CS4224/CS5424 Lecture 6
Data Replication

Data Replication

Objectives: Improve

- System availability
- Performance
- Scalability

Challenge: How to keep replicas synchronized

Data Replication

- Logical data: x
- Physical copies/replicas: $x_A, x_B, \cdots,$
  - x_i denote the replica at Site i
- Replication transparency
  - Transaction issues read/write operations on x
  - Replica control protocol maps operations on logical data to operations on physical replicas

1SR & Mutual Consistency

- One-copy database = non-replicated database
- An execution is one-copy serializable (1SR) if it has the same effect as a serial execution on a one-copy database
- A replicated database is in a mutually consistency state if all the replicas of its data items have identical values

One-Copy Serializability: Example

Assume that initial replicas are synchronized:

$x_A = x_B$, $y_A = y_B$, $z_A = z_B$

Site A	Site B
$R_1(x_A)$	$R_2(x_B)$
$R_1(y_A)$	$R_2(z_B)$
$W_1(z_A)$	$W_2(y_B)$
$W_2(y_A)$	$W_1(z_B)$
$Commit_1$	$Commit_1$
$Commit_2$	$Commit_2$

One-Copy Serializability: Example

Assume that initial replicas are synchronized: $x_A = x_B$

Site A	Site B
$R_1(x_A)$	
$W_1(x_A)$	
$Commit_1$	
	$W_1(x_B)$
	$R_2(x_B)$
$R_3(x_A)$	$W_2(x_B)$
	$Commit_2$
$W_2(x_A)$	
$W_3(x_A)$	
$Commit_3$	
	$W_3(x_B)$

Mutual Consistency

- A replicated database is in a mutually consistency state if all the replicas of its data items have identical values
- Strong mutual consistency
  - All copies of a data item have the same value at the end of update Xact
- Weak mutual consistency
  - Does not require all copies of a data item to be identical at the end of update Xact
  - aka eventual consistency

Example 1: MC but not 1SR

- Site A = $\{x\}$, Site B = $\{x, y\}$, Site C = $\{x, y, z\}$
- Assume that the initial replicas are synchronized: $x_A = x_B = x_C$ & $y_B = y_C$
- Transactions:
  - T_1: $W_1(x)$
  - T_2: $R_2(x)$, $W_2(y)$
  - T_3: $R_3(x)$, $R_3(y)$, $W_3(z)$
- Local schedules
  - S_A : $W_1(x_A)$
  - S_B : $W_1(x_B), R_2(x_B), W_2(y_B)$
  - S_C : $W_2(y_C), R_3(x_C), R_3(y_C), W_3(z_C), W_1(x_C)$

Example 2: Neither MC nor 1SR

- Site A = $\{x\}$, Site B = $\{x\}$
  where the initial replicas are synchronized
- Transactions
  - T_1: $R_1(x)$, $W_1(x)$
  - T_2: $R_2(x)$, $W_2(x)$
- Local schedules:
  - $S_A : R_1(x_A), W_1(x_A), W_2(x_A)$
  - $S_B : R_2(x_B), W_2(x_B), W_1(x_B)$

One-Copy Serializability

- Replicated data (RD) schedules: schedules on replicated database
- One-copy (1C) schedules: schedules on non-replicated database
- T_j reads x from T_i in a RD schedule if
    1. for some copy x_A of x, $W_i(x_A)$ precedes $R_j(x_A)$, and
    2. there is no $W_k(x_A)$, $k \neq i$, that occurs between $W_i(x_A)$ & $R_j(x_A)$

One-Copy Serializability (cont.)

- Let T denote a set of committed transactions
- Let S_{RD} denote a RD schedule over T
- Let S_{1C} denote a 1C schedule over T
- S_{RD} is equivalent to S_{1C} if
  1. T_j reads x from T_i in S_{RD} iff T_j reads x from T_i in S_{1C}, and
  2. for each final write $W_i(x)$ in S_{1C}, $W_i(x_A)$ is a final write in S_{RD} for some copy x_A of x
- A replicated data schedule is one-copy serializable (1SR) if it is equivalent to a serial one-copy schedule

Example 3

- Site A = $\{x, y\}$, Site B = $\{x, y\}$
  where the initial replicas are synchronized
- Transactions
  - T_1: $W_1(x)$
  - T_2: $R_2(x)$, $R_2(y)$
  - T_3: $W_3(y)$
  - T_4: $R_4(x)$, $R_4(y)$
- Local schedules
  - S_A: $W_1(x_A)$, $R_2(x_A)$, $R_2(y_A)$
  - S_B: $W_3(y_B)$, $R_4(x_B)$, $R_4(y_B)$

Example 4

- Site A = $\{x, y\}$, Site B = $\{x, y\}$
  where the initial replicas are synchronized
- Transactions
  - T_1: $W_1(x)$
  - T_2: $R_2(x)$, $R_2(y)$
  - T_3: $W_3(y)$
  - T_4: $R_4(x)$, $R_4(y)$
- Local schedules
  - S_A: $W_1(x_A)$, $R_4(x_A)$, $R_2(x_A)$, $R_2(y_A)$, $R_4(y_A)$
  - S_B: $W_3(y_B)$

How to send updates to replicas?

- Suppose a Xact T has updated data at one site. How to send T's updates to other replicas?
- Replication Methods
  - **DBMS-level replication**
    - Statement-based replication
    - Write-ahead log (WAL) shipping
  - **Application-level replication**
- Statement-based replication
  - Forward T's update/insert/delete SQL statements to replica sites for execution
  - Example: VoltDB

How to send updates to replicas? (cont.)

- Write-ahead log (WAL) shipping
  - Send T's log records to replica sites for synchronization
    - File-based log shipping
    - Record-based log shipping (streaming replication)
  - Physical/Logical replication - format of shipped log records are logical/physical
  - **Physical replication**
    - Storage-based specification of updates (e.g. location of modified bytes on disk block)
    - Examples: Oracle, PostgreSQL (before version 10)
  - **Logical replication**
    - Contains one log record for each new/deleted/updated tuple
    - Examples: Oracle, MySQL, PostgreSQL (version 10 onwards)

How to send updates to replicas? (cont.)

- Application-level replication
  - Implement using triggers & stored procedures
  - More flexibility but higher overhead

Replication Protocols

WHEN \ WHERE	Centralized	Distributed
Eager	Eager Centralized	Eager Distributed
Lazy	Lazy Centralized	Lazy Distributed

- **When** are updates propagated to copies?
- **Where** are updates allowed to occur?

When are updates propagated to copies?

- Eager (or synchronous) update: Propagates updates to all replicas within context of Xact
- Lazy (or asynchronous) update: Xact updates only one replica; updates to remaining replicas are propagated asynchronously

```
T:    Begin transaction
      ...
      Write(x_a)
      Write(x_b)
      Write(x_c)
      ...
      Commit
```

Eager update

```
T:    Begin transaction
      ...
      Write(x_a)
      ...
      Commit
```

```
Sometime later:
      Write(x_b)
      Write(x_c)
```

Lazy update

When are updates propagated to copies? (cont.)

- **Eager Update**
  - Enforces strong mutual consistency
  - Based on Read-One-Write-All (ROWA) protocols
- **Lazy Update**
  - Xact commits as soon as one replica is updated
  - Updates to remaining replicas are propagated asynchronously
    - Refresh Xacts sent to other replica sites after update Xact commits
  - Lazy updates from different Xacts can conflict
  - Need to ensure that updates are applied in the same order to all replicas

Where are updates allowed to occur?

- **Centralized techniques**
  - Update is applied to a master copy first before propagating to other slave copies
    - Master site: site that hosts the master copy
    - Slave site: site that hosts a slave copy
  - aka single master, master-slave, or active-passive replication
- **Distributed techniques**
  - Update is applied to any copy & then propagated to other copies
  - aka multimaster, update anywhere, master-master, or active-active replication

Assumptions for protocol discussions

- Strict 2PL is used for concurrency control
- Statement-based replication method is used to propagate updates

Replication Protocols

1. Eager Centralized Protocols
    - Eager Single-Master
    - Eager Primary Copy
2. Eager Distributed Protocols
3. Lazy Centralized Protocols
    - Lazy Single-Master
4. Lazy Distributed Protocols

Eager Single-Master Protocol

- **Eager** = all replicas updated within context of Xact
- **Centralized** = master copy is updated before slave copies

Eager Single-Master Protocol (cont.)

- There is a single master site containing master copies of all objects
  - TM at master site is the centralized lock manager
  - Coordinating TM for T_i sends each operation of T_i to master site's TM
- For each update operation $W_i(O)$,
  - Master copy of O at master site must be updated first
  - Updates are then propagated to other copies of O at slave sites
- For each read operation $R_i(O)$,
  - Coordinating TM for T_i can read from any one replica of O

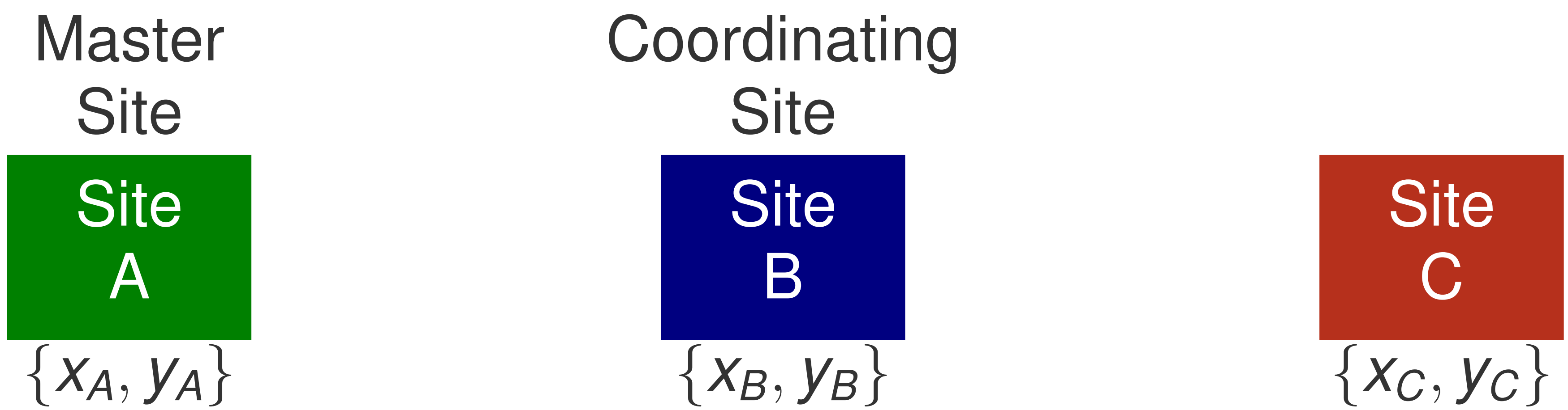
Eager Single-Master Protocol (cont.)

- Site S_A is the master site
- Consider a Xact T_i issued at site S_B
  - TM_B is the coordinating TM for T_i
- To process $R_i(x)$, TM_B sends lock request for $R_i(x)$ to TM_A
- TM_A checks if S-lock on x can be granted to T_i
- If granted,
  - TM_A notifies TM_B that lock request is granted
  - If TM_B has a replica of x, TM_B reads its local copy of x (i.e., $R_i(x_B)$);
  - Otherwise, TM_B sends $R_i(x)$ to any site (say S_C) with a replica of x
    - ★ TM_C executes $R_i(x_C)$ and returns x_C to TM_B
- Otherwise, T_i is blocked

Eager Single-Master Protocol (cont.)

- To process $W_i(x)$, TM_B sends lock request for $W_i(x)$ to TM_A
- TM_A checks if X-lock on x can be granted to T_i
- If granted,
  - TM_A updates its copy of x (i.e., $W_i(x_A)$)
  - TM_A notifies TM_B that lock request is granted
  - TM_B sends $W_i(x)$ to other sites with replicas of x
  - TM_B executes $W_i(x_B)$ if S_B has a replica of x
- Otherwise, T_i is blocked

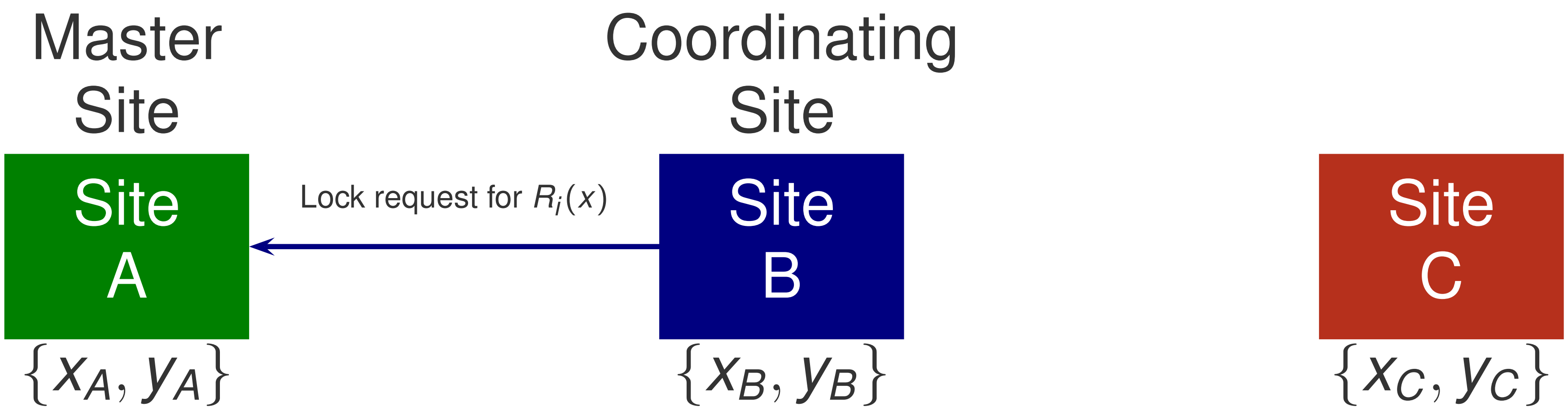
Eager Single-Master Protocol: Example



Transaction T_i needs to read x & write y

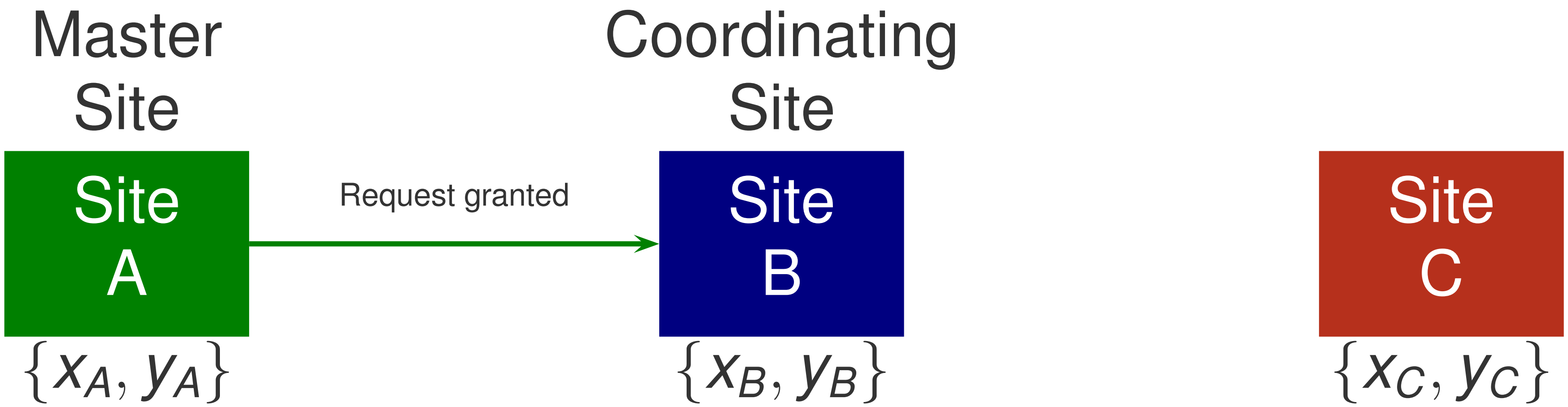
Site B is the coordinating site for Xact T_i

Eager Single-Master Protocol: Example



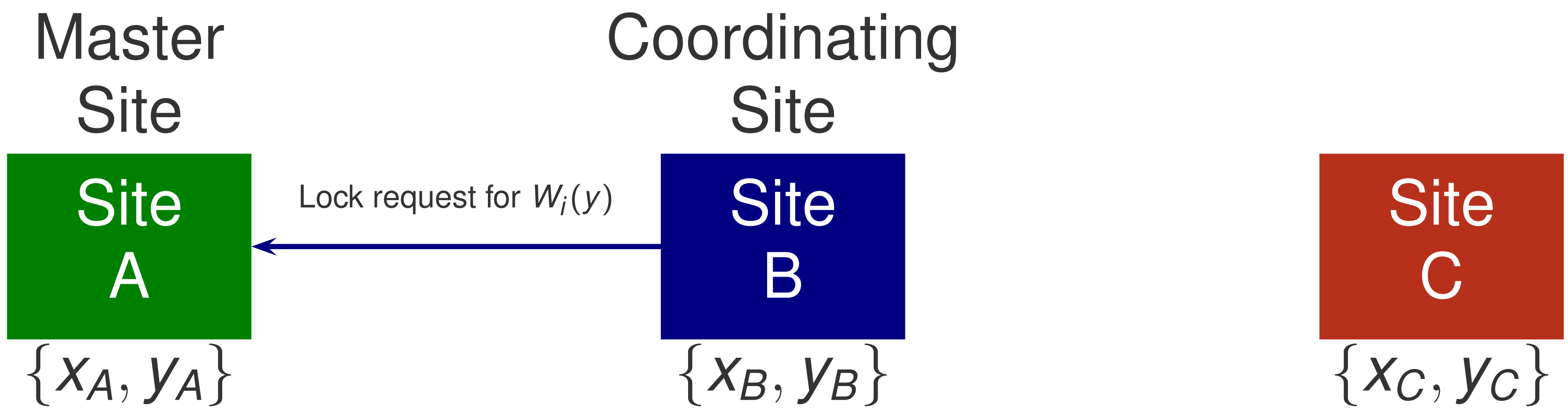
TM_B sends lock request for $R_i(x)$ to TM_A

Eager Single-Master Protocol: Example



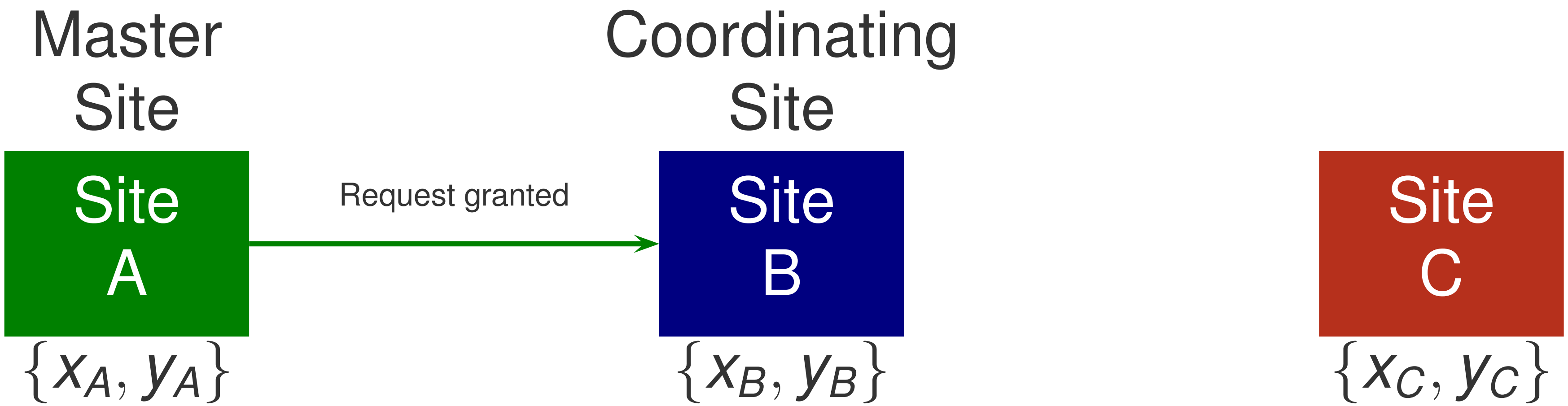
TM_A grants S-lock for $R_i(x)$ & notifies TM_B

Eager Single-Master Protocol: Example



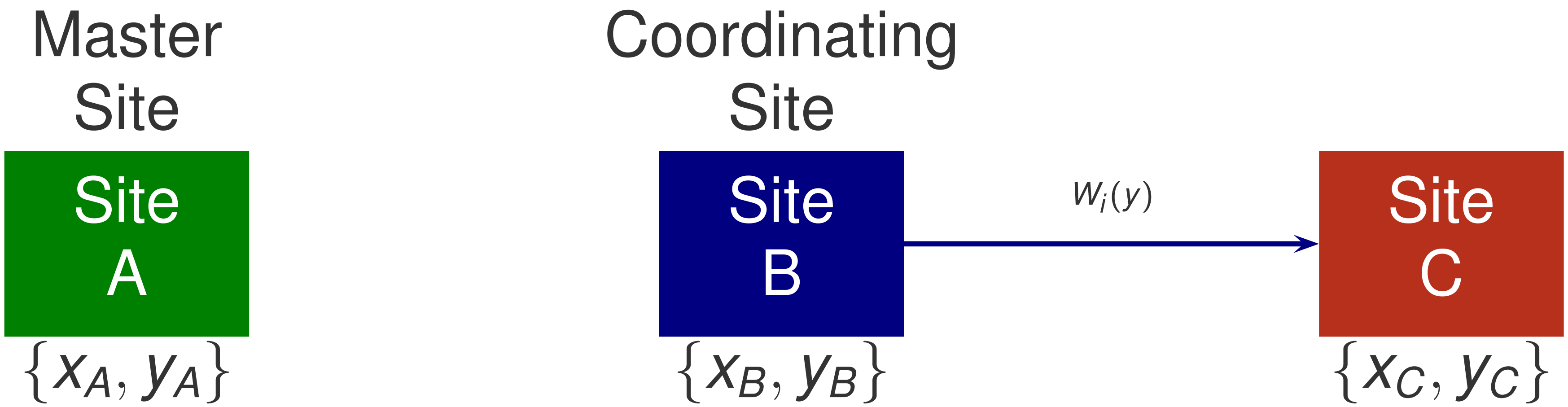
TM_B executes $R_i(x_B)$ &
TM_B sends lock request for $W_i(y)$ to TM_A

Eager Single-Master Protocol: Example



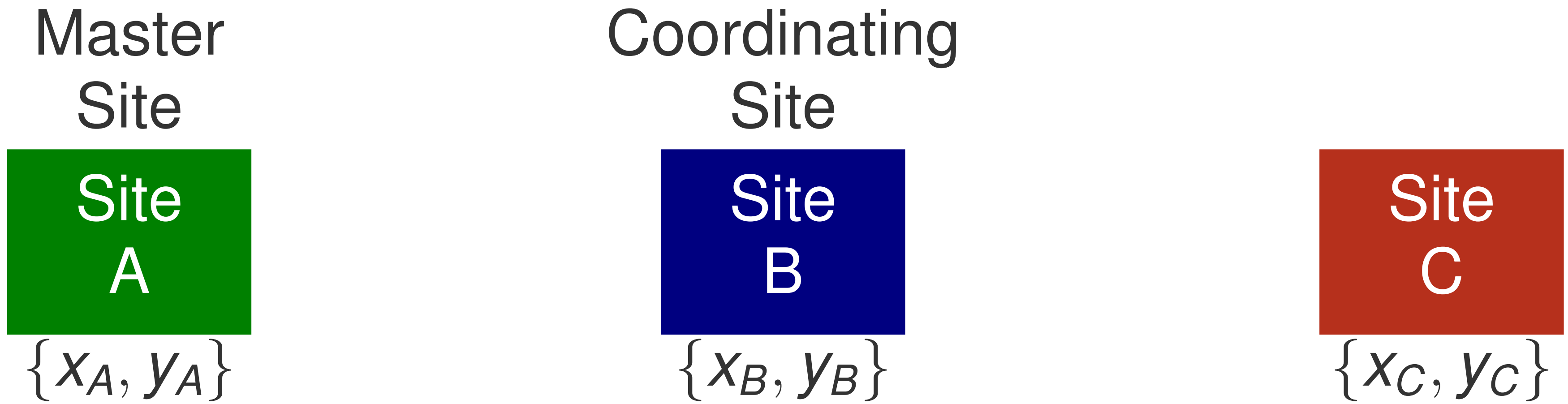
TM_A grants X-lock for $W_i(y)$, executes $W_i(y_A)$ & notifies TM_B

Eager Single-Master Protocol: Example



TM_B executes $W_i(y_B)$ & sends $W_i(y)$ to TM_C

Eager Single-Master Protocol: Example

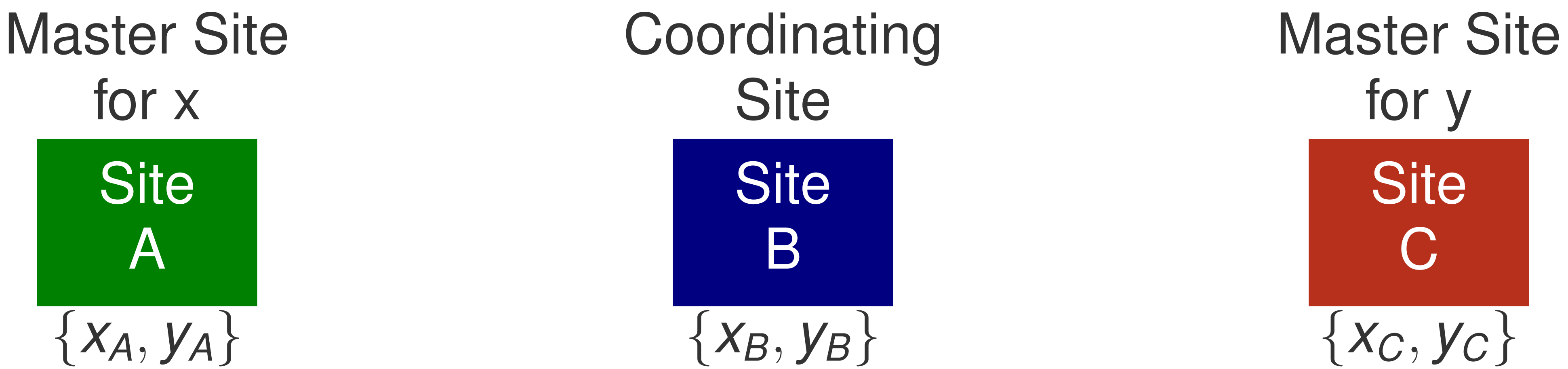


TM_C executes $W_i(y_C)$

Eager Primary Copy Protocol

- Generalization of Eager Single-Master Protocol
- For each replicated object, one of its copies is designated the primary copy
- The master site for object O is the site that stores the primary copy of O
- Each master site runs a lock manager
  - Controls lock requests/releases for primary copies under its control

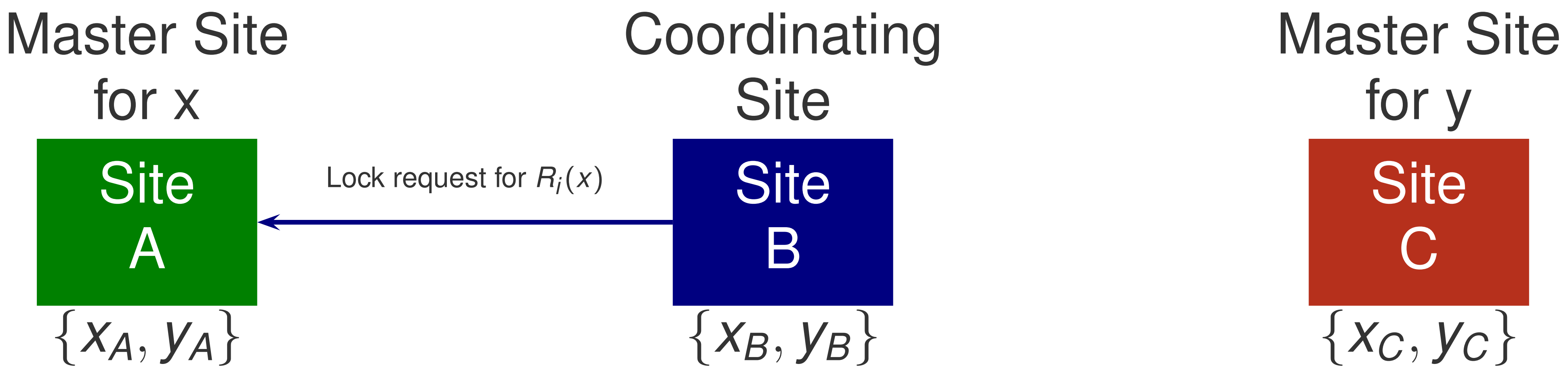
Eager Primary Copy Protocol: Example



Transaction T_i needs to read x & write y

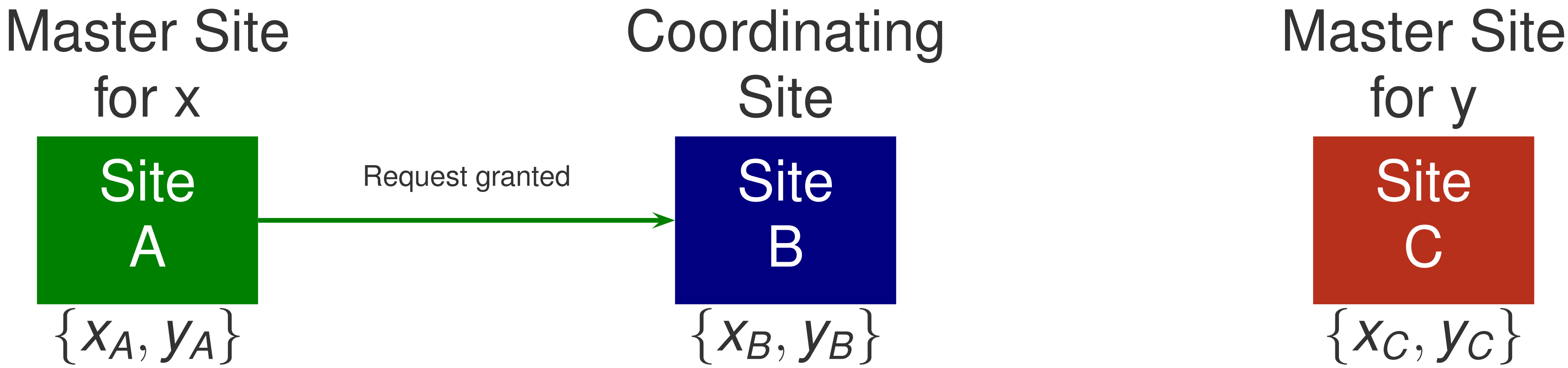
Site B is the coordinating site for Xact T_i

Eager Primary Copy Protocol: Example



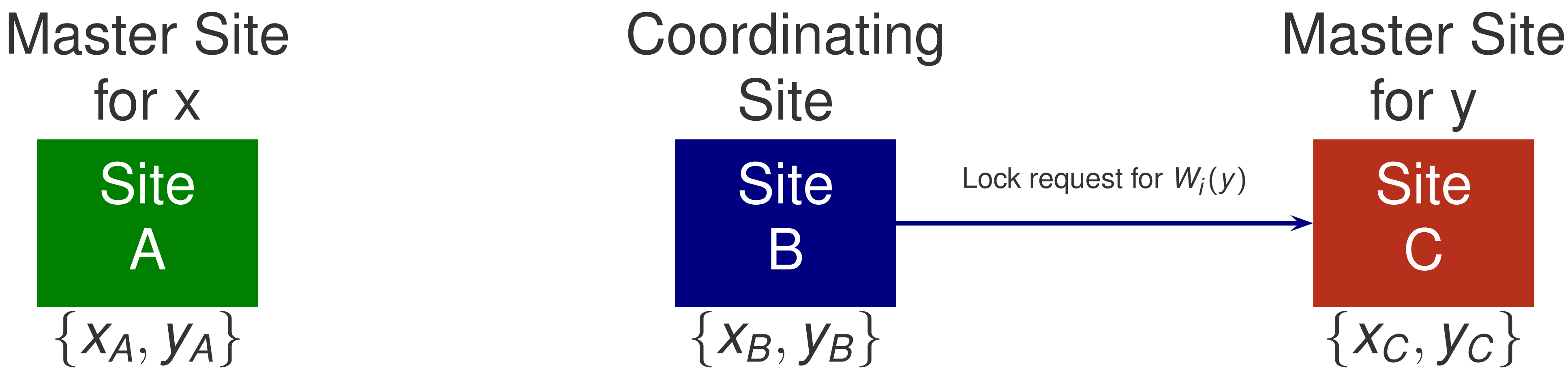
TM_B sends lock request for $R_i(x)$ to TM_A

Eager Primary Copy Protocol: Example



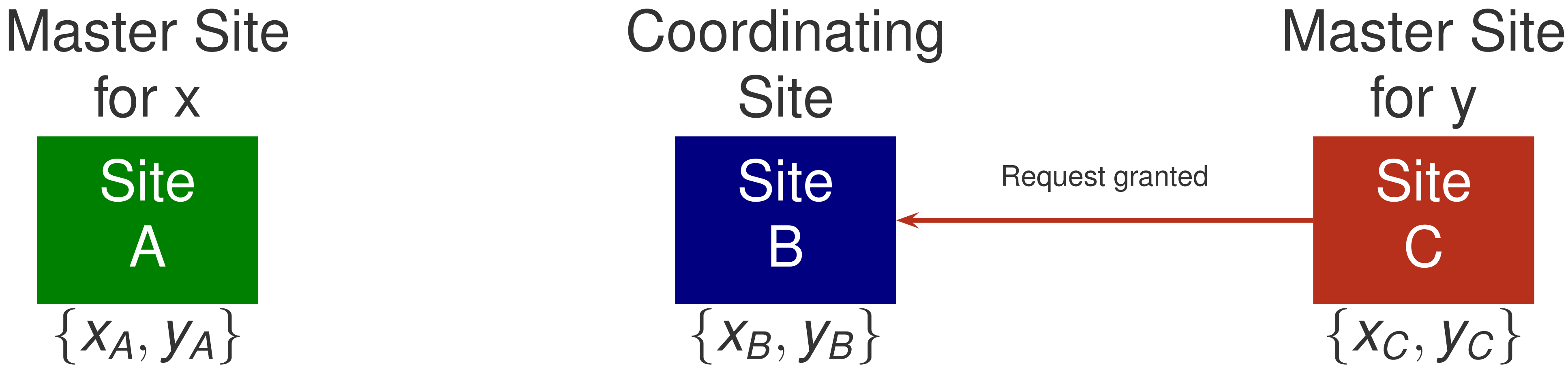
TM_A grants S-lock for $R_i(x)$ & notifies TM_B

Eager Primary Copy Protocol: Example



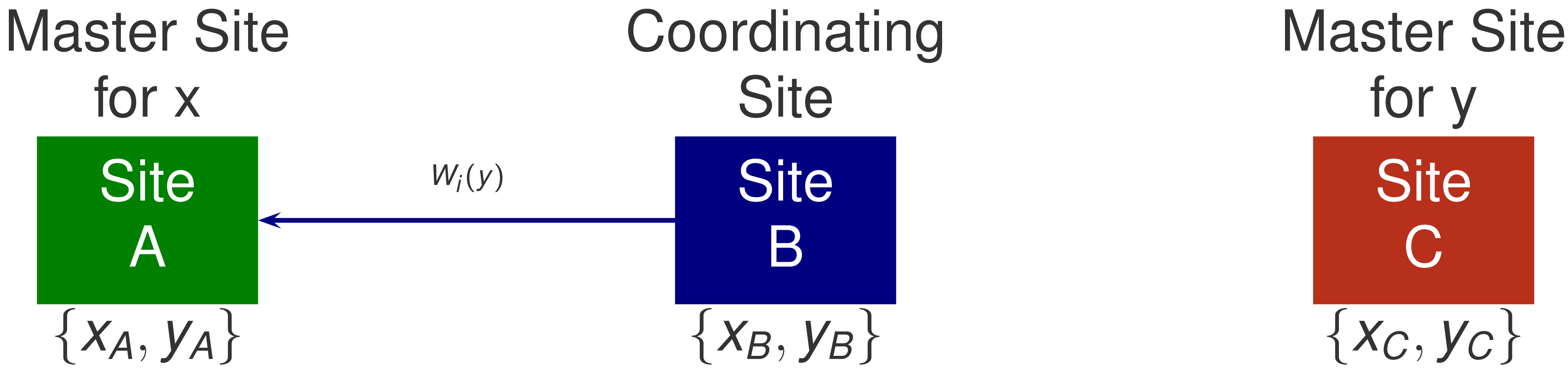
TM_B executes $R_i(x_B)$ &
sends lock request for $W_i(y)$ to TM_C

Eager Primary Copy Protocol: Example



TM_C grants X-lock for $W_i(y)$, executes $W_i(y_C)$ & notifies TM_B

Eager Primary Copy Protocol: Example



TM_B executes $W_i(y_B)$ & sends $W_i(y)$ to TM_A

Eager Primary Copy Protocol: Example

Master Site for x	Coordinating Site	Master Site for y
Site A	Site B	Site C
$\{x_A, y_A\}$	$\{x_B, y_B\}$	$\{x_C, y_C\}$

TM_A executes $W_i(y_A)$

Eager Distributed Protocols

- **Eager** = all replicas updated within context of Xact
- **Distributed** = any replica can be updated first
- Each site runs a lock manager
  - Controls lock requests/releases for its local replicas

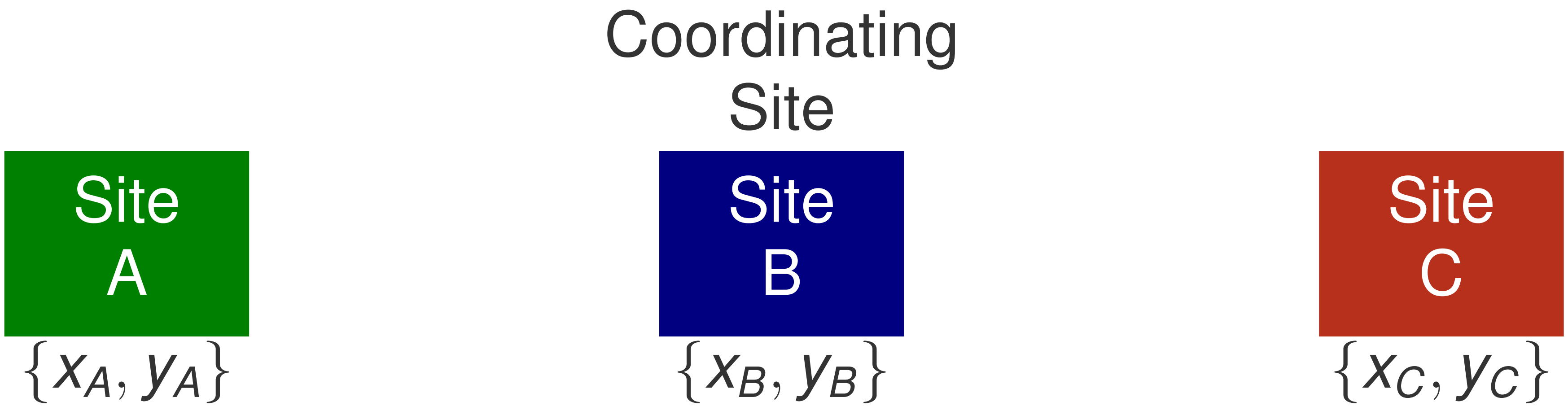
Eager Distributed Protocols (cont.)

- Consider a Xact T_i issued at site S_B
  - TM_B is the coordinating TM for T_i
- To process $R_i(x)$,
  - If site S_B has a local replica of x
    - ★ TM_B checks if S-lock on x can be granted for T_i
    - ★ If S-lock for $R_i(x)$ is granted, TM_B reads from its local replica of x (i.e., $R_i(x_B)$)
    - ★ Otherwise, T_i is blocked
  - Else
    - ★ TM_B sends $R_i(x)$ to any site (say S_C) with a copy of x
    - ★ If S-lock for $R_i(x)$ is granted by TM_C, TM_C reads x_C & returns x_C to TM_B
    - ★ Otherwise, T_i is blocked

Eager Distributed Protocols (cont.)

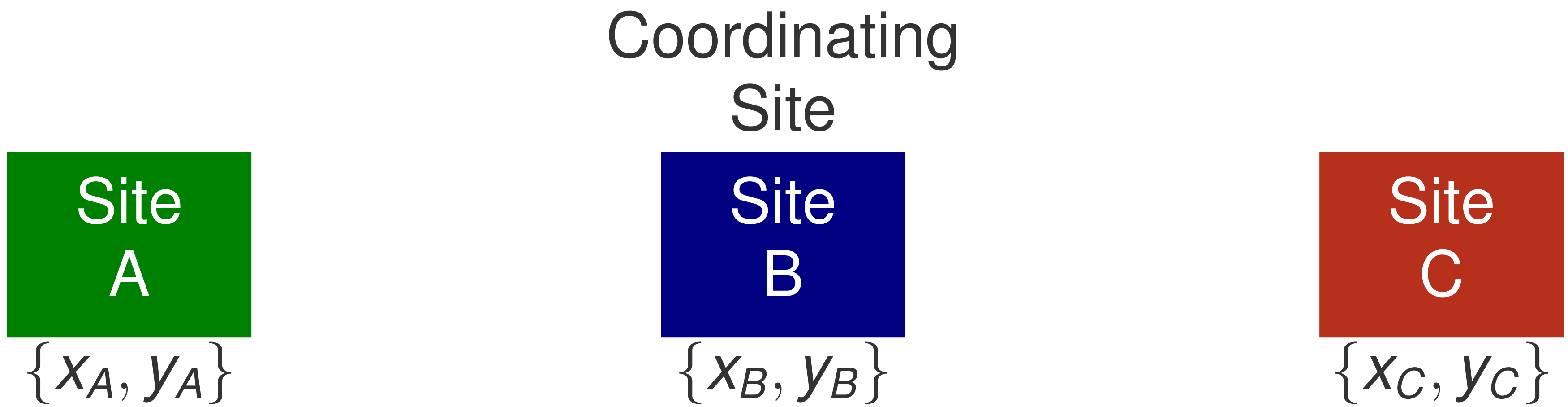
- To process $W_i(x)$,
  - If site S_B has a local replica of x
    - TM_B checks if X-lock on x can be granted for T_i
    - If X-lock for $W_i(x)$ is granted, TM_B updates its local replica of x (i.e., $W_i(x_B)$) & sends $W_i(x)$ to other sites with replicas of x
    - Otherwise, T_i is blocked
  - Else
    - TM_B sends $W_i(x)$ to any site (say S_C) with a copy of x
    - If X-lock for $W_i(x)$ is granted by TM_C, TM_C updates x_C & notifies TM_B that request is granted. TM_B sends $W_i(x)$ to other sites with replicas of x
    - Otherwise, T_i is blocked

Eager Distributed Protocol: Example



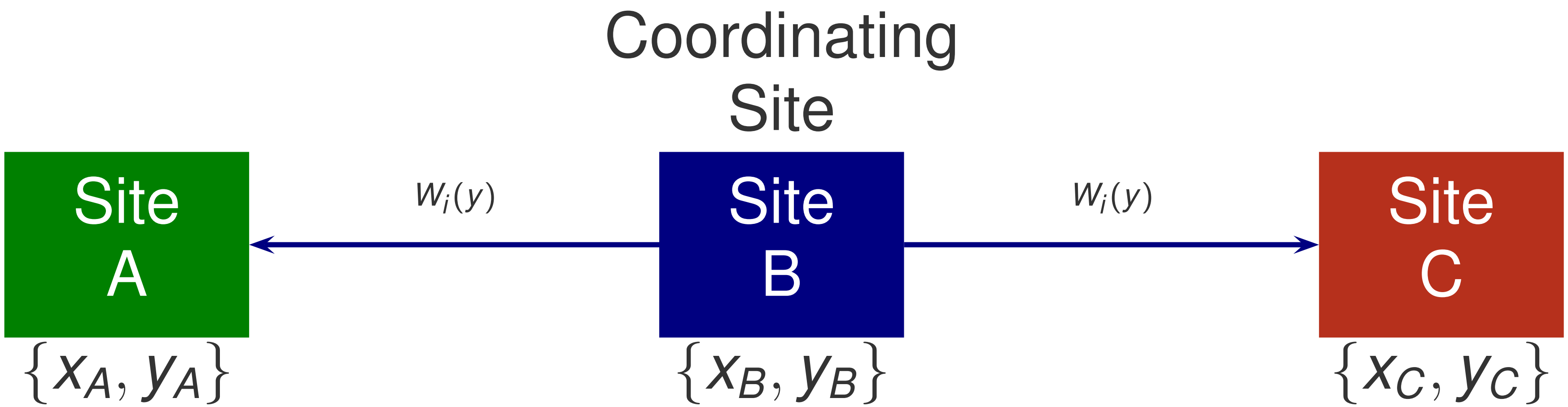
Transaction T_i needs to read x & write y
Site B is the coordinating site for Xact T_i

Eager Distributed Protocol: Example



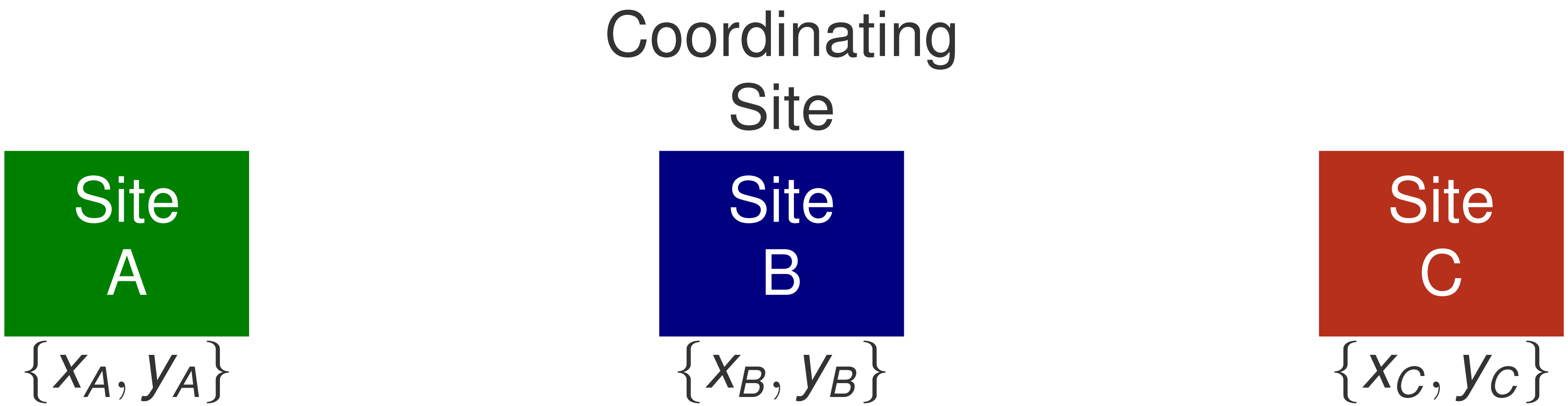
For $R_i(x)$, TM_B grants S-lock for $R_i(x)$ & executes $R_i(x_B)$

Eager Distributed Protocol: Example



For $W_i(y)$, TM_B grants X-lock for $W_i(y)$, executes $W_i(y_B)$ & sends $W_i(y)$ to TM_A & TM_C

Eager Distributed Protocol: Example



TM_A grants X-lock for $W_i(y)$ & executes $W_i(y_A)$
TM_C grants X-lock for $W_i(y)$ & executes $W_i(y_C)$

Lazy Centralized Protocols

- **Lazy** = Only one replica is updated within context of Xact; other replicas are updated asynchronously
  - Refresh Xacts sent to other replica sites after update Xact commits
- **Centralized** = master copy is updated before slave copies

Lazy Single-Master Protocol

- There is a single master site containing master copies of all objects
- For each update operation $W_i(O)$,
  - Master copy of O at master site must be updated first
  - Updates are then propagated asynchronously to other copies of O after T_i commits
- For each read operation $R_i(O)$,
  - Coordinating TM for T_i can read from any one replica of O

Lazy Single-Master Protocol (cont.)

- Site S_A is the master site
- Consider a Xact T_i issued at site S_B
  - TM_B is the coordinating TM for T_i
- To process $R_i(x)$, TM_B sends lock request for $R_i(x)$ to TM_A
- TM_A checks if S-lock on x can be granted to T_i
- If granted,
  - TM_A notifies TM_B that lock request is granted
  - If TM_B has a replica of x, TM_B reads its local copy of x (i.e., $R_i(x_B)$);
  - Otherwise, TM_B sends $R_i(x)$ to any site (say S_C) with a replica of x
    - TM_C executes $R_i(x_C)$ & returns x_C to TM_B
- Otherwise, T_i is blocked

Lazy Single-Master Protocol (cont.)

- To process $W_i(x)$, TM_B sends lock request for $W_i(x)$ to TM_A
- TM_A checks if X-lock on x can be granted to T_i
- If granted,
  - TM_A updates its copy of x (i.e., $W_i(x_A)$)
  - TM_A notifies TM_B that lock request is granted
- Otherwise, T_i is blocked

Lazy Single-Master Protocol (cont.)

- When T_i commits, TM_B sends $Commit_i$ to TM_A
- TM_A executes $Commit_i$ & releases locks for T_i
- TM_A checks if X-locks can be granted for T_i's refresh transactions
- If granted, TM_A sends refresh transactions to other sites to propagate T_i's updates; otherwise, the sending of refresh transactions is blocked

Refresh Transactions

- Let T_i^r denote the refresh transaction for T_i
- Each T_i^r is executed as a local refresh transaction at a slave site
- Important for refresh Xacts to be applied to all slave sites in the same order
  - If $TS(T_i^r)$ & $TS(T_j^r)$ are both required to be executed at multiple sites, then they must be executed in the same order at these sites
- Slave site applies refresh transactions in timestamp order as follows
  - Each T_i^r has a timestamp denoted by $TS(T_i^r)$
  - $TS(T_i^r) = commitTS(T_i)$ which is the commit timestamp of T_i
    - ★ T_i commits before T_j iff $commitTS(T_i) < commitTS(T_j)$
  - T_i^r is executed before T_j^r at a site iff $TS(T_i^r) < TS(T_j^r)$

Refresh Transactions: Example

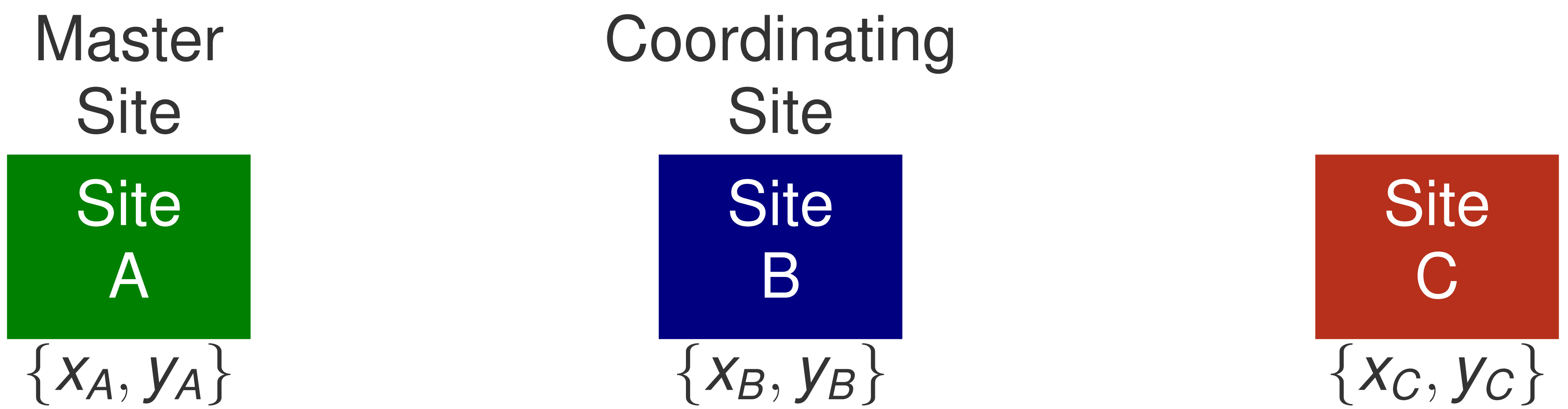
- DDBMS: Site A = $\{x, y\}$, Site B = $\{x\}$, Site C = $\{x, y\}$
- Site A is the master site
- TM_B is the coordinator for the execution of T_i
  - T_i: $R_i(x)$, $R_i(y)$, $W_i(x)$, $W_i(y)$
  - $\mathbf{T_i^r}$ consists of the set of updates $\{\mathbf{W_i^r(x)}, \mathbf{W_i^r(y)}\}$
- Local Schedules:

  S_A: $W_i(x_A), W_i(y_A), C_i$

  S_B: $R_i(x_B),$ $W_i^r(x_B), C_i^r$

  S_C: $R_i(y_C),$ $W_i^r(x_C), W_i^r(y_C), C_i^r$
- Note: C_i & C_i^r denote the commit of T_i & T_i^r, respectively

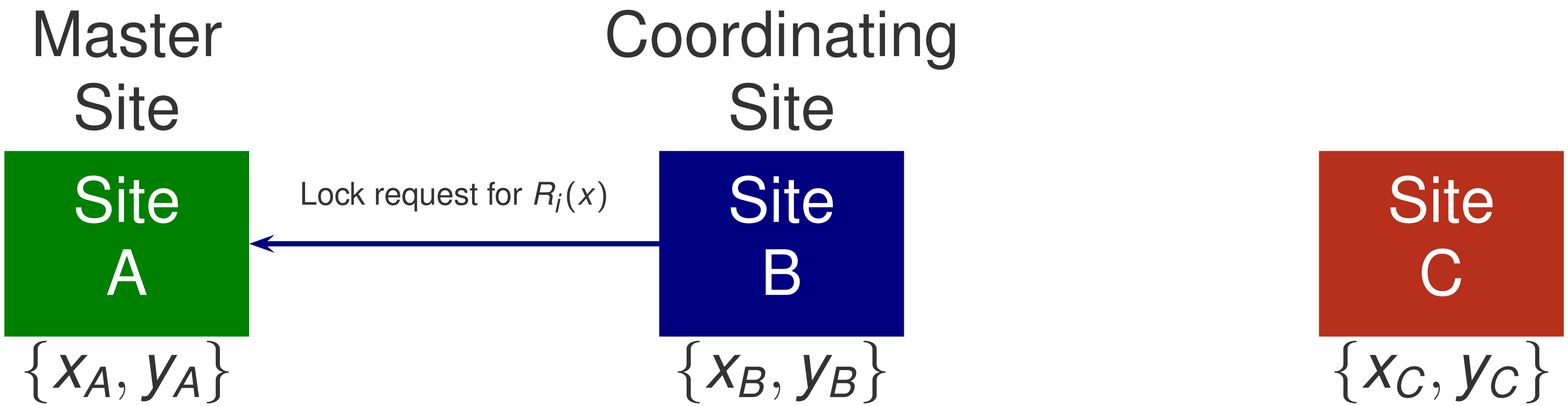
Lazy Single-Master Protocol: Example



Transaction T_i needs to read x & write y

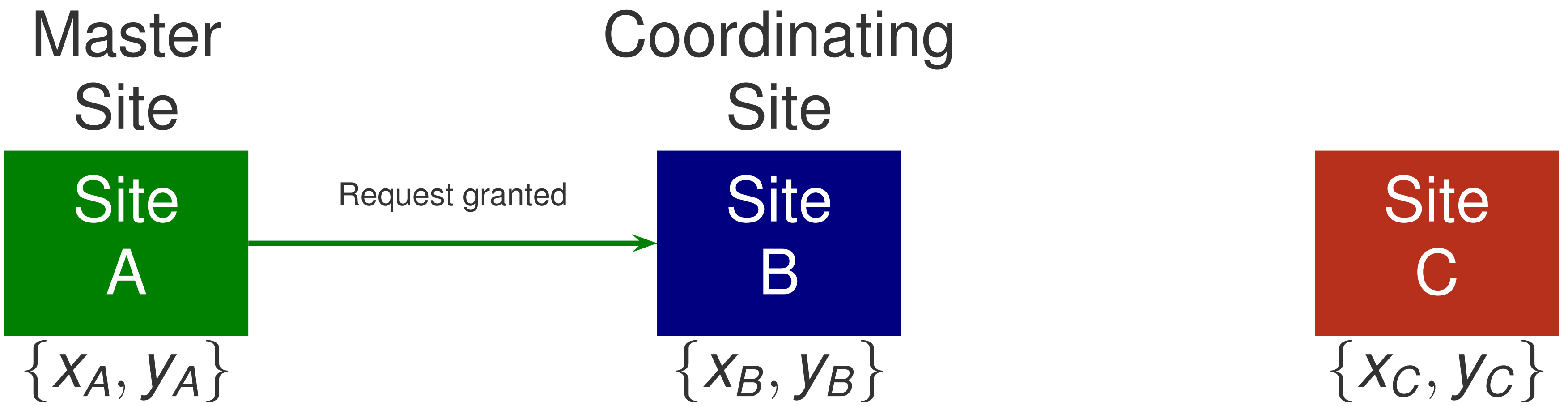
Site B is the coordinating site for Xact T_i

Lazy Single-Master Protocol: Example



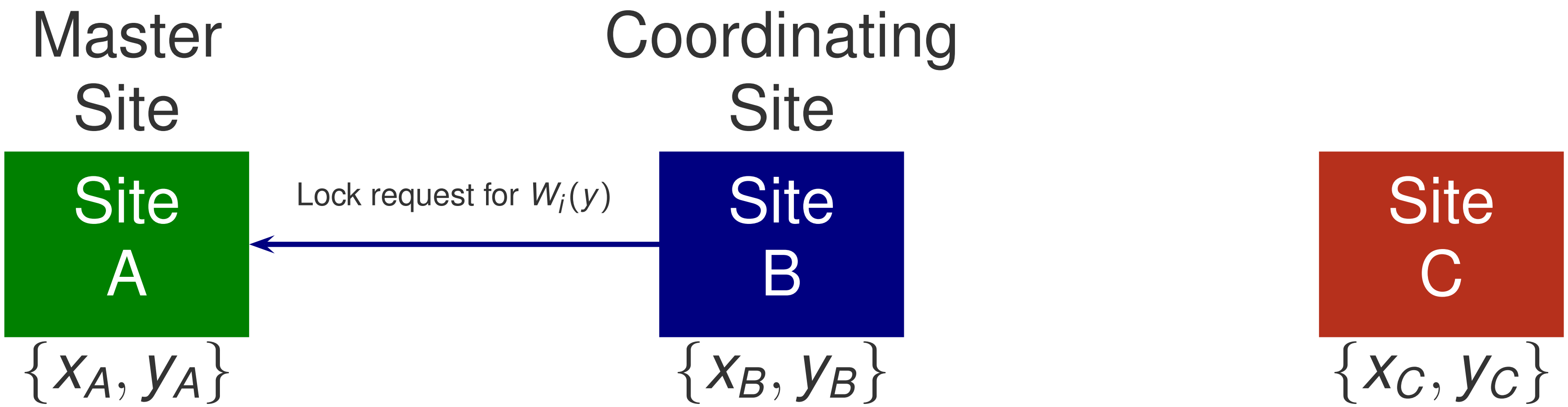
TM_B sends lock request for $R_i(x)$ to TM_A

Lazy Single-Master Protocol: Example



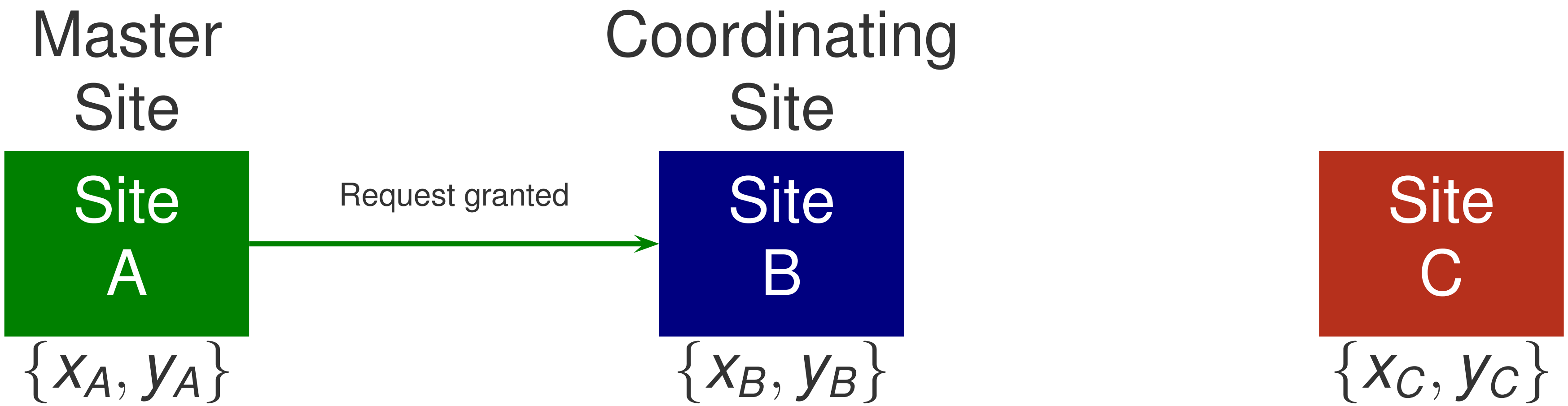
TM_A grants S-lock for $R_i(x)$ & notifies TM_B

Lazy Single-Master Protocol: Example



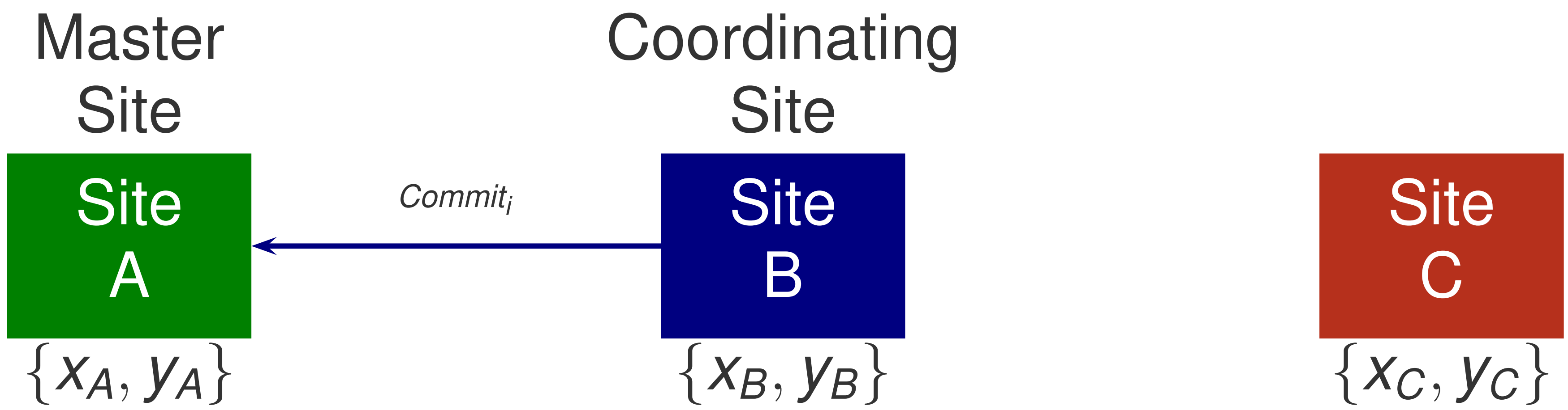
TM_B executes $R_i(x_B)$ & sends lock request for $W_i(y)$ to TM_A

Lazy Single-Master Protocol: Example



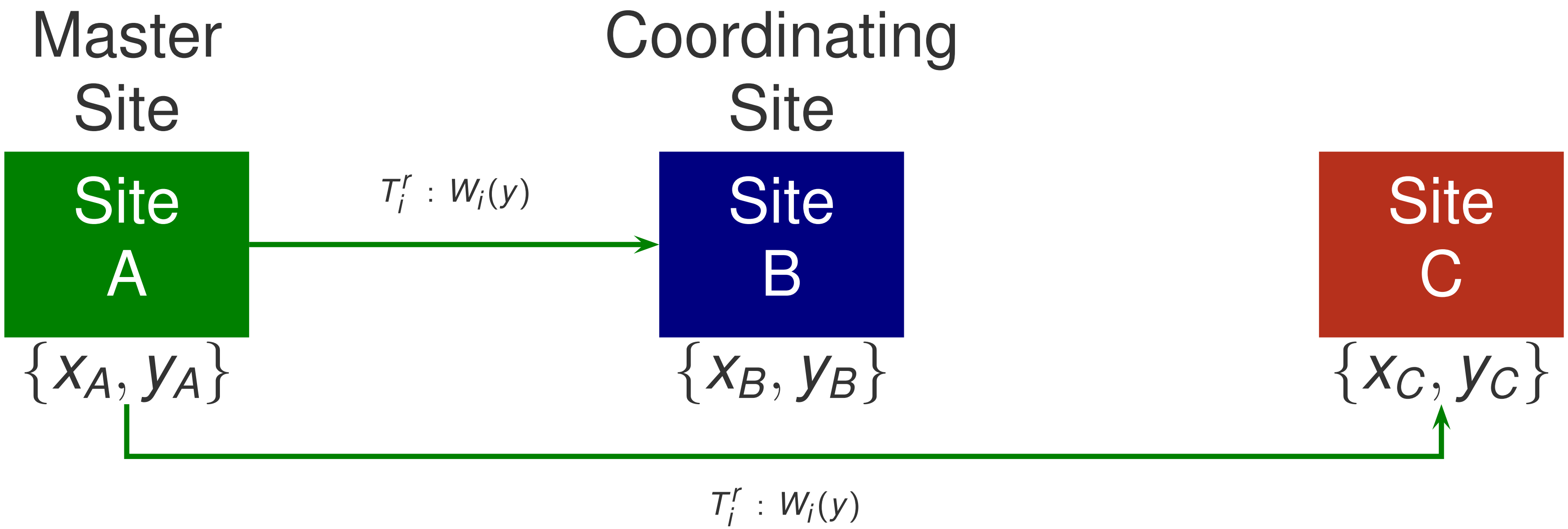
TM_A grants X-lock for $W_i(y)$, executes $W_i(y_A)$ & notifies TM_B

Lazy Single-Master Protocol: Example



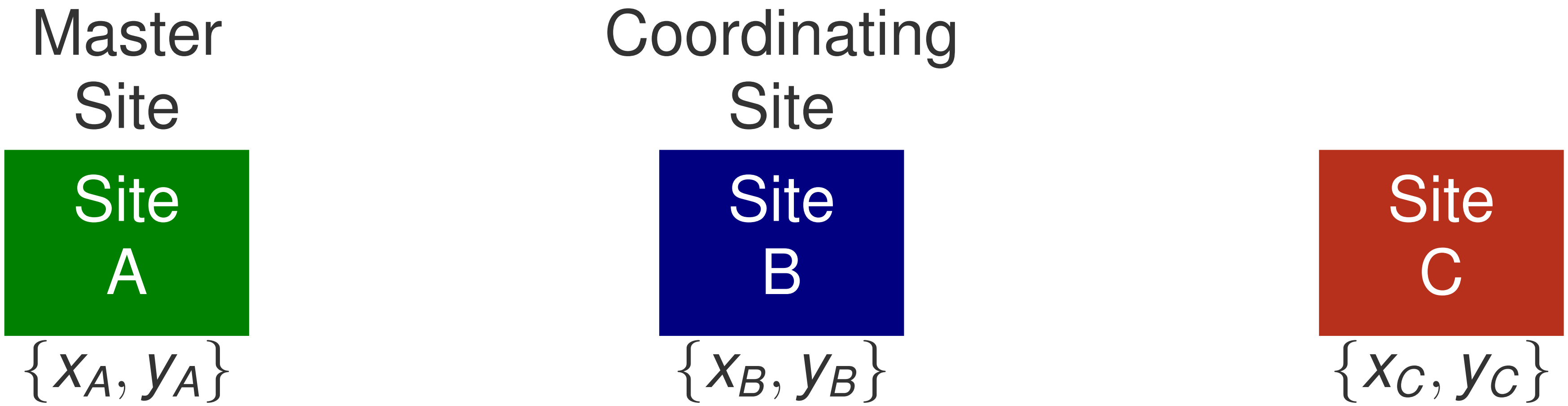
TM_B sends $Commit_i$ to TM_A

Lazy Single-Master Protocol: Example



TM_A executes $Commit_i$, releases locks for T_i, grants X-lock for $W_i^r(y)$, & sends T_i^r: $W_i(y)$ to TM_B & TM_C

Lazy Single-Master Protocol: Example



TM_B executes $W_i^r(y_B)$
TM_C executes $W_i^r(y_C)$

Example 1

- Site A contains master copies of $\{x, y\}$
- Site B contains slave copies of $\{x, y\}$
- Transactions
  - T_1: $R_1(x)$, $W_1(y)$
  - T_2: $W_2(x)$, $W_2(y)$
- Assume T_1 issued at Site A & T_2 issued at Site B
- Local schedules:

S_A: $R_1(x_A), W_1(y_A), C_1, W_2(x_A), W_2(y_A), C_2$
S_B: $W_1^r(y_B), C_1^r, W_2^r(x_B), W_2^r(y_B), C_2^r$

- Note that $TS(T_1^r) < TS(T_2^r)$

Example 2

- Site A contains master copies of $\{x, y\}$
- Site B contains slave copies of $\{x, y\}$
- Transactions
  - T_1: $R_1(x)$, $W_1(y)$
  - T_2: $W_2(x)$, $W_2(y)$
- Assume T_1 & T_2 are issued at Site B
- Local schedules:

$$
\begin{array}{lllllll}
S_A: & W_2(x_A), W_2(y_A), C_2, & & W_1(y_A), & C_1 & \\
S_B: & & R_1(x_B), & & & W_2^r(x_B), W_2^r(y_B), C_2^r, W_1^r(y_B), C_1^r
\end{array}
$$

- Note that $TS(T_2^r) < TS(T_1^r)$

Example 3

- Site A contains master copy of x
- Site B contains slave copy of x
- Transactions:
  - T_1: $W_1(x)$, $R_1(x)$
- Assume T_1 issued at Site B
- Local schedules:

$$
\begin{array}{llll}
S_A: W_1(x_A), & & C_1 & \\
S_B: & R_1(x_B), & & W_1^r(x_B), C_1^r
\end{array}
$$

Lazy Distributed Protocols

- **Lazy** = Only one replica is updated within context of Xact; other replicas are updated asynchronously
- **Distributed** = any replica can be updated first
- Each site runs a lock manager
  - Controls lock requests/releases for its local replicas

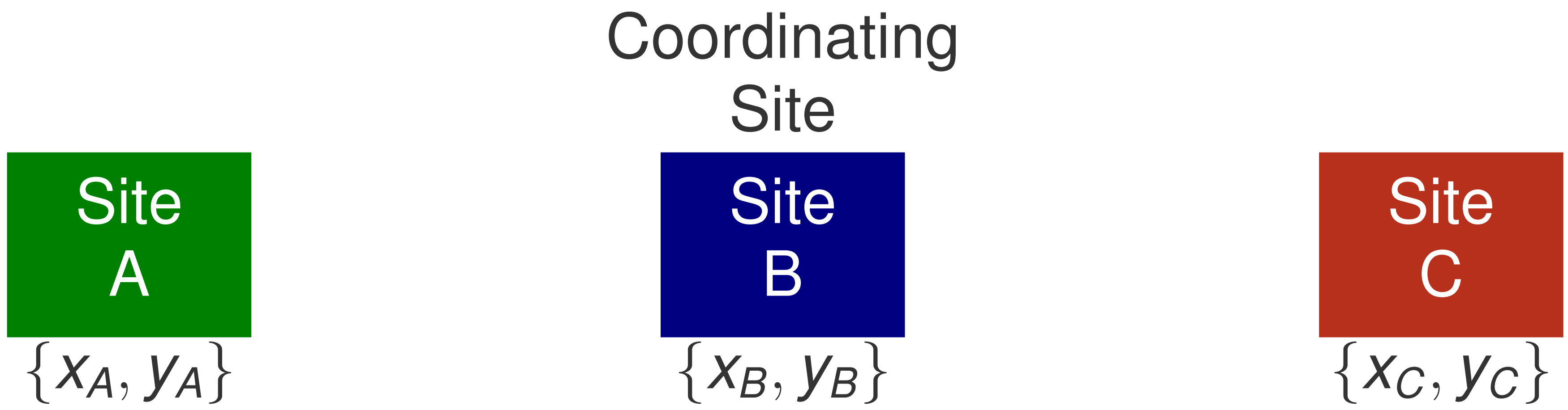
Lazy Distributed Protocols (cont.)

- Consider a Xact T_i issued at site S_B
  - TM_B is the coordinating TM for T_i
- To process $R_i(x)$,
  - If site S_B has a local replica of x
    - ★ TM_B checks if S-lock on x can be granted for T_i
    - ★ If S-lock for $R_i(x)$ is granted, TM_B reads from its local replica of x (i.e., $R_i(x_B)$)
    - ★ Otherwise, T_i is blocked
  - Else
    - ★ TM_B sends $R_i(x)$ to any site (say S_C) with a copy of x
    - ★ If S-lock for $R_i(x)$ is granted by TM_C, TM_C executes $R_i(x_C)$ & returns x_C to TM_B
    - ★ Otherwise, T_i is blocked

Lazy Distributed Protocols (cont.)

- To process $W_i(x)$,
  - If site S_B has a local replica of x
    - TM_B checks if X-lock on x can be granted for T_i
    - If X-lock for $W_i(x)$ is granted, TM_B updates its local replica of x (i.e., $W_i(x_B)$)
    - Otherwise, T_i is blocked
  - Else
    - TM_B sends $W_i(x)$ to any site (say S_C) with a copy of x
    - If X-lock for $W_i(x)$ is granted by TM_C, TM_C updates x_C (i.e., $W_i(x_C)$)
    - Otherwise, T_i is blocked
- When T_i commits,
  - TM_B executes $Commit_i$, releases locks for T_i, & sends refresh transactions to other sites to propagate T_i's updates

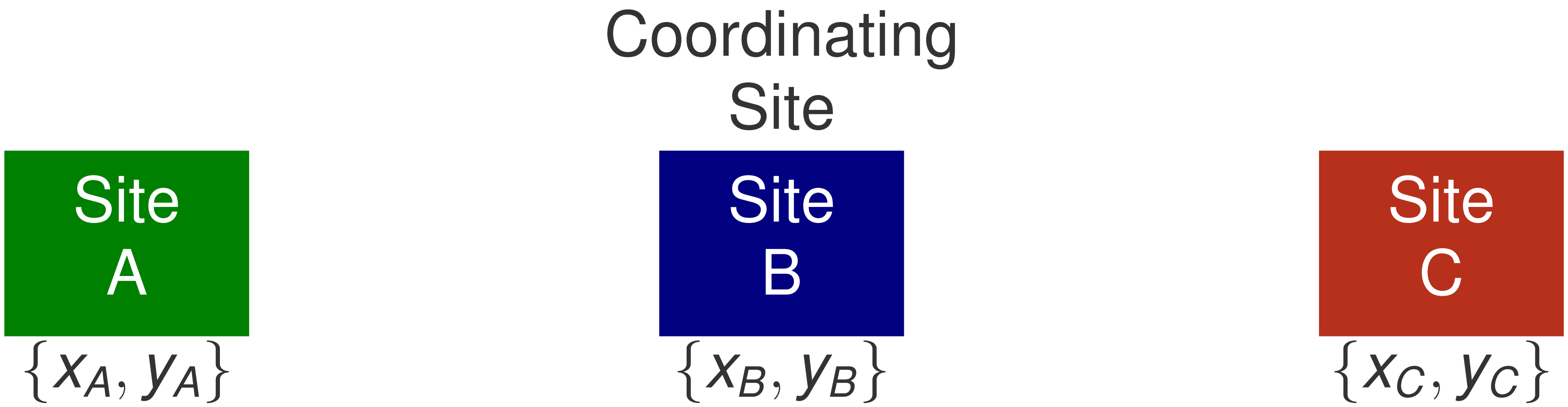
Lazy Distributed Protocol: Example



Transaction T_i needs to read x & write y

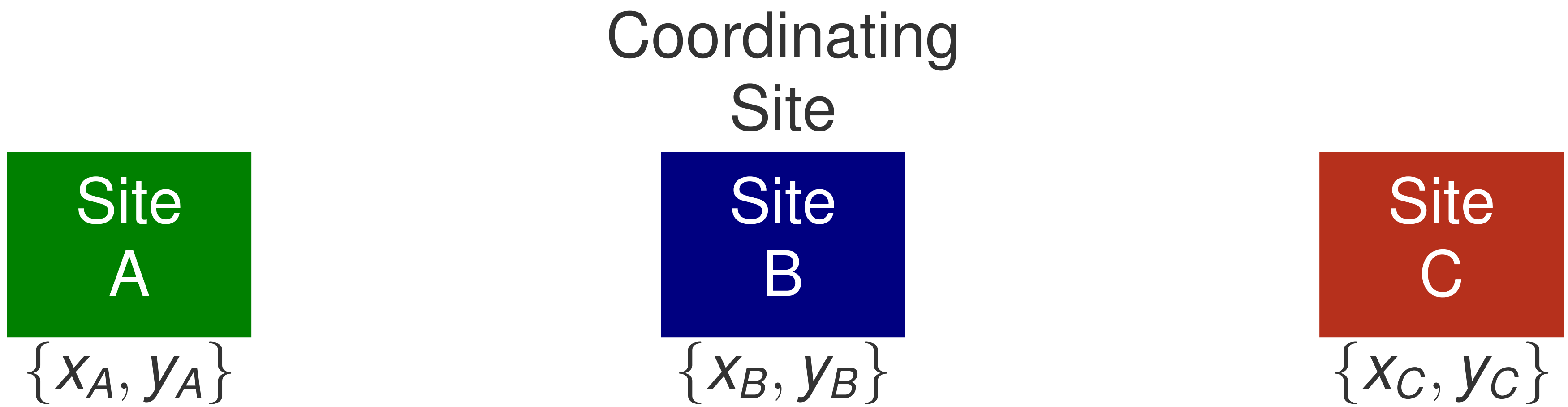
Site B is the coordinating site for Xact T_i

Lazy Distributed Protocol: Example



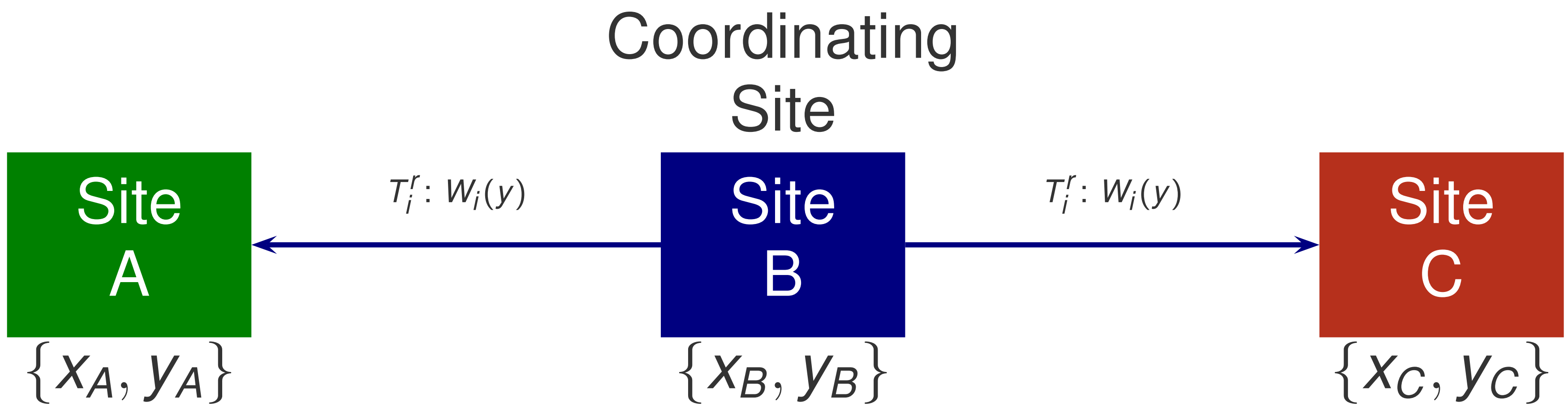
For $R_i(x)$, TM_B grants S-lock for $R_i(x)$ & executes $R_i(x_B)$

Lazy Distributed Protocol: Example



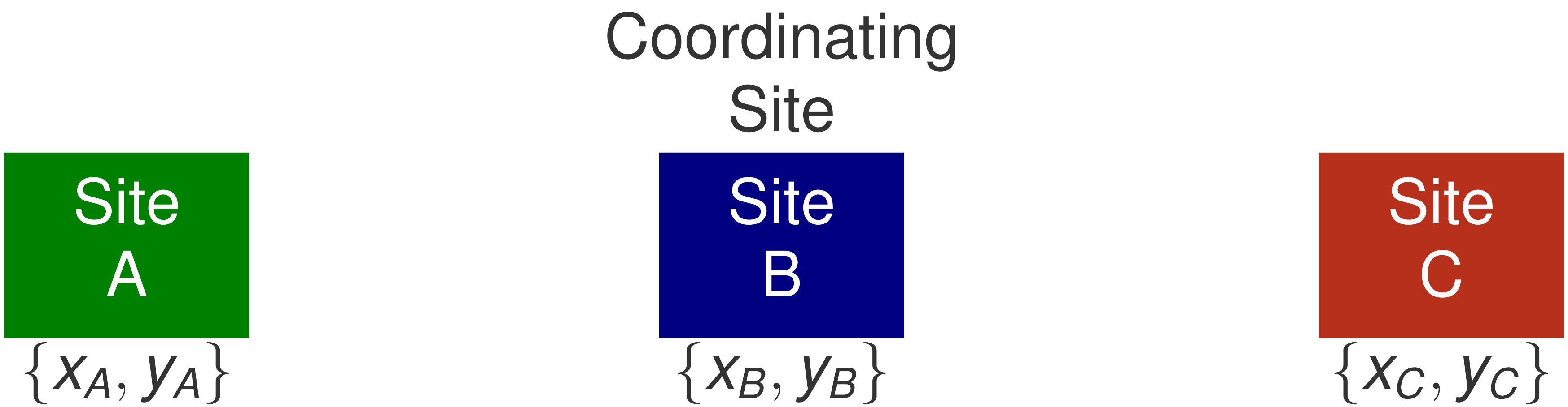
For $W_i(y)$, TM_B grants X-lock for $W_i(y)$ & executes $W_i(y_B)$

Lazy Distributed Protocol: Example



TM_B executes $Commit_i$, releases locks for T_i & sends T_i^r: $W_i(y)$ to TM_A & TM_C

Lazy Distributed Protocol: Example



TM_A grants X-lock for $W_i^r(y)$ & executes $W_i^r(y_A)$
TM_C grants X-lock for $W_i^r(y)$ & executes $W_i^r(y_C)$

Reconciliation of Inconsistent Updates

- Multiple Xacts could update different copies of same data concurrently at different sites
  - Conflicting updates can occur!
- **Example**:

$$
\begin{array}{llll}
S_A: & W_1(x_A), & C_1, & W_2^r(x_A), C_2^r \\
S_B: & W_2(x_B), & C_2, & W_1^r(x_B), C_1^r
\end{array}
$$

- Requires reconciliation procedure
  - **Last-Writer-Wins heuristic** (a.k.a. timestamp order heuristic): apply updates in timestamp order

Last-Writer-Wins Heuristic

- Used to reconcile inconsistent updates
- Last-Writer-Wins Heuristic
  - If there are two concurrent updates $W_i(x)$ & $W_j(x)$, $W_i(x)$ wins if $TS(T_j^r) < TS(T_i^r)$
    - $W_j(x)$ is ignored if x was last updated by T_i & $TS(T_j^r) < TS(T_i^r)$
- **Example:** $x_A = x_B = 1$, $\mathrm{TS}(T_1^r) < \mathrm{TS}(T_2^r)$

Site A	Site B	Comments
$W_1(x_A, 10)$		$x_A = 10$
	$W_2(x_B, 20)$	$x_B = 20$
$Commit_1$	$Commit_2$	
	Receives $W_1^r(x_B, 10, TS(T_1^r))$	Xact is ignored
Receives $W_2^r(x_A, 20, TS(T_2^r))$		
$W_2^r(x_A, 20, TS(T_2^r))$ $Commit_2^r$		$x_A = 20$

Last-Writer-Wins Heuristic (cont.)

- Heuristic only works for updates that are blind writes
- An update $W(x)$ is a blind write if the new value of x is computed independent of its previous value

Example: Non-Blind Updates

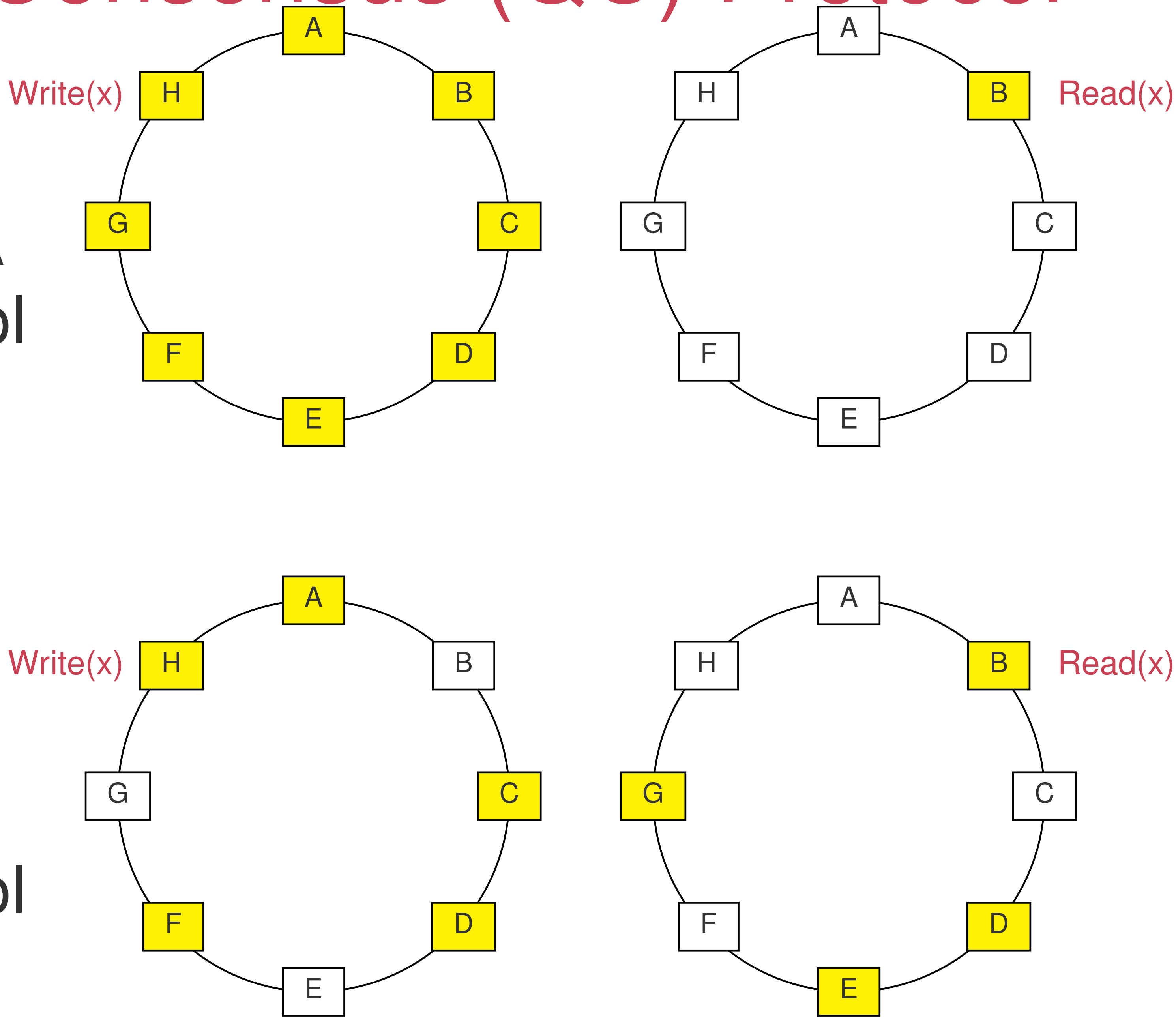
- T_1: $R_1(x)$, $x = x \times 100$, $W_1(x)$
- T_2: $R_2(x)$, $x = x + 10$, $W_2(x)$
- $x_A = 1$, $x_B = 1$, $\text{TS}(T_1^r) < \text{TS}(T_2^r)$

Site A	Site B	Comments
$R_1(x_A)$, $W_1(x_A, 100)$		$x_A = 100$
	$R_2(x_B)$, $W_2(x_B, 11)$	$x_B = 11$
$Commit_1$	$Commit_2$	
	Receives $W_1^r(x_B, 100, TS(T_1^r))$	Xact is ignored
Receives $W_2^r(x_A, 11, TS(T_2^r))$		
$W_2^r(x_A, 11, TS(T_2^r))$ $Commit_2^r$		$x_A = 11$

Quorum Consensus (QC) Protocol

ROWA Protocol

QC Protocol



Quorum Consensus (QC) Protocol

- Each copy O_i of an object O is assigned a non-negative weight, $Wt(O_i)$
- Wt(O) = total weight of all copies of O

$$Wt(O) = \sum_{O_i \in S} Wt(O_i), \quad S = \text{set of all copies of } O$$

- Each object O is associated with two thresholds:
  - Read threshold $T_r(O)$
  - Write threshold $T_w(O)$

Quorum Consensus (QC) Protocol (cont.)

- A read quorum for object O, $Q_r(O)$, is a subset of copies of O such that

$$\sum_{O_i \in Q_r(O)} Wt(O_i) \geq T_r(O)$$

- A write quorum for object O, $Q_w(O)$, is a subset of copies of O such that

$$\sum_{O_i \in Q_w(O)} Wt(O_i) \geq T_w(O)$$

- The read & write thresholds for object O satisfy the following constraints:
    1. $T_r(O) + T_w(O) > Wt(O)$
    2. $2 \times T_w(O) > Wt(O)$

Quorum Consensus (QC) Protocol (cont.)

- Each object copy is associated with a version number to indicate how up-to-date its value is
  - Higher version number means more up-to-date
  - Version number is initialized to 0

Quorum Consensus (QC) Protocol (cont.)

- To read an object O,
  - Acquire S-locks on a read quorum for O, $Q_r(O)$
  - Read all copies in $Q_r(O)$ and return the copy with the highest version number
- To write an object O,
  - Acquire X-locks on a write quorum for O, $Q_w(O)$
  - Let n be the highest version number among all copies in $Q_w(O)$
  - Write all copies in $Q_w(O)$ and update their version numbers to $n+1$

QC Protocol: Example

Replica	Weight	Value	Version
x_A	1	10	0
x_B	1	10	0
x_C	4	10	0
x_D	2	10	0
x_E	3	10	0
x_F	1	10	0
x_G	1	10	0

$$Wt(x) = 13,\ T_r(x) = 5,\ T_w(x) = 9$$

QC Protocol: Example (cont.)

Operation: $W_1(x, 15)$

Replica	Weight	Value	Version
x_A	1	10	0
x_B	1	10	0
x_C	4	$\cancel{10}$ 15	$\cancel{0}$ 1
x_D	2	$\cancel{10}$ 15	$\cancel{0}$ 1
x_E	3	$\cancel{10}$ 15	$\cancel{0}$ 1
x_F	1	10	0
x_G	1	10	0

$$Wt(x) = 13,\ T_r(x) = 5,\ T_w(x) = 9$$

QC Protocol: Example (cont.)

Operation: $R_2(x)$

Replica	Weight	Value	Version
x_A	1	10	0
x_B	1	10	0
x_C	4	15	1
x_D	2	15	1
x_E	3	15	1
x_F	1	10	0
x_G	1	10	0

$$Wt(x) = 13,\ T_r(x) = 5,\ T_w(x) = 9$$

QC Protocol: Example (cont.)

Operation: $R_3(x)$

Replica	Weight	Value	Version
x_A	1	10	0
x_B	1	10	0
x_C	4	15	1
x_D	2	15	1
x_E	3	15	1
x_F	1	10	0
x_G	1	10	0

$$Wt(x) = 13,\ T_r(x) = 5,\ T_w(x) = 9$$

QC Protocol: Example (cont.)

Operation: $W_3(x, 30)$

Replica	Weight	Value	Version
x_A	1	10	0
x_B	1	$\cancel{10}$ 30	$\cancel{0}$ 2
x_C	4	$\cancel{15}$ 30	$\cancel{1}$ 2
x_D	2	$\cancel{15}$ 30	$\cancel{1}$ 2
x_E	3	$\cancel{15}$ 30	$\cancel{1}$ 2
x_F	1	10	0
x_G	1	10	0

$$Wt(x) = 13,\ T_r(x) = 5,\ T_w(x) = 9$$

QC Protocol: Example (cont.)

Operation: $R_5(x)$

Replica	Weight	Value	Version
x_A	1	10	0
x_B	1	30	2
x_C	4	30	2
x_D	2	30	2
x_E	3	30	2
x_F	1	10	0
x_G	1	10	0

$$Wt(x) = 13,\ T_r(x) = 5,\ T_w(x) = 9$$

Handling Failures

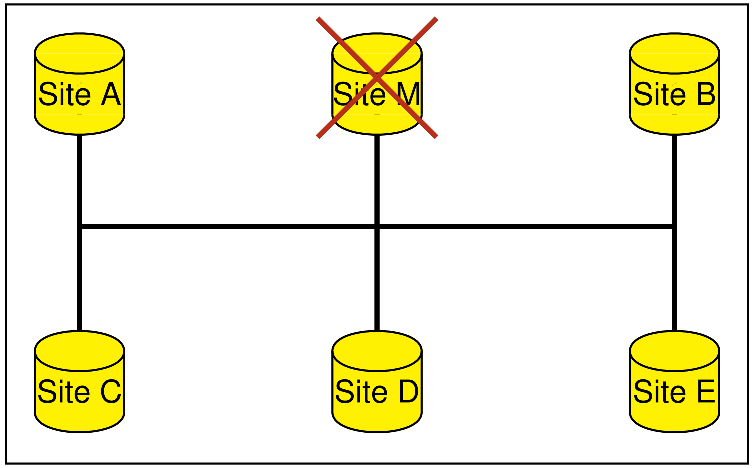
- Detect site failures using timeout mechanism
- Lecture focuses on centralized replication protocol
  - Single-master replication

Failure of Slave Sites

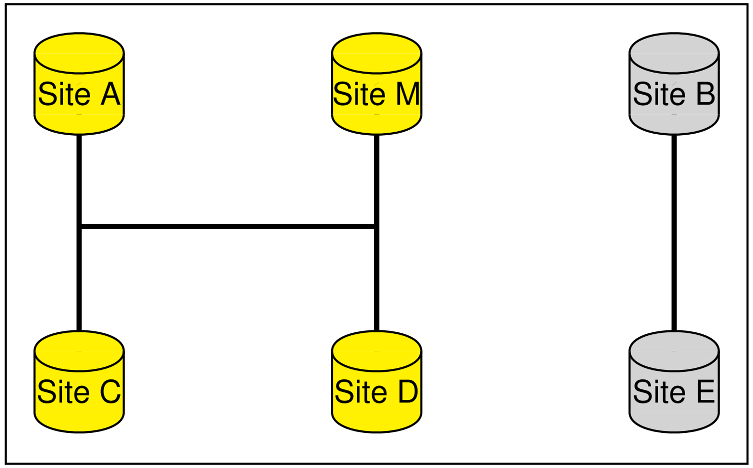
- Suppose some slave site(s) have failed
- Lazy replication
    - Synchronize unavailable replicas later when they become available
- Eager replication
    - Eager replication techniques are based on ROWA protocol
    - Drawback of ROWA: Update Xact can't terminate even if one replica is unavailable
    - Read-One/Write-All Available (ROWAA) protocol
        - Relax ROWA protocol to increase availability
        - Update all available replicas & terminate Xact
        - Synchronize unavailable replicas later when they become available

Failure of Master Site

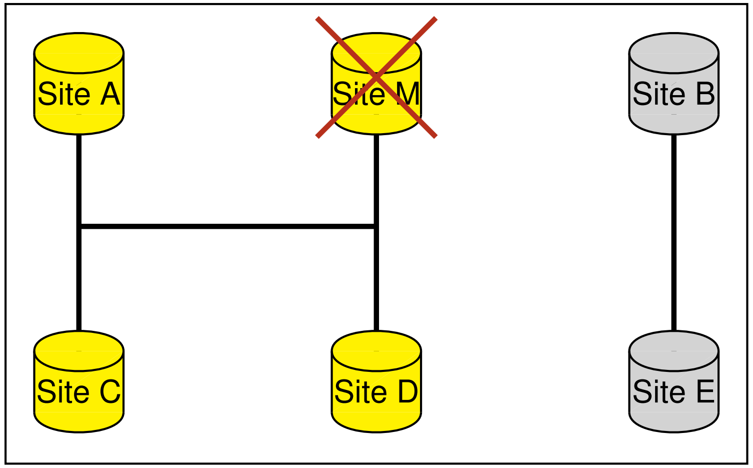
Suppose Site B detects the failure of master site M



Case 1: Master site has indeed failed



Case 2: Partitioned network

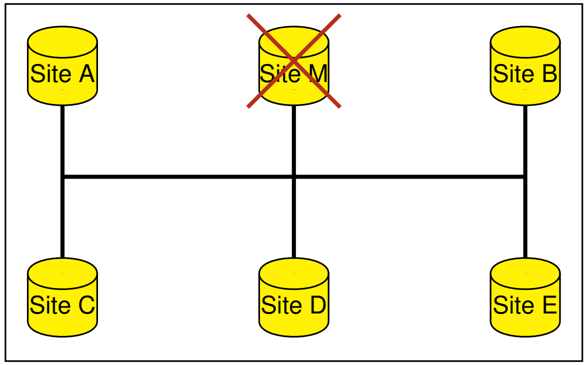


Case 3: Master site has failed & Partitioned network

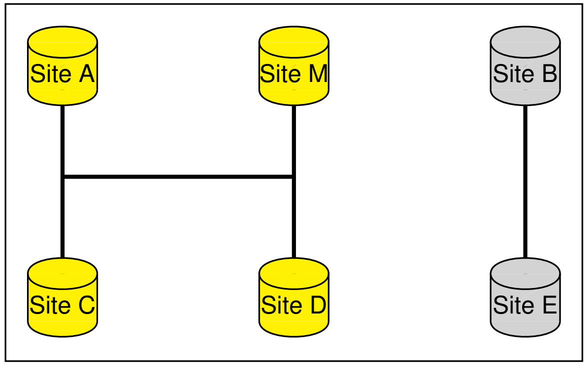
Failure of Master Site (cont.)

Options:

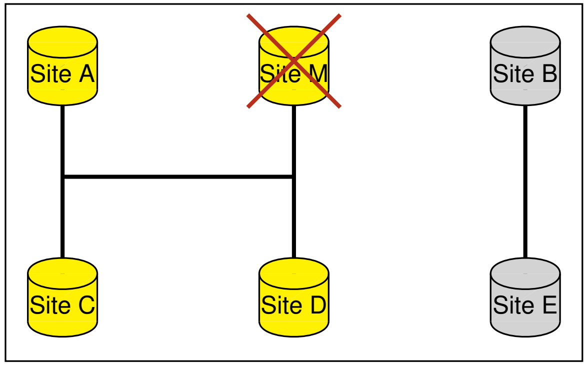
- Wait for recovery of master site / network
  - Not good for availability
- Elect a new master site
  - Need to ensure at most one partition of replicas has an operational master site



Case 1



Case 2



Case 3

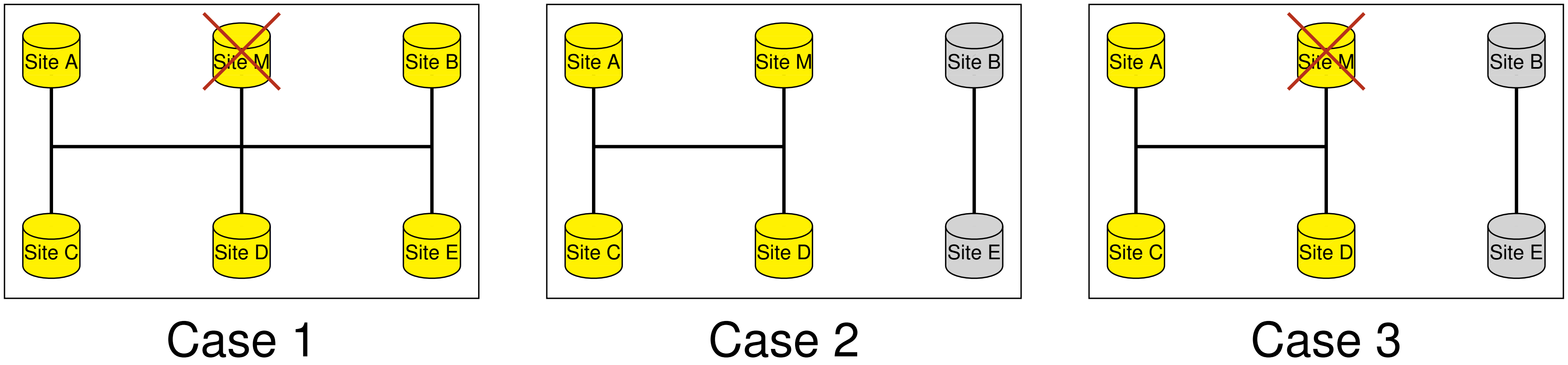
Failure of Master Site (cont.)

Electing a new master site

1. Choose a partition of replicas *P* to remain available
    - Simple algorithm
    - Majority consensus algorithm
    - Quorum consensus algorithm
2. If *P* does not contain an operational master site, elect a new master site in *P*
    - Consensus algorithm

Simple Algorithm

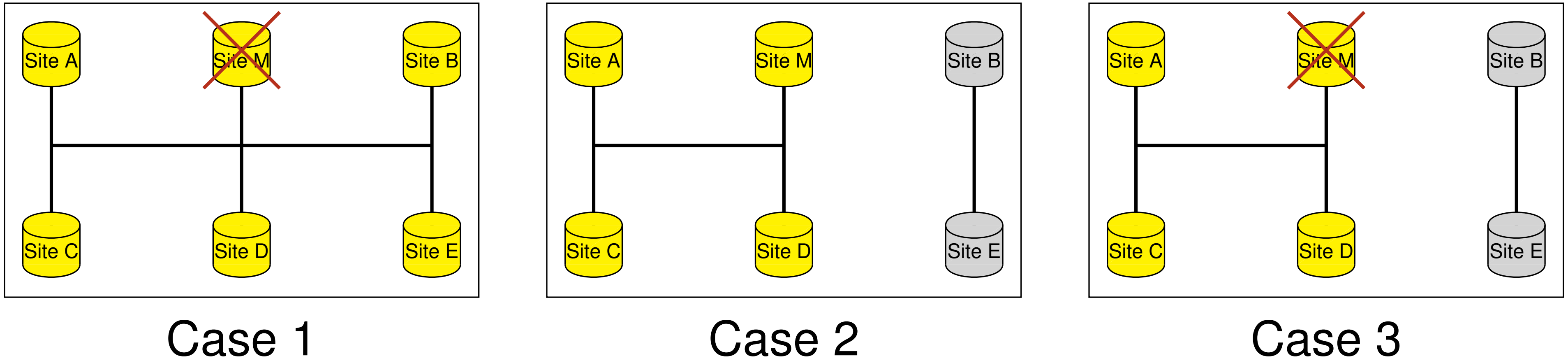
- The partition of replicas that contains an operational master site continues to be available



- **Cases 1 & 3**: System becomes unavailable
- **Case 2**: System remains available with the partition containing site M operational

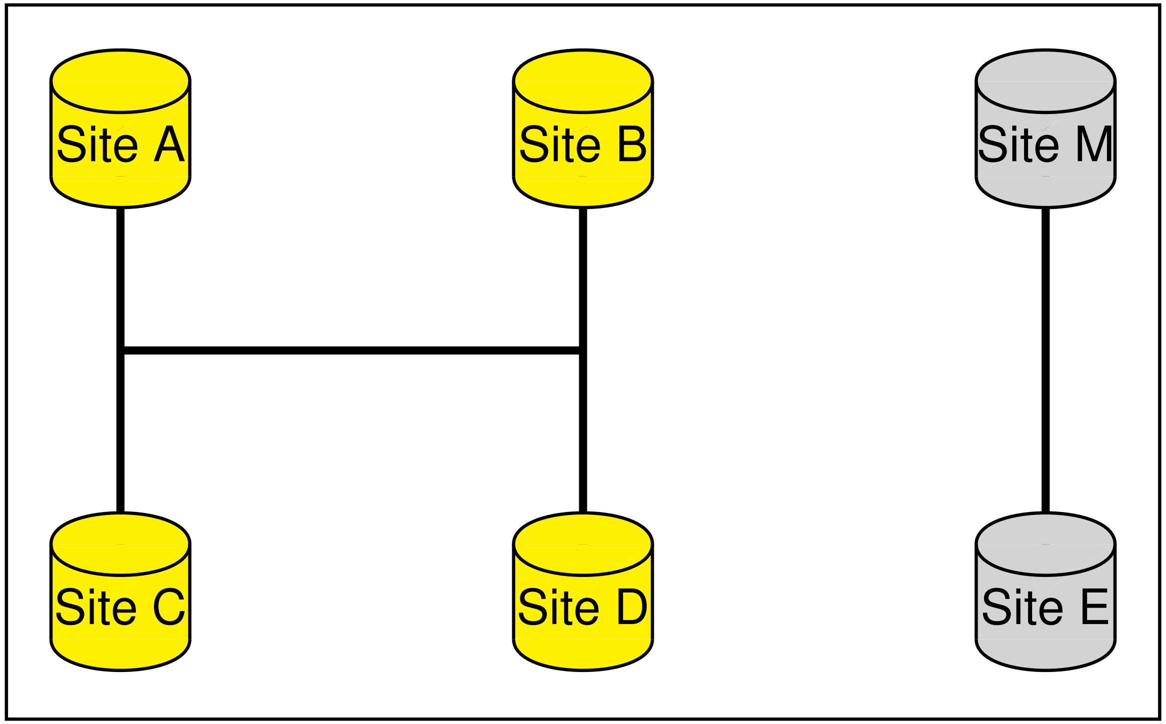
Majority Consensus Algorithm

- A partition of replicas is allowed to have a master site iff the partition includes a majority of the replicas
- Majority means more than $\lfloor \frac{N}{2} \rfloor$, where N is the number of replicas



- **Cases 1 & 2**: Majority partition remains operational after election of a new master site
- **Case 3**: System becomes unavailable

Majority Consensus Algorithm (cont.)

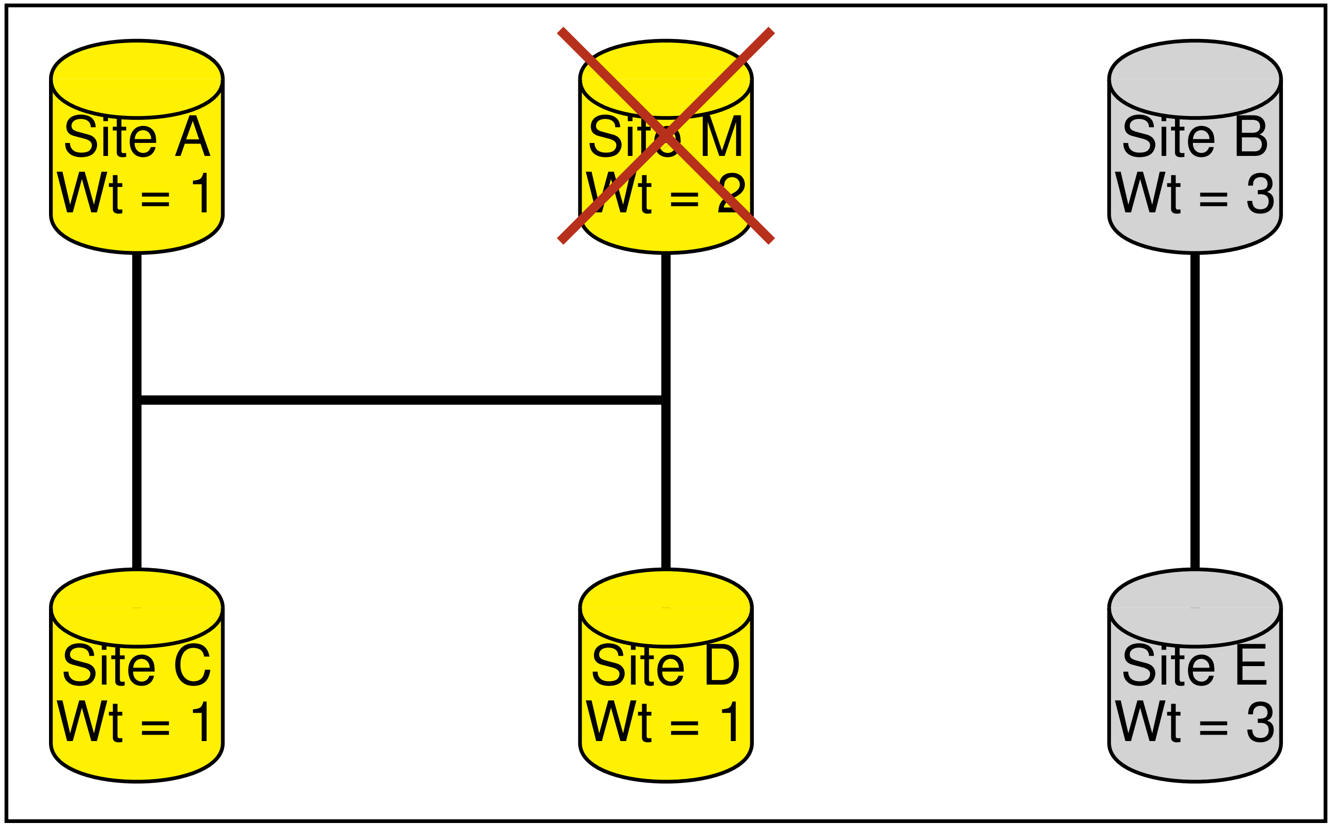


Case 4: Master site is operational in minority partition

- Site M stops functioning as master site
- A new master site is elected in majority partition
- Majority partition remains operational

Quorum Consensus Algorithm

- Each replica has a non-negative weight
- A partition of replicas is allowed to have a master site iff the total weight of the partition exceeds half the total weight of all the replicas

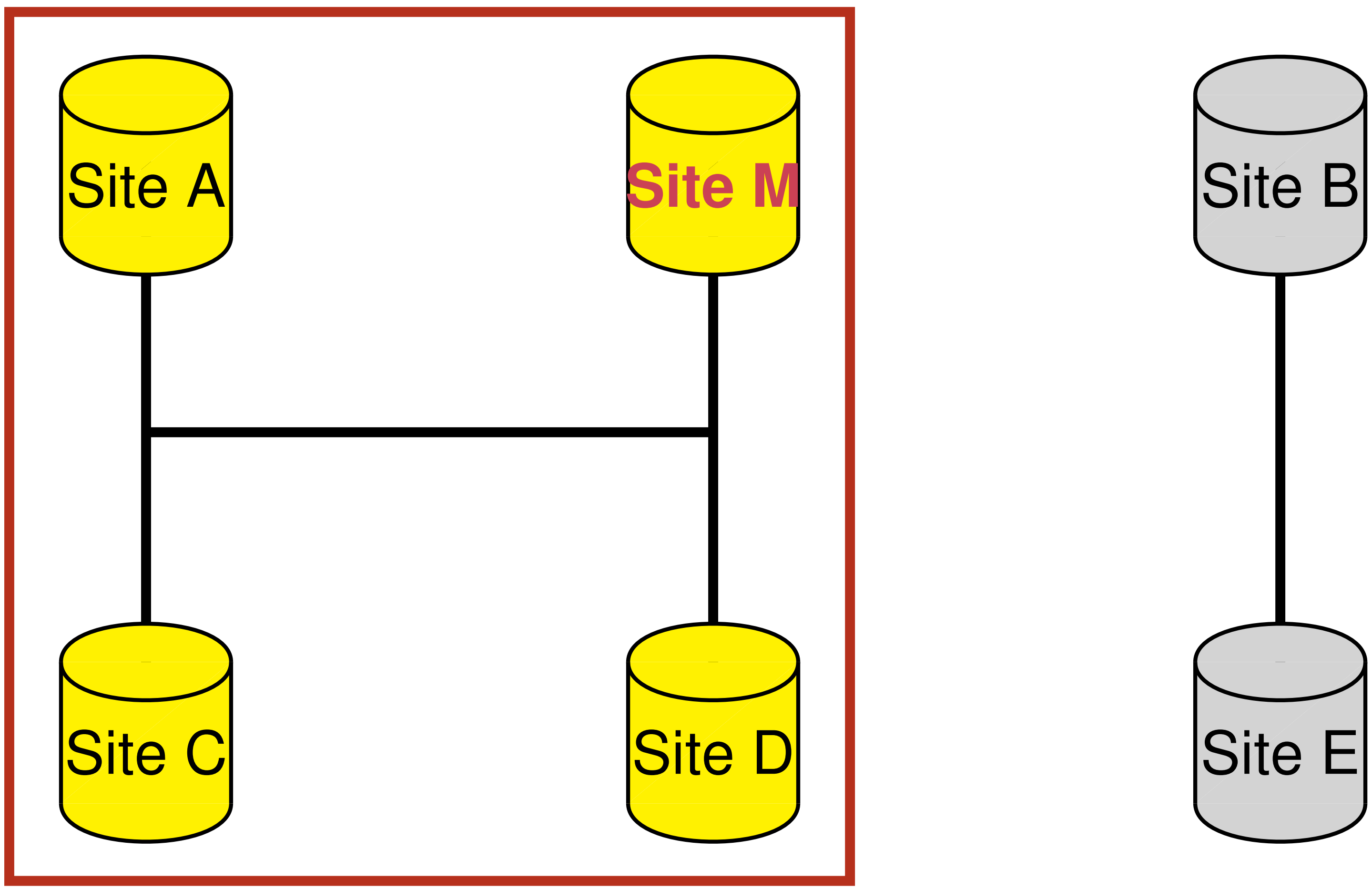


CAP Theorem

- CAP
  - Data **C**onsistency
  - System **A**vailability
  - Tolerance to Network **P**artitions
- **CAP Theorem**: When there's a partitioned network, forfeit either consistency or availability
- Forfeit consistency
  - Resume execution on a selected partition
  - Data could become inconsistent if the selected partition requires a new master site
- Forfeit availability
  - Wait for network to recover before resuming execution

Partitioned Network

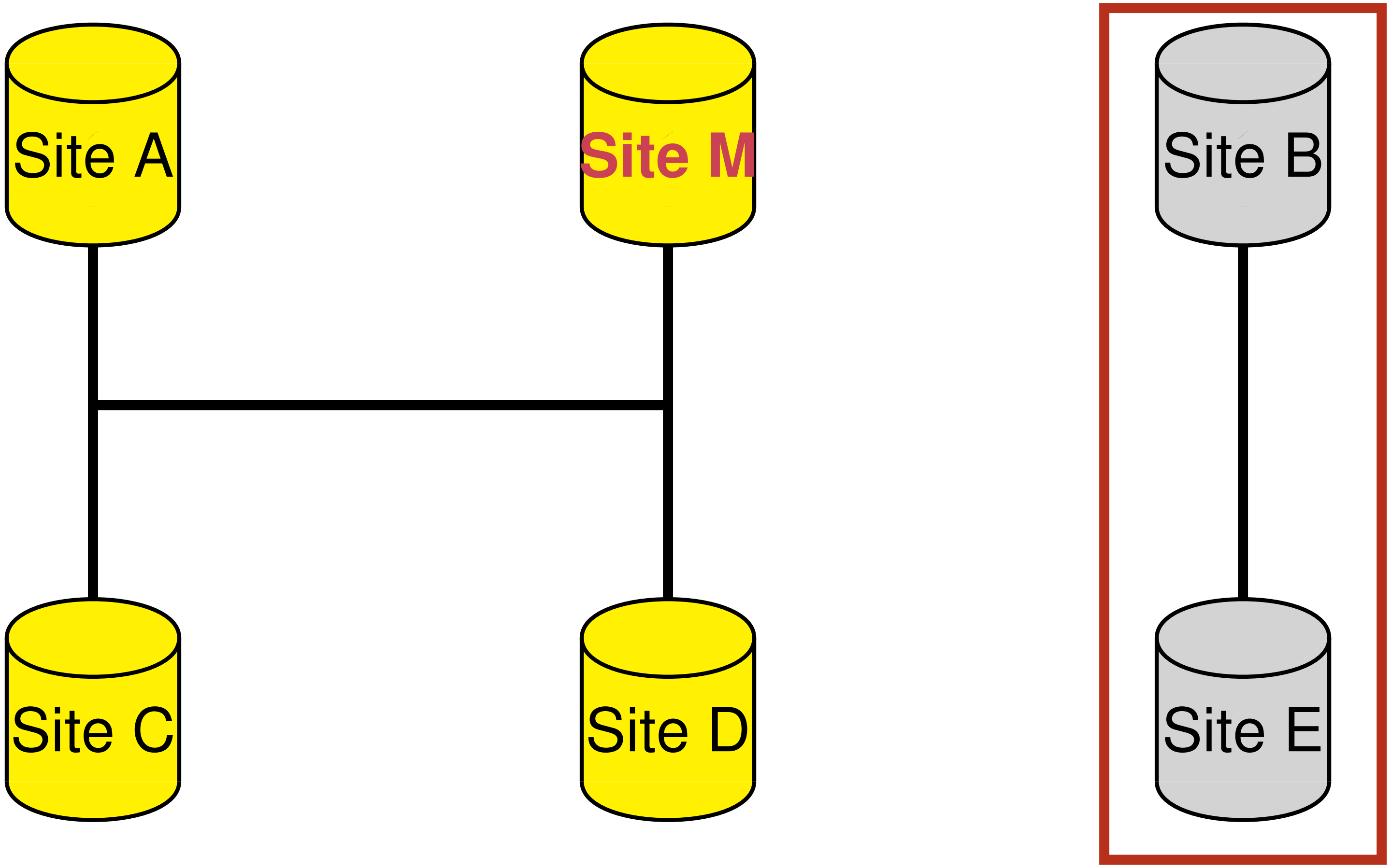
Consistency OK



Selected partition with master site M

Partitioned Network

Could be inconsistent



Selected partition w/o master site

References

- T. Özsu & P. Valdureiz, *Data Replication*, Chapter 6, Principles of Distributed Database Systems, 4^{th} Edition, 2020