Outline

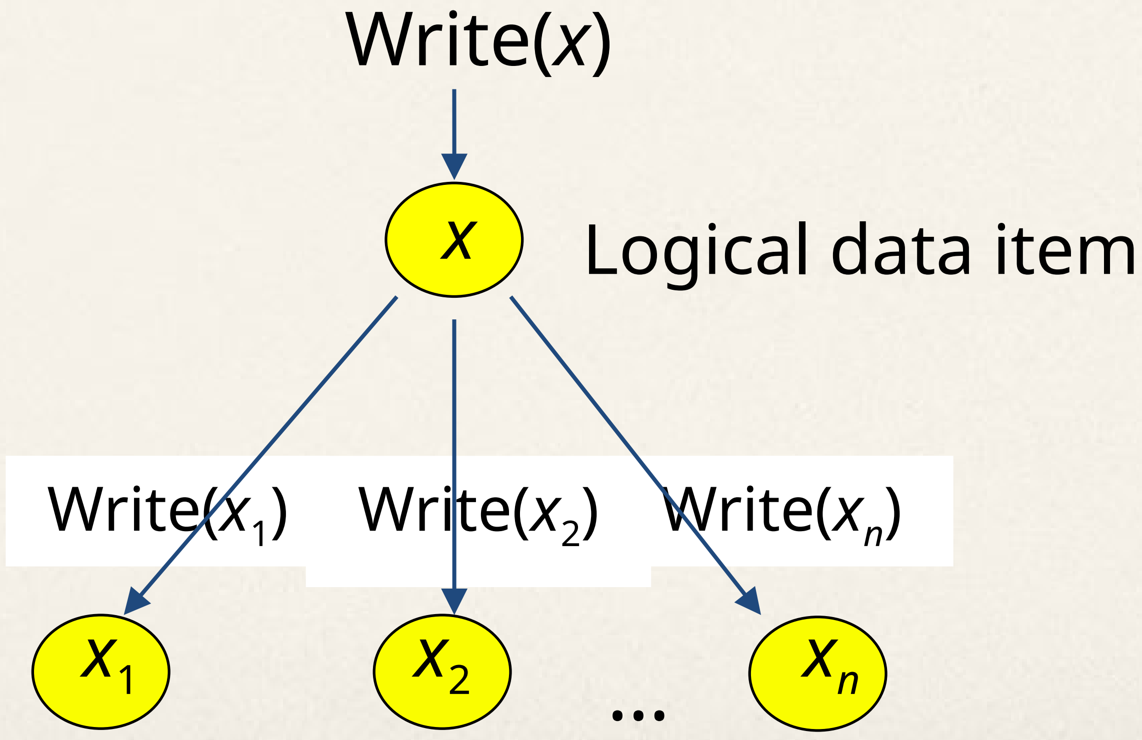
- Introduction
- Background
- Distributed Database Design
- Database Integration
- Semantic Data Control
- Distributed Query Processing
- Distributed Transaction Management
- Data Replication
  - ➡️Consistency criteria
  - ➡️Replication protocols
  - ➡️Replication and failure management
- Parallel Database Systems
- Distributed Object DBMS
- Peer-to-Peer Data Management
- Web Data Management
- Current Issues

Replication

- Why replicate?
  - ➡️System availability
    - ✦ Avoid single points of failure
  - ➡️Performance
    - ✦ Localization
  - ➡️Scalability
    - ✦ Scalability in numbers and geographic area
  - ➡️Application requirements
- Why not replicate?
  - ➡️Replication transparency
  - ➡️Consistency issues
    - ✦ Updates are costly
    - ✦ Availability may suffer if not careful

Execution Model

- There are physical copies of logical objects in the system.
- Operations are specified on logical objects, but translated to operate on physical objects.
- One-copy equivalence
  - The effect of transactions performed by clients on replicated objects should be the same as if they had been performed on a single set of objects.



Write(x)

x Logical data item

Write(x_1) Write(x_2) Write(x_n)

x_1 x_2 … x_n

Physical data item (replicas, copies)

Replication Issues

- Consistency models - how do we reason about the consistency of the “global execution state”?
    - Mutual consistency
    - Transactional consistency
- Where are updates allowed?
    - Centralized
    - Distributed
- Update propagation techniques – how do we propagate updates to one copy to the other copies?
    - Eager
    - Lazy

Consistency

- Mutual Consistency
  - ➡️ How do we keep the values of physical copies of a logical data item synchronized?
  - ➡️ Strong consistency
    - ✦ All copies are updated within the context of the update transaction
    - ✦ When the update transaction completes, all copies have the same value
    - ✦ Typically achieved through 2PC
  - ➡️ Weak consistency
    - ✦ Eventual consistency: the copies are not identical when update transaction completes, but they eventually converge to the same value
    - ✦ Many versions possible:
      - ✓ Time-bounds
      - ✓ Value-bounds
      - ✓ Drifts

Transactional Consistency

- How can we guarantee that the global execution history over replicated data is serializable?
- One-copy serializability (1SR)
  - ➡️The effect of transactions performed by clients on replicated objects should be the same as if they had been performed *one at-a-time* on a single set of objects.
- Weaker forms are possible
  - ➡️Snapshot isolation
  - ➡️RC-serializability

Example 1

(Mutual Consistency versus Transaction Consistency)

	Site A	Site B	Site C
	x	x, y	x, y, z
	T_1: $x \leftarrow 20$ Write(x) Commit	T_2: Read(x) $y \leftarrow x+y$ Write(y) Commit	T_3: Read(x) Read(y) $z \leftarrow (x \ast y)/100$ Write(z) Commit

Consider the three histories:

$H_A=\{W_1(x_A), C_1\}$

$H_B=\{W_1(x_B), C_1, R_2(x_B), W_2(y_B), C_2\}$

$H_C=\{W_2(y_C), C_2, \mathrm{R}_3(x_C), R_3(y_C), W_3(z_C), C_3, W_1(x_C), C_1\}$

Global history non-serializable: H_B: $T_1 \rightarrow T_2$, H_C: $T_2 \rightarrow T_3 \rightarrow T_1$

Mutually consistent: Assume $x_A=x_B=x_C=10$, $y_B=y_C=15$, $z_C=7$ to begin; in the end $x_A=x_B=x_C=20$, $y_B=y_C=35$, $z_C=3.5$

Example 2

(Mutually inconsistent, and globally non-serializable)

Site A	Site B
x	x

T_1:	Read(x)	T_2:	Read(x)
	$x \leftarrow x+5$		$x \leftarrow x \ast 10$
	Write(x)		Write(x)
	Commit		Commit

Consider the two histories:

$H_A=\{R_1(x_A), W_1(x_A), C_1, W_2(x_A), C_2\}$

$H_B=\{R_2(x_B), W_2(x_B), C_2, W_1(x_B), C_1\}$

Global history non-serializable: H_A: $T_1 \rightarrow T_2$, H_B: $T_2 \rightarrow T_1$

Mutually inconsistent: Assume $x_A=x_B=1$ to begin; in the end $x_A=10$, $x_B=6$

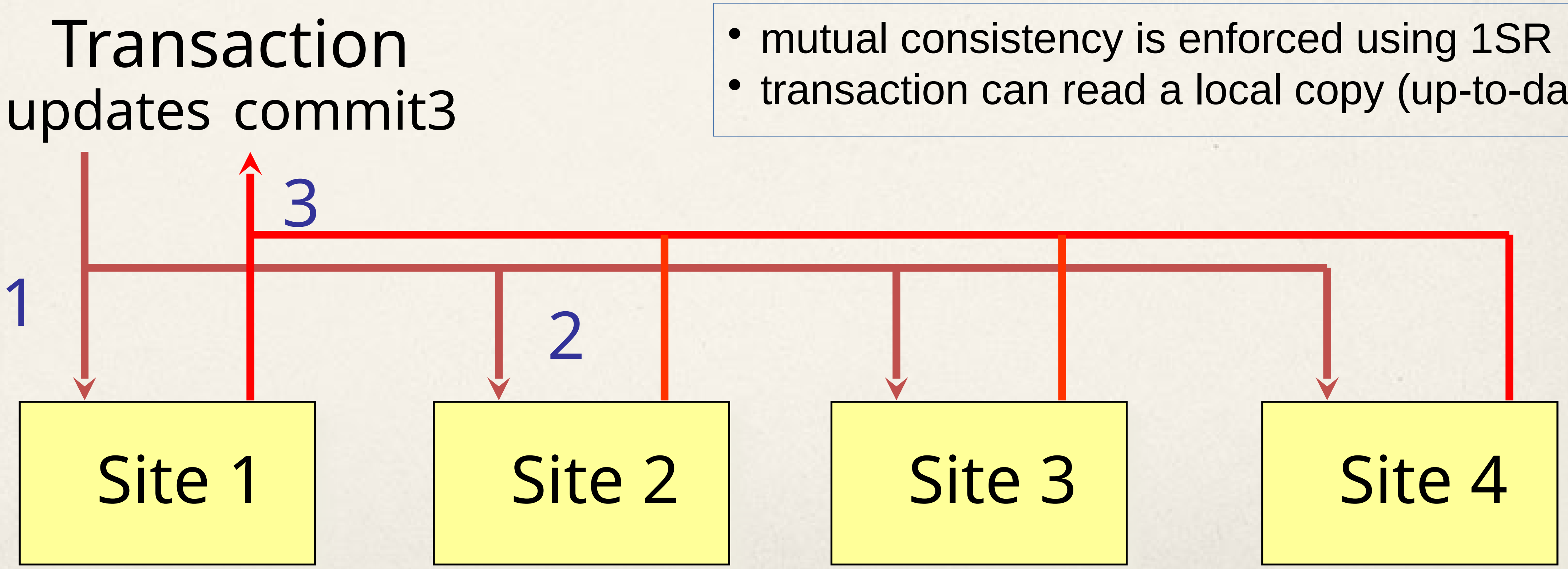
Update Management Strategies

- Depending on when the updates are propagated
  - Eager
  - Lazy
- Depending on where the updates can take place
  - Centralized
  - Distributed

	Centralized	**Distributed**
Eager		
Lazy		

Eager Replication

- Changes are propagated within the scope of the transaction making the changes. The ACID properties apply to all copy updates.
    - Synchronous
    - Deferred
- ROWA protocol: Read-one/Write-all



Transaction

updates commit3

1

2

3

Site 1

Site 2

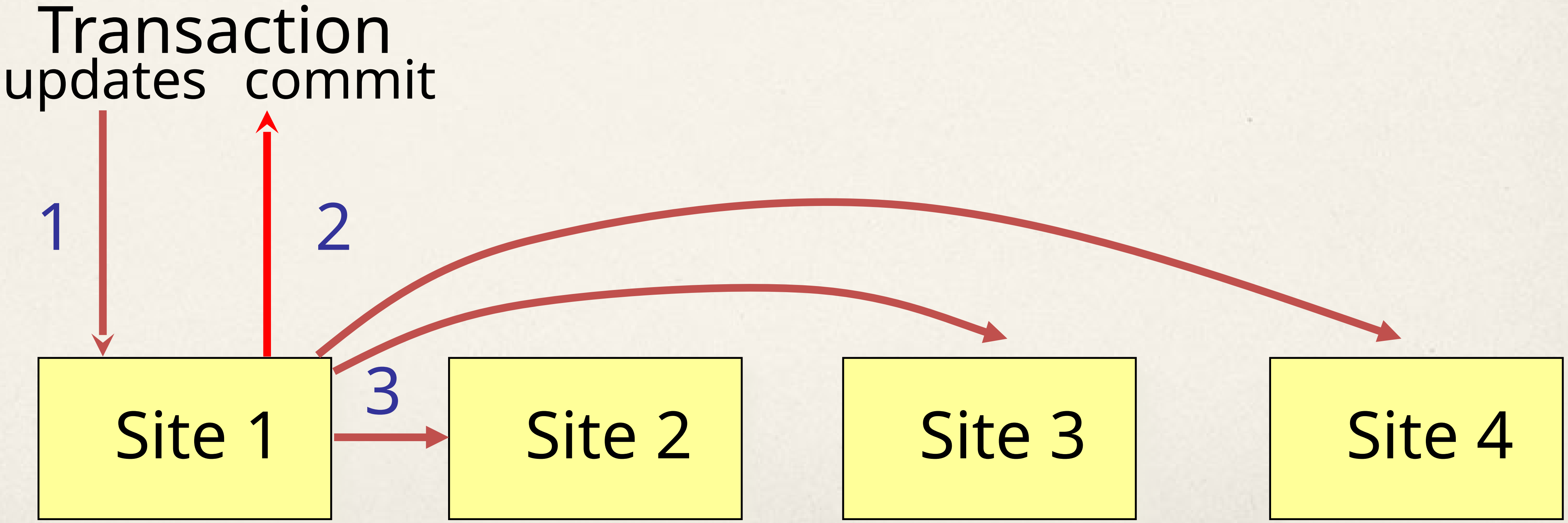
Site 3

Site 4

- mutual consistency is enforced using 1SR
- transaction can read a local copy (up-to-date)

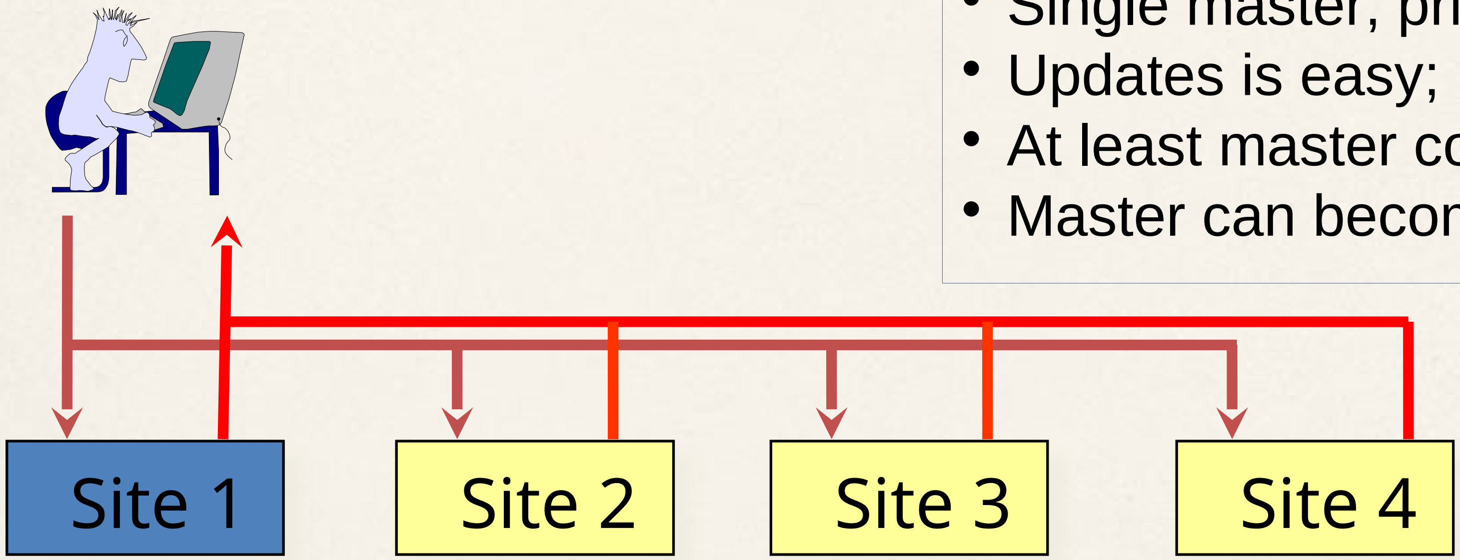
Lazy Replication

- Lazy replication first executes the updating transaction on one copy. After the transaction commits, the changes are propagated to all other copies (refresh transactions)
- While the propagation takes place, the copies are mutually inconsistent.
- The time the copies are mutually inconsistent is an adjustable parameter which is application dependent.

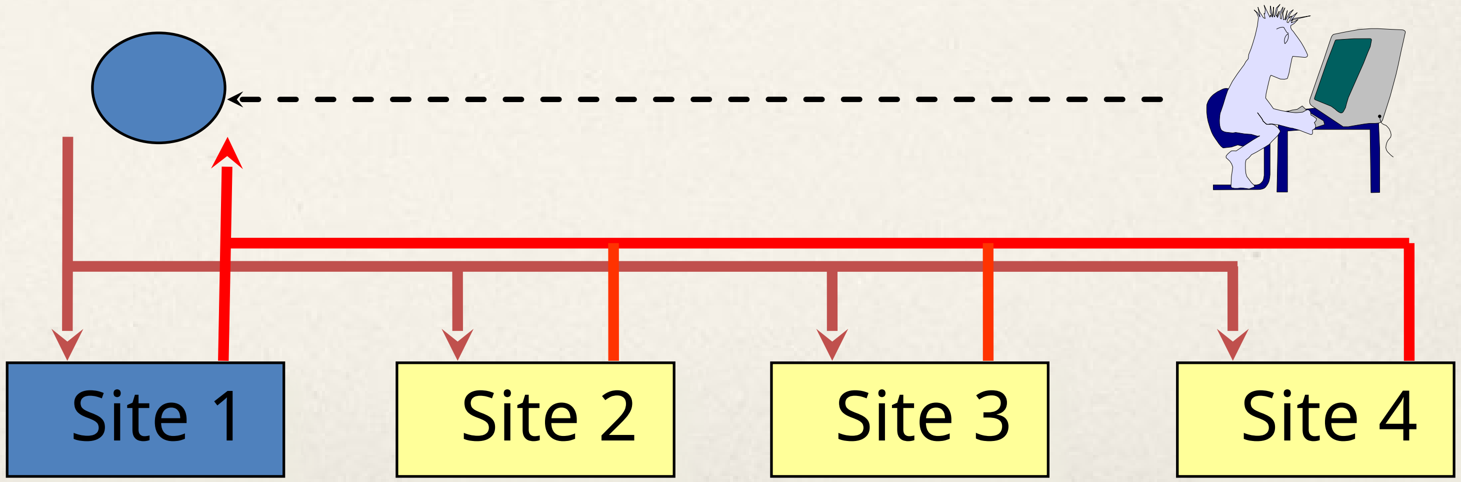


Centralized

- There is only one copy which can be updated (the master), all others (slave copies) are updated reflecting the changes to the master.



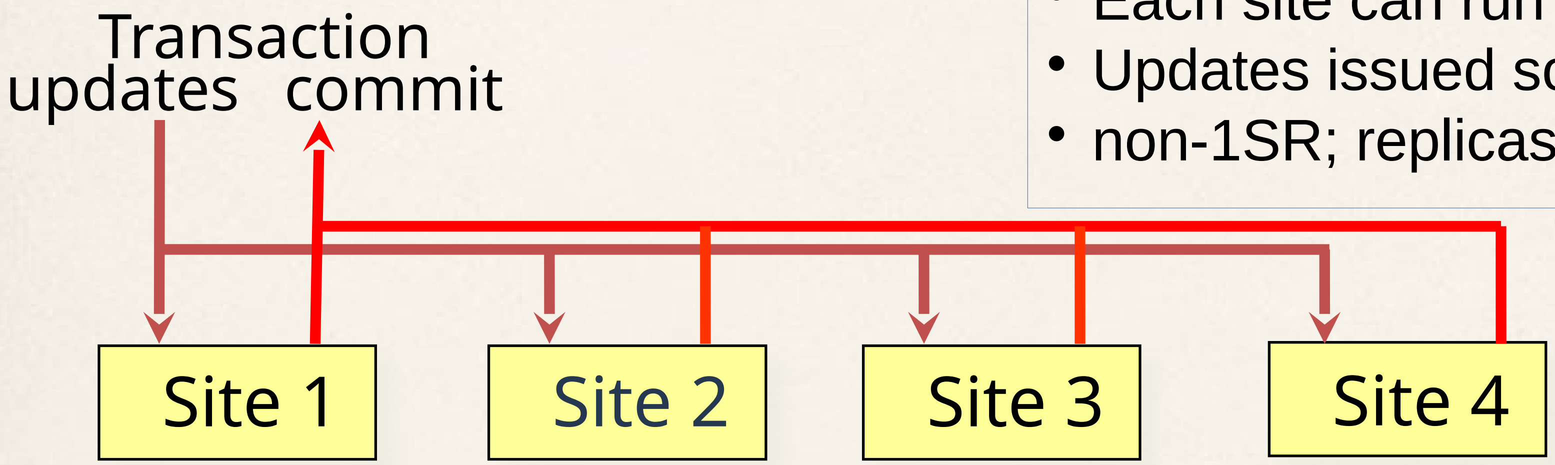
- Single master; primary copy
- Updates is easy; happen at/from master site
- At least master copy is up-to-date
- Master can become a bottleneck



Distributed

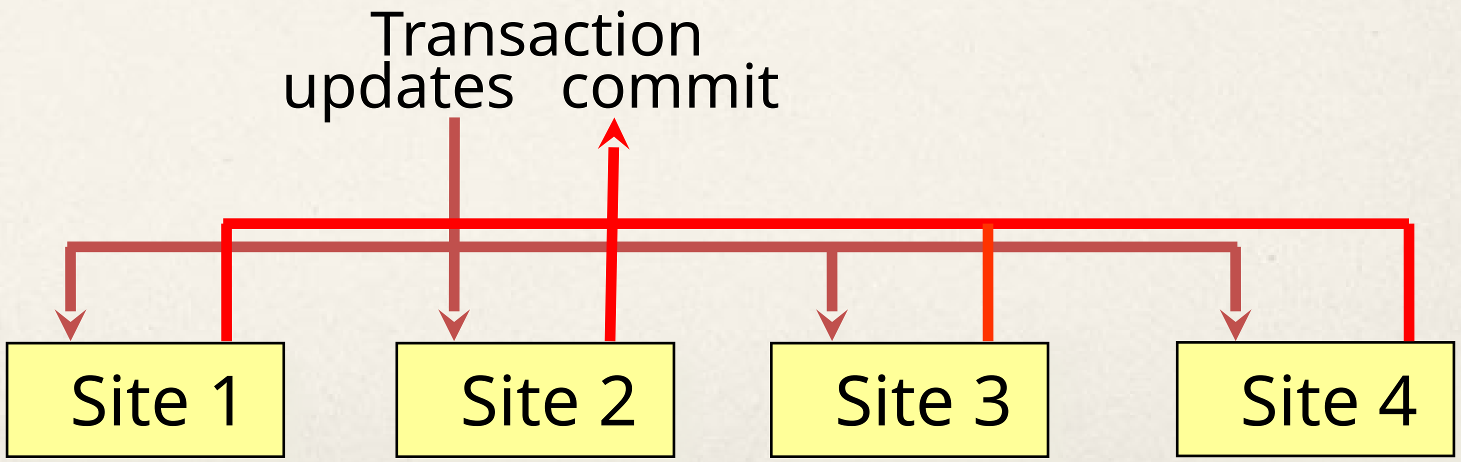
- Changes can be initiated at any of the copies. That is, any of the sites which owns a copy can update the value of the data item.

- Concurrent updates at different sites (masters)
- Each site can run 2PC (eager, no problems)
- Updates issued sometime after commit (lazy)
- non-1SR; replicas out of sync; reconciliation (lazy)



Transaction
updates commit

Site 1
Site 2
Site 3
Site 4



Transaction
updates commit

Site 1
Site 2
Site 3
Site 4

Forms of Replication

Eager

- \+ No inconsistencies (identical copies)
- \+ Reading the local copy yields the most up to date value
- \+ Changes are atomic
- − A transaction has to update all sites
  - − Longer execution time
  - − Lower availability

Lazy

- \+ A transaction is always local (good response time)
- − Data inconsistencies
- − A local read does not always return the most up-to-date value
- − Changes to all copies are not guaranteed
- − Replication is not transparent

Centralized

- \+ No inter-site synchronization is necessary (it takes place at the master)
- \+ There is always one site which has all the updates
- − The load at the master can be high
- − Reading the local copy may not yield the most up-to-date value

Distributed

- \+ Any site can run a transaction
- \+ Load is evenly distributed
- − Copies need to be synchronized

Replication Protocols

The previous ideas can be combined into 4 different replication protocols:

Eager	Eager centralized	Eager distributed
Lazy	Lazy centralized	Lazy distributed
	Centralized	Distributed

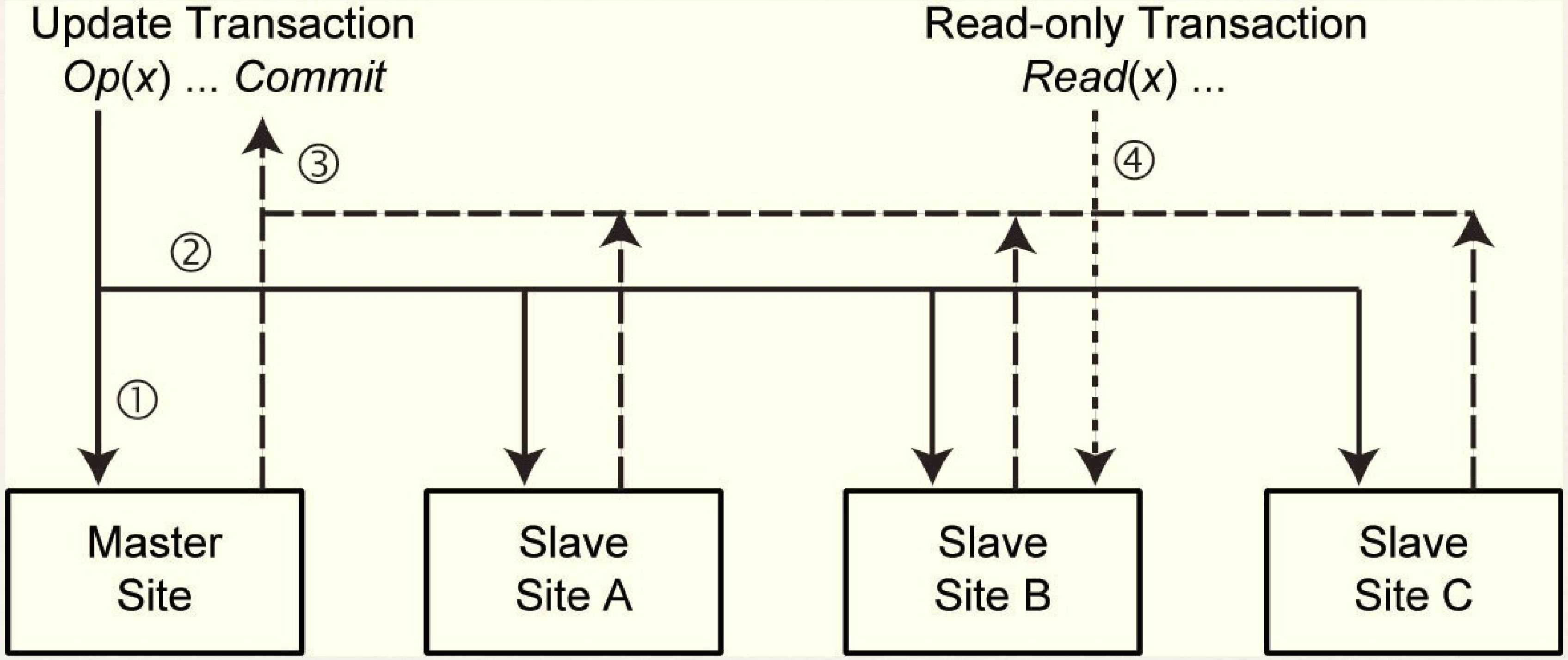
Eager Centralized Protocols

- Design parameters:
  - ➡️ Distribution of master
    - ✦ Single master: one master for all data items
    - ✦ Primary copy: different masters for different (sets of) data items
  - ➡️ Level of transparency
    - ✦ Limited: applications and users need to know who the master is
      - ✓ Update transactions are submitted directly to the master
      - ✓ Reads can occur on slaves
    - ✦ Full: applications and users can submit anywhere and the operations will be forwarded to the master
      - ✓ Operation-based forwarding
- Four alternative implementation architectures, only three are meaningful:
  - ➡️ Single master, limited transparency
  - ➡️ Single master, full transparency
  - ➡️ Primary copy, full transparency

Eager Single Master/Limited Transparency

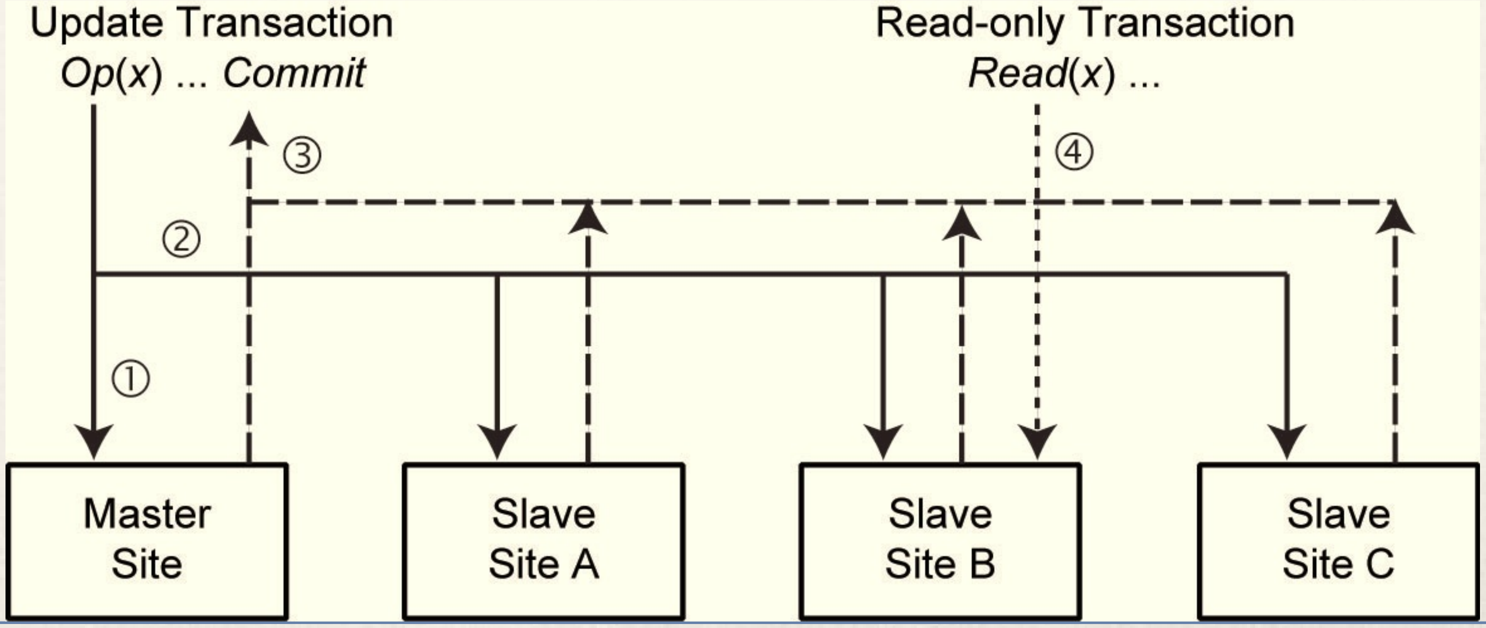
- Applications submit update transactions directly to the master
- Master:
  - ➡️Upon read: read locally and return to user
  - ➡️Upon write: write locally, multicast write to other replicas (in FFO timestamps order)
  - ➡️Centralized CC algorithm at Master's replica site
    - C2PC used for all reads and writes
    - local reads without Master's CC possible
  - ➡️Upon commit request: run 2PC coordinator to ensure that all have really installed the changes
  - ➡️Upon abort: abort and inform other sites about abort
- Slaves
  - ➡️Reads through C2PC protocol (lock request)
    - No CC: One slave reads before write, the other after write; inconsequential from 1SR
  - ➡️Writes are always from master

Eager Single Master/Limited Transparency



Eager Single Master/Limited Transparency (cont'd)

- Relieve master from coordinating reads.
- Applications submit read transactions directly to an approp. slave
- Slave
  - ➡️Upon read: read locally
  - ➡️Upon write from master copy: execute conflicting writes in the proper order (FIFO or timestamp)
  - ➡️Upon write from client: refuse (abort transaction; there is error)
  - ➡️Upon commit request from read-only: commit locally
  - ➡️Participant of 2PC for update transaction running on primary



Eager Single Master/Limited Transparency (Example)

T_1: Write(x)	T_2: Read(x)	T_3: Read(x)
Commit	Commit	Commit

- x is located at site A
- copy avaliable at sites B, C
- T2 is sent to slave at Site B and T3 to slave at Site C.
- T2 reads x at B [Read(xB)] before T1's update is applied at B, while T3 reads x at C [Read(x_C)] after T1 's update at C.

$$H_B = \{R_2(x), C_2, W_1(x), C_1\}$$
$$H_C = \{W_1(x), C_1, R_3(x), C_3\}$$

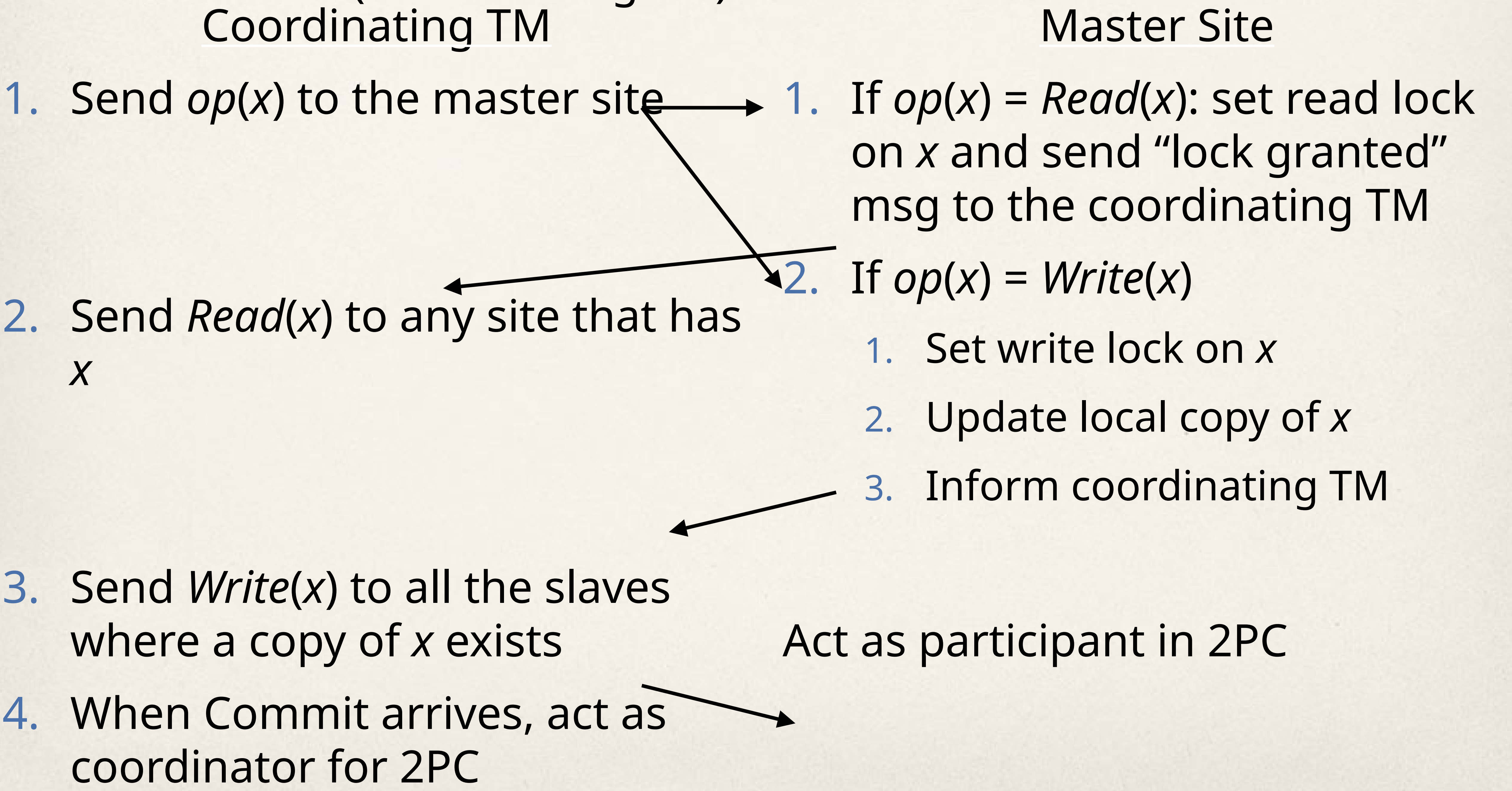
- Site B: T2 → T1 , Site C: T1 → T3 ==> T2 → T1 → T3 (1SR)
- Different x read on B and C

Eager Single Master/ Full Transparency

- How to further relieve central Master's site from heavy load?
  - Use TM at the application site for the coordination
  - Application-site TP handles local reads, updates from master and coordination
    - App-TM could be just a router but this does not solve the problem
- Master's site
  - Runs central TM and LM
  - Performs updates, reads and acks work to coordinating TM
- Coordinating site
  - Handles local reads and update transactions

Eager Single Master/ Full Transparency

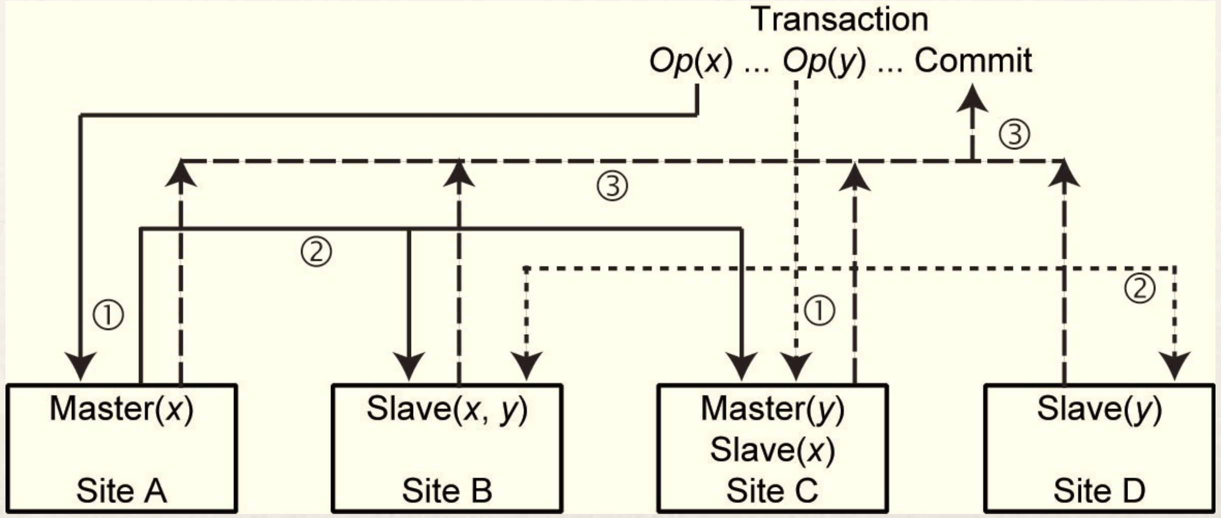
Applications submit all transactions to the Transaction Manager at their own sites (Coordinating TM)



Coordinating TM	Master Site
1. Send *op*(*x*) to the master site	1. If *op*(*x*) = *Read*(*x*): set read lock on *x* and send "lock granted" msg to the coordinating TM
2. Send *Read*(*x*) to any site that has *x*	2. If *op*(*x*) = *Write*(*x*) 1. Set write lock on *x* 2. Update local copy of *x* 3. Inform coordinating TM
3. Send *Write*(*x*) to all the slaves where a copy of *x* exists	Act as participant in 2PC
4. When Commit arrives, act as coordinator for 2PC	

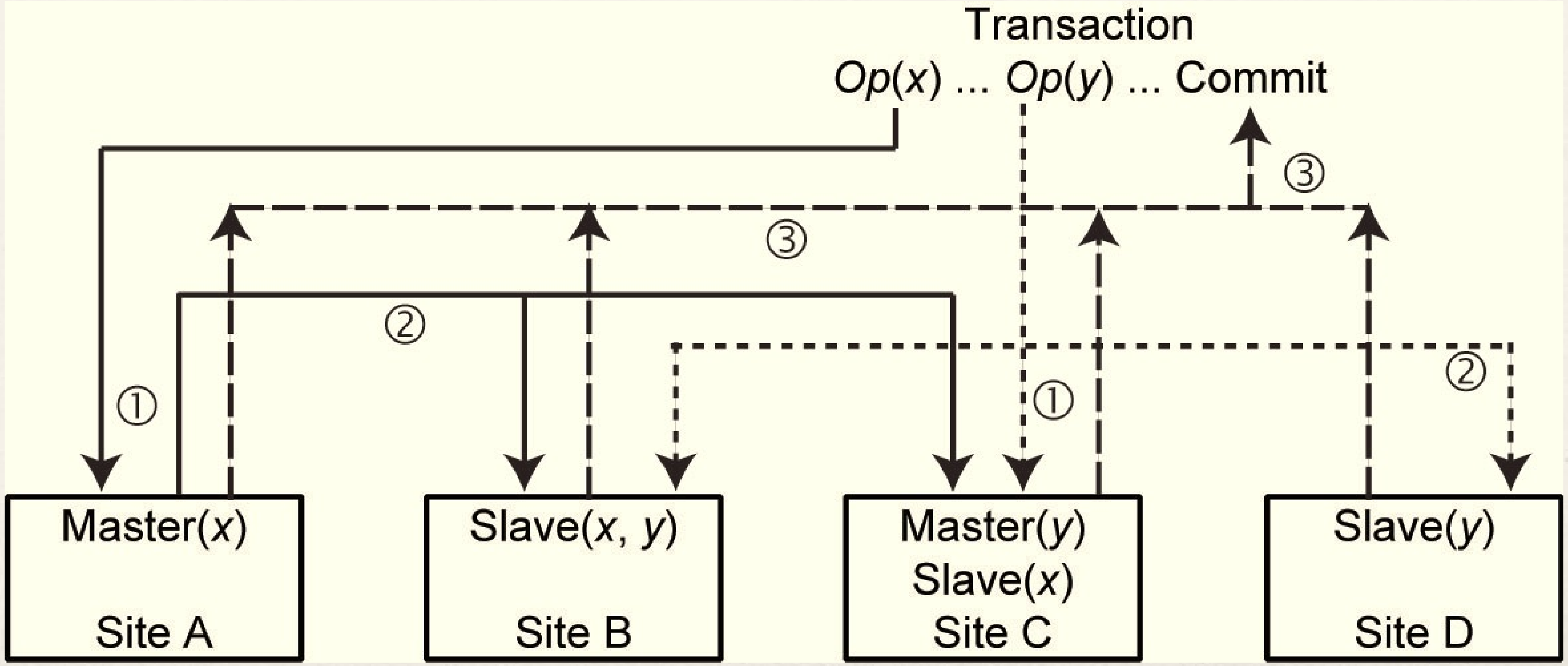
Eager Primary Copy/Full Transparency

- How to distribute the computation (relieve master)?
  - Distribute responsibilities for some data units to primary copy
  - Distributed Ingres, PC2FC
  - Only full transparency makes sense
- Applications submit transactions directly to their local TMs
- Local coordinating TM (application site):
  - ➡️Forward each operation to the primary copy of the data item
  - ➡️Upon granting of locks, submit Read to any slave, Write to all slaves
  - ➡️Coordinate 2PC



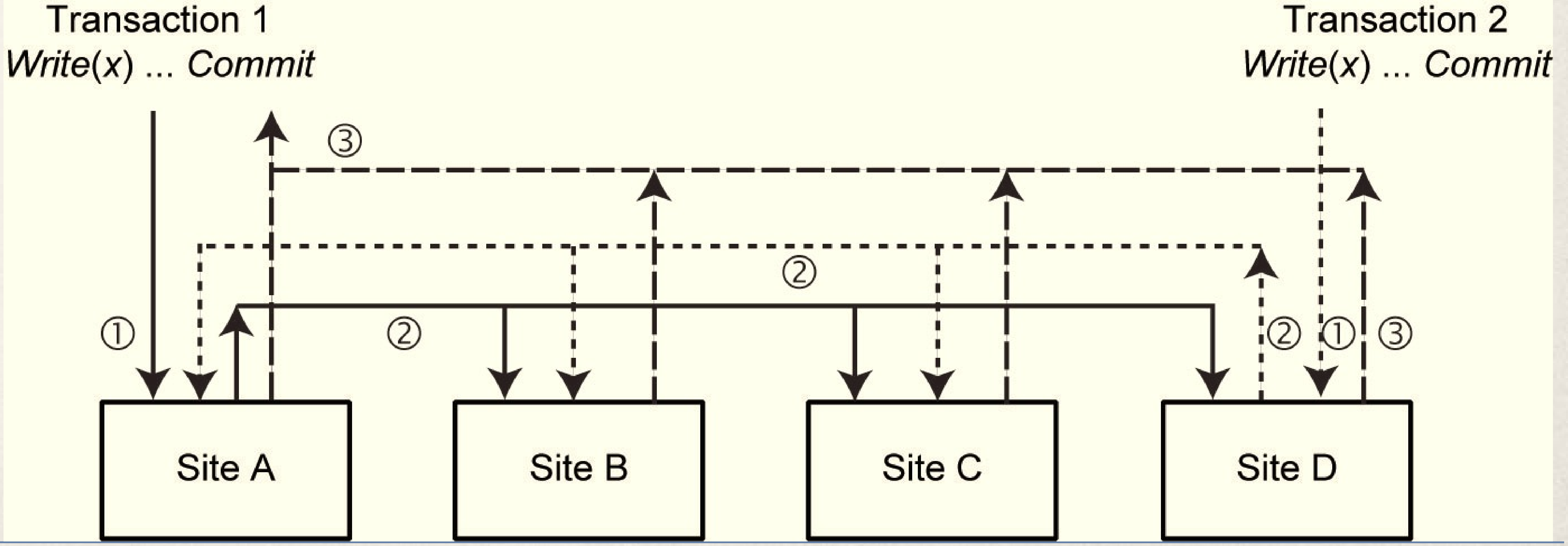
Eager Primary Copy/Full Transparency (cont'd)

- Primary copy site
  - ➡️Read(*x*): lock and reply to TM
  - ➡️Write(*x*): lock *x*, perform update, inform TM
  - ➡️Participate in 2PC
- Slaves: as before



Eager Distributed Protocol

- Updates originate at any copy
    - ➡️Each sites uses 2 phase locking.
    - ➡️Read operations are performed locally.
    - ➡️Write operations are performed at all sites (using a distributed locking protocol).
    - ➡️Coordinate 2PC
- Slaves:
    - ➡️As before



Eager Distributed Protocol (cont'd)

- Critical issue:
  - ➡️ Concurrent Writes initiated at different master sites are executed in the same order at each slave site
  - ➡️ Local histories are serializable (this is easy)
- Advantages
  - ➡️ Simple and easy to implement
- Disadvantage
  - ➡️ Very high communication overhead
    - ✦ *n* replicas; *m* update operations in each transaction: *n*m* messages (assume no multicasting)
    - ✦ For throughput of *k* tps: *k* n*m* messages
- Alternative
  - ➡️ Use group communication + deferred update to slaves to reduce messages

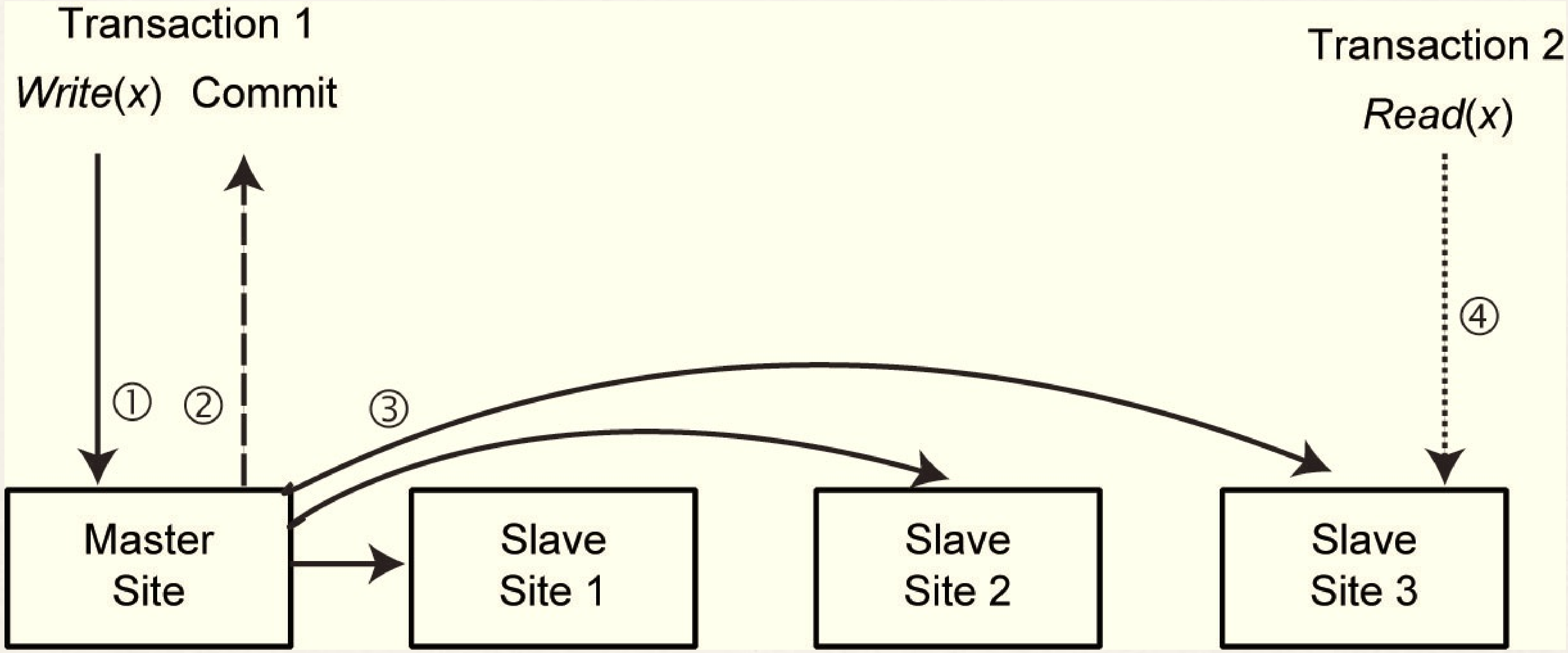
Lazy centralized protocols

- Lazy centralized replication algorithms are similar to eager
  - updates are first applied to a master replica and then propagated to the slaves
  - propagation does not take place within the update transaction
  - after the transaction commits separate refresh Transaction sent to slaves
- Slave site performs a Read(x) operation on its local copy
  - It may read stale (non-fresh) data
  - x may have been updated at the master, but the update may not have yet been propagated to the slaves.

Lazy Single Master/Limited Transparency

- Update transactions submitted to master
- Master:
  - ➡️Upon read: read locally and return to user
  - ➡️Upon write: write locally and return to user
  - ➡️Upon commit/abort: terminate locally
  - ➡️Sometime after commit: multicast updates to slaves (in order)
- Slaves:
  - ➡️Upon read: read locally + return result to the user
  - ➡️Write(x) received by a slave is rejected
  - ➡️Refresh transactions: install updates
- Updates at slaves have to be ordered as the master defines
  - ➡️Updates from a single master => no problem
  - ➡️Use timestamps generated at master

Lazy Single Master/Limited Transparency



Lazy Primary Copy/Limited Transparency

- There are multiple masters
  - Each master execution is similar to lazy single master in the way it handles transactions
  - Write(x) is submitted to the primary copy of x; the rest is straightforward.
- Slave execution complicated:
  - refresh transactions from multiple masters and need to be ordered properly
    - Timestamps (attached to site name) – there is one primary master for each copy!
    - Replication graph

Lazy Primary Copy/Limited Transparency – Slaves

- Assign system-wide unique timestamps to refresh transactions and execute them in timestamp order
  - May cause too many aborts; because of the refresh transactions
  - Problems: Out-of-order transactions (local reads) may be aborted
- Replication graph
  - Similar to serialization graph, but nodes are transactions (T) + sites (S); edge $<T_i,S_j>$ exists iff T_i performs a Write(x) and x is stored in S_j
  - For each operation (op_k), enter the appropriate nodes (T_k) and edges; if graph has no cycles, no problem
  - If cycle exists and the transactions in the cycle have been committed at their masters, but their refresh transactions have not yet committed at slaves, abort T_k; if they have not yet committed at their masters, T_k waits.
- Use group communication

Lazy Single Master/Full Transparency

- This is very tricky
  - ➡️Forwarding operations to a master and then getting refresh transactions cause difficulties
- Two problems:
  - ➡️Violation of 1SR behavior
  - ➡️A transaction may not see its own writes
- Problem arises in primary copy/full transparency as well

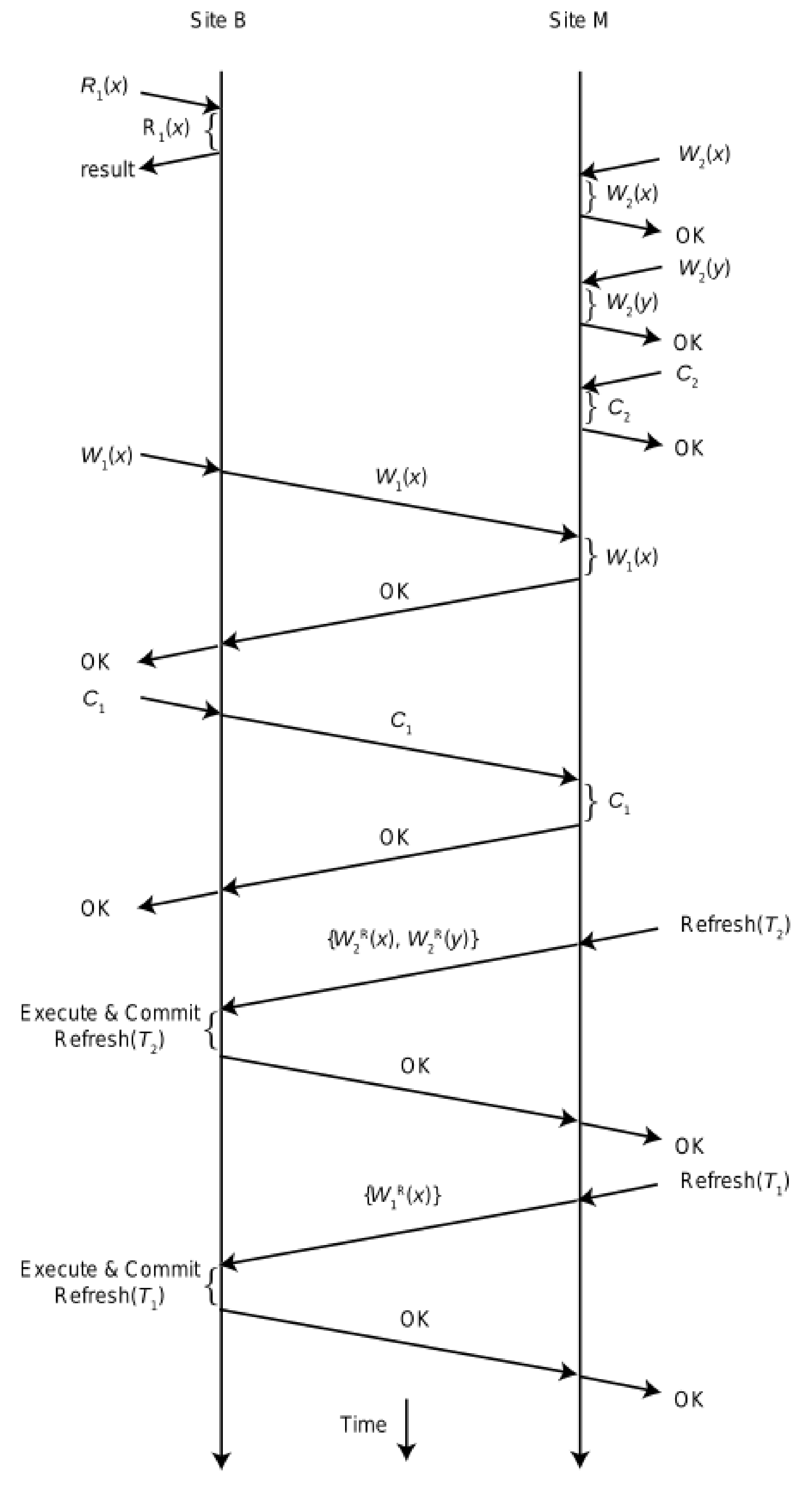
Example 3

Site *M* (Master) holds *x*, *y*;
Site *B* holds slave copies of *x*, *y*
T_1: Read(*x*), Write(*x*), Commit
T_2: Write(*x*), Write(*y*), Commit
T_1: at B
T_2: at M

Non-1SR !

$$H_B = \{W_2(x_M), W_2(y_M), C_2, W_1(y_M), C_1\}$$

$$H_B = \{R_1(x_B), C_1, W_2^r(x_B), W_2^r(y_B), C_2^r, W_1^r(x_B), C_1^r\}$$



Example 4

- Master site *M* holds *x*, site *C* holds slave copy of *x*
- T_3: Write(*x*), Read(*x*), Commit
- Sequence of execution
  1. $W_3(x)$ submitted at *C*, forwarded to *M* for execution
  2. $W_3(x)$ is executed at *M*, confirmation sent back to *C*
  3. $R_3(x)$ submitted at *C* and executed on the local copy
  4. T_3 submits Commit at *C*, forwarded to *M* for execution
  5. *M* executes Commit, sends notification to *C*, which also comm. T_3
  6. *M* sends refresh transaction for T_3 to *C* (for $W_3(x)$ operation)
  7. *C* executes the refresh transaction and commits it
- When *C* reads *x* at step 3, it does not see the effects of Write at step 2

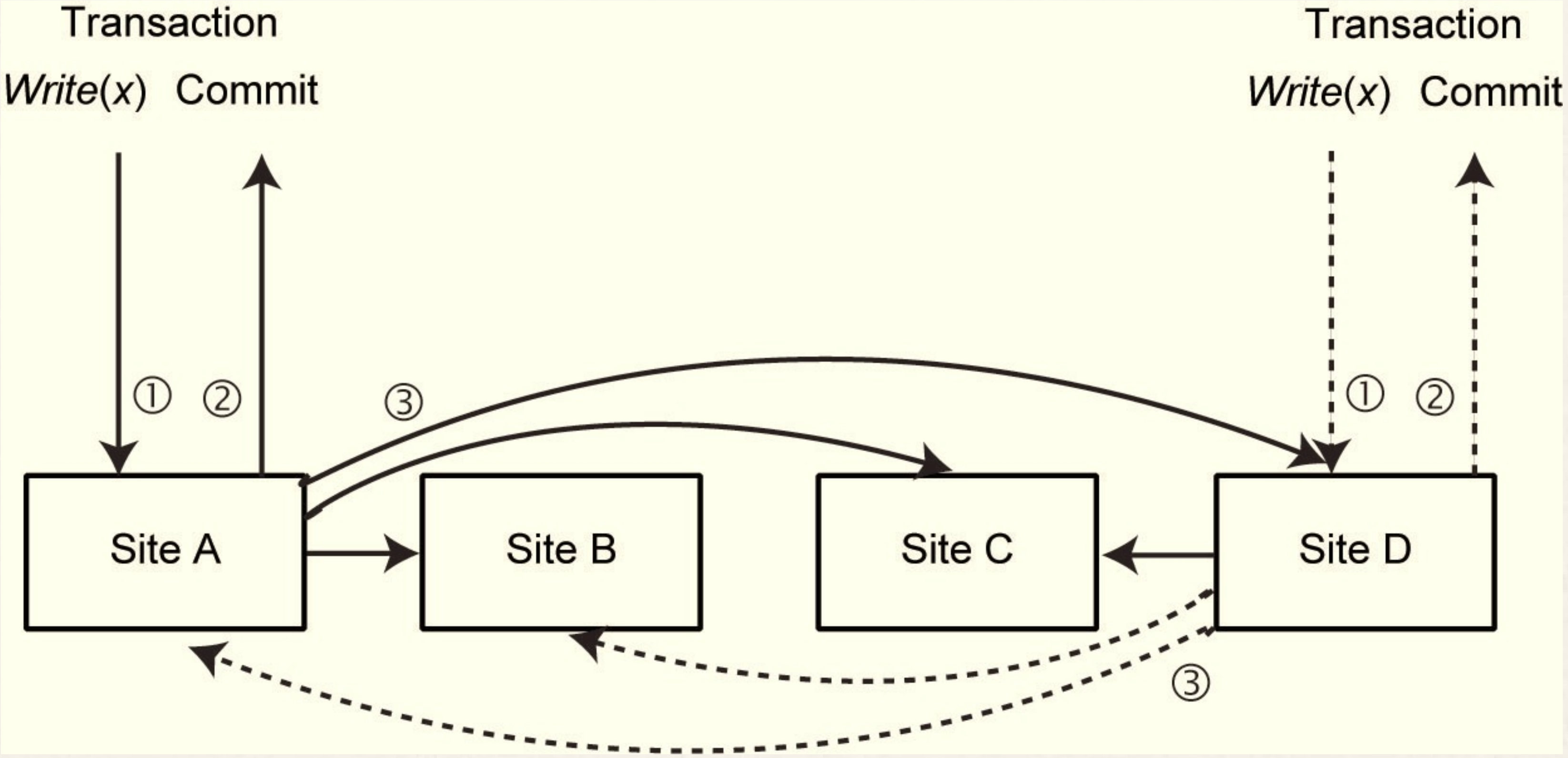
Lazy Single Master/ Full Transparency - Solution

- Assume T = Write(x)
- At commit time of transaction T, the master generates a timestamp for it [$ts(T)$]
- Master sets $last\_modified(x_M) \leftarrow ts(T)$
- When a refresh transaction arrives at a slave site i, it also sets $last\_modified(x_i) \leftarrow last\_modified(x_M)$
- Timestamp generation rule at the master:
    - ➡️ $ts(T)$ should be greater than all previously issued timestamps and should be less than the *last_modified* timestamps of the data items it has accessed. If such a timestamp cannot be generated, then T is aborted.

Lazy Distributed Replication

- Any site:
  - ➡️Upon read:
    - Read locally and return to user
  - ➡️Upon write:
    - Write locally and return to user
  - ➡️Upon commit/abort:
    - Terminate locally
  - ➡️Sometime after commit:
    - Send refresh transaction
  - ➡️Refresh transactions have to be ordered properly!
    - ✦ Possible concurrent change of data item at 2 sites!
    - ✦ Data item updated at two sites and the refresh trans. sent
    - ✦ These changes|ordering need to be reconciled + establish global ordering
  - ➡️Upon message from other site
    - ✦ Detect conflicts
    - ✦ Install changes
    - ✦ Reconciliation may be necessary

Lazy Distributed Replication



Reconciliation

- Such problems can be solved using pre-arranged patterns:
  - ➡️Use timestamps as before. Latest update win (newer updates preferred over old ones)
  - ➡️Site priority (preference to updates from headquarters)
  - ➡️Largest value (the larger transaction is preferred)
- Or using ad-hoc decision making procedures:
  - ➡️Identify the changes and try to combine them
  - ➡️Analyze the transactions and eliminate the non-important ones
  - ➡️Implement your own priority schemas

Replication Strategies

	Centralized	Distributed
Eager	+ Updates do not need to be coordinated + No inconsistencies - Longest response time - Only useful with few updates - Local copies are can only be read	+ No inconsistencies + Elegant (symmetrical solution) - Long response times - Updates need to be coordinated
Lazy	+ No coordination necessary + Short response times - Local copies are not up to date - Inconsistencies	+ No centralized coordination + Shortest response times - Inconsistencies - Updates can be lost (reconciliation)

Group Communication

- A node can multicast a message to all nodes of a group with a delivery guarantee
- Multicast primitives
  - There are a number of them
  - Total ordered multicast: all messages sent by different nodes are delivered in the same total order at all the nodes
- Used with deferred writes, can reduce communication overhead
  - Remember eager distributed requires $k*m$ messages (with multicast) for throughput of $k$tps when there are n replicas and m update operations in each transaction
  - With group communication and deferred writes: $2k$ messages