Self-stabilization Overhead: A Case Study on Coded Atomic Storage

Chryssis Georgiou[1], Robert Gustafsson[2,3], Andreas Lindhé[2,3], and Elad Michael Schiller[2(✉)]

[1] University of Cyprus, Nicosia, Cyprus
chryssis@cs.ucy.ac.cy
[2] Chalmers University of Technology, Gothenburg, Sweden
{robg,lindhea}@student.chalmers.se, elad@chalmers.se
[3] Combitech AB, Linköping, Sweden
{andreas.lindhe,robert.gustafsson1}@combitech.se

Abstract. Shared memory emulation on distributed message-passing systems can be used as a fault-tolerant and highly available distributed storage solution or as a low-level synchronization primitive. Cadambe *et al.* proposed the Coded Atomic Storage (CAS) algorithm, which uses erasure coding to achieve data redundancy with much lower communication cost than previous algorithmic solutions. Recently, Dolev *et al.* introduced a version of CAS where transient faults are included in the fault model, making it self-stabilizing. But self-stabilization comes at a cost, so in this work we examine the overhead of the algorithm by implementing a system we call CASSS (CAS Self-Stabilizing). Our system builds on the self-stabilizing version of CAS, along with several other self-stabilizing building blocks. This provides us with a powerful platform to evaluate the overhead and other aspects of the real-world applicability of the algorithm.

In our case-study, we evaluated the system performance by running it on the world-wide distributed platform PlanetLab. Our study shows that CASSS scales very well in terms of the number of servers, the number of concurrent clients, as well as the size of the replicated object. More importantly, it shows (a) to have only a constant overhead compared to the traditional CAS algorithm and (b) the recovery period (after the last occurrence of a transient fault) is no more than the time it takes to perform a few client (read/write) operations. Our results suggest that the self-stabilizing variation of CAS, which is CASSS, does not significantly impact efficiency while dealing with automatic recovery from transient faults.

1 Introduction

Sharing a data object among decentralized servers that provide distributed storage has been an active research topic for decades. We consider the problem of emulating a shared memory in a way that appears atomic (linearizable) [1]. Early solutions [2,3] do not scale well when it comes to larger data objects due to the

M. F. Atig and A. A. Schwarzmann (Eds.): NETYS 2019, LNCS 11704, pp. 131–147, 2019.
https://doi.org/10.1007/978-3-030-31277-0_9

use of full replication of the data to all servers in the system. Cadambe *et al.* [4] proposed the *Coded Atomic Storage* (CAS) algorithm, which uses erasure coding in order to achieve data redundancy but with much lower communication cost compared with algorithms that use full replication. Although CAS provides an efficient solution that tolerates node crashes, Dolev *et al.* [5,6] solve the same problem while considering an even more attractive notion of fault-tolerance since their solution can recover after the occurrence of *transient faults*. Such faults model any violation of the assumption according to which the system was designed to operate. Dolev *et al.* present a self-stabilizing version of CAS, which we refer to as CASSS (CAS Self-Stabilizing). Unlike CAS, their version guarantees recovery after the occurrence of transient faults. The authors suggests that the variant of CAS from [5] has similar communication costs as CAS [4]. Our results validate [5]'s prediction, but more importantly, they demonstrate the system's ability to recover from transient faults efficiently, while tolerating node failures.

Atomic Shared Memory Emulation. The goal of emulating a shared memory is to allow the clients to access via read and write operations a shared storage in the network. By that, the service hides from the user low-level details, such as message exchange between the clients and the servers. As the shared data is replicated on the servers, data consistency between the replicas (data copies) must be ensured. Atomicity (linearizability) [1] is the strongest consistency guarantee and provides the illusion that operations on the distributed storage are invoked sequentially, even though they can be invoked concurrently. A read (resp. write) operation is invoked with a read (resp. write) request and it *completes* with a response (*e.g.,* an acknowledgment). There are two criteria that need to be satisfied for the atomicity property: (1) Any invocation of a read operation, after a write operation is completed, must return a value at least as recent as the value written by that write operation. (2) A read operation that follows another read operation will return a value at least as recent as the value returned by the first read operation. Thus, the operations appear sequential.

Fault Model. *(i) Benign Failures.* We consider message passing systems in which communication failures may occur during packet transit, such as packet loss, duplication, and reordering. However, the studied algorithms assume *communication fairness*, *i.e.,* if the sender transmits a packet infinitely often, the receiver gets this packet infinitely often. The early solutions [2,3] model node failures as crashes and restrict the number f of failing servers (nodes) to be less than half of the nodes in the system. We follow a similar approach but require that in the presence of transient faults, and only then, a crashed node either restarts (we call this a *detectable restart*) or is removed from the system via a reconfiguration service [7]. Moreover, as specified in [5,6], our restriction on the number of crashes f is similar to the one of CAS [4].

(ii) Transient Faults. We also consider violations of the assumptions according to which the system was designed to operate. We model their impact on the system as arbitrary changes of the *state*, as long as the program code stays intact. Since these faults are rare, our model assumes that the system starts after the last occurrence of these transient faults. Transient faults can, for example, be soft errors (which are sometime called, single event upset) or the event of a CRC code failing to detect a bit error in a transmitted packet.

(iii) Self-stabilization. These design criteria, which requires recovery without external (human) intervention, provides a strong fault-tolerance guarantee in that the system always recovers from transient faults. By the definition given by Dijkstra [8], the correctness proof of a self-stabilizing system needs to show recovery within a bounded period after the last transient fault. That is, when starting from an arbitrary system state, the system needs to exhibit legal behavior within a bounded time.

Dolev *et al.* [7] proposed the following refinement of Dijkstra's design criteria of self-stabilization, which we believe to be convenient for dealing with the asynchronous nature of distributed systems. In the absence of transient faults, the environment is assumed to be asynchronous. Moreover, servers and clients may at any time crash. In the presence of transient faults, it is assumed that (a) all failing servers to recover eventually and (b) there is a sufficiently long period (which allows recovery) in which the system run is fair, *i.e.,* each node makes progress infinitely often.

Related Work. Shared memory can support either a *single-writer and multi-reader* (SWMR) context, *e.g.,* ABD [2], or a *multi-writer and multi-reader* (MWMR) context, *e.g.,* MW-ABD [3]. A discussion on such non-self-stabilizing solutions is given in [9].

The term *reconfiguration* refers to a change from one server configuration to another and requires old configuration members to send the data to the new members; the data is replicated to all configuration members. Shared memory emulation has also been studied under such dynamic server participation, *e.g.,* RAMBO [10]. See [11] for a survey on (non-self-stabilizing) reconfigurable solutions to memory emulation. ARES [12] is a recent solution that supports reconfiguration of a shared memory emulation service and is based on erasure coding. The authors also present the first atomic memory service that uses erasure coding with only two rounds of message exchanges for a client operation. While combining these two creates an efficient solution with respect to liveness, even during configuration collapses, such a solution does not consider self-stabilization.

Nicolaou and Georgiou [13] did an experimental evaluation of four non-self-stabilizing MWMR register emulation algorithms on PlanetLab. The algorithms evaluated were *SWF, APRX-SWF, CwFr* and *SIMPLE*. Algorithm SIMPLE is an MWMR version of ABD for quorum systems (quorums are intersecting sets of servers), similar to the one we use in this work (called MW-ABD) to compare its performance with CAS and CASSS.

Our Contributions. We are the first to implement, and evaluate via experiments, a self-stabilizing algorithm for coded atomic MWMR shared memory emulation. We show that the overhead associated with self-stabilization does not really affect the efficiency advantage associated with erasure coding. We have also implemented a (graceful) counter restart mechanism, based on principles from [7]. The counter restart mechanism can perform a (synchronized) global reset of the entire system while keeping the most recently written data object using an agreement protocol. Additionally, we implemented a self-stabilizing reincarnation number service [5], which provides recyclable client identifiers, and by that helps to deal with detectable client restarts.

Our experiments on PlanetLab shows that our pilot implementations of CAS and CASSS have comparable performances with respect to operation latency. Furthermore, the evaluation shows that our implementation of CASSS scales very well when increasing the number of servers and clients, respectively. More importantly, the overhead caused by self-stabilization in our experiments is only greater than the non-self-stabilizing CAS implementation by a *constant* factor. The system evaluation shows almost no slowdown for data objects up to 512 KiB, and is only slightly slower for data objects up to 1 MiB. Last but not least, the evaluation reveals that the counter restart mechanism is fast – it takes about the same amount of time as three or four normal write operations. This demonstrates CASSS's ability to rapidly recover from transient faults. Encouraged by these evaluation results, we believe that our pilots and their building blocks could be used for implementing other self-stabilizing algorithms and prototypes.

2 System and Background

The system includes a network of N nodes. Each node can host clients and/or servers. Servers use a gossip service for communicating among themselves. Clients interact with the shared-memory service using read and write operations. These operations include multiple communication rounds of requests and responses. Every client performs its operations sequentially, but its operations may be interleaved arbitrarily with operations from other clients.

Servers are arranged into pairwise intersecting sets, or *quorums*, that together form a *quorum system*. The intersection property of quorums enables information communicated to a quorum to be passed (via the common servers) to another quorum. Majorities (subsets containing a majority of the servers) form a simple quorum system (used, for example, in ABD [2]). The *self-stabilizing* quorum system considered in this work follows the one proposed in [5,6]. We note that the quorum system needs to be self-stabilizing. This is, for example, because of the fact that client algorithms often include several phases. The clients and the servers needs to be synchronized both with respect to the phases and the associated object version.

Each server has access to a set of *records*, which are tuples of the form *(tag,data,phase)*. A tag has the form of $(number, clientID)$, *i.e.,* a pair with a sequence number and the unique identifier of the client that is writing this

version. The data field holds either null, or a coded element of an object that is stored in the system. The tag is used to determine the causal relationship among operations, *e.g.,* when retrieving the object's most up-to-date version. The phase field keeps track of which *phase* of the protocol that the data in the record have reached.

2.1 The CAS Algorithm

Coded Atomic Storage (CAS) is based on techniques for reducing communication costs, such as erasure coding and an earlier algorithm [14], by avoiding full replication, as in ABD and MW-ABD. CAS is a quorum based algorithm, where a quorum is any subset Q of the servers, such that $|Q| \geq k_{threshold} = \lceil \frac{N+k}{2} \rceil$; N is the number of servers and k is the coding parameter deciding the least amount of needed elements to decode the object value. CAS allows for up to f server failures. See [4] for full details.

Writer's Procedure. There are three phases: *query*, *pre-write* and *finalize*. *Query:* The query phase is based on MW-ABD [15]'s query, but considers only finalized records, *i.e.,* records that has their *phase* field set to 'fin′ (rather than 'pre′).

Pre-write: p_i's client sends a message, $\langle (x+1, i), m_j, \text{'pre}' \rangle$, to any server p_j and waits for a quorum of replies, where x is the maximum tag number retrieved from the query phase and m_j holds the coded element to the server at p_j.

Finalize: p_i sends a message $\langle (x, i+1), \perp, \text{'fin}' \rangle$, to all servers. After receiving a quorum of acknowledgments, the write operation is finished. The finalize phase hides the write operations that have not been seen by a quorum since the query phase only looks at records with phase 'fin'. Once the client has passed the *pre-write* phase, it knows that at least a whole quorum has enough elements to reconstruct the data and therefore it can be made visible in other operations.

Reader's Procedure. There are two phases: *query* and *finalize*. The query phase is identical to the writer's query phase.

Finalize: client p_i sends out a message $\langle t_{max}, m_j, \text{'fin}' \rangle$ to all servers, where $t_{max} = (x, \bullet)$ is the tag retrieved from the query phase. The client waits until a quorum has responded; each response includes a coded element corresponding to t_{max} (or a null if the server stored no record corresponding to t_{max}). If at least k of the responses include a coded element, the reader decodes the object value and returns it to the application. Otherwise, it just returns as an unsuccessful read.

Server's Events. The servers store different versions of the objects in records of the form $(t, w, label)$, where t is a tag, w is a coded element and $label$ is either 'pre' or 'fin'. The server's procedures include the event handlers corresponding to the client requests: query, pre-write and finalize (of both read and write). Note

Algorithm 1: A description of CASSS, code for p_i's client and server

1 **The client:** For $write(s)$**:** Query all servers for finalized tags. After hearing from a quorum, get the maximum tag (z, j). Encode the elements $w_1, w_2, \ldots, w_N$ using input s, such $p_i \in P$ hosts a server. Send to all servers $((z+1, j), w_i, \text{'pre}')$ and wait for a quorum of replies. For each server $p_j \in P$, send $((z+1, j), \text{'null}', \text{'fin}')$ and wait for a quorum of replies. The algorithm uses the 'FIN$'$ phase label for making sure that at least a quorum of servers store the record $((z+1, j), w_i, \text{'fin}')$. This way, the quorum intersection property guarantees the record visibility w.r.t. prospective client operations. Then, send for each p_j's server $((z+1, j), \text{'null}', \text{'FIN}')$ and wait for a quorum before returning. The algorithm uses the 'FIN$'$ phase label for making sure that any server that has the record $((z+1, j), w_i, \text{'FIN}')$ knows that there is at least a quorum of servers with the record $((z+1, j), w_i, label) : label \in \{\text{'fin}', \text{'FIN}'\}$ regardless of whether the client that invoked this operation has failed or not.;

2 For $read()$**:** Query all servers for 'pre' tags. After hearing from a quorum, get the maximal tag $t := (z, j)$. For each server, send $(t, \perp, \text{'fin}')$ and wait for a quorum of replies with the requested coded elements that are associated with t. If at least $k_{threshold}$ replies include coded elements so that it is possible to decode them, return the decoded value. Otherwise, return $\perp$;

3 **The server:** *Upon query arrival from p_j's client to p_i's server.* If p_j's client is a reader, acknowledge with $(t, \perp, \text{'qry}'))$, where t is the maximal tag of any finalized stored record. Else, acknowledge using t that is the maxim tag that any stored record.

4 *Upon pre-write* $(t, w, \text{'pre}')$ *arrival from the p_j's writer.* Make sure that the stored record include the coded element w and acknowledge using $(t, \perp, \text{'pre}')$.

5 *Upon finalize or FINALIZE* $(t, \perp, d) : d \in \{\text{'fin}', \text{'FIN}'\}$ *arrival from p_j's client to p_i's server.* If (t, w, d) is stored and p_j's client is a reader, then acknowledge using (t, w, d). Else, acknowledge using $(t, \perp, d)$.

6 *Upon gossip* $(pre[j], fin[j], FIN[j])$ *arrival from p_j's server to p_i's server.* Integrate the arriving information with the stored one by making sure that $pre[j]$, $fin[j]$ and $FIN[j]$ are not greater than any of the tags of the stored records with 'pre$'$, 'fin$'$, and respectively, 'FIN$'$ phases. In case there is a quorum of gossip records with the phase 'fin$'$, update the phase label to 'FIN$'$. The updated phase value later allows the servers to consider this record as a candidate for garbage collection (when it becomes not among the δ newest records of phase 'FIN$'$). Gossip to all servers $(pre[i], fin[i], FIN[i])$, where $pre[i]$, $fin[i]$ and $FIN[i]$ are the greatest stored tags of stored records with 'pre$'$, 'fin$'$, and respectively, 'FIN$'$ phases.

that the algorithm clearly tolerates any writer failure (crash) whenever either no server or a quorum receives the finalize message. To the end of establishing the viability of a write operation that only some servers (but not a quorum) store a finalized record, the algorithm employs a reliable gossip mechanism for disseminating among the servers tags of finalized records. This dissemination is invoked once for any arriving finalized message.

2.2 Self-stabilizing CAS

The variation of CAS from [5,6] is both self-stabilizing and privacy-preserving. Our pilot implementation, which we call CASSS (CAS Self-Stabilizing), we only focus on the algorithms self-stabilizing ability (i.e., it's ability to recover after the occurrence of a transient fault). This is modelled by considering transient faults that can corrupt arbitrarily the system state (as long as the program code stays intact). Moreover, it is assumed that the system starts after the last occurrence of these failures [8,16].

1. In the starting system state, the server at node p_i may store tag $t_{\max}$ (in a record that has its phase set to either 'pre' or 'fin'), such that due to the system asynchronous nature, it is not retrieved by any query for an arbitrarily long period. The challenge is to bound the number of write operations in which stale information, such as $t_{\max}$'s record, may reside at the system without having a write that hides $t_{\max}$.
2. Self-stabilizing (reliable) end-to-end communications require that the underlying channels have bounded capacities [16, Chapter 3.2]. Thus, in the context of self-stabilization in asynchronous systems, the quorums that send acknowledgments to the clients might complete write operations at a faster rate than the *reliable* gossip service delivers. Therefore, it is not clear how the writer can avoid blocking in a self-stabilizing system where its channels are bounded (and still deliver all messages).
3. All variables must be bounded, including, for example, the tag values. This means that when the system state encodes the maximum tag values, wrapping around to value zero must not disrupt the algorithm invariants, such as the tags' ability to order events.

Addressing Challenge (1). The servers repeatedly gossips the highest tag value that any server has. Each server includes in these messages the maximum tag that is part of locally stored records, such that their labels are 'pre' and also the maximum tag of records with the labels 'fin' and 'FIN'. Also, any write operation queries for the highest 'pre' tag so that the new tag of this operation is greater than all the (possibly corrupted) pre-write records in the system. (The read procedure is borrowed from CAS.) The correctness proof in [5,6] demonstrates that this modification still preserves atomicity and thus CASSS addresses the first challenge.

Addressing Challenge (2). The proof also shows that the gossip service does not need to guarantee the delivery of all messages and that the message's eventual delivery (or later messages with higher tag values) is sufficient. The server just overwrites the last received message in the buffers when a new gossip message arrives.

Addressing Challenge (3). To bound the storage size for each server, Dolev *et al.* [5] first bound the number of records each server stores and then bound

the tag size. (Note that the client state of CAS is easy to bound and the message size is implied by the bound on its fields.)

Bounding the number of stored records is based in the assumptions that failing clients do not restart and that each client invokes at most one instance of the write procedure. This means that at any time, a client can have at most two relevant records in any server storage (regardless of whether it is failing or not). That is, one of these records might be the one that holds the most recent object value (written by an already completed p_i's operation) and the other record could be of an ongoing p_j's write operation. So, any stored record older than the two most recent records from client p_i is irrelevant, because it is either obsolete or stale. Thus, we can bound the number of relevant records by $2N$, where N is the number of clients. Dolev *et al.* [5] reduce this bound to $N + \delta + 3$ by adding to write procedure a fourth round, labeled by 'FIN', where δ is a bound on the number of read operations that occur concurrently with a write operation.

Bounding the maximum label requires to consider the case in which the system state includes a tag that has reached its overflow value, (MAXINT, $\bullet$). Note that by choosing MAXINT to be a very high value, say, $2^{64}-1$, we can guarantee that such an event happens only after the occurrence of a transient fault (because counting from zero to MAXINT takes much longer time than the lifetime of all relevant practical systems). As an extension to Algorithm 1, Dolev *et al.* [5] propose to let the servers detect the presence of this overflow value and then to stop responding to queries while keeping the gossip service running. By that, the servers disseminate the overflow values in the system while abstaining from supporting new operations from installing pre-write records. This continues until the servers detect, via gossip, that all of them have the same maximum finalized tag value, $t_{\max}$. At that point, the algorithm in [5] invokes a self-stabilizing (graceful) counter restart that allows the preservation of the object value using an agreement protocol (Sect. 3.3). During the counter restart, all clients are forced to perform also a local reset, which causes the abortion of all ongoing operations. Once the agreement procedure is terminated, the servers empty their local storages while keeping only the most recent finalized record and replacing its tag $t_{\max}$ with the initial tag value before resuming operation.

3 Implementation

We call our system CASSS, for CAS Self-Stabilizing. The CASSS pilot was implemented as a library in Python, which can be used by applications in order to provide access to the read and write operations. Calls to the functions `read()` and `write(x)` should behave as if the service was an actual shared memory. Calls to these functions block the calling process until the call returns. A successful read operation returns the data object, and a successful write operation blocks until it is done writing the object (and returns nothing). We proceed to describe the building blocks of the system.

3.1 Gossip and Quorum Communications

We used a self-stabilizing version of the token passing algorithm of [16, Figure 4.1] using UDP/IP as the basis for implementing the gossip and quorum services. CASSS requires the use of a self-stabilizing gossip protocol between servers to periodically share the largest tags for each phase. We used UDP/IP and let the arriving gossip messages overwrite the old ones (even if the old ones were not delivered). Our self-stabilizing quorum system follows the one in [5]. For the sake of improved performance, whenever it was required to transfer large data objects, a new TCP/IP connection was established. Our pilot implementation simply used a configuration file for retrieving the list of available storage servers (rather than an external directory service like DNS), for the sake of simple presentation.

3.2 Reincarnation Service

CAS assumes that clients cannot resume after failing. CASSS includes an extension that allows clients to reincarnate [5]. This is based on extending the client identifier to *uid*, which consists of a unique hardware address and an incarnation number.

The client algorithm performs a periodic task that starts with a query phase to check if its current incarnation number is up to date. It does this by querying all servers and awaits responses from a quorum of servers. The maximum value of all received incarnation numbers is calculated, and if that number differs from the current client incarnation number, a second phase is triggered. During the second phase, the incarnation number is updated both at the client side and in the quorum system. The client takes the maximum of the current incarnation number and all received incarnation numbers, increments that by one and sends it out to all servers. After receiving a quorum of acknowledgements, the client knows that it has been assigned a new valid incarnation number and can thus proceed to operate as usual by updating its *uid* accordingly.

3.3 Graceful Global Counter Restart

We use a graceful reset mechanism for restarting sequence numbers (of tags and incarnation numbers). The algorithm facilitates a wraparound based on the ability to achieve agreement and thus we assume that all servers are alive, *e.g.,* via a self-stabilizing service for quorum reconfiguration [7]. We further borrow ideas from [7, Algorithm 3.1] for performing a global counter restart while preserving the recent object value and a mechanism for recovering from transient faults. The algorithm can be extended to detect failures, and hence not requiring all servers to be alive in order to restart, in partially synchronous settings using the failure detection mechanism of [7]; including such discussion would make the presentation of the algorithm more difficult to follow, without contributing directly to our experimental evaluation.

4 Evaluation Methodology

To the end of evaluating our implementation, we have experimented on a true-to-life distributed system (rather than injecting faults or simulating the system). The implementation code can be accessed via www.self-stabilizing-cloud.net.

4.1 Evaluation Criteria and Platform

A common evaluation criteria in the field is to measure operation latency—the average time it takes for an operation to complete [13]. This includes both communication delay and local processing time. The operation latency is measured both in isolated settings (where no other client is making any requests), and in settings where we have different levels of base load on the servers. For comparison, we have implemented both CAS and CASSS, as well as a MW-ABD, using a self-stabilizing quorum system. We used the PlanetLab-EU platform (www.planet-lab.eu) to have a true-to-life, large-scale distributed system to run the evaluation on.

4.2 Experiment Scenarios

In this section, we describe the experiment settings, and how we measure performance before the details of each experimental scenario.

Baseline Settings. For unifying the evaluation, we often use the same baseline for each of the experiments (unless otherwise noted). The setting that all experiments proceed from is to have 15 machines in total, ten of which run one server process each and five of which run one client process each. When increasing the number of clients or servers beyond the number of physical machines, multiple instances are put on the same physical machine. In order to guarantee a fair latency between a client and a server instance, clients processes are never placed on the same physical machine as server processes. More clients or servers than available nodes are distributed in a round-robin fashion. Operations of a client are invoked sequentially with a random delay in between.

The system is initialized by a 512 KiB data object with random data being written to the quorum system before the experiments start. Each client repeats the operation 50 times, and the fastest and slowest operations are removed in order to mitigate the effect of outliers (by pre-experiment evaluations we were able to identify 50 as a reasonable number, where experiments would complete in reasonable time, while giving consistent results). The final operation latency result is the average of every client's average operation latency. Taking the average over all clients accounts for local variations, which is important since different PlanetLab nodes have different conditions. PlanetLab servers do not have any uptime guarantees, and we, therefore, want to allow a few servers to fail (*i.e.,* $f > 0$). But because k is bounded to be an integer value, such that $1 \leq k \leq N - 2f$, the value of f cannot be chosen freely. It, therefore, stands

clear that if f is constant, N can never be chosen such that k would be forced to be less than one. Therefore, since we want to run an experiment with as few as five servers, we have chosen $f = 2$.

Client Scalability Experiment. This scenario is made to evaluate how the read and write latencies are affected when increasing the number of writers and readers respectively. This tests the servers' ability to handle an increase of concurrent operations. The number of failing nodes tolerated (f) is kept constant, *i.e.,* the quorum size is also constant. Both the reads and writes latency is measured. For reader scalability, we consider 5, 10, 15, 20, 30 and 40 readers, while having 10 writers and 10 servers. Corresponding numbers are used for write operation scalability.

Server Scalability Experiment. The server scalability experiment is constructed to evaluate in what way the read and write latencies are affected when increasing the number of servers. The number of failing nodes tolerated is kept constant, *i.e.,* the quorum grows with the number of servers. So when the servers increase, the number of servers that a client has to access will also increase but the coded elements will be smaller. One interesting aspect to look at when increasing the number of servers is whether the effect of a higher code rate trumps the effects of having a larger quorum. Both read and write latencies are measured. We use 5, 10, 15, 20 and 30 servers while having 10 readers and 10 writers.

Data Object Scalability Experiment. For evaluating how the read and write latencies are affected by the object size, this experiment performs operations using increasingly large data objects. The size is increased to a maximum of 4 MiB, which was found to be enough to demonstrate the scalability. In particular, we consider objects of size 1, 32, 128, 512, 1024, 2048 and 4096 KiB. The number of failing nodes tolerated is kept constant ($f = 2$), as well as the number of servers (10), which means that the quorum size is also constant. The experiment is run in isolation from other client nodes, so that scalability in increasing object sizes can be reliably measured. Both the read and write latencies are measured.

Counter Restart Experiment. This scenario measures how long it takes for the servers to restart their local state after a transient fault. Since this part requires the participation of all servers, we do not allow any server to be unresponsive (*i.e.,* $f = 0$). Because some nodes on PlanetLab were highly unstable, it was hard to run experiments for prolonged stretches of time. Therefore, we limited the number of repetitions for the counter restart experiment (which was expected to take longer than the other experiments) to 20 instead of 50. For the same reason, we restricted the object size to 0.25 KiB. Having to restart the global system state is the worst case scenario when it comes to recovery after a transient fault. The time measured is from a client pre-write phase (with a maximal tag number) until a query ends successfully. As discussed, we set $f = 0$, in

order to know that every server has finished the reset phase, meaning the client has to receive responses from all servers before returning.

Overhead Experiment. In this scenario, we compare the overhead of CASSS with our implementation of CAS. In our case, the CAS implementation builds on the CASSS implementation, but does not include the fourth round ('FIN') nor does it perform any gossiping. In other words, this implementation uses the same number of phases and gossip messages as in [4], but, for a fair comparison, it is based on the same software components as the CASSS implementation. Here we use 10 servers (with $f = 2$), one writer and one reader.

5 Evaluation Results

Client Scalability. Figure 1(a) shows the result of the experiment where the number of concurrent readers was changed, and Fig. 1(b) the corresponding experiment for number of concurrent writers. Both charts shows a rather flat curve, which indicates that none of the experiments reached a point where the system was overwhelmed by the number of concurrent operations.

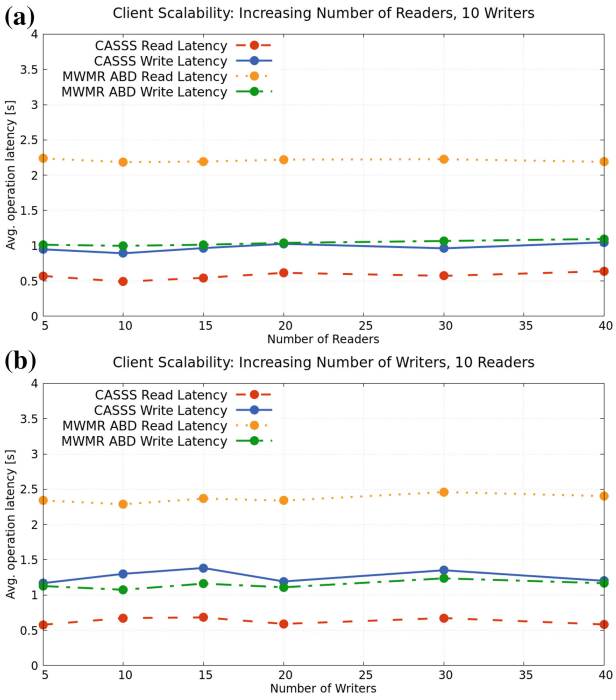

Fig. 1. Operation latency with respect to the number of concurrent (a) readers and (b) writers.

Note the difference between the operations. The fact that MW-ABD read operation is the slowest of the four is not a surprise. Not only does MW-ABD send larger messages, due to the lack of coding, but also its read operation actually transfers data twice: once to fetch the data from the servers, and once during the propagation phase. The MW-ABD and CASSS complete write operations in about the same amount of time. While CASSS writes has two more communication rounds than MW-ABD writes, MW-ABD messages are larger due to the lack of coding. It seems that, with the relatively short RTT between PlanetLab nodes (≈50 ms avg ping time), the cost of two extra rounds seems to be about as expensive as the cost of larger messages. We find that CASSS reads are the fastest ones. This too was expected, since it has as few rounds as MW-ABD writes, but uses coding which decreases the message size.

Fig. 2. Operation latency with respect to the number of servers. The vertical dashed line denotes the point where the parameter f had to be changed.

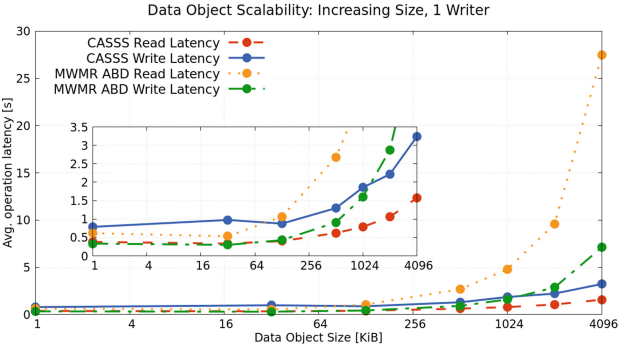

Fig. 3. Operation latency with respect to the size of the data object.

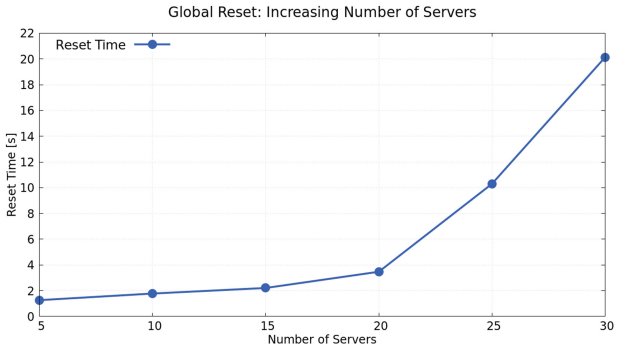

Fig. 4. The time it takes for the Global Reset mechanism to complete, with respect to the number of servers.

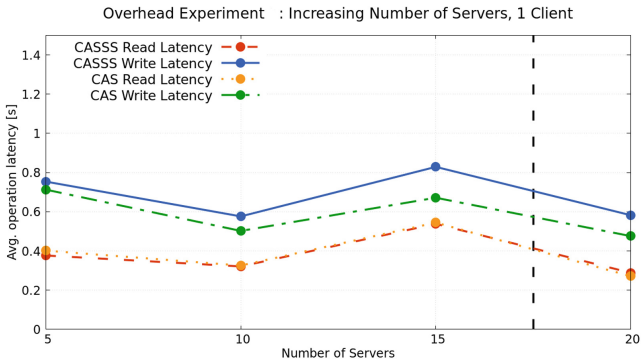

Fig. 5. Comparison between the operation latency of CASSS versus the traditional CAS algorithm. The dashed vertical line denotes the point where the parameter f had to be changed.

Server Scalability. Figure 2 presents the results of the servers scalability experiment. Note that with five servers, both reads and writes of CASSS and MW-ABD writes end up at more or less the same spot. That is because, with only five servers, CASSS effectively performs full replication and the CASSS quorum size is equal to majority quorum. While MW-ABD reads have fewer rounds than CASSS writes, MW-ABD reads transfer more data. This is why it the slowest of all operations.

Looking at the interval between five and ten servers, the operation latency of MW-ABD increases while the operation latency of CASSS decreases or stays the same. That is because when increasing the number of servers, the quorum size grows but so does the code rate. So while both MW-ABD and CASSS waits for responses from more servers, CASSS gains the advantage of decreased message size. The used coding library has a limitation that $k + m \leq 32$. Thus, f could

not be kept at 2 for quorum systems with 20 and 30 servers. For 20 servers, f had to be at least 4, and for 30 servers it had to go all the way up to 14. The point where f is changed is marked by the dashed vertical line.

Data Object Scalability. Figure 3 shows the results of the data object scalability experiment. (Existing solutions [14] show how to transform ABD-like algorithms to more suitable implementations for large data objects.) Up to ca 1 MiB, the operation latency is fairly minimal. MW-ABD begins to escalate at 512 KiB, but CASSS is reasonably fast all the way to 4 MiB. This is of course a consequence of the coding, which reduces the message size.

Global Counter Restart. The global counter restart is triggered only after the occurrence of a transient fault, *i.e.,* it is invoked very rarely. Even so, it is still important that the counter restart terminates within a reasonable amount of time. Figure 4 shows that, for up to 20 servers, the time it takes for the counter restart procedure to finish is equivalent to the time it takes to perform two write operations, *i.e.,* it takes only a few seconds. As the number of servers increases, the likelihood of having to wait for slower servers increases too. If the responsiveness for a server at a given time is normally distributed, the likelihood of having one or more slow servers in the system increases exponentially.

Overhead. Figure 5 depicts the overhead that the extra communication round and intensive gossiping have. The figure has a vertical dashed line, which indicates at which point the variable f was changed due to the coding library requirement discussed previously. Note that CASSS reads and CAS reads are nearly identical. This is exactly what one would expect since CASSS has the same number of rounds for the reads as CAS. The write operations differ slightly, and with CASSS needing one extra communication round to complete the write operation, we expected it to be slightly slower than CAS. The average ping time between the PlanetLab nodes was about 50 ms, so the expected cost for one round is consistent with what we observe in Fig. 5.

6 Conclusion

Our case-study is, to the best of our knowledge, the first work to practically evaluate a system based on a self-stabilizing atomic MWMR coded shared memory emulation, with bounded storage size. We have implemented a system that is based on several self-stabilizing building blocks. This includes both a restart mechanism that performs a synchronized global reset of the entire system in a graceful manner, and a reincarnation number service that provides the failing client another chance to participate. We show that the CASSS system scale very well both in terms of the number of servers and number of concurrent clients. It also scales well with respect to the size of the replicated object. We see that CASSS system has a recovery period of only a few client operations.

Furthermore, it only has a constant overhead compared to the traditional CAS algorithm. This shows that the overhead introduced by self-stabilization can be fairly small, and in many cases negligible – especially when considering the upside of handling transient faults. We view this work as a promising first step in developing an efficient self-stabilizing cloud storage service based on atomic coded shared memory emulation.

A natural extension to our work would be the development of a reconfigurable version of CASSS, similar to the proposal in [17]. We see such extensions of the prototype proposed in this paper, as self-stabilizing building blocks for Cloud systems. We note the existence of other such prototypes, *e.g.,* Renaissance [18,19], as well as other algorithms that we propose as prospective candidates for prototyping, such as self-stabilizing Byzantine tolerant replicated state-machine [20] and self-stabilizing (Byzantine tolerant) end-to-end communications [21,22].

References

1. Herlihy, M.P., Wing, J.M.: Linearizability: a correctness condition for concurrent objects. ACM Trans. Program. Lang. Syst. **12**(3), 463–492 (1990)
2. Attiya, H., Bar-Noy, A., Dolev, D.: Sharing memory robustly in message-passing systems. J. ACM **42**(1), 124–142 (1995)
3. Lynch, N., Shvartsman, A.: Communication and data sharing for dynamic distributed systems. In: Schiper, A., Shvartsman, A.A., Weatherspoon, H., Zhao, B.Y. (eds.) Future Directions in Distributed Computing. LNCS, vol. 2584, pp. 62–67. Springer, Heidelberg (2003). https://doi.org/10.1007/3-540-37795-6_12
4. Cadambe, V.R., Lynch, N., Medard, M., Musial, P.: A coded shared atomic memory algorithm for message passing architectures. Distrib. Comput. **30**(1), 49–73 (2017)
5. Dolev, S., Petig, T., Schiller, E.M.: Self-stabilizing and private distributed shared atomic memory in seldomly fair message passing networks. CoRR abs/1806.03498 (2018)
6. Dolev, S., Petig, T., Schiller, E.M.: Brief announcement: robust and private distributed shared atomic memory in message passing networks. In: ACM Symposium on Principles of Distributed Computing, PODC, pp. 311–313. ACM (2015)
7. Dolev, S., Georgiou, C., Marcoullis, I., Schiller, E.M.: Self-stabilizing reconfiguration. In: Networked Systems NETYS, pp. 51–68 (2017)
8. Dijkstra, E.W.: Self-stabilizing systems in spite of distributed control. Commun. ACM **17**(11), 643–644 (1974)
9. Attiya, H.: Robust simulation of shared memory: 20 years after. Bull. EATCS **100**, 99–113 (2010)
10. Lynch, N., Shvartsman, A.A.: RAMBO: a reconfigurable atomic memory service for dynamic networks. In: Malkhi, D. (ed.) DISC 2002. LNCS, vol. 2508, pp. 173–190. Springer, Heidelberg (2002). https://doi.org/10.1007/3-540-36108-1_12
11. Musial, P.M., Nicolaou, N.C., Shvartsman, A.A.: Implementing distributed shared memory for dynamic networks. Commun. ACM **57**(6), 88–98 (2014)
12. Cadambe, V.R., Nicolaou, N.C., Konwar, K.M., Prakash, N., Lynch, N.A., Médard, M.: ARES: adaptive, reconfigurable, erasure coded, atomic storage. CoRR abs/1805.03727 (2018)
13. Nicolaou, N.C., Georgiou, C.: On the practicality of atomic MWMR register implementations. In: 10th IEEE Parallel and Distributed Processing with Applications, ISPA, pp. 340–347 (2012)

14. Fan, R., Lynch, N.: Efficient replication of large data objects. In: Fich, F.E. (ed.) DISC 2003. LNCS, vol. 2848, pp. 75–91. Springer, Heidelberg (2003). https://doi.org/10.1007/978-3-540-39989-6_6
15. Lynch, N.A., Shvartsman, A.A.: Robust emulation of shared memory using dynamic quorum-acknowledged broadcasts. In: 27th Fault-Tolerant Computing, pp. 272–281 (1997)
16. Dolev, S.: Self-Stabilization. MIT Press, Cambridge (2000)
17. Dolev, S., Georgiou, C., Marcoullis, I., Schiller, E.M.: Self-stabilizing reconfiguration. In: El Abbadi, A., Garbinato, B. (eds.) NETYS 2017. LNCS, vol. 10299, pp. 51–68. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-59647-1_5
18. Canini, M., Salem, I., Schiff, L., Schiller, E.M., Schmid, S.: A self-organizing distributed and in-band SDN control plane. In: ICDCS, IEEE Computer Society, pp. 2656–2657 (2017)
19. Canini, M., Salem, I., Schiff, L., Schiller, E.M., Schmid, S.: Renaissance: a self-stabilizing distributed SDN control plane. In: ICDCS, IEEE Computer Society, pp. 233–243 (2018)
20. Dolev, S., Georgiou, C., Marcoullis, I., Schiller, E.M.: Self-stabilizing byzantine tolerant replicated state machine based on failure detectors. In: Dinur, I., Dolev, S., Lodha, S. (eds.) CSCML 2018. LNCS, vol. 10879, pp. 84–100. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-94147-9_7
21. Dolev, S., Liba, O., Schiller, E.M.: Self-stabilizing byzantine resilient topology discovery and message delivery. In: Gramoli, V., Guerraoui, R. (eds.) NETYS 2013. LNCS, vol. 7853, pp. 42–57. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-40148-0_4
22. Dolev, S., Hanemann, A., Schiller, E.M., Sharma, S.: Self-stabilizing end-to-end communication in (bounded capacity, omitting, duplicating and non-FIFO) dynamic networks. In: Richa, A.W., Scheideler, C. (eds.) SSS 2012. LNCS, vol. 7596, pp. 133–147. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-33536-5_14