficiency.

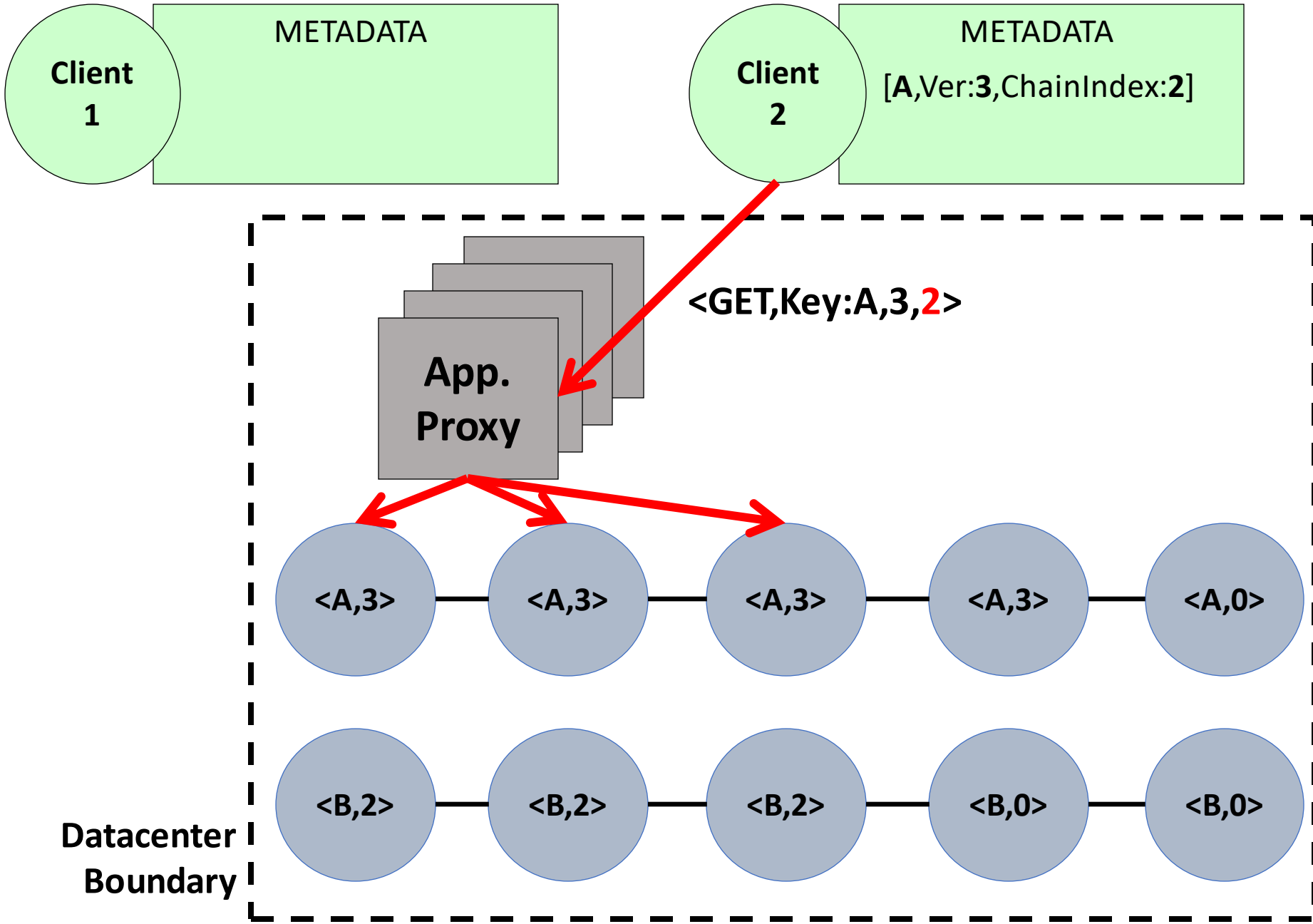
# ChainReaction: Replication Mechanism

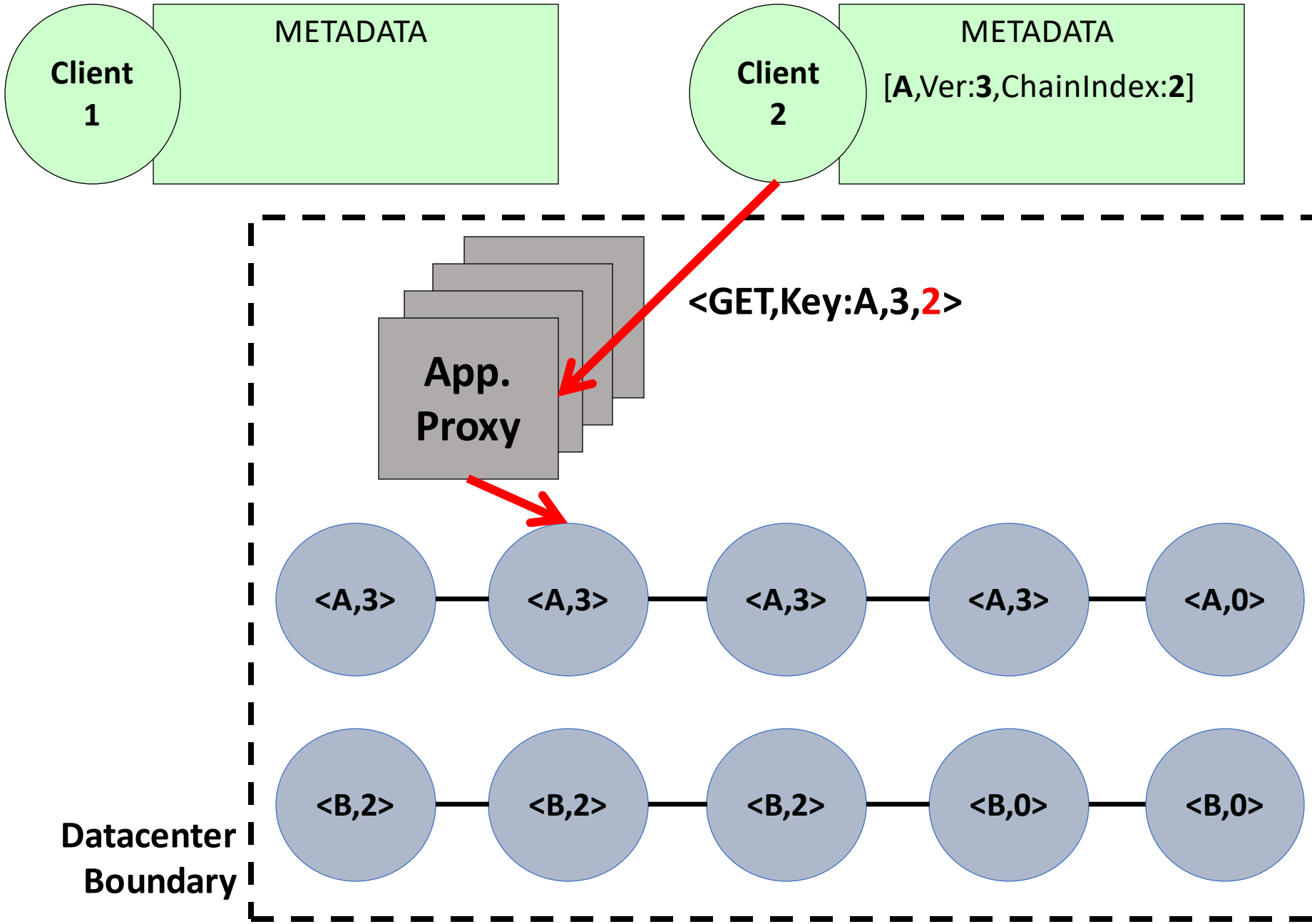
- Supporting reads on all nodes of the chain.

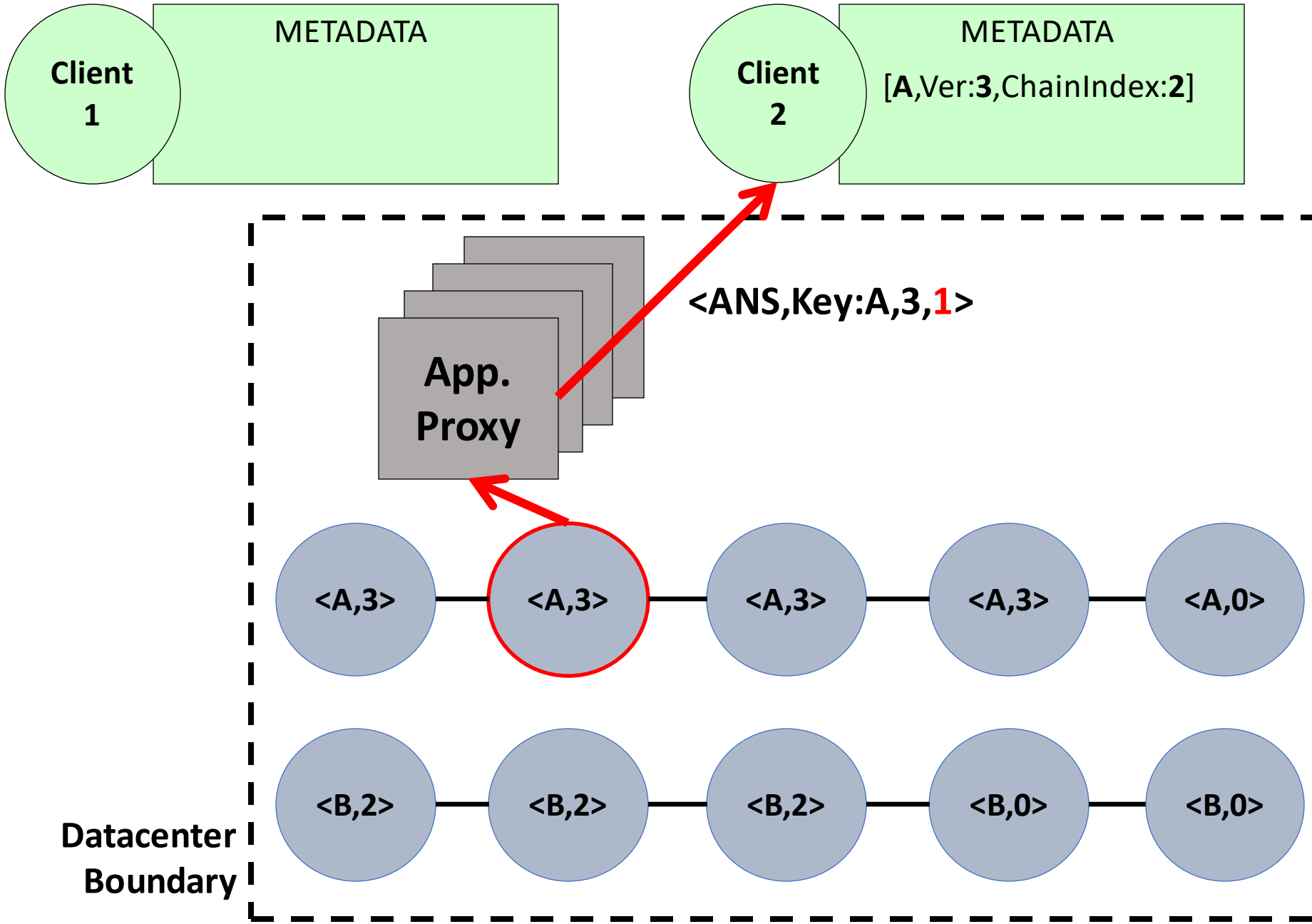
- Clients have to be aware of which node served their last read for each data object (i.e., key).
- This is important to ensure that if a non-stable version (i.e., a version that was not yet propagated and applied all nodes of the chain) is observed a future read will either return the same version or a newer one.

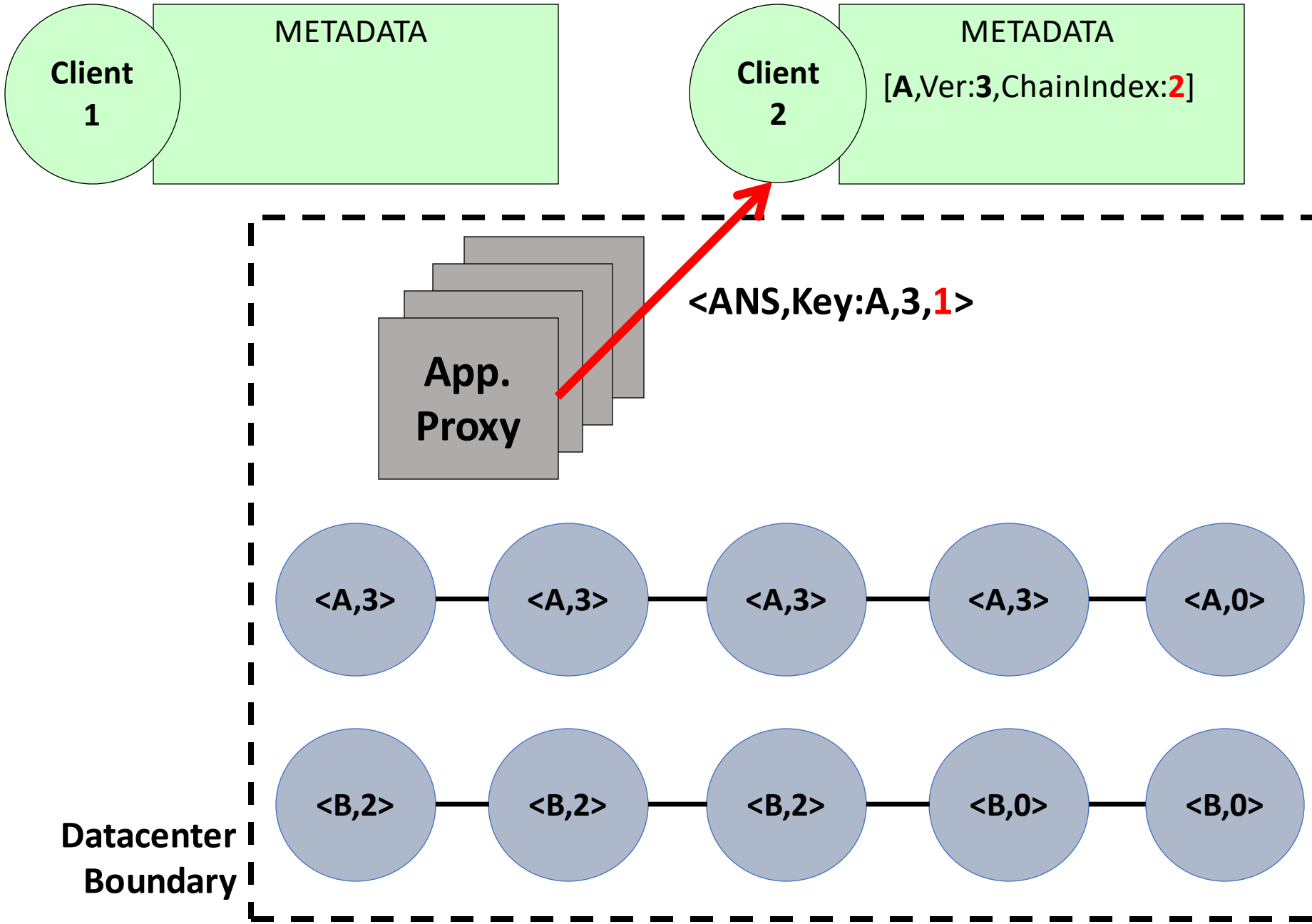

Client 1
METADATA
Client 2
METADATA
[A,Ver:3,ChainIndex:2]
App. Proxy
<A,3>
<A,3>
<A,3>
<A,3>
<A,0>
<B,2>
<B,2>
<B,2>
<B,0>
<B,0>
Datacenter Boundary

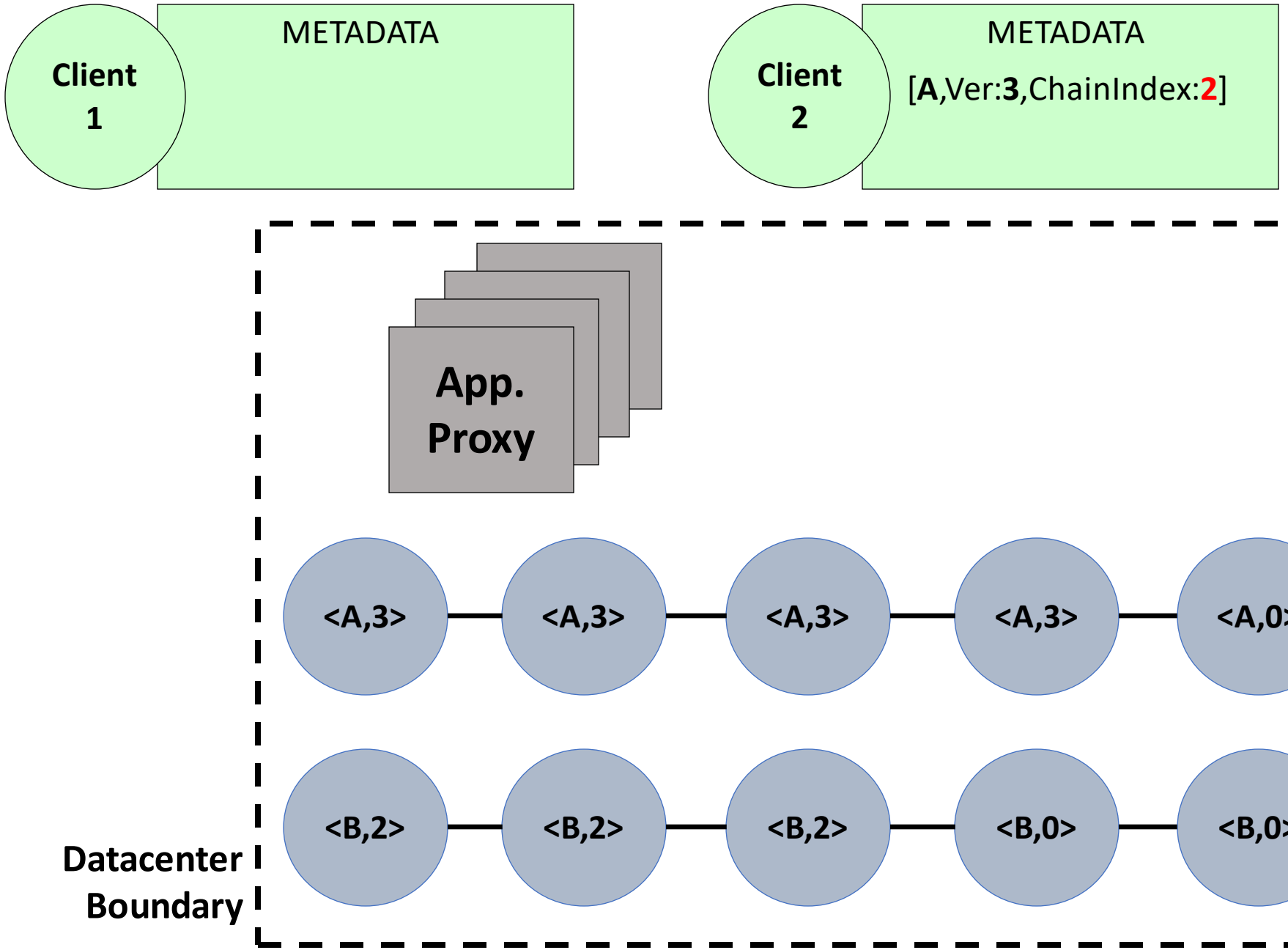

Client 1
METADATA
Client 2
METADATA
[A,Ver:3,ChainIndex:2]
App. Proxy
<GET,Key:A,3,2>
<A,3>
<A,3>
<A,3>
<A,3>
<A,0>
<B,2>
<B,2>
<B,2>
<B,0>
<B,0>
Datacenter Boundary

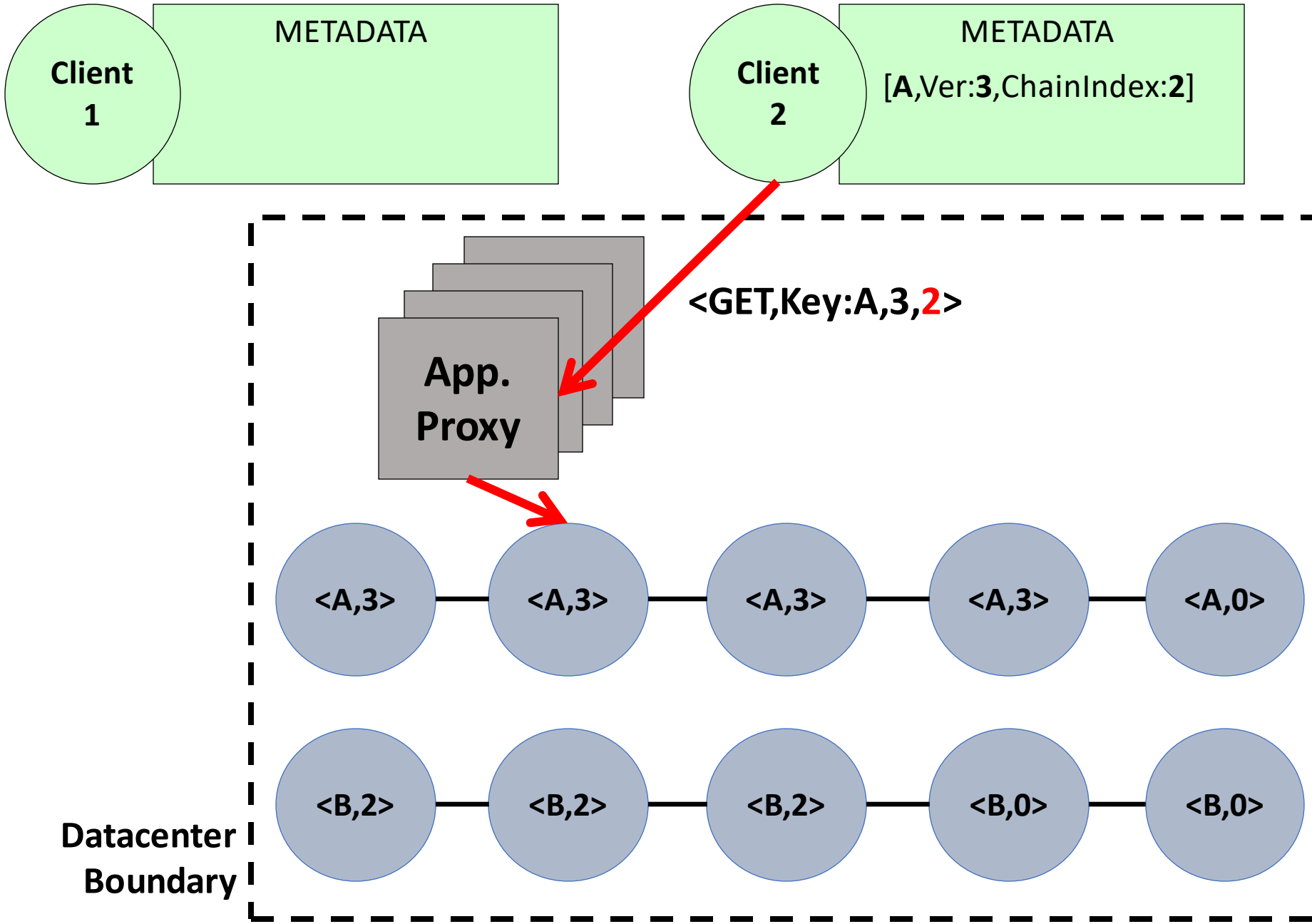

Client 1
METADATA
Client 2
METADATA
[A,Ver:3,ChainIndex:2]
<GET,Key:A,3,2>
App. Proxy
<A,3>
<A,3>
<A,3>
<A,3>
<A,0>
<B,2>
<B,2>
<B,2>
<B,0>
<B,0>
Datacenter Boundary

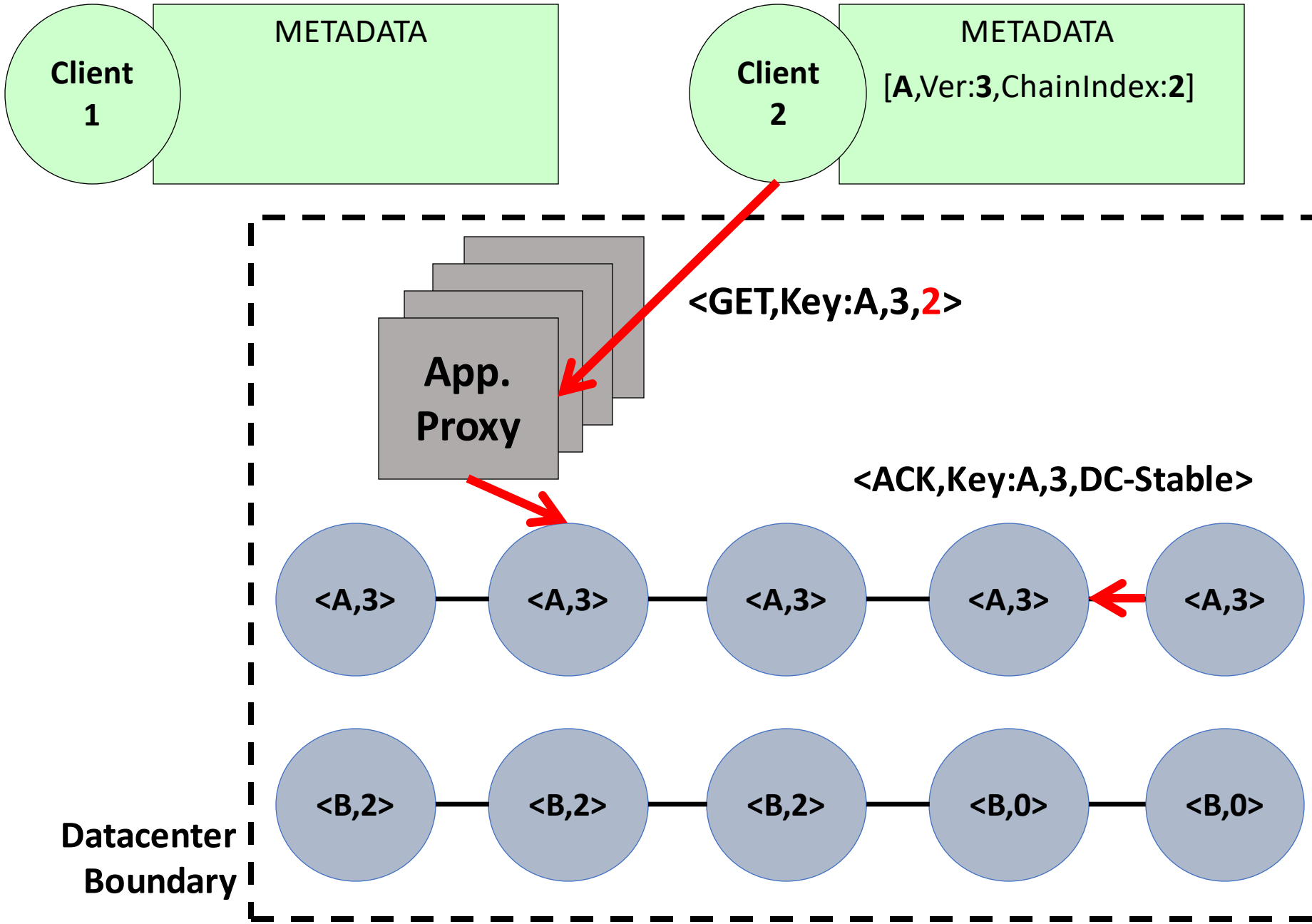

Client 1
METADATA
Client 2
METADATA
[A,Ver:3,ChainIndex:2]
App. Proxy
<GET,Key:A,3,2>
<A,3>
<A,3>
<A,3>
<A,3>
<A,0>
<B,2>
<B,2>
<B,2>
<B,0>
<B,0>
Datacenter Boundary

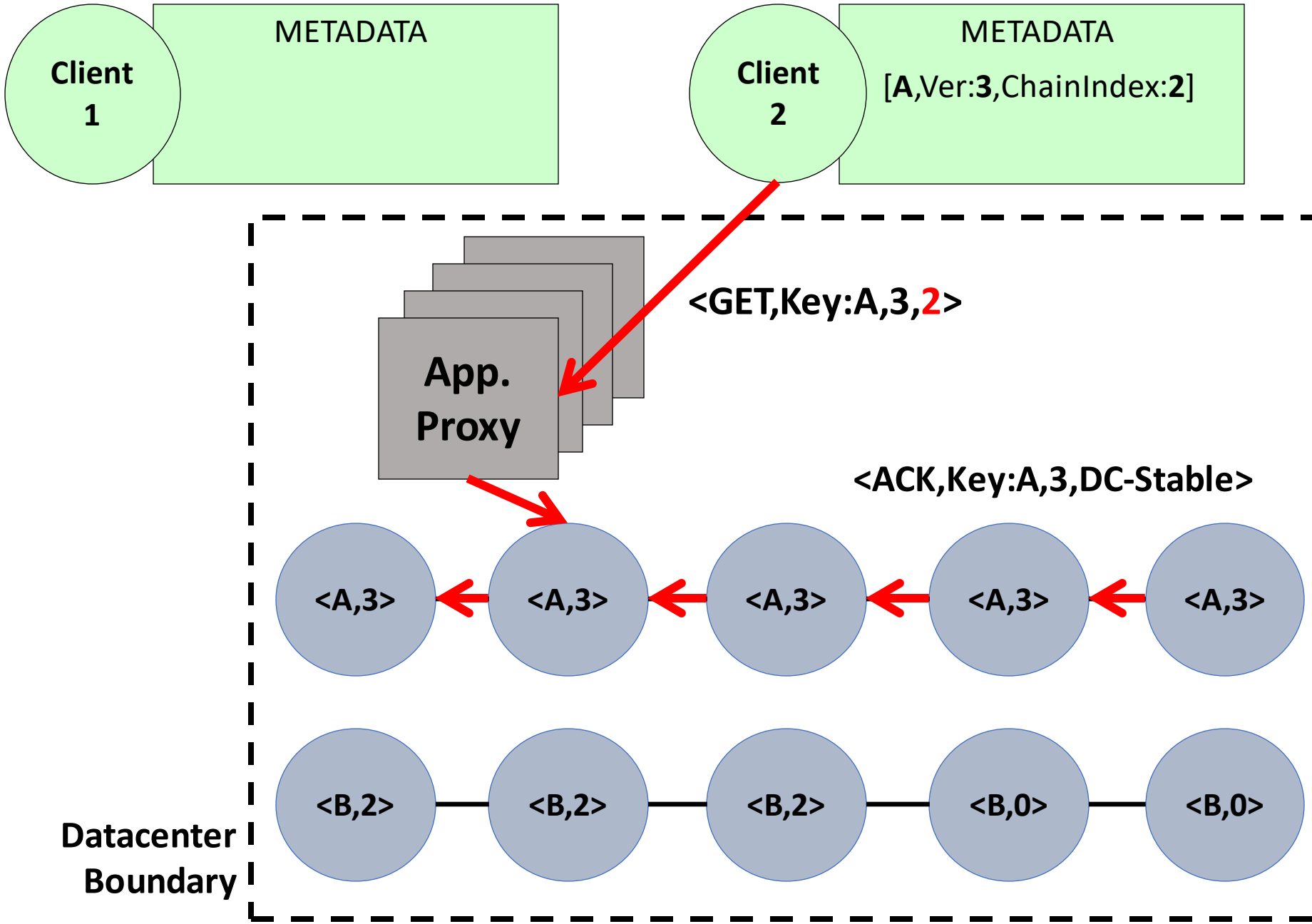

Client 1
METADATA
Client 2
METADATA
[A,Ver:3,ChainIndex:2]
App. Proxy
<ANS,Key:A,3,1>
<A,3>
<A,3>
<A,3>
<A,3>
<A,0>
<B,2>
<B,2>
<B,2>
<B,0>
<B,0>
Datacenter Boundary

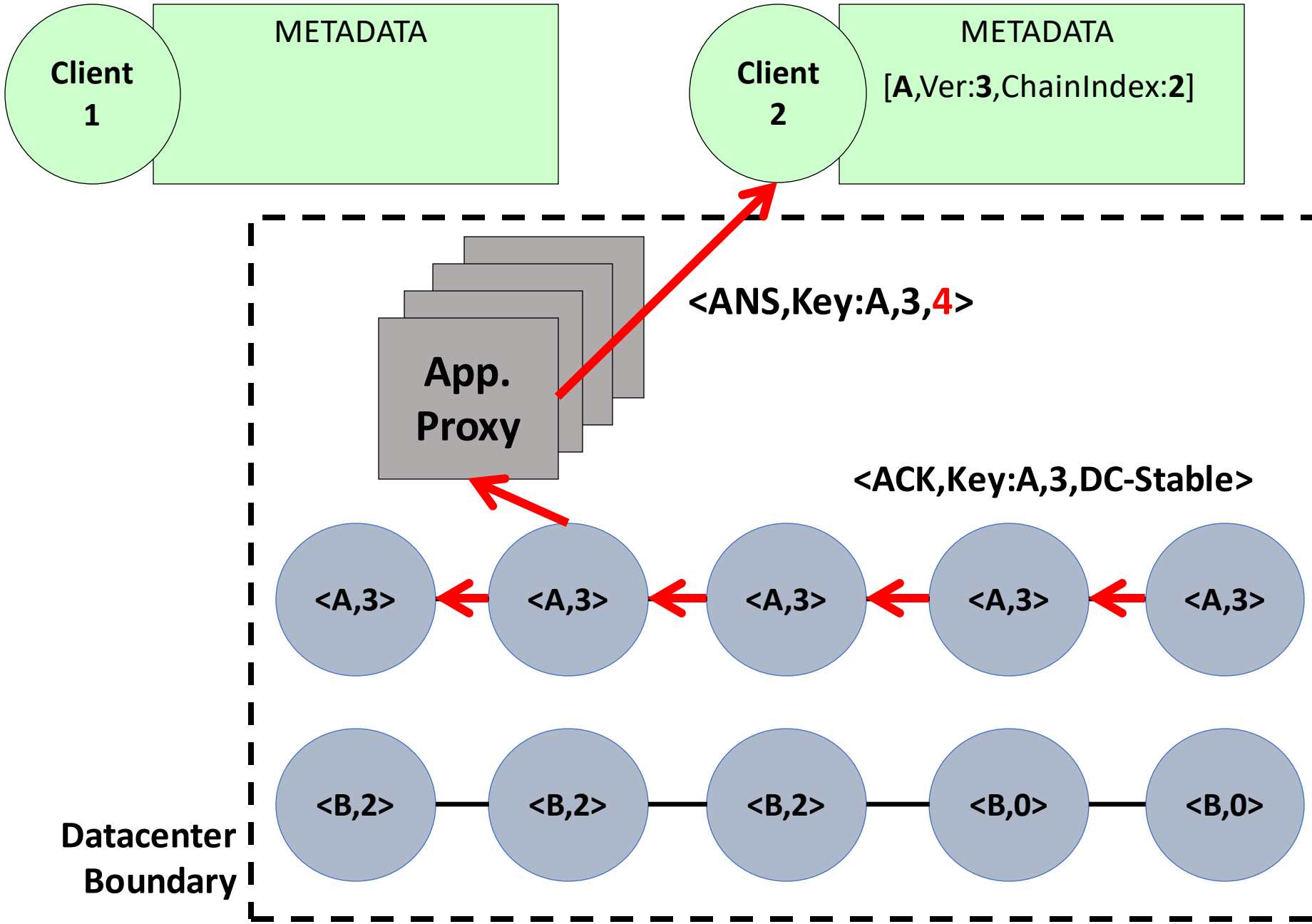

Client 1
METADATA
Client 2
METADATA
[A,Ver:3,ChainIndex:2]
App. Proxy
<ANS,Key:A,3,1>
<A,3>
<A,3>
<A,3>
<A,3>
<A,0>
<B,2>
<B,2>
<B,2>
<B,0>
<B,0>
Datacenter Boundary

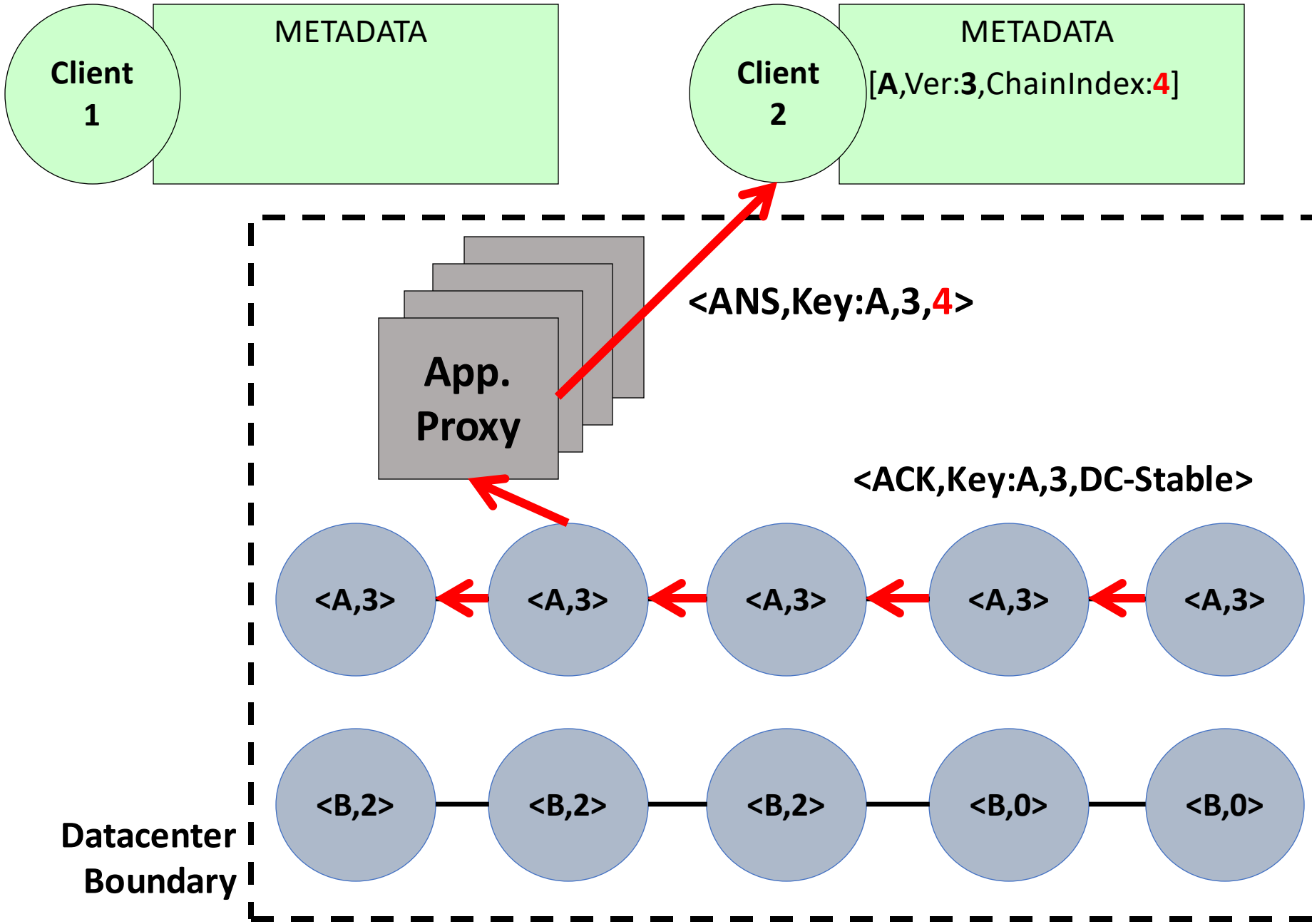

Client 1
METADATA
Client 2
METADATA
[A,Ver:3,ChainIndex:2]
App. Proxy
<A,3>
<A,3>
<A,3>
<A,3>
<A,0>
<B,2>
<B,2>
<B,2>
<B,0>
<B,0>
Datacenter Boundary

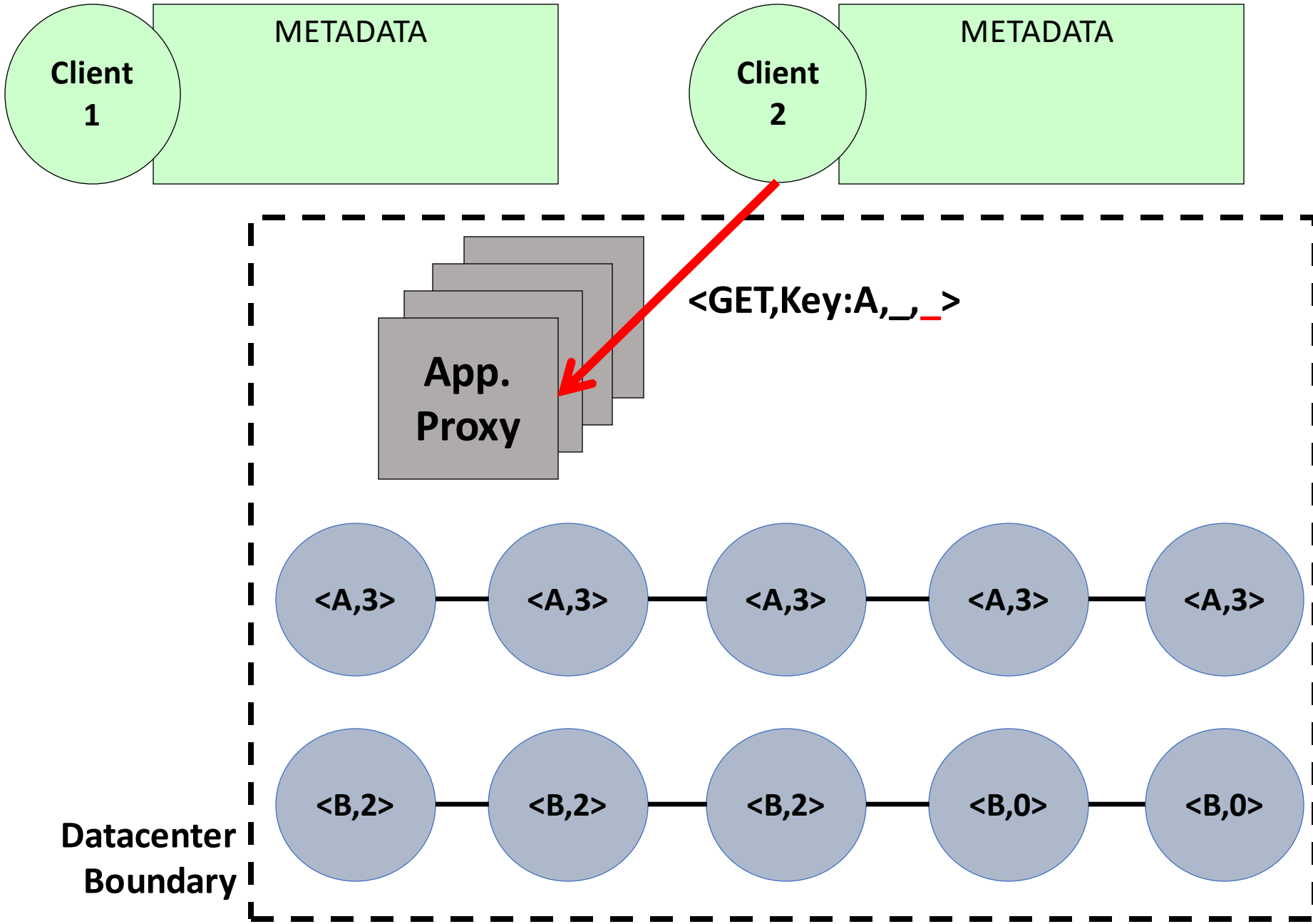

Client 1
METADATA
Client 2
METADATA
[A,Ver:3,ChainIndex:2]
App. Proxy
<GET,Key:A,3,2>
<A,3>
<A,3>
<A,3>
<A,3>
<A,0>
<B,2>
<B,2>
<B,2>
<B,0>
<B,0>
Datacenter Boundary

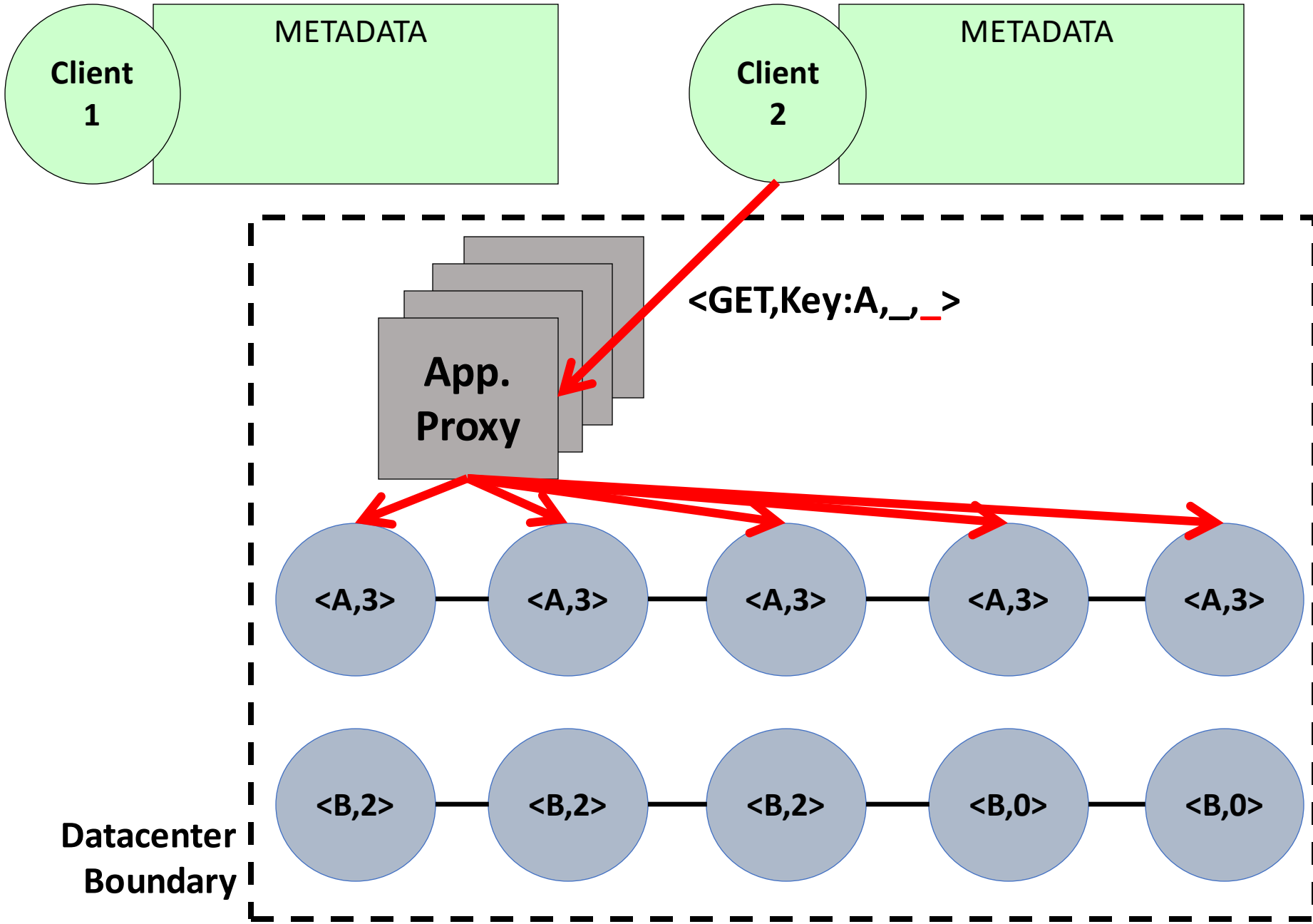

Client 1
METADATA
Client 2
METADATA
[A,Ver:3,ChainIndex:2]
App. Proxy
<GET,Key:A,3,2>
<ACK,Key:A,3,DC-Stable>
<A,3>
<A,3>
<A,3>
<A,3>
<A,3>
<B,2>
<B,2>
<B,2>
<B,0>
<B,0>
Datacenter Boundary


Client 1
METADATA
Client 2
METADATA
[A,Ver:3,ChainIndex:2]
App. Proxy
<GET,Key:A,3,2>
<ACK,Key:A,3,DC-Stable>
<A,3>
<A,3>
<A,3>
<A,3>
<A,3>
<B,2>
<B,2>
<B,2>
<B,0>
<B,0>
Datacenter Boundary


Client 1
METADATA
Client 2
METADATA
[A,Ver:3,ChainIndex:2]
App. Proxy
<ANS,Key:A,3,4>
<ACK,Key:A,3,DC-Stable>
<A,3>
<A,3>
<A,3>
<A,3>
<A,3>
<B,2>
<B,2>
<B,2>
<B,0>
<B,0>
Datacenter Boundary


Client 1
METADATA
Client 2
METADATA
[A,Ver:3,ChainIndex:4]
App. Proxy
<ANS,Key:A,3,4>
<ACK,Key:A,3,DC-Stable>
<A,3>
<A,3>
<A,3>
<A,3>
<A,3>
<B,2>
<B,2>
<B,2>
<B,0>
<B,0>
Datacenter Boundary


Client 1
METADATA
Client 2
METADATA
<GET,Key:A,_,_>
App. Proxy
<A,3>
<A,3>
<A,3>
<A,3>
<A,3>
<B,2>
<B,2>
<B,2>
<B,0>
<B,0>
Datacenter Boundary


Client 1
METADATA
Client 2
METADATA
<GET,Key:A,_,_>
App. Proxy
<A,3>
<A,3>
<A,3>
<A,3>
<A,3>
<B,2>
<B,2>
<B,2>
<B,0>
<B,0>
Datacenter Boundary

# ChainReaction: Replication Mechanism

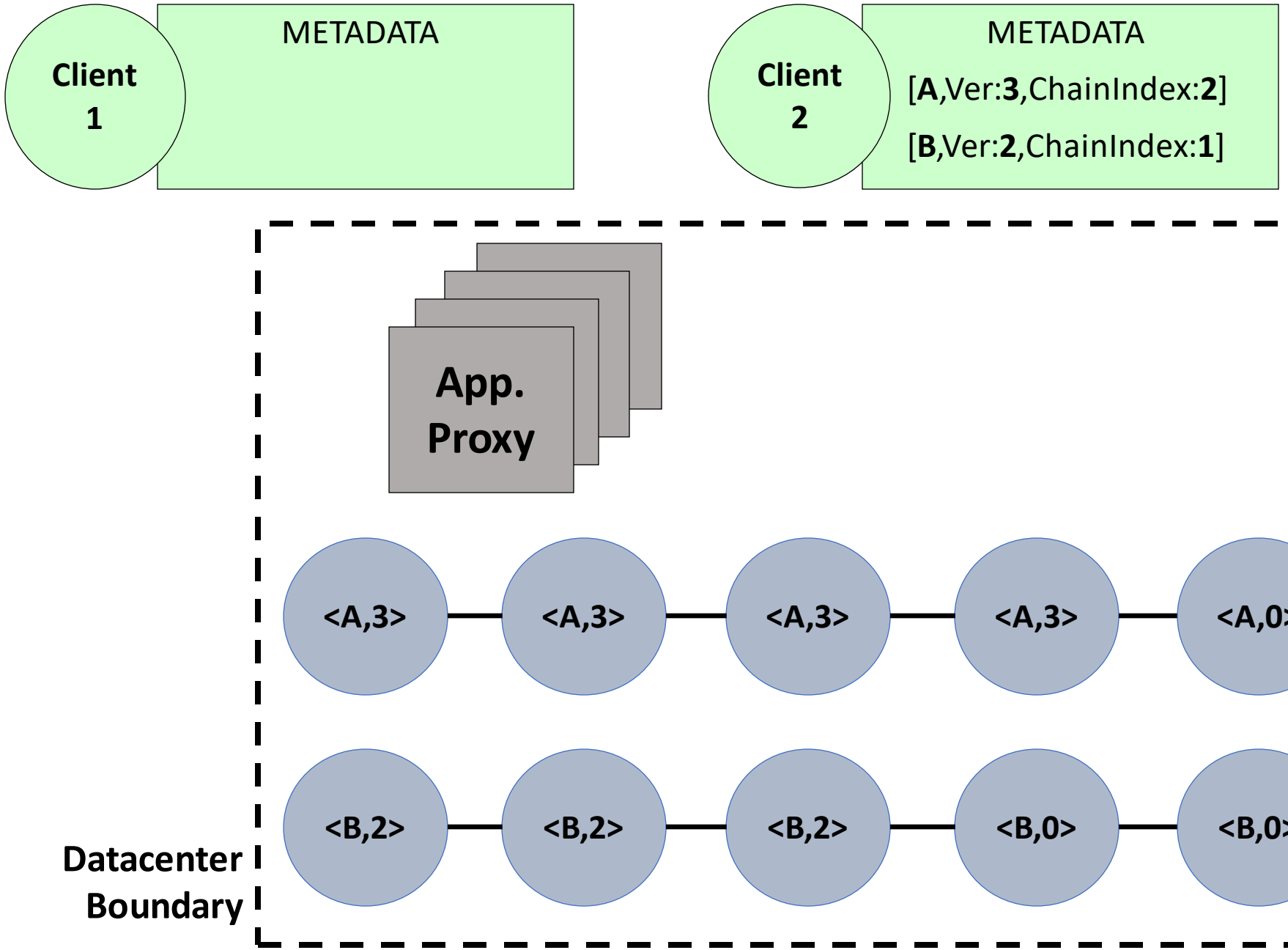
Relies on a specialized version of Chain Replication where:

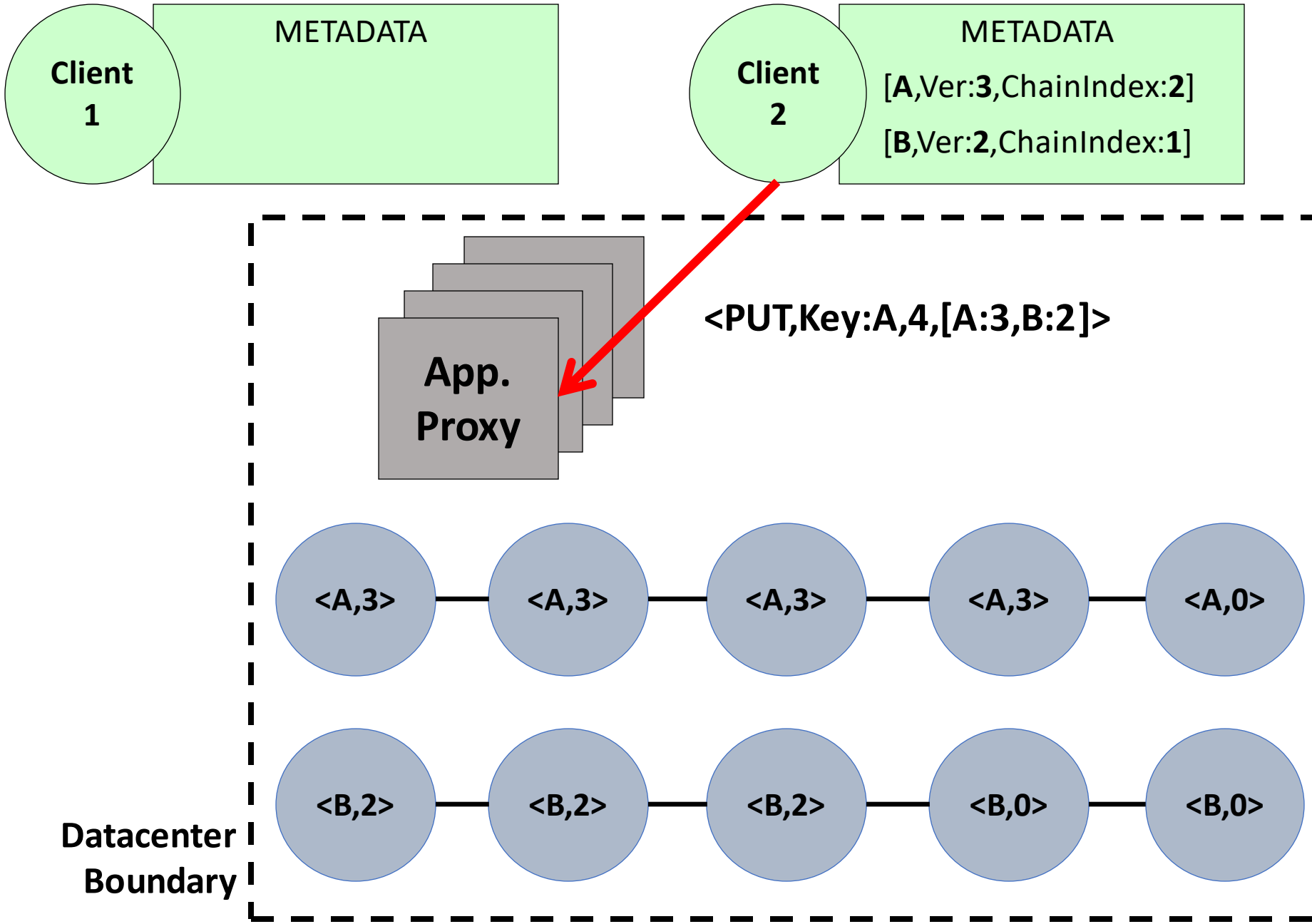
- Allows writes to return before reaching the tail.
- Support reads on all nodes of the chain.
- Trades write efficiency for metadata efficiency.

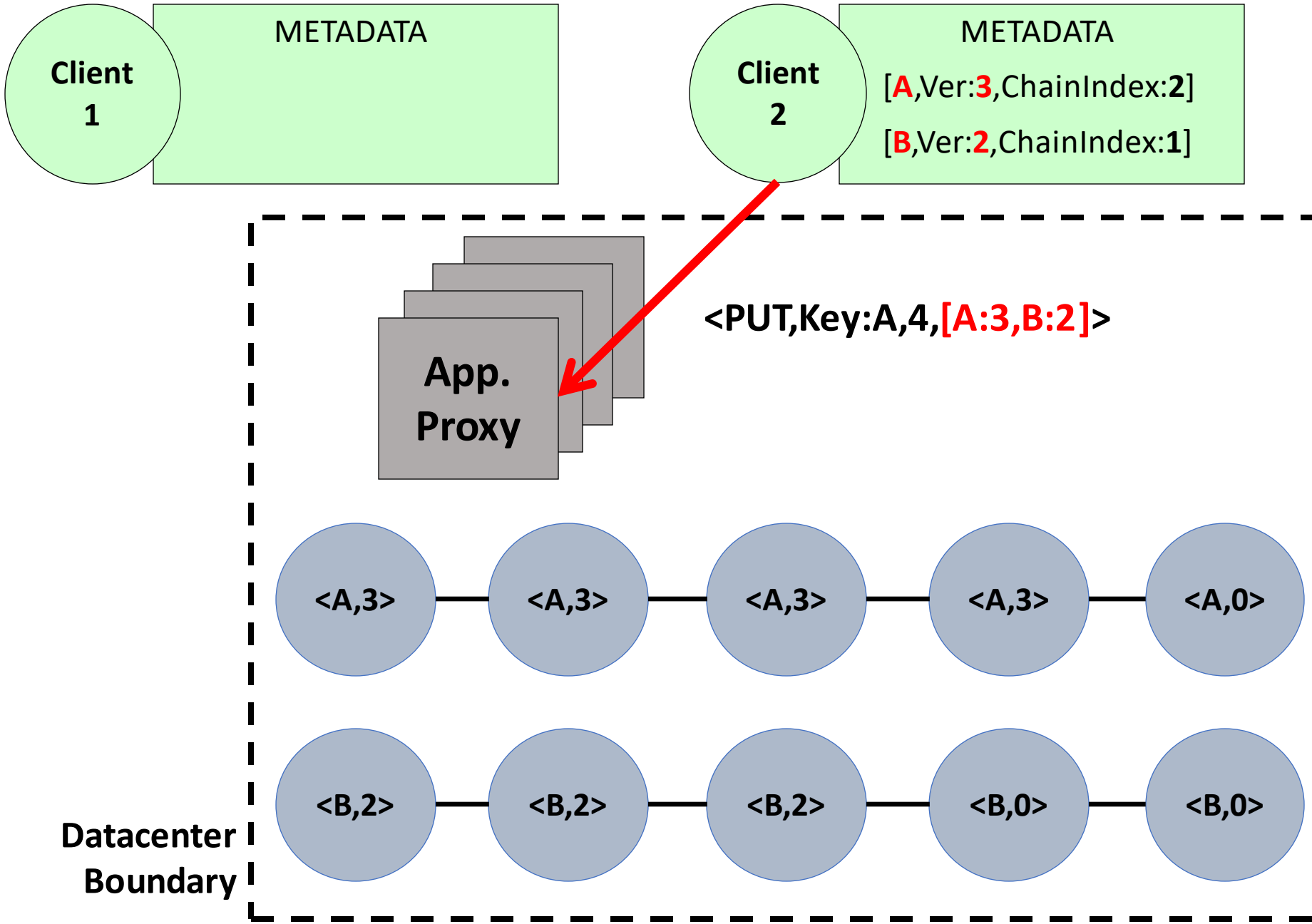
# ChainReaction: Replication Mechanism

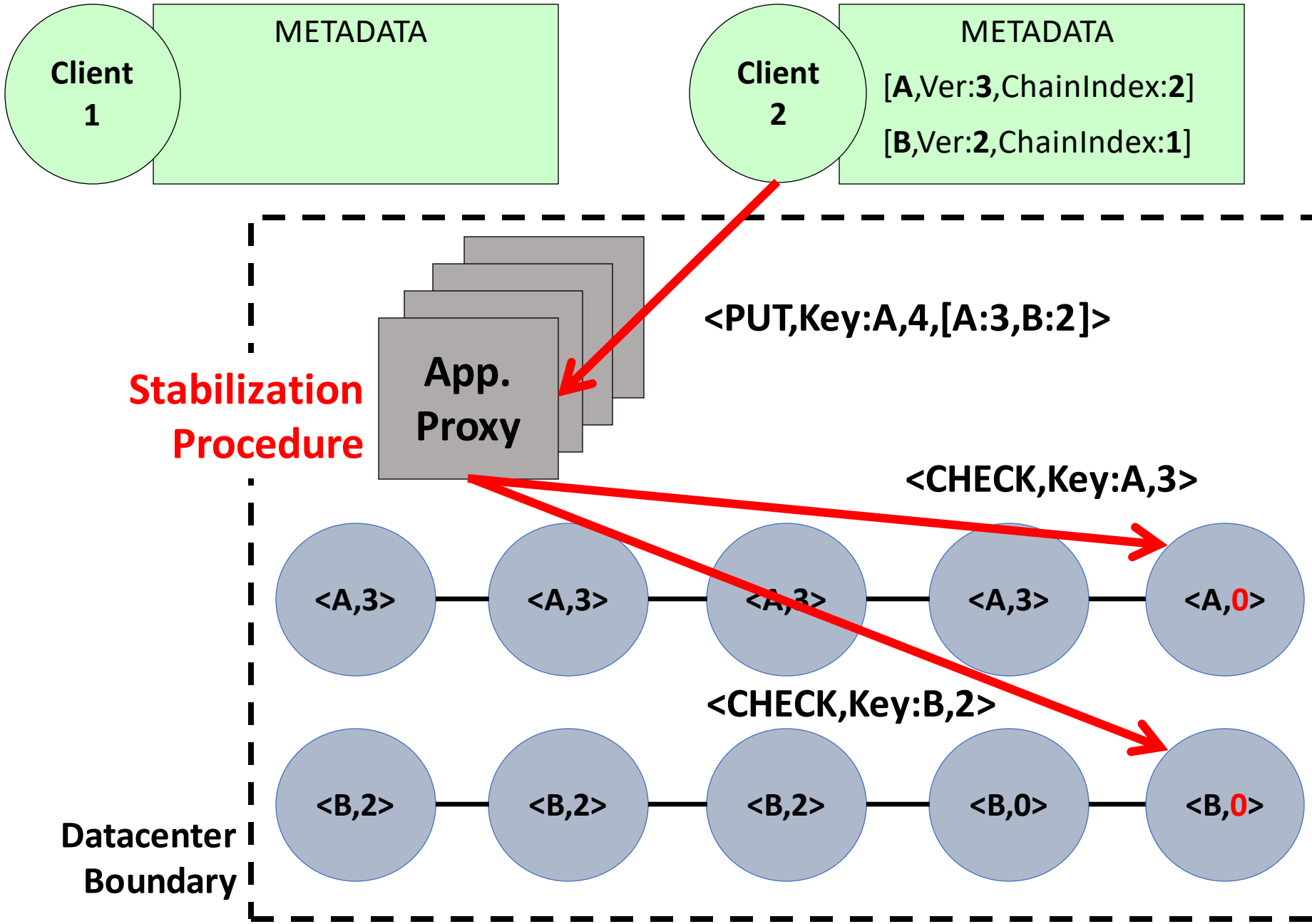
Minimizing metadata by making writes slightly slower:

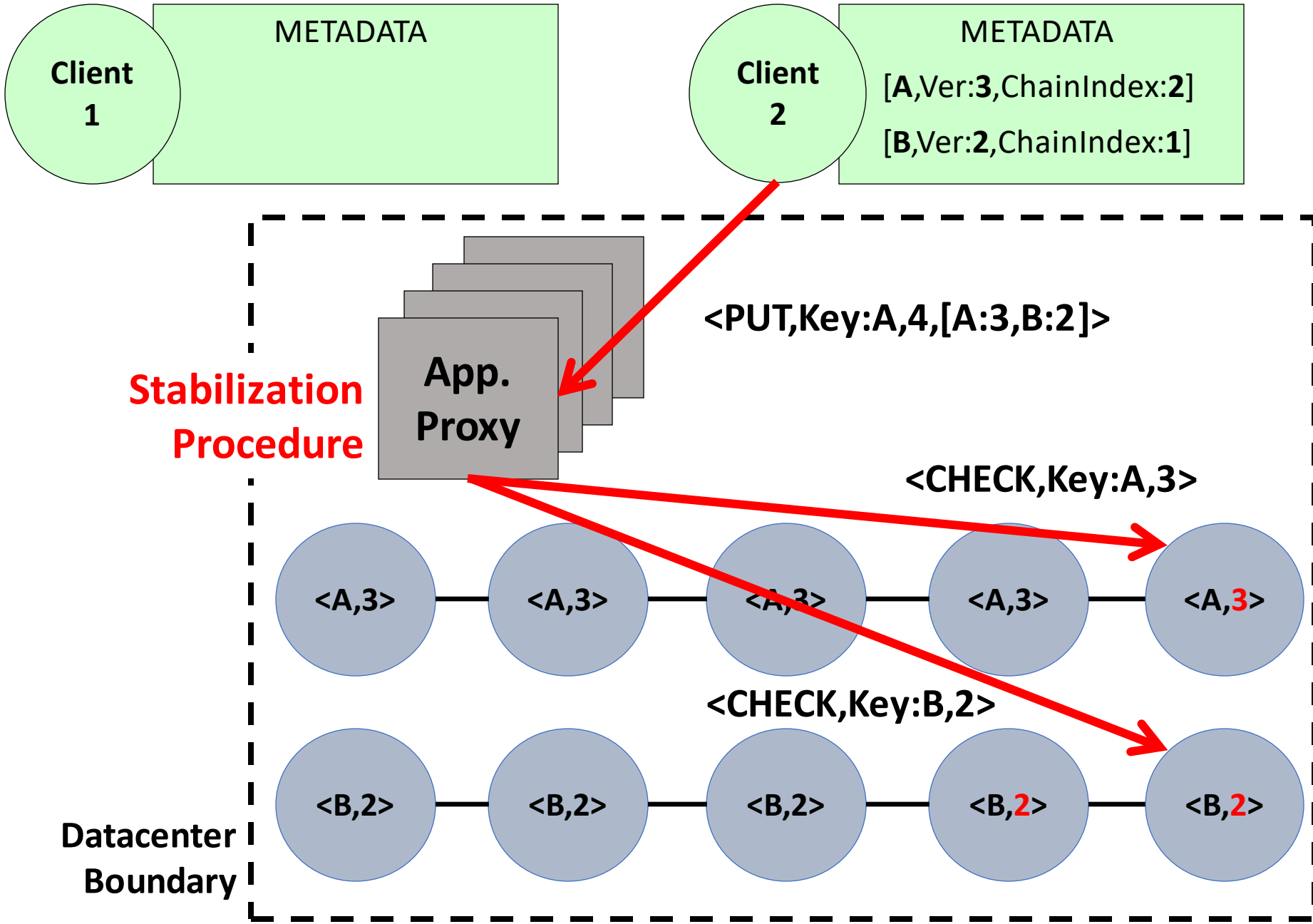
- Leverage on a stabilization procedure performed before each (individual) write operation in which:
  - Ensures that causal+ consistency will not be violated.
  - Lowers the number of metadata entries that must be maintained by each client.

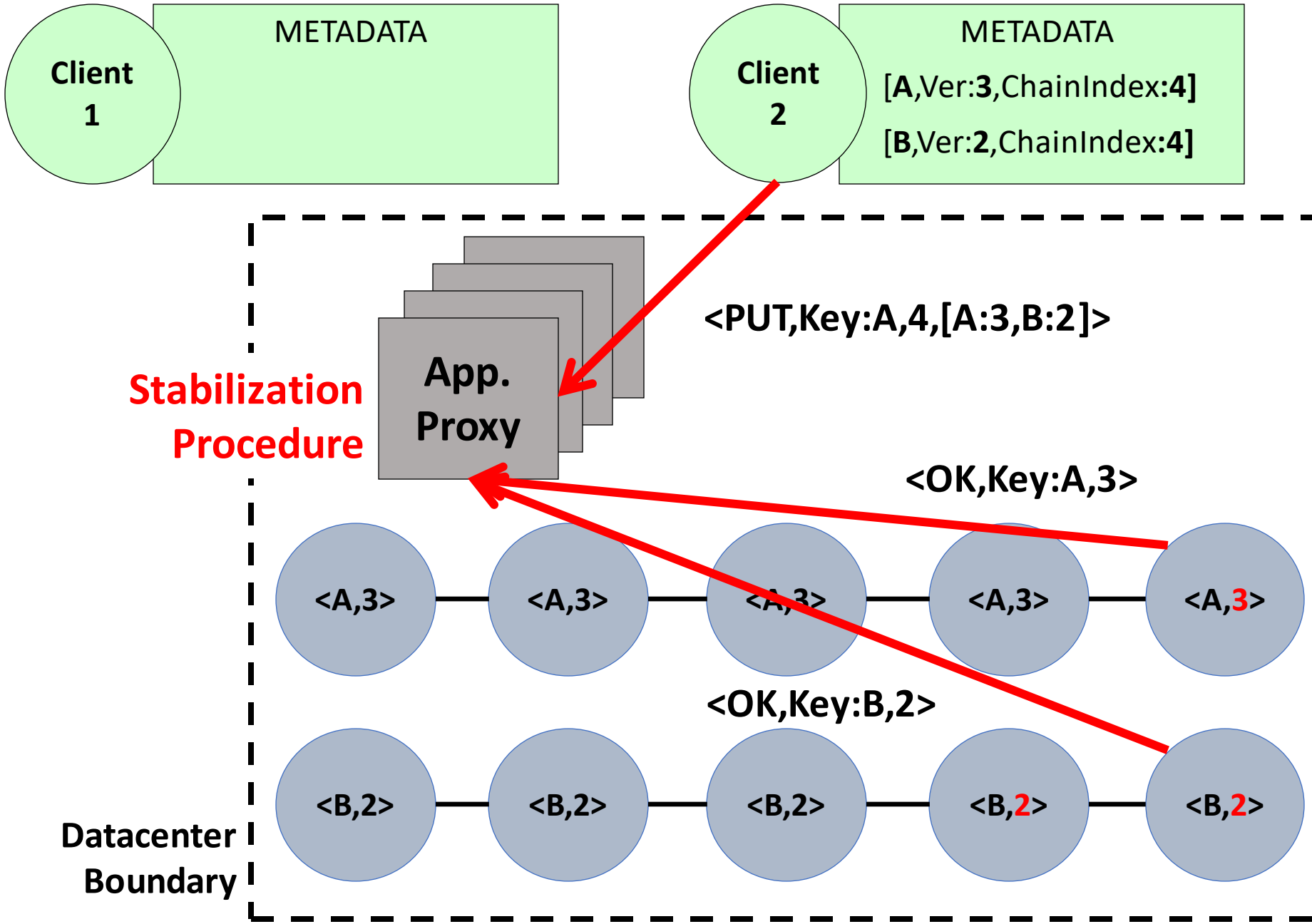

Client 1
METADATA
Client 2
METADATA
[A,Ver:3,ChainIndex:2]
[B,Ver:2,ChainIndex:1]
App. Proxy
<A,3>
<A,3>
<A,3>
<A,3>
<A,0>
<B,2>
<B,2>
<B,2>
<B,0>
<B,0>
Datacenter Boundary

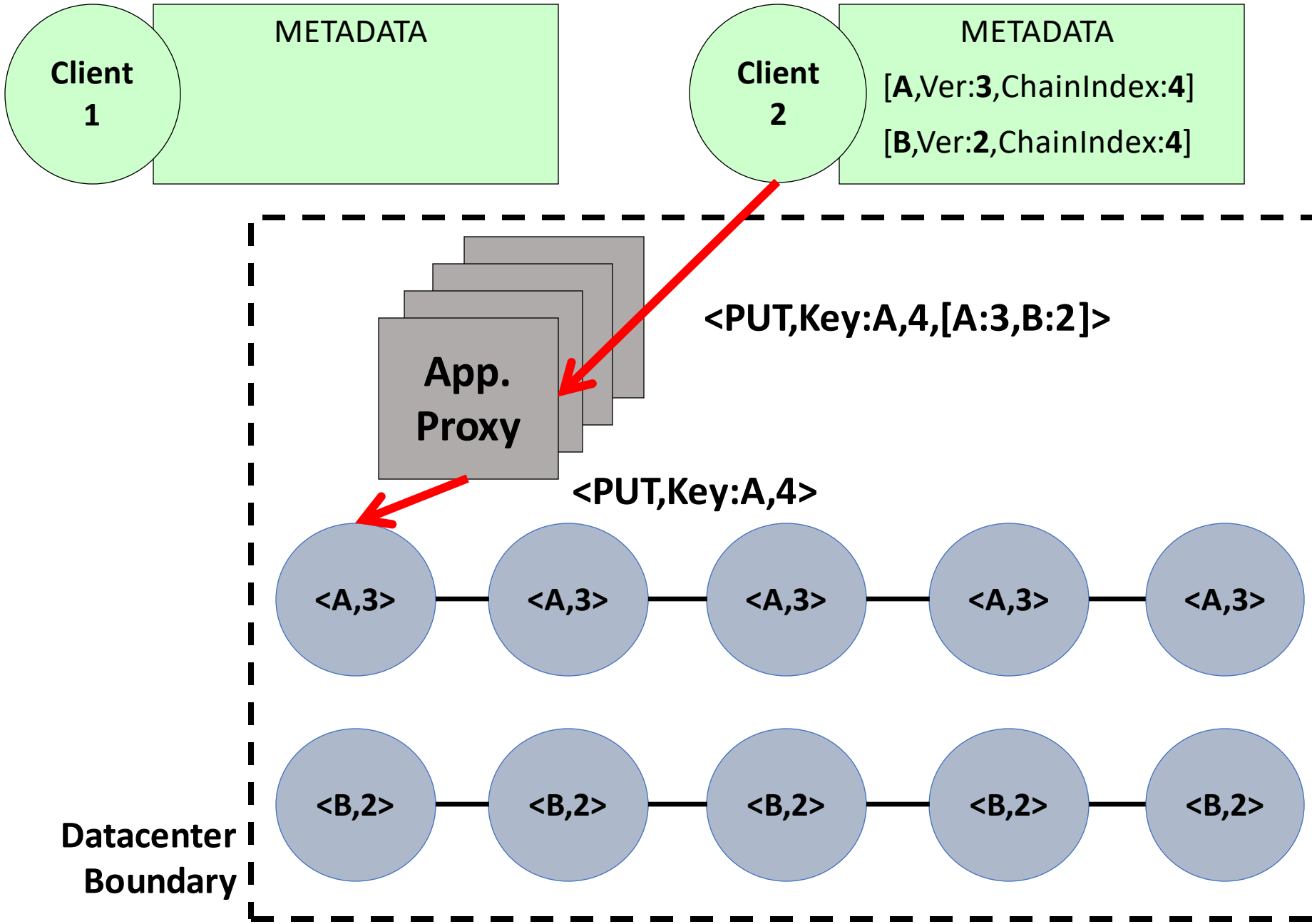

Client 1
METADATA
Client 2
METADATA
[A,Ver:3,ChainIndex:2]
[B,Ver:2,ChainIndex:1]
App. Proxy
<PUT,Key:A,4,[A:3,B:2]>
<A,3>
<A,3>
<A,3>
<A,3>
<A,0>
<B,2>
<B,2>
<B,2>
<B,0>
<B,0>
Datacenter Boundary

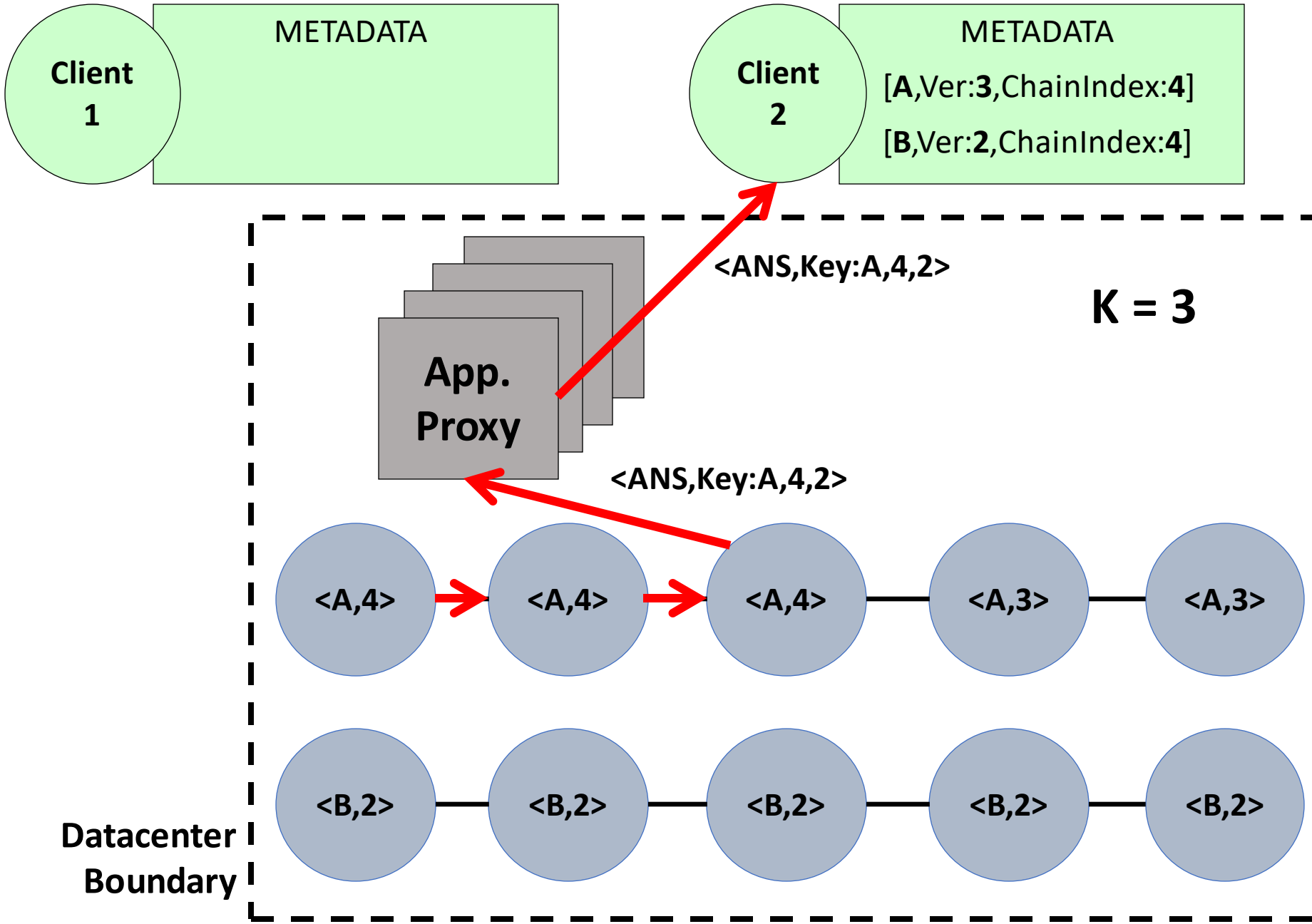

Client 1
METADATA
Client 2
METADATA
[A,Ver:3,ChainIndex:2]
[B,Ver:2,ChainIndex:1]
App. Proxy
<PUT,Key:A,4,[A:3,B:2]>
<A,3>
<A,3>
<A,3>
<A,3>
<A,0>
<B,2>
<B,2>
<B,2>
<B,0>
<B,0>
Datacenter Boundary

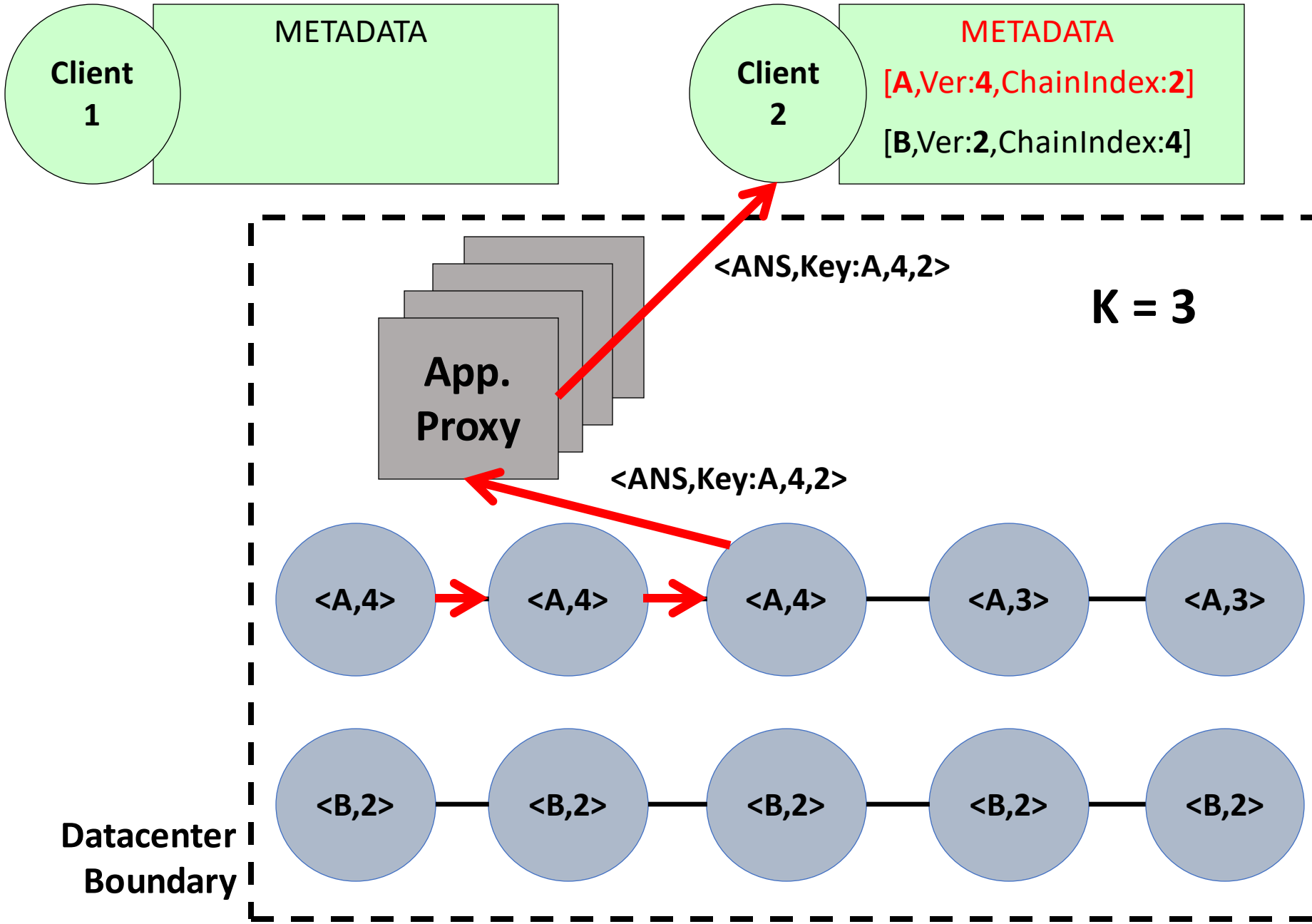

Client 1
METADATA
Client 2
METADATA
[A,Ver:3,ChainIndex:2]
[B,Ver:2,ChainIndex:1]
App. Proxy
<PUT,Key:A,4,[A:3,B:2]>
Stabilization Procedure
<CHECK,Key:A,3>
<A,3>
<A,3>
<A,3>
<A,3>
<A,0>
<CHECK,Key:B,2>
<B,2>
<B,2>
<B,2>
<B,0>
<B,0>
Datacenter Boundary


Client 1
METADATA
Client 2
METADATA
[A,Ver:3,ChainIndex:2]
[B,Ver:2,ChainIndex:1]
App. Proxy
<PUT,Key:A,4,[A:3,B:2]>
Stabilization Procedure
<CHECK,Key:A,3>
<A,3>
<A,3>
<A,3>
<A,3>
<A,3>
<CHECK,Key:B,2>
<B,2>
<B,2>
<B,2>
<B,2>
<B,2>
Datacenter Boundary

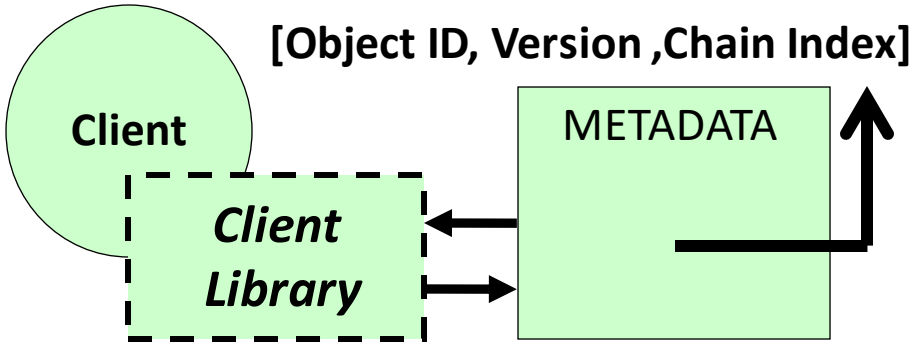

Client 1
METADATA
Client 2
METADATA
[A,Ver:3,ChainIndex:4]
[B,Ver:2,ChainIndex:4]
Stabilization Procedure
App. Proxy
<PUT,Key:A,4,[A:3,B:2]>
<OK,Key:A,3>
<A,3>
<A,3>
<A,3>
<A,3>
<A,3>
<OK,Key:B,2>
<B,2>
<B,2>
<B,2>
<B,2>
<B,2>
Datacenter Boundary

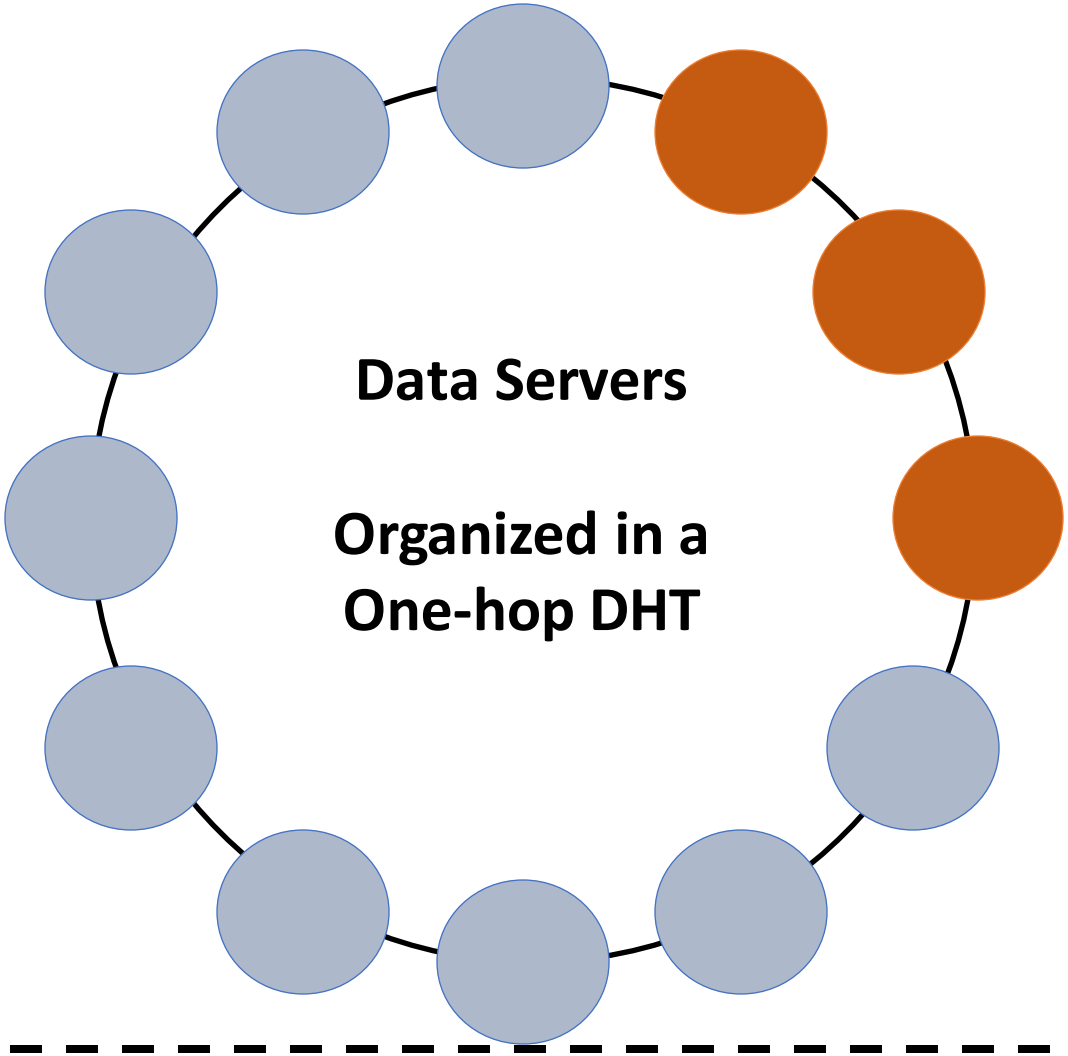

Client 1
METADATA
Client 2
METADATA
[A,Ver:3,ChainIndex:4]
[B,Ver:2,ChainIndex:4]
App. Proxy
<PUT,Key:A,4,[A:3,B:2]>
<PUT,Key:A,4>
<A,3>
<A,3>
<A,3>
<A,3>
<A,3>
<B,2>
<B,2>
<B,2>
<B,2>
<B,2>
Datacenter Boundary

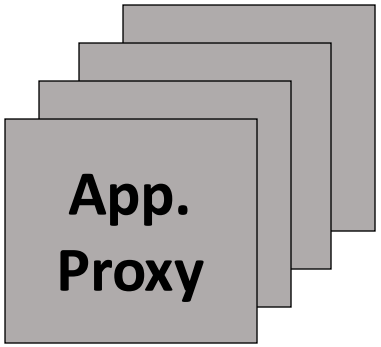

Client 1
METADATA
Client 2
METADATA
[A,Ver:3,ChainIndex:4]
[B,Ver:2,ChainIndex:4]
<ANS,Key:A,4,2>
K = 3
App. Proxy
<ANS,Key:A,4,2>
<A,4>
<A,4>
<A,4>
<A,3>
<A,3>
<B,2>
<B,2>
<B,2>
<B,2>
<B,2>
Datacenter Boundary


Client 1
METADATA
Client 2
METADATA
[A,Ver:4,ChainIndex:2]
[B,Ver:2,ChainIndex:4]
<ANS,Key:A,4,2>
K = 3
App. Proxy
<ANS,Key:A,4,2>
<A,4>
<A,4>
<A,4>
<A,3>
<A,3>
<B,2>
<B,2>
<B,2>
<B,2>
<B,2>
Datacenter Boundary

# Extending the solution to Geo-replicated settings

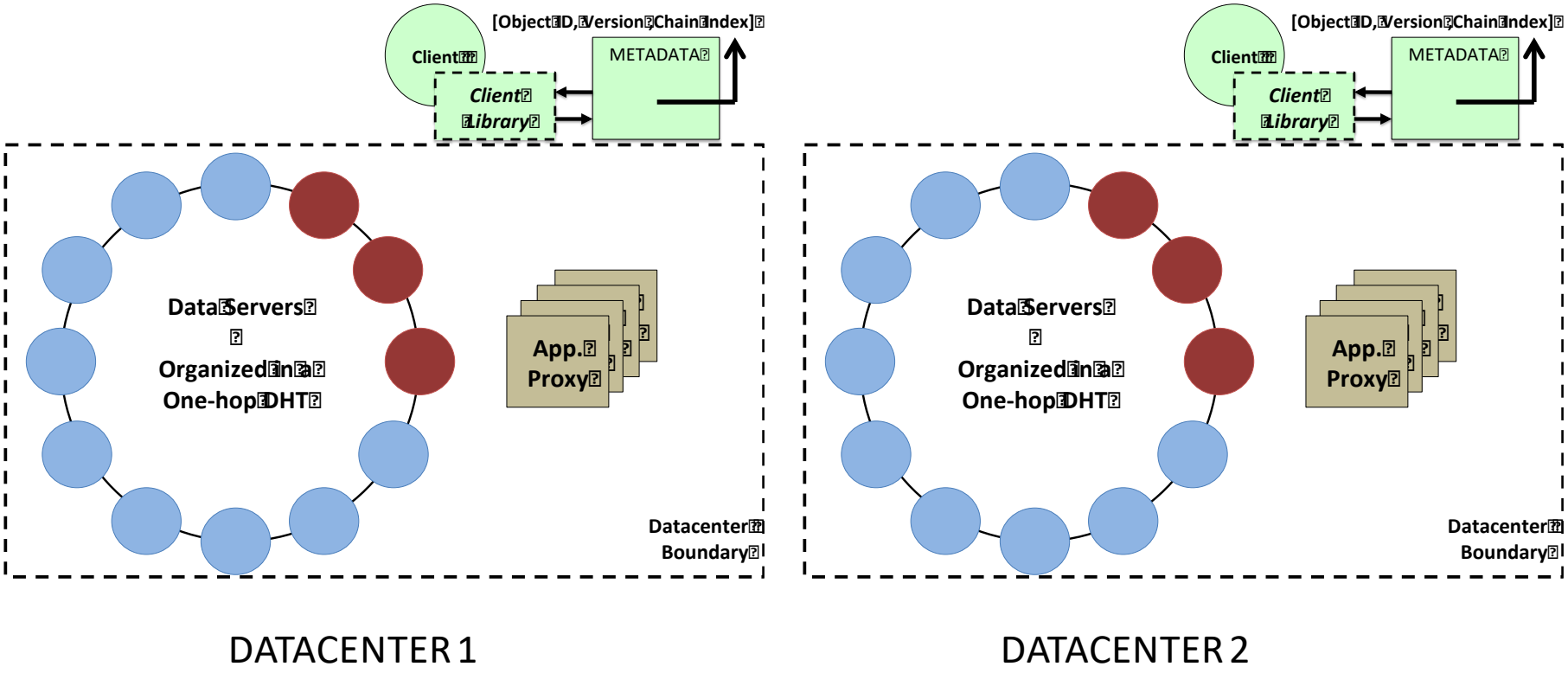



Datacenter Boundary

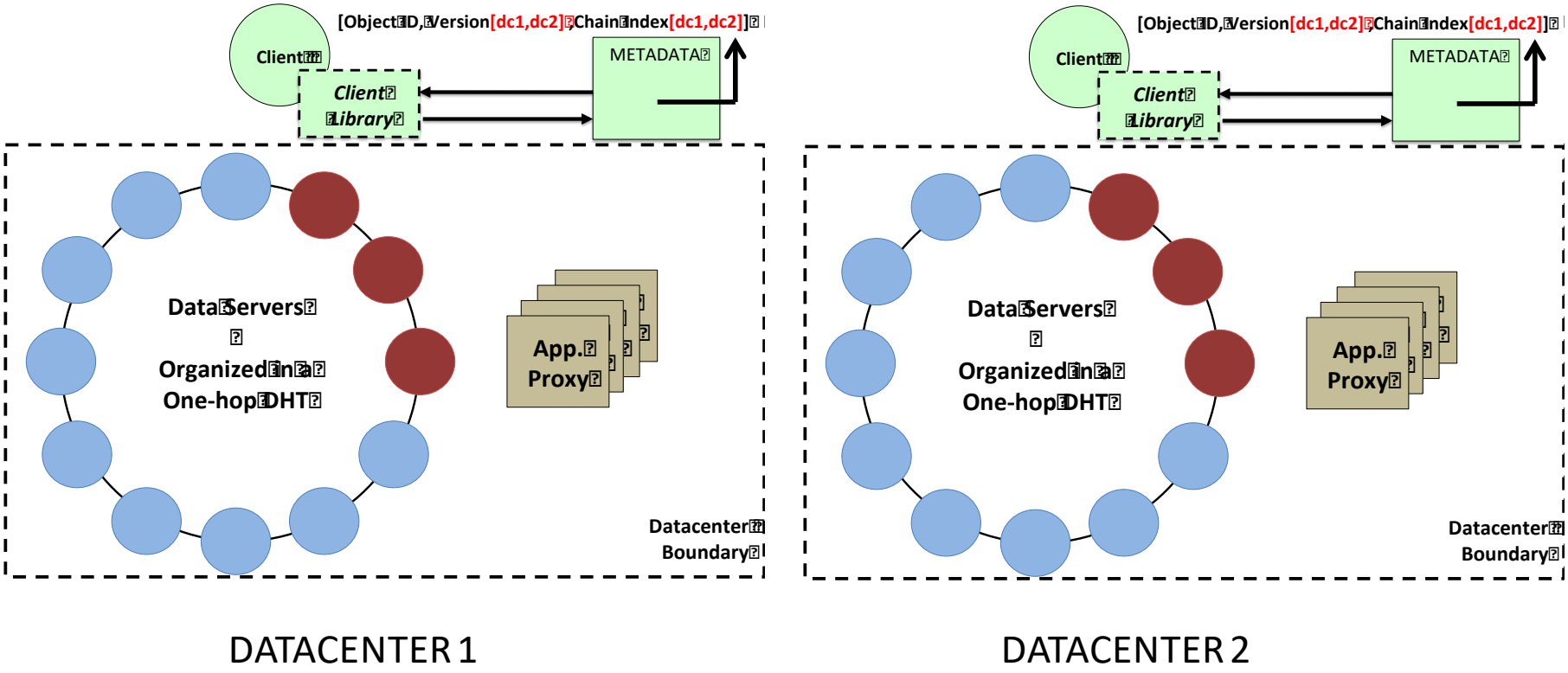
# Extending the solution to Geo-replicated settings




DATACENTER 1

DATACENTER 2

# Extending the solution to Geo-replicated settings




DATACENTER 1

DATACENTER 2

# Extending the solution to Geo-replicated settings

- Read operations:
  - Processed in a single datacenter (without any communication across datacenters).
- Write operations:
  - Return to the client when executed by K nodes in the local datacenter.
  - Propagated in background across datacenters (causality between batches of operations shipped to remote datacenters enforced with logical clocks).
  - Rely on last-writer-wins to merge divergent versions.

# The proof of correctness (intuition)

- In no data center can a write operation start to execute (i.e. *become observable*) without ensuring that all causal dependencies are **stable** in that DC
  - Impossible for a client to observe any version prior.

- When geo-replicating, we ensure that all operations are executed using the same logic and an order compatible with their execution on the original datacenter.

# Quiz!!1!

**What do you remember?**

# Question 1

**P1 : W(x)1 W(x)3**

**P2 : R(x)1 W(x)2**

**P3 : R(x)2 R(x)2 R(x)1**

**P4 : R(x)1 R(x)2 R(x)3**

This execution violates:

- Monotonic Reads
- Monotonic Writes
- Writes Follow Reads
- Read your writes

# Answer 1

**P1 : W(x)1 W(x)3**

**P2 : R(x)1 W(x)2**

**P3 : R(x)2 R(x)2 R(x)1**

**P4 : R(x)1 R(x)2 R(x)3**

This execution violates:

- Monotonic Reads
- Monotonic Writes
- ***Writes Follow Reads*: see P3**
- Read your writes

# Question 2

Causal consistency offers a stronger weak consistency framework than eventual consistency

- True
- False

# Answer 2

Causal consistency offers a stronger weak consistency framework than eventual consistency

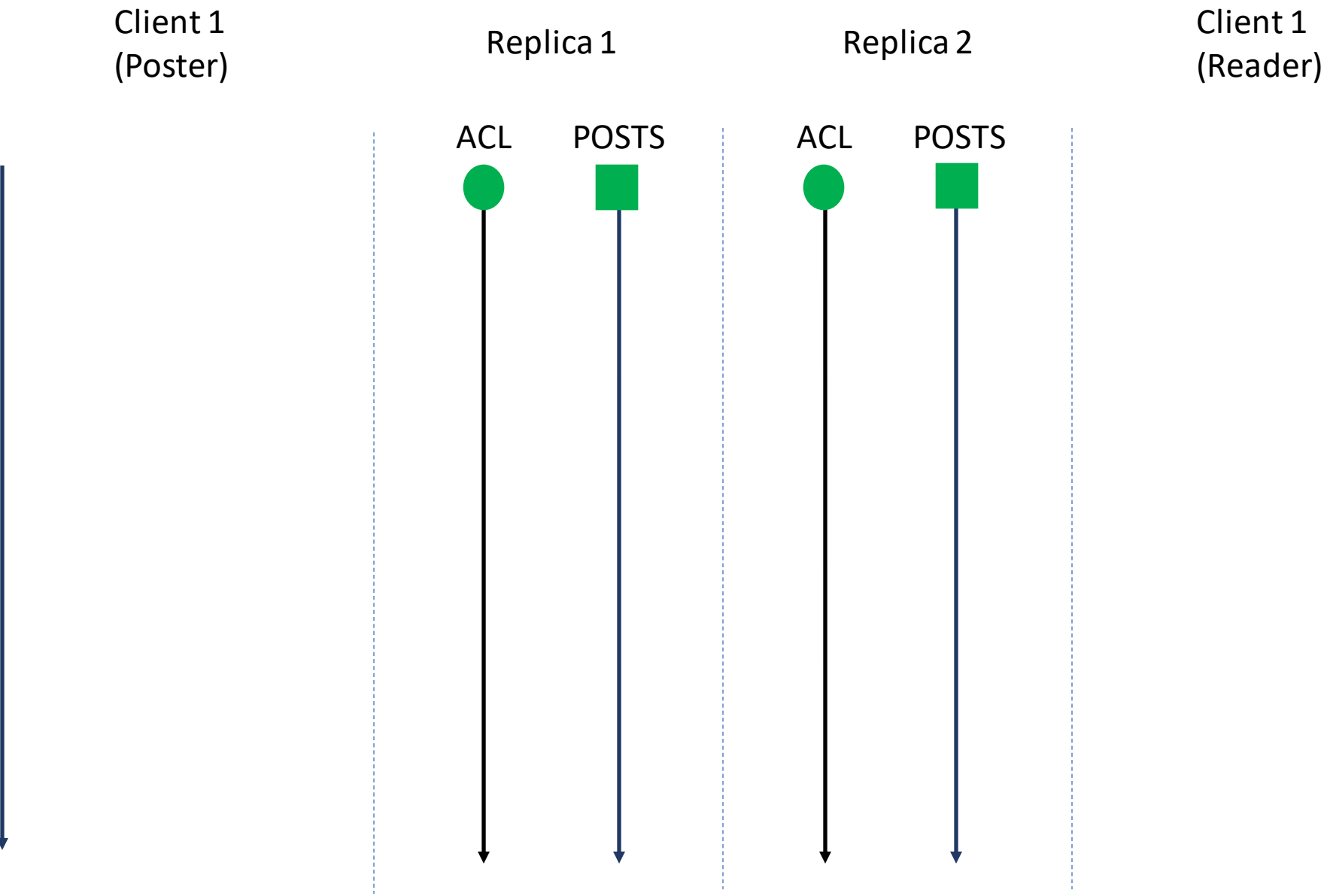
- True
- ***False: eventual convergence is covered in eventual consistency, but not in causal consistency. Causal+ is stronger.***

# Going back to the social-network example

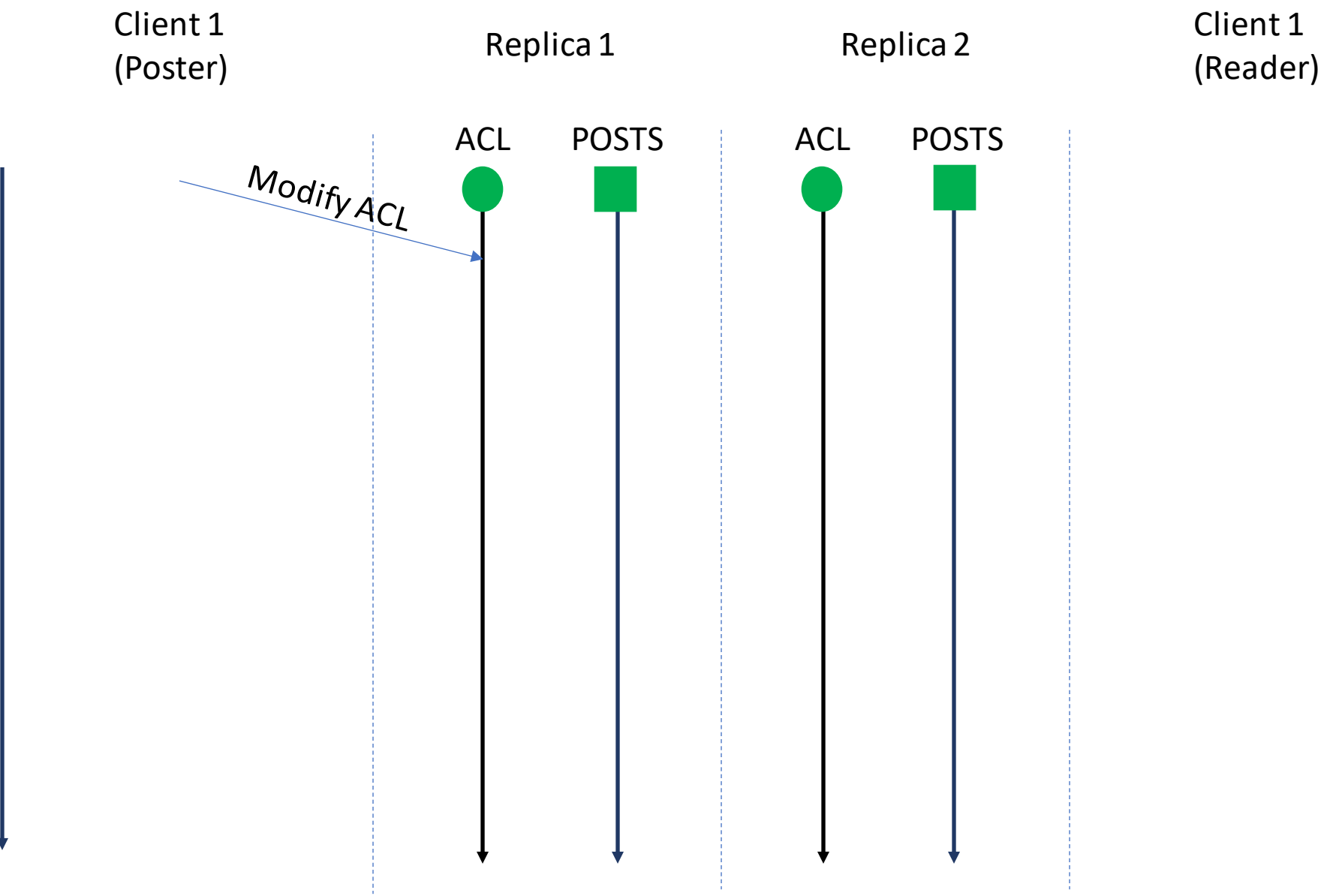
- Can causality solve the problem with the access control list (ACL) and the photo that we saw earlier in the lecture?

  - Let's assume we execute the operation to read a user wall using the same strategy I described before:
  - First: Check the contents of the ACL.
  - Second: If user reading is in the ACL, read and return contents of the user wall.

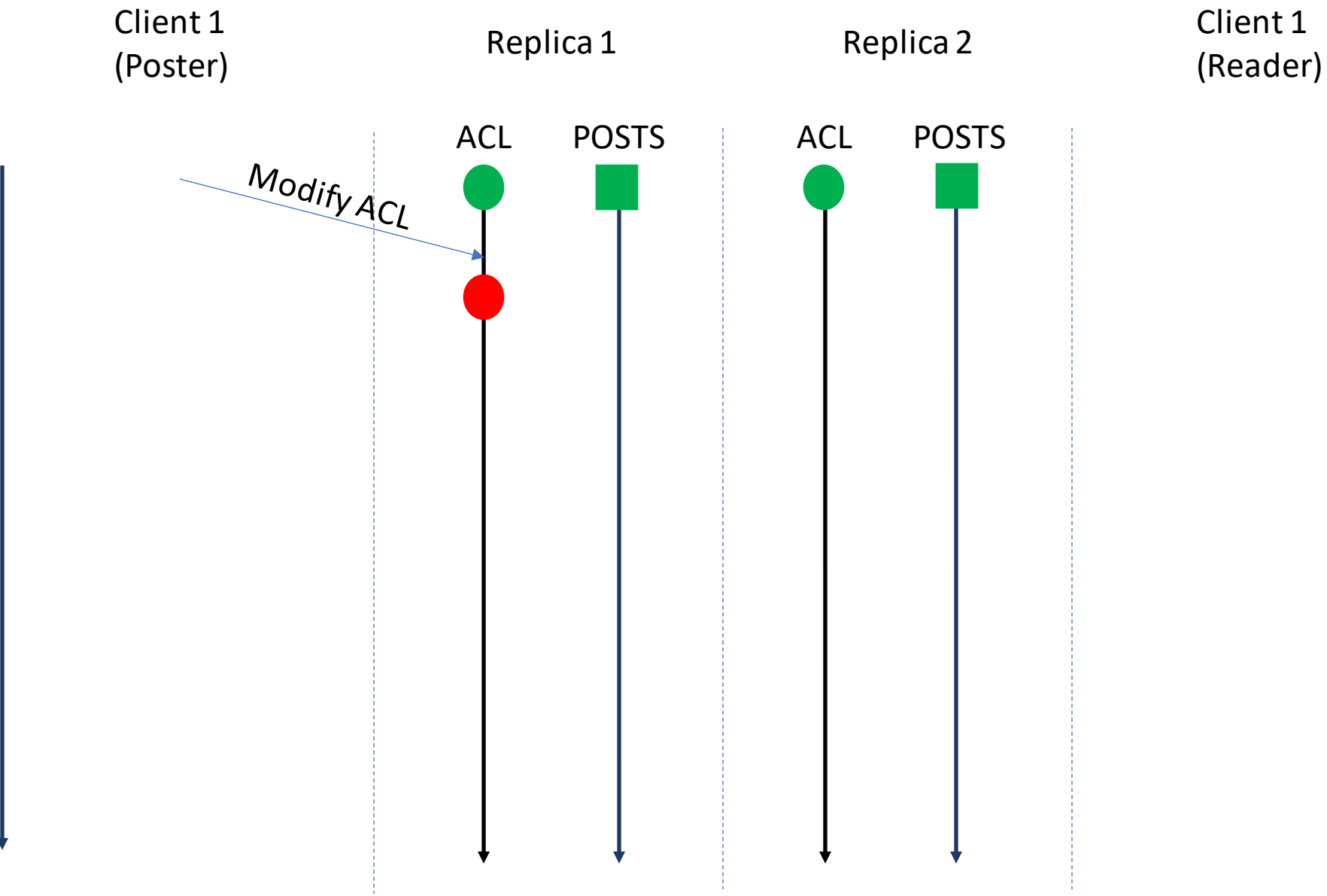
# How can this go Wrong:



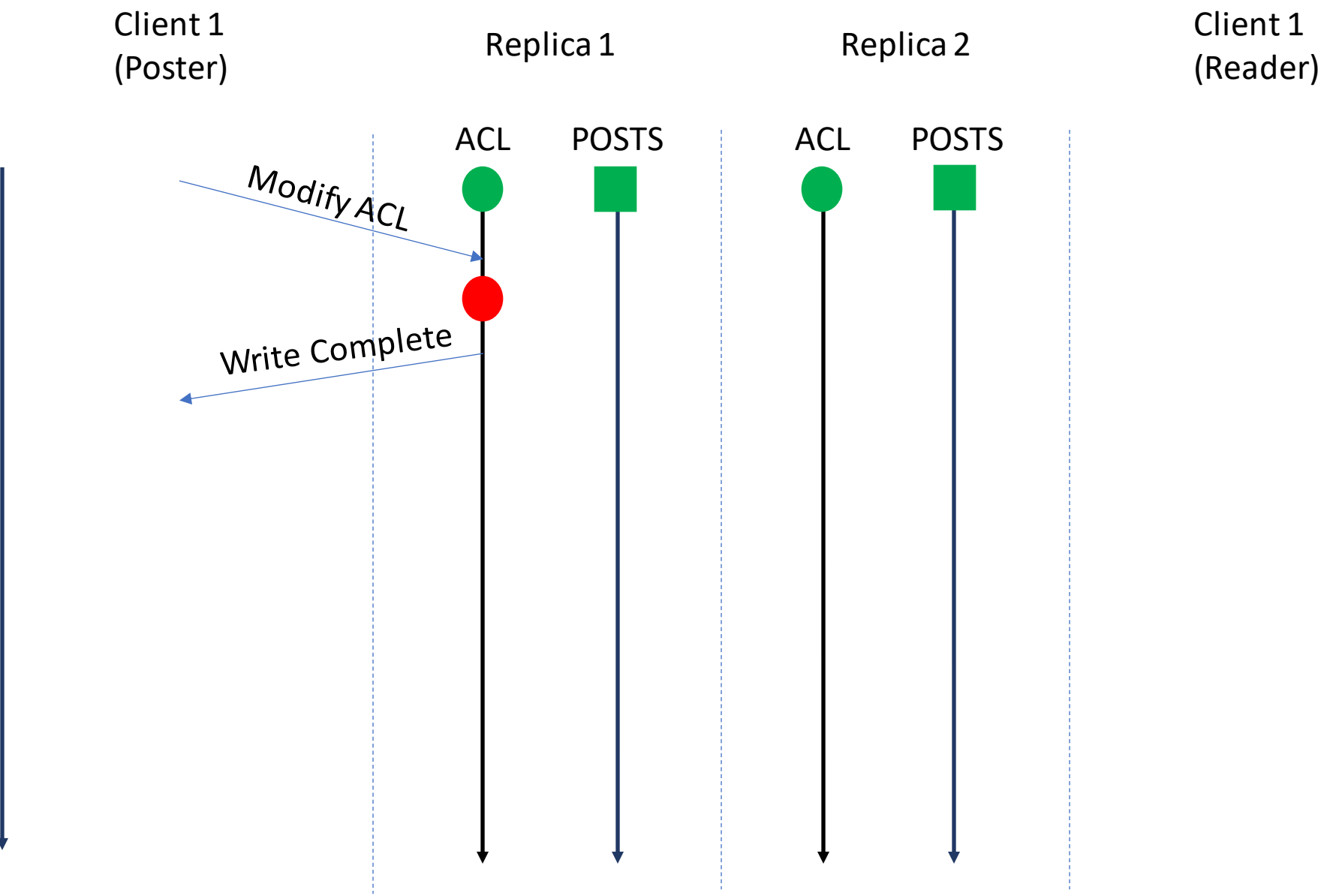
# How can this go Wrong:



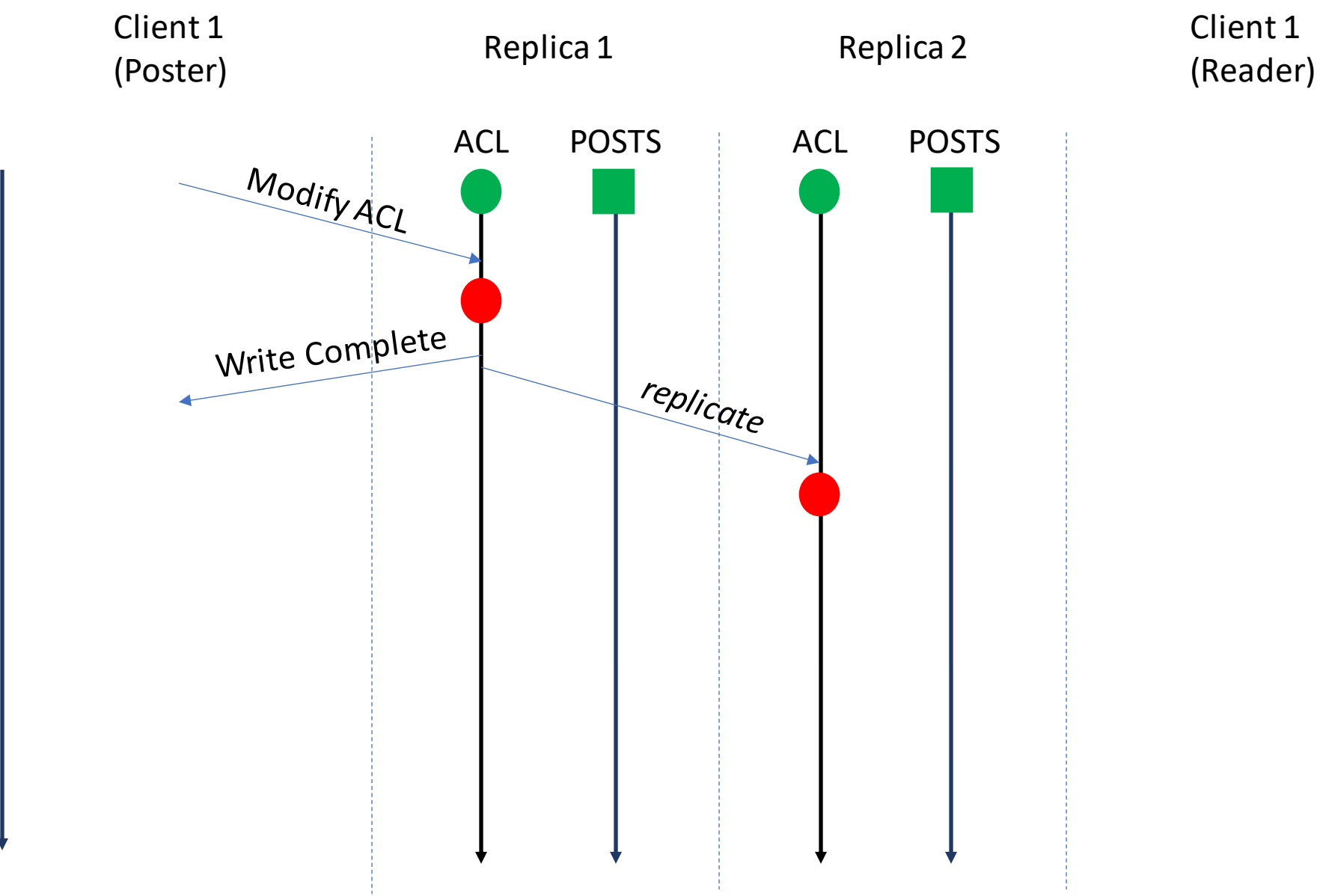
# How can this go Wrong:



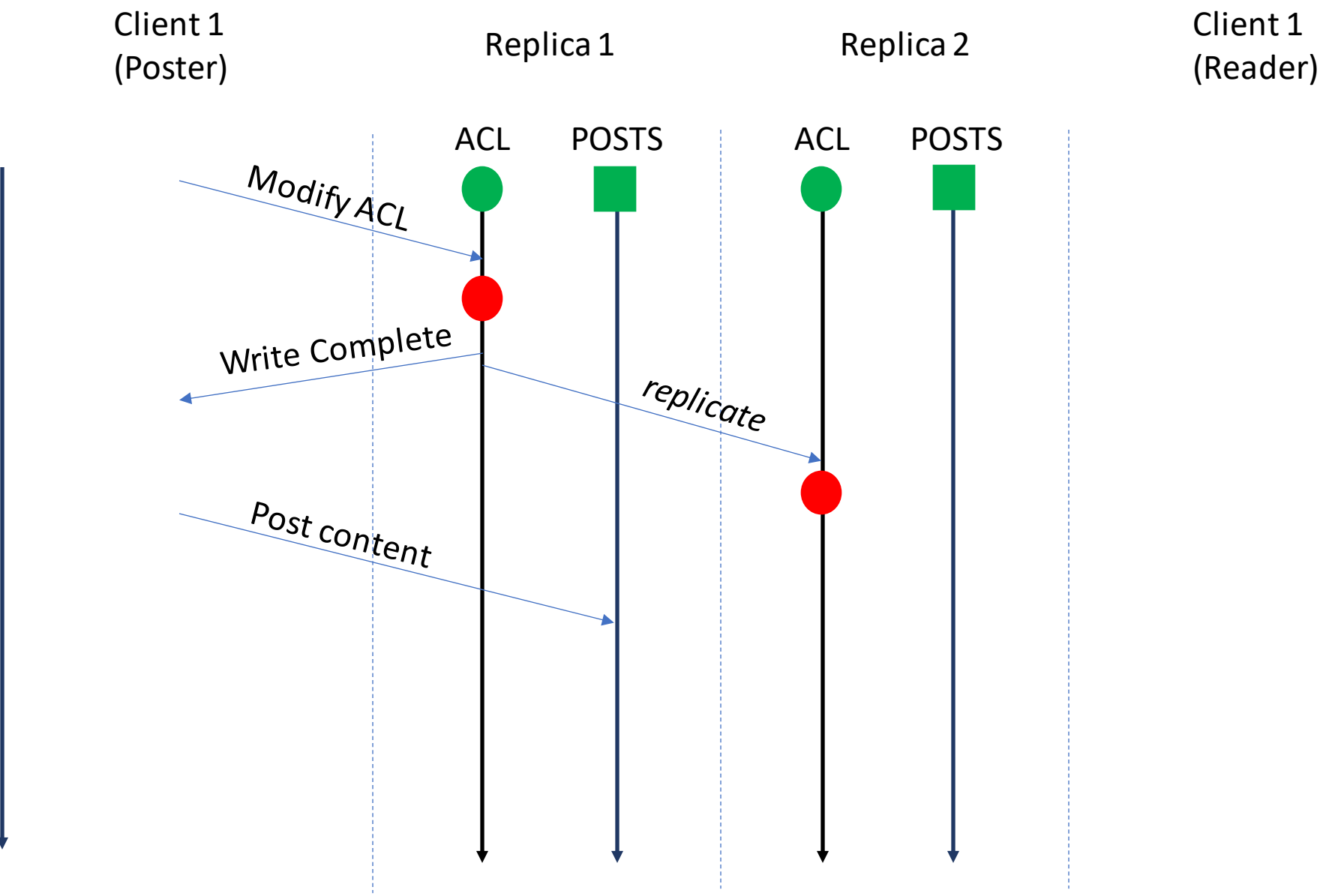
# How can this go Wrong:



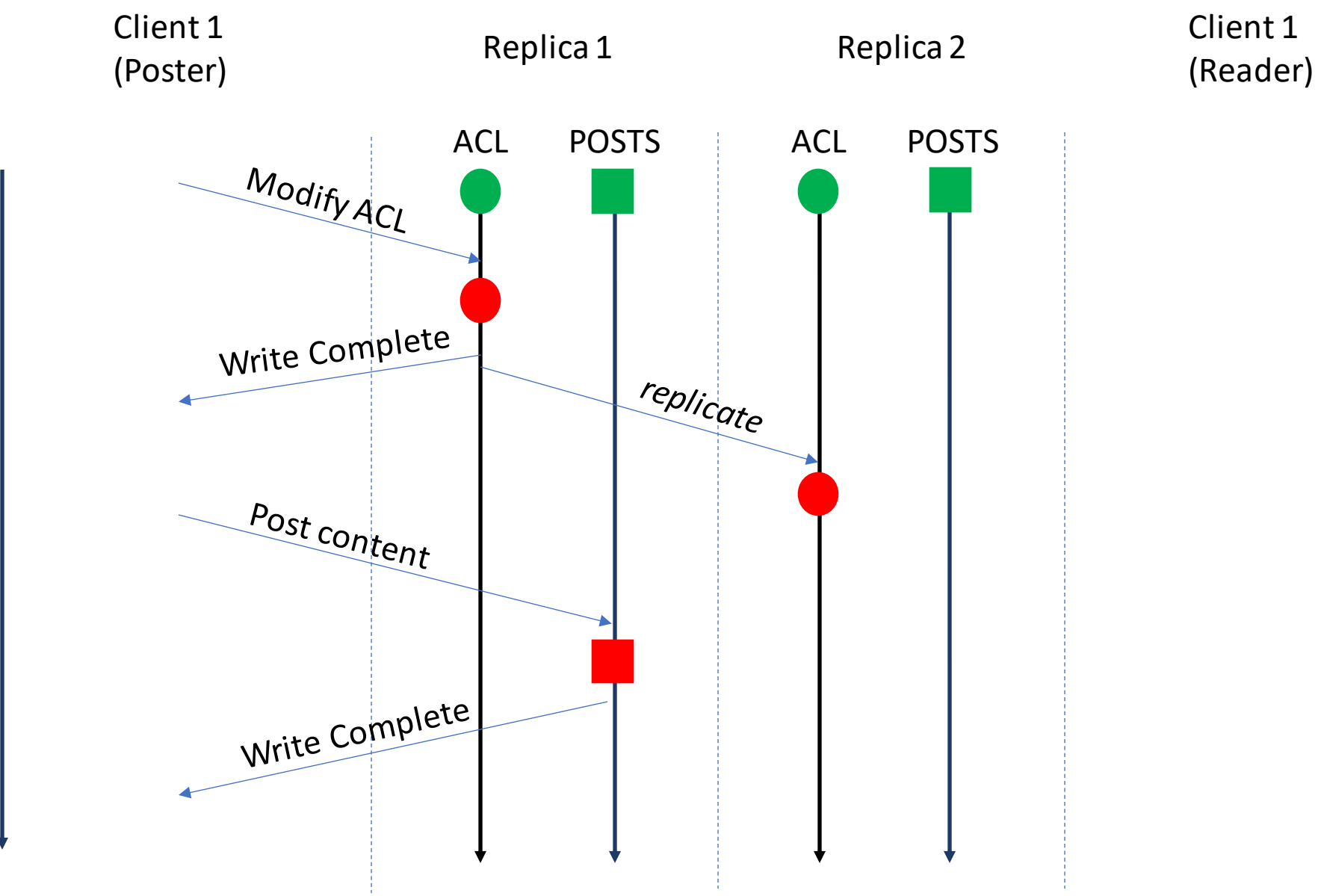
# How can this go Wrong:



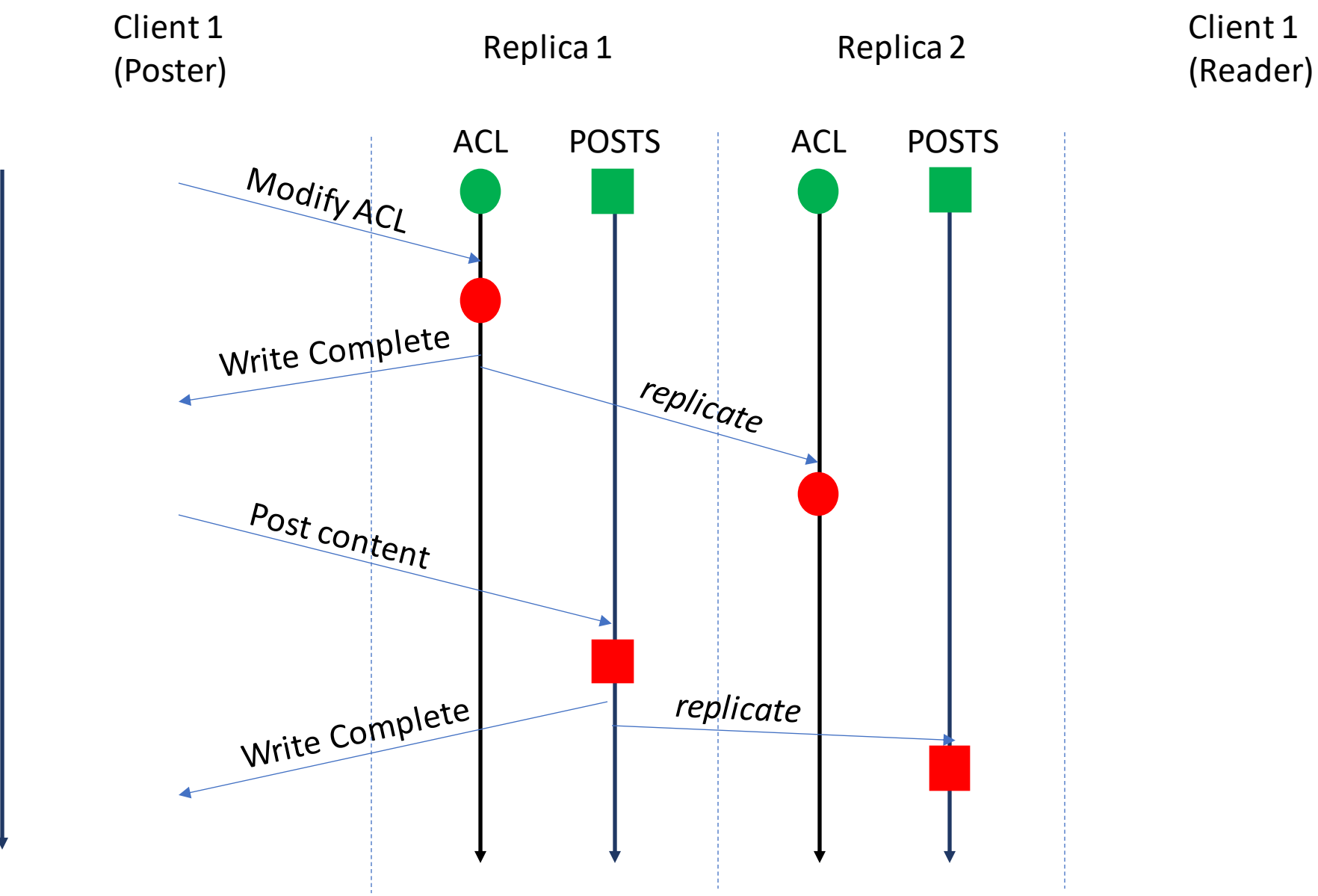
# How can this go Wrong:



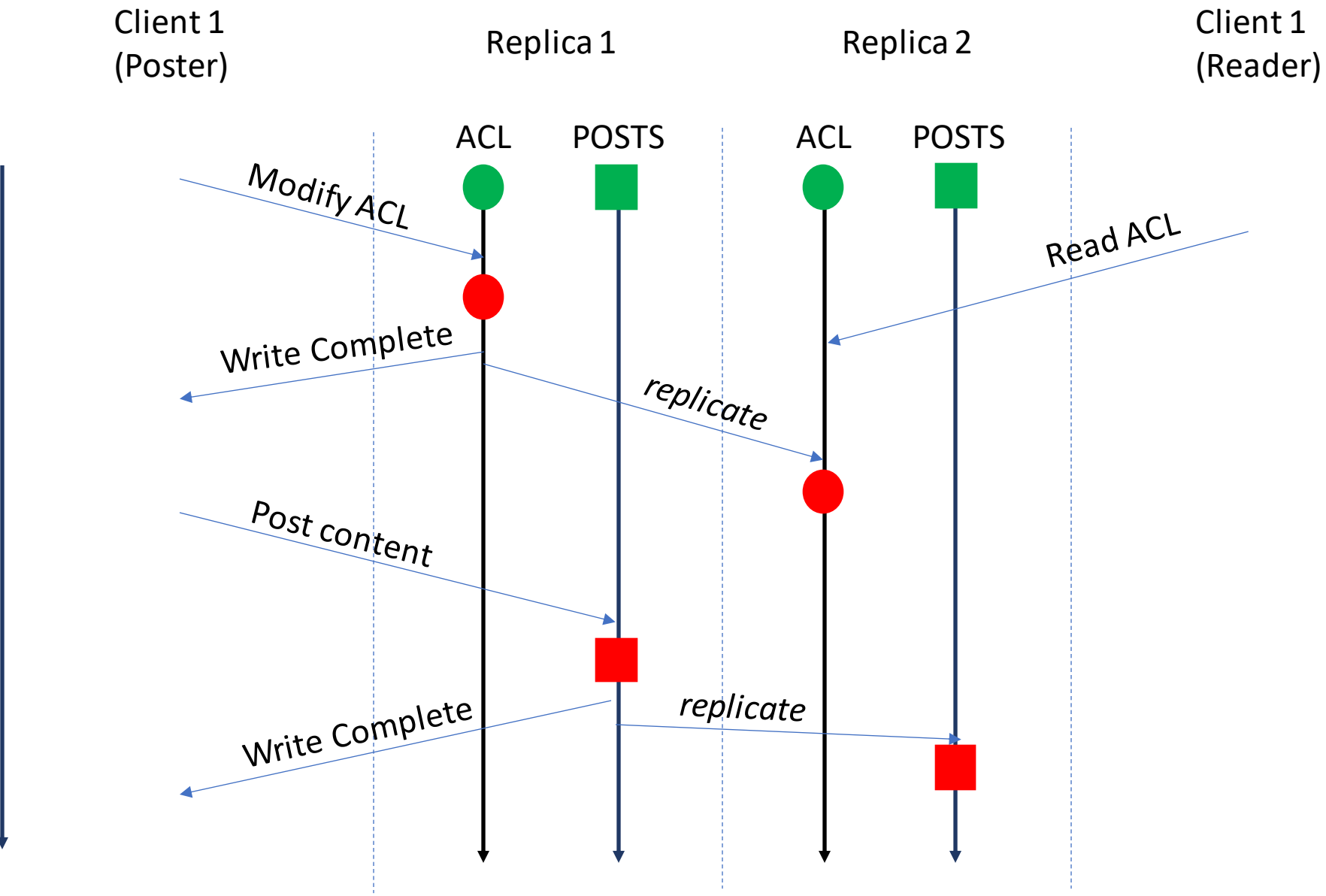
# How can this go Wrong:



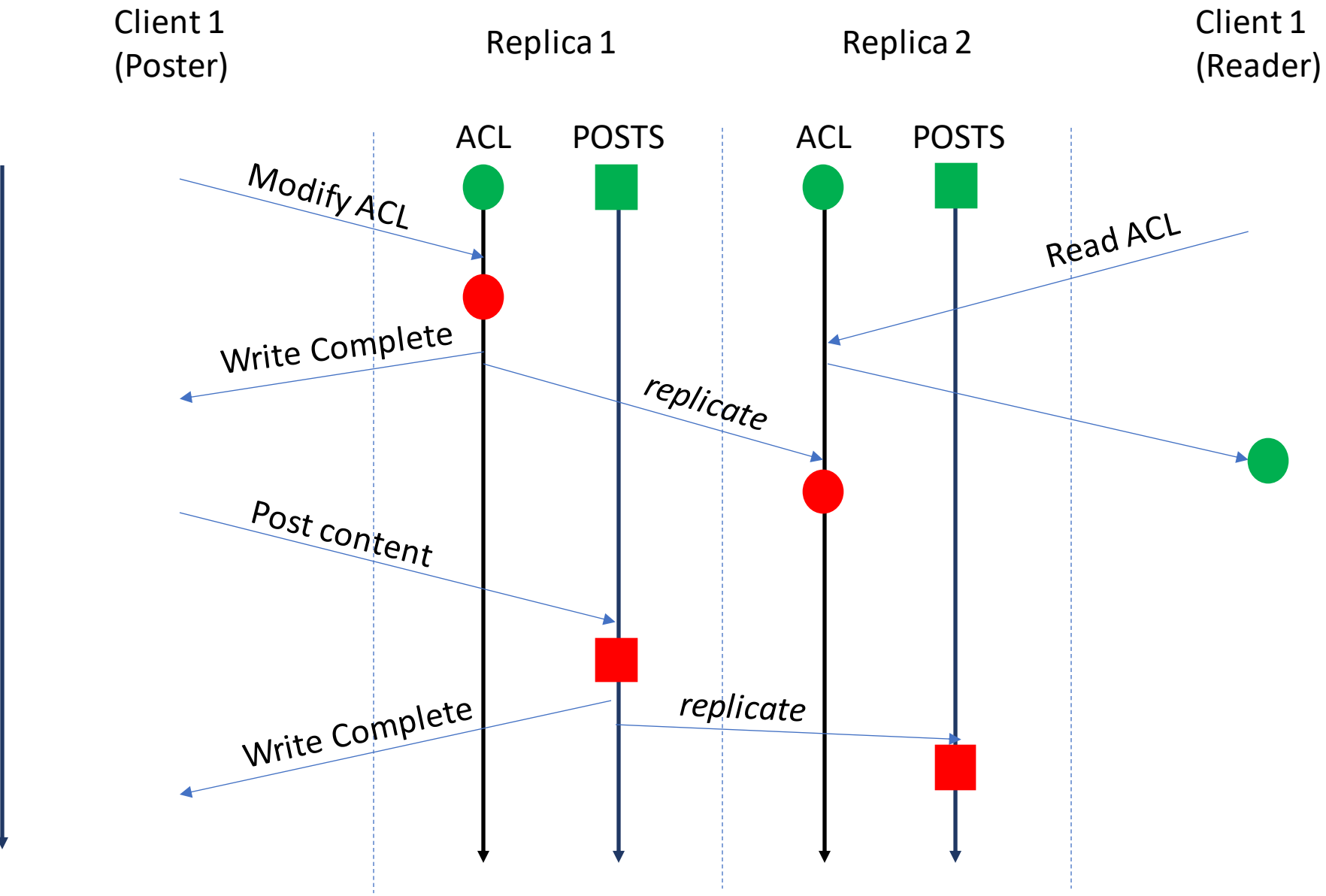
# How can this go Wrong:



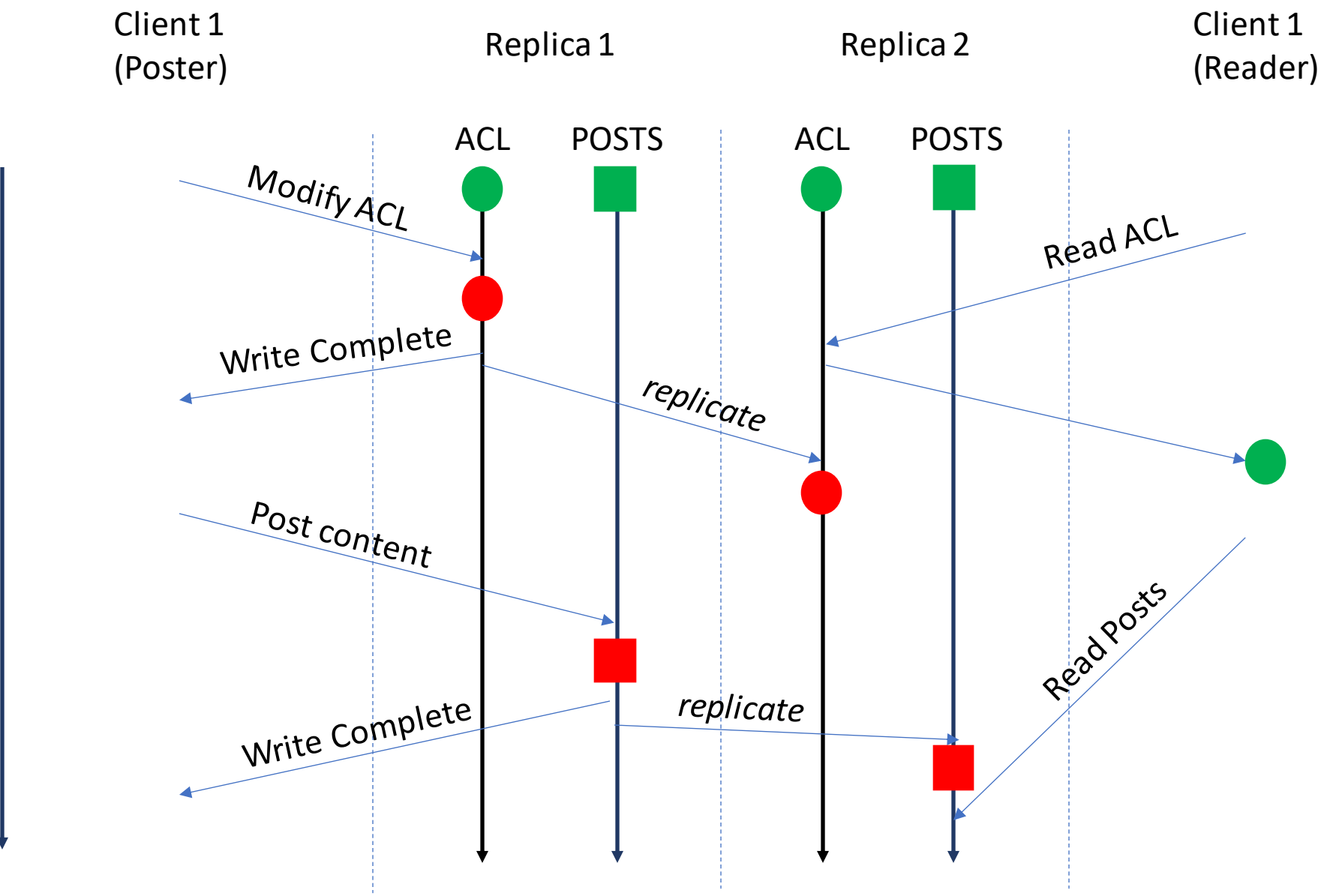
# How can this go Wrong:



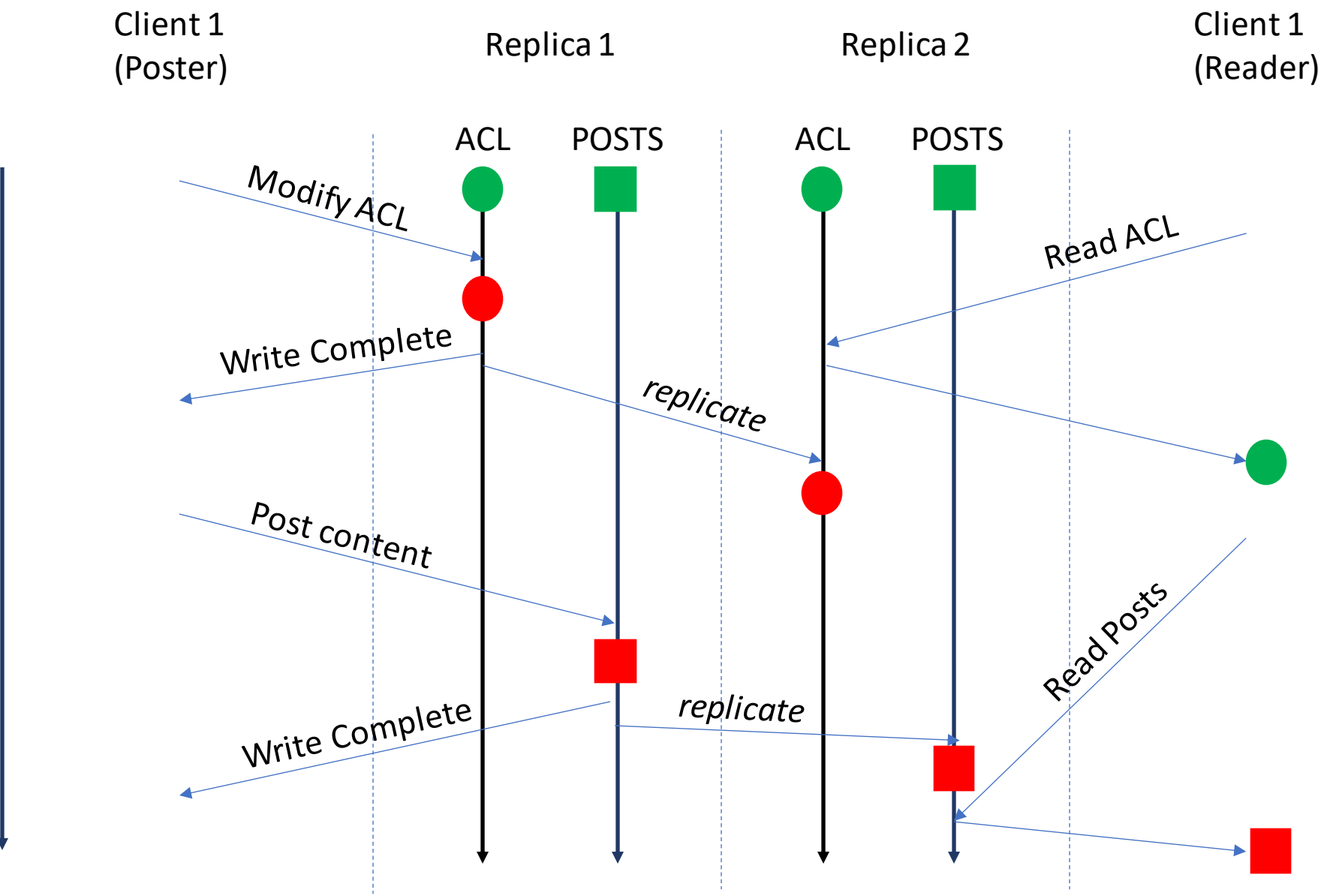
# How can this go Wrong:



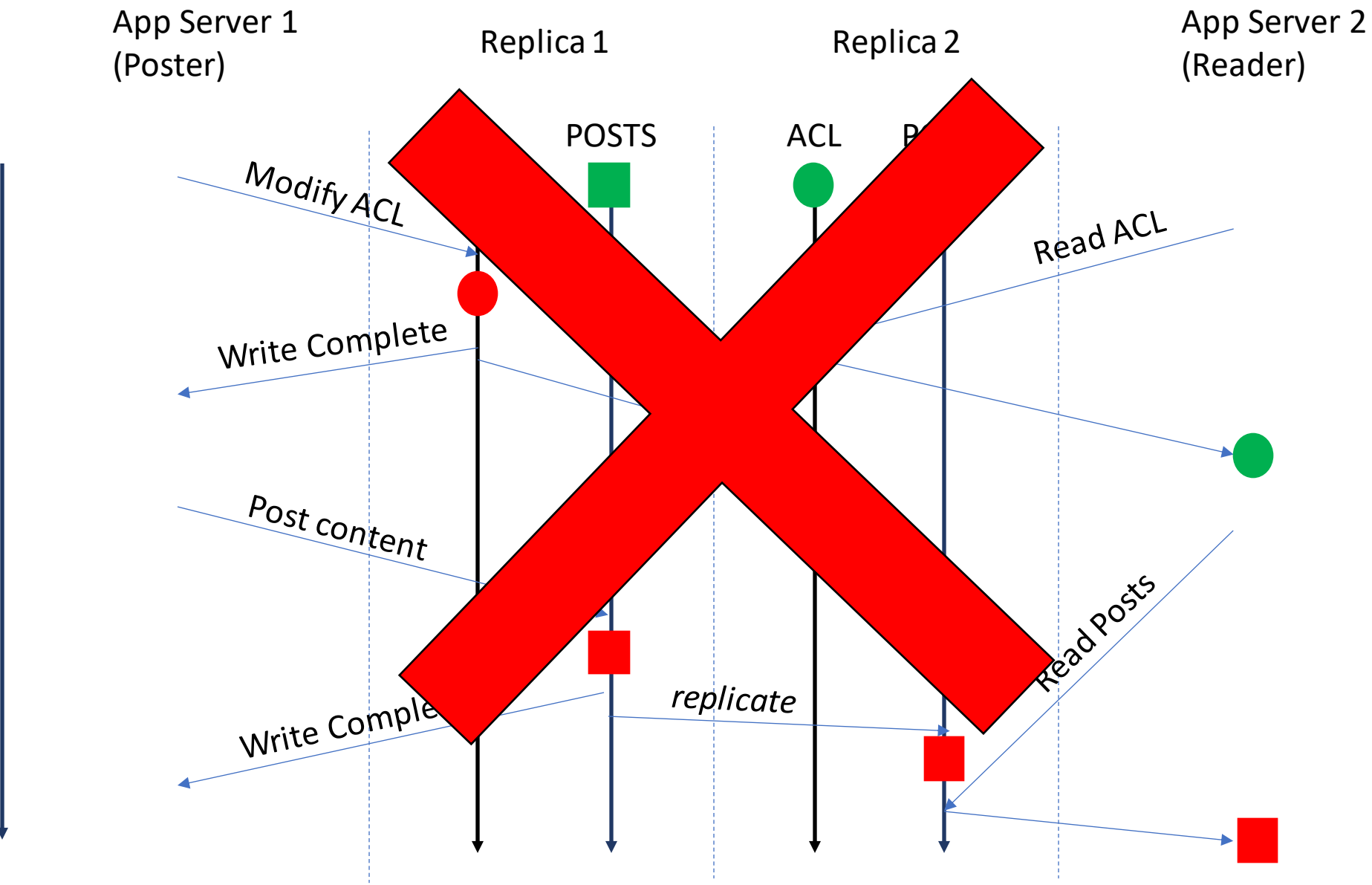
# How can this go Wrong:



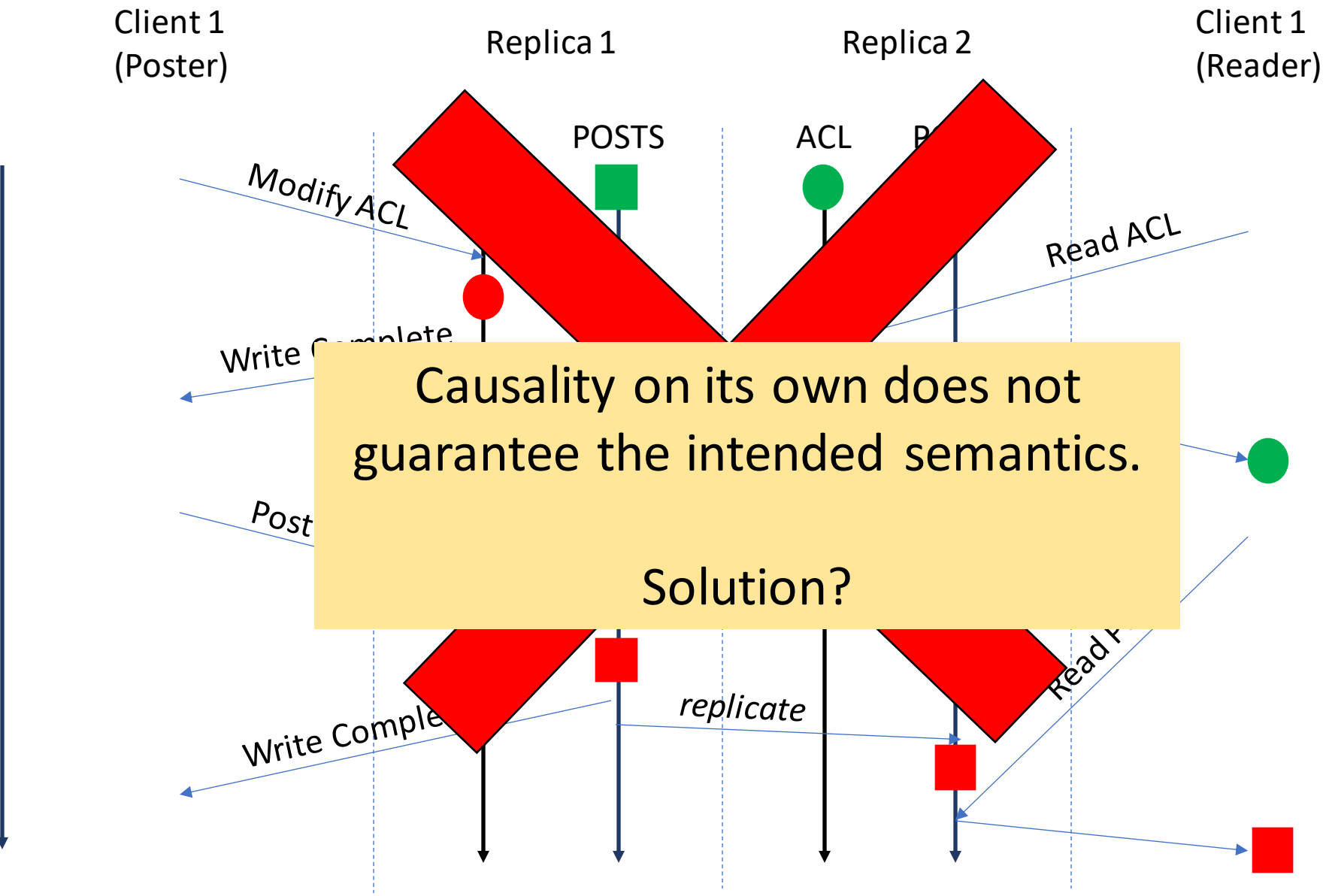
# How can this go Wrong:



# How can this go Wrong:



# How can this go Wrong:



# Why does this not work?

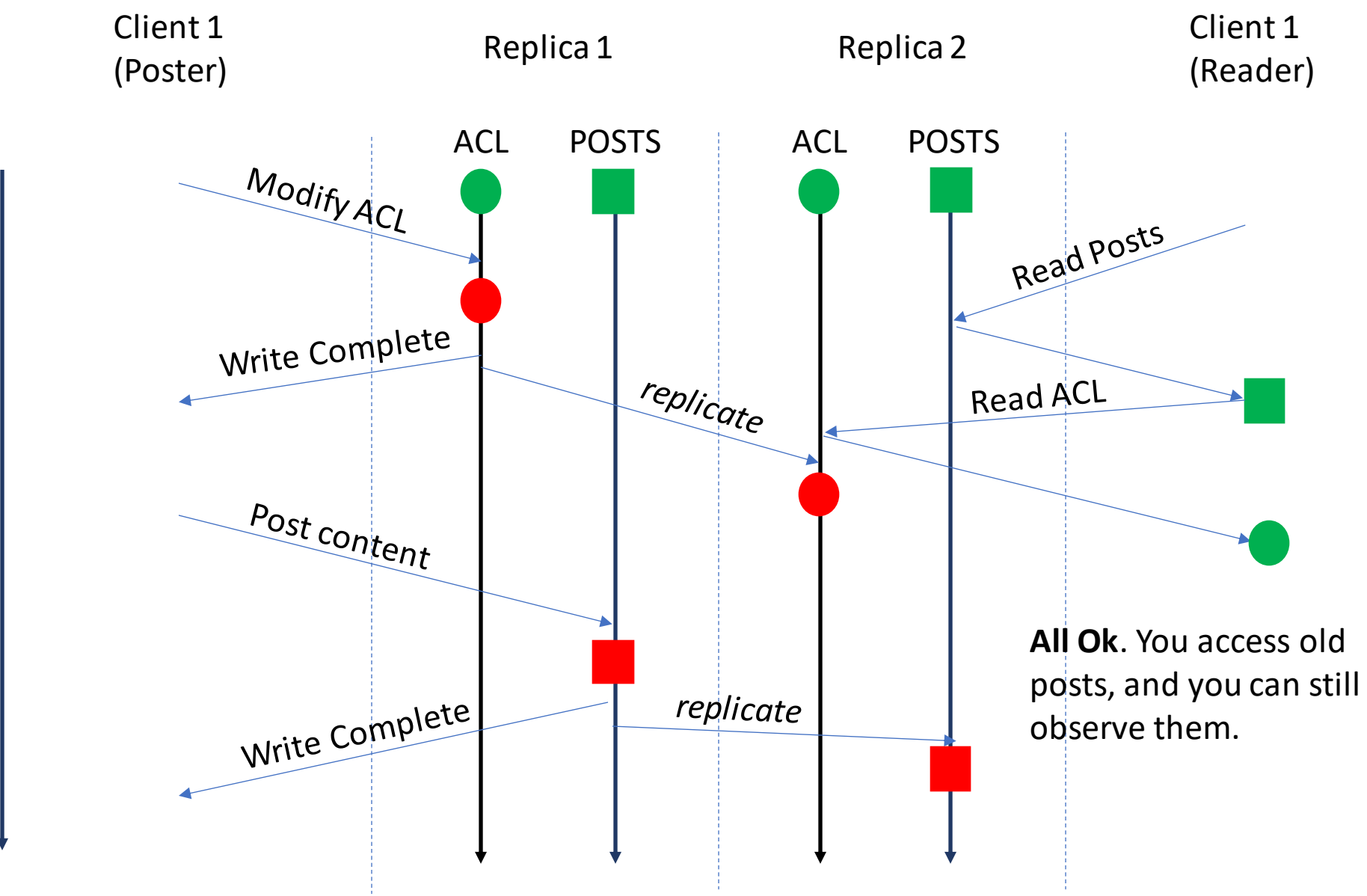
- We can still observe observe the old ACL and then observe new post in the wall of the user.

- No cau
  - Rea
  - Mo
  - Mo
  - Wri

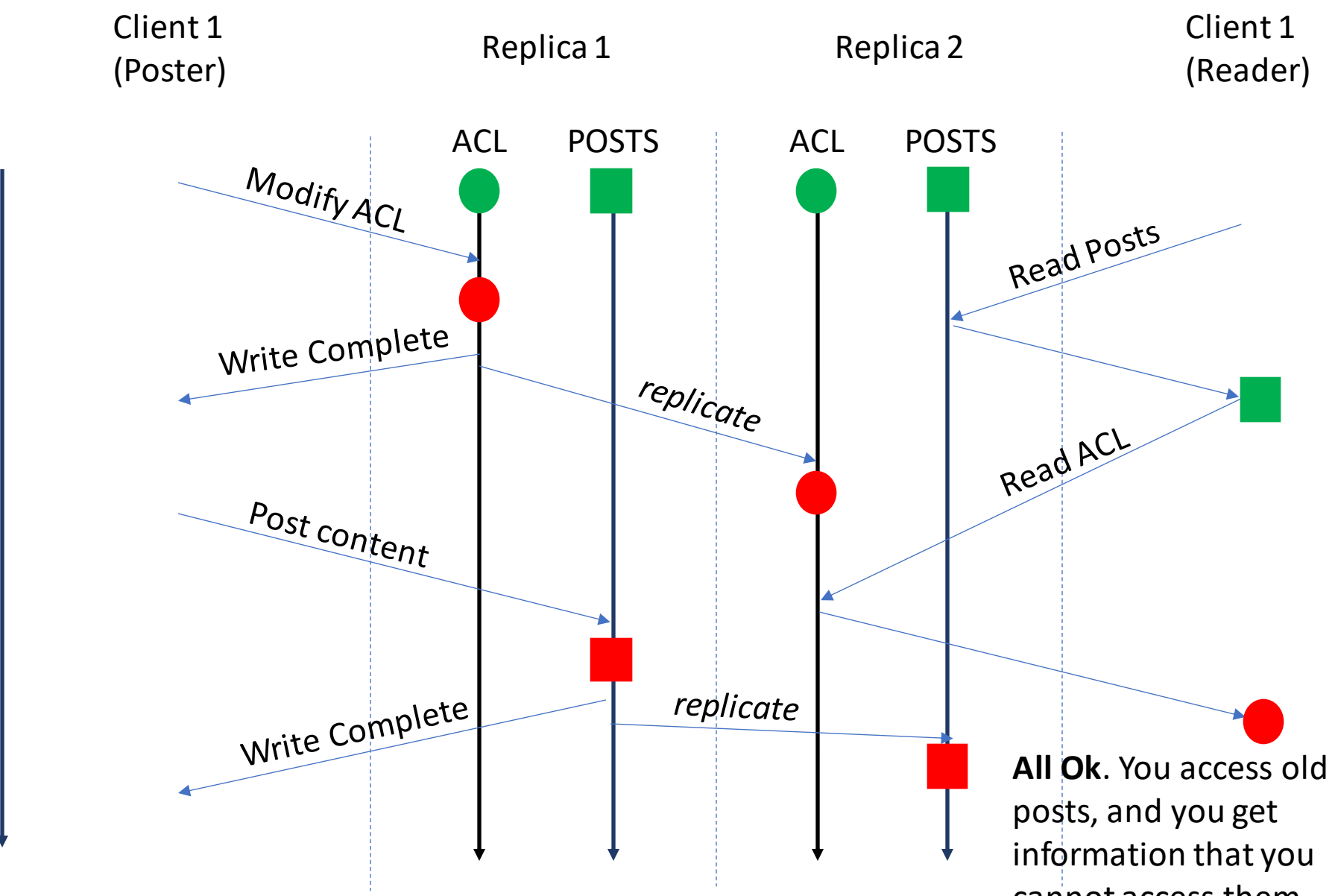
When executing the operation, we must first read the
Posts
and only after read the
ACL

- How can we solve this?

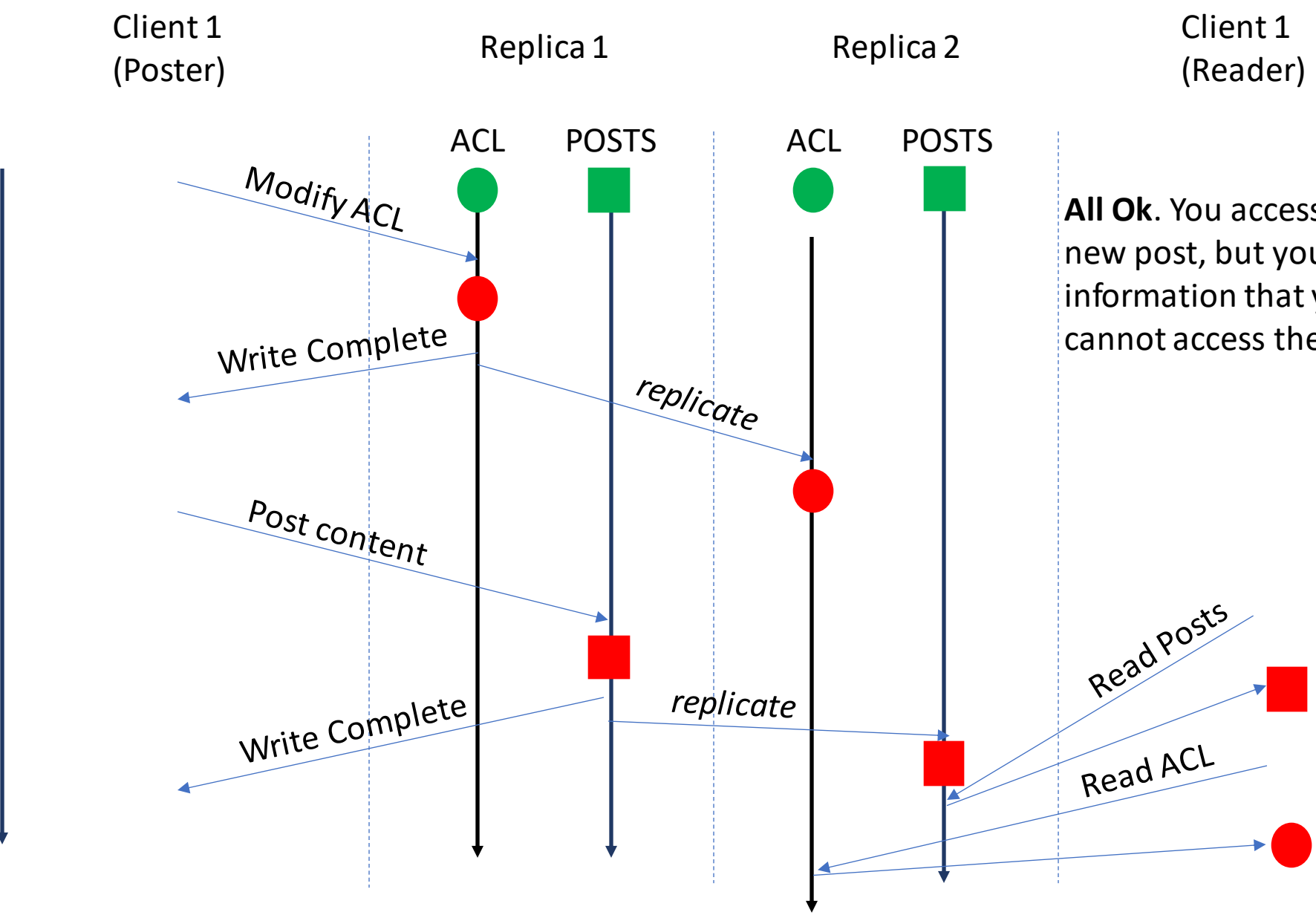
# How can this not go Wrong (Take1):



# How can this not go Wrong (Take2):

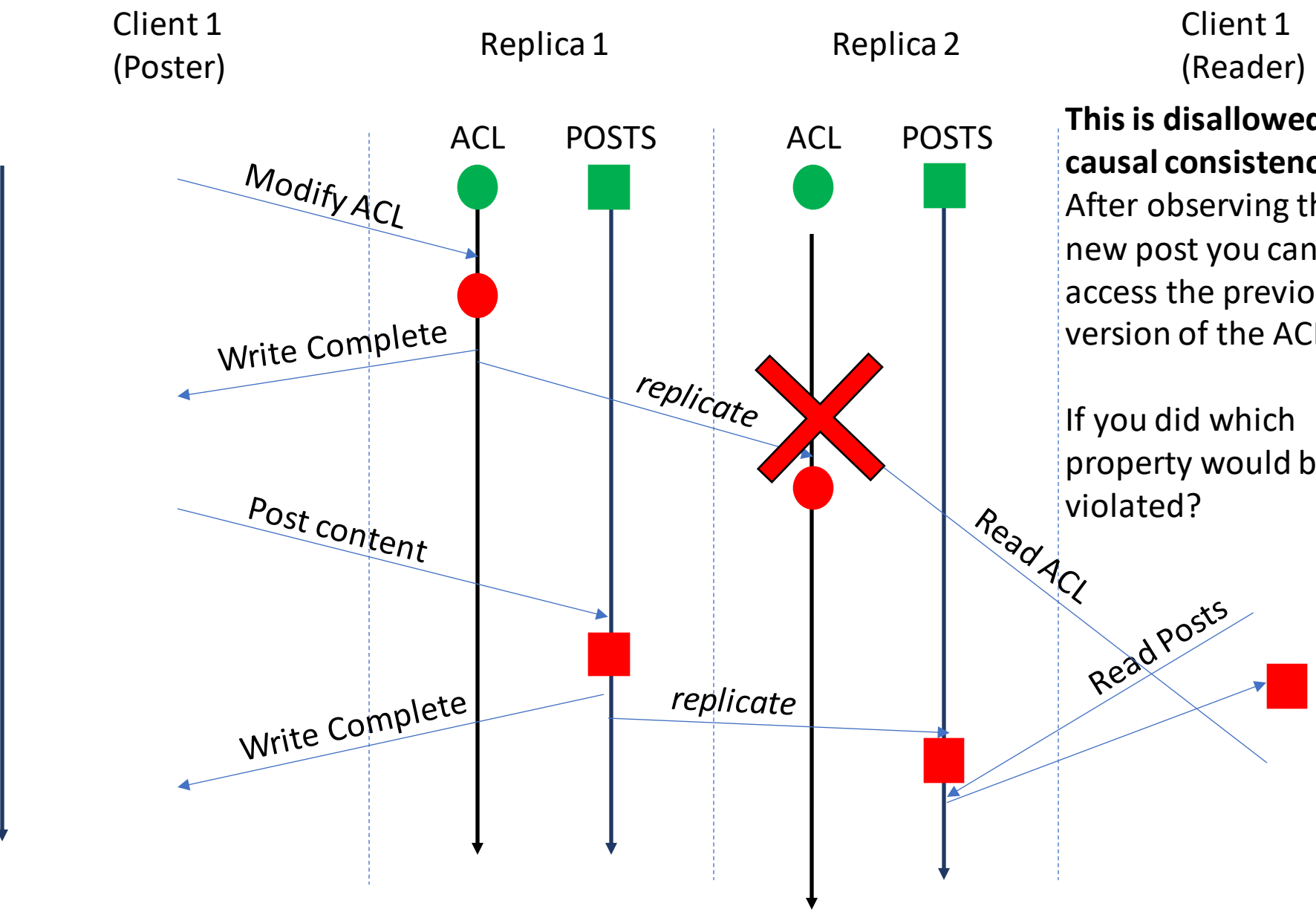


# How can this not go Wrong (Take3):




**All Ok**. You access the new post, but you get information that you cannot access them.

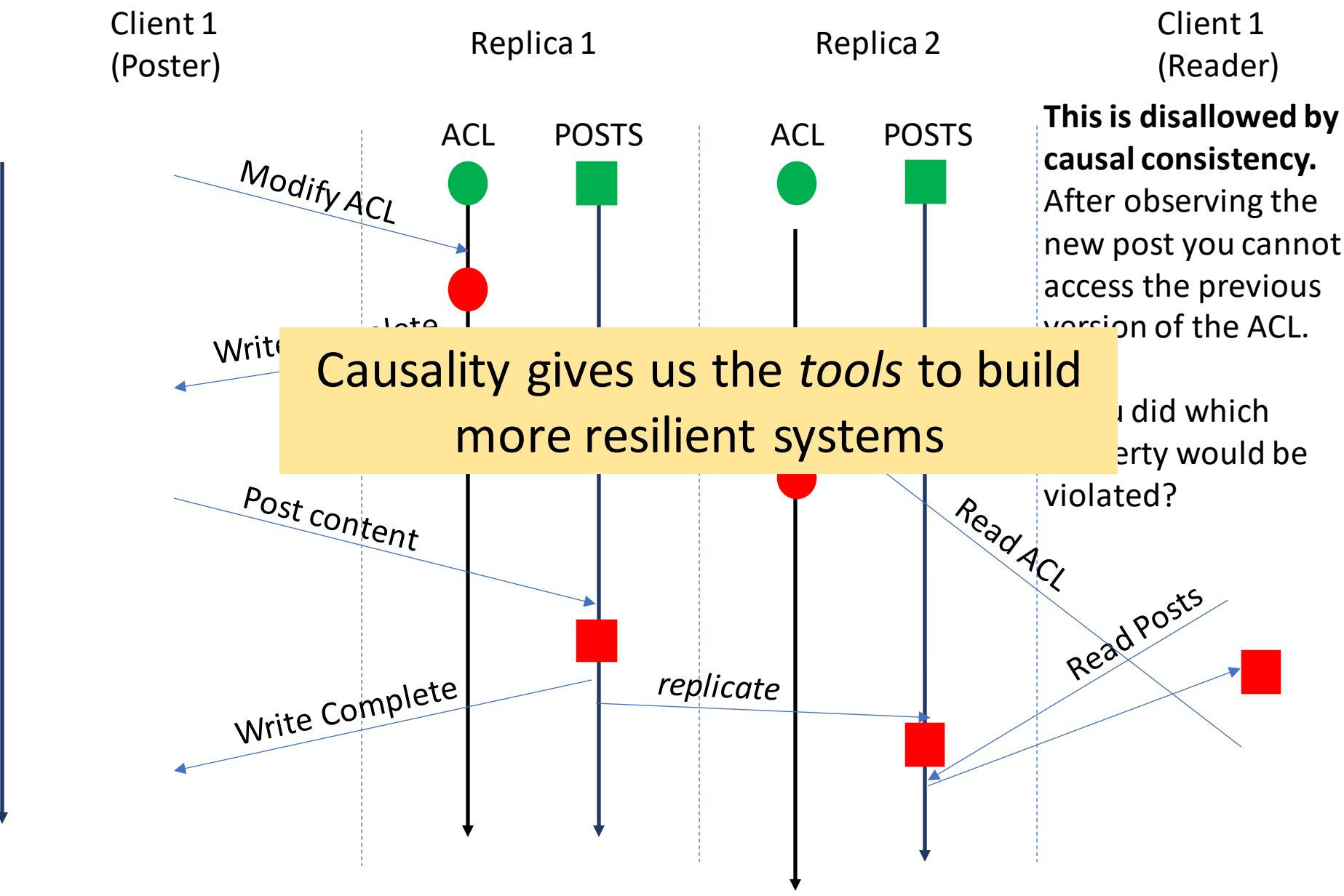
# How can this not go Wrong (Take3):




**This is disallowed by causal consistency.** After observing the new post you cannot access the previous version of the ACL.

If you did which property would be violated?

# How can this not go Wrong (Take3):




**This is disallowed by causal consistency.** After observing the new post you cannot access the previous version of the ACL.

u did which erty would be violated?

Causality gives us the *tools* to build more resilient systems