Consistent Distributed Memory Services: Resilience and Efficiency

Theophanis Hadjistasi
University of Connecticut, Storrs CT, USA
theo@uconn.edu

Alexander A. Schwarzmann
University of Connecticut, Storrs CT, USA
ass@uconn.edu

Abstract

Reading, *'Riting*, and *'Rithmetic*, the three R's underlying much of human intellectual activity, not surprisingly, also stand as a venerable foundation of modern computing technology. Indeed, both the Turing machine and von Neumann machine models operate by reading, writing, and computing, and all practical uniprocessor implementations are based on performing activities structured in terms of the three R's. With the advance of networking technology, communication became an additional major systemic activity. However, at a high level of abstraction, it is apparently still more natural to think in terms of reading, writing, and computing. While it is hard to imagine distributed systems—such as those implementing the World-Wide Web—without communication, we often imagine browser-based applications that operate by retrieving (i.e., reading) data, performing computation, and storing (i.e., writing) the results. In this article, we deal with the storage of shared readable and writable data in distributed systems that are subject to perturbations in the underlying distributed platforms composed of computers and networks that interconnect them. The perturbations may include permanent failures (or crashes) of individual computers, transient failures, and delays in the communication medium. The focus of this paper is on the implementations of distributed atomic memory services. Atomicity is a venerable notion of consistency, introduced in 1979 by Lamport [35]. To this day atomicity remains the most natural type of consistency because it provides an illusion of equivalence with the serial object type that software designers expect. We define the overall setting, models of computation, definition of atomic consistency, and measures of efficiency. We then present algorithms for single-writer settings in the static models. Then we move to presenting algorithms for multi-writer settings. For both static settings we discuss design issues, correctness, efficiency, and trade-offs. Lastly we survey the implementation issues in dynamic settings, where the universe of participants may completely change over time. Here the expectation is that solutions are found by integrating static algorithms with a reconfiguration framework so that during periods of relative stability one benefits from the efficiency of static algorithms, and where during the more turbulent times performance degrades gracefully when reconfigurations are needed. We describe the most important approaches and provide examples.

2012 ACM Subject Classification Networks → Cloud computing,General and reference → Surveys and overviews

Keywords and phrases Atomicity, shared-memory, read/write objects, fault-tolerance, latency

Digital Object Identifier 10.4230/LIPIcs.ICALP.2018.1

Category Invited Paper

1 Introduction

Shared storage services are located at the core of most information-age systems. Shared memory systems surveyed in this work provide objects that support two different access operations. That is, a *read*, that obtains the current value of the object, and a *write* that replaces the old value of the object with a new one. To be useful, such objects need to be *resilient* to failures and perturbations in the underlying computing medium, and must be *consistent* in that there are guarantees regarding relationships between previously written values and the values read by subsequent read operations. Such resilient and consistent object are also called registers. Here we focus on read/write objects, however for objects with more complicated semantics, such as transactions or read-modify-write operations, there exist common implementation challenges that any distributed storage system faces and needs to resolve. Imagine a storage system that is implemented as a central server. The server accepts client requests to perform operations on its data objects and returns responses. Conceptually, this approach is simple, however, two major problems can already be observed. The first is that the central server is a performance bottleneck. The second is that the server is a single point of failure. The quality of service in such an implementation degrades rapidly as the number of clients grows, and the service becomes unavailable if the server crashes (imagine how inadequate a web news service would be were it implemented as a central server). Thus the system must be *available*. This means it must provide its services despite failures within the scope of its specification, for example, the system must be able to mask certain server and communication failures. The system must also support multiple concurrent accesses without imposing unreasonable degradation in performance. The only way to guarantee availability is through *redundancy*, that is, by using multiple servers and by replicating the contents of objects among these servers.

Replication introduces the challenge of ensuring *consistency*. How does the system record new values so that consequently the it can find and return the latest value of a replicated object? This problem was not present with a central server implementation: the server always contains the latest value. In a replicated implementation, one may attempt to consult all replicas in search of the latest value, but this approach is expensive and not fault-tolerant as it assumes that all replicas are accessible. A trivial solution would be in each operation to consult all replicas servers in search of the latest value, however, this is not fault-tolerant (as it assumes all replicas are accessible) and expensive. In any case, none of the implementation issues should be a concern for the clients of the distributed memory service. What the clients should expect to see is the illusion of a single-copy object that serializes all accesses so that each read operation returns the value of the preceding write operation, and that this value is at least as recent as that returned by any preceding read. More generally, the behavior of the object, as observed externally, must be consistent with the abstract sequential data type of the object, and in developing applications that use such objects the clients must be able to rely on the abstract data type of the object. This notion of consistency is formalized as *atomicity* [35] for read/write objects, and equivalently, as *linearizability* [32] that extends atomicity to arbitrary data types. While there is no argument that atomicity is the most convenient notion of consistency, we note that weaker notions have also been proposed and implemented, motivated primarily by efficiency considerations. Atomicity provides strong guarantees, making it more expensive to provide than weaker consistency guarantees [7]. We take the view that it is nevertheless important to provide simple and intuitive, be it more expensive, atomic consistency. Barbara Liskov, a Turing Prize laureate, in a keynote address (at [37]) remarked that atomicity is not cheap, however, if we do not guarantee it, this creates headaches for developers.

Contemporary storage systems may also provide more complex data access primitives implementing atomic *read-modify-write* operations. Such access primitives are much stronger than separate *read* and *write* primitives we consider in this work. Implementing such operations is expensive, and at its core requires atomic updates that in practice are implemented by reducing parts of the system to a *single-writer* model (ex., Microsoft's Azure [10]), by depending on clock synchronization hardware (ex., Google's Spanner [13]), or by relying on complex mechanisms for resolving event ordering such as *vector clocks* (ex., Amazon's Dynamo [15]). Our exposition of atomic read/write storage illustrates challenges that are common to all distributed storage systems.

Document structure. Section 2 describes the general distributed setting for implementing consistent shared memory services, defines atomic conistency, and describes the measures of efficiency. In Sections 3 and 4 we present several approaches that implement consistent shared memory in static services for the SWMR and the MWMR setting respectively. Lastly, we survey several approaches for providing consistent shared memory in dynamic systems in Section 5. We conclude with a discussion in Section 6.

2 Distribution and Consistency

We now describe a general distributed setting for implementing consistent shared memory services.

Modeling distributed platforms. We model the system as a collection of interconnected computers, or nodes, that communicate by sending point-to-point messages. Each node has a unique identifier from some well-ordered set $\mathcal{I}$, local storage, and it can perform local computation. A node may fail by *crashing* at any point of the computation. The time of failure is chosen by and adversary that has knowledge of the past computation and the algorithms implementing a particular distributed service. Any node that crashes stops operating: it does not perform any local computation, it does not send any messages, and any messages sent to it are not delivered. Common approaches to implementing resilient in the face of failures algorithm specify failure models that provide qualitative or quantitative restrictions on the power of adversaries, e.g., by limiting the adversary to causing at most f crashes for some algorithm-specific parameter f.

The system is *asynchronous*, and the nodes have no access to a global clock or synchronization mechanisms. This means that relative processing speeds at the nodes can be arbitrary, and that the nodes do not know the upper bound on time that it takes to perform a local computation. The message delays can also be arbitrary, and the nodes do not know bounds on message latency (although such bounds may exist). Thus algorithms may not rely on assumptions about global time or delays.

We assume that messages can be reordered in transit, however, the messages cannot be corrupted, duplicated, or generated spontaneously. If a message is received then it must have been previously sent. The messages are not lost, but message loss can be modeled as long delays (we do not address techniques for constructing more dependable communication services, e.g., by using retransmission or gossip).

The nodes with ids in the set $\mathcal{I}$ include a set of writers $\mathcal{W}$, a set of readers $\mathcal{R}$, and a set of replica servers $\mathcal{S}$. These sets need not be disjoint, but it is helpful to view separately the roles a participant in the service can play. We categorize a distributed networked system as either *static* or *dynamic* as follows. In the *static* system the set of participating nodes is fixed, and each node may know the identity of all participants; crashes (or voluntary departures) may remove nodes from the system. Static algorithms are commonly designed to tolerate up to $f < |\mathcal{S}|/2$ server crashes and arbitrary number of crashes among readers and writers. In the *dynamic* system the set of nodes may be unbounded, and the set of participating nodes may completely change over time as the result of crashes, departures, and new nodes joining. The failure models considered in dynamic settings are much more complicated, given the systems may dramatically evolve over time.

Distributed shared memory and consistency. A distributed shared memory service emulates a shared memory space comprised of readable and writable objects, often called registers, over a networked platform that where distributed nodes communicate by message passing. Service implementations use replication to ensure survivability and availability of the objects, but the service makes this invisible to the clients. The contents of each object is replicated across several *servers* or *replica hosts.* Clients invoke read and write operations on the objects, where the clients that perform read operations are called *readers*, and those that perform write operations are called *writers* (a client may be both a reader and a writer).

In response to client requests the service invokes a *protocol* that involves communication with the replica hosts. This protocol implements and determines the consistency guarantees of the memory system. Atomic consistency definition involves "shrinking" the duration of each operation in any execution to a chosen *serialization point* between the operation's invocation and response, and requiring that the ordering of the operations according to the serialization points preserves their real-time ordering, and the resulting behavior of the object is consistent with its sequential specification. In particular, if a read is invoked after a write completes, then the read is guaranteed to return either the value of that write, or a value written by subsequent write that precedes the read. Additionally, if a read is invoked after another read completes, it returns the same or a "newer" value than the preceding read.

Whereas atomicity is often defined in terms of an equivalence with a serial memory, the definition given below implies this equivalence (as shown in in Lemma 13.16 in [39]), and is more convenient to use because it provides a usable recipe for proving atomic consistency.

▶ **Definition 1** (Atomicity, [39]). An implementation of an object is atomic, if for any execution if all the read and write operations that are invoked on an object complete, then the read and write operations for the object can be partially ordered by an ordering $\prec$, so that the following conditions are satisfied:

A1. The partial order is consistent with the external order of invocations and responses, that is, there do not exist read or write operations π_1 and π_2 such that π_1 completes before π_2 starts, yet $\pi_2 \prec \pi_1$.

A2. All write operations are totally ordered and every read operation is ordered with respect to all the writes.

A3. Every read operation ordered after any writes returns the value of the last write preceding it in the partial order; any read operation ordered before all writes returns the initial value.

Efficiency, Rounds and Message Exchanges. In assessing the efficiency of read and write operations of an implementation, we measure *communication latency*, *local computation time*, and *message complexity* of operations.

Communication latency of an operation is measured in terms of *communication rounds* or *communication exchanges.* The protocol implementing each operation involves a collection of sends of typed messages and the corresponding receives. A communication round is defined following [17].

▶ **Definition 2** (Communication Round, [17])**.** A process p performs a communication round during an operation π in an execution of an implementation A if all the following hold:

1. process p sends message(s) for operation π to a set of processes $Z \subseteq \mathcal{I}$,
2. upon the delivery of the message for π to process q, $q \in Z$, q sends a reply for π to p without waiting for any other messages, and
3. when p receives the collection of replies that is deemed sufficient by the implementation, it terminates the round. After this either p starts a new round or π completes.

A communication exchange is defined in [30].

▶ **Definition 3** (Communication Exchange, [30])**.** Within an execution of an implementation A, a communication exchange is the set of sends and corresponding matching receives for a specific type of message within the protocol.

We can observe that a round in Definition 2 is composed of two exchanges: the first is comprised of sends in item (1) and the corresponding receives in item (2), and the second is comprised of the reply sends in item (2) and the corresponding receives in item (3). Thus, in essence each exchange constitutes "one half" of a round. Traditional implementations in the style of ABD are structured in terms of communication rounds, cf. [5, 26], each consisting of two exchanges. The first is a broadcast from a reader or writer process to the servers, and the second is a convergecast in which the servers send corresponding responses to the initiating process.

Computation time accounts for all local computation within an operation; here time complexity of local computation may be significant. When local computation is not more than a constant time per each message send and receive, we consider this to be insignificant relative to the communication latency of an operation. Otherwise, computation time needs to be assessed in addition to communication latency.

Message complexity of an operation is determined by the worst case number of messages sent during the operation.

3 SWMR Implementations

Algorithms designed for single-writer static settings assume a fixed set known participants and accommodate some dynamic behaviors, such as asynchrony, transient failures, and permanent crashes within certain limits. A summary of the most relevant results for this setting is given in the first part of Table 1.

We commence by presenting the the seminal work of Attiya, Bar-Noy, and Dolev [5] that provides an algorithm, colloquially referred to as ABD, that implements SWMR atomic objects in message-passing crash-prone asynchronous environments. This work won the Dijkstra Prize in 2011. In ABD replication helps achieve fault-tolerance and availability, and the implementation replicates objects at nodes in the set $\mathcal{S}$, called servers, and it tolerates f replica servers crashes, provided a majority of replicas do not fail, i.e., $|\mathcal{S}| > 2f$. Read and write operations are ordered using logical *timestamps* associated with each written value. These timestamps totally order write operations, and therefore determine the values that read operations return. All operations terminate provided a majority of replicas do not crash.

A pseudocode for ABD is given in Algorithm 1; in referring to the numbered lines of code we use the prefix "L" to stand for "line". Write operations involve a single communication round-trip consisting of *two* communication exchanges. The writer broadcasts its request to all replica servers during the first exchange and terminates once it collects acknowledgments from some majority of servers in the second exchange (L19-23). Each read operation takes two rounds involving in *four* communication exchanges. The reader broadcasts a read request

Table 1 Model, Communication Exchanges, Message Complexities, Participation Bounds, and Predicate Computational Class.

Algorithm	Model	Write Exch.	Read Exch.	Wrt Msg Comp	Rd Msg Comp	Client Participation	Local Complexity
ABD [5]	SWMR	2	4	$2\|\mathcal{S}\|$	$4\|\mathcal{S}\|$	Unbounded	Constant
FAST [17]	SWMR	2	2	$2\|\mathcal{S}\|$	$2\|\mathcal{S}\|$	$\|\mathcal{R}\| < \frac{\|\mathcal{S}\|}{f} - 2$	NP-Hard
SF [26]	SWMR	2	2 or 4	$2\|\mathcal{S}\|$	$4\|\mathcal{S}\|$	$\|\mathcal{V}\| < \frac{\|\mathcal{S}\|}{f} - 1$	NP-Hard
SLIQ [25]	SWMR	2	2 or 4	$2\|\mathcal{S}\|$	$4\|\mathcal{S}\|$	Unbounded	Constant
CCFAST [4]	SWMR	2	2	$2\|\mathcal{S}\|$	$2\|\mathcal{S}\|$	$\|\mathcal{R}\| < \frac{\|\mathcal{S}\|}{f} - 2$	Polynomial
OHSAM [30]	SWMR	2	3	$2\|\mathcal{S}\|$	$2\|\mathcal{S}\| + \|\mathcal{S}\|^2$	Unbounded	Constant
OHSAM′ [30]	SWMR	2	2 or 3	$2\|\mathcal{S}\|$	$3\|\mathcal{S}\| + \|\mathcal{S}\|^2$	Unbounded	Constant
CCHYBRID [3]	SWMR	2	2 or 4	$2\|\mathcal{S}\|$	$4\|\mathcal{S}\|$	Unbounded	Polynomial
OHFAST [3]	SWMR	2	2 or 3	$2\|\mathcal{S}\|$	$\|\mathcal{S}\|^2$	Unbounded	Polynomial
MR [41]	SWMR	2	2 or 3 or 4	$\|\mathcal{S}\|^2$	$4\|\mathcal{S}\|$	Unbounded	Constant
ERATO [21]	SWMR	2	2 or 3	$2\|\mathcal{S}\|$	$3\|\mathcal{S}\| + \|\mathcal{S}\|^2$	Unbounded	Constant
ABD-MW [5, 40]	MWMR	4	4	$4\|\mathcal{S}\|$	$4\|\mathcal{S}\|$	Unbounded	Constant
SFW [18]	MWMR	2 or 4	2 or 4	$4\|\mathcal{S}\|$	$4\|\mathcal{S}\|$	Unbounded	NP-Hard
CWFR [24]	MWMR	4	2 or 4	$4\|\mathcal{S}\|$	$4\|\mathcal{S}\|$	Unbounded	Constant
OHMAM [30]	MWMR	4	3	$4\|\mathcal{S}\|$	$2\|\mathcal{S}\| + \|\mathcal{S}\|^2$	Unbounded	Constant
OHMAM′ [30]	MWMR	4	2 or 3	$4\|\mathcal{S}\|$	$3\|\mathcal{S}\| + \|\mathcal{S}\|^2$	Unbounded	Constant
ERATO-MW [21]	MWMR	4	2 or 3	$4\|\mathcal{S}\|$	$3\|\mathcal{S}\| + \|\mathcal{S}\|^2$	Unbounded	Constant

to all replica servers in the first exchange, collects acknowledgments from some majority of servers in the second exchange, and it discovers the maximum timestamp (L3-7). In order to ensure that any subsequent read will return a value associated with a timestamp at least as high as the discovered maximum, the reader propagates the value associated with the maximum timestamp to at least a majority of servers before completion (L8-11). The correctness of this implementation, that is, atomicity, relies on the fact that any two majorities have a non-empty intersection. The local computation at readers, writers and servers in ABD incurs insignificant computational overhead.

Following ABD, a folklore belief developed that in atomic memory implementations, "reads must write." The work by Dutta et al. [17] refuted this belief by presenting an algorithm, called FAST, in which all read and write operations involve only *two* communication exchanges. Such operations are called *fast*. To avoid the second round in read operations, FAST uses two mechanisms: (*i*) a recording mechanism at the servers, and (*ii*) a predicate that uses the server records at the readers. Here, each server records in a set all processes that witness its local timestamp and resets it whenever it learns a new timestamp. Each reader explores the sets from the different server replies to determine whether "enough" processes witnessed the maximum observed timestamp. If the predicate holds, the reader returns the value associated with the maximum timestamp. Otherwise it returns the value associated with the previous timestamp. The predicate takes in account *which* processes witnessed the latest timestamp as it examines the intersection of the received sets.

It was also shown in [17] that atomic memory implementations are only possible when the number of readers is constrained in with respect to the number of replicas servers and in inverse proportion to the number of crashes as stated in the following theorem.

▶ **Theorem 4** ([17]). *Let $f \geq 1$, $|\mathcal{W}| = 1$ and $|\mathcal{R}| \geq 2$. If $|\mathcal{R}| \geq \frac{|\mathcal{S}|}{f} - 2$, then there is no fast atomic register implementation.*

Algorithm 1 Reader, Writer, and Server Protocols for SWMR algorithm ABD

```
1: At each reader $r$
2: function READ($v$: output)
3: Get: broadcast ⟨get, $i$⟩ to all replica servers
4:         await responses ⟨get-ack, $v'$, $ts'$⟩
5:             from some majority of servers
6:         Let $v$ be the value associated with the
7:         maximum timestamp $maxts$ received
8: Put: broadcast ⟨put, $v$, $maxts$, $i$⟩ to all servers
9:         await responses ⟨put-ack⟩
10:            from some majority of servers
11:        return($v$)

12: At each server $s$
13: State $v$ init $\perp$, $ts$ init 0
14: Upon receive ⟨get, $j$⟩
15:        send ⟨get-ack, $v$, $ts$⟩ to $j$

16: At each writer $w$
17: State $ts$ init 0
18: function WRITE($v$: input)
19: Put: $ts \leftarrow ts + 1$
20:        broadcast ⟨put, $v$, $ts$, $i$⟩ to all servers
21:        await responses ⟨put-ack⟩
22:            from some majority of servers
23:        return()

24: At each server $s$
25: Upon receive ⟨put, $v'$, $ts'$, $j$⟩
26:     if $ts' > ts$ then
27:         $(ts, v) \leftarrow (ts', v')$
28:     send ⟨put-ack⟩ to $j$
```

A recent work by Fernández Anta, Nicolaou, and Popa [4], has shown that, although the result in [17] is efficient in terms of communication, it requires reader processes to evaluate a computationally hard predicate. The authors abstracted the predicate used in Fast as a computational problem that they show to be NP-hard via a reduction from the decision version of the Maximum Edge Biclique Problem [43], which is NP-Complete. This suggest the existence of a trade-off between communication efficiency and computational overhead in atomic memory implementations.

Given the inherent limitation on the number of readers in fast single-writer implementations, Georgiou et al. [26] sought a solution that would remove the limit on the number of readers, in exchange for slowing down some operations, i.e., the goal is to enable fast operations, but allow slower operations, taking more than two communication exchanges, when this is unavoidable. They provided a SWMR algorithm, named Sf, that adopts an approach to implementing readers similar to the one in [17], but uses a polynomial time predicate to determine whether it is safe for a read operation to terminate after two exchanges. In order not to place bounds on the number of readers, the authors group readers into abstract entities, called virtual nodes, serving as enclosures for multiple readers. This refinement has a non-trivial challenge of maintaining consistency among readers within the same virtual node. This solution trades communication for the scalability in the number of participating readers. In Sf significant computational overheads incur in order to determine the speed of an operation (to evaluate the mentioned predicate). At most a single complete read operation performs *four* exchanges for each write operation. Writes and any read operation that precedes or succeeds a *four* exchange read, is *fast*. This development motivated creating a new class of implementations, called *semifast* implementations. Informally, an implementation is *semifast* if either all reads are *fast* or all write operations *fast*. Algorithm Sf becomes fast (same as [17]) when each virtual node contains one reader.

Georgiou et al. [25] showed that fast and semifast quorum-based SWMR implementations are possible iff a common intersection exists among all quorums. Because a single point of failure exists in such solutions (i.e., any server in the common intersection), this renders such implementations not fault-tolerant. The same work introduced Quorum Views, client-side tools that examine the distribution of the latest value among the replicas in order to enable fast read operations (two exchanges) under read and write operation concurrency. The authors derived a a SWMR algorithm, called Sliq, that requires at least one single *slow* read per any write operation, and where all writes are *fast*. No bound is placed on the number of readers. Sliq trades communication for the scalability in the number of participating readers. Here only insignificant computation effort is needed to examine the distribution of object values among the replicas that the reader receives during the read operation.

Another algorithm, called ccFast, with a new predicate, is given by Fernández Anta et al. [4], that allows the operations to be *fast* with only polynomial computation overhead. The idea of the new predicate is to examine the replies received in the first communication round of a read operation and determine *how many* (instead of *which* [17]) processes witnessed the maximum timestamp among those replies. With this modification, the predicate takes polynomial time to decide the value to be returned and it reduces the size of each message sent by the replica nodes. Algorithm ccFast is more practical than [17], but it has the same constraint on the number of readers.

Previous works dealt with algorithms that used only communication between clients (readers and writers) and the replica servers. Hadjistasi, Nicolaou and Schwarzmann [30] explored another approach that exploits server-to-server communication. In particular, they showed that atomic operations do not need to involve complete communication round trips between clients and servers. The authors focused on the gap between one-round and two-round algorithms, seeking implementations where operations can take "one and a half rounds," i.e., be able to complete in *three* exchanges. They presented a SWMR algorithm, called OhSam, which stands for *One and a Half Rounds Single-writer Atomic Memory.* Here reads take *three* exchanges: the first from the client to the servers, the second is server-to-server, and the third is from the servers to the client. Such implementations trade latency for message complexity: the latency is reduced to three exchanges, while server-to-server exchange has quadratic message complexity in the number of servers. A key idea of the algorithm is that the reader returns the value that is associated with the *minimum* timestamp that corresponds to the last complete write operation (cf. the observation in [17]).

In the same work [30], authors revised the protocol implementing read operations of algorithm OhSam to yield a protocol that implements read operations that terminate in either *two* or *three* communication exchanges, OhSam′. The idea here is to let the reader determine "quickly" that a majority of servers hold the same timestamp (or tag) and its associated value. This is done by having the servers send relay messages to each other as well as to the readers. While a reader collects the relays and the read acknowledgments, if it observes in the set of the received relay messages that a majority of servers holds the same timestamp, then it safely returns the associated value and the read operation terminates in *two* exchanges. If that is not the case, then the reader proceeds similarly to algorithm OhSam and terminates in *three* communication exchanges. Algorithms OhSam and OhSam′ do not impose constraints on reader participation and perform a modest amount of local computation, resulting in negligible computation overhead.

Fernández Anta et al. [3] introduced a "multi-speed" algorithm, named ccHybrid that allows operations to terminate in either *two* or *four* communication exchanges. Algorithm ccHybrid does not impose any bounds on the number of the participating readers. ccHybrid uses the polynomial predicate introduced in [4] to determine the speed of a read operation, and it requires at most one *complete* slow operation per written value. This is similar to the semifast algorithm Sf [26]. However, in contrast with Sf, in which processes have to decide NP-hard predicates, ccHybrid performs only linear computation.

The same work [3] explores the idea of combining the polynomial predicate with the *three* communication exchanges read protocol of algorithm OhSam [30]. The resulting "multi-speed" algorithm, called OhFast, allows *one* and *one-and-a-half* round-trip operations, equivalent *two* or *three* communication exchanges. In algorithm OhFast, the decision of whether the read operation must be slow is moved to the replica servers. When replica servers determine that a slow read is necessary, they perform a *relay* phase to inform other servers before replying to the requesting reader. It is interesting that in OhFast not all servers uniformly perform a relay for a particular read operation. Some servers may be replying directly to the requesting reader, whereas others may perform a relay phase. Thus, it is possible for a read operation to terminate before receiving a reply from the server that initiates the relay.

A recent work by Mostefaoui and Raynal [41] defines what a *time-efficient* implementation of atomic registers is based on two different synchrony assumptions. The first assumes bounded message delays and is expressed in terms of delays, and the second assumes *round-based synchrony*. Authors then present a *time-efficient* implementation of atomic registers while trying to keep its design spirit as close as possible to ABD. We refer to this solution as algorithm MR. In the algorithm a write operation takes *two* communication exchanges and a read operation takes *two*, or *three*, or *four* exchanges. The heart of the given algorithm is the *wait predicate* that takes place on the servers side and it is associated with write operations. The wait predicate ensures both atomicity and the fact that the implementation is time-efficient. The trade-off between ABD and this implementation lies in the message complexity of write operations, which for ABD is linear and for MR is quadratic in to the number of replica servers. Algorithm MR is particularly interesting for registers used in read-dominated applications.

Lastly, Georgiou et al. [21] presented algorithm Erato, which stands for *Efficient Reads for ATomic Objects*. The algorithm improves the *three*-exchange read protocol of OhSam [30] to allow reads to terminate in either *two* or *three* exchanges using client-side tools, *Quorum Views*, introduced in algorithm Sliq [25]. The three exchanges of the new read protocol are as follows: (1) reader broadcasts a request to servers; (2) the servers share this information among themselves, including the reader, and (3) once this is "sufficiently" done, servers reply to the reader. During the second exchange, the reader uses Quorum Views [25] to categorize the distribution of timestamps, and determines whether it is able to complete the read. If not, it awaits "enough" messages from the third exchange before completion. Here, the idea of the algorithm is that when the reader is "slow" it returns the value associated with the *minimum* timestamp, i.e., the value of the previous write that is guaranteed to be complete (cf. [30] and [17]). Similarly to ABD, write operations take *two* exchanges.

4 MWMR Implementations

We now turn out attention to the multi-writer/multi-reader (MWMR) implementations of atomic memory. Whereas logical timestamps alone are sufficient to order write operations in the single-writer algorithms, the existence of multiple writers requires a somewhat different approach. The simplest approach is instead of a timestamp to use pairs consisting of a timestamp and processor id to order the written values. Such a pair is termed a *tag*. When a writer performs a write operation it associates the value with a tag $\langle ts, id \rangle$, where ts is a logical timestamp, and id is the writer's unique id that distinguishes the current write operation from all others. Tags are ordered lexicographically in establishing an order on the operations. A summary of the most relevant results for this setting is given in the second part of Table 1.

The work of Lynch and Schwarzmann [40] presented a multi-writer extension of algorithm ABD (and also introduced the notion of reconfigurable memory, where the set of replica servers can be dynamically reconfigured). The static version of their MWMR implementation, that we call ABD-mw, is given in Algorithm 2. In contrast with ABD, where the sole writer generates new timestamps without any communication, the writers in ABD-mw start a

Algorithm 2 Reader, Writer, and Server Protocols for MWMR algorithm ABD-mw

```
 1: At each reader $r$
 2: function READ($v$: output)
 3: Get: broadcast ⟨get, $i$⟩ to all replica servers
 4:         await responses ⟨get-ack, $v'$, $tag'$⟩
 5:             from some majority of servers
 6:         Let $v$ be the value associated with the
 7:         maximum tag $maxtag$ received
 8: Put: broadcast ⟨put, $v$, $maxtag$, $r$⟩ to all servers
 9:         await responses ⟨put-ack⟩
10:             from some majority of servers
11:         return($v$)

12: At each server $s$
13: State $v$ init ⊥, $tag$ init ⟨0, ⊥⟩
14: Upon receive ⟨get, $j$⟩
15:         send ⟨get-ack, $v$, $tag$⟩ to $j$

16: At each writer $w$
17: function WRITE($v$: input)
18: Get: broadcast ⟨get, $i$⟩ to all replica servers
19:         await responses ⟨get-ack, $v'$, $tag'$⟩
20:             from some majority of servers
21:         Let $maxtag = \langle ts, pid \rangle$ be the max tag
22:         Let $newtag = \langle ts+1, w \rangle$
23: Put: broadcast ⟨put, $v$, $newtag$, $w$⟩ to all servers
24:         await responses ⟨put-ack⟩
25:             from some majority of servers
26:         return()

27: At each server $s$
28: Upon receive ⟨put, $v'$, $tag'$, $j$⟩
29:     if $tag' > tag$ then
30:         $(tag, v) \leftarrow (tag', v')$
31:     send ⟨put-ack⟩ to $j$
```

write operation by performing an additional round in which the replica servers are queried for their latest tags. Once tags are received from a majority of servers, the writer increments the timestamp of the highest detected timestamp to produce its new tag. The the second round is performed as in ABD.

In more detail, the writer performs the "Get" round, broadcasting its request to the servers in the first exchange (L18). Servers reply with their latest timestamps in the second exchange (L18-22 and L12-15). The writer determines the highest timestamp among the replies, increments it, produces a new tag that includes its id, and then performs the "Put" round in which it in the third exchange broadcasts the new tag and the new value to all servers (L23-26). On the server side, if the incoming message contains a higher tag, then the server update its local information and send an acknowledgment in the fourth communication exchange (L27-31). The write protocol completes once the writer collects acknowledgments from a majority of servers. The first two exchanges ensure that the writer produces a tag that is higher than that of any preceding write. Thus a write operation for ABD-MW takes *four* exchanges in comparison with the *two* exchanges in ABD. The read protocol is identical to the four-exchange protocol in ABD, the only difference being that tags are used instead of timestamps. The correctness (atomicity) of this implementation, relies on the fact that any two majorities have a non-empty intersection and that in each round, the read and write protocols await responses from at least a majority of servers.

This algorithm places no constrains on the number of readers and writers, and it performs a modest amount of local computation, resulting in negligible computation overhead. This algorithm can also be used used with quorum systems instead of majorities [40, 44], because the only property of majorities that is used is that any two majorities have a non-empty intersection, just like any two quorums. The failure model for the quorum based solution is that any pattern of crashes is tolerated, provided that the servers in at least one quorums do not crash.

Algorithm ABD-MW established that two rounds are sufficient to implement atomic read and write operations. The question of whether *fast* (single round) implementations are possible was answered in the negative in [17], where it was shown that fast reads are possible only in the single-writer model SWMR. In particular, *fast* MWMR implementations are impossible when the set of readers $\mathcal{R}$ and the set of writers $\mathcal{W}$ contain more than two nodes each.

▶ **Theorem 5** ([17]). *Let $|\mathcal{W}| \geq 2$, $|\mathcal{R}| \geq 2$, and $f \geq 1$. Any atomic register implementation has a run in which some complete read or write operation is not fast.*

Moreover, Georgiou et al. [26] showed that *semifast* implementations (recall from Section 3 that in a semifast implementation either all reads are fast or all the write operations fast) are impossible in the MWMR setting.

▶ **Theorem 6** ([26]). *If $|\mathcal{W}| \geq 2$, $|\mathcal{R}| \geq 2$, and the number of server crashes $f \geq 1$, then semifast atomic register implementation is impossible.*

These impossibility results motivated the development of algorithms that allow some operations to complete in less than two rounds or in less than four communication exchanges. The work from Englert et al. [18] proposed hybrid approaches where some operations complete in *two* and other in *four* communication exchanges. Their algorithm SFW uses quorum systems and enables some reads and writes to be fast. In order to decide whether an operation can terminate after its first round, the algorithm employs two specialized predicates. However, the predicates are computationally hard (NP-hard), and fast write operations are enabled only if the quorum system satisfies certain quorum intersection properties, rendering the algorithm impractical.

Georgiou et al. [24] presented a MWMR algorithm, called CWFR, that allows fast read operations. The algorithm uses a generalization of client-side decision tools, Quorum Views, developed for the SWMR setting [25], to analyze the distribution of a value within a quorum of replies from servers to determine whether fast termination is safe. Since multiple writes can occur concurrently, an iterative technique is used to discover the latest potentially complete write operation. Here read operations terminate in either *two* or *four* communication exchanges. The write protocol is essentially the same as in ABD-MW, taking *four* exchanges to complete. Algorithm CWFR does not impose constrains on participation and it performs a modest amount of local computation, resulting in negligible computation overhead.

Hadjistasi et al. [30] sought a MWMR solution that involves *three* or *four* communication exchanges per operation, and developed algorithm OHMAM, which stands for *One and a Half Rounds Multi-writer Atomic Memory*. The authors adopted the three-exchange protocol from the SWMR algorithm OHSAM to the MWMR setting. The read protocol of OHMAM differs in that it uses tags instead of timestamps. The write protocol is identical to the one that ABD-MW uses and completes in *four* communication exchanges.

The authors then revised the protocol implementing read operations of algorithm OHMAM to yield a protocol that implements read operations that terminate in either *two* or *three* communication exchanges, the resulting algorithm is called OHMAM$'$. The idea here is to expedite the reader's determination that a majority of servers hold the same tag and its associated value. This is achieved by having the servers send relay messages to each other as well as to the requesting reader. While a reader collects the relays and the read acknowledgments, if it observes in the set of the received relay messages that a majority of servers holds the same timestamp, then it safely returns the associated value, thus terminating in *two* exchanges. If that is not the case, then the reader proceeds similarly to algorithm OHMAM and terminates in *three* communication exchanges. The operations perform insignificant amount of local computation.

Lastly, Georgiou et al. [21], using the SWMR algorithm ERATO as the basis, developed a MWMR algorithm, ERATO-MW. In adopting the single-writer algorithm, the challenge is that the read protocol cannot be used directly because it relies on the fact that if a write is in progress, then the preceding write is complete. Instead the algorithm implements a *three*-exchange read protocol based on [30] in combination with the iterative technique using Quorum Views as in [24]. This technique determines the completion status of a write operation, and also detects the last potentially complete write operation. Read operations complete in either *two* or *three* exchanges. Writes are similar to ABD-MW and take *four* communication exchanges. This algorithm also has a negligible computation overhead.

5 Shared Memory in Dynamic Settings

Additional challenges arise when a shared memory system must be long-lived and must ensure data longevity. A storage system may be able to tolerate failures of some servers, but over a long period it is conceivable that all servers may need to be replaced, because no servers are infallible, and also due to unavoidable changes or planned upgrades. Additionally, in mobile settings, e.g., remote search-and-rescue or military operations, it may be necessary to provide migration of data from one collection of servers to another, so that the data can move as the needs dictate. Whether our concern is data longevity or mobility, the storage system must provide seamless runtime migration of data: one cannot stop the world and reconfigure the system in response to failures and changing environment.

We now survey several approaches for providing consistent shared memory in more dynamic systems, that is, where nodes may not only crash or depart voluntarily, but where new nodes may join the service, and where the entire collection of servers need to be replaced. In general, the set of object replicas can substantially evolve over time, ultimately migrating to a completely different set of replica hosts. Thus, an implementation designed for static settings, e.g., algorithm ABD, cannot be used directly in dynamic settings because it relies on the majority of original replica hosts to always be available. In order to use an ABD-like approach in dynamic settings, one must provide some means for managing the collections of replica hosts, and to ensure that readers and writers contact suitable such collections.

It is noteworthy that dealing with dynamic settings and managing collections of nodes does not directly address the provision of consistency in memory services. Instead, these issues are representative of the broader challenges present in the realm of dynamic distributed computing. It is illustrative that implementations of consistent shared memory services can sometimes be constructed using distributed building blocks, such as those designed for managing collections of participating nodes, for providing suitable communication primitives, and for reaching agreement (consensus) in dynamic distributed settings. A tutorial covering several of these topics is presented by Aguilera et al. [2].

We start by presenting the consensus problem because it provides a natural basis for implementing an atomic memory service by establishing an agreed-upon order of operations, and because consensus is used in other ways in atomic memory implementations. Next we present group communication services (GCS) solutions that use strong communication primitives, such as totally ordered broadcast, to order operations. Finally we focus on approaches that extend the ideas of algorithm ABD to dynamic settings with explicit management of the evolving collections of replica hosts.

Consensus. Reaching agreement in distributed settings is a fundamental problem of computer science. The agreement problem in distributed settings is called *consensus*. Here a collection of processes need to agree on a value, where each process may propose a value for consideration. Any solution must satisfy the following properties: *Agreement:* no two processes decide on different values; *Validity:* the value decided was proposed by some process; *Termination:* all correct processes reach a decision. Consensus is a powerful tool in designing distributed services [39], however, consensus is a notoriously difficult problem to solve in asynchronous systems, where termination cannot be guaranteed in the presence of even a single process crash [20] (this is the seminal FLP impossibility result of Fischer, Lynch, and Paterson); thus consensus must be used with care.

Consensus algorithms can be used directly to implement an atomic data service by enabling the participants to agree on a global total ordering of all operations [36]. The correctness (atomicity) here is guaranteed regardless of the choice of a specific consensus implementation, but the understanding of the underlying platform characteristics can guide the choice of the implementation for the benefit of system performance (for a *tour de force* of implementations see [39]). Nevertheless, using consensus for each operation is a heavy-handed approach, especially given that perturbations may delay or even prevent termination. Thus, when using consensus, one must avoid invoking it in conjunction with individual memory operations, and make operations independent of the termination of consensus.

We note that achieving consensus is a more difficult problem than implementing atomic read/write objects. In particular, consensus cannot be solved for two or more processes by using atomic read/write registers [31, 38].

Group communication services. Among the most important building blocks for distributed systems are *group communication services* (GCS) [8]. GCSs enable processes at different nodes of a network to operate collectively as a group by means of multicast services that deliver messages to the members of the group, and offer various guarantees about the order and reliability of delivery. The basis of a GCS is a *group membership service*. Each process, at any time, has a unique *view* of the group that includes a list of the processes in the group. Views can change over time, and may become different at different processes. Another important concept introduced by the GCS approach is *virtual synchrony*, where an essential requirement is that processes that proceed together through two consecutive views deliver the same set of messages between these views. This allows the recipients to take coordinated action based on the message, the membership set, and the rules prescribed by the application [8].

GCSs offer one approach for implementing shared memory. For example, one can implement a global totally ordered multicast service on top of a view-synchronous GCS [19]. The ordered multicast is used to impose an order on the memory access operations, yielding atomic memory. The main disadvantage in such solutions is that in GCS implementations, forming a new view takes time, and client memory operations are delayed (or aborted) during the view-formation period.

Another approach is to integrate a GCS with algorithm ABD as done in the dynamic primary configuration GCS of [14] that implements atomic memory by using techniques of [6] within each configuration, where configurations include a group view and a quorum system.

A general methodology for dynamic service replication is presented in [9]. This reconfiguration model unifies the virtual synchrony approach with state machine replication, as used in consensus solutions, in particular, Paxos [36].

DynaStore Algorithm. DynaStore [1] is an implementation of a dynamic atomic memory service for multi-writer/multi-reader objects. The participants start with a default local configuration, that is, some common set of replica hosts. The algorithm supports three kinds of operations: *read*, *write*, and *reconfig*. The read and write operations involve two phases, and in the absence of reconfigurations, the protocol is similar to ABD. If a participant wishes to change its current configuration, it uses the *reconfig* operation and supplies with it a set of incremental changes.

The implementation of *reconfig* involves traversals of DAG's representing possible sequences of changed configurations. In each traversal the DAG may be revised to reflect multiple changes to the same configuration. The assumption that a majority of the involved hosts are not removed and do not crash ensures that there is a path through the DAG that is guaranteed to be common among all hosts. The traversal terminates when a sink node is reached. The *reconfig* protocol involves two phases. The goal of the first phase is similar to the Get phase of ABD: discover the latest value-tag pair for the object. The goal of the second phase is similar to the Put phase of ABD: convey the latest value-tag pair to a suitable majority of replica hosts. The main difference is that these two phases are performed in the context of applying the incremental changes to the configuration, while at the same time discovering the changes submitted by other participants. This "bootstraps" possible new configurations. Given that all of this is done by traversing all possible paths—and thus configurations—in the DAG's ensures that the common path is also traversed.

The *read* follows the implementation of *reconfig*, with the differences being: (a) the set of configuration changes is empty, and (b) the discovered value is returned to the client. The *write* also follows the implementation of *reconfig*, with the differences being: (a) the set of changes is empty, (b) a new, higher tag is produced upon the completion of the first phase, and (c) the new value-tag pair is propagated in the second phase.

We note that DynaStore implementation does not incorporate consensus for reconfiguration. On the other hand, reconfigurations are accomplished by additions and removals of individual nodes and this may lead to larger overheads as compared to approaches that evolve the system by replacing a complete configuration with another. Thus the latency of read and write operations are more dependent on the rate of reconfigurations. Finally, in order to guarantee termination, DynaStore assumes that reconfigurations eventually subside.

Rambo Framework. RAMBO is a dynamic memory service supporting MWMR objects [28]; RAMBO stands for Reconfigurable Atomic Memory for Basic Objects. This algorithm uses *configurations*, each consisting of a set of replica hosts plus a quorum system defined over these hosts, and supports *reconfiguration*, by which configurations can be replaced. Notably, any quorum configuration may be installed at any time, and quorums from distinct configurations are not required to have non-empty intersections. The algorithm ensures atomicity in all executions. During quiescent periods when there are no reconfigurations, the algorithm operates similarly to algorithm ABD [6, 40]. To enable long-term operation of the service, quorum configurations can be reconfigured. Reconfigurations are performed concurrently with any ongoing read and write operations, and do not directly affect such operations. Additionally, multiple reconfigurations may be in progress concurrently. Reconfiguration involves two decoupled protocols: (1) introduction of a new configuration by the component called *Recon*, and (2) upgrade to the new configuration and garbage collection of obsolete configuration(s). *Recon* always emits a unique new configuration. Different reconfiguration proposals are reconciled by executing consensus among the members of an existing configuration. Note that termination of read and write operations does not depend on termination of reconfiguration. It is the duty of a decoupled *upgrade* protocol to garbage collect old configurations and propagate the information about the object to the latest locally-known configuration. The main algorithm performs read and write operations using a two-phase strategy. The first phase gathers information from the quorums of active configurations, then the second phase propagates information to the quorums of active configurations. Note that during each phase new configurations may be discovered. To handle this each phase is terminated by a *fixed point* condition that involves a quorum from each active configuration.

Lastly, RAMBO is used as a framework for refinements and optimizations, and several subsequent works focused on practical considerations [29, 12, 22, 23, 33]. *GeoQuorums* [16] is an approach to implementing atomic shared memory on top of a physical platform that is based on mobile nodes moving in arbitrary patterns. The algorithm simplifies reconfiguration of RAMBO by using a finite set of possible configurations, and as the result it avoids the use of consensus. Here it is sufficient for a mobile node to discover the latest configuration, and contact and propagate the latest register information to all configurations.

6 Discussion

We presented several approaches for implement atomically consistent memory services in distributed message-passing systems. Our focus is on atomic consistency because it is an intuitive notion that hides the complexities of underlying implementations, presenting a convenient abstraction to the software builders. This is particularly valuable because of the common perception that the shared-memory paradigm is easier to deal with than the message-passing paradigm in designing distributed algorithms. The solutions presented in this work are representative of the different design choices available for implementing distributed memory services, and we emphasized the trade-offs present in different approaches.

We discussed in detail the issues of resilience and efficiency in the single-writer and multi-writer models in static settings. We then surveyed several approaches that implement consistent shared memory in these settings. In the established implementations frameworks some open problems remain regarding out ability to further improve the efficiency of services that use variable number of exchanges (two, three, and four) in implementing read and write operations. We also anticipate that additional lower bounds will be established to help better understand limitations on efficient implementations. For the static setting, it is also interesting to investigate the possibility of devising consistent implementations with *zero-delay* operations. That is, where operations are able to complete without additional communication, perhaps only relying on the knowledge obtain through prior communications. If the answer is in the negative, then it will still be interesting to understand the possibility of obtaining such implementations for notions of consistency weaker than atomicity, e.g., eventual consistency, or by weakening the power of adversity. As an example of this direction, the work of Chandra et al. [11] assumes a partially synchronous system using synchronized local clocks. Such solutions are particularly interesting for applications that are either read or write dominated.

The algorithms that we surveyed for the static settings may have difference fault-tolerance guarantees and be subject to efficiency trade-offs. This prompted researchers to perform empirical studies of their proposed algorithms [25, 24, 4, 3, 30, 21]. Here a goal is to understand how the analytical results are are reflected in practical efficiency. In addition to simulations, full scale cloud-based experimental evaluations will be certain to yield valuable observations in realistic settings.

In this paper we also surveyed several approaches for providing consistent shared memory in more dynamic systems, that is, where nodes may not only crash or depart voluntarily, but where new nodes may join, and where servers in one configurations can be replaced with entirely new configurations. Providing efficient atomic implementations remains challenging for dynamic settings. Here the expectation is that solutions are found by integrating static algorithms with a reconfiguration framework so that during periods of relative stability one benefits from the efficiency of static algorithms, and where during the more turbulent times performance degrades gracefully when reconfigurations are needed. One of the open questions here is whether consensus is truly necessary for implementing consistent memory services for long-lived dynamic systems.

The technical challenges and performance overheads in the dynamic setting may be the reasons why the existing distributed storage solutions shy away from atomic consistency guarantees. Commercial solutions, such as Google's File System (GFS) [27], Amazon's Dynamo [15], and Facebook's Cassandra [34], provide less-than-intuitive, unproved guarantees. The concepts discussed in section 5 are echoed in the design decisions of production systems. For instance, consensus is used in GFS [27] to ensure agreement on system configuration as it is done in RAMBO; global time is used in Spanner [13] as it is done in *GeoQuorums*; replica access protocols in Dynamo [15] use quorums as in some approaches surveyed here. These examples provide motivation for pursuing rigorous algorithmic approaches in the study of consistent data services for dynamic networked systems. For a more detailed discussion, we direct the interested reader to related work that surveys atomic shared implementations for dynamic settings [42].

Consistent storage systems continues to be an area of active research and advanced development, and there are good reasons to believe that as high performance memory systems with superior fault-tolerance become available, they will play a significant role in the construction of sophisticated distributed applications. The demand for implementations providing atomic read/write memory will ultimately be driven by the needs of distributed applications that require provable consistency and performance guarantees.

References

1 Marcos K. Aguilera, Idit Keidar, Dahlia Malkhi, and Alexander Shraer. Dynamic atomic storage without consensus. *J. ACM*, 58:7:1–7:32, April 2011. doi:10.1145/1944345.1944348.

2 Marcos K. Aguilera, Idit Keidary, Dahlia Malkhi, Jean-Philippe Martin, and Alexander Shraery. Reconfiguring replicated atomic storage: A tutorial. *Bulletin of the EATCS*, 102:84–081, 2010.

3 Antonio Fernández Anta, Theophanis Hadjistasi, and Nicolas C. Nicolaou. Computationally light "multi-speed" atomic memory. In *20th International Conference on Principles of Distributed Systems, OPODIS 2016, December 13-16, 2016, Madrid, Spain*, pages 29:1–29:17, 2016. doi:10.4230/LIPIcs.OPODIS.2016.29.

4 Antonio Fernández Anta, Nicolas C. Nicolaou, and Alexandru Popa. Making "fast" atomic operations computationally tractable. In *19th International Conference on Principles of Distributed Systems, OPODIS 2015, December 14-17, 2015, Rennes, France*, pages 19:1–19:16, 2015. doi:10.4230/LIPIcs.OPODIS.2015.19.

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

6 Hagit Attiya, Amotz Bar-Noy, and Danny Dolev. Sharing memory robustly in message-passing systems. *J. ACM*, 42(1):124–142, 1995. doi:10.1145/200836.200869.

7 Hagit Attiya and Jennifer L. Welch. Sequential consistency versus linearizability. *ACM Trans. Comput. Syst.*, 12(2):91–122, 1994. doi:10.1145/176575.176576.

8 Ken Birman. A history of the virtual synchrony replication model. In *Replication: Theory and Practice*, volume 5959 of *Lecture Notes in Computer Science*, pages 91–120, 2010.

9 Ken Birman, Dahlia Malkhi, and Robbert Van Renesse. Virtually synchronous methodology for dynamic service replication. Technical report, MSR-TR-2010-151, Microsoft Research, 2010.

10 Brad Calder, Ju Wang, Aaron Ogus, Niranjan Nilakantan, Arild Skjolsvold, Sam McKelvie, Yikang Xu, Shashwat Srivastav, Jiesheng Wu, Huseyin Simitci, Jaidev Haridas, Chakravarthy Uddaraju, Hemal Khatri, Andrew Edwards, Vaman Bedekar, Shane Mainali, Rafay Abbasi, Arpit Agarwal, Mian Fahim ul Haq, Muhammad Ikram ul Haq, Deepali Bhardwaj, Sowmya Dayanand, Anitha Adusumilli, Marvin McNett, Sriram Sankaran, Kavitha Manivannan, and Leonidas Rigas. Windows azure storage: a highly available cloud storage service with strong consistency. In Ted Wobber and Peter Druschel, editors, *Proceedings of the 23rd ACM Symposium on Operating Systems Principles 2011, SOSP 2011, Cascais, Portugal, October 23-26, 2011*, pages 143–157. ACM, 2011. doi:10.1145/2043556.2043571.

11 Tushar Deepak Chandra, Vassos Hadzilacos, and Sam Toueg. An algorithm for replicated objects with efficient reads. In *Proceedings of the 2016 ACM Symposium on Principles of Distributed Computing, PODC 2016, Chicago, IL, USA, July 25-28, 2016*, pages 325–334, 2016. doi:10.1145/2933057.2933111.

12 Gregory Chockler, Seth Gilbert, Vincent Gramoli, Peter M. Musial, and Alexander A. Shvartsman. Reconfigurable distributed storage for dynamic networks. *Journal of Parallel and Distributed Computing*, 69(1):100–116, 2009.

13 James C. Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, J. J. Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, Wilson C. Hsieh, Sebastian Kanthak, Eugene Kogan, Hongyi Li, Alexander Lloyd, Sergey Melnik, David Mwaura, David Nagle, Sean Quinlan, Rajesh Rao, Lindsay Rolig, Yasushi Saito, Michal Szymaniak, Christopher Taylor, Ruth Wang, and Dale Woodford. Spanner: Google's globally-distributed database. In *10th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2012, Hollywood, CA, USA, October 8-10, 2012*, pages 261–264, 2012. URL: https://www.usenix.org/conference/osdi12/technical-sessions/presentation/corbett.

14 Roberto De Prisco, Alan Fekete, Nancy A. Lynch, and Alexander A. Shvartsman. A dynamic primary configuration group communication service. In *Proc. of the 13th Int-l Symposium on Distributed Computing*, pages 64–78. Springer-Verlag, 1999. URL: http://dl.acm.org/citation.cfm?id=645956.675955.

15 Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: amazon's highly available key-value store. *SIGOPS Oper. Syst. Rev.*, 41(6):205–220, 2007. doi:10.1145/1323293.1294281.

16 S. Dolev, S. Gilbert, N. Lynch, A. Shvartsman, and J. Welch. Geoquorums: Implementing atomic memory in mobile ad hoc networks. In *Proceedings of 17th International Symposium on Distributed Computing (DISC)*, 2003.

17 Partha Dutta, Rachid Guerraoui, Ron R. Levy, and Arindam Chakraborty. How fast can a distributed atomic read be? In *Proceedings of the 23rd ACM symposium on Principles of Distributed Computing (PODC)*, pages 236–245, 2004.

18 Burkhard Englert, Chryssis Georgiou, Peter M. Musial, Nicolas Nicolaou, and Alexander A. Shvartsman. On the efficiency of atomic multi-reader, multi-writer distributed memory. In *Proceedings 13th International Conference On Principle Of DIstributed Systems (OPODIS 09)*, pages 240–254, 2009.

19 Alan Fekete, Nancy A. Lynch, and Alexander A. Shvartsman. Specifying and using a partitionable group communication service. *ACM Trans. Comput. Syst.*, 19(2):171–216, 2001. doi:10.1145/377769.377776.

20 Michael J. Fischer, Nancy A. Lynch, and Michael S. Paterson. Impossibility of distributed consensus with one faulty process. *Journal of ACM*, 32(2):374–382, 1985. doi:10.1145/3149.214121.

21 Chryssis Georgiou, Theophanis Hadjistasi, Nicolas C. Nicolaou, and Alexander A. Schwarzmann. Unleeshing and speeding up readers in atomic object implementations. In *Networked Systems - 6th International Conference, NETYS 2018, Essaouria, Morocco, May 9-11, 2018, Proceedings*, 2018.

22 Chryssis Georgiou, Peter M. Musial, and Alex A. Shvartsman. Long-lived RAMBO: Trading knowledge for communication. *Theoretical Computer Science*, 383(1):59–85, 2007.

23 Chryssis Georgiou, Peter M. Musial, and Alexander A. Shvartsman. Developing a consistent domain-oriented distributed object service. *IEEE Transactions of Parallel and Distributed Systems (TPDS)*, 20(11):1567–1585, 2009. A preliminary version of this work appeared in the proceedings of the 4th IEEE International Symposium on Network Computing and Applications (NCA'05).

24 Chryssis Georgiou, Nicolas Nicolaou, Alexander Russel, and Alexander A. Shvartsman. Towards feasible implementations of low-latency multi-writer atomic registers. In *10th Annual IEEE International Symposium on Network Computing and Applications*, 2011.

25 Chryssis Georgiou, Nicolas C. Nicolaou, and Alexander A. Shvartsman. On the robustness of (semi) fast quorum-based implementations of atomic shared memory. In *DISC '08: Proceedings of the 22nd international symposium on Distributed Computing*, pages 289–304, Berlin, Heidelberg, 2008. Springer-Verlag. doi:10.1007/978-3-540-87779-0_20.

26 Chryssis Georgiou, Nicolas C. Nicolaou, and Alexander A. Shvartsman. Fault-tolerant semifast implementations of atomic read/write registers. *Journal of Parallel and Distributed Computing*, 69(1):62–79, 2009. doi:10.1016/j.jpdc.2008.05.004.

27 Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The google file system. In *Proceedings of the nineteenth ACM symposium on Operating systems principles*, SOSP '03, pages 29–43, New York, NY, USA, 2003. ACM. doi:10.1145/945445.945450.

28 S. Gilbert, N. Lynch, and A. Shvartsman. RAMBO: A robust, reconfigurable atomic memory service for dynamic networks. *Distributed Computing*, 23(4):225–272, December 2010.

29 Vincent Gramoli, Peter M. Musial, and Alexander A. Shvartsman. Operation liveness and gossip management in a dynamic distributed atomic data service. In *Proceedings of the ISCA 18th International Conference on Parallel and Distributed Computing Systems, September 12-14, 2005 Imperial Palace Hotel, Las Vegas, Nevada, US*, pages 206–211, 2005.

30 Theophanis Hadjistasi, Nicolas C. Nicolaou, and Alexander A. Schwarzmann. Oh-ram! one and a half round atomic memory. In *Networked Systems - 5th International Conference, NETYS 2017, Marrakech, Morocco, May 17-19, 2017, Proceedings*, pages 117–132, 2017. doi:10.1007/978-3-319-59647-1_10.

31 Maurice Herlihy. Wait-free synchronization. *ACM Trans. Program. Lang. Syst.*, 13(1):124–149, 1991. doi:10.1145/114005.102808.

32 Maurice P. Herlihy and Jeannette M. Wing. Linearizability: a correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 12(3):463–492, 1990. doi:10.1145/78969.78972.

33 Kishori M. Konwar, Peter M. Musial, Nicolas C. Nicolaou, and Alexander A. Shvartsman. Implementing atomic data through indirect learning in dynamic networks. In *Sixth IEEE International Symposium on Network Computing and Applications (NCA 2007), 12 - 14 July 2007, Cambridge, MA, USA*, pages 223–230, 2007. doi:10.1109/NCA.2007.30.

34 Avinash Lakshman and Prashant Malik. Cassandra: a decentralized structured storage system. *SIGOPS Oper. Syst. Rev.*, 44(2):35–40, 2010. doi:10.1145/1773912.1773922.

35 L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess progranm. *IEEE Trans. Comput.*, 28(9):690–691, 1979. doi:10.1109/TC.1979.1675439.

36 Leslie Lamport. The part-time parliament. *ACM Transactions on Computer Systems*, 16(2):133–169, 1998. doi:10.1145/279227.279229.

37 Barbara Liskov. The power of abstraction. In Nancy A. Lynch and Alexander A. Shvartsman, editors, *Distributed Computing, 24th Int-l Symposium, DISC 2010 Proc.*, volume 6343 of *LNCS*. Springer, 2010. doi:10.1007/978-3-642-15763-9.

38 Michael C. Loui and Hosame H. Abu-Amara. Memory requirements for agreement among unreliable asynchronous processes. In Franco P. Preparata, editor, *Parallel and Distributed Computing*, volume 4 of *Advances in Computing Research*, pages 163–183. JAI Press, Greenwich, Conn., 1987.

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

40 Nancy A. Lynch and Alexander A. Shvartsman. Robust emulation of shared memory using dynamic quorum-acknowledged broadcasts. In *Proceedings of Symposium on Fault-Tolerant Computing*, pages 272–281, 1997.

41 Achour Mostéfaoui and Michel Raynal. Time-efficient read/write register in crash-prone asynchronous message-passing systems. In *Networked Systems - 4th International Conference, NETYS 2016, Marrakech, Morocco, May 18-20, 2016, Revised Selected Papers*, pages 250–265, 2016. `doi:10.1007/978-3-319-46140-3_21`.

42 Peter M. Musial, Nicolas C. Nicolaou, and Alexander A. Shvartsman. Implementing distributed shared memory for dynamic networks. *Commun. ACM*, 57(6):88–98, 2014. `doi:10.1145/2500874`.

43 René Peeters. The maximum edge biclique problem is np-complete. *Discrete Applied Mathematics*, 131(3):651–654, 2003. `doi:10.1016/S0166-218X(03)00333-0`.

44 Marko Vukolic. *Quorum Systems: With Applications to Storage and Consensus*. Synthesis Lectures on Distributed Computing Theory. Morgan & Claypool Publishers, 2012. `doi:10.2200/S00402ED1V01Y201202DCT009`.