Two Phase Commit Protocol

Murat Demirbas

[2019-08-26 Mon]
2 phase commit

A transaction is performed over resource managers (RMs)

The transaction manager (TM) finalizes the transaction

- For the transaction to be committed, each participating RM must be prepared to commit it

- Otherwise, the transaction must be aborted
**Definitions**

```
--fair algorithm 2PC{
  variable rmState = [rm ∈ RM → "working"],
    tmState = "init";

define {
  canCommit ≡ ∀ rm ∈ RM : rmState[rm] ∈ {"prepared"}
  canAbort ≡ ∃ rm ∈ RM : rmState[rm] ∈ {"aborted"}
}
```
TM modeling

```plaintext
fair process ( TManager = 0 ) {
    TM: either
    { await canCommit ;
    tmState := "commit" ;
    F1: if ( TMMAYFAIL ) tmState := "unavailable" ; } 
    or
    { await canAbort ;
    tmState := "abort" ;
    F2: if ( TMMAYFAIL ) tmState := "unavailable" ; } 
}
```
TM checks if it canCommit or canAbort and updates tmState accordingly.

TM can also fail making tmState "unavailable"

To keep things simple yet interesting, TM fails only after it makes a decision. These two updates are nonatomic:

- tmState is available for RMs to read for a duration
- labels at fail actions provide nonatomicity
RM modeling

working → prepared → committed

aborted
fair process ( $RManager \in RM$ ) {
    $RS$: while ( $rmState[self] \notin \{ "committed", "aborted" \} ) {
        either {
            await $rmState[self] = "working" ;
            with ( $x \in \{ "prepared", "aborted" \} ) $rmState[self] := x ;
        }
        or {
            await $rmState[self] = "prepared" \land tmState = "commit" ;
            $RC$: $rmState[self] := "committed" ;
        }
        or {
            await $rmState[self] = "prepared" \land tmState = "abort" ;
            $RA$: $rmState[self] := "aborted" ;
        }
    }
}
Invariants

175 \[ \text{Consistency} \triangleq \]
A state predicate asserting that two RMs have not arrived at conflicting decisions.

\[ \forall rm1, rm2 \in RM : \neg \land \land \text{rmState}[rm1] = \text{"aborted"} \]
\[ \land \text{rmState}[rm2] = \text{"committed"} \]

183 \[ \text{Completed} \triangleq \square (\forall rm \in RM : \text{rmState}[rm] \in \{\text{"committed"}, \text{"aborted"}\}) \]

**Consistency** checks that there are no 2 RMs such that one says "committed" and other says "aborted"
Model checking

If TM does not fail, the 2-phase commit algorithm is correct.

When TM fails, termination can be violated.

We add a Backup TM, to take over, and achieve termination.
fair process ( BTManager = 10 ) {
  BTM : either
  { await canCommit \land tmState = "unavailable" ;
  BTC : tmState := "commit" ; }
  or
  { await canAbort \land tmState = "unavailable" ;
  BTA : tmState := "abort" ; }
}
Strengthening canCommit

```plaintext
8  --fair algorithm 2PC{ 
9     variable rmState = [rm ∈ RM → "working"],
10           tmState = "init" ;
12    define { 
13       canCommit  ≜  ∀ rm ∈ RM : rmState[rm] ∈ {"prepared"} 
14           ∨  ∃ rmc ∈ RM : rmState[rmc] ∈ {"committed"}  for BTM
15       canAbort  ≜  ∃ rm ∈ RM : rmState[rm] ∈ {"aborted"} 
16    }
```
BTM modeling

BTM takes over when TM is unavailable and uses the same logic to make decisions.

For simplicity we assume the BTM does not fail.
What if RMs could also fail?

Diagram:
- Working
- Prepared
- Committed
- Aborted
- Unavailable

Arrows indicate transitions between states.
What if RMs could also fail?

```plaintext
fair process ( RManager ∈ RM )
variables pre = "" ; {
RS: while ( rmState[self] ∉ { "committed", "aborted" } ) {
either {
    await rmState[self] = "working" ;
    with ( x ∈ { "prepared", "aborted" } ) rmState[self] := x ;
}
or {
    await rmState[self] = "prepared" ∧ tmState = "commit" ;
    RC: rmState[self] := "committed" ;
}
or {
    await rmState[self] = "prepared" ∧ tmState = "abort" ;
    RA: rmState[self] := "aborted" ;
}
or {
    await RMMAYFAIL ∧ pre ≠ rmState[self] ;
    pre := rmState[self] ;
    rmState[self] := "unavailable" ;
    RR: rmState[self] := pre ;
}
}
```
Strengthening canAbort

\[
\begin{align*}
\text{define } & \{ \\
\text{canCommit} & \triangleq \forall \text{rmc} \in \text{RM} : \text{rmState}[\text{rmc}] \in \{ "prepared" \} \\
& \quad \lor \exists \text{rm} \in \text{RM} : \text{rmState}[\text{rm}] \in \{ "committed" \} \quad \text{for BTM} \\
\text{canAbort} & \triangleq \exists \text{rm} \in \text{RM} : \text{rmState}[\text{rm}] \in \{ "aborted", "unavailable" \} \\
& \quad \land \neg \exists \text{rmc} \in \text{RM} : \text{rmState}[\text{rmc}] = "committed" \quad \text{for BTM}
\end{align*}
\]
Inconsistency problem!
What went on?

- RM1 sees commit from TM
- TM becomes unavailable
- RM2 becomes unavailable
- BTM takes over for TM
- BTM decides on abort seeing <prepared, unavailable, prepared> from RMs. (It may also be that RM1 may also look unavailable due to unreachability)
- RM1 acts on commit from TM
- RM3 sees abort from BTM
- RM3 acts on abort from BTM
- Consistency is violated
Inconsistency problem!

If BTM decides, it may decide incorrectly, violating consistency

If BTM waits, it may be waiting forever on a crashed node, and violating termination (i.e., progress)

Timeouts may be incorrect, due to inopportune timing
In an asynchronous system, it is impossible to solve consensus (both safety and progress) in the presence of crash faults.
What about 3PC?
3PC problems

BTM may act as if TM is down

RM$s go with whatever TM or BTM says (2 leaders)

This asymmetry of information is the root of all evil in distributed systems
"Several 3-phase protocols have been proposed, and a few have been implemented. They have usually attempted to "fix" the 2-Phase Commit protocol by choosing another TM if the first TM fails. However, we know of none that provides a complete algorithm proven to satisfy a clearly stated correctness condition. For example, the discussion of non-blocking commit in the classic text of Bernstein, Hadzilacos, and Goodman fails to explain what a process should do if it receives messages from two different processes, both claiming to be the current TM. Guaranteeing that this situation cannot arise is a problem that is as difficult as implementing a transaction commit protocol."
Paxos! In the realm of possibility, Paxos makes progress. Even when asynchronous execution, faults, and eventually makes progress when consensus gets in the realm of possibility. You can emulate TM with a Paxos cluster of 3 nodes and solve the inconsistent TM/BTM problem (Google Spanner approach).