# Replication

Ashvin Goel

Electrical and Computer Engineering
University of Toronto

Distributed Systems
ECE419

With thanks to Tim Harris and Martin Kleppmann,
Lecture notes on Concurrent and Distributed Systems

# Overview

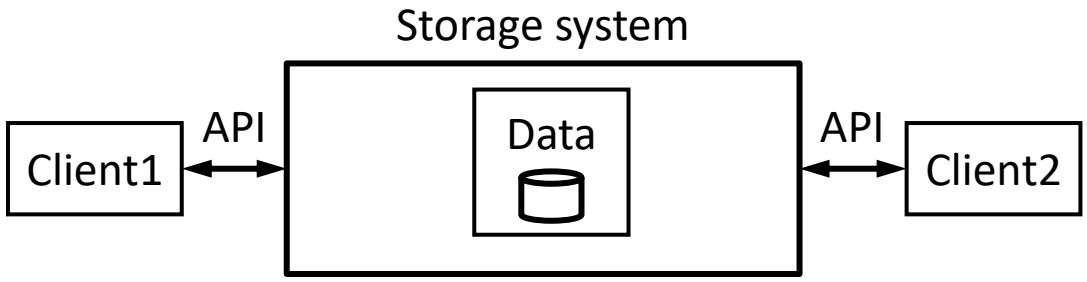
- Introduction to replication
  - What is replication?
  - Why replicate data?
  - Why is replication challenging?
- Replicated storage API
- Replication schemes

# What is replication?

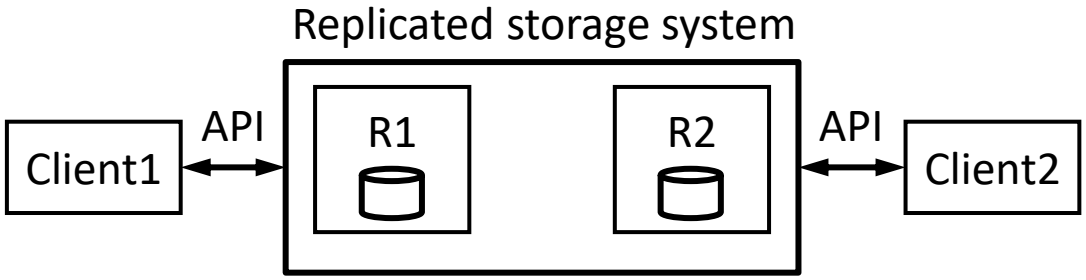
- Storage service keeps multiple copies of data on
  - Different nodes
  - Different datacenters
  - Different countries/continents
- A node that has a copy of the data is called a replica
- Replication is commonly used in file systems, databases, key-value stores
  - E.g., all common cloud storage providers replicate data

# Replicated storage model

- Recall, clients use read/write API to access storage system




- Clients use the same API for a replicated storage system

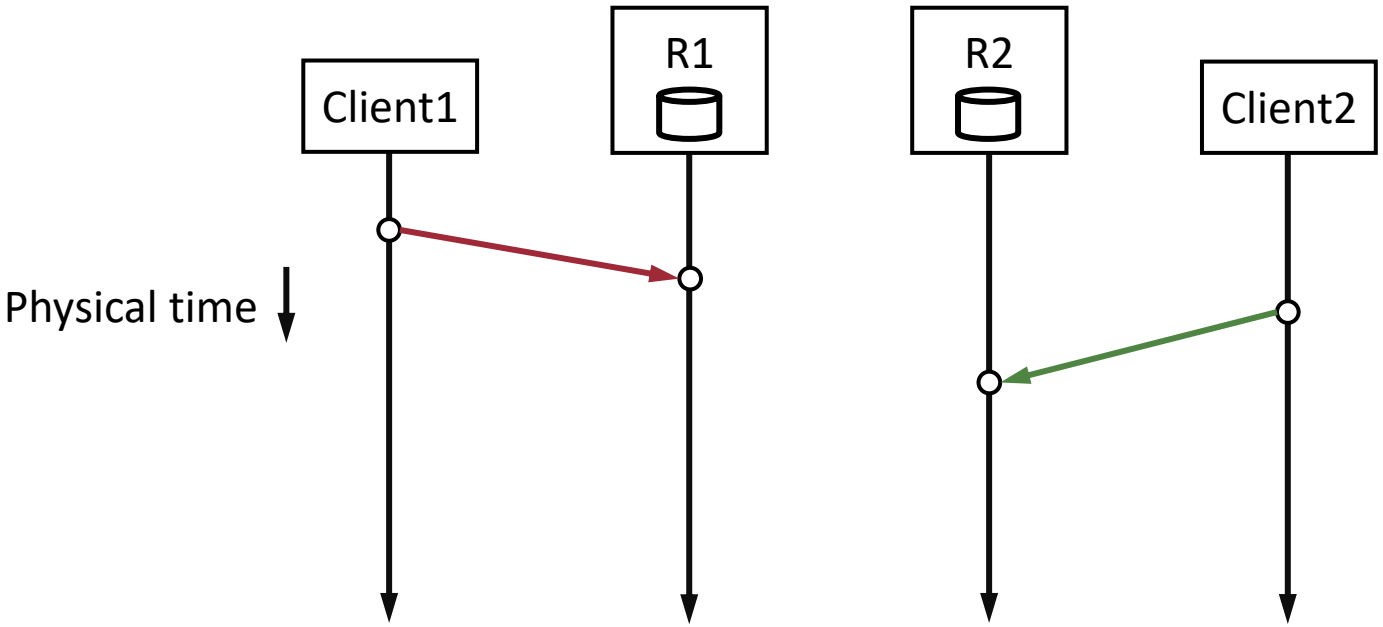


# Why replicate data?

- Scalability
- Availability

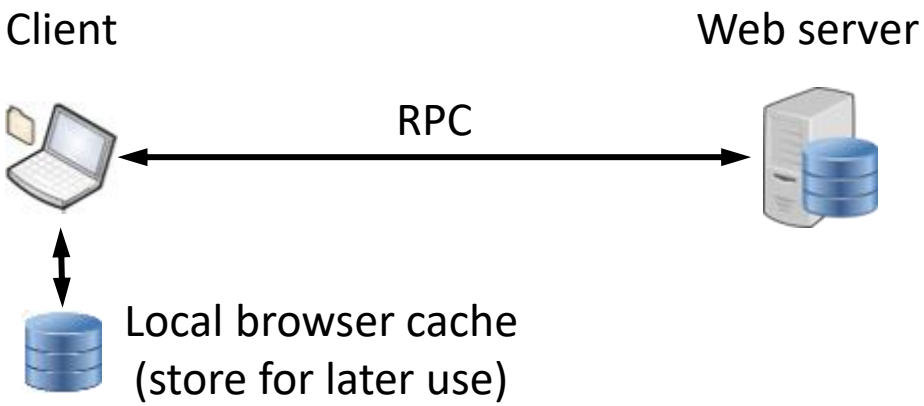
# Replication for scalability

- Improves throughput
  - Clients access different replicas, spreads load across replicas
- Lowers latency
  - Clients can access close-by replica



# Replication for scalability

- Caching is an example of replication
  - Creates copies of data
- Web browser cache
  - Browser caches web server content locally for faster access

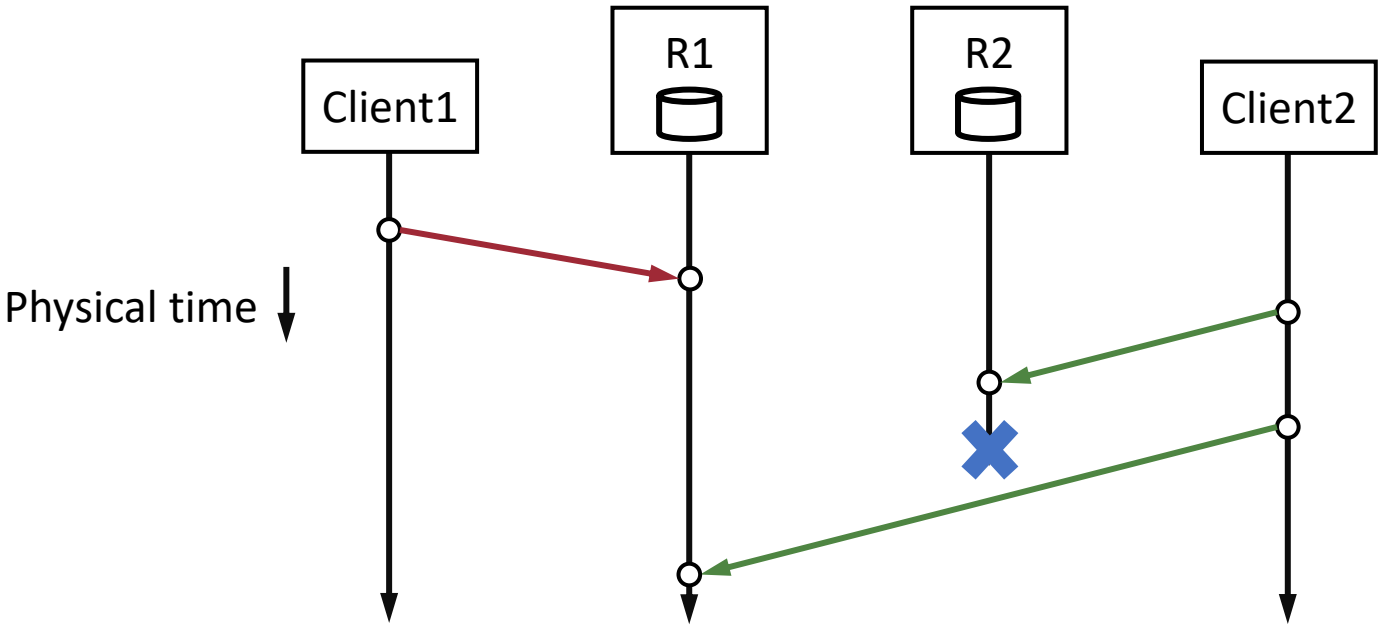


# Replication for scalability

- Caching is an example of replication
  - Creates copies of data
- Geo-replicated caches
  - Storage services replicate data on different continents to reduce latency, improve throughput for geo-distributed clients

# Replication for availability

- When replica fails, clients can access another replica, i.e., replication helps tolerate faults (fault tolerance)
  - Application avoids downtime in case of server failure, i.e., replication helps improve service availability
  - Application avoids losing data in case of storage failure



# Replica availability

- A replica may be unavailable due to
  - Network partition (e.g., node cannot be reached), or
  - Node fault (e.g. crash, hardware issue, planned maintenance)
- Assume a replica has probability p of being unavailable at any one time, and assume faults are independent
  - Assume there are N replicas in the system
  - Probability of all N replicas being faulty: $p^N$
  - Probability of all N replicas being correct: $(1 - p)^N$
  - Probability of at least one replica being faulty: $1 - (1 - p)^N \approx pN$

# Replication and availability

- Example with p=0.01, assuming independent failures

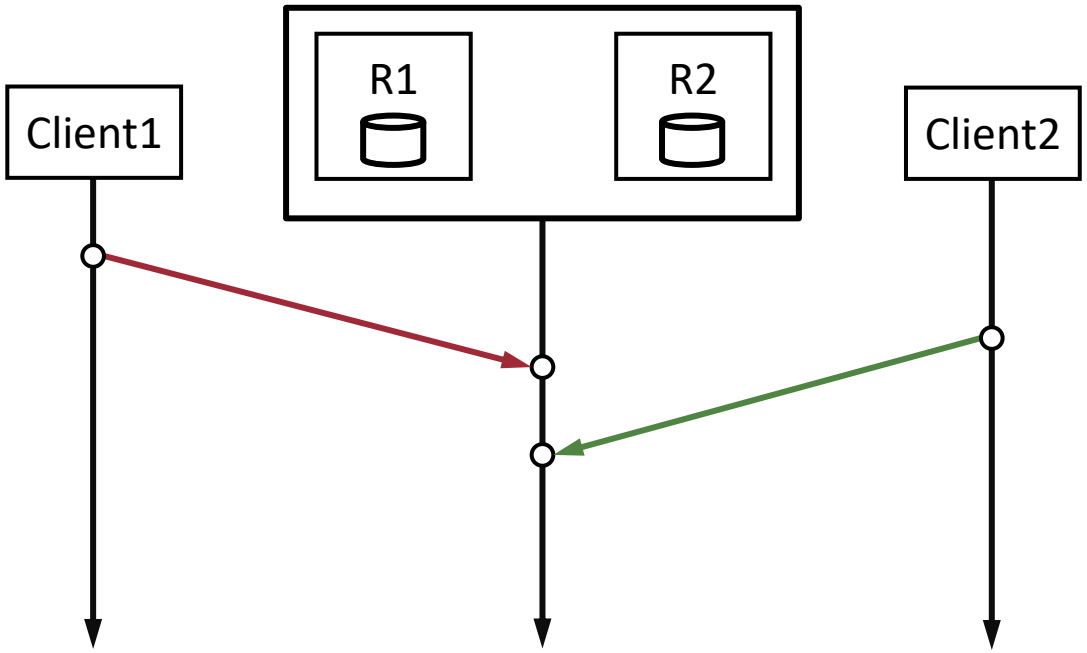
| replicas n | P(>= 1 faulty)<br>At least 1 faulty | P(all n faulty)<br>All faulty | P(>= (n+1)/2 fault)<br>Majority faulty |
|---|---|---|---|
| 1 | 0.01 | 0.01 | 0.01 |
| 3 | 0.03 | $10^{-6}$ | $3.10^{-4}$ |
| 5 | 0.05 | $10^{-10}$ | $1.10^{-5}$ |

If replication requires all replicas to be available, then availability decreases with more replicas!

If replication can tolerate some replicas being unavailable, then availability increases with more replicas!

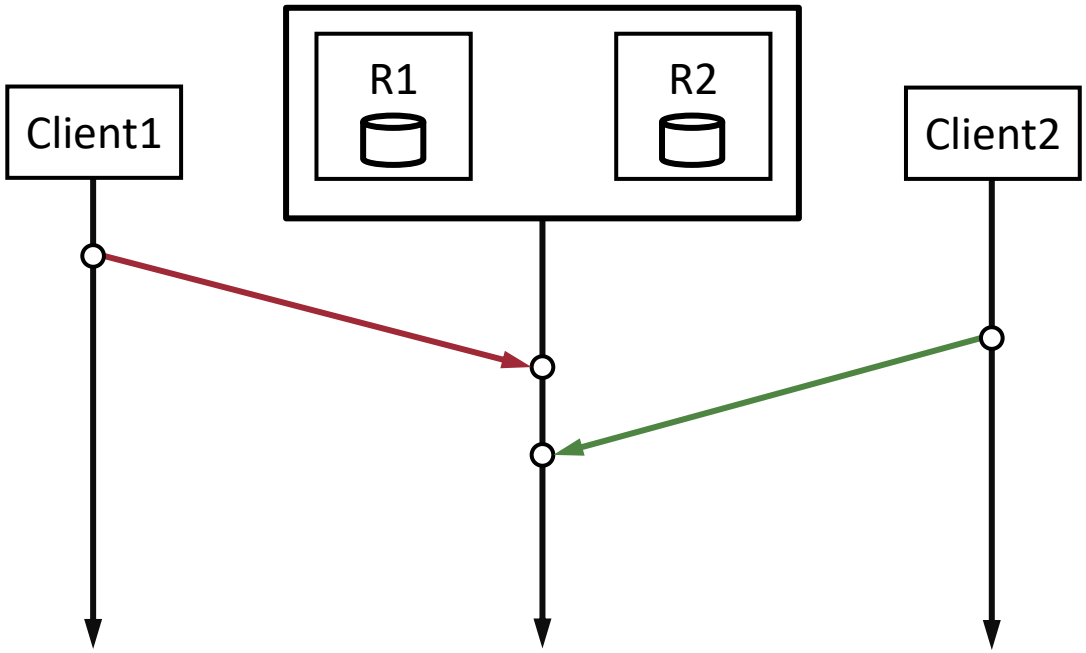
# Replication goals

- Ideally, ensure that clients are unaware of replication, observe the same behavior as a single machine that
  - Provides scalability - high-throughput, low-latency
  - Provides availability - appears to never fail



# Data consistency

- What is “observe same behavior as single machine”?
- Replicated system ensures linearizability
  - All clients observe the same order of writes
  - Clients read latest data (immaterial of replica accessed)

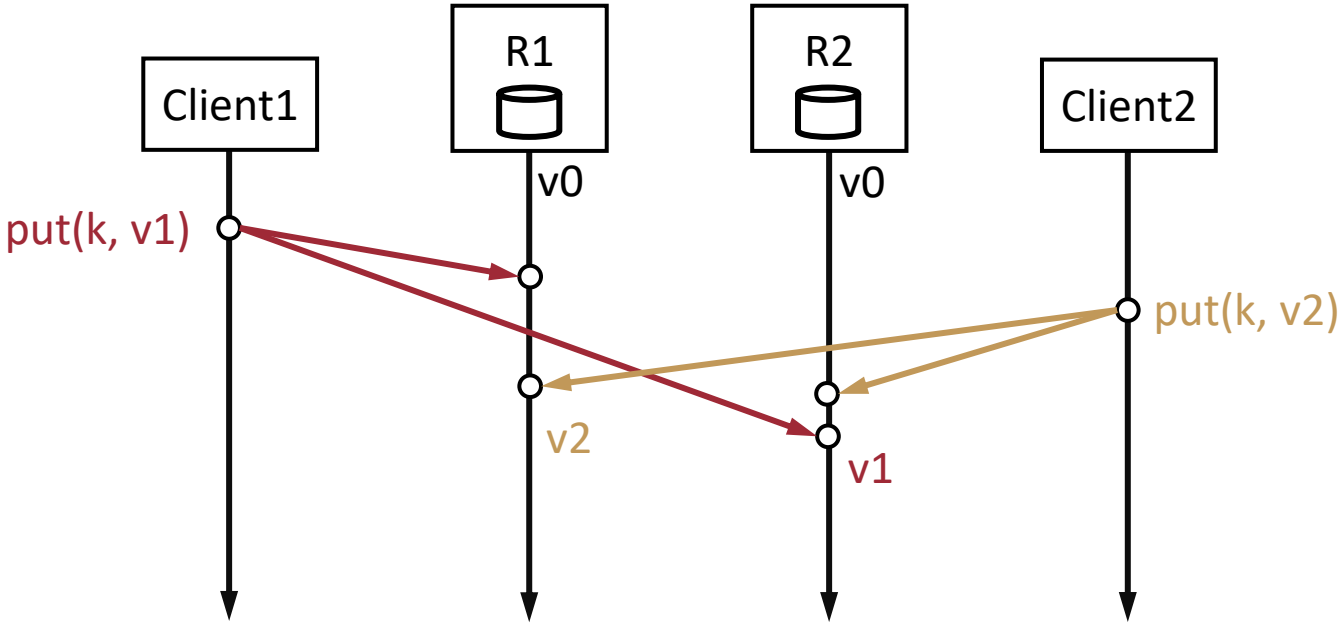


# Why is replication challenging?

- Suppose all accesses are read accesses,
  e.g., accesses to statically replicated web pages
  - Easy to meet replication goals
    - Reads access latest content
    - Reads from different clients can be sent to different replicas, have high throughput, low latency
    - On replica failure, reads can be switched to another replica, ensuring fault tolerance
- Writes complicate replication
- Failures complicate replication

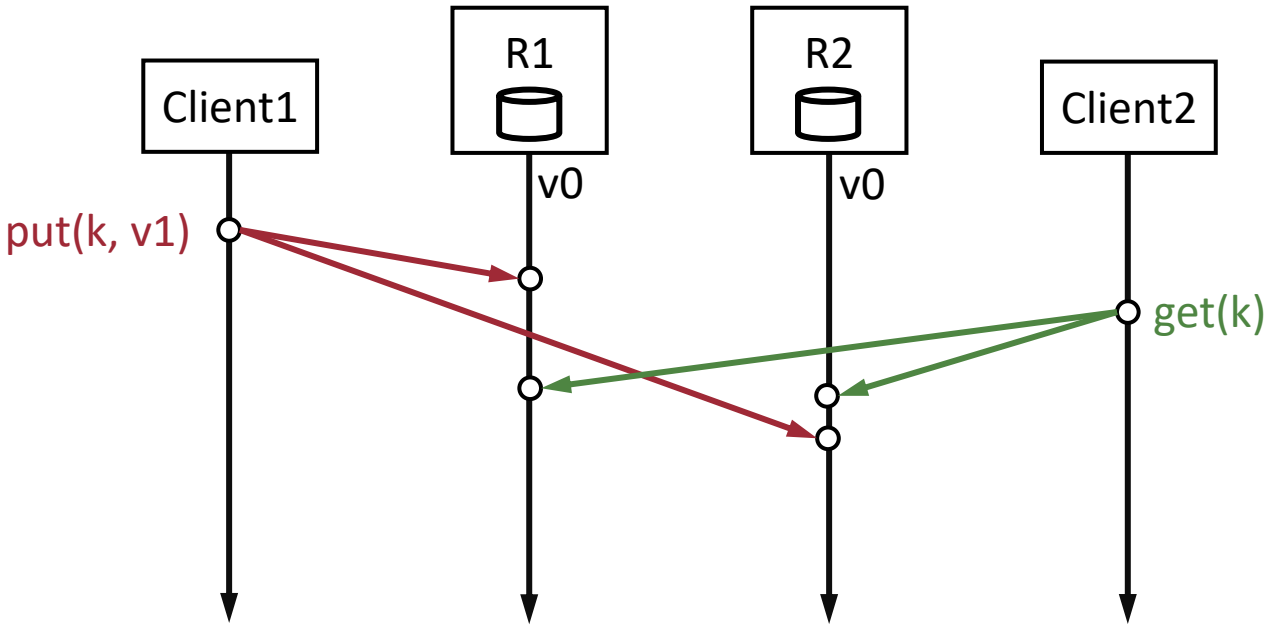
# Writes complicate replication - 1

- Client 1 issues a put(k, v1), Client 2 issues a put(k, v2)
  - Now the replicas are inconsistent!
  - Need to order concurrent put() operations



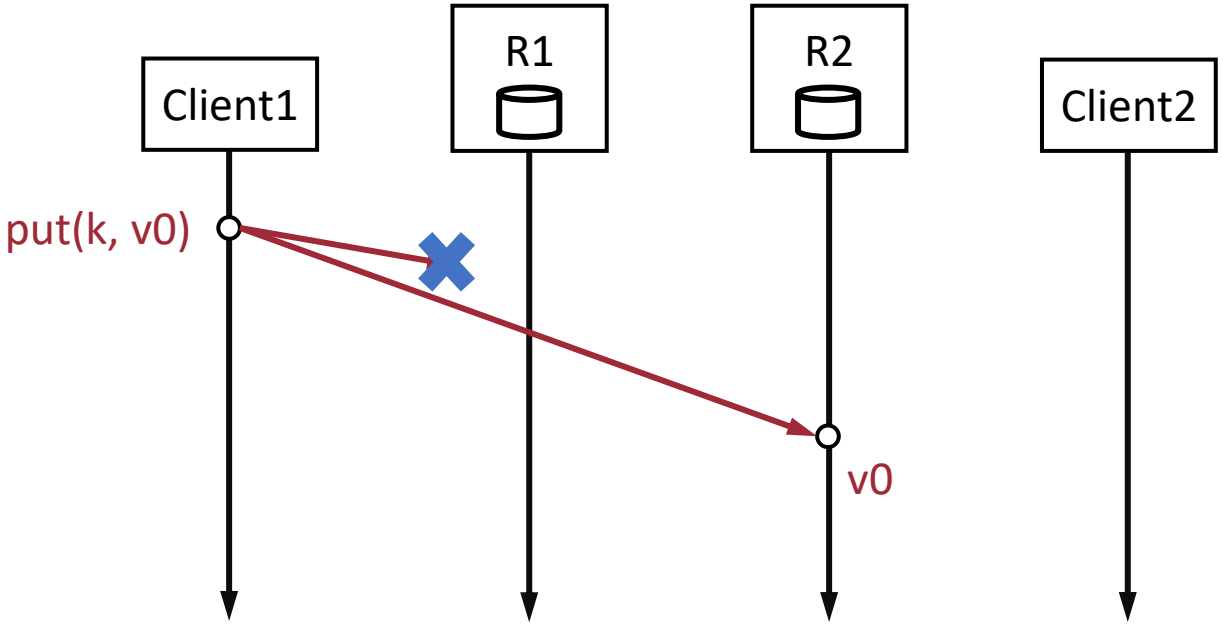
# Writes complicate replication - 2

- Client 1 issues a put(k, v1), Client 2 issues a get(k)
  - get(k) may return v0 and v1, which value is correct?
  - Need to order concurrent put() and get() operations



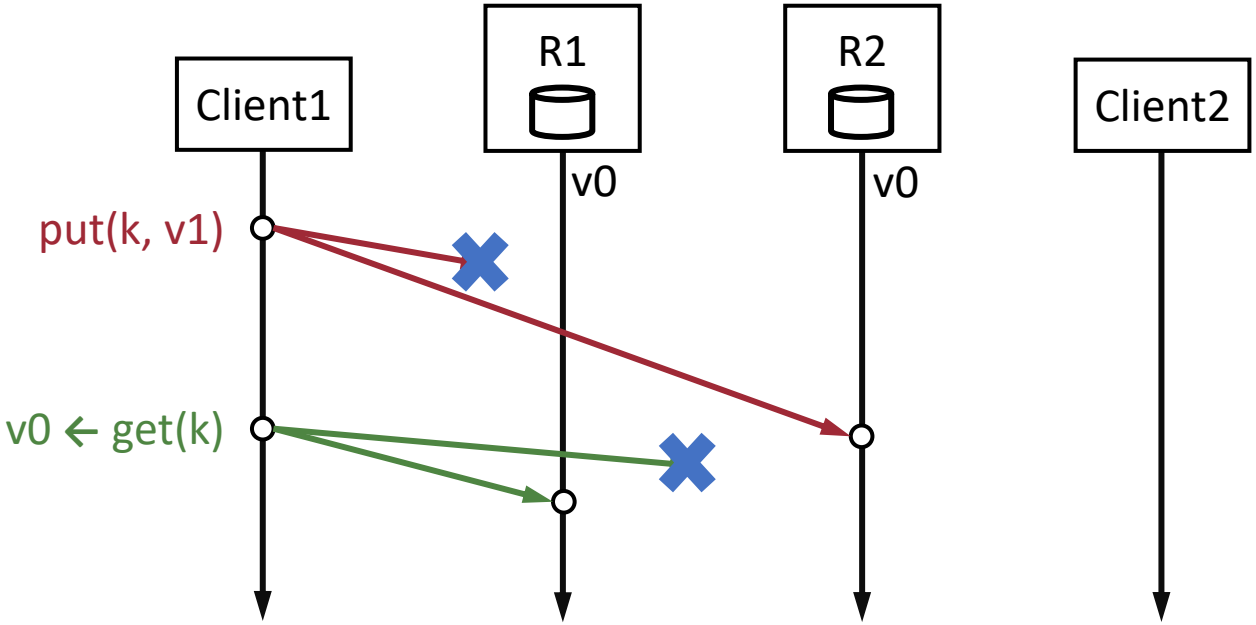
# Failures complicate replication - 1

- Client 1 issues a put(k, v0)
  - Request to R1 fails, now the replicas are inconsistent!
  - Replicas can't tell whether client only added key on R2, or only removed key from R1!
  - Need replicas to distinguish between these cases, handle inconsistency between replicas



# Failures complicate replication - 2

- Client 1 issues a put(k, v1) and then get(k)
  - Client 1 reads a stale value, seems like put() is lost!
- What if put(k, v1) and get(k) wait for both replicas?
  - Then, a single replica failure delays put()/get() indefinitely!
  - Cannot wait for all replicas, or else poor availability



# Replicated Storage API

# Storage API

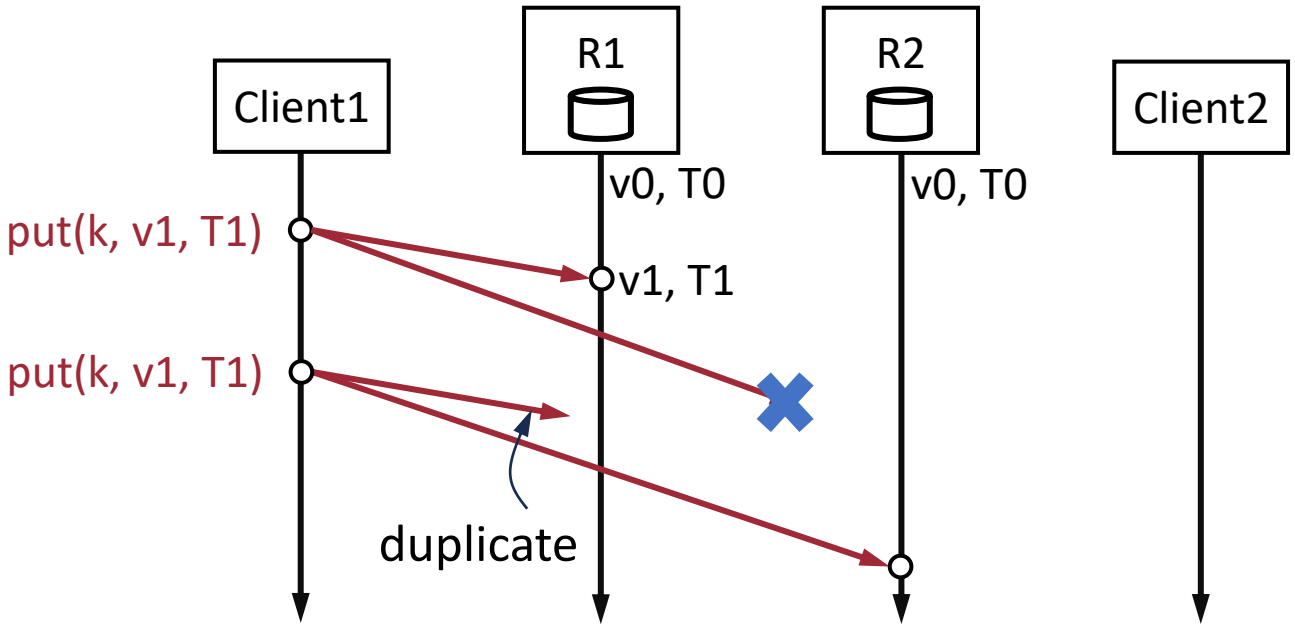
- Assume storage system provides following operations:
  - put(key, value, T) // create or update key with value, timestamp
  - (value, T) = get(key) // return the value and timestamp of key
  - del(key, T) // delete key
- Client generates unique timestamp T for put(), del()
  - T is client's logical or vector clock timestamp (other options are possible)
- Each replica stores kv-pair records:
  - timestamp can be a scalar or vector
  - visible flag indicates record existence
  - Typically, records accessed using an index, and stored on disk in hierarchical format (e.g., B-tree)

| key | value | timestamp | visible |
|---|---|---|---|
| k0 | v0 | T0 | true |
| k1 | v1 | T1 | true |
| k2 | v2 | T2 | false |

key-value records

# Purpose of timestamp

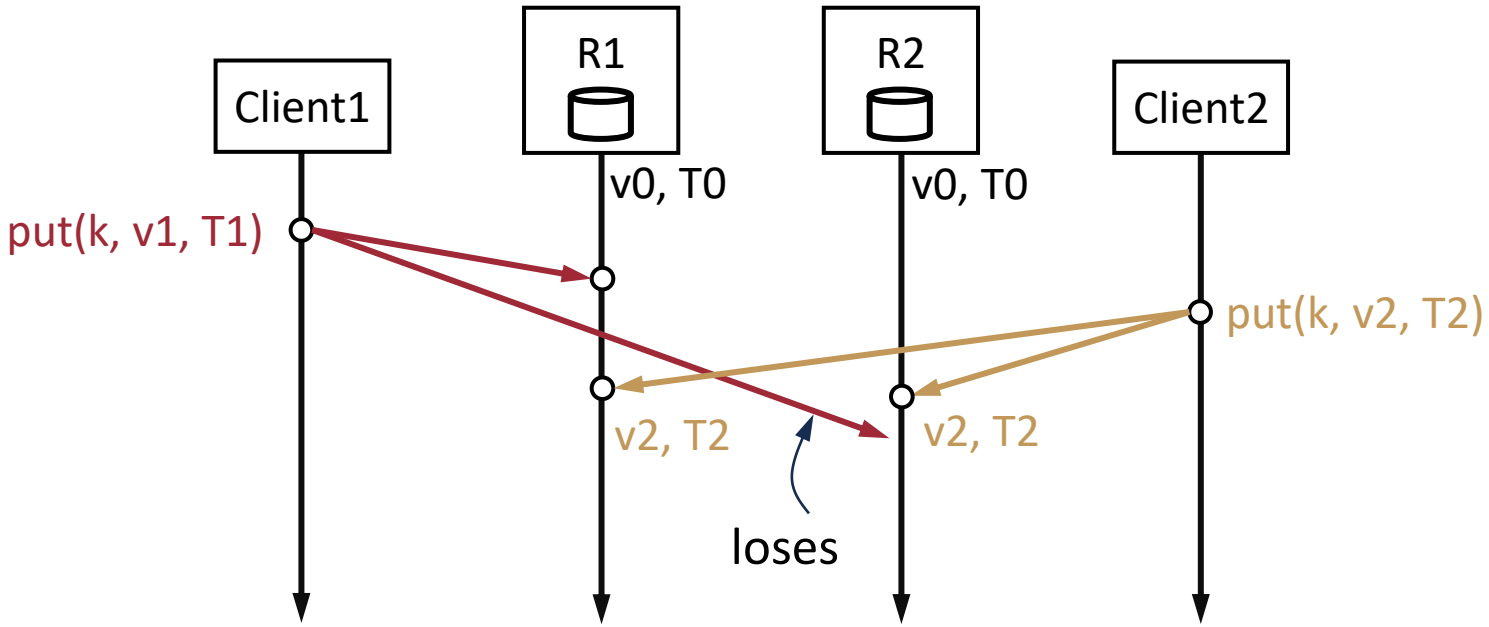
- Timestamps allow ignoring duplicate requests
  - Recall at-most once RPC semantics



# Ordering writes – logical timestamp

- Timestamps also allow ordering concurrent writes, using two common approaches:

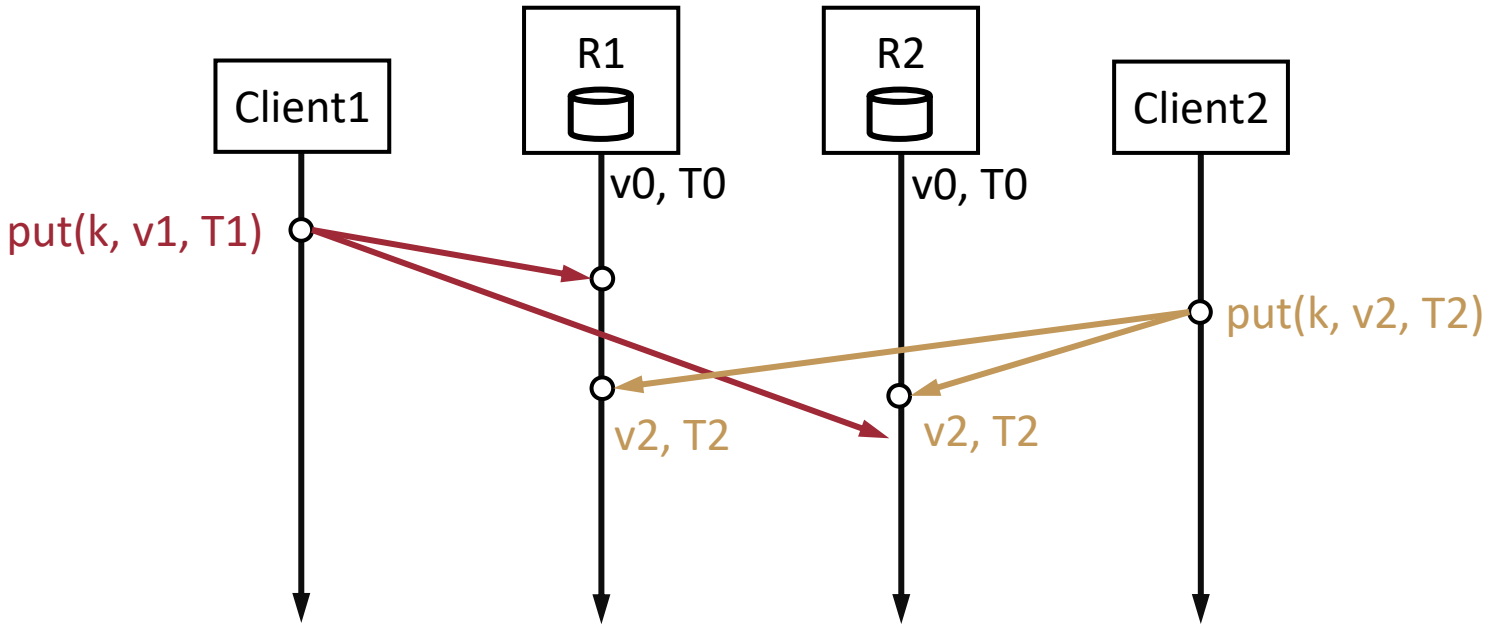
  1. Use total order timestamp, e.g., logical timestamp
     - v2 replaces v1, if T2 > T1;
     - Last writer wins, can lose data



# Ordering writes – vector timestamp

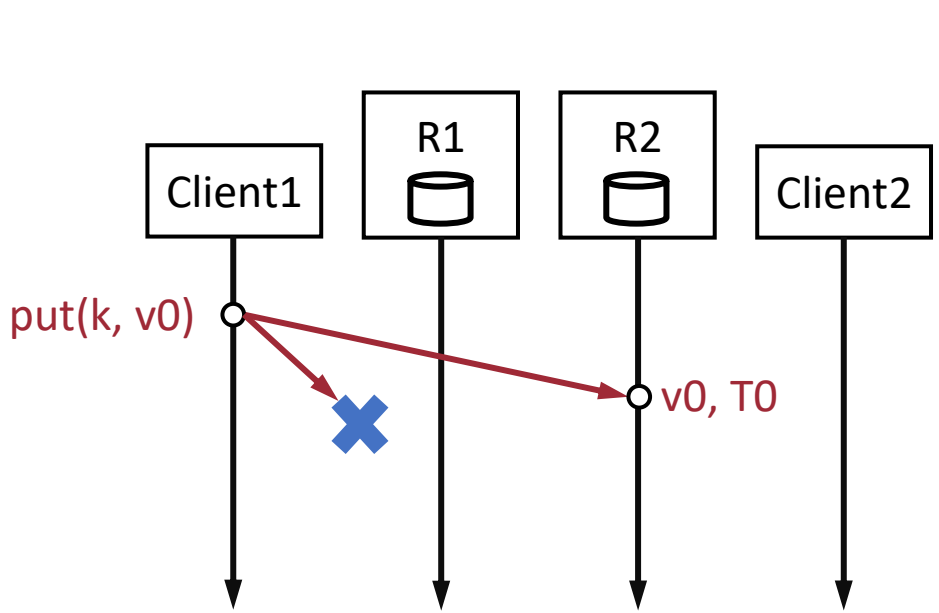
- Timestamps also allow ordering concurrent writes, using two common approaches:

  2. Use partial order timestamp, e.g., vector timestamp
     - v2 replaces v1, if T2 > T1; preserve both {v1, v2} if T1 ‖ T2;
     - Complicated scheme, vector timestamps can become large

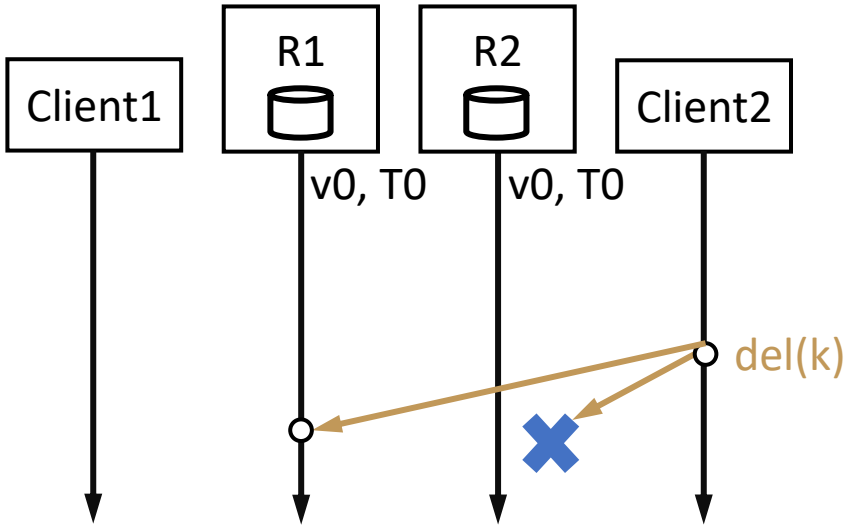


# Purpose of visible flag

- When put() creates a record, replica sets visible to true
- When del() deletes a record, replica will not delete it, instead, it will set visible to false for the record
- Now replicas can tell whether put() or del() didn't succeed




put() not successful on R1




del() not successful on R2

# Reconciling replicas

- Replicated systems need to detect differences between replicas and reconcile them
  - E.g., when replicas are added, when replicas crash and recover
- This reconciliation process (also called anti-entropy) helps ensure that replicas eventually hold same data
- During reconciliation, say
  - Replica R1 has record with visible flag set to false, and
  - Replica R2 has the same record with visible flag set to true
- What should be done?
  - Record timestamps also allow ordering requests during reconciliation

# Replication Schemes

# Replication schemes

- Quorum-based replication
- Broadcast-based replication
  - Primary-backup replication
  - State machine replication
- Optimistic replication (discussed later)

# Replication conundrum

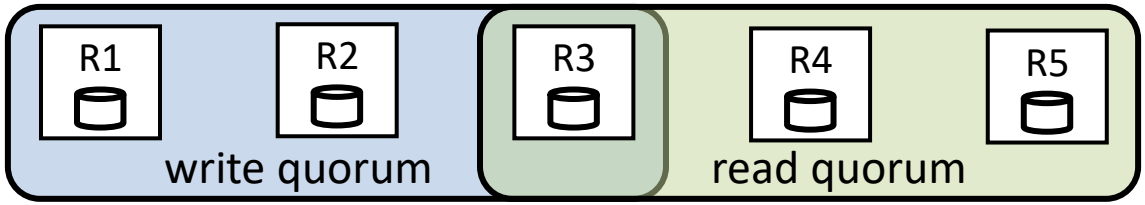
- For data consistency
  - Need to order put() operations across replicas
  - Need to ensure get() operations read latest data
- For high availability
  - Replicas may fail, can't wait for all replicas to respond, or else availability decreases with more replicas
- But then, how do we know that reads return latest data?

# Quorum-based replication

- Assume there are N replicas
- get() and put() use best-effort broadcast,
  i.e., requests may be lost, duplicated or reordered
- With quorum-based replication, assume:
  - A put() returns successfully when W replicas respond successfully
  - A get() returns successfully when R replicas respond successfully

# Use quorums for correct ordering

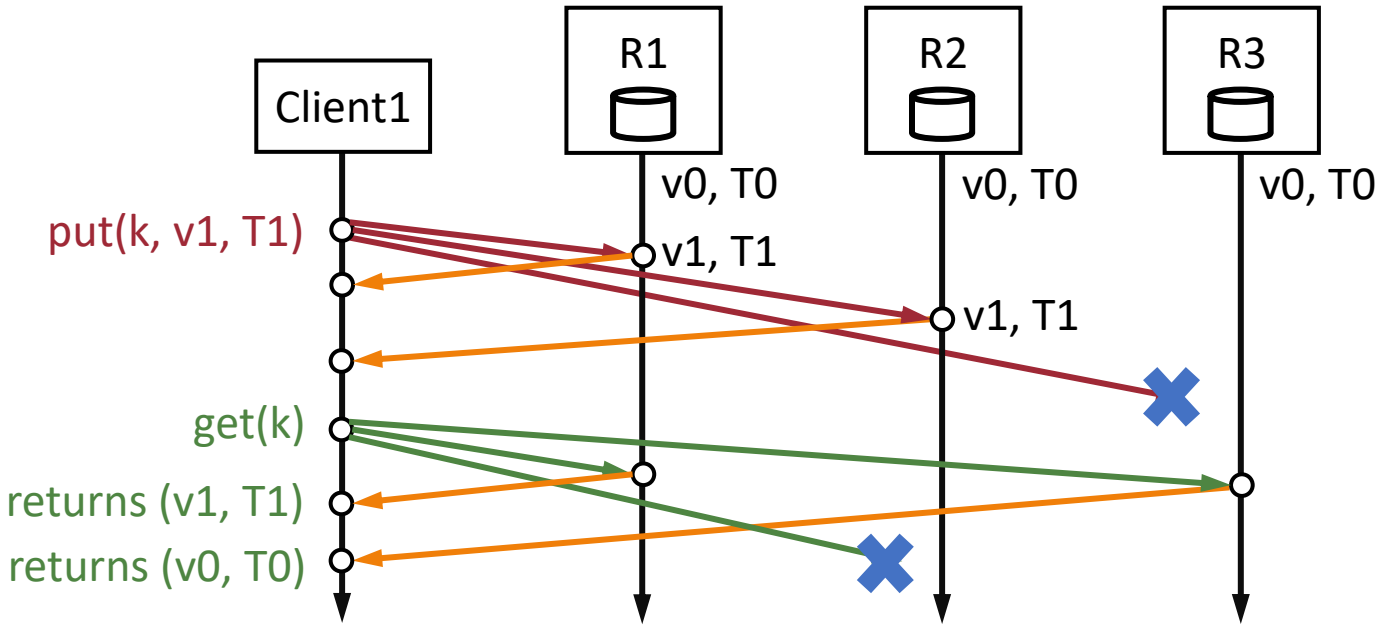
- Choose: R + W > N
  - Typically, a majority quorum is used: R = W = (n+1)/2
    - N = 3, R = 2, W = 2
    - N = 5, R = 3, W = 3
- Since a put() is acknowledged by W replicas,
  and a get() is subsequently reads from R replicas,
  a get() will overlap with last put() at at-least one replica




- So, get() will read latest data from at least one replica,
  even when (n-1)/2 replicas are unavailable!

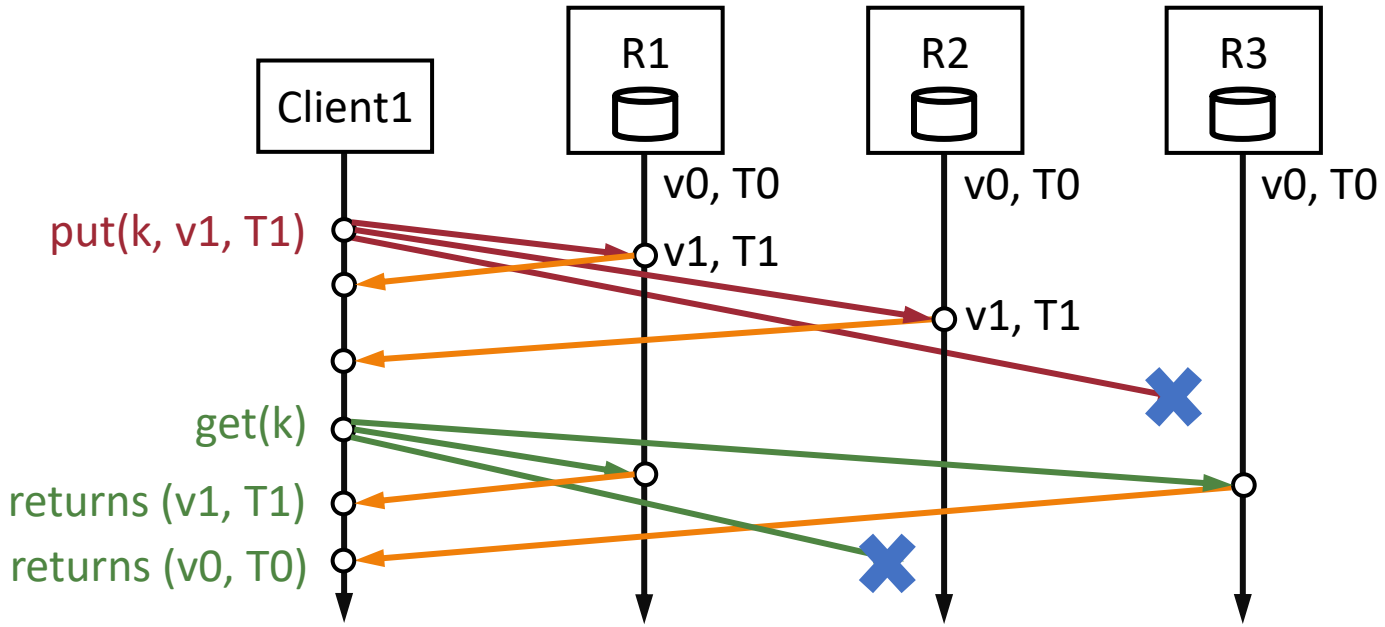
# Quorum replication and availability

- Majority quorum allows 1 replica failure with 3 replicas
  - put(k, v1, T1) succeeds on R1 and R2
  - get(k) succeeds on R1 and R3
    - R1 returns (v1, T1), R3 returns (v0, T0): choose v1 (later one)



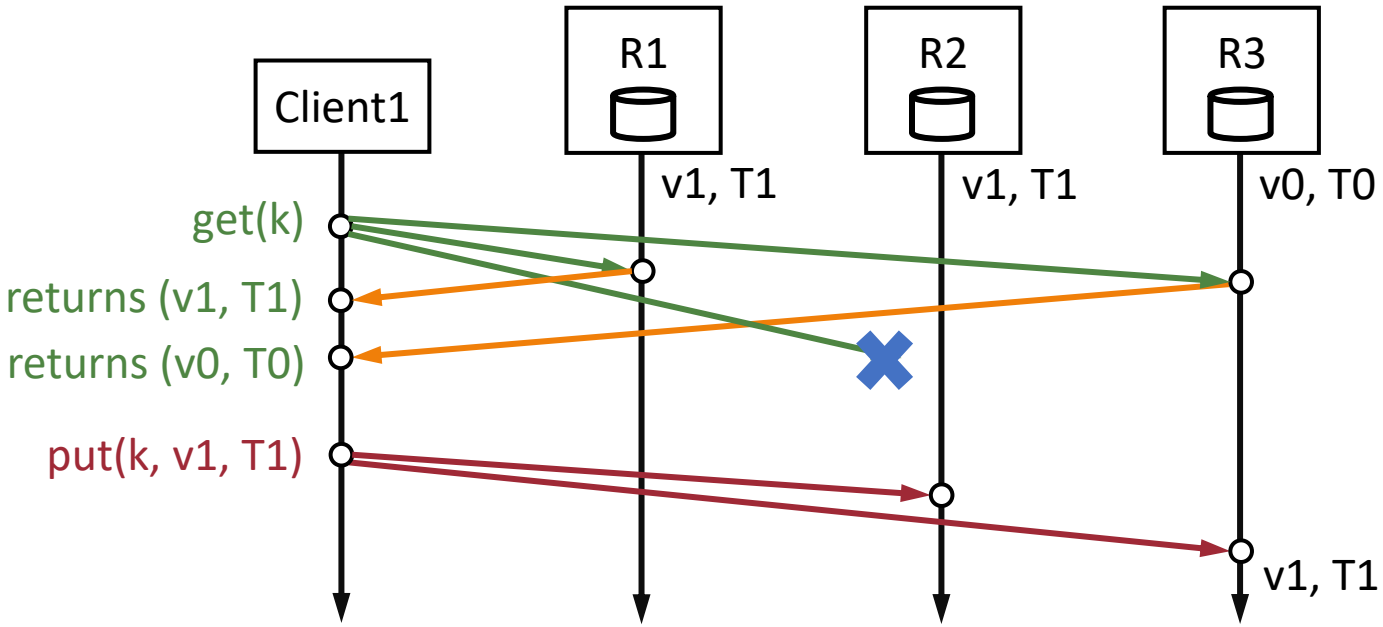
# Quorum and replica synchronization

- put() is not immediately delivered to all replicas ⇒ get() may read stale values from some replicas (e.g., v0)
- We can use reconciliation to synchronize the replicas



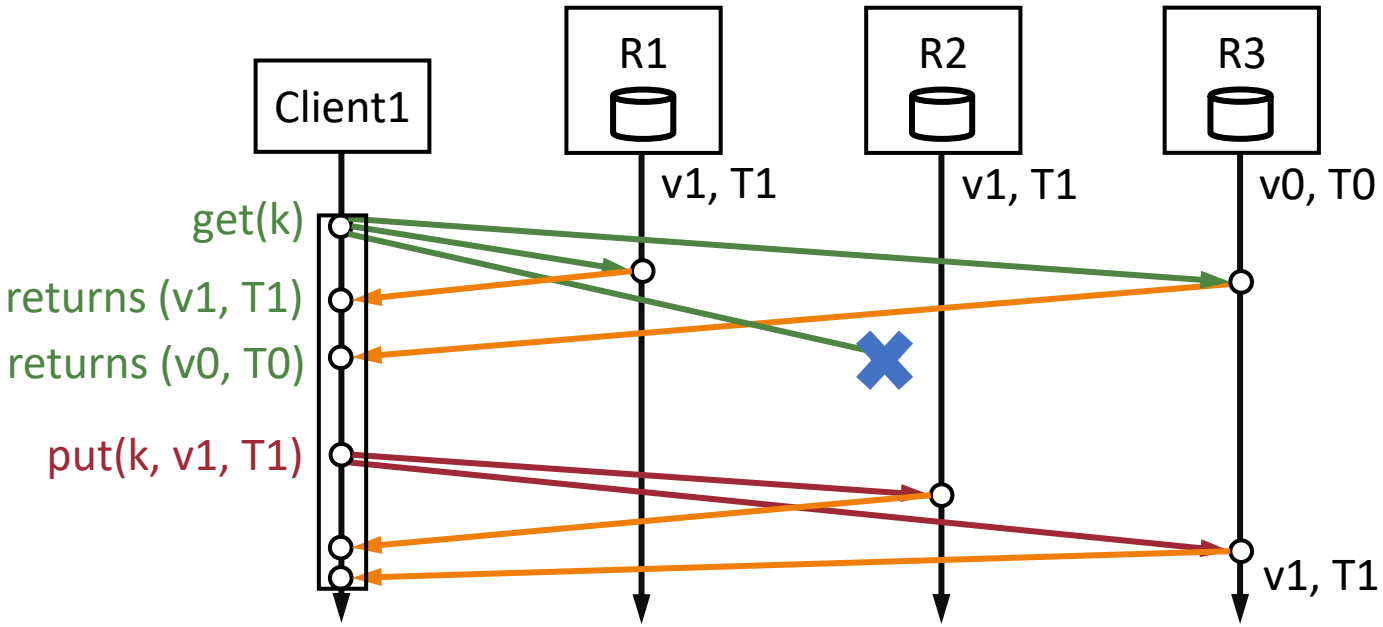
# Quorum and read repair

- put() is not immediately delivered to all replicas ⇒ get() may read stale values from some replicas (e.g., v0)
- Another option is to perform read repair
  - After get() returns, it issues a put() with the latest value to all replicas that responded with stale value or did not respond



# Quorum and linearizability

- If get() returns before read repair is done, it is possible to show that another get() can read stale value
- But, if get() returns only after read repair has finished, then quorum replication can ensure linearizability
  - For more details, look for the ABD algorithm



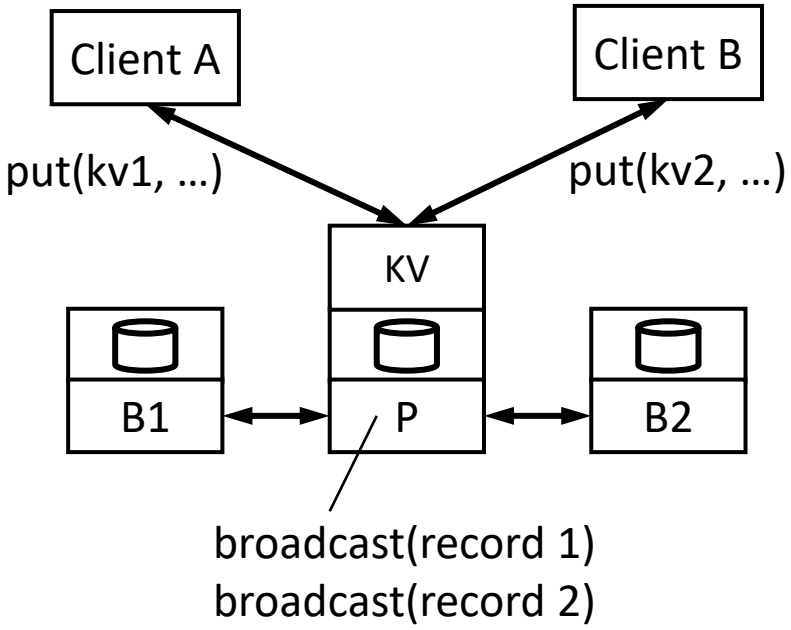
# Broadcast-based replication

- Two schemes based on FIFO-total order broadcast, both can ensure linearizability
  - Primary-backup replication
    - One replica is primary, others backup
      - Primary receives and executes operations
      - Replicates updated state to backup (passive replication)
    - Traditionally, fault tolerance based on timeout
  - State machine replication (SMR)
    - Symmetric replicas
      - Any replica receives and replicates operations
      - All replicas execute operations (active replication)
    - Fault tolerance based on consensus algorithm
- Various hybrid solutions that combine approaches

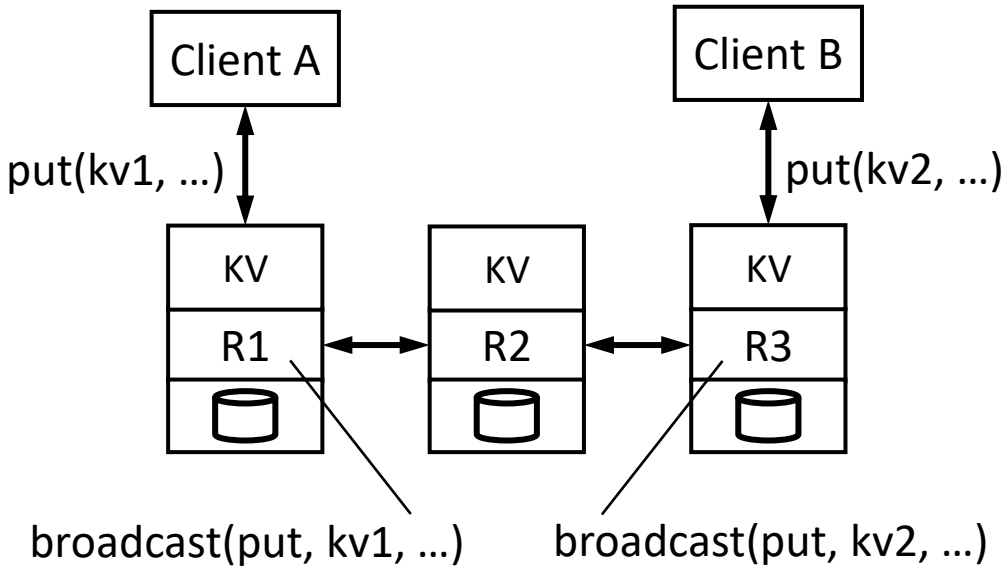
# Replicated KV store

- PB replicates updated records
- SMR replicates KV store operations

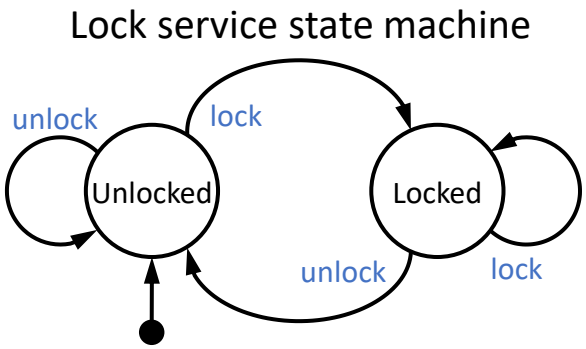
Primary-Backup Replication




State Machine Replication

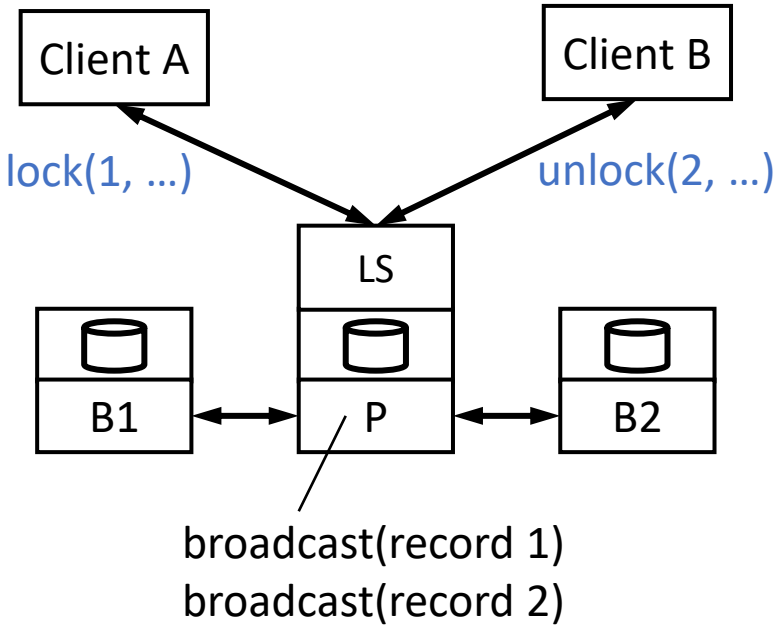


# Replicated lock service

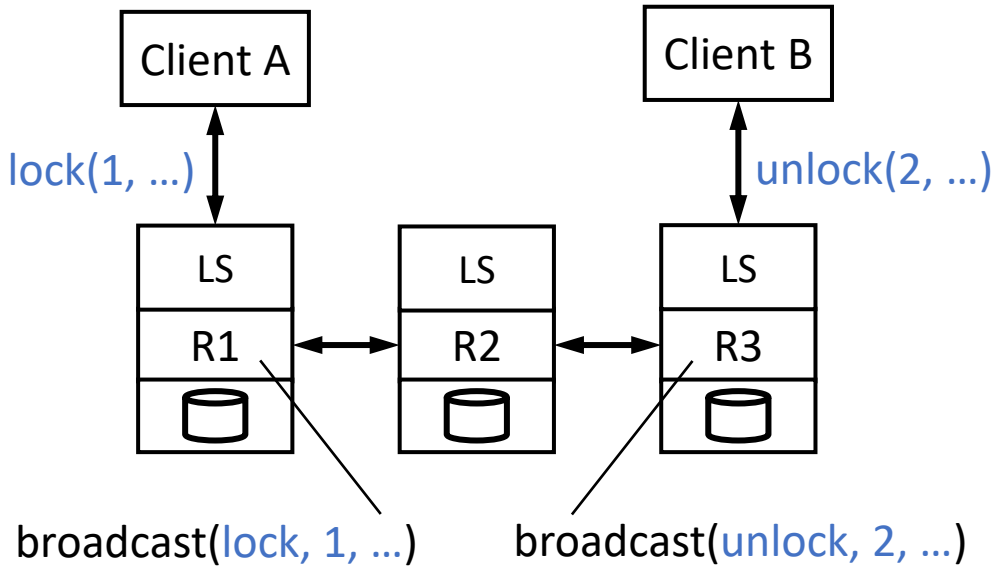



- PB replicates updated records
- SMR replicates lock service operations

Primary-Backup Replication




State Machine Replication



# Primary-backup replication

- Clients send operations to designated primary
- Primary executes each client operation serially
  - Broadcasts state updates to all backups
  - Backups apply state updates in the same order as primary
  - Backups acknowledge when they are done
- Primary waits for acks from all backups, then responds
  - If primary fails, one backup becomes primary
  - If backup fails, primary responsible for starting another backup
- Key requirement
  - Agreement: There should only be one primary

# Handling primary failure

- Like leader-based total order broadcast, handling primary (leader) failures safely is not simple
- Traditionally, a separate server called view server detects a primary failure based on timeout
  - View server elevates an up-to-date backup to a new primary
  - New primary lets all backups know that it is new primary, so backups stop accepting requests from old primary
  - As clients learn about the new primary, they start using it
- What may be the problems with using a view server?

# State machine replication

- Clients send deterministic operations to any replica
  - Replicas may receive concurrent requests
- When a replica receives an operation, it broadcasts that operation to all replicas
- All replicas execute operations in the same order, producing a consistent response for the client
- Key requirements:
  - Initial state: All replicas start in the same state
  - Determinism: All replicas receiving the same input on the same state produce the same output and resulting state
  - Agreement: All replicas process inputs in the same sequence

# SMR fault tolerance

- Fault tolerance in SMR depends on the underlying total order broadcast protocol
- Later, we will look at fault-tolerant total order broadcast
  - Like quorum-based replication, it will provide availability even when (n-1)/2 replicas are unavailable

# Comparing replication methods

| | Quorum | Primary-Backup | SMR |
|---|---|---|---|
| Replication method | • Symmetric replicas<br>• Replicates get()/put() operations | • 1 primary, others backup<br>• Replicates records from primary to backup | • Symmetric replicas<br>• Replicates SM operations |
| Programming model | • get(), put() | • Arbitrary operations | • Deterministic operations |
| Consistency | • Based on quorum and read repair<br>• Cannot provide linearizability for CAS operations<br>• Can be used with weak consistency schemes (later) | • Based on total order of operations<br>• Can provide linearizability | • Based on total order of operations<br>• Can provide linearizability |

# Comparing fault tolerance

| Primary-Backup | Quorum and SMR |
| --- | --- |
| • Pros: requires f+1 replicas to handle f failures<br><br>• Cons: requires separate view server; primary failures visible to clients; timeouts need to be conservative to avoid split brain | • Cons: requires 2f+1 replicas to handle f failures<br><br>• Pros: f failures can be masked from clients; does not depend on timeouts for correctness |

# Conclusions

- Replication helps provides scalability and fault tolerance
  - Commonly used in modern cloud storage systems
- Goal of a replicated storage system is to provide
  - Strong (linearizable) consistency
    - Same behavior as single-copy storage system
  - High performance and availability
- We looked at quorum and broadcast-based replication
  - We will see how they are used in real systems later
- Next, we will look at fault-tolerant total order broadcast
  - Will ensure highly-availability, broadcast-based replication