Sistemas Distribuídos

Distributed Transactions

Eurico Pedrosa

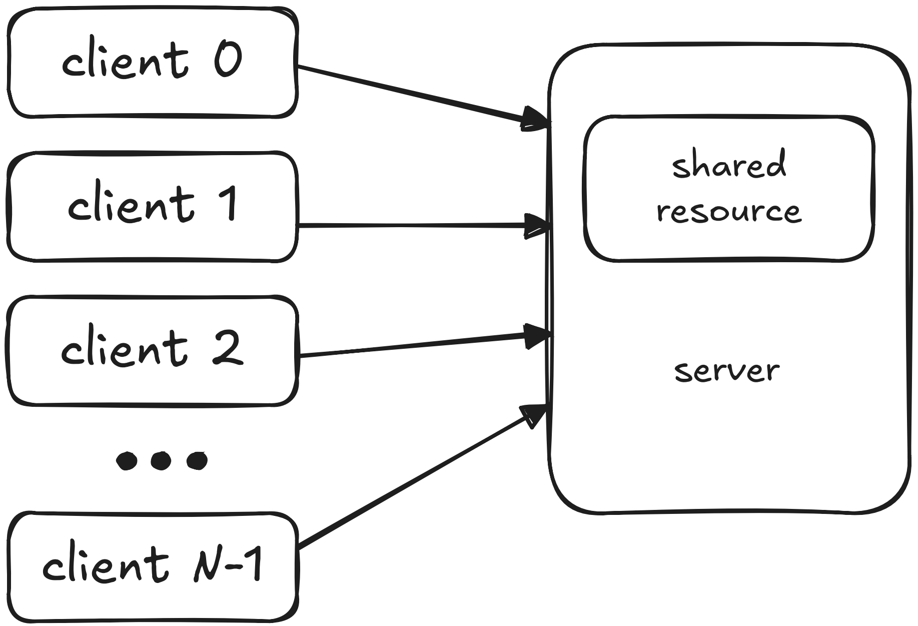
António Rui Borges

Universidade de Aveiro - DETI

2025-05-21

Distributed Transactions

What is a Transaction?



- A **transaction** is a **sequence of operations** (reads and writes) issued by a client.
- These operations target **registers in a shared region** managed by a server.
- A transaction must be treated as an **indivisible unit** by the server.

What is a Transaction?

Server Responsibilities

- Ensure **atomic execution** of read/write operations.
- Apply **all-or-nothing semantics**:
  - Save the result as a whole.
  - Or dismiss the entire transaction.
- Handle **failures gracefully**, ensuring either full commit or full rollback.

ACID Properties of Transactions (Härder and Reuter)

- **Atomicity**: A transaction must be **all or nothing**—either all operations succeed, or none are applied.
- **Consistency**: A transaction must transition the system from **one valid state to another**, preserving invariants.
- **Isolation**: Transactions must be executed **without interference**—their intermediate states must not be visible to other concurrent transactions.
- **Durability**: Once committed, the effects of a transaction must **persist permanently**, even in the event of failures.

Organization of Operations

- A **transaction** is created and managed by a **coordinator process** on the server side.
- Upon receiving a client request:
  - A transaction is **opened** by the coordinator.
  - A **unique transaction ID** is assigned.
- All subsequent **read-like and write-like operations**:
  - Must include the transaction ID.
  - Return a status:
    - **Success**: The client may continue.
    - **Failure**: The transaction is **aborted**.
- The client may **explicitly abort** the transaction at any time.

Completing the Transaction

- The transaction ends when the client issues an **end-of-transaction** command.
- If all operations succeeded:
  - The transaction is **committed**.
  - All changes become **permanent** in the shared region.
- If any operation failed:
  - An **abort** status is returned.
  - All effects of the transaction are **discarded**.
- After an abort, the client may **retry the transaction** later.

Concurrency Issues

- When **multiple transactions** access or modify the **same registers** in a shared region **concurrently**, **race conditions**- can occur, leading to **data inconsistencies**.
- It is essential to manage concurrent access in a **controlled manner** to ensure correctness.

Concurrency Issues

Conflicting Operations

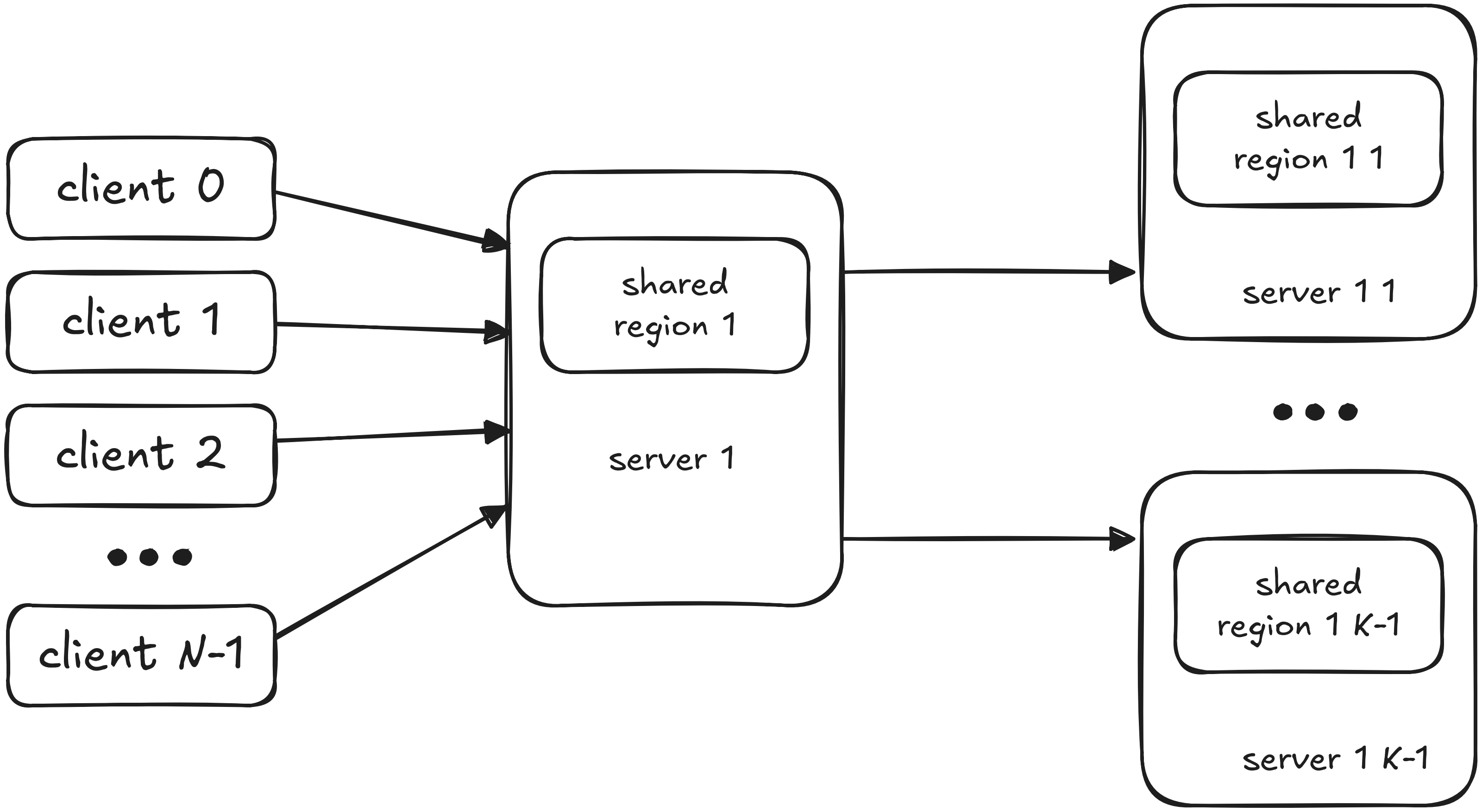
- Special attention must be given to **pairs of read-like and write-like operations** executed by **different transactions**- on the **same register**.
- If **two transactions attempt to write** to the same register:
  - The register must first be **locked** by one transaction.
  - The modification is made on a **local copy**.
  - When the transaction is **committed**:
    - The shared register is updated.
    - The lock is **released**.
  - Only then can the **second transaction proceed**.

Concurrency Issues

Summary

- Proper synchronization, e.g. **locking mechanisms**, is essential to:
  - Avoid **lost updates** and **inconsistent states**.
  - Ensure **serializability** and **isolation** between transactions.

Distributed Transactions


client 0
client 1
client 2
client N-1
shared
region 1
server 1
shared
region 1 1
server 1 1
shared
region 1 K-1
server 1 K-1

- In a **distributed transaction**, the **shared region** is partitioned across **multiple servers**.
- Each server manages a portion of the data and **executes part of the transaction**.
- The challenge: How to ensure that **all parts**- of the transaction are **committed together** or **aborted together**?

Nested Transactions

- A **distributed transaction** can be structured as a **nested transaction**:
  - A **top-level transaction** spawns **subtransactions**.
  - Each subtransaction may **execute on a different machine**.

Key Characteristics

- Subtransactions operate **independently** but are **logically part** of the parent transaction.
- Each subtransaction may **commit locally**, but the effects are **tentative** until the parent commits.

Nested Transactions

Abort Propagation

- If the **top-level (parent) transaction aborts**:
  - **All subtransactions must also be aborted**, regardless of their local commit status.
  - Ensures **global consistency and atomicity**.

Summary

- Nested transactions provide **modularity** and support for **partial execution** across distributed systems.
- Commit decisions are **hierarchically coordinated** to maintain the **ACID properties**.

Two-Phase Commit Protocol (2PC)

To coordinate the outcome across servers, a **two-phase commit protocol**- is used:

Phase 1: Voting

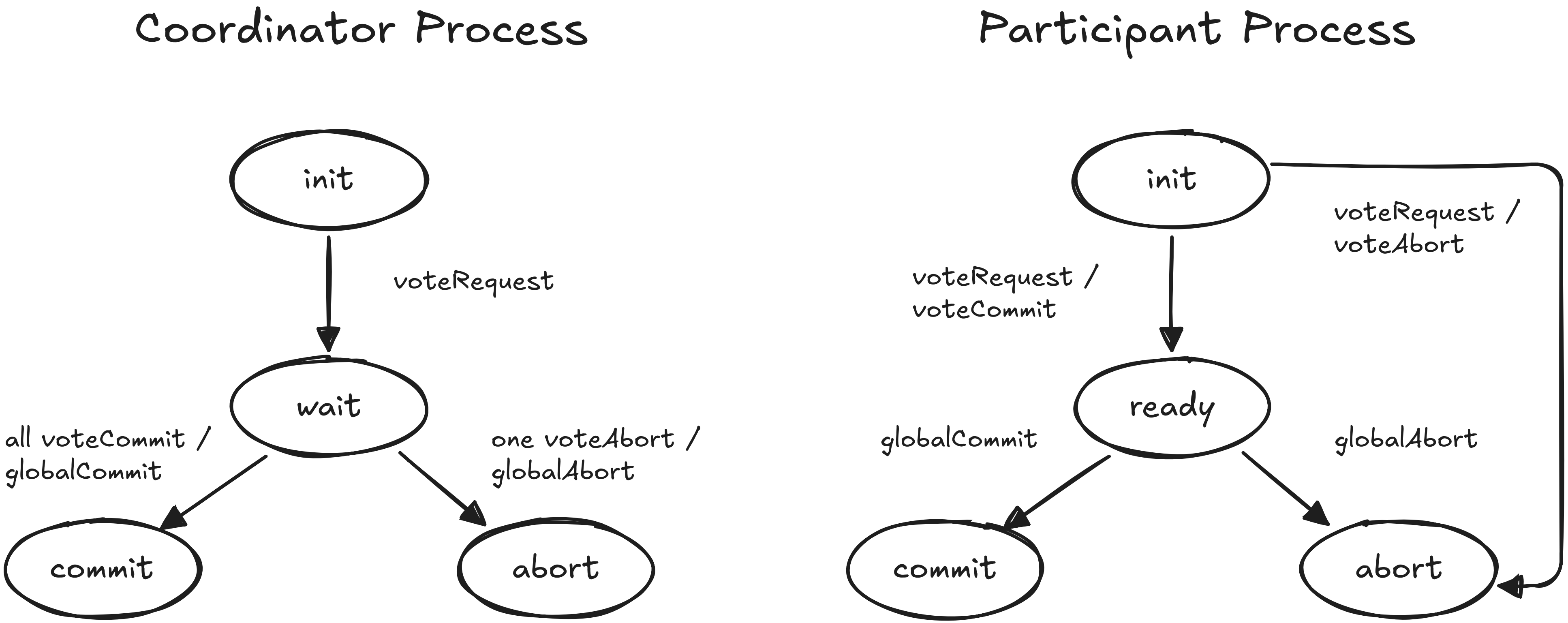
- The **coordinator process** sends a `voteRequest` message to **all participant processes**.
- Each participant responds:
  - `voteCommit` if it is ready to commit its part.
  - `voteAbort` if it cannot commit.

Two-Phase Commit Protocol (2PC)

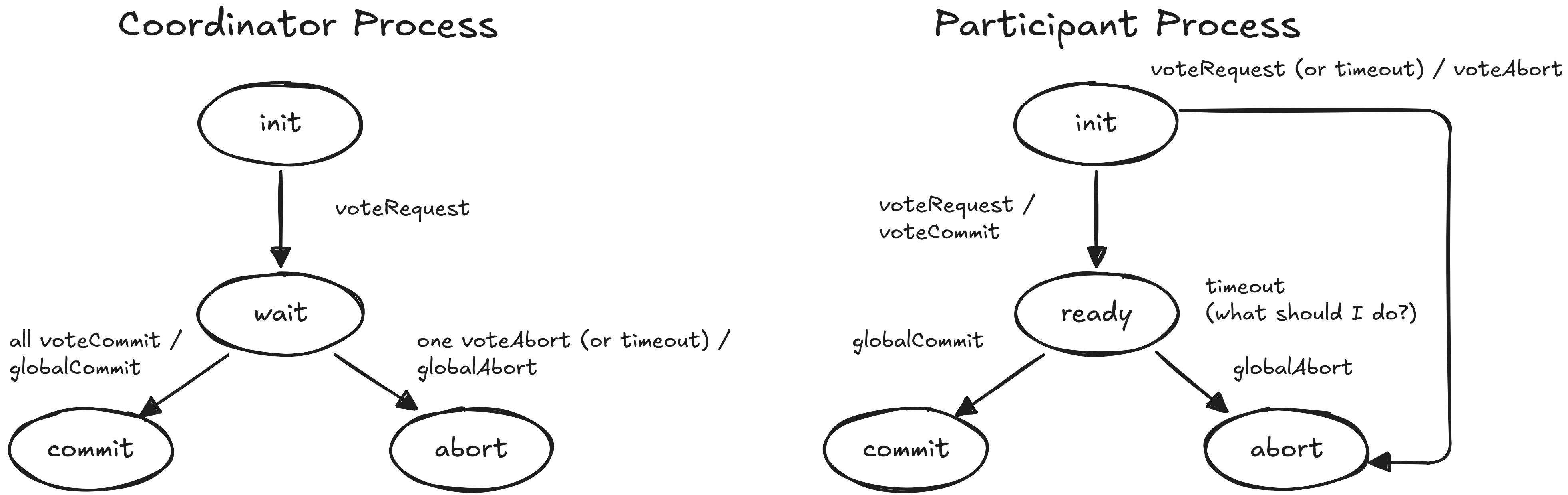
Phase 2: Decision

- The coordinator collects all votes and sends:
  - `globalCommit` to all participants if **all voted `voteCommit`**.
  - `globalAbort` if **at least one voted `voteAbort`**.
- Participation Action Upon receiving the final decision:
  - If `globalCommit`: each participant **commits** its local part.
  - If `globalAbort`: each participant **aborts** and discards its local changes.

Two-Phase Commit Protocol (2PC) (without failures)



Two-Phase Commit Protocol (2PC) (with failures)



Three-Phase Commit (3PC)

- **3PC** extends the **Two-Phase Commit (2PC)** protocol to avoid **blocking** in the event of a coordinator crash.
- Introduces an **intermediate phase** to ensure that participants can make progress without indefinite waiting.

Phase 1: Voting

- The **coordinator** sends a `voteRequest` to all **participants**.
- Each participant replies with:
  - `voteCommit` (ready to commit), or
  - `voteAbort` (cannot commit).

Three-Phase Commit (3PC)

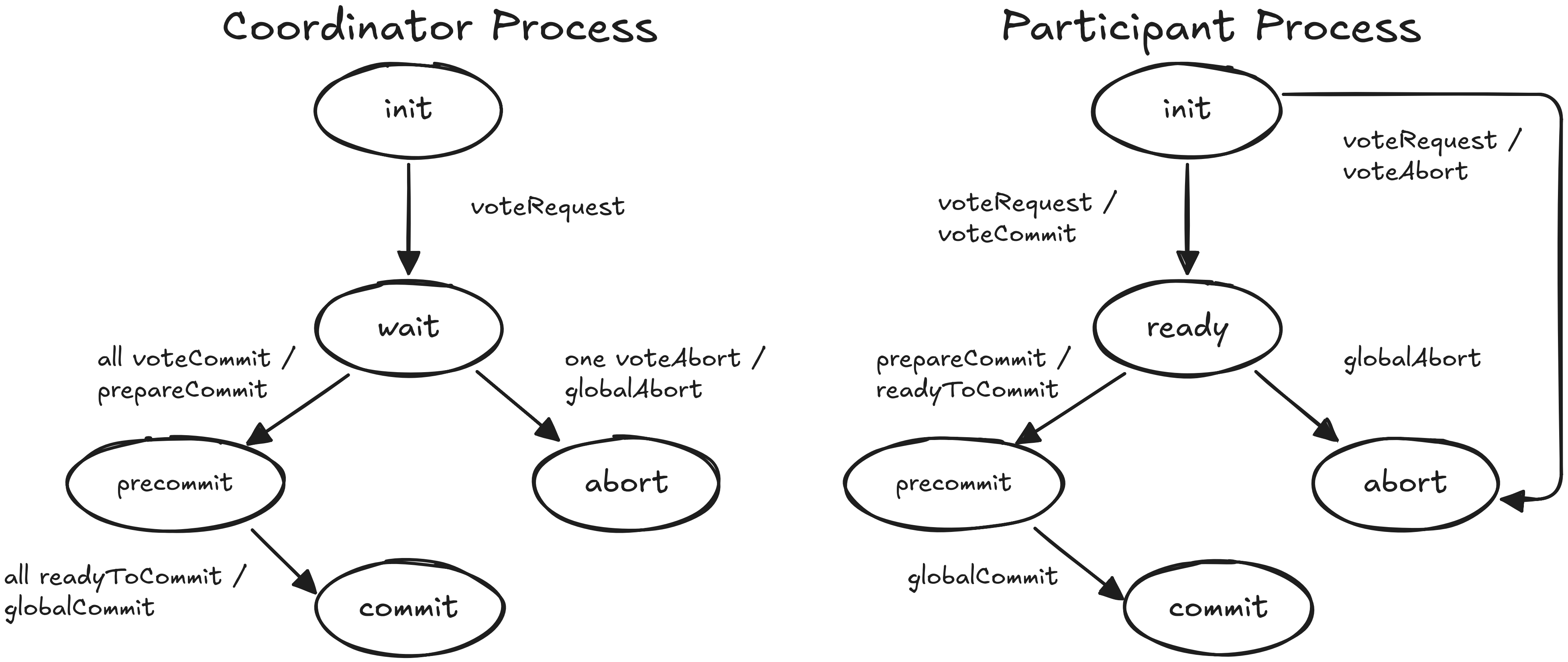
Phase 2: Pre-Commit

- If all participants vote `voteCommit`, the coordinator sends a `preCommit` message.
- Participants:
  - Acknowledge readiness.
  - Enter a **prepared state** (promise to commit but do not commit yet).

Phase 3: Do-Commit

- After receiving all acknowledgments, the coordinator sends a `globalCommit`.
- Participants **perform the actual commit**.

Three-Phase Commit (3PC) (without failures)



Three-Phase Commit (3PC)

Safety and Non-Blocking Guarantees

- **Safe intermediate states** prevent direct transitions to `COMMIT` or `ABORT`.
- **Progress is always possible**: no state requires indefinite waiting on others.

Failure Handling

- If the **coordinator crashes during PreCommit**:
  - Participants can safely **commit** (commit is known to be agreed upon).
- If a **participant times out**, it checks others:

Three-Phase Commit (3PC)

- If others are in `COMMIT` or `ABORT`: **adopt that decision**.
- If all are in `PRECOMMIT`: **safely commit**.
- If all are in `READY`: **abort to ensure safety**.

Why 3PC is Rarely Used

- Adds **significant complexity and overhead**.
- In practice, **2PC suffices** for most use cases in well-managed and reliable systems.

Suggested Reading

- M. van Steen and A.S. Tanenbaum, Distributed Systems, 4th ed., distributed-systems.net, 2023.