Distributed Transactions

Thoai Nam

High Performance Computing Lab (HPC Lab)

Faculty of Computer Science and Engineering

HCMC University of Technology

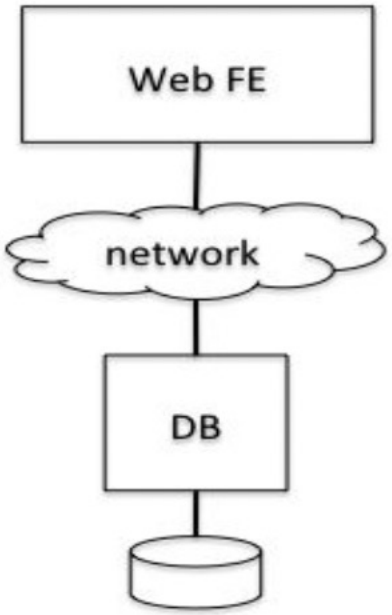
Slides

- Distributed Systems, Roxana Geambasu, Columbia University.

Contents

- Web service architecture
- Two-Phase Commit (2PC)

Web service architecture

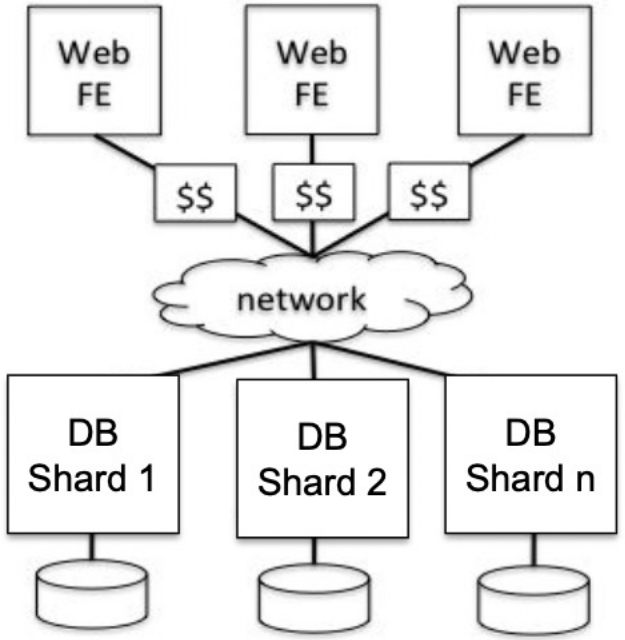


- Web front end (FE), database server (DB), network. FE is stateless, all state in DB
- Suppose the FE implements a banking application (supporting account transfers, listings, and other functionality)
- Suppose the DB supports ACID transactions and the FE uses transactions.

Question: How do we make this:

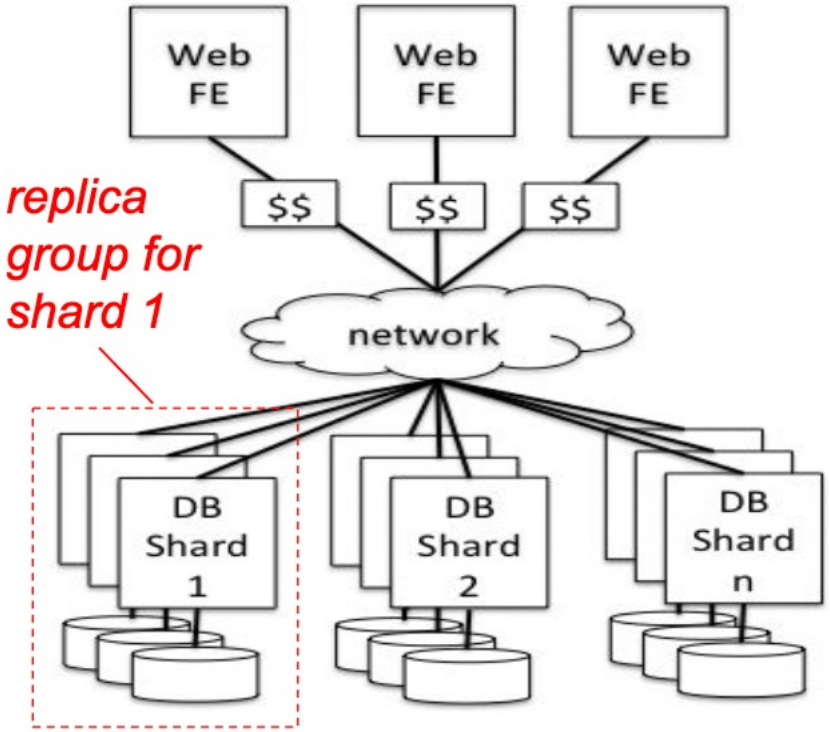
- Scalable?
- Fault tolerant?

Scalability: Sharding



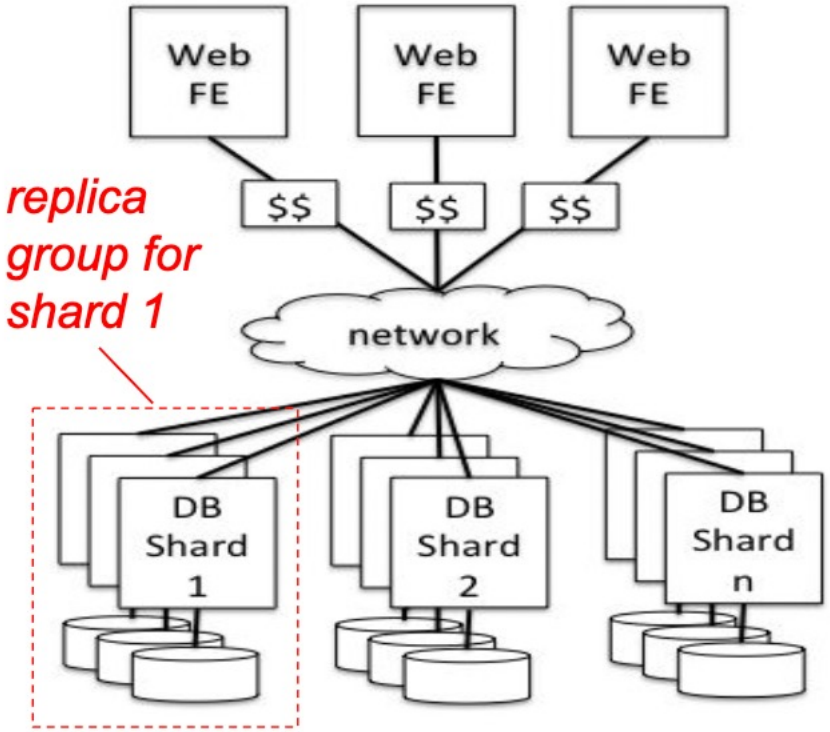
- FE and DB are both sharded:
  - FEs accept requests from end-users' browsers and process them concurrently
  - DB is sharded, say by user IDs.
- Suppose each DB backend is on its own transactional (ACID). Then, FE issues transactions against one or more DB shards.

Fault Tolerance: Replication



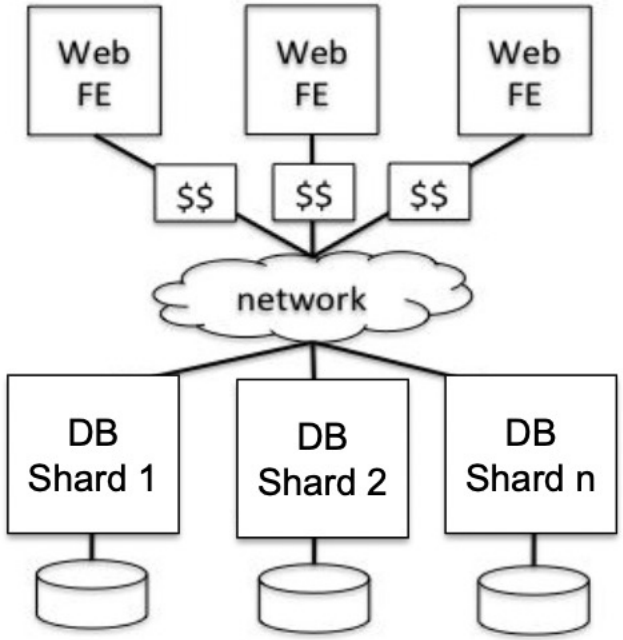
- FE is stateless, so the fact that it is shared means it's also replicated/fault tolerant
- But DB is stateful, so active replication is needed for each shard. Each shard is managed by a *replica group*, which cooperate to keep themselves up to date with respect to the updates
- FE sends requests for DB different shards go to different replica groups.

Challenges



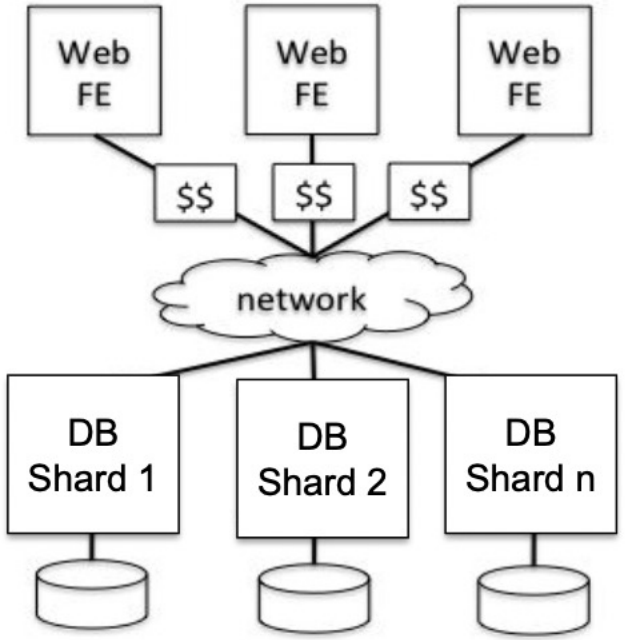
Question: What are the challenges of implementing ACID across the entire **sharded** & **replicated**, DB **service**?

Challenges due to Sharding



- Ignore replication. Implementing ACID across all DB shard servers:
  - Case 1: No transactions ever span multiple shards. Easy: individual DB shard performs transaction.
  - Case 2: Transactions can span multiple shards. Challenge: shards participating on any transaction need to **agree** on (1) whether or not to commit a transaction and (2) when to release the locks.

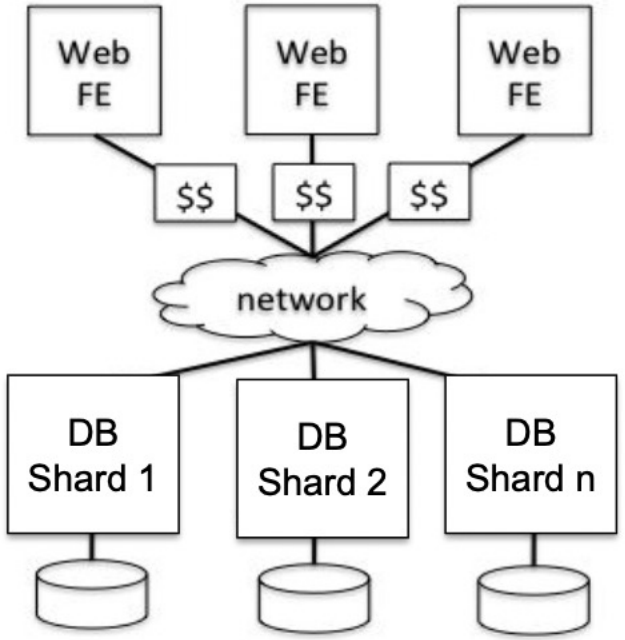
Challenges due to Sharding (cont.)



Example:

- Say FE service is a banking service that supports the TRANSFER and REPORT_SUM functions from the previous lecture.
- If the two accounts are stored on different shards, then the two operations (deduct from one and add to the other) will need to be executed either both or neither.
- Unfortunately, the two machines can fail, or decide to unilaterally abort, *INDEPENDENTLY.*

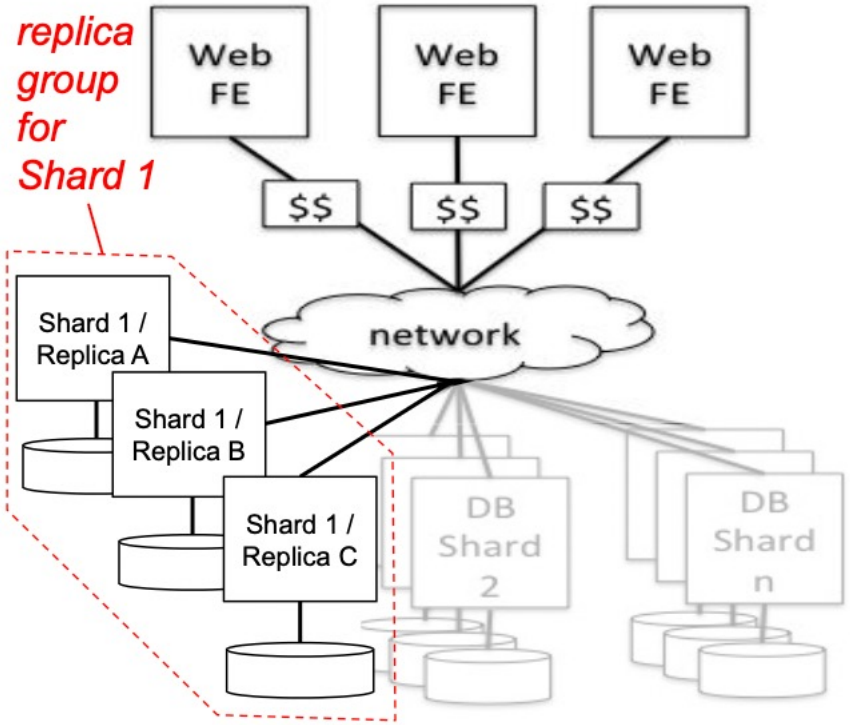
Challenges due to Sharding (cont.)



Example (continued):

- So, you need an agreement protocol, and in this case the most suitable is an **atomic commitment protocol** (why?).
- Well-known atomic commitment protocol: **two-phase commit**.

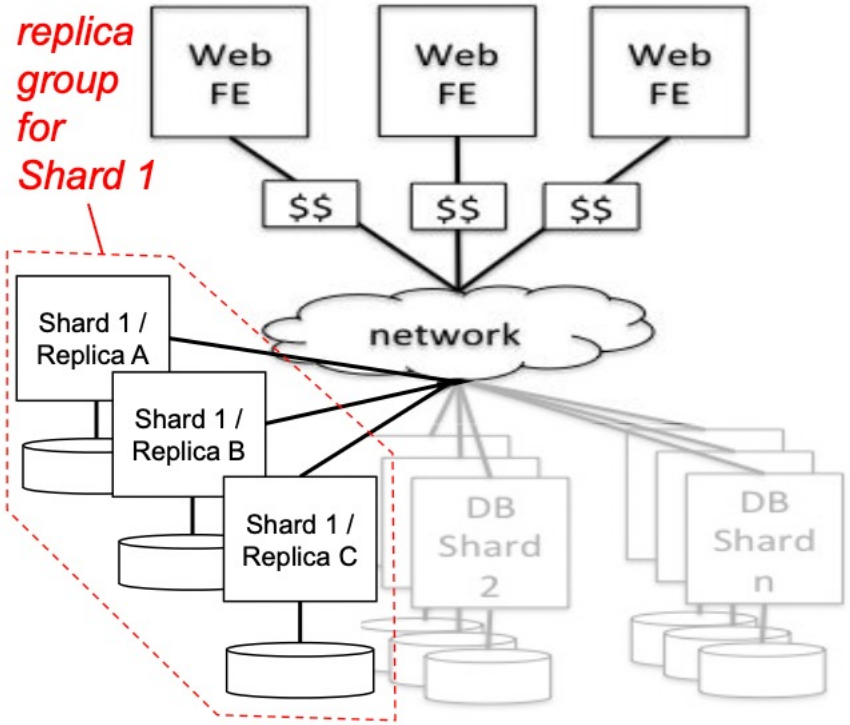
Challenges due to Replication



Ignore sharding. Implementing ACID across all replicas of a given shard:

- Challenge: All replicas of the shard must execute **all operations in the same order.**
- If the operations are deterministic, then agreeing on the order of keeps the copies of the database on the different replicas will evolve identically, i.e., they will all be kept consistent.

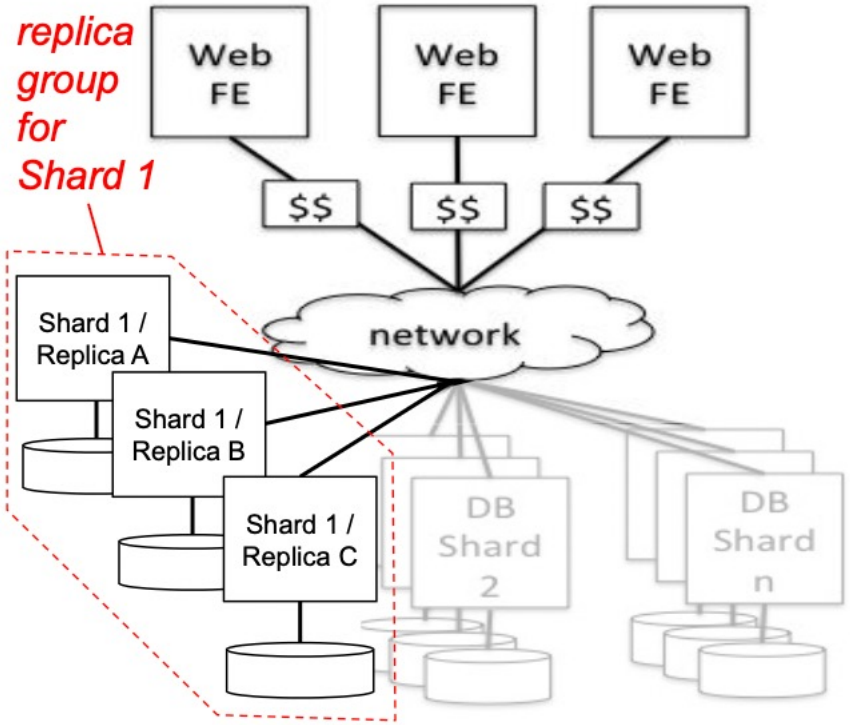
Challenges due to Replication (cont.)



Example:

- Suppose there are two transactions, each with a single operation, against the same cell in the database:
  - TX1: x += 1
  - TX2: x *= 2
- Internally, all three replicas are ACID databases, so they will serialize these transactions, e.g., either (TX1, TX2) OR (TX2, TX1).
- If Replica A processes (TX1, TX2) and Replica B processes (TX2, TX1), then after executing these transactions, the DB copies on the two replicas will diverge to x=8 and x=7, respectively.

Challenges due to Replication (cont.)

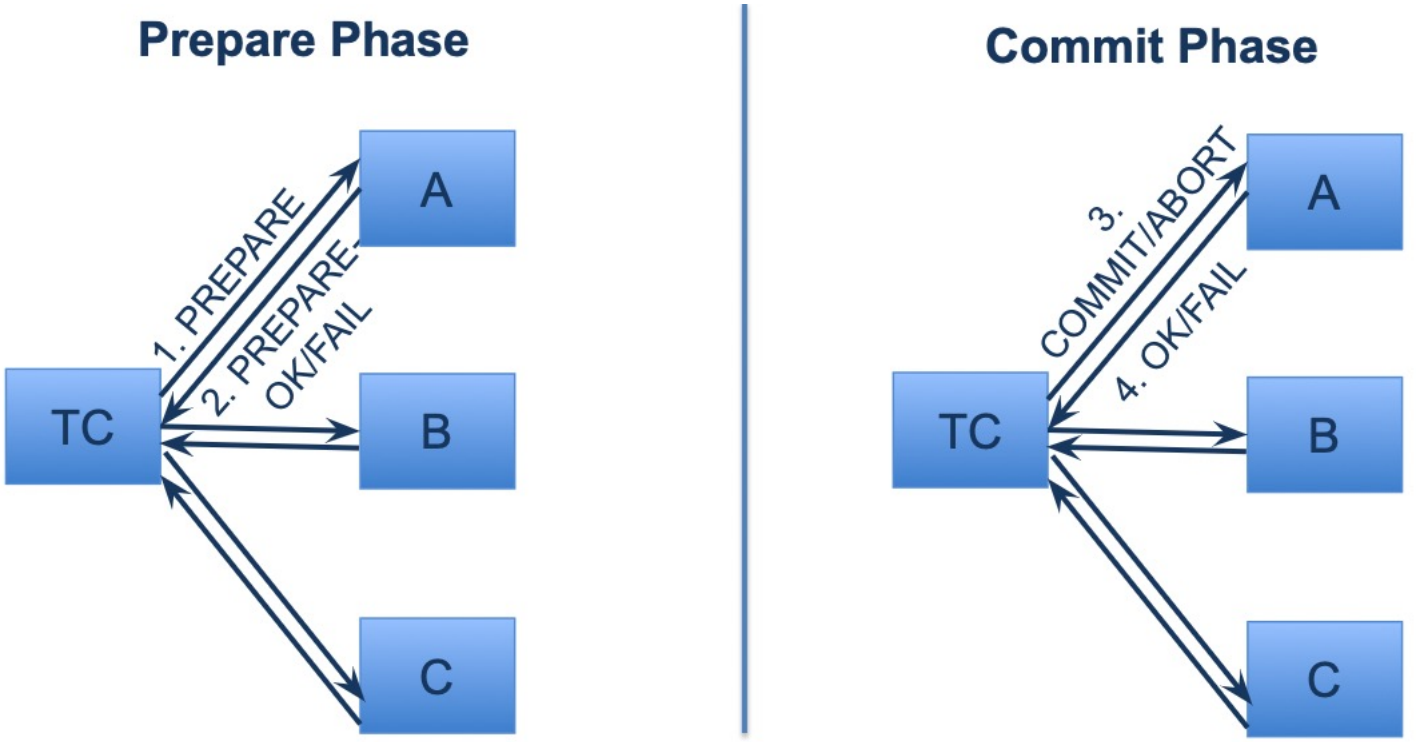


Example (continued):

- The problem of agreement on the order in which to execute operations can be cast as an instance of the **consensus problem** (why?)
- Well known consensus protocol: **Paxos** (see in consensus).

Two-Phase Commit (2PC)

Two-Phase Commit (2PC)



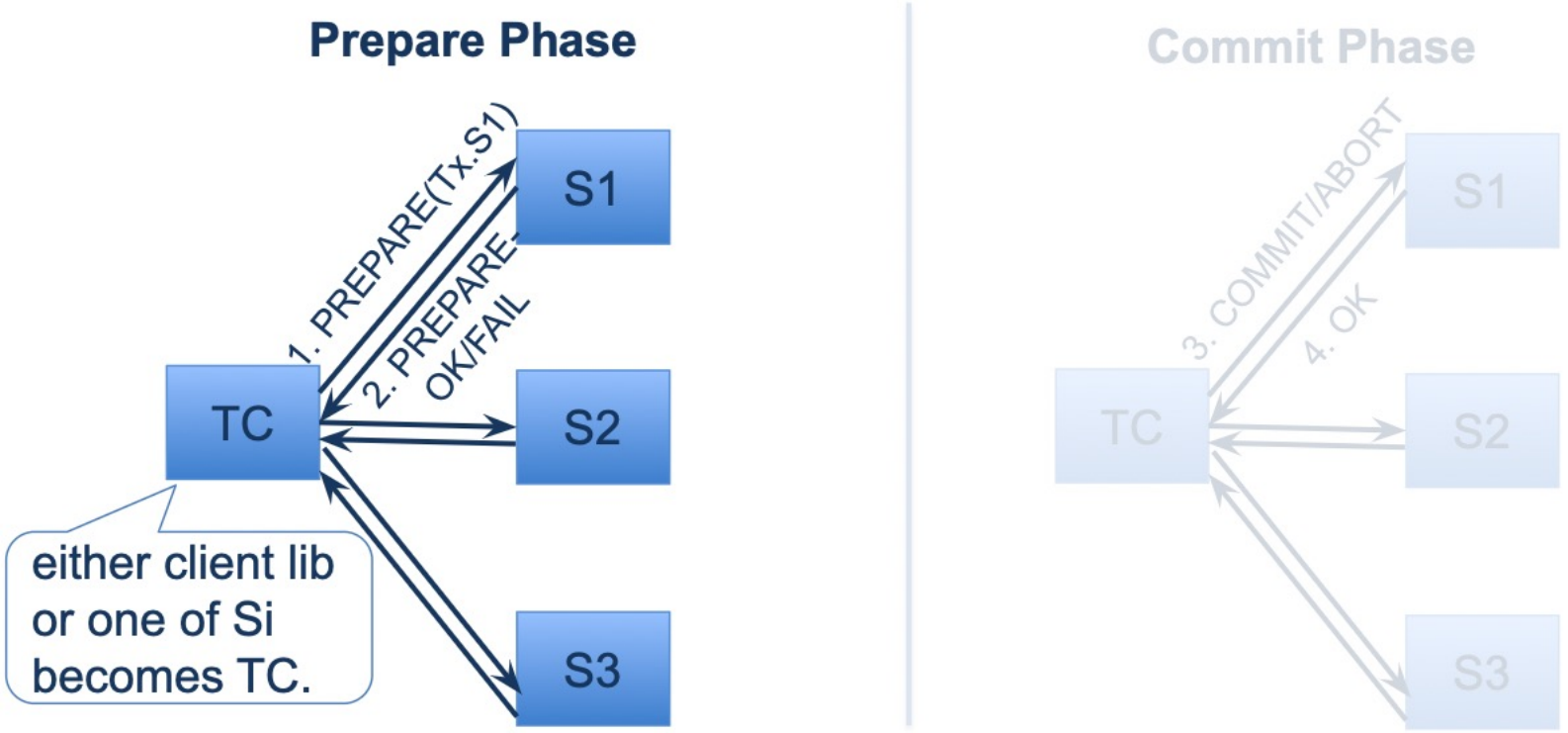
2PC for distributed transactions

How 2PC integrates with WAL, 2PL that we studied for local transactions.

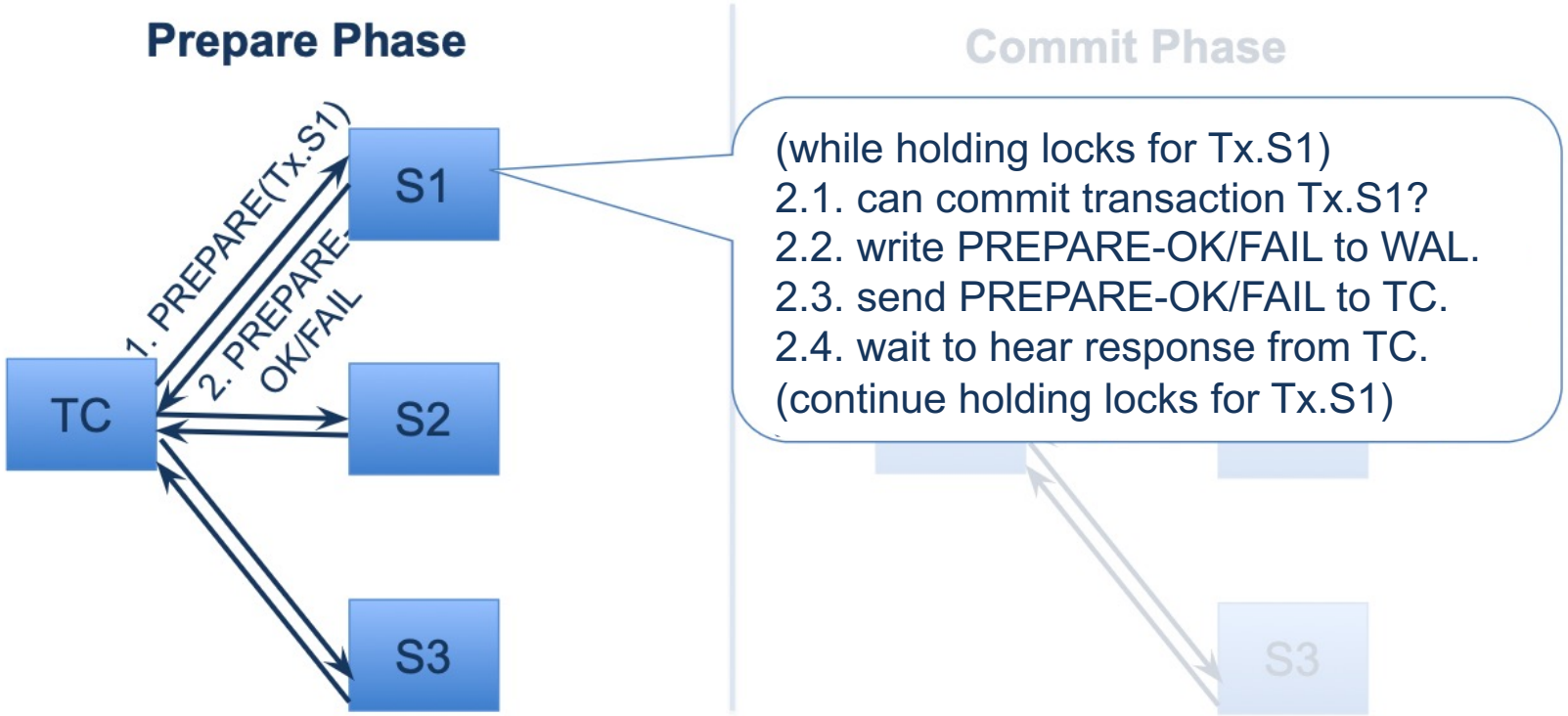
Here's a **rough** description of a client lib for distributed transactions:

- **begin()**: Client lib begin()'s a transaction on each separate shard. This produces a separate txID on each server (Tx.S1, Tx.S2,...).
- As part of the distributed TX, the client sends the operation to the corresponding shard server. Say op1 goes to S1, op2 goes to S2. Each server grabs local locks, adds op to their local WAL.
- **abort()**: Client sends the ABORT message to S1, S2, ...

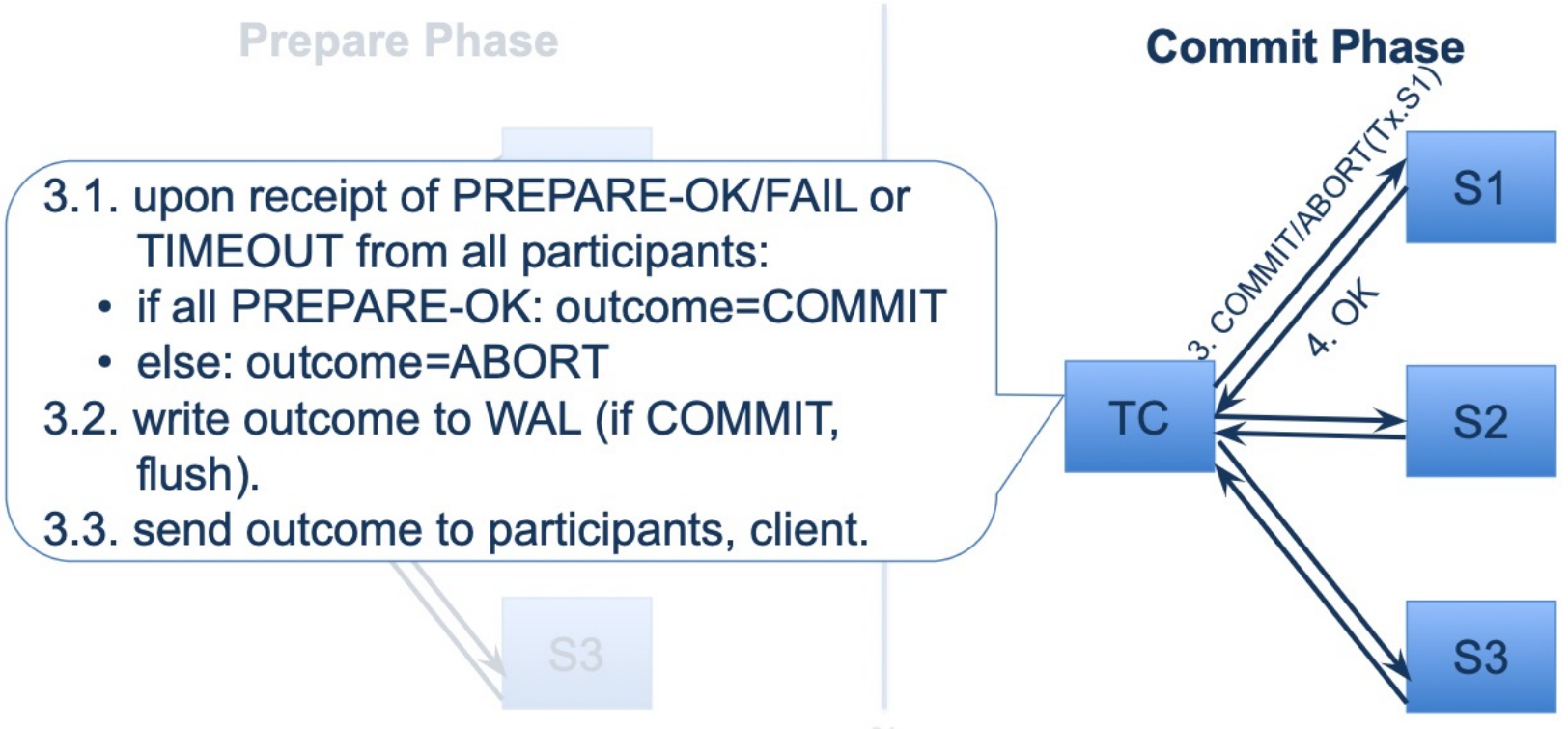
2PC **commit()**


Prepare Phase
1. PREPARE(Tx.S1)
2. PREPARE-OK/FAIL
TC
S1
S2
S3
either client lib or one of Si becomes TC.
Commit Phase
3. COMMIT/ABORT
4. OK
TC
S1
S2
S3

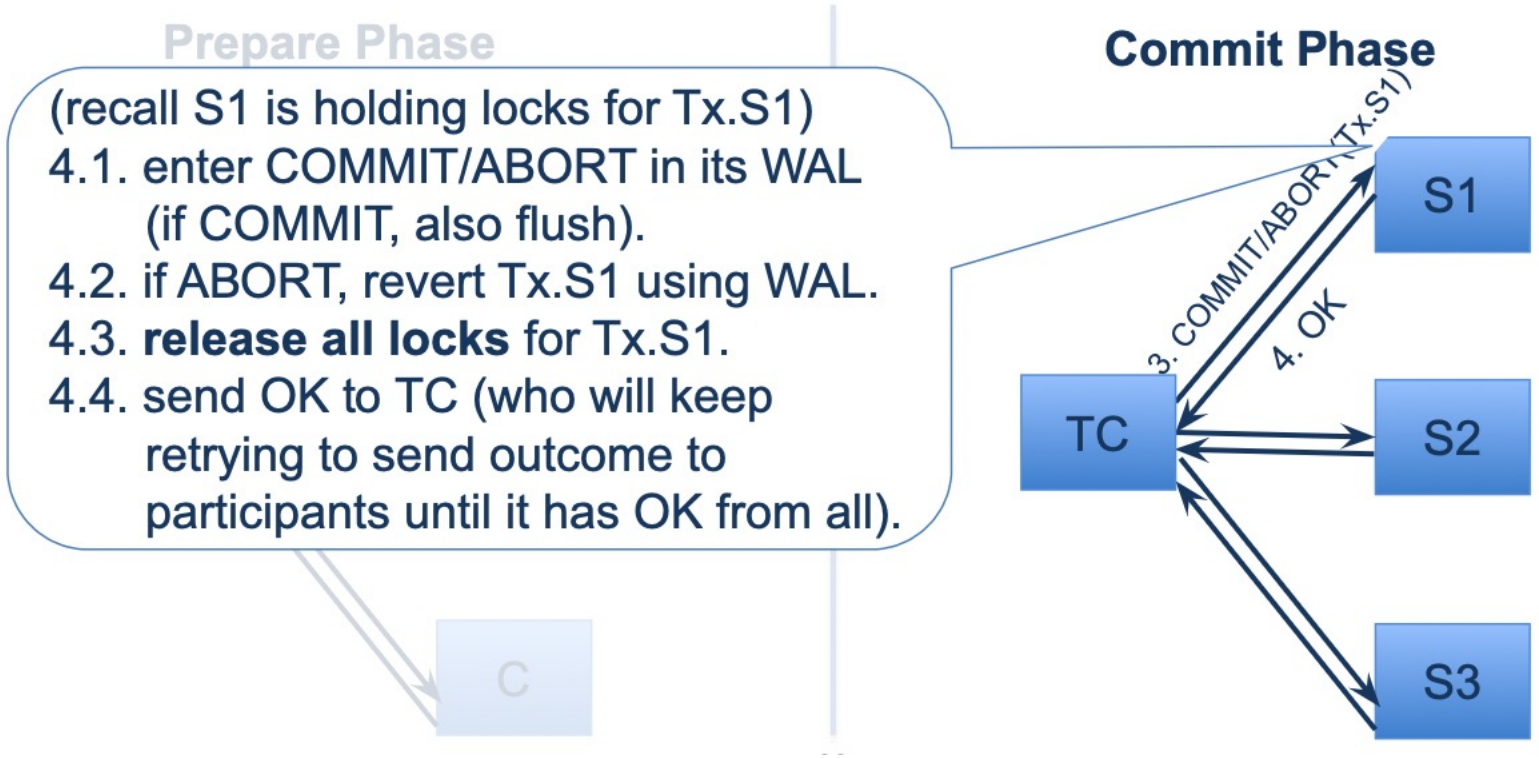
2PC **commit()**


Prepare Phase
Commit Phase
1. PREPARE(Tx.S1)
2. PREPARE-OK/FAIL
TC
S1
S2
S3
(while holding locks for Tx.S1)
2.1. can commit transaction Tx.S1?
2.2. write PREPARE-OK/FAIL to WAL.
2.3. send PREPARE-OK/FAIL to TC.
2.4. wait to hear response from TC.
(continue holding locks for Tx.S1)
S3

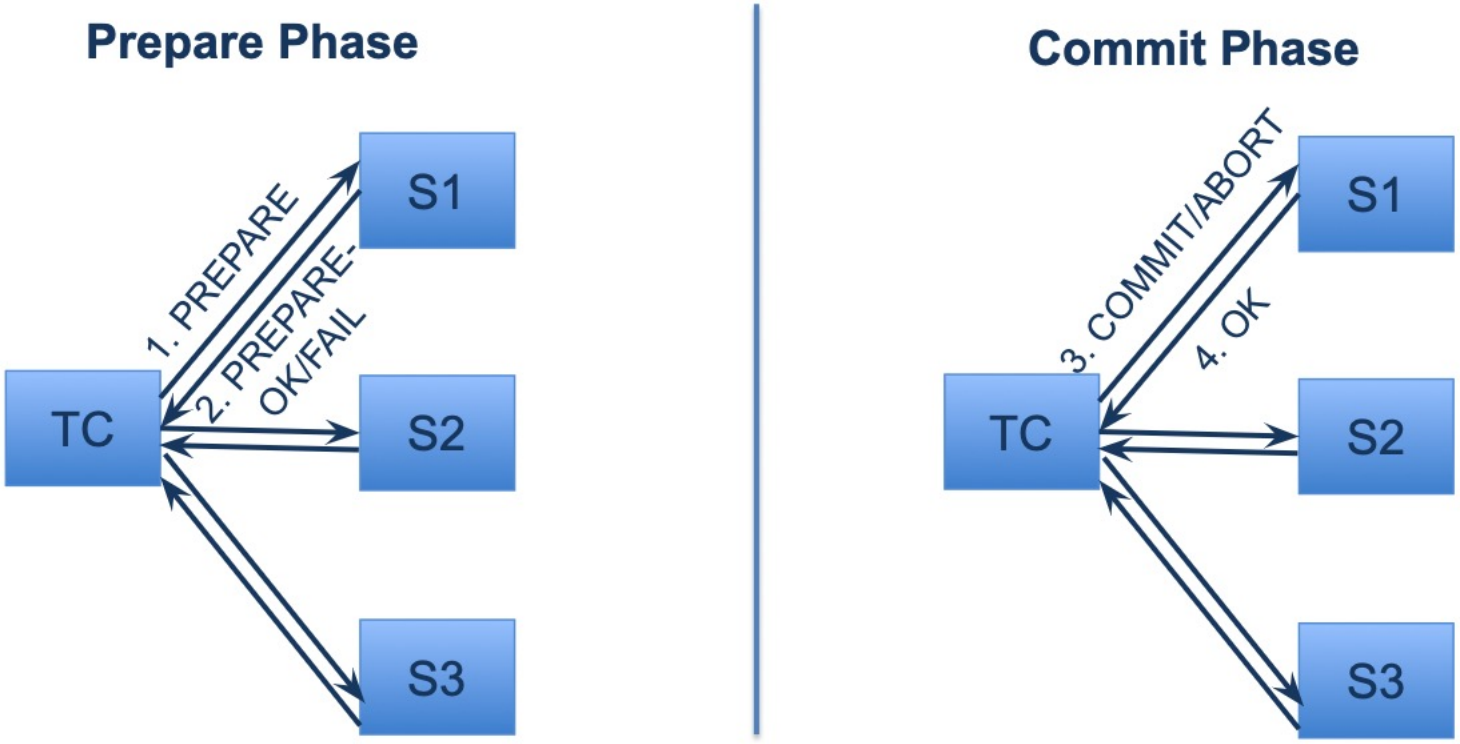
2PC **commit()**


Prepare Phase
Commit Phase
3.1. upon receipt of PREPARE-OK/FAIL or TIMEOUT from all participants:
• if all PREPARE-OK: outcome=COMMIT
• else: outcome=ABORT
3.2. write outcome to WAL (if COMMIT, flush).
3.3. send outcome to participants, client.
3. COMMIT/ABORT(Tx.S1)
4. OK
TC
S1
S2
S3
S3

2PC **commit()**


Prepare Phase
(recall S1 is holding locks for Tx.S1)
4.1. enter COMMIT/ABORT in its WAL
(if COMMIT, also flush).
4.2. if ABORT, revert Tx.S1 using WAL.
4.3. release all locks for Tx.S1.
4.4. send OK to TC (who will keep
retrying to send outcome to
participants until it has OK from all).
C
Commit Phase
3. COMMIT/ABORT (Tx.S1)
4. OK
TC
S1
S2
S3

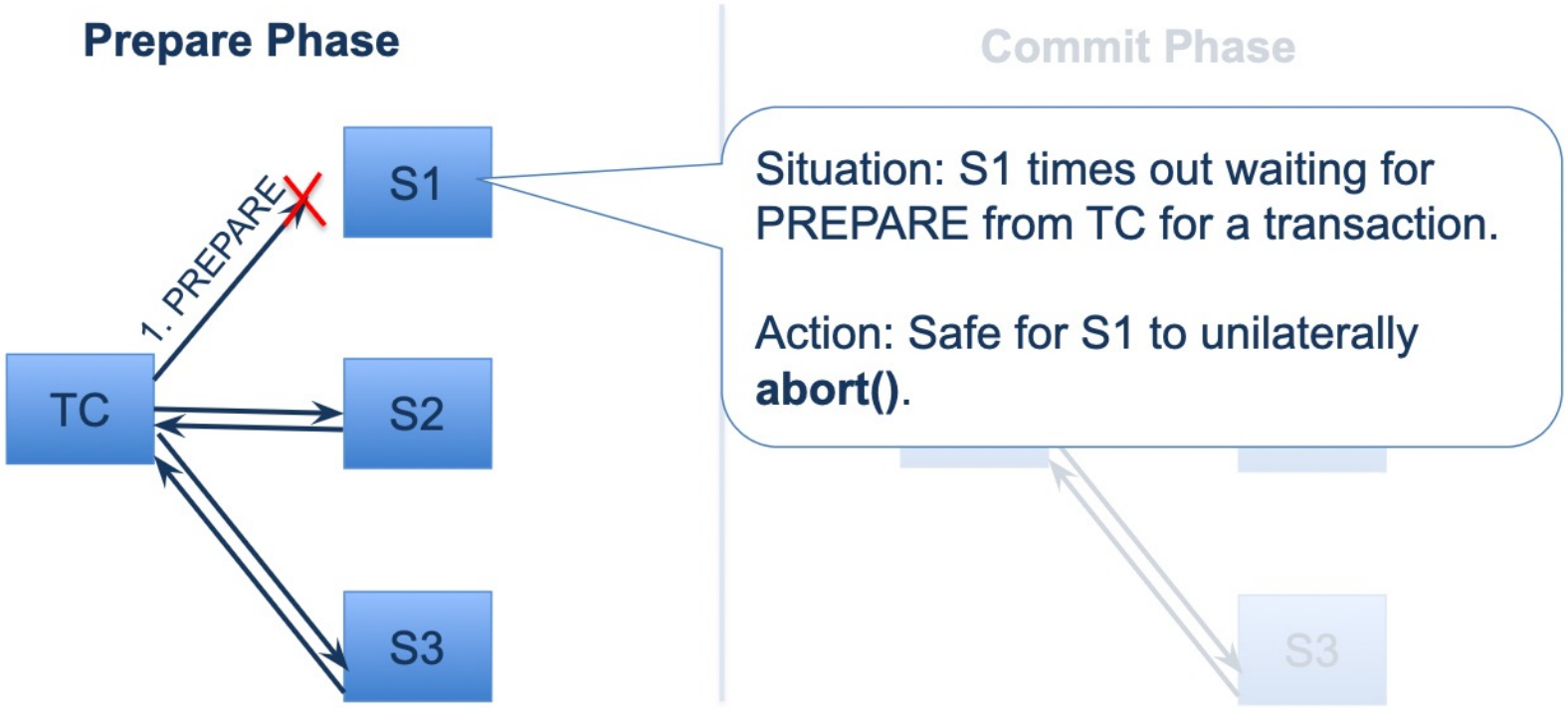
Timeouts and Failures



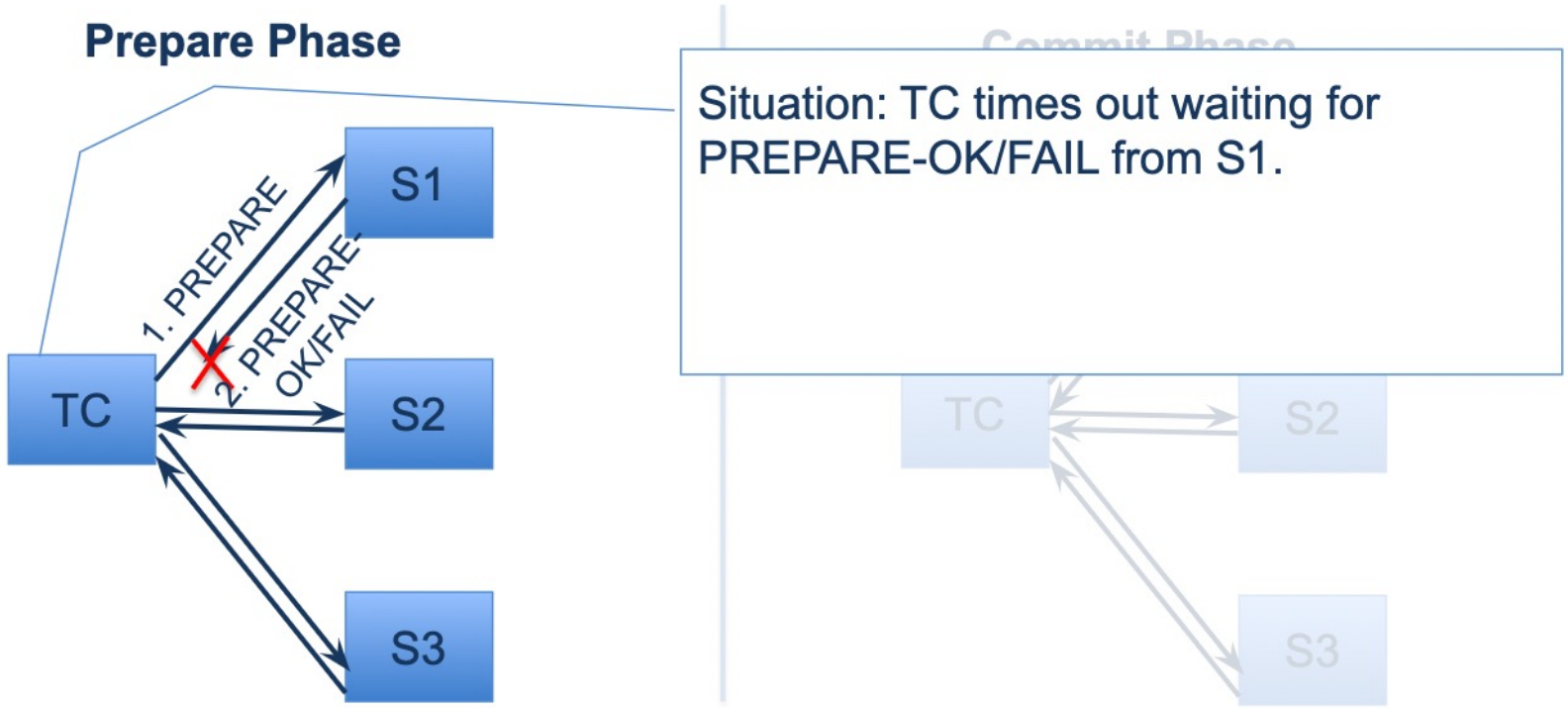
Timeouts


Prepare Phase
Commit Phase
S1
1. PREPARE
TC
S2
S3
Situation: S1 times out waiting for PREPARE from TC for a transaction.
S3

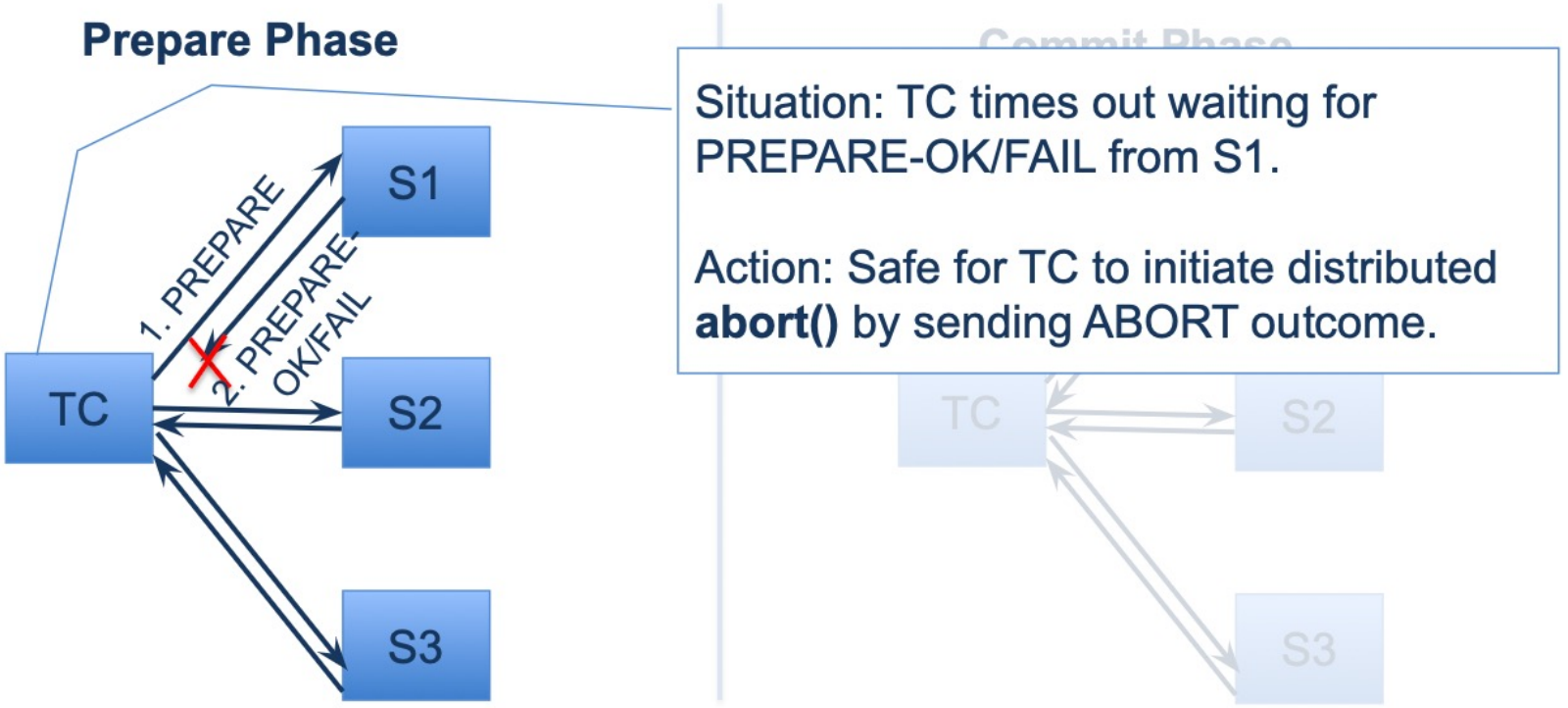
Timeouts


Prepare Phase
Commit Phase
S1
1. PREPARE
TC
S2
S3
Situation: S1 times out waiting for PREPARE from TC for a transaction.
Action: Safe for S1 to unilaterally abort().
S3

Timeouts


Prepare Phase
TC
S1
S2
S3
1. PREPARE
2. PREPARE-OK/FAIL
Situation: TC times out waiting for PREPARE-OK/FAIL from S1.

Timeouts


Prepare Phase
TC
S1
S2
S3
1. PREPARE
2. PREPARE-OK/FAIL
Situation: TC times out waiting for PREPARE-OK/FAIL from S1.
Action: Safe for TC to initiate distributed abort() by sending ABORT outcome.

Timeouts

Situation: S1 times out waiting for outcome from TC.
Action: **Is it safe for S1 to unilaterally commit or abort?**

Timeouts

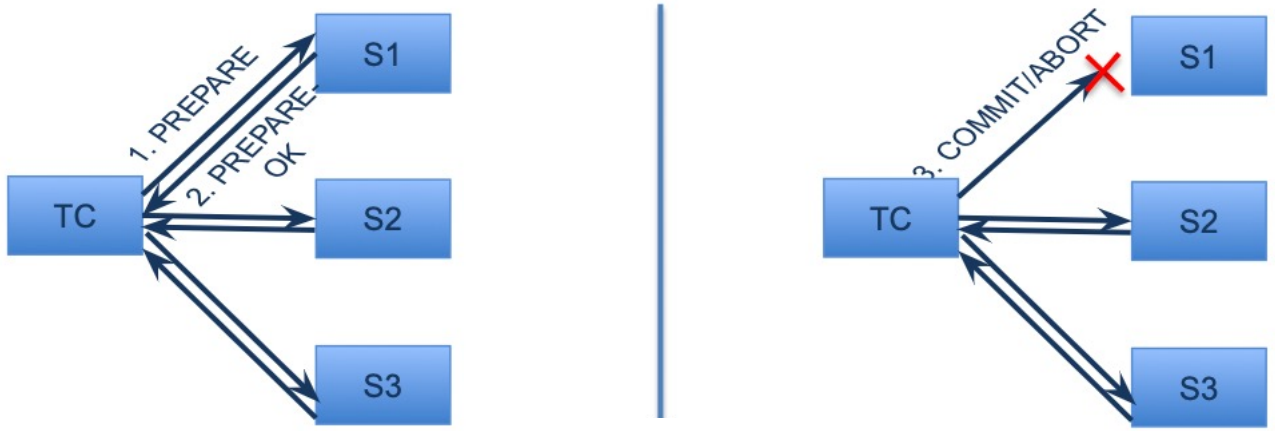
Situation: S1 times out waiting for outcome from TC.
- Case 1: S1 had sent **PREPARE-FAIL** in Prepare phase.
  Action: Safe for S1 to **unilaterally abort**

Timeouts

Situation: S1 times out waiting for outcome from.
• Case 2: S1 had sent **PREPARE-OK** in Prepare phase (S1 is said to be in the **uncertainty period**).
Action: Can't commit/abort. Runs a **termination protocol**.

Termination protocol

- Wait for TC to come back (might take a while and recall Si hold locks!).
- Could also ask other participants whether they got outcome.
  - If one did, they can all terminate the protocol accordingly.
  - If none did (e.g., TC died or got partitioned right before it send outcome), then participants are **BLOCKED** till TC comes back.



Failures

- Similar analysis applies for failures
- Some cases:
  - if participant is not in uncertainty period, on recovery, can decide what to do (unilaterally abort if no decision, otherwise do what decision is.)
  - if participant is in **uncertainty period**, it cannot decide on its own, must invoke the **termination protocol** (which, as before, may not actually terminate if TC fails).

2PC Limitations

2PC is blocking

- A process can block indefinitely in its uncertainty period until a TC or network failure is resolved.
- If TC is also a participant, then a single-site failure can cause 2PC to block indefinitely!
- And it blocks while each shard server is **holding locks**, preventing other transactions that don't even interact with the failed shard server from making progress!
- This is why 2PC is called a **blocking protocol** and *cannot be used as a basis for fault tolerance*.

2PC is expensive

- Time complexity: 3 message latencies on the critical path: PREPARE → PREPARE-OK/FAIL → ABORT/COMMIT.
- Message complexity: common case for n participants + 1 TC: 3n messages.
- That's expensive, esp. if shards are geo distributed.
- Optimizations, or adding an extra phase (3PC), cannot address the blocking/performance problems of 2PC while maintaining its semantic.