Consensus on Transaction Commit

wu

May 11, 2024

1 Transaction Commit

We assume a set of RM processes, each beginning in a **working** state. The goal of the protocol is for the RMs all to reach a **committed** state or all to reach an **aborted** state. Two safety requirements:

1. **Stability**: Once an RM has entered the **committed** or **aborted** state, it remains in that state forever.
2. **Consistency**: It is impossible for one RM to be in the **committed** state and another to be in the **aborted** state.

Each RM also has a **prepared** state. We require that

- An RM can enter the **committed** state only after all RMs have been in the *prepared* state.

These requirements imply that the transaction can commit, meaning that all RMs reach the **committed** state, only by the following sequence of events:

- All the RMs enter the **prepared** state, in any order
- All the RMs enter the **committed** state, in any order

The protocol allows the following event that prevents the transaction from committing:

- Any RM in the **working** state can enter the **aborted** state.

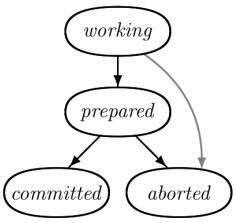

Figure 1: The state-transition diagram for a resource manager. It begins in the *working* state, in which it may decide that it wants to abort or commit. It aborts by simply entering the *aborted* state. If it decides to commit, it enters the *prepared* state. From this state, it can commit only if all other resource managers also decided to commit.

The goal of the algorithm is for all RMs to reach the committed or aborted state, but this cannot be achieved in a non-trivial way if RMs can fail or become isolated through communication failure. Moreover, the classic theorem of Fischer, Lynch, and Paterson implies that a deterministic, purely asynchronous algorithm cannot satisfy the stability and consistency conditions and still guarantee progress in the presence of even a single fault. We therefore require progress only if timeliness hypotheses are satisfied. Our two liveness requirements for a transaction commit protocol are:

1. **Non-Triviality**: If the entire network is nonfaulty throughout the execution of the protocol, then
   (a) if all RMs reach the **prepared** state, then all RMs eventually reach the **committed** state
   (b) if some RM reaches the **aborted** state, then all RMs eventually reach the **aborted** state
2. **Non-Blocking**: If, at any time, a sufficiently large network of nodes is nonfaulty for long enough, then every RM executed on those nodes will eventually reach either the **committed** or **aborted** state.

To specify a transaction commit protocol, we need to specify its set of legal behavious, where a behaviour is a sequence of system states. We specify the safety properties with an initial predicate and a next-state relation that describes all possible steps.

The initial predicate asserts that all RMs are in thw **working** state. To define the next-state relation, we first define two state predicates:

- **canCommit**: True iff all RMs are in the **prepared** or **committed** state
- **notCommitted**: True iff no RM is in the **committed** state.

The next-state relation asserts that each step consists of one of the following two actions performed by a single RM:

- **Prepare**: The RM can change from the **working** state to the **prepared** state
- **Decide**: If the RM is in the **prepared** state and **canCommit** is true, then it can transition to the **committed** state; and if the RM is in either the **working** or **prepared** state and **notCommitted** is true, then it can transaction to the **aborted** state.

2 Two-Phase Commit

2.1 The Protocol

The Two-Phase Commit protocol is an implementation of transaction commit that uses a **transaction manager** (TM) process to coordinate the decision-making procedure.

The RMs have the same states as in the specification of transaction commit. The TM has the following states: **init**, **preparing**, **committed**, and **aborted**.

1. The Two-Phase Commit protocol starts when an RM enters the **prepared** state and sends a **Prepared** message to the TM.
2. Upon receipt of the **Prepared** message, the TM enters the **preparing** state and sends a **Prepare** message to every other PM.
3. Upon receipt of the **Prepare** message, an RM that is still in the **working** state can enter the **prepared** state and send a **Prepared** message to the TM.
4. When it has received a **Prepared** message from all RMs, the TM can enter the **committed** state and send **Commit** messages to all the other processes.
5. The RMs can enter the **committed** state upon receipt of the **Commit** message from the TM.

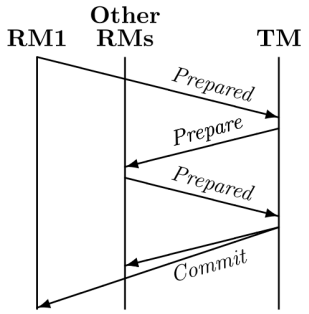

Figure 2: The message flow for Two-Phase Commit in the normal failure-free case, where RM1 is the first RM to enter the *prepared* state.

An RM can spontaneously enter the **aborted** state if it is in the working state; and the TM can spontaneously enter the **aborted** state unless it is in the **committed** state. When the TM aborts, it sends an **abort** message to all RMs. Upon receipt of such a message, an RM enters the **aborted** state. In an implementation, spontaneous aborting can be triggered by a timeout.

2.2 The Cost of Two-Phase Commit

Let N be the number of RMs. The Two-Phase Commit protocol sends the following sequence of messages in the normal case:

- The initial RM enters the prepared state and sends a **Prepared** message to the TM. (1 message)
- The TM sends a **Prepare** message to every other RM. ($N-1$ messages)
- Each other RM sends a **Prepared** message to the TM. ($N-1$ messages)
- The TM sends a **Commit** message to every RM. (N messages)

It is typical for the TM to be on the same node as the initiating RM, leaving $3N - 3$ messages and three message delays.

In addition to the message delays, the two-phase commit protocol incurs the delays associated with writes to stable storage: the write by the first RM to prepare, the writes by the remaining RMs when they prepare, and the write by the TM when it makes the commit decision. This can be reduced to two write delays by having all RMs prepare concurrently.

2.3 The Problem with Two-Phase Commit

The failure of the TM can cause the protocol to block until the TM is repaired. In particular, if the TM fails right after every RM has sent a Prepared message, then the other RMs have no way of knowing whether the TM committed or aborted the transaction.

A non-blocking commit protocol is one in which the failure of a single process does not prevent the other processes from deciding if the transaction is committed or aborted.

3 Paxos Commit

3.1 The Paxos Consensus Algorithm

Processes are called **acceptors** here. It can be shown that, without strict synchrony assumptions, $2F + 1$ acceptors are needed to achieve consensus despite the failure of any F of them.

Paxos uses a series of ballots numbered by nonnegative integers, each with a predetermined coordinator process called the **leader**. The leader of ballot 0 is called the **initial** leader. In the normal, failure-free case when the initial leader receives a proposed value, it sends a phase 2a message to all acceptors containing this value and ballot 0. Each acceptor receives this message and replies with a phase 2b message for ballot 0. When the leader receives these phase 2b messages from a majority of acceptors, it sends a phase 3 message announcing that the value is chosen.

The initial leader may fail, causing ballot 0 not to choose a value. In that case, some algorithm is executed to select a new leader - for example, the algorithm of [**?**]. Selecting a unique leader is equivalent to solving the consensus problem. However, Paxos maintains consistency, never allowing tow different values to be chosen, even if multiple processes think they are the leader. A unique nonfaulty leader is needed only to ensure liveness.

A process that believes itself to be a newly-elected leader initiates a ballot, which proceeds in the following phases

1. **Phase 1**

   (a) **Phase 1a**: The leader chooses a ballot number bal for which it is the leader and that it believes to be larger than any ballot number for which phase 1 has been performed. The leader sends a phase 1a message for ballot number bal to every acceptor

(b) **Phase 1b**: When an acceptor receives the phase 1a message for ballot number bal, if it has not already performed any action for a ballot numbered bal or higher, it responds with a phase 1b message containing its current state, which consists of

- The largest ballot number for which it received a phase 1a message
- The

2. **Phase 2**:

   - **Free**:
   - **Forced**:

   test

   - **Phase 2b**:

3. **Phase 3**:

3.2 The Paxos Commit Algorithm

In the Two-Phase Commit protocol, the TM decides whether to abort or commit, records that decision in stable storage, and informs the RMs of its decision. We could make that fault-tolerant by simply using a consensus algorithm to choose the **committed** / **aborted** decision, letting the TM be the client that proposes the consensus value. Having the RMs tell the leader that they have prepared requires at least one message delay. How our Paxos Commit algorithm eliminates that message delay is described below.

Paxos Commit uses a separate instance of the Paxos consensus algorithm to obtain agreement on the decision each RM makes of whether to prepare or abort - a decision we represent by the values **Prepared** and **Aborted**. So, there is one instance of the consensus algorithm for each RM. The transaction is committed iff each RM's instance chooses **Prepared**; otherwise the transaction is aborted.