Tutorial 3
Distributed Transactions Management

THE UNIVERSITY OF QUEENSLAND
AUSTRALIA

+ Q1 Distributed Transaction

Olympic Game Database

Event(EventID, ***)

Athlete(CompID, ***)

1. **Results**(EventID, CompID, Position)
2. **Medals**(EventID, Medal, CompID) //Medal: Gold/Silver/Bronze
3. **MedalTally**(Country, Medal, Number)
4. **Competitors**(CompID, Country)
5. **Medalists**(CompID, NMedals) // NMedals: total of medals won

- When a race is running, a number of tables need to be updated.
  - Assume that when an event is completed, a file is created at the venue giving CompID and Position associated with that EventID, then a series of transactions are executed which update other tables.

THE UNIVERSITY OF QUEENSLAND AUSTRALIA

+ Q1-1

- Write a program to **perform the updates**, using SQL INSERT INTO and UPDATE commands.
- Write the program as a single transaction bounded by BEGIN TRANSACTION and COMMIT/END TRANSACTION statements.

+ Transaction

- Problems for update queries
  - What happens when two queries update the same data item?
  - How to deal with system failure?
- Transaction
  - A *sequence* of read and write *operations*
  - With *computation* steps
  - Termination
    - Commit
    - Abort

```
Begin_transaction Reservation
begin
    input(flight_no, date, customer_name);
    EXEC SQL UPDATE FLIGHT
                SET     STSOLD = STSOLD + 1
                WHERE   FNO = flight_no
                AND     DATE = date;
    EXEC SQL INSERT
                INTO    FC(FNO,DATE,CNAME,SPECIAL)
                VALUES  (flight_no,date,customer_name, null);
    output("reservation completed")
end.
```

+ Transaction Property

- Atomicity
  - All or Nothing
- Consistency
  - Correctness, from one valid state to another
- Isolation
  - Cannot reveal its results to others before commitment
- Durability
  - After commitment, results are permanent

```
BEGIN TRANSACTION
read inputfile EventIdent        // EventIdent(CompIdent, Pos)
while inputfile not empty
        read inputfile CompIdent, Pos        // Tennis(Murray, 1)
        INSERT INTO Results(EventID, CompID, Position)
        VALUES (EventIdent, CompIdent, Pos)
                                        // Results(Tennis Man Single, Murray, 1)
        if Pos < 4 then
                UPDATE Medalists(CompID, NMedals)        // Medalists(Murray, 1)
                SET NMedals = NMedals + 1
                        WHERE CompID = CompIdent

                if Pos = 1 then
                        MedalAwarded = "Gold"
                else if Pos = 2 then
                        MedalAwarded = "Silver"
                else
                        MedalAwarded = "Bronze"        // Medals (Tennis,Gold, Murray)

                INSERT INTO Medals (EventID, Medal, CompID)
                VALUES (EventIdent, MedalAwarded, CompIdent)

                UPDATE MedalTally(Country, Medal, Number)
                SET Number = Number + 1
                WHERE MedalTally.Country =
                        (SELECT Country FROM Competitors
                                WHERE CompID = CompIdent)
                        AND Medal = MedalAwarded
COMMIT
END TRANSACTION                                 // MedalTally(GBR, Gold, 12)
```

Tennis Man’s single	
Andy Murray	1
Juan Martin Del Potro	2
KEI NISHIKORI	3
RAFAEL NADAL	4

+ Q 1-2 Issues

1. **INSERT INTO Results**(EventID, CompID, Position)
2. **UPDATE Medalists**(CompID, NMedals)
3. **INSERT INTO Medals** (EventID, Medal, CompID)
4. **UPDATE MedalTally**(Country, Medal, Number)

Add new ones

Update existing ones

Transaction 1: 200m butterfly(Michael Phelps, 1)

Transaction 2: 4*100m freestyle(Michael Phelps, 1)

+ Q 1-2 Issues

- Q:For each table, discuss whether or not interference with another transaction is possible. In each case where it is possible, give an example of an interfering transaction.

1. No more than one transaction updating the tables whose primary key includes ***EventID***, because each event occurs only once.

   Results(EventID, CompID, Position)

   Medals(EventID, Medal, CompID)

   Results(200m butterfly, Michael Phelps, 1)
   (4*100m freestyle, Michael Phelps, 1)

   Medals(200m butterfly, Gold, Michael Phelps)
   (4*100m freestyle, Gold, Michael Phelps)

2. Two different events can interfere with each other trying to update since different events can complete at about the same time.

   MedalTally(Country, Medal, Number)

   MedalTally(US, Gold, 10)

T1	T2
Read N	
N = 10+1 =11	Read N
	N = 10+1 =11
Write 11	Write 11
	Result should be 12!

3. The same competitor will compete in different events being updated concurrently

   Medalists(CompID, NMedals)

   Medalists(Michael Phelps, N)

THE UNIVERSITY OF QUEENSLAND AUSTRALIA

So we need locks!

+ Q 1-3 Lock

- Q:Annotate your program with *read-lock*, *write-lock* and *unlock* statements.
  1. The notation should make clear the level of granularity of the locking.
  2. In each case indicates whether the granularity of locking is more than strictly necessary.
  3. Justify your decision based on the characteristics of the application.

+ Q1-3 Lock

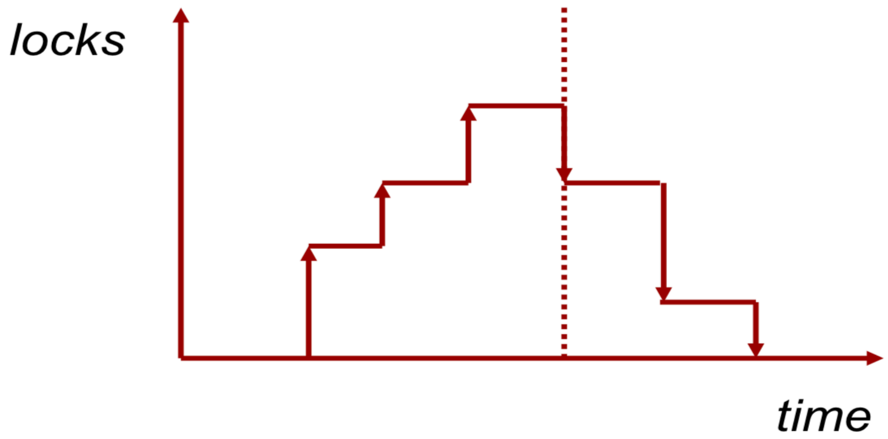

```
BEGIN TRANSACTION
    WRITE LOCK TABLE Results        //not strictly necessary
    WRITE LOCK TABLE Medals         //not strictly necessary
    WRITE LOCK TABLE MedalTally     //not necessary at table level
    WRITE LOCK TABLE Medalists      //not necessary at table level
    READ LOCK TABLE Competitors     //not strictly necessary

    // perform the body of the transaction in (a)

    UNLOCK TABLE Results
    UNLOCK TABLE Medals
    UNLOCK TABLE MedalTally
    UNLOCK TABLE Medalists
    UNLOCK TABLE Competitors
COMMIT
END TRANSACTION
```

+ Q1-4 2PL

- Q: Show that your lock/unlock annotations constitute two-phase locking.
  - No transaction should request a lock after it releases one of its locks
    - A transaction should not release a lock until it is certain that it will not request another lock
  - *Growing Phase*: obtain locks and access data
  - *Shrinking Phase*: release locks
- A: ~



2PL

+ Q1-4 Dead Lock

1. Deadlock occurs on the tables need lock
   - **MedalTally**(Country, Medal, Number)
   - **Medalists**(CompID, NMedals)
2. Transactions do not obtain all the locks they need

T1:

```
WRITE LOCK TABLE MedalTally

//waiting for Medalists available
UPDATE Medalists(CompID, NMedals)
SET NMedals = NMedals + 1
WHERE CompID = CompIdent

SET Number = Number + 1
WHERE MedalTally.Country =
(SELECT Country FROM Competitors
WHERE CompID = CompIdent)
AND Medal = MedalAwarded


UNLOCK TABLE MedalTally
```

T2:

```
WRITE LOCK TABLE Medalists

UPDATE Medalists(CompID, NMedals)
SET NMedals = NMedals + 1
WHERE CompID = CompIdent
//waiting for MedalTally available
SET Number = Number + 1
WHERE MedalTally.Country =
(SELECT Country FROM Competitors
WHERE CompID = CompIdent)
AND Medal = MedalAwarded


UNLOCK TABLE Medalists
```

Please note: 2PL can make sure our results are correct.

This is a sufficient condition

But 2PL can trigger Dead Lock as well.

Example: Lecture Notes @Week 4 Page 18: T_1' & T_2'

+ Q1-5 Failures

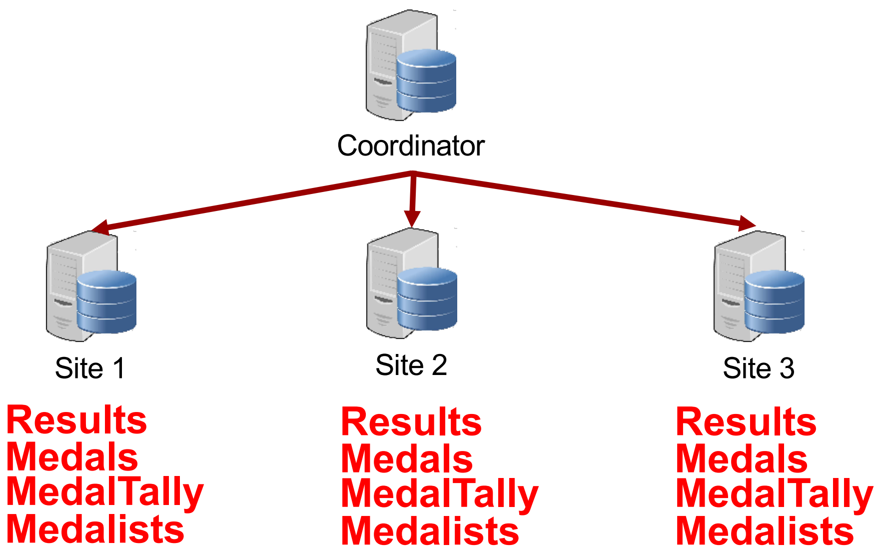
- Q:Is there any circumstance in which the program would have to abort the transaction?
- Transaction Failures
  - Incorrect input data
  - Present/Potential deadlock
  - Data are accessed by another transaction
- System Failures
  - Media failure, processor failure, communication break, power outrage…
- Our Program?
  - Why we need DBMS?
  - Error handling

+ Q2 Distributed Replication

- Each table in the above question can be stored at a different site, and some tables, such as **Medals**, **Medalists** and **MedalTally** may need to be replicated at several sites.
  - Clearly, the transaction in the previous question will comprise of a set of sub-transactions executing at different sites to updating data there.
  - The transaction can only finish when all its sub-transactions successfully finish.
  - This process is enforced by using the two-phase commit (2PC) protocol.

+ Q2 Distributed Replication

- For each Transaction
  - Break into sub-transactions for each site



THE UNIVERSITY OF QUEENSLAND
AUSTRALIA

Q2 Distributed Data Replication

- Purposes
  - System Availability
  - Performance
  - Scalability
  - Application Requirements
- Replicas converge to the same value
- Issue
  - Consistency
    - Where updates are performed
    - How the updates propagate

+ Q2-1 Synchronous Replication

- When the update commits, all the copies have the same value
  - Benefits
    - Consistent
    - No need for remote read
    - Easy to handle failure
  - Drawbacks
    - A transaction has to update all the copies before it can terminate
      - The **response time** performance suffers from the slowest one
      - If **any is unavailable**, then the whole transaction cannot commit

+ Q2-1 Synchronous Replication

- Q:Suppose there are a large number of replicas for **MedalTally**. Should we adopt the synchronous replication strategy for this table?

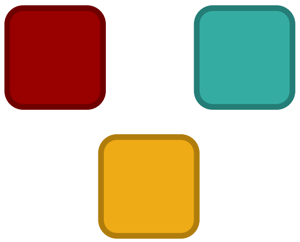
- All copies of the table should be consistent with each other at all time.
- The more sites, the higher possibility of failure \ Locks
- If one of the replica sites fails during the execution of a transaction with update operations, the whole transaction needs to be aborted.

+ Q2-2 Asynchronous Replication

- Partially updated
  - Tolerate inconsistency for better performance
    - Lower response time for update transaction
    - Not all consistent, some replicas may be **out-of-date**
  - Eventually consistency

+ Q2-2 Asynchronous Replication

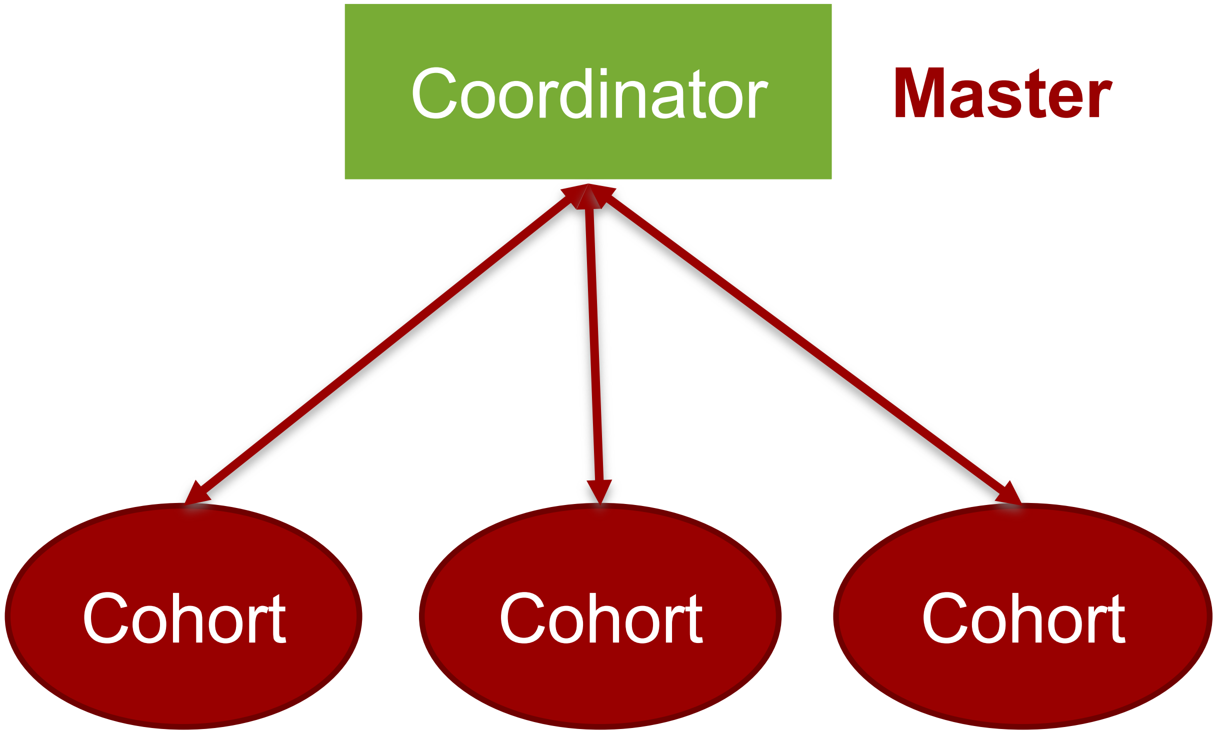
- Q:Would you recommend using asynchronous replication for this application?
- Read
  - Tolerable
- Update
  - Need strategies



+ Q2-3 Distributed Reliability

- Q:Suppose one of the **MedalTally** replica sites fails during the transaction. Show the exchange of messages among sub-transactions resulting in the two-phase commit (2PC) protocol to issue an abort instruction.
  - Atomicity and Durability
    - All or Nothing
    - Commit then permanent
  - All sites involved in the execution of a distributed transaction agree to commit before its effects are made permanent

+ Q2-3 Distributed Database Structure



+ Q2-3 2PC

- Protocol
  - TCP/IP, HTTP, SSH …
  - Two Phase Commit Protocol

1. **Commit-Request Phase** (or Voting Phase)
   1. Coordinator ask cohorts to prepare/work
   2. Cohorts response yes/no
      - Does not know the situation of the whole transaction, only about itself
      - If responses yes, the cohort has to wait for the response from coordinator.
      - Cannot abort by itself

2. **Commit Phase**
   3. If all yes, ask cohorts to commit
      - Else, tell cohorts to abort
   4. Cohorts ACK to coordinator

+ Q2-3 Success

//execution/Preparation phase
Coordinator prepare to UpdateResults
Coordinator prepare to UpdateMedalists
Coordinator prepare to UpdateMedals
Coordinator prepare to UpdateMedalTally

// prepare-to-commit phase
UpdateResults yes to Coordinator
UpdateMedalists yes to Coordinator
UpdateMedals yes to Coordinator
UpdateMedalTally yes to Coordinator

//commit-or-abort phase
Coordinator commit to UpdateResults
Coordinator commit to UpdateMedalists
Coordinator commit to UpdateMedals
Coordinator commit to UpdateMedalTally

//execution status reporting phase
UpdateResults ack to Coordinator
UpdateMedalists ack to Coordinator
UpdateMedals ack to Coordinator
UpdateMedalTally ack to Coordinator

+ Q2-3 Failure

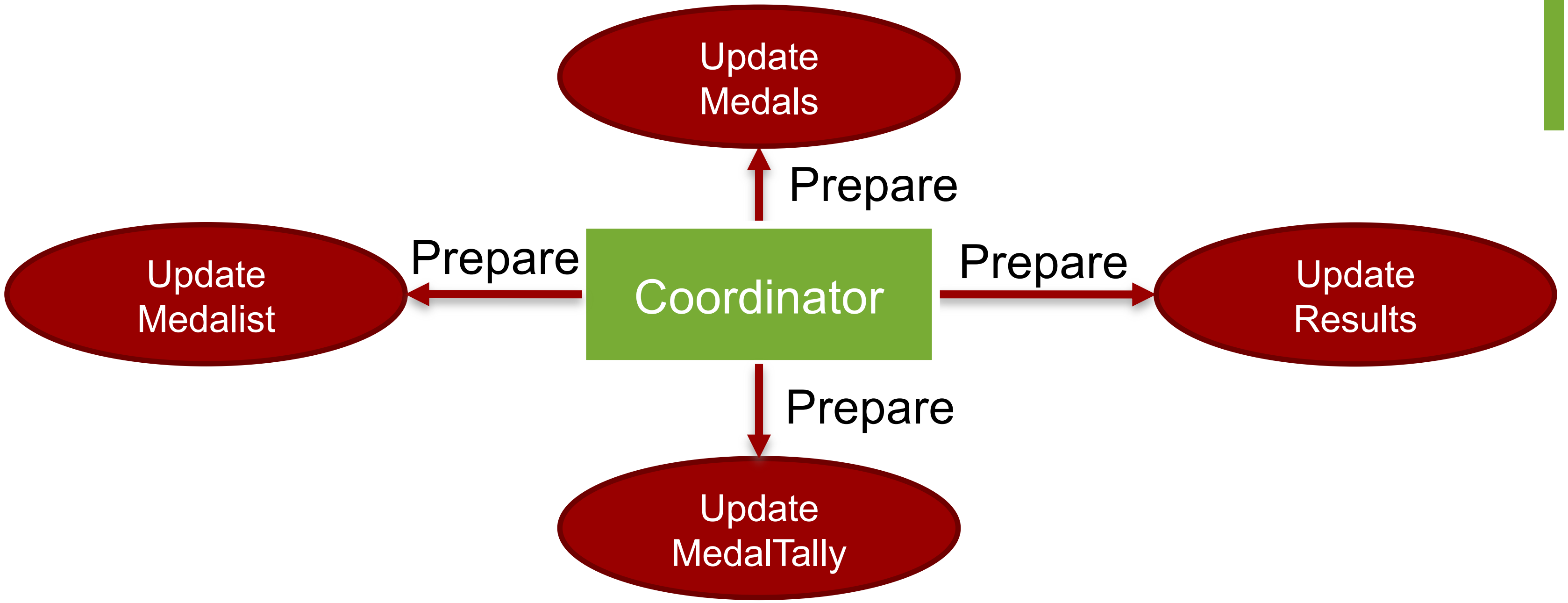
//execution/Preparation phase
Coordinator prepare to UpdateResults
Coordinator prepare to UpdateMedalists
Coordinator prepare to UpdateMedals
Coordinator prepare to UpdateMedalTally

// prepare-to-commit phase
UpdateResults yes to Coordinator
UpdateMedalists yes to Coordinator
UpdateMedals yes to Coordinator
UpdateMedalTally **no** to Coordinator
//failed subtransaction

//commit-or-abort phase
Coordinator abort to UpdateResults
Coordinator abort to UpdateMedalists
Coordinator abort to UpdateMedals
Coordinator abort to UpdateMedalTally

//execution status reporting phase
UpdateResults ack to Coordinator
UpdateMedalists ack to Coordinator
UpdateMedals ack to Coordinator
UpdateMedalTally ack to Coordinator

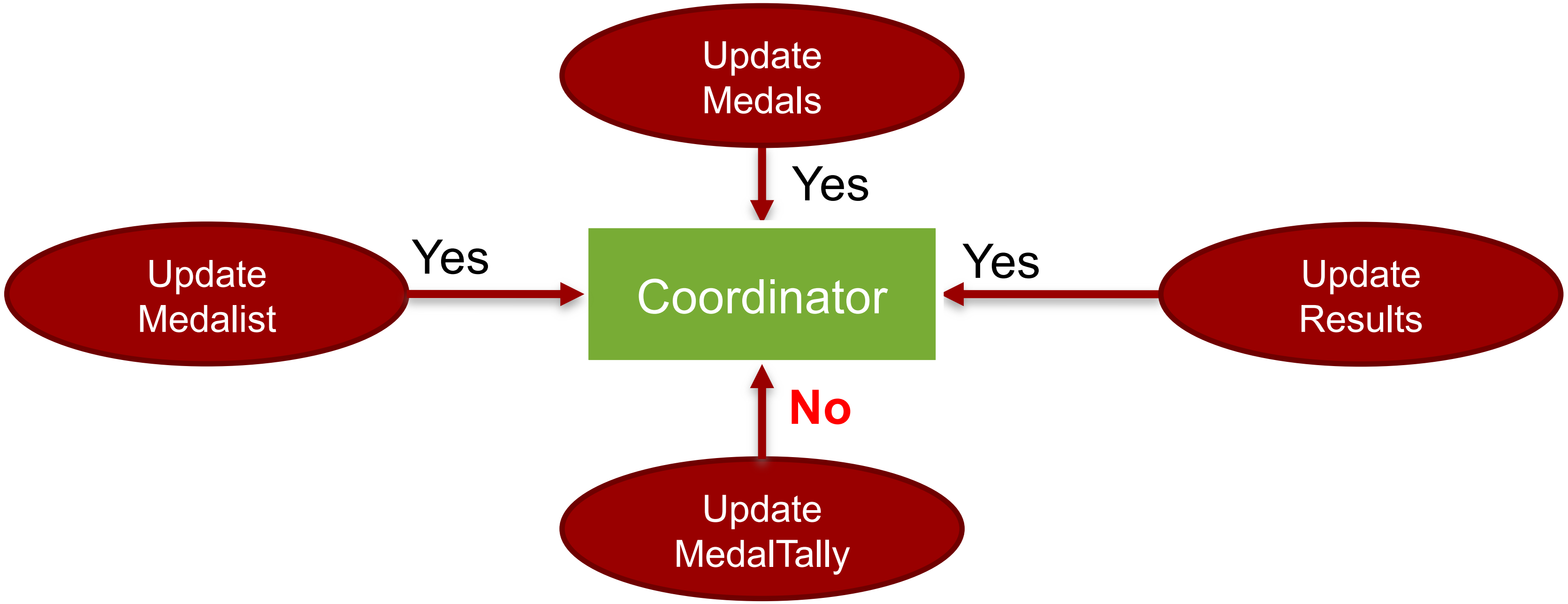
+ Q2-3 2PC Voting Phase 1



//Execution/Preparation phase

Coordinator execute to UpdateResults
Coordinator execute to UpdateMedalists
Coordinator execute to UpdateMedals
Coordinator execute to UpdateMedalTally

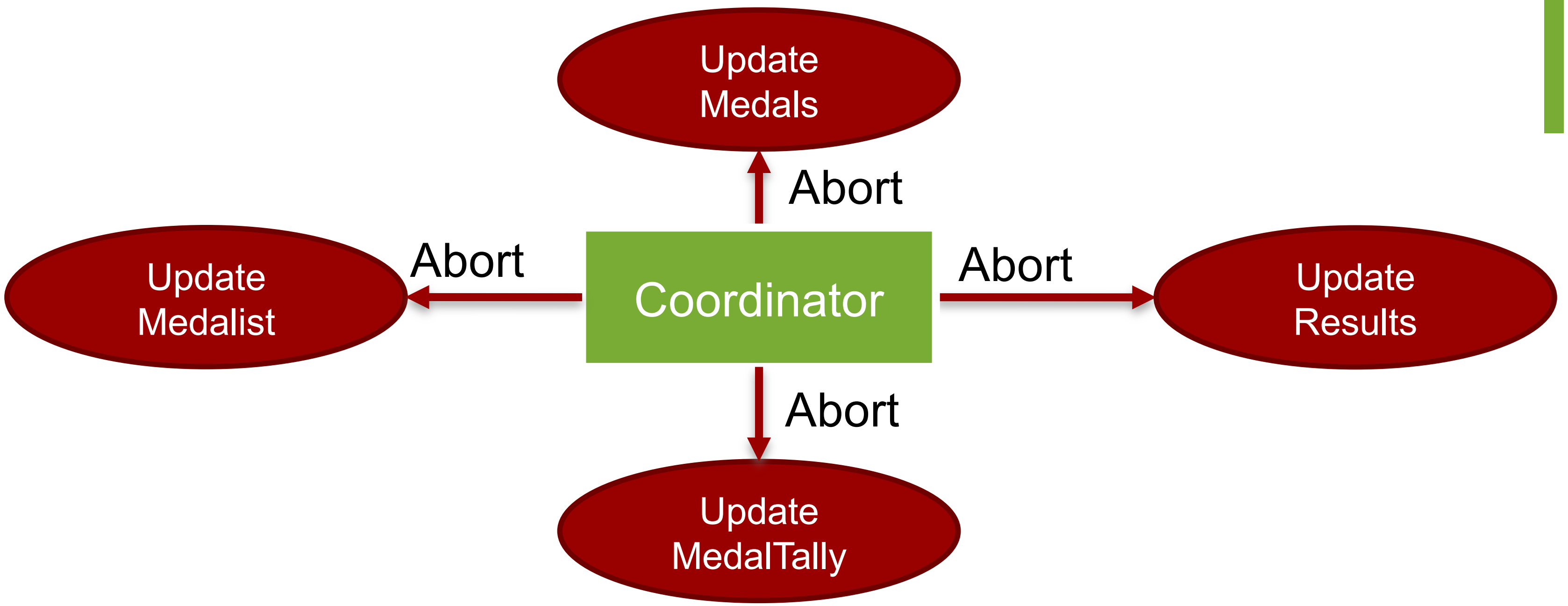
+ Q2-3 2PC Voting Phase 2



// prepare-to-commit phase
UpdateResults yes to Coordinator
UpdateMedalists yes to Coordinator
UpdateMedals yes to Coordinator
UpdateMedalTally **no** to Coordinator
//failed subtransaction

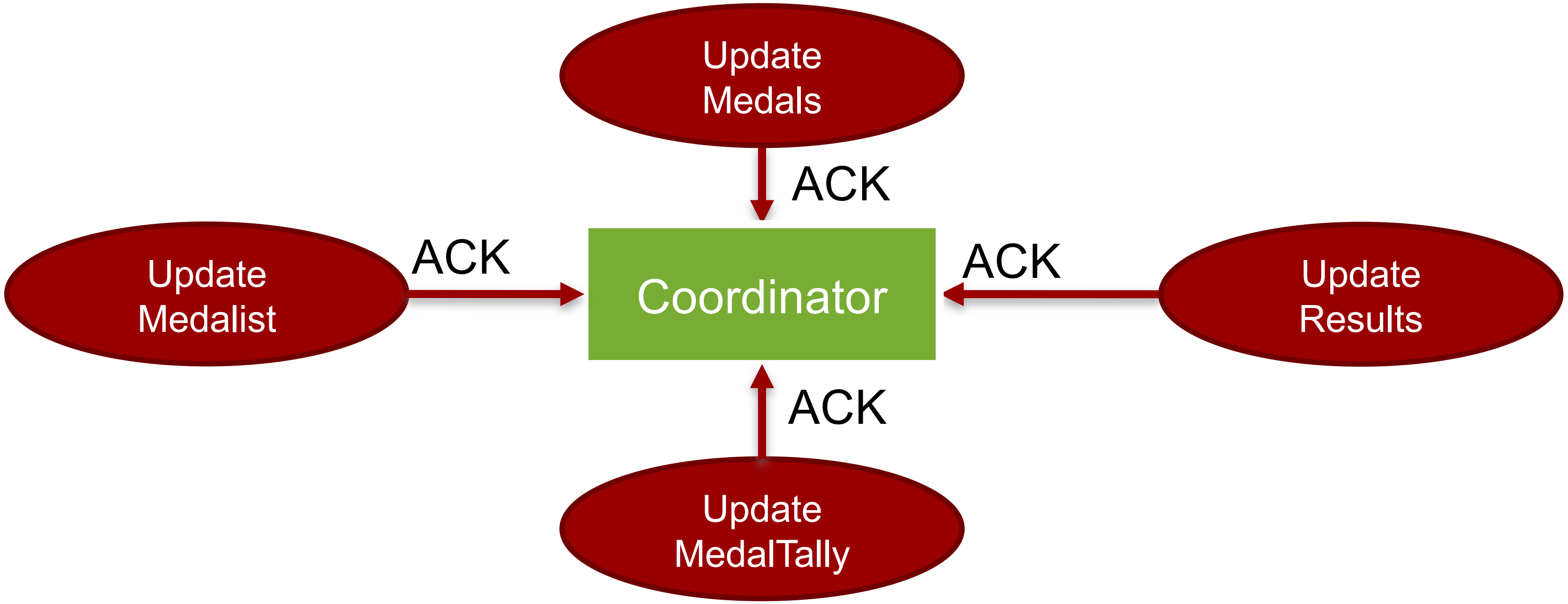
- All or Nothing
  - If all yes, commit
  - If any no, abort

+ Q2-3 2PC Commit/Abort Phase 1



// prepare-to-commit phase
UpdateResults yes to Coordinator
UpdateMedalists yes to Coordinator
UpdateMedals yes to Coordinator
UpdateMedalTally **no** to Coordinator
//failed subtransaction

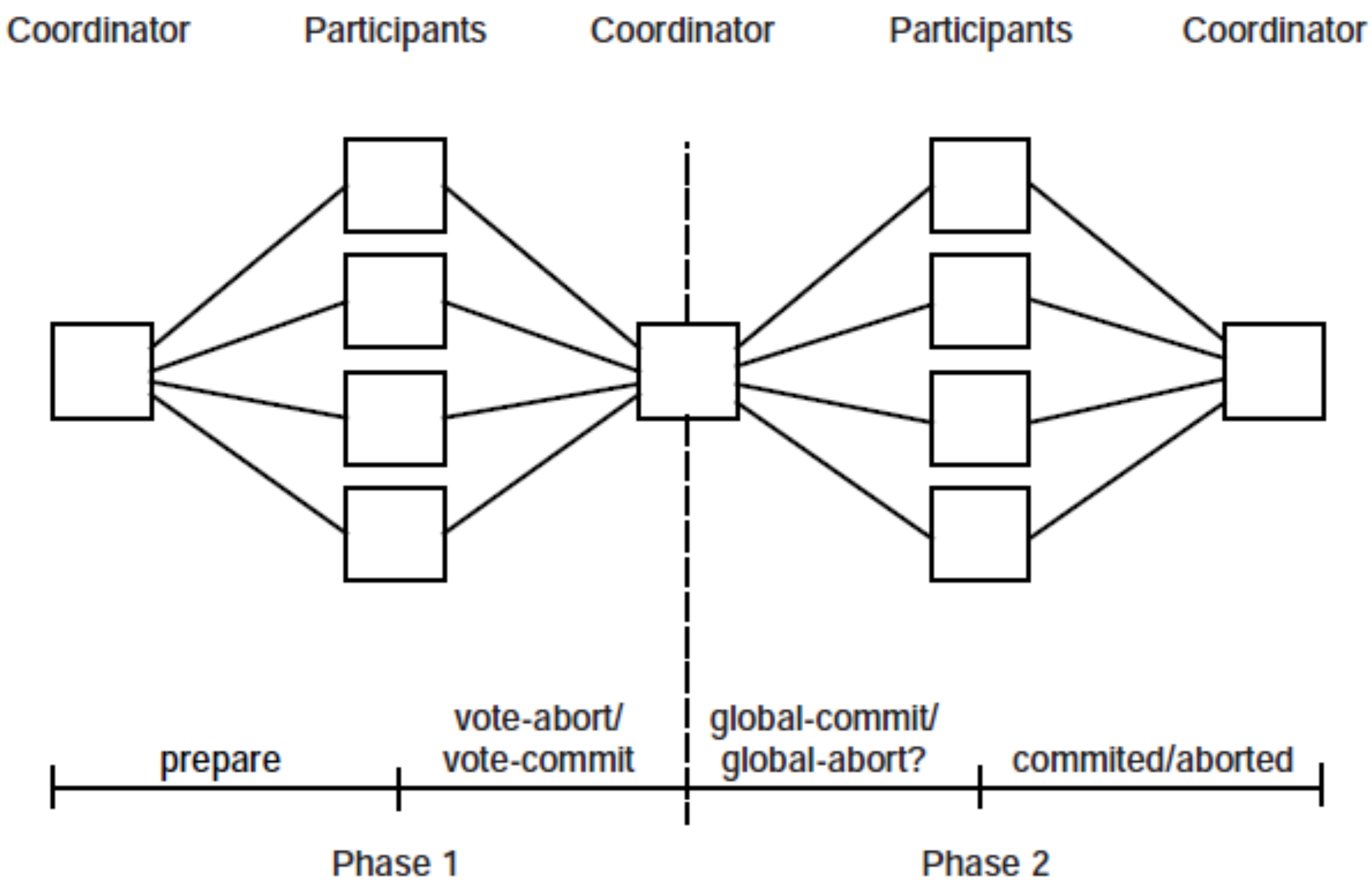
+ Q2-3 Commit/Abort Phase 2



//execution status reporting phase

UpdateResults ack to Coordinator
UpdateMedalists ack to Coordinator
UpdateMedals ack to Coordinator
UpdateMedalTally ack to Coordinator

+ Q2-3 2PC Communication Structure



+ Q3 2PC Failure Handling

- When 2PC is used as the commit protocol, explain how the system recovers from failure and deals with particular *transaction T* in each of the following cases:

a. A subordinate site for *T* fails before receiving a *prepare* message.

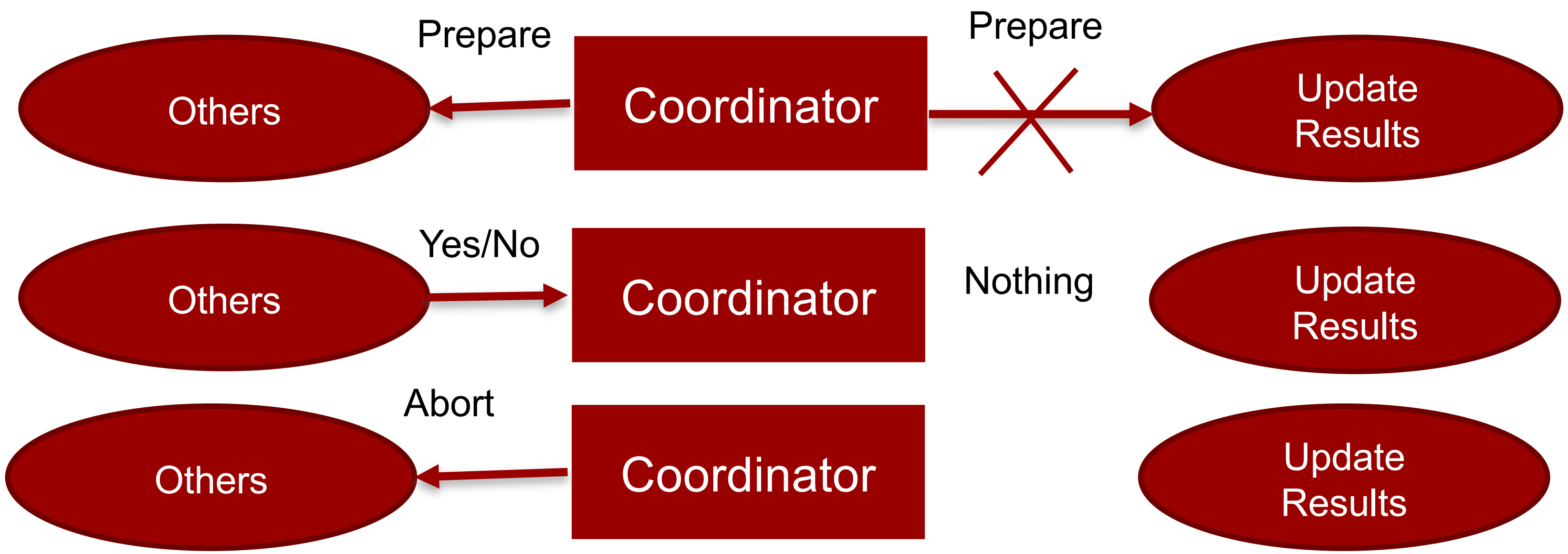
b. A subordinate site for *T* fails after receiving a *prepare* message but before making a decision.

c. The coordinator site for *T* fails before sending a *prepare* message.

d. The coordinator site for *T* fails after writing an *abort* log record but before sending any further messages to its subordinates.

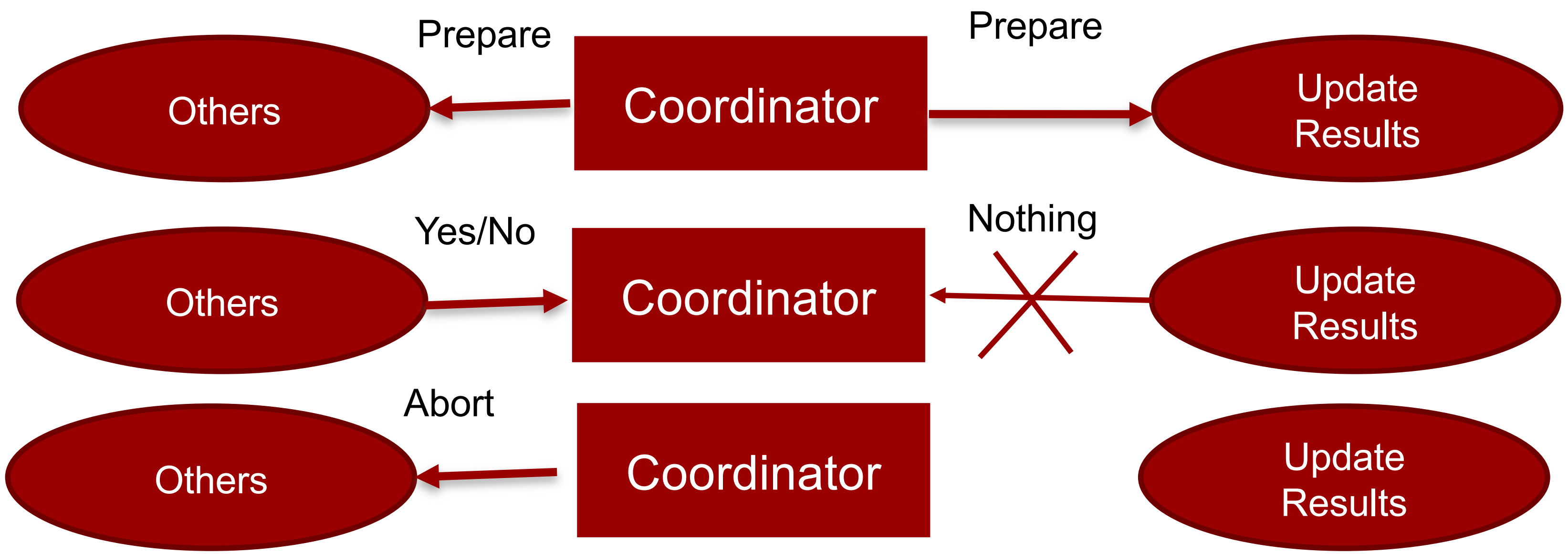
+ Q3-1 Cohort Fail

- Q: A subordinate site for *T* fails before receiving a *prepare* message
- A: The coordinator will not receive either a yes or a no message from the subordinate, and will *abort* the transaction (once time-outed).
- It will instruct all the subordinates that replied yes to abort.
- Once the subordinate recovers, it will check with the coordinator, which will send the default abort message and the sub transaction will also be aborted.



+ Q3-2 Cohort Fail

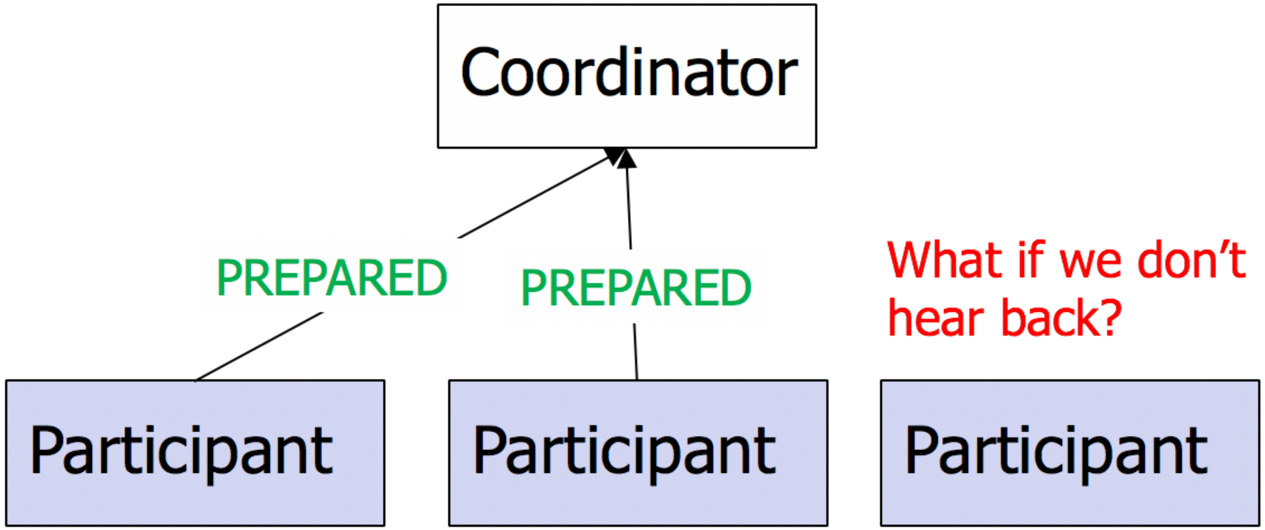
- Q :A subordinate site for *T* fails after receiving a *prepare* message but before making a decision
- A:The situation is the same as above, since the subordinate has not yet made a decision. No message has been sent to the coordinator.



+ Q3-2 Cohort Fail

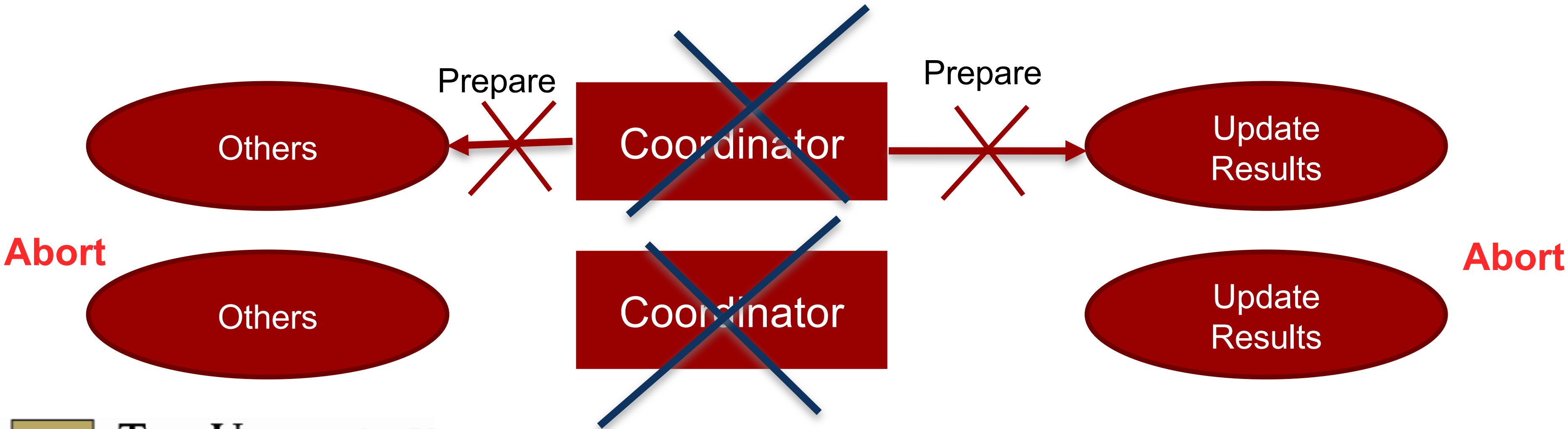
- Q :A subordinate site for *T* fails after receiving a *prepare* message but before making a decision

- A:The situation is the same as above, since the subordinate has not yet made a decision. No message has been sent to the coordinator.



+ Q3-3 Coordinator Fail

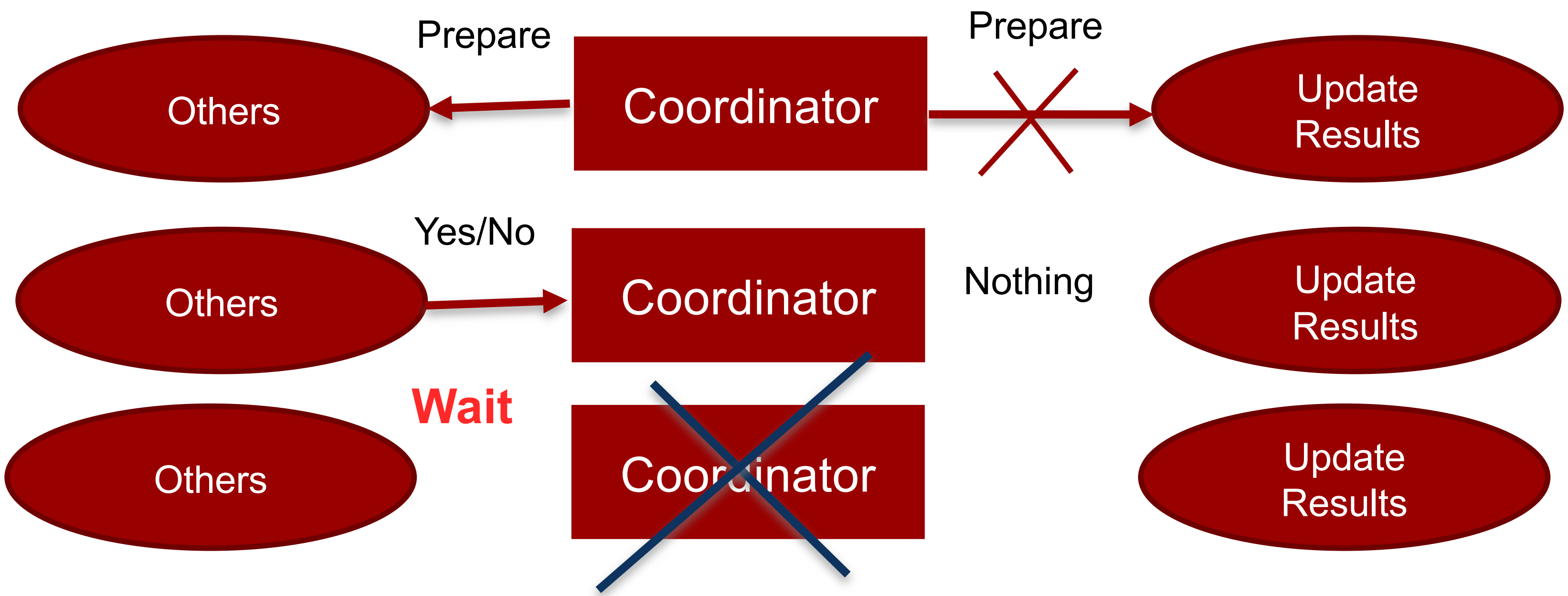
- Q: The coordinator site for *T* fails before sending a *prepare* message.
- A:Since there is no prepare, commit or abort log record, T can be unilaterally aborted and undone, and an end log record written. Since the coordinator did not send the prepare message before it crashed, the subordinates may abort once time-outed.



THE UNIVERSITY OF QUEENSLAND
AUSTRALIA

Q3-4 Coordinator Fail

- Q:The coordinator site for *T* fails after writing an *abort* log record but before sending any further messages to its subordinates.



Subordinates cannot abort unilaterally after voted yes.

So 2PC is a blocking protocol → 3PC

+ Q3 3PC

- **Assumptions**
  1. Each site uses the write-ahead-log protocol
  2. At most one site can fail during the execution of the transaction
  3. Set a timer for abort

- 2PC
  - Phase 1:
    - Cohorts know the transaction ahead
    - Does not know others' conditions
  - Phase 2:
    - Waiting for the next instruction from coordinator
    - Commit/Abort

- 3PC
  - Phase 1:
    - Cohorts know the transaction ahead
    - Does not know others' conditions
  - Phase 2:
    - PreCommit
      - Cohorts know the transaction will start
        - Even if coordinator fails, no "Commit" received, it is OK to commit when time out
        - If one of the cohorts fails after voting "yes", coordinator does not have enough "ack", coordinator sends abort
        - If one of the cohorts fails after receiving "PreCommit", it still commits after reading its log
    - Abort/Received nothing
      - Cohorts know the transaction will abort
  - Phase 3:
    - Commit/Abort

+ Q3 3PC

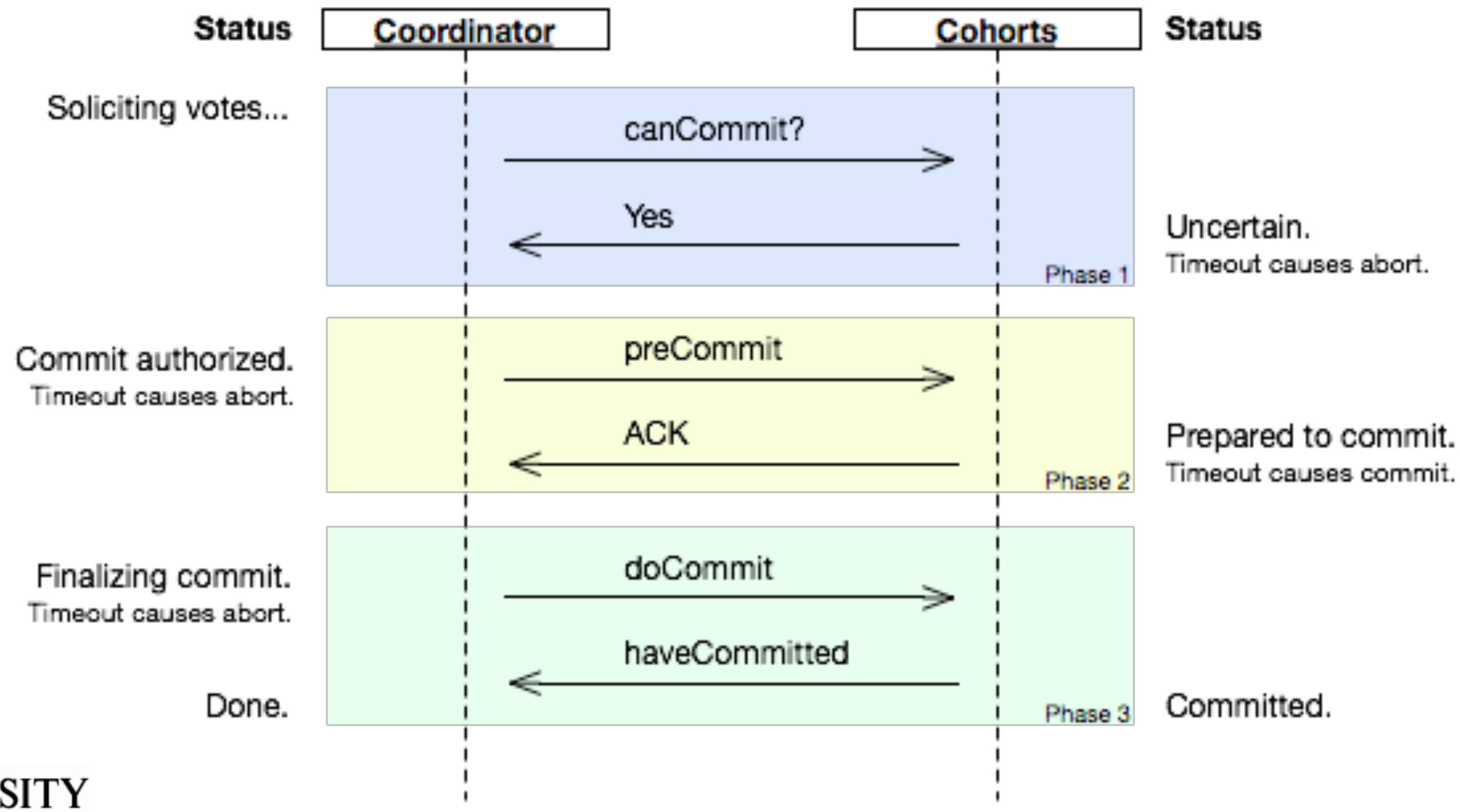
Further reading: https://en.wikipedia.org/wiki/Three-phase_commit_protocol

- Non-blocking
  - Places an upper bound on the amount of time required before a transaction either commits or aborts

Phase 1 in 2PC

New in 3PC

Phase 2 in 2PC

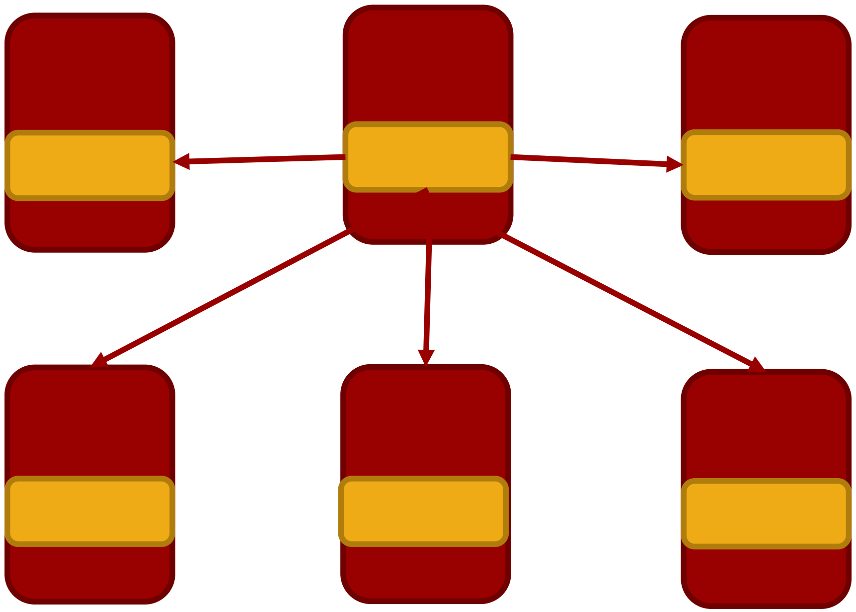


+ Q4 Distributed Data Consistency

- Distributed transaction management can use primary-copy based approach or voting-based approach to maintain data consistency among multiple copies.
  - Please explain how these two approaches work, and compare their advantages and disadvantages.

+ Q4 Primary Copy

- The **primary-copy approach** selects one copy as the *master* copy (e.g., the one at the 'birth site' of the data item).



- Simple to update
- Primary Site
  - Primary site overload
  - Primary Copy
- Others Temporary Inconsistency

+ Q4 Voting Based

- Version Number
  - Monotonically increasing
- N copies
  - Write at least m
  - Read at least n
  - m + n > N
  - Guarantee can read the latest version

Version No.

....

0102

0103

0104

m+n > Total number

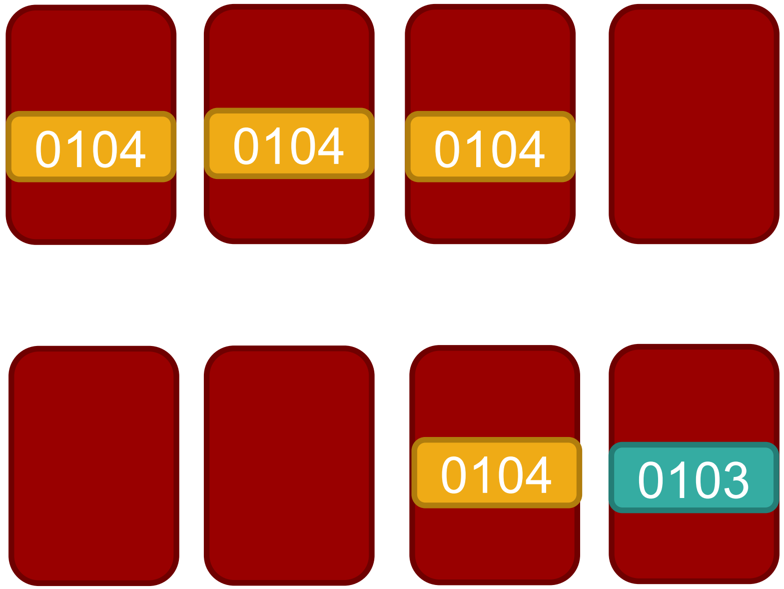
- Advantages
  - More reliable
  - Writes fewer copies.
- Disadvantages
  - Read transaction needs to check many sites

Write: 3

(m)

Read: ?

(n)



THE UNIVERSITY OF QUEENSLAND AUSTRALIA