2 Phase Commit Modeling

Murat Demirbas

Game plan

2 phase commit protocol

Modeling in PlusCal & model checking

Adding recovery & model checking

Beyond 2 phase commit

2 phase commit modeling with message passing

2 phase commit protocol

2 phase commit (2PC)

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

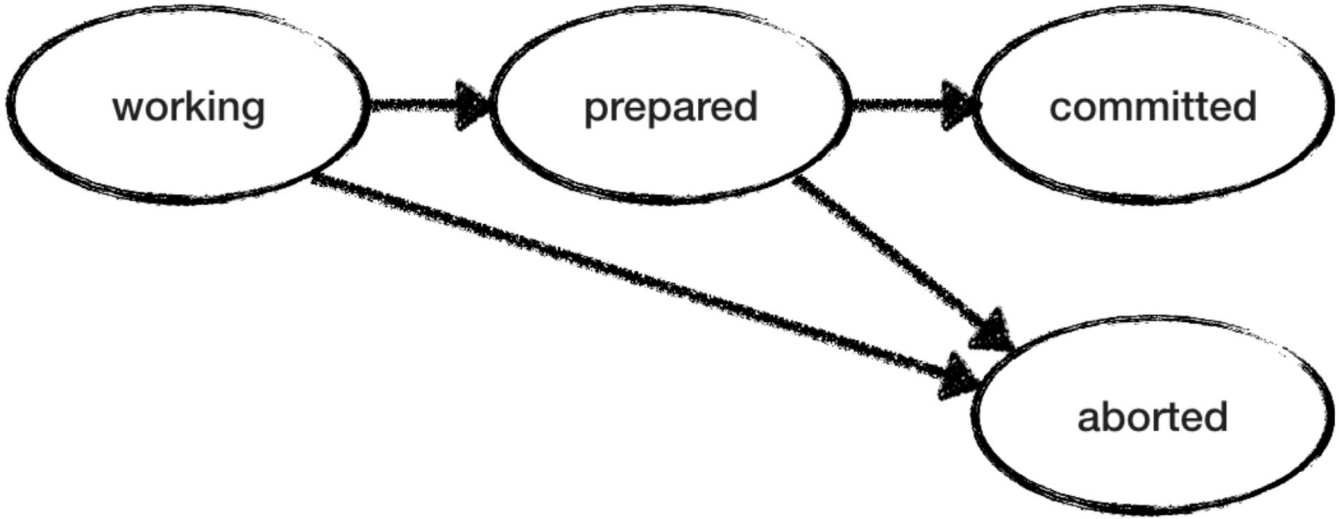
TM modeling

TM checks if it canCommit or canAbort and updates tmState accordingly

TM can also fail making tmState "unknown"

- To keep things simple yet interesting, TM fails only after it makes a decision
- These two updates are nonatomic, so some RMs can read state before TM fails

RM modeling



2 phase commit in Pluscal

```
1  ---------------------------- MODULE 2PC ----------------------------
2  EXTENDS Integers, FiniteSets, TLC
3  CONSTANT
4      TMMAYFAIL   \* whether TM may fail
5
6  (* --algorithm 2PC {
7    variable
8      rmState = [rm \in RM |-> "working"],
9      tmState = "unknown";
10
11   define {
12     RM == 1..3  \* set of resource managers
13
14     canCommit == \A rm \in RM : rmState[rm] \in {"prepared"}
15     canAbort  == \E rm \in RM : rmState[rm] = "aborted"
16
17     \* Type invariant
18     TypeOK == /\ rmState \in [RM -> {"working", "prepared", "committed", "aborted"}]
19               /\ tmState \in {"commit","abort","unknown"}
20
21     \* Invariant: no two RMs arrive at conflicting decisions
22     Consistent ==
23         ~ \E rm1, rm2 \in RM : /\ rmState[rm1] = "aborted"
24                                /\ rmState[rm2] = "committed"
25
26     \* Bait invariants to test TLC output
27     NotCommitted == \A rm \in RM : rmState[rm] # "committed"
28   }
```

```
30      macro Prepare(p) {
31        await rmState[p] = "working";
32        rmState[p] := "prepared" ;
33      }
34
35      macro Decide(p) {
36        either {
37            await tmState="commit";
38            rmState[p] := "committed";}
39        or {
40            await rmState[p]="working" \/ tmState="abort";
41            rmState[p] := "aborted";}
42      }
43
44
45      fair process (RManager \in RM) {
46    RS:   while (rmState[self] \in {"working", "prepared"}) {
47            either
48                Prepare(self)
49            or
50                Decide(self)
51            }
52      }
53
```

```
54
55      fair process (TManager=0) {
56    TS: either {
57            await canCommit;
58    TC:     tmState := "commit";}
59        or {
60            await canAbort;
61    TA:     tmState := "abort";};
62
63    TF: if (TMMAYFAIL) tmState := "unknown"; \* tm state becomes inaccessible
64        }
65
66    } \* end algorithm
67    *)
68
```

Model checking

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

If TM does not fail, the 2PC algorithm is correct

When TM fails, termination can be violated

We can add a Backup TM, to take over, and achieve termination

BTM modeling

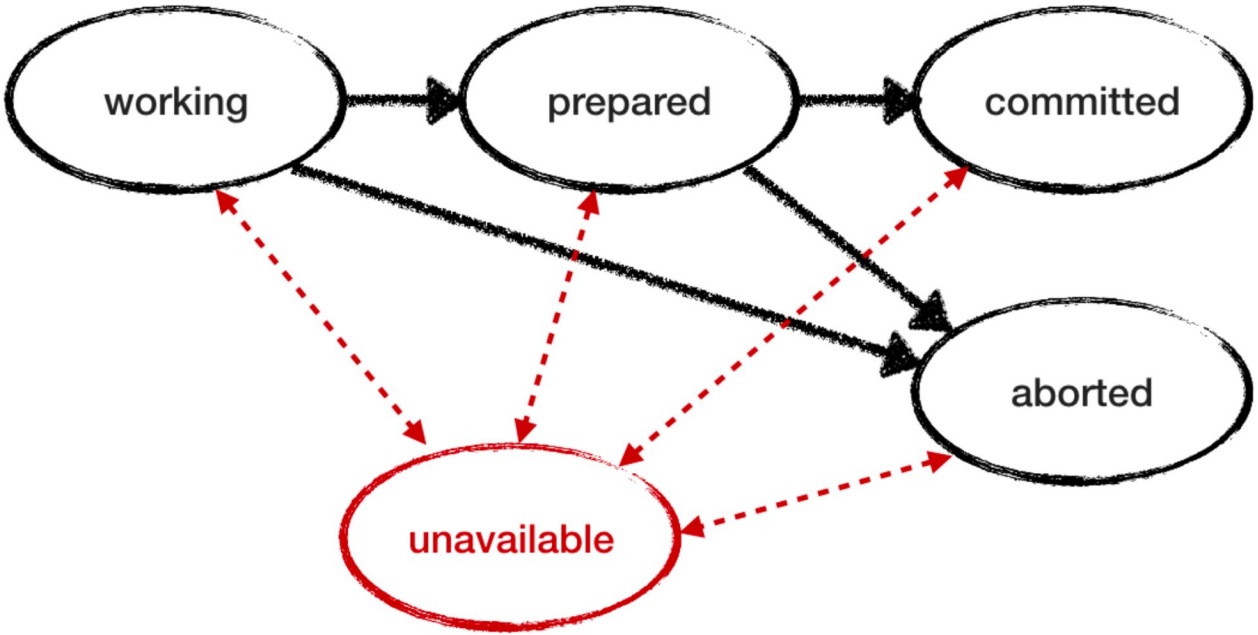
BTM takes over when TM is unknown and uses the same logic to make decisions

For simplicity we assume the BTM does not fail

```
63      fair process (TManager=0) {
64    TS:  either {
65             await canCommit;
66    TC:      tmState := "commit";}
67           or {
68             await canAbort;
69    TA:      tmState := "abort";};
70
71    TF:      if (TMMAYFAIL) tmState := "unknown";
72      }
73
74
75      fair process (BTManager=10) {
76    BTS: either {
77             await canCommit /\ tmState = "unknown";
78    BTC:     tmState := "commit";}
79           or {
80             await canAbort /\ tmState = "unknown";
81    BTA:     tmState := "abort";}
82      }
83    }
```

Adding RM recovery & model checking

What if RMs could also fail



```
39      fair process (RManager \in RM)
40      variables pre=""; {
41    RS:  while (/\ rmState[self] \notin {"committed", "aborted"}
42                 /\ pre \notin {"committed", "aborted"}) {
43             either {
44                await rmState[self] = "working";
45                rmState[self] := "prepared"; }
46             or {
47                await rmState[self]="prepared" /\ tmState="commit";
48    RC:         rmState[self] := "committed";}
49             or {
50                await rmState[self]="working"
51                 \/  (rmState[self]="prepared" /\ tmState="abort");
52    RA:         rmState[self] := "aborted";}
53             or {
54                await RMMAYFAIL  /\  pre # rmState[self];
55    RF:         pre := rmState[self];     \* Log state pre-failure
56                rmState[self] := "unknown";  \* Fail
57    RR:         rmState[self]:= pre;      \* Recover state pre-failure
58                }
59          }
60      }
```

Model checking reveals an inconsistency problem!

We have two decision makers TM and BTM looking at the state of the RMs at different times, and making different decisions

This asymmetry of information is the root of all evil in distributed systems

1. RM1 sees commit from TM

2. TM becomes unavailable

3. RM2 becomes unavailable

4. BTM takes over for TM

5. BTM decides on abort seeing <prepared, unavailable, prepared> from RMs.
(It may also be that RM1 may also look unavailable due to unreachability)

6. RM1 acts on commit from TM

7. RM3 sees abort from BTM

8. 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

FLP (1985) impossibility: In an asynchronous system, it is impossible to solve consensus (both safety and progress) in the presence of crash faults

Beyond 2PC

Lamport & Gray on 3PC

BTM may act as if TM is down; RMs go with whatever TM or BTM says (2 leaders)

Several 3-phase protocols have been proposed, and a few have been implemented. They have usually attempted to "fix" the 2PC 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

Paxos is always safe even in presence of inaccurate failure detectors, asynchronous execution, faults, and eventually makes progress when consensus gets in the realm of possibility

Paxos makes progress when the system is outside the realm of consensus impossibility

You can emulate TM with a Paxos cluster of 3 nodes and solve the inconsistent TM/BTM problem

2PC with message passing

What would message passing version look like

Message passing version of 2PC could be modeled as an educational exercise

We use "network" set as a shared whiteboard to add msgs and read from

- Set filtering did the heavy-lifting for receiving a message of interest.
- We can also converted global rmState/tmState into process-local vars.
- This resulted in having to move the Invariants at the end of TLA translation.

State-space increased by 5 folds, model got uglier, with not much insight/benefit for additional behaviors to check

```
(* --algorithm 2PCMP {
  variable
    network = {}; \* all communication happens here via shared whiteboard model: add/read

  define {
    RM == 1..3  \* set of resource managers

    \* useful definitions to filter/query network
    PreparedMsgs(i) == {m \in network: m.t="prepared"   /\ m.s=i}
    AbortedMsgs(i)  == {m \in network: m.t="aborted"    /\ m.s=i}
    TCommitMsg(i)   == {m \in network: m.t="t-commit"   /\ m.d=i}
    TAbortMsg(i)    == {m \in network: m.t="t-abort"    /\ m.d=i}

    canCommit == \A rm \in RM : PreparedMsgs(rm) # {}
    canAbort  == \E rm \in RM : AbortedMsgs(rm) # {}

    \* example of message constructors
    msg_prepared(i) == [ t |-> "prepared", s |-> i ]
    msg_aborted(i)  == [ t |-> "aborted",  s |-> i ]

    \* All possible protocol messages
    Messages == [t:{"prepared"}, s:RM]
        \union  [t:{"aborted"},  s:RM]
        \union  [t:{"t-commit"}, d:RM]
        \union  [t:{"t-abort"},  d:RM]
```

```
44
45
46      \* macro for sending a message to the network
47      macro Send(m)  { network := network \union {m}; }
48
49
50      macro Prepare(p) {
51        await rmState = "working";
52        rmState := "prepared" ;
53        Send(msg_prepared(p));
54      }
55
56      macro Decide(p) {
57        either {
58            \* await tmState="commit";
59            await TCommitMsg(p) # {};
60            rmState := "committed";}
61        or {
62            \* await rmState[p]="working" \/ tmState="abort";
63            await rmState="working" \/ TAbortMsg(p) #{};
64            rmState := "aborted";
65            Send(msg_aborted(p));}
66      }
```

Discussion

Discussion

What did you find surprising about 2 PC modeling?

What are benefits of Pluscal over TLA+ for modeling with processes?

What will you choose as an entry project to model in TLA+/Pluscal?

Resources

https://muratbuffalo.blogspot.com/2018/12/2-phase-commit-and-beyond.html