Optimistic Erasure-Coded Distributed Storage[*]

Partha Dutta[1], Rachid Guerraoui[2], and Ron R. Levy[2]

[1] IBM India Research Lab, Bangalore, India
[2] EPFL IC, Lausanne, Switzerland

Abstract. We study erasure-coded atomic register implementations in an asynchronous crash-recovery model. Erasure coding provides a cheap and space-efficient way to tolerate failures in a distributed system. This paper presents ORCAS, Optimistic eRasure-Coded Atomic Storage, which consists of two separate implementations, ORCAS-A and ORCAS-B. In terms of storage space used, ORCAS-A is more efficient in systems where we expect large number of concurrent writes, whereas, ORCAS-B is more suitable if not many writes are invoked concurrently. Compared to replication based implementations, both ORCAS implementations significantly save on the storage space. The implementations are optimistic in the sense that the used storage is lower in synchronous periods, which are considered common in practice, as compared to asynchronous periods. Indirectly, we show that tolerating asynchronous periods does not increase storage overhead during synchronous periods.

1 Introduction

1.1 Motivation

Preventing data loss in storage devices is one of the most critical requirements in any storage system. Enterprise storage systems in particular have multiple levels of redundancy built in for fault tolerance. The cost of a specialized centralized storage server is very high and yet it does not offer protection against unforseen consequences such as fires and floods. Distributed storage systems based on commodity hardware, as alternatives to their centralized counterparts, have gained in popularity since they are cheaper, can be more reliable and offer better scalability. However, implementing such systems is more complicated due to their very distributed nature.

Most existing distributed storage systems rely on data replication to provide fault tolerance [15]. Recently however, it has been argued that erasure coding is a better alternative to data replication since it reduces the cost of ensuring fault tolerance [7, 8]. In erasure-coded storage systems, instead of keeping an identical version of a data V on each server, V is encoded into n fragments such that V can be reconstructed from any set of at least k fragments (called k-of-n encoding), where the size of each fragment is roughly $|V|/k$. A different encoded

[*] Part of this work was done when Partha Dutta and Ron R. Levy were at Bell Labs Research, India.

fragment is stored on each of the n servers, and ideally such a system can tolerate the failure of $f = n - k$ servers.

The main advantage of erasure-coded storage over replicated storage is its *storage usage*, i.e., less storage space is used to provide fault tolerance. For instance, it is well-known that a replicated storage system with 4 servers can tolerate at most 1 failure in an asynchronous environment. If each server has a storage capacity of 1 TB, the total capacity of the replicated storage system is still 1 TB. In this case the storage overhead (total capacity/useable capacity) is 4, i.e. only 1/4 of the total capacity is available. Erasure coding allows the reduction of this overhead to 2 in an asynchronous system, i.e., makes 2 TB useable. In a synchronous system (with 4 servers and at most 1 failure), it is even possible to further reduce this overhead and make 3 TB available to the user. Clearly, the synchronous erasure-coded storage is more desirable in terms of storage usage. Unfortunately, synchrony assumptions are often not realistic in practice. Even if we expect the system to be synchronous most of the time, it is good practice to tolerate asynchronous periods. The idea underlying our contribution is the common practice of designing distributed systems that can cope with worst case conditions (e.g., asynchrony and failures) but are optimized for best case situations (e.g., synchrony and no failures) that are considered common in practice.

1.2 Contributions

In this paper we investigate one of the fundamental building blocks of a fault-tolerant distributed storage – multi-writer multi-reader atomic register implementations [3, 13, 15]. An atomic register is a distributed data-structure that can be concurrently accessed by multiple processes and yet provide an "illusion" of a sequential register. (A sequential register is a data-structure that is accessed by a single process with read and write operations, where a read always returns the last value written.) We consider implementations over a set of n server processes in an asynchronous crash-recovery message-passing system where (1) each process may crash and recover but has access to a stable storage, (2) in a run, at most f out of n servers are faulty (i.e., eventually crash and never recover), and (3) channels are fair-lossy.

We present two wait-free atomic register implementations ORCAS-A and ORCAS-B (Optimistic eRasure-Coded Atomic Storage). Our implementations are the first wait-free atomic register implementations in a crash-recovery model that have an "optimistic" (stable) storage usage. Suppose that all possible write values are of a fixed size Δ.[1] Then in both of our implementations, during synchronous periods with q alive (non-crashed) servers and when there is no write operation in progress, the stable storage used at every alive server is $\frac{\Delta}{q-f}$, whereas during asynchronous periods when there is no write operation in progress, the storage used is $\frac{\Delta}{n-2f}$ at all but f servers (in ORCAS-A, at most f servers may use

[1] Through out the paper we assume that, other than the write value and its encoded fragments, all other values (e.g., timestamp) at a server are of negligible size.

Δ). However, the two implementations differ in their storage usage when there is a write in progress. In ORCAS-A, when one or more writes are in progress, the storage used at a server can be Δ, but even in the worst-case the storage used is never higher than Δ. In contrast, if there are w concurrent writes in progress in ORCAS-B then, in the worst-case, the storage used at a server can be $\frac{w\Delta}{n-2f}$. Thus in terms of storage space used, ORCAS-A is more efficient in systems where we expect large number of concurrent writes, whereas, ORCAS-B is more suitable if not many writes are invoked concurrently. We also show how the number of messages exchanged in ORCAS-A can be significantly reduced by weakening the termination condition of the read from wait-free to Finite Write (FW) termination [1].[2]

Both ORCAS implementations are based on a simple but effective idea. The write first "gauges" the number of alive servers in the system by sending a message to all servers and counting the number of replies received during a short waiting period. Depending on the number of replies, the write decides how to erasure code its value. Additionally, to limit the communication overhead, the ORCAS implementations ensure that the write value or the encoded fragments are sent to the servers in only one of the phases of a write; later, the servers can locally compute the final encoded fragments on receiving a small message that specifies how the value needs to be encoded (but the message does not contain the final encoded fragments).

In particular, in ORCAS-A, the write sends the unencoded write value to all servers and waits for replies. If it receives replies from q servers, the write sends a message to the servers that requests them to locally encode the received value with $(q-f)$-of-n encoding. (Note that $q \geq n-f$ because at most f servers can be faulty.) Roughly speaking, a subsequent read can contact at least $q-f$ of the servers that reply to the write, and (1) either the read receives an unencoded value from one of those servers, or (2) it receives $q-f$ encoded fragments. In both cases, as the write does a $(q-f)$-of-n encoding, the read can reconstruct the written value. Note that, as $q-f \geq n-2f$, in the worst-case ORCAS-A does a $(n-2f)$-of-n encoding, and in synchronous periods with q alive servers, it does a $(q-f)$-of-n encoding.

On the other hand, ORCAS-B, like previous erasure-coded atomic register implementations, never sends an unencoded write value to the servers. Ideally, to obtain the same storage usage as ORCAS-A, in a write of ORCAS-B we would like to send an $(n-2f)$-of-n encoded fragment to each server, and on receiving replies from q servers, request the servers to keep a $(q-f)$-of-n encoded fragment. However, in general, it is not possible to extract a particular fragment of a $(q-f)$-of-n encoding from a *single* fragment of a $(n-2f)$-of-n encoding. Thus with this naive approach, either a write would need to send another set of fragments to the servers or the servers would need to exchange their fragments, resulting in significant increase in communication overhead. We solve this problem in ORCAS-B by a novel approach of storing multiple, much smaller fragments at

[2] A similar idea was earlier used in [10] where the read satisfied obstruction-free termination.

each server (instead of a single large one). Suppose the write estimates that there are q alive servers. Then it encodes the value with x-of-nz encoding, where x is any common multiple of $n-2f$ and $q-f$, and $z = \frac{x}{n-2f}$, and sends z fragments to each server. If the write receives replies from q servers, then it requests each server to *trim* its stored fragments in the next phase, i.e., retain any $y = \frac{x}{q-f}$ out of its z fragments and delete the rest. Roughly speaking, since a read can miss at most f servers that replied to the write, if a subsequent read sees a trimmed server then it will eventually receive y fragments from at least $q-f$ servers, and if the read does not see a trimmed server, then it will receive z fragments from at least $n-2f$ servers. In both cases, it receives $y(q-f) = z(n-2f) = x$ fragments, and therefore, can reconstruct the written value. The advantage of our approach over the naive approach is that, our approach has the same storage usage as latter, but has lower communication overhead.

The detailed presentation of our algorithms can be found in [6]. Due to space limitations, this paper focusses on ORCAS-A. Also in [6], we show lower bounds on storage space usage in atomic register implementation in synchronous and asynchronous systems, for a specific class of implementations (that include both ORCAS-A and ORCAS-B, and the implementation in [7]). Roughly speaking, we show that implementations – (1) which at the end of a write store equal number of encoded fragments in the stable storage of the servers, and (2) do not use different encoding schemes in the same operation – cannot have a stable storage usage better than ORCAS-A (and ORCAS-B) in either synchronous or asynchronous periods.

1.3 Related Work

Recently there has been lot of work on erasure-coded distributed storage [2, 5, 7, 8, 10, 11]. We discuss below three representative papers that are close to our work.

Frolund et al. [7] describe an erasure-coded distributed storage (called FAB) in the same system model as this paper, i.e., an asynchronous crash-recovery model. The primary algorithm in FAB implements an atomic register. Servers have stable-storage and keep a log containing old versions of the data, which is periodically garbage collected. The main difference with our approach is that in FAB the stable storage is not used optimistically. In particular, ORCAS-B has the same storage overhead as FAB during asynchronous periods (even when writes are in progress) but performs better during synchronous periods. Another difference is that FAB provides strict linearizability, which ensures that partial write operations appear to take effect before the crash or not at all. The price that is paid by FAB is to give up wait-freedom: concurrent operations may abort. ORCAS-B ensures that write operations are at worst completed upon recovery of the writer and guarantees wait-freedom: all operations invoked by correct clients eventually terminate despite the concurrent invocations of other clients.

Aguilera et al. [2] present an erasure-coded storage (that we call AJX) for *synchronous systems* that is optimized for $f << n$. AJX provides the same low storage overhead as ORCAS during failure-free synchronous periods, and performs better than ORCAS when there are failures. However, AJX provides consistency guarantees of only a regular register and puts a limit on the maximum number of client failures. Also, wait-freedom is not ensured since concurrent writes may abort.

Cachin et al. [5] propose a wait-free atomic register implementation for the byzantine model. It uses a reliable broadcast like primitive to disseminate the data fragments to all servers, thus guaranteeing that if one server receives a fragment, then all do. The storage required at the servers when there is no write in progress is $\frac{\Delta}{n-f}$. At first glance, one might be tempted to compare our implementations with a crash-failure restriction of the algorithm in [5], and conclude that our implementations have worse storage requirements in asynchronous periods ($\frac{\Delta}{n-2f}$). However, one of the implications of our lower bounds in [6] is that there is no obvious translation of the algorithm in [5] to a crash-recovery model while maintaining the same storage usage. (We discuss this comparison further in [6].)

2 Model and Definitions

Processes. We consider an asynchronous message passing model, without any assumptions on communication delay or relative process speeds. For presentation simplicity, we assume the existence of a global clock. This clock however is inaccessible to the servers and clients.

The set of servers is denoted by S and $|S| = n$. The j^{th} server is denoted by s_j, $1 \leq j \leq n$. The set of clients is denoted by C and it is bounded in size. Clients know all servers in S, but the set of clients is unknown to the servers. A client or a server is also called a *process*.

Every process executes a deterministic algorithm assigned to it, unless it *crashes*. (The process does not behave maliciously.) If it crashes, the process simply stops its execution, unless it possibly *recovers*, in which case the process executes a *recovery procedure* which is part of the algorithm assigned to it. (Note that in this case we assume that the process is aware that it had crashed and recovered.) A process is *faulty* if there is a time after which the process crashes and never recovers. A non-faulty process is also called a *correct* process. The set of faulty processes in a run is not known in advance. In particular, any number of clients can fail in a run. However, there is an known upper bound $f \geq 1$ on the number of faulty servers in a run. We also assume $f < n/2$ which is necessary to implement a register in asynchronous model.

Every process has a volatile storage and a stable storage (e.g., hard disk). If a process crashes and recovers, the content of its volatile storage is lost but the content of its stable storage is unaffected. Whenever a process updates one of its variables, it does so in its volatile storage by default. If the process decides to store information in its stable storage, it uses a specific operation **store**: we also say that the process *logs* the information. The process retrieves the logged information using the operation **retrieve**.

Fair-lossy channels. We assume that any pair of processes, say p_i and p_j, communicate using fair-lossy channels [4, 14], which satisfies the following three properties: (1) If p_j receives a message m from p_i at time t then p_i sent m to p_j at time t, (2) if p_i sends a message m to p_j a finite number of times, then p_j receives the message a finite number of times, and (3) if p_i sends a message m to p_j an infinite number of times and p_j is correct, then p_j receives m from p_i an infinite number of times.

On top of the fair-lossy channels we can implement more useful stubborn communication procedures (*s-send* and *s-receive*) which are used to send and receive messages reliably [4]. In addition to the first two properties of fair-lossy channels, stubborn procedures satisfy the following third property: If p_i s-sends a message m to a correct process p_j at some time t, and p_i does not crash after time t, then p_j eventually s-receives m. We would like to note that stubborn primitives can be implemented without using stable storage [4].

Registers. A sequential register is a data structure accessed by a single process that provides two operations: write(v), which stores v in the register and returns OK, and read(), which returns the last value stored in the register. (We assume that the initial value of the register is $\perp$, which is not a valid input value for a write operation.) An atomic register is a distributed data-structure that can be concurrently accessed by multiple processes and yet provide an "illusion" of a sequential register to the accessing processes [13, 14]. An algorithm *implements* an atomic register if all runs of the algorithm satisfy the *atomicity* and *termination* properties. We follow the definition of atomicity for a crash-recovery model given in [9], which in turn extends the definition given in [12]. We recall the definition in [6].

We use the following two termination conditions in this paper. (1) An implementation satisfies wait-free termination (for clients) if for every run where at most f of the servers are faulty (and any number of clients are faulty), every operation invoked by a correct client completes. (2) An implementation satisfies Finite-Write (FW) termination [1] if for every run where at most f of the servers are faulty (and any number of clients are faulty), every write invocation by a correct client is complete, and moreover, either every read invocation by a correct client is complete, or infinitely many writes are invoked in the run. (Note that wait-free termination implies FW-termination.)

Erasure coding. A k-of-n erasure coding [17] is defined by the following two primitives:

- **encode**(V, k, n) which returns a vector $[V[1], \ldots, V[n]]$, where $V[i]$ denotes i^{th} encoded fragment. (For presentation simplicity, we will assume that encode returns a *set* of n encoded fragments of V, where each fragment is tagged by its fragment number.)

- **decode**(X, k, n) which given a set X of at least k fragments of V (that were generated by encode(V, k, n)), returns V.

For our algorithms, we make no assumption on the specific implementation of the primitives except the following one: each fragment in a k-of-n encoding of V is roughly of size $|V|/k$.

In the next two sections, we present two algorithms that implement erasure-coded, multi-writer multi-reader, atomic registers in a crash-recovery model, ORCAS-A and ORCAS-B. Both implementations have low storage overhead when no write operation is in progress. The implementations differ in the storage overhead during a write, and in their message sizes.

3 ORCAS-A

We now present our first implementation which we call ORCAS-A. (The pseudocode is given in Figures 1 and 2.) The implementation is inspired by the well-known atomic register implementations in [3, 15]. Also, the registration process of a read at the servers is inspired by the listeners communication pattern in [16]. The first two phases of the write function are similar to that in [3, 15]− they store the unencoded values at $n - f$ (a majority) of servers with an appropriate timestamp. Additionally in ORCAS-A, depending on the number of servers from which the write receives a reply, it selects an encoding r-of-n. Then, the write performs another round trip where it requests the servers to encode the value using r-of-n encoding and retain the fragment corresponding to its server id. The crucial parts of the implementation are choosing an encoding r-of-n and the condition for waiting for fragments at a read, such that, any read can recover the written value without blocking permanently. We now describe the implementation in more detail.

3.1 Description

Local variables. The clients maintain the following local variables: (1) ts: part of the timestamp of the current write operation, and (2) wid, rid: the identifiers of write and read operations, respectively, which are used to distinguish between messages from different operations of the same client, and (3) a timer T_c whose timeout duration is set to the round-trip time for contacting the servers in synchronous periods. The pair $[ts, wid]$ form the timestamp for the current write. The local variables at a server s_j are as follows: (1) A_j: its share of the value stored in the register, which can either be the unencoded value or the j^{th} encoded fragment, (2) τ, δ: the ts and the wid, respectively, associated with the value in A_j, and (3) ρ: the encoding associated with the value in A_j, namely, A_j is the j^{th} fragment of a ρ-of-n encoding of some value. (In particular, $\rho = 1$ implies that A_j contains an unencoded value.)

Write operation. The write operation consists of three phases, where each phase is a round-trip of communication from the client to the servers. The first phase is used to compute the timestamp for the servers, the second phase to write the unencoded value at the servers, and the final phase is used to encode the value at the servers. On invoking a write(V), the client first increments and logs its wid. This helps in distinguishing messages from different operations of the same server even across a crash-recovery. It also logs $ts = 0$ so as to detect an incomplete write across a crash-recovery. Next, the client sends get_ts

```
1:  function initialization:
2:      $ts, wid, rid \leftarrow 0; r \leftarrow 1; T_c \leftarrow timer()$   $\{at\ every\ client\}$
3:      $A_j \leftarrow \perp; \tau, \delta \leftarrow 0; \rho \leftarrow 1$   $\{at\ every\ server\ s_j\}$

4:  function write $(V)$ at client $c_i$
5:      $wid \leftarrow wid + 1$; $ts \leftarrow 0$
6:      store$(wid, ts)$
7:      repeat
8:          send$(\langle get\_ts, wid\rangle, S)$
9:      until s-receive $\langle ts\_ack, *, wid\rangle$ from $n - f$ servers
10:     $ts \leftarrow 1+$ max$\{ts_j$ : s-received $\langle ts\_ack, ts_j, wid\rangle\}$
11:     store$(ts, V)$
12:     trigger$(T_c)$
13:     repeat
14:         send$(\langle write, ts, wid, 0, V\rangle, S)$
15:     until s-receive $\langle w\_ack, ts, wid, 0\rangle$ from $n - f$ servers and expired$(T_c)$
16:     $r \leftarrow$ (number of servers from which s-received $\langle w\_ack, ts, wid, 0\rangle$ messages) $-f$
17:     if $r > 1$ then
18:         repeat
19:             $S' \leftarrow$ set of servers from which s-received $\langle w\_ack, ts, wid, 0\rangle$ until now
20:             send $(\langle encode, ts, wid, r\rangle, S')$
21:         until s-receive $\langle enc\_ack, ts, wid, r\rangle$ from $n - f$ servers
22:     return(OK)

23: upon receive $\langle get\_ts, wid\rangle$ from client $c_i$ at server $s_j$ do
24:     s-send$(\langle ts\_ack, \tau, wid\rangle, \{c_i\})$

25: upon receive $\langle write, ts', wid', rid', V'\rangle$ from client $c_i$ at server $s_j$ do
26:     if $rid' > 0$ then
27:         $\mathcal{R} \leftarrow \mathcal{R} \setminus \{[rid', *, *, i]\}$
28:     if $V' \neq \perp$ then
29:         if $[ts', wid'] >_{lex} [\tau, \delta]$ then
30:             $\tau \leftarrow ts'; \delta \leftarrow wid'; \rho \leftarrow 1; A_j \leftarrow V'$
31:             store$(\tau, \delta, \rho, A_j)$
32:         for all $[rid, ts, id, l] \in \mathcal{R}$ do
33:             s-send$(\langle r\_ack, rid, ts', wid', 1, V'\rangle, \{c_l\})$
34:     s-send$(\langle w\_ack, ts', wid', rid'\rangle, \{c_i\})$

35: upon receive $\langle encode, ts', wid', r'\rangle$ from client $c_i$ at server $s_j$ do
36:     if $[ts', id'] = [\tau, \delta]$ then
37:         $A_j \leftarrow j^{th}$ fragment of encode$(A_j, r', n)$
38:         $\rho \leftarrow r'$
39:         store$(\rho, A_j)$
40:     s-send$(\langle enc\_ack, ts', wid', r'\rangle, \{c_i\})$

41: upon recovery() at server $s_j$ do
42:     $[\tau, \delta, \rho, A_j] \leftarrow$ retrieve()

43: upon recovery() at client $c_i$ do
44:     $[rid, ts, wid, r, V] \leftarrow$ retrieve()
45:     if $ts \neq 0$ then
46:         repeat
47:             send$(\langle write, ts, wid, 0, V\rangle, S)$
48:         until s-receive $\langle w\_ack, ts, wid, 0\rangle$ from $n - f$ servers
```

Fig. 1. ORCAS-A: initialization, write and recovery procedures

messages to all servers and waits until it receives ts from at least $n - f$ servers. (The notation $send(m, X)$ is a shorthand for the following: for every processes $p \in X$, send the message m to p. It is not an atomic operation.) To overcome the effect of the fair-lossy channels, a client encloses the sending of its messages to the servers in a repeat-until loop, and the servers reply back using the s-send primitive. On receiving the ts from at least $n - f$ servers, the client increments

```
1: function read() at client $c_i$
2:     $rid \leftarrow rid + 1$; $\Gamma \leftarrow 0$; $M \leftarrow \emptyset$; $once \leftarrow false$
3:     store($rid$)
4:     repeat
5:         send($\langle read, rid \rangle, S$)
6:         $\mathcal{M} \leftarrow \{msg = \langle r\_ack, rid, *, *, *, * \rangle$ : s-received $msg\}$
7:         $TS \leftarrow \max_{lex}\{[ts, id]$ : $\langle r\_ack, rid, ts, id, *, * \rangle \in \mathcal{M}\}$
8:         if ($\mathcal{M}$ contains messages from at least $n - f$ servers) and ($once = false$) then
9:             $\Gamma \leftarrow TS$; $once \leftarrow true$
10:            if $TS = [0, 0]$ then return($\perp$)
11:    until ($once = true$) and ($\exists$ $r', ts', id'$ such that ($[ts', id'] \geq_{lex} \Gamma$) and ($|\{A_j$ : $\langle r\_ack, rid, ts', id', r', A_j \rangle \in \mathcal{M}\}| \geq r'$))
12:    $\mathcal{A} \leftarrow$ set of $A_j$ satisfying the condition in line 11
13:    if $r' = 1$ then
14:        $V \leftarrow$ any $A_j$ in $\mathcal{A}$; $V' \leftarrow V$
15:    else
16:        $V \leftarrow$ decode($\mathcal{A}, r', n$); $V' \leftarrow \perp$
17:    repeat
18:        send($\langle write, ts', id', rid, V' \rangle, S$)
19:    until s-receive $\langle w\_ack, ts', id', rid \rangle$ from $n - f$ servers
20:    return($V$)

21: upon receive $\langle read, rid \rangle$ from client $c_i$ at server $s_j$ do
22:    if $\mathcal{R}$ does not contain any $[rid, *, *, i]$ then
23:        $\mathcal{R} \leftarrow \mathcal{R} \cup [rid, \tau, \delta, i]$
24:        s-send($\langle r\_ack, rid, \tau, \delta, \rho, A_j \rangle, \{c_i\}$)
```

Fig. 2. ORCAS-A: read procedure

by one the maximum ts received, to obtain the ts for this write. It then logs ts and V so that in case of a crash during the write, the client can complete the write upon recovery. Next, it starts its timer, and sends a *write* message with the timestamp $[ts, wid]$ and the value V, to the servers. (To distinguish this message from the *write* message sent by a read operation, the message also contains a *rid* field which is set to 0.) A server on receiving a *write* message with a higher timestamp than its current timestamp $[\tau, wid]$, updates A_j, τ and δ to V, ts and wid of the message, respectively. It also updates the encoding ρ to 1 (to denote that the contents of A_j is unencoded), and logs the updated variables. (The server also sends some message to the readers which we will discuss later.) The client waits until it receives *w_ack* messages from at least $n - f$ servers, and the timer expires. (Waiting for the timer to expire ensures that the client receives a reply from all non-crashed processes in synchronous periods.)

Next, the client select the encoding for the write to be $r = q - f$, where q is the number of *w_ack* messages received by the client. Note $r \geq 1$ because $q \geq n - f$ and $f < n/2$. Then the client sends an *encode* message to all servers which have replied to the *write* message. A server s_j on receiving this message encodes it value A_j using r-of-n encoding, and retains only the j^{th} fragment in A_j. It also updates its encoding ρ to r, logs A_j and ρ, and replies to the client. The client returns from the write on receiving $n - f$ replies. (Note that the encode phase is skipped if $r = 1$, because 1-of-n encoding is same as not encoding the value at all.)

Read operation. The read operation consists of two phases. The first phase gathers enough fragments to reconstruct a written value, and the second phase writes back the value at the servers to ensure that any subsequent reader does not read an older value.

On invoking a read, the client increments and logs its *rid*. It then sends a *read* message to the servers. On receiving a *read* message, a server *registers* the read[3] by appending it to a local list $\mathcal{R}$ with the following parameters: the *rid* of the *read* message, and the timestamp $[\tau, \delta]$ at the server when the *read* message was received. (The client *de-registers* in the second phase of the read: line 27, Figure 1.) The server then replies with its current value of A_j and its associated timestamp and encoding. In addition, whenever the server receives a new *write* message with a higher timestamp, it forwards it to its registered readers. The client on the other hand, first chooses a timestamp Γ which is greater than or equal to the timestamp seen at $n - f$ processes,[4] and then waits for enough fragments to reconstruct a written value that has an associated timestamp greater than or equal to Γ: the condition in line 11 of Figure 2 simply requires that (1) the client receives r_ack from at least $n-f$ servers, and (2) there is an encoding r' and timestamp $[ts', id']$ such that the client has received at least r' fragments of the associated value, and $[ts', id']$ is greater than or equal to Γ. In [6], we show that this condition is eventually satisfied for every read whose invoking client does not crash.

The second phase of a read is very similar to the second phase of a write except for the following case. If the read selects a value in the first phase that was encoded by the corresponding write ($r' > 1$), then the read does not need to write back the value to the servers because the write has already completed its second phase. In this case, the second phase of the read is only used to deregister the read at the servers.

Recovery Procedures. The recovery procedure at a server is straightforward: it retrieves all the logged values. The client, in addition to retrieving the logged values, also completes any incomplete write. (Note than, even if the last write invocation, before the crash at a client, is complete, *ts* can be greater than 0. In this case, the recovery procedure tries to rewrite the same value with the same timestamp. It is easy to see that this attempt to rewrite the value is harmless.)

3.2 Correctness

The proof of the atomicity of ORCAS-A is similar to the implementations in [3, 15]. The only non-trivial argument in the proof of wait-free termination is proving that the waiting condition in line 11 in Figure 2 eventually becomes true in every run where the client does not crash after invoking the read. In this section, we give an intuition for this proof by considering a simple case where a (possibly incomplete) write is followed by a read, and there are no other operations.

Suppose there is a write(V) that is followed by a read(). We claim that the read() can always reconstruct V or the initial value of the register, and it can

[3] When there is no ambiguity, we also say that the server registers the client.

[4] The Γ selected in this way is higher than or equal to the timestamp of all preceding writes because two server sets of size $n - f$ always has a non-empty intersection.

always reconstruct V if the write is complete. The write() operation has two phases that modify the state of the servers: the write phase and the encode phase. Suppose that during the write phase, the writer receives replies from q servers (denoted by set Q) such that $q \geq n - f > f$. If the writer fails without completing this phase, the read() can return the initial value of the register, which does not violate atomicity. In the encode phase, an r-of-n encode message is sent to all servers, where $r = q - f \geq n - 2f > 0$. If the writer crashes, this message reaches an arbitrary subset of servers. Subsequently, the read() contacts a set R containing at least $n - f$ servers. We denote the intersection of the read and write sets, by U, i.e. $U = Q \cap R$, and it follows that $|U| \geq q - f = r > 0$. There are two cases:

Case 1: There is at least one server in U which still has the unencoded value V. The read can thus directly obtain V from this server.

Case 2: All the servers in U have received the encode message and encoded V. Since $|U| \geq r$ and an r-of-n erasure code was used, there are enough fragments for the read to reconstruct V.

However, we must also consider the case where the read() is concurrent with multiple writes. If there is a series of consecutive writes, the write procedure ensures that all values are eventually encoded. If the read is slow, it could receive an encoded fragment of a different write from each server, making it impossible for the read to reconstruct any value. But the reader registration ensures that the servers will send all new fragments to the reader until the reader is able to reconstruct some written value. A detailed proof of wait-freedom is given in [6].

3.3 Algorithm Complexity

In this section we discuss the theoretical performance of ORCAS-A.

Timing guarantees. For timing guarantees we consider periods of a run where links are timely, local computation time is negligible, at least $n - f$ servers are alive, and no process crashes or recovers. It is easy to show that a write operation completes in three round-trips (i.e., six communication steps), as compared to two round-trips in the implementation of [15]. (We discuss this comparison further in Section 5.) Also it is straightforward to show that a read can complete in two round-trips if there is no write in progress. In [6], we show that even in the presence of concurrent writes, the read registration ensures that a read operation terminates within five communication steps.

Messages. Except the *r_ack* messages, the number of messages used by an operation is linear in the number of servers. In [6], we show how to circumvent the reader registration by slightly weakening the termination condition of the read. Message sizes in ORCAS-A are as large as those in the replication based register implementations of [3, 15]: the first phase of the write in ORCAS-A sends the unencoded value to all servers.

Worst-case bound on storage. Suppose that all possible write values are of a fixed size Δ, and the size of variables, other than those containing a value of a write operation or an encoded fragment of such a value, is negligible. Consider a partial run *pr* that has no incomplete write invocation. (An invocation that has no matching return event in the partial run is called incomplete.) The r computed in line 16 of Figure 1 of every write is at least $n-2f$. Thus, every encoded fragment is at most of size $\Delta/(n-2f)$. Since, there are no incomplete write invocation in *pr*, and every write encodes the value at $n-f$ processes before it returns, the size of (stable) storage at $n-f$ servers is at most $\Delta/(n-2f)$ at the end of *pr*. In addition, note that the size of the storage at *all* servers is *always* bounded by Δ. This is in contrast to the implementation in [7] and ORCAS-B implementation that we describe later, where the worst-case storage size is dependent on the maximum number of concurrent writes.

Bound on storage in synchronous periods. Consider a partial run *pr* which has no incomplete write invocation. Let *wr* be the write with the highest timestamp in *pr*. Let t be the time when *wr* was invoked. Now, assume that (1) the links were timely in *pr* from time t onwards, and (2) at least $n-f$ servers are alive at time t, and no process crashes or recovers from time t onwards. Let $q \geq n-f$ be the set of servers that are alive at time t. Then, it is easy to see that the r computed in line 16 of Figure 1 is $q-f$ in *wr*, and hence, the size of storage at *all* alive servers is at most $\Delta/(q-f)$ at the end of *pr*. It also follows that, if *pr* is a synchronous failure-free partial run, then the size of storage at all servers is at most $\Delta/(n-f)$ at the end of *pr*.

FW-termination. Consider the case in the above implementation when a client invokes a read, registers at all the servers, and then crashes. If a server does not crash, its s-send module will send the *r_ack* message to the client forever. Since, these messages are of large sizes, it may significantly increase the load on the system. Following [1], we show in [6] that if we slightly weaken the waif-free termination condition of the read to Finite-Write (FW) termination, then such messages are not required.

4 ORCAS-B

Although the ORCAS-A implementation saves storage space in synchronous periods, it has two important drawbacks because it sends the unencoded values to the servers in the first phase of the write. First, it uses larger messages compared to implementations which never send any unencoded values to the servers. Second, if a client crashes before sending an *encode* message during a write, servers are left with an unencoded value in the stable storage.[5] In this section, we present our second implementation, ORCAS-B, which like most erasure-coded register implementations, never sends an unencoded value to the servers.

Due to lack of space, we discuss only those parts of ORCAS-B that significantly differ from ORCAS-A. (The pseudocode is presented in [6].) The crucial

[5] In practice, the second case might not cause a significant overhead because any subsequent complete write will erase such unencoded values.

difference between ORCAS-A and ORCAS-B is how the write value is encoded during a write and how it is reconstructed during a read. In ORCAS-B, the write consists of three phases. The first phase finds a suitable timestamp for the write, and tries to guess the number of alive servers, say r'. The write then encodes the value such that the following three conditions holds. (1) If the second phase of the write succeeds in contacting only $n - f$ servers (a worst-case scenario), a subsequent read can reconstruct the value. (2) If the second phase succeeds in contacting r' servers (the optimistic case), then in the third phase, the write can "trim" (i.e., reduce the size of) the stored encoded value at the servers, and still a subsequent read can reconstruct the value. (3) The size of the stored encoded value at a server should be equal to the size of a fragment in $(n-2f)$-of-n encoding in the first case, and $(r' - f)$-of-n encoding in the second case. The motivation behind these three conditions is to have the same optimistic storage requirements as in ORCAS-A.

It is not difficult to see that if the write uses a $(n - 2f)$-of-n encoding, then a server cannot *locally* extract its trimmed fragment in the third phase from the encoded fragment it receives in the second phase, without making extra assumptions about n or the erasure coding algorithm. Thus with $(n - 2f)$-of-n encoding in the second phase, in the third phase of the write, either the write needs to send the trimmed fragment to each server, or the servers need to exchange their (second-phase) fragments. In ORCAS-B we avoid this issue by simply storing multiple fragments at a server, while still satisfying the three conditions above.

We define the following variables: (1) $r = r' - f$, (2) x be the lcm (least common multiple) or r and $n-2f$, (3) $z = x/(n-2f)$, and (4) $y = x/r$. Now the second phase of the write encodes the value using x-of-(nz) encoding. It then tries to store z fragments at each server. If the write succeed in storing the fragments at r' servers, then in the next phase, it sends a *trim* message that requests the servers to retain y out of its z fragments (and delete the remaining fragments). Now it is easy to verify the above three conditions. If the second phase of the write stores the fragments at $n - f$ servers, a subsequent read can access at least $n - 2f$ of those servers, and thus receive at least $(n - 2f)z = x$ fragments. On the other hand, if the stored fragments at some server are trimmed, then at least r' servers have at least y fragments, and therefore a subsequent read receives y fragments from at least $r' - f = r$ servers; i.e., $ry = x$ fragments in total. In both cases, since the write has used x-of-(nz) encoding, the read can reconstruct the value. To see that the third condition is satisfied, notice that the total size of z stored fragments at a server after the second phase of the write is $z(\Delta/x) = \Delta/(n - 2f)$. After trimming, the size of the stored fragments become $y(\Delta/x) = \Delta/(r' - f)$.

Another significant difference between ORCAS-A and ORCAS-B is the condition for deleting an old value at a server. In ORCAS-A, whenever a server receives an unencoded value with a higher timestamp, the old fragment or the old unencoded value is overwritten. However in ORCAS-B, if the server receives fragments with a timestamp ts that is higher than its current timestamp, the server adds the fragments to a set L of received fragments. Subsequently, if it receives a trim message (i.e., a message from the third phase of a write) with timestamp ts, it deletes all fragments in L with a lower timestamp. Also, the server sends the whole set L in its r_ack reply messages to a read. (Thus the trim message also acts as a garbage collection message.) This modification is necessary in ORCAS-B because, until a sufficient number of encoded fragments are stored at the servers, the newly written value is not recoverable from the stored data obtained from any set of servers. The trim message acts as a confirmation that enough fragments of the new value have been stored. A similar garbage collection mechanism is also present in the implementation in [7]. On the other hand, since a server in ORCAS-A receives an unencoded value first, it can directly overwrite values with lower timestamps. An important consequence of this modification is that the worst-case storage size of ORCAS-B (and the implementation in [7]) is proportional to the number of concurrent writes, whereas, the storage requirement in ORCAS-A is never worse than that in replication (i.e., storing the unencoded value at all servers). We show the wait-free termination property of ORCAS-B in [6].

5 Discussion and Future Work

There are two related disadvantages of ORCAS-A when compared to most replication based implementations. The write needs three phases to complete as compared to two phases in the latter. Also, the write needs four stable storage accesses (in its critical path) as compared to two such accesses in replication based implementations. Both disadvantages primarily result from the last phase that is used for encoding the value at the servers, and which can be removed if we slightly relax the requirement on the storage space. ORCAS-A ensures that the stable storage is encoded whenever there is no write in progress. Instead, if we require that the stable storage is *eventually* encoded whenever there is no write in progress, then (with some minor modifications in ORCAS-A) the write operation can return without waiting for the last phase. The last phase can then be executed "lazily" by the client. (The two disadvantages and the above discussion hold for ORCAS-B as well.) On a similar note, in ORCAS-A, if a read selects a value in the first phase that is already encoded at some server, then it can return after the first phase, and lazily complete the second phase (which in this case is used only for deregistering at the servers, and not for writing back the value). It follows that, a read that has no concurrent write in ORCAS-A can return after the first phase.

An important open problem is to study storage lower bounds on register implementations in a crash-recovery model. In particular, it would be interesting to study if our lower bounds (that are presented in [6]) hold when some of the underlying assumptions are removed. Another interesting direction for investigation can be implementations that tolerate both process crash-recovery with fair-lossy channels and malicious processes.

References

1. Abraham, I., Chockler, G., Keidar, I., Malkhi, D.: Byzantine disk paxos: optimal resilience with byzantine shared memory. Distributed Computing 18(5), 387–408 (2006)
2. Aguilera, M.K., Janakiraman, R., Xu, L.: Using erasure codes efficiently for storage in a distributed system. In: Proceedings of the International Conference on Dependable Systems and Networks (DSN), pp. 336–345 (2005)
3. Attiya, H., Bar-Noy, A., Dolev, D.: Sharing memory robustly in a message passing system. Journal of the ACM 42(1), 124–142 (1995)
4. Boichat, R., Guerraoui, R.: Reliable and total order broadcast in the crash-recovery model. Journal of Parallel and Distributed Computing 65(4), 397–413 (2005)
5. Cachin, C., Tessaro, S.: Optimal resilience for erasure-coded byzantine distributed storage. In: Proceedings of the International Conference on Dependable Systems and Networks (DSN), pp. 115–124 (2006)
6. Dutta, P., Guerraoui, R., Levy, R.R.: Optimistic erasure-coded distributed storage. Technical report, EPFL-IC-LPD, Lausanne, Switzerland (2008)
7. Frolund, S., Merchant, A., Saito, Y., Spence, S., Veitch, A.: A decentralized algorithm for erasure-coded virtual disks. In: Proceedings of the International Conference on Dependable Systems and Networks (DSN), pp. 125–134 (2004)
8. Goodson, G.R., Wylie, J.J., Ganger, G.R., Reiter, M.K.: Efficient byzantine-tolerant erasure-coded storage. In: Proceedings of the International Conference on Dependable Systems and Networks (DSN) (2004)
9. Guerraoui, R., Levy, R.R., Pochon, B., Pugh, J.: The collective memory of amnesic processes. ACM Transactions on Algorithms 4(1) (2008)
10. Hendricks, J., Ganger, G.R., Reiter, M.K.: Low-overhead byzantine fault-tolerant storage. In: Proceedings of the 21st ACM Symposium on Operating Systems Principles (SOSP), pp. 73–86 (2007)
11. Hendricks, J., Ganger, G.R., Reiter, M.K.: Verifying distributed erasure-coded data. In: Proceedings of the Twenty-Sixth Annual ACM Symposium on Principles of Distributed Computing (PODC), pp. 139–146 (2007)
12. Herlihy, M.: Wait-free synchronization. ACM Transactions on Programming Languages and Systems 13(1), 124–149 (1991)
13. Lamport, L.: On interprocess communication - part i: Basic formalism, part ii: Algorithms. DEC SRC Report, 8 (1985); Also in Distributed Computing, 1, pp. 77-101 (1986)
14. Lynch, N.A.: Distributed Algorithms. Morgan Kaufmann Publishers, San Mateo (1996)
15. Lynch, N.A., Shvartsman, A.A.: Robust emulation of shared memory using dynamic quorum-acknowledged broadcasts. In: Proceedings of the International Symposium on Fault-Tolerant Computing Systems (FTCS) (1997)
16. Martin, J.-P., Alvisi, L., Dahlin, M.: Minimal byzantine storage. In: Proceedings of the International Symposium on Distributed Computing (DISC), pp. 311–325 (2002)
17. Reed, I.S., Solomon, G.: Polynomial codes over certain finite fields. SIAM Journal of Applied Mathematics 8, 300–304 (1960)