Motivation

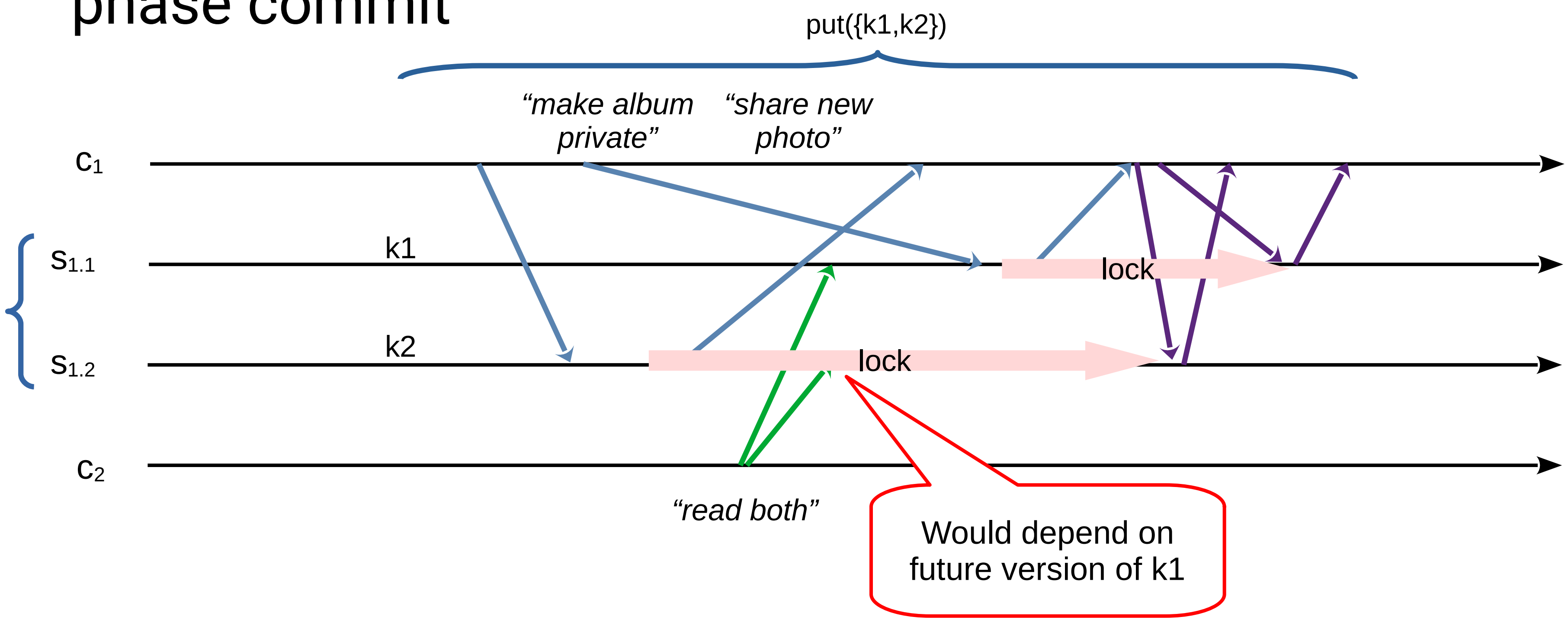
- Example: Read a photo album and its access control list
- Option 1, concurrently:
  - Read photos, then read ACL
  - Delete private photo, make album public
    - Might read private photo and see it as public
- Option 2, concurrently:
  - Read ACL, then read photos
  - Make album private, add private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

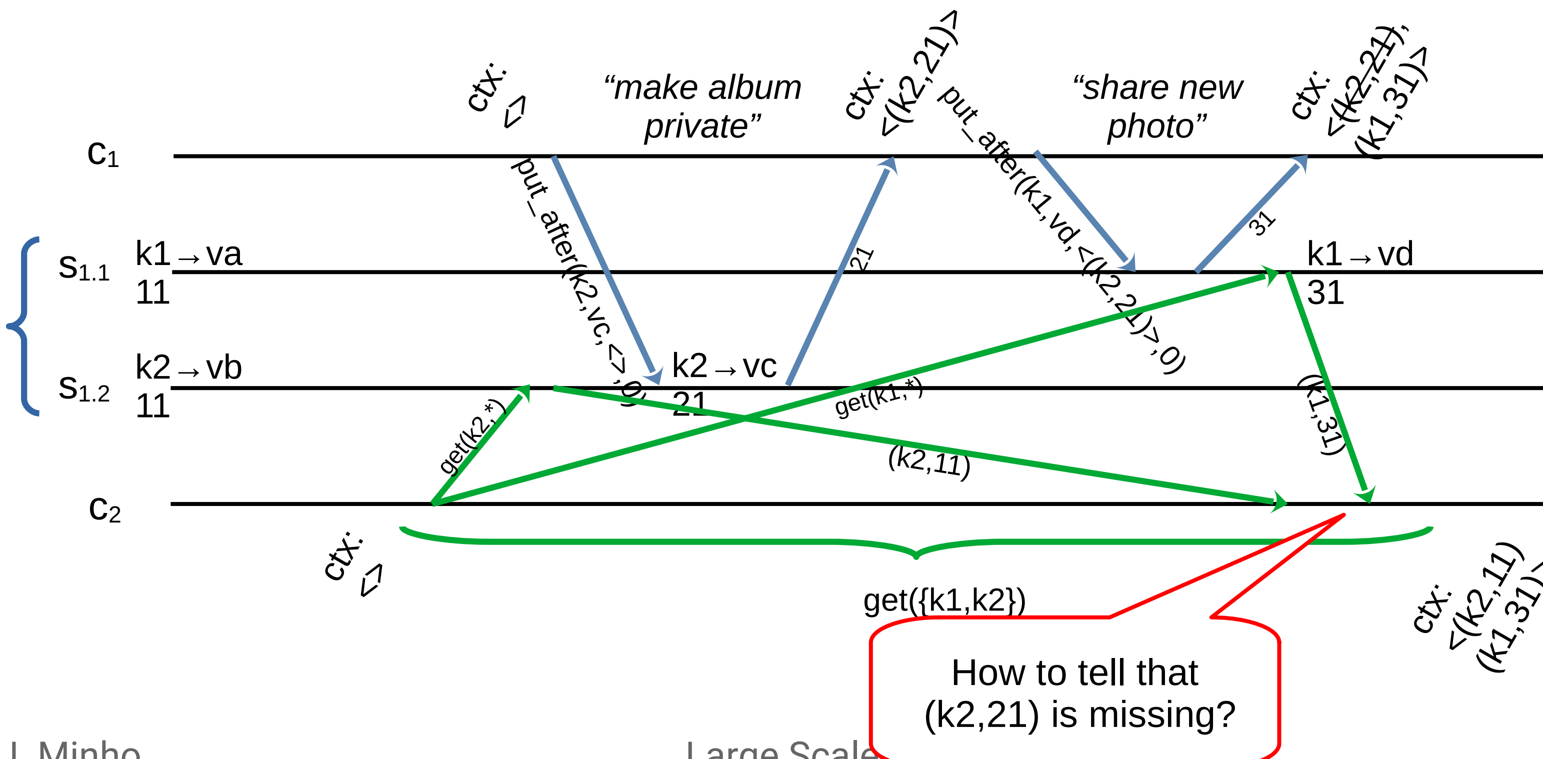
Write transactions: Challenges

- Reads assume that all dependencies are committed
- An atomic update of two servers would need locking and 2-phase commit



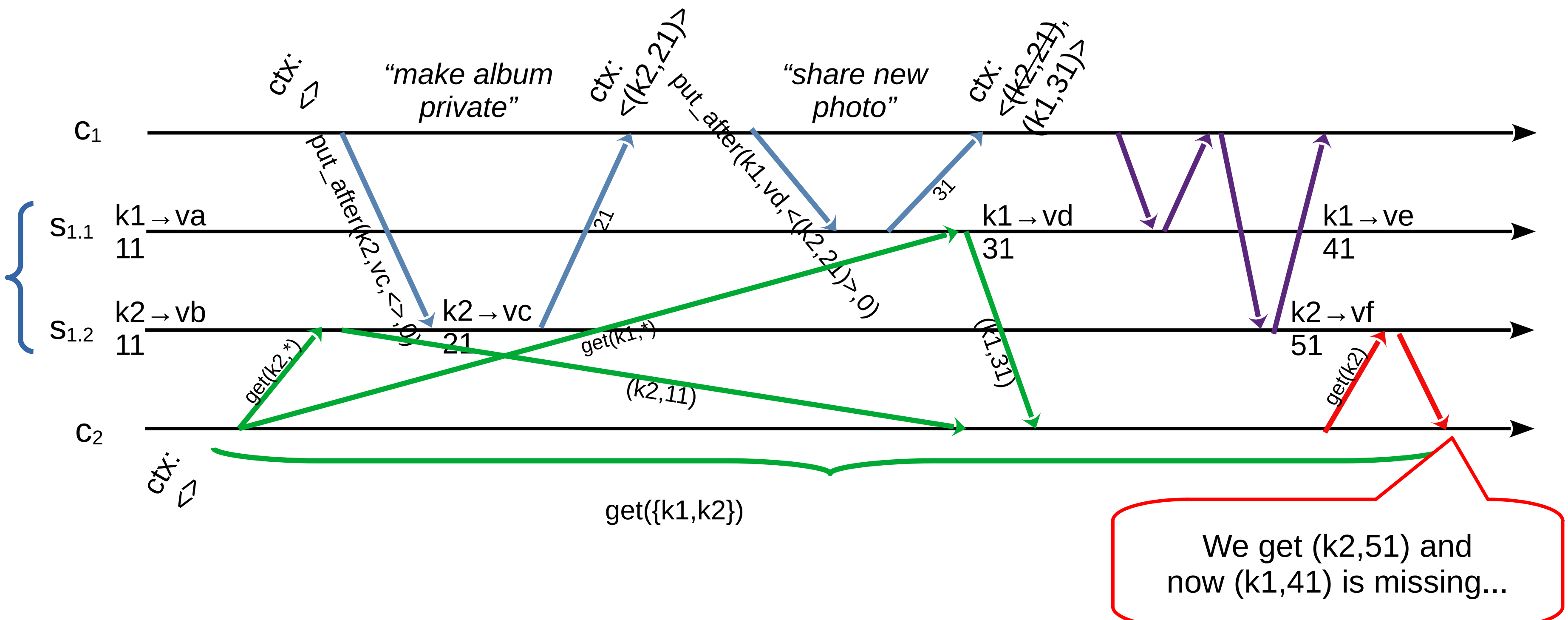
Read transactions: Challenges

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



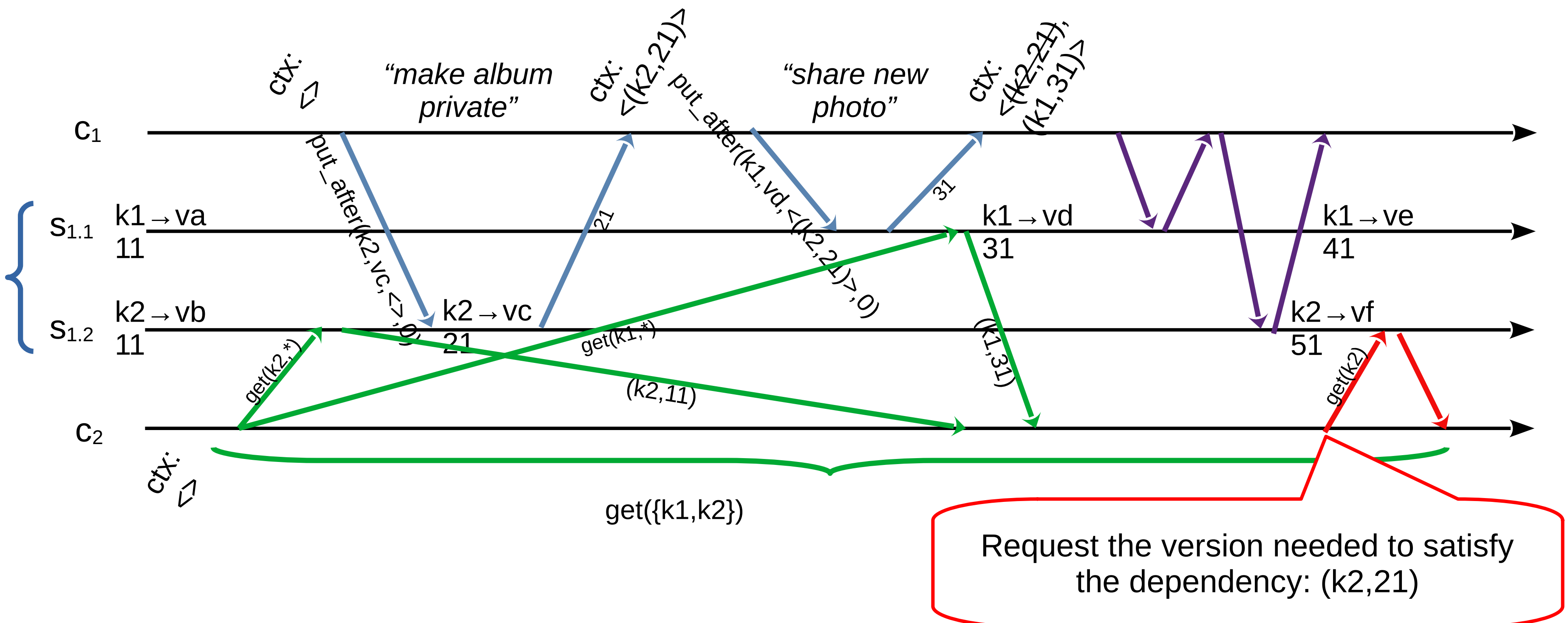
Read transactions: Challenges

- Assuming that we know what is missing... re-read it
- How to get it without introducing new dependencies?



Multi-version

- Keep preceding versions of each item
- Return the exact version needed by the transaction



Read from snapshot

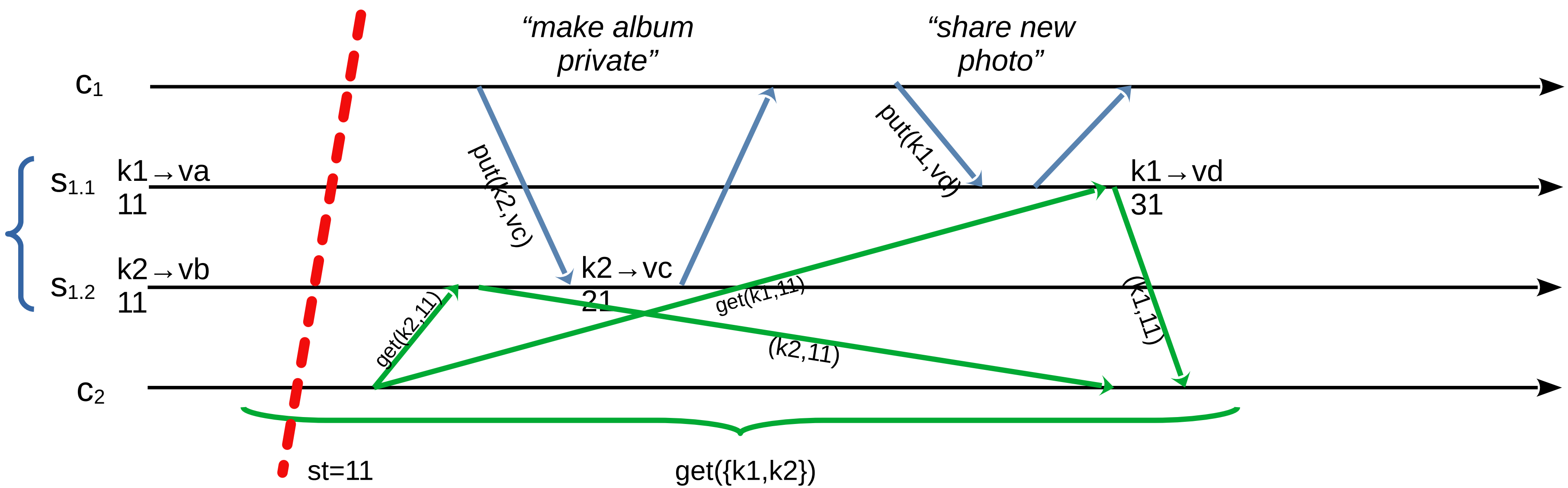
- How to avoid keeping detailed dependency information?
- The problem arises when reading some item that was written after the transaction has started:
  - (k1,31) in the example

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

- <u>Logical time</u>: Read only versions that existed when the transaction started

Read from snapshot

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



Snapshot assignment

- Stability of some snapshot *t*: all updates (hence, their dependencies) are available in all hosts
  - Local stability: updates originating from the same data center
  - Global stability: upates originating from remote data centers
- The latest stable snapshot can efficiently be computed with epidemic protocols

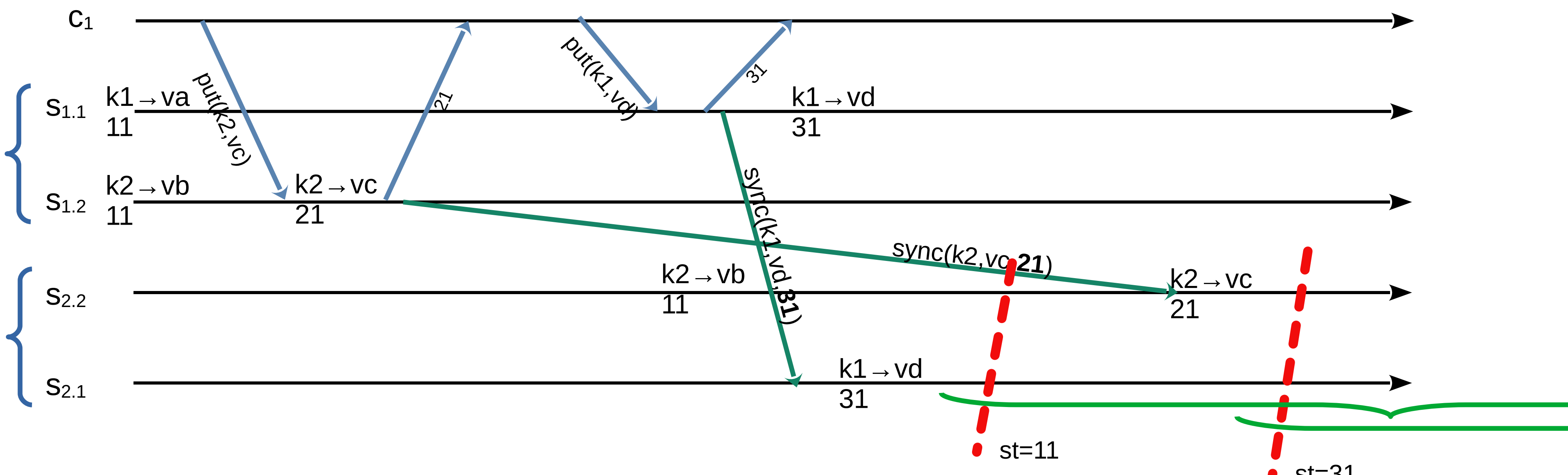
- Start by assuming global stability...

Write and commit

- Invariant: transaction *st* ≤ process clock, for all transactions and processes
- Commit items with a *ct* incrementing the clock at some server (the commit coordinator)
  - This means that *ct* > *st*, for all currently active transactions
  - Therefore, writes are invisible to concurrent transactions

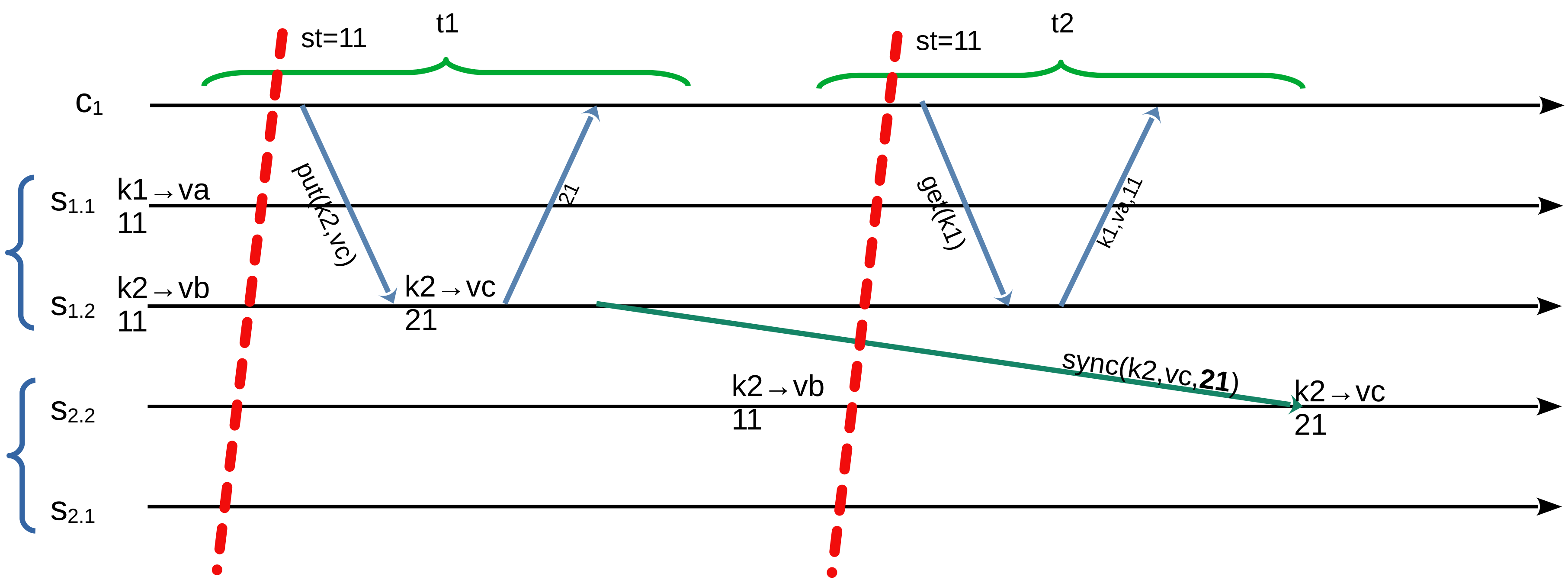
Write propagation with snapshots

- Bonus of snapshots: no need to wait before applying remote updates



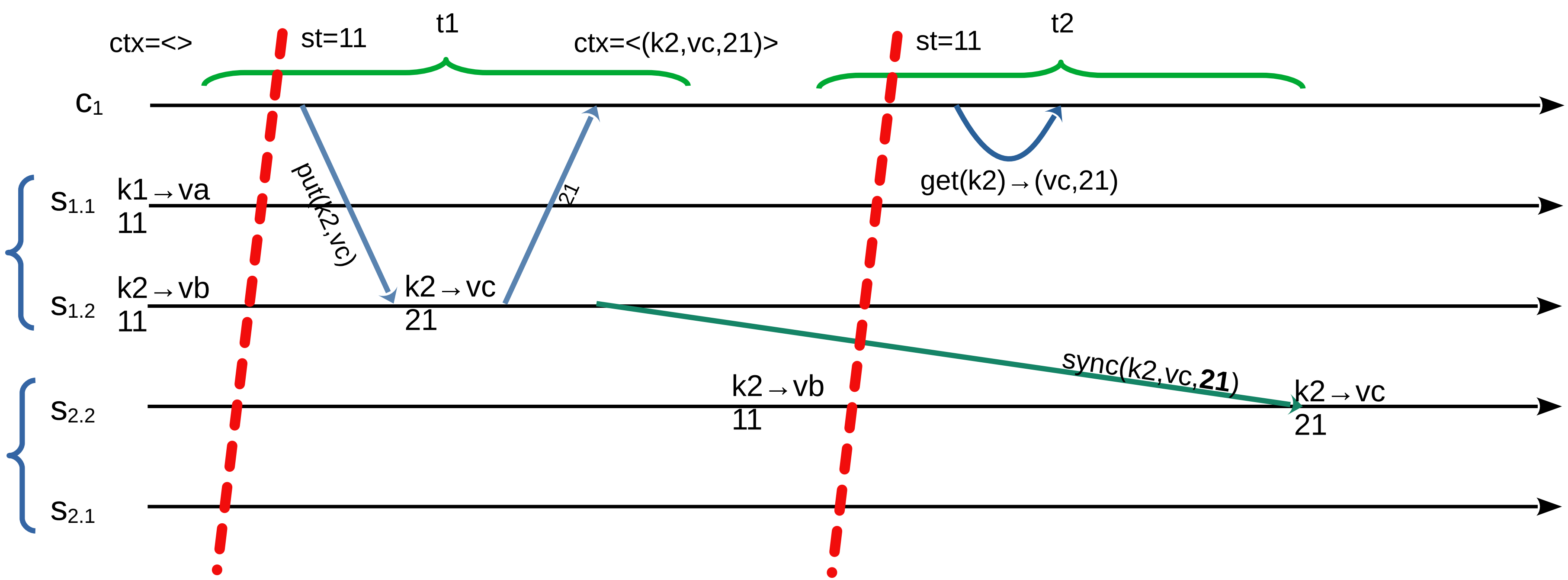
Read your own writes

- A second transaction by the same client might not read values that have just been written
  - Either commit *t1* or start *t2* would have to block



Read your own writes

- Waiting for values that have just been written!?
- Keep recent writes in a client-side context



Client context

- When writing: Keep values with commit timestamp
- When reading: If item is available in local context, use its value
- Remove items from local context when:
  - Writing a newer version of the same item
  - Starting a transaction with *st* higher that item version (this means that value is alread available at servers)

Local vs global stability

- The limiting factor for stability (advancing *st*) is receiving updates from remote data centers
- Note that <u>local updates that are locally stable cannot depend on missing remote updates</u>…
- Separately track stability and make *st* = (*lst*, *gst*):
  - First read from client context
  - Then, read local updates with $v \leq lst$
  - Finally, read any updates with $v \leq gst$

Summary

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

References

- K. Spirovska, D. Didona, and W. Zwaenepoel, “**Wren: Nonblocking Reads in a Partitioned Transactional Causally Consistent Data Store**,” in 2018 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN), Jun. 2018, http://dx.doi.org/10.1109/DSN.2018.00014