Parallel and Distributed Transaction Processing

Transaction Concept

- A **transaction** is a *unit* of program execution that accesses and possibly updates various data items.
- E.g., transaction to transfer $50 from account A to account B:
  1. **read**(*A*)
  2. *A* := *A* – 50
  3. **write**(*A*)
  4. **read**(*B*)
  5. *B* := *B* + 50
  6. **write**(*B)*
- Two main issues to deal with:
  - Failures of various kinds, such as hardware failures and system crashes
  - Concurrent execution of multiple transactions

Required Properties of a Transaction

- Transaction to transfer $50 from account A to account B:
    1. **read**(*A*)
    2. *A* := *A* – 50
    3. **write**(*A*)
    4. **read**(*B*)
    5. *B* := *B* + 50
    6. **write**(*B)*
- **Atomicity requirement**
    - If the transaction fails after step 3 and before step 6, money will be “lost” leading to an inconsistent database state
        - Failure could be due to software or hardware
    - The system should ensure that updates of a partially executed transaction are not reflected in the database

Required Properties of a Transaction

- Transaction to transfer $50 from account A to account B:
   1. **read**(*A*)
   2. *A* := *A* – 50
   3. **write**(*A*)
   4. **read**(*B*)
   5. *B* := *B* + 50
   6. **write**(*B)*

- **Durability requirement** — once the user has been notified that the transaction has completed (i.e., the transfer of the $50 has taken place), the **updates to the database** by the transaction must persist even if there are software or hardware failures.

Required Properties of a Transaction (Cont.)

- Transaction to transfer $50 from account A to account B:
  1. **read**(*A*)
  2. *A* := *A* – 50
  3. **write**(*A*)
  4. **read**(*B*)
  5. *B* := *B* + 50
  6. **write**(*B)*

- **Consistency requirement** in above example:
- The sum of A and B is unchanged by the execution of the transaction
- A transaction, when starting to execute, must see a consistent database.
- During transaction execution the database may be temporarily inconsistent.
- When the transaction completes successfully the database must be consistent

Required Properties of a Transaction (Cont.)

- **Isolation requirement** — if between steps 3 and 4, another transaction **T2** is allowed to access the partially updated database, it will see an inconsistent database (the sum *A* + *B* will be less than it should be).

T1	**T2**
1. **read**(*A*)	
2. *A* := *A* – 50	
3. **write**(*A*)	
	read(A), read(B), print(A+B)
4. **read**(*B*)	
5. *B* := *B* + 50	
6. **write**(*B*	

- Isolation can be ensured trivially by running transactions **serially**
  - That is, one after the other.

Required Properties of a Transaction (Cont.)

- **Isolation requirement** — if between steps 3 and 6, another transaction **T2** is allowed to access the partially updated database, it will see an inconsistent database (the sum *A* + *B* will be less than it should be).

T1	**T2**
1. **read**(*A*)	
2. *A* := *A* – 50	
3. **write**(*A*)	
	read(A), read(B),
4. **read**(*B*)	
5. *B* := *B* + 50	
6. **write**(*B*	

However, executing multiple transactions concurrently has significant benefits.

- Isolation can be ensured trivially by running transactions **serially**
  - That is, one after the other.

ACID Properties

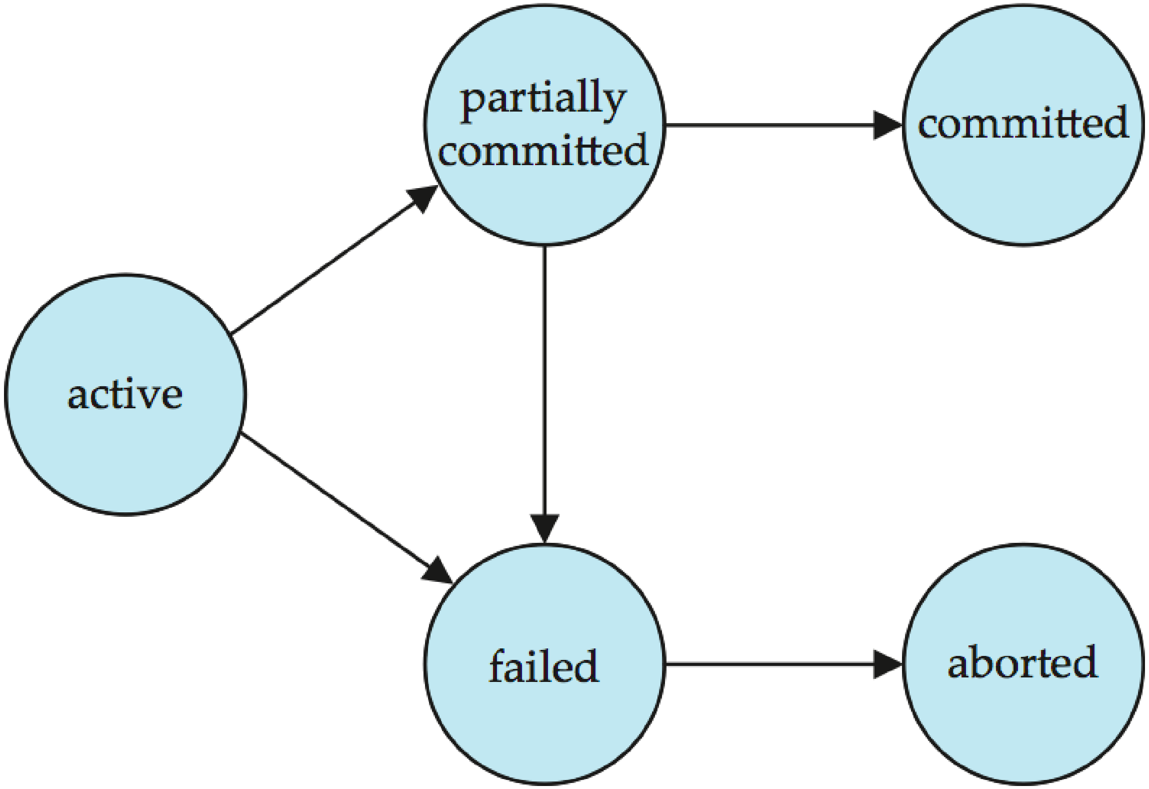
A **transaction** is a unit of program execution that accesses and possibly updates various data items. To preserve the integrity of data the database system must ensure:

- **Atomicity.** Either all operations of the transaction are properly reflected in the database or none are.
- **Consistency.** Execution of a transaction in isolation preserves the consistency of the database.
- **Isolation.** Although multiple transactions may execute concurrently, each transaction must be unaware of other concurrently executing transactions. Intermediate transaction results must be hidden from other concurrently executed transactions.
  - That is, for every pair of transactions T_i and T_j, it appears to T_i that either T_j, finished execution before T_i started, or T_j started execution after T_i finished.
- **Durability.** After a transaction completes successfully, the changes it has made to the database persist, even if there are system failures.

Transaction State

- **Active** – the initial state; the transaction stays in this state while it is executing
- **Partially committed** – after the final statement has been executed.
- **Failed --** after the discovery that normal execution can no longer proceed.
- **Aborted** – after the transaction has been rolled back and the database restored to its state prior to the start of the transaction. Two options after it has been aborted:
  - Restart the transaction
    - can be done only if no internal logical error
  - Kill the transaction
- **Committed** – after successful completion.

Transaction State (Cont.)

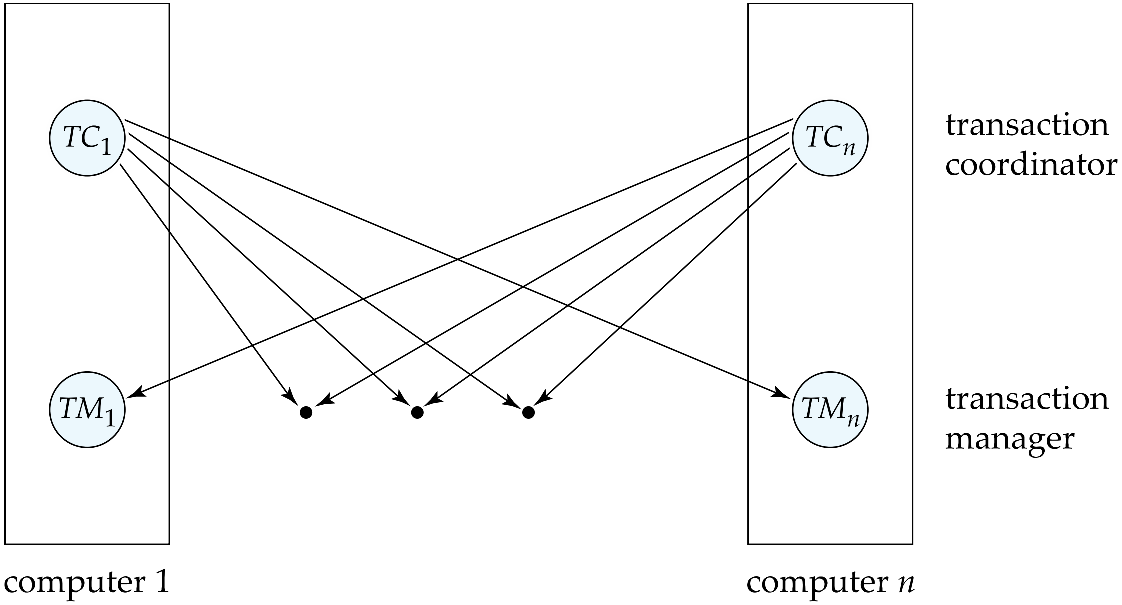


Distributed Transactions

- **Local transactions**
  - Access/update data at only one database
- **Global transactions**
  - Access/update data at more than one database
- Key issue: how to ensure ACID properties for transactions in a system with global transactions spanning multiple database

Distributed Transactions

- Transaction may access data at several sites.
  - Each site has a local **transaction manager**
  - Each site has a **transaction coordinator**
    - Global transactions submitted to any transaction coordinator

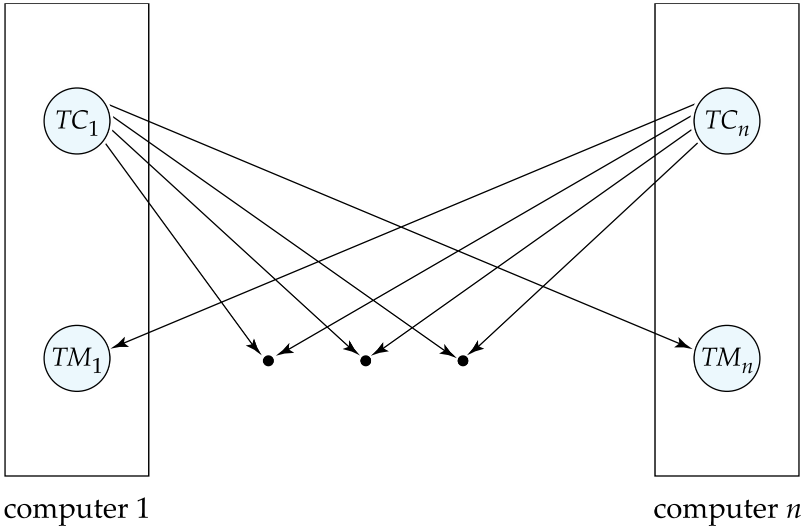


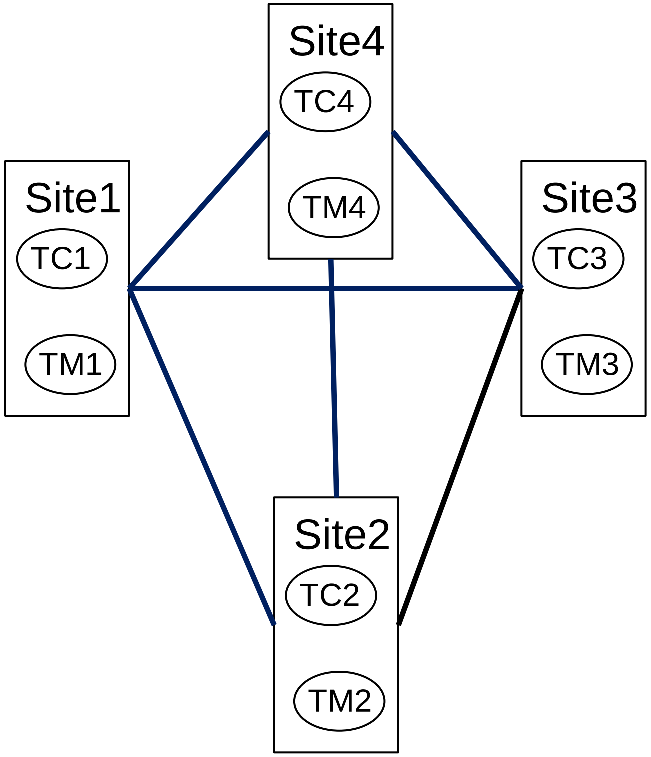
Distributed Transactions

Each transaction coordinator is responsible for:

- Starting the execution of transactions that originate at the site.
- Distributing sub-transactions at appropriate sites for execution.
- Coordinating the termination of each transaction that originates at the site
  - transaction must be committed at all sites or aborted at all sites.

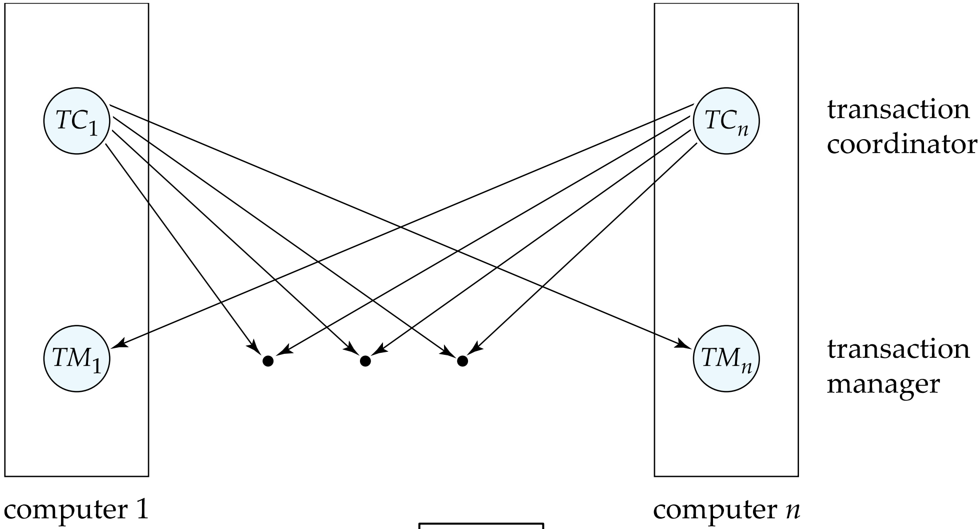
Question-20-1: Site 3 has initiated Transaction T to transfer Tk. 1000 from account P at site 4 to account Q at site 2. Write down the tasks of TC3.



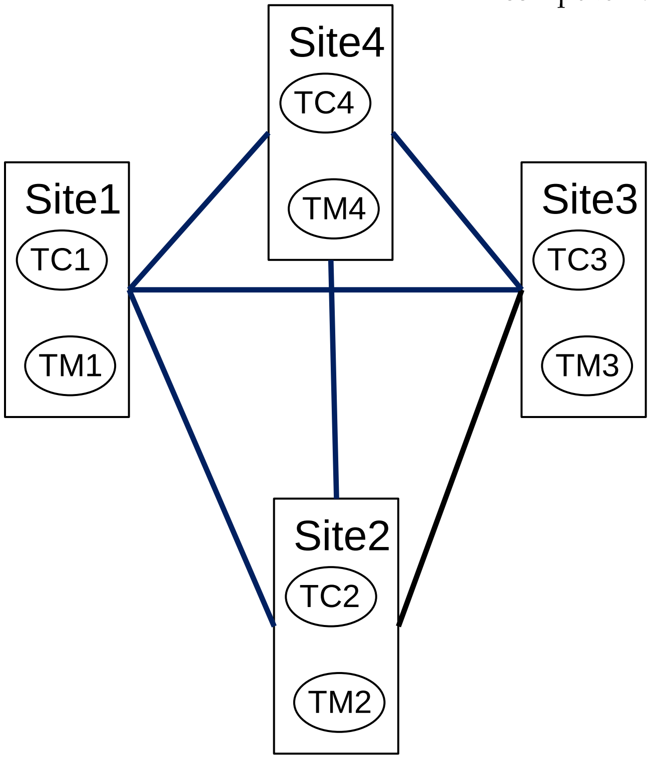


Distributed Transactions

- Each local transaction manager responsible for:
  - Maintaining a log for recovery purposes
  - Coordinating the execution and commit/abort of the transactions executing at that site.

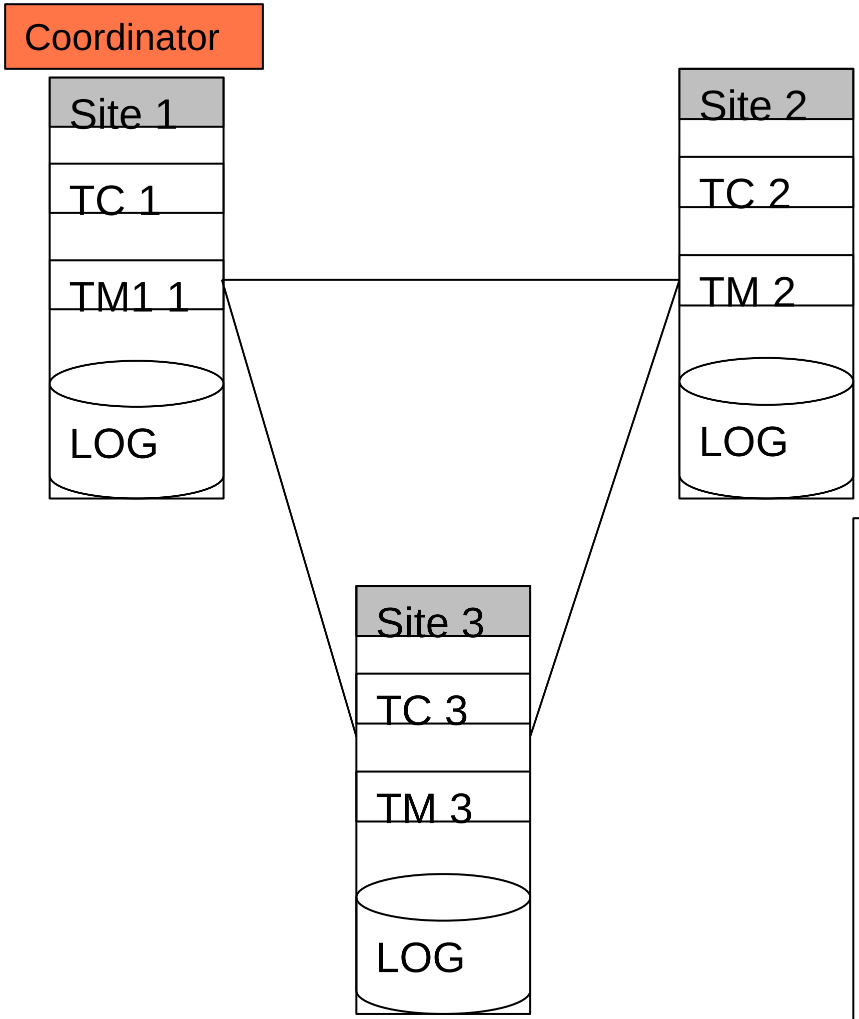


Question 20-2:
Site 1 has initiated Transaction T to transfer Tk. 1000 from account A at site 3 to account B at site 2 and transaction T1 to add Tk. 5000 to account C at site 3. Write down the tasks of TM3.



System Failure Modes

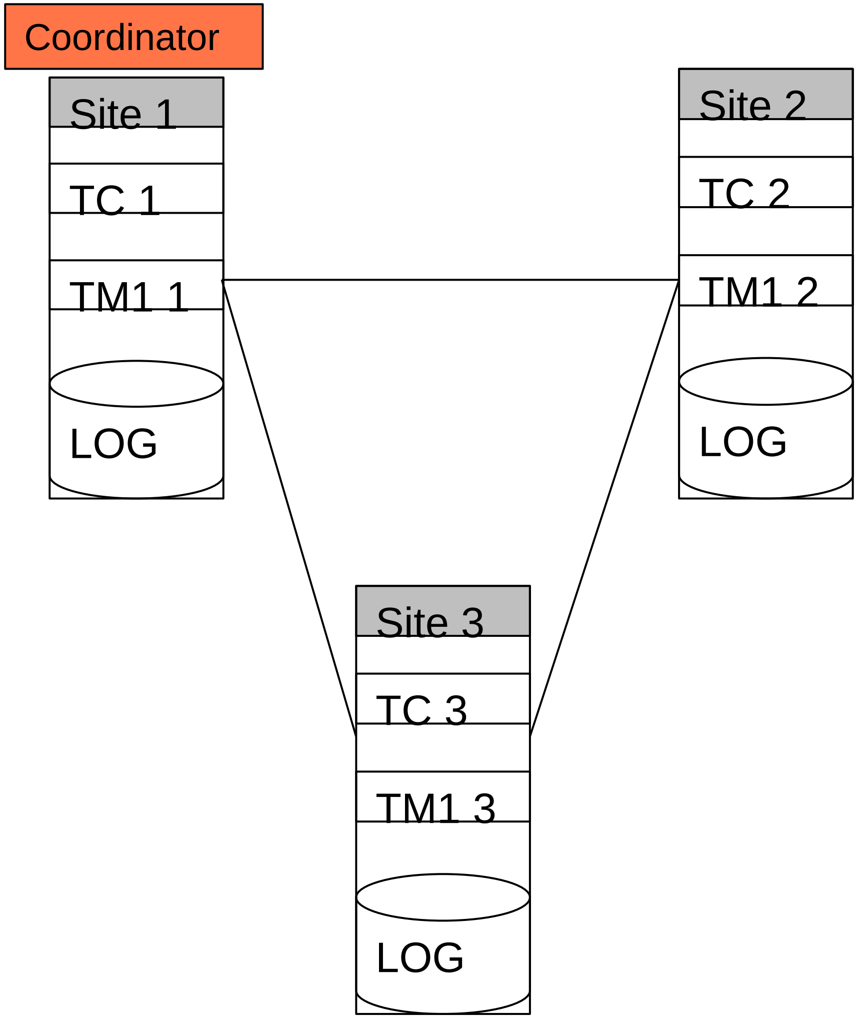
- Failures unique to distributed systems:
  - Failure of a site.
  - Loss of massages
    - Handled by network transmission control protocols such as TCP-IP
  - Failure of a communication link
    - Handled by network protocols, by routing messages via alternative links
  - **Network partition**
    - A network is said to be **partitioned** when it has been split into two or more subsystems that lack any connection between them
      - Note: a subsystem may consist of a single node
- Network partitioning and site failures are generally indistinguishable.



Transaction: Transfer Tk. 5000 from account A to account B. A in site 2 and B in site 3. Transaction has been initiated from site 1.

The transaction with lock

<Start T> by TC1
Lock (A) request by TC1 granted by TM2
READ (A)
A = A- 5000
Write (A)
Lock (B) request by TC1 granted by TM3
READ (B)
B=B+5000
WRITE (B)
COMMIT (T)
UNLOCK (A)
UNLOCK (B)



<Start T> by TC1
Lock (A) request by TC1 granted by TM1
READ (A)
A = A- 5000
Write (A)
Lock (B) request by TC1 granted by TM2
READ (B)
B=B+5000
WRITE (B)
COMMIT (T)
UNLOCK (A)
UNLOCK (B)

Commit protocols are used to ensure atomicity across sites

A transaction which executes at multiple sites must either be committed at all the sites, or aborted at all the sites.

Cannot have transaction committed at one site and aborted at another

Question 21-1: Why is COMMIT protocol needed to COMMIT the transaction T initiated by TC1? Explain.

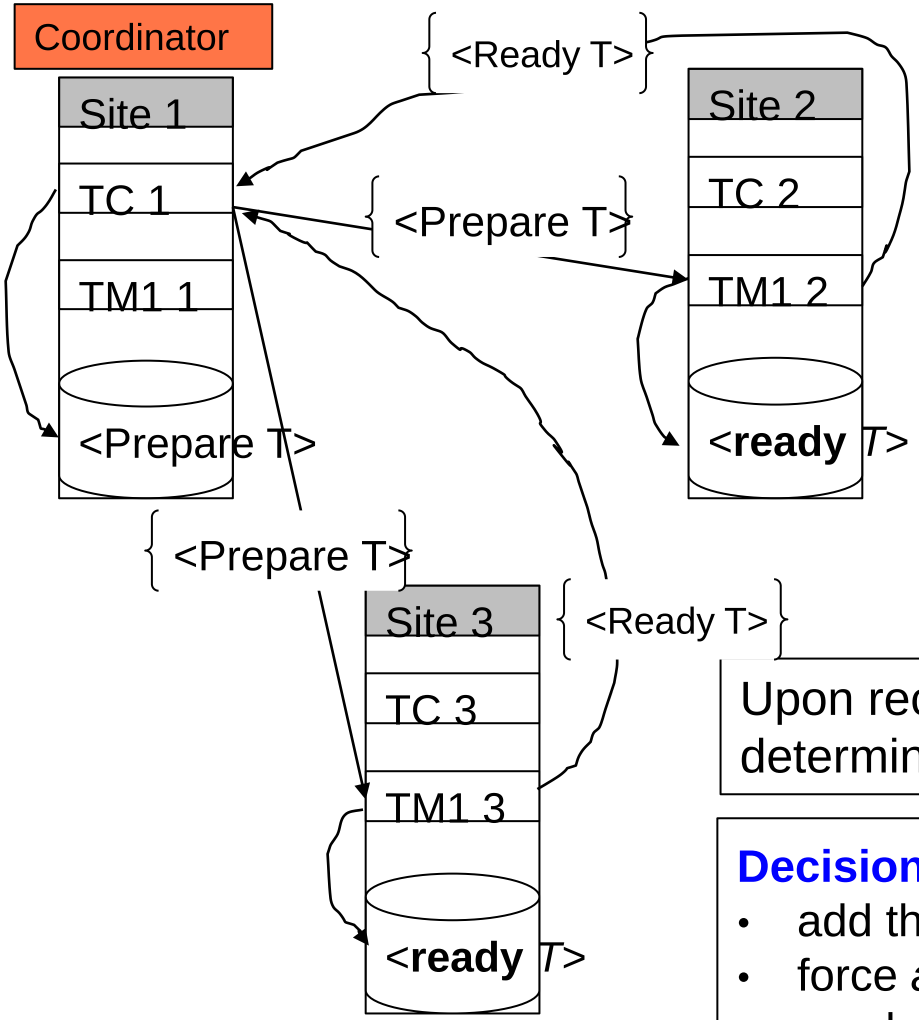
Commit Protocols

- The *two-phase commit* (2PC) protocol is widely used
- *Three-phase commit* (3PC) protocol avoids some drawbacks of 2PC, but is more complex
- *Consensus protocols* solve a more general problem, but can be used for atomic commit
- The protocols we study all assume **fail-stop** model – failed sites simply stop working, and do not cause any other harm, such as sending incorrect messages to other sites.

Two Phase Commit Protocol (2PC)

- Execution of the protocol is initiated by the coordinator after the last step of the transaction has been reached.
- The protocol involves all the local sites at which the transaction executed
- Protocol has two phases
- Let *T* be a transaction initiated at site *S*1 and let the transaction coordinator at *S*1 be T*C*1

Phase 1: Obtaining a Decision



Coordinator asks all participants to *prepare* to commit transaction *T*

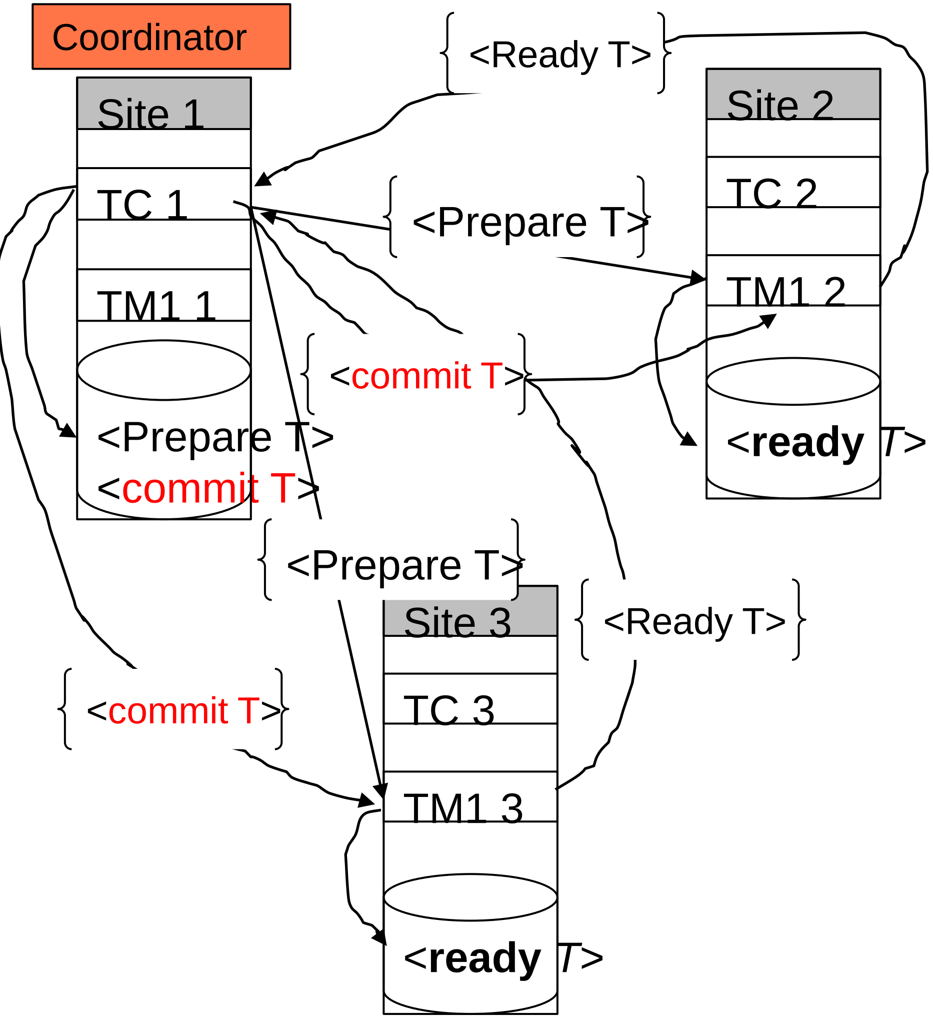
TC_1 adds the records <**prepare** *T*> to the log and forces log to stable storage sends **prepare** *T* messages to all sites at which *T* executed

Upon receiving message, transaction manager at site determines if it can commit the transaction or not

Decision1: All sites wants to COMMIT

- add the record <**ready** *T*> to the log
- force *all records* for *T* to stable storage
- send **ready** *T* message to TC1

Phase 2: Recording the Decision



Coordinator

Site 1

TC 1

TM1 1

<Prepare T>
<commit T>

Site 2

TC 2

TM1 2

<**ready** *T*>

Site 3

TC 3

TM1 3

<**ready** *T*>

<Ready T>

<Prepare T>

<commit T>

<Prepare T>

<Ready T>

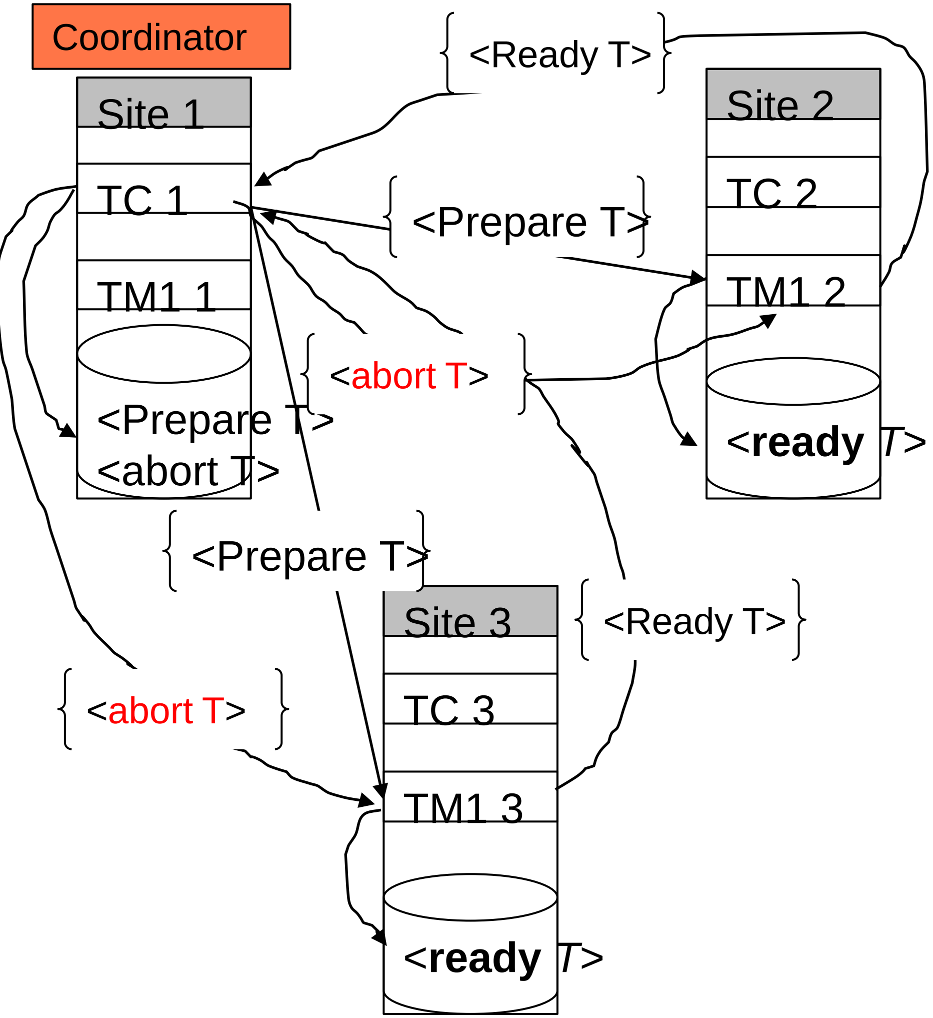
<commit T>

Transaction COMMIT by TC1

- T can be committed of TC1 received a <**ready** T> message from all the participating sites.
- Coordinator adds a decision record, <**commit** *T*> to the log and forces record onto stable storage.
- Once the record stable storage it is irrevocable (even if failures occur)
- Coordinator sends a message to **commit** to each participant informing it of the decision.
- Participants take appropriate action locally.

Question 21-2: Write down the activities of the coordinator of site 1 in phase 1 and phase 2 for the case when all the sites including coordinator wants to commit

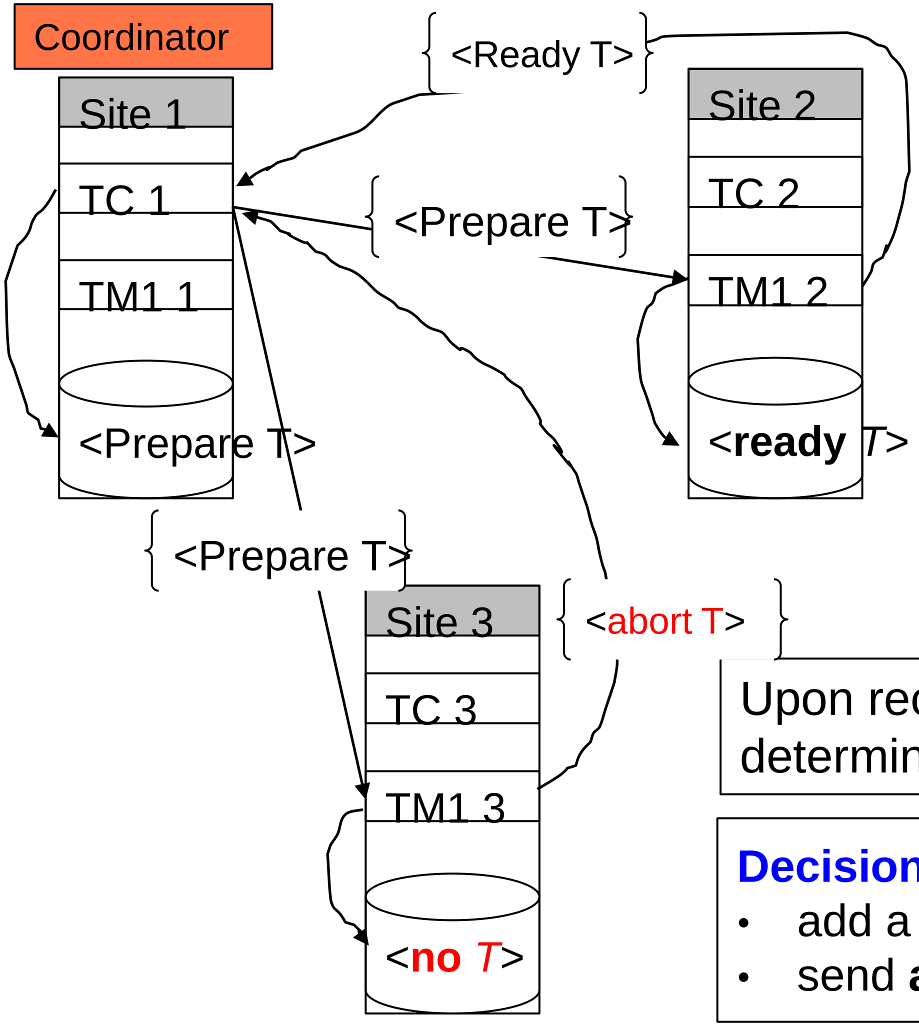
Phase 2: Recording the Decision



Transaction **ABORT** by TC1

- Even after receiving **<ready** T> messages by TC1 from all the participating sites, TC1 has the authority to abort the transaction T.
- Coordinator adds a decision record, **<abort** *T*> to the log and forces record onto stable storage.
- Once the record stable storage it is irrevocable.
- Coordinator sends a message to **abort** to each participant informing it of the decision.
- Participants take appropriate action locally.

Phase 1: Obtaining a Decision



Coordinator asks all participants to *prepare* to commit transaction *T*

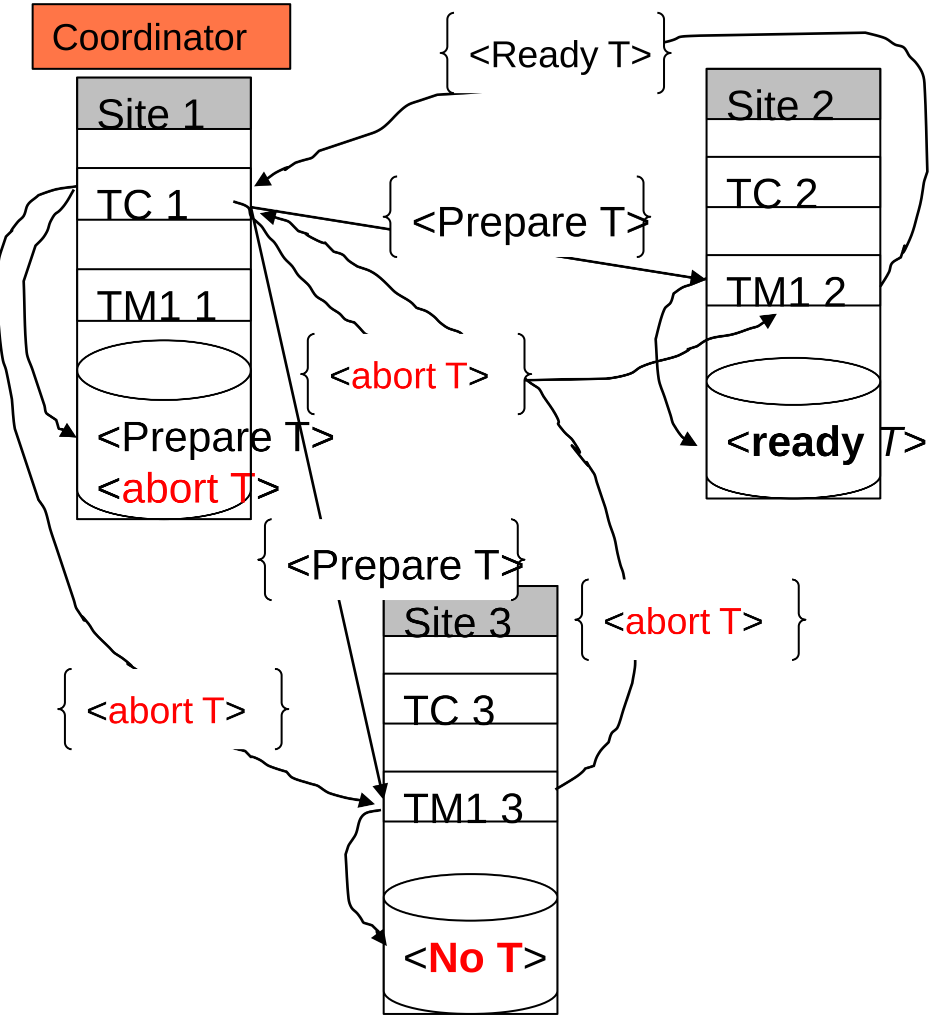
TC$_1$ adds the records <**prepare** *T*> to the log and forces log to stable storage sends **prepare** *T* messages to all sites at which *T* executed

Upon receiving message, transaction manager at site determines if it can commit the transaction or not

Decision2: One site don't want to COMMIT (Site3)

- add a record <**no** *T*> to the log and
- send **abort** *T* message to TC1

Phase 2: Recording the Decision

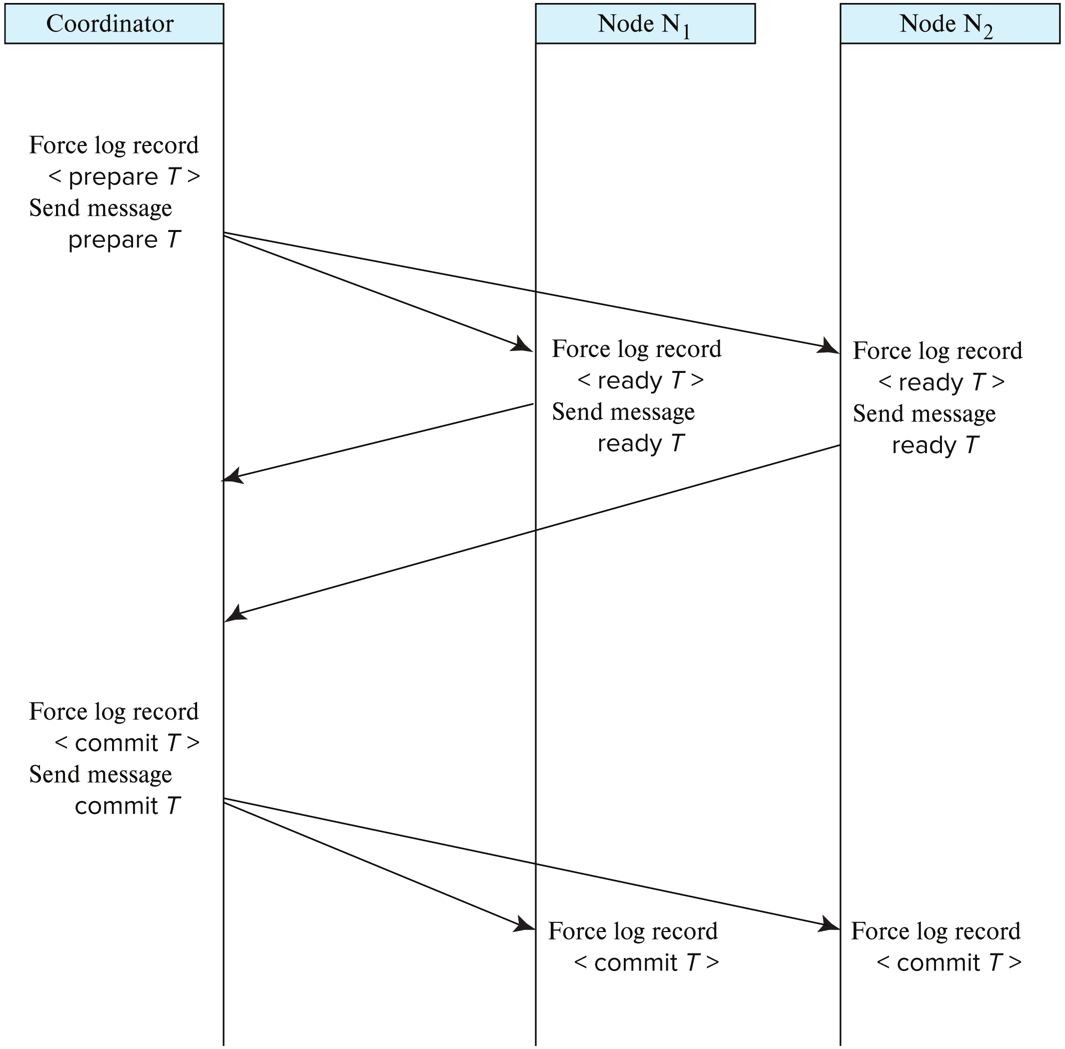


Transaction must be **aborted** by TC1

- T must be aborted by TC1 if it received a <**abort** T> message from **any of** the participating sites.
- Coordinator adds a decision record, <**abort***T*> to the log and forces record onto stable storage.
- Once the record stable storage it is irrevocable (even if failures occur)
- Coordinator sends a message to **abort** to each participant informing it of the decision.
- Participants take appropriate action locally.

Question 21-3: Write down the activities of the site 3 in phase 1 and coordinator in phase 2 in the case when site 3 do not want to commit.

Two-Phase Commit Protocol



Handling of Failures - Site Failure

Site 2 was failed
When site *S2* recovers, it examines its log to determine the fate of
transactions active at the time of the failure.

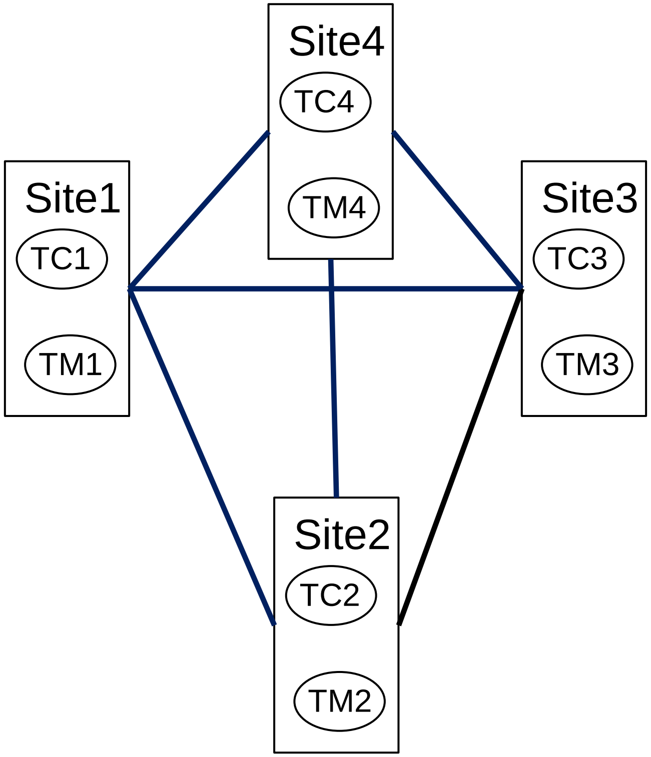
Case 1: Log contain <**commit** *T*> record:
site 2 executes **redo** (*T*)
Question 4-1: Why redo (T)?

Case 2: Log contains <**abort** *T*> record:
site 2 executes **undo** (*T*)
Question 4-2: Why undo (T)?

Case 3: Log contains <**ready** *T*> record:
site must consult TC1 to determine the fate of *T*.

If *T* committed, **redo** (*T*)

If *T* aborted, **undo** (*T*)



Handling of Failures - Site Failure

Site 2 was failed

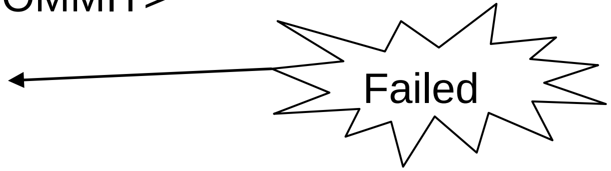
When site *S2* recovers, it examines its log to determine the fate of
transactions active at the time of the failure.

Case 1: Log contain <**commit** *T*> record: site 2 executes **redo** (*T*)

Question 4-1: Why redo (T)?

Site 2 log records

<Start T>
<T, A, 20000, 15000>
<Ready T>
<T COMMIT>



<Start T>   by TC1
Lock (A) request by TC1 granted by TM2
READ (A)
A = A- 5000
Write (A)
Lock (B) request by TC1 granted by TM3
READ (B)
B=B+5000
WRITE (B)
COMMIT (T)
UNLOCK (A)
UNLOCK (B)

Site 2 log	Site 2 recovery
<Start T>	
<T, A, 20000, 15000>	
<Ready T>	
<T COMMIT>	
	Set A = 15000

Handling of Failures - Site Failure

Site 2 was failed

When site *S2* recovers, it examines its log to determine the fate of
transactions active at the time of the failure.

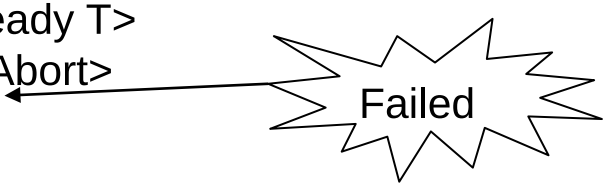
Case 2: Log contains <**abort** *T*> record:
site 2 executes **undo** (*T*)

Question 4-2: Why undo (T)?

```
<Start T>      by TC1
Lock (A)  request by TC1 granted by TM2
READ (A)
A = A- 5000
Write (A)
Lock (B)  request by TC1 granted by TM3
READ (B)
B=B+5000
WRITE (B)
COMMIT (T)
UNLOCK (A)
UNLOCK (B)
```

Site 2 log records

<Start T>
<T, A, 20000, 15000>
<Ready T>
<T Abort>



Site 2 log	Site 2 recovery
<Start T>	
<T, A, 20000, 15000>	
<Ready T>	
<T Abort>	
	Set A = 20000

Handling of Failures - Site Failure

Site 2 was failed

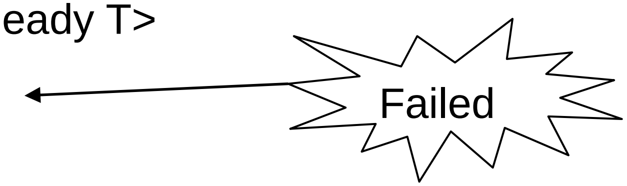
When site *S2* recovers, it examines its log to determine the fate of
transactions active at the time of the failure.

Case 3: Log contains <**ready** *T*> record: site must consult TC1 to determine the fate of *T*.

If *T* committed, **redo** (*T*)

Site 2 log records

<Start T>
<T, A, 20000, 15000>
<Ready T>


Failed

<Start T>   by TC1
Lock (A) request by TC1 granted by TM2
READ (A)
A = A- 5000
Write (A)
Lock (B) request by TC1 granted by TM3
READ (B)
B=B+5000
WRITE (B)
COMMIT (T)
UNLOCK (A)
UNLOCK (B)

Site 2 log

<Start T>
<T, A, 20000, 15000>
<Ready T>
<T COMMIT>

Site 2 recovery

Set A = 20000

Handling of Failures - Site Failure

Site 2 was failed

When site *S2* recovers, it examines its log to determine the fate of transactions active at the time of the failure.

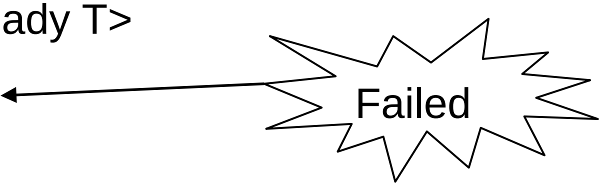
Case 3: Log contains <**ready** *T*> record: site must consult TC1 to determine the fate of *T*.

If *T* aborted, **undo** (*T*)

<Start T> by TC1
Lock (A) request by TC1 granted by TM2
READ (A)
A = A- 5000
Write (A)
Lock (B) request by TC1 granted by TM3
READ (B)
B=B+5000
WRITE (B)
COMMIT (T)
UNLOCK (A)
UNLOCK (B)

Site 2 log records

<Start T>
<T, A, 20000, 15000>
<Ready T>



Site 2 log	Site 2 recovery
<Start T>	
<T, A, 20000, 15000>	
<Ready T>	
<T Abort>	Set A = 20000

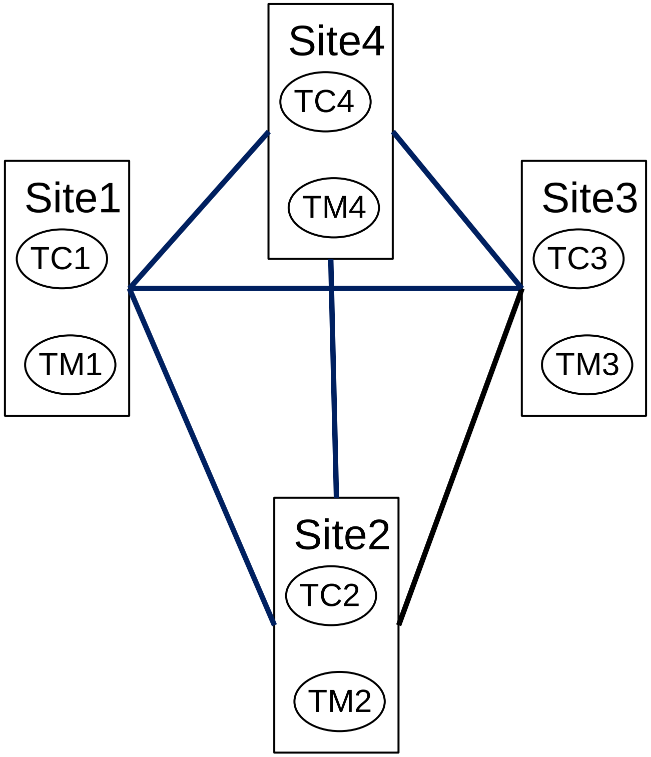
Handling of Failures - Site Failure

Site 2 was failed

Case 4: The log contains no control records concerning *T* implies that S2 failed before responding to the **prepare** *T* message from TC1.

Since the failure of *S2* precludes the sending of such a response *TC1* must abort *T*

S2 must execute **undo** (*T*)



Discuss when the transaction will be undo(T) and when it will be redo(T)

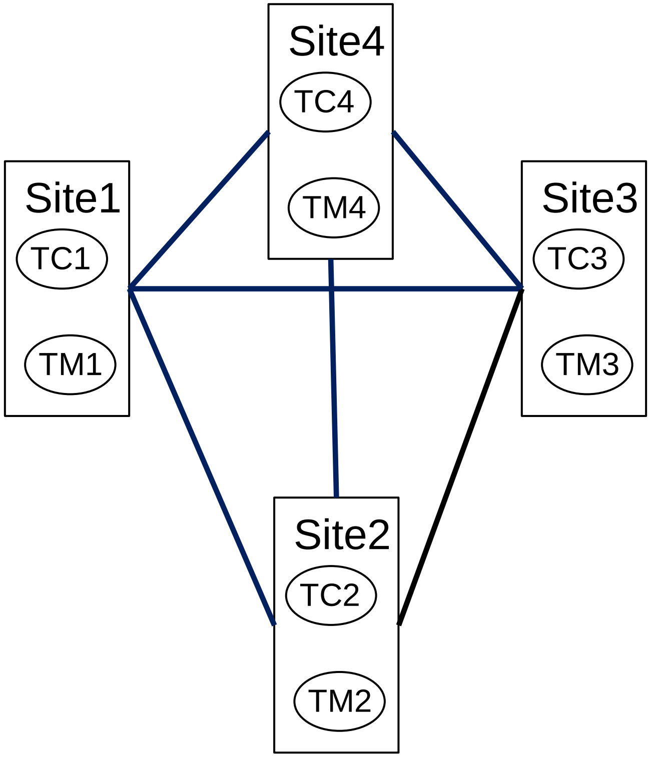
Handling of Failures - Coordinator Failure

If coordinator fails while the commit protocol for *T* is executing then participating sites must decide on *T*' s fate

Case 1: If an active site contains a <**commit** *T*> record in its log, then *T* must be committed.

Case 2: If an active site contains an <**abort** *T*> record in its log, then *T* must be aborted.

Case 3: If some active participating site does not contain a <**ready** *T*> record in its log, then the failed coordinator C_i cannot have decided to commit *T*. Can therefore abort *T*.

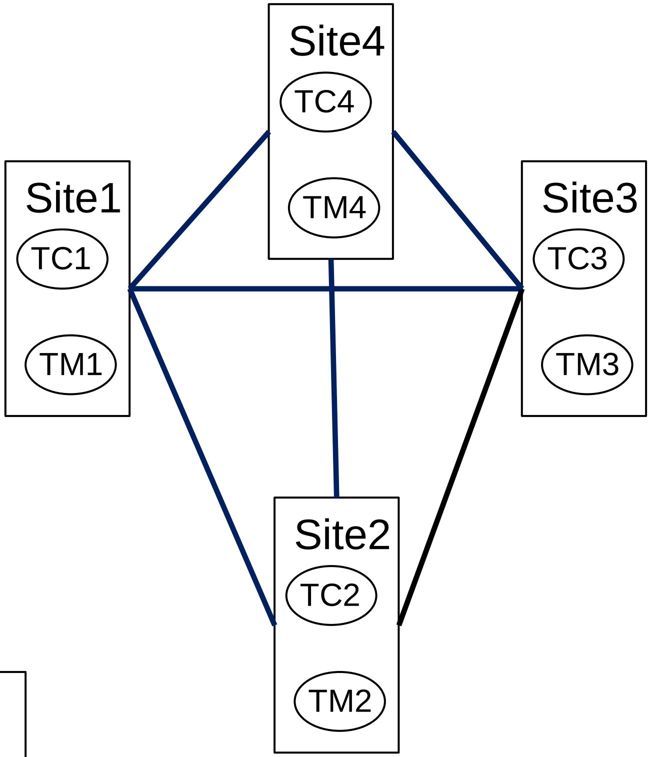


Handling of Failures - Coordinator Failure

If coordinator fails while the commit protocol for *T* is executing then participating sites must decide on *T*' s fate

Case 4: If none of the above cases holds, then all active sites must have a <**ready** *T*> record in their logs, but no additional control records (such as <**abort** *T*> or <**commit** *T*>). In this case active sites must wait for C_i to recover, to find decision.

Blocking problem: active sites may have to wait for failed coordinator to recover.



Question-22-1: Site 3 has initiated Transaction T3 to transfer Tk. 1000 from account P at site 4 to account Q at site 2.
Explain the atomicity of T3 for different types of site and coordinator failures.

Recovery and Concurrency Control

In-doubt transactions: When a failed site is recovered, the transactions those have a <**ready** *T*>, but neither a <**commit** *T*>, nor an <**abort** *T*> log record.

The recovering site must determine the **commit / abort** status of **In-doubt transactions** by contacting other sites;

1. this can slow the recovery (**How?**)
    - Because after recovery, all active transactions must have redo or undo based on Commit / abort status
2. Or can block the recovery (**How?**)
    - In case of coordinator failure, site cannot decide commit/abort and indefinite wait for coordinator

Delay in Recovery (Example)

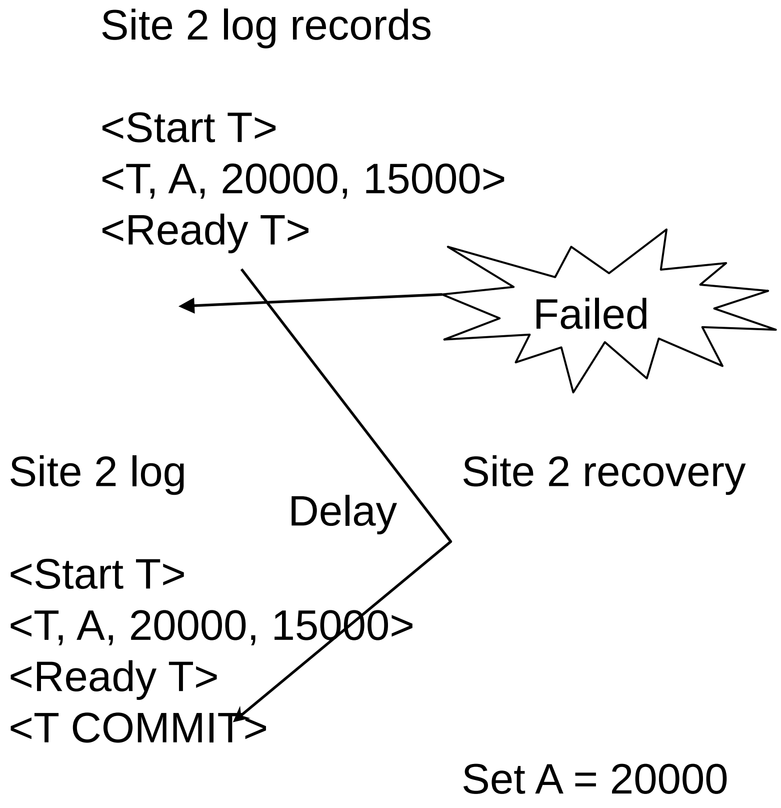
Site 2 was failed

When site *S2* recovers, it examines its log to determine the fate of transactions active at the time of the failure.

Case 3: Log contains <**ready** *T*> record: site must consult TC1 to determine the fate of *T*.

If *T* committed, **redo** (*T*)

<Start T> by TC1
Lock (A) request by TC1 granted by TM2
READ (A)
A = A- 5000
Write (A)
Lock (B) request by TC1 granted by TM3
READ (B)
B=B+5000
WRITE (B)
COMMIT (T)
UNLOCK (A)
UNLOCK (B)



Site 2 log records

<Start T>
<T, A, 20000, 15000>
<Ready T>

Site 2 log

<Start T>
<T, A, 20000, 15000>
<Ready T>
<T COMMIT>

Site 2 recovery

Set A = 20000

Delay in Recovery (Example)

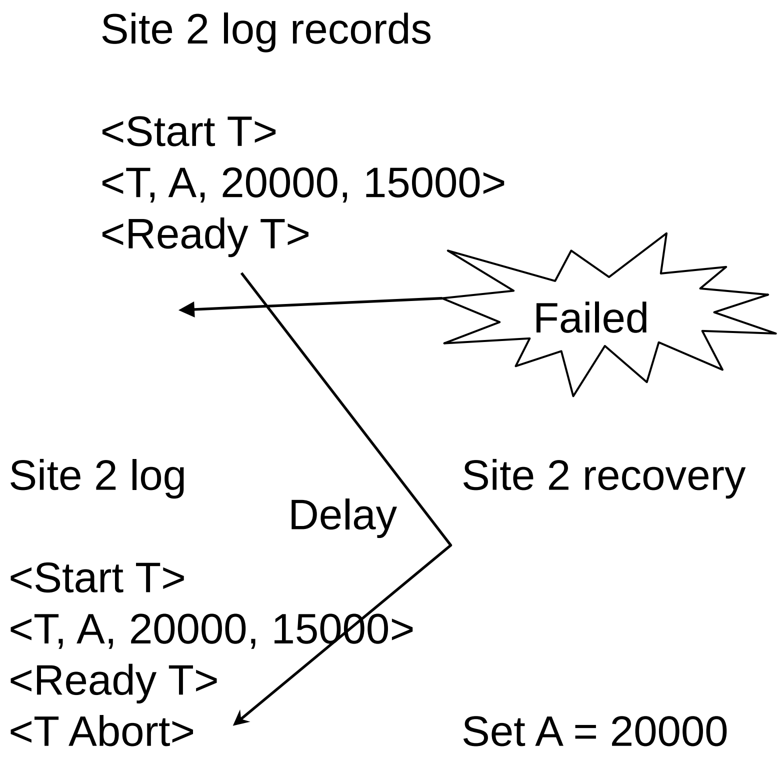
Site 2 was failed

When site *S2* recovers, it examines its log to determine the fate of transactions active at the time of the failure.

Case 3: Log contains <**ready** *T*> record: site must consult TC1 to determine the fate of *T*.

If *T* aborted, **undo** (*T*)

<Start T>   by TC1
Lock (A) request by TC1 granted by TM2
READ (A)
A = A- 5000
Write (A)
Lock (B) request by TC1 granted by TM3
READ (B)
B=B+5000
WRITE (B)
COMMIT (T)
UNLOCK (A)
UNLOCK (B)



Site 2 log records

<Start T>
<T, A, 20000, 15000>
<Ready T>

Site 2 log

<Start T>
<T, A, 20000, 15000>
<Ready T>
<T Abort>

Site 2 recovery

Set A = 20000

Recovery and Concurrency Control

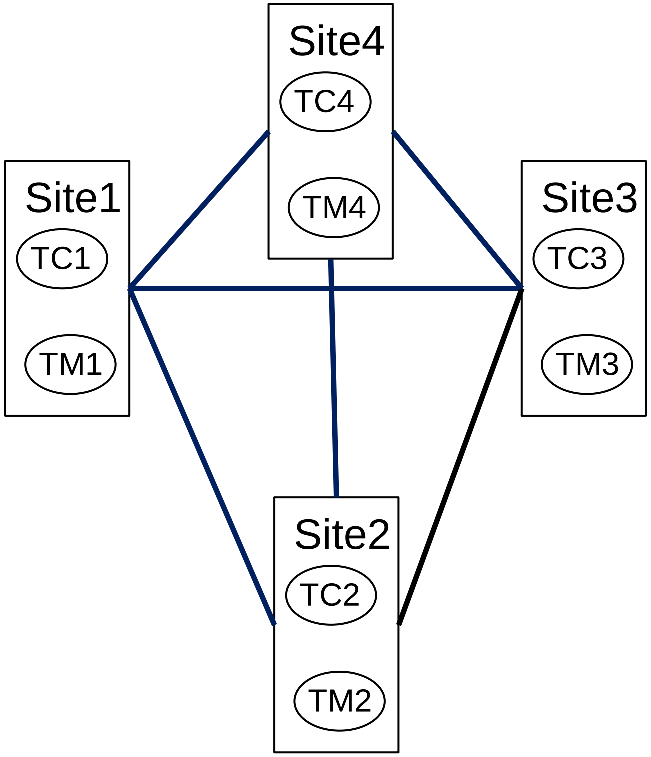
- **For faster recovery,** Recovery algorithms can note lock information in the log.
  - Instead of <**ready** *T*>, write out <**ready** *T*, *L*> *L* = list of locks held by *T* when the log is written (read locks can be omitted).
    - **Question:** A Transaction T5 has Read lock (A), Write Lock (D) and Write Lock (E) at site 3. T5 has Write Lock (G) at site 2. T5 is initiated by site 1. The data item A, D and E are in site 3. Site 3 wants to commit T5. What log record will be written by site 3? How will it be used for recovery?
    - <**ready** *T5*, *{D,E}*>
  - For every in-doubt transaction *T*, all the locks noted in the <**ready** *T*, *L*> log record are reacquired.
- After lock reacquisition, transaction processing can resume; the commit or rollback of in-doubt transactions is performed concurrently with the execution of new transactions.

Avoiding Blocking Using Consensus

- Blocking problem of 2PC is a serious concern
- **How it arises?**
- T1 is initiated by site 1 (coordinator)
- T1 has data from site 2, 3 and 4. Site 2, 3 and 4 are ready to commit and has written <ready T1> to their log.

Just after that, site 1 failed.

- Site 2, 3 and 4 are **blocked** for T1 and wait until site 1 is restored.
- Idea: involve multiple nodes in decision process, so failure of a few nodes does not cause blocking as long as majority don't fail

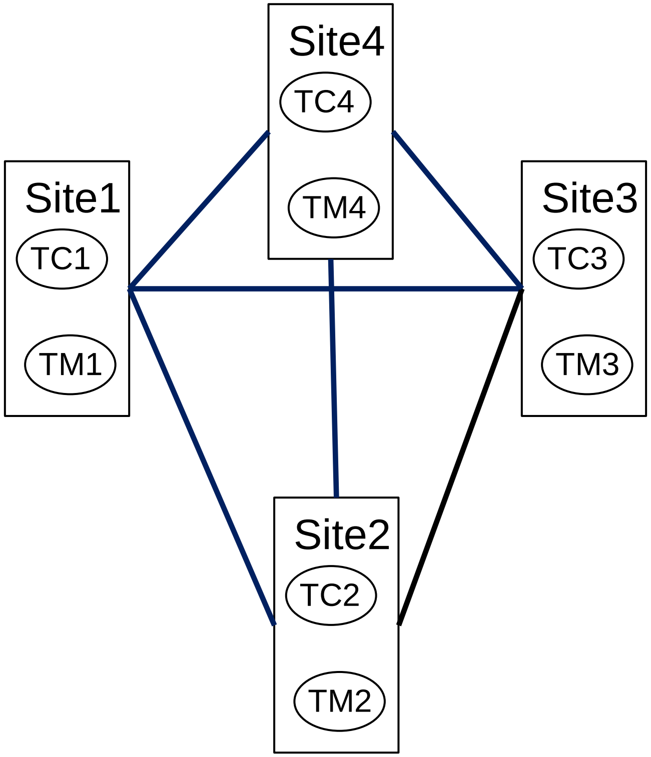


Avoiding Blocking Using Consensus

Solution of Blocking Problem

- **Distributed consensus protocol**

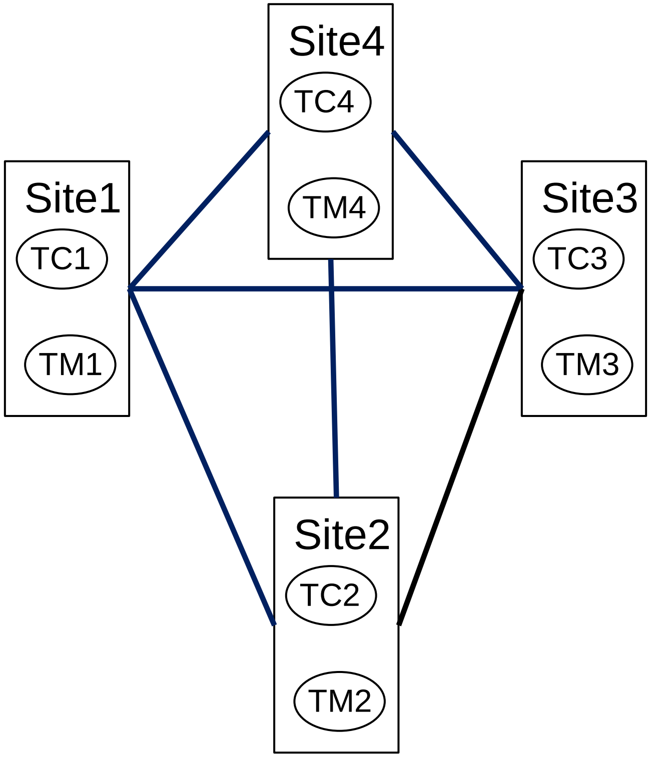
- A set of *n* nodes need to agree on a decision
- Inputs to make the decision are provided to all the nodes, and then each node votes on the decision
- The decision should be made in such a way that all nodes will “learn” the same value for the transaction even if some nodes fail during the execution of the protocol.



Using Consensus to Avoid Blocking

Solution of Blocking Problem

- **Distributed consensus protocol**
- After getting response from 2PC participants, coordinator can initiate distributed consensus protocol by sending its decision to a set of participants who then use consensus protocol to commit the decision
- If coordinator fails before informing all consensus participants
  - Choose a new coordinator, which follows 2PC protocol for failed coordinator
  - If a commit/abort decision was made as long as a majority of consensus participants are accessible, decision can be found without blocking



Using Consensus to Avoid Blocking

Solution of Blocking Problem

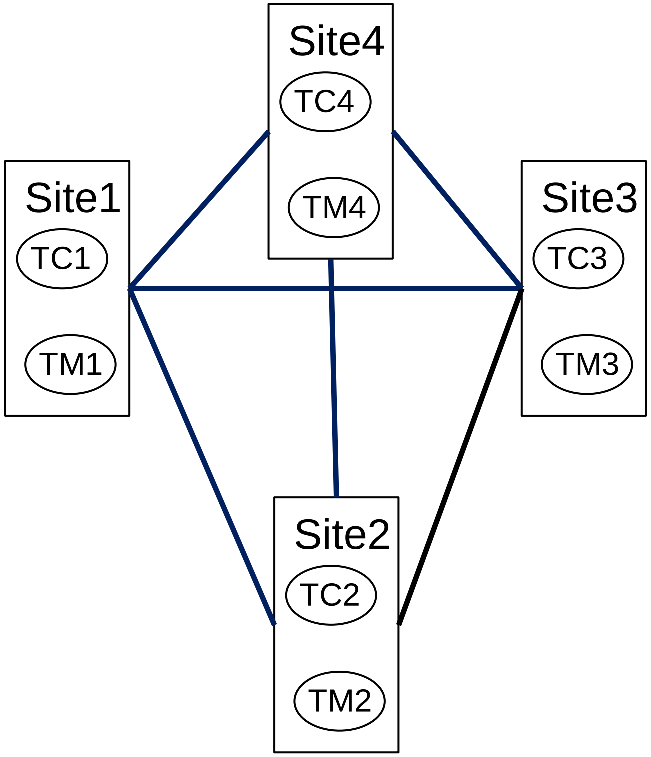
- **Distributed consensus protocol**

- Case 1: Coordinator (Site 1) failed before sending commit proposal to site 2, 3 and 4.

Steps:

1. A new coordinator will be selected (e.g., site 3).
2. Site 3 will initiate 2PC on behalf of site 1
3. Commit/abort will be performed as per consensus protocol.

No blocking of site 2, 3 and 4 for T1.

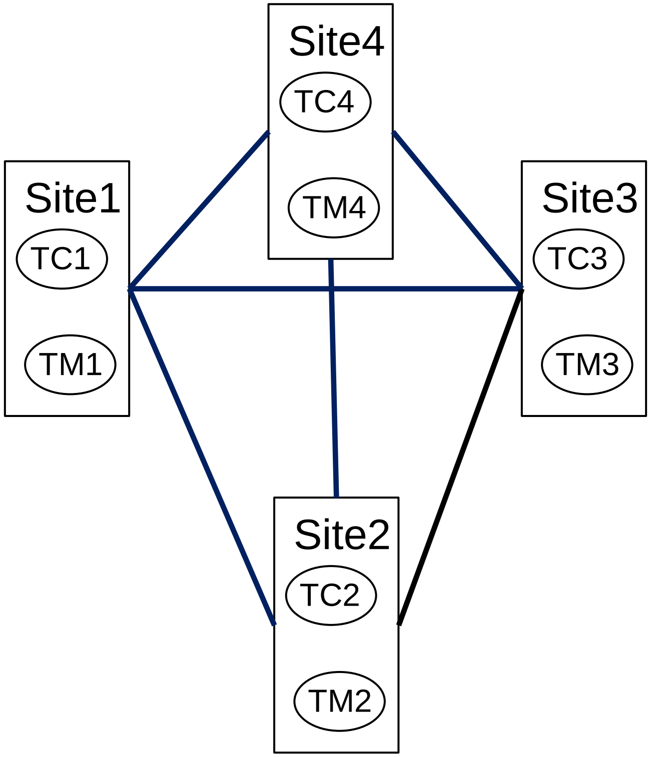


Using Consensus to Avoid Blocking

Solution of Blocking Problem

- **Distributed consensus protocol**
- Case 2: Coordinator (Site 1) failed after sending commit proposal to site 2, 3 and 4.
- Commit/abort will be performed as per consensus protocol.

No blocking of site 2, 3 and 4 for T1.



Distributed Transactions via Persistent Messaging

- Notion of a single transaction spanning multiple sites is inappropriate for many applications
  - E.g., transaction crossing an organizational boundary
  - Latency of waiting for commit from remote site

- **Question:** There is an account A under organization 1 and an account B under organization 2. Transfer Tk. 5000 from account A to account B using persistent messaging protocol.

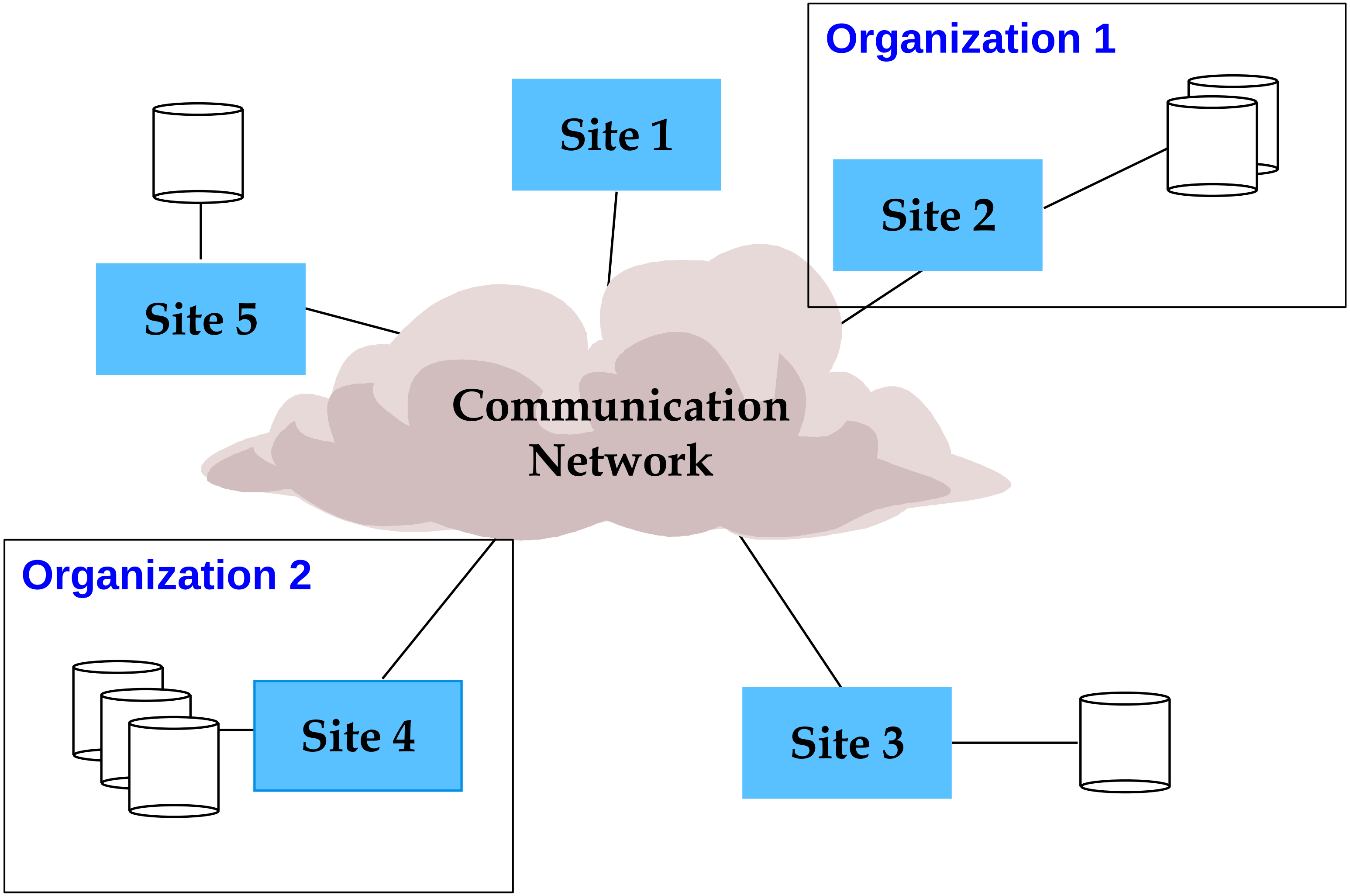
Persistent messaging systems are systems that provide transactional properties to messages

Persistent messages are guaranteed to be delivered exactly once

Debit Tk. 5000 from account A and send a message to organization 2

Organization 2 receives message and credits Tk. 5000 to account B

Distributed DBMS Environment



Persistent Messaging

Atomicity issue

Once transaction sending a message is committed, message must guaranteed to be delivered

Guarantee as long as destination site is up and reachable, code to handle undeliverable messages must also be available

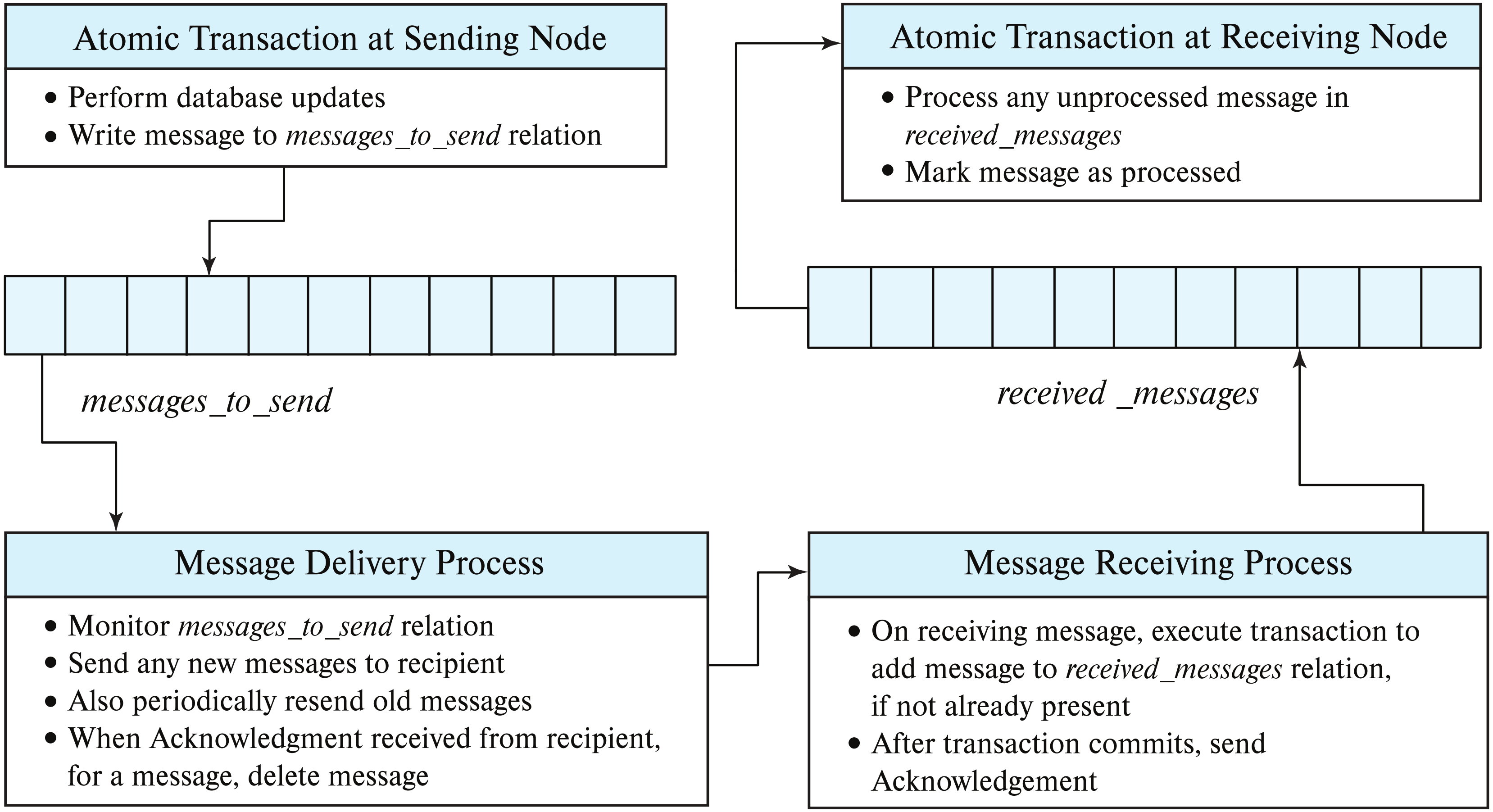
e.g., credit money back to source account.

If sending transaction aborts, message must not be sent

Error Conditions with Persistent Messaging

- Code to handle messages has to take care of variety of failure situations (even assuming guaranteed message delivery)
- E.g., if destination account does not exist, failure message must be sent back to source site
- When failure message is received from destination site, or destination site itself does not exist, money must be deposited back in source account
    - Problem if source account has been closed
        - get humans to take care of problem

Persistent Messaging Implementation



Atomic Transaction at Sending Node

- Perform database updates
- Write message to *messages_to_send* relation

Message Delivery Process

- Monitor *messages_to_send* relation
- Send any new messages to recipient
- Also periodically resend old messages
- When Acknowledgment received from recipient, for a message, delete message

Message Receiving Process

- On receiving message, execute transaction to add message to *received_messages* relation, if not already present
- After transaction commits, send Acknowledgement

Atomic Transaction at Receiving Node

- Process any unprocessed message in *received_messages*
- Mark message as processed

Question 23-1: There is an account A under organization 1 and an account B under organization 2.

a. Show all the steps to transfer Tk. 5000 from account A to account B using persistent messaging protocol.
b. Discuss all types of failures and atomicity issues.

Persistent Messaging (Cont.)

- Receiving site may get duplicate messages after a very long delay
- To avoid keeping processed messages indefinitely
  - Messages are given a timestamp
  - Received messages older than some cutoff are ignored
  - Stored messages older than the cutoff can be deleted at receiving site

Persistent Messaging (Cont.)

- **Workflows** provide a general model of transactional processing involving multiple sites and possibly human processing of certain steps

- E.g., when a bank receives a loan application, it may need to

  - Contact external credit-checking agencies

  - Get approvals of one or more managers

  and then respond to the loan application

- Persistent messaging forms the underlying infrastructure for workflows in a distributed environment