RADON: Repairable Atomic Data Object in Networks

Kishori M. Konwar, N. Prakash, Nancy Lynch, Muriel Médard

Department of EECS, Massachusetts Institute of Technology
{kishori, lynch}@casil.mit.edu, {prakashn, medard }@mit.edu

Abstract. In this paper, we provide fault-tolerant algorithms, for implementing atomic memory service in an asynchronous network of storage nodes, with the ability to perform background repair of crashed nodes, thereby increasing the durability of the storage service. A crashed node is assumed to lose all its data, both from the volatile memory as well as the 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 re-enters active state, and resumes participation in ongoing/future read, write and repair operations.

We argue that, under arbitrary conditions where there is no restriction on the number of repair processes being performed in relation to reads and writes, it is not possible to achieve liveness and atomicity, simultaneously. Therefore, we introduce two network "stability" conditions, $N1$ and $N2$, which are mostly likely to be respected by practical storage networks. Next, we design the $RADON_L$ algorithm which guarantees liveness and safety as long as condition $N1$ holds in the network. However, the algorithm may violate safety of an execution if $N1$ is not observed by the network. Next, under the assumption of the slightly stronger network condition $N2$, we design the algorithm $RADON_S$, which guarantees atomicity of any execution, but liveness is guaranteed only if $N2$ holds. The guarantee of safety in $RADON_S$ comes at the cost of adding an additional phase in the read and write operations, compared to $RADON_L$. Both $RADON_L$ and $RADON_S$ use replication of data for fault tolerance.

Further, we provide a third algorithm, called $RADON_C$, that is based on erasure codes. $RADON_C$ guarantees liveness and atomicity under the assumption of $N1$, and significantly improves upon the storage and communication costs of $RADON_L$ and $RADON_S$, under scenarios when the number of write operations concurrent with a read or a repair operation is bounded.

Keywords: Atomicity, repair, fault-tolerance, storage cost, erasure codes, **Submission Type:** Regular

1 Introduction

Distributed storage systems (DSS) that store massive data sets across hundreds of commodity storage servers are increasingly being used for both industrial and scientific applications, ranging from sequencing genomic data to e-commerce. Many large-scale applications demand concurrent and consistent access to the stored data by multiple writers and multiple readers (MWMR). The most desirable consistency model for a distributed system is *atomicity*. In simple terms, atomic consistency gives the users of the data service the impression that the various concurrent read and write operations happen sequentially. Implementations of atomicity on an asynchronous system under message passing framework, in the presence of node failure, is a challenge. The ability to tolerate failures and network delays, recovering and repairing of crashed nodes without disrupting the service, (asynchrony) are essential features of any robust DSS.

One of most popular form of distributed computing is cloud computing, its ability ot scale on demand is highly desirable. Cloud computing services allow users to add or remove nodes to running distributed applications, a term commonly referred to as *autoscaling* [1–4]. While addition or removal of nodes is mostly a controlled process, node crashes that result in loss of stored data, are uncontrolled events. Crashes are a norm, rather than exception, in today's large scale data

centers, primarily due to the use of commodity storage devices to offer affordable services. Leaving the crashed machines without restarting or repairing will lead to gradual degradation of the service over time. Repair of failed servers is an integral feature of many cloud based infrastructures [5, 6].

Most replication based algorithms for implementing atomic memory objects such as in [7] and [8], do not provide mechanisms for repairing crashed nodes while keeping the service available. Although algorithms such as RAMBO [9] and DynaStore [10] implement MWMR atomic memory object in a dynamic network, and allow joining and leaving of nodes, they are not designed for repairs. Furthermore, it is not clear how these algorithms can be modified to handle repairs, or extended to allow erasure-coding based storage. Though repairs are an integral part of daily operations of large-scale data-centers, [11], to the best of our knowledge, the literature lacks commonly agreed upon definitions of crash and repair, that permit design and analysis of algorithms.

In this paper, we define a model for crash and repair, which is simple enough to facilitate analysis of algorithms, while being relevant to large-scale systems. We also define a liveness property for repair operations. It is natural to expect a restriction on the number of repair operations that are carried out during the duration of a write or read operation. In fact, the first result of this paper shows if there is no such restriction, it is impossible to implement an atomic memory service that also guarantees liveness of operations. We formulate network stability conditions $N1$ and $N2$, that can further be used to limit the number of repairs operations overlapping with a client operation. These stability conditions are algorithm independent, and most likely to be satisfied in any practical storage network. At a high level, the condition $N1$ restricts set of servers that can be in the crashed or repair state any time a process (client or server) wants to *ping* all the n servers with corresponding messages. The condition $N2$ goes beyond $N1$, 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 responds back to the sender of the message.

We present three algorithms for implementing atomic memory service in MWMR setting. These algorithms guarantee liveness of read, writes and repairs under appropriate settings of the network conditions. The first algorithm, $RADON_L$, guarantees atomicity and liveness of operations under $N1$, if more than $3/4^{\text{th}}$ of all servers remain active during any ping operation. Though the latter condition on the network is reasonable for most scenarios of practical interest, if it is not satisfied in a rare instance, it is possible to have executions of $RADON_L$ that are not atomic. The second algorithm, $RADON_S$, guarantees atomicity of every execution, but liveness is guaranteed only under the stronger condition $N2$, with more than $3/4^{\text{th}}$ of all servers remain active during any ping-pong operation. The guarantee of atomicity of every execution not only demands a slightly stringent network condition, but also needs extra phases in the algorithm when compared to $RADON_L$. 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. The issue often arises in the context of discussion on the CAP Theorem [12]. For example, BigTable, a DSS by Google, prefers safety over liveness [13], whereas, Amazon's Dynamo does not compromise liveness but settles for *eventual consistency* [14]. Both $RADON_L$ and $RADON_L$ use replication of data for fault-tolerance.

Our third algorithm, $RADON_C$, uses Maximum Distance Separable (MDS) erasure codes [15] for fault-tolerance, and help in significantly reducing storage and communication costs of reads and writes, under the scenario when the number of writes concurrent with a read or repair operation, is known to be limited. Storage and communication costs are important performance measures

in large scale DSS. 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 [16], erasure code based algorithms like in [17], [18], [19], or [20] for implementing atomic memory service offer significant storage and communication cost savings over replication based algorithms, in many regimes of operation. For instance CASGC [19] improves the costs under the scenario when the number of writes concurrent with a read is known to be limited, whereas SODA [20] trades-off write cost to achieves optimal storage cost, which is meaningful in systems with infrequent writes. A summary of algorithms in this paper appears in Table 1.

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

Table 1. Performance comparison of $RADON_L$, $RADON_S$ and $RADON_C$, where n is the number of servers, and δ is the maximum number of writes concurrent with a read or a repair operation. For $RADON_C$, we assume usage of $[n, k]$ MDS codes, see Section 6. See Appendix G for a justification on the costs.

Other Related Work Applications of erasure codes to Byzantine fault tolerant DSSs are discussed in [21], [22], [23]. In [16], the authors consider algorithms that use erasure codes for emulating *regular* registers. Regularity [24], [25] is a weaker consistency notion than atomicity. Recently, a large class of new erasure/network codes for storage have been proposed (see [26] for a survey), also and tested in networks [27], [28], [29], where the focus is efficient storage of immutable data, such as, archival data. It will be interesting to see, as part of future work, if these codes can be used in conjunction with the $RADON_C$ algorithm, to further improve storage and communication costs.

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.

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*, $\mathcal{R}$ and 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. Any reader or writer can initiate a new operation only after a previous operation, if any, at the same client has completed, referred to as the *well-formedness* property of an execution. We assume that every pair of processes is connected through a reliable communication link, i.e., 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 an repair event to enter the *repair* state. Crash and repair events are assumed to be externally triggered. It is possible that a server in the *repair* state experiences another crash event, and goes 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* in an execution 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 of the algorithm, any read or write operation respectively initiated by 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.

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 this section, we first argue that without such restrictions, it is impossible to implement a shared atomic memory service, which guarantees liveness of operations. The impossibility holds even if there is at most one server in the 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 every repair operation completes, and takes the repaired server back to the active state.*

Proof. We give a proof sketch here, full proof appears in Appendix A. Consider a write operation which sends messages to a set $\mathcal{S}_w \subset \mathcal{S}$ of servers, before expecting any response. We create a sequence of crashes and repairs of the servers in $\mathcal{S}_w$, where we crash and repair one server at a time. We delay the writer messages to the servers in $\mathcal{S}_w$ such that each message arrives at a server,

while it is in crashed state, but the server undergoes a successful repair soon thereafter before any other server gets the respective message from the writer. In this case, liveness is compromised if the writer expects response due to messages sent to servers $\mathcal{S}_w$. In case, the algorithm is such that the writer decides to terminate even before expecting any response, it would violate atomicity.

3.2 Network Stability Conditions $N1$ and $N2$

We begin with some useful definitions.

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. Essentially, this 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 when $m_i = m, \forall i$, we simply write *group-send*(m). We note that the model allows the sender to fail after partially executing the *group-send* operation, in which case only a subset of the n servers receive their corresponding messages.

Effective Consumption of message m 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 1 (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 $S_\alpha \subseteq \mathcal{S}$ of $|S_\alpha| = \lceil \alpha n \rceil$ servers, $0 < \alpha < 1$, all of which effectively consume their respective messages from the group-send operation, (ii) all the servers in S_α 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 S_α 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 $s_i \in S_\alpha$, then all servers in S_α 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 S_α.

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, the condition $N2$ implies the condition $N1$. Also, 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. We would also like to note that in condition $N2$, the process p might crash before completing the effective consumption of the responses from the servers in $\mathcal{S}_\alpha$ (if responses are expected). 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, in the case the process p crashes after partial execution of the group-send operation.

4 The $RADON_L$ Algorithm

In this section, we present the $RADON_L$ algorithm, and prove its liveness and atomicity properties for networks that satisfy the stability 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.

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

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

read:
get-data:
  group-send(QUERY-TAG-DATA)
  Wait for responses from a majority
  Select pair, (t_r, v_r), with max tag.
put-data :
  group-send$((t_r, v_r))$
  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 **to writer** w:
  if $status = active$ **then**
    Send t_{loc} to w
get-data-resp **to reader** r:
  if $status = active$ **then**
    Send (t_{loc}, v_{loc}) to r

put-data-resp, **received** (t, v) **from** 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(QUERY-TAG-DATA)
  Wait for responses from a majority
  Select pair, (t_{rep}, v_{rep}), with max tag
  $(t_{loc}, v_{loc}) \leftarrow (t_{rep}, v_{rep})$
  $status \leftarrow active$
init-repair-resp **to server** s':
  if $status = active$ **then**
    Send (t_{loc}, v_{loc}) to s'

The write and read operations are very similar to those in the ABD algorithm [7], 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 [7], 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 (see Fig. 1) to an incoming message only if it is in the active state.

A repair operation is initiated via the action *init-repair*, by an external trigger, at a server which is 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. As mentioned earlier, note that 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 $RADON_L$

Liveness of operations in $RADON_L$ follows immediately if we assume condition $N1$ with $\alpha > \frac{3}{4}$. This is because liveness of operations depends on sufficient number of responses from the servers during the various phases of the any 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 $\alpha > \frac{3}{4}$. Also, note that the liveness of a repair operation is straight forward 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 [30], a restatement of which is presented in Appendix C for convenience. 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 $RADON_L$, 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 a specific repair operation that is concurrent with π_1 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 1, we show that any completed 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 least of the tags stored in any majority of active servers at time T. This property essentially depends on the algorithm, and not on $N1$. A similar property holds for reads or writes as well. 2) We next show (as part of proof of Theorem 3), under the assumption of $N1$, 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. Below the proof of Lemma 1 is provided in Appendix B, that of Theorem 2 is straightforward and that of Theorem 3 appears in C.

Lemma 1. *Let β denote a well-formed execution of $RADON_L$. 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, at time T. Then, if π denotes any completed repair or read operation that is initiated after time T, we have $tag(\pi) \geq \min_{s \in S_m} t_s$. Also, if π denotes any completed write operation that is initiated after time T, we have $tag(\pi) > \min_{s \in S_m} t_s$.*

Theorem 2 (Liveness). *Let γ denote a well-formed execution of $RADON_L$, operating under the condition $N1$ with $\alpha > \frac{3}{4}$. Also, let Π denote the set of all client operations that take place during the execution. Then every operation $\pi \in \Pi$ associated with a non-faulty client completes.*

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

5 The $RADON_S$ Algorithm

The $RADON_L$ algorithm in the last section guarantees safety whenever $N1$ holds with $\alpha > \frac{3}{4}$. In most practical settings, we expect violation of $N1$ to be very rare; but if $N1$ fails to hold, it is possible to have executions of $RADON_L$ algorithm that are not atomic. In this section, we present the $RADON_S$ algorithm having the property that every execution is atomic. However, liveness is guaranteed only the under the stronger network stability condition $N2$ with $\alpha > \frac{3}{4}$. The algorithm has extra phases for both read and write operations, in order to guarantee safety of every execution.

Fig. 2 The protocols for writer, reader, and any server $s \in \mathcal{S}$ in $RADON_S$.

write(v):

get-tag:
group-send(QUERY-TAG)
Wait for responses from a majority (say from $\mathcal{S}_\alpha$)
Select the max tag t^*

put-data:
Create new tag $t_w = (t^*.z + 1, w)$.
group-send$((t_w, v))$
Wait for $\lceil \frac{3n+1}{4} \rceil$ acks

confirm-data:
group-send$((\text{CONFIRM-DATA}, t_w))$
Terminate after acks from majority from among servers in $\mathcal{S}_\alpha$

read:

get-data:
group-send(QUERY-TAG-DATA)
Wait for responses from a majority
Select pair, (t_r, v_r), with max tag.

put-data :
group-send$((t_r, v_r))$
Wait for $\lceil \frac{3n+1}{4} \rceil$ acks (say from $\mathcal{S}_\alpha$)

confirm-data:
group-send$((\text{CONFIRM-DATA}, t_r))$
Wait for acks from a majority of servers from among 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 **to writer** w**:**
if $status = active$ **then**
Send t_{loc} to w

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

put-data-resp, **received** (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,**:**
(received (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 **:**
$status \leftarrow repair$
$(t_{loc}, v_{loc}) \leftarrow (t_0, v_0)$
Set $Seen$ to empty
group-send(QUERY-TAG-DATA)
Wait for responses from a majority.
Select pair,(t_{rep}, v_{rep}), with max tag
$(t_{loc}, v_{loc}) \leftarrow (t_{rep}, v_{rep})$
$status \leftarrow active$

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

The write operation has three phases (see Fig. 2). The first two phases are identical to those of $RADON_S$ during which the writer queries for the highest tag, 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. 2) 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 $RADON_L$, except for the use of the *Seen* variable in the server during the *put-data* phase. The extra third phase is once again simply the *confirm-data* phase, so that the write-back operation resembles a write operation. The repair operation has one phase, and is exactly identical to that of $RADON_L$.

5.1 Analysis of $RADON_S$

We overview the proofs of liveness and atomicity before formal claims. For liveness, 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 $N2$ 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 $RADON_L$. 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 $RADON_L$, we do this in two parts: Lemma 1 holds as it is for $RADON_S$ as well. Recall that Lemma 1 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 $RADON_L$, we used the $N1$ to argue this,

in $RADON_S$, we do not use $N2$; instead we rely on the third *confirm-data* phase of the first write operation π_1. Full proofs of atomicity and liveness appear in Appendices E and D, respectively.

Theorem 4 (Liveness). *Let β denote a well-formed execution of $RADON_S$ operating under condition $N2$ with $\alpha > \frac{3}{4}$. Also, let Π denote the set of all client operations that take place during the execution. Then every operation $\pi \in \Pi$ associated with a non-faulty client completes.*

Theorem 5 (Atomcity). *Every execution of the $RADON_S$ algorithm is atomic.*

6 Algorithm $RADON_C$

In this section, we present the erasure-code based $RADON_C$ algorithm for implementing atomic memory service, and performing repair of crashed nodes. The $RADON_C$ algorithm has significantly reduced stable-storage and communication cost requirements than both $RADON_L$ and $RADON_S$, under scenarios when the number of write operations that are concurrent with a read or repair operation is bounded and limited. Liveness and atomicity are guaranteed under the $N1$ network stability condition with $\alpha \geq \frac{3n+k}{4}$, while using $[n, k]$ MDS codes for storage. A quick background on erasure codes appears next.

Background on Erasure coding: We use an $[n, k]$ linear MDS code [15] 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 a 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 often referred to as the codeword corresponding to the value v. Each coded element c_i also has a size $\frac{1}{k}$. In our scheme we store one coded element per server. We use 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, $1 \leq i \leq n$.

Algorithm Description The algorithm (see Fig. 3) is a natural generalization of the $RADON_L$ 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. Note that the *group-send* here uses a vector of length n, with the i^{th} element denoting the message for the i^{th} server, $1 \leq i \leq n$. Each server keeps a *List* of up to $(\delta + 1)$ of (tag, coded-element) pairs. Every time a (tag, coded-element) message arrives from a writer, the pair gets added to the *List*, and the *List* is pruned to retain 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. The motivation for awaiting for $\lceil \frac{n+k}{2} \rceil$ responses comes from the fact that any two subsets $\mathcal{S}_1$ and $\mathcal{S}_2$ of the n servers, each with cardinality $\lceil \frac{3n+k}{4} \rceil$, have at least $\lceil \frac{n+k}{2} \rceil$ servers in common. This fact is important to prove the atomicity properties of $RADON_C$. Once the reader receives *Lists* from $\lceil \frac{n+k}{2} \rceil$ servers, it selects the highest tag t_r whose corresponding value

[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.

Fig. 3 The protocols for write, reader, and any server $s_i \in \mathcal{S}$ in $RADON_C$.

write(v):

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

put-data:
  Create new tag $t_w = (t^*.z + 1, w)$.
  Create CODED-ELEMENTS $= [(t_w, c_1), \ldots, (t_w, c_n)]$, $c_i = \Phi_i(v)$
  group-send(CODED-ELEMENTS)
  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:
  Create CODED-ELEMENTS $= [(t_r, c_1), \ldots, (t_r, c_n)]$, $c_i = \Phi_i(v_r)$
  group-send(CODED-ELEMENTS)
  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 **to writer** w:
  if *status* $=$ *active* **then**
    Let $t^* = \max_{(t,c) \in List} t$
    Send t^* to w

get-data-resp **to reader** r:
  if *status* $=$ *active* **then**
    Send *List* to r

put-data-resp, **received** (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(QUERY-LIST)
  Wait for $\lceil \frac{n+k}{2} \rceil$ *Lists*
  Find all (tag, value) pairs that are decodable using the received *Lists*.
  Restore local *List* via re-encoding and retaining the (tag, coded-element) pairs corresponding to the (at most) $\delta + 1$ highest tags, from among the above pairs
  status $\leftarrow$ *active*

init-repair-resp **to server** s':
  if *status* $=$ *active* **then**
    Send *List* to s'

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. An important fact to be proved to show liveness of read operations is that there exists a set of at least k *Lists* among the responses, all of which contain coded elements corresponding to a certain common tag, so that the reader is not stuck not being able to return any value.

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 the server is the i^{th} server, the 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. 3). The fact that the server attempts to restore as many (tag, coded-element) pairs as possible, and not just the pair corresponding to the highest tag, is a key idea that ensures liveness of read operations.

Below we formally state the claims regarding liveness and atomicity. The analysis involve developing the equivalent of Lemma 1 first, which will then be used to prove both liveness and atomicity. The entire analysis appears in Appendix F.

Theorem 6 (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 a read or a repair. Also, let Π denote the set of all client operations that take place during the execution. Then every operation $\pi \in \Pi$ associated with a non-faulty client completes.*

Theorem 7 (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 read or a repair is δ.*

7 Conclusions

In this paper, we examine the problem of implementing atomic memory in the presence of failures and repairs, under the asynchronous message passing model. We show the necessity of imposing restrictions on the number of repair operations that are concurrent with write operations, in order to achieve atomicity and liveness of operations, simultaneously. Algorithm independent network stability conditions were introduced, which in our view holds in most practical networks. While $RADON_L$ favors liveness, $RADON_S$ guarantees atomicity of every execution, but demands a slightly stringent network. We further show how erasure codes help improve storage and communication costs, under appropriate scenarios. Ongoing efforts include latency analysis of these algorithms, exploring possibility of using repair-efficient erasure codes [26] in $RADON_C$, and testbed evaluations on cloud based infrastructure.

References

1. Amazon Web Services, "Auto Scaling, Developers Guide," http://docs.aws.amazon.com/AutoScaling/latest/DeveloperGuide/as-scale-based-on-demand.html.
2. "IBM Bluemix," https://console.ng.bluemix.net/catalog/services/auto-scaling.
3. "Rackspace," https://blog.rackspace.com/easily-scale-your-cloud-with-rackspace-auto-scale/.
4. Microsoft, "Microsoft Azure," https://azure.microsoft.com/en-us/documentation/articles/cloud-services-how-to-scale/.
5. Google, "Google Cloud Platform," https://cloud.google.com/compute/docs/instances/setting-instance-scheduling-options.
6. "Docker Engine," https://docs.docker.com/engine/admin/host_integration/.
7. H. Attiya, A. Bar-Noy, and D. Dolev, "Sharing memory robustly in message passing systems," *Journal of the ACM*, vol. 42(1), pp. 124–142, 1996.
8. R. Fan and N. Lynch, "Efficient replication of large data objects," in *Distributed algorithms*, ser. Lecture Notes in Computer Science, 2003, pp. 75–91.
9. 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)*, 2002, pp. 173–190.
10. M. K. Aguilera, I. Keidar, D. Malkhi, and A. Shraer, "Dynamic atomic storage without consensus," *Journal of the ACM*, pp. 7:1–7:32, 2011.
11. L. Barroso and U. Hlzle, *The Datacenter as a Computer:An Introduction to the Design of Warehouse-Scale Machines*. Morgan & Claypool Publishers, 2009.
12. *The Growing Impact of the CAP Theorem*. IEEE Computer Society, feb 2012.
13. 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.*, vol. 26, no. 2, pp. 4:1–4:26, jun 2008.
14. 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*, ser. SOSP '07. New York, NY, USA: ACM, 2007, pp. 205–220.
15. W. C. Huffman and V. Pless, *Fundamentals of error-correcting codes*. Cambridge university press, 2003.
16. A. Spiegelman, Y. Cassuto, G. Chockler, and I. Keidar, "Space Bounds for Reliable Storage: Fundamental Limits of Coding," *ArXiv e-prints, 1507.05169*, Jul. 2015.
17. 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)*, 2005, pp. 336–345.

18. 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.
19. 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.
20. 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.
21. 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.
22. 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*, 2013, pp. 285–298.
23. 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.
24. L. Lamport, "On interprocess communication," *Distributed computing*, vol. 1, no. 2, pp. 86–101, 1986.
25. C. Shao, J. L. Welch, E. Pierce, and H. Lee, "Multiwriter consistency conditions for shared memory registers," *SIAM Journal on Computing*, vol. 40, no. 1, pp. 28–62, 2011.
26. A. G. Dimakis, K. Ramchandran, Y. Wu, and C. Suh, "A survey on network codes for distributed storage," *Proceedings of the IEEE*, vol. 99, no. 3, pp. 476–489, 2011.
27. 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)*, 2012, pp. 15–26.
28. 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*, 2013, pp. 325–336.
29. 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)*, 2015, pp. 81–94.
30. N. A. Lynch, *Distributed Algorithms*. Morgan Kaufmann Publishers, 1996.

Appendix

A Proof of Theorem 1

The lemma is restated for convenience.

Theorem 8. *(***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 every repair operation completes, and takes the repaired server back to the active state.*

Proof. We prove this result by contradiction, by assuming an algorithm A_{alg} that guarantees liveness and atomicity, and also is such that every repair operation completes, and takes the repaired server back to the active state. Let the initial value stored in the system be $v_o \in V$, where V is the domain of all values. Consider a non-faulty writer w, and suppose w initiates a write operation π^w with the value v_1, such that $v_1 \neq v_0$. Let $\mathcal{S}_w \subseteq \mathcal{S}$ be the set of all servers that writer w sends messages before w expects any response from any of the servers in $\mathcal{S}$. Without loss of generality[2] let $\mathcal{S}_w = \{s_1, s_2, \ldots, s_k\}$, for some $k \leq n$, and let m_i denote the message sent by w to server $s_i, 1 \leq i \leq k$. Note that if w sends two or messages to a particular server, say s_1, all these can be combined into m_1, since all these messages are sent without expecting any response.

[2] Clearly, the writer must send a message to at least one server, so we ignore the trivial case when $\mathcal{S}_w$ is empty.

Consider an execution which starts with all the servers in the active state, the operation π^w begins, messages get sent out to servers in $\mathcal{S}_w$. Delay the messages such that message m_1 arrives at server s_1 before any other server in $\mathcal{S}_w$ receives the respective message. Assume that s_1 is in the crashed state when m_1 arrives, so s_1 does not receive m_1. Further assume that all the other servers are in the active state at this point of execution. Let server s_1 undergo a successful repair operation, before any other server in $\mathcal{S}_w$ receives its respective message. Next, consider the case when server s_2 receives the message m_2, and delay the messages to all other servers, and assume that s_2 is in the crashed state when m_2 arrives. The sequence of crash and repair can be carried out in this manner one-by-one for every server in $\mathcal{S}_w$, where all these servers end up losing the writer message, though they get repaired. Now if the algorithm is such that the writer expects a response from any of the servers in $\mathcal{S}$, clearly it will not happen, since no server in $\mathcal{S}$ has received any message from w while the server is in the active state. Thus liveness of write is compromised.

We next consider the case when the writer decides to terminate without expecting any response from any server in $\mathcal{S}$, and show that such a method of guaranteeing liveness results in violation of atomicity. Let us call this execution fragment (as discussed above) with such a write as $\beta^w(v_1)$. After the write π_w completes, a read π_r associated with a non-faulty reader, begins. By liveness of read, and atomicity, the read must return v_1. Let the execution fragment associated with the read be denoted as β^r, so that the overall execution fragment under consideration is $\beta^w(v_1) \circ \beta^r$. Next, consider the execution fragment $\beta^w({v'}_1)$ obtained by replacing v_1 with v_1' such that ${v_1}' \neq v_1$. Since a crash causes a server to lose its entire state, it is clear that to the reader r there is no distinction between the state of the system after $\beta^w(v_1)$, and the state of the system after $\beta^w(v_1')$. In this case, if we consider the execution $\beta^w(v_1') \circ \beta^r$, the read returns v_1 ($\neq v_1'$), since in the execution $\beta^w(v_1) \circ \beta^r$ also, r returned v_1. However it violates atomicity of $\beta^w(v_1') \circ \beta^r$, which completes the proof.

B Proof of Lemma 1

The lemma is restated for convenience.

Lemma 2 (Lemma 1). *Let β denote a well-formed execution of the RADON$_L$ algorithm. Suppose T denotes a point of time in the execution 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 the time T. Also, let t_s denote the value of the local tag at server s, at time T. Then, if π denotes any completed repair or read operation that is initiated after time T, we have $tag(\pi) \geq \min_{s \in S_m} t_s$. Also, if π denotes any completed write operation that is initiated after time T, then we have $tag(\pi) > \min_{s \in S_m} t_s$.*

Proof. We use ρ to denote $\min_{s \in S_m} t_s$. Also, for any state variable $x(s)$ that is stored in server s, we write $x(s)|_T$ to denote the value x at time T. Below, we separately consider the cases when π denotes a successful repair, read and write operations, in this respective order.

(a) π is a successful repair operation: We prove the statement by contradiction, by starting with the assumption that $tag(\pi) < \rho$. Let T_π denote the point of time in the execution β at which the operation π completes. Let Π_R' denote the set of all successful repair operations which start after the time T, but start before T_π, and is such that $\forall \pi' \in \Pi_R'$, we have $tag(\pi') < \rho$. Clearly, $\pi \in \Pi_R'$. Let $\pi^* \in \Pi_R'$ denote the repair operation, which completes first. Note that π^* exists since the set Π_R' is finite. Now, let $\hat{S}$ denote the set of majority servers based on whose responses the operation π^* completed. Clearly, $|\hat{S} \cap S_m| \geq 1$. For any server $s \in \hat{S} \cap S_m$, let T_s denote the

point of time in the execution at which the server s responds to π^* with its local (tag, value) pair. Clearly, the server s must have remained in the active state during the entire interval $[T, T_s]$. This follows because s is active at time T, π^* is the first completed repair operation that started after T, and due to the fact that a server responds to a repair request only if it is in the active state. In this case, we know that[3] $t_{loc}(s)|_{T_s} \geq t_{loc}(s)|_T \geq \rho$ for any s in $\hat{S} \cap S_m$. Therefore, we have $tag(\pi^*) = \max_{s \in \hat{S}} t_{loc}(s)|_{T_s} \geq \max_{s \in \hat{S} \cap S_m} t_{loc}(s)|_{T_s} \geq \rho$, which contradicts the existence of $\pi^* \in \Pi'_R$. From, this we conclude that the set Π'_R must be empty to avoid contradictions, and hence $tag(\pi) \geq \rho$.

(b) π is a successful read operation: We prove this by contradiction by starting with the assumption that $tag(\pi) < \rho$. Let $\hat{S}$ denote the set of majority servers based on whose responses during the *get-data* phase (see Fig. 1), the read operation completed. As in Part a), we know that $|\hat{S} \cap S_m| \geq 1$. In this case, let T_s denote the point of time during the execution at which the server $s \in \hat{S} \cap S_m$ responded to the reader. Next, note that in the *get-data* phase, the reader picks the response with the highest tag. Thus, since we assume that $tag(\pi) < \rho$, it must be true that $t_{loc}(s)|_{T_s} < \rho, s \in \hat{S} \cap S_m$. Since the server $s \in \hat{S} \cap S_m$ is active at time T such that $t_{loc}(s)|_T \geq \rho$, this would imply that server s experienced a crash event after time T, and came back to the *active* state before the time T_s via a successful repair operation ϕ such that $tag(\phi) < \rho$. But then, this contradicts Part a) of the theorem which we proved above, and hence we conclude that $tag(\pi) \geq \rho$.

(c) π is a write operation: Once again we prove via contradiction, by starting with the assumption that $tag(\pi) \leq \rho$. Let $\hat{S}$ denote the set of majority servers based on whose responses during the *get-tag* phase, the writer determined $tag(\pi)$. We know from the algorithm that $tag(\pi)$ is strictly larger than all the tags among the responses from $\hat{S}$. Since $|\hat{S} \cap S_m| \geq 1$, we argue like in Part b), and arrive at a contradiction to Part a).

C Proof of Theorem 3

The theorem is restated here for convenience.

Theorem 9 (Theorem 3). *Every execution of the $RADON_L$ algorithm operating under the $N1$ network stability condition with $\alpha > \frac{3}{4}$, is atomic.*

C.1 Some Preliminaries

Partial Order on read and write operations : Consider any well-formed execution β of $RADON_L$, all of whose invoked read or write operations complete. Let Π_{RW} denote the set of all (completed) read and write operations in β. We first define a partial order ($\prec$) on Π_{RW}. Towards this, recall that for any completed write operation π, we defined $tag(\pi)$ as the tag created by the writer during the *write-put* phase. Also, recall that for any completed read operation π, we define $tag(\pi)$ as the tag corresponding to the value returned by the read. The partial order ($\prec$) in Π_{RW} is defined as follows: For any $\pi, \phi \in \Pi_{RW}$, we say $\pi \prec \phi$ if one of the following holds: (i) $tag(\pi) < tag(\phi)$, or (ii) $tag(\pi) = tag(\phi)$, and π and ϕ are write and read operations, respectively. The proof of atomicity will be based on the following lemma, which is simply a restatement of the sufficiency condition for atomicity presented in [30].

[3] Any read or write operation cannot decrease the local tag that is stored in an active server.

Lemma 3. *Consider any well-formed execution β of the algorithm, such that all the invoked read and the write operations are complete. Now, suppose that all the invoked read and write operations in β can be partially ordered by an ordering $\prec$, so that the following properties are satisfied:*

P1. *The partial order ($\prec$) is consistent with the external order of invocation and responses, i.e., there are no operations π_1 and π_2, such that π_1 completes before π_2 starts, yet $\pi_2 \prec \pi_1$.*
P2. *All operations are totally ordered with respect to the write operations, i.e., if π_1 is a write operation and π_2 is any other operation then either $\pi_1 \prec \pi_2$ or $\pi_2 \prec \pi_1$.*
P3. *Every read operation returns the value of the last write preceding it (with respect to $\prec$), and if no preceding writes is ordered before it, then the read returns the initial value of the object.*

The execution β is atomic.

C.2 Proof of Atomicity under $N1$ with $\alpha > 3/4$

We need to prove the properties $P1$, $P2$ and $P3$ of Lemma 3. We do this under $N1$ with $\alpha > 3/4$, using Lemma 1. Let ϕ and π denote two operations in Π_{RW} such that ϕ completed before π started. Also, let c_ϕ and c_π denote the clients that initiated the operations ϕ and π, respectively.

Property $P1$ We want to show that $\pi \not\prec \phi$. We show this in detail only for the case when ϕ and π are both write operations. The proofs of other three cases[4] are similar, and hence omitted. By virtue of the definition of the partial order $\prec$, it is enough to prove that $tag(\pi) > tag(\phi)$. Consider the *put-data* phase of ϕ, where the writer sends the pair (t_w, v) to all servers via the *group-send* operation. Under the condition $N1$ with $\alpha > 3/4$, we know that there exists a set $S_\alpha \subseteq S$ of $\lceil n\alpha \rceil \geq \lceil \frac{3n+1}{4} \rceil$ servers all of which 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 during the execution where all of the servers in S_α complete effective consumption (including sending ack to the writer c_ϕ) of the message (t_w, v). Also, let $S' \subseteq S$ denote the set of $\lceil \frac{3n+1}{4} \rceil$ servers whose acks are used by the writer to decide the completion of the write operation. Clearly, $|S' \cap S_\alpha| > \frac{n}{2}$. Let T denote the earliest point of time during the execution when all servers in $S' \cap S_\alpha$ complete their respective effective consumption of the message (t_w, v). In this case note that *a*) T occurs before the point of completion of the write operation, *b*) all servers in $S' \cap S_\alpha$ are in the active state at T, and *c*) $t_{loc}(s)|_T \geq tag(\phi), \forall s \in S' \cap S_\alpha$. We now apply Lemma 1 to conclude that $tag(\pi) > tag(\phi)$.

Property $P2$ This follows from the construction of tags, and the definition of the partial order ($\prec$).

Property $P3$ This follows from the definition of tags, and by noting that value returned by a read operation π is simply the value associated with $tag(\pi)$.

D Proof of Theorem 4

The theorem is restated for convenience.

[4] These correspond to the case when ϕ and π are both read operations, and the cases where one of them is a write and the other is a read.

Theorem 10. *(Theorem 4) Let β denote a well-formed execution of $RADON_S$ operating under condition $N2$ with $\alpha > \frac{3}{4}$. Also, let Π denote the set of all client operations that take place during the execution. Then every operation $\pi \in \Pi$ associated with a non-faulty client completes.*

We will prove that a write operation associated with a non-faulty client always completes, the proof for a read is similar and hence is omitted. The main step is to show the completion of the $confirm - data$ phase. Consider the $put - data$ phase, and note that under $N2$ with $\alpha > \frac{3}{4}$, we are guaranteed that there exists of set of $\mathcal{S}_\alpha \subset \mathcal{S}$ severs, such that 1) $|\mathcal{S}_\alpha| \geq \lceil \frac{3n+1}{4} \rceil$, and 2) every server in $\mathcal{S}_\alpha$ remain active from the point of time T_1 of initiation of the group-send operation of $put - data$ phase till the point of time T_1', when the writer effectively consumes all responses (acks) from the servers in $\mathcal{S}_\alpha$. Next, let $\mathcal{S}_1 \subset \mathcal{S}$ denote the set of $\lceil \frac{3n+1}{4} \rceil$ whose acks are received by the writer before moving on to the $confirm - data$ phase. First of all note that the existence of the set $\mathcal{S}_1$ is clearly guaranteed under $N2$ with $\alpha > \frac{3}{4}$ (since the set $\mathcal{S}_\alpha$ is a candidate for $\mathcal{S}_1$). Secondly, we note that the group-send operation in the $confirm - data$ phase forms part of the effective consumption of the last ack that is received from the servers in $\mathcal{S}_1$. This follows from the definition of effective-consumption, and by noting the execution of the group-send operation in the $confirm - data$ phase does not depend on any more input after all the acks in the $put - data$ phase are received. Let T_2 denote the point of time at which the group-send operation in the $confirm - data$ phase gets initiated. Note that $T_1' \geq T_2$, in fact if $\mathcal{S}_1 \neq \mathcal{S}_\alpha$, we have[5] $T_1' > T_2$. Next we apply the network condition to the group-send operation in the $confirm - data$ phase. From the $N1$ part of $N2$, we know that there exists a $\mathcal{S}_\alpha'$ of $\lceil \frac{3n+1}{4} \rceil$ servers, all of which receive and effectively consume the message from the group-send operation, and remain active from T_1' till the point of time T_2' when the last of the servers in $\mathcal{S}_\alpha'$ completes effective consumption. Now if we let $\mathcal{S}_\gamma = \mathcal{S}_\alpha \cap \mathcal{S}_\alpha'$, observe that 1) $|\mathcal{S}_\gamma| > \frac{n}{2}$, and 2) all the servers in $\mathcal{S}_\gamma$ remain active from T_1 till T_2'. The second part follows from our earlier observation that $T_1' \geq T_2$. In this case, we infer that all the servers in $\mathcal{S}_\gamma$ does indeed acknowledge back to writer as part of their effective consumption of the $confirm - data$ message, and since $\mathcal{S}_\gamma \subset \mathcal{S}_\alpha$ is at least a majority, we conclude that the write operation associated with the non faulty writer eventually completes.

E Proof of Theorem 5

Theorem 11. *(Theorem 5) Every execution of the $RADON_S$ algorithm is atomic.*

E.1 Some Preliminaries

The proof is based on Lemma 3, and the equivalent of Lemma 1 for $RADON_S$, which we state below for the sake of completion:

[5] In this case, some of the acks from the servers in $\mathcal{S}_\alpha$ get effectively consumed only after the required number $\lceil \frac{3n+1}{4} \rceil$ have already been consumed, the last of which includes execution of the group-send operation of the $confirm - data$ phase. We note that the effective consumption of these additional acks from servers in $\mathcal{S}_\alpha$ is the operation where server simply ignores these, which is not explicitly mentioned in the algorithm. We also note that the notion of atomicity of any sequences of effective consumptions that are local to a server, is implicitly used when we argue that $T_1' > T_2$. By this we mean that if a server receives a message m_1 before m_2, the effective consumption of message m_1 is assumed to be entirely completed before the effective consumption of the message m_2 starts.

Lemma 4. *Let β denote a well-formed execution of $RADON_S$. 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, at time T. Then, if π denotes any completed repair or read operation that is initiated after time T, we have $tag(\pi) \geq \min_{s \in S_m} t_s$. Also, if π denotes any completed write operation that is initiated after time T, we have $tag(\pi) > \min_{s \in S_m} t_s$.*

Proof. Similar to the proof of Lemma 1.

Next, in order to apply Lemma 3, consider any well-formed execution β of $RADON_S$, all of whose invoked read and write operations, denoted by the set Π_{RW}, complete. Recall the discussion in Section 5, where we noted that tags for completed operations in $RADON_S$ are defined exactly as we had done for $RADON_L$. Thus, for any completed write operation π, we define $tag(\pi)$ as the tag created by the writer during the *write-put* phase. For any completed read operation π, we define $tag(\pi)$ as the tag corresponding to the value returned by the read. Further, we define a partial order $(\prec)$ on Π_{RW} like in the proof of Theorem 3 for case of $RADON_L$. These are restated for the sake of completion: For any $\pi, \phi \in \Pi_{RW}$, we say $\pi \prec \phi$ if one of the following holds: (i) $tag(\pi) < tag(\phi)$, or (ii) $tag(\pi) = tag(\phi)$, and π and ϕ are write and read operations, respectively.

E.2 Proof of Atomicity

*Property P*1 Consider two successful operations ϕ and π such that ϕ completes before π begins. We want to prove that $\pi \not\prec \phi$. Consider the case when both ϕ and π are write operations (the other cases are similar, so only one case is discussed). By virtue of the definition of the partial order $(\prec)$, it is enough to prove that $tag(\pi) > tag(\phi)$. Let $\mathcal{S}_\alpha$ and $\mathcal{S}_1$ respectively denote the set of servers whose responses were used by the writer during the $put - data$ and $confirm - data$ phases of ϕ. Let T denote the time of initiation of the *confirm-data* phase of ϕ. From the algorithm (see Fig. 2), we know that $\mathcal{S}_1 \subset \mathcal{S}_\alpha$. Further, based on the discussion in Section 5, it must be true that all servers in $\mathcal{S}_1$ (which is a majority) are active at time T, such that $t_{loc}(s)|_T \geq tag(\phi)$. In this case, we apply Lemma 4 to conclude that $tag(\pi) > tag(\phi)$.

*Property P*2 This follows from the construction of tags, and the definition of the partial order $(\prec)$.

*Property P*3 This follows from the definition of tags, and by noting that value returned by a read operation π is simply the value associated with $tag(\pi)$.

F Analysis of $RADON_C$ Algorithm

We first state three lemmas which together form the equivalent of Lemma 1, for the $RADON_C$. We will argue about the liveness and atomicity properties of $RADON_C$ based on these lemmas, in subsections F.1 and F.2, respectively. The lemmas will themselves be proved subsequently in subsections F.3, F.4, and F.5, respectively. Tags for completed read and write operations are defined in the same manner as we did for both $RADON_L$ and $RADON_S$; we avoid repeating them here.

We wish to clarify a technical point about reads and repairs, before stating the lemmas. Note from the algorithm in Fig. 3 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. Having said that, the first lemma below shows that under $N1$ with $\alpha \geq \frac{3n+k}{4}$, the repaired *List* is never in fact empty; there is always at least one (tag, coded-element) pair in the repaired *List*. The possibility of allowing a trivial completion of repair in the algorithm, allows us to first identify a point of time of completion of the repair operation, and then argue about the fact that the repaired *List* is not empty (see proof of Lemma 5).

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. We show in the second lemma below that the read never gets stuck under $N1$ with $\alpha \geq \frac{3n+k}{4n}$.

We know that in the $RADON_C$ algorithm, the initial value of $List(s_i) = (t_0, \Phi_i(v_0)), \forall s_i \in \mathcal{S}$. We associate this initial value with a completed write operation π_{init}. In other words π_{init} is a write operation which completes at the point of time when the execution starts.

Number of Writes Concurrent with a Read or a Repair: Liveness and atomicity of $RADON_C$ will be proved only under the assumptions that 1) the number of write operations concurrent with a read operation, associated with a non-faulty reader, is at most δ, and 2) the number of write operations concurrent with any completed repair operation is at most δ. We note that concurrent write operations also include all the incomplete write operations initiated by writers, which failed after initiation, as long the write was initiated before the start of the read or repair.

Lemma 5 (Lemma 1 equivalent for repairs). *Consider any well-formed execution β of $RADON_C$ operating under the network stability condition $N1$ with $\alpha \geq \frac{3n+k}{4n}$. Consider any repair operation π in β, on a server $s \in \mathcal{S}$, such that π completes, and thus takes server s back to the active state. Let $\Sigma = \{\sigma : \sigma$ is a read or a write in β that completes before π begins$\}$, and also let $\sigma^* = \arg\max_{\sigma \in \Sigma} tag(\sigma)$. In this case, the repaired List of server s due to repair operation π contains the pair $(tag(\sigma^*), c_s^*)$, as long as the number of write operations concurrent with π is at most δ. Here, c_s^* denotes the coded-element of server s corresponding to the value v^*, associated with $tag(\sigma^*)$.*

Lemma 6 (Lemma 1 equivalent for reads). *Consider any well-formed execution β of $RADON_C$ operating under the network stability condition $N1$ with $\alpha \geq \frac{3n+k}{4n}$. Consider any read operation π associated with a non-faulty reader r, and let $\mathcal{S}_1$ denote the set of $\lceil \frac{n+k}{2} \rceil$ servers whose responses (which are lists), say $\{L_\pi(s), s \in \mathcal{S}_1\}$, are used by r, to attempt decoding of a value in the $get - data$ phase. Let $\Sigma = \{\sigma : \sigma$ 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, as long as the number of write operations concurrent with π is at most δ, there exists $\mathcal{S}_2 \subseteq S_1$, $|S_2| = k$, such that $\forall s \in S_2, (tag(\sigma^*), c_s^*) \in L_\pi(s)$.*

Lemma 7 (Lemma 1 equivalent for writes). *Consider any well-formed execution β of $RADON_C$ operating under the network stability condition $N1$ with $\alpha \geq \frac{3n+k}{4n}$. Consider any write operation π associated with a non-faulty writer w, and let $\mathcal{S}_1$ denote the set of majority servers whose responses are used by w, to compute max-tag in the $get - tag$ phase. Let $\Sigma = \{\sigma : \sigma$ 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, there exists a server $s \in \mathcal{S}_1$, whose response tag $t_s \geq tag(\sigma^*)$.*

We note that in all the lemmas above, we have $\pi_{init} \in \Sigma$. The write operation π_{init} was introduced only to avoid the trivial case where the set Σ becomes empty.

F.1 Liveness: Proof of Theorem 6

The theorem is restated below for easy reference:

Theorem 12. *(Theorem 6) Let β denote a well-formed execution of $RADON_C$, operating under the network stability condition $N1$ with $\alpha \geq \frac{3n+k}{4n}$ and δ be the maximum number of write operations concurrent with a read or a repair. Also, let Π denote the set of all client operations that take place during the execution. Then every operation $\pi \in \Pi$ associated with a non-faulty client completes.*

Proof. Liveness of writes depends only on sufficient number of responses in the two phases. The maximum number of responses expected in either of the two phases is $\frac{3n+k}{4n}$, which we know is guaranteed under $N1$ with $\alpha \geq \frac{3n+k}{4n}$. Liveness of reads follow directly from Lemma 6 (for decodability of a value), and liveness of writes (for the write-back phase).

F.2 Atomicity: Proof of Theorem 7

The theorem is restated first:

Theorem 13. *(Theorem 7) 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 read or a repair is δ.*

Proof. The proof is based on Lemmas 3, 6 and 7. In order to apply Lemma 3, consider any well-formed execution β of $RADON_C$, all of whose invoked read and write operations, denoted by the set Π_{RW}, complete. We define a partial order ($\prec$) on Π_{RW} like in the proof of Theorem 3 for case of $RADON_L$. To prove Property $P1$ of Lemma 3, consider two successful operations ϕ and π such that ϕ completes before π begins. Firstly, consider the case π is a write, and ϕ is either a read or write. We need to show that $tag(\pi) > tag(\phi)$, which we note follows directly from Lemma 7. Next, consider the case when consider the case π is a read, and ϕ is either a read or write. We need to show that $tag(\pi) \geq tag(\phi)$, which we note follows directly from Lemma 6. This completes the proof of Property $P1$. Proofs of Properties $P2$ and $P3$ are similar to those of the corresponding properties in Theorem 3, where we proved atomicity of $RADON_L$.

F.3 Proof of Lemma 5

Consider the set Σ and the operation σ^* as defined in the statement of Lemma 5. Without loss of generality, let us assume that σ^* is a write operation. Since we assume condition $N1$ with $\alpha \geq \frac{3n+k}{4n}$, there exists a set $\mathcal{S}_\alpha$ of $\lceil \frac{3n+k}{n} \rceil$ servers that respects $N1$ for the group-send operation (say gp*) in the $put-data$ phase of σ^*. If $\mathcal{S}_1$ denotes the set of $\lceil \frac{3n+k}{n} \rceil$ servers, whose responses are used by the writer to decide termination, we then know that 1) $|\mathcal{S}_\alpha \cap \mathcal{S}_1| \geq \lceil \frac{n+k}{2} \rceil$, and 2) if T_{prop} denotes the earliest point of time during the execution when all the servers in $\mathcal{S}_{prop} = \mathcal{S}_\alpha \cap \mathcal{S}_1$ complete effective consumption of their respective messages from the group-send operation gp*, then every server in $\mathcal{S}_{prop}$ remains active at T_{prop}, and has not experienced a crash after its effective consumption, until T_{prop}. Our goal is to show that the repair operation π always receives at least

k responses from among the servers in $\mathcal{S}_{prop}$, and must be able to decode (and then re-encode) the value corresponding to $tag(\sigma^*)$. Below we consider the effects of concurrent writes having higher tags, and ongoing/future repairs (before π) starts, both of which can potentially remove coded elements corresponding to $tag(\sigma^*)$, from lists of various servers. We show under the assumptions of the lemma, that neither of these cause a problem.

Let us first consider the effect of concurrent writes. Towards this, if we consider $List(s)|_{T_{prop}}, s \in \mathcal{S}_{prop}$, we know based on the write part of $RADON_C$ algorithm (see Fig. 3) that the this list will contain the pair corresponding to $tag(\sigma^*)$, unless there are "many concurrent writes" which can eliminate this pair, since we only keep $\delta + 1$ pairs, corresponding to the highest tags. To examine if this can happen, let us define the set $\Lambda = \{\lambda :$ write operation which starts before completion of π, such that $tag(\lambda) > tag(\sigma^*)\}$. Observe that any $\lambda \in \Lambda$ cannot complete before the start of π, since this will contradict the maximality of σ^* in Σ. In other words, any $\lambda \in \Lambda$ is concurrent with the repair operation π. By assumption, there are at most δ writes that can be concurrent with a repair, and thus $|\Lambda| \leq \delta$. Thus it is clear that if a server $s \in \mathcal{S}_{prop}$ does not crash in the interval $[T_{prop}\ \ T]$, then $List(s)|_T$ contains the pair corresponding to $tag(\sigma^*)$, for any T such that $T_{prop} < T \leq T_{end}(\pi)$. Here $T_{end}(\pi)$ denotes the point of completion of π.

Let us next consider the effect of repairs, let $\widetilde{\Pi} = \{\widetilde{\pi} :$ a repair which start after T_{prop}, but also start before the completion of $\pi\}$. Clearly, $\pi \in \widetilde{\Pi}$. Let $\widetilde{\pi}^* \in \widetilde{\Pi}$ denote the repair operation which completes first. Clearly, it must be true that $T_{prop} < T_{end}(\widetilde{\pi}^*) \leq T_{end}(\pi)$. We prove the Lemma 5 for $\widetilde{\pi}^*$ first. Using this result, we prove the lemma for the repair operation in $\widetilde{\Pi}$ which completes second. We continue in an inductive manner (on the finite set $\widetilde{\Pi}$), until we hit π. Towards proving the lemma for $\widetilde{\pi}^*$, consider the group-send operation, where $\widetilde{\pi}^*$ requests for local $Lists$ from all servers. Let $\mathcal{S}_\theta \subset \mathcal{S}_{prop}$ denote the servers among $\mathcal{S}_{prop}$ which are not in the active state when the repair request arrives. Also, let $\mathcal{S}_a \subset \mathcal{S}$ denote the set of all servers which are in the active state when the repair request arrives. Clearly, $|\mathcal{S}_a| \leq n - |\mathcal{S}_\theta|$. Next, let $\mathcal{S}_{ack} \subset \mathcal{S}_a$ denote the set of $\lceil \frac{n+k}{2} \rceil$ servers based on whose responses the repair operation $\widetilde{\pi}^*$ completes. Now, since $\mathcal{S}_{prop} \backslash \mathcal{S}_\theta \subset \mathcal{S}_a$, we have

$$(\mathcal{S}_{prop} \backslash \mathcal{S}_\theta) \cup \mathcal{S}_{ack} \subset \mathcal{S}_a \tag{1}$$

$$\implies |\mathcal{S}_{prop} \backslash \mathcal{S}_\theta| + |\mathcal{S}_{ack}| - |(\mathcal{S}_{prop} \backslash \mathcal{S}_\theta) \cap \mathcal{S}_{ack}| \leq \mathcal{S}_a \tag{2}$$

$$\implies |\mathcal{S}_{prop}| - |\mathcal{S}_\theta| + \lceil \frac{n+k}{2} \rceil - |(\mathcal{S}_{prop} \backslash \mathcal{S}_\theta) \cap \mathcal{S}_{ack}| \leq n - |\mathcal{S}_\theta| \tag{3}$$

$$\implies |(\mathcal{S}_{prop} \backslash \mathcal{S}_\theta) \cap \mathcal{S}_{ack}| \geq k, \tag{4}$$

where the last inequality follows from our earlier observation that $|\mathcal{S}_{prop}| \geq \lceil \frac{n+k}{2} \rceil$. Next, note that any server s in $(\mathcal{S}_{prop} \backslash \mathcal{S}_\theta) \cap \mathcal{S}_{ack}$ remains active from T_{prop} until the point when s responds to the repair request from $\widetilde{\pi}^*$. This follows because of the facts that 1) s is active at T_{prop}, 2) a server responds to a repair request only if it is in the active state, and 3) since $\widetilde{\pi}^*$ is the first repair operation that completes after T_{prop}. Also, recall our earlier observations that 1) if a server $s \in \mathcal{S}_{prop}$ does not crash in the interval $[T_{prop}\ \ T]$, then $List(s)|_T$ contains the pair corresponding to $tag(\sigma^*)$, for any T such that $T_{prop} < T \leq T_{end}(\pi)$, and 2) $T_{prop} < T_{start}(\widetilde{\pi}^*) < T_{end}(\widetilde{\pi}^*) \leq T_{end}(\pi)$. In this case, we know that the responses of all the servers in $(\mathcal{S}_{prop} \backslash \mathcal{S}_\theta) \cap \mathcal{S}_{ack}$ to $\widetilde{\pi}^*$, contain the pair corresponding to $tag(\sigma^*)$. From (4), it follows that the repaired list for $\widetilde{\pi}^*$, before pruning to $(\delta + 1)$ entries, contains the pair corresponding to $tag(\sigma^*)$. Finally the fact that $tag(\sigma^*)$ is among the highest $\delta + 1$ tags, and hence part of the pruned list, follows from our earlier observations that 1)

$|\Lambda| \leq \delta$, and 2) $T_{end}(\widetilde{\pi}^*) \leq T_{end}(\pi)$. This completes our proof of Lemma 5 for the repair operation $\widetilde{\pi}^*$.

We next prove the lemma for the repair operation $\pi_2 \in \widetilde{\Pi}$, which completes second. The proof is mostly identical, and we will only highlight the place where we use the result on $\widetilde{\pi}^*$. Clearly, since we carry out the induction only until we hit π, it must be true that $T_{prop} < T_{end}(\pi_2) \leq T_{end}(\pi)$. Consider the group-send operation, where π_2 requests for local *Lists* from all servers. Let $\mathcal{S}_\theta^{(2)} \subset \mathcal{S}_{prop}$ denote the servers among $\mathcal{S}_{prop}$ which are not in the active state when the repair request arrives. Also, let $\mathcal{S}_a^{(2)} \subset \mathcal{S}$ denote the set of all servers which are in the active state when the repair request arrives. As before, $|\mathcal{S}_a^{(2)}| \leq n - |\mathcal{S}_\theta^{(2)}|$. Next, let $\mathcal{S}_{ack}^{(2)} \subset \mathcal{S}_a^{(2)}$ denote the set of $\lceil \frac{n+k}{2} \rceil$ servers based on whose responses the repair operation π_2 completes. Along the lines of (1)-(4), one can show that $|(\mathcal{S}_{prop} \backslash \mathcal{S}_\theta^{(2)}) \cap \mathcal{S}_{ack}^{(2)}| \geq k$. Next, if we consider the set $(\mathcal{S}_{prop} \backslash \mathcal{S}_\theta^{(2)}) \cap \mathcal{S}_{ack}^{(2)}$, at most one of the servers in this set would have undergone a crash after the time T_{prop}, and got repaired before the time the server responded to π_2. Note that more than one repair operation on $(\mathcal{S}_{prop} \backslash \mathcal{S}_\theta^{(2)}) \cap \mathcal{S}_{ack}^{(2)}$ cannot happen, since this will contradict the assumption that π_2 is the second repair operation to complete after T_{prop}. Further, if one repair operation among a server in $(\mathcal{S}_{prop} \backslash \mathcal{S}_\theta^{(2)}) \cap \mathcal{S}_{ack}^{(2)}$ has indeed occurred, this must be the operation $\widetilde{\pi}^*$ which we considered above. Further, we know that the repaired *List* due to $\widetilde{\pi}^*$ contains the pair corresponding to $tag(\sigma^*)$. In other words, irrespective of whether one repair operation occurred among the servers in $(\mathcal{S}_{prop} \backslash \mathcal{S}_\theta^{(2)}) \cap \mathcal{S}_{ack}^{(2)}$, or not, the responses of all the servers in $(\mathcal{S}_{prop} \backslash \mathcal{S}_\theta^{(2)}) \cap \mathcal{S}_{ack}^{(2)}$ contain the pair corresponding to $tag(\sigma^*)$. The rest of the proof is similar to that of $\widetilde{\pi}^*$, where we argue that the pruned list after repair contains the pair corresponding to $tag(\sigma^*)$. The rest of the induction is similar, and this completes the proof of Lemma 5.

F.4 Proof of Lemma 6

The proof follows mostly along the lines of proof of Lemma 5. We will only highlight the main steps here. Consider the set Σ and the operation σ^* as defined in the statement of Lemma 6. Without loss of generality, let us assume that σ^* is a write operation. Also, define the time T_{prop} and the set $\mathcal{S}_{prop}$ exactly in the same way as what we defined in the proof of Lemma 5. Let T_1 denote the earliest point of time during the execution when the reader receives responses from all the servers in $\mathcal{S}_1$, where $\mathcal{S}_1$ is as defined in the statement of this lemma. Define $\Lambda = \{\lambda :$ write operation which starts before time T_1 such that $tag(\lambda) > tag(\sigma^*)\}$. Clearly, any write $\lambda \in \Lambda$ does not complete before the start of π, since this contradicts the maximality of σ^*. In this case, from the assumption on concurrency in the lemma statement, we get that $|\Lambda| \leq \delta$. We claim then that if we consider a server $s \in \mathcal{S}_{prop}$ which remains active at a time T such that $T_{prop} < T < T_1$, then $List(s)|_T$ contains the pair corresponding to $tag(\sigma^*)$. The last statement is clearly true, if server $s \in \mathcal{S}_{prop}$ does not crash (and undergo repair) during the interval $[T_{prop}\ \ T]$, because of the facts that 1) s is active at T_{prop}, 2) $List(s)|_{T_{prop}}$ contains the pair corresponding to $tag(\sigma^*)$, and 3) $|\Lambda| \leq \delta$. Now, if server $s \in \mathcal{S}_{prop}$ undergoes a crash and repair operation (say ρ) during the interval $[T_{prop}\ \ T]$ (so that it is active again at T), we can argue exactly like in the proof of 5 and show that the repaired *List* due to ρ contains the pair corresponding to $tag(\sigma^*)$. This can be done by considering the set $\widetilde{\Pi} = \{\widetilde{\pi} :$ a repair which start after T_{prop}, but also start before $T_1\}$, and applying induction on $\widetilde{\Pi}$ based on the order of completion times of the repair operations. The completes the proof of our claim about $List(s)|_T$.

The rest of the proof follows simply by noting $|\mathcal{S}_1 \cap \mathcal{S}_{prop}| \geq k$, and thus the value corresponding to $tag(\sigma^*)$ is surely a candidate for decoding, since we know that an $[n, k]$ linear MDS code can be uniquely decoded given any k out of the n coded-elements.

Note 1. We remark that we cannot directly apply the result from Lemma 5, to prove Lemma 6. Rather, we repeat a lot of steps that appear in the proof of Lemma 5, here as well. This was not the case with $RADON_L$, where to prove the claim reading read-tags in Lemma 1, we simply relied on the claim regarding repair-tags from the same lemma. The difference in the two cases comes because of the fact that for a direct equivalent of Lemma 1 here, require introducing notation similar to T_{prop} into the statement of Lemma 5, which was intentionally avoided to keep the statement simple.

F.5 Proof of Lemma 7

Similar to that of Lemma 6, and hence omitted.

G Storage and Communication Costs of Algorithms

We give a quick justification of the storage and communication cost numbers of the three algorithms, appear in Table 1. These values are written under the assumption that the size of the object value v is 1 unit. Further, we completely neglect storage and communication costs due to meta data like tags, variables like $Seen$, process ID, etc. The storage cost corresponds to the worst case total storage across all the n servers. Like wise, communication cost associated with a read or write operation is the (worst case) size of the total data that gets transmitted in the messages sent as part of the operation.

Under these definitions, it is clear that both $RADON_L$ and $RADON_S$ 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, since the reader gets entire $Lists$ during the $get - data$ phase, this 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 $(\delta + 2)\frac{n}{k}$.