Concurrency Control 3

Instructor: Matei Zaharia

Outline

What makes a schedule serializable?

Conflict serializability

Precedence graphs

Enforcing serializability via 2-phase locking
- » Shared and exclusive locks
- » Lock tables and multi-level locking

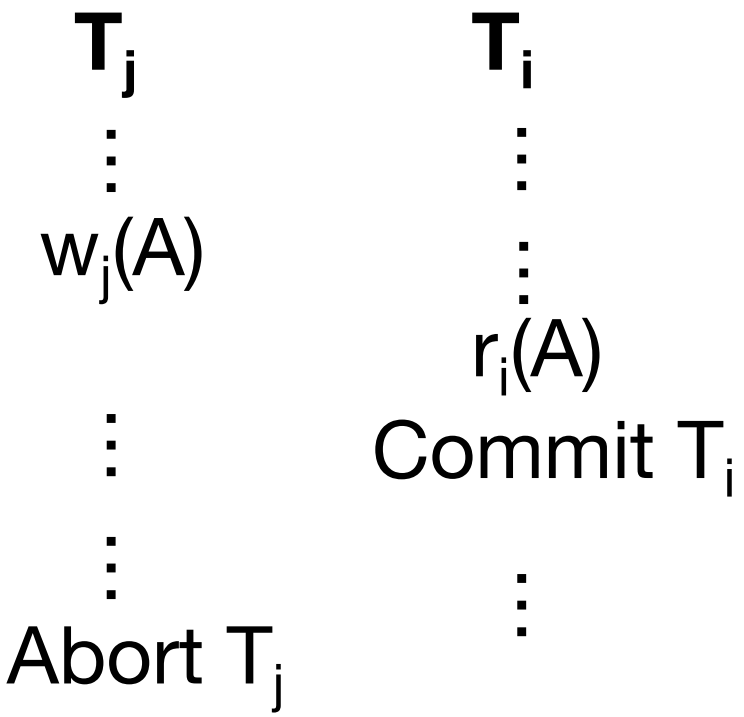
Optimistic concurrency with validation

Concurrency control + recovery

Beyond serializability

Concurrency Control & Recovery

Example:

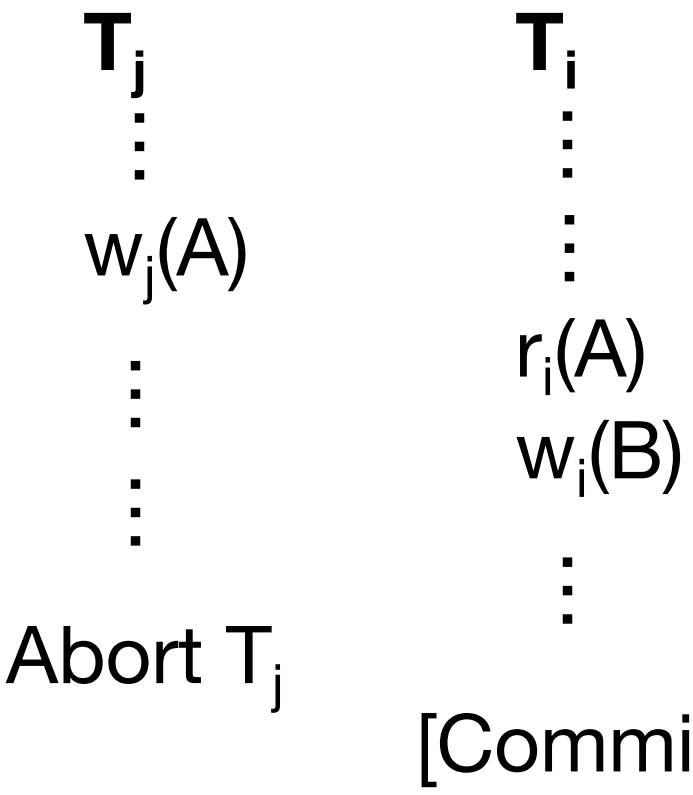


Non-persistent commit (bad!)

avoided by *recoverable* schedules

Concurrency Control & Recovery

Example:



Cascading rollback (bad!)

avoided by *avoids-cascading -rollback (ACR)* schedules

Core Problem

Schedule is conflict serializable

$T_j \longrightarrow T_i$

But not recoverable

To Resolve This

Need to mark the “final” decision for each transaction in our schedules:

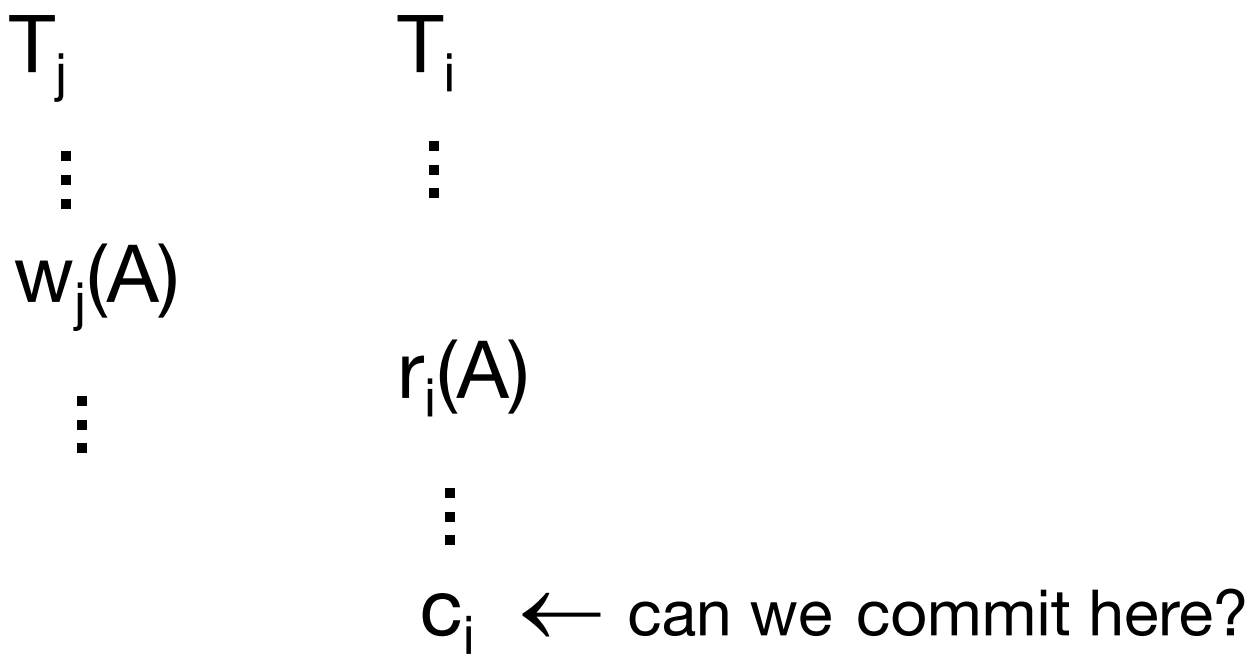
- **Commit decision:** system guarantees transaction will or has completed
- **Abort decision:** system guarantees transaction will or has been rolled back

Model This as 2 New Actions:

c_i = transaction T_i commits

a_i = transaction T_i aborts

Back to Example



Definition

T_i **reads from** T_j in S ($T_j \Rightarrow_S T_i$) if:

1. $w_j(A) <_S r_i(A)$
2. $a_j \not<_S r(A)$ ($\not<_S$: does not precede)
3. If $w_j(A) <_S w_k(A) <_S r_i(A)$ then $a_k <_S r_i(A)$

Definition

Schedule S is **recoverable** if

whenever $T_j \Rightarrow_S T_i$ and $j \neq i$ and $c_i \in S$

then $c_j <_S c_i$

Notes

In all transactions, reads and writes must precede commits or aborts

⇔ If $c_i \in T_i$, then $r_i(A) < c_i$, $w_i(A) < c_i$

⇔ If $a_i \in T_i$, then $r_i(A) < a_i$, $w_i(A) < a_i$

Also, just one of c_i, a_i per transaction

How to Achieve Recoverable Schedules?

With 2PL, Hold Write Locks Until Commit ("Strict 2PL")

T_j	T_i
$w_j(A)$	⋮
⋮	⋮
c_j	⋮
$u_j(A)$	⋮
⋮	$r_i(A)$

With Validation, No Change!

Each transaction's validation point is its commit point, and only write after

Definitions

S is **recoverable** if each tx commits only after all txs from which it read have committed

S **avoids cascading rollback** if each tx may read only values written by committed txs

S is **strict** if each tx may read and write only items previously written by committed txs

(≡ strict 2PL)

Relationship of Recoverable, ACR & Strict Schedules

Examples

Recoverable:

$w_1(A)\ w_1(B)\ w_2(A)\ r_2(B)\ c_1\ c_2$

Avoids Cascading Rollback:

$w_1(A)\ w_1(B)\ w_2(A)\ c_1\ r_2(B)\ c_2$

Strict:

$w_1(A)\ w_1(B)\ c_1\ w_2(A)\ r_2(B)\ c_2$

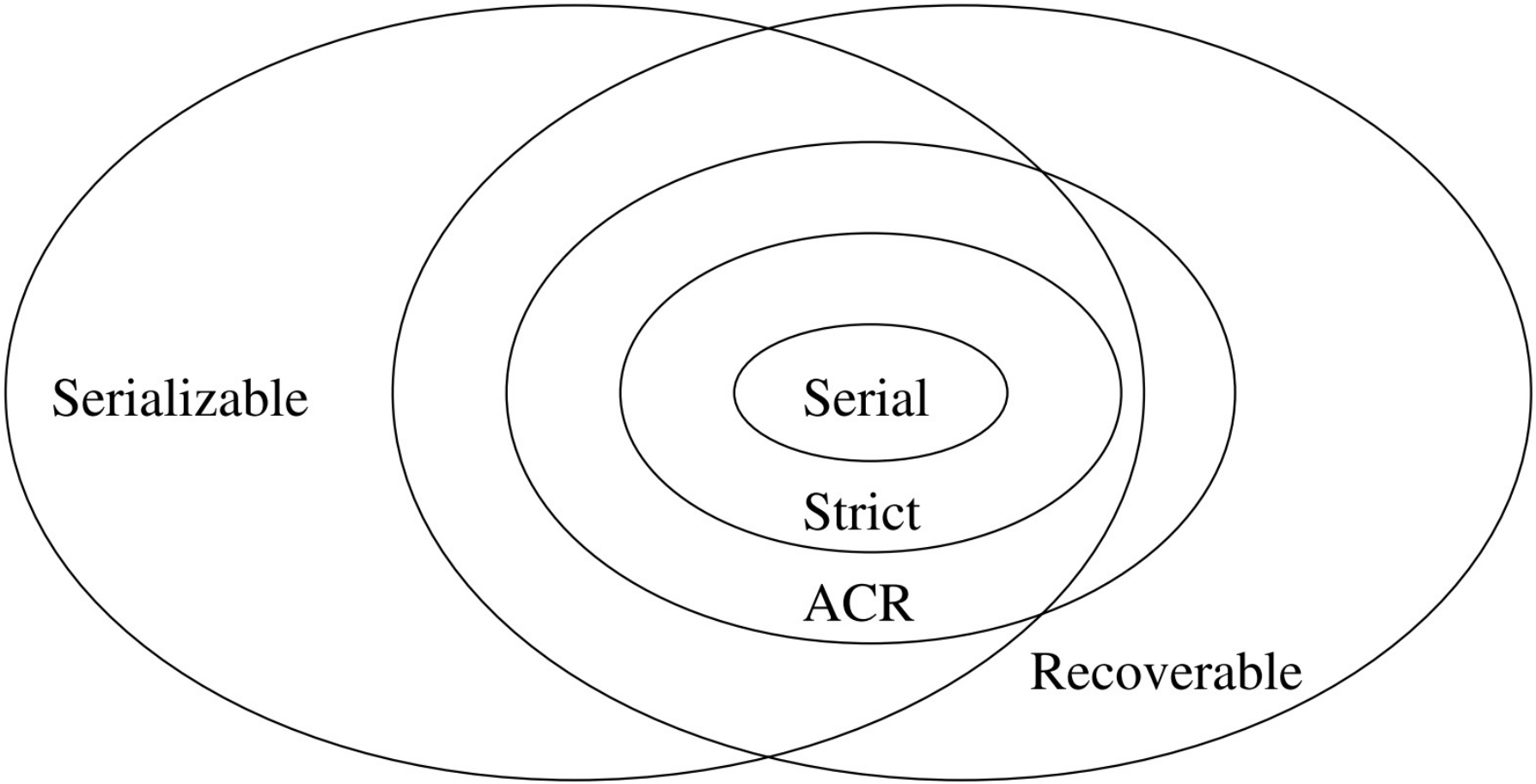
Recoverability & Serializability

Every strict schedule is serializable

Proof: equivalent to serial schedule based on the order of commit points

- » Only read/write from previously committed transactions

Recoverability & Serializability



Outline

What makes a schedule serializable?

Conflict serializability

Precedence graphs

Enforcing serializability via 2-phase locking
- » Shared and exclusive locks
- » Lock tables and multi-level locking

Optimistic concurrency with validation

Concurrency control + recovery

Beyond serializability

Weaker Isolation Levels

Dirty reads: Let transactions read values written by other uncommitted transactions

» Equivalent to having long-duration write locks, but no read locks

Read committed: Can only read values from committed transactions, but they may change

» Equivalent to having long-duration write locks (X) and short-duration read locks (S)

Weaker Isolation Levels

Repeatable reads: Can only read values from committed transactions, and each value will be the same if read again

» Equivalent to having long-duration read & write locks (X/S) but not table locks for insert

Remaining problem: phantoms!

Weaker Isolation Levels

Snapshot isolation: Each transaction sees a consistent snapshot of the whole DB (as if we saved all committed values when it began)

- » Often implemented with multi-version concurrency control (MVCC)

Still has some anomalies! Example?

Weaker Isolation Levels

Snapshot isolation: Each transaction sees a consistent snapshot of the whole DB (as if we saved all committed values when it began)

- » Often implemented with multi-version concurrency control (MVCC)

Write skew anomaly: txs write different values

- » Constraint: $A+B \geq 0$
- » T_1: read A, B; if $A+B \geq 1$, subtract 1 from A
- » T_2: read A, B; if $A+B \geq 1$, subtract 1 from B
- » Problem: what if we started with A=1, B=0?

Interesting Fact

Oracle calls its snapshot isolation level “serializable”, and doesn’t have the real thing!

Distributed Databases

Instructor: Matei Zaharia

Why Distribute Our DB?

Store the same data item on multiple nodes to survive node failures (**replication**)

Divide data items & work across nodes to increase scale, performance (**partitioning**)

Related reasons:

» Maintenance without downtime

» Elastic resource use (don’t pay when unused)

Outline

Replication strategies

Partitioning strategies

Atomic commitment & 2PC

CAP

Avoiding coordination

Parallel query execution

Outline

Replication strategies

Partitioning strategies

Atomic commitment & 2PC

CAP

Avoiding coordination

Parallel query execution

Replication

General problems:

» How to tolerate server failures?

» How to tolerate network failures?

The Eight Fallacies of Distributed Computing

Peter Deutsch

Essentially everyone, when they first build a distributed application, makes the following eight assumptions. All prove to be false in the long run and all cause *big* trouble and *painful* learning experiences.

1. The network is reliable
2. Latency is zero
3. Bandwidth is infinite
4. The network is secure
5. Topology doesn't change
6. There is one administrator
7. Transport cost is zero
8. The network is homogeneous

For more details, read the article by Arnon Rotem-Gal-Oz

Replication

Store each data item on multiple nodes!

Question: how to read/write to them?

Primary-Backup

Elect one node “primary”

Store other copies on “backup”

Send requests to primary, which then forwards operations or logs to backups

Backup coordination is either:

- » Synchronous (write to backups before acking)
- » Asynchronous (backups slightly stale)

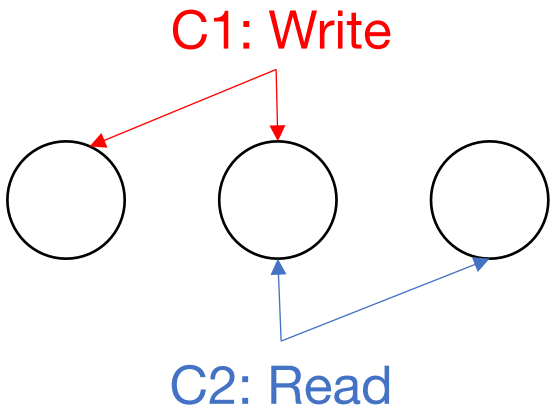
Quorum Replication

Read and write to intersecting sets of servers; no one “primary”

Common: majority quorum

» More exotic ones exist, like grid quorums

Surprise: primary-backup is a quorum too!



What If We Don’t Have Intersection?

What If We Don't Have Intersection?

Alternative: "eventual consistency"

- » If writes stop, eventually all replicas will contain the same data
- » Basic idea: asynchronously broadcast all writes to all replicas

When is this acceptable?

How Many Replicas?

In general, to survive F fail-stop failures, we need F+1 replicas

Question: what if replicas fail arbitrarily? Adversarially?

What To Do During Failures?

Cannot contact primary?

What To Do During Failures?

Cannot contact primary?

- » Is the primary failed?
- » Or can we simply not contact it?

What To Do During Failures?

Cannot contact majority?

- » Is the majority failed?
- » Or can we simply not contact it?

Solution to Failures

Traditional DB: page the DBA

Distributed computing: use **consensus**

- » Several algorithms: Paxos, Raft
- » Today: many implementations
  - Apache Zookeeper, etcd, Consul
- » Idea: keep a reliable, distributed shared record of who is “primary”

Consensus in a Nutshell

Goal: distributed agreement

» On one value or on a log of events

Participants broadcast votes [for each event]

» If a majority of notes ever accept a vote v, then they will eventually choose v

» In the event of failures, retry that round

» Randomization greatly helps!

Take CS 244B for more details

What To Do During Failures?

Cannot contact majority?

- » Is the majority failed?
- » Or can we simply not contact it?

Consensus can provide an answer!

- » Although we may need to stall…
- » (more on that later)

Replication Summary

Store each data item on multiple nodes!

Question: how to read/write to them?

- » Answers: primary-backup, quorums
- » Use consensus to agree on operations or on system configuration

Outline

Replication strategies

Partitioning strategies

Atomic commitment & 2PC

CAP

Avoiding coordination

Parallel query execution

Partitioning

General problem:

- » Databases are big!
- » What if we don’t want to store the whole database on each server?

Partitioning Basics

Split database into chunks called “partitions”

- » Typically partition by row
- » Can also partition by column (rare)

Place one or more partitions per server

Partitioning Strategies

Hash keys to servers
  » Random assignment

Partition keys by range
  » Keys stored contiguously

What if servers fail (or we add servers)?
  » Rebalance partitions (use consensus!)

Pros/cons of hash vs range partitioning?

What About Distributed Transactions?

Replication:

- » Must make sure replicas stay up to date
- » Need to reliably replicate the commit log! (use consensus or primary/backup)

Partitioning:

- » Must make sure all partitions commit/abort
- » Need **cross-partition** concurrency control!

Outline

Replication strategies

Partitioning strategies

Atomic commitment & 2PC

CAP

Avoiding coordination

Parallel query execution

Atomic Commitment

Informally: either all participants commit a transaction, or none do

“participants” = partitions involved in a given transaction

So, What's Hard?

So, What's Hard?

All the problems of consensus…

…plus, if *any* node votes to *abort*, all must decide to *abort*

» In consensus, simply need agreement on "some" value

Two-Phase Commit

Canonical protocol for atomic commitment (developed 1976-1978)

Basis for most fancier protocols

Widely used in practice

Use a transaction *coordinator*

» Usually client – not always!

Two Phase Commit (2PC)

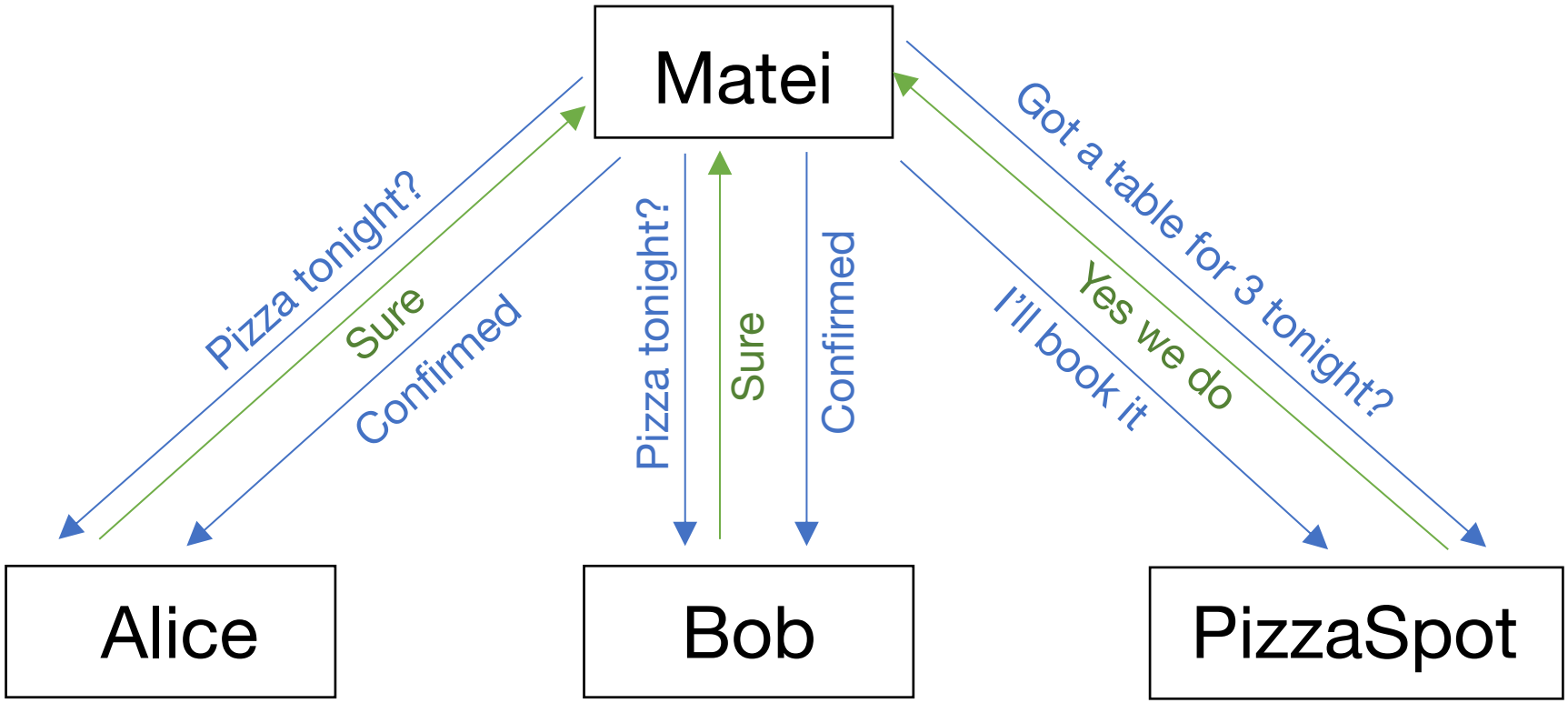
1. Transaction coordinator sends *prepare* message to each participating node

2. Each participating node responds to coordinator with *prepared* or *no*

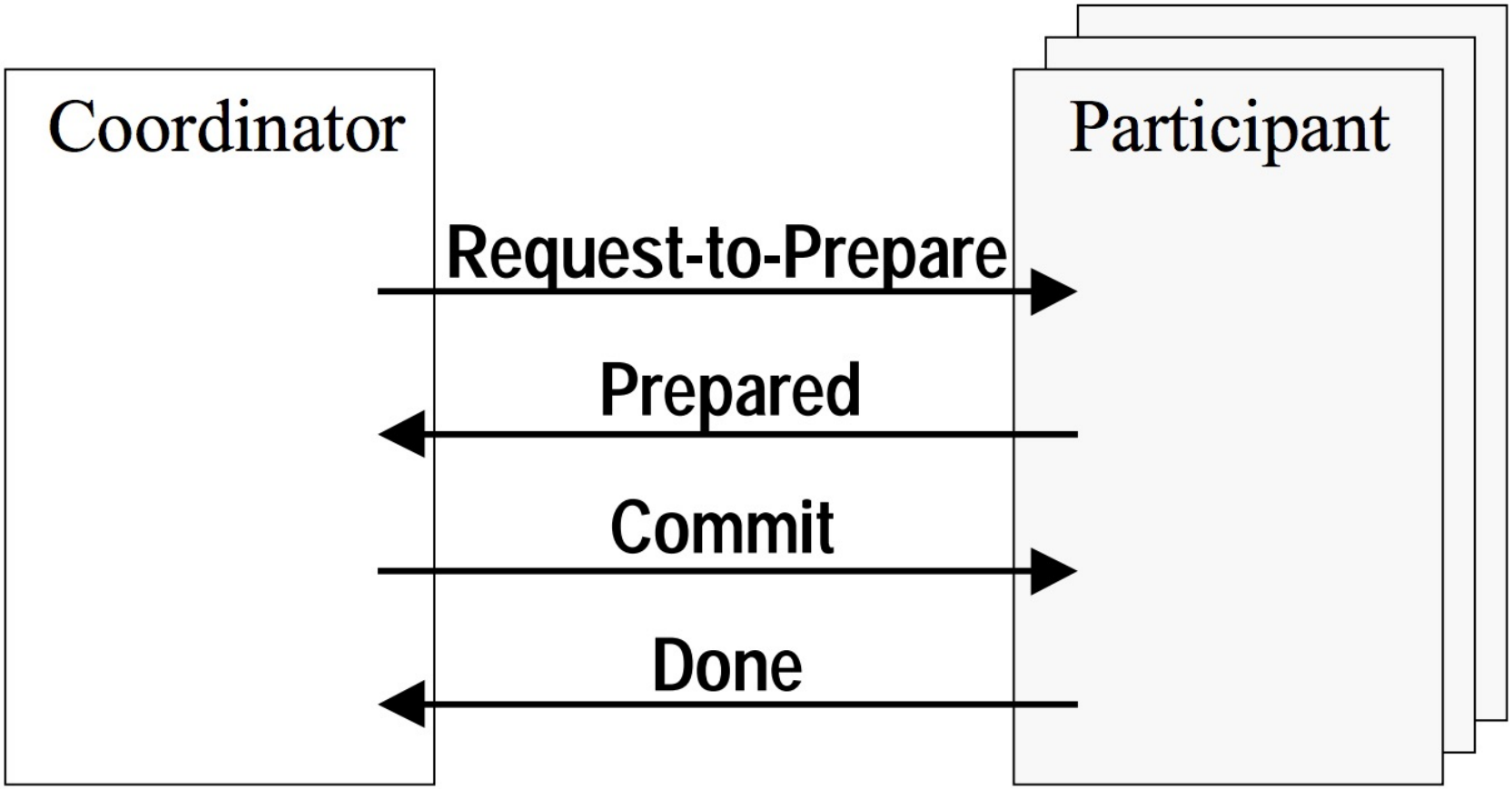
3. If coordinator receives all *prepared*:
   - » Broadcast *commit*

4. If coordinator receives any *no:*
   - » Broadcast *abort*

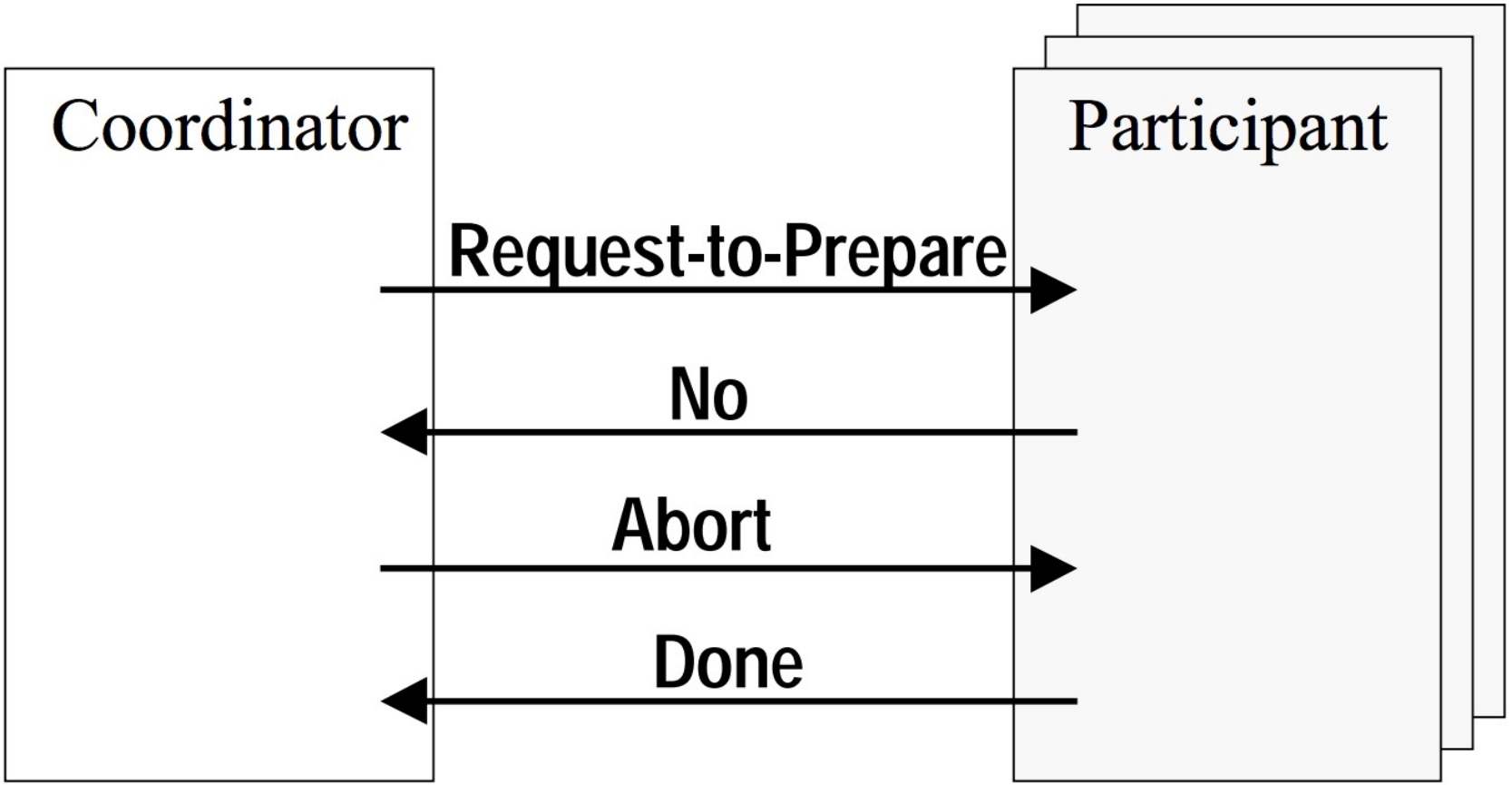
Informal Example



Case 1: Commit



Case 2: Abort



2PC + Validation

Participants perform validation upon receipt of *prepare* message

Validation essentially blocks between *prepare* and *commit* message

2PC + 2PL

Traditionally: run 2PC at commit time

» i.e., perform locking as usual, then run 2PC to have all participants agree that the transaction will commit

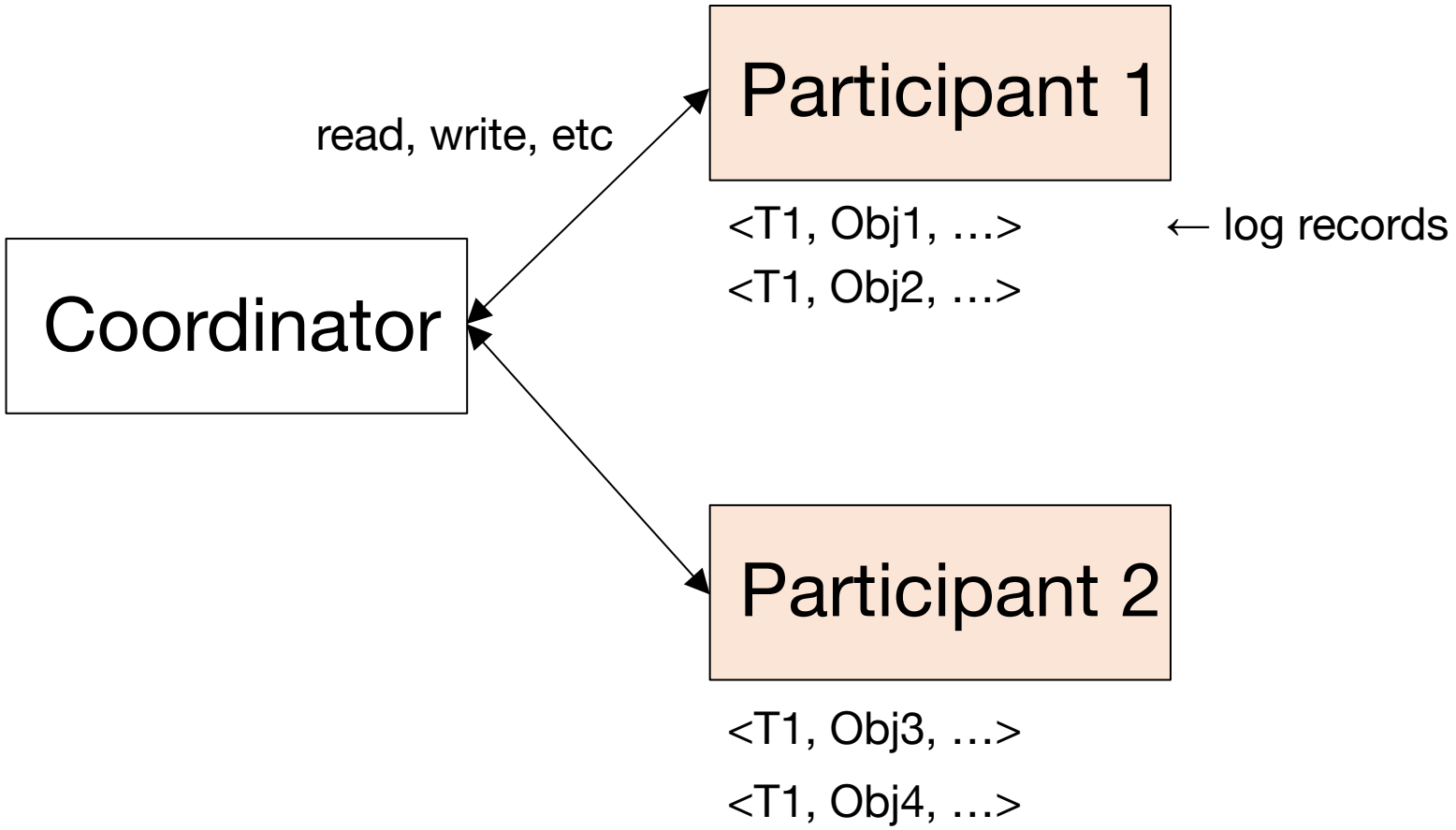
Under strict 2PL, run 2PC before unlocking the write locks

2PC + Logging

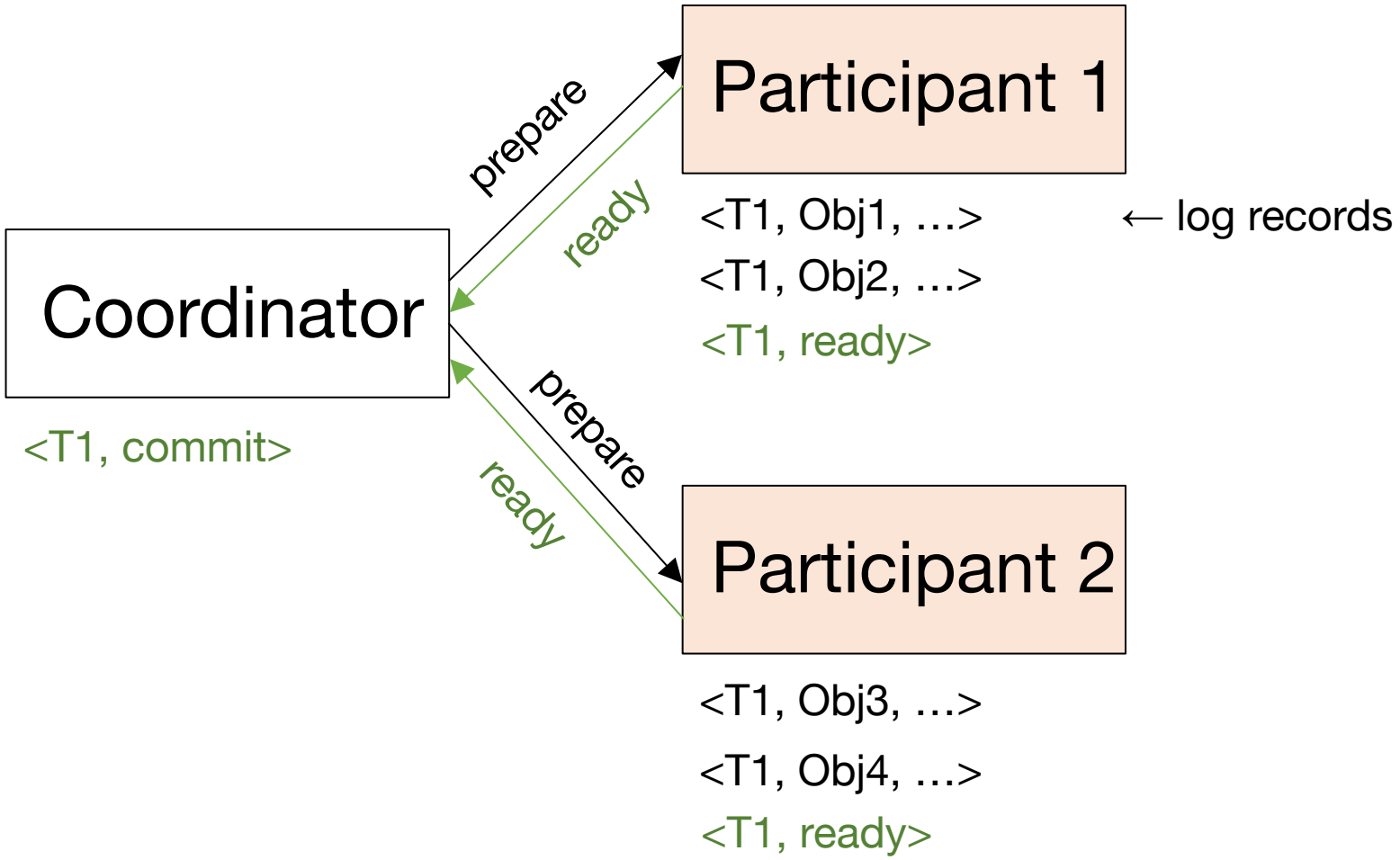
Log records must be flushed to disk on each participant before it replies to *prepare*

» The participant should log how it wants to respond + data needed if it wants to commit

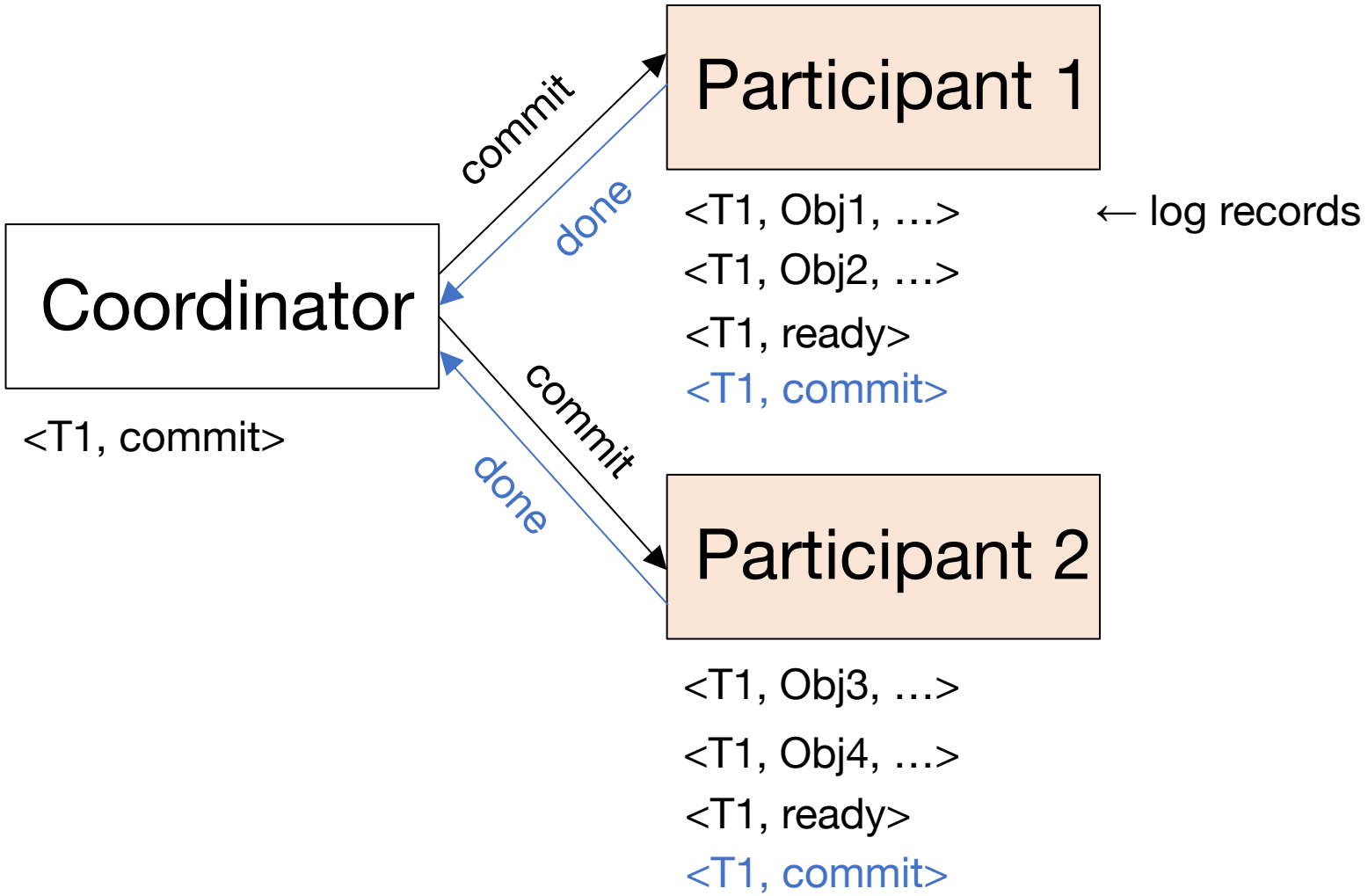
2PC + Logging Example



2PC + Logging Example



2PC + Logging Example



Optimizations Galore

Participants can send *prepared* messages to each other:

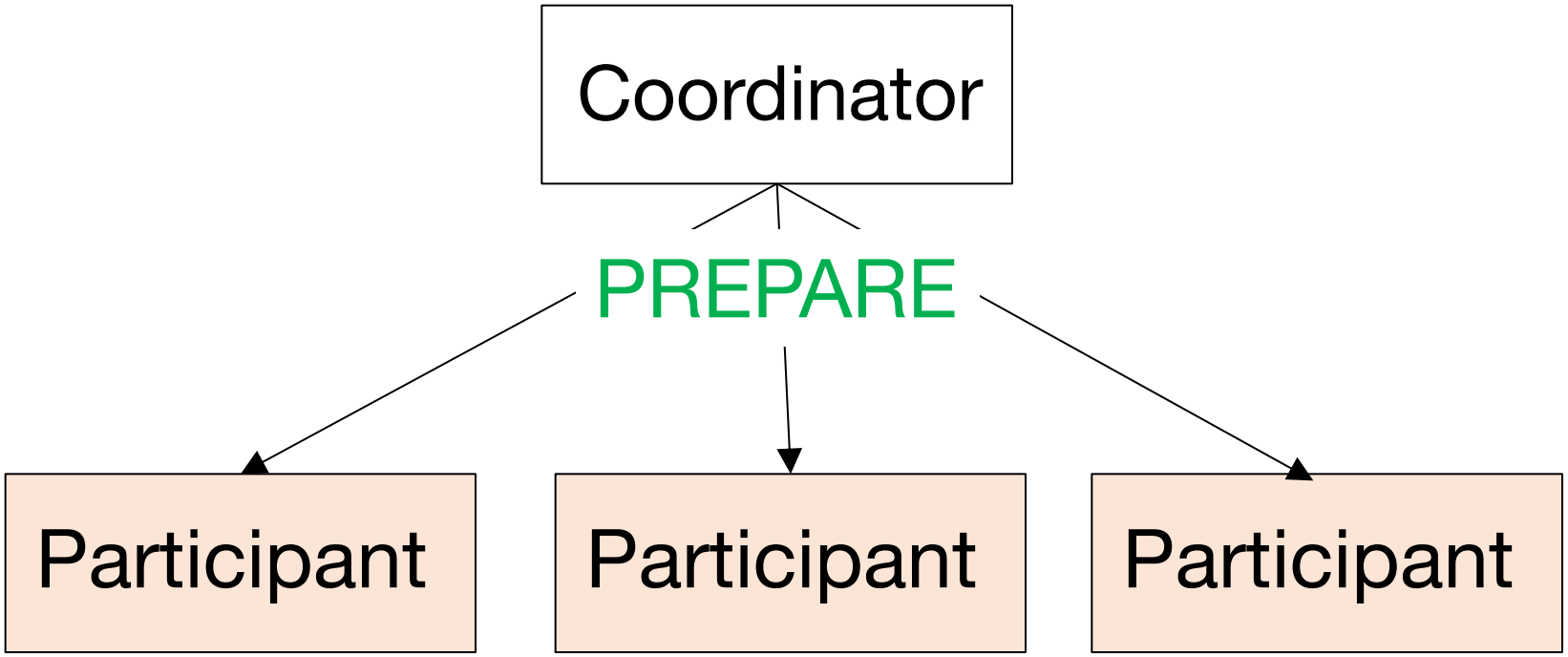
» Can commit without the client

» Requires $O(P^2)$ messages

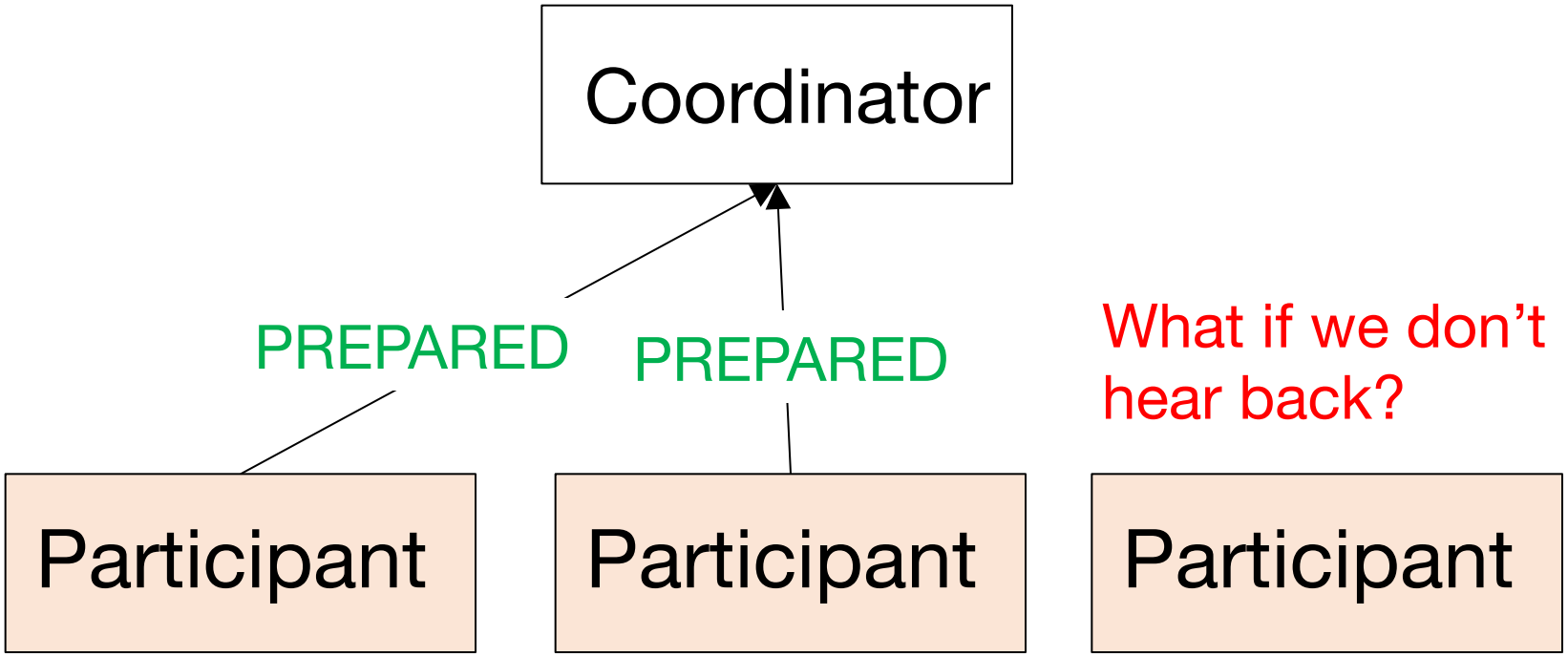
Piggyback transaction's last command on *prepare* message

2PL: piggyback lock "unlock" commands on *commit/abort* message

What Could Go Wrong?



What Could Go Wrong?



Case 1: Participant Unavailable

We don’t hear back from a participant

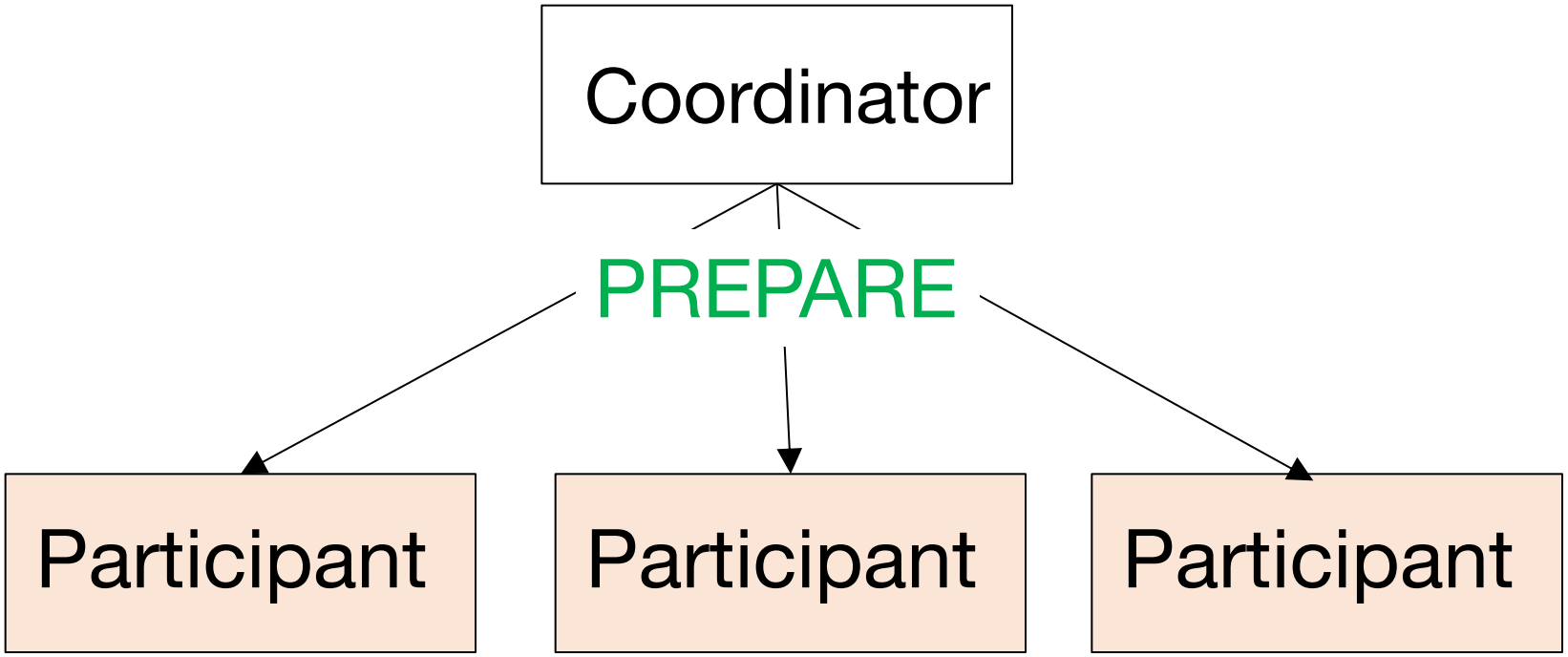
Coordinator can still decide to *abort*

» Coordinator makes the final call!

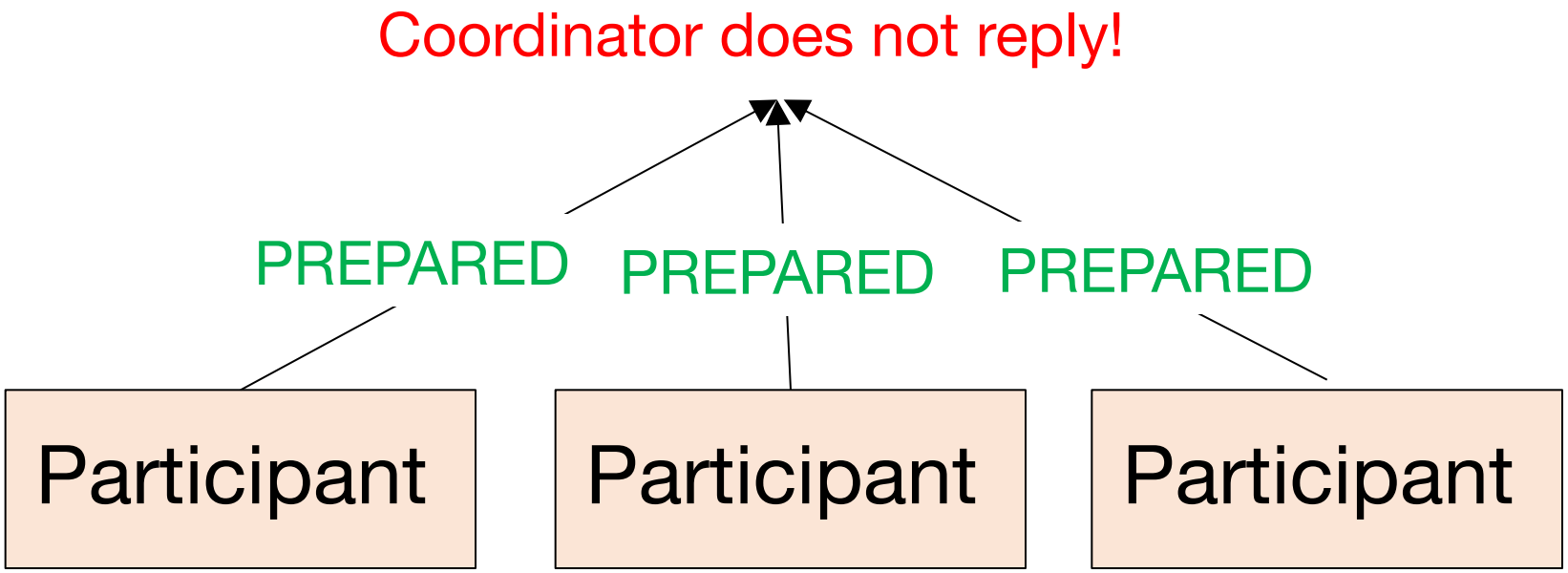
Participant comes back online?

» Will receive the *abort* message

What Could Go Wrong?



What Could Go Wrong?



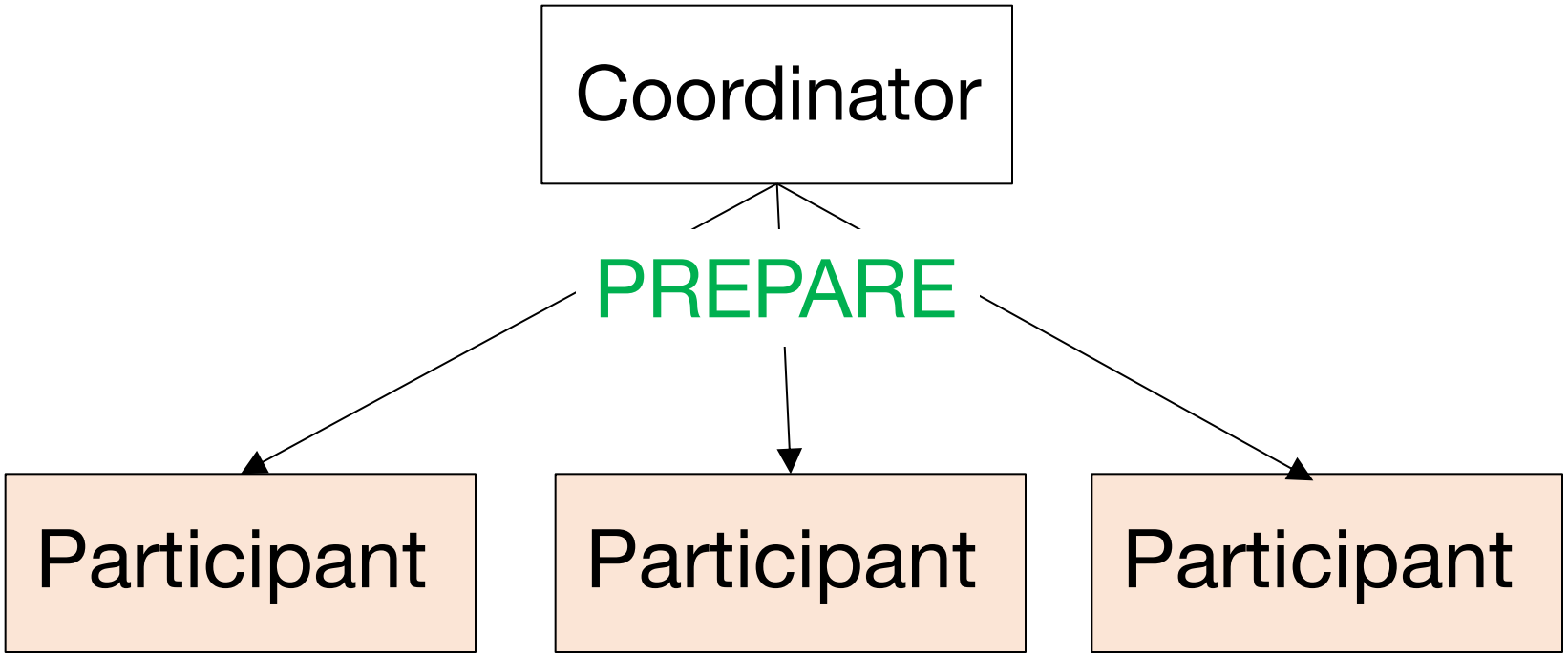
Case 2: Coordinator Unavailable

Participants cannot make progress

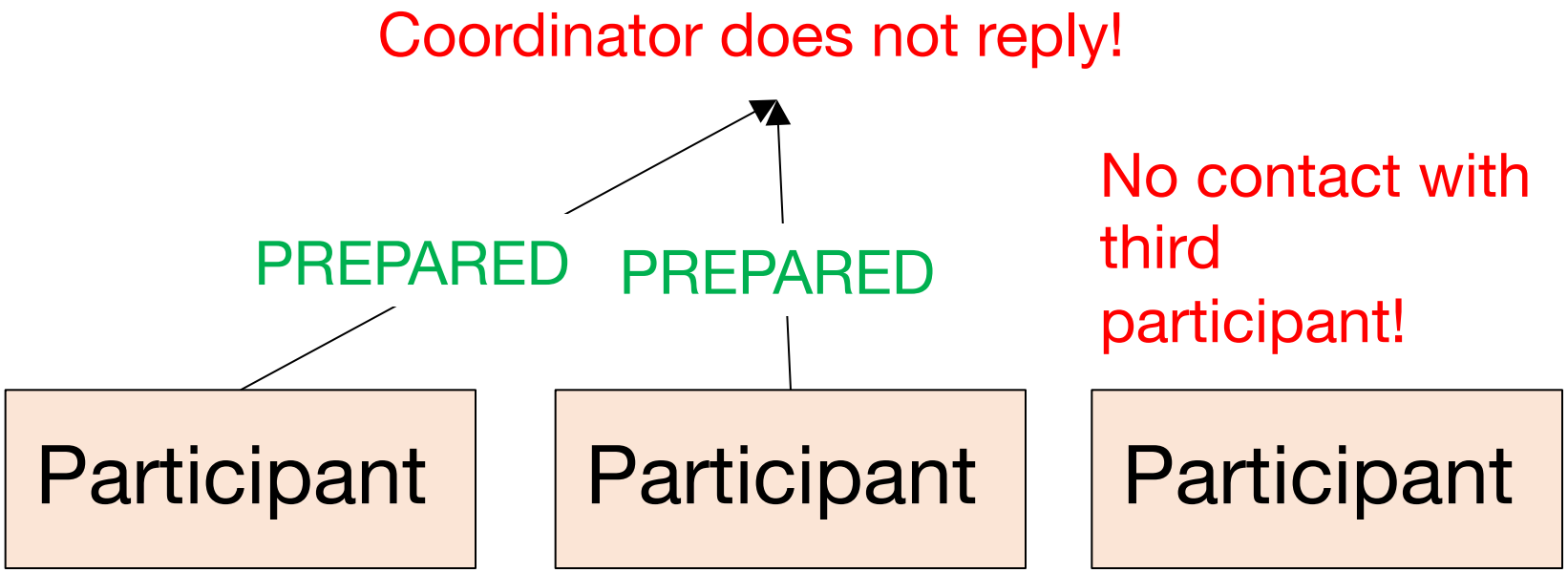
But: can agree to elect a *new* coordinator, never listen to the old one (using consensus)

- » Old coordinator comes back? Overruled by participants, who reject its messages

What Could Go Wrong?



What Could Go Wrong?



Case 3: Coordinator and Participant Unavailable

Worst-case scenario:

- » Unavailable/unreachable participant voted to *prepare*
- » Coordinator hears back all *prepare*, broadcasts *commit*
- » Unavailable/unreachable participant *commits*

Rest of participants *must* wait!!!

Other Applications of 2PC

The “participants” can be any entities with distinct failure modes; for example:

- Add a new user to database and queue a request to validate their email
- Book a flight from SFO -> JFK on United and a flight from JFK -> LON on British Airways
- Check whether Bob is in town, cancel my hotel room, and ask Bob to stay at his place

Coordination is Bad News

Every atomic commitment protocol is *blocking* (i.e., may stall) in the presence of:

- » Asynchronous network behavior (e.g., unbounded delays)
  - Cannot distinguish between delay and failure
- » Failing nodes
  - If nodes never failed, could just wait

Cool: actual theorem!

Outline

Replication strategies

Partitioning strategies

Atomic commitment & 2PC

CAP

Avoiding coordination

Parallel processing

CONTACT US | INKTOMI WORLDWIDE | TECH SUPPORT

Inktomi®

PRODUCTS | SOLUTIONS | SERVICES | CUSTOMERS | PARTNERS | COMPANY | NEWS & EVENTS

SEARCH THIS SITE GO

powered by Inktomi®

ENTERPRISE PORTALS
CUSTOMER SELF SERVICE
CALL CENTERS

Home > Solutions > Customer Self-Service

INKTOMI SOLUTIONS FOR SELF-SERVICE

The Problem

Customer satisfaction is directly related to h
answer questions.

Eric Brewer

SYSTEMS MAIN

Inktomi Files for $26 Million AOL Software Deal

Dow Jones Newswires
Updated April 16, 1998 2:06 p.m. ET

WASHINGTON -- The software concern Inktomi Corp. said Thursd
it plans to sell up to 2.2 million shares in an initial public offering o
stock that could raise between $26.4 million and $30.8 million.

Asynchronous Network Model

Messages can be arbitrarily delayed

Can’t distinguish between delayed messages and failed nodes in a finite amount of time

CAP Theorem

In an asynchronous network, a distributed database can either:

- » guarantee a response from any replica in a finite amount of time (“availability”) **OR**
- » guarantee arbitrary “consistency” criteria/constraints about data

but not both

CAP Theorem

Choose either:

» Consistency and “Partition Tolerance”

» Availability and “Partition Tolerance”

Example consistency criteria:

» Exactly one key can have value “Matei”

“CAP” is a reminder:

» No free lunch for distributed systems

Brewer's Conjecture and the Feasibility of Consistent, Available, Partition-Tolerant Web Services

Seth Gilbert and Nancy Lynch
Laboratory for Computer Science
Massachusetts Institute of Technology
Cambridge, MA 02139
sethg@mit.edu,lynch@theory.lcs.mit.edu

Abstract

When designing distributed web services, there are three properties that are commonly desired: consistency, availability, and partition tolerance. It is impossible to achieve all three. In this note, we prove this conjecture in the asynchronous network model, and then discuss solutions to this dilemma in the partially synchronous model.

1 Introduction

At PODC 2000, Brewer[1], in an invited talk [2], made the following conjecture: it is impossible for a web service to provide the following three guarantees:

- Consistency
- Availability
- Partition-tolerance

All three of these properties are desirable – and expected – from real-world web services. In this note, we will first discuss what Brewer meant by the conjecture; next we will formalize these concepts and prove the conjecture; finally, we will describe and attempt to formalize some real-world solutions to this practical difficulty.

[1]Eric Brewer is a professor at the University of California, Berkeley, and the co-founder and Chief Scientist of Inktomi.

Why CAP is Important

Pithy reminder: “consistency” (serializability, various integrity constraints) is expensive!

- » Costs us the ability to provide “always on” operation (availability)
- » Requires expensive coordination (synchronous communication) even when we don’t have failures