A Primary-Backup Protocol for In-Memory Database Replication*

Lásaro Camargos[‡,⋆] Fernando Pedone[⋆] Rodrigo Schmidt[†,⋆]

[⋆]University of Lugano (USI), Switzerland
[†]École Polytechnique Fédérale de Lausanne (EPFL), Switzerland
[‡]Universidade Estadual de Campinas (Unicamp), Brazil

Abstract

The paper presents a primary-backup protocol to manage replicated in-memory database systems (IMDBs). The protocol exploits two features of IMDBs: coarse-grain concurrency control and deferred disk writes. Primary crashes are quickly detected by backups and a new primary is elected whenever the current one is suspected to have failed. False failure suspicions are tolerated and never lead to incorrect behavior. The protocol uses a consensus-like algorithm tailor-made for our replication environment. Under normal circumstances (i.e., no failures or false suspicions), transactions can be committed after two communication steps, as seen by the applications. Performance experiments have shown that the protocol has very low overhead and scales linearly with the number of replicas.

Keywords: *primary-backup replication, in-memory databases, agreement protocols.*

1. Introduction

Demand for high performance combined with plummeting hardware prices have led to the widespread emergence of large computing clusters [24, 27]. Applications in these environments often rely on shared storage systems, accessible to the application processes running on remote servers. Storage systems are used to keep shared information, managed concurrently by different application processes, as well as to provide fault tolerance by allowing processes to save their state and later retrieve it for recovery and migration. Whatever the use, high-availability, good performance, and strong consistency are key requirements of a storage service. In such environments, in-memory databases (IMDBs) [10] have been successfully used to increase the performance of short transactions [4]. This paper shows how high-availability can be introduced in a replicated IMDB setting with very small overhead—in our experimental setup, as small as 3% for managing 3 replicas and 14% for managing 8 replicas.

IMDBs provide high transaction throughput and low response time by avoiding disk I/O. The key characteristic of an IMDB is that the database resides in the server's main memory—virtual memory can also be used, but maximum performance is achieved when data fits the server's physical memory. Since transactions do not have to wait for data to be fetched from disk, concurrency becomes less important for performance and some IMDBs rely on coarse-grain concurrency control such as *multiple-readers single-writer*[4]. Read-only transactions can execute concurrently, but update transactions are serialized. Executing update transactions serially has several advantages: first, serializability is trivially guaranteed; second, the gain in performance from not having to deal with synchronization mechanisms (e.g. locks, timestamps) usually overcomes the loss in concurrency; and third, deadlocks in an IMDB cannot happen.

Although IMDBs rely on main memory only for transaction execution, a transaction log must be kept on disk for recovery. Read-only transactions execute in main memory only; update transactions have to log information on disk before committing. In fact, storing information on disk is the main overhead of update transactions executing in an IMDB. To improve performance, disk writes can be deferred until after the transaction commits. This approach, however, risks losing data in case of database crashes.

This paper presents a low-overhead primary-backup protocol to handle replicated IMDBs. Our solution consists in orchestrating IMDBs without changing their internals. Transaction termination is implemented at the middleware level, by a distributed algorithm we call *Strap*. Strap exploits two features of IMDBs: multiple-readers single-writer concurrency control and deferred disk writes. Assuming that a single update transaction modifies the database at a time allowed us to reduce to only two communication steps the latency needed to terminate update transactions, as seen by the application. In our experimental setup, this represented a reduction of 8%–25% in the response time of update transactions.

*The work presented in this paper has been partially funded by the Hasler Foundation, Switzerland (project #1899) and SNSF, Switzerland (project #200021-103556).

Deferred disk writes allowed us to reduce the overhead of committing update transactions. Transaction durability is ensured by Strap, which logs transaction information at termination, and not by the database. Traditional middleware solutions to strong-consistency database replication normally rely on two disk accesses for transaction termination: one done by the middleware to ensure correctness despite arbitrary crashes and recoveries, and another done by the database. By concentrating durability at Strap, a single disk access is needed. In our experimental setup, such an improvement represented a reduction of 51%–64% in the response time of update transactions.

Our protocol is based on the primary-backup strategy. Read-only transactions can be processed at any replica. Update transactions are executed first by the primary, and then by the backups. From the application's viewpoint, all replicas behave as a single-copy database. Primary crashes are detected by the backups and a new primary is elected whenever the current one fails. Fast reaction to primary failures is implemented through aggressive timeouts. False failure suspicion caused by aggressive failure detection is handled by allowing more than one primary to coexist at the same time without violating correctness. Provided that failures and suspicions cease, the protocol ensures that the system converges to a state in which only one primary exists.

This paper's contribution is two-fold: First, it shows how the replication and concurrency control models can be combined to terminate transactions in two communication steps, as seen by the clients. This is done without relying on strong timing assumptions about process execution and communication (e.g., no perfect failure detection). Second, the paper shows how to exploit deferred disk writes, typical of IMDBs, to ensure transaction durability outside the database. In doing so, we reduce to one the number of necessary disk writes during transaction termination.

The remainder of the paper is structured as follows. Section 2 presents the system model and some definitions. Sections 3 and 4 describe, respectively, our replication protocol, and the synchronization algorithm it relies upon. Section 5 discusses recovery issues. Section 6 contains an analytical evaluation and some experimental results. Section 7 reviews related work, and Section 8 concludes the paper. Correctness proofs for both our protocols are given in the appendix.

2. System model and definitions

2.1. Processes, communication and failures

We assume a system composed of two disjoint sets of processes, the servers, $\mathcal{S} = \{S_1, ..., S_n\}$, and the clients, $\mathcal{C} = \{C_1, ..., C_m\}$. Processes (i.e., clients and servers) communicate with each other by exchanging messages. The system is asynchronous, that is, we make no assumptions about the time it takes for processes to execute and for messages to be exchanged. Messages may be lost but if two processes remain up "long enough" simultaneously, they can exchange messages successfully (i.e., by retransmitting lost messages).

Processes may crash and recover during the execution but never behave maliciously. Regardless of how many processes fail, the system is always *safe* (i.e., it does not produce "bad" results). In order to ensure that it is also *live* (i.e., it does produce some "good" results), we assume that eventually a subset of $\mathcal{S}$ is permanently up. Servers of this kind are called *stable*.[1] The number of stable servers needed for termination is determined by Strap, the underlying synchronization protocol. Our current implementation of Strap requires $\lceil (2n+1)/3 \rceil$ stable servers.

Our primary-backup protocol assumes the existence of an unreliable failure detection oracle, which allows processes to avoid waiting forever for events that will never take place (e.g., receiving a message from a process that has crashed). The oracle is unreliable in that it may make mistakes [5]. For example, processes may incorrectly suspect that a process has crashed while in reality it is very slow, albeit fully operational. In order to ensure that progress can be made, we assume that (a) there is one stable server that is eventually not suspected by the other processes, and (b) if a process crashes, then after some time it is suspected by every stable server.

2.2. Database and transactions

Each server runs a local in-memory database (IMDB) with a copy of all data items. Transactions are short, composed only by read operations, in the case of *read-only* transactions, or read and write operations, in the case of *update* transactions. The consistency criterion is *serializability*, that is, any concurrent execution is equivalent to some serial execution with the same transactions [3].

We assume the IMDB implements *multiple-readers single-writer* concurrency control: read-only transactions can execute concurrently but update transactions are serialized with read-only and other update transactions.

For recovery reasons, IMDBs have to perform disk writes when committing update transactions. To increase performance, such writes can be deferred until after one or more update transactions have committed. The performance improvement comes with a high risk though: in case of failures some committed transactions may be lost. In the worst case, no writes are performed and should a failure occur the database loses all committed update trans-

[1] In practice it suffices for stable servers to be up for a "reasonably long" period of time, for example, enough to execute a few transactions to completion.

actions. Our primary-backup protocol combines deferred writes with the recovery properties of its underlying synchronization primitive, Strap, to avoid lost transactions and still provide good performance.

2.3. Strap's specification

Strap is a consensus-like protocol tailor-made for primary-backup database replication. Our protocol uses several instances of Strap. In each instance k any process can optimistically or conservatively propose a value v (e.g., commit requests) using, respectively, the primitives *opt-propose*$^k(v)$ and *csv-propose*$^k(v)$. Instance k's decision v is delivered through Strap primitive *deliver*$^k(v)$. Any process (i.e., client or server) can participate in an instance of Strap by opt-proposing or csv-proposing a value. Servers participate in every instance, even if they do not propose a value; clients only participate in instances in which they propose some value.

Strap's primitives satisfy the following properties:

Property 2.1 *(Delivery) If two processes deliver values v and v' in instance k and at most one value is opt-proposed in k (possibly multiple times), then $v = v'$.*

Property 1 ensures safety as long as optimistic proposals are executed by one process only and this process does not opt-propose different values. There are no restrictions concerning the execution of conservative proposes.

Property 2.2 *(Optimistic propose.) If only one process keeps opt-proposing some value v in k and no process csv-proposes a value in k, then every process that participates in instance k and does not crash delivers v.*

Property 2.3 *(Conservative propose.) If one or more processes keep csv-proposing some value in k, then every process that participates in instance k and does not crash delivers a proposed value.*

Properties 2.2 and 2.3 provide liveness. The optimistic propose primitive is used to ensure termination in the most common cases (i.e., in the absence of failures and failure suspicions). It can be implemented more efficiently than the conservative propose primitive, which is used to handle less common cases such as the election of a new primary.

2.4. The client's perspective

From the client's perspective, our primary-backup protocol provides two properties, one ensuring consistency (i.e., safety) and the other termination (i.e., liveness). The safety property is *one-copy serializability*: every concurrent execution in the replicated system is equivalent to a serial execution of the same transactions in a single-copy database [3]. The liveness property is *eventual completion*: if a client submits a transaction "enough times" and it is not suspected by the servers to have crashed, then the transaction will be eventually executed and committed. The liveness guarantee allows transactions to be aborted an unbounded number of times. This may happen for example in case of failures. Intuitively it ensures that as long as failures cease to occur for some reasonable period of time, clients will manage to find a server that will execute and commit their transactions.

3. The primary-backup protocol

To execute read-only transactions, clients can connect and submit their requests to any server. To execute update transactions, clients have first to find the current primary and then submit their requests to it.

Servers execute a sequence of *steps*, each one corresponding to a different instance of Strap. During "normal" execution periods there are no failures and no suspicions, which may happen in "abnormal" periods. In normal periods, steps are completed with clients requesting to commit or abort their update transactions.

3.1. Starting an update transaction

In normal periods, clients discover the primary by contacting any server. Since servers keep the identity of the current primary, in its first attempt a client either contacts the right server or finds out which one currently plays the role of primary. The client can then send its transaction requests to the primary, who will execute them as soon as it has finished processing previously received transactions.

If a client submits an update transaction to a backup, this will return an error message to the client with the identity of the server it believes to be the current primary. During abnormal execution periods, servers may store outdated information about the current primary, but if failures cease to occur, clients willing to run an update transaction eventually find the primary server, connect to it, and submit their requests.

After all transaction's operations are executed, the client may either abort or commit it. To abort a transaction, the client simply sends an abort request to the primary. Commit requests are handled by Strap, as explained next.

3.2. Committing update transactions

To commit an update transaction t in step k, a client gathers all requests executed on behalf of t and propagates them to all servers using Strap's opt-propose primitive (see Figure 1). Since only clients execute the optimistic propose,

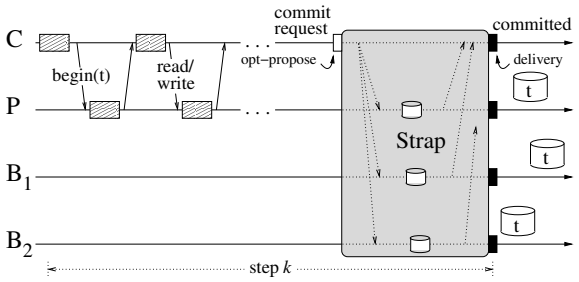

Figure 1. Normal execution period

and at most one update transaction is active at a time as a consequence of transaction requests being serialized by the primary, only one client may execute opt-propose in k. From Strap's Property 2.2, all processes that participate in instance k and do not crash deliver the commit request.

Upon delivering the commit request the primary simply submits t's commit operation to the local database; the backups submit all t's operations to their local database and then commit t. When the client delivers the commit request it has the guarantee that t will be eventually committed. This may happen before the transaction is stored in stable storage by the servers (due to deferred disk writes).

To ensure one-copy serializability, all servers must execute update transactions in the same order. Thus, if transactions t and t' have been delivered on steps k and k', $k < k'$, respectively, each server should commit t before executing t'. Since transactions are executed sequentially, this ensures that local databases will not reverse the committing order chosen by the primary for transactions t and t'.

3.3. Handling abnormal events

If the primary suspects that the client has crashed, it proposes to terminate the current step and abort the client's ongoing transaction. If a backup suspects the current primary it tries to replace it by proposing to elect a new one in the current step. Both requests are proposed using the conservative propose primitive.

Therefore, three values can be proposed in any instance k of Strap: a commit request, opt-proposed by the client, an abort request, csv-proposed by the primary, and a primary election request, proposed by backups. Due to wrong suspicions more than one value can be proposed in the same instance. However, since there exists at most one primary per step and transactions are executed sequentially, at most one client uses the opt-propose primitive in the same step.

At the end of step k, three outcomes are possible:

1. *The value delivered is a commit request.* In this case, unless it crashes, every server will execute and commit the delivered transaction.
2. *The value delivered is an abort request.* The primary aborts the transaction. All servers go to step $k + 1$.
3. *The value delivered is a primary election request.* All servers that deliver in k will determine the identity of the next primary using some deterministic function $nextPrimary(primary)$.[2] If the primary election request was triggered by a false suspicion, the primary will deliver the value, abort any update transaction in execution, and notify the client, who can re-try with the next primary.

Process crashes, message losses or anomalous communication delays may prevent a step from terminating in the first attempt. To enforce termination, according to Strap's properties, clients must eventually give up opt-proposing and start csv-proposing a commit request, and csv-proposals (from clients or servers) must be continuously repeated until a value is delivered.

4. The Strap protocol

Strap ensures safety regardless of the number of stable servers. For termination, optimistic propose requires a majority of stable servers; the conservative propose requires $\lceil (2n + 1)/3 \rceil$ stable servers. Our conservative propose algorithm is inspired by Rabin's consensus algorithm [25].

4.1. Weak-ordering oracles

A weak-ordering oracle is a communication abstraction that allows servers to broadcast messages to other servers without necessarily ensuring atomicity and order. Traditional atomic broadcast ensures that if a server delivers a message, all servers do it as well (i.e., atomicity), and that messages are delivered in the same order by all servers (i.e., total order). Weak-ordering oracles allow messages to be delivered by a subset of the servers, and in different orders. Weak-ordering oracles are intended to model link-level broadcast (e.g., Ethernet broadcast), and thus can be cheaply implemented in a cluster of computers.

Weak-ordering oracles are defined by the primitives w-broadcast$(-)$ and w-deliver$(-)$. Servers w-broadcast and w-deliver tuples of type (r, m) where r is an integer defining a *round number* and m is a broadcast message. The oracle is not reliable, allowing messages to be lost or delivered out of order. However, it extends the asynchronous model in that for an unknown but infinite sequence of round numbers $r_1, r_2, ...$, if messages are w-broadcast in r_i, all stable servers w-deliver the same first message for that round.

[2] Function $nextPrimary(primary)$ should ensure that the primary will rotate through all servers. One way to satisfy this constraint is to have for example, $nextPrimary(primary) = (primary \textbf{ mod } n) + 1$.

Readers may refer to [23] for a complete explanation and examples on weak-ordering oracles.

4.2. Optimistic propose execution

The client executing an opt-propose simply sends a request to all servers and waits for a reply from a majority of them. On receiving an opt-propose, a server logs its response to make sure that it will not forget it in case of failures and then replies to the client executing the opt-propose and to all other servers. If all replies accept the request, client and servers deliver the value.

Algorithm 1 presents the optimistic execution. Only one *when* clause executes at a time. If more than one *when* condition holds at the same time, any one is executed; but the execution is fair: unless the process crashes, every *when* clause with a true condition is eventually executed. Messages are distinguished using tags (e.g., OPTROUND). Log operations are denoted by "$\{...\}_{log}$." If a server executes $\{optVal \leftarrow v\}_{log}$ and fails, after it recovers, $optVal = v$. In the algorithms, $\perp$ and $\top$ are special values never proposed by any process.

Algorithm 1 Optimistic execution (for any p)

```
1:  Initialization:
2:    optVal_p ← ⊥
3:  To opt-propose value v_p do as follows:
4:    send (OPTROUND, v_p) to u, ∀u ∈ S
5:  Deciding a value is performed as follows:
6:    when receive (OPTROUND, v_q) from q
7:      if optVal_p = ⊥ then {optVal_p ← v_q}_log
8:      send (OPTVOTE, optVal_p) to u, ∀u ∈ {q} ∪ S
9:    when [for ⌈(n + 1)/2⌉ processes q:
        received (OPTVOTE, optVal_q) from q]
10:     if all optVal_q = v ≠ ⊤ then decide v
```

4.3. Conservative propose execution

The conservative part of Strap is inspired by the atomic broadcast algorithm based on weak-ordering oracles presented in [23]. Differently from that, however, Strap assumes that processes may crash and recover ([23] assumes the crash-stop model of failures where failed processes do not recover). The algorithm proceeds in rounds; a process can only csv-propose some value in round r if it has completed round $r - 1$.

Algorithm 2 presents the conservative execution. To execute a csv-propose, a process should first find out whether some opt-proposed value has been delivered by sending a CSVROUND request to all servers (B1) and waiting for a reply from a majority of them (B2). If a server replies to a CSVROUND request, it no longer accepts opt-propose requests (B3). A process that has already executed a csv-propose and has $phase1 = True$ starts directly executing parts B4 and B5 of the algorithm.

Algorithm 2 Conservative execution (for any p)

```
11: Initialization:
12:   r_p ← 0
13:   ∀r : estimate_p^r ← ⊥
14:   csvVal_p ← ⊥
15:   phase1 ← False
16: To csv-propose value v_p do as follows:
17:   if csvVal_p = ⊥ then csvVal_p ← v_p
18:   if ¬ phase1 then
19:     send (CSVROUND) to u, ∀u ∈ S                                  B1
20:   else
21:     w-broadcast (FIRST, r_p, csvVal_p)                            B4
22: Deciding a value is performed as follows:
23:   when receive (CSVROUND) from q                                  B3
24:     if optVal_p = ⊥ then {optVal_p ← ⊤}_log
25:     send (LASTVOTE, optVal_p) to q
26:   when [for ⌈(n + 1)/2⌉ processes q:
        received (LASTVOTE, optVal_q) from q]                         B2
27:     if ∃q | optVal_q ∉ {⊥, ⊤} then
28:       csvVal_p ← optVal_q
29:     phase1 ← True
30:     w-broadcast (FIRST, r_p, csvVal_p)
31:   when w-deliver (FIRST, r_q, value_q)                            B5
32:     if estimate_p^{r_q} = ⊥ then {estimate_p^{r_q} ← value_q}_log
33:     send (SECOND, r_q, estimate_p^{r_q}) to u, ∀u ∈ {q} ∪ S
34:   when [for ⌈(2n + 1)/3⌉ processes q:
        received (SECOND, r_q, estimate_q^{r_q}) from q]
35:     if all estimate_q^{r_q} = v ≠ ⊥ then decide v
36:     csvVal_p ← { v   majority of estimate_q^{r_q} = v ≠ ⊥
                   { ⊥   otherwise
37:     r_p ← r_p + 1
```

Figure 2 depicts a conservative execution of Strap in the absence of crashes and message losses. The csv-propose is initiated by server S_1. All servers deliver a value after receiving a message tagged SECOND from more than two thirds of the servers.

5. Recovering from failures

When a server recovers from a crash, it has to catch up with the servers that were operational while it was down. Some committed transactions may already be stable (i.e.,

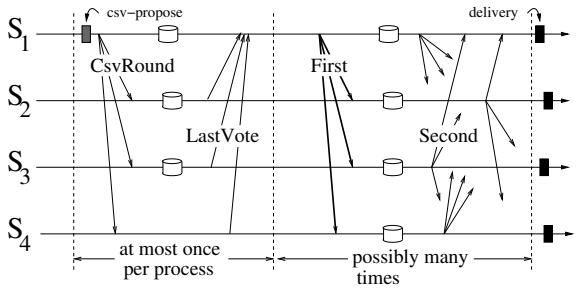

Figure 2. Strap's conservative execution

they were stored in the local disk by the IMDB), and therefore will be automatically recovered by the database upon restart. The recovery procedure has to make sure that any committed transactions lost due to the crash or committed by other servers during downtime will be (re-)executed. Intuitively, this involves discovering the last update transaction locally written to disk and learning the missing committed updated transactions. Once the missing transactions are submitted to the IMDB, the server can resume normal execution.

Finding out the last stable update transaction. In order to do so, we extend the database with a special table of a single record. This record stores the step corresponding to the last update transaction committed by the IMDB. Thus, besides its normal operations, each transaction also updates the record with the step in which it executed. (This operation is transparent to the clients, and automatically executed when the client requests the transaction's commit.) So, to learn the last update transaction safely stored by the IMDB on disk, it suffices to read the special database record.[3] We use a similar technique to identity the primary when the transaction committed. This information together with the delivery values of the steps executed after the last stable transaction allow to determine the current primary.

Learning missing committed update transactions. This boils down to learning the values delivered by other servers while the recovering server was down. Assume the last stable transaction at the recovering server executed in step k. To know what value was delivered in step $k+1$, the recovering server csv-proposes a null value in $k+1$. The propose executed by the recovering server has to be conservative because a client may have executed an opt-propose, in which case safety would not be ensured by Strap. If null is delivered, the recovering server knows that it has all committed transactions; if not, the server takes the necessary actions (i.e., according to Sections 3.2 and 3.3) and repeats the procedure for $k+2, k+3, ...$ until the delivered value is null. Notice that batches of steps can be executed in parallel to speed up recovery.

6. Performance analysis

The performance of our primary-backup protocol is intimately related to the performance of Strap's primitives in normal and abnormal execution periods. We analyze the performance of opt-propose and csv-propose analytically and experimentally under different circumstances, and compare them to those of Paxos, a well-known efficient consensus algorithm for the crash-recovery model [16].

[3]Notice that since update transactions execute serially, forcing update transactions to modify the same record does not increase contention. The only extra overhead is the space needed to store the new value, a few bytes, and the additional update operation, a fraction of a millisecond.

Table 1 compares Strap's and Paxos' primitives. The latency to terminate a transaction, which corresponds to the time between the client executes a propose and learns the delivered value, is 2δ and 4δ for Strap's optimistic and conservative proposes, respectively, where δ is the network delay. Since the minimum latency a client can observe for the termination protocol is 2δ [17], the opt-propose primitive is optimal. Paxos' "pre-reserved" propose needs an extra message delay to send the proposed value to the leader. Although Paxos' classic propose primitive has the same expected latency as csv-propose, the first phase of Strap's csv-propose (two message delays) is omitted in consecutive invocations whereas Paxos has to execute entirely to ensure progress. Both algorithms are similar considering the number of messages, both point-to-point and broadcast. When it comes to resilience, the csv-propose primitive requires $f < n/3$, where f is the number of unstable processes. The major difference between the two protocols relies on the synchrony assumptions needed for liveness.

We have built a prototype of our primary-backup protocol to measure its performance experimentaly. The experiments were conducted in a cluster of Intel 2.4 GHz Pentium IV processor servers with 1GB of main memory, running Fedora Linux, and connected through a Gigabit Ethernet. We used FirstSQL/J [9], a pure java IMDB. FirstSQL/J implements deferred disk writes by granting full control about when and how data is made stable to the application layer.

To measure the performance during normal execution, we had a client constantly submitting update transactions to the primary. The results are shown in Figure 3(a). In all the curves we consider only the transaction termination time, i.e., the time it takes for the client to deliver a commit request after its submission. Transactions are composed of three update operations. Since we are concerned with termination time only, our results depend solely on write operations. All measured points represent the average of 1000 executions and have a negligible confidence interval of 95%, not appearing in the graph. Figure 3(a) shows that our protocol combined with Strap (PBackup/Strap) has a termination delay slightly worse than a non-replicated approach, even for a system with 8 replicas. In the client/server implementation we used FirstSQL/J with immediate disk writes at commit time in order to ensure transaction durability. Besides the small constant overhead of PBackup/Strap, the graph also shows that response time increases linearly with the number of replicas. An interesting question is whether the major performance overhead comes from log writes or CPU/memory contention by committing the transaction in the IMDB. As Figure 3(a) shows, by disabling each of these features it turns out that the log writes represent an overhead much lower than the (in-memory) database commit.

We also show how performance would be if we run our protocol with immediate disk writes at the databases. This

Protocol	Expected Latency	Number of Messages		Resilience	Assumptions for Liveness
		Point-to-Point	Broadcast		
opt-propose	$2\,\delta$	n^2-1	$n+1$	$f < n/2$	—
csv-propose	$4\,\delta$	$n(n+2)-3$	$2n+1$	$f < n/3$	Weak-ordering Oracle
Paxos (pre-reserved)	$3\,\delta$	n^2	$n+2$	$f < n/2$	Unique Leader
Paxos (classic)	$4\,\delta$	$n(n+2)-3$	$2n+1$	$f < n/2$	Unique Leader

Table 1. Analytical comparison between Strap's and Paxos' primitives

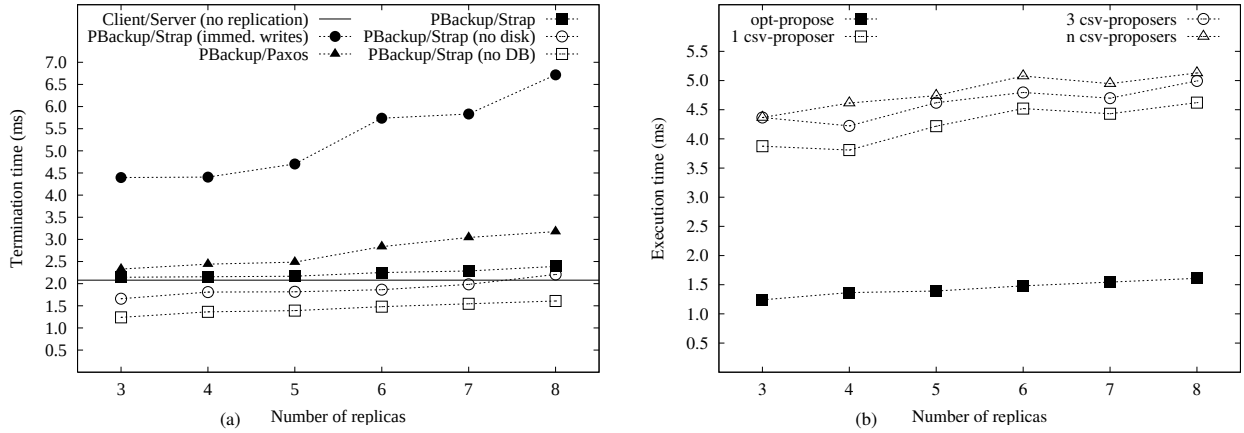

Figure 3. (a) Execution in normal periods and (b)comparison between Strap's primitives

would be the expected performance of usual replication approaches that treat the group communication primitives and the database as black boxes. For a transaction to commit, the server first logs the pre-delivery of the commit decision and, just after that, writes the database log. Furthermore, one could think of using Paxos instead of Strap in our protocol. In that case, during normal execution periods the "pre-reserved" optimization of Paxos could be used. From Figure 3(a) the extra message delay makes a difference in performance, favoring the use of Strap.

Finally, we analyze the cost of csv-propose, used to elect a new primary if the current one is suspected to have failed. That is shown in Figure 3(b), where each point represents the average of 1000 executions with a negligible confidence interval of 95%. We compare the execution time of an opt-propose with that of a csv-propose when there is only one proposer, three processes propose concurrently, and all servers propose concurrently.

7. Related Work

It is shown in [12] that eagerly updating replicated data can lead to high abort rates as a consequence of concurrent accesses. This result motivated much research in group-communication-based database replication as a way of synchronizing concurrent updates and reducing the abort rate. These works differ from ours either (a) by not considering message losses or process recovery (e.g., [1, 13, 20, 21, 22, 26]), (b) by having bigger latency for update propagation due to the view-synchrony protocols used [2, 14], or (c) by requiring modifications to the database engine [14, 15]. An approach that resembles ours with respect to optimism and the way clients behave is [8]. However, besides the differences in the type of replication and adopted model, the work in [8] requires servers to be able to roll back a suffix of their local computation.

The Paxos [16] atomic broadcast protocol and related approaches [6, 7] could be used in the conservative part of Strap. Conversely, some of the extensions of Paxos are independent of the basic consensus algorithm and could be applied to algorithms like Strap (e.g., [18]). In [11], it was shown an optimization that could give Paxos the same latency as Strap during normal execution. However, by using this version of Paxos instead of Strap, the leader used by Paxos would have to be the same one used by our protocol, which would not work with the deterministic election we have. If the current leader and the next one in the list have failed, Paxos would have to use a different leader to ensure liveness. Bypassing this problem is possible, but it would increase the design complexity of our protocol.

We are unaware of group-communication-based protocols that explicitly take advantage of IMDBs' properties to implement replication. Replication for IMDBs (e.g. [9]) has been traditionally based on primary-backup techniques in which the primary propagates "system-level" information (e.g., database pages) to the backups. The backups monitor the primary and in case they suspect the primary to have crashed, some backup takes over. If the primary

is wrongly suspected, more than one primary may co-exist, leading to data inconsistencies. As a consequence, failure detection tends to be conservative to avoid mistakes, leading to slow reaction to failures, which is inadequate for time-critical applications. Mirroring has been proposed as an alternative approach. In [28] virtual-memory-mapped communication is used to achieve fast failover by mirroring the primary's memory on the backups. Perseas [19] is a transaction library that mirrors user space memory on remote machines connected by a high-speed network. It may not ensure strong consistency in certain failure scenarios or if processes disagree on the primary.

8. Final Remarks

This paper introduces a new data management protocol for in-memory database clusters. IMDBs have been known for quite some time now. Until recently, however, their use was reserved to specific contexts (e.g., embedded and real-time applications). Continuously increasing demand for performance and decreasing semiconductor memory prices have broadened their scope. Should network speeds continue to increase and memory prices continue to fall, more and more performance-critical applications will be able to benefit from IMDBs clusters. Although this protocol exploits some features of IMDBs, one could think of applying a similar approach to disk-based databases. The biggest challenges are to handle the non-determinism created by concurrent execution transactions and the performance overhead from non-sequential disk access. Investigating such issues is theme of future work.

References

[1] D. Agrawal, G. Alonso, A. E. Abbadi, and I. Stanoi. Exploiting atomic broadcast in replicated databases. In *Proceedings of EuroPar (EuroPar'97)*, Passau (Germany), Sept. 1997.

[2] Y. Amir and C. Tutu. From total order to database replication. In *International Conference on Distributed Computing Systems (ICDCS)*, July 2002.

[3] P. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley, 1987.

[4] S. Blott and H. F. Korth. An almost-serial protocol for transaction execution in main-memory database systems. In *Proceedings of the 28th International Conference on Very Large Data Bases (VLDB)*, pages 706–717, 2002.

[5] T. D. Chandra and S. Toueg. Unreliable failure detectors for reliable distributed systems. *Journal of the ACM*, 43(2):225–267, Mar. 1996.

[6] G. Chockler and D. Malkhi. Active disk paxos with infinitely many processes. *Distributed Computing*, 18(1):73–84, 2005.

[7] P. Dutta, F. Frølund, R. Guerraoui, and B. Pochon. An efficient universal construction for message-passing systems. In *Proceedings of the 16th International Symposium on Distributed Computing (DISC'02), Springer-Verlag, LNCS*, pages 133–147, October 2002.

[8] P. Felber and A. Schiper. Optimistic active replication. In *Proceedings of* 21^{st} *International Conference on Distributed Computing Systems*, 2001.

[9] FirstSQL Inc. The FirstSQL/J in-memory database system. http://www.firstsql.com.

[10] H. Garcia-Molina and K. Salem. Main memory database systems: An overview. *IEEE Transactions on Knowledge and Data Engineering*, 4(6):509–516, 1992.

[11] J. Gray and L. Lamport. Consensus on transaction commit. Technical Report MSR-TR-2003-96, Microsoft, 2003.

[12] J. N. Gray, P. Helland, P. O'Neil, and D. Shasha. The dangers of replication and a solution. In *Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data*, Montreal (Canada), June 1996.

[13] J. Holliday, D. Agrawal, and A. E. Abbadi. The performance of database replication with group multicast. In *Proceedings of International Symposium on Fault Tolerant Computing (FTCS29)*, pages 158–165. IEEE Computer Society, 1999.

[14] B. Kemme and G. Alonso. Don't be lazy, be consistent: Postgres-r, a new way to implement database replication. In *Proceedings of 26th International Conference on Very Large Data Bases*, Sept. 2000.

[15] B. Kemme and G. Alonso. A new approach ro developing and implementing eager database replication protocols. *ACM Transactions on Database Systems (TODS)*, 25(3), September 2000.

[16] L. Lamport. The part-time parliament. *ACM Transactions on Computer Systems*, 16(2):133–169, May 1998.

[17] L. Lamport. Lower bounds on consensus algorithms. Private communication, 2004.

[18] L. Lamport and M. Massa. Cheap paxos. In *Proceedings of the International Conference on Dependable Systems and Networks (DSN'2004)*, July 2004.

[19] A. E. Papathanasiou and E. P. Markatos. Lightweight transactions on networks of workstations. In *Proceedings of the the 18th International Conference on Distributed Computing Systems (ICDCS'98)*, pages 544–552, Washington, DC, USA, 1998.

[20] M. Patino-Martínez, R. Jiménez-Peris, B. Kemme, and G. Alonso. Scalable replication in database clusters. In *Disctributed Computing (DISC)*, 2000.

[21] F. Pedone and S. Frølund. Pronto: A fast failover protocol for off-the-shelf commercial databases. In *Proceedings of* 19^{th} *IEEE Symposium on Reliable Distributed Systems (SRDS'2000)*, Nürnberg, Germany, 2000.

[22] F. Pedone, R. Guerraoui, and A. Schiper. Transaction reordering in replicated databases. In *Proceedings of the 16th IEEE Symposium on Reliable Distributed Systems*, Durham (USA), Oct. 1997.

[23] F. Pedone, A. Schiper, P. Urban, and D. Cavin. Solving agreement problems with weak ordering oracles. In *4th European Dependable Computing Conference (EDCC-4)*, Toulouse, France, Oct. 2002.

[24] G. F. Pfister. *In search of clusters*. Prentice Hall, 1998.

[25] M. Rabin. Randomized byzantine generals. In *Proceedings of* 24^{th} *Annual ACM Symposium on Foundations of Computer Science*, 1983.

[26] L. Rodrigues, H. Miranda, R. Almeida, J. Martins, and P. Vicente. Strong replication in the GLOBDATA middleware. In *Workshop on Dependable Middleware-Based Systems*, 2002.

[27] D. F. Savarese and T. Sterling. Beowulf. In *High Performance Cluster Computing*, volume 1, Architectures and Systems, pages 625–645. Prentice Hall PTR, 1999.

[28] Y. Zhou, P. Chen, and K. Li. Fast cluster failover using virtual memory-mapped communication. Technical Report TR-591-99, Princeton University, Jan. 1999.

9 Appendix

9.1 Primary Backup protocol correctness

For the sake of brevity we refer to our primary backup protocol simply as PBackup in the following.

Property 3.1 *The protocol is one-copy serializable.*

PROOF: Let E be the committed projection of some execution of PBackup containing only update transactions (we consider queries later). From the algorithm, each transaction t in E was committed in a particular step k_t. We prove that E is 1SR, that is, one-copy serializable, by induction on k. Let $\xi(E,k)$ be the prefix of E containing transactions committed in steps $1, ..., k$.

Base step. We show that $\xi(E,1)$ is 1SR. Either $\xi(E,1)$ contains a single transaction t, in which case t's commit request was delivered in step 1, or it is empty, if t's commit request was not the value delivered in step 1. Since transactions only read committed data, it is easy to see that $\xi(E,1)$ is 1SR.

Induction step. Assume that $\xi(E,k-1)$ is 1SR, for every $k > 1$; we show that $\xi(E,k)$ is also 1SR. If the value delivered in k is not a commit request, then $\xi(E,k-1) = \xi(E,k)$, and the proposition is trivially true. So, assume that a commit request, say t's, was delivered in step k. Let t_α be the last committed transaction in $\xi(E,k-1)$, k_α the step in which t_α's commit request was delivered. Either (a) t_α and t executed in the same primary S_i, or (b) they executed in different primaries, say S_i and S_j. We show that t only started after t_α committed.

The statement clearly holds for (a). In (b), for a contradiction, assume that t starts before t_α is committed. Before terminating step k, S_j terminated step k_α. Upon delivering t_α's commit request, S_j aborted any transaction in execution, contradicting the fact that t is committed, and concluding the proof. □

Property 3.2 *The protocol ensures eventual completion.*

PROOF: We show that if a client keeps resubmiting a transaction t and is not suspected by the servers, then t will be eventually executed and committed. The proof is a consequence of the following facts about PBackup :

Fact 1. A primary never blocks executing a transaction. The fact obviously holds if the primary crashes. Assume it does not crash, and remains blocked during the execution of some transaction. Thus, the primary eventually gives up waiting for the client (e.g., suspects the client) and terminates the step proposing to abort the ongoing transasction. If the value delivered is a commit request, the transaction is committed; otherwise the transaction is aborted. In either case the primary does not remain blocked.

Fact 2. There is a time τ after which some server S_p becomes primary forever and never crashes. From the definition of stable servers, there exists some time τ_1 after which stable servers are permanently up. Assume that none of these servers is primary. Then the primary keeps crashing and possibly subsequently recovering. By the properties of the failure detection mechanism, such a primary is eventually suspected and another one elected to replace it. Thus, at some time $\tau_2 \geq \tau_1$ the primary is a stable server. Assume now that no server is permanently elected primary. So, every stable server keeps being suspected by the other servers, contradicting the fact that some stable server is eventually not suspected by the other servers. We conclude that $\tau \geq \tau_2$.

The proof continues by contradiction. Assume t is never committed. Then the client keeps submitting t, possibly to different primaries. After τ, t is submitted for execution to S_p. Since becomes primary forever (from Fact 2) and does not block (from Fact 1), it will execute and commit t, a contradiction that concludes the proof. □

9.2 Strap's Correctness

First of all, some facts must be noted. The algorithm changes the value of $optVal$ only once, since changes are logged and properly recovered after failures. Changes on $optVal$'s value are done on lines 7 and 25 only. On Line 7, it is changed to v, the value received on the first message ($\underline{\text{OPTVOTE}}$,$v$) to arrive, while on Line 25 it is only changed to $\top$. So we can conclude Fact 3:

Fact 3: If only a single process proposes and if it does so all the times with the same value v, then for all process p, $optVal_p \in \{v, \bot, \top\}$.

Before sending a ($\underline{\text{OPTVOTE}}, w$) or a ($\underline{\text{LASTVOTE}}, w$) message, the algorithm always checks if $optVal = \bot$ and, if it does, changes it to v, on Line 7, or to $\top$, on Line 25. Summing this to Fact 3, we have two new conclusions:

Fact 4: If only a single process proposes and if it does so all the times with the same value v, for all ($\underline{\text{OPTVOTE}}, w$) and ($\underline{\text{LASTVOTE}}, w$) messages, $w \in \{v, \top\}$

Fact 5: For any instance k and a given process p, ($\underline{\text{OPTVOTE}}, optVal_p$) and ($\underline{\text{LASTVOTE}}, optVal_p$) messages always carry the same and definitive value of $optVal_p$.

A similar behavior is verified with respect to $estimate$, also due to the use of stable storage and the checking done before sending a $\underline{\text{SECOND}}$ message, bringing out Fact 6:

Fact 6: A process will never send two different $Second$ messages referring to the same round.

Property 2.1 *If two processes deliver values v and v' in instance k and at most one value is opt-proposed in k (possibly multiple times), then $v = v'$.*

PROOF: We prove it by means of a contradiction. Let p_1 and p_2 be processes that have decided on v_1 and v_2, $v_1 \neq v_2$, respectively. We have three cases to analyze:

Case 1: If both processes have decided through the optimistic part of the algorithm. Before deciding on v_1 (resp. v_2), p_1 (resp. p_2) has received $\lceil (n+1)/2 \rceil$ ($\underline{\text{OPTVOTE}}, v_1$) (resp. v_2) messages. As there are just n servers that take part on the optimistic algorithm, some process p_3 must have sent both ($\underline{\text{OPTVOTE}}$,$v_1$) and ($\underline{\text{OPTVOTE}}, v_2$) messages. This is known to be impossible due to Fact 5.

Case 2: If p_1 decides through the optimistic part of the algorithm and p_2 through the conservative part. In order to decide, p_1 has received ($\underline{\text{OPTVOTE}}, v_1$) messages from a set Q of processes such that $|Q| = \lceil (n+1)/2 \rceil$. Hence, due to Fact 5, $\forall q \in Q, optVal_q = v_1 \neq \top$ and (A)all $\underline{\text{LASTVOTE}}$ messages sent by a member of Q will have the form ($\underline{\text{LASTVOTE}}, v_1$).

Before sending a $\underline{\text{FIRST}}$ message, processes gather $\lceil (n+1)/2 \rceil$ $\underline{\text{LASTVOTE}}$ messages and recalculates the $csvVal$'s value (Line 22 is allowed to execute only if this gathering has already been performed, what is signaled through by $phase1$ variable, achieving the same effect). Because there are only n servers, at least one of these $\lceil (n+1)/2 \rceil$ $\underline{\text{LASTVOTE}}$ messages came from a process in Q. Due to this, to Fact 4, and to (A), proposal will surely be attributed v_1, and (B)any $\underline{\text{FIRST}}$ message sent will have the form ($\underline{\text{FIRST}}, , v_1$).

In order to decide v_2, p_2 received ($\underline{\text{SECOND}}, r, v_2$) messages from $\lceil (2n+1)/3 \rceil$ different processes. By the algorithm, a ($\underline{\text{SECOND}}, r, v_2$) implies $estimate^r_{p_3} = v_2$ for some process p_3. But, as $estimate^r_{p_3}$ value equals v of the first ($\underline{\text{FIRST}}, r, v$) received and, as is known from (B), $v = v_1$, v_1 must equals v_2, a contradiction.

Case 3: If p_1 and p_2 decides through the conservative part. Here, two sub-cases are possible: *(3.1)* p_1 and p_2 decided on the same round; or *(3.2)* in different rounds.

(3.1) Let r be the first round in which p_1 and p_2 decide. In order to decide p_1 (resp. p_2) received ($\underline{\text{SECOND}}, r, v_1$) (resp. v_2) messages from $\lceil (2n+1)/3 \rceil$ different processes. As there are only n servers, there must be process p_3 that sent both ($\underline{\text{SECOND}}, r, v_1$) and ($\underline{\text{SECOND}}, r, v_2$) messages. As shown in Fact 6, this cannot happen.

(3.2) Let r_1 and r_2, $r_1 < r_2$, be the rounds in which p_1 and p_2 decide, respectively. p_1 received $\lceil (2n+1)/3 \rceil$ messages ($\underline{\text{SECOND}}, r_1, v_1$) before deciding. So, there must be a set Q of processes such that $|Q| = \lceil (2n+1)/3 \rceil$ and $\forall p \in Q$, $estimate^{r_1}_p$ permanently equals v_1. Any process p_3 that finishes round r_1 will execute Part B.4 of algorithm, i.e., will gather $\lceil (2n+1)/3 \rceil$ ($\underline{\text{SECOND}}, r1, \_$) messages; as there are only n processes, a majority of these messages will come from processes on Q and, consequently, $csvVal_{p_3}$ will be set to v_1. Hence, any such process p_3 that csv-proposes on round $r_1 + 1$ will do a wo-broadcast($\underline{\text{FIRST}}, r_1 + 1, v_1$) and any process that wo-deliver this message will permanently set $estimate^{r_1+1} = v_1$. Follows from an induction on round number that no ($\underline{\text{SECOND}}, r', v$) message, $r_1 < r' \leq r_2$ will have $v \neq v_1$, that makes p_2 decision on $v_2 \neq v_1$ impossible. □

Property 2.2 *(Optimistic propose.) If only one process keeps opt-proposing some value v in k and no process csv-proposes a value in k, then every process that participates in instance k and does not crash delivers v.*

PROOF: The proof is by contradiction. Let p_2 be the process that keeps opt-proposing and p_1 a process that participates in instance k, does not crash and does not deliver v. Due to Property 2.1, p_1 cannot deliver any value other than v, and so, p_1 delivers nothing at all.

Due to Fact 4 and to the fact that no csv-proposal is done, i. e., Line 25 is never executed, any $\underline{\text{OPTVOTE}}$ message will have the form ($\underline{\text{OPTVOTE}}, v$).

There is a time t after which p_1 does not crash, nor $\lceil (n+1)/2 \rceil$ stable servers. In order not to decide, p_1 does not receive the $\lceil (n+1)/2 \rceil$ equal $\underline{\text{OPTVOTE}}$ messages that would be needed. As seen, p_1 cannot have received ($\underline{\text{OPTVOTE}}, w$), $w \neq v$, so it has not received $\lceil (n+1)/2 \rceil$ $\underline{\text{OPTVOTE}}$ messages after t. Had the stable servers kept sending such messages, p_1 would have received them, so we can conclude that the stable servers have stop sending these $\underline{\text{OPTVOTE}}$ messages after t. Furthermore, we can conclude that stable servers have stopped receiving $\underline{\text{OPTROUND}}$ messages as, after t, any $\underline{\text{OPTROUND}}$ message received would result in sending a $\underline{\text{OPTVOTE}}$ message. Otherwise p_1 would have received enough $\underline{\text{OPTVOTE}}$ messages. By the model, $\underline{\text{OPTROUND}}$ messages cannot stop being received unless they have stopped being sent, a clear contradiction to our assumption. □

Property 2.3 *(Conservative propose.) If one or more processes keep csv-proposing some value in k, then every process that participates in instance k and does not crash delivers a proposed value.*

PROOF: The proof is by contradiction. Let p_2 be a process that keep csv-proposing and p_1 a process that takes part in instance k, does not crash and does not deliver v. Due to property 2.1, if p_1 does not deliver v, then it delivers nothing at all.

There is a time t after which p_1 does not crash, nor the $\lceil (2n+1)/3 \rceil$ stable servers in the system. In order not to decide, after t p_1 does not receive the $\lceil (2n+1)/3 \rceil$ equal $\underline{\text{SECOND}}$ messages that would be needed, where equal means "with the same estimate". So, or (A)p_1 does not receive $\lceil (2n+1)/3 \rceil$ messages or (B)they are not equal. Lets first analyze (A).

Case (A): If p_1 does not receive the messages, they must have stopped being sent. As there are $\lceil (2n+1)/3 \rceil$ servers

that would send them after receiving a FIRST message, we can assume that FIRST messages have stopped being sent. Following the same reasoning we finally conclude that CSVROUNDhave ceased being broadcast, what would not happen unless process p_2 stops csv-proposing. If p_1 does not deliver v, case (B) must hold.

Case (B): In order to this case hold, SECOND messages with different values must have been sent. Because of Fact 6 we know that different processes must have wo-delivered different FIRST messages. Once messages keep arriving, as seen in the previous paragraph, processes keep initiating new rounds. As we use a WAB to broadcast messages of FIRST type, an infinite number of rounds that run after t would deliver the same FIRST message on all running servers, generating SECOND messages with the same value. Eventually, in one of those rounds, p_1 will deliver these equally valued SECOND messages and so, (B) cannot hold. □