# Limitations Imposed by the Multi-Writer Setting on the Fastness of Read/Write Atomic Register Implementations *

Chryssis Georgiou † Nicolas C. Nicolaou† ‡

**Abstract**

We survey recent literature to determine the limitations that a multi-writer multi-reader (MWMR) setting may impose on the operation latency of read/write atomic register implementations under crash-prone processes. We first present the challenges for devising atomic register implementations in an asynchronous, message passing, failure-prone environment. Then we present the techniques proposed to implement an atomic read/write register and examine operation latency boundaries on such techniques. In this survey we consider algorithms designed for both SWMR and MWMR settings.

Technical Report TR-10-04
Department of Computer Science
University of Cyprus
December 2010

*This work is supported by the Cyprus Research Promotion Foundation's grant ΠΕΝΕΚ/0609/31 and the European Regional Development Fund

†Department of Computer Science, University of Cyprus, Cyprus. Email: {chryssis,nicolasn}@cs.ucy.ac.cy.

‡Department of Computer Science and Engineering, University of Connecticut, CT, USA.

# 1 Challenges in Implementing Atomic Read/Write Registers

Availability of network storage technologies (e.g., SAN, NAS [16]) and cheap commodity disks increased the popularity of reliable distributed storage systems. To ensure data availability and survivability, such systems replicate the data among multiple basic storage units – disks or servers. A popular method for data replication and maintenance uses redundant arrays of independent disks (RAID) [4, 31]. Although a RAID system may sometimes offer both performance boosting and data availability, it usually resides in a single physical location, is controlled via a single disk controller, and is connected to the clients via a single network interface. Thus, this single physical location with its single interface constitutes a single point of failure and a performance bottleneck. In contrast, a distributed storage system implements reliable data storage by replicating data in geographically dispersed nodes, ensuring data survivability even in cases of complete site disasters. Researchers often focus on implementing abstract objects that allow primitive operations, like read and write registers. Read/write registers can be used as building blocks for more complex storage systems or to directly implement file storage systems, making them interesting in their own right.

A distributed read/write register implementation involves two distinct sets of participating entities: the *replica hosts* and the *clients*. Each replica host maintains a copy of the replicated register. Each client is a *reader* or a *writer* and performs *read* or *write* operations on the register, respectively. In the message-passing environment, clients access the replicated register by exchanging messages with the replica hosts. A reader performs a read operation as follows: (i) accepts a read request from its environment, (ii) exchanges messages with the replica hosts to obtain the value of the register, and (iii) returns the value discovered to the environment. Similarly, a writer performs a write operation as follows: (i) accepts a value to be written on the register, (ii) exchanges messages with the replica hosts to write this value on the register, and (iii) reports completion to the environment.

Replication allows several clients to access different replicas of the register concurrently, leading to challenges in guaranteeing replica consistency. To define the exact operation guarantees in situations where the register can be accessed concurrently, researchers introduced different *consistency models*. The strongest consistency model is *atomicity* that provides the illusion that operations are performed in a sequential order, when in reality they are performed concurrently. In addition to atomicity, atomic register implementations must ensure *fault-tolerance*. That is, any operation that is invoked by a non-faulty client terminates, despite the failures in the system.

Two obstacles in implementing an atomic read/write register are *asynchrony* and *failures*. A *communication round-trip* (or simply round) between two participants A and B, involves a message sent by A to B, then a message sent by B to A. Due to asynchrony, every message sent between two participants experiences an unpredictable *communication delay*. As a result, a communication round-trip involves two communication delays. To obtain the value of the register during a read operation, a reader requires at least one round and thus two communication delays for: (a) delivery of a read message from the reader to at least a single replica host, and (b) delivery of the reply from the replica host to the reader. Similarly, to modify the value of the register during a write operation, a writer requires at least one round and thus two communication delays for: (a) delivery of a write message from the writer to at least a single replica host, and (b) delivery of an acknowledgment from the replica host to the writer. Although the writer may enclose the value to be written in its write message, the write operation cannot terminate before receiving an acknowledgment from the replica host. In fact, this could lead to the termination of the write operation before the replica host receives the write message, either due to delay or due to replica host failure. In any case, atomicity may be violated as a subsequent operation will be unaware of the existence of the write operation. Consequently, both read and write operations require at least *two* communication delays, that is, a *single round* before terminating. We refer to operations that terminate after their first round as *fast*.

Fault-tolerance is not guaranteed if an operation communicates with a single replica host. A

crash failure may prevent the delivery of messages to that host, keeping clients waiting for a reply and preventing them from terminating. Additionally, if two operations communicate with different replica hosts, they may observe different replica values, thus atomicity may be violated, as the second operation may return an older value than the one written or read by the first operation. Therefore, a client needs to send messages to a *subset* of replica hosts. To tolerate failures, such a subset should contain more replica hosts than the maximum number of allowed replica host failures. Moreover, to ensure that operations are aware of each other they must obtain information from overlapping subsets of replica hosts.

Communicating with overlapping subsets of replicas may be insufficient to guarantee atomicity. Suppose a write operation communicates with a subset $A$ and a succeeding read operation with a subset $B \neq A$ where $A \cap B \neq \emptyset$. The read operation obtains the value written from the replica hosts in the intersection $A \cap B$. As the read succeeds the write, it returns the value written. Consider, a different scenario where the write operation is delayed and communicates only with the replica hosts in $A \cap B$ before the read communicates with the replica hosts in $B$. The read operation cannot differentiate the two scenarios and thus returns the value being written. A second read operation may communicate with a subset $C$, such that $A \cap C \neq \emptyset$, $B \cap C \neq \emptyset$, and $A \cap B \cap C = \emptyset$. Thus, the read is not aware of the delayed write and hence returns an older value, violating atomicity. To ensure that any succeeding operation observes the written value, the first read operation can either: (i) ensure that the written value is propagated to enough replica hosts by waiting for hosts not in $A$ to reply, or (ii) propagate the value to a subset of replica hosts that overlaps with the subset obtained by any subsequent operation. As hosts not in $A$ may crash, waiting for more replies may prevent the read operation from terminating. So it remains for the read operation to perform another round to propagate the written value. As a result, atomic register implementations may contain operations that may experience four communication delays before terminating.

In general the efficiency of atomic read/write register implementations is measured in terms of the latency of read and write operations. The latency of an operation is affected by two factors: (a) *communication*, and (b) *computation*.

**Document Structure:** The rest of the report presents the current research in distributed systems regarding implementations of consistent distributed read/write (R/W) storage objects. We begin with an overview of the *consistency semantics* in Section 2. In Section 3 we talk about mathematical tools that were extensively used for implementing atomic register implementatios, called *Quorum Systems*. In Section 4, we discuss implementations that establish consistent distributed storage in message-passing, failure prone, and asynchronous environments. Finally in Section 5 we enumerate our conclusions from this survey on the limitations that a MWMR setting may impose on the operation latency of atomic R/W register implementations.

## 2 Consistency Semantics

Lamport in [24], defined three consistency semantics for a R/W register abstraction in the SWMR environment: *safe*, *regular*, and *atomic*.

The *safe register* semantic ensures that if a read operation is not concurrent with a write operation, it returns the last value written on the register. Otherwise, if the read is concurrent with some write, it returns any arbitrary value that is allowed to be written to the register. The latter property renders this consistency semantic insufficient for a distributed storage system: a read operation that is concurrent with some write may return a value that was *never* written on the register.

A stronger consistency semantic is the *regular register*. As in the safe register, regularity ensures that a read operation returns the latest written value if the read is not concurrent with a write. In the event of read and write concurrency, the read returns either the value written by the last preceding

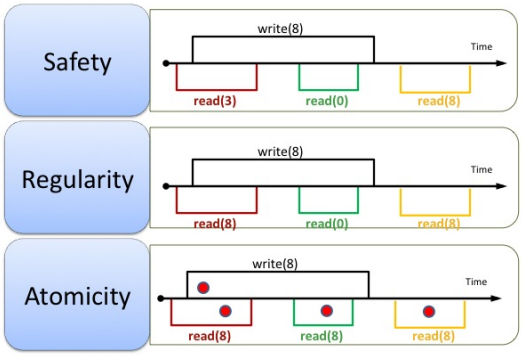


Figure 1: Safe, Regular, and Atomic semantics among a set of read/write operations.

write operation, or the value written by the concurrent write. In any case, regularity guarantees that a read returns a value that is written on the register, and is not older than the value written by the read's last preceding write operation.

Although regularity is sufficient for many applications that exploit distributed storage systems, it does not provide the consistency guarantees of a traditional sequential storage. In particular, it does not ensure that two read operations overlapping the same write operation will return values as if they were performed sequentially. If the two reads do not overlap then regularity allows the succeeding read to return an older value than the one returned by the first read. This is known as new-old read inversion. *Atomic semantics* overcome this problem by ensuring that a read operation does not return an older value than the one returned by a preceding read operation. In addition, it preserves all the properties of the regular register. Thus, atomicity provides the illusion that operations are ordered sequentially. Figure 1 provides a pictorial of an example that demonstrates the difference between the three consistency semantics.

Herlihy and Wing in [21] introduce *linearizability*, generalizing the notion of atomicity to any type of distributed object. That same paper presented two important properties of linearizability: *locality* and *non-blocking*. These properties distinguish linearizability from correctness conditions like *sequential consistency* (introduced by Lamport in [23]) and *seriazability* (introduced by Papadimitriou in [30]). An in-depth comparison between sequential consistency and linearizability was conducted by Attiya and Welch in [3]. As defined in [21], a property $P$ of a concurrent system is *local* if the system satisfies $P$ whenever each individual object satisfies $P$. Thus, locality allows a system to be linearizable as long as every individual object of the system is linearizable. Non-blocking allows processes to complete some operation without waiting for any other operation to complete. *Wait-freedom*, is stronger than non-blocking, and is defined by Herlihy in [20]: any process completes an operation in a finite number of steps regardless of the operation conducted by other processes. While wait-freedom ensures non-blocking on an operation level, weakest non-blocking progress conditions guarantee only that *some* (and not *all*) operation complete in finite number of steps (*lock-freedom*) or require conflicting operations to abort and retry (*obstruction-freedom*). Both non-blocking and locality properties enhance concurrency. Also, locality improves modularity (since every object can be verified independently), and non-blocking favors the use of linearizability in time critical applications.

Subsequent works revisited and redefined the definitions provided in [24, 21] for more specialized distributed systems. Lynch [26] provided an equivalent definition of atomicity of [24] to describe atomic R/W objects in the MWMR environment. This definition, totally orders write operations, and partially orders read operations with respect to the write operations.


(a) (b)

Figure 2: Examples of quorums systems: (a) Majority (b) Matrix.

## 3 Quorum Systems

Intersecting collections of sets can be used to achieve synchronization and coordination of concurrent accesses on distributed data objects. A *Quorum System* is a collection of sets known as *quorums*, such that every pair of such sets intersects. More specifically, given a quorum system $\mathbb{Q}$, for every two quorums $Q_1, Q_2 \in \mathbb{Q}$, it holds that $Q_1 \cap Q_2 \neq \emptyset$.

Several families of quorums systems have been defined [17, 33, 11, 35, 32, 29]. Figure 2 depicts examples of the majority [17] and matrix [35] quorums systems. In the latter, elements are placed on a virtual matrix, and a quorum is defined as a row and a column of the matrix. Observe that every two quorums share at least two elements, but they might share much more (e.g., a whole row or column).

Gifford [17] and Thomas [33] used quorums to achieve mutual exclusion on concurrent file and database access control respectively. Garcia-Molina and Babara [11], compared the counting (or vote assignment) strategy presented in [17, 33], with a strategy that explicitly defines a priori the set of intersecting groups (i.e., the quorum system). Their investigation revealed that although the two strategies appear to be similar, they are not equivalent since one may devise quorum systems for which there exist no vote assignment. Following this finding, quorum systems for distributed services adhere to one of the following design principles:

- **Voting:** Quorums are defined by the number of distributed objects collected during an operation.
- **Explicit Quorums:** Quorum formulation is specified before the deployment and use of the quorum system.

As efficient tools for collaboration and coordination, quorums attracted the attention of researchers studying implementations of distributed shared memory. Upfal and Wigderson in [34], introduced an atomic emulation of a synchronous R/W shared memory model, where a set of processes shared a set of data items. To allow faster discovery of a single data item and fault-tolerance, the authors suggested its replication among several memory locations. Retrieval (read) or modification (write) of the value of a data item involved the access of the *majority* of the replicas. The authors exploited coordination mechanisms to allow only a single read or write operation per data item at a time. This work was the first to introduce and use $\langle value, timestamp \rangle$ pairs to order the written values, where $timestamp \in \mathbb{N}$.

Vitanyi and Awerbuch in [35] give an implementation of atomic shared memory for the MWMR environment under asynchrony. Their work organized the register replicas in an $n \times n$ matrix construction, where $n$ is the number of client processes. A process $p_i$ is allowed to access the distinct $i^{th}$ row and distinct $i^{th}$ column per write and read operation respectively. This strategy allows reads to be aware of any preceding write due to the intersection of any row with any column of the matrix. To accommodate concurrent write operations, the authors use $\langle value, tag \rangle$ pairs to order the written values. A *tag* is a tuple of the form $\langle timestamp, WID \rangle$, where the $timestamp \in \mathbb{N}$ and $WID$ is a writer identifier. Tags are compared lexicographically. Namely, $tag_1 > tag_2$ if either $tag_1.timestamp > tag_2.timestamp$, or $tag_1.timestamp = tag_2.timestamp$ and $tag_1.WID > tag_2.WID$.

# 4 Atomic Memory Implementations

The works presented in [34] and [35] were designed for the synchronous and failure-free environments. These approaches are inapplicable in the asynchronous, failure-prone, message-passing model. As discussed by Chockler et al. in [6], implementations in these environments must be wait-free, tolerate various types of failures, and support concurrent accesses on replicated data.

A seminal paper by Attiya et al. [2] was the first to introduce a solution to the problem, by devising an algorithm that implements a Single-Writer, Multi-Reader (SRMW) atomic read/write register in the asynchronous message-passing model. Their algorithm overcomes crash failures of any subset of readers, the writer, and up to $f$ out of $2f + 1$ replica hosts. The correctness of the algorithm is based on the use of majorities, a quorum construction established by voting. This work adopts the idea of [34] and uses $\langle$*value,timestamp*$\rangle$ pairs to impose a partial order on read and write operations. A write operation, involves a single round: the writer has to increment its local timestamp, associate the new timestamp with the value to be written, and send the new pair to the majority $(f + 1)$ of the replica hosts. A read operation requires two rounds: during the first round the reader collects the timestamp-value pairs from a majority of the replica hosts, discovers the maximum timestamp among those, and propagates (in the second round) the maximum timestamp-value pair to the majority of the replica hosts. Although the value of the read is established after the first round, skipping the second round can lead to violations of atomicity when read operations are concurrent with a write operation.

Lynch and Shvartsman [28] generalized the majority-based approach of [2] to the MWMR environment using quorum systems. To preserve data availability in the presence of failures, the atomic register is replicated among all the service participants. To preserve consistency, they utilize a quorum system (refer to it as *quorum configuration*). This allows read and write operations to terminate a round as soon as the value of the replicated data object (register) was collected from all the members of a single quorum (instead of collecting the majority of the replicas as in [2]). To order the values written, the algorithm utilizes the $\langle$*tag,value*$\rangle$ pairs as those presented by Vitanyi and Awerbuch in [35], and requires every write operation to perform *two* rounds to complete. Read and write operations are implemented symmetrically. In the first round a read (resp. write) obtains the latest $\langle$tag,value$\rangle$ pair from a complete quorum. In the second round, a read propagates the maximum tag-value pair to some complete quorum. A write operation increments the timestamp enclosed in the maximum tag, and generates a new tag including the new timestamp and the writer's identifier. Then, the writer associates the new tag with the value to be written and propagates the tag-value pair to a complete quorum. To enhance longevity of the service the authors in [28] suggest the reconfiguration (replacement) of quorums. Transition from the old to the new configuration could lead to violations of atomicity as operations can communicate with quorums of either the old or the new configuration during that period. Thus, the authors suggest a blocking mechanism to suspend the read and write operations during the transition.

A follow up work by Englert and Shvartsman in [10] made a valuable observation: taking the union of the new with the old configuration defines a valid quorum system. Based on this observation they allow R/W operations to be active during reconfiguration, by requiring that any operation communicates with both new and old configurations. Both [10] and [28] dedicate a single reconfigurer to propose the next replica configuration. The reconfiguration involves three rounds. During the first round the reconfigurer notifies the readers and writers about the new configuration and collects the latest register information. During the second round it propagates the latest register information in the members of the new configuration. Finally, during the third round the reconfigurer acknowledges the establishment of the new configuration. Read and write operations involve two rounds when they do not discover that a reconfiguration is in progress. Otherwise they may involve multiple rounds to ensure that they are going to reach the latest proposed configuration.

A new implementation of atomic registers for dynamic networks, called RAMBO, was developed

by Gilbert, Lynch, and Shvartsman in [18]. The RAMBO approach improves the longevity of implementations in [10, 28], by introducing multiple reconfigurers (and thus circumventing the failure of the single reconfigurer) and a new mechanism to garbage-collect old and obsolete configurations. The new service preserves atomicity while allowing participants to join and fail by crashing. The use of multiple reconfigurers increases the complexity of the reconfiguration process. To enable the existence of multiple reconfigurers, the service incorporates a consensus algorithm (e.g., Paxos by Lamport in [25]) to allow reconfigurers to agree on a consistent configuration sequence.

A string of refinements followed to improve the efficiency and practicality of that service. Gramoli et al. in [19] reduce the communication cost of the service and locally optimize the liveness of R/W operations. To improve reconfiguration and operation latency, Chockler et al. in [5], propose the incorporation of an optimized consensus protocol, based on Paxos. Aiming to improve the longevity of RAMBO, Georgiou et al. in [12] implement graceful participant departures. They also deploy an incremental gossip protocol that reduce dramatically the message complexity of RAMBO, both with respect to the number of messages and the message size. The same authors in [13] combine multiple instances of the service to compose a complete shared memory emulation. To decrease the communication complexity of the service, Konwar et al. in [22] suggest the departure from the all-to-all gossiping in RAMBO, and propose an indirect communication scheme among the participants. Retargetting [27] to ad-hoc mobile networks, Dolev et al. in [7] formulate the GeoQuorums approach where replicas are maintained by stationary *focal points* that in turn were implemented by mobile nodes. To expedite write operations, the algorithm relies on a global positioning system (GPS) [1] clock to order the written values. A write operation terminates in a *single* round by associating the value to be written with the time obtained from the GPS service.

### Fastness in Atomic Memory Implementations

Following the development in [2], a folklore belief formed that "atomic reads must write", i.e., a read operation needs to perform a second round. If that second round is avoided then atomicity may be violated: a read operation may return an older value than the one returned by a preceding read operation.

Dolev et al. in [7] introduced single round read operations in the MWMR environment. According to their approach – later used by Chockler et al. in [5] – a read operation could return a value in a single round when it was confirmed that the write phase that propagated that value completed. To assess the status of each write operation the algorithm associated a binary variable, called *confirmed*, with each tag. A participant would set this variable for a tag $t$ in two cases: (i) it completed a write phase and propagated a value associated with $t$ to a full quorum, or (ii) it discovered that $t$ was marked as confirmed by some other participant. A read operation can complete in a single round if the largest discovered tag is marked as confirmed. This can happen iff some write phase that propagated the tag completed. Despite the improvement achieved in the operation latency in [7, 5], this strategy is unable to overcome the problem presented in [2]: every read operation requires a second round– and thus a "write" – whenever it is concurrent with a write operation.

Dutta et al. in [8] are the first to present *fast operations* that are not affected by read and write concurrency. Assuming the SWMR environment the authors establish that if the number of readers is appropriately constrained with respect to the number of replicas, then implementations that contain *only* single round reads and writes, called *fast*, are possible. The register is replicated among a set $\mathcal{S}$ of replica hosts (servers), out of which $f < \frac{|\mathcal{S}|}{2}$ (the minority) is allowed to crash. To implement fast writes, the algorithm adopts the write protocol in [2] and involves the use of $\langle$*timestamp, value*$\rangle$ pairs to order the written values. The only difference is that the write operation propagates the written value to $|\mathcal{S}| - f$ servers, instead of a strict majority of $\frac{|\mathcal{S}|}{2} + 1$ required in [2]. The main departure of the new algorithm involves the server and reader implementations. In particular, each server maintains a bookkeeping mechanism to record any reader that inquires its local timestamp-value pair. This

information is enclosed in every message sent by the server to any requested operation. The recorded information is utilized by the readers to achieve fast read operations. The read protocol requires the reader to send messages to all the servers, and wait for $|\mathcal{S}| - f$ replies. When those replies are received, the reader discovers the maximum timestamp ($maxTs$) among the replies, and collects all the messages that contain that timestamp. Then, a predicate is applied over the bookkeeping information contained in those messages. If the predicate holds, the reader returns the value associated with $maxTs$; otherwise it returns the value associated with the previous timestamp ($maxTS - 1$). Note that the safety of the algorithm in the latter case is preserved because of the single writer and the assumption that a process can invoke a single operation at a time. Thus, the initiation of the write operation with $maxTs$ implies that the write operation with timestamp $maxTs - 1$ has already been completed. On the other hand, if the read operation decides to return $maxTs$ then the validation of the read predicate ensures the safety of the algorithm. The predicate is based on the following key observation: the number of servers that reply with $maxTs$ to any two subsequent read operations may differ by at most $f$. The authors show that fast operations are only possible if the number of readers is $R < \frac{|\mathcal{S}|}{f} - 2$. It is also shown that fast implementations are *impossible* in the MWMR environment even assuming two writers, two readers, and a single server crash.

Georgiou et al. [15] tried to relax the constraint on the number of readers of [8]. To this end, the authors traded the operation latency of read operations for scalability. In particular, they define the notion of *semifast* implementations where a single read operation per write needs to be "slow" (perform two rounds). To allow unbounded number of readers, they introduced the notion of *virtual nodes*. Each virtual node serves as an enclosure for multiple reader participants. Adapting the techniques presented in [8], each virtual node is treated as a separate participating entity, allowed to perform read operations. Their algorithm requires that the number of virtual nodes $V$ is inferior to $\frac{S}{t} - 2$; this does not prevent multiple readers as long as at least one virtual node exists. Finally, the authors show that it is *impossible* to devise semifast MWMR implementations.

Georgiou et al. [14] showed that fast and semifast quorum-based SWMR implementations are possible if and only if a common intersection exists among all quorums. Hence a single point of failure exists in such solutions (i.e., any server in the common intersection), making such implementations not fault-tolerant. To trade efficiency for improved fault-tolerance, *weak-semifast* implementations in [14] require at least one single slow read per write operation, and where all writes are fast. To obtain a weak-semifast implementation they introduced a client-side decision tool called *Quorum Views* that enables fast read operations under read/write concurrency when *general quorum systems* are used.

Recently, Englert *et al.* [9] developed an atomic MWMR register implementation, called algorithm SFW, that allows both reads and writes to complete in a *single round*. To handle server failures, their algorithm uses *n-wise quorum systems*: a set of subsets of servers, such that each $n$ of these subsets intersect. The parameter $n$ is called the *intersection degree* of the quorum system. The algorithm relies on $\langle tag, value \rangle$ pairs to totally order write operations. In contrast with traditional approaches, the algorithm uses the *server side ordering* (SSO) approach that transfers the responsibility of incrementing the tag from the writers to the servers. This way, the *query* round of write operations is eliminated. The authors proved that fast MWMR implementations are possible if and only if they allow not more than $n-1$ successive write operations, where $n$ is the intersection degree of the quorum system. If read operations are also allowed to modify the value of the register then from the provided bound it follows that a fast implementation can accommodate up to $n - 1$ readers and writers.

# 5 Conclusion: Implications of the MWMR setting on Atomic Memory Implementations

In summary our findings suggest that:

1. Implementations where all operations are fast, i.e. complete in a single communication round, are not possible in the MWMR setting,

2. Implementations that allow a single slow, two communication round read, are also not possible in the MWMR setting, and

3. If $n$ is the intersection degree of the underlying quorum system, then (i) a MWMR safe register implementation may allow up to $n-1$ successive fast write operations, and (ii) no fast safe register implementation can be obtained if more than $n-1$ readers and writers participate in the service.

From the above observations we may conclude that fast operations in the MWMR setting are affected both by the construction of the quorum system they use, as well as by the number of reader and writer participants. Algorithm SFW developed in [9] satisfies the above restrictions.

As SFW is the only known algorithm to allow fast reads and writes in the MWMR setting, an interesting research direction is to explore and address the algorithm's weaknesses. The algorithm is close to optimal in terms of the number of successive single round operations it allows: $\frac{n}{2}$ successive fast write operations where $n$ the intersection degree of the underlying quorum system. To achieve such performance the authors relied on two predicates. These predicates have the following weaknesses:

- they are computationally demanding, and
- rely on the construction of the underlying quorum system

Up to this point researchers focused on reducing the number of communication rounds that each operation needed to perform. Practicality, however, is negatively affected if computation costs are higher than communication costs. Thus, it is essential to analyze the computation burden of algorithm SFW and attempt to devise algorithms to improve its overall (computation + communication) performance.

Moreover, SFW is unable to validate the predicates when the intersection degree of the underlying quorum system is lower than 4. In other words, SFW is unable to guarantee any fast operations when we deploy traditional quorum systems with pairwise intersections. So it is interesting to explore whether we can devise algorithms that do not depend on the construction of the underlying quorum system and they still allow some operations to be fast. Such algorithms would substantially improve the practicality of MWMR atomic register implementations.

**Acknowledgments.** We thank Alexander A. Shvartsman and Alexander C. Russell for several discussions.

# References

[1] Global positioning system (GPS). http://www.gps.gov/.

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

[3] Attiya, H., and Welch, J. L. Sequential consistency versus linearizability. *ACM Trans. Comput. Syst. 12*, 2 (1994), 91–122.

[4] Chen, P. M., Lee, E. K., Gibson, G. A., Katz, R. H., and Patterson, D. A. Raid: high-performance, reliable secondary storage. *ACM Computing Surveys 26*, 2 (1994), 145–185.

[5] Chockler, G., Gilbert, S., Gramoli, V., Musial, P. M., and Shvartsman, A. A. Reconfigurable distributed storage for dynamic networks. *Journal of Parallel and Distributed Computing 69*, 1 (2009), 100–116.

[6] Chockler, G., Keidar, I., Guerraoui, R., and Vukolic, M. Reliable distributed storage. *IEEE Computer* (2008).

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

[8] Dutta, P., Guerraoui, R., Levy, R. R., and Chakraborty, A. How fast can a distributed atomic read be? In *Proceedings of the 23rd ACM symposium on Principles of Distributed Computing (PODC)* (2004), pp. 236–245.

[9] Englert, B., Georgiou, C., Musial, P. M., Nicolaou, N., and Shvartsman, A. A. On the efficiency of atomic multi-reader, multi-writer distributed memory. In *Proceedings 13th International Conference On Principle Of DIstributed Systems (OPODIS 09)* (2009), pp. 240–254.

[10] Englert, B., and Shvartsman, A. A. Graceful quorum reconfiguration in a robust emulation of shared memory. In *Proceedings of International Conference on Distributed Computing Systems (ICDCS)* (2000), pp. 454–463.

[11] Garcia-Molina, H., and Barbara, D. How to assign votes in a distributed system. *Journal of the ACM 32*, 4 (1985), 841–860.

[12] Georgiou, C., Musial, P. M., and Shvartsman, A. A. Long-lived RAMBO: Trading knowledge for communication. *Theoretical Computer Science 383*, 1 (2007), 59–85.

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

[14] Georgiou, C., Nicolaou, N. C., and Shvartsman, A. A. 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* (Berlin, Heidelberg, 2008), Springer-Verlag, pp. 289–304.

[15] Georgiou, C., Nicolaou, N. C., and Shvartsman, A. A. Fault-tolerant semifast implementations of atomic read/write registers. *Journal of Parallel and Distributed Computing 69*, 1 (2009), 62–79. A preliminary version of this work appeared in the proceedings 18th ACM Symposium on Parallelism in Algorithms and Architectures (SPAA'06).

[16] Gibson, G. A., and Van Meter, R. Network attached storage architecture. *Commun. ACM 43*, 11 (2000), 37–45.

[17] Gifford, D. K. Weighted voting for replicated data. In *SOSP '79: Proceedings of the seventh ACM symposium on Operating systems principles* (1979), pp. 150–162.

[18] Gilbert, S., Lynch, N. A., and Shvartsman, A. A. Rambo: a robust, reconfigurable atomic memory service for dynamic networks. *Distributed Computing 23*, 4 (2010), 225–272.

[19] Gramoli, V., Musial, P. M., and Shvartsman, A. A. 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* (2005), pp. 206–211.

[20] Herlihy, M. Wait-free synchronization. *ACM Trans. Program. Lang. Syst. 13*, 1 (1991), 124–149.

[21] Herlihy, M. P., and Wing, J. M. Linearizability: a correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems (TOPLAS) 12*, 3 (1990), 463–492.

[22] Konwar, K. M., Musial, P. M., Nicolaou, N. C., and Shvartsman, A. A. 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* (2007), pp. 223–230.

[23] Lamport, L. How to make a multiprocessor computer that correctly executes multiprocess progranm. *IEEE Trans. Comput. 28*, 9 (1979), 690–691.

[24] Lamport, L. On interprocess communication, part I: Basic formalism. *Distributed Computing 1*, 2 (1986), 77–85.

[25] Lamport, L. The part-time parliament. *ACM Transactions on Computer Systems 16*, 2 (1998), 133–169.

[26] Lynch, N. *Distributed Algorithms*. Morgan Kaufmann Publishers, 1996.

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

[28] Lynch, N. A., and Shvartsman, A. A. Robust emulation of shared memory using dynamic quorum-acknowledged broadcasts. In *Proceedings of Symposium on Fault-Tolerant Computing* (1997), pp. 272–281.

[29] Malkhi, D., Reiter, M. K., Wool, A., and Wright, R. N. Probabilistic quorum systems. *Inf. Comput. 170*, 2 (2001), 184–206.

[30] Papadimitriou, C. H. The serializability of concurrent database updates. *Journal of ACM 26*, 4 (1979), 631–653.

[31] Patterson, D. A., Gibson, G., and Katz, R. H. A case for redundant arrays of inexpensive disks (raid). *SIGMOD Rec. 17*, 3 (1988), 109–116.

[32] Peleg, D., and Wool, A. Crumbling walls: A class of high availability quorum systems. In *Proceedings of 14th ACM Symposium on Principles of Distributed Computing (PODC)* (1995), pp. 120–129.

[33] Thomas, R. H. A majority consensus approach to concurrency control for multiple copy databases. *ACM Trans. Database Syst. 4*, 2 (1979), 180–209.

[34] Upfal, E., and Wigderson, A. How to share memory in a distributed system. *Journal of the ACM 34(1)* (1987), 116–127.

[35] Vitanyi, P., and Awerbuch, B. Atomic shared register access by asynchronous hardware. In *Proceedings of 27th IEEE Symposium on Foundations of Computer Science (FOCS)* (1986), pp. 233–243.