CISC 5597 / 6935: Distributed Systems

Lecture 9: Concurrency Control

Multiple-objects, multiple-servers

- Consistency for multiple-objects, multiple-servers
  - Single Server Case
    - Two Phase Locking
  - Distributed Transactions
    - Two Phase Commit

Concurrency Control

- In the context of concurrency, what have we been focused on avoiding?
  - Generally so far, we have been focused on various ways to ensure safe concurrent access to a single resource.
  - Unsafe concurrent access to shared resources.
    - e.g., race proposals in Paxos.
  - Going forward, we want to handle composite operations– we want to build concurrent systems which act on multiple distinct objects

Composite Operations

- Let's start with an example of composite operations.
- Imagine a banking application with 3 functions that can access account information:

- If each of these 3 functions were implemented in a thread-safe way, we should be OK, right?
  - NO

Composite Operations

- Muti-threading works?

Assuming these 2 threads are concurrently executing... What can happen?

```
// thread 1: transfer 100
// from savings->current
 withdraw(savings, 100)
 deposit(current, 100)
```

```
// thread 2: check balance
 s = getBalance(savings)
 c = getBalance(current)
 tot = s + c
```

Problems with Composite Operations

- Two issues
  - Insufficient Isolation
    - Individual operations each being atomic is not enough
    - Instead, we want the deposit & withdraw making up the transfer to happen as one atomic operation…
  - Fault Tolerance
    - In the real-word, programs (or systems) can fail
    - Need to make sure we can recover safely if failure happens (particularly– no corrupted data!)

Transactions!

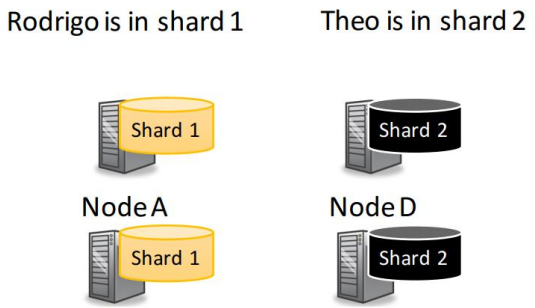
Yep! The database kind.

What is Transaction?

- A set of operations that need to be performed together.
  - Example 1: transferring money between accounts
  - Example 2: shopping cart checkout

  - Initially Theo and Rodrigo have $100.
  - Goal: Transfer $50 from Rodrigo to Theo.

Read(R)
Update(R, $50)
Read(T)
Update(T, $150)



Ideal: either the whole 4 operations happen, or none happen Worst case: only a subset occur

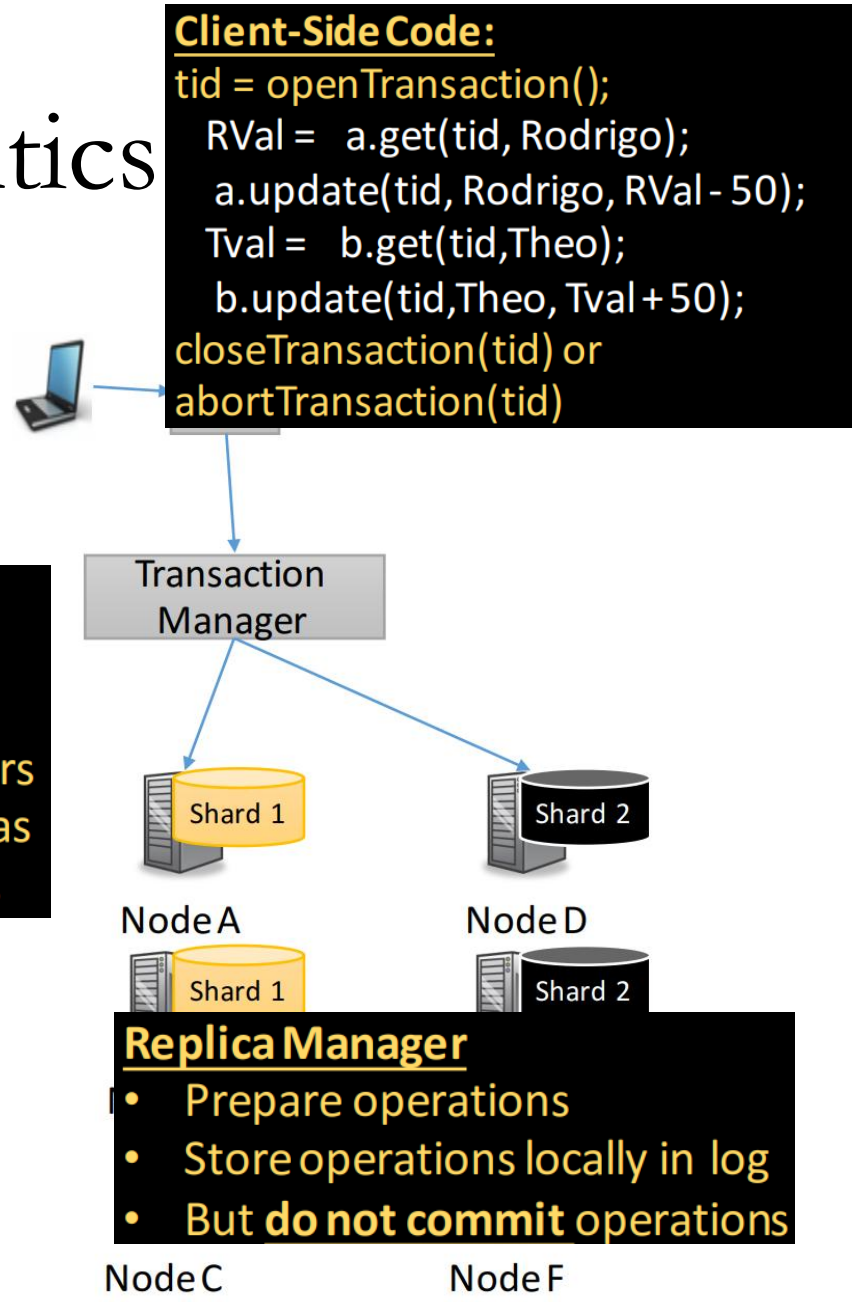
Transaction Background

- Transactions are defined as having the 4 ACID properties.
  - **Atomicity**: Each transaction completes in its entirely or is aborted. If aborted, should not have effect on the shared global state.
  - Example: Update account balance on multiple servers
  - **Consistency**: Each transaction preserves a set of invariants about global state. (Nature of invariants is system dependent).
  - Example: in a bank system, law of conservation of $$
  - **Isolation**: Also means serializability. Each transaction executes as if it were the only one with the ability to read/write shared global state.
  - **Durability**: Once a transaction has been completed, or "committed" there is no going back. In other words, there is no "undo".

Distributed Transactions Semantics

- Transaction Manager (TM)
  - Server in charge of orchestrating the transaction
- Steps for transaction
  - Client initiates a transaction
  - TM gives client a Transactio
  - Client submits operations to
  - TM relays operations to replicas
  - Client commits transaction
  - TM performs two phase commit

Transaction Manager
- TM hands out TIDs
- TM manages and relays operations to Replica Leaders
- TM keeps track of all Replicas involved in the transactions



A Transaction Example: Bank

- Array Bal[i] stores balance of Account "i"
- **Goal**: Implement: xfer, withdraw, deposit

```
xfer(i, j, v):
   if withdraw(i, v):
      deposit(j, v)
   else
      abort
```

```
deposit(j, v):
   Bal[j] += v
```

```
withdraw(i, v):
   b = Bal[i]          // Read
   if b >= v           // Test
      Bal[i] = b-v     // Write
      return true
   else
      return false
```

A Transaction Example: Bank

- Imagine the following scenario:
  - Bal[x] = 100, Bal[y] = Bal[z] = 0

- 2 transactions:
  - T1: xfer(x,y,60)
  - T2: xfer(x,z,70)

- **What will happen if these 2 transactions follow ACID properties and happen in some serial order?**

A Transaction Example: Bank

- T1; T2:
  - T1 Succeeds;
  - T2 Fails.
  - Bal[x]=40, Bal[y]=60

- T2; T1:
  - T2 Succeeds.
  - T1 Fails.
  - Bal[x]=30, Bal[z]=70

- What if we weren't careful about ACID properties? Is there a race condition?
  - ACID VIOLATION: NOT ISOLATED, NOT DURABLE
  - Updating Bal[x] with Read/Write interleaving of T1,T2
  - Bal[x] = 30 or 40; Bal[y] = 60; Bal [z] = 70

A Transaction Example: Bank

- **Goal**: Implement: xfer, withdraw, deposit
- For Consistency, we can implement **sumbalance()**
  - **Do you see anything wrong with this?**
    - **State invariant sumbalance=100 violated!**

```
xfer(i, j, v):
   if withdraw(i, v):
      deposit(j, v)
   else
      abort
```

```
deposit(j, v):
   Bal[j] += v
```

```
withdraw(i, v):
   b = Bal[i]        // Read
   if b >= v         // Test
      Bal[i] = b-v   // Write
      return true
   else
      return false
```

```
sumbalance(i, j, k):
   return Bal[i]+Bal[j]+ Bal[k]
```

sumbalance()

Implementing Transaction with Locks

- We can use locks to wrap xfer.

```
xfer(i, j, v):
   lock()
   if withdraw(i, v):
      deposit(j, v)
   else
      abort
   unlock()
```

Solution 2:
– Grab short-term fine-grained locks on an item before access
– Lock(A) Read(A), Unlock(A), Lock(B) Write(B), Unlock(B) ....

```
xfer(i, j, v):
   lock(i)
   if withdraw(i, v):
      unlock(i)
      lock(j)
      deposit(j, v)
      unlock(j)
   else
      unlock(i)
      abort
```

- Solution 1:
  - Grab global lock before transaction starts – Release global lock after transaction commits
- However, is this a good approach?
  - Sequential bottleneck due to global lock!

Problem Solved?

Nope. Consistency violation. Releasing lock i early before another transaction can cause consistency violation.

Implementing Transaction with Locks

- Potential fix:
  - Release locks when update of all state variables complete.
  - Works now?
  - Nope. Deadlock!
    - Bal[x]=Bal[y]=100
    - xfer(x,y,40) and xfer(y,x,30)

```
xfer(i, j, v):
    lock(i)
    if withdraw(i, v):
        lock(j)
        deposit(j, v)
        unlock(i)
        unlock(j)
    else
        unlock(i)
        abort
```

Strict Two-Phase Locking (2PL)

Problems for Solution 2 above

T1

```
x = Read(A)
y = Read(B)
Write(A, x-100)
Write(B, y+100)
```

T2

```
x = Read(A)
y= Read(B)
Print(x+y)
```

Possible? (short-term, fine-grained locks on reads/writes)
R(A),R(B), W(A), R(A),R(B) C W(B) C

Locks on writes should be held till end of transaction

Read an uncommitted value

Solution 3

- Solution 3
  - fine-grained locks
  - **long-term locks for writes**
    - grab lock before write, release lock after tx commits/aborts
  - **short-term locks for reads**

Solution 3's problem

T1

```
x = Read(A)
y = Read(B)
Write(A, x-100)
Write(B, y+100)
```

T2

```
x = Read(A)
y= Read(B)
Print(x+y)
```

Possible? long-term locks for writes, short-term locks for reads

R(A),R(B), W(A), R(A),R(B), C W(B),C

Read locks must be held till commit time

Non-repeatable reads

R(A), R(A),R(B), W(A), W(B),C R(B)

Transactions: Split Into 2 Phases

- For reliability, transactions are split into two phases:

- **Phase 1 (Growing): Preparation:**
  - Determine what has to be done, how it will change state, without actually altering it.
  - Generate Lock set "L"
  - Generate List of Updates "U"
- **Phase 2 (Shrinking): Commit or Abort:**
  - Everything OK, then update global state
  - Transaction cannot be completed, leave global state as is
  - In either case, RELEASE ALL LOCKS

Strict 2 phase locking (2PL)

- 2 phase locking (2PL)
  - A growing phase in which the transaction is acquiring locks
  - A shrinking phase in which locks are released

- In practice,
  - The growing phase is the entire transaction
  - The shrinking phase is at the commit time

- Optimization:
  - Use read/write locks instead of exclusive locks
  - **Read lock (Shared lock):** others can still read it
  - **Write lock (Exclusive lock):** others CANNOT access to it

Strict Two-phase Locking (Strict 2PL) Protocol:

TXNs obtain:

- An **X (*exclusive*) lock** on object before **writing**.
  - If a TXN holds, no other TXN can get a lock (S or X) on that object.
- An **S (*shared*) lock** on object before **reading**
  - If a TXN holds, no other TXN can get *an X lock* on that object
- All locks held by a TXN are released when TXN completes.

Note: Terminology here- "exclusive", "shared"- meant to be intuitive- no tricks!

These policies ensure that no conflicts (RW/WR/WW) occur!

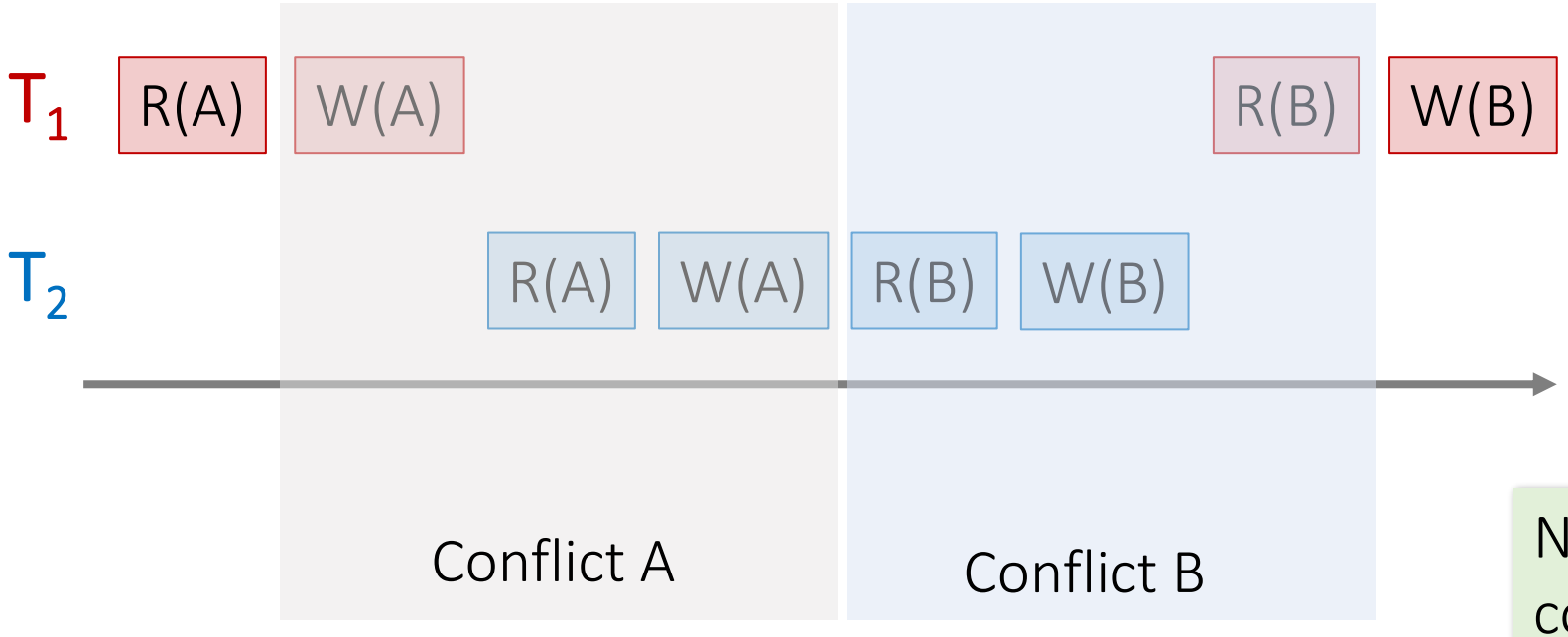
2PL in practice: an example

RLock(A)
x = Read(A)
RLock(B)
y = Read(B)
WLock(A)
buffer A←x-100
WLock(B)
buffer B←y+100
T1 issues commit :
log (A←0, B←200)
Write(A, 0)
Unlock(A)
Write(B, 200)
Unlock(B)
Commit returns

RLock(A)
x = Read(A)
RLock(B)
y = Read(B)
Print(x+y)
Unlock(A)
Unlock(B)

Example

A schedule that is **not** conflict serializable:



No way for the actions of conflicts A & B to ***both*** happen in this order in a serial schedule!

Conflict Dependency Graph

- Node for each committed TXN $T_1 \ldots T_N$

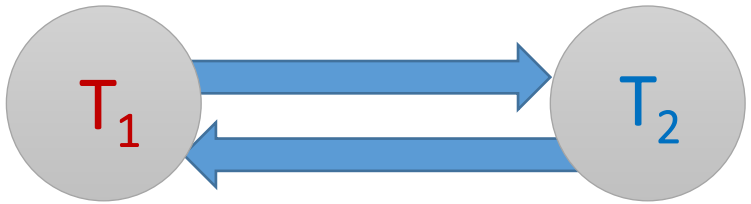
- Edge from $T_i \rightarrow T_j$ if an actions in T_i **precedes** and **conflicts** with an action in T_j

Theorem: Schedule is **conflict serializable** if and only if its dependency graph is acyclic

Conflict Dependency Graph

Example:

Conflict dependency graph:



A non-conflict serializable schedule has a **cyclic** conflict dependency graph!

More on 2PL

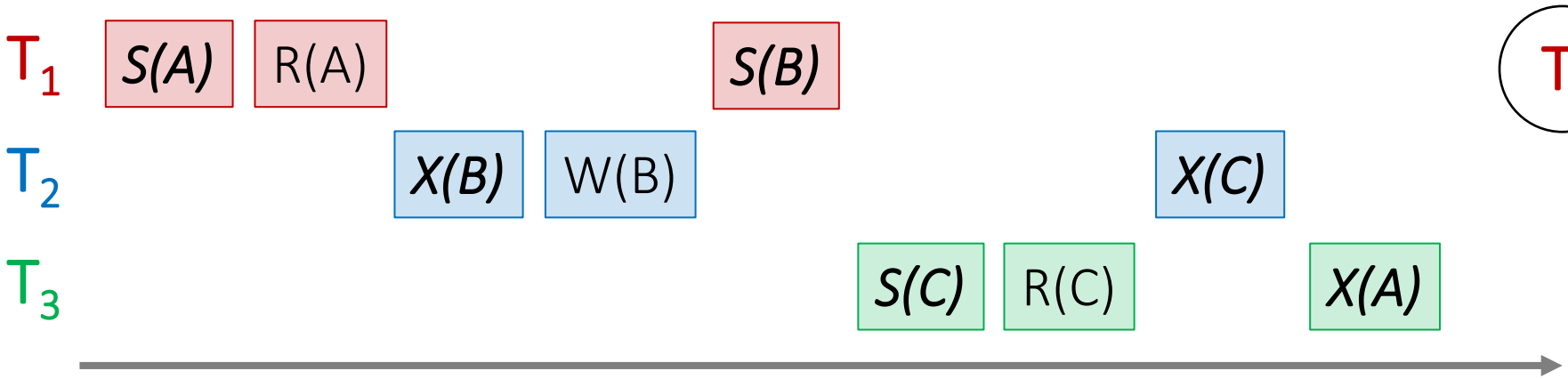
- What if a lock is unavailable?
  - Wait
  - Wait-for graph

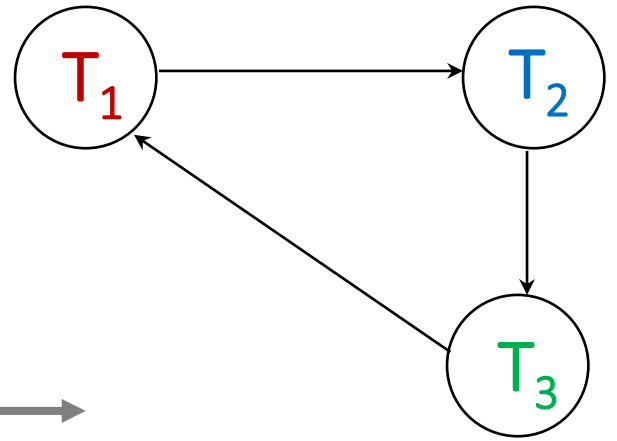
  - Deadlocks possible?

- How to cope with deadlock?
  - Grab locks in order? No always possible
  - Transaction manager detects deadlock cycles and aborts affected transactions
  - Alternative: timeout and abort yourself wait detect & abort

Deadlock Detection

Example:





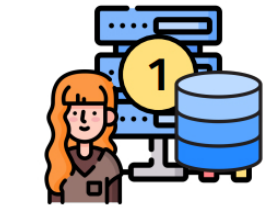
Deadlock!

In general, must search through this big graph. Sounds expensive! Is it?

Distributed Transactions
--- 2 Phase Commit

Distributed Transactions

- Partition databases across multiple machines for scalability
  - E.g., machine 1 responsible for account m, machine 2 responsible for account n



account m

account n

  - Transactions often touch more than one data partition.
  - How do we guarantee that all of the partitions commit the transactions, or none commit the transactions?
    - Transferring money from m to n.
    - Requirement: both machines do it, or neither.

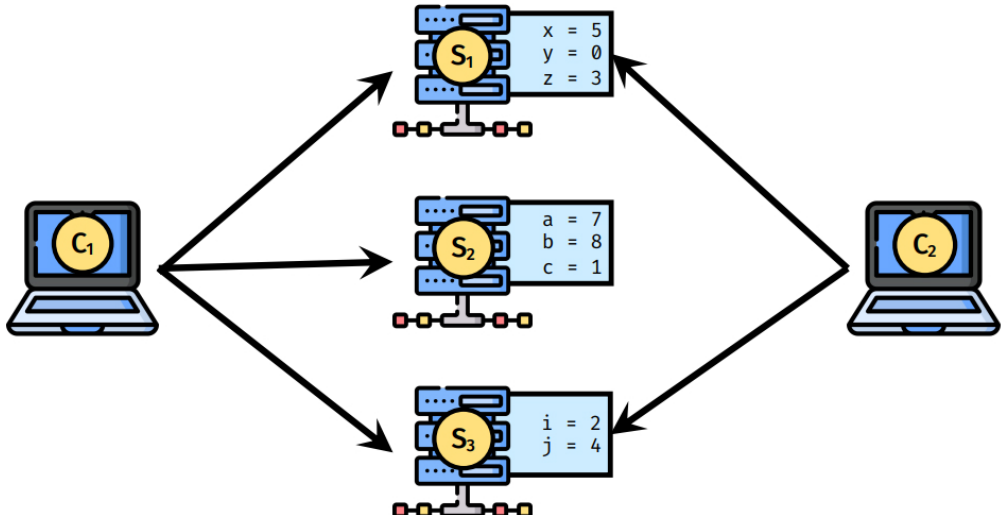
Modeling Distributed Transactions

- Multiple servers (S1, S2, S3, …), each holding some objects which can be read and written within client transactions

- Multiple concurrent clients (C1, C2, …) who perform transactions that interact with one or more servers
  - E.g. T1 reads x, z from S1, writes a on S2, reads + writes j on S3
  - E.g. T2 reads i, j from S3, then writes z on S1

- A successful commit implies agreement at all servers

```
T1 transaction {
  if (x<2) abort
  a:= z
  j:= j + 1
}
```




```
T2 transaction {
  z:= (i+j)
}
```

Enabling Distributed Transactions

- Similar idea as before, but:
  - State spread across servers
  - Single transaction should be able to read/modify global state while maintaining ACID properties.
  - Failures!

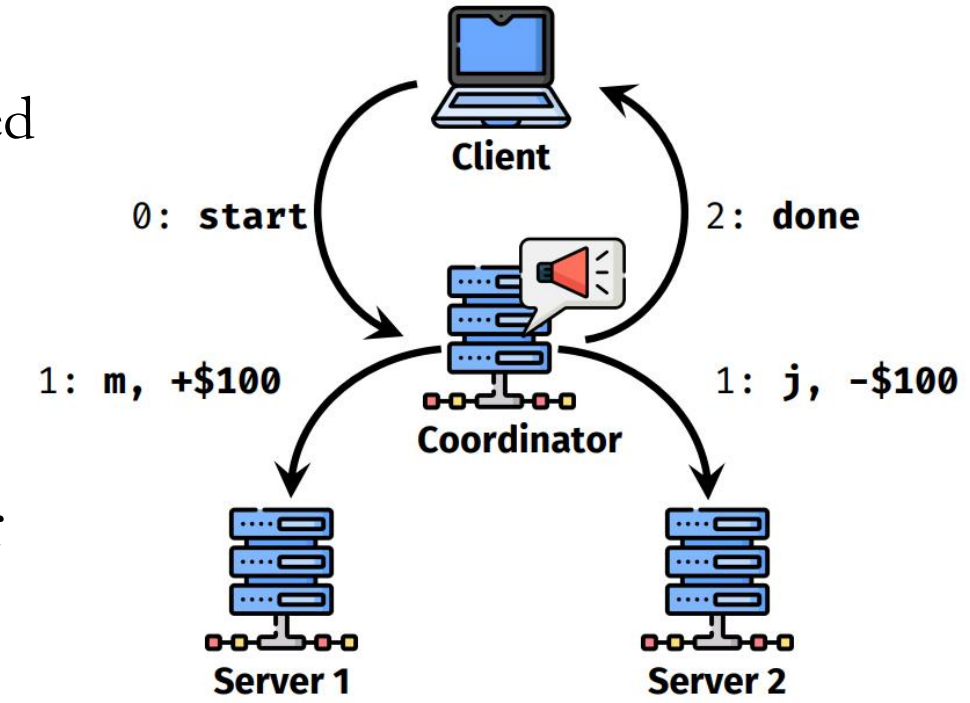
- Overall Idea:
  - Client initiates transaction.
  - Makes use of "coordinator"
  - All other relevant servers operate as "participants"
  - Coordinator assigns unique transaction ID (TxID)

- **Distributed Commit**
  - **2-phase commit protocol (2PC)**

Distributed Commit

- A simple distributed commit protocol:
  - Coordinator tells other processes that are involved (called participants) whether or not to locally perform the operation in question.
  - This is called the one-phase commit protocol.

- Even without failures, a lot can go wrong, for example:
  - Account j on Server 2 has only $90
  - Account m doesn't exist!



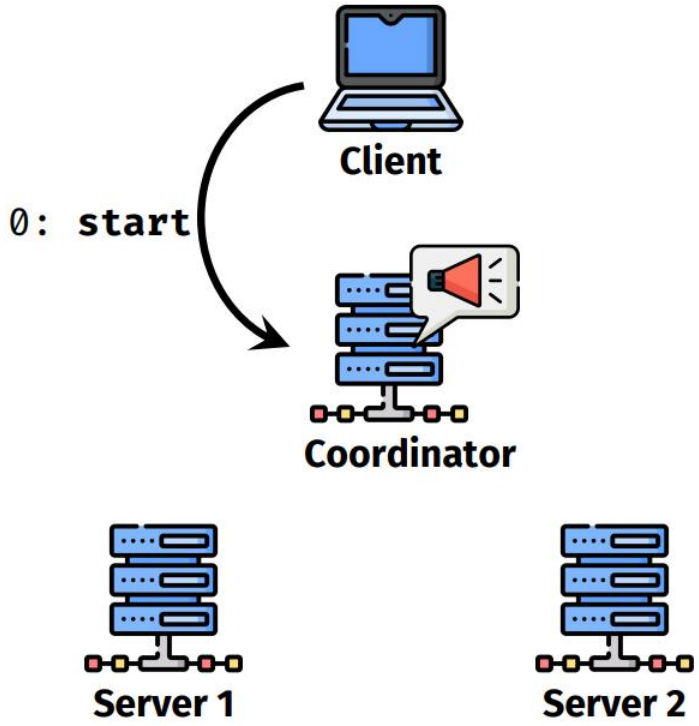
2-Phase Commit

- **Phase 1: Prepare & Vote**
  - Participants figure out all state changes
  - Each determines if it can complete the transaction
  - Communicate with coordinator

- **Phase 2: Commit**
  - Coordinator broadcasts to participants: COMMIT / ABORT
  - If COMMIT, participants make respective state changes

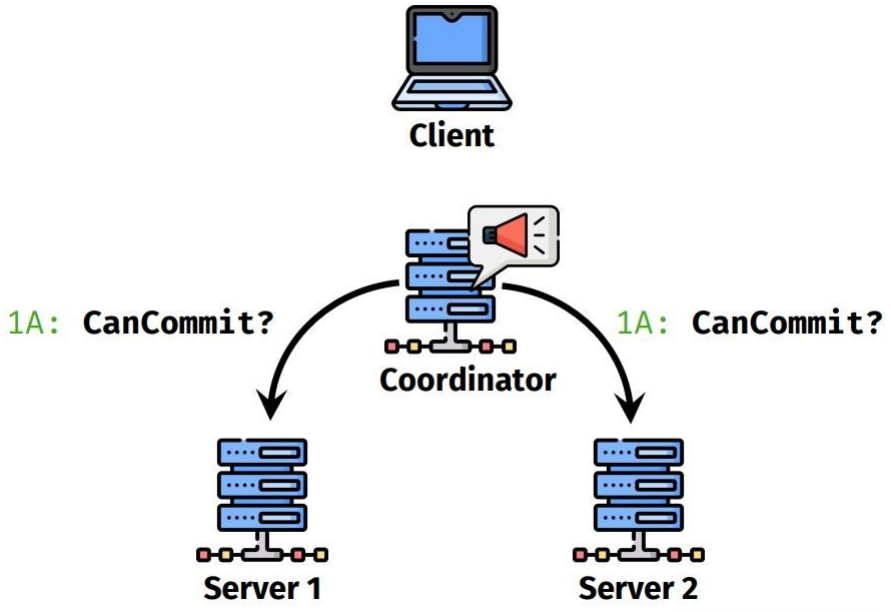
Implementing 2-Phase Commit

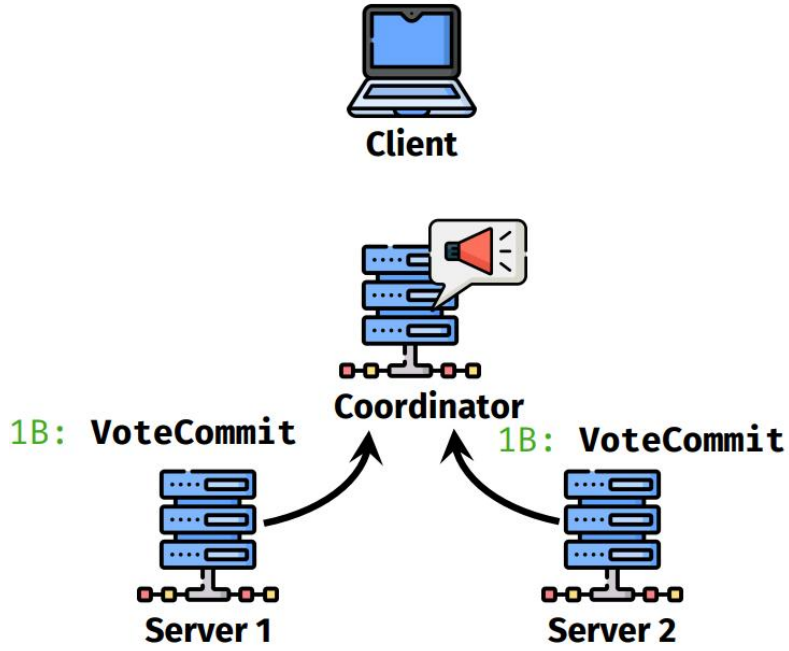
- Implemented as a set of messages.



Implementing 2-Phase Commit

- Messages in first phase
  - 1A: Coordinator sends **CanCommit?** Message to participants
  - 1B Participants respond: **VoteCommit** or **VoteAbort**

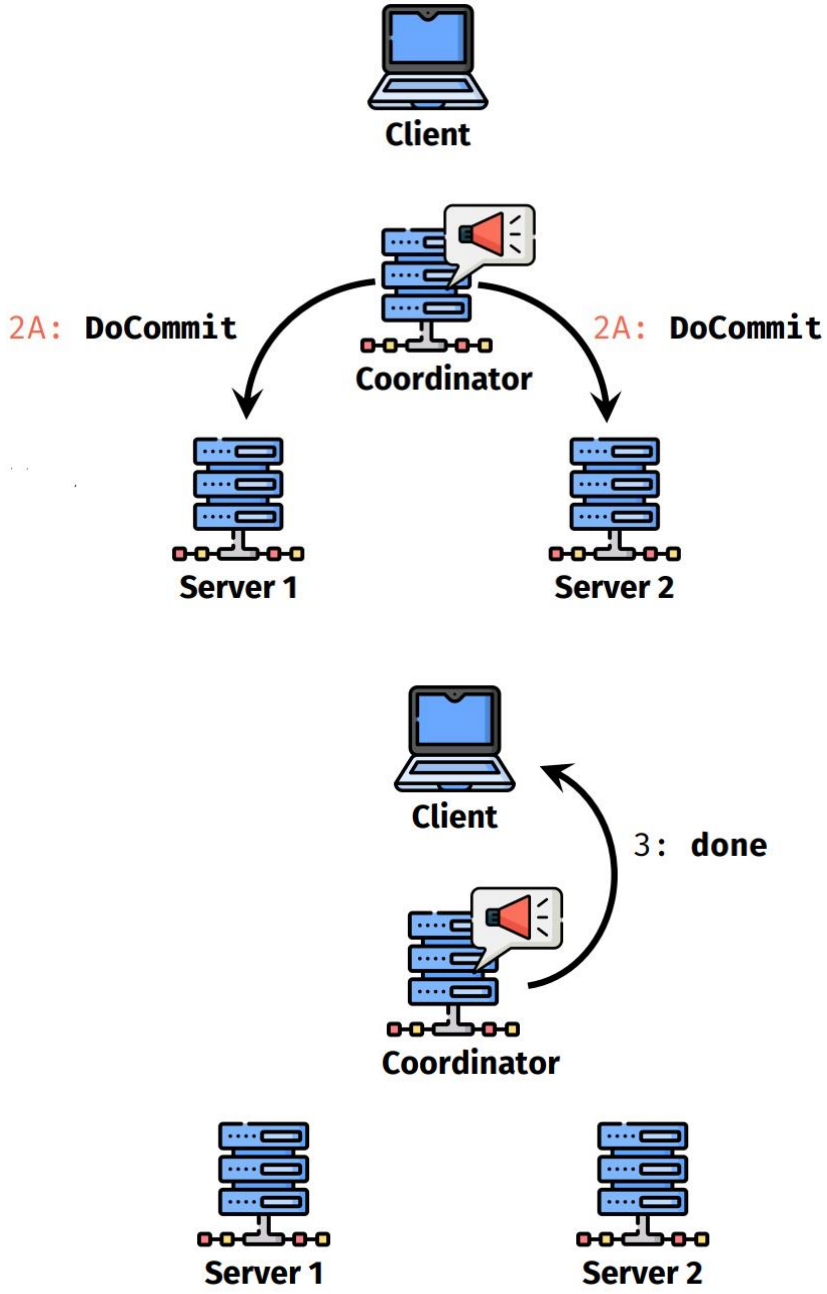




Implementing 2-Phase Commit

- Messages in first phase
  - 1A: Coordinator sends **CanCommit?** Message to participants
  - 1B Participants respond: **VoteCommit** or **VoteAbort**

- Messages in the second phase
  - 2A: All VotedCommit: , Coordinator sends **DoCommit**
  - If any VoteAbort: abort transaction. Coordinator sends **DoAbort** to everyone => release locks



Timeout and Failure Cases (1)

- Coordinator times out after CanCommit?
  - Hasn't sent any commit messages, safely abort
  - Conservative. Why?
  - Preserve correctness, sacrifice performance
- Participant times out after VoteAbort
  - Can safely abort unilaterally.
  - Why?

Timeout and Failure Cases (2)

- Participant times out after VoteCommit
  - Are unilateral decisions possible? Commit, Abort?
  - Participant could wait forever

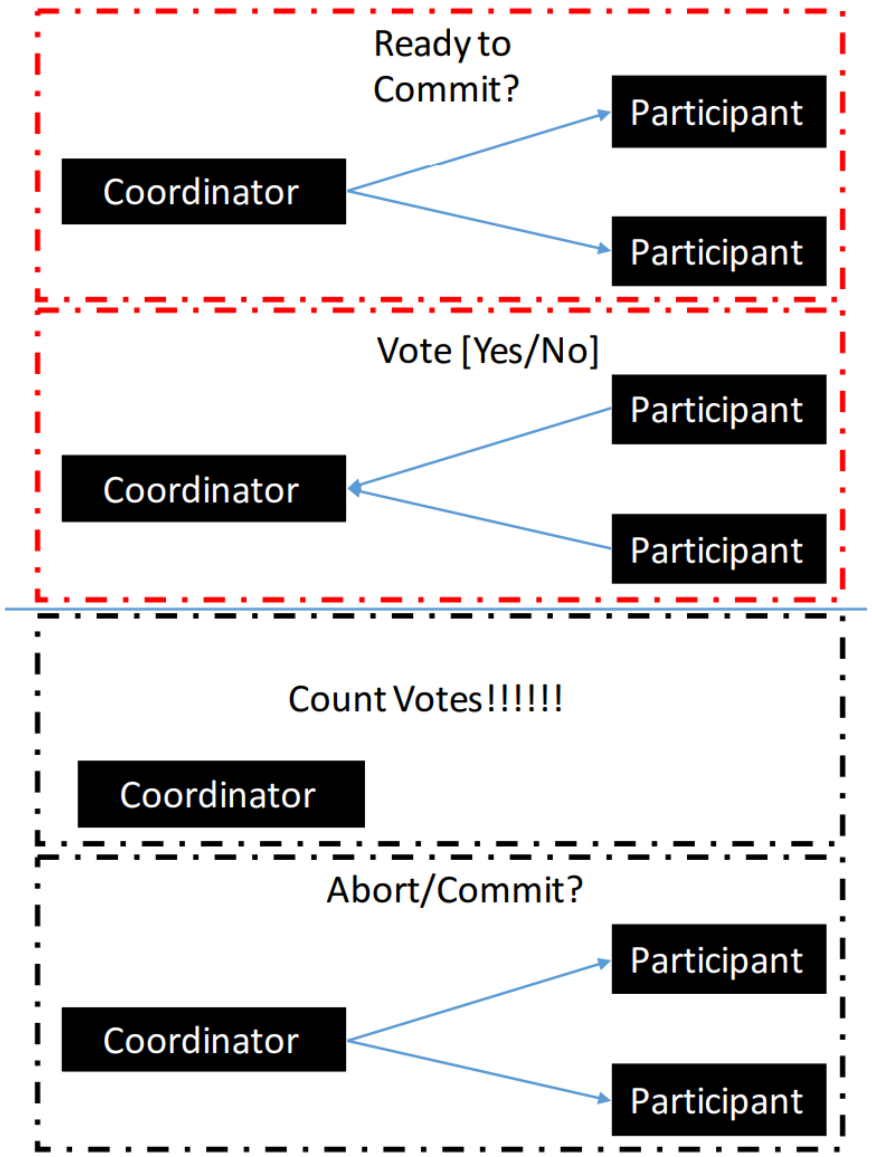
- Solution: ask another participant (gossip protocol)
  - Learn coordinator's decision: do the same
  - Other participant hasn't voted: abort is safe. Why?
  - Coordinator has not made decision
  - No reply or other participant also VoteCommit: wait
  - 2PC is "blocking protocol" → 3PC (not covered).

2-Phase Commit

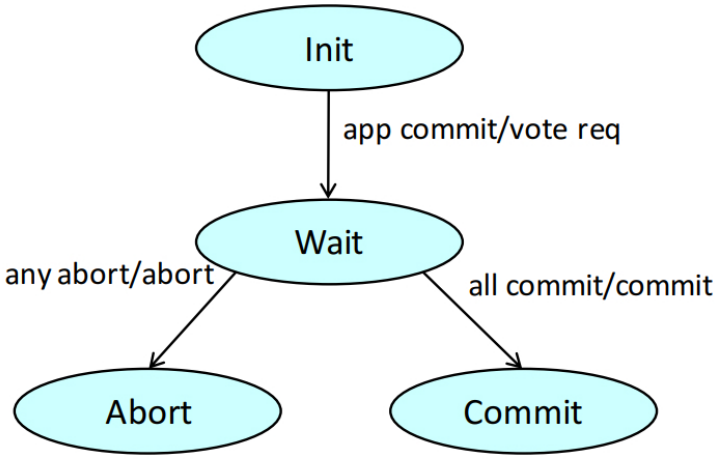
- Phase 1:
  - Coordinator sends request for votes
  - Participants vote

- Phase 2:
  - Coordinator counts votes
  - Coordinator informs of transaction status

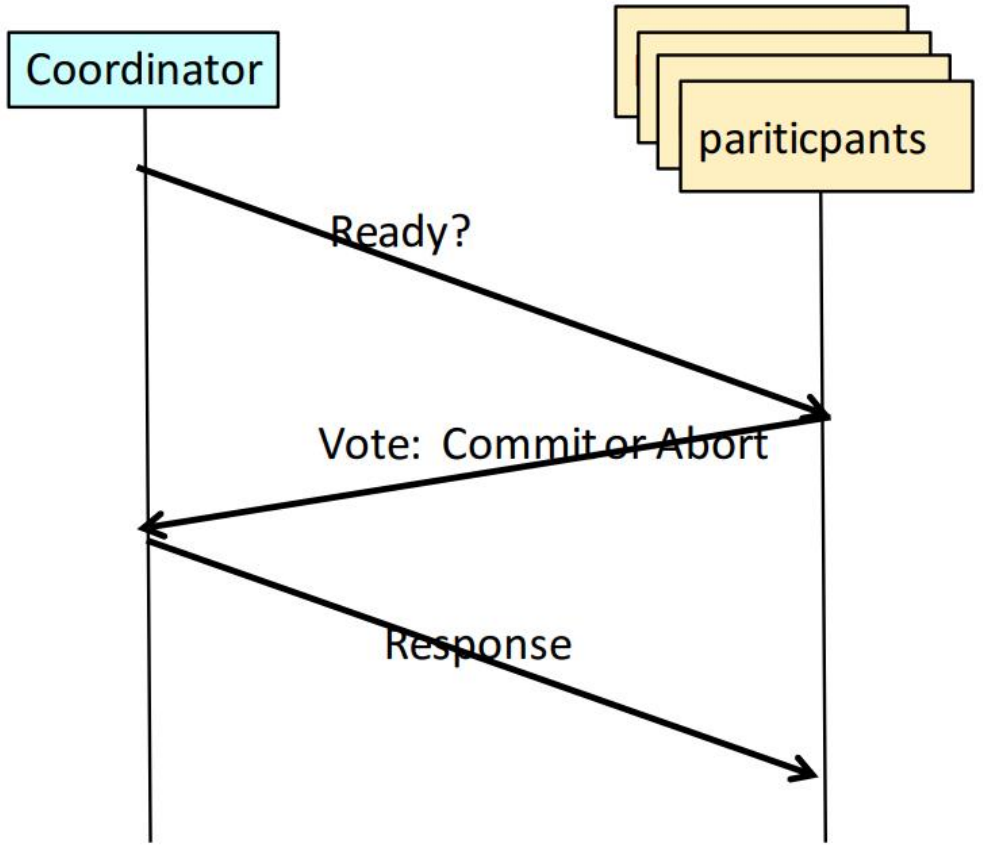
- At the end of Phase 2: either all participants commit or all abort



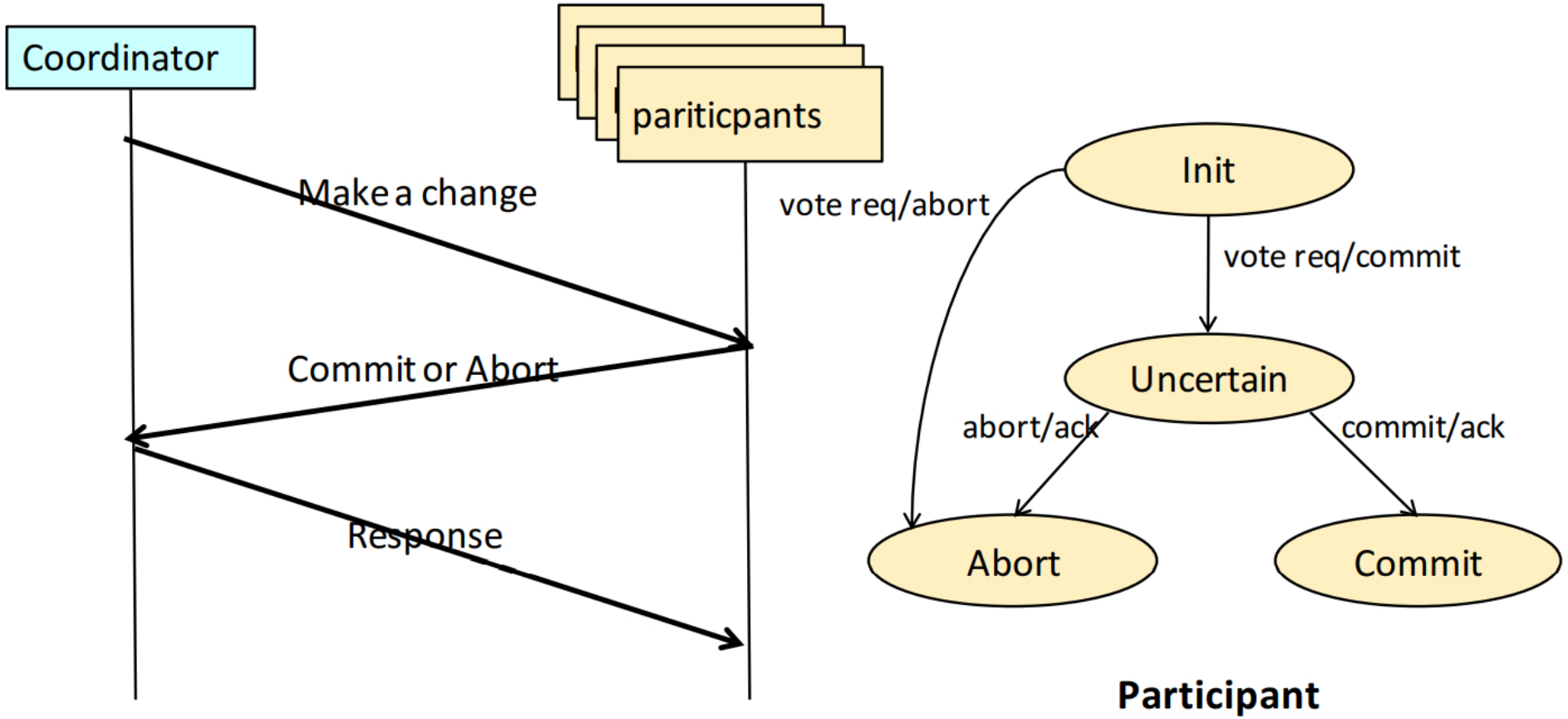
Coordinator State Diagram



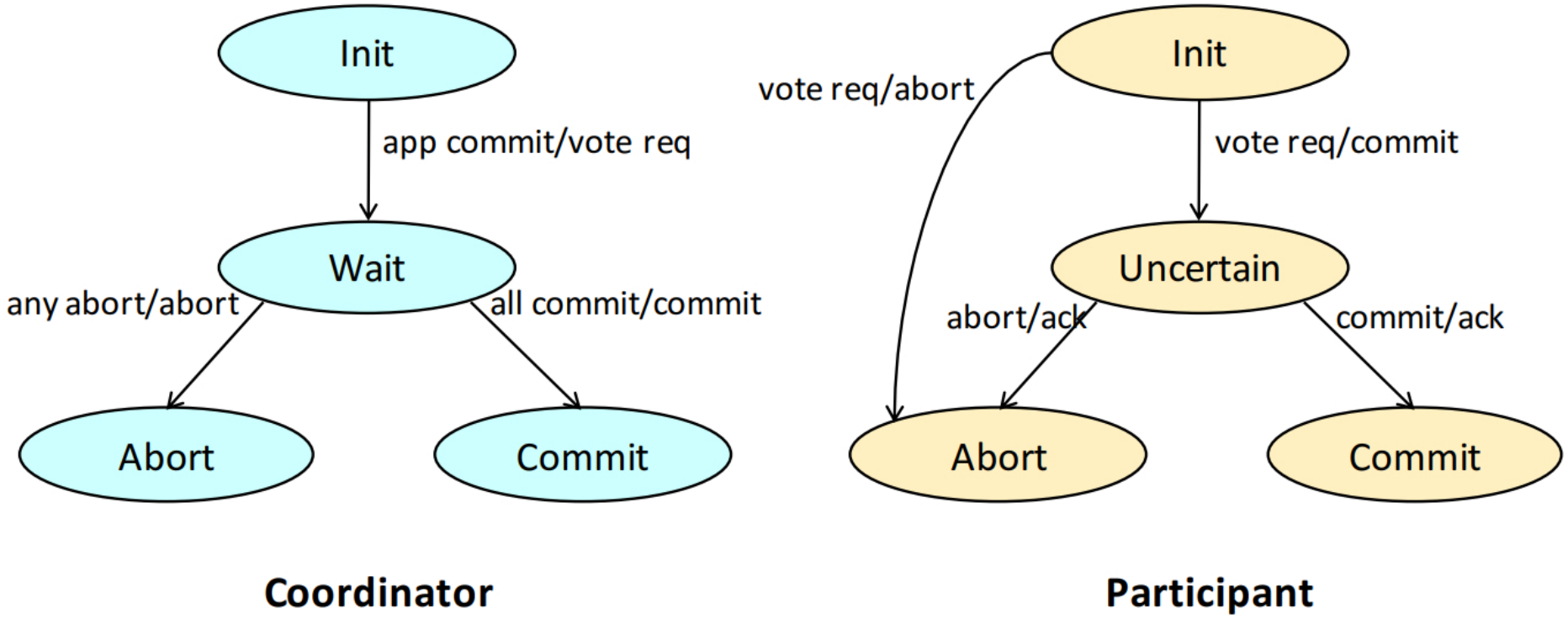
Coordinator



Participant State Diagram



State Diagrams for Two Phase Commit

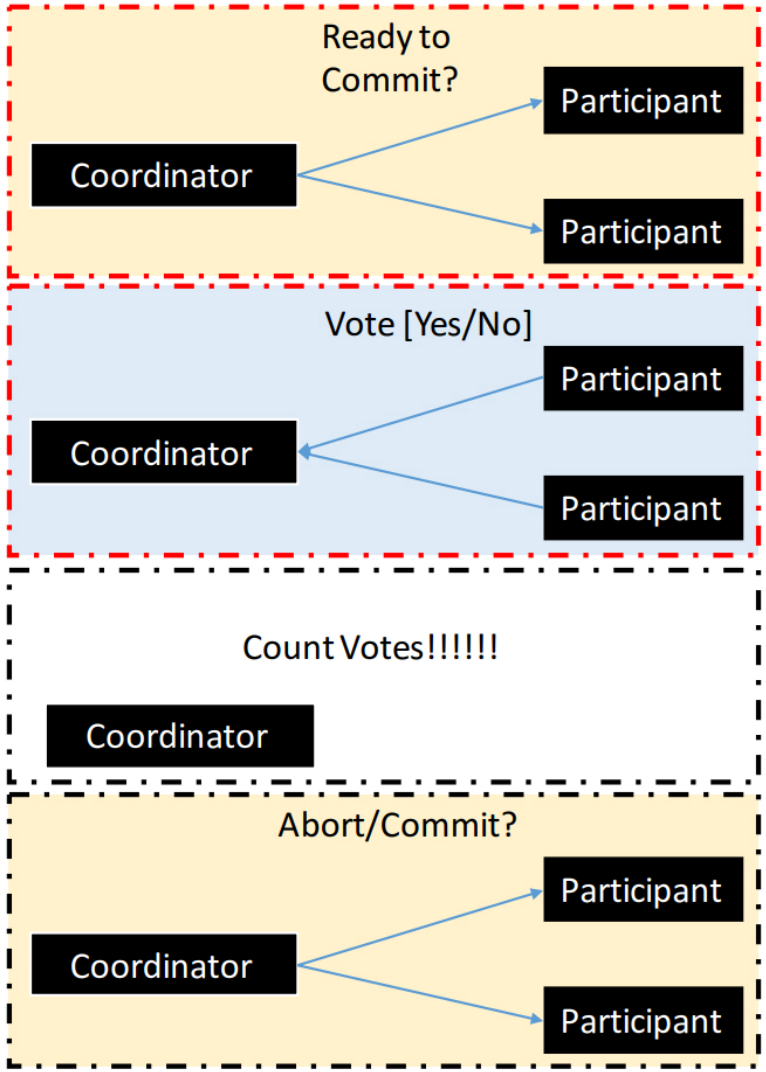


Two Phase Commit With Failures

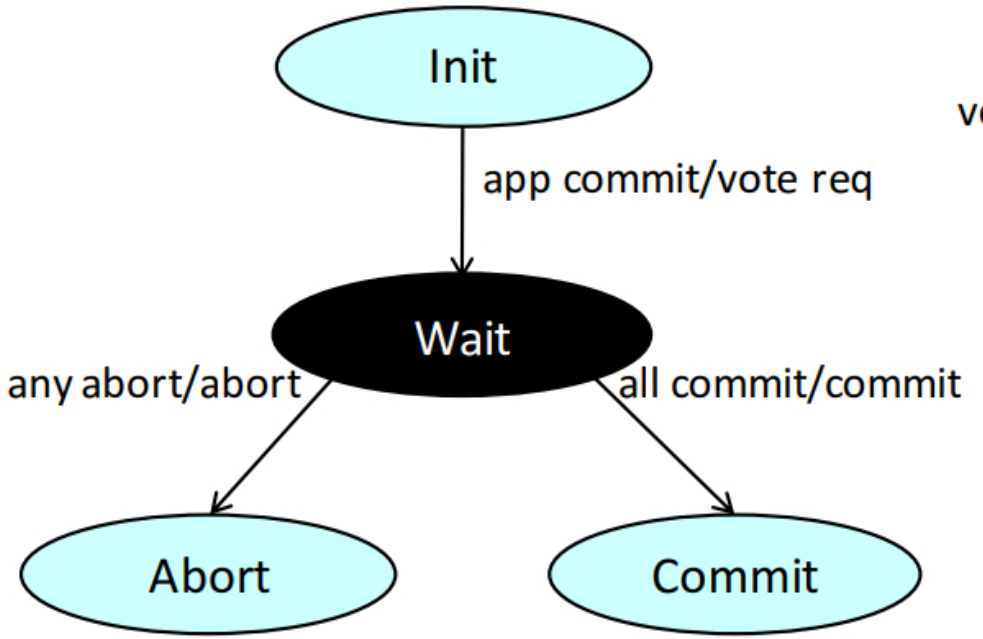
- What is the impact of failures on 2PC?
- 2PC is synchronous
  - Failure == node failure or network failure
  - Failure --> the protocol blocks/stalls

Coordinator fails, participants will be uncertain (waiting)

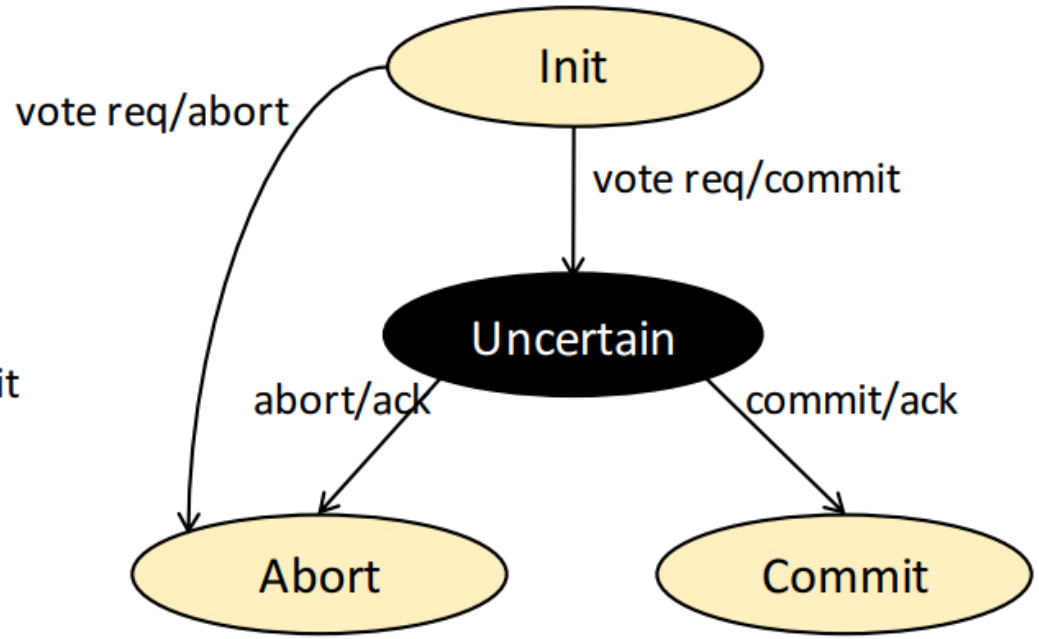
Participant fails, coordinator will be waiting



Crash Points



Coordinator

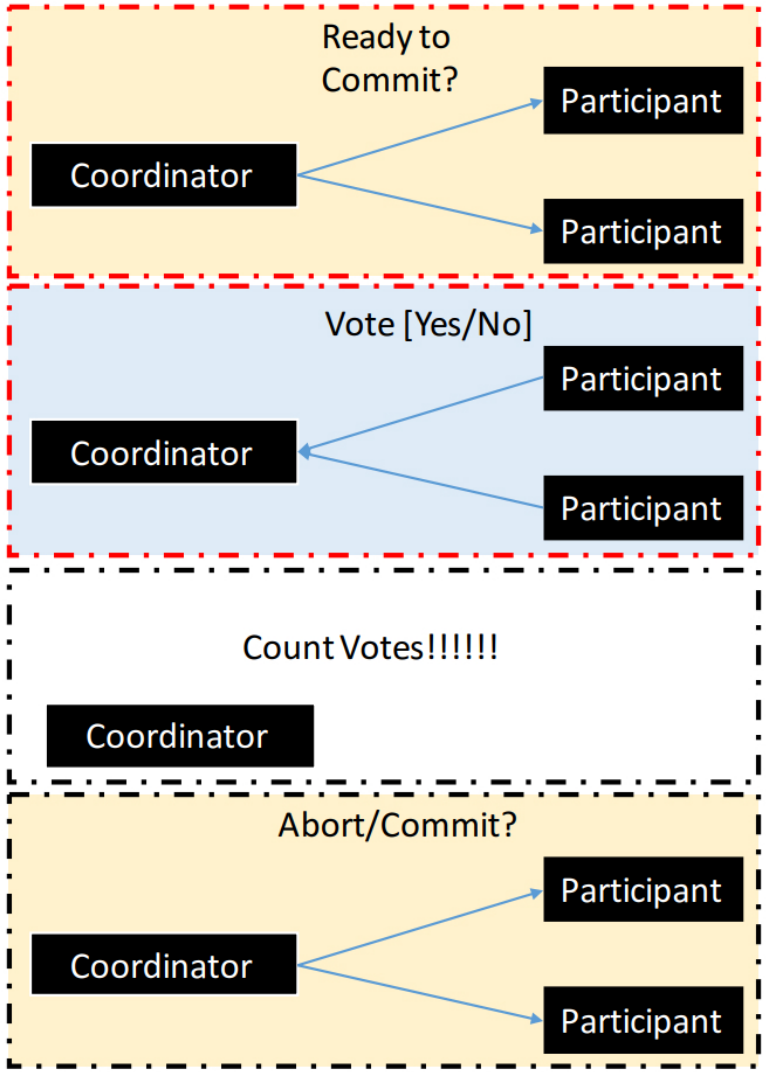


Participant

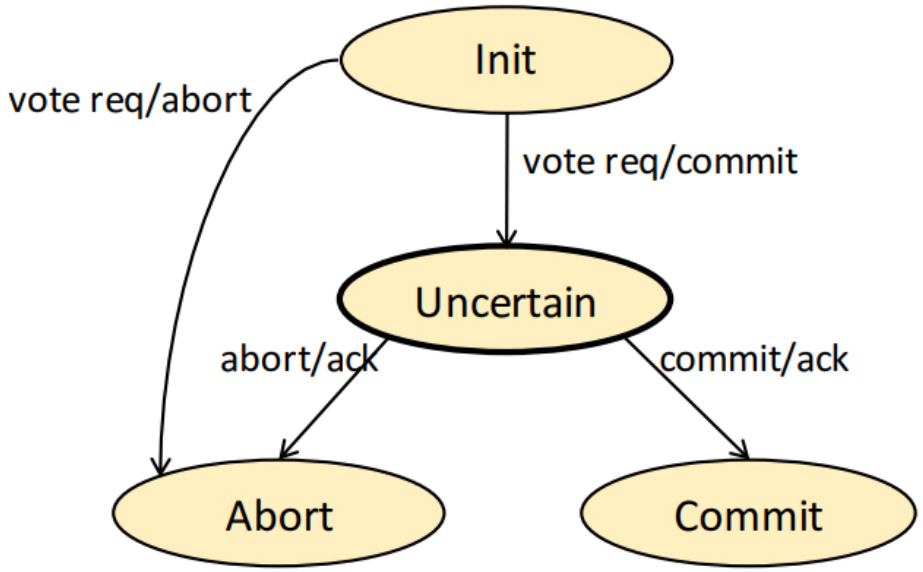
Two Phase Commit With Failures

- What is the impact of failures on 2PC?
- 2PC is synchronous
  - Failure == node failure or network failure
  - Failure --> the protocol blocks/stalls

- Detect failure using Timeouts
  - Coordinator detects participant failure and assumes ABORT à Transaction terminates
  - Participant detects Coordinator failure
  - Why Can't Participant automatically ABORT?



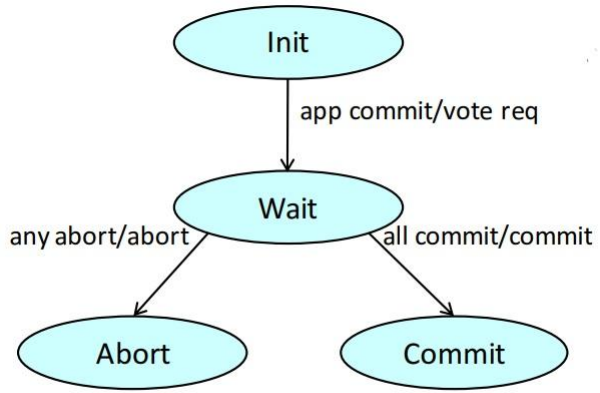
Participant Recovery from Coordinator Failure



Participant

- **Participant in Uncertain state**
  - **waiting for coordinator to say "commit" or "abort"**
- **It detects failure of coordinator**
  - **Using timeout**
- **Participant in *Uncertain* state**
  - **can't assume either outcome**
    - **BAD things happen if participant makes wrong assumptions**
  - **waits for coordinator to restart**
    - **On restart contact coordinator for final outcome**

Coordinator Recovery from Participant Failure

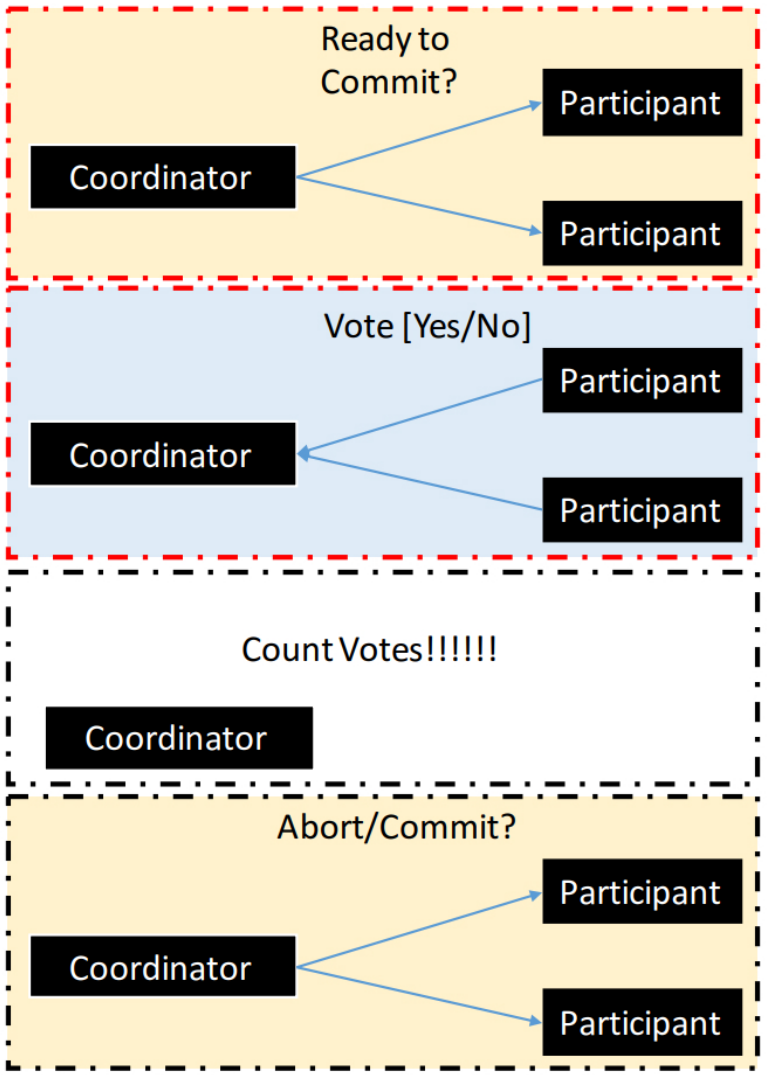


Coordinator

- **Coordinator in wait state**
  - **waiting for participant to say "commit" or "abort"**
- **It detects failure of participant**
  - **Using timeout**
- **Coordinator assumes they said 'no'**
  - **Takes no response as an abort**
  - **Abort transaction!!**
- **If participant Fails, Coordinator can make progress**

Two Phase Commit With Failures

- Detect failure using Timeouts
  - Coordinator detects participant failure and assumes ABORT a transaction terminates
  - Participant detects Coordinator failure
  - The participant must wait for coordinator
  - The transaction is stalled!



Properties of 2-Phase Commit

- Correctness
  - Neither can commit unless both agreed to commit
- Performance
  - 3N messages per transaction
- How to handle failure?
  - Timeouts à performance bad in case of failure!

Deadlocks and Livelocks

- Distributed deadlock

- Cyclic dependency of locks by transactions across servers
  - Q: How can this happen?

- In 2PC this can happen if participants unable to respond to voting request
  - e.g. still waiting on a lock on its local resource.
  - Q: How can we fix this?

- Handled with a timeout. Participants times out, then votes to abort. Retry transaction again.
  - Addresses the deadlock concern
  - However, danger of LIVELOCK – keep trying!

Summary

- Distributed consistency management
- ACID Properties desirable
- Single Server case: use locks + 2-phase locking (**strict 2PL**), transactional support for locks
- Multiple server distributed case: use **2-phase commit** for distributed transactions. Need a coordinator to manage messages from participants
- 2PC can become a performance bottleneck