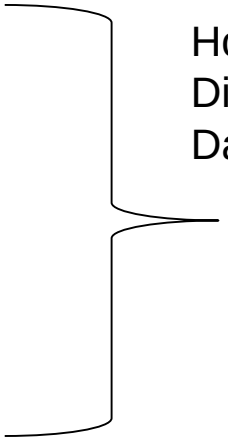
Chapter 19: Distributed Databases

Database System Concepts, 6th Ed.

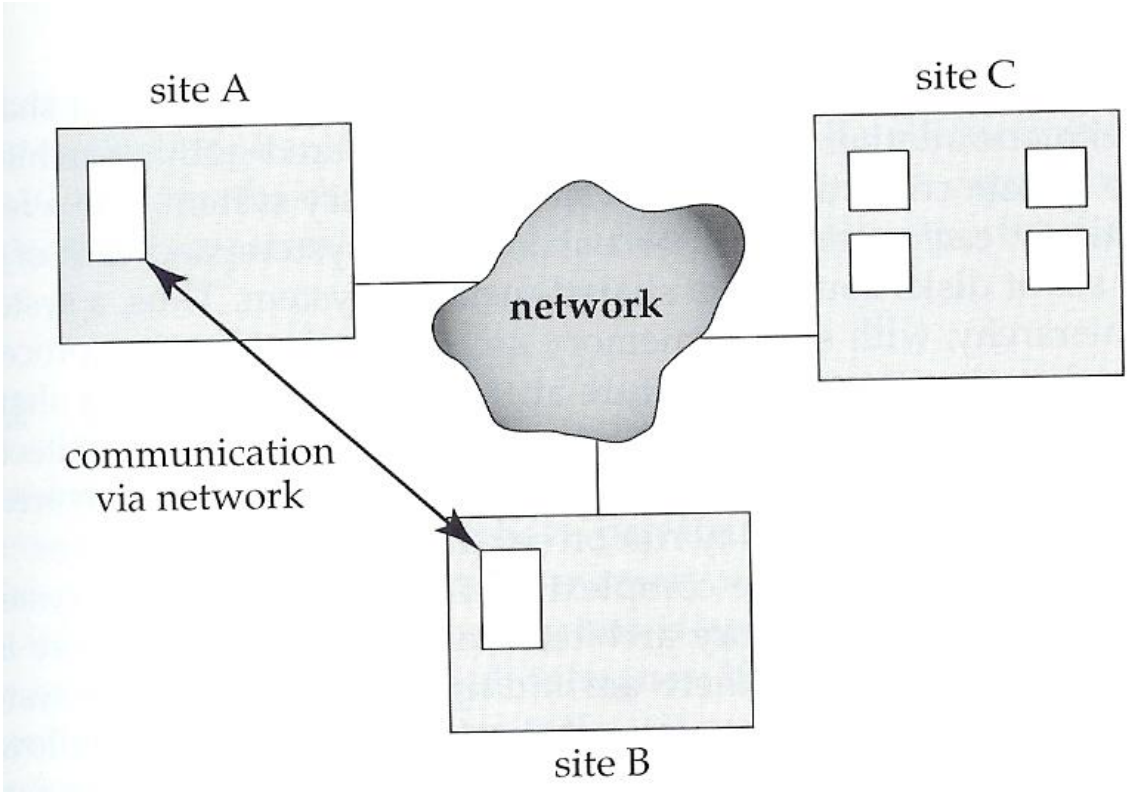
Chapter 19: Distributed Databases

- 19.1 Heterogeneous and Homogeneous Databases
- 19.2 Distributed Data Storage
- 19.3 Distributed Transactions
- 19.4 Commit Protocols
- 19.5 Concurrency Control in Distributed Databases
- 19.6 Availability
- 19.7 Distributed Query Processing
- 19.8 Heterogeneous Distributed Databases
- 19.9 Cloud-Based Databases
- 19.10 Directory Systems



Distributed Database System [1/2]

- A distributed database system consists of loosely coupled sites that share no physical component
  - Database systems that run on each site are independent of each other
  - Transactions may access data at one or more sites



Distributed Database System [2/2]

- In a homogeneous distributed database
  - All sites have identical software
  - Are aware of each other and agree to cooperate in processing user requests.
  - Each site surrenders part of its autonomy in terms of right to change schemas or software
  - Appears to user as a single system
- In a heterogeneous distributed database
  - Different sites may use different schemas and software
    - Difference in schema is a major problem for query processing
    - Difference in software is a major problem for transaction processing
  - Sites may not be aware of each other and may provide only limited facilities for cooperation in transaction processing

Chapter 19: Distributed Databases

- 19.1 Heterogeneous and Homogeneous Databases
- 19.2 Distributed Data Storage
- 19.3 Distributed Transactions
- 19.4 Commit Protocols
- 19.5 Concurrency Control in Distributed Databases
- 19.6 Availability
- 19.7 Distributed Query Processing
- 19.8 Heterogeneous Distributed Databases
- 19.9 Cloud-Based Databases
- 19.10 Directory Systems

Distributed Data Storage

- Assume the relational data model
- Replication
  - System maintains multiple copies of data, stored in different sites, for faster retrieval and fault tolerance
- Fragmentation
  - Relation is partitioned into several fragments stored in distinct sites
- Replication and fragmentation can be combined
  - Relation is partitioned into several fragments: system maintains several identical replicas of each such fragment

Data Replication

- A relation or fragment of a relation is **replicated** if it is stored redundantly in two or more sites
  - Full replication of a relation is the case where the relation is stored at all sites
  - Fully redundant DB means every site contains a copy of the entire DB
- Advantages of Replication
  - Availability: failure of site containing relation *r* does not result in unavailability of *r* is replicas exist
  - Parallelism: queries on *r* may be processed by several nodes in parallel
  - Reduced data transfer: relation *r* is available locally at each site containing a replica of *r*
- Disadvantages of Replication
  - Increased cost of updates: each replica of relation *r* must be updated
  - Increased complexity of concurrency control: concurrent updates to distinct replicas may lead to inconsistent data unless special concurrency control mechanisms are implemented
    - One solution: choose one copy as **primary copy** and apply concurrency control operations on primary copy

Data Fragmentation

- Division of relation r into fragments r_1, r_2, …, r_n which contain sufficient information to reconstruct relation r
- **Horizontal fragmentation**: each tuple of *r* is assigned to one or more fragments
- **Vertical fragmentation**: the schema for relation *r* is split into several smaller schemas
  - All schemas must contain a common candidate key (or superkey) to ensure lossless join property
  - A special attribute, the tuple-id attribute may be added to each schema to serve as a candidate key

- Example : Bank database with following schema

  Account = (*branch_name*, *account_number, balance*)

  Employee_Info = (employee_id, name, destination, salary)

Horizontal Fragmentation of *account* Relation

Account = (*branch_name*, *account_number, balance*)

$account_1 = \sigma_{branch\_name=\text{“Hillside”}}(account)$

branch_name	*account_number*	*balance*
Hillside	A-305	500
Hillside	A-226	336
Hillside	A-155	62

$account_2 = \sigma_{branch\_name=\text{“Valleyview”}}(account)$

branch_name	*account_number*	*balance*
Valleyview	A-177	205
Valleyview	A-402	10000
Valleyview	A-408	1123
Valleyview	A-639	750

Vertical Fragmentation of *employee_info* Relation

Employee_Info = (branch_name, customer_name, account_number, balance)

$deposit_1 = \Pi_{branch\_name,\ customer\_name,\ tuple\_id}(employee\_info)$

branch_name	customer_name	tuple_id
Hillside	Lowman	1
Hillside	Camp	2
Valleyview	Camp	3
Valleyview	Kahn	4
Hillside	Kahn	5
Valleyview	Kahn	6
Valleyview	Green	7

$deposit_2 = \Pi_{account\_number,\ balance,\ tuple\_id}(employee\_info)$

account_number	balance	tuple_id
A-305	500	1
A-226	336	2
A-177	205	3
A-402	10000	4
A-155	62	5
A-408	1123	6
A-639	750	7

Advantages of Fragmentation

- Horizontal fragmentation:
  - allows parallel processing on fragments of a relation
  - allows a relation to be split so that tuples are located where they are most frequently accessed
- Vertical fragmentation:
  - allows tuples to be split so that each part of the tuple is stored where it is most frequently accessed
  - tuple-id attribute allows efficient joining of vertical fragments
  - allows parallel processing on a relation
- Vertical and horizontal fragmentation can be mixed
  - Fragments may be successively fragmented to an arbitrary depth

Data Transparency

- **Data transparency**
  - Degree to which system user may remain unaware of the details of how and where the data items are stored in a distributed system
- Consider transparency issues in relation to:
  - Fragmentation transparency: unware of how a relation is fragmented
  - Replication transparency: unware of what data have been duplicated
  - Location transparency: unware of physical location of data items

- Naming of data items in distributed data environment

  Criteria 1. Every data item must have a system-wide unique name

  Criteria 2. It should be possible to find the location of data items efficiently

  Criteria 3. It should be possible to change the location of data items transparently

  Criteria 4. Each site should be able to create new data items autonomously

Naming a relation, naming a tuple in a distributed database

Naming Schemes [1/2]

- Central Name-Server Scheme
  - Structure:
    - Name-server assigns all names and the list of names
    - Each site maintains a record of local data items
    - Sites ask name-server to locate non-local data items
  - Advantages:
    - Satisfies the naming criteria 1, 2, and 3
  - Disadvantages:
    - Does not satisfy the naming criteria 4
    - Name-server is a potential performance bottleneck
    - Name-server is a single point of failure

Naming Schemes [2/2]

- Site-Id Prefix Scheme
  - Each site prefixes its own site identifier to any name that it generates
    - i.e., *site* _17.*account*
  - Fulfills having a unique identifier, and avoids problems associated with central control
  - However, fails to achieve network transparency

- Alias Scheme:
  - Create a set of **aliases** for data items
    - *i.e., site* _17.*account* ➔ *snu_acct*
  - Each site stores the mapping of aliases to the real names

alias	Physical addr
snu_acct	*site* _17.*account*
…	…

  - With aliases, the user can be
    - unaware of the physical location of a data item
    - unaffected if the data item is moved from one site to another

Chapter 19: Distributed Databases

- 19.1 Heterogeneous and Homogeneous Databases
- 19.2 Distributed Data Storage
- 19.3 Distributed Transactions
- 19.4 Commit Protocols
- 19.5 Concurrency Control in Distributed Databases
- 19.6 Availability
- 19.7 Distributed Query Processing
- 19.8 Heterogeneous Distributed Databases
- 19.9 Cloud-Based Databases
- 19.10 Directory Systems

Distributed Transactions

- Local transactions are those that access and update data in only one local database
- Global transactions are those that access and update data in several local databases
  - May use the term “distributed transactions”
  - May consist of several local transactions
  - Are controlled by the transaction coordinator initiating the global transaction

- Issues
  - Atomicity of global transactions (ch 19.3)
    - Recovery of global transactions
    - Commit of global transactions
  - Concurrency control of global transactions (ch 19.4)

Global Transaction GT와 GT에 속한 Local Transaction LT들은 운명을 함께 한다!!!

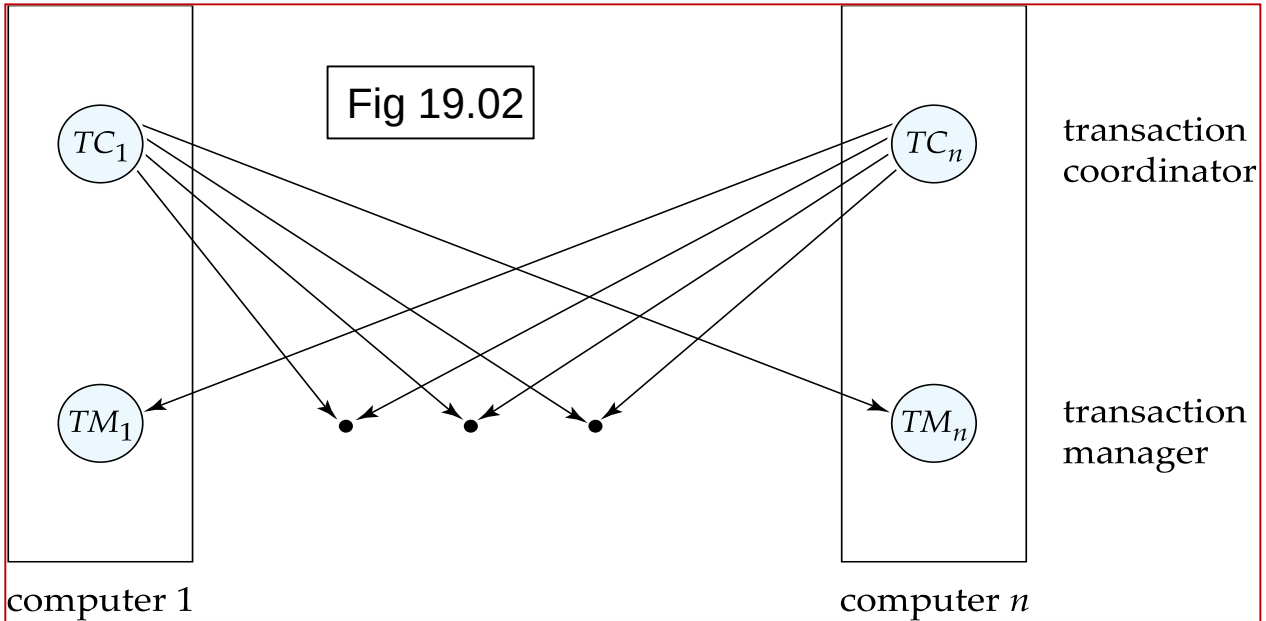
Distributed Transaction 예제

- Autonomous한 분산DB환경에서의 Distributed Transactions
  - 예1) Money transfer between different banks (우리은행 ➔ 하나은행)
  - 예2) 농협의 전국 지점 distributed database에서 환전고객들에 대한 공식환률을 변경시키는 업무 (매시간별로 변동한다고 가정하고)
  - 예3) A user "Kim" is planning a travel from C1 to C3 through C2 with Korean Air
    - C1 ➔ C2 : Korean Air is using Oracle Database
    - C2 ➔ C3: Delta Air is using IBM DB2 Database
    - Suppose Korean Air Reservation System is arranging the whole itinerary
    - Global transaction GT: buying tickets from C1 to C3 through C2
    - Local transaction LT1: buying a ticket from C1 to C2 in Korean Air Database
    - Local transaction LT2: buying a ticket from C2 to C3 in Delta Air Database
  - 예4) 전세계 26개 KAL지점에서 flight seat database를 copy해서 가지고 있다면 어느지점에서 flight seat 1개를 팔았다면 전세계 26개 KAL 지점의 database copy에 똑 같이 반영되야 하는 distributed transaction

- Primary site P with back-up site B 같은 분산환경에서는 Distributed Transaction 은 해당이 없음
  - P site performs transaction T, and P is down, then T is transferred to B site

System Structure of Distributed Databases

- Each site has a local transaction manager responsible for:
  - Maintaining a log for recovery purposes
  - Participating in coordinating the concurrent execution of the transactions executing at that site
- Each site has a transaction coordinator responsible for:
  - Starting the execution of transactions that are originated at the site
  - Distributing subtransactions at appropriate sites for execution
  - Coordinating the termination of each transaction that originates at the site, which may result in the transaction being committed at all sites or aborted at all sites



Failure Modes in Distributed System

- Distributed transactions should cope with various failures
  - Distributed commit protocol includes recovery from failures
  - Distributed concurrency control protocol
  - 예3) A user “Kim” is planning a travel from C1 to C3 through C2 with Korean Air

- 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

Chapter 19: Distributed Databases

- 19.1 Heterogeneous and Homogeneous Databases
- 19.2 Distributed Data Storage
- 19.3 Distributed Transactions
- 19.4 Commit Protocols
- 19.5 Concurrency Control in Distributed Databases
- 19.6 Availability
- 19.7 Distributed Query Processing
- 19.8 Heterogeneous Distributed Databases
- 19.9 Cloud-Based Databases
- 19.10 Directory Systems

Commit Protocols for Global Transaction

- How about atomicity (all-or-nothing) of global transaction?
- Commit protocols are used to ensure atomicity across distributed sites
  - a transaction which executes at multiple sites must either be committed at all the sites, or aborted at all the sites
  - not acceptable to have a transaction committed at one site and aborted at another
- The *two-phase commit* (2PC) protocol is widely used

- The *three-phase commit* (3PC) protocol
  - more complicated and more expensive
  - but avoids some drawbacks of two-phase commit protocol
  - This protocol is not used in practice

Two Phase Commit Protocol (2PC)

- Assumes **the fail-stop model** in 2PC
  - Failed-sites simply stop working,
  - and do not cause any other harm to other sites (such as sending incorrect messages)

- Execution of the 2PC protocol is initiated by the coordinator after the last step of the transaction has been reached
  - The 2PC protocol involves all the local sites at which the transaction executed
  - Let T be a transaction initiated at site S_i
  - Let the transaction coordinator at S_i be C_i

Phase 1 of 2PC: Obtaining a Decision

Let *T* be a transaction initiated at site S_i, and let the transaction coordinator at S_i be C_i

- Coordinator asks all participants to *prepare* to commit transaction T_i
  - C_i adds the records <**prepare** *T*> to the log and forces them 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
  - if not, add a record <**no** *T*> to the log and send "**abort** *T*" message to C_i
  - if the transaction can be committed, then:
    - add the record <**ready** *T*> to the log
    - force *all records* for *T* to stable storage
    - send "**ready** *T*" message to C_i

Phase 2 of 2PC: Recording the Decision

Let *T* be a transaction initiated at site S_i, and let the transaction coordinator at S_i be C_i

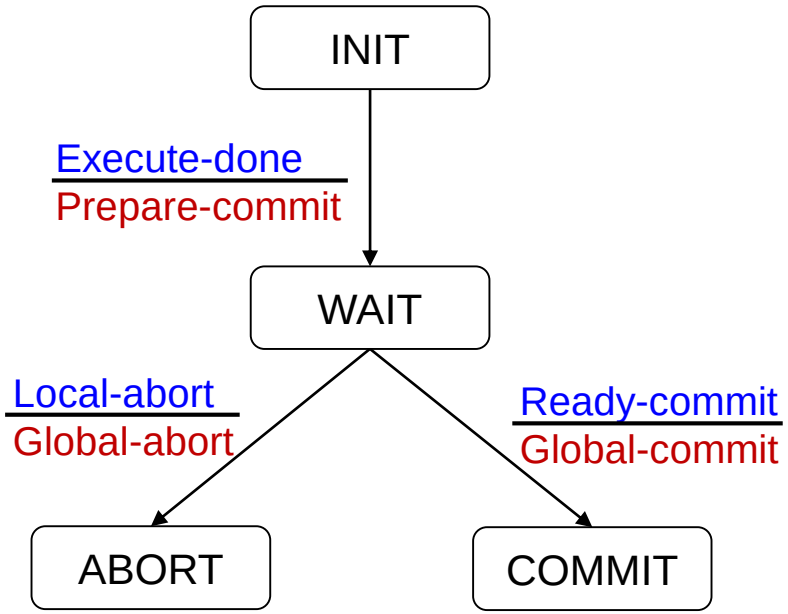
- *T* can be committed if C_i received a **ready** *T* message from all the participating sites: otherwise *T* must be aborted
- Coordinator adds a decision record, <**commit** *T*> or <a**bort** *T*>, to the log and forces the decision record onto stable storage
  - Once the record is in stable storage it is irrevocable (even if failures occur)
- Coordinator sends a decision message to each participant informing it of the decision (commit or abort)

  ➔ At this moment, T is committed or aborted according to the decision
- Participants take appropriate action locally
- Participants send ACK message to C_i (this step is rather inessential)

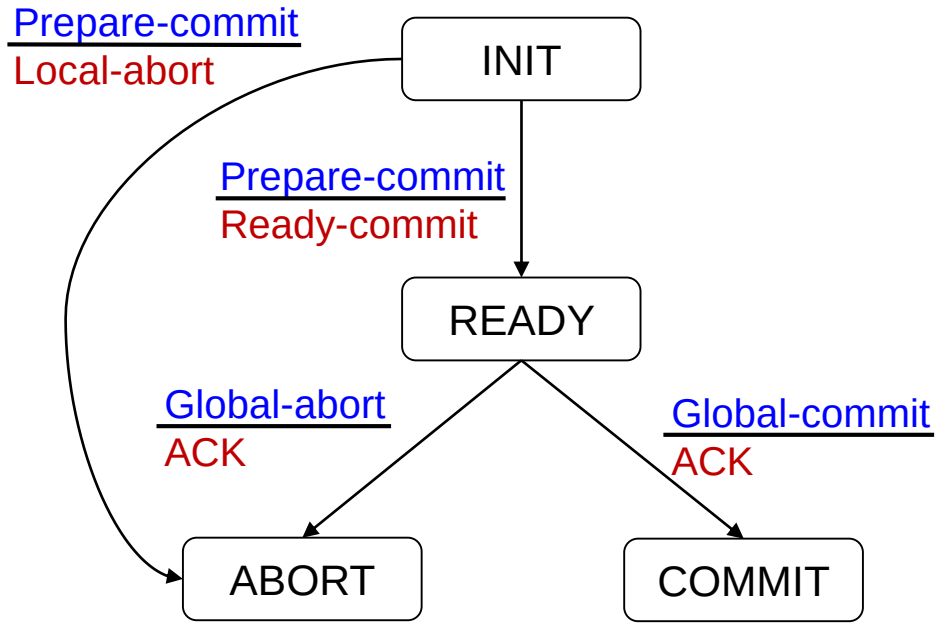
2PC: Finite State Machine

Input message

output message



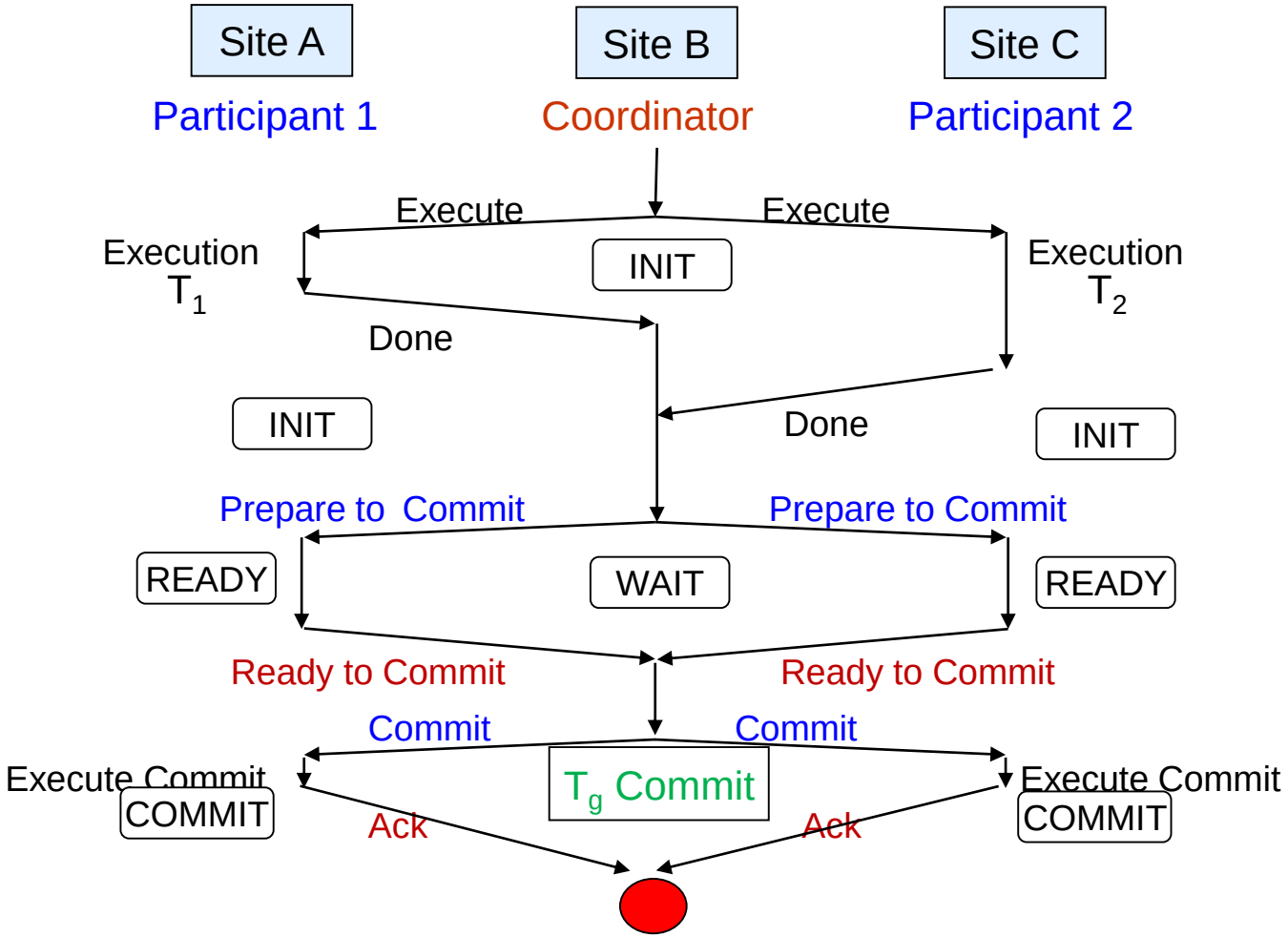
Coordinator



Each participant

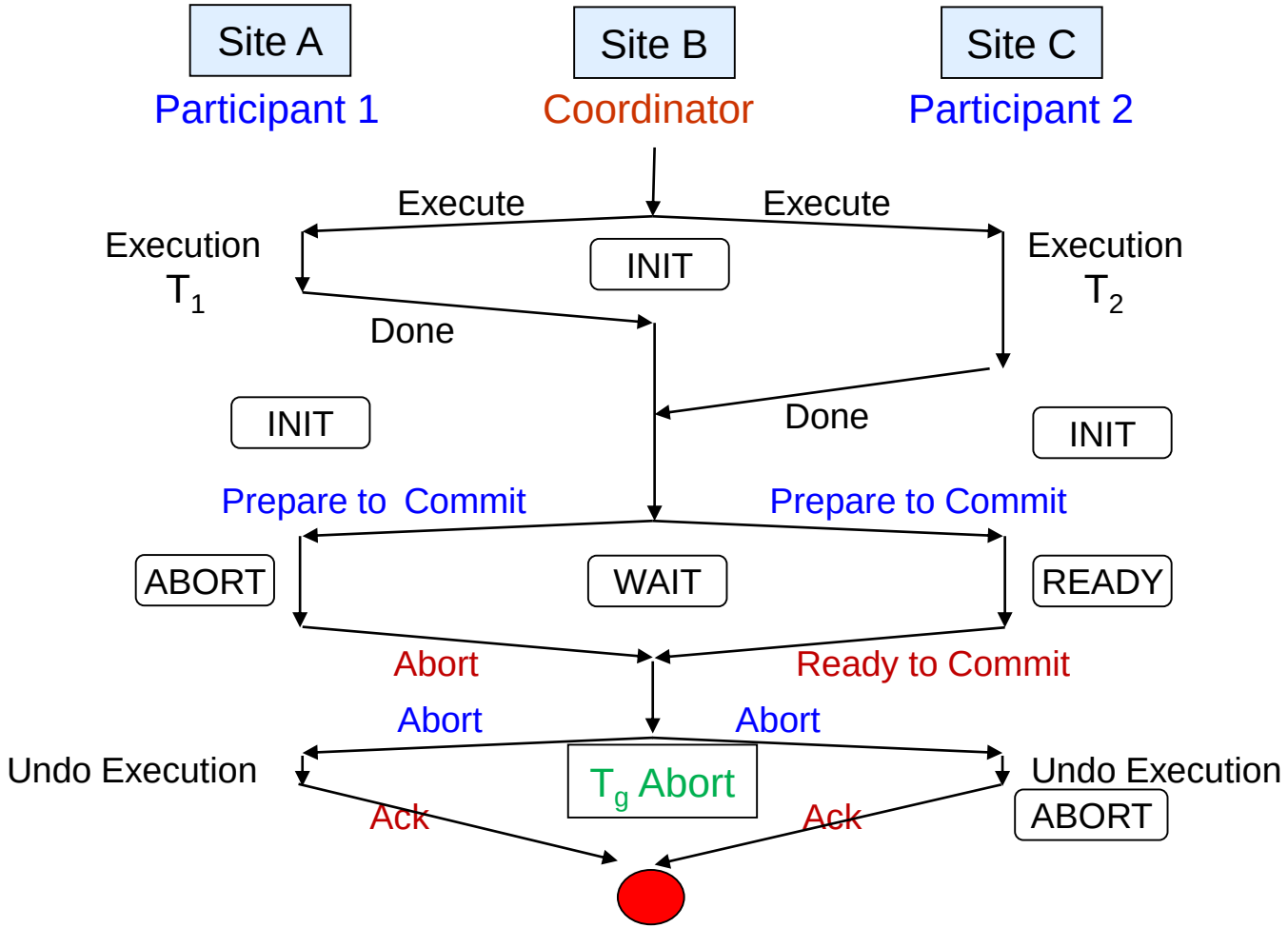
2PC: Commit

Assume the coordinator received a global transaction T_g, which is composed of local transaction T_1 and T_2



2PC: Abort

Assume the coordinator received a global transaction T_g, which is composed of local transaction T_1 and T_2



Handling of Failures in 2PC – Participating Site Failure

When a participating site S_k recovers, it examines its log to determine the fate of transactions active at the time of the failure

- Case1: Log contains <**commit** *T*> record: the site S_k executes **redo** (*T*)
- Case 2: Log contains <**abort** *T*> record: the site S_k executes **undo** (*T*)
- Case 3: Log contains <**ready** *T*> record:

  The site S_k must consult C_i to determine the fate of *T*
  - If *T* committed, **redo** (*T*)
  - If *T* aborted, **undo** (*T*)
- Case 4: If the log contains no control records (commit, abort, ready) concerning *T:*

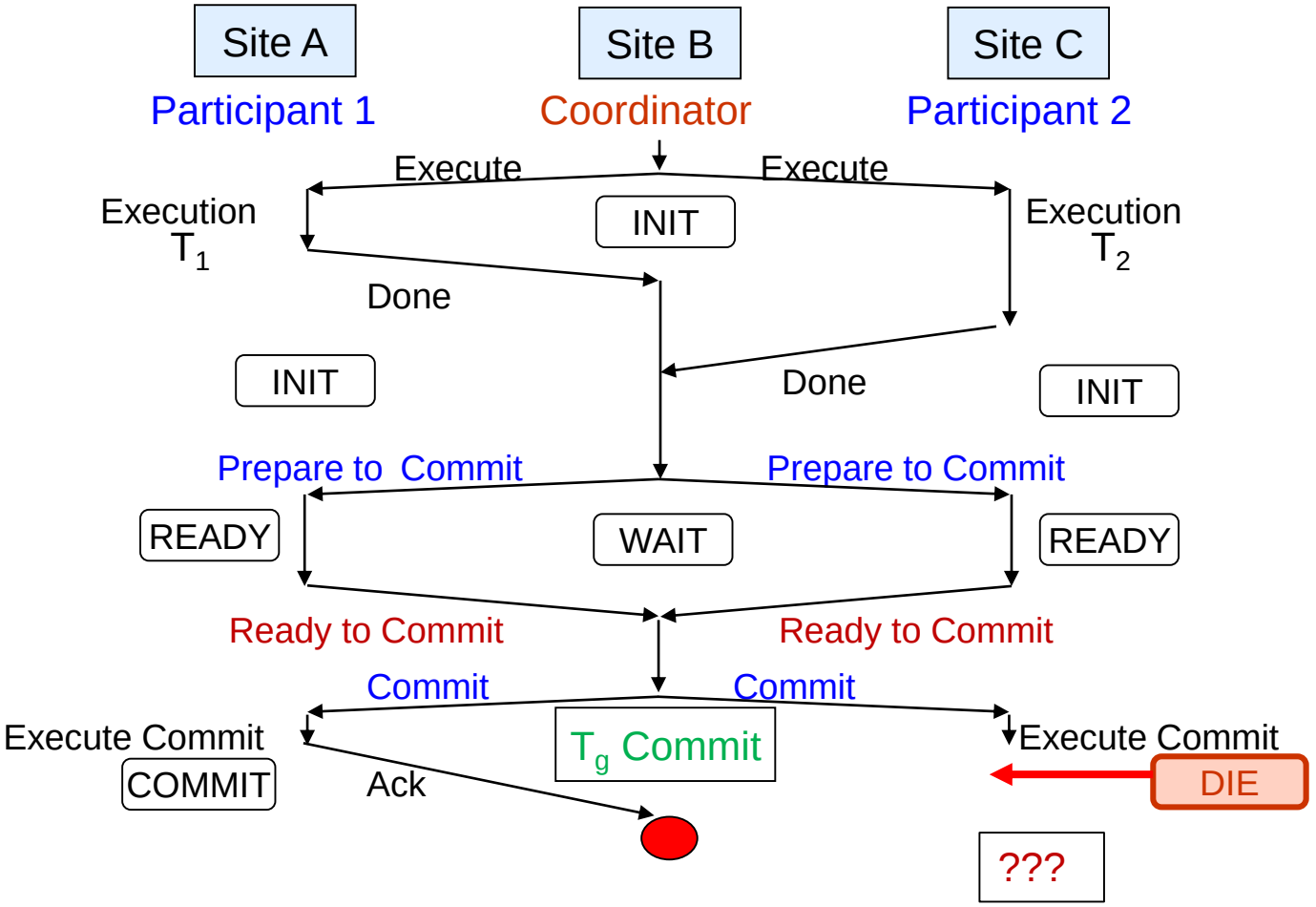
  It implies that S_k failed before responding to the **prepare** *T* message from C_i
  - Since the failure of S_k precludes the sending of reply, C_1 must have aborted *T*
  - Therefore, S_k must execute **undo** (*T*)

Handling Failure of a Participating Site in 2PC [1/4]

CASE 1: Participant 2 dies **during commit execution**

Participant 2 knew the commit before it dies

→ During recovery, Participant 2 must **redo** T_2

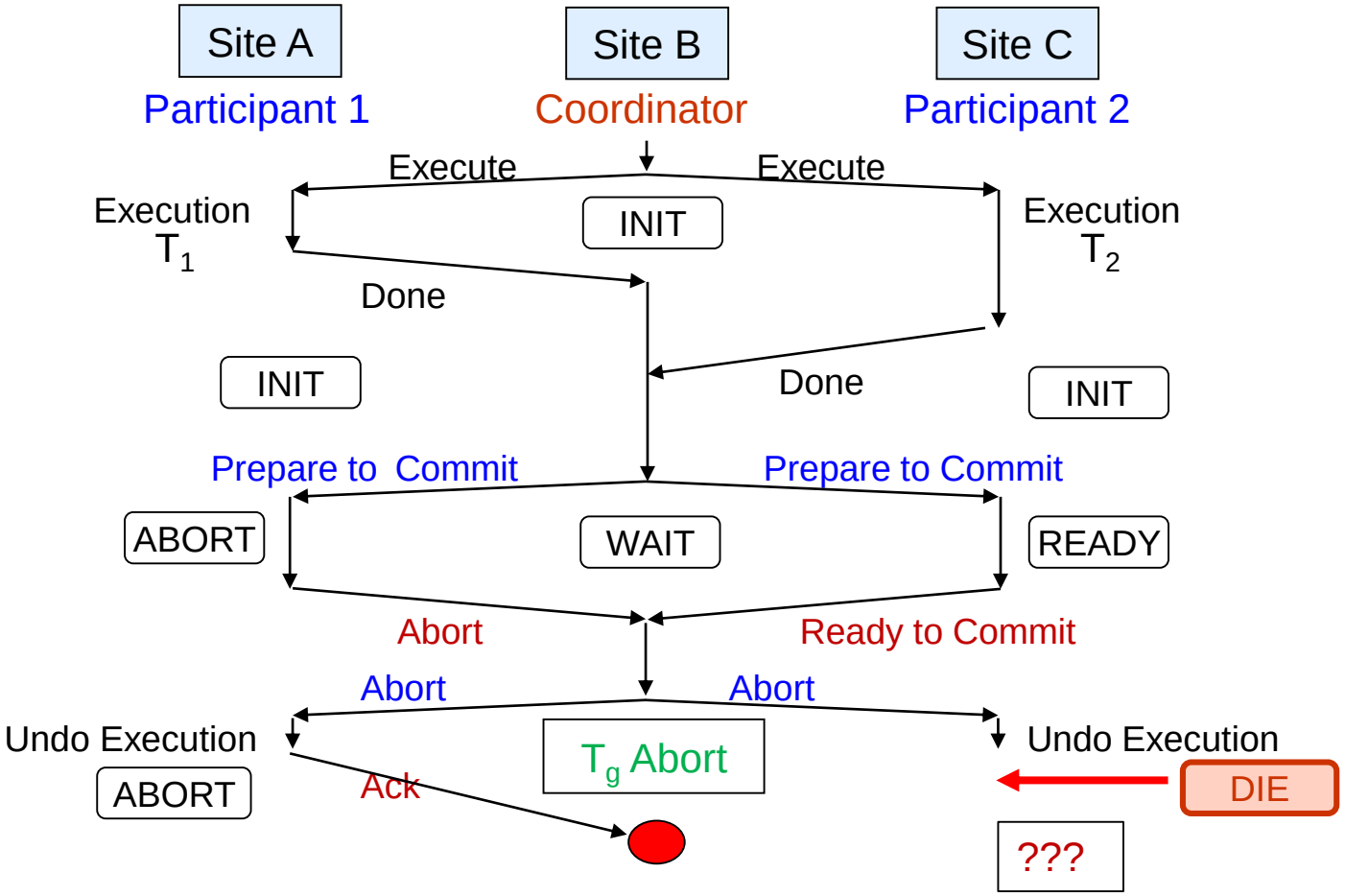


Handling Failure of a Participating Site in 2PC [2/4]

CASE 2: Participant 2 dies **during undo execution**

Participant 2 knew the abort before it dies

→ During recovery, Participant 2 must **undo** T_2



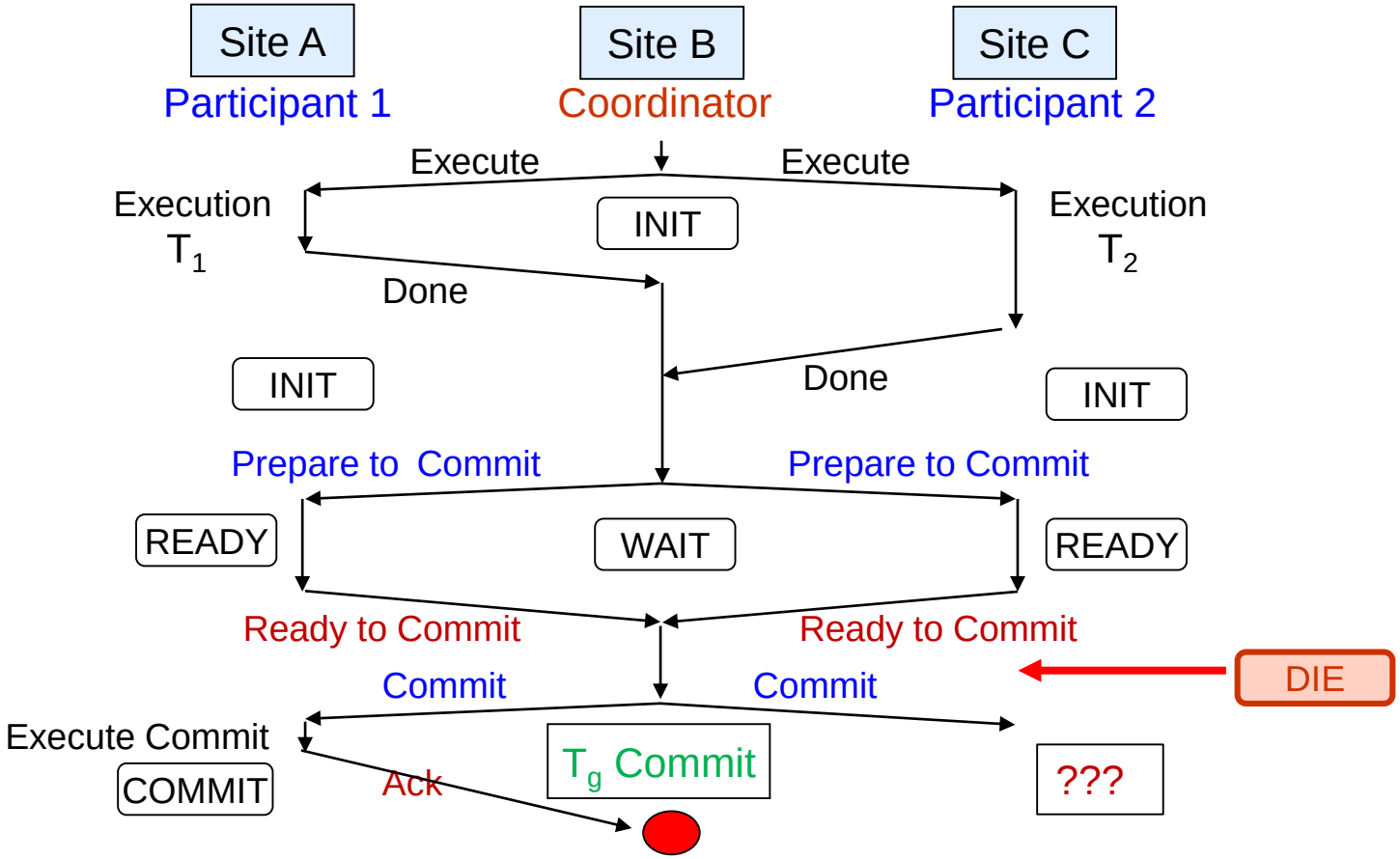
Handling Failure of a Participating Site in 2PC [3/4]

CASE 3: Participant 2 dies **during ready state, before receiving commit/abort message**

Participant 2 don't know the fate of the transaction

→ During recovery, Participant 2 must **consult the coordinator**

(redo T_2 if T_g is commited, undo T_2 if T_g is aborted.)

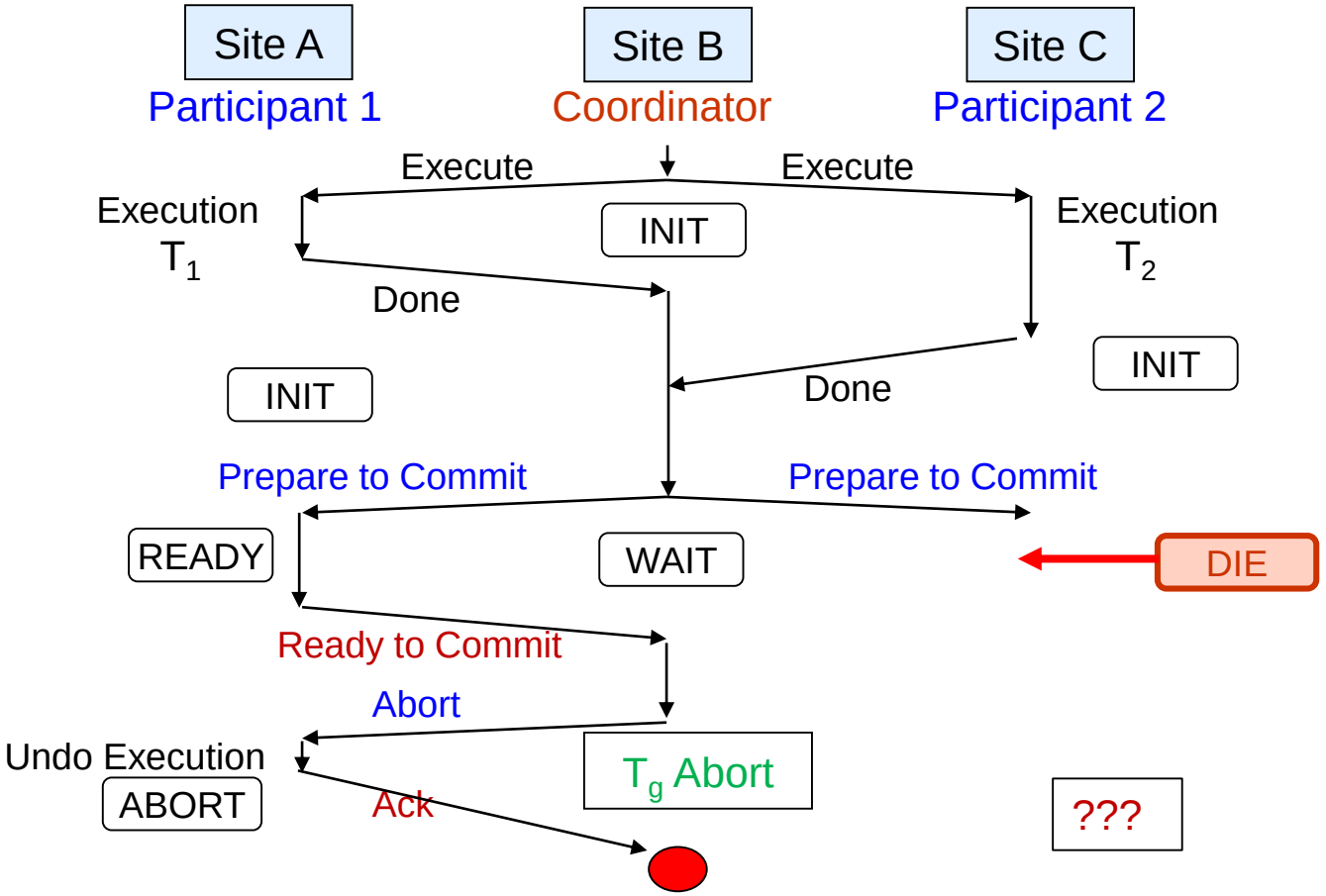


Handling Failure of a Participating Site in 2PC [4/4]

CASE 4: Participant 2 dies **before it responds to the prepare to commit message**

T_g cannot be commited because Participant 2 didn't send a ready to commit message

→ During recovery, Participant 2 must **undo** T_2 without consulting with the coordinator



Handling of Failures in 2PC: Coordinator Failure

- If the coordinator C_i fails while the commit protocol for T is executing, participating sites must decide on T's fate by asking each other:
  1. If an active site contains a <**commit** T> record in its log, then T must be committed
  2. If an active site contains an <**abort** T> record in its log, then T must be aborted
  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
     1. Rather than wait for C_i to recover, it is preferable to abort T
  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> of <**commit** T>)
     1. In this case active sites must wait for C_i to recover, to find decision

- **Blocking problem**: active sites may have to wait for the failed coordinator to recover

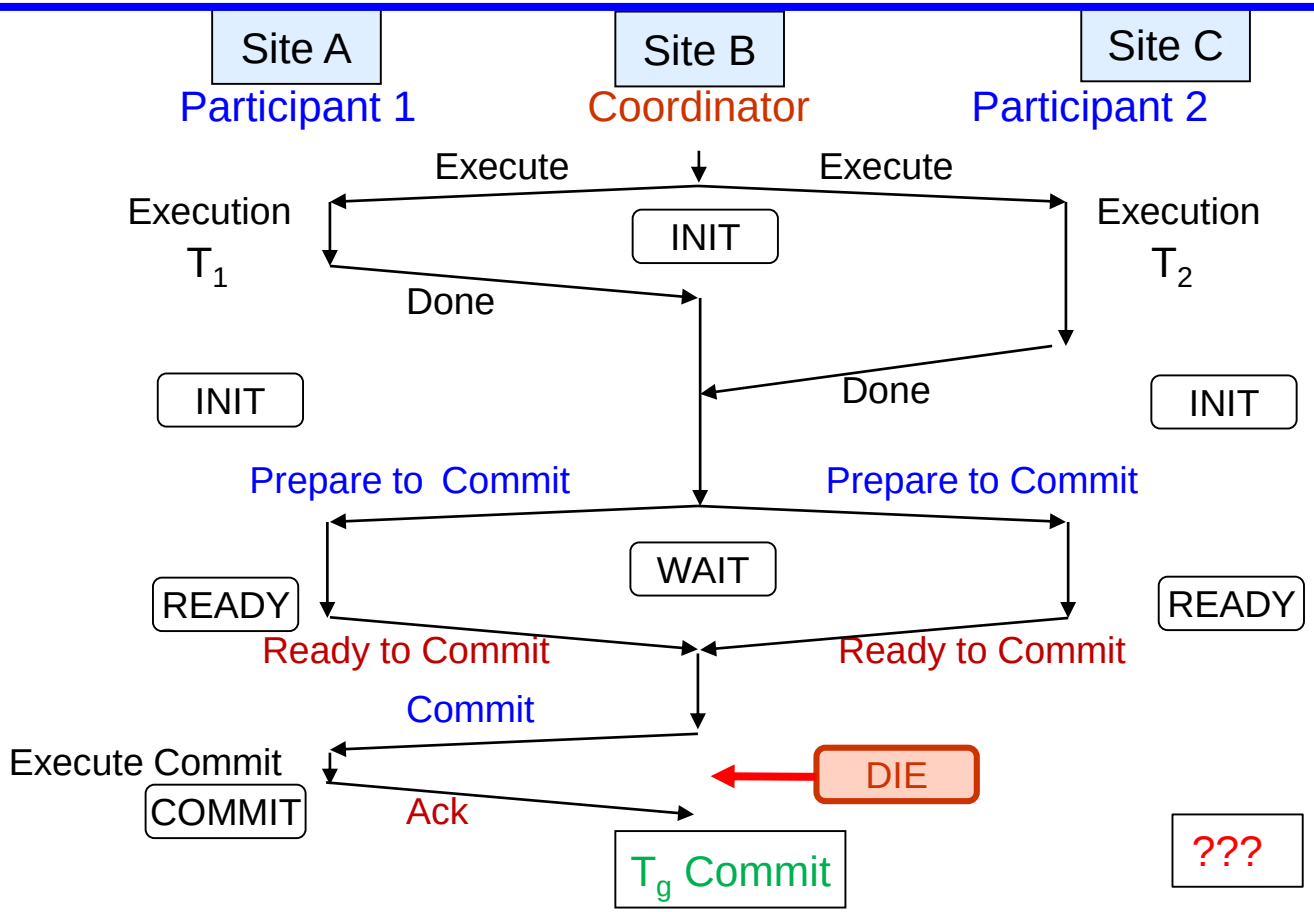
Handling Coordinator Failure in 2PC [1/4]

CASE 1: Suppose the coordinator dies **while it sends "commit" messages and an active site is in COMMIT state**

* When the coordinator fails, Participant 2 asks Participant 1

→ If Participant 1 has <commit> log record ➔ Participant 2 just **commit** T_2

* The coordinator will commit T_g after recovery



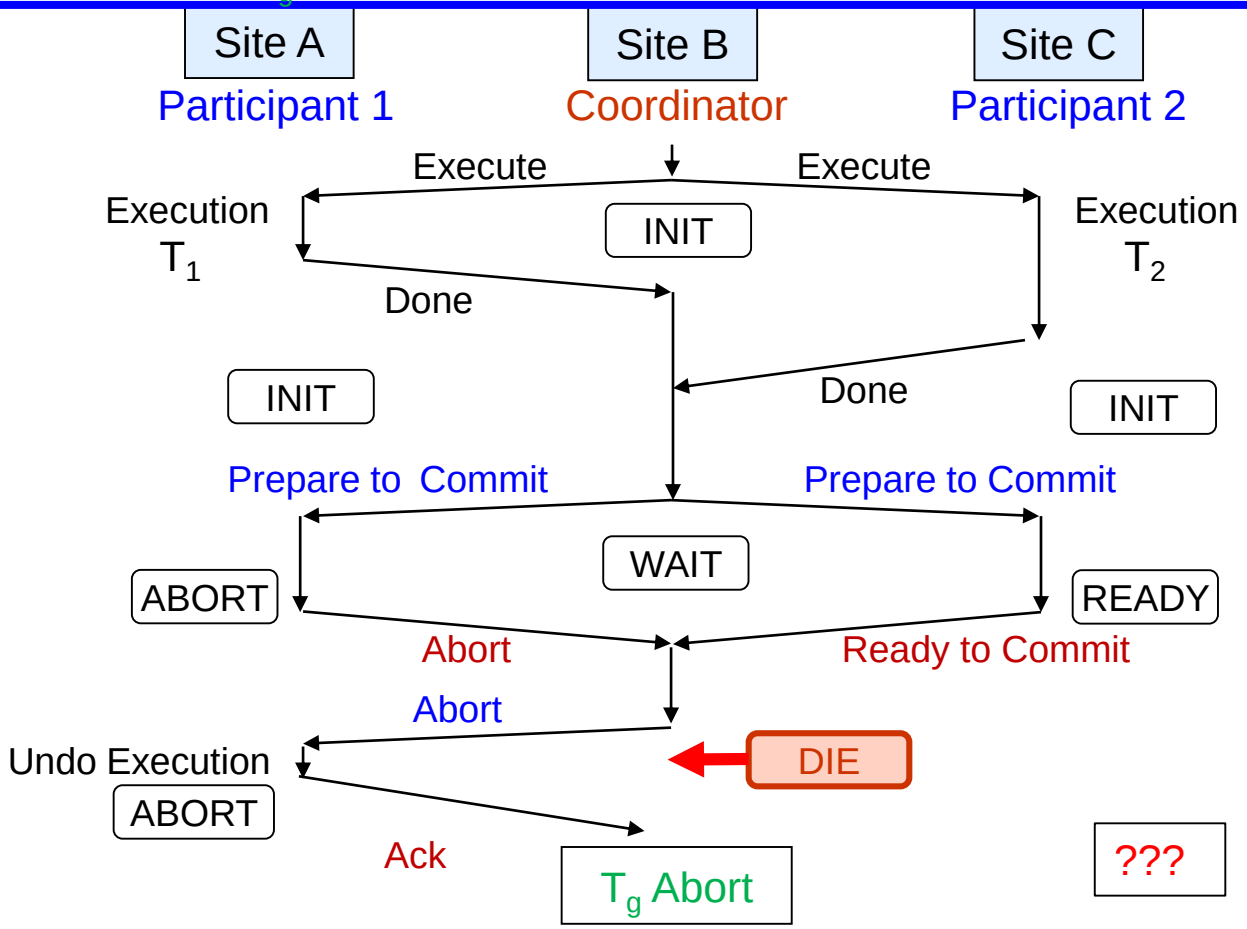
Handling Coordinator Failure in 2PC [2/4]

CASE 2: Suppose the coordinator dies **while it sends "abort" messages and an active site is in ABORT state**)

* When coordinator fails, Participant 2 asks Participant 1

→ If Participant 1 has <abort> log record ➔ Participant 2 just **abort** T_2

* The coordinator will abort T_g after recovery



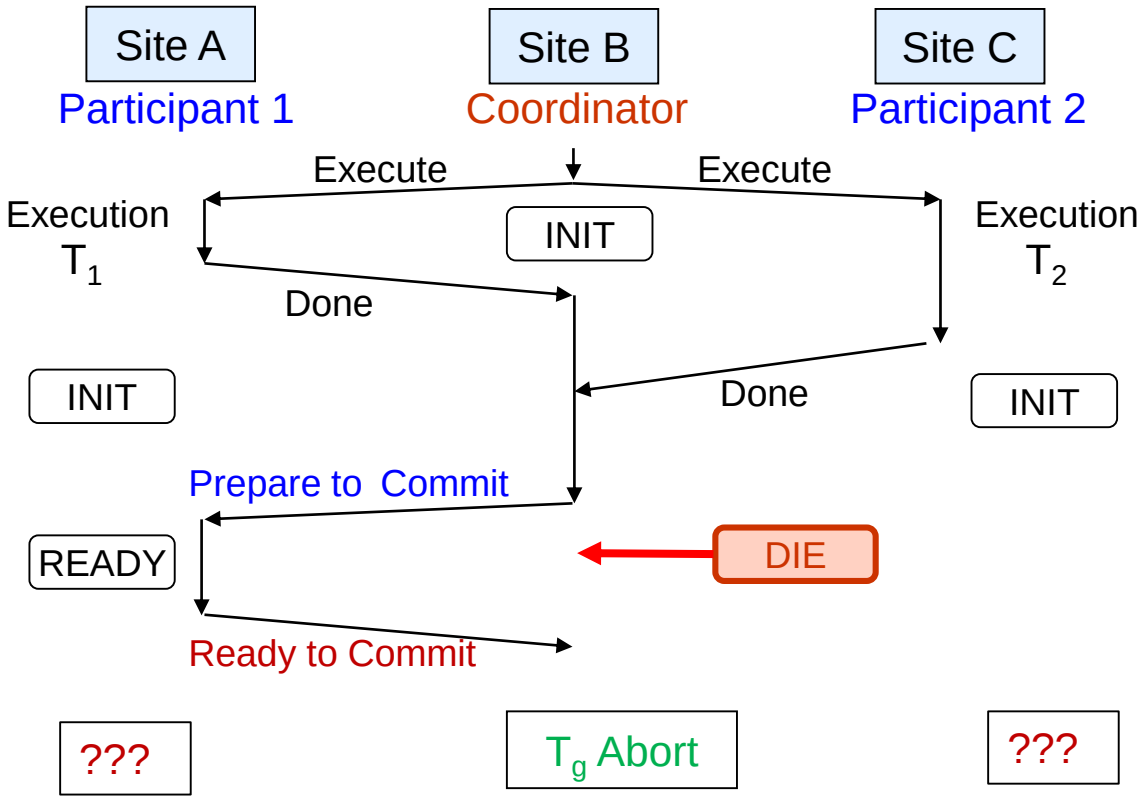
Handling Coordinator Failure in 2PC [3/4]

CASE 3: Coordinator dies **while it sends "prepare to commit" messages** (i.e., **some active sites are in INIT state**)

T_g cannot be committed because Participant 2 didn't receive a "prepare to commit" message

→ All participants must **abort** the transactions

* The coordinator will abort T_g after recovery



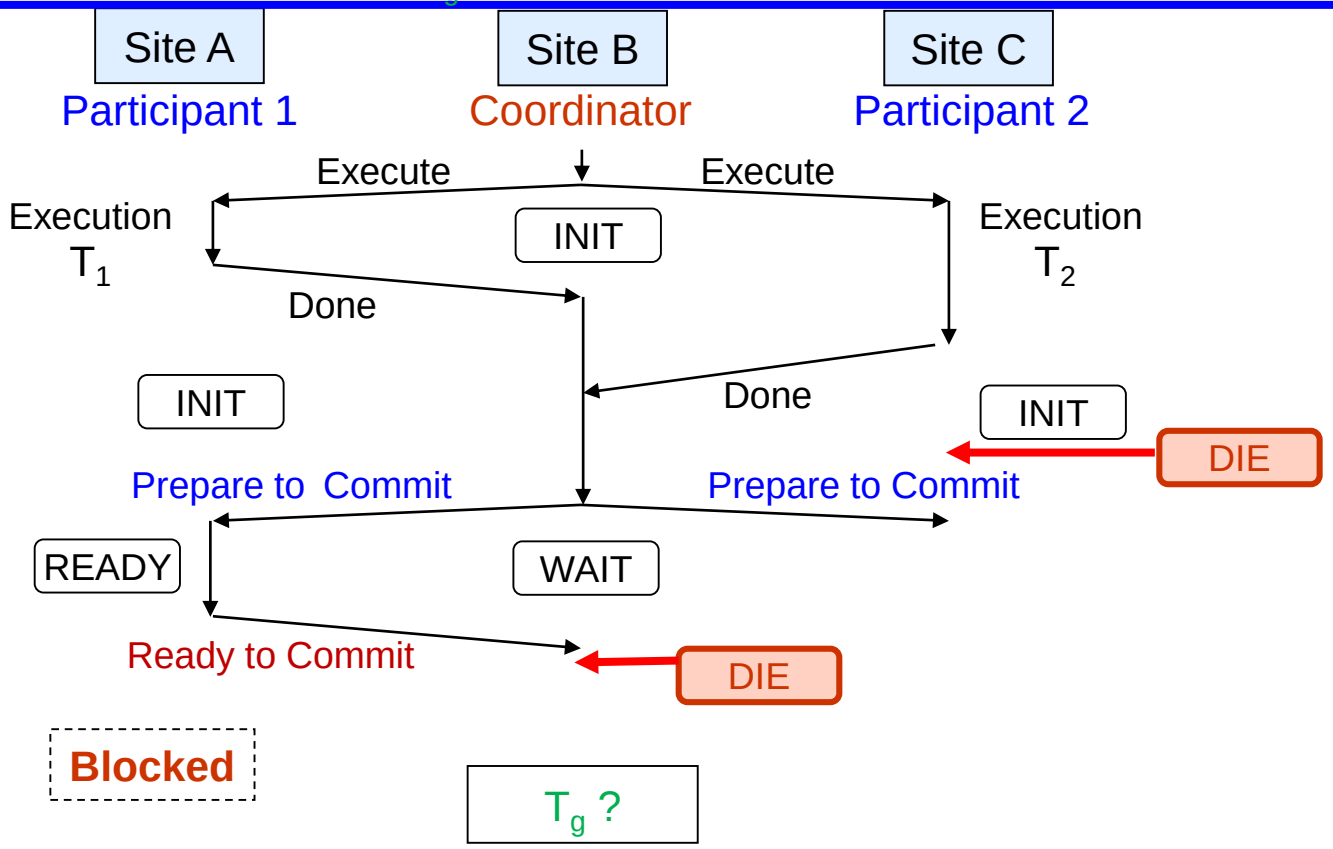
Handling Coordinator Failure in 2PC [4-a / 4]

CASE 4-a: Suppose Coordinator dies and **all active sites are in READY state and** some dead sites may be in **INIT** state, or coordinator may not have received all votes before crash)

* When coordinator fails, all participants don't know the fate of the transaction

→ **Blocking**: all participants cannot proceed until either the coordinator or dead sites are available

* The coordinator cannot decide the fate T_g after recovery



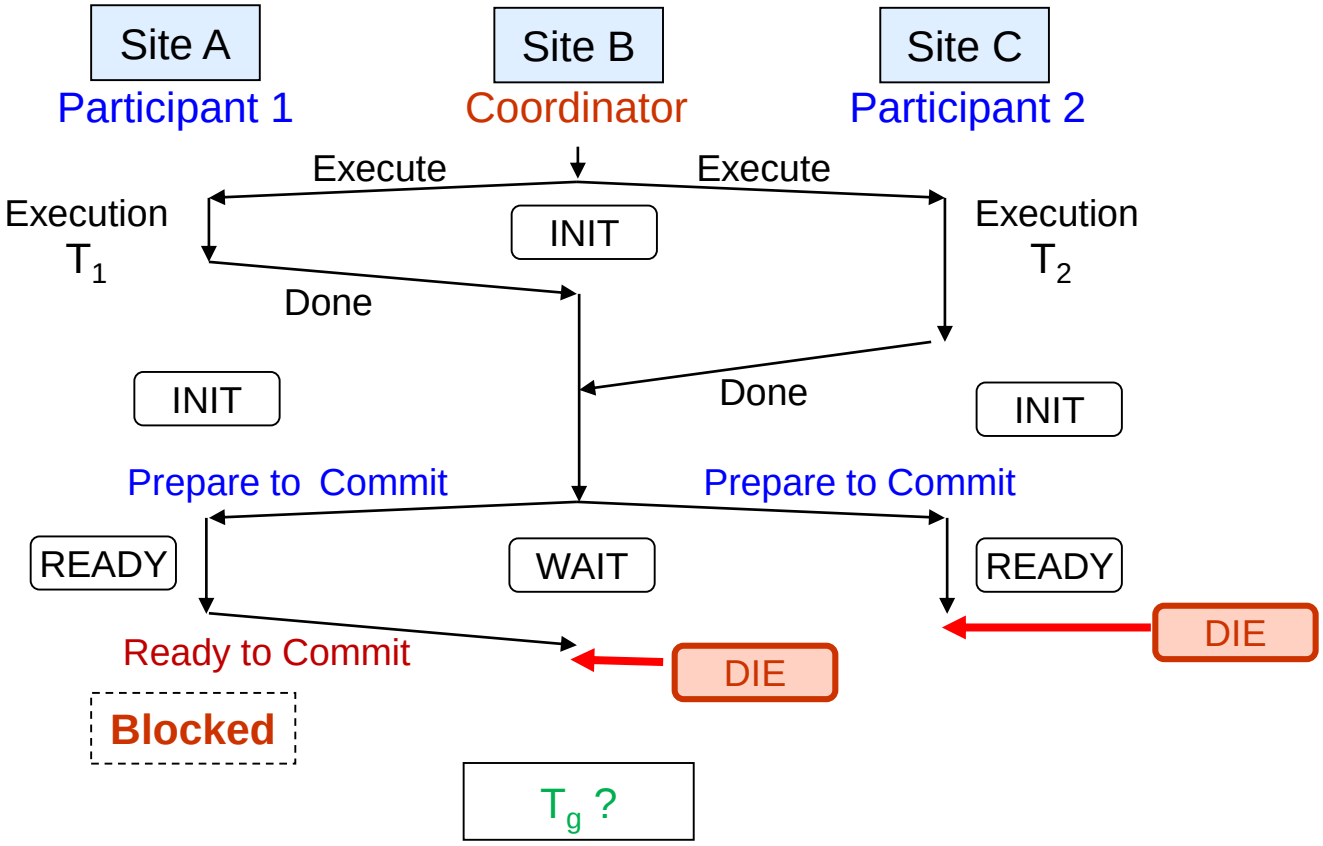
Handling Coordinator Failure in 2PC [4-b / 4]

CASE 4-b: Suppose the coordinator dies and **all active sites are in READY state and** some dead sites may be in **READY** state, or coordinator may not have received all votes before crash)

* When coordinator fails, all participants don't know the fate of the transaction

→ **Blocking** – all participants cannot proceed until either the coordinator or dead sites are available

* The coordinator cannot decide the fate T_g after recovery



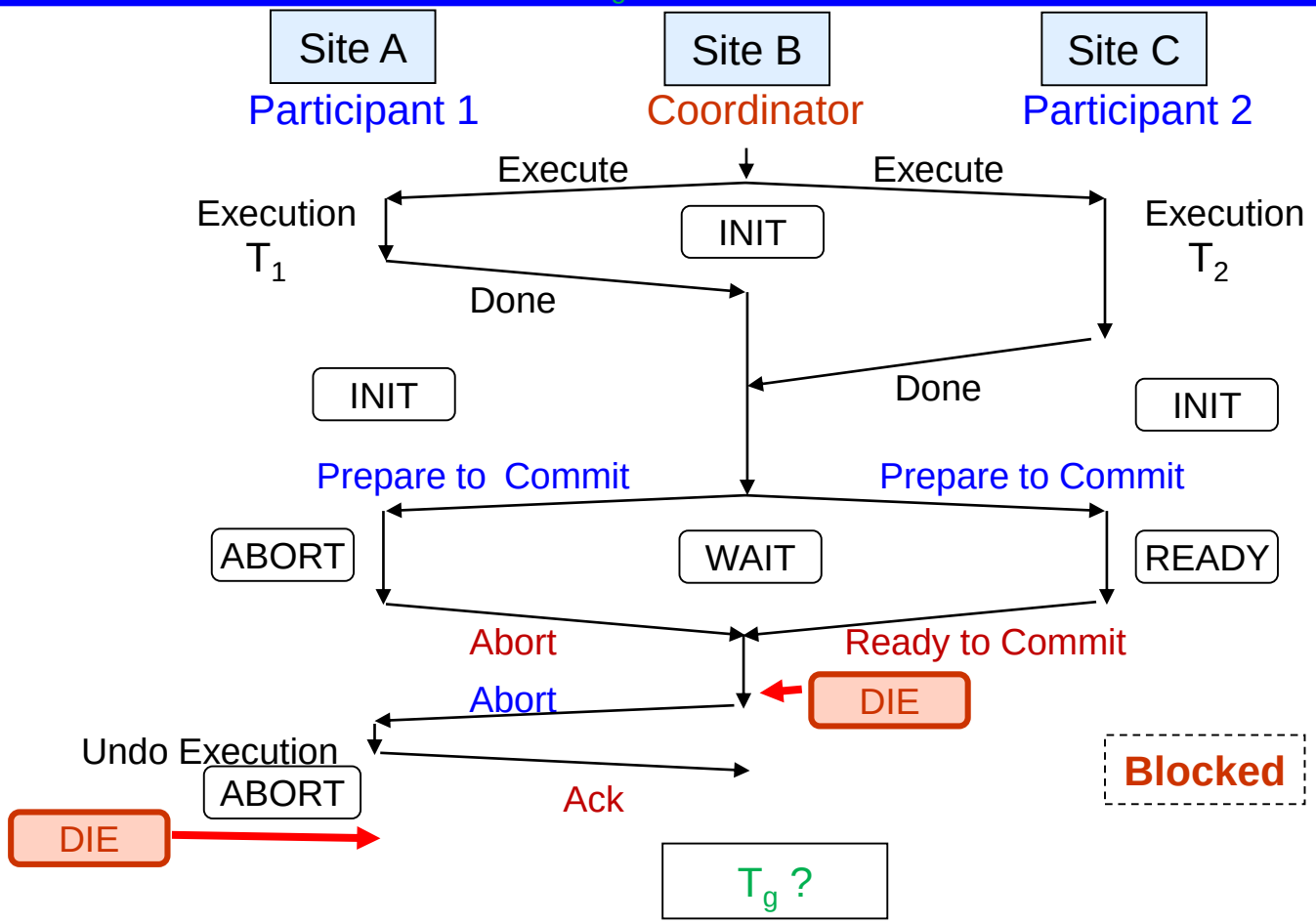
Handling Coordinator Failure in 2PC [4-c / 4]

CASE 4-c: Suppose the coordinator dies and **all active sites are in READY state and** some dead sites may be in **ABORT** state, or coordinator may not have received all votes before crash)

* When coordinator fails, all participants don't know the fate of the transaction

→ **Blocking**: all participants cannot proceed until either the coordinator or dead sites are available

* The coordinator cannot decide the fate T_g after recovery



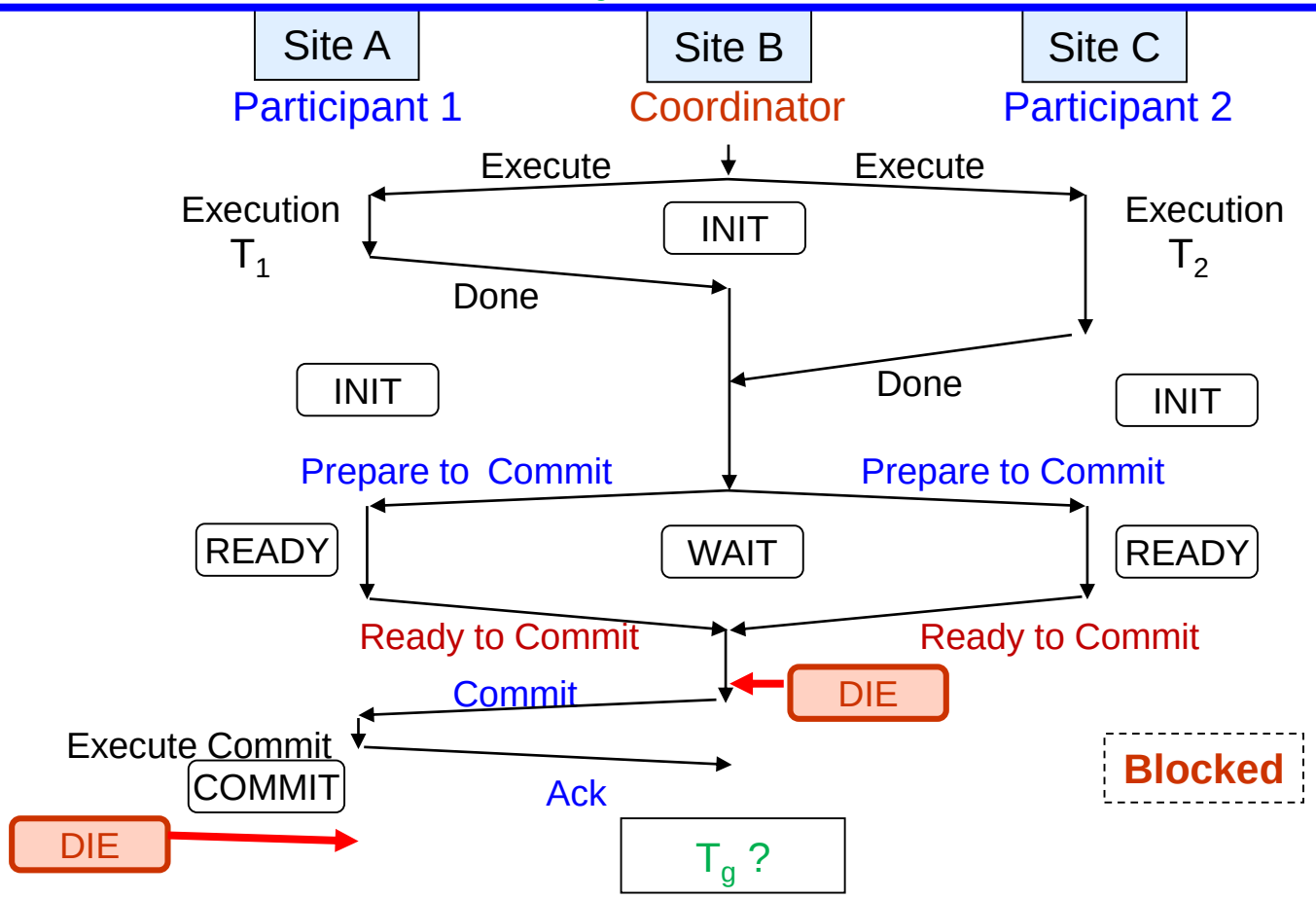
Handling Coordinator Failure in 2PC [4-d / 4]

CASE 4-d: Suppose the coordinator dies and **all active sites are in READY state and** some dead sites may be in **COMMIT** state, or coordinator may not have received all votes before crash)

* When coordinator fails, all participants don't know the fate of the transaction

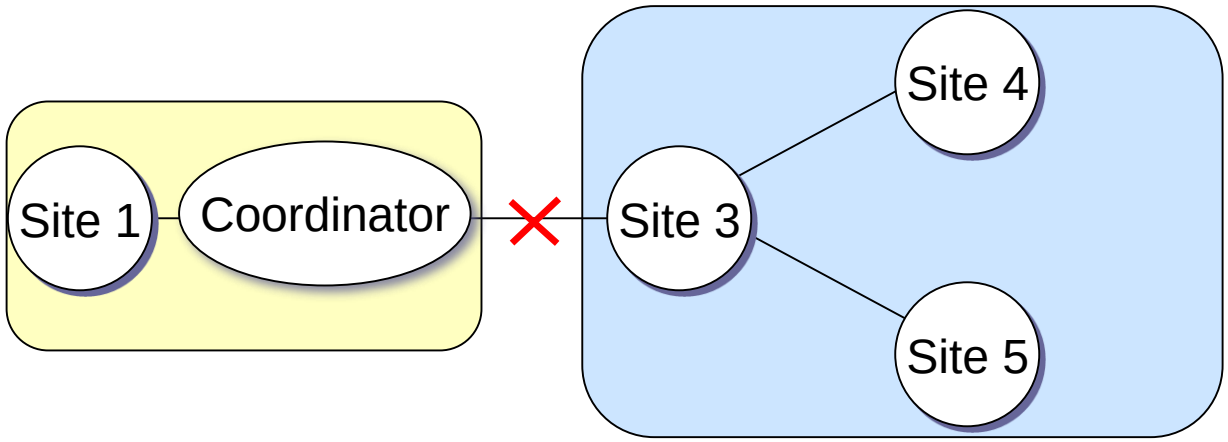
→ **Blocking** – all participants cannot proceed until either the coordinator or dead sites are available

* The coordinator cannot decide the fate T_g after recovery



Handling of Failures in 2PC: Network Partition

- If the coordinator and all its participants remain in one partition, the failure has no effect on the commit protocol
- If the coordinator and its participants belong to several partitions:
  - Sites that are not in the partition containing the coordinator think the coordinator has failed, and execute the protocol to deal with failure of the coordinator
    - No harm results, but sites may still have to wait for decision from coordinator
  - The coordinator and the sites are in the same partition as the coordinator think that the sites in the other partition have failed, and follow the usual commit protocol
    - Again, no harm results



Fast Processing of In-Doubt Transactions in 2PC

- The recovering site must determine the commit-abort status of in-doubt transactions by contacting other sites (by sending messages each other)
  - **In-doubt transactions** are transactions that have a <**ready** *T*>, but neither a <**commit** *T*>, nor an <**abort** *T*> log record
  - Finding the status of in-doubt transactions can be slow (마냥 답변을 기다린다면)
  - Fast processing of in-doubt transactions is crucial for the recovery of distributed transactions
- Recovery algorithms in 2PC 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)
  - For every in-doubt transaction *T*, all the locks noted in the <**ready** *T*, *L*> log record are reacquired
  - After lock reacquisition, processing of in-doubt transaction can resume
  - 즉 다른 site에서 답변이 안오더라도 일단 진행을 하고, write를 완료한 data는 unlock 을 해서 다른 transaction도 일단 진행을 하게 하고
    - 그러다가 commit 이나 abort 답변이 오면 그것이 따라서 행동하고
  - The commit or rollback of in-doubt transactions is performed concurrently with the execution of new transactions

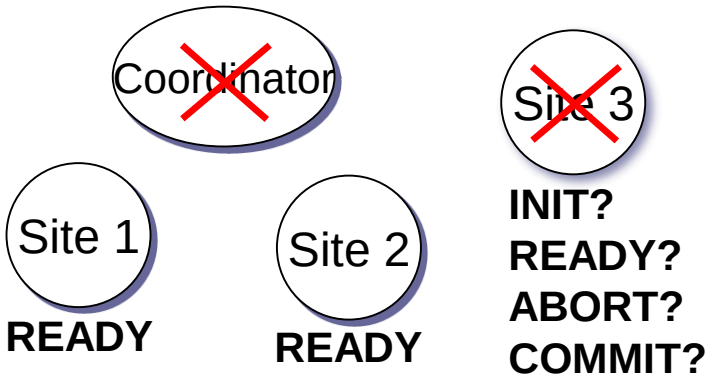
2PC vs. 3PC

- 2PC may cause blocking when coordinator fails

→ 3PC chooses "abort" instead of the blocking process

→ After aborting, the participants can resume the normal activity

When **coordinator dies** and **all active sites are in READY state**



2PC ➔ Participants go to "blocking"

3PC ➔ Participants go to "abort" state by having "precommit" state

- Basic idea of 3PC:

1. By having PRECOMMIT state, there is no single state from which it is possible to make a transition directly to either a COMMIT or an ABORT state
2. By having PRECOMMIT state, there is no state (i.e., blocking state) in which it is not possible to make a final decision, and from which a transition to a COMMIT state can be made

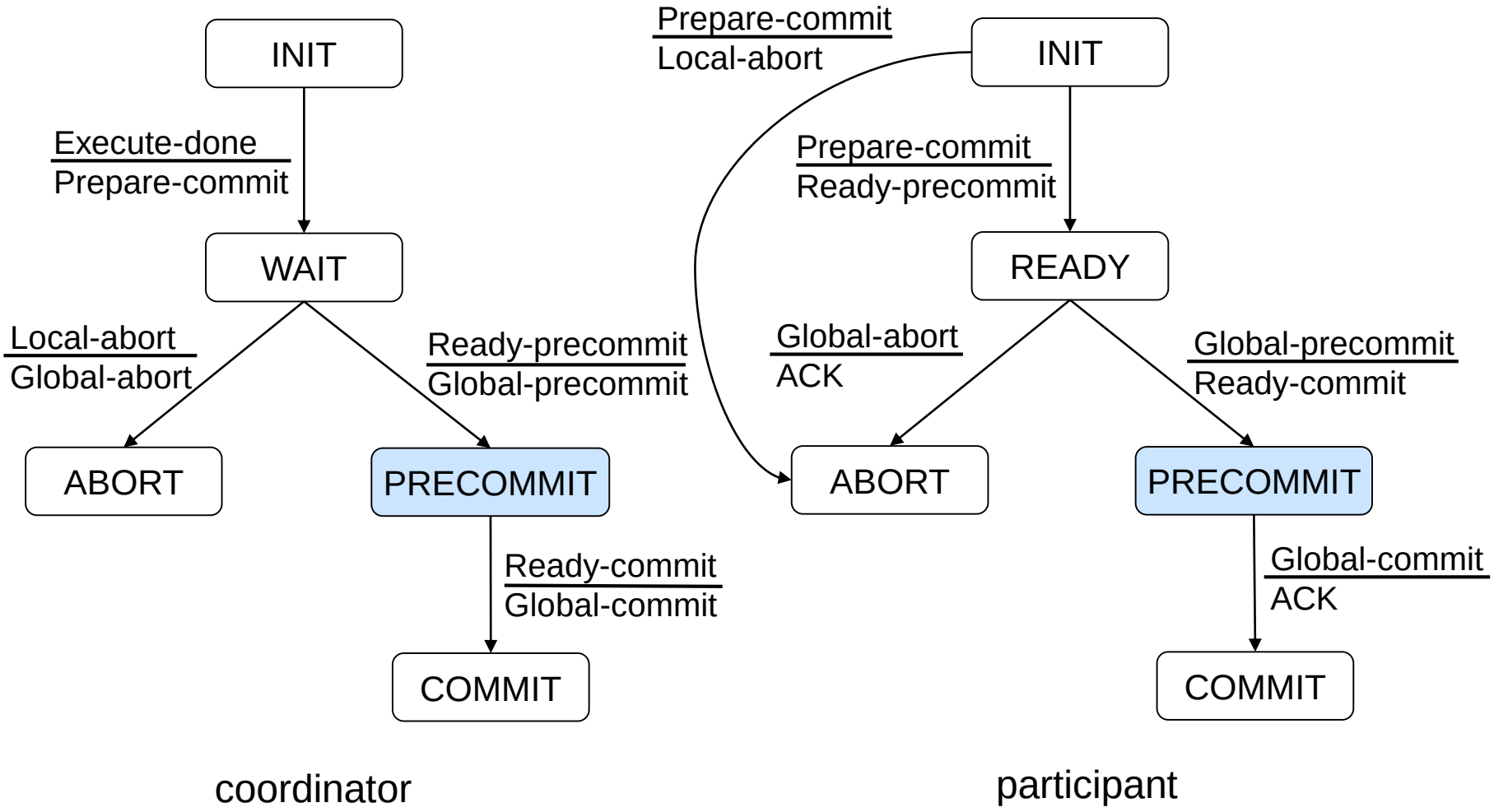
Three Phase Commit Protocol (3PC) [1/2]

- Three phases
  - Phase 1 :
    - Coordinator check if T can commit, participants send their choice to coordinator
  - Phase 2 : Coordinator makes decision
    - If commit, send precommit message to participants
    - If abort, send abort message to participants
  - Phase 3 :
    - If commit, final commit decision is broadcast and everyone commits

Three Phase Commit (3PC) [2/2]

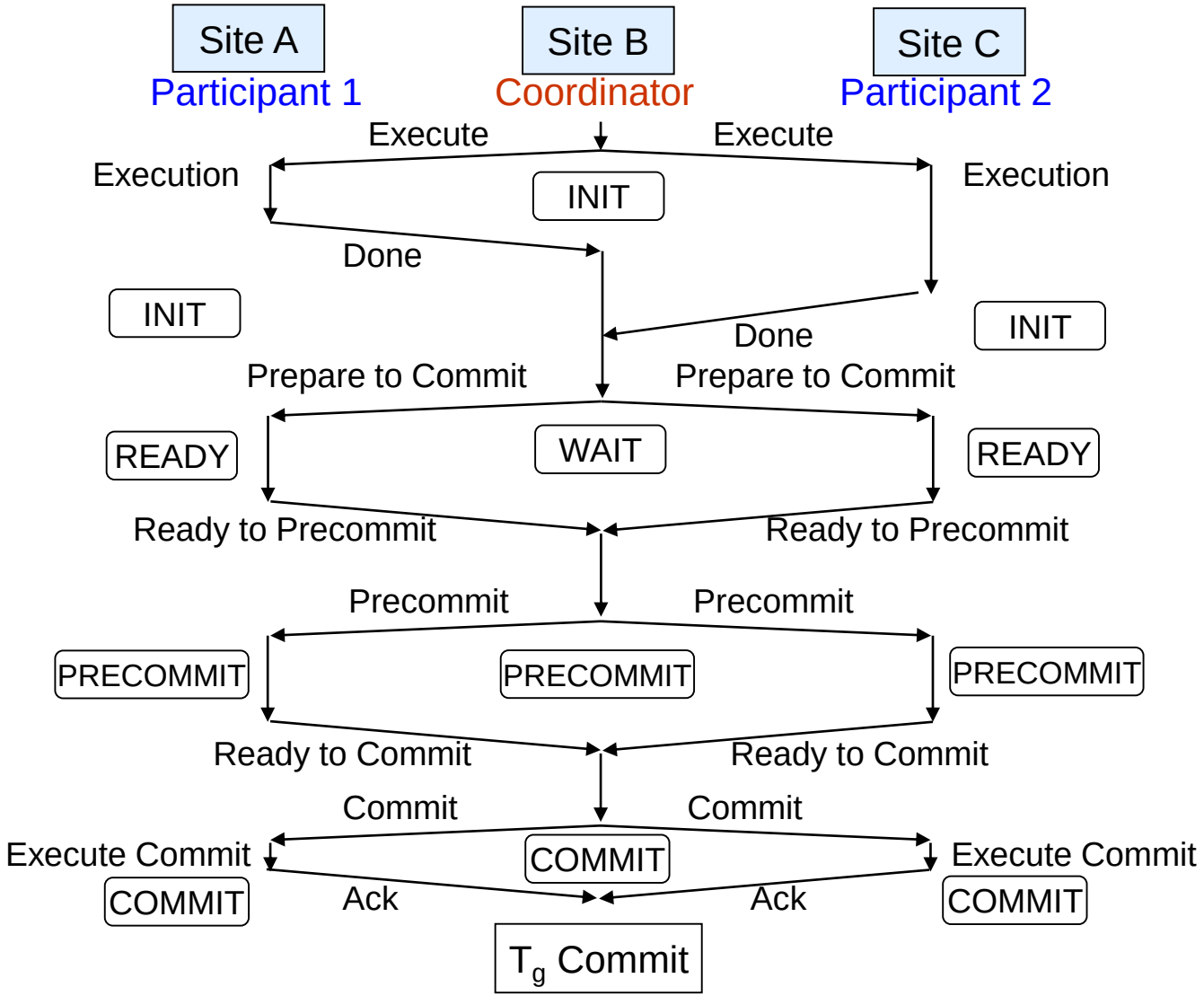
- Assumptions:
  - No network partitioning
  - At any point, at least one site must be up
  - At most K sites (participants as well as coordinator) can fail
- Phase 1: Obtaining Preliminary Decision: Identical to 2PC Phase 1
  - Every site is ready to commit if instructed to do so
- Phase 2 of 2PC is split into 2 phases, Phase 2 and Phase 3 of 3PC
  - In phase 2 coordinator makes a decision as in 2PC (called the **pre-commit decision**) and records it in multiple (at least K) sites
  - In phase 3, coordinator sends commit/abort message to all participating sites
- Under 3PC, knowledge of pre-commit decision can be used to commit despite coordinator failure
  - Avoids the blocking problem in 2PC coordinate failure as long as < K sites fail
- Drawbacks:
  - higher overheads
  - assumptions may not be satisfied in practice

3PC: Finite State Machine



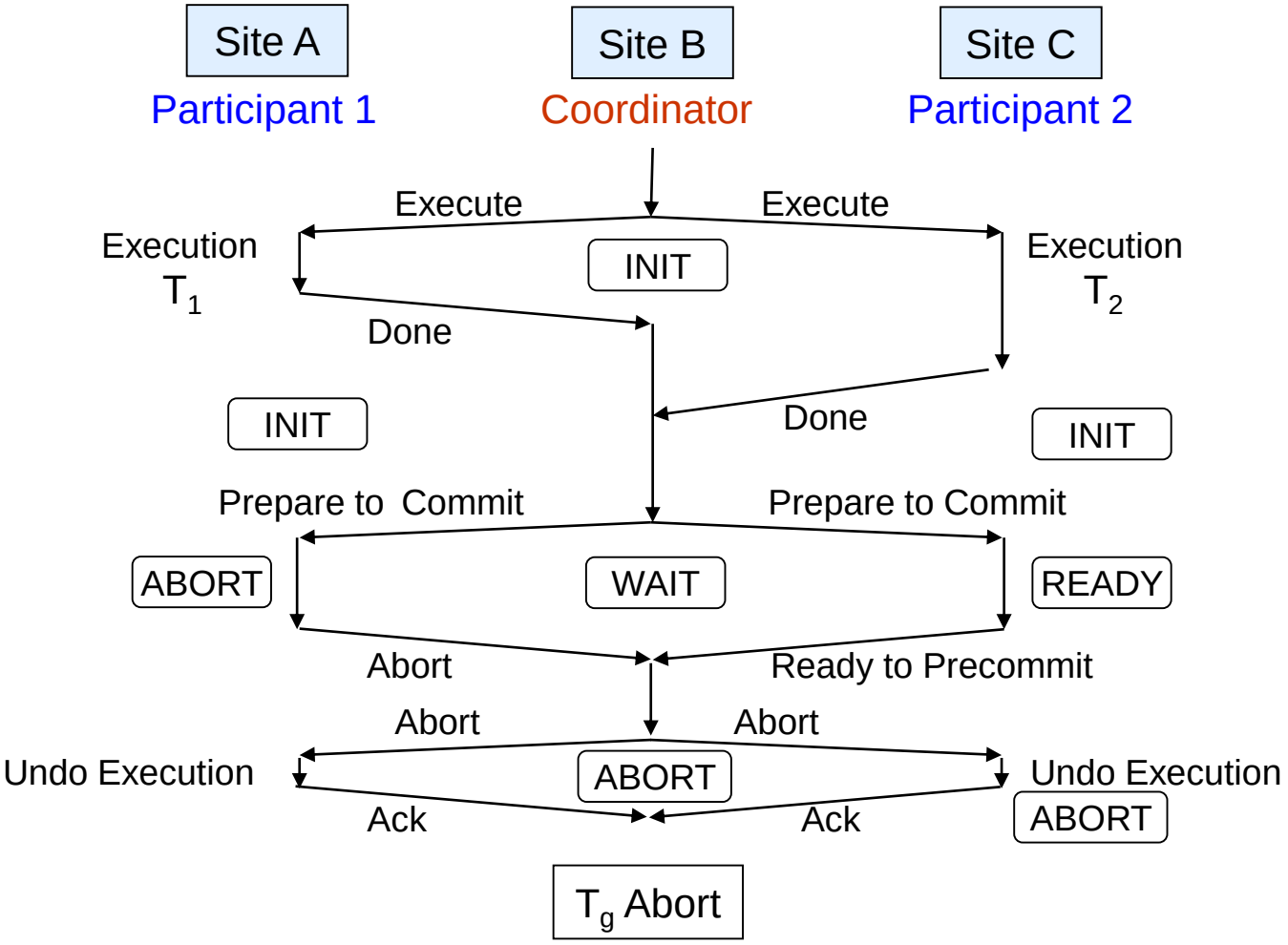
3PC: Commit

Assume the coordinator received a global transaction T_g containing local transaction T_1 and T_2



3PC: Abort

Assume the coordinator received a global transaction T_g containing local transaction T_1 and T_2



Handling of Failures in 3PC: Site Failure

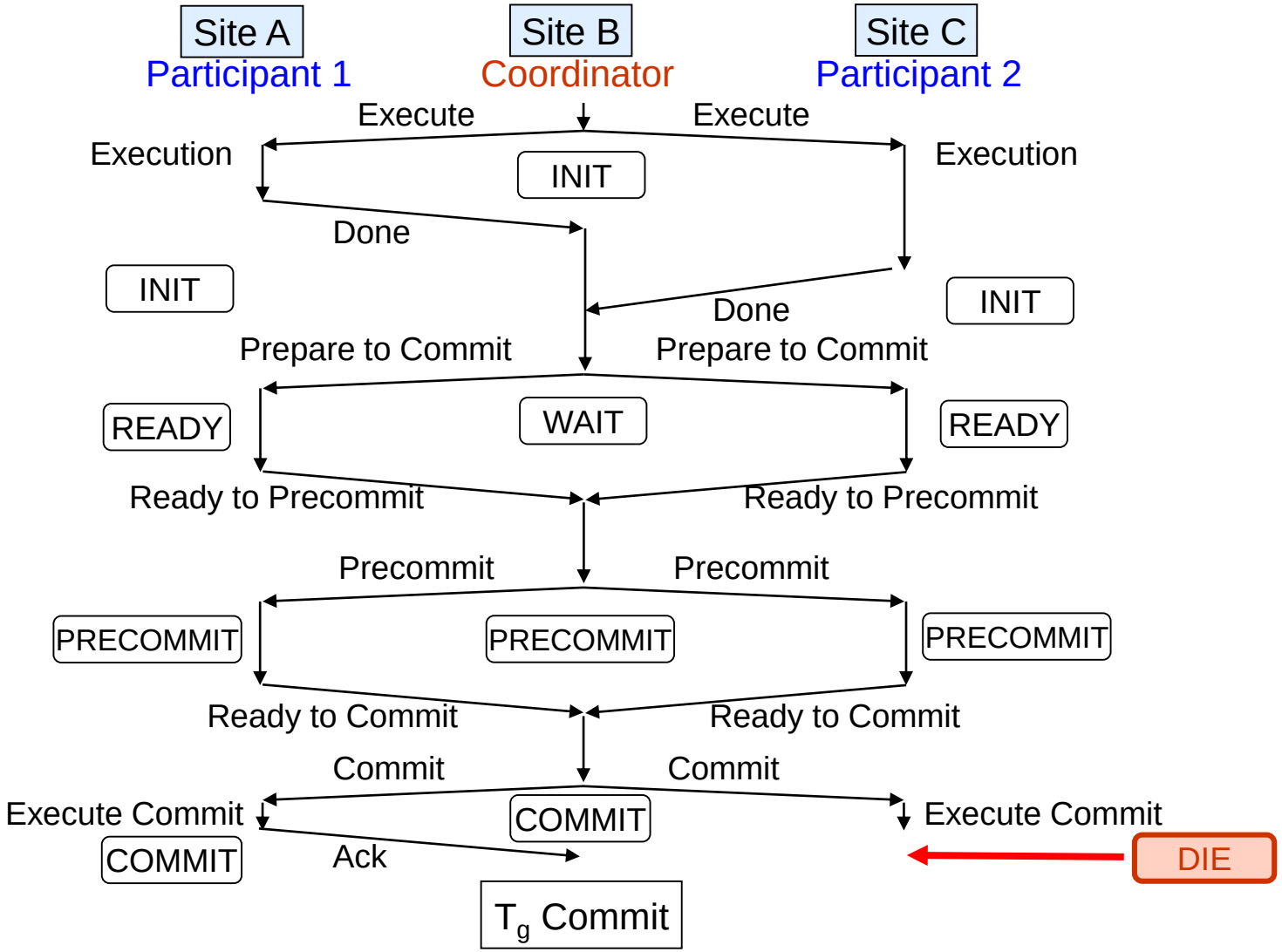
When site S_i recovers, it examines its log to determine the fate of transactions active at the time of the failure

- Log contain <**commit** *T*> record: site executes **redo** (*T*)
- Log contains <**abort** *T*> record: site executes **undo** (*T*)
- Log contains <**ready** *T*> or <**precommit** *T*> record: site must consult C_i to determine the fate of *T*
  - If *T* committed, **redo** (*T*)
  - If *T* aborted, **undo** (*T*)
- The log contains no control records concerning *T* replies that S_k failed before responding to the **prepare** *T* message from C_i
  - since the failure of S_k precludes the sending of such a response C_1 must abort *T*
  - S_k must execute **undo** (*T*)

Handling Site Failure in 3PC [1/4]

CASE 1: Participant 2 dies **during commit execution**

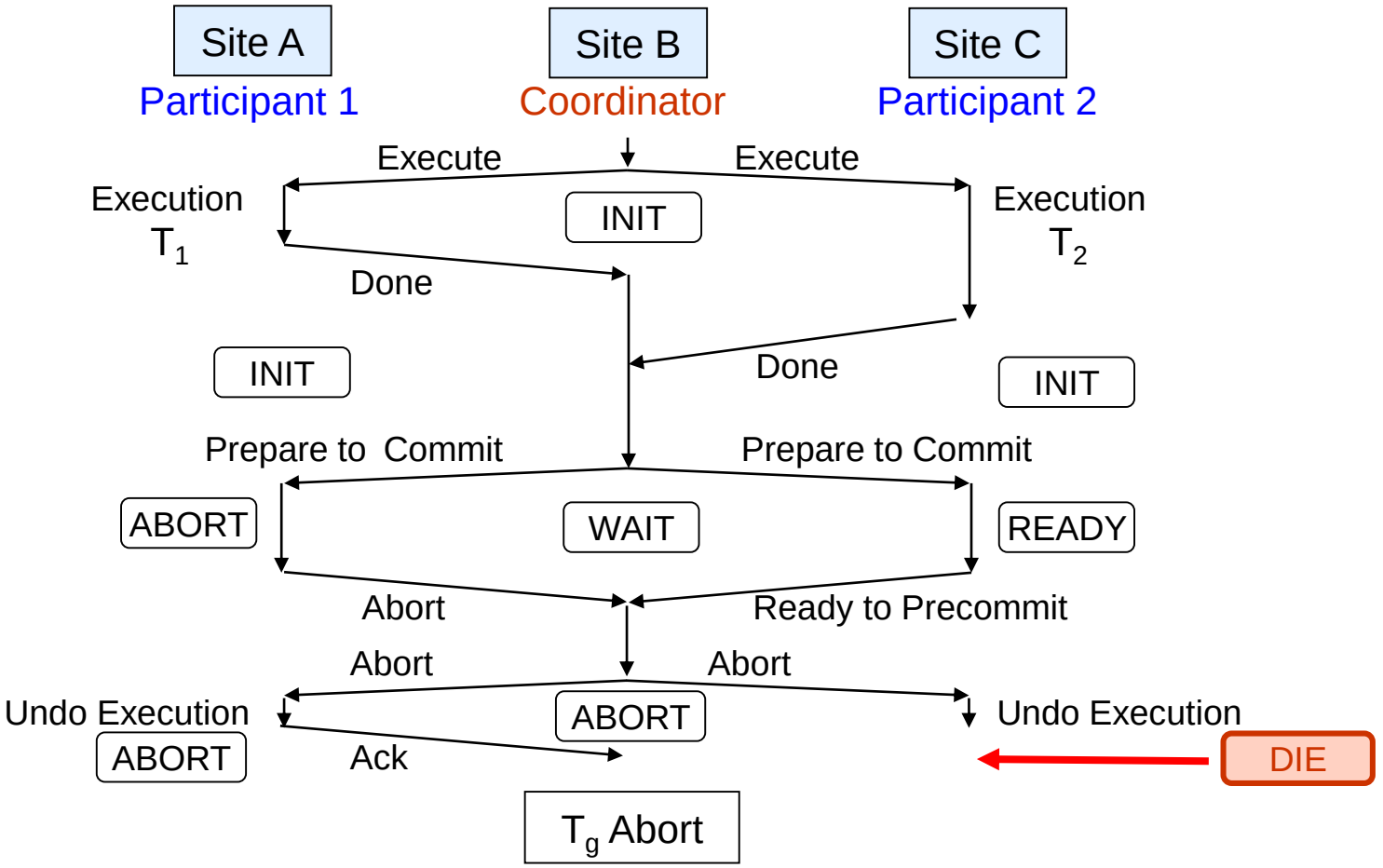
Participant 2 knew the commit before it dies → During recovery, Participant 2 must **redo** T_2



Handling Site Failure in 3PC [2/4]

CASE 2: Participant 2 dies **during undo execution**

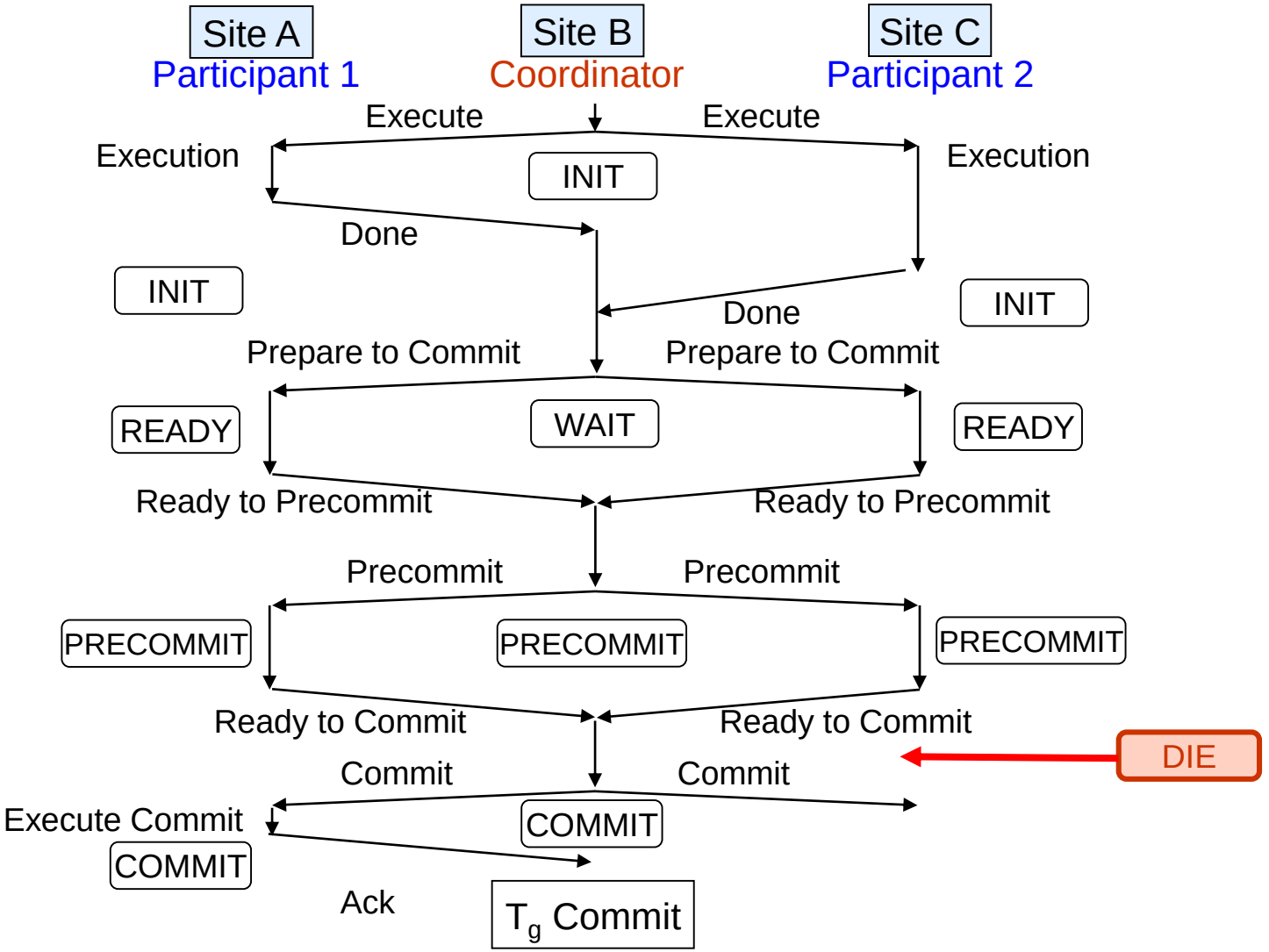
Participant 2 knew the abort before it dies. → During recovery, Participant 2 must **undo** T_2



Handling Site Failure in 3PC [3/4]

CASE 3: Participant 2 dies **during ready or precommit state**

Participant 2 don't know the fate of the transaction → During recovery, Participant 2 must **consult the coordinator**
In this case, redo T_2 (redo T_2 if T_g is commited, undo T_2 if T_g is aborted)

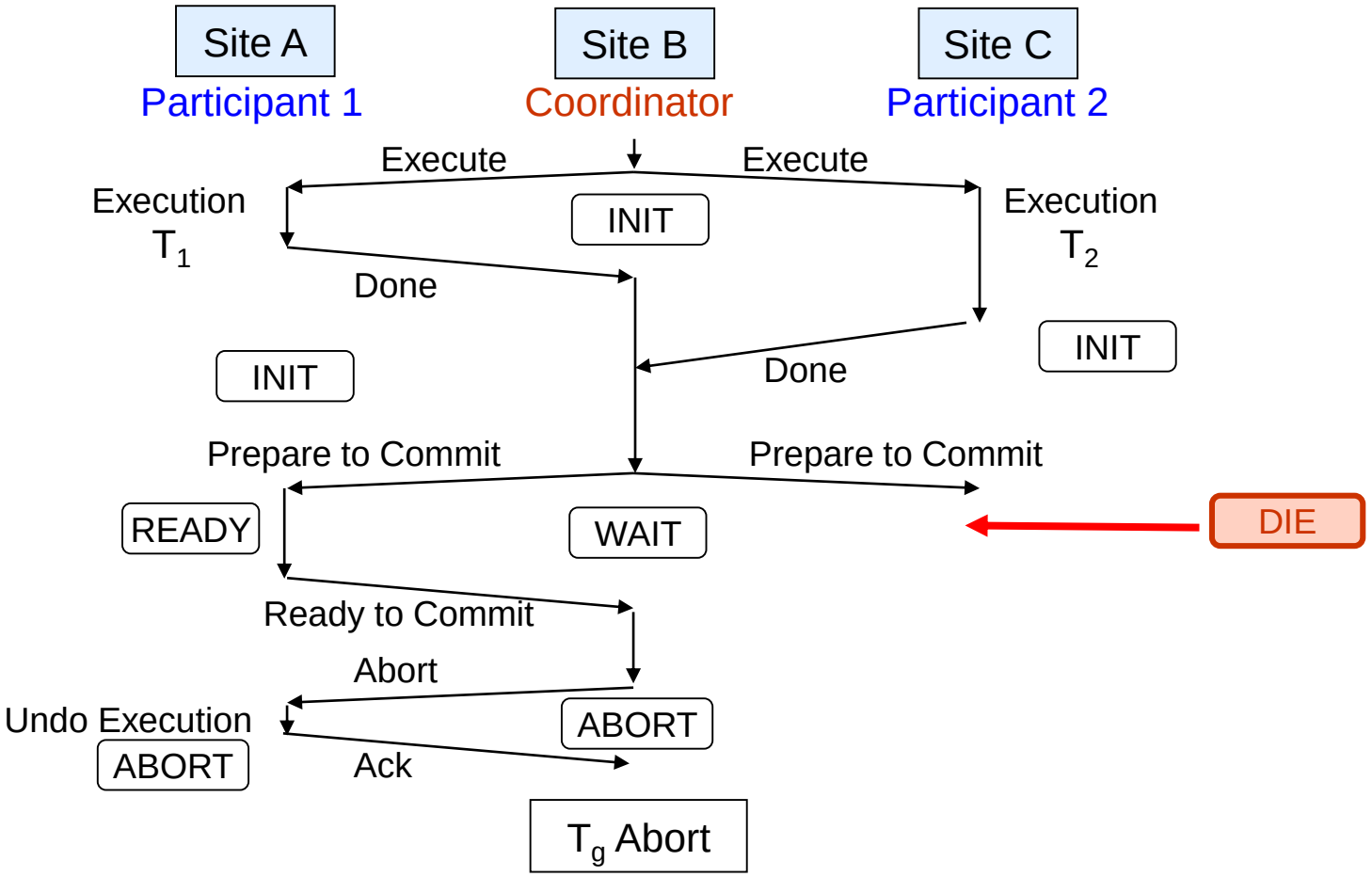


Handling Site Failure in 3PC [4/4]

CASE 4: Participant 2 dies **before it responds to the prepare to commit message**

T_g cannot be commited because Participant 2 didn't send a ready to commit message

→ During recovery, Participant 2 must **undo** T_2



Handling of Failures in 3PC - Coordinator Failure

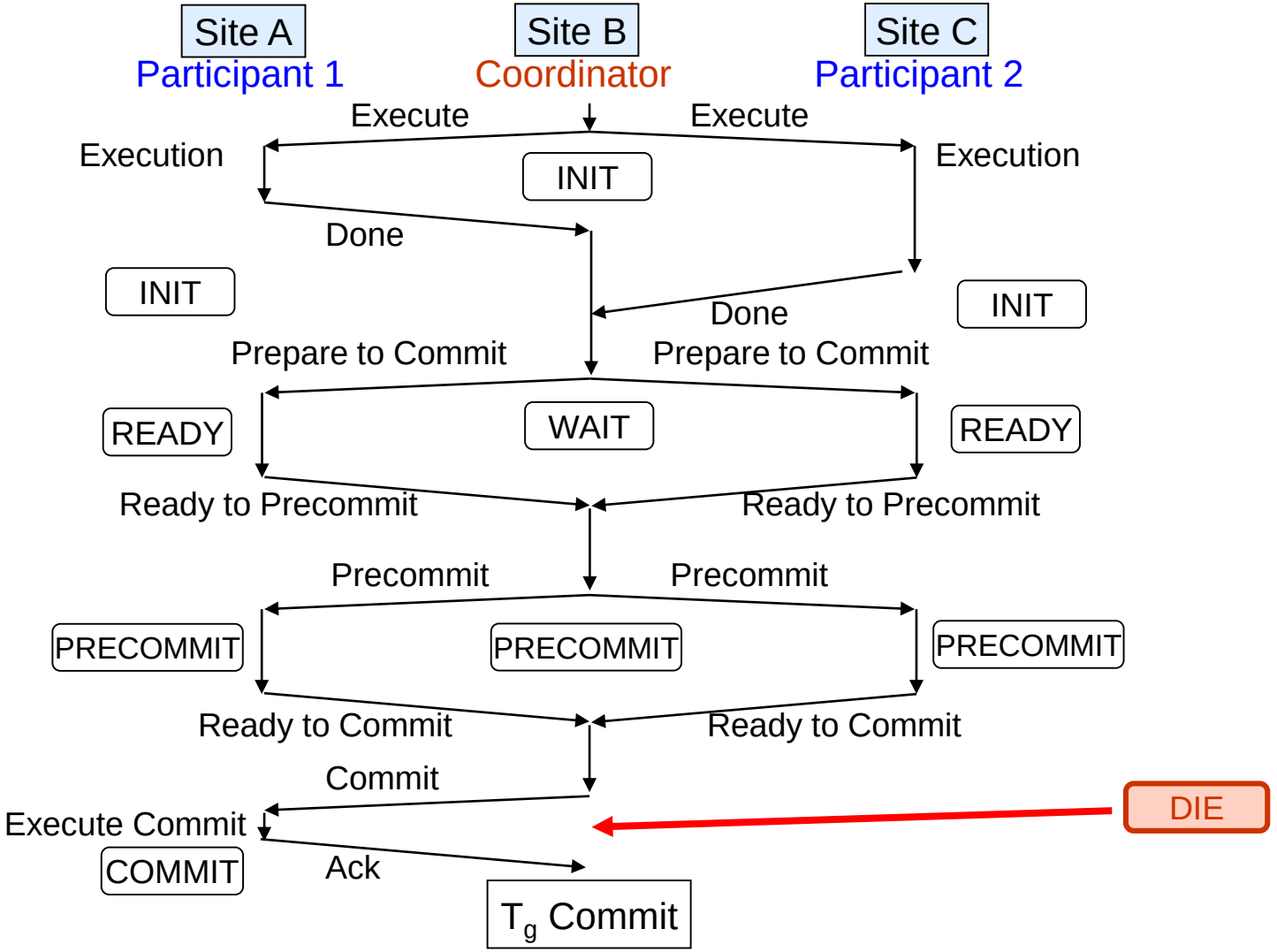
- If coordinator fails while the commit protocol for *T* is executing then participating sites must decide on *T*'s fate:
  1. If an active site contains a <**commit** *T*> record in its log, then *T* must be committed
  2. If an active site contains a <**abort** *T*> record in its log, then *T* must be aborted
  3. If some active participating site does not contain a <**ready** *T*> record in its log (i.e., if some participant is still in **INIT** state), then the failed coordinator C_i cannot have decided to precommit *T*
     1. Can therefore abort *T*.
  4. If none of the above cases holds, then all active sites must have a <**ready** *T*> record or a <**precommit** *T*> record in their logs
     1. If some active sites have a <**ready** *T*> record and some active sites have a <**precommit** *T*> record, then *T* must be committed since all must have voted commit
     2. If all active sites have a <**ready** *T*> record, then *T* must be aborted since some dead parts may have voted abort, or coordinator C_i may not have received all votes before crash
        1. If all active sites voted commit before crash, it cannot do any harm to still abort the transaction *T*

Handling Coordinator Failure in 3PC [1/5]

CASE 1: Coordinator dies **while it sends commit messages**
(i.e., **an active site is in COMMIT state**)

When coordinator fails, Participant 2 can learn of the commit from Participant 1

→ Participant 2 must **commit** T_2

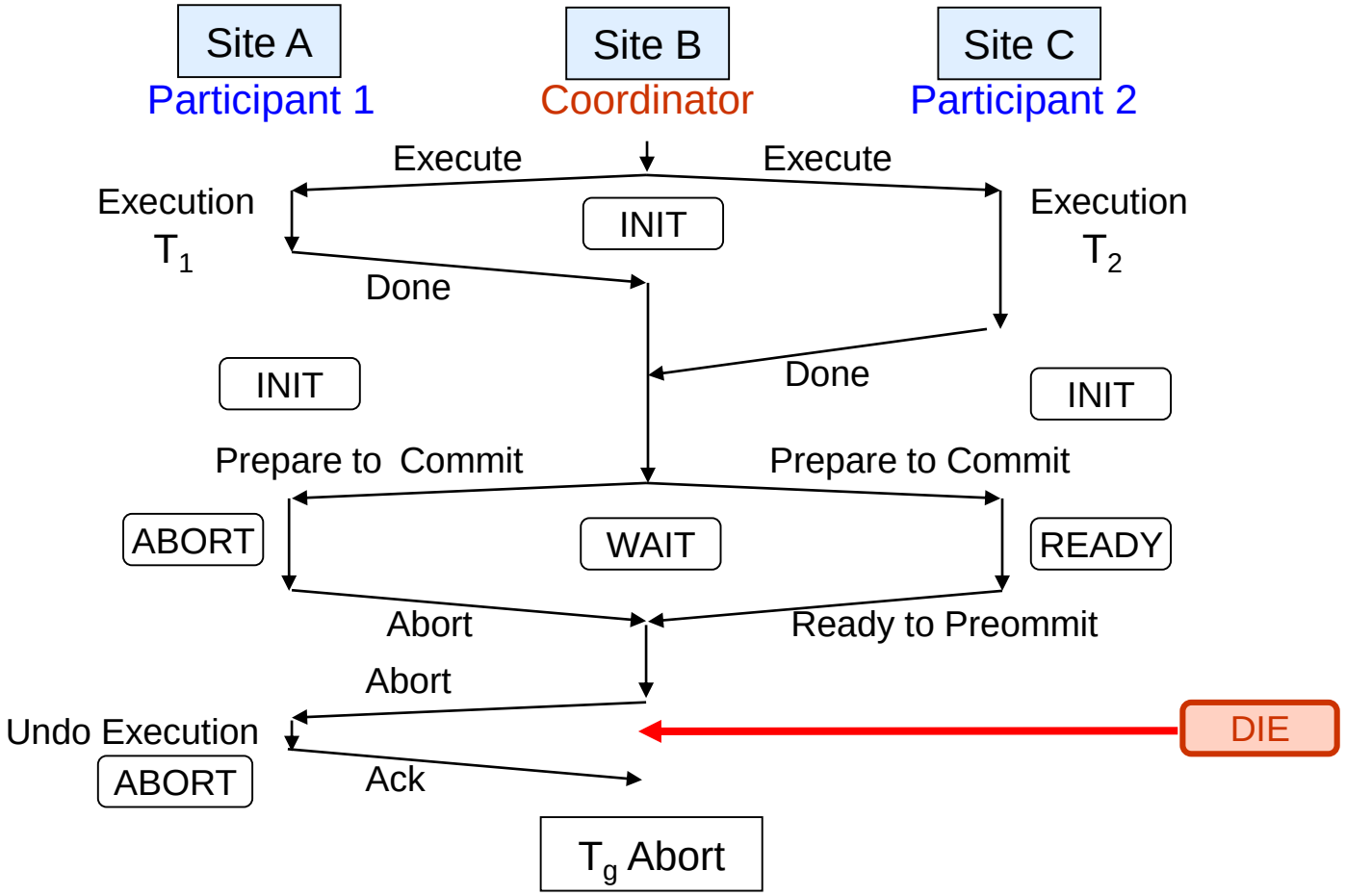


Handling Coordinator Failure in 3PC [2/5]

CASE 2: Coordinator dies **while it sends abort messages**
(i.e., **an active site is in ABORT state**)

When coordinator fails, Participant 2 can learn of the abort from Participant 1

→ Participant 2 must **abort** T_2

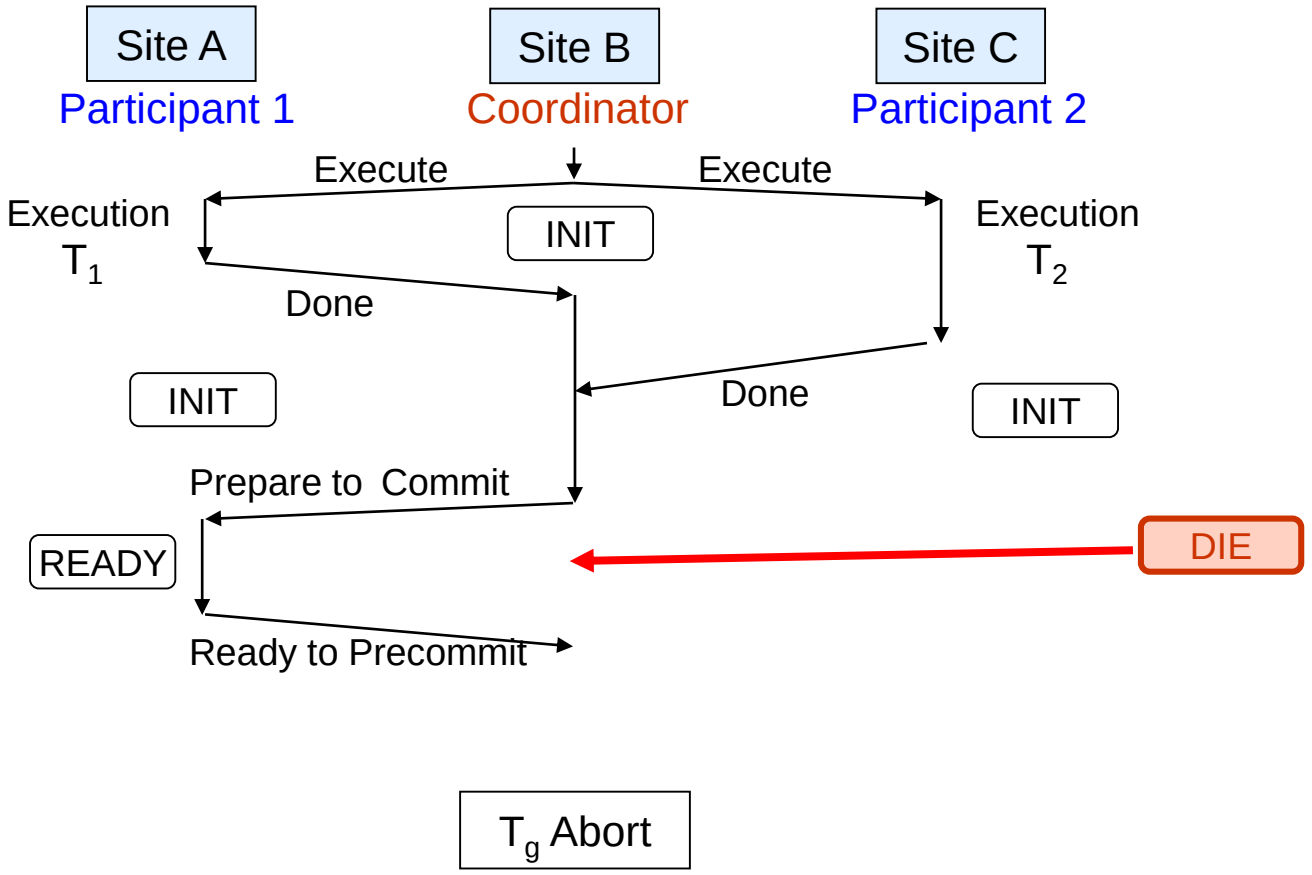


Handling Coordinator Failure in 3PC [3/5]

CASE 3: Coordinator dies **while it sends prepare to commit messages** (i.e., **some active sites are in INIT state**)

T_g cannot be commited because Participant 2 didn't receive a prepare to commit message

→ All participants must **abort** the transactions

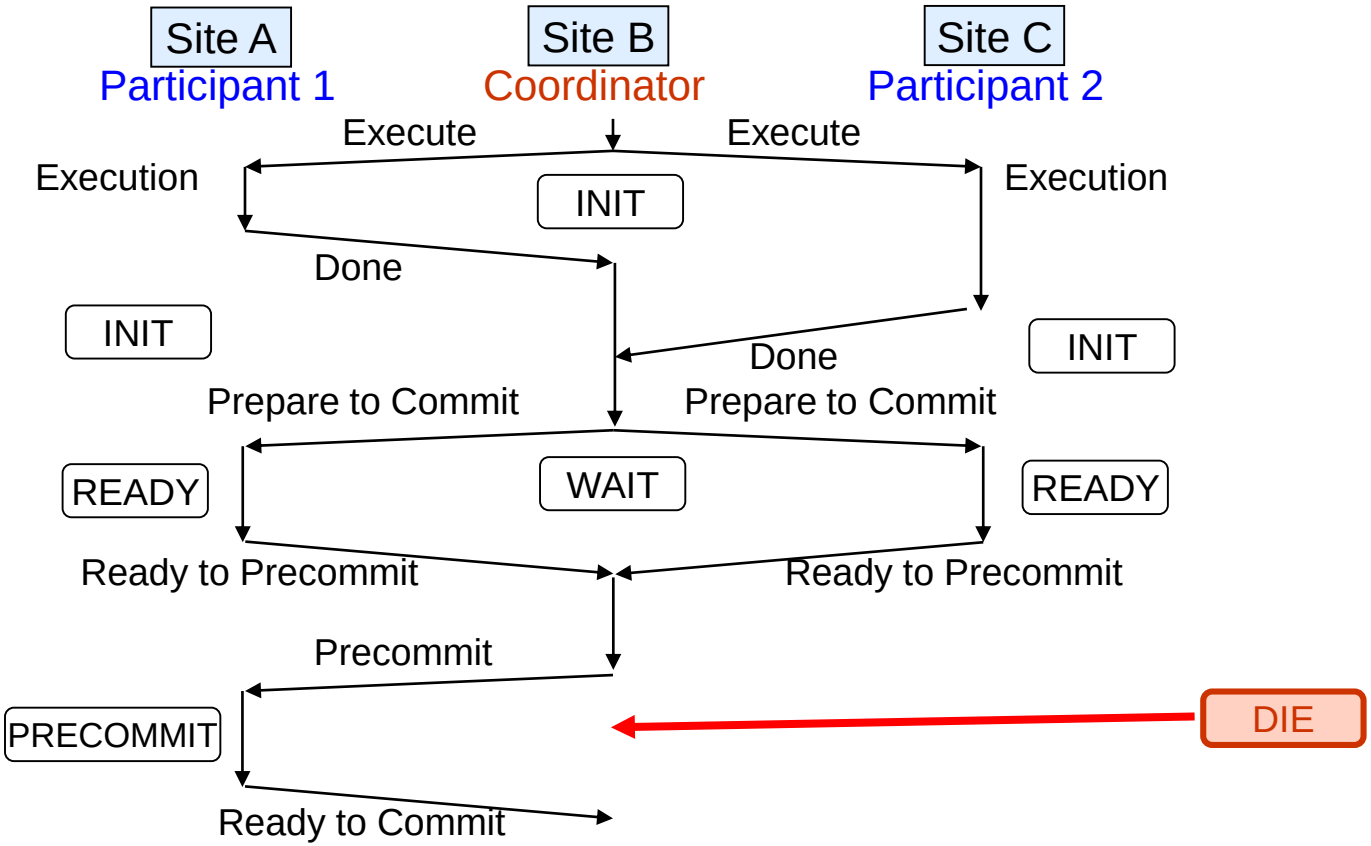


Handling Coordinator Failure in 3PC [4/5]

CASE 4a: Coordinator dies **while it sends precommit messages**
(i.e., **some active sites are in PRECOMMIT state**)

All site must have voted commit if some active sites are in precommit state

→ All participant must **commit** the transactions

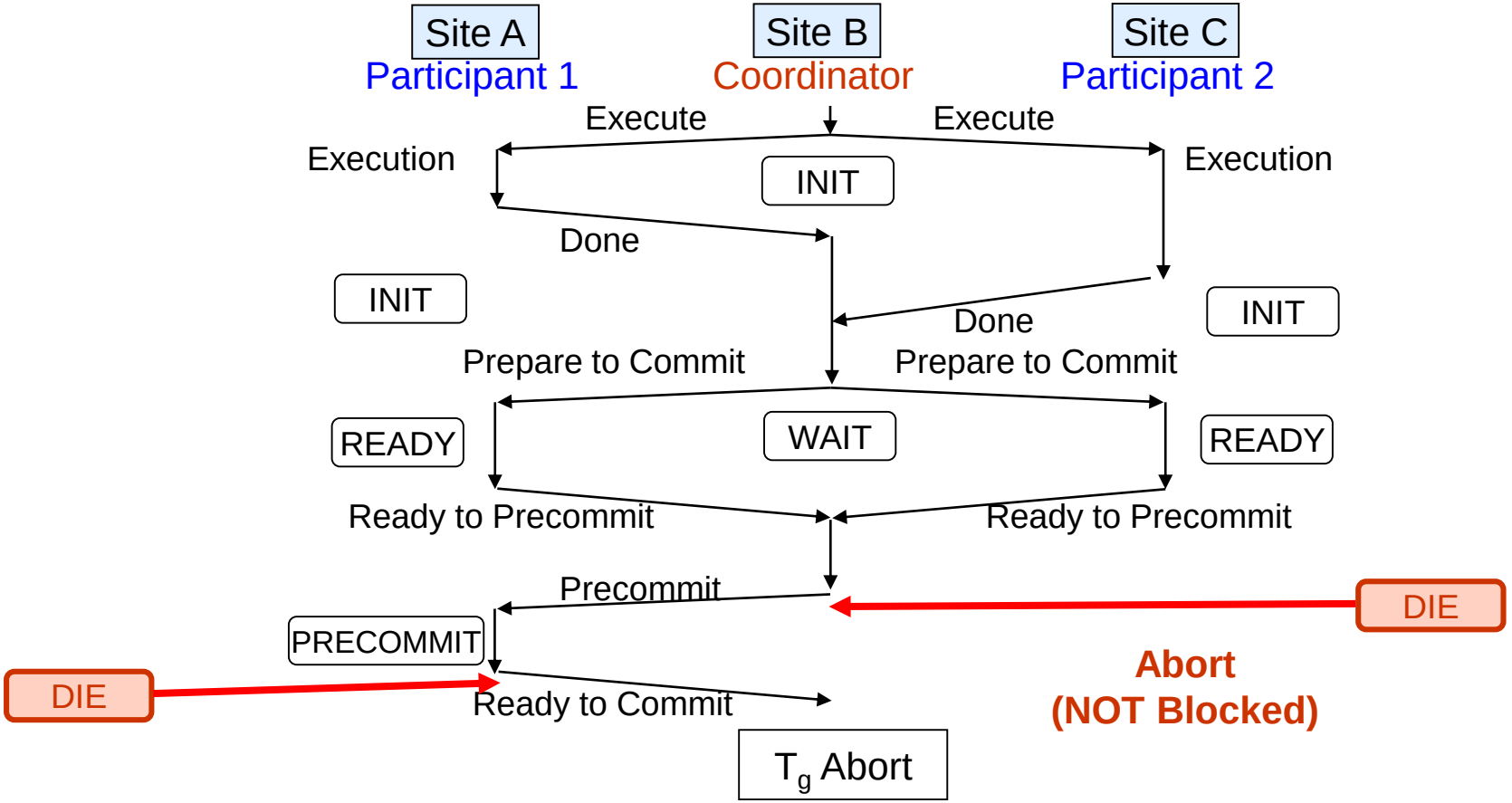


T_g Commit

Handling Coordinator Failure in 3PC [5/5]

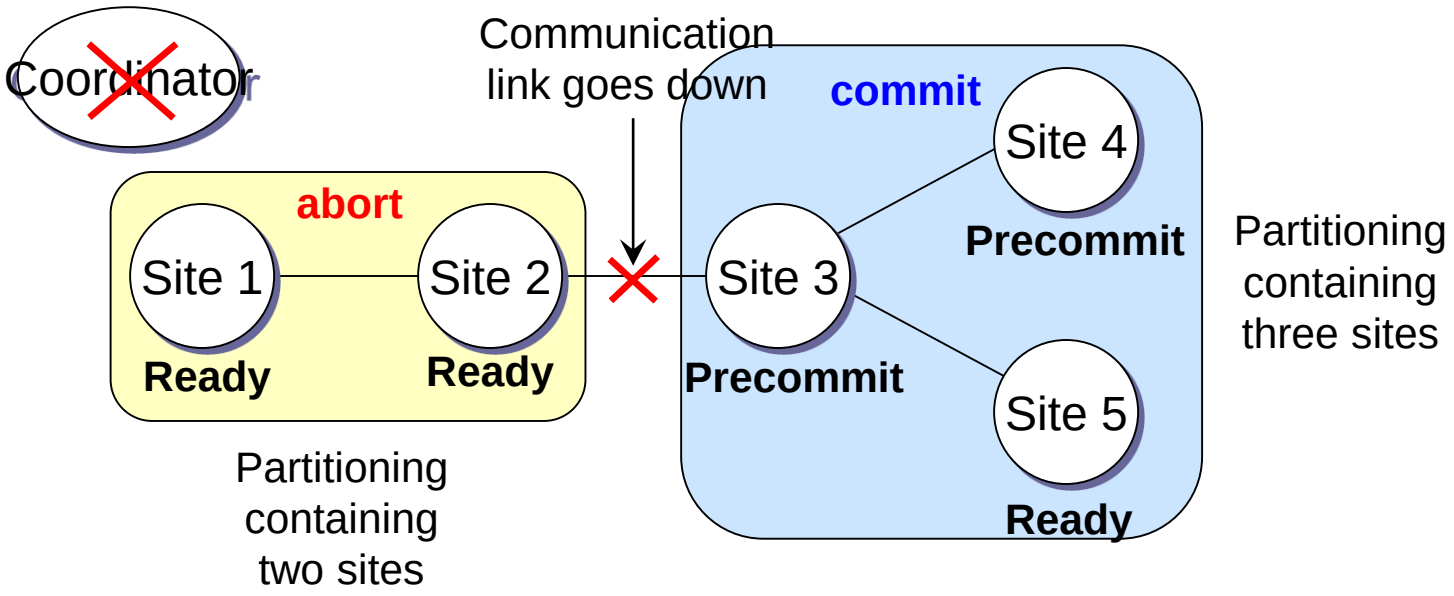
CASE 4b: Coordinator dies and **all active sites are in READY state**
(some dead parts may be in INIT, READY, ABORT, or **PRECOMMIT** state,
or coordinator may not have received all votes before crash.)

We cannot sure whether all participants voted commit → All participant must **abort** the transactions



Pros and Cons of 3PC

- **No blocking problem**
- **Network partition problem – All precommit sites may reside in one section**



- Adds another set of messages to be exchanged between the coordinator and participants
- Not widely used

Alternative Models of Transaction Processing [1/2

- Distributed transaction model with 2PC (a single global transaction spanning crossing an organizational boundary) is inappropriate for many applications
  - Global transaction with local transactions
  - No organization would like to permit an externally initiated transaction to block local transactions for an indeterminate period
  - Synchronous coordination of multi transactions may cause blocking

- Alternative models carry out transactions by sending persistent messages
  - Messaging is asynchronous and free from blocking local transactions
  - Code to handle messages must be carefully designed to ensure atomicity and durability properties for updates
    - Isolation cannot be guaranteed, in that intermediate stages are visible, but code must ensure no inconsistent states result due to concurrency
  - Persistent messaging systems are systems that provide transactional properties to messages
    - Will discuss implementation techniques later

Alternative Models of Transaction Processing [2/2]

- Motivating example: Fund transfer between two banks
  - 2PC would have the potential to block updates on the accounts involved in fund transfer (한 개의 global transaction과 2개의 local transaction으로 구성)
  - Alternative solution based on messaging:
    - (Bank A) Debit money from source account and send a message to other site
    - (Bank B) Site receives message and credits destination account
  - Messaging has long been used for distributed transactions (even before computers were invented!)
- Atomicity issue in messaging
  - Messages are guaranteed to be delivered exactly once regardless of failures if the transaction sending the message commits
    - 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
  - Messages are guaranteed not to be delivered if the transaction sending the message aborts

Persistent Messaging: Error Conditions

- Code to handle messages has to take care of variety of failure situations (even assuming guaranteed message delivery)
  - 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
- User code executing transaction processing using 2PC does not have to deal with such failures
  - 2PC의 recovery algorithm이 모든 failures를 담당하므로
- There are many situations where extra effort of error handling is worth the benefit of absence of blocking local transactions
  - E.g. pretty much all transactions across organizations

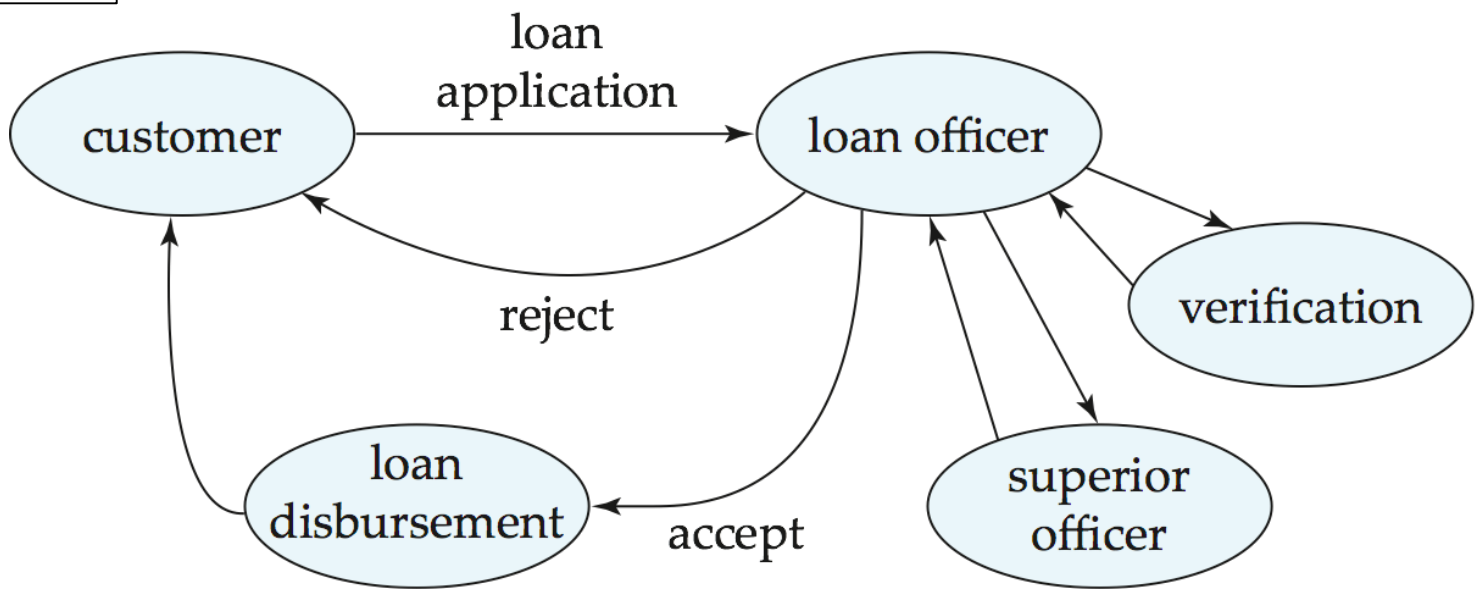
Persistent Messaging and Workflows

- **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
  - We study workflows in Chapter 25
  - Persistent messaging forms the underlying infrastructure for workflows in a distributed environment

- Commercial Workflow Management Systems
  - MicroSoft BizTalk Server
  - IBM WebSphere Business Integration Server Foundation
  - BEA WebLogic Process Edition

Loan Processing Workflow

Fig 26.04



- In the past, workflows were handled by creating and forwarding paper forms
- Computerized workflows aim to automate many of the tasks
- But the humans still play role e.g., in approving loans
- The above workflow consists of several transactions and human actions

Persistent Messaging: Implementation [1/2]

- Sending site protocol
- Sending transaction writes message to a special relation *messages-to-send*
  - The message is also given a unique identifier
  - Writing to this relation is treated as any other update, and is undone if the transaction aborts
  - The message remains locked until the sending transaction commits
- A message delivery process monitors the *messages-to-send* relation
  - When a new message is found, the message is sent to its destination
  - When an acknowledgment is received from a destination, the message is deleted from *messages-to-send*
  - If no acknowledgment is received after a timeout period, the message is resent
    - This is repeated until the message gets deleted on receipt of acknowledgement, or the system decides the message is undeliverable

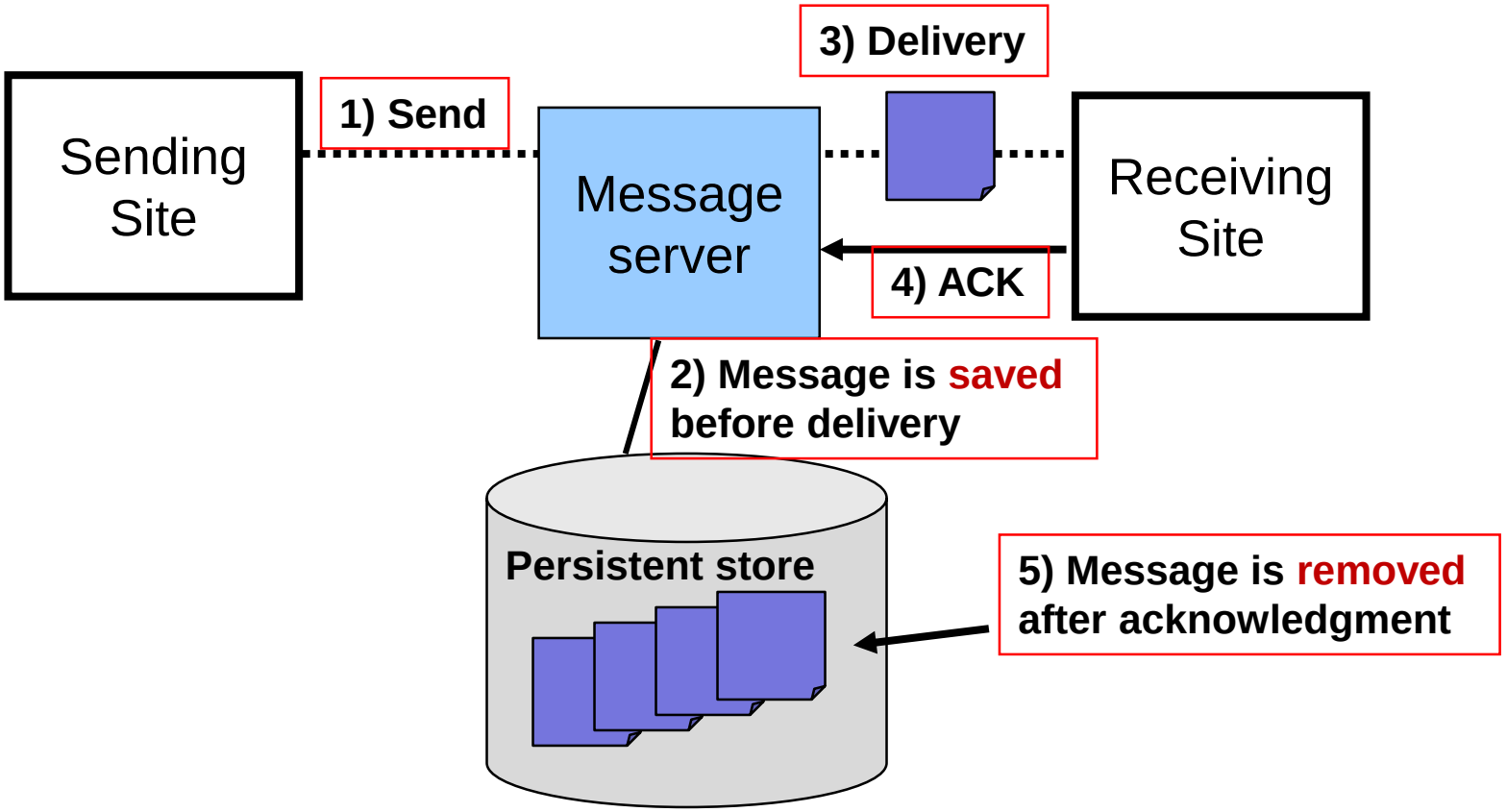
Persistent Messaging: Implementation [2/2]

- Receiving site protocol
  - When a message is received
  1. It is written to a *received-messages* relation
     - if it is not already present (the message id is used for this check).
  2. The transaction performing the write is committed
  3. An acknowledgement is then sent to the sending site

- There may be very long delays in message delivery coupled with repeated messages
  - Could result in processing of duplicate messages if we are not careful
  - Option 1: messages are never deleted from *received-messages*
  - Option 2: messages are given timestamps
    - Messages older than some cut-off are deleted from *received-messages*
    - Received messages are rejected if older than the cut-off

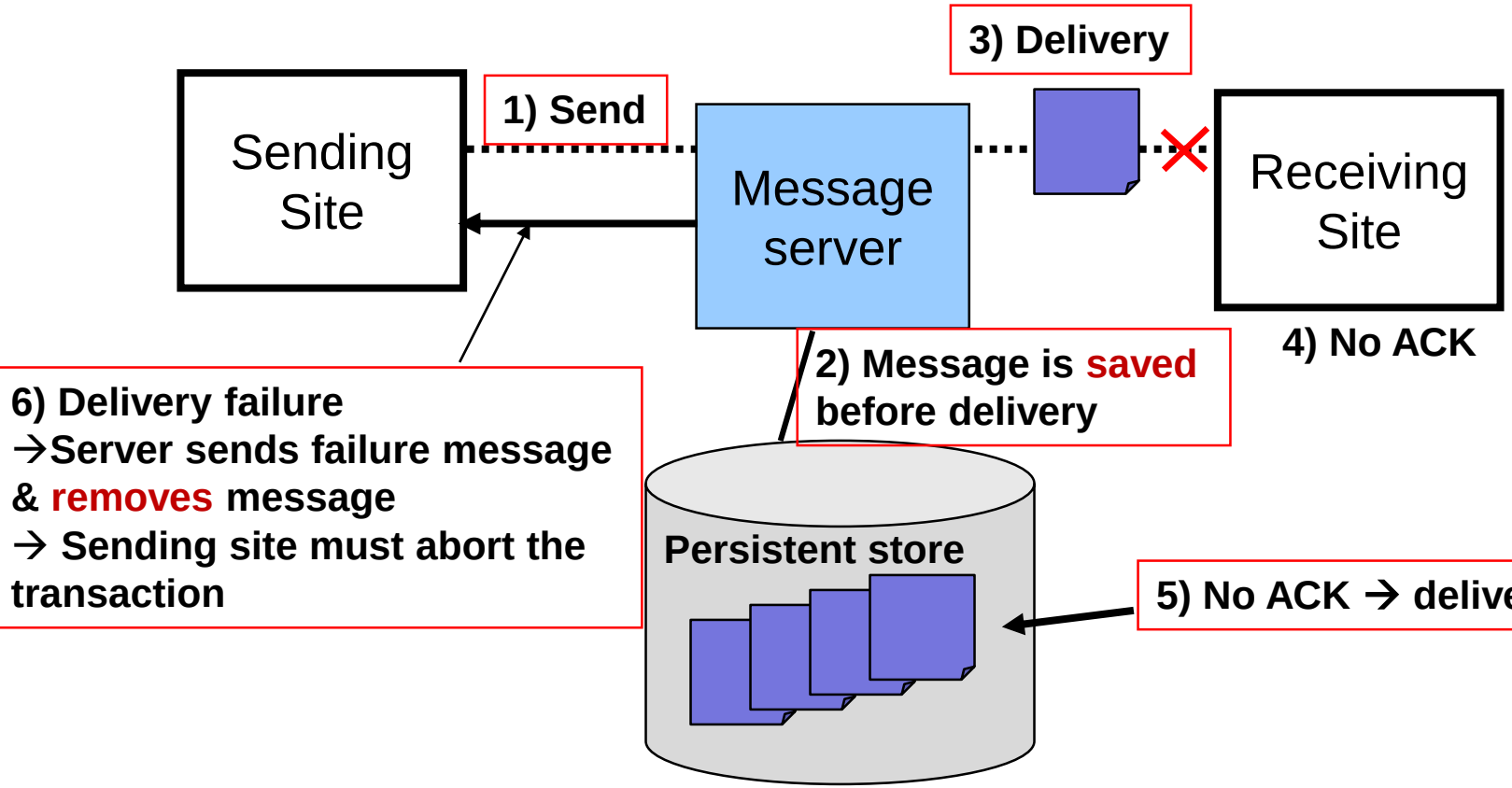
Persistent Messaging Structure

Guarantee successful message delivery for enhancing the robustness of distributed transaction processing instead of “coordinator”



Persistent Messaging – Error Handling [1/2]

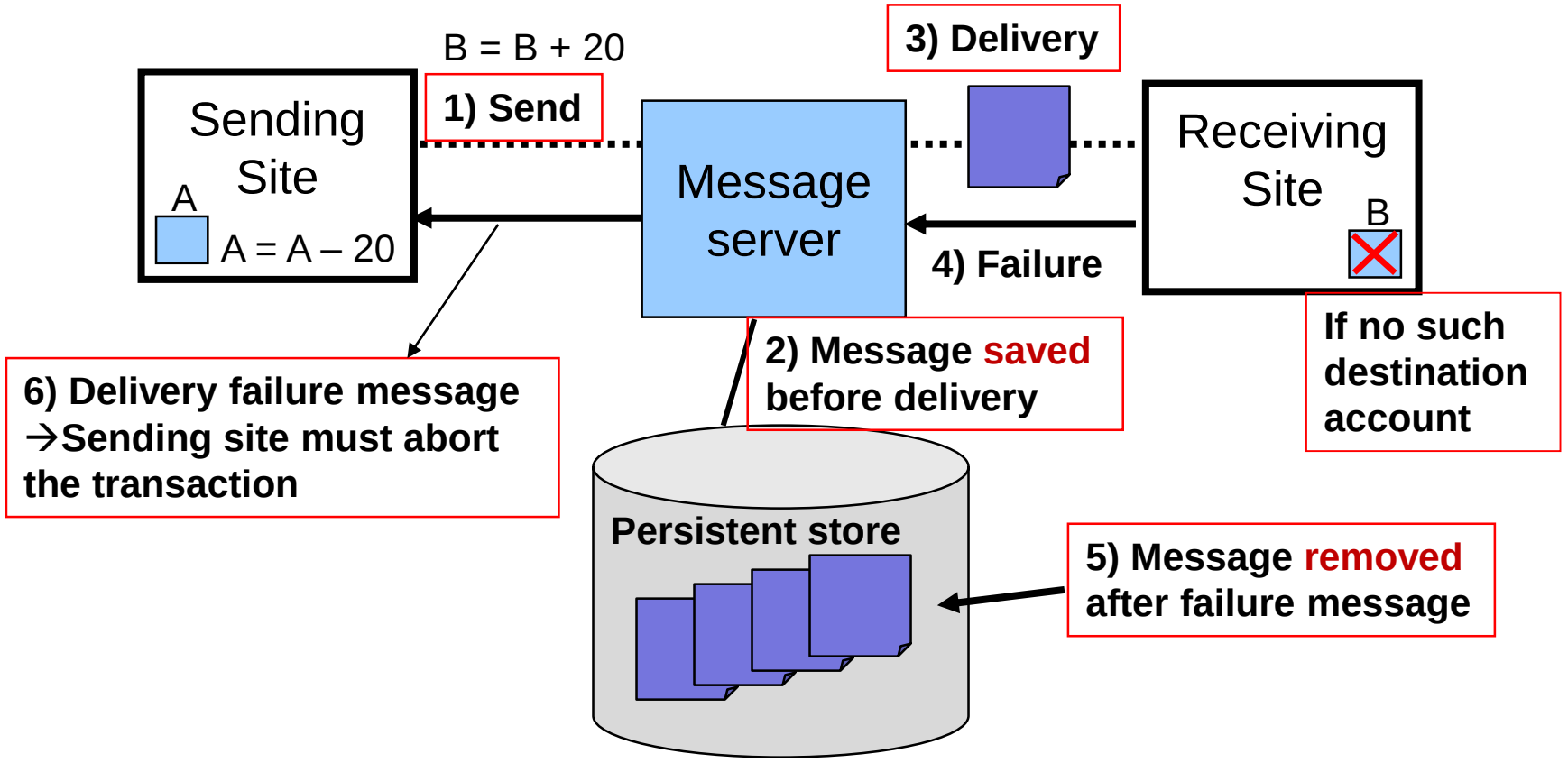
Case 1: Message Failure



Persistent Messaging – Error Handling [2/2]

Example: Fund transfer between two banks (A = A – 20, B = B + 20)

Case 2: If the account B does not exist in the destination site, failure message must be sent back to source site having the account A



Chapter 19: Distributed Databases

- 19.1 Heterogeneous and Homogeneous Databases
- 19.2 Distributed Data Storage
- 19.3 Distributed Transactions
- 19.4 Commit Protocols
- 19.5 Concurrency Control in Distributed Databases
- 19.6 Availability
- 19.7 Distributed Query Processing
- 19.8 Heterogeneous Distributed Databases
- 19.9 Cloud-Based Databases
- 19.10 Directory Systems

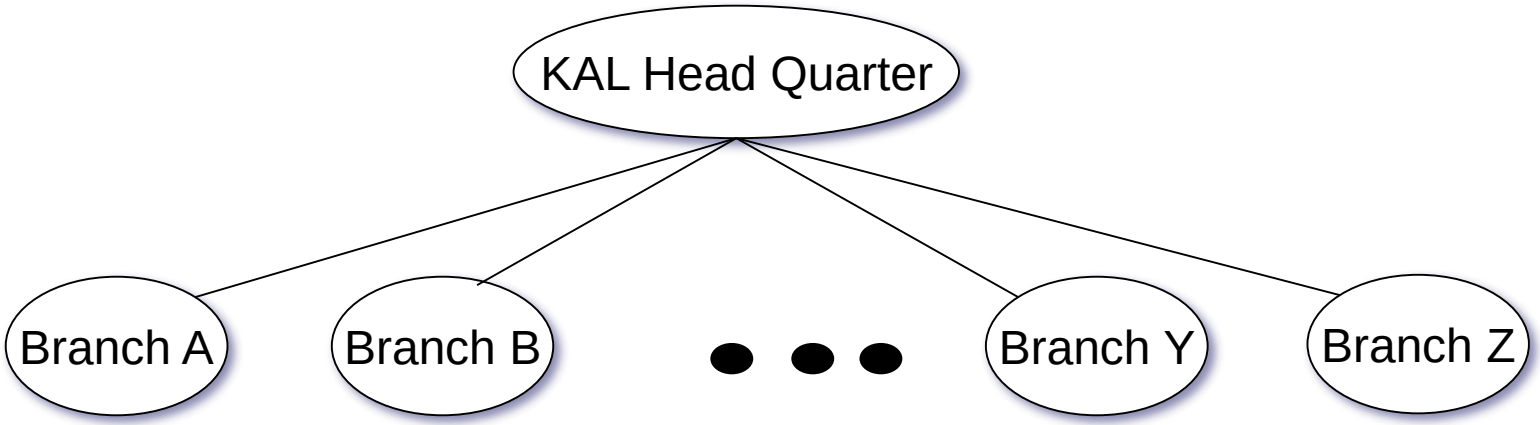
Concurrency Control in Distributed Databases

- Modify earlier concurrency control schemes for use in distributed environment
- We assume that each site participates in the execution of a commit protocol to ensure global transaction atomicity
  - For example, Distributed transaction model with 2PC
- We assume all replicas of any item are updated
  - Will see how to relax this in case of site failures later

- Distributed Locking Protocol
  - Single-Lock-Manager Approach
  - Distributed-Lock-Manager Approach
    - Primary Copy
    - Majority Protocol
    - Bias Protocol
    - Quorum Consensus Protocol
- Distributed Time Stamping Approach
- Replication Approach

Distributed Databases CC 환경

- Flight Ticketing
  - KAL headquarter is in Seoul, Korea
  - 26 KAL branch offices in the world have its own copy of flight seat database
  - Branch offices are selling the flight tickets all over the world
  - 20 copies of flight seat databases should be identical all the time
  - Flight ticket transactions are initiated from one of branch offices
  - Several flight ticket transactions may compete for a same seat
  - Seoul-LA ticket 을 파는 transaction은 branch 1곳에서 시작했지만 27개 flight seat database를 전부 update하는 1개의 global transaction과 26개의 local transaction들로 구성된 distributed transaction으로 볼수 있다

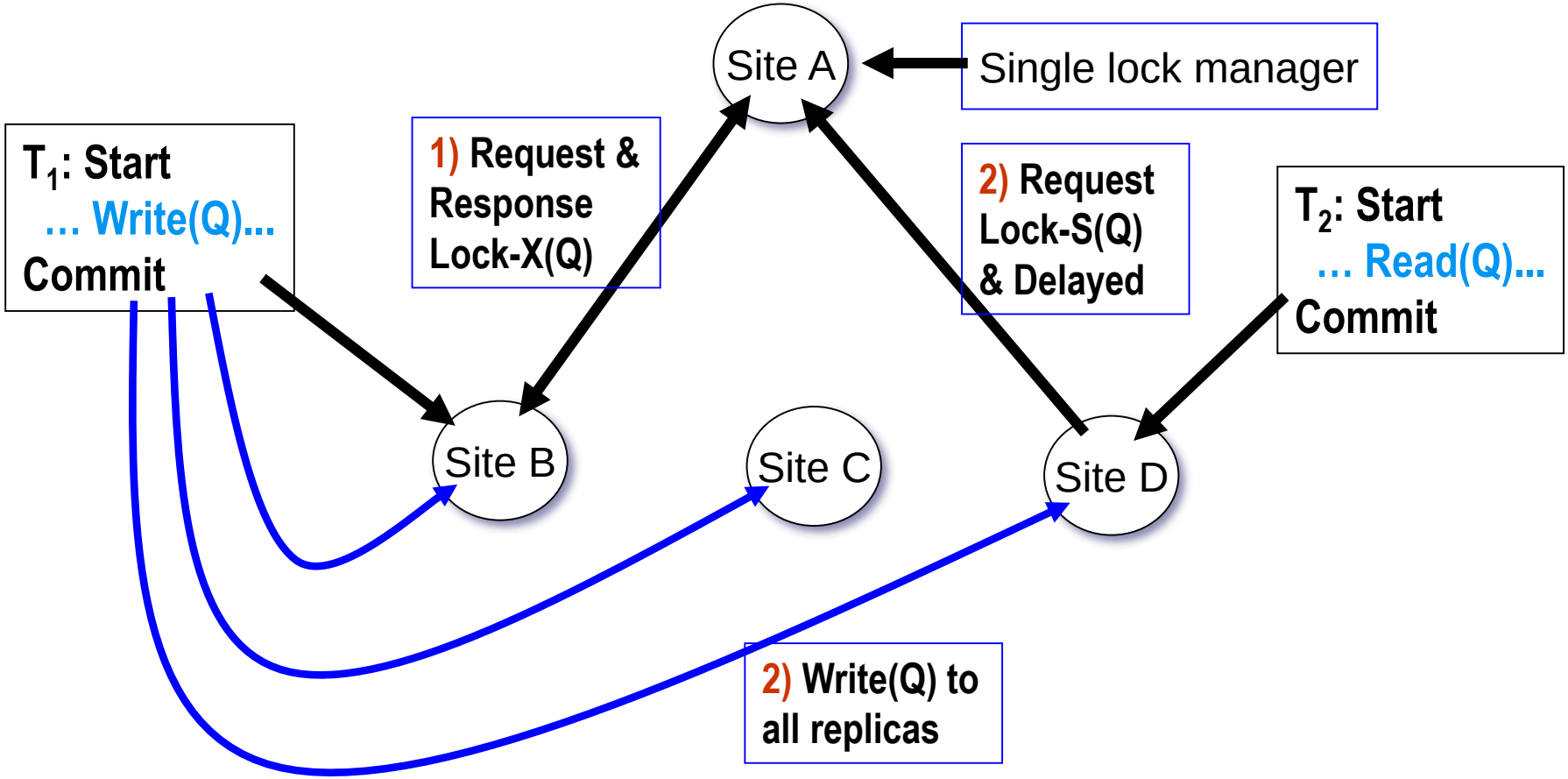


Single-Lock-Manager Approach

- System maintains a *single* lock manager that resides in a *single* chosen site, say S_i
- When a transaction needs to lock a data item, it sends a lock request to S_i and lock manager determines whether the lock can be granted immediately
  - If yes, lock manager sends a message to the site which initiated the request
  - If no, request is delayed until it can be granted, at which time a message is sent to the initiating site
- The transaction can read the data item from *any* one of the sites at which a replica of the data item resides
- Writes must be performed on all replicas of a data item
- Advantages of single-lock manager scheme:
  - Simple implementation
  - Simple deadlock handling
- Disadvantages of single-lock manager scheme:
  - Bottleneck: lock manager site becomes a bottleneck
  - Vulnerability: system is vulnerable to lock manager site failure

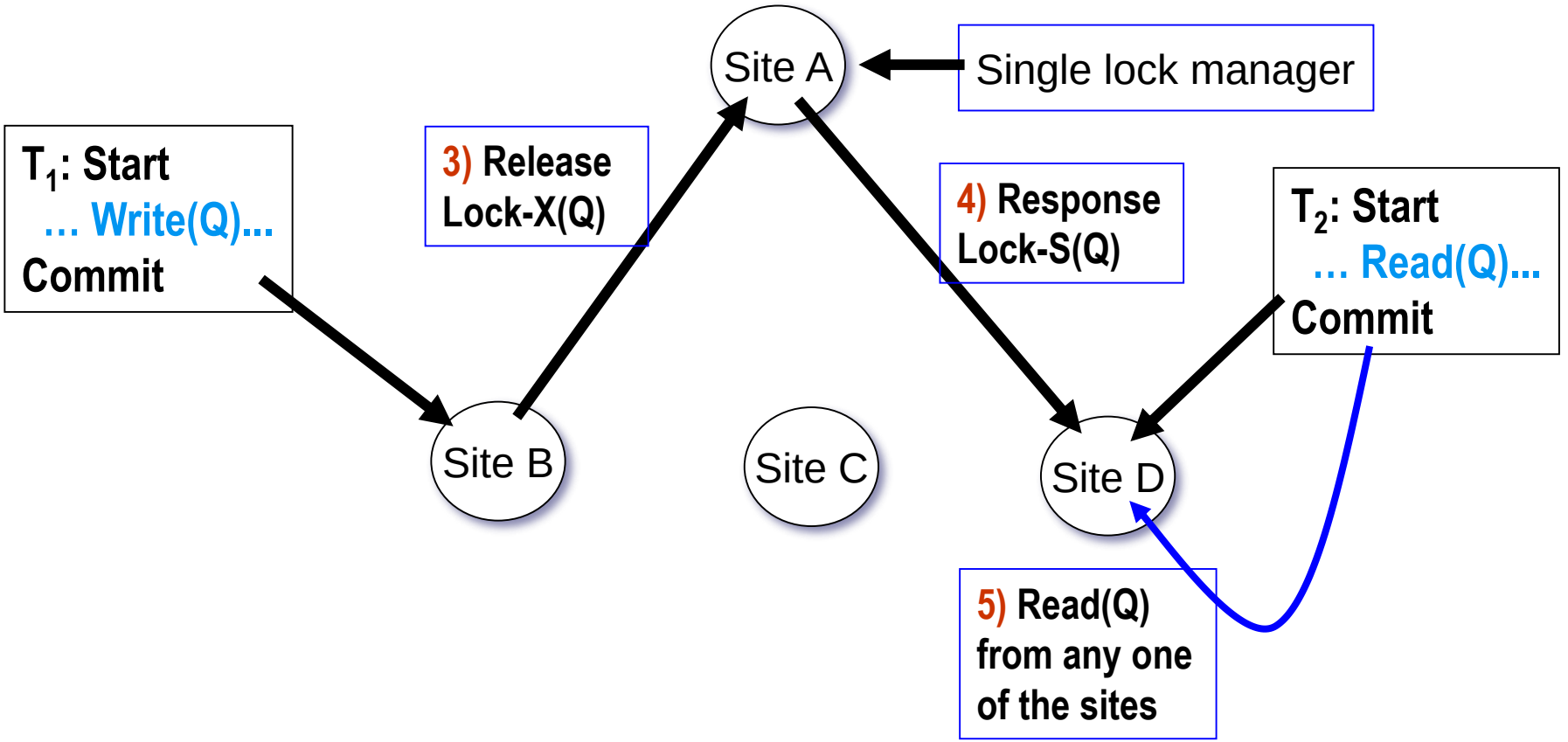
Single-Lock Manager Distributed CC – Example [1/2]

Assume Site A is the single lock manager, and
Site B, C, D contains a replica of data item Q



Single-Lock Manager Distributed CC – Example [2/2]

Assume Site A is the single lock manager, and
Site B, C, D contains a replica of data item Q



Distributed Lock Manager

- Functionality of locking is implemented by lock managers at each site
  - Lock managers control access to local data items
    - But special protocols may be used for replicas
- Advantage: work is distributed and can be made robust to failures
- Disadvantage: deadlock detection is more complicated
  - Lock managers cooperate for deadlock detection (More on this later)

- Several variants of distributed locking
  - Primary copy
  - Majority protocol
  - Biased protocol
  - Quorum consensus

Primary Copy Distributed Locking

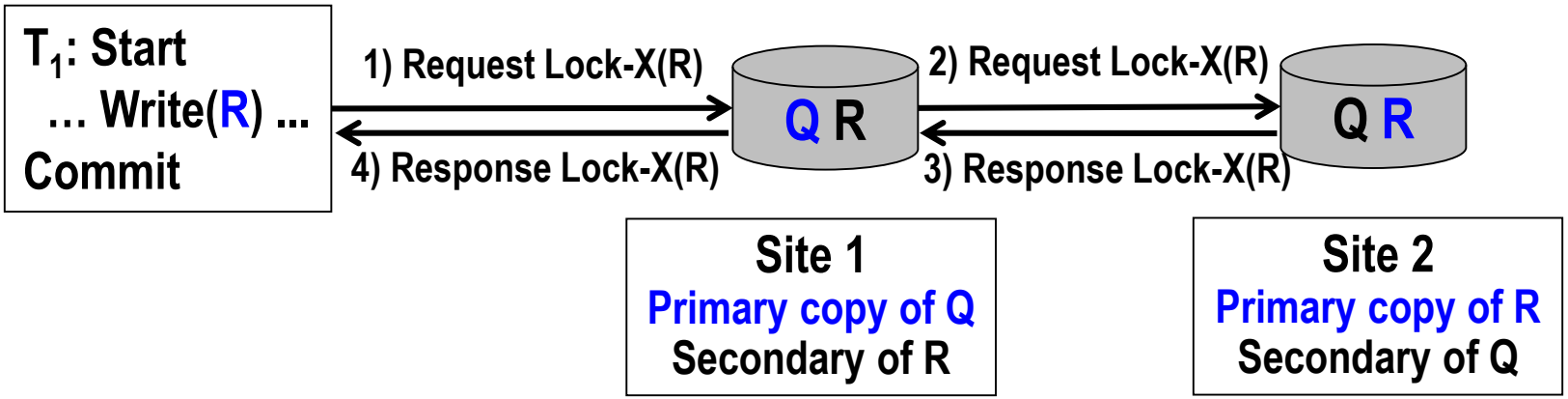
- Choose one replica of data item to be the **primary copy**
  - Site containing the replica is called the **primary site** for that data item
  - Different data items can have different primary sites
- When a transaction needs to lock a data item *Q*, it requests a lock at the primary site of *Q*
  - Implicitly gets lock on all replicas of the data item
- Benefit
  - Concurrency control for replicated data handled similarly to unreplicated data - simple implementation
- Drawback
  - If the primary site of *Q* fails, *Q* is inaccessible even though other sites containing a replica may be accessible

Primary Copy Distributed Locking– Example

When T_1 needs to lock a data item R

It requests a lock at the primary site of R

Concurrency control for replicated data handled like unreplicated data



If the site fails, T1 cannot proceed

Majority-based Distributed Locking Protocol [1/2]

- Local lock manager at each site administers lock and unlock requests for data items stored at that site
- When a transaction wishes to lock an unreplicated data item *Q* residing at site S_i, a message is sent to S_i 's lock manager
  - If *Q* is locked in an incompatible mode, then the request is delayed until it can be granted
  - When the lock request can be granted, the lock manager sends a message back to the initiator indicating that the lock request has been granted
- In case of replicated data Q
  - If *Q* is replicated at n sites, then a lock request message must be sent to more than half of the n sites in which *Q* is stored
  - The transaction does not operate on *Q* until it has obtained a lock on a majority of the replicas of *Q*
  - When writing the data item, transaction performs writes on *all* replicas

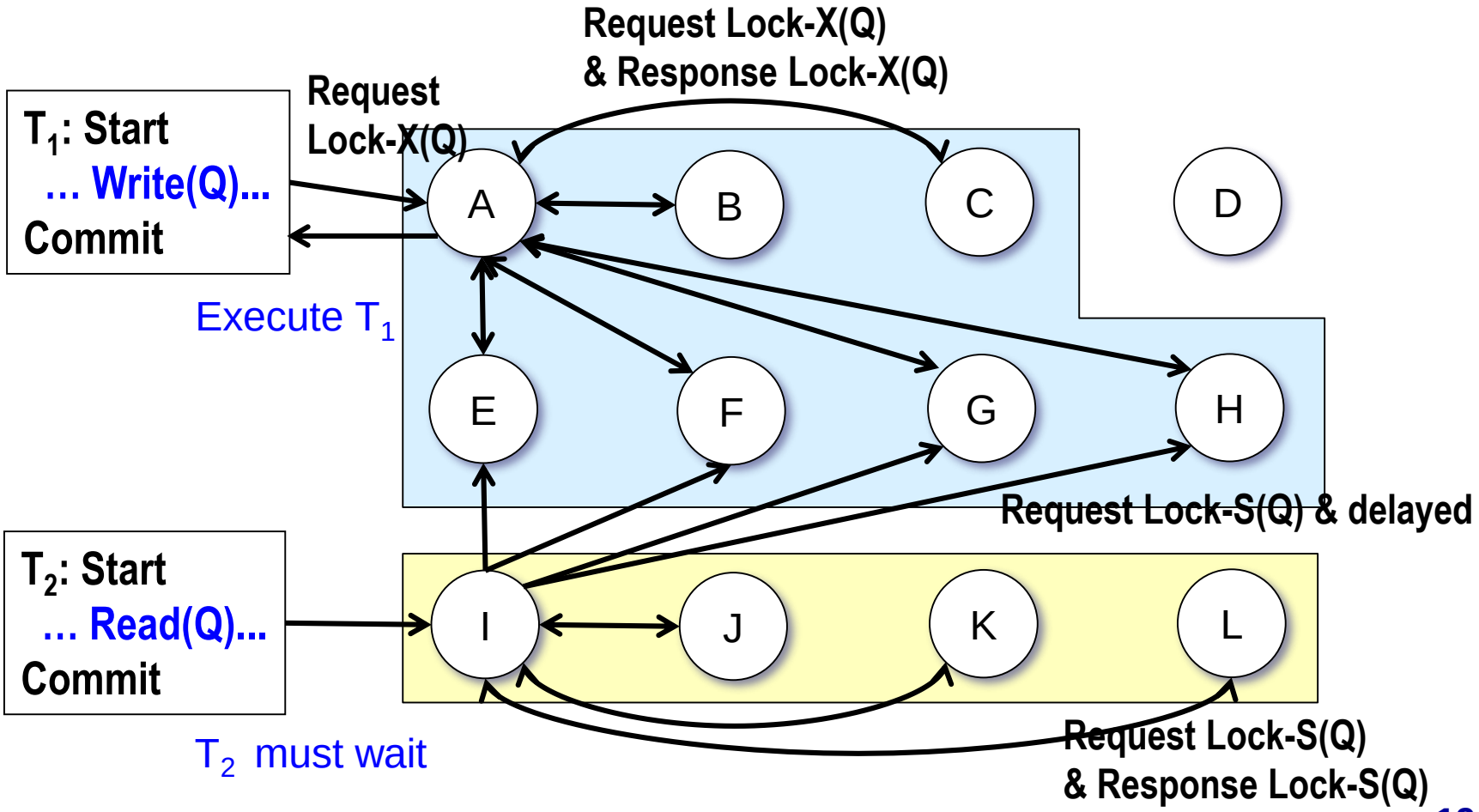
Majority-based Distributed Locking Protocol [2/2]

- Benefit
  - Can be used even when some sites are unavailable
    - details on how handle writes in the presence of site failure later
- Drawback
  - Requires $2(n/2 + 1)$ messages for handling lock requests, and $(n/2 + 1)$ messages for handling unlock requests
  - Potential for deadlock even with single item
    - e.g., each of 3 transactions may have locks on 1/3rd of the replicas of a data

Majority-based Distributed Locking Protocol: Example [1/2

Key idea : locks from **more than one-half** sites required for read or write

Assume 12 sites(A~L) containing a replica of Q → 7 locks required for read / write

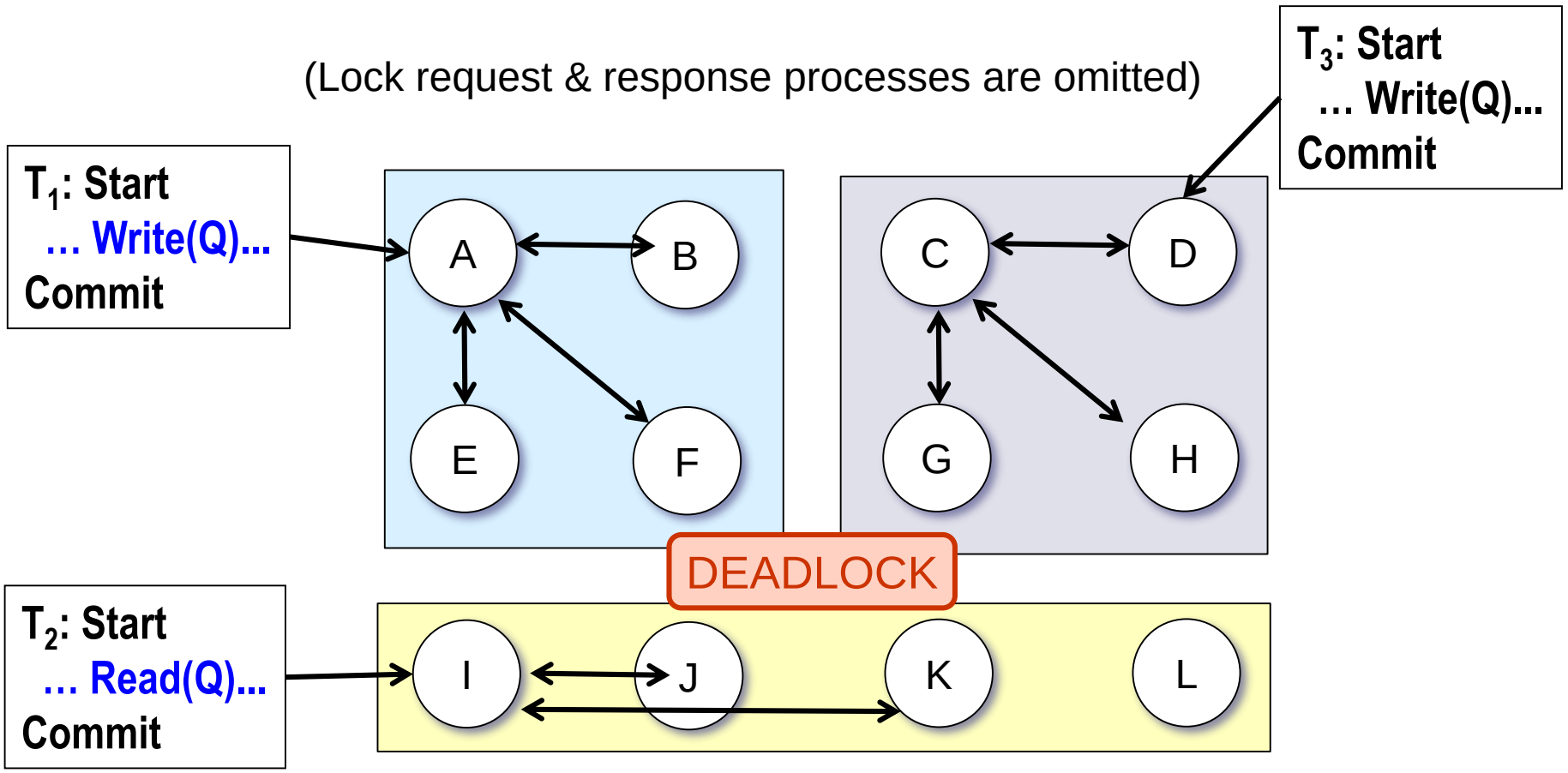


Majority-based Distributed Locking Protocol: Example [2/2

Key idea : locks from **more than one-half** sites required for read or write

Assume 12 sites(A~L) containing a replica of Q → 7 locks required for read / write

(Lock request & response processes are omitted)



Biased Distributed Locking Protocol

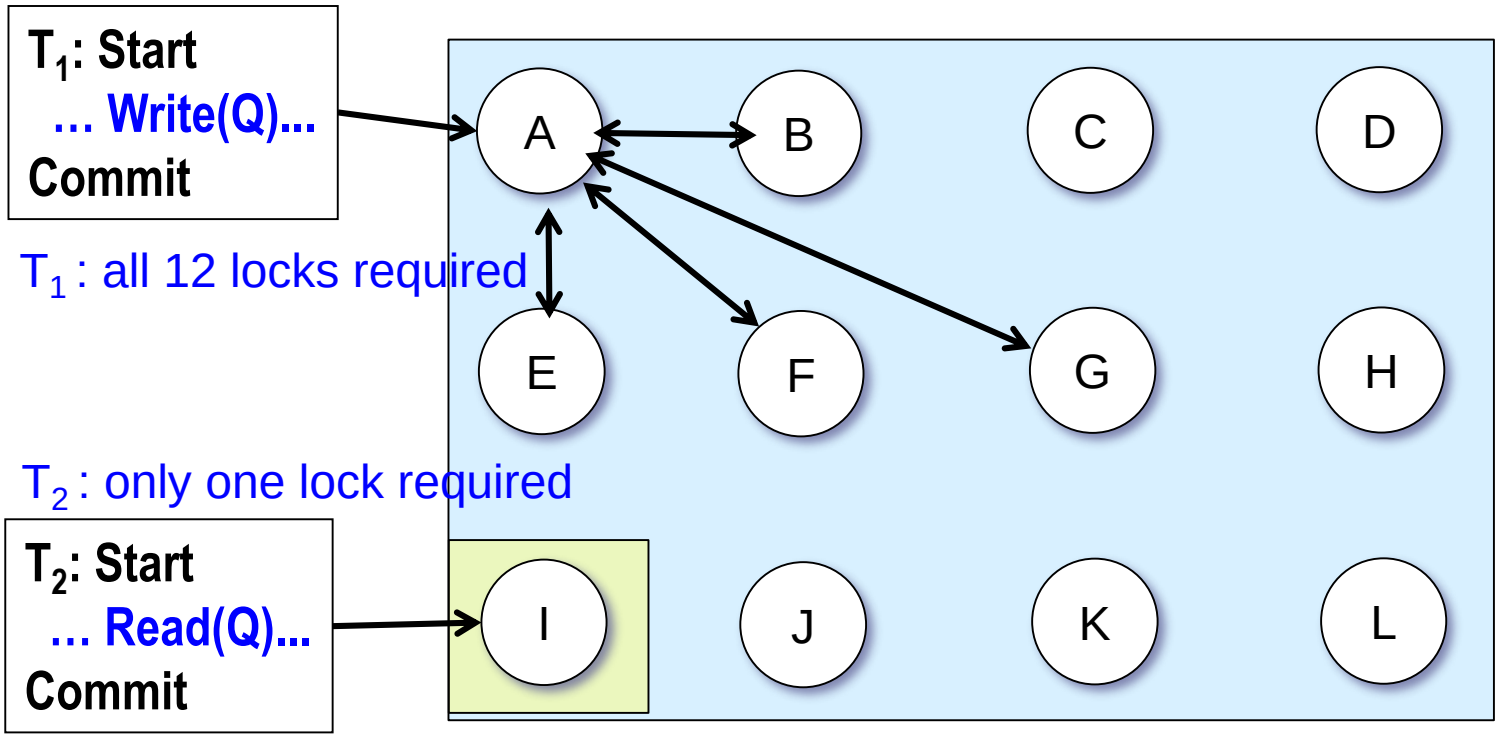
- Local lock manager at each site as in majority protocol, however, requests for shared locks are handled differently than requests for exclusive locks
- **Shared locks**
  - When a transaction needs to lock data item *Q*, it simply requests a lock on *Q* from the lock manager at one site containing a replica of *Q*
- **Exclusive locks**
  - When transaction needs to lock data item *Q*, it requests a lock on *Q* from the lock manager at all sites containing a replica of *Q*
- Advantage - imposes less overhead on **read** operations
- Disadvantage - additional overhead on **writes**

Biased Distributed Locking Protocol: Example

Key idea : **only one** lock is required for read while **locks from all sites** is required for writ

Assume 12 sites(A~L) containing a replica of Q

(Lock request & response processes are omitted)



Quorum Consensus Protocol

- A generalization of both majority and biased protocols
  - More flexible than majority or biased protocols ➔ More concurrency
- Each site is assigned a weight
  - Let S be the total of all site weights
- Choose two values read quorum Q_r and write quorum Q_w
  - Such that $Q_r + Q_w > S$ and $2 * Q_w > S$
  - Quorums can be chosen (and S computed) separately for each item
- Each read must lock enough replicas that the sum of the site weights is ≥ Q_r
- Each write must lock enough replicas that the sum of the site weights is ≥ Q_w
- For now we assume all replicas are written
  - Extensions to allow some sites to be unavailable described later

  - Quorum: 정원, 정족수

Quorum Consensus Distributed Locking Protocol: Example [1/2]

Key idea : $Q_r + Q_W > S$ and $2 * Q_W > S$ (Q_r : read quorum, Q_w : write quorum, S: # of sites)

Assume 12 sites(A~L) containing a replica of Q

Case 1: $Q_r = 3$, $Q_W = 10$

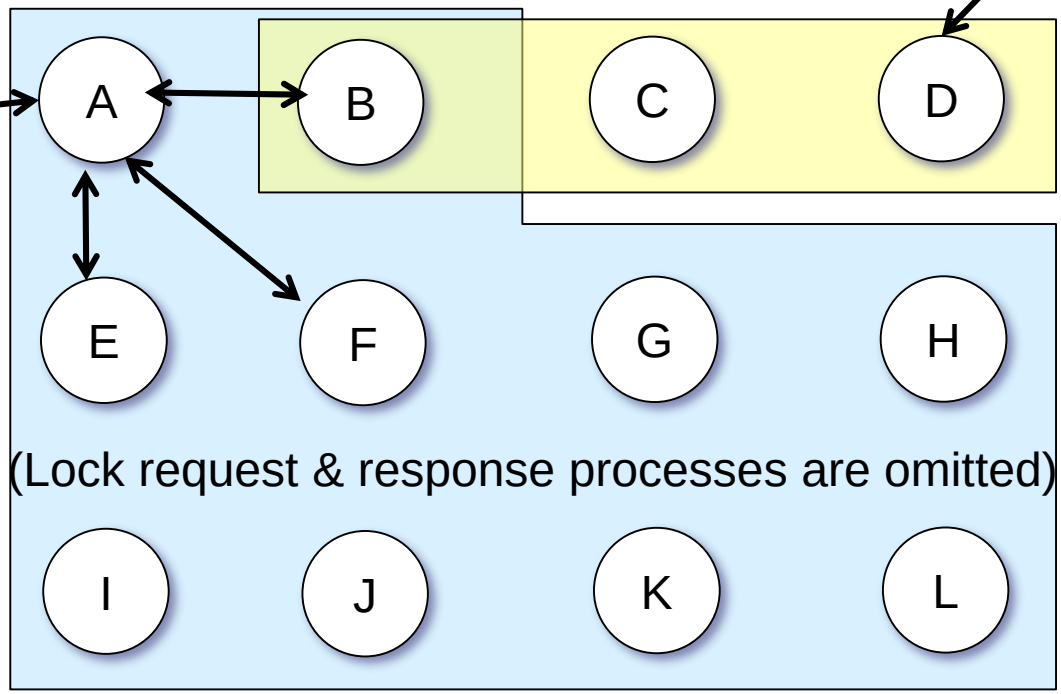
3 + 10 > 12, 2 * 10 > 12 → correct choice

T_1: Start
… Write(Q)...
Commit

T_1 : 10 locks required

T_2: Start
… Read(Q)...
Commit

T_2 : 3 locks required



Quorum Consensus Distributed Locking Protocol : Example [2/2]

Case 2: $Q_r = 7$, $Q_W = 6$

7 + 6 > 12, but ~~2 * 6 > 12~~ → incorrect choice

Case 3: $Q_r = 1$, $Q_W = 12$ (ROWA: Read-One, Write-All = Biased protocol)

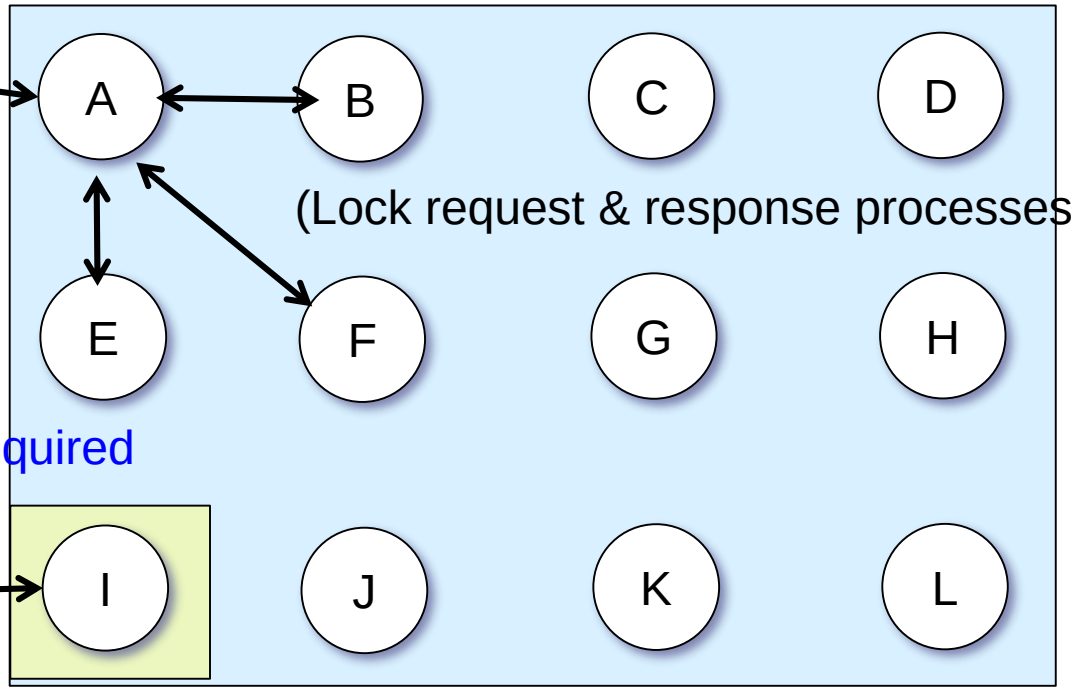
1 + 12 > 12, 2 * 12 > 12 → correct choice

T_1: Start
… Write(Q)...
Commit

T_1 : all 12 locks required

T_2 : only one lock required

T_2: Start
… Read(Q)...
Commit



Timestamping Distributed CC [1/2]

- Timestamp based concurrency-control protocols can be used in distributed systems

  Read of T_i is allowed If $TS(T_i) \geq$ W-timestamp(Q)

  Write of T_i is allowed If ($TS(T_i) \geq$ R-timestamp(Q) and $TS(T_i) \geq$ W-timestamp(Q))

- Each transaction must be given a unique timestamp
- Main problem: how to generate a timestamp in a distributed fashion
  - Each site generates a unique local timestamp using either a logical counter or the local clock
  - Global unique timestamp is obtained by concatenating the unique local timestamp with the unique identifier

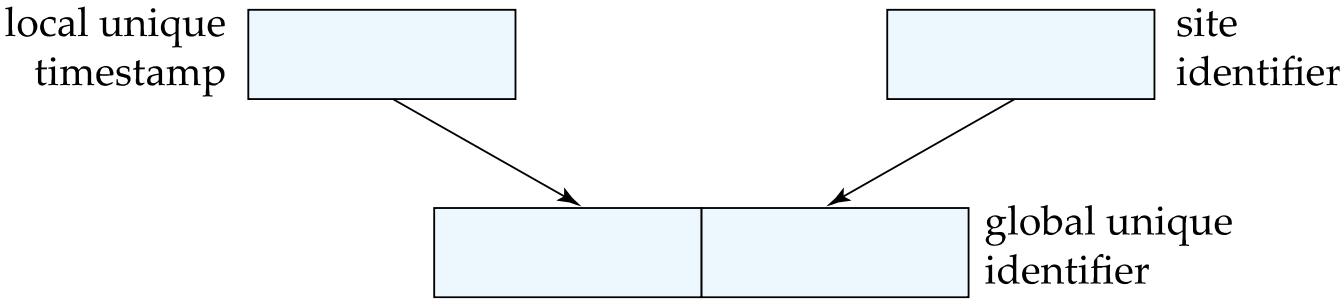


Fig 19.03

Timestamping Distributed CC [2/2]

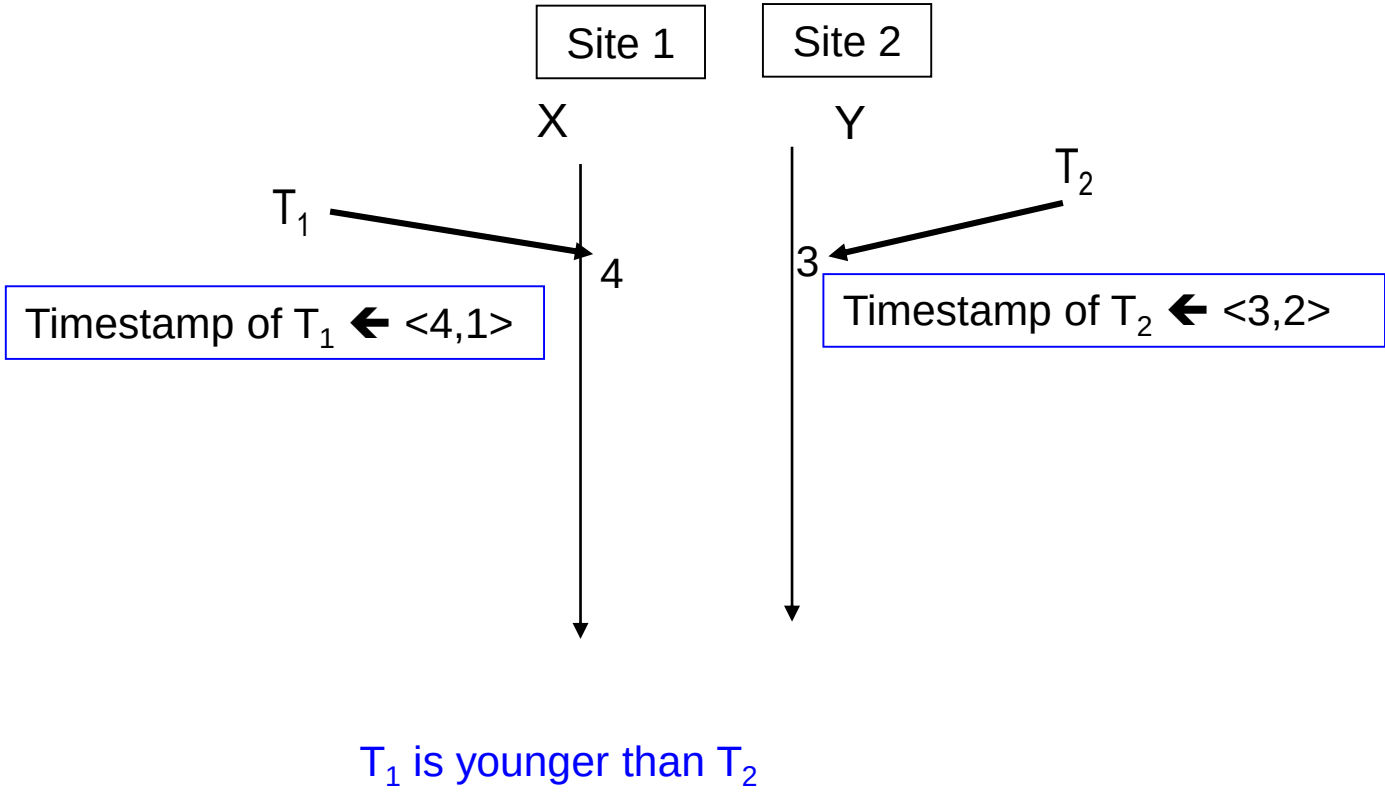
- A site with a slow clock will assign smaller timestamps
  - Still logically correct: serializability not affected
  - But: “disadvantages” transactions
- To fix this problem
  - Define within each site S_i a **logical clock** (LC_i), which generates the unique local timestamp
  - Require that S_i advance its logical clock whenever a request is received from a transaction Ti with timestamp <*x,y*> and x is greater that the current value of Lc_i *and y is a site id*
  - In this case, site S_i advances its logical clock to the value *x* + 1.

Timestamping Distributed CC – Example [1/3]

Assume Site 1 contains a data item X and logical clock "4",
Site 2 contains a data item Y and logical clock "3".

Assume R-Timestamp(X) = W-Timestamp(X) = <3,1>
R-Timestamp(Y) = W-Timestamp(Y) = <2,2>

T_1	T_2
read(*X*) write(*X*)	
	read(*Y*)
read(*Y*) write(*Y*)	
	read(*X*) write(*X*)

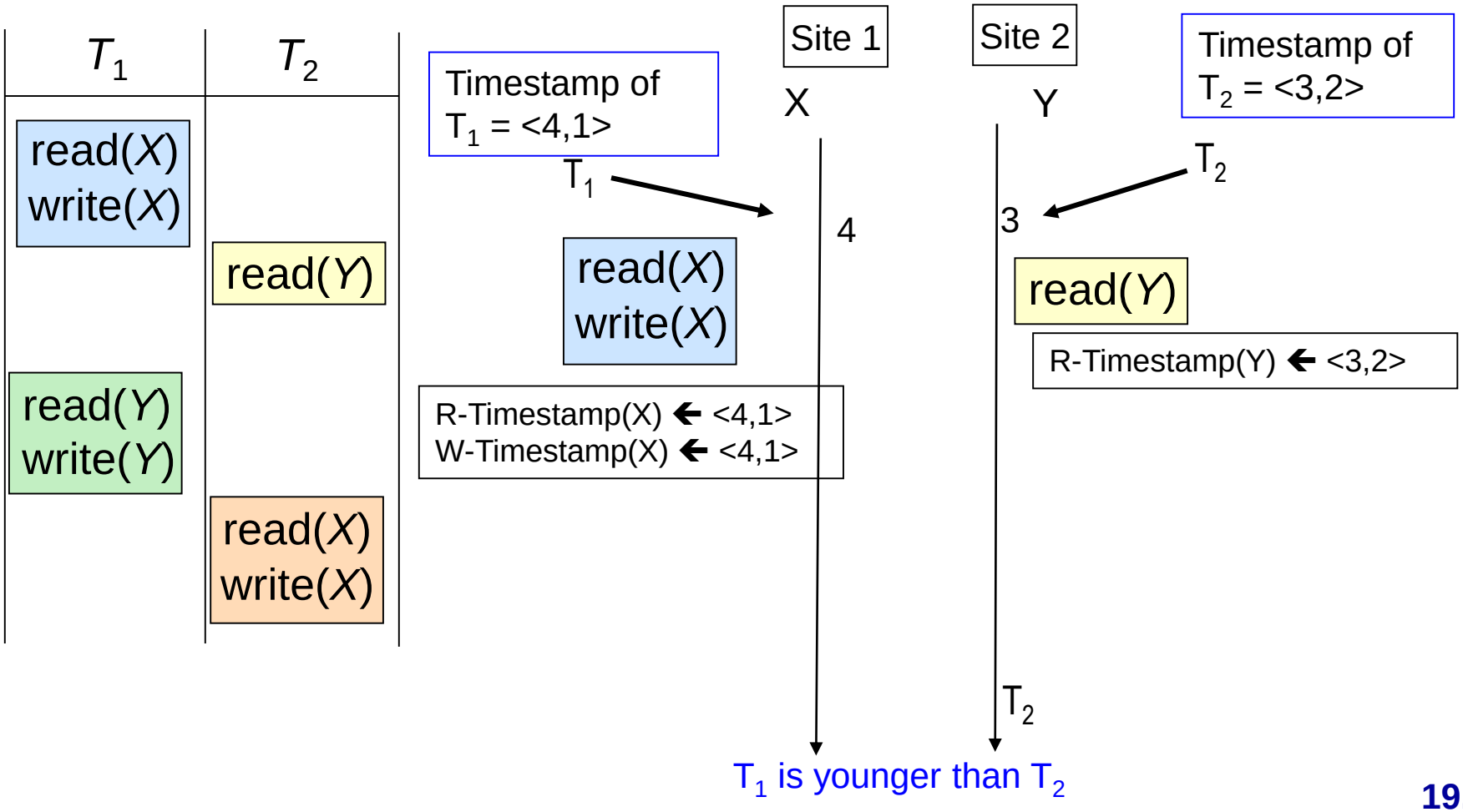


T_1 is younger than T_2

Timestamping Distributed CC – Example [2/3]

Assume Site 1 contains a data item X and logical clock "4",
Site 2 contains a data item Y and logical clock "3".

Assume R-Timestamp(X) = W-Timestamp(X) = <3,1>
R-Timestamp(Y) = W-Timestamp(Y) = <2,2>



Timestamping Distributed CC – Example [3/3]

Assume Site 1 contains a data item X and logical clock "4",
Site 2 contains a data item Y and logical clock "3".

Assume R-Timestamp(X) = W-Timestamp(X) = <3,1>
R-Timestamp(Y) = W-Timestamp(Y) = <2,2>

T_1	T_2
read(*X*)	
write(*X*)	
	read(*Y*)
read(*Y*)	
write(*Y*)	
	read(*X*)
	write(*X*)

Timestamp of T_1 = <4,1>

Site 1 — X

Site 2 — Y

Timestamp of T_2 = <3,2>

T_1

4

read(*X*)
write(*X*)

T_2

3

read(*Y*)

R-Timestamp(Y) ← <3,2>

R-Timestamp(X) ← <4,1>
W-Timestamp(X) ← <4,1>

read(*Y*)
write(*Y*)

T_1

5

R-Timestamp(Y) ← <4,1>
W-Timestamp(Y) ← <4,1>

T_2

Abort!

T_1 is younger than T_2

Replication-based Distributed CC with Weak Consistency

- Many commercial databases support replication of data with weak degrees of consistency (I.e., without guarantee of serializability)
  - Neither distributed locks nor distributed timestamping
  - Two kinds of sites: Master site (master copy) and Slave site (replicas)
  - Useful for running read-only queries to slave sites
- Master-slave replication distributed concurrency control
  - Updates are allowed only in a “master” site, and propagated to “slave” sites
    - There could be inconsistency between Master site and Slave sites
  - Single-Master vs. Multi-Master

- Replicas should see a **transaction-consistent snapshot** of the database
  - That is, a state of the database reflecting all effects of all transactions up to some point in the serialization order, and no effects of any later transactions
- E.g. Oracle provides a **create snapshot** statement to create a snapshot of a relation or a set of relations at a remote site
  - Snapshot refresh either by recomputation or by incremental update
  - Automatic refresh (continuous or periodic) or manual refresh

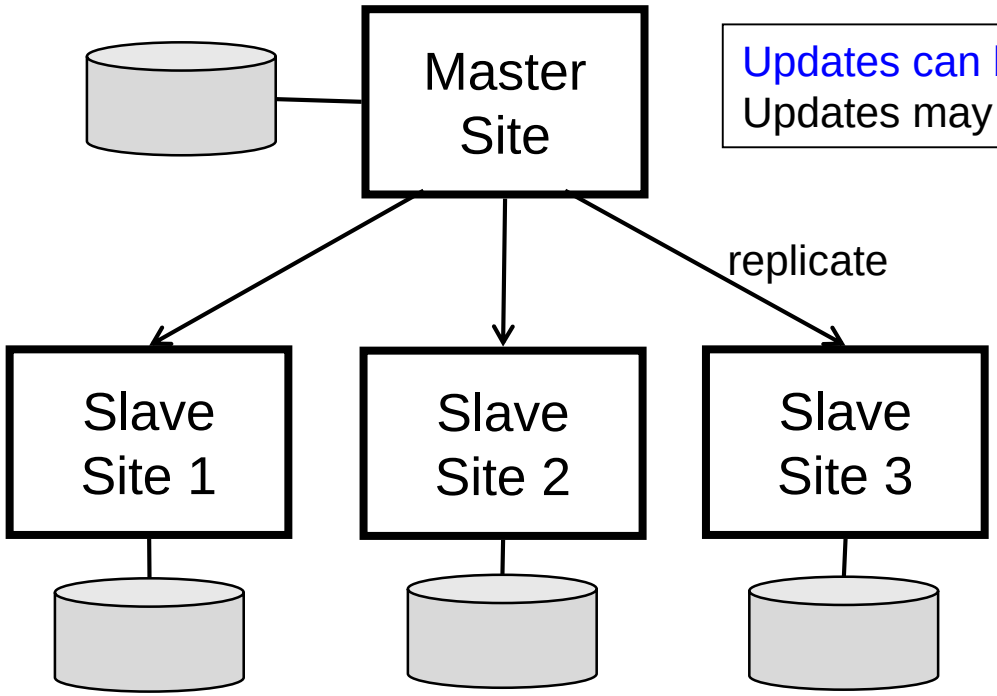
Master-Slave Replication Distributed CC

- **Master-Slave Replication Distributed CC**
  - Updates are performed at a single "master" site, and propagated to "slave" sites
  - Propagation is not part of the update transaction: it is decoupled
    - May be immediately after transaction commits & May be periodic
  - Data may only be read at slave sites, not updated
    - No need to obtain locks at any remote site
  - Particularly useful for distributing information
    - E.g. from central office to branch-office
  - Also useful for running read-only queries offline from the main database

Master-Slave Replication Distributed CC– Example

The database allows updates at a primary site,
and propagates updates to replicas at other sites.

Useful case: 1) replication from a central office to branch offices
2) run large read-only queries without interferences



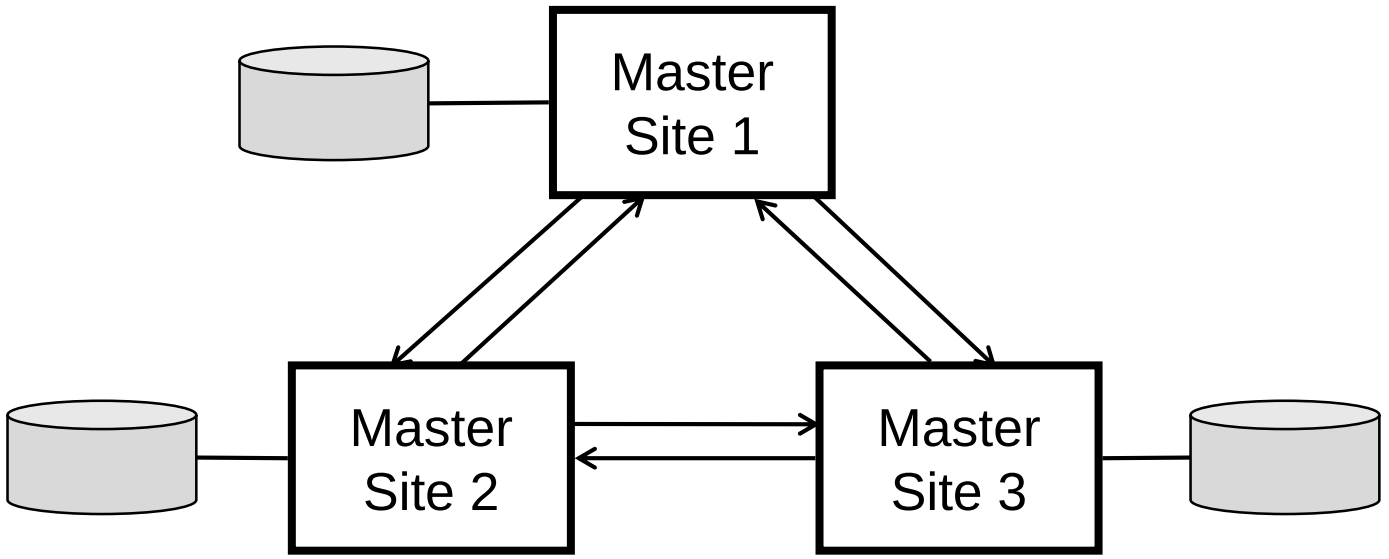
Updates can be occurred at master site only
Updates may be propagated periodically

Read-only queries are allowed only in slave sites

Multimaster Replication Distributed CC with Lazy Propagation

- With multimaster replication (also called update-anywhere replication) updates are permitted at any replica, and are automatically propagated to all replicas
  - Basic model in distributed databases, where transactions are unaware of the details of replication, and database system propagates updates as part of the same transaction
    - Coupled with 2 phase commit
- Many systems support **lazy propagation** where updates are transmitted after transaction commits
  - Allows updates to occur even if some sites are disconnected from the network, but at the cost of consistency

Multimaster Replication Distributed CC with Lazy Propagation: example



- Lazy propagation – two approaches
  - Updates at replicas are translated into updates at a primary site, which are then propagated lazily to all replicas.
  - Updates are performed at any replica and propagated to all other replicas

  → The above schemes should be used with care. (resolving conflict required)

Deadlock in Lock-based Distributed CC

Consider the following two transactions and history, with item X and transaction T_1 at site 1, and item Y and transaction T_2 at site 2:

T_1: write (X)
write (Y)

T_2: write (Y)
write (X)

X-lock on X write (X)	
	X-lock on Y write (Y) wait for X-lock on X
Wait for X-lock on Y	

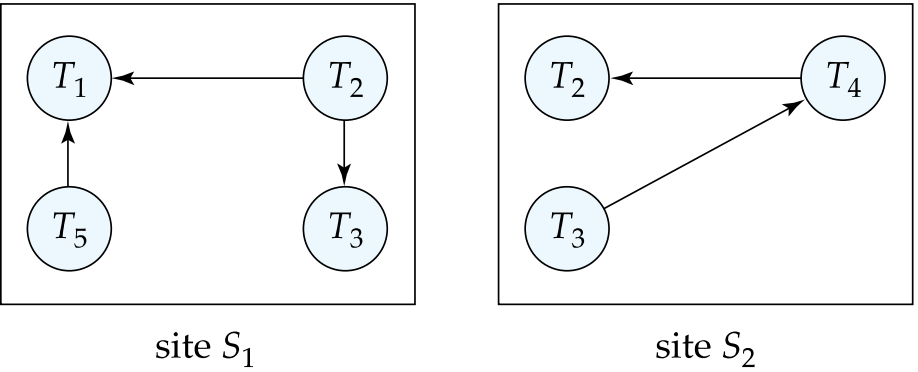
Result: deadlock which cannot be detected locally at either site

Centralized Approach for Deadlock Handling

- A global wait-for graph is constructed and maintained in a *single* site (i.e., the deadlock-detection coordinator)
  - *Real graph*: Real, but unknown, state of the system
  - *Constructed graph*: Approximation generated by the controller during the execution of its algorithm
- The global wait-for graph can be constructed when:
  - a new edge is inserted in or removed from one of the local wait-for graphs
  - a number of changes have occurred in a local wait-for graph
  - the coordinator needs to invoke cycle-detection
- If the coordinator finds a cycle, it selects a victim and notifies all sites
  - The sites roll back the victim transaction

Local

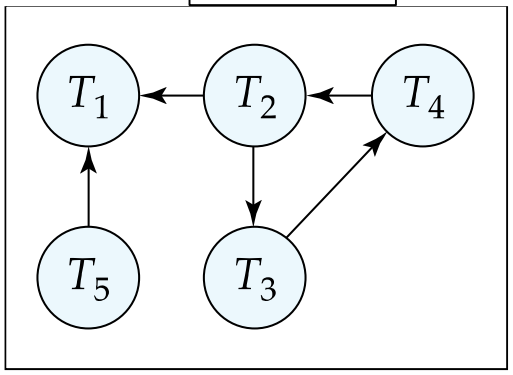
Fig 19.04



site S_1 site S_2

Global

Fig 19.05



Unnecessary Rollbacks due to False Cycles in Global Wait-For Graph [1/2]

Initial state:

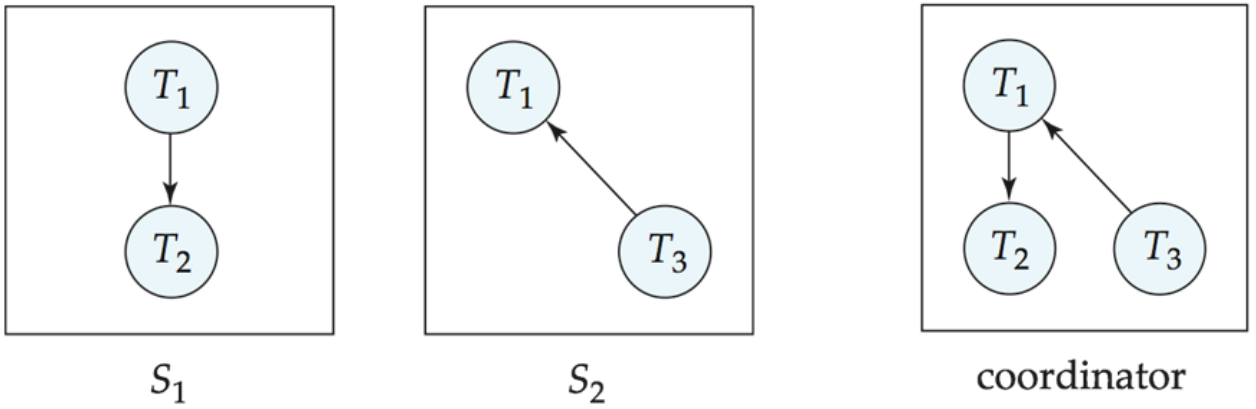
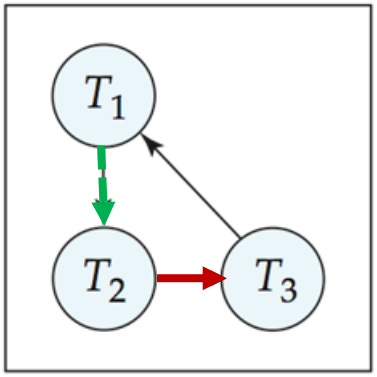


Fig 19.06

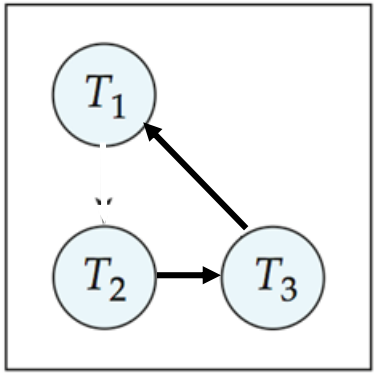
- Suppose that starting from the state shown in the above figure
  1. T_2 releases resources at S_1
     - send "remove $T_1 \rightarrow T_2$ " message from S_1 to the coordinator
  2. And then T_2 requests a resource held by T_3 at site S_2
     - send "insert $T_2 \rightarrow T_3$ " message from S_2 to the coordinator
- Suppose further that the **insert** message reaches before the **remove** message
  - this can happen due to network delays

Unnecessary Rollbacks due to False Cycles in Global Wait-For Graph [2/2]

False Cycle:



Reality:



- The coordinator would then find a false cycle: $T_1 \rightarrow T_2 \rightarrow T_3 \rightarrow T_1$
  - The false cycle above never existed in reality
  - False cycles cannot occur if two-phase locking is used
- Unnecessary rollbacks can result from false cycles in the global wait-for graph; however, likelihood of false cycles is low

- Unnecessary rollbacks may also result
  - when deadlock has indeed occurred and a victim has been picked,
  - and meanwhile one of the transactions was aborted for reasons unrelated to the deadlock

Chapter 19: Distributed Databases

- 19.1 Heterogeneous and Homogeneous Databases
- 19.2 Distributed Data Storage
- 19.3 Distributed Transactions
- 19.4 Commit Protocols
- 19.5 Concurrency Control in Distributed Databases
- 19.6 Availability
- 19.7 Distributed Query Processing
- 19.8 Heterogeneous Distributed Databases
- 19.9 Cloud-Based Databases
- 19.10 Directory Systems

High Availability of Distributed Databases

- High availability: time for which system is not fully usable should be extremely low (e.g. 99.99% availability)
- Robustness: ability of system to function spite of failures of components
- Failures are more likely in large distributed systems

- To be robust, a distributed system must
  - Detect failures
  - Reconfigure the system so computation may continue
  - Recovery / Reintegration when a site or link is repaired
    - Coordinator election

- Failure detection: distinguishing link failure from site failure is hard
  - (partial) solution: have multiple links, multiple link failure is likely a site failure

Reconfiguration for Availabilty [1/2]

- Suppose that site S_i has discovered that a failure has occurred, initiate the reconfiguration procedure
  - Must abort all transactions that were active at a failed site
    - Making them wait could interfere with other transactions since they may hold locks on other sites
    - However, in case only some replicas of a data item failed, it may be possible to continue transactions that had accessed data at a failed site (more on this later)
  - If replicated data items were at failed site, update system catalog to remove them from the list of replicas
    - This should be reversed when failed site recovers, but additional care needs to be taken to bring values up to date
  - If a failed site was a central server for some subsystem, an **election** must be held to determine the new central server
    - E.g. name server, concurrency coordinator, global deadlock detector

Reconfiguration for Availability [2/2]

- Since network partition may not be distinguishable from site failure, the following situations must be avoided
  - Two ore more central servers elected in distinct partitions
  - More than one partition updates a replicated data item
- Updates must be able to continue even if some sites are down
- The majority based reconfiguration scheme
  - Modifying the majority-based distributed CC scheme for failures

- Alternative: The “read one write all available” reconfiguration scheme
  - is tantalizing but causes problems

Tantalize: bother

Tantalizing: 안타까운

Majority-Based Reconfiguration Scheme [1/2]

- The majority protocol for distributed concurrency control can be modified to work even if some sites are unavailable
  - Each replica of each item has a **version number** which is updated when the replica is updated, as outlined below
  - A read/write lock request is sent to at least ½ the sites at which item replicas are stored and operation continues only when a lock is obtained on a majority of the sites
- Read operations
  - Look at all replicas locked, and read the value from the replica with largest version number
  - May write this value and version number back to replicas with lower version numbers (no need to obtain locks on all replicas for this task)
- Write operations
  - Find highest version number like reads, and set new version number to old highest version + 1
  - Writes are then performed on all locked replicas and version number on these replicas is set to new version number

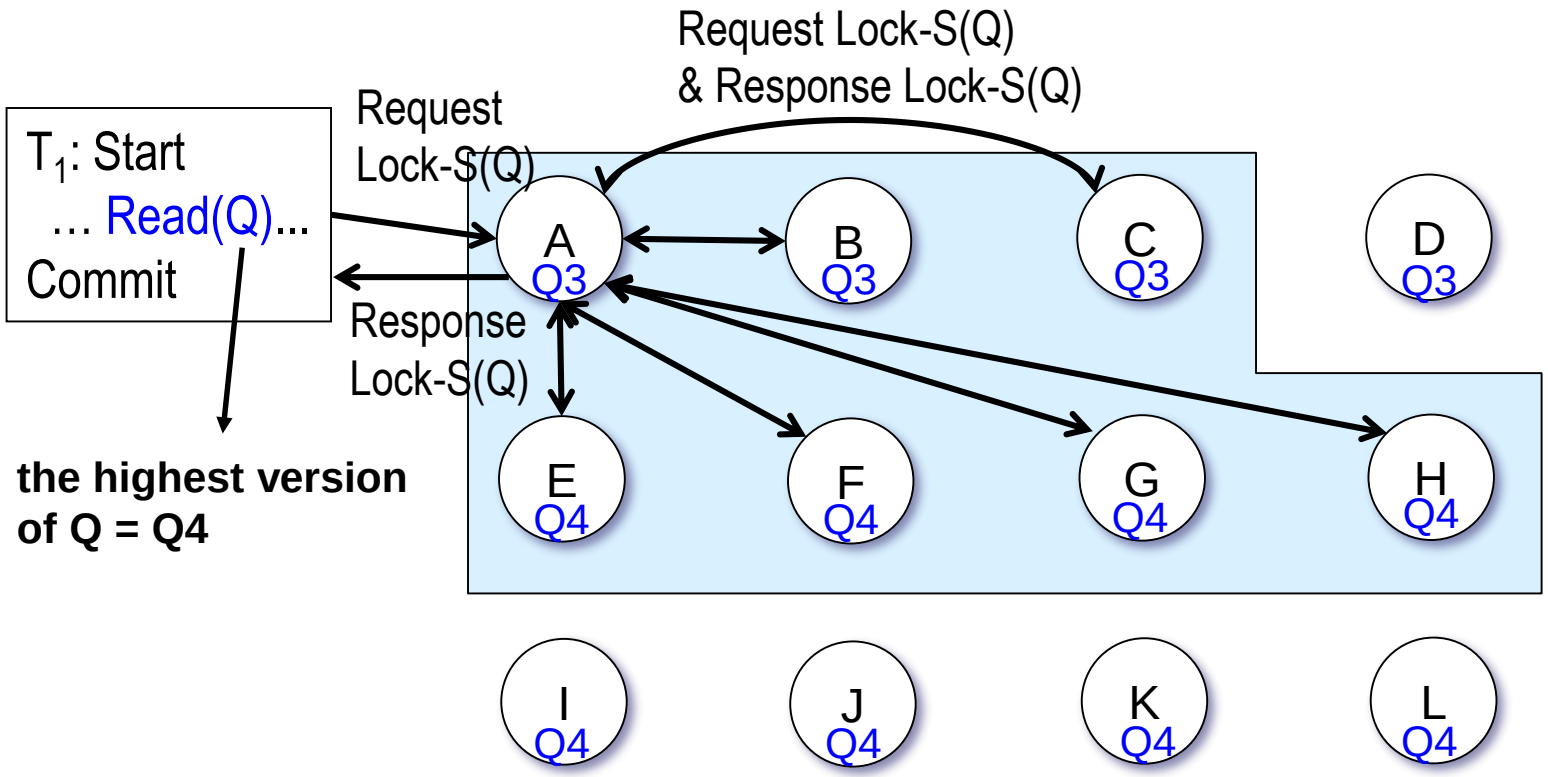
Majority-Based Reconfiguration Scheme [2/2]

- Failures (network and site) cause no problems as long as
  - Sites at commit contain a majority of replicas of any updated data items
  - During reads a majority of replicas are available to find version numbers
  - Subject to above, 2 phase commit can be used to update replicas
- Note: reads are guaranteed to see the latest version of data item
- Reintegration is trivial: nothing needs to be done
- Quorum consensus distributed CC algorithm can be similarly extended

Majority-Based Reconfiguration - Read

Key idea : **read the highest version** among replicas in **more than one-half** sites
write (the highest version + 1) to replicas in **more than one-half** sites

Assume 12 sites(A~L) containing a replica of Q → 7 locks required for read / write

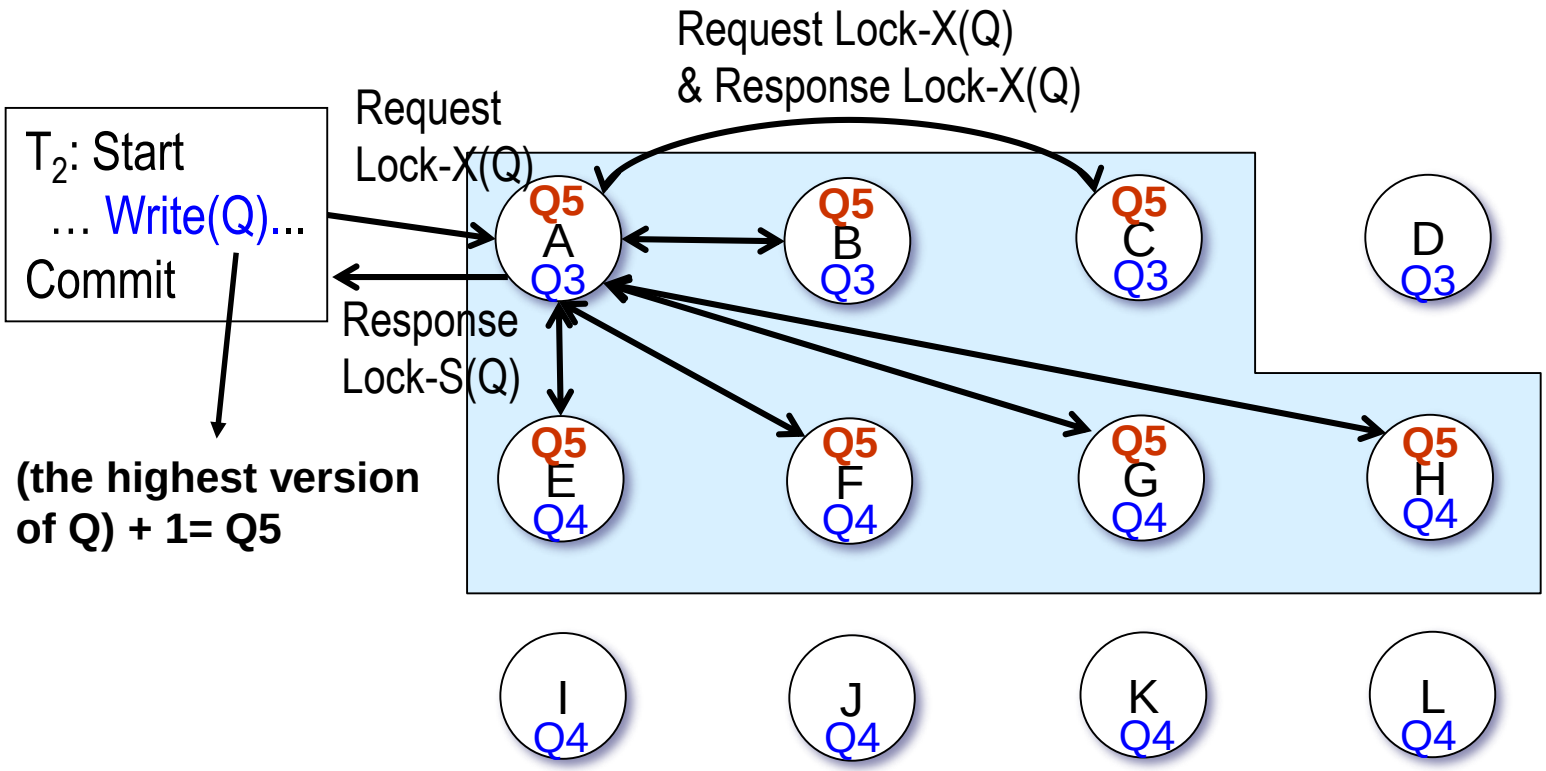


- Failures (network and site) cause no problems as long as more than one-half sites are alive
- Reintegration is trivial: nothing needs to be done

Majority-Based Reconfiguration - Write

Key idea : **read the highest version** among replicas in **more than one-half** sites
write (the highest version + 1) to replicas in **more than one-half** sites

Assume 12 sites(A~L) containing a replica of Q → 7 locks required for read / write



- Failures (network and site) cause no problems as long as more than one-half sites are alive
- Reintegration is trivial: nothing needs to be done

Read-One Write-All Reconfiguration Scheme

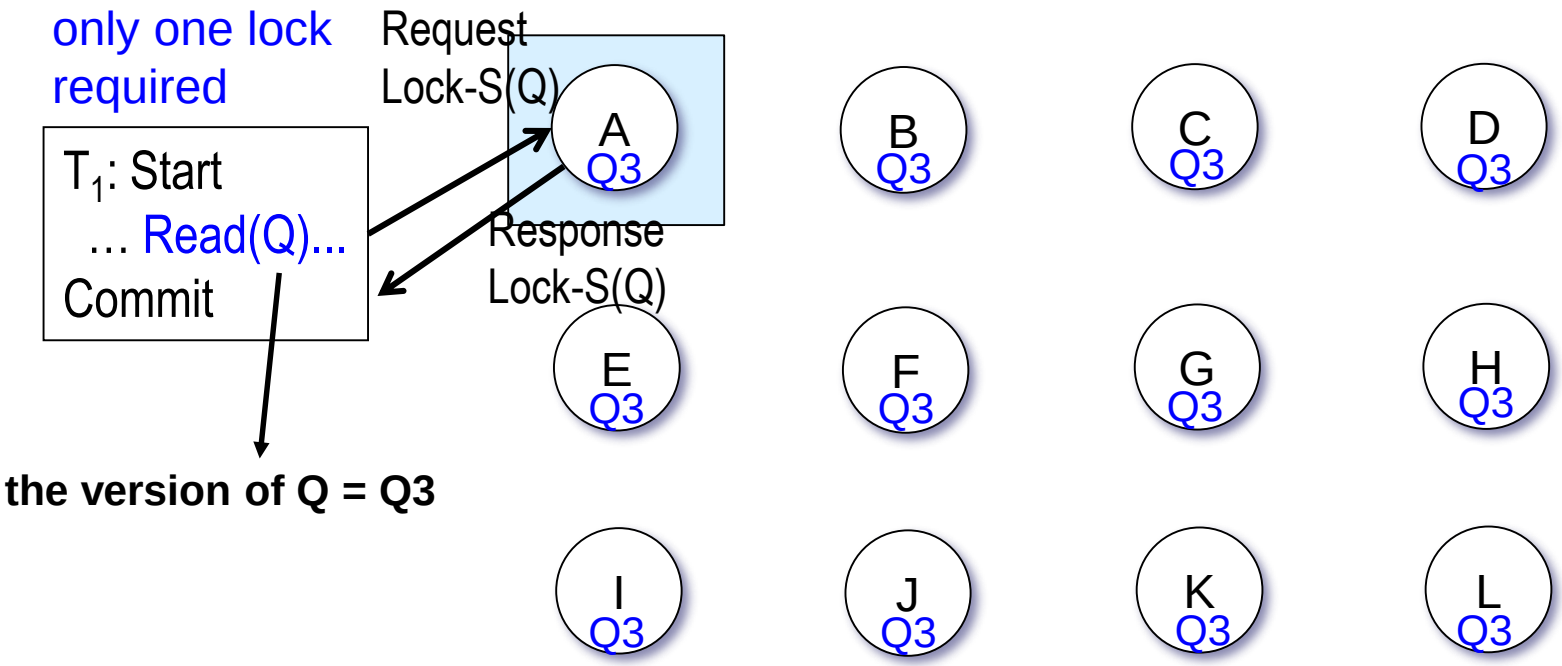
- Biased protocol is a special case of quorum consensus
  - Allows reads to read any one replica but updates require all replicas to be available at commit time (called **read one write all (ROMA)**)
- Read one write all available (ignoring failed sites) is attractive, but incorrect
  - Temporary communication failure results in write failure in a disconnected site
    - If a disconnected site was aware of failure, reintegration could have been performed, but no way to guarantee this
    - If a failed link may come back up, the disconnected site then has old values, and a read from the disconnected site would return an incorrect value
  - With network partitioning, sites in each partition may update the same item concurrently, believing sites in other partitions have all failed
- ROMA can be used if there is never any network partitioning
  - Otherwise, ROMA results in data inconsistencies

Read-One Write-All Reconfiguration - Read

Key idea : **read any replica** from **any site.**
write (the version + 1) to replicas of **all** sites.

Assume 12 sites(A~L) containing a replica of Q

In ROWA, all sites has the same version



- Temporary communication failures ➔ no write & no reintegration actions are performed
- Network partition ➔ each partition may independently update the same data item

Read-One Write-All Reconfiguration - Write

Key idea : **read any replica** from **any site.**
write (the version + 1) to replicas of **all** sites.

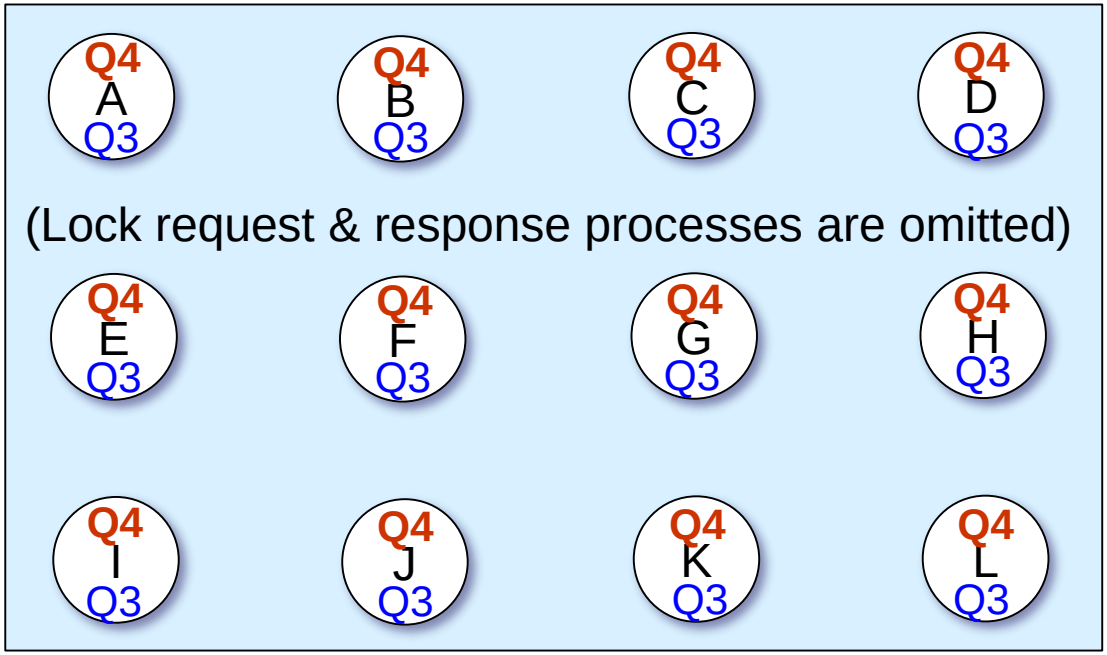
Assume 12 sites(A~L) containing a replica of Q

In ROWA, all sites has the same version

all 12 locks required

T_2: Start
… Write(Q)...
Commit

(the previous version of Q) + 1 = Q4



- Temporary communication failures ➔ no write & no reintegration actions are performed
- Network partition ➔ each partition may independently update the same data item

Site Reintegration after Failure Recovery

- When a failed site recovers, it must catch up with all updates that it missed while it was down
  - Problem: updates may be happening to items whose replica is stored at the site while the site is recovering
  - Solution 1: halt all updates on system while reintegrating a site
    - ➔ Unacceptable disruption
  - Solution 2: lock all replicas of all data items at the site, update to the latest version, then release locks
    - Other solutions with better concurrency also available

Distributed DB vs. Remote Backup

- Remote backup (hot spare) systems (Section 17.10) are also designed to provide high availability
  - Remote backup systems are simpler and have lower overhead
  - All actions performed at a single site, and only log records shipped
  - No need for distributed concurrency control, or 2 phase commit

- Using distributed databases with replicas of data items
  - can provide higher availability by having multiple (> 2) DB replicas and using the majority protocol
  - Also avoid failure detection and switchover time associated with remote backup systems

Coordinator Selection in Distributed DB

- Backup coordinator
  - is a site which maintains enough information locally to assume the role of coordinator if the actual coordinator fails
  - executes the same algorithms and maintains the same internal state information as the actual coordinator fails
  - allows fast recovery from coordinator failure but involves overhead during normal processing
  - does not take any action that affects other sites

- Election algorithms
  - used to elect a new coordinator in case of failures
  - Example: Bully Algorithm - applicable to systems where every site can send a message to every other site

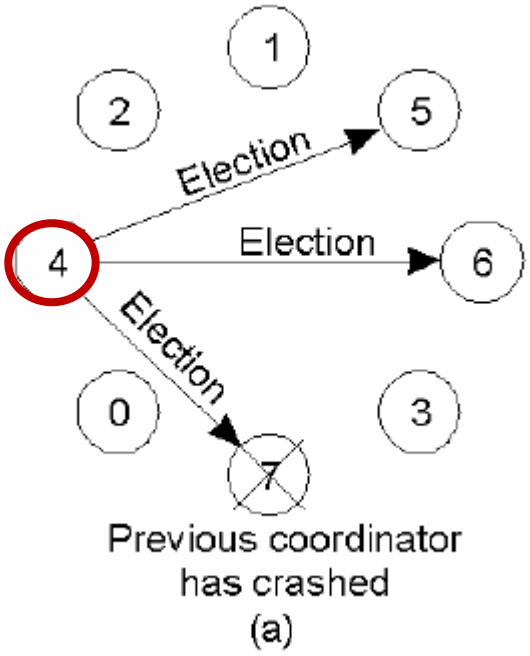
Coordinator Selection in Distributed DB: The Bully Algorithm

- If a site S_i sends a request that is not answered by the coordinator within a time interval *T*, the site S_i assume that the coordinator has failed
- S_i tries to elect itself as the new coordinator
- S_i sends an election message to every site with a higher identification number, then S_i waits for any of these processes to answer within *T*
- If no response within *T*, assume that all sites with number greater than *i* have failed, S_i elects itself the new coordinator
- If an answer is received, S_i begins another time interval *T'*, waiting to receive a message that a site with a higher identification number has been elected
- If no message is received within*T'*, assume the site with a higher number has failed; then S_i restarts the algorithm
- After a failed site recovers, it immediately begins execution of the same algorithm
- If there are no active sites with higher numbers, the recovered site forces all processes with lower numbers to let it become the coordinator site, even if there is a currently active coordinator with a lower number

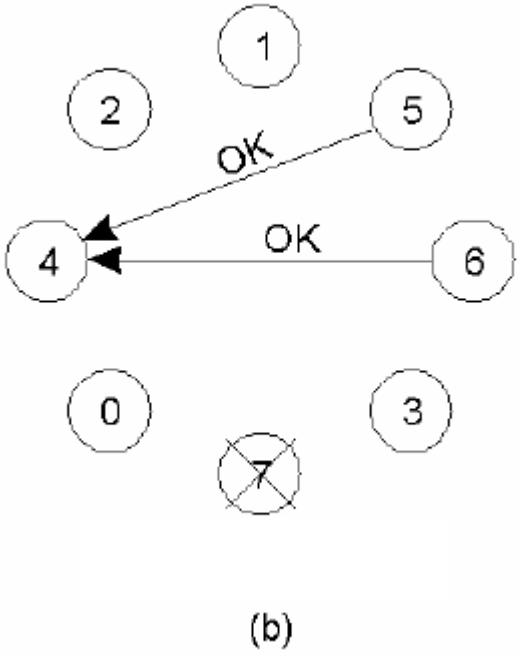
Bully: 골목대장, 조폭

Bully Algorithm – Example [1/2]

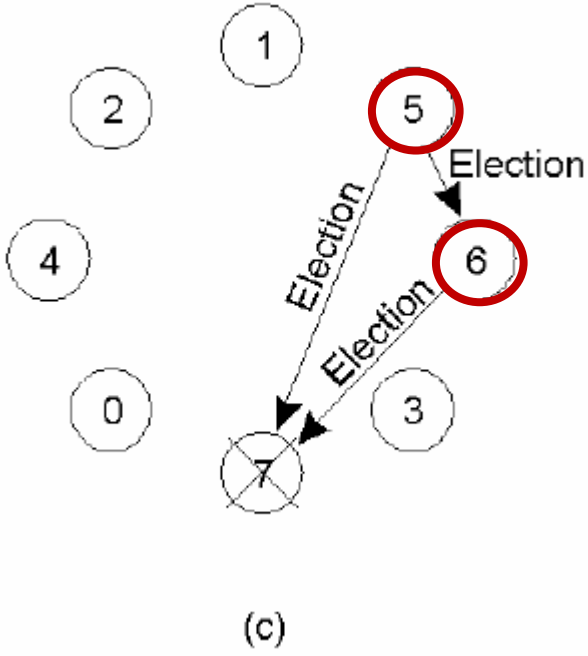
Suppose Site 4 notices the crash of coordinator site 7.
Site rank: site 7, site 6, site 5, ……, site 0



- Site 4 initiates an election
- Sends an Election message to all sites with higher #

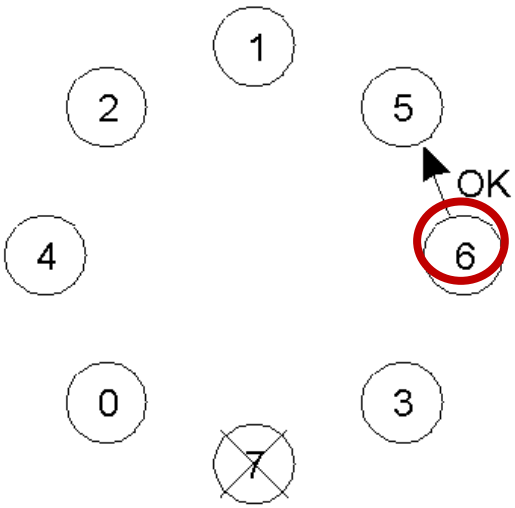


- Site 5 and 6 respond
- Site 4's job is over



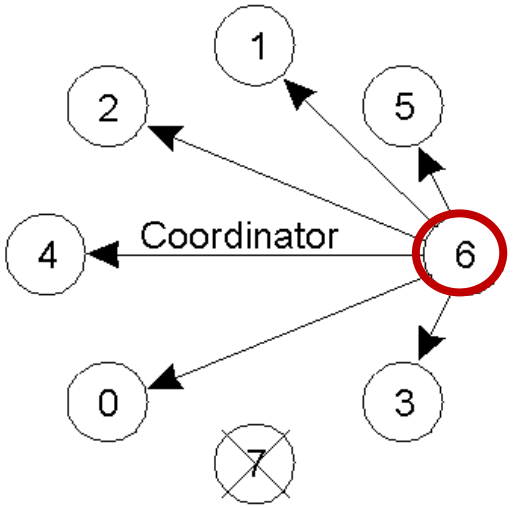
- Both 5 and 6 hold elections
- Send Election messages to those sites with higher # than itself

Bully Algorithm – Example [2/2]



(d)

- Site 6 tells site 5 that it will take over



(e)

- Site 6 announces the takeover by sending a Coordinator message to all sites

* When the original coordinator (ie. 7) comes back on-line, it simply sends out a COORDINATOR message, as it is the highest numbered process (and it knows it.)
* Site 7 takes over the coordinator position after receiving Oks from all sites.

Chapter 19: Distributed Databases

- 19.1 Heterogeneous and Homogeneous Databases
- 19.2 Distributed Data Storage
- 19.3 Distributed Transactions
- 19.4 Commit Protocols
- 19.5 Concurrency Control in Distributed Databases
- 19.6 Availability
- 19.7 Distributed Query Processing
- 19.8 Heterogeneous Distributed Databases
- 19.9 Cloud-Based Databases
- 19.10 Directory Systems

Distributed Query Processing

- For centralized systems, the primary criterion for measuring the cost of a particular strategy is the number of disk accesses
- In a distributed system, other issues must be taken into account:
  - The cost of a data transmission over the network
  - The potential gain in performance from having several sites process parts of the query in parallel

- Primary Concerns
  - Transforming queries on fragments
  - Reducing data transmission for joining relations in distributed sites (semijoin)
  - Parallel inter-join strategy exploiting distributed sites

Query Transformation in Distributed DB

- Translating algebraic queries on fragments
  - It must be possible to construct relation *r* from its fragments
  - Replace relation *r* by the expression to construct relation *r* from its fragments
- Consider the horizontal fragmentation of the *account* relation into

  $account_1 = \sigma_{branch\_name = \text{“Hillside”}}(account)$

  $account_2 = \sigma_{branch\_name = \text{“Valleyview”}}(account)$

  The query $\sigma_{branch\_name = \text{“Hillside”}}(account)$ becomes

  $\sigma_{branch\_name = \text{“Hillside”}}(account_1 \cup account_2)$ which is again optimized into

  $\sigma_{branch\_name = \text{“Hillside”}}(account_1) \cup \sigma_{branch\_name = \text{“Hillside”}}(account_2)$
  - Since $account_1$ has only tuples pertaining to the Hillside branch, we can eliminate the selection operation
  - Apply the definition of $account_2$ to obtain

    $\sigma_{branch\_name = \text{“Hillside”}}(\sigma_{branch\_name = \text{“Valleyview”}}(account))$
  - This expression is the empty set regardless of the contents of the *account* relation
- Final strategy is for the Hillside site to return $account_1$ as the result of the query

Simple Join Processing

- Consider the following relational algebra expression in which the three relations are neither replicated nor fragmented

  $account \bowtie depositor \bowtie branch$

  suppose *account* is stored at site S_1 & *depositor* at S_2 & *branch* at S_3
- For a query issued at site S_I, the system needs to produce the result at site S_I
- Strategy 1: Ship copies of all three relations to site S_I and choose a strategy for processing the entire locally at site S_I
- Strategy 2:
  - Ship a copy of the account relation to site S_2 and compute $temp_1 = account \bowtie depositor$ at S_2
  - Ship $temp_1$ from S_2 to S_3, and compute $temp_2 = temp_1 \bowtie branch$ at S_3
  - Ship the result $temp_2$ to S_I
  - Devise similar strategies, exchanging the roles S_1, S_2, S_3
- Must consider following factors:
  - amount of data being shipped
  - cost of transmitting a data block between sites
  - relative processing speed at each site

Semijoin Strategy

- Let r_1 be a relation with schema R_1 stores at site S_1

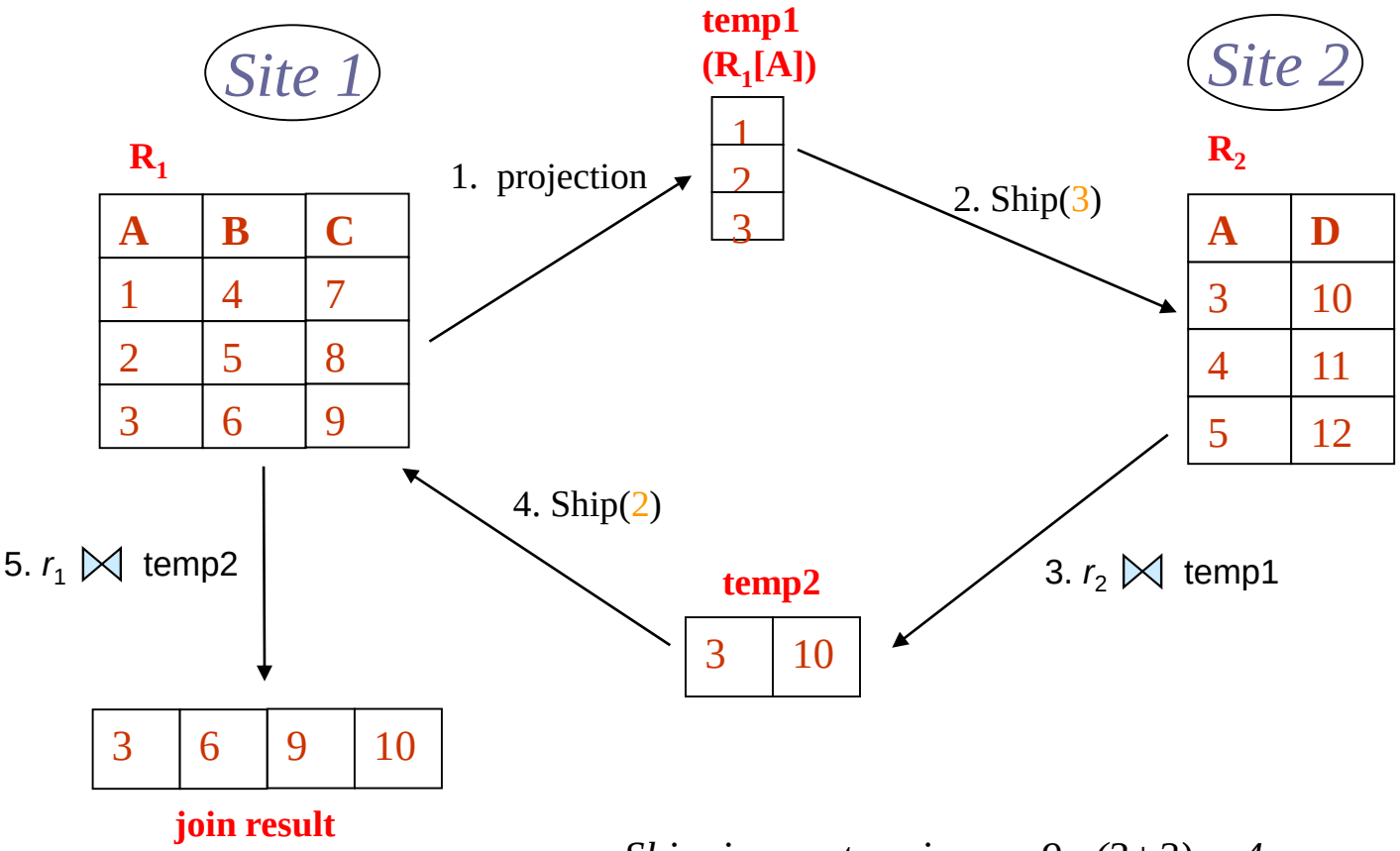
  Let r_2 be a relation with schema R_2 stores at site S_2
- Evaluate the expression $r_1 \bowtie r_2$ and obtain the result at S_1

  1. Compute $temp_1 \leftarrow \prod_{R1 \cap R2}$ (r1) at $S1$
  2. Ship $temp_1$ from S_1 to S_2
  3. Compute $temp_2 \leftarrow r_2 \bowtie$ temp1 at S_2
  4. Ship $temp_2$ from S_2 to S_1
  5. Compute $r_1 \bowtie temp_2$ at S_1. This is the same as $r_1 \bowtie r_2$

- The **semijoin** of r_1 with r_2, is denoted by: $r_1 \ltimes r_2 = \prod_{R1} (r_1 \bowtie r_2)$
- Thus, $r_1 \ltimes r_2$ selects those tuples of r_1 that contributed to $r_1 \bowtie r_2$
- In step 3 above, $temp_2 = r_2 \bowtie r_1$
- For joins of several relations, the above strategy can be extended to a series of semijoin steps

Semi Join – Example

$r_1 \bowtie r_2$



Shipping cost savings = 9 –(3+2) = 4

Join Strategies that Exploit Parallelism

- Consider $r_1 \bowtie r_2 \bowtie r_3 \bowtie r_4$ where relation r_i is stored at site S_i

  The result must be presented at site S_1

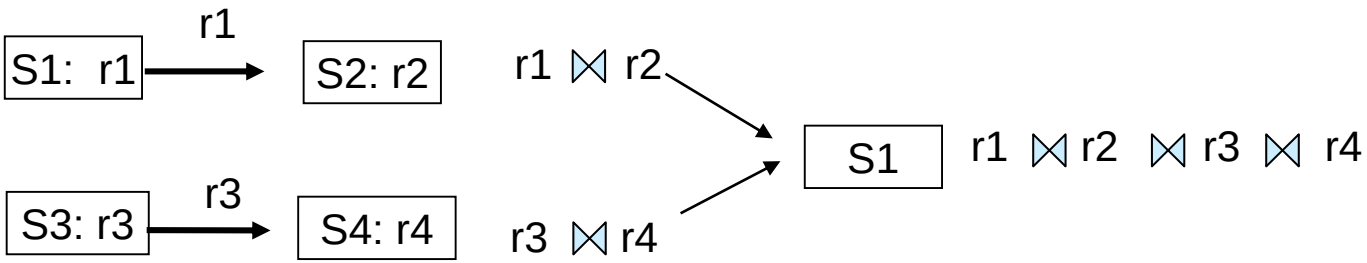
- r_1 is shipped to S_2 and $r_1 \bowtie r_2$ is computed at S_2

  Simultaneously r_3 is shipped to S_4 and $r_3 \bowtie r_4$ is computed at S_4

- S_2 ships tuples of $(r_1 \bowtie r_2)$ to S_1 as they produced;
  S_4 ships tuples of $(r_3 \bowtie r_4)$ to S_1

- Once tuples of $(r_1 \bowtie r_2)$ and $(r_3 \bowtie r_4)$ arrive at S_1:

  $(r_1 \bowtie r_2) \bowtie (r_3 \bowtie r_4)$ is computed in parallel with the computation of $(r_1 \bowtie r_2)$ at S_2 and the computation of $(r_3 \bowtie r_4)$ at S_4

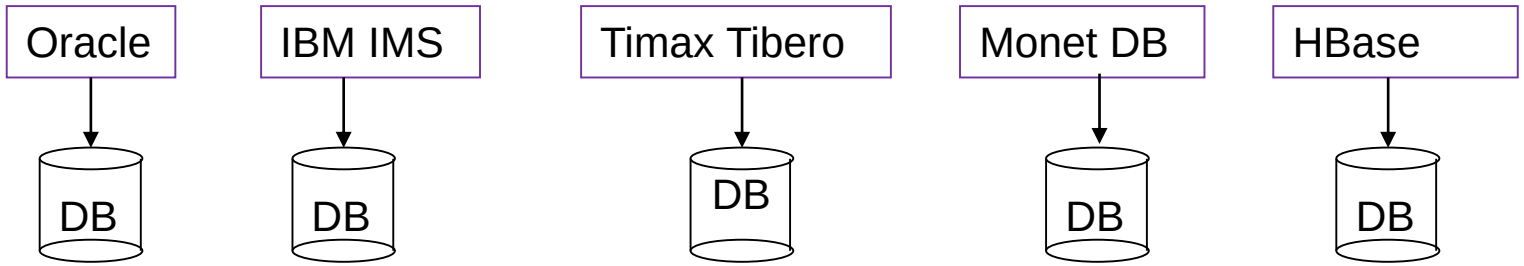


Chapter 19: Distributed Databases

- 19.1 Heterogeneous and Homogeneous Databases
- 19.2 Distributed Data Storage
- 19.3 Distributed Transactions
- 19.4 Commit Protocols
- 19.5 Concurrency Control in Distributed Databases
- 19.6 Availability
- 19.7 Distributed Query Processing
- 19.8 Heterogeneous Distributed Databases
- 19.9 Cloud-Based Databases
- 19.10 Directory Systems

Heterogeneous Distributed Databases

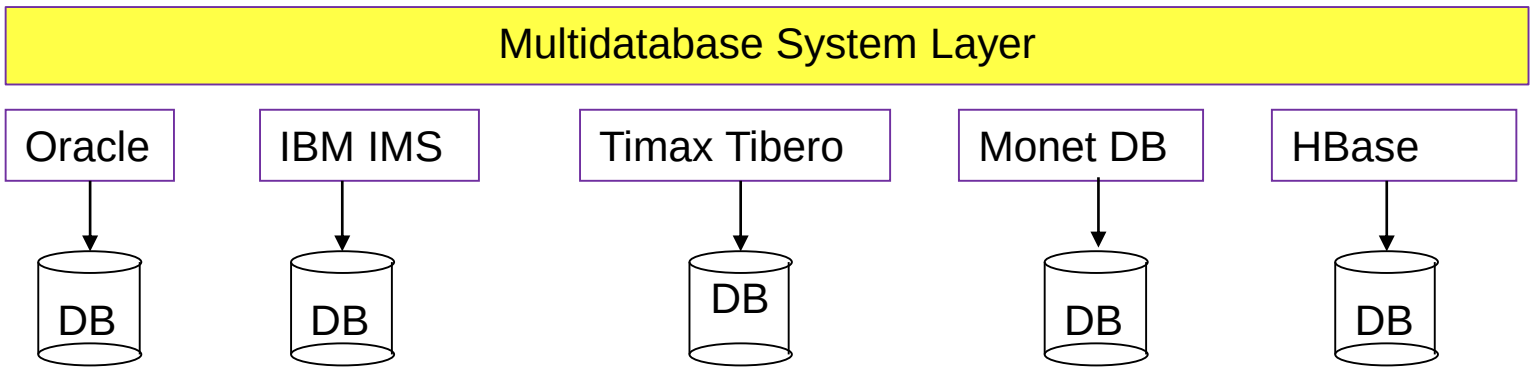
- Many database applications require data from a variety of preexisting databases located in a heterogeneous collection of hardware and software platforms
  - Data models may differ (hierarchical, relational , etc.)
  - CC schemes and transaction commit protocols may be incompatible
  - System-level details are totally incompatible
- Advantages of Heterogeneous Distributed Databases
  - Investment saving in existing hardware/system software/applications
  - Allows use of additional special-purpose DBMSs
- Difficulties towards a unified homogeneous DBMS
    - Technical difficulties and cost of conversion
    - Organizations do not want to give up control on their data
    - Local databases wish to retain a great deal of **autonomy**



Multidatabase System

- Multidatabase System
  - a software layer on top of existing database systems, which is designed to manipulate information in heterogeneous databases
  - creates an illusion of logical database integration without any physical database integration

- Issues of Multidatabase System
  - Unified View of Data
  - Query Processing
  - Tranasaction Management



Unified View of Data in Heterogeneous DB

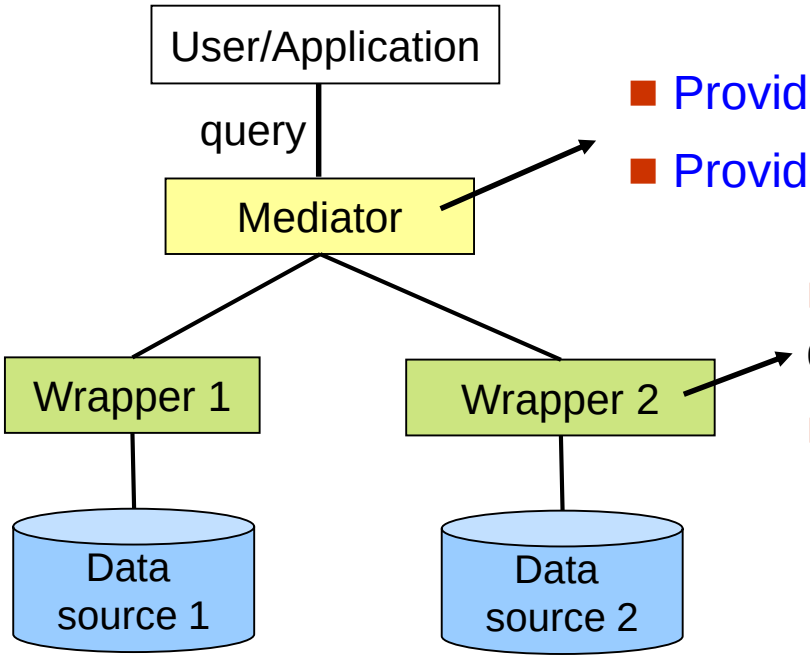
- Agreement on a common data model
  - Typically the relational model
- Agreement on a common conceptual schema
  - Different names for same relation/attribute
  - Same relation/attribute name means different things
- Agreement on a single representation of shared data
  - E.g. data types, precision, character sets (ASCII vs EBCDIC), sort order
- Agreement on units of measure
- Variations in names
  - E.g. Köln vs Cologne, Mumbai vs Bombay

Query Processing in Heterogeneous DB

- Several issues in query processing in a heterogeneous database
- Schema translation
  - Write a **wrapper** for each data source to translate data to a global schema
  - Wrappers must also translate updates on global schema to updates on local schema
- Limited query capabilities
  - Some data sources allow only restricted forms of selections
    - E.g. web forms, flat file data sources
  - Queries have to be broken up and processed partly at the source and partly at a site issuing the query
- Removal of duplicate information when sites have overlapping information
  - Query: find accounts with 50 < balance < 100
  - Site1: accounts with balance > 50, Site2: accounts with balance < 100
- Global query optimization

Mediator Systems in Heterogeneous DB

- **Mediator** systems are systems that integrate multiple heterogeneous data sources by providing an integrated global view, and providing query facilities on global view
  - Unlike full fledged multidatabase systems, mediators generally do not bother about transaction processing
  - But the terms mediator and multidatabase are sometimes used interchangeably
  - The term **virtual database** is also used to refer to mediator/multidatabase systems

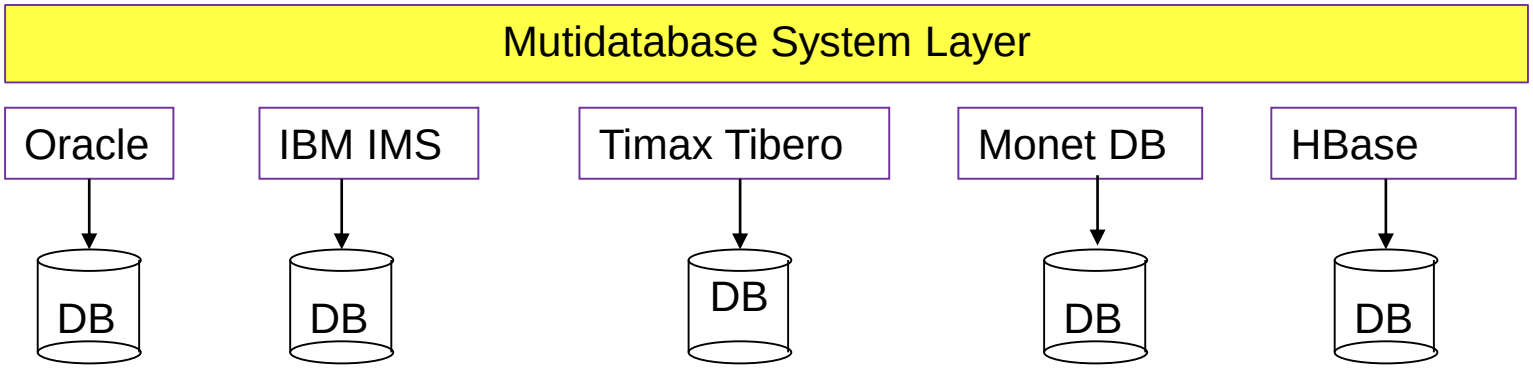


- Provide an integrated global view of the data
- Provide query facilities on the global view

- Translate queries on the global schema into queries on the local schema
- Translate results back into the global schema

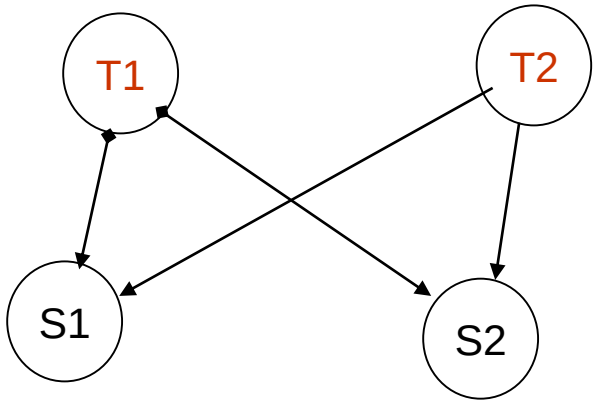
Transaction Management in Multidatabases

- **Local transactions** are executed by each local DBMS, outside of the MDBS system control.
- **Global transactions** are executed under the multidatabase control.
- **Local autonomy** - local DBMSs cannot communicate directly to synchronize global transaction execution and the multidatabase has no control over local transaction execution
  - Local concurrency control scheme needed to ensure that local DBMS's schedule is serializable
  - Local DBMS must be able to guard against local deadlocks
  - Additional mechanisms are needed to ensure global serializability



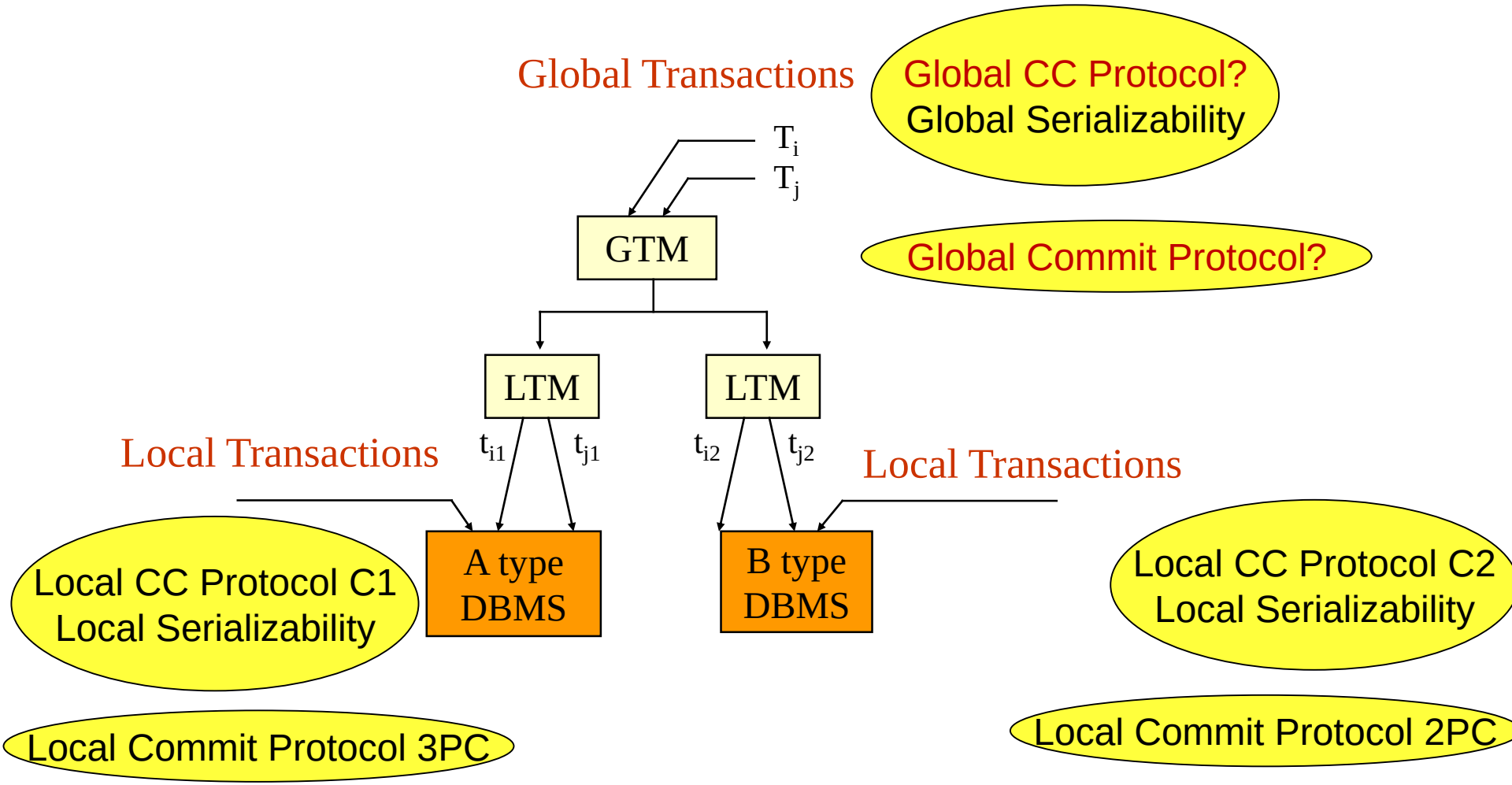
Local vs. Global Serializability

- The guarantee of local serializability is not sufficient to ensure global serializability
  - As an illustration, consider two global transactions T1 and T2 , each of which accesses and updates two data items, A and B, located at sites S1 and S2 respectively
  - It is possible to have a situation where, at site S1 , T2 follows T1 , whereas, at S2 , T1 follows T2, resulting in a nonserializable global schedule.
- If the local systems permit control of locking behavior and all systems follow 2PL
  - the multidatabase system can ensure that global transactions lock in a two-phase manner
  - the lock points of conflicting transactions would then define their global serialization order



Transaction Management in Multidatabase Systems – Example

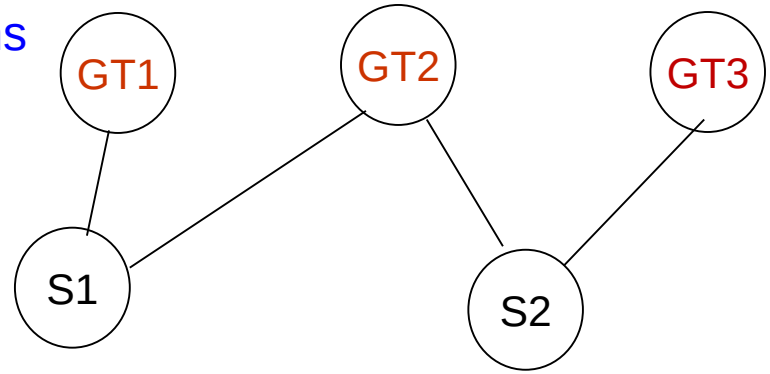
GTM : Global Transaction Manager
LTM : Local Transaction Manager



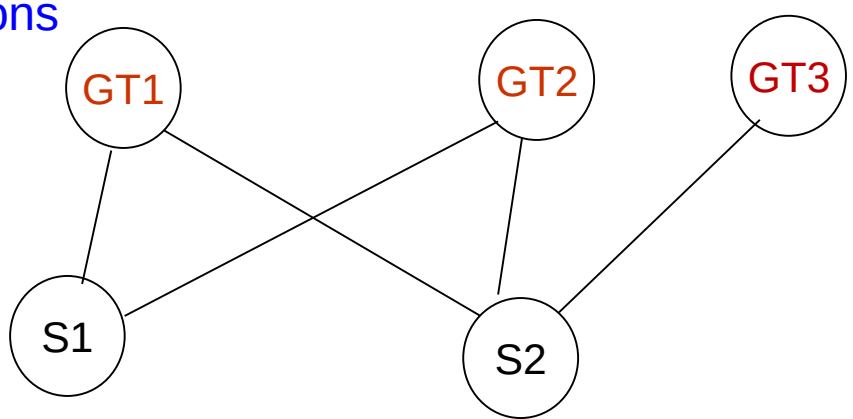
Global Serializability in Multidatabase

- Even if no information is available concerning the structure of the various concurrency control schemes, a very restrictive protocol that ensures serializability is available
- Transaction-graph : a graph with vertices being global transaction names and site names
  - An undirected edge (T_i, S_k) exists if T_i is active at site S_k
  - Global serializability is assured if transaction-graph contains no undirected cycles

- Global serializable transactions



- Global non-serializable transactions

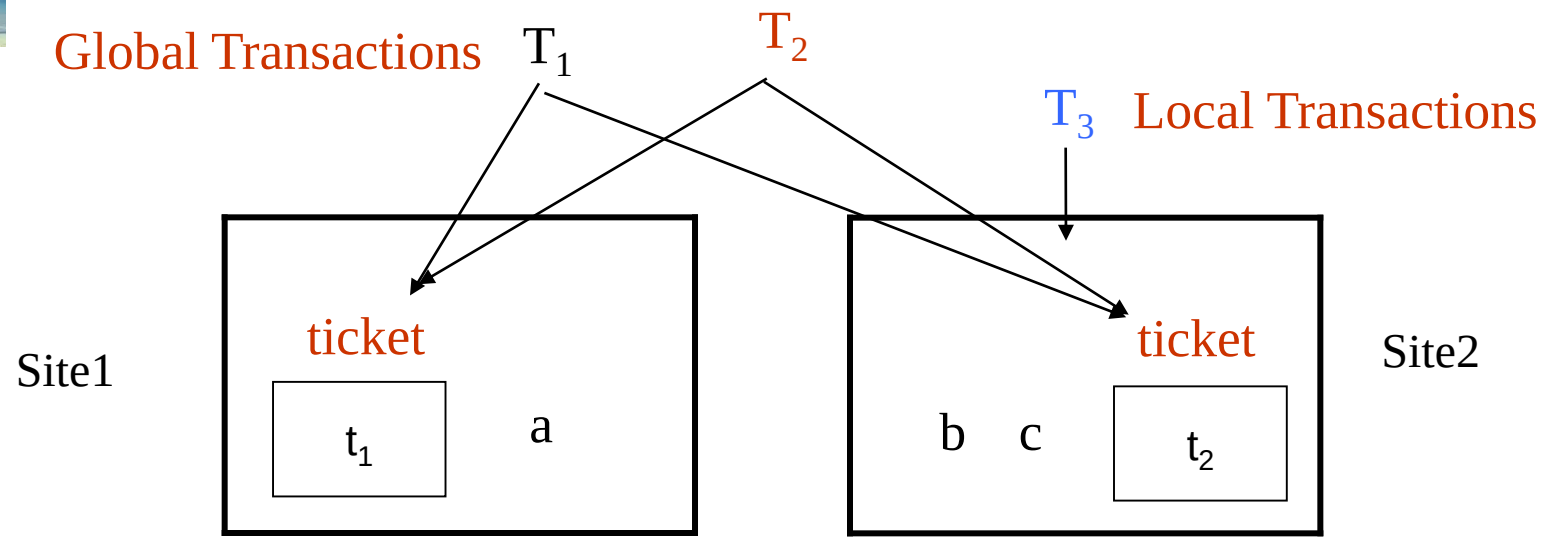


GT1과 GT2가 S1과 S2에서 동시에 active하면 globally non-serializble한 schedule을 발생할수 있다

Ensuring Global Serializability in Multidatabase

- The ticket CC scheme
  - Each site S_i has a special data item, called **ticket**
  - Every global transaction T_i that runs at site S_k writes to the ticket at site S_i
- The ticket CC scheme ensures global transactions are serialized at each site, regardless of local concurrency control method, so long as the method guarantees local serializability
- Global transaction manager decides a serial ordering of global transactions by controlling order in which tickets are accessed
- However, the above protocol results in low concurrency between global transactions

Ticket Protocol Example for Global Serializability



Server 1 : $r_1(a)\ c_1$ $w_2(a)c_2$

Server 2 : $r_3(b)$ $w_1(b)\ c_1$ $r_2(c)\ c_2$ $w_3(c)\ c_3$

Not good enough

Server 1 : $r_1(t_1)w_1(t_1{+}{+})r_1(a)c_1$ $r_2(t_1)w_2(t_1{+}{+})\ w_2(a)c_2$

Server 2 : $r_3(b)$ $r_1(t_2)w_1(t_2{+}{+})w_1(b)c_1$ $r_2(t_2)w_2(t_2{+}{+})r_2(c)\ c_2$ $w_3(c)\ c_3$

Global transaction T1 과 T2는 site1에서 ticket t1을 write하면서, site2에서는 ticket t2를 write하면서 순서가 정해진다. 모든 global transaction의 read와 write앞에 read(ticket)write(ticket++)를 붙여서 Write conflict를 유발시켜서 global order를 설정
T1과 T2가 같은 site에서 동시에 active할수가 없게 만드는것!!!

With the ticket CC, the transaction graph with sites and global transactions never generate a cycle

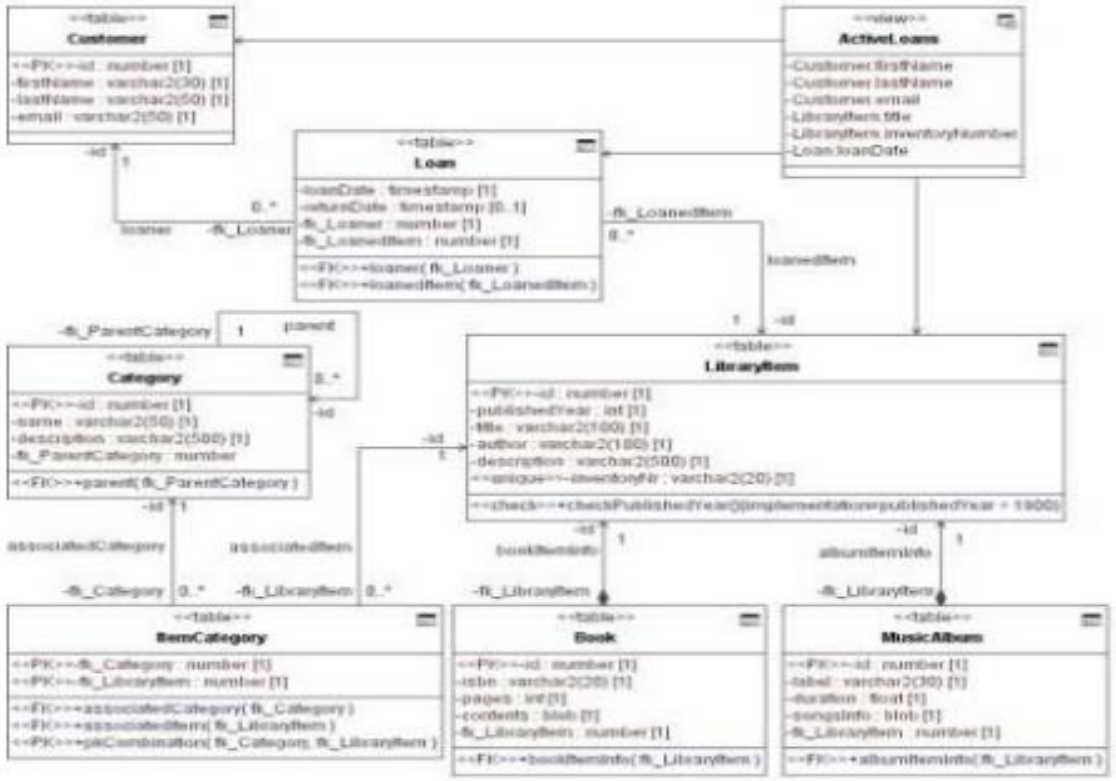
Chapter 19: Distributed Databases

- 19.1 Heterogeneous and Homogeneous Databases
- 19.2 Distributed Data Storage
- 19.3 Distributed Transactions
- 19.4 Commit Protocols
- 19.5 Concurrency Control in Distributed Databases
- 19.6 Availability
- 19.7 Distributed Query Processing
- 19.8 Heterogeneous Distributed Databases
- 19.9 Cloud-Based Databases
- 19.10 Directory Systems

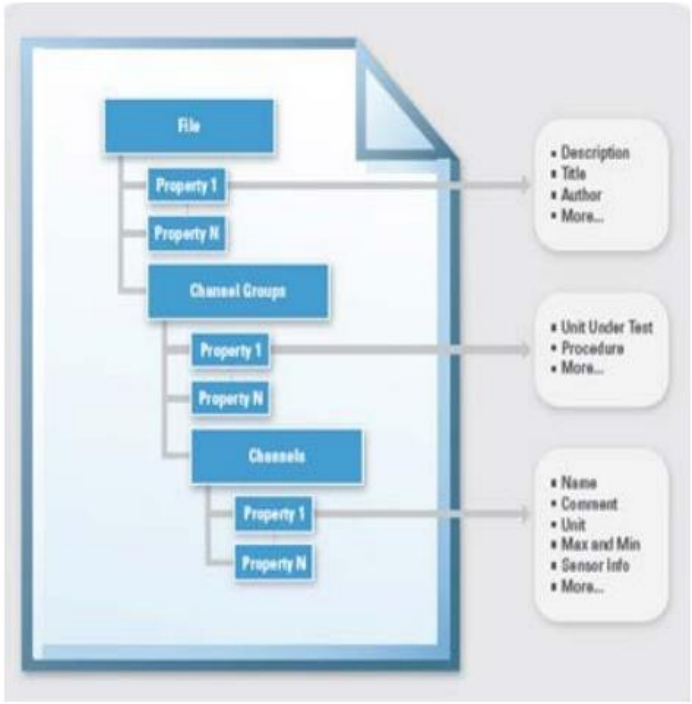
Data Storage on the Cloud

- Need to store and retrieve massive amounts of data
- Traditional parallel databases not designed to scale to 1000's of nodes (and expensive)
- Initial needs did not include full database functionality
  - Store and retrieve data items by key value is minimum functionality
    - **Key-value stores**
- Several implementations of Cloud Data Storage System
  - Google Cloud Platform: a cloud computing service by Google
    - Bigtable is a compressed, high performance, and proprietary data storage system built on Google File System (GFS) which is a distributed fault-tolerant file system
    - HBase, an open source clone of Bigtable
  - Sherpa/PNUTS: Yahoo's Cloud Storage Platform
  - Simple Storage Service (S3) of Amazon provides web interface to Dynamo
    - Dynamo, which is a key-value storage system from Amazon
  - Cassandra, data storage system from FaceBook
  - Azure environment of Microsoft has a data storage component

Data Model in the Cloud



User Profile Data



Document

Relational data model cannot handle the above complex data

Web Scale Computing

- Global distribution of data and servers
- Unpredictable Growth: Scalability
- Evolving applications: Flexibility
- High Availability: 24 hours / 7 days
- Consistency is less critical
- Simpler Transactions
- BASE Properties (compared to ACID properties of RDBMS)
  - Basically available
  - Soft state
  - Eventually consistent

- Traditional RDBMS cannot effectively provide the scalability, availability, elasticity, latency required for Web scale services
  - Rigid ACID transactions
  - Complex data & Query model
  - Difficult configuration/maintenance

ACID vs BASE

ACID vs. BASE

Inktomi

ACID	BASE
Strong consistency	Weak consistency
Isolation	– stale data OK
Focus on "commit"	Availability first
Nested transactions	Best effort
Availability?	Approximate answers OK
Conservative (pessimistic)	Aggressive (optimistic)
Difficult evolution (e.g. schema)	Simpler!
	Faster
	Easier evolution

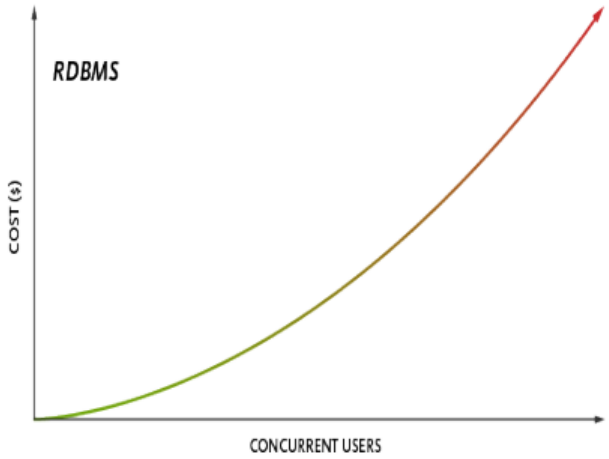
But I *think* it's a *spectrum*

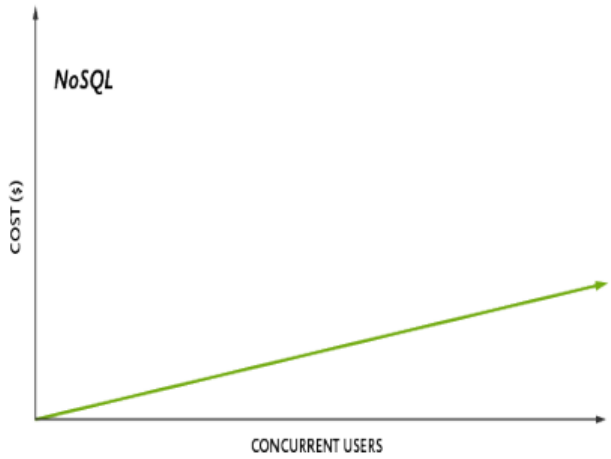
PODC Keynote, July 19, 2000

NoSQL DBMS

- Not Only SQL
- Simple data model & Simple queries
- A number of features exists that NoSQL DBMS share in common
  - Non-relational
  - Dynamic Schema
  - Support for scalability and distributed system
  - High performance
  - Handle large-volume of data

Comparison Cost/Users Between RDBMS and NoSQL





Key Value Stores

- Key-value stores support
  - put(key, value): used to store values with an associated key,
  - get(key): which retrieves the stored value associated with the specified key
- Some systems such as Bigtable additionally provide range queries on key values
- Multiple versions of data may be stored, by adding a timestamp to the key

Examples:	Key	Value
Directory	Company: Algo-Logic	Phone #: (408) 707-3740
Forwarding Tables	IP Address: 204.2.34.5	Interface : MAC Address: Eth6 : 02:33:29:F2:AB:CC
Data De-duplication	Content Hash: XYZ	Storage Block ID: 948830038411
Stock Trading	Order ID: ATY11217911101	Symbol, Side, Price: AAPL, B, 126.75
Graph Search	Virtex: v140	Edge List: v201, v206, v225

Data Representation of Key-Value

- Records in many big data applications need to have a flexible schema
  - Not all records have same structure
  - Some attributes may have complex substructure
- XML and JSON data representation formats widely used
- An example of a JSON object for key-value pairs is:

```
{
    "ID": "22222",
    "name": {
            "firstname: "Albert",
            "lastname: "Einstein"
    },
    "deptname": "Physics",
    "children": [
            { "firstname": "Hans", "lastname": "Einstein" },
            { "firstname": "Eduard", "lastname": "Einstein" }
    ]
}
```

name.firstname

children[1].firstname
children[2].firstname

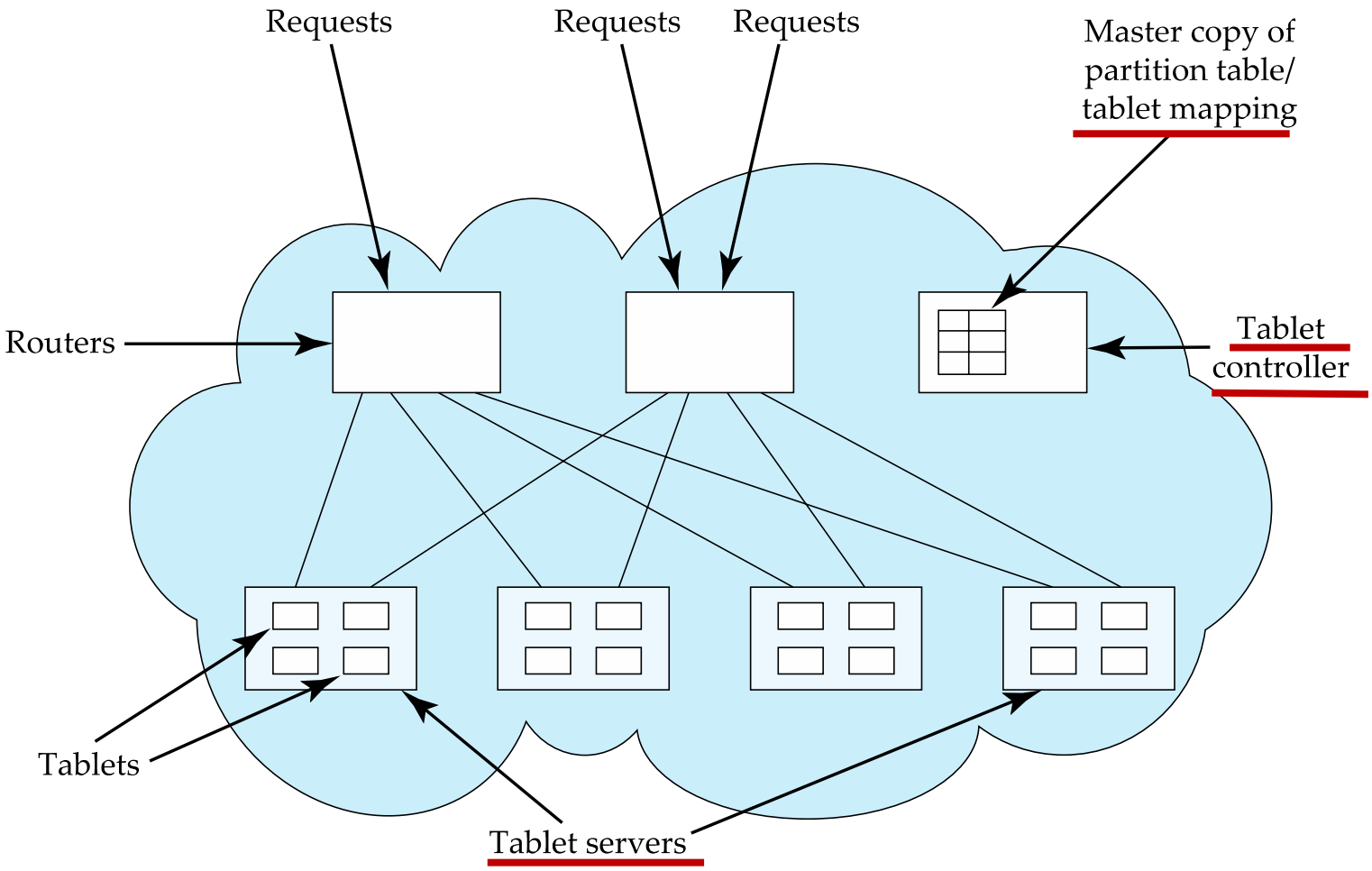
- Key-value pair of web “crawled” page
  - Value → www.cs.yale.edu/people/korth.html
  - Key ➔ edu.yale.cs.www/people/korth.html

Partitioning and Retrieving Data

- Key-value stores partition data into relatively small units (hundreds of megabytes)
- These partitions are often called tablets (a tablet is a fragment of a table)
- Partitioning of data into tablets is rather dynamic: (data가 들어오기전에 알수가 없다)
  - As data are inserted, if a tablet grows too big, it is broken into smaller parts
  - If the load (get/put operations) on a tablet is excessive, the tablet may be broken into smaller tablets, which can be distributed across two or more sites to share the load
  - The number of tablets is much larger than the number of sites
    - Similar to virtual partitioning in parallel databases
  - Range partitioning / Hash Partitioning

- Each get/put request must be routed to the correct site
- **Tablet controller** tracks the partitioning function and tablet-to-site mapping
  - Map a get() request to one or more tablets,
  - Tablet mapping function to track which site responsible for which tablet

Figure 19.7 Cloud Storage System Architecture



Transaction and Replication on Cloud-Based Database

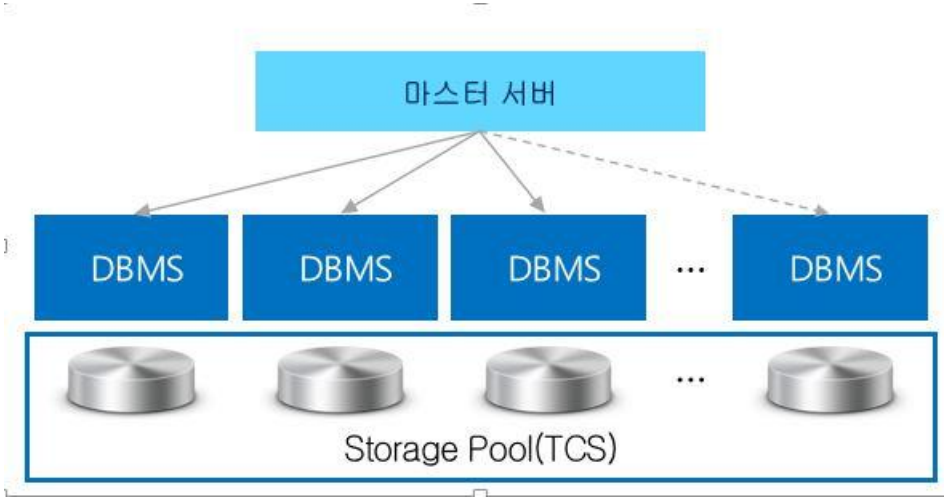
- A data-storage system replicates data (such as tablets) to multiple machines in a cluster
- A copy of the data is likely to be available even if some machines of a cluster are down
- A cluster is a collection of machines

- Transaction inside a tablet
  - Data storage system on the cloud do not support ACID transactions
  - Singe master site가 control하는 single tablet 내부에서만 transaction을 지원
  - Yahoo Sherpa/PNUT: data item에 version을 두고, test-and-set function을 이용하여 validation-based CC로 transaction 지원

- Tablets replication
  - 특정 Tablet은 master site에서만 update되고, replicated된 table들은 read-only에서 이용
  - Master site이 down되면, 다른 master site을 선발하고 가지고 있었던 tablet들을 이동
  - Google BigTable: tablets replication 운영을 GFS를 이용하여 Google Cloud Platform에서 구현
  - Yahoo Sherpa/PNUT: tablets 을 multiple nodes in a cluster 에 복제하고, reliable distributed-messaging system을 이용하여

Traditional Databases on Cloud-Based Database

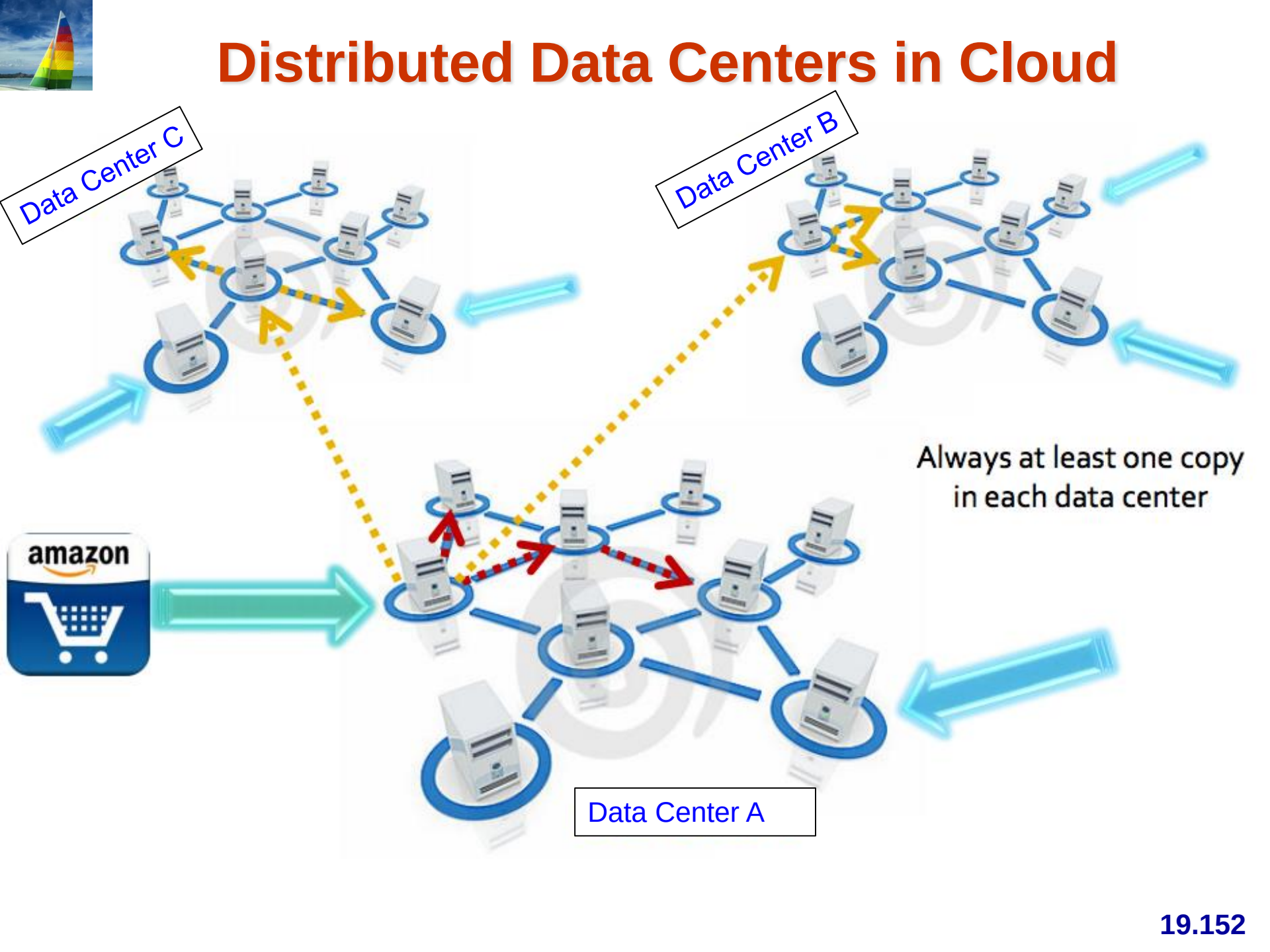
- Virtual machine
  - A user is given a illusion of having a private computer, while in reality a single computer simulated several virtual machines
- Having a set of virtual machines works well for parallelizing DBMSs
- Homogeneopus distributed database system
  - Each virtual machine can run database system locally



Challenges with Cloud-Based Database

- DBMS를 구매하지 않고 Cloud Database를 이용하면 좋은점이 많지만, But!
- Distributed DBMSs on cloud requires frequent communication and coordination among sites
  - Access to data on another physical machine
  - Obtaining locks on remote data
  - Ensuring atomic transaction commit by 2PC
- The above technical problems are affected by the controls of the cloud vendor (not the user)
  - Physical placement of data
  - Data replication
- Legal issues of data on the cloud
  - What if the vendor is replicating German Company's data on a serve in New York
- Various degree of cloud services (선택해야 할 일도 많다)
  - Providing Just raw storage and virtual machines
  - Providing varying degree of control over data placement and replication
  - Providing even DBMS service

Distributed Data Centers in Cloud



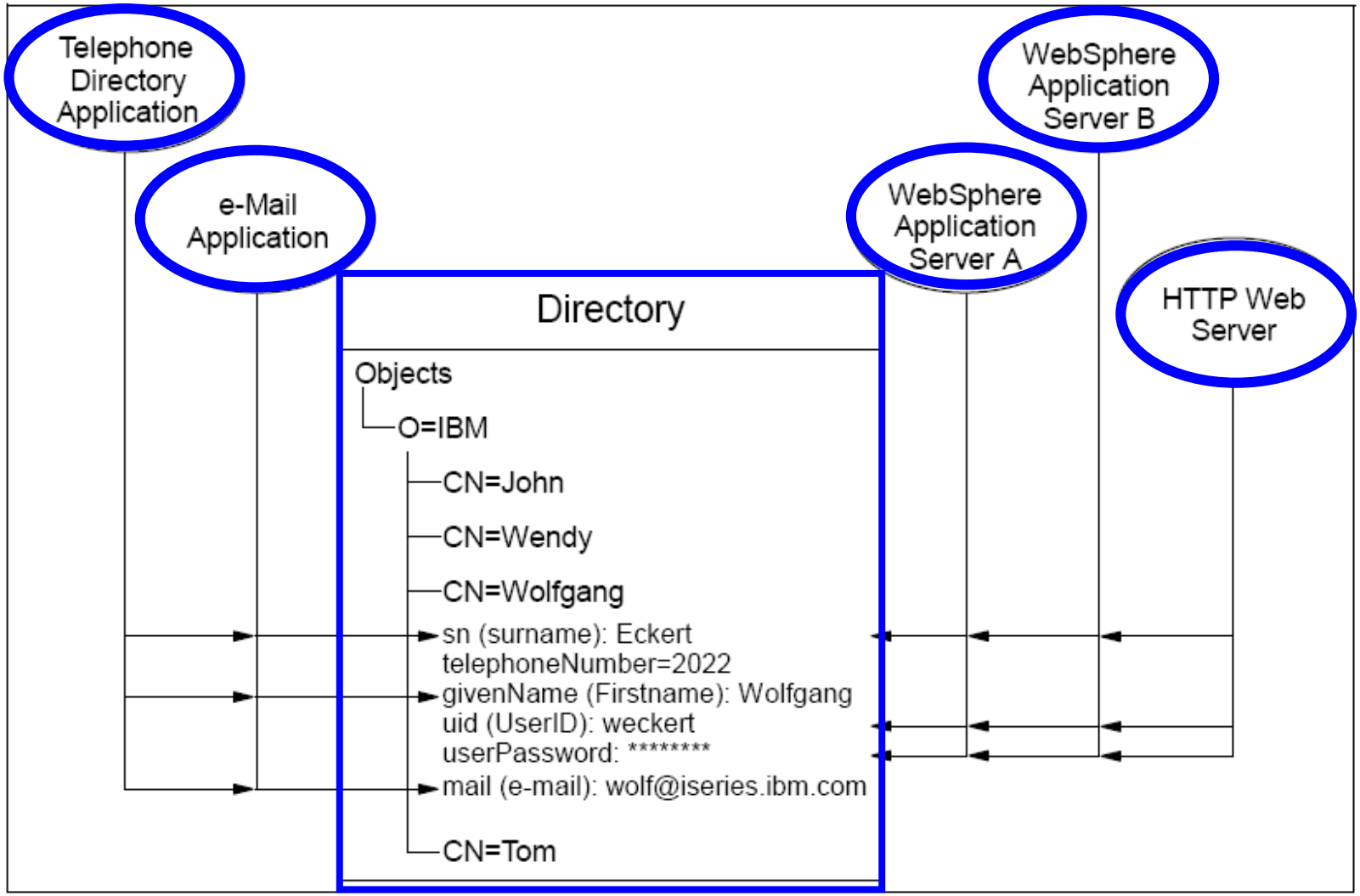
Chapter 19: Distributed Databases

- 19.1 Heterogeneous and Homogeneous Databases
- 19.2 Distributed Data Storage
- 19.3 Distributed Transactions
- 19.4 Commit Protocols
- 19.5 Concurrency Control in Distributed Databases
- 19.6 Availability
- 19.7 Distributed Query Processing
- 19.8 Heterogeneous Distributed Databases
- 19.9 Cloud-Based Databases
- 19.10 Directory Systems

Directory Systems

- Typical kinds of directory information
  - Employee information such as name, id, email, phone, office addr, ..
  - Company information (미국 6백만개의 회사)
  - Even personal information to be accessed from multiple places
    - e.g. Web browser bookmarks
- White pages having entries organized by name or identifier
  - Meant for forward lookup to find more about an entry (such as 진화춘, 중국집)
- Yellow pages having entries organized by properties (such as 중국집, 진화춘)
  - For reverse lookup to find entries matching specific requirements
- When directories are to be accessed across an organization
  - Alternative 1: Web interface implementation
    - Good for human, Not good for accesses from other program
  - Alternative 2: Through specialized **directory access protocols**
    - Coupled with specialized user interfaces
- Why this subsection is in the “Distributed Databases” chapter?
  - Distributed Directory Tree is a kind of distributed database

Directory Systems – Example



Several applications using attributes of the same entry

DBMS vs LDAP

- DBMS위에 Web Interface로 Directory를 만들면 되지?
  - Why LDAP? Why not use database protocols like ODBC/JDBC?
- Answer:
  - Simplified protocols for a limited type of data access
    - ODBC/JDBC is rather complicated to use
  - Can be optimized to economically provide more applications with rapid access to directory data in large distributed environments
    - Because directories are not intended to provide as many functions as general-purpose relational databases
  - Provide a nice hierarchical naming mechanism similar to file system directories
    - Data can be partitioned amongst multiple servers for different parts of the hierarchy, yet give a single view to user
      - E.g. different servers for Bell Labs Murray Hill and Bell Labs Bangalore
  - Directories may use databases as storage mechanism

Directory Access Protocols

- Most commonly used directory access protocol:
  - LDAP (Lightweight Directory Access Protocol)
  - Simplified from earlier X.500 protocol

- LDAP
  - LDAP Data Model
  - Data Manipulation
  - Distributed Directory Trees

- LDAP SWs
  - Apache Directory Server/Studio
  - MicroSoft Active Directory Explorer
  - IBM Tivoli Directory Server

LDAP Data Model [1/2]

- LDAP directories store **entries**
  - Entries are similar to objects
- Each entry must have unique **distinguished name (DN)**
- DN made up of a sequence of **relative distinguished names (RDNs)**
- Example of a DN
  - cn = Silberschatz, ou = Bell Labs, o = Lucent, c = USA
  - Standard RDNs (can be specified as part of schema)
    - cn: common name
    - ou: organizational unit
    - o: organization
    - c: country
  - Similar to paths in a file system but written in reverse direction

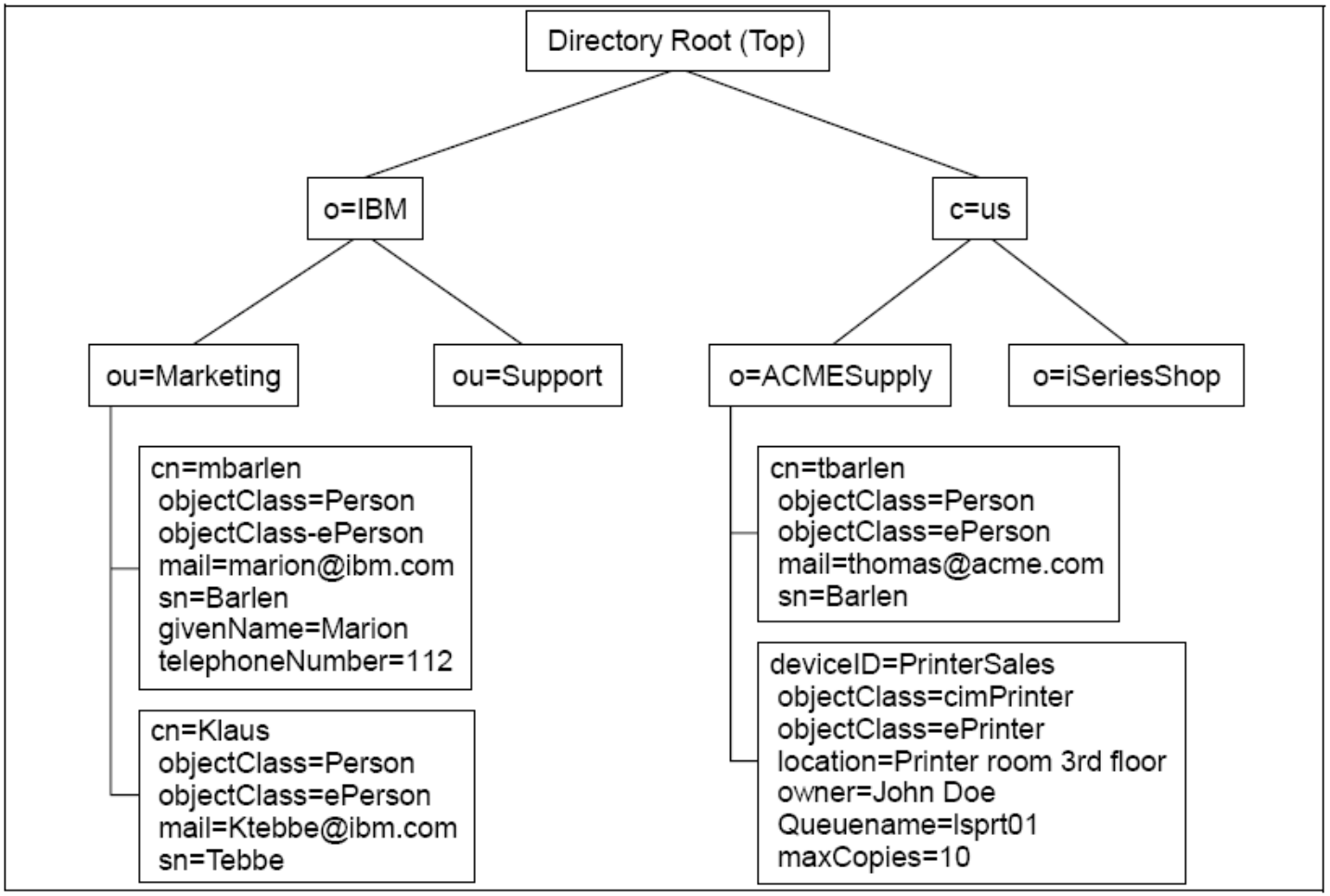
LDAP Data Model [2/2]

- Entries can have attributes
  - Attributes are multi-valued by default
  - LDAP has several built-in types
    - Binary type, String type, Time type
    - Tel type: telephone number PostalAddress type: postal address
- LDAP allows definition of **object classes**
  - Object classes specify attribute names and types
  - Can use inheritance to define object classes
  - Entry can be specified to be of one or more object classes
    - No need to have single most-specific type

Directory Information Tree (DIT)

- Entries organized into a **directory information tree** (DIT) according to their DNs
  - Leaf level usually represent specific objects
  - Internal node entries represent objects such as organizational units, organizations or countries
  - Children of a node inherit the DN of the parent, and add on RDNs
    - E.g. internal node with DN c=USA
      - Children nodes have DN starting with c=USA and further RDNs such as o or ou
    - DN of an entry can be generated by traversing path from root
  - Leaf level can be an alias pointing to another entry
    - Entries can thus have more than one DN
      - E.g. person in more than one organizational unit

Directory Information Tree – Example



- Standard RDNs (can be specified as part of schema)
- cn: common name
- ou: organizational unit
- o: organization
- c: country

Application programs can connect to DIT via LDAP Query, LDIF, URL and API

LDAP Data Manipulation

- Unlike SQL, LDAP does not define DDL or DML
- Instead, it defines a network protocol for DDL and DML
  - Users use an API or vendor specific front ends
  - LDAP also defines a file format
    - LDAP Data Interchange Format (LDIF)

```
# e.g., adding single attribute
# The add directive follows a changetype: modify directive and
# defines the name of the attribute(s) to be added to an existing entry.

dn: cn=Robert Smith,ou=people,dc=example,dc=com
changetype: modify
add: telephonenumber
telephonenumber: 123-111
```

- Querying mechanism is very simple: only selection & projection

LDAP Queries

- LDAP query specifies only selection & projection
- LDAP query must specify
  - Base: a node in the DIT from where search is to start
  - A search condition
    - Boolean combination of conditions on attributes of entries
      - Equality, wild-cards and approximate equality supported
  - A scope
    - Just the base, the base and its children, or the entire subtree from the base
  - Attributes to be returned
  - Limits on number of results and on resource consumption
  - May also specify whether to automatically dereference aliases
- LDAP URLs are one way of specifying query
- LDAP API is another alternative

LDAP Query by LDAP URLs

- First part of LDAP URL specifies server and second part is DN of base
  - ldap://aura.research.bell-labs.com/o=Lucent,c=USA
- Optional further parts separated by ? symbol
  - ldap://aura.research.bell-labs.com/o=Lucent,c=USA??sub?cn=Korth
  - Optional parts specify
    1. ? ➔ attributes to return (empty means all)
    2. ?sub ➔ Scope (sub indicates entire subtree)
    3. ?cn=korth ➔ Search condition (cn=Korth)

LDAP Query by LDAP C-API [1/2]

- LDAP API also has functions to create, update and delete entries
- Each function call behaves as a separate transaction
  - LDAP does not support atomicity of updates

```
#include <stdio.h>
#include <ldap.h>
main( ) {
  LDAP *ld;
  LDAPMessage *res, *entry;
  char *dn, *attr, *attrList [ ] = {"telephoneNumber", NULL};
  BerElement *ptr;
  int vals, i;

  // Open a connection to server
  ld = ldap_open("aura.research.bell-labs.com", LDAP_PORT);
  ldap_simple_bind(ld, "avi", "avi-passwd");
  ... actual query (next slide) ...
  ldap_unbind(ld);
}
```

C Code using LDAP API [2/2]

```
ldap_search_s(ld, "o=Lucent, c=USA", LDAP_SCOPE_SUBTREE,
                        "cn=Korth",    attrList,    /* attrsonly */ 0,    &res);
          /*attrsonly = 1 ➔ return only schema not actual results*/

printf("found%d entries", ldap_count_entries(ld, res));
for (entry = ldap_first_entry(ld, res); entry != NULL;
                        entry = ldap_next_entry(id, entry)) {
        dn = ldap_get_dn(ld, entry);
        printf("dn: %s", dn);   /* dn: DN of matching entry */
        ldap_memfree(dn);


      for(attr = ldap_first_attribute(ld, entry, &ptr); attr != NULL;
                        attr = ldap_next_attribute(ld, entry, ptr))     // for each attribute
        {  printf("%s:", attr);                  // print name of attribute
           vals = ldap_get_values(ld, entry, attr);
           for (i = 0; vals[i] != NULL; i ++)
                        printf("%s", vals[i]);  // since attrs can be multivalued
           ldap_value_free(vals);
        }
}
ldap_msgfree(res);
```

Distributed Directory Information Trees

- Organizational information may be split into multiple directory information trees (DITs)
  - Suffix of a DIT gives RDN to be tagged onto to all entries to get an overall DN
    - E.g. two DITs, one with suffix o=Lucent, c=USA
      and another with suffix o=Lucent, c=India
  - Organizations often split up DITs based on geographical location or by organizational structure
  - Many LDAP implementations support replication (master-slave or multi-master replication) of DITs (not part of LDAP 3 standard)
- A node in a DIT may be a **referral** to a node in another DIT
  - E.g. "ou= Bell Labs" may have a separate DIT, and DIT for "o=Lucent" may have a leaf with "ou=Bell Labs" containing a referral to the Bell Labs DIT
  - Referrals are the key to integrating a distributed collection of directories
  - When a server gets a query reaching a referral node, it may either
    - Forward query to referred DIT and return answer to client, or
    - Give referral back to client, which transparently sends query to referred DIT (without user intervention)

Distributed Directory Tree – Example

E.g., Suppose Frank in Bell Labs, USA is transferred to India for some time.