RADON: Repairable Atomic Data Object in Networks*

Kishori M. Konwar[1], N. Prakash[2], Nancy A. Lynch[3], and Muriel Médard[4]

1 **Department of EECS, MIT, Cambridge, MA, USA**
`kishori@csail.mit.edu`
2 **Department of EECS, MIT, Cambridge, MA USA**
`prakashn@mit.edu`
3 **Department of EECS, MIT, Cambridge, MA, USA**
`lynch@csail.mit.edu`
4 **Department of EECS, MIT, Cambridge, MA, USA**
`medard@mit.edu`

Abstract

Erasure codes offer an efficient way to decrease storage and communication costs while implementing atomic memory service in asynchronous distributed storage systems. In this paper, we provide erasure-code-based algorithms having the additional ability to perform background repair of crashed nodes. A repair operation of a node in the crashed state is triggered externally, and is carried out by the concerned node via message exchanges with other active nodes in the system. Upon completion of repair, the node re-enters active state, and resumes participation in ongoing and future read, write, and repair operations. To guarantee liveness and atomicity simultaneously, existing works assume either the presence of nodes with stable storage, or presence of nodes that never crash during the execution. We demand neither of these; instead we consider a natural, yet practical network stability condition $N1$ that only restricts the number of nodes in the crashed/repair state during broadcast of any message.

We present an erasure-code based algorithm $RADON_C$ that is always live, and guarantees atomicity as long as condition $N1$ holds. In situations when the number of concurrent writes is limited, $RADON_C$ has significantly improved storage and communication cost over a replication-based algorithm $RADON_R$, which also works under $N1$. We further show how a slightly stronger network stability condition $N2$ can be used to construct algorithms that never violate atomicity. The guarantee of atomicity comes at the expense of having an additional phase during the read and write operations.

1998 ACM Subject Classification C.2.4 Distributed Systems

Keywords and phrases Atomicity, repair, fault-tolerance, storage cost, erasure codes

Digital Object Identifier 10.4230/LIPIcs.OPODIS.2016.28

1 Introduction

We consider the problem of designing algorithms for distributed storage systems (DSSs) that offer consistent access to stored data. Large scale DSSs are widely used by several industries, and also widely studied by academia for a variety of applications ranging from e-commerce

* The work is supported in part by AFOSR under grants FA9550-14-1-043, FA9550-14-1-0403, and in part by NSF under awards CCF-1217506, CCF-0939370.

to sequencing genomic-data. The most desirable form of consistency is atomicity, which in simple terms, gives the users of the data service the impression that the various concurrent read and write operations take place sequentially. Implementations of atomicity on an asynchronous system under message passing framework, in the presence of failures, is often challenging. Traditional implementations[4], [14] use replication of data as the mechanism of fault-tolerance; however they suffer from the problem of having high storage cost, and communication costs for read and write operations.

Erasure codes provide an efficient way to decrease storage and communication cost in atomicity implementations. An $[n, k]$ erasure code splits the value v, say 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. 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. While it is known that usage of erasure codes in asynchronous decentralized storage systems do not offer all the advantages as in synchronous centralized systems [29], erasure code based algorithms like in [1], [13], [8], or [19] for implementing consistent memory service offer significant storage and communication cost savings over replication based algorithms, in many regimes of operation. For instance CASGC [8] improves the costs under the scenario when the number of writes concurrent with a read is known to be limited, whereas SODA [19] trades-off write cost in order to optimize storage cost, which is meaningful in systems with infrequent writes. Both CASGC and SODA are based on MDS codes.

In this work, we consider the additional important issue of repairing crashed nodes without disrupting the storage service. Failure of storage nodes is a norm rather than an exception in large scale DSSs today, primarily because of the usage of commodity hardware for affordability and scalability reasons. Replication based algorithms in [4], [14] and erasure-code based algorithms in [1], [8], or [19] do not consider repair of crashed nodes; instead assume that a crashed node remains so for the rest of the execution. Algorithms in [13], [15] consider background repair of crashed nodes; however they assume either the presence of nodes having stable storage, whose content is unaffected by crashes, or presence of a subset of nodes that never crash during the entire execution. We relax both these assumptions in this work. In our model, any one of the storage nodes can crash; further, we assume that a crashed node loses all its data, both volatile as well as stable storage. A repair operation of a node in the crashed state is triggered externally, and is carried out by the concerned node via message exchanges with other active nodes in the system. Upon completion of repair, the node (with the same id) re-enters active state, and resumes participation in ongoing and future read, write, and repair operations.

It is natural to expect a restriction on the number of crash and repair operations in relation to the read and write operations; the authors of [15] show an impossibility result in this direction, for guaranteeing liveness and atomicity, simultaneously. We formulate network stability conditions $N1$ and $N2$, which can be used to limit the number of crash and repairs operations overlapping with a client operation. These conditions are algorithm independent, and most likely to be satisfied in any practical storage network. At a high level, the condition $N1$ restricts the set of servers that can be in the crashed or repair state any time a process (client or server) *pings* all the n servers with corresponding messages. Condition $N2$ is slightly stronger than $N1$, and restricts the set of servers that can be in the crashed or repair state if the process wants to *ping-pong* a fraction of the servers. In a ping-pong, it is expected that the servers which receive a message also respond back to the sender of the message.

1.1 Summary of Our Contributions

We first present an impossibility result for an asynchronous DSS allowing background repair of crashed nodes, where there is no restriction on the number of crash and repair operations that occur during a client operation. We show that it is impossible to simultaneously achieve liveness and atomicity in such a system, even if all the crash and repair operations occur sequentially during the execution (i.e., at most one node remains in the crash or repair state at any point during the execution).

We then consider the problem of erasure-code based algorithm design under the network stability condition $N1$. We present the algorithm in two stages. First we present a replication-based algorithm $RADON_R$, which performs background-repair, and guarantees atomicity and liveness of operations under $N1$, if more than $3/4^{\text{th}}$ of all servers remain active during any ping operation. The write and read phases are almost identical to those of the ABD algorithm [4], except that during a write we expect responses from more than $3/4^{\text{th}}$ of all the servers, while in ABD responses are expected only from a majority of servers. A repair operation in $RADON_R$ is simply a read operation initiated by the concerned server. Thus the algorithm itself is simple; however, the proof of atomicity gets complicated because of the fact that a repair operation can potentially restore the contents of a node to a version that is older than what was present before the crash. We show how the network stability condition can be used to prove atomicity, and this proof is the key takeaway from $RADON_R$ towards constructing the erasure-code based algorithm.

Our erasure-code based algorithm $RADON_C$ uses $[n, k]$ MDS codes, and is a natural adaptation of $RADON_R$ for the usage of codes. A key challenge while using erasure codes is ensuring liveness of read operations, in the presence of concurrent write operations. Various techniques are known in literature to handle this challenge; for instance, [13] assumes synchronous write phases, [8] limits the number of writes concurrent with a read, while [19] uses an $O(n^2)$ write protocol to guarantee liveness of reads. In this work, like in [8], we make the assumption that the number of write operations concurrent with any read operation is limited by a parameter δ, which is known a priori. However, the usage of the concurrency bound differs from that of the CASGC algorithm in [8]; for instance, CASGC has three rounds for write operations, while $RADON_C$ uses only two rounds. In $RADON_C$, each server maintains a list of up to $\delta + 1$ coded elements, corresponding to the latest $\delta + 1$ versions received as a result of the various write operations. In comparison with $RADON_R$ where a writer expects responses from more than $3/4^{\text{th}}$ of all servers, a write operation in $RADON_C$ expects responses from more than $\frac{3n+k}{4}$ servers. During a read operation, the client reads the lists from more than $\frac{n+k}{2}$ nodes before decoding the value v. Like in $RADON_R$, a repair operation in $RADON_C$ is essentially a read operation by the concerned node; however this time the concerned node creates a list (instead of just one version) by decoding as many possibles versions that it can from the $\lceil \frac{n+k}{2} \rceil$ responses. Liveness and atomicity of operations are proved under network stability condition $N1$, if more than $\frac{3n+k}{4}$ servers remain active during any ping operation. $RADON_C$ has substantially improved storage and communication costs than $RADON_R$, when the concurrency bound δ is limited; see Table 1 for a comparison.

In both $RADON_R$ and $RADON_C$, violation of the network stability condition $N1$ can result in executions that are not atomic, which might not be preferable in certain applications. The choice of consistency over liveness, or vice versa, is the subject matter of a wide range of discussions and perspectives among system designers and software engineers. For example, BigTable, a DSS by Google, prefers safety over liveness [9], whereas, Amazon's Dynamo does not compromise liveness but settles for *eventual consistency* [10]. Our third algorithm

Table 1 Performance comparison of $RADON_R$, $RADON_C$ and $RADON_R^{(S)}$, where n is the number of servers, and δ is the maximum number of writes concurrent with a read or a repair operation. See Section 7 for a justification of the costs.

Algorithm	Write Cost	Read Cost	Storage Cost	Safe under	Live under
$RADON_R$	n	$2n$	n	$N1$	$N1$
$RADON_C$	$\frac{n}{k}$	$(\delta+2)\frac{n}{k}$	$(\delta+1)\frac{n}{k}$	$N1$	$N1$
$RADON_R^{(S)}$	n	$2n$	n	*always*	$N2$

$RADON_R^{(S)}$, which is replication-based, is designed to guarantee atomicity during every execution. Liveness is guaranteed under the slightly more stringent condition of $N2$, with more than $3/4^{\text{th}}$ of all servers remaining active during any ping-pong operation. The guarantee of atomicity of every execution also needs extra phases for read and write operations, when compared to $RADON_R$. The design of an erasure-coded version of $RADON_R^{(S)}$ that never violates atomicity, is an interesting direction that we leave out for future work.

1.2 Other Related Work

Dynamic Reconfiguration: Our setting is closely related to the problem of implementing a consistent memory object in a dynamic setting, where nodes are allowed to voluntarily leave and join the network. The problem involves dynamic reconfiguration of the set of nodes that take part in client operations, which is often implemented via a *reconfig* operation that is initiated by any of the participating processes, including the clients. Any node that wants to leave/join the network makes an announcement, via a *leave/join* operation, before doing so. The problem is extensively studied in the field of distributed algorithms [22], [3], [30], [6], [5]; review and tutorial articles appear in [2], [31], [24].

In our context, the problem of node repair could in fact be thought of as one of dynamic reconfiguration, wherein an involuntary crash is simulated by a voluntary leave operation without an explicit announcement. In this case, a new node joins as a replacement node via the *join* operation, which can be considered as the analogue of a *repair* operation. In the setting of dynamic reconfiguration, every node has a distinct identity; thus the replacement node joins the network with a new identity that is different from the identity of the crashed node [2]. This demands a reconfiguration of the set of participating nodes after every repair. Such reconfigurations get in the way of client operations, and add to the latency of read and write operations [24], in practical implementations. Clearly, a repair operation as considered in this work does not demand any reconfiguration, since a repaired node has the same identity as the crashed node. Also, the current work shows that modeling repair via a static system, permits design of algorithms where clients remain oblivious to the presence of repair operations. Furthermore, addressing storage and communication costs is not the focus of the works in dynamic reconfigurations; specifically, it is not known as to how erasure codes can be advantageously used in dynamic settings. Our $RADON_C$ algorithm shows that when repair is carried out under a static model, it is indeed possible to advantageously use erasure to reduce costs, when the number of concurrent writes are limited.

We make additional comparisons between our model and results to those found in works on dynamic reconfiguration. Several impossibility results exist in the context of implementing a dynamic atomic register and simultaneously guaranteeing liveness; the authors in [30] argue impossibility if there are infinitely many reconfigurations during an execution, while the authors in [6] argue an impossibility when there is no upper bound on message delay. We see, not surprisingly, that even in the problem of repair, we need to suitably limit the number

of crash and repair operations that occur in an execution, even if all crash and repairs are sequentially ordered. In [5], the authors implement a dynamic atomic register under a model that has an (unknown) upper bound D on any point-to-point message delay, and where the number of reconfigurations in any D units of time is limited. Our network condition $N1$ is similar, except that 1) we limit the number of crash and repairs during any broadcast messaging, instead of point-to-point messaging, and 2) we do not assume any bound on the message delay. In practice, limiting number of repairs during broadcast instead of every point-to-point messaging offers resiliency against *straggler* nodes, which refer to the nodes having the worst delays among all nodes. We would also like to note that the algorithm in [5] does not guarantee atomicity, if the number of reconfigurations in D units of time is higher than a set number. This appears similar to $RADON_R$, where atomicity is not guaranteed if we do not satisfy stability condition $N1$. While we show how the slightly tighter model $N2$ can be used to always guarantee atomicity, it is an interesting question as to whether the model $N2$ can be adopted in the work of [5] so as to always guarantee atomicity.

Repair-Efficient Erasure Codes for Distributed Storage: Recently, a large class of new erasure/network codes for storage have been proposed (see [11] for a survey), and also tested in networks [17], [27], [25], where the focus is efficient storage of immutable data, such as, archival data. These new codes are specifically designed to optimize performance metrics like repair-bandwidth and repair-time (of failed servers), and offer significant performance gains when compared to the traditional Reed-Solomon MDS codes [26]. It needs to be explored if these codes can be used in conjunction with the $RADON_C$ algorithm, to further improve the performance costs.

Other Works on using Erasure Codes: Applications of erasure codes to Byzantine fault tolerant DSSs are discussed in [7], [12], [16]. In [29], the authors consider algorithms that use erasure codes for emulating *regular* registers. Regularity [21], [28] is a weaker consistency notion than atomicity.

The rest of the document is organized as follows. Our system model appears in Section 2. The impossibility result, and the network stability conditions appear in Section 3. The three algorithms appear in Sections 4, 5 and 6, respectively. In Section 7, we discuss the storage and communication costs of the algorithms. Section 8 concludes the paper. Due to lack of space, detailed proofs are omitted here; these can be found in the extended version [20].

2 Models and definitions

Processes and Asynchrony: We consider a distributed system consisting of *asynchronous* processes, each with a unique identifier (ID), of three types: a set of *readers*, $\mathcal{R}$; a set of *writers*, $\mathcal{W}$; and a set of n *servers*, $\mathcal{S}$. The readers and writers are together referred to as clients. The set $\mathcal{R} \cup \mathcal{W} \cup \mathcal{S}$ forms a totally ordered set under some defined relation ($>$). The reader and writer processes initiate *read* and *write* operations respectively, and communicate with the servers using messages. A reader or writer can invoke a new operation only after all previous operations invoked by it has completed. The property is referred to as the *well-formedness* property of an execution. We assume that every client/server is connected to every other server via a reliable communication link; thus as long as the destination process is non-faulty, any message sent on the link eventually reaches the destination process.

Crash and Recovery: A client may fail at any point during the execution. At any point during the execution, a server can be in one (and only one) of the following three states: *active*, *crashed* or *repair*. A crash event triggers a server to enter the *crashed* state from an *active* state. The server remains in the *crashed* state for an arbitrary amount of time, but eventually is triggered by a repair event to enter the *repair* state. Crash and repair events are assumed to be externally triggered. A server in the *repair* state can experience another crash event, and go back to the *crashed* state. A server in the *crashed* state does not perform any local computation. The server also does not send or receive messages in the *crashed* state, i.e., any message reaching the server in a *crashed* state is lost. A server which enters the *repair* state has all its local state variables set to default values, i.e., a crash event causes the server to lose all its state variables. A server in the *repair* state can perform computations like in the *active* state.

Atomicity and Liveness: We aim to implement only one atomic read/write memory object, say x, under the MWMR setting on a set of servers, because any shared atomic memory can be emulated by composing individual atomic objects. The object value v comes from some set V; initially v is set to a distinguished 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 is *incomplete* when its invocation step does not have the associated response step; otherwise it is *complete*.

By *liveness of a read or a write operation*, we mean that during any well-formed execution, any read or write operation respectively initiated by a non-faulty reader or writer completes, despite the crash failure of any other client. By *liveness of repair* associated with a crashed server, we mean that the server which enters a crashed state eventually re-enters the active state, unless it experiences a crash event during every repair operation that the server attempts. The liveness of repair holds despite the crash failure of any other client.

Background on Erasure coding: In $RADON_C$, we use an $[n, k]$ linear MDS code [18] over a finite field $\mathbb{F}_q$ to encode and store the value v among the n servers. An $[n, k]$ MDS code has the property that any k out of the n coded elements can be used to recover (decode) the value v. For encoding, v is divided[1] 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, $1 \leq i \leq n$.

Storage and Communication Cost: We define the total storage cost as the size of the data stored across all servers, at any point during the execution of the algorithm. The communication cost associated with a read or write operation is the size of the total data that

[1] In practice v is a file, which is divided into many stripes based on the choice of the code, various stripes are individually encoded and stacked against each other. We omit details of representability of v by a sequence of symbols of $\mathbb{F}_q$, and the mechanism of data striping, since these are fairly standard in the coding theory literature.

gets transmitted in the messages sent as part of the operation. We assume that metadata, such as version number, process ID, etc. used by various operations is of negligible size, and is hence ignored in the calculation of storage and communication cost. Further, we normalize both the costs with respect to the size of the value v; in other words, we compute the costs under the assumption that v has size 1 unit.

3 Network Stability Conditions

3.1 An Impossibility Result

The crash and recovery model described in Section 2 does not impose any restriction on the *rate of crash events, and repair operations* that happen in the system. In other words, the model described above does not limit in any manner the number of crash events/repair operations, which can overlap with any a client operation. In [15], the authors showed that without such restrictions, it is impossible to implement a shared atomic memory service, which guarantees liveness of operations. Below, we state an impossibility result which holds even if there is at most one server in the crashed/repair state at any point during the execution. We then introduce network stability conditions that enable us impose restrictions on the number of crash/repair events that overlap with any operation.

▶ **Theorem 1.** *It is impossible to implement an atomic memory service that guarantees liveness of reads and writes, under the system model described in Section 2, even if* 1*) there is at most one server in the crashed/repair state at any point during the execution, and* 2*) every repair operation completes, and takes the repaired server back to the active state.*

3.2 Network Stability Conditions N1 and N2

We begin with the notions of a group-send operation, and effective consumption of a message.

Group-send operation: The group-send operation is used to abstract the operation of a process sending a list of n messages $\{m_1, \cdots, m_n\}$ to the set of all n servers $\{s_1, \ldots, s_n\} = \mathcal{S}$, where message m_i is send to server $s_i, 1 \leq i \leq n$. Note that this is a mere abstraction of the process sending out n point-to-point messages sequentially to n servers, without interleaving the "send" operations with any significant local computations or waiting for any external inputs. The operation is no more powerful then sending n consecutive messages. The operation is written as *group-send*$([m_1, m_2, \cdots, m_n])$. In the event $m_i = m, \forall i$, we simply write *group-send*(m). Our model allows the sender to fail while executing the *group-send* operation, in which case only a subset of the n servers receive their corresponding messages.

Effective Consumption: We say a process effectively consumes a message m, if it receives m, and executes all steps of the algorithm that depend only on the local state of the process, and the message m; in other words, the process executes all the steps that do not require any further external messages.

▶ **Definition 2** (Network Stability Conditions)**.** Consider a process p executing a *group-send* $([m_1, m_2, \cdots, m_n])$ operation, and consider the following statements:

(a) (i) There exists a subset $\mathcal{S}_\alpha \subseteq \mathcal{S}$ of $|\mathcal{S}_\alpha| = \lceil \alpha n \rceil$ servers, $0 < \alpha < 1$, all of which effectively consume their respective messages from the group-send operation, and (ii) all the servers in $\mathcal{S}_\alpha$ remain in the active state during the interval $[T_1\ T_2]$, where T_1 denotes the point of time of invocation of the *group-send* operation, and T_2 denotes the earliest

point of time in the execution at which all of the servers in $\mathcal{S}_\alpha$ complete the effective consumption of their respective messages.

(b) Further, if effective consumption of the message m_i by server s_i involves sending a response back to the process p, for all $s_i \in \mathcal{S}_\alpha$, then all servers in $\mathcal{S}_\alpha$ remain in the active state during the interval $[T_1\ T_3]$, where T_3 denotes the earliest point of time in the execution at which the process p completes effective consumption of the responses from the all the servers in $\mathcal{S}_\alpha$.

If the network satisfies Statement (a) for every execution of a group-send operation by any process, we say that it satisfies network stability condition $N1$ with parameter α. If the network satisfies Statements (a) and (b) for every execution of a group-send operation by any process, we say that it satisfies network stability condition $N2$ with parameter α.

Clearly, $N2$ implies $N1$. Note that the set $\mathcal{S}_\alpha$ which needs to satisfy the conditions need not be the same for various invocations of group-send operations by either the same or distinct processes. Also, note that in condition $N2$, the process p might crash before completing the effective consumption of the responses from the servers in $\mathcal{S}_\alpha$. In this case we only expect Statement (a) to be satisfied, and not Statement (b). Furthermore, in both $N1$ and $N2$, we do not expect any of these statements to be true, if process p crashes after partial execution of the group-send operation.

4 The $RADON_R$ Algorithm

In this section, we present the $RADON_R$ algorithm, and prove its liveness and atomicity properties for networks that satisfy the network condition $N1$ with $\alpha > \frac{3}{4}$. We begin with some useful notation. Tags are used for version control of the object values. A tag t is defined as a pair (z, w), where $z \in \mathbb{N}$ and $w \in \mathcal{W}$ denotes the ID of a writer. We use $\mathcal{T}$ to denote the set of all the possible tags. For any two tags $t_1, t_2 \in \mathcal{T}$, we say $t_2 > t_1$ if (i) $t_2.z > t_1.z$ or (ii) $t_2.z = t_1.z$ and $t_2.w > t_1.w$. Note that $(\mathcal{T}, >)$ is a totally ordered set.

The protocols for writer, reader, and servers are shown in Fig. 1. Each server stores two state variables (i) (t_{loc}, v_{loc}) – a tag and value pair, initially set to (t_0, v_0), (ii) *status* – a variable that can be in either *active* or *repair* state.

The write and read operations are very similar to those in the ABD algorithm [4], and each consists of two phases. In the first phase, *get-tag*, of a write operation π, the writer queries all servers for local tags, awaits responses from a majority of servers, and selects the maximum tag t^* from among the responses. Next, the writer executes the *put-data* phase, during which a new tag $t_w = tag(\pi)$ is created by incrementing the integer part of t^*, and by incorporating the writer's own ID. The writer then sends pair (t_w, v) to all servers, and awaits acknowledgments (acks) from $\lceil \frac{3n+1}{4} \rceil$ servers before completing the operation. The two phases are identical to those of the ABD algorithm [4], except for the fact that during the second phase, ABD expects acks from only a majority of servers, whereas here we need from $\lceil \frac{3n+1}{4} \rceil$ servers. During a read operation ρ, the reader in the *get-data* phase queries all the servers in S for the respective local tag and value pairs. Onces it receives responses from a majority of servers in S, it picks the pair with the highest tag, which we designate as $t_r = tag(\pi)$. In the subsequent *put-data* phase, the reader writes back the tag t_r and the corresponding value v_r to all servers, and terminates after receiving acknowledgments from $\lceil \frac{3n+1}{4} \rceil$ servers. Once again, we remark that both phases in the read are identical to those of the ABD algorithm, except for the difference in the number of the servers from which acks are expected in the second write-back phase. Note that, during both the write and operations, a server responds to an incoming message only if it is in the active state.

Fig. 1 The protocols for writer, reader, and any server $s \in \mathcal{S}$ in $RADON_R$.

write(v):
get-tag:
group-send(QUERY-TAG)
Await responses from majority
Select the max tag t^*
put-data:
$t_w = (t^*.z + 1, w)$
group-send((PUT-DATA, (t_w, v)))
Terminate after $\lceil \frac{3n+1}{4} \rceil$ acks.

read:
get-data:
group-send(QUERY-TAG-DATA)
Await responses from majority
Select (t_r, v_r), with max tag.

put-data :
group-send((PUT-DATA, (t_r, v_r)))
Wait for $\lceil \frac{3n+1}{4} \rceil$ acks
Return v_r

Server $s \in \mathcal{S}$:
State Variables:
$(t_{loc}, v_{loc}) \in \mathcal{T} \times \mathcal{V}$, initially (t_0, v_0)
$status \in \{active, repair\}$, initially $active$

get-tag-resp, **recv** QUERY-TAG **from writer** w:
if $status = active$ **then**
Send t_{loc} to w

get-data-resp, **recv** QUERY-TAG-DATA **from reader** r:
if $status = active$ **then**
Send (t_{loc}, v_{loc}) to r

put-data-resp, **recv** PUT-DATA, (t, v) **from client** c :
if $status = active$ **then**
if $t > t_{loc}$ **then**
$(t_{loc}, v_{loc}) \leftarrow (t, v)$
Send ack to c.

init-repair :
$status \leftarrow repair$
$(t_{loc}, v_{loc}) \leftarrow (t_0, v_0)$
group-send(REPAIR-TAG-DATA)
Await responses from majority
Select (t_{rep}, v_{rep}), for max tag
$(t_{loc}, v_{loc}) \leftarrow (t_{rep}, v_{rep})$
$status \leftarrow active$

init-repair-resp, **recv** REPAIR-TAG-DATA **from** s':
if $status = active$ **then**
Send (t_{loc}, v_{loc}) to s'

A repair operation is initiated via the action *init-repair*, by an external trigger, at a server which is in the crashed state. Note that we do not explicitly define a *crashed* state since a crash is not a part of the algorithm. We assume that as soon as the repair operation starts, the variable *status* is set to the *repair* state, and also the local (tag, value) pair is set to the default sate (t_0, v_0). The repair operation is essentially the first phase of the read operation, during which the server queries all the servers for the respective local tag and value pairs, and stores the tag and value pair corresponding to the highest tag after receiving responses from a majority of servers. Finally, the repair operation is terminated setting variable *status* to *active* state. A server in S responds to a request generated from *init-repair* phase only if it is in the active state.

4.1 Analysis of $RADON_R$

Liveness of read, write and repair operations in $RADON_R$ follows immediately if we assume condition $N1$ with $\alpha > \frac{3}{4}$. This is because liveness of any operation depends on sufficient number of responses from the servers during the various phases of the operation. From Fig. 1, we know that the maximum number of responses that is expected in any phase is $\lceil \frac{3n+1}{4} \rceil$, which is guaranteed under $N1$ with $\alpha > \frac{3}{4}$.

The tricky part is to prove atomicity of reads and writes. The proof is based on Lemma 13.16 of [23], a restatement of which can be found in [20]. Consider two completed write operations π_1 and π_2, such that, π_2 starts after the completion of π_1. For any completed write operation π, we define $tag(\pi) = t_w$, where t_w is the tag which the writer uses in the *put-data* phase. In this case, one of the requirements the algorithm needs to satisfy to ensure atomicity is $tag(\pi_2) > tag(\pi_1)$. While this fact is straightforward to prove for an algorithm like ABD, which does not have background repair, in $RADON_R$, we need to consider the effect of those repair operations that overlap with π_1, and also those that occur in between π_1

and π_2. The point to note is that such repair operations can potentially restore the contents of the repaired node such that the restored tag is less than $tag(\pi_1)$. We then need to show the absence of propagation of older tags (older than $tag(\pi_1)$) into a majority of nodes, due to a sequence of repairs which happen before π_2 decides its tag. We do this via the following two observations: 1) In Lemma 3, we show that any successful repair operation, which begins after a point of time T, always restores value to one, which corresponds to a tag which is at least as high as the minimum of the tags stored in any majority of active servers at time T. This fact is in turn used to prove a similar property for reads and writes, as well. 2) We next show (as part of proof of Theorem 5), under the assumption of $N1$ with $\alpha > 3/4$, the existence of a point of time T before the completion of π_1 such that a majority of nodes are active at T, and all of whose tags are at least as high as $tag(\pi_1)$. The two steps are together used to prove that $tag(\pi_2) > tag(\pi_1)$. A similar sequence of steps are used to show atomicity properties of read operations, as well.

For a completed read operation π, $tag(\pi) = t_r$, where t_r is the tag corresponding to the value v_r returned by the reader. For a completed repair π, $tag(\pi) = t_{rep}$, where t_{rep} is the tag corresponding to the value restored during the repair operation.

▶ **Lemma 3.** *Let β denote a well-formed execution of $RADON_R$. Suppose T denotes a point of time in β such that there exists a majority of servers $\mathcal{S}_m$, $\mathcal{S}_m \subset \mathcal{S}$ all of which are in the active state at time T. Also, let t_s denote the value of the local tag at server $s \in \mathcal{S}_m$, at time T. Then, if π denotes any completed repair or read operation that is initiated after time T, we have $tag(\pi) \geq \min_{s\in\mathcal{S}_m} t_s$. Also, if π denotes any completed write operation that is initiated after time T, we have $tag(\pi) > \min_{s\in\mathcal{S}_m} t_s$.*

▶ **Theorem 4** (Liveness)**.** *Let γ denote a well-formed execution of $RADON_R$, under the condition $N1$ with $\alpha > \frac{3}{4}$. Then every operation initiated by a non-faulty client completes.*

▶ **Theorem 5** (Atomcity)**.** *Every execution of the $RADON_R$ algorithm operating under the $N1$ network stability condition with $\alpha > \frac{3}{4}$, is atomic.*

We note that, though Lemma 3 gives a result about completed operations, condition $N1$ is not a prerequisite for the result in Lemma 3. In other words, the result in Lemma 3 holds for any completed operation, even if condition $N1$ is violated. As we will see, this is an important fact that we will use to establish atomicity of $RADON_R^{(S)}$ for any execution.

5 Algorithm $RADON_R$

In this section, we present the erasure-code based $RADON_C$ algorithm for implementing atomic memory service, and performing repair of crashed nodes. The algorithm uses $[n, k]$ MDS codes for storage. Liveness and atomicity are guaranteed under the following assumptions: 1) the $N1$ network stability condition with $\alpha \geq \frac{3n+k}{4n}$, 2) the number of write operations concurrent with a read or repair operation is at most δ. The precise definition of concurrency depends on the algorithm itself, and appears later in this section. The $RADON_C$ algorithm has significantly reduced storage and communication cost requirements than $RADON_R$, when δ is limited.

The algorithm (see Fig. 2) is a natural generalization of the $RADON_R$ algorithm accounting for the fact that we use MDS codes. The write operation has two phases, where the first phase finds the maximum tag in the system based on majority responses. During the second phase, the writer computes the coded elements for each of the n servers and uses the group-send operation to disperse them. The *group-send* operation here uses a vector of

Fig. 2 The protocols for write, reader, and any server $s_i \in \mathcal{S}$ in $RADON_C$.

write(v):

get-tag:
group-send(QUERY-TAG)
Await responses from majority
Select the max tag t^*

put-data:
$t_w = (t^*.z + 1, w)$.
code-elems $= [(t_w, c_1), \ldots, (t_w, c_n)]$, $c_i = \Phi_i(v)$
group-send(CODE-ELEMENTS, *code-elems*)
Terminate after $\lceil \frac{3n+k}{4} \rceil$ acks

read:

get-data:
group-send(QUERY-LIST)
Wait for $\lceil \frac{n+k}{2} \rceil$ *Lists*
Select the max tag, t_r, whose corresponding value, v_r, is decodable using the *Lists*.

put-data:
code-elems $= [(t_r, c_1), \ldots, (t_r, c_n)]$, $c_i = \Phi_i(v_r)$
group-send(CODE-ELEMENTS, *code-elems*)
Wait for $\lceil \frac{3n+k}{4} \rceil$ acks
Return v_r

Server $s_i \in \mathcal{S}$:

State Variables:
status $\in \{active, repair\}$, initially *active*
List $\subseteq \mathcal{T} \times \mathcal{C}_s$, initially $\{(t_0, \Phi_i(v_0))\}$

get-tag-resp,**recv** QUERY-TAG **from writer** w:
if *status* $=$ *active* **then**
$t^* = \max_{(t,c) \in List} t$
Send t^* to w

get-data-resp, **recv** QUERY-LIST **from reader** r:
if *status* $=$ *active* **then**
Send *List* to r

put-data-resp, **recv** CODE-ELEMENTS, (t, c_i) **from** p :
if *status* $=$ *active* **then**
List $\leftarrow$ *List* $\cup \{(t, c_i)\}$
if $|List| > \delta + 1$ **then**
Retain the (tag, coded-element) pairs for the $\delta + 1$ highest tags in *List*, and delete the rest.
Send ack to p.

init-repair :
status $\leftarrow$ *repair*
group-send(REPAIR-LIST)
Wait for $\lceil \frac{n+k}{2} \rceil$ *Lists*
Find (tag, value) pairs decodable from *Lists*.
Restore local *List* via re-encoding and retaining the (tag, coded-element) pairs corresponding to at most $\delta + 1$ highest tags, from the above pairs
status $\leftarrow$ *active*

init-repair-resp, **recv** REPAIR-LIST **from server** s':
if *status* $=$ *active* **then**
Send *List* to s'

length n, where the i^{th} element denotes the message for the i^{th} server, $1 \leq i \leq n$. Each server keeps a *List* of up to $(\delta + 1)$ (tag, coded-element) pairs. Every time a (tag, coded-element) message arrives from a writer, the pair gets added to the *List*, which is then pruned to at most $(\delta + 1)$ pairs, corresponding to the highest tags. The writer terminates after getting acks from $\lceil \frac{3n+k}{4} \rceil$ servers.

During a read operation, the reader queries all servers for their entire local *Lists*, and awaits responses from $\lceil \frac{n+k}{2} \rceil$ servers. Once the reader receives *Lists* from $\lceil \frac{n+k}{2} \rceil$ servers, it selects the highest tag t_r whose corresponding value v_r can be decoded using the using the coded elements in the lists. The read operation completes following a write-back of (t_r, v_r) using the *put-data* phase.

The repair operation is very similar to the first phase of the read operation, during which a server collects lists from $\lceil \frac{n+k}{2} \rceil$ servers. But this time, the server figures out the set of all the possible tags that can be decoded from among the *Lists*, and prunes the set to the highest $(\delta + 1)$ tags. The repaired *List* then consists of (tag, coded-element) pairs corresponding these (at most) $(\delta + 1)$ tags. Assuming repair of server i, the creation of a coded-element corresponding to a value v involves first decoding the value v, and then computing $\Phi_i(v)$ (referred to as re-encoding in Fig. 2).

5.1 Analysis of $RADON_C$

Throughout this section, we assume network stability condition $N1$ with $\alpha \geq \frac{3n+k}{4n}$. Tags for completed read and write operations are defined in the same manner as we did for $RADON_R$; we avoid repeating them here. We first discuss liveness properties of $RADON_C$. Let us

first consider liveness of repair operations. Towards this, note from the algorithm in Fig. 2 that a repair operation never gets stuck even if it does not find any set of k *Lists* among the responses, all of which have a common tag. In such a case, the algorithm allows the possibility that the repaired *List* is simply empty, at the point of execution when the server re-enters the active state. In other words, liveness of a repair operation is trivially proved, i.e., a server in a repair state always eventually reenters the active state, as long as it does not experience a crash during the repair operation. The triviality of liveness of repair operations, observed above, does not extend to read operations. For a read operation to complete the *get-data* phase, it must be able to find a set of k *Lists* among the responses all of which contain coded-elements corresponding to a common tag; otherwise a read operation gets stuck. The discussion above motivates the following definitions of valid read and valid repair operations.

▶ **Definition 6** (Valid Read and Repair Operations)**.** A read operation will be called as a valid read if the associated reader remains alive at least until the reception of the $\lceil \frac{n+k}{2} \rceil$ responses during the *get-data* phase. Similarly, a repair operation will be called a valid repair if the associated server does not experience a further crash event during the repair operation.

▶ **Definition 7** (Writes Concurrent with a Valid Read (Repair))**.** Consider a valid read (repair) operation π. Let T_1 denote the point of initiation of π. For a valid read, let T_2 denote the earliest point of time during the execution when the associated reader receives all the $\lceil \frac{n+k}{2} \rceil$ responses. For a valid repair, let T_2 denote the point of time during the execution when the repair completes, and takes the associated server back to the active state. Consider the set $\Sigma = \{\sigma : \sigma$ is any write operation that completes before π is initiated$\}$, and let $\sigma^* = \arg\max_{\sigma \in \Sigma} tag(\sigma)$. Next, consider the set $\Lambda = \{\lambda : \lambda$ is any write operation that starts before T_2 such that $tag(\lambda) > tag(\sigma^*)\}$. We define the number of writes concurrent with the valid read (repair) operation π to be the cardinality of the set Λ.

The above definition captures all the write operations that overlap with the read, until the time the reader has all data needed to attempt decoding a value. However, we ignore those write operations that might have started in the past, and never completed yet, if their tags are less than that of any write that completed before the read started. This allows us to ignore write operations due to failed writers, while counting concurrency, as long as the failed writes are followed by a successful write that completed before the read started.

The following lemma could be considered as the analogue of Lemma 3 for $RADON_C$. The first part of the lemma shows that under $N1$ with $\alpha \geq \frac{3n+k}{4n}$, the repaired *List* is never empty; there is always at least one (tag, coded-element) pair in the repaired *List*. Parts 2 and 3 are used to prove liveness and atomicity of client operations.

▶ **Lemma 8.** *Consider any well-formed execution β of $RADON_C$ operating under the network stability condition $N1$ with $\alpha \geq \frac{3n+k}{4n}$. Further assume that the number of writes concurrent with any valid read or repair operation is at most δ. For any operation π, consider the set $\Sigma = \{\sigma : \sigma$ is a read or a write in β that completes before π begins$\}$, and also let $\sigma^* = \arg\max_{\sigma \in \Sigma} tag(\sigma)$. Then, the following statements hold:*

- *If π denotes a completed repair operation on a server $s \in \mathcal{S}$, then the repaired List of server s due to π contains the pair $(tag(\sigma^*), c_s^*)$.*
- *If π denotes a read operation associated with a non-faulty reader r, and further, if $\mathcal{S}_1$ denotes the set of $\lceil \frac{n+k}{2} \rceil$ servers whose responses, say $\{L_\pi(s), s \in \mathcal{S}_1\}$, are used by r to attempt decoding of a value in the get-data phase, then there exists $\mathcal{S}_2 \subseteq \mathcal{S}_1$, $|\mathcal{S}_2| = k$, such that $\forall s \in \mathcal{S}_2, (tag(\sigma^*), c_s^*) \in L_\pi(s)$.*

- *If π denotes a write operation associated with a non-faulty writer w, and further if $\mathcal{S}_1$ denotes the set of majority servers whose responses are used by w to compute max-tag in the get-tag phase, then there exists a server $s \in \mathcal{S}_1$, whose response tag $t_s \geq tag(\sigma^*)$.*

Here, c_s^* denotes the coded-element of server s for value v^*, associated with $tag(\sigma^*)$.

▶ **Theorem 9** (Liveness). *Let β denote a well-formed execution of $RADON_C$, operating under the $N1$ network stability condition with $\alpha \geq \frac{3n+k}{4n}$ and δ be the maximum number of write operations concurrent with any valid read or repair operation. Then every operation initiated by a non-faulty client completes.*

▶ **Theorem 10** (Atomicity). *Any execution of $RADON_C$, operating under condition $N1$ with $\alpha \geq \frac{3n+k}{4n}$, is atomic, if the maximum number of write operations concurrent with a valid read or repair operation is δ.*

6 The $RADON_R$ Algorithm

In this section, we present the $RADON_R^{(S)}$ algorithm having the property that every execution is atomic. Liveness is guaranteed under the slightly stronger network stability condition $N2$ with $\alpha > \frac{3}{4}$. In comparison wtih $RADON_R$, the algorithm has extra phases for both read and write operations, in order to guarantee safety of every execution.

The write operation has three phases (see Fig. 3). The first two phases are identical to those of $RADON_R$ during which the writer queries for the local tags, and then sends out the new (tag, value) pair, respectively. In the third phase, called *confirm-data*, the writer ensures the presence of at least a majority of servers, which the writer knows for sure that received its data during the second phase, *put-data*. In order to facilitate the *confirm-data* phase, the servers maintain a *Seen* variable. Any time the server receives a value from a writer, the server adds the corresponding (tag, writer ID) pair to the *Seen* list. Next, during the *confirm-data-resp* phase, the server responds to the writer only if this (tag, writer ID) pair is part of the *Seen* variable. The idea is that if the server experiences a crash and a successful repair operation in between the *put-data* and *confirm-data* phases, the server no longer has the (tag, writer ID) pair in its *Seen* variable, and hence does not respond to the *confirm-data* phase. This is because, a crash removes all state variables, including *Seen*, and the repair algorithm (see Fig. 3) simply restores the *Seen* variable to its default value, the empty set. Further, by ensuring that the writer expects acks from among a majority of servers in *confirm-data*, from among the $\frac{3n+1}{4}$ servers whose acks were obtained during *put-data*, we can guarantee that any execution is atomic.

The read operation also has three phases, first two of which are identical to those of $RADON_R$, except for the use of the *Seen* variable in the server during the *put-data* phase. The third phase is the *confirm-data* phase as in the write operation. The repair operation has one phase, and is nearly exactly identical to that of $RADON_R$. Note that the *Seen* variable gets reset to its initial value during repair.

6.1 Analysis of $RADON_R^{(S)}$

We overview the proofs of liveness and atomicity before formal claims. For liveness of writes, we assume $N2$ with $\alpha > \frac{3}{4}$, and argue the existence of a majority $\mathcal{S}_m$ of servers all of which remain active from the point of time at which the *group-send* operation gets initiated in the *put-data* phase, till the point of time all the servers in $\mathcal{S}_m$ effectively consume requests for *confirm-data* from the writer. In this case, write operation completes after receiving acks from servers in $\mathcal{S}_m$ during the *confirm-data* phase. The set $\mathcal{S}_m$ exists because, under $N2$

Fig. 3 The protocols for writer, reader, and any server $s \in \mathcal{S}$ in $RADON_R^{(S)}$.

write(v):
get-tag:
  group-send(QUERY-TAG)
  Await responses from majority
  Select the max tag t^*
put-data:
  $t_w = (t^*.z + 1, w)$.
  group-send((PUT-DATA, (t_w, v)))
  Wait for $\lceil \frac{3n+1}{4} \rceil$ acks (say from $\mathcal{S}_\alpha$)
confirm-data:
  group-send((CONFIRM-DATA, t_w))
  Terminate after acks from majority from among servers in $\mathcal{S}_\alpha$

read:
get-data:
  group-send(QUERY-TAG-DATA)
  Await responses from majority
  Select (t_r, v_r), with max tag.
put-data :
  group-send((PUT-DATA, (t_r, v_r)))
  Wait for $\lceil \frac{3n+1}{4} \rceil$ acks (say from $\mathcal{S}_\alpha$)
confirm-data:
  group-send((CONFIRM-DATA, t_r))
  Await acks from a majority of servers in $\mathcal{S}_\alpha$
  Return v_r

Server $s \in \mathcal{S}$:
State Variables:
$(t_{loc}, v_{loc}) \in \mathcal{T} \times \mathcal{V}$, initially (t_0, v_0)
$status \in \{active, repair\}$, initially *active*
$Seen \subseteq \mathcal{T} \times \{\mathcal{W} \cup \mathcal{R}\}$, initially empty

get-tag-resp, **recv** QUERY-TAG **from writer** w:
  if $status = active$ **then**
    Send t_{loc} to w
get-data-resp, **recv** QUERY-TAG-DATA **from reader** r:
  if $status = active$ **then**
    Send (t_{loc}, v_{loc}) to r
put-data-resp, **recv** (PUT-DATA, (t, v)) **from** c :
  if $status = active$ **then**
    if $t > t_{loc}$ **then**
      $(t_{loc}, v_{loc}) \leftarrow (t, v)$
    $Seen \leftarrow Seen \cup \{(t, c)\}$
    Send ack to c.

confirm-data-resp, **recv** (CONFIRM-DATA, t) **from** c:
  if $status = active$ **then**
    if $(t, c) \in Seen$ **then**
      Remove (t, c) from $Seen$
      Send ack to client c.

init-repair :
  $status \leftarrow repair$
  $(t_{loc}, v_{loc}) \leftarrow (t_0, v_0)$
  $Seen \leftarrow \emptyset$
  group-send(REPAIR-TAG-DATA)
  Await responses from majority.
  Select (t_{rep}, v_{rep}), with max tag
  $(t_{loc}, v_{loc}) \leftarrow (t_{rep}, v_{rep})$
  $status \leftarrow active$

init-repair-resp, **recv** REPAIR-TAG-DATA **from** s':
  if $status = active$ **then**
    Send (t_{loc}, v_{loc}) to s'

with $\alpha > \frac{3}{4}$, a set $\mathcal{S}_\alpha$ of $\lceil \frac{3n+1}{4} \rceil$ servers remain alive from the start of the group-send, till the effective consumption of the acks by the writer in *put-data* phase. Also, a second set $\mathcal{S}'_\alpha$ of $\lceil \frac{3n+1}{4} \rceil$ servers remain active from the start of the group-send in the *confirm-data* phase, till all servers in $\mathcal{S}'_\alpha$ complete the respective effective consumption from this group-send. We note that $\mathcal{S}'_\alpha \cap \mathcal{S}_\alpha$ is at least a majority. We next use the observation that the *group-send* operation in the *confirm-data* phase forms part of the effective consumption of the last of the acks in the *put-data* phase. Using this, we argue that the servers in $\mathcal{S}'_\alpha \cap \mathcal{S}_\alpha$ remain active till they effectively consume message from *group-send* operation of the *confirm-data* phase, and thus $\mathcal{S}'_\alpha \cap \mathcal{S}_\alpha$ is a candidate for $\mathcal{S}_m$. The liveness of read is similar to that of write, while liveness of repair is straightforward under $N2$ with $\alpha > \frac{3}{4}$.

Towards proving atomicity of reads and writes, we first define tags for completed reads, writes and repair operations exactly in the same manner as we did in $RADON_R$. Consider two completed write operations π_1 and π_2 such that π_2 starts after the completion of π_1, and we need to show that $tag(\pi_2) > tag(\pi_1)$. As in $RADON_R$, we do this in two parts: Lemma 3 holds as it is for $RADON_R^{(S)}$ as well. Recall that Lemma 3 essentially shows that if a majority of active nodes is locked-on to any particular tag, say t', at a specific point of time T during the execution of the algorithm, then any repair operation which begins after the time T always restores the tag to one which is at least as high as t'. The challenge now is to show the existence of these favorable points of time instants T as needed in the assumption

of the lemma. While in $RADON_R$, we used the $N1$ to argue this, in $RADON_R^{(S)}$, we do not use $N2$; instead we rely on the third *confirm-data* phase of the first write operation π_1.

▶ **Theorem 11** (Liveness). *Let β denote a well-formed execution of $RADON_R^{(S)}$ under condition $N2$ with $\alpha > \frac{3}{4}$. Then every operation initiated by a non-faulty client completes.*

▶ **Theorem 12** (Atomcity). *Every execution of the $RADON_R^{(S)}$ algorithm is atomic.*

7 Storage and Communication Costs of Algorithms

We give a justification of storage and communication cost numbers of the three algorithms, appearing in Table 1. Recall that the size of value v is assumed to be 1 and also that we ignore the costs due to metadata. It is clear that both $RADON_R$ and $RADON_R^{(S)}$ have storage cost n, write cost n, and read cost $2n$ (due to write back). For $RADON_C$, each server stores at most $\delta + 1$ coded-elements, where each element has size $\frac{1}{k}$. Thus storage cost of $RADON_C$ is $(\delta + 1)\frac{n}{k}$. The write cost of $RADON_C$ is simply $\frac{n}{k}$, and the contribution comes from the writer sending one coded-element to each of the n servers. For a read, getting the entire *Lists* during the $get - data$ phase incurs a cost of $(\delta + 1)\frac{n}{k}$. The write-back phase incurs an additional cost of $\frac{n}{k}$. Thus, the total read cost in $RADON_C$ is $(\delta + 2)\frac{n}{k}$.

8 Conclusions

In this paper, we provided an erasure-code-based algorithm for implementing atomic memory, having the ability to perform repair of crashed nodes in the background, without affecting client operations. We assumed a static model with a fixed, finite set of nodes, and also a practical network condition $N1$ to facilitate repair. We showed how the usage of MDS codes significantly improve storage and communication costs over a replication based solution, when the number of writes concurrent with a read or repair is limited. Liveness and atomicity are guaranteed as long as $N1$ is satisfied; however violation of $N1$ can lead to non-atomic executions. We further showed how a slightly stringent network condition $N2$ can be used to construct a replication based algorithm that always guarantees atomicity. Ongoing efforts include exploring possibility of using repair-efficient erasure codes [11] in $RADON_C$, and testbed evaluations on cloud based infrastructure.

References

1 M. K. Aguilera, R. Janakiraman, and L. Xu. Using erasure codes efficiently for storage in a distributed system. In *Proceedings of International Conference on Dependable Systems and Networks (DSN)*, pages 336–345, 2005.

2 M. K. Aguilera, I. Keidar, D. Malkhi, J. P. Martin, and A. Shraery. Reconfiguring replicated atomic storage: A tutorial. *Bulletin of the EATCS*, 102:84–081, 2010.

3 M. K. Aguilera, I. Keidar, D. Malkhi, and A. Shraer. Dynamic atomic storage without consensus. *Journal of the ACM*, pages 7:1–7:32, 2011.

4 H. Attiya, A. Bar-Noy, and D. Dolev. Sharing memory robustly in message passing systems. *Journal of the ACM*, 42(1):124–142, 1996.

5 H. Attiya, H. C. Chung, F. Ellen, S. Kumar, and J. L. Welch. Simulating a shared register in an asynchronous system that never stops changing - (extended abstract). In *Distributed Computing - 29th International Symposium, DISC 2015, Tokyo, Japan, October 7-9, 2015, Proceedings*, pages 75–91, 2015.

6 R. Baldoni, S. Bonomi, A. M. Kermarrec, and M. Raynal. Implementing a register in a dynamic distributed system. In *Distributed Computing Systems, 2009. ICDCS'09. 29th IEEE International Conference on*, pages 639–647, June 2009.

7 C. Cachin and S. Tessaro. Optimal resilience for erasure-coded byzantine distributed storage. In *Proceedings of International Conference on Dependable Systems and Networks (DSN)*, pages 115–124, 2006.

8 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)*, pages 253–260, 2014.

9 F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber. Bigtable: A distributed storage system for structured data. *ACM Trans. Comput. Syst.*, 26(2):4:1–4:26, jun 2008.

10 G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. Dynamo: Amazon's highly available key-value store. In *Proceedings of Twenty-first ACM SIGOPS Symposium on Operating Systems Principles*, SOSP'07, pages 205–220, New York, NY, USA, 2007. ACM.

11 A. G. Dimakis, K. Ramchandran, Y. Wu, and C. Suh. A survey on network codes for distributed storage. *Proceedings of the IEEE*, 99(3):476–489, 2011.

12 D. Dobre, G. Karame, W. Li, M. Majuntke, N. Suri, and M. Vukolić. Powerstore: proofs of writing for efficient and robust storage. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, pages 285–298, 2013.

13 P. Dutta, R. Guerraoui, and R. R. Levy. Optimistic erasure-coded distributed storage. In *Proceedings of the 22nd international symposium on Distributed Computing (DISC)*, pages 182–196, Berlin, Heidelberg, 2008.

14 R. Fan and N. Lynch. Efficient replication of large data objects. In *Distributed algorithms*, Lecture Notes in Computer Science, pages 75–91, 2003.

15 R. Guerraoui, R. R. Levy, B. Pochon, and J. Pugh. The collective memory of amnesic processes. *ACM Trans. Algorithms*, 4(1):1–31, 2008.

16 J. Hendricks, G. R. Ganger, and M. K. Reiter. Low-overhead byzantine fault-tolerant storage. *ACM SIGOPS Operating Systems Review*, 41(6):73–86, 2007.

17 C. Huang, H. Simitci, Y. Xu, A. Ogus, B. Calder, P. Gopalan, J. Li, and S. Yekhanin. Erasure coding in windows azure storage. In *Proc. USENIX Annual Technical Conference (ATC)*, pages 15–26, 2012.

18 W. C. Huffman and V. Pless. *Fundamentals of error-correcting codes*. Cambridge university press, 2003.

19 K. M. Konwar, N. Prakash, E. Kantor, N. Lynch, M. Medard, and A. A. Schwarzmann. Storage-optimized data-atomic algorithms for handling erasures 124 and errors in distributed storage systems. In *30th IEEE International Parallel & Distributed Processing Symposium (IPDPS)*, 2016.

20 K. M. Konwar, N. Prakash, M. Medard, and N. Lynch. RADON: Repairable atomic data object in networks. *CoRR*, abs/1605.05717, 2016.

21 L. Lamport. On interprocess communication. *Distributed computing*, 1(2):86–101, 1986.

22 N. Lynch and A. A. Shvartsman. RAMBO: A reconfigurable atomic memory service for dynamic networks. In *Proceedings of 16th International Symposium on Distributed Computing (DISC)*, pages 173–190, 2002.

23 N. A. Lynch. *Distributed Algorithms*. Morgan Kaufmann Publishers, 1996.

24 Peter Musial, Nicolas Nicolaou, and Alexander A. Shvartsman. Implementing distributed shared memory for dynamic networks. *Communications of the ACM*, 57(6):88–98, 2014.

25 K. V. Rashmi, P. Nakkiran, J. Wang, N. B. Shah, and K. Ramchandran. Having your cake and eating it too: Jointly optimal erasure codes for i/o, storage, and network-bandwidth. In *13th USENIX Conference on File and Storage Technologies (FAST)*, pages 81–94, 2015.

26 I. S. Reed and G. Solomon. Polynomial codes over certain finite fields. *Journal of the society for industrial and applied mathematics*, 8(2):300–304, 1960.

27 M. Sathiamoorthy, M. Asteris, D. Papailiopoulos, A. G. Dimakis, R. Vadali, S. Chen, and D. Borthakur. XORing elephants: novel erasure codes for big data. In *Proceedings of the 39th international conference on Very Large Data Bases*, pages 325–336, 2013.

28 C. Shao, J. L. Welch, E. Pierce, and H. Lee. Multiwriter consistency conditions for shared memory registers. *SIAM Journal on Computing*, 40(1):28–62, 2011.

29 A. Spiegelman, Y. Cassuto, G. Chockler, and I. Keidar. Space Bounds for Reliable Storage: Fundamental Limits of Coding. In *Proceedings of the International Conference on Principles of Distributed Systems (OPODIS2015)*, 2015.

30 A. Spiegelman and I. Keidar. On liveness of dynamic storage. *CoRR*, abs/1507.07086, 2015. URL: `http://arxiv.org/abs/1507.07086`.

31 A. Spiegelman, I. Keidar, and D. Malkhi. Dynamic reconfiguration: A tutorial. *OPODIS 2015*, 2015.