UNIVERSITY OF Southampton

Timestamps and Advanced Transactions

COMP3211 Advanced Databases

Dr Nicholas Gibbins – nmg@ecs.soton.ac.uk

2020-2021

Overview

- Timestamps
- Savepoints
- Chained transactions
- Nested transactions
- Sagas

Timestamps

Timestamps

- An alternative to locks – deadlock cannot occur
- Timestamps are unique identifiers for transactions – the transaction start time: TS(T)
- For each resource X, there is:
  - A read timestamp, read-TS(X)
  - A write timestamp, write-TS(X)
- read-TS(X) and write-TS(X) are set to the timestamp of the most recent corresponding transaction that accessed resource X

Timestamp Ordering

Transactions are ordered based on their timestamps

- Schedule is serialisable
- Equivalent serial schedule has the transactions in order of their timestamps

For each resource accessing by conflicting operations, the order in which the resource is accessed must not violate the serialisability order

Basic Timestamp Ordering

TS(T) is compared with read-TS(X) and write-TS(X)

- Has this item been read or written before transaction T has had an opportunity to read/write?
- Ensure that timestamp ordering is not violated

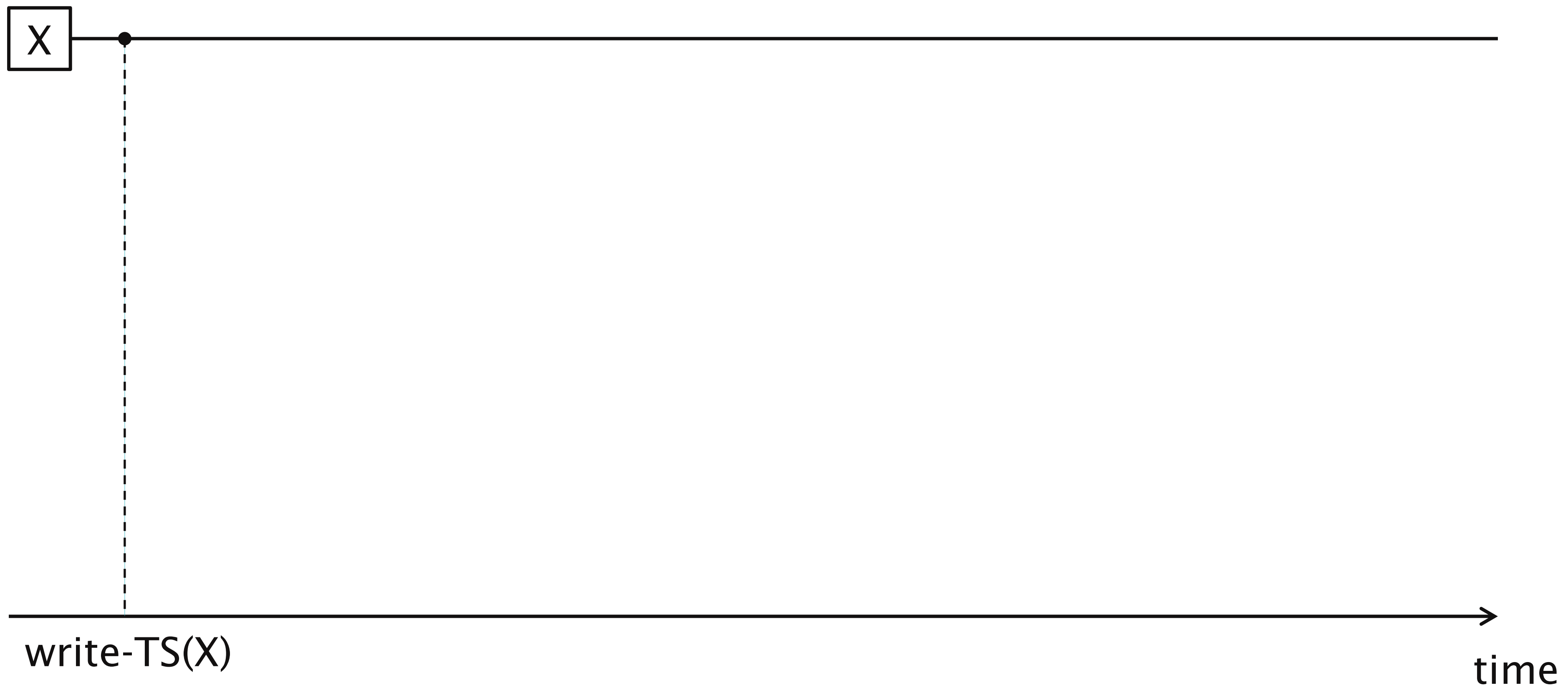
If timestamp ordering is violated, transaction is aborted and resubmitted with a new timestamp

Basic Timestamp Ordering: write(X)

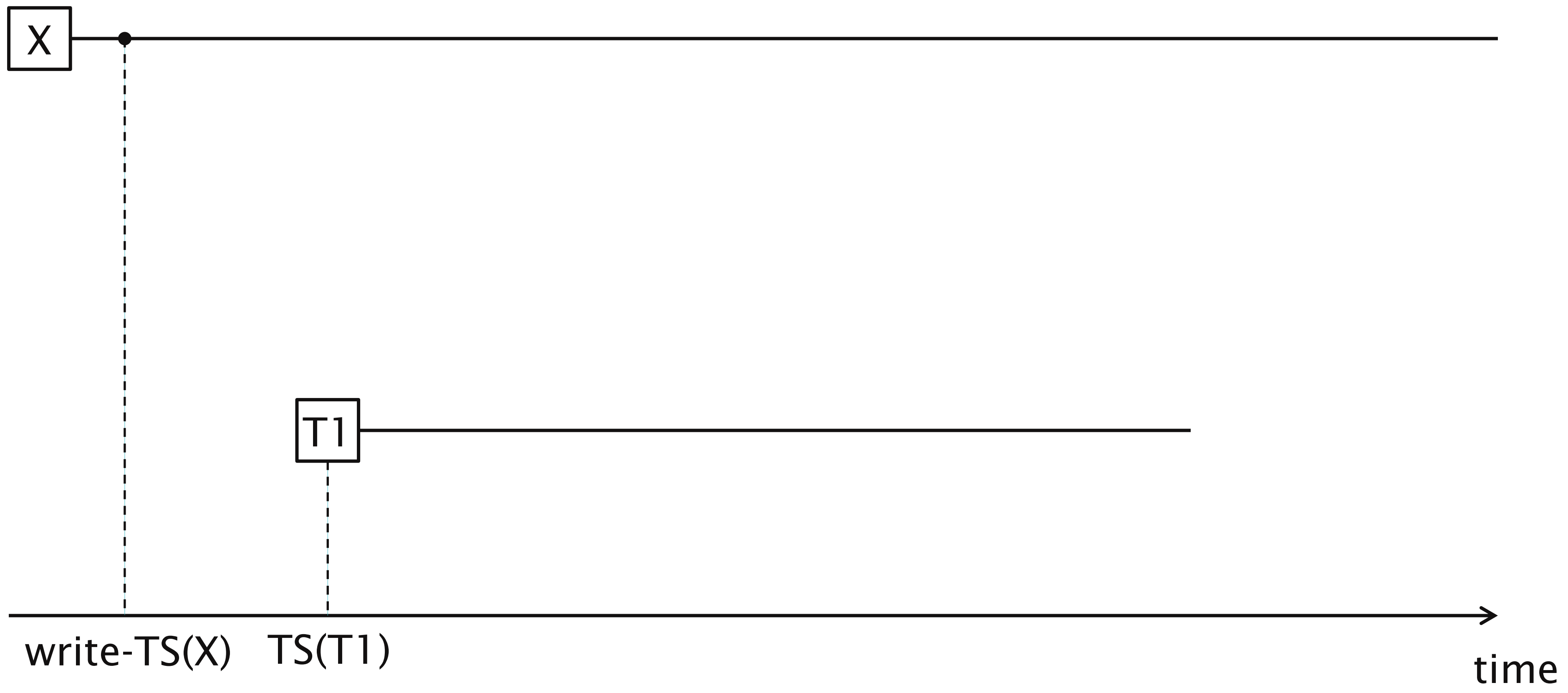
```
if read-TS(X) > TS(T) or write-TS(X) > TS(T)
then
      abort and rollback T and reject operation
else
      execute write(X)
      set write-TS(X) to TS(T)
```

Basic Timestamp Ordering

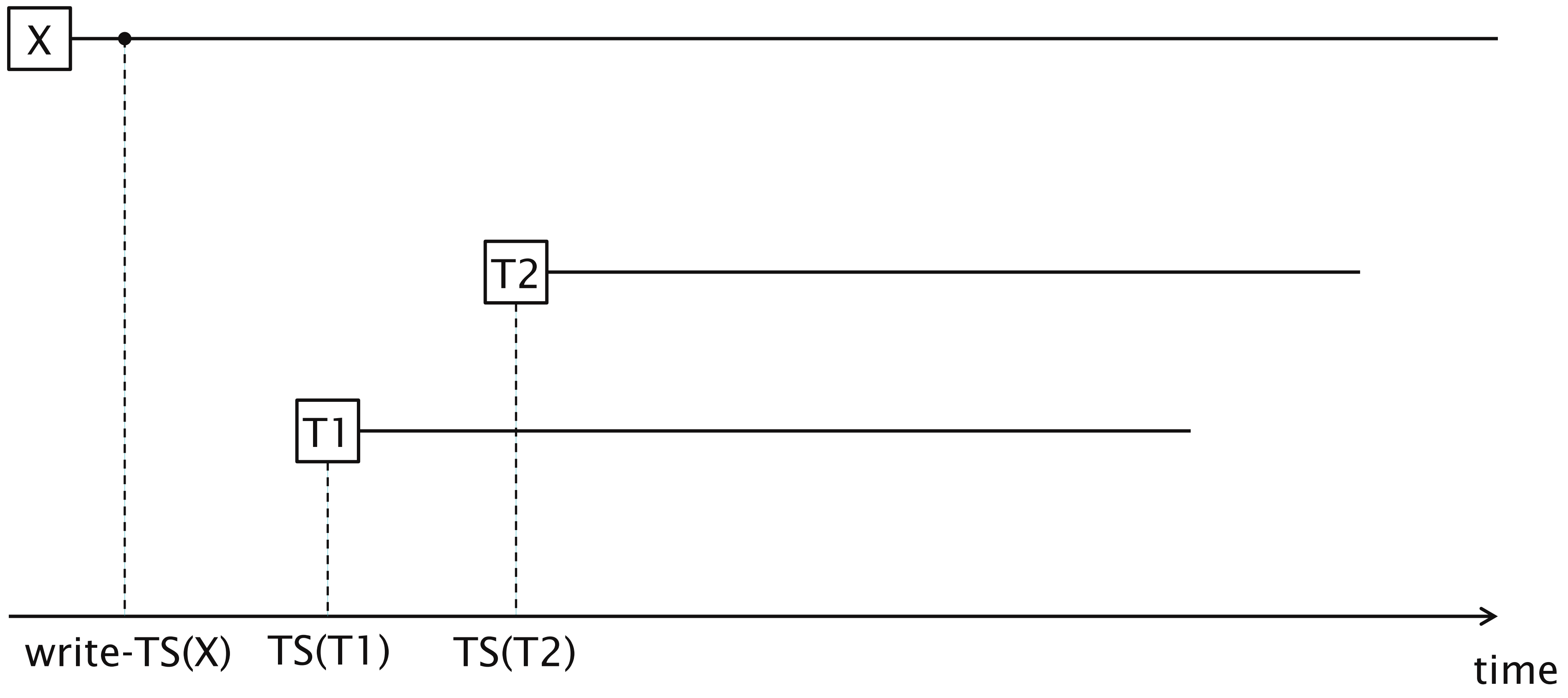
Basic Timestamp Ordering



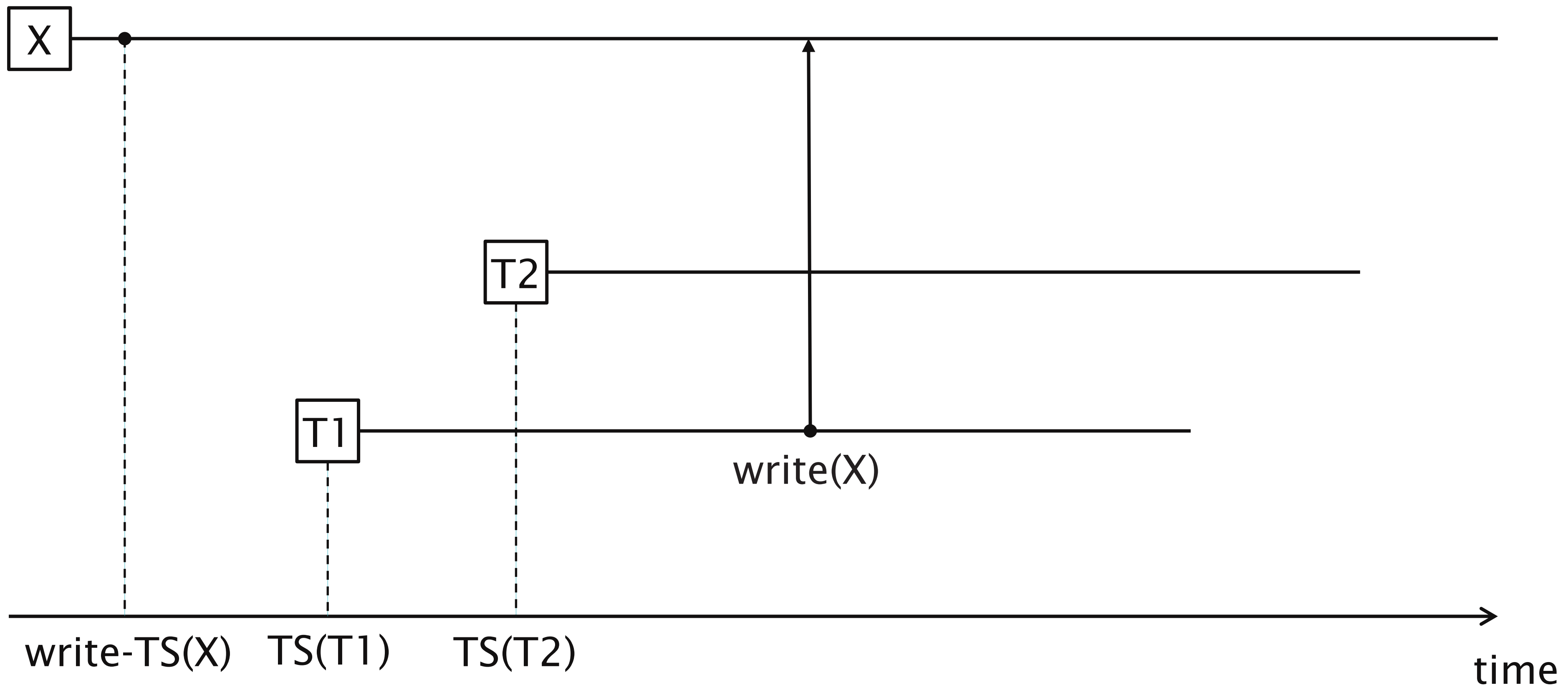
Basic Timestamp Ordering



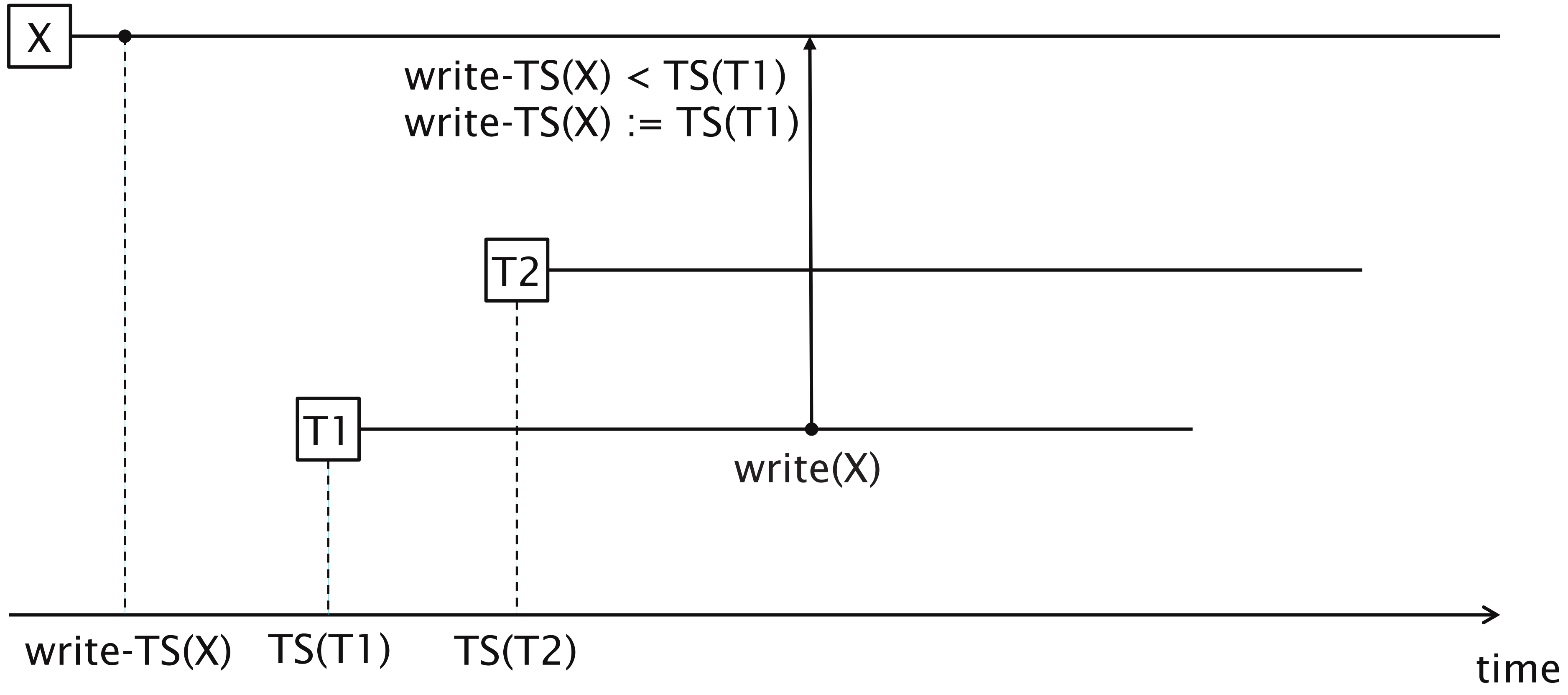
Basic Timestamp Ordering



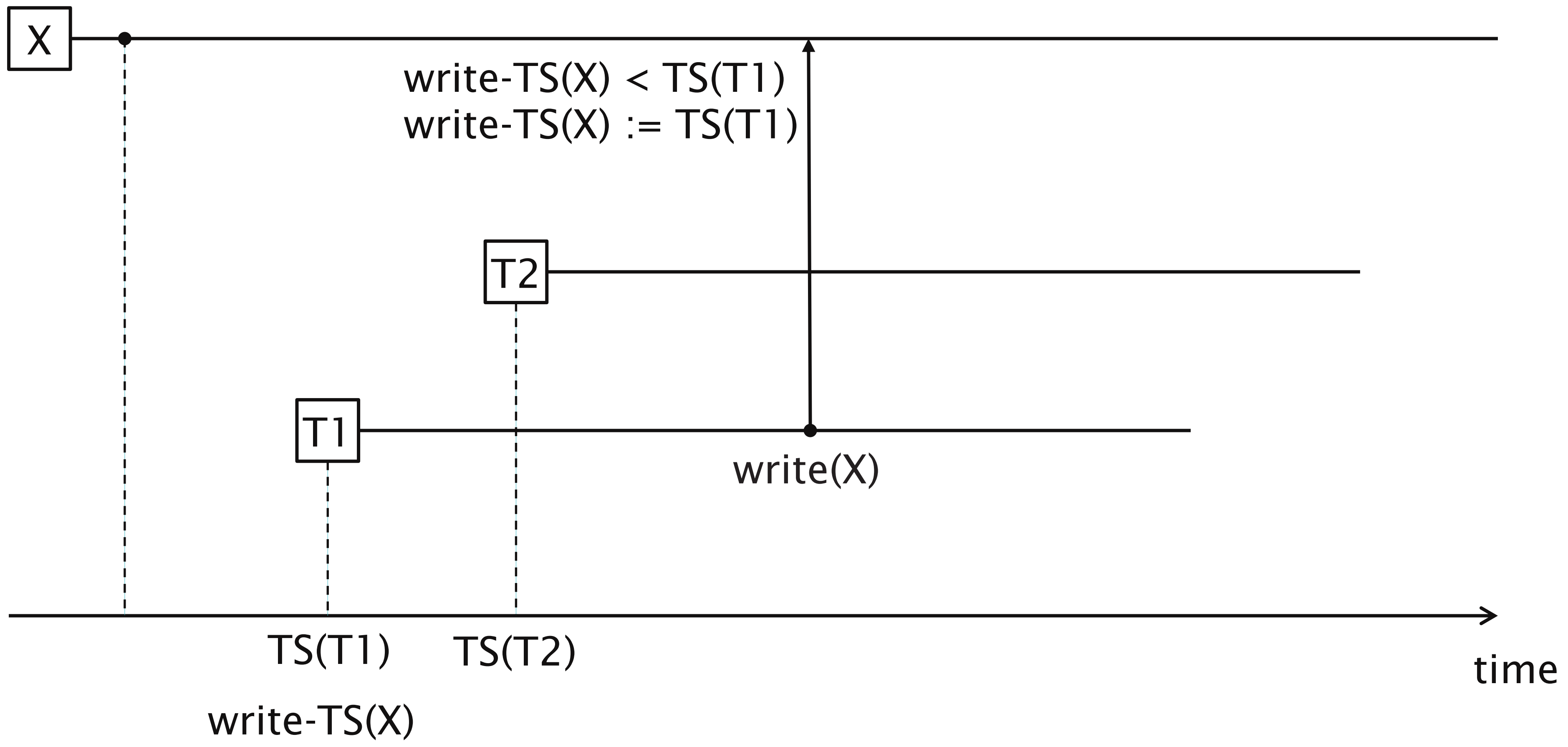
Basic Timestamp Ordering



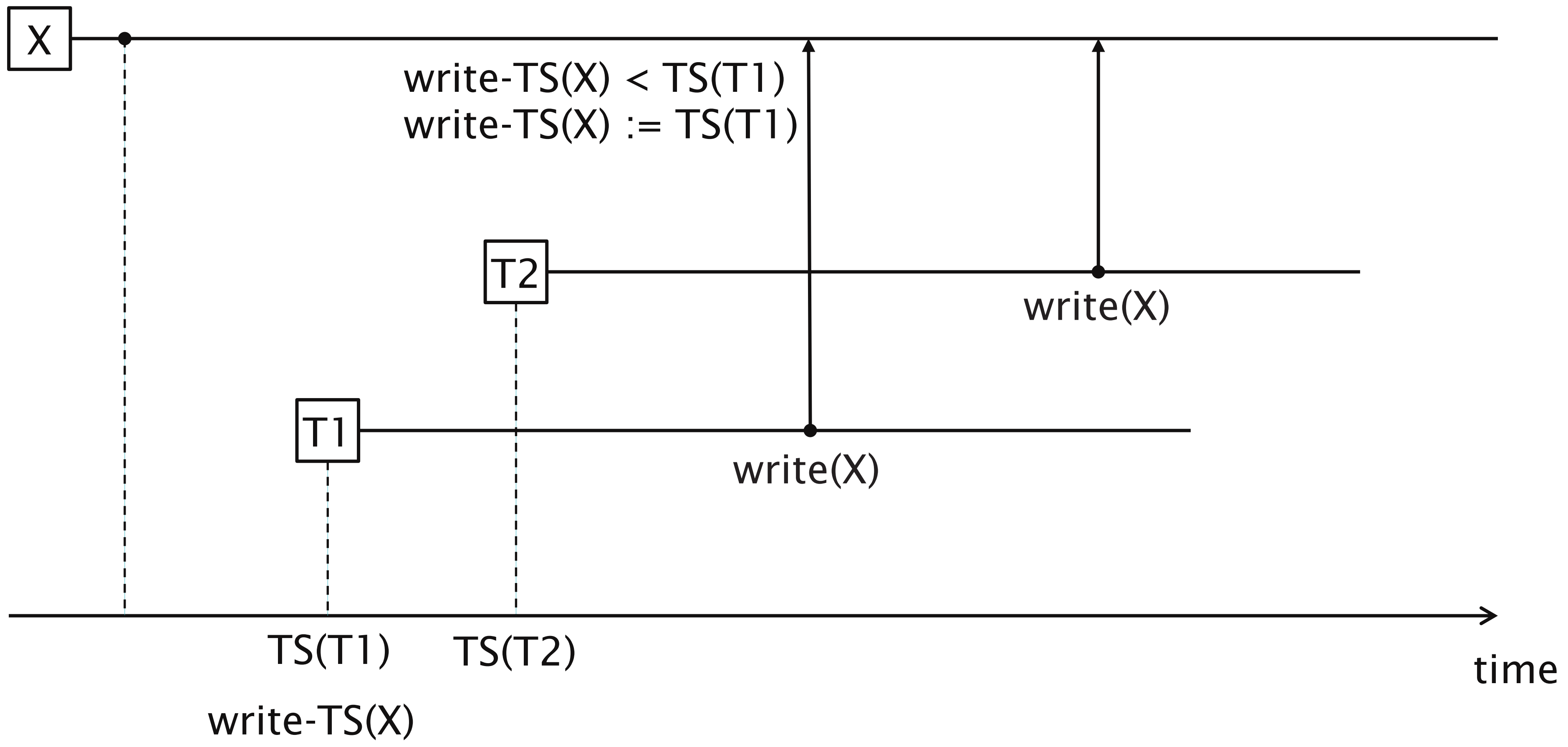
Basic Timestamp Ordering



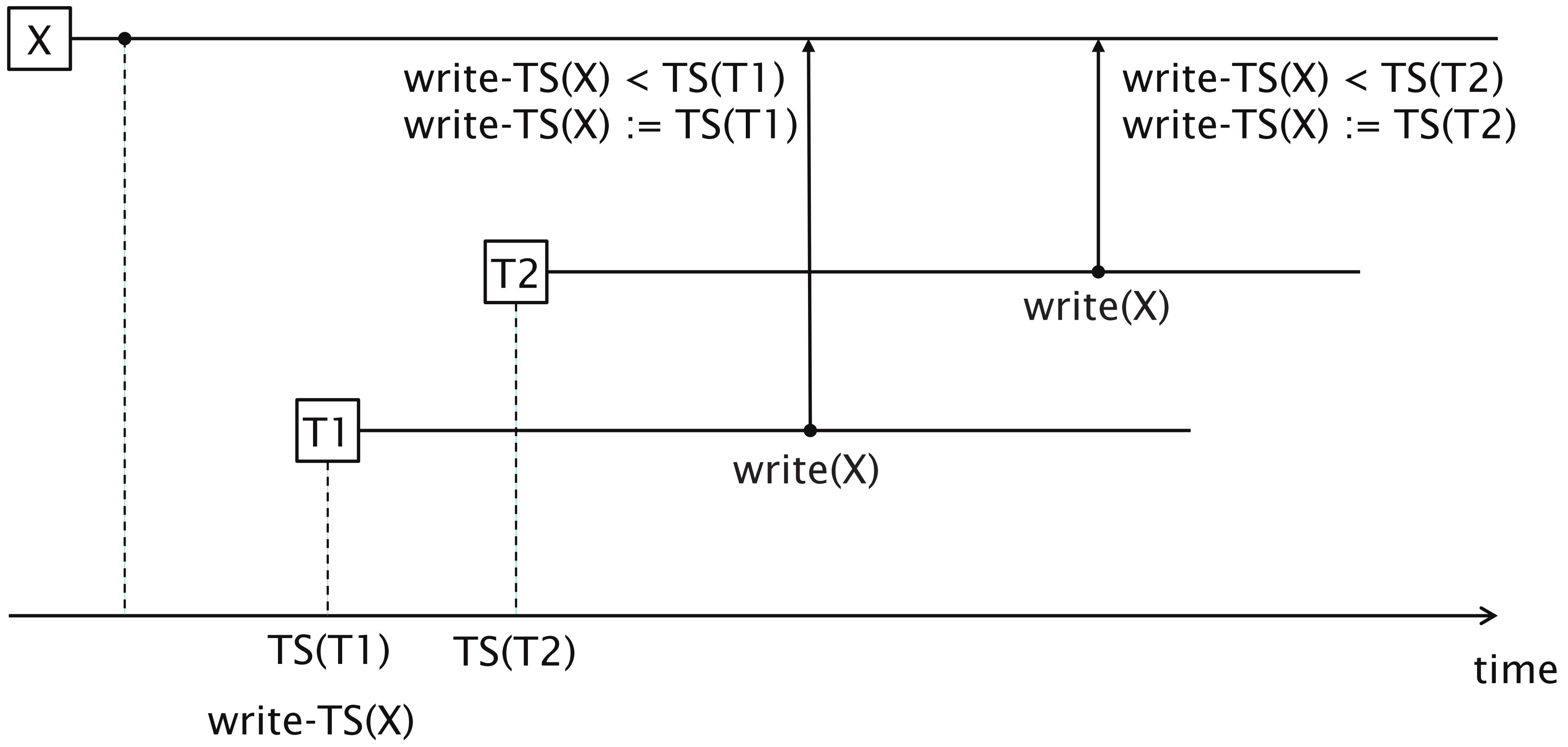
Basic Timestamp Ordering



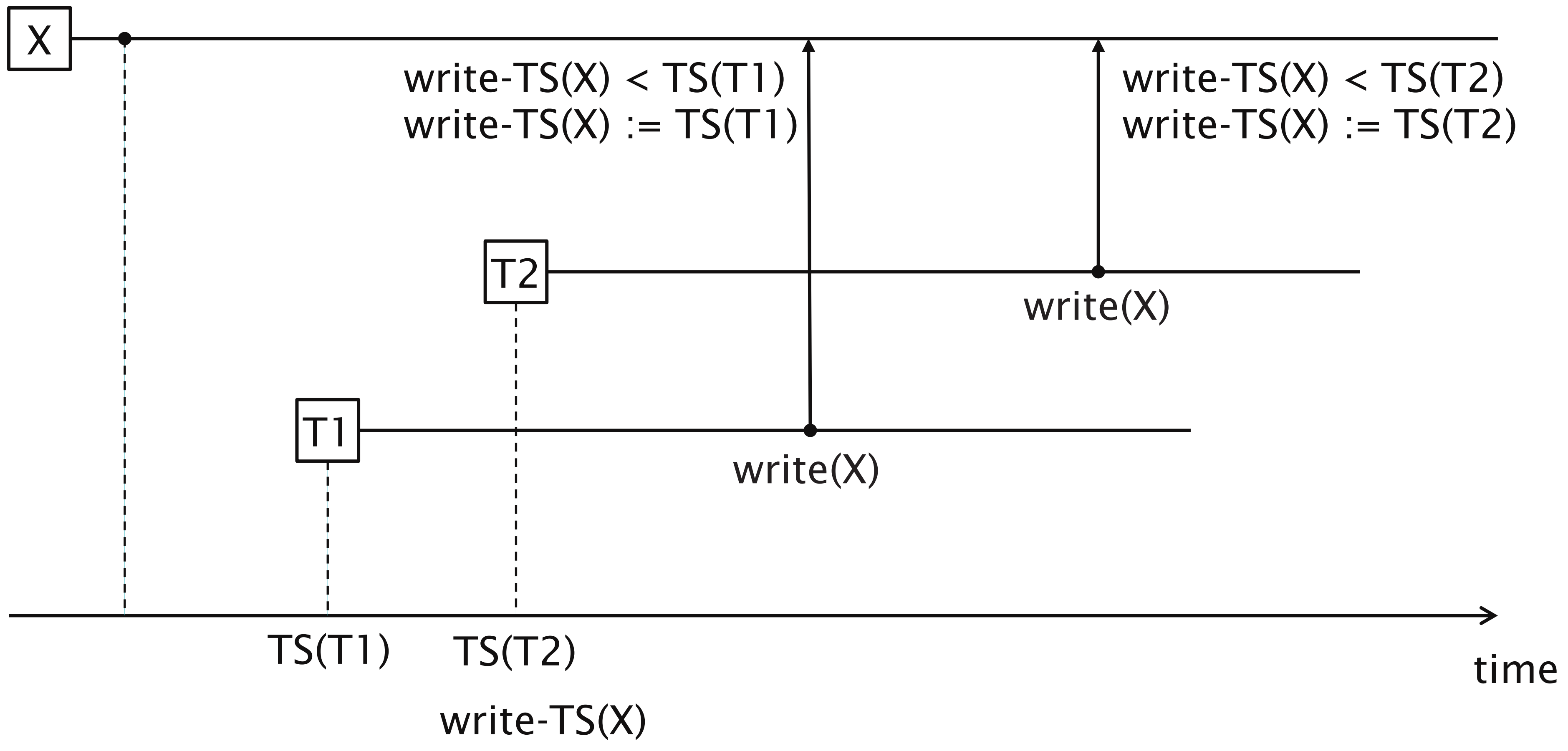
Basic Timestamp Ordering



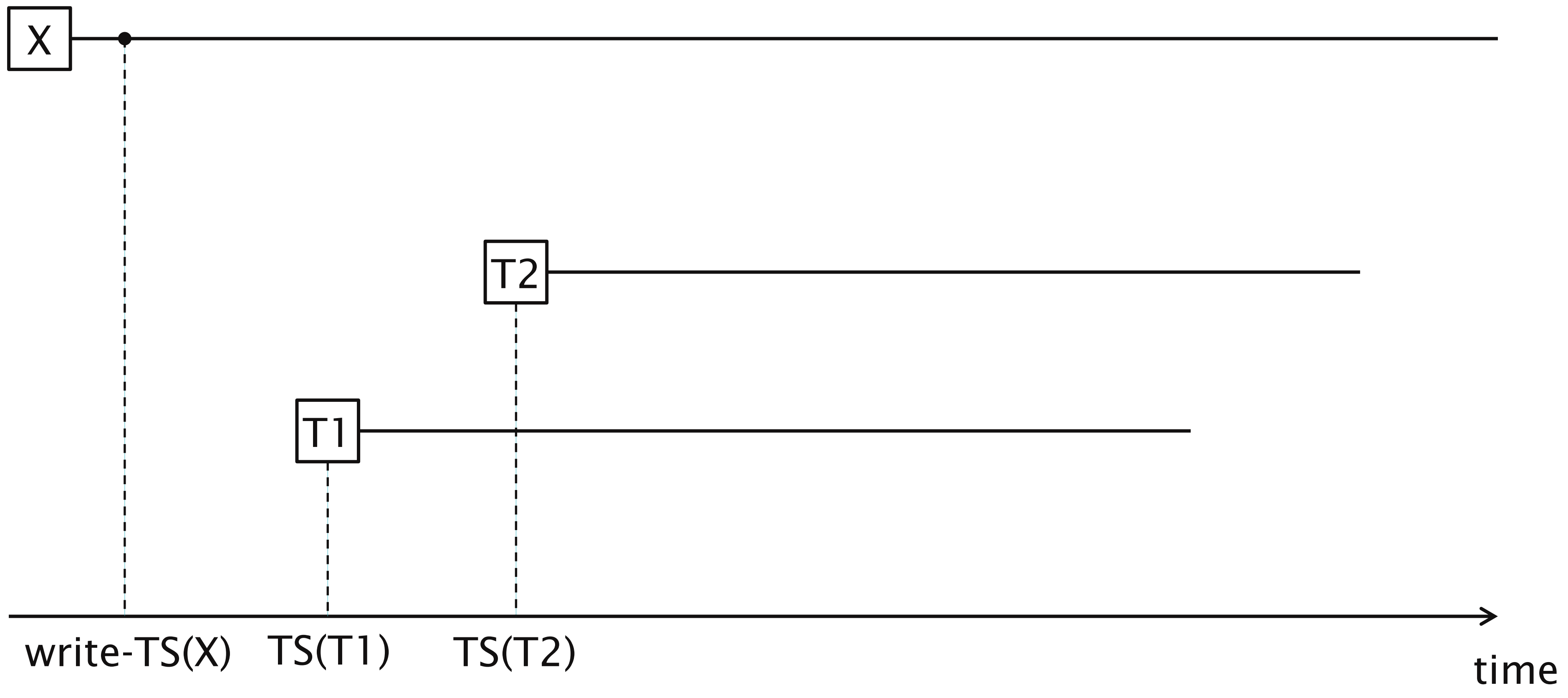
Basic Timestamp Ordering



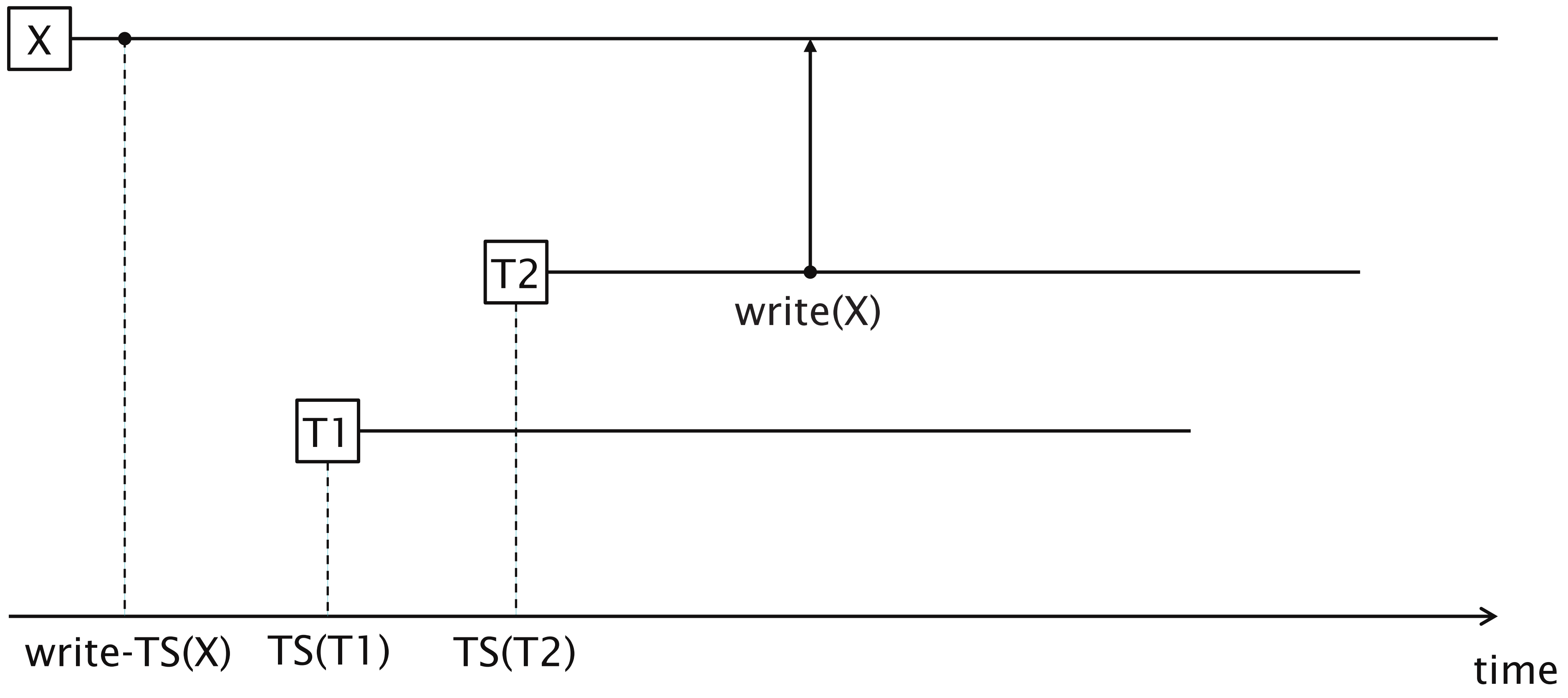
Basic Timestamp Ordering



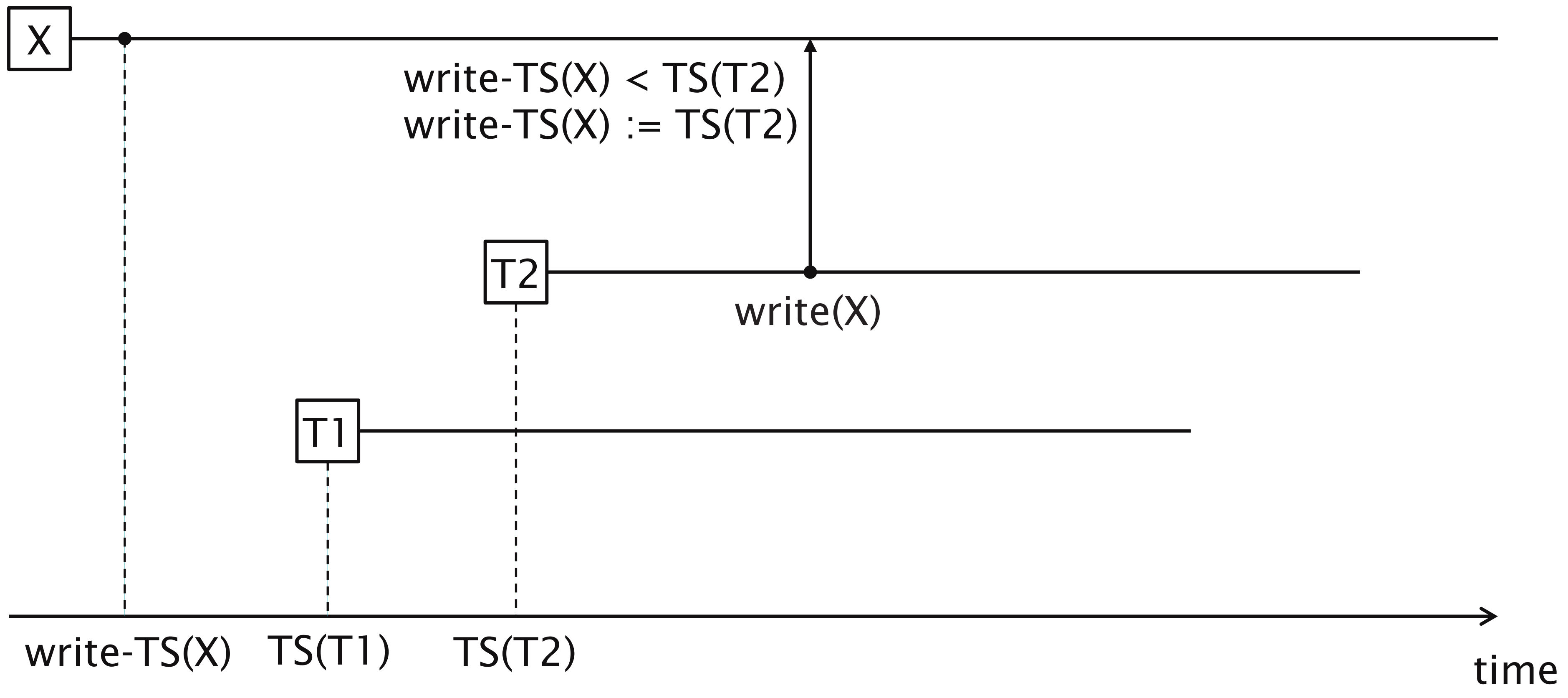
Basic Timestamp Ordering



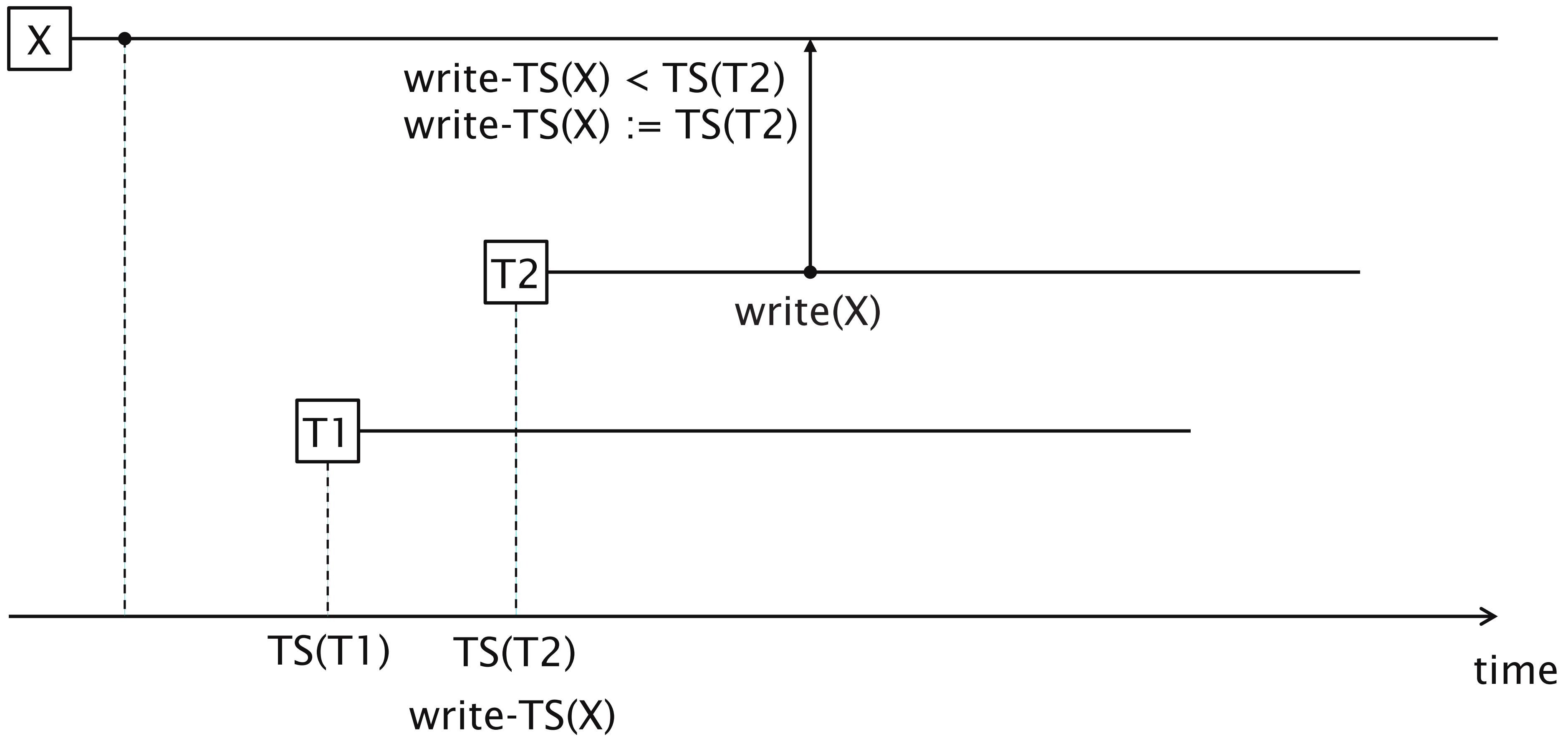
Basic Timestamp Ordering



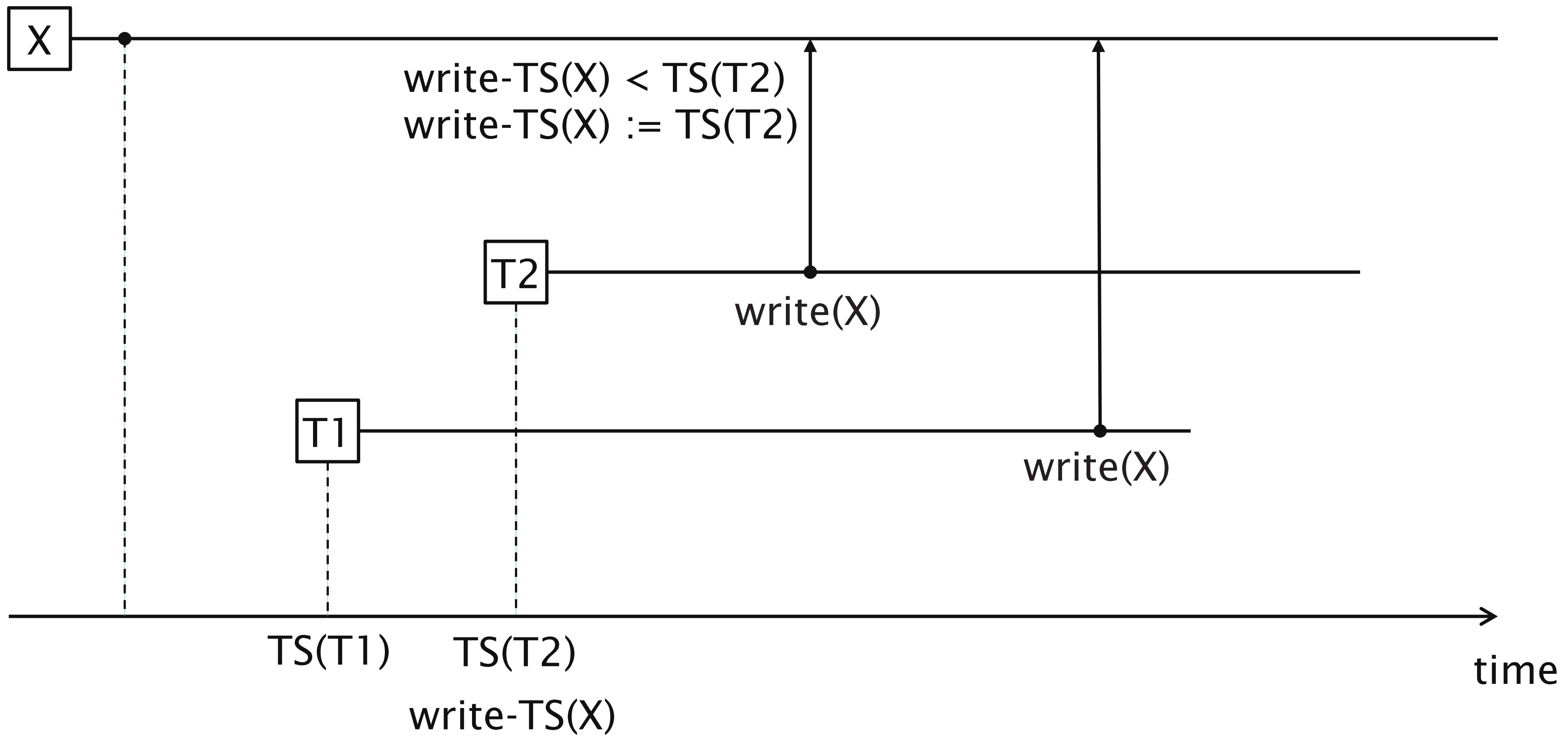
Basic Timestamp Ordering



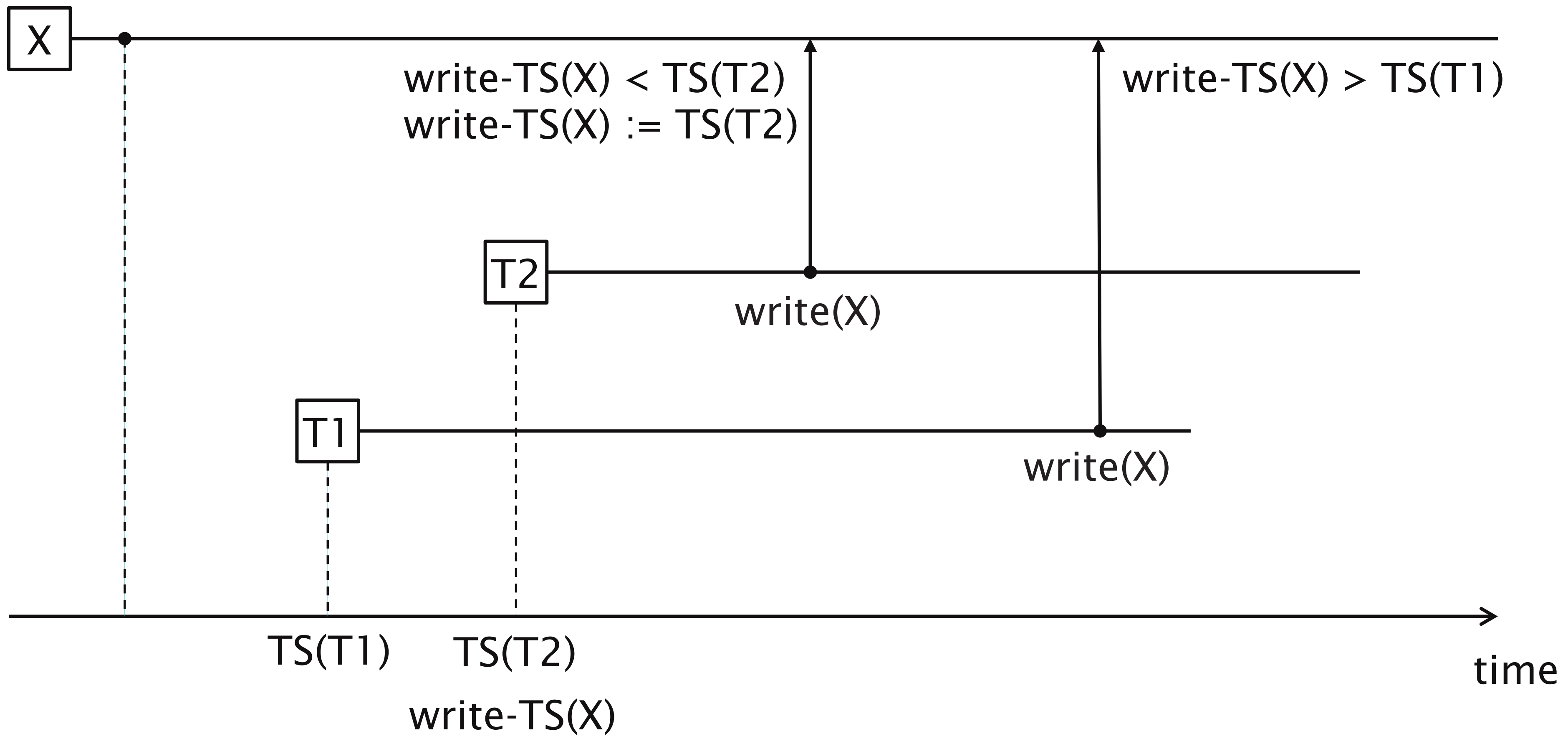
Basic Timestamp Ordering



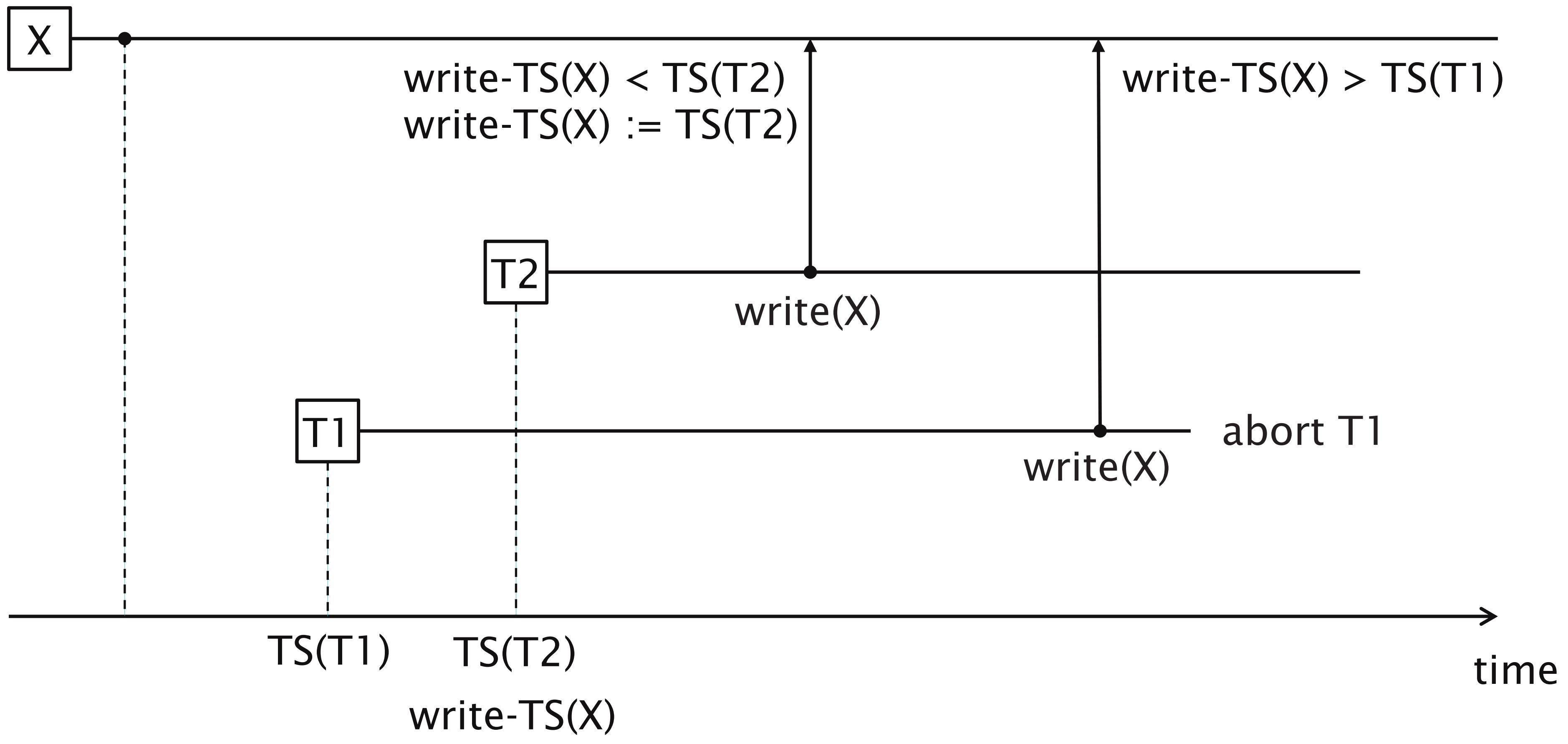
Basic Timestamp Ordering



Basic Timestamp Ordering



Basic Timestamp Ordering



Basic Timestamp Ordering: read(X)

```
if write-TS(X) > TS(T)
then
        abort and rollback T and reject operation
else
        execute read(X)
        set read-TS(X) to max(TS(T), read-TS(X))
```

Thomas's Write Rule

- Modification of Basic TO that rejects fewer write operations
- Weakens the checks for write (X) so that obsolete write operations are ignored
- Does not enforce serialisability

Thomas's Write Rule

```
if read-TS(X) > TS(T)
then
      roll back T and reject operation
if write-TS(X) > TS(T)
then
      do not execute write (X)
      continue processing
else
      execute write(X)
      set write-TS(X) to TS(T)
```

Flat Transactions

Transactions considered so far are flat transactions

- Basic building block
- Only one level of control by the application
- All-or-nothing (commit or abort)
- The simplest type of transaction!

Long Duration Transactions

Transactions considered so far are short duration

- Banking or ticket reservations as example applications
- Transactions complete in minutes, if not seconds

Long-lived transactions present particular challenges

- More susceptible to failure (and rollback not acceptable)
- May lock and access many data items (increases chance of deadlock)

Savepoints

Savepoints

Savepoint: an identifiable point in a flat transaction representing a partially consistent state which can be used as an internal restart point for the transaction

Used for deadlock handling

- partially rollback transaction in order to release required locks

Savepoints may be persistent

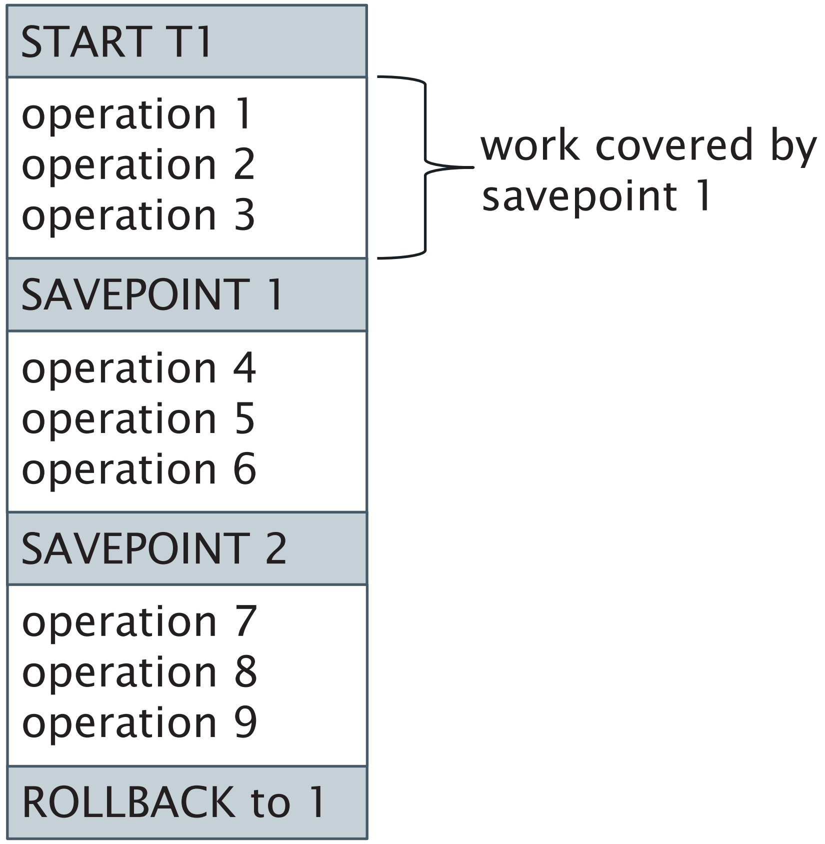
- Following a system crash, restart active transactions from their most recent savepoints

UNIVERSITY OF Southampton

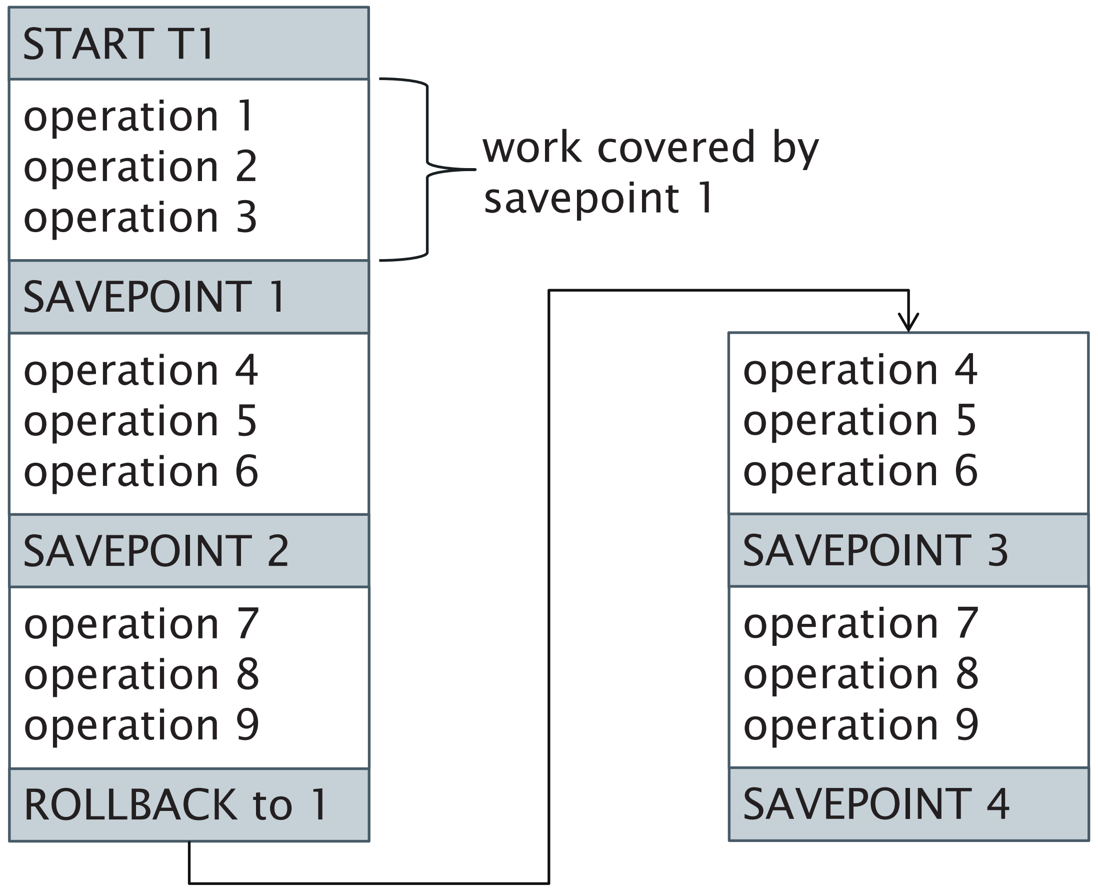
Savepoints

START T1
operation 1 operation 2 operation 3
SAVEPOINT 1
operation 4 operation 5 operation 6
SAVEPOINT 2
operation 7 operation 8 operation 9
ROLLBACK to 1

Savepoints


START T1
operation 1
operation 2
operation 3
SAVEPOINT 1
operation 4
operation 5
operation 6
SAVEPOINT 2
operation 7
operation 8
operation 9
ROLLBACK to 1
work covered by savepoint 1

Savepoints


START T1
operation 1
operation 2
operation 3
work covered by savepoint 1
SAVEPOINT 1
operation 4
operation 5
operation 6
SAVEPOINT 2
operation 7
operation 8
operation 9
ROLLBACK to 1
operation 4
operation 5
operation 6
SAVEPOINT 3
operation 7
operation 8
operation 9
SAVEPOINT 4

Chained Transactions

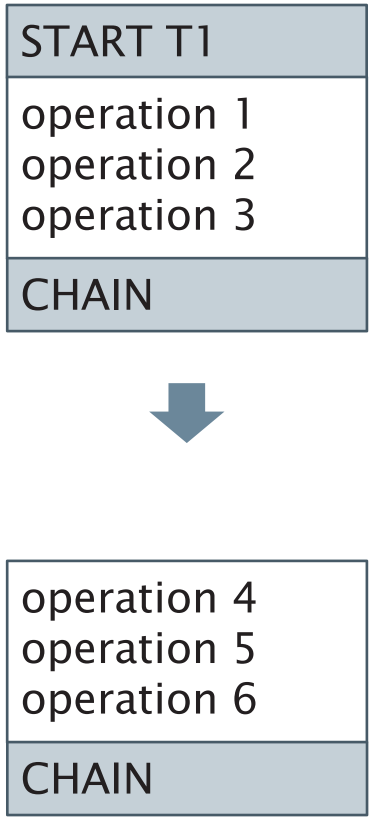
Chained Transactions

Transaction broken into subtransactions which are executed serially

On chaining to the next subtransaction:

- commit results
- keep (some) locks

Cannot rollback to previous subtransaction



Savepoints versus Chained Transactions

- Both allow substructure to be imposed on a long-running application program
  - Database context is preserved
  - Cursors are kept
- Commit vs Savepoint
  - Chained - rollback only to previous ‘savepoint’
  - Savepoints - can rollback to arbitrary savepoint
- Locks
  - Chained frees unwanted locks

Savepoints versus Chained Transactions

- Work lost
  - Savepoints more flexible than flat transactions, as long as the system does not crash
- Restart
  - Chained transactions can restart from most recent commit, as long as no processing context hidden in local programming variables
- Both organise work into a sequence of actions

Nested Transactions

Nested Transactions

Transaction forms a hierarchy of subtransactions
(partial order on set of subtransactions)

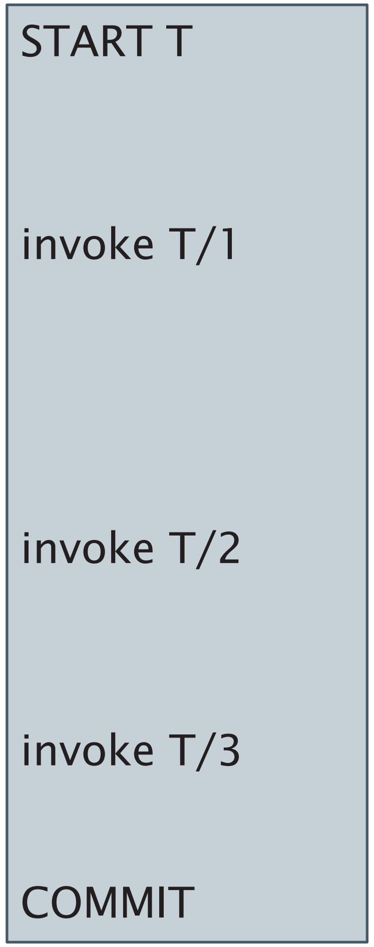
Subtransactions may abort without aborting their parent transaction

- May restart subtransaction

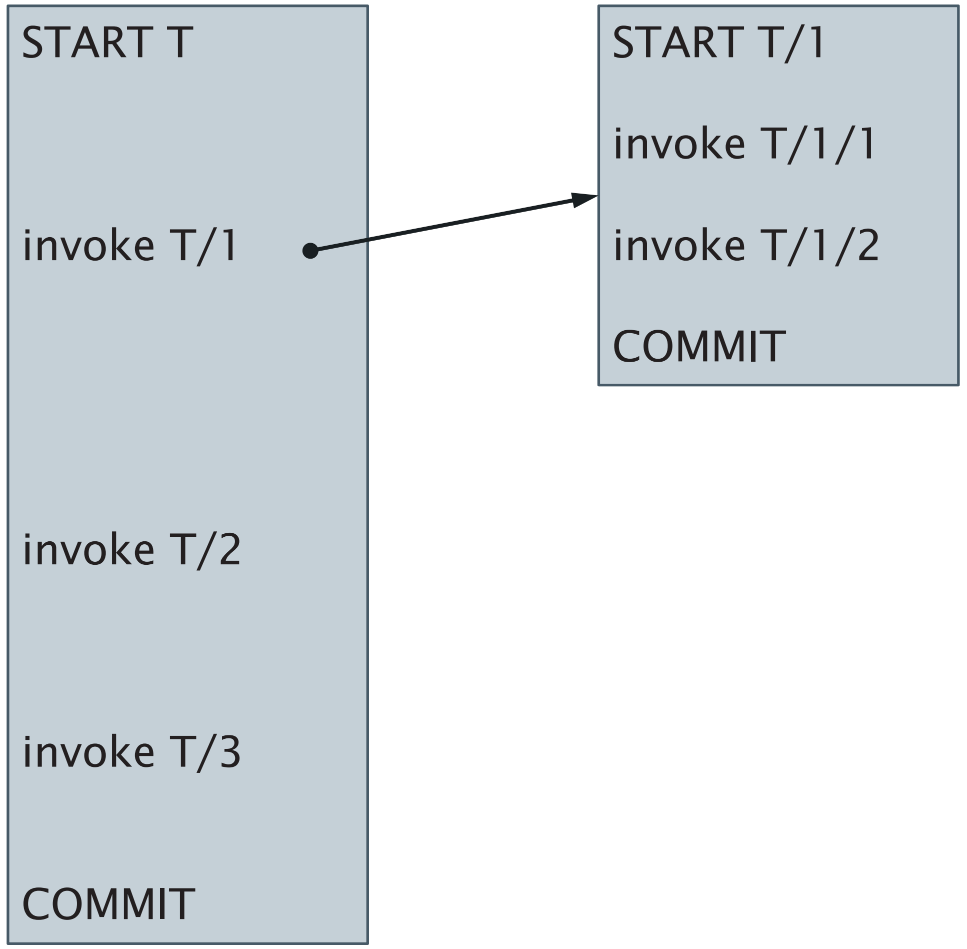
Three rules for nested transactions:

- Commit Rule
- Rollback Rule
- Visibility Rule

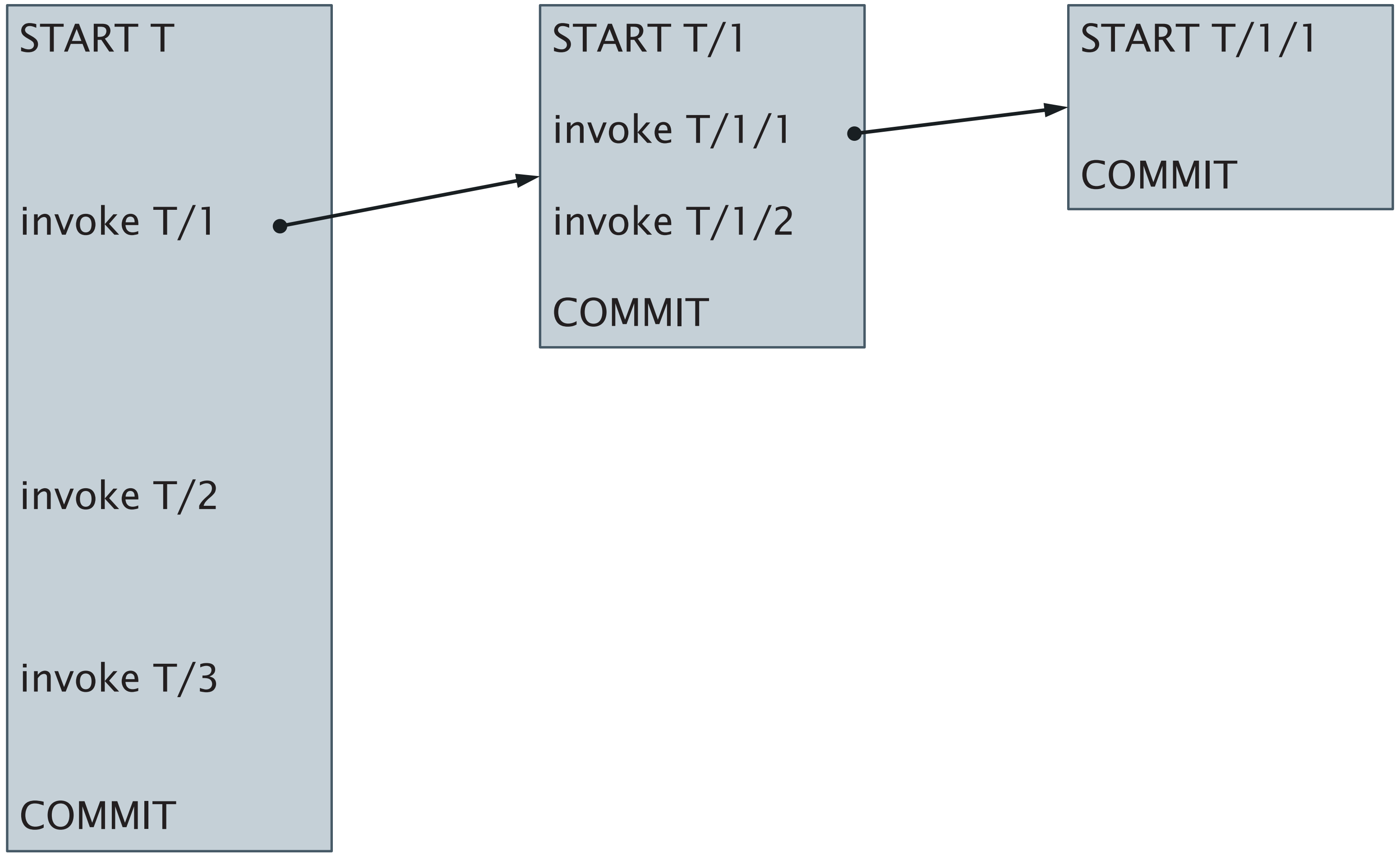
Nested Transactions



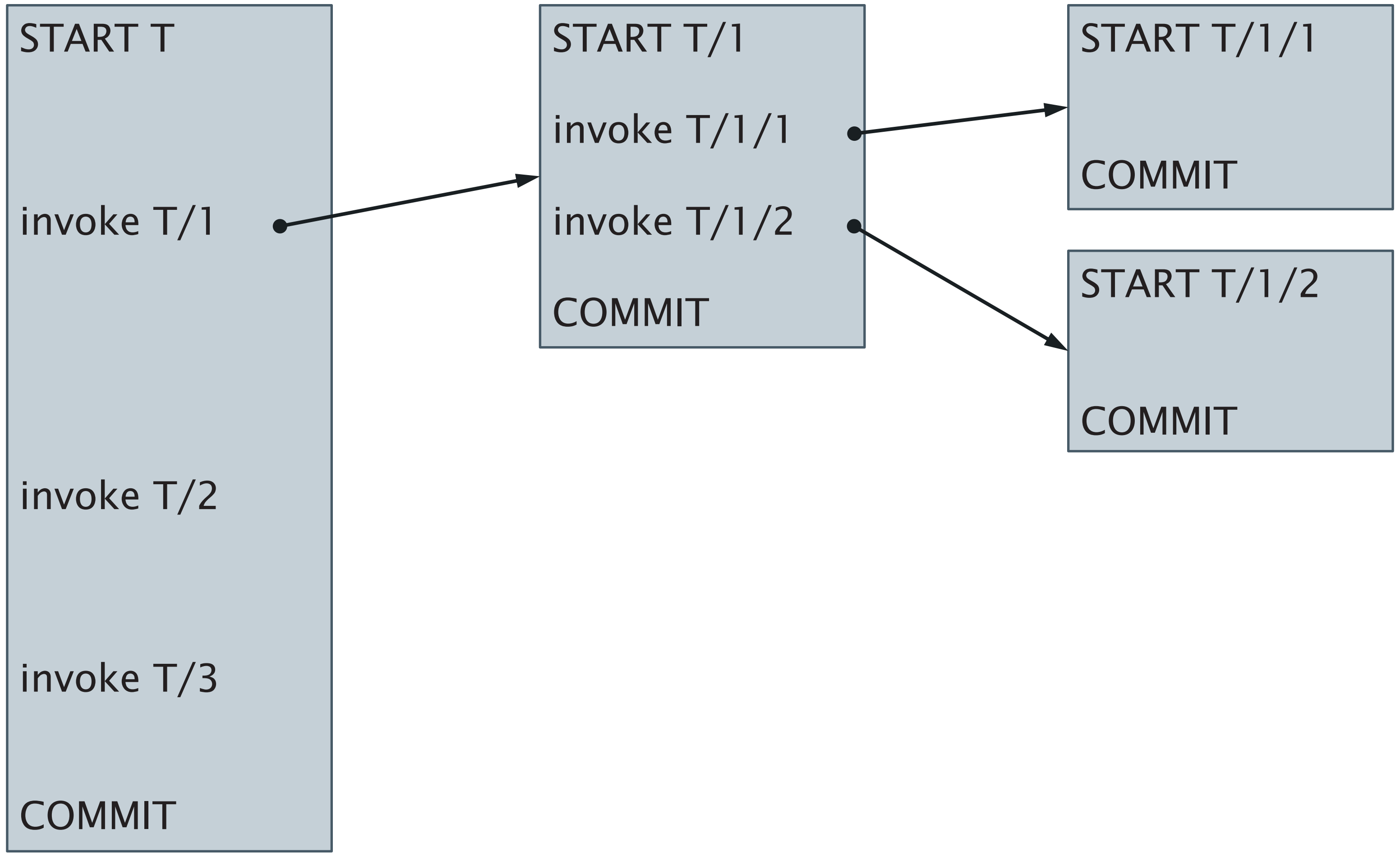
Nested Transactions



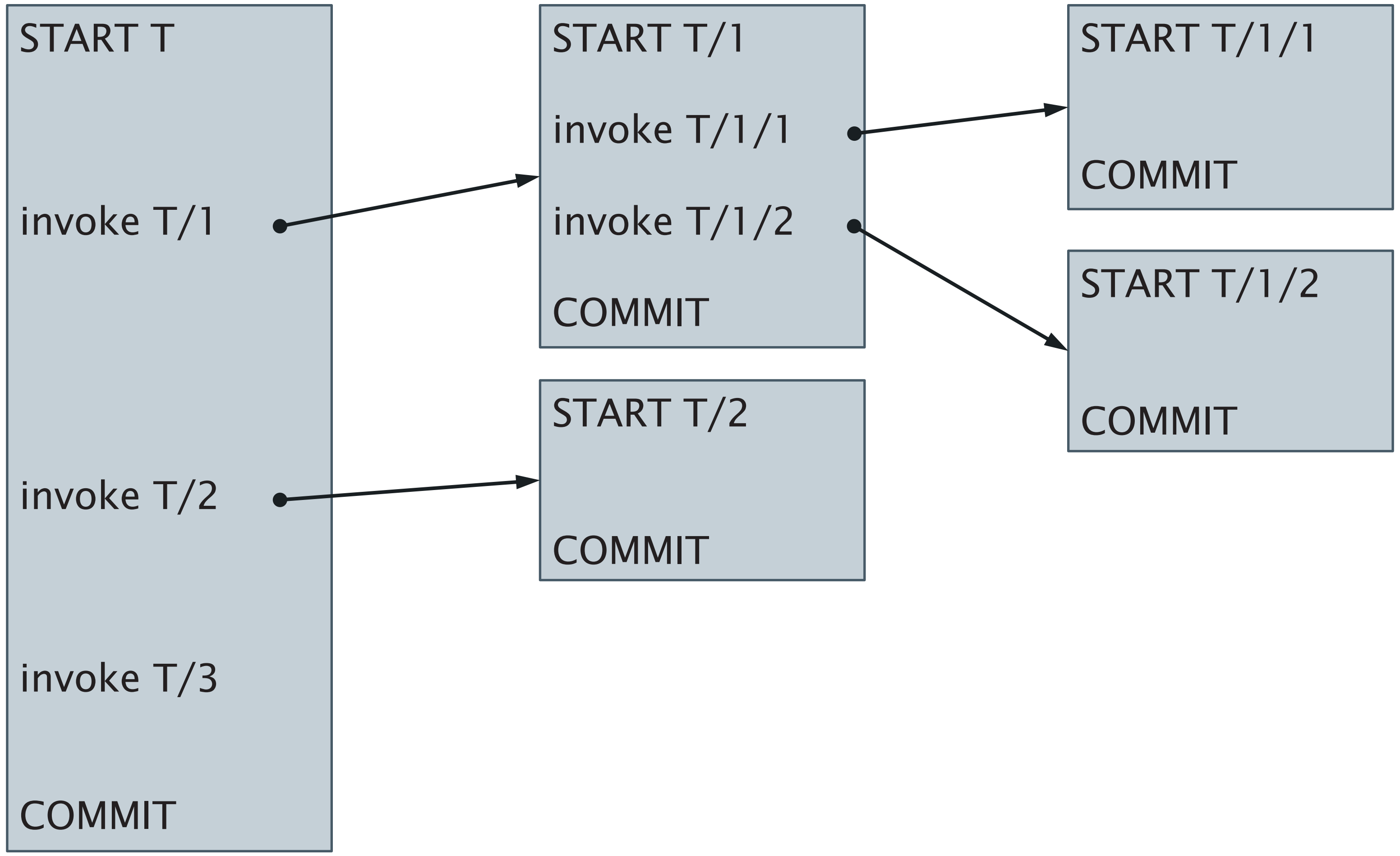
Nested Transactions



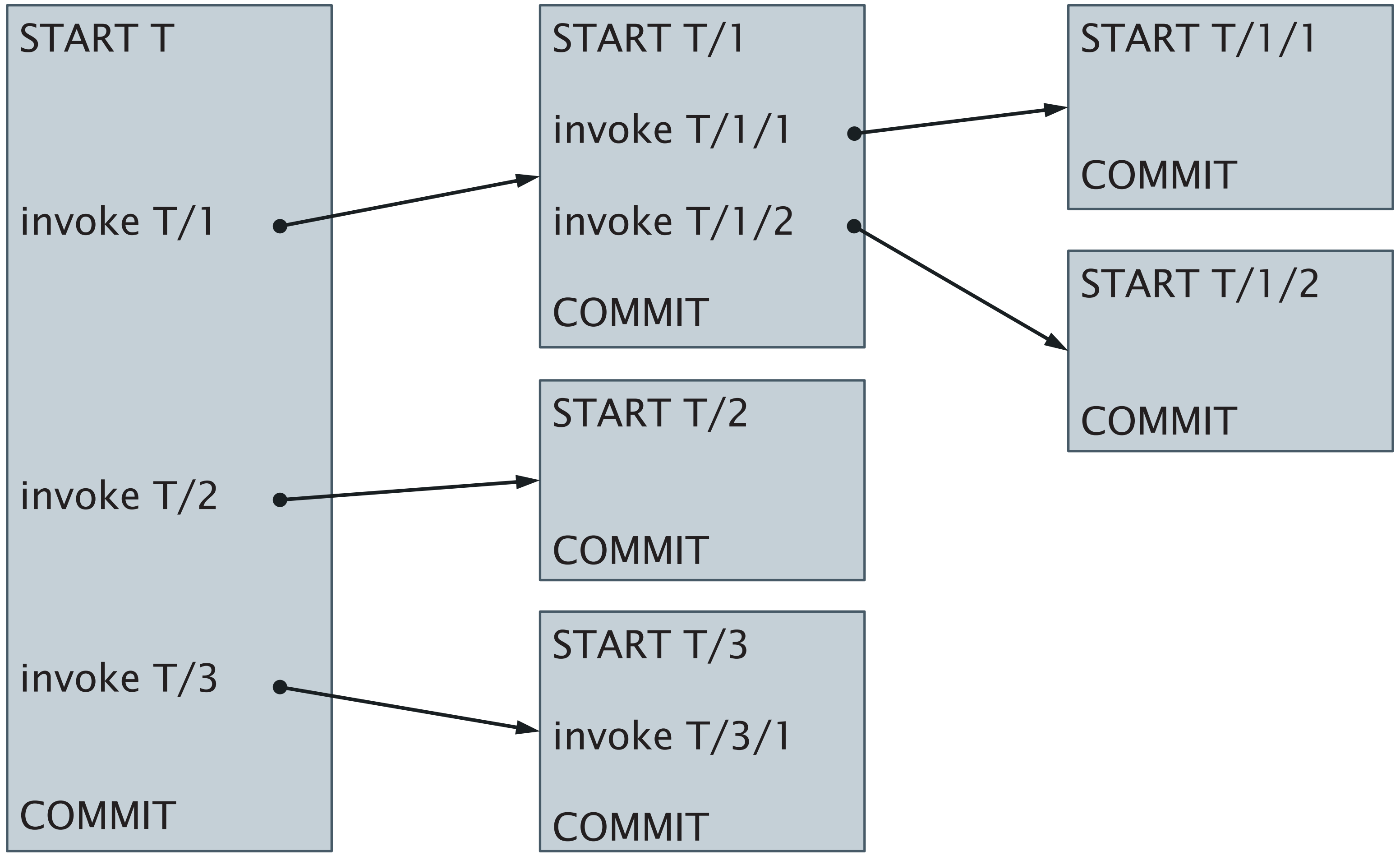
Nested Transactions



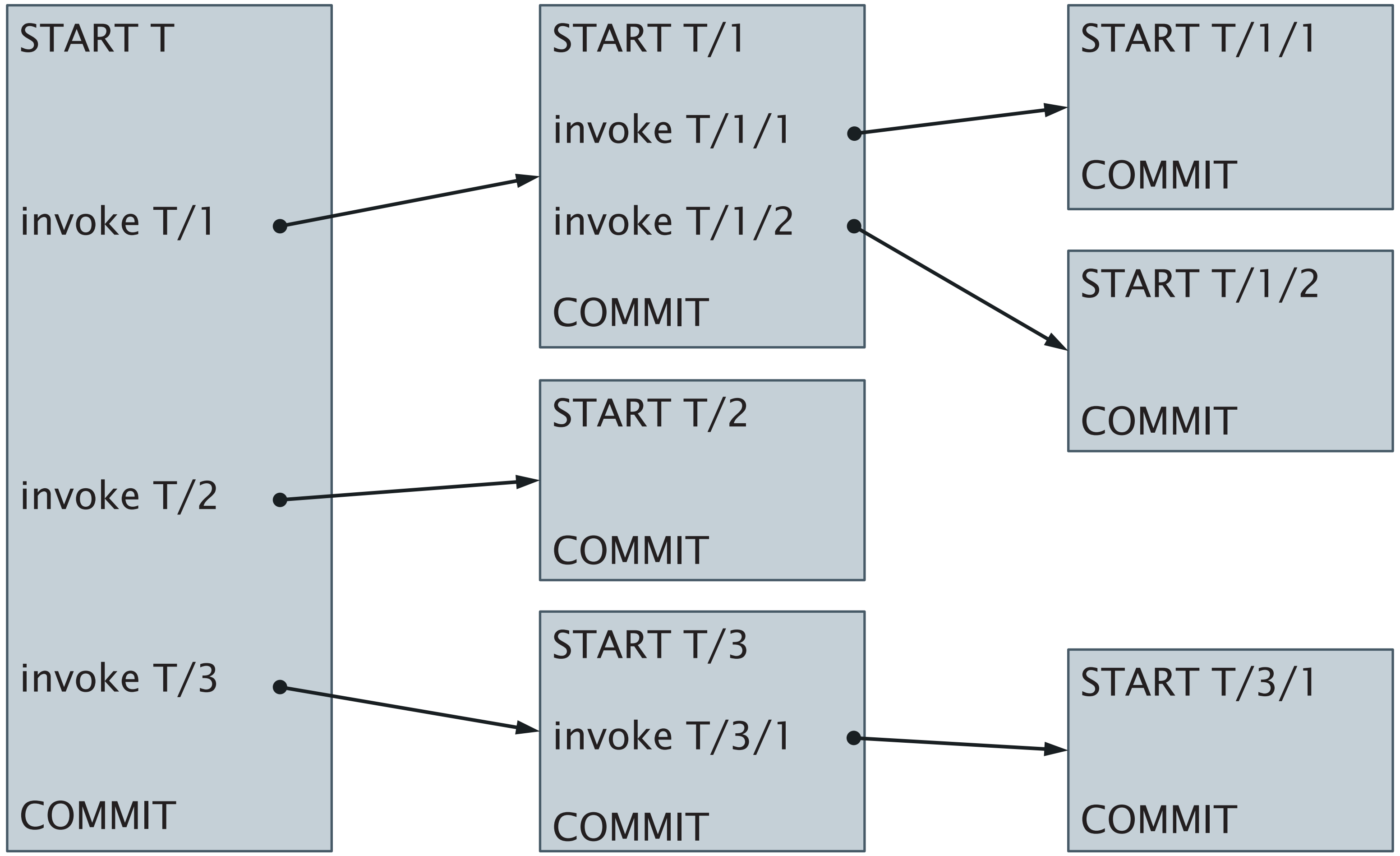
Nested Transactions



Nested Transactions



Nested Transactions



Commit Rule

The commit of a subtransaction makes the results accessible only to the parent

The final commit happens only when all ancestors finally commit

Rollback Rule

If any [sub]transaction rolls back, all of its subtransactions roll back

Visibility Rule

Changes made by a subtransaction are visible to its parent

Objects held by a parent can be made accessible to subtransactions

Changes made by a subtransaction are not visible to its siblings

Observations

Subtransactions are not fully equivalent to flat transactions:

- Atomic
- Consistency preserving
- Isolated
- Not durable, because of the commit rule

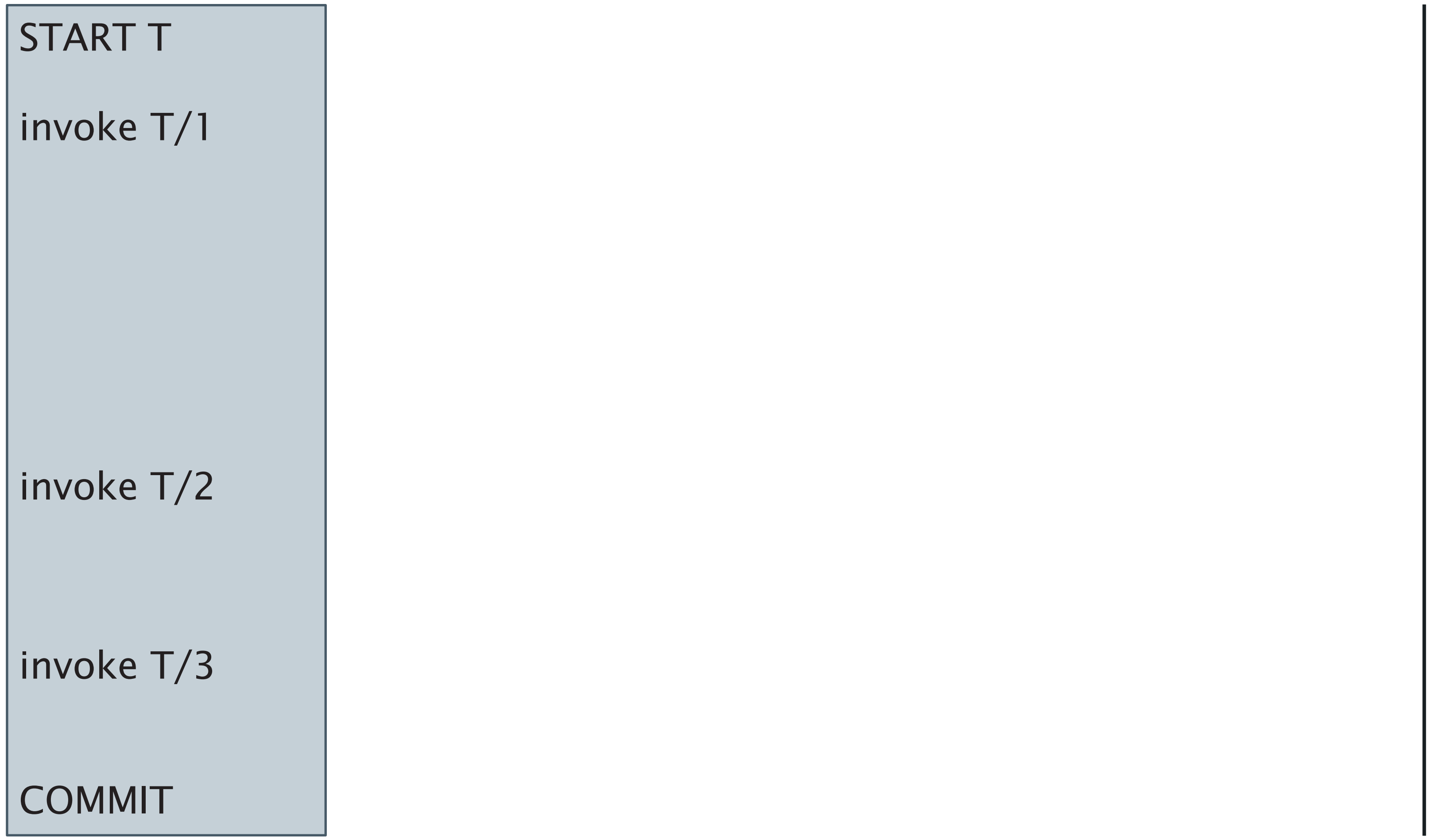
Observations

Nesting and program modularisation complement each other

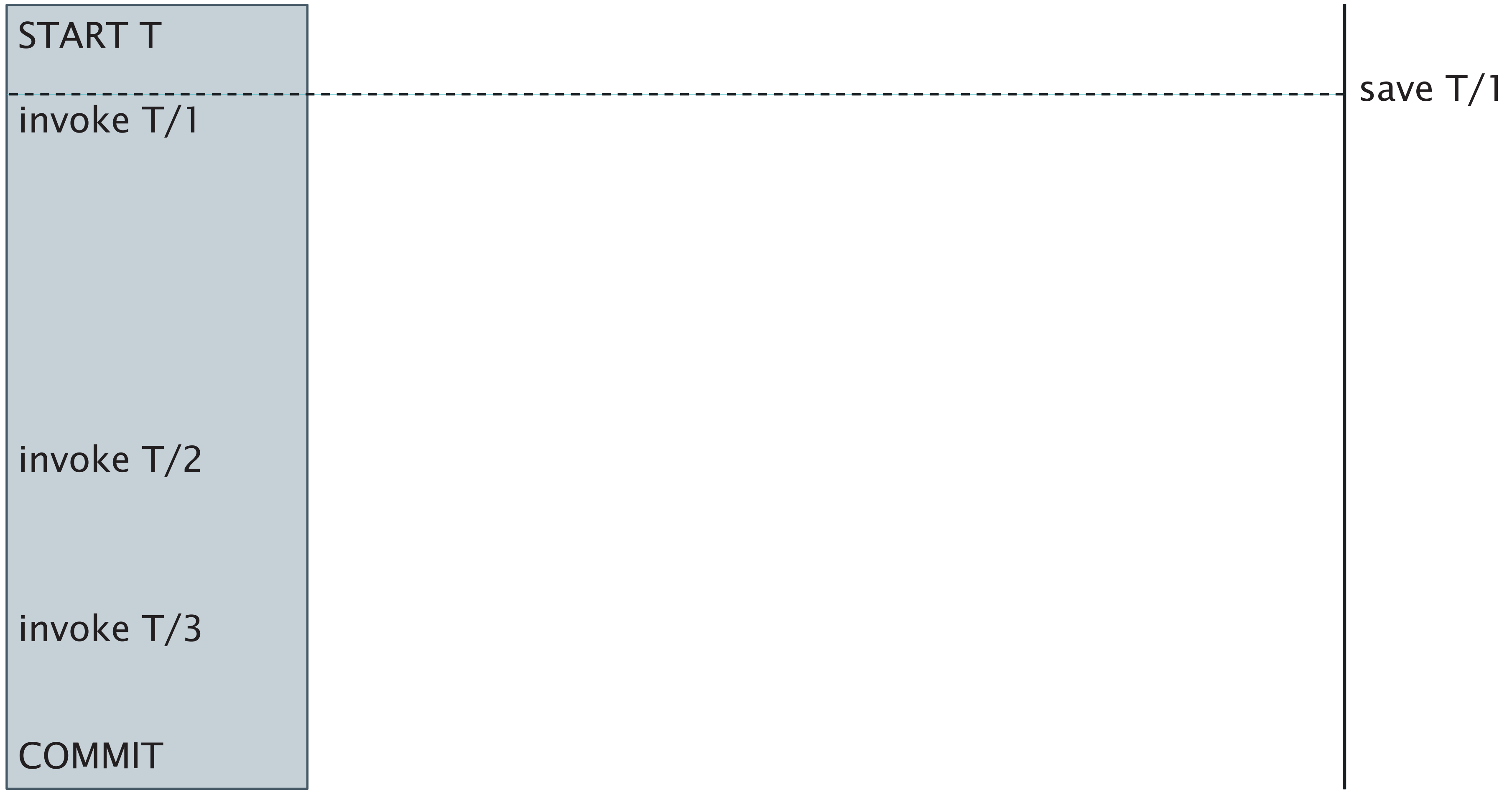
- Well designed module has a clean interface, and no global variables
- If it touches the database, the database is a large global variable
- If the module is protected as a subtransaction, then database changes are kept clean too

Nested transactions permit intra-transaction parallelism

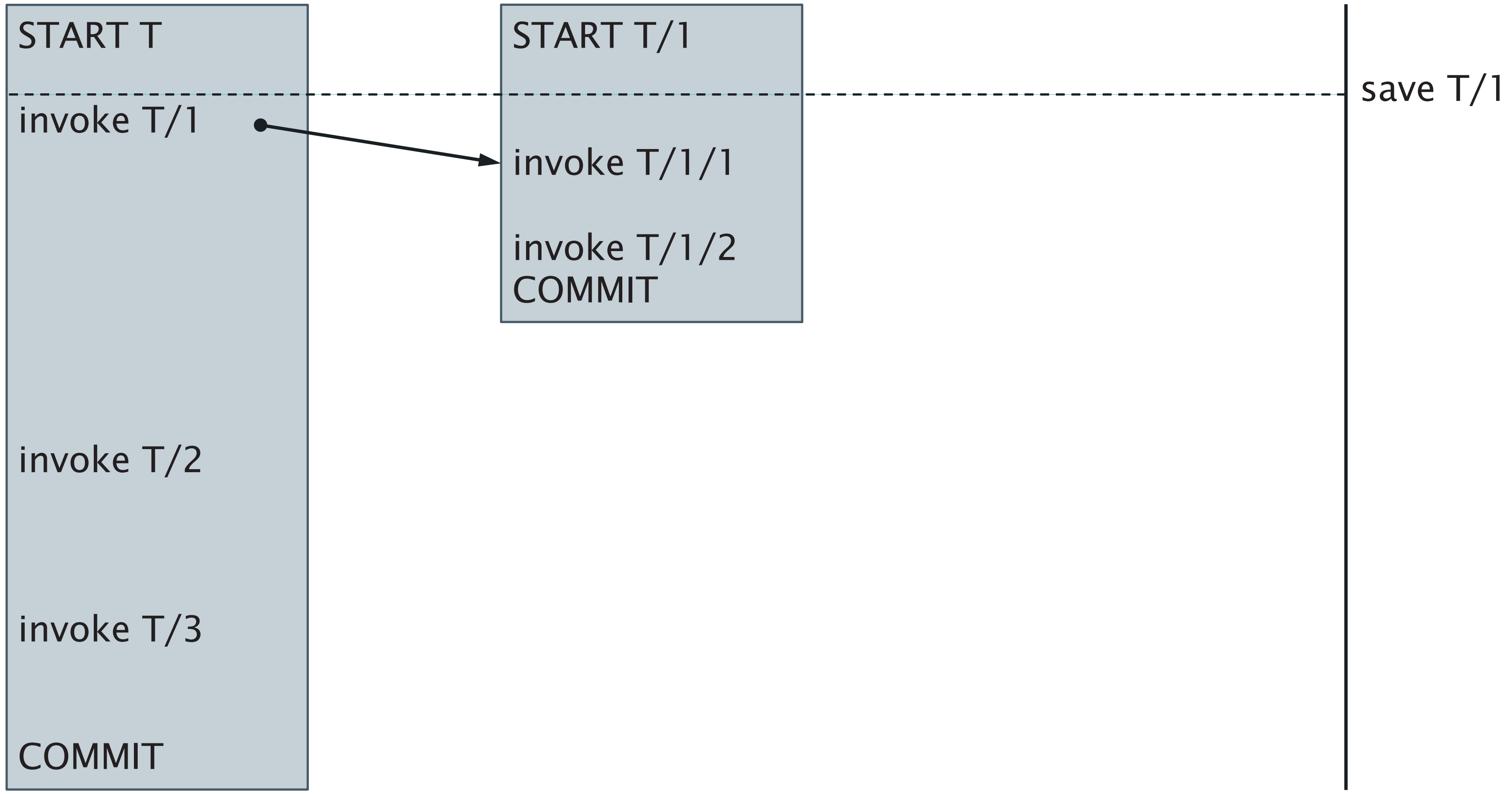
Emulating Nesting with Savepoints



Emulating Nesting with Savepoints

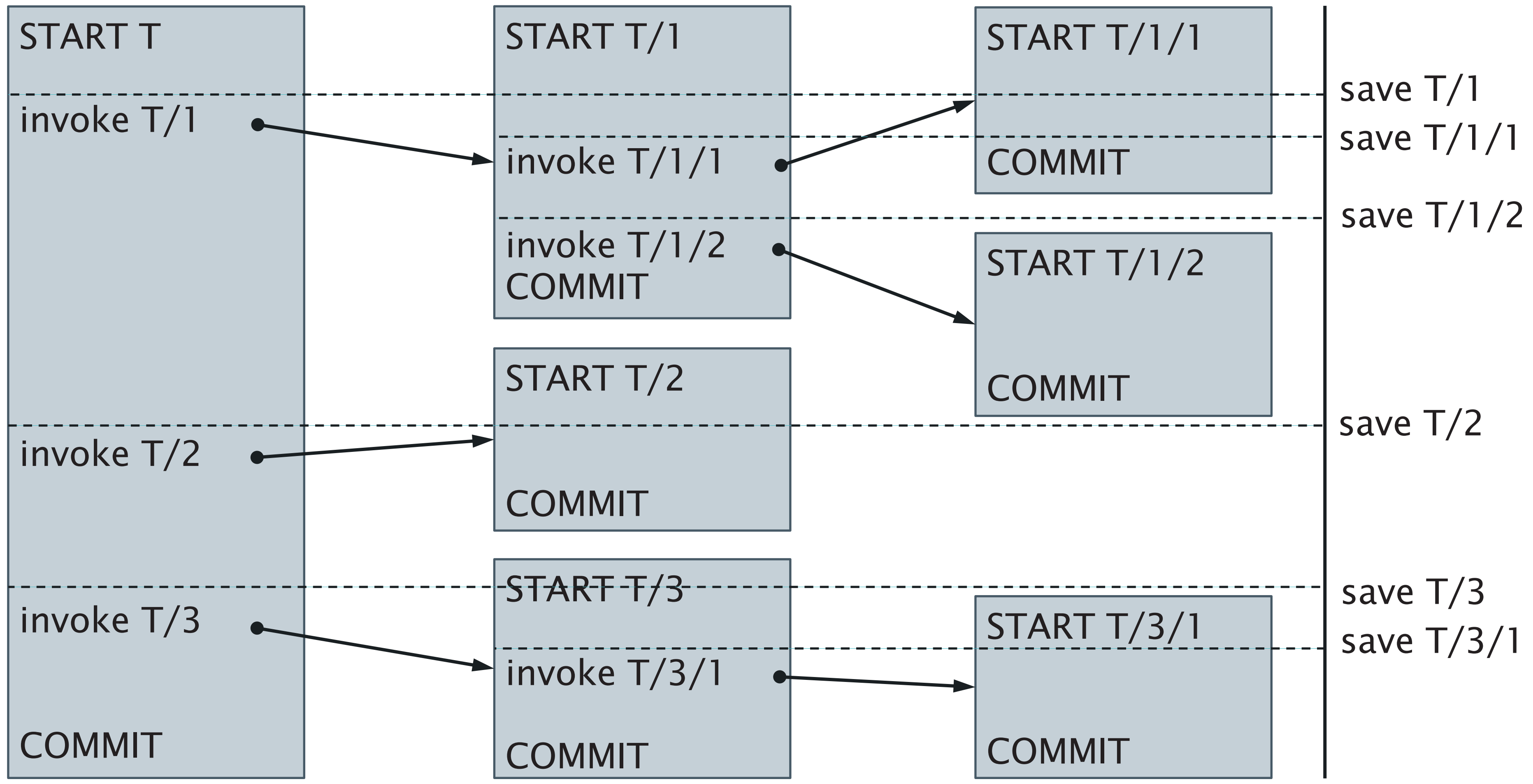


Emulating Nesting with Savepoints



UNIVERSITY OF Southampton

Emulating Nesting with Savepoints



Observations

Using savepoints is more flexible than nested transactions for internal recovery

- Can roll back further

True nested transactions are needed in order to run subtransactions in parallel (Intra-transaction parallelism)

- Emulating with savepoints needs 'subtransactions' to be run in strict sequence

True nested can pass locks selectively

- More flexible than savepoints
- “Similar but different”

Sagas

Sagas

Saga: a collection of actions (= flat transactions) that form a long-duration transaction

Execution based around notion of **compensating transactions**

- Inverse of actions that allow them to be selectively rolled back
- Used to recover from partial execution

Sagas

Sagas specified as a digraph

- Nodes are either actions or the terminal nodes **abort** and **complete**
- One node is designated the start

Paths in graph represent sequences of actions

- Paths leading to **abort** are sequences of actions that cause the overall transaction to be rolled back
- Paths leading to **complete** are successful sequences that make persistent changes to the database

Saga Execution

Each action A has a compensating transaction A^{-1}

Assume that if A is an action and α a sequence of legal actions, then $A\alpha A^{-1} \equiv \alpha$

If execution of a saga leads to **abort**, roll back the saga by executing the compensating transactions

Next Lecture: Logging and Recovery