A Low-Latency Non-Blocking Commit Service[*][†]

R. Jiménez-Peris[1], M. Patiño-Martínez[1], G. Alonso[2], and S. Arévalo[3]
[1] Technical University of Madrid (UPM), Facultad de Informática,
E-28660 Boadilla del Monte, Madrid, Spain, {rjimenez, mpatino}@fi.upm.es
[2] Swiss Federal Institute of Technology (ETHZ), Department of Computer Science,
CLU E1, ETH-Zentrum, CH-8092 Zürich, Switzerland, alonso@inf.ethz.ch
[3] Universidad Rey Juan Carlos, Escuela de Ciencias Experimentales, Móstoles,
Madrid, Spain, s.arevalo@escet.urjc.es

July 25, 2001

Abstract

Atomic commitment is one of the key functionalities of modern information systems. Conventional distributed databases, transaction processing monitors, or distributed object platforms are examples of complex systems built around atomic commitment. The vast majority of such products implement atomic commitment using some variation of 2 Phase Commit (2PC) although 2PC may block under certain conditions. The alternative would be to use non-blocking protocols but these are seen as too heavy and slow. In this paper we propose a non-blocking distributed commit protocol that exhibits the same latency as 2PC. The protocol combines several ideas (optimism and replication) to implement a scalable solution that can be used in a wide range of applications.

1 Introduction

Atomic commitment (AC) protocols are used to implement atomic transactions. *Two-phase commit* (2PC) [Gra78] is the most widely used AC protocol although its blocking behavior is well known. There are also non-blocking protocols but they have an inherent higher cost [DS83, KR01] usually translated in either a explicit extra round of messages (*3 phase commit* (3PC) [Ske81, Ske82, KD95]) or an implicit one (when using uniform multicast [BT93]).

[*]This research has been partially funded by the Spanish National Research Council CICYT under grant TIC98-1032-C03-01.

[†]Preliminar version of the paper published in Proc. of Distributed Computing Conference (DISC), 2001, LNCS, Springer, Lisboa, Portugal.

The reason why 2PC is the standard protocol for atomic commitment is that transactional systems pay as much attention to performance as they do to consistency. For instance, most systems summarily abort those transactions that have not committed after a given period of time so that they do not keep resources locked. Existing non-blocking protocols resolve the consistency problem by increasing the latency and, therefore, are not practical. A realistic non-blocking alternative to 2PC needs to consider both consistency and transaction latency. Ideally, the non-blocking protocol should have the same latency as 2PC. Our goal is to implement such a protocol by addressing the three main sources of delay in atomic commitment: message overhead, forced writes to the log, and the *convoy effect* caused by transactions waiting for other transactions to commit.

To obtain non-blocking behavior, it is enough for the coordinator to use a virtual synchronous uniform multicast protocol to propagate the outcome of the transaction [BT93]. This guarantees that either all or none of participants know about the fate of the transaction. Uniformity [HT93] ensures the property holds for any participant, even if it crashes during the multicast. Unfortunately, uniformity is very expensive in terms of the delay it introduces. In addition, since the delay depends on the size of the group, using uniformity seriously compromises the scalability of the protocol. To solve this two limitations, we use two different strategies. First, to increase the scalability, uniform multicast is used only within a small group of processes (the *commit servers*) instead of using it among all participants in the protocol. The idea is to employ a hierarchical configuration where a small set of processes run the protocol on behalf of a larger set of participants. Second, to minimize the latency caused by uniformity, we resort to a novel technique based on optimistic delivery that overlaps the processing of the transactional commit with the uniform delivery of the multicast. The idea here is to hide the latency of multicast behind operations that need to be performed anyway. This is accomplished by processing messages in an optimistic manner and hoping that most decisions will be correct although in some cases transactions might need to be aborted. This approach builds upon recent work in optimistic multicast [PS98] and a more aggressive version of optimistic delivery proposed in the context of Postgres-R [KPAS99] and later used to provide high performance eager replication in clusters [PJKA00]. We use an optimistic uniform multicast that delivers messages in two steps. In the first step messages are delivered optimistically as soon as they are received. In the second step messages are delivered uniformly when they become stable. This optimistic uniform multicast is equivalent to a uniform multicast with safe indications [VKCD99].

Forced writes to the log are another source of inefficiencies in AC protocols. To guarantee correctness in case of failures, participants must flush to disk a log entry before sending their vote. This log entry contains all the information needed by a participant to recall its own actions in the event of a crash. The coordinator is also required to flush the outcome of the protocol before communicating the decision to the participants (this log entry can be skipped by using the so called *presume commit* or *presume abort* protocols [MLO86]). Flushing log records adds to the overall latency as messages cannot be sent or responded to before writing to the log. In the protocol we propose, this delay is reduced by allowing sites to send messages instead of flushing log records. The idea is to use the main memory of a replicated group (the commit servers mentioned above) as stable memory instead of using a mirrored log with careful writes.

Finally, to minimize the waiting time of transactions, in our protocol locks are released optimistically. The idea is that a transaction can be optimistically committed pending the confirmation provided by the uniform multicast. By optimistically committing the transaction, other transactions can proceed although they risk a rollback if the transaction that was optimistically committed ended up aborting. In our protocol, the optimistic commit is performed in such a way that aborts are confined to a single level. In addition, transactions are only optimistically committed when all their participants have voted affirmatively, thereby greatly reducing the risk of having to abort the transaction. This contrasts with other optimistic commit protocols, e.g., [GHR97], where transactions that must abort (because one or more participants voted abort) can be optimistically committed although they will rollback anyway producing unnecessary cascading aborts.

With these properties the protocol we propose satisfactorily addresses all design concerns related to non-blocking AC and can thus become an important contribution to future distributed applications. The paper is organized as follows, Section 2 describes the system model. Section 3 and 4 present the commit algorithm and its correctness. Section 5 concludes the paper.

2 Model

2.1 Communication Model

The system consists of a set of fail-crash processes connected through reliable channels. Communication is asynchronous and by exchanging messages. A failed process can later recover with its permanent storage intact and re-join the system. Failures are detected using a (possibly unreliable) failure detector[1] [CT96].

A virtual synchronous multicast service [BSS91, Bir96, SR93] is used. This service delivers multicast messages and views. Views indicate which processes are perceived as up and connected. We assume a virtual synchrony with the following properties: (1) *Strong virtual synchrony* [FvR95] or sending view delivery [VKCD99] that ensures that messages are delivered in the same view they were sent; (2) *Majority or primary component views* that ensure that only members within a majority view can progress, while the rest of the members block until joining again the majority view; (3) *Liveness*, when a member fails or it is partitioned from the majority view, a view excluding the failed member will be

[1]The failure detector must allow the implementation of the virtual synchrony model described below (e.g., the one proposed in [SR93]) and the non-blocking atomic commitment (e.g., the one in [GLS95]).

eventually delivered.

The protocol uses two different multicast primitives [HT93, SS93]: reliable (*rel-multicast*) and uniform multicasts (*uni-multicast*). Three primitives define optimistic uniform reliable multicast[2]: *Uni-multicast(m, g)* multicasts message m to a group g. *Opt-deliver(m)* delivers m reliably to the application. *Uni-deliver(m)* delivers m to the application uniformly. We say that a process is $v_i - correct$ in a given view v_i if it does not fail in v_i and if v_{i+1} exists, it transits to it. The rel- and uni-multicasts preserve the following properties, where m is a message, g a group of processes and v_i a view within this group:

OM-Validity: If a correct process rel or uni-multicast m to g in v_i, m will be eventually opt-delivered by every v_i-correct process.

OM-Agreement: If a v_i-correct process opt-delivers m in v_i, every v_i-correct process will eventually opt-deliver m.

OM-Integrity: Any message is opt and uni-delivered by a process at most once. A message is opt-delivered only if it has been previously multicast.

Uni-multicast additionally fulfills the following properties:

OM-Uniform-Agreement: If a majority of v_i-correct processes opt-deliver m, they will eventually uni-deliver m. If a process uni-delivers m in v_i, every v_i-correct process will eventually uni-deliver m.

OM-Uniform-Integrity: A message is uni-delivered in v_i only if it was previously opt-delivered by a majority of processes in v_i.

2.2 Transaction Model

Clients interact with the database by issuing transactions. A transaction is a partially ordered set of *read* and *write* operations followed by either a *commit* or an *abort* operation. The decision whether to commit or abort is made after executing an optimistic atomic commitment protocol. The protocol can decide (1) to immediately abort the transaction, (2) to perform an *optimistic commit*, or (3) decide to commit or abort.

We assume a distributed database with n sites. Each site i has a *transaction manager* process TM_i. When a client submits a transaction to the system, it chooses a site as its *local* site. The local TM_i decides which other sites should get involved in processing the transaction and initiates the commitment protocol.

We assume the database uses standard mechanisms like *strict 2 phase locking* (2PL) to enforce *serializability* [BHG87]. The only change over known protocols is introduced during the commit phase of a transaction. When a transaction t is optimistically committed, all its write locks are changed to *opt* locks and all its read locks are released[3]. *Opt* locks are compatible with all other types of

[2]Senders are not required to belong to the target group.

[3]Once a transaction concludes, all its read locks can be released without compromising correctness independently of whether the transaction commits or aborts [BHG87].

locks. That is, other transactions are allowed to set compatible locks on data held under an *opt* lock. Such transactions are said to be *on-hold*, while the rest of transactions are said to be on *normal* status. When the outcome of t is finally determined, its *opt* locks are released and all transactions that were on-hold due to these *opt* locks are returned to their normal state. A transaction that is on-hold cannot enter the commit phase until it returns to the normal state.

2.3 System configuration

For the purposes of this paper, we will assume there are two disjoint groups of processes in the system. The first will conform the distributed database and will be referred as the *transaction managers* or TM group ($TM = \{TM_1, ..., TM_n\}$). Sites in this group are responsible for executing transactions and for triggering the atomic commitment protocol. By participants in the protocol, we mean processes in this group. The second group, *commit server* or CS group ($CS = \{C_1, ..., C_n\}$) is a set of replicated processes devoted to perform the AC protocol. We assume that in any two consecutive views, there is a process that transits from the old view to the new one[4].

2.4 Problem definition

A non-blocking AC protocol should satisfy: (1) NBAC-Uniform validity, a transaction is (opt) committed only if all the participants voted yes; (2) NBAC-Uniform-Agreement, no two participants decide differently; (3) NBAC-Termination, if there is a time after which there is a majority view sequence in the CS group that permanently contains at least a correct process, then the protocol terminates; (4) NBAC-Non-Triviality [Gue95], if all participants voted yes, and there no failures or false suspicions, then commit is decided.

3 A Low Latency Commit Algorithm

3.1 Protocol Overview

The AC protocol starts when a client requests to commit a transaction. The commit request arrives at a transaction manager, TM_i, which then starts the protocol. The protocol involves several rounds of messages in two phases:

First phase

1. Upon delivering the commit request, TM_i multicasts a reliable *prepare to commit* message to the TM group. This message contains the transaction

[4]It might seem a strong assumption for safety that at least one server must survive between views. However, this assumption is no stronger than the usual one that assumes that the log is never lost. The strength of any of the assumptions depends on the probability of the corresponding catastrophic failures.

identifier (*tid*) to be committed and the number of participants involved (the number of TMs contacted during the execution of the transaction).

2. Upon delivering the *prepare to commit* message, each participant uni-multicasts its *vote* and the number of participants to the CS group. If a participant has not yet written the corresponding entries to its local log when the *prepare to commit* message arrives, it sends the log entry in addition to its vote without waiting to write to the log. After the message has been sent, it then writes the log entry to its local disk.

Second phase

1. Upon *opt-delivering* a *vote* message, the processes of the commit server decide who will act as *proxy coordinator* for the protocol based on the tid of the transaction and the current view. Assume this site is C_i. The rest of the processes in the CS group act as backup in case C_i fails. If a no vote is opt-delivered, the transaction is aborted immediately and an *abort* message is reliable multicast to the TM group. If all votes are yes, as soon as the last vote is opt-delivered at C_i, C_i sends a reliable multicast with an *opt-commit* message to the TM group.

2. Upon delivering an *abort* message, a participant aborts the transaction. Upon delivering an *opt-commit* message, the participant changes the transaction locks to *opt* mode.

3. If all votes are affirmative, when they have been *uni-delivered* at C_i, C_i reliable multicasts to the TM group a *commit* message.

4. When a participant delivers a *commit* or *abort* message, it releases all locks (both opt and non-opt) held by the transaction and return the corresponding transactions that were on hold to their normal state.

5. If all the votes are affirmative, the coordinator opt-commits the transaction before being excluded from the majority view (before being able to commit the transaction), and one or more votes do not reach the majority view, the transaction will be aborted by the new coordinator.

This protocol reduces the latency of the non-blocking commit in several ways. First, at no point in time in the protocol must a site wait to write a log entry to the disk before reacting to a message. The CS group acts as stable storage for both the participants (sites at the TM which could not yet write their vote and other transaction information to disk when the *prepare to commit* vote arrives) and the CS group itself (the coordinator does not need to write an entry to the log before sending the *opt-commit* message). Second, the coordinator in the CS group provides an outcome without waiting for the *vote* messages to be uniform. This reduces the overhead of uniform multicast as it overlaps its cost with that of committing the transaction.

3.2 The Protocol

The protocol uses the CS group to run the atomic commitment. The processes in the TM group[5] only act as participants and the CS group acts as coordinator. We use two tables, *trans_tab* and *vote_tab*, to store information in main memory about the state of a transaction and the decision of each participant regarding a given transaction at each CS_i. We also use a number of functions to change and access the values of the attributes in these tables. *Trans_tab* contains the attributes *tid* (the transaction's identifier), *n_participants* (number of participants in that transaction, all sites in the TM group), *timestamp* (of the first vote for timeout purposes), *coordinator* (id of the coordinator site in the CS group; this attribute is initially set with the function store_trans and updated with the function store_coordinator), and *outcome* (the state of the transaction; initially it is undecided, the state can be changed to aborted, opt-committed or committed by invoking the function *store_outcome*). *Vote_tab* contains the attributes *tid* (the transaction's identifier), *participant_id* (site emitting the vote, which must be a site in the TM group), *vote* (the actual vote), *vote_status* (initially optimistic, when set with the function store_opt_vote, and later definitive, when set with the function store_def_vote), and *log* (any log entry the participant may have sent with the vote). There are additional functions to consult the attributes associated to each tid in the *trans_tab*. These functions are denoted with the same name as the attribute but starting with capital letter (e.g., Timestamp). There are also functions to consult the *vote_tab*: Log (to obtain the log sent by a participant), *N_opt_yes_votes* (number of yes votes delivered optimistically for a particular transaction), *N_def_yes_votes* (similarly for uni-delivered yes votes). An additional function, *Coordinator*, is used to obtain the id of the coordinator of a transaction given its tid and the current view.

TM Group actions:

```
TM.A  Upon delivering Prepare(tid):
  if prepared in advance then
    uni-multicast(CS, Vote(tid, n_participants, my_id, vote, empty))
  else
    uni-multicast(CS, Vote(tid, n_participants, my_id, vote, log_record))
  end if
```

```
TM.B  Upon delivering Opt-commit(tid):
  Change transaction tid locks to opt-mode
```

```
TM.C  Upon delivering Commit/Abort(tid):
  Commit/Abort the transaction and release transaction tid locks
  Change the corresponding on-hold transactions to normal status
```

CS Group actions:

[5]For simplicity, messages are multicast to all TM processes. Processes for which the message is not relevant just discard it.

```
$\boxed{\text{CS.A}}$ Upon **opt-delivering Vote**(tid, n_participants, participant_id, vote, log):
 store_opt_vote(vote_tab, tid, participant_id, vote, log)
 – *the transaction outcome is still undecided*
 if Outcome(trans_tab, tid) = undecided then
    if vote = no then
       if Coordinator(current_view, tid) = my_id then
          rel_multicast(TM, Abort(tid))
       end if
       store_outcome(trans_tab, tid, aborted)
    else – *vote = yes*
       if N_opt_yes_votes(vote_tab, tid) = 1 then – *it is the first vote*
          timestamp = current_time
          if Coordinator(current_view, tid) = my_id then
             set_up_timer(tid, timestamp+waiting_time)
          end if
          store_trans(trans_tab, tid, n_participants, timestamp)
       end if
       if N_opt_yes_votes(vote_tab, tid) = n_participants(trans_tab, tid) then – *all voted yes*
          store_outcome(trans_tab, tid, opt-committed)
          if Coordinator(current_view, tid) = my_id then
             disable_timer(tid)
             rel_multicast(TM, Opt-commit(tid))
          end if
       end if
    end if
$\boxed{\text{CS.B}}$ Upon **uni-delivering Vote**(tid, n_participants, participant_id, vote, log):
 store_def_vote(trans_tab, tid, participant_id)
 if (N_def_yes_votes(vote_tab, tid) = n_participants(trans_tab, tid))
       and (Outcome(trans_tab, tid) $\neq$ abort) then
    store_outcome(trans_tab, tid, committed)
    if Coordinator(current_view, tid) = my_id then
       rel_multicast(TM, Commit(tid))
    end if
 end if
$\boxed{\text{CS.C}}$ Upon **expiring Timer**(tid):
 store_outcome(trans_tab, tid, aborted)
 uni_multicast(CS, Timeout(tid))
$\boxed{\text{CS.D}}$ Upon **uni-delivering Timeout**(tid):
 store_outcome(trans_tab, tid, aborted)
 if Coordinator(current_view, tid) = my_id then
    rel_multicast(TM, Abort(tid))
 end if
```

```
$\boxed{\textbf{CS.E}}$ Upon **delivering ViewChange**($v_i$):
  current_view = $v_i$
  – *State synchronization with new members*
  if *my_id is the lowest in $v_i$ that belonged to $v_{i-1}$* then
    for every $C_i \in v_i$ do
      if $C_i \notin v_{i-1}$ then
        send($C_i$, State(trans_tab, vote_tab))
      end if
    end for
  elsif my_id $\notin v_{i-1}$ then *I am a new member*
    receive(State(trans_tab, vote_tab))
  end if
  – *Assignment of new coordinators in $v_i$*
  for each tid $\in$ trans_tab do
    if Coordinator($v_i$, tid) = my_id then
      if Outcome(trans_tab, tid) = committed then
        rel_multicast(TM, Commit(tid))
      elsif Outcome(trans_tab, tid) = aborted then
        rel_multicast(TM, Abort(tid))
      else
        set_up_timer(Timestamp(trans_tab, tid) + waiting_time)
      end if
    end if
  end for
end if
```

3.2.1 Dealing with coordinator failures

Since sites in the TM group only act as participants, failures in the TM group do not affect the protocol. In the CS group, all processes are replicas of each other. Strong virtual synchrony ensures that any pending message sent in the previous view is delivered before delivering a new view. Thus, when a process fails (or it is falsely suspected), a new view is eventually delivered to a majority of available connected CS processes. Once the new view is available, a working CS process takes over as coordinator for all the on-going commitment protocols coordinated by the failed process (CS.E). For each on-going transaction commit, the new coordinator checks the delivery time of the first vote and sets up a timer accordingly (CS.E). The actions taken by the new coordinator at this point in time depend on the protocol stage. If the transaction outcome is already known (all the votes have been opt-delivered at all CS members or a no vote message has been opt-delivered), the new coordinator multicasts the outcome to the participants (CS.E). If the outcome is undecided (i.e., all previously delivered votes were affirmative and there are pending votes), the protocol proceeds as normal and the new coordinator waits until all vote messages (or a no vote) have been *opt-delivered* (CS.A).

The problematic case is when the coordinator has decided to commit the transaction and then it is excluded from the view. It can be the case that the coordinator has had time to opt-commit the transaction, but not to commit it, and that there are missing votes in the majority view. In traditional 2PC, this situation is avoided by blocking. In our protocol, the blocking situation is avoided by the use of uniform multicast within the server group. The surviving sites can safely ignore the previous coordinator: due to uniformity, at worst, they will be aborting an opt-committed transaction, which does not violate consistency. A more accurate characterization of the rollback situation follows:

- All the votes are yes and have been opt-delivered at the coordinator.
- The coordinator successfully multicasts the opt-commit to the participants.
- The vote from at least one of the participants (e.g., p) is not uni-delivered.
- The coordinator and p become inoperative (e.g., due to a crash, a false suspicion, etc.) in such a way that the multicast does not reach any other member of the new primary view.

Although such a situation is possible, it is fair to say that it is extremely unlikely. Even during instability periods where false suspicions are frequent and many views are delivered, the odds for a message being opt-delivered only at the coordinator and both the coordinator and p become inoperative before the multicast of missing votes reaches any other CS member are very low. Additionally, this has to happen after the opt-commit is effective, as otherwise there would not be any rollback. Being such a rare situation, the amount of one-level aborts will be minimal even if the protocol is making optimistic decisions. It is also possible to enhance the protocol by switching optimism off during instability periods.

3.2.2 Bounding commit duration

To guarantee the liveness of the protocol and to prevent unbounded resource contention it is necessary to limit the duration of the commit phase of a transaction. This limitation is enforced by setting a timer at the coordinator when it receives the first vote from a transaction (CS.A). The rest of the members timestamp the transaction with the current time when they opt-deliver the first vote. If all participant votes have reached the coordinator before the timer expires, the timer is disabled (CS.A). Otherwise, the coordinator decides to abort the transaction but it does not immediately multicast the abort decision to the TM group (CS.C). Instead, it uni-multicasts a *timeout* message to the CS group. When this message is uni-delivered at the coordinator, a message is sent to the participants with the abort decision (CS.D).

It could be the case that the coordinator multicast a timeout message and, before uni-delivering it, the missing votes are opt-delivered at the coordinator. In that case the transaction will be aborted (its outcome is not undecided when the vote is opt-delivered since in CS.C the outcome is set to abort when the timer expires). The rest of the CS members will also abort the transaction, no matter the order in which those messages are delivered. If the missing votes are delivered before the timeout message, the transaction outcome will be set to commit (CS.A) until the timeout message is uni-delivered (if so). Upon uni-delivery of the timeout message the outcome is changed to abort (CS.D). If the timeout message is uni-delivered before the last vote at a CS member, the transaction outcome will be initially set to abort (CS.D) and remain so (CS.A).

The coordinator can be excluded from the majority view during this process and a new coordinator will take over. If the new coordinator has uni-delivered the timeout message, the outcome of the transaction will be abort (CS.E). This will happen independently of whether the old coordinator sent the abort message to the TM group. If the new coordinator has not uni-delivered the timeout message before installing the new view, the failed coordinator did not uni-deliver it either (due to uniformity) and the new coordinator will not deliver that message (strong virtual synchrony). Therefore, the new coordinator will behave as a regular coordinator and will set the timer and wait for any pending vote (CS.E to set the timer, and CS.A in the event a vote arrives).

Despite the majority view approach, the protocol would not terminate if all the coordinators assigned to a transaction are excluded from the view before deciding the outcome. For instance, the CS group can transit perpetually between views {1,2,3,4} and {1,2,3,5}, with processes 4 and 5 being the coordinators of a transaction t in each view. In this case, t will never commit. This scenario can be avoided by, whenever possible, choosing a coordinator that has not previously coordinated the transaction. It there is at least a correct process, this will guarantee that the outcome of t will be eventually decided, thereby, ensuring the liveness of the algorithm.

3.2.3 Maintaining consistency across partitions

Although partitions always lead to blocking, our protocol maintains consistency even when partitions occur. That is, no replica decides differently on the outcome of a transaction even when the network partitions. Consistency is enforced by combining uniformity, strong virtual synchrony, and majority views. To see why, we will only consider partitions in the CS group. Partitions in the TM may lead to delays in the vote delivery (which may result in a transaction abort) and to delays in the propagation of the transaction outcome (thus, resulting in blocking during the partition). Partitions that leave the coordinator of a transaction in the majority partition of the CS group are not a problem, as the minority partition gets blocked (due to the majority view virtual synchrony). Since the transaction outcome is always decided after the uni-delivery of a message (either a vote or timeout message), uniformity guarantees that the decision will be taken by every process in the majority view. When the coordinator of a transaction is in a minority partition, undecided transactions cannot create problems as the coordinator, once in the minority partition, will block. When this happens, a new coordinator can make any decision regarding undecided transactions without compromising consistency. Only transactions whose outcome has been decided by the coordinator during the partition may lead to inconsistencies. There are four cases to consider:

- The coordinator optimistically commits a transaction when it opt-delivers the last vote (and all votes have been affirmative). Assume a partition leaves the coordinator in a minority partition. The new coordinator may (1) opt-deliver all the votes or (2) never deliver one or more votes. In the first case, it will opt-commit the transaction (CS.A) thereby agreeing with the old coordinator. In the second case, it will abort the transaction once the timer expires (CS.C). Since the transaction was only optimistically committed by the old coordinator, the new coordinator is free to decide to abort without violating consistency.

- If the old coordinator committed a transaction, the new coordinator will do the same. A transaction is committed when all the votes have been uni-delivered (CS.B). If the votes were uni-delivered at the old coordinator, all the processes in that view also uni-delivered them in that view (uniformity and strong virtual synchrony). Thus, the new coordinator will also commit the transaction (CS.E).
- The old coordinator opt-delivered a no vote and aborted the transaction. The new coordinator will either have delivered the no vote or timed out. In both cases the new coordinator will also abort the transaction (CS.A and CS.C, respectively) thereby agreeing with the old coordinator.
- The old coordinator timed out and aborted the transaction. The transaction will not be effectively aborted until the timeout message is uni-delivered. Uniformity guarantees that the timeout message, if uni-delivered, will be uni-delivered to both the old and the new coordinator, thereby preventing any inconsistency.

3.2.4 Replica recovery and partition merges

In order, to maintain an appropriate level of availability, it is necessary to enable new (or recovered) replicas to join the CS group and to allow partitioned groups to merge again. When a new process joins the CS group, virtual synchrony guarantees that the new process will deliver all the messages delivered by the other replicas after installing the new view (and thus, after state synchronization). The installation of the new view will trigger state synchronization (CS.E). This involves sending from an old member of the group (one that transits from the previous view to the current one, that it is guaranteed to exist due to the majority view approach) a *State* message with the `vote_tab` and the `trans_tab` tables to the new member. Members from a minority partition that join a majority view will be treated as recovered members, that is, they will be sent the up-to-date tables from a process belonging to the previous majority view.

The state transfer and the assumption that at least a process from the previous view transits to the next view guarantees that a new member acting as coordinator will use up-to-date information, thereby ensuring the consistency of the protocol. The recovery of a participant has not been included in the algorithm due to its simplicity (upon recovery it will just ask to the CS group about the fate of some transactions).

4 Correctness

Lemma 1 (NBAC-Uniform-Validity) *A transaction (opt)commits only if all the participants voted yes.* □

Proof (lemma 1): (Opt) Commit is decided when the coordinator multicasts such message after the transaction has been recorded as (opt) committed in the *trans_tab* (CS.A). This can only happen when all participant votes have been (opt) uni-delivered and they are yes votes. □

Lemma 2 (NBAC-Uniform-Agreement) *No two CS members decide differently.* □

Proof (lemma 2): In the absence of failures the lemma is proved trivially, as only the coordinator decides about the outcome of the transaction. The rest of them just logs the information about the transaction in case they have to take over. The only way for two members to decide on the same transaction is that one is a coordinator of the transaction and then it is excluded from the majority view before deciding the outcome. Then, a CS member takes over as new coordinator and decides about the transaction.

Assume that a coordinator makes a decision and due to its exclusion from view v_i, a new coordinator takes over and makes a different decision. Let us assume without loss of generality that the new coordinator takes over in view v_{i+1} (in general, it will be in v_{i+f}). The old coordinator can have decided to:

1. *Commit.* The old coordinator can only decide commit if all votes have been uni-delivered, and they all were affirmative. If the new coordinator decides to abort, it can only be because it has not uni-delivered one or more votes neither before the view change nor before its timer expires. In this situation, there are two cases to consider:

   a. The new coordinator belonged to v_i. Hence, all votes were uni-delivered in v_i at the old coordinator (which needed all yes votes to decide to commit) but not at the new coordinator (otherwise it would also decide to commit) what violates multicast uniformity.

   b. The new coordinator joined the CS group after a recovery or a partition merge. From the recovery procedure, the new coordinator has gotten the most up-to-date state during the state transfer triggered by the view change. If it decides to abort, it is because a process in view v_i (the one which sent its tables in the state transfer) did not uni-deliver one or more votes. This again violates uniformity and it is therefore impossible.

2. *Abort due to a no vote.* If the old coordinator aborts the transaction, it does so as soon as the no vote is opt-delivered (CS.A). In order to decide to commit, the new coordinator needs to uni-deliver all votes and that all votes are yes. Since a participant votes only once, this situation cannot

occur (for it to occur, a participant needs to say no to the old coordinator and yes to the new one).

3. *Abort due to a timeout.* If the old coordinator decided to abort due to a timeout, then it uni-delivered its own timeout message (CS.C). If the new coordinator decides to commit, then it must have uni-delivered all the votes before its timer expires and before uni-delivering the timeout message. This implies that the timeout message has been delivered to the old coordinator in view v_i and not to the new coordinator. Now there are two cases to consider:

   a. If the new coordinator was in view v_i, the fact that the old coordinator has not received the timeout message violates multicast uniformity. It is therefore not possible for the new coordinator to have been in v_i.

   b. If the new coordinator was not in view v_i then it has joined the group in view v_{i+1}, and thus during the state synchronization it has received the most up-to-date tables. However, this implies that some process in the CS group was in view v_i, transited to v_{i+1}, but did not uni-deliver the timeout message. Again this violates uniformity.

From here, since in all possible cases the new coordinator cannot make a different decision than the old coordinator once the latter has made a decision (abort or commit), the lemma is proven. □

Lemma 3 (NBAC-Termination) *If there is a time after, which there is a majority view sequence in the CS group that permanently contains at least a correct process, then the protocol terminates.* □

Proof (lemma 3): Assume for contradiction that the protocol never ends. This means that either:

- A correct coordinator never decides. A correct coordinator will either: (1) Opt-deliver a no vote, in which case the transaction is aborted, or (2) uni-deliver all the votes and are all yes, in which case the transaction is committed, or (3) uni-deliver the timeout message before opt-delivering all the votes (and being all previous votes affirmative), in which case the transaction is aborted. Therefore, if there is a correct process, there will eventually be a correct coordinator that will decide the transaction outcome and multicast it to the participants, thus terminating the protocol.

- There is an infinite sequence of unsuccessful coordinators that do not terminate the protocol. The NewCoordinator function, whenever possible, chooses a fresh coordinator (a process that did not previously coordinate the transaction). This means that a correct process p will eventually coordinate the transaction. Since p is correct and belongs to the majority view, it will eventually terminate the protocol as shown before.

Thereby, it is proven that the protocol eventually terminates. □

Lemma 4 (NBAC-Non-Triviality) *If all participants votes yes there are no failures nor false suspicions then commit will be decided.* □

Proof (lemma 4): The abort decision can only be taken when the coordinator receives a no vote, or because it times out. Otherwise, the decision is commit. □

Theorem 1 (NBAC-Correctness) *The protocol presented in the paper fulfills the non-blocking atomic commitment properties: NBAC-Validity, NBAC-Uniform-Agreement, NBAC-Termination, and NBAC-Non-Triviality.* □

Proof (theorem 1): It follows from lemmas 1, 2, 3, and 4 □

5 Conclusions

Atomic commitment is an important feature in distributed transactional systems. Many commercial products and research prototypes use it to guarantee transactional atomicity (and, with it, data consistency) across distributed applications. The current standard protocol for atomic commitment is 2PC which offers reasonable performance but might block when certain failures occur. In this paper we have proposed a non-blocking atomic commitment protocol that offers the same reasonable performance as 2PC but that is non-blocking. Unlike previous work in the area, we have emphasized several practical aspects of atomic commitment. First, the new protocol does not create any additional message overhead when compared with 2PC. Second, by using a replicated group as stable memory instead of having to flush log records to the disk, the protocol is likely to exhibit a shorter response time than standard 2PC. Third, the fact that the second round of the commit protocol is run only by a small subset of the participants minimizes the overall overhead. Fourth, and most relevant in practice, the new protocol can be implemented on top of the same interface as that used for 2PC. This is because, unlike most non-blocking protocols that have been previously proposed, the participants only need to understand a *prepare to commit* message (or *vote-request*) and then a *commit* or *abort* message. This is exactly the same interface required for 2PC and it is implemented in all transactional applications. Because of these features, we believe the protocol constitutes an important contribution to the design of distributed transactional systems. We are currently evaluating the protocol empirically to get performance measures and are looking into several possible implementations to further demonstrate the advantages it offers.

References

[BHG87] P. A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems.* Addison Wesley, Reading, MA, 1987.

[Bir96] K.P. Birman. *Building Secure and Reliable Network Applications.* Prentice Hall, NJ, 1996.

[BSS91] K. P. Birman, A. Schiper, and P. Stephenson. Lightweight Causal and Atomic Group Multicast. *ACM Transactions on Computer Systems*, 9(3):272–314, August 1991.

[BT93] O. Babaoglu and S. Toueg. Understanding Non-Blocking Atomic Commitment. In *Distributed Systems*. Addison Wesley, 1993.

[CT96] T. D. Chandra and S. Toueg. Unreliable Failure Detectors for Reliable Distributed Systems. *Journal of the ACM*, 43(2):225–267, March 1996.

[DS83] C. Dwork and D. Skeen. The Inherent Cost of Nonblocking Commitment. In *Proc. of ACM PODC*, pages 1–11, 1983.

[FvR95] R. Friedman and R. van Renesse. Strong and Weak Virtual Synchrony in Horus. Technical report, CS Dep., Cornell Univ., 1995.

[GHR97] R. Gupta, J. Haritsa, and K. Ramamritham. Revisiting Commit Processing in Distributed Database Systems. In *Proc. of the ACM SIGMOD*, 1997.

[GLS95] R. Guerraoui, M. Larrea, and A. Schiper. Non-Blocking Atomic Commitment with an Unreliable Failure Detector. In *Proc. of the 14th IEEE Symp. on Reliable Distributed Systems*, Bad Neuenahr, Germany, September 1995.

[GLS96] R. Guerraoui, M. Larrea, and A. Schiper. Reducing the Cost for Non-Blocking in Atomic Commitment. In *Proc. of IEEE ICDCS*, 1996.

[Gra78] J. Gray. *Notes on Data Base Operating Systems. Operating Systems: An Advanced Course.* Springer, 1978.

[GS95] R. Guerraoui and A. Schiper. The Decentralized Non-Blocking Atomic Protocol. In *Proc. of IEEE SPDP*, 1995.

[Gue95] R. Guerraoui. Revistiting the Relationship Between Non-Blocking Atomic Commitment and Consensus. In *Proc. of the WDAG'95*, 1995.

[HT93] V. Hadzilacos and S. Toueg. Fault-Tolerant Broadcasts and Related Problems. In *Distributed Systems*, pages 97–145. Addison Wesley, 1993.

[KD95] I. Keidar and D. Dolev. Increasing the Resilience of Atomic Commit at No Additional Cost. In *Proc. of ACM PODS*, 1995.

[KPAS99] B. Kemme, F. Pedone, G. Alonso, and A. Schiper. Processing Transactions over Optimistic Atomic Broadcast Protocols. In *Proc. of 19th IEEE Int. Conf. on Distributed Computing Systems (ICDCS)*, pages 424–431, 1999.

[KR01] I. Keidar and S. Rajsbaum. On the Cost of Faul-Tolerant Consensus Where There No Faults - A Tutorial. Technical Report MIT-LCS-TR-821, 2001.

[MLO86] C. Mohan, B. Lindsay, and R. Obermark. Transaction Management in the R* Distributed Database System. *ACM Transactions on Database Systems*, 11(4), February 1986.

[PJKA00] M. Patiño Martínez, R. Jiménez Peris, B. Kemme, and G. Alonso. Scalable Replication in Database Clusters. In *Proc. of Distributed Computing Conf., DISC'00. Toledo, Spain*, volume LNCS 1914, pages 315–329, October 2000.

[PS98] F. Pedone and A. Schiper. Optimistic Atomic Broadcast. In S. Kutten, editor, *Proc. of 12th Distributed Computing Conference*, volume LNCS 1499, pages 318–332. Springer, September 1998.

[Ske81] D. Skeen. Nonblocking Commit Protocols. *ACM SIGMOD*, 1981.

[Ske82] D. Skeen. A Quorum-Based Commit Protocol. In *Proc. of the Works. on Distributed Data Management and Computer Networks*, pages 69–80, 1982.

[SR93] A. Schiper and A. Ricciardi. Virtually Synchronous Communication Based on a Weak failure Suspector. In *Proc. of FTCS-23*, pages 534–543, 1993.

[SS93] A. Schiper and A. Sandoz. Uniform Reliable Multicast in a Virtually Synchronous Environment. In *Proc. of ICDCS-13*, pages 561–568, 1993.

[VKCD99] R. Vitenberg, I. Keidar, G. V. Chockler, and D. Dolev. Group Communication Specifications: A Comprehensive Study. Technical Report CS99-31, Hebrew Univ., September 1999. To be published in ACM Comp. Surveys.