Distributed Systems

The second half of *Concurrent and Distributed Systems*

https://www.cl.cam.ac.uk/teaching/current/ConcDisSys

Dr. Martin Kleppmann (mk428@cam)

University of Cambridge

Computer Science Tripos, Part IB

Lecture 5

Replication

Replication

- Keeping a copy of the same data on multiple nodes
- Databases, filesystems, caches, . . .
- A node that has a copy of the data is called a **replica**

Replication

- Keeping a copy of the same data on multiple nodes
- Databases, filesystems, caches, . . .
- A node that has a copy of the data is called a **replica**
- If some replicas are faulty, others are still accessible
- Spread load across many replicas

Replication

- Keeping a copy of the same data on multiple nodes
- Databases, filesystems, caches, . . .
- A node that has a copy of the data is called a **replica**
- If some replicas are faulty, others are still accessible
- Spread load across many replicas
- Easy if the data doesn't change: just copy it
- We will focus on data changes

Replication

- Keeping a copy of the same data on multiple nodes
- Databases, filesystems, caches, . . .
- A node that has a copy of the data is called a **replica**
- If some replicas are faulty, others are still accessible
- Spread load across many replicas
- Easy if the data doesn't change: just copy it
- We will focus on data changes

Compare to **RAID** (Redundant Array of Independent Disks): replication within a single computer

- RAID has single controller; in distributed system, each node acts independently
- Replicas can be distributed around the world, near users

Retrying state updates

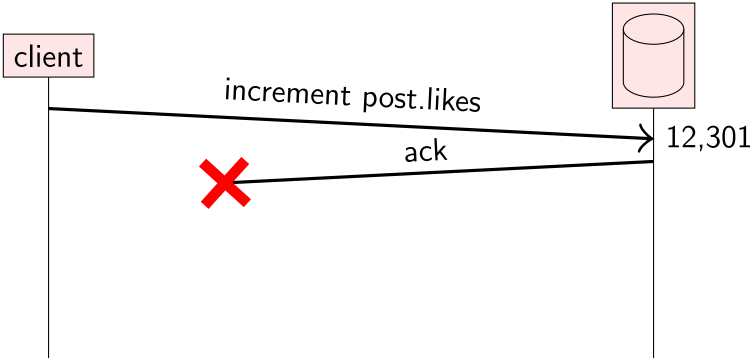
User A: The moon is not actually made of cheese!

👍 Like   12,300 people like this.

Retrying state updates

User A: The moon is not actually made of cheese!

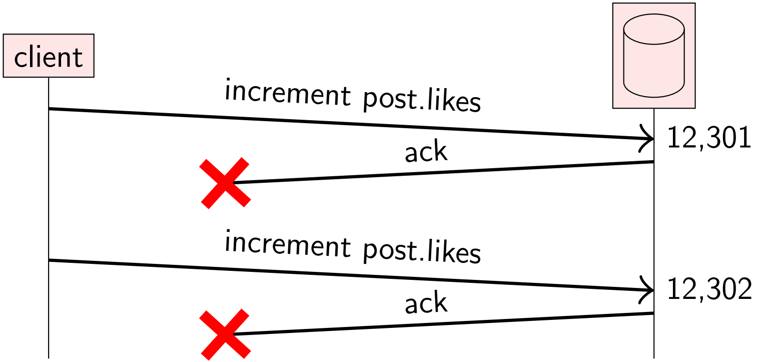
👍 Like   12,300 people like this.



Retrying state updates

User A: The moon is not actually made of cheese!

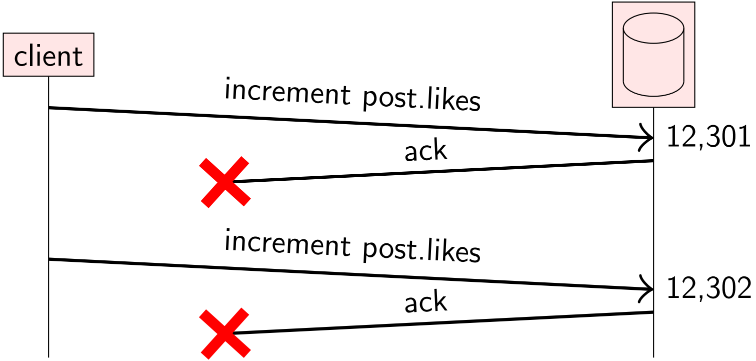
👍 Like 12,300 people like this.



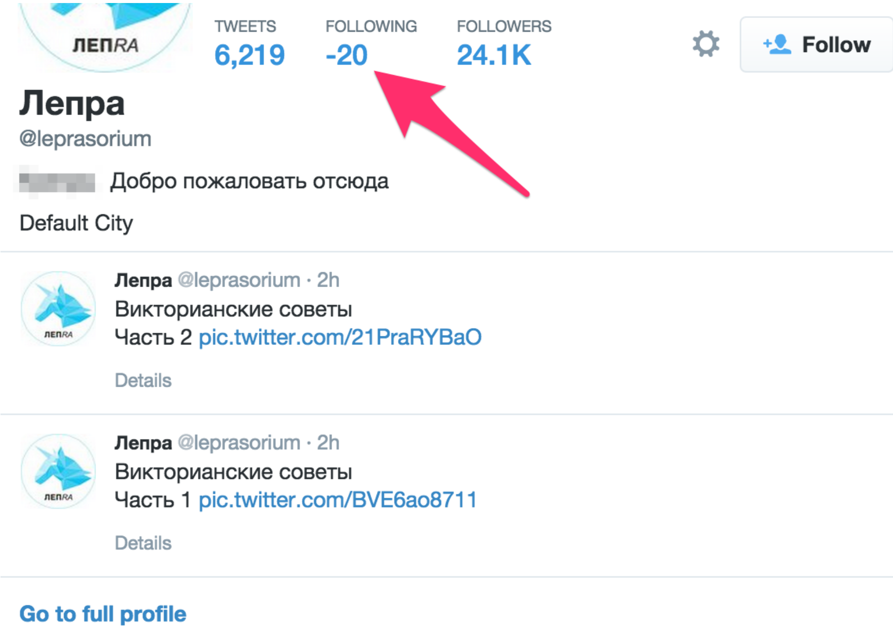
Retrying state updates

User A: The moon is not actually made of cheese!

👍 Like   12,300 people like this.



Deduplicating requests requires that the database tracks which requests it has already seen (in stable storage)



TWEETS 6,219

FOLLOWING -20

FOLLOWERS 24.1K

Лепра

@leprasorium

[redacted] Добро пожаловать отсюда

Default City

Лепра @leprasorium · 2h

Викторианские советы
Часть 2 pic.twitter.com/21PraRYBaO

Details

Лепра @leprasorium · 2h

Викторианские советы
Часть 1 pic.twitter.com/BVE6ao8711

Details

Go to full profile

Idempotence

A function f is idempotent if $f(x) = f(f(x))$.

- **Not idempotent:** $f(likeCount) = likeCount + 1$
- **Idempotent:** $f(likeSet) = likeSet \cup \{userID\}$

Idempotent requests can be retried without deduplication.

Idempotence

A function f is idempotent if $f(x) = f(f(x))$.

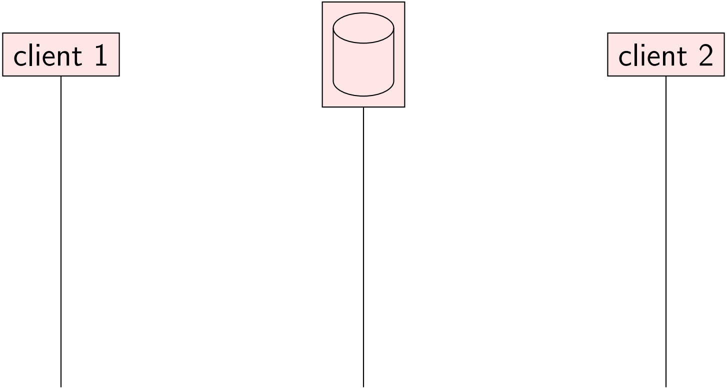
- **Not idempotent:** $f(likeCount) = likeCount + 1$
- **Idempotent:** $f(likeSet) = likeSet \cup \{userID\}$

Idempotent requests can be retried without deduplication.

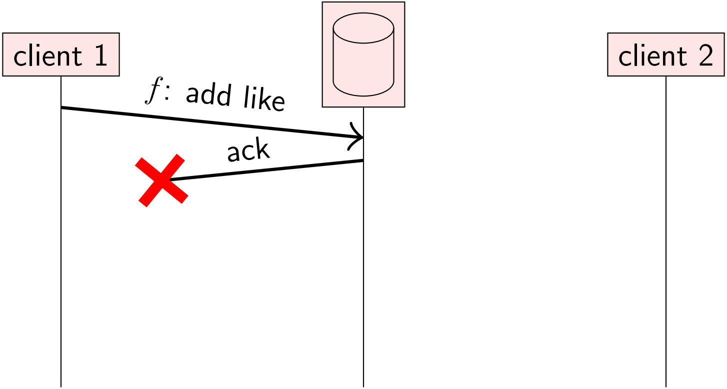
Choice of retry behaviour:

- **At-most-once** semantics:
  send request, don't retry, update may not happen
- **At-least-once** semantics:
  retry request until acknowledged, may repeat update
- **Exactly-once** semantics:
  retry $+$ idempotence or deduplication

Adding and then removing again

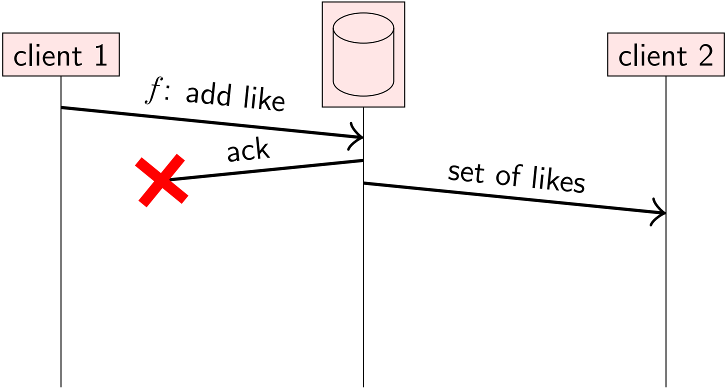


Adding and then removing again



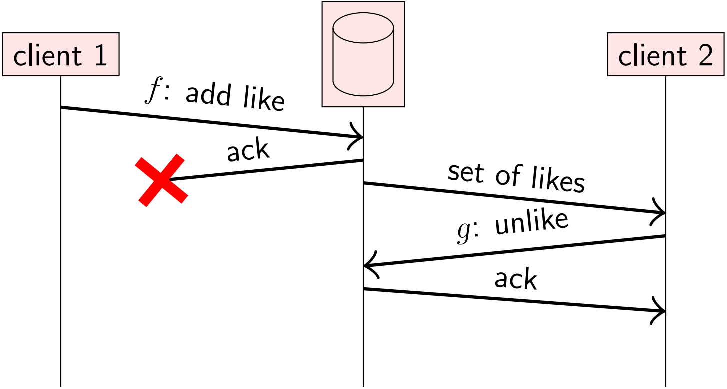
$$f(likes) = likes \cup \{userID\}$$

Adding and then removing again



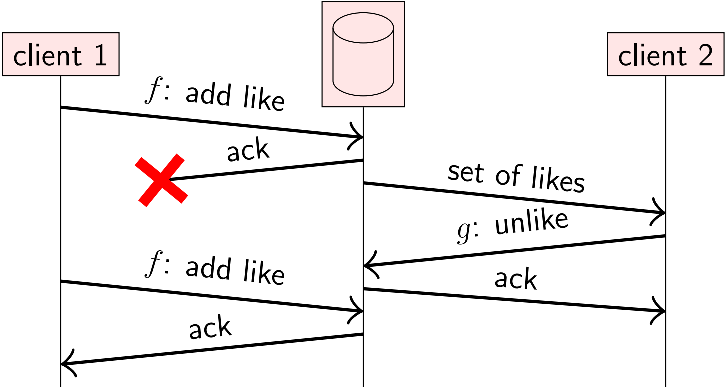
$$f(likes) = likes \cup \{userID\}$$

Adding and then removing again



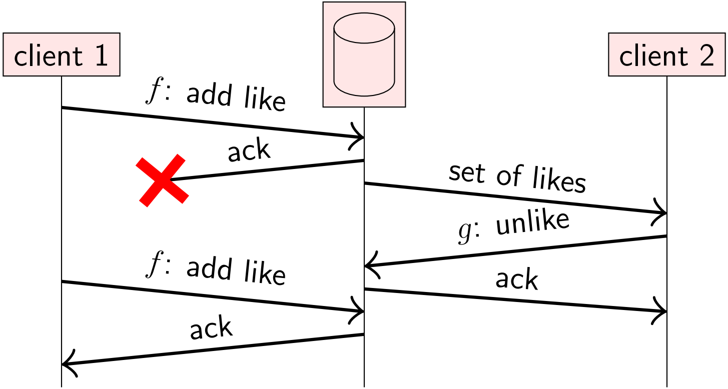
$$f(likes) = likes \cup \{userID\}$$
$$g(likes) = likes \setminus \{userID\}$$

Adding and then removing again



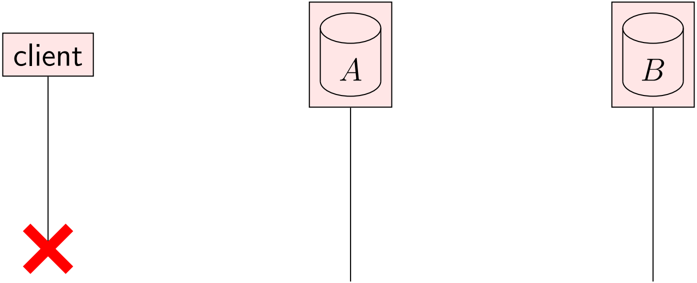
$$f(\mathit{likes}) = \mathit{likes} \cup \{\mathit{userID}\}$$
$$g(\mathit{likes}) = \mathit{likes} \setminus \{\mathit{userID}\}$$

Adding and then removing again

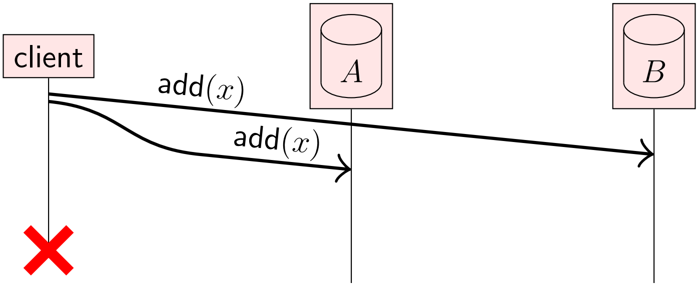


$f(likes) = likes \cup \{userID\}$
$g(likes) = likes \setminus \{userID\}$
Idempotent? $f(f(x)) = f(x)$ but $f(g(f(x)) \neq g(f(x))$

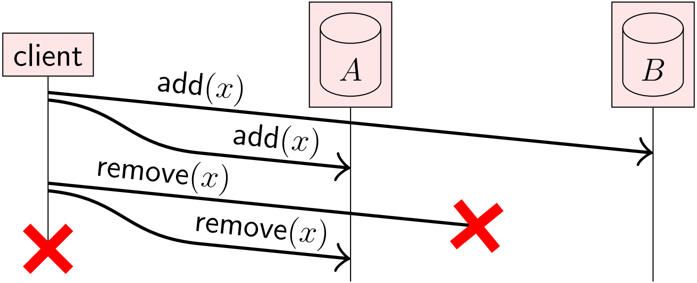
Another problem with adding and removing



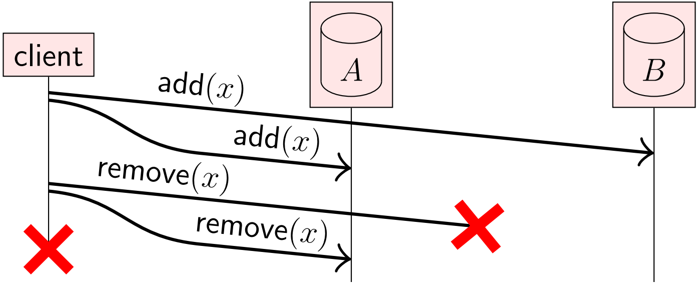
Another problem with adding and removing



Another problem with adding and removing

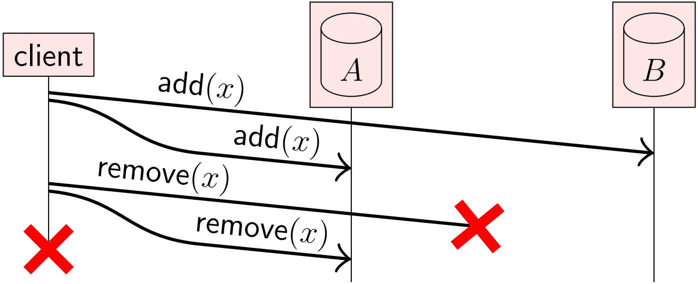


Another problem with adding and removing

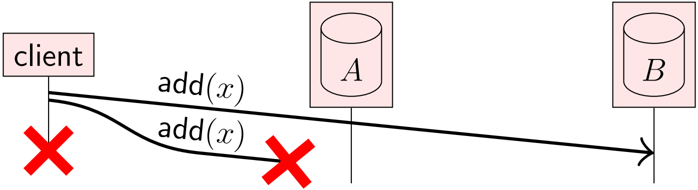


Final state ($x \notin A$, $x \in B$) is the same as in this case:

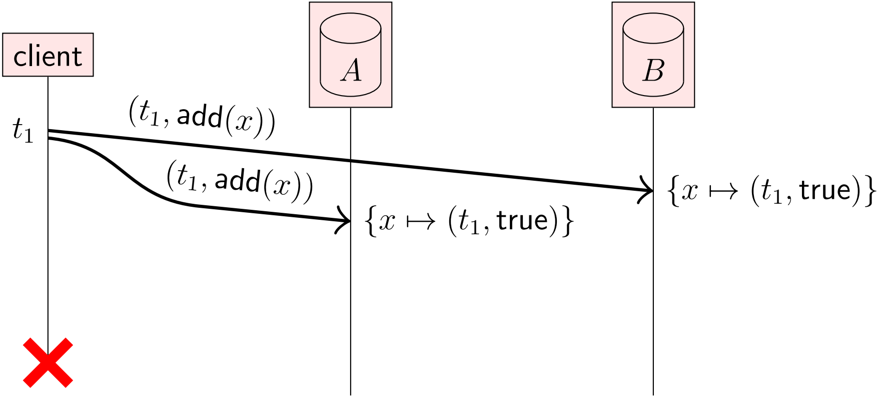
Another problem with adding and removing



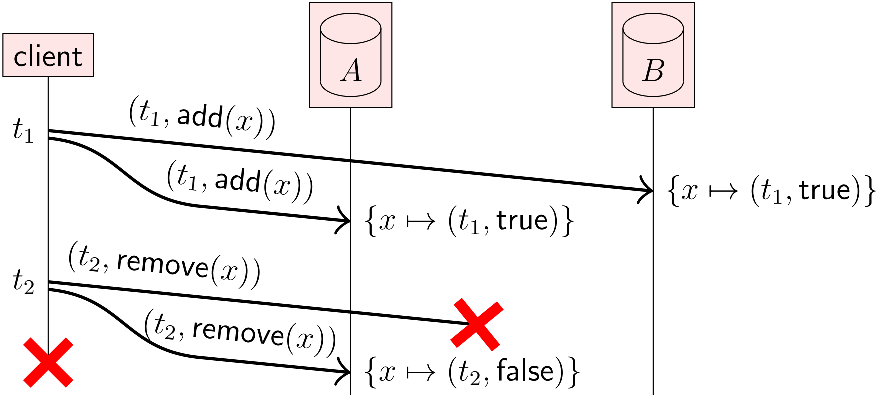
Final state ($x \notin A$, $x \in B$) is the same as in this case:



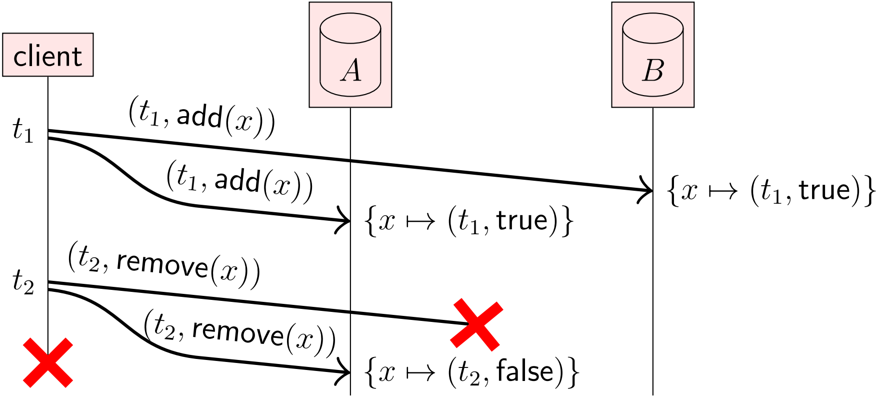
Timestamps and tombstones


client
A
B
t_1
$(t_1, \mathsf{add}(x))$
$(t_1, \mathsf{add}(x))$
$\{x \mapsto (t_1, \mathsf{true})\}$
$\{x \mapsto (t_1, \mathsf{true})\}$

Timestamps and tombstones

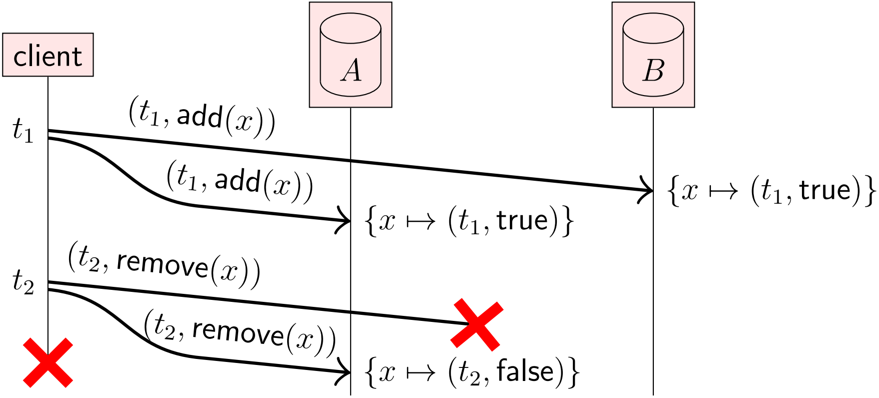

client
A
B
t_1
$(t_1, \mathsf{add}(x))$
$(t_1, \mathsf{add}(x))$
$\{x \mapsto (t_1, \mathsf{true})\}$
$\{x \mapsto (t_1, \mathsf{true})\}$
t_2
$(t_2, \mathsf{remove}(x))$
$(t_2, \mathsf{remove}(x))$
$\{x \mapsto (t_2, \mathsf{false})\}$

Timestamps and tombstones


client
A
B
t_1
$(t_1, \mathsf{add}(x))$
$(t_1, \mathsf{add}(x))$
$\{x \mapsto (t_1, \mathsf{true})\}$
$\{x \mapsto (t_1, \mathsf{true})\}$
t_2
$(t_2, \mathsf{remove}(x))$
$(t_2, \mathsf{remove}(x))$
$\{x \mapsto (t_2, \mathsf{false})\}$

"remove(x)" doesn't actually remove x: it labels x with "false" to indicate it is invisible (a **tombstone**)

Timestamps and tombstones

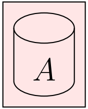

client
A
B
t_1
$(t_1, \mathsf{add}(x))$
$(t_1, \mathsf{add}(x))$
$\{x \mapsto (t_1, \mathsf{true})\}$
$\{x \mapsto (t_1, \mathsf{true})\}$
t_2
$(t_2, \mathsf{remove}(x))$
$(t_2, \mathsf{remove}(x))$
$\{x \mapsto (t_2, \mathsf{false})\}$

"remove(x)" doesn't actually remove x: it labels x with "false" to indicate it is invisible (a **tombstone**)

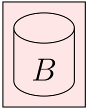
Every record has **logical timestamp** of last write

Reconciling replicas

Replicas periodically communicate among themselves to check for any inconsistencies.



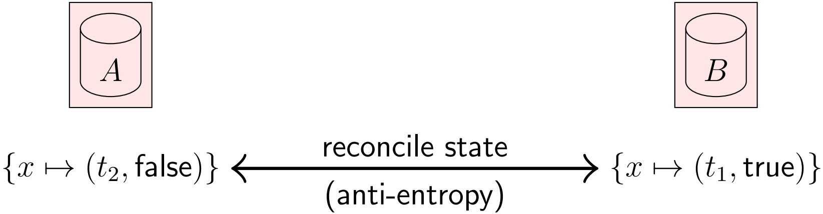
$\{x \mapsto (t_2, \textsf{false})\}$



$\{x \mapsto (t_1, \textsf{true})\}$

Reconciling replicas

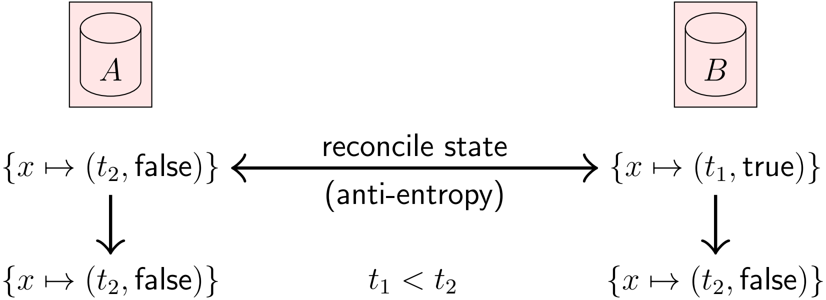
Replicas periodically communicate among themselves to check for any inconsistencies.



$\{x \mapsto (t_2, \text{false})\}$ reconcile state (anti-entropy) $\{x \mapsto (t_1, \text{true})\}$

Reconciling replicas

Replicas periodically communicate among themselves to check for any inconsistencies.



A

B

$\{x \mapsto (t_2, \text{false})\}$

reconcile state

(anti-entropy)

$\{x \mapsto (t_1, \text{true})\}$

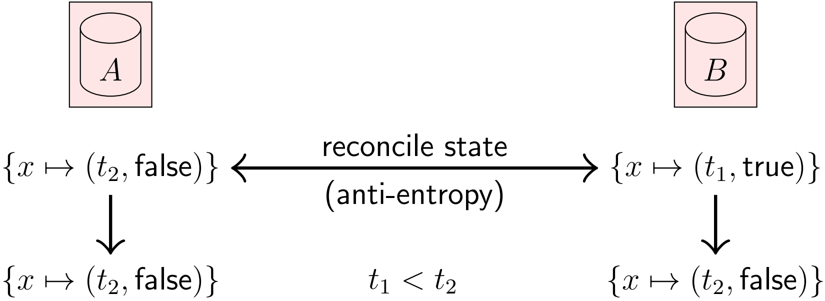
$\{x \mapsto (t_2, \text{false})\}$

$t_1 < t_2$

$\{x \mapsto (t_2, \text{false})\}$

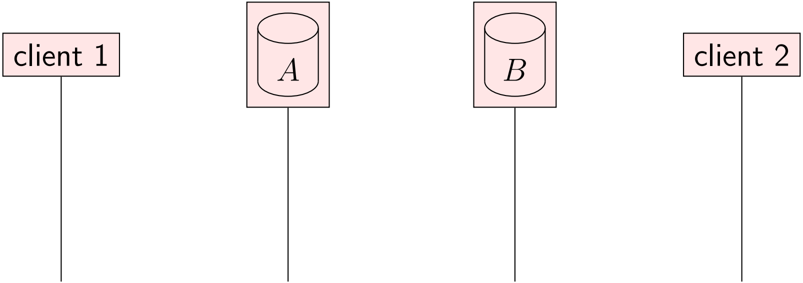
Reconciling replicas

Replicas periodically communicate among themselves
to check for any inconsistencies.

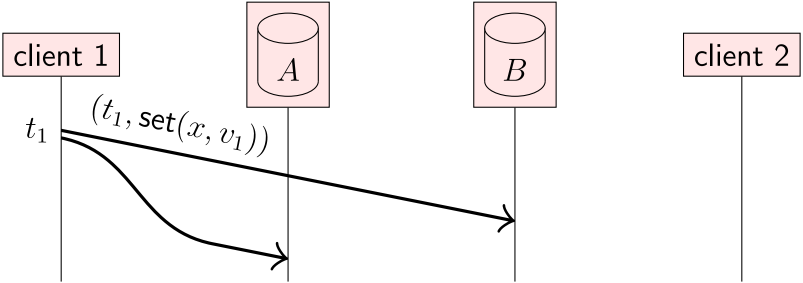

A
B
$\{x \mapsto (t_2, \text{false})\}$
reconcile state
(anti-entropy)
$\{x \mapsto (t_1, \text{true})\}$
$\{x \mapsto (t_2, \text{false})\}$
$t_1 < t_2$
$\{x \mapsto (t_2, \text{false})\}$

Propagate the record with the latest timestamp,
discard the records with earlier timestamps
(for a given key).

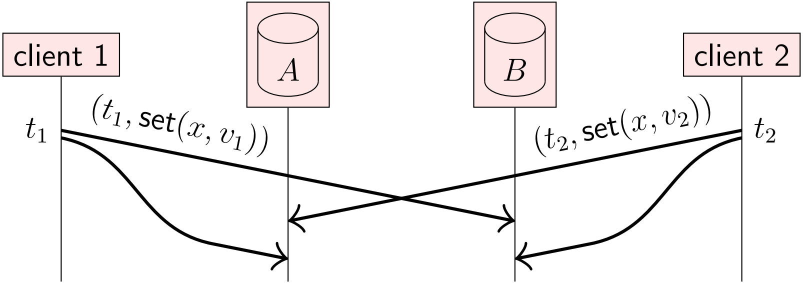
Concurrent writes by different clients



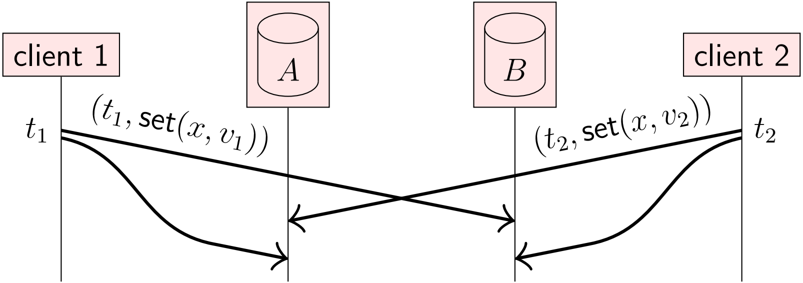
Concurrent writes by different clients



Concurrent writes by different clients



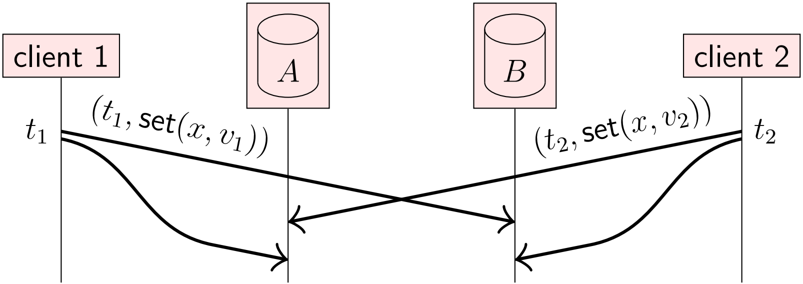
Concurrent writes by different clients



Two common approaches:

- **Last writer wins** (LWW):
  Use timestamps with total order (e.g. Lamport clock)
  Keep v_2 and discard v_1 if $t_2 > t_1$. Note: **data loss**!

Concurrent writes by different clients



Two common approaches:

- **Last writer wins** (LWW):
  Use timestamps with total order (e.g. Lamport clock)
  Keep v_2 and discard v_1 if $t_2 > t_1$. Note: **data loss**!
- **Multi-value register**:
  Use timestamps with partial order (e.g. vector clock)
  v_2 replaces v_1 if $t_2 > t_1$; preserve both $\{v_1, v_2\}$ if $t_1 \parallel t_2$

Probability of faults

A replica may be **unavailable** due to network partition or node fault (e.g. crash, hardware problem).

Probability of faults

A replica may be **unavailable** due to network partition or node fault (e.g. crash, hardware problem).

Assume each replica has probability p of being faulty or unavailable at any one time, and that faults are independent.
(Not actually true! But okay approximation for now.)

Probability of faults

A replica may be **unavailable** due to network partition or node fault (e.g. crash, hardware problem).

Assume each replica has probability p of being faulty or unavailable at any one time, and that faults are independent.
(Not actually true! But okay approximation for now.)

Probability of **all** n replicas being faulty: p^n
Probability of ≥ 1 out of n replicas being faulty: $1 - (1 - p)^n$

Probability of faults

A replica may be **unavailable** due to network partition or node fault (e.g. crash, hardware problem).

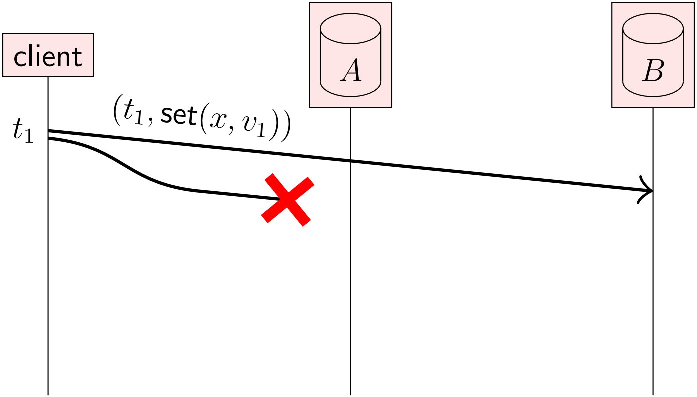
Assume each replica has probability p of being faulty or unavailable at any one time, and that faults are independent. (Not actually true! But okay approximation for now.)

Probability of **all** n replicas being faulty: p^n
Probability of ≥ 1 out of n replicas being faulty: $1 - (1 - p)^n$

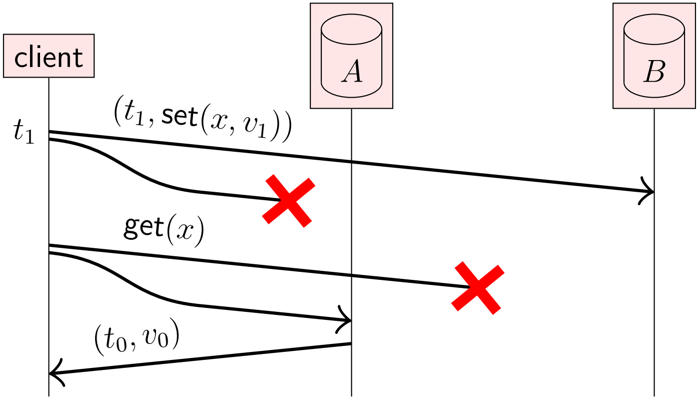
Example with $p = 0.01$:

replicas n	$P(\geq 1$ faulty$)$	$P(\geq \frac{n+1}{2}$ faulty$)$	$P($all n faulty$)$
1	0.01	0.01	0.01
3	0.03	$3 \cdot 10^{-4}$	10^{-6}
5	0.049	$1 \cdot 10^{-5}$	10^{-10}
100	0.63	$6 \cdot 10^{-74}$	10^{-200}

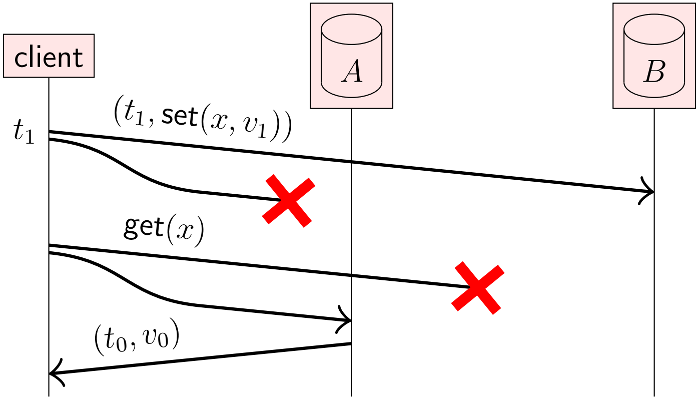
Read-after-write consistency



Read-after-write consistency

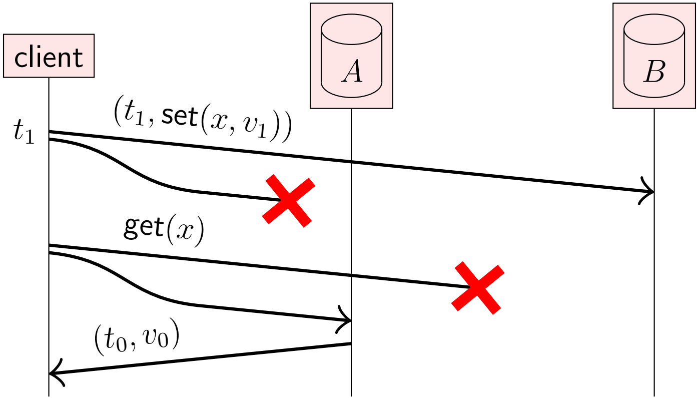


Read-after-write consistency



Writing to one replica, reading from another: client does not read back the value it has written

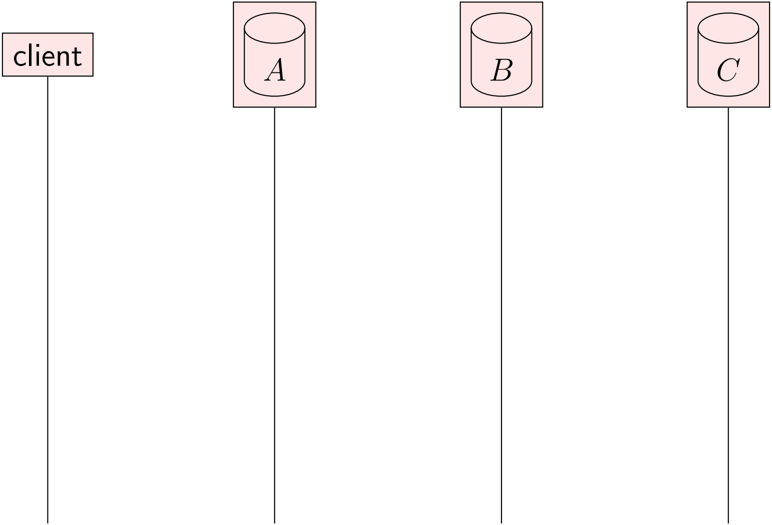
Read-after-write consistency



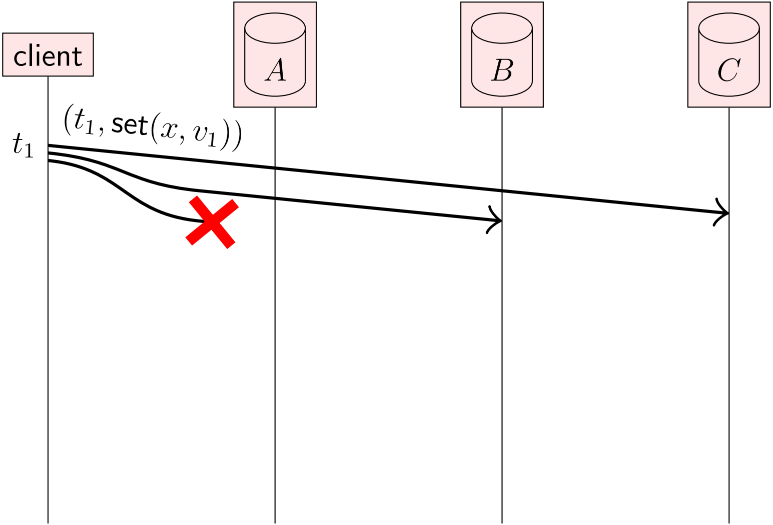
Writing to one replica, reading from another: client does not read back the value it has written

Require writing to/reading from both replicas $\Longrightarrow$ cannot write/read if one replica is unavailable

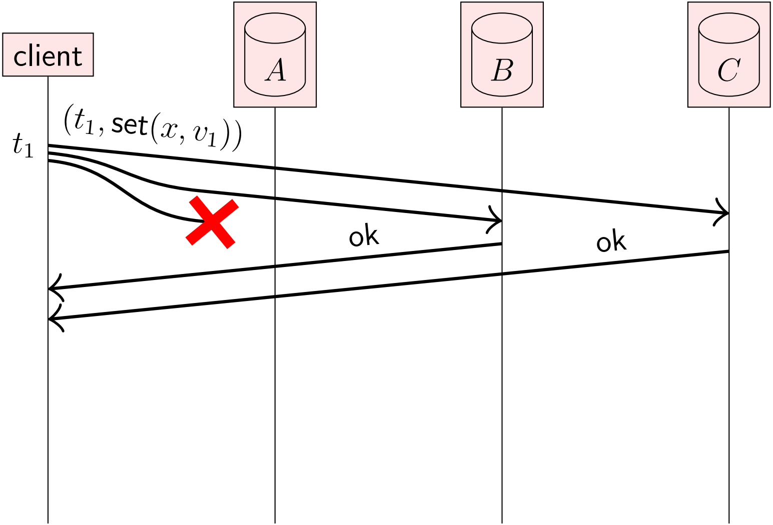
Quorum (2 out of 3)



Quorum (2 out of 3)

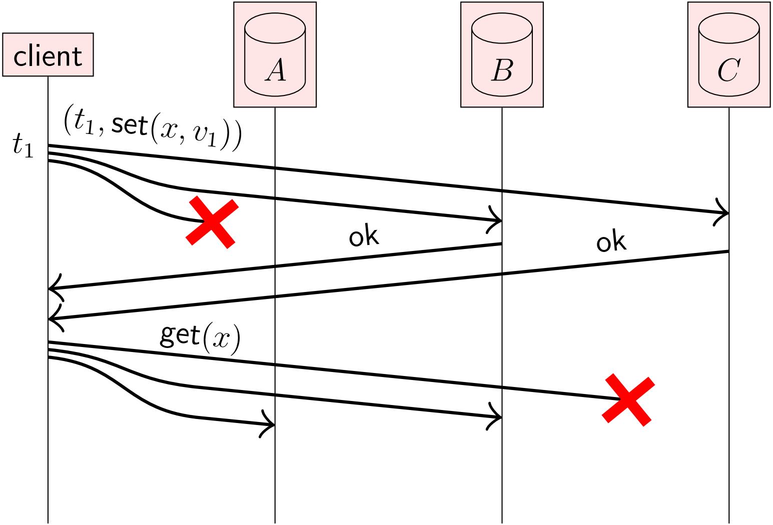


Quorum (2 out of 3)



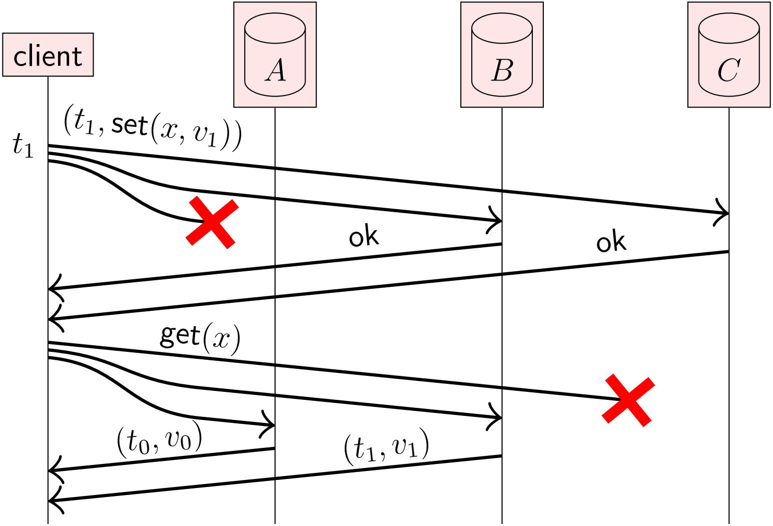
Write succeeds on B and C

Quorum (2 out of 3)



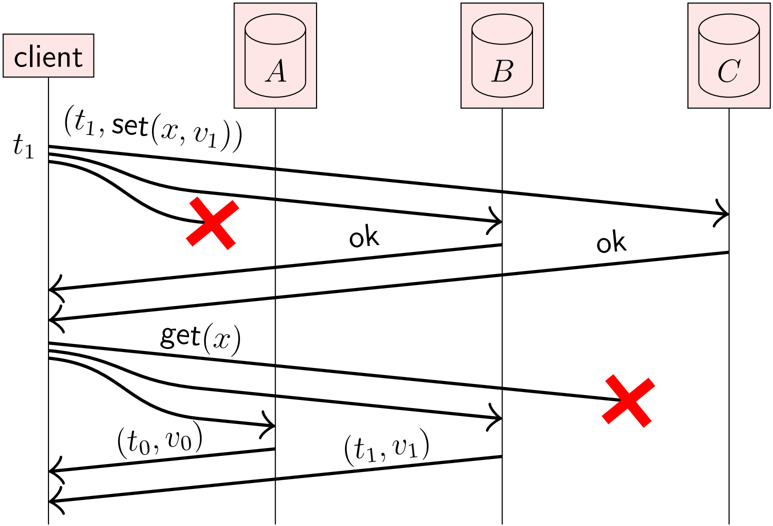
Write succeeds on B and C

Quorum (2 out of 3)



Write succeeds on B and C; read succeeds on A and B

Quorum (2 out of 3)



Write succeeds on B and C; read succeeds on A and B

Choose between (t_0, v_0) and (t_1, v_1) based on timestamp

Read and write quorums

In a system with n replicas:

- If a write is acknowledged by w replicas (**write quorum**),

Read and write quorums

In a system with n replicas:

- If a write is acknowledged by w replicas (**write quorum**),
- and we subsequently read from r replicas (**read quorum**),
- and $r + w > n$,

Read and write quorums

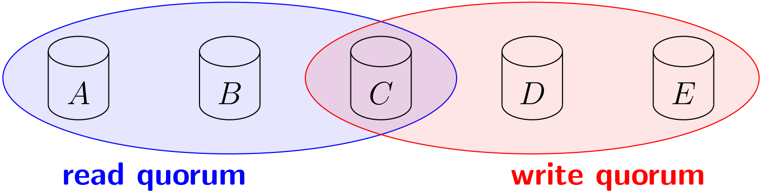
In a system with n replicas:

- If a write is acknowledged by w replicas (**write quorum**),
- and we subsequently read from r replicas (**read quorum**),
- and $r + w > n$,
- …then the read will see the previously written value (or a value that subsequently overwrote it)

Read and write quorums

In a system with n replicas:

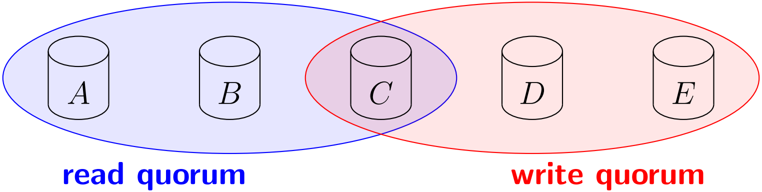
- If a write is acknowledged by w replicas (**write quorum**),
- and we subsequently read from r replicas (**read quorum**),
- and $r + w > n$,
- . . . then the read will see the previously written value (or a value that subsequently overwrote it)
- Read quorum and write quorum share ≥ 1 replica



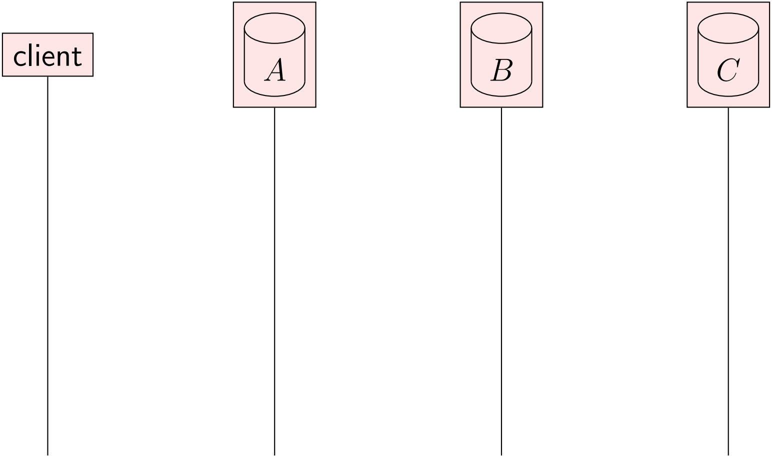
Read and write quorums

In a system with n replicas:

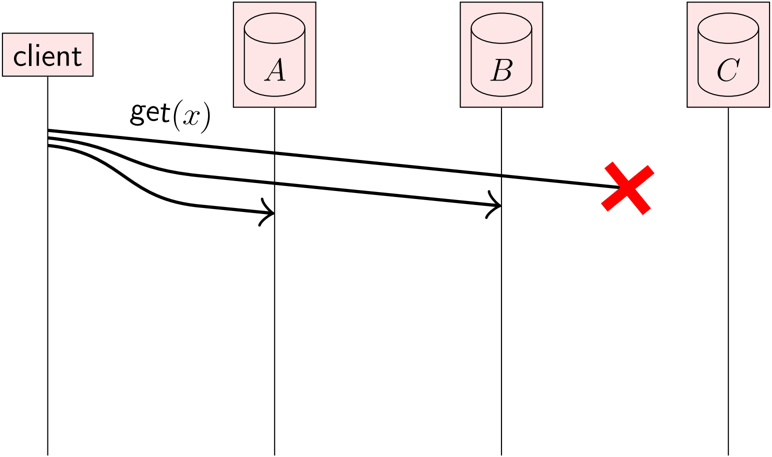
- If a write is acknowledged by w replicas (**write quorum**),
- and we subsequently read from r replicas (**read quorum**),
- and $r + w > n$,
- …then the read will see the previously written value (or a value that subsequently overwrote it)
- Read quorum and write quorum share ≥ 1 replica
- Typical: $r = w = \frac{n+1}{2}$ for $n = 3, 5, 7, \ldots$ (majority)
- Reads can tolerate $n - r$ unavailable replicas, writes $n - w$



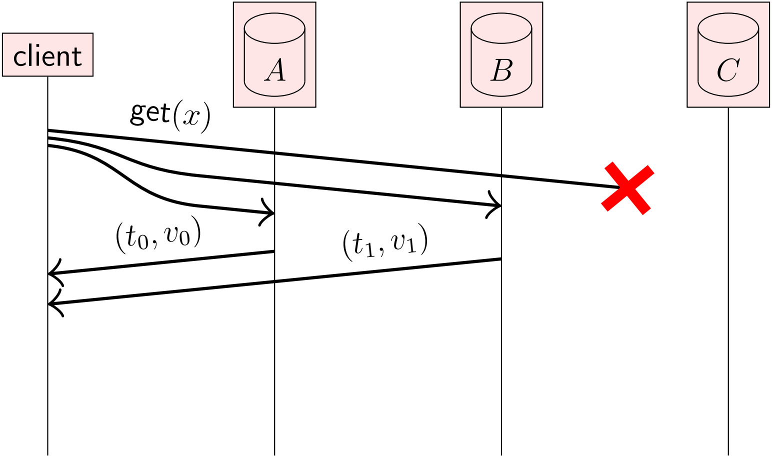
Read repair



Read repair

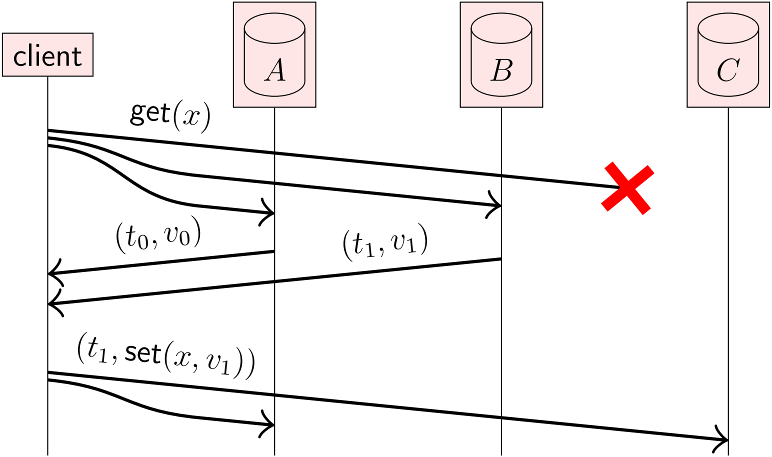


Read repair



Update (t_1, v_1) is more recent than (t_0, v_0) since $t_0 < t_1$.

Read repair



Update (t_1, v_1) is more recent than (t_0, v_0) since $t_0 < t_1$.
Client helps **propagate** (t_1, v_1) to other replicas.

State machine replication

So far we have used best-effort broadcast for replication.
What about stronger broadcast models?

State machine replication

So far we have used best-effort broadcast for replication. What about stronger broadcast models?

Total order broadcast: every node delivers the **same messages** in the **same order**

State machine replication

So far we have used best-effort broadcast for replication. What about stronger broadcast models?

Total order broadcast: every node delivers the **same messages** in the **same order**

State machine replication (SMR):

- FIFO-total order broadcast every update to all replicas
- Replica delivers update message: apply it to own state

State machine replication

So far we have used best-effort broadcast for replication. What about stronger broadcast models?

Total order broadcast: every node delivers the **same messages** in the **same order**

State machine replication (SMR):

- FIFO-total order broadcast every update to all replicas
- Replica delivers update message: apply it to own state
- Applying an update is deterministic

State machine replication

So far we have used best-effort broadcast for replication. What about stronger broadcast models?

Total order broadcast: every node delivers the **same messages** in the **same order**

State machine replication (SMR):

- FIFO-total order broadcast every update to all replicas
- Replica delivers update message: apply it to own state
- Applying an update is deterministic
- Replica is a **state machine**: starts in fixed initial state, goes through same sequence of state transitions in the same order $\Longrightarrow$ all replicas end up in the same state

State machine replication

```
on request to perform update $u$ do
    send $u$ via FIFO-total order broadcast
end on

on delivering $u$ through FIFO-total order broadcast do
    update state using arbitrary deterministic logic!
end on
```

State machine replication

```
on request to perform update $u$ do
    send $u$ via FIFO-total order broadcast
end on

on delivering $u$ through FIFO-total order broadcast do
    update state using arbitrary deterministic logic!
end on
```

Closely related ideas:

- Serializable transactions (execute in delivery order)

State machine replication

```
on request to perform update $u$ do
    send $u$ via FIFO-total order broadcast
end on

on delivering $u$ through FIFO-total order broadcast do
    update state using arbitrary deterministic logic!
end on
```

Closely related ideas:

- Serializable transactions (execute in delivery order)
- Blockchains, distributed ledgers, smart contracts

State machine replication

```
on request to perform update $u$ do
    send $u$ via FIFO-total order broadcast
end on

on delivering $u$ through FIFO-total order broadcast do
    update state using arbitrary deterministic logic!
end on
```

Closely related ideas:

- Serializable transactions (execute in delivery order)
- Blockchains, distributed ledgers, smart contracts

Limitations:

- Cannot update state immediately, have to wait for delivery through broadcast

State machine replication

```
on request to perform update $u$ do
    send $u$ via FIFO-total order broadcast
end on

on delivering $u$ through FIFO-total order broadcast do
    update state using arbitrary deterministic logic!
end on
```

Closely related ideas:

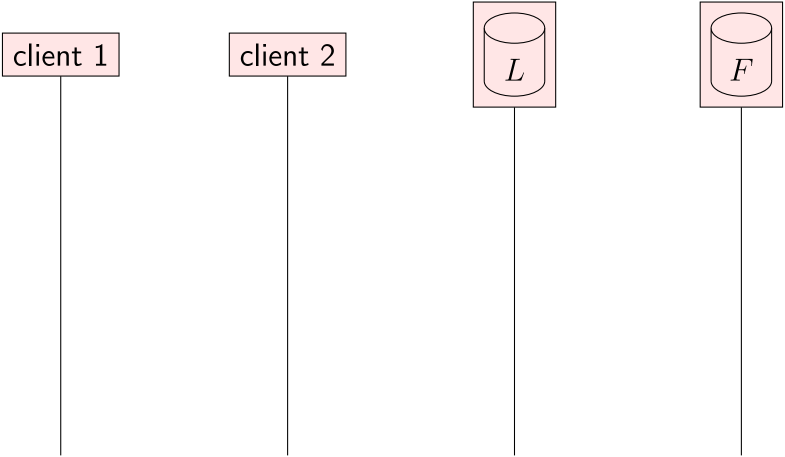
- Serializable transactions (execute in delivery order)
- Blockchains, distributed ledgers, smart contracts

Limitations:

- Cannot update state immediately, have to wait for delivery through broadcast
- Need fault-tolerant total order broadcast: see lecture 6

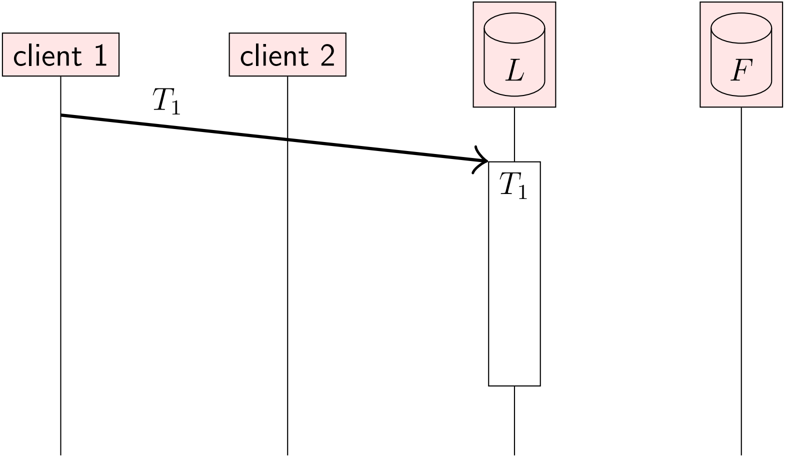
Database leader replica

Leader database replica L ensures total order broadcast



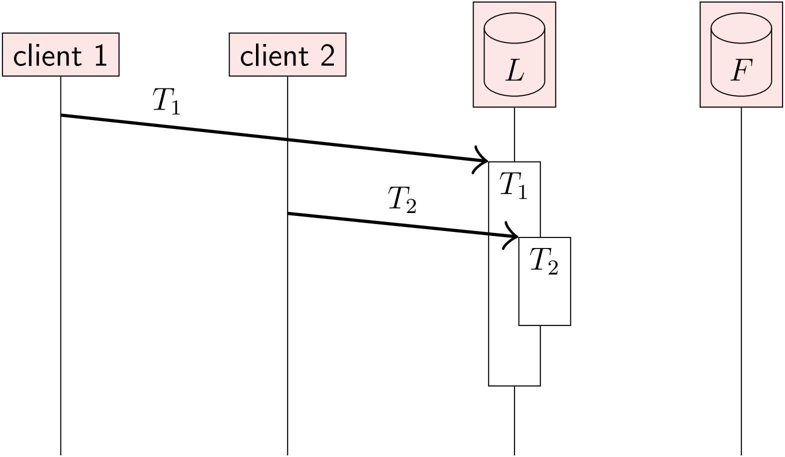
Database leader replica

Leader database replica L ensures total order broadcast



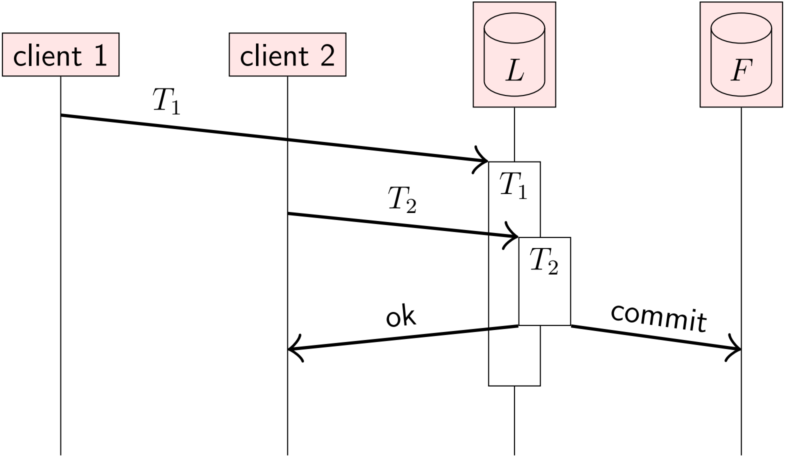
Database leader replica

Leader database replica L ensures total order broadcast



Database leader replica

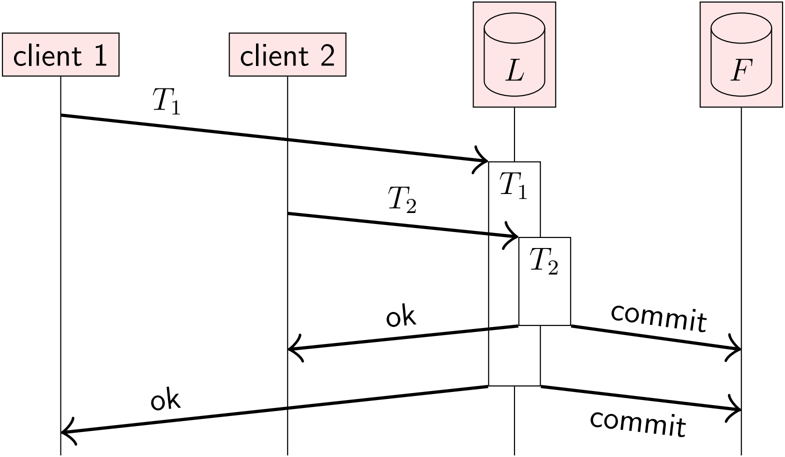
Leader database replica L ensures total order broadcast



Follower F applies transaction log in commit order

Database leader replica

Leader database replica L ensures total order broadcast



Follower F applies transaction log in commit order

Replication using causal (and weaker) broadcast

State machine replication uses (FIFO-)total order broadcast.
Can we use weaker forms of broadcast too?

Replication using causal (and weaker) broadcast

State machine replication uses (FIFO-)total order broadcast. Can we use weaker forms of broadcast too?

If replica state updates are **commutative**, replicas can process updates in different orders and still end up in the same state.

Updates f and g are commutative if $f(g(x)) = g(f(x))$

Replication using causal (and weaker) broadcast

State machine replication uses (FIFO-)total order broadcast. Can we use weaker forms of broadcast too?

If replica state updates are **commutative**, replicas can process updates in different orders and still end up in the same state.

Updates f and g are commutative if $f(g(x)) = g(f(x))$

broadcast	**assumptions about state update function**
total order	deterministic (SMR)

Replication using causal (and weaker) broadcast

State machine replication uses (FIFO-)total order broadcast. Can we use weaker forms of broadcast too?

If replica state updates are **commutative**, replicas can process updates in different orders and still end up in the same state.

Updates f and g are commutative if $f(g(x)) = g(f(x))$

broadcast	**assumptions about state update function**
total order	deterministic (SMR)
causal	deterministic, concurrent updates commute

Replication using causal (and weaker) broadcast

State machine replication uses (FIFO-)total order broadcast. Can we use weaker forms of broadcast too?

If replica state updates are **commutative**, replicas can process updates in different orders and still end up in the same state.

Updates f and g are commutative if $f(g(x)) = g(f(x))$

broadcast	**assumptions about state update function**
total order	deterministic (SMR)
causal	deterministic, concurrent updates commute
reliable	deterministic, all updates commute

Replication using causal (and weaker) broadcast

State machine replication uses (FIFO-)total order broadcast. Can we use weaker forms of broadcast too?

If replica state updates are **commutative**, replicas can process updates in different orders and still end up in the same state.

Updates f and g are commutative if $f(g(x)) = g(f(x))$

broadcast	**assumptions about state update function**
total order	deterministic (SMR)
causal	deterministic, concurrent updates commute
reliable	deterministic, all updates commute
best-effort	deterministic, commutative, idempotent, tolerates message loss