Optimistic Algorithms for Partial Database Replication[⋆]

Nicolas Schiper[1], Rodrigo Schmidt[2,1], and Fernando Pedone[1]

[1] University of Lugano, Switzerland
[2] EPFL, Switzerland

Abstract. In this paper, we study the problem of partial database replication. Numerous previous works have investigated database replication, however, most of them focus on full replication. We are here interested in genuine partial replication protocols, which require replicas to permanently store only information about data items they replicate. We define two properties to characterize partial replication. The first one, *Quasi-Genuine Partial Replication*, captures the above idea; the second one, *Non-Trivial Certification*, rules out solutions that would abort transactions unnecessarily in an attempt to ensure the first property. We also present two algorithms that extend the Database State Machine [8] to partial replication and guarantee the two aforementioned properties. Our algorithms compare favorably to existing solutions both in terms of number of messages and communication steps.

1 Introduction

Database replication protocols based on group communication have recently received a lot of attention [5, 6, 8, 13]. The main reason for this stems from the fact that group communication primitives offer adequate properties, namely agreement on the messages delivered and on their order, to implement synchronous database replication. Most of the complexity involved in synchronizing database replicas is handled by the group communication layer.

Previous work on group-communication-based database replication has focused mainly on full replication. However, full replication might not always be adequate. First, sites might not have enough disk or memory resources to fully replicate the database. Second, when access locality is observed, full replication is pointless. Third, full replication provides limited scalability since every update transaction should be executed by each replica. In this paper, we extend the Database State Machine (DBSM) [8],

[⋆] The work presented in this paper has been partially funded by the Hasler Foundation, Switzerland (project #1899) and SNSF, Switzerland (project #200021-107824).

a group-communication-based database replication technique, to partial replication. The DBSM is based on the deferred update replication model [1]. Transactions execute locally on one database site and their execution does not cause any interaction with other sites. Read-only transactions commit locally only; update transactions are atomically broadcast to all database sites at commit time for certification. The certification test ensures *one-copy serializability*: the execution of concurrent transactions on different replicas is equivalent to a serial execution on a single replica [1]. In order to execute the certification test, every database site keeps the *writesets* of committed transactions. The certification of a transaction T consists in checking that T's *readset* does not contain any outdated value, i.e., no committed transaction T' wrote a data item x after T read x.

A straightforward way of extending the DBSM to partial replication consists in executing the same certification test as before but having database sites only process update operations for data items they replicate. But as the certification test requires storing the writesets of all committed transactions, this strategy defeats the whole purpose of partial replication since replicas may store information related to data items they do not replicate. We would like to define a property that captures the legitimacy of a partial replication protocol. Ideally, sites should be involved only in the certification of transactions that read or write data items they replicate. Such a strict property, however, would force the use of an atomic multicast protocol as the group communication primitive to propagate transactions. Since existing multicast protocols are more expensive than broadcast ones [4], this property restricts the performance of the protocol. More generally, we let sites receive and momentarily store transactions unrelated to the data items they replicate as long as this information is shortly erased. Moreover, we want to make sure each transaction is handled by a site at most once. If sites are allowed to completely forget about past transactions, this constraint cannot obviously be satisfied. We capture these two requirements with the following property:

– *Quasi-Genuine Partial Replication:* For every submitted transaction T, correct database sites that do not replicate data items read or written by T permanently store not more than the identifier of T.[3]

Consider now the following modification to the DBSM, allowing it to ensure Quasi-Genuine Partial Replication. Besides atomically broadcasting transactions for certification, database sites periodically broad-

[3] Notice that even though transaction identifiers could theoretically be arbitrarily large, in practice, 4-byte identifiers are enough to uniquely represent 2^{32} transactions.

cast "garbage collection" messages. When a garbage collection message is delivered, a site deletes all the writesets of previously committed transactions. When a transaction is delivered for certification, if the site does not contain the writesets needed for its certification, the transaction is conservatively aborted. Since all sites deliver both transactions and garbage collection messages in the same order, they will all reach the same outcome after executing the certification test. This mechanism, however, may abort transactions that would be committed in the original DBSM. In order to rule out such solutions, we introduce the following property:

– *Non-Trivial Certification:* If there is a time after which no two conflicting transactions are submitted, then eventually transactions are not aborted by certification.

In this paper we present two algorithms for partial database replication that satisfy Quasi-Genuine Partial Replication and Non-Trivial Certification. Both algorithms make optimistic assumptions to ensure better performance. Our first algorithm is simpler and assumes *spontaneous total order*: with high probability messages sent to all servers in the cluster reach all destinations in the same order, a property usually verified in local-area networks. As a drawback, it processes a single transaction at a time. Our second algorithm is able to certify multiple transactions at a time and, as explained in Section 4, does not assume spontaneous total order.

To the best of our knowledge, [5] and [12] are the only papers addressing partial database replication using group communication primitives. In [5], every read operation is multicast to the sites replicating the data items read; write operations are multicast together with the transaction's commit request. A final atomic commit protocol ensures transaction atomicity. In [12], the authors extend the DBSM for partial replication by adding an extra atomic commit protocol. Each replica uses as its vote for atomic commit the result of the certification test. Both of our algorithms compare favorably to those presented in [5] and [12]: they either have a lower latency or make weaker assumptions about the underlying model, i.e., they do not require perfect failure detection.

2 System Model and Definitions

We consider a system $\Pi = \{s_1, .., s_n\}$ of database sites. Sites communicate through message passing and do not have access to a shared memory or a global clock. We assume the crash-stop failure model. A site that never crashes is *correct*, and a site that is not correct is *faulty*. The system is asynchronous, i.e., message delays and the time necessary to execute a step can be arbitrarily large but are finite. Furthermore, the communication channels do not corrupt or duplicate messages, and are (quasi-)reliable: if a correct site p sends a message m to a correct site q, then q eventually receives m.

Throughout the paper, we assume the existence of a *Reliable Broadcast* primitive. Reliable Broadcast is defined by primitives *R-bcast(m)* and *R-deliver(m)*, and satisfies the following properties [2]: if a correct site R-bcasts a message m, then it eventually R-delivers m (*validity*), (ii) if a correct site R-delivers a message m, then eventually all correct sites R-deliver m (*agreement*) and (iii) for every message m, every site R-delivers m at most once and only if it was previously R-bcast (*uniform integrity*). Reliable Broadcast does not ensure agreement on the message delivery order, that is, two broadcast messages might be delivered in different orders by two different sites. In local-area networks, some implementations of Reliable Broadcast can take advantage of network hardware characteristics to deliver messages in total order with high probability [9]. We call such a primitive Weak Ordering Reliable Broadcast, WOR-Broadcast.

Our algorithms also use a *consensus* abstraction. In the consensus problem, sites propose values and must reach agreement on the value decided. Consensus is defined by the primitives *propose(v)* and *decide(v)*, and satisfies the following properties: (i) every site decides at most once (*uniform integrity*), (ii) no two sites decide differently (*uniform agreement*), (iii) if a site decides v, then v was proposed by some site (*uniform validity*) and (iv) every correct site eventually decides (*termination*).

A database $\Gamma = \{x_1, .., x_n\}$ is a finite set of data items. Database sites have a partial copy of the database. For each site s_i, $Items(s_i) \subseteq \Gamma$ is defined as the set of data items replicated on s_i. A transaction is a sequence of read and write operations on data items followed by a commit or abort operation. For simplicity, we represent a transaction T as a tuple (id, rs, ws, up), where id is the unique identifier of T, rs is the readset of T, ws is the writeset of T and up contains the updates of T. More precisely, up is a set of tuples $(x,\ v)$, where, for each data item x in ws, v is the value written to x by T. For every transaction T, $Items(T)$ is defined as the set of data items read or written by T. Two transactions T and T' are said to be conflicting, if there exists a data item $x \in Items(T) \cap Items(T') \cap (T.ws \cup T'.ws)$. We define $Site(T)$ as the site on which T is executed. Furthermore, we assume that for every data item $x \in \Gamma$, there exists a correct site s_i which replicates x, i.e., $x \in \mathit{Items}(s_i)$. Finally, we define $\mathit{Replicas}(T)$ as the set of sites which replicate at least one data item written by T, i.e., $\mathit{Replicas}(T) = \{s_i \mid s_i \in \Pi \wedge \mathit{Items}(s_i) \cap T.ws \neq \emptyset\}$.

3 The Database State Machine Approach

We now present a generalization of the Database State Machine approach. The protocol in [8] is an instance of our generalization in the fully replicated context. For the sake of simplicity, we consider a replication model where a transaction T can only be executed on a site s_i if $\mathit{Items}(T) \subseteq \mathit{Items}(s_i)$. Moreover, to simplify the presentation, we consider a client c that sends requests on behalf of a transaction T to $\mathit{Site}(T)$. In the following, we comment on the states in which a transaction can be in the DBSM.

- *Executing:* Read and write operations are executed locally at $\mathit{Site}(T)$ according to the strict two-phase locking rule (strict 2PL). When c requests to commit T, it is immediately committed and passes to the Committed state if it is a read-only transaction, event which we denote $Committed(T)_{Site(T)}$; if T is an update transaction, it is submitted for certification and passes to the Submitted state at $\mathit{Site}(T)$. We represent this event as $Submitted(T)_{Site(T)}$. In the fully replicated case, to submit T, sites use an atomic broadcast primitive; in a partial replication context, the algorithms of Section 4 are used.
- *Submitted:* When T enters the Submitted state, its read locks are released at $\mathit{Site}(T)$ and T is eventually certified. With full replication, the certification happens when T is delivered; Section 4 explains when this happens in a partially replicated scenario. Certification ensures that if a committed transaction T' executed concurrently with T, and T read a data item written by T' then T is aborted. T' is concurrent with T if it committed at $Site(T)$ *after* T entered the Submitted state at $Site(T)$. Therefore, T passes the certification test on site s_i if for every T' already committed at s_i the following condition holds:

$$\begin{array}{c} Committed(T')_{Site(T)} \rightarrow Submitted(T)_{Site(T)} \\ \vee \\ T'.ws \cap T.rs = \emptyset, \end{array} \qquad (1)$$

  where $\rightarrow$ is Lamport's happened before relation on events [7].
  In the fully replicated DBSM, transactions are certified locally by each site upon delivery. In the partially replicated DBSM, to ensure *Quasi-*

Genuine Partial Replication, sites only store the writesets of committed transactions that wrote data items they replicate. Therefore, sites might not have enough information to decide on the outcome of all transactions. Hence, to satisfy *Non-trivial Certification*, we introduce a voting phase where each site sends the result of its certification test to the other sites. Site s_i can safely decide to commit or abort T when it has received votes from a *voting quorum* for T. Intuitively, a voting quorum VQ for T is a set of databases such that for each data item read by T, there is at least one database in VQ replicating this item. More formally, a quorum of sites is a voting quorum for T if it belongs to $VQS(T)$, defined as follows:

$$VQS(T) = \{VQ | VQ \subseteq \Pi \wedge T.rs \subseteq \bigcup_{s \in VQ} Items(s)\} \tag{2}$$

For T to commit, every site in a voting quorum for T has to vote *yes*. If one site in the quorum votes *no*, it means that T read an old value and should be aborted; committing T would make the execution non-serializable. Notice that $Site(T)$ is a voting quorum for T by itself, since for every transaction T, $Items(T) \subseteq Items(Site(T))$. If T passes the certification test at s_i, it requests the write locks for the data items it has updated. If there exists a transaction T' on s_i that holds conflicting locks with T's write locks, the action taken depends on T''s state on s_i and on T''s type, read-only or update:

1. *Executing:* If T' is in execution on s_i then one of two things will happen: if T' is a read-only transaction, T waits for T' to terminate; if T' is an update transaction, it is aborted.
2. *Submitted:* This happens if T' executed on s_i, already requested commit but was not committed yet. In this case, T's updates should be applied to the database before T''s. How this is ensured is implementation specific.[4]

Once the locks are granted, T applies its updates to the database and passes to the Committed state. If T fails the certification test, it passes to the Aborted state.

– *Committed/Aborted:* These are final states.

[4] For example, a very simple solution would be for s_i to abort T'; if T' later passes certification, its writes would be re-executed. The price paid for simplicity here is the double execution of T''s write operations.

4 Partially-replicated DBSM

In this section, we present two algorithms for the termination protocol of the DBSM in a partial replication context. These protocols ensure both one-copy serializability [1] and the following liveness property: if a correct site submits a transaction T, then either $Site(T)$ aborts T or eventually all correct sites in $Replicas(T)$ commit T. The algorithms also satisfy Quasi-Genuine Partial Replication and Non-Trivial Certification. The proof of correctness can be found in [11].

4.1 The "One-at-a-time" Algorithm

Sites execute a sequence of *steps*. In each step, sites decide on the outcome of one transaction. A step is composed of two phases, a consensus phase and a voting phase. Consensus is used to guarantee that sites agree on the commit order of transactions. In the voting phase, sites exchange the result of their certification test to ensure that the commit of a transaction T in step K induces a serializable execution.

The naive way to implement the termination protocol is to first use consensus to determine the next transaction T in the serial order and then execute the voting phase for T. We take a different approach: Based on the observation that with a high probability messages broadcast in a local-area network are received in total order [9], we overlap the consensus phase with the voting phase to save one communication step. If sites receive the transaction to be certified in the same order, they vote for the transaction before proposing it to consensus. Luckily, by the time consensus decides on a transaction T, every site will already have received the votes for T and will be able to decide on the outcome of T.

Algorithm 1 is composed of three concurrent tasks. Each line of the algorithm is executed atomically. The state transitions of transactions are specified in the right margin of lines 10, 28, and 30. Notice that the state transition happens after the corresponding line has been executed. Every transaction T is a tuple $(id, site, rs, ws, up, past, order)$. We added three fields to the definition of a transaction (c.f. Section 2), namely *site*, *past*, and *order*: *site* is the database site on which T is executed; *past* is the order of T's submission; and *order* is T's commit order. The algorithm also uses five global variables: K stores the *step* number; *UNDECIDED* and *DECIDED* are (ordered) sequences of, respectively, pending transactions and transactions for which the outcome is known; *COMMITTED* is the set of committed transactions; and the set *VOTES* stores the votes received, i.e., the results of the certification test. We use the operators

$\oplus$ and $\ominus$ for the concatenation and decomposition of sequences. Let *seq1* and *seq2* be two sequences of transactions. Then, $seq_1 \oplus seq_2$ is the sequence of transactions in *seq1* followed by all the transactions in *seq2*, and $seq_1 \ominus seq_2$ is the sequence of transactions in *seq1* that are not in *seq2*. Transactions are matched using their identifiers.

To take advantage of spontaneous total order, database sites use the WOR-Broadcast primitive to submit transactions (line 10). When no consensus instance is running and *UNDECIDED* is not empty, sites first execute the *Vote* procedure for T at the head of *UNDECIDED* (line 17) and then propose T (line 18). In the *Vote* procedure, T is certified and the result of the certification is sent in a message of type *VOTE*.

Notice that even though $Site(T)$ is a voting quorum for T by itself ($Items(T) \subseteq Items(Site(T))$), in the algorithm, all sites replicating a data item read by T vote. This is done to tolerate the crash of $Site(T)$. If only $Site(T)$ voted, the following undesirable scenario could happen: $Site(T)$ submits T and crashes just after executing line 10. Databases *WOR-Deliver* T, propose T and decide on T. In this execution, sites would wait forever at line 24, as $Site(T)$ crashed before voting for T.

Two further remarks concern the *Vote* procedure. First, to be able to certify transactions, we need to implement the precedence relation $\rightarrow$ between events. For two transactions T and T', this is done by comparing the value of their *past* and *order* fields. If $T.order < T'.past$, we are sure that T committed before T' was submitted, because K is incremented after transactions commit. Second, notice that *VOTE* messages contain the *step number* K in which T was certified. This information is necessary because a transaction can be certified in different steps and the result of the certification test in steps K and K' might be different. This is precisely why sites wait for *VOTE* messages coming from step number K at line 24. Moreover, even if sites receive votes from different voting quorums, they will agree on the outcome of the transaction. Intuitively, this holds because we only take into account *voting quorums* that voted in *step* K, therefore they consider the same sequence of committed transactions. Finally, by verifying that transactions T and T' are the same at line 20, sites check if the spontaneous total order holds. If it is not the case, sites need to vote for the transaction decided by consensus.

4.2 The "Many-at-a-time" Algorithm

The previous algorithm certifies transactions sequentially. Thus, if many transactions are submitted, an ever-growing chain of uncommitted transactions can be formed. Algorithm 2 solves that problem by allowing a

Algorithm 1 The "One-at-a-time" algorithm - Code of database site s

```
 1: Initialization
 2:   $K \leftarrow 1$, $UNDECIDED \leftarrow \epsilon$, $DECIDED \leftarrow \epsilon$, $COMMITTED \leftarrow \emptyset$, $VOTES \leftarrow \emptyset$

 3: function Certify($T$)
 4:   return $\forall (id, order, ws) \in COMMITTED \ : \ order < T.past \ \vee \ ws \ \cap \ T.rs = \emptyset$

 5: procedure Vote($T$)
 6:   if $T.rs \cap Items(s) \neq \emptyset$ then
 7:     send(VOTE, $T.id, K, Certify(T)$) to all $q$ in $Replicas(T)$

 8: To submit transaction $T$                                   {Task 1}
 9:   $T.past \leftarrow K$
10:   WOR-Broadcast(VOTE_REQ, $T$)              {Executing → Submitted}

11: When receive(VOTE, $T.id, K', vote$) from q               {Task 2}
12:   $VOTES \leftarrow VOTES \cup (T.id, q, K', vote)$

13: When WOR-Deliver(VOTE_REQ, $T$) $\wedge$ $T.id \notin DECIDED$   {Task 3}
14:   $UNDECIDED \leftarrow UNDECIDED \oplus T$

15: When $UNDECIDED \neq \epsilon$
16:   $T \leftarrow head(UNDECIDED)$
17:   Vote($T$)
18:   Propose($K, T$)
19:   wait until Decide($K, T'$)
20:   if $T'.id \neq T.id$ then Vote($T'$)
21:   $UNDECIDED \leftarrow UNDECIDED \ominus T'$
22:   $DECIDED \leftarrow DECIDED \oplus T'.id$
23:   if $T'.ws \cap Items(s) \neq \emptyset$ then
24:     wait until $\exists VQ \in VQS(T') \ : \ \forall q \in VQ \ : \ (T'.id, q, K, -) \in VOTES$
25:     if $\forall q \in VQ \ : \ (T'.id, q, K, yes) \in VOTES$ then
26:       $T'$.order $\leftarrow K$
27:       $COMMITTED \leftarrow COMMITTED \cup (T'.id, T'.order, T'.ws \cap Items(s))$
28:       commit $T'$                           {Submitted → Committed}
29:     else
30:       if $s = T'.site$ then abort $T'$         {Submitted → Aborted}
31:   $K \leftarrow K + 1$
32:   $VOTES \leftarrow \{(tid, q, K', v) \in VOTES \mid K' \geq K\}$
```

sequence of transactions to be proposed in consensus instances and by changing the certification test accordingly.

Algorithm 2 follows the same structure and uses the same global variables as Algorithm 1. The difference lies in Task 3 and the auxiliary procedures used. In the general case, when sites notice that there is a sequence of pending transactions that have not been committed or aborted ("$UNDECIDED \neq \epsilon$" at line 25), this sequence is voted for and proposed in consensus instance K (lines 26–27). In the *Vote* procedure, every pending transaction is certified considering only the previously committed transactions (lines 3–9). The results are gathered in a set and later sent to all sites that have data items updated by some transaction in the pending sequence (lines 10–12). The "$VOTES \neq \emptyset$" condition at line 25 is there for garbage collection purposes: it forces the proposal of empty sequences in case there are votes for undelivered vote requests (a possible situation due to failures that would violate *Quasi-Genuine Partial Replication*).

After the K-th instance of consensus has decided on a sequence SEQ of transactions (line 28), sites verify whether they have voted for all transactions in SEQ; if it is not the case, they vote for the sequence SEQ (lines 29–30). Then, sites replicating data items updated by one of the transactions in SEQ, sequentially certify all transactions in SEQ following their order (lines 35–45). The certification of transaction T is divided into two parts. First, T is certified considering the transactions committed in steps lower than K by taking into account the votes of a voting quorum (line 37). Second, sites certify T considering committed transactions that have been decided in the same consensus instance (line 38). This is done by gathering committed transactions in a set called $LCOMMIT$ and by verifying that there does not exist a transaction T' in this set that writes a data item read by T. If T passes both certifications and updates a data item in $Items(s)$, it is treated in exactly the same way as certified transactions in Algorithm 1 (lines 41–43).

Differently from Algorithm 1, Algorithm 2 does not rely on spontaneous total order. This is because sequences of transactions are used when voting and proposing values to a consensus instance, and the order of transactions in this sequence does not matter when it comes to voting. Recall that the vote phase in step K consists in independently certifying undecided transactions against transactions committed in previous steps (line 11 and function *Certify* at lines 3–9). This phase does not take into consideration conflicts within the sequence itself since they are solved after the consensus instance is decided. Nevertheless, votes are still optimistic in Algorithm 2 as they are sent before the consensus instance has decided on its outcome.

The optimistic assumption that allows a transaction T to be certified as soon as consensus instance K decides on a sequence containing T is that every member of at least one correct voting quorum VQ for T has voted for any sequence containing T before consensus instance K (line 26). Notice that the sequences considered by different members of VQ do not have to be the same, the only requirement is that they all contain T.

We could further relax the optimistic assumptions required at the price of having a higher number of VOTE messages. In the way both

Algorithm 2 The “Many-at-a-time” algorithm - Code of database site s

```
Initialization
  $K \leftarrow 1$, $UNDECIDED \leftarrow \epsilon$, $DECIDED \leftarrow \epsilon$, $COMMITTED \leftarrow \emptyset$, $VOTES \leftarrow \emptyset$

function Certify($SEQ$)
  $V \leftarrow \emptyset$
  for all $T \in SEQ$ do
    if $\forall (id, order, ws) \in COMMITTED : order < T.past \ \vee \ ws \ \cap \ T.rs = \emptyset$ then
      $V \leftarrow V \cup (T.id, yes)$
    else $V \leftarrow V \cup (T.id, no)$
  return $V$

procedure Vote($SEQ$)
  if $\exists T \in SEQ \ : \ T.rs \ \cap \ Items(s) \neq \emptyset$ then
    send (VOTE, Strip($SEQ$), $K$,Certify($SEQ$)) to $\{q \mid \exists T \in SEQ \ : \ q \in Replicas(T)\}$

function Strip($SEQ$)
  $RESULT \leftarrow \epsilon$
  for all $T \in SEQ$ in order do
    $RESULT \leftarrow RESULT \oplus T.id$
  return $RESULT$

To submit transaction $T$                                                  {Task 1}
  $T.past \leftarrow K$
  R-bcast (VOTE_REQ, $T$)                            $\{Executing \rightarrow Submitted\}$

When receive (VOTE, $IDSEQ, K', V$) from $q$                               {Task 2}
  $VOTES \leftarrow VOTES \cup (IDSEQ, q, K', V)$

When R-deliver (VOTE_REQ, $T$) $\wedge$ $T.id \notin DECIDED$                  {Task 3}
  $UNDECIDED \leftarrow UNDECIDED \ \oplus \ T$

When $UNDECIDED \neq \epsilon \vee VOTES \neq \emptyset$
  Vote($UNDECIDED$)
  Propose($K, UNDECIDED$)
  wait until Decide($K$,$SEQ$)
  if $\exists T \ : \ T \in SEQ \ \wedge \ T \notin UNDECIDED$ then
    Vote($SEQ$)
  $DECIDED \leftarrow DECIDED \ \oplus$ Strip($SEQ$)
  $UNDECIDED \leftarrow UNDECIDED \ \ominus \ SEQ$
  if $\exists T \in SEQ \ : \ T.ws \cap Items(s) \neq \emptyset$ then
    $LCOMMIT \leftarrow \emptyset$
    for all $T \in SEQ$ in order do
      wait until
          $\exists VQ \in VQS(T) : \forall q \in VQ : \exists (SEQ_q, q, K, V_q) \in VOTES \ : \ T \in SEQ_q$
      if $(\forall q \in VQ : \exists (SEQ_q, q, K, V_q) \in VOTES : T \in SEQ_q \wedge (T.id, yes) \in V_q)$
          $\wedge \ (\nexists T' \in LCOMMIT \ : \ T'.ws \ \cap \ T.rs \neq \emptyset)$ then
        $LCOMMIT \leftarrow LCOMMIT \cup \{T\}$
        if $T.ws \ \cap \ Items(s) \neq \emptyset$ then
          $T.order \leftarrow K$
          $COMMITTED \leftarrow COMMITTED \cup (T.id, T.order, T.ws \ \cap \ Items(s))$
          commit $T$                                 $\{Submitted \rightarrow Committed\}$
      else
        if $s = T.site$ then abort $T$                  $\{Submitted \rightarrow Aborted\}$
  $K \leftarrow K + 1$
  $VOTES \leftarrow \{(tid, q, K', v) \in VOTES \mid K' \geq K\}$
```

algorithms are described, sites vote for a transaction only before it is proposed to the next consensus instance (line 17 of Algorithm 1, line 26 of Algorithm 2). Consider a scenario where the vote request for a transaction T is delivered by a site s right after s has proposed transaction(s) to consensus. Site s will therefore have to wait until the instance finishes to send its vote concerning T. However, T's vote request might have been delivered earlier by some other site and might even have been proposed to the current instance of consensus. If that is the case, and T is part of the consensus decision, the optimistic assumptions will not hold and the protocols might need an extra message step to certify T. This problem can be avoided if sites are allowed to vote while solving a consensus instance. In our example scenario, site s would vote for T even though it has already voted for its consensus proposal. Both votes would then be received by other sites and they would be used to decide on the outcome of T. This optimization relieves the need for spontaneous total order in Algorithm 1 and relaxes even more the optimistic assumption of Algorithm 2. As a secondary effect, it reduces the average latency of transaction certification since votes are sent right after the vote request is received.

5 Related Work and Final Remarks

In this section we compare our algorithms to the related work and conclude the paper. We focus here on the related works satisfying *Quasi-Genuine Partial Replication.*

In [5] the authors propose a database replication protocol based on group multicast. Every read operation on data item x is multicast to the group replicating x; writes are multicast along with the commit request. The delivered operations are executed on the replicas using strict two-phase locking and results are sent back to the client. A final atomic commit protocol ensures transaction atomicity. In the atomic commit protocol, every group replicating a data item read or written by a transaction T sends its vote to a *coordinator* group, which collects the votes and sends the result back to all participating groups. The protocol ensures *Quasi-Genuine Partial Replication* because a transaction operation on data item x is only multicast to the group replicating x and the atomic commit protocol is executed among groups replicating data item read or written by the transaction. In [12] the authors extend the DBSM to partial replication. They use an optimistic atomic broadcast primitive and a variation of atomic commit, called resilient atomic commit. In contrast to atomic commit, resilient atomic commit may decide to commit a transaction even though some participants crash. When a transaction T is optimistically delivered, replicas certify T and execute a resilient atomic commit protocol using the result of the certification test as their vote. If the optimistic order of T corresponds to the final order, the protocol ends; otherwise when the final order is known, T is certified again and a second resilient atomic commit protocol is executed. The protocol ensures *Quasi-Genuine Partial Replication*, since only sites replicating data item written by T keep T in their committed transaction sequence.

We now compare the cost of the protocols in [5, 12] with the two algorithms presented in this paper. We compare the number of communication steps and the number of messages exchanged during the execution of a transaction T. To simplify the analysis, we assume that all messages have a delay of δ. We consider two cases, one where the algorithms' respective optimistic assumption hold and one where it does not (c.f. Section 4). In both cases, we consider the best achievable latency and the minimum number of messages exchanged, when neither failures nor failure suspicions occur, the most frequent case in practical settings. We first present in Figure 1 the cost of known algorithms used by the protocols compared in this section. Variable k is the total number of participants in the protocol.

Problem	steps	unicast msgs.	broadcast msgs.
Non-Uniform R. Broadcast (RBcast) [2]	1	$k(k-1)+1$	k
Uniform Consensus (Consensus) [10]	2	$2k(k-1)$	$2k$
Non-Blocking A. Commit (NBAC) [3] [5]	2	$2k(k-1)$	$2k$
Uniform A. Broadcast (ABcast) [2]	3	$3k(k-1)+1$	$3k$
Uniform A. Multicast (AMcast) [4]	4	$4k(k-1)+1$	$4k$

Fig. 1. Cost of different agreement problems

Figure 2 presents the cost of the different algorithms. To compute the cost of the execution of T, we consider that T consists of a read and a write operation on the same data item x. For all the protocols, we consider that d database sites replicate data item x and that n is the total number of database sites in the system.

In [5], one multicast is used to read x, d messages are sent containing the result of the read, one multicast is used to send the write along with the commit request and a final atomic commit protocol among d participants is executed. Notice that none of the optimistic assumptions

[5] This cost corresponds to the case where all participants spontaneously start the protocol. This assumption makes sense here because in [5] participants deliver a transaction's commit request before starting the atomic commit protocol.

Algorithm	steps	unicast msgs.	broadcast msgs.
[5]	11	$10d^2 - 9d + 2$	$11d$
[12]	3	$3n(n-1) + d(d-1) + 1$	$3n + d$
Algorithms 1 & 2	3	$3n(n-1) + d(d-1) + 1$	$3n + d$

(a) Optimistic assumption holds

Algorithm	steps	unicast msgs.	broadcast msgs.
[5]	11	$10d^2 - 9d + 2$	$11d$
[12]	4	$3n(n-1) + 2d(d-1) + 1$	$3n + 2d$
Algorithms 1 & 2	4	$3n(n-1) + 2d(d-1) + 1$	$3n + 2d$

(b) Optimistic assumption does not hold

Fig. 2. Comparison of the database replication protocols

assumed by the algorithms in this paper influence the cost of this protocol. In [12], the transaction is atomically broadcast and one communication step later it is optimistically delivered. A resilient atomic commit protocol is then executed among the d database sites. Resilient atomic commit is implemented in one communication step, in which all participants exchange their votes. To guarantee agreement on the outcome of a transaction, the implementation requires perfect failure detection, an assumption that we do not need in this paper. In the best-case scenario, i.e., spontaneous total order holds, the number of communication steps is equal to $max(2, steps(ABcast))$. If the optimistic order is not the final order of the transaction, another resilient atomic commit protocol is needed and therefore the number of communication steps becomes $steps(ABcast) + 1$.

For Algorithms 1 and 2, the cost is computed as follows. In the best-case scenario, the number of communication steps is equal to $steps(RBcast)$ $+\ max(steps(Consensus), steps(vote\ phase))$, where *vote phase* corresponds to d broadcast messages. If the algorithms' respective optimistic assumptions do not hold, after deciding on T in consensus, another *vote phase* has to take place and therefore the number of communication steps becomes $steps(RBcast) + steps(Consensus) + steps(vote\ phase)$. For simplicity, we assume that in this second *vote phase*, all participants vote, generating an extra $d(d-1)$ messages.

Considering latencies, Algorithms 1, 2, and [12] give the best results. However, to achieve such latency, [12] uses perfect failure detection. In terms of the number of messages generated, [5] is cheaper than the DBSM-based solutions if d is much smaller than x. Nonetheless, this protocol has a serious drawback: its number of communication steps highly depends on the number of read operations, as every read operation adds 5 message steps (4 for the multicast and 1 to send back the result). As a final remark, notice that in this analysis we consider the execution of only one transaction. The cost of the protocols might however change if we considered multiple transactions. In this scenario, the following observations can be made. First, even though Algorithms 1 and 2 have equal costs in Figure 2, the overhead might be higher for Algorithm 1 when multiple transactions are submitted. This stems from the fact that in Algorithm 2, the cost of running consensus might be shared among a set of transactions, therefore reducing the number of generated messages. Second, in [5, 12], each transaction requires a separate instance of atomic commit to decide on its outcome. In Algorithm 2, however, at most two voting phases are needed to decide on the outcome of the sequence of transactions decided in the same consensus instance. Therefore, the longer this sequence, the cheaper Algorithm 2 will be compared to [5, 12].

References

1. Philip A. Bernstein, Vassos Hadzilacos, and Nathan Goodman. *Concurrency Control and Recovery in Database Systems.* Addison-Wesley, 1987.
2. T. D. Chandra and S. Toueg. Unreliable failure detectors for reliable distributed systems. *Journal of the ACM*, 43(2):225–267, March 1996.
3. J. Gray and L. Lamport. Consensus on transaction commit. Technical Report MSR-TR-2003-96, Microsoft Research, 2004.
4. R. Guerraoui and A. Schiper. Total order multicast to multiple groups. In *Proceedings of the 17th International Conference on Distributed Computing Systems (ICDCS)*, pages 578–585, Baltimore, USA, May 1997.
5. Udo Fritzke Jr. and Philippe Ingels. Transactions on partially replicated data based on reliable and atomic multicasts. In *Proceedings of the 21st International Conference on Distributed Computing Systems (ICDCS)*, pages 284–291, 2001.
6. B. Kemme and G. Alonso. A suite of database replication protocols based on group communication primitives. In *Proceedings of the 18th International Conference on Distributed Computing Systems (ICDCS)*, pages 156–163, 1998.
7. L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, July 1978.
8. F. Pedone, R. Guerraoui, and A. Schiper. The database state machine approach. *Journal of Distributed and Parallel Databases and Technology*, 14(1):71–98, 2003.
9. F. Pedone, A. Schiper, P. Urban, and D. Cavin. Solving agreement problems with weak ordering oracles. In *Proceedings of the 4th European Dependable Computing Conference (EDCC)*, October 2002.
10. André Schiper. Early consensus in an asynchronous system with a weak failure detector. *Distributed Computing*, 10(3):149–157, 1997.
11. N. Schiper, R. Schmidt, and F. Pedone. Optimistic algorithms for partial database replication. Technical Report 2006, University of Lugano, 2006.
12. A. Sousa, F. Pedone, R. Oliveira, and F. Moura. Partial replication in the database state machine. In *Proceedings of the 1st International Symposium on Network Computing and Applications (NCA)*, October 2001.
13. I. Stanoi, D. Agrawal, and A. E. Abbadi. Using broadcast primitives in replicated databases. In *Proceedings of the 18th International Conference on Distributed Computing Systems (ICDCS)*, pages 148–155, 1998.