▼ Consistency when executing operations concurrently

1. Consistency when executing operations concurrently
- ▼ Review: NFS and GFS
  2. Review: NFS
  3. Review: GFS
- ▼ Case study: 3FS
  4. Files behind AI workloads
  5. 3FS keeps many designs of GFS and NFS
  6. What has changed since GFS?
  7. What does locality-oblivious mean?
  8. Why locality-oblivious?
  9. Where does 3FS keep metadata?
  10. What is a transactional key-value store?
  11. Why not a single master?
  12. Which opened files does 3FS track?
  13. Why not track opened files?
  14. How do applications access 3FS?
  15. What is FUSE?
  16. Why is FUSE not enough?
- ▼ Review: consistency in NFS and GFS
  17. Review: consistency in NFS
  18. Review: consistency in GFS
  19. Review: GFS orders the writes
  20. GFS reads: clients may see different data
- ▼ Motivation: a chat app
  21. The topics of today's lecture
  22. Motivation case: Implementing a ChatAPP
  23. We call such a case `inconsistent`, i.e., the order is different
  24. Implementing ChatAPP: a (naïve) centralized approach
  25. Question: can the centralized approach reorder operations?
  26. Cons #1 (Performance): long update time
  27. Cons #2 (Performance): slow reads, and slower as the history grows
  28. Cons #3 (Reliability): cannot work offline
  29. Cons #4 (Reliability): cannot deal with server failures
- ▼ Replication: dodging the centralized costs
  30. Existing systems (like WeChat) replicate + cache the system state
  31. Challenge: the implementations of reads/writes
  32. The remedy for the update: sync the operation with all the replicas
  33. For performance/reliability, sync needs to be done on a few replicas
  34. Next question: what about reads?
  35. Reading the local copy with periodic syncs
  36. Reordering is possible in a replicated implementation
  37. Another issue: update conflicts
  38. Challenge of the merge: whose order do we keep?
- ▼ Consistency model
  39. How to cope with the complexity of distributed read/write sync?
  40. Consistency model
  41. Spectrum of consistency models
- ▼ Eventual consistency
  42. Back to our example (ChatAPP)
  43. What are the goals of a chat application
  44. Eventual consistency
  45. Implementing an eventually consistent system
- ▼ Ordering the operations
  46. Syncing: exchange the updates received so far
  47. Solution: deterministically sort all updates upon syncing
  48. Question: how to sort?
  49. Some design choices about the ordering of the sort
  50. getTimeOfDay() cannot guarantee a total order
  51. More choices about the ordering of the sort
  52. How inappropriate orders can happen upon sync
  53. Why can such an ordering happen? Unsynchronized clock time
  54. Order matters, and timing on a device alone is insufficient
  55. An ideal order is causal ordering
  56. We need a sort key (timestamp) that preserves the causal order!
  57. Solution: Lamport clock
  58. Lamport clock algorithm
  59. Questions on Lamport clock
  60. Recap of today's lecture
- ▼ 接龙: total order is too strong
  61. Motivating case: 接龙
  62. Question: can two people take the same slot
  63. Problem: what if the application requires a strong ordering
  64. Concurrent updates: invent an order, or report a conflict?
  65. Lamport clock is insufficient, we need partially ordered timestamps
  66. Realizing partial order: vector clock
  67. Two independent events: incomparable with a vector clock
  68. Take the max of every entry to preserve causality
  69. Back to 接龙: two paths towards fixing the issue
- ▼ Toward stronger consistency models: Strict consistency and Linearizability
  70. What does a strong consistency model present?
  71. Question: which order is the equivalent order?
  72. The desired order for 接龙: ordered by the issue time
  73. Question: how to implement a strong consistency model?
  74. Question: can we implement strict consistency?
  75. Challenge #1: unsynchronized clocks for the issue time
  76. Challenge #2: unbounded wait
  77. Strict consistency: strong but not practical
  78. Order maintained by the centralized design: concurrent requests
  79. Order maintained by the centralized design: non-overlapping ones
  80. Strong consistency model: what is the order provided?
  81. Implementing linearizability in replicated data storage
  82. Implementing reads/writes in a linearizable system
  83. Primary-backup model (mimics the centralized approach)
  84. What does "in order" mean? (Recall GFS)
  85. In-order updates: the primary must use some seq number
  86. Drawback of primary-backup?
  87. Question: can we optimize the implementation?
  88. Back to the spectrum
  89. Eventual vs. linearizability
- 90. Summary of today's lecture
- 91. 参考文献

Consistency when executing operations concurrently

Xingda Wei

Review: NFS

Goal: transparent access to files on a remote server

- Applications `mount` a remote directory, then use the same file system API

Design

- File system API → RPCs (`lookup`, `read`, `write`, ...)
- Stateless server: no `open` RPC; each RPC carries the file handle and the offset
  - After a failure, the client simply retries
- File handle = inode number + generation number
- Clients cache file data for performance

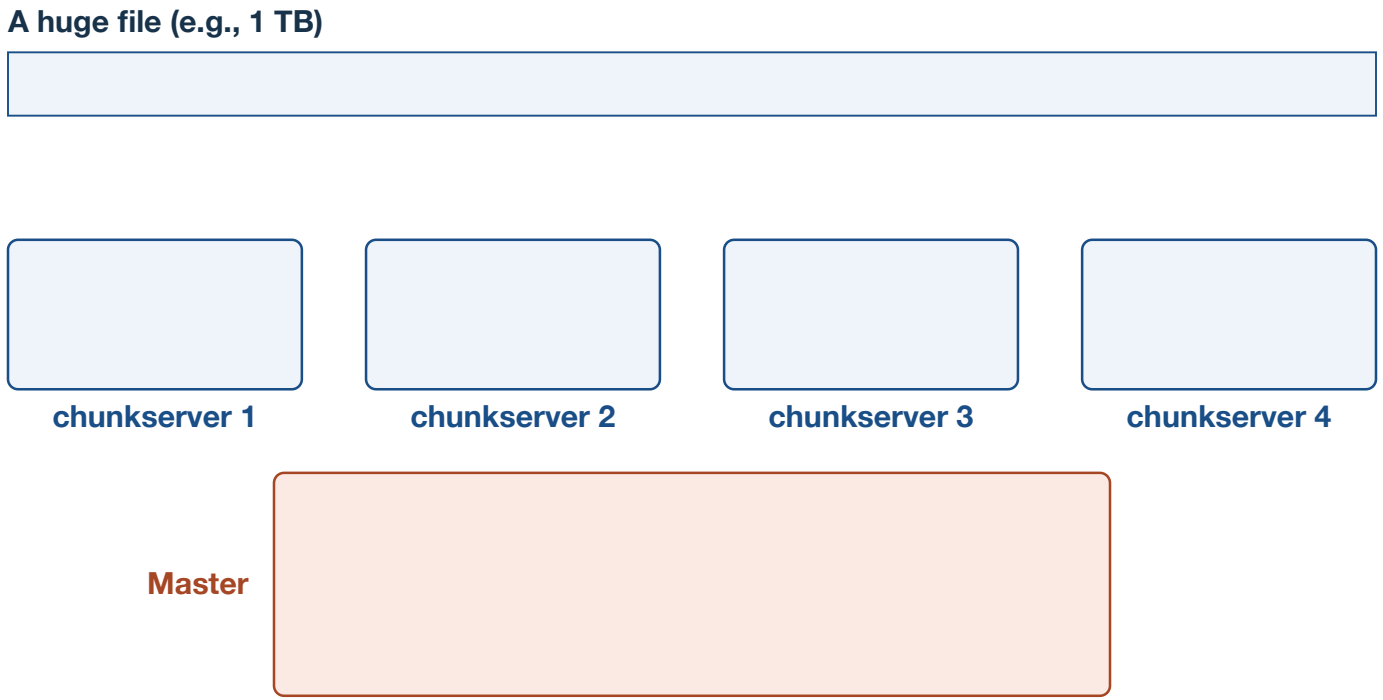
Limitation: all files live on one server

- Capacity: a file cannot be larger than the server's disks
- Fault tolerance: when the server crashes, no client can access its files

Review: GFS

GFS removes the one-server limitation of NFS

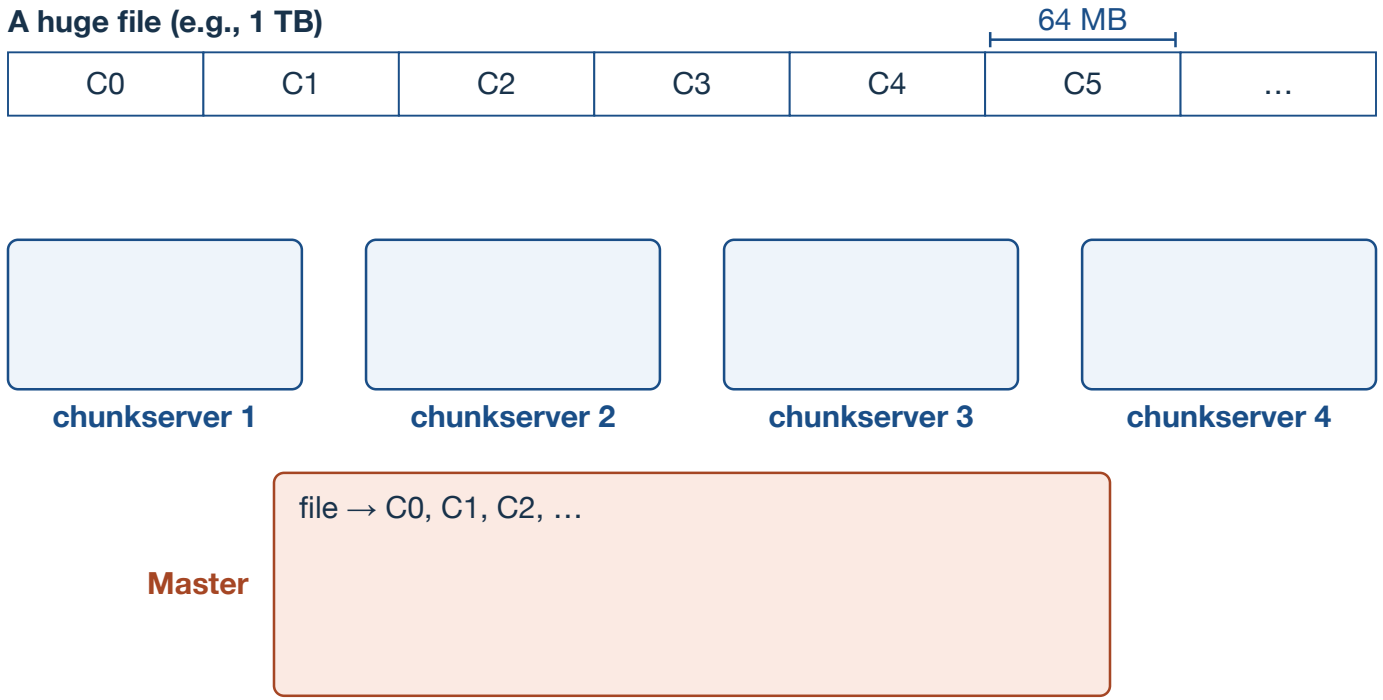
- Capacity: a file is split into 64 MB chunks, stored across many chunkservers
- Fault tolerance: each chunk is replicated on three chunkservers
- One master keeps the metadata; clients read and write data directly with chunkservers



Review: GFS

GFS removes the one-server limitation of NFS

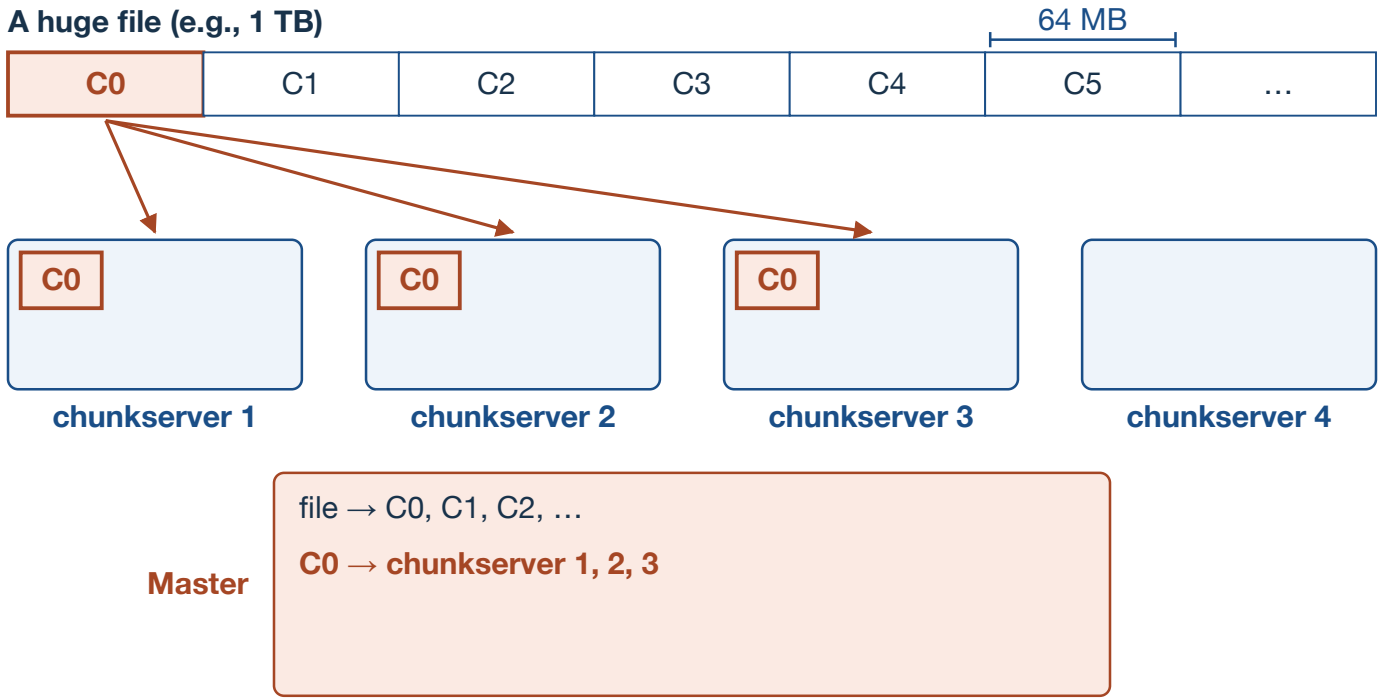
- Capacity: a file is split into 64 MB chunks, stored across many chunkservers
- Fault tolerance: each chunk is replicated on three chunkservers
- One master keeps the metadata; clients read and write data directly with chunkservers



Review: GFS

GFS removes the one-server limitation of NFS

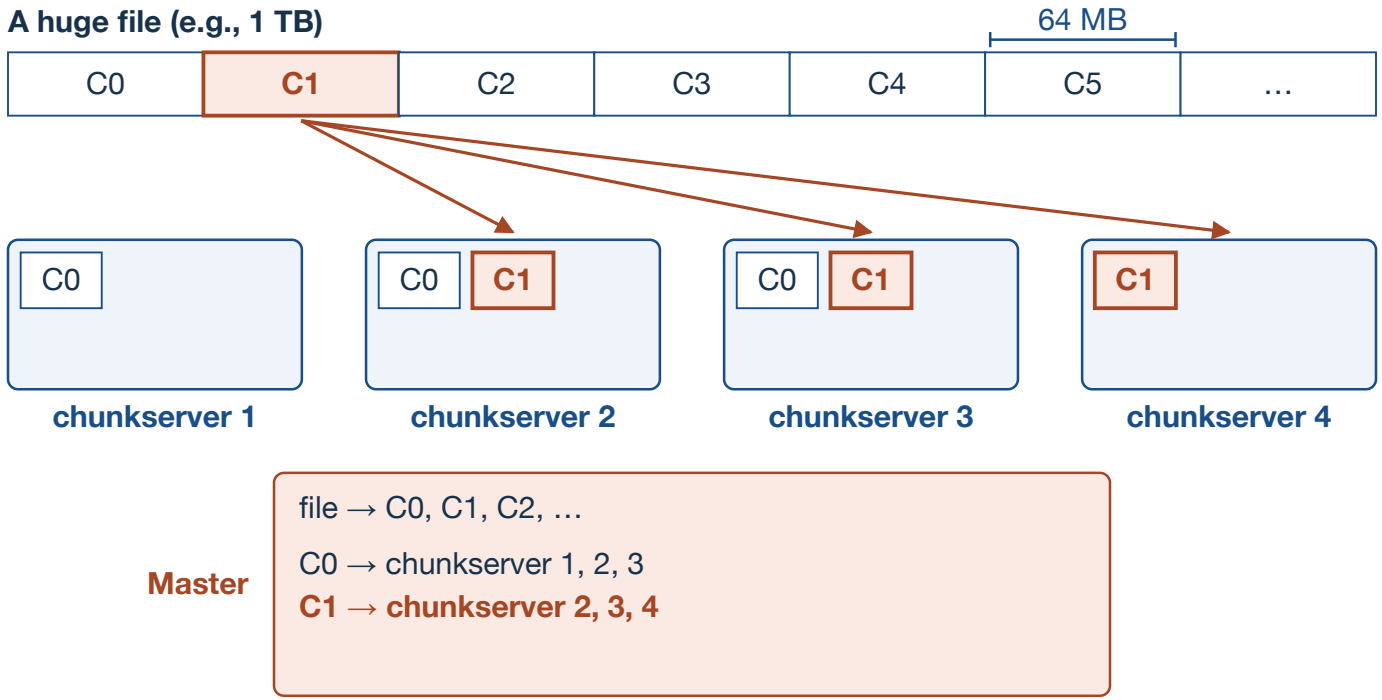
- Capacity: a file is split into 64 MB chunks, stored across many chunkservers
- Fault tolerance: each chunk is replicated on three chunkservers
- One master keeps the metadata; clients read and write data directly with chunkservers



Review: GFS

GFS removes the one-server limitation of NFS

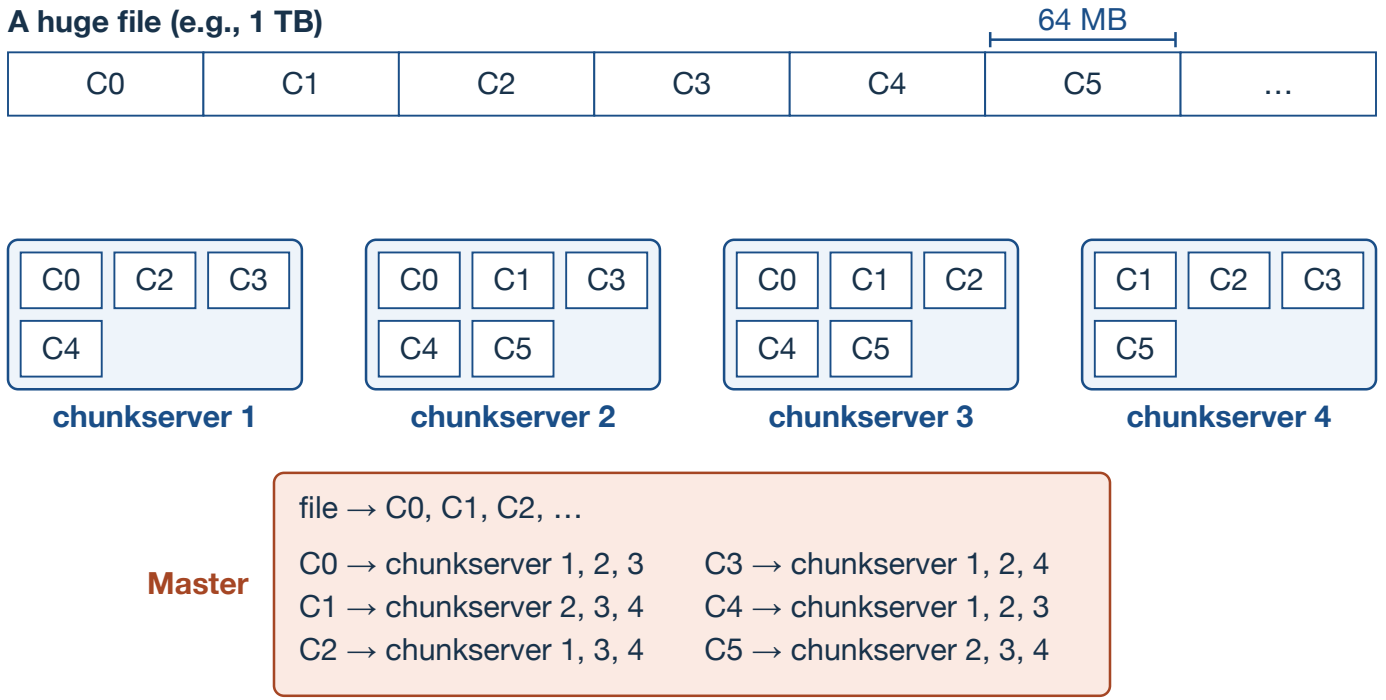
- Capacity: a file is split into 64 MB chunks, stored across many chunkservers
- Fault tolerance: each chunk is replicated on three chunkservers
- One master keeps the metadata; clients read and write data directly with chunkservers



Review: GFS

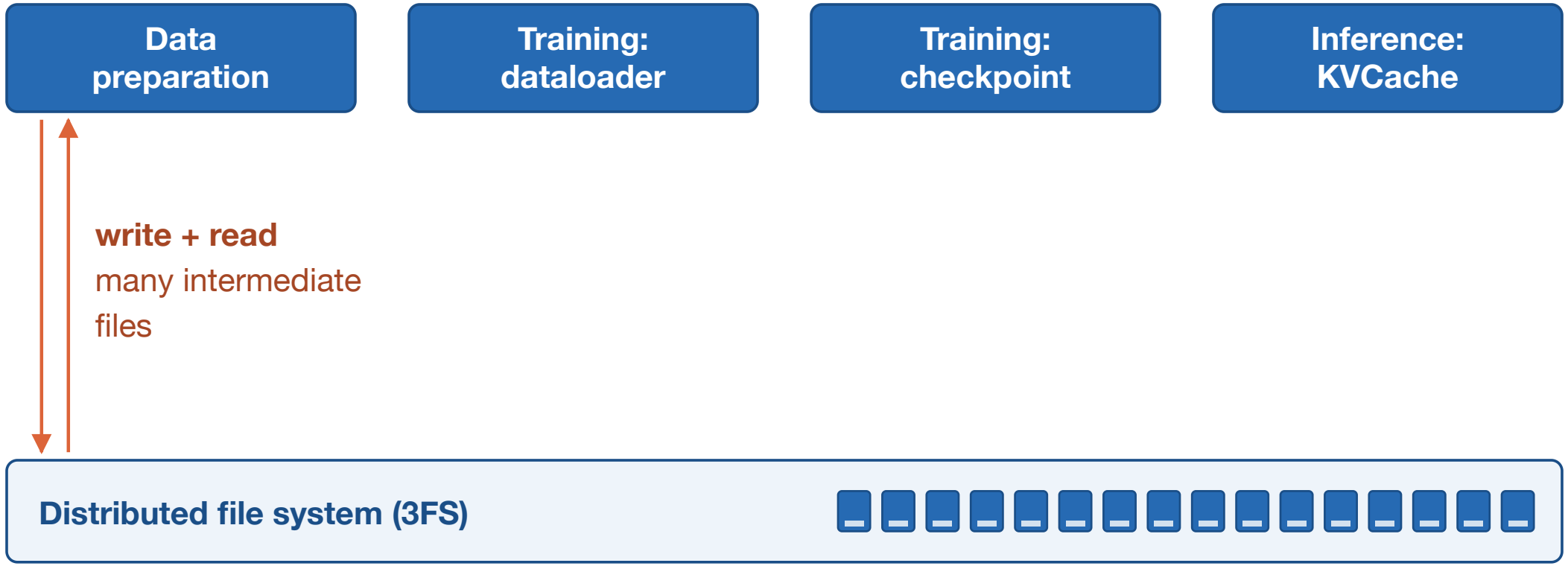
GFS removes the one-server limitation of NFS

- Capacity: a file is split into 64 MB chunks, stored across many chunkservers
- Fault tolerance: each chunk is replicated on three chunkservers
- One master keeps the metadata; clients read and write data directly with chunkservers



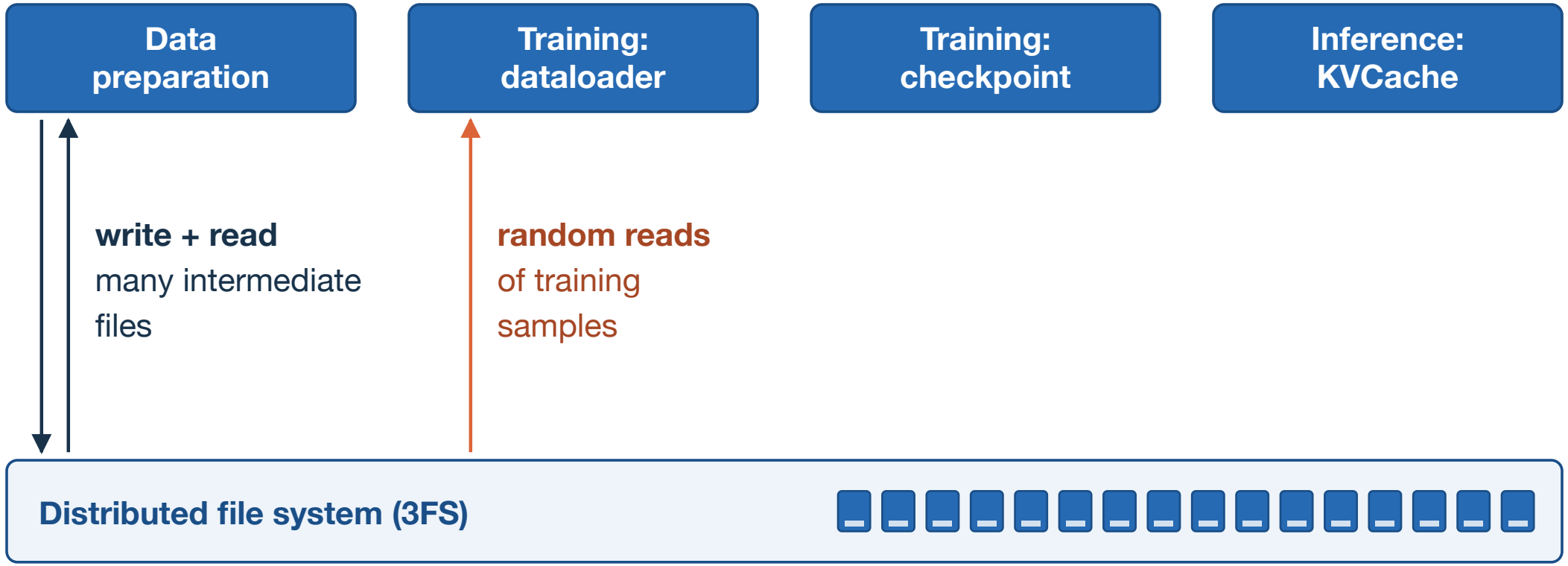
Files behind AI workloads

Training and inference keep their data as files on one shared distributed file system



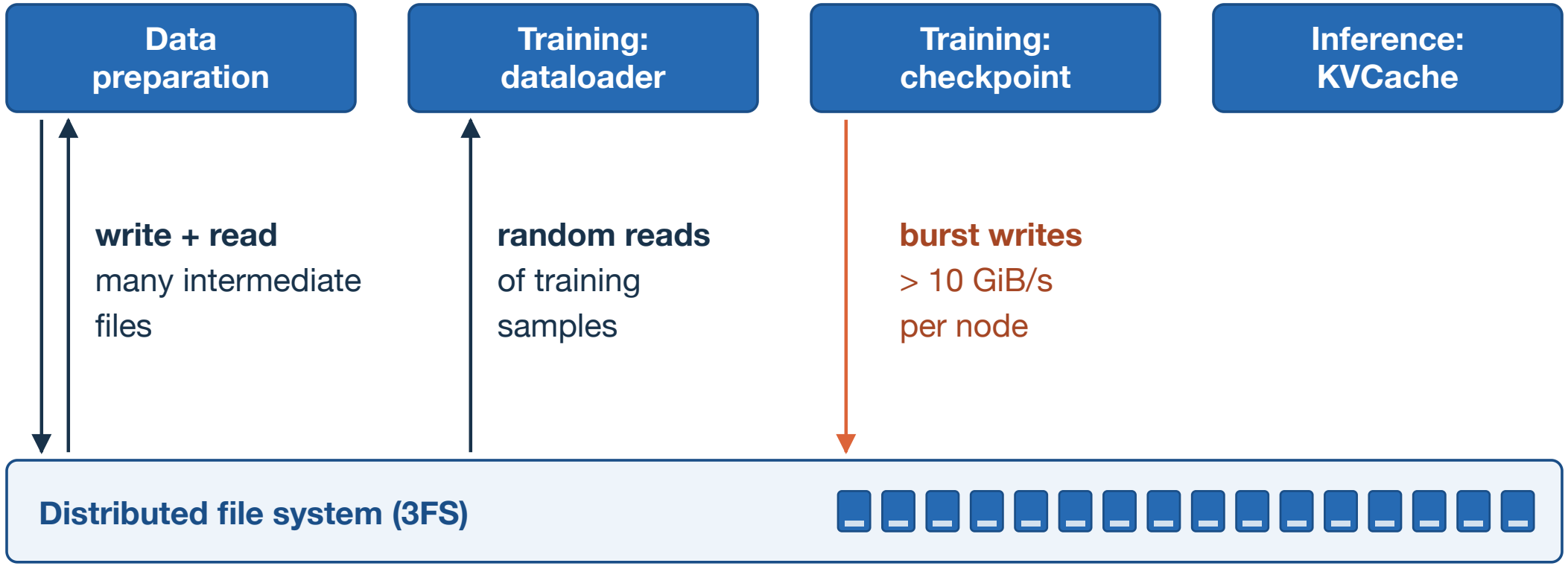
Files behind AI workloads

Training and inference keep their data as files on one shared distributed file system



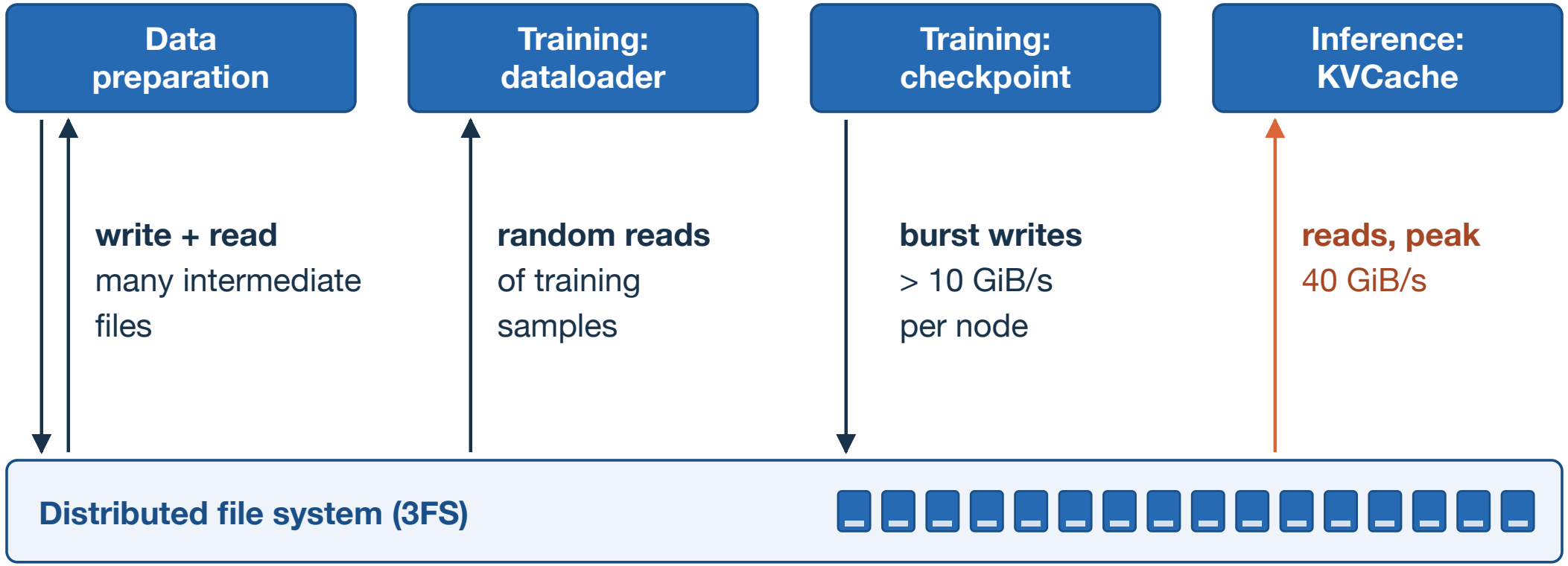
Files behind AI workloads

Training and inference keep their data as files on one shared distributed file system



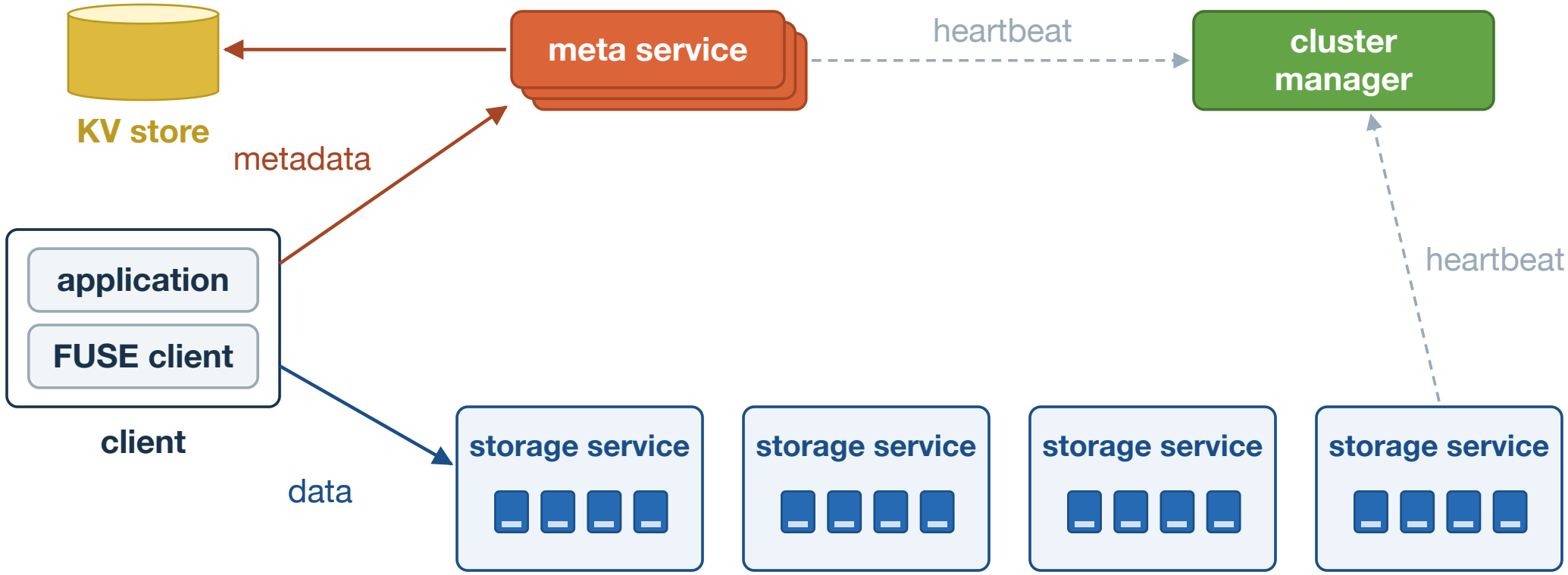
Files behind AI workloads

Training and inference keep their data as files on one shared distributed file system



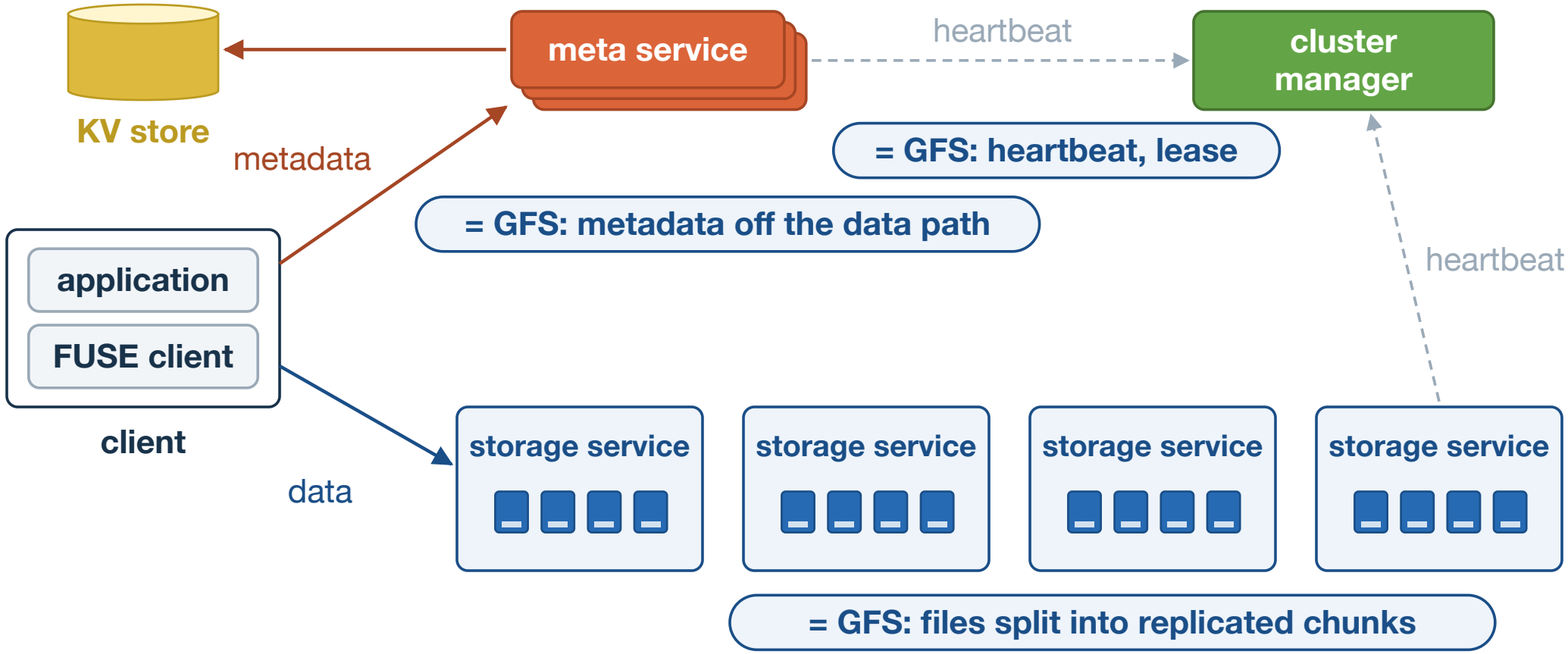
3FS keeps many designs of GFS and NFS

Four roles: cluster manager, meta service, storage service and client



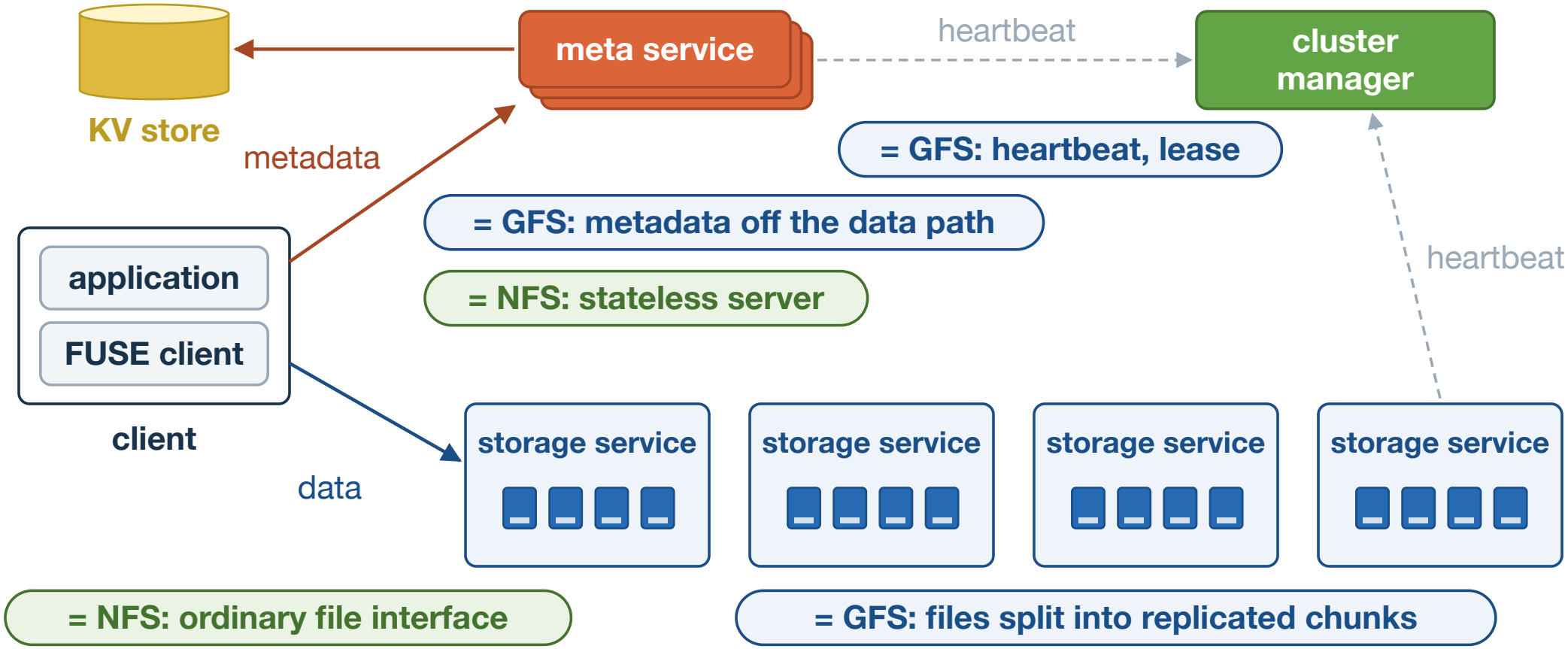
3FS keeps many designs of GFS and NFS

Four roles: cluster manager, meta service, storage service and client



3FS keeps many designs of GFS and NFS

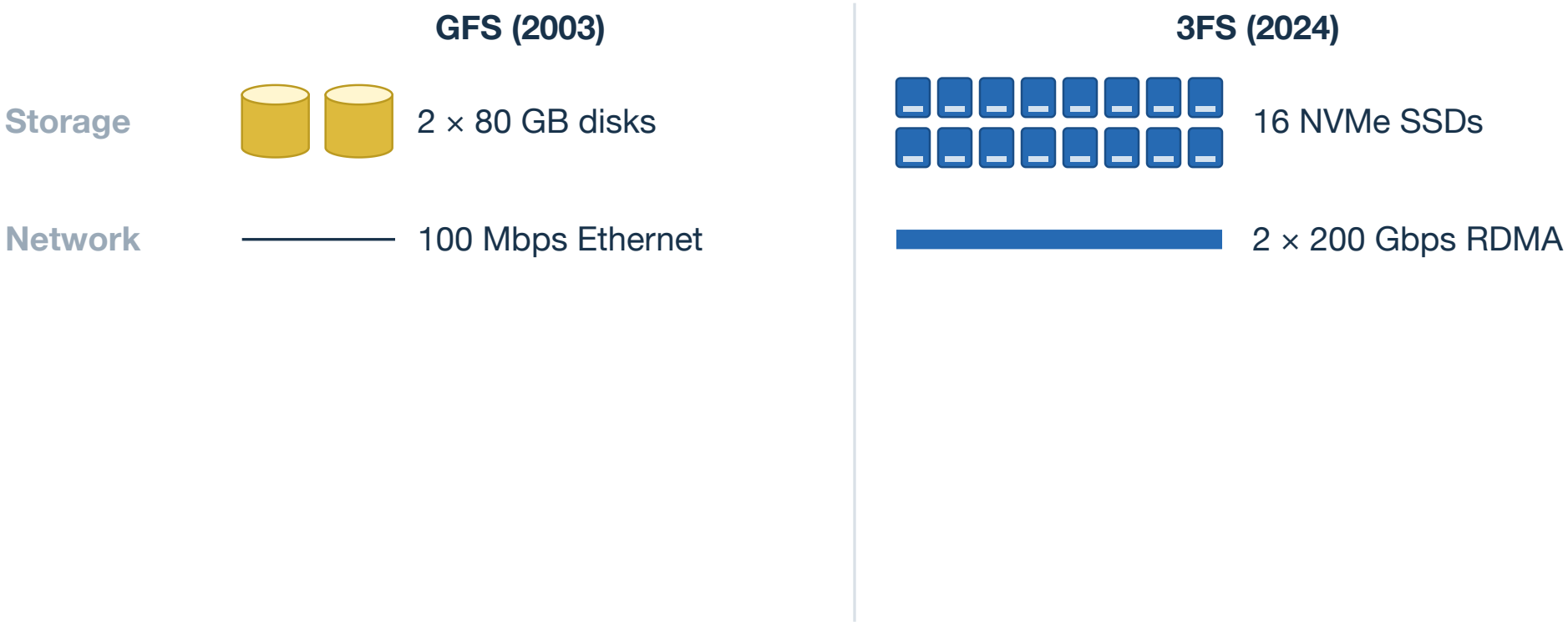
Four roles: cluster manager, meta service, storage service and client



What has changed since GFS?

Hardware per storage node: disks → NVMe SSDs; network 100 Mbps → 2 × 200 Gbps (4000×)

Workload: a few huge files read sequentially → many small files read randomly[1]

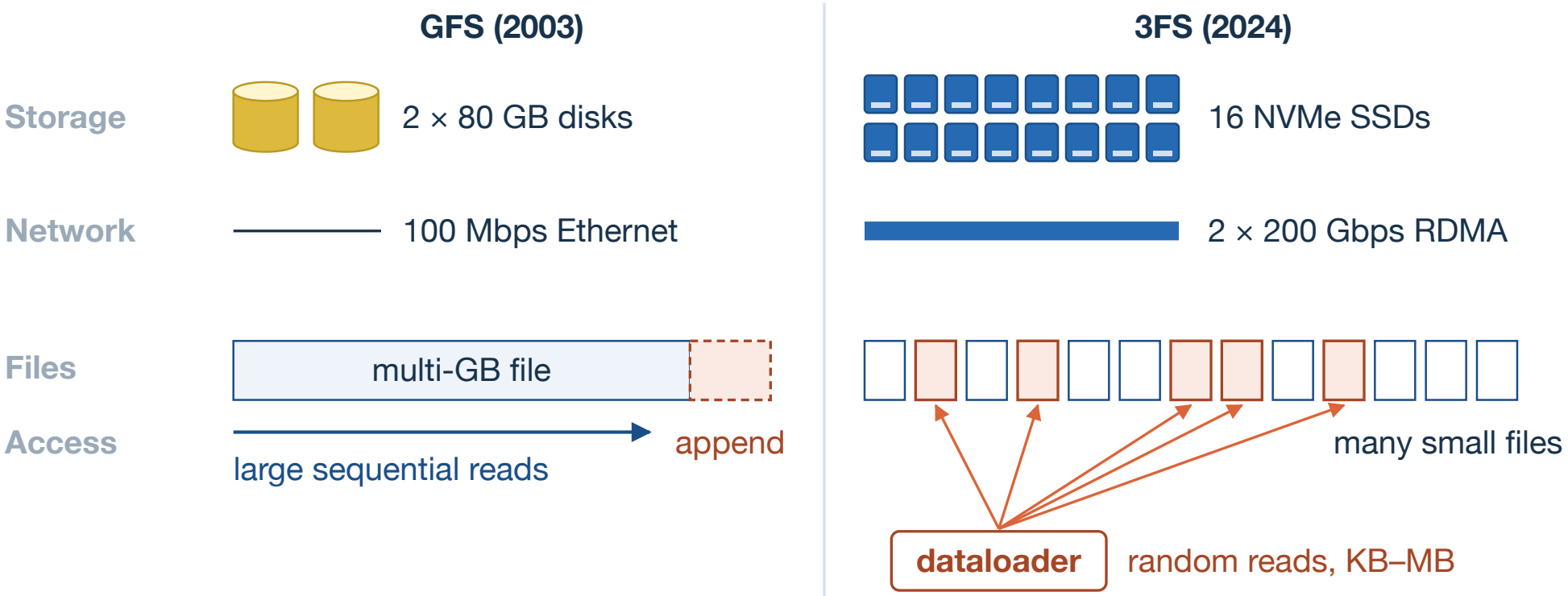


[1] 3FS design notes: "a large number of small files"; "random access to dataset samples ... a few kilobytes to several megabytes"

What has changed since GFS?

Hardware per storage node: disks → NVMe SSDs; network 100 Mbps → 2 × 200 Gbps (4000×)

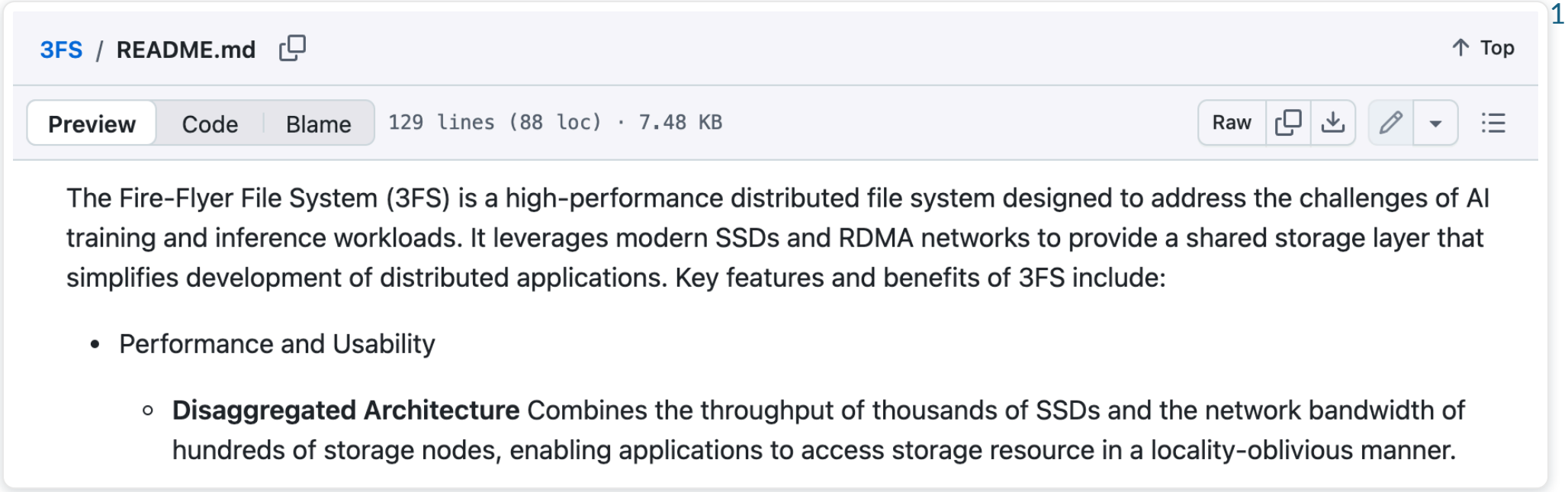
Workload: a few huge files read sequentially → many small files read randomly[1]



[1] 3FS design notes: "a large number of small files"; "random access to dataset samples ... a few kilobytes to several megabytes"

What does locality-oblivious mean?

The 3FS README: a disaggregated architecture, accessed in a locality-oblivious manner



3FS / README.md ↑ Top

Preview Code Blame 129 lines (88 loc) · 7.48 KB Raw

The Fire-Flyer File System (3FS) is a high-performance distributed file system designed to address the challenges of AI training and inference workloads. It leverages modern SSDs and RDMA networks to provide a shared storage layer that simplifies development of distributed applications. Key features and benefits of 3FS include:

- Performance and Usability
  - **Disaggregated Architecture** Combines the throughput of thousands of SSDs and the network bandwidth of hundreds of storage nodes, enabling applications to access storage resource in a locality-oblivious manner.

[1]

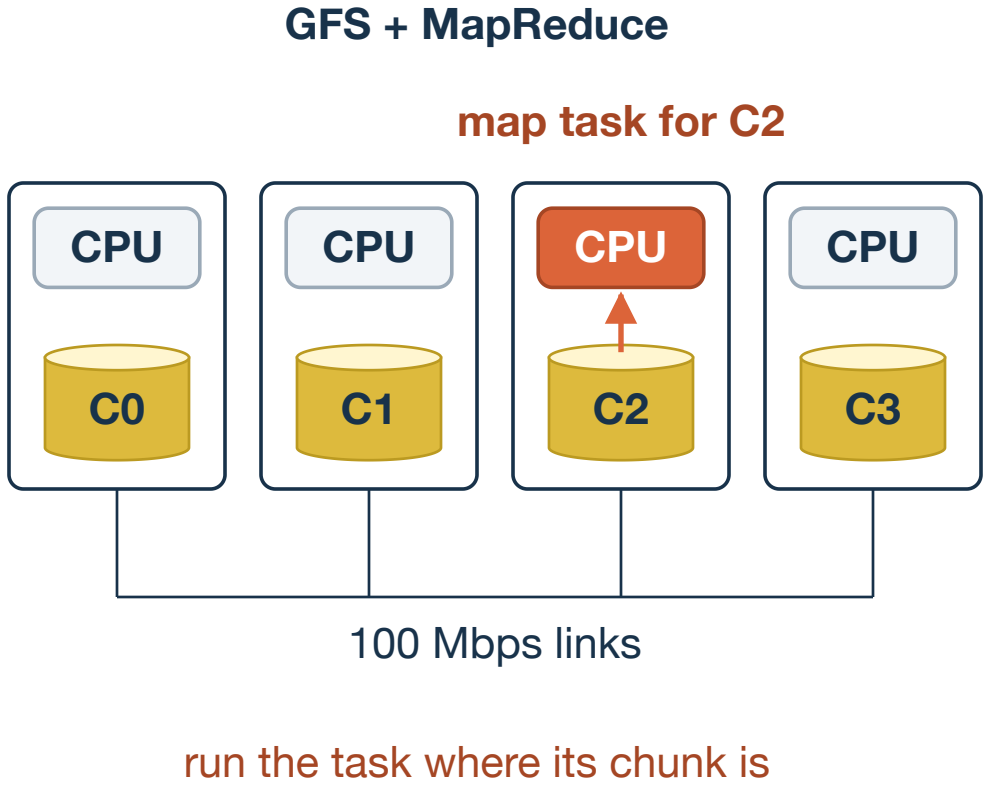
Disaggregated: GPU nodes and storage nodes are separate machines

Locality-oblivious: a client ignores where data lives; no task is placed near its data

[1] Source: https://github.com/deepseek-ai/3FS

Why locality-oblivious?

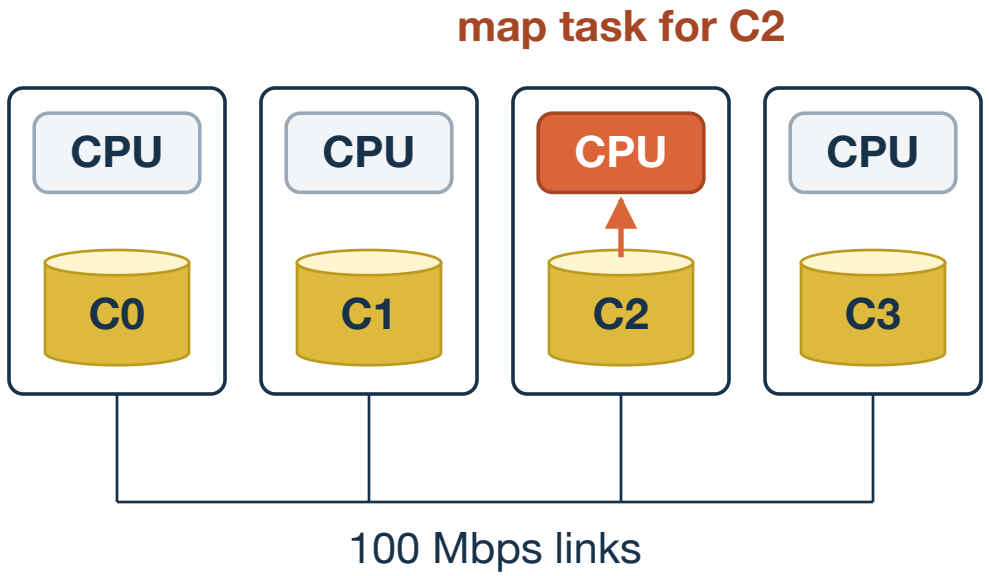
MapReduce: “network bandwidth is a relatively scarce resource” → run a task where its data is



Why locality-oblivious?

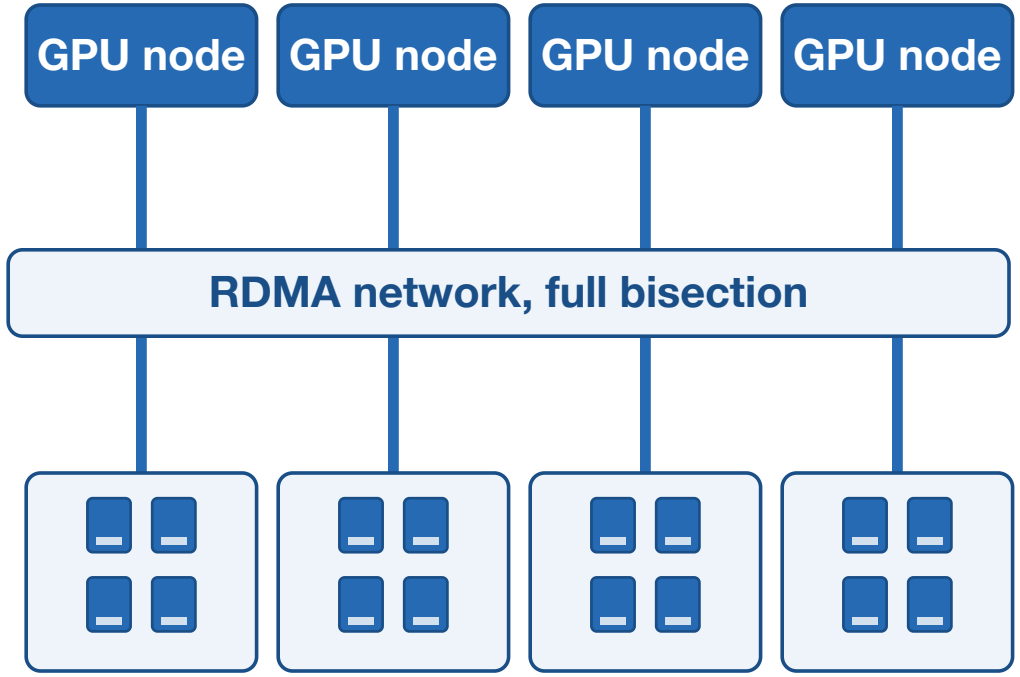
MapReduce: “network bandwidth is a relatively scarce resource” → run a task where its data is

GFS + MapReduce



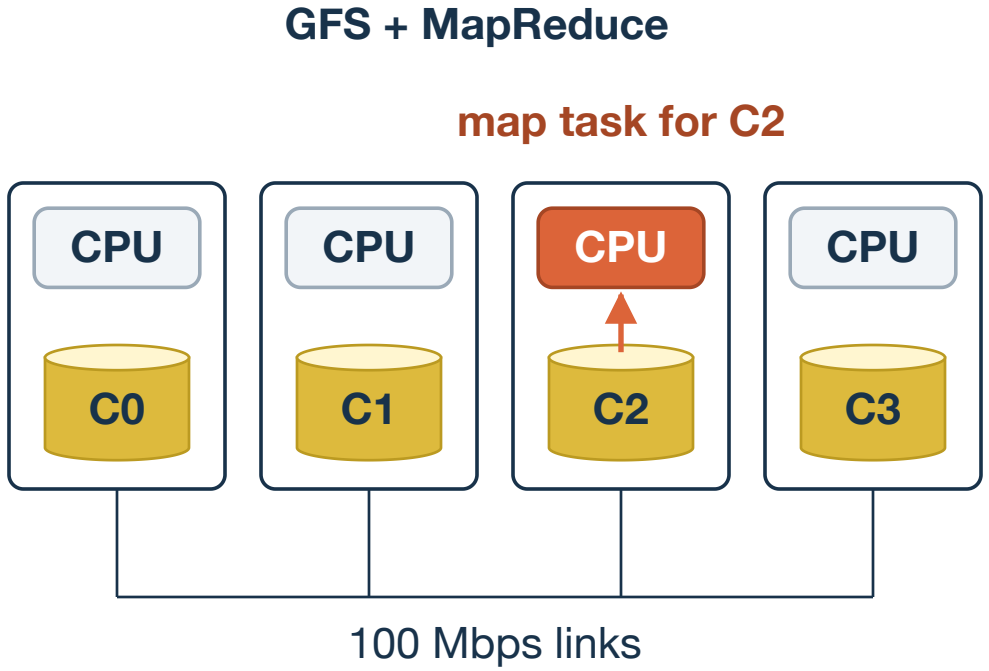
run the task where its chunk is

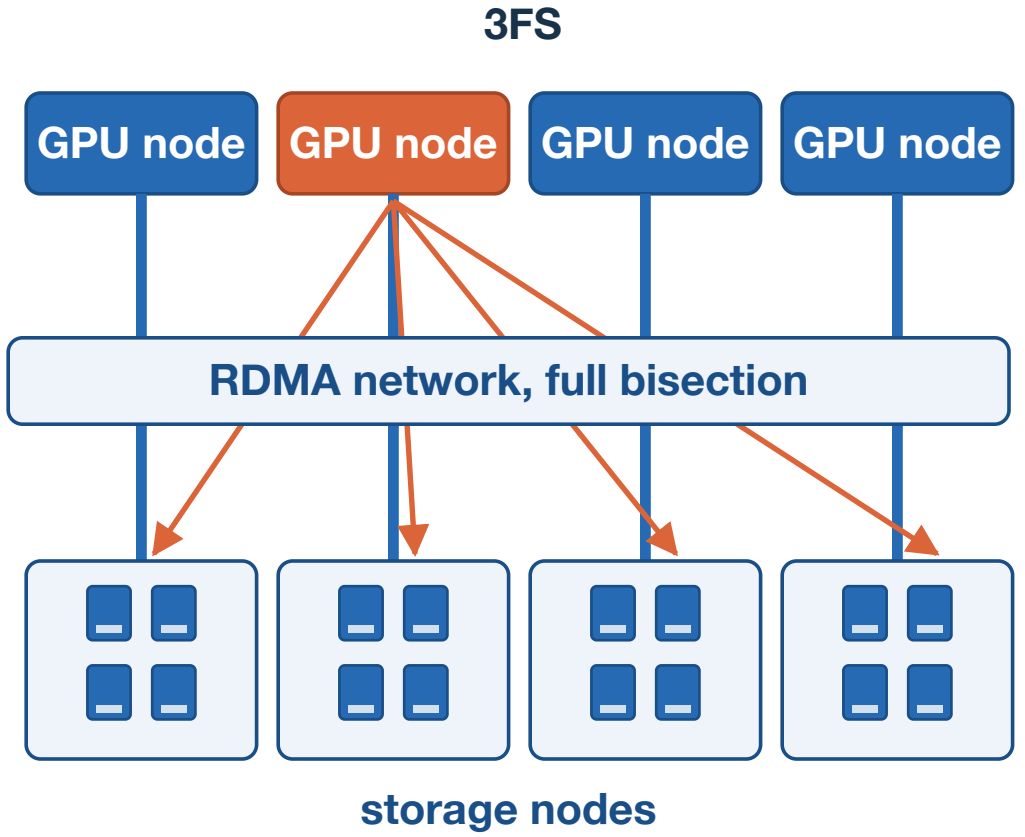
3FS



Why locality-oblivious?

MapReduce: “network bandwidth is a relatively scarce resource” → run a task where its data is





3FS: any client reads any SSD at full speed → GPUs and storage scale separately

Where does 3FS keep metadata?

The Fire-Flyer paper: metadata lives in a distributed key-value store; meta services keep no state of their own

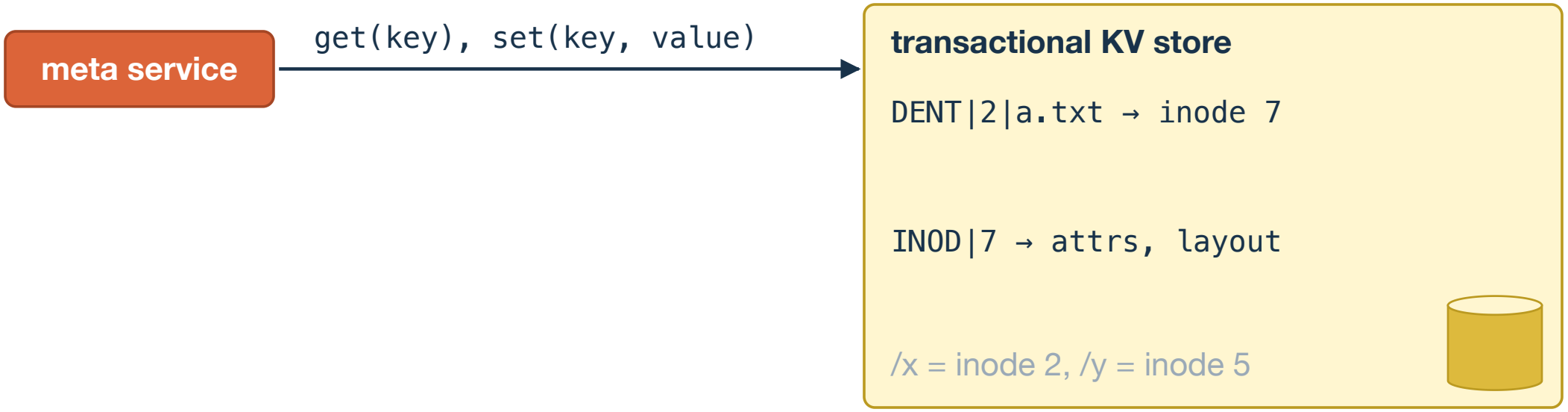
> File system meta data are stored in tables of a distributed key-value storage system. Each file or directory has a unique inode ID. The File inode/directory ID and meta data, such as file size and location information of the file content data, are stored as key-value pairs in the inode table. A separate directory entry table stores key-value pairs of $(parent_dir_inode_id, entry_name)$: $(entry_inode_id, ...)$ to support iterating entries in a directory and resolving file/directory paths. All states of meta services are persisted on the distributed key-value storage system. Several meta services run concurrently to handle meta requests from clients.

The store is FoundationDB, a transactional key-value store

- Whose design is quite complicated, and will be covered in later lectures

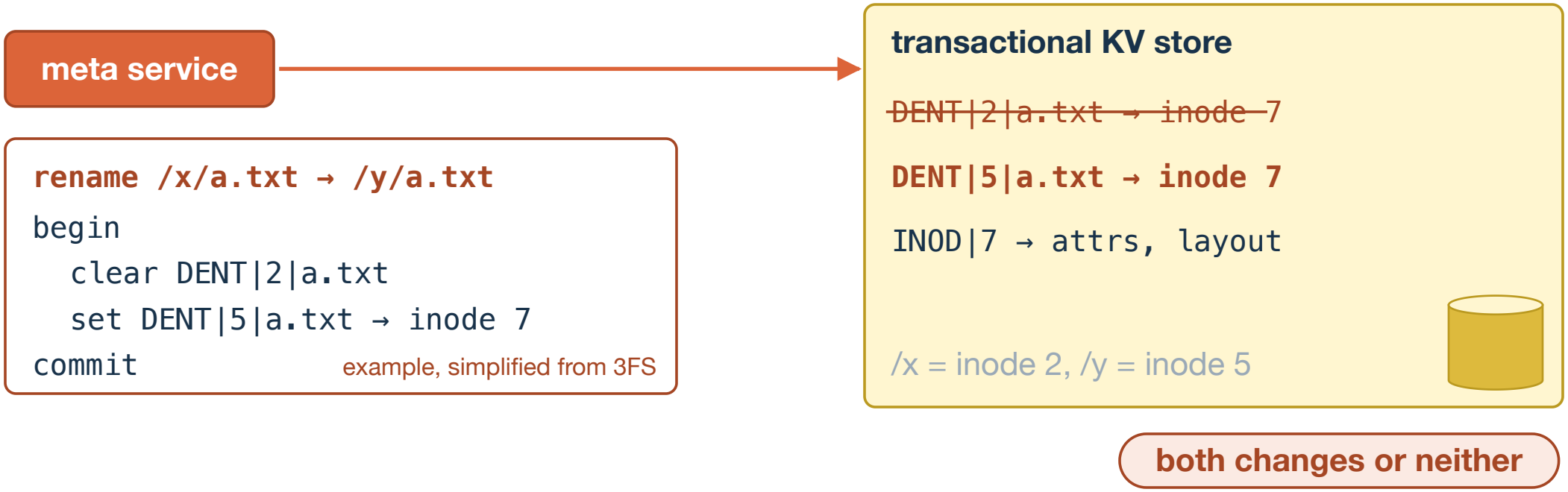
What is a transactional key-value store?

Transaction: several reads and writes, applied **all or nothing** (see later lectures)



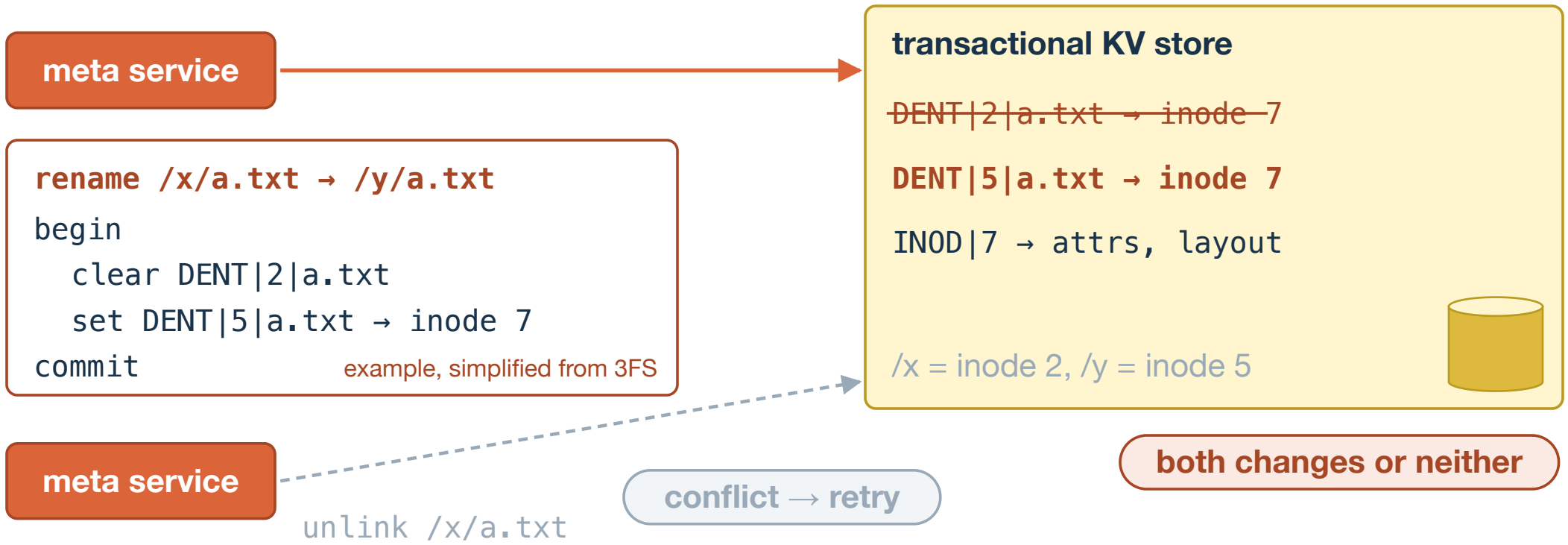
What is a transactional key-value store?

Transaction: several reads and writes, applied **all or nothing** (see later lectures)



What is a transactional key-value store?

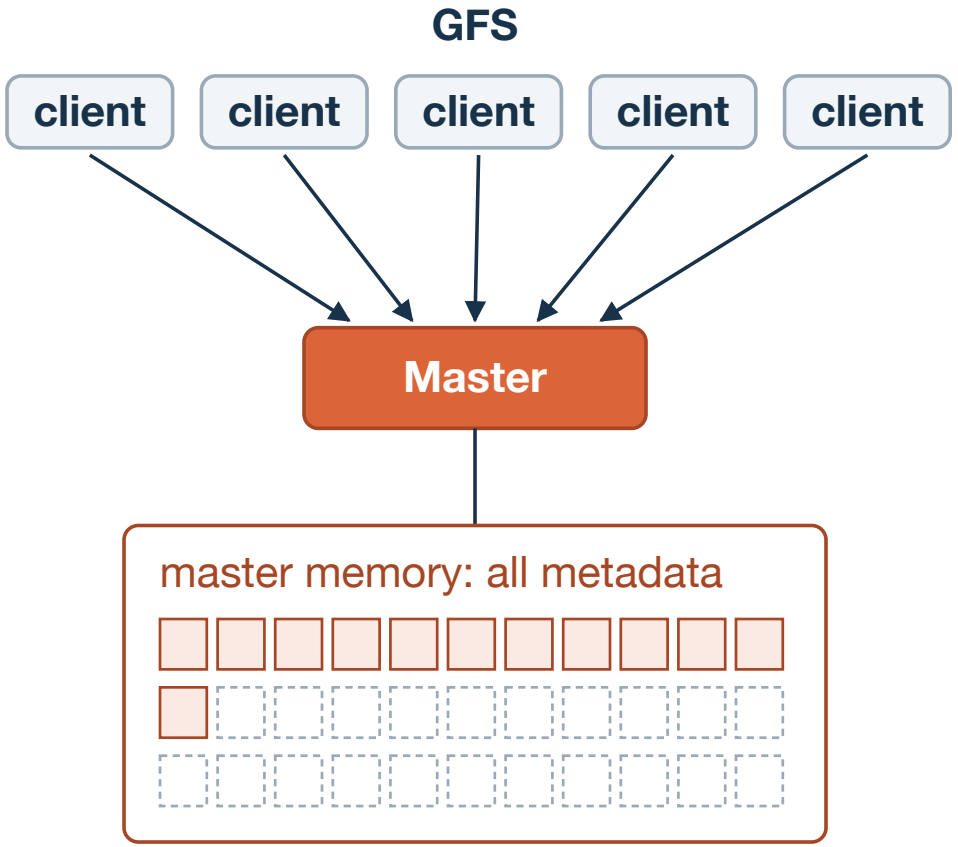
Transaction: several reads and writes, applied **all or nothing** (see later lectures)



3FS keeps inodes and directory entries as key-value pairs in FoundationDB

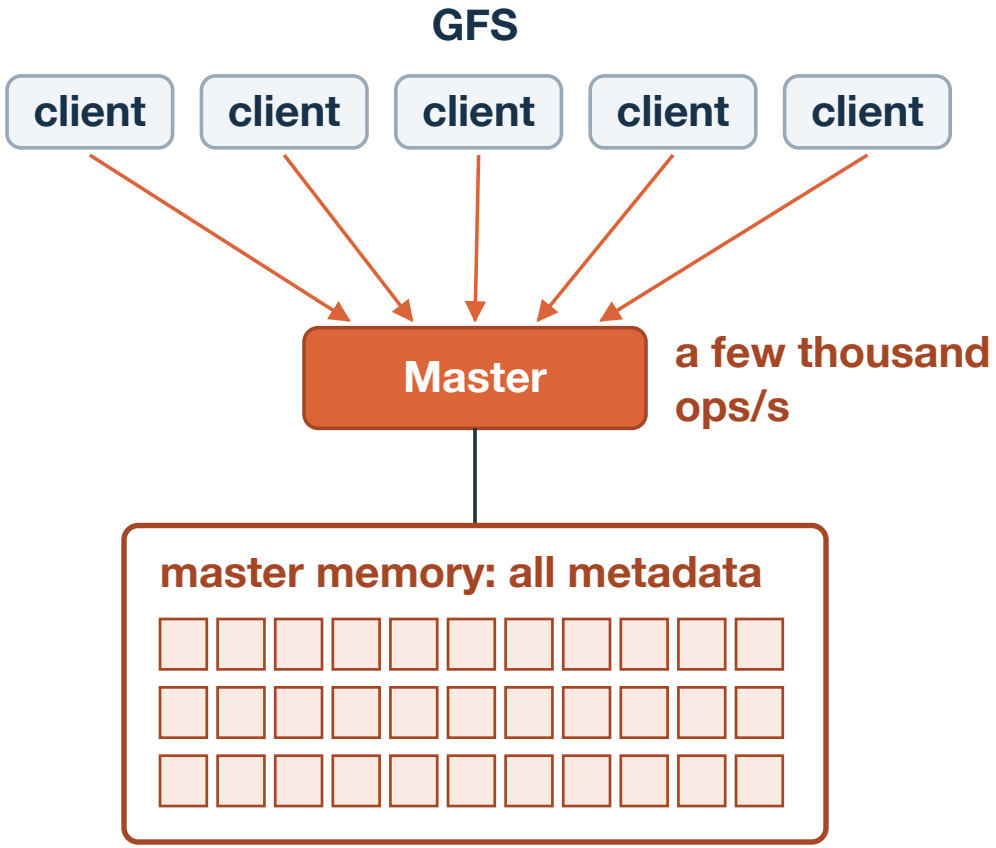
Why not a single master?

GFS: one master keeps all metadata in memory; many files and many clients make it the limit



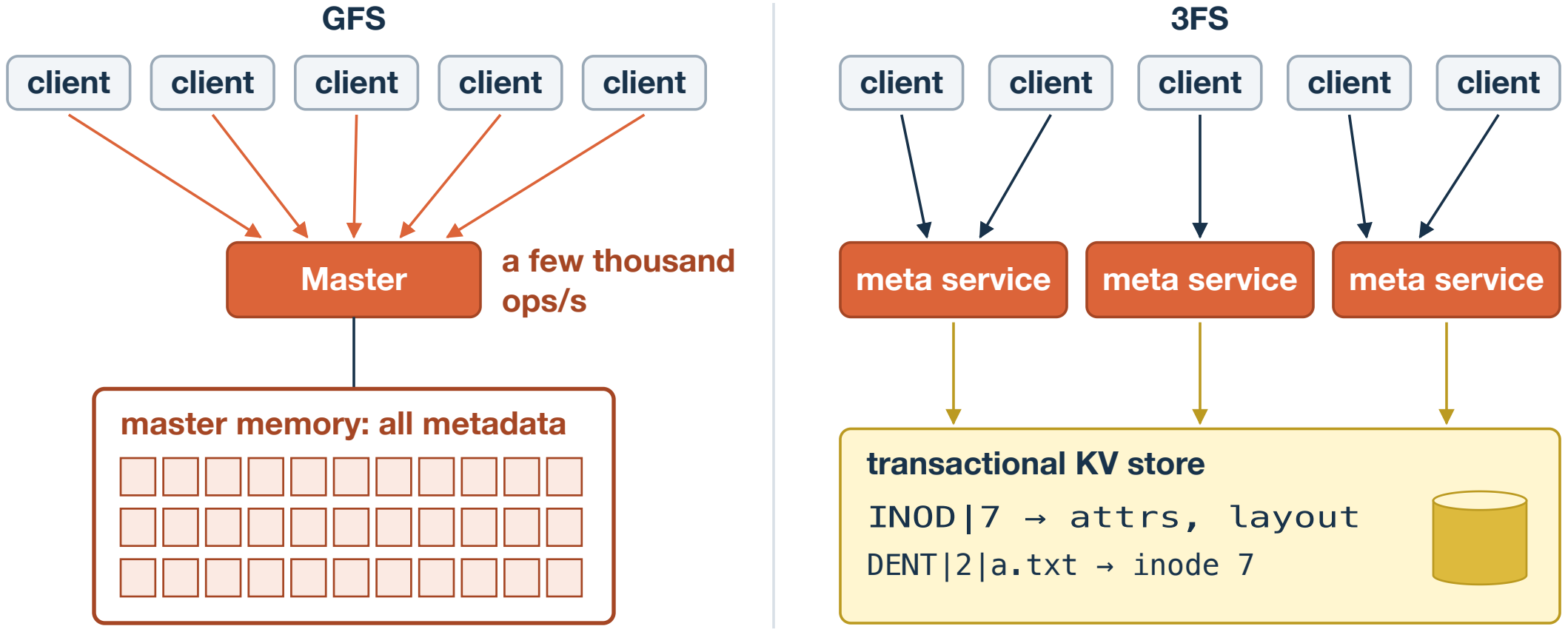
Why not a single master?

GFS: one master keeps all metadata in memory; many files and many clients make it the limit



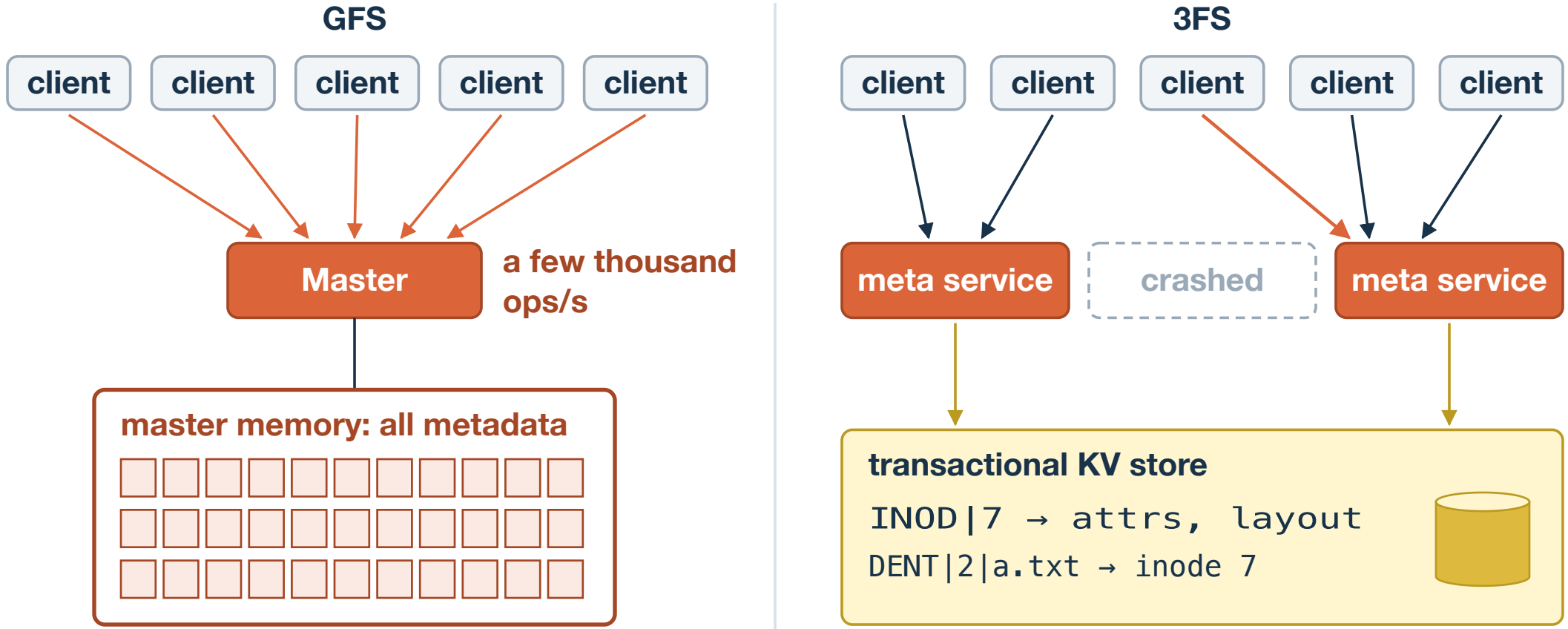
Why not a single master?

GFS: one master keeps all metadata in memory; many files and many clients make it the limit



Why not a single master?

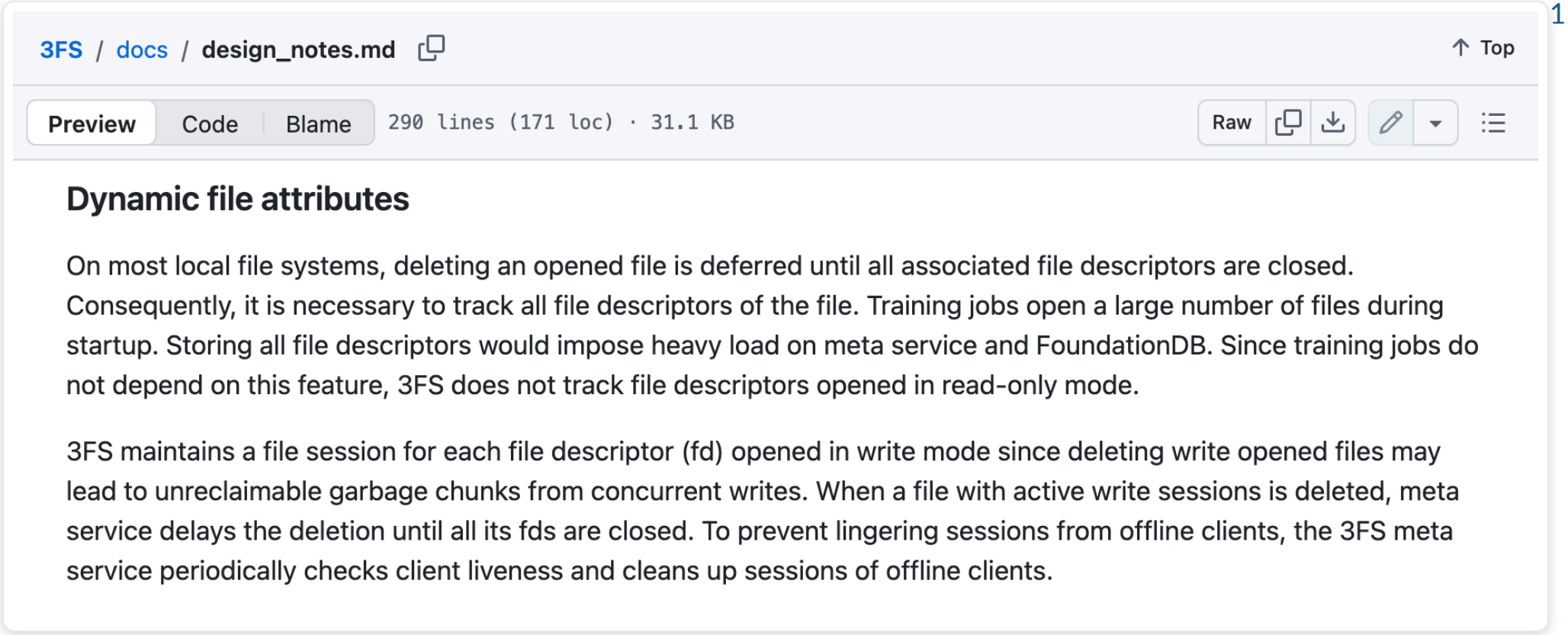
GFS: one master keeps all metadata in memory; many files and many clients make it the limit



`rename` = one KV transaction; meta services keep no state

Which opened files does 3FS track?

The 3FS design notes: only files opened for writing

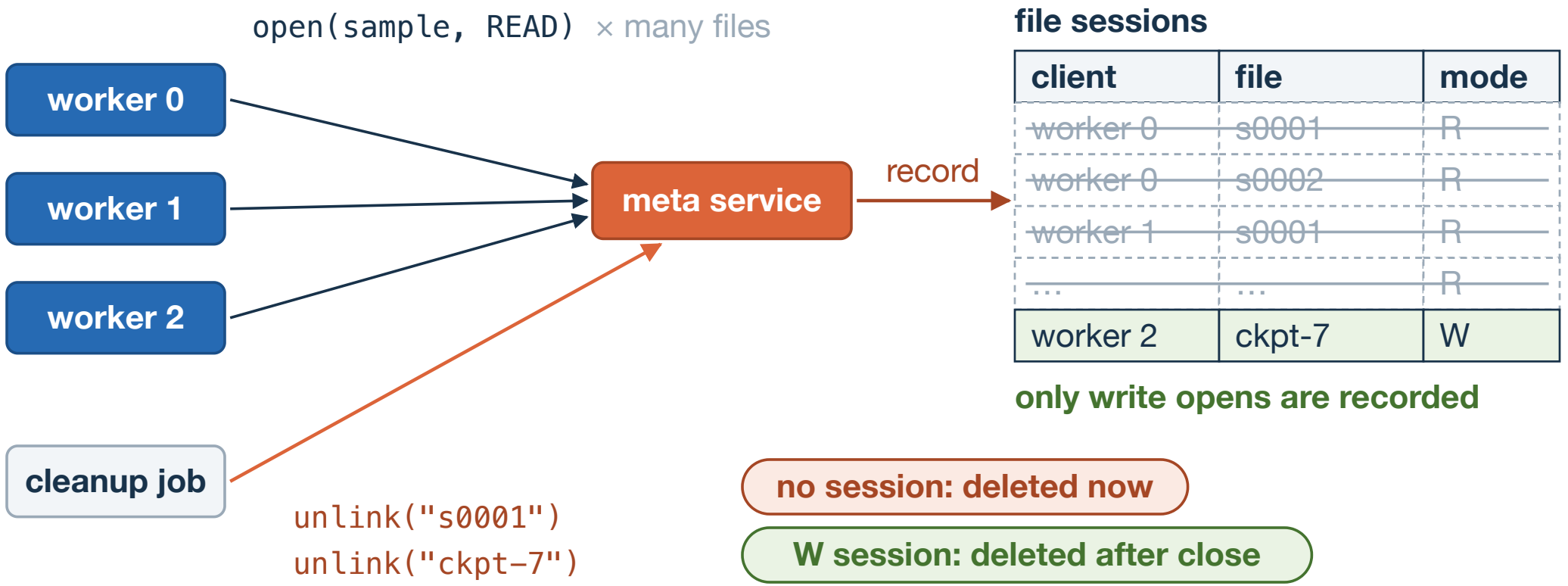


[1]

[1] Source: https://github.com/deepseek-ai/3FS/blob/main/docs/design_notes.md

Why not track opened files?

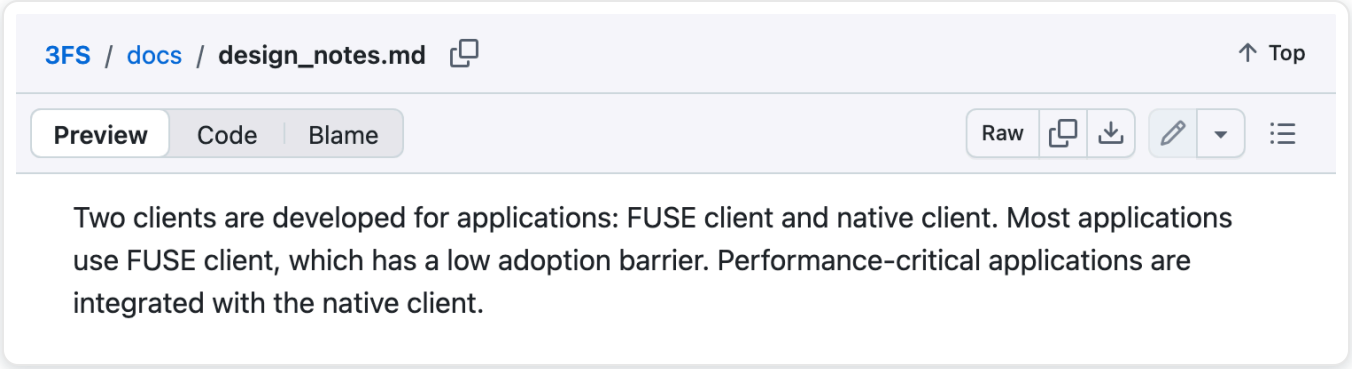
A training job opens a large number of files at startup; recording every open is costly



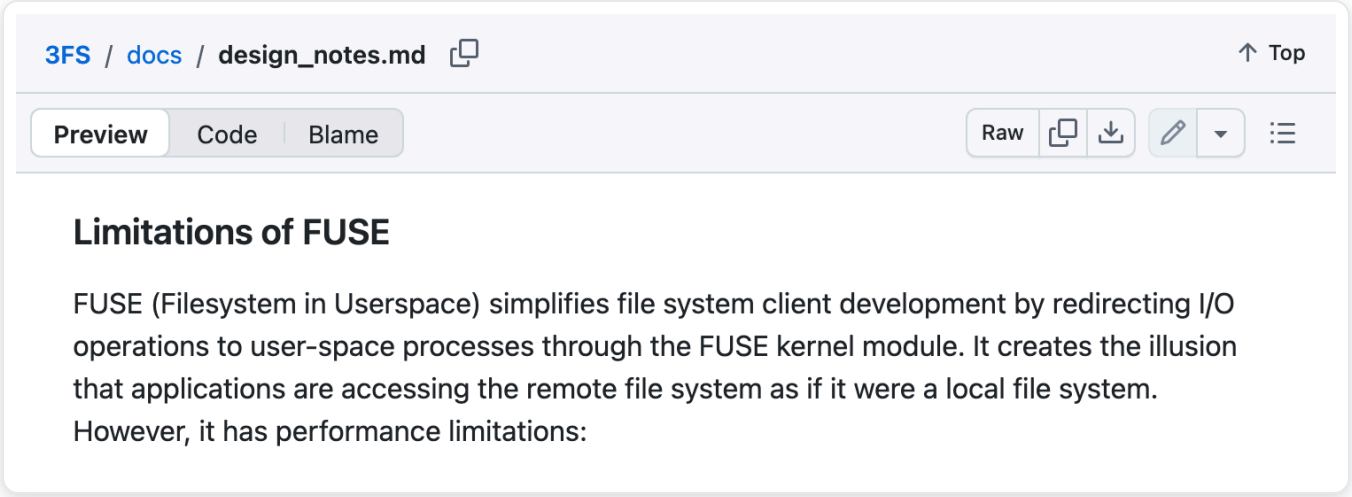
Same choice as NFS (recall: delete after open); NFS: simple crash recovery, 3FS: cheap opens

How do applications access 3FS?

The 3FS design notes describe two clients



and explain why the FUSE client alone is not enough

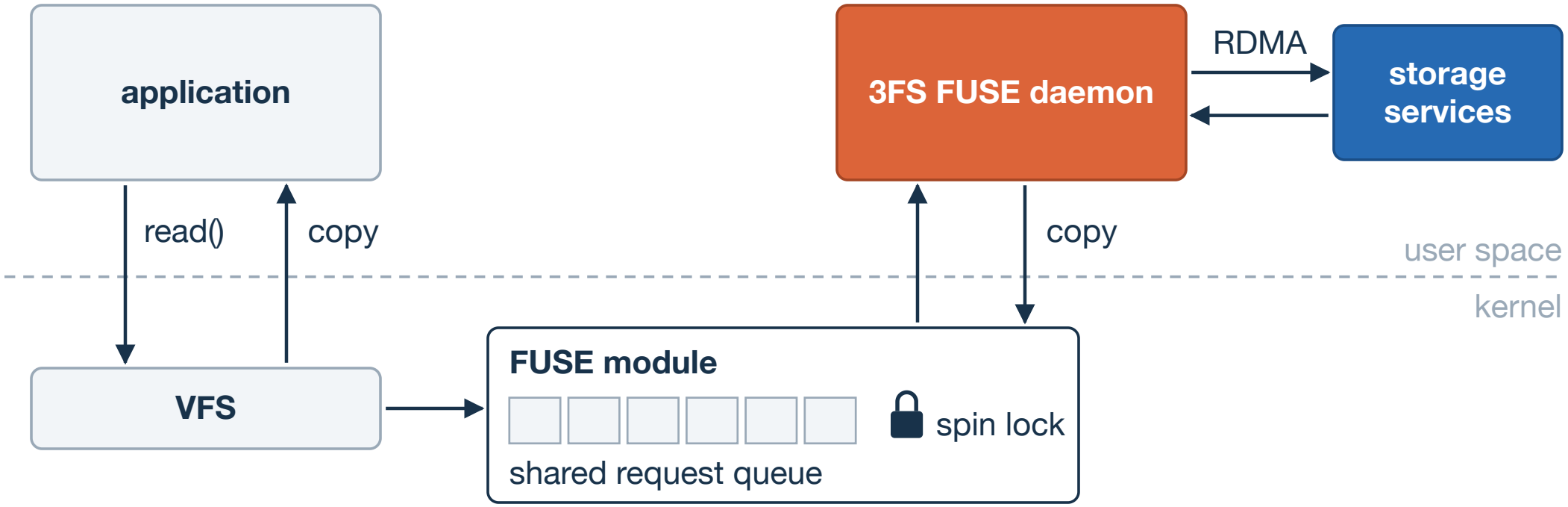


[1]

[1] Source: https://github.com/deepseek-ai/3FS/blob/main/docs/design_notes.md

What is FUSE?

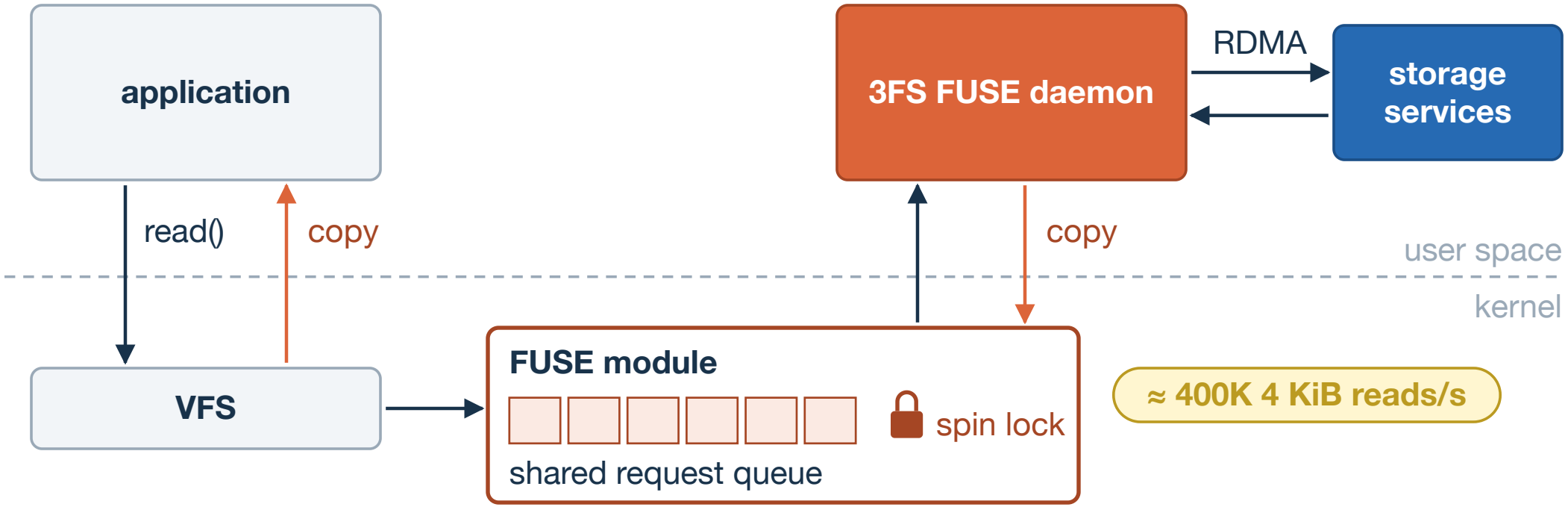
FUSE (Filesystem in Userspace): the kernel forwards file system calls to a user-space daemon, which talks to the storage services



No new API to learn, and no kernel module whose bugs crash the machine

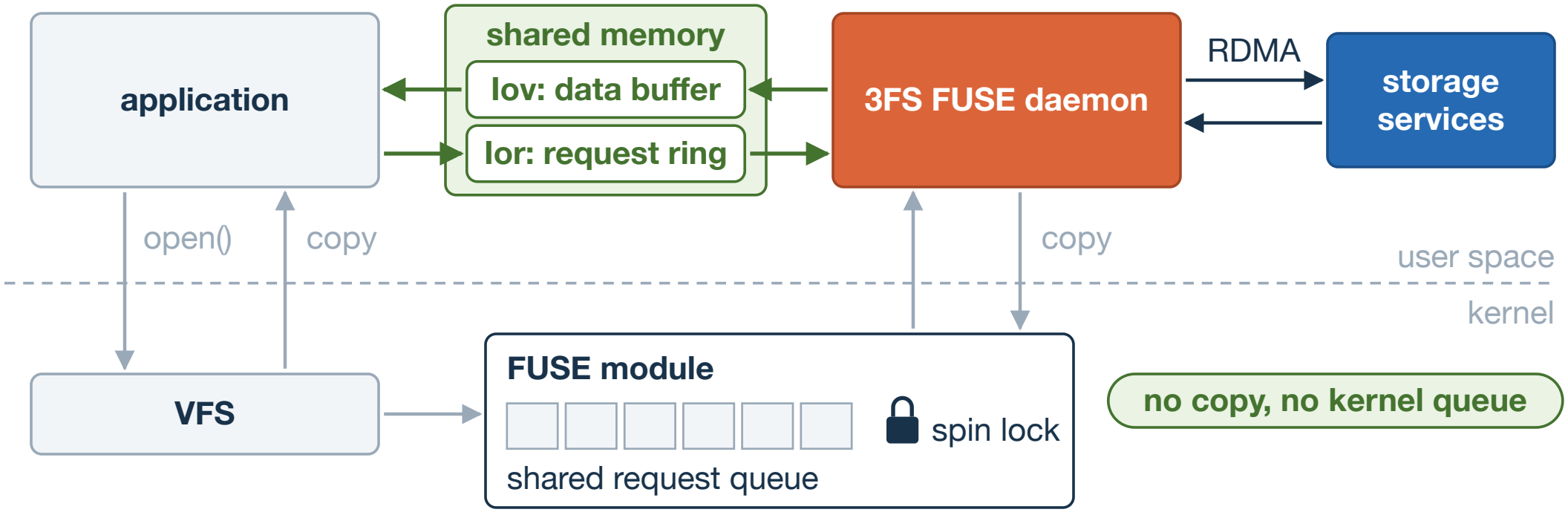
Why is FUSE not enough?

With fast SSDs and RDMA, the FUSE software path becomes the bottleneck: every request is copied across the kernel boundary and waits on one spin lock



Why is FUSE not enough?

With fast SSDs and RDMA, the FUSE software path becomes the bottleneck: every request is copied across the kernel boundary and waits on one spin lock



Native API: `open` via FUSE, then I/O through a shared-memory ring (modeled after `io_uring`)[1]

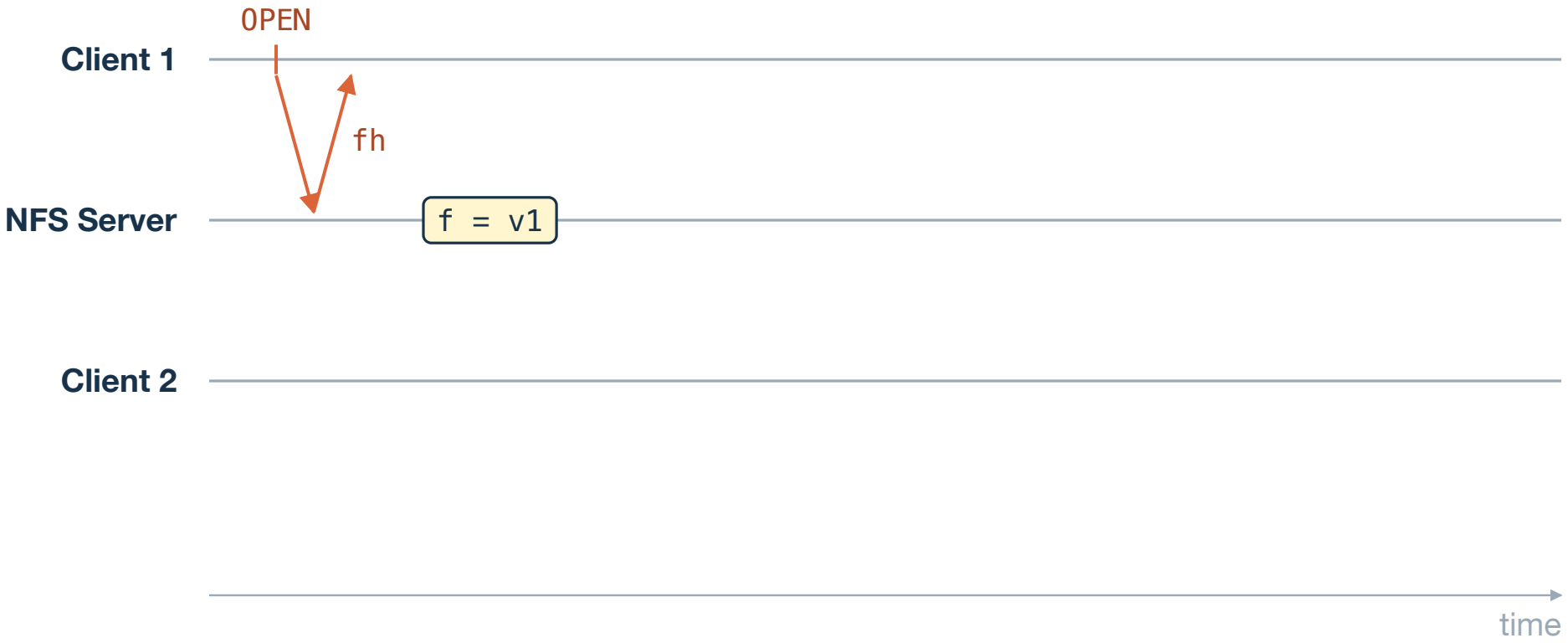
[1] 3FS design notes: "The user-space file system daemon cannot access application memory. Data transfer between kernel and user spaces consumes memory bandwidth and increases end-to-end latency"; "FUSE places these requests into a multi-threaded shared queue, protected by a spin lock … FUSE only handles approximately 400K 4KiB reads per second"; "The asynchronous, zero-copy API is inspired by Linux `io_uring`"

3FS has many more designs

Many are less related to file systems; we skip them

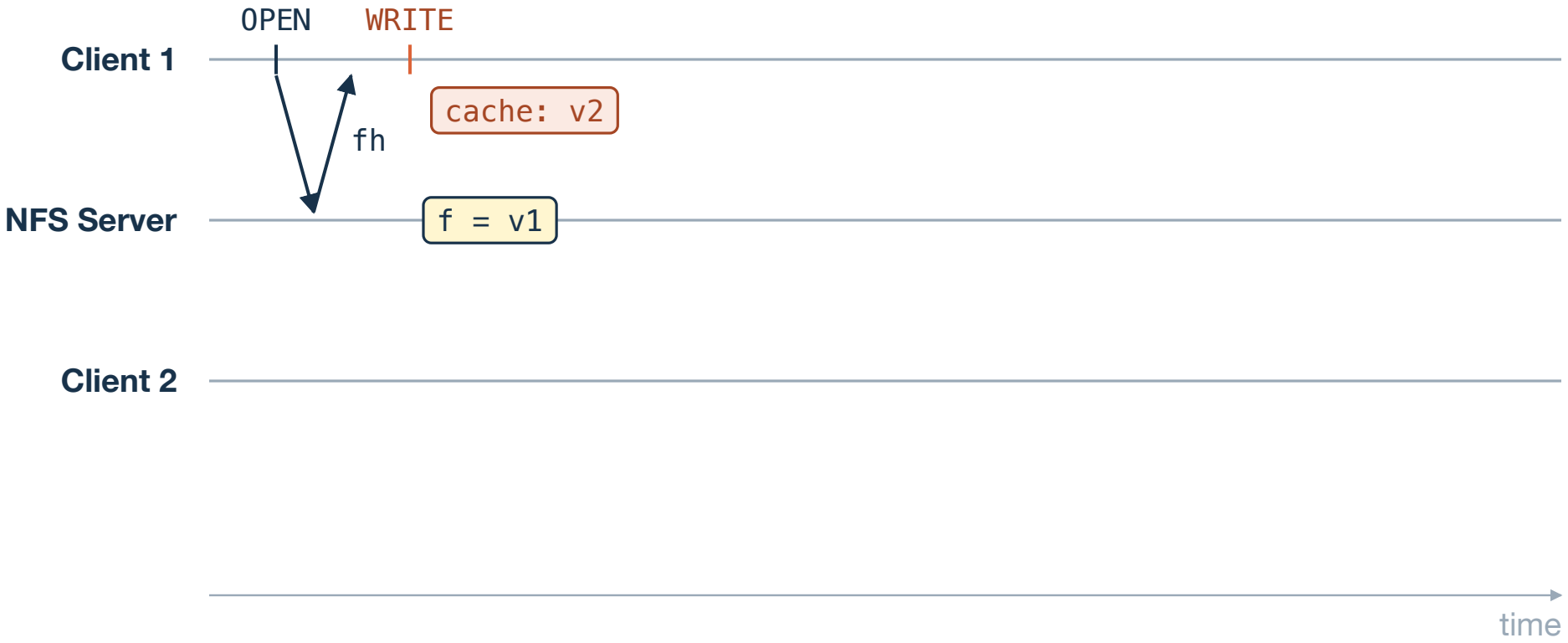
Review: consistency in NFS

Clients cache file data, so a file has several copies: one on the server, one in each client's cache



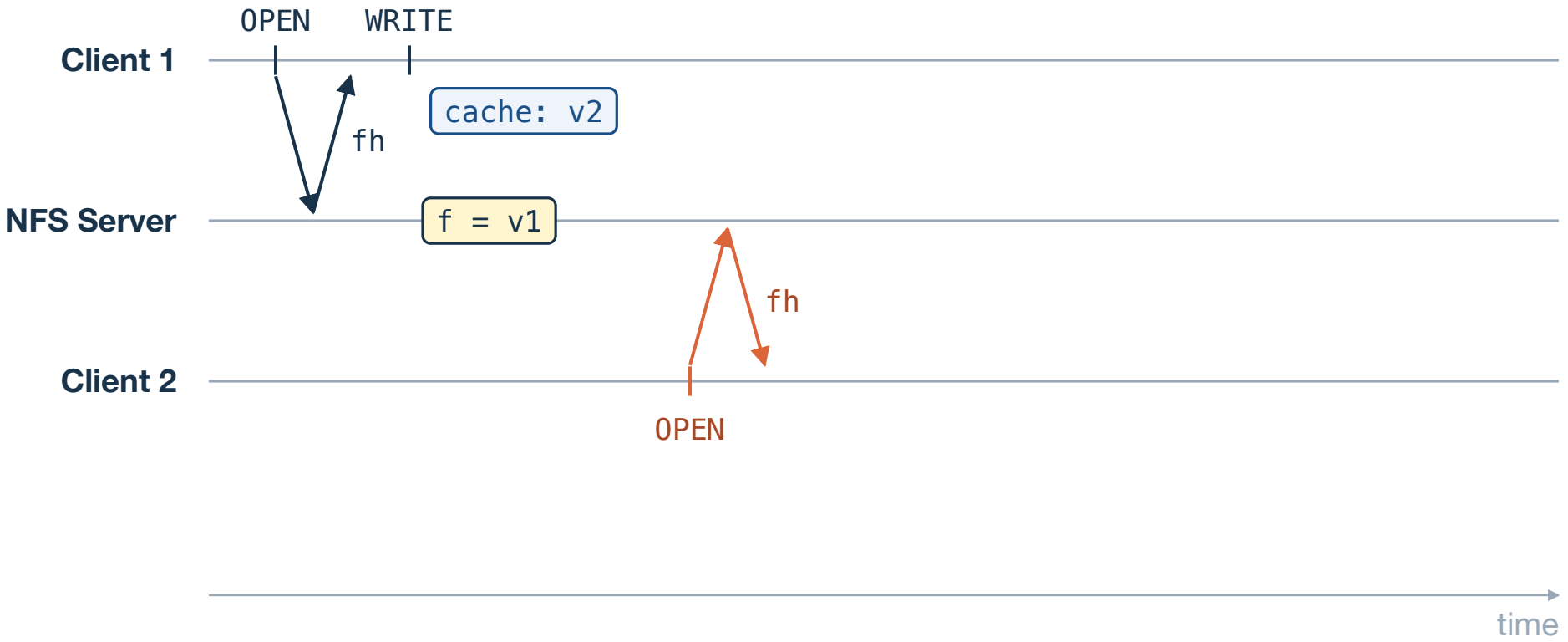
Review: consistency in NFS

Clients cache file data, so a file has several copies: one on the server, one in each client's cache



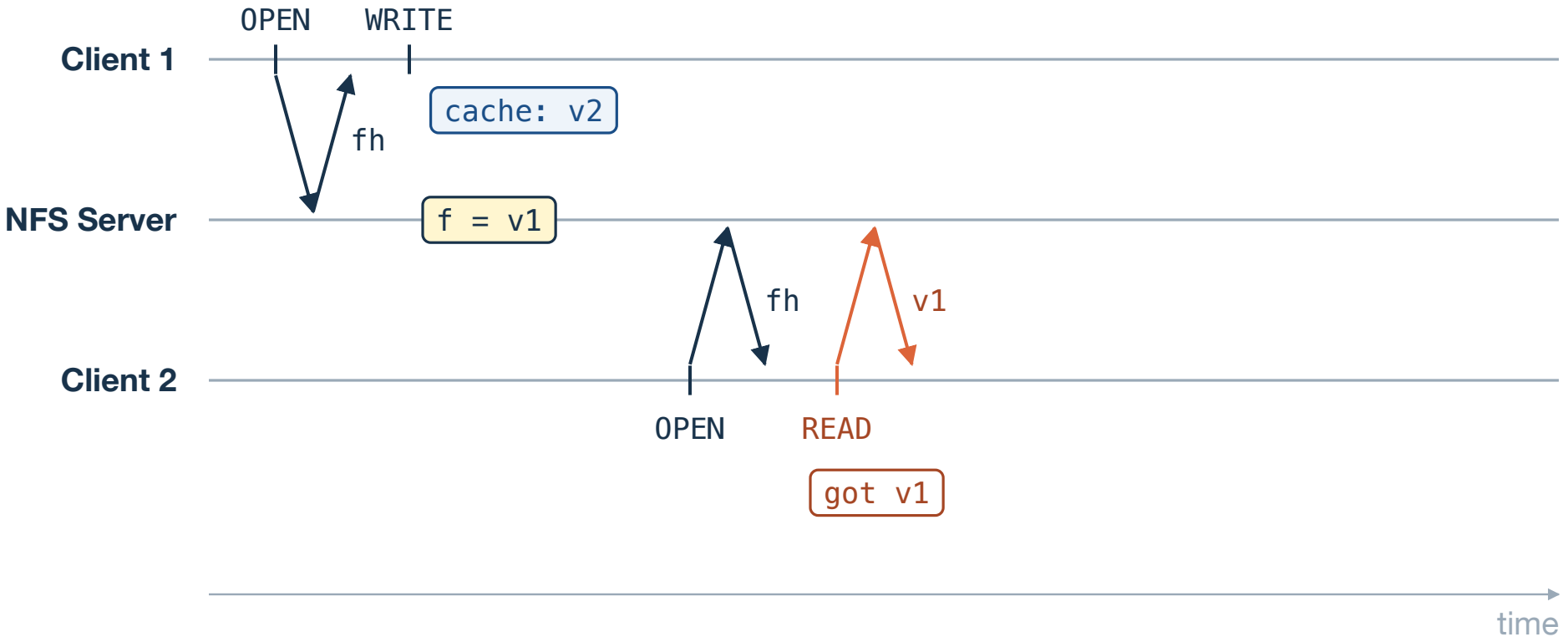
Review: consistency in NFS

Clients cache file data, so a file has several copies: one on the server, one in each client's cache



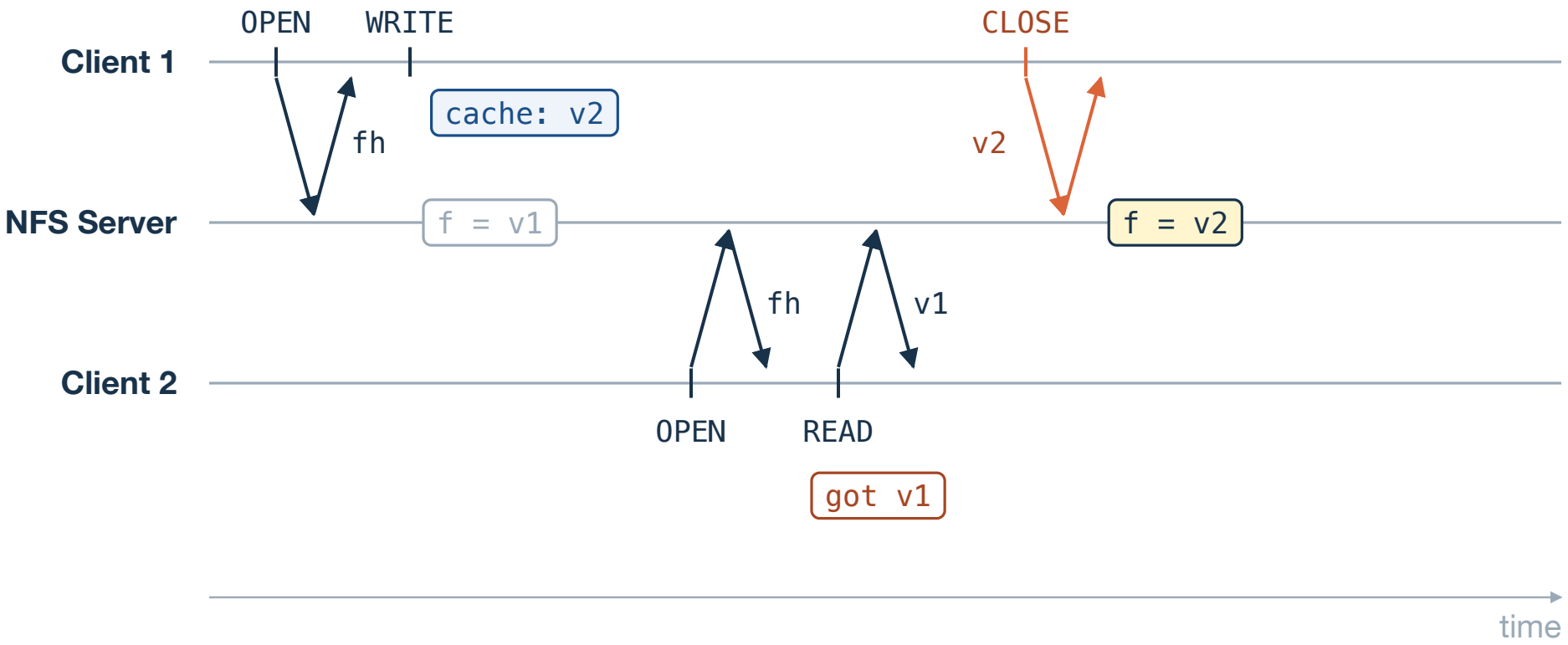
Review: consistency in NFS

Clients cache file data, so a file has several copies: one on the server, one in each client's cache



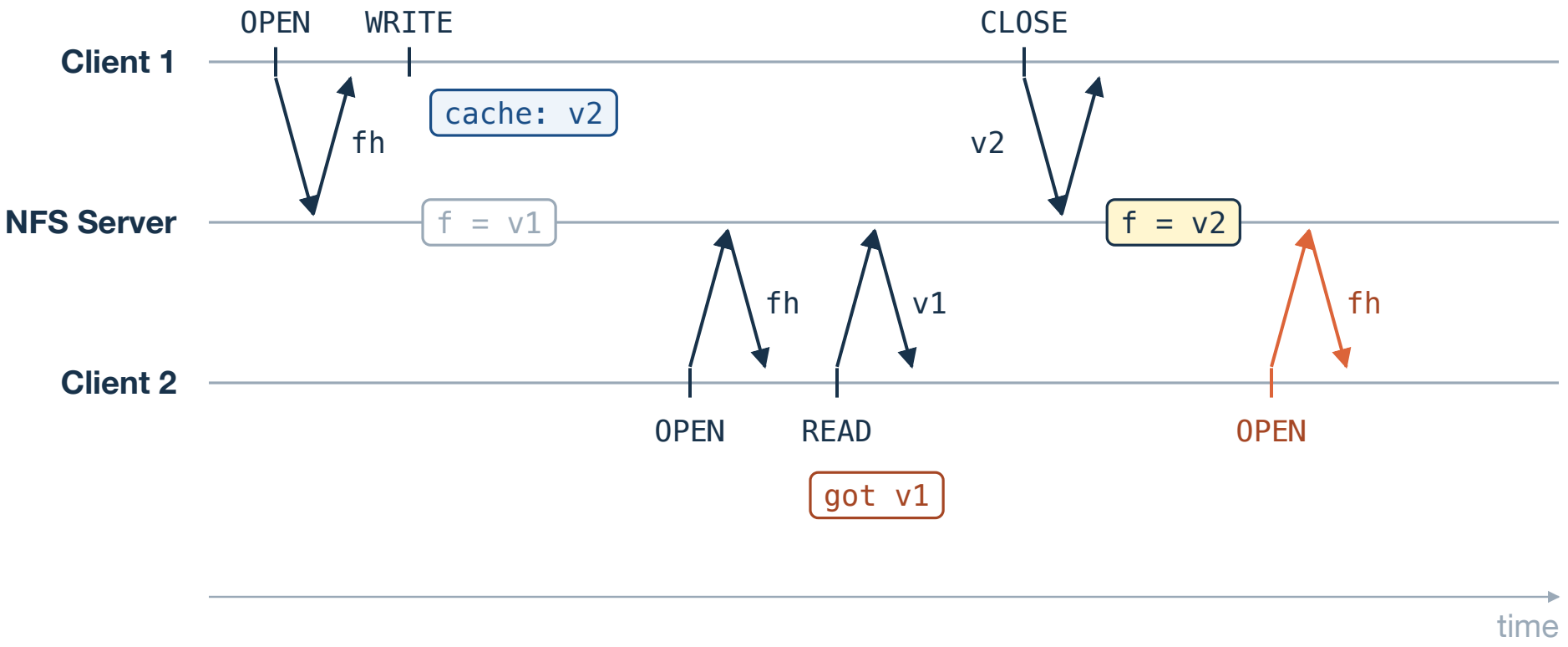
Review: consistency in NFS

Clients cache file data, so a file has several copies: one on the server, one in each client's cache



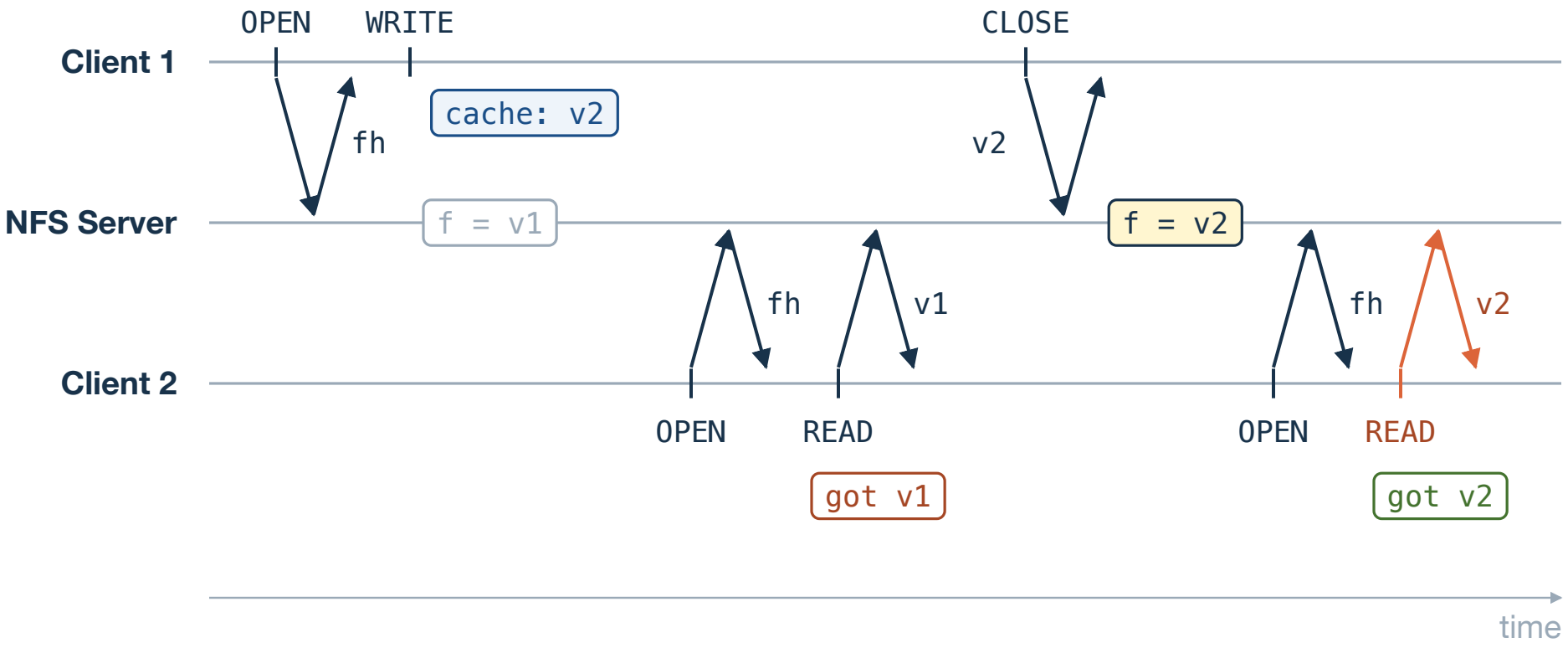
Review: consistency in NFS

Clients cache file data, so a file has several copies: one on the server, one in each client's cache



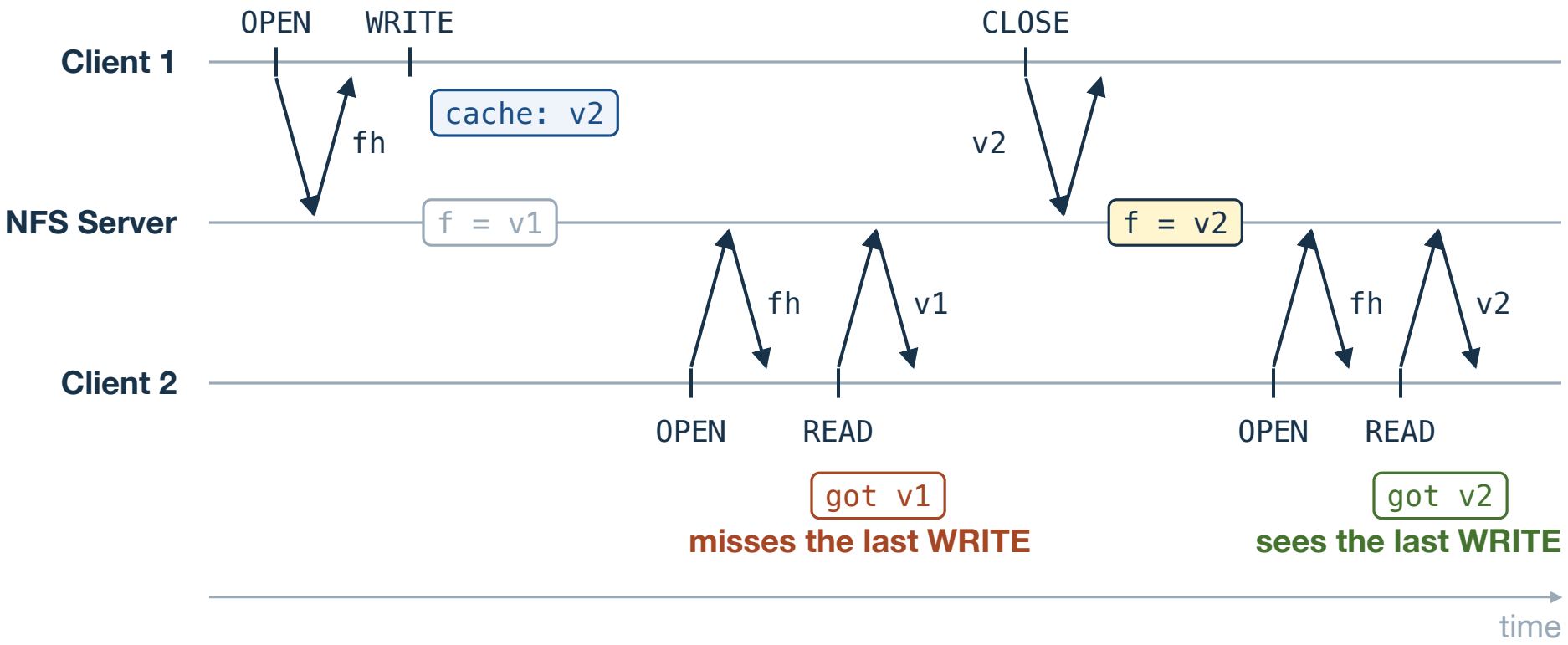
Review: consistency in NFS

Clients cache file data, so a file has several copies: one on the server, one in each client's cache



Review: consistency in NFS

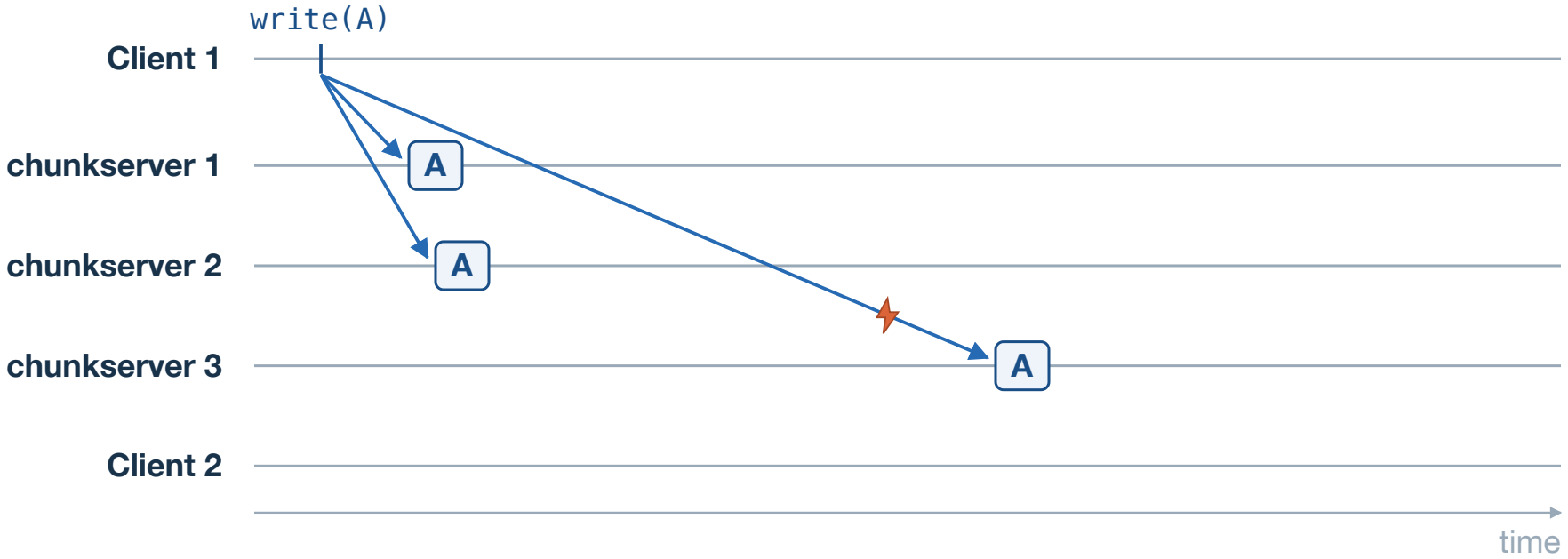
Clients cache file data, so a file has several copies: one on the server, one in each client's cache



NFS: close-to-open consistency; a `READ` may miss a `WRITE` that the writer has not closed yet

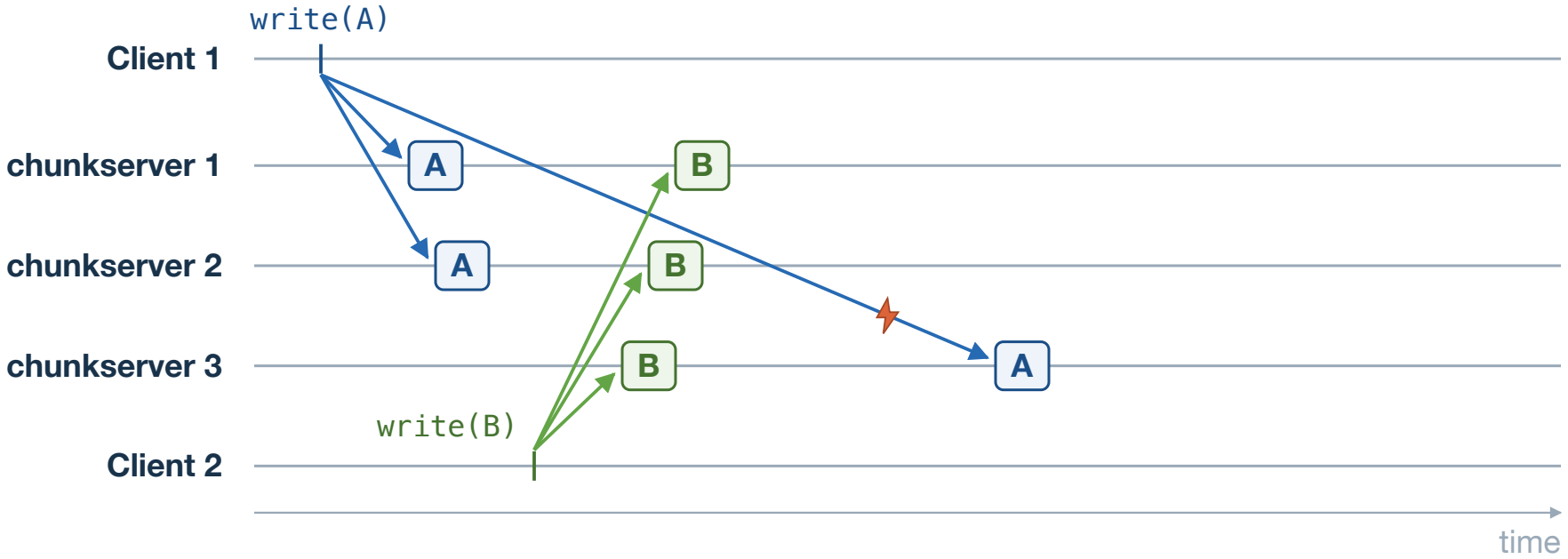
Review: consistency in GFS

Each chunk has three replicas; concurrent writes may reach them in different orders



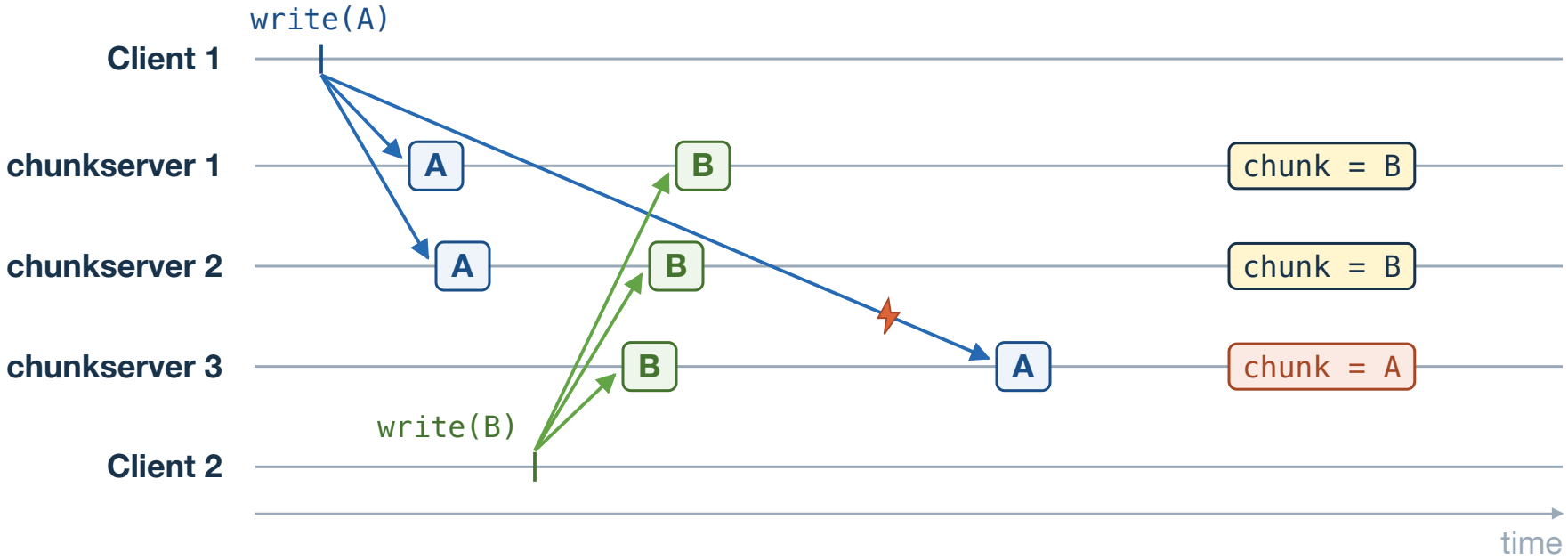
Review: consistency in GFS

Each chunk has three replicas; concurrent writes may reach them in different orders



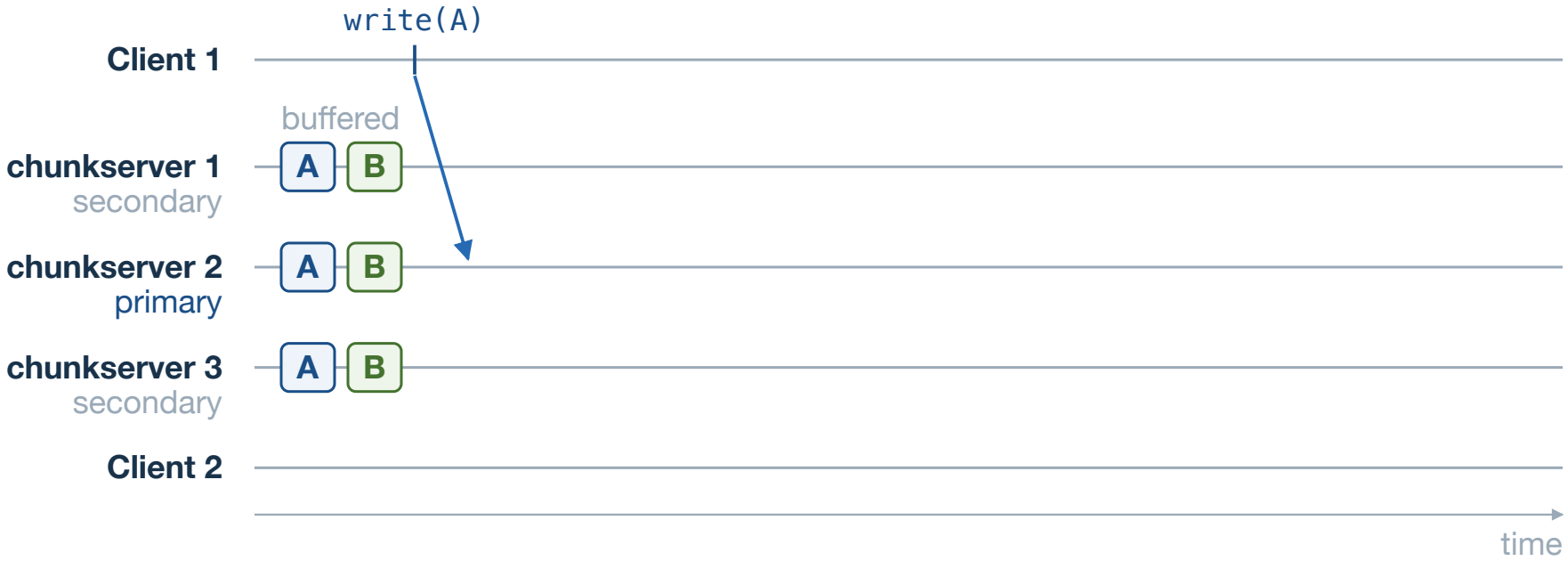
Review: consistency in GFS

Each chunk has three replicas; concurrent writes may reach them in different orders



Review: GFS orders the writes

The primary picks the order; all replicas apply writes in that order



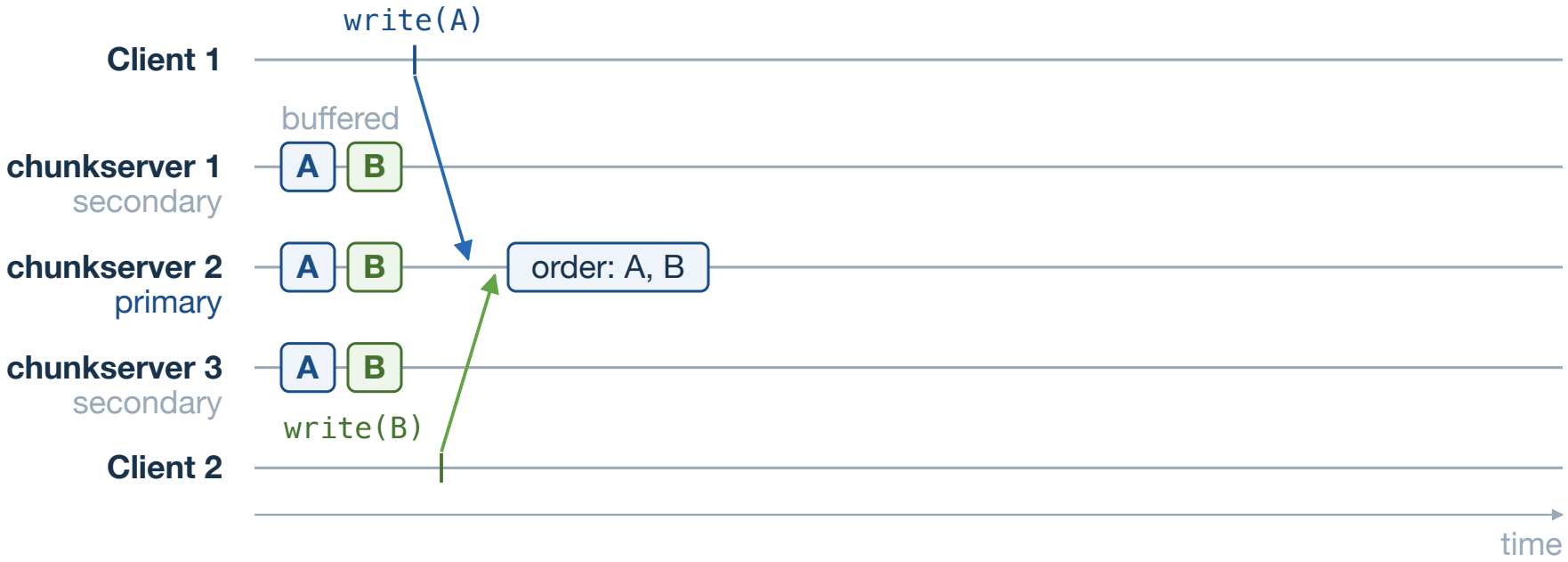
Review: GFS orders the writes

The primary picks the order; all replicas apply writes in that order



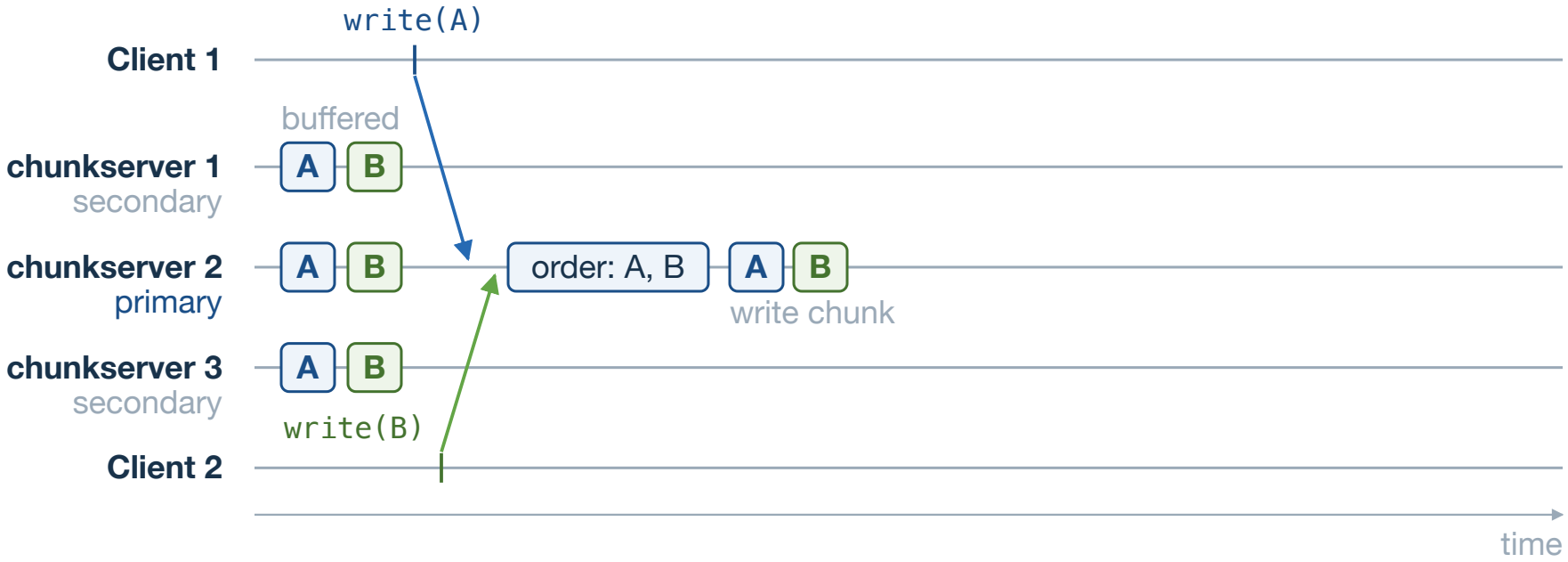
Review: GFS orders the writes

The primary picks the order; all replicas apply writes in that order



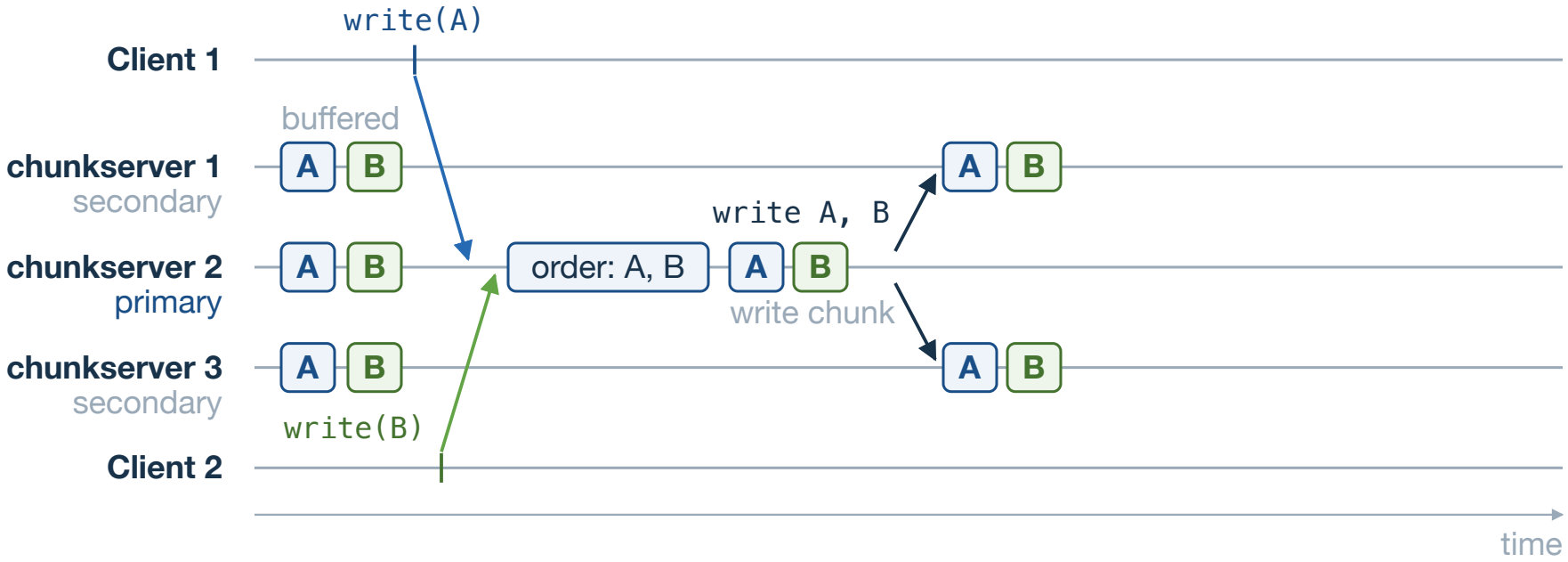
Review: GFS orders the writes

The primary picks the order; all replicas apply writes in that order



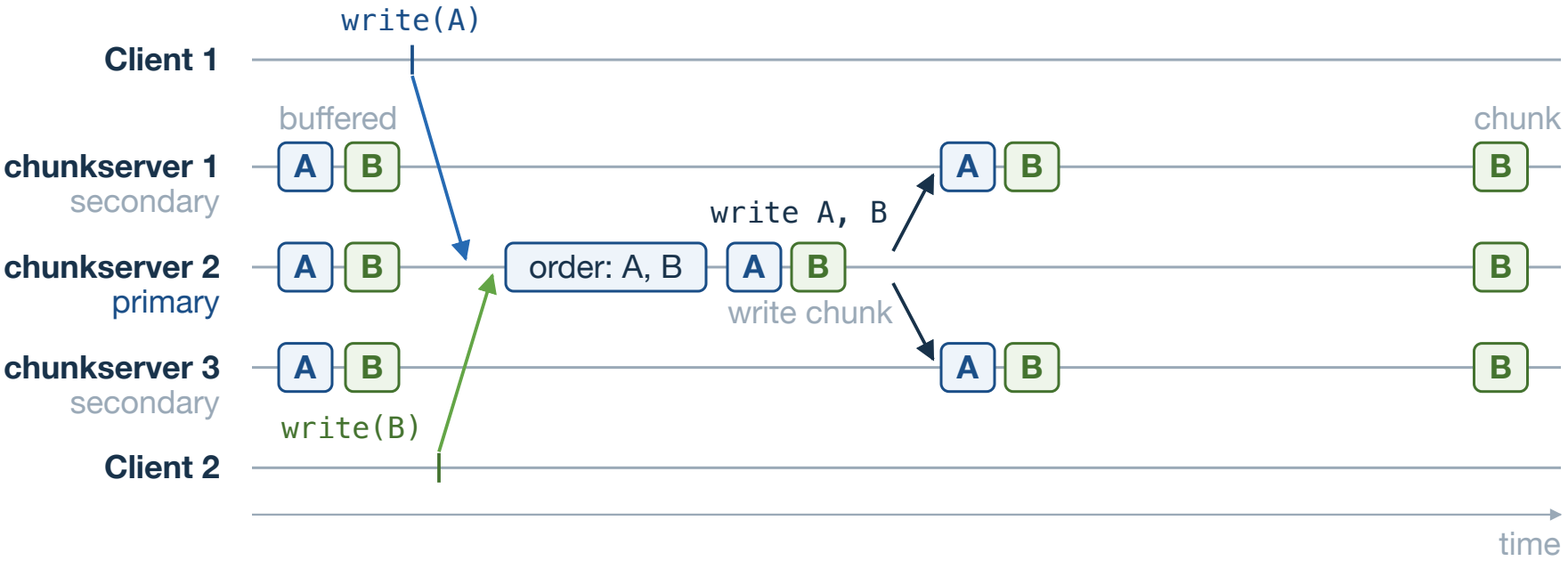
Review: GFS orders the writes

The primary picks the order; all replicas apply writes in that order



Review: GFS orders the writes

The primary picks the order; all replicas apply writes in that order



A write succeeds only if all replicas apply it and reply to the primary

- Then every replica holds the same data
- An error at any replica: the primary reports the write as failed to the client [1]

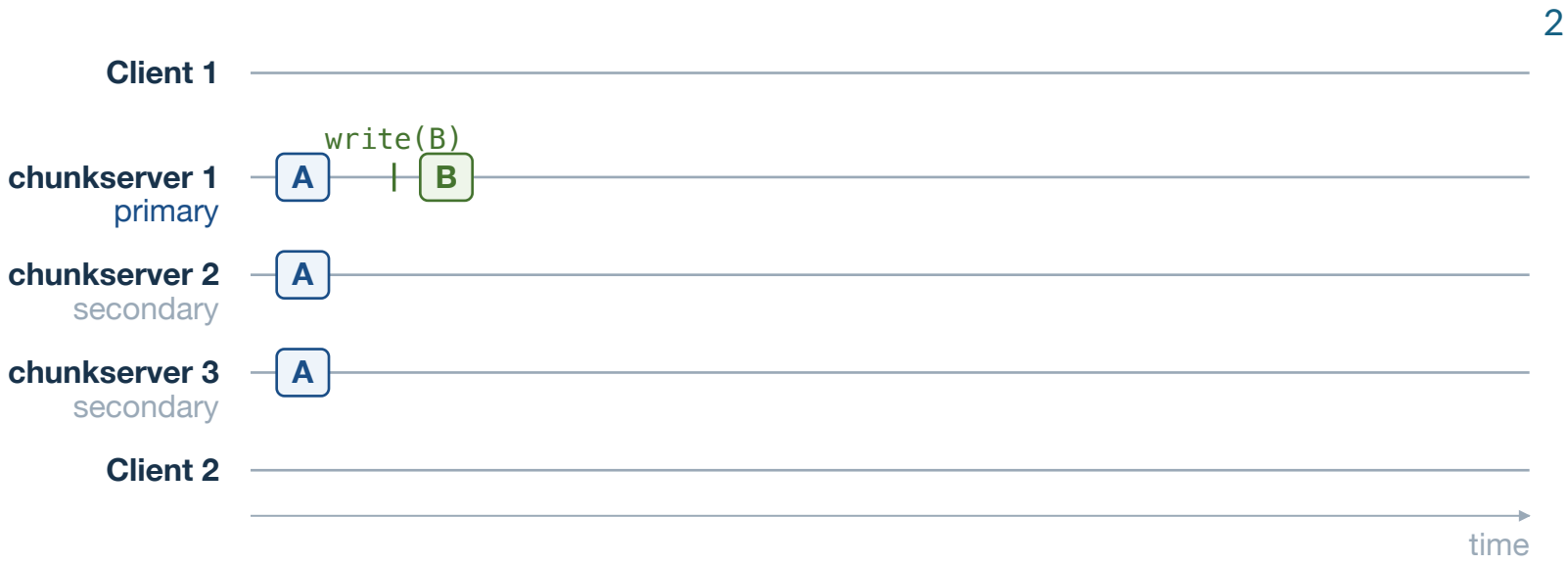
[1] The Google File System, Section 3.1: "The secondaries all reply to the primary indicating that they have completed the operation. The primary replies to the client. Any errors encountered at any of the replicas are reported to the client"

Question: GFS orders the writes. What about the reads?

GFS reads: clients may see different data

Read rule: the client reads from one replica, most likely the closest one [1]

A failed write may still be applied on some replicas

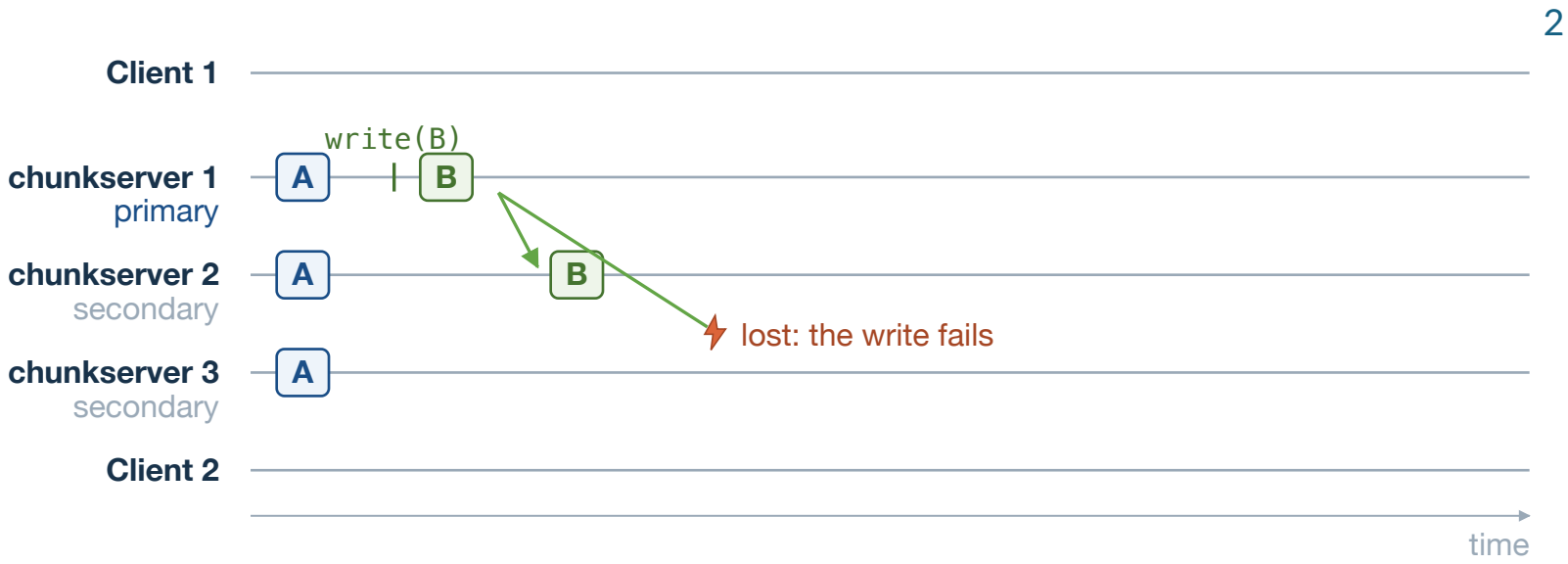


[1] The Google File System, Section 2.4: "The client then sends a request to one of the replicas, most likely the closest one"
[2] The Google File System, Section 2.7.1: "A failed mutation makes the region inconsistent: different clients may see different data at different times"

GFS reads: clients may see different data

Read rule: the client reads from one replica, most likely the closest one [1]

A failed write may still be applied on some replicas

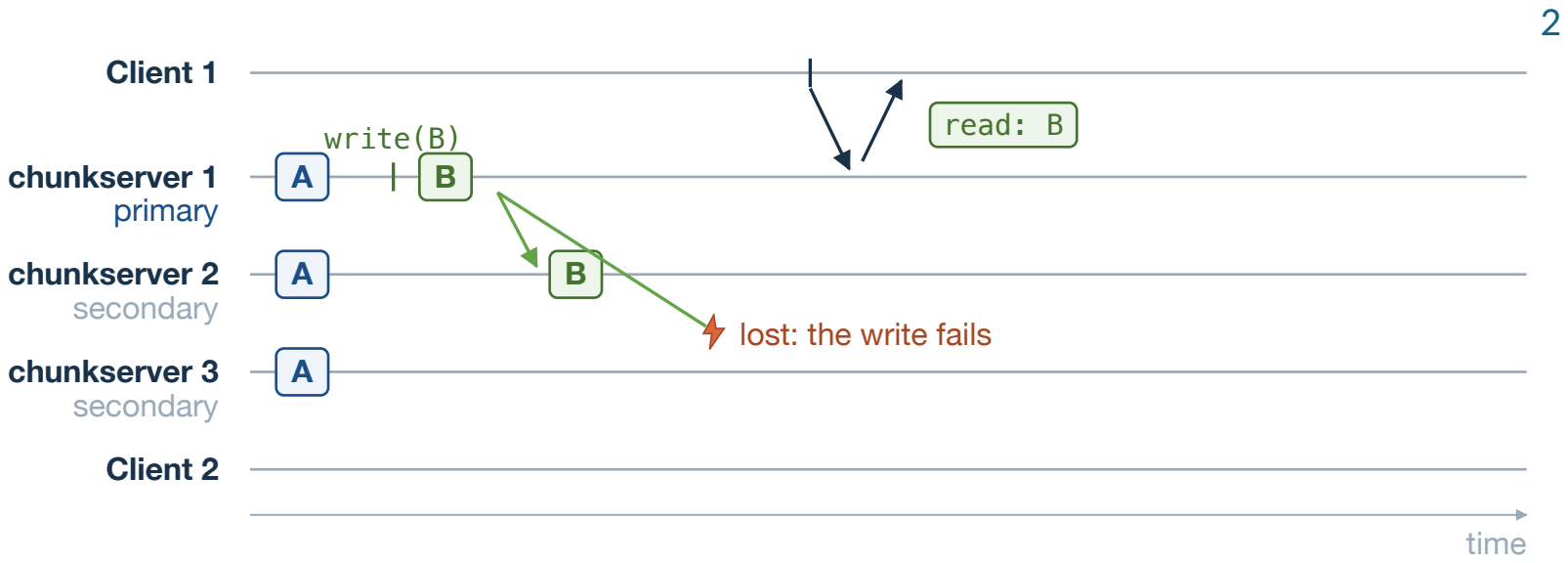


[1] The Google File System, Section 2.4: "The client then sends a request to one of the replicas, most likely the closest one"
[2] The Google File System, Section 2.7.1: "A failed mutation makes the region inconsistent: different clients may see different data at different times"

GFS reads: clients may see different data

Read rule: the client reads from one replica, most likely the closest one [1]

A failed write may still be applied on some replicas

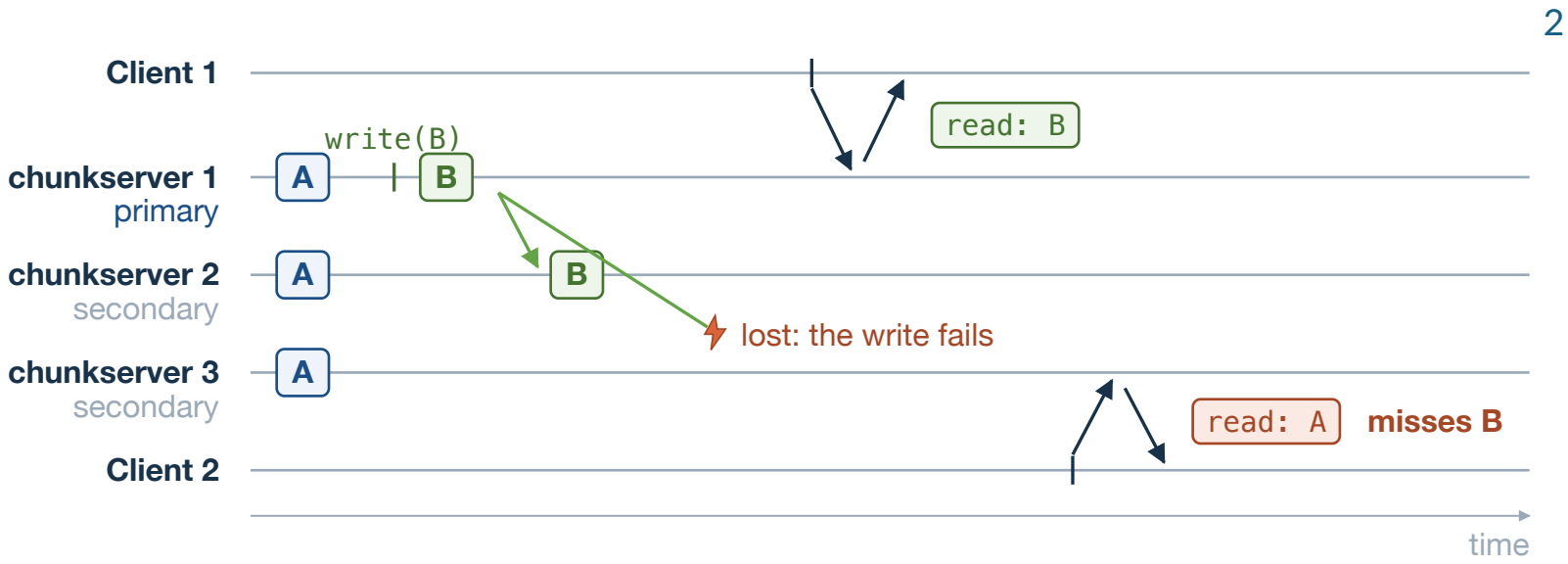


[1] The Google File System, Section 2.4: "The client then sends a request to one of the replicas, most likely the closest one"
[2] The Google File System, Section 2.7.1: "A failed mutation makes the region inconsistent: different clients may see different data at different times"

GFS reads: clients may see different data

Read rule: the client reads from one replica, most likely the closest one [1]

A failed write may still be applied on some replicas



[2]

Client 2 reads after Client 1, but does not see the write

- This typically means inconsistency

[1] The Google File System, Section 2.4: "The client then sends a request to one of the replicas, most likely the closest one"
[2] The Google File System, Section 2.7.1: "A failed mutation makes the region inconsistent: different clients may see different data at different times"

The topics of today's lecture

What is a consistency model & why we need it

The eventual consistency model

- How it is defined and used in daily life

Implementing eventual consistency w/ various timestamps

- Non-trivial once we consider user experience

More models: linearizability and its implementations

Motivation case: Implementing a ChatAPP

Have you experienced this? Why does this happen? Is there a fundamental trade-off?



We call such a case `inconsistent`, i.e., the order is different

Order: the different states of a system observed by different users

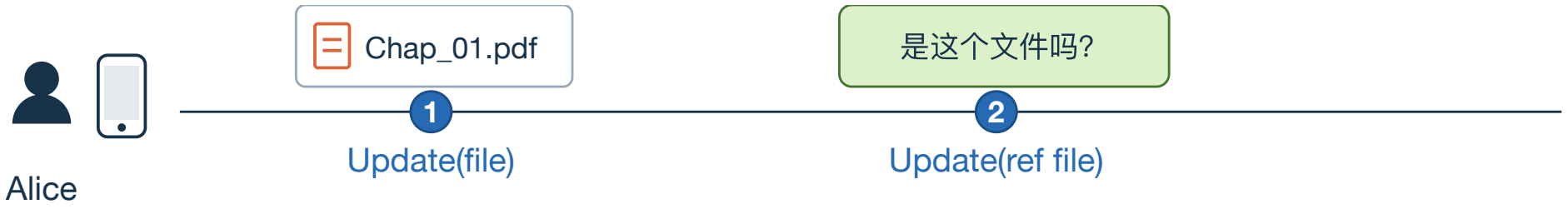
- State: the chat history in our case (can be others)
- Operations that change the state: update
- Operations that observe the state: read



We call such a case `inconsistent`, i.e., the order is different

Order: the different states of a system observed by different users

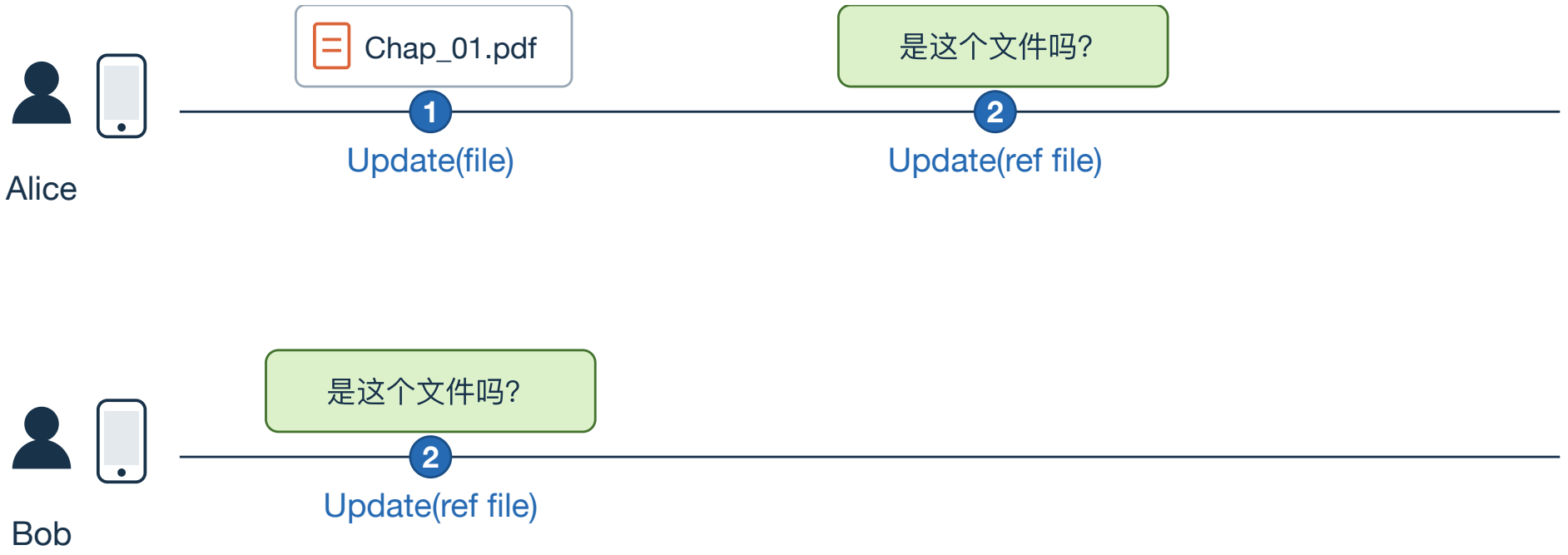
- State: the chat history in our case (can be others)
- Operations that change the state: update
- Operations that observe the state: read



We call such a case `inconsistent`, i.e., the order is different

Order: the different states of a system observed by different users

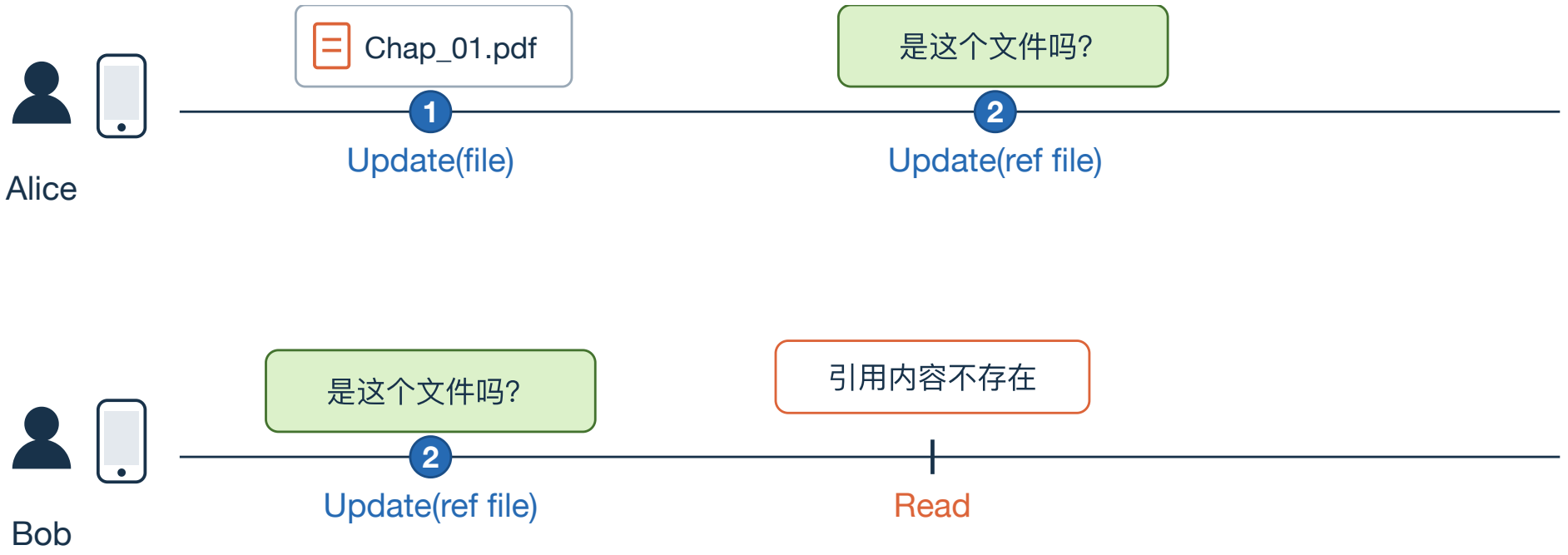
- State: the chat history in our case (can be others)
- Operations that change the state: update
- Operations that observe the state: read



We call such a case `inconsistent`, i.e., the order is different

Order: the different states of a system observed by different users

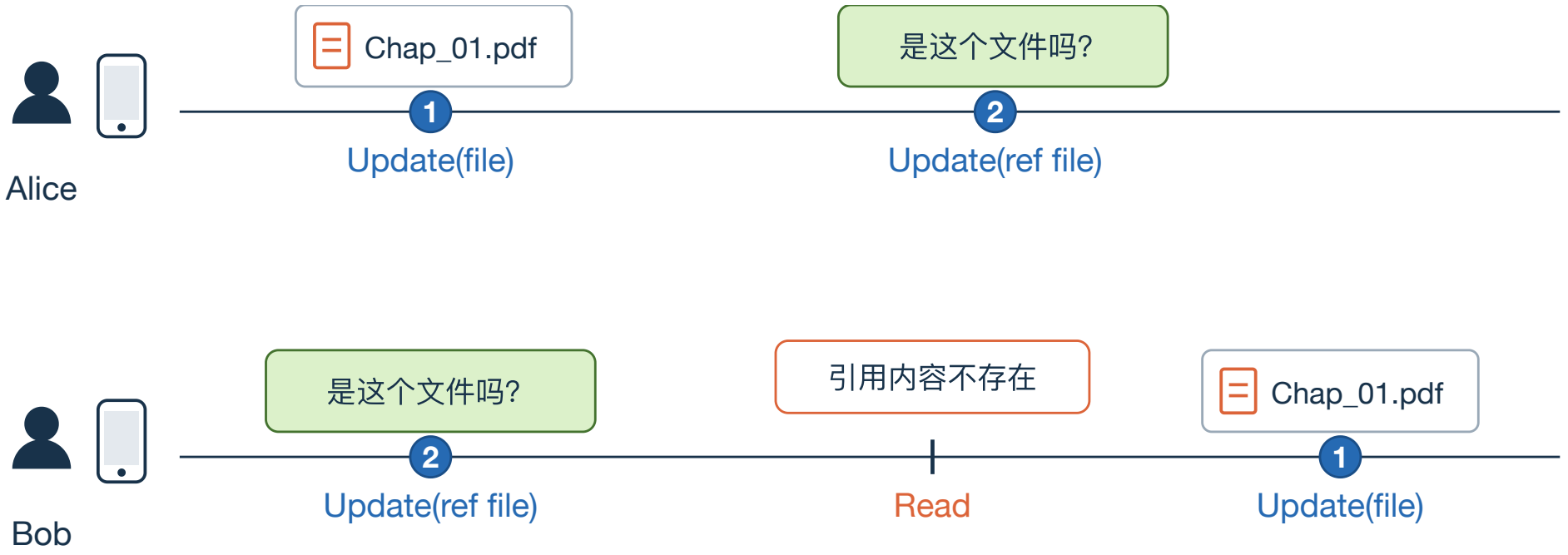
- State: the chat history in our case (can be others)
- Operations that change the state: update
- Operations that observe the state: read



We call such a case `inconsistent`, i.e., the order is different

Order: the different states of a system observed by different users

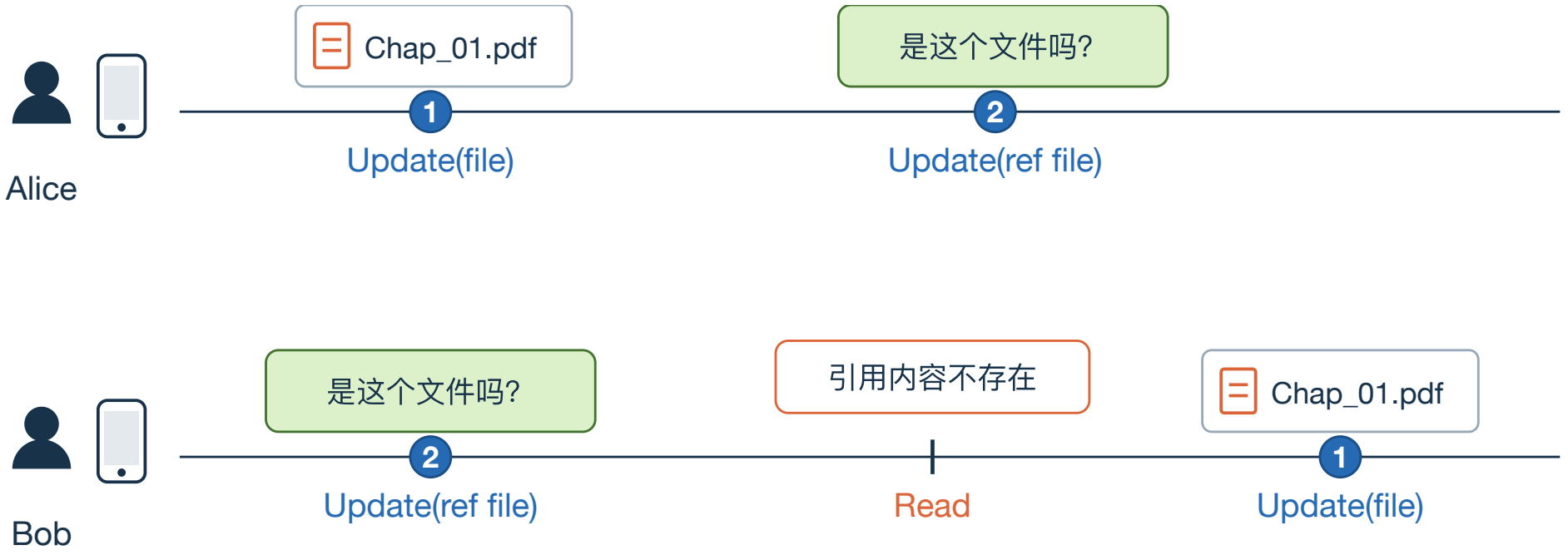
- State: the chat history in our case (can be others)
- Operations that change the state: update
- Operations that observe the state: read



We call such a case `inconsistent`, i.e., the order is different

Order: the different states of a system observed by different users

- State: the chat history in our case (can be others)
- Operations that change the state: update
- Operations that observe the state: read



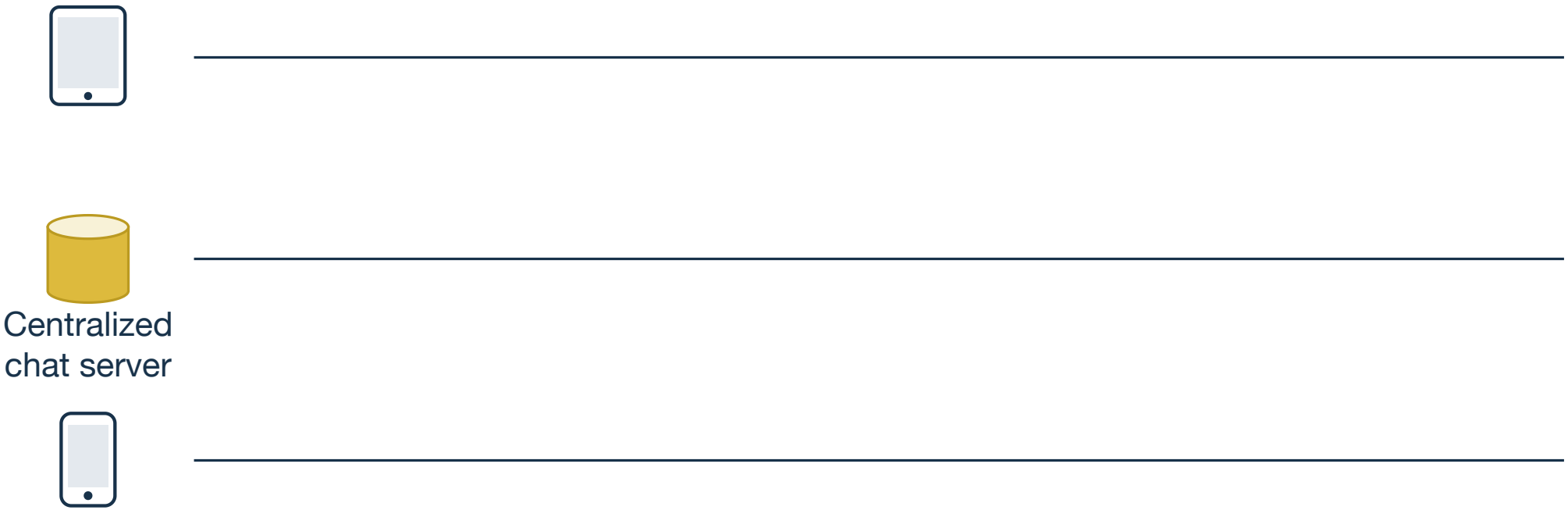
The two see different orders

To see why reordering can happen, we need to see how the underlying system is implemented

Implementing ChatAPP: a (naïve) centralized approach

System model: a centralized server stores all the state

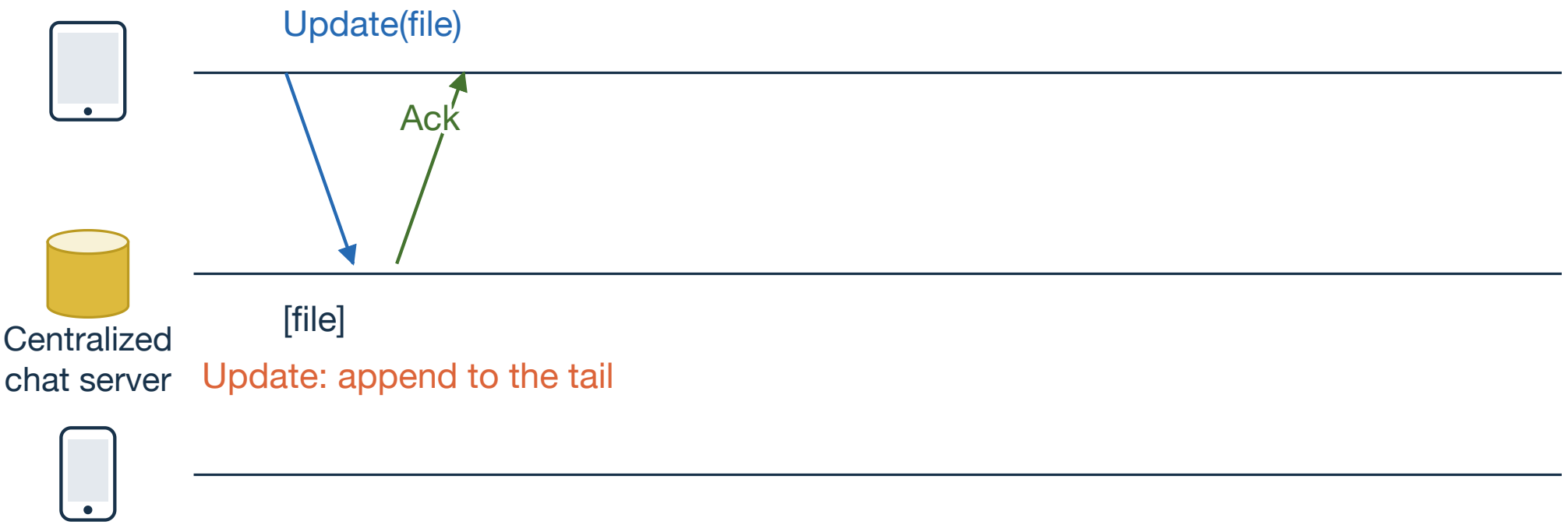
- Detailed data structure of the state: a `list` of items, each item a `string` or a file
- Operations: `update(msg/file)`, `read(history)`, both via RPC



Implementing ChatAPP: a (naïve) centralized approach

System model: a centralized server stores all the state

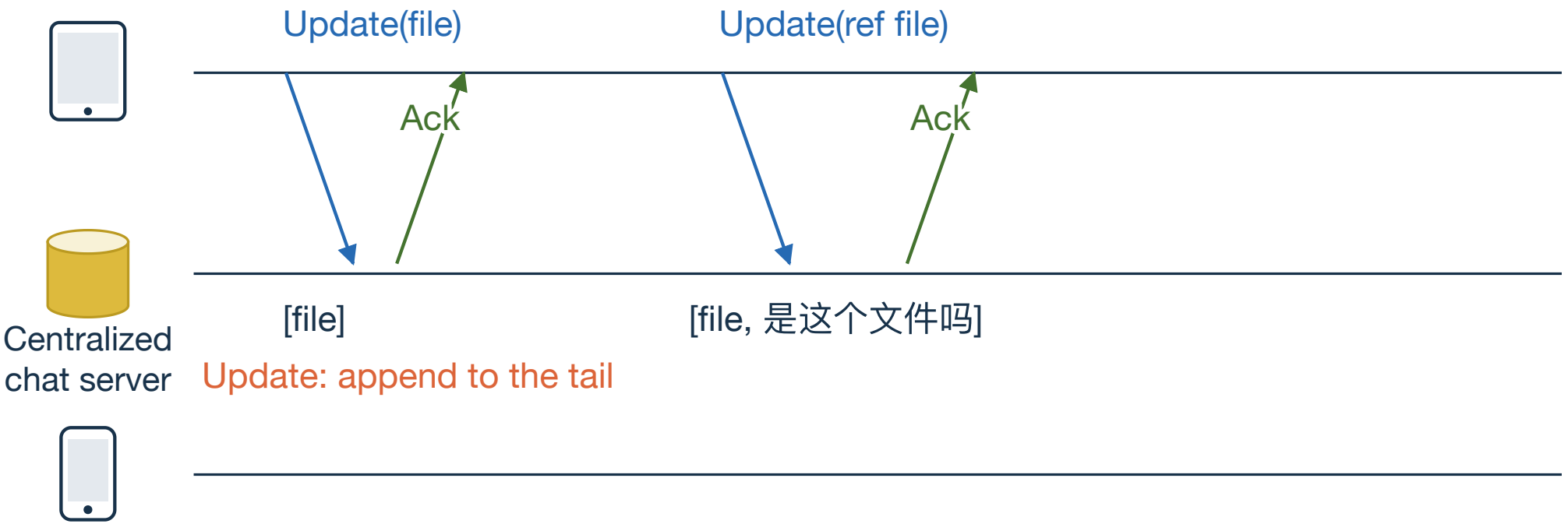
- Detailed data structure of the state: a `list` of items, each item a `string` or a file
- Operations: `update(msg/file)`, `read(history)`, both via RPC



Implementing ChatAPP: a (naïve) centralized approach

System model: a centralized server stores all the state

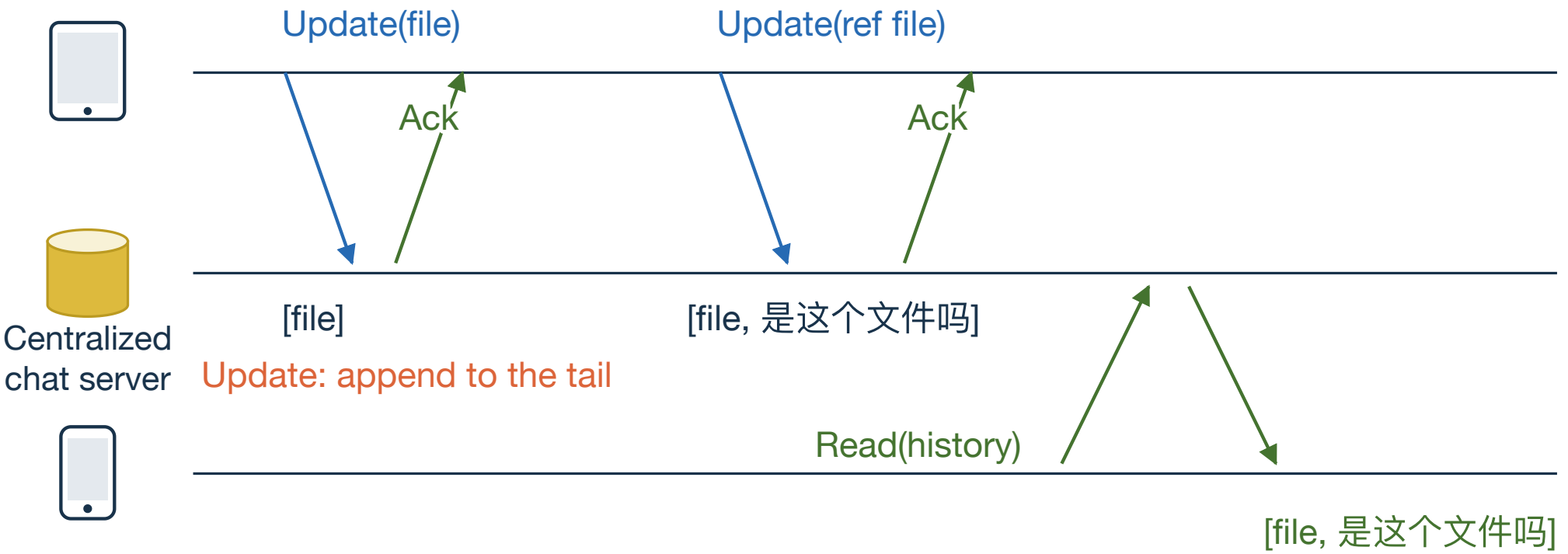
- Detailed data structure of the state: a `list` of items, each item a `string` or a file
- Operations: `update(msg/file)`, `read(history)`, both via RPC



Implementing ChatAPP: a (naïve) centralized approach

System model: a centralized server stores all the state

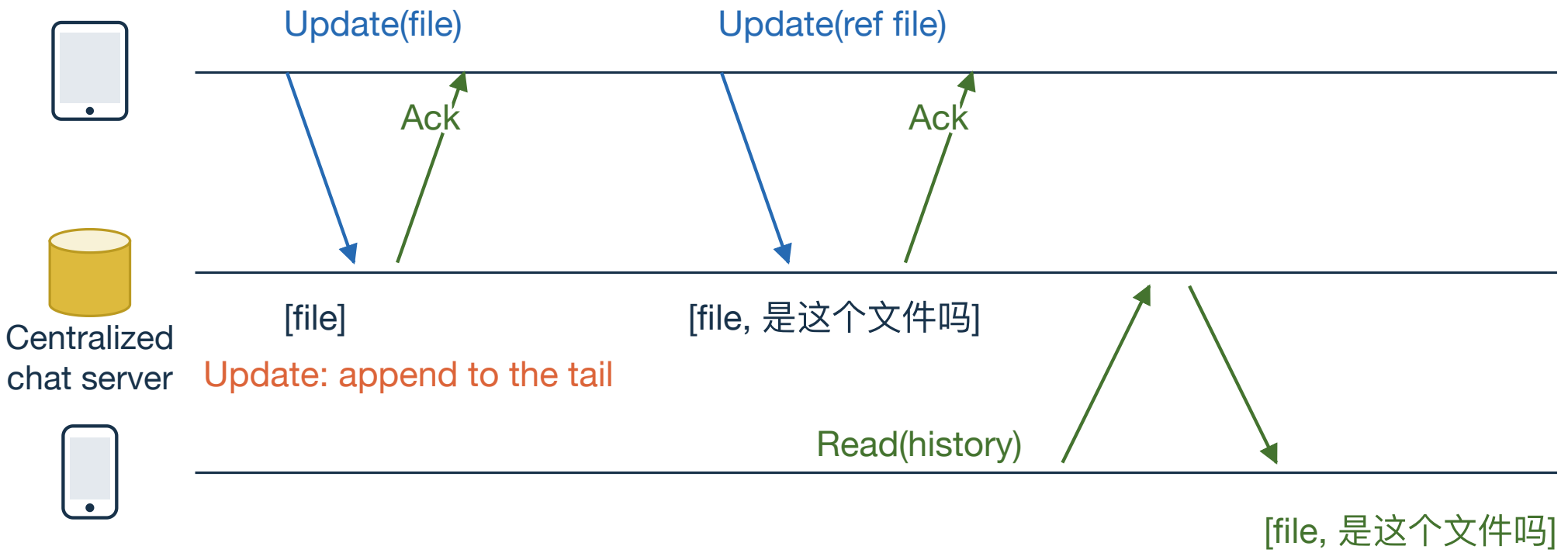
- Detailed data structure of the state: a `list` of items, each item a `string` or a file
- Operations: `update(msg/file)`, `read(history)`, both via RPC



Question: can the centralized approach reorder operations?

No — as long as two conditions hold

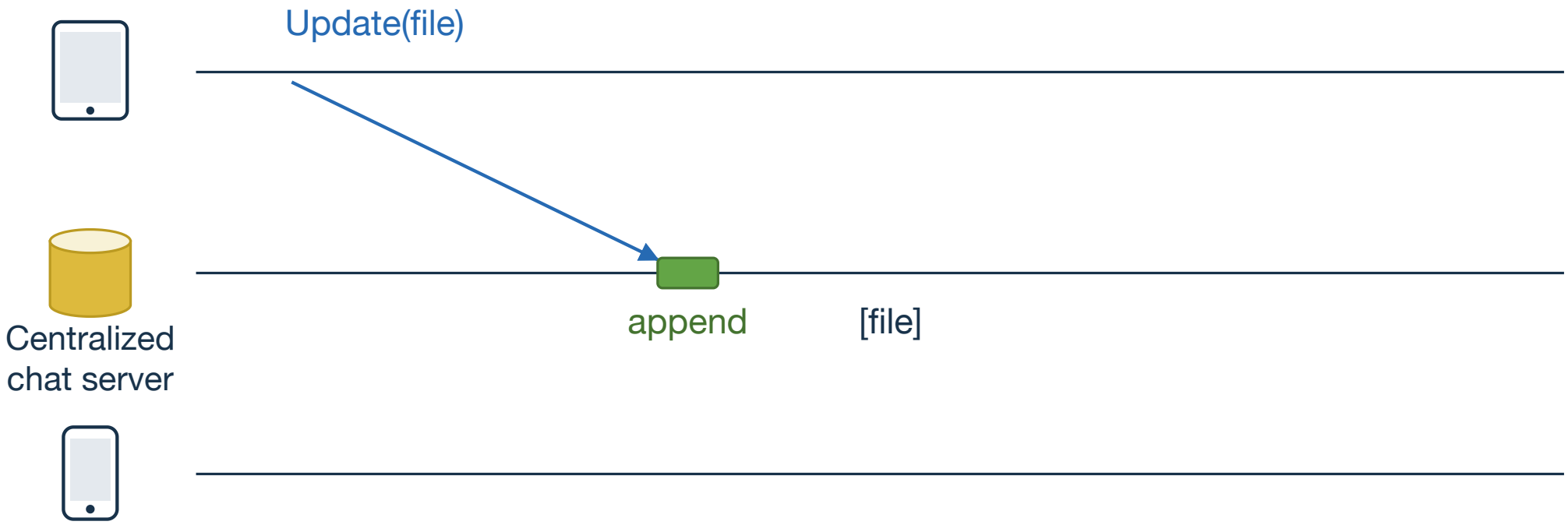
- There is a single server: it appends every update in the order it arrives
- Each user waits for the ack before issuing its next RPC



Cons #1 (Performance): long update time

In practice, we rarely see a ChatAPP adopt this implementation

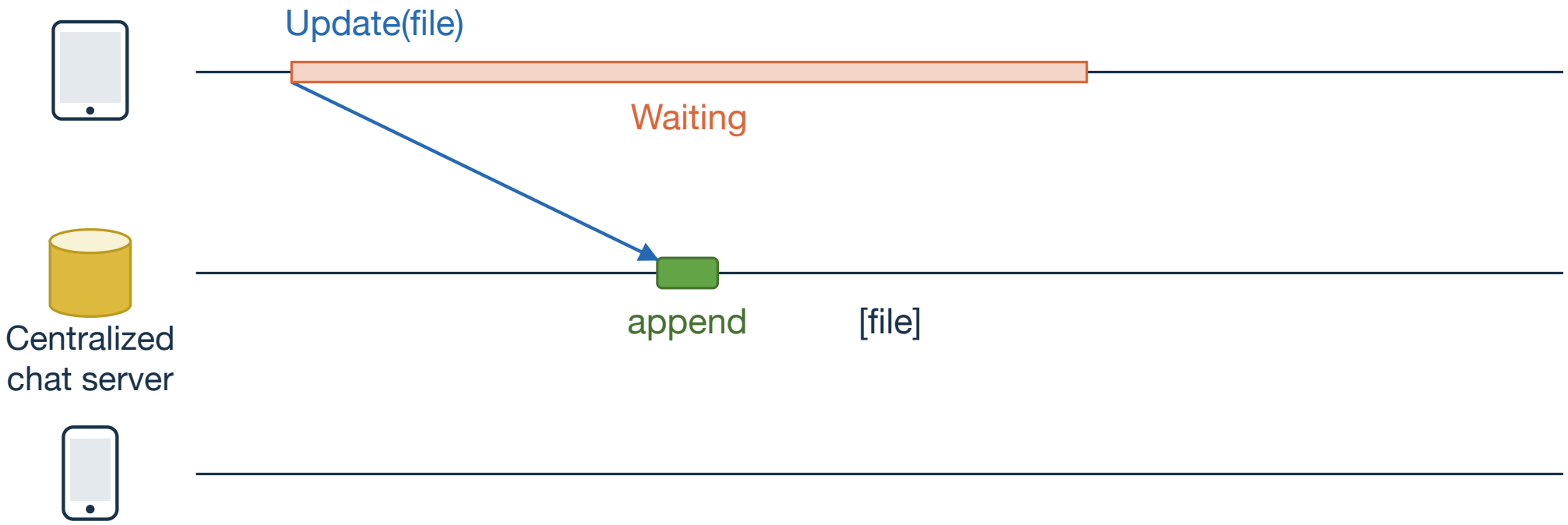
- Every request must wait for the server's ack, which can take a long time



Cons #1 (Performance): long update time

In practice, we rarely see a ChatAPP adopt this implementation

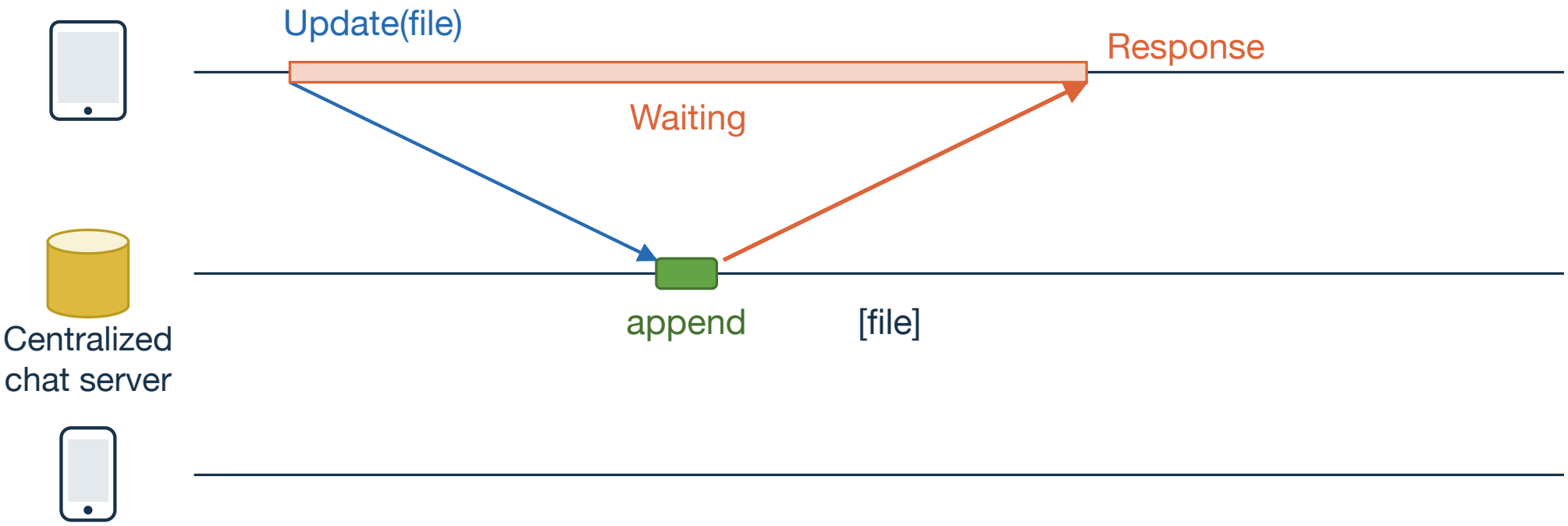
- Every request must wait for the server's ack, which can take a long time



Cons #1 (Performance): long update time

In practice, we rarely see a ChatAPP adopt this implementation

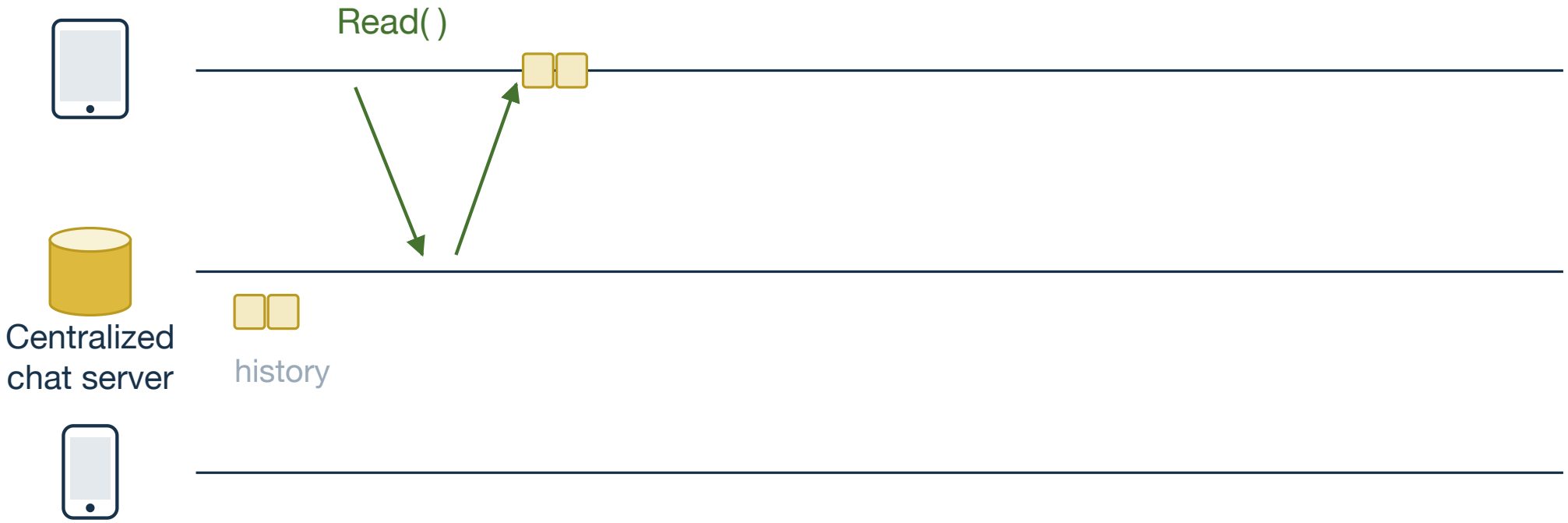
- Every request must wait for the server's ack, which can take a long time



Cons #2 (Performance): slow reads, and slower as the history grows

In practice, we rarely see a ChatAPP adopt this implementation

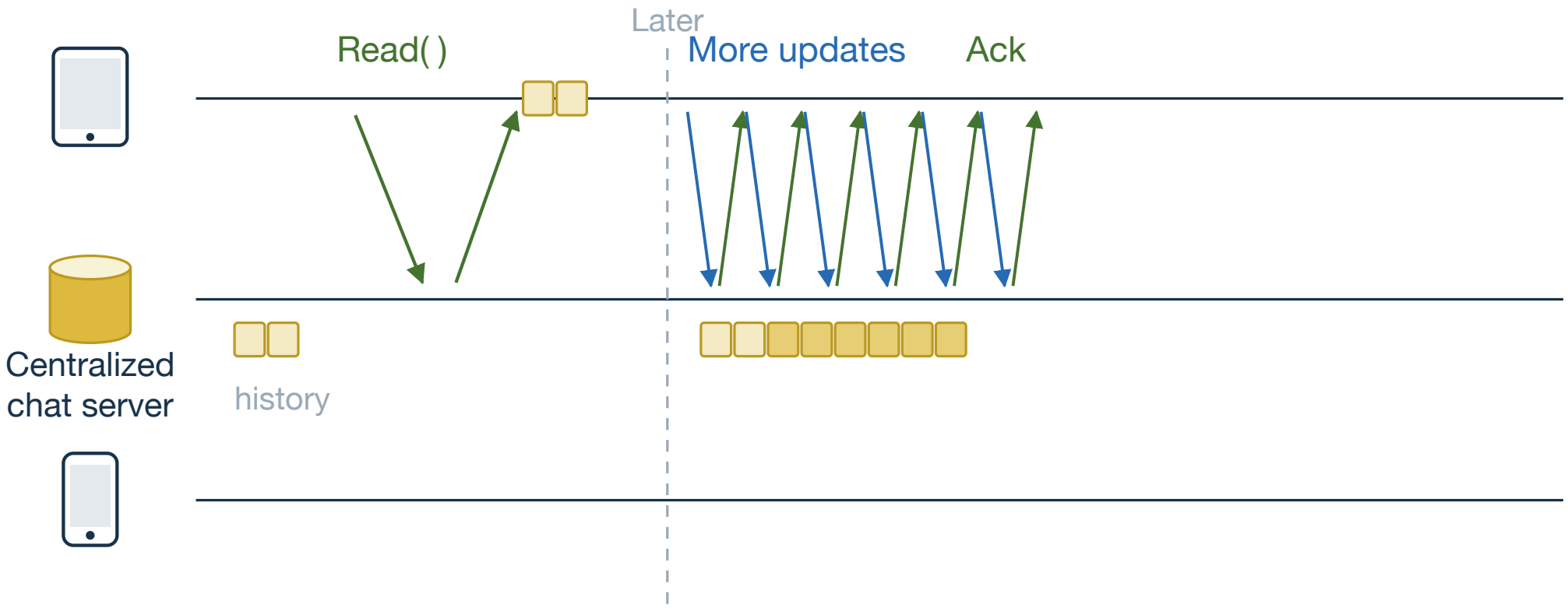
- A read ships the whole history back
- The longer the chat has been going, the longer every read takes



Cons #2 (Performance): slow reads, and slower as the history grows

In practice, we rarely see a ChatAPP adopt this implementation

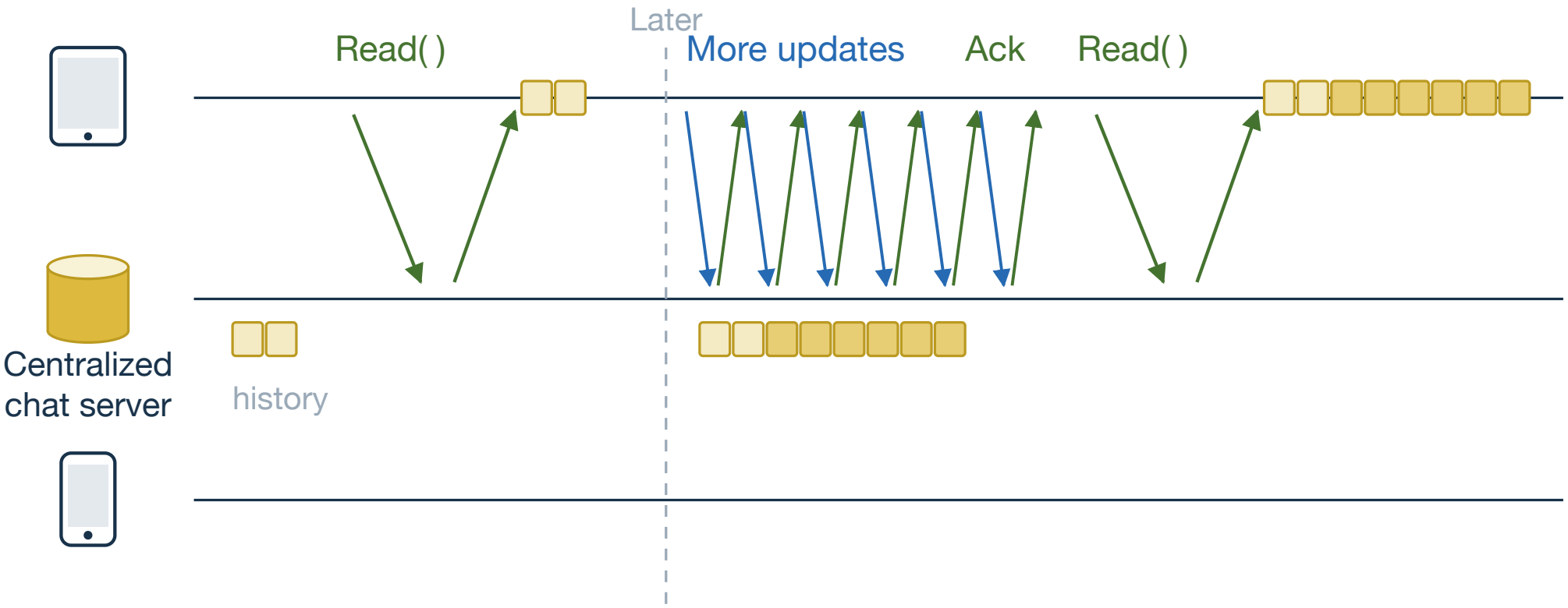
- A read ships the whole history back
- The longer the chat has been going, the longer every read takes



Cons #2 (Performance): slow reads, and slower as the history grows

In practice, we rarely see a ChatAPP adopt this implementation

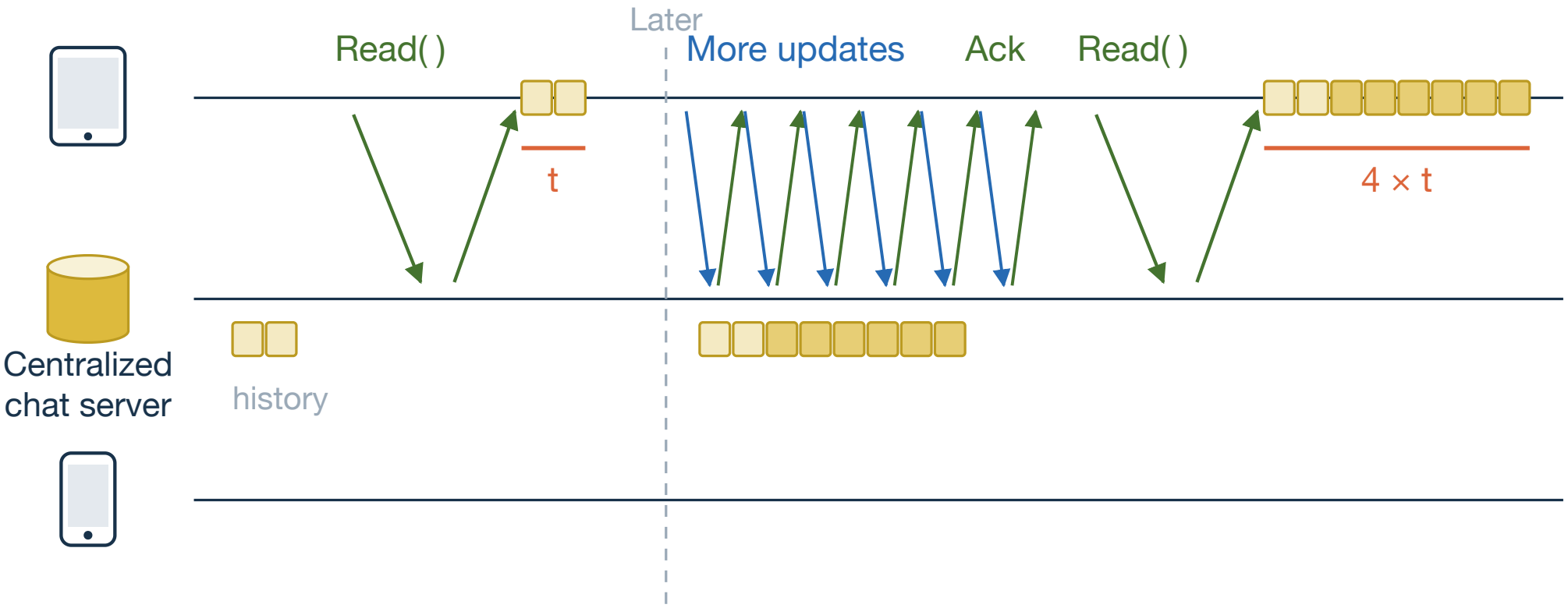
- A read ships the whole history back
- The longer the chat has been going, the longer every read takes



Cons #2 (Performance): slow reads, and slower as the history grows

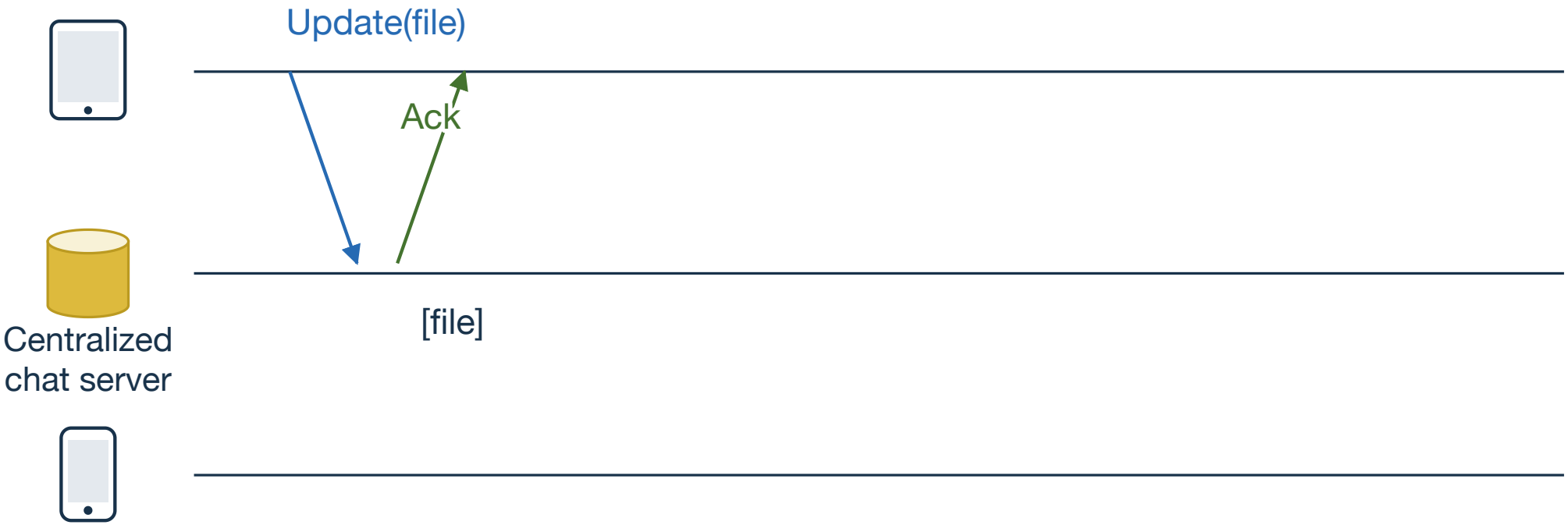
In practice, we rarely see a ChatAPP adopt this implementation

- A read ships the whole history back
- The longer the chat has been going, the longer every read takes



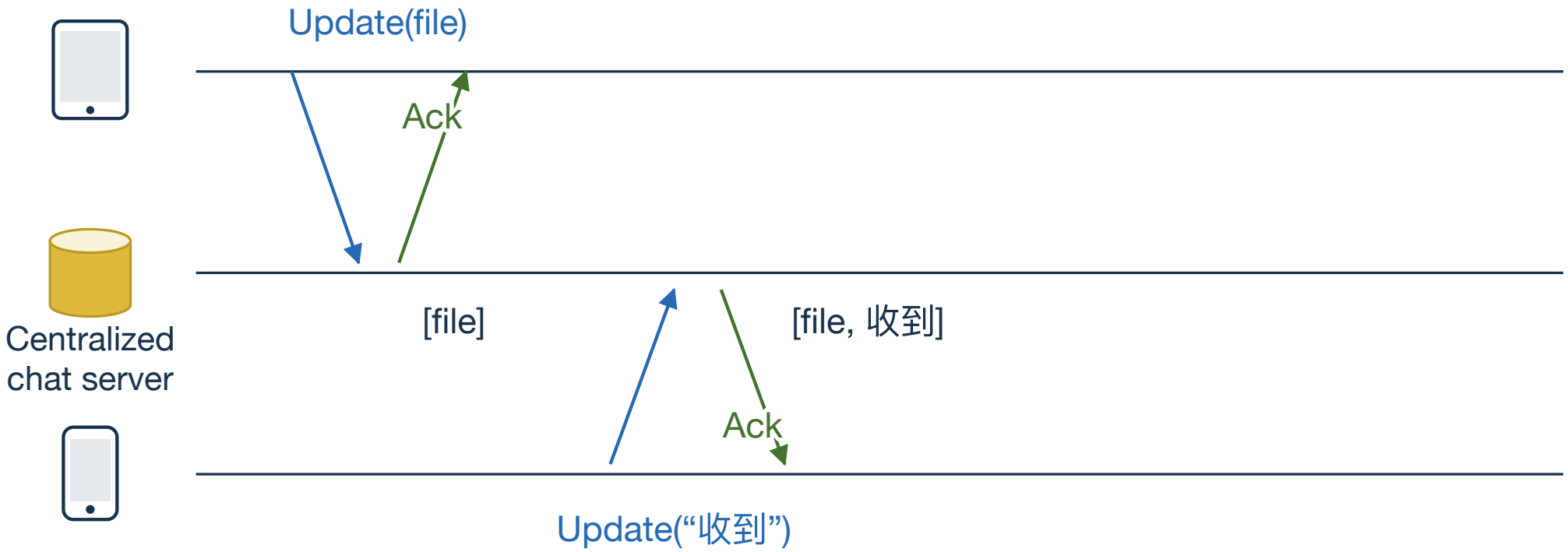
Cons #3 (Reliability): cannot work offline

No network, no chat — not even the messages you sent yourself



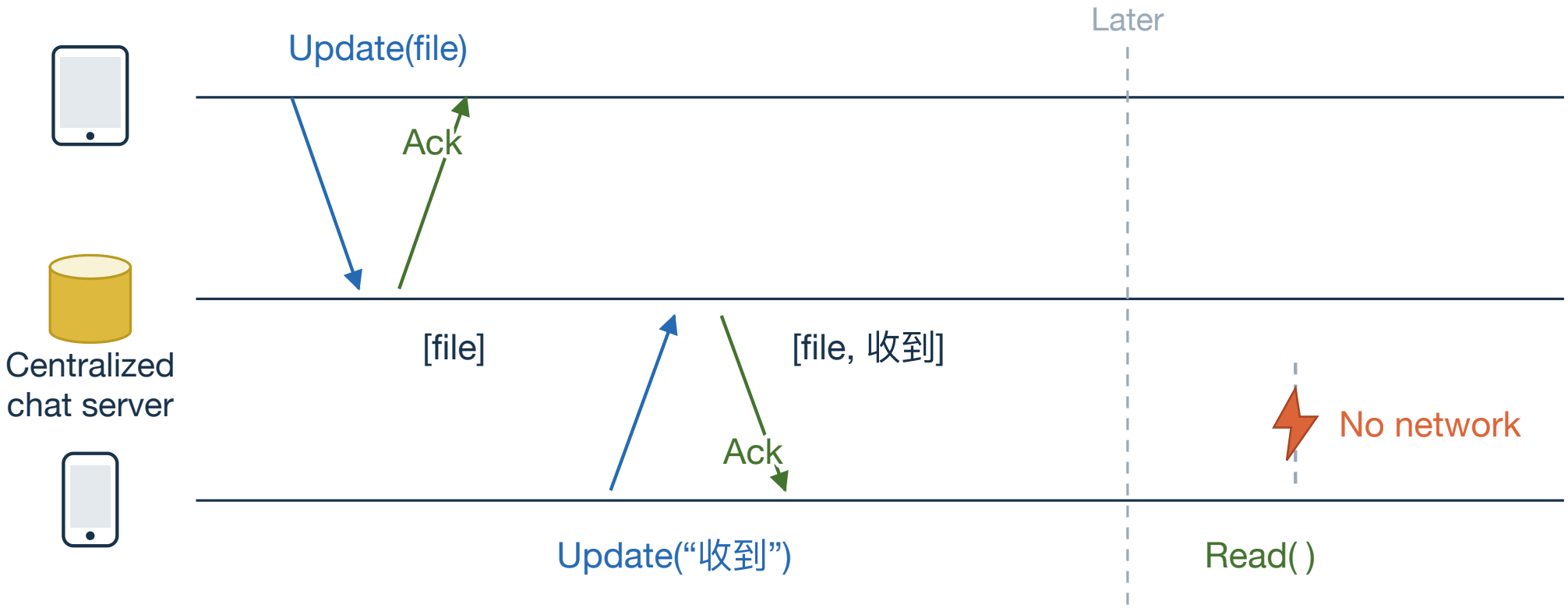
Cons #3 (Reliability): cannot work offline

No network, no chat — not even the messages you sent yourself



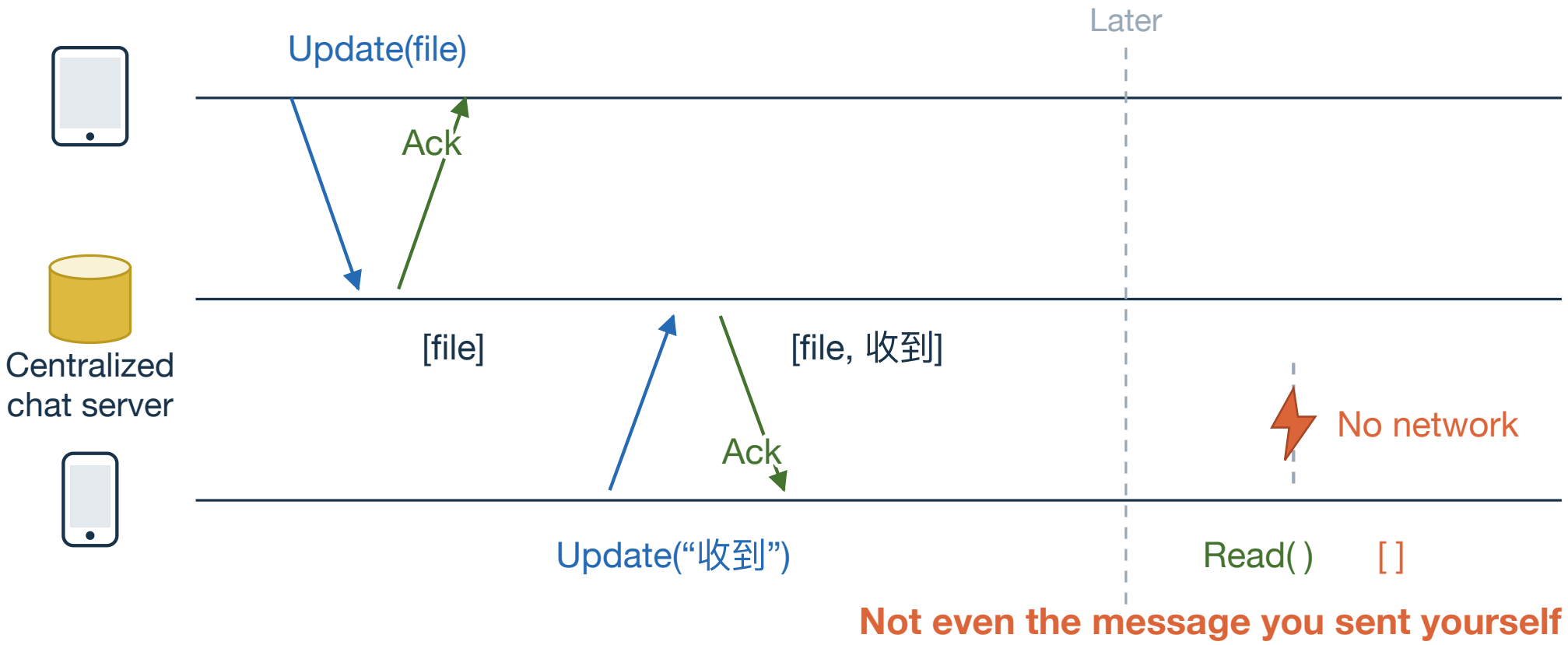
Cons #3 (Reliability): cannot work offline

No network, no chat — not even the messages you sent yourself



Cons #3 (Reliability): cannot work offline

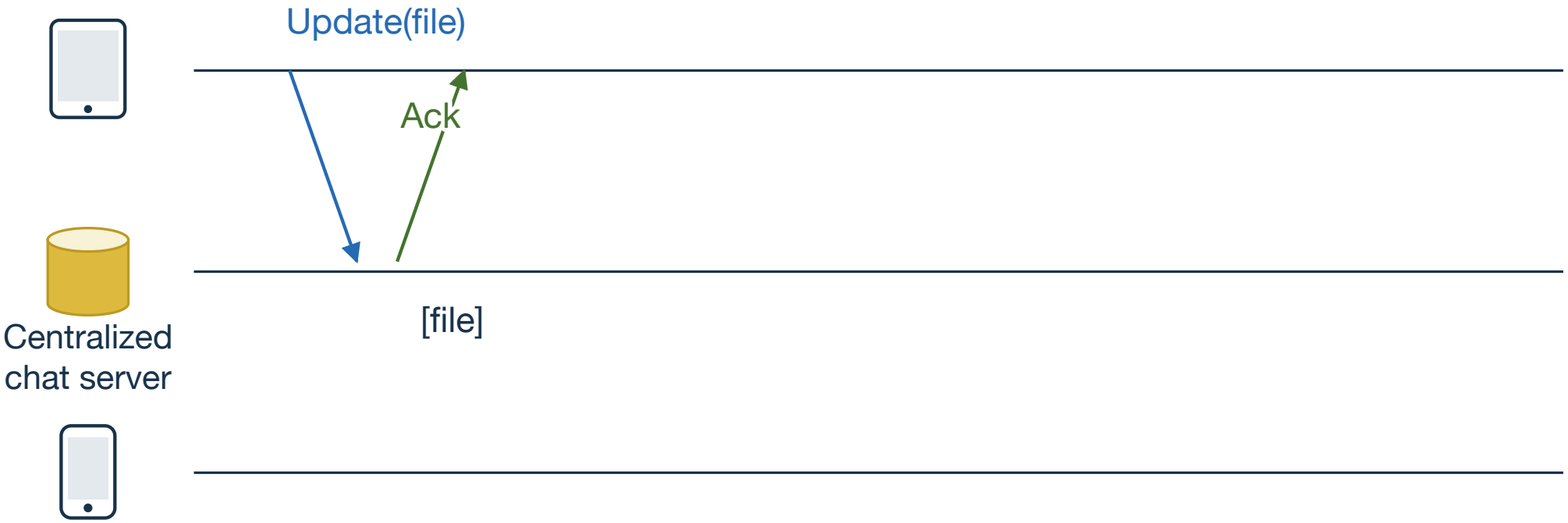
No network, no chat — not even the messages you sent yourself



Cons #4 (Reliability): cannot deal with server failures

In practice, we rarely see a ChatAPP adopt this implementation

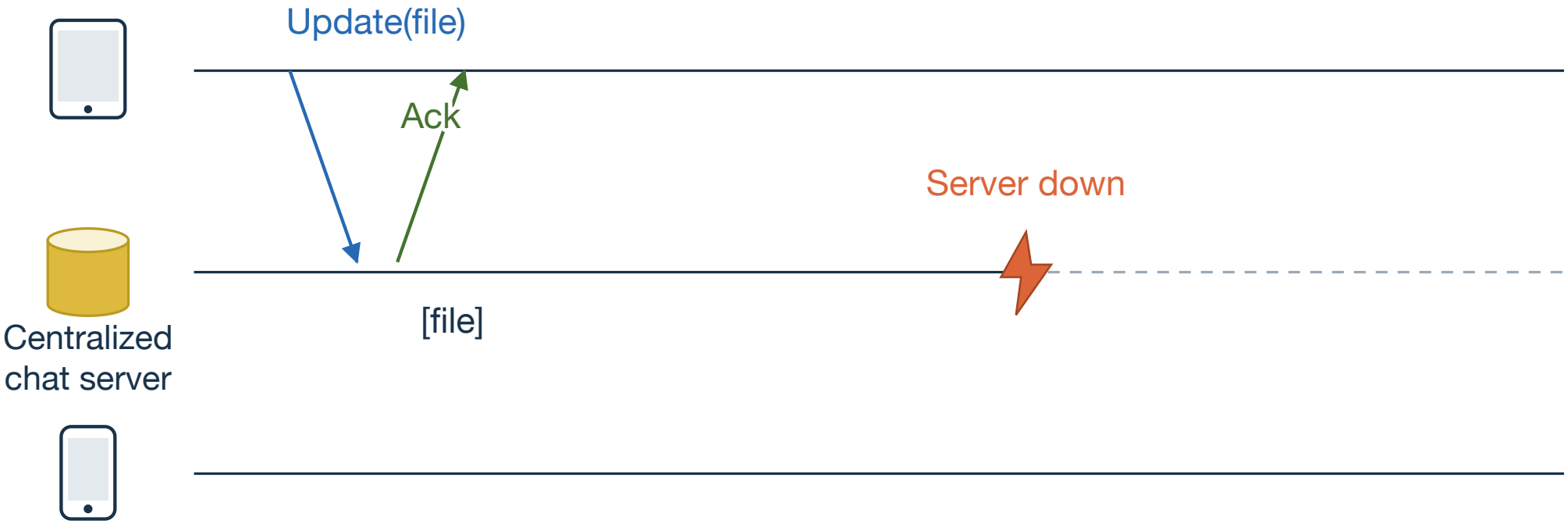
- The server is down, everybody stops
- Though we can make the server highly available (see the Paxos and Raft lecture)



Cons #4 (Reliability): cannot deal with server failures

In practice, we rarely see a ChatAPP adopt this implementation

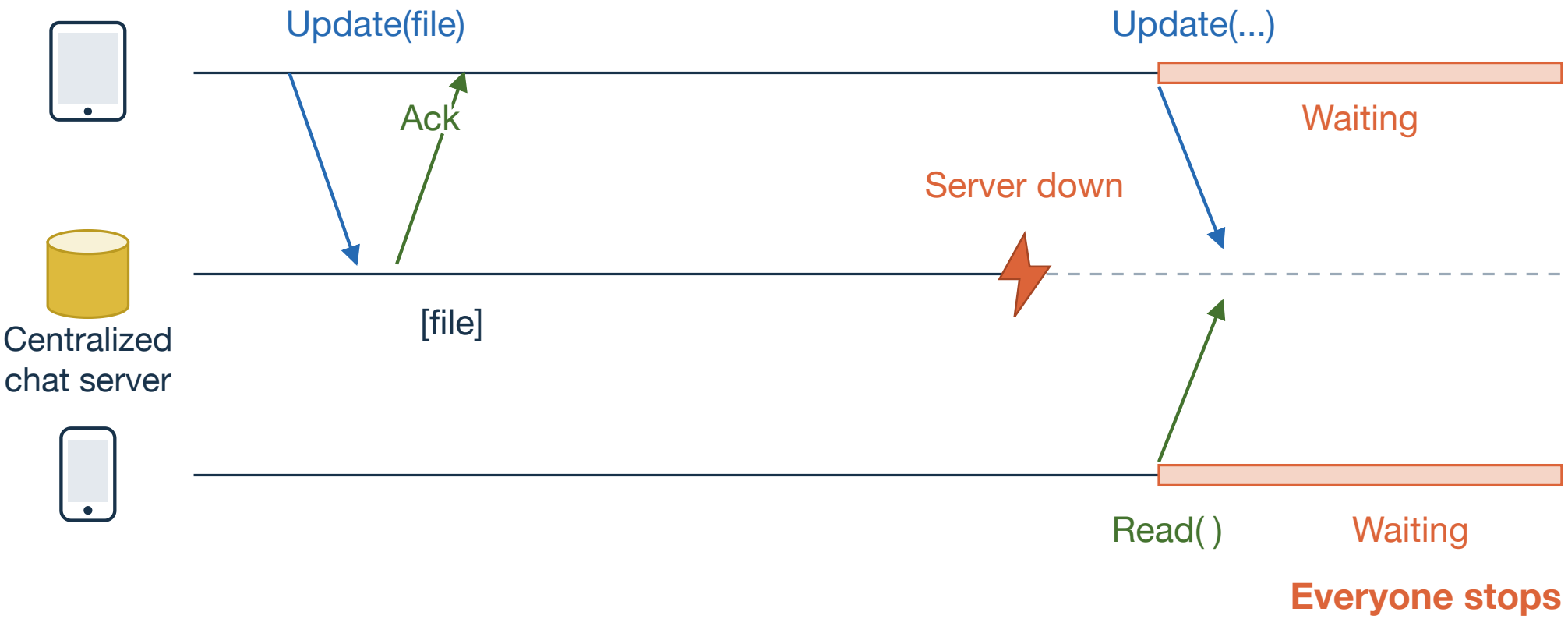
- The server is down, everybody stops
- Though we can make the server highly available (see the Paxos and Raft lecture)



Cons #4 (Reliability): cannot deal with server failures

In practice, we rarely see a ChatAPP adopt this implementation

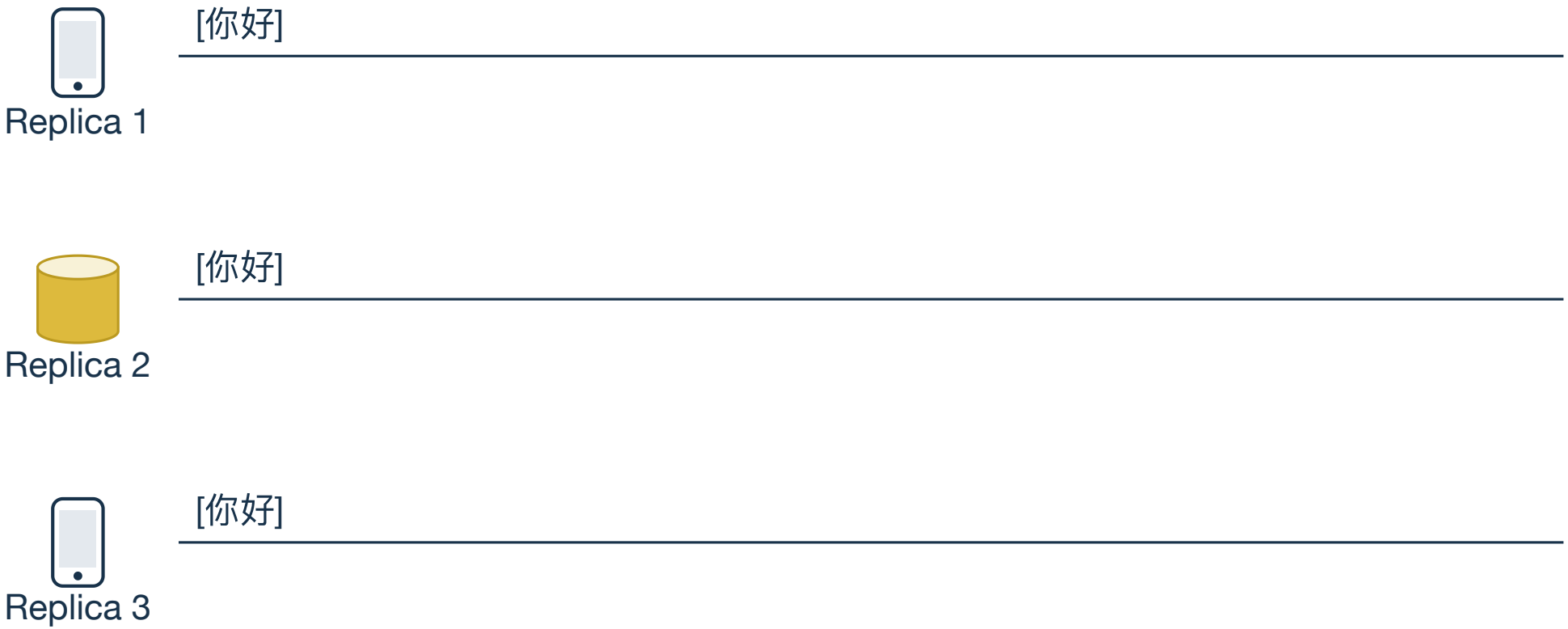
- The server is down, everybody stops
- Though we can make the server highly available (see the Paxos and Raft lecture)



Existing systems (like WeChat) replicate + cache the system state

Store the chat on every participant — a replicated list

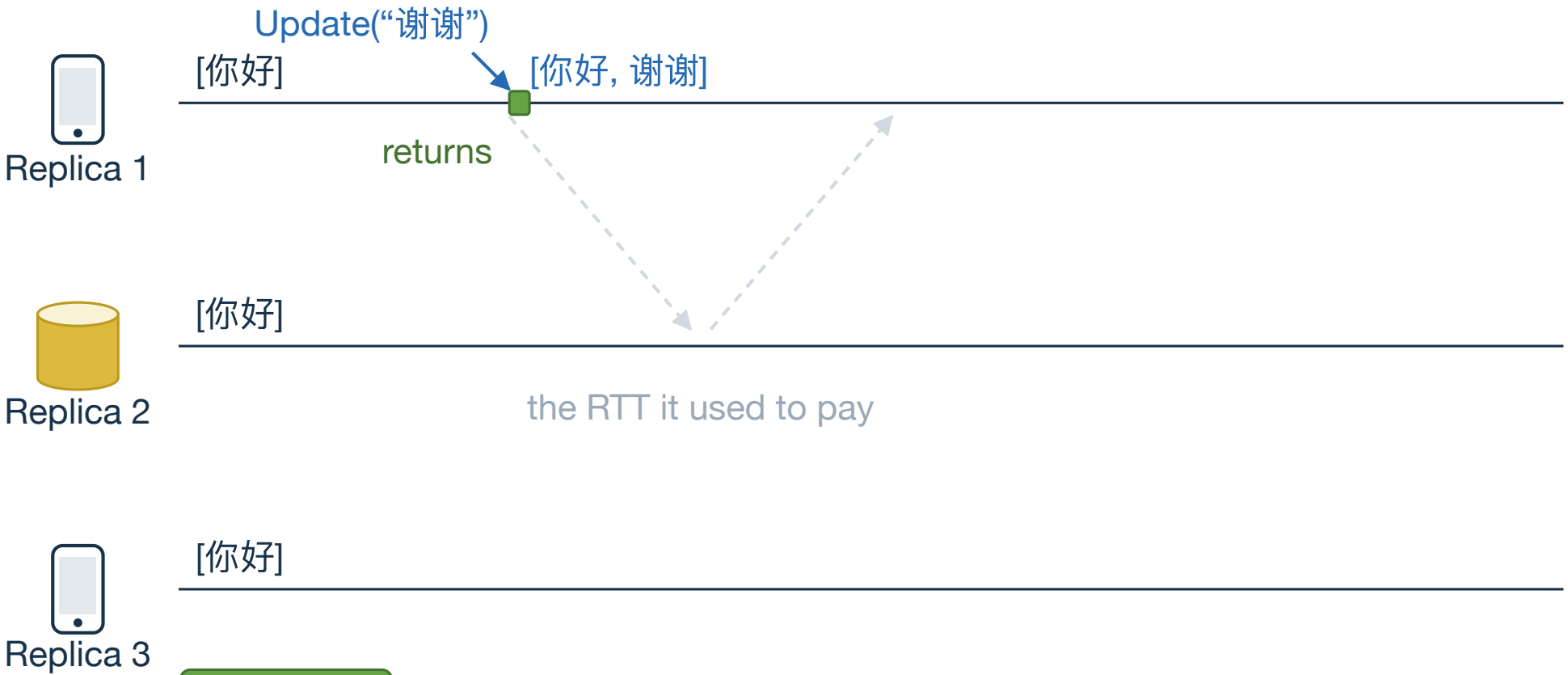
- Here: the two devices and the server, each with a copy of its own
- One copy per device addresses all the four cons of the centralized design



Existing systems (like WeChat) replicate + cache the system state

Store the chat on every participant — a replicated list

- Here: the two devices and the server, each with a copy of its own
- One copy per device addresses all the four cons of the centralized design



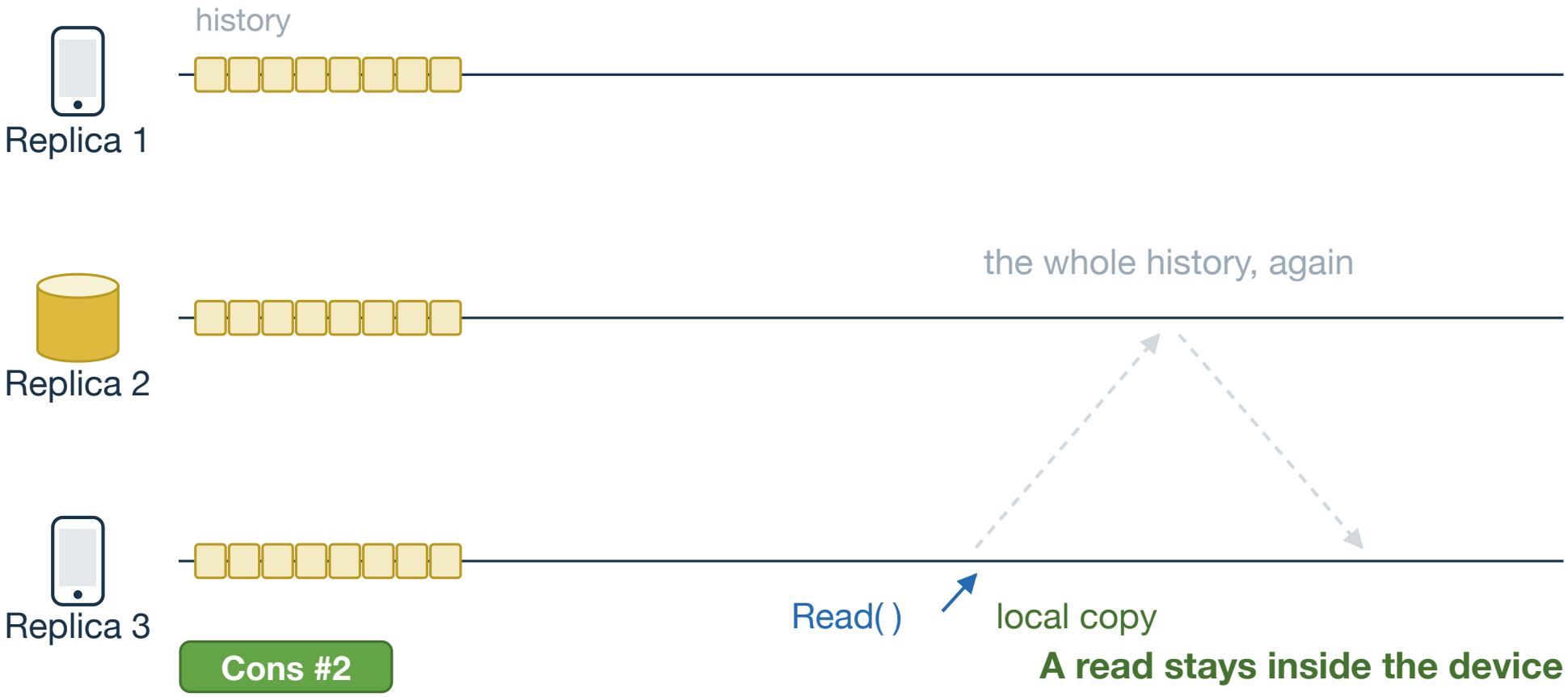
Cons #1

The update no longer waits for an RTT

Existing systems (like WeChat) replicate + cache the system state

Store the chat on every participant — a replicated list

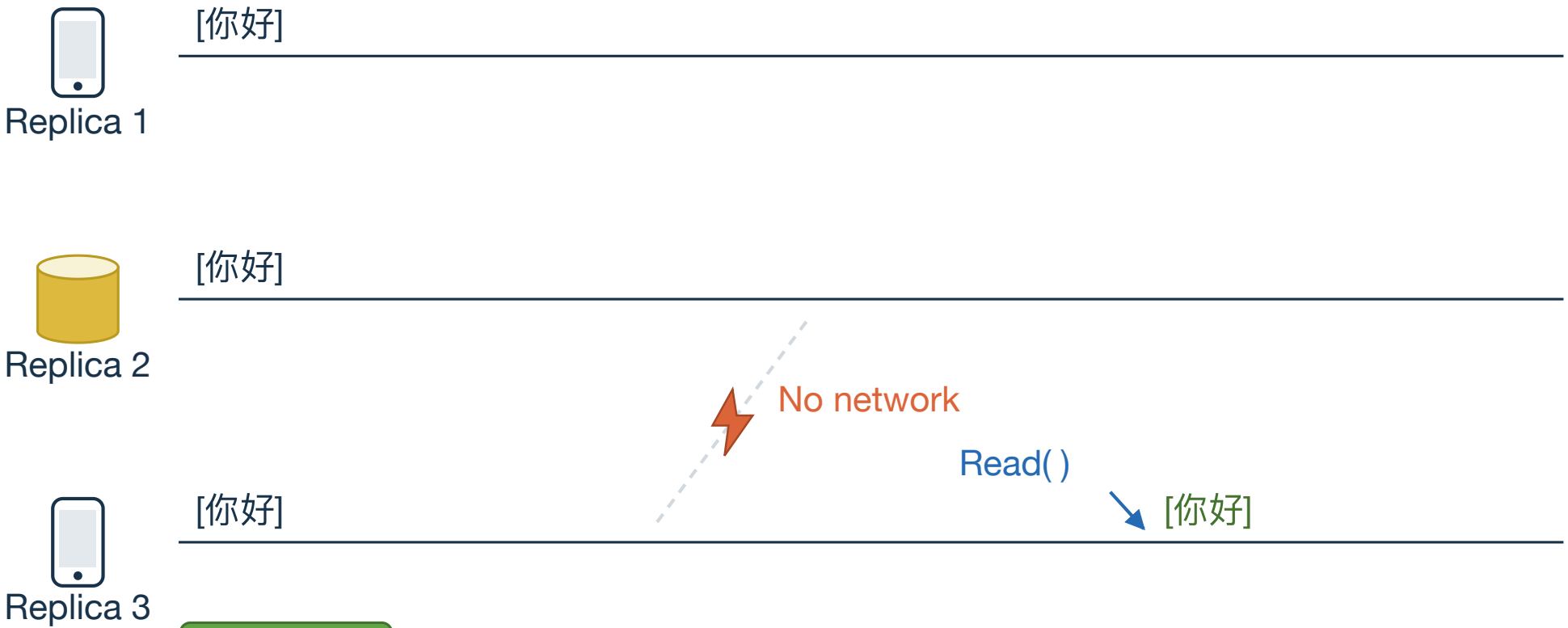
- Here: the two devices and the server, each with a copy of its own
- One copy per device addresses all the four cons of the centralized design



Existing systems (like WeChat) replicate + cache the system state

Store the chat on every participant — a replicated list

- Here: the two devices and the server, each with a copy of its own
- One copy per device addresses all the four cons of the centralized design



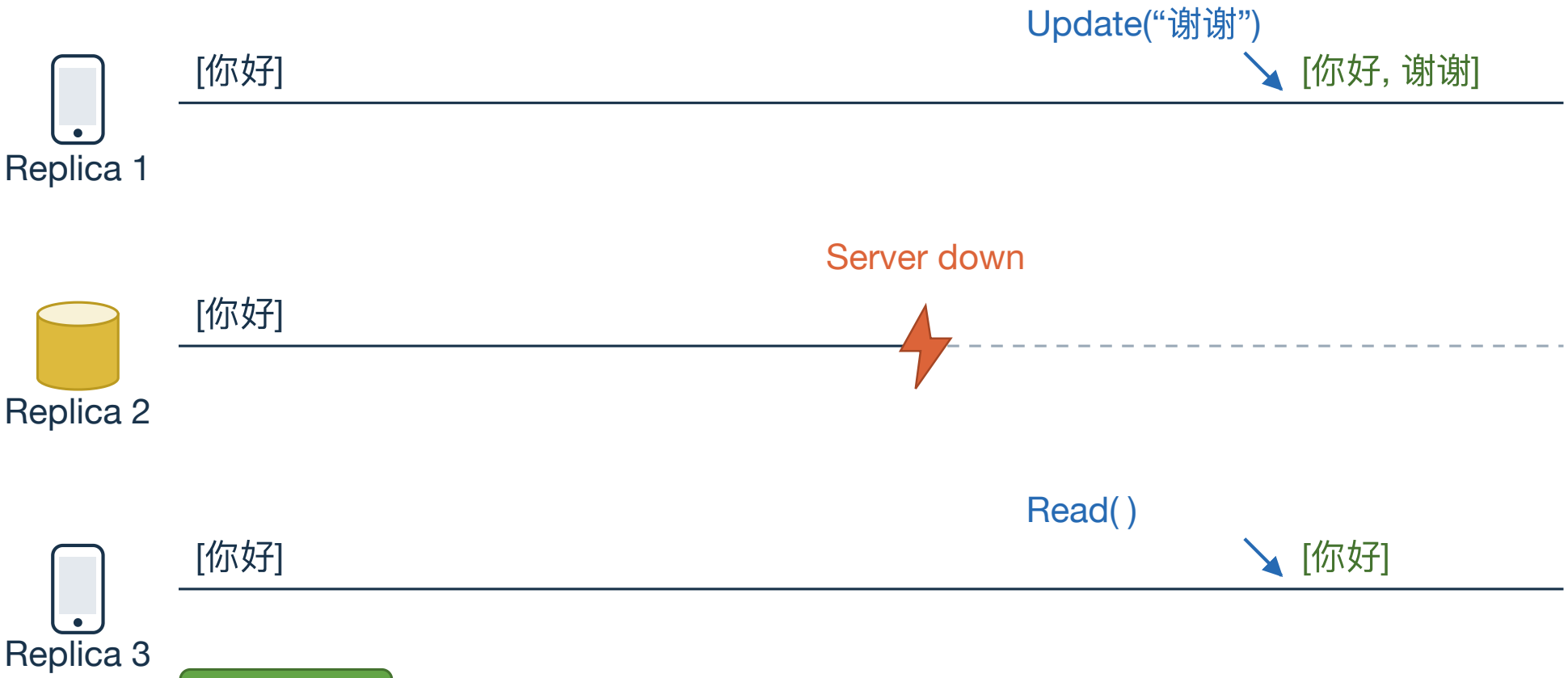
Cons #3

The local copy still answers

Existing systems (like WeChat) replicate + cache the system state

Store the chat on every participant — a replicated list

- Here: the two devices and the server, each with a copy of its own
- One copy per device addresses all the four cons of the centralized design



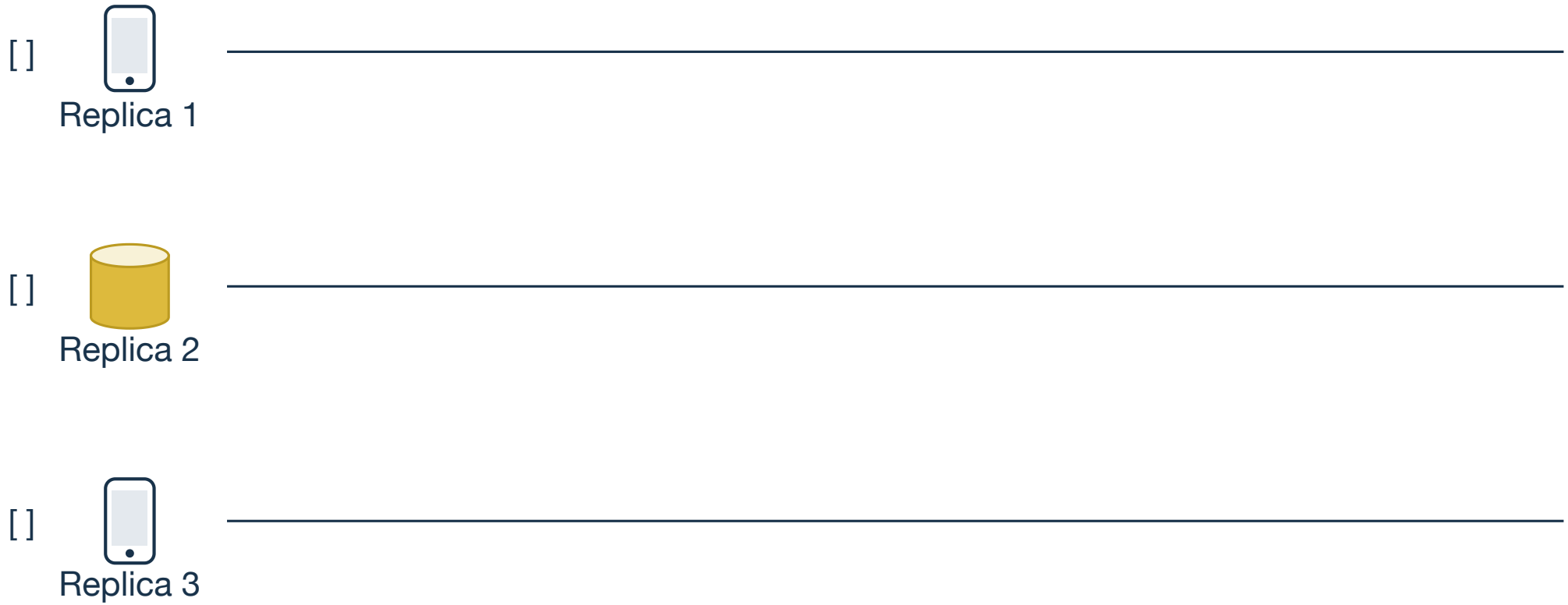
Cons #4

Every copy keeps serving its user

Challenge: the implementations of reads/writes

Take `update` as an example. One choice: only modify the local copy

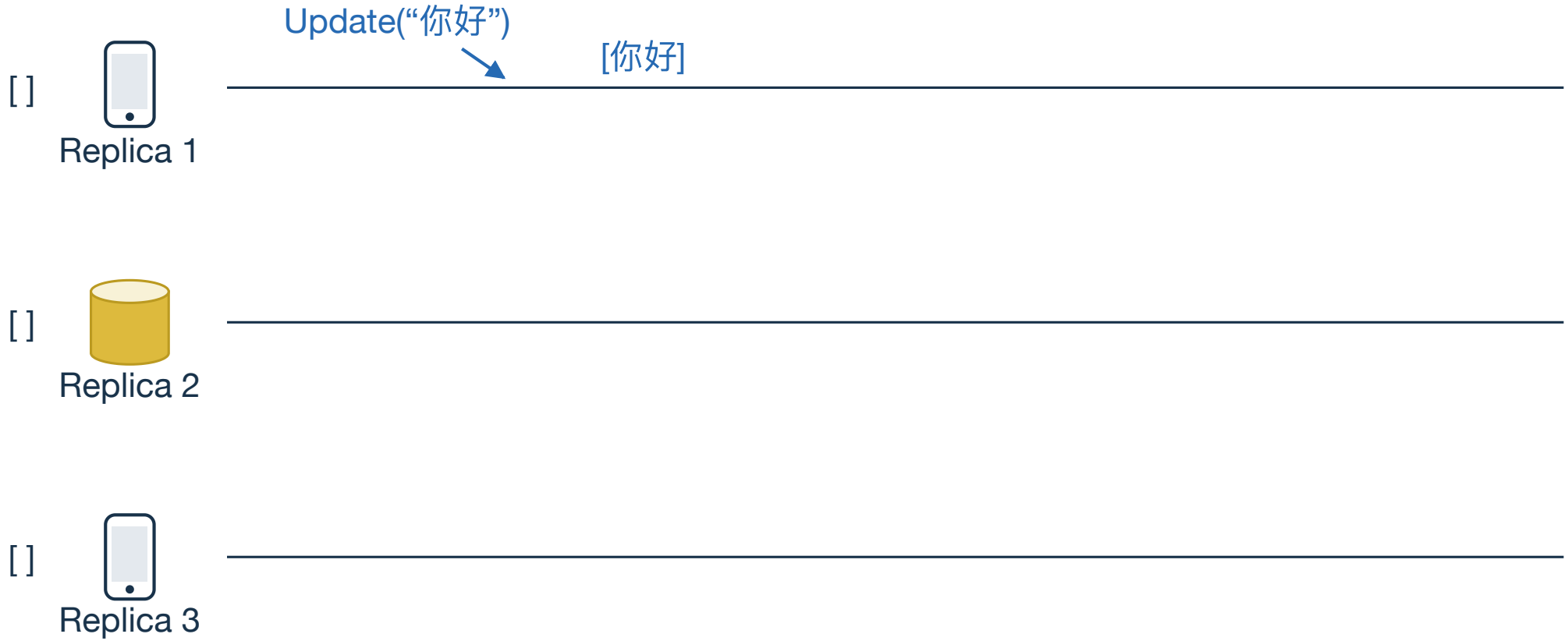
- The fastest possible update: no network on the critical path
- Problem: the replicas now diverge — how do we synchronize them?



Challenge: the implementations of reads/writes

Take `update` as an example. One choice: only modify the local copy

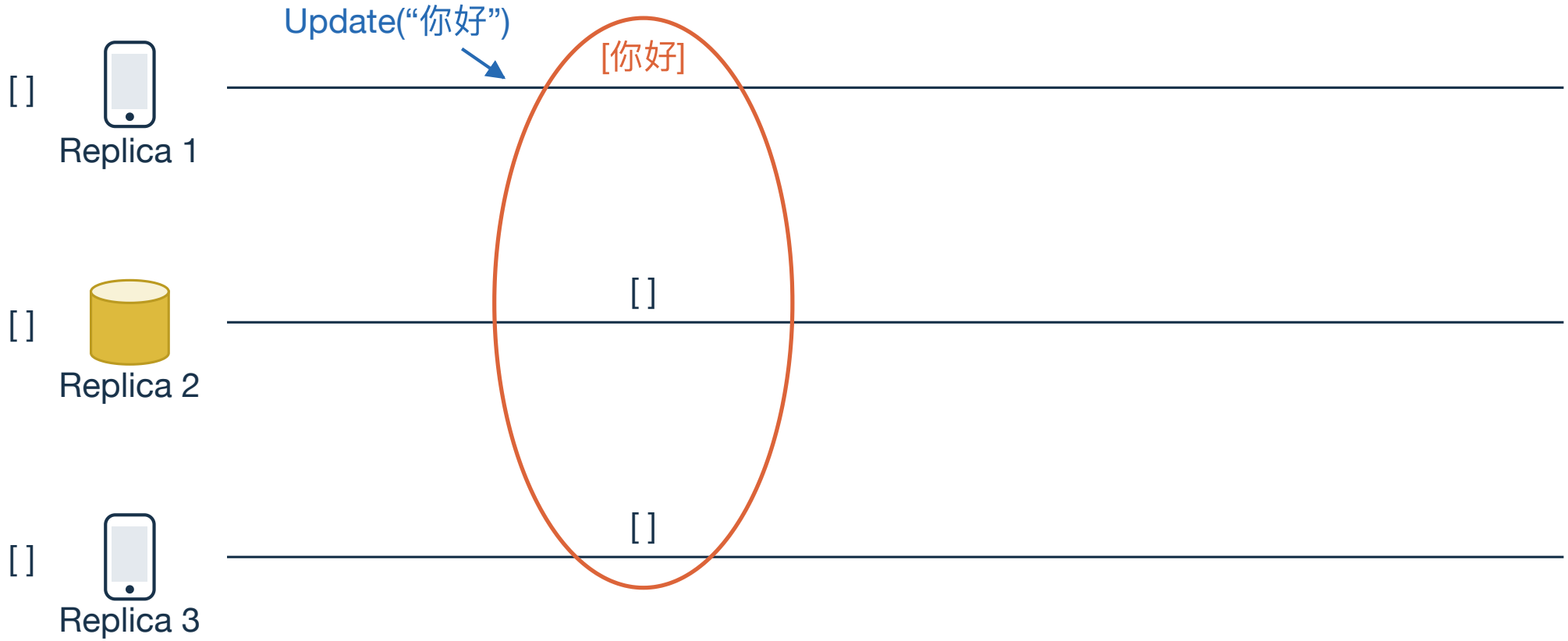
- The fastest possible update: no network on the critical path
- Problem: the replicas now diverge — how do we synchronize them?



Challenge: the implementations of reads/writes

Take `update` as an example. One choice: only modify the local copy

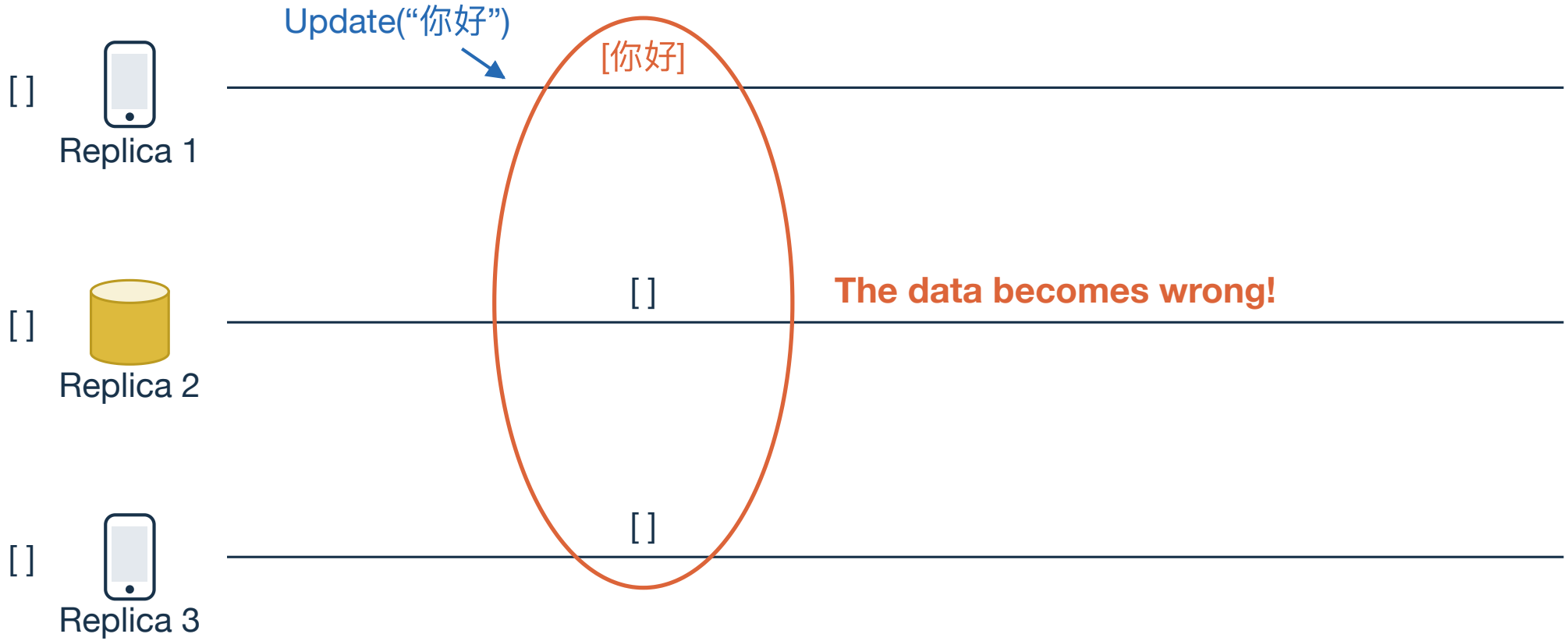
- The fastest possible update: no network on the critical path
- Problem: the replicas now diverge — how do we synchronize them?



Challenge: the implementations of reads/writes

Take `update` as an example. One choice: only modify the local copy

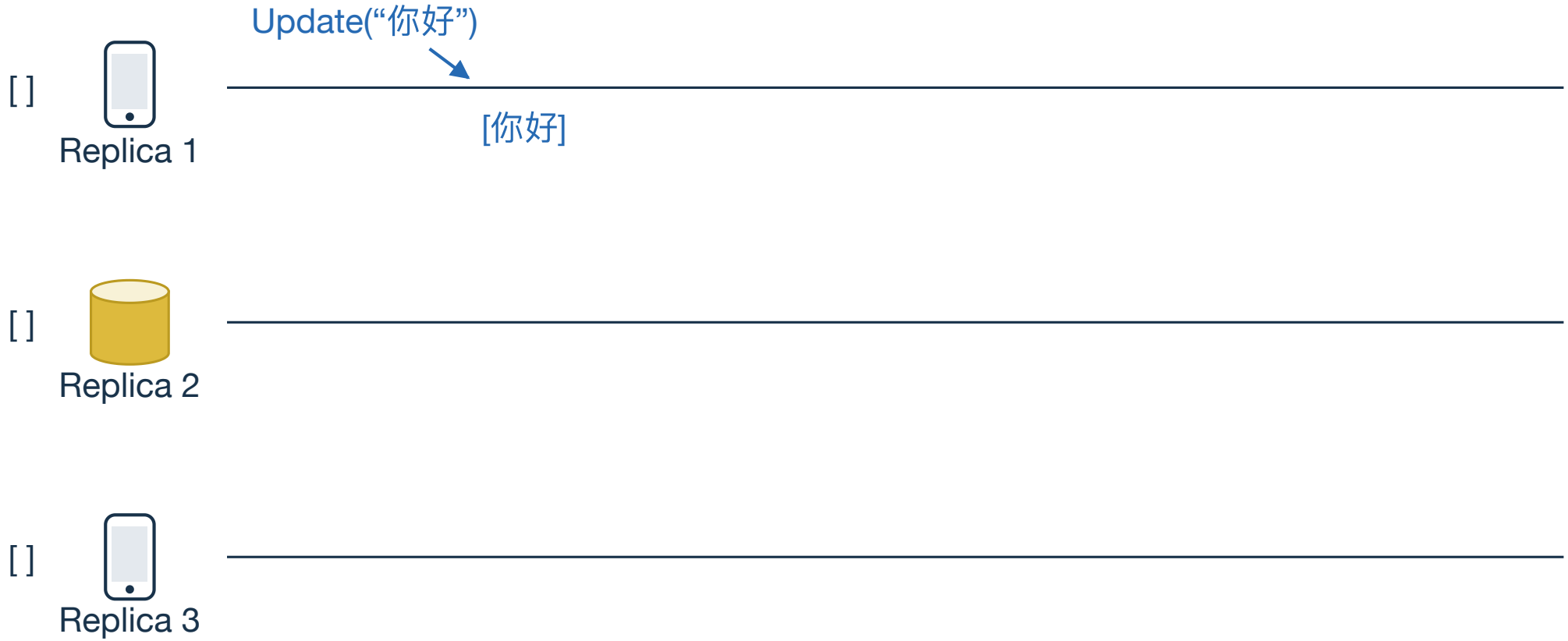
- The fastest possible update: no network on the critical path
- Problem: the replicas now diverge — how do we synchronize them?



The remedy for the update: sync the operation with all the replicas

Another question arises: how many replicas must an update reach before it is done?

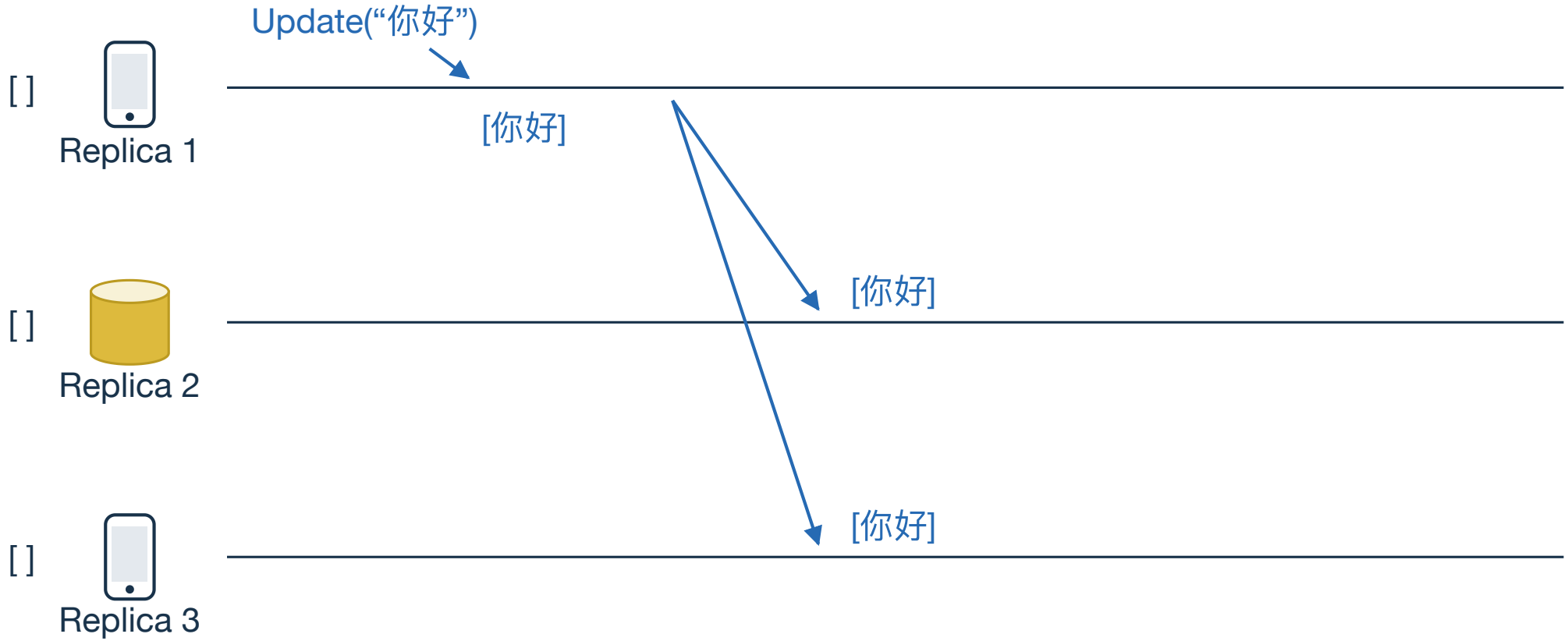
- Approach #1: sync with all of them, and reply only after every ack
- Issues: the slowest replica sets the latency, and one failure blocks the update



The remedy for the update: sync the operation with all the replicas

Another question arises: how many replicas must an update reach before it is done?

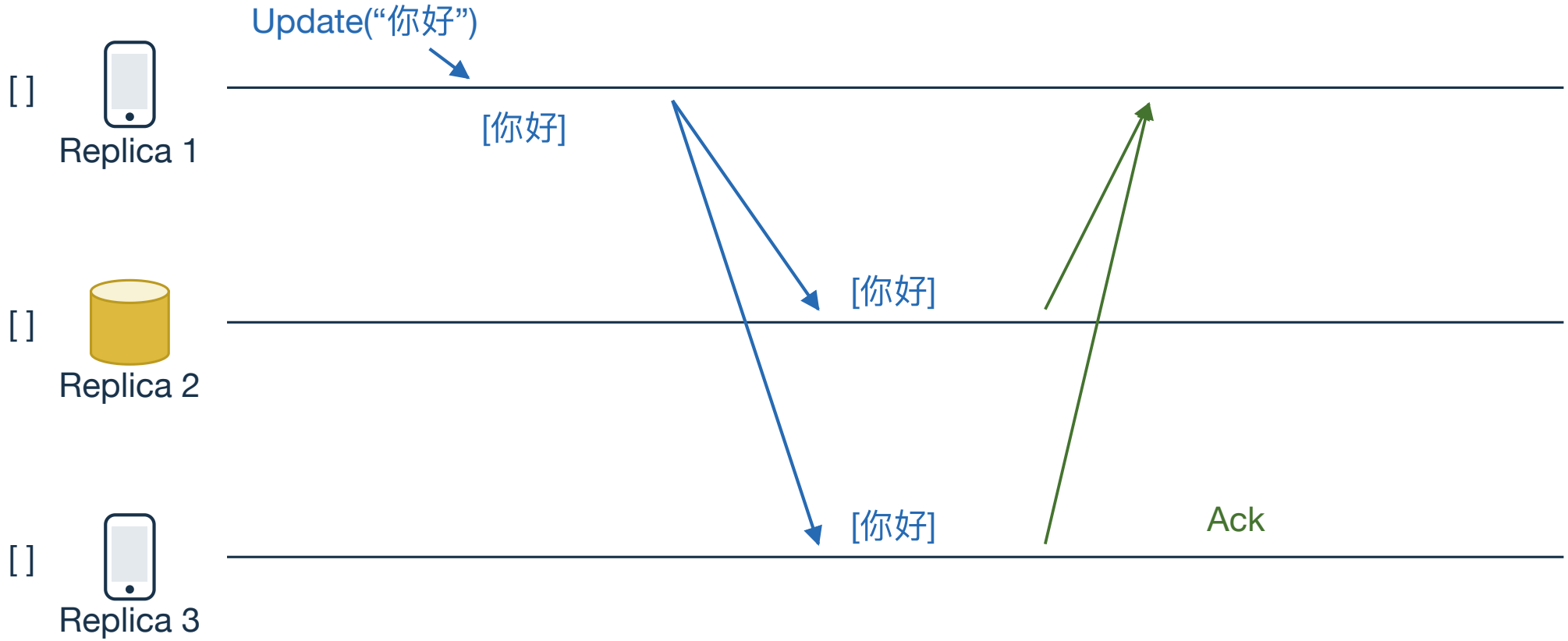
- Approach #1: sync with all of them, and reply only after every ack
- Issues: the slowest replica sets the latency, and one failure blocks the update



The remedy for the update: sync the operation with all the replicas

Another question arises: how many replicas must an update reach before it is done?

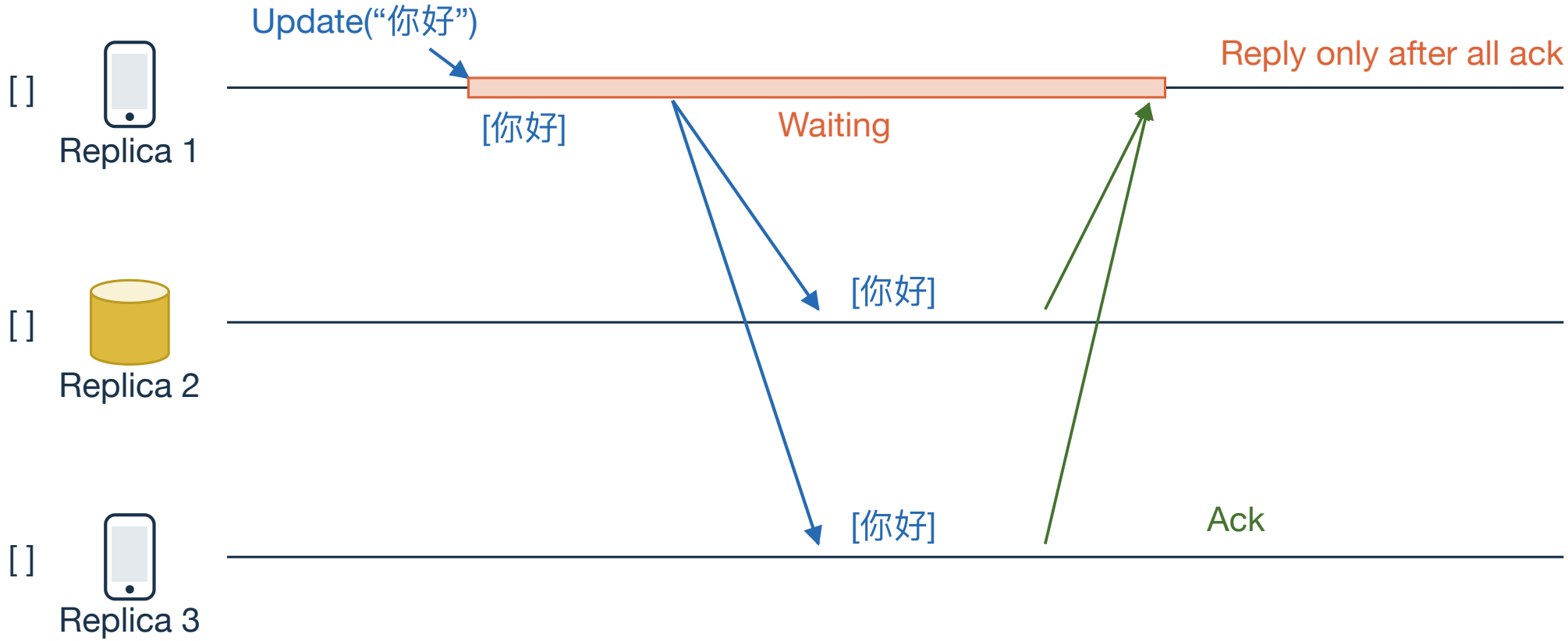
- Approach #1: sync with all of them, and reply only after every ack
- Issues: the slowest replica sets the latency, and one failure blocks the update



The remedy for the update: sync the operation with all the replicas

Another question arises: how many replicas must an update reach before it is done?

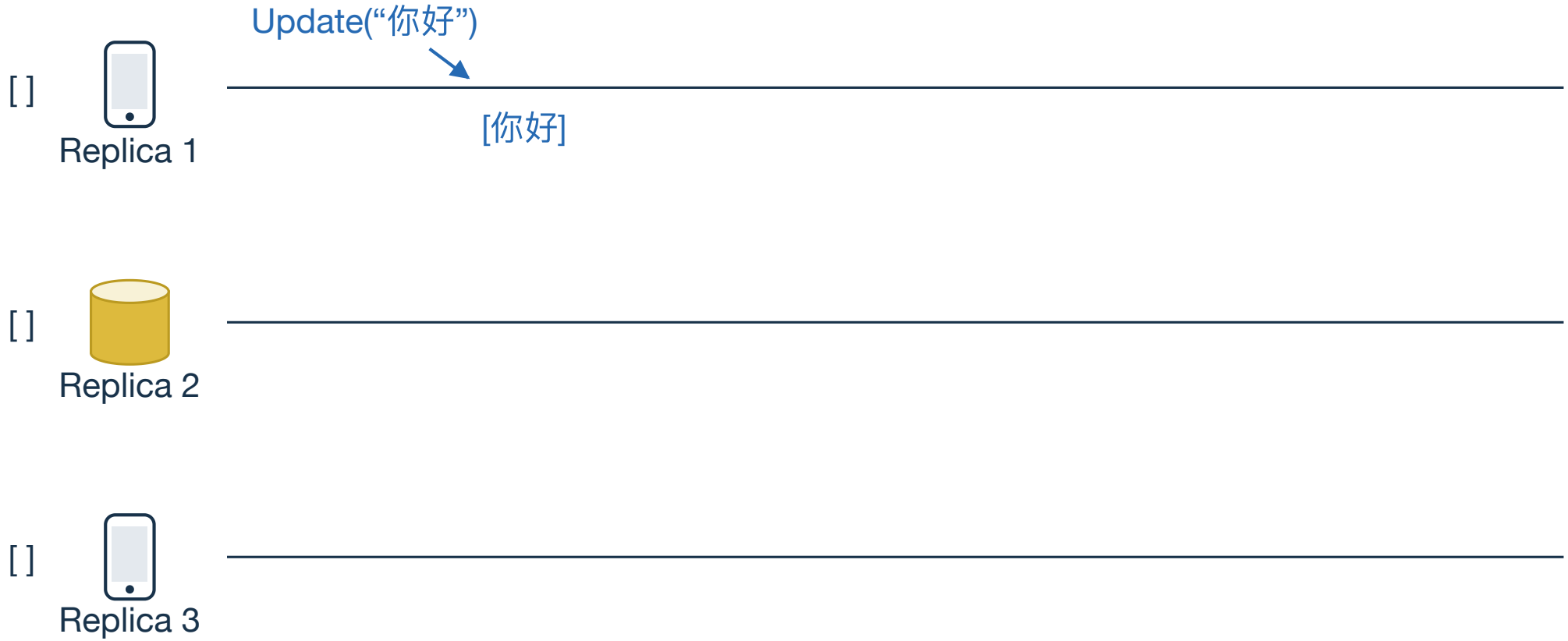
- Approach #1: sync with all of them, and reply only after every ack
- Issues: the slowest replica sets the latency, and one failure blocks the update



For performance/reliability, sync needs to be done on a few replicas

How many replicas must an update reach before it is done?

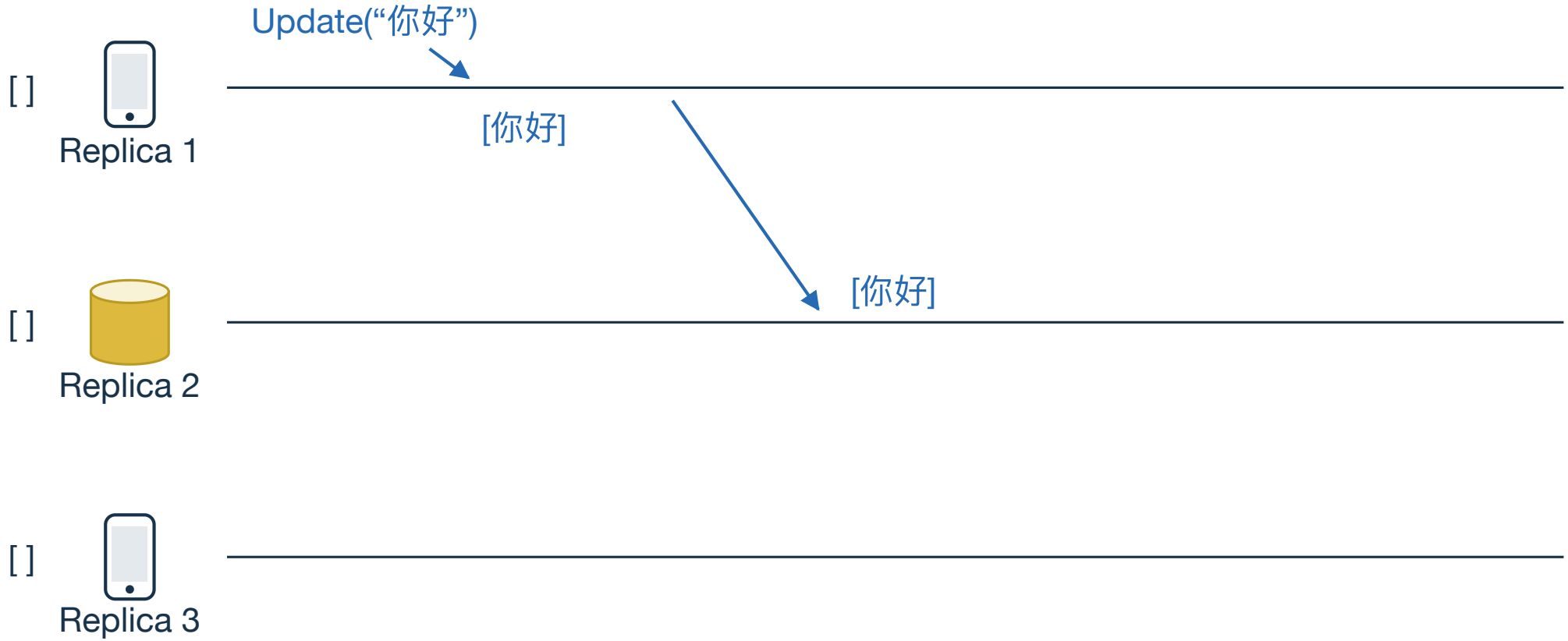
- Common approach: contact a few, reply once some (e.g., the server) have acknowledged
- The replica the update missed can catch up later, in the background



For performance/reliability, sync needs to be done on a few replicas

How many replicas must an update reach before it is done?

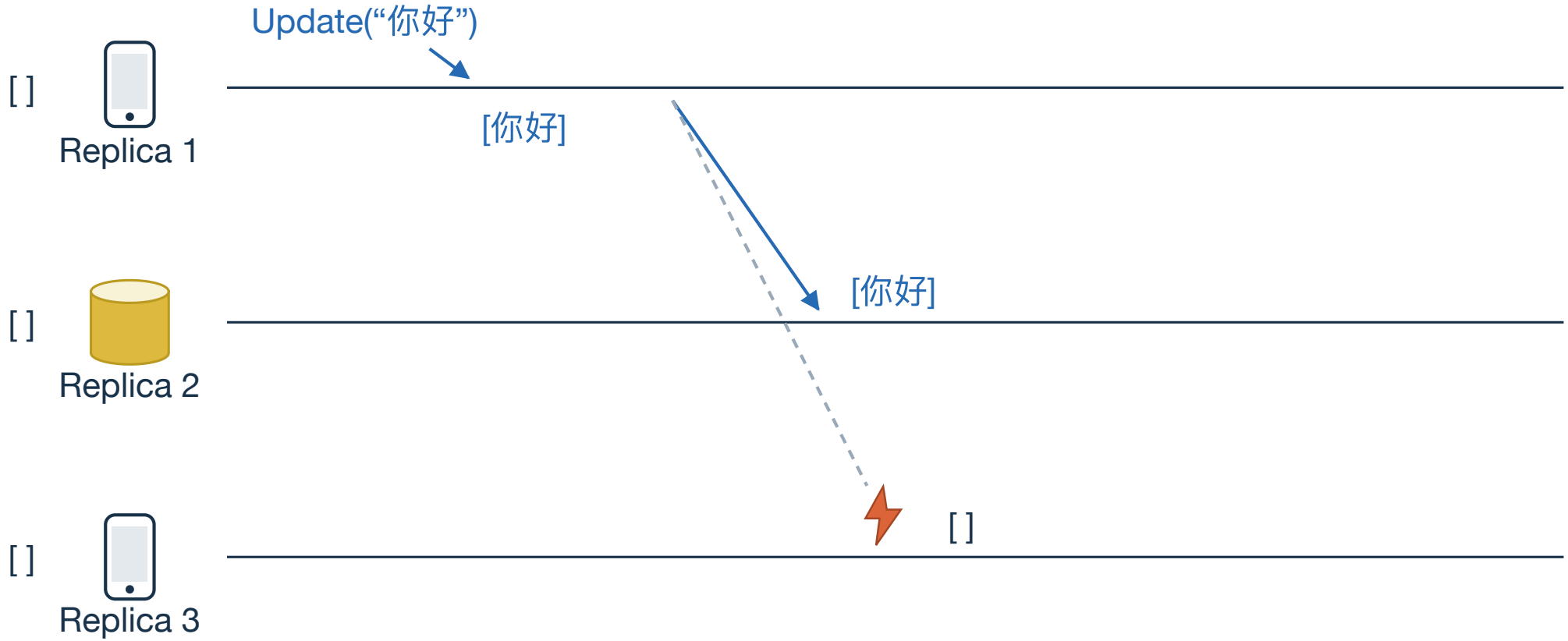
- Common approach: contact a few, reply once some (e.g., the server) have acknowledged
- The replica the update missed can catch up later, in the background



For performance/reliability, sync needs to be done on a few replicas

How many replicas must an update reach before it is done?

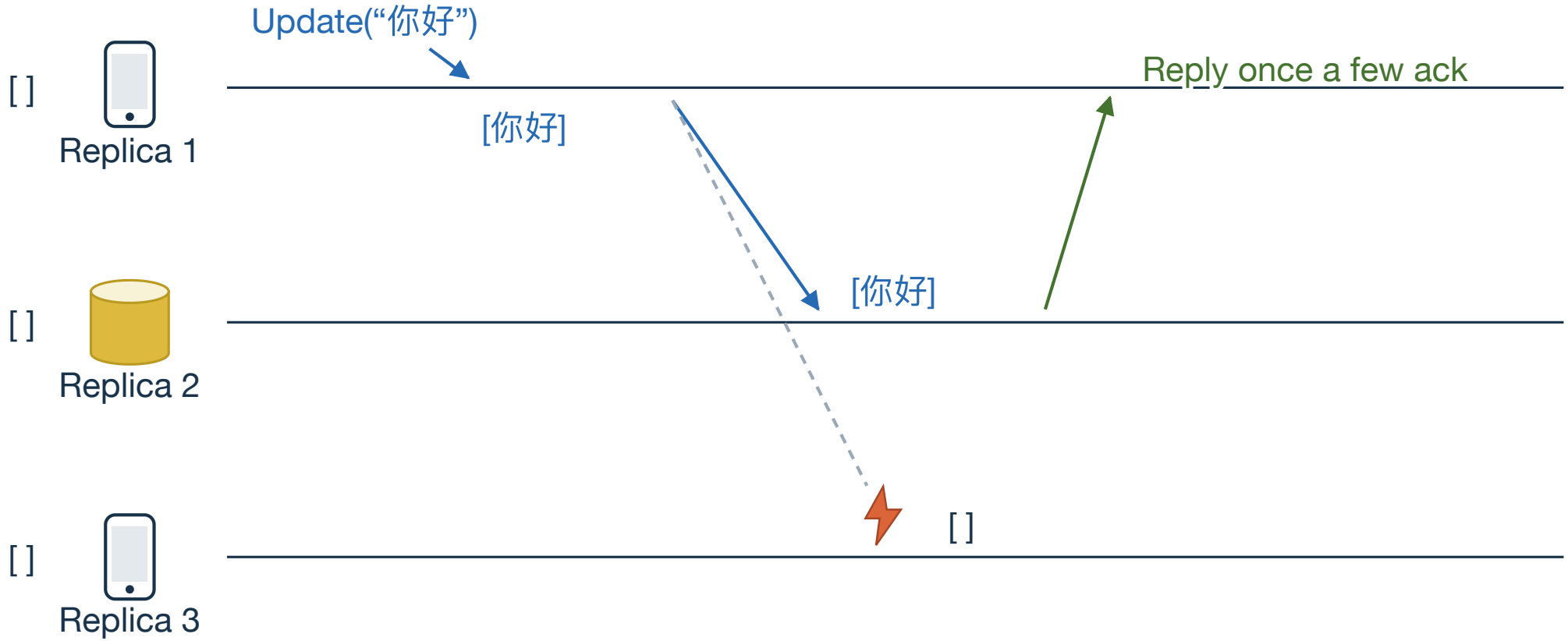
- Common approach: contact a few, reply once some (e.g., the server) have acknowledged
- The replica the update missed can catch up later, in the background



For performance/reliability, sync needs to be done on a few replicas

How many replicas must an update reach before it is done?

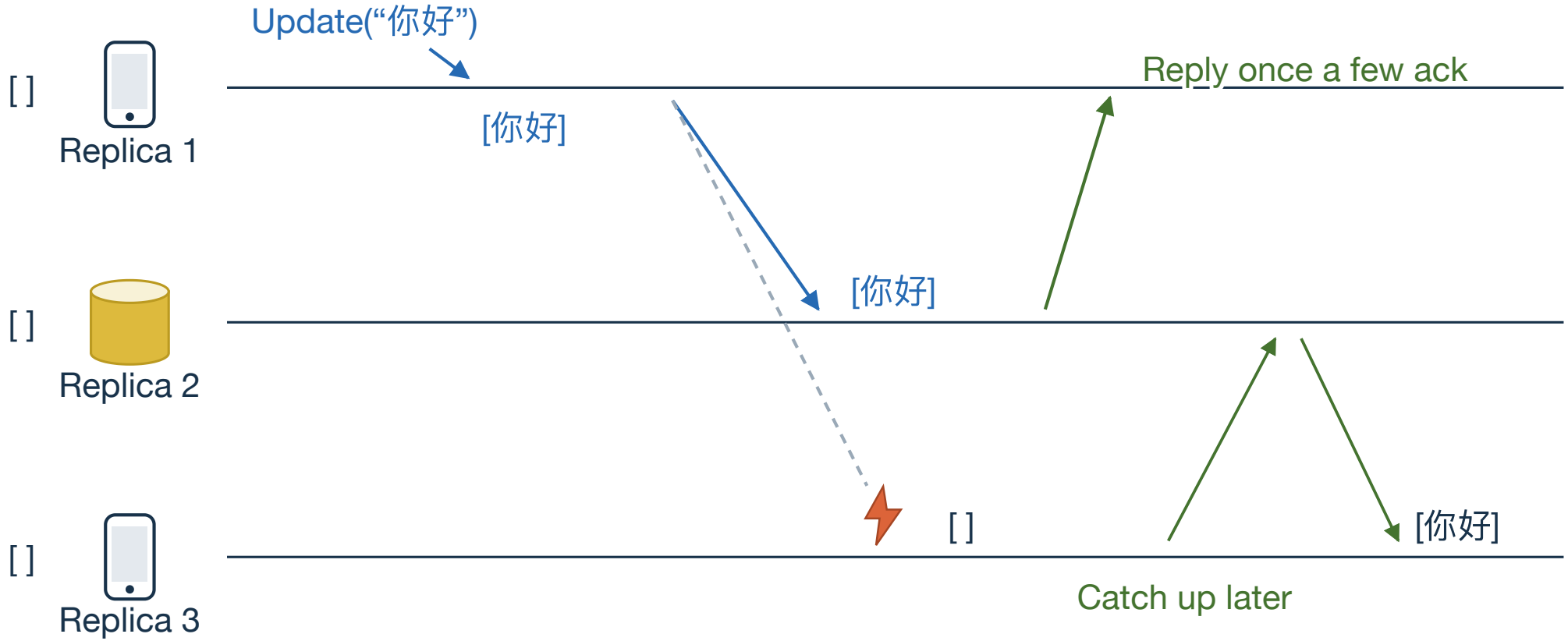
- Common approach: contact a few, reply once some (e.g., the server) have acknowledged
- The replica the update missed can catch up later, in the background



For performance/reliability, sync needs to be done on a few replicas

How many replicas must an update reach before it is done?

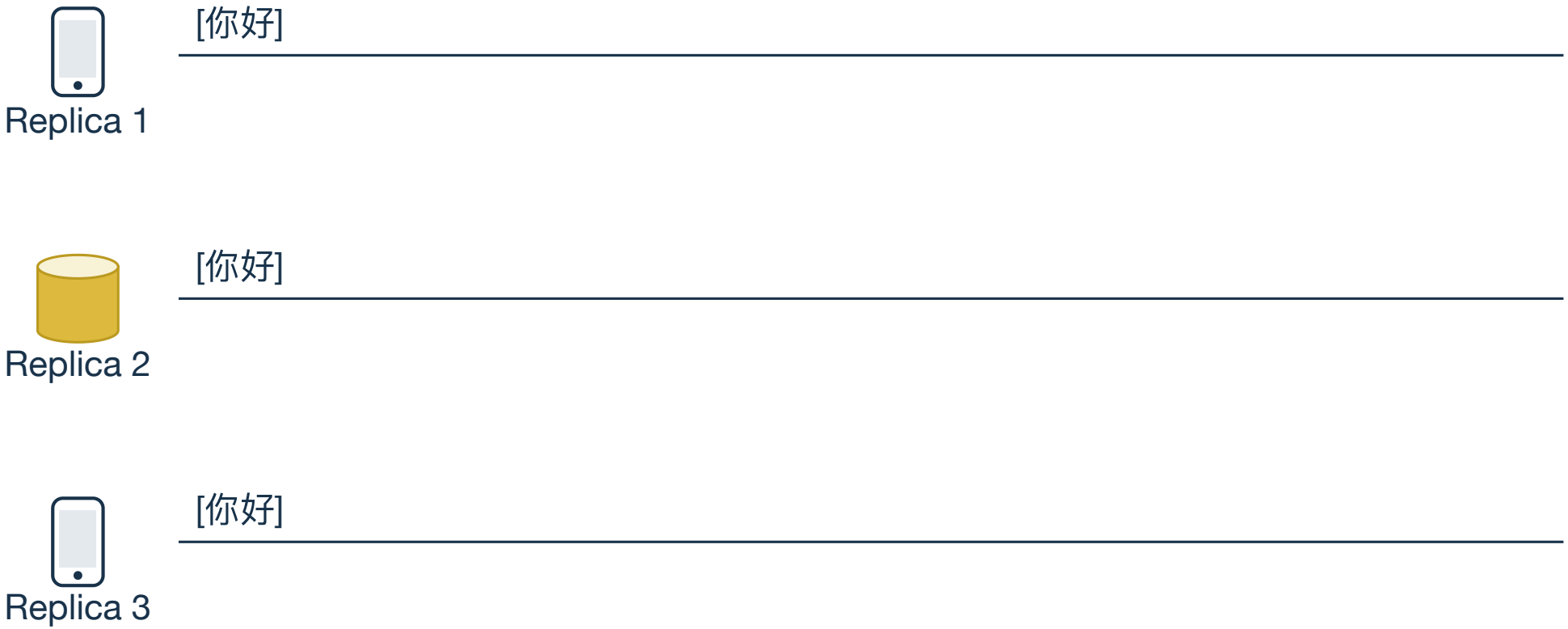
- Common approach: contact a few, reply once some (e.g., the server) have acknowledged
- The replica the update missed can catch up later, in the background



Next question: what about reads?

Most efficient: read the local copy

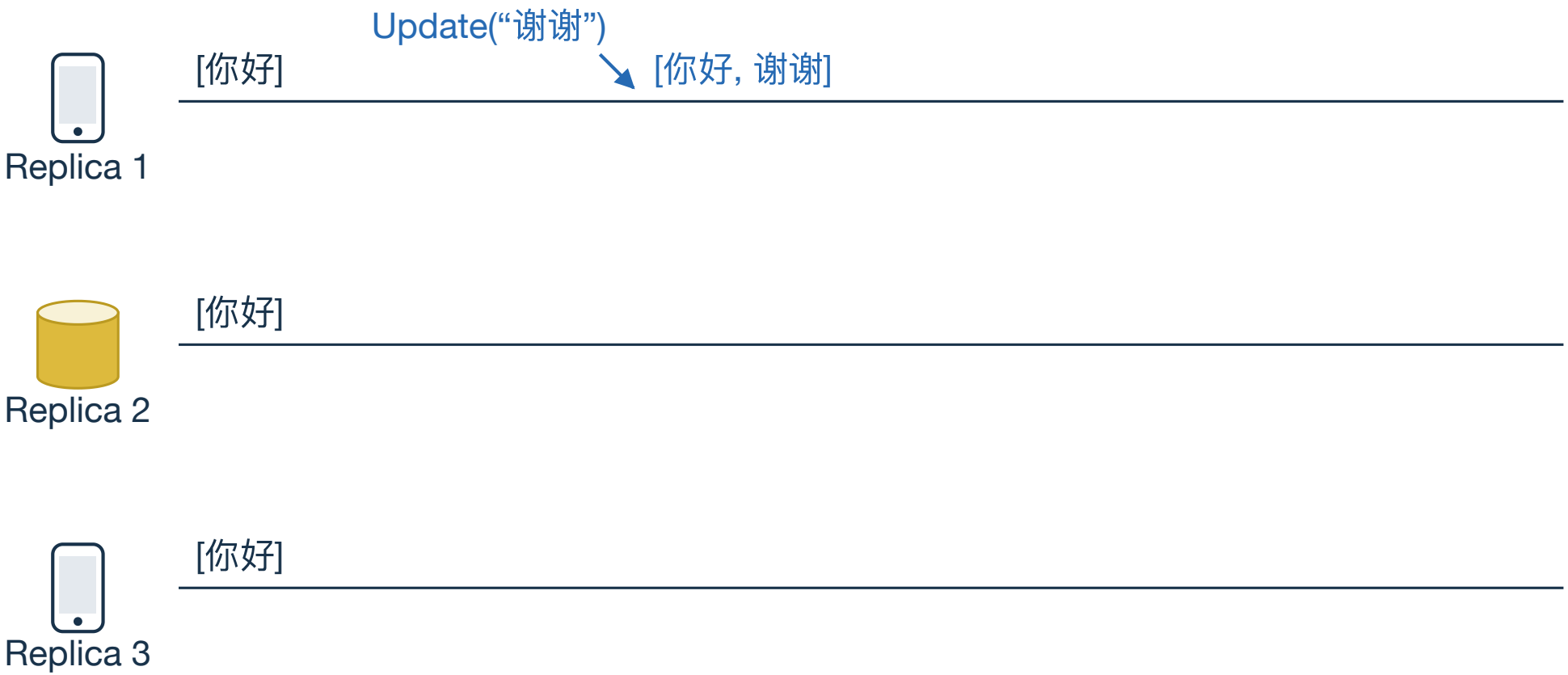
- Pros: no network on the read path, fast and works offline
- Cons: may miss the most up-to-date data



Next question: what about reads?

Most efficient: read the local copy

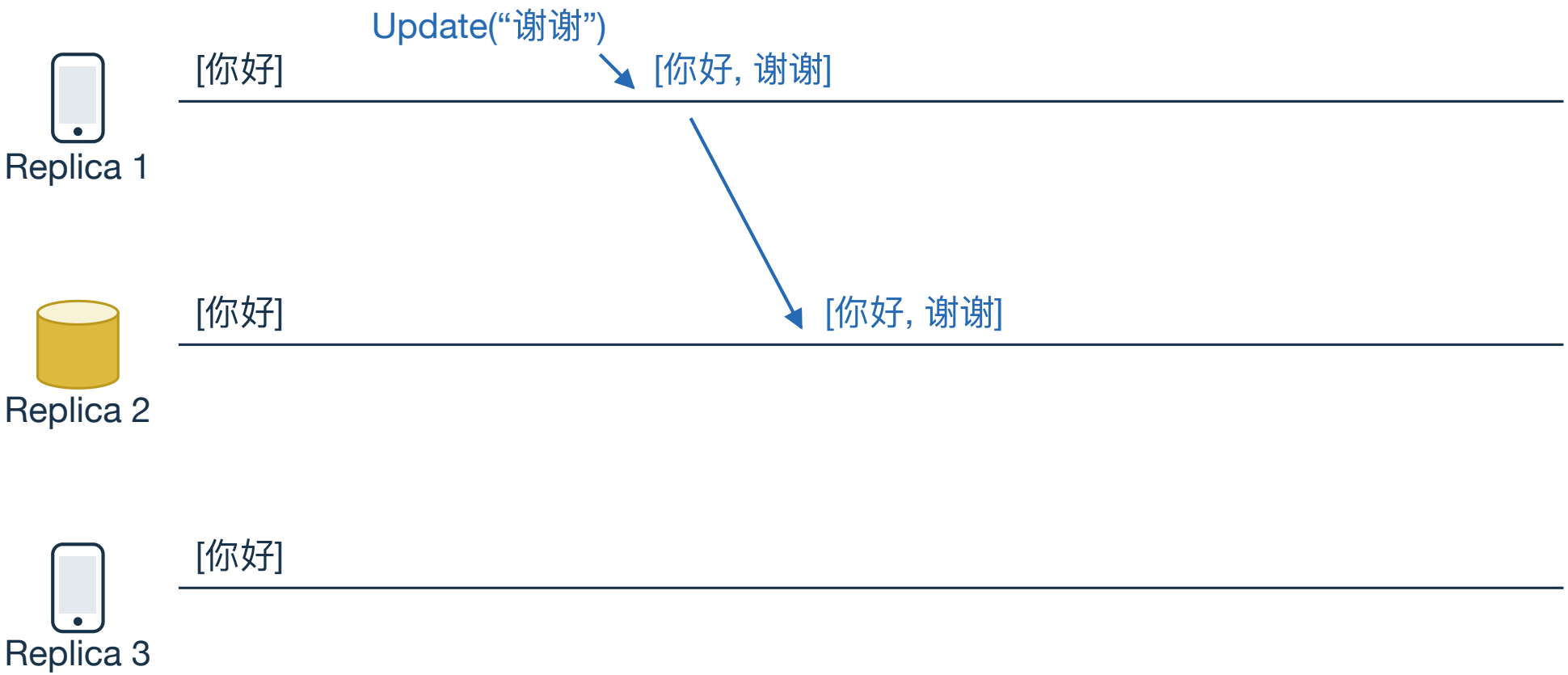
- Pros: no network on the read path, fast and works offline
- Cons: may miss the most up-to-date data



Next question: what about reads?

Most efficient: read the local copy

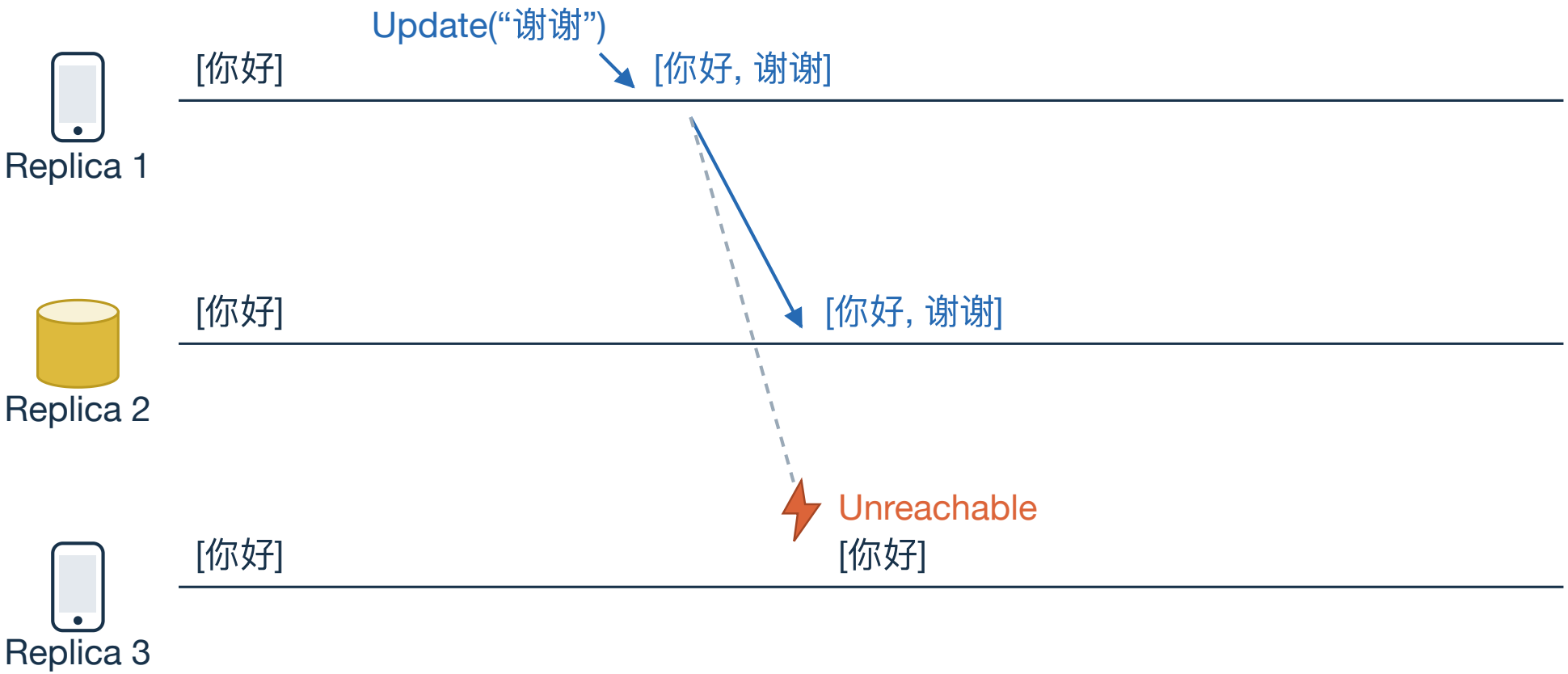
- Pros: no network on the read path, fast and works offline
- Cons: may miss the most up-to-date data



Next question: what about reads?

Most efficient: read the local copy

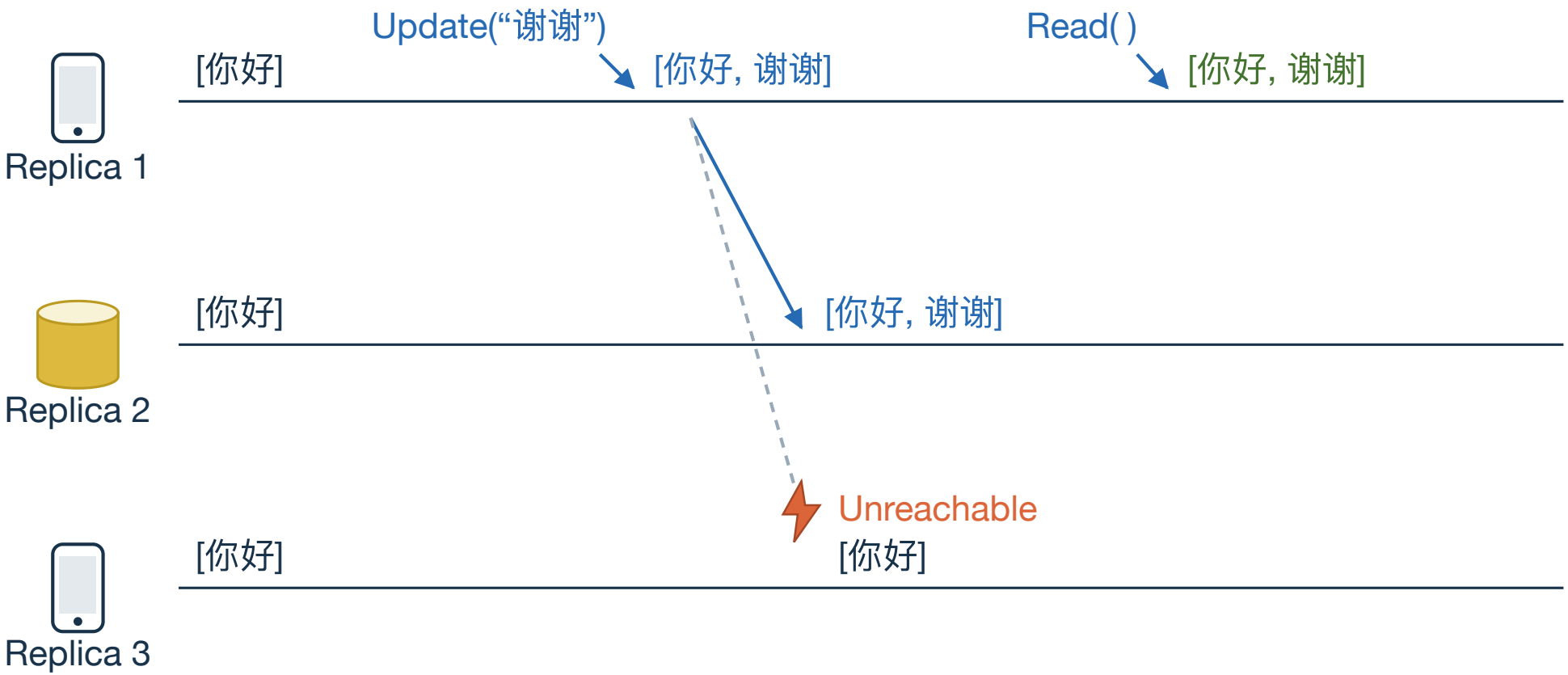
- Pros: no network on the read path, fast and works offline
- Cons: may miss the most up-to-date data



Next question: what about reads?

Most efficient: read the local copy

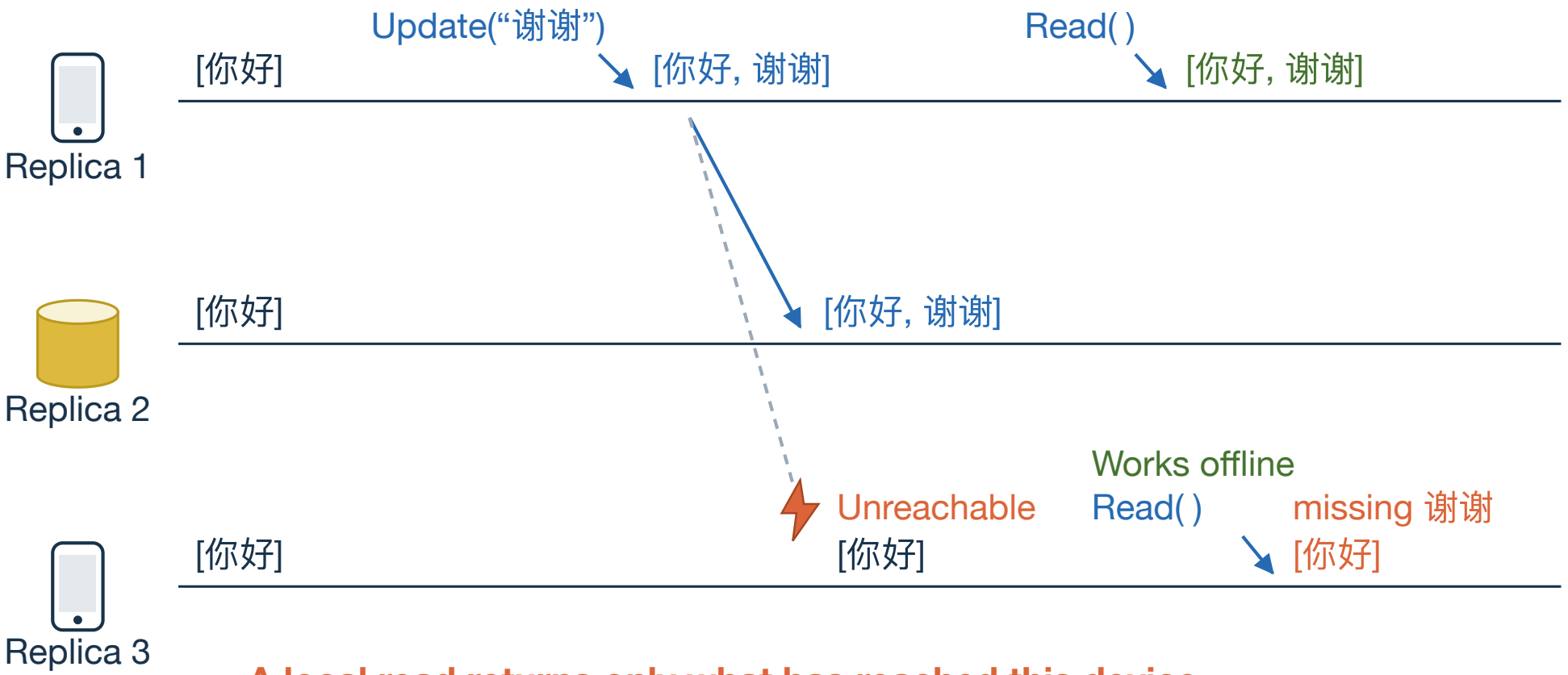
- Pros: no network on the read path, fast and works offline
- Cons: may miss the most up-to-date data



Next question: what about reads?

Most efficient: read the local copy

- Pros: no network on the read path, fast and works offline
- Cons: may miss the most up-to-date data

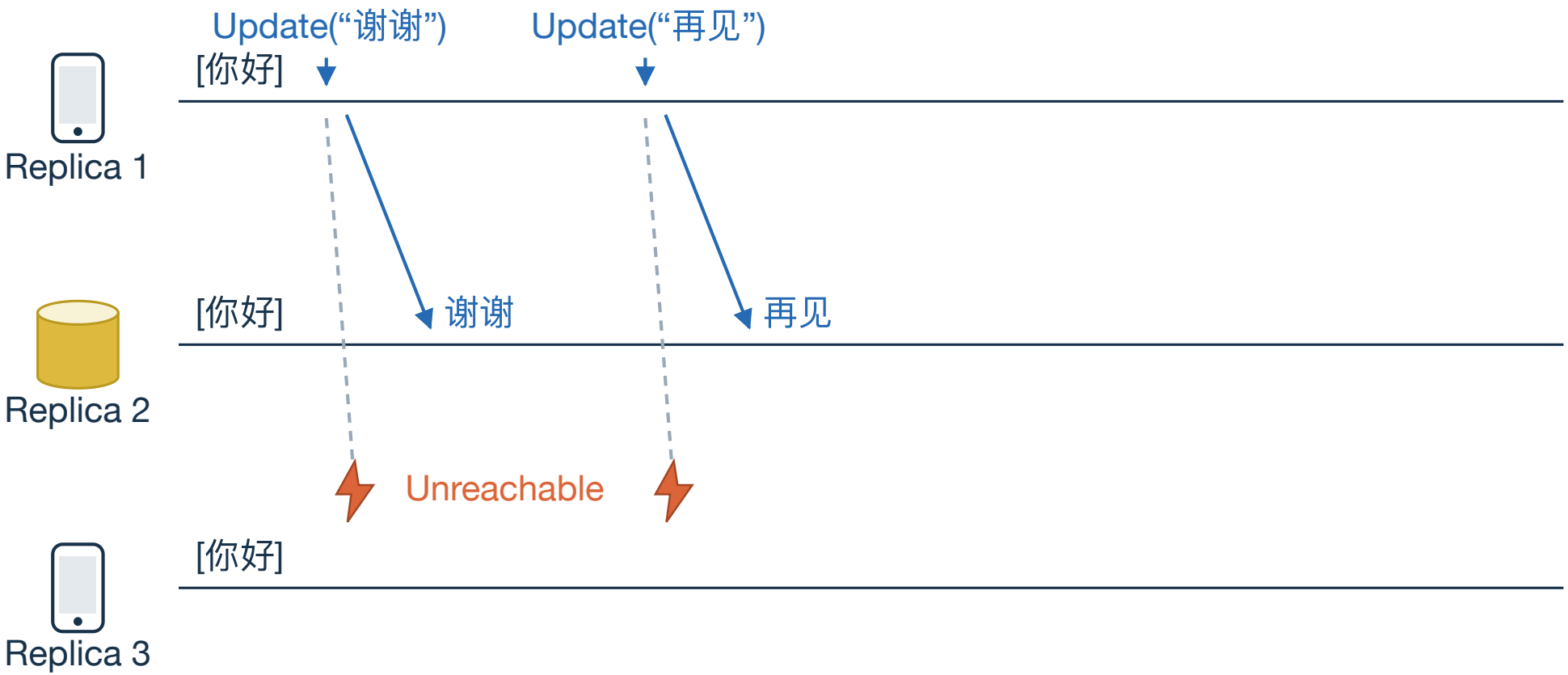


A local read returns only what has reached this device

Reading the local copy with periodic syncs

With periodic syncs, reading the local copy can recover most of the reads

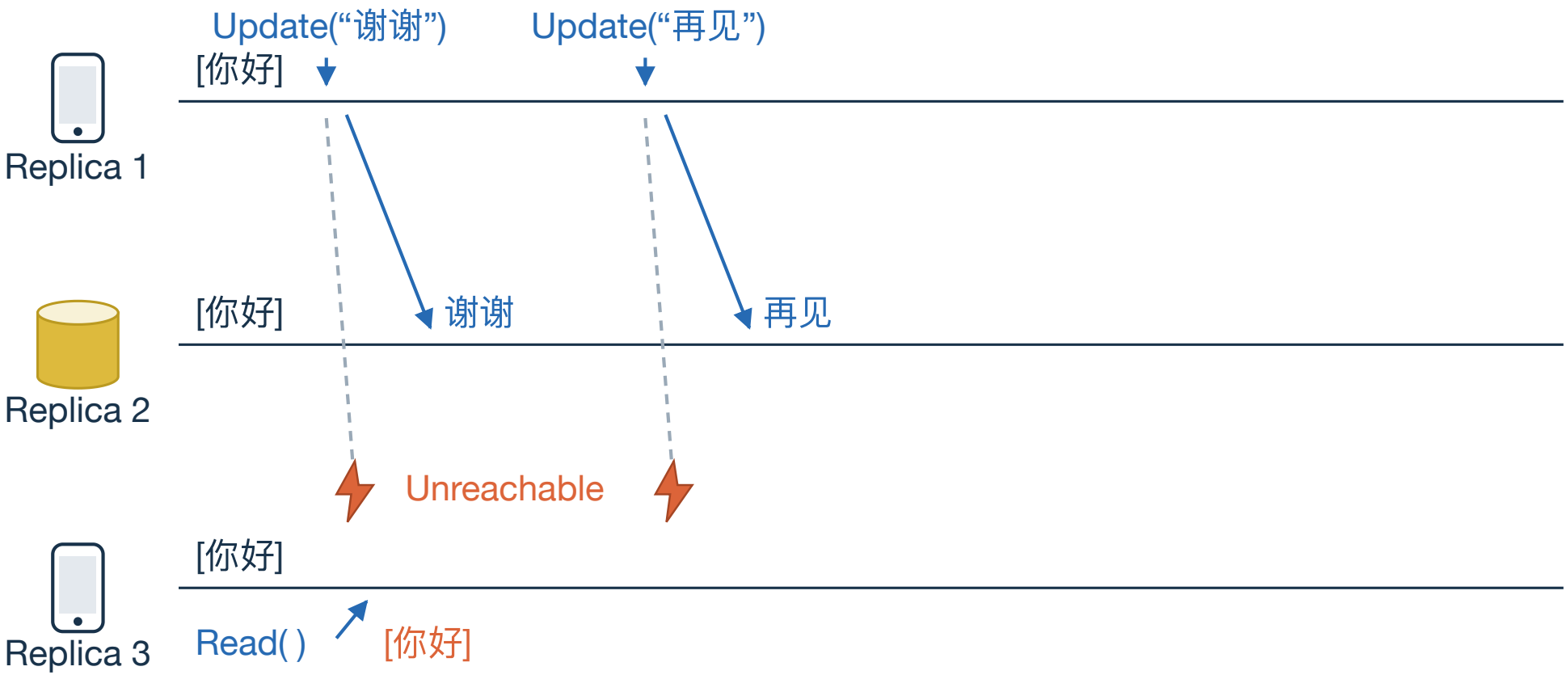
- E.g., a periodic heartbeat, say every 10 seconds
- Or via a service (e.g., Apple Push Notification), batching many syncs



Reading the local copy with periodic syncs

With periodic syncs, reading the local copy can recover most of the reads

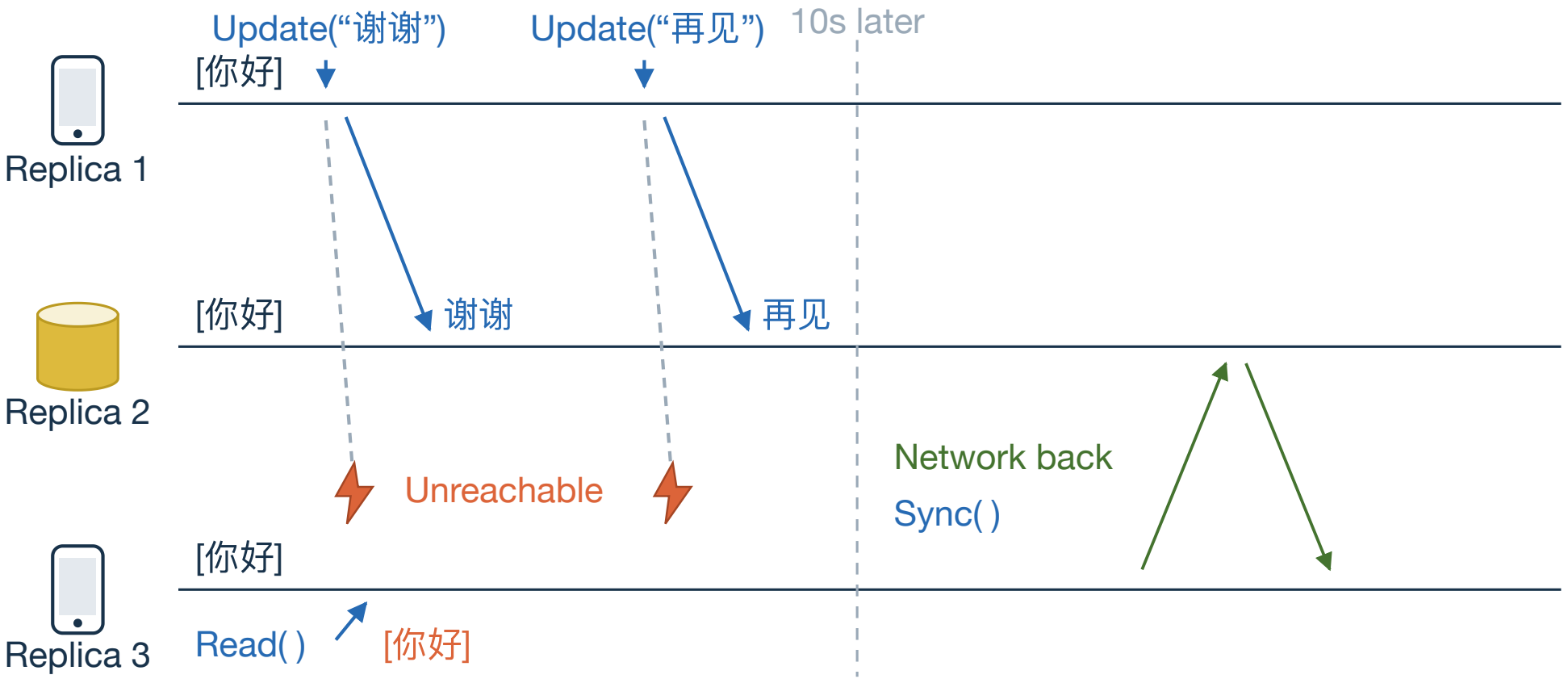
- E.g., a periodic heartbeat, say every 10 seconds
- Or via a service (e.g., Apple Push Notification), batching many syncs



Reading the local copy with periodic syncs

With periodic syncs, reading the local copy can recover most of the reads

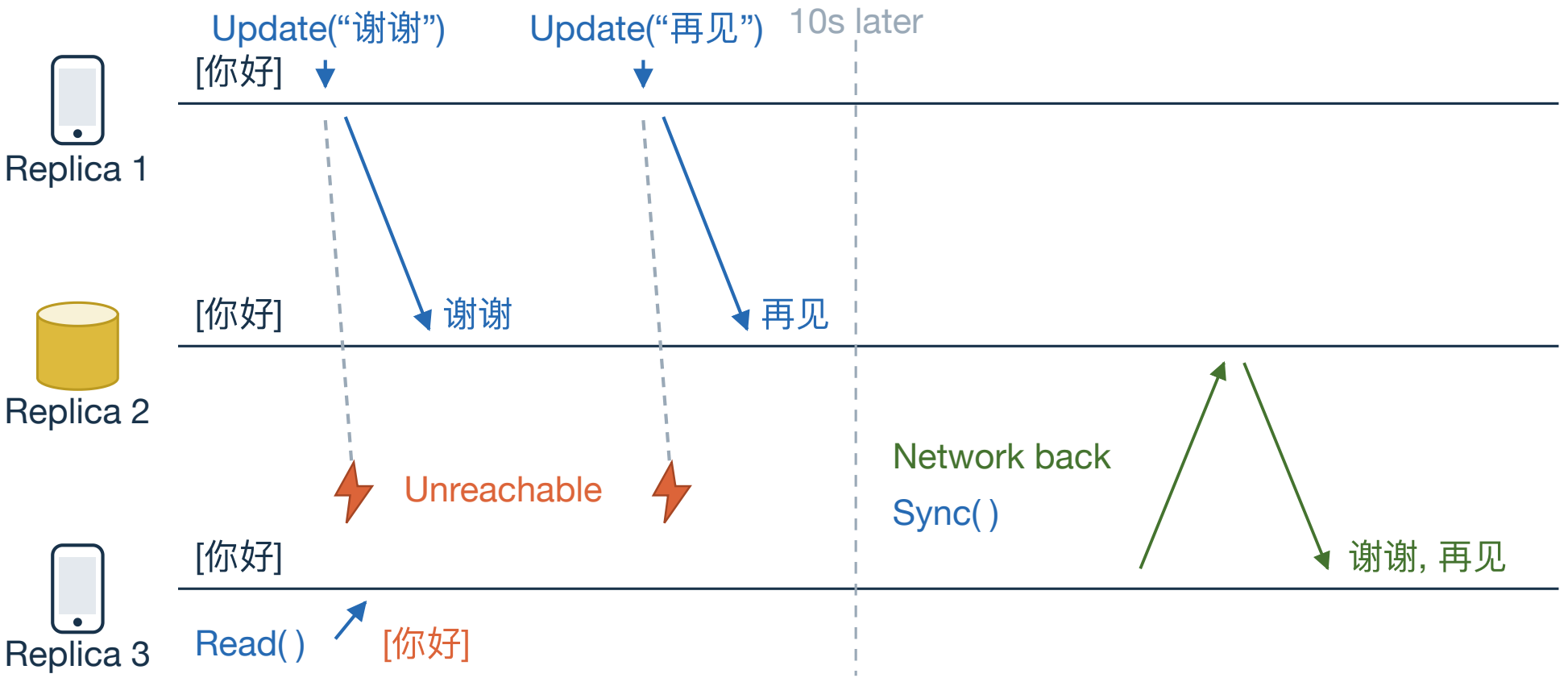
- E.g., a periodic heartbeat, say every 10 seconds
- Or via a service (e.g., Apple Push Notification), batching many syncs



Reading the local copy with periodic syncs

With periodic syncs, reading the local copy can recover most of the reads

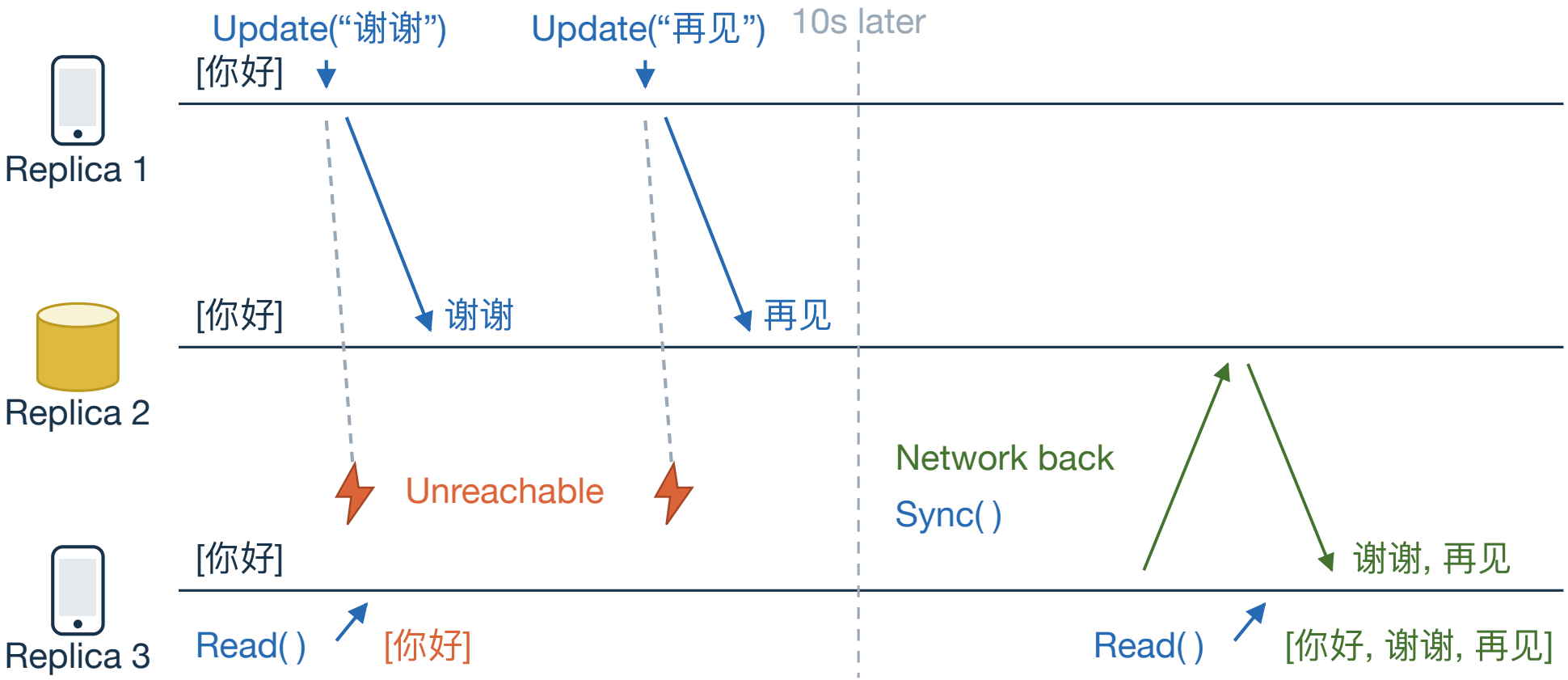
- E.g., a periodic heartbeat, say every 10 seconds
- Or via a service (e.g., Apple Push Notification), batching many syncs



Reading the local copy with periodic syncs

With periodic syncs, reading the local copy can recover most of the reads

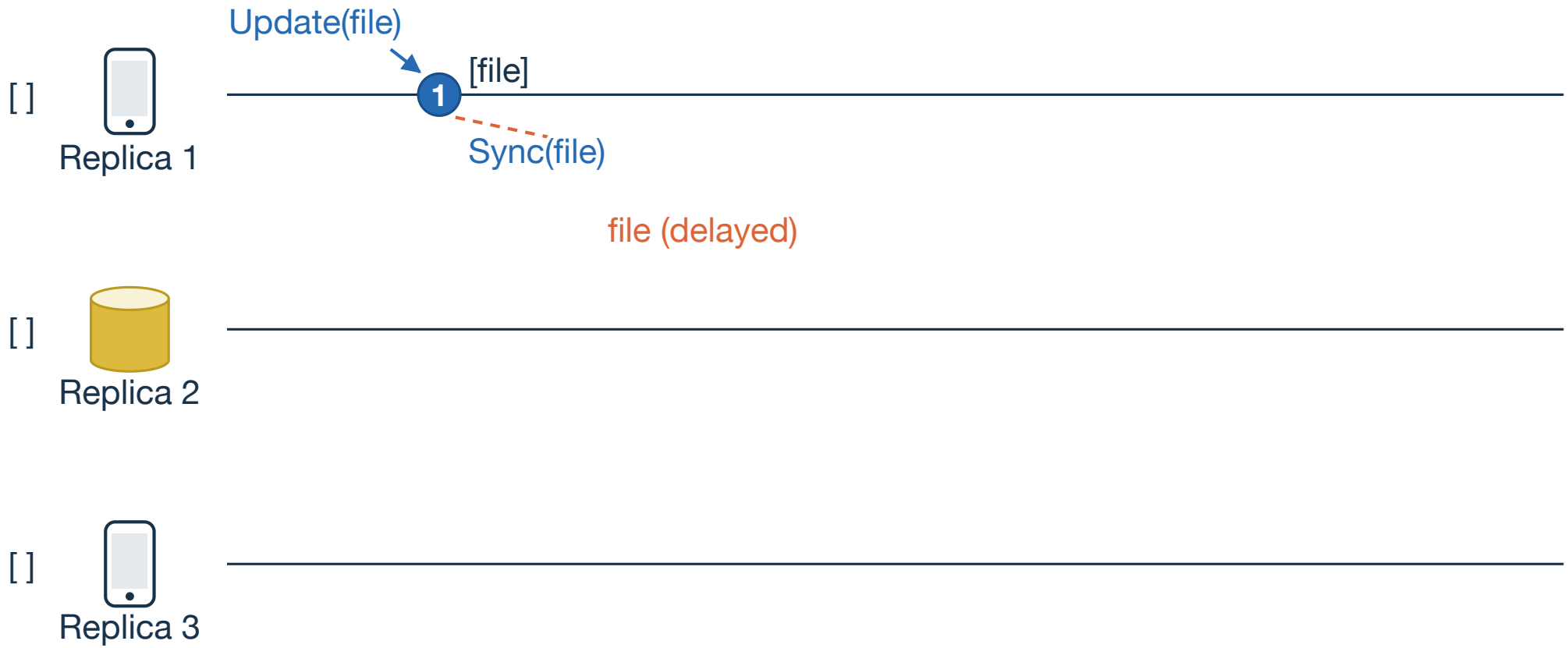
- E.g., a periodic heartbeat, say every 10 seconds
- Or via a service (e.g., Apple Push Notification), batching many syncs



Reordering is possible in a replicated implementation

Cause: a delayed sync carries no order with it

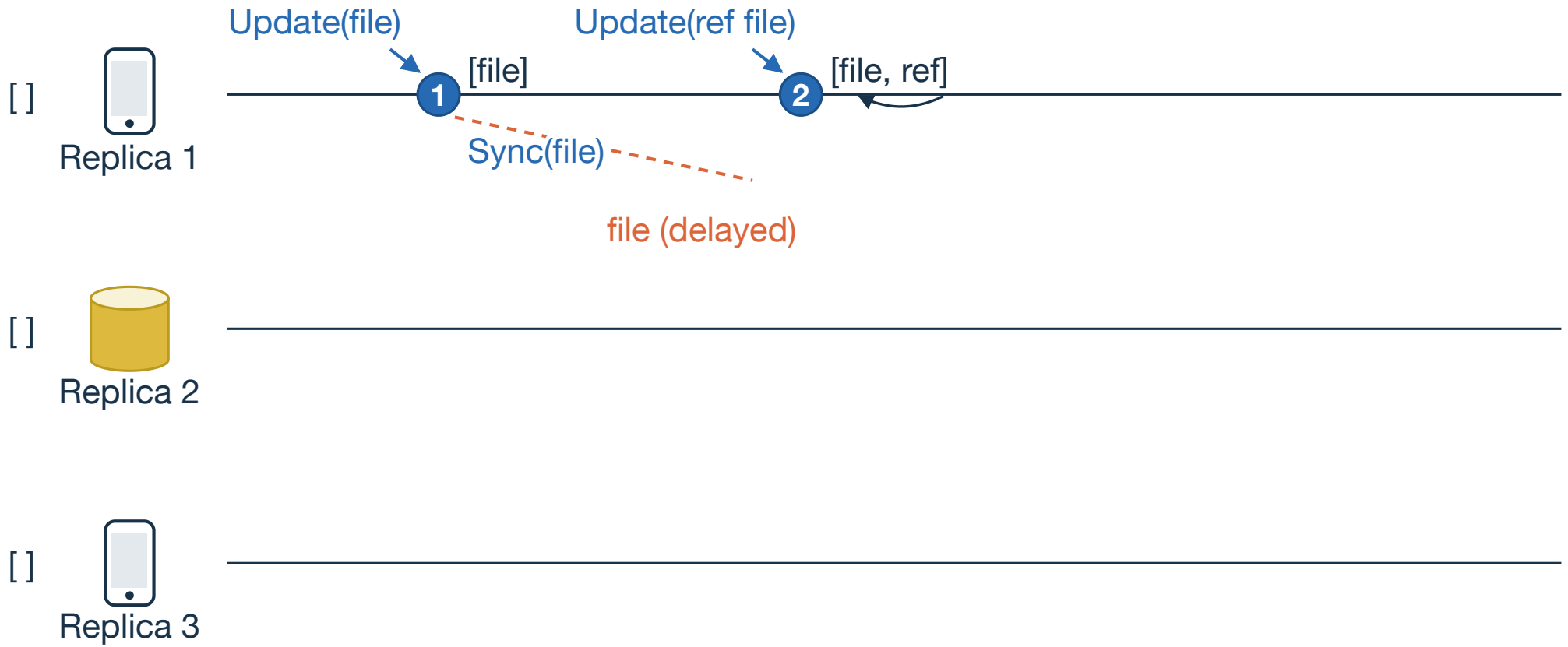
- ①② are the same two updates as in the motivating case, applied as they arrive
- ①'s sync starts first, but the file arrives after the update citing it



Reordering is possible in a replicated implementation

Cause: a delayed sync carries no order with it

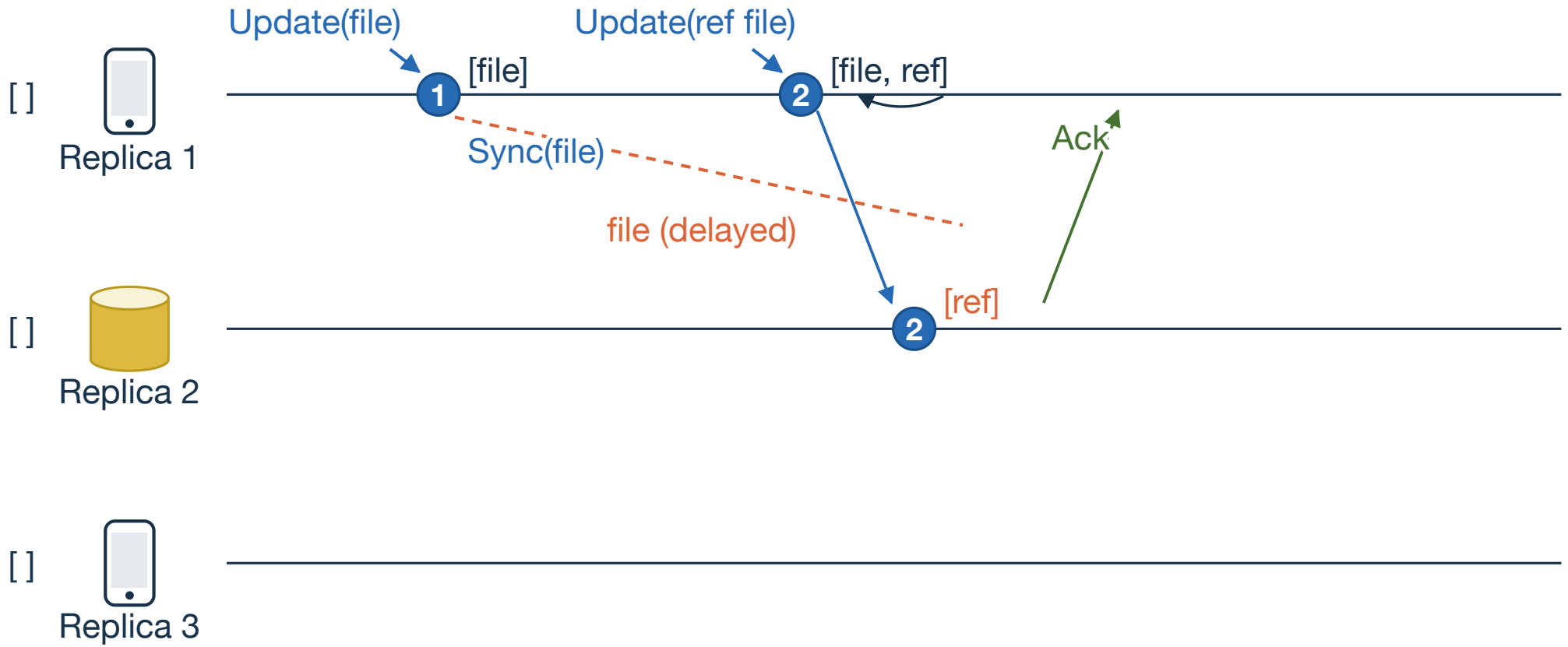
- ①② are the same two updates as in the motivating case, applied as they arrive
- ①'s sync starts first, but the file arrives after the update citing it



Reordering is possible in a replicated implementation

Cause: a delayed sync carries no order with it

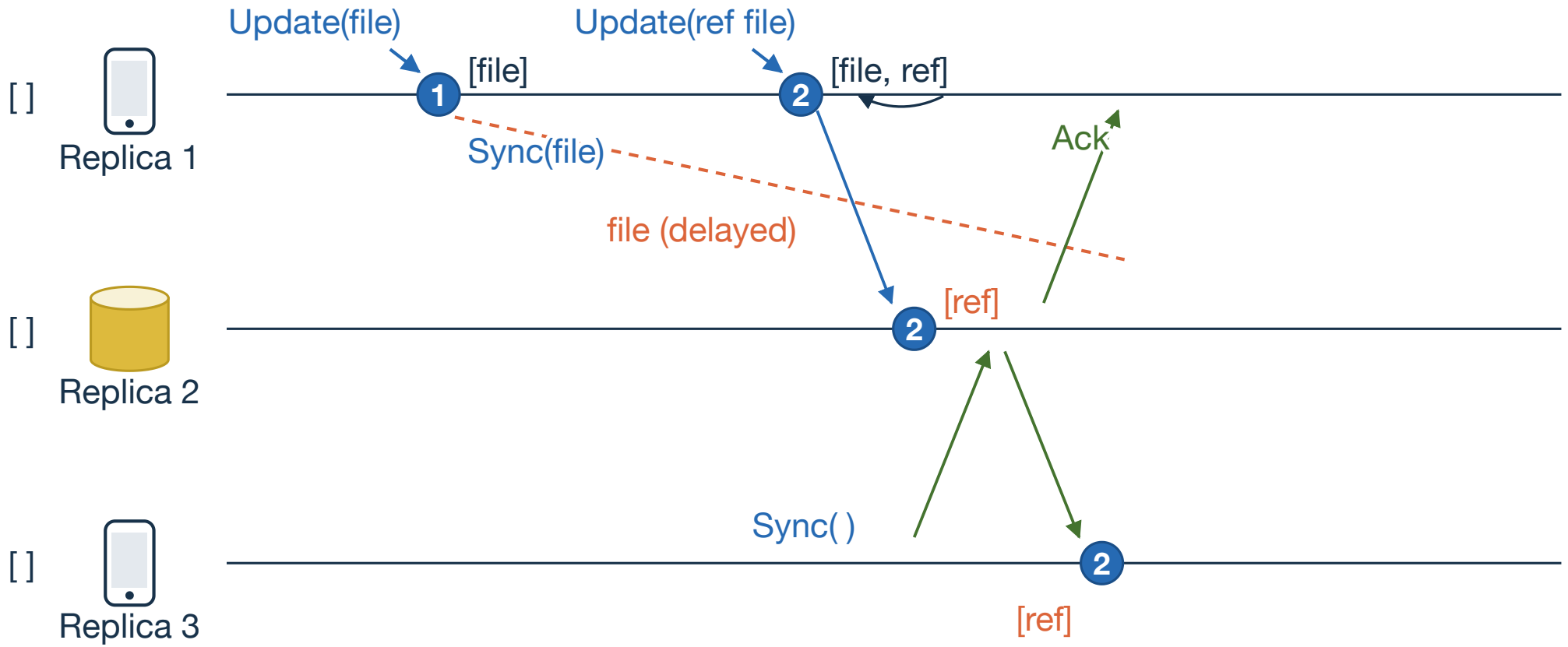
- ①② are the same two updates as in the motivating case, applied as they arrive
- ①'s sync starts first, but the file arrives after the update citing it



Reordering is possible in a replicated implementation

Cause: a delayed sync carries no order with it

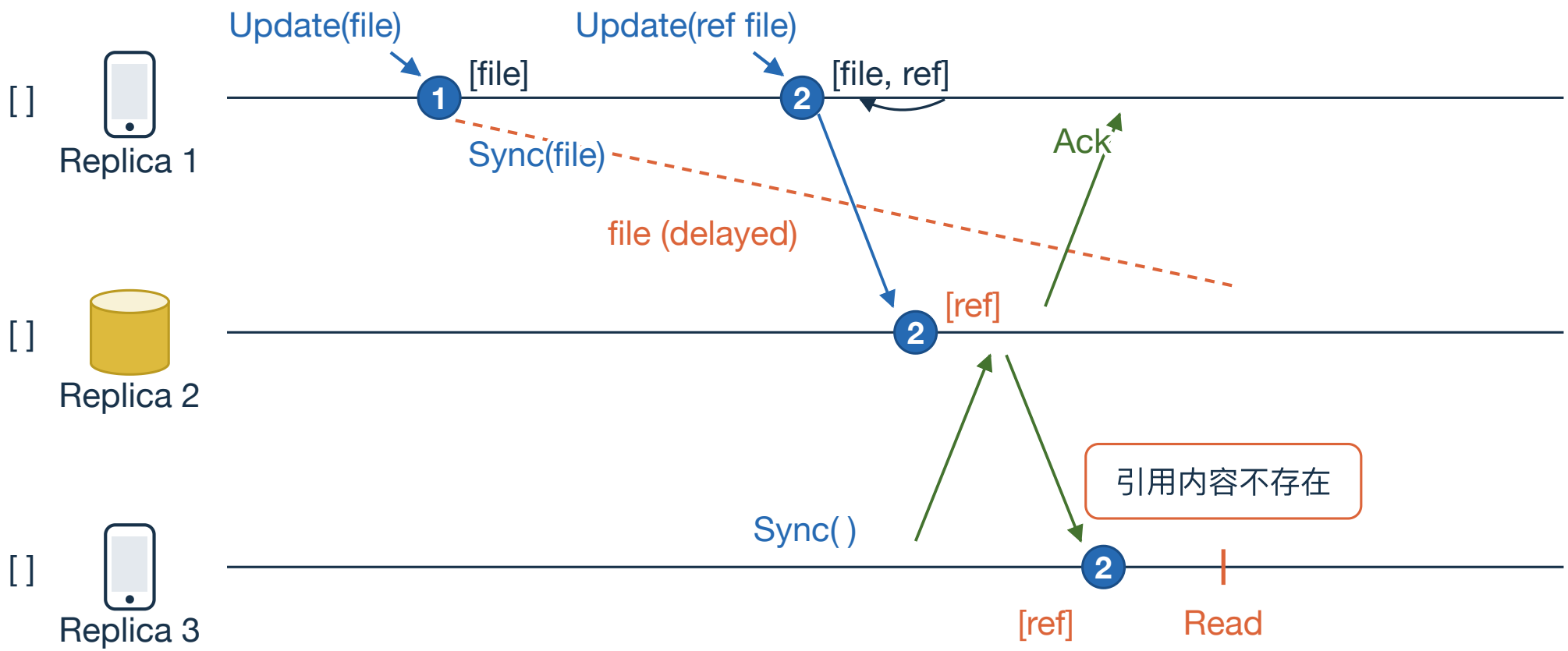
- ①② are the same two updates as in the motivating case, applied as they arrive
- ①'s sync starts first, but the file arrives after the update citing it



Reordering is possible in a replicated implementation

Cause: a delayed sync carries no order with it

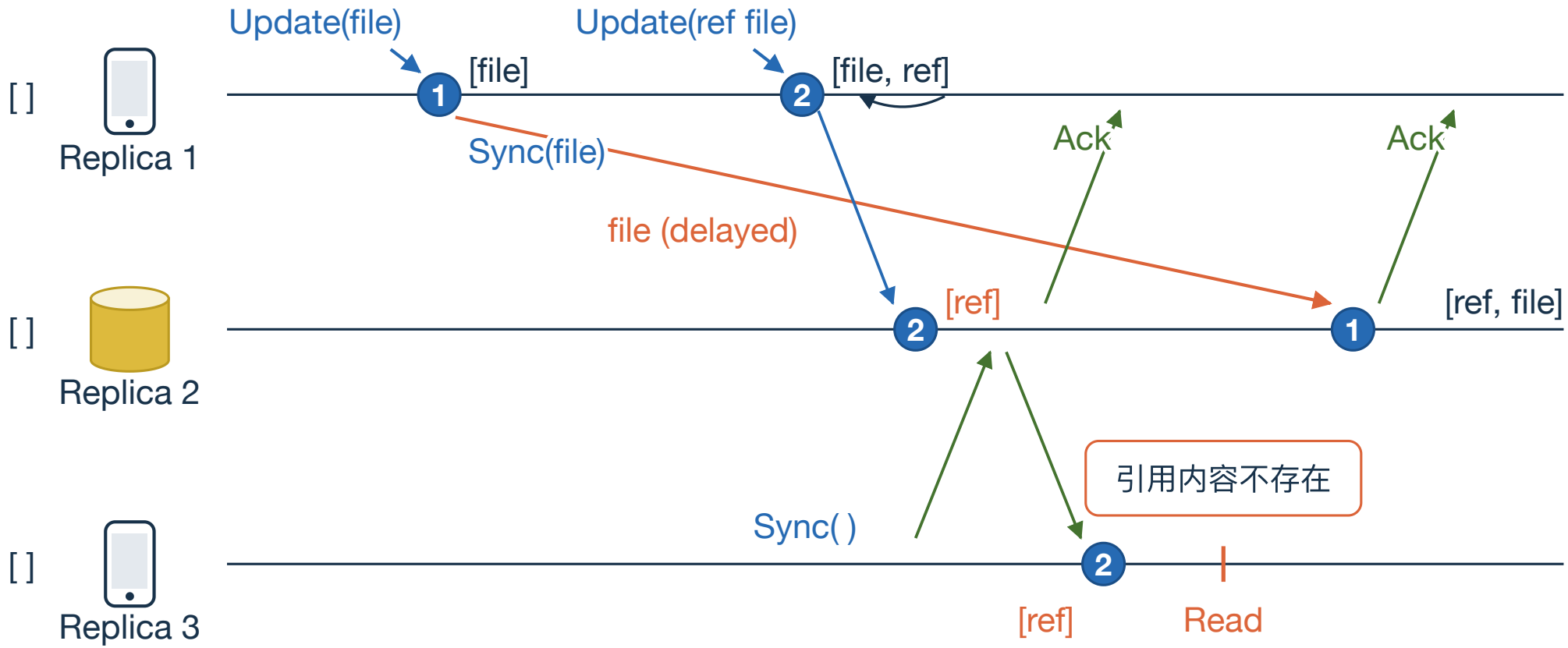
- ①② are the same two updates as in the motivating case, applied as they arrive
- ①'s sync starts first, but the file arrives after the update citing it



Reordering is possible in a replicated implementation

Cause: a delayed sync carries no order with it

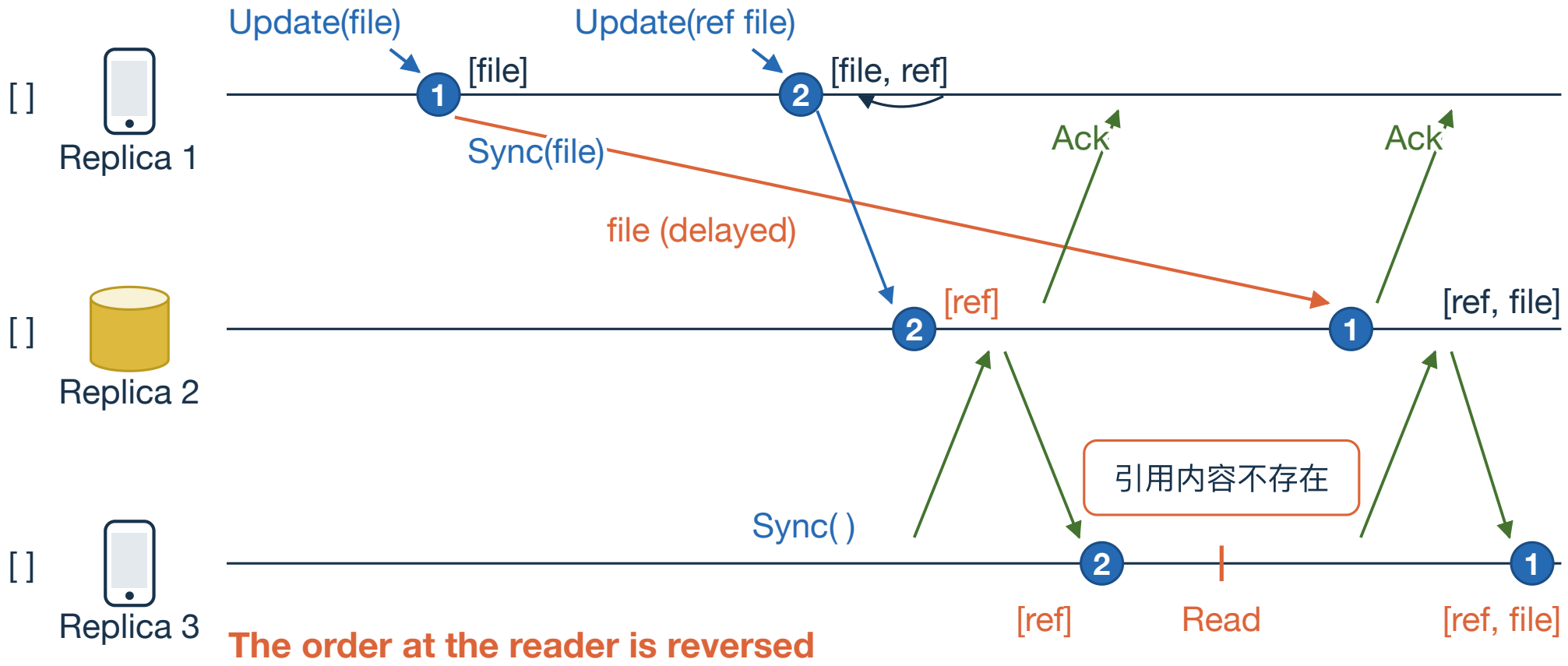
- ①② are the same two updates as in the motivating case, applied as they arrive
- ①'s sync starts first, but the file arrives after the update citing it



Reordering is possible in a replicated implementation

Cause: a delayed sync carries no order with it

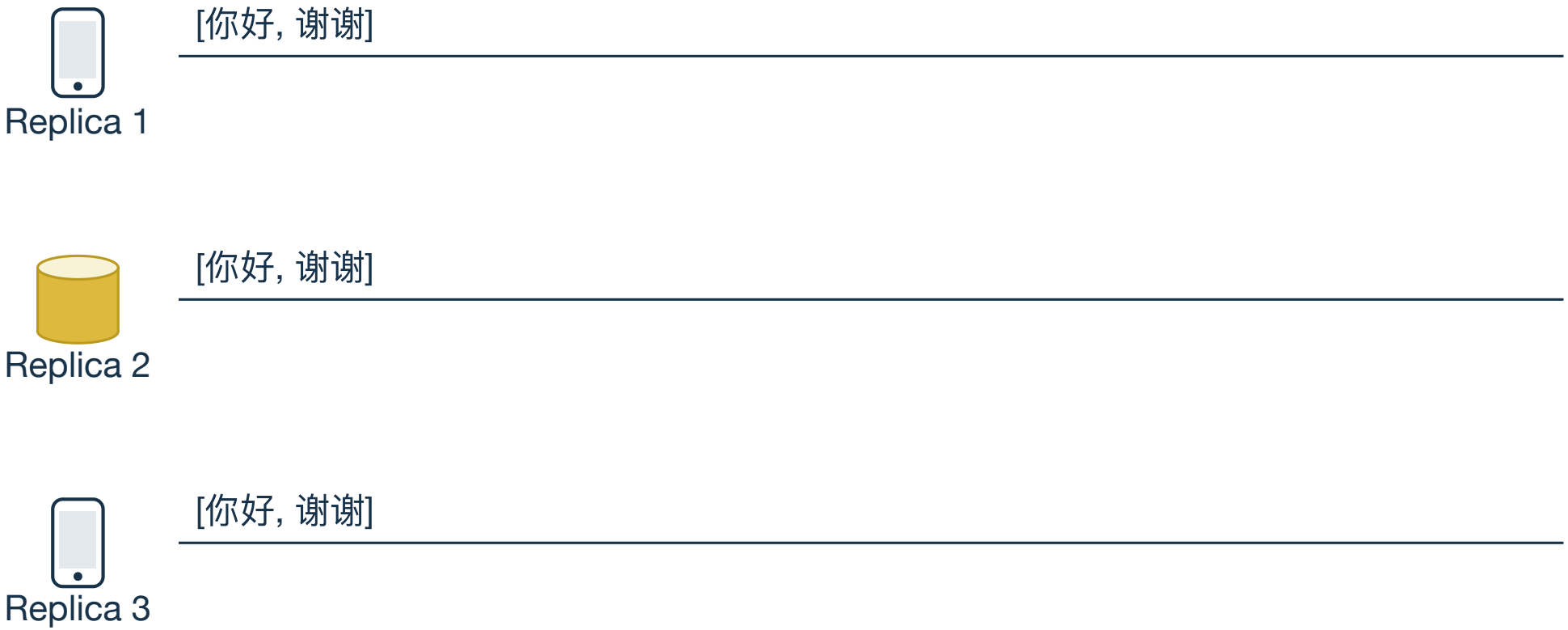
- ①② are the same two updates as in the motivating case, applied as they arrive
- ①'s sync starts first, but the file arrives after the update citing it



Another issue: update conflicts

A local update returns before it is synced, so two replicas can modify the same list

- By the time the sync runs, both updates extend [你好, 谢谢] — neither copy is stale
- The sync must resolve the conflict: merge the operations



Another issue: update conflicts

A local update returns before it is synced, so two replicas can modify the same list

- By the time the sync runs, both updates extend [你好, 谢谢] — neither copy is stale
- The sync must resolve the conflict: merge the operations



Another issue: update conflicts

A local update returns before it is synced, so two replicas can modify the same list

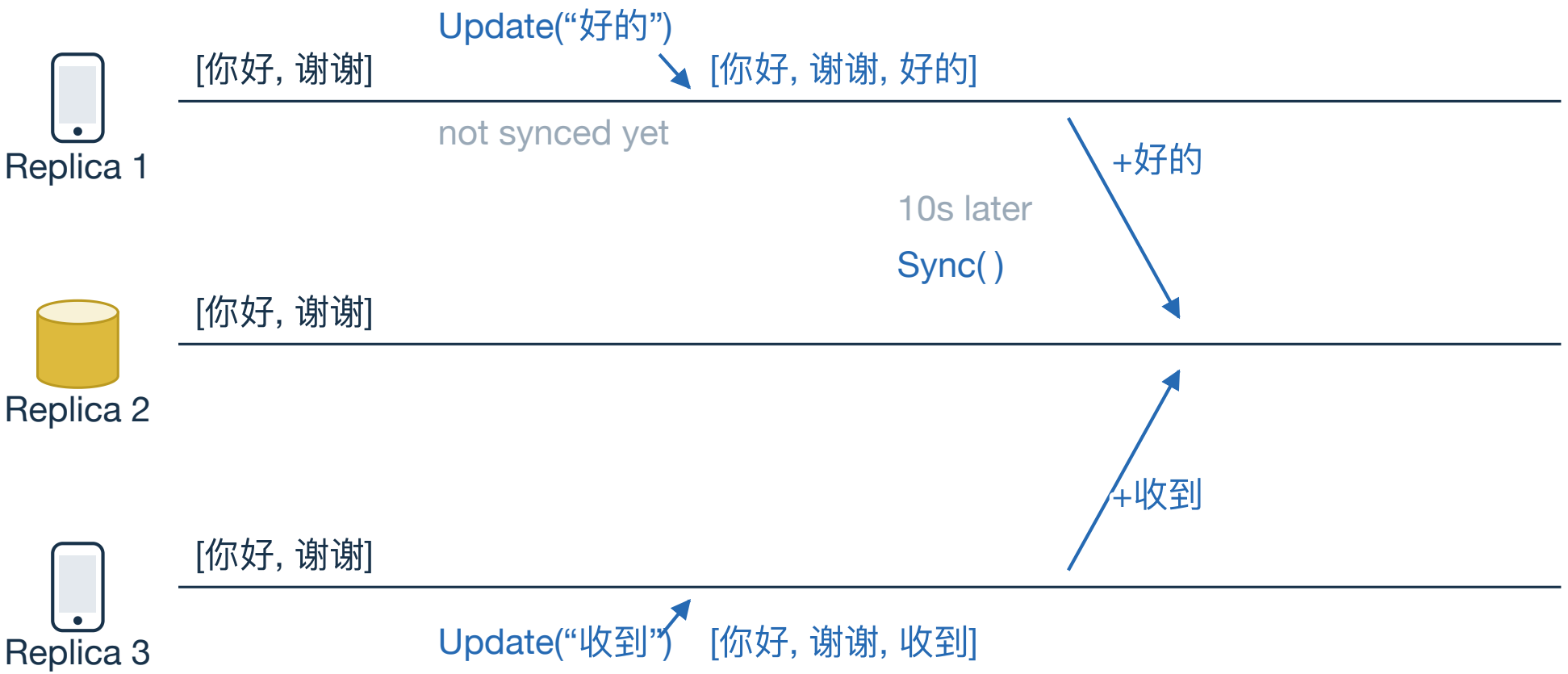
- By the time the sync runs, both updates extend [你好, 谢谢] — neither copy is stale
- The sync must resolve the conflict: merge the operations



Another issue: update conflicts

A local update returns before it is synced, so two replicas can modify the same list

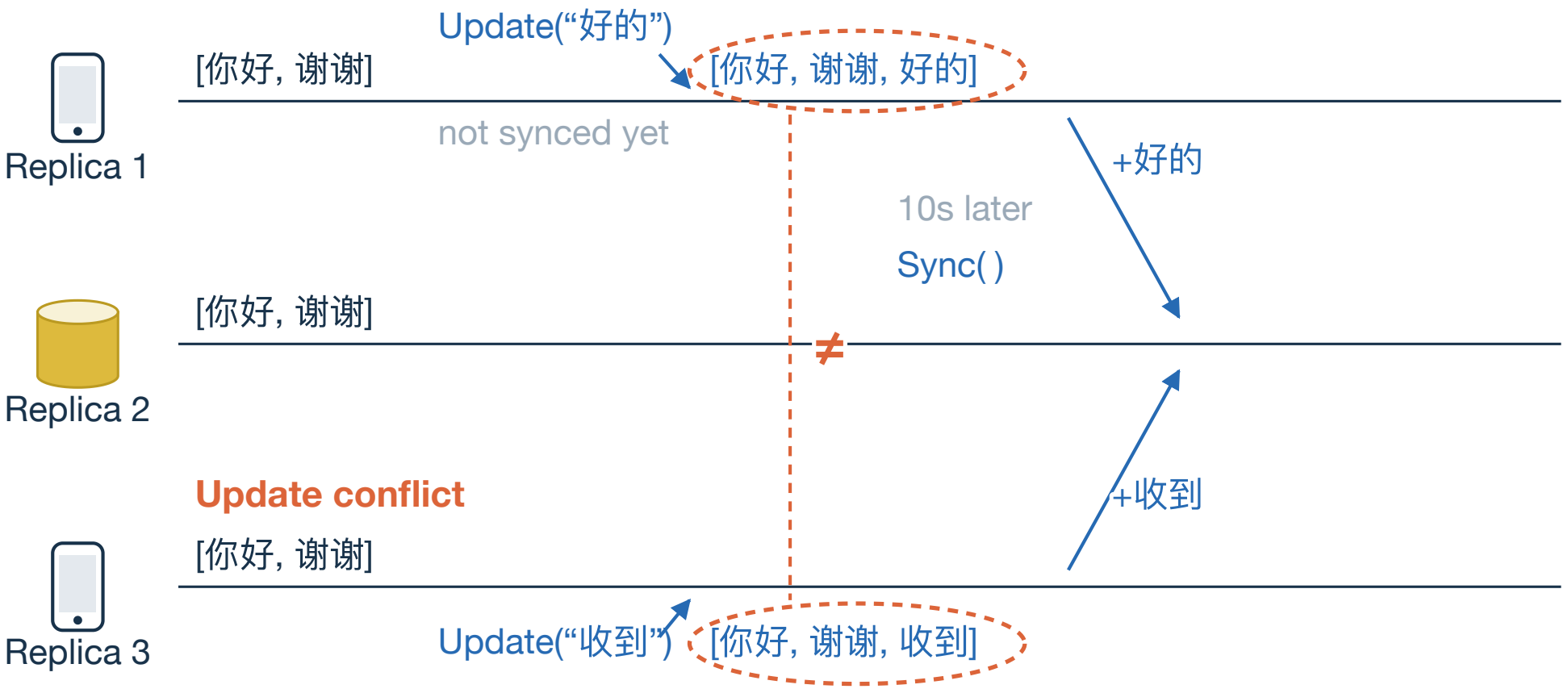
- By the time the sync runs, both updates extend [你好, 谢谢] — neither copy is stale
- The sync must resolve the conflict: merge the operations



Another issue: update conflicts

A local update returns before it is synced, so two replicas can modify the same list

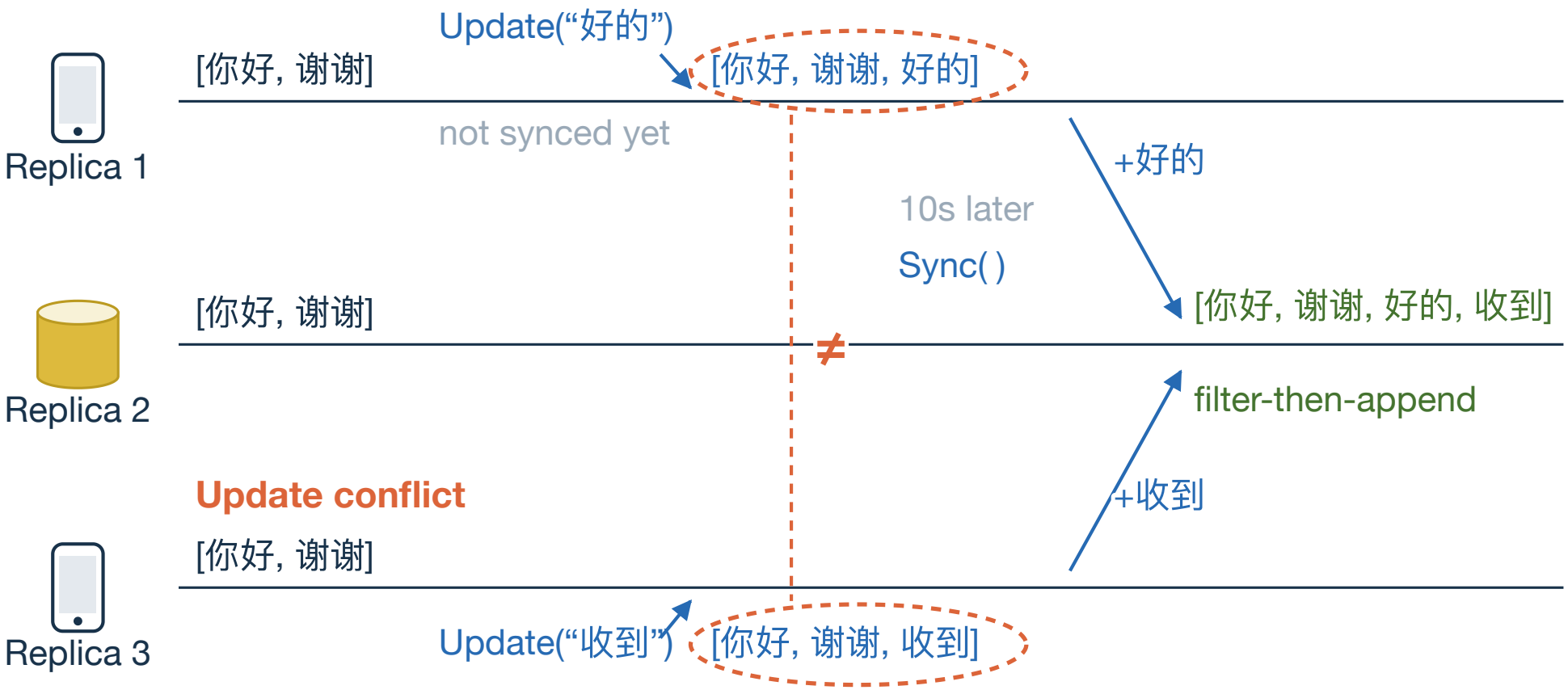
- By the time the sync runs, both updates extend [你好, 谢谢] — neither copy is stale
- The sync must resolve the conflict: merge the operations



Another issue: update conflicts

A local update returns before it is synced, so two replicas can modify the same list

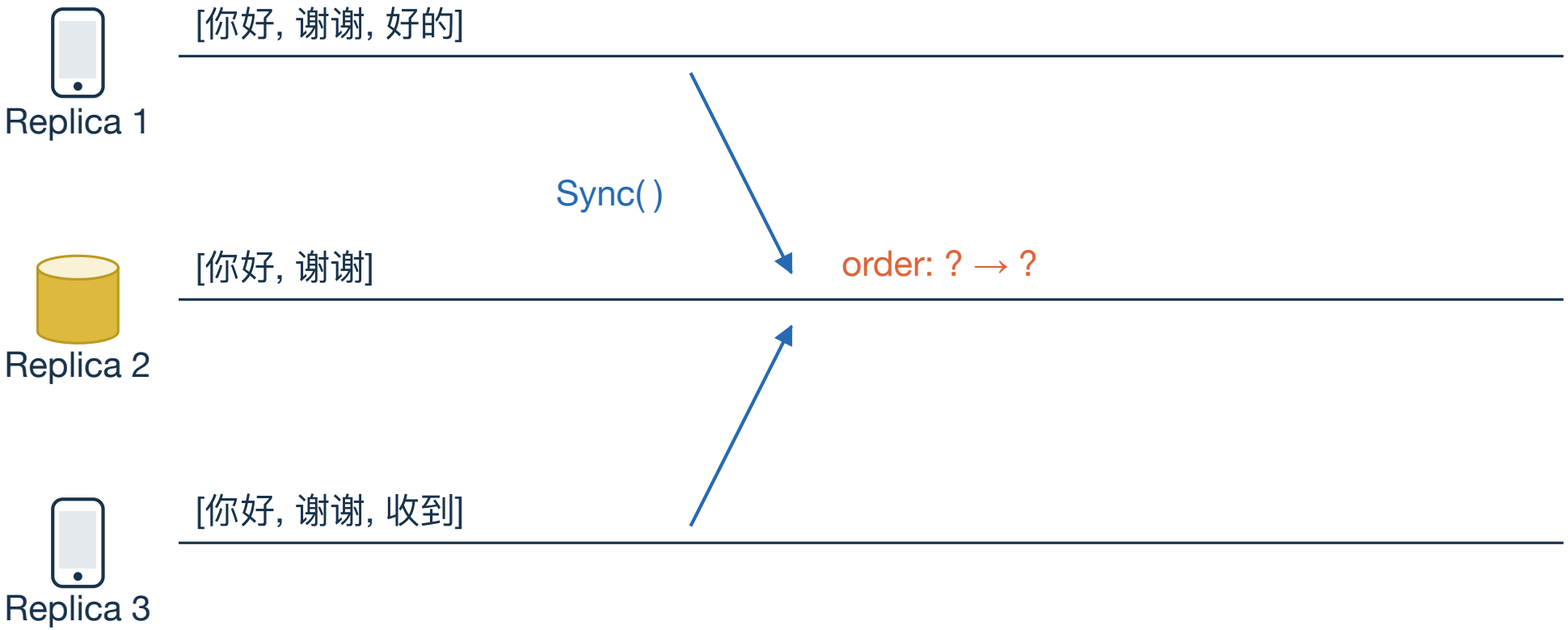
- By the time the sync runs, both updates extend [你好, 谢谢] — neither copy is stale
- The sync must resolve the conflict: merge the operations



Challenge of the merge: whose order do we keep?

Whichever order the merge picks, one of the users saw an inconsistent order

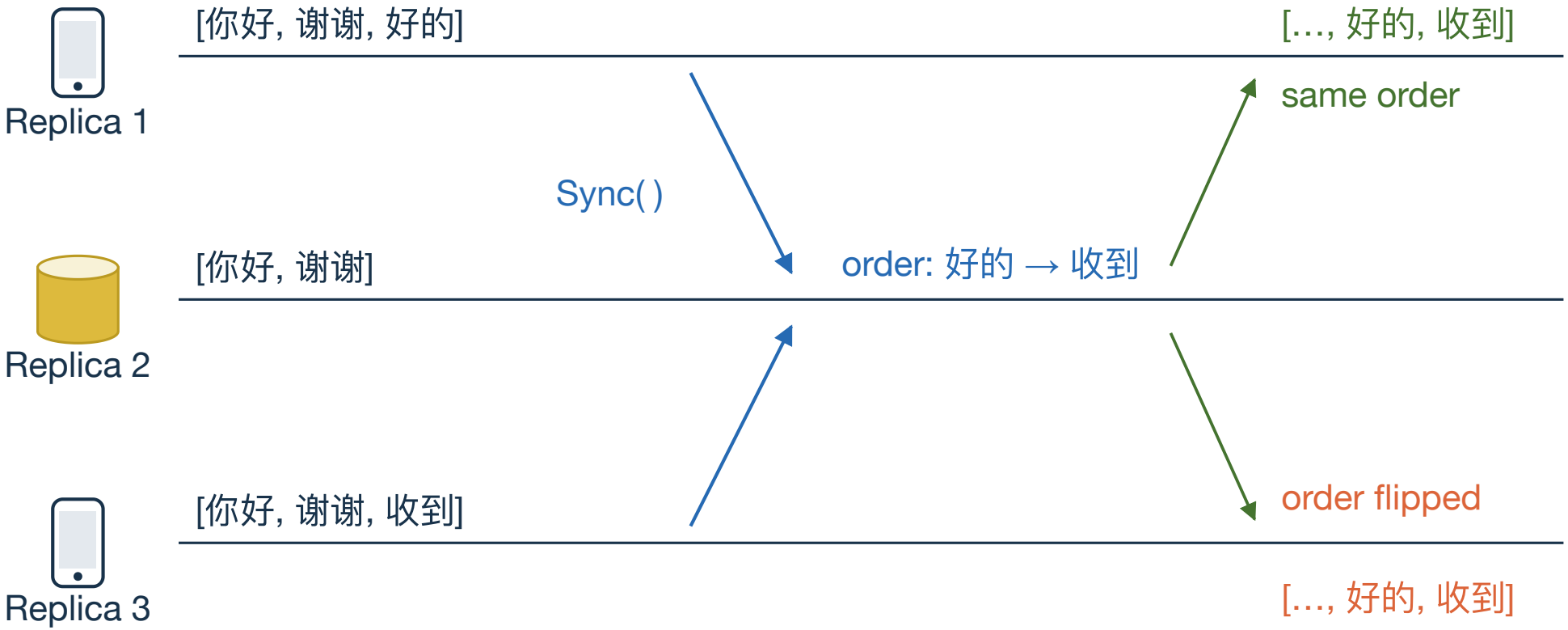
- 好的 first: replica 3 had already shown 收到 above 好的
- 收到 first: replica 1 had already shown 好的 above 收到



Challenge of the merge: whose order do we keep?

Whichever order the merge picks, one of the users saw an inconsistent order

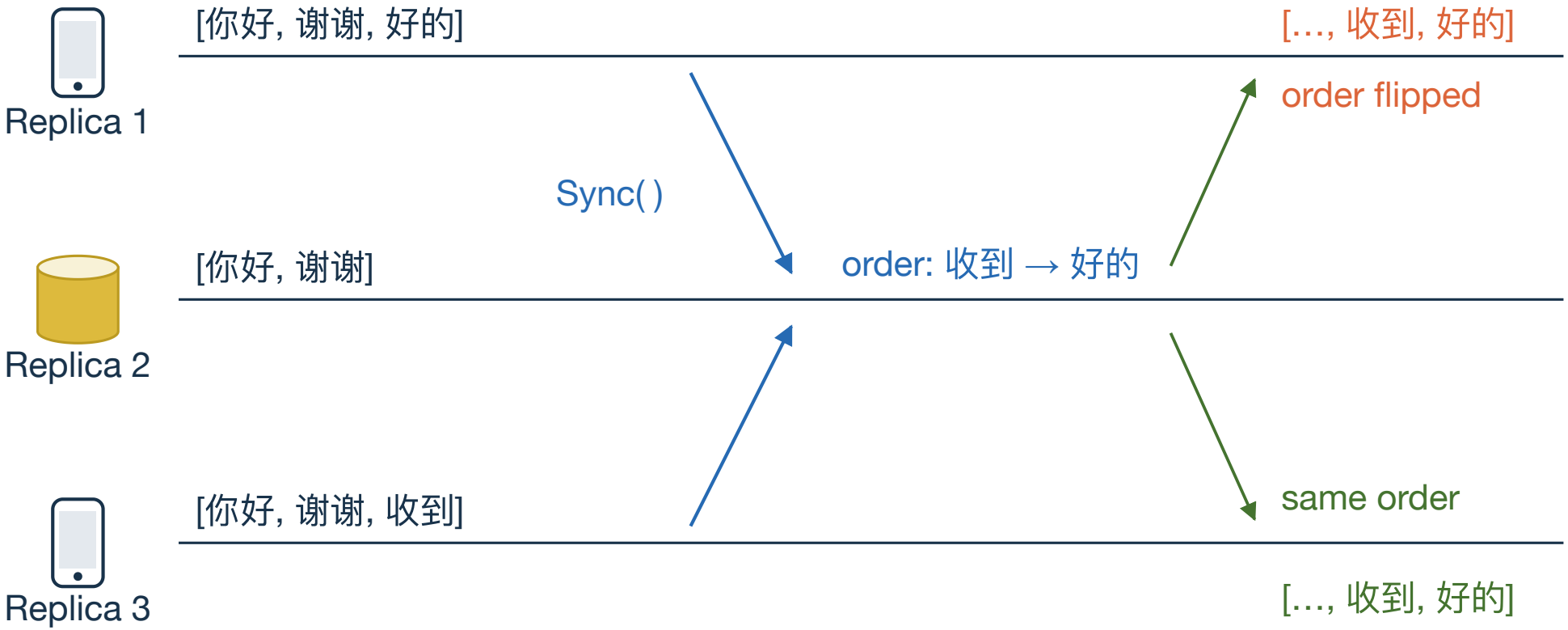
- 好的 first: replica 3 had already shown 收到 above 好的
- 收到 first: replica 1 had already shown 好的 above 收到



Challenge of the merge: whose order do we keep?

Whichever order the merge picks, one of the users saw an inconsistent order

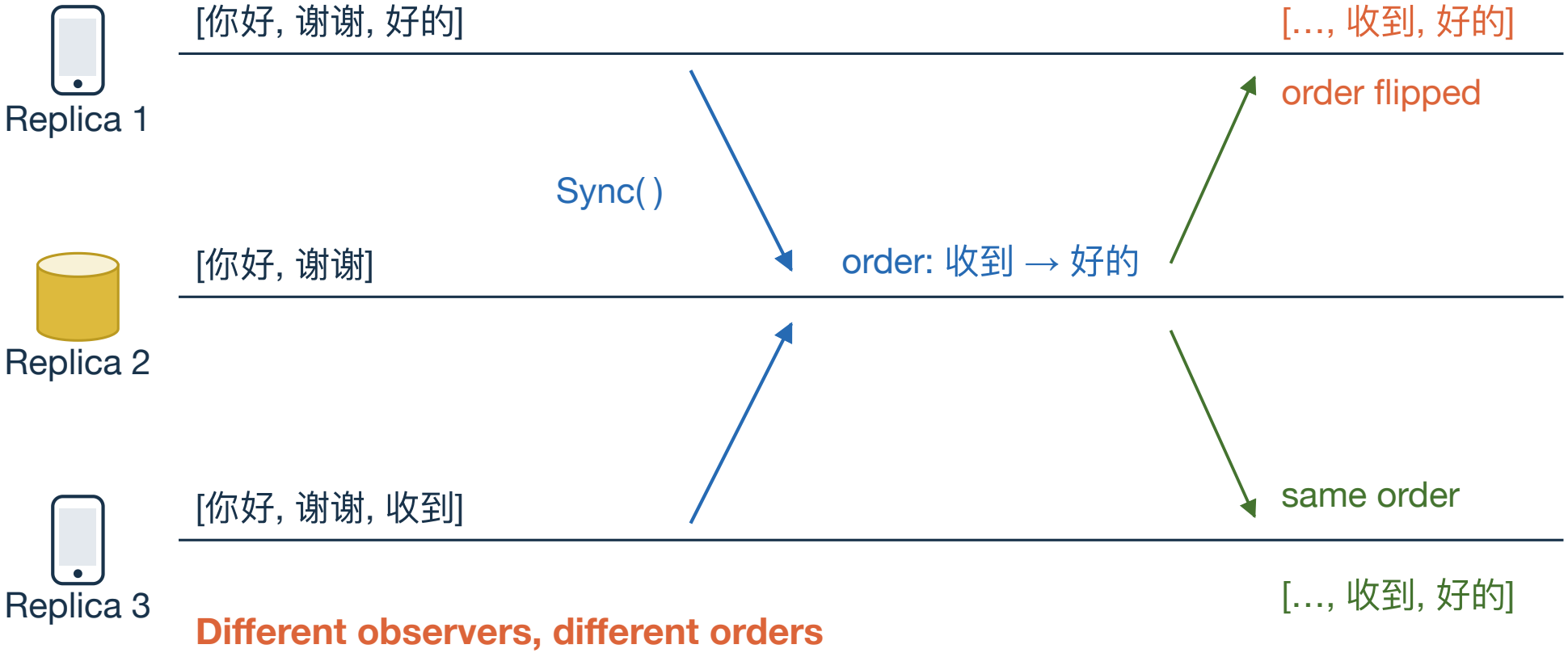
- 好的 first: replica 3 had already shown 收到 above 好的
- 收到 first: replica 1 had already shown 好的 above 收到



Challenge of the merge: whose order do we keep?

Whichever order the merge picks, one of the users saw an inconsistent order

- 好的 first: replica 3 had already shown 收到 above 好的
- 收到 first: replica 1 had already shown 好的 above 收到



We need a systematic method to hide the complexity of understanding and coping with inconsistency under a distributed (replicated) setup.

How to cope with the complexity of distributed read/write sync?

Application developers typically assume an implementation like centralized setup

- The simplest form, no inconsistency, but is slow and unreliable
- The consistency notion can be defined around it!

Ideal goal: consistency like a centralized setups, high performance, and reliable

- But typically we cannot achieve all, so we must make trade-offs

Different trade-offs result in different consistency models

- Weaker consistency model may be slower than stronger ones

Note that a strong model does not mean a centralized implementation

- It can be much more performant and reliable

Consistency model



Todd Lipcon： Consistency model defines rules for the apparent order and visibility of updates (by different observers), and it is a continuum with tradeoffs.

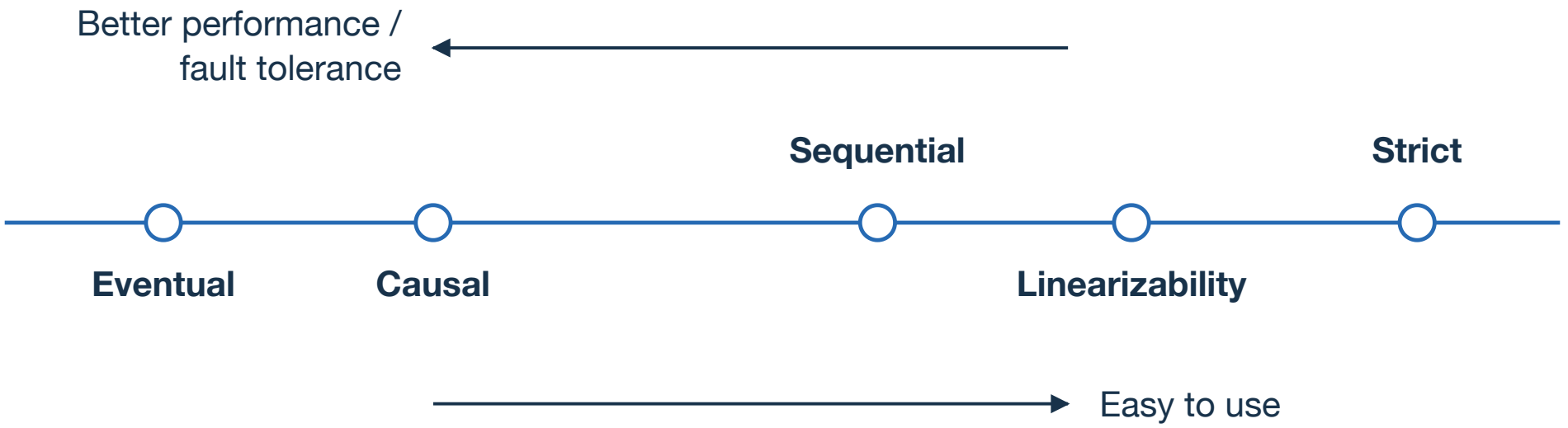
There are no right or wrong consistency models

- Only tradeoffs between ease of programmability and performance/reliability

Note that consistency can arise in other system designs

- Data replication
- Caching (also a form of replication)
- Concurrency (multi-core and multi-host)
- Failures (e.g., single-machine and multi-machine)

Spectrum of consistency models



There are other models exist (e.g., close-open consistency)

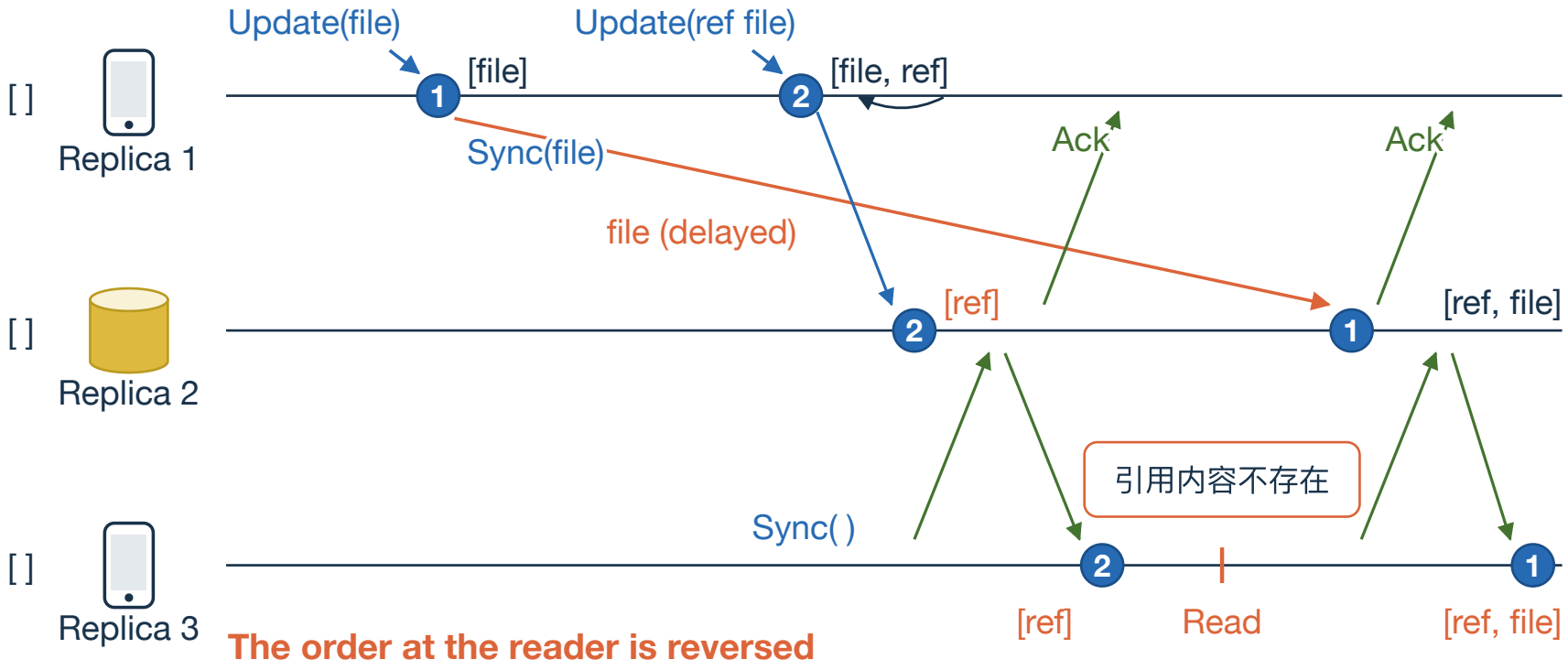
What is the desired model of a system?
And is it possible to implement?

Back to our example (ChatAPP)

What is the desired model of our replicated list?

The model depends on the application requirements (or system goals)

- e.g., if it can tolerate some inconsistency, we can really make the system fast and reliable, as we have shown before



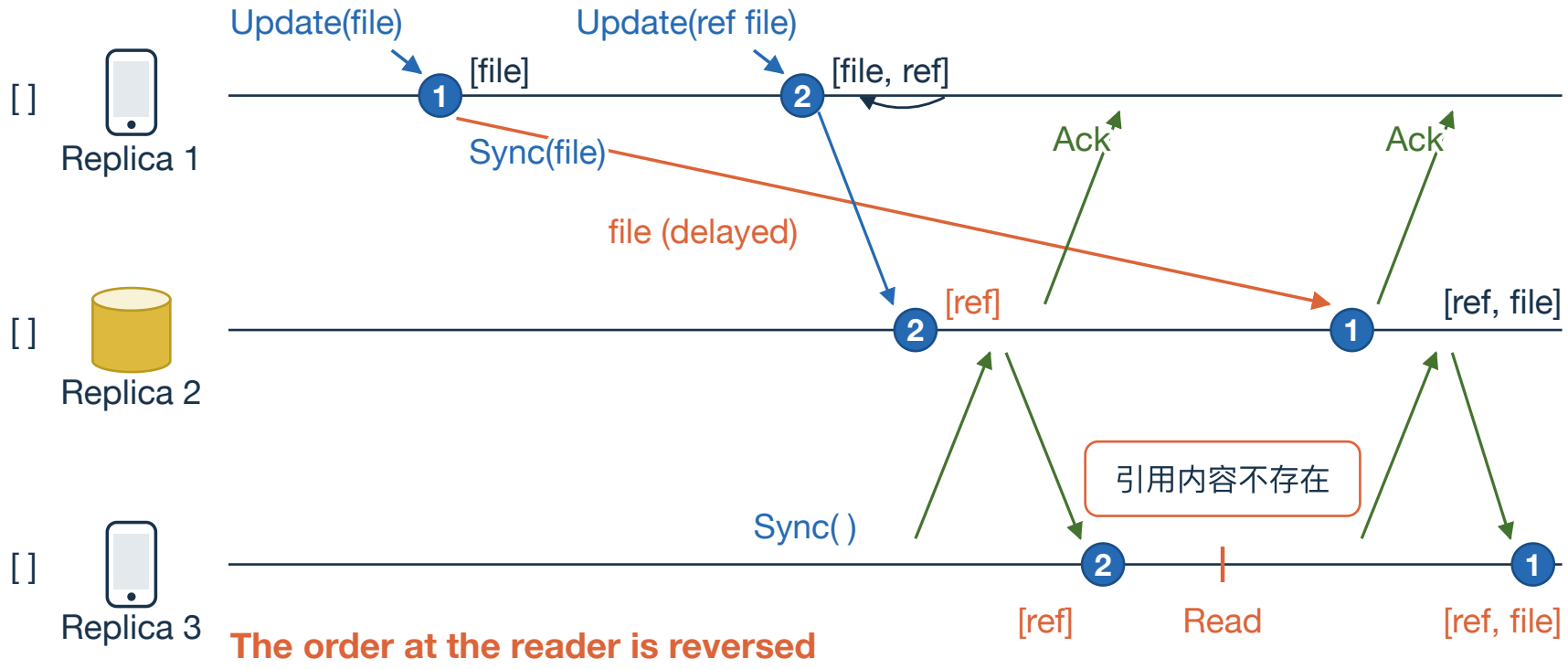
What are the goals of a chat application

Goal #1: always on

- If my phone is connected to the network, we should be able to read the message

Goal #2: read consistency, i.e., the read eventually returns a correct result

- i.e., all my devices will eventually have the same data



Eventual consistency

A model designed for prioritizing performance & fault tolerance

Informally

- All replicas eventually receive all writes, and servers holding the same set of writes will have the same state (~= same order of write operations)
- Thus, if no new updates are made to the data, eventually all accesses will return the last updated value

No restrictions on read/write, so we can freely choose an "always-on" solution

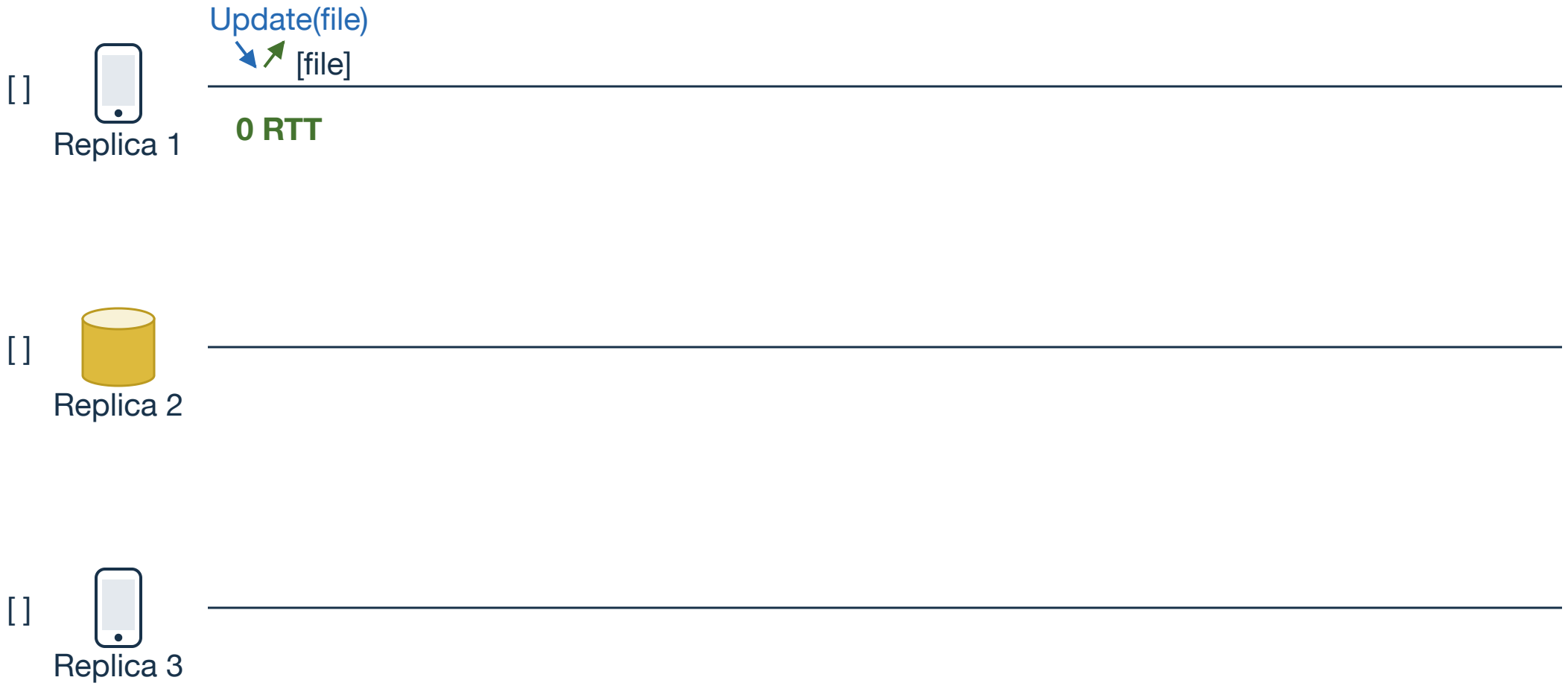
- Read: return the latest local copy of the data
- Write: write locally and return directly, then propagate the write to all the servers in the background — which is optimal

As long as we can sync the replicas to reach the same state!

Implementing an eventually consistent system

Write locally, then sync the write to others (e.g., at-least-once RPC)

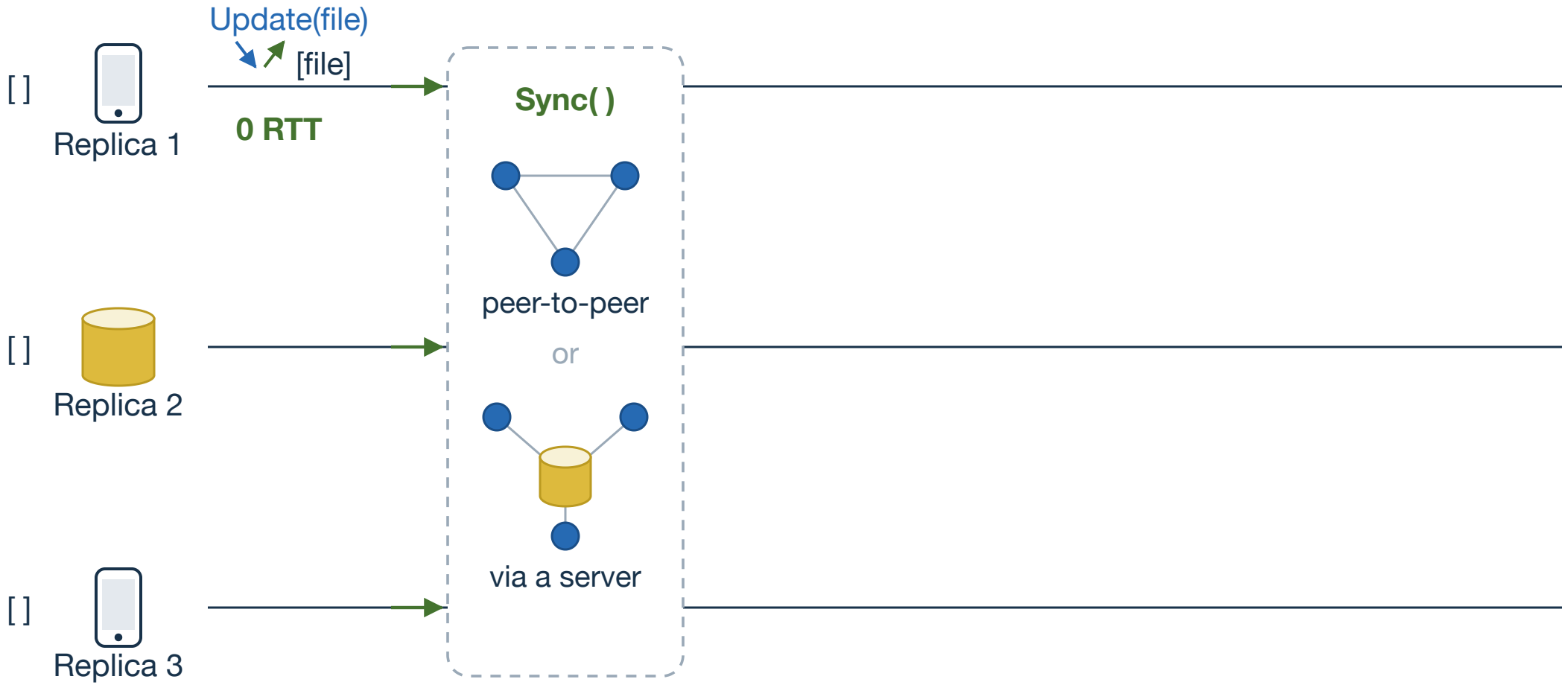
- Note that there are many ways for this implementation



Implementing an eventually consistent system

Write locally, then sync the write to others (e.g., at-least-once RPC)

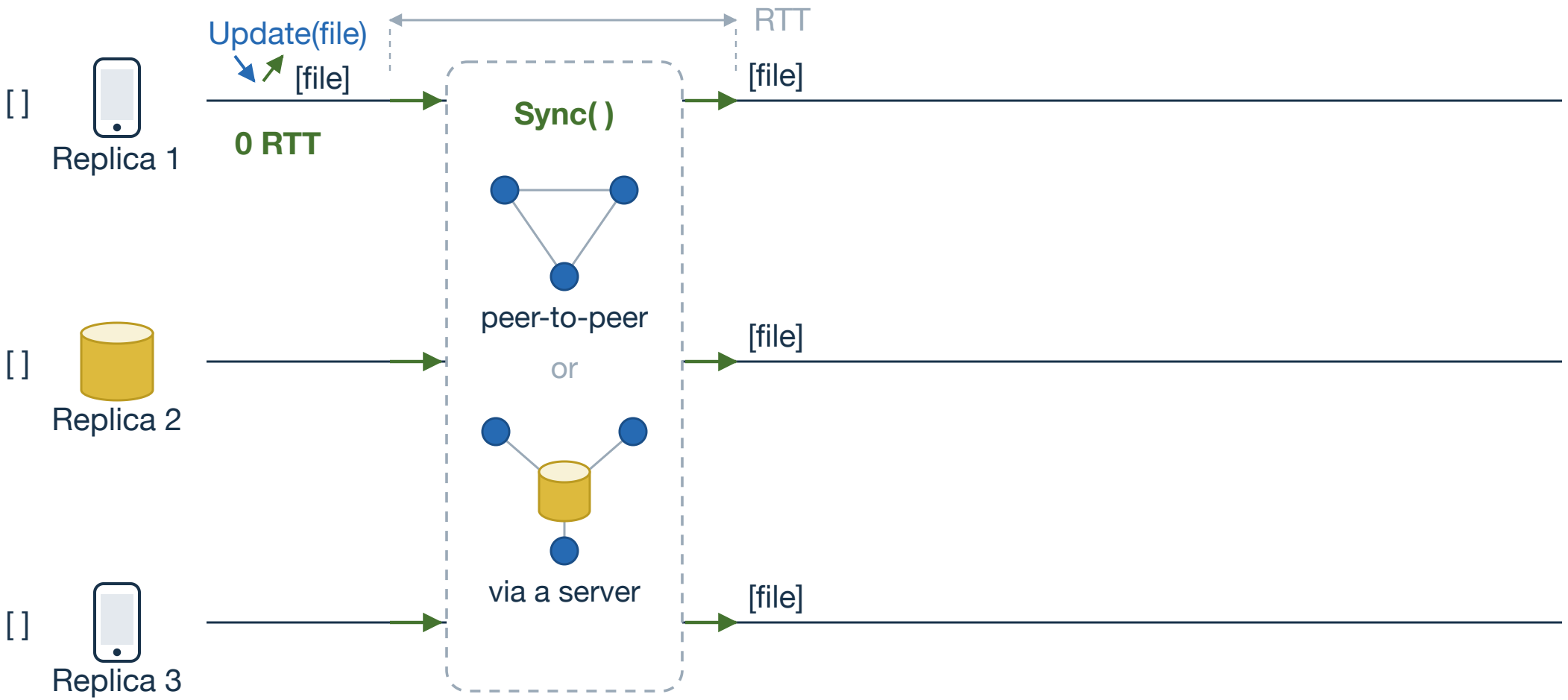
- Note that there are many ways for this implementation



Implementing an eventually consistent system

Write locally, then sync the write to others (e.g., at-least-once RPC)

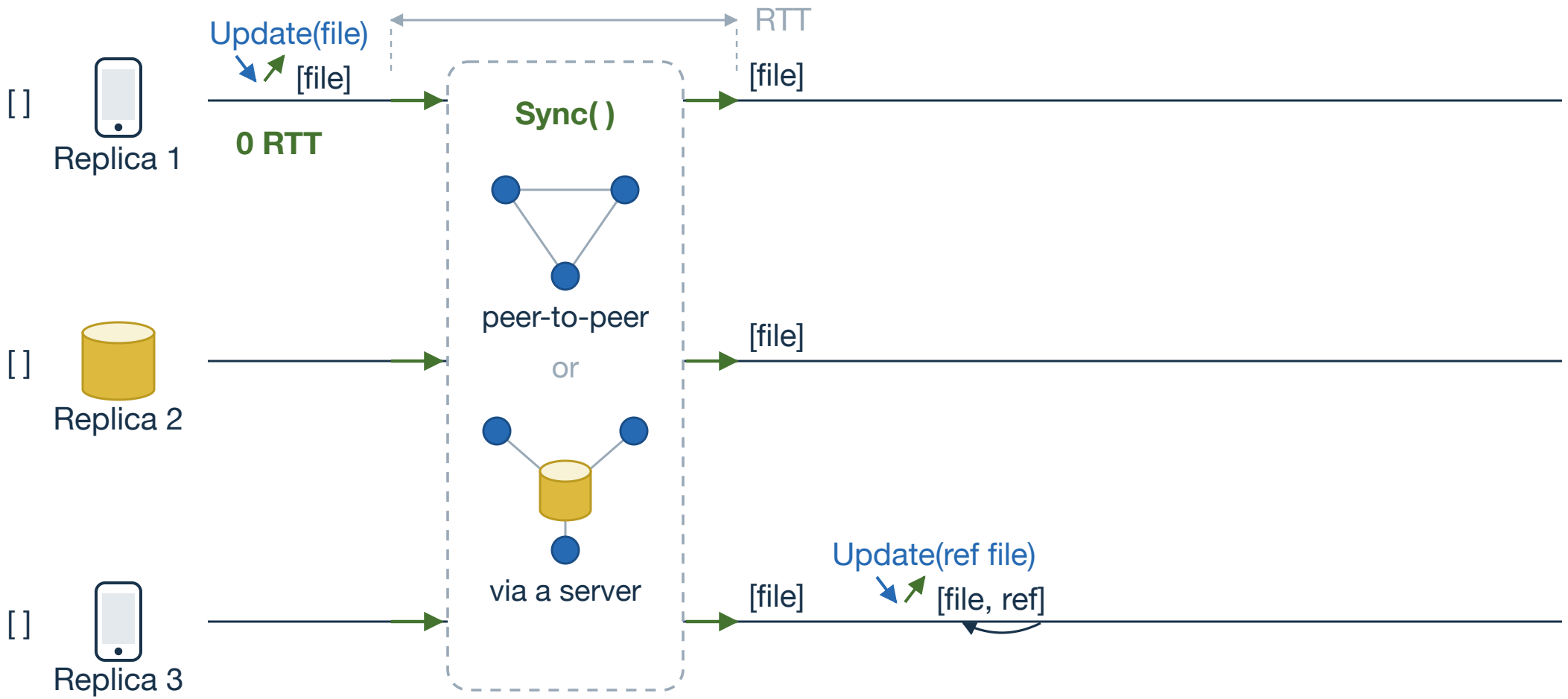
- Note that there are many ways for this implementation



Implementing an eventually consistent system

Write locally, then sync the write to others (e.g., at-least-once RPC)

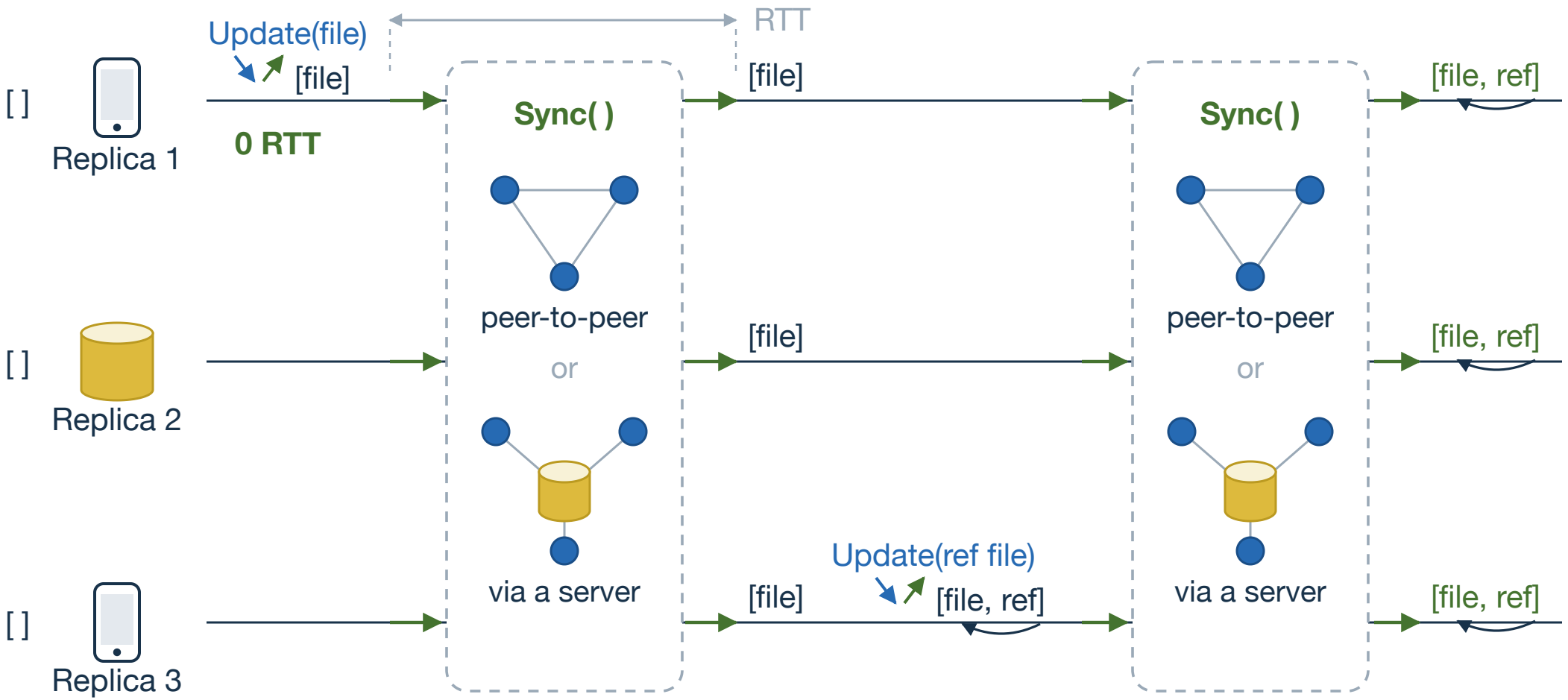
- Note that there are many ways for this implementation



Implementing an eventually consistent system

Write locally, then sync the write to others (e.g., at-least-once RPC)

- Note that there are many ways for this implementation

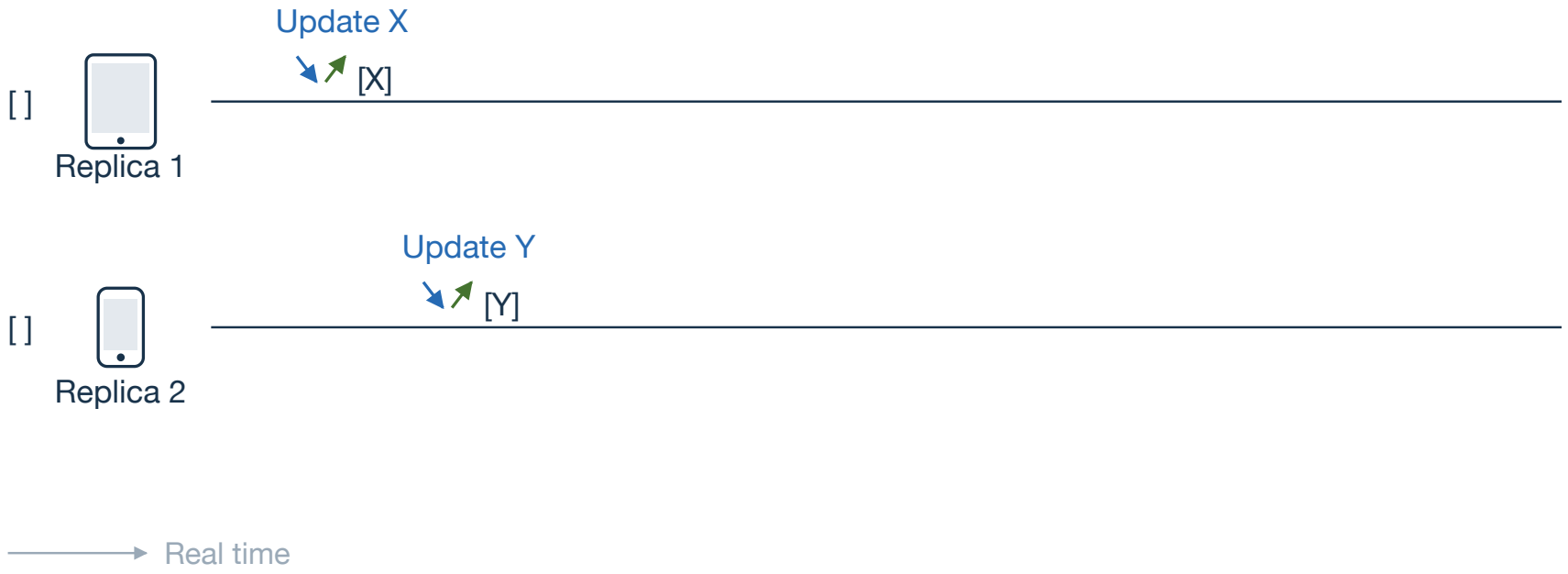


Syncing: exchange the updates received so far

Deciding the order:

- Will Replica 1 show X in front of Y, and Replica 2 show Y in front of X?

Challenge: how to decide the ordering?

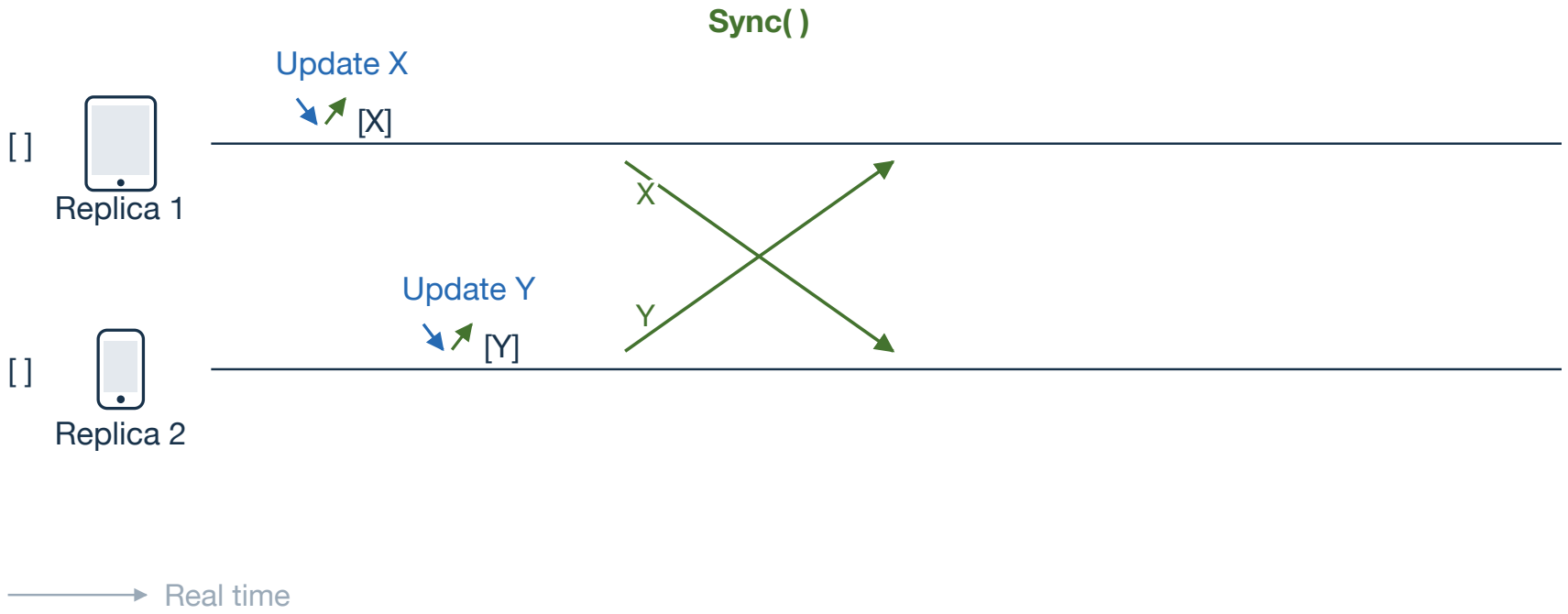


Syncing: exchange the updates received so far

Deciding the order:

- Will Replica 1 show X in front of Y, and Replica 2 show Y in front of X?

Challenge: how to decide the ordering?

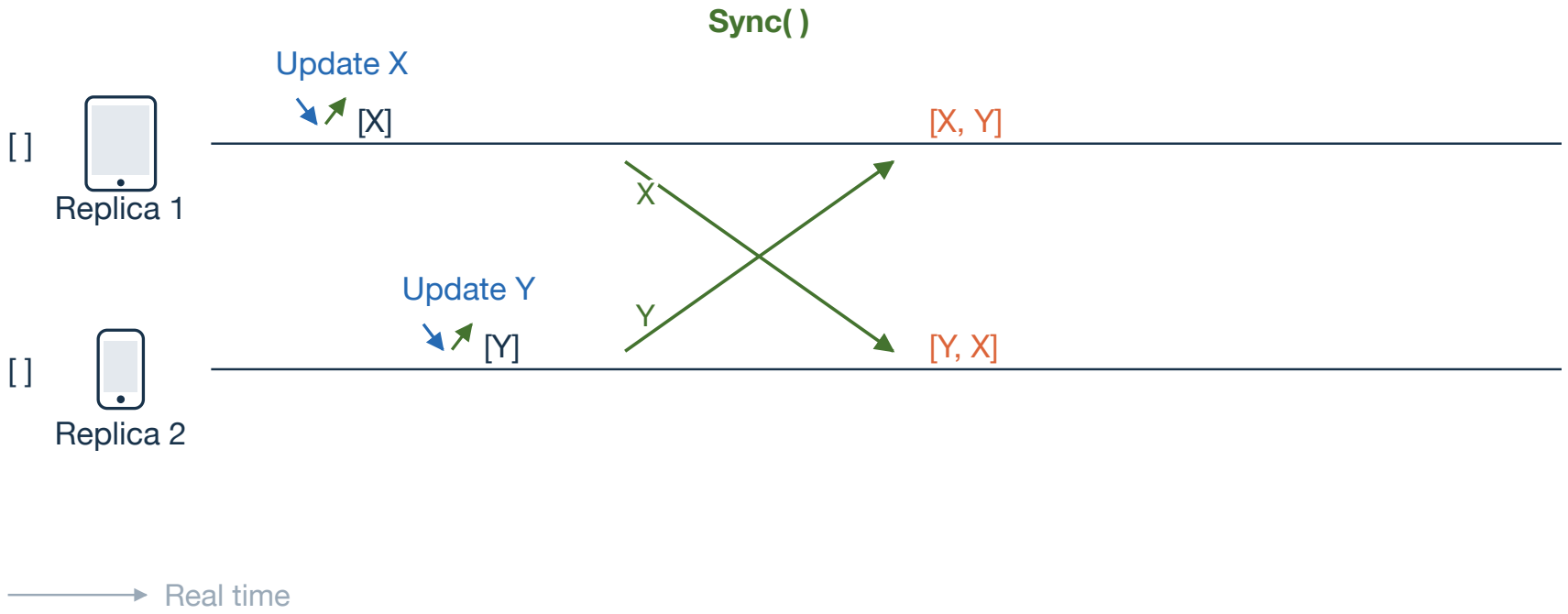


Syncing: exchange the updates received so far

Deciding the order:

- Will Replica 1 show X in front of Y, and Replica 2 show Y in front of X?

Challenge: how to decide the ordering?

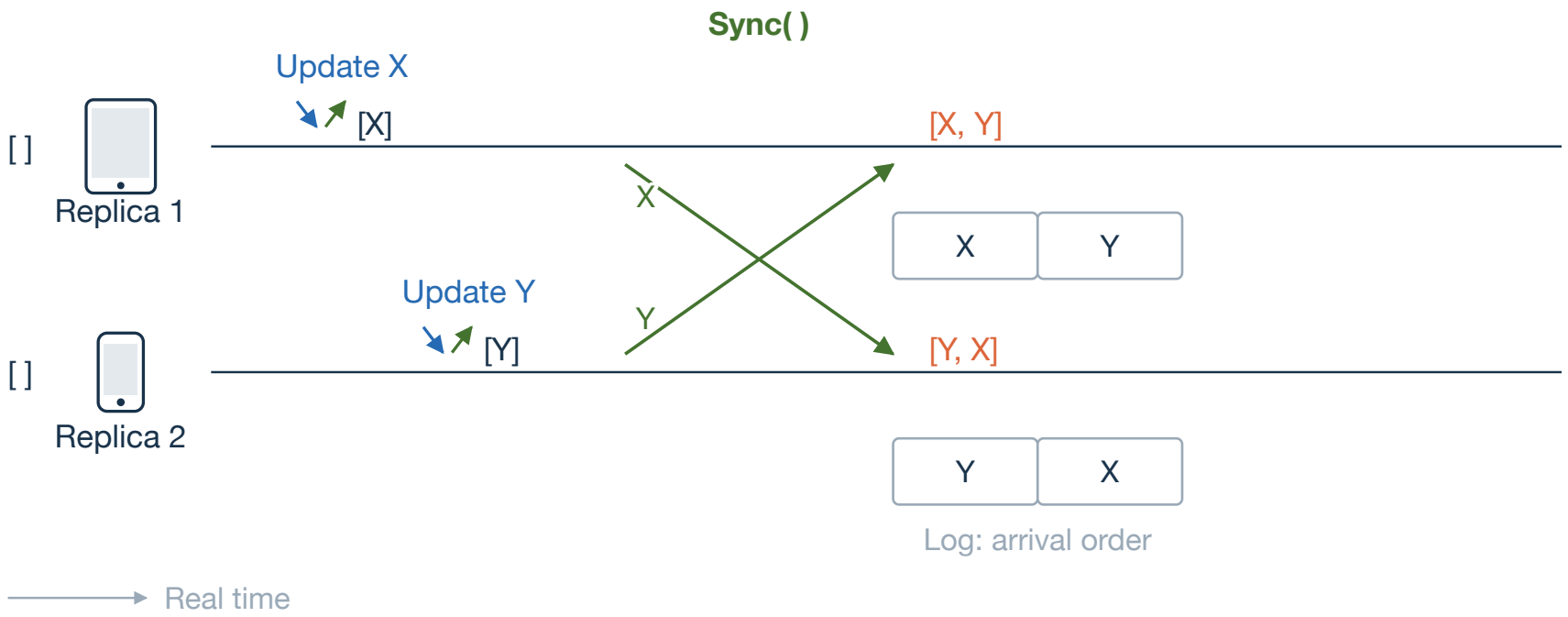


Solution: deterministically sort all updates upon syncing

Syncing ensures both nodes have the same set of updates

- Sort every update by some key, clear the state; then redo all the updates

We typically use a `log` to buffer all the updates

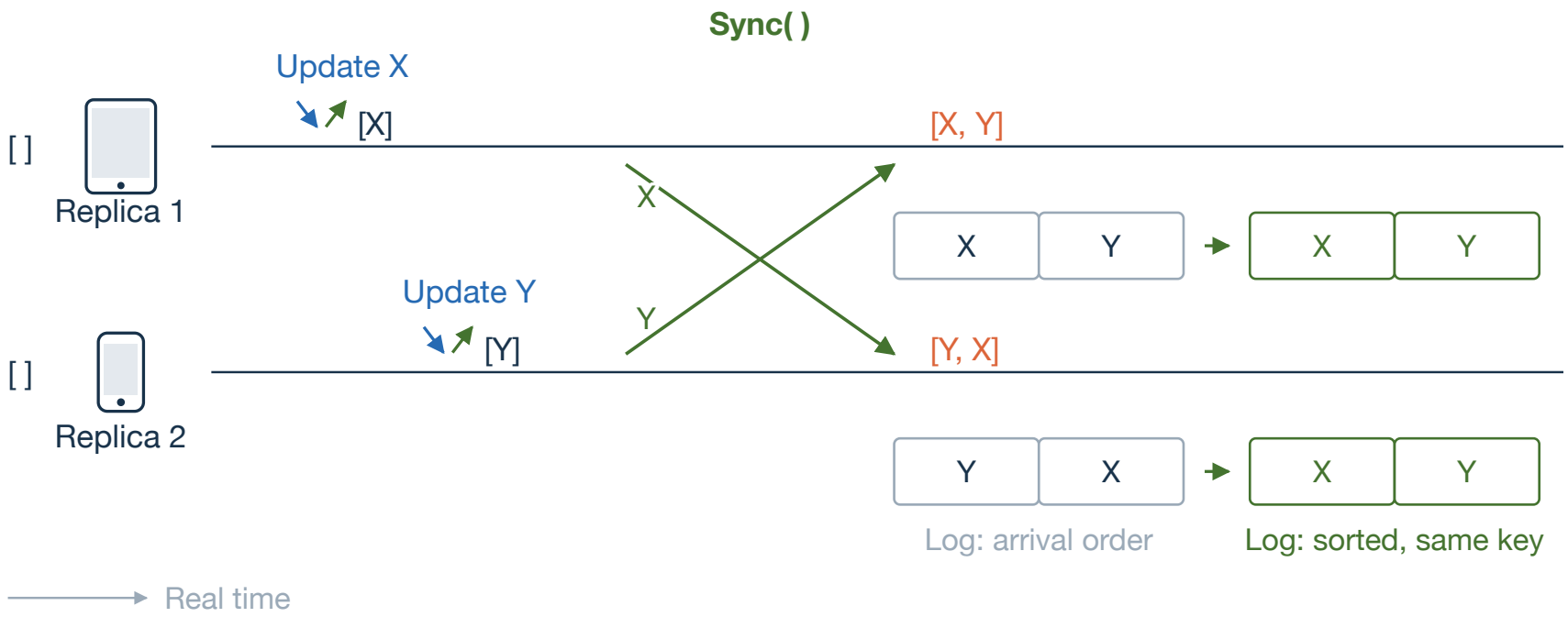


Solution: deterministically sort all updates upon syncing

Syncing ensures both nodes have the same set of updates

- Sort every update by some key, clear the state; then redo all the updates

We typically use a `log` to buffer all the updates

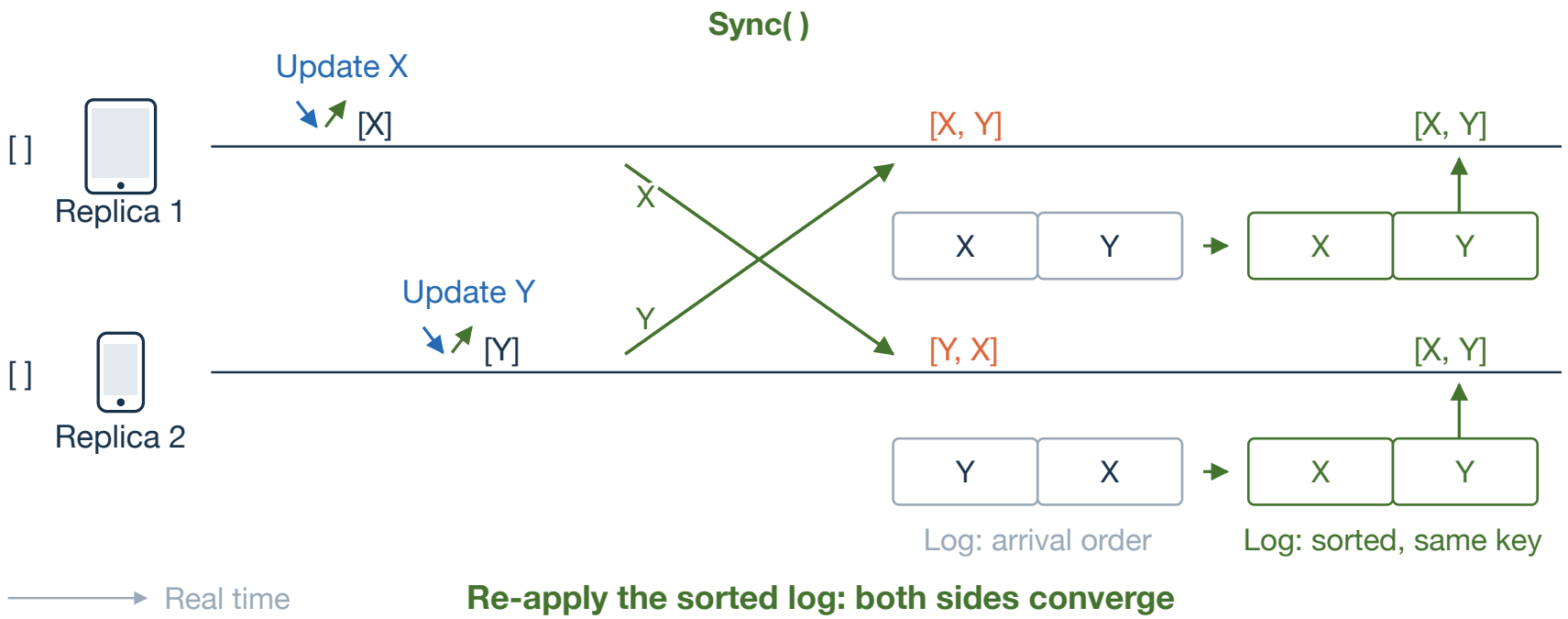


Solution: deterministically sort all updates upon syncing

Syncing ensures both nodes have the same set of updates

- Sort every update by some key, clear the state; then redo all the updates

We typically use a `log` to buffer all the updates



Question: how to sort?

The sort rule must be deterministic: the same set of updates must sort into the same order on every machine

But deterministic sorting is insufficient: not all sort orders are equal!

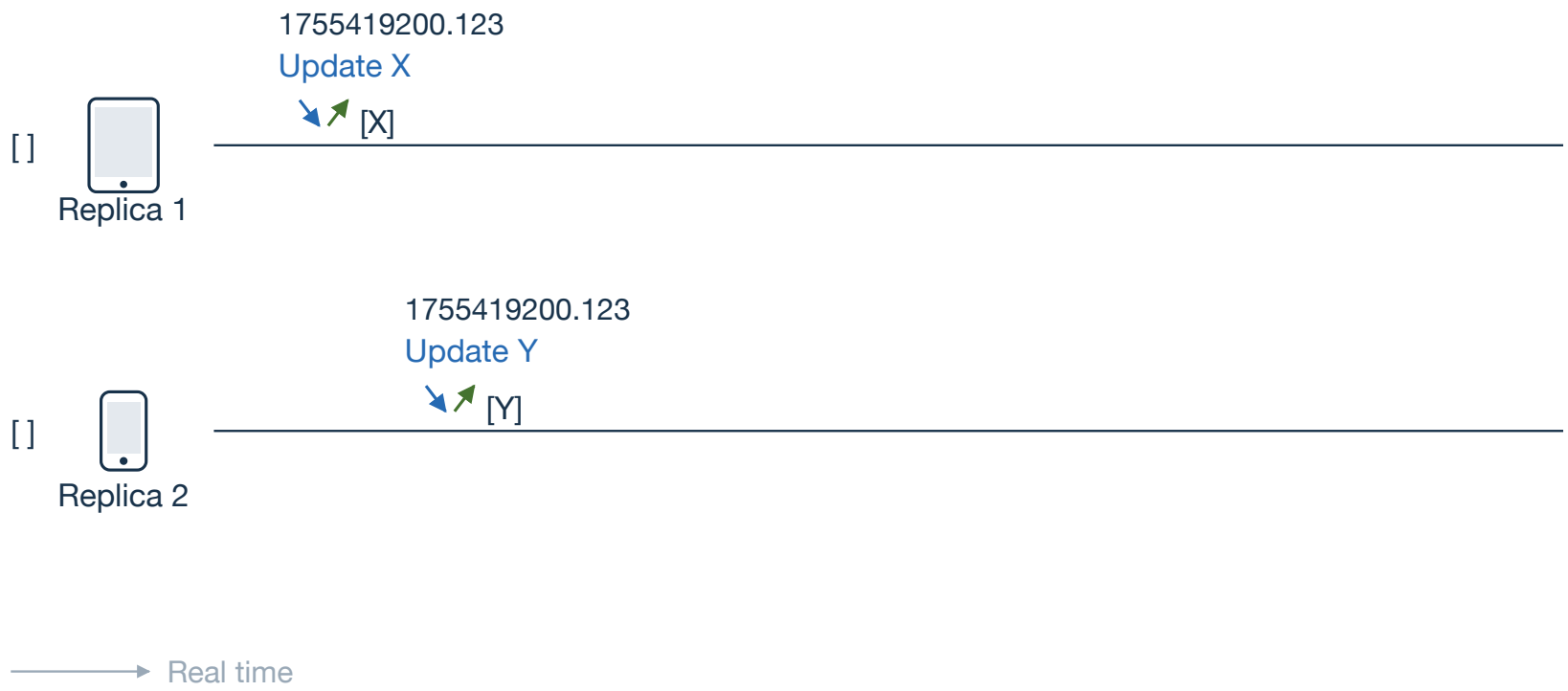
- i.e., one sort order is better than the others

As a result, we must carefully choose the sort key

Some design choices about the ordering of the sort

Can we use the time T as the sort key?

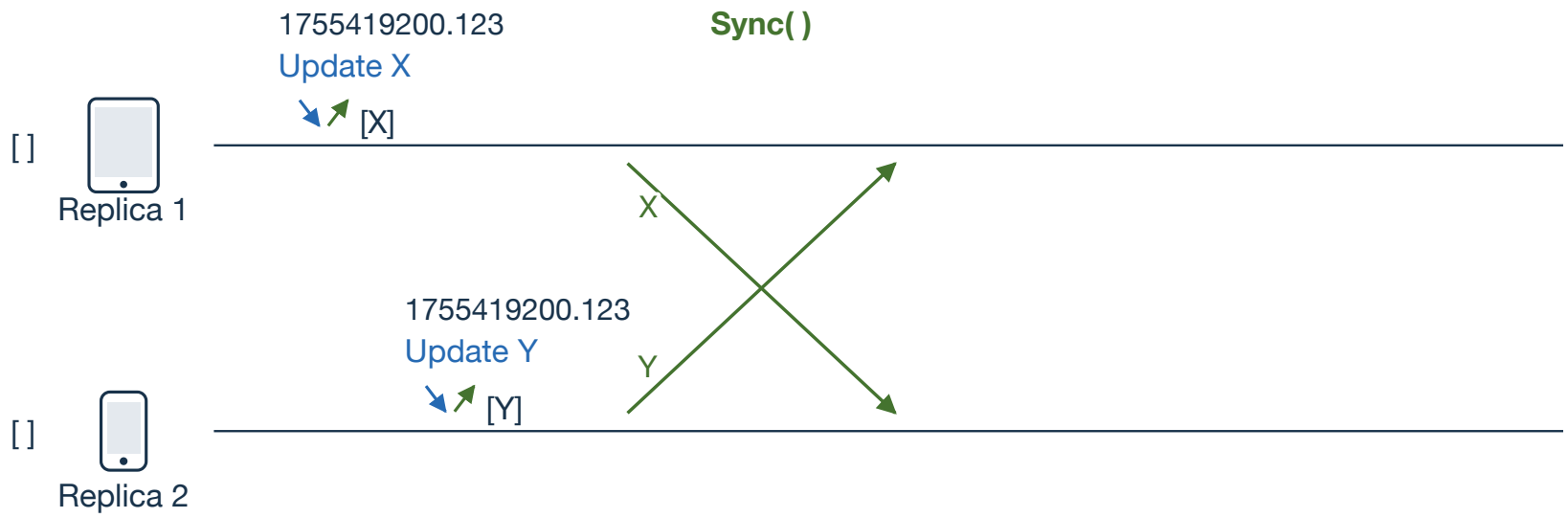
- E.g., `getTimeOfDay()` on the local device
- Intuition: a request sent first in device time must appear first



Some design choices about the ordering of the sort

Can we use the time T as the sort key?

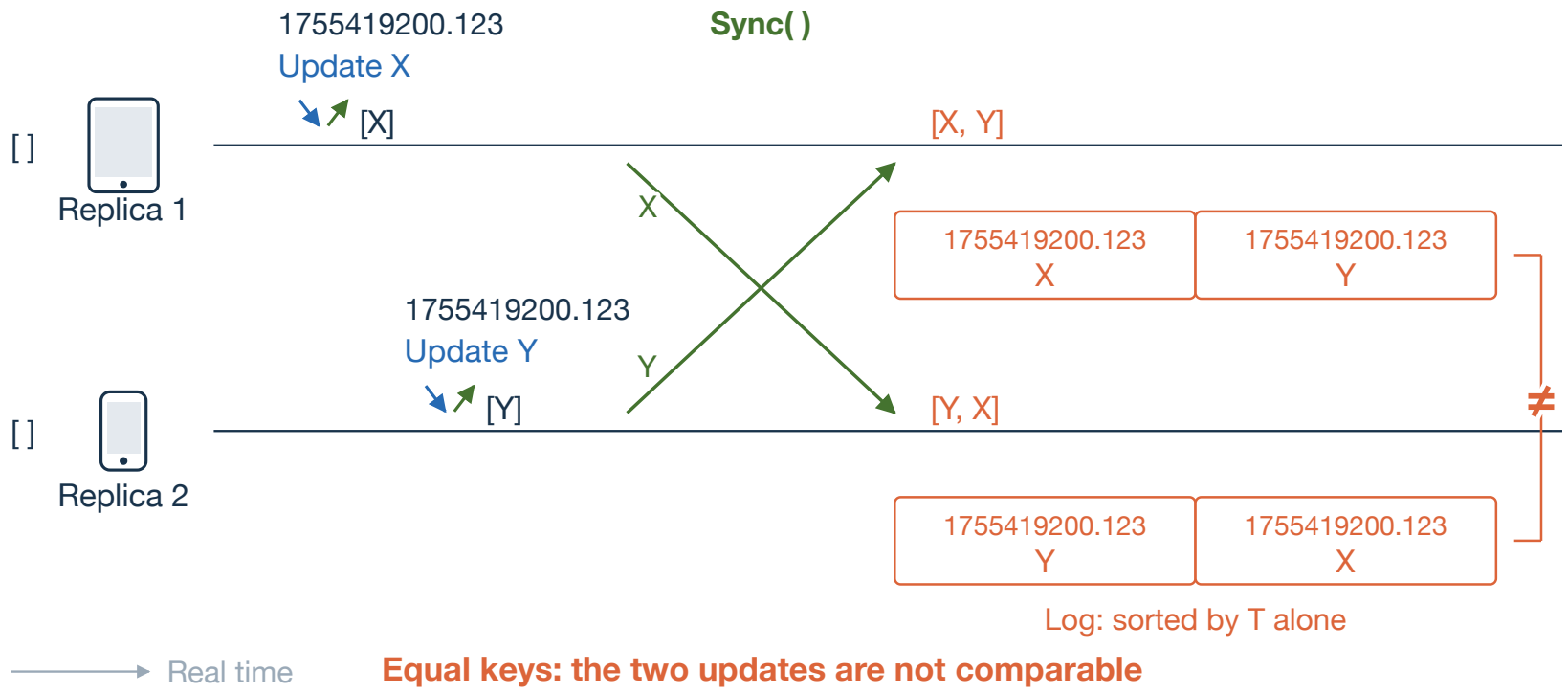
- E.g., `getTimeOfDay()` on the local device
- Intuition: a request sent first in device time must appear first



Some design choices about the ordering of the sort

Can we use the time T as the sort key?

- E.g., `getTimeOfDay()` on the local device
- Intuition: a request sent first in device time must appear first

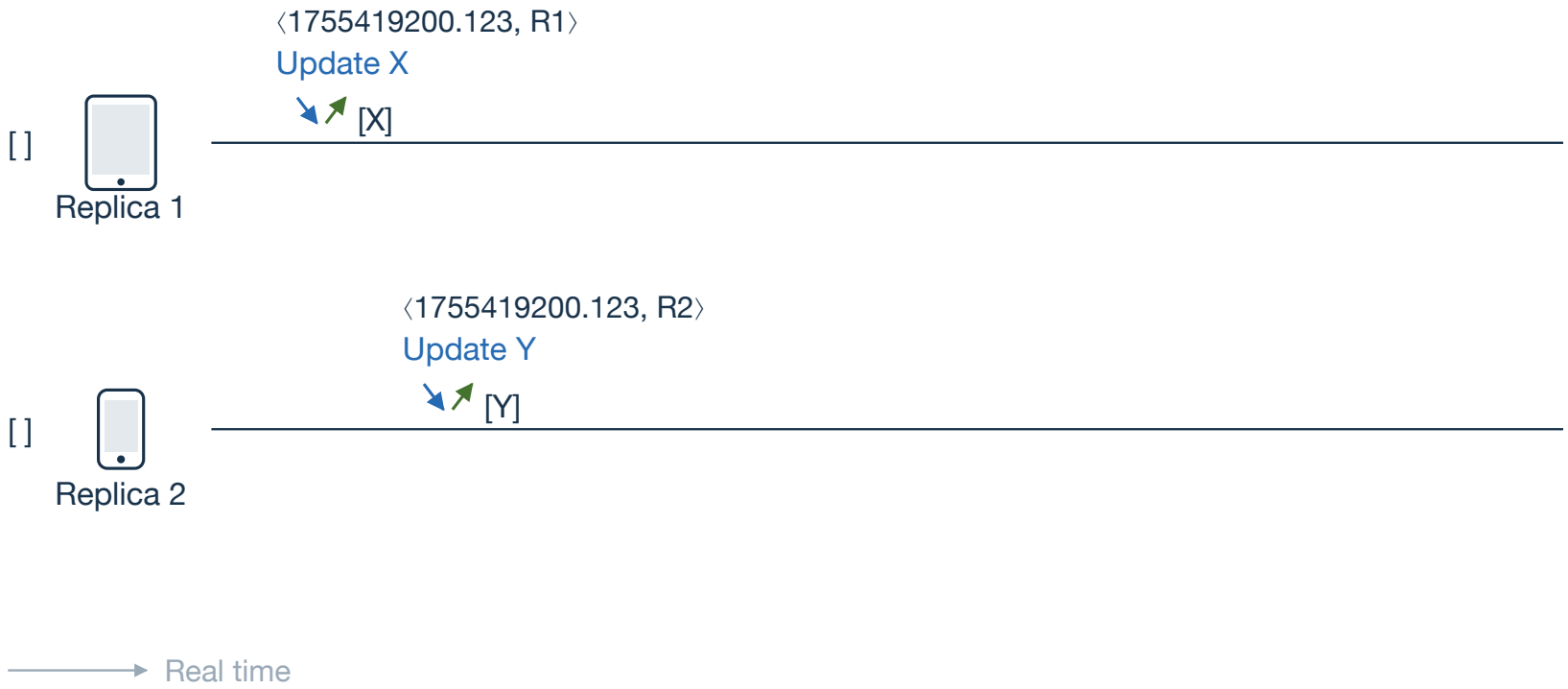


getTimeOfDay() cannot guarantee a total order

Total order: every pair of keys is comparable under the sort order (`<`)

- A clock is essentially an `integer`, so two machines can read the same T
- Equal keys cannot be ordered, and the sort is no longer deterministic

Sort key: ⟨time T, node ID⟩ — break the tie using the node ID to achieve a total order

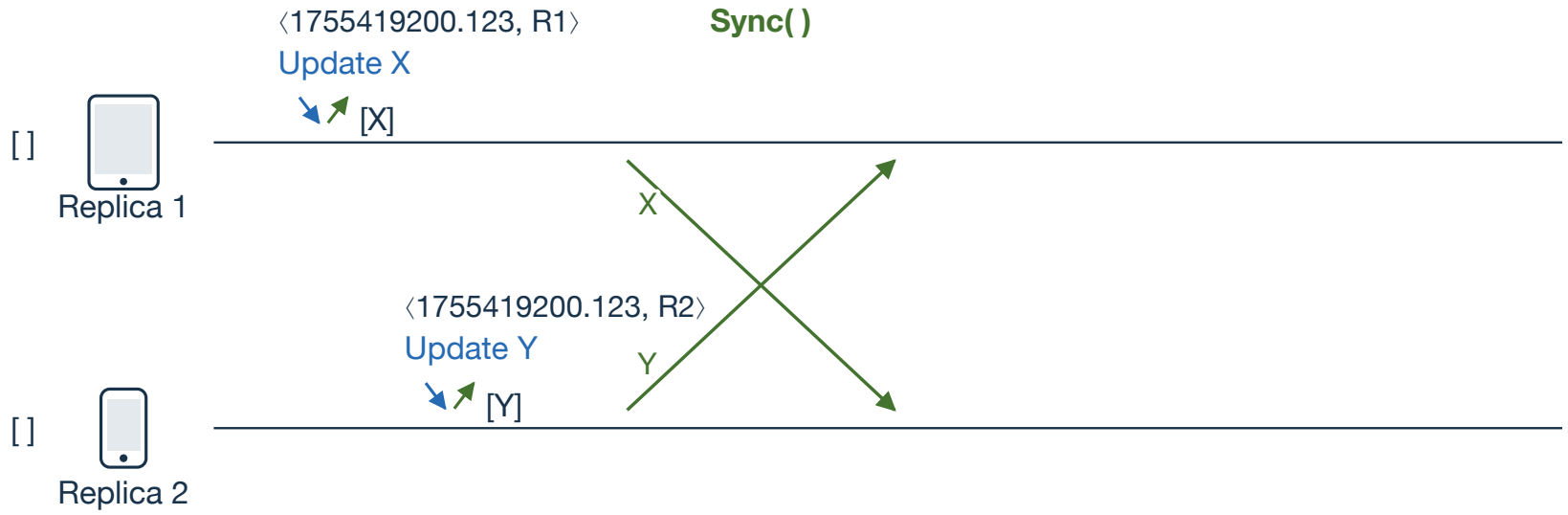


getTimeOfDay() cannot guarantee a total order

Total order: every pair of keys is comparable under the sort order (`<`)

- A clock is essentially an `integer`, so two machines can read the same T
- Equal keys cannot be ordered, and the sort is no longer deterministic

Sort key: ⟨time T, node ID⟩ — break the tie using the node ID to achieve a total order

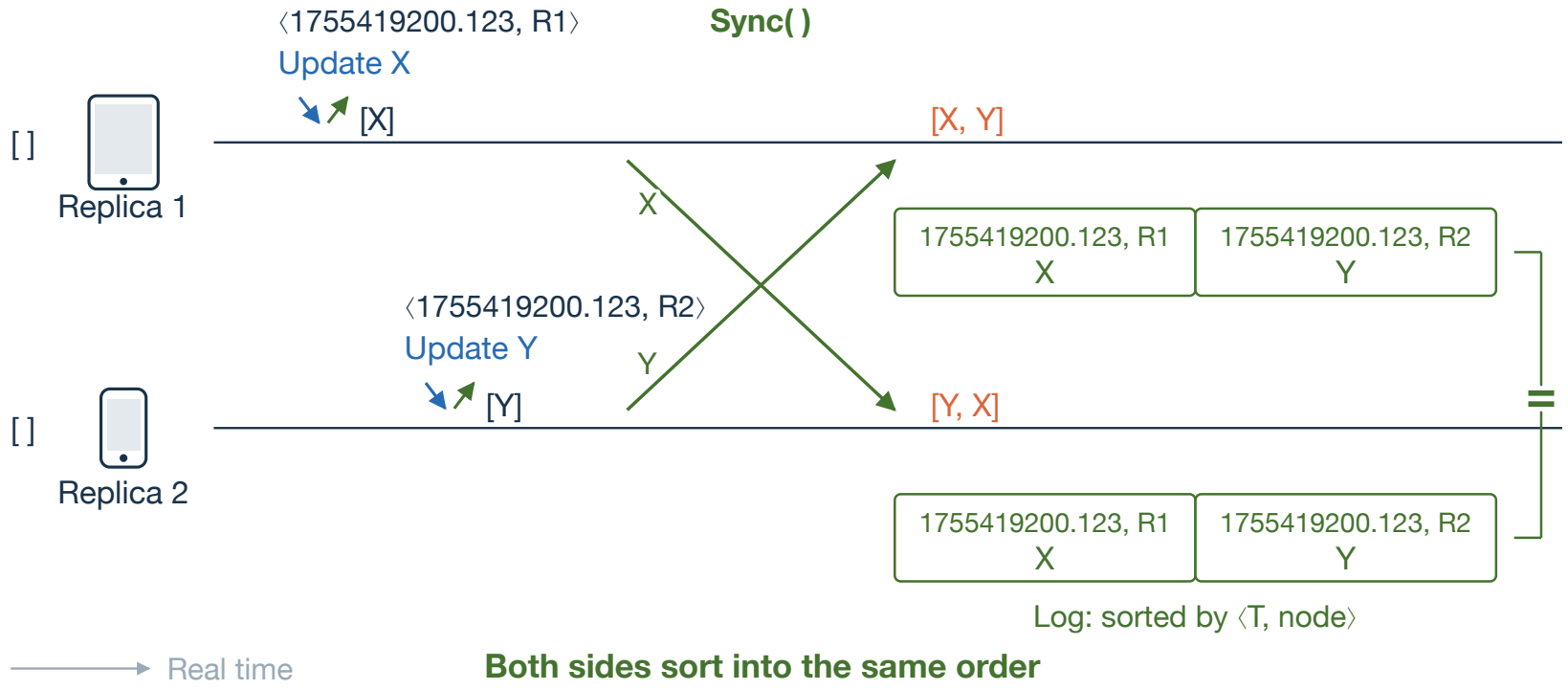


getTimeOfDay() cannot guarantee a total order

Total order: every pair of keys is comparable under the sort order (`<`)

- A clock is essentially an `integer`, so two machines can read the same T
- Equal keys cannot be ordered, and the sort is no longer deterministic

Sort key: ⟨time T, node ID⟩ — break the tie using the node ID to achieve a total order



More choices about the ordering of the sort

Can we use the time T of getTimeOfDay alone as the sort key?

One solution: use ⟨node ID, time T⟩ as the sort key

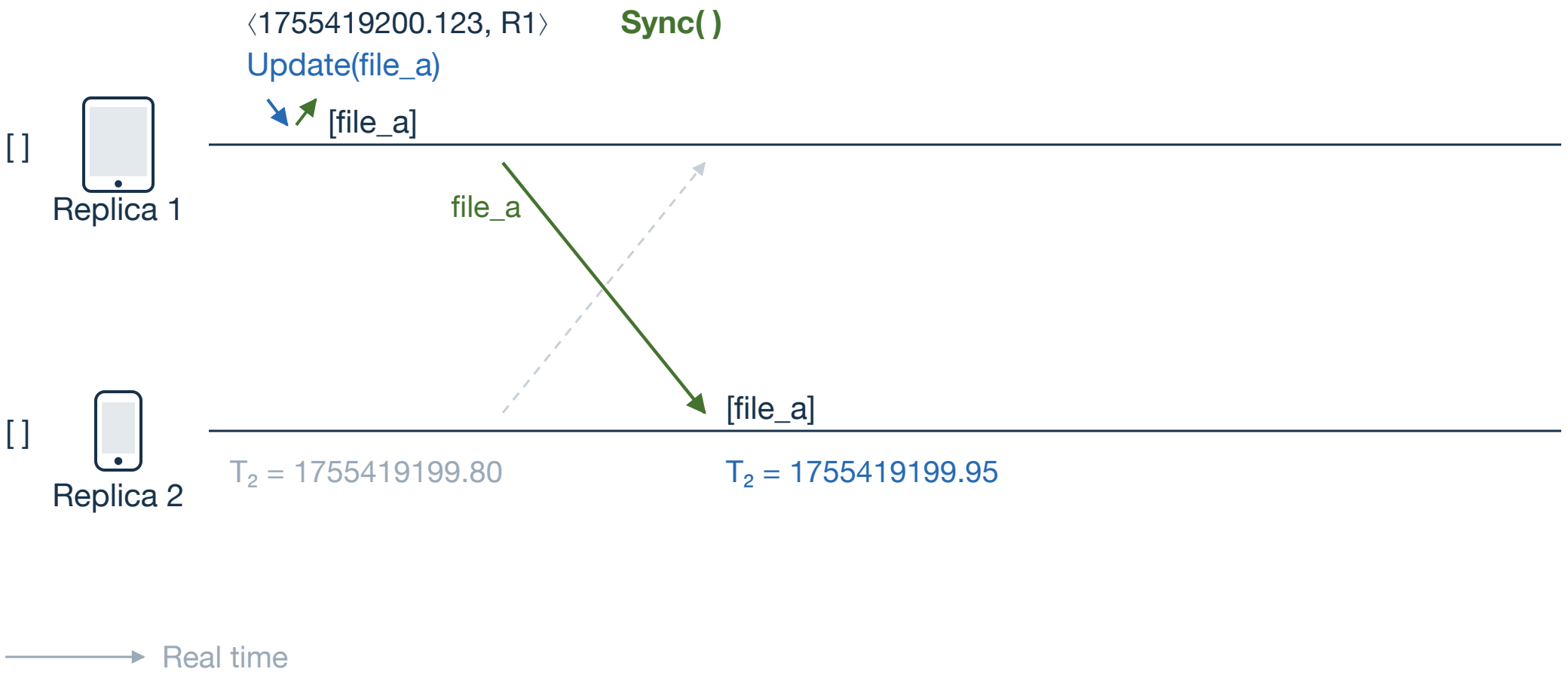
Can using ⟨time T, node ID⟩ as the key prevent the following from happening?

- Can we get the following state: ["请看文件 ref(file_a)", file_a] ?

Some orderings turn out better for the user, so we still need to take care of the sort key!

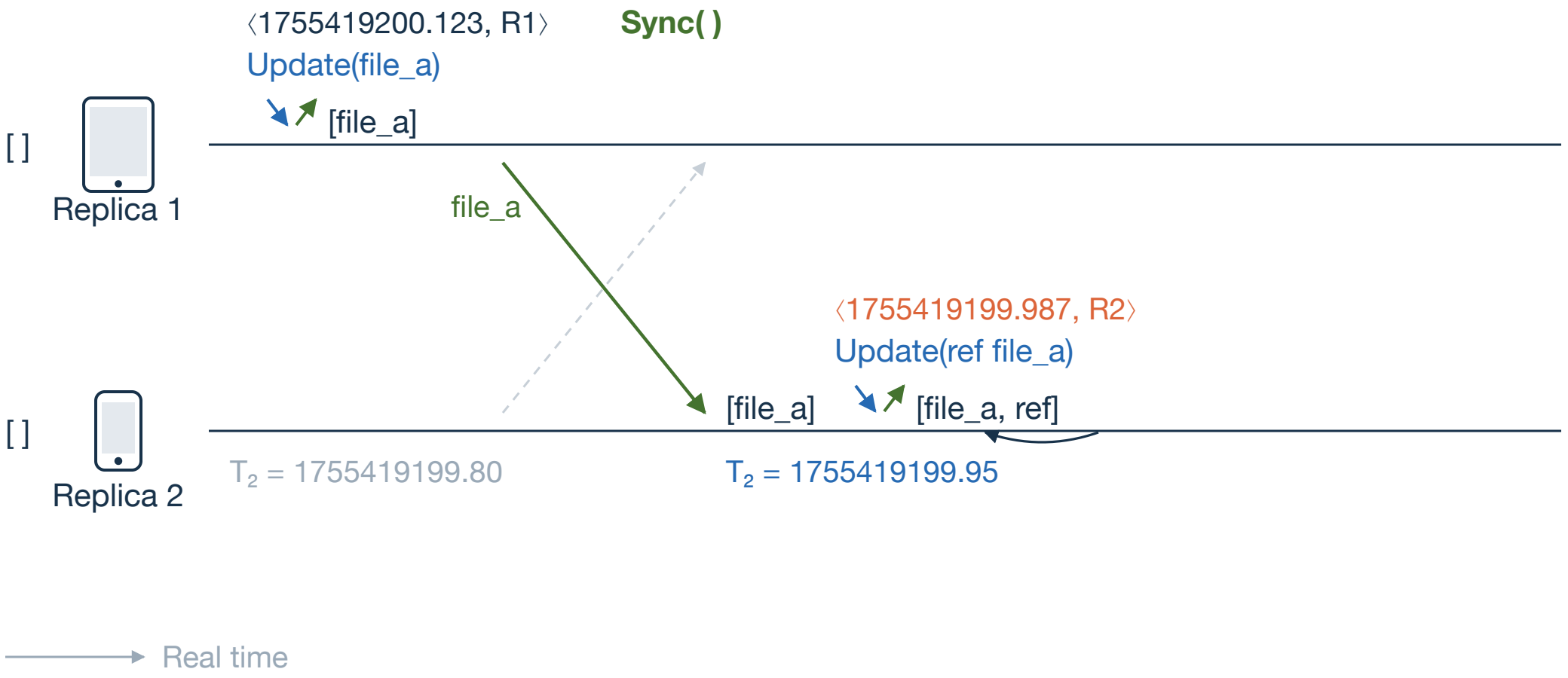
How inappropriate orders can happen upon sync

Assume updates are applied to the local replica immediately, so that we can see the latest results; inconsistency is rolled back and replayed later



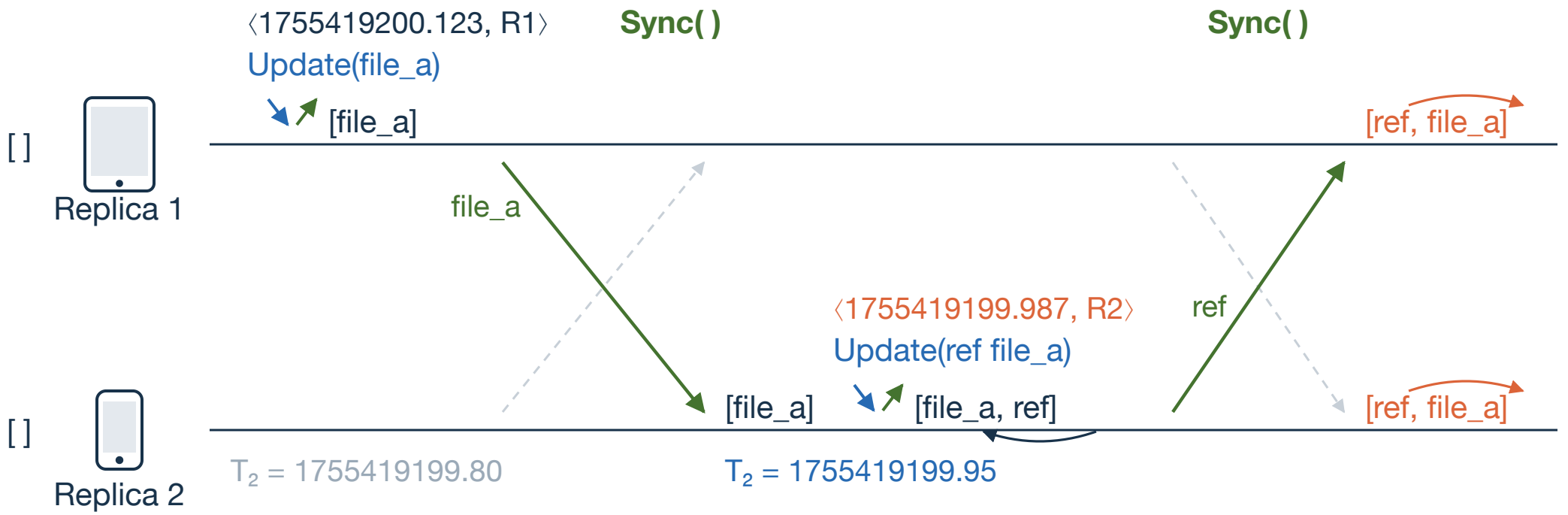
How inappropriate orders can happen upon sync

Assume updates are applied to the local replica immediately, so that we can see the latest results; inconsistency is rolled back and replayed later



How inappropriate orders can happen upon sync

Assume updates are applied to the local replica immediately, so that we can see the latest results; inconsistency is rolled back and replayed later



Consistent — but the ref precedes the file

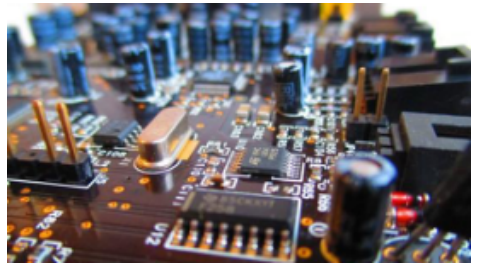
Why can such an ordering happen? Unsynchronized clock time

A computer measures time by counting oscillator cycles

- time period = count / frequency



A quartz crystal



…soldered onto the board

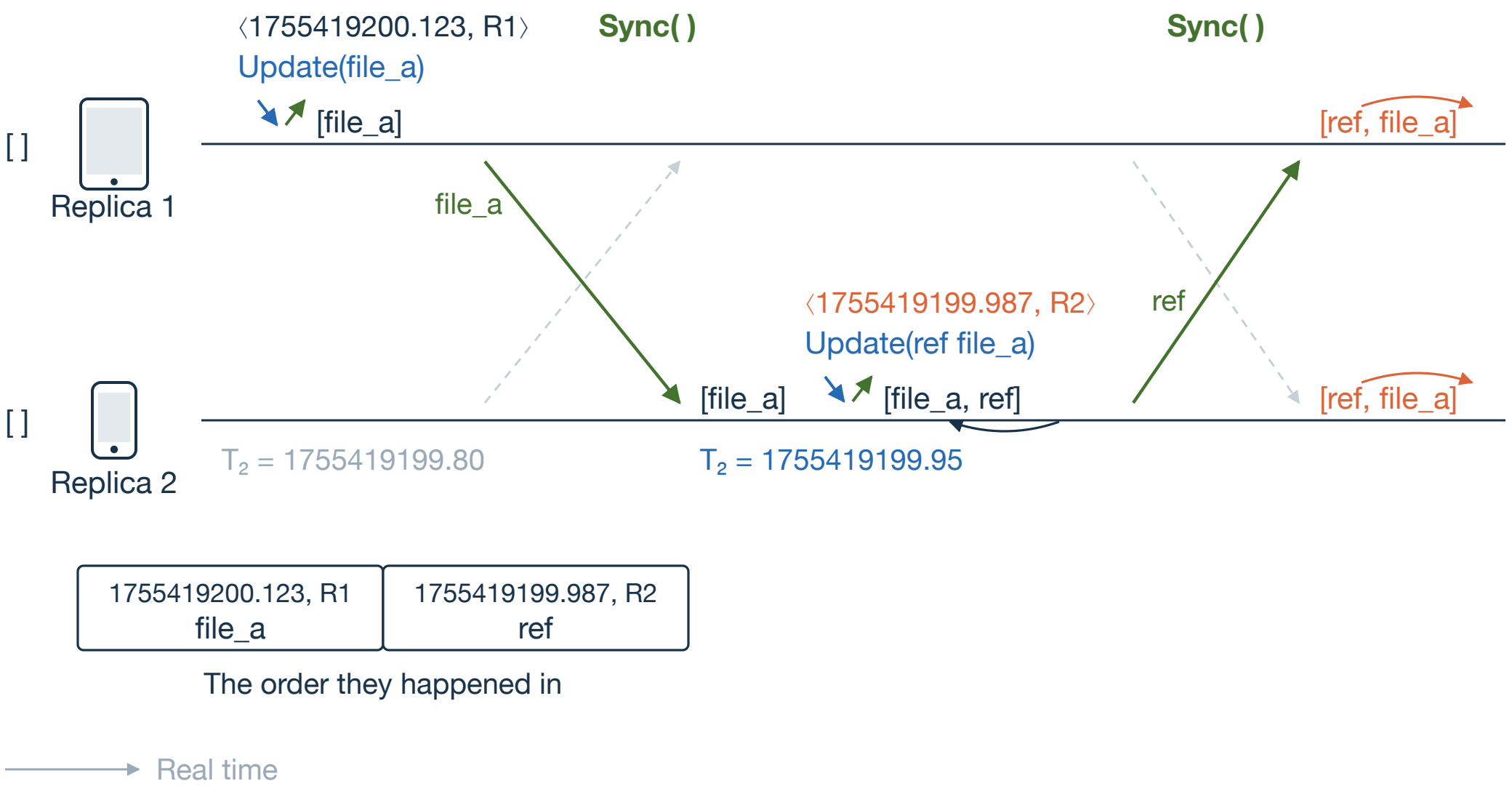
The clock between devices are incremented independently

- And factors like temperature can affect the frequency

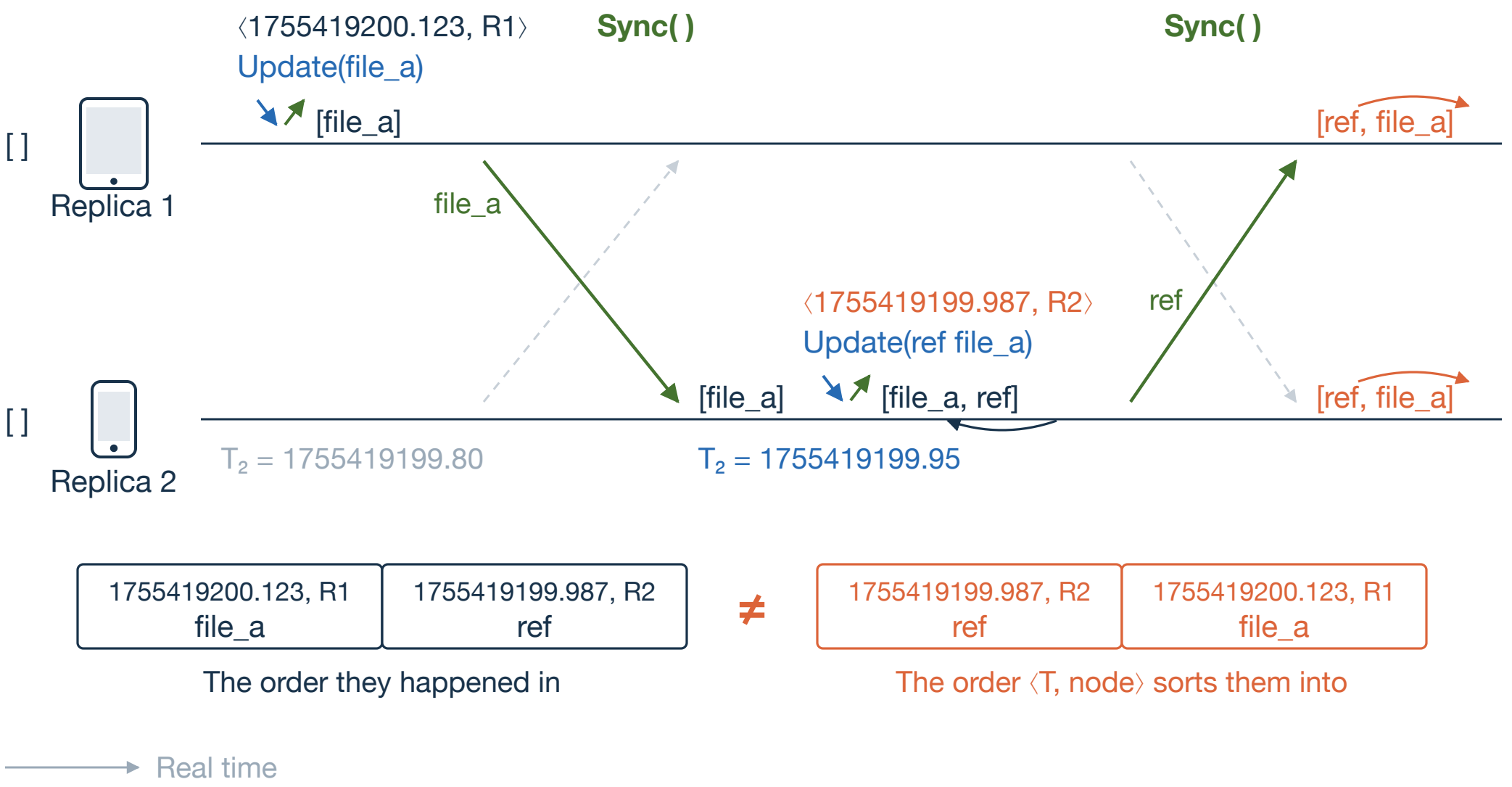
Within one device the clock follows the update order; across devices it does not

- Replica 1 went first in wall-clock time with ⟨1755419200.123, R1⟩
- Replica 2 could still generate the update ID ⟨1755419199.987, R2⟩

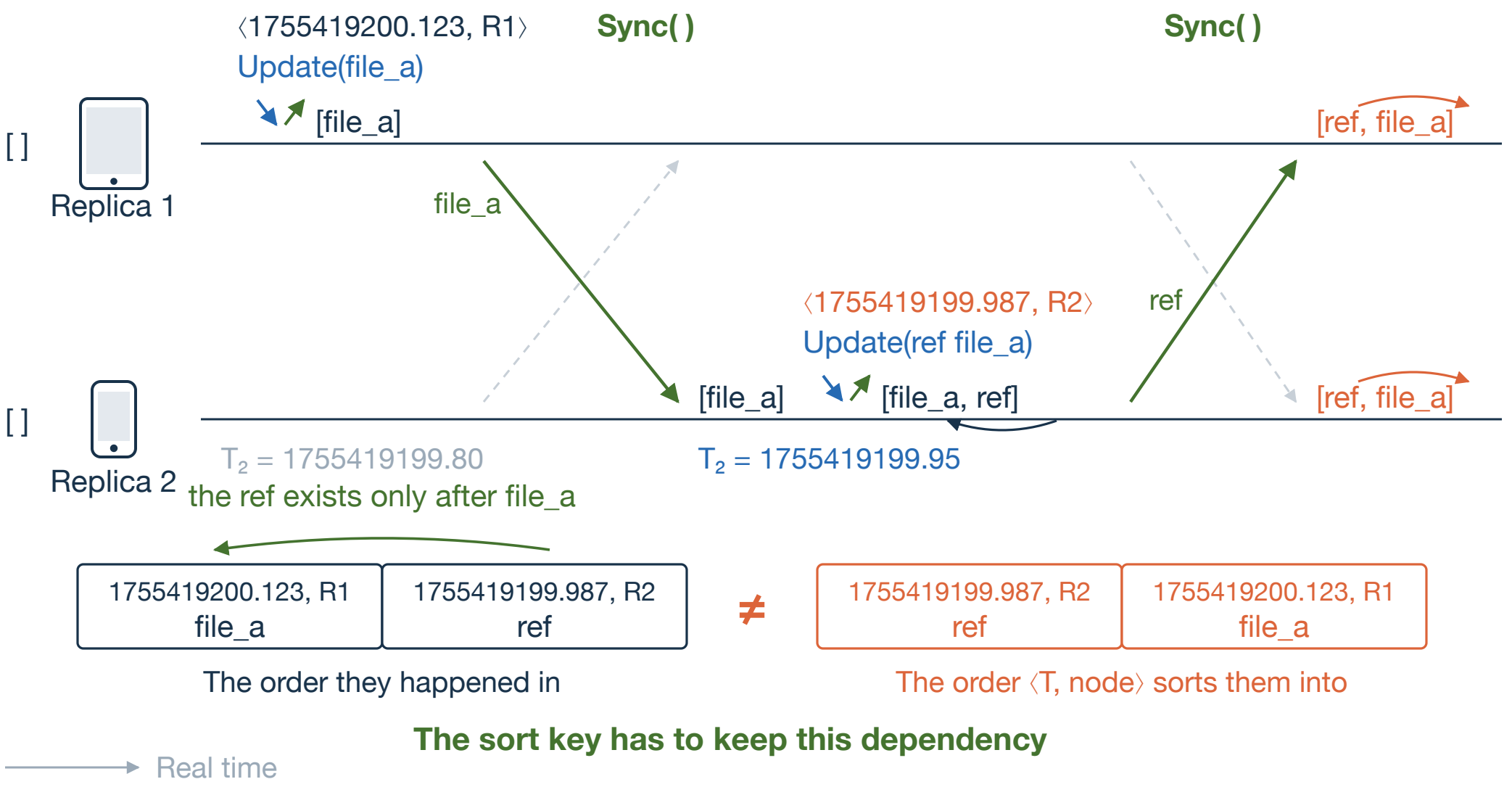
Order matters, and timing on a device alone is insufficient



Order matters, and timing on a device alone is insufficient



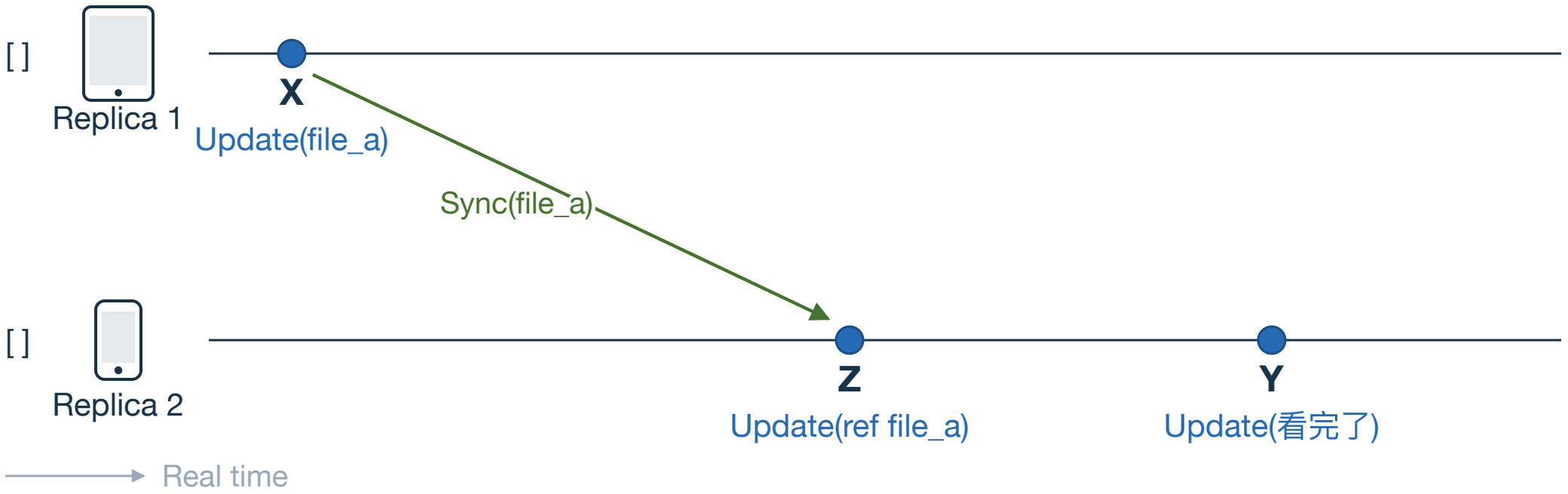
Order matters, and timing on a device alone is insufficient



An ideal order is causal ordering

Definition of the causal relationship: X → Y iff

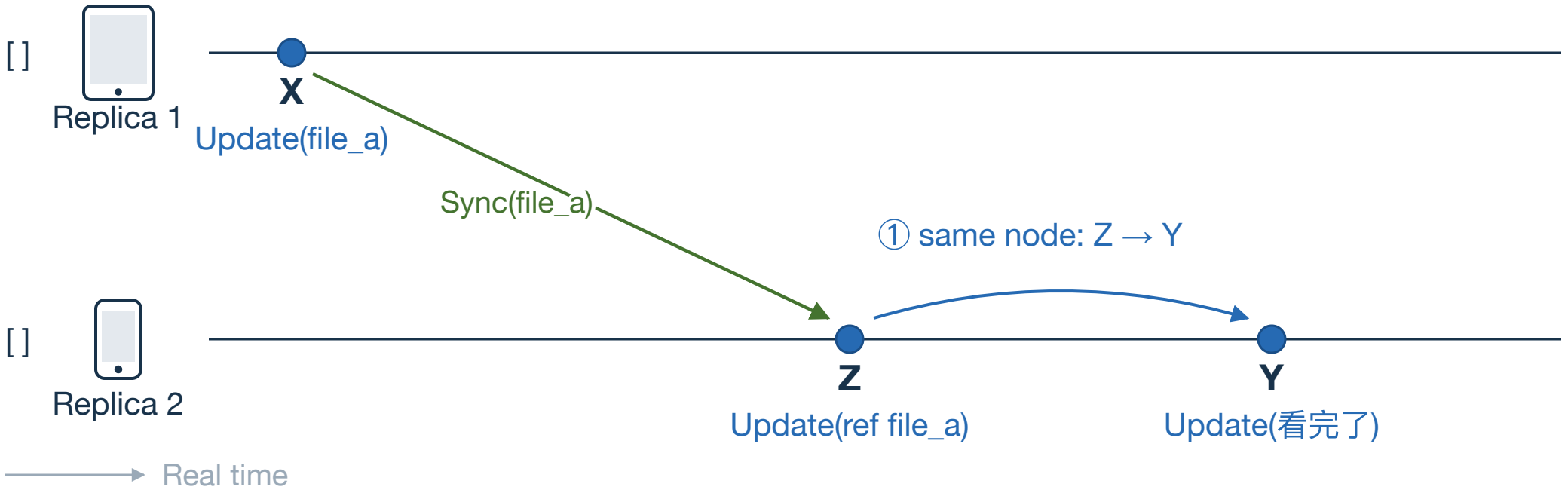
- X, Y are created on the same node and X happens before Y (single thread)
- X is created by node i and Y by node j, and X "causes" Y
- There exists Z such that X → Z → Y



An ideal order is causal ordering

Definition of the causal relationship: X → Y iff

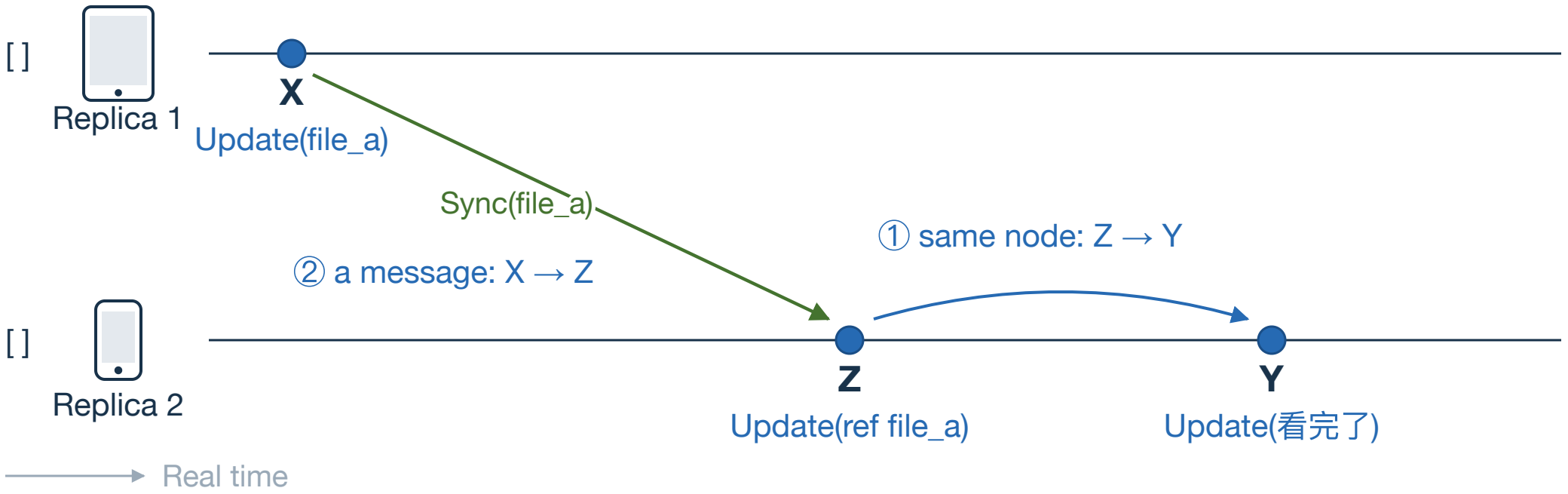
- X, Y are created on the same node and X happens before Y (single thread)
- X is created by node i and Y by node j, and X "causes" Y
- There exists Z such that X → Z → Y



An ideal order is causal ordering

Definition of the causal relationship: X → Y iff

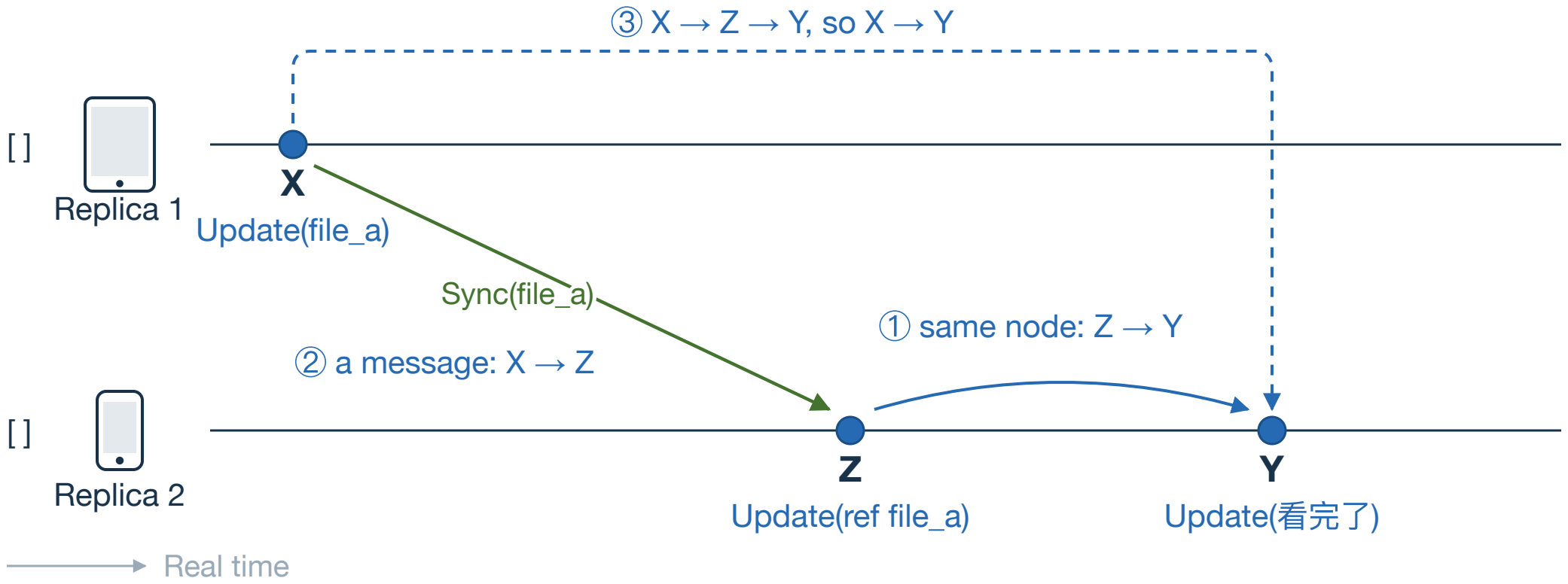
- X, Y are created on the same node and X happens before Y (single thread)
- X is created by node i and Y by node j, and X "causes" Y
- There exists Z such that X → Z → Y



An ideal order is causal ordering

Definition of the causal relationship: X → Y iff

- X, Y are created on the same node and X happens before Y (single thread)
- X is created by node i and Y by node j, and X "causes" Y
- There exists Z such that X → Z → Y



We need a sort key (timestamp) that preserves the causal order!

⟨time, node ID⟩ does not preserve the causal ordering

- Because the clocks on different nodes are unsynchronized

Goal

- The clock should reflect the causal order

Idea

- Sync the clocks using the dependencies between events
- Why is this possible? Because the causal relationship in the computing world is typically triggered by communications!

Solution: Lamport clock

The goal: we need a timestamp reflecting the causal order between events

Lamport clock: a logical clock used to assign timestamps to events at each node, with the following properties:

- If `event #1` and `event #2` are created on the same node and `event #1` is before `event #2`, then `TS(event #1) < TS(event #2)`
- If `event #1` "causes" event #2, then `TS(event #1) < TS(event #2)`

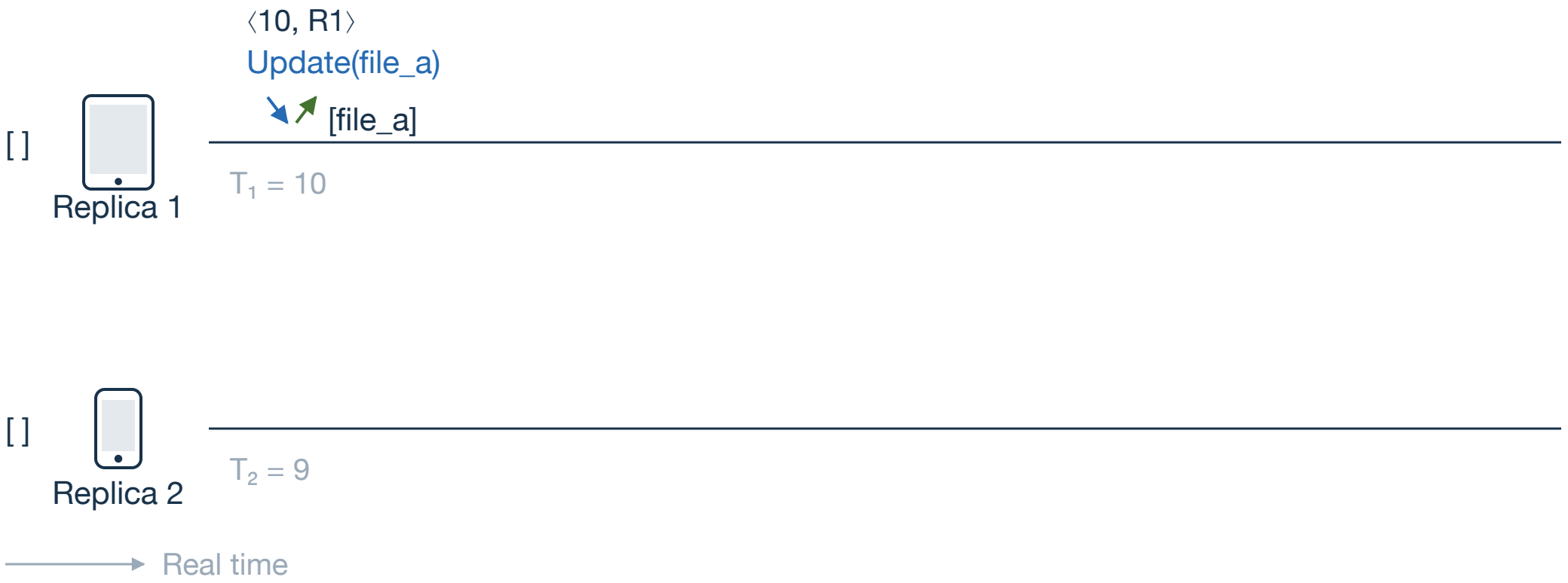
How does event #1 cause event #2? By receiving messages

- After receiving a message of event #1, all events on this node happen after event #1

Lamport clock algorithm

Each server keeps a clock T

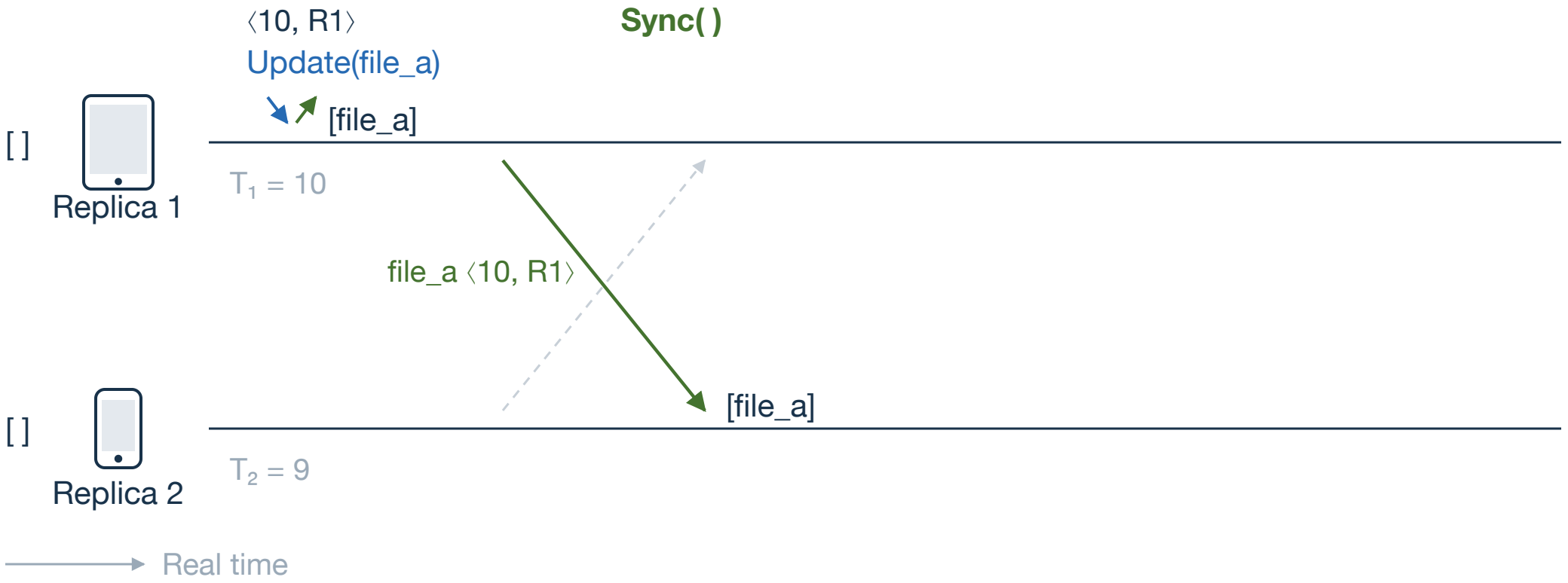
- Increment T as real time passes, e.g., one second per second (or upon events)
- Modify T = max(T, T′ + 1) if it sees T′ from another server



Lamport clock algorithm

Each server keeps a clock T

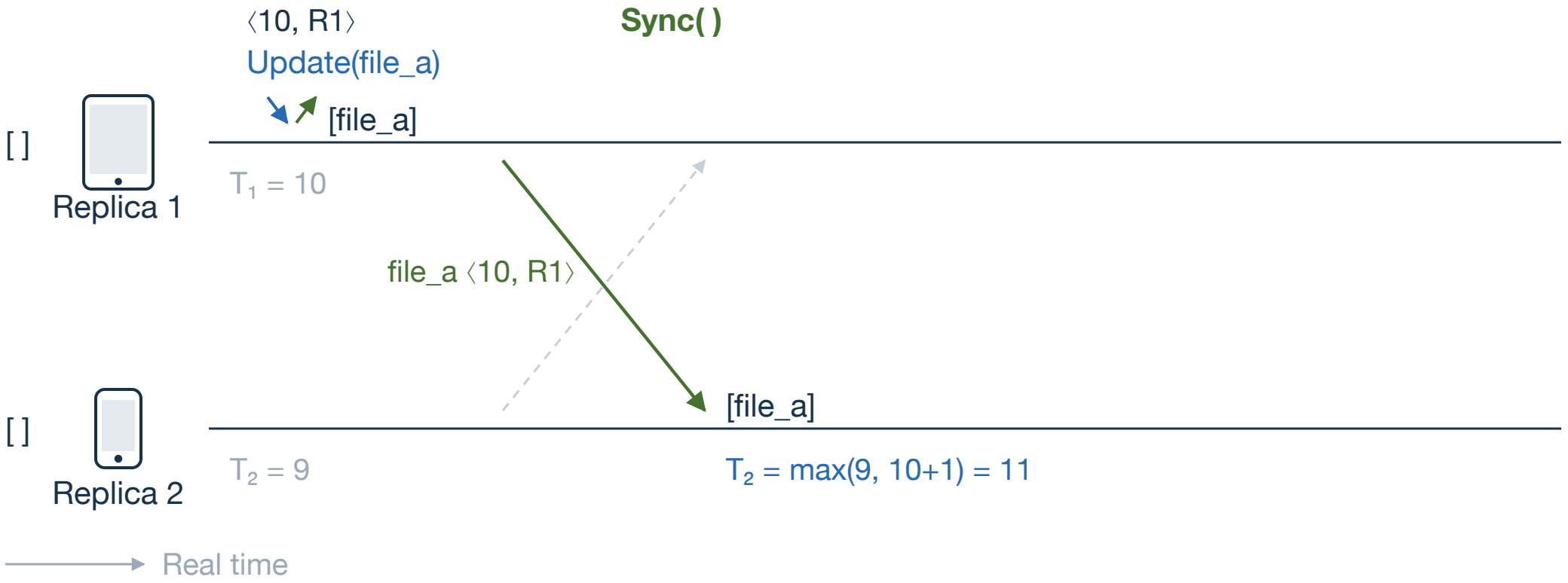
- Increment T as real time passes, e.g., one second per second (or upon events)
- Modify T = max(T, T′ + 1) if it sees T′ from another server



Lamport clock algorithm

Each server keeps a clock T

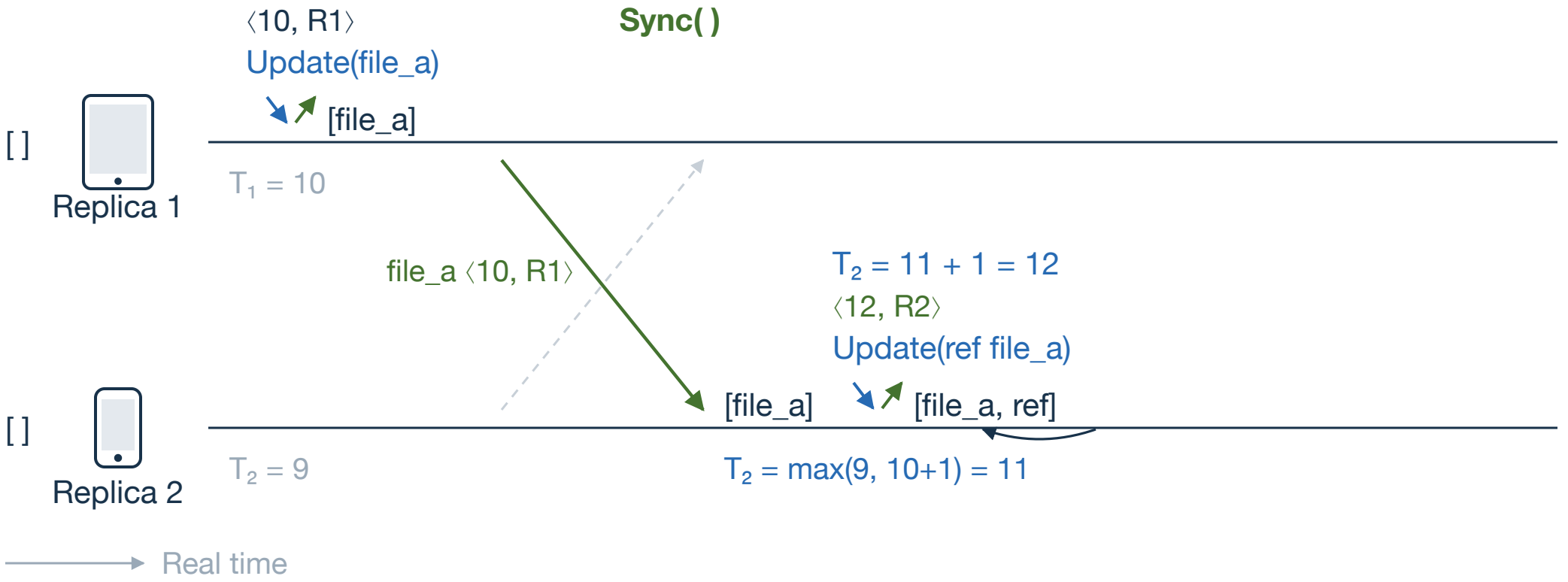
- Increment T as real time passes, e.g., one second per second (or upon events)
- Modify T = max(T, T′ + 1) if it sees T′ from another server



Lamport clock algorithm

Each server keeps a clock T

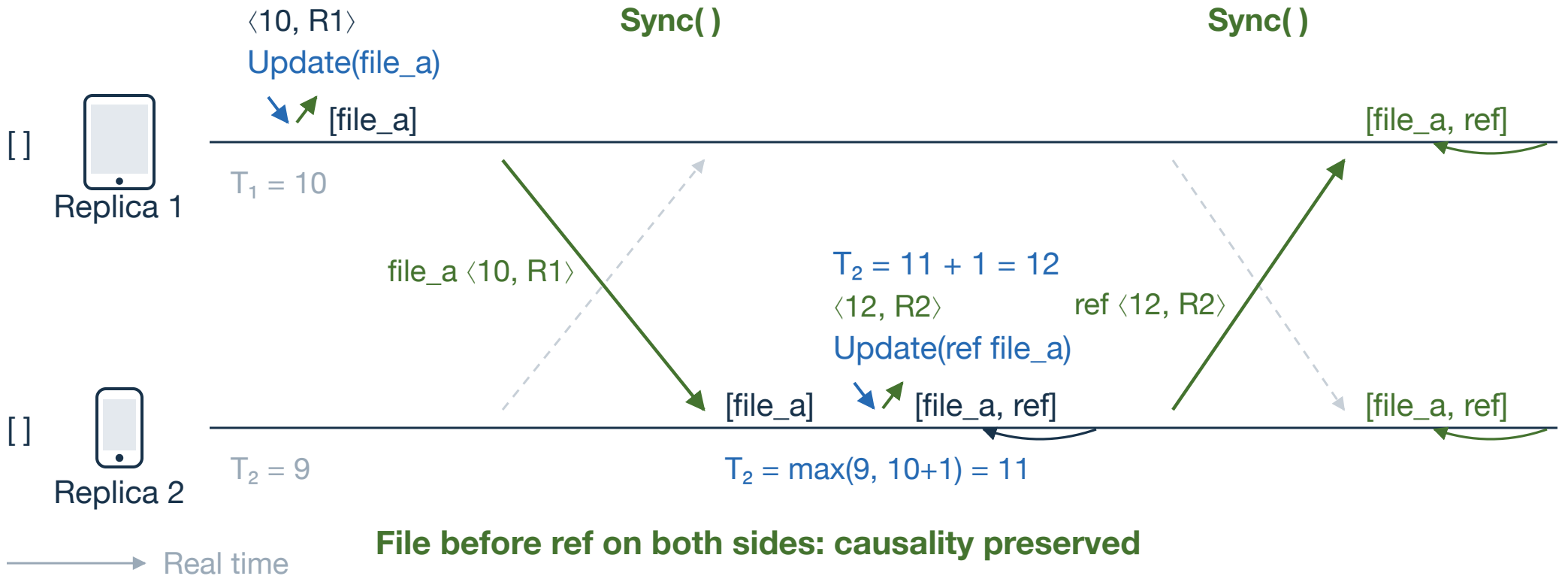
- Increment T as real time passes, e.g., one second per second (or upon events)
- Modify T = max(T, T′ + 1) if it sees T′ from another server



Lamport clock algorithm

Each server keeps a clock T

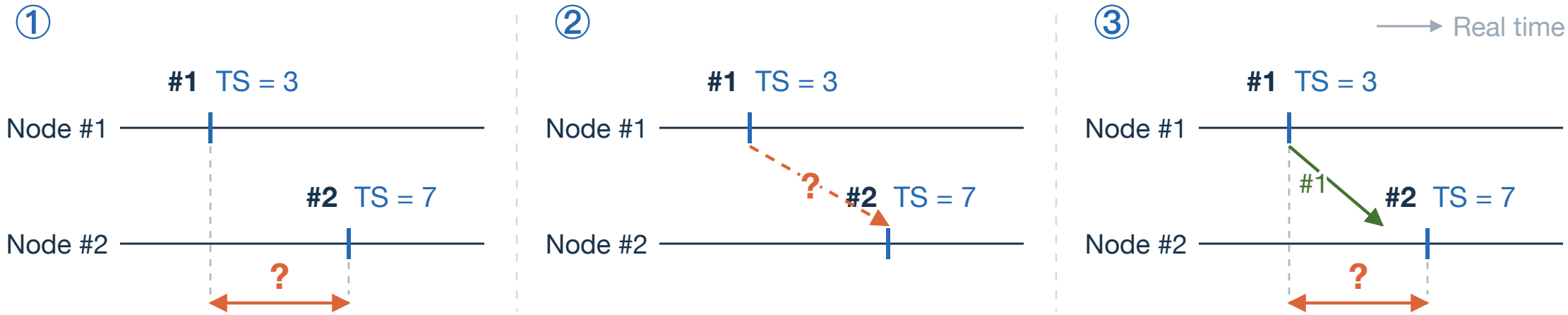
- Increment T as real time passes, e.g., one second per second (or upon events)
- Modify T = max(T, T′ + 1) if it sees T′ from another server



File before ref on both sides: causality preserved

Questions on Lamport clock

If `TS(event #1) < TS(event #2)`, what does that say about event #1 (created on node #1) and event #2 (created on node #2)?



1. Event #1 occurred at a physical time earlier than event #2
2. Node #1 must have communicated with node #2
3. If event #1 has been synced to node #2, then event #1 must have occurred at a physical time earlier than event #2

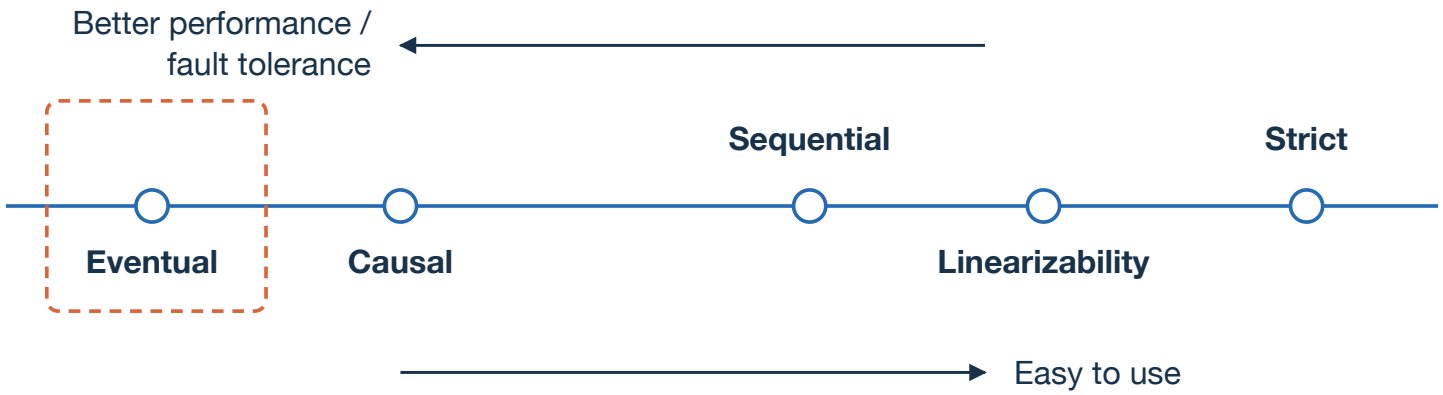
Recap of today's lecture

What is a consistency model & why we need it

The eventual consistency model

Implementing eventual consistency w/ various timestamps

- Non-trivial once we consider user experience: preserving causality takes a logical clock



Question: is eventual consistency desirable for every application scenario?

Motivating case: 接龙

接龙: the first to sign up gets the bonus

- Update: append yourself to the 接龙
- Read: fetch whatever the 接龙 holds right now



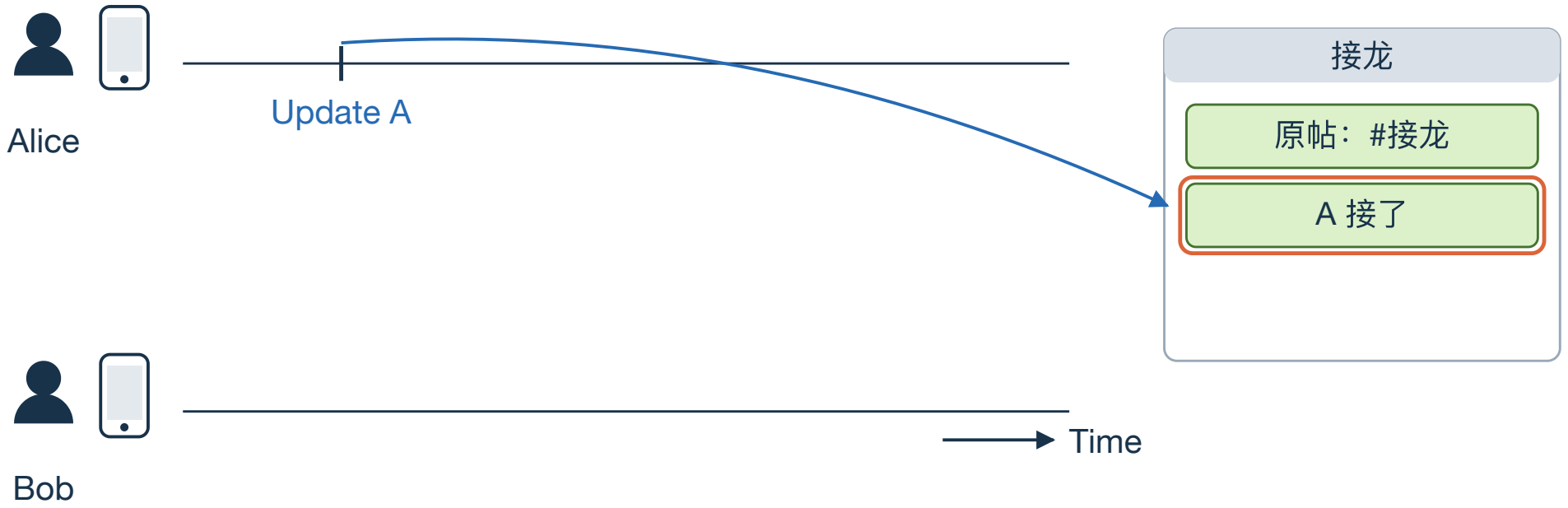


Motivating case: 接龙

接龙: the first to sign up gets the bonus

- Update: append yourself to the 接龙
- Read: fetch whatever the 接龙 holds right now



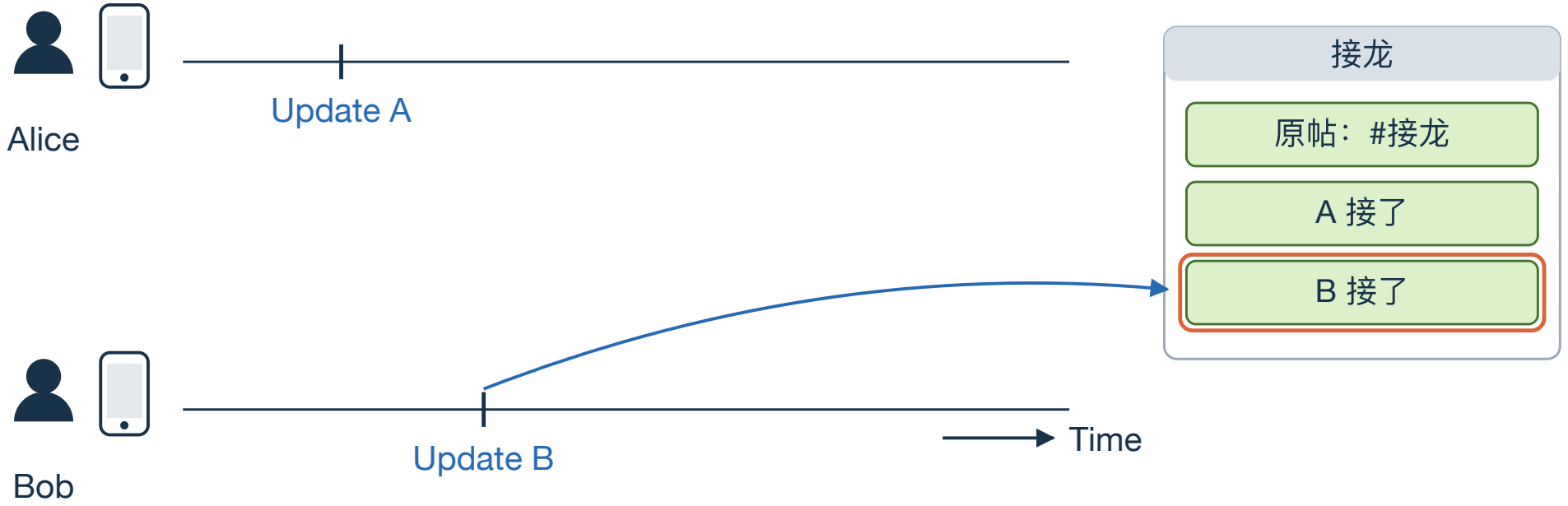


Motivating case: 接龙

接龙: the first to sign up gets the bonus

- Update: append yourself to the 接龙
- Read: fetch whatever the 接龙 holds right now



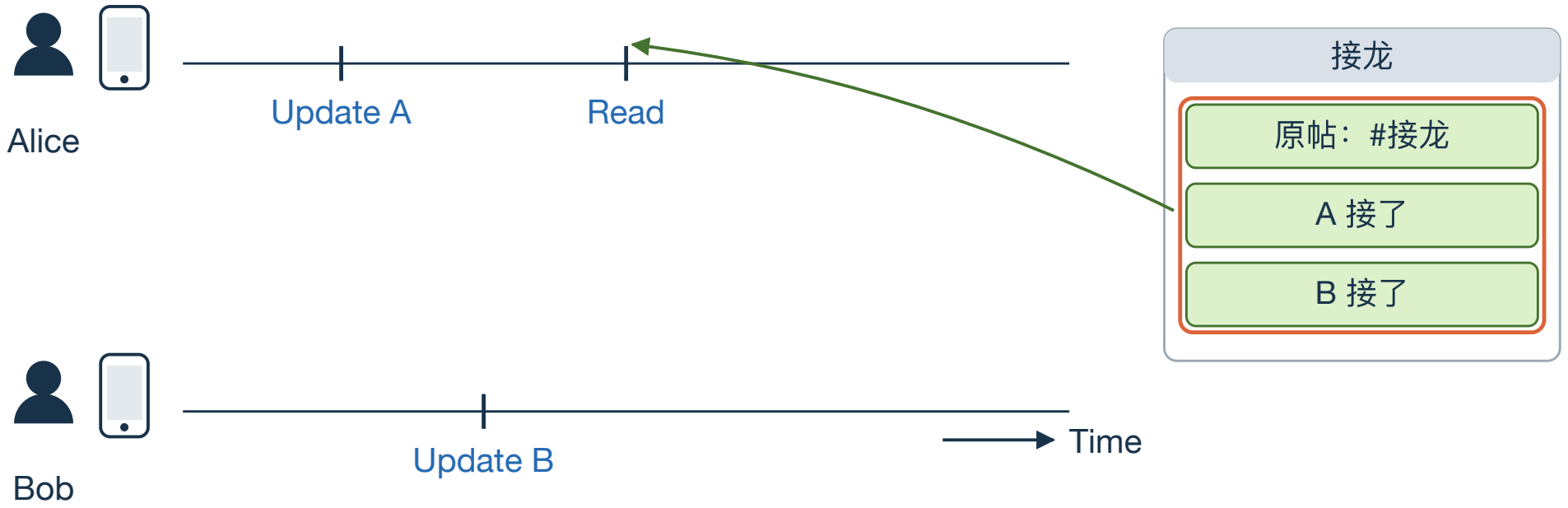


Motivating case: 接龙

接龙: the first to sign up gets the bonus

- Update: append yourself to the 接龙
- Read: fetch whatever the 接龙 holds right now



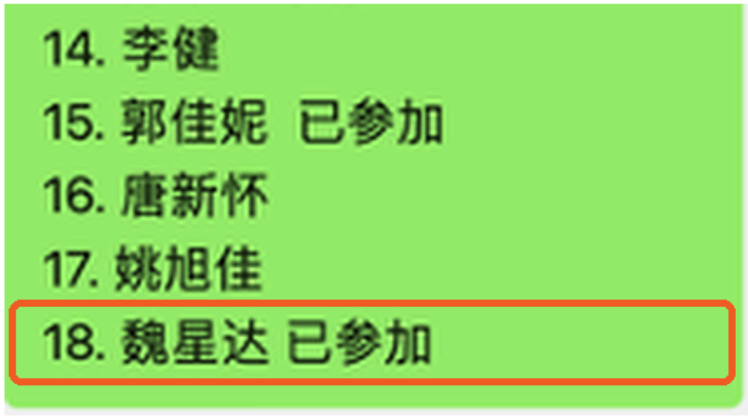


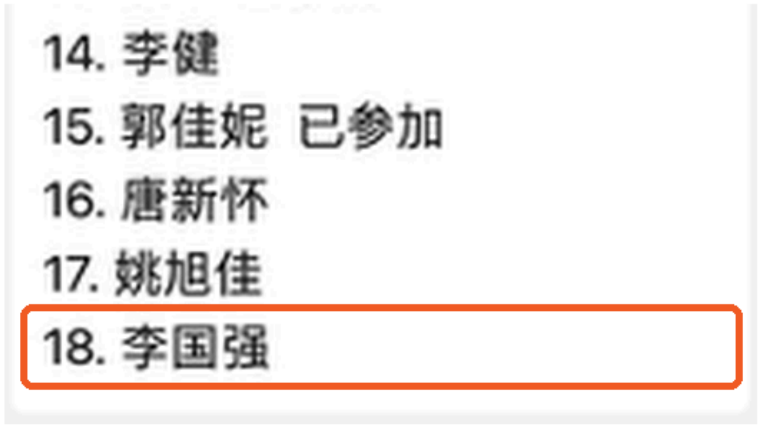
Question: can two people take the same slot

The same 接龙, updated by two of its participants

- Identical through 17; both of them appended themselves as 18







Problem: what if the application requires a strong ordering

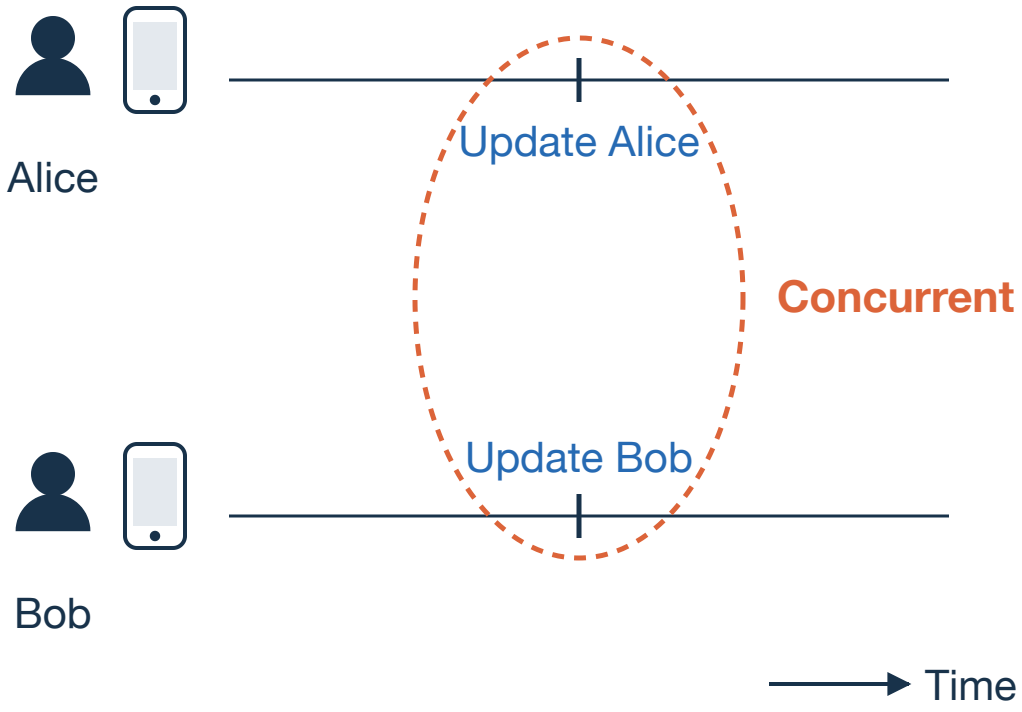
Reading temporary results that can be reordered is not desirable

- E.g., if we say the first can get the prize in 接龙

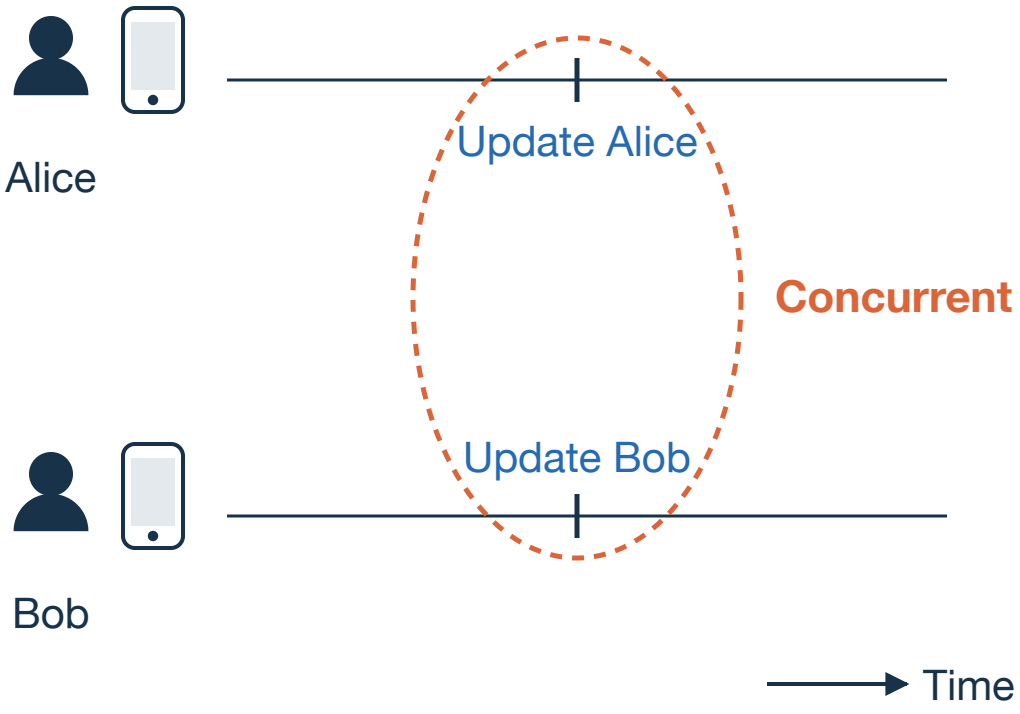
Two solutions for our 接龙

1. If we cannot determine the order of events, e.g., there is no causal relationship, tell the application events are not comparable
2. We choose a consistency model that cannot expose reordered events

Concurrent updates: invent an order, or report a conflict?

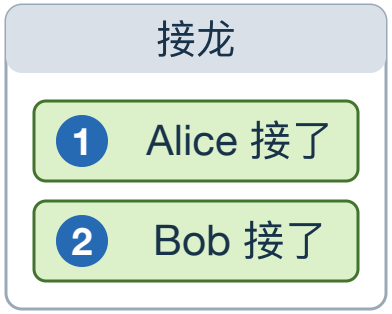


Concurrent updates: invent an order, or report a conflict?



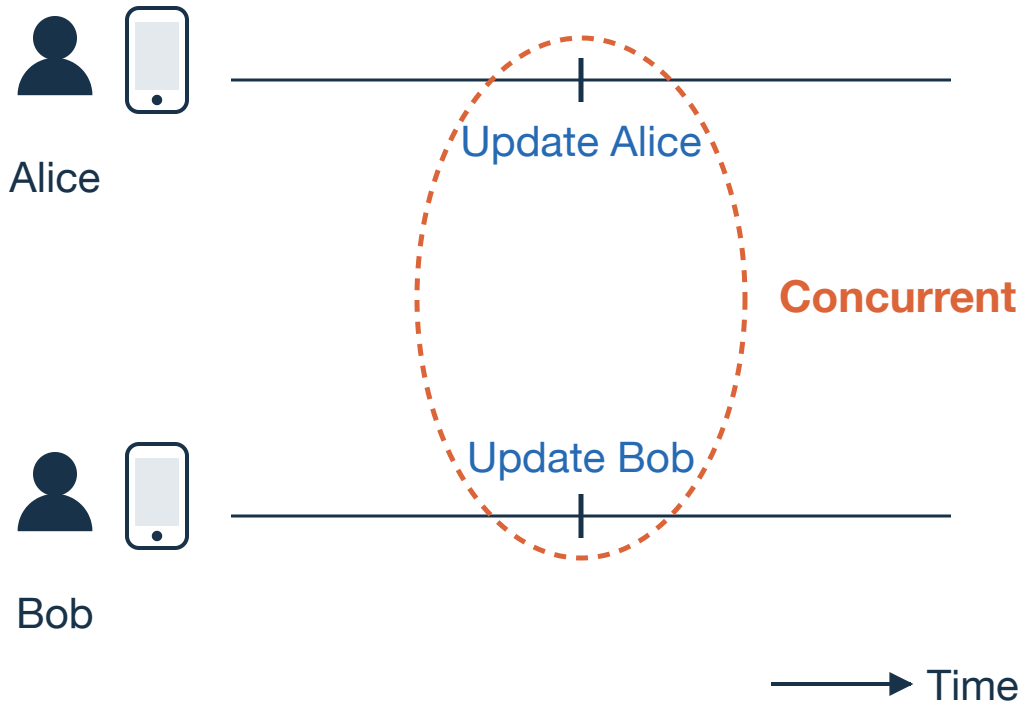
Order them anyway

(Lamport clock)



Alice 拿奖金

Concurrent updates: invent an order, or report a conflict?



Order them anyway

(Lamport clock)

接龙

1 Alice 接了

2 Bob 接了

Alice 拿奖金

Report the conflict

接龙

接龙冲突
请重新接龙

Lamport clock is insufficient, we need partially ordered timestamps

Total order: for every pair of events, we can compare their time

- I.e., either `T(e1) < T(e2)` or `T(e1) > T(e2)`

Problem:

- The order may disagree with the external observer, especially for unrelated events

Partial order. Consider two events e1 and e2 — three outcomes should be allowed

- T(e1) < T(e2)
- T(e2) < T(e1)
- Incomparable

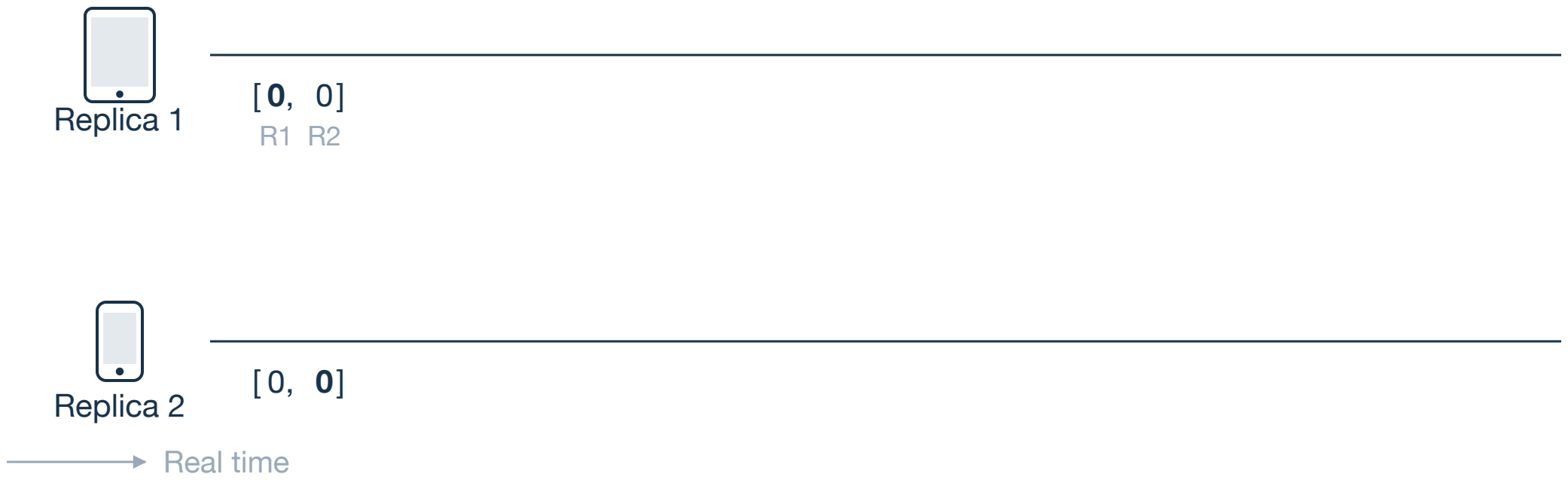
Here < means "happens before"

Realizing partial order: vector clock

Clocks are represented as a vector, one entry per server

- Its own entry is that server's local Lamport clock
- The other entries are cached copies of the clocks on other servers

Increment the local T as real time passes: only its own entry moves

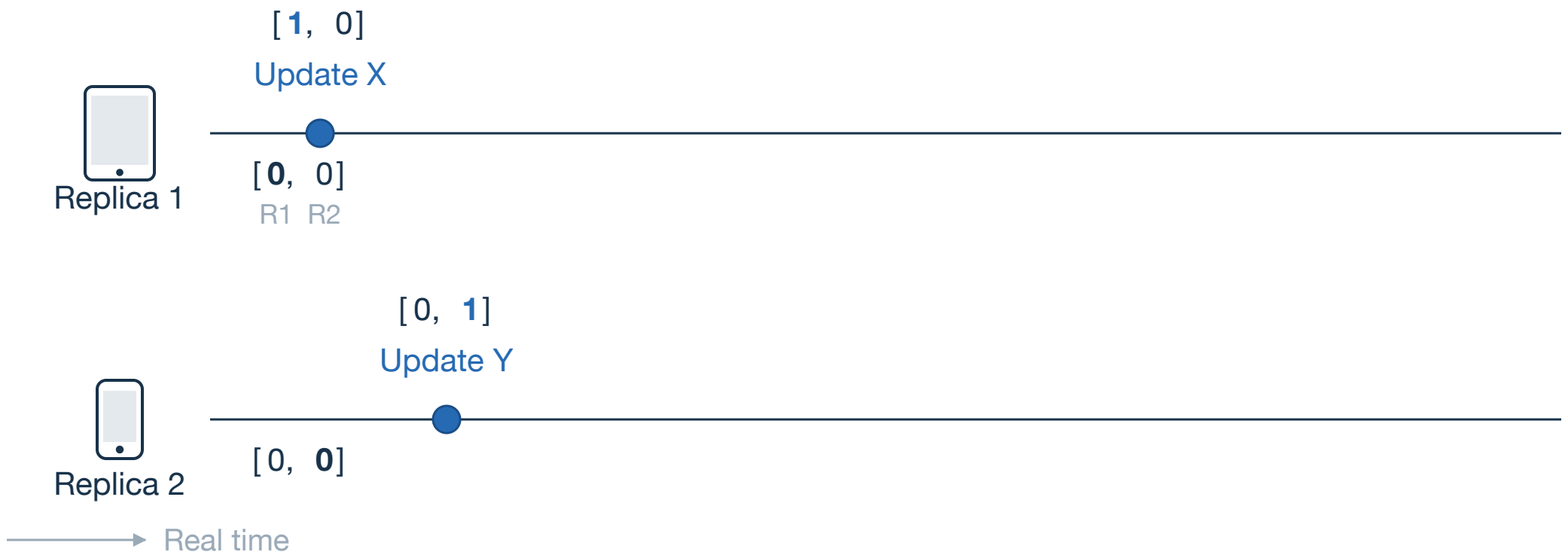


Realizing partial order: vector clock

Clocks are represented as a vector, one entry per server

- Its own entry is that server's local Lamport clock
- The other entries are cached copies of the clocks on other servers

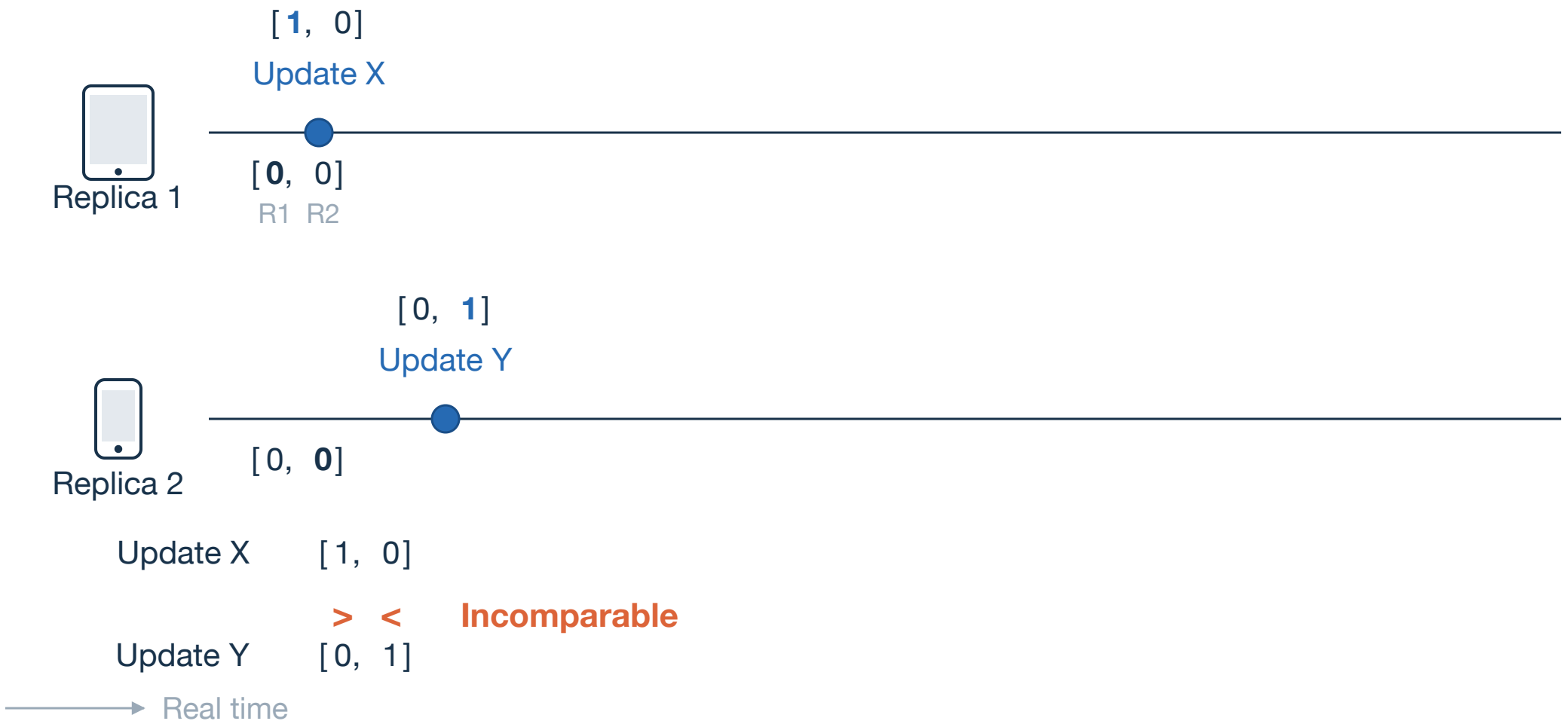
Increment the local T as real time passes: only its own entry moves



Two independent events: incomparable with a vector clock

R1 issues update X with [1, 0], R2 issues update Y with [0, 1]

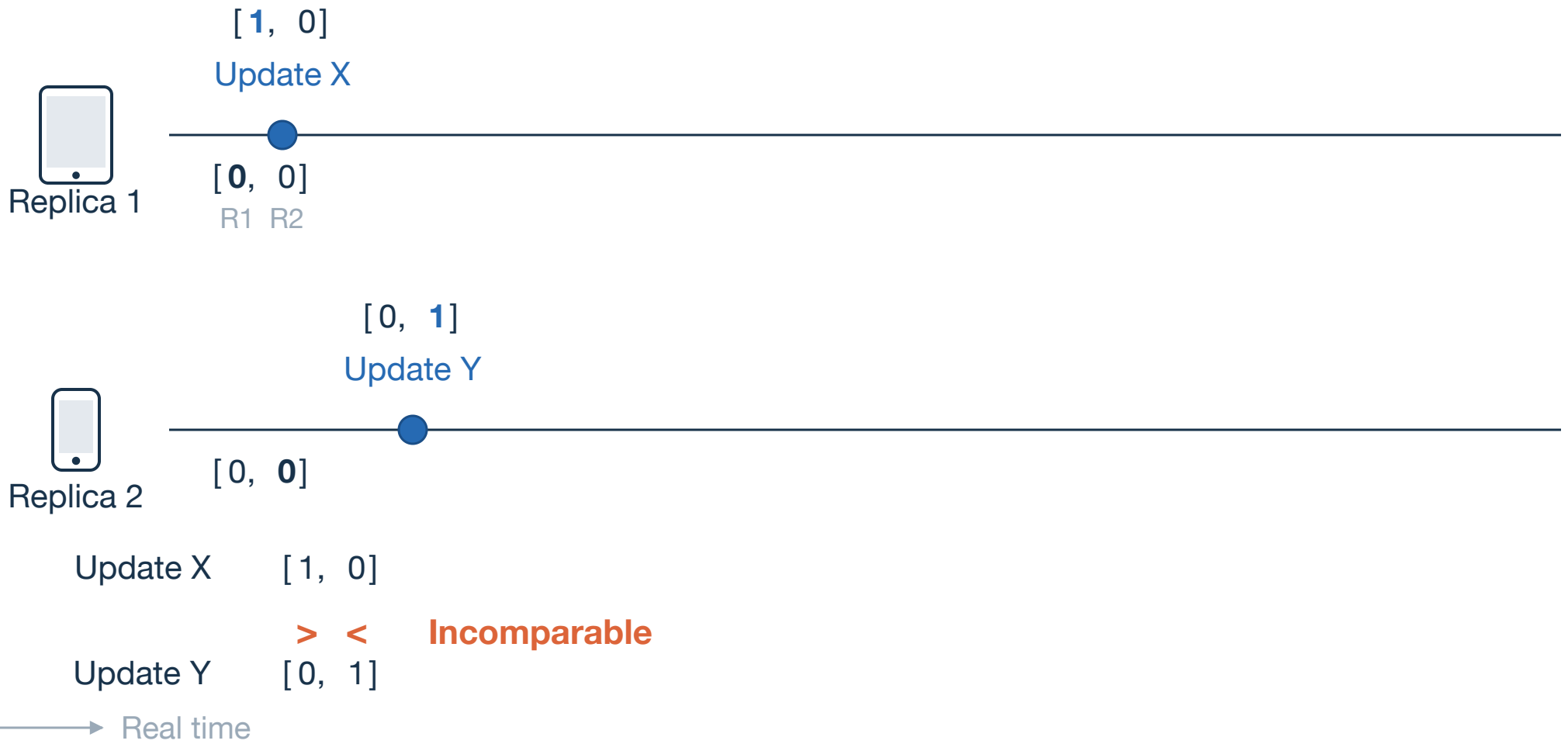
- 1 > 0, but 0 < 1 — neither vector dominates the other



Take the max of every entry to preserve causality

Modify T = max(T, T′) entry by entry if it sees T′ from another server

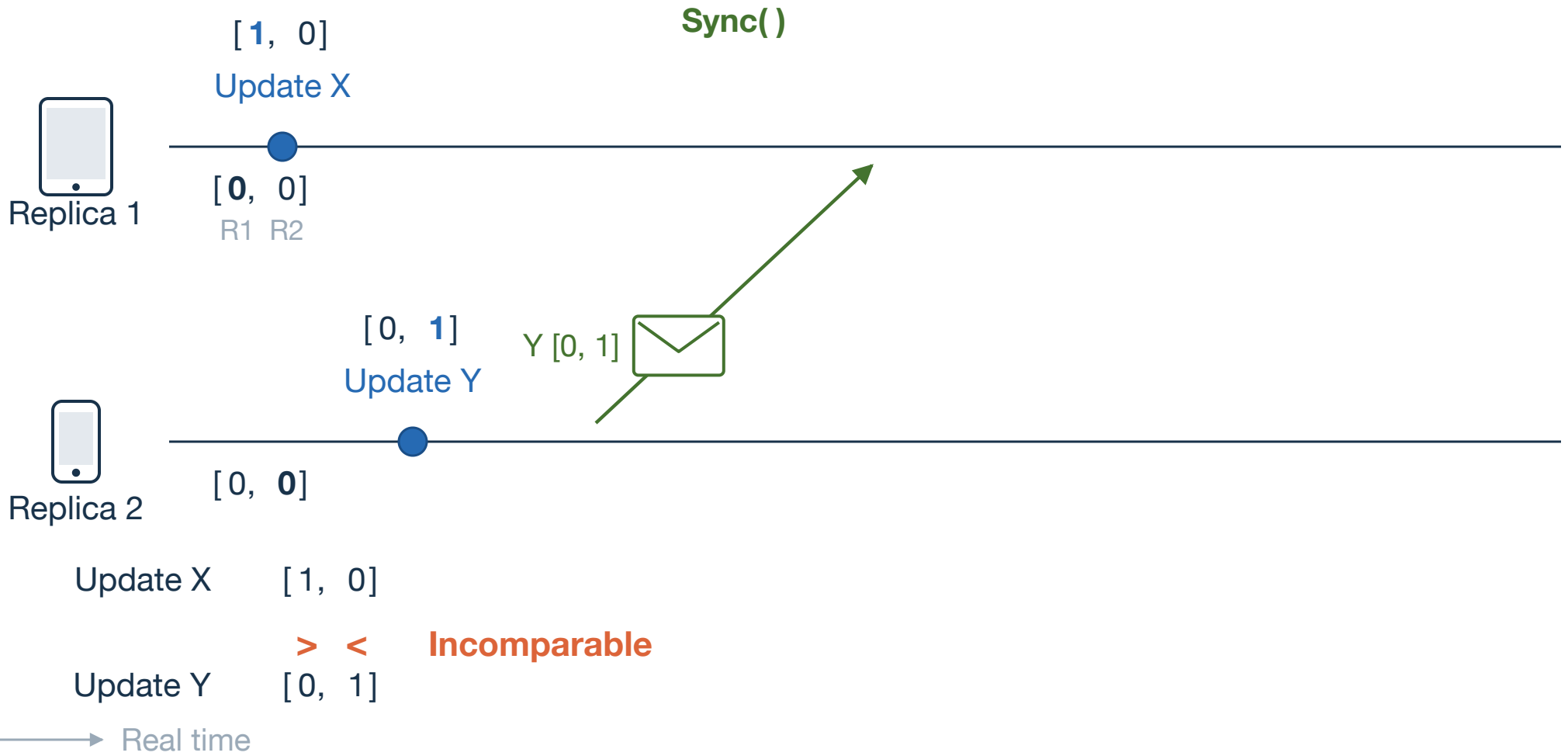
- Then increment its own entry, as for any other event



Take the max of every entry to preserve causality

Modify T = max(T, T′) entry by entry if it sees T′ from another server

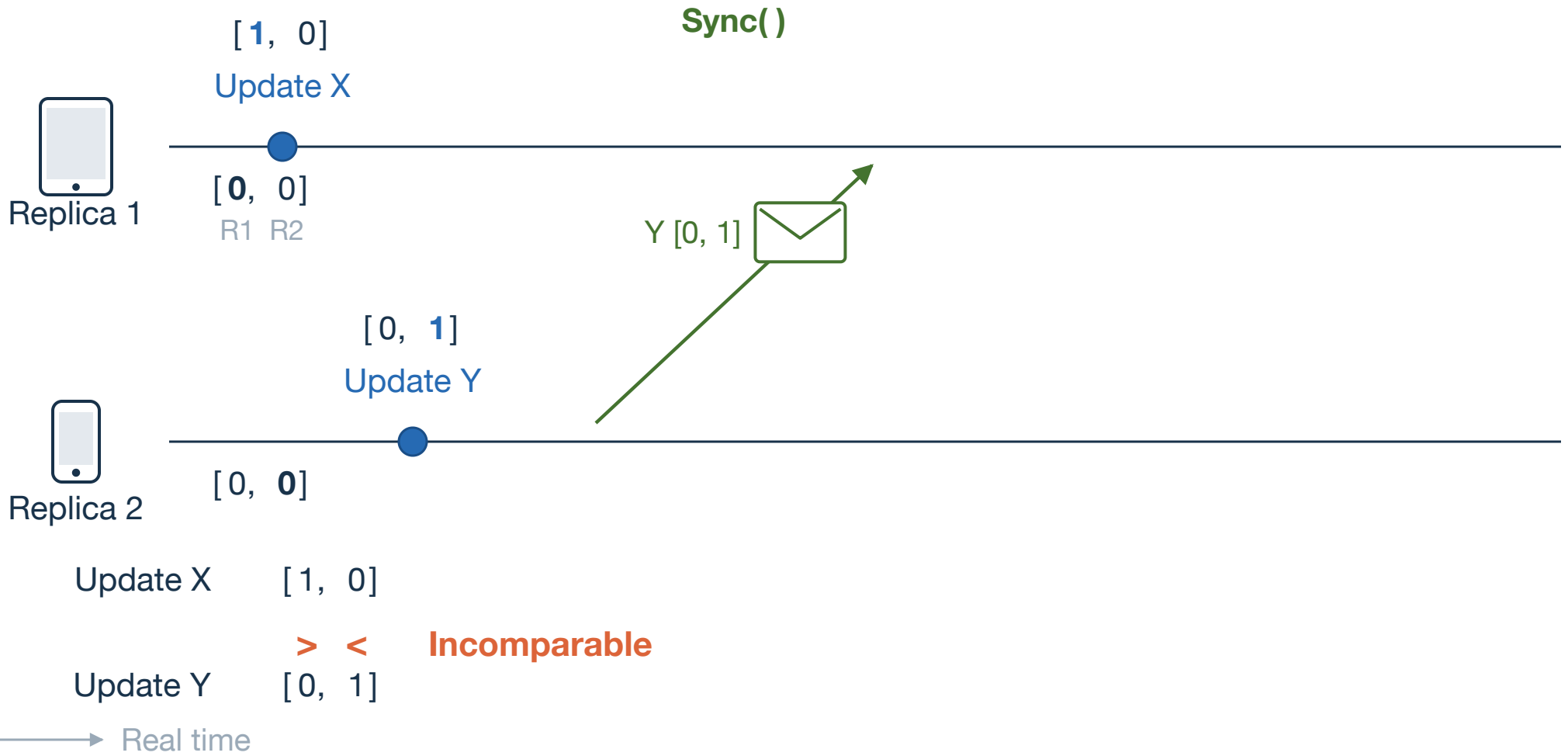
- Then increment its own entry, as for any other event



Take the max of every entry to preserve causality

Modify T = max(T, T′) entry by entry if it sees T′ from another server

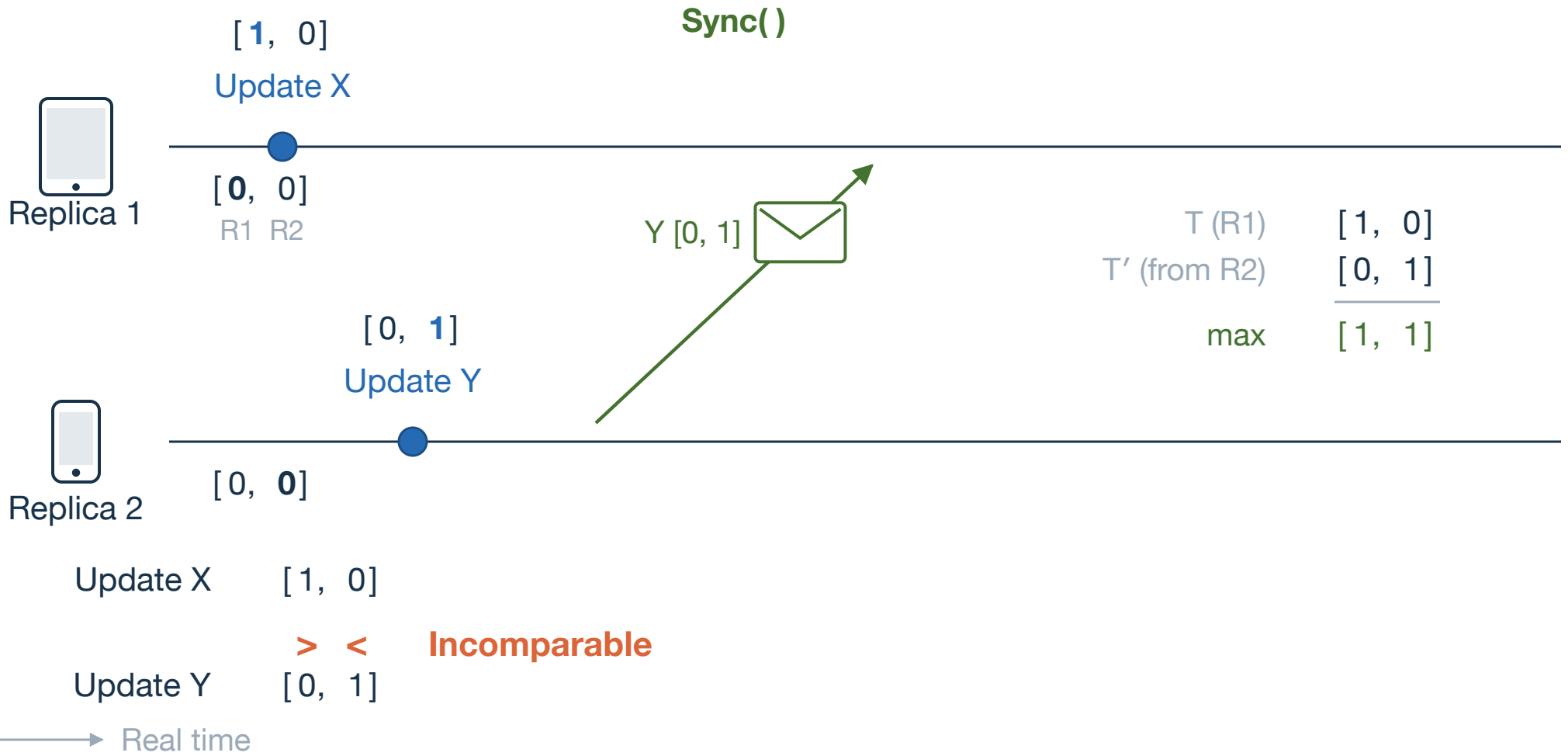
- Then increment its own entry, as for any other event



Take the max of every entry to preserve causality

Modify T = max(T, T′) entry by entry if it sees T′ from another server

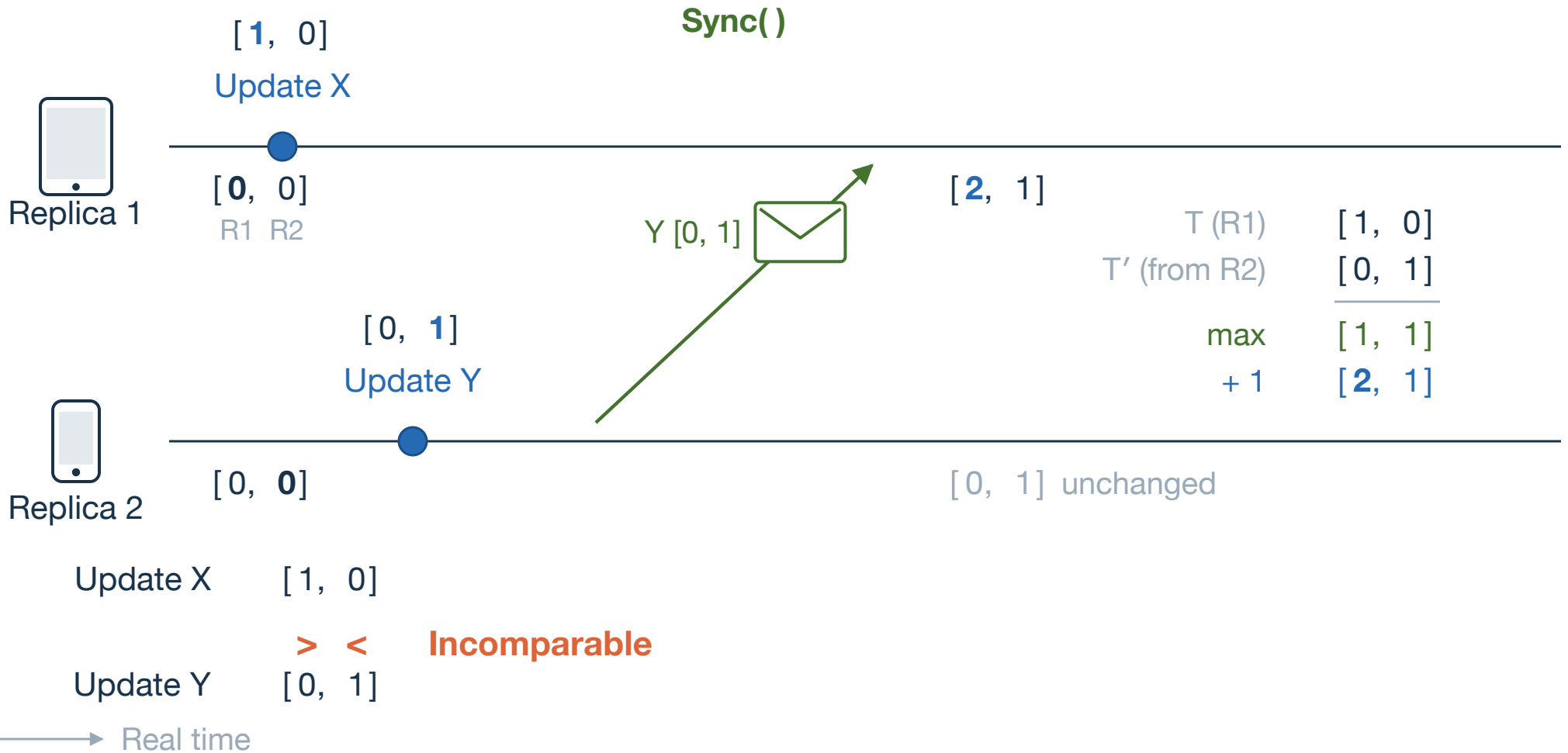
- Then increment its own entry, as for any other event



Take the max of every entry to preserve causality

Modify T = max(T, T′) entry by entry if it sees T′ from another server

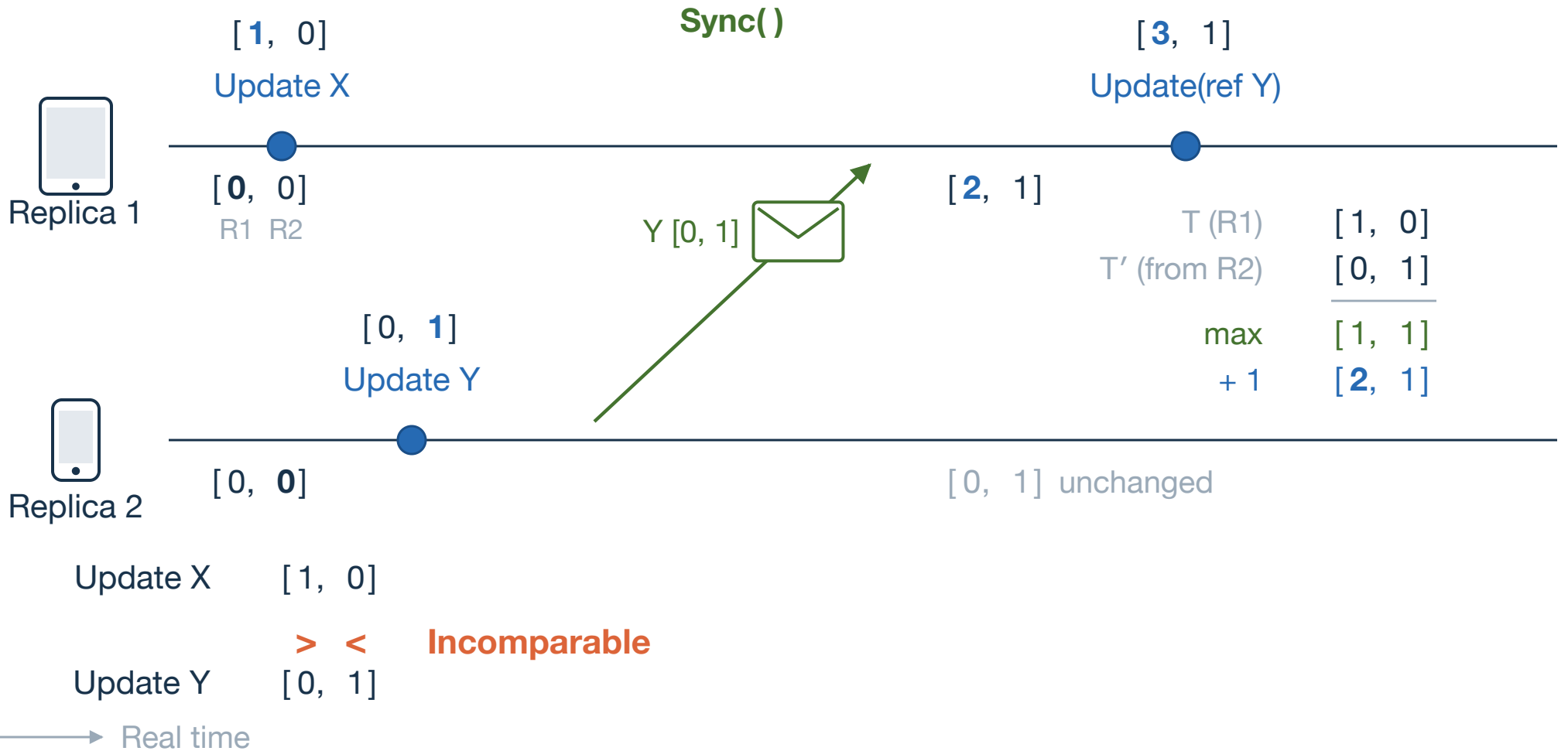
- Then increment its own entry, as for any other event



Take the max of every entry to preserve causality

Modify T = max(T, T′) entry by entry if it sees T′ from another server

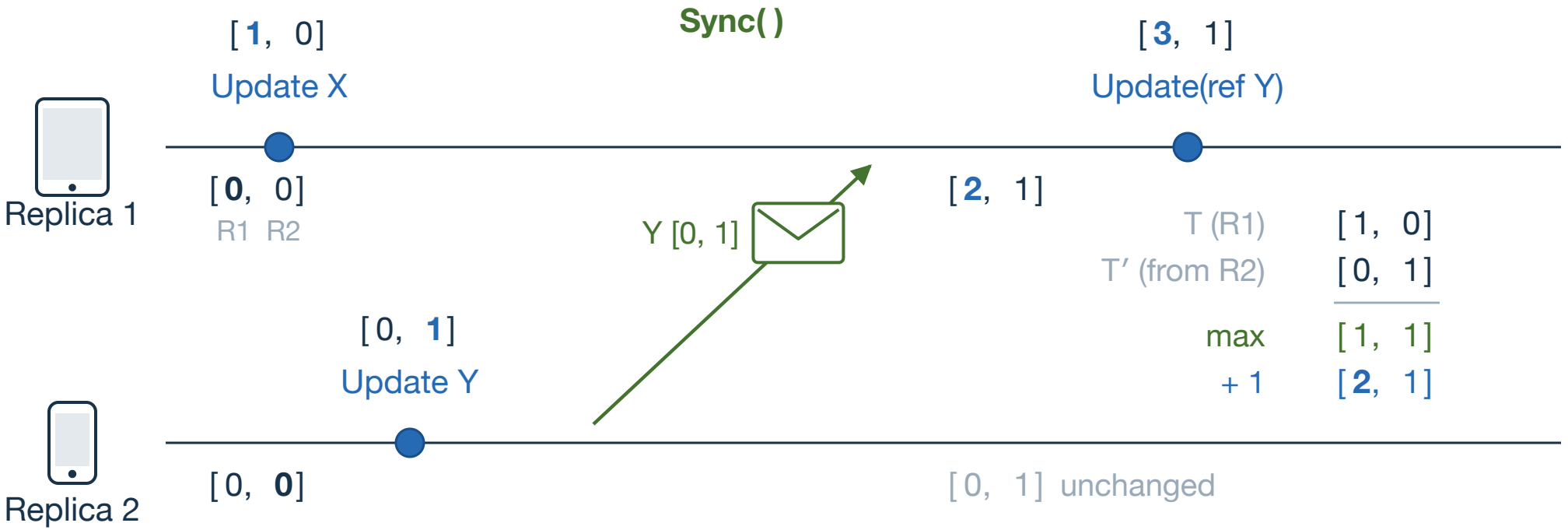
- Then increment its own entry, as for any other event



Take the max of every entry to preserve causality

Modify T = max(T, T′) entry by entry if it sees T′ from another server

- Then increment its own entry, as for any other event

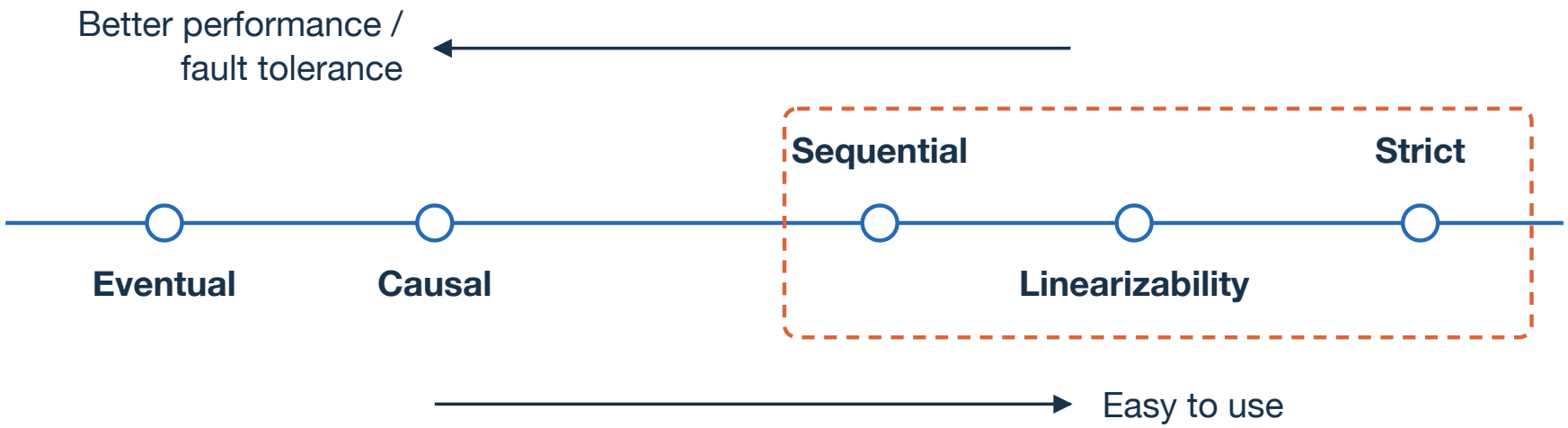


Back to 接龙: two paths towards fixing the issue

Path #1: what cannot be ordered is handed to the application to order

- Drawback: requires application modifications — every app on top has to handle conflicts itself
- Also, the first one can lose if there are concurrent events!

Path #2: choose a consistency model that cannot expose reordered events



What does a strong consistency model present?

From a developer's perspective

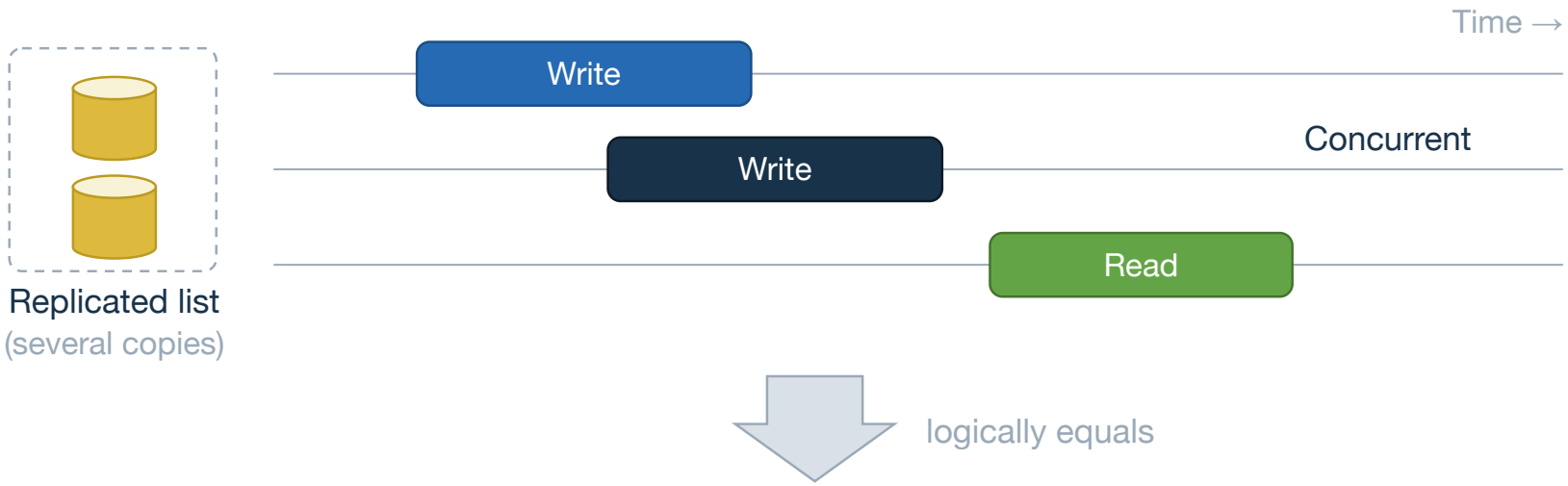
- Everything has one copy (logically), like a centralized design
- The overall operations executed by the system correspond to a serial execution (logically)



What does a strong consistency model present?

From a developer's perspective

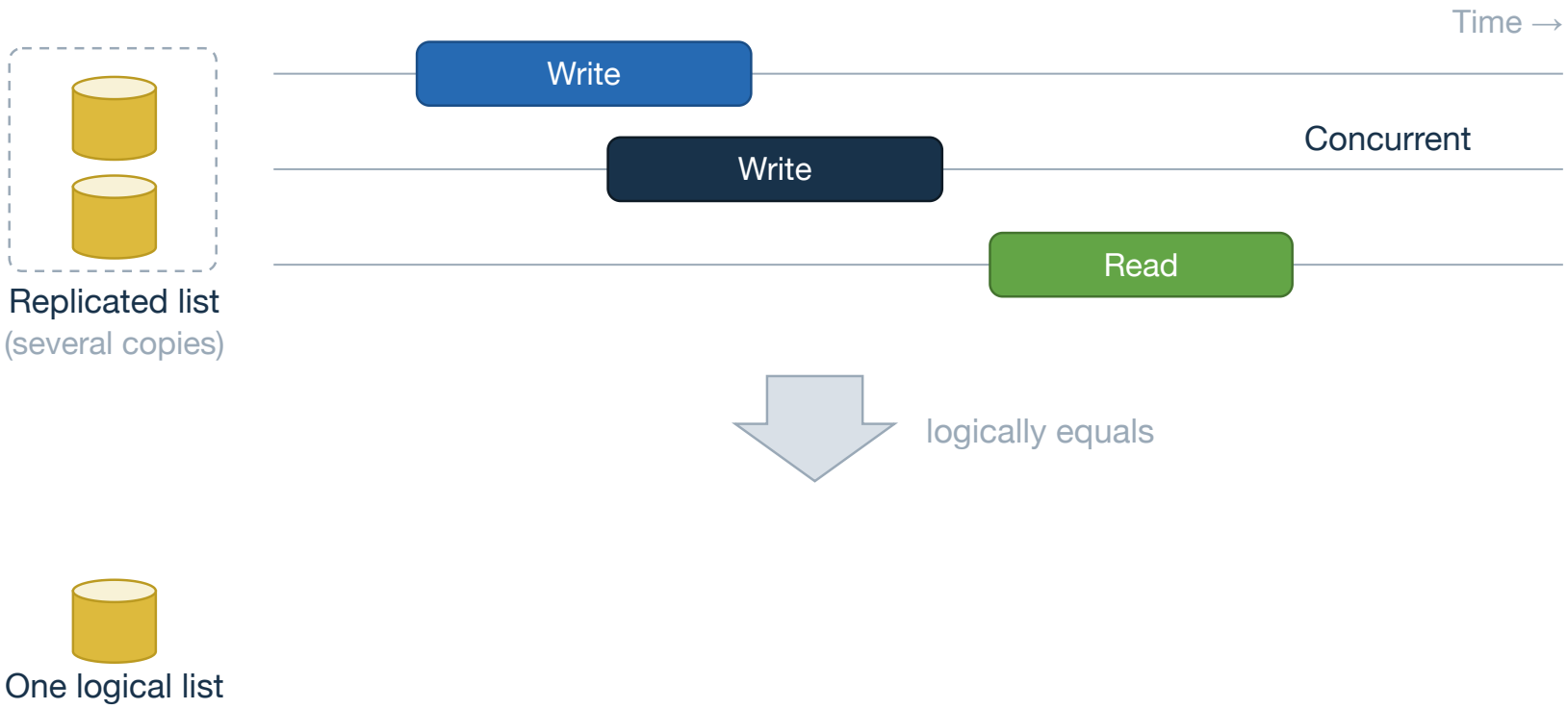
- Everything has one copy (logically), like a centralized design
- The overall operations executed by the system correspond to a serial execution (logically)



What does a strong consistency model present?

From a developer's perspective

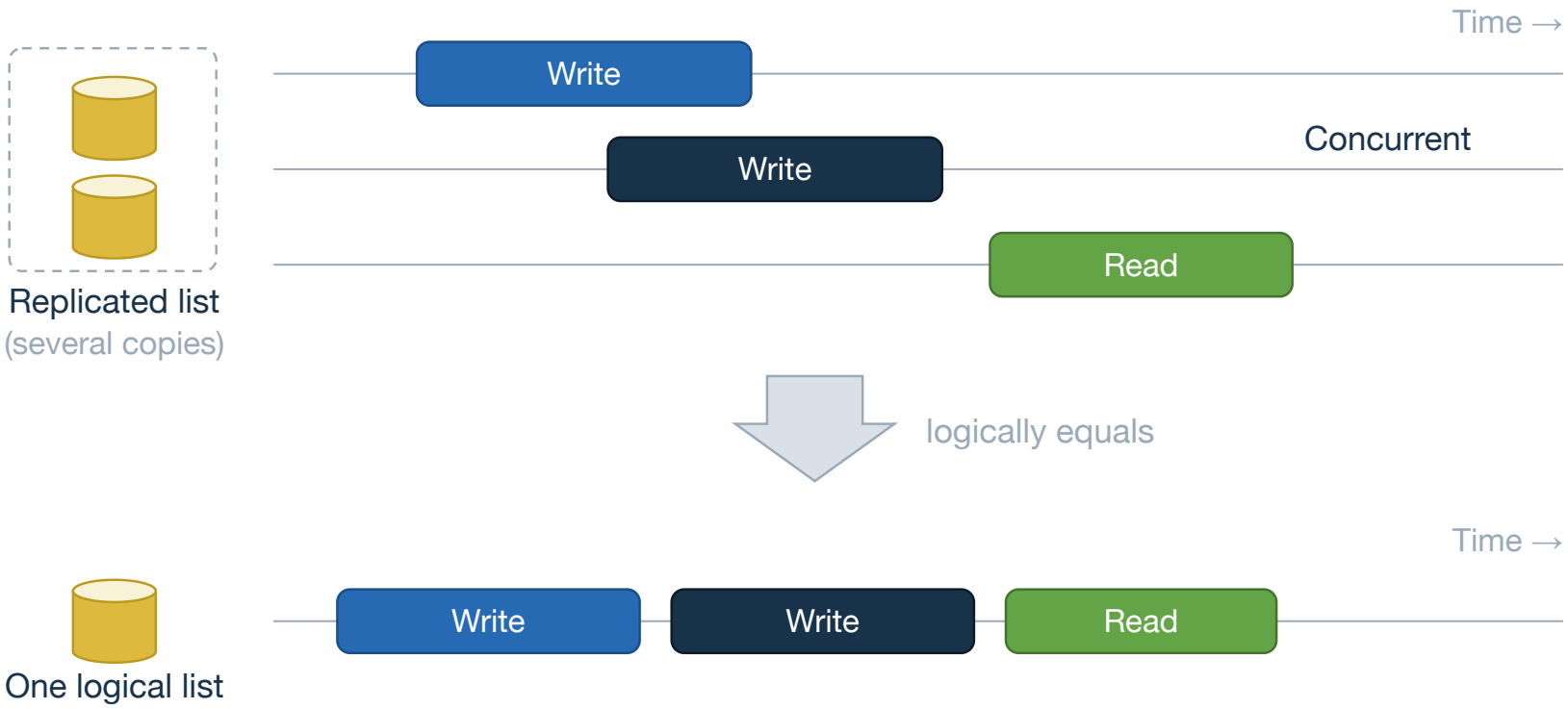
- Everything has one copy (logically), like a centralized design
- The overall operations executed by the system correspond to a serial execution (logically)



What does a strong consistency model present?

From a developer's perspective

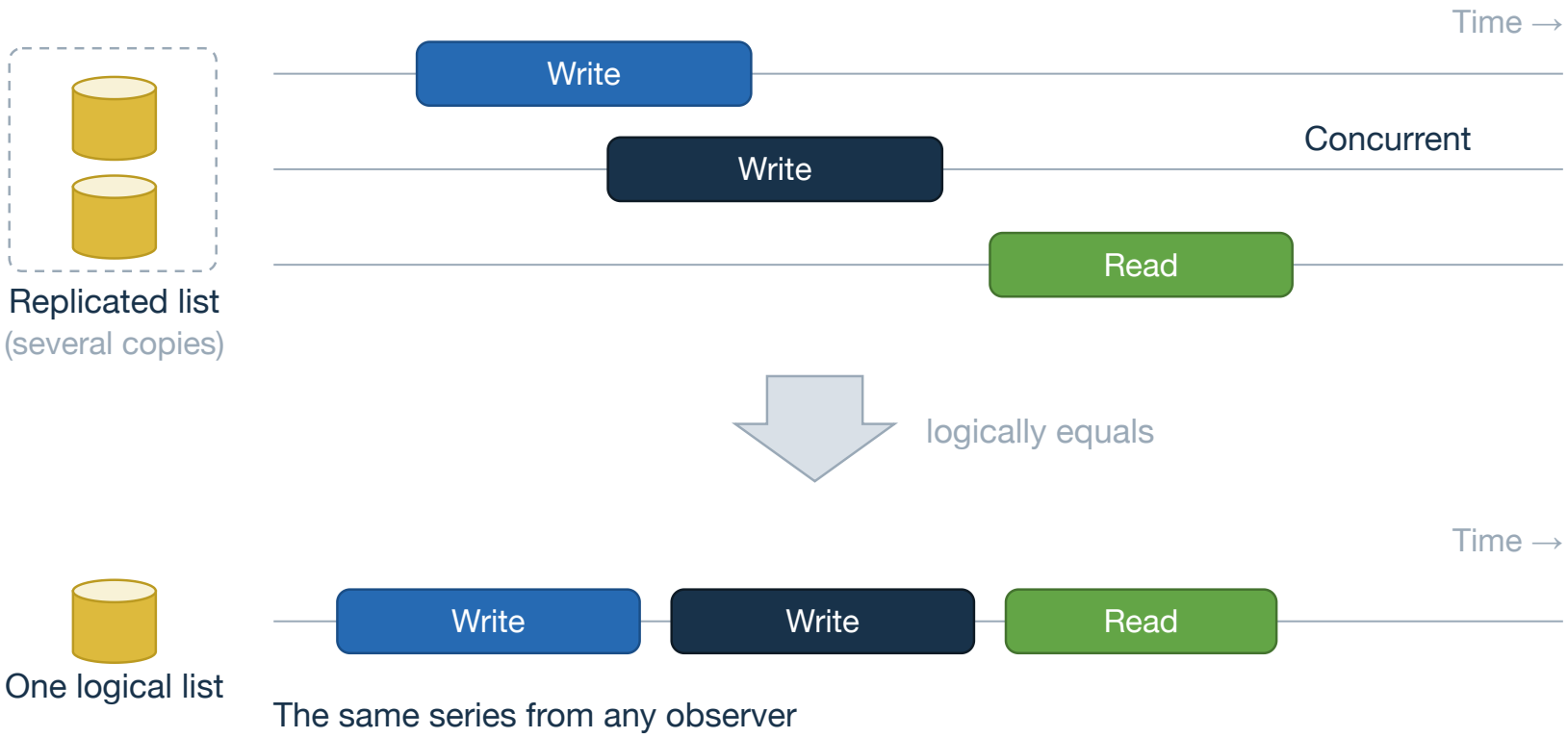
- Everything has one copy (logically), like a centralized design
- The overall operations executed by the system correspond to a serial execution (logically)



What does a strong consistency model present?

From a developer's perspective

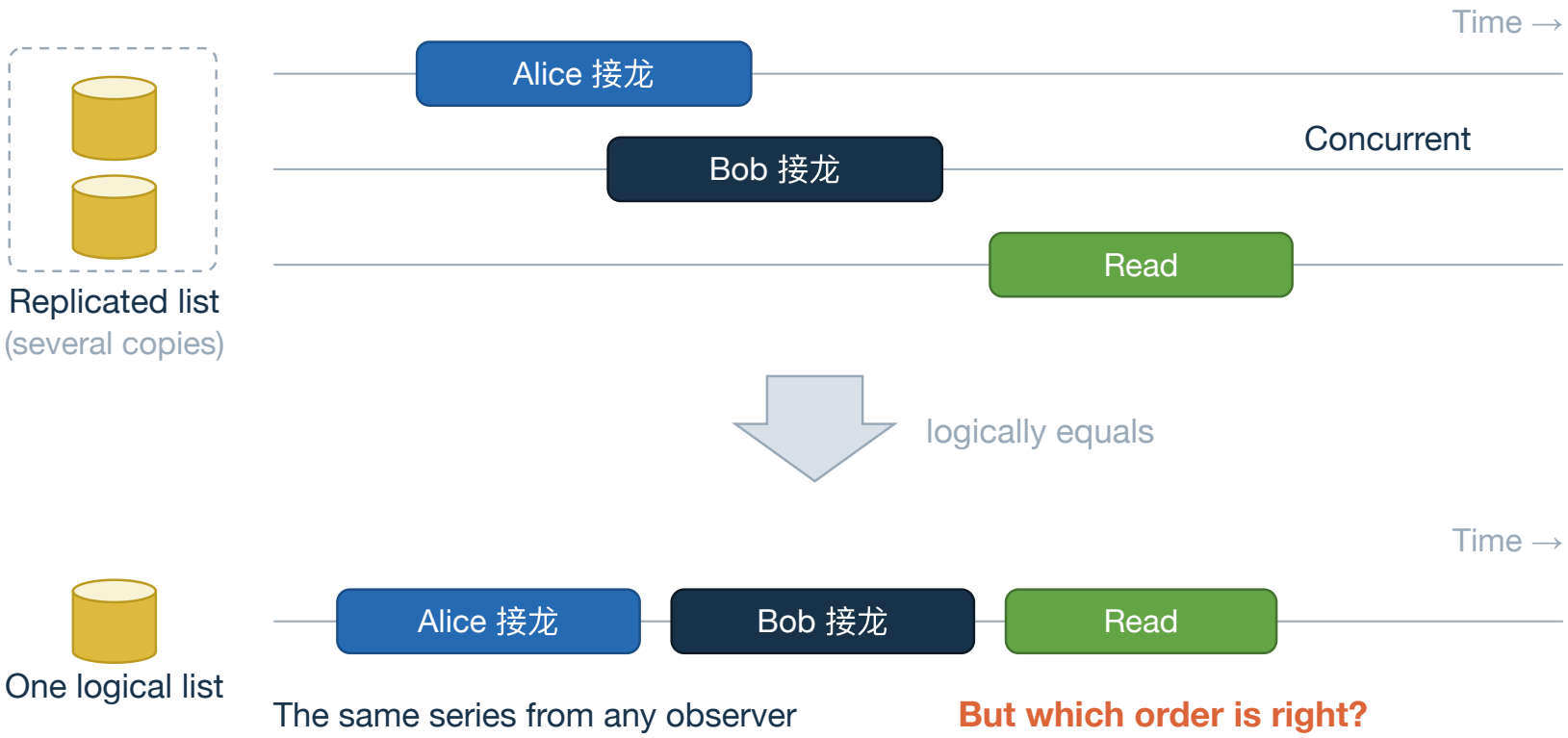
- Everything has one copy (logically), like a centralized design
- The overall operations executed by the system correspond to a serial execution (logically)



Question: which order is the equivalent order?

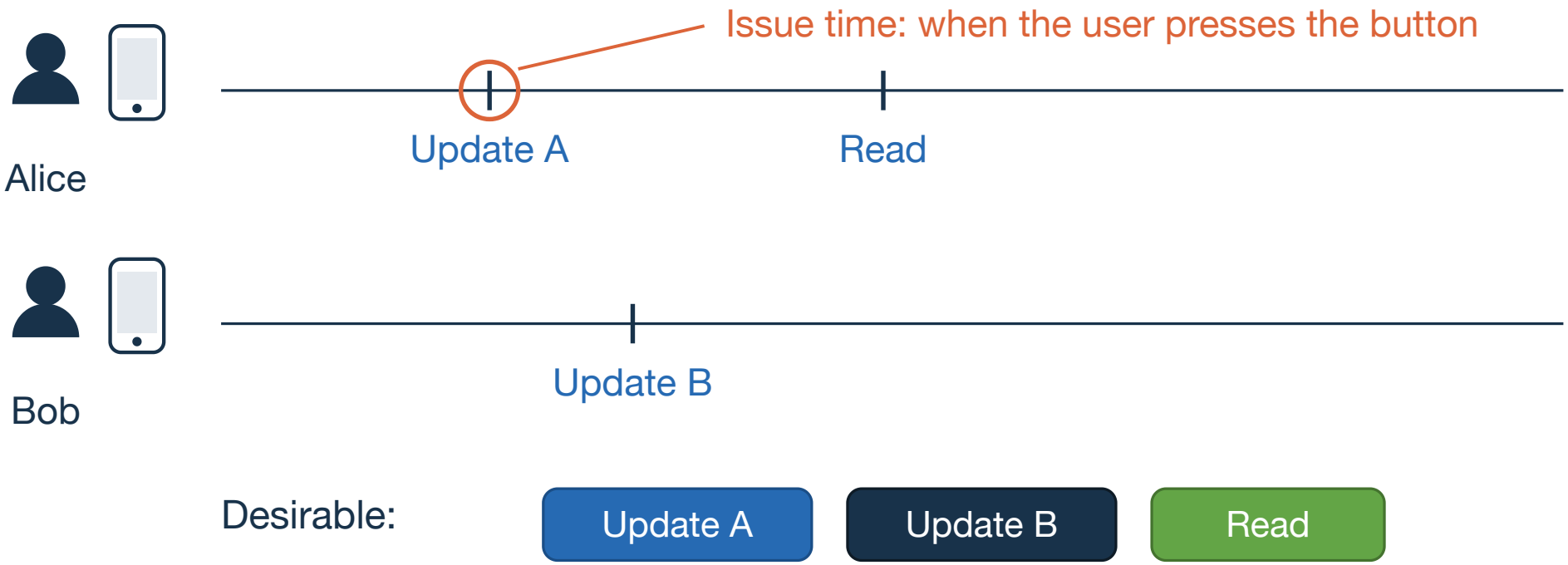
A different order observed means a different user experience

- E.g., if Alice updates first, it should appear earlier in the list



The desired order for 接龙: ordered by the issue time

I.e., operations are ordered by the issue time of the operation



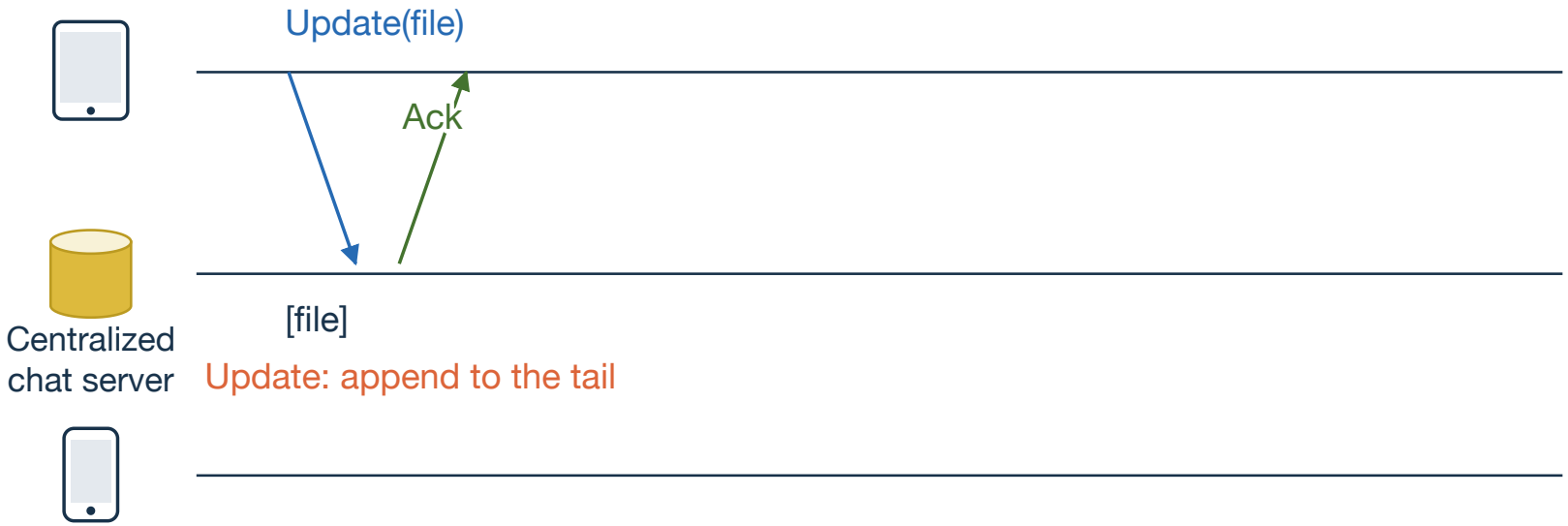
Ordering by the issue time is called strict consistency

Question: how to implement a strong consistency model?

Implementation goals (e.g., for strict consistency)

- Everything has one copy (in a logical view)
- The overall operations correspond to a serial execution, and the order matches the issue time

One possible approach: a centralized design

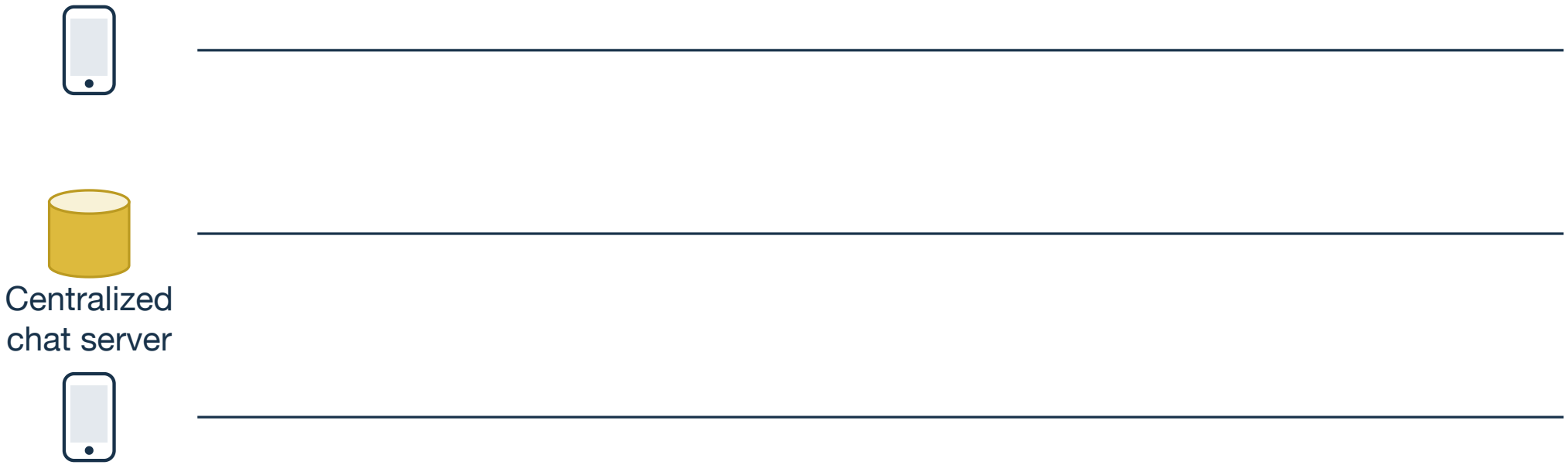


Question: can we implement strict consistency?

Assume we implement strict consistency using a centralized KVS

- The simplest setup we can get in a distributed setting

Question: how can the server determine to process Update A first?

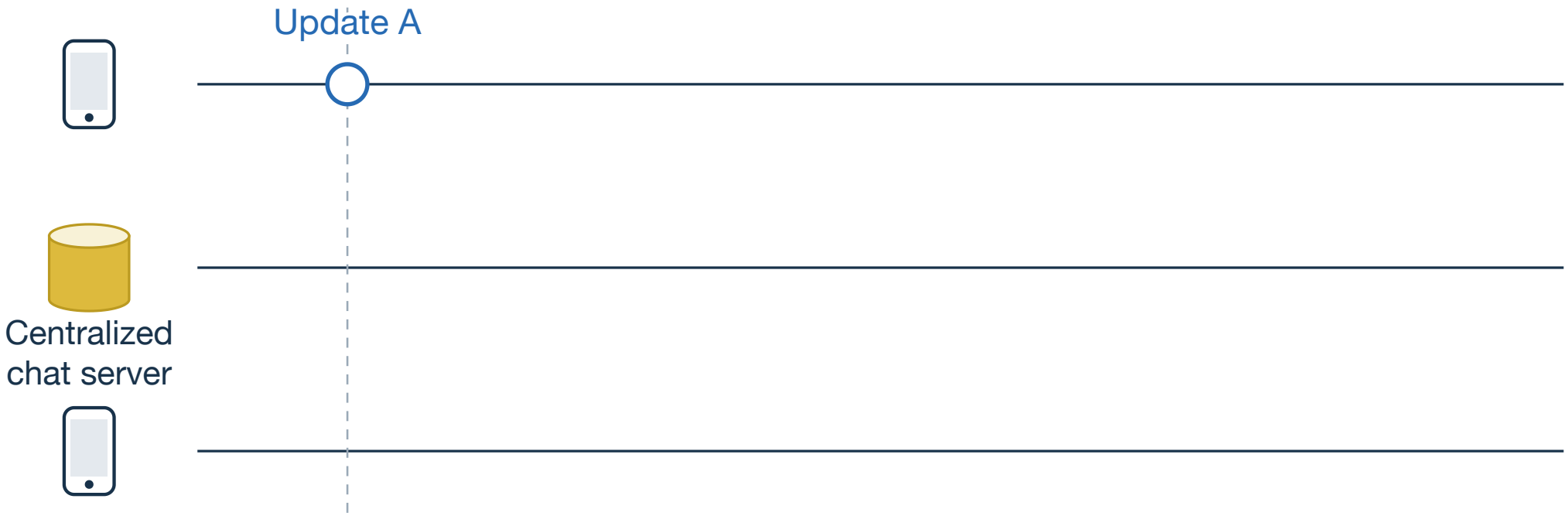


Question: can we implement strict consistency?

Assume we implement strict consistency using a centralized KVS

- The simplest setup we can get in a distributed setting

Question: how can the server determine to process Update A first?

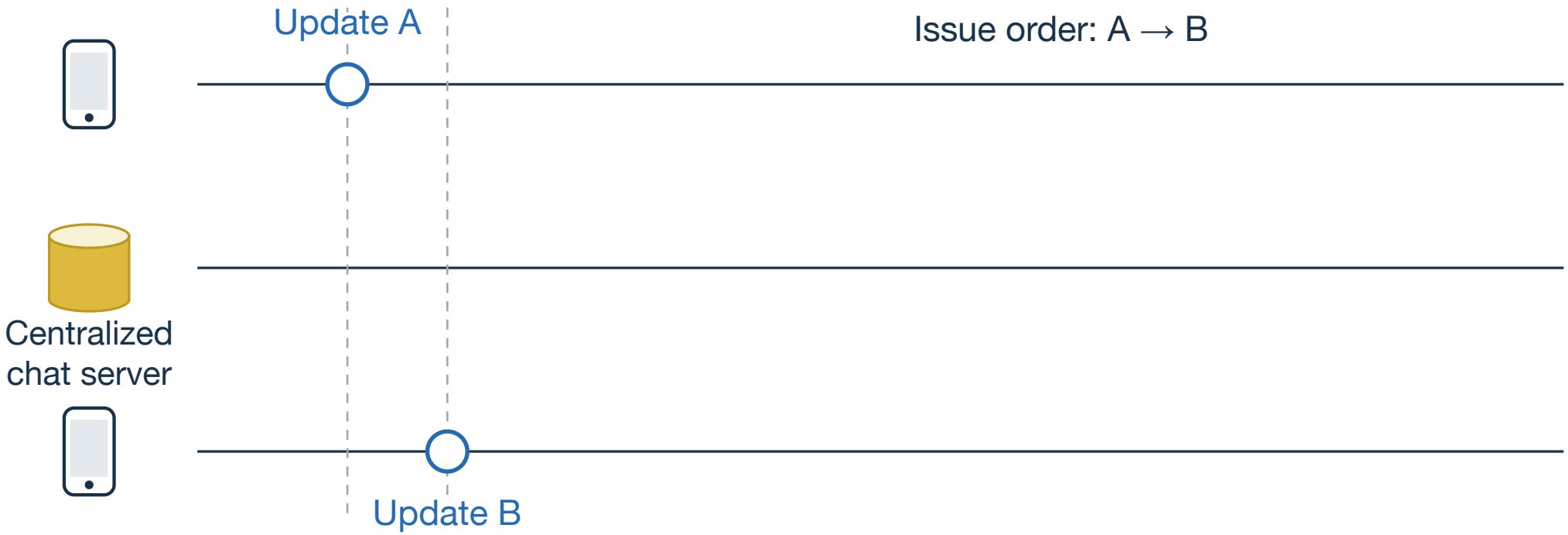


Question: can we implement strict consistency?

Assume we implement strict consistency using a centralized KVS

- The simplest setup we can get in a distributed setting

Question: how can the server determine to process Update A first?

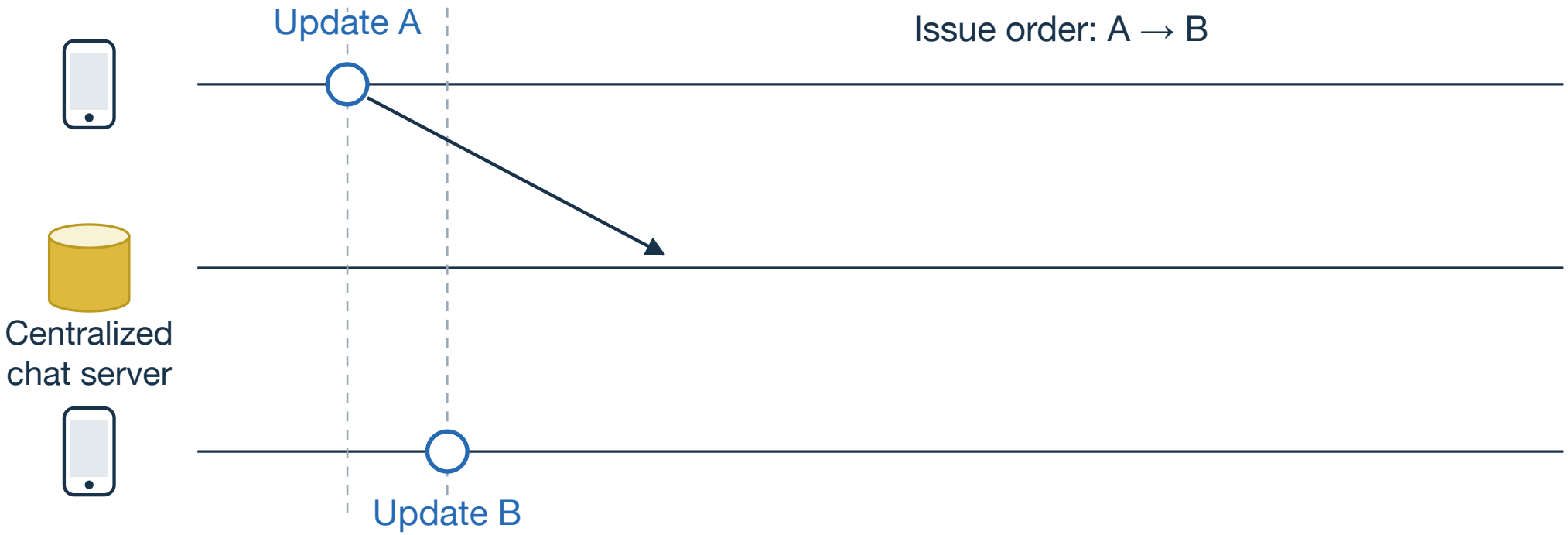


Question: can we implement strict consistency?

Assume we implement strict consistency using a centralized KVS

- The simplest setup we can get in a distributed setting

Question: how can the server determine to process Update A first?

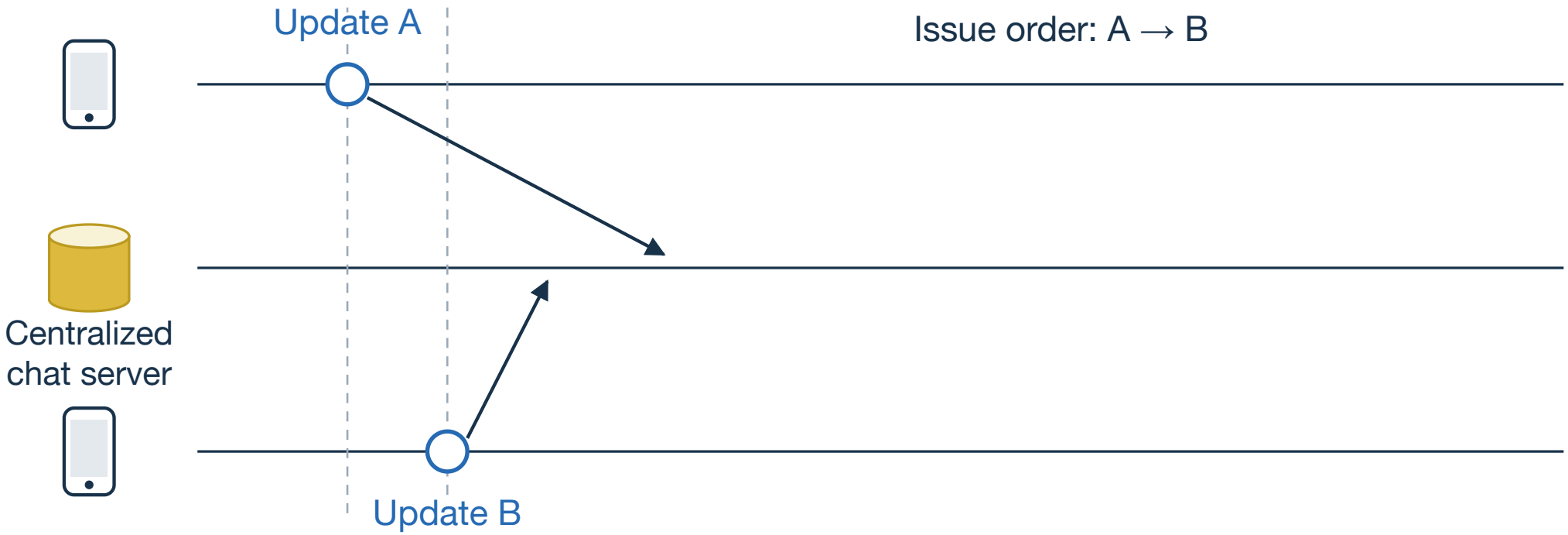


Question: can we implement strict consistency?

Assume we implement strict consistency using a centralized KVS

- The simplest setup we can get in a distributed setting

Question: how can the server determine to process Update A first?

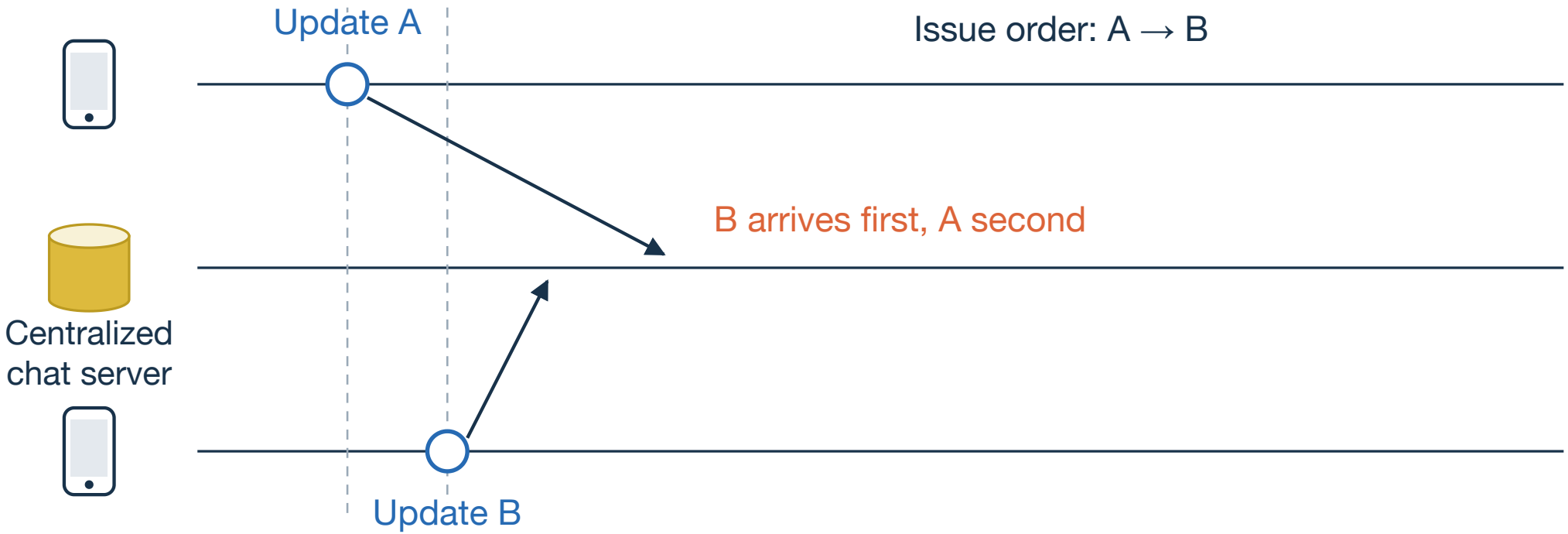


Question: can we implement strict consistency?

Assume we implement strict consistency using a centralized KVS

- The simplest setup we can get in a distributed setting

Question: how can the server determine to process Update A first?

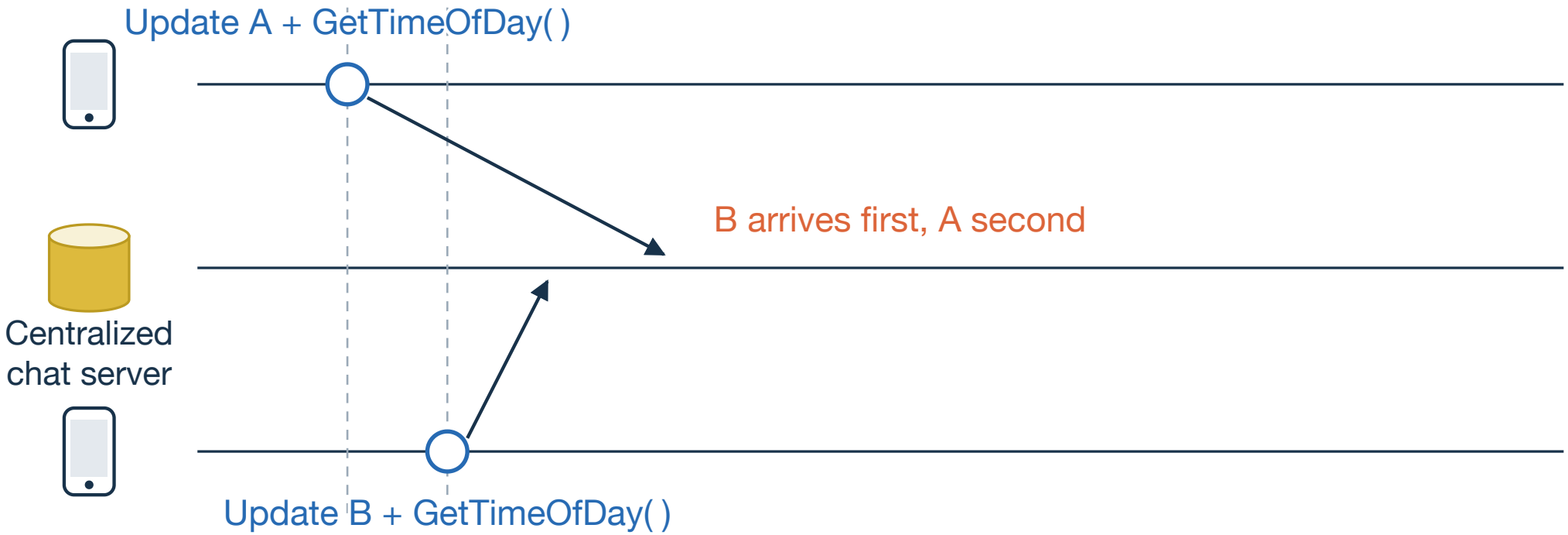


Challenge #1: unsynchronized clocks for the issue time

Without information, the server cannot order the operations

Question: can we add information for the server to order operations?

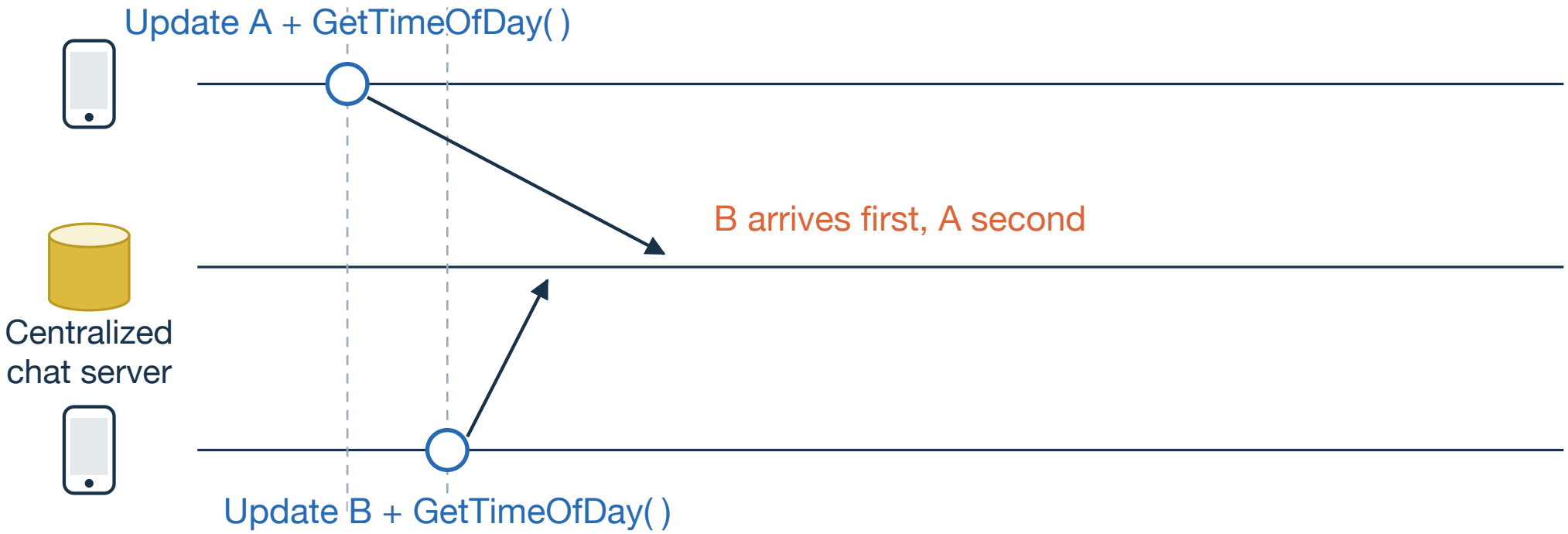
- E.g., using the machine's timestamp at the issue time



Challenge #2: unbounded wait

Even with synchronized clocks, is our implementation efficient?

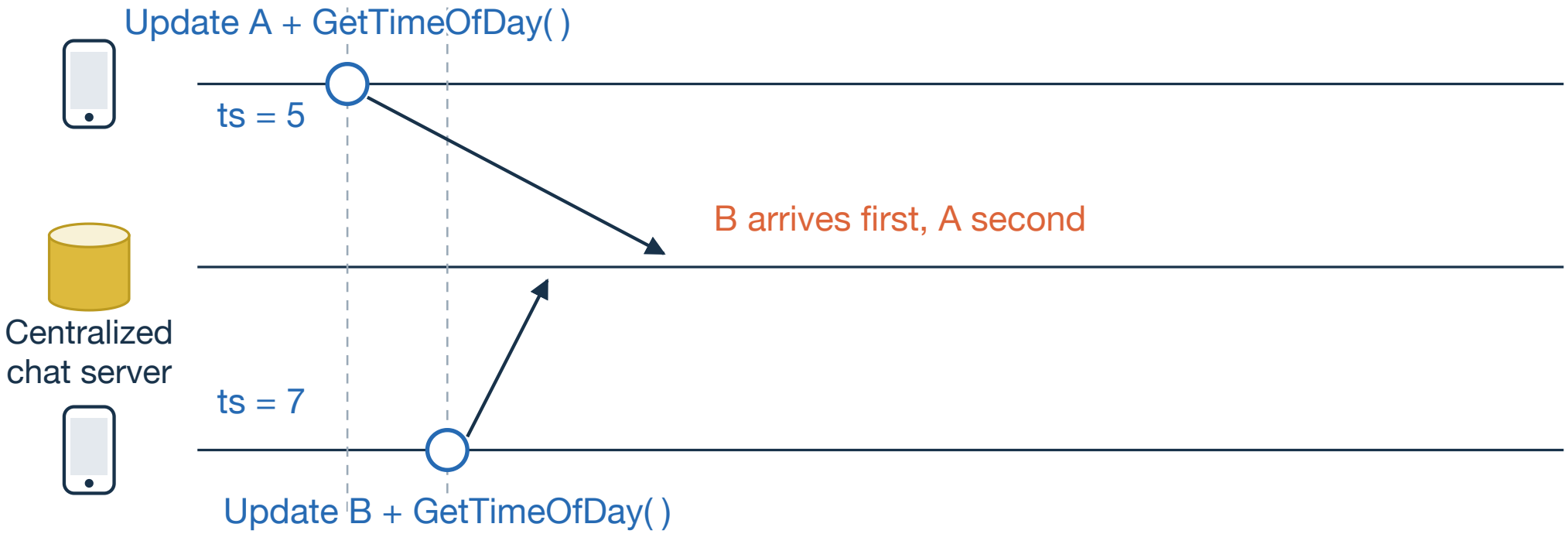
- No, because the server needs to wait for out-of-order requests
- And we don't even know how many to wait for (what about packet loss?)



Challenge #2: unbounded wait

Even with synchronized clocks, is our implementation efficient?

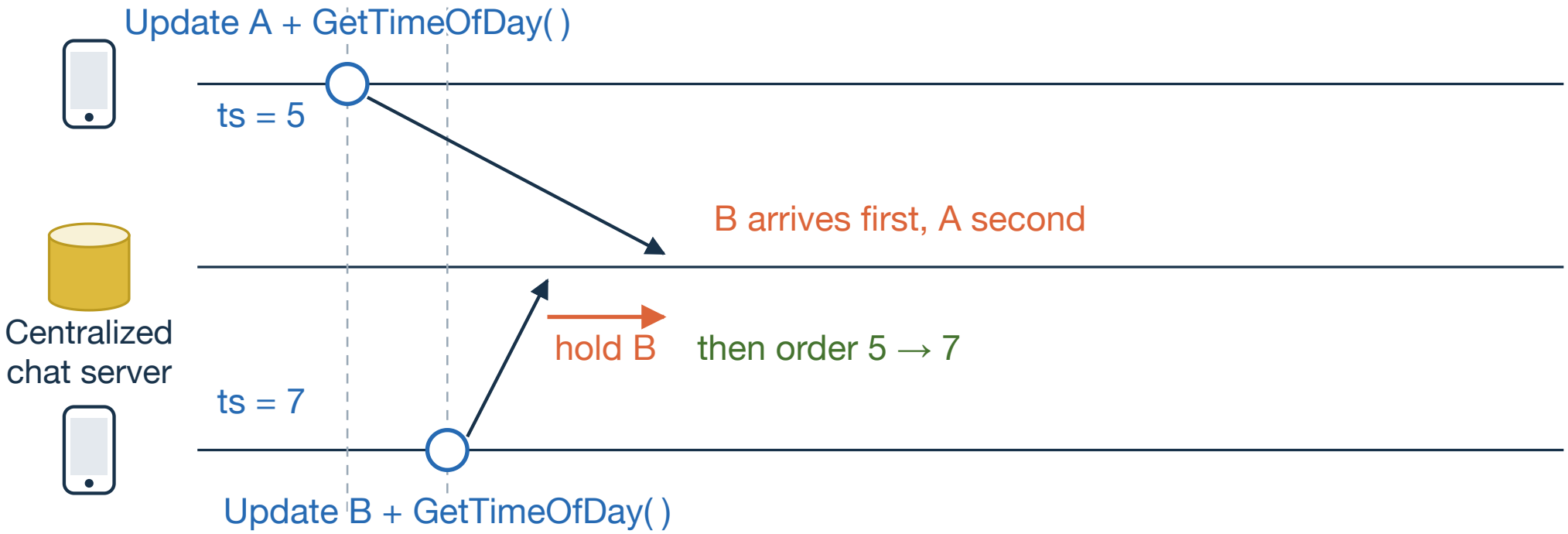
- No, because the server needs to wait for out-of-order requests
- And we don't even know how many to wait for (what about packet loss?)



Challenge #2: unbounded wait

Even with synchronized clocks, is our implementation efficient?

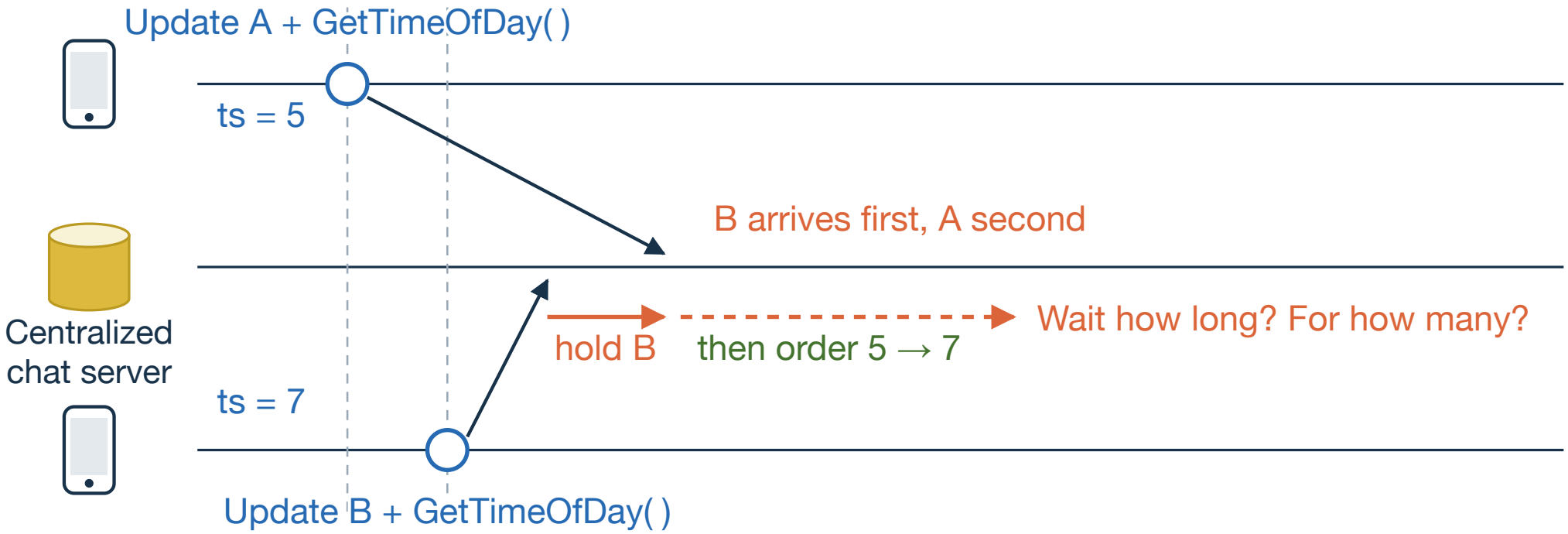
- No, because the server needs to wait for out-of-order requests
- And we don't even know how many to wait for (what about packet loss?)



Challenge #2: unbounded wait

Even with synchronized clocks, is our implementation efficient?

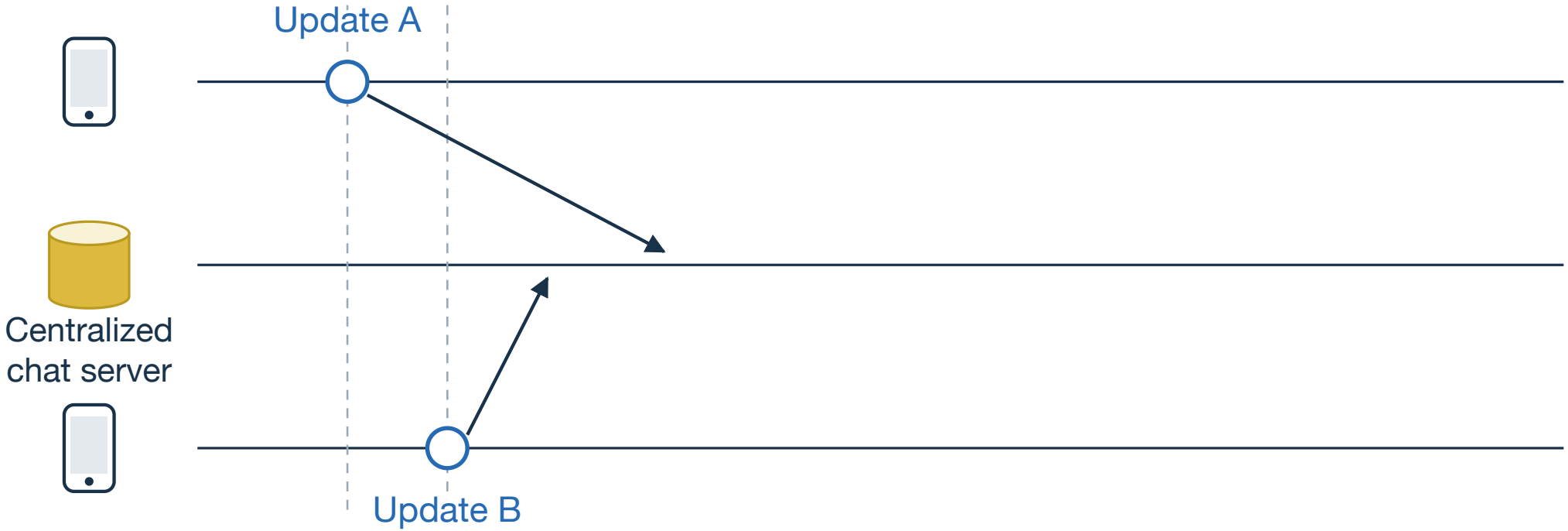
- No, because the server needs to wait for out-of-order requests
- And we don't even know how many to wait for (what about packet loss?)



Strict consistency: strong but not practical

What is the order maintained by our centralized design?

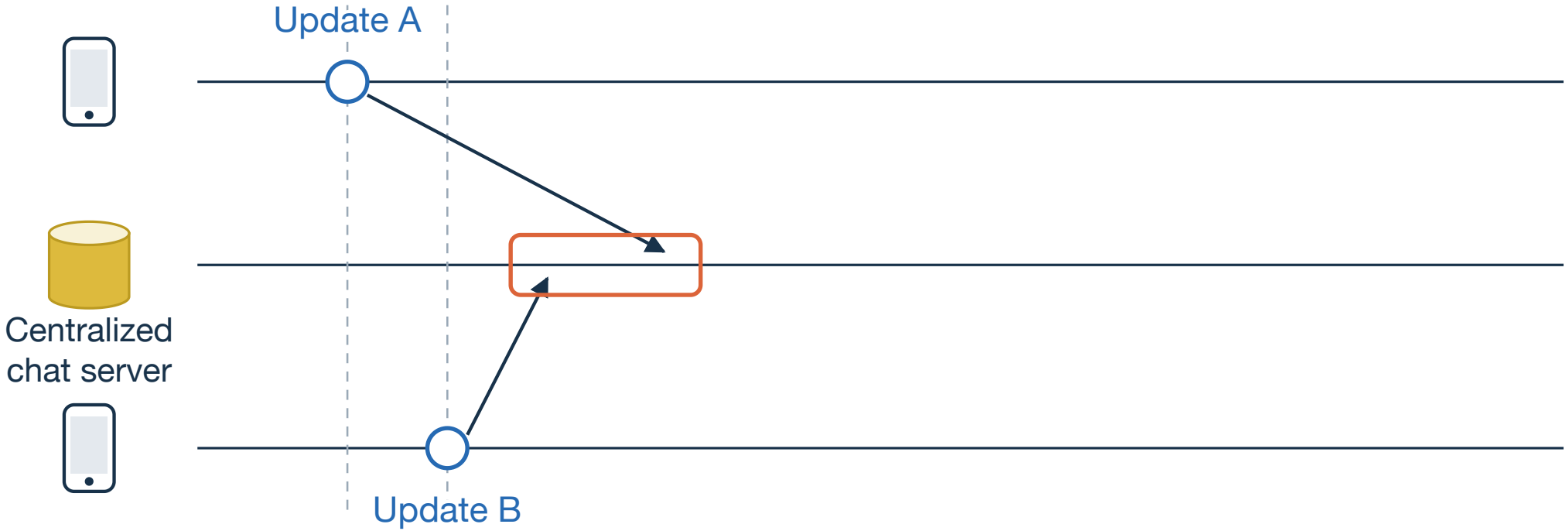
- The order is determined by how the server processes the requests



Strict consistency: strong but not practical

What is the order maintained by our centralized design?

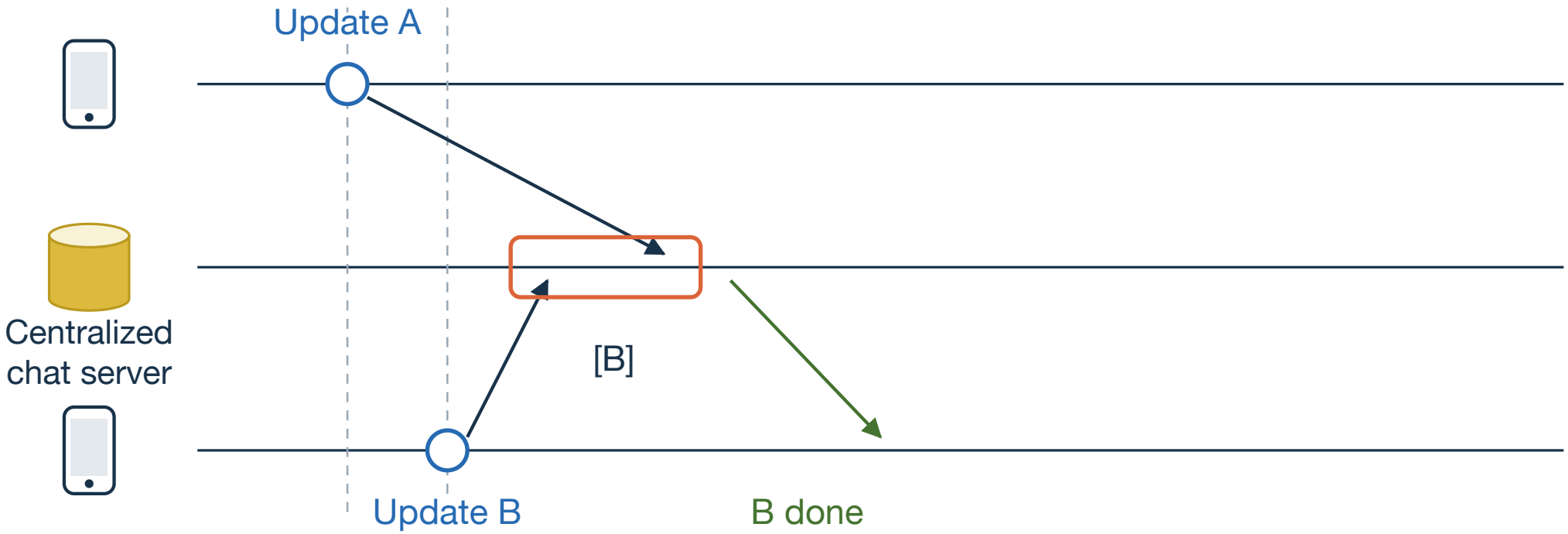
- The order is determined by how the server processes the requests



Strict consistency: strong but not practical

What is the order maintained by our centralized design?

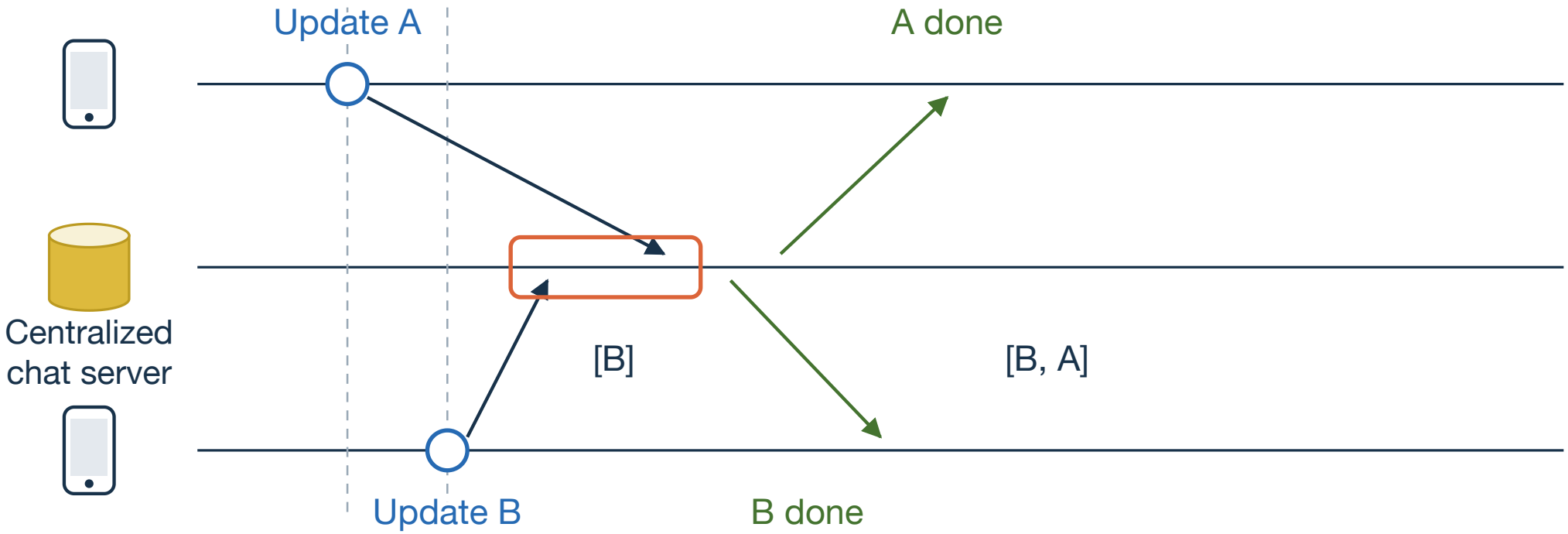
- The order is determined by how the server processes the requests



Strict consistency: strong but not practical

What is the order maintained by our centralized design?

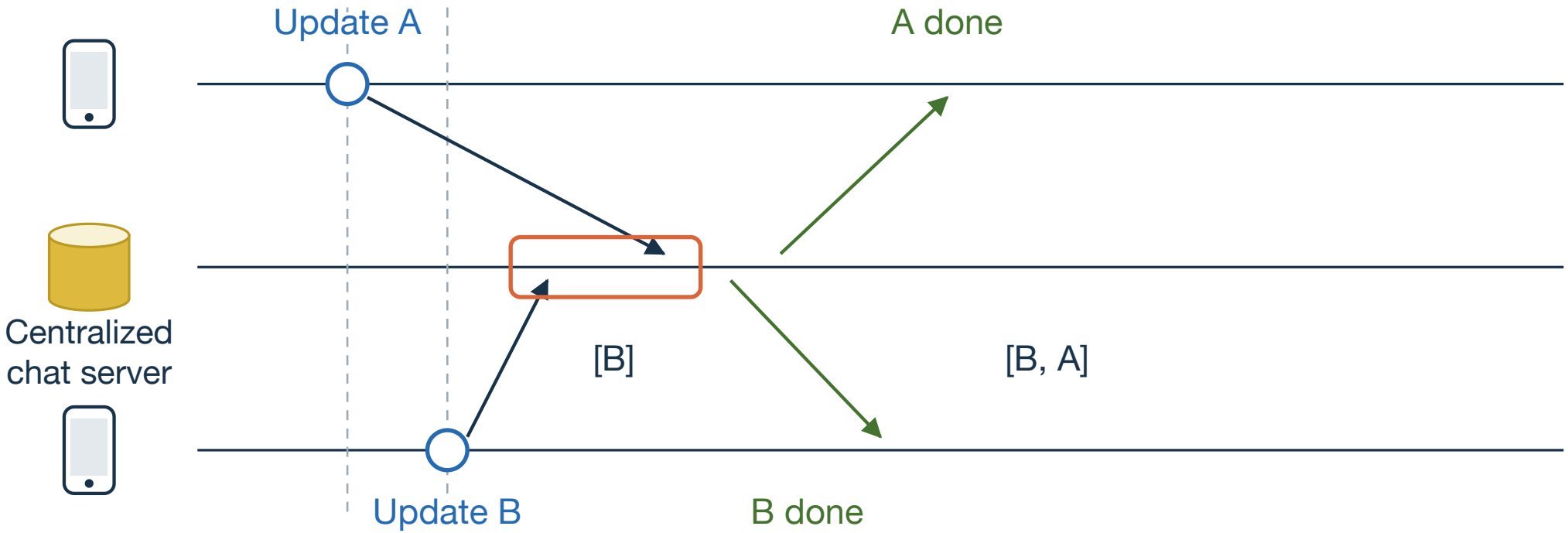
- The order is determined by how the server processes the requests



Strict consistency: strong but not practical

What is the order maintained by our centralized design?

- The order is determined by how the server processes the requests

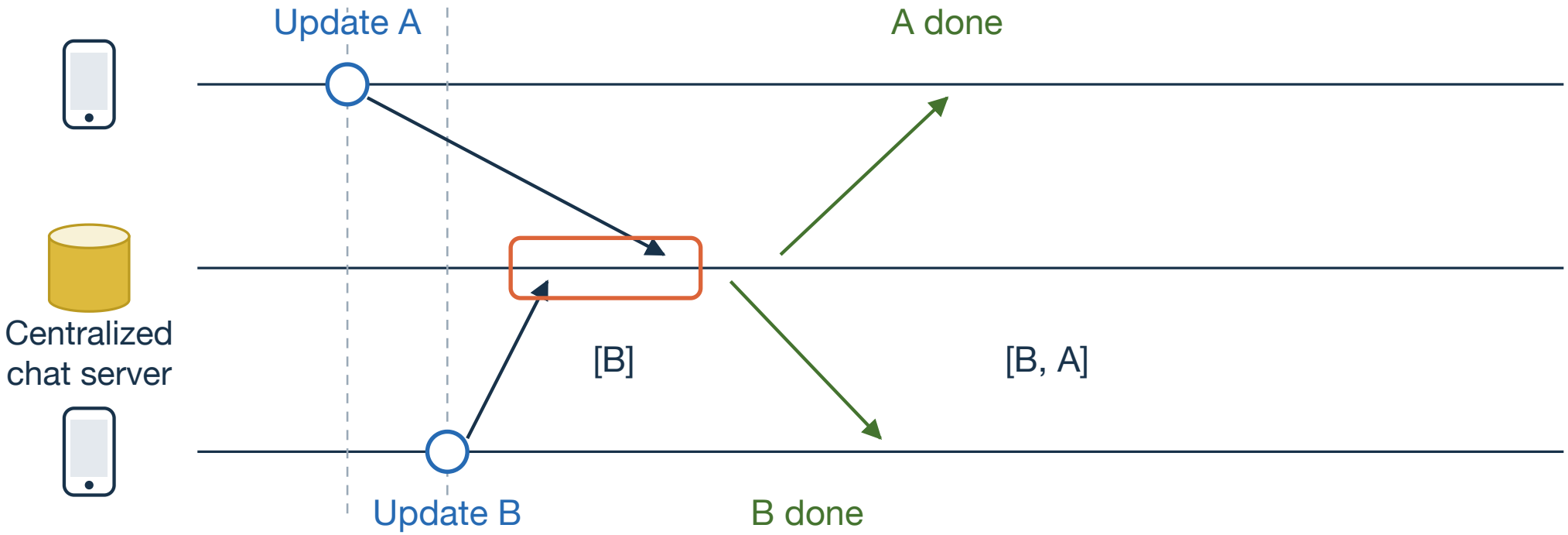


A vs. B is decided only by which arrives first

Order maintained by the centralized design: concurrent requests

For requests executed concurrently, we are unaware of the order

- Whether Update A or Update B goes first is totally determined by when the server received the requests



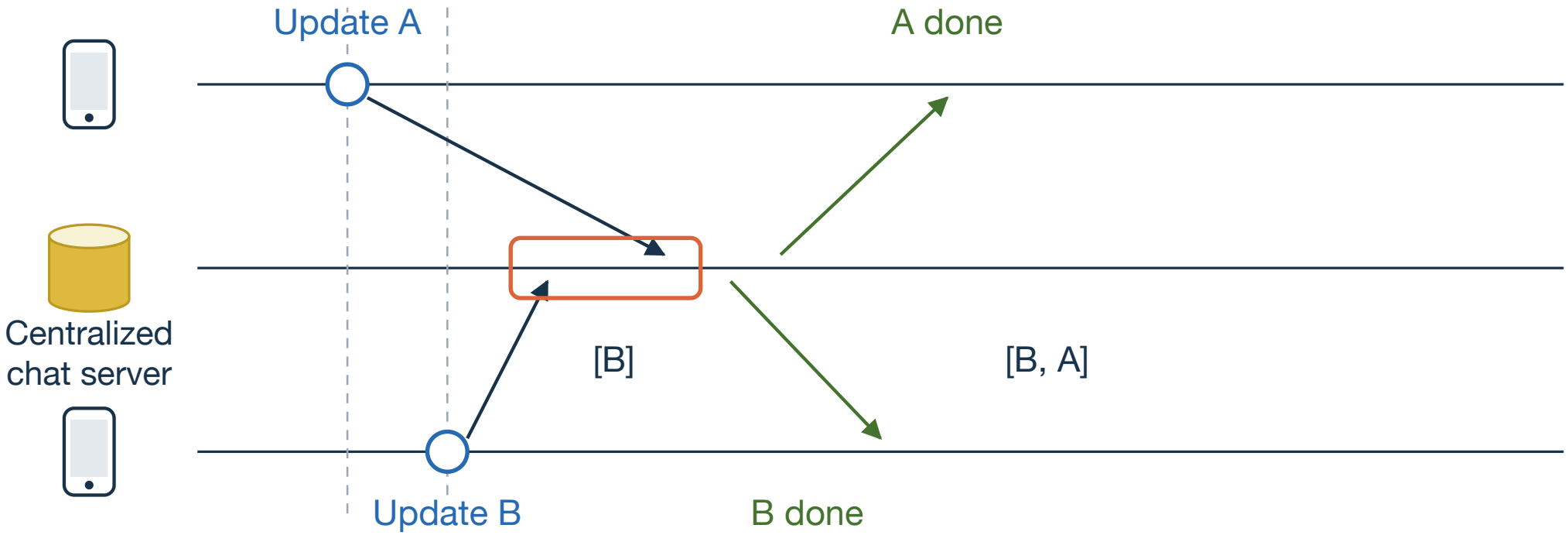
A vs. B is decided only by which arrives first

Order maintained by the centralized design: non-overlapping ones

For requests with a known completion → issue relationship, we know the order

- Why? A completion indicates an operation is done

The read is issued after Update B completes, so it must return B

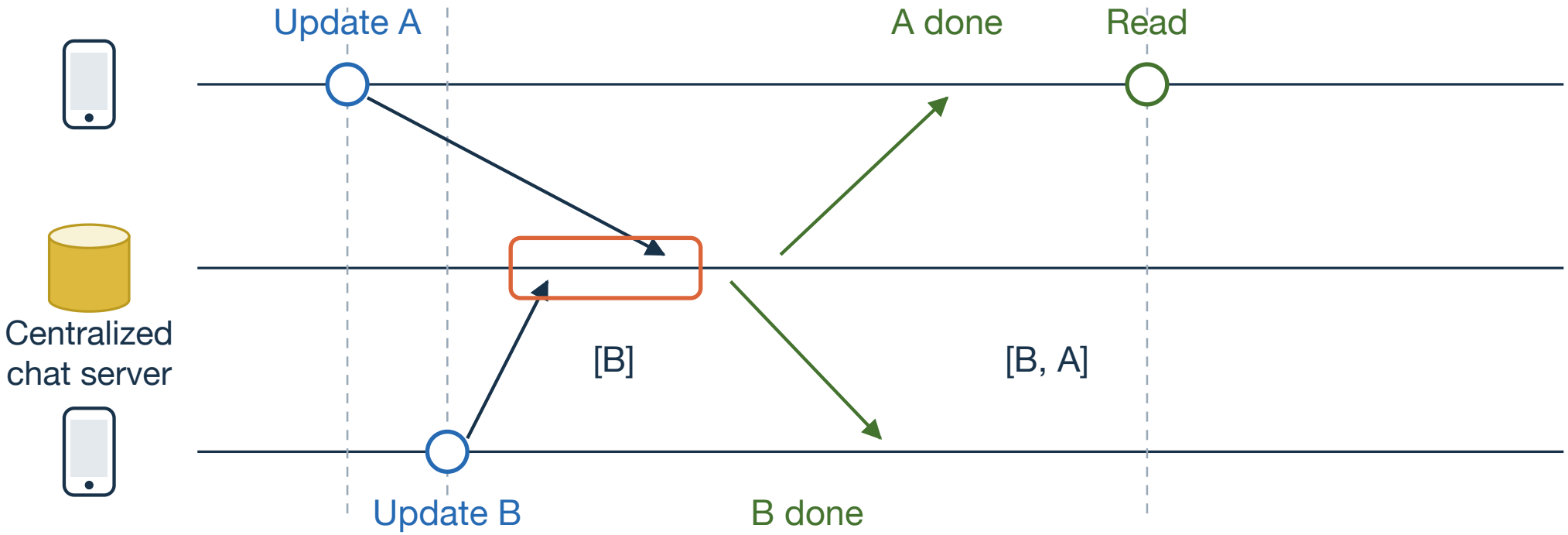


Order maintained by the centralized design: non-overlapping ones

For requests with a known completion → issue relationship, we know the order

- Why? A completion indicates an operation is done

The read is issued after Update B completes, so it must return B

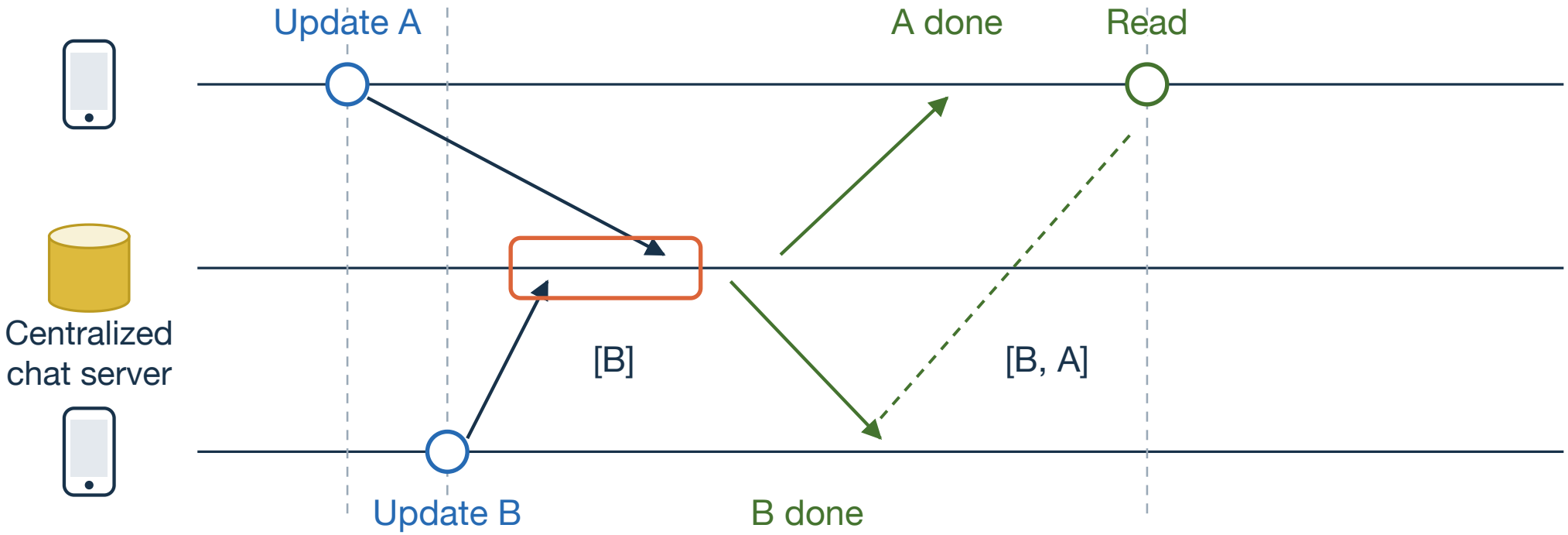


Order maintained by the centralized design: non-overlapping ones

For requests with a known completion → issue relationship, we know the order

- Why? A completion indicates an operation is done

The read is issued after Update B completes, so it must return B

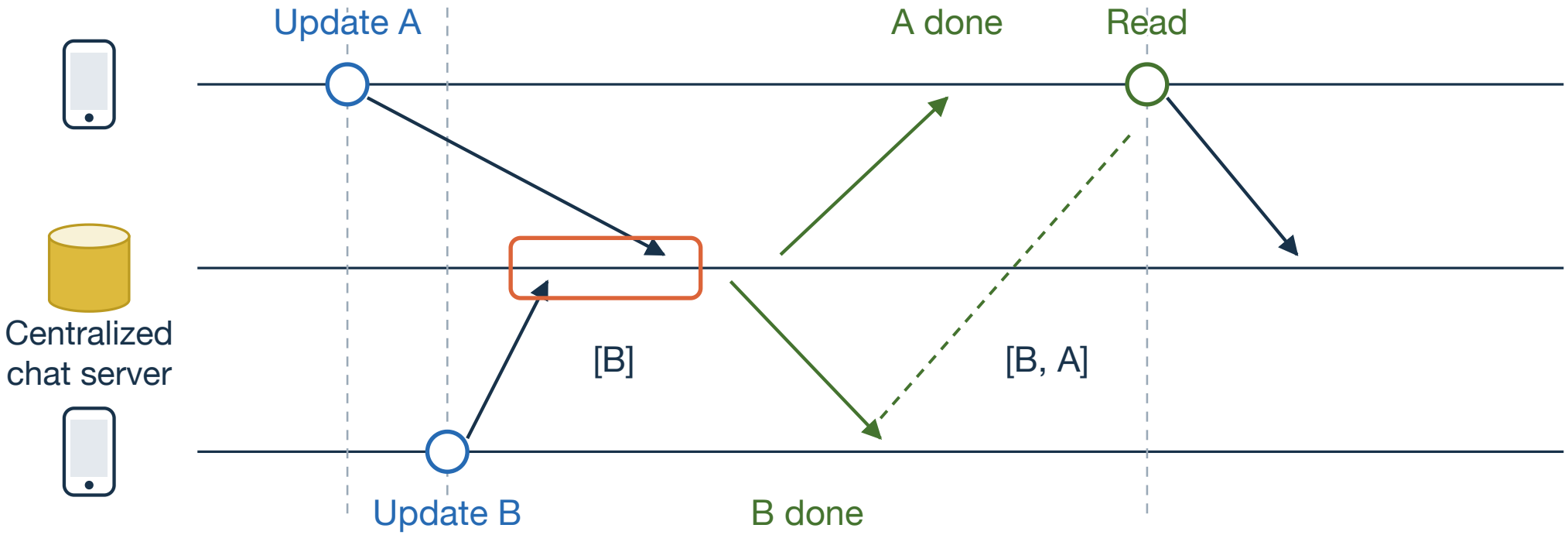


Order maintained by the centralized design: non-overlapping ones

For requests with a known completion → issue relationship, we know the order

- Why? A completion indicates an operation is done

The read is issued after Update B completes, so it must return B

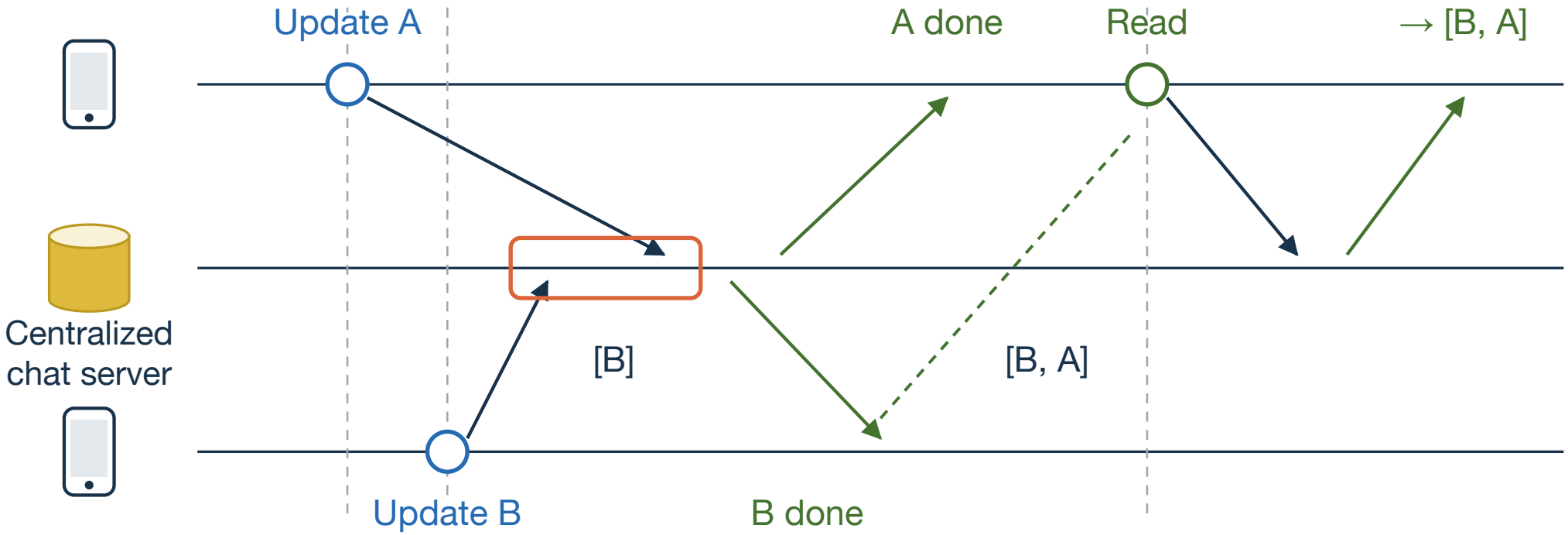


Order maintained by the centralized design: non-overlapping ones

For requests with a known completion → issue relationship, we know the order

- Why? A completion indicates an operation is done

The read is issued after Update B completes, so it must return B

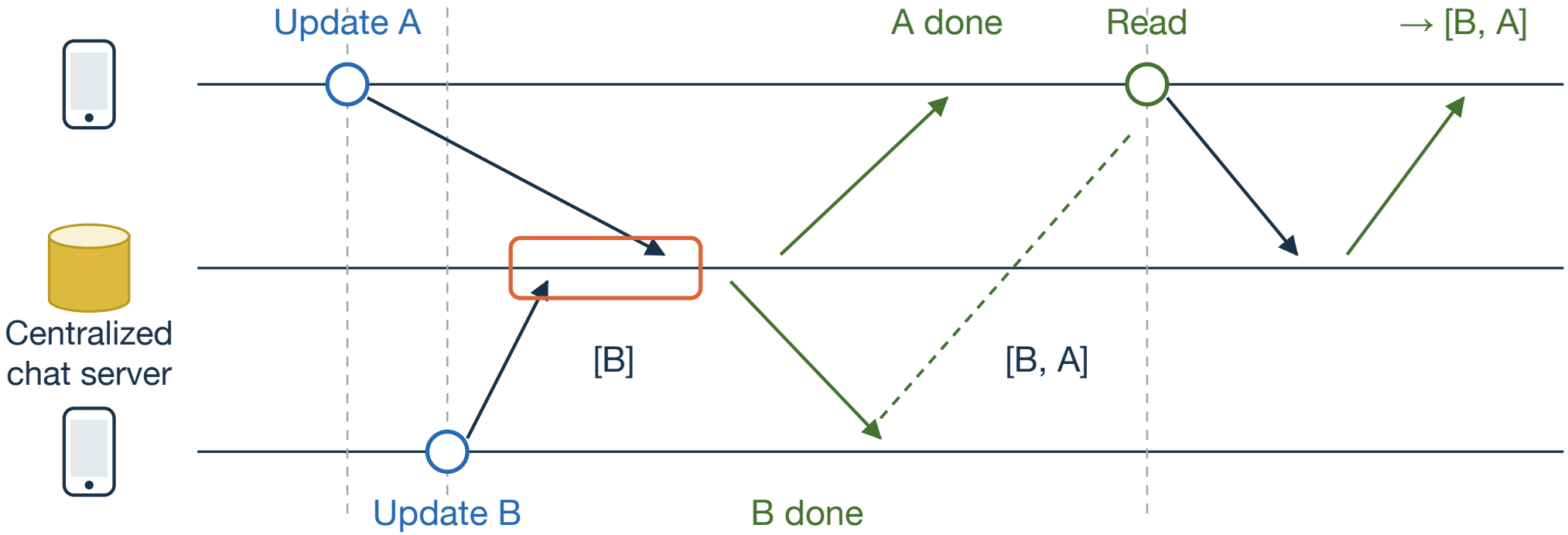


Order maintained by the centralized design: non-overlapping ones

For requests with a known completion → issue relationship, we know the order

- Why? A completion indicates an operation is done

The read is issued after Update B completes, so it must return B



B completes before the read is issued: the read must see B

Strong consistency model: what is the order provided?

#1. Global issuing order — strict consistency

#2. Global "completion-to-issuing" order — linearizability

- This is what our centralized approach gives

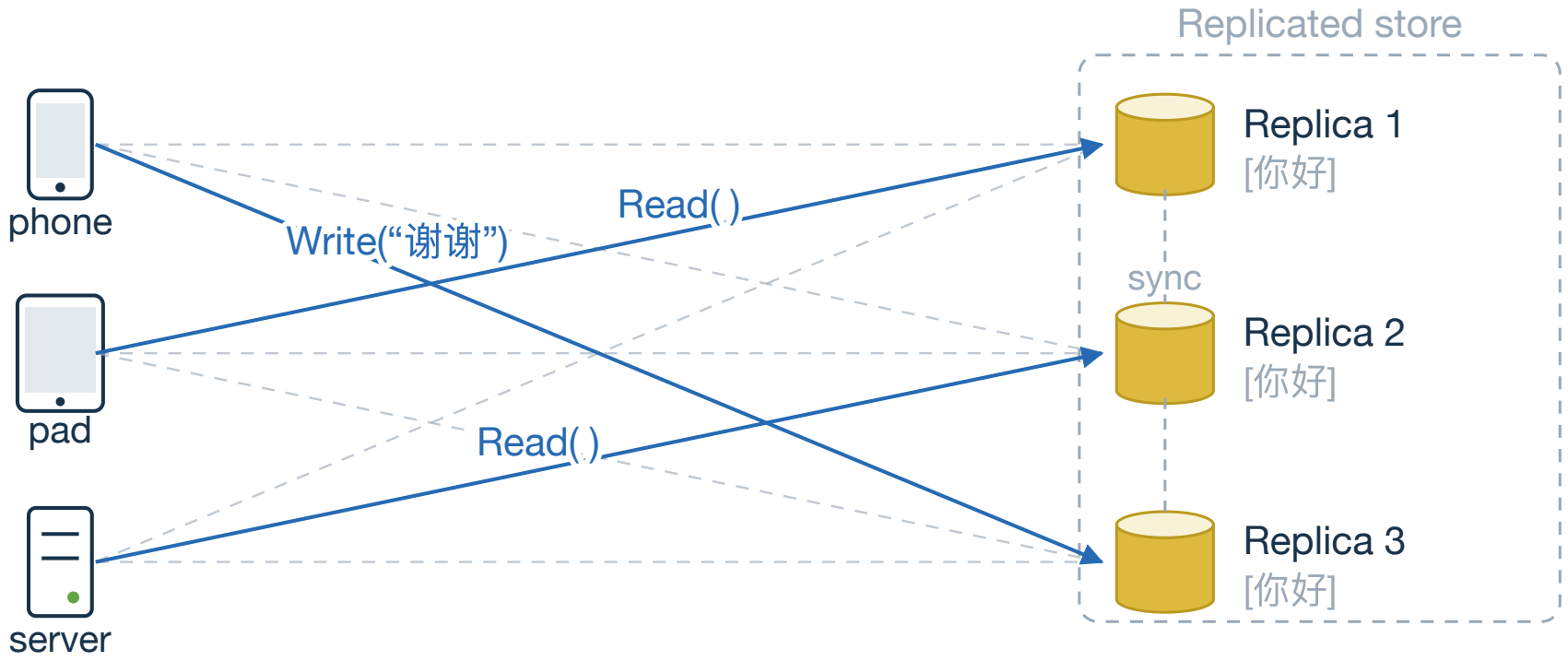
Why is linearizability important?

- It can be implemented in a distributed setup
- It is the strongest consistency model practically implemented in a distributed setup
- In contrast, strict consistency cannot be practically implemented

Implementing linearizability in replicated data storage

Setup: the data (e.g., the chat history) is replicated on multiple servers

- Read/Write are submitted by different processes, each to any replica



Goal: all observers observe a single serial execution

Implementing reads/writes in a linearizable system

Can a write be processed locally and then reordered upon sync?

- No, because different devices would then view different orders

Solution & principle: the one writer rule

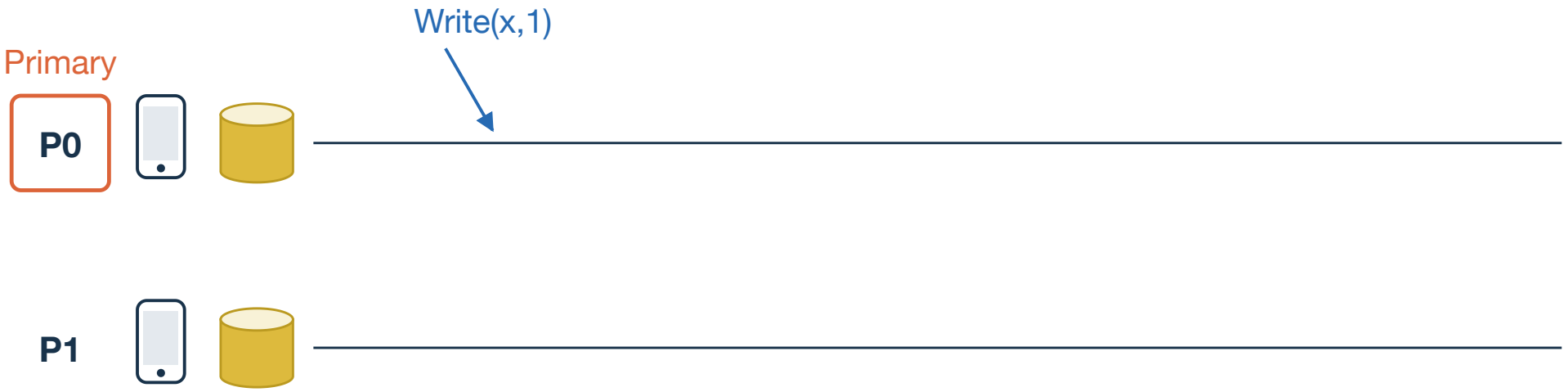
- Only one process is allowed to decide the read and write order
- The write order is then propagated to the other replicas

Primary-backup model (mimics the centralized approach)

For each object, clients send all reads/writes to one designated machine (primary)

For writes

- The primary forwards the write to all the replicas (backups)
- Each replica executes the write locally (in order)

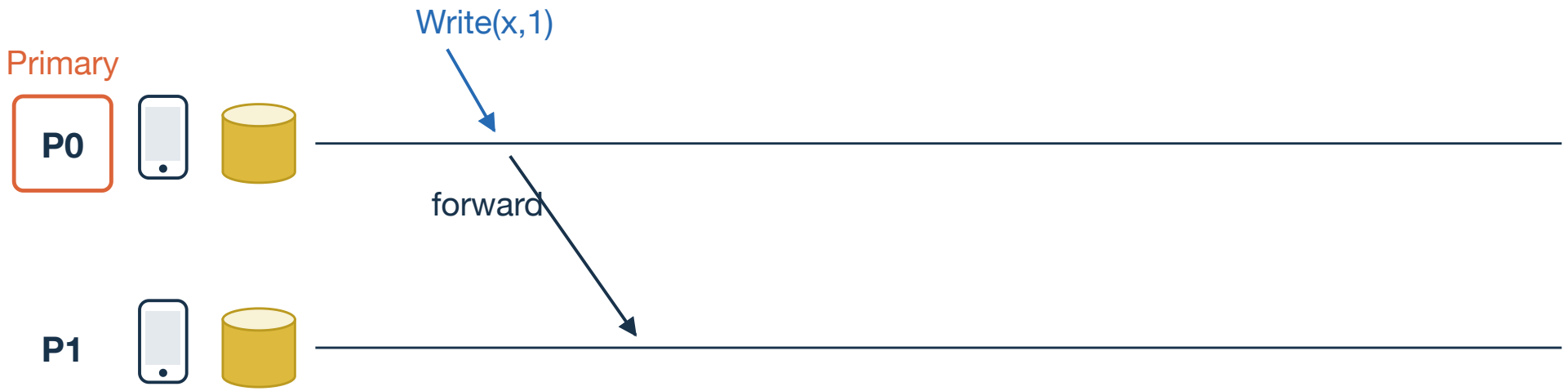


Primary-backup model (mimics the centralized approach)

For each object, clients send all reads/writes to one designated machine (primary)

For writes

- The primary forwards the write to all the replicas (backups)
- Each replica executes the write locally (in order)

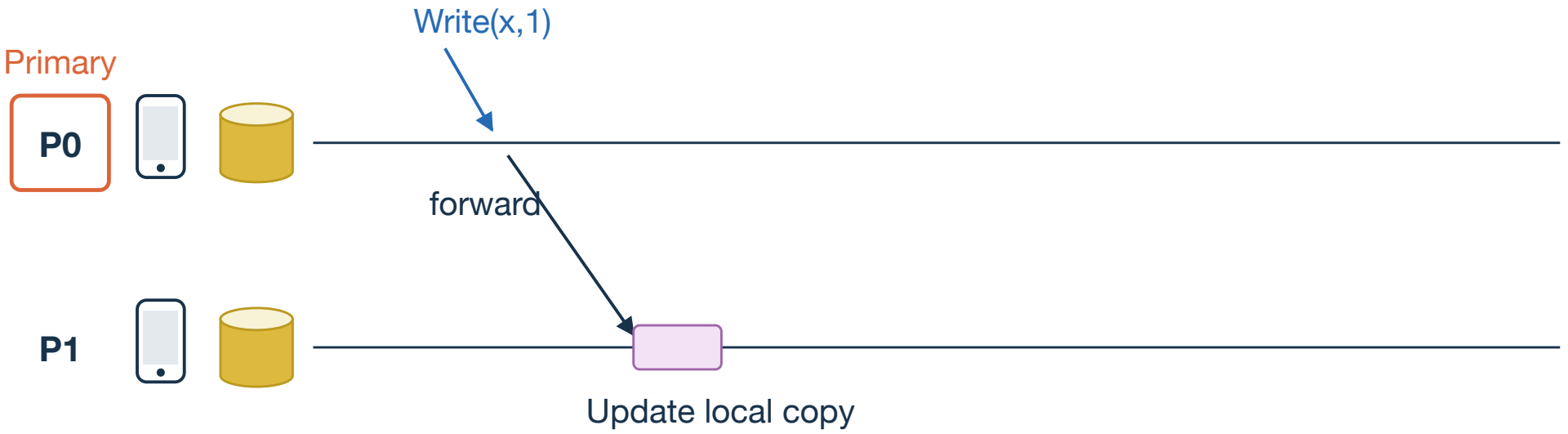


Primary-backup model (mimics the centralized approach)

For each object, clients send all reads/writes to one designated machine (primary)

For writes

- The primary forwards the write to all the replicas (backups)
- Each replica executes the write locally (in order)

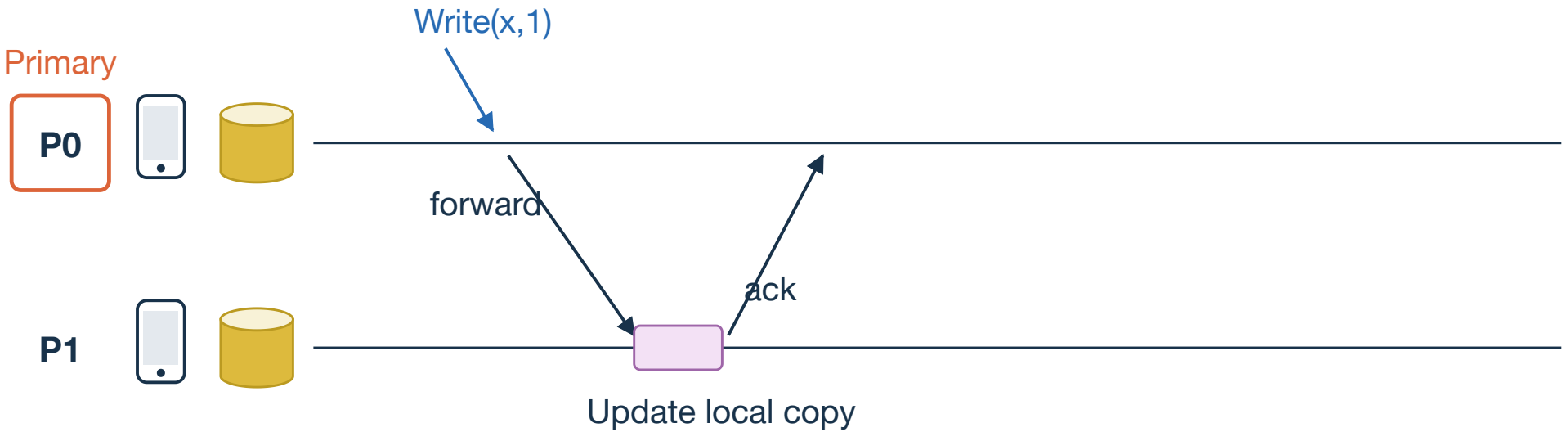


Primary-backup model (mimics the centralized approach)

For each object, clients send all reads/writes to one designated machine (primary)

For writes

- The primary forwards the write to all the replicas (backups)
- Each replica executes the write locally (in order)

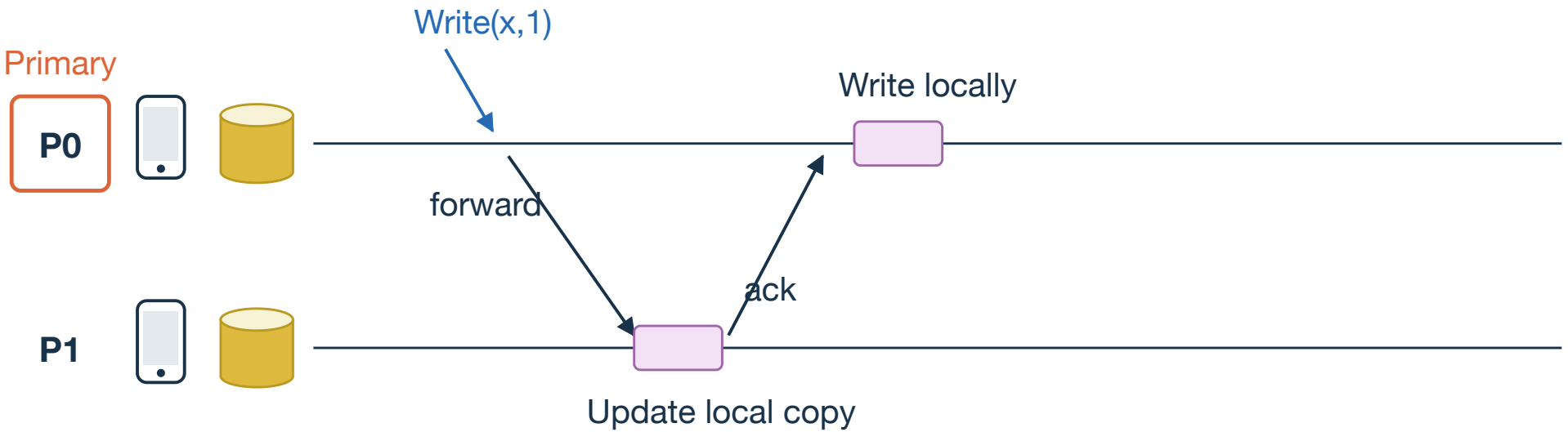


Primary-backup model (mimics the centralized approach)

For each object, clients send all reads/writes to one designated machine (primary)

For writes

- The primary forwards the write to all the replicas (backups)
- Each replica executes the write locally (in order)

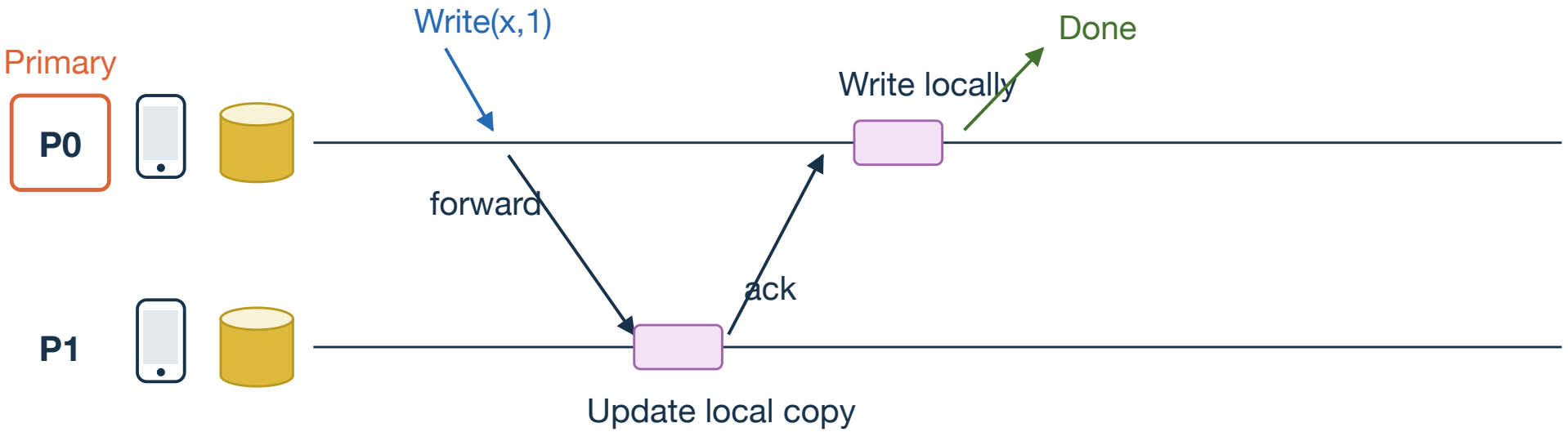


Primary-backup model (mimics the centralized approach)

For each object, clients send all reads/writes to one designated machine (primary)

For writes

- The primary forwards the write to all the replicas (backups)
- Each replica executes the write locally (in order)

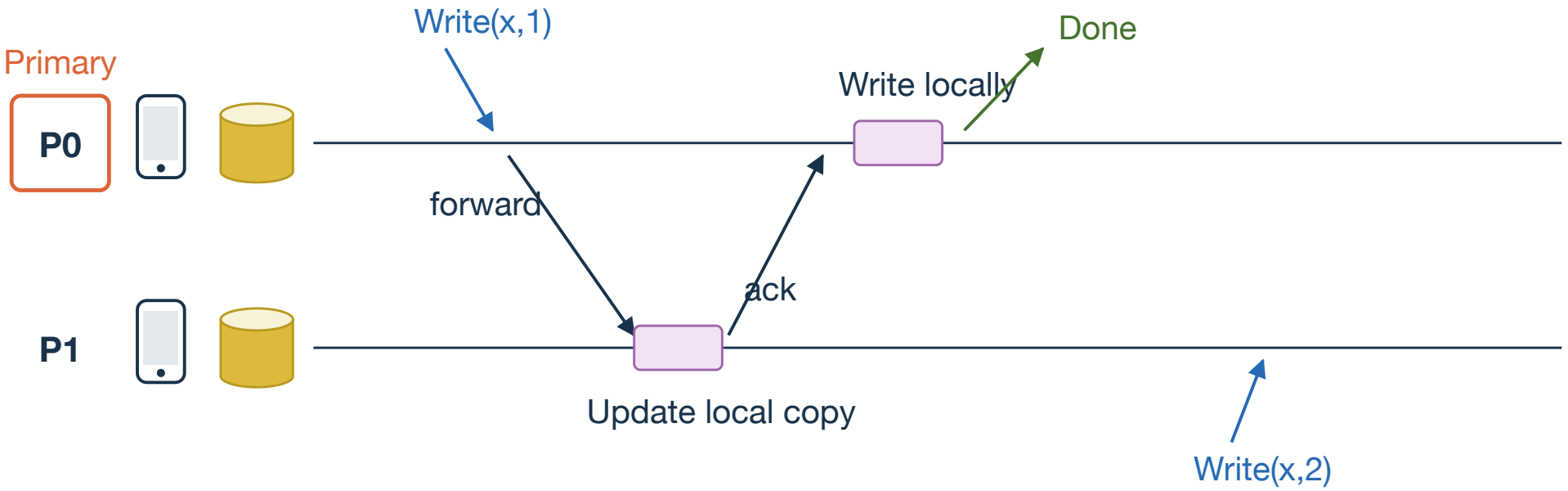


Primary-backup model (mimics the centralized approach)

For each object, clients send all reads/writes to one designated machine (primary)

For writes

- The primary forwards the write to all the replicas (backups)
- Each replica executes the write locally (in order)

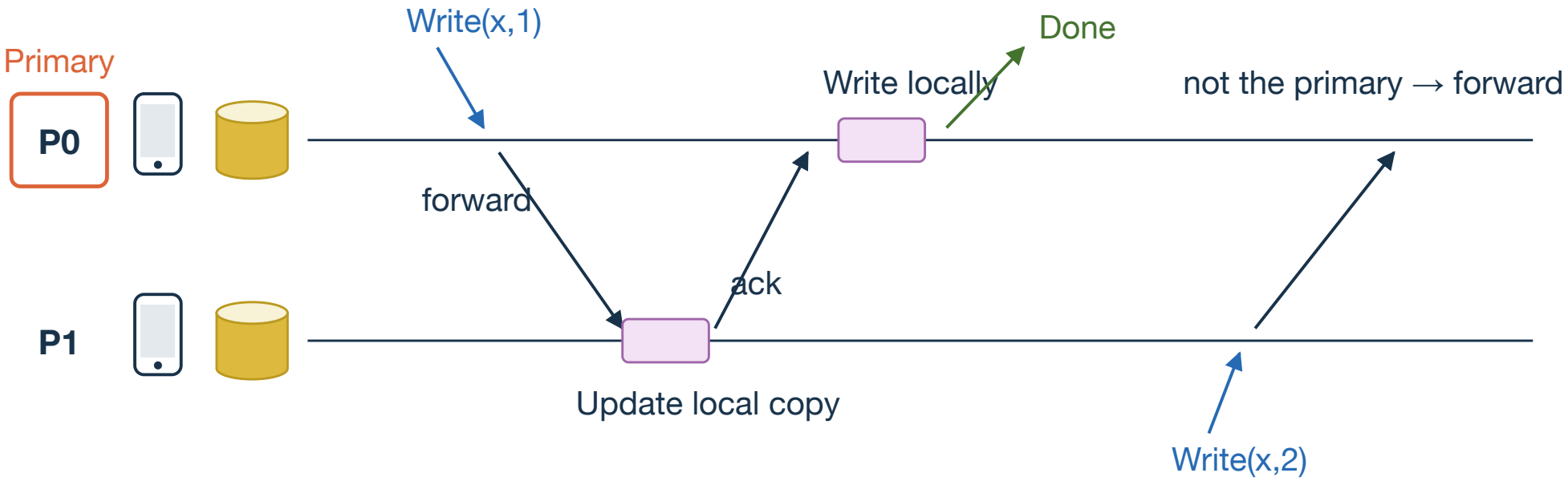


Primary-backup model (mimics the centralized approach)

For each object, clients send all reads/writes to one designated machine (primary)

For writes

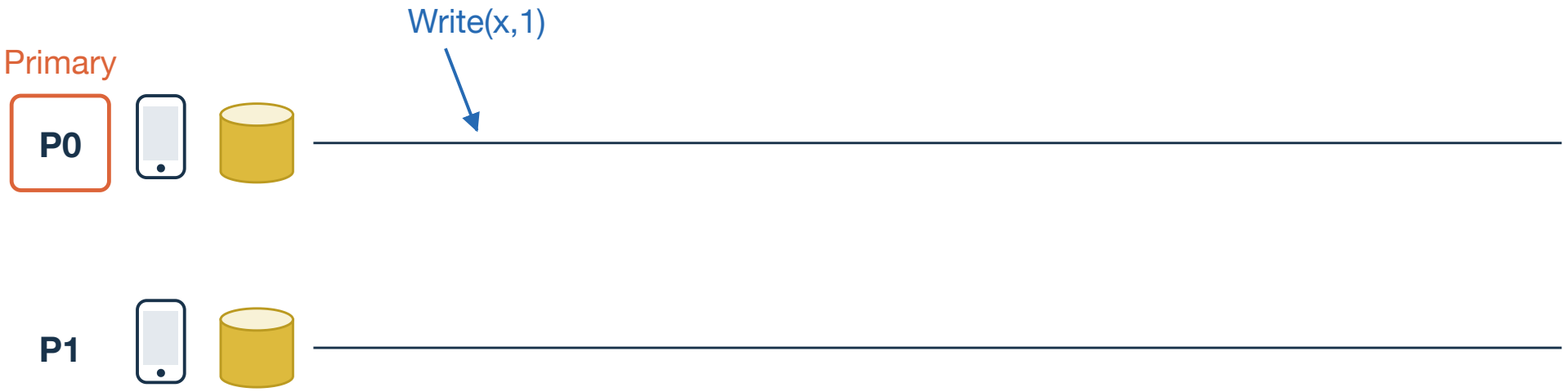
- The primary forwards the write to all the replicas (backups)
- Each replica executes the write locally (in order)



What does "in order" mean? (Recall GFS)

Suppose two writes are sent to the primary

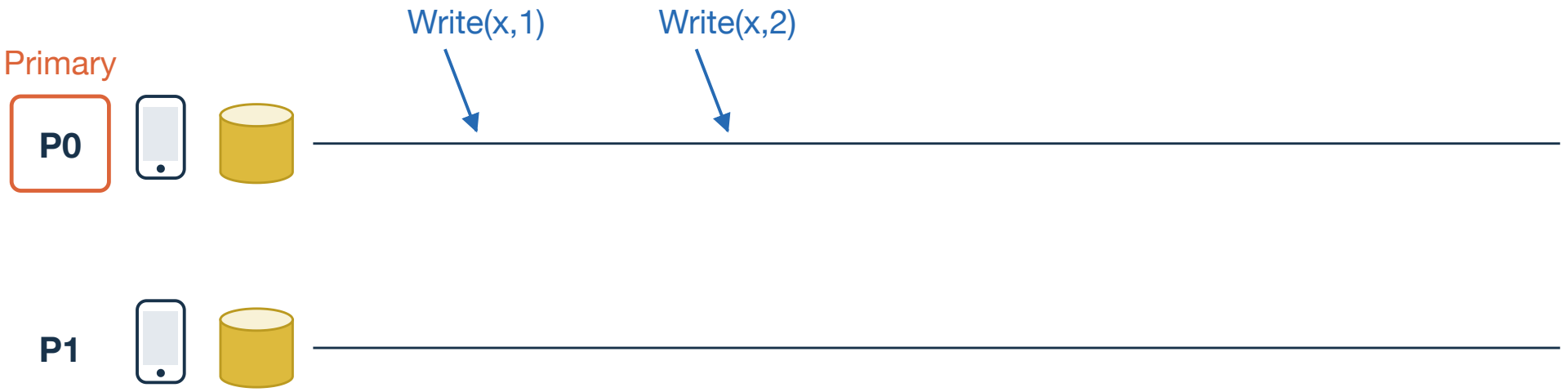
- Due to network problems, the messages may be reordered
- If a reorder happens, two replicas may apply the same writes in a different order



What does "in order" mean? (Recall GFS)

Suppose two writes are sent to the primary

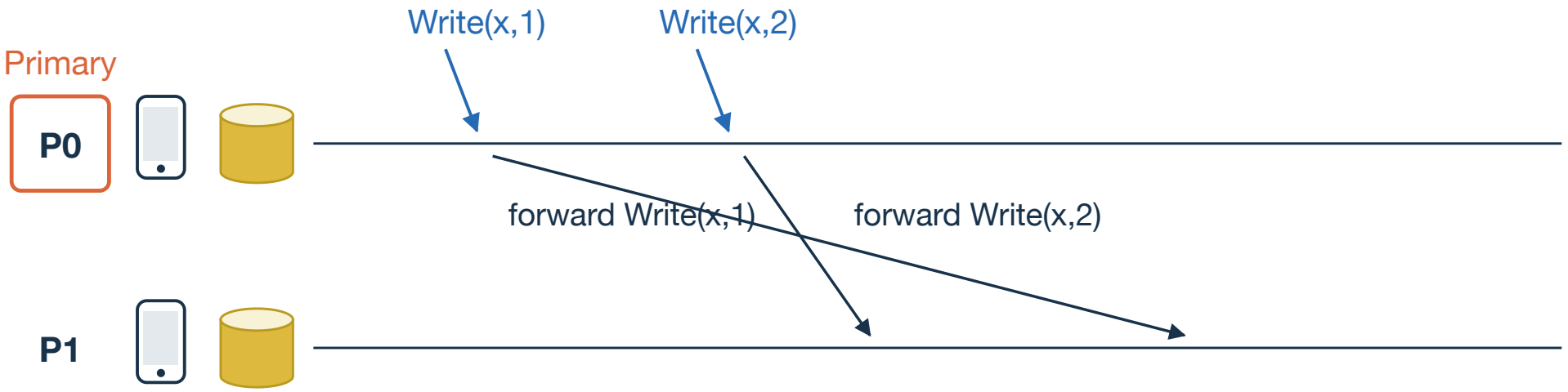
- Due to network problems, the messages may be reordered
- If a reorder happens, two replicas may apply the same writes in a different order



What does "in order" mean? (Recall GFS)

Suppose two writes are sent to the primary

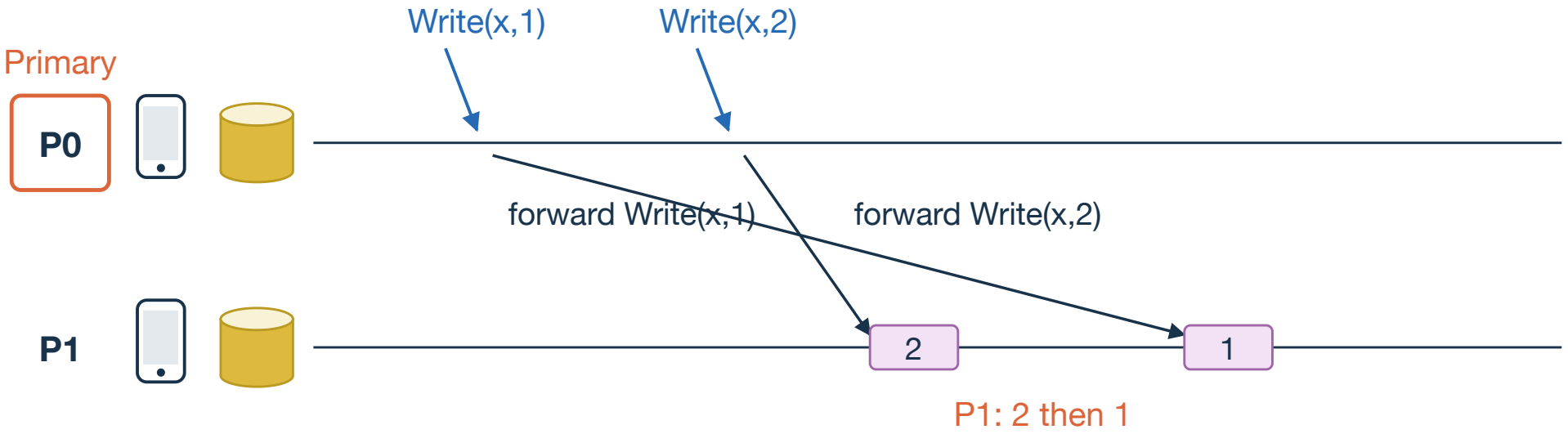
- Due to network problems, the messages may be reordered
- If a reorder happens, two replicas may apply the same writes in a different order



What does "in order" mean? (Recall GFS)

Suppose two writes are sent to the primary

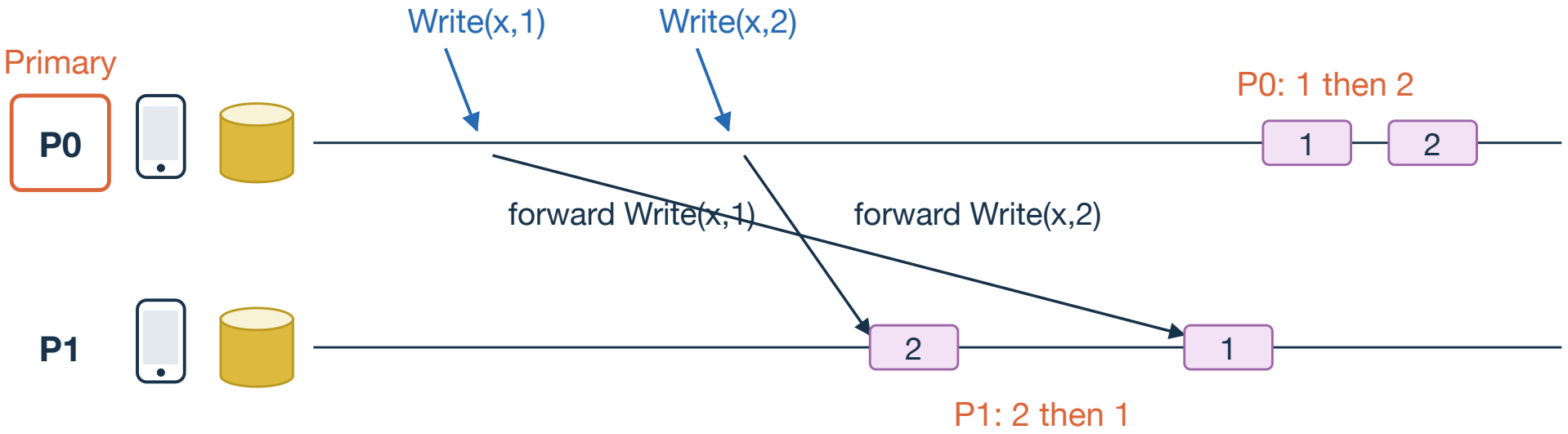
- Due to network problems, the messages may be reordered
- If a reorder happens, two replicas may apply the same writes in a different order



What does "in order" mean? (Recall GFS)

Suppose two writes are sent to the primary

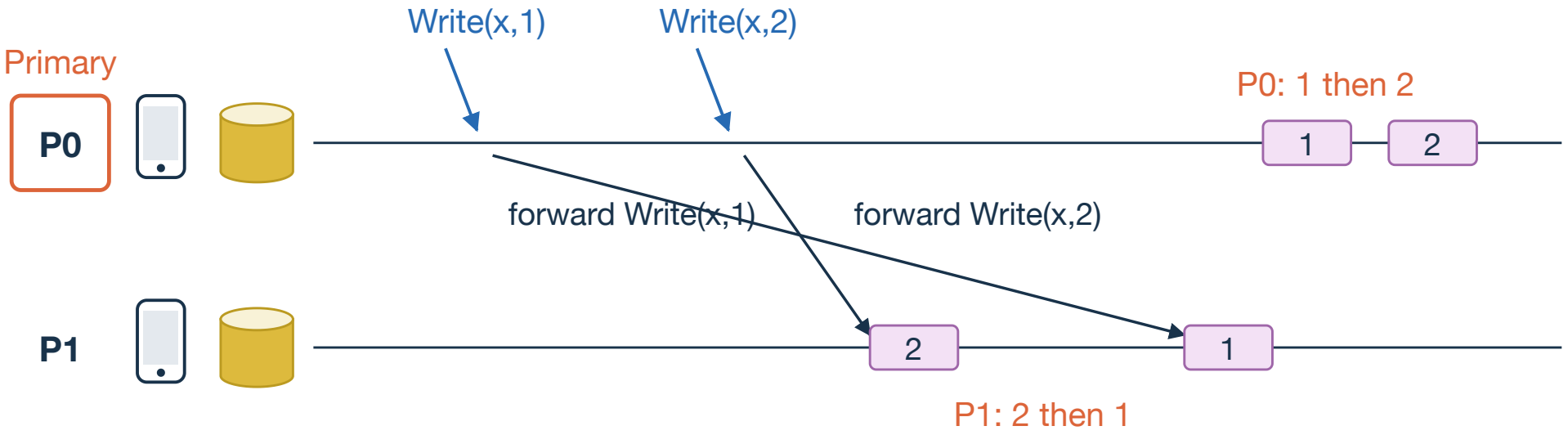
- Due to network problems, the messages may be reordered
- If a reorder happens, two replicas may apply the same writes in a different order



What does "in order" mean? (Recall GFS)

Suppose two writes are sent to the primary

- Due to network problems, the messages may be reordered
- If a reorder happens, two replicas may apply the same writes in a different order



Same two writes, two replicas apply different orders

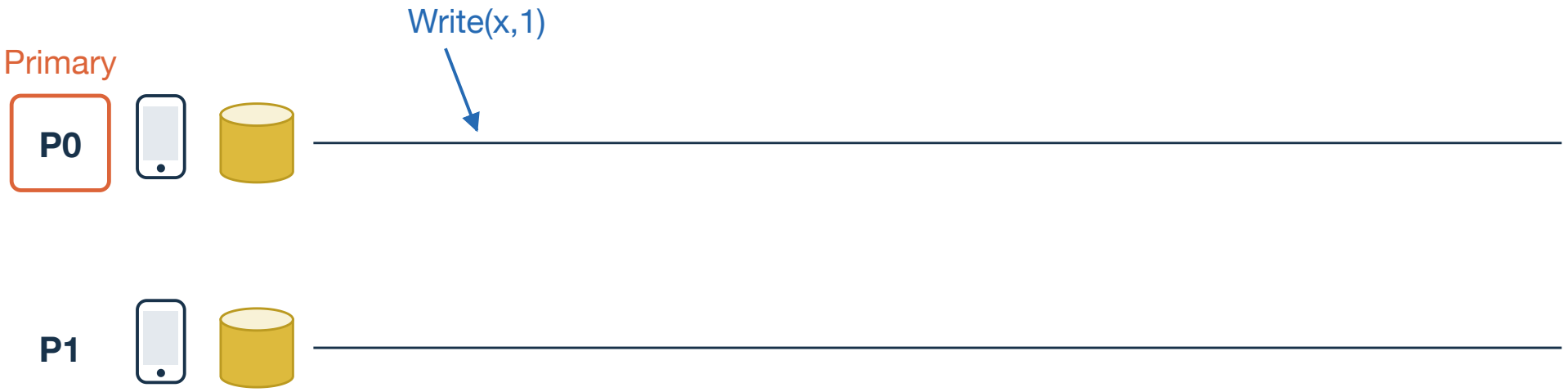
In-order updates: the primary must use some seq number

The seq number is the order of the updates

- A possible implementation: a counter incremented upon each write

All replicas apply writes in seq number order

- Delay a write if the previous one has not finished



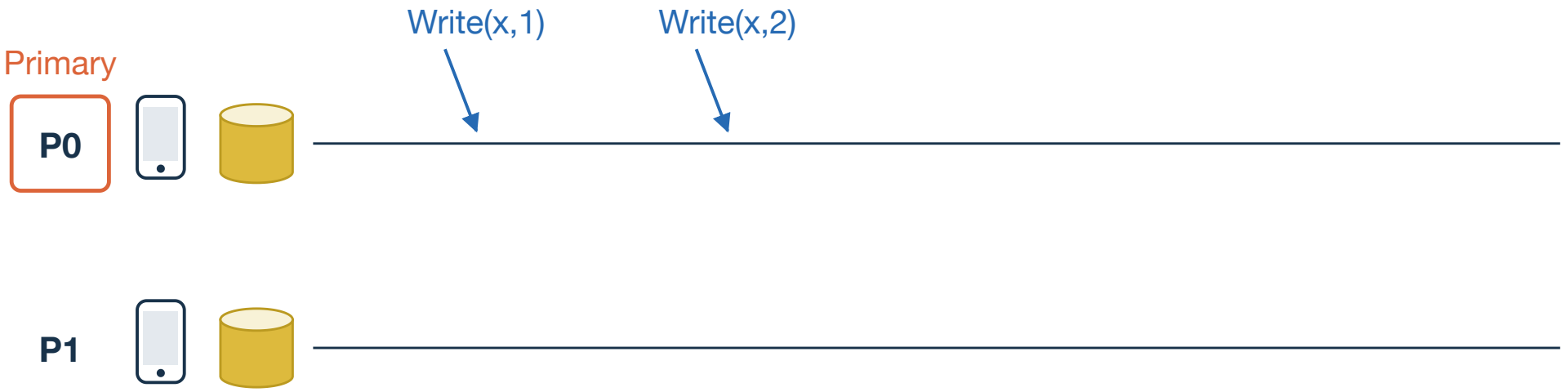
In-order updates: the primary must use some seq number

The seq number is the order of the updates

- A possible implementation: a counter incremented upon each write

All replicas apply writes in seq number order

- Delay a write if the previous one has not finished



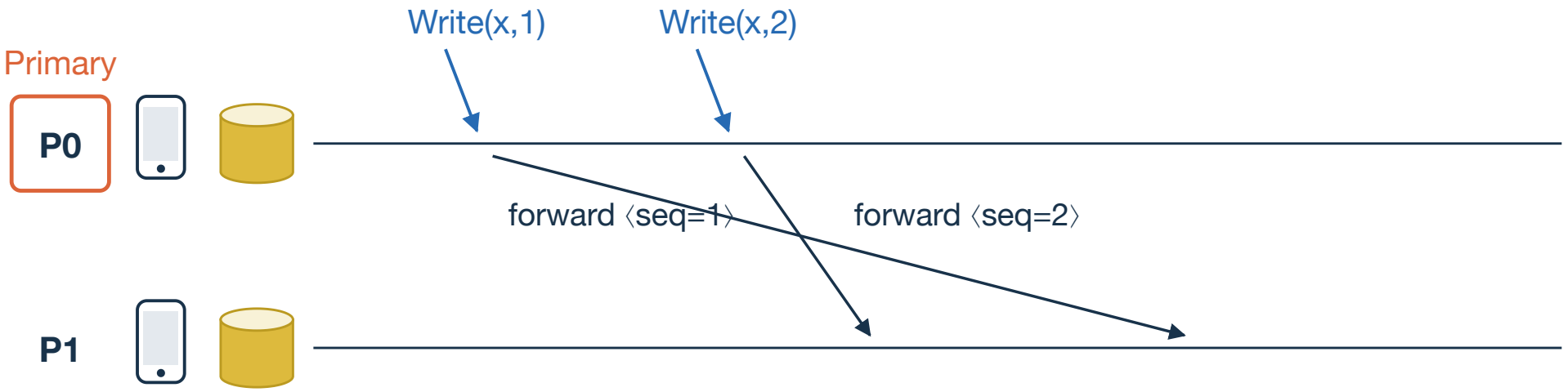
In-order updates: the primary must use some seq number

The seq number is the order of the updates

- A possible implementation: a counter incremented upon each write

All replicas apply writes in seq number order

- Delay a write if the previous one has not finished



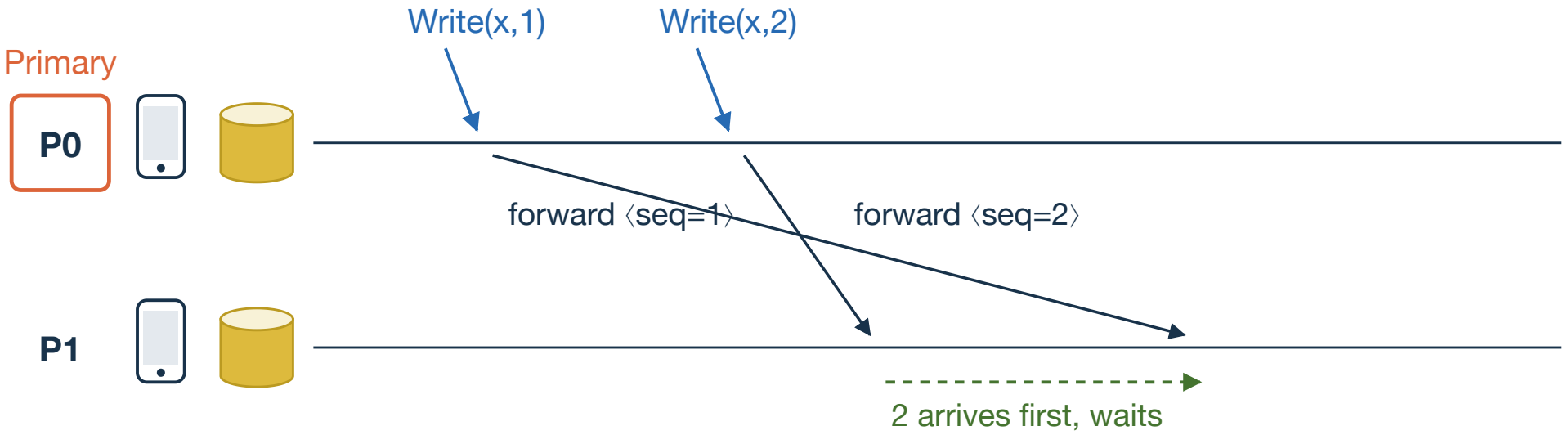
In-order updates: the primary must use some seq number

The seq number is the order of the updates

- A possible implementation: a counter incremented upon each write

All replicas apply writes in seq number order

- Delay a write if the previous one has not finished



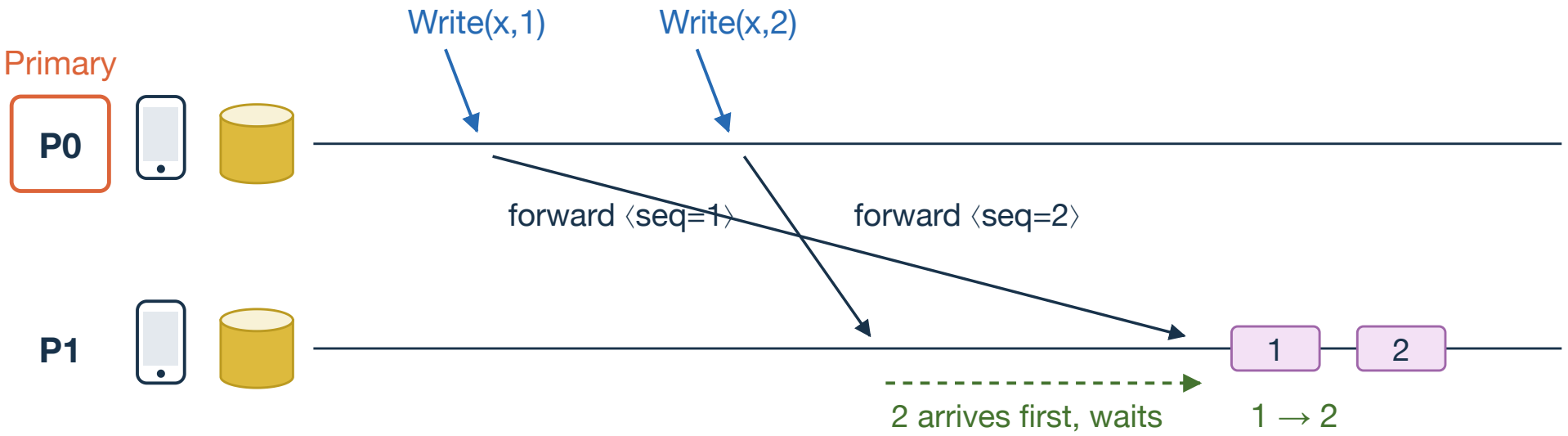
In-order updates: the primary must use some seq number

The seq number is the order of the updates

- A possible implementation: a counter incremented upon each write

All replicas apply writes in seq number order

- Delay a write if the previous one has not finished



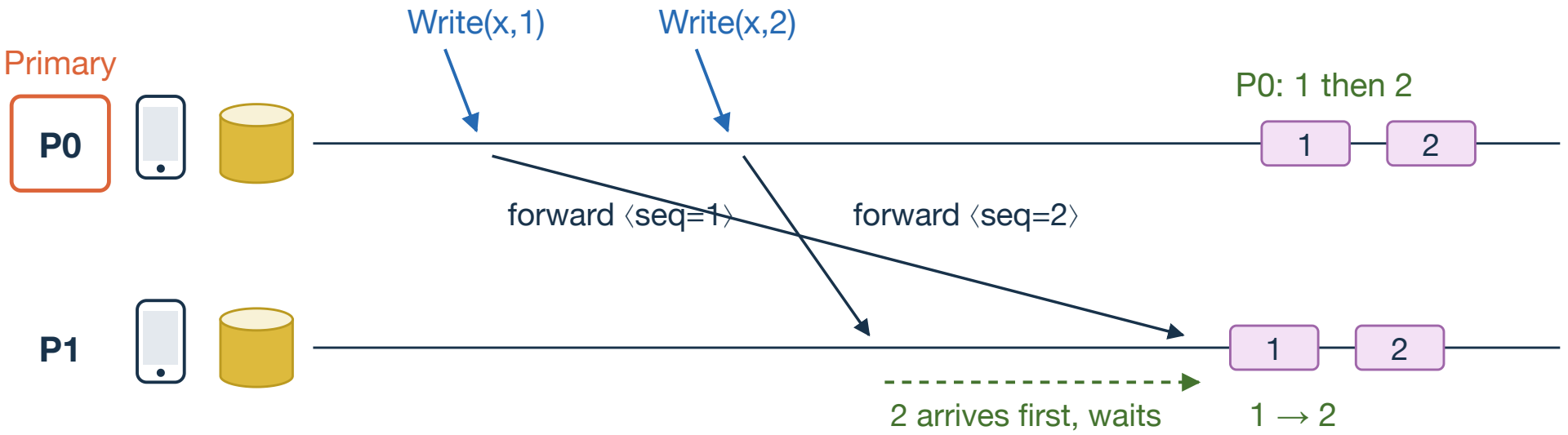
In-order updates: the primary must use some seq number

The seq number is the order of the updates

- A possible implementation: a counter incremented upon each write

All replicas apply writes in seq number order

- Delay a write if the previous one has not finished



Drawback of primary-backup?

Performance issues

- Falls back to a centralized design, with even lower performance, e.g., an extra roundtrip

Reliability issues

- If the primary crashes, the others cannot work
- If a backup crashes, the primary cannot finish a write — the same problem as the naïve "sync with all" approach

We will come back to the reliability issue in later lectures

Question: can we optimize the implementation?

E.g., have a read return the local copy, without asking the primary

- Unfortunately, it is not linearizable — see below

This model is sequential consistency: per-process issuing/completion order

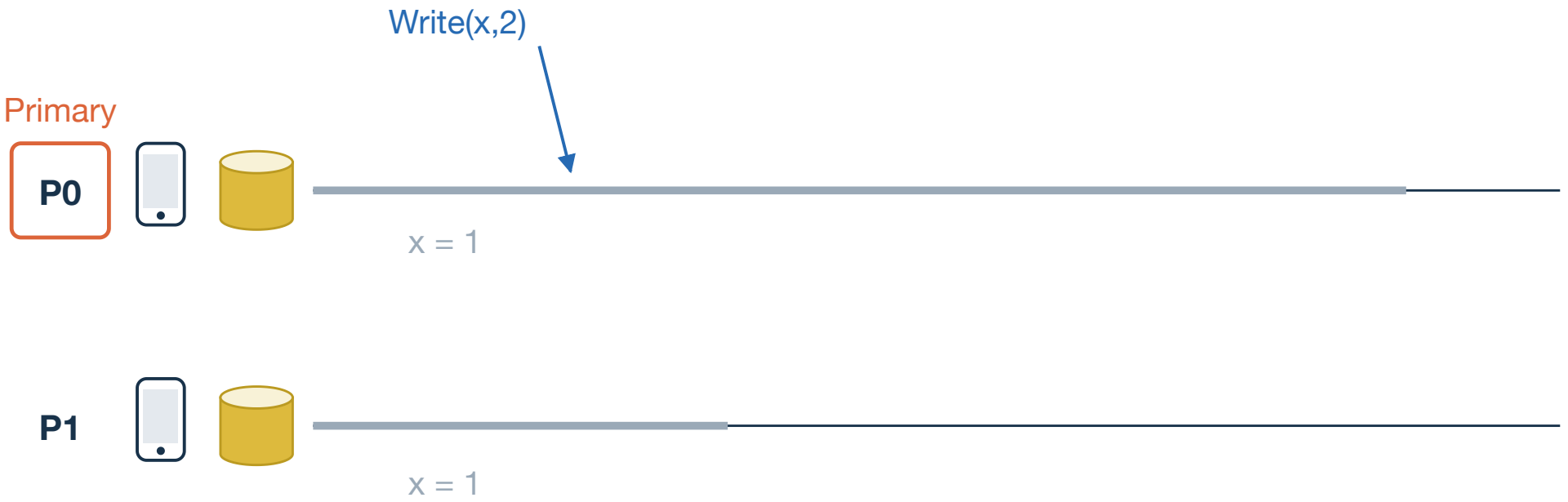


Question: can we optimize the implementation?

E.g., have a read return the local copy, without asking the primary

- Unfortunately, it is not linearizable — see below

This model is sequential consistency: per-process issuing/completion order

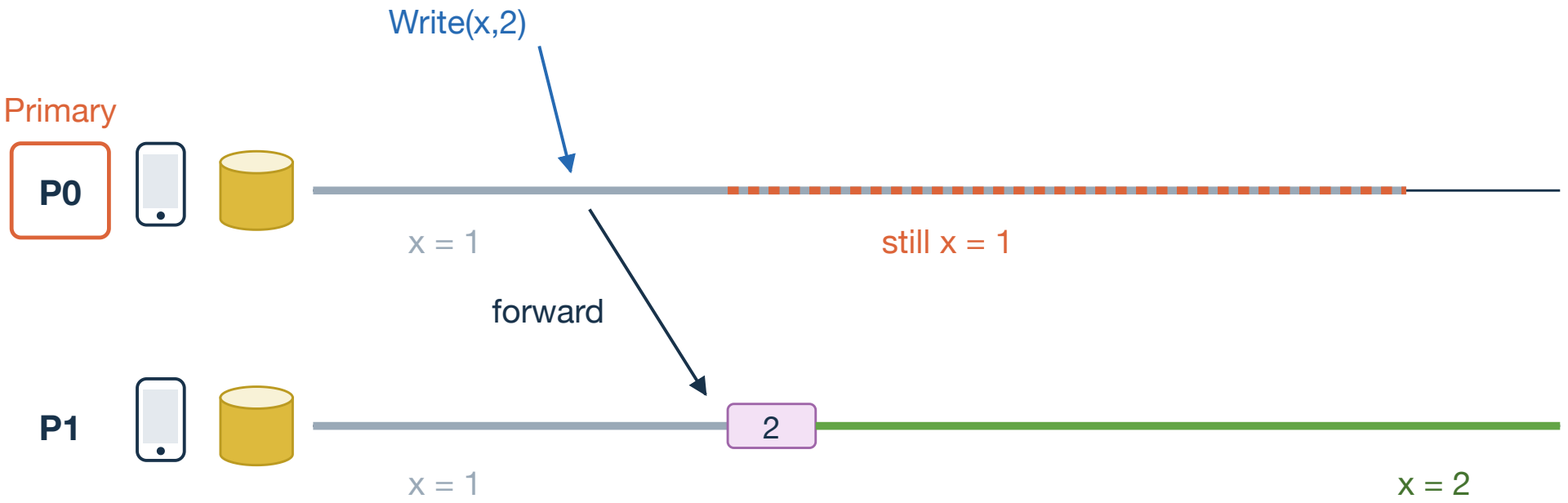


Question: can we optimize the implementation?

E.g., have a read return the local copy, without asking the primary

- Unfortunately, it is not linearizable — see below

This model is sequential consistency: per-process issuing/completion order

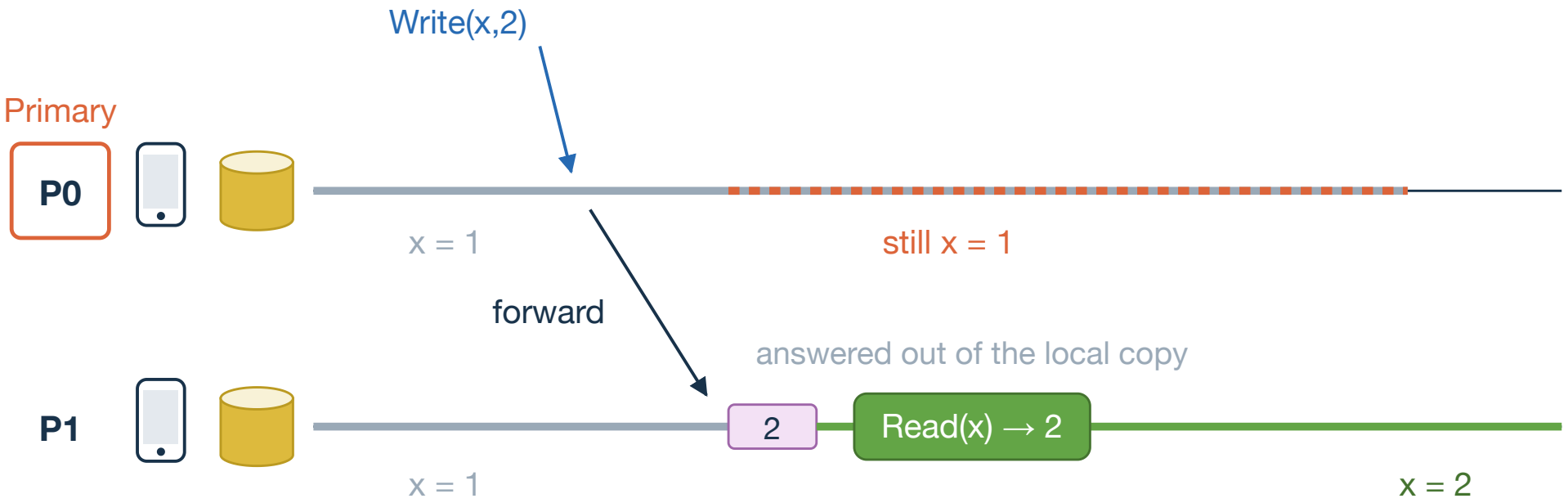


Question: can we optimize the implementation?

E.g., have a read return the local copy, without asking the primary

- Unfortunately, it is not linearizable — see below

This model is sequential consistency: per-process issuing/completion order

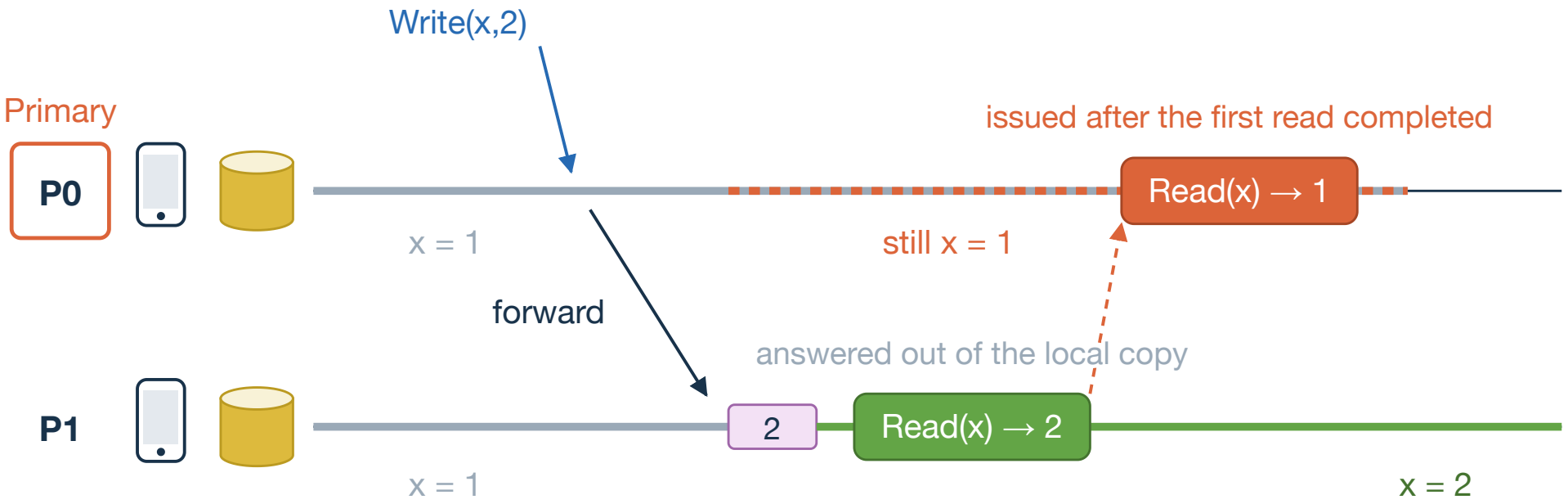


Question: can we optimize the implementation?

E.g., have a read return the local copy, without asking the primary

- Unfortunately, it is not linearizable — see below

This model is sequential consistency: per-process issuing/completion order

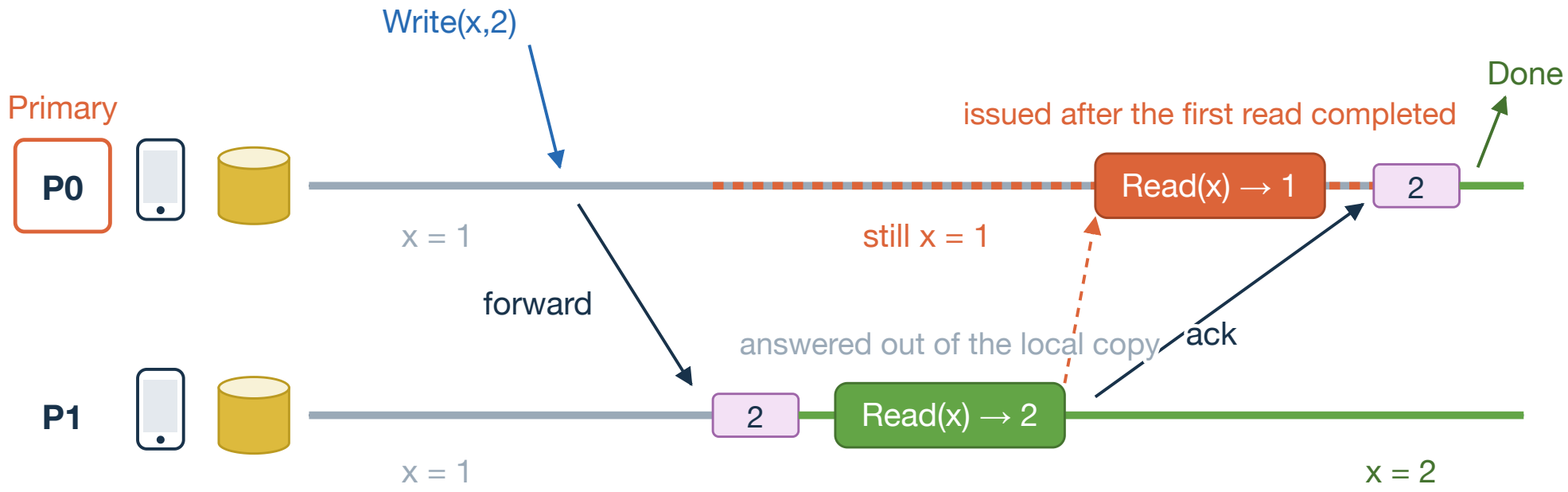


Question: can we optimize the implementation?

E.g., have a read return the local copy, without asking the primary

- Unfortunately, it is not linearizable — see below

This model is sequential consistency: per-process issuing/completion order

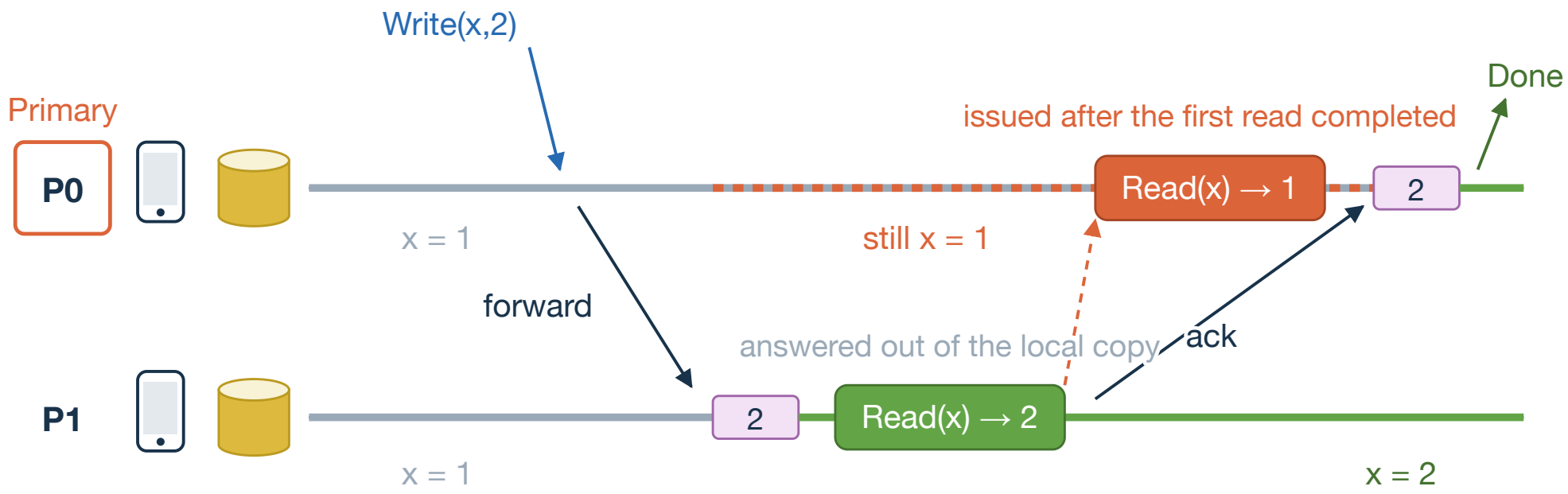


Question: can we optimize the implementation?

E.g., have a read return the local copy, without asking the primary

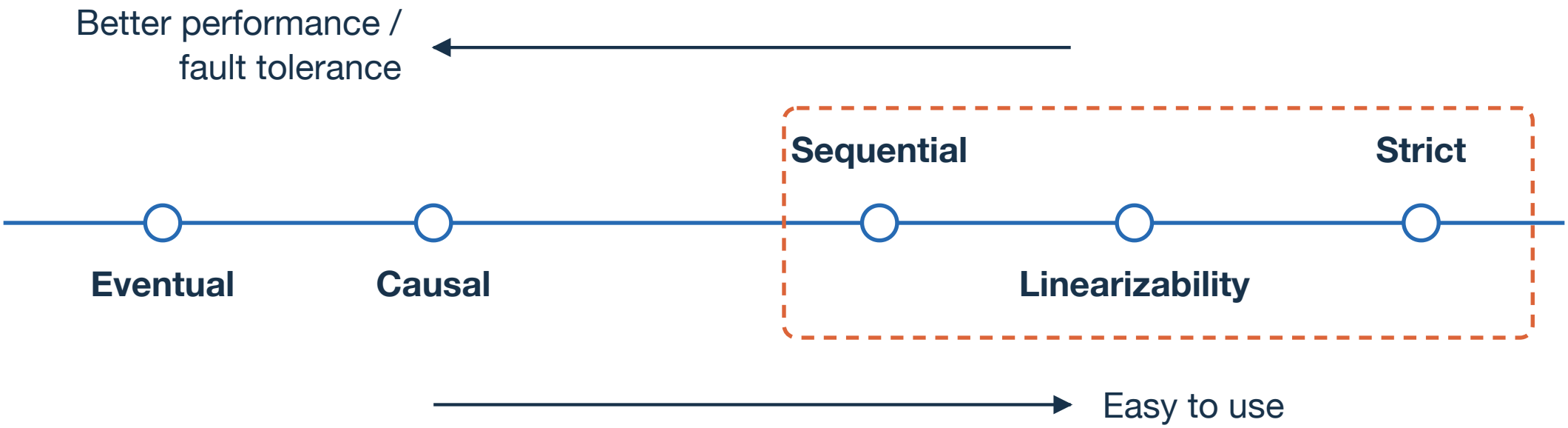
- Unfortunately, it is not linearizable — see below

This model is sequential consistency: per-process issuing/completion order



The second read is issued after the first completes, yet it returns the older value
It breaks the completion → issuing order: not linearizable

Back to the spectrum



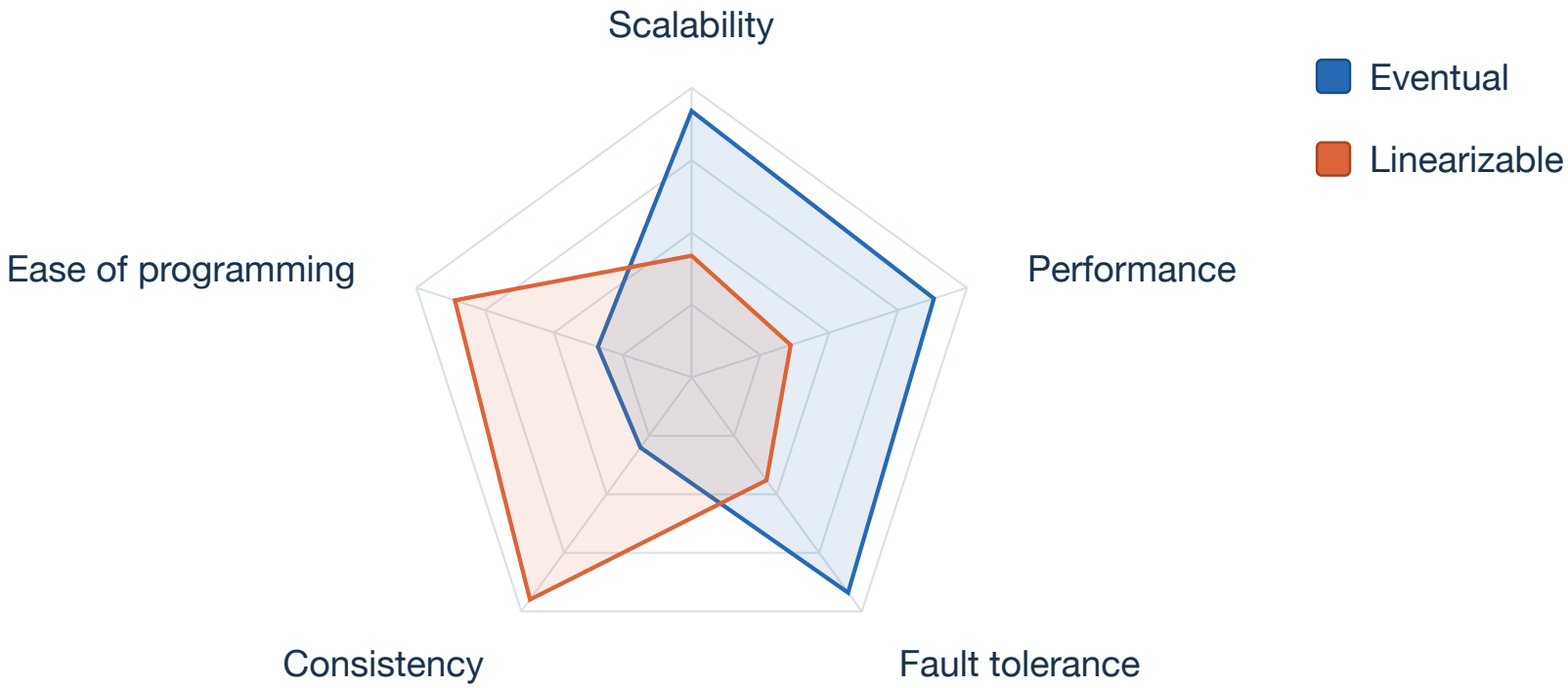
The further right, the fewer optimizations the implementer is allowed

Eventual vs. linearizability

Scalability: there is still a single primary, but the operations are much lighter

Ease of programming: not so easy — update functions and anomalies

Fault tolerance: as long as a single replica is up, the client can work



Summary of today's lecture

What we covered

- What is a consistency model & why we need it
- The eventual consistency model
- Implementing eventual consistency w/ various timestamps — non-trivial once we consider user experience
- Strong consistency models: strict consistency & linearizability

We used a ChatAPP as our example, but the model generalizes

- The data can be any data structure
- The operations can be any operations updating the data

A consistency model = what can happen + what cannot

参考文献

[1] Fire-Flyer File System (3FS) — DeepSeek, 2025 (P4, P7, P8)

[2] Fire-Flyer AI-HPC: A Cost-Effective Software-Hardware Co-Design for Deep Learning — An et al., 2024. SC (P4, P6, P9)

[3] 3FS design notes — DeepSeek, 2025 (P5, P6, P9, P10, P11, P12, P13, P14, P15, P16)

[4] The Google File System — Ghemawat, Gobioff & Leung, 2003. SOSP (P6, P19, P20)

[5] MapReduce: Simplified Data Processing on Large Clusters — Dean & Ghemawat, 2004. OSDI (P8)

[6] GFS: Evolution on Fast-forward — McKusick & Quinlan, 2009. ACM Queue (P11)

[7] Eventually Consistent — Werner Vogels, 2009. Communications of the ACM (P44)

[8] Time, Clocks, and the Ordering of Events in a Distributed System — Leslie Lamport, 1978. Communications of the ACM (P57)

[9] Linearizability: A Correctness Condition for Concurrent Objects — Herlihy and Wing, 1990. ACM TOPLAS (P80)