Atomic Updates of Replicated Data

Rachid Guerraoui Rui Oliveira André Schiper

Département d'Informatique
Ecole Polytechnique Fédérale de Lausanne
1015 Lausanne, Switzerland
e-mail: guerraoui@di.epfl.ch

Abstract. Although several replication strategies have been described and compared in the literature, very few work has been published on the underlying protocols needed to support these strategies. In fact, the classical transactional protocols that are usually assumed are not fault-tolerant, and thus create a window of vulnerability for the "fault-tolerant" strategies they intend to support. In this paper, we point out this issue for "quorum voting" replication strategies. We describe a fault-tolerant protocol that enables to adequately support these strategies. We present some performance figures showing that, in addition to its higher resilience, our protocol provides better performance than the other possible alternatives.

1 Introduction

When compared to hardware-based techniques, replication is considered as a software-based cheap way of ensuring fault-tolerance. Availability of data objects can be increased by replication. If an object is replicated on several sites, it can still be available despite failures. However, concurrent updates on different replicas of the same object may lead to inconsistent states. A replicated database is considered correct if it behaves as if each object has only one copy, as far as the user can tell. This property is called *one-copy equivalence* [1].

A great effort has been put on finding strategies that ensure *one-copy equivalence*. These strategies are usually classified by stressing their trade-off between consistency and availability, and their resilience to site failures [4]. However, when focusing on the strategies, the necessary underlying support protocols are usually overlooked. It is often implicitly assumed that the underlying protocol does not condition the characteristics of the strategy. Nevertheless, the fact that the strategy is tightly dependent on system services such as reliable communication, commit protocols, and concurrency control, contradicts this assumption [5].

In this paper, we focus on *quorum voting* replication strategies, which was shown to preserve availability and consistency despite unreliable failure detections [7]. For the sake of simplicity of the presentation, but without loss of generality, we will consider the simplest version of quorum voting strategies, that is, *majority voting with a single object and a static configuration.* We first point out the fact that the classical transactional protocols, usually assumed to support replication strategies, are blocking, i.e. are not fault-tolerant. Then we show that existing non-blocking alternatives, actually suffer from *unilateral abort* or *missing opportunities* drawbacks, i.e. abort transactions unnecessarily in the case of false failure suspicions.

The protocol we present, called *Replica Majority Commit (RMC)*, is intended to be used at the commit time of transactions updating replicated objects. RMC can be viewed as a variation of a decentralized *three phase commit* protocol [13], dedicated to update replicated objects. RMC preserves the fault-tolerance of the *majority voting* replication strategy, and exploits all commit opportunities of the transactions. Furthermore, as shown by the performance figures we present, RMC provides better performance then the other alternatives.

We give a precise characterization of the correctness of RMC by showing that safety is never violated, and liveness is ensured as long as (i) a majority of replica managers are correct, and (ii) the failure detector is of class $\diamond\mathcal{S}$ [2]. The assumption that a majority of replica managers are correct is necessary for any non-blocking commit protocols assuming unreliable failure detectors [2], and is also necessary for the liveness of the majority voting strategy itself. Failure detectors of class $\diamond\mathcal{S}$ are those which ensure that (1) eventually every process that crashes is suspected by all correct processes, and (2) eventually some correct process is never suspected by any correct process. Such failure detectors are unreliable in the sense that they can make an infinity of false failure suspicions. It has been shown that such failure detectors are the weakest ones that allow to solve consensus-like problems [3], e.g non-blocking atomic commitment.

The rest of the paper is organized as follows. The next section discusses related work and point out the characteristics of our RMC protocol. Section 3 describes our model of an asynchronous system augmented with unreliable failure detectors, and defines the problem to be solved in a precise way. In Section 4, we present the RMC protocol and we prove its correctness. In Section 5, we give some performance figures and compare RCM with the other possible alternatives. In Section 6, we discuss extensions of RMC to more general quorum voting replication strategies. Finally, Section 8 summarizes the main contributions of this paper. The appendix discusses the termination protocol used by our RMC protocol.

2 Related work and motivation

2.1 Replication strategies

A replication strategy can be viewed as a rule on the way data objects can be accessed, i.e. read and written. The *available* copy strategy ensures *one-copy equivalence* by a “read one/write all” rule: a read operation can be performed on any available copy while a write operation must be performed on all available copies [1]. Whenever a copy crashes, it is removed from the set of available copies. A major limitation of this strategy is that it is based on the assumption of reliable failure detectors (which never make false suspicions), and thus exclude the possibility of partitions. As shown in [1], unreliable failure detections can lead to the reading of old and inconsistent replicas. The *quorum voting* strategies overcome this limitation. The basic idea was initially introduced by Gifford [7]: it consists in assigning votes to every replica, and defining read quorums and write quorums, for each object, such that: (1) read quorums and write quorums intersect, and (2) two write quorums intersect.

In this paper we discuss the underlying protocol needed for updating data objects, according to a quorum voting strategy. As we will show in the following section (Section 2.2), choosing a protocol which preserves the fault-tolerance of the strategy, and exploits all the commit opportunities, is not straightforward. For the sake of simplicity of the presentation, we will consider the simplest version of quorum voting strategies, that is, the *majority voting strategy with a single object and a static configuration.* In other words, we assume that a single object is accessed, a quorum is simply defined by a majority (i.e. weights are all 1), and the set of replicas on which a majority must be reached is statically defined. In Section 6, we will consider more general models.

2.2 Underlying protocols

The blocking problem: Replication strategies, such as majority voting, usually assume an underlying transactional system. They rely on traditional protocols, such as *locking* to ensure serialisability, and *two phase commit* (2PC) to ensure failure atomicity [9]. A locking protocol introduces a window of vulnerability as it is not itself fault-tolerant. Indeed, a locking protocol can block all replica managers if the holder of the lock crashes. A way to circumvent the blocking issue of locking is to use an optimistic technique where concurrency control conflicts are resolved at commit time (within the atomic commitment protocol). That is, no locking is issued during a transaction execution. At the time of commit, if a replica manager had a concurrency control conflict, it refuses to commit the transaction. In this case, both the serialisability and failure atomicity of the transaction rely on the atomic commitment protocol. Nevertheless, the atomic commitment protocol that is generally used, namely 2PC, is not fault-tolerant, as it leads to block all replica managers, if the coordinator crashes while the replica managers are waiting for a decision [1].

The unilateral abort problem: An alternative non-blocking atomic commitment protocol is the three phase commit (3PC) [12]. It has been shown in [13, 8], that 3PC can be designed for systems with unreliable failure detectors and can be resilient to a minority of crash failures. Nevertheless, a careful look at 3PC reveals its limitation in the context of replication. Indeed, the *unilateral abort* aspect of 3PC leads a transaction to be aborted even if a single replica manager crashes. More precisely, 3PC leads a transaction to be aborted, if any replica crashes in the first step of the protocol. This is in contradiction with the aim of a majority voting strategy, which is to enable data object updates (i.e. commit), despite the crash of a minority of replica managers.

The missing opportunities problem: A way to circumvent this is to use nested transactions prior to the execution of 3PC [6]. In this model, access to replicas is done on-line using nested transactions. A nested transaction can abort whereas its parent transaction can commit. If a majority of nested transactions commit, the parent transaction can commit. The commit protocol involves then only the replicas on which nested transactions have suceeded. Moreover, it should involve a strict majority of the replicas otherwise the *unilateral abort* problem remains. Choosing a strict majority from the set of successful nested transactions introduces however a subtle problem of *missing opportunities*. If one of the chosen replicas crashes, the transaction is aborted even if any of the discarded replicas is correct. As an example, suppose an initial set of five replica managers $\{p_1, p_2, p_3, p_4, p_5\}$, the nested transactions on the first four replicas commit (p_5 aborts or crashes) and the commit protocol is started over, say $\{p_1, p_2, p_4\}$, as they are a strict majority of the initial set. If then one of the replica managers crashes, for instance p_4, the transaction aborts. Given that p_3 is correct, the opportunity of commit by choosing $\{p_1, p_2, p_3\}$ is missed. This phenomena can occur with unreliable failure detectors, since they can make false suspicions and afterwards reconsider them.

Basically, the intuitive idea underlying our RMC protocol, is that it enables to reach commit eventhough a minority of replica crashes. RMC avoids both the *unilateral abort* and the *missing opportunities* drawbacks of the alternatives discussed above. RMC can be viewed as a variation of a 3PC protocol (more precisely of a decentralized version of 3PC [13]), integrating nested transactions. This integration leads to use the same messages and the same communication steps for different purposes, and hence leads to better performances than the alternatives we have discussed above (see Section 5).

3 Model and problems

3.1 System Model

We consider a distributed system composed of a finite set of processes. Processes fail by crashing; we do not consider Byzantine failures. A *correct* process is a process that does not fail. The processes are completely connected through a set of channels. Communication is by message passing, *asynchronous* and *reliable*. Asynchrony means that there is no bound on communication delays. A reliable channel ensures that a message sent by a process p_i to a process p_j is eventually received by p_j, if p_i and p_j are correct. This assumption does not exclude link failures, if we require that any link failure is eventually repaired.

We assume that our asynchronous system is augmented with a failure detector [2]. A failure detector can be viewed as a distributed oracle. Each process has access to a local failure detector module. This module maintains a list of processes that it currently *suspects* to have crashed. We do not make any assumption on how a failure detector is implemented. It can for example simply be based on time-outs, and failure detector modules can consult each other before suspecting a process. Failure detectors are abstractly characterized by their *completeness* and *accuracy* properties. In the following, we consider failure detectors of the class $\diamond\mathcal{S}$ (*Eventually Strong*) [2]. These failure detectors guarantee (1) *strong completeness*, i.e eventually every process that crashes is permanently suspected by every correct process, and (2) *eventual weak accuracy*, i.e eventually some correct process is never suspected by any correct process. Notice that failure detectors of class $\diamond\mathcal{S}$ are unreliable as they can make mistakes, i.e false failure suspicions. The fundamental characteristic of $\diamond\mathcal{S}$ is that it is the *weakest* class of failure detectors that allow to solve consensus-like problems [3].

3.2 The majority voting replication strategy

In the majority voting replication strategy, to read or write a logical object, a transaction must access some majority of the replicas.

To ensure that each read returns the last written value (one-copy equivalence), every replica is assigned a version number which is incremented every time the replica is written. To read a logical object, a transaction accesses some majority of the replicas, chooses one with the highest version number, and returns the replica's value. To write a logical object, a transaction must also access a majority of replicas. To calculate the version of the write, the transaction chooses a replica with the highest version number and increments it. After the write, all replicas in the accessed majority are updated with this new version number.

The above version management is done on every write access and need to be embedded in the replica update protocol. Read accesses, on the other hand, have no effect on the update protocol. Due to the common need to always choose a replica with the highest version number, some optimizations can be envisioned for read/write transactions [1]. However this is not our goal here, and in the following, we concentrate solely on write-only transactions.

3.3 The update majority problem

Here we precisely define the problem to be solved when updating a replicated object according to majority voting strategy. We call this problem the *update majority* problem, and we show later that our RMC protocol (Section 4.4) solves exactly this problem.

The problem consists for a set of replica managers to agree on an outcome of a transaction. The outcome depends on the votes *(yes* or *no)* provided by the replica managers. Each vote reflects the ability of the replica manager to perform the updates. A replica manager will vote *no* if it detects a local concurrency control conflict. The outcome can be either *(commit,number)* or *abort*. The meaning of a *(commit,number)* outcome is that the replica managers have agreed to commit the transaction, and the new version number is *number*. This can be the case only if a majority of replica managers vote *yes*. The meaning of an *abort* outcome is that a majority of *yes* could not be obtained. This can be the case for example if some replica managers that vote *yes* are suspected

to have crashed. In order to exclude trivial situations where replica managers always decide *abort*, and to exclude *unilateral abort* and *missing opportunities* situations, we require the transaction to be committed if there is a majority of replica managers which vote *yes*, and do not suspect each others.
We define the update majority problem by the following properties:

1. *Uniform-agreement:* no two replica managers decide differently.
2. *Non-blocking:* every correct replica manager that starts the protocol eventually decides.
3. *Uniform-validity:* (i) the decision can be *(commit,number)* only if a majority of replica managers have voted *yes*, and *number* is the highest version number plus 1; (ii) the decision must be *(commit,number)*, if there is a majority of replica managers which vote *yes* and do not suspect each others.

The *uniform-agreement* property is a safety property, whereas the *non-blocking* property is a liveness property. The *uniform-validity* property can be viewed both as a liveness and as a safety property, as it defines conditions under which the replicas can/must be updated.

4 The Replica Majority Commit protocol

In this section we describe the *Replica Majority Commit* protocol, or simply *RMC*, and we show how it solves the update majority problem above. First we give a global overview of RMC (Section 4.1), then we describe it in detail (Sections 4.2 and 4.3), and finally we prove its correctness (Section 4.4).

4.1 Overview of RMC

The structure of RMC is similar to that of the *decentralized three phase commit (D3PC)* protocol of Skeen [13]. The first difference is in the way of deciding to commit or to abort a transaction: RMC requires a majority to vote *yes*, whereas *D3PC* requires all to vote *yes*. Hence no *unilateral abort* is permitted with RMC. The second difference is that RMC integrates the version number management of the majority voting replication strategy.

To reach *commit*, the RMC protocol needs three communication steps. Figure 1 illustrates these steps with the set of replica managers $\{p_1, p_2, p_3, p_4, p_5\}$ (as in the example of Section 2.2). For this simple description, we assume that (1) the coordinator is p_1, (2) replica managers p_4 and p_5 crash before sending their votes, and (3) the replica managers in $M = \{p_1, p_2, p_3\}$ (3.1) are correct, (3.2) they vote *yes*, and (3.3) they do not suspect each others.

In step 1, the coordinator p_1 asks all replica managers to vote. When receiving this "request to vote" message, every replica manager sends its vote, together with its version number, to all the replica managers. In step 2, every replica manager in M receives *yes* votes from all members of M, and sends a *commit* proposal to all replica managers. In step 3, every replica manager that receives a *commit* proposal from all members of M decides *commit* [1]. Note that, because

[1] Hence *commit* is decided despite the crash of p_4 and p_5.

we assume unreliable failure detectors, messages are always sent to all replica managers (including suspected ones). In figure 1 for example, the messages are also sent to p_4 and p_5.

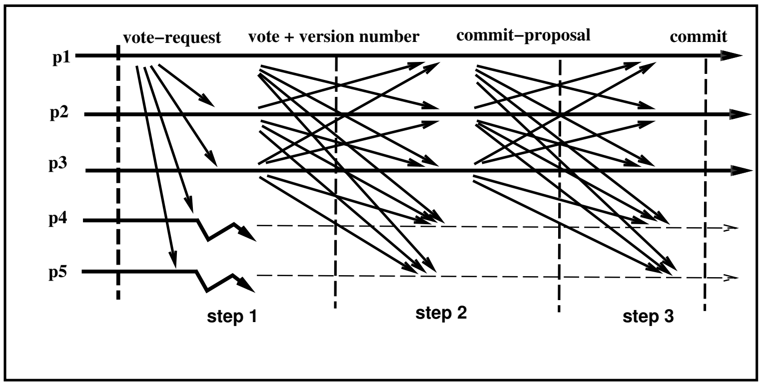

Fig. 1. Communication steps in RMC: coordinator is p_1

Negative votes and failure suspicions can prevent from reaching the *commit* decision. If a replica manager cannot collect *yes* votes from a majority in step 2, it invokes a termination protocol by proposing to abort the transaction. If a replica manager cannot collect *commit-proposals* from a majority in step 3, it invokes the termination protocol by proposing to commit the transaction. The termination protocol guarantees that replica managers agree on the same decision, which must be *commit* if a majority has proposed *commit*.

In the following, we describe the RMC protocol in more details. We assume the existence of a termination protocol invoked in the case of failure suspicions. We will discuss the termination protocol in Section 4.3.

4.2 Description of RMC

The detailed RMC protocol is given by the function *RMC()* in figure 2. This function is periodically called by every correct replica manager p_i, to receive and perform updates. The termination protocol is encapsulated inside the function *Termination()*, which takes as argument either $(commit, number)$ or *abort*, and returns either of these values. While executing the RMC protocol, every replica manager p_i is informed by its local failure detector module of crash suspicions: the notation $p_j \in \mathcal{D}_i$ (eg. line 9) indicates that process p_i suspects process p_j. Every replica manager p_i maintains a number, $number_i$, which indicates the version number of the replica handled by the replica manager. This number is updated whenever the replica manager updates a transaction. Finally, every replica manager p_i has a vote, $vote_i$, reflecting the ability of p_i to commit the transaction. We do not discuss here how the vote is determined, eg. the vote

```
function RMC()

cobegin
|| /* Task 1 */                                                          /* Step 1 */
    if p_i = coord then   /* Only the coordinator executes: */
        send(coord, step1, voteRequest) to all replica managers ;

    when [received(coord, step1, voteRequest)]   /* Starting the protocol */
        send(coord, step1, voteRequest) to all replica managers ;
        send(p_i, step2, (vote_i, number_i)) to all replica managers ;

    for k = 1 to ⌈(n+1)/2⌉                                               /* Step 2 */
        wait until [for some p_j: received(p_j, step2, (vote_j, number_j))] ;
    for k = 1 to ⌈(n-1)/2⌉
        wait until [for some p_j: received(p_j, step2, (vote_j, number_j)) or p_j in D_{p_i}] ;
    if (for ⌈(n+1)/2⌉ replica managers p_j: received(p_j, step2, (yes, number_j))) then
        number := largest number_j such that received(p_j, step2, (yes, number_j)) ;
        send(p_i, step3, (commit, number)) to all replica managers ;
    else
        send(p_i, startCons) to all replica managers ;
        outcome := Termination(abort) ;   /* Deciding through the termination protocol */
        if outcome = (commit, number) then number_i := number ;
        return outcome ;   /* End of the protocol */

    for k = 1 to ⌈(n+1)/2⌉                                               /* Step 3 */
        wait until [for some p_j: received(p_j, step3, (commit, number)) or received(p_j, startCons)] ;
    for k = 1 to ⌈(n-1)/2⌉
        wait until [for some p_j: received(p_j, step3, (commit, number)) or received(p_j, startCons) or p_j in D_i] ;
    if (for ⌈(n+1)/2⌉ replica managers p_j: received(p_j, step3, (commit, number))) then
        send(p_i, decide, (commit, number)) to all replica managers ;
        number_i := number ;
        return (commit, number) ;   /* End of the protocol */
    send(p_i, startCons) to all replica managers ;
    outcome := Termination((commit,number)) ;   /* Deciding through the termination protocol */
    if outcome = (commit, number) then number_i := number ;
    return outcome ;   /* End of the protocol */

|| /* Task 2 */
    wait until [for some p_j: received(p_j, decide, outcome)] ;
    send(p_i, decide, outcome) to all ;
    if outcome = (commit, number) then number_i := number ;
    return outcome ;   /* End of the protocol */
coend
```

Fig. 2. The RMC protocol

can be *"no"* if the replica manager has detected a concurrency control conflict, and *"yes"* otherwise.

The function *RMC()* triggers two concurrent tasks, noted *Task 1* (lines 1 to 29) and *Task 2* (lines 30 to 33), and terminates by the execution of a "**return** v" statement, where v is either $(commit, number)$ or $abort$ (lines 17, 25, 29, and 33). When p_i executes **return** v, it *decides* on the outcome v of the transaction.

- *Task 2* is a decision task. It waits for a decision message $(p_j, decide, outcome)$, and forwards the *outcome* to the other replica managers (line 31). This ensures that if one correct replica manager decides on an outcome, then all the correct replica managers also decide on the outcome.

- *Task 1* executes the main protocol, i.e. step 1, step 2 and step 3 below.

  1. **Step 1** (lines 1-5): At line 2, the coordinator sends the *voteRequest* message to all. From there on, the protocol is decentralized: there is no coordinator. A replica manager p_i that receives the message *(coord,step1,voteRequest)* starts the protocol and forwards this message to all. This ensures that if one correct replica manager starts the protocol, every correct replica manager also starts the protocol (despite the failure of the coordinator). Then p_i sends its vote, together with its version number, to all replica managers, and moves to step 2 (line 6).
  2. **Step 2** (lines 6-17): Every replica manager p_i that starts step 2, waits until (i) it receives votes from $\lceil (n+1)/2 \rceil$ participants (i.e. a majority), and (ii) it receives *votes* from, or it suspects, each of the rest ($\lceil (n-1)/2 \rceil$) of the replica managers (lines 6-9). (1) If p_i has not received a majority of *yes* votes, p_i sends a "$startconsensus$" notification to all and starts the termination protocol by invoking the *Termination* function with *abort* as initial value. The value returned by *Termination* is the value decided by p_i. (2) If p_i has received a majority of *yes* votes, p_i defines the new version number as the maximum of the numbers received, sends this number together with a *commit* notification to all (line 12), and moves to step 3 (line 18).
  3. **Step 3** (lines 18-29): Every replica manager p_i that starts step 3, waits until: (i) it receives *commit* proposals or "$startconsensus$" notifications from $\lceil (n+1)/2 \rceil$ participants, and (ii) it receives a *commit* proposal or a "$startconsensus$" notification from, or it suspects, each of the rest ($\lceil (n-1)/2 \rceil$) of the replica managers. (1) If p_i has not received a majority of *commit* proposals, p_i sends a "$startconsensus$" notification to all and starts the termination protocol by invoking the *Termination* function with $(commit, number)$ as initial value. The value returned by *Termination* is the value decided by p_i. (2) If p_i has received a majority of *commit* proposals, p_i informs all about its *commit* decision, updates its version number and decides $(commit, number)$.

4.3 The termination protocol

Up to now, we have assumed the existence of a termination protocol which is used to handle "bad terminations". In this section we discuss such a protocol. The problem that must be solved by the termination protocol is the *majority consensus* problem which we have introduced in [8]. This problem is actually a variation of the well known *uniform consensus* problem [2].

In both problems, a set of processes (resp. replica managers) propose, each an initial binary value 0 or 1 (resp. *abort* or $(commit, number)$) , and must decide on a common final binary value 0 or 1 (resp. *abort* or $(commit, number)$). The uniform consensus problem is defined by the following properties: (1) *agreement*: no two processes decide differently; (2) *non-blocking*: every correct process eventually decides; (3) *uniform-validity*: the value decided must be one of the initial values proposed; The majority consensus problem is defined by (1) the *agreement* and (2) the *non-blocking* properties of uniform consensus, and also the following (3) *majority-validity* property: (i) 1 can only be decided if at least one process has proposed 1 as initial value, and (ii) 0 can only be decided if at least a majority of the processes have proposed 0 as initial value.

Rather than presenting a complete majority consensus protocol, we adopt a modular approach and we show how to obtain such a protocol from any uniform consensus protocol. For space limitation reasons, the algorithm that transforms any uniform consensus protocol into a majority consensus protocol is given in the appendix. The transformation can thus be applied for example to the centralized uniform consensus protocol of Chandra and Toueg described in [2]. The Chandra-Toueg protocol solves uniform consensus, given that there is a majority of correct processes and the failure detector is of class $\diamond\mathcal{S}$ (i.e. exactly under our assumptions).

We thus assume in the following that the termination protocol used in RMC is a majority consensus protocol. Hence the termination protocol, invoked through function *Termination* in figure 2, satisfies the *agreement*, *non-blocking*, and *majority-validity* properties above.

4.4 Correctness of RMC

We show in the following that our RMC protocol solves the update majority problem (defined in Section 3.3), assuming any failure detector of class $\diamond\mathcal{S}$, and a majority of replica managers are correct. We introduce a lemma for each property of the update majority problem.

Lemma 1. *(uniform-agreement) No two replica managers decide differently.*

PROOF. A replica manager decides *abort* (resp. *(commit,number)*) in Task 2 only if some replica manager has decided *abort* (resp. *(commit,number)*) in Task 1 (either at line 17, 25 or 29). We show in the following that if two replica managers decide in Task 1, they decide the same outcome. Assume that some replica manager p_i decides *(commit,number)* at line 25. This means that p_i has received a majority of *(commit,number)* messages (line 22), which implies that less than a half of the replica managers have started *Termination* with *abort* as initial value (line 15). By the *majority-validity* property of *Termination*, no replica manager can thus decide *abort* (either at line 17 or 29). If no replica manager decides at line 25, then every replica manager that decides, decides through *Termination* (either at line 17 or 29). By the *agreement* property of *Termination*, no two replica managers decide differently. □

Lemma 2. *(non-blocking) If a majority of replica managers are correct, and the failure detector is of class $\diamond\mathcal{S}$, then every replica manager that starts the protocol eventually decides.*

PROOF. Assume that some correct replica manager starts the protocol. By the reliable channel assumption, if any correct replica manager receives the *voteRequest* message at line 3 (i.e starts the protocol), every correct replica manager also receives the *voteRequest* message (and starts the protocol). Also by the reliable channel assumption, if some correct replica manager decides either at line 25, or in Task 2, every correct replica manager eventually receives the decision message and decides (in Task 2). Assume in the following that no correct replica manager decides either at line 25, or in Task 2. By (1) the assumption of a majority of correct replica managers, (2) the *strong completeness* property of $\diamond\mathcal{S}$, and (3) the assumption of reliable channels, no correct replica manager remains blocked forever either at line 7 or 9. Hence every replica manager either starts *Termination* (line 15) or proceeds to step 3. In a similar way, by (1) the assumption of a majority of correct replica managers, (2) the *strong completeness* property of $\diamond\mathcal{S}$, and (3) the assumption of reliable channels, no correct replica manager remains blocked forever either at line 19 or 21. As we assume that no replica manager decides at line 25, thus every correct replica manager starts *Termination*. By the *non-blocking* property of *Termination*, every correct replica manager eventually decides. □

Lemma 3. *(uniform-Validity) (i) the decision can be (commit,number) only if a majority of replica managers have voted yes, and number is the highest version number plus* 1*; (ii) the decision must be (commit,number), if there is a majority of replica managers which vote yes and do not suspect each others.*

PROOF. (i) A replica manager can decide *(commit,number)* either in Task 1 or in Task 2. To decide *(commit,number)* in Task 2, at least one replica manager must have decided *(commit,number)* in Task 1. Let p_i be a replica manager that decides *(commit,number)* in Task 1: p_i can do so either at line 17, 25, or 29. To decide *(commit,number)* at line 17, p_i must have got *(commit,number)* as the result of *Termination* at line 15. By the *majority-validity* property of *Termination*, some participant p_j must have started *Termination* with *commit* as its initial value, i.e at line 27. For any p_i to reach any of the lines 25, 27 or 29, p_i must have received a *yes* vote from a majority or replica managers (line 10). By construction, if *(commit,number)* is decided, then *number* is the higher version number plus 1. (ii) A replica manager cannot reach any decision unless a majority of replica managers have started the protocol. As a majority of replica managers are correct, then at least some correct replica manager must have started the protocol. This implies that all correct replica managers also start the protocol. As a majority of correct replica managers vote *yes* and are not suspected, then these replica managers send *yes* messages to all at line 12. Consequently, at least some correct replica manager receives these $(commit, number)$ messages at line line 22, sends its decision to all and decides $(commit, number)$. Hence (by Lemma 1), every correct replica manager decides $(commit, number)$ and no replica manager decides *abort*. □

By Lemma 1, Lemma 2, and Lemma 3, we have:

Proposition. *If there is majority of correct replica managers, and the failure detector is of class $\diamond\mathcal{S}$, then the RMC protocol (figure 2) solves the update majority problem.*

5 Evaluation of RMC

In this section, we compare the performances of RMC with other alternatives: that is 2PC (two phase commit) [9], 3PC (three phase commit) [12] and D3PC (decentralized three phase commit) [13], considering a system with n replica managers. The 2PC algorithm is used as a reference but cannot really be viewed as an alternative as it is not fault-tolerant. We do not consider here OD-3PC (optimized decentralized three phase commit) [8] as it actually is an optimization of D3PC, and the optimization could also be applied to our RMC protocol. For 2PC, 3PC and D3PC, we also consider their extensions with nested transactions [6].

We compare the protocols only for the case where *commit* is directly decided, i.e. without passing through the termination protocol. This corresponds to the most frequent case. The table in figure 3 recalls the resilience characteristics of each protocol (Section 2), and shows, for each protocol, the number of communication steps and the number of messages, required to reach commit. To reach commit, 2PC, D3PC, and RMC, require 3 communication steps, whereas 3PC needs 5 communication steps. When nested transactions are used, two additional communication steps are required. When establishing the number of messages, we distinguish two cases: (a) using point-to-point communication, and (b) using broadcast communication (broadcast network). For instance, with point-to-point communications, $2PC$ requires $3n$ messages, and when used together with nested transactions $5n$ messages are required.

The tables in figure 4 compare the performances of the protocols, (a) using point-to-point communication, and (b) using broadcast communication. The measures have been obtained on FDDI (100Mb/s), with SPARC20 machines (running Solaris 2.3), using the UDP transport protocol. Different replica managers were located on different machines.

Protocol	Resilience	Number of steps	Number of messages point–to–point communication	Number of messages broadcast communication
2PC	*blocking unilateral abort*	*3*	*3n*	*n+2*
2PC + N	*blocking missing opportunities*	*5*	*5n*	*2n+3*
3PC	*non–blocking unilateral abort*	*5*	*5n*	*2n+3*
3PC + N	*non–blocking missing opportunities*	*7*	*7n*	*3n+4*
D3PC	*non–blocking unilateral abort*	*3*	*n(2n+1)*	*2n+1*
D3PC + N	*non–blocking missing opportunities*	*5*	*n(2n+3)*	*3n+2*
RMC	*non–blocking*	*3*	*n(2n+1)*	*2n+1*

Fig. 3. Steps and messages for direct *commit*

The measures convey an average time to commit a transaction. More precisely, every replica manager was, in its turn, the initiator of a transaction. Whenever a replica manager commits a transaction and is the new coordinator, it starts a new RMC protocol. The figures show that both with point to point communications and with broadcast communications, RMC provides better performances than other non-blocking alternatives. Furthermore, with broadcast communications, the price to pay for having a non-blocking protocol like RMC is not very high, when compared to 2PC (which is blocking). The reason why RMC provides sensibly better performances than D3PC, although the same number of communication steps and messages are required, is that with RMC, a replica manager needs only a majority of *yes* votes to move to step 3, whereas in D3PC, all *yes* votes must be received. Hence, beside the fact that RMC does not suffer from the unilateral abort and missing opportunities drawbacks of other protocols, RMC provides better performances.

6 Extensions of RMC

For the sake of clarity of presentation and better focus on the more relevant issues, we have considered some model simplifications in the protocol. More precisely, we considered the simplest version of quorum consensus strategies, that is, the majority strategy with static configurations. Furthermore, we considered transactions accessing single objects. In the following we show how RMC can be easily extended to cover a more general model.

(a) point–to–point communication

Protocol	3 managers	6 managers	9 managers	12 managers
2PC	4.4 ms. ± 0.0	6.5 ms. ± 0.0	10.1 ms. ± 0.1	12 ms. ± 0.1
2PC + N	7 ms. ± 0.0	11 ms. ± 0.0	18 ms. ± 0.1	20 ms. ± 0.1
3PC	7.5 ms. ± 0.0	11.4 ms. ± 0.0	17.7 ms. ± 0.1	21.2 ms. ± 0.2
3PC + N	9 ms. ± 0.0	14 ms. ± 0.0	21 ms. ± 0.2	25 ms. ± 0.2
D3PC	5.4 ms. ± 0.0	10 ms. ± 0.0	17 ms. ± 0.1	22 ms. ± 0.2
D3PC + N	8.5 ms. ± 0.0	13 ms. ± 0.0	20 ms. ± 0.2	26 ms. ± 0.2
RMC	5.4 ms. ± 0.0	10 ms. ± 0.0	16 ms. ± 0.0	20 ms. ± 0.1

(b) broadcast communication

Protocol	3 managers	6 managers	9 managers	12 managers
2PC	5ms ± 0.0	7.5 ms. ± 0.0	10 ms. ± 0.1	12.3 ms. ± 0.1
2PC + N	8ms. ± 0.0	12 ms. ± 0.0	16ms . ± 0.1	15 ms. ± 0.1
3PC	8ms. ± 0.0	11.8 ms ± 0.0	16 ms. ± 0.1	21 ms. ± 0.2
3PC + N	10ms. ± 0.0	14 ms. ± 0.0	19ms. ± 0.2	25ms. ± 0.2
D3PC	5.2 ms. ± 0.0	8 ms. ± 0.0	12 ms. ± 0.1	17.5 ms. ± 0.2
D3PC + N	7ms. ± 0.0	10ms. ± 0.0	15ms. ± 0.2	20ms. ± 0.2
RMC	5.2 ms. ± 0.0	8 ms. ± 0.0	11 ms. ± 0.0	17 ms. ± 0.1

Fig. 4. Time to commit a transaction.

6.1 Quorum consensus

In a general quorum strategy [7], quorum conditions are (1) read quorums and write quorums intersect, and (2) two write quorums intersect. A strict majority is the smallest permitted write quorum. Read availability can be increased by reducing the size of read quorums. Our protocol can be easily tuned to accommodate different quorums. To do this, the required number of "good" messages at decision stages of the algorithm must be changed accordingly. Also the *majority-validity* property of our *Termination* protocol need to take the write quorum size in consideration. The size of the write quorum establishes a trade-off with the resilience of the protocol. Assuming a set of n physical replicas of an object the resilience is given by $n - WriteQuorum$. This trade-off availability with resilience of the protocol.

6.2 Dynamic configuration

With a static configuration, replica crashes compromise the liveness of the system. To cope with this problem and maintain availability dynamic adjust of quorums is usually adopted [10]. To ensure consistency, it is required that any update quorum includes a majority of the replicas from the last one.

Since each quorum depends on the set of replicas that participated in the last commit, the system must keep track of this set. To achieve this, along with its version number, each replica also holds the set of replicas that composed its versioned update. In the algorithm this set can be recorded at lines 16 and 28 as an outcome of *Termination* or at lines 24 and 32 as part of the $(p_i, decide, (commit, number))$ message. When deciding on a majority (lines 10 and 22) the condition is strengthened to ensure that a majority of the highest versioned quorum is present. The *majority-validity* property of *Termination* does not need to be changed. Enlarging the number of replicas does not arise problems providing that the new replica is updated through a read quorum.

6.3 Multiple objects

In order to handle multiple-object transactions, the algorithm must involve all the logical objects accessed by the transaction. This means that to commit, the transaction has to aggregate in a logical *and*, the commitment of each object's majority of replicas. The coordinator, for each accessed object, requests its replicas to vote and thus to start RMC. The outcome of RMC can be commit only if for every object's replica set the outcome is commit. In the algorithm of figure 2 this can be obtained by: in line 5 broadcast a vote from the replica of each involved object, waiting for the votes for each object by repeating the loops of lines 6 to 9 and 18 to 21; both decisions at lines 10 and 22 logically *and* the majorities for each object, and *Termination* is invoked for each one of the objects (lines 15 and 27) returning the logical *and* of their outcomes.

7 Concluding remarks

To ensure mutual consistency replica management must prevent concurrent updates of the replicas. Ensuring this the replica strategy is said to preserve *one-copy equivalence*. Quorum strategies are appealing for their trade-off between availability and resilience to failures. However, the proposals of replica update strategies often overlook the necessary underlying protocols.

The contribution of this paper is two-fold. First, we pointed out the inadequacy of traditional atomic commitment protocols. We argued on their intrinsic lack of fault-tolerance compromising the supported strategies. Second, we presented the fault-tolerant RMC protocol, which can be used with a simple transaction model and can be viewed as integrating nested transactions inside an atomic commitment protocol. RMC ensures the promised resilience of quorum strategies and still provides performant figures. The protocol relies on a canonical asynchronous system augmented with unreliable failure detectors. An interesting characteristic of the protocol is that it "accommodates" wrong suspicions therefore exploiting commitment opportunities.

With the adequate adaptations, RMC is clearly suited for any optimized quorum strategy (eg. voting with witnesses [11] or voting with ghosts [14]).

References

1. P.A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems.* Addison Wesley. 1987.
2. T. Chandra and S. Toueg. Unreliable failure detectors for reliable distributed systems. Technical Report 94, Department of Computer Science, Cornell Univ, 1994. A preliminary version appeared in *Proceedings of the 10th ACM Symposium on Principles of Distributed Computing*, pp 325-340. ACM Press. August 1991.
3. T. Chandra, V. Hadzilacos and S. Toueg. The weakest failure detector for solving consensus. In *Proceedings of the 11th ACM Symposium on Principles of Distributed Computing*, pp 147-159. ACM Press. August 1992.
4. S. Ceri, M. Houstsma, A. Keller and P. Samarati. A Classification of Update Methods for Replicated Databases. Technical Report CS-TR-91-1392, Department of Computer Science, Stanford Univ, 1991.
5. S-W. Chen and C. Pu. A Structural Classification of Integrated Replica Control Mechanisms. Technical Report CUCS-006-92, Department of Computer Science, Columbia University, 1992.
6. K. Goldman and N. Lynch. Quorum Consensus in Nested Transaction Systems. In *ACM Transactions on Database Systems* 19 (4), pp 537-587. December 1994.
7. D. K. Gifford. Wheighted Voting for Replicated Data. In *Proceeding of the 7th ACM Symposium on Operating System Principles*, pp 150-161. ACM Press. 1979.
8. R. Guerraoui, M. Larrea and A. Schiper. Reducing the Cost of Non-Blocking in Atomic Commitment. To appear in *Proceedings of the 16th IEEE Conference on Distributed Computing Systems.* Hong Kong, May 1996.
9. J. Gray. Notes on Database Operating Systems. In *Operating Systems: An Advanced Course*, pp 10-17. Lecture Notes in Computer Science (60), Springer Verlag. 1978.
10. S. Jajodia and D. Mutchler. Dynamic voting algorithms for maintainning the consistency of replicated databases. In *ACM Transactions on Database Systems* 15, pp 230-280. June 1990.
11. J.-F. Pâris. Voting with Witnesses: A Consistency Scheme for Replicated Files. In *Proceedings of the 6th International IEEE Conference on Distributed Computing Systems*, pp 606-612. 1986.
12. D. Skeen. NonBlocking Commit Protocols. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pp 133-142. ACM Press. 1981.
13. D. Skeen. A Quorum-Based Commit Protocol. In *Proceedings of the Berkeley Workshop on Distributed Data Management and Computer Networks* (6), pp 69-80. February 1982.
14. R. van Renesse and A. Tanenbaum. Voting with Ghosts. In *Proceedings of the 8th International IEEE Conference on Distributed Computing Systems*, pp 456-462. 1988.

Appendix: the termination protocol

In the following, we present a simple algorithm that transforms any uniform consensus protocol into a majority consensus protocol, to be used as a termination for our RMC protocol. The transformation below can thus be applied for example to the Chandra-Toueg protocol described in [2], which solves uniform consensus, given that there is a majority of correct processes and the failure detector is of class $\diamond\mathcal{S}$.

Consider a function $UnifConsensus()$ that solves the uniform consensus problem. This function, which can be called by every process p_i, takes a binary value as argument (the initial value), and returns a binary value (the decision value of uniform consensus). Using $UnifConsensus()$, a solution to the majority consensus problem can be obtained as follows:

1. Every replica manager p_i broadcasts its initial value to all replica managers;
2. If the initial value of p_i is 1, p_i calls $UnifConsensus(1)$ and decides on the returned value;
3. If the initial value of p_i is 0, p_i waits until it receives initial values from a majority of replica managers. If one of these values is 1, p_i calls $UnifConsensus(1)$ and decides on the returned value. If all received values are 0, p_i calls $UnifConsensus(0)$ and decides on the returned value.

Proposition. *If there is a majority of correct processes, then any uniform consensus protocol, together with the above transformation, solves solves the majority consensus problem.*

PROOF. The *uniform-agreement* and *non-blocking* properties of majority consensus are trivially ensured by the *uniform-agreement* and *non-blocking* properties of uniform consensus. The *majority-validity* property of majority consensus is satisfied since, (i) by the *uniform-validity* property of uniform consensus, 1 can only be decided if 1 has been proposed, and (ii) if less than a majority of initial values are 0, then every process will call $UnifConsensus(1)$. By the *uniform-validity* property of uniform consensus, no correct process decides 0. □