A Single-Phase Non-Blocking Atomic Commitment Protocol[1]

Maha Abdallah, Philippe Pucheral

University of Versailles, PRiSM Laboratory
45, avenue des Etats-Unis
78035 Versailles - France
{Maha.Abdallah, Philippe.Pucheral}@prism.uvsq.fr

Abstract. Transactional standards have been promoted by OMG and X/Open to allow heterogeneous resources to participate in an Atomic Commitment Protocol (ACP), namely the two-phase commit protocol (2PC). Although widely accepted, 2PC is a blocking protocol and it introduces a substantial time delay (two phases to commit). Several optimized protocols and non-blocking protocols have been proposed. Optimized protocols generally violate site autonomy while non-blocking protocols are inherently more costly in time and increase communication overhead. This paper proposes a new ACP that provides the non-blocking property while (1) having a lower latency than other ACPs (one phase to commit), and (2) preserving site autonomy, which makes it compatible with existing DBMSs. This protocol relies on the assumption that all participants are ruled by a rigorous concurrency control protocol. Performance analysis shows that our protocol is more efficient in terms of time delay and message complexity than other ACPs.

1 Introduction

Recently, much emphasis has been laid on the development of distributed systems integrating several data sources. Standards like DTP from X/Open [6] and OTS from OMG [8] define transactional interfaces that allow distributed data sources to participate in a two-phase commit (2PC) protocol [7], generally coordinated by TP-Monitors. 2PC is used to preserve the atomicity of distributed transactions even in the presence of failures. Although 2PC is the most widely used A*tomic Commit Protocol* (ACP), it has two major drawbacks. First, it is *blocking:* if the coordinator crashes between the *vote* phase and the *decision* phase, a transaction can hold system resources for an unbounded period. Second, it incurs three sequences of message rounds (request for vote, vote and decision) that introduce a substantial delay in the system even in the absence of failures. This makes 2PC not adequate to today's highly reliable distributed platforms.

[1] This work has been partially funded by the CEC under the OpenDREAMS Esprit project n°20843.

Several solutions have been proposed to eliminate the voting phase of the 2PC. The *early prepare* protocol [9] forces each participant to enter in a *prepare* state after the execution of each operation. A participant is thus ready to commit a transaction at any time thereby making its vote implicit. The main drawback comes from the fact that each operation has to be registered in the participant's log on disk, thus introducing a blocking I/O. The *coordinator log* protocol [9] exploits the *early prepare* idea and avoids the blocking I/O on the participants by centralizing the participant's log on the coordinator. However, this violates site autonomy by forcing participants to externalize their log records. More recently, the IYV (*implicit yes-vote*) protocol [1] has been proposed to exploit the performance and reliability properties of future gigabit-networked database systems. IYV capitalizes on the *early prepare* and *coordinator log* protocols. The underlying assumption in IYV is that all sites employ a rigorous concurrency control protocol [4]. Participants in a transaction pass in the acknowledgment messages their redo log files and read locks to the coordinator. Thus the coordinator can forward recover a transaction on failed participants. Although well adapted to gigabit-networked DBMSs, this protocol (i) does not preserve site autonomy by forcing participants to externalize logging and locking information, (ii) puts strong hypothesis on the network bandwidth and (iii) increases the probability of blocking since a coordinator crash that occurs at any time during the transaction processing will block the participants until the coordinator's recovery.

A number of non-blocking commit protocols have been proposed in the literature. The simplest is the *three phase commit protocol* (3PC) [10]. 3PC is non blocking at the expense of two extra message rounds needed to terminate a transaction even in the absence of failures. This high latency makes 3PC not adapted to today's systems with long mean time between failures. To avoid this problem, a new protocol, noted ACP-UTRB, was proposed [5]. ACP-UTRB shares the basic structure of 2PC (two phases to commit) and achieves non-blocking in synchronous systems by exploiting the properties of the underlying communication primitive it uses to broadcast decision messages.

In this paper, we propose a new atomic commitment protocol, noted NB-SPAC (Non-Blocking Single-Phase Atomic Commit). NB-SPAC has a *low latency* (one phase to commit), is *non-blocking* and preserves *site autonomy*. During normal execution as well as in case of one or more site failures, a transaction is committed in a single phase under the assumption that participating DBMSs are ruled by a rigorous concurrency control protocol [4]. This assumption is exploited to eliminate the voting phase of 2PC and to guarantee that our recovery mechanism preserves serializability. Non blocking is achieved by using a reliable broadcast primitive to deliver the decision messages [5]. Finally, NB-SPAC preserves site autonomy by exploiting techniques introduced in multidatabase systems to recover from failures [3].

This paper is organized as follows. Section 2 introduces the distributed transaction model we consider. Section 3 presents SPAC, our single phase atomic commitment protocol. Section 4 details NB-SPAC, the non-blocking version of SPAC. The two protocols share the same structure but differ in the way communications between the sites are handled. Section 5 introduces the recovery protocol, common to SPAC and NB-SPAC, needed to recover after site failures. In section 6, we compare the performance of NB-SPAC with other well known ACP protocols. Finally, section 7 summarizes the main contributions of this work.

2 Distributed Transaction Model

We consider distributed database systems built from pre-existing and independent local DBMSs located at different sites. Distributed transactions are allowed to access objects residing at these DBMSs under the control of a Global Transaction Manager (GTM). Each distributed transaction is decomposed by the GTM into one branch per accessed site. There is no restriction on parallel or sequential processing by the GTM of the operations belonging to the same transaction or to different distributed transactions. However, we consider that each operation is acknowledged by the local DBMS executing it. Each operation is sent to and executed by a single DBMS.

An important objective of the considered distributed transaction model is the preservation of *site autonomy*. This means that : (i) participating DBMSs do not have to be aware of the distribution, (ii) local DBMS information (e.g., log records or lock tables) is not externalized and cannot be exploited by the GTM. Site autonomy is a preliminary condition to the integration of existing systems [3]. Each participating DBMS is assumed to guarantee, by means of a local transaction manager (LTM), serializability and fault tolerance of all transactions (i.e., GTM's transaction branches) submitted to it. The GTM is responsible for the global serializability and the global atomicity of distributed transactions. A local GTM agent, noted GTMAgent, can be used to ease the integration of each local DBMS. This doesn't violate site autonomy as far as the existing interface of each local DBMS is preserved. To simplify notations, we consider a single DBMS per site (i.e., site S_k is synonym of $DBMS_k$) and a single GTM in the architecture. A typical example of a GTM is a TP-monitor. Clearly, several instances of GTMs can co-exist in the same architecture and may share the same GTMAgents. Note that the presence of several GTMs does not impact the algorithms. The distributed transaction model we consider is presented in Fig. 1.

We summarize below notations that will be used all along the paper :

• $op1_i$, $op2_i$, ..., opn_i is the sequence of operations performed by transaction T_i.

• ack(op) refers to the acknowledgment of a given operation op.

• T_{ik} refers to the branch of transaction T_i executed on site S_k.

• $commit_i$ (resp., $abort_i$) refers to the commit (resp., abort) of the transaction T_i.

• $commit_{ik}$ (resp., $abort_{ik}$) refers to the commit (resp., abort) of T_{ik}.

• $send_k(m)$ and $receive_k(m)$ respectively refer to the sending to site S_k and the receipt from site S_k of a given message m.

• $T_i \xrightarrow{RW} T_j$ represents a dependency between T_i and T_j due to a Read/Write conflict, that is T_j performed a write operation on a given object O which conflicts with a preceding T_i read operation on O. Similar notations are used to express Write/Write (WW) and Write/Read (WR) conflicts. $T_i \rightarrow T_j$ represents any dependency between T_i and T_j.

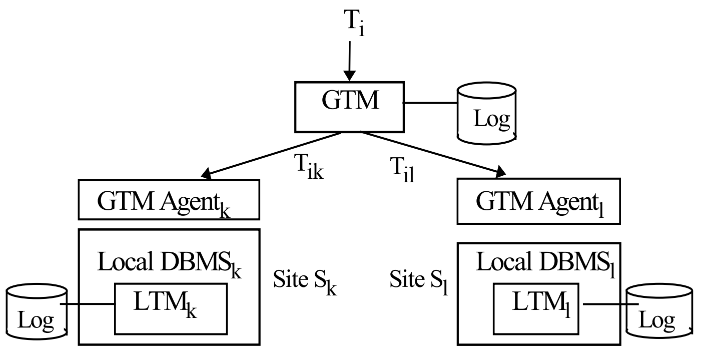


Fig. 1. Distributed transaction model

3 The Single-Phase Atomic Commit Protocol

3.1 Informal Description

The objective of this section is to introduce a Single-Phase Atomic Commitment protocol, noted SPAC, that preserves site autonomy (i.e., that can be used with existing DBMSs). For the rest of the paper, the words GTM (resp., GTM agents) and coordinator (resp., participants) will be used interchangeably to designate the coordinator (resp., participants) of the atomic commitment protocol.

The basic idea on which SPAC relies is eliminating the voting phase of 2PC by exploiting the properties of the local DBMSs serialization protocols. Let us assume that all participating DBMSs serialize their transactions using a pessimistic concurrency control protocol and that this protocol avoids cascading aborts (e.g., strict 2-Phase Locking protocol [4]). In this context, a transaction that executes successfully all its operations can no longer be aborted due to a serialization problem. Thus, this transaction is guaranteed to be committed in a failure free environment. When the acknowledgments of all operations of a transaction T_i are received, this means that all the transaction branches T_{ik} have been successfully executed till completion. At this point, the application submits its commit demand to the GTM which can directly ask each site accessed by the transaction T_i to commit, with no synchronization between the sites. If a transaction branch, say T_{ik} is aborted by S_k during its execution for any problem (e.g., integrity constraint violation or non serializability), the GTM simply asks each accessed site to abort that transaction.

While the SPAC and the IYV protocols [1] share the same assumption concerning the serialization policies of the local DBMSs, they strongly differ in the way failures are handled. Assume that site S_k crashes during the one-phase commit of transaction T_i. Meanwhile, T_i may have been committed at other sites. To guarantee T_i atomicity, the effects of the transaction branch T_{ik} have to be forward recovered on S_k. In the IYV protocol, participants in a transaction pass their redo log files and read locks to the coordinator. Thus, the coordinator can forward recover a transaction on failed participants. Obviously, this approach violates site autonomy.

To enforce transaction atomicity without sacrificing site autonomy, SPAC maintains a logical redo log at the GTM that contains the list of operations sent to each site. In case of a participant crash during the one-phase commit, the failed transaction branches will be re-executed due to the operations registered in the GTM redo log. Logging logical operations instead of physical redo records is mandatory to preserve site autonomy. Global serializability and redo correctness are maintained due to the hypotheses made on the DBMSs serialization protocol properties. These hypotheses are detailed in the next section.

3.2 Hypotheses on Participating DBMSs

In order for our protocol to work properly, the participating DBMSs must satisfy the two following properties:

P1: If all operations of T_{ik} have been acknowledged by S_k, then T_{ik} is serializable on S_k

P2: Local transactions are strongly isolated

Assuming all sites satisfy property *P1* guarantees that if the GTM receives the acknowledgment of all operations of transaction T_i, T_i is serializable on all sites. Property *P2* means that local schedules are cascadeless. Moreover, *P2* is necessary in order to handle failures properly (see Section 5). If both *P1* and *P2* are satisfied and if the commit decision is submitted after having received the acknowledgment of all operations, the assertion *A1* given below is true. It follows that a single-phase commit protocol is sufficient to guarantee the atomicity of distributed transactions in a failure free environment.

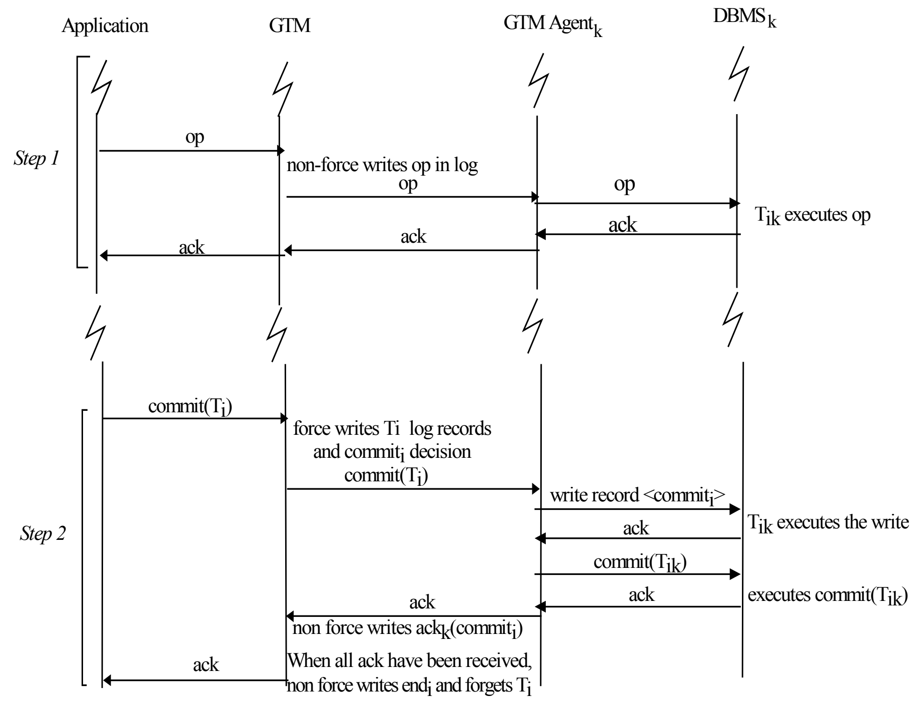
A1: $[\forall k,\ \text{send}_k(\text{commit}_i)]$ by the GTM => T_{ik} will be committed if S_k doesn't crash

In [2], we showed that only rigorous DBMSs (i.e., DBMSs ruled by a rigorous concurrency control protocol) satisfy *P1* and *P2*. This restriction is not so strong since almost all today's DBMSs (either relational or object oriented) are ruled by strict 2PL which is a rigorous serialization protocol. However, commercial relational DBMSs are likely to use levels of isolation standardized by SQL2 [11]. These DBMSs do not satisfy properties *P1* and *P2* since they accept some non serializable schedules. In [2], we showed that NB-SPAC can be extended to deal with DBMSs providing levels of isolation.

3.3 Protocol Description

The standard scenario induced by the SPAC protocol is depicted in Fig. 2. To simplify the figure, we show the actions executed by a single GTM and a single site, assuming that all GTMs and sites follow the same scenario.

Step 1 represents the sequence of operations executed on behalf of T_{ik}. The coordinator registers in its log each operation that is to be executed. Note that this registration is done by a non-forced write. Non-forced writes are buffered in main memory and do not generate blocking I/O. Operations are then sent to and locally executed by $DBMS_k$. Every operation is acknowledged up to the application.



Application
GTM
GTM Agent$_k$
DBMS$_k$
Step 1
op
non-force writes op in log
op
op
T_{ik} executes op
ack
ack
ack
Step 2
commit(T_i)
force writes Ti log records
and commit$_i$ decision
commit(T_i)
write record <commit$_i$>
T_{ik} executes the write
ack
commit(T_{ik})
ack
executes commit(T_{ik})
ack
non force writes ack$_k$(commit$_i$)
ack
When all ack have been received, non force writes end$_i$ and forgets T_i

Fig. 2. Basic SPAC protocol scenario

Step 2 concerns the atomic commitment of T_i. If the application issues a commit request to the GTM, this means that all operations of T_i have been successfully executed (consequently T_i is serializable at all sites). The GTM can thus take the commit decision. First, the GTM force writes all T_i log records along with its commit decision on stable storage in the same step. Assuming that the GTM log is a sequential file, this step will generate a single blocking I/O, the size of which depends on the activity of T_i. Second, the GTM asks its agents to commit T_i by broadcasting the commit decision to all.

In order to be able to know whether or not a failed DBMS has effectively committed a transaction and without violating the DBMS autonomy, each agent creates in the local database a special record containing the commit decision for T_i. This record is created by means of an ordinary write operation inside T_i.[2] This operation will be treated by the DBMS in exactly the same manner as the other operations belonging to $T_{i,}$ that is either all committed or all aborted atomically. Once the write operation of the commit decision record has been acknowledged by the DBMS, the GTM agent

[2] Note that this operation cannot generate a dependency cycle since it is the last operation executed in any transaction that has to be committed.

asks the local DBMS to commit T_i. In a failure free environment, this commit always succeeds and is acknowledged up to the GTM. For every transaction T_i, the GTM non-force writes each $ack_k(commit_i)$ in its log. When the acknowledgment is received from all the participants in the transaction, the GTM non-force writes an end_i record and forgets the transaction.

If transaction T_i is to be aborted, Step 2 is simpler than its committing counterpart. The GTM discards all T_i log records and broadcasts the abort decision to all. SPAC is a presumed abort protocol. Thus, abort messages are not acknowledged and the abort decision and transaction initiations are not recorded in the GTM log.

To be correct, an atomic commitment protocol must satisfy the following property :

AC: all participants that decide reach the same decision.

In [2], we showed that SPAC satisfies property *AC* of an atomic commitment protocol. A key step in our protocol is the dissemination of the decision message by the GTM to all participants in a transaction. In SPAC, this is achieved through a simple broadcast primitive. This primitive, noted *broadcast*, can be implemented using multiple *send* and *receive* operations [5]. Note that this primitive is not reliable (i.e., not atomic): if the GTM crashes in the middle of the broadcast of the decision message, some participants may deliver the decision while others never do so. Consequently, the SPAC protocol may produce blocking situations in case of a GTM failure. Assume the GTM crashes during the transaction execution or during the broadcast of the decision. In this case, each GTMAgent that did not receive the decision remains blocked until the GTM recovers. Since no termination protocol can solve all blocking situations [5], we propose in the next section an extension of the SPAC protocol which guarantees the non-blocking property in all possible crash failure cases.

4 Achieving Non-Blocking

4.1 System Model

In the previous section, we made no assumptions whatsoever concerning message transfer delays or communication failures since the SPAC protocol can be used in systems that are subject to site failures, communication failures as well as unknown time delays to transport messages. In order to achieve non-blocking, we follow the model and terminology used in [5] and assume, for the rest of the paper, a synchronous system composed of a set of DBMSs completely connected through a set of communication channels.[3] Since it is well known that distributed systems with unreliable communication do not admit non-blocking solutions to the atomic commitment problem [7], we also assume a reliable communication between the sites. A synchronous system means that there is a known upper bound on communication delays, that is each message is received within δ time units after being sent. In such a model, site failures can be reliably detected by any *reliable failure detector* and reported to any

[3] An adaptation of our non-blocking protocol to asynchronous systems is currently being studied.

operational site. Reliable failure detectors can be easily implemented by means of time-out. At any given time, a site may be *operational* or *down*. Operational sites may go down due to crash failures. A site that is down may become operational again by executing a recovery protocol. Finally, we say that a site is *correct* if it has never crashed ; otherwise it is *faulty*.[4]

4.2 The Non-Blocking Single-Phase ACP Algorithm

SPAC has failure scenarios for which no termination protocol can lead to a decision. This is due to the unreliability of the broadcast primitive used in SPAC, which allows faulty GTMAgents to deliver a message that is never delivered by correct ones. To solve this problem, we need a broadcast primitive, called Reliable Broadcast [5], which guarantees the following properties :

Uniform Agreement : if any GTMAgent, correct or not, delivers a message *m*, then all correct GTMAgents will eventually deliver *m*.

Δ-Timeliness : There exists a known constant Δ such that if the broadcast of a message *m* is initiated at time t, then no participant receives *m* after time t+Δ.

This primitive, noted *R_broadcast*, can be implemented as follows [5]. Every participant (GTMAgent) relays the message it receives for the first time to all the others before delivering it to the local DBMS. It is obvious that this implementation satisfies the Uniform Agreement property. To prove that it satisfies the Δ-Timeliness property, [5] showed that there exists a constant delay $\Delta = (F+1)\delta$, where F denotes the maximum number of sites that may crash during the execution of the atomic commitment protocol, by which the delivery of *m* must occur.

The blocking SPAC can be made non-blocking by replacing the unreliable broadcast primitive used by the GTM to disseminate its decision message with the reliable primitive *R_broadcast*. Fig. 4 illustrates the algorithm implementing the Non-Blocking Single-Phase Atomic Commit protocol, noted NB-SPAC.

GTM algorithm

```
1- do forever
2-    wait for (receipt of a message m from the application)
3-       case(m) of:
4-          op1_i():     send_{GTMAgent_k}(op1_i) ;
5-          opn_i():     send_{GTMAgent_k}(opn_i) ;
6-          abort_i:     R_broadcast_A(abort_i) ;[5]
7-          commit_i:    force write Ti log records and
                            T_i commit decision in GTM's log ;
8-                       R_broadcast_A(commit_i) ;
```

[4] The period of interest of these definitions is the duration of the atomic commitment protocol.

[5] A stands for the set of all GTM agents participating in transaction T_i.

$GTMAgent_k$ algorithm

```
9- do forever
10-  wait for (receipt of a message m from the GTM or delivery of a decision message m
                 by the broadcast or receipt of GTM crash notification )
11-      if (receipt or delivery of m) then treat message m
12-      else set time-out to $\Delta$ ;                  // GTM crash notification
13-          wait until (delivery of a decision by the broadcast or time-out expiration )
14-              if (delivery of a decision) then $send_{localDBMS}(decision_{ik})$
15-              else $send_{localDBMS}(abort_{ik})$ ;
```

Fig. 4. NB-SPAC algorithm

An atomic commitment protocol is said to be non-blocking if it satisfies the following property in addition to property *AC*:

NBAC: Every correct participant involved in the atomic commitment protocol eventually decides.

We showed in [2] that NB-SPAC satisfies properties *AC* and *NBAC* of an atomic commitment protocol.

5 Recovering Failures

To preserve global atomicity even in the presence of site failures, we need to forward recover a globally committed transaction on participants that failed during the commit procedure. This section describes the recovery algorithm executed after a participant crash, that allows such a recovery. The GTM recovery algorithm was presented in [2]. Due to space limitations, we omit its presentation in this paper. Fig. 5 details the participant's recovery algorithm common to SPAC and NB-SPAC. For the sake of clarity, step numbers correspond here to the steps ordering.

Step 1 and Step 2 represent the standard recovery procedure executed by the local DBMS on the crashed site. To preserve site autonomy, we make no assumptions on the way these steps are handled. Anyway, we assume that each local DBMS integrates a Local Transaction Manager that guarantees atomicity and durability of all transactions submitted to it. Since the GTM agents do not maintain a private log, Step 3 is necessary to determine if some globally committed transactions have to be locally re-executed at the crashed site S_k. A $GTMAgent_k$ must contact all GTMs that may have run transactions on site S_k. To simplify, we assume that each active instance of GTM is recorded by each GTM agent by means of records stored in the local DBMS. This can be done at the first connection of a GTM to the GTMAgent in order to avoid, during a transaction processing, the blocking I/O needed to record the GTM associated with the transaction. Other solutions are possible to record active GTMs.[6] Anyway, they do not impact the rest of the recovery procedure.

[6] In practical situations, a local DBMS is often under the control of a single GTM (e.g., TP-monitor) recorded statically. In more general situations, active GTMs may be registered by external traders in distributed platforms.

In Step 4, the GTM aborts all still active transactions in which the crashed site S_k participates. Step 5 checks if there exists some globally committed transaction T_i for which the GTMAgent$_k$ did not acknowledge the commit. This may happen in two situations. Either the participant crashed before the commit of T_{ik} was achieved and T_{ik} has been backward recovered during Step 2, or T_{ik} is locally committed but the crash occurred before the acknowledgment was sent to the GTM. Note that these two situations must be carefully differentiated. Re-executing T_{ik} in the latter case may lead to inconsistencies if T_{ik} contains non-idempotent operations (i.e., $op(op(x)) \neq op(x)$). Thus, the GTM must ask the GTMAgent$_k$ about the status of T_{ik} (Step 6). During Step 7, if GTMAgent$_k$ finds a record <commit$_i$> in the underlying database, this proves that T_{ik} has been successfully committed by the local DBMS, since write<commit$_i$> is an operation performed on behalf of T_{ik}. If, on the other hand, <commit$_i$> is not present in the database, this means that T_{ik} has been backward recovered during Step 2 and must be entirely re-executed. This re-execution is performed by exploiting the GTM log (Step 8). In both cases, the GTMAgent$_k$ informs the GTM about the status of T_{ik}. Once the recovery procedure is completed, new distributed transactions are accepted by the GTM (Step 9) and by the GTM agent (Step 10).

```
                         Local DBMS algorithm
1-  forward recover the transactions that reached their commit state before the crash.
2-  backward recover all other transactions
                         GTM agent algorithm
3-  once the local DBMS has restored a consistent state, contact the GTM(s)
7-  if the GTM asks about the local status of a transaction branch during step 6,
    check the corresponding commit decision record and answer the GTM
10- accept new transactions
                         GTM algorithm
If contacted by the GTM agent of site $S_k$ during step 3, do:
    for each transaction $T_i$ in which $S_k$ participates
        4- if (commit$_i$) $\notin$ GTM_log then
               broadcast (abort$_i$) to all other $T_i$ participants and forget $T_i$
        5- if (commit$_i$) $\in$ GTM_log,
               if ack$_k$(commit$_i$) $\notin$ GTM_log, then
                   6- ask GTMAgent$_k$ for the exact state of $T_{ik}$
                   8- if $T_{ik}$ has not been locally committed, then
                          restart a new transaction $T'_{ik}$ on $S_k$
                          re-execute all $T_{ik}$ operations within $T'_{ik}$
9- accept new distributed transactions
```

Fig. 5. Participant recovery algorithm

This recovery algorithm guarantees that re-executing the concerned transaction branches in Step 8: (i) will not produce undesirable conflicts and (ii) will generate exactly the same database state than after their first execution.

Let us call φ the set of all transaction branches that have to be re-executed at S_k during Step 8.

$\varphi = \{T_{ik}$ / commit$_i \in$ GTM_log $\wedge$ ack$_k$(commit$_i$) $\notin$ GTM_log $\wedge$ <commit$_i$> $\notin$ S_k database$\}$.

Proof of condition (i): Since local DBMSs are rigorous, $\forall T_{ik}, T_{jk} \in \varphi, \neg\exists(T_{jk} \rightarrow T_{ik})$. Otherwise, T_{ik} would have been blocked during its first execution, waiting for the termination of T_{jk}, and would not have complete all its operations, which contradicts $T_{ik} \in \varphi$. This means that T_{ik} and T_{jk} do not compete on the same resources, thereby precluding the possibility of conflict occurrence.

Proof of condition (ii): Step 2 of the recovery algorithm guarantees that all resources accessed by any $T_{ik} \in \varphi$ are restored to their initial state (i.e., the state before T_{ik} execution). The re-execution of all other transaction branches belonging to φ cannot change this state due to condition (i). Since Step 8 precedes Step 9 and Step 10, new transactions that may modify T_{ik} resources are executed only after T_{ik} re-execution. Consequently, T_{ik} is guaranteed to re-access the same state of its resources and consequently, will generate the same database state as after its first execution.

Since the recovery procedure may lead to a decision, we need to prove that this decision is not in conflict with any decision taken by correct participants or by the GTM. We showed in [2] that the recovery algorithms guarantee property *AC* stating that all participants (faulty or not) that decide, reach the same decision.

6 Performance Analysis

In this section, we compare the performance of SPAC and NB-SPAC with 2PC, the most well-known *blocking* ACP, and ACP-UTRB, the most efficient *non-blocking* protocol for synchronous systems, respectively. In Table1, *n* denotes the number of participants in the transaction. The performance of NB-SPAC and ACP-UTRB depends on the performance of the reliable broadcast primitive. [5] proposed different optimizations to this primitive that reduce the number of messages from quadratic to linear. These optimizations can be directly used in NB-SPAC, making it even more efficient than 2PC.

Table 1. Steps, time delay and message complexity for SPAC, 2PC, NB-SPAC and ACP-UTRB

Protocol	*Number of steps*	*Time Delay*	*Message Complexity*
SPAC	1	δ	n
2PC	2	3δ	3n
NB-SPAC	1	Δ	n^2
ACP-UTRB	2	$2\delta+\Delta$	$2n+n^2$

7 Conclusion

In this paper, we proposed an atomic commitment protocol, named NB-SPAC, which provides the non-blocking property while having the lowest latency possible for an ACP (one phase to commit). The underlying assumption in NB-SPAC is that all participants are ruled by a rigorous concurrency control protocol. In [2], we showed that this assumption can be relaxed to accept participants supporting SQL2 levels of isolation [11], making it practical for most commercial DBMSs. It is worth noting that NB-SPAC preserves site autonomy making it compatible with existing DBMSs.

While SPAC requires no assumption on the underlying network, we considered a synchronous communication system in the implementation of NB-SPAC. We are currently studying the possibility of extending NB-SPAC to deal with asynchronous systems.

References

1. Y. Al-Houmaily and P. Chrysanthis. Two-phase Commit in Gigabit-Networked Distributed Databases. In *Proceedings of the 8th International Conference on Parallel and Distributed Computing Systems*, September 1995
2. M. Abdallah and P. Pucheral. A Single-Phase Non-Blocking Atomic Commitment Protocol. Technical Report 97/019, PRiSM Laboratory, University of Versailles, November 1997. Available from ftp ://ftp.prism.uvsq.fr/pub/reports/1997/1997.019.ps.gz
3. Y. Breitbart, H. Garcia-Molina, and A. Silberschatz. Overview of Multidatabase Transaction Management. *VLDB Journal*, October 1992
4. P. A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems*, Addison Wesley, 1987
5. O. Babaoglu and S.Toueg. Non-Blocking Atomic Commitment. In Sape Mullender, editor, *Distributed Systems*, ACM Press, 1993
6. CAE Specification, *Distributed Transaction Processing: the XA Specification*, XO/CAE/91/300, 1991
7. J. Gray. Notes on Database Operating Systems. In *Operating Systems: An Advanced Course*. R. Bayer, R.M. Graham and G. Seegmuller editors, LNCS 60, Springer Verlag, 1978
8. *Object Transaction Service,* OMG Document 94.8.4, OMG editor, August 1994.
9. J. Stamos and F. Cristian. A Low-Cost Atomic Commit Protocol. In *Proceedings of ninth Symposium on Reliable Distributed Systems*, October 1990
10. D. Skeen. Non-Blocking Commit Protocols. In *Proceedings of the ACM-SIGMOD International Conference on Management of Data*, 1981
11. International Standardization Organization IS, *Information Processing Systems - Database Language SQL*, ISO/IEC 9075, 1992