# Simulating a Shared Register in an Asynchronous System that Never Stops Changing

Hagit Attiya, Hyun Chul Chung, Faith Ellen, Saptaparni Kumar, Jennifer L. Welch

# Simulating a Shared Register in an Asynchronous System that Never Stops Changing
## (Extended Abstract)

Hagit Attiya[1], Hyun Chul Chung[2,*], Faith Ellen[3],
Saptaparni Kumar[2], and Jennifer L. Welch[2]

[1] Department of Computer Science, Technion
[2] Department of Computer Science and Engineering, Texas A&M University
[3] Department of Computer Science, University of Toronto

**Abstract.** Simulating a shared register can mask the intricacies of designing algorithms for asynchronous message-passing systems subject to crash failures, since it allows them to run algorithms designed for the simpler shared-memory model. The simulation replicates the value of the register in multiple servers and requires readers and writers to communicate with a majority of servers. The success of this approach for static systems, where the set of nodes (readers, writers, and servers) is fixed, has motivated several similar simulations for dynamic systems, where nodes may enter and leave. However, all existing simulations need to assume that the system eventually stops changing for a long enough period or that the system size is fixed.

This paper presents the first simulation of an atomic read/write register in a crash-prone asynchronous system that can change size and withstand nodes continually entering and leaving. The simulation allows the system to keep changing, provided that the number of nodes entering and leaving during a fixed time interval is at most a constant fraction of the current system size.

## 1 Introduction

Simulating a shared read/write register can mask the intricacies of designing algorithms for asynchronous message-passing systems subject to crash failures, since it allows them to run algorithms designed for the simpler shared-memory model. The ABD simulation [5] replicates the value of the register in server nodes. It assumes that a majority of the server nodes do not fail. Consider the simplified case of a single writer and a single reader. To write the value $v$, the writer sends $v$, tagged with a sequence number, to all servers and waits for acknowledgements from a majority of them. Similarly, to read, the reader contacts all servers, waits to receive values from a majority of them and then, returns the value with the highest sequence number. This approach can be extended to the case of multiple writers and multiple readers by having each operation consist of a read phase, used by a writer to determine its sequence number and used by a reader to obtain the return value, followed by a write phase, used by a writer to disseminate the

[*] Hyun Chul Chung is currently working at Epoch Labs, Inc. Austin, TX USA.

value (and sequence number) and used by a reader to announce the sequence number of the value it is about to return [16].

The success of this approach for static systems, where the set of readers, writers, and servers is fixed, has motivated several similar simulations for dynamic systems, where nodes may enter and leave, a phenomenon called *churn*. (See [21] for a survey.) However, existing simulations rely either on the assumption that churn eventually stops for a long enough period (e.g., [2, 7]) or on the assumption that the system size never changes (e.g., [6]).

In this paper, we take a different approach: *we allow churn to continue forever, while still ensuring that read and write operations complete and nodes can join the system.* Our churn model puts an upper bound on the number of nodes that can enter or leave during any time interval of a certain length. The upper bound is a constant fraction of the number of nodes that are present in the system at the beginning of the time interval. So, as the system size grows, the allowable number of changes to its composition grows as well. Similarly, as the system size shrinks, the allowable number of changes shrinks.

The time interval with respect to which the churn is bounded is set as the maximum message delay. We assume an *unknown* upper bound $D$ on the delay of any message (between nonfaulty nodes). Our churn model is that, in any time interval of length $D$, the number of nodes that can enter or leave in the interval is at most a constant fraction $\alpha$ of the number of nodes in the system at the beginning of the interval. It is important to note that we set no lower bound on the delay of messages, so consensus cannot be solved in this model even in the static case with no nodes entering or leaving but the possibility of one node crashing.

We believe ours is a reasonable churn model. For instance, if each node has the same probability of leaving in a time interval, then the number of leaves is expected to be a fixed fraction of the total number of nodes. (See [15] for a discussion of churn behavior in practice.)

Our algorithm, called CCREG (for *Continuous Churn Register*), combines the simple static algorithm for multiple readers and multiple writers outlined above with a joining protocol and careful estimations of the number of nodes from which responses should be received for joining, reading, and writing. In order to join, a newly entered node announces its entry and waits to receive sufficiently many acknowledgements. Then it joins as a participating node and announces that it has done so. A node leaves the system by announcing its departure. Each node maintains a set of changes to the composition of the system, based on the announcements of nodes entering, joining and leaving. This information is also propagated through appropriate echo messages and by having each node append its changes set to its messages that echo enter announcements.

A joining node calculates the number of acknowledgements it needs as a fraction (depending on $\alpha$) of the number of nodes it believes are in the system when it first receives an acknowledgement from a node that has already joined. Then it subtracts $f$, the maximum number of crashes. This number must be large enough to ensure that at least one acknowledgement is from a node $p$ that has been in the system sufficiently long, so that $p$ has up-to-date information. This ensures that information about the system composition is propagated properly. The number of necessary acknowledgements must also be small enough to ensure that the node will eventually receive enough of them.

Each reader and writer keeps track of the number of servers that have joined, but not left. We call these *members*. The read and write phases of operations wait for responses from a fixed fraction of the servers believed to be members, plus $f/2$. As in the joining protocol, this number of responses must be small enough so that termination is guaranteed. To prove CCREG is linearizable, we consider two cases: If a read occurs close to a write, then we must ensure that the sets of servers contacted by the two operations are intersecting. This is analogous to the situation in the static, majority algorithm. If operations are farther apart in time, then, as in the join protocol, we ensure that information about writes to the register is propagated properly.

Our churn model has the pleasing property that it is algorithm-independent: It only refers to nodes that enter or leave and ignores whether they complete the join protocol.

**Related Work**: A simple simulation of a single-writer, multi-reader register in a static network was presented in [5]. It was followed by extensions that, for example, reduce complexity [4,10,13,14], support multiple writers [16], or tolerate Byzantine failures [1, 3, 18, 20]. To optimize load and resilience, the simple majority quorums used in these papers can be replaced by other, more complicated, quorum systems (e.g., [19, 24]).

RAMBO [17] was the first simulation of a multi-writer, multi-reader register in a dynamic system, where nodes may enter and leave. It includes a dedicated reconfiguration module for handling configuration changes and for installing a new quorum system. This module relies on eventually-terminating consensus. As long as the consensus does not terminate, the protocol communicates with quorums from a possibly large number of different configurations. This assumption is also made in other variants of RAMBO (e.g., [8, 9, 11, 12]). These papers assume that churn eventually stops.

DynaStore [2] simulates a multi-writer, multi-reader register in a dynamic system, by reconfiguring the servers without using consensus. Dynastore and its variant [22] also assume that churn eventually stops.

One simulation whose model has a similar flavor to ours is [6], in that at most a *fixed* fraction of nodes enter and leave periodically and there is an unknown upper bound on message delay. However, in their model, the system size is assumed to be constant (and known to the nodes), i.e., the number of nodes entering is the same as the number of nodes leaving at each point in time. Our model is more general, as we do not require that the system size is always the same. Instead, in our model, the system can grow, shrink, or alternately grow and shrink.

Baldoni et al. [6] also prove that it is impossible to simulate a register when there is no upper bound on message delay. Their proof works by considering scenarios in which at least half of the nodes fail or leave. Then they invoke the lower bound in [5], which shows that simulating a register is impossible unless fewer than half the nodes are faulty. Their proof can be adapted to hold when there is an unknown upper bound, $D$, on message delay and half the nodes can be replaced during any time interval of length $D$, provided that nodes are not required to announce when they leave. This means that *leaves are essentially the same as crashes*.

In the same vein, the discrepancy between our result and those in [23] and a footnote in [2] claiming that a finite number of changes is necessary for liveness can be attributed to differences in the churn models. An important difference between our simulation and those in [2, 17] is that they ensure safety even when their churn and synchrony

assumptions are violated, whereas ours does not when the churn is very large. One of the contributions of this paper is to point out that by making different, yet still reasonable, assumptions on churn it is possible to get a solution with different, yet still reasonable, properties and, in particular, to overcome the prior constraint that churn must stop to ensure liveness. That is, we are suggesting a different point in the solution space.

## 2 Model

We consider an asynchronous message-passing system, with nodes running *client* (*reader* or *writer*) and *server* threads. Each node runs exactly one server thread, at most one reader thread, and at most one a writer thread. Nodes can enter and leave the system during an execution. A node that leaves the system cannot re-enter the system. (This restriction is easy to remove by giving a new name to a node that wants to re-enter.) We assume that at most $f \geq 0$ nodes can crash during an execution.

We say that a node is *present* at time $t$ if it has entered but has not left by time $t$ and we let $N(t)$ denote the number of servers whose nodes are present at time $t$. We assume that there are always at least $N_{min}$ servers whose nodes are present in the system, i.e., at all times $t$, $N(t) \geq N_{min}$.

Nodes communicate through a broadcast service that provides a mechanism to send the same message to all nodes in the system. If a server wants to send a message to one of the clients, it can do so by broadcasting the message and indicating that it should be ignored by the other clients. A message that is broadcast by a node $p$ at time $t$ is guaranteed to arrive at each node $q \neq p$ within $D$ units of time, provided that $q$ is present throughout the interval $[t, t+D]$. If $q$ is present for some but not all of $[t, t+D]$, then $q$ might or might not receive the message. Nodes that enter after time $t + D$ do not (directly) receive the message. All messages broadcast by $p$ are received by $q$ in the order in which $p$ sent them. In addition to the maximum transmission delay, $D$ includes the maximum time for handling the message at both the sender and the receiver. There is no lower bound on the actual length of time it takes for a message to be transmitted, nor on the amount of time to perform local computation at a node, i.e., they could take an aribtrarily small amount of time.

Nodes do not have clocks, so they cannot determine the current time nor directly measure how much time has elapsed since some event. They also do not know the value of $D$. The system is essentially asynchronous as there is no bound on the ratio between the fastest and slowest messages. In fact, any problem that can be solved in our model can be solved in the same model, but without the upper bound, $D$, on message delivery time. To see why, consider any execution in an asynchronous message passing model. Suppose that step $i$ of this execution occurs at time $1 - 2^{-i}$. Then every message that is received by a process is received within time $D = 1$. Moreover, if, in the original execution, messages are received along a link in the order they were sent, then the same is true in this timed execution. Hence, consensus cannot be solved in our model.

We assume the set of nodes that are present does not change too quickly: For all times $t$, at most $\alpha \cdot N(t)$ nodes enter or leave during the interval $[t, t + D]$. We call $\alpha$ the *churn rate* and we assume that the value of $\alpha$ is known to all nodes.

Let $S_0$ denote the set of nodes that are present initially, i.e. at time 0; $|S_0| = N(0)$.

## 3 The CCREG Algorithm

The algorithm combines a mechanism for tracking the composition of the system, with a simple algorithm, very similar to [16], for reading and writing the register.

In order to track the composition of the system (Algorithm 1), each node $p$ maintains a set of events, $Changes_p$, concerning the nodes that have entered the system. When a node $q$ enters, it adds $enter(q)$ to $Changes_q$ and broadcasts an enter message requesting information about prior events. When a node $p$ finds out that $q$ has entered the system, either by receiving this message or by learning indirectly from another node, it adds $enter(q)$ to $Changes_p$. When $q$ has received sufficiently many messages in response to its request, it knows relatively accurate information about prior events and the value of the register. (Setting the bound on the number of messages that should be received is a key challenge in the algorithm.) When this happens, $q$ adds $join(q)$ to $Changes_q$, sets its $is\_joined_q$ flag to $true$, and broadcasts a message saying that it has joined. We say that $q$ *joins* when this broadcast is sent. When $p$ finds out that $q$ has joined, either by receiving this message or by learning indirectly from another node, it adds $join(q)$ to $Changes_p$. When $q$ leaves, it simply broadcasts a leave message. When $p$ finds out that $q$ has left the system, either by receiving this message or by learning indirectly from another node, it adds $leave(q)$ to $Changes_p$.

When a node $p$ receives an enter message from a node $q$, it responds with an enter-echo message containing $Changes_p$, its current estimate of the register value (together with its timestamp), $is\_joined_p$ (indicating whether $p$ has joined yet), and $q$. When $q$ receives an enter-echo in response (i.e., that ends with $q$), it increments its *join-counter*. The first time $q$ receives such an enter-echo from a joined node, it computes $join\_bound$, the number of enter-echo messages it needs in response before it can join.

Once a node has joined, its reader and writer threads can handle read and write operations. A node is a *member* at time $t$ if it has joined but not left by time $t$.

Initially, $Changes_p = \{enter(q), join(q) \mid q \in S_0\}$, if $p \in S_0$, and $\emptyset$ otherwise. A node $p$ also maintains the set $Present_p = \{q \mid enter(q) \in Changes_p \wedge leave(q) \notin Changes_p\}$ of nodes that $p$ thinks are present, i.e., nodes that have entered, but have not left, as far as $p$ knows.

The server, reader and writer threads at the node share the variable *Changes* as well as its derived variable *Present*.

The client thread treats read and write operations in a similar manner (Algorithm 2). Both operations start with a read phase, used to obtain the current value of the register, using a query message, followed by a write phase, using an update message. A read operations just broadcasts the value it is about to return, keeping its sequence number. As in [5], write-back is needed to ensure linearizability of read operations. A write operation broadcasts the new value it wishes to write, with a sequence number one larger than the largest sequence number it has seen. Both the read phase and the read phase wait to receive sufficiently many response messages. (Again, setting the bound on the number of messages that should be received is a key challenge in the algorithm.)

A client $p$ maintains a sequence number, $tag$, which it increments at the beginning of each read phase. This is used to identify responses with the right read or write phase.

The server thread is simple (Algorithm 3). The server maintains the latest value of the register it knows about. When it receives an update message with a newer value

**Algorithm 1** CCREG—Common code, for node $p$.

**Local Variables:**

$is\_joined$ // Boolean to check if $p$ has joined the system; initially $false$

$join\_counter$ // for counting the number of enter-echo messages received by $p$; initially 0

$join\_bound$ // if non-zero, the number of enter-echo $p$ should receive before joining; initially 0

*Changes* // set of $enter(\cdot)$'s, $leave(\cdot)$'s, and $join(\cdot)$'s known by $p$; initially $\{enter(q), join(q) \mid q \in S_0\}$ if $p \in S_0$, and $\emptyset$, otherwise

$val$ // latest register value known to $p$; initially $\perp$

$seq$ // sequence number of latest value known to $p$; combined with next variable to make a unique timestamp for the write; initially 0

$id$ // id of node that wrote latest value known to $p$; initially $\perp$

**Derived Variable:**

*Present* $= \{q \mid enter(q) \in$ *Changes* $\wedge\ leave(q) \notin$ *Changes*$\}$

**When $p$ enters the system:**

1: bcast $\langle$"enter", $p\rangle$
2: Add $enter(p)$ to *Changes*

**When $\langle$"enter", $q\rangle$ is received:**

3: add $enter(q)$ to *Changes*
4: bcast $\langle$"enter-echo", *Changes*, $(val, seq, id)$, $is\_joined$, $q\rangle$

**When $\langle$"enter-echo", $C$, $(v, s, i)$, $j$, $q\rangle$ is received:**

5: **if** $(s, i) > (seq, id)$ **then**
6: $(val, seq, id) := (v, s, i)$
7: *Changes* $:=$ *Changes* $\cup\ C$
8: **if** $\neg is\_joined \wedge (p = q)$ **then**
9: **if** $(j = true) \wedge (join\_bound = 0)$ **then**
10: $join\_bound := \gamma \cdot |Present| - f$
11: $join\_counter$++
12: **if** $join\_counter \geq join\_bound > 0$ **then**
13: $is\_joined := true$
14: add $join(p)$ to *Changes*
15: bcast $\langle$"joined", $p\rangle$

**When $\langle$"joined", $q\rangle$ is received:**

16: add $join(q)$ to *Changes*
17: add $enter(q)$ to *Changes*
18: bcast $\langle$"joined-echo", $q\rangle$

**When $\langle$"joined-echo", $q\rangle$ is received:**

19: add $join(q)$ to *Changes*
20: add $enter(q)$ to *Changes*

**When $p$ leaves the system:**

21: bcast $\langle$"leave", $p\rangle$

**When $\langle$"leave", $q\rangle$ is received:**

22: add $leave(q)$ to *Changes*
23: bcast $\langle$"leave-echo", $q\rangle$

**When $\langle$"leave-echo", $q\rangle$ is received:**

24: add $leave(q)$ to *Changes*

for the register, it updates the current value. (Note that $(seq, id)$ pairs are compared lexicographically.) When it receives a query, it responds with the current value.

The correctness of CCREG relies on the following relations between the parameters:

$$f/(1-\alpha)^3 < N_{min} \tag{A}$$

$$\frac{3f/2(1-\alpha)^2}{(1-\alpha)^3/(1+\alpha)^2 - \beta} \leq N_{min} \tag{B}$$

**Algorithm 2** CCREG—Client code, for node $p$.

**Local Variables:**
$rw\_value$ // temporary storage for the written value or the return value
$tag$ // used to uniquely identify read and write phases of an operation; initially 0
$quorum\_size$ // stores the quorum size for a read or write phase; initially 0
$heard\_from$ // the number of responses/acks received for a read/write phase; initially 0
$rp\_pending$ // Boolean indicating whether a read phase is in progress; initially $false$
$wp\_pending$ // Boolean indicating whether a write phase is in progress; initially $false$
$read\_pending$ // Boolean indicating whether a read is in progress; initially $false$
$write\_pending$ // Boolean indicating whether a write is in progress; initially $false$

**When READ is invoked:**
30: $read\_pending := true$
31: call BeginReadPhase()

**When WRITE($v$) is invoked:**
32: $write\_pending := true$
33: $rw\_value := v$
34: call BeginReadPhase()

**Procedure** BeginReadPhase()
35: $tag$++
36: bcast $\langle$"query", $tag, p\rangle$
37: $quorum\_size := \beta|\mathit{Members}| + f/2$
38: $heard\_from := 0$
39: $rp\_pending := true$

**When $\langle$"response", $(v, s, i), rt\rangle$ is received:**
40: **if** $rp\_pending \wedge (rt = tag)$ **then**
41: **if** $(s, i) > (seq, id)$ **then**
42: $(val, seq, id) := (v, s, i)$
43: $heard\_from$++
44: **if** $heard\_from \geq quorum\_size$ **then**
45: $rp\_pending := false$
46: call BeginWritePhase($(val, seq, id)$)

**Procedure** BeginWritePhase($(v, s, i)$)
47: **if** $write\_pending$ **then**
48: $seq$++
49: bcast $\langle$"update", $(rw\_value, seq, p)$, $tag, p\rangle$
50: **if** $read\_pending$ **then**
51: bcast $\langle$"update", $(v, s, i), tag, p\rangle$
52: $quorum\_size := \beta|\mathit{Members}| + f/2$
53: $heard\_from := 0$
54: $wp\_pending := true$

**When $\langle$"ack", $wt\rangle$ is received:**
55: **if** $wp\_pending \wedge (wt = tag)$ **then**
56: $heard\_from$++
57: **if** $heard\_from \geq quorum\_size$ **then**
58: $wp\_pending := false$
59: **if** $read\_pending$ **then**
60: $read\_pending := false$
61: RETURN $rw\_value$
62: **if** $write\_pending$ **then**
63: $write\_pending := false$
64: ACK

$$[(1+\gamma)(1-\alpha)^3 - (1+\alpha)^3]N_{min} \geq 2f \quad \text{(C)}$$

$$(1-\alpha)^3/(1+\alpha)^3 \geq \gamma \quad \text{(D)}$$

$$\frac{(1+\alpha)^5 - 1}{(1-\alpha)^4} < \beta \quad \text{(E)}$$

$$(1 + 6\alpha + 2\alpha^3)/(2 - 2\alpha + \alpha^2) < \beta \quad \text{(F)}$$

These assumptions hold for $\alpha = 0.04$ and $N_{min} = 10f$, when taking $\beta = 0.65$ and $\gamma = 0.5$. Taking a smaller churn rate $\alpha = 0.02$ reduces the minimal size to $N_{min} = 5f$,

**Algorithm 3** CCREG—Server code, for node $p$.

```
    When ⟨"update", (v, s, i), wt, q⟩
      is received:
70: if (s, i) > (seq, id) then
71:     (val, seq, id) := (v, s, i)
72: if is_joined then
73:     send ⟨"ack", wt⟩ to
          (write-phase invoker) q
74: bcast ⟨"update-echo", (val, seq, id)⟩
    When ⟨"query", rt, q⟩ is received:
75: if is_joined then
76:     send ⟨"response", (val, seq, id), rt⟩ to
          (read-phase invoker) q
    When ⟨"update-echo", (v, s, i)⟩
      is received:
77: if (s, i) > (seq, id) then
78:     (val, seq, id) := (v, s, i)
```

with $\beta = 0.58$ and $\gamma = 0.56$. Note that for both these values of $\alpha$,

$$-1/\log_2(1-\alpha) \geq 4 \tag{G}$$

# 4 Correctness Proof

Consider any execution. We begin by putting bounds on the number of nodes that enter and leave during an interval of time and the number of nodes that are present at the end of the interval, as compared to the number present at the beginning. Extra work is required in the proof of Lemma 2 as the calculation of the maximum number of nodes that leave during an interval is complicated by the possibility of nodes entering during an interval and thus allowing additional nodes to leave.

**Lemma 1.** *For all $i \in \mathbb{N}$ and all $t \geq 0$, at most $((1+\alpha)^i - 1)N(t)$ nodes enter during $(t, t+Di]$ and $(1-\alpha)^i N(t) \leq N(t+Di) \leq (1+\alpha)^i N(t)$.*

**Lemma 2.** *For all nonegative integers $i \leq -1/\log_2(1-\alpha)$ and all $t \geq 0$, at most $(1-(1-\alpha)^i)N(t)$ nodes leave during $(t, t+Di]$.*

We say that a node is *active* at time $t$ if it has entered by time $t$, but has not left or crashed by time $t$. The next lemma shows that some node remains active throughout any interval of length $3D$.

**Lemma 3.** *For every $t > 0$, at least one node is active throughout $[\max\{0, t-2D\}, t+D]$.*

We define $\mathit{SysInfo}^I = \{enter(q) \mid t_q^e \in I\} \cup \{join(q) \mid t_q^j \in I\} \cup \{leave(q) \mid t_q^\ell \in I\}$ to be the set of all enter, join, and leave events that occur during time interval $I$. In particular, $\mathit{SysInfo}^{[0,0]} = \{enter(q) \mid q \in S_0\} \cup \{join(q) \mid q \in S_0\}$. The next observation holds since a node $p$ that is active throughout $[t_p^e, t+D]$ will directly receive all enter, joined, and leave messages broadcast during $[t_p^e, t]$ within $D$ time.

**Observation 1** *For every node $p$ and all times $t \geq t_p^e$, if $p$ is active at time $t+D$, then $\mathit{SysInfo}^{[t_p^e, t]} \subseteq \mathit{Changes}_p^{t+D}$.*

Together with the assumption that $\mathit{SysInfo}^{[0,0]} \subseteq \mathit{Changes}_p^0$ for all $p \in S_0$, we get:

**Observation 2** *For every node $p \in S_0$, if $p$ is active at time $t \geq 0$, then* $\mathit{SysInfo}^{[0,\max\{0,t-D\}]} \subseteq \mathit{Changes}_p^t$.

The purpose of Lemmas 4, 5, and 6 is to show that information about nodes entering, joining, and leaving is propagated properly, via the *Changes* sets.

**Lemma 4.** *Suppose a node $p \notin S_0$ receives an enter-echo message at time $t''$ from a node $q$ that sent it at time $t'$ in response to an enter message from $p$. If $p$ is active at time $t+2D$ and $q$ is active throughout $[\max\{0, t'-2D\}, t+D]$, where $\max\{0, t''-2D\} \leq t \leq t_p^e$, then* $\mathit{SysInfo}^{(\max\{0,t'-2D\},t]} \subseteq \mathit{Changes}_p^{t+2D}$.

*Proof.* Consider any node $r$ that enters, joins, or leaves at time $\hat{t}$, where $\max\{0, t' - 2D\} < \hat{t} \leq t$. If $q$ receives the message about this change from $r$ before the enter message from $p$, then the change is in $\mathit{Changes}_p^{t''} \subseteq \mathit{Changes}_p^{t+2D}$. Otherwise, $q$ receives the message from $r$ after the enter message from $p$ and sends an echo message in response by time $\hat{t} + D$. Since $p$ receives this message from $q$ by time $\hat{t} + 2D \leq t + 2D$, it follows that the change is in $\mathit{Changes}_p^{t+2D}$. Thus, $\mathit{SysInfo}^{(\max\{0,t'-2D\},t]} \subseteq \mathit{Changes}_p^{t+2D}$. □

**Lemma 5.** *For every node $p$, if $p$ is active at time $t \geq t_p^e + 2D$, then* $\mathit{SysInfo}^{[0,t-D]} \subseteq \mathit{Changes}_p^t$.

**Lemma 6.** *For every node $p \notin S_0$, if $p$ joins at time $t_p^j$ and is active at time $t \geq t_p^j$, then* $\mathit{SysInfo}^{[0,\max\{0,t-2D\}]} \subseteq \mathit{Changes}_p^t$.

*Proof.* Let $p \notin S_0$ be a node that joins at time $t_p^j \leq t$ and suppose the claim holds for all nodes that join before $p$. If $t \geq t_p^e + 2D$, then the claim follows by Lemma 5. So, assume that $t < t_p^e + 2D$.

Before $p$ joins, it receives an enter-echo message from a joined node in response to its enter message. Suppose $p$ first receives such an enter-echo message at time $t''$ and this enter-echo was sent by $q$ at time $t'$. Then $t_p^e \leq t' \leq t'' \leq t_p^j$. Since $q$ joined prior to $p$ and is active at time $t' \geq t_q^j$, $\mathit{SysInfo}^{[0,\max\{0,t'-2D\}]} \subseteq \mathit{Changes}_q^{t'} \subseteq \mathit{Changes}_p^{t''} \subseteq \mathit{Changes}_p^t$. If $t \leq 2D$ then $\max\{0, t - 2D\} = 0$ and the claim is true. So, assume that $t > 2D$.

Let $S$ be the set of nodes present at time $\max\{0, t' - 2D\}$, so $|S| = N(\max\{0, t' - 2D\})$. By Lemma 2 and Assumption (G), at most $(1-(1-\alpha)^3)|S|$ nodes leave during $(\max\{0, t'-2D\}, t'+D]$. Since $t'' \leq t'+D$, it follows that $|\mathit{Present}_p^{t''}| \geq |S|-(1-(1-\alpha)^3)|S| = (1-\alpha)^3|S|$. Hence, $p$ waits until it has received at least $\gamma|\mathit{Present}_p^{t''}| - f \geq \gamma(1-\alpha)^3|S| - f$ enter-echo messages before joining.

By Lemma 1, the number of nodes that enter during $(\max\{0, t' - 2D\}, t' + D]$ is at most $((1+\alpha)^3 - 1)|S|$. The number of nodes that leave during this interval is at most $(1-(1-\alpha)^3)|S|$ and at most $f$ nodes crash. Note that $p$ enters during $[\max\{0, t' - 2D\}, t'+D]$, but does not receive an enter-echo message from itself. Hence, the number enter-echo messages $p$ receives before joining from nodes that were active throughout

$[\max\{0, t' - 2D\}, t' + D]$ is at least

$$\gamma(1-\alpha)^3|S| - f - [((1+\alpha)^3 - 1)|S| + (1-(1-\alpha)^3)|S| + f - 1]$$
$$= [(1+\gamma)(1-\alpha)^3 - (1+\alpha)^3]|S| - 2f + 1$$

This is at least 1, since $\gamma = (1-\alpha)^3/(1+\alpha)^3$ and $f \leq |S|[(1-\alpha)^6 + (1-\alpha)^3(1+\alpha)^3 - (1+\alpha)^6]/2(1+\alpha)^3$. (By Assumption (C).)

Hence $p$ receives an enter-echo message by time $t_p^j$ from a node $q'$ that is active throughout $[\max\{0, t' - 2D\}, t' + D] \supseteq [\max\{0, t' - 2D\}, t - D]$.

Since $\max\{0, t'' - 2D\} \leq t - 2D \leq t_p^e \leq t' < t_p^e + D$, Lemma 4 implies that $\mathit{SysInfo}^{[\max\{0,t'-2D\},t-2D]} \subseteq \mathit{Changes}_p^t$. However, $\mathit{SysInfo}^{[0,\max\{0,t'-2D\}]} \subseteq \mathit{Changes}_p^t$, and hence, $\mathit{SysInfo}^{[0,\max\{0,t-2D\}]} \subseteq \mathit{Changes}_p^t$. □

Next we prove that every node that remains active sufficiently long after it enters succeeds in joining.

**Theorem 1.** *Every node $p \notin S_0$ that is active at time $t_p^e + 2D$ joins by time $t_p^e + 2D$.*

*Proof.* Let $p \notin S_0$ be a node that enters at time $t_p^e$ and is active at time $t_p^e + 2D$. Suppose the claim is true for all nodes that enter before $p$.

By Lemma 3, there is a node $q$ that is active throughout $[\max\{t_p^e - 2D, 0\}, t_p^e + D]$. If $q \in S_0$, then $q$ joins at time 0. If not, then $t_q^e < t_p^e$, so, by the induction hypothesis, $q$ joins by $t_q^e + 2D < t_p^e$. Since $q$ is active at time $t_p^e + D$, it receives the enter message from $p$ during $[t_p^e, t_p^e + D]$ and sends an enter-echo message in response. Since $p$ is active at time $t_p^e + 2D$, it receives the enter-echo message from $q$ by time $t_p^e + 2D$. Hence, by time $t_p^e + 2D$, $p$ received at least one enter-echo message from a joined node in response to its enter message.

Suppose the first enter-echo message $p$ received from a joined node in response to its enter message was sent by node $q'$ at time $t'$ and received by $p$ at time $t''$. By Lemma 6, $\mathit{SysInfo}^{[0,\max\{0,t'-2D\}]} \subseteq \mathit{Changes}_{q'}^{t'} \subseteq \mathit{Changes}_p^{t''}$.

Let $S$ be the set of nodes present at time $\max\{0, t' - 2D\}$. Then, by Lemma 1, $N(t' - D) \leq (1+\alpha)|S|$ and $N(t') \leq (1+\alpha)^2|S|$. Since $t'' \leq t' + D$, it follows from the churn assumption that at most $\alpha(1 + (1+\alpha) + (1+\alpha)^2)|S|$ nodes entered during $(t' - 2D, t'']$. Thus, $|\mathit{Present}_p^{t''}| \leq (1 + \alpha(1 + (1+\alpha) + (1+\alpha)^2))|S| = (1+\alpha)^3|S|$ and $join\_bound_p \leq \gamma(1+\alpha)^3|S| - f$.

By Lemma 2 and Assumption (G), at most $(1 - (1-\alpha)^3)|S|$ nodes leave during $(\max\{0, t' - 2D\}, t' + D]$. Since $t_p^e \leq t' \leq t_p^e + D$ and at most $f$ nodes crash, at least $(1-\alpha)^3|S| - f$ nodes in $S$ were active throughout $[t_p^e, t_p^e + D]$ and, hence, sent enter-echo messages in response to $p$'s enter message. By time $t_p^e + 2D$, $p$ receives all these enter-echo messages. Since $(1-\alpha)^3 = \gamma(1+\alpha)^3$ (Assumption (D)), node $p$ joins by time $t_p^e + 2D$. □

We now proceed to show that all read and write operations terminate. The key is to show that the number of responses for which an operation waits is small enough so that it is guaranteed to receive at least that many.

Since $enter(q)$ is added to $\mathit{Changes}_p$ whenever $join(q)$ is, we get:

**Observation 3** *For every time $t \geq 0$ and every node $p$ that is active at time $t$, $\mathit{Members}_p^t \subseteq \mathit{Present}_p^t$.*

Lemma 7 relates the number of nodes present in the system $2D$ time in the past to the value of a node's current estimate of the number of nodes present. Lemma 8 relates the number of nodes present in the system $4D$ time in the past to the value of a node's current estimate of the number of nodes that are members. These are useful for showing that a node's calculated quorum size is close to reality.

**Lemma 7.** *For every node $p$ and every time $t \geq t_p^j$ at which $p$ is active,*
$(1-\alpha)^2 \cdot N(\max\{0, t-2D\}) \leq |\mathit{Present}_p^t| \leq (1+\alpha)^2 \cdot N(\max\{0, t-2D\})$.

**Lemma 8.** *For every node $p$ and every time $t \geq t_p^j$ at which $p$ is active,*
$(1-\alpha)^4 \cdot N(\max\{0, t-4D\}) \leq |\mathit{Members}_p^t| \leq (1+\alpha)^4 \cdot N(\max\{0, t-4D\})$.

The next lemma shows a lower bound on the number of nodes that will reply to an operation's query or update message.

**Lemma 9.** *If node $p$ is active at time $t \geq t_p^j$, then the number of nodes that join by time $t$ and are still active at time $t+D$ is at least $\frac{(1-\alpha)^3}{(1+\alpha)^2}|\mathit{Present}_p^t| - f$.*

**Theorem 2.** *Every read or write operation completes.*

*Proof.* Each operation consists of a read phase and a write phase. Thus, if both the read and write phases of an operation terminate, then the operation itself terminates. We show that each phase terminates within $2D$ time, provided the client does not crash.

Consider a phase of an operation by client $p$ that starts at time $t$. Every node that joins by time $t$ and is still active at time $t+D$ receives $p$'s query or update message and replies with a response or ack message by time $t+D$. By Lemma 9, there are at least $\frac{(1-\alpha)^3}{(1+\alpha)^2}|\mathit{Present}_p^t| - f$ such nodes.

From Lemma 7 and Assumption (B),

$$|\mathit{Present}_p^t| \geq (1-\alpha)^2 N(\max\{0, t-2D\}) \geq (1-\alpha)^2 N_{min} \geq \frac{3f/2}{(1-\alpha)^3/(1+\alpha)^2 - \beta},$$

so

$$|\mathit{Present}_p^t| \left( \frac{(1-\alpha)^3}{(1+\alpha)^2} - \beta \right) \geq \frac{3f}{2}.$$

Hence, by Observation 3,

$$\frac{(1-\alpha)^3}{(1+\alpha)^2}|\mathit{Present}_p^t| - f \geq \beta|\mathit{Present}_p^t| + f/2 \geq \beta|\mathit{Members}_p^t| + f/2 = quorum\_size_p^t.$$

Thus, by time $t+2D$, $p$ receives sufficiently many response or ack messages to complete the phase. □

Now we prove linearizability of the CCREG algorithm.

A write operation $w$ by node $p$ consists of a read phase followed by a write phase. Let $t_w$ denote the time at the beginning of its write phase. At time $t_w$, node $p$ broadcasts an update message (on Line 49 or Line 51 of Algorithm 2) containing a triple $(v, s, i)$,

where $value(w) = v$ is the *value written by* $w$ and $ts(w) = (s, i) = (seq_p^{t_w}, id_p^{t_w})$ is the *timestamp* of $w$.

For any node $p$, let $ts_p^t = (seq_p^t, id_p^t)$ denote the *timestamp* of node $p$ at time $t$. Note that timestamps are created by write operations (on Line 48 of Algorithm 2) and are sent via enter-echo, update, and update-echo messages. Initially, $ts_p^0 = (0, \perp)$ for all nodes $p$. For any read or write operation $o$ by node $p$, the *timestamp of its read phase* is $ts^{rp}(o) = ts_p^t$, where $t$ is the time at the end of its read phase (i.e., when the conditional in Line 44 of Algorithm 2 is true). The *timestamp of its write phase* is $ts^{wp}(o) = ts_p^t$, where $t$ is the time at the beginning of its write phase (i.e., when it broadcasts on Line 49 or Line 51 of Algorithm 2). Note that $ts(w) = ts^{wp}(w)$ for every write operation $w$. Likewise, $ts(r) = ts^{rp}(r)$ is the *timestamp of a read operation* $r$.

The next series of lemmas (10 through 13) show that information about writes propagates properly throughout the system, and is analogous to previous results relating to the propagation of information about nodes entering, joining, and leaving (Observation 2 and Lemmas 4 through 6).

**Lemma 10.** *If $o$ is an operation whose write phase starts at $t_w$, node $p$ is active at time $t \geq t_w + D$, and $t_p^e \leq t_w$, then $ts_p^t \geq ts^{wp}(o)$.*

**Lemma 11.** *Suppose a node $p \notin S_0$ receives an enter-echo message at time $t''$ from a node $q$ that sends it at time $t'$ in response to an enter message from $p$. If $o$ is an operation whose write phase starts at $t_w$, $p$ is active at time $t \geq \max\{t'', t_w + 2D\}$, and $q$ is active throughout $[t_w, t_w + D]$, then $ts_p^t \geq ts^{wp}(o)$.*

**Lemma 12.** *If $o$ is an operation whose write phase starts at $t_w$ and node $p$ is active at time $t \geq \max\{t_p^e + 2D, t_w + D\}$, then $ts_p^t \geq ts^{wp}(o)$.*

**Lemma 13.** *If $o$ is an operation whose write phase starts at $t_w$, node $p \notin S_0$ joins at time $t_p^j$, and $p$ is active at time $t \geq \max\{t_p^j, t_w + 2D\}$, then $ts_p^t \geq ts^{wp}(o)$.*

**Theorem 3.** CCREG *ensures linearizability.*

*Proof.* Given an execution, we order all the read and write operations in the execution as follows. First, order the write operations in order of their timestamps. Note that all write operations have different timestamps, since each write operation by node $p$ has a timestamp with second component $p$ and first component larger than any timestamp $p$ has previously seen. Then insert each read operation immediately following the write operation with the same timestamp. Break ties among read operations by their start times. By construction, this total order is legal. It remains to show that if $op_1$ finishes before $op_2$ starts, then the construction orders $op_1$ before $op_2$.

Since each operation consists of a read phase followed by a write phase, it suffices to show that $ts^{wp}(op_1) \leq ts^{rp}(op_2)$. For convenience, we will refer to $ts^{wp}(op_1)$ as $\tau_w$ and $ts^{rp}(op_2)$ as $\tau_r$.

Let $w$ denote the write phase of $op_1$ and let $r$ denote the read phase of $op_2$. Let $p_1$ be the node that invokes $op_1$ and let $p_2$ be the node that invokes $op_2$. Let $t_w$ be the start time of $w$ and $t_r$ be the start time of $r$. Then $t_w < t_r$. Let $Q_w$ be the set of nodes that $p_1$ hears from during $w$ (i.e. that sent messages causing $p_1$ to increment $heard\_from$ on Line 56 of Algorithm 2) and $Q_r$ be the set of nodes that $p_2$ hears from during $r$ (i.e.

that sent messages causing $p_2$ to increment $heard\_from$ on Line 43 of Algorithm 2). Let $P_w$ and $M_w$ be the sizes of the *Present* and *Members* sets of $p_1$ at time $t_w$, and $P_r$ and $M_r$ be the sizes of the *Present* and *Members* sets of $p_2$ at time $t_r$.

**Case I:** $t_r > t_w + 2D$.

We start by showing there exists a node $q$ in $Q_r$ such that $t_q^j \leq \max\{0, t_r - 2D\}$. Each node $q \in Q_r$ receives and responds to $r$'s query, so it joins by time $t_r + D$. By Theorem 1, the number of nodes that can join in $(t_r - 2D, t_r + D]$ is at most the number of nodes that can enter in $(\max\{0, t_r - 4D\}, t_r + D]$. By Lemma 1, the number of nodes that can enter in $(\max\{0, t_r - 4D\}, t_r + D]$ is at most $((1+\alpha)^5 - 1) \cdot N(\max\{0, t_r - 4D\})$. By Lemma 8, $N(\max\{0, t - 4D\}) \leq M_r/(1-\alpha)^4$. From the code, $|Q_r| \geq \beta M_r + f/2$, which is larger than $\beta M_r$. By Assumption (E), it follows that $\beta M_r > M_r((1+\alpha)^5 - 1)/(1-\alpha)^4$, which is at most the number of nodes that can enter in $(\max\{0, t_r - 4D\}, t_r + D]$. Thus $|Q_r|$ is larger than the number of nodes that join in $(\max\{0, t_r - 2D\}, t_r + D]$.

Suppose $q$ receives $r$'s query message at time $t' \geq t_r$. If $q \in S_0$, then $t_q^j = 0$ and, by Lemma 10, $ts_q^{t'} \geq \tau_w$. So, suppose $q \notin S_0$. Then $0 < t_q^j \leq t_r - 2D < t'$. Since $t_w + 2D < t_r \leq t'$, Lemma 13 implies that $ts_q^{t'} \geq \tau_w$. Thus, $q$ responds to $r$'s query message with a timestamp at least as large as $\tau_w$ and, as a result, $\tau_r \geq \tau_w$.

**Case II:** $t_r \leq t_w + 2D$.

Let $J$ be the set of nodes that could reply to $r$'s query. Then $J = \{p \mid t_p^j < t_r \text{ and } p \text{ is active at time } t_r\} \cup \{p \mid t_r \leq t_p^j \leq t_r + D\}$. By Theorem 1, all nodes that are present at time $\max\{0, t_r - 2D\}$ join by time $t_r$ if they remain active. Therefore all nodes in $J$ are either active at time $\max\{0, t_r - 2D\}$ or enter during $(\max\{0, t_r - 2D\}, t_r + D]$ and, by Lemma 1, $|J| \leq N(\max\{0, t_r - 2D\}) + ((1+\alpha)^3 - 1)N(\max\{0, t_r - 2D\}) = (1+\alpha)^3 N(\max\{0, t_r - 2D\})$.

Let $K = \{p \mid t_p^j \leq t_r, p \text{ is active at } t_r + D, \text{ and } ts_p^{t_r} \geq \tau_w\}$. Note that $K$ contains all the nodes in $Q_w$ that do not leave or fail during $[t_w, t_r + D] \subseteq [\max\{0, t_r - 2D\}, t_r + D]$. By Lemma 2 and Assumption (G), at most $(1 - (1-\alpha)^3)N(\max\{0, t_r - 2D\})$ nodes leave during this interval and at most $f$ fail. From the code, $|Q_r| \geq \beta M_r + f/2$ and, by Lemma 8, $M_r \geq (1-\alpha)^4 N(\max\{0, t_r - 4D\})$. Similarly, $|Q_w| \geq \beta(1-\alpha)^4 N(\max\{0, t_w - 4D\}) + f/2$. Therefore, $|K| \geq (\beta(1-\alpha)^4 N(\max\{0, t_w - 4D\}) + \frac{f}{2}) - (1 - (1-\alpha)^3)N(\max\{0, t_r - 2D\}) - f$. Since $t_w < t_r \leq t_w + 2D$, Lemma 1 implies that $N(\max\{0, t_w - 4D\}) \geq (1-\alpha)^{-2} N(\max\{0, t_r - 4D\})$. Also by Lemma 1, $N(\max\{0, t_r - 4D\}) \geq (1-\alpha)^{-2} N(\max\{0, t_r - 2D\})$.

By Assumption (F), $\beta > (1 + 6\alpha + 2\alpha^3)/(2 - 2\alpha + \alpha^2)$. Hence

$$\begin{aligned}
|Q_r| + |K| &\geq \beta(1-\alpha)^4 N(\max\{0, t_r - 4D\}) \\
&\quad + \beta(1-\alpha)^4(1-\alpha)^{-2} N(\max\{0, t_r - 4D\}) \\
&\quad - (1 - (1-\alpha)^3)N(\max\{0, t_r - 2D\}) \\
&= \beta(1-\alpha)^2(2 - 2\alpha + \alpha^2) N(\max\{0, t_r - 4D\}) \\
&\quad - (3\alpha - 3\alpha^2 + \alpha^3)N(\max\{0, t_r - 2D\}) \\
&\geq \beta(2 - 2\alpha + \alpha^2) N(\max\{0, t_r - 2D\}) \\
&\quad - (3\alpha - 3\alpha^2 + \alpha^3)N(\max\{0, t_r - 2D\}).
\end{aligned}$$

$$\begin{aligned}\text{Thus}\quad |Q_r| + |K| &> [(1 + 6\alpha + 2\alpha^3) - (3\alpha - 3\alpha^2 + \alpha^3)]N(\max\{0, t_r - 2D\}) \\ &= [1 + 3\alpha + 3\alpha^2 + \alpha^3]N(\max\{0, t_r - 2D\}) \\ &= (1 + \alpha)^3 N(\max\{0, t_r - 2D\}) \\ &\geq |J|.\end{aligned}$$

This implies that $K$ and $Q_r$ intersect, since $K, Q_r \subseteq J$. For each node $p$ in the intersection, $ts_p \geq \tau_w$ when $p$ sends its response to $r$ and, thus, $\tau_r \geq \tau_w$. □

## 5 Discussion

We have shown how to simulate an atomic read/write register in a crash-prone asynchronous system where nodes can enter and leave, as long as the number of nodes entering and leaving during each time interval of length $D$ is at most a constant fraction of the current system size.

It would be nice to improve the constants for the churn rate and the maximum fraction of faulty nodes, perhaps with a tighter analysis. Proving lower bounds or tradeoffs on these parameters is an interesting avenue for future work. In fact, it might be possible to completely avoid the bound $\alpha$ on the churn rate, by spreading out the handling of node joins and leaves: To ensure a minimal number of nonfaulty nodes, a node might need to obtain permission before leaving, similarly to joins. This will also mean that the algorithm will maintain safety even when the churn bound is exceeded.

CCREG sends increasingly large Changes sets. The amount of information communicated might be reduced by sending only recent events, or by removing very old events. Another interesting research direction is to extend CCREG to tolerate more severe kinds of failures.

**Acknowledgments**: This work is supported by the Israel Science Foundation (grants 1227/10 and 1749/14), by Yad HaNadiv foundation, by the Natural Science and Engineering Research Council of Canada, and by the US National Science Foundation grant 0964696.

## References

1. Abraham, I., Chockler, G., Keidar, I., Malkhi, D.: Byzantine disk paxos: optimal resilience with Byzantine shared memory. Dist. Comp. 18(5), 387–408 (2006)
2. Aguilera, M.K., Keidar, I., Malkhi, D., Shraer, A.: Dynamic atomic storage without consensus. J. ACM 58(2), 7 (2011)
3. Aiyer, A.S., Alvisi, L., Bazzi, R.A.: Bounded wait-free implementation of optimally resilient byzantine storage without (unproven) cryptographic assumptions. In: Proceddings of 21st International Symposium on Distributed Computing. pp. 7–19 (2007)
4. Attiya, H.: Efficient and robust sharing of memory in message-passing systems. J. Alg. 34(1), 109–127 (Jan 2000)
5. Attiya, H., Bar-Noy, A., Dolev, D.: Sharing memory robustly in message-passing systems. J. ACM 42(1), 124–142 (Jan 1995)
6. Baldoni, R., Bonomi, S., Kermarrec, A.M., Raynal, M.: Implementing a register in a dynamic distributed system. In: IEEE International Conference on Distributed Computing Systems. pp. 639–647 (2009)

7. Baldoni, R., Bonomi, S., Raynal, M.: Implementing a regular register in an eventually synchronous distributed system prone to continuous churn. IEEE Transactions on Parallel and Distributed Systems 23(1), 102–109 (2012)
8. Beal, J., Gilbert, S.: RamboNodes for the metropolitan ad hoc network. In: Workshop on Dependability in Wireless Ad Hoc Networks and Sensor Networks (2003)
9. Chockler, G., Gilbert, S., Gramoli, V., Musial, P.M., Shvartsman, A.A.: Reconfigurable distributed storage for dynamic networks. J. Par. Dist. Comp. 69(1), 100–116 (2009)
10. Dutta, P., Guerraoui, R., Levy, R.R., Chakraborty, A.: How fast can a distributed atomic read be? In: Proceedings of the 23rd Annual ACM Symposium on Principles of Distributed Computing. pp. 236–245 (2004)
11. Georgiou, C., Musial, P.M., Shvartsman, A.A.: Long-lived RAMBO: Trading knowledge for communication. Theo. Comp. Sci. 383(1), 59–85 (2007)
12. Gilbert, S., Lynch, N.A., Shvartsman, A.A.: Rambo: A robust, reconfigurable atomic memory service for dynamic networks. Dist. Comp. 23(4), 225–272 (2010)
13. Guerraoui, R., Levy, R.: Robust emulations of shared memory in a crash-recovery model. In: Proceedings of the International Conference on Distributed Computing Systems. pp. 400–407 (2004)
14. Guerraoui, R., Vukolić, M.: Refined quorum systems. In: Proceedings of the 26th Annual ACM Symposium on Principles of Distributed Computing. pp. 119–128 (2007)
15. Ko, S.Y., Hoque, I., Gupta, I.: Using tractable and realistic churn models to analyze quiescence behavior of distributed protocols. In: IEEE Symposium on Reliable Distributed Systems. pp. 259–268 (2008)
16. Lynch, N.A., Shvartsman, A.A.: Robust emulation of shared memory using dynamic quorum-acknowledged broadcasts. In: Proceedings of the 27th International Symposium on Fault-Tolerant Computing. pp. 272–281 (1997)
17. Lynch, N.A., Shvartsman, A.A.: Rambo: A Reconfigurable Atomic Memory Service for Dynamic Networks. In: Proceedings of the 16th International Conference on Distributed Computing. pp. 173–190 (2002)
18. Malkhi, D., Reiter, M.K.: Byzantine quorum systems. Dist. Comp. 11(4), 203–213 (1998)
19. Malkhi, D., Reiter, M.K., Wool, A., Wright, R.N.: Probabilistic quorum systems. Information and Computation 170(2), 184–206 (2001)
20. Martin, J.P., Alvisi, L., Dahlin, M.: Minimal byzantine storage. In: Proceedings of the 16th International Conference on Distributed Computing. pp. 311–325 (2002)
21. Musial, P., Nicolaou, N., Shvartsman, A.A.: Implementing distributed shared memory for dynamic networks. Commun. ACM 57(6), 88–98 (2014)
22. Shraer, A., Martin, J.P., Malkhi, D., Keidar, I.: Data-centric reconfiguration with network-attached disks. In: Proceedings of the 4th International Workshop on Large Scale Distributed Systems and Middleware. pp. 22–26 (2010)
23. Spiegelman, A., Keidar, I.: On liveness of dynamic storage. CoRR abs/1507.07086 (Jul 2015), http://arxiv.org/abs/1507.07086
24. Vukolic, M.: Quorum Systems: With Applications to Storage and Consensus. Synthesis Lectures on Distributed Computing Theory, Morgan & Claypool Publishers (2012)