Motivation

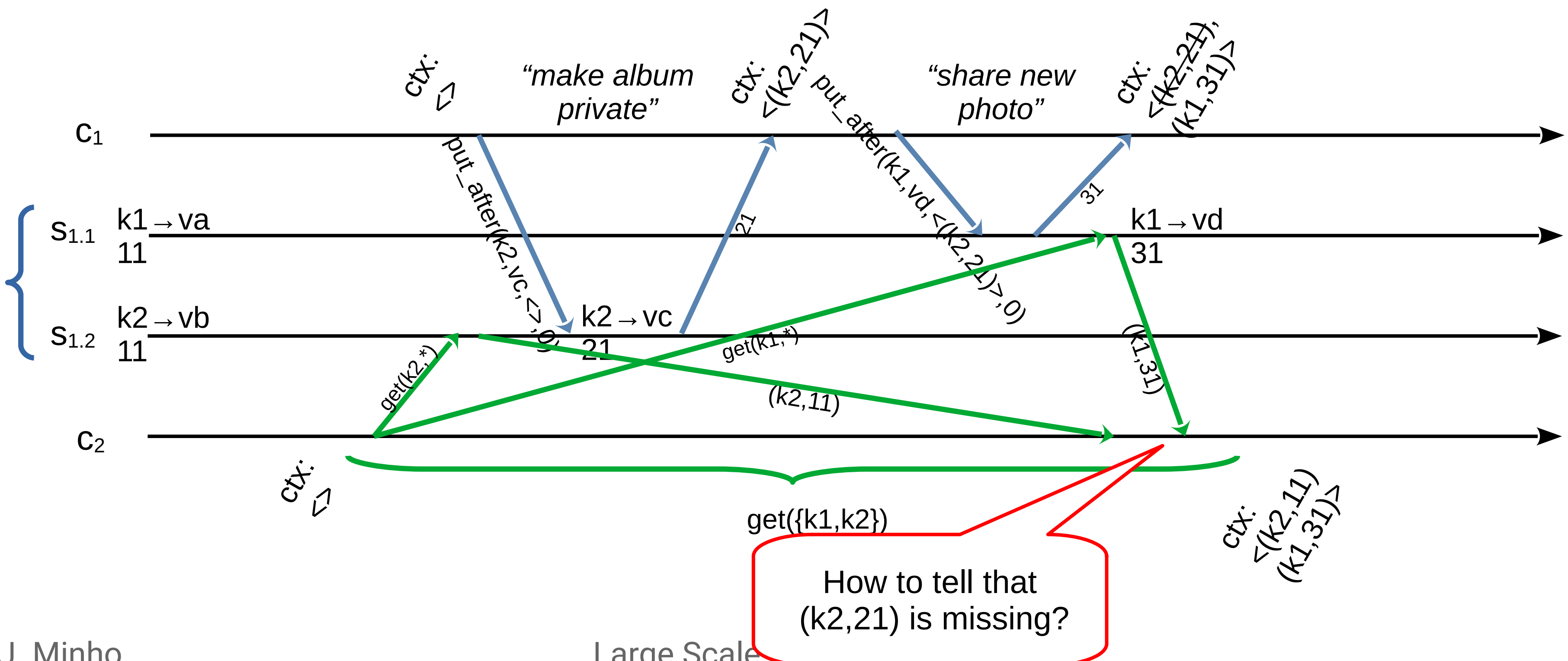
- Example: Read a photo album and its access control list
- Option 1, concurrently:
  - Read photos, then read ACL
  - Delete private photo, make album public
    - Might read private photo and see it as public
- Option 2, concurrently:
  - Read ACL, then read photos
  - Make album private, add private photo
    - Might see album as public and then read private photo

Transactions

- Read transactions:
  - Avoid missing dependencies in values read
  - Solves the problem if writes issued in the correct order
- Write transactions:
  - Ensures atomicity (mutual dependency) of values written
  - Allows any write order

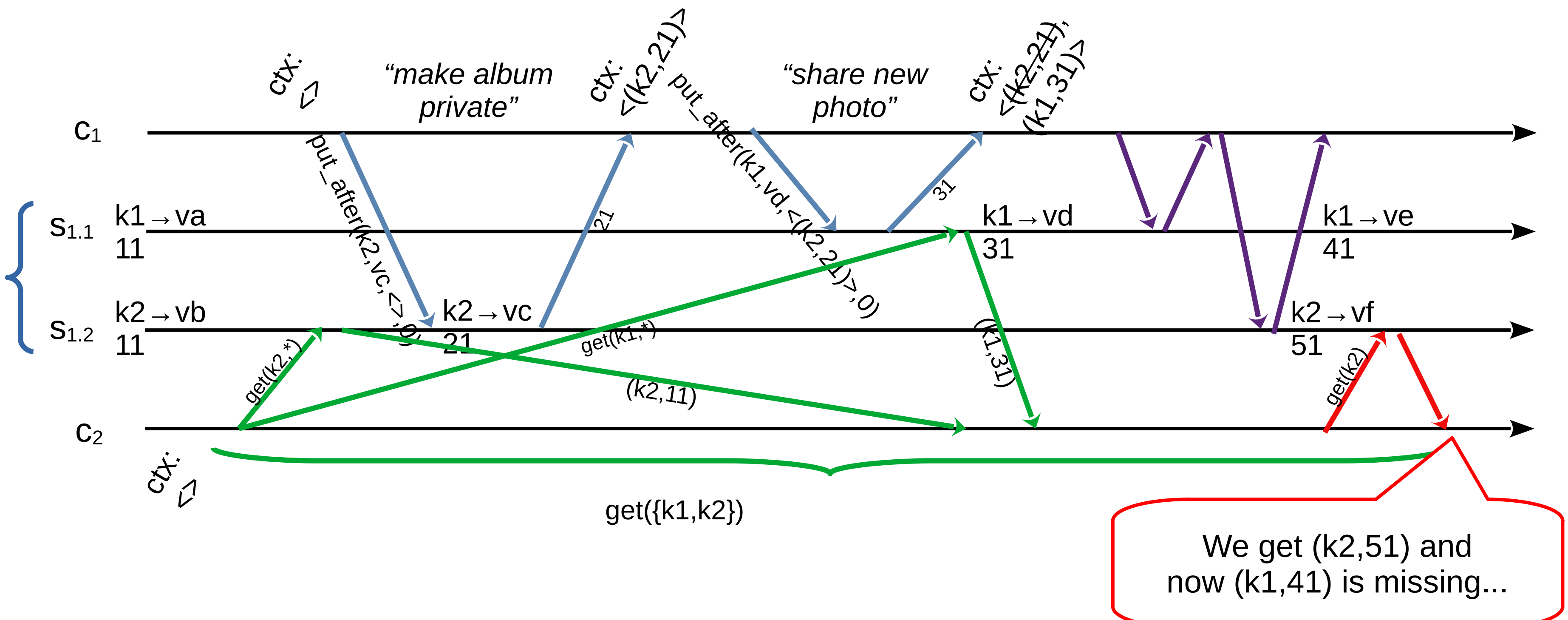
Read transactions: Challenges

- Dependency information is not stored in the server or returned to clients



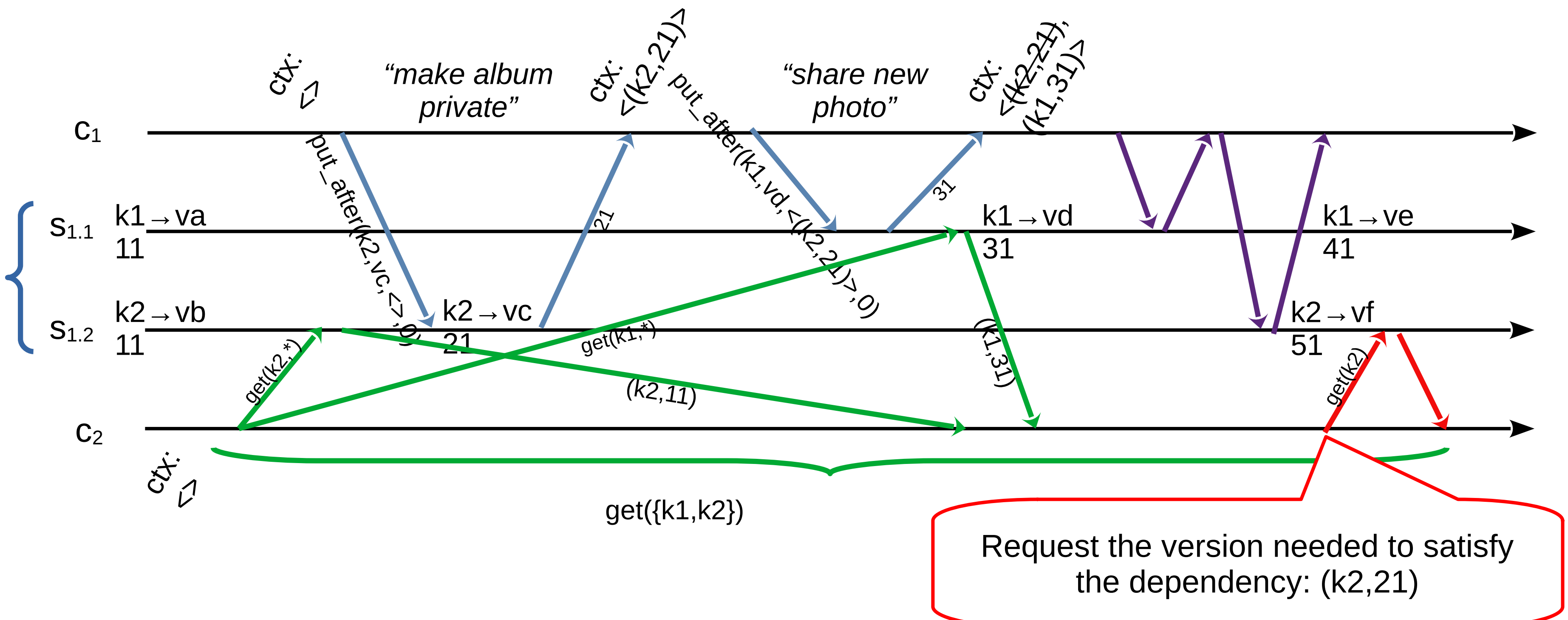
Stored dependencies

- Keep dependencies to detect what is missing
- How to get it without introducing new dependencies?



Multi-version

- <u>Keep preceding versions</u> of each item
- Return the exact version needed by the transaction



Read from snapshot

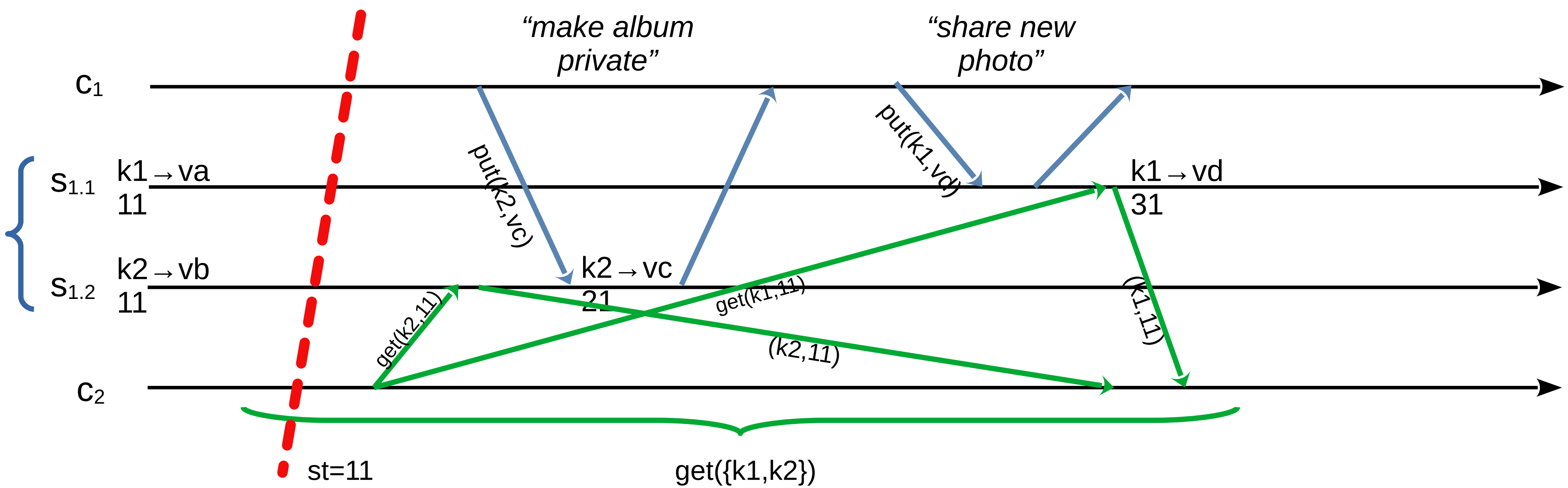
- How to avoid keeping detailed dependency information?
- The problem arises when reading some item that was written after the transaction has started:
  - (k1,31) in the example

  as it may introduce a dependency:
  - (k2,21) in the example

- <u>Stable time</u>: Latest time for which all dependencies are known

Read from snapshot

- Obtain a <u>stable</u> start timestamp *st* at the <u>start</u> of the transaction
- Read latest version $v \leq st$

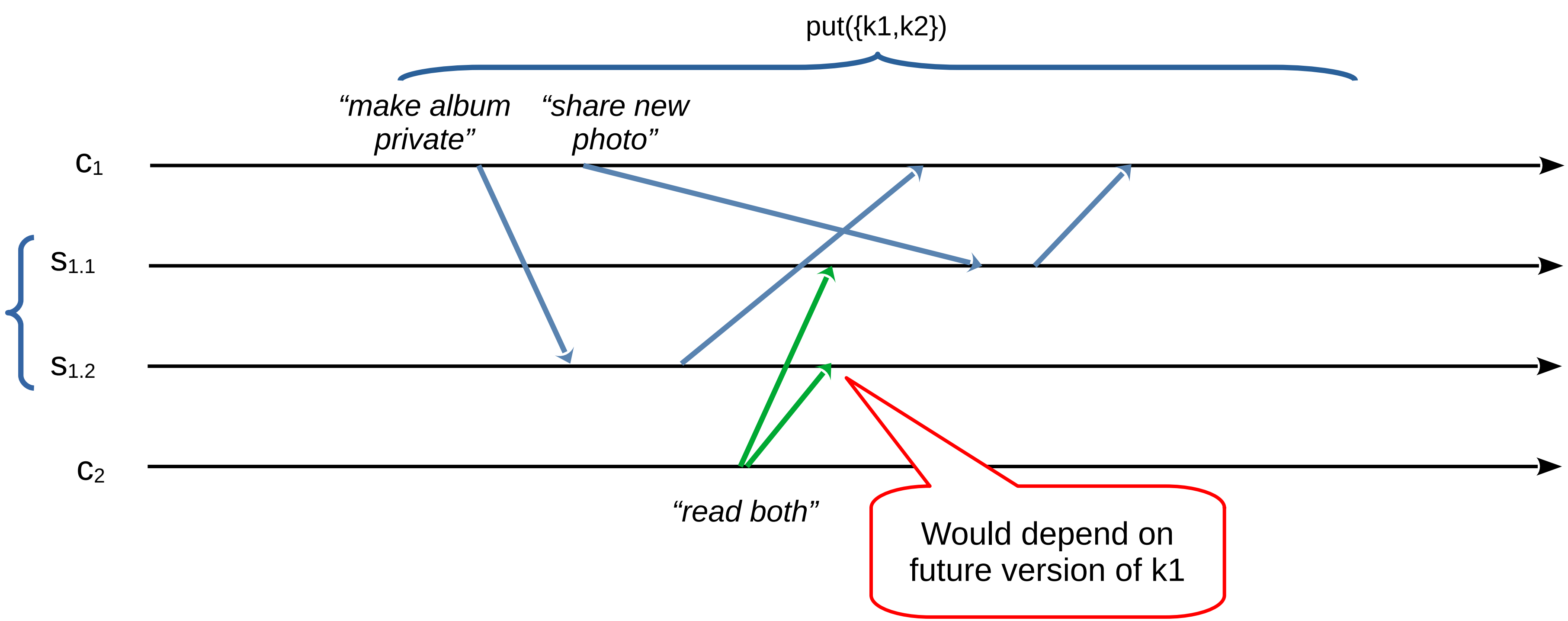


Snapshot assignment

- Start timesampt *st* is stable: all updates (hence, their dependencies) are available in all hosts
- Start timestamp *st* is greater than all previous reads and writes from the same client
  - Ensures RYOW and causality across different transactions
  - May <u>need to wait</u> for stability to catch up to recent updates
- The latest stable snapshot can be computed with <u>epidemic protocols</u>

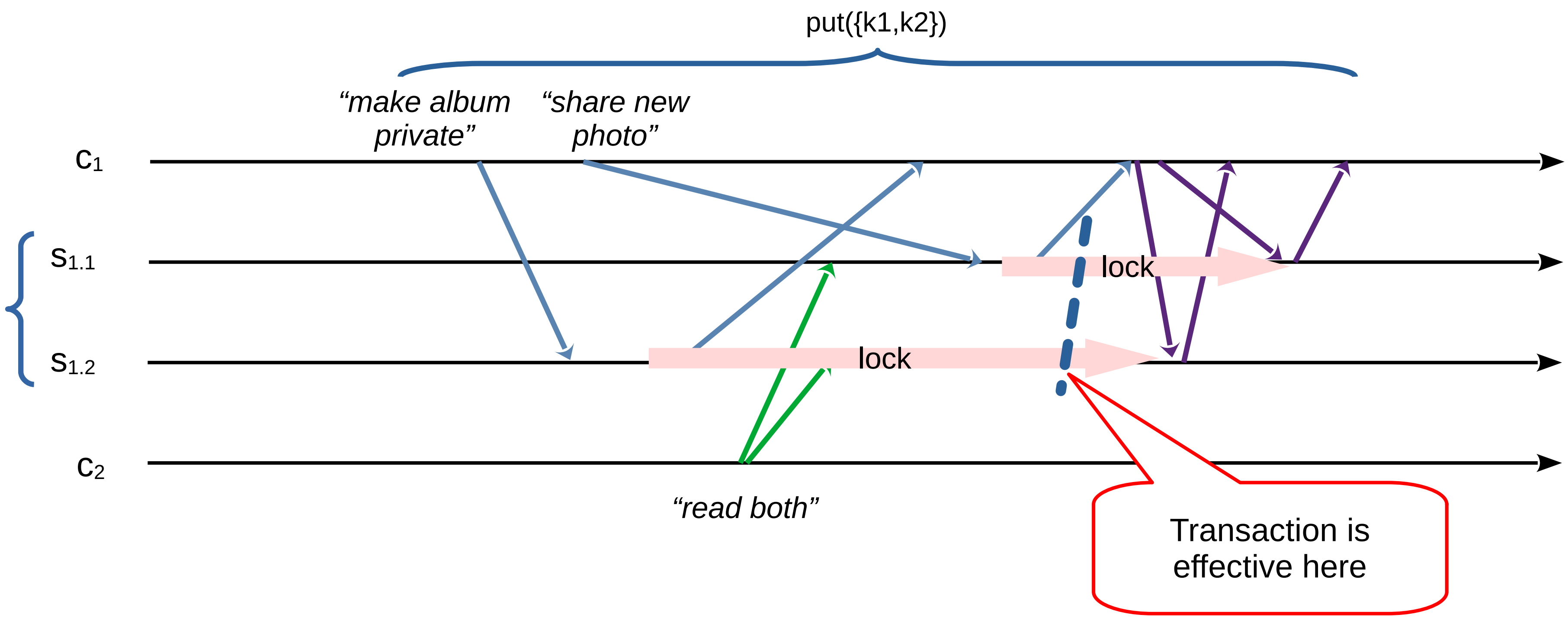
Write transactions: Challenges

- Reads assume that all dependencies are known and committed
  - Not true with write transactions



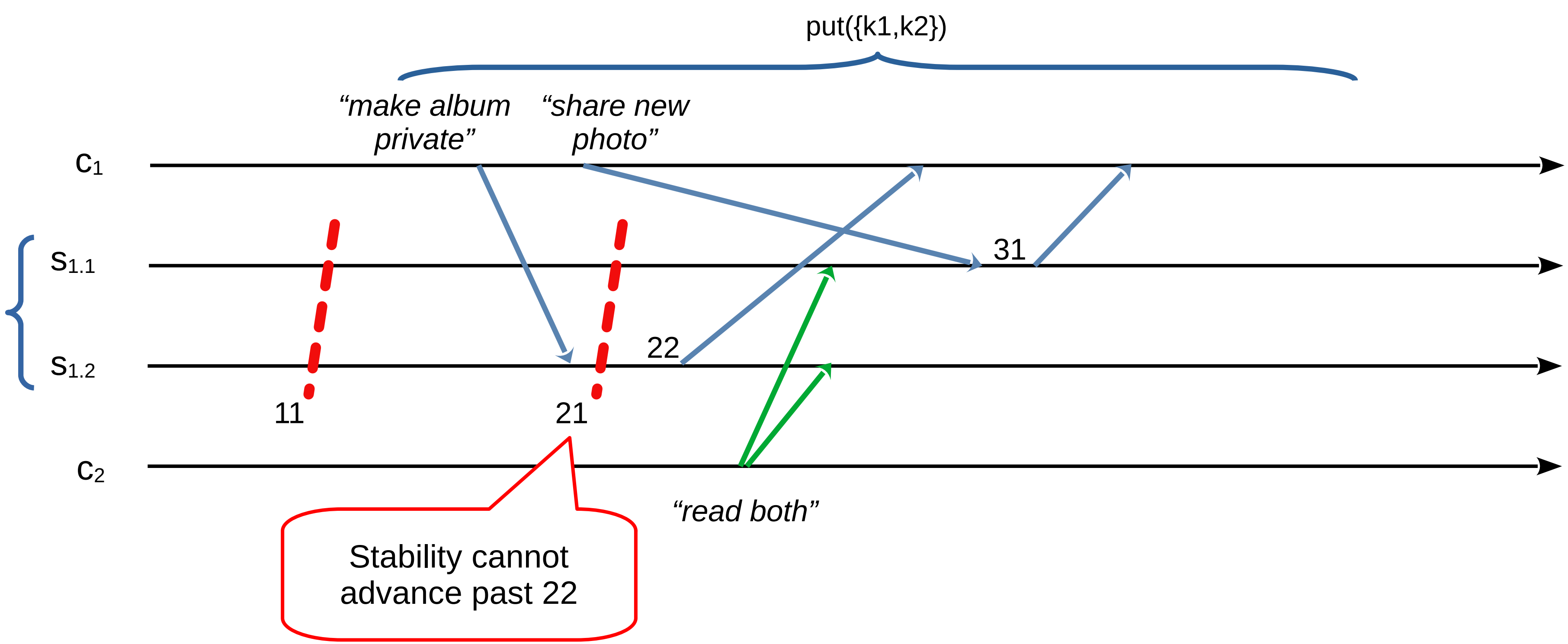
Two phase commit

- An atomic update of two servers needs 2-phase commit
- Locking avoids reading incomplete transactions



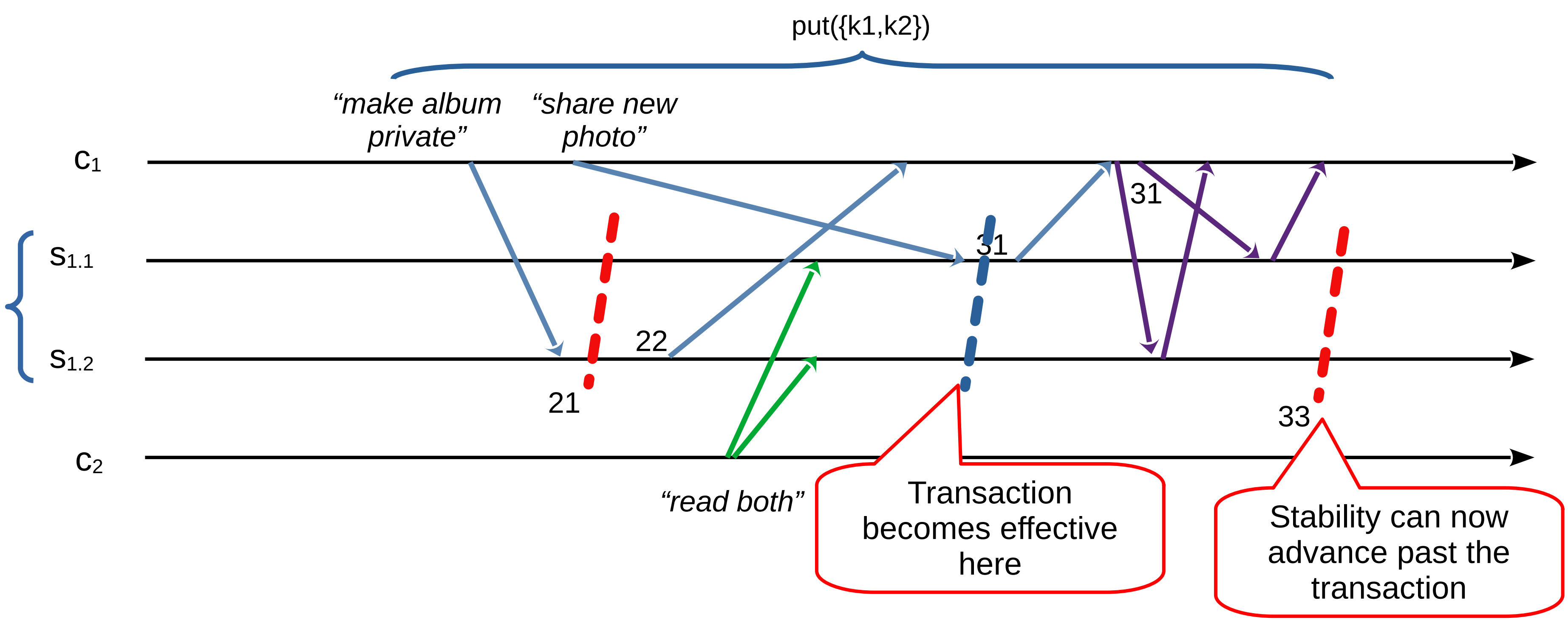
2PC with snapshots

- Each participant proposes a timestamp
- Prepared transactions block stability



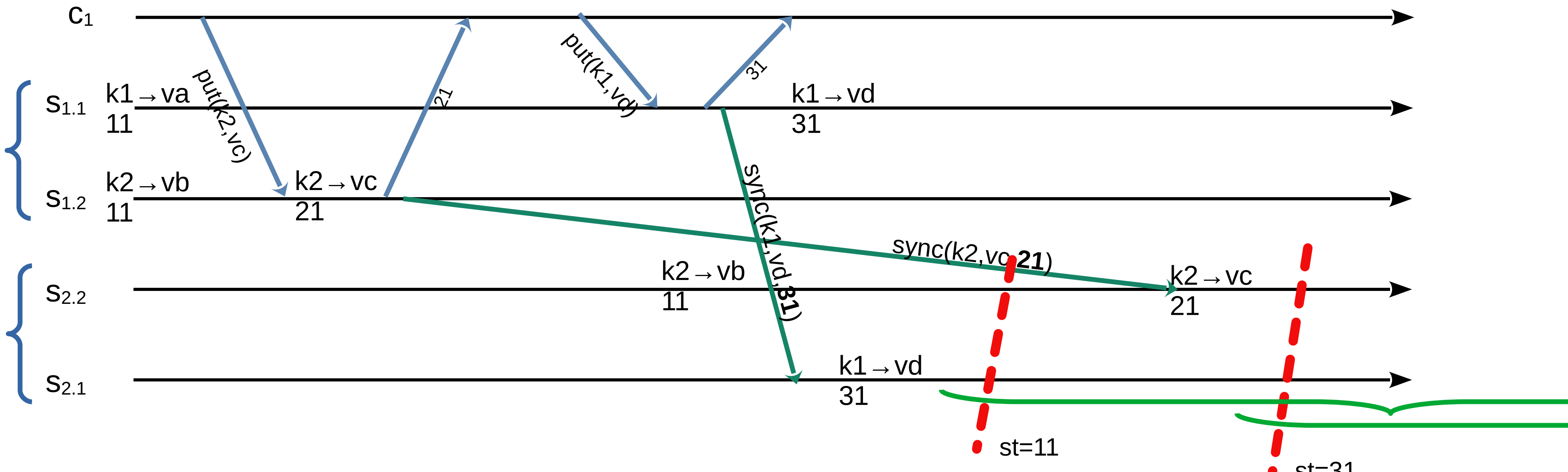
2PC with snapshots

- The latest timestamp is used as the global commit time
  - Known to be after current stable time

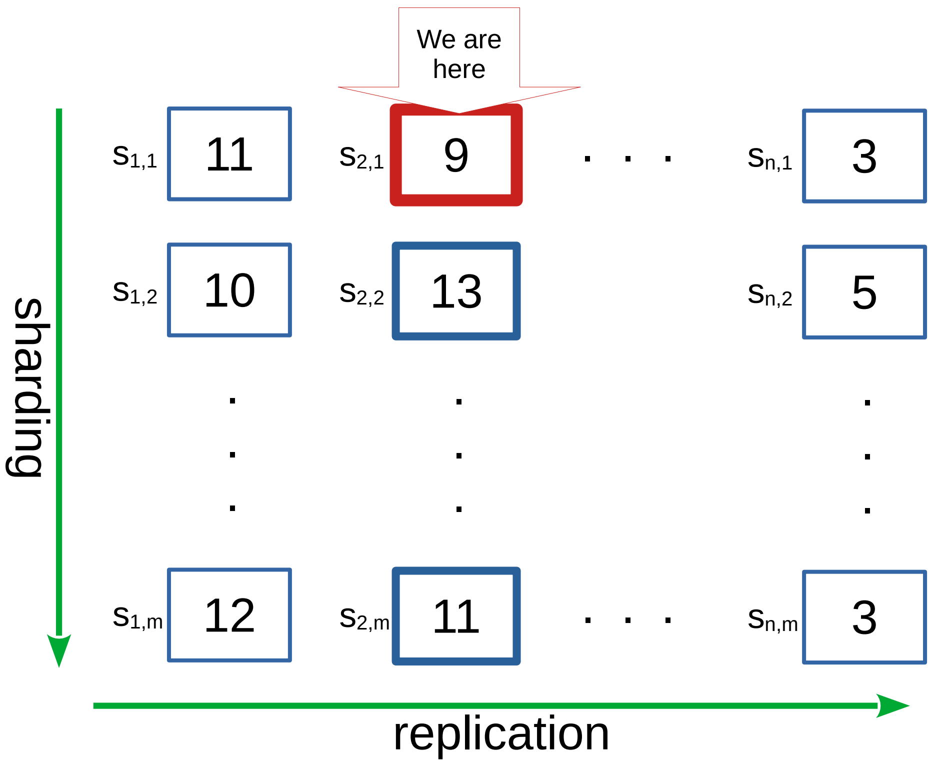


Write propagation with snapshots

- Bonus of snapshots: no need to wait before applying remote updates
  - They remain invisible until stable

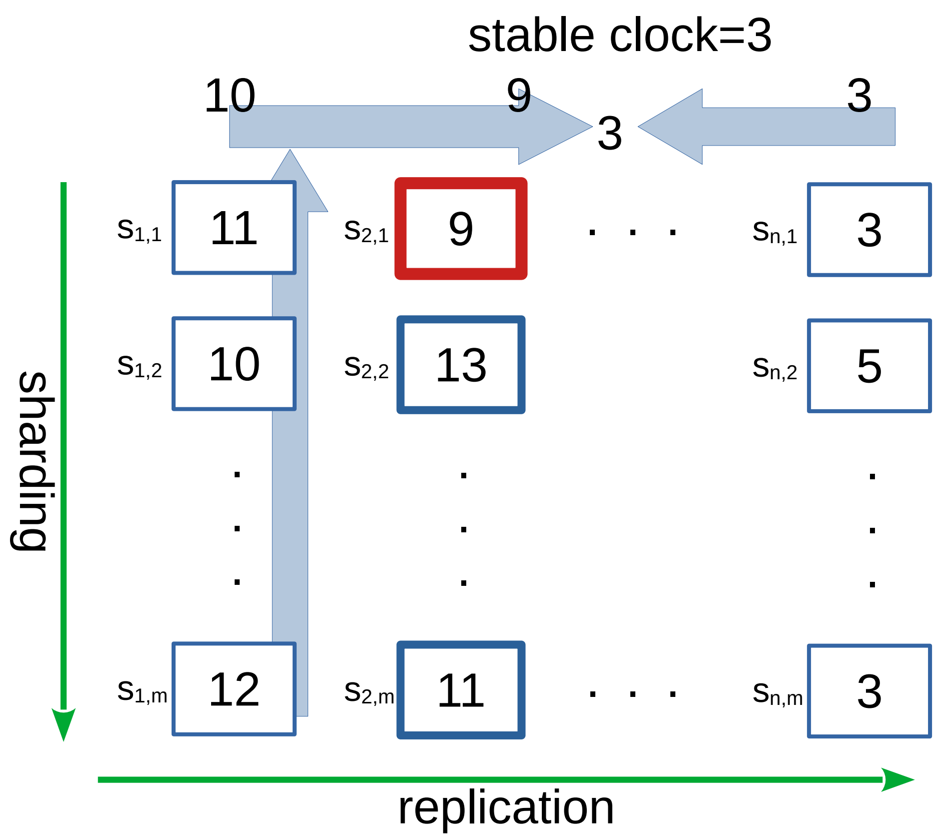


Stability



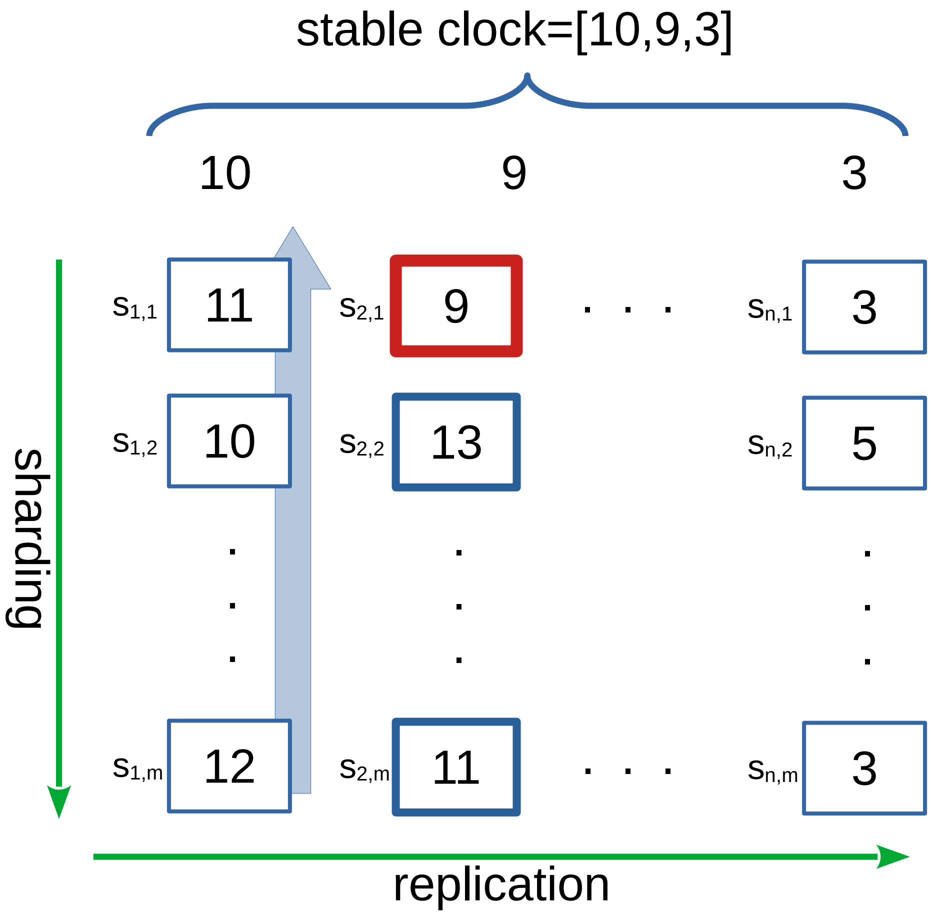
- Each server has its own clock
  - *n* replicas
  - *m* shards
- Knowledge about other servers (esp. other DCs) may be late….

Scalar timestamp



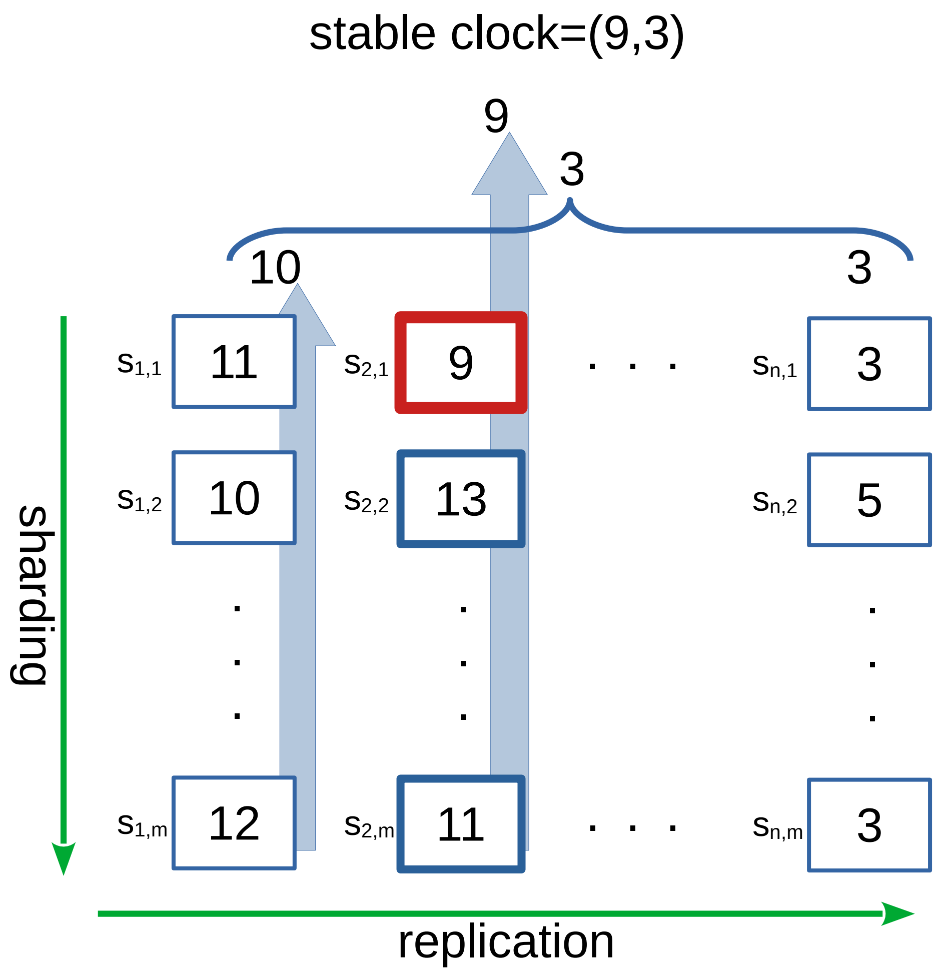
- Stable snapshot is the global minimum
- Visibility determined by oldest replica/shard
  - Blocks if a data center is disconnected
  - Not "AP"

Vector timestamp



- Stable snapshot is the minimum for each DC
- Commit timestamp is vectorial
- Values become visible when *st* is larger than the transactions *ct*
  - Remember that *st* ≤ stable clock
- Does not block when DCs are partitioned

Two scalars



- Stable snapshot:
  - Local component (used for local transactions)
  - Remote component (used for others)
- Does not block local updates when DCs are partitioned
- Blocks all remote when a single DC is partitioned

Summary

- Interactive causal transactions
  - Consistent reads
  - Atomic writes
- Non-blocking reads and writes
- Trade-off: Freshness of values read waiting for stability detection

References

- D. D. Akkoorath et al., “**Cure: Strong Semantics Meets High Availability and Low Latency**,” in 2016 IEEE 36th International Conference on Distributed Computing Systems (ICDCS), Jun. 2016 http://dx.doi.org/10.1109/ICDCS.2016.98