Storage-Optimized Data-Atomic Algorithms for Handling Erasures and Errors in Distributed Storage Systems

Kishori M. Konwar1, N. Prakash1, Erez Kantor2, Nancy Lynch1, Muriel Médard1, and Alexander A. Schwarzmann3

1Department of EECS, MIT, MA, USA, 2Department of Computer Science, NEU, MA, USA, 3Department of CSE, UConn., Storrs, CT, USA

Abstract—Erasure codes are increasingly being studied in the context of implementing atomic memory objects in large scale asynchronous distributed storage systems. When compared with the traditional replication based schemes, erasure codes have the potential of significantly lowering storage and communication costs while simultaneously guaranteeing the desired resiliency levels. In this work, we propose the Storage-Optimized Data-Atomic (SODA) algorithm for implementing atomic memory objects in the multi-writer multi-reader setting. SODA uses Maximum Distance Separable (MDS) codes, and is specifically designed to optimize the total storage cost for a given fault-tolerance requirement. For tolerating \( f \) server crashes in an \( n \)-server system, SODA uses an \([n,k]\) MDS code with \( k = n - f \), and incurs a total storage cost of \( \frac{n}{n-k} \). SODA is designed under the assumption of reliable point-to-point communication channels. The communication cost of a write and a read operation are respectively given by \( O(f^2) \) and \( \frac{n}{n-k}(\delta_w + 1) \), where \( \delta_w \) denotes the number of writes that are concurrent with the particular read. In comparison with the recent CASGC algorithm [1], which also uses MDS codes, SODA offers lower storage cost while pays more on the communication cost.

We also present a modification of SODA, called SODAerr, to handle the case where some of the servers can return erroneous coded elements during a read operation. Specifically, in order to tolerate \( f \) server failures and \( e \) error-prone coded elements, the SODAerr algorithm uses an \([n,k]\) MDS code such that \( k = n - 2e - f \). SODAerr also guarantees liveness and atomicity, while maintaining an optimized total storage cost of \( \frac{n}{n-f-2e} \).

Keywords—atomicity, multi-writer multi-reader, codes for storage, storage cost, communication cost

I. INTRODUCTION

The demand for efficient and reliable large-scale distributed storage systems (DSSs) has grown at an unprecedented scale in the recent years. DSSs that store massive data sets across several hundreds of servers are increasingly being used for both industrial and scientific applications, ranging from sequencing genomic data to those used for e-commerce. Several applications demand concurrent and consistent access to the stored data by multiple writers and readers. The consistency model we adopt is atomicity. Atomic consistency gives the users of the data service the impression that the various concurrent read and write operations happen sequentially. Also, ability to withstand failures and network delays are essential features of any robust DSS.

The traditional solution for emulating an atomic fault-tolerant shared storage system involves replication of data across the servers. Popular replication-based algorithms appear in the work by Attiya, Bar-Noy and Dolev [2] (we refer to this as the ABD algorithm) and also in the work by Fan and Lynch [3] (which is referred to as the LDR algorithm). Replication based strategies incur high storage costs; for example, to store a value (an abstraction of a data file) of size 1 TB across a 100 server system, the ABD algorithm replicates the value in all the 100 servers, which blows up the worst-case storage cost to 100 TB. Additionally, every write or read operation has a worst-case communication cost of 100 TB. The communication cost, or simply the cost, associated with a read or write operation is the amount of total data in bytes that gets transmitted in the various messages sent as part of the operation. Since the focus in this paper is on large data objects, the storage and communication costs include only the total sizes of stable storage and messages dedicated to the data itself. Ephemeral storage and the cost of control communication is assumed to be negligible. Under this assumption, we further normalize both the storage and communication costs with respect to the size of the value, say \( v \), that is written, i.e., we simply assume that the size of \( v \) is 1 unit (instead of 1 TB), and say that the worst-case storage or read or write cost of the ABD algorithm is \( n \) units, for a system consisting of \( n \) servers.

Erasure codes provide an alternative way to emulate fault-tolerant shared atomic storage. In comparison with replication, algorithms based on erasure codes significantly reduce both the storage and communication costs of the implementation. An \([n,k]\) erasure code splits the value \( v \) of size 1 unit into \( k \) elements, each of size \( \frac{1}{k} \) units, creates \( n \) coded elements, and stores one coded element per server. The size of each coded element is also \( \frac{1}{k} \) units, and thus the total storage cost across the \( n \) servers is \( \frac{n}{k} \) units. For example, if we use an \([n = 100, k = 50] \) MDS code, the storage cost is simply 2 TB, which is almost two orders of magnitude lower than the storage in the case of ABD. A class of erasure codes known as Maximum Distance Separable (MDS) codes have the property that value \( v \) can be reconstructed from any \( k \) out of these \( n \) coded elements. In systems that are centralized and synchronous, the parameter \( k \) is simply chosen as \( n - f \), where \( f \) denotes the number of server crash failures that
need to be tolerated. In this case, the read cost, write cost and total storage cost can all be simultaneously optimized. The usage of MDS codes to emulate atomic shared storage in decentralized, asynchronous settings is way more challenging, and often results in additional communication or storage costs for a given level of fault tolerance, when compared to the synchronous setting. Even then, as has been shown in the past [1], [4], significant gains over replication-based strategies can still be achieved while using erasure codes. In [1] and [4] contain algorithms based on MDS codes for emulating fault-tolerant shared atomic storage, and offer different trade-offs between storage and communication costs.

A. Our Contributions

In this work we propose the Storage-Optimized Data-Atomic (SODA) algorithm for implementing atomic memory objects. SODA uses \([n,k]\) MDS codes, and is specifically designed to optimize the total storage cost for a given fault tolerance level. We also present a modification of SODA, called SODAerr, in order to handle the case where some of the non-faulty servers can return erroneous coded elements during a read operation. A summary of the algorithms and their features are provided below:

The SODA Algorithm: SODA assumes reliable point-to-point communication channels between any two processes - the collection of all readers, writers and servers - in the system. In a system consisting of \(n\) servers, for tolerating \(1 \leq f \leq \frac{n-1}{2}\) server crashes, SODA uses an \([n,k]\) MDS code with \(k = n - f\). Each server at any point during the execution of the algorithm stores at most one coded element, and thus, SODA has a worst-case total storage cost of \(\frac{n}{n-f}\). We prove the liveness and atomicity properties of SODA in the multi-writer multi-reader (MWMR) setting, for executions in which at most \(f\) servers crash. Any number of writer or reader processes may fail during the execution.

We construct a message-disperse primitive and use it in the write and read operations in SODA. The primitive is used by a process \(p\) to disperse a message \(m\) to all the non-faulty servers. The message \(m\) can be either meta-data alone or one that involves the value \(v\) along with a tag (where the tag is used to identify the version associated with the value); slightly differing implementations are used in the two cases. Meta-data refers to data such as ids, tags etc. which are used by various operations for book-keeping. In situations where \(m\) consists only of meta-data, the primitive ensures that if a server \(s \in S\) receives \(m\), then the same message \(m\) is sent to every server \(s' \in S\) by some process in the set \(\{p\} \cup S\). Thus if \(s'\) is non-faulty, it eventually receives \(m\) since the point-to-point channels are assumed reliable. During write operations, the writer uses the message-disperse primitive where \(m\) is the value \(v\) to be written. In this case, the primitive ensures that every non-faulty server receives the coded element that is targeted for local storage in that server. The primitive can tolerate up to \(f\) server failures and also the failure of the process \(p\). The idea here is to ensure that the uncoded value \(v\) is sent to \(f+1\) servers, so that at least one non-faulty server receives \(v\). This non-faulty server further computes and sends the corresponding coded elements to the remaining \(n - f\) servers. We show that the communication cost for a write operation, implemented on top of the message-disperse primitive, is upper bounded by \(\frac{5f^2}{n}\).

The read operations in SODA use a reader-registration and relaying technique similar to the one used in [5], where the authors discuss the use of erasure codes for Byzantine fault tolerance. For successful decoding, a reader must collect \(k\) coded elements corresponding to one particular tag. The reader registers itself with all non-faulty servers, and these servers send their respective (locally stored) coded elements back to the reader. Further, each non-faulty server also sends to the reader the coded elements it receives as part of concurrent write operations. Such relaying, by the servers, is continued until the reader sends a message acknowledging read completion. SODA uses a server-to-server communication mechanism to handle the case where a reader might fail after invoking a read operation. This internal communication mechanism exchanges only metadata and ensures that no non-faulty server relays coded elements forever to any reader. No such mechanism is used in [5] to handle the case of a failed reader. The read cost of SODA is given by \(\frac{n}{n-f} (\delta_w + 1)\), where \(\delta_w\) denotes the number of writes that are concurrent with the particular read. Since \(\delta_w\) might vary across different reads, the cost also varies across various reads, and hence we say that the read cost is elastic. The parameter \(\delta_w\) appears only as part of the analysis; its knowledge is not necessary to ensure liveness or atomicity.

We also carry out a latency analysis of successful write/read operations in SODA. The analysis assumes that latency arises only from the time taken for message delivery, and that computations at processes are fast. Under the assumption that the delivery time of any message is upper bounded by \(\Delta\) time units, we show that every successful write and read operation completes in \(5\Delta\) and \(6\Delta\) time units, respectively. The read time in this model of latency analysis turns out to be independent of the number of concurrent writers in the system.

The SODAerr Algorithm: The SODAerr algorithm is designed to handle the additional case where some of the servers can return erroneous coded elements during a read operation. The added feature of the algorithm is useful in large scale DDSs, where commodity hard disks are often used to achieve scalability of storage at low costs. In such systems, a coded element accessed by the server from its local hard-disk can be erroneous, i.e., the server obtains an arbitrary valued element instead of what was expected; however the server is not aware of the error when it sends this element back to the reader. The SODAerr algorithm provides a framework for tackling local disk read errors via the overall
erasure code across the various servers, without the need for expensive error protection mechanisms locally at each server. Specifically, in order to tolerate \( f \) server failures (like in SODA) and \( e \) error-prone coded elements, SODA\(_{err}\) uses an \([n, k]\) MDS code such that \( n - k = 2e + f \). We assume that no error occurs either in meta data or in temporary variables, since these are typically stored in volatile memory instead of local hard disk. SODA\(_{err}\) also guarantees liveness and atomicity in the MWMR setting, while maintaining an optimized total storage cost of \( \frac{n-n-f-2e}{n-f-2e} \). The write cost is upper bounded by \( 5f^2 \), and the read cost is given by \( \frac{n}{n-f-2e}(\delta _w + 1) \).

B. Comparison with Other Algorithms, and Related Work

We now compare SODA with the algorithms in [1] and [4], which are also based on erasure codes for emulating fault-tolerant atomic memory objects. In [1], the authors provide two algorithms - CAS and CASGC - based on \([n, k]\) MDS codes, and these are primarily motivated with a goal of reducing the communication costs. Both algorithms tolerate up to \( f = \frac{n-k}{2} \) server crashes, and incur a communication cost (per read or write) of \( \frac{n}{n-2f} \). The CAS algorithm is a precursor to CASGC, and its storage cost is not optimized. In CASGC, each server stores coded elements (of size \( \frac{1}{r} \)) for up to \( \delta + 1 \) different versions of the value \( v \), where \( \delta \) is an upper bound on the number of writes that are concurrent with a read. A garbage collection mechanism, which removes all the older versions, is used to reduce the storage cost. The worst-case total storage cost of CASGC is shown to be \( \frac{n}{n-2f}(\delta + 1) \). Liveness and atomicity of CASGC are proved under the assumption that the number of writes concurrent with a read never exceeds \( \delta \). In comparison, SODA is designed to optimize the storage cost rather than communication cost. We now note the following important differences between CASGC and SODA. (i) In SODA, we use the parameter \( \delta _w \), which indicates the number of writes concurrent with a read, to bound the read cost. However, neither liveness nor atomicity of SODA depends on the knowledge of \( \delta _w \); the parameter appears only in the analysis and not in the algorithm. (ii) While the effect of the parameter \( \delta \) in CASGC is rather rigid, the effect of \( \delta _w \) in SODA is elastic. In CASGC, any time after \( \delta + 1 \) successful writes occur during an execution, the total storage cost remains fixed at \( \frac{n}{n-2f}(\delta + 1) \), irrespective of the actual number of concurrent writes during a read. (iii) For a given \([n, k]\) MDS code, CASGC tolerates only up to \( f = \frac{n-k}{2} \) failures, whereas SODA tolerates up to \( f = n - k \) failures. A comparison of the performance numbers at \( f_{max} = \left\lfloor \frac{n-1}{2} \right\rfloor \) is shown in Table I. Note that \( f_{max} \) is the maximum number of failures for which either of these algorithms can be designed. Also note that \( f_{max} \) denotes the maximum of failures that can be tolerated by the ABD algorithm, as well.

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Write Cost</th>
<th>Read Cost</th>
<th>Total storage cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>ABD</td>
<td>( \frac{n}{f} )</td>
<td>( \frac{n}{f} )</td>
<td>( \frac{n}{f} )</td>
</tr>
<tr>
<td>CASGC</td>
<td>( \frac{n}{2} )</td>
<td>( \frac{n}{2} )</td>
<td>( \frac{n}{2}(\delta + 1) )</td>
</tr>
<tr>
<td>SODA</td>
<td>( O(n^2) )</td>
<td>( \leq 2(\delta _w + 1) )</td>
<td>( \leq 2 )</td>
</tr>
</tbody>
</table>

Table I

Performance comparison of ABD, CASGC and SODA, for \( f = f_{max} = \frac{n}{2} - 1 \). We assume \( n \) to be an even number.

In [6], the authors consider algorithms that use erasure codes for emulating regular registers. Regularity [7], [8] is a weaker consistency notion than atomicity. Distributed storage systems based on erasure codes, and requiring concurrency/consistency are also considered in [9]. Applications of erasure codes to Byzantine fault tolerant DSSs are discussed in [5], [10], [11]. RAMBO [12] and DynaStore [13] are implementations of MWMR atomic memory objects in dynamic DSSs, where servers can enter or leave the system.

Document Structure: Models and definitions appear in Section II. Implementation and properties of the message-disperse primitives are discussed in Section III. Description and analysis of the SODA algorithm are in Sections IV and V, respectively. SODA\(_{err}\) algorithm is presented in Section VI. Section VII concludes. Due to space constraints, proofs are omitted.

II. MODELS AND DEFINITIONS

In this section, we describe the models of computation, explain the concepts of atomicity, erasure codes, and the performance metrics used in the paper.

Asynchrony and Crash Failures: We consider a distributed system consisting of asynchronous processes of three types: a set of readers and writers, called clients, and a set of \( n \) servers. Each of these processes is associated with a unique identifier, and we denote the sets of IDs of the readers, writers and servers as \( R \), \( W \) and \( S \), respectively. The set of IDs forms a totally ordered set. The reader and writer processes initiate read and write operations, respectively, and communicate with the servers using messages. Also, any client initiates a new operation only after the previous
operations, if any, at the same client has completed. We refer to this as the **well-formedness** property of an execution. All processes run local computations until completion or crash failure. Any of the number of clients can fail. We assume up to \( f \), such that, \( f \leq \frac{n-1}{2} \), servers (out of the total \( n \)) may crash during any execution.

We assume that every client is connected to every server through a reliable communication link. This means that as long as the destination process is non-faulty, any message sent on the link is guaranteed to eventually reach the destination process. The model allows the sender process to fail after placing the message in the channel; message-delivery depends only on whether the destination is non-faulty. We also assume reliable connectivity between every pair of servers in the system. We do not make any assumption regarding relative order of message delivery in the same channel.

**Liveness:** By liveness, we mean that during any well-formed execution of the algorithm, any read or write operation initiated by non-faulty reader or writer completes, despite the crash failure of any other clients and up to \( f \) server crashes.

**Atomicity:** A shared atomic memory can be emulated by composing individual atomic objects. Therefore, we aim to implement only one atomic read/write memory object, say \( v \), in an execution when the invocation step of \( v \) is requested by a writer \( w \). For any execution of a memory service, if an operation \( \pi \) is incomplete in an execution when the invocation step of \( \pi \) does not have the associated response step; otherwise we say that \( \pi \) is complete. In an execution, we say that an operation (read or write) \( \pi_1 \) precedes another operation \( \pi_2 \), if the response step for \( \pi_1 \) precedes the invocation step of \( \pi_2 \). Two operations are concurrent if neither precedes the other. The following lemma is a restatement of the sufficiency condition for atomicity presented in [14].

**Lemma 2.1:** For any execution of a memory service, if all the invoked read and write operations are complete, then the operations can be partially ordered by an ordering \( \prec \), so that the following properties are satisfied:

**P1.** The partial order (\( \prec \)) is consistent with the external order of invocation and responses, i.e., there are no operations \( \pi_1 \) and \( \pi_2 \), such that \( \pi_1 \) completes before \( \pi_2 \) starts, yet \( \pi_2 \prec \pi_1 \).

**P2.** All operations are totally ordered with respect to the write operations, i.e., if \( \pi_1 \) is a write operation and \( \pi_2 \) is any other operation then either \( \pi_1 \prec \pi_2 \) or \( \pi_2 \prec \pi_1 \).

**P3.** Every read operation ordered after any writes returns the value of the last write preceding it (with respect to \( \prec \)), and if no preceding writes is ordered before it then it returns the initial value of the object.

**Erasure coding:** We use \([n, k]\) linear MDS codes [15] to encode and store the value \( v \) among the \( n \) servers. An \([n, k, d]\) linear code \( C \) over a finite field \( \mathbb{F}_q \) (containing \( q \) elements) is a \( k \)-dimensional subspace of the vector space \( \mathbb{F}_q^n \). The parameter \( d \) is known as the minimum distance of the code \( C \) and denotes the minimum Hamming weight of any non-zero vector in \( C \). The well known Singleton bound [16] states that \( d \leq n - k + 1 \). Codes that achieve equality in this bound are known as MDS codes, such codes are known to exist for any \((n, k)\) pair such that \( k \leq n \), e.g., Reed-Solomon codes [17].

We use functions \( \Phi \) and \( \Phi^{-1} \) to denote the encoder and decoder associated with the code \( C \). For encoding, \( v \) is divided into \( k \) elements \( v_1, v_2, \ldots, v_k \) with each element having a size \( \frac{1}{k} \). As mentioned in Section I, we assume that the value \( v \) is of size 1 unit. The encoder takes the \( k \) elements as input and produces \( n \) coded elements \( c_1, c_2, \ldots, c_n \) as output, i.e., \( [c_1, \ldots, c_n] = \Phi([v_1, \ldots, v_k]) \). For ease of notation, we will simply write \( \Phi(v) \) to mean \([c_1, \ldots, c_n] \). The vector \([c_1, \ldots, c_n] \) is often referred to as the codeword corresponding to the value \( v \). Each coded element \( c_i \) also has a size \( \frac{1}{k} \). In our scheme we store one coded element per server. We use \( \Phi_i \) to denote the projection of \( \Phi \) on to the \( i \)th output component, i.e., \( c_i = \Phi_i(v) \). Wlog, we associate the coded element \( c_i \) with server \( i \), \( 1 \leq i \leq n \).

A code with minimum distance \( d \) can tolerate up to \( d - 1 \) erasures among the \( n \) coded elements. Since we wan tolerate up to \( f \) server failures while using MDS codes, we pick the dimension \( k \) of the MDS code as \( k = n - f \). Since we store one coded element per server, by using an \([n, n-f]\) MDS code, we get the property that the original value \( v \) can be recovered given the contents of any \( n - f \) servers. If \( C = \{c_i, i \in \mathcal{I}\} \) denotes any multiset of \( k \) coded elements for some \( \mathcal{I} \subset [n], |\mathcal{I}| = k \), we write \( v = \Phi^{-1}(C) \) to indicate that \( v \) is decodable from \( C \). We implicitly assume that the process that is invoking the decoder is aware of the index set \( \mathcal{I} \) corresponding to the \( k \) coded elements.

**Storage and Communication Cost:** We define the (worst-case) total storage cost as the size of the data stored across all servers, at any point of the execution of the algorithm. As mentioned in Section I, the storage cost is normalized with respect to the size of the value \( v \), which is equivalent to computing the storage cost under the assumption that \( v \) has size 1 unit. We assume metadata, such as version number, process ID, used by various operations is of negligible size and is hence ignored in the calculation of storage or communication cost. The communication cost associated with a read or write operation is the size of the total data that gets transmitted in the messages sent as part of the operation. As with storage cost, we ignore the communication cost associated with metadata transmissions.
III. THE message-disperse PRIMITIVES

Now we discuss the message-disperse services that are used to disseminate messages in SODA. They have the property that if a message is delivered to any server in \( S \), then the same message (or a derived message) is eventually delivered to every non-faulty server in \( S \). The services are implemented on top of point-to-point reliable channels. The services are provided in terms of (i) the MD-META primitive, used for the metadata delivery, and (ii) the MD-VALUE primitive, used for delivering the coded elements for the values. The MD-META (or MD-VALUE) primitive is invoked by the send-event \( md-meta-send \) (or \( md-value-send \)) at some process \( p \), and results in delivery-event \( md-meta-deliver \) (or \( md-value-deliver \)) at any non-faulty process \( s \). In order to reason about the properties of the protocols, we require precise descriptions of the flow of messages among the client, server and communication channel processes. Therefore, we specify their implementations using the language of IO Automata (see Lynch [14]). Due to space constraints, only the MD-VALUE primitive is discussed in detail. The MD-META primitive differs from the MD-VALUE primitive only in a minor way, and the difference alone will be discussed.

A. MD-VALUE primitive

The MD-VALUE primitive is to be used in SODA to deliver the coded elements and the associated tags, which are unique version identifiers for the object value, to every non-faulty server. Below we first define the primitive, and its desired consistency properties. Subsequently, we present the implementation of the primitive.

Definition 1: MD-VALUE primitive sends message containing tag \( t \) and value \( v \) from a sender process \( p \in S \) to the set of server processes in \( S \), such that each non-faulty process in \( S \) delivers its corresponding coded elements. The following events define the primitive, to an external user: (i) \( md-value-send(t,v)_p \): an invocation event, at a writer \( p \in W \), that submits the version \( t \) and the value \( v \) for delivery of the coded elements, and (ii) \( md-value-deliver(t,c_p)_p \): an output event, at a server \( p \in S \), that delivers the coded element \( c_p = \Phi_p(v) \) to the server \( p \).

Following are the consistency properties that we expect from an implementation (also called as protocol) of the primitive, under the assumption that all executions are well-formed.

Definition 2: Consistency-Properties (i) validity: if event \( md-value-deliver(t,c_s)_s \) takes place at some server \( s \in S \), then it is preceded by the event \( md-value-send(t,v)_w \) at a writer \( w \), where \( t \in T \) and \( c_s = \Phi_s(v) \); and (ii) uniformity: if event \( md-value-deliver(t,c_p)_s \) takes place at some server \( s \in S \), and as long as the number of server crashes during the execution is at most \( f \), then the event \( md-value-deliver(t,c'_p)_s \) occurs at every non-faulty process \( s' \in S \), where \( c'_p = \Phi_{s'}(v) \).

We note that the uniformity property must hold even if the writer \( w \) itself crashes after the invocation of the event \( md-value-send(t,v)_w \).

Implementation: The IO Automata specifications of a sender, MD-VALUE-SENDER\( _p,p \in W \), and the receiving servers, MD-VALUE-SERVER\( _s,s \in S \), for the MD-VALUE protocol are given in Figs. 1 and 2, respectively. The overall protocol is obtained by composing the above two automata and the underlying automata for the point-to-point reliable channels (see Lynch [14]). We first describe the data types, state variables and transitions that appear in the protocol, and then present a description of the protocol.

Data Types and State Variables: In the IO Automata specification of MD-VALUE, for any value \( v \in V \) the coded element corresponding to \( s \in S \) is denoted as \( c_s = \Phi_s(v) \). \( M_{ID} \equiv S \times \mathbb{N} \) is the set of unique message identifiers. Each message is one of two types: \( \text{TYPES} = \{ \text{"full"}, \text{"coded"} \} \). In MD-VALUE-SENDER\( _p \) boolean state variables \( \text{failed} \) and \( \text{active} \) are initially false. The state variable, \( mCount \), keeps track of the number of times \( md-value-send(\ast)_p \) has been invoked at sender process \( p \), and initially this is 0. The variable \( \text{send buffs} \) is a FIFO queue with elements of the form \( (M_{ID} \times (T \times V) \times \text{TYPES}) \times S \), and initially this is empty. State variable \( mID \in M_{ID} \) holds a unique message identifier corresponding to an invocation of the protocol, initially \((0,p)\). Variable \( currMsg \) holds the message that is being sent, initially \( \bot \). In an automaton MD-VALUE-SERVER\( _s \) we have the following state variables. The state variable \( \text{failed} \) is initially set to false. The variable \( \text{status} \) is a map from keys in \( M_{ID} \) to a value in \{ready, sending, delivered\}. The variable \( \text{content} \) is a map from keys in \( M_{ID} \) to a message in \( \mathbb{F}_p \), initially \( \bot \). The variable \( \text{outQueue} \) is a FIFO queue with elements of the form \( (M_{ID} \times (V \cup \mathbb{F}_p) \times \text{TYPES}) \), initially empty. Transitions: In MD-VALUE-SENDER\( _p \) the input action \( md-value-send(t,v)_p \) invokes the protocol with tag \( t \) and value \( v \), and the output transition \( md-value-send-ack(t,v)_p \) occurs when all the messages with \( t \) and \( v \) are sent. The action send(\ast)_p adds messages to the channels. Automaton MD-VALUE-SERVER\( _s \) has two input actions \( \text{recv(\ast)}_s,s \) corresponding to the “full” and “coded” types for receiving the values and coded elements, respectively, and the action send(\ast)_s sends message to other servers through the channels. The output action \( md-value-deliver(t,c)_s \) delivers the tag \( t \) and coded element \( c \) corresponding to server \( s \). Explanation of the Protocol: The basic idea of the MD-VALUE implementation is as follows: the sender \( p \in W \) invokes input action \( md-value-send(t,v)_p \) at the automaton MD-VALUE-SENDER\( _p \). The tag \( t = \text{"full"} \) and value \( v \) are sent to the set of first \( f + 1 \) servers \( D = \{s_1,s_2,\ldots,s_{f+1}\} \) among the set of all servers. Recall that in our model, we assume an ordering of the \( n \) servers.
in the system, and hence it makes sense to talk about the first \(f+1\) servers. Further, the message \(m = (t, v)\) is sent to the servers respecting the ordering of the servers, i.e., \(p\) sends \(m\) to \(s_i\) before sending to \(s_{i+1}, 1 \leq i \leq f\).

Let us next explain MD-VALUE-SERVER, \(s \in S\). In this, let us first consider the case when \(s = s_i \in D = \{s_i, 1 \leq i \leq f + 1\}\). In this case, the server \(s_i\) upon receiving \(m\) for the first time, sends \(m\) to every process in the set \(\{s_{i+1}, s_{i+2}, \ldots, s_{f+1}\}\). Once again the message is sent to these \(f+1−i\) servers respecting the ordering of the servers. As a second step, the server \(s_i\), for every server \(s_i' \in S−D\), computes the coded element \(c_{s_i'} = \Phi_{s_i'}(v)\) and sends the message \((t = \text{"coded"}, c_{s_i'} = \Phi_{s_i'}(v))\) to the server \(s_i'\). Finally, the server \(s_i\) computes its own coded element \(c_s = \Phi_s(v)\) and delivers it locally via the output action md-value-deliver\((t, c_s)_s\). Let us next consider the case when \(s \in S−D\). In this case, the server \(s\) simply delivers the received coded-element \(c_s\) via the output action md-value-deliver\((t, c_s)_s\). Next, we claim the properties of the protocol.

**Theorem 3.1:** Any well-formed execution of the MD-VALUE protocol satisfies the consistency properties.

Next theorem says that once a message is delivered via the primitive, all the associated messages get automatically removed from the system, i.e., there is no bloating-up of state variables.

**Theorem 3.2:** Consider a well-formed execution \(\beta\) of the MD-VALUE protocol such that the event \(\text{md-value-send}(t, v)_p\) appears in \(\beta\). Then, for any \(s \in S\) there exists a state \(\sigma^*\) in \(\beta\) after the event \(\text{md-value-send}(t, v)_p\) such that in the automats MD-VALUE-SENDER, \(p\) and MD-VALUE-SERVER, \(s\) for every \(s \in S\), the following is true: \((i)\) either failed, is true or \((ii)\) in any state in \(\beta\) following \(\sigma^*\), none of the state variables in automata contains \(v\) or any of the coded elements \(c_s\) for \(s' \in S\).

### B. MD-META primitive

The MD-META primitive ensures that if a server \(s \in S\) delivers some metadata \(m\), from a metadata alphabet \(M_m\), then it is delivered at every non-faulty server \(s' \in S\). The primitive is defined via the events \(\text{md-meta-send}(m)_p\) and \(\text{md-meta-deliver}(m)_p\). The difference with respect to the MD-VALUE primitive is that here we simply deliver the transmitted message \(m\) itself at all the servers, while the MD-VALUE only delivered the corresponding coded-elements. Thus the implementation of MD-META primitive is in fact simpler; the main difference is that while sending messages to the servers in \(S−D\) by a server \(s_i \in D\), \(s_i\) simply sends \(m\), whereas in the MD-VALUE protocol recall that \(s_i\) calculated and sent only the corresponding coded elements.

**IV. SODA ALGORITHM**

In this section, we present the SODA algorithm. The algorithm employs majority quorum, and uses erasure codes to reduce storage cost. Detailed algorithmic steps for the reader, writer and server processes are presented in Fig. 3, 4 and 5, respectively. For simplicity, we only present the pseudo-code instead of a formal description using IO Automata. SODA uses an \([n, k]\) MDS code with \(k = n−f\). Atomicity and liveness are guaranteed under the assumption that at most \(f\) servers crash during any execution. SODA can be designed for any \(f\) such that \(f < \frac{n−1}{2}\). For version control of the object values we use tags. A tag \(t\) is defined as a pair \((z, w)\), where \(z \in \mathbb{N}\) and \(w \in \mathcal{W}\) is ID of a writer. We use \(T\) to denote the set of all possible tags. For any two tags \(t_1, t_2 \in T\) we say \(t_2 > t_1\) if \((i)\) \(t_2.z > t_1.z\) or \((ii)\) \(t_2.z = t_1.z\) and \(t_2.w > t_1.w\).

Each server stores three state variables: \((i)\) \((t, c_s)\), tag and coded element pair, which is initially set to \((t_0, c_0)\), \((ii)\) \(R_c\), a set of pairs of the form \((r, t_r)\), where the pair \((r, t_r)\) indicates the fact that the reader \(r\) is being currently served by this server. Here \(t_r\) denotes the tag requested by the reader \(r\). Initially, \(R_c = \emptyset\), \((iii)\) \(H\), a set of tuples \((t, s', r)\) that is used to indicate the fact that the server \(s'\) has sent a coded element corresponding to the tag \(t\), to reader \(r\). Initially, \(H = \emptyset\).

Two types of messages are sent, messages that carry metadata, and messages that comprise in part or full an object value. The messages sent from the clients are labeled with phase names, viz., READ-GET, READ-VALUE, READ-COMPLETE and WRITE-GET. The server to server messages are labeled as READ-DISPERSE. Also, in some phases of SODA, the message-disperse primitives MD-META and MD-
**Fig. 1** MD-VALUE-SENDER<sub>p</sub> Automaton: Signature, State and Transitions at sender p ∈ W.

**Signature:**
2: Input: md-value-send(<t, v>), t ∈ T, v ∈ V
4: Internal: fail<sub>s</sub>
6: Output: send((<mID, (t, v), “full”>), p<sub>s</sub>), mID ∈ M<sub>D</sub>, t ∈ T, v ∈ V
8: md-value-send-ack(t)<sub>p</sub>, t ∈ T

**State:**
10: failed, a Boolean, initially false
12: write, a value
14: mID ∈ N × S, initially (0, p)
16: currTag ∈ T, initially ⊥

**Transitions:**
18: Transitions: md-value-send(<t, v>), t ∈ T, v ∈ V
20: if ¬failed then
22: let D = {s<sub>1</sub>, ..., s<sub>F</sub>} be the subset of first f + 1 servers of S
24: send_buff ← 

**Effect:**
26: Output md-value-send-ack(t)<sub>p</sub>, t ∈ T
28: Preconditions: ¬failed
30: send_buff ← ∅
32: i = currTag
34: Effect: active ← false
currTag ← ⊥
36: Output send((<mID, (t, v), “full”>), p<sub>s</sub>), mID ∈ M<sub>D</sub>, t ∈ T, v ∈ V
38: ¬failed
40: Effect: send_buff ← tail(send_buff)

**Fig. 2** MD-VALUE-SERVER<sub>s</sub> Automaton: Signature, State and Transitions at server s ∈ S<sub>i</sub>, 1 ≤ i ≤ n

**Signature:**
2: Input: recv((<mID, (t, v), “full”>), r<sub>s</sub>), mID ∈ M<sub>D</sub>, t ∈ T, v ∈ V, r ∈ S
4: Internal: fail<sub>s</sub>
6: Output: md-values-deliver(t)<sub>C</sub>, t ∈ T, c ∈ T<sub>q</sub>
8: md-values-deliver(c)<sub>s</sub>, mID ∈ M<sub>D</sub>, t ∈ T, u ∈ V ∪ T<sub>q</sub>, r ∈ S
10: State: failed, a Boolean initially false
12: write, a value
14: outQueue, a queue, initially empty

**Transitions:**
18: Transitions: recv((<mID, (t, v), “full”>), r<sub>s</sub>)
20: if ¬failed then
22: let D = {v<sub>1</sub>, ..., v<sub>F</sub>} be a subset of S s.t. |D| = f + 1 + i
24: for s′ ∈ D do
26: status(mID) ← sending content(mID) ← (t, v)<sub>s</sub>
30: if status(mID) ≠ delivered then
32: Effect: if ¬failed then
34: Outputs send((<mID, (t, u), “full”>), s<sub>i</sub>), r<sub>s</sub>, mID ∈ M<sub>D</sub>, t ∈ T, v ∈ V, r ∈ S
36: if outQueue(mID) = ∅ then
38: stat(mID) ← ready
40: Effect: status(mID) ← delivered content(mID) ← ⊥

**Value** are used as services.

**Write Operation:** Assume that a writer w wishes to write a value v. Recall that an [n, k] MDS code creates n coded elements after encoding v. The goal is to store one coded element per server. In order to optimize storage cost, at any point of the execution, each server only stores the coded element corresponding to one particular tag. The write operation consists of two phases. In the first phase, the writer queries all the servers for the local tags that are stored, waits for a response from a majority and then picks the highest tag \( m_{\text{max}} \). The writer w creates a new tag given by \( t_w = (m_{\text{max}} + 1, w) \). In the second phase, the writer sends the message \((t_w, v)\) to all servers in \( S \), via md-meta-send(READ-GET, (r, t_w)). The algorithm is designed so that \( r \) decodes a value corresponding to some tag \( t \geq t_w \). Any server that receives \( m \) registers the \((r, t)\) pair locally. Here, we use the term register \((r, t)\) to mean adding the pair \((r, t)\) to \( R_c \). By executing the step \( R_c \leftarrow R_c \cup \{(r, t)\} \) during the read-value phase at the server.

\((t_w, c_w)\). In any case, the server sends an acknowledgment back to the writer w. A few additional steps are performed by the server while responding to the message \((t_w, c_w)\) (in response 3, Fig. 3). These will be explained as part of the read operation. Finally, the writer terminates after receiving acknowledgment from at least \( k \) servers.

**Read Operation:** Like a writer, a reader \( r \) during the first phase polls all the servers for the locally stored tags, waits for a response from a majority and then picks the highest tag \( m_{\text{max}} \), which we call here as \( t_r \). In the second phase, the reader sends the message \( m = (r, t_r) \) to all servers in \( S \), via md-meta-send(READ-GET, (r, t_r)). The algorithm is designed so that \( r \) decodes a value corresponding to some tag \( t \geq t_r \). Any server that receives \( m \) registers the \((r, t)\) pair locally. Here, we use the term register \((r, t)\) to mean adding the pair \((r, t)\) to \( R_c \). By executing the step \( R_c \leftarrow R_c \cup \{(r, t)\} \) during the read-value phase at the server.
Similarly, by unregister we mean the opposite, i.e., remove the pair from $R_c$. The server sends the locally available $(t, c_s)$ pair to the reader if $t \geq t_r$. Furthermore, every time a new message $(t_w, c_s)$ is received at the server, due to some concurrent write with $(t_w, v)$, the server sends the message $(t_w, c_s)$ to reader if $t_w > t_r$. Note that there can be situations where the server does not store $c_s$, locally, for instance, if the local tag $t$ is higher than the writer’s tag $t_w$, but simply sends the coded element $c_s$ to the reader. The reader keeps accumulating $(t, c_s)$ pairs it receives from various servers, until the reader has $k$ coded elements corresponding to some tag $t_r$. At this point the reader decodes the value $(t_r, v)$. Before returning the value $v$, the reader sends a READ-COMplete message, by calling md-meta-send(READ-COMplete, (r, t_r)), to the servers, so that the reader can be unregistered by the active servers, i.e., $(r, t_r)$ is removed from their local variable $R_c$.

The algorithm ensures that a failed reader is not sent messages indefinitely by any server. Assume that the pair $(r, t_r)$ is registered at server $s$, to continue sending coded elements from new writes for tags higher than or equal to $t_r$. Once $k$ distinct coded elements for such a tag is known to have been sent, reader $r$ will be unregistered and server $s$ no longer sends messages for that tag. In order to implement this, any server $s'$ that sends a coded element corresponding to tag $t'$ to reader $r$ also sends $(s', t', v)$ to all the other servers, by calling md-meta-send(READ-DISPERSE, (s', t', r)). The server $s$ which receives the $(s', t', r)$ tuple adds it to a local history variable $H$, and is able to keep track of the number of coded elements sent to the registered reader $r$. So, server $s$ eventually unregisters reader $r$ and also cleans up history variable $H$ by removing the tuples corresponding to $r$.

Additional Notes on SODA: (1) Server $s$ accumulates any received $(s', t', r')$ tuple in its history variable $H$, even if reader $r'$ has not yet been registered by the server. The use of the message-disperse primitive by $r'$, by calling md-meta-send(READ-DISPERSE, (s', t', r')), to register the pair $(r', t', r)$, ensures that $s$ will also eventually register $r'$. Once $r'$ gets registered at $s$, these entries will be used by $s$ to figure out if $r'$ can be unregistered.

(2) Since we do not assume any order in message arrivals, a READ-COMplete message may arrive at server $s$ from reader $r$ even before the server $s$ receives the request for registration from $r$. In this case, during the response to READ-COMplete phase, the server adds the tuple $(t_0, s, r)$ to the set variable $H$, where $t_0$ is a dummy tag. If the server is non-faulty, we know that the registration request from the reader will arrive at $s$ at some future point in time. The reader $r$ is registered by server $s$ in response to read-value phase only if the tuple $(t_0, s, r)$ is not in $H$.

(3) During each read operation the reader appends a unique identifier (eg: a counter or a time stamp) in addition to its own id $r$. Though we show in the next Section that every server will eventually stop sending coded elements to any reader $r$, it can happen that the entries in $H$ corresponding to $r$ are not entirely cleared. The usage of unique identifiers for distinct read operations from the same reader ensures that the stale entries in $H$ do not affect new reads. To keep the presentation simple, we do not explicitly indicate these identifiers in Fig. 4.

V. ANALYSIS OF SODA

In this section, we present our claims regarding the liveness and atomicity properties of the SODA algorithm. We also give bounds on the storage and communication costs.

A. Liveness and Atomicity

Recall that by liveness, we mean that during any execution of the SODA, any read or write operation initiated by non-
faulty reader or writer completes, despite the crash failure of any other client and up to \( f \) server crash failures.

**Theorem 5.1**: Let \( \beta \) be a well-formed execution of SODA. Also, let \( \Pi \) denote the set of all client operations that take place during the execution. Then every operation \( \pi \in \Pi \) associated with a non-faulty client completes.

In order to prove the atomicity property of SODA for any well-formed execution \( \beta \), we define a partial order \( (\prec) \) in \( \Pi \) and then show that \( \prec \) satisfies the properties \( P1, P2 \) and \( P3 \) given in Lemma 2.1. For every operation \( \pi \in \Pi \) corresponding to a non-faulty reader or writer, we associate a \((tag, value)\) pair that we denote as \((tag(\pi), value(\pi))\). For a write operation \( \pi \), we define the \((tag(\pi), value(\pi))\) pair as the message \((t_w, v)\) which the writer sends in the write-put phase. If \( \pi \) is a read, we define the \((tag(\pi), value(\pi))\) pair as \((t_{read}, v)\) where \( v \) is the value that gets returned in the read-complete phase, and \( t_{read} \) is the associated tag.

The partial order \( (\prec) \) in \( \Pi \) is defined as follows: For any \( \pi, \phi \in \Pi \), we say \( \pi \prec \phi \) if one of the following holds: (i) \( tag(\pi) \prec tag(\phi) \), or (ii) \( tag(\pi) = tag(\phi) \), and \( \pi \) and \( \phi \) are write and read operations, respectively.

**Theorem 5.2**: Any well-formed execution \( \beta \) of the SODA algorithm respects the atomicity properties \( P1, P2 \) and \( P3 \) given in Lemma 2.1.

**B. Storage and Communication Costs**

Below we state the storage cost associated with SODA. Recall our assumption that, for storage cost, we count only the data corresponding to coded elements that are locally stored, and storage cost due to meta-data and temporary variable are ignored.

**Theorem 5.3**: The worst-case total storage cost of SODA algorithm is given by \( \frac{n-k}{n-f} \).

We next state the communication cost for the write and read operations in SODA. Once again, note that we ignore the communication cost arising from exchange of meta-data.

**Theorem 5.4**: The communication cost of a successful write in SODA is upper bounded by \( 5f^2 \), i.e., \( O(f^2) \).

Towards deriving the read communication cost, we first observe the fact that no reader will be indefinitely sent messages by any non-faulty server.

**Theorem 5.5**: During the execution of SODA algorithm, any non-faulty server which registers a reader also unregisters it, eventually.

**Number of Writes Concurrent with a Read**: Consider a read operation initiated by a reader \( r \). Let \( T_1 \) denote the earliest time instant at which the reader \( r \) is registered by at least one of the servers. Also, let \( T_2 \) denote the earliest time instant at which \( r \) is unregistered by all non-faulty servers. From Theorem 5.5, we know that the time instant \( T_2 \) indeed exists (i.e., it is finite). We define the parameter \( \delta_w \) as the number of write operations which get initiated during the time interval \([T_1, T_2]\). The following theorem bounds the communication cost for a read operation in terms of \( \delta_w \).

**Theorem 5.6**: In SODA algorithm, the communication cost associated with a read operation is at most \( \frac{1}{n-\delta_w}(\delta_w+1) \).

**C. Latency Analysis**

In this section, we provide conditional latency bounds for successful read/write operations in SODA. Although SODA is designed for asynchronous message passing settings, in the case of a reasonably well-behaved network we can bound the latency of an operation. Assume that any message sent on a point-to-point channel is delivered at the corresponding destination (if non-faulty) within a duration \( \Delta > 0 \), and local computations take negligible amount of time compared to \( \Delta \). We do not assume knowledge of \( \Delta \) inside the algorithm. Thus, latency in any operation is dominated by the time taken for the delivery of all point-to-point messages involved. Under these assumptions, the latency bounds for successful write and read operations in SODA are as follows.

**Theorem 5.7**: The duration of a successful write and read operation in SODA is at most \( 5\Delta \) and \( 6\Delta \), respectively.

**VI. SODA_{err} FOR HANDLING ERRORS AND ERASURES**

In this section, we explain the usage of \([n, k]\) MDS codes for the SODA_{err} algorithm. Here the parameter \( k \) is chosen as \( k = n - f - 2e \). The encoding and distribution of \( n \) coded elements among the \( n \) servers remain same as above. While decoding, we require that we are able to tolerate any \( f \) missing coded elements as well as \( e \) erroneous coded-elements among the remaining elements. For example, assume that \( c_1, \ldots, c_{n-f} \) are available to the decoder - the servers which store the remaining coded elements might have crashed, where \( e \) out of these \( n-f \) elements are erroneous, and the decoder does not know the error locations. It is well known that \([n, k]\) MDS codes can tolerate any pattern of \( f \) erasures and \( e \) errors if \( k = n - f - 2e \). We use \( \Phi_{err}^{-1} \) to denote the decoder used to recover the value \( v \); in this example we have \( v = \Phi_{err}^{-1}(\{c_1, \ldots, c_{n-f}\}) \). Once again, we make the assumption that the decoder is aware of the index set \( \mathcal{I} \) corresponding to the \( n - f = k + 2e \) coded elements that are being used in the decoder.

Now we describe the modifications needed in SODA to implement SODA_{err}. In SODA, read errors can occur during the read-value phase, where the server is expected to send the locally stored coded element to the reader. We do not assume any error in situations where the server is only relaying a coded element, in response to the write-get phase, since this does not involve local disk reads. Also, we assume that tags are never in error, because tags being negligible in size can be either stored entirely in memory, or replicated locally for protection against disk read errors. SODA_{err} is same as SODA except for two steps (Fig. 6), which we describe next.

(i) **read-value** phase initiated by the reader: Any reader must wait until it accumulates \( k + 2e \) coded elements..
corresponding to a tag before it can decode. Recall that in the SODA algorithm, we only needed k coded elements before the reader can decode. Also note that the decoder for the \text{SODA}_\text{err} (which we denote as \Phi^{-1}_\text{err}) is different from that used for SODA, since now we must accept k + 2e coded elements of which certain e elements are possibly erroneous.

(ii) On recv\{\text{READ-DISPERSE}, (t, s', r)\}: A server checks if the number of coded elements sent (from (various) servers) to reader r corresponding to tag t is at least k + 2e, before deciding to unregister the reader r. We now state our claims regarding the performance guarantees of the \text{SODA}_\text{err}.

\textbf{Theorem 6.1: (Liveness):} Let $\beta$ be a well-formed execution of the \text{SODA}_\text{err} algorithm. Then every operation $\pi \in \Pi$ associated with a non-faulty client completes.

\textbf{Theorem 6.2: (Atomicity):} Any well-formed execution fragment $\beta$ of the \text{SODA}_\text{err} respects atomicity properties.

\textbf{Theorem 6.3:} (i) The total storage cost of $\text{SODA}_\text{err}$ is $\frac{n}{n-f-2e}$. (ii) The write cost of $\text{SODA}_\text{err}$ is at most $5f^2$, i.e., $O(f^2)$, and (iii) The read cost of $\text{SODA}_\text{err}$ is $\frac{n}{n-f-2e}(\delta_m+1)$, where $\delta_m$ is the number of writes which are concurrent with a read. The definition of $\delta_m$ is the same as in SODA.

\textbf{VII. Conclusion}

In this paper, we proposed the SODA algorithm based on $[n, k]$ MDS codes to emulate shared atomic objects in asynchronous DSSs. SODA tolerates $f = n-k$ crash failures and achieves an optimized storage cost of $\frac{n}{n-f}$: $\text{SODA}_\text{err}$, a modification of SODA, which tolerates both crash failures and data read errors. Next we plan to extend this work to (i) dynamic settings where servers enter or leave the system, and (ii) scenarios where background repairs are carried out to restore the contents of a crashed server.

\textbf{References}


