On Constructing a Byzantine Linearizable SWMR Atomic Register from SWSR Atomic Registers

Ajay D. Kshemkalyani
University of Illinois Chicago
Chicago, United States
ajay@uic.edu

Manaswini Piduguralla
Indian Institute of Technology Hyderabad
Hyderabad, India
cs20resch11007@iith.ac.in

Sathya Peri
Indian Institute of Technology Hyderabad
Hyderabad, India
sathya_p@cse.iith.ac.in

Anshuman Misra
Purdue University Fort Wayne
Fort Wayne, United States
misra47@pfw.edu

Abstract

The SWMR atomic register is a fundamental building block in shared memory distributed systems and implementing it from SWSR atomic registers is an important problem. While this problem has been solved in crash-prone systems, it has received less attention in Byzantine systems. Recently, Hu and Toueg gave such an implementation of the SWMR register from SWSR registers. While their definition of register linearizability is consistent with the definition of Byzantine linearizability of a concurrent history of Cohen and Keidar, it has several drawbacks.

In this paper, we give a stronger definition of a Byzantine linearizable register that overcomes these drawbacks. The construction of a Byzantine linearizable SWMR atomic register from SWSR registers that meets our stronger definition is given in the full arxiv report. The construction is correct when $n > 3f$, where n is the number of readers, f is the maximum number of Byzantine readers, and the writer can also be Byzantine. The construction relies on a public-key infrastructure.

CCS Concepts

• **Theory of computation** → **Distributed algorithms**; **Concurrent algorithms**; • **Computer systems organization** → *Dependable and fault-tolerant systems and networks.*

Keywords

Byzantine fault tolerance, SWMR atomic register, Linearizability, SWSR register

ACM Reference Format:
Ajay D. Kshemkalyani, Manaswini Piduguralla, Sathya Peri, and Anshuman Misra. 2025. On Constructing a Byzantine Linearizable SWMR Atomic Register from SWSR Atomic Registers. In *26th International Conference on Distributed Computing and Networking (ICDCN 2025), January 04–07, 2025, Hyderabad, India.* ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/3700838.3700839

ICDCN 2025, January 04–07, 2025, Hyderabad, India

ACM ISBN 979-8-4007-1062-9/25/01
https://doi.org/10.1145/3700838.3700839

1 Introduction

Implementing shared registers from weaker types of registers is a fundamental problem in distributed systems and has been extensively studied [2, 3, 5, 6, 8, 13, 14, 17–22]. We consider the problem of implementing a single-writer multi-reader register (SWMR) from single-writer single-reader (SWSR) registers in a system with Byzantine processes. This SWMR register in a Byzantine setting is of great importance in recent research. For example, Mostefaoui et al. [16] prove that in message-passing systems with Byzantine failures, there is a f-resilient implementation of a SWMR register if and only if $f < n/3$ processes are faulty, where f is the number of Byzantine processes and n is the total number of processes. It was the first to give the definition of a linearizable SWMR register in the presence of Byzantine processes and [4] generalized it to objects of any type. Aguilera et al. [1] use atomic SWMR registers to solve some agreement problems in hybrid systems subject to Byzantine process failures. Cohen and Keidar [4] give f-resilient implementations of three objects – asset transfer, reliable broadcast, atomic snapshots – using atomic SWMR registers in systems with Byzantine failures where at most $f < n/2$ processes are faulty. Their implementations were based on their definition of Byzantine linearizability of a concurrent history.

In other related work, a SWMR register was built above a message-passing system where processes communicate using send/receive primitives with the constraint that $f < n/3$ [11, 16]. These works do not use signatures. Unbounded history registers were required in [11] whereas [16] used $O(n^2)$ messages per write operation. Although building SWMR registers over SWSR registers or over message-passing systems is equivalent as SWSR registers can be emulated over send/receive and vice versa, this is a round-about and expensive solution. A similar problem for the client-server paradigm in message-passing systems was solved in [15] using cryptography.

1.1 Motivation

The SWMR atomic register is seen to be a basic building block in shared memory distributed systems and implementing it from SWSR atomic registers is an important problem. While this problem has been solved in crash-prone systems, it has received recent attention in Byzantine systems. Recently, Hu and Toueg gave such an implementation of the SWMR register from SWSR registers

[9, 10]. While their definition of register linearizability is consistent with the definition of Byzantine linearizability of a concurrent history of Cohen and Keidar [4], both [4, 9, 10] as well as [11, 16] have the following drawbacks.

(1) If the writer is Byzantine, the register is vacuously linearizable no matter what values the correct readers return. Reads by correct processes can return any value whatsoever including the initial value while the register meets their definition of linearizability. In particular, there is no view consistency. For example, in the Hu-Toueg algorithm, consider a scenario where a Byzantine writer writes a different data value associated with the same counter value to the various readers' SWSR registers. The correct readers will return different data values associated with the same counter value, thus having inconsistent views. An example application where this is a problem is collaborative editing for a document hosted on a single server. Another reason why this is problematic is that it violates the agreement clause of the well-known consensus/Byzantine agreement problem, which requires that all non-faulty processes must agree on the same value even if the source is Byzantine. We require view consistency.
(2) Their definition of register linearizability does not factor in, or ignores, those values written by a Byzantine writer, by honestly following the writer protocol for those values. We need a stronger notion of a *correct write operation* that factors in such values as being written correctly. Also, note that the Byzantine writer is in control of the execution both above and below the SWMR register interface and hence the value that it writes in a correct write operation can be assumed to be the value intended to be written (correctly) and not altered by Byzantine behavior.
(3) Their definition of register linearizability allows a value written by a Byzantine writer to just a single reader's SWSR register to be returned by a correct process. In order to validate that the writer intended to write that value honestly, we would like a minimum threshold number of readers' SWSR registers to be written that same value to enable that value to become eligible for being returned to a correct reader. This validates the intention of the Byzantine writer to write that particular value.
(4) In their definition of register linearizability, their notion of a "current" value returned by a correct reader is not related to the most recent value written by a correct write operation of a Byzantine writer. We need a more up to date version of the value that can be returned by a correct reader. This helps give a stronger guarantee of progress from the readers' perspective.

Our definition of a Byzantine linearizable register is stronger than not just that of [4, 9, 10] but also that of [11, 15, 16] and overcomes the above drawbacks. Further, we are interested in implementing the SWMR register over SWSR registers directly in the shared memory model.

1.2 Contributions

(1) In this paper, we give a stronger definition of a *Byzantine linearizable register* that overcomes all the above drawbacks of [4, 9, 10] and [11, 15, 16]. We introduce the concept of a *correct write operation* by a Byzantine writer as one that conforms to the write protocol. We also introduce the notion of a *pseudo-correct write operation* by a Byzantine writer, which has the effect of a correct write operation. Only correct and pseudo-correct writes may be returned by correct readers. The correct and pseudo-correct writes are totally ordered and this order is the total order in which the writes are performed.
(2) The construction of a Byzantine linearizable SWMR atomic register from SWSR atomic registers that meets our stronger definition is given in [12]. The construction is correct when $n > 3f$, where n is the number of readers, f is the maximum number of Byzantine readers, and the writer can also be Byzantine. The construction relies on a public-key infrastructure (PKI).
The construction develops the idea of the readers validating the logical timestamp of the writing of the values set aside for them by the writer. A sufficient number of correct readers will validate this consistently, and that forms the basis of the total order used to ensure Byzantine register linearizability. As compared to the algorithm in [9, 10] which can tolerate any number of Byzantine readers, our algorithm requires $f < n/3$. Also, in the algorithm in [9, 10], a reader that stops reading also stops taking implementation steps whereas our algorithm requires a reader helper thread to take infinitely many steps even if it has no read operation to apply. The algorithm in [9, 10] as well as our algorithm use a PKI.

2 Model and Preliminaries

2.1 Model Basics

We consider the shared memory model of a distributed system. The system contains a set P of asynchronous processes. These processes access some shared memory objects. All inter-process communication is done through an API exposed by the objects. Processes invoke operations that return some response to the invoking process. We assume reliable shared memory but allow for an adversary to corrupt up to f processes in the course of a run. A corrupted process is defined as being *Byzantine* and such a process may deviate arbitrarily from the protocol. A non-Byzantine process is *correct* and such a process follows the protocol and takes infinitely many steps.

We also assume a PKI. Using this, each process has a public-private key pair used to sign data and verify signatures of other processes. A values v signed by process p is denoted $\langle v \rangle_p$.

We give an algorithm that emulates an object O, viz., a SWMR register from SWSR registers. We assume that there is adequate access control such that a SWSR register can be accessed only by the single writer and the single reader between whom the register is set up, and that another (Byzantine) process cannot access it. The algorithm is organized as methods of O. A method execution is a sequence of steps. It begins with the *invoke* step, goes through steps that access lower-level objects, viz., SWSR registers, and ends with a *return* step. The invocation and response delineate the method's execution interval. In an *execution* σ, each correct process invokes methods sequentially, and steps of different processes are interleaved. Byzantine processes take arbitrary steps irrespective of

the protocol. The *history* H of an execution σ is the sequence of high-level invocation and response events of the emulated SWMR register in σ. A history H defines a partial order $\prec_H$ on operations. $op_1 \prec_H op_2$ if the response event of op_1 precedes the invocation event of op_2 in H. op_1 is concurrent with op_2 if neither precedes the other.

In our algorithm, we assume that each reader process has a helper thread that takes infinitely many steps even if the reader stops reading the implemented register. These steps are outside the invocation-response intervals of the readers' own operations. Also, the linearization point of a pseudo-correct write operation may fall after the invocation-response interval. These are non-standard features of our shared memory model.

2.2 Linearizability of a Concurrent History

Linearizability, a popular correctness condition for concurrent objects, is defined using an object's sequential specification.

Definition 1. *(Linearization of a concurrent history:) A* linearization *of a concurrent history H of object o is a sequential history H' such that:*

(1) *After removing some pending operations from H and completing others by adding matching responses, it contains the same invocations and responses as H',*
(2) *H' preserves the partial order $\prec_H$, and*
(3) *H' satisfies o's sequential specification.*

A SWMR register as well as a SWSR register expose the *read* and *write* operations. The sequential specification of a SWMR and a SWSR register states that a read operation from register *Reg* returns the value last written to *Reg*. Following Cohen and Keidar [4], we manage Byzantine behavior in a way that provides consistency to correct processes. This is achieved by linearizing correct processes' operations and offering a degree of freedom to embed additional operations by Byzantine processes.

Let $H|_{correct}$ denote the projection of history H to all correct processes. History H is Byzantine linearizable if $H|_{correct}$ can be augmented by (some) operations of Byzantine processes such that the completed history is linearizable. Thus, there is another history with the same operations by correct processes as in H, and additional operations by at most f Byzantine processes.

Definition 2. *(Byzantine linearization of a concurrent history [4]:) A history H is Byzantine linearizable if there exists a history H' such that $H'|_{correct} = H|_{correct}$ and H' is linearizable.*

An object supports Byzantine linearizable executions if all of its executions are Byzantine linearizable. SWMR registers support Byzantine linearizable executions because before every read from such a register, invoked by a correct process, one can add a corresponding Byzantine write.

2.3 Linearizability of Register Implementations

Hu and Toueg defined register linearizability in a system with Byzantine processes as follows [9, 10]. They let v_0 be the initial value of the implemented register and v_k be the value written by the kth write operation by the writer w of the implemented register.

Definition 3. (Register Linearizability [9, 10]:) *In a system with Byzantine process failures, an implementation of a SWMR register is linearizable if and only if the following holds. If the writer is not malicious, then:*

- *(Reading a "current" value) If a read operation* R *by a process that is not malicious returns the value v then:*
  - *there is a write v operation that immediately precedes* R *or is concurrent with* R*, or*
  - *$v = v_0$ (the initial value) and no write operation precedes* R.
- *(No "new-old" inversion) If two read operations* R *and* R' *by processes that are not malicious return values v_k and $v_{k'}$, respectively, and* R *precedes* R'*, then $k \leq k'$.*

Note that Hu-Toueg specified this register linearizability only if the writer is non-malicious. While this definition of register linearizability is consistent with the definition of a Byzantine linearization of a concurrent history (Definition 2), in the sense that both are concerned only with correct processes' views, it is not ideal for the reasons given in Section 1.1. Therefore the register should meet stronger criteria of a linearizable register, in the face of Byzantine processes, to accommodate the behavior of the Byzantine writer when it is behaving (writing) correctly. We term such a register as a *Byzantine linearizable register*. In this paper, we first define a Byzantine linearizable register, and then solve the problem of constructing a Byzantine linearizable SWMR register from SWSR registers.

3 Characterization of Byzantine Register Linearizability

The object SWMR register supports Byzantine linearizable executions [4]. However, we need to construct a SWMR register from SWSR registers. Here we characterize the requirements for such a construction, culminating in Definition 7 of Byzantine Register Linearizability. The writer as well as the reader processes can be Byzantine. As a Byzantine reader can return any value whatsoever, the linearizability specification is based on values that correct readers return.

In general, when an object O_1, denoted a *high-level object (HLO)* is simulated or constructed using objects of another type O_2, denoted a *low-level object (LLO)*, there are two interfaces. A process interacts with the HLO through a *high-level interface (HLI)* through alternating invocations and matching responses. Between such a pair of matching invocation and response, the process interacts with the LLO through a *low-level interface (LLI)* using alternating invocations and responses. Such interactions are in software.

For our problem, the HLO is the Byzantine-tolerant SWMR atomic register and the HLI is the read and write operation. The LLO is the SWSR atomic register and the LLI is also the read and write operation. We term the program code executed below the HLI and above the LLI for a single invocation of a write/read at the HLI as the code or protocol for the (HLI) write operation/read operation, respectively.

In the face of Byzantine readers as well as a Byzantine writer, we need to define a correct write operation. In the sequel, we use u or v to refer to the actual data value written. A $write(v)$ invocation at the HLI may be converted at a Byzantine writer into possibly multiple operation invocations for different $write(v')$ at the LLI to all or some subset of the various instances of the LLO. If a $write(v)$ invocation at the HLI is converted by a Byzantine writer into an

invocation of $write(v')$ and it executes the protocol exactly for this value v', it is considered as a correct write operation because that can be taken to be the value the writer writes or intended to write. Likewise if the $write(v)$ at the HLI is converted into multiple serial invocations of $write(v')$ (for different values of v') and the protocol for each of these v' is correctly followed, these various $write(v')$ are considered correct write operations because that sequence of write operations can be taken to be the values the writer writes or intended to write. This is because the invocation/response at the HLI is at a Byzantine process which controls the execution of code above the LLI and above the HLI. In a correct write operation, the code between the HLI and the LLI is followed correctly by the Byzantine process.

Definition 4. *A* correct (write) operation *is a (write) operation that follows the (write) protocol, but possibly with a different value than that passed down at the HLI.*

So far in the literature [4, 9, 10], any behavior of a Byzantine writer is allowable in the linearizability definition. We accommodate a Byzantine writer differently and introduce the concept of a *pseudo-correct write operation* (Definition 5), which is a Byzantine write operation that has the effect of a correct write operation, i.e., whose actions that are visible to correct readers cannot be distinguished from the actions of a correct write operation by correct readers. This is first informally motivated as follows. A Byzantine write operation can, for example,

(1) write multiple values (possibly resulting in multiple pseudo-correct write operations) or
(2) together with earlier write operations write a single value (possibly resulting in a pseudo-correct write operation), or
(3) together with earlier write operations that wrote different values write those values (possibly resulting in multiple pseudo-correct write operations).

Thus, there is no longer a one-one mapping from write operations issued to the HLI object interface to values written to the object; it is a many-many mapping.

Definition 5. *A* pseudo-correct (write) operaton *is a (write) operation such that whatever steps the (writer) process performs in it and that results in a value being returned to correct readers, is indistinguishable to correct readers' executions below the HLI from an actual correct (write) operation's steps.*

We now give a stronger definition of a Byzantine linearization of a concurrent history than Definition 2 of Cohen-Keidar. We tame the Byzantine behavior in a stronger way to provide consistency to correct processes. We linearize the correct processes' operations and offer a *(more) limited* degree of freedom by way of embedding *only* correct and pseudo-correct write operations by Byzantine processes. History H is Byzantine linearizable if $H|_{correct}$ can be augmented by (some) pseudo-correct and correct operations (and not any arbitrary operations) of Byzantine processes such that the completed history is linearizable.

Definition 6. *(Byzantine linearization of a concurrent history (newly proposed definition):) A history H is Byzantine linearizable if there exists a history H' containing correct and pseudo-correct write operations by Byzantine processes and writes and reads by correct processes such that $H'|_{correct} = H|_{correct}$ and H' is linearizable.*

Counter-example: The Hu-Toueg algorithm (Algorithm 2, with unforgeable signatures) does not have a Byzantine linearization for concurrent histories as per our Definition 6. Consider the same example from Section 1; the Byzantine writer writes $\langle k, v_i \rangle$ and $\langle k, v_j \rangle$ values with the same counter value k to two correct readers x and y's SWSR registers, respectively. Reader x returns v_i in read operation $R_{x,1}$ after which y invokes a read operation $R_{y,1}$ which can return v_j. This history cannot be Byzantine linearized; if we insert the pseudo-correct $write(\langle k, v_i \rangle)$ before $R_{x,1}$, then $write(\langle k, v_j \rangle)$ does not qualify as a pseudo-correct write (to be inserted before $R_{y,1}$) because reader y's execution below the HLI would see two values ($\langle k, v_i \rangle$ and $\langle k, v_j \rangle$) with the same counter value k – which could never have been written by correct write operations as per their writer protocol.

We now present the final definition of the Byzantine linearizable register using physical time and HLOs. Let v^i be the value written by the ith correct or pseudo-correct write W^i in a Byzantine linearization of a concurrent history (Definition 6), which is used in Definition 7, following the notation in [7]. Note that to determine i, v^i and W^i requires knowing what happened below the HLI and above the LLI because of the nature of pseudo-correct writes; but there is actually no need to determine i, v^i, and W^i.

Definition 7. *(Byzantine Linearizable Register). In a system with Byzantine process failures, an implementation of a SWMR register is linearizable if and only if the following two properties are satisfied in a Byzantine linearization of a concurrent history.*

(1) **Reading a current value:** *When a read operation* R *by a non-Byzantine process returns the value v:*
   (a) *if $v = v_0$ then no correct or pseudo-correct write operation precedes* R
   (b) *else if $v \neq v_0$ then v was written by the correct or pseudo-correct write operation that immediately precedes* R.
(2) **No "new-old" inversions:** *If read operations* R *and* R' *by non-Byzantine processes return values v^i and v^j, respectively, and* R *precedes* R', *then $i \leq j$.*

A pseudo-correct (write) operation looks like a correct write operation to correct readers; however the writer may still not follow the write protocol exactly. Taming a Byzantine write and making visible what a Byzantine write does as part of a pseudo-correct write is done by correct readers in their steps below the HLI and above the LLI. As the Byzantine writer may exit its write protocol prematurely, the linearization point of a pseudo-correct write may be after the invocation-response interval(s) of the HLI operations that triggered the pseudo-correct write. In fact, a pseudo-correct write by a Byzantine process is not defined to have any invocation-response operations.

3.1 Towards an Algorithm

We assume WLOG that there are n SWSR registers R_init_{wi} writable by the single writer w and readable by reader $i \in [i, n]$. The Byzantine writer can behave anyhow and can write different values to the SR registers, or write different values to different subsets of SR registers while not writing to some of them at all, or write multiple different values over time to the same some or all SR registers, as part of the same write operation. t of the n readers are Byzantine.

The writer writes (k, u), where k is a monotonically increasing sequence number and u is a data value, to the various R_init_{wi}. Although there is a many-many mapping from write operations issued to the HLI object interface to values written to the object, this does not pose any ambiguity as the different values that are returned to the correct readers have different logical timestamps k.

Correct write operations are totally ordered in time. This total order is also the logical time ordering of their timestamps. *To be indistinguishable to correct readers below the HLI from correct write operations (to satisfy Definition 5), write operations whose values can be returned should be totally ordered along with the set of correct write operations, by their logical timestamps.* Based on this above *principle*, we proceed to define pseudo-correct write operations.

Definition 8. *A* potential pseudo-correct write operation *of value (k, v) is a write operation, timestamped k, that may not follow the write protocol but*

(1) *there is a quorum of size $\geq n - 2t$ indices i of correct readers such that (k, v) was written to R_init_{wi}, and*
(2) *$k > k'$ for all (k', v') already read from these R_init_{wi}.*

Definition 9. *A write operation* stabilizes *if its value can be returned by a correct reader.*

A correct write operation always stabilizes whereas a potential pseudo-correct write may stabilize, depending on run-time dynamic data races due to the asynchronous readers, steps of Byzantine readers and the Byzantine writer, and the algorithm. Only all write operations that stabilize have a linearization point.

Definition 10. *A* pseudo-correct write operation *is a potential pseudo-correct write operation that stabilizes.*

Definition 11. *(*Monotonicity/Total Order of stabilized write operation timestamps Property*:) The set of write operation timestamps that stabilize is totally ordered.*

The algorithm we have designed [12] satisfies this property. Only correct and pseudo-correct writes may be returned by correct readers. A correct reader cannot distinguish between a correct and a pseudo-correct write operation.

4 Conclusions

This paper studied Byzantine tolerant construction of a SWMR atomic register from SWSR atomic registers. It is the first to propose a definition of Byzantine register linearizability by non-trivially taking into account Byzantine behavior of the writer and readers, and by overcoming the drawbacks of the definition used by previous works. We introduced the concept of a correct write operation by a Byzantine writer. We also introduced the notion of a pseudo-correct write operation by a Byzantine writer, which has the effect of a correct write operation. Only correct and pseudo-correct writes may be returned by correct readers. The correct and pseudo-correct writes are totally ordered by their linearization points and this order is the total order in logical time in which the writes were performed. An algorithm to construct a Byzantine tolerant SWMR atomic register from SWSR atomic registers that meets our definition of Byzantine register linearizability is given in the full version [12].

References

[1] Marcos K. Aguilera, Naama Ben-David, Rachid Guerraoui, Virendra J. Marathe, and Igor Zablotchi. 2019. The Impact of RDMA on Agreement. In *Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing*, Peter Robinson and Faith Ellen (Eds.). ACM, 409–418. https://doi.org/10.1145/3293611.3331601

[2] Hagit Attiya, Amotz Bar-Noy, and Danny Dolev. 1995. Sharing Memory Robustly in Message-Passing Systems. *J. ACM* 42, 1 (1995), 124–142. https://doi.org/10.1145/200836.200869

[3] James E. Burns and Gary L. Peterson. 1987. Constructing Multi-reader Atomic Values From Non-atomic Values. In *Proceedings of the Sixth Annual ACM Symposium on Principles of Distributed Computing*, Fred B. Schneider (Ed.). ACM, 222–231. https://doi.org/10.1145/41840.41859

[4] Shir Cohen and Idit Keidar. 2021. Tame the Wild with Byzantine Linearizability: Reliable Broadcast, Snapshots, and Asset Transfer. In *35th International Symposium on Distributed Computing, DISC 2021 (LIPIcs, Vol. 209)*, Seth Gilbert (Ed.). 18:1–18:18. https://doi.org/10.4230/LIPICS.DISC.2021.18

[5] Sibsankar Haldar and K. Vidyasankar. 1995. Constructing 1-Writer Multireader Multivalued Atomic Variable from Regular Variables. *J. ACM* 42, 1 (1995), 186–203. https://doi.org/10.1145/200836.200871

[6] Maurice Herlihy. 1991. Wait-Free Synchronization. *ACM Trans. Program. Lang. Syst.* 13, 1 (1991), 124–149. https://doi.org/10.1145/114005.102808

[7] Maurice Herlihy and Nir Shavit. 2008. *The Art of Multiprocessor Programming*. Morgan-Kaufmann.

[8] Maurice Herlihy and Jeannette M. Wing. 1990. Linearizability: A Correctness Condition for Concurrent Objects. *ACM Trans. Program. Lang. Syst.* 12, 3 (1990), 463–492. https://doi.org/10.1145/78969.78972

[9] Xing Hu and Sam Toueg. 2022. On Implementing SWMR Registers from SWSR Registers in Systems with Byzantine Failures. In *36th International Symposium on Distributed Computing, DISC 2022, October 25-27, 2022, Augusta, Georgia, USA (LIPIcs, Vol. 246)*, Christian Scheideler (Ed.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 36:1–36:19. https://doi.org/10.4230/LIPICS.DISC.2022.36

[10] Xing Hu and Sam Toueg. 2024. On implementing SWMR registers from SWSR registers in systems with Byzantine failures. *Distributed Comput.* 37, 2 (2024), 145–175. https://doi.org/10.1007/S00446-024-00465-5

[11] Damien Imbs, Sergio Rajsbaum, Michel Raynal, and Julien Stainer. 2016. Read-/write shared memory abstraction on top of asynchronous Byzantine message-passing systems. *J. Parallel Distributed Comput.* 93-94 (2016), 1–9. https://doi.org/10.1016/J.JPDC.2016.03.012

[12] Ajay D. Kshemkalyani, Manaswini Piduguralla, Sathya Peri, and Anshuman Misra. 2024. Construction of a Byzantine Linearizable SWMR Atomic Register from SWSR Atomic Registers. *CoRR* abs/2405.19457 (2024). https://doi.org/10.48550/ARXIV.2405.19457 arXiv:2405.19457

[13] Leslie Lamport. 1986. On Interprocess Communication. Part I: Basic Formalism. *Distributed Comput.* 1, 2 (1986), 77–85. https://doi.org/10.1007/BF01786227

[14] Leslie Lamport. 1986. On Interprocess Communication. Part II: Algorithms. *Distributed Comput.* 1, 2 (1986), 86–101. https://doi.org/10.1007/BF01786228

[15] Dahlia Malkhi and Michael K. Reiter. 1998. Secure and Scalable Replication in Phalanx. In *The Seventeenth Symposium on Reliable Distributed Systems, SRDS 1998, West Lafayette, Indiana, USA, October 20-22, 1998, Proceedings*. IEEE Computer Society, 51–58. https://doi.org/10.1109/RELDIS.1998.740474

[16] Achour Mostéfaoui, Matoula Petrolia, Michel Raynal, and Claude Jard. 2017. Atomic Read/Write Memory in Signature-Free Byzantine Asynchronous Message-Passing Systems. *Theory Comput. Syst.* 60, 4 (2017), 677–694. https://doi.org/10.1007/S00224-016-9699-8

[17] Gary L. Peterson. 1983. Concurrent Reading While Writing. *ACM Trans. Program. Lang. Syst.* 5, 1 (1983), 46–55. https://doi.org/10.1145/357195.357198

[18] Gary L. Peterson and James E. Burns. 1987. Concurrent Reading While Writing II: The Multi-writer Case. In *28th Annual Symposium on Foundations of Computer Science*. IEEE Computer Society, 383–392. https://doi.org/10.1109/SFCS.1987.15

[19] Ambuj K. Singh, James H. Anderson, and Mohamed G. Gouda. 1987. The Elusive Atomic Register Revisited. In *Proceedings of the Sixth Annual ACM Symposium on Principles of Distributed Computing*, Fred B. Schneider (Ed.). ACM, 206–221. https://doi.org/10.1145/41840.41858

[20] K. Vidyasankar. 1988. Converting Lamport's Regular Register to Atomic Register. *Inf. Process. Lett.* 28, 6 (1988), 287–290. https://doi.org/10.1016/0020-0190(88)90175-5

[21] K. Vidyasankar. 1991. A Very Simple Construction of 1-Writer Multireader Multivalued Atomic Variable. *Inf. Process. Lett.* 37, 6 (1991), 323–326. https://doi.org/10.1016/0020-0190(91)90149-C

[22] Paul M. B. Vitányi and Baruch Awerbuch. 1986. Atomic Shared Register Access by Asynchronous Hardware (Detailed Abstract). In *27th Annual Symposium on Foundations of Computer Science*. IEEE Computer Society, 233–243. https://doi.org/10.1109/SFCS.1986.11