

Atomic Commitment



COMS4113

COMS4113: Fundamentals of Distributed Systems

Instructor: Hubertus Franke

Columbia University

<https://frankeh.github.io/cu-ds1-2026-fall>

Atomic Commitment Protocols

Two-phase commit, and why one failure can stop everything

- We return to the distributed setting: how **distributed transactions** are actually implemented.
- Why do we distribute at all, and what are the mechanisms?
 - **Scalability** -> sharding: split the data across servers.
 - **Fault tolerance** -> replication: keep copies on several servers.
- Both mechanisms raise semantic problems that need protocols:
 - **Two-phase commit** - an *atomic commitment* protocol. It is what sharding needs, and it is today's subject.
 - **Paxos** - a *consensus* protocol. It is what replication needs, and it is the next lecture.

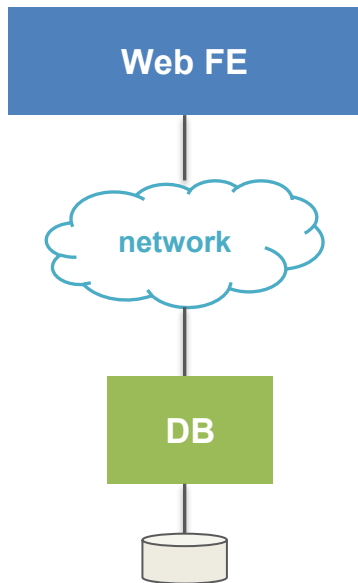
Lecture 6 gave us ACID on one node. The question now is what survives when data spans machines that fail independently.

- A motivating example: a sharded, replicated web service
- **Two-phase commit (2PC)** - the protocol, step by step
- Timeouts, failures, and the **uncertainty period**
- 2PC's limitations: it **blocks**, and it is **expensive**
- 2PC in production, and how systems escape the blocking result
- Next time: **Paxos**, then **Spanner**, which combines both

Motivating Example

From one database to a sharded, replicated service

Example: Web Service Architecture



The starting point

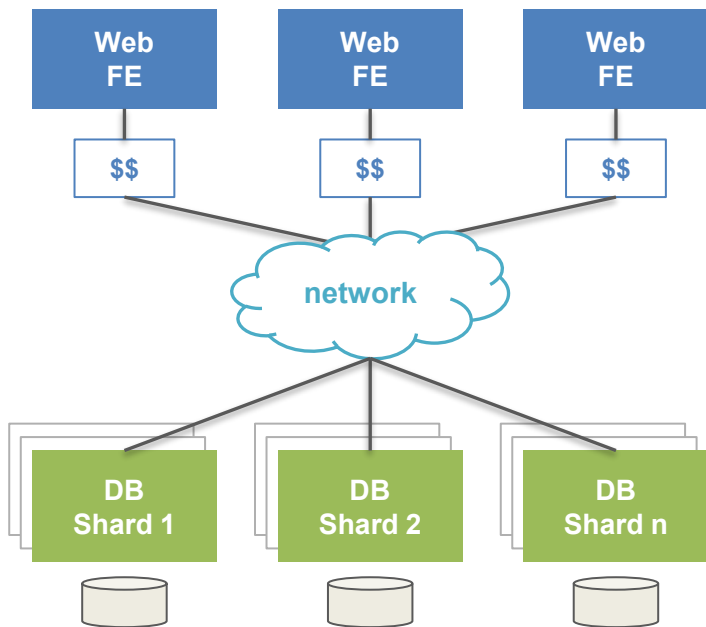
A web front end (**FE**), a database server (**DB**), and a network between them. The FE is **stateless** - all state lives in the DB. The FE runs a banking application: transfers, listings, and so on. The DB gives it ACID transactions, and the FE uses them.

Question

How do we make this

- **scalable?**
- **fault tolerant?**

Scalability: Sharding



Both tiers are sharded

FEs accept requests from browsers and process them concurrently.

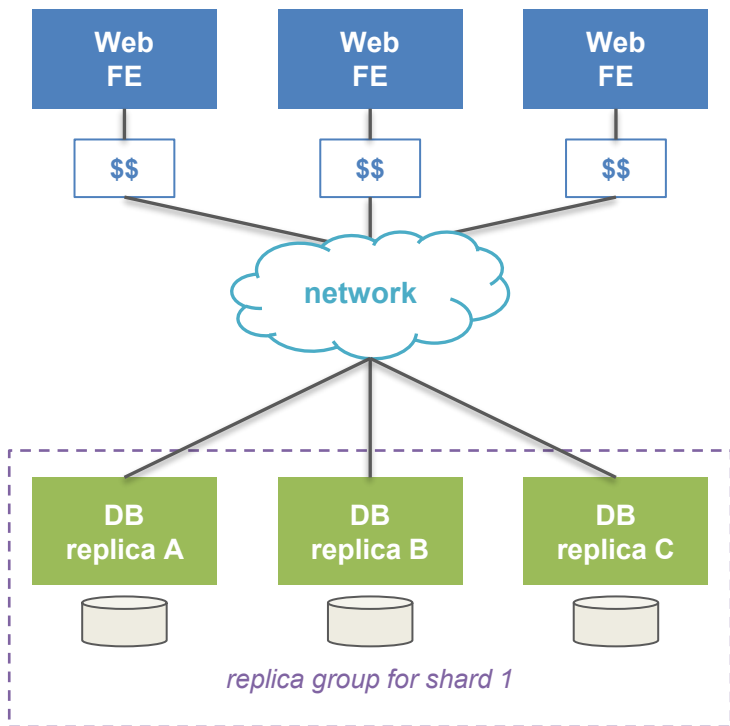
The **DB is sharded** - say, by user ID.

Each DB backend is independently transactional (ACID), so an FE issues transactions against **one or more** DB shards.

The seed of the problem

"One **or more**" is doing a lot of work in that sentence. Hold onto it.

Fault Tolerance: Replication



Stateless is easy; stateful is not

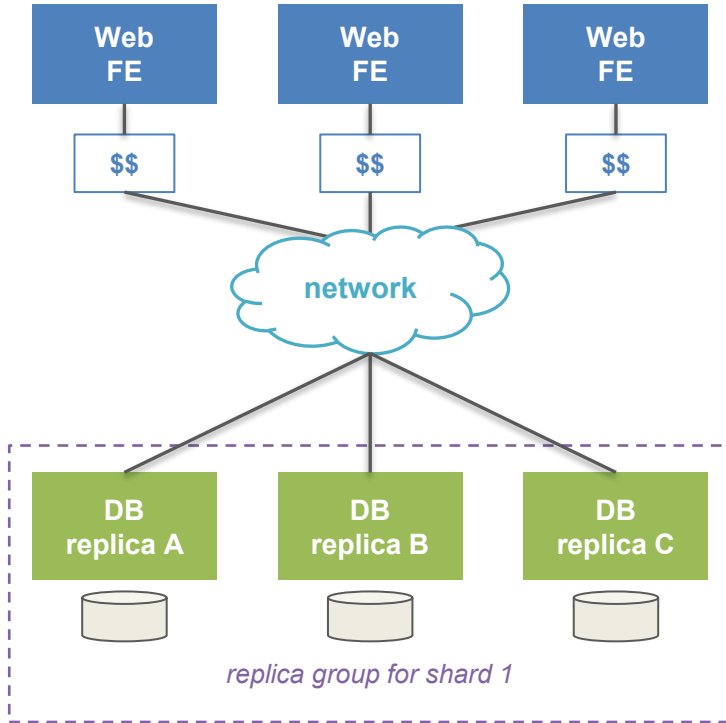
The **FE is stateless**, so sharding it already makes it replicated and fault tolerant - any FE can serve any request.

The **DB is stateful**, so each shard needs **active replication**: a *replica group* whose members cooperate to stay up to date.

Putting it together

The FE sends requests for different shards to different replica groups. Now every tier is both **sharded** and **replicated**.

Challenges



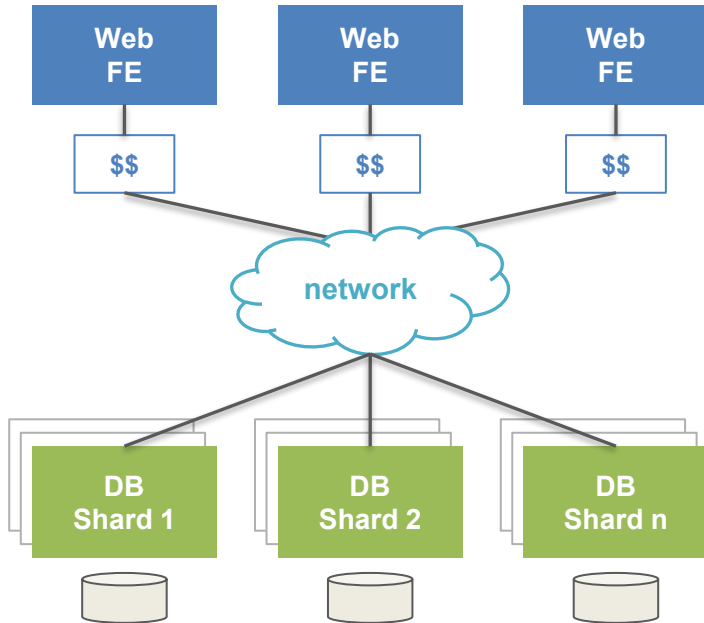
Question

What are the challenges of implementing **ACID** across the whole **sharded and replicated** DB service?

We take them one at a time

First ignore replication and look at sharding. Then ignore sharding and look at replication.

Challenges due to Sharding



Ignore replication for now

Implementing ACID across all DB shard servers:

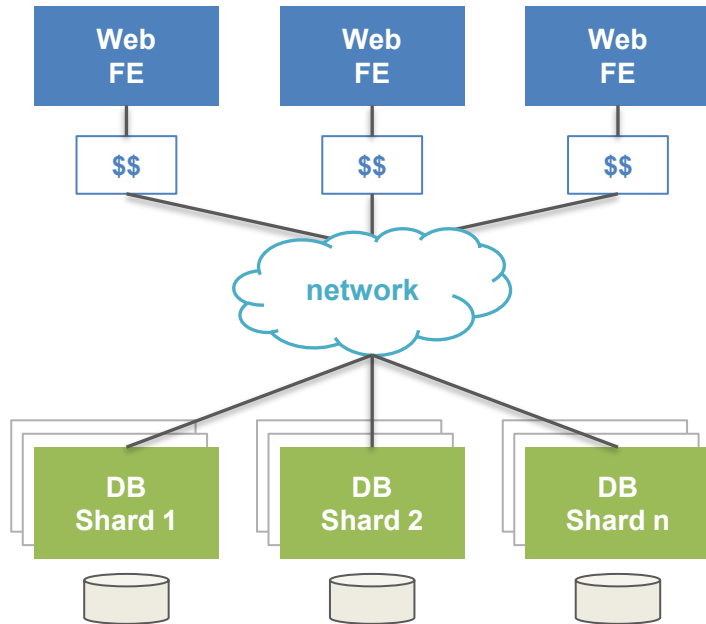
Case 1 - single-shard only

No transaction ever spans shards. **Easy**: the individual shard just runs the transaction, exactly as in Lecture 6.

Case 2 - transactions span shards

The challenge. Shards in a transaction must **agree** on (1) whether to commit, and (2) when to release locks.

Challenges due to Sharding (cont.)



Example

The FE is the banking service with **TRANSFER** and **REPORT_SUM** from Lecture 6.

If the two accounts live on **different shards**, the two operations - deduct from one, add to the other - must happen **both or neither**.

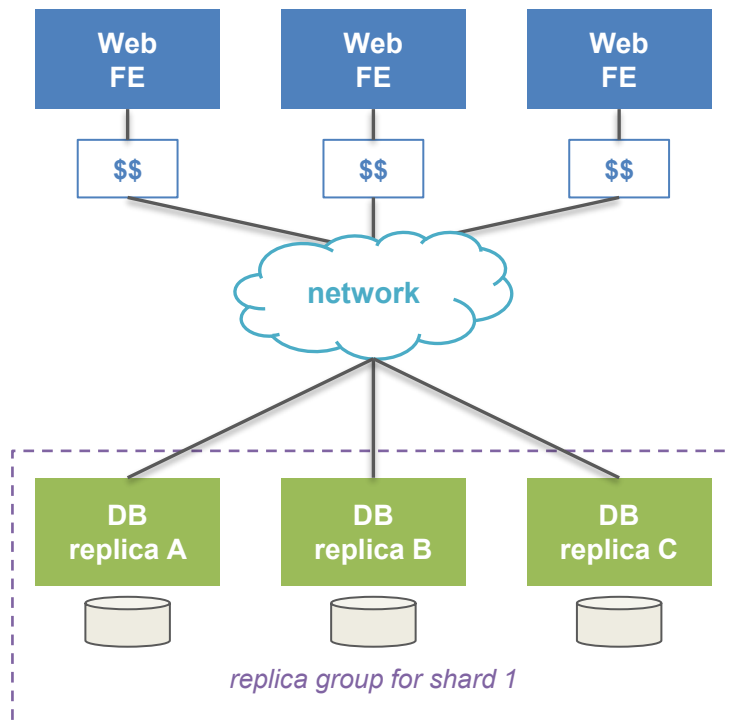
But

The two machines can fail, or unilaterally abort, *INDEPENDENTLY*.

So you need an agreement protocol

Here the right one is an **atomic commitment protocol**. The best known is **two-phase commit**.

Challenges due to Replication



Now ignore sharding

Implementing ACID across all replicas of one shard:

Challenge

All replicas of the shard must execute **all operations in the same order**.

Why that is enough

If the operations are **deterministic**, agreeing on their order makes every replica's copy evolve identically - so they stay **consistent**.

Challenges due to Replication (cont.)

Example

Two transactions, one operation each, on the same cell:

TX1: $x += 1$ TX2: $x \times= 2$

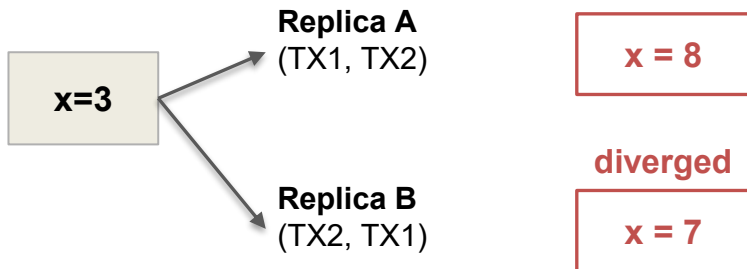
Each replica is internally ACID, so each serializes them: either (TX1, TX2) or (TX2, TX1).

So what problem is this?

Agreeing on the **order** of operations is exactly multiple instances of the **consensus problem**.

Well-known consensus protocol: **Paxos**.

Suppose $x = 3$ initially:



Where each one goes

2PC - atomic commitment, for sharding: **today**.

Paxos - consensus, for replication: **prev lecture**.

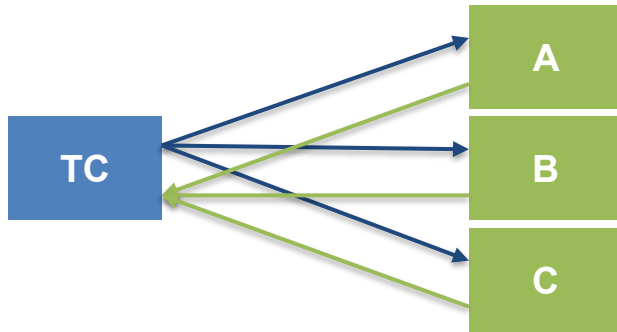
Two-Phase Commit (2PC)

One coordinator, n participants, two rounds

Two-Phase Commit (2PC)

Prepare Phase

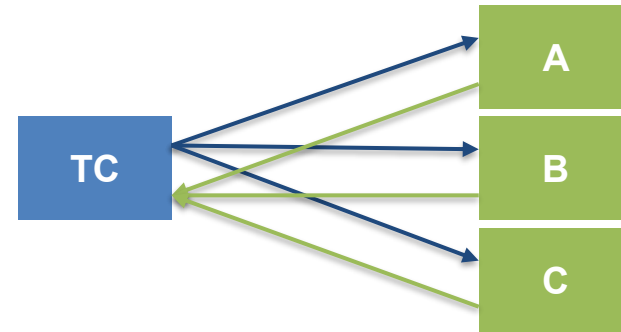
1. PREPARE



2. PREPARE-OK / FAIL

Commit Phase

3. COMMIT / ABORT



4. OK

TC = transaction coordinator. **A, B, C** = participants. One round out, one round back - twice.

Participants are named A/B/C while the protocol is generic; from the next slide on they are the shards S1/S2/S3.

2PC for Distributed Transactions

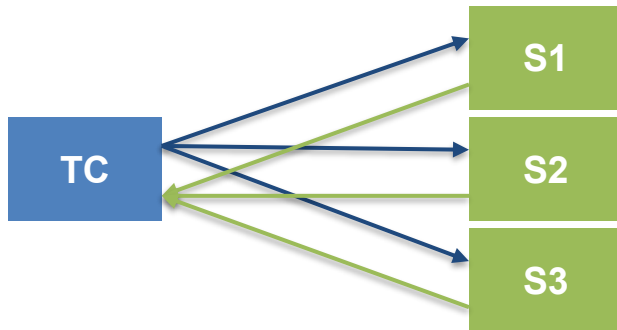
- How 2PC fits together with the **WAL** and **2PL** from Lecture 7. A *rough* sketch of a client library for distributed transactions:
- **begin()** - the client library calls `begin()` on each shard separately. Each returns its own txID: `Tx.S1`, `Tx.S2`, ...
- Operations are routed to the shard that owns the data - `op1` to **S1**, `op2` to **S2**. Each shard takes its **local locks** (2PL) and appends the operation to its **local WAL**.
- **abort()** - the client sends `ABORT` to `S1`, `S2`, ... Each reverts locally using its own WAL, exactly as in Lecture 7.
- **commit()** - this is the hard one, and it is what 2PC is for. Next slide.

Note what is already familiar: the locks and the log are Lecture 7's, unchanged. 2PC only coordinates the decision.

2PC commit() - the Prepare phase

Prepare Phase

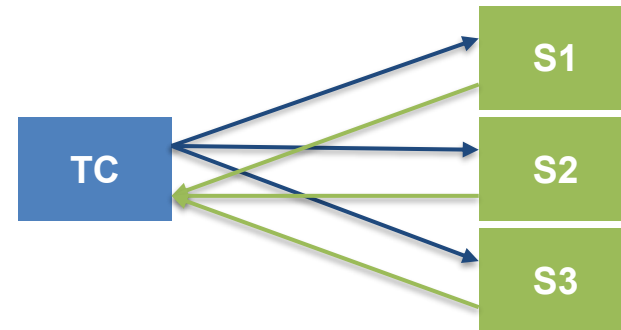
1. PREPARE



2. PREPARE-OK / FAIL

Commit Phase

3. COMMIT / ABORT



4. OK

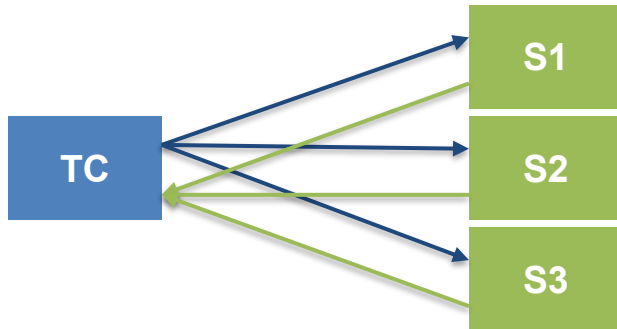
Each participant, on PREPARE

2.1 can I commit Tx.S1? **2.2** write the vote to my WAL **2.3** send **PREPARE-OK / FAIL** **2.4** wait for the TC
Locks for Tx.S1 are held throughout - before, during and after the vote.

2PC commit() - the Commit phase

Prepare Phase

1. PREPARE



2. PREPARE-OK / FAIL

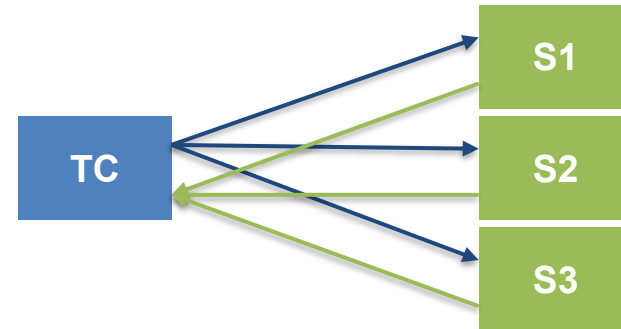
The TC, once votes are in

3.1 all OK -> **COMMIT**, else **ABORT**

3.2 write it to WAL - **the commit point**; 3.3 send it out

Commit Phase

3. COMMIT / ABORT



4. OK

Each participant, on the outcome

4.1 write it to WAL; 4.2 revert if ABORT

4.3 **release all locks**; 4.4 send OK to the TC

The whole protocol on one page

TC		Participant S_i	Locks held?
op	-	2PL locks taken, op in WAL	yes
1. PREPARE	sends PREPARE	-	yes
2. vote	-	writes vote to WAL , sends it	yes
		<i>uncertainty period begins if OK</i>	
3. decide	writes outcome to WAL (flush if COMMIT)	waits	yes
4. apply	retries until all ack	writes outcome, applies or reverts, releases locks	no

Read the last column

The locks are held from the first operation until the outcome arrives - across **both** round trips. That is the cost, and on slide 28 it becomes the problem.

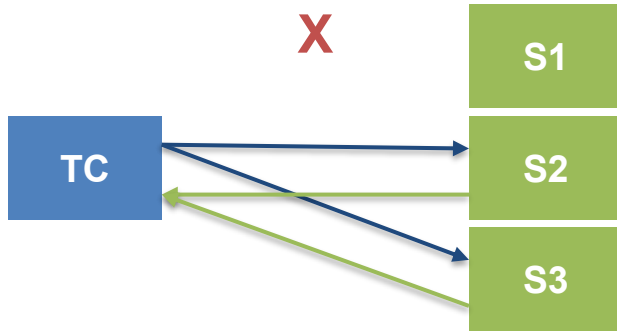
Timeouts and Failures

What can each side safely do when a message never arrives?

Timeout: awaiting PREPARE

Prepare Phase

1. PREPARE



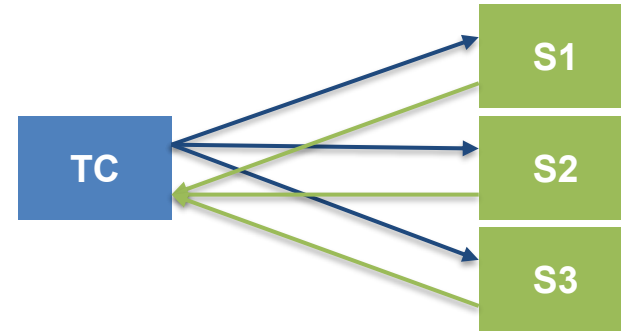
2. PREPARE-OK / FAIL

Situation

S1 times out awaiting **PREPARE** from the TC.

Commit Phase

3. COMMIT / ABORT



4. OK

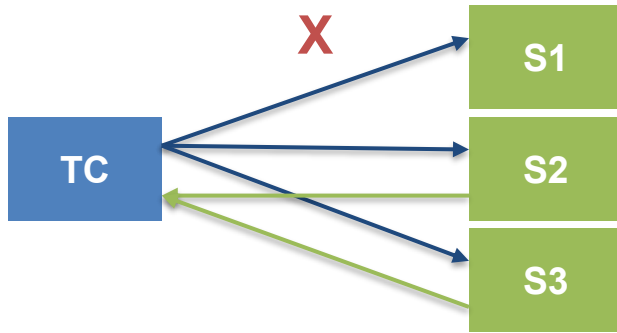
Action

Safe for S1 to **unilaterally abort()**.
S1 has not voted, so no outcome can depend on it.

Timeout: the TC awaits a vote

Prepare Phase

1. PREPARE



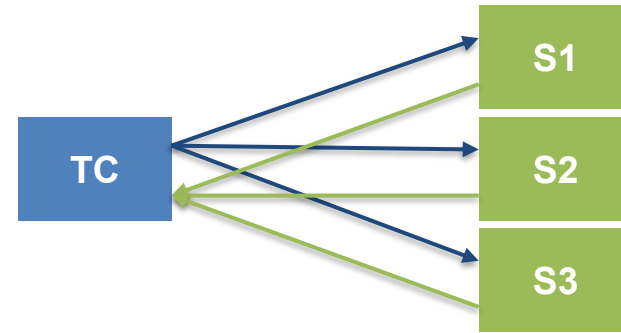
2. PREPARE-OK / FAIL

Situation

The **TC** times out waiting for S1's vote.

Commit Phase

3. COMMIT / ABORT



4. OK

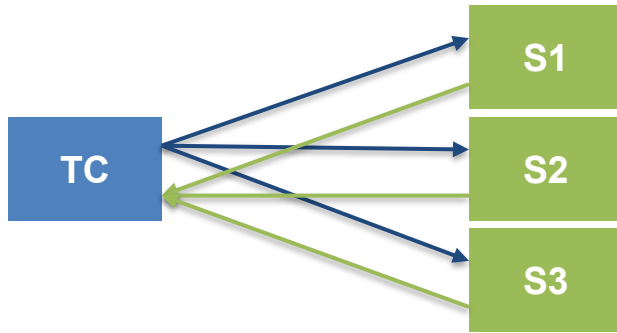
Action

Safe for the TC to **abort()** the transaction.
COMMIT needs unanimity; silence counts as a FAIL.

Timeout: awaiting the outcome

Prepare Phase

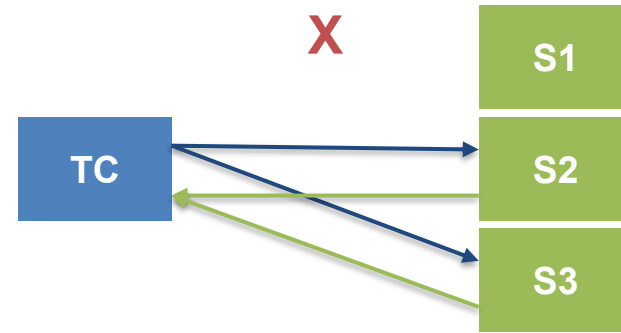
1. PREPARE



2. PREPARE-OK / FAIL

Commit Phase

3. COMMIT / ABORT



4. OK

S1 times out awaiting the outcome. Is it safe for S1 to unilaterally commit or abort?

Case 1 - S1 voted PREPARE-FAIL

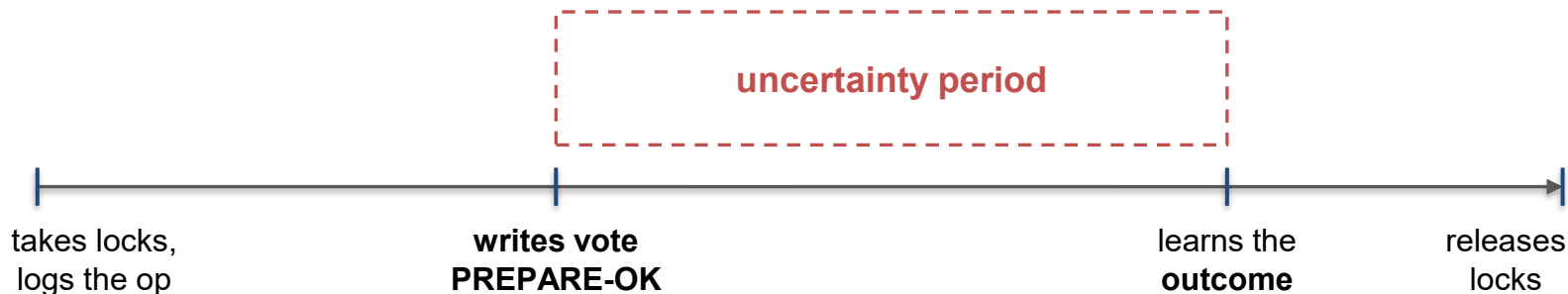
Safe to **unilaterally abort()**; its FAIL rules COMMIT out.

Case 2 - S1 voted PREPARE-OK

Cannot decide - it must run a **termination protocol**.

The uncertainty period, precisely

S1 has given up its veto and does not yet know the outcome



Why the window cannot be closed

Before voting, S1 held a **veto**, so ABORT was always safe.

After voting **OK** that veto is spent: the outcome depends on votes S1 cannot see. Committing might contradict a FAIL elsewhere; aborting might contradict a COMMIT already sent to the client.

[Skeen-1983] formalises this window and proves that no protocol of this shape can avoid it.

Termination Protocol

Option 1 - wait for the TC to come back

Correct, and always safe. But it can take arbitrarily long, and **Si holds its locks the whole time**.

Option 2 - ask the other participants

If **any** of them already learned the outcome, they can all terminate accordingly. This often works, and is why real implementations do it.

But if none of them knows either

If the TC died or was partitioned *right before* it sent the outcome, no participant knows it - and they are all **BLOCKED** until the TC returns.

And what is stuck is not just this transaction: it is every transaction wanting those locks, on shards that never failed.

- The same analysis applies to **crashes**, not just timeouts - a recovering participant reads its own WAL and asks the same question: *what did I already promise?*
- **Not in the uncertainty period** - it either finds no vote (so it may safely abort) or finds the outcome already logged (so it does that). It can decide **on its own**.
- **In the uncertainty period** - it voted OK but never logged an outcome. It cannot decide alone, and must run the **termination protocol** - which, as before, may not terminate if the TC is gone.
- So crash recovery does not rescue 2PC. The uncertainty period is a property of **the protocol**, not of any particular failure.

This is why the WAL writes in steps 2.2, 3.2 and 4.1 are ordered the way they are: the log is what a recovering node interrogates.

2PC Limitations

Two problems: it blocks, and it costs three round trips

2PC is Blocking

The bound

A participant can block **indefinitely** in its uncertainty period, until a TC or network failure is resolved.

It gets worse if the TC is also a participant

Then a **single-site failure** can block 2PC indefinitely.

And it blocks while holding locks

Each shard holds its locks throughout - so transactions that never touch the failed shard also stall.

Therefore

2PC is a **blocking protocol**, and *cannot be used as a basis for fault tolerance*.

2PC is Expensive

Three message latencies, in series

1. PREPARE

TC -> all participants

2. vote

all participants -> TC

3. outcome

TC -> all participants

Step 4's OK is off the critical path - the client already has its answer.

Message complexity

For n participants and one TC, the common case costs **$3n$ messages**.

With geo-distributed shards, each of those three latencies is a **WAN** round trip.

What about 3PC?

Adding a phase - or other optimizations - **cannot fix 2PC's blocking and performance problems while keeping its semantics**. 3PC buys non-blocking only by assuming *bounded* message delay and response time; under real asynchrony it loses atomicity instead.

2PC in production, 2026

2PC ships in the database you are most likely to use - and its manual reads like this lecture.

PostgreSQL's prepare record is this protocol

PREPARE TRANSACTION "prepares the current transaction for two-phase commit... its state is fully stored on disk". That is step 2.2, in a shipping product.

The docs state our lock problem as an operational warning

"the transaction **continues to hold whatever locks it held**", and leaving one prepared "will interfere with... VACUUM... and in extreme cases could cause the database to **shut down**".

So it ships disabled

max_prepared_transactions defaults to **0**. The docs are blunt: "unless you're writing a transaction manager, you probably shouldn't be using PREPARE TRANSACTION".

Escaping the blocking result

Read slide 28's verdict narrowly: **2PC alone** cannot be a basis for fault tolerance. Look at *why*.

The single point of failure

The protocol blocks because there is **one TC**, and it can be gone. Nothing else in 2PC is at fault.

So make the coordinator fault tolerant

Replicate the TC's decision with a **consensus** protocol - the very thing the replication branch needed. "The TC is gone" stops being reachable.

This is a named result - and it is what Spanner does

Paxos Commit [Gray-Lamport-2006] stores that decision with Paxos, so a TC failure blocks no one. **Spanner** runs 2PC across shards, each shard a **Paxos group** - both protocols composed. That is Lecture 9.

Cutting the latency

An example: how one production system cuts a round trip.

Slide 29's *three* message latencies are not a law of nature - the **blocking** bound is hard, but the **latency** is negotiable.

The example: CockroachDB's Parallel Commits

An "optimized atomic commit protocol that cuts the commit latency of a transaction in half, from **two rounds of consensus down to one**".

How: don't wait to find out you succeeded

At commit time the coordinator writes a **STAGING** record listing every write still **in flight**. STAGING *plus* all those writes at consensus = **implicitly committed**.

The honest caveat

Intent resolution still takes two rounds of consensus, so **highly contended** workloads benefit less. And recovery is only needed if the coordinator dies mid-commit.

Fault tolerance through replication

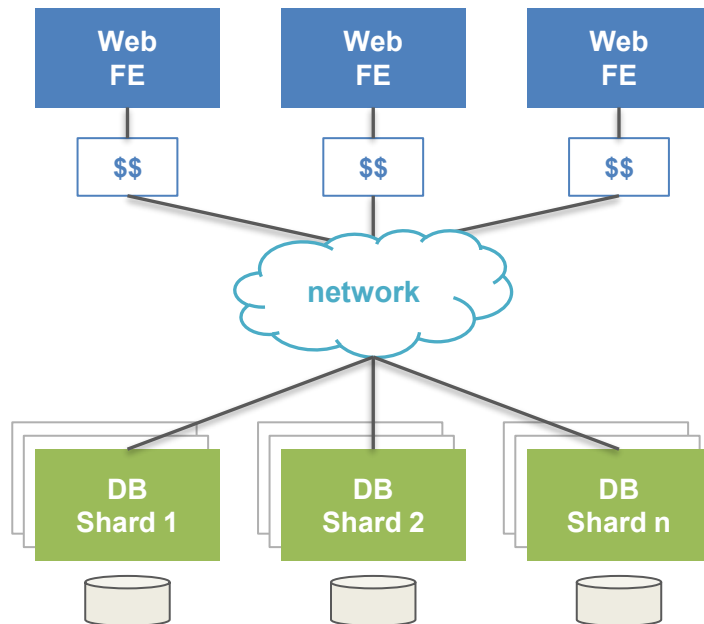
The other half of the motivating example, and the branch we deferred on slide 13.

Cast as consensus, not atomic commitment

Unlike sharded DBs, a replicated DB's problems are **consensus** - and consensus needs only a **majority**, which is exactly why it does not block.

Paxos, then Spanner

Paxos is the best-known consensus protocol. After it, we can finally build the full Web service architecture from Lecture 1.



- **[Skeen-1981]** Dale Skeen. *Nonblocking Commit Protocols*. SIGMOD 1981, pp. **133-142** - proves 2PC blocks, and that a non-blocking protocol needs an extra phase. The citation this deck most needed.
- **[Skeen-1983]** Skeen & Stonebraker. *A Formal Model of Crash Recovery in a Distributed System*. IEEE TSE **9(3):219-228**, 1983 - formalises the uncertainty period.
- **[Gray-Lamport-2006]** Jim Gray & Leslie Lamport. *Consensus on Transaction Commit*. ACM TODS **31(1):133-160**, 2006 - Paxos Commit: 2PC with a replicated coordinator.
- **[Lampson-Sturgis-1979]** Butler Lampson & Howard Sturgis. *Crash Recovery in a Distributed Data Storage System*. **Unpublished** technical report, Xerox PARC, June 1979.
- **[Lampson-1981]** Butler Lampson (with Sturgis). *Atomic Transactions*. In *Distributed Systems - Architecture and Implementation*, LNCS **105**, Springer, 1981, ch. 11, pp. **246-265**.
- **[Mohan-1986]** Mohan, Lindsay & Obermarck. *Transaction Management in the R* Distributed Database Management System*. ACM TODS **11(4):378-396**, 1986 - presumed abort/commit optimizations.
- **[Bernstein-1987]** Bernstein, Hadzilacos & Goodman. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley, 1987, ch. 7 - the standard treatment; freely available from the authors.
- **[Corbett-2012]** Corbett et al. *Spanner: Google's Globally-Distributed Database*. OSDI 2012, pp. **251-264** - 2PC over Paxos groups. Lecture 9.