University of
Southampton

University of Southampton

Transactions and Concurrency

COMP3211 Advanced Databases

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

University of Southampton

Overview

- Transaction processing
- Transaction problems
- Transaction lifecycle
- ACID
- Schedules and serialisability
- Locking (including 2PL)
- Timestamps

University of Southampton

Concurrency

- In a multi-user DBMS, many users may use the system concurrently
- Stored data items may be accessed concurrently by user programs

University of Southampton

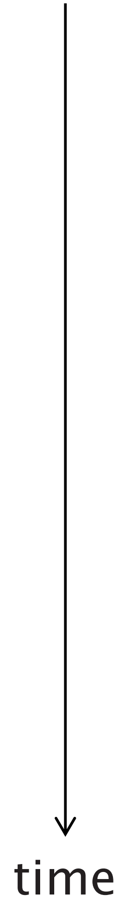
Concurrency

- In a multi-user DBMS, many users may use the system concurrently
- Stored data items may be accessed concurrently by user programs

Transaction: a logical unit of work that changes the contents of a database

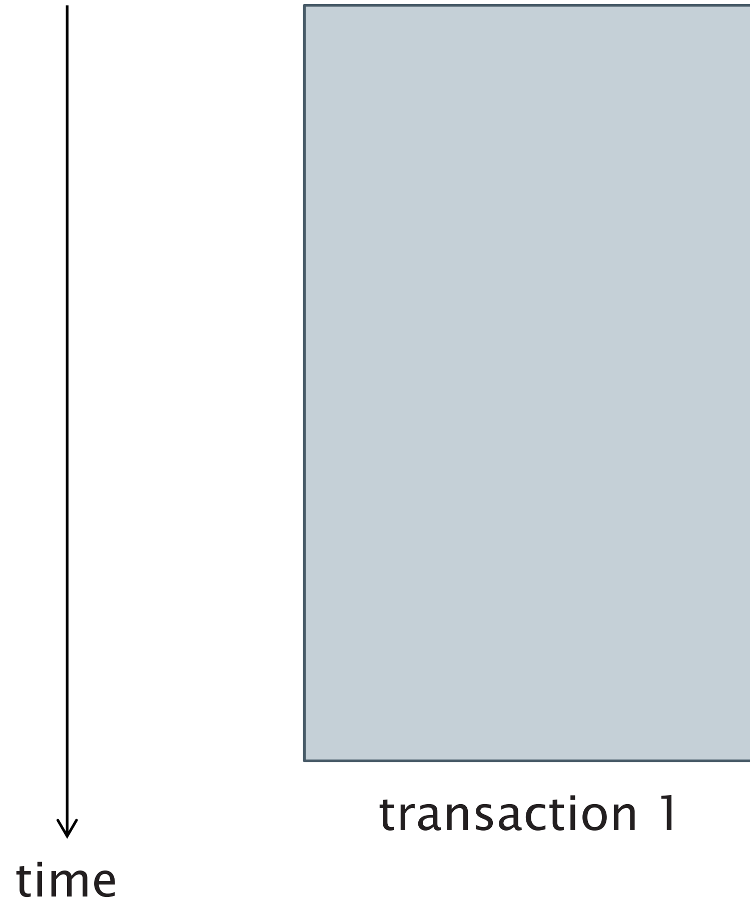
- Group of database operations that are to be executed together

When updates go wrong, part one

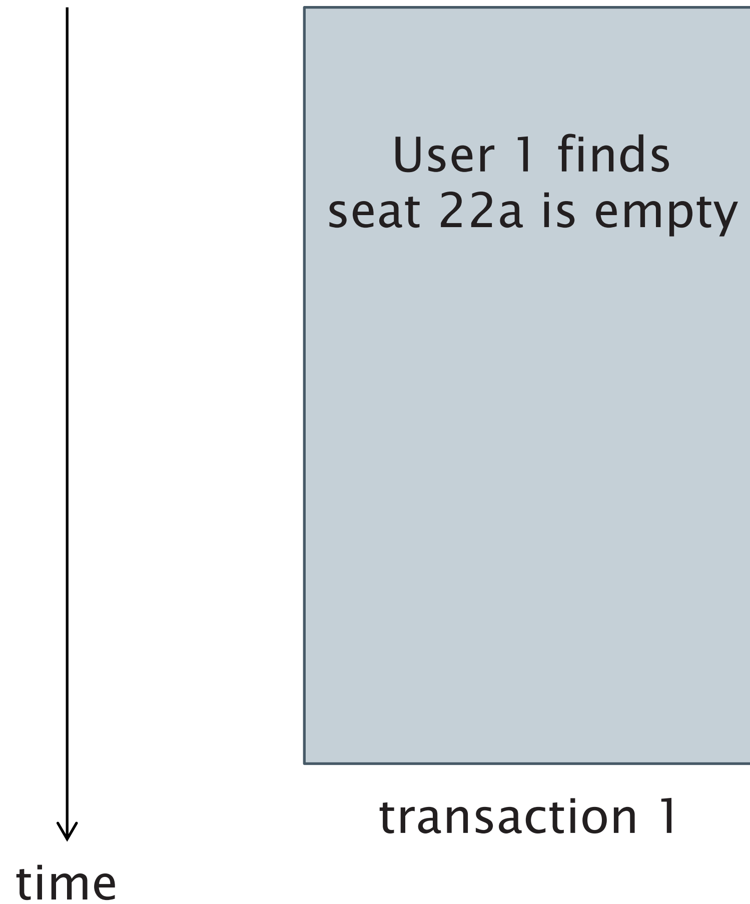


University of Southampton

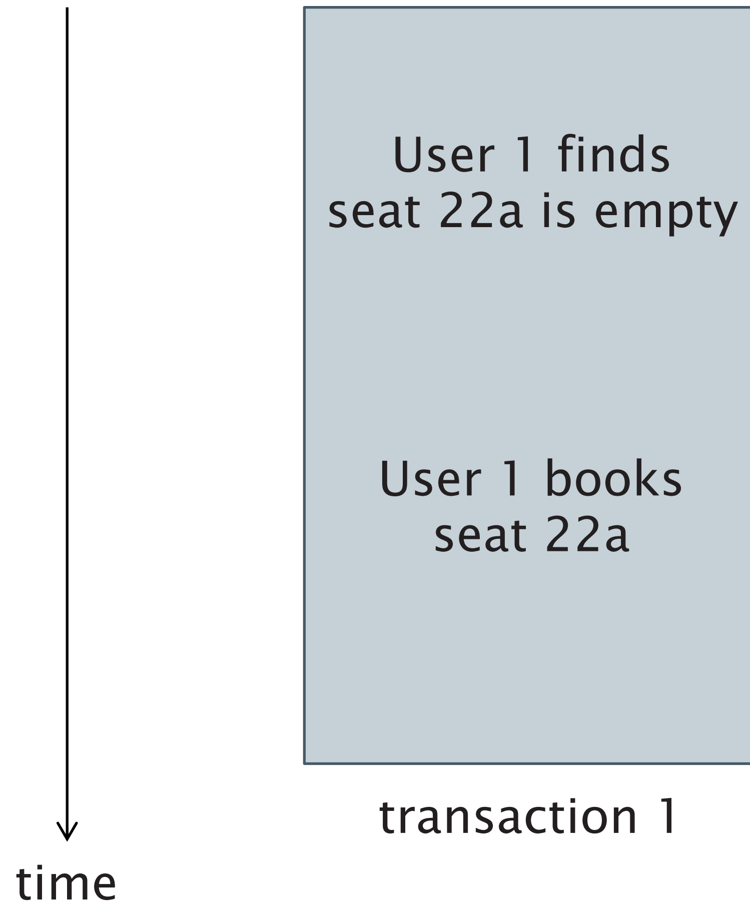
When updates go wrong, part one



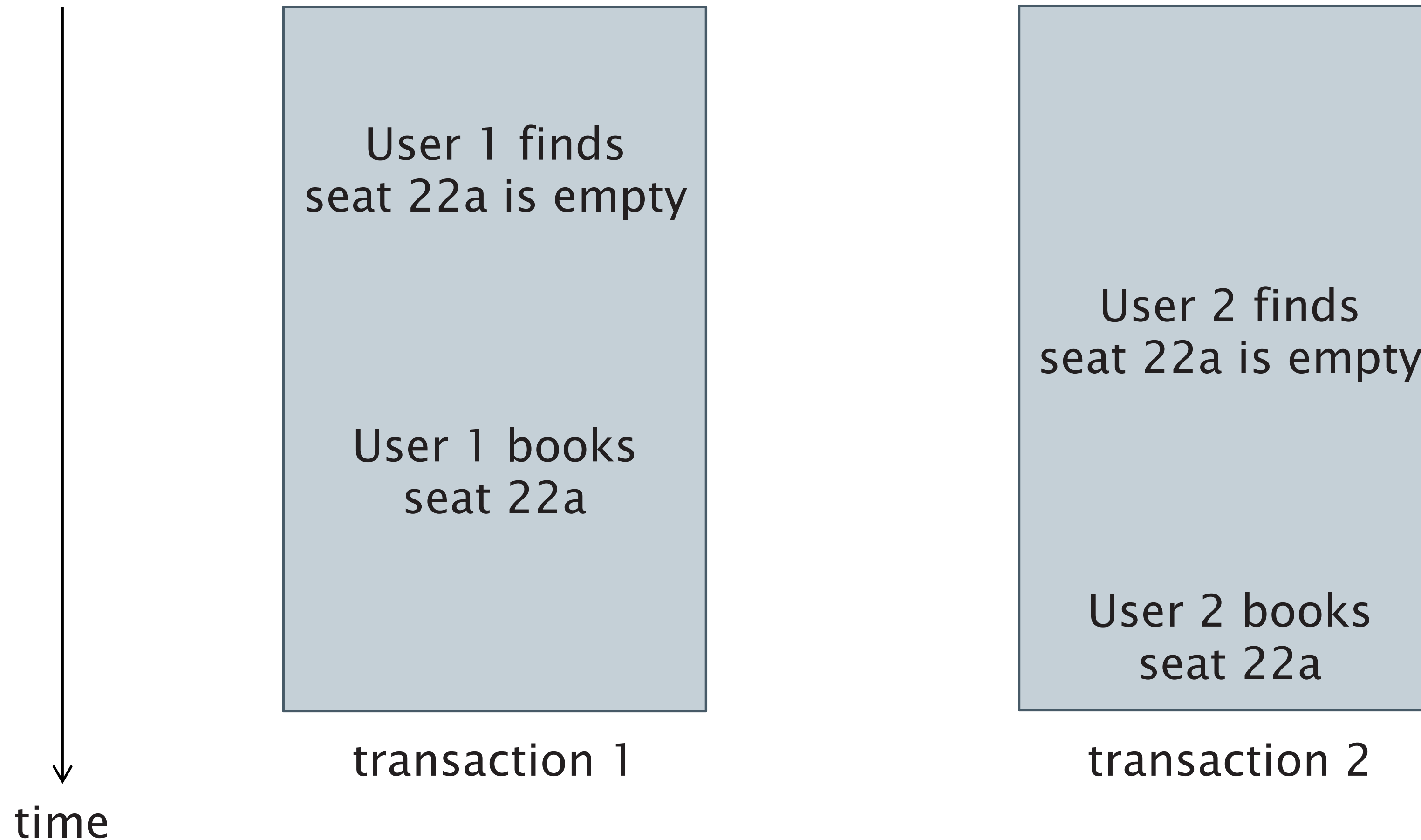
When updates go wrong, part one



When updates go wrong, part one



When updates go wrong, part one



Serial versus Serialisable

In an ideal world, we would run transactions **serially**

- Transactions runs one at a time, with no overlap

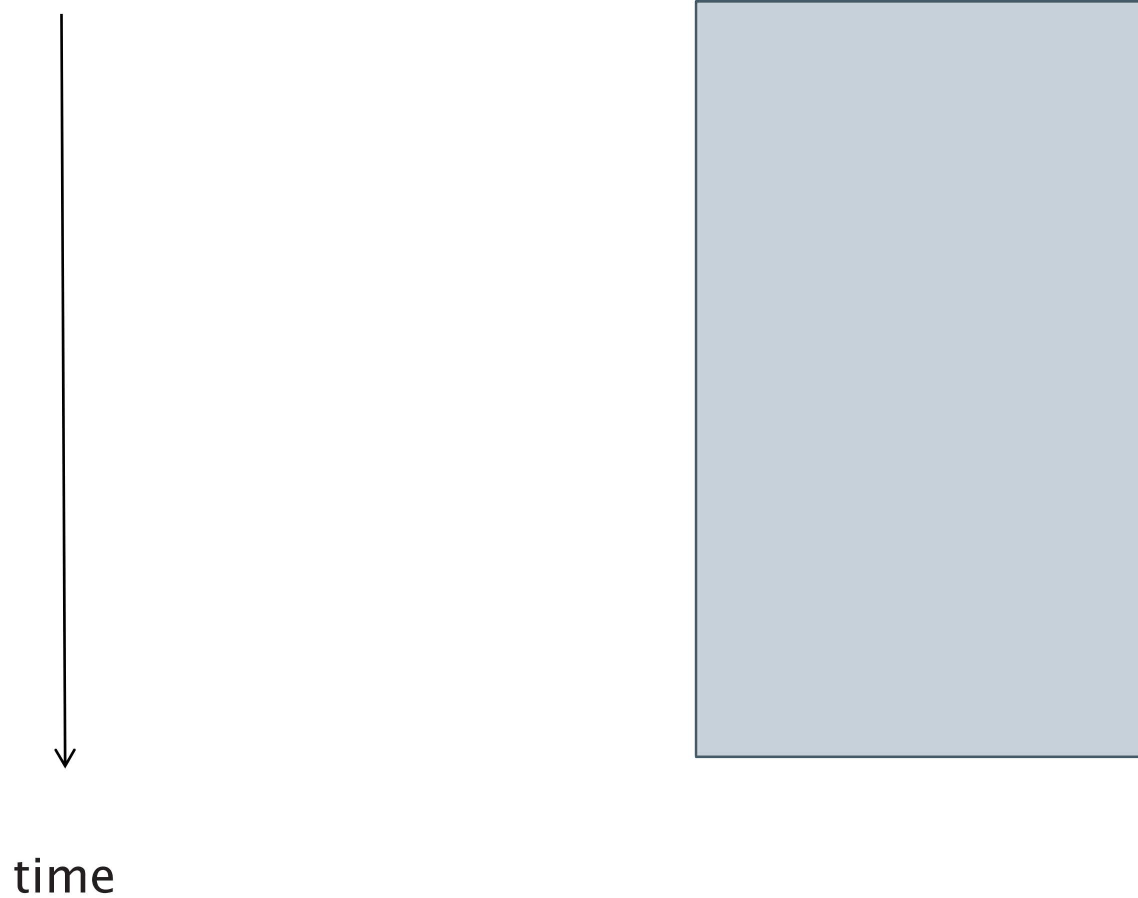
In practice, some parallelism is required

- Too many transactions for serial execution!

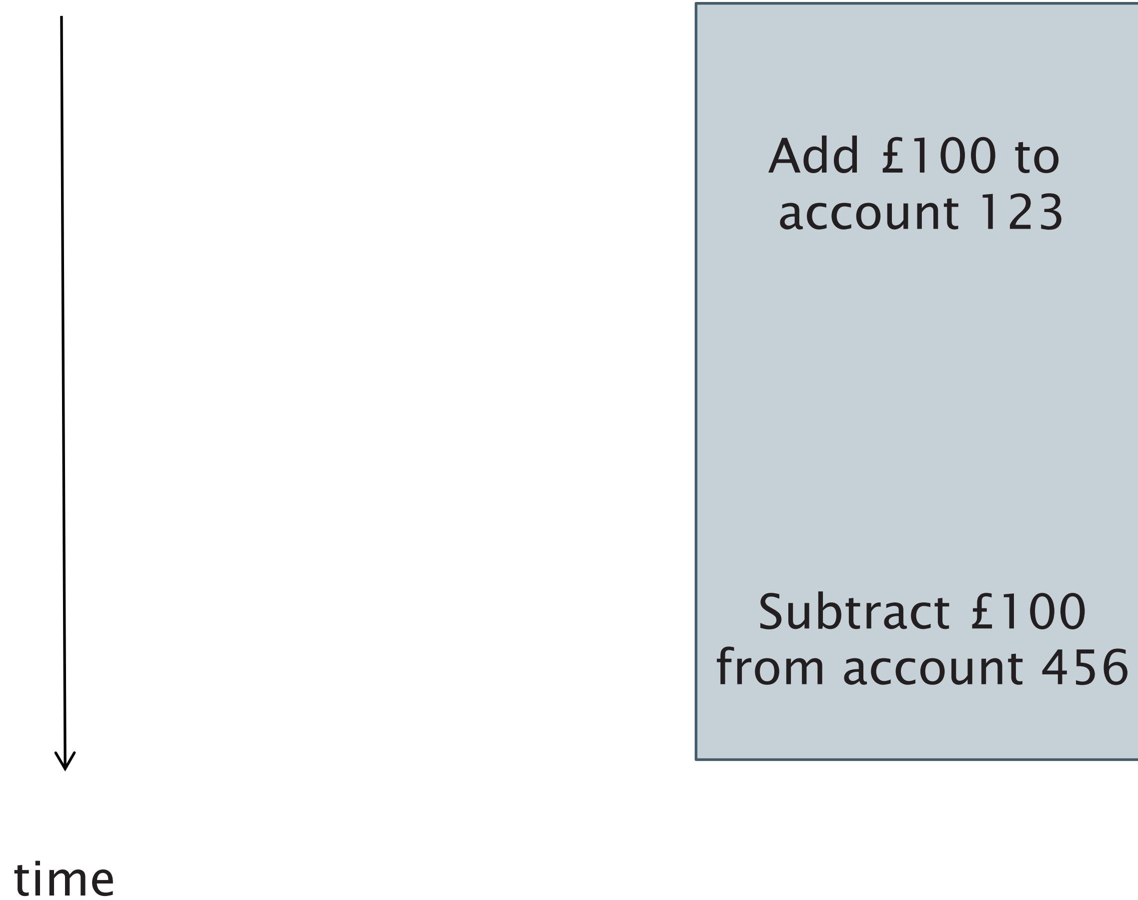
Transactions should be **serialisable**

- Should behave as if they were serial, but may be executed concurrently

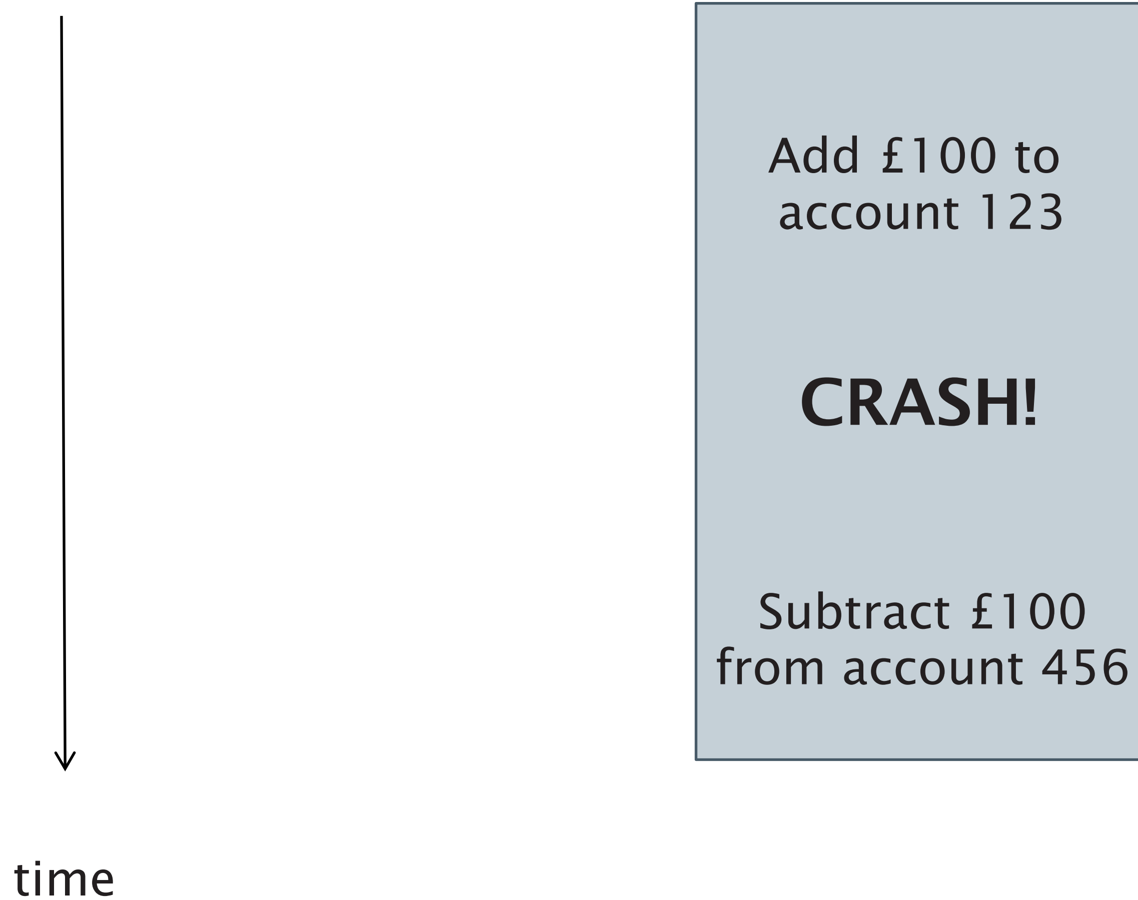
When updates go wrong, part two



When updates go wrong, part two



When updates go wrong, part two



University of Southampton

Atomicity

System failure partway through a transaction may leave the database in an inconsistent state

Transactions are **atomic**: operations within a transaction should either all be executed successfully or not be executed at all

Transaction Problems

University of Southampton

Basic database access operations

read(X)

Reads a database item X_d into a program variable X_T in transaction T

write(X)

Writes the value of program variable X_T in transaction T into the database item X_d

Example Transactions

T1

read(X)
X := X – 10
write(X)
read(Y)
Y := Y+10
write(Y)

T2

read(X)
X := X + 5
write(X)

Initial values: X=20, Y=50

Final values: X=15, Y=60

Concurrency

- Understanding transactions is important for concurrency
- Operations within a transaction may be interleaved with those from another transaction
- Depending on how operations are interleaved, database items may have incorrect values

University of Southampton

The Lost Update Problem

Two transactions have operations interleaved so that some DB items are incorrect

University of Southampton

T1	T2	X_{T1}	Y_{T1}	X_{T2}	Y_{T2}	X_d	Y_d
						20	50

T1	T2	X_{T1}	Y_{T1}	X_{T2}	Y_{T2}	X_d	Y_d
						20	50
read(X)		**20**				20	50

University of Southampton

T1	T2	X_{T1}	Y_{T1}	X_{T2}	Y_{T2}	X_d	Y_d
						20	50
read(X)		**20**				20	50
X := X - 10		**10**				20	50

University of Southampton

T1	T2	X_{T1}	Y_{T1}	X_{T2}	Y_{T2}	X_d	Y_d
						20	50
read(X)		**20**				20	50
X := X - 10		**10**				20	50
	read(X)	10		**20**		20	50

University of Southampton

T1	T2	X_{T1}	Y_{T1}	X_{T2}	Y_{T2}	X_d	Y_d
						20	50
read(X)		**20**				20	50
X := X - 10		**10**				20	50
	read(X)	10		**20**		20	50
	X := X + 5	10		**25**		20	50

University of Southampton

T1	T2	X_{T1}	Y_{T1}	X_{T2}	Y_{T2}	X_d	Y_d
						20	50
read(X)		**20**				20	50
X := X – 10		**10**				20	50
	read(X)	10		**20**		20	50
	X := X + 5	10		**25**		20	50
write(X)		10		25		**10**	50

T1	T2	X_{T1}	Y_{T1}	X_{T2}	Y_{T2}	X_d	Y_d
						20	50
read(X)		**20**				20	50
X := X - 10		**10**				20	50
	read(X)	10		**20**		20	50
	X := X + 5	10		**25**		20	50
write(X)		10		25		**10**	50
read(Y)		10	**50**	25		10	50

T1	T2	X_{T1}	Y_{T1}	X_{T2}	Y_{T2}	X_d	Y_d
						20	50
read(X)		**20**				20	50
X := X – 10		**10**				20	50
	read(X)	10		**20**		20	50
	X := X + 5	10		**25**		20	50
write(X)		10		25		**10**	50
read(Y)		10	**50**	25		10	50
	write(X)	10	50	25		**25**	50

University of Southampton

T1	T2	X_{T1}	Y_{T1}	X_{T2}	Y_{T2}	X_d	Y_d
						20	50
read(X)		**20**				20	50
X := X - 10		**10**				20	50
	read(X)	10		**20**		20	50
	X := X + 5	10		**25**		20	50
write(X)		10		25		**10**	50
read(Y)		10	**50**	25		10	50
	write(X)	10	50	25		**25**	50
Y := Y+10		10	**60**	25		25	50

University of Southampton

T1	T2	X_{T1}	Y_{T1}	X_{T2}	Y_{T2}	X_d	Y_d
						20	50
read(X)		**20**				20	50
X := X – 10		**10**				20	50
	read(X)	10		**20**		20	50
	X := X + 5	10		**25**		20	50
write(X)		10		25		**10**	50
read(Y)		10	**50**	25		10	50
	write(X)	10	50	25		**25**	50
Y := Y+10		10	**60**	25		25	50
write(Y)		10	60	25		25	**60**

University of Southampton

The Temporary Update (Dirty Read) Problem

One transaction updates a DB item and then fails. Item is accessed before reverting to original value.

T1	T2	X_{T1}	Y_{T1}	X_{T2}	Y_{T2}	X_d	Y_d
						20	50
read(X)		**20**				20	50
X := X – 10		**10**				20	50
write(X)		10				**10**	50
	read(X)	10		**10**		10	50
	X := X + 5	10		**15**		10	50
	write(X)	10		15		**15**	50
read(Y)		10	**50**	15		15	50
CRASH!							
rollback						**20**	50

The Incorrect Summary Problem

One transaction calculates an aggregate summary function on multiple records while other transactions update records

Aggregate function may read some values before they are updated, and some after

T1	T2	X_{T1}	Y_{T1}	S	X_{T2}	Y_{T2}	X_d	Y_d
							20	50
S := 0				**0**			20	50
	read(X)			0	**20**		20	50
	X := X – 10			0	**10**		20	50
	write(X)			0	10		**10**	50
read(X)		**10**		0	10		10	50
S := S + X		10		**10**	10		10	50
read(Y)		10	**50**	10	10		10	50
S := S + Y		10	50	**60**	10		10	50
	read(Y)	10	50	60	10	**50**	10	50
	Y := Y + 10	10	50	60	10	**60**	10	50
	write(Y)	10	50	60	10	60	10	**60**

University of Southampton

The Unrepeatable Read Problem

One transaction reads an item twice, while another changes the item between the two reads

T1:	**T2:**
read(X)	
	read(X)
	X := X – 10
	write(X)
read(X)	

Transaction Processing

When a transaction is submitted for execution, the system must ensure that:

- All operations in the transaction are completed successfully, with effect recorded permanently in the database, or
- There is no effect on the database or other transactions

Transactions may be **read-only** or **update**

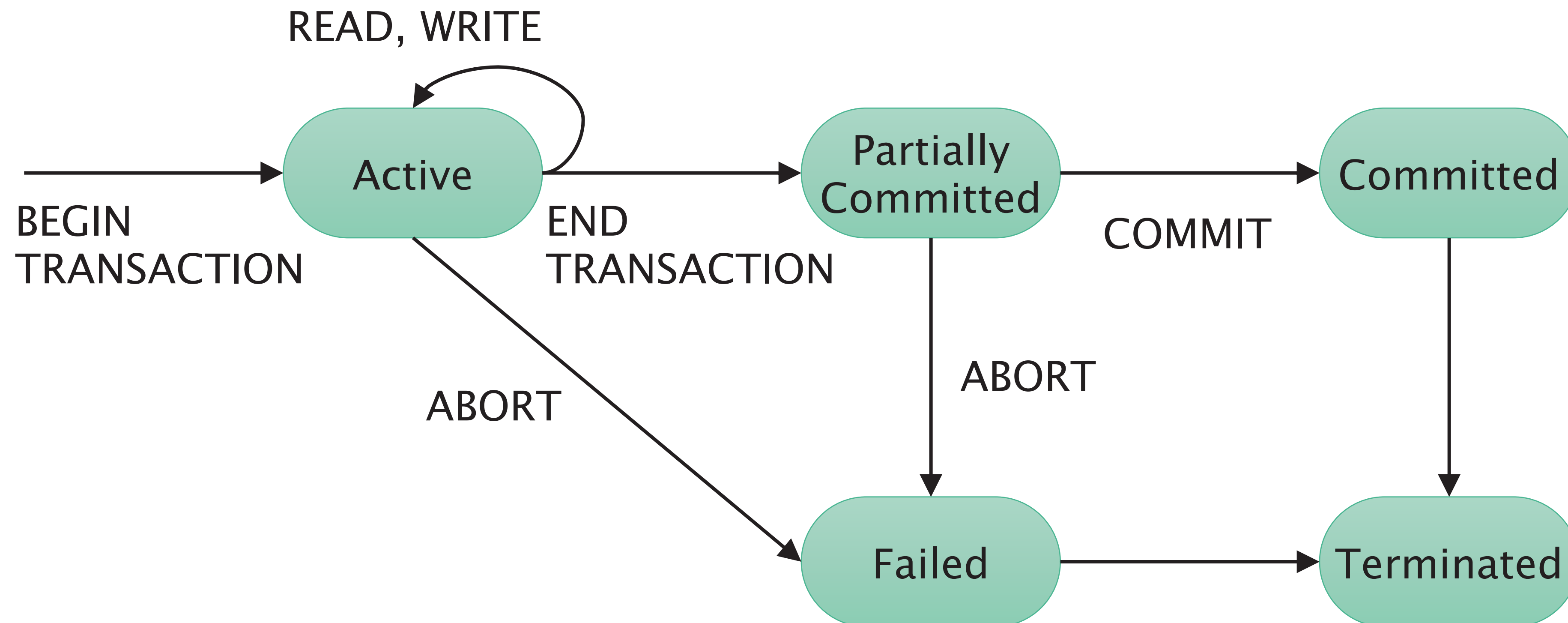
University of Southampton

Transaction Life Cycle

Need to track start and end of transactions, and commit and abort of transactions

- BEGIN_TRANSACTION
- READ, WRITE
- END_TRANSACTION
- COMMIT_TRANSACTION
- ROLLBACK (or ABORT)

Transaction Life Cycle



ACID

ACID Properties

Atomicity

A transaction is either performed completely or not at all

Consistency

Correct transaction execution must take the database from one consistent state to another

Isolation

A transaction should not make updates externally visible (to other transactions) until committed

Durability

Once database is changed and committed, changes should not be lost because of failure

University of Southampton

Schedules

A **schedule** S of n transactions is an ordering of the operations of the transactions, subject to the constraint that for each transaction T that participates in S, the operations in T must appear in the same order in S that they do in T

Two operations in a schedule are **conflicting** if:

- They belong to different transactions and
- They access the same data item and
- At least one of the operations is a write()

Serial and Serialisable

A schedule is **serial** if, for each transaction T in the schedule, all operations in T are executed consecutively (no interleaving), otherwise it is **non-serial**

A schedule S of n transactions is **serialisable** if it is equivalent to some serial schedule of the same n transactions

Schedule Equivalence

Two schedules are **result equivalent** if they produce the same final state on the database

Two schedules are **conflict equivalent** if the order of any two conflicting operations is the same in both schedules

Serial Schedule T1;T2

T1	T2	X_{T1}	Y_{T1}	X_{T2}	Y_{T2}	X_d	Y_d
						20	50
read(X)		**20**				20	50
X := X – 10		**10**				20	50
write(X)		10				**10**	50
read(Y)		10	**50**			10	50
Y := Y + 10		10	**60**			10	50
write(Y)		10	60			10	**60**
	read(X)	10	60	**10**		10	60
	X := X + 5	10	60	**15**		10	60
	write(X)	10	60	15		**15**	60

Serial Schedule T2;T1

T1	T2	X_{T1}	Y_{T1}	X_{T2}	Y_{T2}	X_d	Y_d
						20	50
	read(X)			**20**		20	50
	X := X + 5			**25**		20	50
	write(X)			25		**25**	50
read(X)		**25**		25		25	50
X := X – 10		**15**		25		25	50
write(X)		15		25		**15**	50
read(Y)		15	**50**	25		15	50
Y := Y + 10		15	**60**	25		15	50
write(Y)		15	60	25		15	**60**

Non-Serial and Non-Serialisable Schedule

T1	T2	X_{T1}	Y_{T1}	X_{T2}	Y_{T2}	X_d	Y_d
						20	50
read(X)		**20**				20	50
X := X – 10		**10**				20	50
	read(X)	10		**20**		20	50
	X := X + 5	10		**25**		20	50
write(X)		10		25		**10**	50
read(Y)		10	**50**	25		10	50
	write(X)	10	50	25		**25**	50
Y := Y+10		10	**60**	25		25	50
write(Y)		10	60	25		25	**60**

Non-Serial but Serialisable Schedule

T1	T2	X_{T1}	Y_{T1}	X_{T2}	Y_{T2}	X_d	Y_d
						20	50
read(X)		**20**				20	50
X := X – 10		**10**				20	50
write(X)		10				**10**	50
	read(X)	10		**10**		10	50
	X := X + 5	10		**15**		10	50
	write(X)	10		15		**15**	50
read(Y)		10	**50**	15		15	50
Y := Y + 10		10	**60**	15		15	50
write(Y)		10	60	15		15	**60**

Locking

University of Southampton

University of Southampton

Locking

Locks are used to synchronise access by concurrent transactions to a database

Typically, two lock modes: **shared** and **exclusive**

- Shared: for reading
- Exclusive: for writing

Binary locks (equivalent to exclusive mode only) are also possible, but generally too restrictive

University of Southampton

Lock Operations

lock-shared(X)

Attempt to acquire a shared lock on X

lock-exclusive(X)

Attempt to acquire an exclusive lock on X

unlock(X)

Relinquish all locks on X

Lock Outcome

The result of an attempt to obtain a lock is either:

- Grant lock (able to access the item)
- Wait for lock to be granted (not yet able to access the item)
- (Abort)

		Lock Requested	
		Shared	**Exclusive**
Lock held in mode	**Shared**	Grant	Wait
	Exclusive	Wait	Wait

University of Southampton

Locking Rules

1. Must issue lock-shared(X) or lock-exclusive(X) before a read(X) operation
2. Must issue lock-exclusive(X) before a write(X) operation
3. Must issue unlock(X) after all read(X) and write(X) operations are completed
4. Cannot issue lock-shared(X) if already holding a lock on X
5. Cannot issue lock-exclusive(X) if already holding a lock on X
6. Cannot issue unlock(X) unless holding a lock on X

Lock Conversion

Rules 4 and 5 may be relaxed in order to allow lock conversion

- A lock-shared(X) may be *upgraded* to a lock-exclusive(X)
- A lock-exclusive(X) may be *downgraded* to a lock-shared(X)

Locking Example

T1:

lock-shared(Y)
read(Y)
unlock(Y)
lock-exclusive(X)
read(X)
X := X + Y
write(X)
unlock(X)

T2:

lock-shared(X)
read(X)
unlock(X)
lock-exclusive(Y)
read(Y)
Y := Y + X
write(Y)
unlock(Y)

University of Southampton

Locking Example

Two possible serial schedules:

- T1;T2
- T2;T1

Take X=20 and Y=50 as initial values

University of Southampton

T1	T2	X_{T1}	Y_{T1}	X_{T2}	Y_{T2}	X_d	Y_d
						20	50
lock-shared(Y)						20	50
read(Y)			**50**			20	50
unlock(Y)			50			20	50
lock-exclusive(X)			50			20	50
read(X)		**20**	50			20	50
X := X + Y		**70**	50			20	50
write(X)		70	50			**70**	50
unlock(X)		70	50			70	50
	lock-shared(X)	70	50			70	50
	read(X)	70	50	**70**		70	50
	unlock(X)	70	50	70		70	50
	lock-exclusive(Y)	70	50	70		70	50
	read(Y)	70	50	70	**50**	70	50
	Y := Y + X	70	50	70	**120**	70	50
	write(Y)	70	50	70	120	70	**120**
	unlock(Y)	70	50	20	120	70	120

T1	T2	X_{T1}	Y_{T1}	X_{T2}	Y_{T2}	X_d	Y_d
						20	50
	lock-shared(X)					20	50
	read(X)			**20**		20	50
	unlock(X)			20		20	50
	lock-exclusive(Y)			20		20	50
	read(Y)			20	**50**	20	50
	Y := Y + X			20	**70**	20	50
	write(Y)			20	70	20	**70**
	unlock(Y)			20	70	20	70
lock-shared(Y)				20	70	20	70
read(Y)			**70**	20	70	20	70
unlock(Y)			70	20	70	20	70
lock-exclusive(X)			70	20	70	20	70
read(X)		**20**	70	20	70	20	70
X := X + Y		**90**	70	20	70	20	70
write(X)		90	70	20	70	**90**	70
unlock(X)		90	70	20	70	90	70

University of Southampton

Serial Schedules

After T1;T2, we have: X=70, Y=120

After T2;T1, we have: X=90, Y=70

What about a non-serial schedule?

T1	T2	X_{T1}	Y_{T1}	X_{T2}	Y_{T2}	X_d	Y_d
						20	50
lock-shared(Y)						20	50
read(Y)			**50**			20	50
unlock(Y)			50			20	50
	lock-shared(X)		50			20	50
	read(X)		50	**20**		20	50
	unlock(X)		50	20		20	50
	lock-exclusive(Y)		50	20		20	50
	read(Y)		50	20	**50**	20	50
	Y := Y + X		50	20	**70**	20	50
	write(Y)		50	20	70	20	**70**
	unlock(Y)		50	20	70	20	70
lock-exclusive(X)			50	20	70	20	70
read(X)		**20**	50	20	70	20	70
X := X + Y		**70**	50	20	70	20	70
write(X)		70	50	20	70	**70**	70
unlock(X)		70	50	20	70	70	70

University of Southampton

Locking Example

After schedule, we have: X=70, Y=70

- The schedule is not serialisable
  (not result equivalent to either of the serial schedules)

- Locking, by itself, isn’t enough

Two-Phase Locking (2PL)

University of Southampton

Locking and Serialisability

Using locks doesn't guarantee serialisability by itself

Extra rules for handling locks:

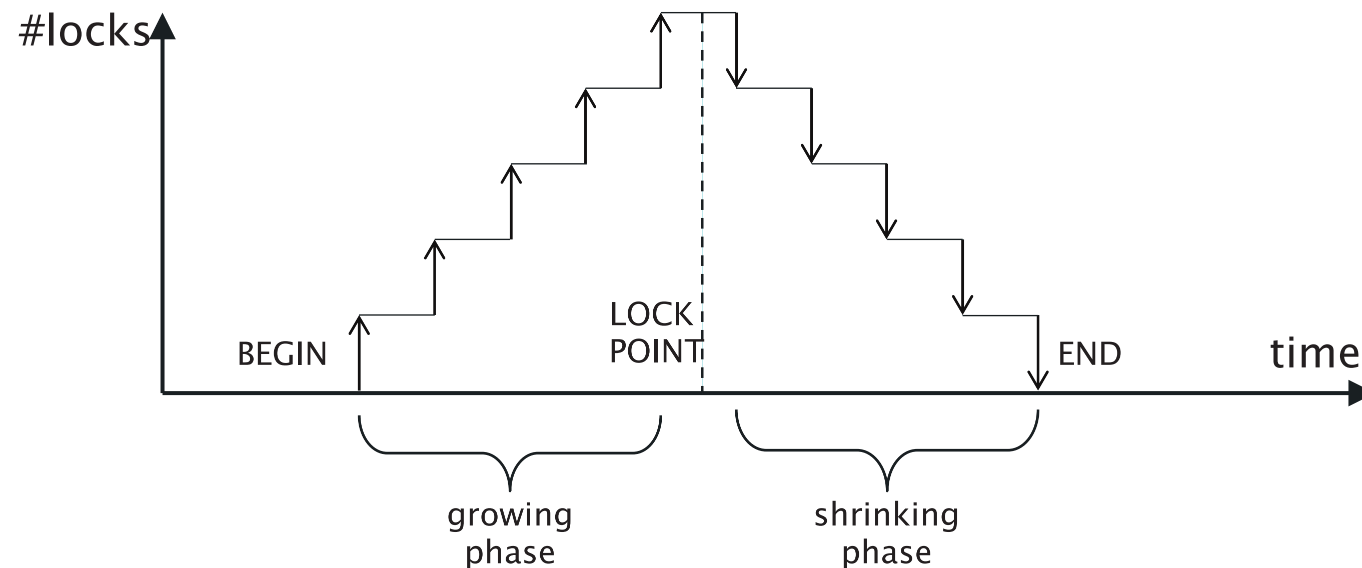
- All locking operations precede the first unlock operation in a transaction
- Locks are only released after a transaction commits or aborts

Two-Phase Locking

Two phases:

- Growing phase: obtain locks, access data items
- Shrinking phase: release locks

Guarantees serialisable transactions



University of Southampton

Two-Phase Locking Example

T1:

lock-shared(Y)

read(Y)

lock-exclusive(X)

unlock(Y)

read(X)

X := X + Y

write(X)

unlock(X)

T2:

lock-shared(X)

read(X)

lock-exclusive(Y)

unlock(X)

read(Y)

Y := X + Y

write(Y)

unlock(Y)

Deadlock

University of Southampton

When 2PL goes wrong

Consider the following schedule of T1 and T2

T1:	**T2:**
lock-shared(Y)	
read(Y)	
	lock-shared(X)
	read(X)
lock-exclusive(X)	
unlock(Y)	
	lock-exclusive(Y)
	unlock(X)
…	…

When 2PL goes wrong

Consider the following schedule of T1 and T2

T1:	**T2:**
lock-shared(Y)	
read(Y)	
	lock-shared(X)
	read(X)
lock-exclusive(X)	
unlock(Y)	
	lock-exclusive(Y)
	unlock(X)
…	…

T1 can't get an exclusive lock on X; T2 already has a shared lock on X

When 2PL goes wrong

Consider the following schedule of T1 and T2

T1:	**T2:**
lock-shared(Y)	
read(Y)	
	lock-shared(X)
	read(X)
lock-exclusive(X)	
unlock(Y)	
	lock-exclusive(Y)
	unlock(X)
...	...

T1 can't get an exclusive lock on X; T2 already has a shared lock on X

T2 can't get an exclusive lock on Y; T1 already has a shared lock on Y

University of Southampton

Deadlock

Deadlock exists when two or more transactions are waiting for each other to release a lock on an item

Several conditions must be satisfied for deadlock to occur

- Concurrency: two processes claim exclusive control of one resource
- Hold: one process continues to hold exclusively controlled resources until its need is satisfied
- Wait: processes wait in queues for additional resources while holding resource already allocated
- Mutual dependency

Deadlock

- Final condition for deadlock is that some mutual dependency must exist
- Breaking deadlock requires that one transaction is aborted

Processes	Resource List	Wait List
A	1, 10	8
B	3, 4, 15	10
C	2, 0	
D	6, 8	15

Dealing with Deadlock

Deadlock prevention

- Every transaction locks all items it needs in advance; if an item cannot be obtained, no items are locked
- Transactions updating the same resources are not allowed to execute concurrently

Deadlock detection - detect and reverse one transaction

- Wait-for graph
- Timeouts

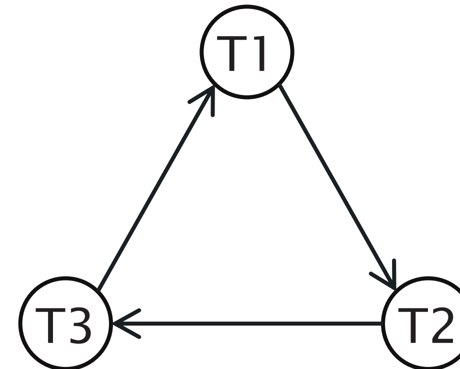
Wait-For Graph

Representation of interactions between transactions

Directed graph containing:

- A vertex for each transaction that is currently executing
- An edge from T1 to T2 if T1 is waiting to lock an item that is currently locked by T2

Deadlock exists iff the WFG contains a cycle



University of Southampton

Timeouts

If a transaction waits for a resource for longer than a given period (the timeout), the system assumes that the transaction is deadlocked and aborts it

Granularity and Concurrency

Granularity of Data Items

What should be locked?

- Record
- Field value of record
- Disc block
- File
- Database

Coarser granularity gives lower degree of concurrency

Finer granularity gives higher overhead

Timestamps

University of Southampton

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 accessed 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)

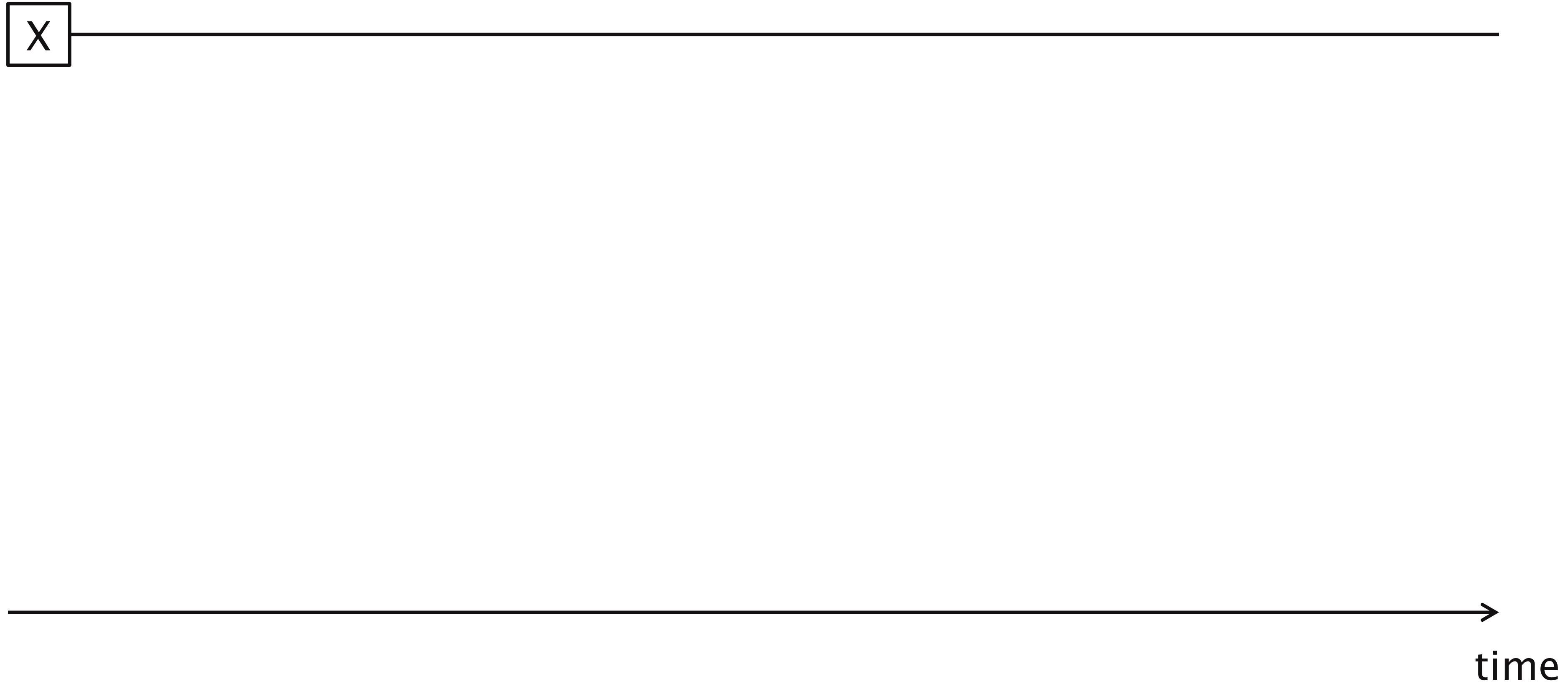
- Has this item been read or written before transaction T has had an opportunity to read/write?
- Ensure that timestamp ordering is not violated

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

Basic Timestamp Ordering: write(X)

```
if TS(T) ≥ read-TS(X) and TS(T) ≥ write-TS(X)
then
      execute write(X)
      set write-TS(X) to TS(T)
else
      abort and rollback T
```

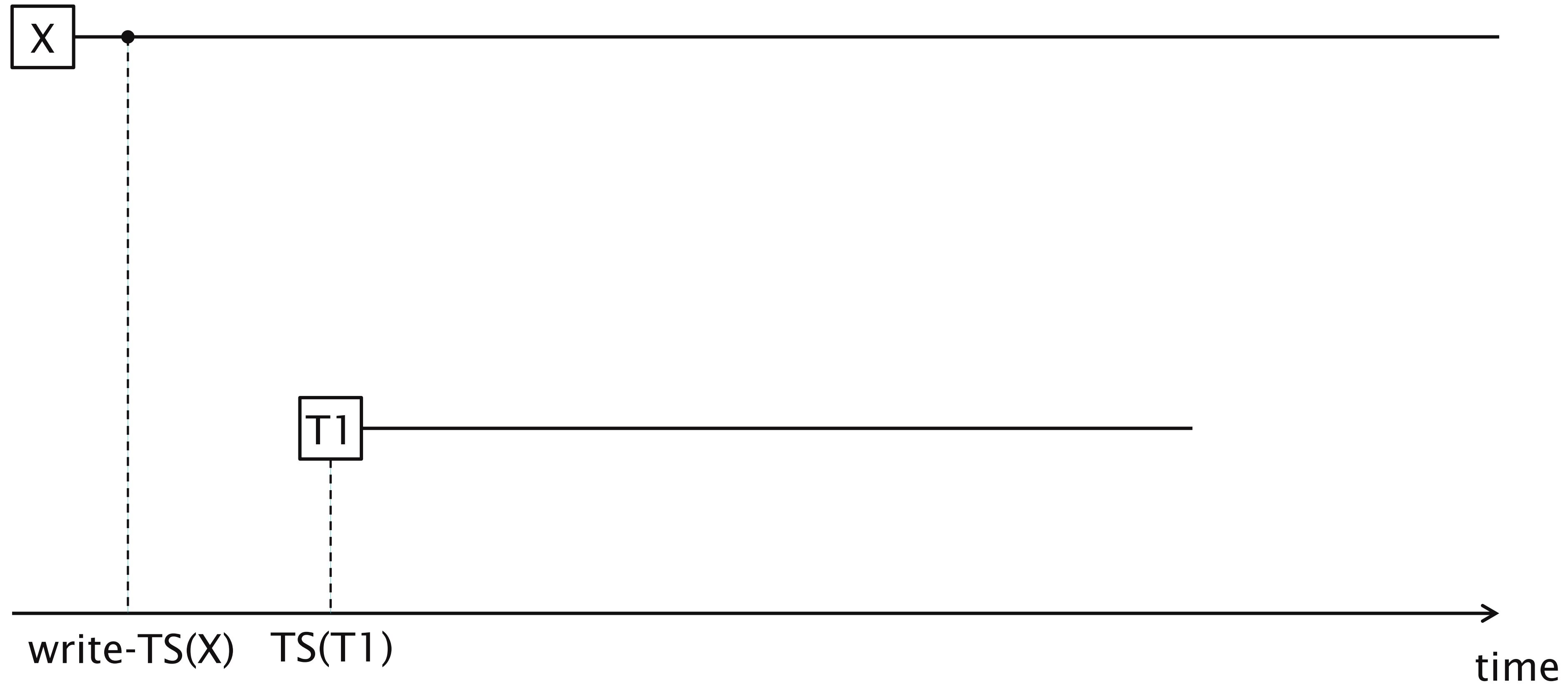
Basic Timestamp Ordering



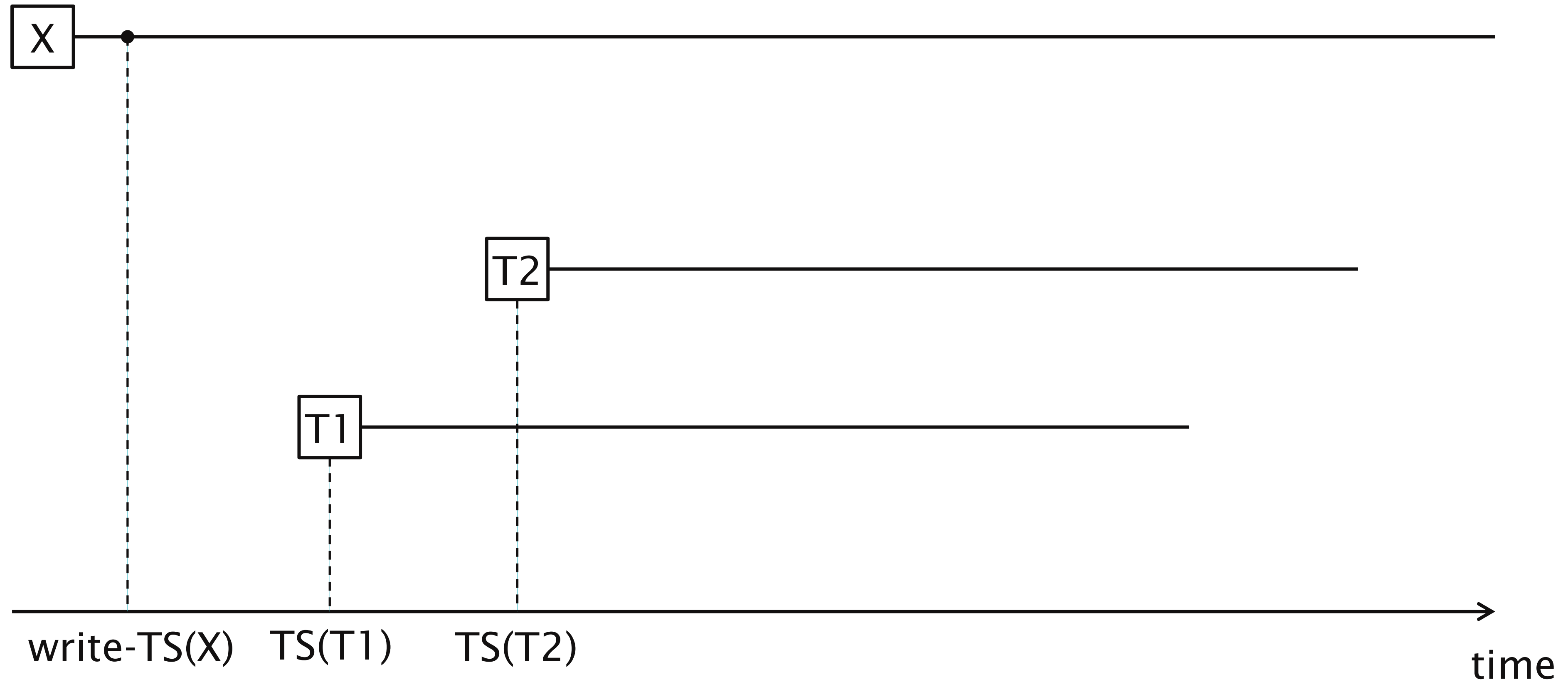
University of Southampton

Basic Timestamp Ordering

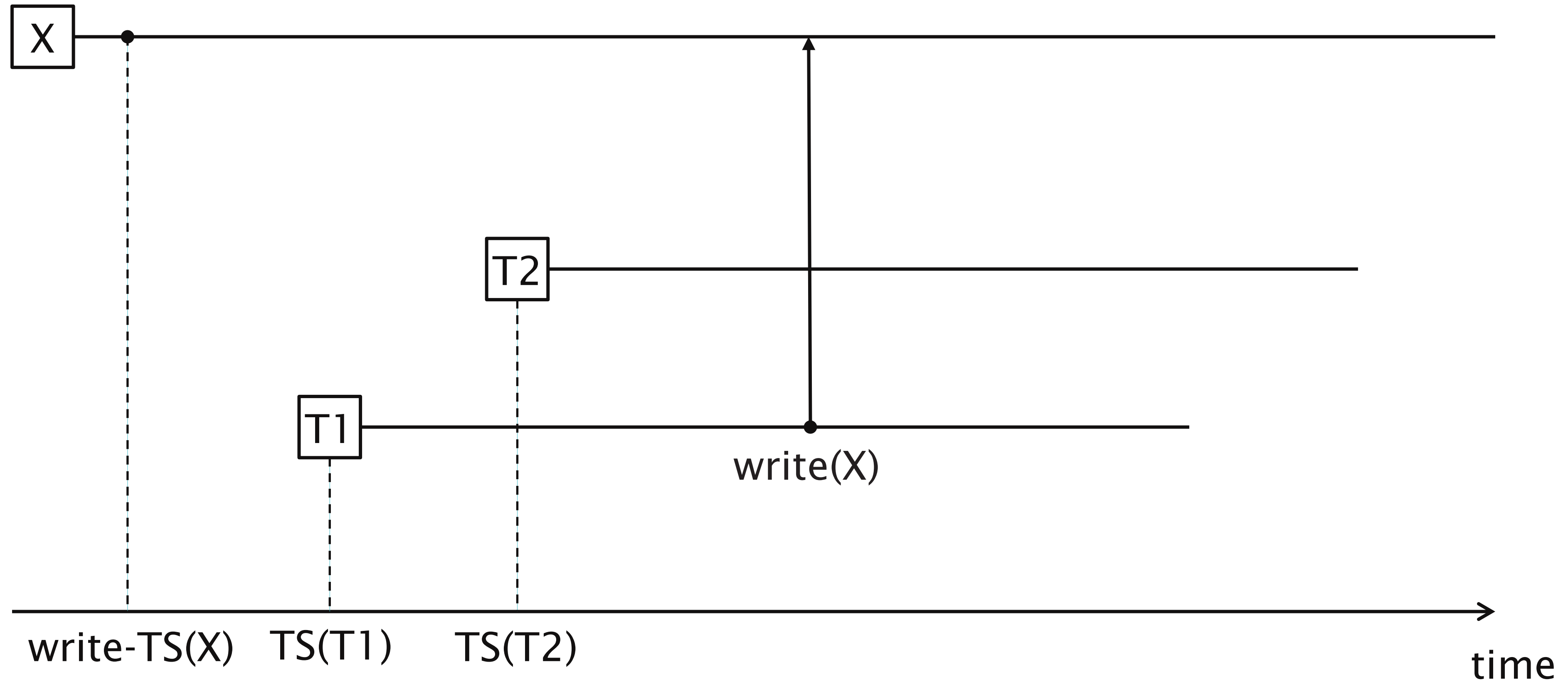
Basic Timestamp Ordering



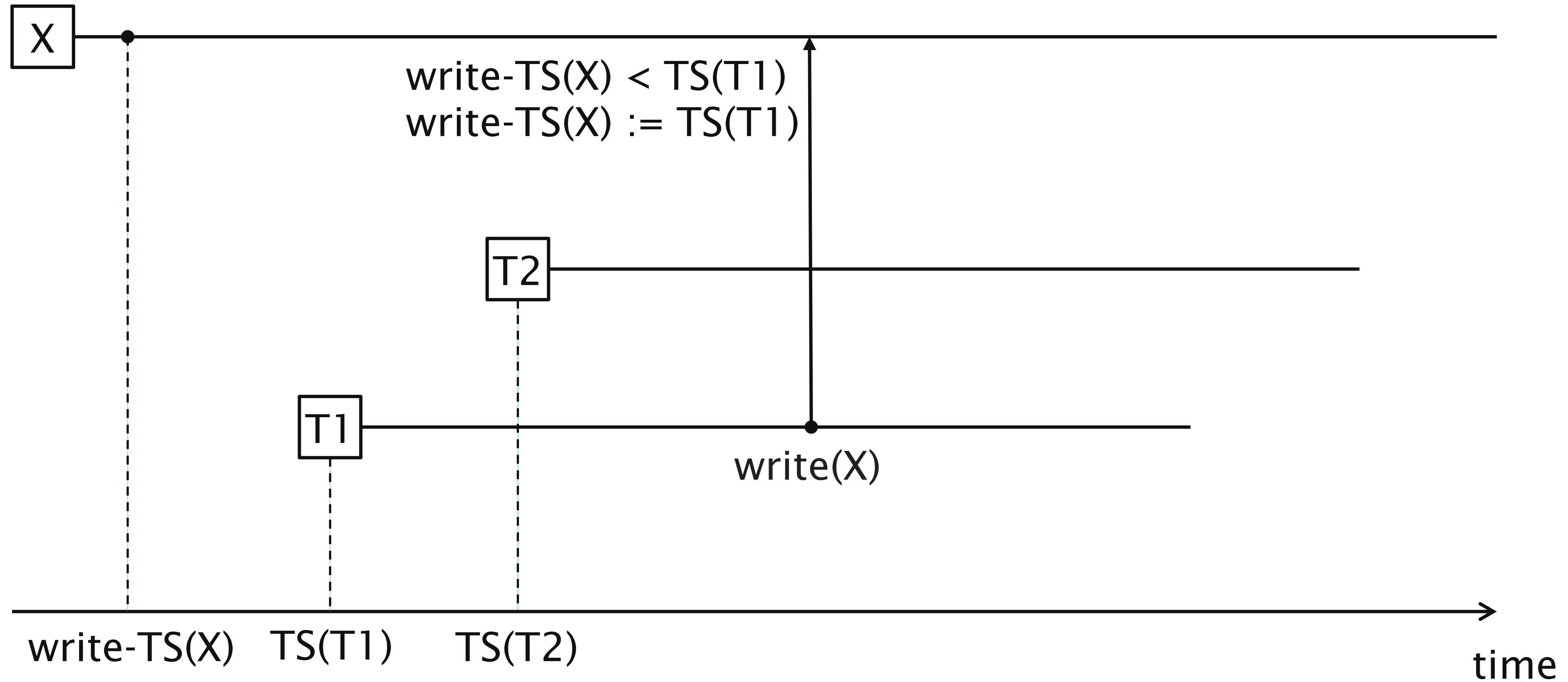
Basic Timestamp Ordering



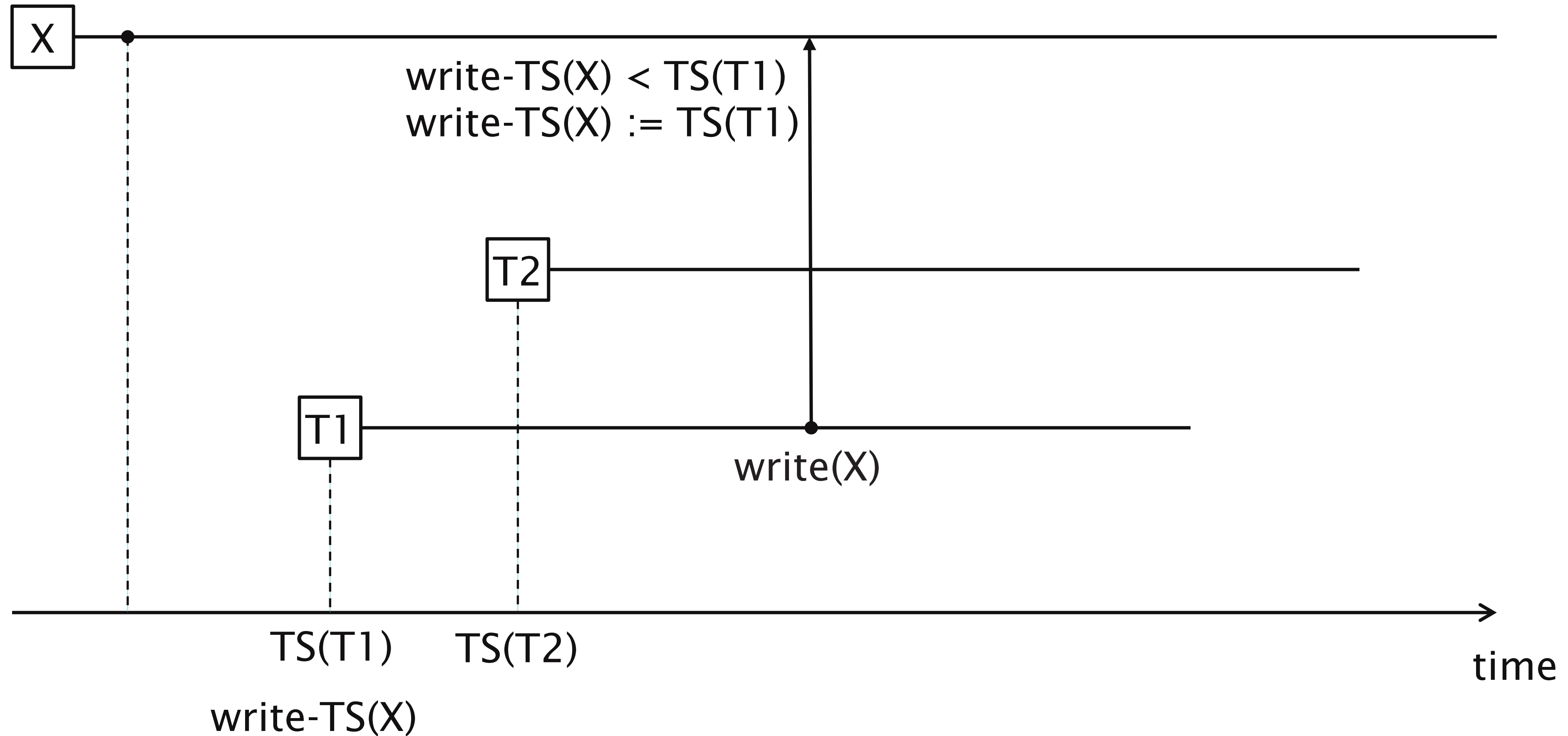
Basic Timestamp Ordering



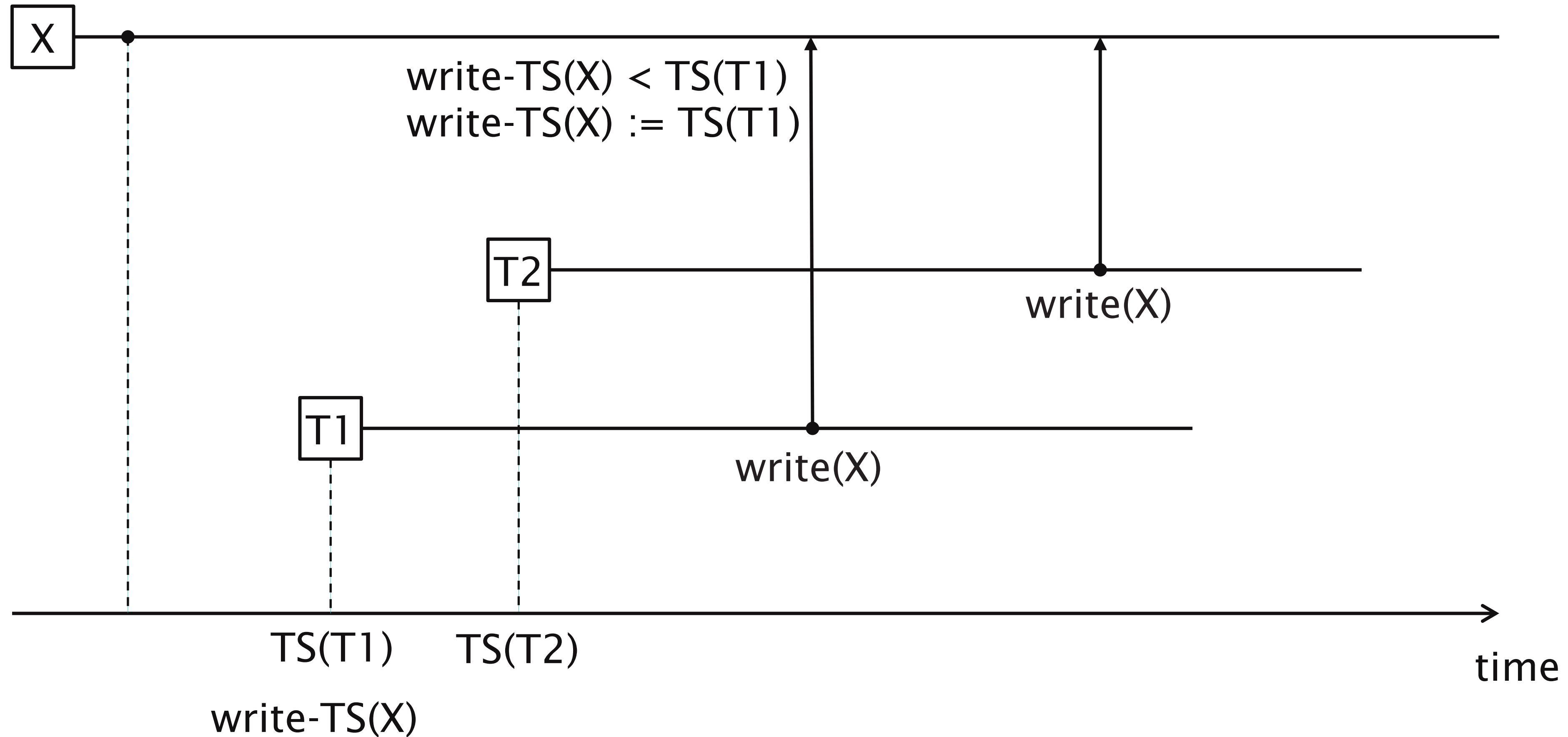
Basic Timestamp Ordering



Basic Timestamp Ordering

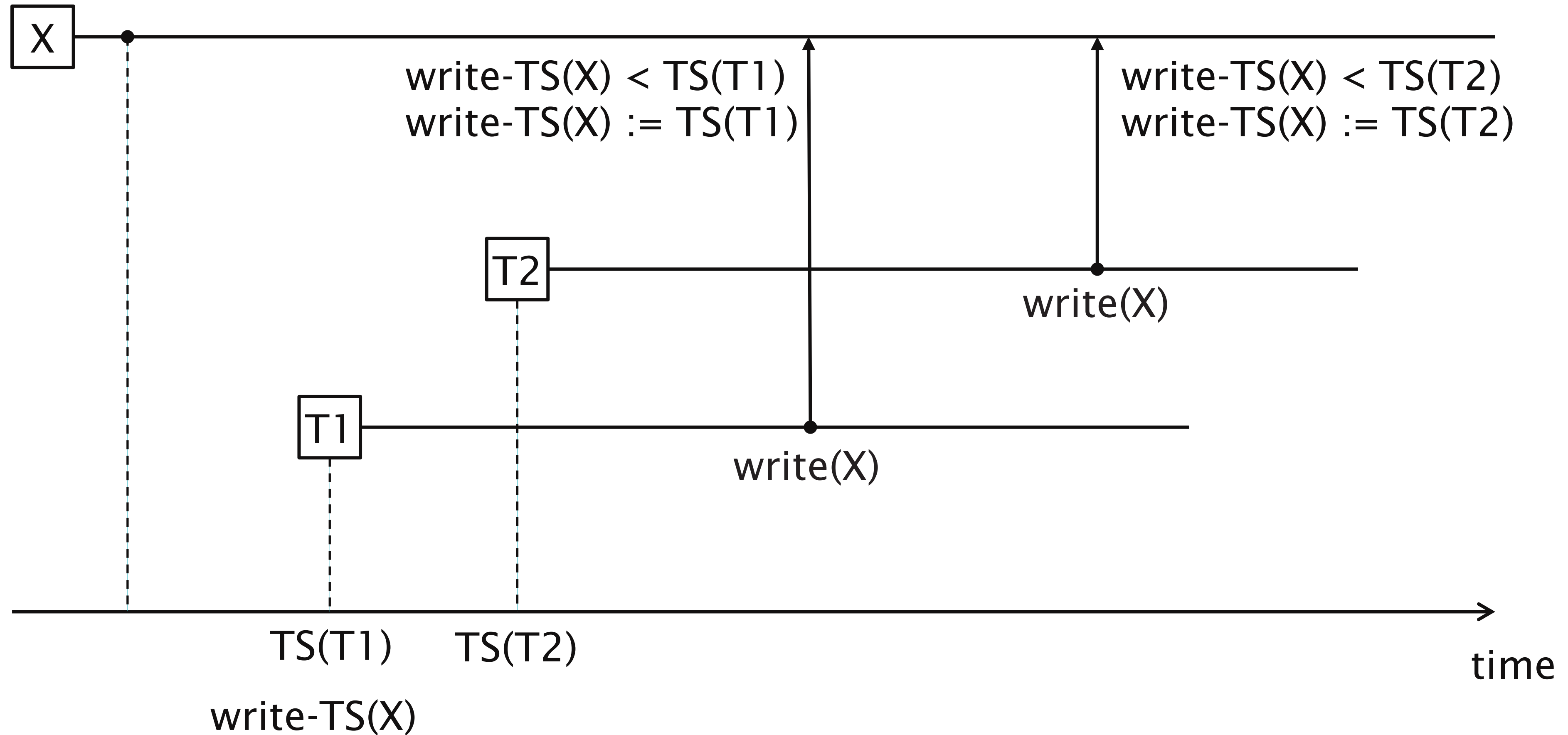


Basic Timestamp Ordering

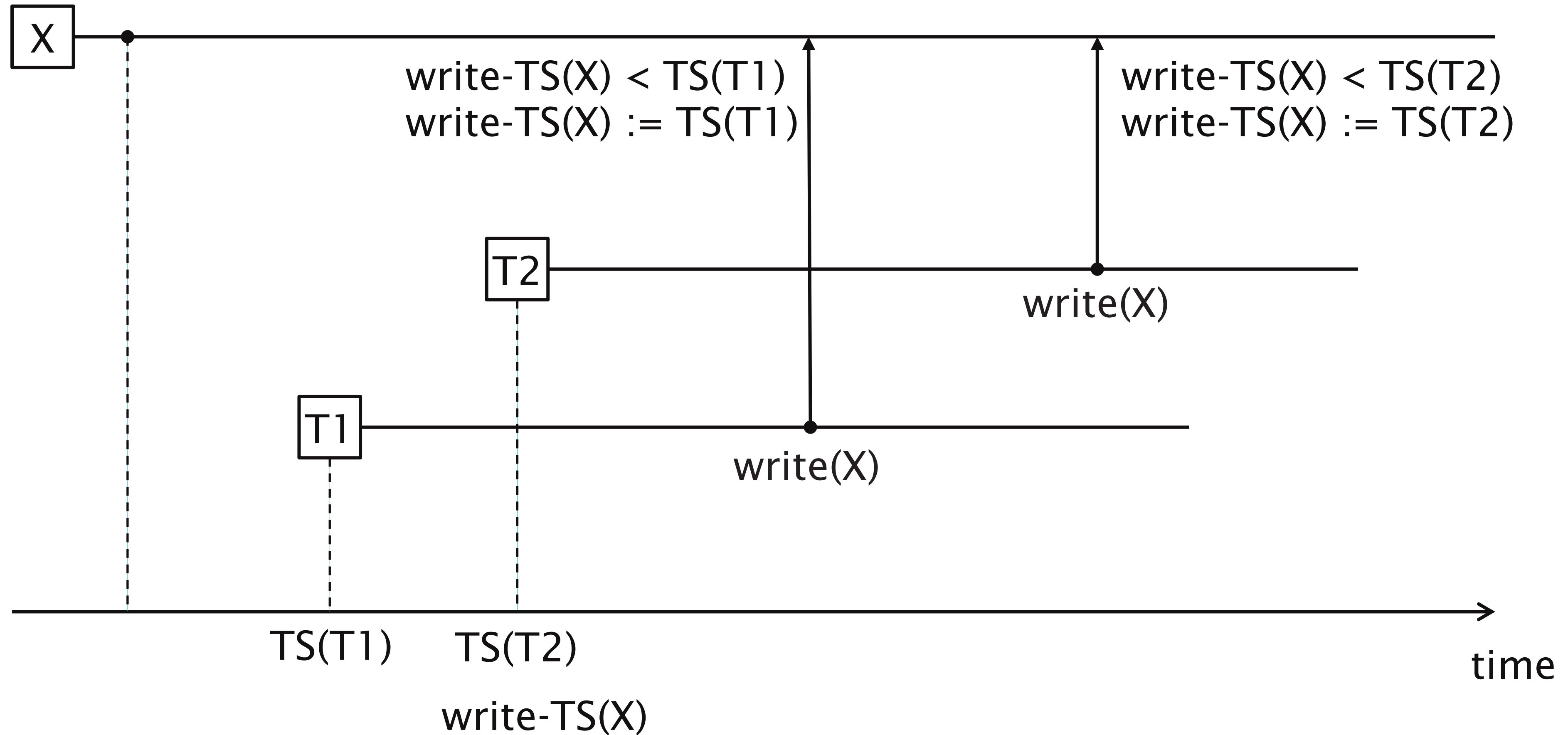


University of Southampton

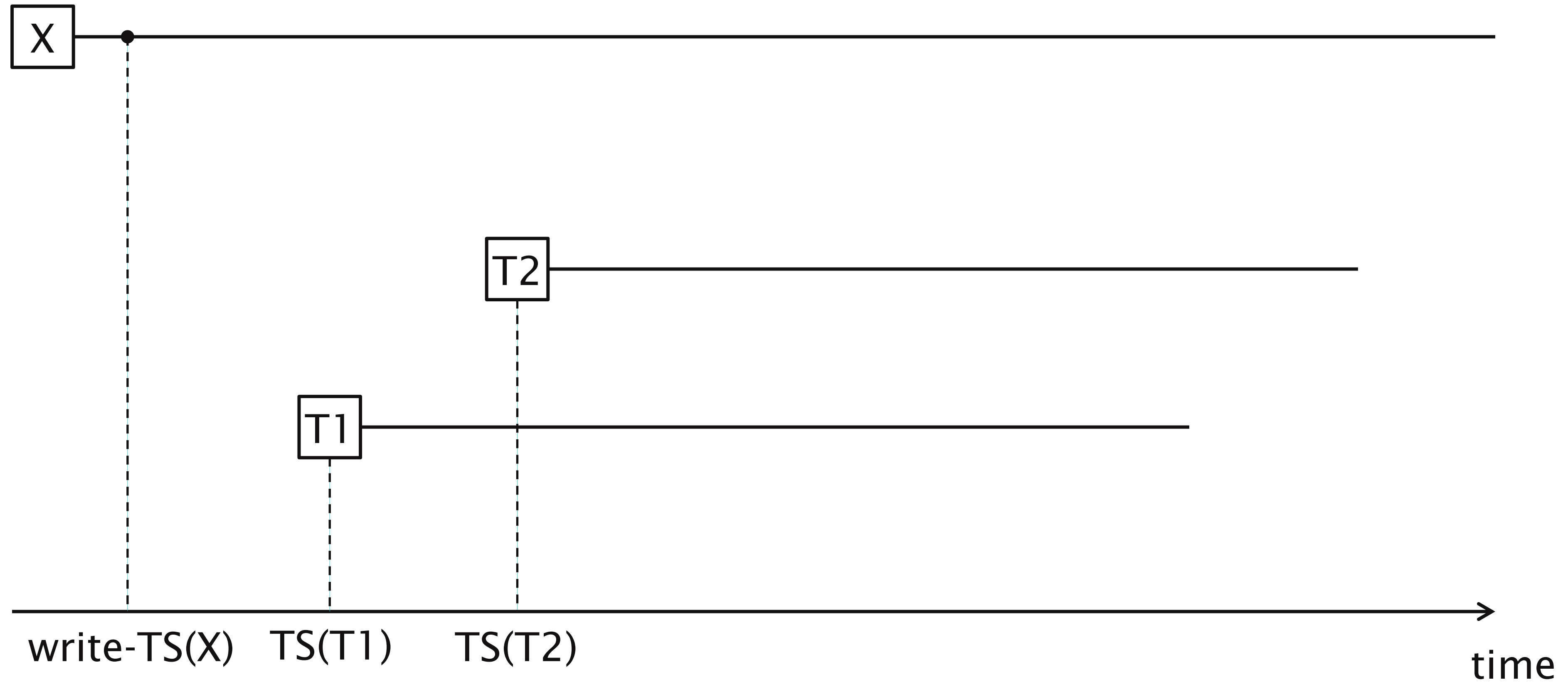
Basic Timestamp Ordering



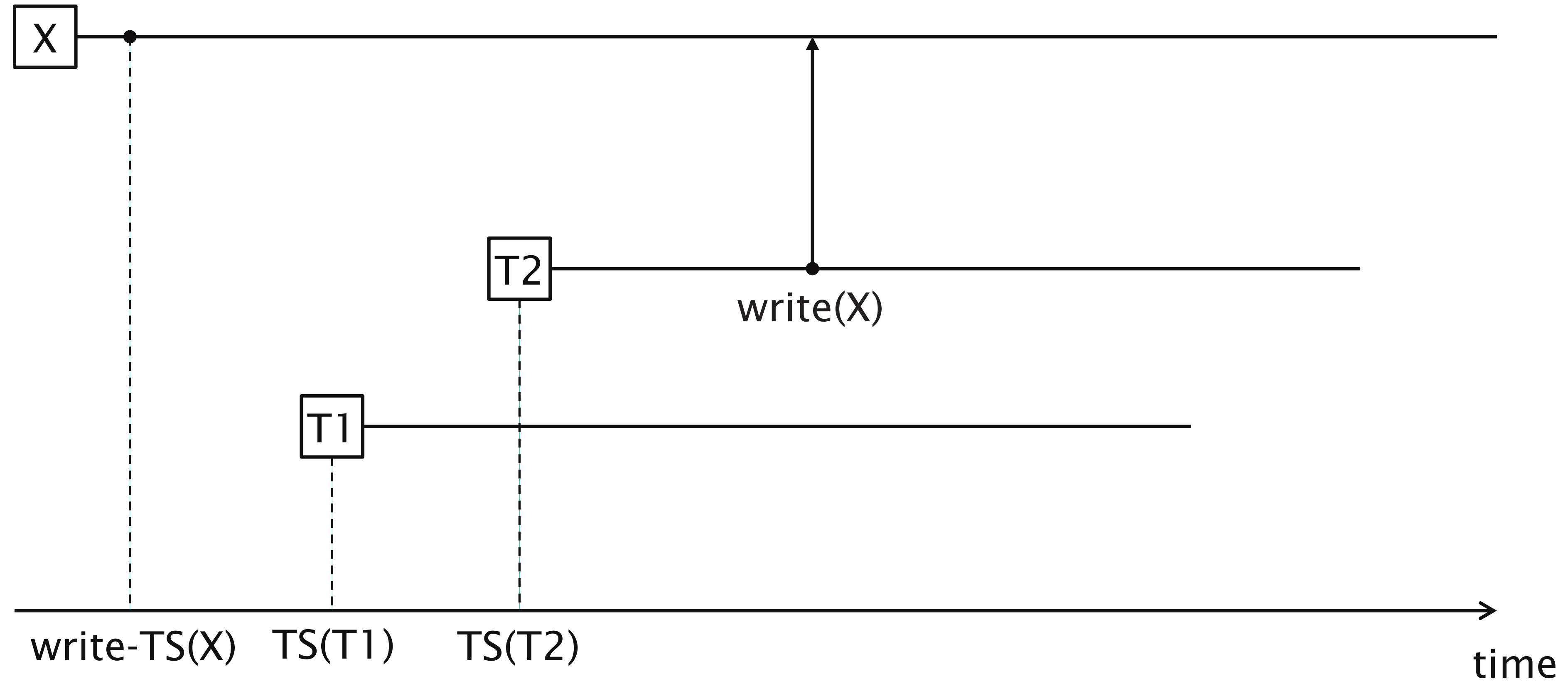
Basic Timestamp Ordering



Basic Timestamp Ordering

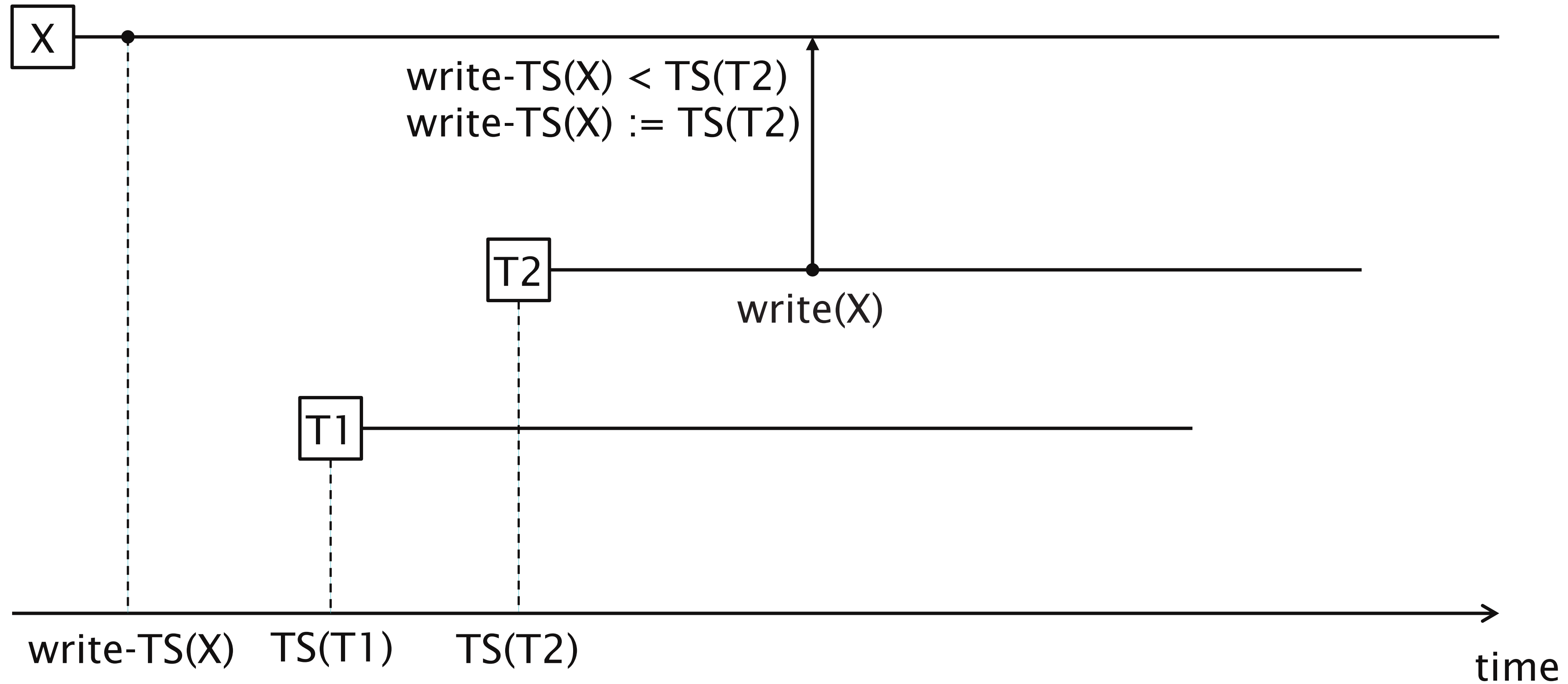


Basic Timestamp Ordering



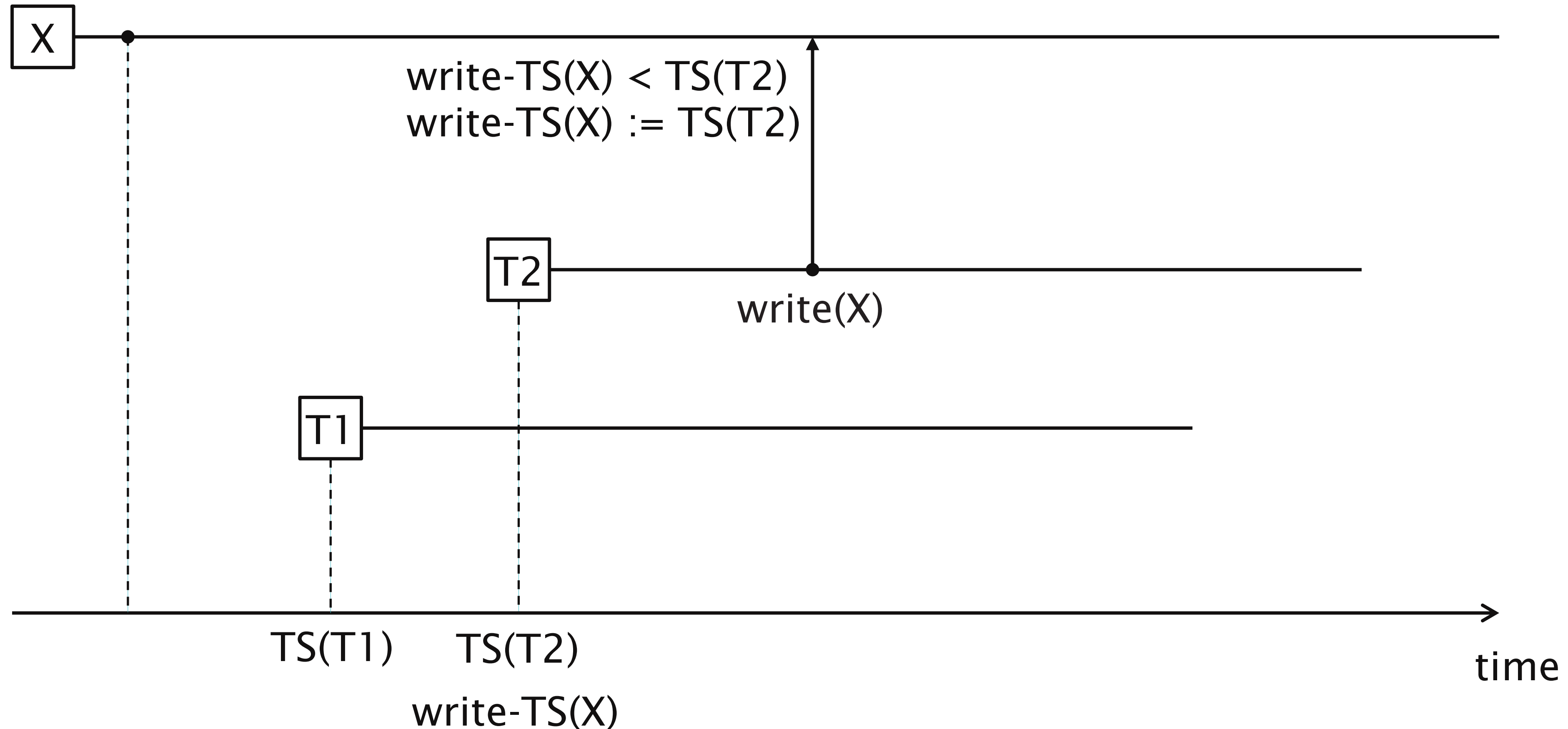
University of Southampton

Basic Timestamp Ordering

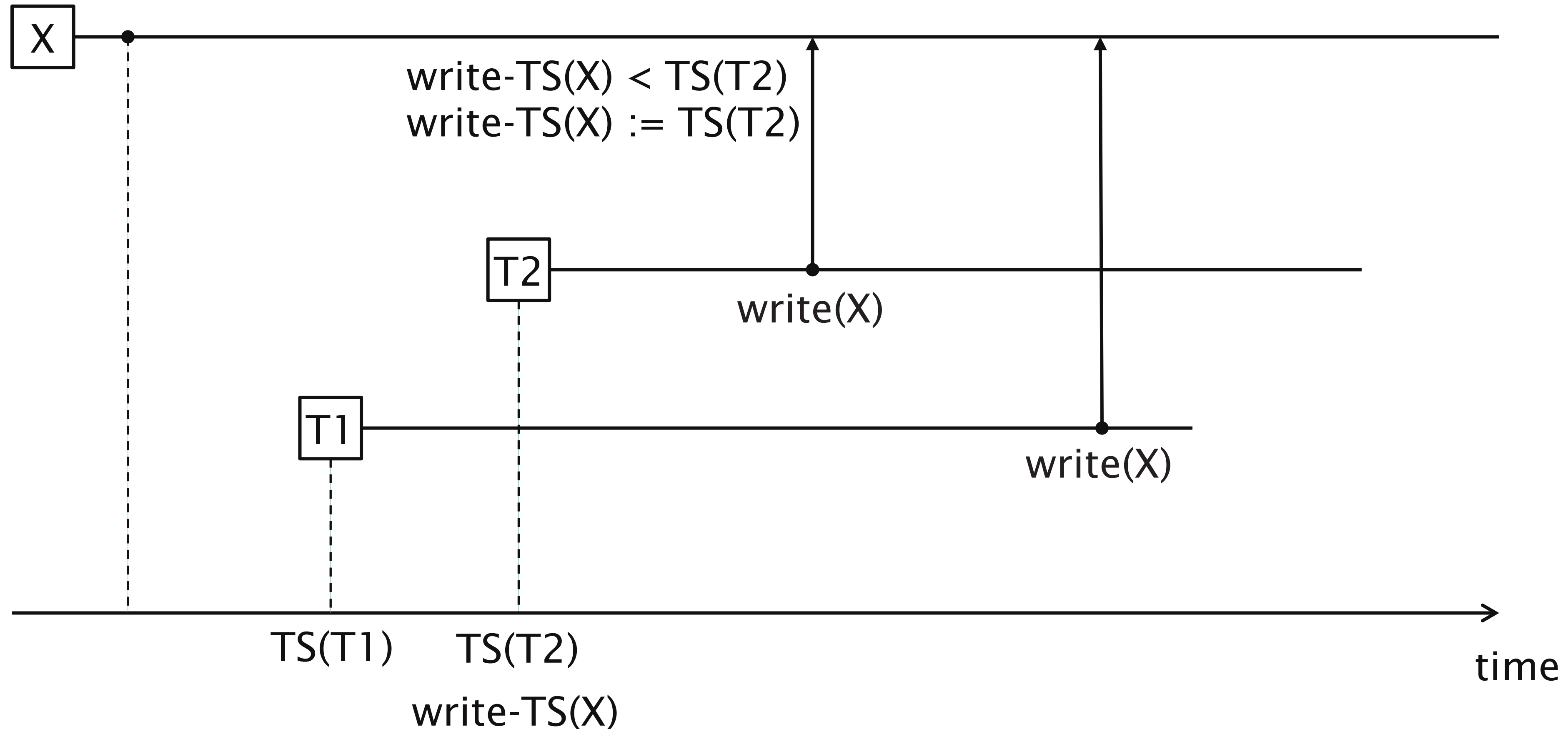


University of Southampton

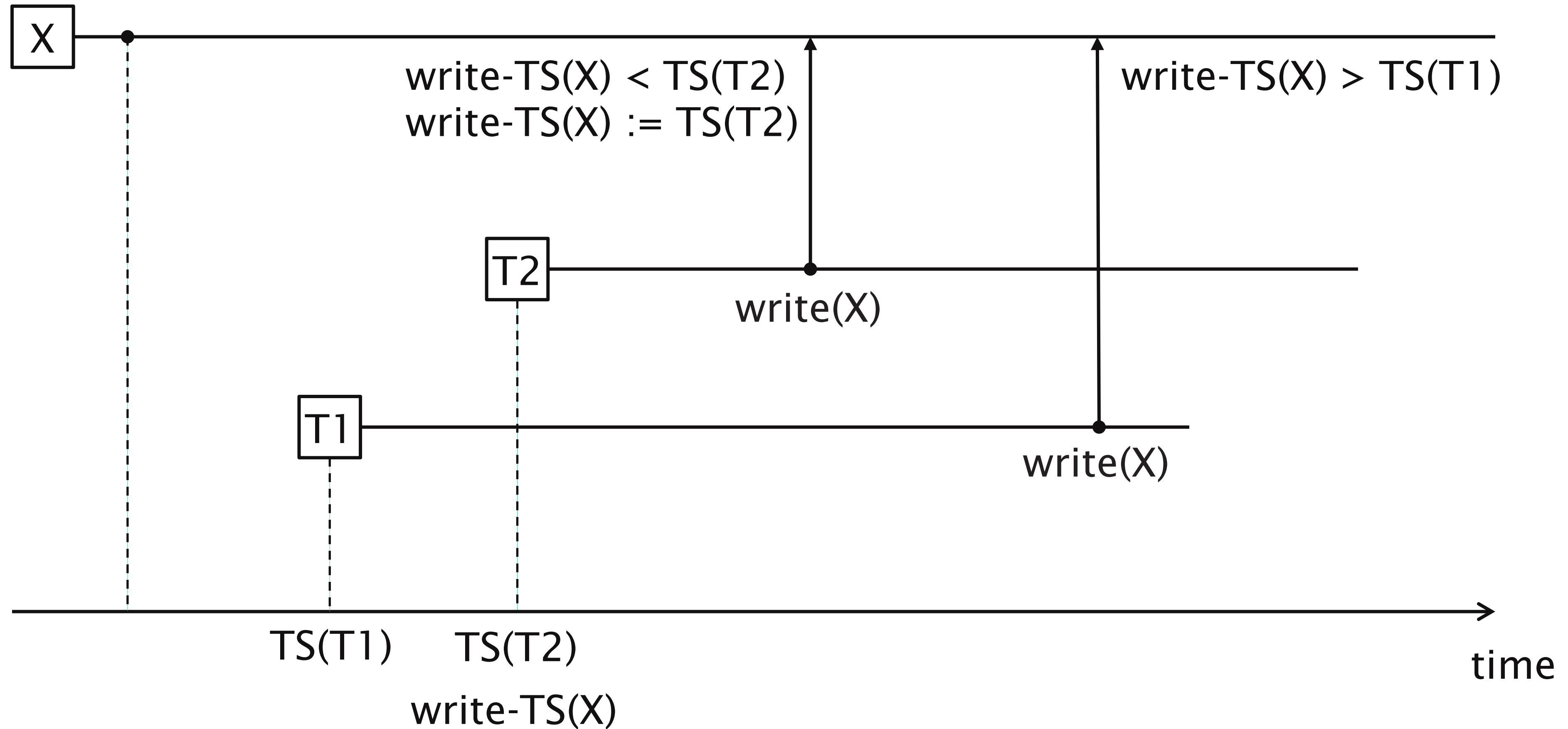
Basic Timestamp Ordering



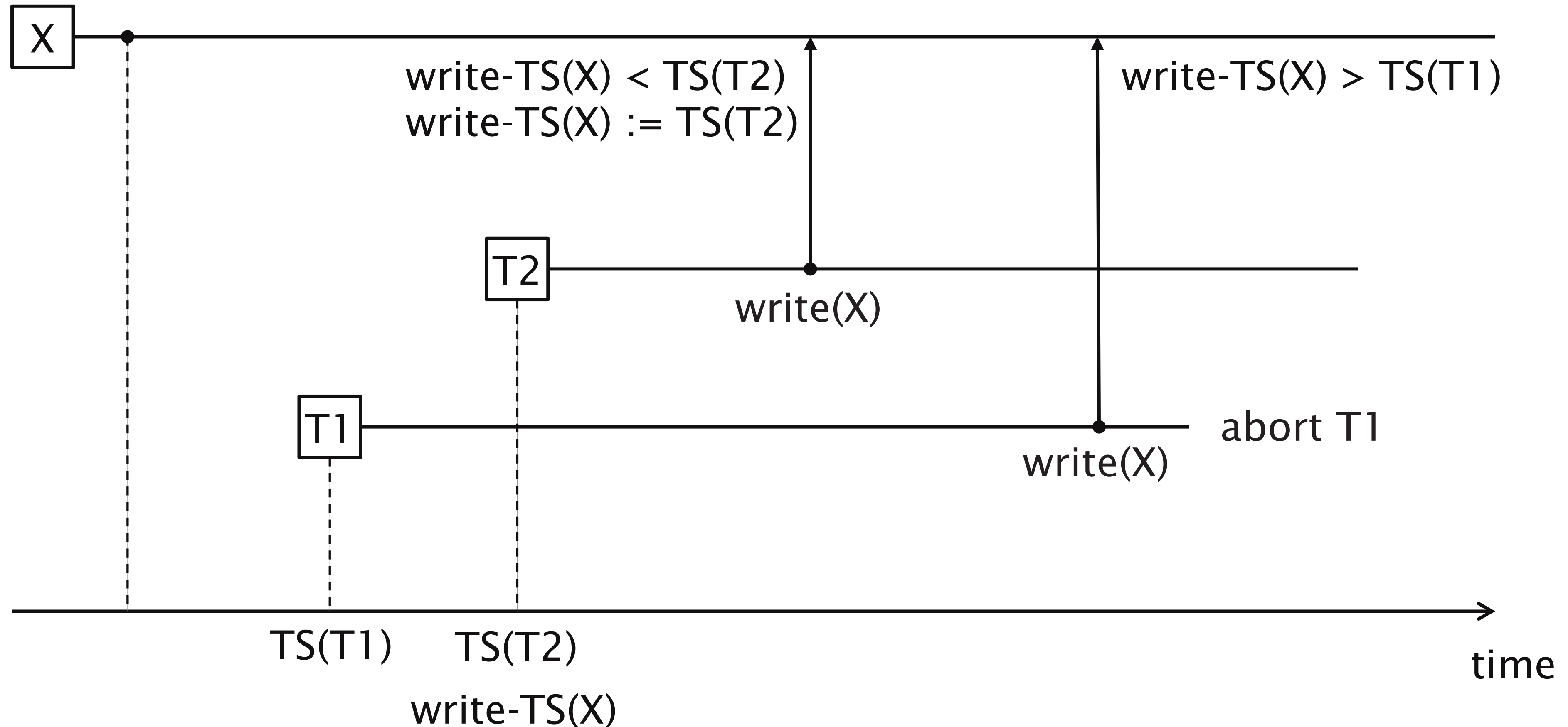
Basic Timestamp Ordering



Basic Timestamp Ordering



Basic Timestamp Ordering

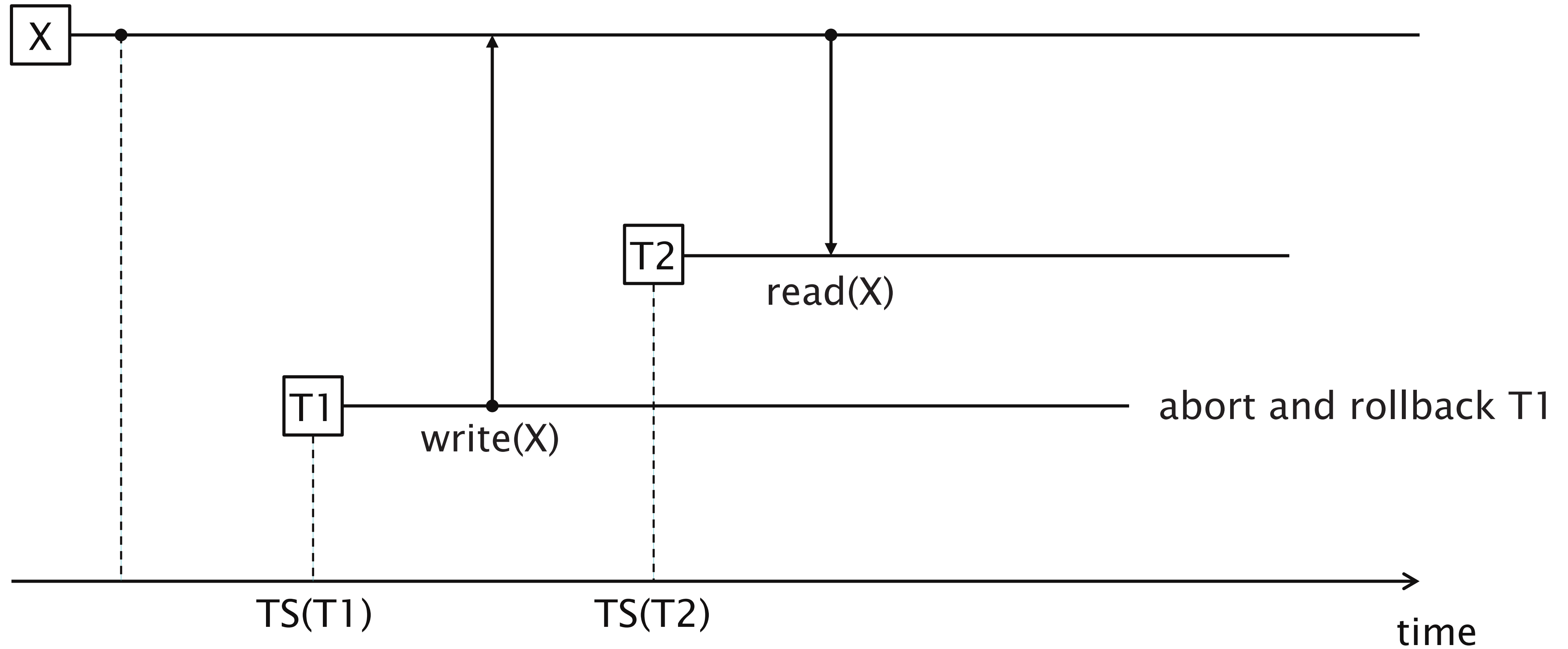


University of Southampton

Basic Timestamp Ordering: read(X)

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

Dirty Read



Dirty Read

- In the previous example, T2 has an inconsistent local value for X once T2 has been rolled back
- We can address the dirty read by adding a commit bit commit(X) that indicates whether the most recent transaction to write to X has been committed
- This will change the rules for reading and writing as follows

Basic Timestamp Ordering: read(X)

```
if TS(T) ≥ write-TS(X)
then
      if commit(X) is true
      then
            execute read(X)
            set read-TS(X) to max(TS(T), read-TS(X))
      else
            delay execution until commit(X) is true
else
      abort and rollback T
```

Basic Timestamp Ordering: write(X)

```
if TS(T) ≥ read-TS(X) and TS(T) ≥ write-TS(X)
then
      execute write(X)
      set commit(X) to false
      set write-TS(X) to TS(T)
else
      abort and rollback T
```

University of Southampton

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 TS(T) ≥ read-TS(X) and TS(T) ≥ write-TS(X)
then
      execute write(X)
      set commit(X) to false
      set write-TS(X) to TS(T)
else if TS(T) ≥ read-TS(X) and TS(T) < write-TS(X)
      if commit(X) is true
            ignore write (X)
      else
            delay execution until commit(X) is true
else
      abort and rollback T
```

Advanced Transactions

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

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

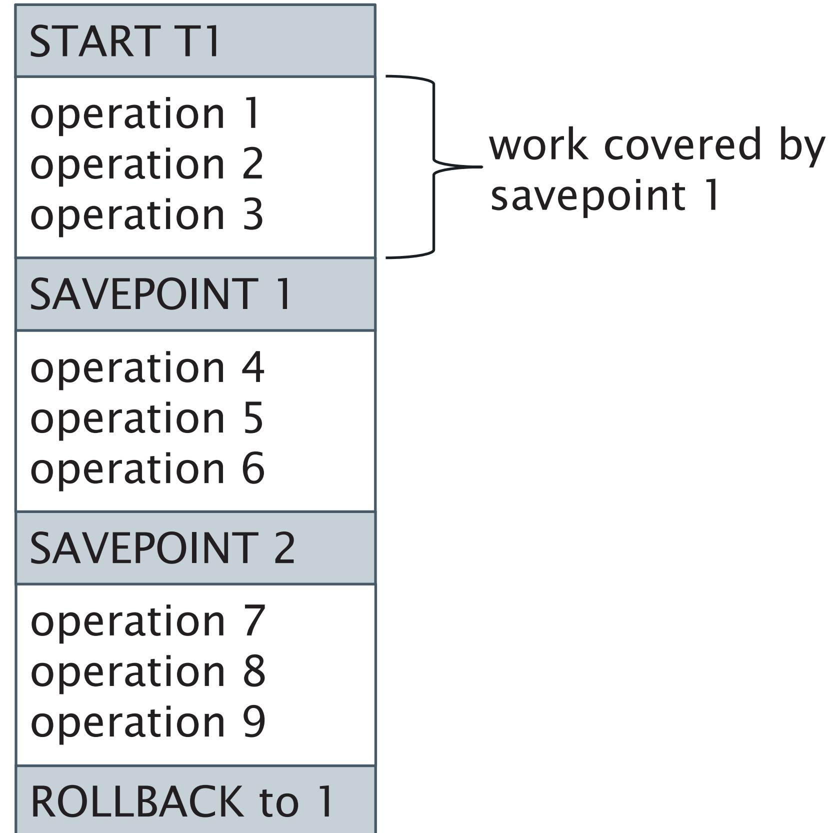
Savepoints may be persistent

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

Savepoints

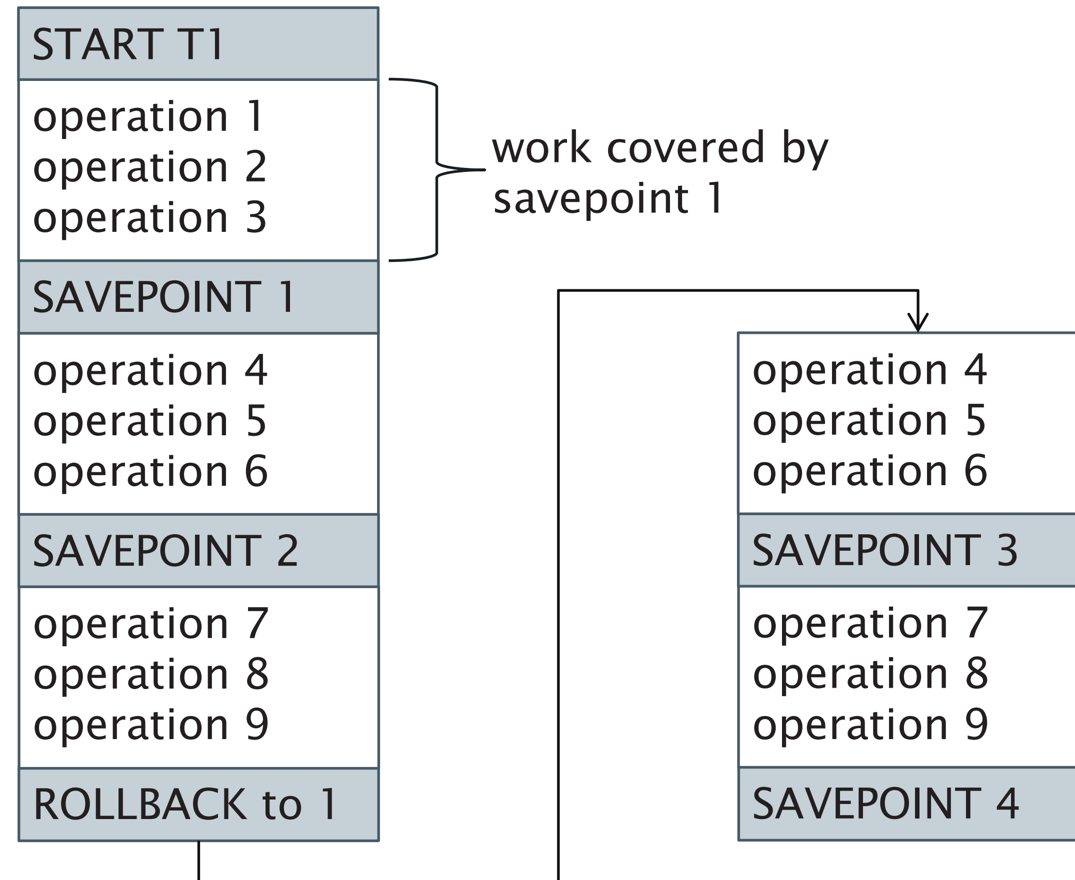
START T1
operation 1 operation 2 operation 3
SAVEPOINT 1
operation 4 operation 5 operation 6
SAVEPOINT 2
operation 7 operation 8 operation 9
ROLLBACK to 1

Savepoints



work covered by
savepoint 1

Savepoints



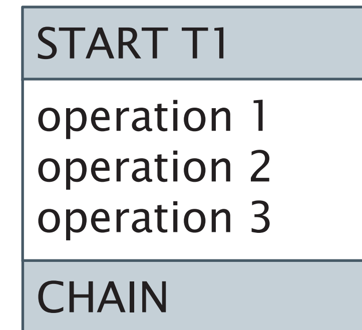
Chained Transactions

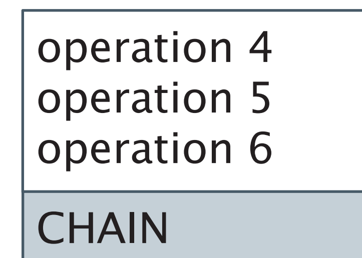
Transaction broken into subtransactions which are executed serially

On chaining to the next subtransaction:

- commit results
- keep (some) locks

Cannot rollback to previous subtransaction





University of Southampton

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

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

Subtransactions may abort without aborting their parent transaction

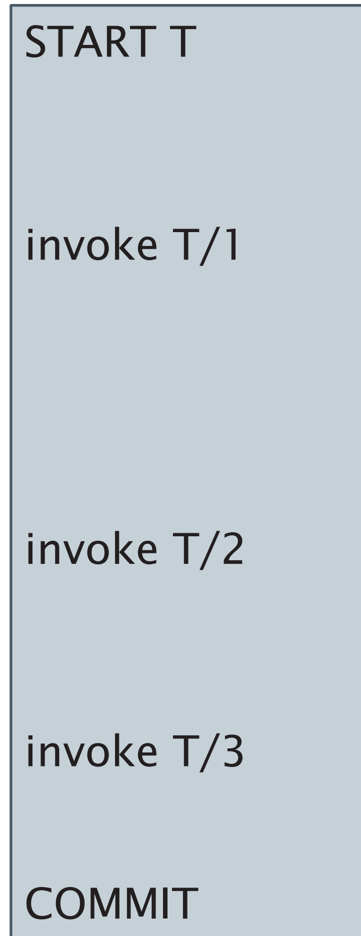
- May restart subtransaction

Three rules for nested transactions:

- Commit Rule
- Rollback Rule
- Visibility Rule

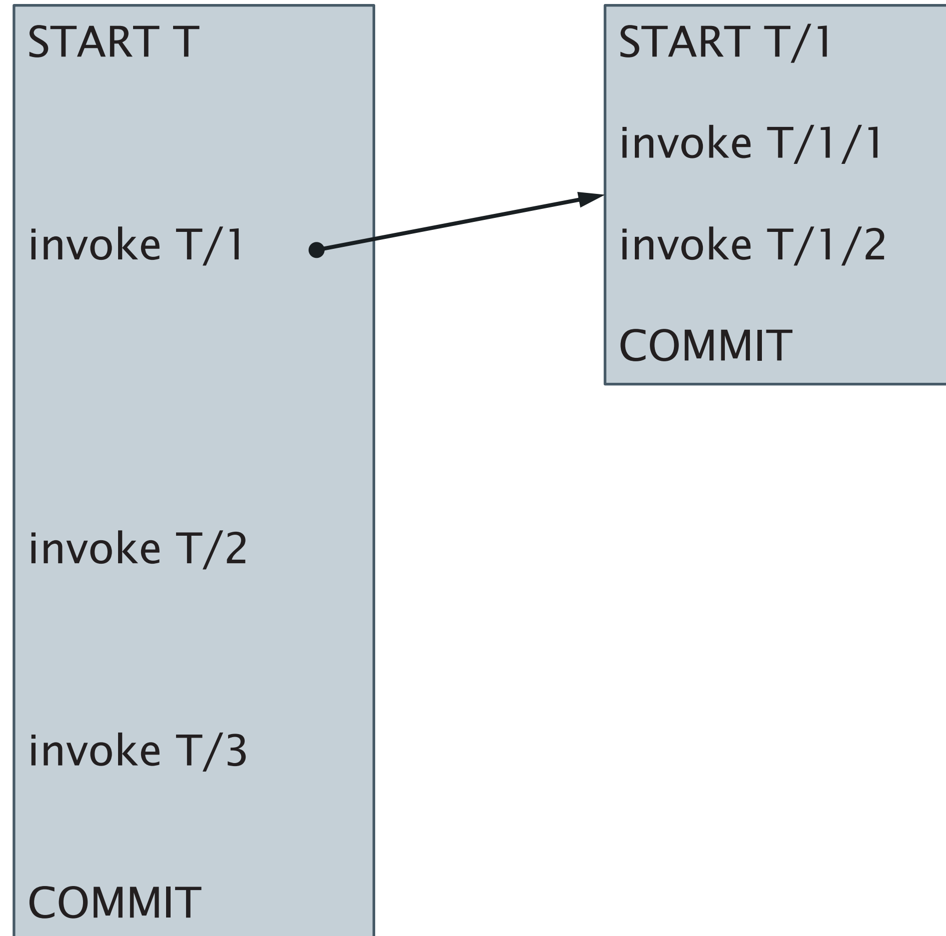
University of Southampton

Nested Transactions

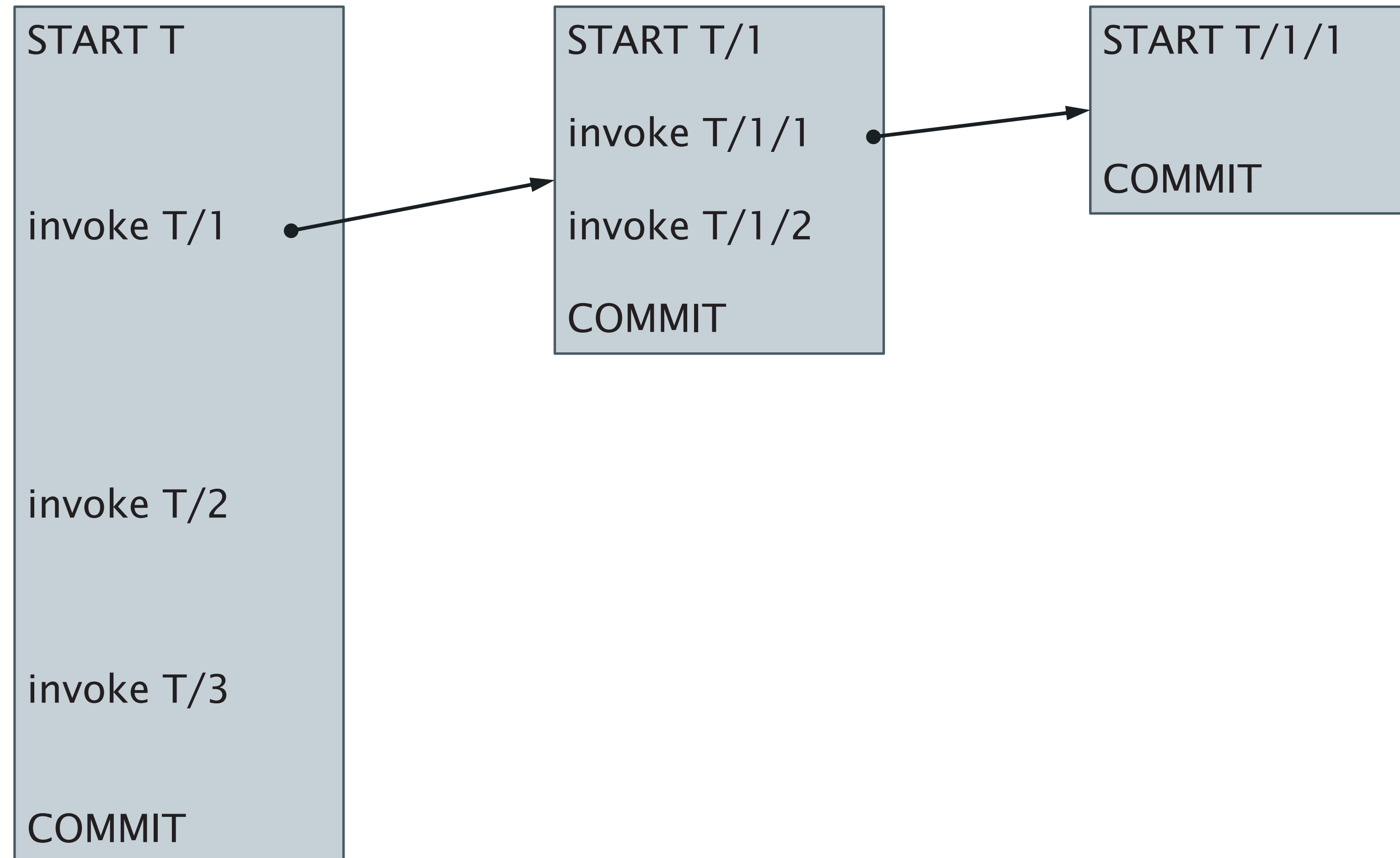


University of Southampton

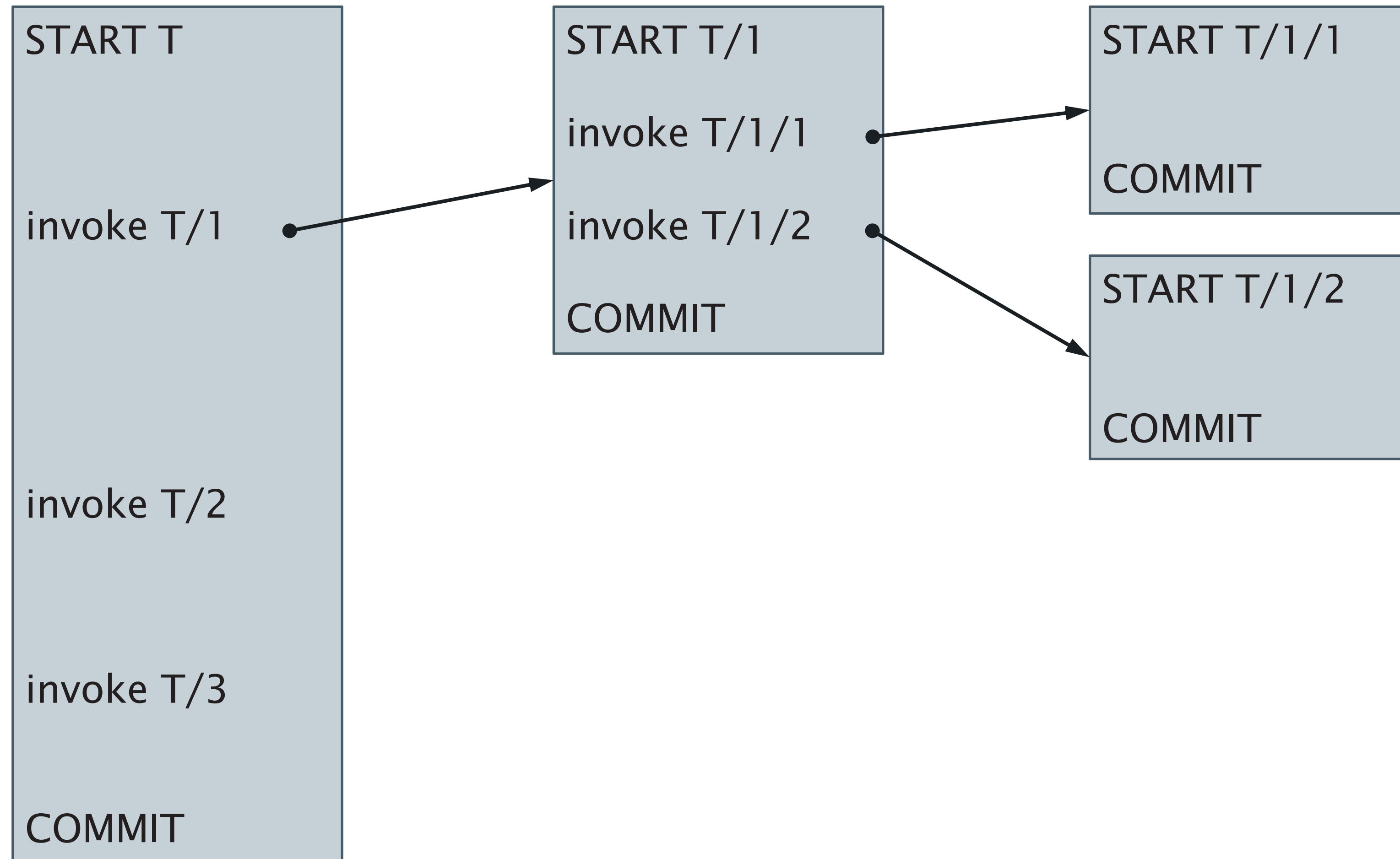
Nested Transactions



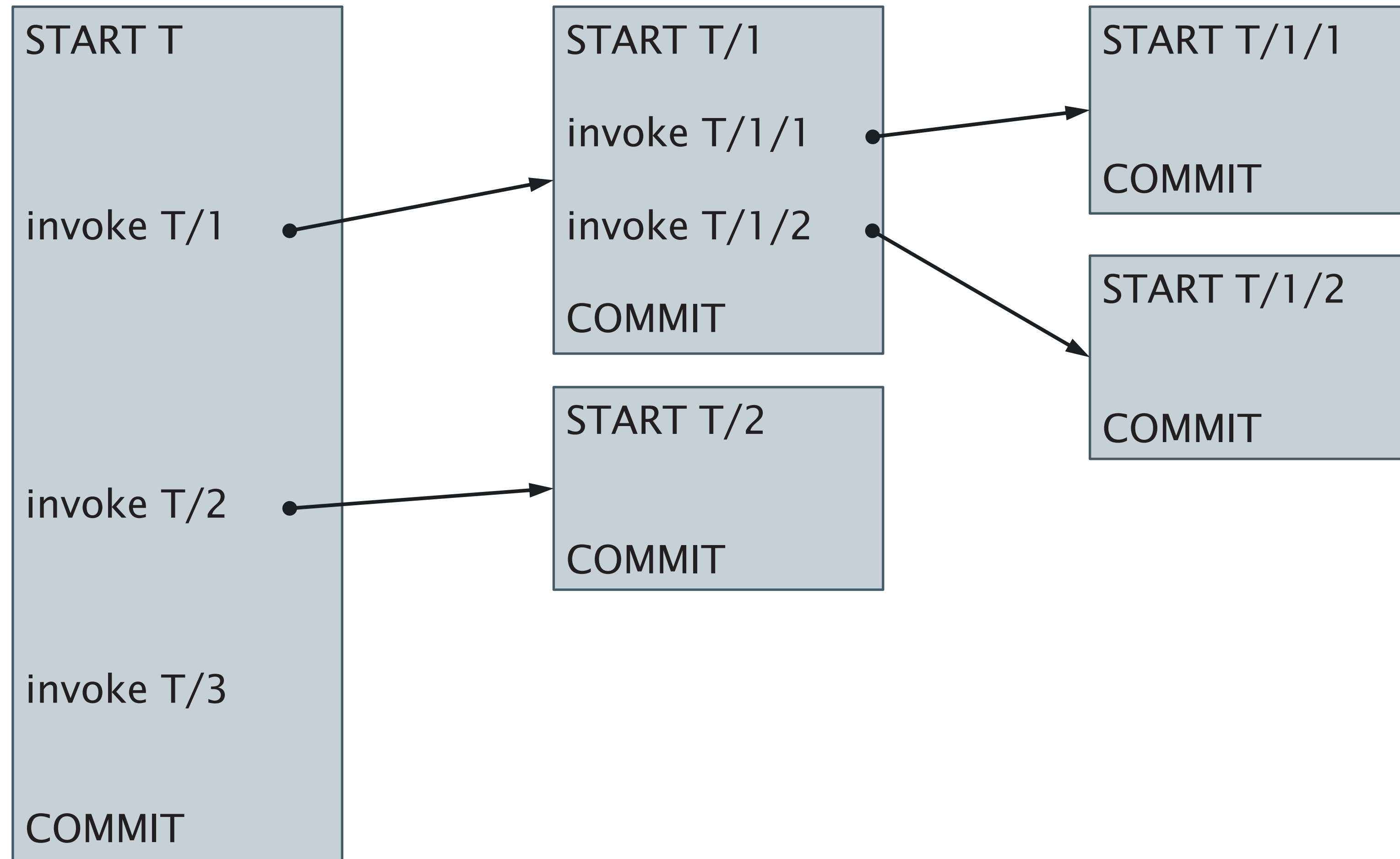
Nested Transactions



Nested Transactions

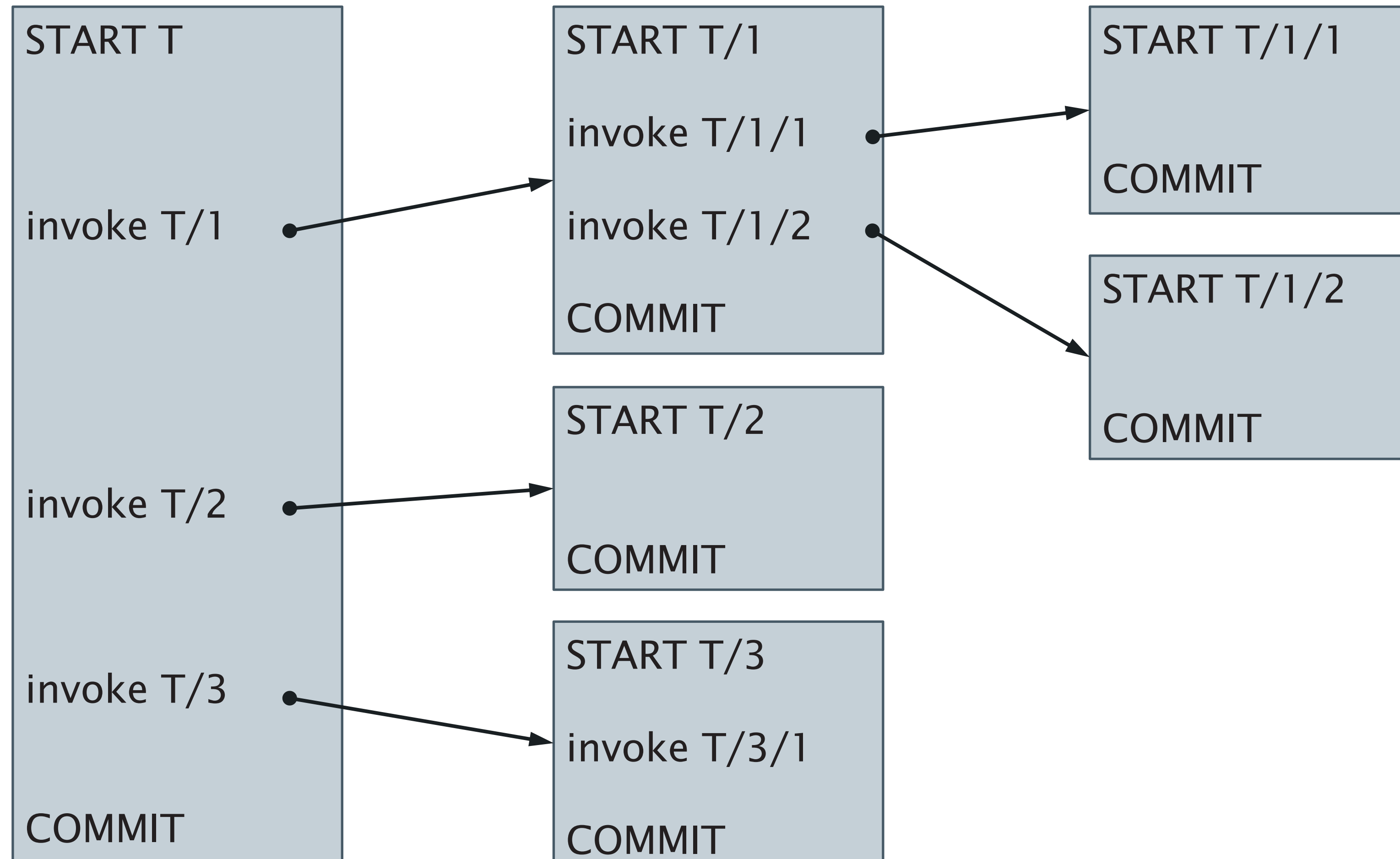


Nested Transactions

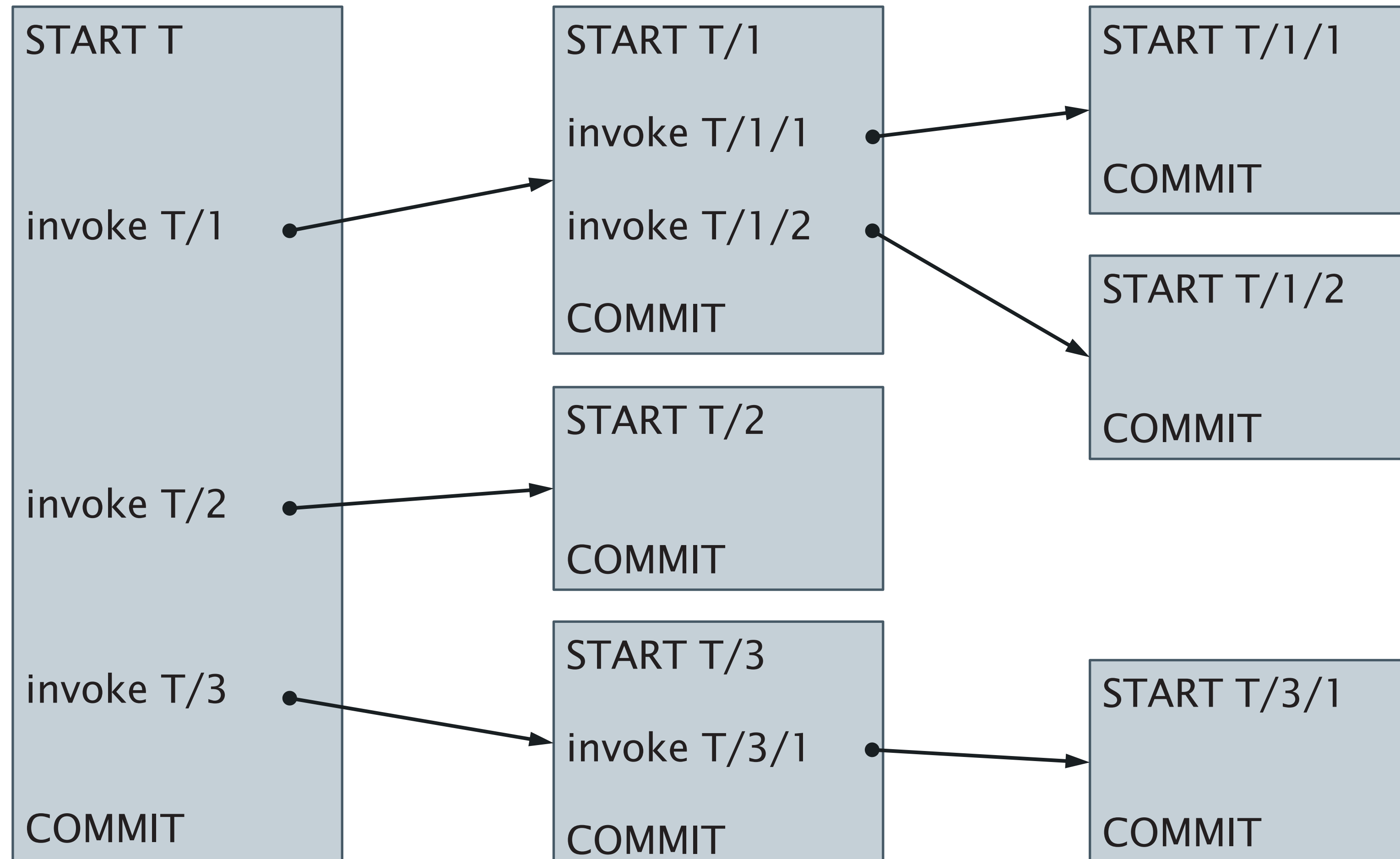


University of Southampton

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

University of Southampton

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

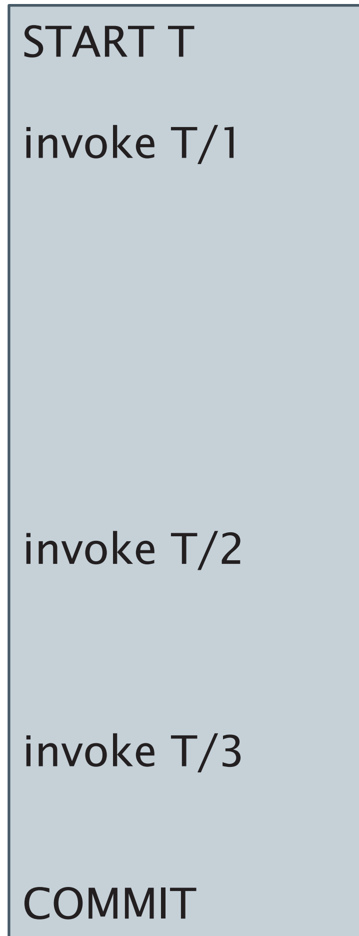
Observations

Nesting and program modularisation complement each other

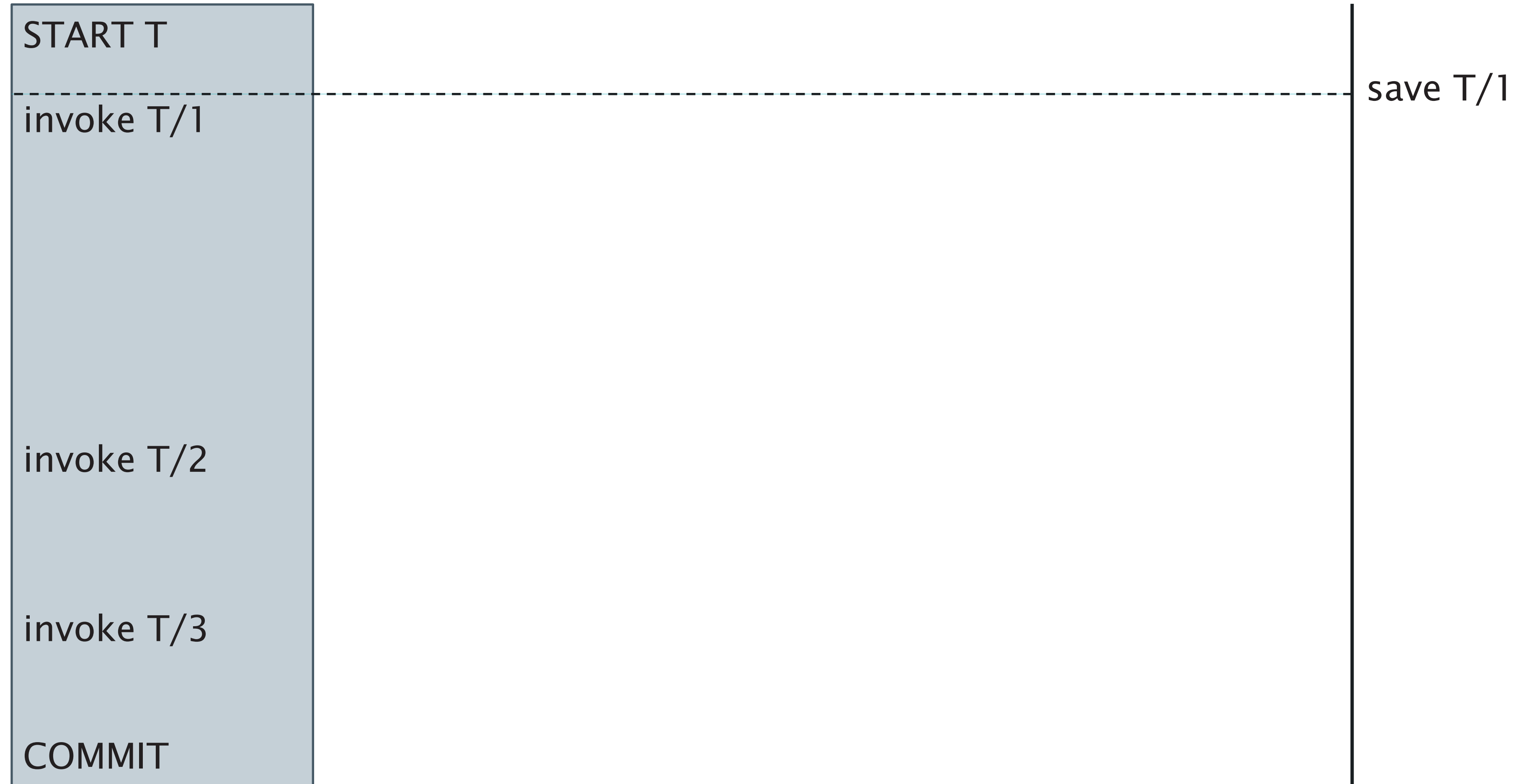
- Well designed module has a clean interface, and no global variables
- If it touches the database, the database is a large global variable
- If the module is protected as a subtransaction, then database changes are kept clean too

Nested transactions permit intra-transaction parallelism

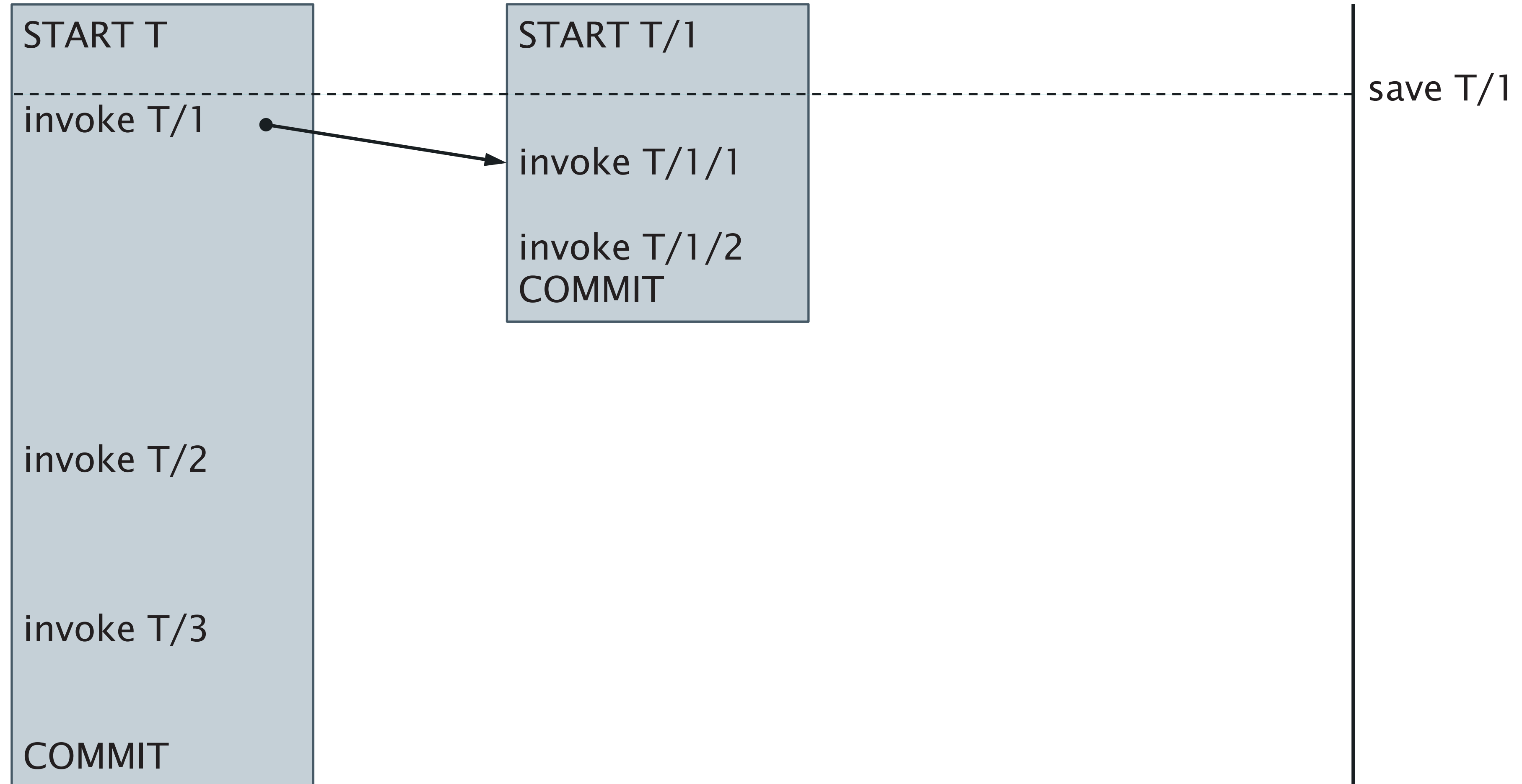
Emulating Nesting with Savepoints



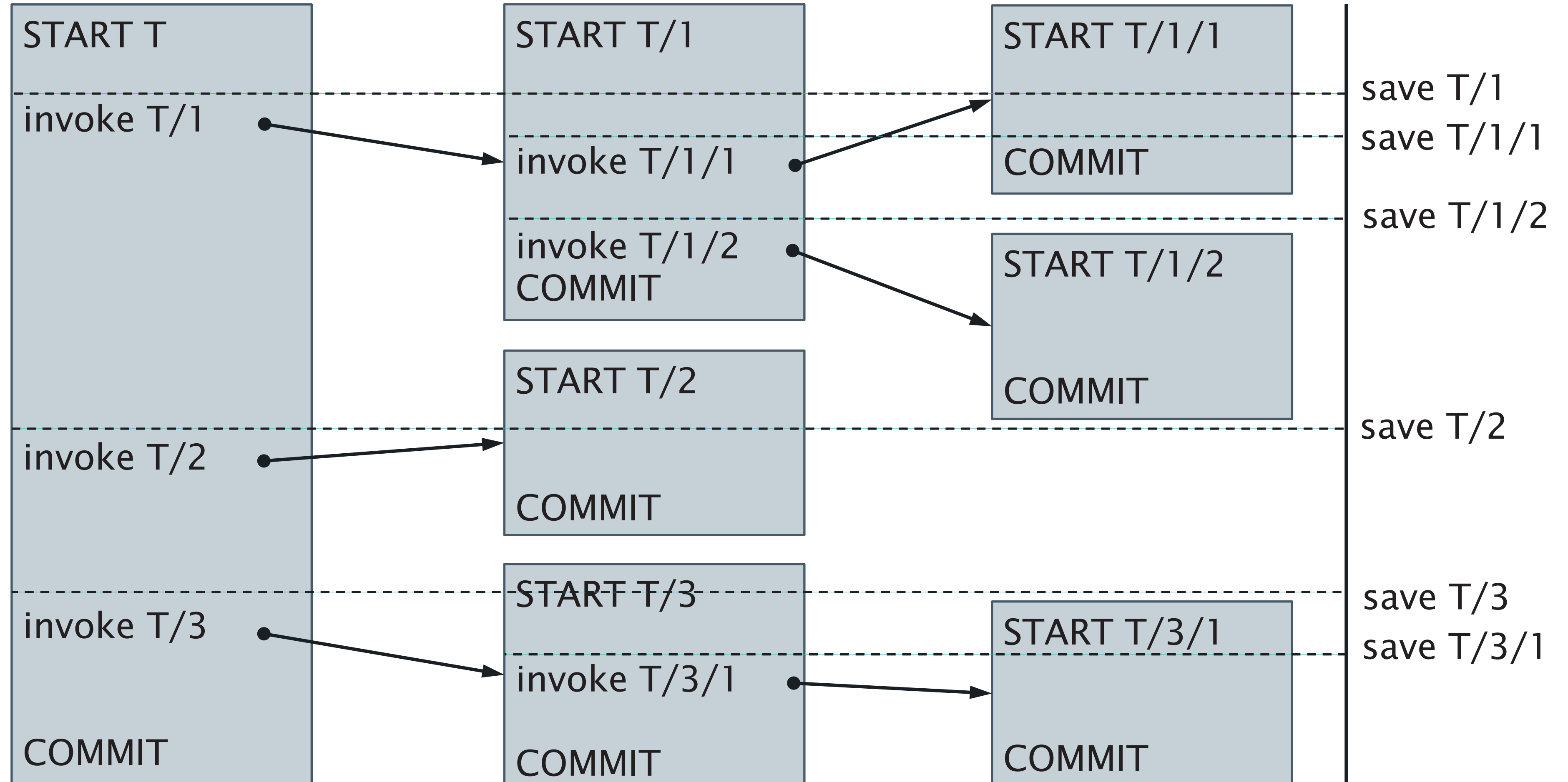
Emulating Nesting with Savepoints



Emulating Nesting with Savepoints



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

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

University of Southampton

Next Lecture: Logging and Recovery