Distributed Transaction Processes

Distributed Systems
Santa Clara University

Transaction Concept

- Transactions:
  - Transparent concurrency
  - Transparent recovery
- **A**tomic
  - Transaction is completely executed or not at all
- **C**onsistency
  - Consistency constraints are preserved by a transaction
- **I**solation
  - Each transaction behaves as if it were operating alone
- **D**urability
  - Updates made by a committed transaction are durable

Transaction Concept

- Computational model
  - Elementary operations on data objects
  - Transactions are sequences of these operations
  - The execution of several transactions is described by a schedule or history
  - Histories where ACID properties are guaranteed are correct
  - Generate algorithms / protocols

Transaction Concept

- Computational model: Page model
  - Transaction is a sequence of elementary operations
    - read, write
  - on a single page
  - Can talk about the value of a page at each level in the history

Transaction Concept

- Computational model
  - Example transaction:
    - $t = r(x)r(y)r(z)w(u)w(x)$
  - Transaction reads three values
  - Can use these values to write new values
  - One value has potentially changed

Transaction Concept

- Not necessary to assume that the steps in a transaction are sequential
- Just need a partial order between the steps

Concurrency Control

- Canonical Concurrency Problems
  - Lost update problem

t1:	t2:
`r(x)`	
	`r(x)`
`x+=30`	
	`x+=120`
`w(x)`	
	`w(x)`

Concurrency Control

- Inconsistent read problem
  - Assume the constraint x+y==0

t1:	t2:
	`r(x)`
	`x-=10`
	`w(x)`
	`r(y)`
`r(x)`	`y+=10`
`r(y)`	
	`w(y)`

Concurrency Control

- Dirty read problem (Reading uncommitted data)

t1:	t2:
`r(x)`	
`x+=100`	
`w(x)`	
	`r(x)`
`rollback`	`x-=100`
	`w(x)`

Concurrency Control

- History
    - The sequence of all elementary operations of all transactions
- Schedule
    - A prefix of a history
- Serial History
    - A history where all elementary operations of one transaction are done before those of another transaction or where all elementary operations of one transaction are done after those of another transaction

Concurrency Control

- Herbrand Semantics
  - A notion to make precise what is the result of a history without specifying values
  - Because we could change a non-serial history to a serial one by altering values written
  - For example, set all transfer, withdrawal, and deposit amounts to 0 to make any history in a bank database serial

Concurrency Control

- Herbrand value of a read is the Herbrand value of the last write
- Herbrand value of a write is a generic functions of all reads of the executor of the write

Final state serializability

- Final state serializability
  - Final state equivalence
    - Two histories are <u>*final state equivalent*</u> if
      - They contain the same operations
      - They have the same Herbrand semantics

Final state serializability

- Final state equivalency example

$$
\begin{aligned}
s_1 &= r_1(x)r_2(y)w_1(y)r_3(z)w_3(z)r_2(x)w_2(z)w_1(x) \\
s_2 &= r_3(z)w_3(z)r_2(y)r_2(x)w_2(z)r_1(x)w_1(y)w_1(x)
\end{aligned}
$$

Final state serializability

- Final state equivalency example

$$
\begin{aligned}
s_1 &= r_1(x)r_2(y)w_1(y)r_3(z)w_3(z)r_2(x)w_2(z)w_1(x) \\
s_2 &= r_3(z)w_3(z)r_2(y)r_2(x)w_2(z)r_1(x)w_1(y)w_1(x)
\end{aligned}
$$

Herbrand value of *x* depends on what 1 saw, which is *x* unaltered in both schedules
Herbrand value of *y* depends on what 1 saw
Herbrand value of *z* depends on what 2 saw, which is *x* and *y* unaltered

Final state serializability

- Finite state equivalency example:

$$
\begin{aligned}
s_a &= r_1(x)r_2(y)w_1(y)w_2(y) \\
s_b &= r_1(x)w_1(y)r_2(y)w_2(y)
\end{aligned}
$$

Final state serializability

- Finite state equivalency example:

$$\begin{aligned} s_a &= r_1(x)r_2(y)w_1(y)w_2(y) \\ s_b &= r_1(x)w_1(y)r_2(y)w_2(y) \end{aligned}$$

The Herbrand value of *y* is the original value of *y* in the first schedule and depends on the write by 1 in the second schedule.

Final state serializability

- We can decide final state serializability with the Life-Reads-From relation
  - Reads-from
    - A transaction reads a value after it has been last written by another transaction
  - Alive
    - Final value depends on the transaction
- Two schedules are final state serializable iff
  - they consist of the same operations
  - they have the same life-reads-from relation

Final state serializability

- A schedule is final-state serializable
  - IFF it is final state equivalent to a serial schedule
  - IFF it has the same life-read-from relation as a serial schedule

View Serializability

- Final state serializability is still insufficient
  - Lost update anomaly is detected (good)

$$r_1(x) r_2(x) w_1(x) w_2(x)$$

  - Inconsistent read is still allowed by final state serializability (bad)

$$r_2(x) w_2(x) r_1(x) r_1(y) r_2(y) w_2(y)$$

Transaction 1 makes an inconsistent read, but the final state ignores it.

View Serializability

- View equivalence
  - Two schedules are view equivalent if
    - They have the same set of operations
    - The Herbrand values of their schedules are equal
    - The Herbrand values at each read or write step are equivalent

View Serializability

- View equivalence
  - Two schedules are view equivalent
  - IFF they have the same read-from relation

Conflict Serializability

- Easier to test than view serializability
- Conflict relation:
  - Two operations are in conflict
    - If they access the same data item
    - At least one of them is a write
  - Conflict relation: transitive closure
- Conflict equivalence
  - Two schedules are conflict equivalent
    - They have the same set of operations
    - Their conflict set is the same

Conflict Serializability

$$w_1(x)r_2(x)w_2(y)r_1(y)w_1(y)w_3(x)w_3(y)$$

Conflict Serializability

$$w_1(x) r_2(x) w_2(y) r_1(y) w_1(y) w_3(x) w_3(y)$$

$$\{(w_1(x), r_2(x)), (w_1(x), w_3(x)), (r_2(x), w_3(x)), (w_2(y), r_1(y)), (w_2(y), w_1(y)), (w_1(y), w_3(y))\}$$

Conflict Serializability

- A schedule is conflict serializable
  - IFF it is conflict equivalent to a serial schedule

Conflict Serializability

- Algebraic notation:
  - C1 $r_i(x)r_j(y) \sim r_j(y)r_i(x) \text{ if } i \neq j$
  - C2 $r_i(x)w_j(y) \sim w_j(y)r_i(x) \text{ if } i \neq j \text{ and } x \neq y$
  - C3 $w_i(x)w_j(y) \sim w_j(y)w_i(x) \text{ if } i \neq j \text{ and } x \neq y$

Conflict Serializability

- Two schedules with the same operations are equivalent
  - Iff one can be transformed to the other using the commutativity rules

Commit Serializability

- Conflict serializability does not detect the dirty read problem
  - Since it does not pay attention to commit and abort operations

- Correctness criterion should only take committed transactions into account
- Since systems can crash
  - All prefixes of a schedule have to be correct

Commit Serializability

- A schedule is commit conflict serializable iff
  - Projection on committed transactions is conflict serializable

Concurrency Control Algorithms

- How to deal with bad situations:
  - Strategy 1: Never get into a bad situation
  - Strategy 2: Know how to get out of a bad situation (rollback)

Concurrency Control Algorithms

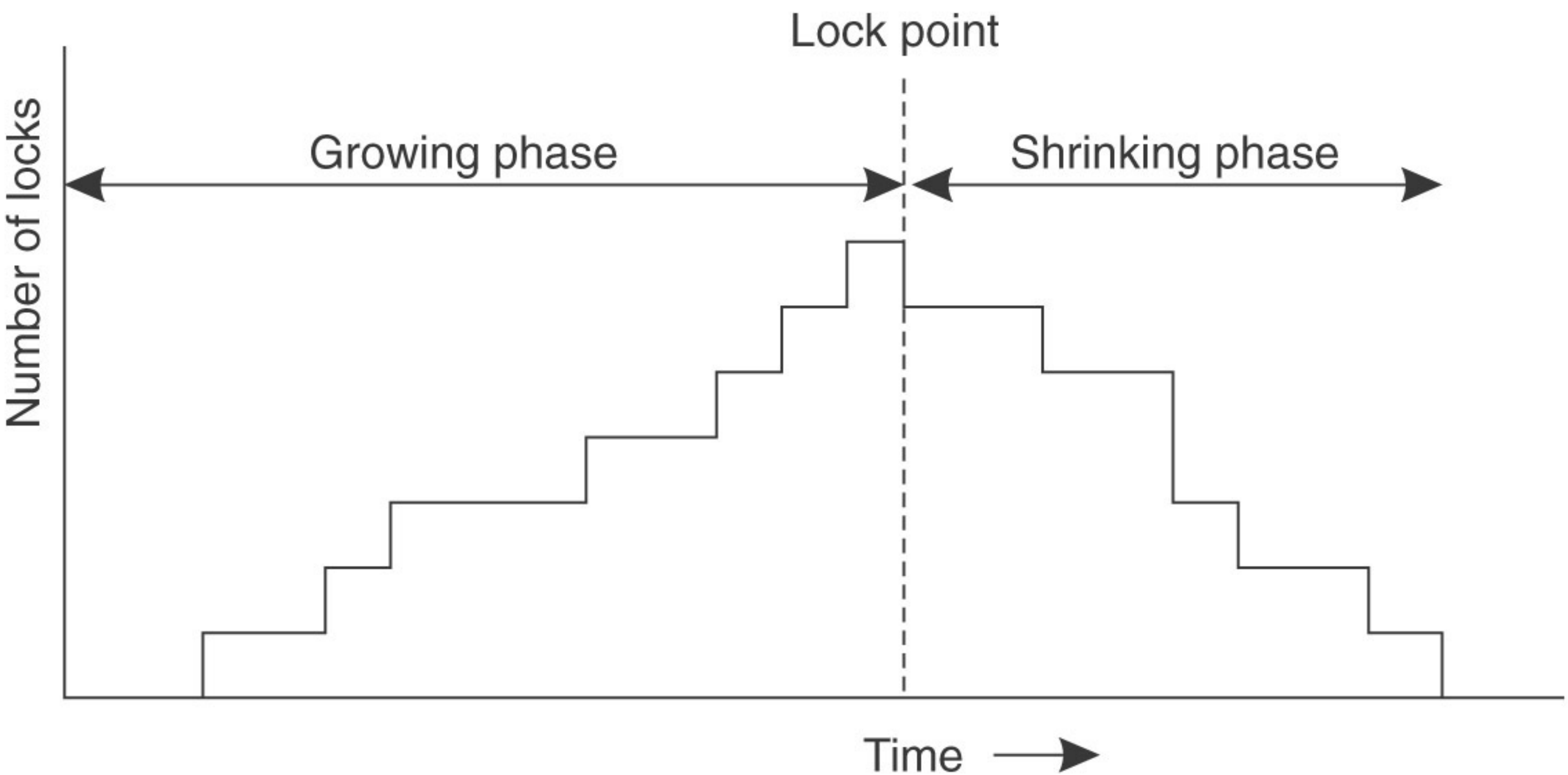
- Locking scheduler: Never get into a bad situation
  - Use locks on data items
    - Shared / Read Locks
    - Exclusive / Write Locks
- Locking problems:
  - Need to make sure that transactions release locks
    - Kill all Zombies!!!!!
  - Need to avoid deadlock
  - Need to avoid life lock

Locking Algorithms

- No restrictions on locking are hard to get right
- ***Two phase locking*** (2PL)
  - All transactions pass first through a phase
    - where they only acquire locks
  - Then through a phase
    - where they only release locks
- A schedule with 2PL is conflict serializable
  - But not every conflict serializable schedule can be created with 2PL

Locking Algorithms

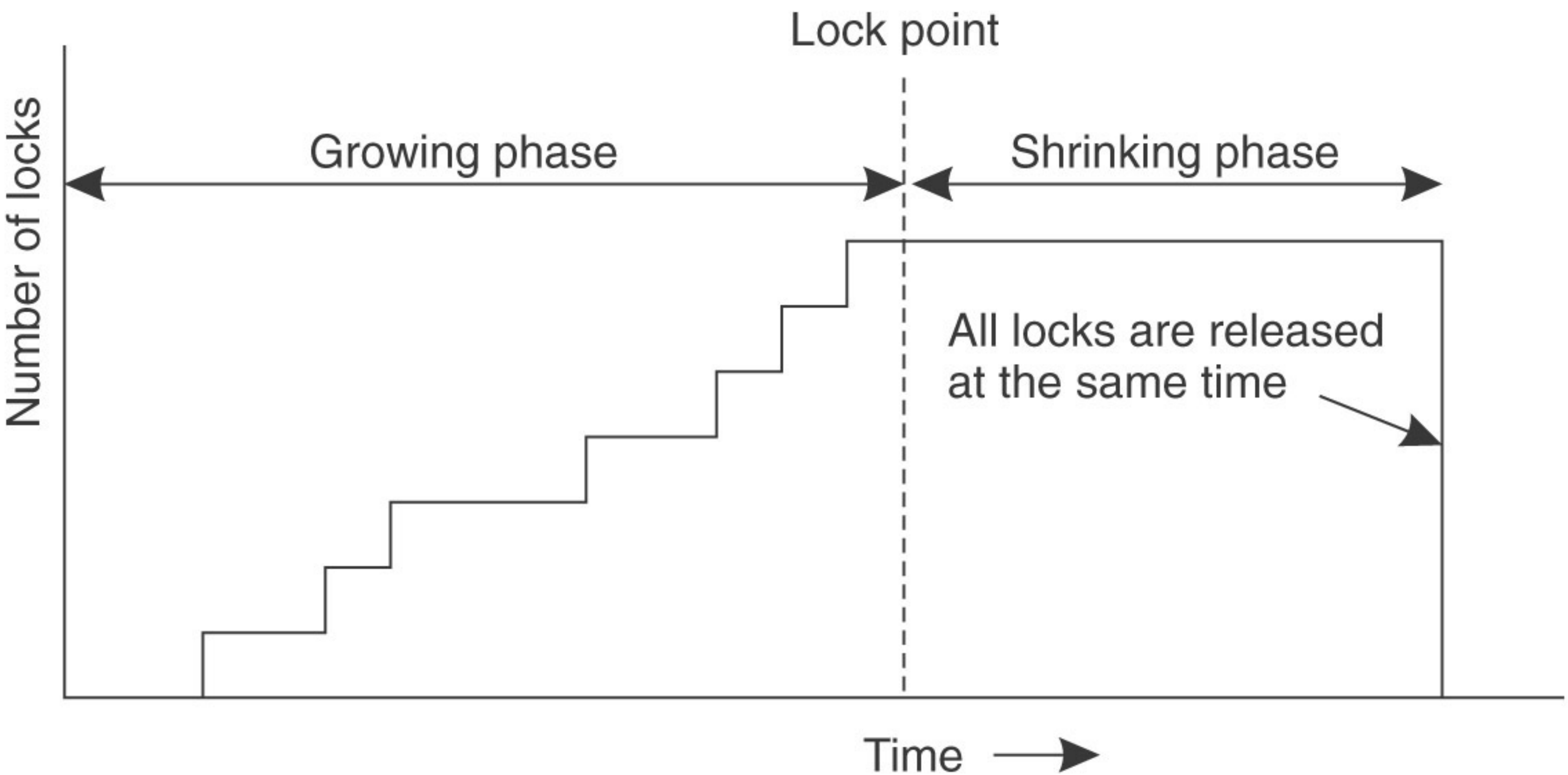
2PL



Locking Algorithms

- 2PL
  - Dirty reads:
    - Allows transactions to read from transaction that are later aborted
- Strict 2PL
  - Release locks only at the end of a transaction
    - Allows easy automatization
      - Application wants to read an item
        - Automatically request lock
      - When committing, automatically release all locks

Locking Algorithms



Locking Algorithms

- Deadlock Handling
  - Normal lock requests can lead to deadlock

$$r_1(x)w_2(y)w_2(x)c_2w_1(y)c_1$$

  - Yields

$$l_1(x)r_1(x)L_2(y)w_2(y)\ldots$$

    - *l* read-lock
    - *L* write-lock

Locking Algorithms

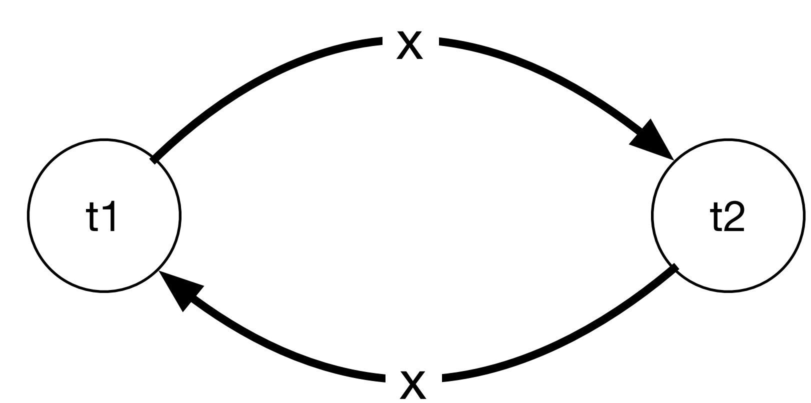
- Deadlock handling
  - Lock conversion can also lead to deadlock

$$
\begin{aligned}
t_1 : \quad & r_1(x)w_1(x) \\
t_2 : \quad & r_2(x)w_2(x) \\
& l_1(x)r_1(x)l_2(x)r_2(x)????
\end{aligned}
$$

Locking Algorithms

- Deadlock Detection
  - Wait for graph
    - Nodes are transactions
    - Edges represent “waiting for”

$$t_1 : \quad r_1(x)w_1(x)$$
$$t_2 : \quad r_2(x)w_2(x)$$
$$l_1(x)r_1(x)l_2(x)r_2(x)????$$



Locking Algorithms

- Deadlock detection
    - Continuous detection
        - WFT is always kept cycle free
    - Periodic detection
        - Check WFT for cycles periodically

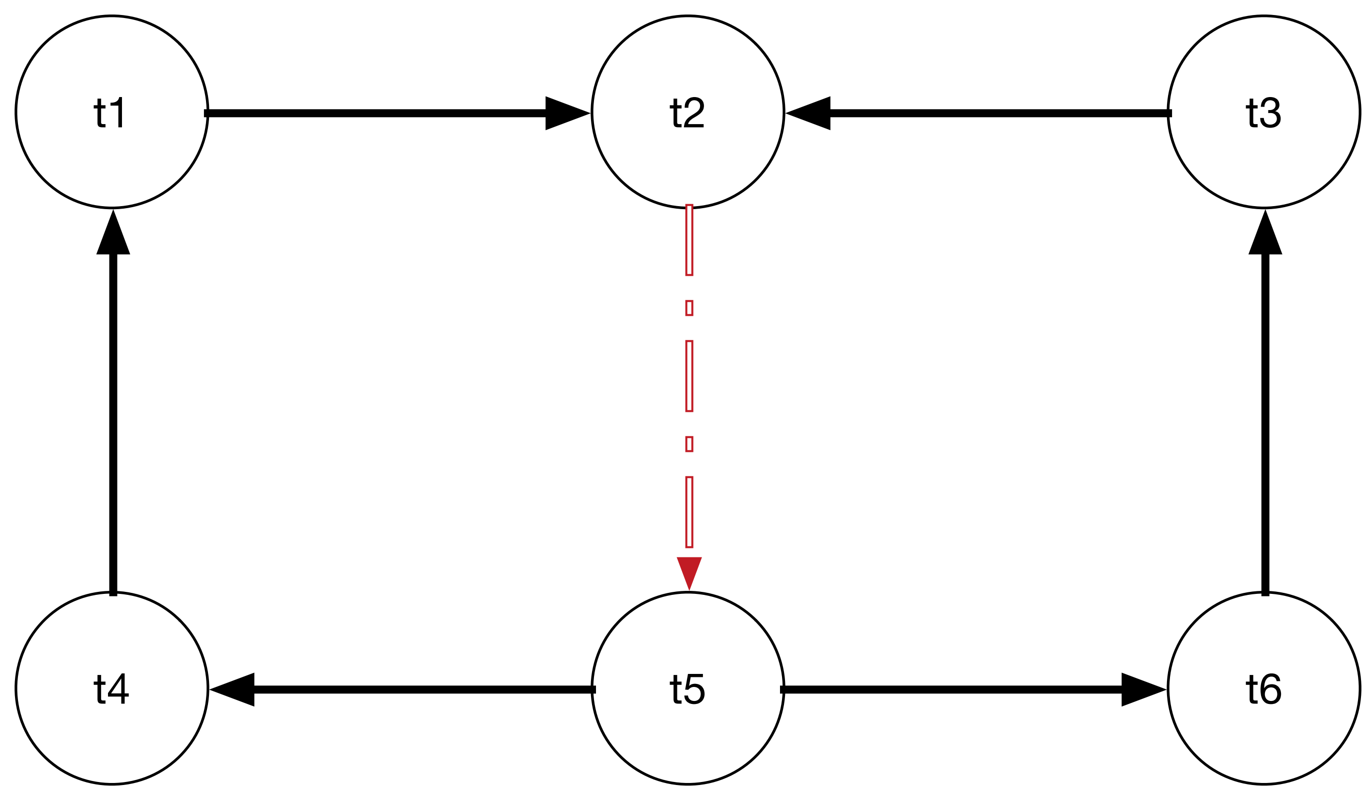
Locking Algorithms

- Deadlock detection
  - If a scheduler detects a cycle:
    - Aborts a *victim* transaction
      - according to heuristics
        - Last blocked
        - Random
        - Youngest
        - Minimum locks
        - Minimum work
        - Most cycles
        - Most edges

Locking Algorithms

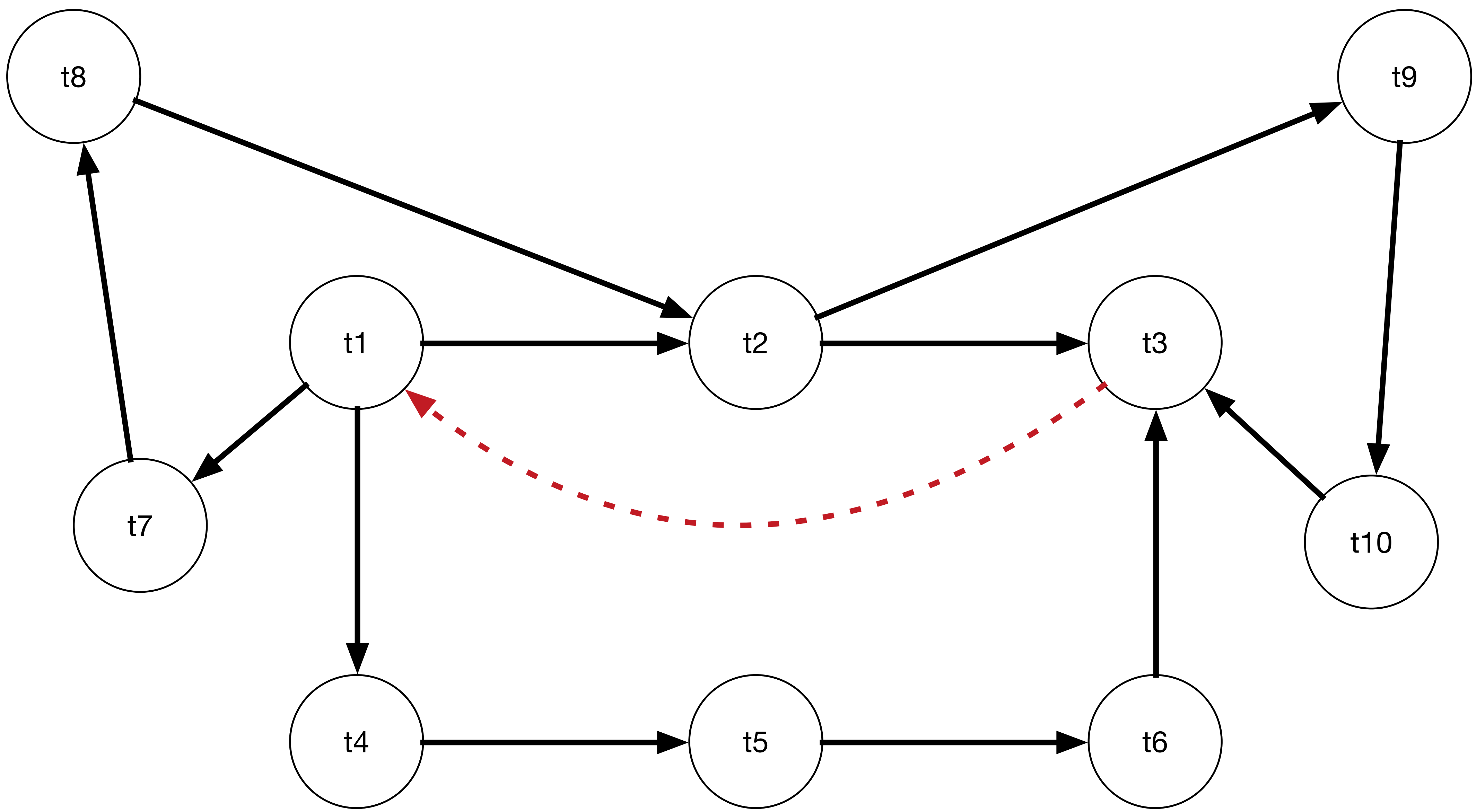
- Life-lock
  - All victim selection mechanism can create live-lock
    - Incarnations of the same transaction are always chosen
    - Various heuristics to avoid life-lock

Locking Algorithms



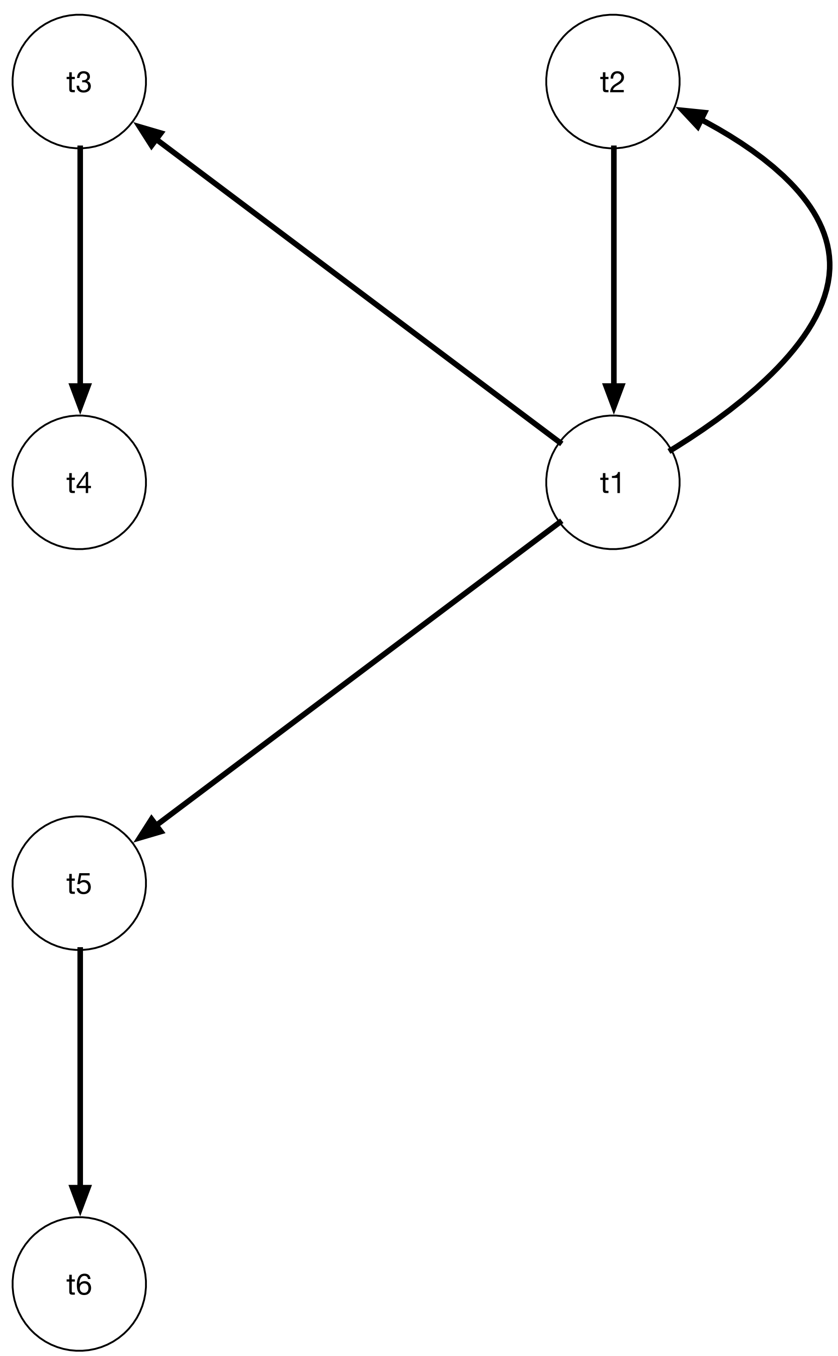
Most cycle heuristics:
Remove dashed edge by aborting
t2 or t5

Locking Algorithms



Dashed cycle breaks the most cycles

Locking Algorithms



WFG with cycle and two candidate victims (t1, t2)

Most edges option:
Abort t1: Two edges remain
Abort t2: Four edges remain

Hence: Abort t1

Locking Algorithms

- Deadlock prevention
  - Abort transaction whose lock request would create a cycle

Non-Locking Algorithms

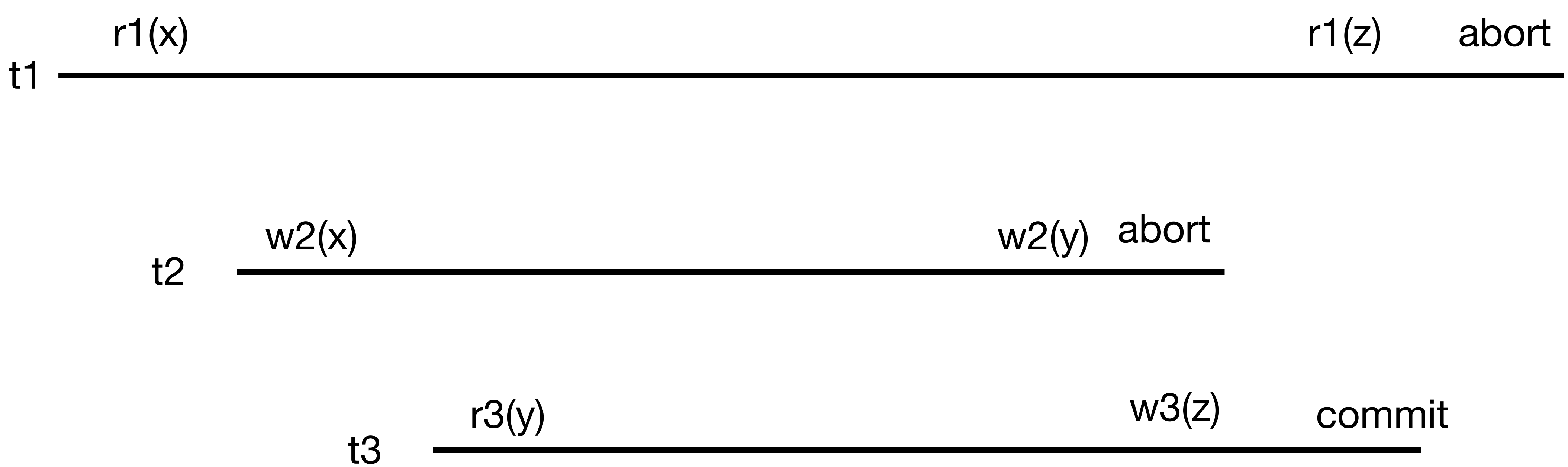
- Timestamp ordering
  - Each transactions gets a unique timestamp
    - Timestamp Ordering rule
      - All operations inherit their timestamp from the transaction
      - If two operations are in conflict, then the one with the smaller timestamp has to be done first
      - If this impossible, abort the offending transaction

Non-Locking Algorithms

- Basic time-stamp ordering (BTO)
  - For all data items, maintain the largest timestamp for
    - a read operation: max-r-scheduled(x)
    - a write operation: max-w-scheduled(x)

Non-Locking Algorithms

- BTO: Transactions can be too late and are aborted
  - w2(x) succeeds since t1 < t2
  - r3(y) succeeds since t3 < t2
  - w2(y) fails since t2<t3, t2 is aborted
  - r1(z) fails since t1 < t3, t1 is aborted



Non-Locking Algorithms

- Optimistic protocols
  - Assume that conflicts are reasonably rare
  - Let transactions go ahead
  - But *validate* their serializability

Non-Locking Algorithms

- Optimistic Protocols
    - Read phase
        - Transaction is executed, but all writes are not committed
            - Write “private items”
    - Validation phase
        - If a transaction is ready to commit, check whether its execution has been correct
    - Write phase
        - Write private items to database

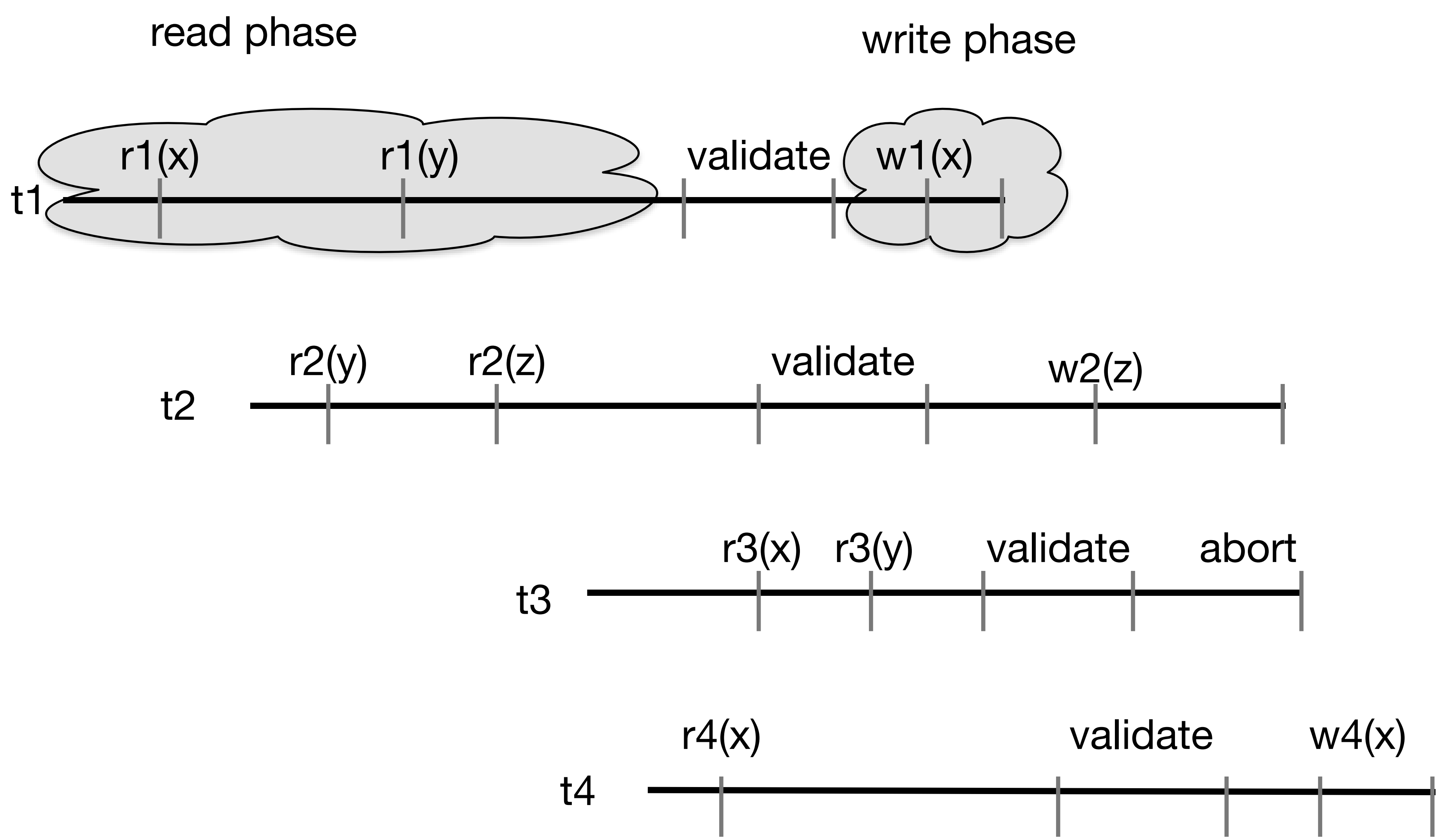
Non-Locking Algorithms

- Backward oriented optimistic concurrency control (BOCC)
  - Validate a transaction against those transactions already committed
- Forward oriented optimistic concurrency control (FOCC)
  - Validate a transaction against those transactions that are in their read phase

Non-Locking Algorithms

- BOCC:
  - The validate-write phase needs to be atomic
  - Transaction *j* is validated if for every committed transaction *i*:
    - *i* has ended before *j* started
  - or
    - The pages touched by *j* have not been written by *i*
      - Thus, *j* had no chance to read from *i*

Non-Locking Algorithms



t3 is aborted because its read set {x,y} overlaps with the write set {x} of t1

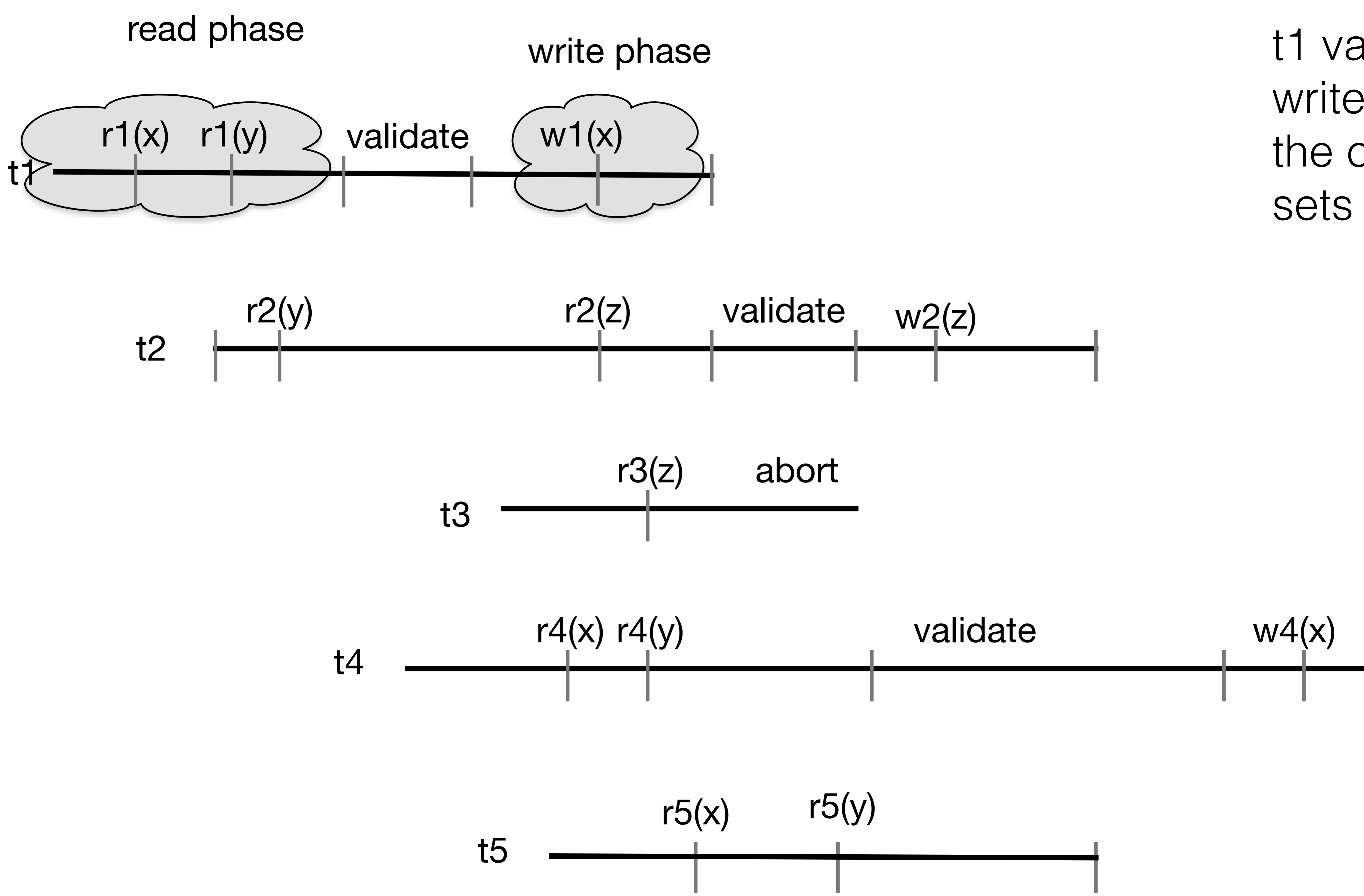
Non-Locking Algorithms

- Schedule
  - … r2(x) … w1(x) …validate1… validate2 …
  - t2 will get aborted as its read set overlaps with the write set of t1.
  - This is clear once t1 writes
- Therefore Forward-oriented optimistic Concurrency Control (FOCC

Non-Locking Algorithms

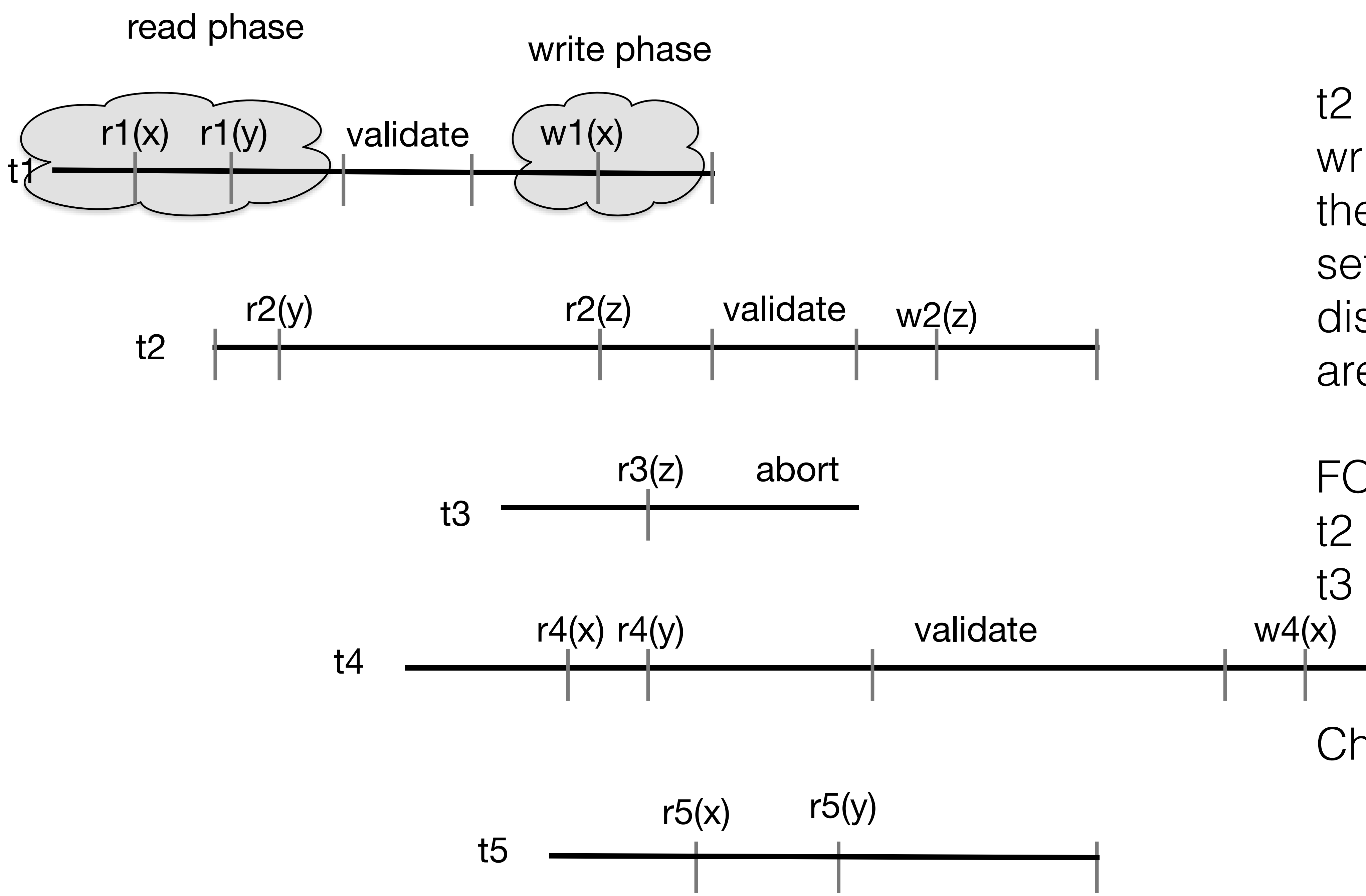
- Forward-oriented concurrency control
  - Accept a transaction tj if for all transactions ti that are currently reading
    - Writeset(tj) is disjoint from Readset(ti) at current time
- Example: FOCC accepts all read-only transactions

Non-Locking Algorithms



t1 validates its write set {x} against the current read sets {y}

Non-Locking Algorithms

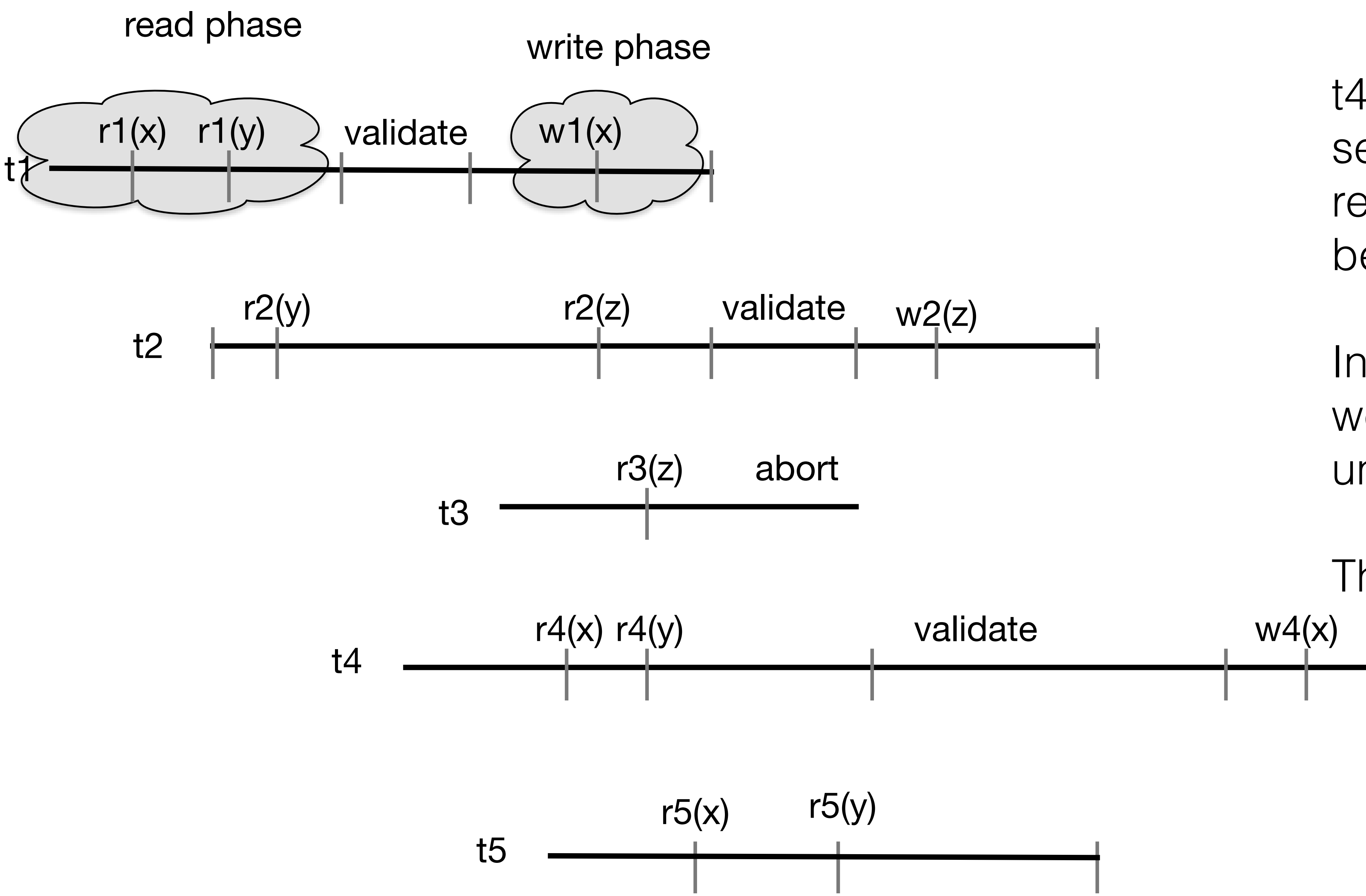


t2 validates its write set {z} against the current read sets {z, x, y} and discovers that they are not disjoint.

FOCC allows:
t2 could abort
t3 could abort

Chooses to abort t3

Non-Locking Algorithms



t4 validates its write set against current read sets {x, y, z} because of t5.

Instead of aborting, we can have t4 wait until t4 terminates

Then t4 validates

Multi-version Concurrency Control

- Allow a single value to have multiple versions
- Multi-version schedule
  - Note values read
  - Example
    - r1(x0)w1(x1)r2(x1)w2(y2)r1(y0)w1(z1)c1c2
      - Transaction 1 reads an earlier version than the value written by transaction 2

Multi-version Concurrency Control

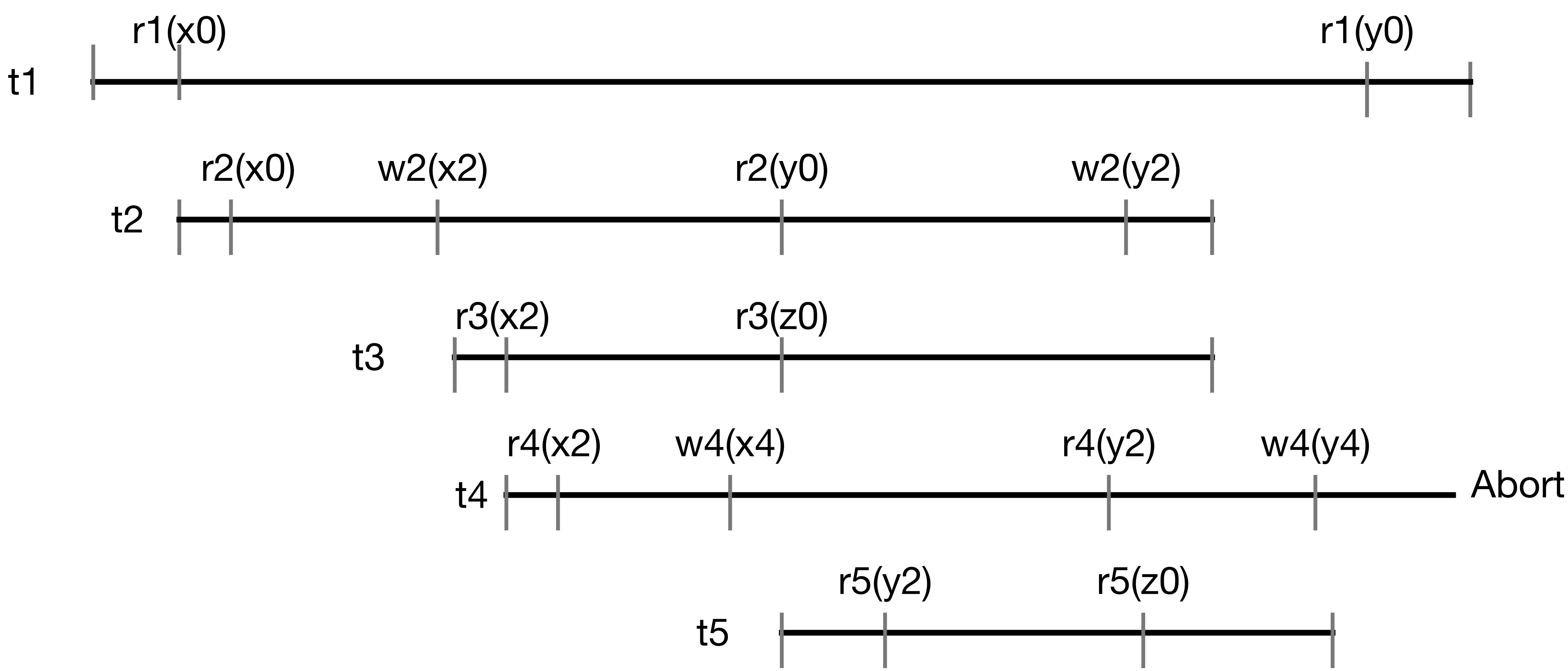
- Multi-version timestamp ordering (MVTO)
  - Each version carries the timestamp of the transaction that created it
  - Each read reads the last version that was written before the timestamp of the transaction
  - Writes:
    - wi(x)
      - If there was a read rj(xk) with
        - time(tk)<time(ti)<time(tj)
      - abort ti
      - Otherwise, write x with timestamp ts(tk)

Multi-version Concurrency Control

- In order to avoid dirty reads:
  - Delay commits until all other transactions that have written a new version of what we read have finished

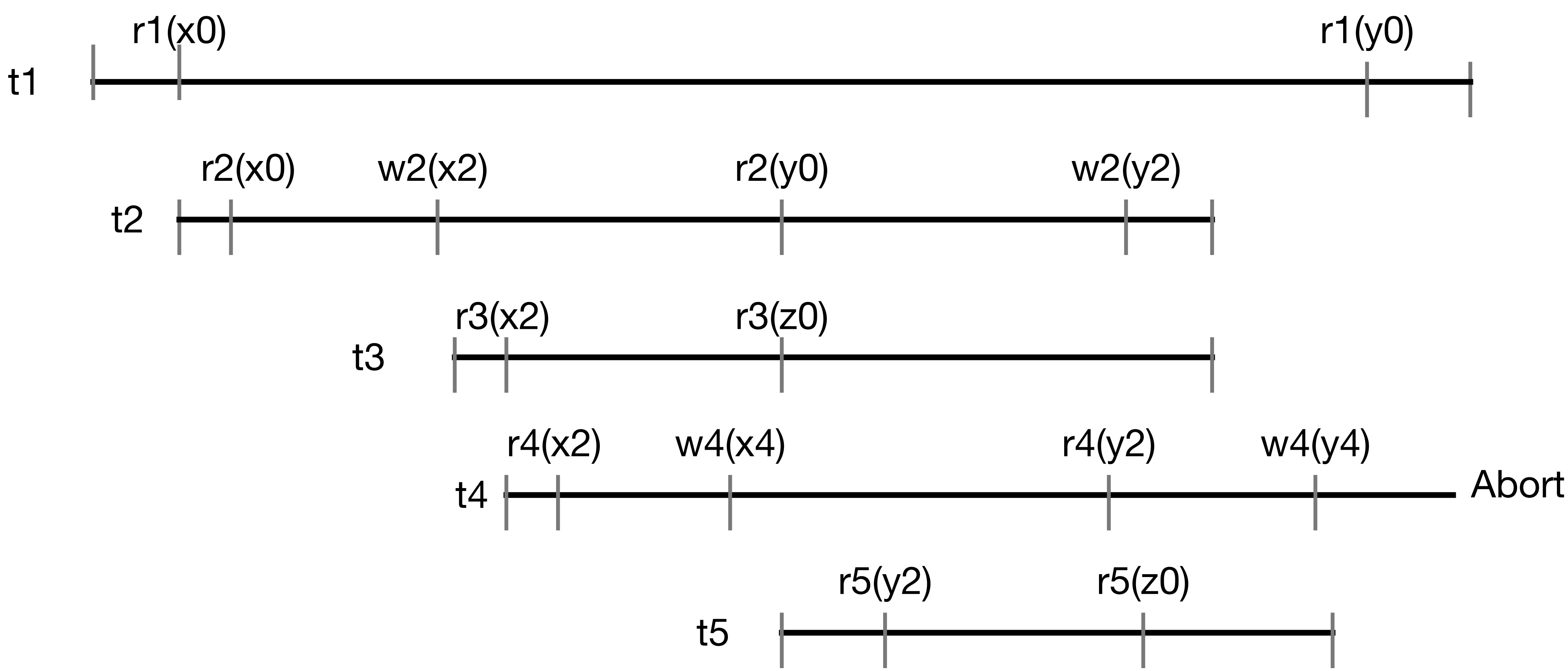
Multi-version Concurrency Control

t1 and t2 are interleaved



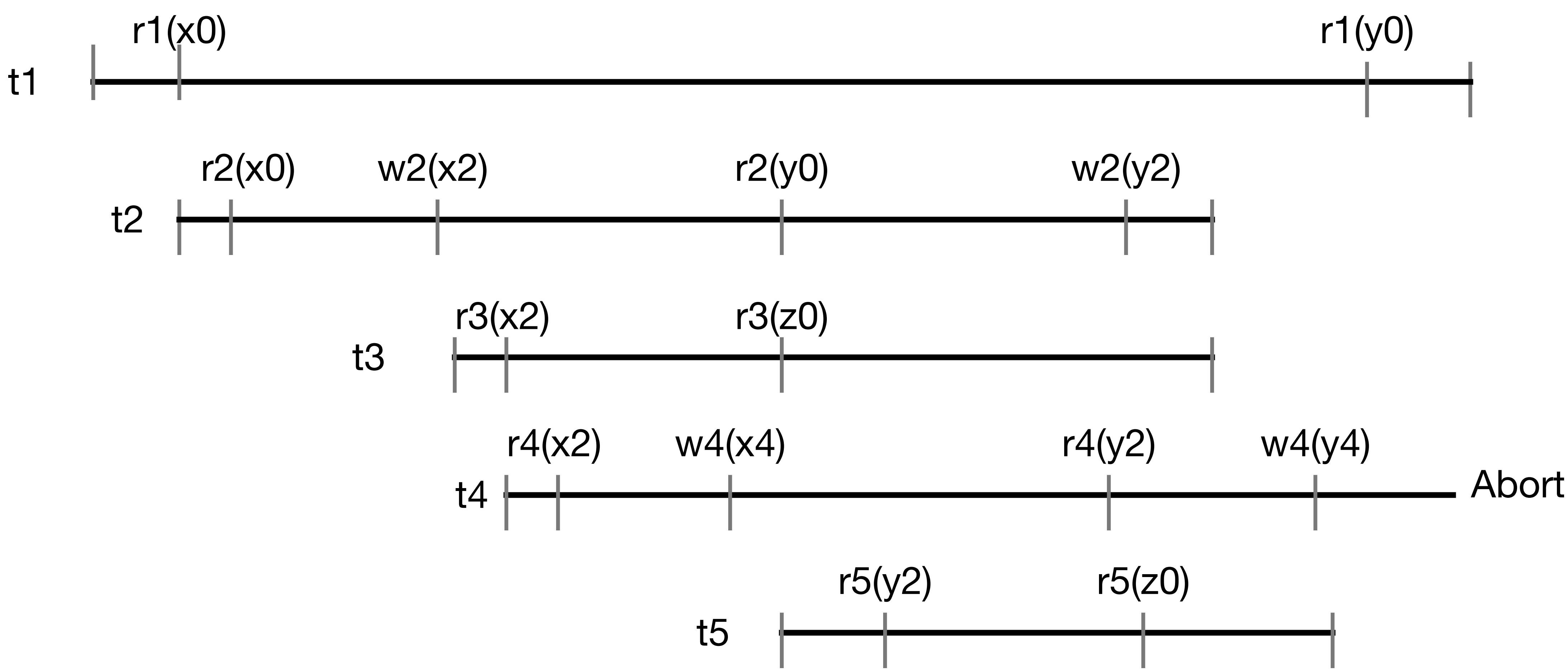
Multi-version Concurrency Control

t3 needs to wait until t2 terminates, because it read x2
Since its timestamp is larger, it has to read this value instead of x0



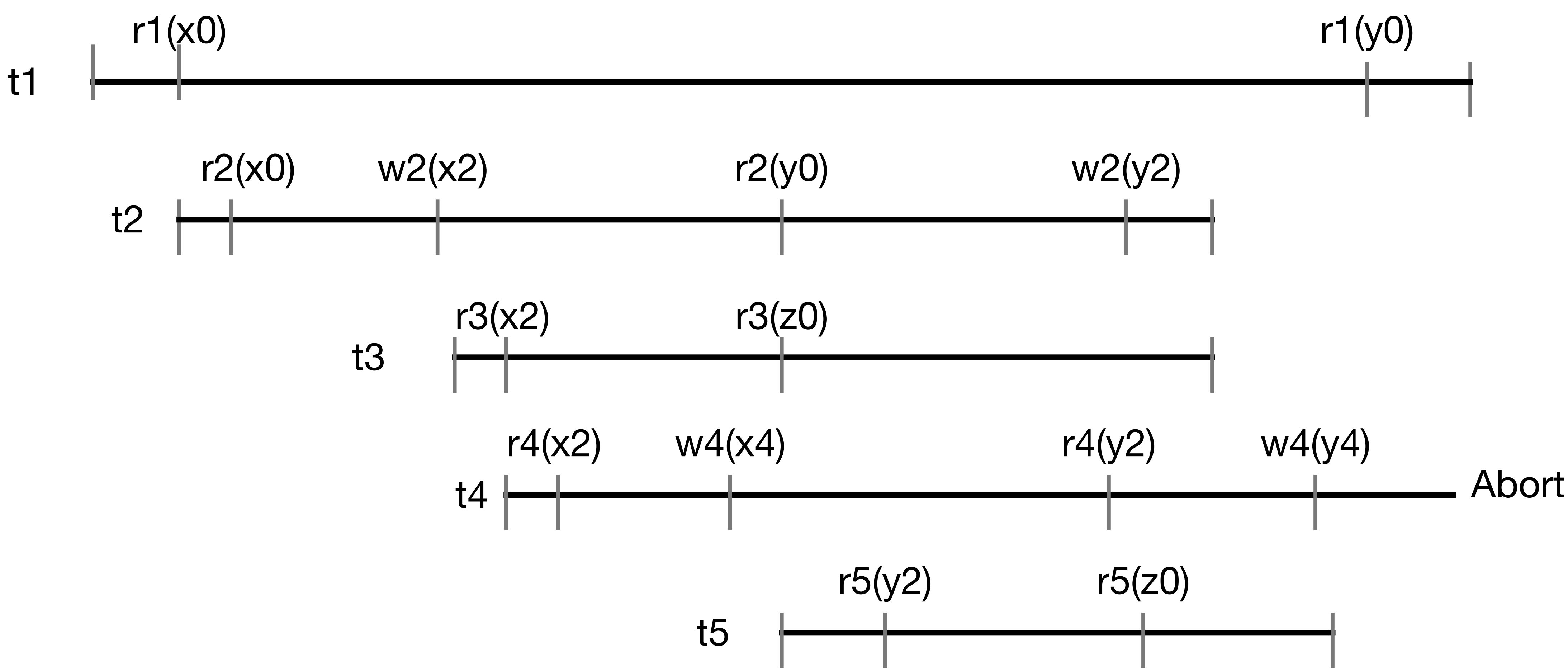
Multi-version Concurrency Control

t4 is a “late writer”. t5 (with a later timestamp) has already read y2, and t4 can no longer change it. So, t4 needs to be aborted.



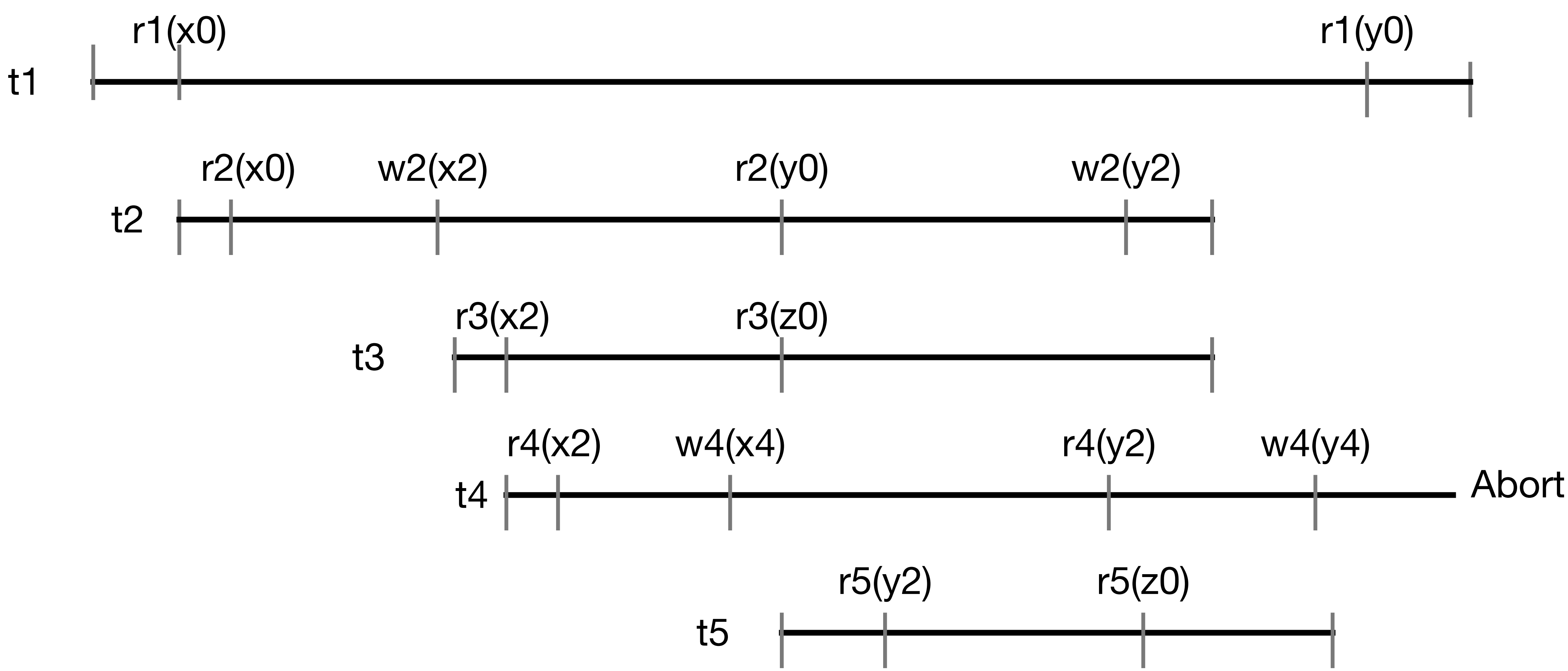
Multi-version Concurrency Control

t1 and t2 are interleaved



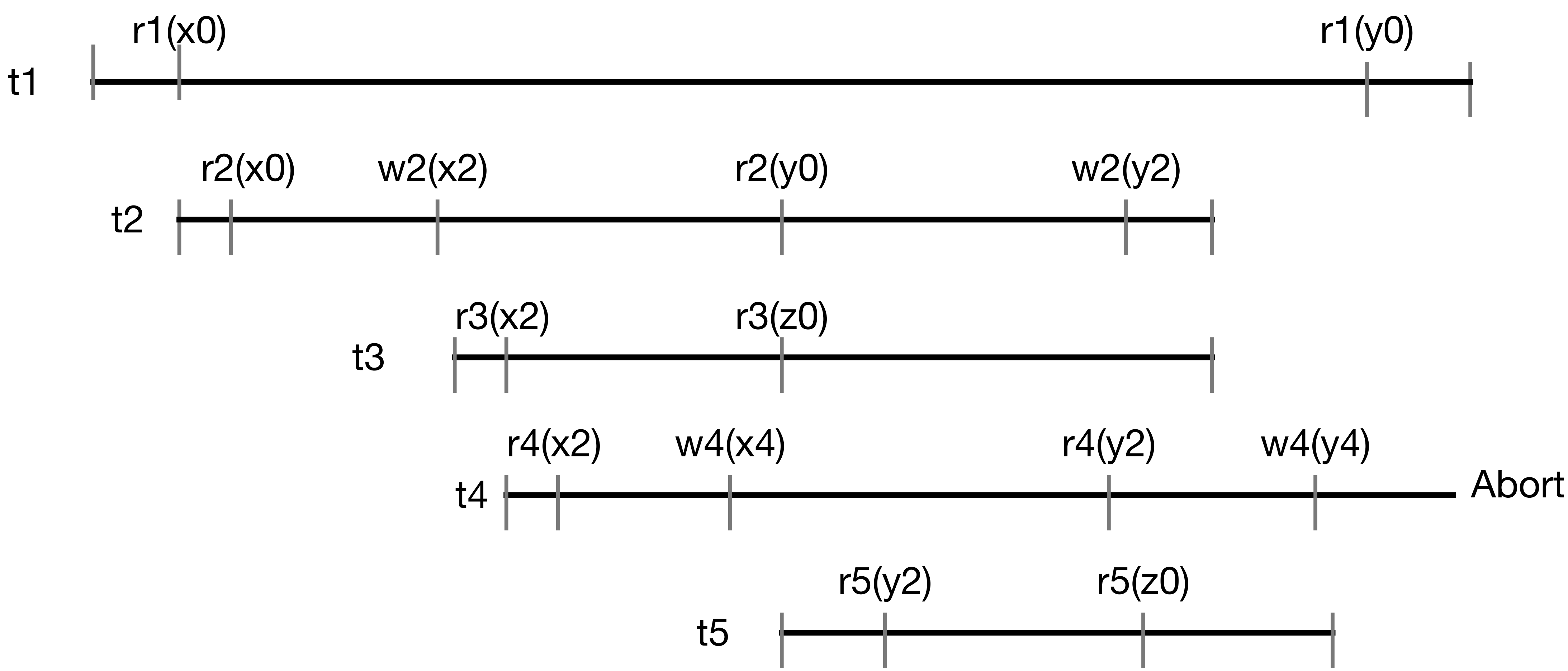
Multi-version Concurrency Control

t1 and t2 are interleaved



Multi-version Concurrency Control

t1 and t2 are interleaved



Write-Ahead Logging

- A family of protocols that use a log
  - Recovery after a crash
  - Aborting transactions
- Each site writes the operation it is about to perform on a page to the write-ahead log

```
x = 0;
y = 0;
BEGIN_TRANSACTION;
    x = x + 1;
    y = y + 2;
    x = y * y;
END_TRANSACTION;
```

(a)

Log	Log	Log
[x = 0/1]	[x = 0/1]	[x = 0/1]
	[y = 0/2]	[y = 0/2]
		[x = 1/4]
(b)	(c)	(d)

Write-Ahead Logging

- ARIES
  - Write-ahead log
  - Repeating history
    - After crash, retrace the actions to bring database up to the moment of crash
    - Undo transactions that were then pending
  - Logging Undo operations
    - Log undo operations in order to avoid repeating actions after repeated crashes

Write-Ahead Logging

- ARIES
  - Dirty Page Table (DPT)
  - Transaction Table (TT)
  - Log
    - Sequence Number, Transaction ID, Page ID, Redo, Undo, Previous Sequence Number
      - Redo and Undo: Information to redo and undo a transaction

Write-Ahead Logging

- ARIES
  - Analysis
    - Calculate the necessary information from the log
      - From last checkpoint:
        - Add all transactions started to TT
        - Remove transactions in the TT when finding a END LOG statement
        - Update the dirty pages table

Write-Ahead Logging

- ARIES
  - Redo
    - Use the DPT to calculate the minimal sequence number of a dirty page
    - From this sequence number, redo operations for pages in the DPT

Write-Ahead Logging

- ARIES
  - Undo
    - Undo the changes of uncommitted transactions
    - Run backward through the log
      - For each transaction in the TT
        - Undo the change for each touched page
      - Write the changes in a compensation log
      - (In case of a crash during recovery)

Distributed Transactions

- Issues:
  - Distributed decision making
    - by leader: Leader election
    - Commit protocols
  - Distributed locks:
    - Deadlock detection
    - Deadlock avoidance
  - Distributed checkpoints

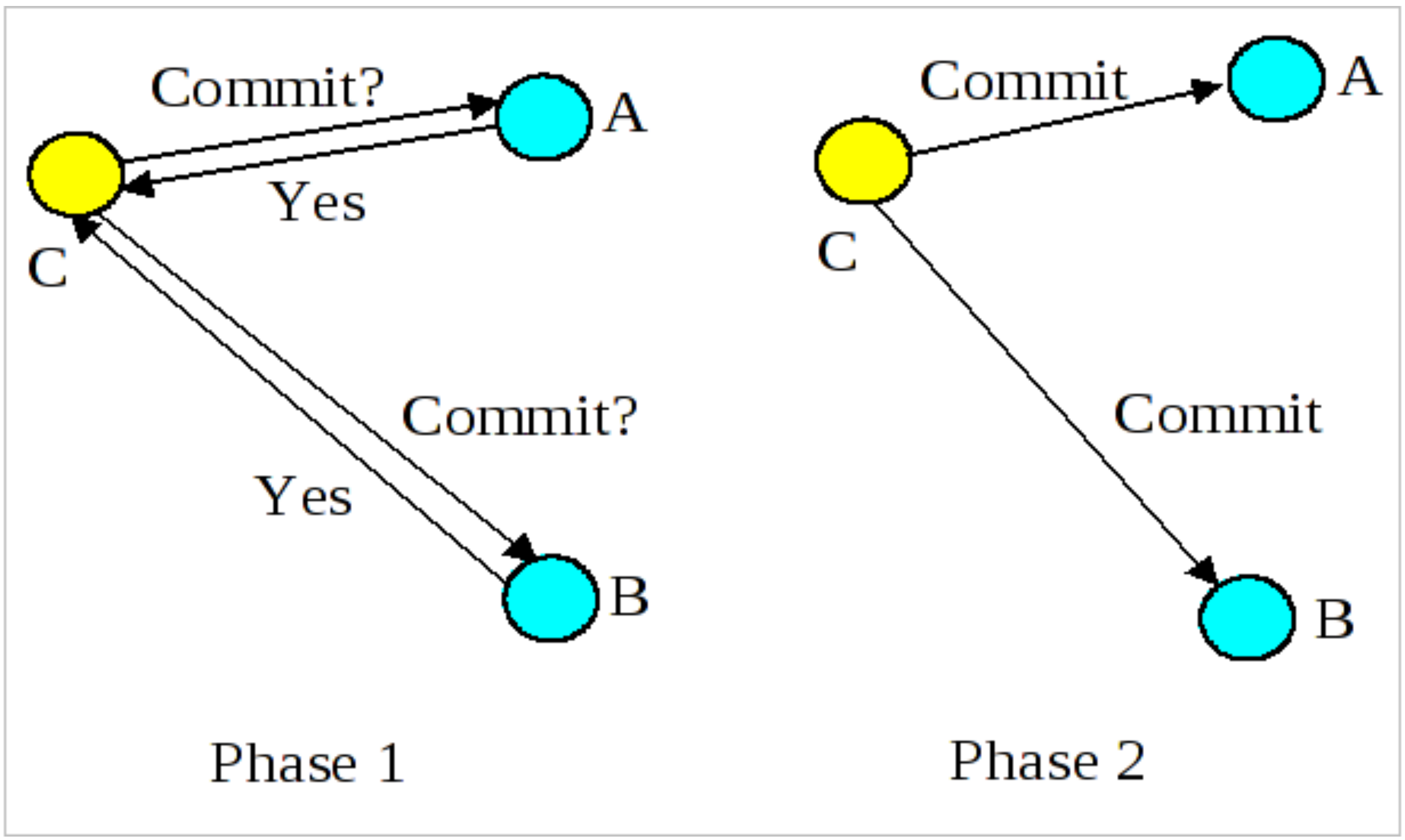
Distributed Transactions

- Distributed commit
  - All members of a group need to perform an action or none
  - One phase commit:
    - Single coordinator
      - Sends a “commit” or a “not-commit” message
    - Has no feedback from participants
    - Cannot be used in practice

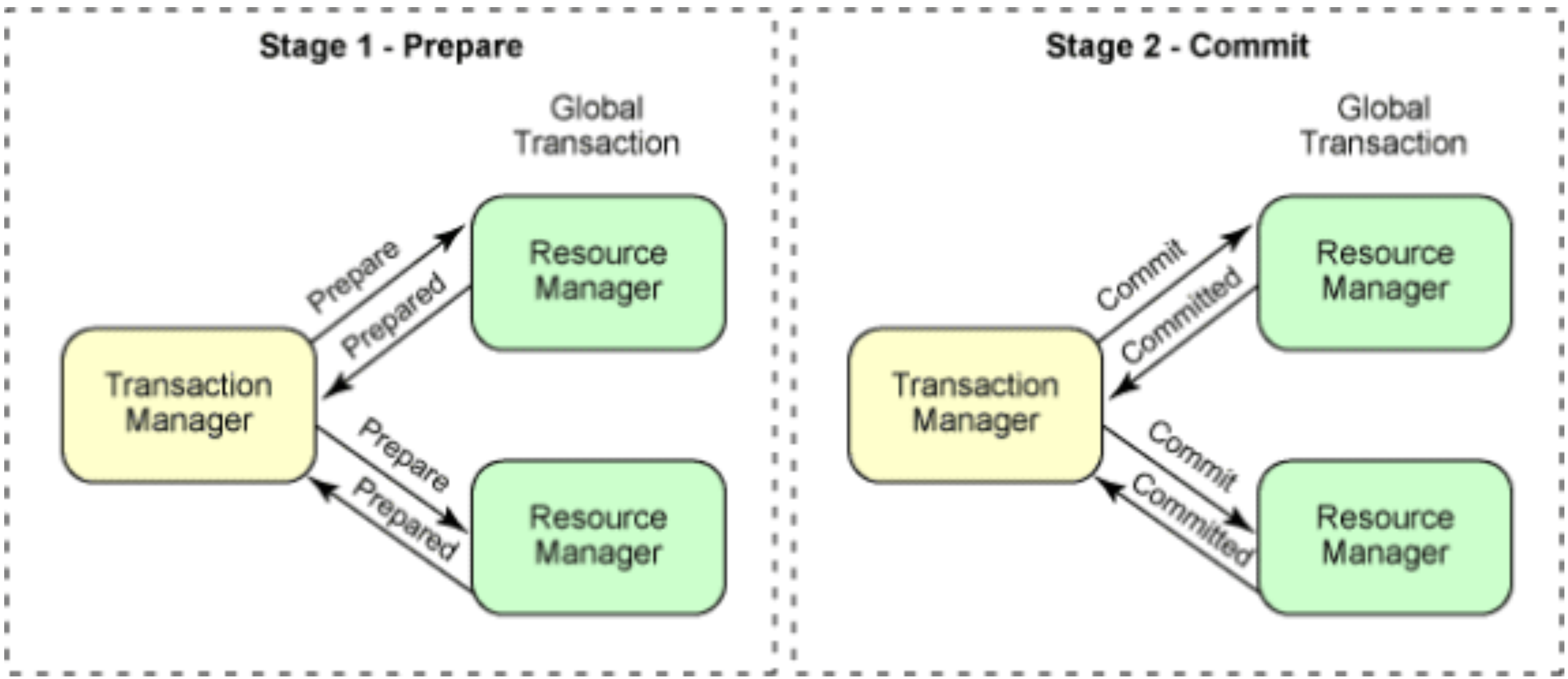
Distributed Transactions

- Two phase commit (Jim Gray)
  - Phase 1:
    - Coordinator sends vote request
    - Participants vote on whether they want to commit
  - Phase 2:
    - Coordinator decides vote
      - Single no is a veto
    - Coordinator sends participants message

Distributed Transactions



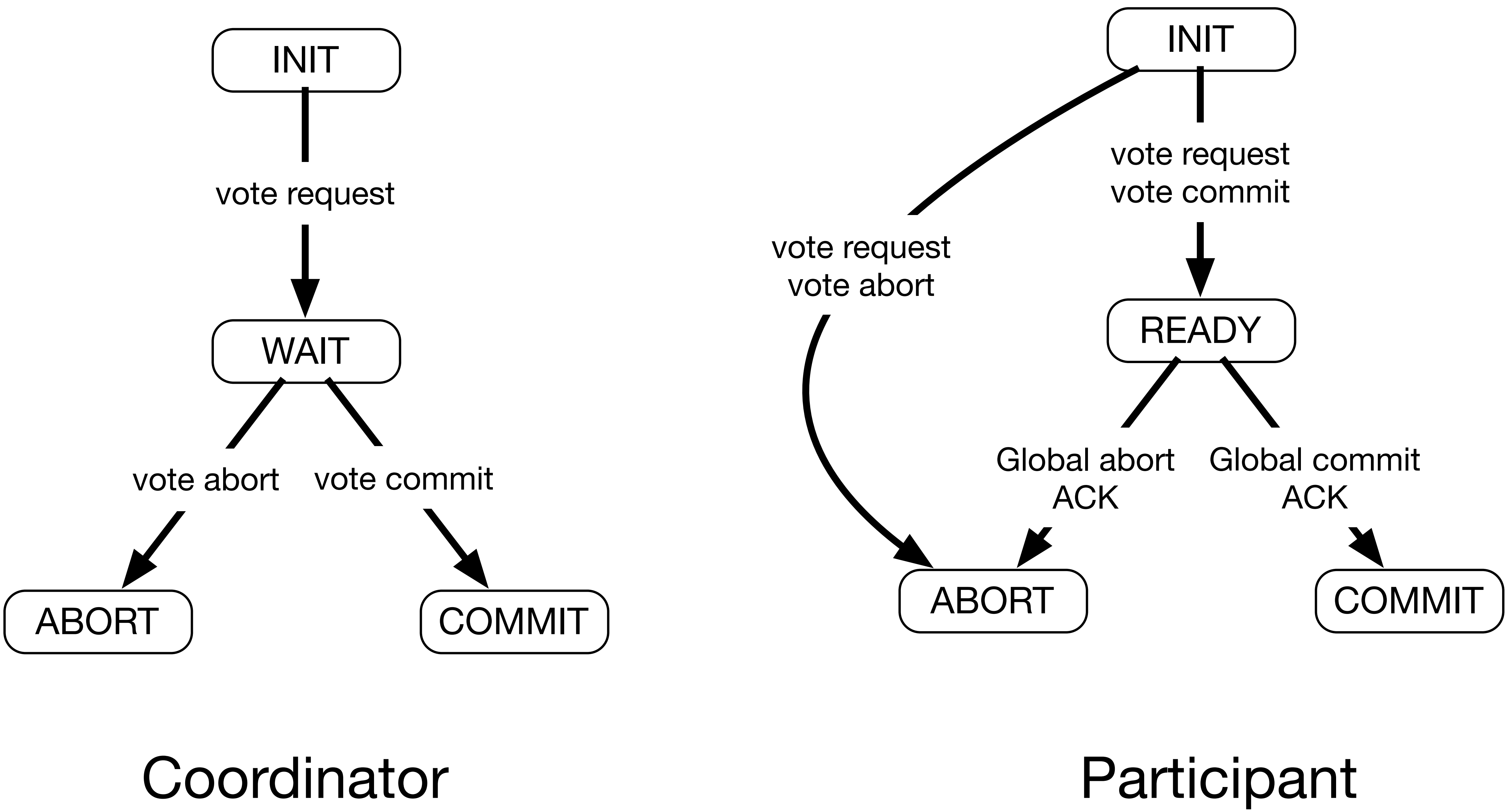
Distributed Transactions



Distributed Transactions

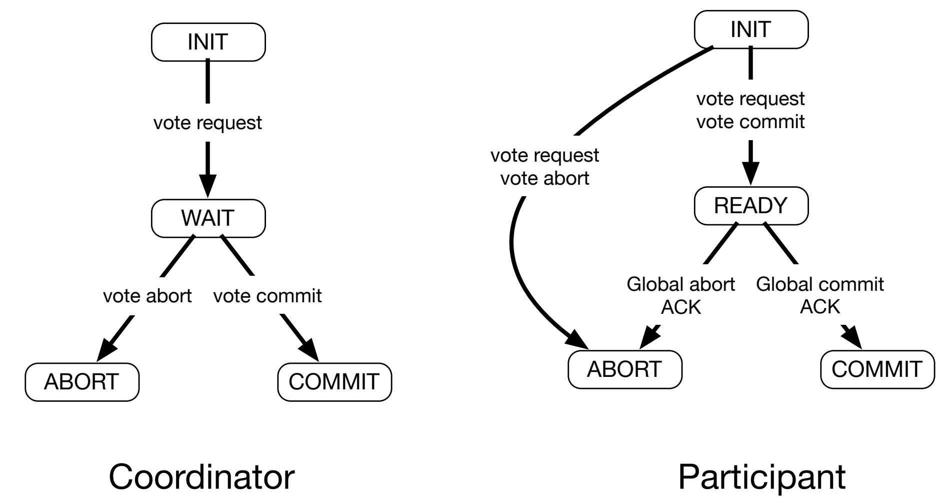
- 2PC can have problems with failures
  - Failure of coordinator or participants leads to blocking

Distributed Transactions



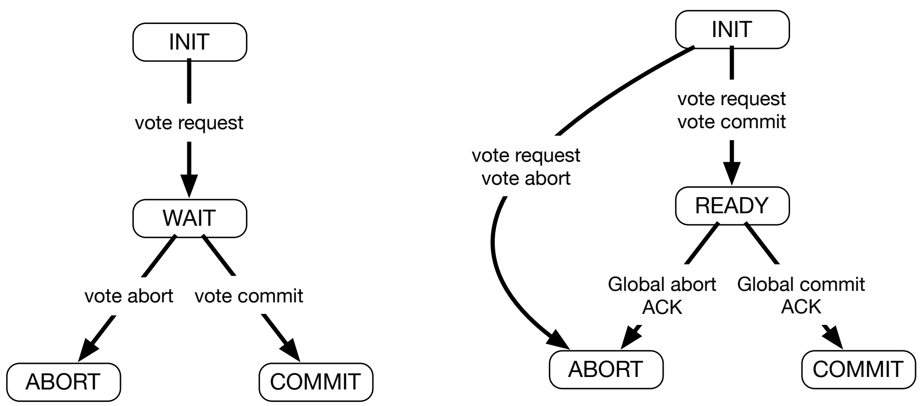
Distributed Transactions

- Participant in INIT waiting for request to vote
  - Can locally abort transaction



Distributed Transactions

- Coordinator in WAIT waiting for answers
- Can send Global_Abort

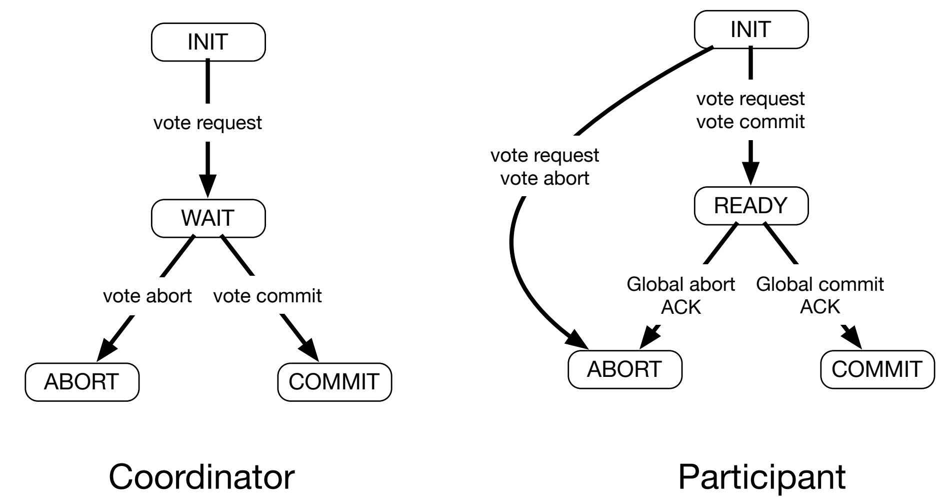


Coordinator

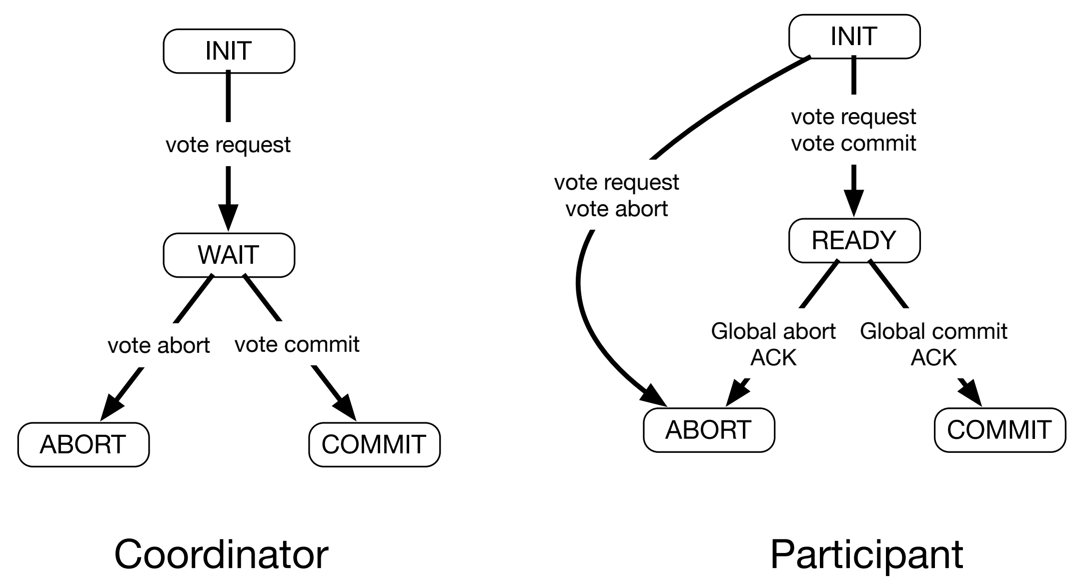
Participant

Distributed Transactions

- Participant in READY waiting for coordinator
- Cannot decide
  - Block until coordinator recovers
  - Or talk to other participant
  - If all participants are in state READY, no decision can be taken



Distributed Transactions



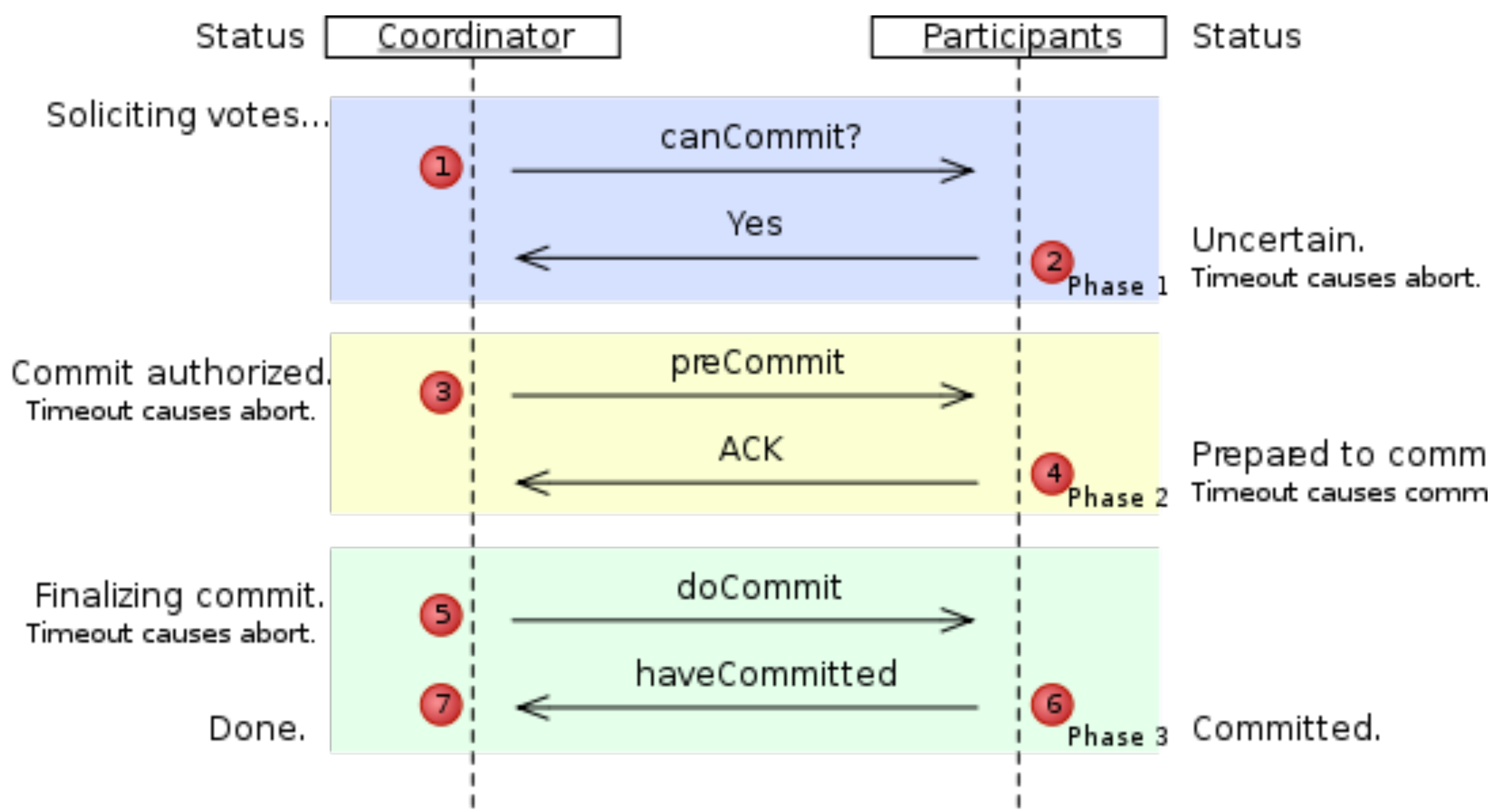
State of Q	Action by P
COMMIT	Make transition to COMMIT
ABORT	Make transition to ABORT
INIT	Make transition to ABORT
READY	Contact another participant

Distributed Transactions

- Non-blocking solution
  - Use multicast primitive
    - Receiver immediately multicasts received message to all participants

Distributed Transactions

- Three Phase Commit
  - Crashed coordinator might leave participants hanging Solution is to add another phase



Distributed Time Order

- TO rule:
  - If pi(x) and qj(x) are operations in conflict then
    $$p_i(x) \text{ is executed before } q_j(x) \iff ts(t_i) < ts(t_j)$$
  - Use Lamport clock to generate time stamps

$$\begin{array}{llll} \text{Server 1}: & r_1(x) & w_2(x) & \ldots \\ \text{Server 2}: & r_2(y) & w_1(y) & \ldots \end{array}$$

  - Both transactions have local timestamp 1 but are ordered so that 2nd transaction aborts

Distributed Transactions

- Locking protocols
  - Need to reach global decision when to release a lock
  - Primary 2PL:
    - All locking is done at a *primary* site
  - Distributed 2PL
    - Strict 2PL with commit releases locks

Distributed Transactions

- Optimistic protocols
  - Protect validation / write phase by using 2PC or 3PC

Distributed Transactions

- Distributed deadlock handling
    - Detection
        - Time-outs
        - Edge chasing
            - Blocked transaction sends out a probe to all transactions it is waiting for
            - Those forward probe
            - When probe returns to sending transaction:
                - Deadlock exist
                - Path(s) of returned probe(s) indicates which transaction to abort
        - Path pushing
            - Collect local “waits-for” graphs at a single server