Semi-Fast Byzantine-tolerant Shared Register without Reliable Broadcast

Kishori M. Konwar
Broad Institute, MIT
kishori@csail.mit.edu

Saptaparni Kumar, Lewis Tseng
Boston College
{saptaparni.kumar, lewis.tseng}@bc.edu

***Abstract*—Shared register emulations on top of message-passing systems provide an illusion of a simpler shared memory system which can make the task of a system designer easier. Numerous shared register applications have a considerably high read to write ratio. Thus having algorithms that make reads more efficient than writes is a fair trade-off.**

Typically such algorithms for reads and writes are asymmetric and sacrifice the stringent consistency condition atomicity as it is impossible to have fast reads for multi-writer atomicity. Safety is a consistency condition has has gathered interest from both the systems and theory community as it is weaker than atomicity yet provides strong enough guarantees like "strong consistency" or read-my-write consistency. One requirement that is assumed by many researchers is that of the *reliable broadcast* (RB) primitive, which ensures the all or none property during a broadcast. One drawback is that such a primitive takes 1.5 rounds to complete.

This paper implements an efficient multi-writer multi-reader safe register without using a reliable broadcast primitive. Moreover, we provide fast reads or one-shot reads – our read operation can be completed in one round of client-to-server communication. Of course, this comes with the price of requiring more servers when compared to prior solutions assuming reliable broadcast. However, we show that this increased number of servers is indeed necessary as we prove a tight bound on the number of servers required to implement Byzantine-fault tolerant safe registers in a system without reliable broadcast.

We extend our results to data stored using erasure coding as well. We present an emulation of single-writer multi-reader safe register based on MDS code. The usage of MDS code reduces storage cost and communication cost. On the negative side, we also show that to use MDS code and achieve one-shot read at the same time, we need even more servers.

***Index Terms*—Byzantine faults, MDS codes, Safe registers, Reliable broadcast.**

I. Introduction

A longstanding vision of distributed computing has been to design algorithms that emulate reliable systems on unreliable components. For example, fault tolerant storage [1], reliable broadcast [2], atomic broadcast [3], failure detectors [4], etc. As we are increasingly dependent on services (online banking software, e-shopping, online auctions, etc) provided by distributed systems over the Internet, "fault-tolerance" is becoming a necessary property of these services in order to make them usable.

These faults can range from benign (crash-faults) to extremely adversarial (Byzantine faults). The term "Byzantine" fault represents the whole spectrum of failures imaginable ranging from malicious attacks and operator mistakes to software errors and conventional crashes. Driscoll et al. [5] presented a case study of Byzantine faults in several real-world applications. Naturally, over the past four decades, there has been a significant work on developing Byzantine-fault tolerant solutions to distributed problems like consensus [6] and read/write registers [7]. In this paper, we focus on Byzantine-tolerant read/write registers, as it has prevalent usage in industry, e.g., geo-replicated key-value storage (Cassandra [8]), and distributed shared memory (Redis [9]).

A. Read/Write Register

The emulation of shared storage on top of a message passing system allows the system designers to develop applications designed for the simpler shared memory model. Typically such emulation algorithms provide fault-tolerant storage of shared data by replicating the data on to several data stores. The ABD algorithm [1] showed that it is possible to implement a shared atomic register on top of an n-process asynchronous message-passing system if just a majority of servers were not crash-faulty.

This paper set forth a wave of research papers on fault-tolerant read/write registers. Although the theory literature has a huge body of work implementing atomic registers, in practice atomicity can be quite expensive and unnecessary. For example, popular geo-replicated storages such as Amazon Dynamo and Cassandra all adopt a weaker notion of consistency. Bailis et al. [10] quantitatively demonstrate how such eventually consistent systems offer significant latency benefits over full quorums.

Consistency conditions like safety [11] and regularity are weaker than atomicity and can be cheaper to implement with improved latencies of operations, especially read operations. In many applications [12][1], the number of read operations are much larger than the number of writes. As a result it is desirable to have reads that are faster than writes.

In a *semi-fast* register implementation the read (or write) operation is fast and the write (or read) operation is slow. Since it is impossible to implement a semi-fast multi-writer/multi-reader (MWMR) atomic register even with just crash failures [13], we choose to sacrifice atomicity to achieve a semi-fast safe

Authors ordered alphabetically.

[1]This paper states that across all Facebook products, read requests form around 99.8% of of all operations while write requests form only around 0.2%

register with fast reads. In our emulation, our read only takes one round of client-to-server communication, achieving *optimal round complexity*.

B. To RB or Not to RB

It is a common practice in the literature of shared read/write registers (e.g., [14], [15]) to assume the existence of a reliable broadcast primitive [2] or communication layer that ensures the *eventual "all or none"* property. Closely related work in the literature is discussed in Section VI.

In this paper, we take a different route. We show that it is possible to implement a Byzantine-tolerant safe register *without* the existence of a reliable broadcast mechanism if we assume more redundancy, i.e., our model assumes f more servers than the results [15] that assume reliable broadcast where f is the maximum number of servers that can be Byzantine faulty.

Again, our design choice is motivated by the observation that reducing response time is one of the most important goals in many latency-sensitive applications. Plus, most storage systems can be built upon commodity machines, making the cost of adding f extra machines minimal. Finally, reliable broadcast is simple to argue with, but it comes with a cost. Such an "all-or-nothing" property is typically *not* guaranteed by any networking protocol. Reliable broadcast implementation on top of reliable point-to-point channel typically requires 1.5 rounds of delay, e.g., [2]. If an emulation algorithm uses reliable broadcast for every message exchanged, the accumulated latency will eventually become blow up by a factor of 1.5.

C. Erasure Coding

To further reduce latency, we also consider a practical approach – reducing communication bandwidth consumption. This will be particularly useful when network has limited bandwidth or the data is too large. We achieve this by using erasure coding-based approach. An added benefit of using erasure codes is the reduced storage space at each server.

In comparison with the traditional replication-based approach (i.e., all servers storing the same copy of data), consistent memory emulation algorithms that use erasure codes reduce both the storage and communication costs of the implementation. In an erasure coding-based algorithm, 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. Same analysis applies to bandwidth consumption as well.

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. The usage of MDS codes to emulate consistent and fault-tolerant shared storage is an ongoing research topic, including tolerating crash-failures [16], [17] and Byzantine failures [18], [14]. None of the prior solutions that we know of provides fast read or one-shot read.

D. Main Contributions

Motivated by the practical interests in reducing latency, we have the following positive results:

- An algorithm emulating Multi-Writer-Multi-Reader replicated-based safe register with one-shot read (Section III).
- An algorithm emulation Single-Writer-Multi-Reader [2] erasure coding-based safe register with one-shot read (Section IV).

We also briefly discuss two methods to extend our work to provide an even stronger guarantee, *Multi-Writer Regularity*. All our algorithms do *not* assume the existence of reliable broadcast.

Our replication-based algorithm requires $n \geq 4f+1$, whereas our erasure coding-based algorithm requires $n \geq 5f+1$. These numbers are worse than prior solutions that only require $n \geq 3f+1$, e.g., [15]. However, we show in Section V these two numbers are necessary. In other words, the propose solutions have optimal resilience and optimal round complexity of reads.

II. Models and definitions

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

A. System Model

We consider a distributed system consisting of *asynchronous* processes of three types: a set of *readers*, a set of *writers* and a set of n *servers*. The readers and writers are together referred to as clients. Asynchronous means processes progress at their own speed, which can vary with time and remains always unknown to the other processes. There is also no bound on the message delays.

Each process is associated with a unique identifier (ID), and we denote the sets of IDs of the readers, writers and servers as $\mathcal{R}$, $\mathcal{W}$ and $\mathcal{S}$, respectively. The set $\mathcal{R} \cup \mathcal{W} \cup \mathcal{S}$ forms a totally ordered set under some defined relation ($>$), for example lexicographical order if the IDs are alphanumeric strings.

The reader and writer processes initiate *read* and *write* operations, respectively, and communicate with the servers using messages. At all times, at most one operation can run on a client.

At most f servers can be *Byzantine* faulty, i.e., it can behave arbitrarily and deviate from the algorithm in any way. For example, a Byzantine server can send incorrect register values, incorrect timestamp values, no reply or multiple replies to a certain request, etc All clients may suffer crash failures; otherwise, they follow the protocol specification.

We assume that every client is connected to every sever by bidirectional *reliable* communication channels that neither loses, duplicates nor creates messages, but may reorder messages arbitrarily. The model allows the sender process to fail after

[2]Our implementation actually provides stronger guarantees than Single-Writer-Multi-Reader safety. It can tolerate multiple writers as long as writes are not concurrent.

placing the message in the communication channel; message-delivery depends only on whether the destination is non-faulty. 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. We do not make any assumptions regarding relative order of message delivery in the same channel. The communication channels connecting servers and clients provide message authentication using digital signatures [19]. This prevents Byzantine servers from spreading misinformation about the sender of a message.

B. Shared Memory Abstraction

Our system consists of a finite set of *shared variables* (or read/write objects). The object values v come from some set V; initially, v is set to a distinguished default value v_0 ($\in V$). Reader r requests a read operation on object x. Similarly, a write operation is requested by a writer w.

Each operation at a non-faulty client begins with an *invocation step* and terminates with a *response step*. An operation π is *incomplete* in an execution when the invocation step of π does not have the associated response step; otherwise, we say that π is *complete*. An *execution* is set of possibly infinite sequences (one for each process) of invocation and responses. In an execution, we say that an operation (read or write) π_1 *precedes* another operation π_2, if the response step for π_1 precedes the invocation step of π_2. Two operations are *concurrent* if neither operation precedes the other.

A response event e_r is said to *match* an invocation event e_i if (i) they are associated with the same client, and (ii) e_i precedes e_r at the invoking process. An execution is said to be *valid* if for every client process: (i) it is empty (client crashes before invoking any operation), or its first event is an invocation; (ii) an invocation can only be followed by a client crash event or a matching response event; (iii) every invocation event has at most one matching response event; and (iv) there are no more events after the client's crash event (if there is any). An operation is said to be *live* if it terminates within a finite amount of time.

C. Definitions

We define our consistency conditions over complete operations in an execution. We present the definition of the correctness conditions safety [11] and regularity [15], [20] for multi-writer multi-reader (MWMR) registers.

Definition 1. *A MWMR register is **safe** if it satisfies the following (i) a read r that is not concurrent with any write returns the value of some write w that precedes r, as long as no other write falls completely between w and r (ii) otherwise the value returned is within the register's allowed range of values.*

Definition 2. *A MWMR register is **regular** if it satisfies the safety property above and the linearization of any two reads agree on the ordering of all writes that began before both the reads complete.*

Note that both consistency conditions imply some form of "strong consistency" used widely in industry – no stale version of value will be read. The discussion is intentionally informal, as there is no standard definition of "strong consistency" used in industry. Here, we follow the RIAK discussion on strong consistency.[3]

Another important property is the fast read or one-shot read. For a read operation, a *round* (or round-trip delay) consists of a read request message from the client initiating the read operation to the server, and subsequently, the server sending at least one version of the object value in its response. No server-to-server is involved in the process.

Definition 3 (One-Shot Read)**.** *One-shot reads require that each read operation completes in one round of client-to-server communication.*

III. BSR Algorithm

In this section, we present the algorithm for Byzantine replication-based Safe Register (BSR). In our algorithm, we rely on the notion of *witness* at a reader client C_r. A server p is called a witness of value v if p sends v to C_r as a response to a read request from C_r.

A. Pseudocode

Below we describe the key elements of the emulation algorithm BSR before presenting the pseudocode in Figures 1, 2, and 3

a) State Variables: Every server s_i, for $i \in [n]$, contains a list L of pairs of tags and values corresponding to each written value stored. Initially L has only the initial value v_0 of the register.

b) Write Operation: A write operation initiated by some writer w consists of two phases *get-tag* and *put-data*. In the *get-tag* phase, w sends a message of type QUERY-TAG to all servers in $\mathcal{S}$ and waits for responses from $n - f$ servers from which it selects the $(f + 1)$-th highest tag t. Then the writer initiates the *put-data* phase where it creates a new tag as $(t.num + 1, w)$, and sends the message (PUT-DATA, (t_w, v)) to server s_i in $\mathcal{S}$ for any $i \in [n]$; v is the value the writer wants to write to the register. Once w receives acknowledgements from $n - f$ servers w completes the write.

c) Read operation: A read operation initiated by some reader r consists of only one phase, which we refer to as *get-data*. During this phase r sends out read request messages in the form of (QUERY-DATA) to servers in $\mathcal{S}$ and waits for responses from $n - f$ servers in $\mathcal{S}$. Reader r completes the read by returning value with the highest tag that was verified by at least $f + 1$ servers. otherwise it returns the most recent value it has previously heard of (which may be the initial value v_0).

d) Server responses: The servers responses correspond to the two phases for the writers and one phase for the reader.

[3] https://docs.riak.com/riak/kv/latest/learn/concepts/strong-consistency/index.html

Response to writers: When any server s_i receives a (QUERY-TAG) message from a writer w, a *get-tag-resp* occurs, during which s_i responds by sending the maximum of tag stored in its L to w. When a server s_i receives a message (PUT-DATA, (t_{in}, v_{in})), a *put-data-resp* occurs at s_i where if the incoming tag t_{in} is higher then the tags in L then s_i adds (t_{in}, v_{in}) to L, and sends an acknowledgement to w.

Response to readers: When s_i receives a (QUERY-DATA) from a reader r, a *get-data-resp* occurs where s_i sends the locally available (t_{max}, c_{max}) pair (corresponding to the highest tag t_{max} in L) to r.

Fig. 1 BSR for write operation $\mathbf{write}(v)_w$, for writer $w \in \mathcal{W}$.

1: *get-tag*:
2: Send QUERY-TAG to servers in $\mathcal{S}$
3: Wait for responses from $n - f$ servers in $\mathcal{S}$
4: Select the $(f + 1)$-th highest tag t
5: *put-data*:
6: Create new tag $t_w = (t.num + 1, w)$.
7: Send (PUT-DATA, (t_w, v)) to servers in $\mathcal{S}$
8: Wait for ACKs from $n - f$ servers in $\mathcal{S}$

Fig. 2 BSR for read operation, $read_r$, for reader $r \in \mathcal{R}$.

1: Initially $(t_{local}, v_{local}) = (0, \perp, v_0)$
2: *get-data*:
3: Send (QUERY-DATA, t_{req}) to servers in $\mathcal{S}$
4: Wait for responses from $n - f$ servers in $\mathcal{S}$
5: $\mathcal{P} \leftarrow$ the set of all pairs with $\geq f + 1$ witnesses
6: Select pair, say (t_r, v_r), with the highest tag among $\mathcal{P}$
7: **if** $(\mathcal{P} \neq \emptyset$ & $(t_r, v_r) > (t_{local}, v_{local}))$ **then**
8: $(t_{local}, v_{local}) = (t_r, v_r)$
9: Return value v_{local}

Fig. 3 BSR for any server $s \in \mathcal{S}$.

1: **State Variables:**
$L \subseteq \mathcal{T} \times \mathcal{V}$, initially $\{(t_0, \perp)\}$
2: *get-tag-resp* (QUERY-TAG) **from** $w \in \mathcal{W}$:
3: Send $\max\{t : (t, *) \in L\}$ to w
4: *put-data-resp* (PUT-DATA, (t_{in}, v_{in})) **received:**
5: **if** t_{in} is higher than the locally available tag **then**
6: $L \leftarrow L \cup \{(t_{in}, v_{in})\}$
7: Send ACK to writer w of tag t_{in}
8: *get-data-resp* (QUERY-DATA) **from** $r \in \mathcal{R}$:
9: Send the locally available (t, v) pair.

B. Correctness of BSR

Theorem 1. *BSR satisfies liveness if at any point of the time, $n - f$ servers are available, i.e., at most f servers crash.*

Proof. All operations (reads and writes) wait for only $n - f$ replies from the servers. Thus liveness is satisfied if there are $n - f$ correct servers in the system. □

Lemma 1. *For each pair of read r and write w, if w completes before r begins and r is not concurrent with any write, then the timestamp of $w \leq$ the timestamp of r.*

Proof. A client has to hear from at least $n - f$ servers for an operation to complete. The write operation w completes by contacting server set W_1, and, $|W_1| \geq n-f$. Let the timestamp of the write be m. Thus the timestamp at the servers in W_1 after w completes will be $\geq m$. Suppose the read operation r contacts the server set R_1 with $|R_1| \geq n - f$. Since we assume that $n \geq 4f + 1$, $|W_1 \cap R_1| \geq 2f + 1$, which means the timestamp $\geq m$ was written to at least $2f + 1$ servers that r reads from. Of these at most f are Byzantine faulty, and thus, the (-th highest tag will be $\geq m$. □

Lemma 2. *The write operations are totally ordered by timestamp, and the resulting order respects the real-time order.*

Proof. Every write operation has two phases: get-tag and put-data. The get-tag phase collects the tags of the most recent writes known by the servers and the put-data phase sends out the new value and timestamp to be updated by the servers. Note that servers update their local value only as a response to put-data. Every pair of writes w_1 and w_2, will satisfy one of the following conditions:

- **Case 1:** The writes are not concurrent.
  WLOG, let w_1 completes before w_2 starts and let the timestamp of w_1 be m. This means, at least $n - f$ servers have heard of w_1 and thus their timestamp is $\geq m$. When w_2 is invoked, at least $n - 2 \geq 2f + 1$ of these nodes reply to the get-tag query for w_2. Since at most f nodes can be Byzantine faulty, the $f + 1$th highest tag (by line number 4) id at least m. This tag value gets incremented by 1 in the next phase. Thus the tag value of w_1 is strictly smaller than that of w_2
- **Case 2:** The writes are concurrent.
  This means that the real time of w_1 and w_2 overlap with each other. In this case, there may be two scenarios:
  - One write hears about the other concurrent write in the get-tag phase. WLOG, let w_2 hears about w_1 in its *get-tag* phase[4]. This scenario is similar to Case 1 and the tag of w_2 will be larger than that of w_1.
  - The two writes are unaware of each other. In this case, the get-tag query returns the same tag value for both the write operations. Since a writer can have at most one write operation the "tie" is broken by the total order on the ids of the two writers.

□

Lemma 3. *For every read operation r, there exists w denoted $\rho(r)$ such that it is some write operation that began before r began, such that r returns the value written by w or the initial value v_0 and has the same timestamp as w or 0.*

Proof. Any value v returned by a reader R at the end of a read operation r must have $f + 1$ witnesses. If not R returns the last known value, which is v_0 initially. This ensures that at least one fault-free server S has sent v to R. Since S is

[4]This is possible if at least one server has heard of the put-data phase of w_1 and has sent out her ack but w_1 hasn't completed because some messages related to this write are still in transit

fault-free, v is written by some write w_i or is the initial value v_0. Moreover, by definition, when R receives the value v, the read operation r is not terminated yet. In other words, w must have begun before r terminates. Letting $\rho(r) = w$ completes the proof. □

A completed write is a write operation that executes line number 8 of Algorithm 1 and a completed read is read operation that executes line number 9 of Algorithm 2.

Theorem 2. *Every well-formed execution of BSR satisfies safety given $n \geq 4f + 1$.*

Proof. We show that, for every execution, there is a total order on each completed read r and completed writes that begin before r begins. We now show that the execution satisfies safety. Intuitively, the lemma holds because for each completed read r we can construct a total order on r and completed writes

It is easy to see why a natural construction based on timestamps leads to a legal order. There are three cases to consider:

- A write followed by a read: due to Lemma 1
- A write followed by a write: due to Lemma 2
- A read followed by a write: due to Lemma 3.

Consider any read r. We now show how to construct a total order τ_r on the set consisting of r and all write operations as follows:

- **Case I:** If r is not concurrent with any write:
  1) Order all the writes that began before r before any write that began after r.
  2) Order all the writes that began before r in timestamp order among themselves.
  3) Order all the writes that began after r in timestamp order among themselves.
  4) Order r immediately after $\rho(r)$ before the following write.
- **Case II:** If r is concurrent with some write:
  1) Order all writes based on timestamps
  2) Order r immediately after $\rho(r)$ before the following write.

From lemma 3, we know that $\rho(r)$ exists; hence, such a construction is always feasible. The total order is legal by construction because a read concurrent with a write returns some valid value ($\in V$) and a read not concurrent with any write returns the value of the most recent write. Using the construction above, the total order on the writes respects real-time order (from Lemma 2).

This completes the proof of Theorem 2. □

C. MWMR Regular Register from BSR algorithm

Regularity is a stronger consistency condition than safety as indicated by the following theorem.

Theorem 3. *The BSR algorithm does not satisfy regularity*

Proof. Consider a system with $n = 5$ and $f = 1$ as follows: 5 servers $\{S_1, \cdots, S_5\}$, 5 writers $\{W_1 \cdots W_5\}$ and 1 reader $\{R\}$. WLOG, suppose server S_5 is faulty. Writer W_1 performs a write of value v_1 which completes by contacting all the servers. After w_1 completes, writers W_2 to W_5 initiate one write each that write values v_2, v_3, v_4 and v_5 respectively. The *get-tag* phase (line number 4 of Algorithm 1) of each of these writes completes quickly. The PUT-DATA message (line number 7 of Algorithm 1) sent out for each of these writes reaches servers S_2, S_3, S_4 and S_5 quickly and these servers S_i update their local value to v_i. However the other messages sent out in the *put-data* phase are slow. Assume the R starts a read after each server S_i has updated its value to v_i. All messages related to this read are fast and servers $S_1, \cdots, S_5$ reply with their local values which are currently v_1, v_2, v_3, v_4 and v_5 respectively. As a result, the set $\mathcal{P}$ on line number 5 for the read is empty and the reader R returns the initial value v_0, violating regularity. □

In order to ensure regularity of operations we can modify the BSR algorithm in two different ways:

- We can send the history of writes to a reader. We change line number 9 of Algorithm 3 to send the entire history of writes (L) instead of sending just the locally available (t, v) pair.
- We make the reads slow. Each read has two phases as well. A *get-tag* phase and a *get-data* phase. The *get-tag* phase is similar to the one in the writer algorithm. But the *get-tag-resp* from the servers is different. Instead of the most recent tag value, the sever sends a history of all the tags back to the reader. The reader chooses the largest tag t verified by $\geq f + 1$ servers and in the *get-data* phase, asks the servers for the write corresponding to this tag t. A read is complete when the reader receives $f + 1$ matching replies corresponding to the *get-data* request.

These two different approaches to BSR takes into account the worst case scenario mentioned in Theorem 3 and guarantees regularity. The full algorithms and proofs will be presented in the future technical report.

IV. BCSR Algorithm

In this section, we present the algorithm for Byzantine Coded Safe Register (BCSR). We will first focus our discussion on SWMR safety, as it will be easy to show that there does *not* exist a one-shot erasure coding-based MWMR safe register.

A. MDS Erasure Code

In our algorithm, we rely on the MDS code that fix erroneous coded elements. Particularly, we will use $[n, k]$ MDS code for

$$k = n - f - 2e,$$

where f is the maximum number of erasures and e is the maximum number of erroneous coded elements used in decoding. The meaning of erroneous element will become clear later. In our emulation algorithm, e is bounded by $2f$. Therefore, $k = n - 5f$.

In BCSR, we use a linear $[n, k]$ MDS erasure code [21] over a finite field $\mathbb{F}_q$ to encode and store the value v among the n servers. An $[n, k]$ MDS erasure code has the property

that any k out of the n coded elements, computed by encoding v, can be used to recover (decode) the value v.

For encoding, v is divided into k elements $v_1, v_2, \ldots v_k$ with each element having size $\frac{1}{k}$ (assuming size of v is 1). The encoder takes the k elements as input and produces n coded elements $c_1, c_2, \ldots, c_n$ as output, i.e.,

$$[c_1, \ldots, c_n] = \Phi([v_1, \ldots, v_k]),$$

where Φ denotes the encoder. For ease of notation, we simply write $\Phi(v)$ to mean $[c_1, \ldots, c_n]$.

The vector $[c_1, \ldots, c_n]$ is referred to as the codeword corresponding to the value v. Each coded element c_i also has size $\frac{1}{k}$. In our scheme we store one coded element per server. We use Φ_i to denote the projection of Φ on to the i^{th} output component, i.e., $c_i = \Phi_i(v)$. Without loss of generality, we associate the coded element c_i with server i, $1 \leq i \leq n$.

Recall that the decoder function Φ^{-1} can correctly decode the original value if the input contains $n - f$ coded elements and among these used elements, up to e may be erroneous. In our scenarios, a coded element may be erroneous if the server is Byzantine faulty or the server is slow (hence, returning stale data, i.e., an earlier version of the value).

B. BCSR Pseudocode

Below we describe the key elements of the emulation algorithm BCSR before presenting the pseudocode in Figures 4, 5, and 6:

a) State Variables: Every server s_i, for $i \in [n]$, contains a list L of pairs of tags and coded elements corresponding to the i-th coded elements for values stored. Initially L has a set of initial values, corresponding to the default or initial value stored in the object.

b) Write Operation: A write operation is very similar to the one in Fig 1. The only difference is that the writers send out a coded element (PUT-DATA, (t_w, c_i)) instead of the complete value of the simulated register.

c) read operation: The query part of the read operation is similar to the read operation in BSR. After the (QUERY-DATA) message is sent out, r waits for responses from $n - f$ servers in $\mathcal{S}$, to receive coded element c_i from server $i \in \mathcal{S}$. Reader r completes the read by returning the decoded value, v_r, $\leftarrow \Phi^{-1}(c_i)$, if possible, otherwise, the default value v_0 is returned.

d) Server responses: The *get-tag-resp* at any server is the same as in BSR. The *put-data-resp* is similar to BSR except that the server stores the timestamp, code pair instead of the entire value of the register. The *get-data-resp* occurs when s_i receives a (QUERY-DATA) from a reader r, where s_i sends the locally available (t_{max}, c_{max}) pair corresponding the highest tag t_{max} in L.

Fig. 4 BCSR: write operation $\mathbf{write}(v)_w$ for writer $w \in \mathcal{W}$.

1: *get-tag*:
2: Send QUERY-TAG to servers in $\mathcal{S}$
3: Wait for responses from $n - f$ servers in $\mathcal{S}$
4: Select the $(f + 1)$-th highest tag t
5: *put-data*:
6: Create new tag $t_w = (t.num + 1, w)$.
7: Send (PUT-DATA, (t_w, c_i)) to s_i for $i \in [n]$, where $c_i = \Phi_i(v)$
8: Wait for ACKs from $n - f$ servers in $\mathcal{S}$

Fig. 5 BCSR for read operation, $read_r$, $r \in \mathcal{R}$.

1: *get-data*:
2: Send (QUERY-DATA) to servers in $\mathcal{S}$
3: Wait for responses from $n - f$ servers in $\mathcal{S}$, say receiving coded element c_i from server $i \in \mathcal{S}$
4: Return the decoded value $v_r \leftarrow \Phi^{-1}(c_i)$, if possible; O.W., return v_0

C. Correctness of BCSR

We now prove that BCSR is correct when $n \geq 5f + 1$. The following theorem can be proved similarly as in the previous section.

Theorem 4. *The BSCR algorithm is live if any reader may crash fail, and any writer and up to f servers can exhibit Byzantine failure.*

Lemma 4. *Every well-formed execution of BCSR is safe given $n \geq 5f + 1$.*

Proof. Suppose a write $W(v)$ that is *not* concurrent with another write completes, and a subsequent non-overlapping read R will return the the value v. We only need to consider this scenario due to the definition of MWMR safeness condition.

First, by the time $W(v)$ completes, at least $n-2f$ fault-free servers i have the pair (t, c_i) for some t in their local state variable.

Second, a reader then will receive $\geq n - 3f$ correct pair (t, c_i), i.e., the coded element is *not* corrupted by a faulty server, and corresponds to the most recent value written by $W(V)$. Therefore, among all the coded elements received by the reader, $\leq (n - f) - (n - 3f) = 2f$ are erroneous. Hence, by assumption of the decoder function, the reader will be able to decode the correct value v. □

V. Lower Bound for Safety of One-Shot reads

In the proofs below, we assume that there are n servers of which $\leq f$ are Byzantine. We first show that there need to be enough "witnesses" to safely return a value.

Lemma 5. *Safety is violated if value returned by a read operation is not witnessed by at least $f + 1$ servers*

Proof. Suppose not by way of contradiction. Since a read requires at most f witness servers, the f Byzantine servers may choose to send a value $v_b \notin V$. If a read returns v_b, validity (and thus safety) of the register is violated. □

The following lemma directly follow form our model definition and liveness property.

Fig. 6 BCSR for any server $s \in \mathcal{S}$.

1: **State Variables:**
$L \subseteq \mathcal{T} \times \mathcal{V}$, initially $\{(t_0, c_0^s)\}$
2: *get-tag-resp* (QUERY-TAG) **from** $w \in \mathcal{W}$**:**
3: Send $\max\{t : (t, *) \in L\}$ to w
4: *put-data-resp* (PUT-DATA, (t_{in}, c_{in})) **received:**
5: **if** t_{in} is higher than the locally available tags **then**
6: $L \leftarrow L \cup \{(t_{in}, c_{in})\}$
7: Send ACK to writer w of tag t_{in}
8: *get-data-resp* (QUERY-DATA) **from** $r \in \mathcal{R}$**:**
9: Send (t_{max}, c_{max}) pair, where $t_{max} \equiv \max\{t : (t, *) \in L\}$

Lemma 6. *Liveness cannot be guaranteed if an operation needs to wait for more than $n - f$ replies from servers.*

For Lemma 7 and Theorem 5, we consider a system and a replication-based safe register algorithm $\mathbb{A}$, with four servers s_0, s_1, s_2 and s_3. The system contains two writers w_1 and w_2, and one reader r. Suppose $n = 4$ and $f = 1$. Moreover, assume that server s_0 is Byzantine faulty. Let v_0 be the initial default value stored in the replicated register.

Lemma 7. *Safety cannot be guaranteed if write operations do not communicate with at least $3f$ servers.*

Proof. Assume by contradiction $\mathbb{A}$ emulates a safe register where writes do not communicate with at least 3 servers. Suppose there is only one write operation $W(v_1)$, writing value v_1 initiated by writer w_1. Let the writer communicate with just two servers: s_0 and s_1.

Now, consider a read operation R from reader r that starts after W completes. Lemma 6 states that to guarantee liveness, any operation can wait for at most $n - f = 3$ servers. Suppose R receives replies from servers s_0, s_2 and s_3. Servers s_2 and s_3, have not heard of the write W and hence they return the initial value of the register v_0. Additionally, since s_0 is Byzantine, it falsely returns the value v_0 as a response to R. Therefore, R completes by returning the initial value v_0. This violates safety leading to a contradiction. Thus, any write must communicate with at least 3 servers. By a simple simulation argument, the lemma is implied. □

Theorem 5. *It is impossible to emulate a replicated safe register and guarantee liveness of reads and writes where reads are one-shot, with n servers of which at most $f \geq \frac{n}{4}$ may be Byzantine faulty.*

Proof. Assume by contradiction, $\mathbb{A}$ emulates a safe register with 4 servers of which s_0 is Byzantine faulty. Consider the following scenario: Two write operations $W_1(v_1)$ and $W_2(v_2)$ are invoked on the register by writers w_1 and w_2, respectively.

Let $W_2(v_2)$ be invoked after $W_1(v_1)$ completes. A read operation R is invoked by a reader r other than w_1 and w_2 after $W_2(v_2)$ completes. By the correctness of $\mathbb{A}$, R completes.

By Lemmas 6 and 7, each write waits for exactly 3 responses. Suppose write operation W_1 communicates with servers s_0, s_1, s_2 and write operation $W_2(v_2)$ communicates with servers s_0, s_2, s_3. Servers s_0, s_1 and s_2 respond to the invocation of R ny returning the whole history. Since s_0 is Byzantine it can return any arbitrary message to reader r in its response. Suppose s_0 returns v_1 instead of v_2. Thus, R receives the values v_1 from s_0 and s_1, and v_1, v_2 from s_2.

Lemma 5 states that R cannot return v_2, as it is witnessed by only $f = 1$ server. Thus R returns v_1, violating safety. Again, by the simulation argument, the theorem follows. □

Remark 1. *Our proof works because we assume that the algorithms or the networking layer do not use nor provide the **reliable broadcast** abstraction. In many prior works [15], [20], [22], a reader can wait until all the fault-free servers receive enough messages regarding recent write operations. In our case, it is possible that two fault-free servers would never receive the same set of messages, especially if the sender crashes.*

Remark 2. *Our proof can be extended to the case with multiple client-to-server communication. The only thing we exclude is the server-to-server communication. Since in that case, the reliable broadcast primitive can be implemented on top of reliable channel with $n \geq 3f + 1$ servers [2].*

Erasure-Coding-based Register

We now focus on the EC-based register emulation. Suppose the emulation uses $[n, k]$ code, where $k = n - f - 2e$. The proof is similar to the one above. The difference is that now the quorum intersection between readers and writers need to be larger due to the usage of erasure code. We include the full proof below for completeness.

First, is should be easy to observe the following lemma if we have only $5f$ servers.

Lemma 8. *Safety cannot be guaranteed if write operations do not communicate with at least $4f$ servers.*

Then, we are ready to present the key proof.

Theorem 6. *It is impossible to emulate a safe register and guarantee liveness of reads and writes where reads are one-shot, with $n \leq 5f$.*

Proof. Assume by contradiction, $\mathbb{A}$ emulates a safe register with 5 servers of which s_0 is Byzantine faulty with $f = 1$. Consider the following scenario: Two write operations $W_1(v_1)$ and $W_2(v_2)$ are invoked on the register by writer w_1.

Let $W_2(v_2)$ be invoked after $W_1(v_1)$ completes. A read operation R is invoked by a reader r other than w_1 after $W_2(v_2)$ completes. By the correctness of $\mathbb{A}$, R completes.

By Lemmas 6 and 8, each write waits for exactly 4 responses. Suppose write operation W_1 communicates with servers s_0, s_1, s_2, s_3 and write operation $W_2(v_2)$ communicates with servers s_0, s_2, s_3, s_4.

Servers s_0, s_1, s_2, s_3 respond to the invocation of R by returning the whole history of values. Since s_0 is Byzantine, it can return any arbitrary message to reader r in its response. Suppose s_0 returns the coded element corresponding to v_1 instead of v_2. Thus, R receives the coded element corresponding to v_1 from s_0, s_1, s_2, s_3, and coded element to v_2 from s_2 and s_3. Recall that to use the MDS code, we need to have $n - f$

coded elements with up to e erroneous coded elements. Hence, the only choice is to return v_1. If the algorithm chose to return v_1, then there are one missing coded element (which is stored at s_4), and two erroneous (in this case, stale) coded elements (which are returned by s_0, s_1). Consequently, the decoding function does not work.

Finally, by the simulation argument, the theorem follows. □

VI. Related Work

Reliable computing must handle malfunctioning components that give potentially conflicting information to different parts of the system. The term "Byzantine" nodes was introduced by Pease et al. to describe such malfunctioning components. In this paper [6], the problem of reaching agreement or consensus was studied in the presence of Byzantine faults. Lamport et al. [22] showed that using unforgeable or digitally signed [19] messages, this problem is solvable if and only if more than two-thirds of the nodes not Byzantine faulty. Martin et al. [23] present protocols for asynchronous Byzantine Quorum Systems (BQS) built on top of reliable channels by using read and write quorums of different sizes and show how to remove dependency on reliable networking. The paper by Marko Vukolic [24] provides a detailed survey of the evolution of quorum systems for distributed storage and consensus.

Shared memory implementations on top of message-passing has been of great interest to the distributed systems community as it makes programming (for the shared memory model) simpler. The register is one of the most basic objects of computer science as most shared data structures are built using using registers as a building block at some point in their construction. Lamport [11] defines three classes of shared read/write registers called *safe, regular,* and *atomic* which vary in their consistency conditions in the presence of concurrency.

The most famous atomic register implementation [1] on top of an n-process asynchronous message-passing system is termed the ABD algorithm after its authors Attiya, Bar-Noy and Dolev. They showed that $f < n/2$ (where f is the maximal number of processes that may crash and n is the number of servers in the system) is a necessary and sufficient requirement to build an atomic register on top of a crash-prone asynchronous message passing system.

Dutta et al. [25] showed how to obtain fast crash-tolerant implementations of a single-writer/multi-reader atomic registers in which both read and write operations complete in one communication round-trip, under the constraint that the number of readers is less than $\frac{n}{f} - 2$. Georgiou et al. [13] prove that no semi-fast implementation exists for the multi-reader/multi-writer atomic register even with crash failures. Mostefaoui et al. [26] devise a new time-efficient asynchronous algorithm for atomic registers that reduces latency in many cases. Several papers [27], [28] have explored mechanisms to implement wait-free multivalued registers from a collection of binary registers.

Taubenfeld [29] considers generalizations of Lamport's notions, called k-safe, k-regular and k-atomic and provides constructions for implementing 1-atomic registers (the strongest type) in terms of k-safe registers (the weakest type). These constructions allow solving of classical synchronization problems, such as mutual exclusion [30] and l-exclusion, using SWMR k-safe bits, for any $k \geq 1$. Shao et al. [20] modify Lamport's definitions of a regular register to give four new consistency conditions, implementing some form of regularity for a MWMR register. They also provide algorithms to emulate multi-writer regularity for each of these newly defined consistency conditions. In [15], Kanjani et al. present a simple replication-based register emulation that tolerates Byzantine faults. They assume the existence of reliable broadcast, and they rely on a technique called *relay*[5].

The importance of the register has has motivated many directions for research on this simple data structure. Pozzo et al. [31] emulate a regular read/write register in a synchronous distributed system with anonymous clients. In their paper servers may be *rational* malicious Byzantine processes. The authors model this problem as a Bayesian game and design a protocol implementing the regular register that forces the rational malicious server to behave correctly. In addition to building registers from scratch, much effort has been put into building stronger wait-free registers from weaker ones. Johnen and Higham [32] present a wait-free implementation of a single-writer/single-reader regular register using single-writer/single-reader safe registers. Jayanti et al. [33] implement a single-writer/single-reader atomic register from two regular registers.

Significant gains over replication-based strategies can still be achieved while using erasure codes that can tolerate crash-failures [16], [17] and Byzantine failures [18], [14]. Here, we focus on Byzantine-tolerant work. In [18], the erasure code is used, along with cryptographic hashes to provide strong consistency guarantees. [14] provides an erasure coded implementation of strong consistency with Byzantine failure. These works do not provide one-shot read, and assumed a reliable broadcast primitive.

VII. Conclusion

This paper identifies optimal approaches to emulate MWMR replication-based safe register and SWMR erasure coding-based safe register with one-shot read. The first algorithm requires $\geq 4f + 1$ servers, whereas the second one requires $\geq 5f + 1$ servers. We prove that these numbers are optimal.

References

[1] H. Attiya, A. Bar-Noy, and D. Dolev, "Sharing memory robustly in message-passing systems," J. ACM, vol. 42, no. 1, pp. 124–142, Jan. 1995. [Online]. Available: http://doi.acm.org/10.1145/200836.200869

[2] G. Bracha, "Asynchronous byzantine agreement protocols," Inf. Comput., vol. 75, no. 2, pp. 130–143, 1987. [Online]. Available: https://doi.org/10.1016/0890-5401(87)90054-X

[3] X. Défago, A. Schiper, and P. Urbán, "Total order broadcast and multicast algorithms: Taxonomy and survey," ACM Comput. Surv., vol. 36, no. 4, pp. 372–421, 2004. [Online]. Available: https://doi.org/10.1145/1041680.1041682

[4] T. D. Chandra and S. Toueg, "Unreliable failure detectors for reliable distributed systems," J. ACM, vol. 43, no. 2, pp. 225–267, 1996. [Online]. Available: https://doi.org/10.1145/226643.226647

[5] In this paper, we show that by adding f more nodes into the system, we are able to make the system more practical by eliminating the usage of reliable broadcast and relay.

[5] K. Driscoll, B. Hall, M. Paulitsch, P. Zumsteg, and H. Sivencrona, "The real byzantine generals," in The 23rd Digital Avionics Systems Conference (IEEE Cat. No.04CH37576). IEEE. [Online]. Available: https://doi.org/10.1109/dasc.2004.1390734

[6] M. C. Pease, R. E. Shostak, and L. Lamport, "Reaching agreement in the presence of faults," J. ACM, vol. 27, no. 2, pp. 228–234, 1980. [Online]. Available: http://doi.acm.org/10.1145/322186.322188

[7] M. Raynal, Atomic Read/Write Registers in the Presence of Byzantine Processes. Cham: Springer International Publishing, 2018, pp. 155–170. [Online]. Available: https://doi.org/10.1007/978-3-319-94141-7_9

[8] http://cassandra.apache.org/, "Cassandra."

[9] https://redis.io, "Redis."

[10] P. Bailis, S. Venkataraman, M. J. Franklin, J. M. Hellerstein, and I. Stoica, "Probabilistically bounded staleness for practical partial quorums," VLDB Endowment, vol. 5, no. 8, pp. 776–787, 2012.

[11] L. Lamport, "On interprocess communication. part I: basic formalism," Distributed Computing, vol. 1, no. 2, pp. 77–85, 1986. [Online]. Available: https://doi.org/10.1007/BF01786227

[12] N. Bronson, Z. Amsden, G. Cabrera, P. Chakka, P. Dimov, H. Ding, J. Ferris, A. Giardullo, S. Kulkarni, H. C. Li, M. Marchukov, D. Petrov, L. Puzar, Y. J. Song, and V. Venkataramani, "TAO: facebook's distributed data store for the social graph," in 2013 USENIX Annual Technical Conference, San Jose, CA, USA, June 26-28, 2013, 2013, pp. 49–60. [Online]. Available: https://www.usenix.org/conference/atc13/technical-sessions/presentation/bronson

[13] C. Georgiou, N. C. Nicolaou, and A. A. Shvartsman, "Fault-tolerant semifast implementations of atomic read/write registers," J. Parallel Distrib. Comput., vol. 69, no. 1, pp. 62–79, 2009. [Online]. Available: https://doi.org/10.1016/j.jpdc.2008.05.004

[14] J. Hendricks, G. R. Ganger, and M. K. Reiter, "Low-overhead byzantine fault-tolerant storage," in ACM SIGOPS Operating Systems Review, vol. 41, no. 6, 2007, pp. 73–86.

[15] K. Kanjani, H. Lee, W. L. Maguffee, and J. L. Welch, "A simple byzantine-fault-tolerant algorithm for a multi-writer regular register," IJPEDS, vol. 25, no. 5, pp. 423–435, 2010.

[16] V. R. Cadambe, N. A. Lynch, M. Médard, and P. M. Musial, "A coded shared atomic memory algorithm for message passing architectures," in Proceedings of 13th IEEE International Symposium on Network Computing and Applications (NCA), 2014, pp. 253–260.

[17] P. Dutta, R. Guerraoui, and R. R. Levy, "Optimistic erasure-coded distributed storage," in Proceedings of the 22nd international symposium on Distributed Computing (DISC), Berlin, Heidelberg, 2008, pp. 182–196.

[18] C. Cachin and S. Tessaro, "Optimal resilience for erasure-coded byzantine distributed storage," in Proceedings of International Conference on Dependable Systems and Networks (DSN), 2006, pp. 115–124.

[19] M. J. Ganley, "Digital signatures and their uses," Computers & Security, vol. 13, no. 5, pp. 385–391, 1994. [Online]. Available: https://doi.org/10.1016/0167-4048(94)90031-0

[20] C. Shao, J. L. Welch, E. Pierce, and H. Lee, "Multiwriter consistency conditions for shared memory registers," SIAM J. Comput., vol. 40, no. 1, pp. 28–62, 2011.

[21] W. C. Huffman and V. Pless, Fundamentals of error-correcting codes. Cambridge university press, 2003.

[22] L. Lamport, R. E. Shostak, and M. C. Pease, "The byzantine generals problem," ACM Trans. Program. Lang. Syst., vol. 4, no. 3, pp. 382–401, 1982. [Online]. Available: http://doi.acm.org/10.1145/357172.357176

[23] J. Martin, L. Alvisi, and M. Dahlin, "Small byzantine quorum systems," in 2002 International Conference on Dependable Systems and Networks (DSN 2002), 23-26 June 2002, Bethesda, MD, USA, Proceedings, 2002, pp. 374–388. [Online]. Available: https://doi.org/10.1109/DSN.2002.1028922

[24] M. Vukolic, Quorum Systems: With Applications to Storage and Consensus, ser. Synthesis Lectures on Distributed Computing Theory. Morgan & Claypool Publishers, 2012. [Online]. Available: https://doi.org/10.2200/S00402ED1V01Y201202DCT009

[25] P. Dutta, R. Guerraoui, R. R. Levy, and A. Chakraborty, "How fast can a distributed atomic read be?" in Proceedings of the Twenty-Third Annual ACM Symposium on Principles of Distributed Computing, PODC 2004, St. John's, Newfoundland, Canada, July 25-28, 2004, 2004, pp. 236–245. [Online]. Available: https://doi.org/10.1145/1011767.1011802

[26] A. Mostéfaoui, M. Raynal, and M. Roy, "Time-efficient read/write register in crash-prone asynchronous message-passing systems," Computing, vol. 101, no. 1, pp. 3–17, 2019. [Online]. Available: https://doi.org/10.1007/s00607-018-0615-8

[27] M. J. Kosa, "Wait-free lazy-writer registers with eager readers," in Proceedings of the 36th Annual ACM Southeast Regional Conference, April 1-3, 1998, Marietta, GA, USA, 1998, pp. 274–276. [Online]. Available: https://doi.org/10.1145/275295.275369

[28] E. Ruppert, "Brief announcement: Readers of wait-free unbounded registers must write," in Proceedings of the ACM Symposium on Principles of Distributed Computing, PODC 2017, Washington, DC, USA, July 25-27, 2017, 2017, pp. 93–94. [Online]. Available: https://doi.org/10.1145/3087801.3087859

[29] G. Taubenfeld, "Weak read/write registers," in Distributed Computing and Networking, 14th International Conference, ICDCN 2013, Mumbai, India, January 3-6, 2013. Proceedings, 2013, pp. 423–427. [Online]. Available: https://doi.org/10.1007/978-3-642-35668-1_29

[30] L. Lamport, "A new solution of dijkstra's concurrent programming problem," Commun. ACM, vol. 17, no. 8, pp. 453–455, 1974. [Online]. Available: https://doi.org/10.1145/361082.361093

[31] A. D. Pozzo, S. Bonomi, R. Lazzeretti, and R. Baldoni, "Building regular registers with rational malicious servers and anonymous clients," in Cyber Security Cryptography and Machine Learning - First International Conference, CSCML 2017, Beer-Sheva, Israel, June 29-30, 2017, Proceedings, 2017, pp. 50–67. [Online]. Available: https://doi.org/10.1007/978-3-319-60080-2_4

[32] C. Johnen and L. Higham, "Fault-tolerant implementations of regular registers by safe registers with applications to networks," in Distributed Computing and Networking, 10th International Conference, ICDCN 2009, Hyderabad, India, January 3-6, 2009. Proceedings, 2009, pp. 337–348. [Online]. Available: https://doi.org/10.1007/978-3-540-92295-7_41

[33] P. Jayanti, J. E. Burns, and G. L. Peterson, "Almost optimal single reader, single writer atomic register," J. Parallel Distrib. Comput., vol. 60, no. 2, pp. 150–168, 2000. [Online]. Available: https://doi.org/10.1006/jpdc.1998.1505