MaBE Agents and n-Phase-Commit

Roland Schaffer

June 24, 2004

Abstract

This paper describes the differences between n-Phase-Commit algorithms and it shows - and proofs - why only a 3-Phase-Commit transaction handling can satisfy business process requirements as they are planned in MaBE.

1 Overview

It is depicted why distributed transactions have to be atomic and what conditions have to be met to reach this goal. This paper also describes why distributed committing is needed and which algorithms exist. It also shows where the differences between n-Phase-Commit algorithms are and it shows why only a 3-Phase-Commit transaction handling can truly satisfy business process requirements.

2 Distributed Commit

A transaction has to be an atomic action. Hence a distributed transaction has to be atomic too. There are a few protocols available that provide atomic properties, those are so called *Atomic Commit Protocol(s)* (ACPs). A transaction is either commited (state: COMMIT) or aborted (state: ABORT or CANCEL). The execution of a distributed transaction is a very complex action which can fail for the very simplest reasons, e.g. a failure of a modem-line or locked dataset or due to other contrarities, there are lots of them.

The now discussed n-Phase-Commit algorithms are controlling (in the most abstract point of view arbitrary) changings in heterogeneous systems. From the point of view of MaBE the system is not heterogeneous (the system used is JADE).

The following n-Phase-Commit methods are controlling the formerly mentioned changings (in heterogeneous systems). This commit takes place in the very simplest case on database level and in more complex cases on business processlevel.

To be able to execute distributed transactions a so called *coordinator* [CDK96, p 411] is required. This coordinator can be a dedicated server or a service layer. In the case of MaBE there is no service layer, thus a Coordinator-Agent or an agent which is involved in the transaction takes the role of the coordinator. Bernstein came up with a set of rules for an atomic commit.

Bernsteins conditions[BGR87, p 224] which an Atomic-Commit-Protocol must meet are:

AC1 all processes which makes a decision, make the same decision

AC2 a process can not change a desicion which it has already made

AC3 a COMMIT decision can only be made if all processes did vote YES

AC4 the decision must be COMMIT if there were no errors and all processes voted YES

AC5 based on the assumption that there are only errors which the algorithm has to tolerate according to its design, then the following rule applies: at any moment of its execution, when all errors which occured are corrected and no new errors occured long enough, then all involved processes will make a decision eventually.

3 1-Phase-Commit

As in Bernsteins laws can be seen, the atomicity of a distributed transaction requires the transaction to fail on all involved systems or to take place on all involved systems. A simple way to achieve this goal is to let the coordinator send a COMMIT or ABORT command to the involved systems until it receives an execution confirmation of all of them[CDK96, p 414]. This means, that 1-Phase-Commit is not useful because after a transaction took place, there is no way to assure that all systems did the same action. The coordinator does receive a reply from the systems but this reply can be a commit or an abort. After receiving the reply, the 1-Phase-Commit is over and therefor this algorithm is of no use in a business environment.

4 2-Phase-Commit

The algorithm, simplified [CCPS00, p 357 et sqq.]:

Phase 1: The transaction service (aka coordinator) receives a *COMMIT* request from a resource and sends a *PREPARE* to all involved resources. The involved resources executes a voting and reply with *COMMIT* or *VETO*.

Phase 2: The transaction service verifies the replys. When there is no single *VETO*, denn it executes a *COMMIT* else a *ROLLBACK*.

Unfortunately this protocol does provides lots of possibilities for failures. In the worst case there are various systems with equal many communication channels involved. Every system communicates - in case of success - four times (or five times, depends on implementation) with the coordinator:

1. transaction data
2. coordinator to all systems: *Will the transaction be successful?*
3. individual system to coordinator: *YES!*
4. coordinator to all systems: *Execute transaction!*
5. individual system to coordinator: *Transaction done.* (optional, not required by algorithmus).

The last message (5) is not counted as an expense ([CDK96, p 417]) due to the fact that 2-Phase-Commit can be done without this message.

Now there can be any of this messages erronious and have to be sent arbitrary often by the hardware, thus it is - according to Colouris - possible that a 2-Phase-Commit can be done successfully but it is not possible to provide a deadline for the transaction. That means the involved systems can become deadlocked.

The big advantage of the 2-Phase-Commit protocol is the guaranty of global atomicity. The very drawback is that it is a blocking protocol and thus able to deadlock its resources. To check Bersteins laws it harms the 5th condition($\rightarrow$ Chap. 2 / p 2): as soon as the coordinator or its communication channels crashes all involved systems have to wait for its - the coordinator or the network - restart.

In figure 1 / p 4 is q the start state, p the prepare state (transition to phase 2), c the commit and end state (successful end state) and a the abort and end state (failure end state). The i names the appropriate system:

$$i \in \{2, 3, 4, \cdots\} : \text{resource}, i = 1 : \text{coordinator}$$

The start state is q_i, in this state the coordinator sends the *PREPARE* message and receives *YES* or *NO* replies. If there is a system crash during the transaction, lets say all agents including the coordinator agent crashes except agent A_3, then this agent can only be in state p_3. Now Agent A_3 has the following options:

$$A_3(p_3) \overset{ABORT}{\longrightarrow} A_3(a_3) \text{ or } A_3(p_3) \overset{COMMIT}{\longrightarrow} A_3(c_3)$$

But in this very moment it does not matter what the decision of agent A_3 was because there is a high probability that another system voted *YES* or *NO*. For this

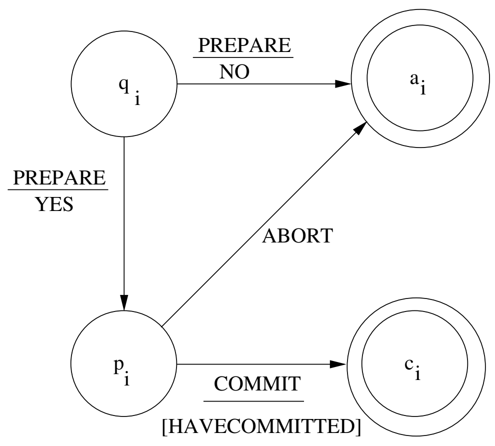

Figure 1: 2-Phase-Commit Statediagram

reason, agent A_3 has to wait for all other agents to come back online. Thus proofs the protocol blocking (see [Gad95, p 3] and [BGR87, p 227 et sqq.]).

To avoid blocking there could timeouts be applied. But this will work only at a first glance. After the timeout occurs, the timeout event would transit the transaction into a predefined state, say *ABORT*. This would work very nice for all messages except for the last one. As described in the previous paragraph, the agent has to wait. If it does not - due to a timeout - it would ruin the successful transaction because one glance after the timeout all crashed systems could have be back online and sent a *YES* and, worse, after the timeout the agent has no possibility to inform the coordinator of its *ABORT* which would harm Bernsteins 2nd condition (→ Chap. 2 / p 2): a process is not allowed to change its decision.

5 3-Phase-Commit

The 3-Phase-Commit protocol is not blocking. It has the following properties[Gad95, p 1]:

- not blocking in case of system crash
- all failed systems have to come to the same result (YES/NO) - afer their restart - as before the failure occured
- working systems shall agree upon the result of the transaction by considering their own result states
- an already made decision must be consistent with the final result of all working systems

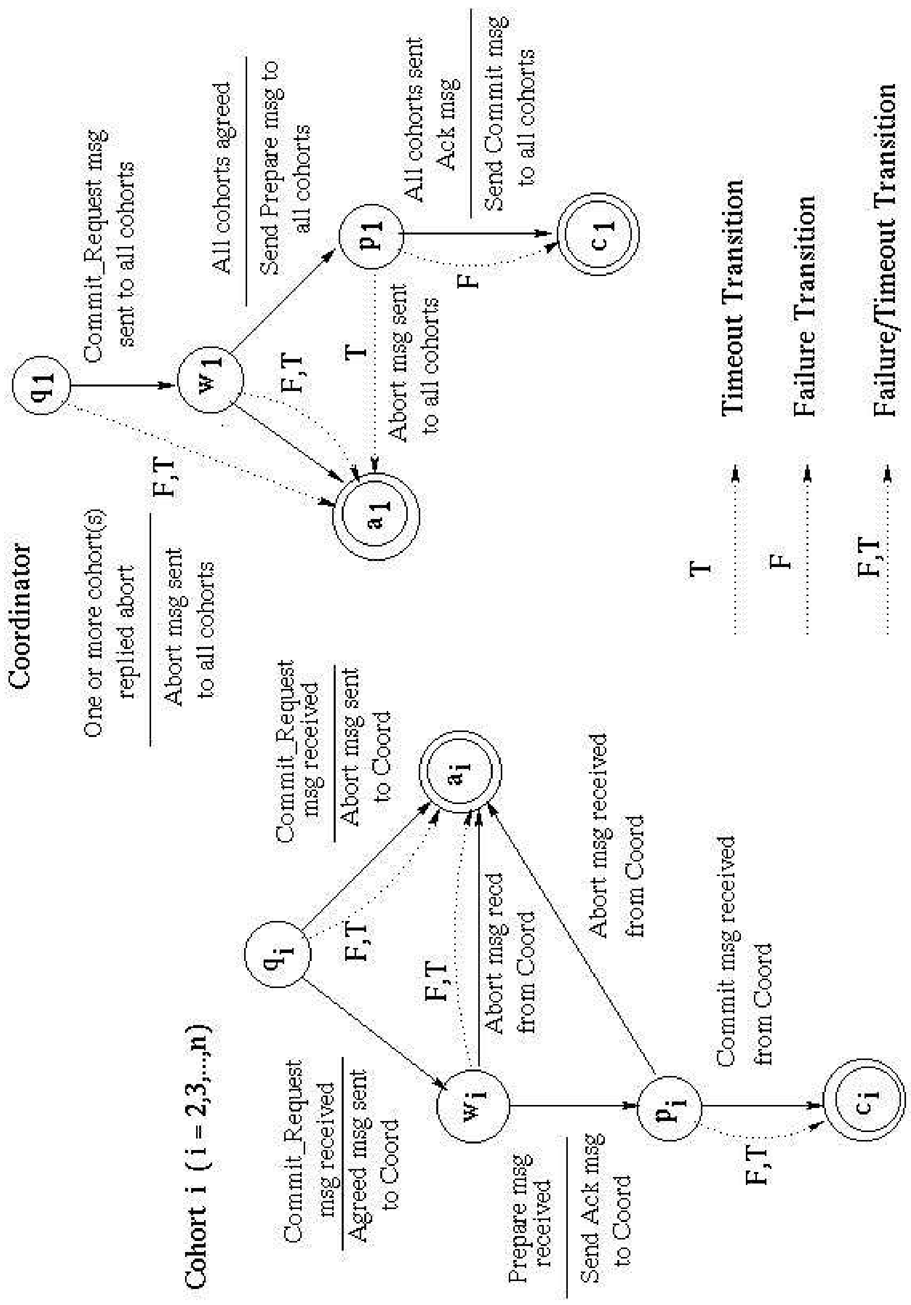
Coordinator
q1
w1
p1
a1
c1
Commit_Request msg sent to all cohorts
One or more cohort(s) replied abort / Abort msg sent to all cohorts
F,T
All cohorts agreed / Send Prepare msg to all cohorts
F,T
T
Abort msg sent to all cohorts
All cohorts sent Ack msg / Send Commit msg to all cohorts
F
Cohort i (i = 2,3,..,n)
qi
wi
pi
ai
ci
Commit_Request msg received / Abort msg sent to Coord
F,T
Commit_Request msg received / Agreed msg sent to Coord
F,T
Abort msg recd from Coord
Abort msg received from Coord
Prepare msg received / Send Ack msg to Coord
Commit msg received from Coord
F,T
T
Timeout Transition
F
Failure Transition
F,T
Failure/Timeout Transition

Figure 2: 3-Phase-Commit Statediagram [Gad95, p 2]

The 3PC algorithm according to Colouris [CDK96, p 446] (→ Figure 2 / p 5):

Phase 0: The coordinator receives a *COMMIT*-inquiry from any system. This initiating phase does NOT count as a step!

Phase 1: The coordinator sends a *PREPARE* (or a *VOTE-REQUEST*) statement to all involved systems. These systems perform a voting and answers with *COMMIT* or *VETO*.

Phase 2: The coordinator collects all votings and makes a decision. If there was a *VETO*, then its decision is *ABORT* and it informs all systems that voted *COMMIT* about the cancellation. If all systems voted *COMMIT* - that means no *VETO* arrived - then the coordinator sends a *PRE-COMMIT* to all systems, else it sends an *ABORT*. The systems reply with *ACKNOWLEDGE*.

Phase 3: The coordinator collects the *ACK*'s and after all arrived it sends *COMMIT* to all systems. If a system receives a *COMMIT* the commit gets executed.

Be aware of the fact, that if the coordinator crasches in state p_1 at least 1 participant is in state p_i too, so all YES-Voters can in state w_i can transpose to p_i and eventually commit. If no participant is in p_i state, nothing happens, because none did receive or invoke a commit.

If a participant cannot receive a message in the wait state w_i, it asks all remaining participants if one of them is in PRE-COMMIT state. If there is one, the participant will shift to PRE-COMMIT and finally commit, otherwise it will go to ABORT.

The very trick of 3PC is that in the first stage the default action is to ABORT opposite to the final stage where it is COMMIT and that it is not possible that two nodes are more than 1 stages apart. That means that at every moment no single node i is in Phase-1 if another node j is in Phase-3, in other words: the state distance between any two nodes can be at most 1.

$$\forall i, j \in \{1, \cdots, n\} : \text{statedistance}_j^i \leq 1$$

6 MaBE-Agents and 3-Phase-Commit

There must be a coordinator and one or more participants. I recommend the invoking agent as the coordinator. The advantage is, that the invoking agent wants to commit, so it will not opt against the transaction and therefore its VOTE precondition can safely be set to YES.

It is a more or less political question, whether a company decides to use 3PC or not. In most applications a 2PC transaction model is implemented because it is a little bit simpler and causes less traffic overhead.

In case of MaBE I really do recommend to use 3PC because agents are talking over large and long distances. Anything can happen to those communication lines and servers where the JADE platforms are running, thus a blocking protocol - which puts an agent into a deadlock - is not only not appropriate but even more dangerous for all involved business partners also known as agents. A lot of nasty things can happen to the involved programs, platforms and lines and the weak assertion *Blocking events will occur seldom* as it is used to argument the 2PC protocol is too weak.

In the worst case all involved agents get stuck for a pretty long time. Deadlock detection is - especially in this case - a pretty hard piece of work. So the recommendation can only be to use a 3-Phase-Commit protocol in the MaBE environment.

Based upon the 2PC pseudo code [TvS02, p 396 et sqq.] from Tanenbaum I wrote a 3PC pseudo code (but with only one thread, Tanenbaum used two of them):

6.1 Coordinator

```
/* COORDINATOR */
/* P1 */
write START_3PC to local log;
multicast PREPARE to all participants;

/* P2 */
while not all votes have been collected {
  wait for any incoming vote;
  if time out {
    write GLOBAL_ABORT to local log;
    multicast GLOBAL_ABORT to all participants;
    exit;
  }
  record vote;
}
if all participants sent VOTE_COMMIT and coordinator votes COMMIT {
  write PRE_COMMIT to local log;
  multicast PRE_COMMIT to all participants;
} else {
  write GLOBAL_ABORT to local log;
  multicast  GLOBAL_ABORT to all participants;
  exit;
}

/* P3 */
while not all acknowledges have been collected {
  wait for any incoming acknowledge;
  if one sends NOT_ACK { /* NOT_ACK will NEVER(!) occur due to Bernstein's law!!! */
    write GLOBAL_ABORT to local log;
    multicast GLOBAL_ABORT to all participants;
    exit;
  }else{
    if timeout {
       ignore it, participant performs commit after recovery;
    }
  }
  record acknowledge;
}

if all participants sent ACK {
  write GLOBAL_COMMIT to local log;
  multicast GLOBAL_COMMIT to all participants;
} else {
  the else (ABORT/exit) has already been done in the WHILE part
}
```

6.2 Participant

```
/* PARTICIPANT */

/* P1 */
write INIT to local log;
wait for VOTE_REQUEST from coordinator;
if time out {
  write VOTE_ABORT to local log;
  exit;
}

if participant votes COMMIT {
  write VOTE_COMMIT to local log;
  send VOTE_COMMIT to coordinator;
} else { /* participant votes ABORT */
  write VOTE_ABORT to local log;
  send VOTE_ABORT to coordinator;
  exit;
}

/* P2 */
wait vor DECISION from coordinator;
if timeout {
  multicast DECISION AND PHASE STATE REQUEST to other participants;
  wait until DECISION AND PHASE is received; /*remain blocked*/
  write DECISION to local log;
}
if DECISION == PRE_COMMIT and and least one other PHASE == PRE_COMMIT
  write PRE_COMMIT to local log;
else if DECISION == GLOBAL_ABORT or no other PHASE == PRE_COMMIT {
  write GLOBAL_ABORT to local log;
  exit;
}

/* P3 */
send ACK to coordinator;
wait until DECISION is received;
if timeout or DECISION == COMMIT {
  write GLOBAL_COMMIT to local log;
  perform the commit;
} else { /* this will never occur */
  write GLOBAL_ABORT to local log;
  exit;
}
```

Be careful at implementation time. If one of those two agents (coordinator or participant) crashes, they have to be recovered from the local log. Be really careful to recover to the proper state or the whole thing will be an undefined beast. Be sure to use a simple logfile and not a persistence tool. The whole thing depends on a fast, rigid and reliable construction. A persistance tool with a database on its backend will only add a lot of possible failure conditions.

References

[BGR87] Philip A. Bernstein, Nathan Goodman, and Vassos Radzilacos. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley, ISBN: 0201107155, 6 1987.

[CCPS00] Andrea Conzett, Ryan Cox, Joaquin Picon, and Gianni Scenini. *IBM WebSphere Application Server Enterprise Edition Component Broker 3.0: First Steps, SG24-2033-03*. IBM Redbooks. IBM, Boulder, CO, 8 2000. ISBN 0738419184.

[CDK96] George Couloris, Jean Dollimore, and Tim Kindberg. *Distributed Systems*. Addison Wesley, England, 2nd edition, 1996.

[Gad95] Srinivas R. Gaddam. Three-phase commit protocol. WWW, 4 1995. URL: http:// ei.cs.vt.edu / cs5204 / fall99 / distributedDBMS / sreenu / 3pc.html.

[TvS02] Andrew S. Tanenbaum and Maarten van Steen. *Distributed Systems*. Prentice Hall, New Jersey, 2002.