2 Phase Commit

Lakshmi Chaitanya Chapala
lakshmic@buffalo.edu
UBNo:50290974

Distributed Systems

1 Introduction

2PC protocol is a transaction management protocol for distributed set up which aims to achieve atomic transactions in distributed set up similar to atomic transactions in single database instances.

1.1 Termination

It is a temporal property which is default for processes to check to be done state.

1.2 Completed

Completed is a temporal property which will make sure that protocol doesn't hang on forever i.e., eventually each RM reaches to a COMMITTED or ABORTED state. If any of the RM's did not reach these terminal states , this property is said to be violated.

1.3 Consistency

Completed is an Invariant i.e., it is initially given stays stable for the rest of the execution. In this task the consistency property makes sure that NO 2RM's have not arrived at a conflicting decision. Which means there will be no 2RM's such that one says "Committed" and the other "Aborted".

In this project we will see that whether these properties will hold for different scenario's. And for all the cases we consider RM = 1..4

2 Task-1: No RM Failure, No BTM

2.1 Scenario-1: TMMAYFAIL:=FALSE

This is the case where we assume that the transaction manager never fails. In this case all the properties hold and no errors were identified. Means the program terminates as all the processes in the RM reach the terminal states either "committed" or "aborted" and every action is consistent and the TM terminates without any failure.

Note: In windows I could'nt find error trace when the program terminates successfully so was not able to display states.

Sub-actions of next-state | (at 00:00:03)

Module	Action	Location	States Found	Distinct States
proj1	Terminating	line 127, col 1 to line 127, col 11	2	0
proj1	Init	line 65, col 1 to line 65, col 4	1	1
proj1	F1	line 108, col 1 to line 108, col 2	256	1

Figure 1: No Error

2.2 Scenario-2: TMMAYFAIL:=TRUE

This is the condition where the TM is allowed to fail which in real-time it is possible for TM to fail. And as we see in the fig2, the state has all RM's turned from "working" to "prepared" state ready to receive instructions from TM which sends "COMMIT" message.

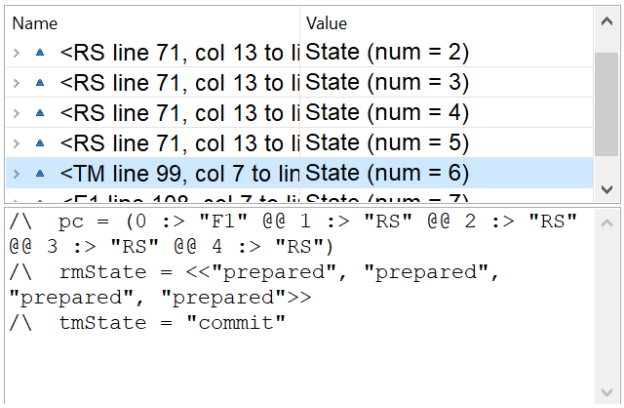

Figure 2: State1.

In this last state of Fig3, before all the RM's receive the "COMMIT" from the TM , TM becomes unavailable leaving all the RM's in non-terminal state "prepared" violating the "COMPLETED" property.

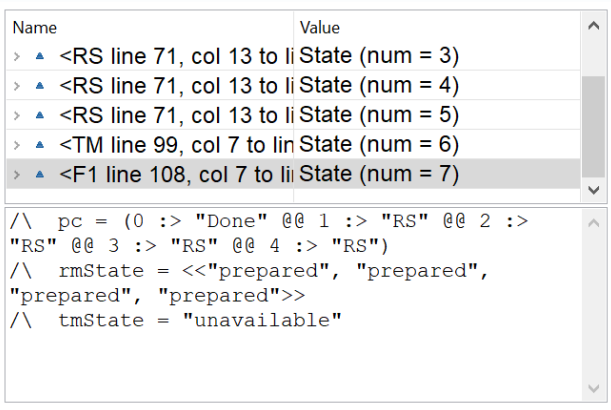

Figure 3: State2.

Deadlock has reached the state.An algorithm is deadlocked if it has not terminated, but it can take no further step. And as the TM became unavailable ,algorithm doesn't know what to do next

and also RM's are in non-terminal state.

Figure 4: Kind of Error.

3 Task-2:Update the code to model the cases of RM failures

In real-time RM's also might prone to failures. And in this task we will modify the code such that the failure of the RM's is also handled.

- We will use RMMAYFAIL variable to indicate whether RM's are reliable or not.
- We allow the crashed RM to recover back.
- we create a variable which will store the states of the current RM's whenever they become unavailable and restore them back when available form the stored value.
- And we also should make sure that when TM is checking to canAbort condition, it should also consider unavailable state.

```
define {
   canCommit ==    \A rm \in RM  : rmState[rm] \in {"prepared"}
   canAbort ==     \E rm \in RM : rmState[rm] \in {"aborted","unavaible"}
}


  fair process (RManager \in RM)
  variables pre = "" ; {
RS:  while (pre \notin {"committed", "aborted"}) {
        either {
           await rmState[self] = "working";
           rmState[self] := "prepared" ; }
        or {
           await rmState[self]="prepared" /\ tmState="commit";
RC:        rmState[self] := "committed";}
        or {
           await rmState[self]="working"
            \/  (rmState[self]="prepared" /\ tmState="abort");
RA:        rmState[self] := "aborted";}
        or {
            await (RMMAYFAIL) /\ pre /= rmState[self];
            pre := rmState[self];
            rmState[self] := "unavailable";
 RU:        rmState[self] := pre ;
            }
      }
  }
```

Figure 5: Code of RMMAYFAIL.

3.1 Scenario 1: TMMAYFAIL:=FALSE, RMMAYFAIL:=TRUE

In this scenario we are ensuring that the TM will never fail and the RM may fail. As our code is making sure all the properties are achieved we see no error.(fig.6)

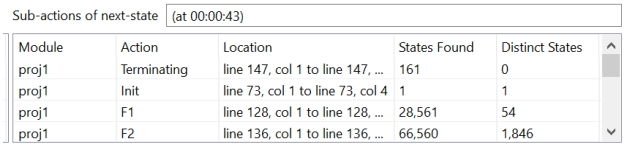

Figure 6: No Error for Scenario-1.

3.2 Scenario 2: TMMAYFAIL:=TRUE, RMMAYFAIL:=TRUE

In this scenario both RM and TM are prone to failure. In this case our TM does not know how to still deal with it's failure.

As you see in the fig.7 in State No-13 , RM-3 and 4 are still unavailable and are learning about their state from the pre while the Tm went "unavailable".

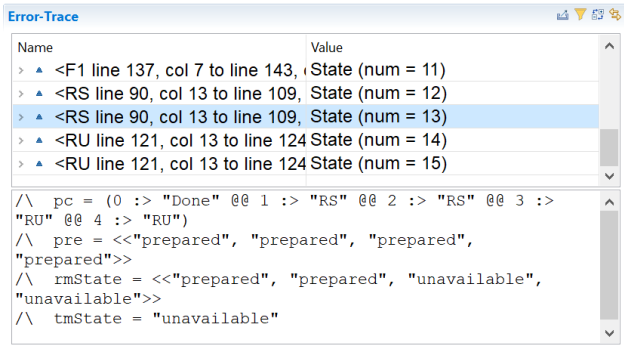

Figure 7: Kind of Error.

And as in Fig.8 although in the next state RM-3 learned about it's state and changed into prepared, while the tm is still unavailable.

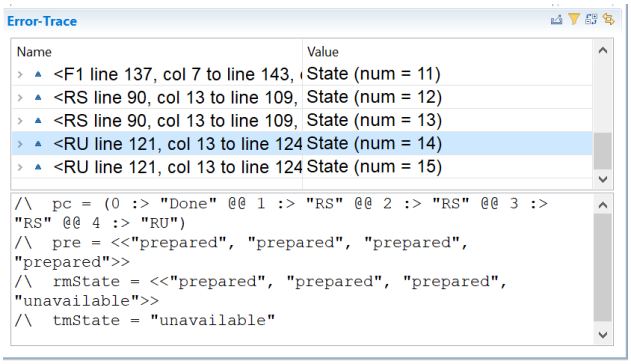

Figure 8: Kind of Error.

In the final state when the deadlock is confirmed TM is still in unavailable state where the RM-4 reverted back to it's prepared state after failure.So as we see although there is so much happening with states of the RM , the TM remained state unchanged reaching the fixed point. And we will fix this problem in the following task.

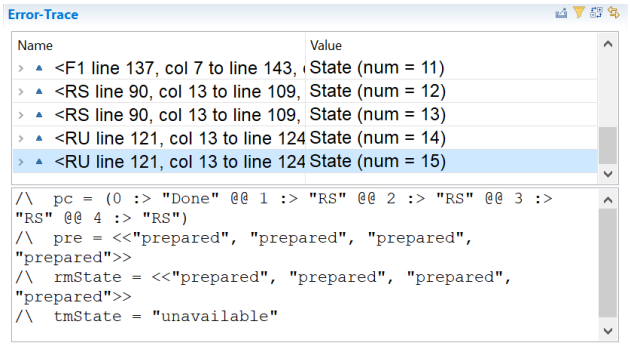

Figure 9: Kind of Error.

4 Task-3: Including BTM

As we have seen in the Task-2 scenario-2, although our algorithm can account for RMFAILURE , it still can't handle the case of TM failure. The TM in the above case did not respond once it crashes. So we introduce one more component called Back-up Transaction Manager(BTM), which will make sure that it takes control of the actions that are performed by the TM once it fails. Unlike RM, we need one more Manager as TM itself is something that manages the tasks of both TM and RM's. And here in this case we assume that the BTM never fails.

- The code part of BTM is almost similar to that of TM.
- The only thing that we need to do is to send commit and abort messages when the TM becomes unavailable, so we check for that condition in the BTM modelling.
- And also we will modify canCommit and canAbort so that the BTM takes care of any committed and aborted RM's that are left when the TM failed.

```
  fair process (BTManager = 10){
BTM: either
    { await canCommit /\ tmState = "unavailable";
BTC:  tmState := "commit";}
  or
   {await canAbort /\ tmState = "unavailable";
BTA: tmState := "abort";}
  }
```

Figure 10: BTM code.

4.1 TMMAYFAIL:=TRUE, RMMAYFAIL:=FALSE

In this scenario we will check how or algorithm works when TM fails and RM does'nt fail. In state-5 as we see in fig11, the tm will send unavailable when it's gets crashed while it was sending abort messages to the RM's. Only 1st RM received the message where the rest did not know yet.

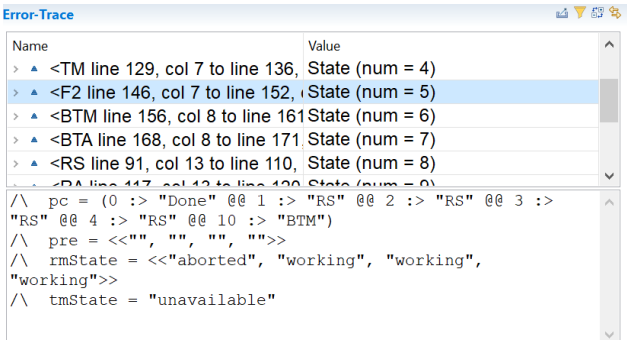

Figure 11: State5, When TM fails.

And immediately in the next state-6 the BTM takes over.

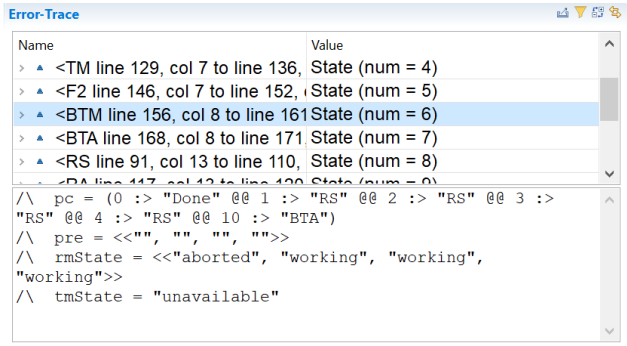

Figure 12: State 6, BTM takes over TM.

And in the following states BTM wills send the abort message to the rest of the RM's.And yet the program reaches deadlock as the original tm doesn't become available.

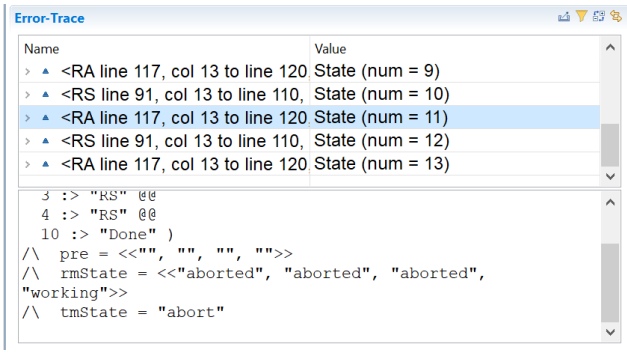

Figure 13: State -11 .

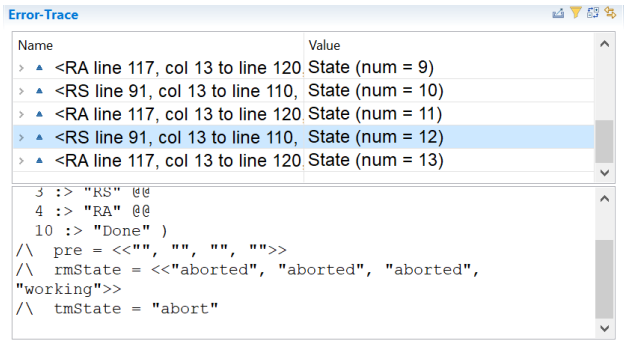

Figure 14: State -12.

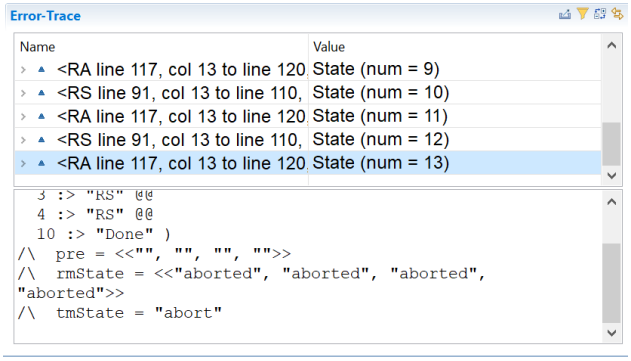

Figure 15: State-13.

4.2 TMMAYFAIL:=TRUE, RMMAYFAIL:=TRUE

And the last case that we are checking is the case where both RM and TM are prone to failure and surprisingly our algorithm seem to handle it well and terminated.

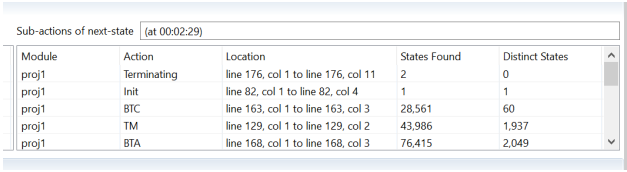

Figure 16: Scenario-2.

5 Observation

If we observe it although we added BTM our system still has problems and reaching the deadlock.That is considered as FLP(Fischer, Lynch and Patterson)impossibility, which states that in an asynchronous system, it is impossible to solve consensus (satisfy both safety and progress conditions) in the presence of crash faults.

And the reasons here might be because,

1.BTM may decide incorrectly(consistency violated) or

2.If BTM waits(termination violation)