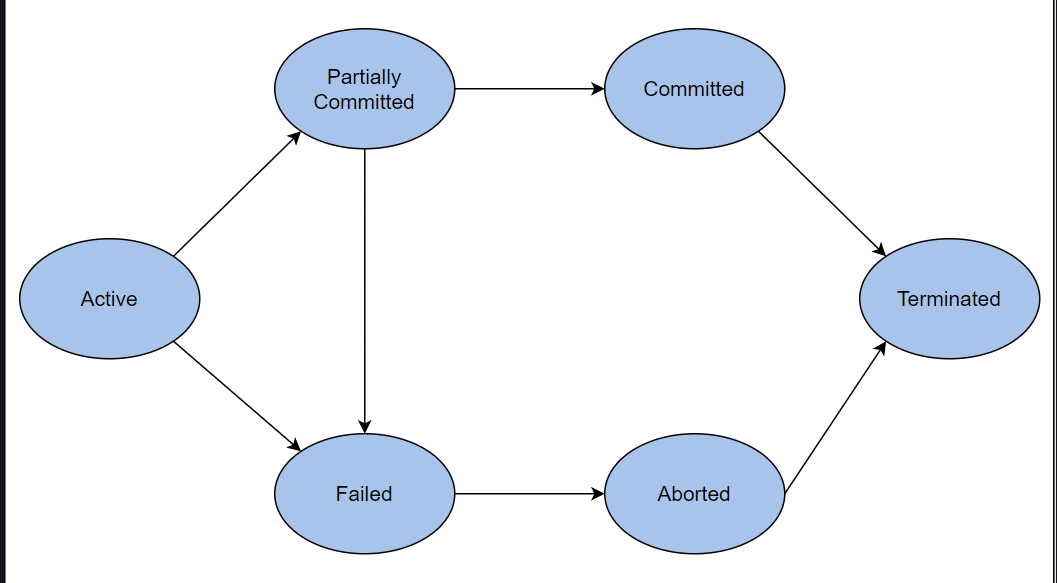
**What are transactions?**

A database transaction represents a logical unit of work. It is a sequence of operations executed as a single unit so that the database either executes all the operations successfully or rolls back. A transaction keeps the database state consistent with no partial updates. It also ensures that concurrent updates on the same piece of data do not leave the data in an inconsistent state.

**Transaction states**

Each transaction goes through a lifecycle of state transitions that dictate the end state of the transaction. These are the important transaction states:

* **Active**: The initial state of the transaction currently in execution. If the database executes all the operations successfully, it transitions to a partially committed state. Otherwise, it transitions to a failed state. The changes made by the sequence of operations reside in the buffer pool and are not considered durable.
* **Partially committed**: A database enters this state after executing the last operation of the transaction. However, since the data is not on disk, it is still not considered durable.
* **Committed**: A transaction transitions from partially committed to committed after the changes are persisted durably in a reliable store such as the disk.
* **Failed**: The database transitions the transaction from active or partially committed to a failed state on failures during execution. It indicates an inability to continue running operations in the transaction.
* **Aborted**: Once the transaction transitions to the failed state, the database should undo the changes made during the execution of the transaction. Once it rolls back the changes, the database transitions the transaction to an aborted state.
* **Terminated**: This is the final state of the transaction from the committed state after successful execution or aborted state after rollback.



ACID Properties:

A Atomicity

C Consistency

I Isolation

D Durability

Distributed Transactions:

1. 2 Phase Commit Protocol
2. 3 Phase Commit Protocol
3. SAGA

**2 Phase Commit Protocol:**

The 2-phase commit protocol consists of two phases, hence the name.

The protocol contains two different roles. Their names reflect their actual responsibilities in the protocol.

The coordinator is responsible for coordinating the different phases of the protocol

The participants correspond to all the nodes that participate in the transaction

**Note:** that one of the participants could also play the role of the coordinator.

**Voting phase:**

In this phase, the coordinator sends the transaction to all the participants. The participants execute the transaction up to

the point where they are supposed to commit.

In some cases, the operations of each transaction are executed separately and before the voting phase, which starts after

all the operations of a transaction has been executed. Agreement protocols like this usually involve some locking protocol as well,

so that other concurrent transactions cannot make participants that have already voted change their mind on whether they can commit or not.

For example, the 2-phase commit protocol can be combined with the 2-phase locking protocol.

Then, participants respond to the coordinator with a vote that shows if the transaction’s operations are executed successfully (“Yes” vote)

or there is some error that means the transaction cannot be committed (“No” vote).

**Commit phase:**

In this phase, the coordinator gathers all the votes from the participants. If all the votes are “Yes”, then the coordinator messages

the participants again with an instruction to commit the transaction.

Otherwise, if at least one vote is “No”, the coordinator instructs the participants to abort the transaction.

Finally, the participants reply with an acknowledgment and close this phase.

The fact that a unanimous positive vote is needed for a commit means that the transaction will either commit to all the participants, or

will be aborted to all of them (atomicity property).

The coordinator and the participants make use of a write-ahead-log, where they persist their decisions during the various steps so

that they can recover in case of a failure.

The coordinator also uses a timeout when waiting for the responses from the first phase. If the timeout expires, the coordinator

interprets this timeout as a No vote and considers the node as failed.

On the other hand, the participants do not apply any timeouts while waiting for the coordinator’s messages, since that could lead

to participants reaching different conclusions due to timing issues.

**Handling failures:**

Since the happy path is straightforward, let’s examine how the protocol handles various kinds of failures.

***Failure of a participant in the voting phase***

If a participant fails in the voting phase before replying to the coordinator, the coordinator will timeout waiting and assume a

No vote on behalf of this participant.

This means that the protocol will end up aborting the transaction.

***Failure of a participant in the commit phase***

In this scenario, a participant votes in the voting phase but then fails before it receives the message from the coordinator

and completes the transaction (either by committing or abort).

In this case, the protocol will conclude without this node. If this node recovers, later on, it will identify that pending transaction and communicate with the coordinator to find out what the result was, and conclude it in the same way.

So, if the result of the transaction is successful, any crashed participant will eventually find out upon recovery and commit it.

The protocol does not allow aborting it unilaterally. Thus, atomicity is maintained.

Some readers may have noticed that there is a chance that the participants may fail at the point they try to commit

the transaction and break their promise, e.g., because they are out of disk space. Indeed, this is true.

Thus, participants have to make the minimum work possible as part of the commit phase to avoid this.

For example, the participants can write all the necessary data on the disk during the first phase so that they can signal a

transaction is committed by doing minimal work during the second phase (e.g., flipping a bit).

**Network failures**

Network failures have similar results to those described previously, since timeouts make them equivalent to node failures.

Even though a 2-phase commit can handle all the aforementioned failures gracefully, there’s a single point of failure:

the coordinator.

**Blocking nature of 2-phase commit protocol**

Because of the blocking nature of the protocol, failures of the coordinator at specific stages of the protocol can bring the whole system to a halt.

More specifically, if a coordinator fails after sending a prepared message to the participants, the participants will block.

The participants will wait for the coordinator to recover and find out the outcome of the transaction, so that they commit or abort it as needed.

This means that failures of the coordinator can decrease the availability of the system significantly.

Moreover, if the data from the coordinator’s disk cannot be recovered (e.g., due to disk corruption),

the result of pending transactions cannot be discovered, and manual intervention might be needed to unblock the protocol.

**Usage of the 2-phase commit protocol:**

Despite the blocking nature of the protocol, the 2-phase commit is widely used.

A specification, called the eXtended Architecture (XA), has also been released.

In this specification, each of the participant nodes is referred to as resources, and they must implement the interface of a resource manager.

The specification also defines the concept of a transaction manager that acts as the coordinator that starts, coordinates, and ends transactions.

**Conclusion:**

The 2PC protocol satisfies the safety property that ensures all participants always arrive at the same decision (atomicity).

However, it does not satisfy the liveness property that implies it will always make progress.

**3 Phase Commit Protocol:**

**The problem with 2-phase commit protocol**

As we described previously, the main bottleneck of the 2-phase commit protocol was failures of the coordinator leading the system to a blocked state.

Ideally, we would like the participants to be able to take the lead in some way and continue the execution of the protocol in this case,

but this is not so easy.

The underlying reason is the fact that in the commit phase, the participants are not aware of the state of the other participants—only

the coordinator is So, taking the lead without waiting for the coordinator can result in breaking the atomicity property.

For instance, imagine the following scenario: in the commit phase of the protocol, the coordinator manages to send a commit (or abort) message

to one of the participants but then fails, and this participant also fails. If one of the other participants takes the lead, it will only be able

to query the live participants. So, it will be unable to make the right decision without waiting for the failed participant (or the coordinator)

to recover.

**Tackling the 2PC problem with 3-phase commit protocol**

The 2-phase commit problem could be tackled by splitting the first round (voting phase) into 2 sub-rounds, where the coordinator first

communicates the votes result to the nodes, waits for an acknowledgment, and then proceeds with the commit or abort message.

In this case, the participants would know the result from the votes and complete the protocol independently in case of a coordinator failure.

This is essentially the 3-phase commit protocol (3PC).

Wikipedia contains a detailed description of the various stages of the protocol and a nice visual demonstration.

Feel free to refer to this resource for additional study on the protocol.

**Benefit of 3PC**

The main benefit of this protocol is that the coordinator stops being a single point of failure.

In case of a coordinator failure, the participants are able to take over and complete the protocol.

A participant taking over can commit the transaction if it receives a prepare-to-commit, knowing that all the participants have voted “Yes”.

If it does not receive a prepare-to-commit, it can abort the transaction, knowing that no participant has committed, without all the

participants receiving a prepare-to-commit message first.

As a result, the 3PC protocol increases availability and prevents the coordinator from being a single point of failure.

However, this comes at the cost of correctness, since the protocol is vulnerable to failures such as network partitions.

In this case, one side of the partition has participants that receive a prepare-to-commit and continue with committing the transaction.

However, the participants at the other side of the partition do not receive a prepare-to-commit message and, thus,

unilaterally abort the transaction.

This can seem like a failure case that is very unlikely to happen. However, the consequences are disastrous if it happens,

since the system is at an inconsistent state after the network partition is fixed. The atomicity property of the transaction has been violated.

**Conclusion**

The 3PC protocol satisfies the liveness property that ensures it will always make progress,

at the cost of violating the safety property of atomicity.

**Long Live Transactions and SAGA pattern:**

Long-lived transactions

There is a specific class of transactions, called long-lived transactions (LLT).

These are transactions that by their nature have a longer duration in the order of hours or even days, instead of milliseconds. This can happen because this transaction processes a large amount of data, requires human input to proceed, or needs to communicate with third party systems that are slow.

**Examples of LLTs**

* Batch jobs that calculate reports over big datasets
* Claims at an insurance company, containing various stages that require human input
* An online order of a product that spans several days from order to delivery

As a result, running these transactions using the common concurrent mechanisms degrades performance significantly, since they need to hold resources for long periods of time, while not operating on them.

Sometimes, long-lived transactions do not really require full isolation between each other, but they still need to be atomic, so that consistency is maintained under partial failures. Thus, researchers came up with a new concept: the saga.

However, it’s guaranteed that either all of the transactions will succeed, or none of them will, maintaining the atomicity guarantee.

**Benefits of the saga**

The concept of saga transactions can be really useful in distributed systems. As demonstrated in the previous sections, distributed transactions are generally hard and can only be achieved by making compromises on performance and availability.

There are cases where we can use a saga transaction instead of a distributed transaction. This will satisfy all of our business requirements while keeping our systems loosely coupled and achieving good availability and performance.

**Example scenario**

Let’s imagine we are building an e-commerce application, where every order of a customer requires several discrete steps: credit card authorization, checking warehouse inventory, item shipping, invoice creation and delivery, etc.

One approach could be to perform a distributed transaction across all these systems for every order. However, in this case, the failure of a single component (i.e., the payment system) could potentially bring the whole system to a halt.

An alternative, leveraging the saga pattern, would be to model the order operation as a saga operation, consisting of all these sub-transactions, where each of them is associated with a compensating transaction.

For example, debiting a customer’s bank account could have a compensating transaction that would give a refund. Then, we can build the order operation as a sequential execution of these transactions, as shown in the following transactions. In case any of these transactions fail, we can rollback the transactions that have been executed and run their corresponding compensating transactions.

