Distributed Transactions in Spanner 1

جامعة الملك عبدالله
للعلوم والتقنية
King Abdullah University of
Science and Technology

CS 240: Computing Systems and Concurrency
Lecture 20

Marco Canini

Credits: Michael Freedman and Kyle Jamieson developed much of the original material.
Contents adapted from Haonan Lu, Wyatt Lloyd.

Recap: Distributed Storage Systems

- Concurrency control
  - Order transactions across shards

- State machine replication
  - Replicas of a shard apply transactions in the same order decided by concurrency control

Google’s Setting

- Dozens of zones (datacenters)
- Per zone, 100-1000s of servers
- Per server, 100-1000 partitions (tablets)
- Every tablet replicated for fault-tolerance (e.g., 5x)

Why Google built Spanner

2005 – BigTable [OSDI 2006]

- Eventually consistent across datacenters
- Lesson: “don’t need distributed transactions”

2008? – MegaStore [CIDR 2011]

- Strongly consistent across datacenters
- Option for distributed transactions
  - Performance was not great…

2011 – Spanner [OSDI 2012]

- Strictly Serializable Distributed Transactions
- “We wanted to make it easy for developers to build their applications”

A Deeper Look at Motivation
-- Performance-consistency tradeoff

- Strict serializability
  - Serializability + linearizability
  - As if coding on a single-threaded, transactionally isolated machine
  - Spanner calls it external consistency
- Strict serializability makes building correct application easier
- Strict serializability is expensive
  - Performance penalty in concurrency control + Replication
    - OCC/2PL: multiple round trips, locking, etc.

A Deeper Look at Motivation

-- Read-Only Transactions

- Transactions that only read data
  - Predeclared, i.e., developer uses READ_ONLY flag / interface
- Reads dominate real-world workloads
  - FB’s TAO had 500 reads : 1 write [ATC 2013]
  - Google Ads (F1) on Spanner from 1? DC in 24h:
    - 31.2 M single-shard read-write transactions
    - 32.1 M multi-shard read-write transactions
    - 21.5 B read-only (~340 times more)
- Determines system overall performance

Can we design a strictly serializable, geo-replicated, sharded system with very fast (efficient) read-only transactions?

Before we get to Spanner …

- How would you design SS read-only transactions?
- OCC or 2PL
  - Multiple round trips and locking
- Can always read in local datacenters like COPS?
  - Maybe involved in Paxos agreement
  - Or must contact the leader
- Performance penalties
  - Round trips increase latency, especially in wide area
  - Distributed lock management is costly, e.g., deadlocks

Goal is to …

- Make read-only transactions efficient
  - One round trip
    - Could be wide-area
  - Lock-free
    - No deadlocks
    - Processing reads do not block writes, e.g., long-lived reads
  - Always succeed
    - Do not abort

- And strictly serializable

Leveraging the Notion of Time

- Strict serializability: a matter of real-time ordering
  - If txn T2 starts after T1 finishes, then T2 must be ordered after T1
    - If T2 is a ro-txn, then T2 should see the effects of all writes that finished before T2 started

Leveraging the Notion of Time

- Task 1: when committing a write, tag it with the current physical time
- Task 2: when reading the system, check which writes were committed before the time this read started
- How about the serializable requirement?
  - Physical time naturally gives a total order

Invariant:

If T2 starts after T1 commits (finishes), then T2 must have a larger timestamp

Trivially provided by perfect clocks

Challenges

- Clocks are not perfect
  - Clock skew: some clocks are faster/slower
  - Clock skew may not be bounded
  - Clock skew may not be known a priori
- T2 may be tagged with a smaller timestamp than T1 due to T2's slower clock
- Seems impossible to have perfect clocks in distributed systems. What can we do?

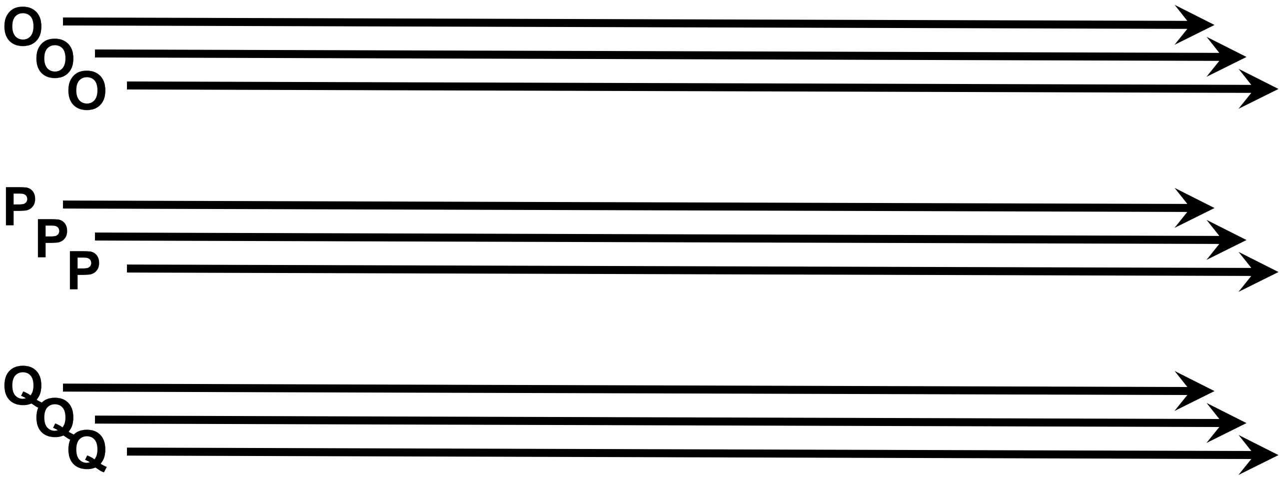
Nearly perfect clocks

- Partially synchronized
  - Clock skew is bounded and known a priori
  - My clock shows 1:30PM, then I know the absolute (real) time is in the range of 1:30 PM +/- X
    - e.g., between 1:20PM and 1:40PM if X = 10 mins
- Clock skew is short
  - E.g., X = a few milliseconds
- Enable something special, e.g., Spanner!

Spanner: Google’s Globally-Distributed Database

OSDI 2012

Scale-out vs. fault tolerance



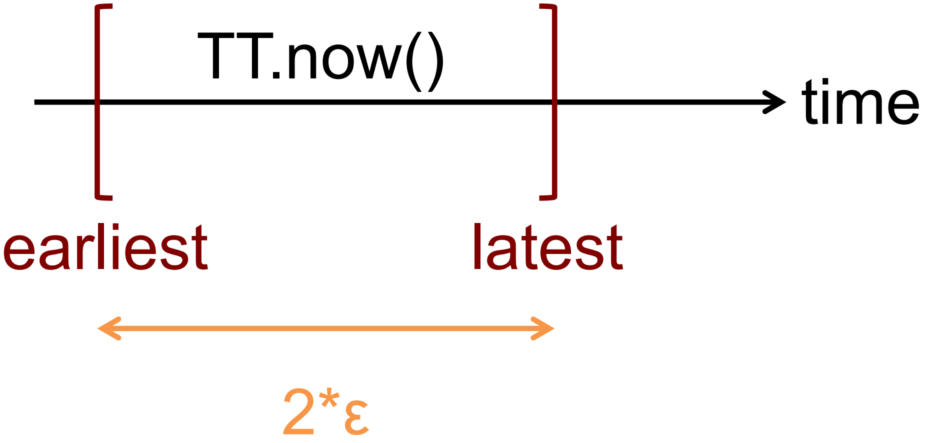
- Every tablet replicated via MultiPaxos
- So every “operation” within transactions across tablets actually is a replicated operation within Paxos RSM
- Paxos groups can stretch across datacenters!

Strictly Serializable Multi-Shard Transactions

- How are clocks made “nearly perfect”?
- How does Spanner leverage these clocks?
  - How are writes done and tagged?
  - How read-only transactions are made efficient?

TrueTime (TT)

- "Global wall-clock time" with bounded uncertainty
  - ε is worst-case clock divergence
  - Spanner's time notion becomes intervals, not single values
  - ε is 4ms on average, 2 ε is about 10ms



Consider event e_{now} which invoked tt = TT.now():

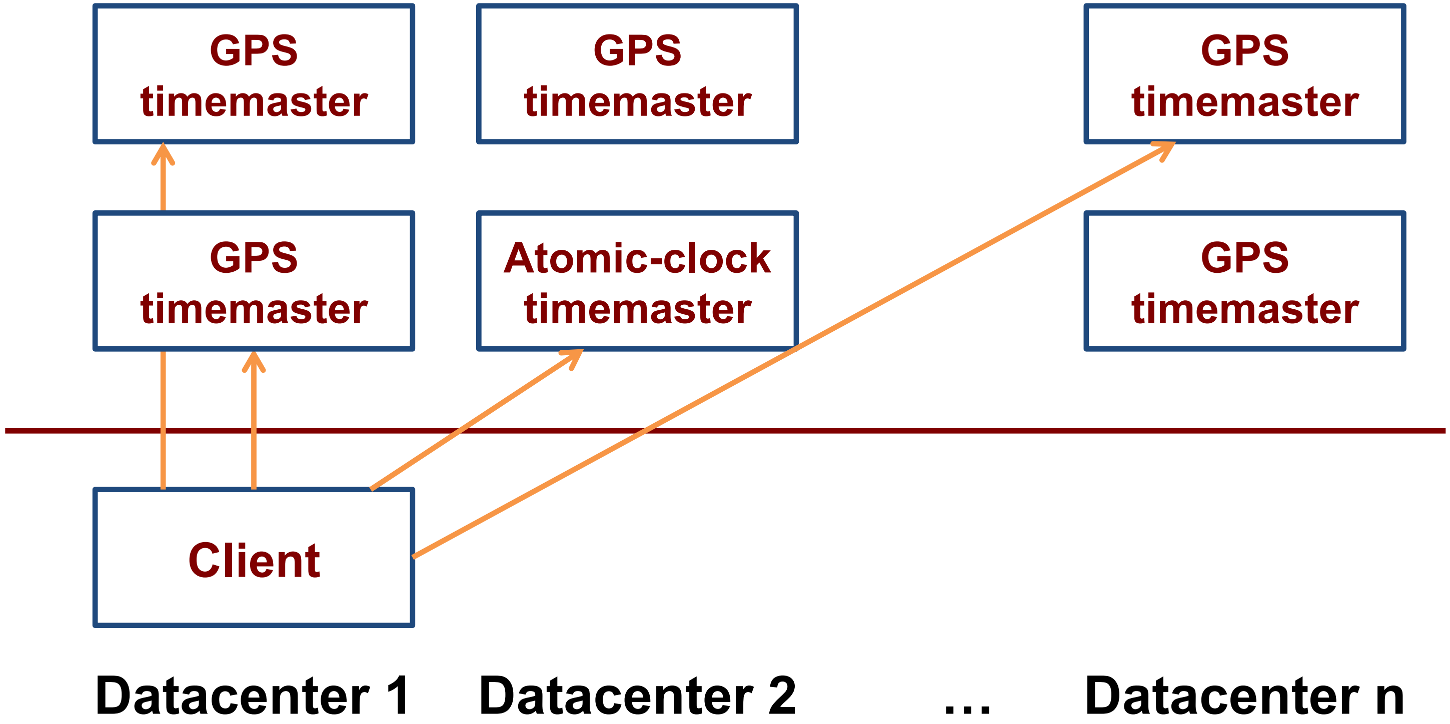
Guarantee: $tt.earliest <= t_{abs}(e_{now}) <= tt.latest$

TrueTime (TT)

- Interface
  - TT.now() = [earliest, latest] # latest – earliest = 2*ε
  - TT.after(t) = true if t has passed
    - TT.now().earliest > t (b/c t_{abs} >= TT.now().earliest)
  - TT.before(t) = true if t has not arrived
    - TT.now().latest < t (b/c t_{abs} <= TT.now().latest)

- Implementation
  - Relies on specialized hardware, e.g., GPS satellite and atomic clocks

TrueTime Architecture



GPS timemaster

GPS timemaster

GPS timemaster

GPS timemaster

Atomic-clock timemaster

GPS timemaster

Client

Datacenter 1

Datacenter 2

…

Datacenter n

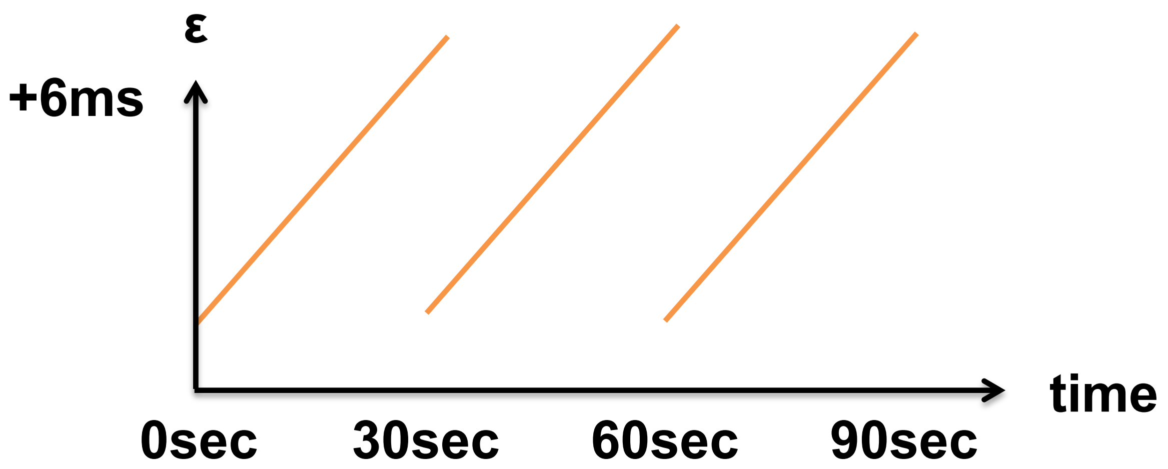
Compute reference [earliest, latest] = now ± ε

TrueTime implementation

now = reference now + local-clock offset

ε = reference ε + worst-case local-clock drift

= 1ms + 200 μs/sec

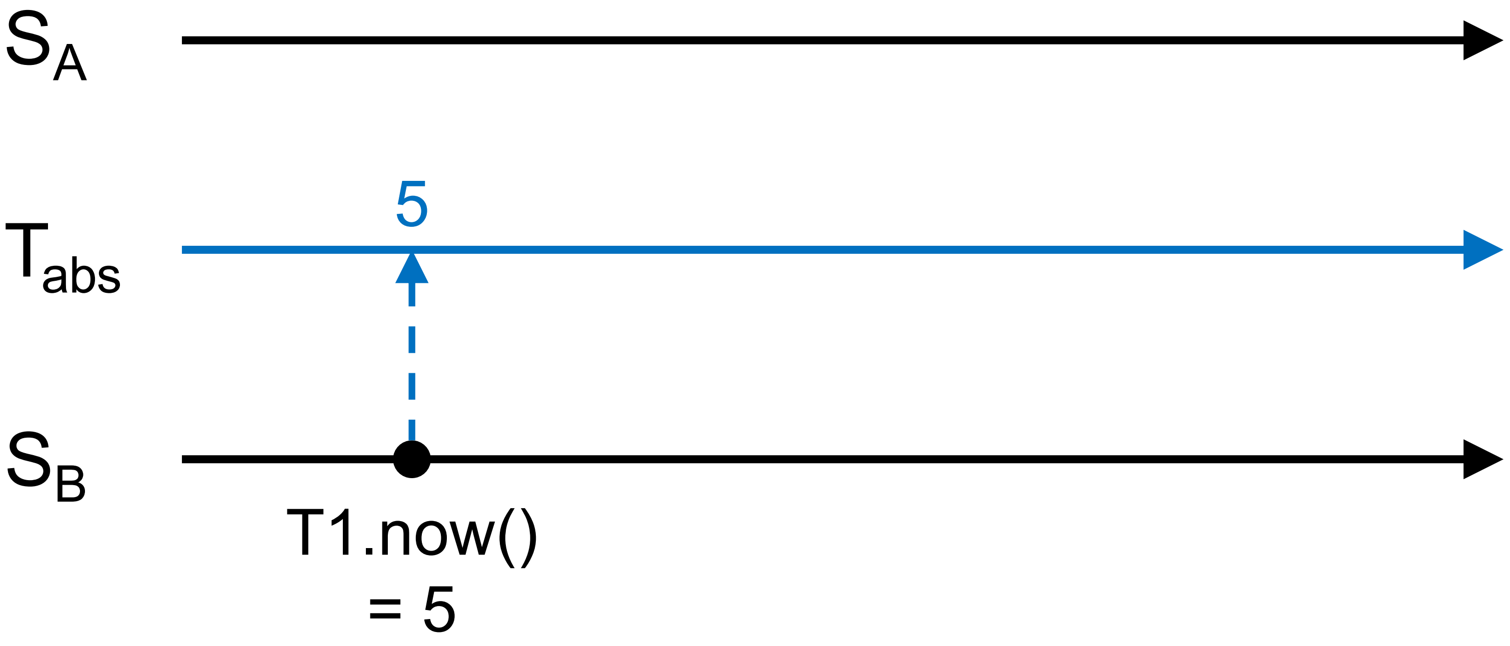


- What about faulty clocks?
  - Bad CPUs 6x more likely in 1 year of empirical data

Enforcing the Invariant

If T2 starts after T1 commits (finishes), then T2 must have a larger timestamp

Let T1 write S_B and T2 write S_A

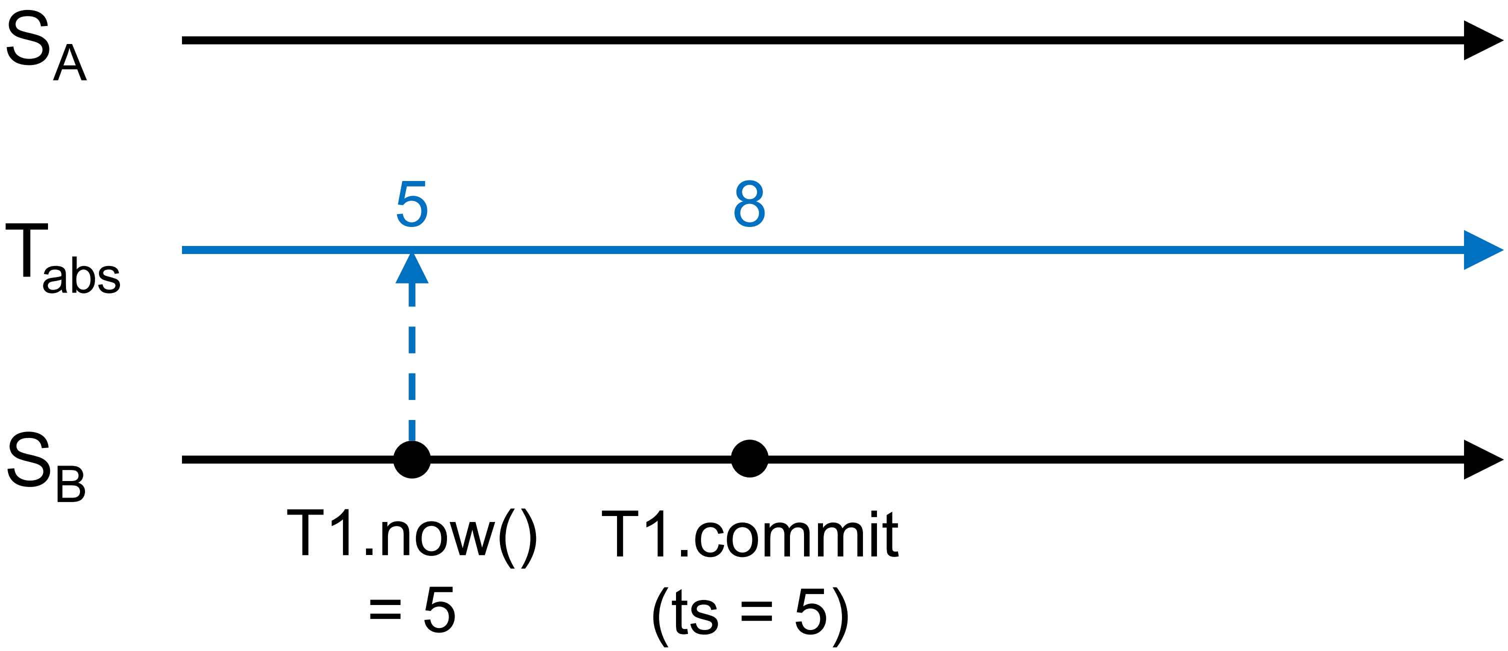


Perfect Clocks

Enforcing the Invariant

If T2 starts after T1 commits (finishes), then T2 must have a larger timestamp

Let T1 write S_B and T2 write S_A



S_A

T_{abs}

5

8

S_B

T1.now() = 5

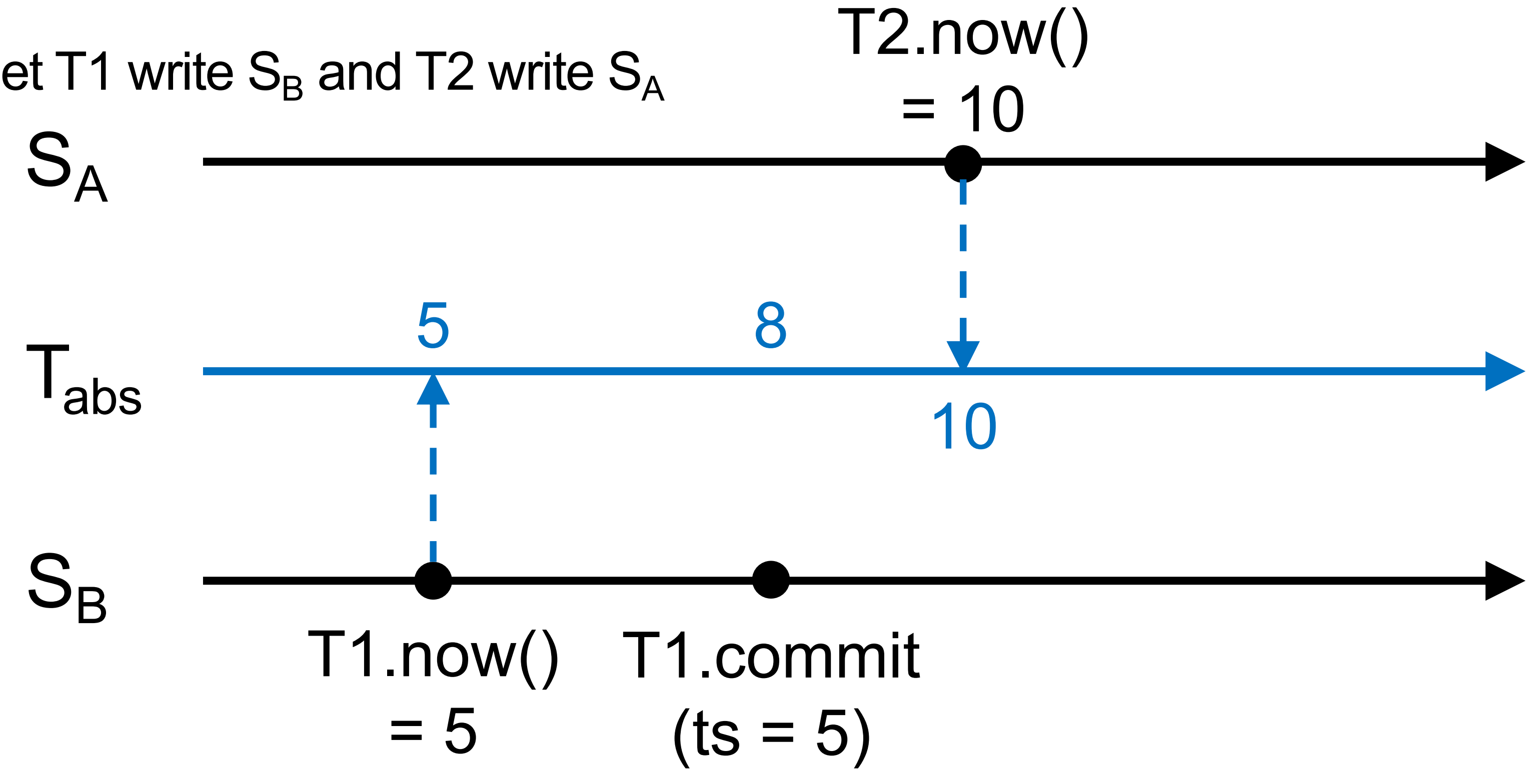
T1.commit (ts = 5)

Perfect Clocks

Enforcing the Invariant

If T2 starts after T1 commits (finishes), then T2 must have a larger timestamp

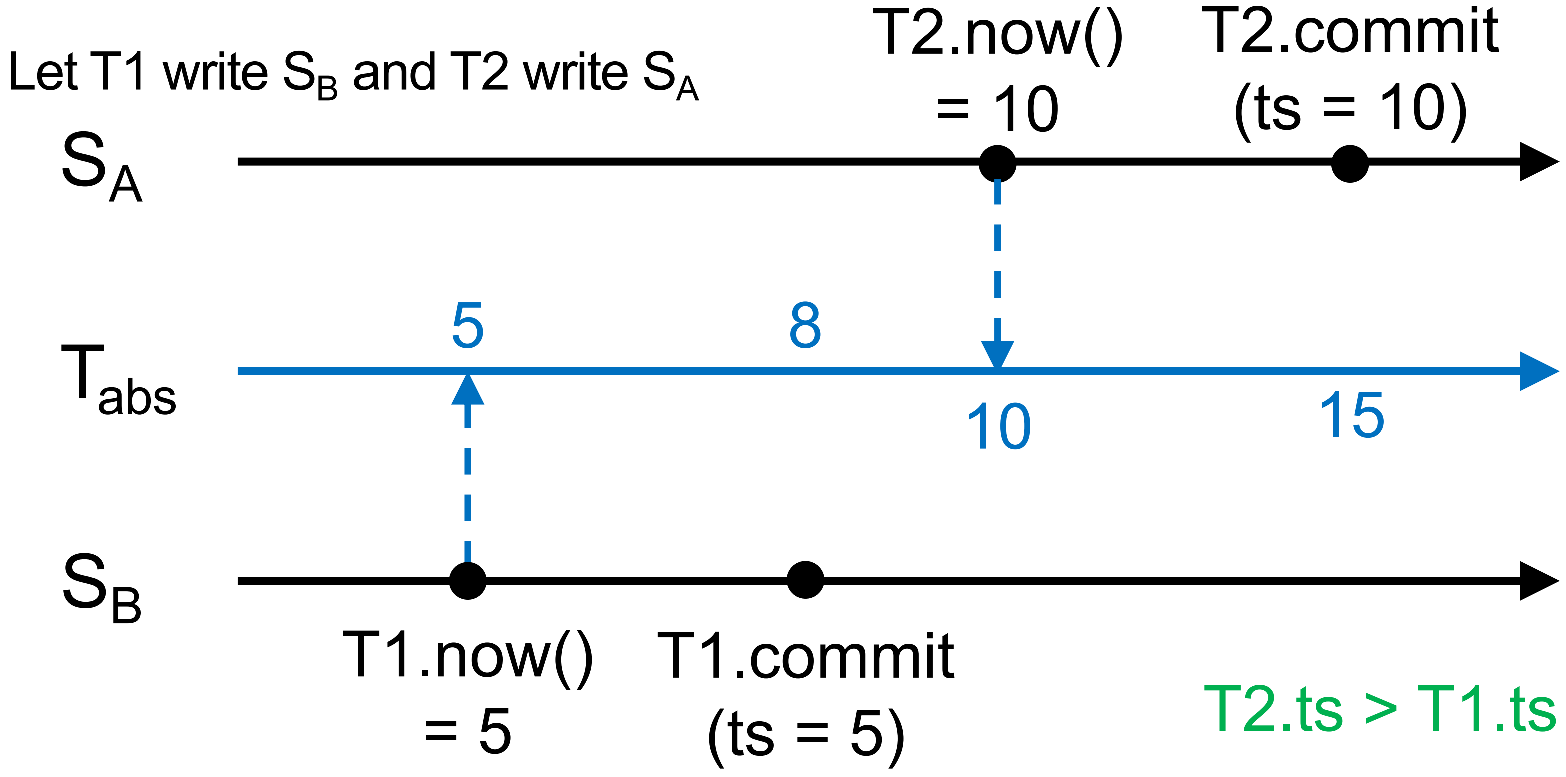
Let T1 write S_B and T2 write S_A



Perfect Clocks

Enforcing the Invariant

If T2 starts after T1 commits (finishes), then T2 must have a larger timestamp



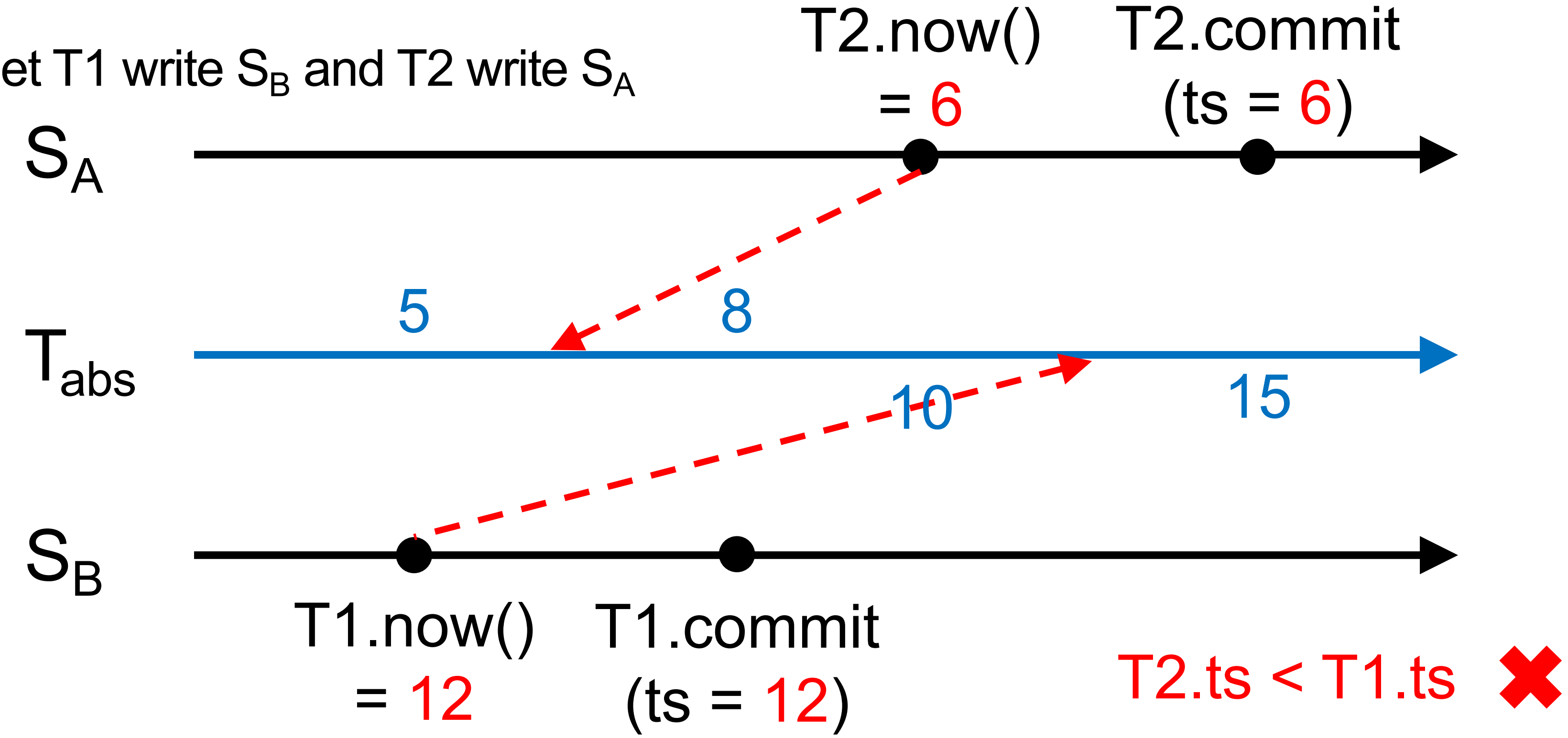
T2.ts > T1.ts

Perfect Clocks

Enforcing the Invariant

If T2 starts after T1 commits (finishes), then T2 must have a larger timestamp

Let T1 write S_B and T2 write S_A



T2.now() = 6

T2.commit (ts = 6)

S_A

T_{abs}

5

8

10

15

S_B

T1.now() = 12

T1.commit (ts = 12)

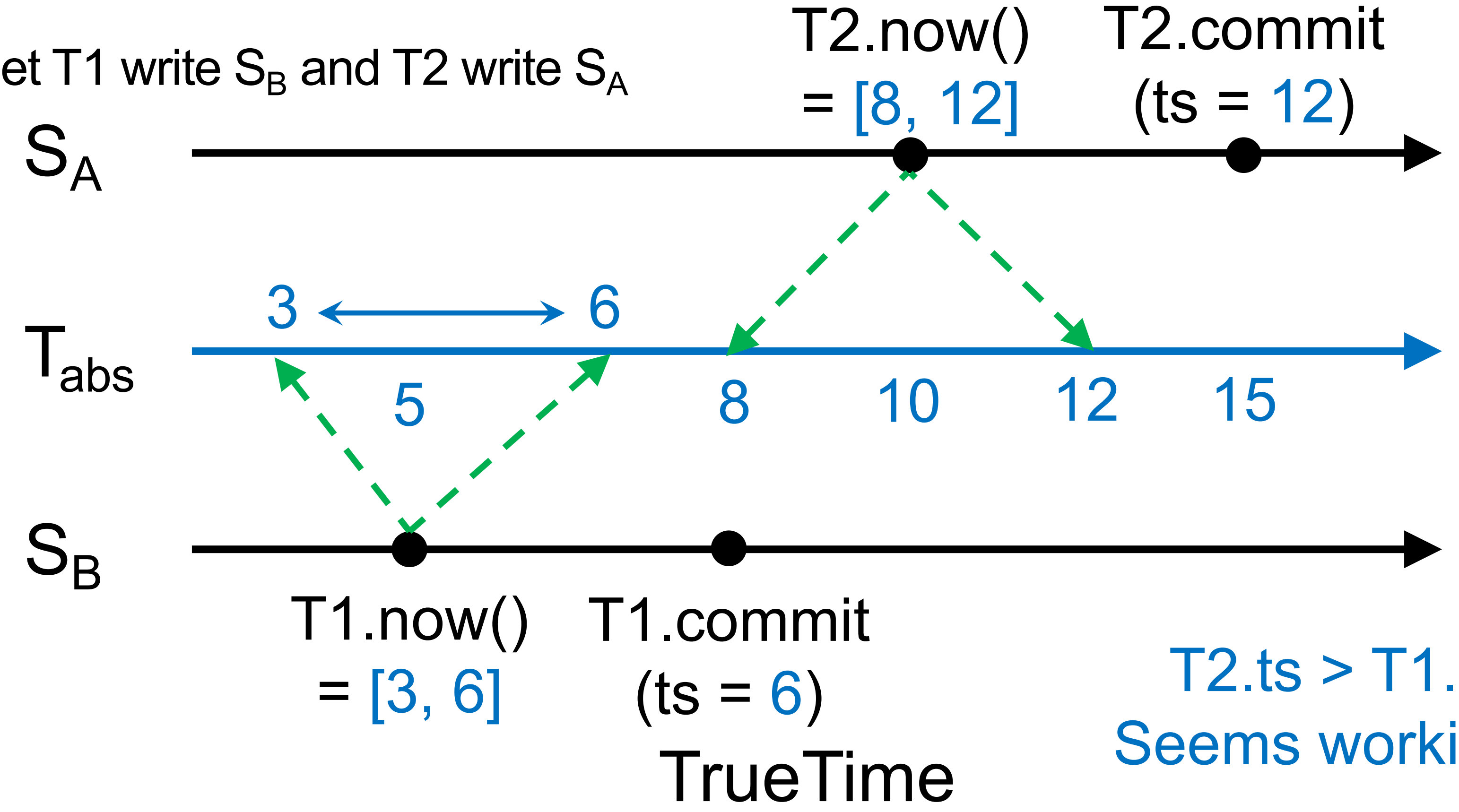
T2.ts < T1.ts ✖

Imperfect Clocks

Enforcing the Invariant

If T2 starts after T1 commits (finishes), then T2 must have a larger timestamp

Let T1 write S_B and T2 write S_A



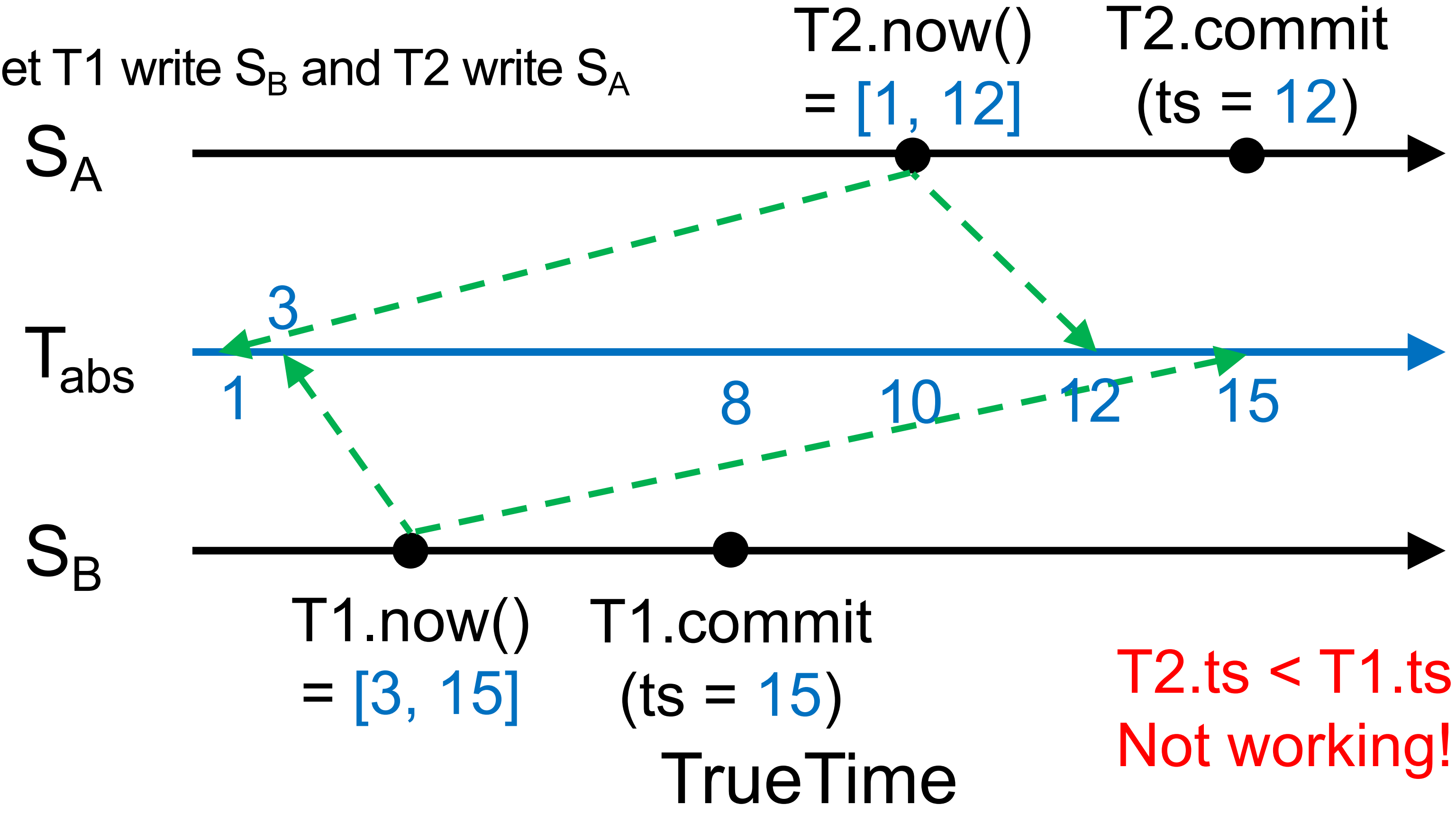
T2.ts > T1.ts

Seems working?

Enforcing the Invariant

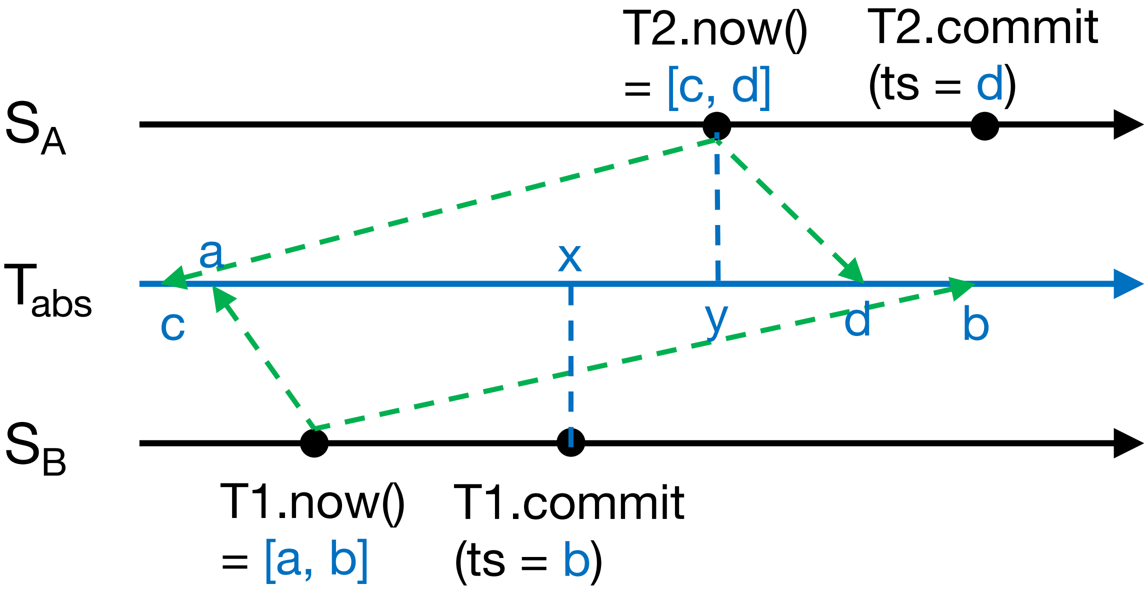
If T2 starts after T1 commits (finishes), then T2 must have a larger timestamp

Let T1 write S_B and T2 write S_A



T2.ts < T1.ts
Not working!

A brain teaser puzzle

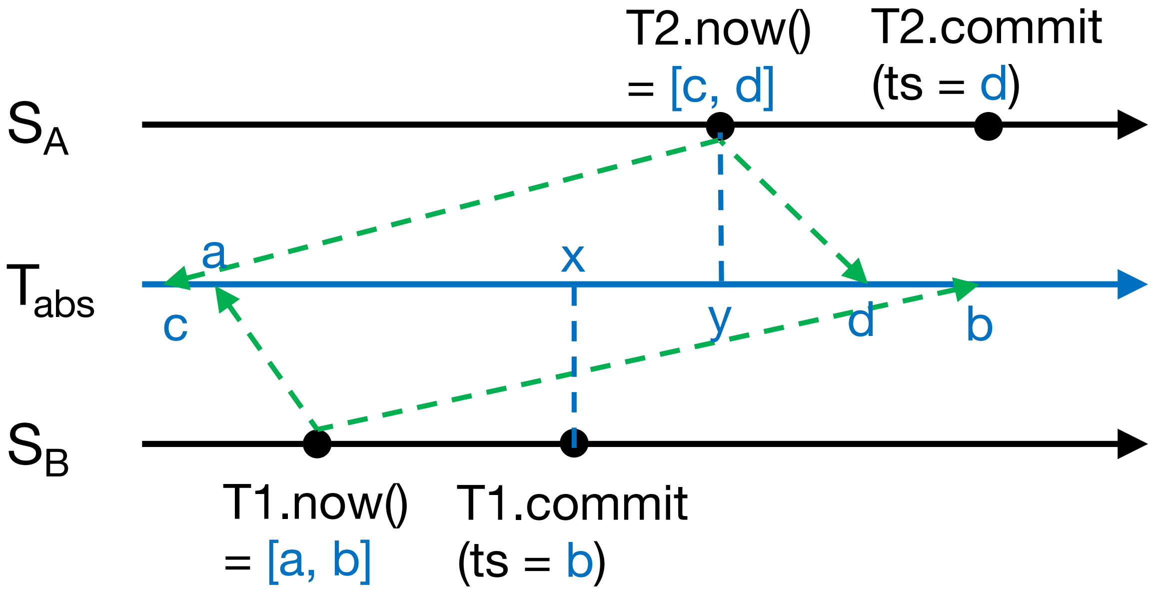


We know:

1. x < y, b/c T2 in real-time after T1 (the assumption)
2. c <= y <= d, b/c TrueTime
3. T1.ts = b, T2.ts = d, b/c how ts is assigned

We want: it is always true that b < d, how?

A brain teaser puzzle



We know:

1. x < y, b/c T2 in real-time after T1 (the assumption)
2. c <= y <= d, b/c TrueTime
3. T1.ts = b, T2.ts = d, b/c how ts is assigned

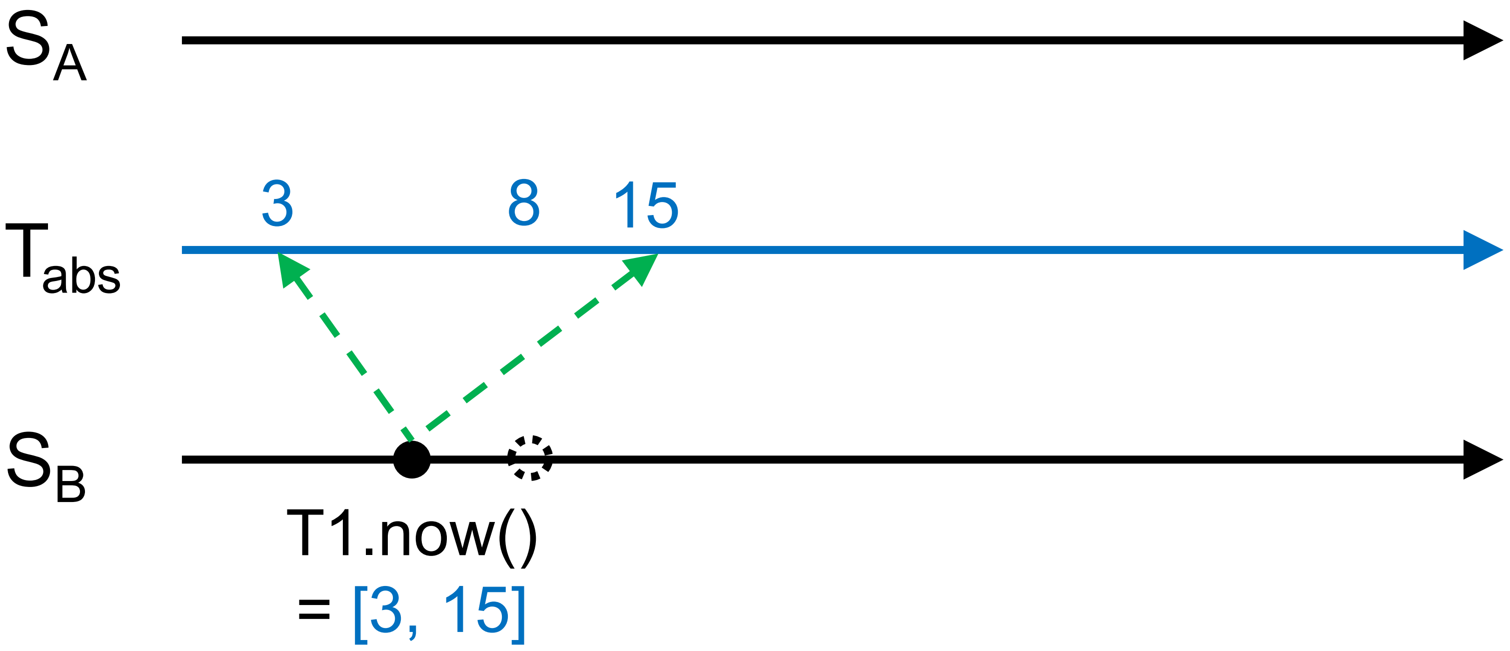
We want: it is always true that b < d, how?

1 and 2 → x < d; we need to ensure b < x; then b < x < d, done

Enforcing the Invariant with TT

If T2 starts after T1 commits (finishes), then T2 must have a larger timestamp

Let T1 write S_B and T2 write S_A

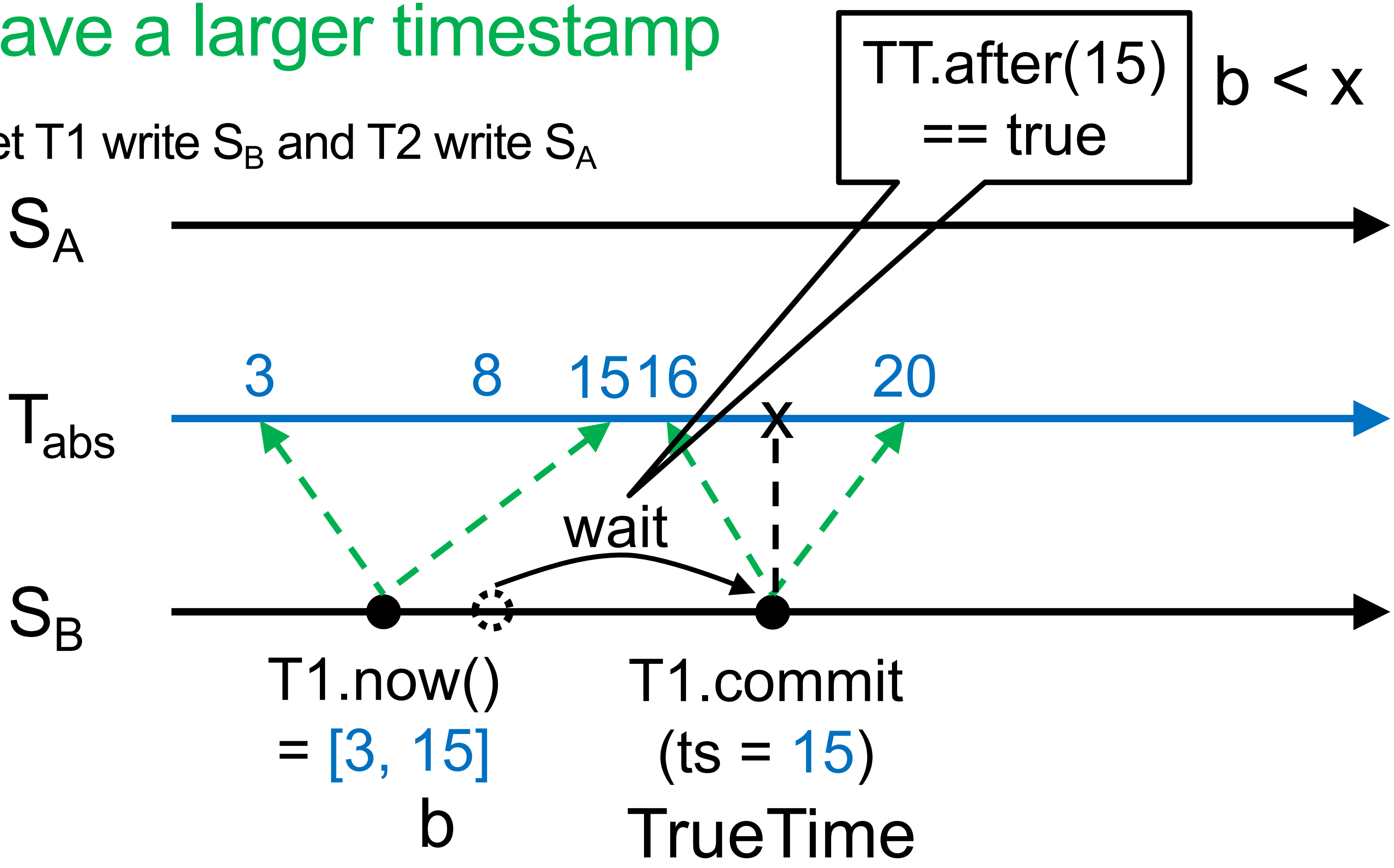


TrueTime

Enforcing the Invariant with TT

If T2 starts after T1 commits (finishes), then T2 must have a larger timestamp

Let T1 write S_B and T2 write S_A



TT.after(15) == true

b < x

S_A

T_{abs}

3 8 15 16 20 x

wait

S_B

T1.now() = [3, 15]

b

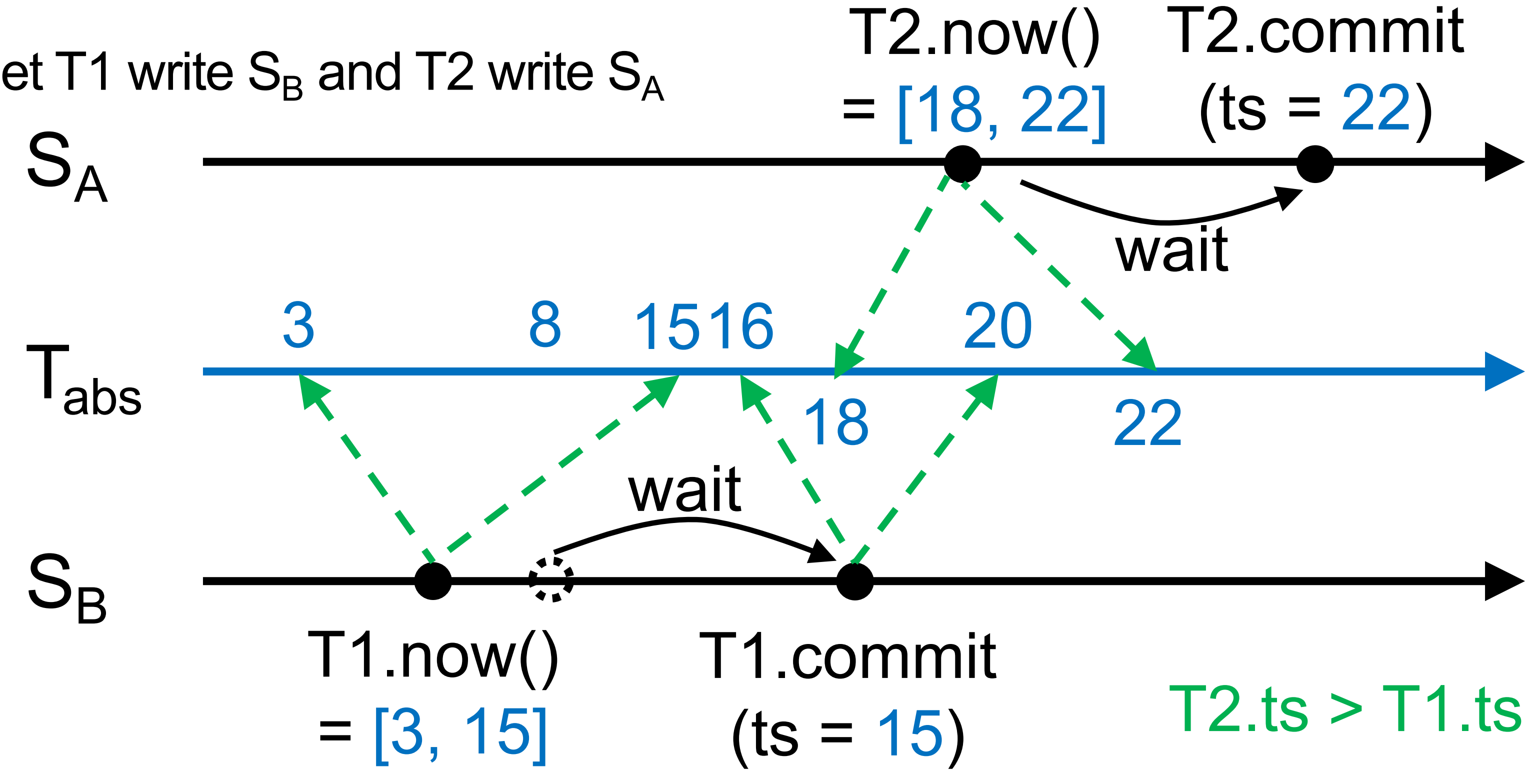
T1.commit (ts = 15)

TrueTime

Enforcing the Invariant with TT

If T2 starts after T1 commits (finishes), then T2 must have a larger timestamp

Let T1 write S_B and T2 write S_A



T2.ts > T1.ts

TrueTime

Takeaways

- The invariant is always enforced: If T2 starts after T1 commits (finishes), then T2 must have a larger timestamp
- How big/small ε is does not matter for correctness
- Only need to make sure:
  - TT.now().latest is used for ts (in this example)
  - Commit wait, i.e., TT.after(ts) == true
- ε must be known a priori and small so commit wait is doable!

After-class Puzzles

- Can we use TT.now().earliest for ts?
- Can we use TT.now().latest – 1 for ts?
- Can we use TT.now().latest + 1 for ts?
- Then what’s the rule of thumb for choosing ts?