Distributed Databases

Instructor: Matei Zaharia

cs245.stanford.edu

Outline

Replication strategies

Partitioning strategies

AC & 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?

All the problems as 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)

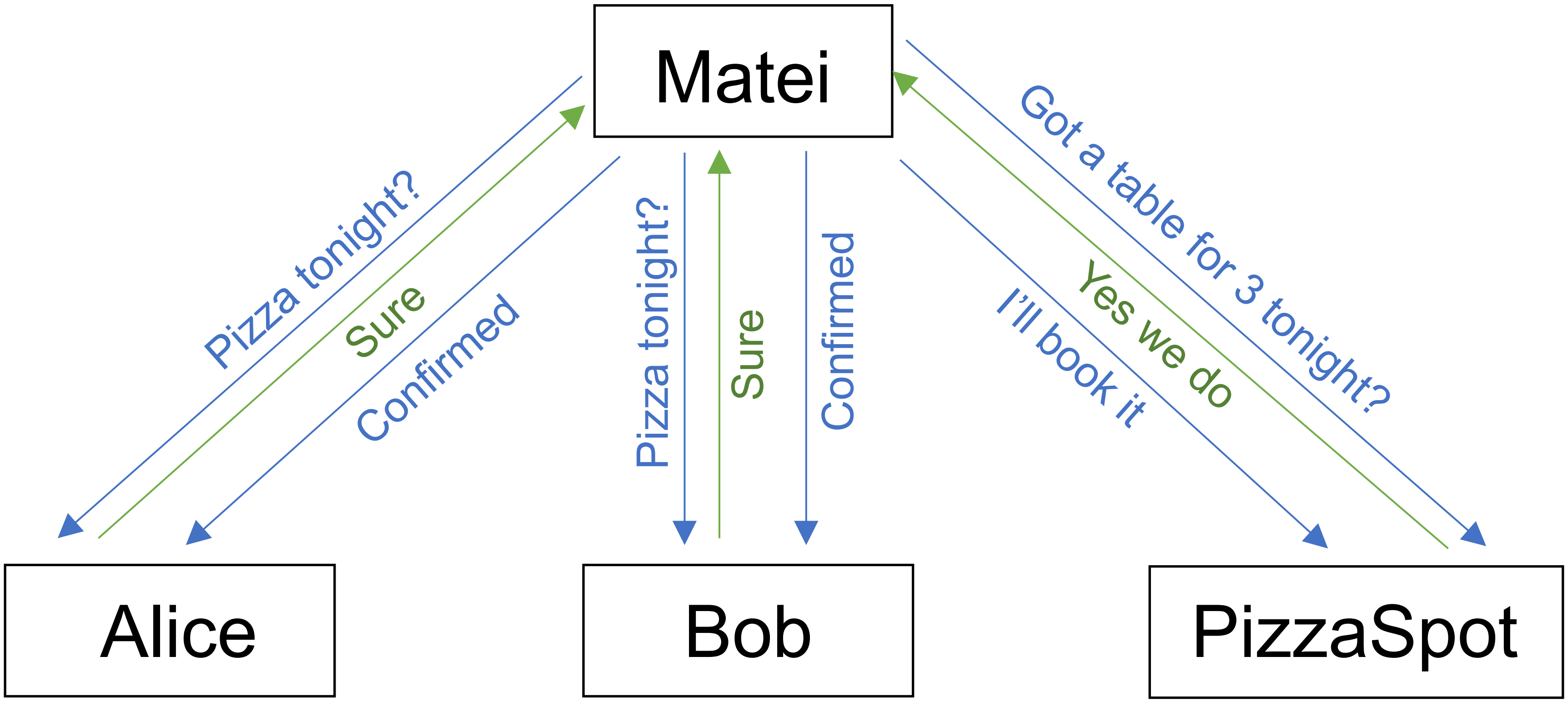
1. Transaction coordinator sends *prepare* message to each participating node

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

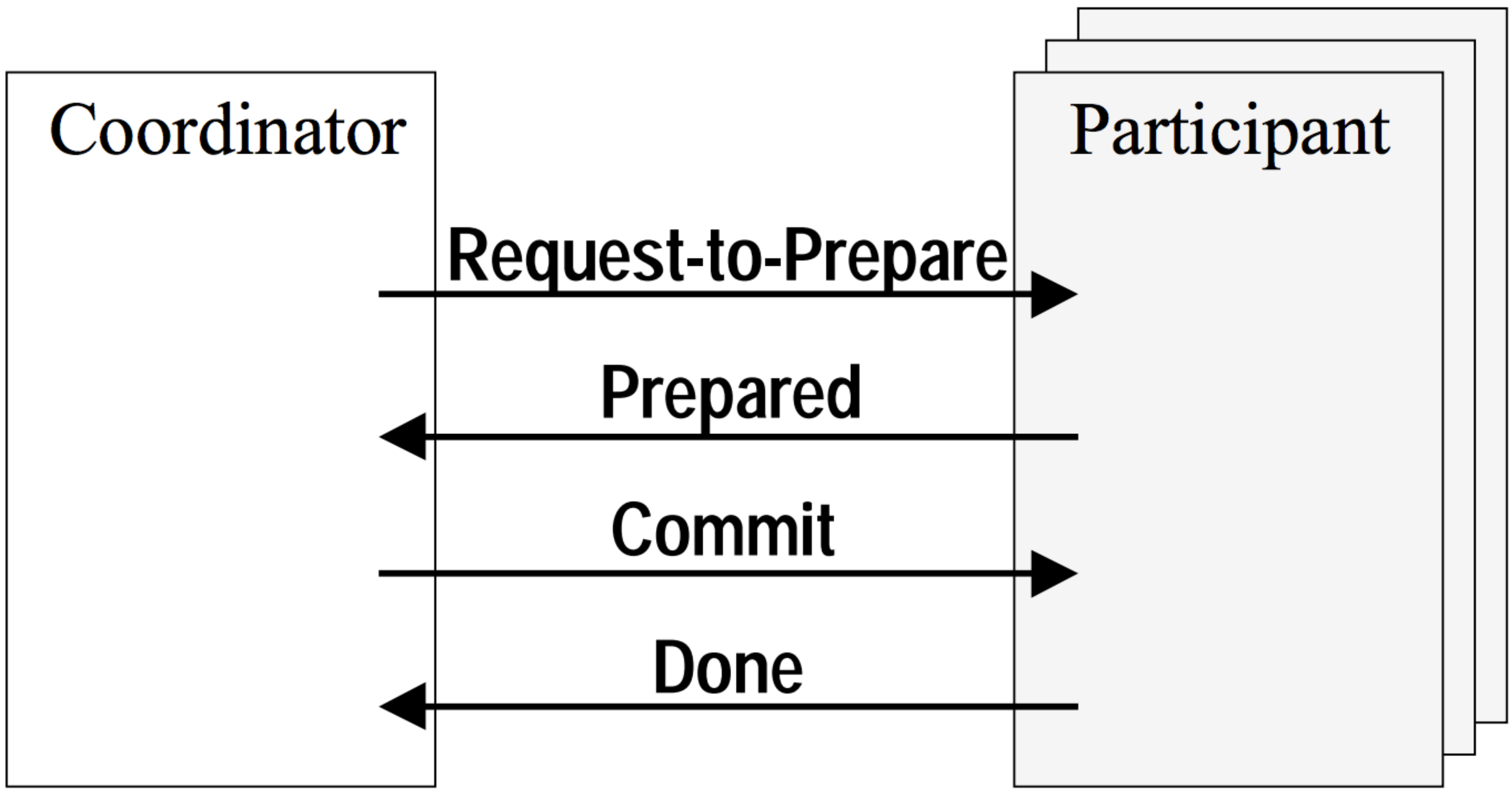
3. If coordinator receives all *prepared*:
   - » Broadcast *commit*

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

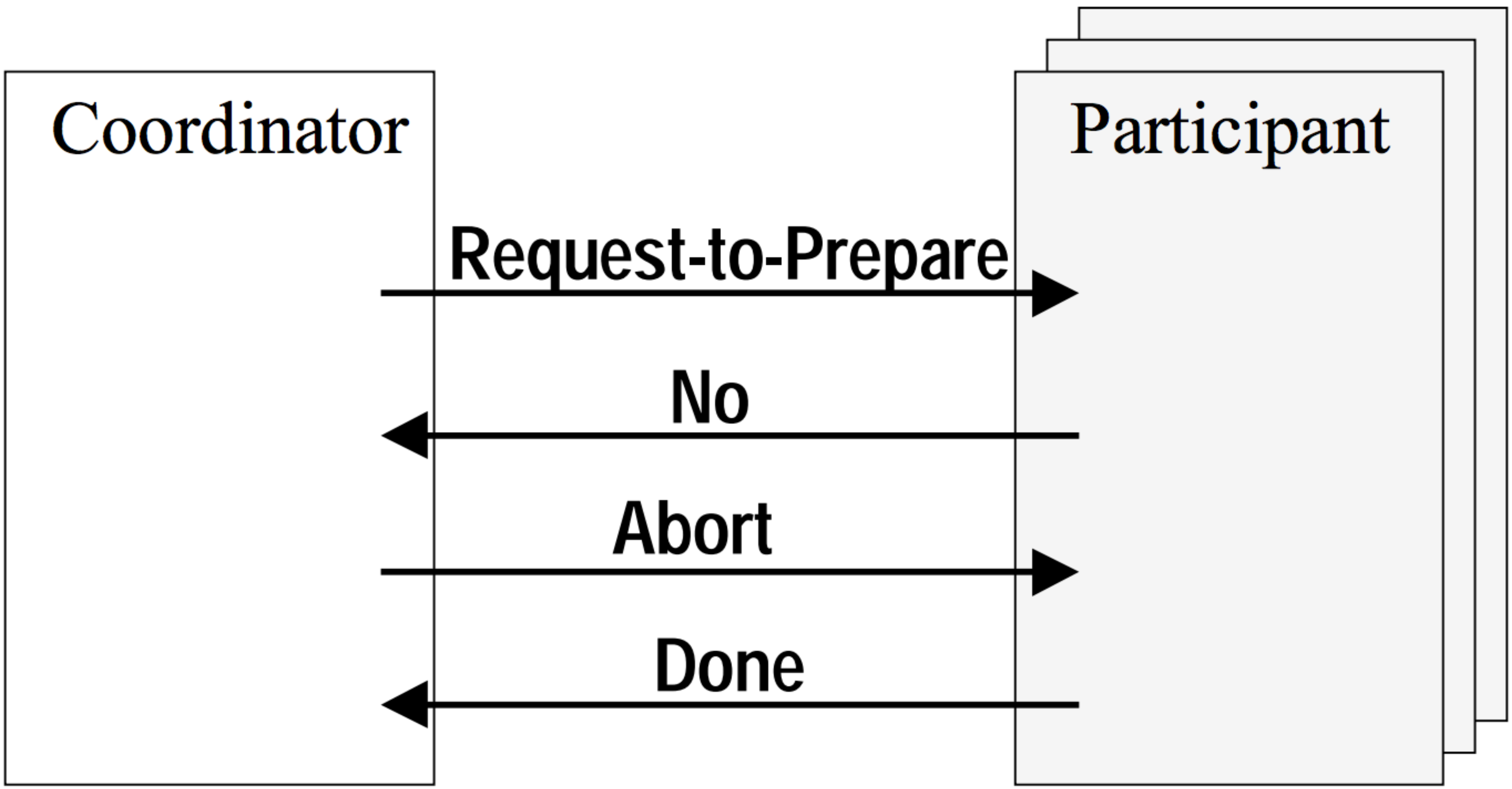
Informal Example



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

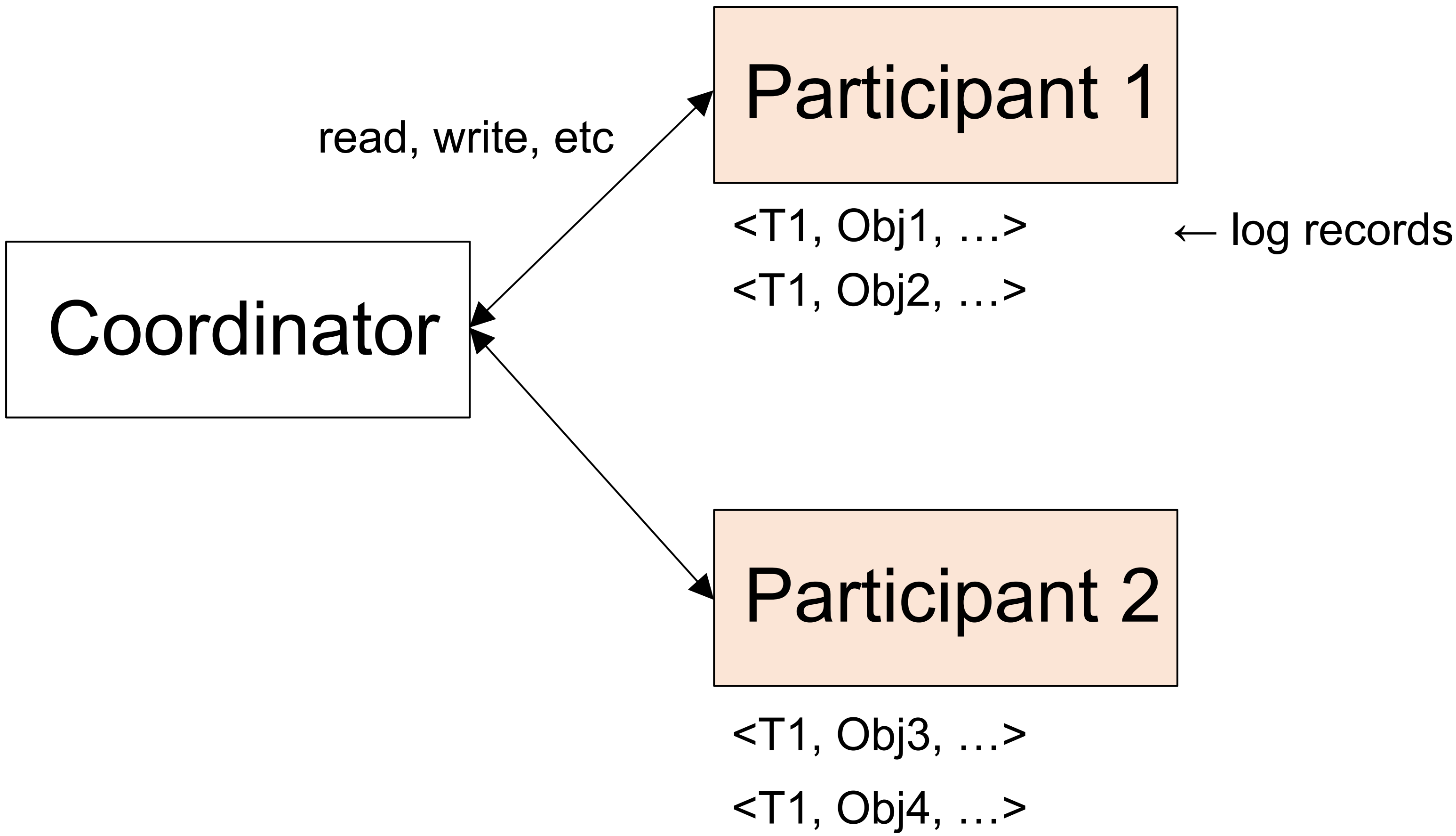
Under strict 2PL, run 2PC before unlocking the write locks

2PC + Logging

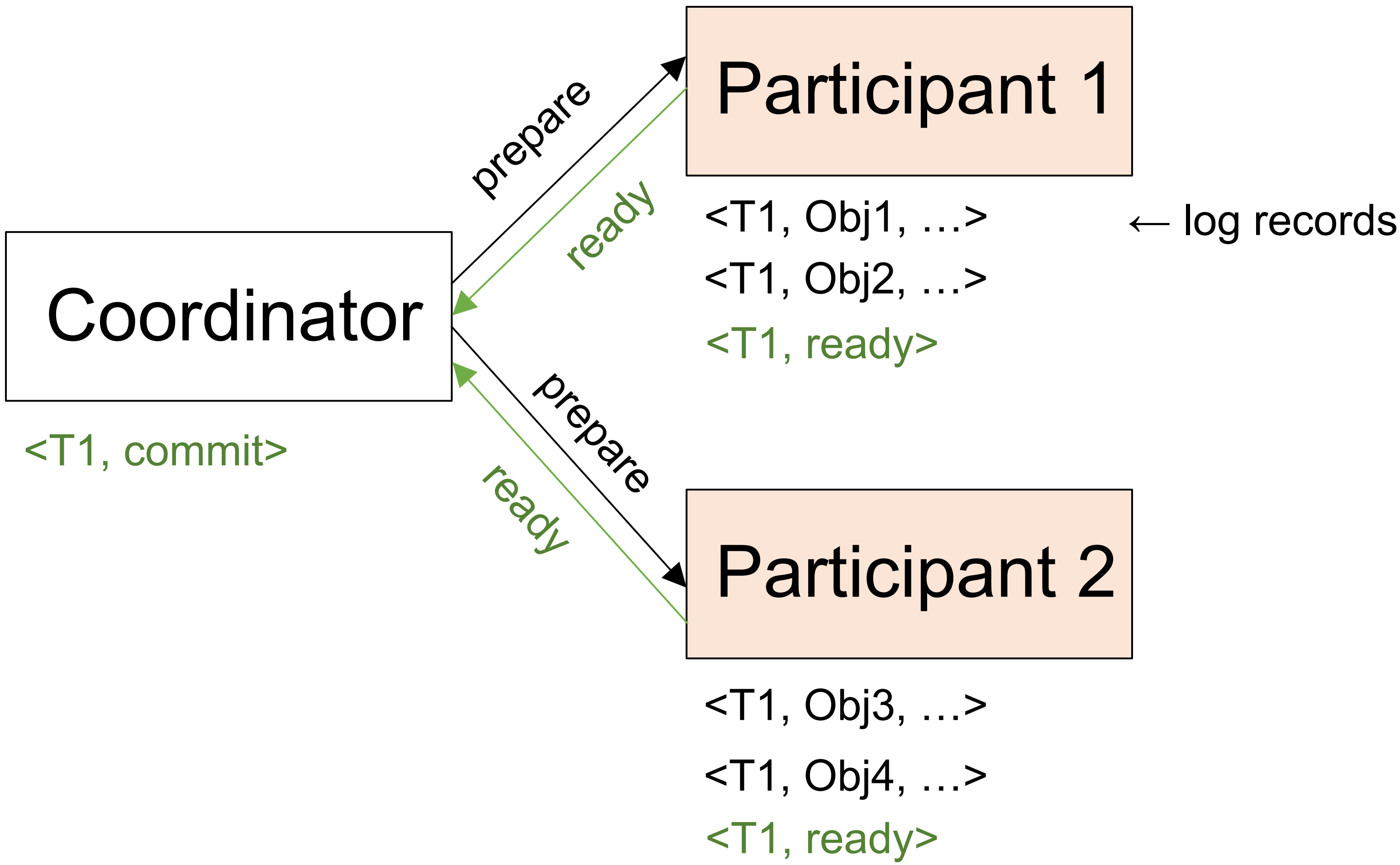
Log records must be flushed to disk on each participant before it replies to *prepare*

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

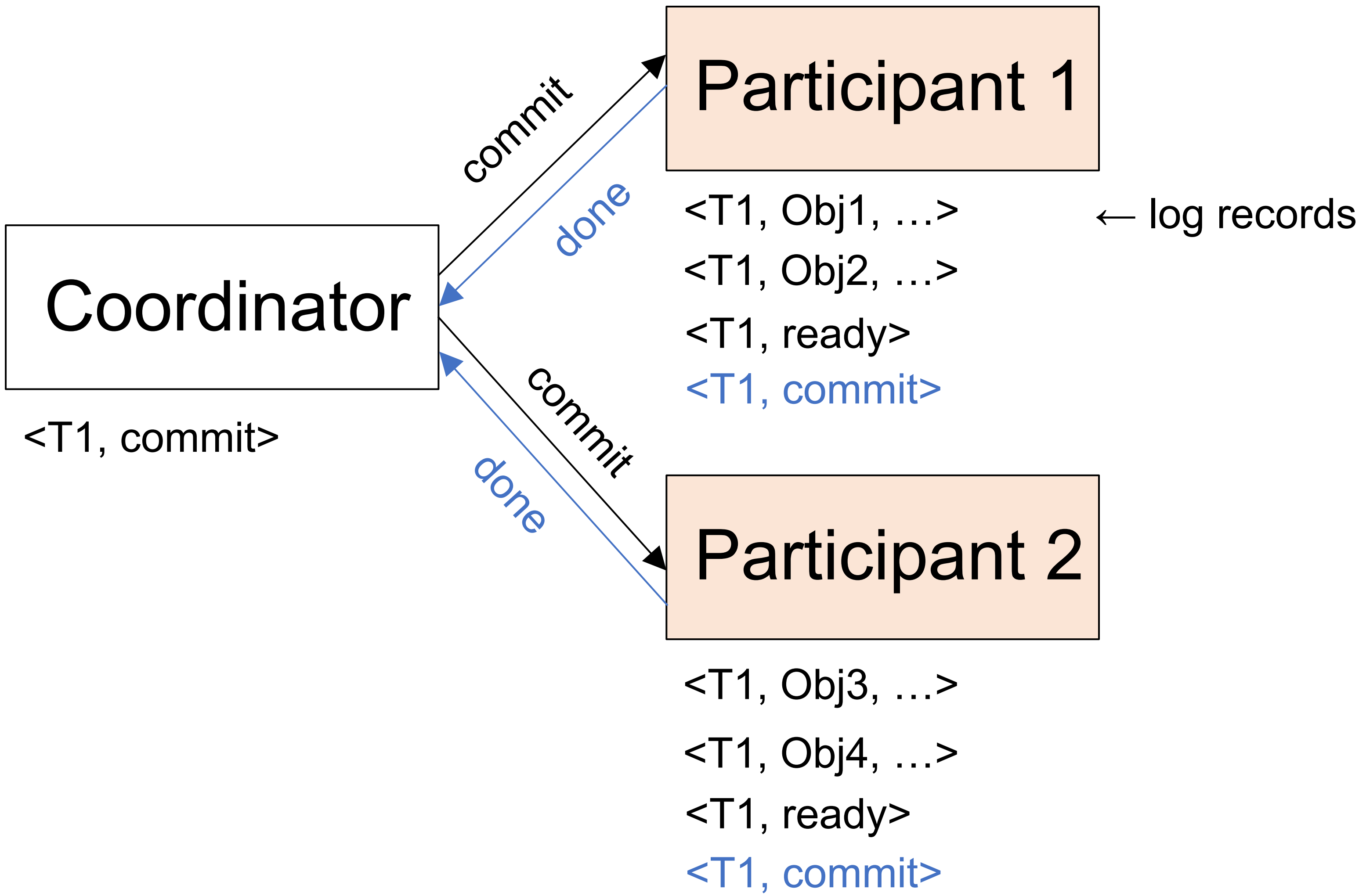
2PC + Logging Example



2PC + Logging Example



2PC + Logging Example



Optimizations Galore

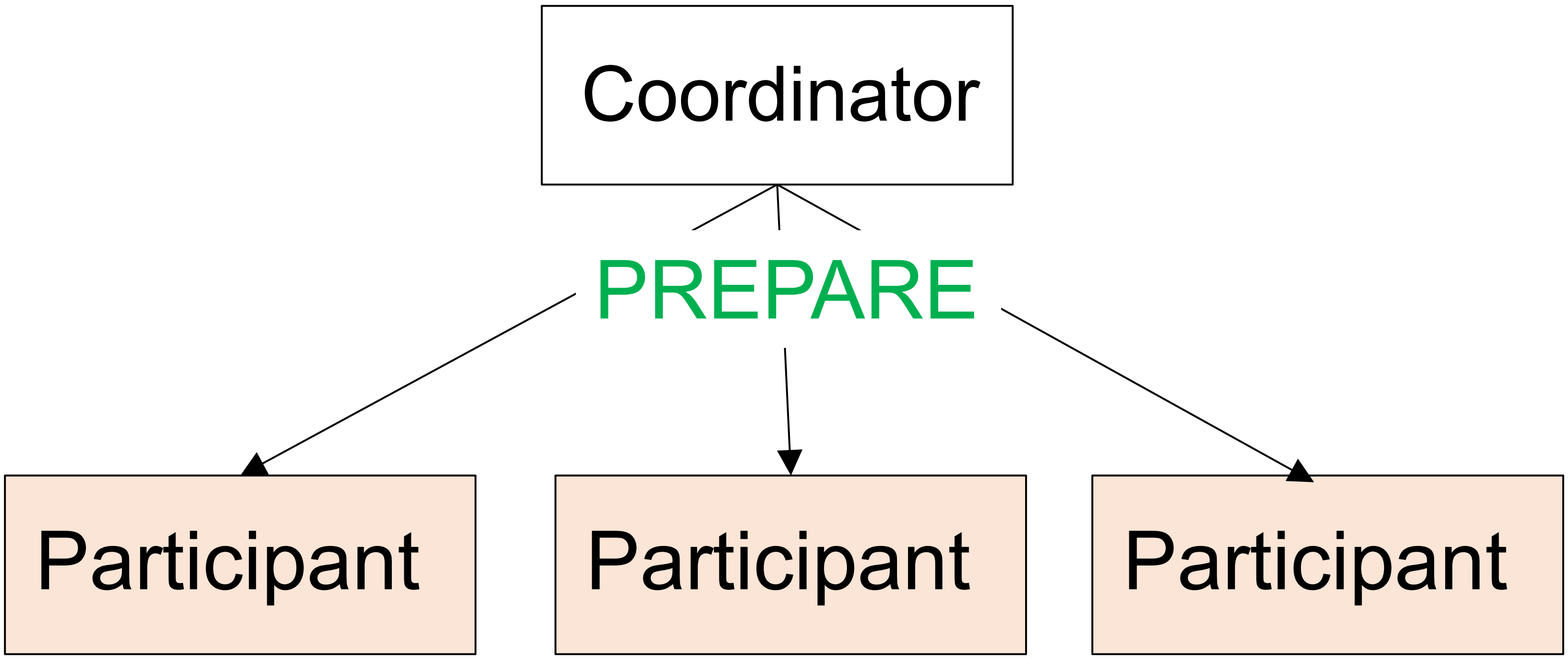
Participants can send *prepared* messages to each other:

- Can commit without the client
- Requires $O(P^2)$ messages

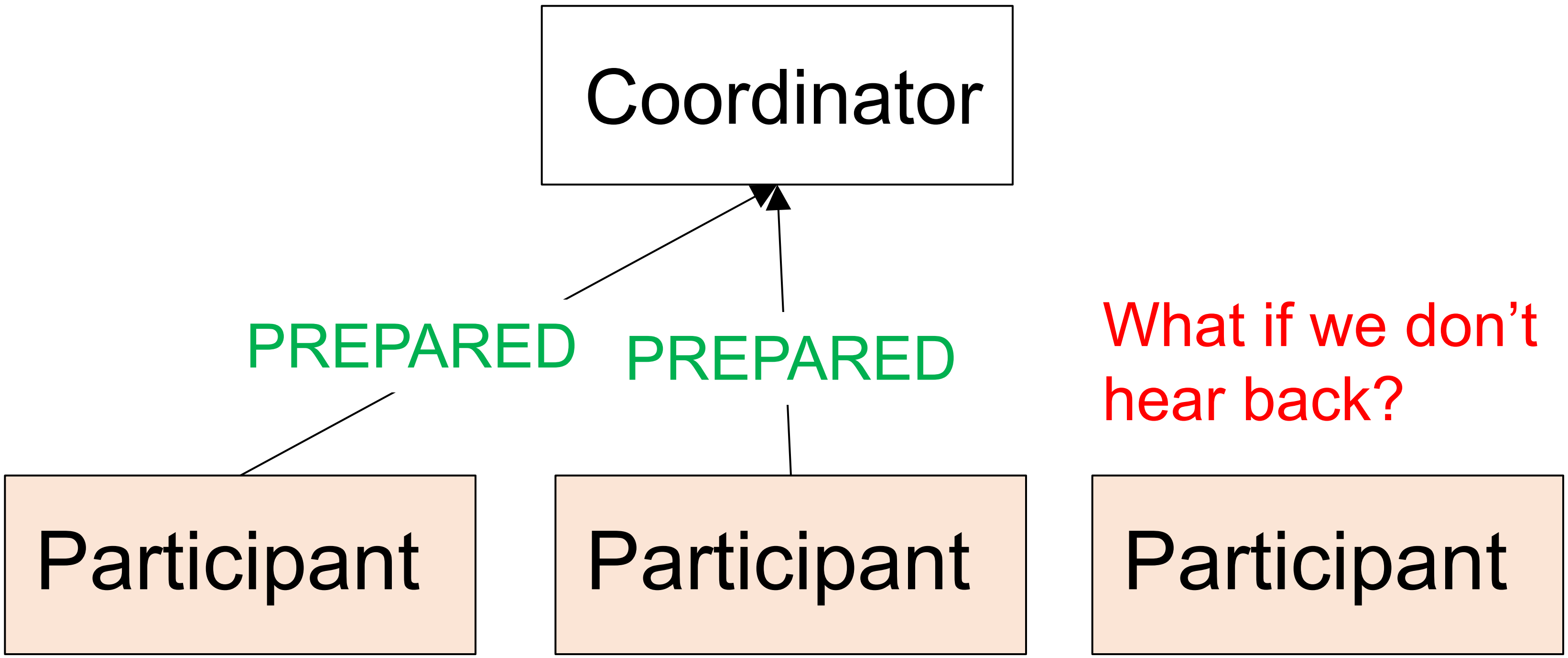
Piggyback transaction’s last command on *prepare* message

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

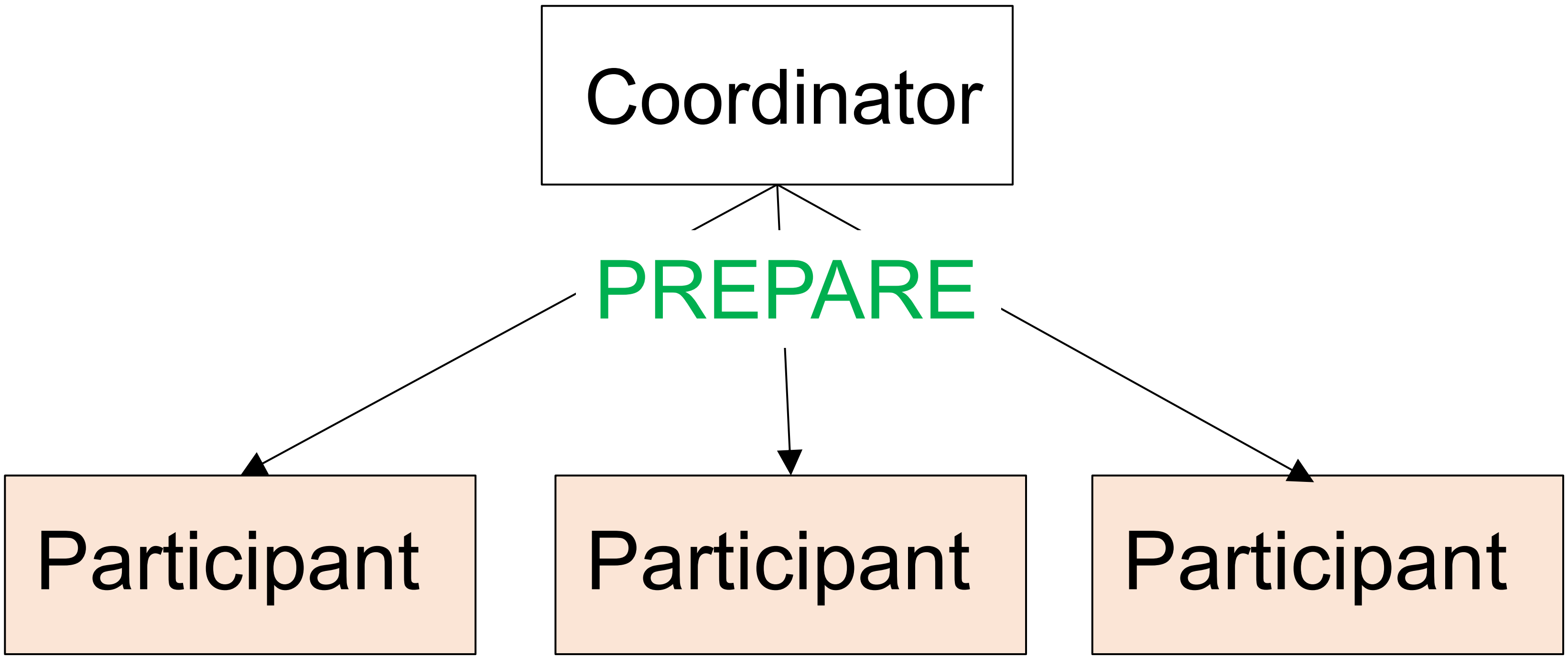
Coordinator can still decide to *abort*

» Coordinator makes the final call!

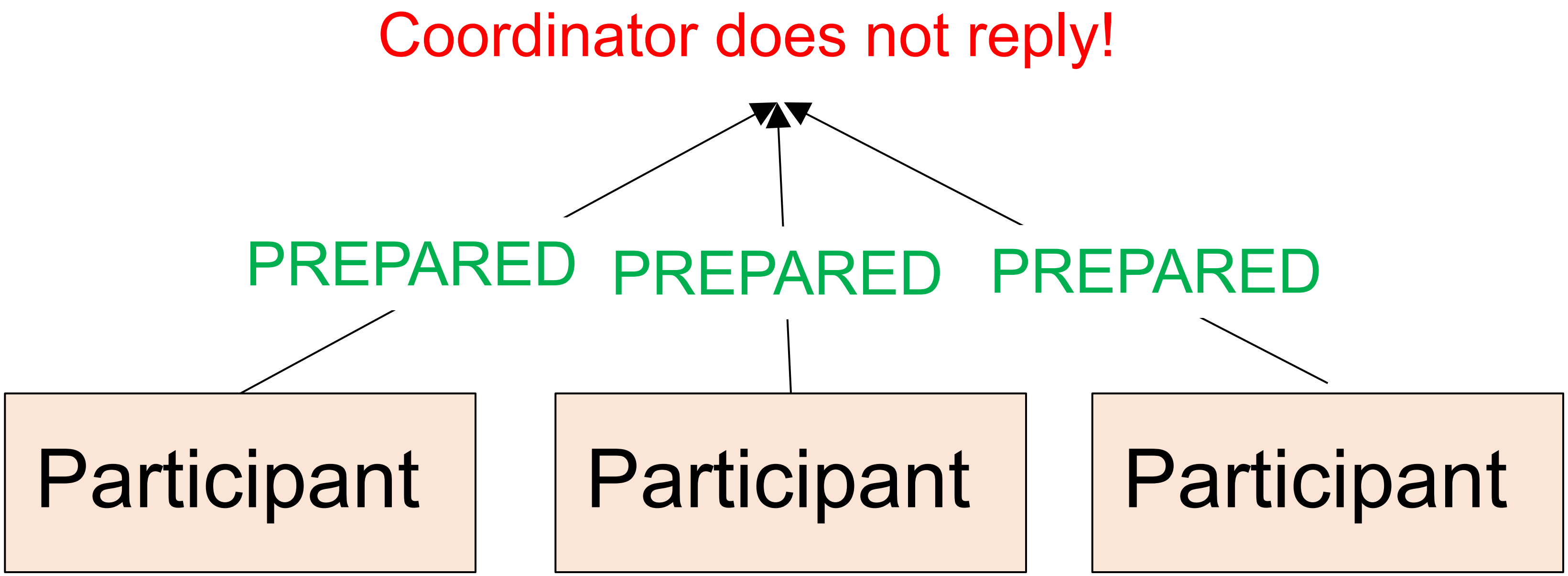
Participant comes back online?

» Will receive the *abort* message

What Could Go Wrong?



What Could Go Wrong?



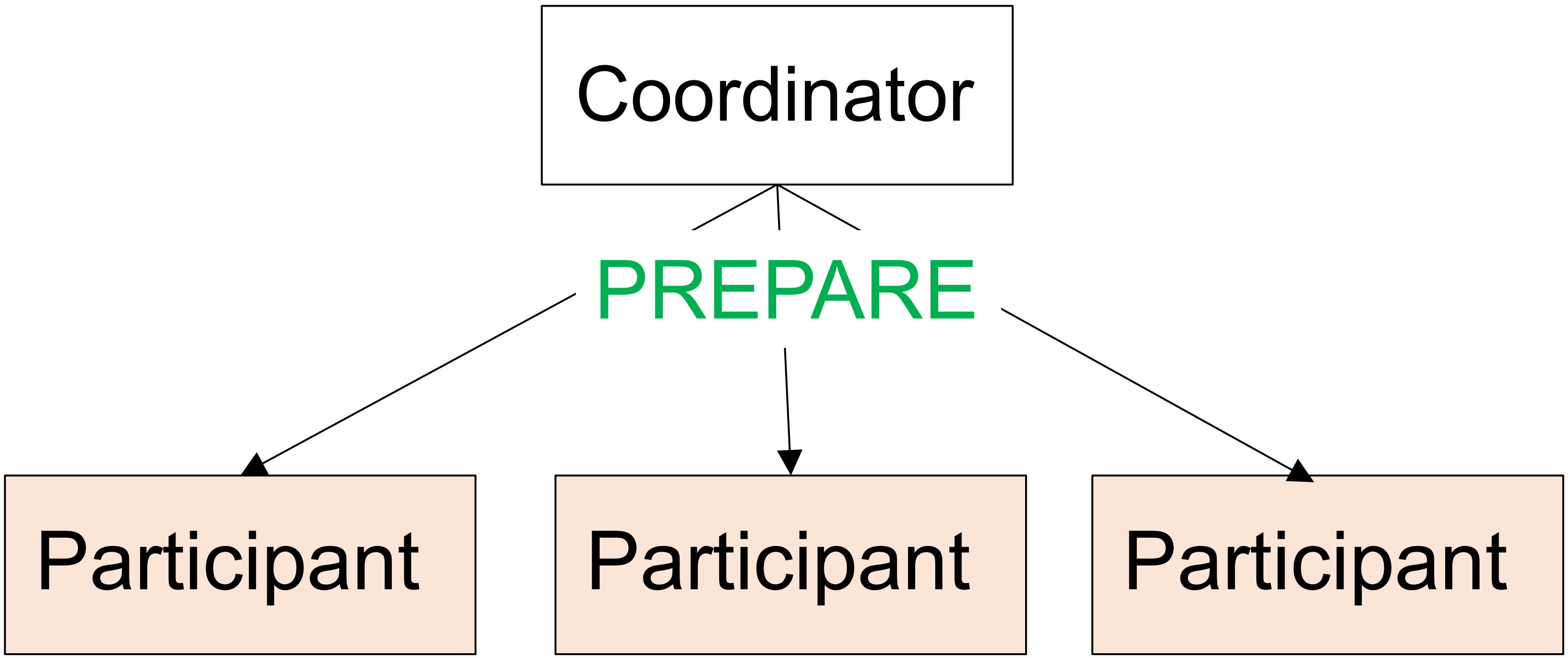
Case 2: Coordinator Unavailable

Participants cannot make progress

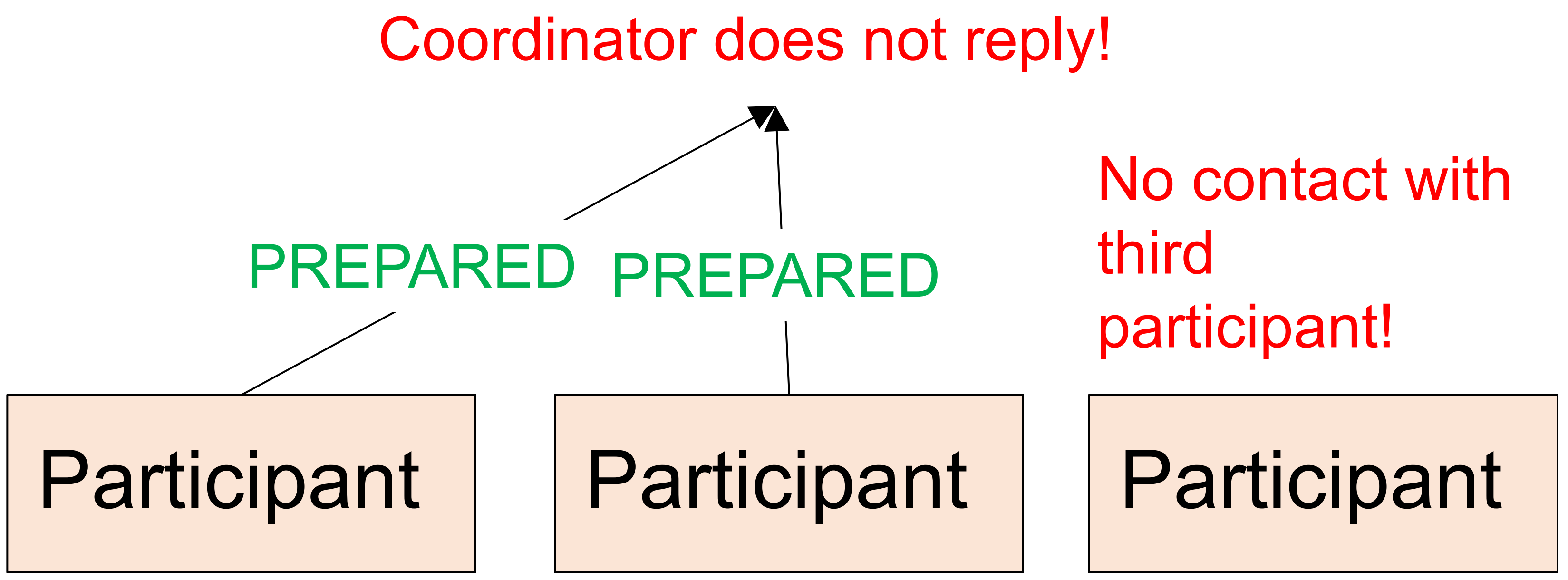
But: can agree to elect a *new* coordinator, never listen to the old one (using consensus)

» 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

AC & 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.

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.

Eric Brewer

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

Let's Talk About Coordination

If we're "AP", then we don't have to talk even when we can!

If we're "CP", then we have to talk all the time

How fast can we send messages?

Let’s Talk About Coordination

If we’re “AP”, then we don’t have to talk even when we can!

If we’re “CP”, then we have to talk all the time

How fast can we send messages?

» Planet Earth: 144ms RTT

- (77ms if we drill through center of earth)

» Einstein!

Multi-Datacenter Transactions

Message delays often much worse than speed of light (due to routing)

44ms apart? maximum 22 conflicting transactions per second

- » Of course, no conflicts, no problem!
- » Can scale out

Pain point for many systems

Do We *Have* to Coordinate?

Is it possible achieve some forms of "correctness" without coordination?

Do We *Have* to Coordinate?

Example: no user in DB has address=NULL

» If no replica assigns address=NULL on their own, then NULL will never appear in the DB!

Whole topic of research!

» Key finding: most applications have a few points where they need coordination, but many operations do not

So Why Bother with Serializability?

For arbitrary integrity constraints, non-serializable execution can break constraints

Serializability: just look at reads, writes

To get “coordination-free execution”:

» Must look at application semantics

» Can be hard to get right!

» Strategy: start coordinated, then relax

Punchlines:

Serializability has a provable cost to latency, availability, scalability (if there are conflicts)

We can avoid this penalty if we are willing to look at our application *and* our application does not require coordination

- » Major topic of ongoing research

Outline

Replication strategies

Partitioning strategies

AC & 2PC

CAP

Avoiding coordination

Parallel query execution

Avoiding Coordination

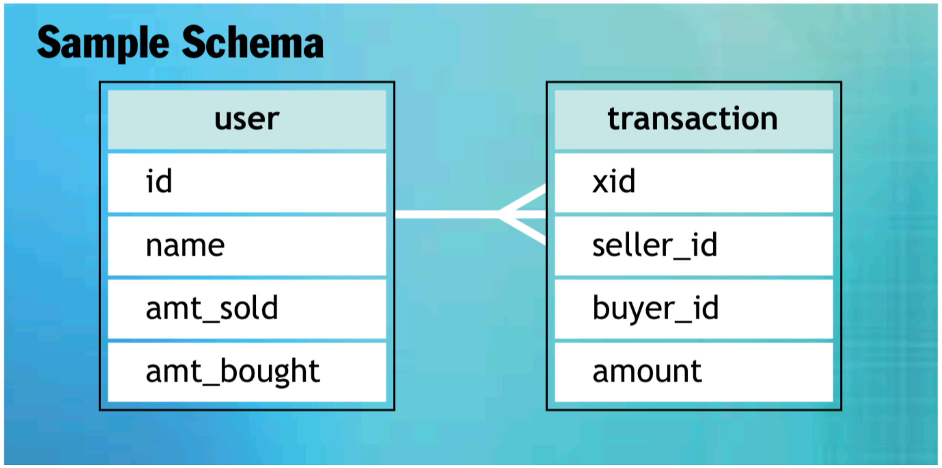
Several key techniques; e.g. BASE ideas

» Partition data so that most transactions are local to one partition

» Tolerate out-of-date data (eventual consistency):

- Caches
- Weaker isolation levels
- Helpful ideas: idempotence, commutativity

Example from BASE Paper

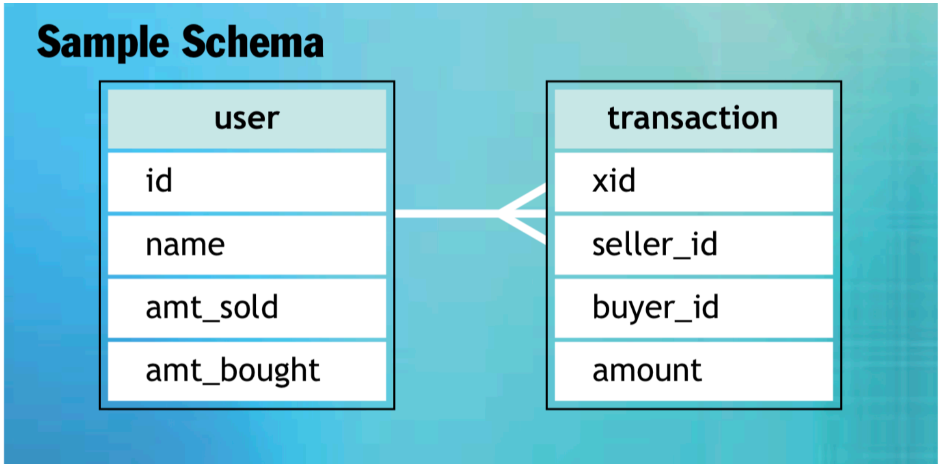


Constraint: each user's amt_sold and amt_bought is sum of their transactions

ACID Approach: to add a transaction, use 2PC to update transactions table + records for buyer, seller

One BASE approach: to add a transaction, write to transactions table + a persistent queue of updates to be applied later

Example from BASE Paper



Constraint: each user's amt_sold and amt_bought is sum of their transactions

ACID Approach: to add a transaction, use 2PC to update transactions table + records for buyer, seller

Another BASE approach: write new transactions to the transactions table and use a periodic batch job to fill in the users table

Helpful Ideas

When we delay applying updates to an item, must ensure we only apply each update once

- Issue if we crash while applying!
- **Idempotent operations:** same result if you apply them twice

When different nodes want to update multiple items, want result independent of msg order

- **Commutative operations:** $A \circledast B = B \circledast A$

Example Weak Consistency Model: Causal Consistency

Very informally: transactions see **causally ordered** operations in their causal order

- Causal order of ops: $O_1 \prec O_2$ if done in that order by one transaction, or if write-read dependency across two transactions

Causal Consistency Example

Shared Object:
group chat log for
{Matei, Alice, Bob}

Matei's Replica

Matei: pizza tonight?
Bob: sorry, studying :(
Alice: sure!

Alice's Replica

Matei: pizza tonight?
Alice: sure!
Bob: sorry, studying :(

Bob's Replica

Matei: pizza tonight?
Bob: sorry, studying :(
Alice: sure!

BASE Applications

What example apps (operations, constraints) are suitable for BASE?

What example apps are unsuitable for BASE?

Outline

Replication strategies

Partitioning strategies

AC & 2PC

CAP

Avoiding coordination

Parallel query execution

Why Parallel Execution?

So far, distribution has been a chore, but there is 1 big potential benefit: **performance**!

Read-only workloads (analytics) don't require much coordination, so great to parallelize

Challenges with Parallelism

Algorithms: how can we divide a particular computation into pieces (efficiently)?

» Must track both CPU & communication costs

Imbalance: parallelizing doesn't help if 1 node is assigned 90% of the work

Failures and stragglers: crashed or slow nodes can make things break

Whole course on this: CS 149

Amdahl's Law

If p is the fraction of the program that can be made parallel, running time with N nodes is

$$T(n) = 1 - p + p/N$$

Result: max possible speedup is $1 / (1 - p)$

Example: 80% parallelizable ⇒ 5x speedup

Example System Designs

Traditional "massively parallel" DBMS

» Tables partitioned evenly across nodes

» Each physical operator also partitioned

» Pipelining across these operators

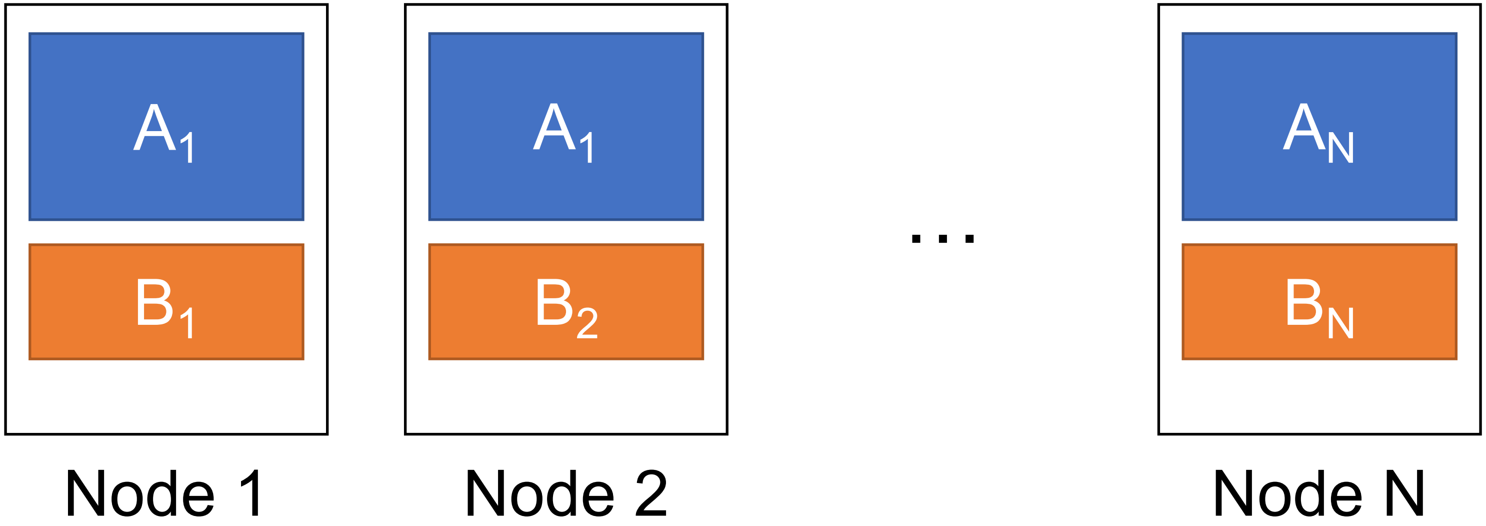
MapReduce

» Focus on unreliable, commodity nodes

» Divide work into idempotent *tasks*, and use dynamic algorithms for load balancing, fault recovery and straggler recovery

Example: Distributed Joins

Say we want to compute $A \bowtie B$, where A and B are both partitioned across N nodes:



Example: Distributed Joins

Say we want to compute A ⋈ B, where A and B are both partitioned across N nodes

Algorithm 1: shuffle hash join

» Each node hashes records of A, B to N partitions by key, sends partition i to node I

» Each node then joins the records it received

Communication cost: (N-1)/N (|A| + |B|)

Example: Distributed Joins

Say we want to compute A ⋈ B, where A and B are both partitioned across N nodes

Algorithm 2: broadcast join on B

- » Each node broadcasts its partition of B to all other nodes
- » Each node then joins B against its A partition

Communication cost: (N-1) |B|

Takeaway

Broadcast join is much faster if $|B| \ll |A|$

How to decide when to do which?

Takeaway

Broadcast join is much faster if $|B| \ll |A|$

How to decide when to do which?

» Data statistics! (especially tricky if B derived)

Which algorithm is more resistant to load imbalance from data skew?

Takeaway

Broadcast join is much faster if $|B| \ll |A|$

How to decide when to do which?

» Data statistics! (especially tricky if B derived)

Which algorithm is more resistant to load imbalance from data skew?

» Broadcast: hash partitions may be uneven!

What if A, B were already hash-partitioned?

Planning Parallel Queries

Similar to optimization for 1 machine, but most optimizers also track data partitioning

» Many physical operators, such as shuffle join, naturally produce a partitioned dataset

» Some tables already partitioned or replicated

Example: Spark and Spark SQL know when an intermediate result is hash partitioned

» And APIs let users set partitioning mode

Handling Imbalance

Choose algorithms, hardware, etc that is unlikely to cause load imbalance

OR

Load balance dynamically at runtime

» Most common: “over-partitioning” (have #tasks ≫ #nodes and assign as they finish)

» Could also try to split a running task

Handling Faults & Stragglers

If uncommon, just ignore / call the operator / restart query

Problem: probability of something bad grows fast with number of nodes

» E.g. if one node has 0.1% probability of straggling, then with 1000 nodes,

$$P(\text{none straggles}) = (1 - 0.001)^{1000} \approx 0.37$$

Fault Recovery Mechanisms

Simple recovery: if a node fails, redo its work since start of query (or since a checkpoint)

» Used in massively parallel DBMSes, HPC

Analysis: suppose failure rate is f failures / sec / node; then a job that runs for $T \cdot N$ seconds on N nodes and checkpoints every C sec has

$$\begin{aligned} E(\text{runtime}) &= (T/C)\ E(\text{time to run 1 checkpoint}) \\ &= (T/C)\ (C \cdot (1 - f^N)^C + c_{\text{checkpoint}}) \end{aligned}$$

Grows fast with N, even if we vary C!

Fault Recovery Mechanisms

Parallel recovery: over-partition tasks; when a node fails, redistribute its tasks to the others

» Used in MapReduce, Spark, etc

Analysis: suppose failure rate is f failures / sec / node; then a job that runs for T·N sec on N nodes with task of size ≪ 1/f has

E(runtime) = T / (1-f)

This doesn’t grow with N!

Summary

Parallel execution can use many techniques we saw before, but must consider 3 issues:

» **Communication cost:** often $\gg$ compute (remember our lecture on storage)

» **Load balance:** need to minimize the time when last op finishes, not sum of task times

» **Fault recovery** if at large enough scale