Erasure-Coded Byzantine Storage with Separate Metadata

Elli Androulaki[1], Christian Cachin[1], Dan Dobre[2], and Marko Vukolić[3]

[1] IBM Research - Zurich, Rüschlikon, Switzerland
{lli,cca}@zurich.ibm.com
[2] Work Done at NEC Labs Europe, Germany
dan@dobre.net
[3] Department of Computer Science, ETH Zurich, Switzerland and Eurécom, Sophia Antipolis, France
vukolic@eurecom.fr

Abstract. Although many distributed storage protocols have been introduced, a solution that combines the strongest properties in terms of availability, consistency, fault-tolerance, storage complexity, and concurrency has been elusive so far. Combining these properties is difficult, especially if the resulting solution is required to be efficient and incur low cost.

We present AWE, the first *erasure-coded* distributed implementation of a multi-writer multi-reader read/write register object that is, at the same time: (1) asynchronous, (2) wait-free, (3) atomic, (4) amnesic, (i.e., nodes store a bounded number of values), and (5) Byzantine fault-tolerant (BFT), using the optimal number of nodes. AWE maintains metadata separately from bulk data, which is encoded into fragments with a k-out-of-n erasure code and stored on dedicated *data nodes* that support only simple reads and writes. Furthermore, AWE is the first BFT storage protocol that uses only $n = 2t + k$ data nodes to tolerate t Byzantine faults, for any $k \geq 1$. Metadata, on the other hand, is stored using an atomic snapshot object, which may be realized from $3t + 1$ *metadata nodes* for tolerating t Byzantine faults.

AWE is efficient and uses only lightweight cryptographic hash functions. Moreover, we show that hash functions are needed by any BFT distributed storage protocol that stores the bulk data on $3t$ or fewer data nodes.

1 Introduction

Erasure coding is a key technique that saves space and retains robustness against faults in distributed storage systems. In short, an erasure code splits a large data value into n *fragments* such that from any k of them the input value can be reconstructed. Erasure coding is used by several large-scale storage systems [24, 28] that offer large capacity, high throughput, resilience to faults, and efficient use of storage space.

Whereas the storage systems in production use today only tolerate crashes or outages, storage systems in the *Byzantine failure model* survive also more severe faults, ranging from arbitrary state corruption to malicious attacks on processes. In this paper, we consider a model where multiple *clients* concurrently access a storage service provided by a distributed set of *nodes* , where t out of n nodes may be Byzantine. We model the storage service as an abstract read/write register object.

Although Byzantine-fault tolerant (BFT) erasure-coded distributed storage systems have received some attention in the literature [5, 9, 15, 18, 21], our understanding of their properties is not mature. The role of different quorums, the semantics of concurrent access, the latency of protocols, and the processing capabilities of the nodes have been investigated thoroughly for protocols based on *replication* [12,27]; in contrast, our knowledge about *erasure-coded* distributed storage is far more limited. In fact, the existing BFT erasure-coded storage protocols suffer from multiple drawbacks: some require nodes to store an unbounded number of values [18] or rely on node-to-node communication [9], others need computationally expensive public-key cryptography [9, 21] or may block clients due to concurrent operations of other clients [21].

Contribution. This paper introduces AWE, the first erasure-coded distributed implementation of a multi-reader multi-writer (MRMW) register that is, at the same time, (1) asynchronous, (2) wait-free, (3) atomic, (4) amnesic, (5) tolerates the optimal number of Byzantine nodes, and (6) does not use public-key cryptography.

These properties are desirable, as *wait-freedom* [22] and *atomicity* (or *linearizability*) [23] are not only the most fundamental but also the strongest liveness and consistency properties (respectively) of distributed storage. Roughly, wait-free liveness means that any correct client operation terminates irrespective of the behavior of the faulty nodes and clients, whereas atomicity means that all operations appear to take effect instantaneously. Therefore, guaranteeing wait-freedom and atomicity under the weakest possible assumptions (asynchrony, Byzantine faults) is highly desirable. Furthermore, *amnesic* storage [11] in combination with erasure-coding minimizes the storage overhead, another important measure for distributed storage. Roughly speaking, in amnesic storage nodes store a bounded number of values and erase obsolete data. Finally, the absence of public-key cryptography contributes to an efficient implementation of AWE. Although different subsets of these robustness properties have been demonstrated so far, they have never been achieved together for erasure-coded storage. Combining these desirable properties, has been a longstanding open problem [18].

AWE distinguishes between *metadata* (short control information) and *bulk data* (the erasure-coded stored values) and introduces two separate *classes* of nodes that store metadata and bulk data. With this approach, AWE beats the lower bound of $n > 3t$ nodes needed for distributed BFT storage [26], for the class of *data nodes* (that store bulk data). This makes AWE novel, as all known erasure-coded BFT storage solutions comply with this bound for their bulk data storage.

More specifically, with a k-out-of-n erasure code, protocol AWE needs only $2t + k$ data nodes, for any $k \geq 1$. This approach saves resources in practice, as storage costs for the bulk data often dominate. The data nodes may be passive objects that support read and write operations but cannot execute code, as in Disk Paxos [1]. In practice, such services may be provided by the key-value stores (KVS) popular in cloud storage.

We formulate AWE in a modular way using an abstract *metadata service* that stores control information with an *atomic snapshot* object. A snapshot object may be realized in a distributed asynchronous system from simple read/write registers [3]. For making this implementation fault-tolerant, these registers must still be emulated from $n > 3t$ different *metadata nodes* , in order to tolerate t Byzantine nodes.

Finally, AWE uses simple cryptographic hash functions but no expensive public-key operations. To explain the use of cryptography in AWE, we show that separating data from metadata and reducing the number of data nodes to $3t$ or less implies the use cryptographic techniques. This result is interesting in its own right, as it implies that *any* distributed BFT storage protocol that uses $3t$ or fewer nodes for storing bulk data must involve cryptographic hash functions and place a bound on the computational power of the Byzantine nodes. As all existing BFT erasure-coded storage protocols (including AWE) rely on cryptography, this result does not pose a restriction on practical systems. However, it illustrates a fundamental limitation that is particularly relevant for $k = 1$, i.e., for replication-based BFT storage protocols.

Structure. The paper continues with the overview of related work in Section 2. The model is given in Section 3 and Protocol AWE is presented in Section 4. The communication and storage complexities of AWE are compared to those of existing protocols in Section 5. Section 6 establishes the necessity of cryptographic assumptions for BFT storage with less than $3t$ data nodes. Finally, Section 7 concludes the paper. Detailed proofs appear in a technical report [4].

2 Related Work

Table 1 summarizes this section that gives a brief overview of the relevant related work.

Earlier designs for erasure-coded distributed storage have suffered from potential aborts due to contention [16] or from the need to maintain an unbounded number of fragments at data nodes [18]. In the crash-failure model, ORCAS [15] and CASGC [10] achieve optimal resilience $n > 2t$ and low communication overhead, combined with wait-free (ORCAS) and FW-termination (CASGC), respectively. FW-termination ensures that read operations always progress only in executions with a finite number of writes.

In the model with Byzantine nodes, Cachin and Tessaro (CT) [9] introduced the first wait-free protocol with atomic semantics and optimal resilience $n > 3t$. CT uses a verifiable information dispersal protocol but needs node-to-node communication, which lies outside our model. Hendricks et al. (HGR) [21] present an optimally resilient protocol that comes closest to our protocol among the existing solutions. It offers many desirable features, that is, it has as low communication cost, works asynchronously, achieves optimal resilience, atomicity, and is amnesic. Compared to our work, it (1) uses public-key cryptography, (2) achieves only FW-termination instead of wait-freedom, and (3) requires *processing* by the nodes, i.e., the ability to execute complex operations beyond simple reads and writes.

To be fair, much of the (cryptographic) overhead inherent in the CT and HGR protocols defends against poisonous writes from Byzantine clients, i.e., malicious client behavior that leaves the nodes in an inconsistent state. We do not consider Byzantine clients in this work, since permitting arbitrary client behavior is problematic [20]. Such a client might write garbage to the storage system and wipe out the stored value at any time. However, even without the steps that protect against poisonous writes, HGR still requires processing by the nodes and is not wait-free.

Table 1. Comparison of erasure-coded distributed storage solutions. An asterisk (*) denotes optimal properties. The column labeled *Type* states the computation requirements on nodes: *Proc.* denotes processing; *Msg.* means sending messages to other nodes, in addition to processing; *R/W* denotes a read/write register.

Protocol	BFT	Liveness	Data nodes	Type	Amnesic	Cryptogr.
ORCAS [15]	—	Wait-free	$2t+1$	Proc.	—	N/A
CASGC [10]	—	FW-term.	$2t+1$	Proc.	$\surd^*$	N/A
CT [9]	$\surd^*$	Wait-free *	$3t+1$	Msg.	—	Public-key
HGR [21]	$\surd^*$	FW-term.	$2t+k$, for $k>t$	Proc.	$\surd^*$	Public-key
M-PoWerStore [13]	$\surd^*$	Wait-free *	$3t+1$	Proc.	—	Hash func. *
DepSky [5]	$\surd^*$	Obstr.-free	$3t+1$	R/W *	—	Public-key
AWE (Sec. 4)	$\surd^*$	Wait-free *	$2t+k$, for $k \geq 1$ *	R/W *	$\surd^*$	Hash func. *

The M-PoWerStore protocol [13] employs a cryptographic "proof of writing" for wait-free atomic erasure-coded distributed storage without node-to-node communication. Similar to other protocols, M-PoWerStore uses $n > 3t$ nodes (with processing capabilities) and is not amnesic.

Several systems have recently addressed how to store erasure-coded data on multiple redundant cloud services but only few of them focus on wait-free concurrent access. HAIL [6], for instance, uses Byzantine-tolerant erasure coding and provides data integrity through proofs of retrievability; however, it does not address concurrent operations by different clients. DepSky [5] achieves regular semantics and uses lock-based concurrency control; therefore, one client may block operations of other clients.

A key aspect of AWE lies in the differentiation of (small) metadata from (large) bulk data: this enables a modular protocol design and an architectural separation for implementations. The concept also resembles the separation between agreement and execution used in the context of BFT replicated state machines in partially synchronous systems [29].

FARSITE [2] first introduced such a separation of metadata and data for replicated storage; their data nodes and their metadata abstractions require processing, however, in contrast to AWE. Non-explicit ways of separating metadata from data can already be found in several previous erasure coding-based protocols. For instance, the cross checksum, a vector with the hashes of all n fragments, has been replicated on the data nodes to ensure consistency [9, 18]. Separation of metadata has been also used in practical replicated crash-tolerant systems such as Hadoop Distributed File System.

Finally, Cachin et al. [7] have recently shown in a predecessor to this work that also with replication, separating metadata from bulk data has benefits. Their asynchronous wait-free BFT distributed storage protocol, called MDStore, reduces the number of data nodes to only $2t+1$. When protocol AWE is reduced to use replication with the trivial erasure code ($k = 1$), it uses as few nodes as MDStore to achieve the same wait-free atomic semantics; unlike AWE, however, MDStore is not amnesic and uses processing nodes.

The connection between separating data from metadata, reducing the number of data nodes, and the necessity of cryptographic techniques appears novel. In a sense, this paper shows a novel connection between the resilience of a distributed BFT protocol and the existence of a cryptographic primitive.

3 Definitions

We use a standard asynchronous deterministic distributed system of processes that communicate with each other. Processes comprise a set $\mathcal{C}$ of m *clients*, and a set $\mathcal{D}$ of n *data nodes* $d_1, \ldots, d_n$. Clients can only crash and up to t data nodes can be Byzantine and exhibit NR-arbitrary faults.

Protocols are presented in a modular event-based notation [8]. Processes interact through events that are qualified by the process identifier to which the event belongs. An event *Sample* of a process *m* with a parameter x is denoted by $\langle$ *m-Sample* $\mid x \rangle$. Processes execute *operations*, defined in terms of *invocation* and *response* events. We use the standard notions of operation precedence, histories, and linearizability [23].

A *read/write register r* is an object that stores a value from a domain $\mathcal{V}$ and supports exactly two operations: (1) a *Write* operation to *r* with invocation $\langle$ *r-Write* $\mid v \rangle$, taking a value $v \in \mathcal{V}$ that terminates with a response $\langle$ *r-WriteAck* $\rangle$; and (2) a *Read* operation from *r* with invocation $\langle$ *r-Read* $\rangle$ that terminates with a response $\langle$ *r-ReadResp* $\mid v \rangle$, containing a parameter $v \in \mathcal{V}$. The behavior of a register is given through its sequential specification, which requires that every *r-Read* operation returns the value written by the last preceding *r-Write* operation in the execution, or the special symbol $\perp \notin \mathcal{V}$ if no such operation exists.

The goal of this work is to describe a protocol that emulates a linearizable register abstraction among the clients; such a register is also called *atomic*. Some of the clients may crash and some nodes may be Byzantine. A protocol is called *wait-free* [22] if every operation invoked by a correct client eventually completes, irrespective of how other clients and nodes behave.

We make use of cryptographic hash functions modeled by a distributed oracle accessible to all processes [8]. A hash function H maps a bit string x of arbitrary length to a short, unique representation of fixed length. We use a *collision-free* hash function; this property means that no process, not even a Byzantine process, can find two distinct values x and x' such that $H(x) = H(x')$.

4 Protocol AWE

Erasure code. An (n, k)*-erasure code (EC)* with domain $\mathcal{V}$ is given by an encoding algorithm, denoted *Encode*, and a reconstruction algorithm, called *Reconstruct*. We consider only *maximum-distance separable codes*, which achieve the Singleton bound in the following sense. Given a (large) value $v \in \mathcal{V}$, algorithm $\mathit{Encode}_{k,n}(v)$ produces a vector $[f_1, \ldots, f_n]$ of n *fragments*, which are from a domain $\mathcal{F}$. A fragment is typically much smaller than the input, and any k fragments contain all information of v, that is, $|\mathcal{V}| \approx k|\mathcal{F}|$.

For an n-vector $F \in \left(\mathcal{F} \cup \{\perp\}\right)^n$, whose entries are either fragments or the symbol $\perp$, algorithm $\mathit{Reconstruct}_{k,n}(F)$ outputs a value $v \in \mathcal{V}$ or $\perp$. An output value of $\perp$ means that the reconstruction failed. The *completeness* property of an erasure code requires that an encoded value can be reconstructed from any k fragments. In other words, for every $v \in \mathcal{V}$, when one computes $F \leftarrow \mathit{Encode}_{k,n}(v)$ and then erases up to $n - k$ entries in F by setting them to $\perp$, algorithm $\mathit{Reconstruct}_{k,n}(F)$ outputs v.

Metadata service. The metadata service is implemented by a standard *atomic snapshot object* [3], called *dir*, that serves as a *directory*. A snapshot object extends the simple storage function of a register to a service that maintains one value for each client and allows for better coordination. Like an array of multi-reader single-writer (MRSW) registers, it allows every client to *update* its value individually; for reading it supports a *scan* operation that returns the vector of the stored values, one for every client. More precisely, the operations of *dir* are:

– An *Update* operation to *dir* is triggered by an invocation $\langle$ *dir-Update* $\mid c, v \rangle$ by client c that takes a value $v \in \mathcal{V}$ as parameter and terminates by generating a response $\langle$ *r-UpdateAck* $\rangle$ with no parameter.
– A *Scan* operation on *dir* is triggered by an invocation $\langle$ *dir-Scan* $\rangle$ with no parameter; the snapshot object returns a vector V of $m = |\mathcal{C}|$ values to c as the parameter in the response $\langle$ *r-ScanResp* $\mid V \rangle$, with $V[c] \in \mathcal{V}$ for $c \in \mathcal{C}$.

The sequential specification of the snapshot object follows directly from the specification of an array of m MRSW registers (hence, the snapshot initially stores the special symbol $\perp \notin \mathcal{V}$ in every entry). When accessed concurrently from multiple clients, its operations appear to take place atomically, i.e., they are linearizable. Snapshot objects are weak — they can be implemented from read/write registers [3], which, in turn, can be implemented from a set of a distributed processes subject to Byzantine faults. Wait-free amnesic implementations of registers with the optimal number of $n > 3t$ processes are possible using existing constructions [14, 19].

Data nodes. Data nodes provide a simple key-value store interface. We model the state of data nodes as an array $\mathit{data}[\mathit{ts}] \in \Sigma^*$, initially $\perp$, for $\mathit{ts} \in \mathit{Timestamps}$. Every value is associated to a timestamp, which consists of a sequence number *sn* and the identifier c of the writing client, i.e., $\mathit{ts} = (\mathit{sn}, c) \in \mathit{Timestamps} = N_0 \times (\mathcal{C} \cup \{\perp\})$; timestamps are initialized to $T_0 = (0, \perp)$. Data node d_i exports three operations:

– $\langle d_i$*-Write* $\mid \mathit{ts}, v \rangle$, which assigns $\mathit{data}[\mathit{ts}] \leftarrow v$ and returns $\langle d_i$*-WriteAck* $\mid \mathit{ts} \rangle$;
– $\langle d_i$*-Read* $\mid \mathit{ts} \rangle$, which returns $\langle d_i$*-ReadResp* $\mid \mathit{ts}, \mathit{data}[\mathit{ts}] \rangle$; and
– $\langle d_i$*-Free* $\mid \mathit{TS} \rangle$, which assigns $\mathit{data}[\mathit{ts}] \leftarrow \perp$ for all $\mathit{ts} \in \mathit{TS}$, and returns $\langle d_i$*-FreeAck* $\mid \mathit{TS} \rangle$.

4.1 Protocol Overview

AWE uses the metadata directory *dir* to maintain pointers to the fragments stored at the data nodes. The directory stores an entry for every writer; it contains the timestamp of its most recently written value, the identities of those nodes that have acknowledged to store a fragment of it, a vector with the hashes of the fragments for ensuring data integrity, and additional metadata to support concurrent reads and writes. The linearizable semantics of protocol AWE are obtained from the atomicity of the metadata directory.

At a high level, the writer first invokes *dir-Scan* on the metadata to read the highest stored timestamp, increments it, and uses this as the timestamp of the value to be written. Then it encodes the value to n fragments and sends one fragment to each data node. The data nodes store it and acknowledge the write. After the writer has received acknowledgments from $t + k$ data nodes, it writes their identities (together with the timestamp and the hashes of the fragments) to the metadata through *dir-Update*. The reader proceeds accordingly: it first invokes *dir-Scan* to obtain the entries of all writers; it determines the highest timestamp among them and extracts the fragment hashes and the identities of the data nodes; finally, it contacts the data nodes and reconstructs the value after obtaining k fragments that match the hashes in the metadata.

Although this simplified algorithm achieves atomic semantics, it does not address timely garbage-collection of obsolete fragments, the main problem to be solved for amnesic erasure-code distributed storage. If a writer would simply replace the fragments with those of the value written next, it is easy to see a concurrent reader may stall.

Protocol AWE uses two mechanisms to address this: first, the writer *retains* those values that may be accessed concurrently and exempts them from garbage collection so that their fragments remain intact for concurrent readers, which gives the reader enough time to retrieve its fragments. Secondly, some of the retained values may also be *frozen* in response to concurrent reads; this forces a concurrent read to retrieve a value that is guaranteed to exist at the data nodes rather than simply the newest value, thereby effectively limiting the amount of stored values. A similar freezing method has been used for wait-free atomic storage with replicated data [14, 19], but it must be changed for erasure-coded storage with separated metadata. The retention technique together with the separation of metadata appears novel. More specifically, metadata separation prevents straightforward applications of existing "freezing" techniques, whereas storage that is simultaneously wait-free and amnesic requires garbage collection method that we show here for the first time.

For the two mechanisms, i.e., retention and freezing, every reader maintains a *reader index*, both in its local variable *readindex* and in its metadata. The reader index serves for coordination between the reader and the writers. The reader increments its index whenever it starts a new *r-Read* and immediately writes it to *dir*, thereby announcing its intent to read. Writers access the reader indices after updating the metadata for a write and before (potentially) erasing obsolete fragments. Every writer w maintains a table *frozenindex* with its most recent recollection of all reader indices. When the newly obtained index of a reader c has changed, then w detects that c has started a new operation at some time after the last write of w.

When w detects a new operation of c, it does not know whether c has retrieved the timestamp from *dir* before or after the *dir-Update* of the current write. The reader may access either value; the writer therefore *retains* both the current and the preceding value for c by storing a pointer to them in *frozenptrlist* and in *reservedptrlist*. Clearly, both values have to be excluded from garbage collection by w in order to guarantee that the reader completes.

However, the operation of the reader c may access *dir* after the *dir-Update* of one or more subsequent write operation by w, which means that the nodes would have to retain every value subsequently written by w as well. To prevent this from happening and to limit the number of stored values, w *freezes* the currently written timestamp (as well as the value) and forces c to read this timestamp when it accesses *dir* within the same operation. In particular, the writer stores the current timestamp in *frozenptrlist* at index c and updates the reader index of c in *frozenindex*; then, the writer pushes both tables, *frozenindex* and *frozenptrlist*, to the metadata service during its next *r-Write*. The values designated by *frozenptrlist* (they are called *frozen*) and *reservedptrlist* (they are called *reserved*) are retained and excluded from garbage collection until w detects the next read of c, i.e., the reader index of c increases. Thus, the current read may span many concurrent writes of w and the fragments remain available until c finishes reading.

On the other hand, a reader must consider frozen values. When a slow read operation spans multiple concurrent writes, the reader c learns that it should retrieve the frozen value through its entry in the *frozenindex* table of the writer.

The protocol is amnesic because each writer retains at most two values per reader, a frozen value and a reserved value. Every data node therefore stores at most two fragments for every reader-writer pair plus the fragment from the currently written value. The combination of freezing and retentions ensures wait-freedom.

4.2 Details

Data structures. We use abstract data structures for compactness. In particular, given a timestamp $ts = (sn, c)$, its two fields can be accessed as $ts.sn$ and $ts.c$. A data type *Pointers* denotes a set of tuples of the form $(ts, set, hash)$ with $ts \in Timestamps$, $set \subseteq [1, n]$, and $hash[i] \in \Sigma^*$ for $i \in [1, n]$. Their initialization value is $Nullptr = ((0, \perp), \emptyset, [\perp, \ldots, \perp])$.

A *Pointers* structure contains the relevant information about one stored value. For example, the writer locally maintains $writeptr \in Pointers$ designating to the most recently written value. More specifically, *writeptr.ts* contains the timestamp of the written value, *writeptr.set* contains the identities of the nodes that have confirmed to have stored the written value, and *writeptr.hash* contains the cross checksum, the list of hash values of the data fragments, of the written value.

The metadata directory *dir* contains a vector M with a tuple for every client $p \in \mathcal{C}$ of the form

$$M[p] = \big(writeptr, frozenptrlist, frozenindex, readindex\big),$$

where the field $writeptr \in Pointers$ represents the *written value*, the field *frozenptrlist* is an array indexed by $c \in \mathcal{C}$ such that $frozenptrlist[c] \in Pointers$ denotes a value *frozen by p for reader c*, and the integer *readindex* denotes the reader-index of p.

For preventing that concurrently accessed fragments are cleaned up too early, the writer maintains two tables, *frozenptrlist*, and *reservedptrlist*, each containing one *Pointers* entry for every reader in $\mathcal{C}$. The second one, *reservedptrlist*, is stored only locally, together with the *frozenindex* table, which denotes the writer's most recently obtained copy of the reader indices. For the operations of the reader, only the local *readindex* counter is needed.

Every client maintains the following variables between operations: *writeptr*, *frozenptrlist*, *frozenindex*, and *reservedptrlist* implement freezing, reservations, and retentions for writers as mentioned, and *readindex* counts the reader operations. When clients access *dir*, they may not be interested to retrieve all fields or to update all fields; for clarity, we replace the fields to be ignored by $*$ in *dir-Scan* and *dir-Update* operations.

Algorithm 1. Protocol AWE, atomic register instance r for client c (part 1).

```
State
    // State maintained across write and read operations
    $writeptr \in Pointers$, initially $Nullptr$                // Metadata of the currently written value
    $frozenptrlist[p] \in Pointers$, initially $Nullptr$, for $p \in \mathcal{C}$        // Frozen and retained for $p$
    $reservedptrlist[p] \in Pointers$, initially $Nullptr$, for $p \in \mathcal{C}$      // Reserved and retained for $p$
    $frozenindex[p] \in N_0$, initially 0, for $p \in \mathcal{C}$          // Last known reader index of $p$
    $readindex \in N_0$, initially 0                             // Reader index of $c$
    // Temporary state during operations
    $prevptr \in Pointers$, initially $Nullptr$           // Metadata of the value written by $c$ before
    $readptr \in Pointers$, initially $Nullptr$              // Metadata of the value to be read by $c$
    $readlist[i] \in \Sigma^*$, initially $\perp$, for $i \in [1, n]$  // List of nodes that have responded during read

upon $\langle$ r-Write $| v \rangle$ do
    $prevptr \leftarrow writeptr$
    invoke $\langle$ dir-Scan $\rangle$; wait for $\langle$ dir-ScanResp $| M \rangle$
    $(wsn, *) \leftarrow \max\{M[p].writeptr.ts \mid p \in \mathcal{C}\}$          // Highest ts field in a writeptr in $M$
    $writeptr.ts \leftarrow (wsn + 1, c)$          // Construct metadata of the currently written value
    $writeptr.set \leftarrow \emptyset$
    $[v_1, \ldots, v_n] \leftarrow Encode_{k,n}(v)$
    forall $i \in [1, n]$ do
        $writeptr.hash[i] \leftarrow H(v_i)$
        invoke $\langle d_i$-Write $| writeptr.ts, v_i \rangle$

upon $\langle d_i$-WriteAck $| ats \rangle$ such that $ats = writeptr.ts \wedge |writeptr.set| < t + k$ do
    $writeptr.set \leftarrow writeptr.set \cup \{i\}$
    if $|writeptr.set| = t + k$ then
        // Update metadata at dir with currently written value and with frozen values
        invoke $\langle$ dir-Update $| c, (writeptr, frozenptrlist, frozenindex, *) \rangle$
        wait for $\langle$ dir-UpdateAck $\rangle$
        // Obtain current reader indices
        invoke $\langle$ dir-Scan $\rangle$; wait for $\langle$ dir-ScanResp $| M \rangle$
        $freets \leftarrow \{prevptr.ts\}$
        forall $p \in \mathcal{C} \setminus \{c\}$ do
            $(*, *, *, index) \leftarrow M[p]$
            if $index > frozenindex[p]$ then
                // Client $p$ may be concurrently reading prevptr or writeptr
                $freets \leftarrow freets \cup \{frozenptrlist[p].ts, reservedptrlist[p].ts\}$
                $frozenptrlist[p] \leftarrow writeptr$; $frozenindex[p] \leftarrow index$
                $reservedptrlist[p] \leftarrow prevptr$
        $freets \leftarrow freets \setminus \bigcup_{p \in C}\{frozenptrlist[p].ts, reservedptrlist[p].ts\}$
        forall $j \in [1, n]$ do       // Clean up fragments except for current, frozen, and reserved
            invoke $\langle d_j$-Free $| freets \rangle$
        invoke $\langle$ r-WriteAck $\rangle$
```

Operations. At the start of a write operation, the writer w saves the current value of *writeptr* in *prevptr*, to be used later during its operation, if w should reserve and retain

Algorithm 2. Protocol AWE, atomic register instance r for client c (part 2).

```
upon ⟨ r-Read ⟩ do
    forall $i \in [1, n]$ do $readlist[i] \leftarrow \perp$
    $readindex \leftarrow readindex + 1$
    invoke ⟨ dir-Update | $c, (*, *, *, readindex)$ ⟩; wait for ⟨ dir-UpdateAck ⟩
    // Parse the content of dir and extract the highest timestamp, potentially frozen for $c$
    invoke ⟨ dir-Scan ⟩; wait for ⟨ dir-ScanResp | $M$ ⟩
    $readptr \leftarrow highestread(M, c, readindex)$
    if $readptr.ts = (0, \perp)$ then
        invoke ⟨ r-ReadResp | $\perp$ ⟩
    else // Contact the data nodes to obtain the data fragments
        forall $i \in readptr.set$ do
            invoke ⟨ $d_i$-Read | $readptr.ts$ ⟩

upon ⟨ $d_i$-ReadResp | $vts, v$ ⟩ such that $vts = readptr.ts \wedge readlist[i] = \perp$ do
    if $v \neq \perp \wedge H(v) = readptr.hash[i]$ then
        $readlist[i] \leftarrow v$
        if $\left|\{j | readlist[j] \neq \perp\}\right| = k$ then
            $readptr \leftarrow Nullptr$
            $retval \leftarrow Reconstruct_{k,n}(readlist)$
            invoke ⟨ r-ReadResp | $retval$ ⟩
```

that value. Then w determines the timestamp of the current operation, which is stored in *writeptr.ts*. After computing the fragments of v, sending them to the data nodes, and obtaining $t+k$ acknowledgements, the writer updates its metadata entry. It writes *writeptr*, pointing to v, together with *frozenptrlist* and *frozenindex*, as they resulted after the previous write to *dir*. Then w invokes *dir-Scan* and acquires the current metadata M, which it uses to determine values to freeze and to retain. It compares the acquired reader indices with the ones obtained during its last write (as stored in *frozenindex*). When w detects a read operation by c because $M[c].readindex > frozenindex[c]$, it freezes the current value (by setting $frozenptrlist[p]$ to *writeptr*) and reserves the previously written value (by setting $reservedptrlist[p]$ to *prevptr*). Finally, the writer deletes all fragments at the data nodes except for those of the currently written and the retained values.

To determine the timestamps for retrieving fragments, the reader uses the following two functions:

```
function readfrom($M, c, p, index$) is
    if $index > M[p].frozenindex[c]$ then
        return $M[p].writeptr$
    else    // $index = M[p].frozenindex[c]$
        return $M[p].frozenptrlist[c]$
```

```
function highestread($M, c, index$) is
    $max \leftarrow Nullptr$
    forall $p \in \mathcal{C}$ do
        $ptr \leftarrow readfrom(M, c, p, index)$
        if $ptr.ts > max.ts$ then
            $max \leftarrow ptr$
    return $max$
```

Upon retrieving the array M from *dir*, the reader sets

$$\textit{readptr} \leftarrow \textit{highestread}(M, c, \textit{readindex}),$$

which implements the logic of accessing frozen timestamps. The details of AWE appear in Algorithms 1–2.

Remarks. AWE does not rely on a majority of correct data nodes for correctness, as this is encapsulated in the directory service. For liveness, though, the protocol needs responses from $t + k$ data nodes during write operations, which is only possible if $n \geq 2t + k$. Furthermore, several optimizations may reduce the storage overhead in practice, e.g., readers can clean up values that are no longer needed by anyone.

5 Complexity Comparison

This section compares the communication and storage complexities of AWE to existing erasure-coded distributed storage solutions, in a setting with n data nodes and m clients. We denote the size of each stored value $v \in \mathcal{V}$ by $\ell = \lceil \log_2 |\mathcal{V}| \rceil$. In line with the intended deployment scenarios, we assume that ℓ is much larger (by several orders of magnitude) than n^2 and m^2 , i.e., $\ell \gg n^2$ and $\ell \gg m^2$.

We examine the worst-case communication and storage costs incurred by a client in protocol AWE and distinguish metadata operations (on *dir*) from operations on the data nodes. The metadata of one value written to *dir* consists of a pointer, containing the cross checksum with n hash values, the $t+k$ identities of the data nodes that store a data fragment, and a timestamp. Moreover, the metadata entry of one writer contains also the list of m pointers to frozen values, the m indices relating to the frozen values, and the writer's reader index. Assuming a collision-resistant hash function with output size λ bits and timestamps no larger than λ bits, the total size of the metadata is $O(m^2 n\lambda)$. In the remainder of this section, the size of the metadata is considered to be negligible and is ignored, though it would incur in practice.

According to the above assumption, the complexity of AWE is dominated by the data itself. When writing a value $v \in \mathcal{V}$, the writer sends a fragment of size ℓ/k and a timestamp of size λ to each of the n data nodes. Assuming further that $\ell \gg \lambda$, the total storage space occupied by v at the data nodes amounts to $n\ell/k$ bits. Similarly, a read operation incurs a communication cost of $(t + k)k/\ell$ bits.

With respect to storage complexity, protocol AWE freezes and reserves two timestamps and their fragments for each writer-reader pair, and additionally stores the fragments of the last written value for each writer. This means that the storage cost is at most $2m^2 n\ell/k$ bits in total. The improvement described in a remark of Section 4.2 reduces this to $2mn\ell/k$ in the best case.

Table 2 shows the communication and storage costs of protocol AWE and the related protocols. Observe that in CASGC [10] and HGR [21], a read operation concurrent with an unbounded number of writes may not terminate, hence we state their cost as ∞. Moreover, in contrast to AWE, DepSky [5] is neither wait-free nor amnesic and M-PoWerStore [13] is not amnesic. It is easy to see that the communication complexity of AWE is lower than that of most storage solutions.

Table 2. Comparison of the communication and space complexities of erasure-coded distributed storage solutions. There are m clients, n data nodes, the erasure code parameter is $k = n - 2t$, and the data values are of size ℓ bits. An asterisk (*) denotes optimal properties.

Protocol	**Communication cost**		**Storage cost**
	Write	*Read*	
ORCAS-A [15]	$(1+m)n\ell$	$2n\ell$	$n\ell$
ORCAS-B [15]	$(1+m)n\ell/k$	$2n\ell/k$	$mn\ell/k$
CASGC [10]	$n\ell/k\ ^*$	∞	$mn\ell/k$
CT [9]	$(n+m)n\ell/(k+t)$	$\ell\ ^*$	$n\ell/(k+t)\ ^*$
HGR [21]	$n\ell/k\ ^*$	∞	$mn\ell/k$
M-PoWerStore [13]	$n\ell/k\ ^*$	$n\ell/k$	∞
DepSky [5]	$n\ell/k\ ^*$	$n\ell/k$	∞
AWE (Sec. 4)	$n\ell/k\ ^*$	$(t+k)\ell/k$	$2m^2n\ell/k$

6 Necessity of Cryptography

In this section, we show that every BFT storage protocol that maintains bulk data (as opposed to short metadata) on $3t$ or fewer nodes while tolerating t Byzantine faults implies the existence of cryptographic hash functions. We strengthen this result by considering single-writer single-reader implementations of a register object with value domain $\mathcal{V}$ where n data nodes are aided by one *metadata service (MDS)* process; intuitively, the role of the MDS in an implementation is to store coordination data, but not values. Up to t data nodes may exhibit Byzantine faults, yet the MDS is a correct process. We do not rely on self-verifying data [25] — the processes have no way to check to tell apart "valid" from "invalid" values.

We consider a *computational* model and adopt a cryptographic security notion [17]. Let κ be a security parameter. Suppose every process is implemented by an *efficient* algorithm, that is, an algorithm whose running time is bounded by some polynomial in κ; the length of the input values and the internal state of every process are also bounded by this polynomial. We assume the storage emulation takes inputs of length $\ell(\kappa)$, a polynomial in κ, i.e., $|\mathcal{V}| \leq 2^{\ell(\kappa)}$. Suppose that any MDS implementation has *small state* in the sense that its internal memory is restricted to $\phi(\kappa)$ bits such that there exists a constant $c > 1$ such that for all $\kappa > 0$, $\ell(\kappa) > \phi(\kappa)^c$. This ensures that a register emulation cannot simply store the written at the MDS.

We abstract the hash function as follows.

Definition 1 (Digest oracle). *A* digest oracle *D is a distributed atomic object accessible to all processes. It supports only one operation that takes a bit string x of arbitrary length as input and outputs a bit string d (denoted $D(x)$) of fixed length $\lambda(\kappa)$, where λ is a polynomial in κ.*

The operation of D may be probabilistic but it implements a mathematical function in the sense that when queried with an input that has already been queried before, it returns again the same output. Furthermore, D satisfies the following collision-resistance *property. Consider any efficient adversarial process $\mathcal{A}$ with access to D that attempts to find a collision in D. The probability that $\mathcal{A}$ outputs two values x and x' such that*

$D(x) = D(x')$ is negligible in κ. (A function μ is called negligible *when for every integer $c > 0$ there exists an integer κ_c such that for all $\kappa > \kappa_c$, it holds $|\mu(\kappa)| < \kappa^{-c}$.)*

The principal result of this section, stated next in Theorem 1, combines a standard indistinguishability argument about a concurrent system with a cryptographic reduction.

Theorem 1. *Consider a deterministic emulation Π of a safe register, which uses a meta-data service MDS and $n \leq 3t$ data nodes such that up to t of the data nodes may be Byzantine and controlled by an adversary. If MDS has small state, then a collision-free digest oracle D can be implemented.*

Proof. We first define D, which is implemented from a simulation of the storage protocol Π that uses MDS. More precisely, to compute the digest of a value x, a simulator executes Π by simulating one writer process w that executes *write*(x), the n data nodes, and MDS. Then the simulator outputs the internal state md of process MDS as the return value of D. Whenever D is invoked, the simulator starts from the initial state and uses the same schedule; this ensures that two invocations of D with the same input give the same output.

We now show that D constructed from Π is collision-free. Towards a *contradiction*, assume there exist two distinct values a and b in $\mathcal{V}$ such that $D(a) = D(b)$. We now argue that Π is not a safe register emulation by describing multiple executions of Π. For simplicity, assume that $n = 3t$ and divide the n data nodes into three groups of t each, called A, B, and F.

Consider first an execution α of Π where initially w writes a using the schedule of the emulation of D. Suppose the nodes in A and F participate in this emulation and let t_α denote the time when the simulation of D returns md_a, the state of M. No messages from the writer are delivered to nodes in B.

Second, in execution β of Π, the value b is written. The execution is the same as α, except that the nodes in B participate instead of those in A and no messages from the writer are delivered to nodes in A. Note that $md_b = D(b) = D(a) = md_a$ by the assumption on a and b — the state of MDS is the same after *write*(a) in α as after *write*(b) in β.

Consider now an execution $\bar{\alpha}$ that extends α beyond t_α. At time t_α, the processes in A are being delayed indefinitely and do not take any further steps; as in α, no messages from w to nodes in B are ever delivered before the execution ends and the nodes in B continue operating from their initial state. Next, a reader r invokes *read*, interacts with the nodes in $B \cup F$ and with MDS, and returns a according to the safety property of the storage emulation.

Finally, consider an execution $\bar{\beta}$ that extends β beyond t_α. Here, the processes in B are delayed indefinitely from time t_α onward. Again, the nodes in A have still their initial state and continue now to participate in the execution. Furthermore, all nodes in F exhibit a Byzantine fault and *replace their state* with their state at time t_α in α; after that they again follow Π. Next, a reader r invokes *read* and only interacts with the nodes in $A \cup F$ and with MDS. Recall the state of MDS in β is the same as in α at time t_α. Since the nodes in A have the initial state and those in F and process MDS have the same state as in α at time t_α, execution $\bar{\beta}$ resumes from the same state as in $\bar{\alpha}$ except that the roles of the nodes in A and B are exchanged. However, as the emulation is deterministic, the reader cannot distinguish $\bar{\beta}$ from $\bar{\alpha}$ and returns a. This violates the safety of the storage emulation as *write*(b) precedes *read* in $\bar{\beta}$ but *read* returns a. A contradiction.

7 Conclusion

This paper has presented AWE, the first *erasure-coded* distributed implementation of a multi-writer multi-reader read/write register object that is, at the same time, (1) asynchronous, (2) wait-free, (3) atomic, (4) amnesic, (i.e., with data nodes storing a bounded number of values) and (5) Byzantine fault-tolerant (BFT) using the optimal number of nodes. AWE is efficient since it does not use public-key cryptography and requires data nodes that support only reads and writes, further reducing the cost of deployment and ownership of a distributed storage solution. Notably, AWE stores metadata separately from k-out-of-n erasure-coded fragments. This enables AWE to be the first BFT protocol that uses as few as $2t + k$ data nodes to tolerate t Byzantine nodes, for any $k \geq 1$.

Future work should address how to optimize protocol AWE and to reduce the storage consumption for practical systems; this could be done at the cost of increasing its conceptual complexity and losing some of its ideal properties. For instance, when the metadata service is moved from a storage abstraction to a service with processing, it is conceivable that fewer values have to be retained at the nodes.

Acknowledgment. We thank Radu Banabic, Nikola Knežević, and Alessandro Sorniotti for inspiring discussions during the early stages of this work. This work is supported in part by the EU CLOUDSPACES (FP7-317555) and SECCRIT (FP7-312758) projects.

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] Adya, A., Bolosky, W.J., Castro, M., Cermak, G., Chaiken, R., Douceur, J.R., Howell, J., Lorch, J.R., Theimer, M., Wattenhofer, R.P.: FARSITE: Federated, available, and reliable storage for an incompletely trusted environment. In: Proc. Symp. Operating Systems Design and Implementation (2002)
[3] Afek, Y., Attiya, H., Dolev, D., Gafni, E., Merritt, M., Shavit, N.: Atomic snapshots of shared memory. Journal of the ACM 40(4), 873–890 (1993)
[4] Androulaki, E., Cachin, C., Dobre, D., Vukolić, M.: Erasure-coded Byzantine storage with separate metadata. Report arXiv:1402.4958, CoRR (2014)
[5] Bessani, A., Correia, M., Quaresma, B., André, F., Sousa, P.: DepSky: Dependable and secure storage in a cloud-of-clouds. In: Proc. European Conference on Computer Systems, pp. 31–46 (2011)
[6] Bowers, K.D., Juels, A., Oprea, A.: HAIL: A high-availability and integrity layer for cloud storage. In: Proc. ACM Conference on Computer and Communications Security, pp. 187–198 (2009)
[7] Cachin, C., Dobre, D., Vukolić, M.: Separating data and control: Asynchronous BFT storage with $2t + 1$ data replicas. In: Felber, P., Garg, V. (eds.) SSS 2014. LNCS, vol. 8756, pp. 1–17. Springer, Heidelberg (2014)

[8] Cachin, C., Guerraoui, R., Rodrigues, L.: Introduction to Reliable and Secure Distributed Programming, 2nd edn. Springer (2011)

[9] Cachin, C., Tessaro, S.: Optimal resilience for erasure-coded Byzantine distributed storage. In: Proc. Dependable Systems and Networks, pp. 115–124 (2006)

[10] Cadambe, V.R., Lynch, N., Medard, M., Musial, P.: Coded atomic shared memory emulation for message passing architectures. CSAIL Technical Report MIT-CSAIL-TR-2013-016. MIT (2013)

[11] Chockler, G., Guerraoui, R., Keidar, I.: Amnesic distributed storage. In: Pelc, A. (ed.) DISC 2007. LNCS, vol. 4731, pp. 139–151. Springer, Heidelberg (2007)

[12] Chockler, G., Guerraoui, R., Keidar, I., Vukolić, M.: Reliable distributed storage. IEEE Computer 42(4), 60–67 (2009)

[13] Dobre, D., Karame, G., Li, W., Majuntke, M., Suri, N., Vukolić, M.: PoWerStore: Proofs of writing for efficient and robust storage. In: Proc. ACM Conference on Computer and Communications Security (2013)

[14] Dobre, D., Majuntke, M., Suri, N.: On the time-complexity of robust and amnesic storage. In: Baker, T.P., Bui, A., Tixeuil, S. (eds.) OPODIS 2008. LNCS, vol. 5401, pp. 197–216. Springer, Heidelberg (2008)

[15] Dutta, P.S., Guerraoui, R., Levy, R.R.: Optimistic erasure-coded distributed storage. In: Taubenfeld, G. (ed.) DISC 2008. LNCS, vol. 5218, pp. 182–196. Springer, Heidelberg (2008)

[16] Frølund, S., Merchant, A., Saito, Y., Spence, S., Veitch, A.: A decentralized algorithm for erasure-coded virtual disks. In: Proc. Dependable Systems and Networks, pp. 125–134 (2004)

[17] Goldreich, O.: Foundations of Cryptography, vol. I & II. Cambridge University Press (2001–2004)

[18] Goodson, G.R., Wylie, J.J., Ganger, G.R., Reiter, M.K.: Efficient Byzantine-tolerant erasure-coded storage. In: Proc. Dependable Systems and Networks, pp. 135–144 (2004)

[19] Guerraoui, R., Levy, R.R., Vukolić, M.: Lucky read/write access to robust atomic storage. In: Proc. Dependable Systems and Networks, pp. 125–136 (2006)

[20] Hendricks, J.: Efficient Byzantine Fault Tolerance for Scalable Storage and Services. Ph.D. thesis, School of Computer Science, Carnegie Mellon University (2009)

[21] Hendricks, J., Ganger, G.R., Reiter, M.K.: Low-overhead Byzantine fault-tolerant storage. In: Proc. ACM Symposium on Operating Systems Principles (2007)

[22] Herlihy, M.: Wait-free synchronization. ACM Transactions on Programming Languages and Systems 11(1), 124–149 (1991)

[23] Herlihy, M.P., Wing, J.M.: Linearizability: A correctness condition for concurrent objects. ACM Transactions on Programming Languages and Systems 12(3), 463–492 (1990)

[24] Huang, C., Simitci, H., Xu, Y., Ogus, A., Calder, B., Gopalan, P., et al.: Erasure coding in Windows Azure Storage. In: Proc. USENIX Annual Technical Conference (2012)

[25] Malkhi, D., Reiter, M.K.: Byzantine quorum systems. Distributed Computing 11(4), 203–213 (1998)

[26] Martin, J.P., Alvisi, L., Dahlin, M.: Minimal Byzantine storage. In: Malkhi, D. (ed.) DISC 2002. LNCS, vol. 2508, pp. 311–325. Springer, Heidelberg (2002)

[27] Vukolić, M.: Quorum Systems: With Applications to Storage and Consensus. Synthesis Lectures on Distributed Computing Theory. Morgan & Claypool (2012)

[28] Wong, W.: Cleversafe grows along with customers' data storage needs. Chicago Tribune (2013)

[29] Yin, J., Martin, J.P., Alvisi, A.V.L., Dahlin, M.: Separating agreement from execution in Byzantine fault-tolerant services. In: Proc. ACM Symposium on Operating Systems Principles, pp. 253–268 (2003)