Robust Simulation of Shared Memory: 20 Years After*

Hagit Attiya
Department of Computer Science, Technion
and School of Computer and Communication Sciences, EPFL

Abstract

The article explores the concept of simulating the abstraction of a *shared memory* in message passing systems, despite the existence of failures. This abstraction provides an atomic register accessed with read and write operations. This article describes the Attiya, Bar-Noy and Dolev simulation, its origins and generalizations, as well as its applications in theory and practice.

Dedicated to the 60th birthday of Danny Dolev

1 Introduction

In the summer of 1989, I spent a week in the bay area, visiting Danny Dolev, who was at IBM Almaden at that time, and Amotz Bar-Noy, who was a post-doc at Stanford, before going to PODC in Edmonton.[1] Danny and Amotz have already done some work on equivalence between shared memory and message-passing communication primitives [14]. Their goal was to port specific renaming algorithms from message-passing systems [10] to shared-memory systems, and therefore, their simulation was tailored for the specific construct—*stable vectors*–used in these algorithms.

Register constructions were a big fad at the time, and motivated by them, we were looking for a more generic simulation of shared-memory in message passing systems.

*This article is based on my invited talk at SPAA 2008.

[1] During the same visit, Danny and I also worked on the first snapshot algorithm with bounded memory; these ideas, tremendously simplified and improved by our coauthors later, lead to the atomic snapshots paper [2].

In PODC 1990, we have published the fruits of this study in an extended abstract of a paper [8] describing a simulation of a single-writer multi-reader register in a message-passing system. Inspired by concepts from several areas, the paper presented a simple algorithm, later nicknamed the *ABD simulation*, that supports porting of shared-memory algorithms to message-passing systems. The ABD simulation allowed to concentrate on the former model, at least for studying computability issues.

The simulation, vastly extended to handle dynamic changes in the system and adverse failures, served also as a conceptual basis for several storage systems, and for universal service implementations (*state-machine* replication).

In this article, I describe the ABD simulation and its origins, and survey the follow-up work that has spanned from it.

2 Inspirations

This section describes the context of our simulation, discussing specifications, algorithms and simulations, which provided inspiration for our approach. The section also lays down some basic terminology used later in this article.

2.1 Sharing Memory

When we started to work on this research project, we were well-aware of the paper of Upfal and Wigderson [50][2] simulating a *parallel random access machine* (PRAM) [24] on a synchronous interconnect, like the one used, for example, in the NYU Ultracomputer [30].

Upfal and Wigderson assume a synchronous system, which does not allow any failures. The paper further assumes a complete communication graph (clique) or a concentrator graph and shows how to emulate a PRAM step, namely, a permutation where each node reads from some memory location, or writes to some memory location.

Because many nodes may access the same memory location, their simulation replicates the values of memory locations and stores each of them in several places, in order to reduce latency. To pick the correct value, a node accesses a majority of the copies, and the intersection between these majority sets ensures that the correct value is obtained.

Upfal and Wigderson concentrate mostly on avoiding communication bottlenecks, and hence the emphasis in their paper is on spreading copies in a way that

[2] Indeed, our title is a takeoff on the title of their paper.

minimizes the load across nodes. As we shall discuss (Section 3.3), this consideration will make a comeback also in the context of asynchronous, failure-prone systems.

2.2 Handling Asynchrony

In contrast to Upfal and Wigderson, who assumed that nodes do not fail and that communication is synchronous, we were interested in *asynchronous* systems where nodes may fail. Our original simulation assumed that failed nodes simply *crash* and stop taking steps; later work addressed the possibility of malicious, *Byzantine* failures (see Section 4.2).

The combination of failures and asynchrony poses an additional challenge of a possible *partitioning* of the system. It can easily be seen that if more than a majority of the nodes may fail, then two operations may proceed without being aware of each other. This might cause a read operation to miss the value of a preceding write operation. Therefore, the construction assumes that a majority of the nodes do not fail; thus, the number of nodes n is more than twice the number of possible failures f; that is, $n > 2f$.

Armed with this assumption, it is possible to rely on Thomas's majority consensus approach [49] for preserving the consistency of a replicated database.[3]

The *majority consensus* algorithm coordinates updates to the database replica sites, by having sites (nodes) vote on the acceptability of update requests. For a request to be accepted and applied to all replicas, only a majority need approve it; an update is approved only if the information upon which the request is based is valid; validity, in turn, is determined according to version numbers associated with data values.

This indicates that version numbers may provide the key for coping with failures in an asynchronous system.

2.3 Atomic Registers

A critical step in the simulation was deciding which abstraction to use; indeed, Bar-Noy and Dolev simulated a very specific communication construct, which is not sufficiently generic to use in other contexts, while Upfal and Wigderson simulated a full-fledged PRAM with the associated cost and complication. Luckily for us, the hippest research trend at the time were *register constructions*, namely, algorithms to simulate a *register* with certain characteristic out of registers with weaker features.

[3] This paper makes an interesting read because it deals with concurrency control prior to the introduction of the notion of *serializability* [46].

The "holy grail" of this research area was an *atomic multi-writer multi-reader* register, and many papers presented algorithms for simulating it from weaker types of registers. This is a fairly general data structure accessed by read and write operations, allowing all processes to write to and read from the register, and ensuring that operations appear to occur instantaneously (more on this below).

A multi-writer multi-reader atomic register is a very general and convenient to use abstraction. However, since many of these algorithms were wrong, or at best, complicated, we decided to simulate a slightly weaker kind of register, *atomic single-writer multi-reader* register, which can be written only by a single node. This decision turned out to simplify our algorithm considerably; a multi-writer register could still be simulated by porting the shared-memory algorithms. Later, it turned out that simulating an even weaker type of register, with a single writer *and a single reader*, could lead to a more efficient simulation [7].

To simulate a register in a message-passing system, one must provide two procedures: one for *read* and the other for *write*. These procedures translate the operation into a sequence of message sending and receiving, combined with some local computation. When these procedures are executed together by several nodes their low-level events (message sending and receiving) are interleaved, and we need to state their expected results.

The expected behavior in interleaved scenarios is specified through *linearizability* [34]. Like *sequential consistency* [37], linearizability requires the results of operations to be as if they were executed sequentially. Furthermore, linearizability also demands that this order respects the order of *non-overlapping* operations, in which one operation completes before the other starts (we say that the early operation *precedes* the later one, and that the later one *follows* the early one).

3 The ABD Simulation in a Nutshell

One reason the ABD simulation is well-known is due to its simplicity, at least in the unbounded version. This section explains the basic behavior of the algorithm.

We consider a simple system with one node being the *writer* of a register; all other nodes are *readers*. All n nodes also store a copy of the current value of the register; this is done in a separate thread from their roles as a writer or reader.

Each value written is associated with an unbounded version number, an integer denoted *version#*.

To write a value, the writer sends a message write(*val*,*version#*), with the new value and an incremented version number, to all nodes and waits for $n - f$ acknowledgments. Figure 1(a) shows an example of the communication pattern: the writer sends the message to all seven nodes, one node does not receive the message (indicated by a dashed line); of the nodes that receive the message, one does

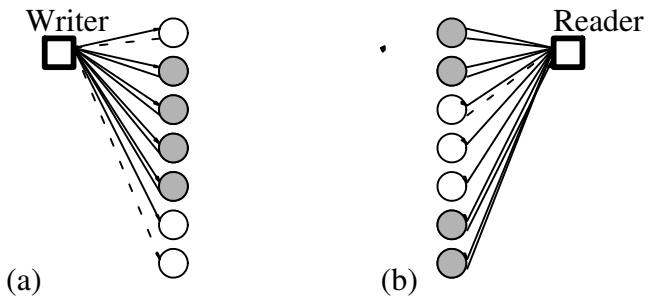

Figure 1: Execution examples for the ABD simulation; dark nodes have acknowledged the message.

not respond at all, another responds but the message is delayed (indicated by a dashed line), so the writer receives acknowledgments from four nodes (a majority out of seven).

To read a value, a reader queries all nodes, and, after receiving at least $n - f$ responses, picks the value with the highest version number. Figure 1(b) shows an example of the communication pattern: the reader sends the message to all seven nodes; all nodes receive the message, two do not respond at all, while another responds but the message is delayed (indicated by a dashed line), so the reader receives values from four nodes (a majority out of seven).

The key to the correctness of the algorithm is to notice that each operation communicates with a majority of the nodes: since $n > 2f$ it follows that $n - f > n/2$. Thus, there is a common node communicating with each pair of write and read operations. (As illustrated by Figure 1.) This node ensures that the value of the latest preceding (non-overlapping) write operation is forwarded to the later read operation; the read will pick this value, unless it receives an even later value (with a higher version number).

A formal proof can be found in the original paper or in [13, Chapter 10].

A slight complication happens because two non-overlapping reads, both overlapping a write, might obtain out-of-order values. This can happen if the writer sends a new value (with a higher version number) to all nodes, and node p gets it first. The early read operation might already obtain the new value from p (among the $n - f$ it waits for), while the later read operation does not wait to hear from p and returns the old value. (See Figure 2.)

This behavior is avoided by having a reader write back the value it is going to return, in order to ensure atomicity of reads.

An argument based on communication with a majority of nodes shows that the value returned by a read operation is available to each following read operation, which returns it unless it has obtained an even later value. (See more details in [8]

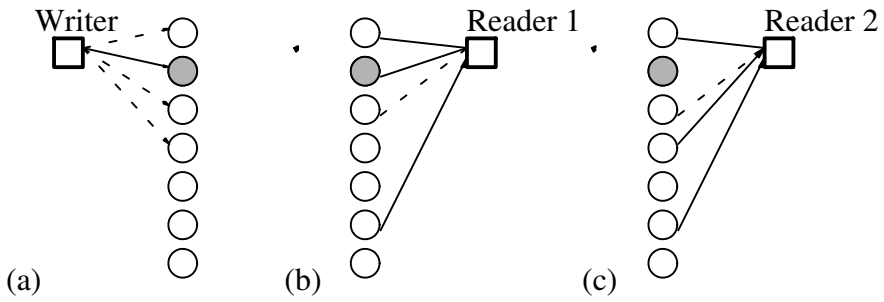

Figure 2: Old-new inversion between read operations; the dark nodes holds the new value of a write operation. In (a), process p already stores the new value of the register; in (b), the first read operation receives the new value, while in (c), a later read operation does receive the new value. Note that the write operation does not complete.

or [13].)

3.1 Bounding the version numbers

A large part of the original paper is spent on bounding the version numbers that are appended to each register value that is sent. The key to bounding the version numbers is knowing which of them are currently in use in the system; once this set is known, the writer can make sure that a newly generated sequence number is larger than all existing ones [35]. Tracking the version numbers is complicated by the fact that, although all version numbers are generated by a single node (the writer), they are forwarded with the values exchanged between readers. Tracking the version numbers is therefore done by having a reader "record" a version number before forwarding it with a value.

Recording is akin to writing, but it need not preserve atomicity, and hence, can be implemented in a much simpler manner, without forwarding or having to recursively record forwarded values. (See [8] for more details.)

A construction of a *single-writer single-reader* atomic register [7] avoids this complication, since it needs only to linearize reads from the same node. This reduces the cost of the bounded simulation, even when a multi-reader register is then simulated on top of the single-reader register.

3.2 Immediate Implications

The simulation allowed to port many algorithms designed for shared memory to message-passing systems. This includes atomic snapshots [2], approximate agree-

ment [12], failure detectors [51], and condition-based consensus [44].

This has reduced the interest in the asynchronous message-passing model with crash failures. This model has become virtually "obsolete" when studying computability, as argued for example, by Borowsky and Gafni [16], Herlihy and Shavit [33], as well as Mostefaoui, Rajsbaum and Raynal [44].

3.3 The Quorum Point-of-View

The consistency mechanisms of the ABD simulation can be easily decoupled from its communication pattern. Specifically, the communication pattern of communicating with a majority of the nodes can be replaced with the more general concept of communication with a *quorum*. This conceptual modification appeared in a paper of Lynch and Shvartsman [39], which extended the unbounded ABD simulation to a *multi-writer* register, in a more dynamic situation. A similar idea appeared, at about the same time, in work by Malkhi and Reiter [41], dealing with Byzantine failures. (Both works are discussed in more detail in the next section.)

A *quorum system* is a collection of subsets of nodes, with the property that each pair of sets have at least one common node, that is, each pair of sets have a nonempty intersection. It is quite obvious that the ABD simulation can be paraphrased so that each operation must communicate with a quorum. Indeed, the simple *majority quorum* system, containing all sets with a majority of nodes, is a straightforward example of a quorum system.

Quorums have been used in the context of data and file replication [27, 29]. These works further separate between *write quorums* and *read quorums*, so that every read quorum intersects with every write quorum. A simple example of a read-write quorum system is when nodes are organized in a grid (two-dimensional array), the read quorums contain all the sets of nodes in the same column, and the write quorums contain all the sets of nodes in the same row.

Clearly, the ABD simulation can be modified so that each write operation (all by the same node) communicates with some write quorum, while each read operation communicates with some read quorum.

This refactoring admits a separation of concerns and paves the way to optimizing the communication pattern without changing the overall workings of the algorithm. For example, it is possible to choose a different quorum system when fewer processes may fail, or so as to optimize the performance features of the quorums, e.g., their load and availability [45].

4 Making the Simulation More Robust

Concentrating on the communication pattern, through a quorum system, allowed to make the register simulation more robust, and most notably, to handle dynamic system changes and tolerate more malicious, *Byzantine* failures.

4.1 Dynamic Changes

Lynch and Shvartsman [39, 40] address dynamic systems, where nodes can join and leave the system, in their reconfigurable atomic memory service for dynamic networks.

A key concept in these simulations is the notion of a *configuration*, which includes the set of nodes participating in the simulation, together with the set of read and write quorums. Clearly, when a configuration is fixed, a register can be simulated by running the ABD simulation with the read and write quorums, essentially as described before. Therefore, the core challenge of the algorithm is in *reconfiguring* the system when changes happen.

Reconfiguration modifies the set of participating nodes, and installs a new quorum system appropriate for the new set. Originally [39], Lynch and Shvartsman relied on a special node to manage reconfiguration. In a later paper [40], they presented a decentralized algorithm, nicknamed *RAMBO*, in which nodes propose new configurations and use a *safe consensus* protocol to decide on an agreed new configuration. *Safe consensus* ensures that nodes agree on the same configuration, which was proposed by one of them, but it does not guarantee termination. (This is necessary since full-fledged consensus cannot be solved in an asynchronous system [23].) A neat feature of the algorithm is that the safe consensus algorithm is implemented from shared registers, which, in turn, are simulated over the message-passing system.

At certain points during the execution of the algorithm, several configurations exist: Some configuration might already be agreed on by some nodes, while other nodes might still hold to previous configurations. A key idea in RAMBO is to communicate with a representative quorum from every known configuration. This leads to a component that needs to “garbage collect" old configurations, so as to reduce the amount of communication after transitional periods.

Improvements to RAMBO have appeared in several works, for example, [17, 21, 28].

Very recently, Aguilera et al. [5] presented a variant of RAMBO that sidesteps the need to reach consensus during reconfiguration, assuming the number of reconfigurations is finite.

4.2 Byzantine Failures

Another interesting research thread deals with *Byzantine* failures. These failures model the erroneous behavior that is caused by non-deterministic software errors or even malicious attacks.

Malkhi and Rieter [41] considered the behavior of the simulation when some nodes may experience Byzantine failures. When f Byzantine failures may occur, we need to assume that $n > 3f$; otherwise, it is possible to violate the known lower bound on the ratio of Byzantine failures that can be tolerated [22].

In the simplest case, we can take quorums in which any set of $2f + 1$ nodes is a quorum; this ensures that each pair has a large ($> f$) intersection, containing at least one nonfaulty node. When information from correct nodes is self-verifying (e.g., values are digitally signed), this *dissemination quorum system* suffices for simulating a shared register since this nonfaulty node will forward the correct value.

Malkhi and Rieter also define *opaque masking quorum systems*, which allow to simulate a shared register without assuming that data values are self-verifying. Such quorum systems assure that a node gets many copies of the current value of the register, *all with the same version number*. This requires a higher ratio of nonfaulty nodes.

Abraham, Chockler, Keidar and Malkhi [1] present two simulations that only assume $n > 3f$, but provide weaker properties: one simulates only a safe register, while the other simulates a regular register, and is guaranteed to terminate only if the number of writes is finite. Aiyer, Alvisi and Bazzi [6] simulate an atomic register, with Byzantine readers and (up to one-third of) Byzantine servers; their protocol relies on a secret sharing scheme.

Like vanilla quorum systems, it is possible to design more sophisticated dissemination and opaque masking quorum systems, so as to optimize various parameters, e.g., [42].

5 Application: Replicated Services

Many systems cannot afford to guarantee a majority of nonfaulty processing nodes, seemingly implying that fault-tolerance cannot be obtained. However, systems contain other types of components, for example, independent disk drives. Because these components are cheaper than computers, it is feasible to replicate them in order to achieve fault tolerance. Disk drivers are not programmable, but they can respond to client requests; clients may stop taking steps, and disks may stop responding to requests.

Disk Paxos [26] implements an arbitrary fault-tolerant service on such a *stor-*

age area network containing processing nodes and disks; it provides consistency with any number of asynchronous non-Byzantine *node* failures.

Disk Paxos is based on a shared-memory version of the classic Paxos algorithm [38]; this algorithm is then ported to a storage area network using an optimized version of the ABD simulation: To write a value, a processor writes the value to a majority of the disks. To read, a processor reads a majority of the disks. (This provides a somewhat weaker register, called *regular*, which however, suffices for their shared-memory Paxos algorithm.) The algorithm avoids the version numbers used in the ABD simulation by piggybacking on the version numbers of the Paxos algorithm.

A somewhat similar approach was taken with erasure-coded disks [48], where redundancy is used beyond simple replication to tolerate (non-malicious) disk errors, in addition to partitioning and non-responding disks. It incorporates a quorum reconfiguration algorithm, which allows client requests to proceed unimpeded. This algorithm is, in some sense, *obstruction-free*, since a request may abort when it encounters a concurrent request, yielding an efficient read operation (within a single communication round). This concept was later abstracted by Aguilera et al. to define an *abortable object* [4], which may abort an operation when there are concurrent operations.

A service can be replicated even when servers are Byzantine, by relying on the register simulation tolerating Byzantine failures, see for example, [47].

6 Closing Remarks

One of my goals in this article was to show how picking the right abstraction can bring forth many applications. Finding the right abstraction is in many ways a key for designing good systems: it should hide enough capabilities "under its hood" to provide good leverage in application design, yet, not too much, so the implementation is efficient (or easily admits optimizations).

The are several other research directions that were not discussed in detail here; many of them still pose significant challenges.

Many papers studied the response time or the message complexity of the simulation and improved it, especially in the *best case*, i.e., where the system is well-behaved, by having few failures or maintaining synchronization [19, 32], or by reducing the amount of storage it requires [31]. While there are many pieces of information about the complexity of the simulation [20], it is still open to characterize the exact conditions and scenarios that admit a "fast read" (within a single-communication round) or a "fast write".

The exact bounds for a simulation tolerating Byzantine failures are also not clear yet: What is the highest ratio of failures that can be tolerated with a constant

number of communication rounds, while still providing linearizability and always terminating.

Several papers, e.g., [15, 18, 25], attempt to port the simulation to modern network technologies, which are more ad-hoc and contain mobile nodes, and to sensor networks. The dynamic nature of these systems means these algorithms have to rely on the more robust simulations described in Section 4.1, increasing the importance of developing better understanding of the storage and communication requirements in these systems. These might be traded off with the ratio of faulty nodes that can be tolerated. A related question is to handle a more uniform model, where the number of nodes is unknown in advance.

Needless to say, it is generally unknown how to tolerate Byzantine failures in dynamic systems.

We have described, in brief, how the ABD simulation contributes to service and storage replication, assuming servers might fail. We did not address the problem of dealing with *client failures*, which may lead to incorrect operations applied to the register; one approach was to consider this as a *faulty shared memory* [3,36], and to apply concepts from there [11]. Other approaches deal with faulty clients in a more direct manner, e.g., [43], but this aspect deserves further study.

Acknowledgements: I would to thank Keren Censor and Seth Gilbert for helpful comments.

References

[1] Ittai Abraham, Gregory Chockler, Idit Keidar, and Dahlia Malkhi. Byzantine disk paxos: optimal resilience with byzantine shared memory. *Distributed Computing*, 18(5):387–408, 2006. Previous version in PODC 2004.

[2] Yehuda Afek, Hagit Attiya, Danny Dolev, Eli Gafni, Michael Merritt, and Nir Shavit. Atomic snapshots of shared memory. *Journal of the ACM*, 40(4):873–890, September 1993. Previous version in PODC 1990.

[3] Yehuda Afek, David S. Greenberg, Michael Merritt, and Gadi Taubenfeld. Computing with faulty shared objects. *Journal of the ACM*, 42(6):1231–1274, 1995. Previous version in PODC 1992.

[4] Marcos K. Aguilera, Svend Frolund, Vassos Hadzilacos, Stephanie L. Horn, and Sam Toueg. Abortable and query-abortable objects and their efficient implementation. In *Proceedings of the 26th annual ACM symposium on Principles of distributed computing (PODC)*, pages 23–32, 2007.

[5] Marcos Kawazoe Aguilera, Idit Keidar, Dahlia Malkhi, and Alexander Shraer. Dynamic atomic storage without consensus. In *Proceedings of the 28th ACM symposium on Principles of distributed computing (PODC)*, pages 17–25, 2009.

[6] Amitanand S. Aiyer, Lorenzo Alvisi, and Rida A. Bazzi. Bounded wait-free implementation of optimally resilient byzantine storage without (unproven) cryptographic assumptions. In *Proceddings of 21st International Symposium on Distributed Computing (DISC)*, pages 7–19, 2007.

[7] Hagit Attiya. Efficient and robust sharing of memory in message-passing systems. *Journal of Algorithms*, 34(1):109–127, January 2000. Previous version in WDAG 1996.

[8] Hagit Attiya, Amotz Bar-Noy, and Danny Dolev. Sharing memory robustly in message-passing systems. *Journal of the ACM*, 42(1):121–132, January 1995. Previous version in PODC 1990.

[9] Hagit Attiya, Amotz Bar-Noy, Danny Dolev, Daphne Koller, David Peleg, and Rudiger Reischuk. Achievable cases in an asynchronous environment. In *Proceedings of the 28th Annual Symposium on Foundations of Computer Science (FOCS)*, pages 337–346, 1987.

[10] Hagit Attiya, Amotz Bar-Noy, Danny Dolev, David Peleg, and Rudiger Reischuk. Renaming in an asynchronous environment. *Journal of the ACM*, 37(3):524–548, July 1990. Previous version in [9].

[11] Hagit Attiya and Amir Bar-Or. Sharing memory with semi-byzantine clients and faulty storage servers. *Parallel Processing Letters*, 16(4):419–428, 2006. Previous version in SRDS 2003.

[12] Hagit Attiya, Nancy Lynch, and Nir Shavit. Are wait-free algorithms fast? *Journal of the ACM*, 41(4):725–763, 1994. Previous version in FOCS 1990.

[13] Hagit Attiya and Jennifer Welch. *Distributed Computing: Fundamentals, Simulations and Advanced Topics*. John Wiley& Sons, second edition, 2004.

[14] Amotz Bar-Noy and Danny Dolev. A partial equivalence between shared-memory and message-passing in an asynchronous fail-stop distributed environment. *Mathematical Systems Theory*, 26(1):21–39, 1993. Previous version in PODC 1989.

[15] J. Beal and S. Gilbert. RamboNodes for the metropolitan ad hoc network. In *Workshop on Dependability in Wireless Ad Hoc Networks and Sensor Networks*, 2003.

[16] Elizabeth Borowsky and Eli Gafni. Generalized FLP impossibility result for t-resilient asynchronous computations. In *Proceedings of the 25th Annual ACM Symposium on Theory of Computing (STOC)*, pages 91–100, 1993.

[17] 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. Previous version in OPODIS 2005.

[18] Shlomi Dolev, Seth Gilbert, Nancy A. Lynch, Alexander A. Shvartsman, and Jennifer L. Welch. Geoquorums: Implementing atomic memory in mobile ad hoc networks. *Distributed Computing*, 18(2):125–155, 2005. Previous version in DISC 2003.

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

[20] Burkhard Englert, Chryssis Georgiou, Peter M. Musial, Nicolas C. Nicolaou, and Alexander A. Shvartsman. On the efficiency of atomic multi-reader, multi-writer distributed memory. In *Proceedings of the 13th International Conference on Principles of Distributed Systems (OPODIS)*, pages 240–254, 2009.

[21] Burkhard Englert and Alexander Shvartsman. Graceful quorum reconfiguration in a robust emulation of shared memory. In *Proceedings of the 20th International Conference on Distributed Computing Systems (ICDCS)*, pages 454–463, 2000.

[22] Michael J. Fischer, Nancy A. Lynch, and Michael Merritt. Easy impossibility proofs for distributed consensus problems. *Distributed Computing*, 1(1):26–39, 1986.

[23] Michael J. Fischer, Nancy A. Lynch, and Michael S. Paterson. Impossibility of distributed consensus with one faulty process. *Journal of the ACM*, 32(2):374–382, 1985. Previous version in PODC 1984.

[24] Steven Fortune and James Wyllie. Parallelism in random access machines. In *Proceedings of the 10th annual ACM symposium on Theory of computing (STOC)*, pages 114–118, 1978.

[25] Roy Friedman, Gabi Kliot, and Chen Avin. Probabilistic quorum systems in wireless ad hoc networks. In *Proceedings of the 9th IEEE International Conference on Dependable Systems and Networks (DSN)*, pages 277–286, 2008.

[26] Eli Gafni and Leslie Lamport. Parallelism in random access machines. *Distributed Computing*, 16:1–20, February 2003. Previous version in DISC 2000.

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

[28] Chryssis Georgiou, Peter M. Musial, and Alexander A. Shvartsman. Long-lived RAMBO: Trading knowledge for communication. *Theoretical Computer Science*, 383(1):59–85, 2007. Previous version in SIROCCO 2004.

[29] David K. Gifford. Weighted voting for replicated data. In *Proceedings of the 7th ACM symposium on Operating Systems Principles (SOSP)*, pages 150–162, 1979.

[30] Allan Gottlieb, Ralph Grishman, Clyde P. Kruskal, Kevin P. McAuliffe, Larry Rudolph, and Marc Snir. The NYU Ultracomputer—designing a MIMD, shared-memory parallel machine. In *Proceedings of the 9th annual symposium on Computer Architecture (ISCA)*, pages 27–42, 1982.

[31] Rachid Guerraoui and Ron Levy. Robust emulations of shared memory in a crash-recovery model. In *Proceedings of the International Conference on Distributed Computing Systems (ICDCS)*, volume 24, pages 400–407, 2004.

[32] Rachid Guerraoui and Marko Vukolić. Refined quorum systems. In *Proceedings of the 26th annual ACM symposium on Principles of distributed computing (PODC)*, pages 119–128, 2007.

[33] Maurice P. Herlihy and Nir Shavit. The topological structure of asynchronous computability. *Journal of the ACM*, 46(6):858–923, 1999. Previous version in STOC 1993 and STOC 1994.

[34] Maurice P. Herlihy and Jeannette M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems*, 12(3):463–492, July 1990.

[35] Amos Israeli and Ming Li. Bounded time-stamps. *Distributed Computing*, 6(4):205–209, July 1993. Previous version in FOCS 1987.

[36] Prasad Jayanti, Tushar D. Chandra, and Sam Toueg. Fault-tolerant wait-free shared objects. *Journal of the ACM*, 45(3):451–500, 1998. Previous version in FOCS 1992.

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

[38] Leslie Lamport. Paxos made simple. *ACM SIGACT News*, 32(4):18–25, 2001.

[39] Nancy A. Lynch and Alexander A. Shvartsman. Robust emulation of shared memory using dynamic quorum-acknowledged broadcasts. In *Proceedings of the 27th International Symposium on Fault-Tolerant Computing (FTCS)*, pages 272–281, 1997.

[40] Nancy A. Lynch and Alexander A. Shvartsman. Rambo: A Reconfigurable Atomic Memory Service for Dynamic Networks. In *Proceedings of the 16th International Conference on Distributed Computing (DISC)*, pages 173–190, 2002.

[41] Dahlia Malkhi and Michael K. Reiter. Byzantine quorum systems. *Distributed Computing*, 11(4):203–213, 1998. Previous version in STOC 1997.

[42] Dahlia Malkhi, Michael K. Reiter, Avishai Wool, and Rebecca N. Wright. Probabilistic quorum systems. *Information and Computation*, 170(2):184–206, 2001. Previous version in PODC 1997.

[43] Jean-Philippe Martin, Lorenzo Alvisi, and Michael Dahlin. Minimal byzantine storage. In *Proceedings of the 16th International Conference on Distributed Computing (DISC)*, pages 311–325, 2002.

[44] Achour Mostefaoui, Sergio Rajsbaum, and Michel Raynal. Conditions on input vectors for consensus solvability in asynchronous distributed systems. *Journal of the ACM*, 50(6):922–954, 2003. Previous version in STOC 2001.

[45] Moni Naor and Avishai Wool. The load, capacity, and availability of quorum systems. *SIAM J. Comput.*, 27(2), 1998. Previous version in FOCS 1994.

[46] Christos H. Papadimitriou. The serializability of concurrent database updates. *Journal of the ACM*, 26(4):631–653, October 1979.

[47] Rodrigo Rodrigues, Barbara Liskov, and Liuba Shrira. The design of a robust peer-to-peer system. In *Proceedings of the 10th workshop on ACM SIGOPS European workshop*, pages 117–124, 2002.

[48] Yasushi Saito, Svend Frølund, Alistair Veitch, Arif Merchant, and Susan Spence. FAB: building distributed enterprise disk arrays from commodity components. *SIGOPS Oper. Syst. Rev.*, 38(5):48–58, 2004. From ASPLOS 2004.

[49] Robert H. Thomas. A majority consensus approach to concurrency control for multiple copy databases. *ACM Transactions on Database Systems*, 4(2):180–209, 1979.

[50] Eli Upfal and Avi Wigderson. How to share memory in a distributed system. *Journal of the ACM*, 34(1):116–127, 1987. Previous version in FOCS 1984.

[51] Jiong Yang, Gil Neiger, and Eli Gafni. Structured derivations of consensus algorithms for failure detectors. In *Proceedings of the 17th annual ACM symposium on Principles of Distributed Computing (PODC)*, pages 297–306, 1998.