Wait-Free Regular Storage from Byzantine Components

Ittai Abraham[*] Gregory Chockler[†] Idit Keidar[‡] Dahlia Malkhi[§]

July 26, 2006

Abstract

We consider the problem of implementing a wait-free regular register from storage components prone to Byzantine faults. We present a simple, efficient, and self-contained construction of such a register. Our construction utilizes a novel building block, called a 1-regular register, which can be efficiently implemented from Byzantine fault-prone components.

1 Introduction

In recent years, many systems that construct storage solutions from components prone to Byzantine faults in asynchronous settings have been suggested, e.g., [10, 11, 7, 1, 4, 14, 9]. Such systems can be formally modeled using an asynchronous shared memory model where up to a threshold t out of n memory objects may fail by being non-responsive or returning arbitrary values [6]; this failure model is called *non-responsive arbitrary (NR-Arbitrary)* [6]. Like most of the aforementioned systems, e.g., [10, 7, 4, 14], we assume that $n \geq 4t + 1$. In previous work [1], we show that this assumption is necessary for implementations as efficient (in terms of round complexity) as the one in this paper.

This paper focuses on constructing a wait-free single-writer regular register. Wait-freedom refers to independence of clients from one another's liveness and activity, but not from the memory objects from which the register is built, $n - t$ of which should be correct. A *regular register* guarantees

[*]School of Computer Science and Engineering, The Hebrew University of Jerusalem.
[†]Lab for Computer Science and Artificial Intelligence. Massachusetts Institute of Technology.
[‡]Department of Electrical Engineering, The Technion – Israel Institute of Technology.
[§]School of Computer Science and Engineering, The Hebrew University of Jerusalem, Israel, and Microsoft Research, Silicon-Valley.

that every read operation returns either the register's value before the read is invoked (the value written by the last write operation that returns before the read is invoked, or the initial value if no value is written before the read) or a value that is written concurrently with the read [8]. A regular register is weaker than an atomic one, as its history is not always linearizable. Our focus on regular registers is motivated by recent studies that indicate that storage with *regular* semantics is sufficient in many cases [3, 15, 1]. But if atomicity is required, it is straightforward to extend our protocol to provide atomic semantics.

Most existing wait-free Byzantine-resilient storage constructions, e.g. [6, 10, 1], implement *safe* registers. A safe register guarantees that every read operation that does not overlap any write returns the latest written value, or the initial value if no value was written; the result of a read operation that overlaps a write operation may be arbitrary [8]. In contrast to regular registers, safe registers, by themselves, are too weak to be directly useful for applications. The focus on these semantics has been justified by the existence of known reductions to wait-free safe registers from regular and atomic ones [8]. However, this approach results in constructions that are not tailored to the requirements of a distributed storage system, since traditional constructions of strong wait-free objects from weaker ones, e.g., [12, 8, 16, 17, 5], were not designed with distributed storage in mind. In particular, such constructions typically focus on bounding the memory size rather than reducing the number of shared memory accesses. In a distributed setting, however, every memory access incurs a latency of two message delays, whereas storage space is typically abundant. Therefore, a practical distributed construction should focus on simplicity and reducing communication costs, even at the cost of storing timestamps along with values[1]. This is precisely the approach we take in this paper.

We present a complete wait-free construction of a single-writer regular register. Our construction is simple, efficient, and feasible in distributed storage environments. We give a modular construction by introducing a novel building block, called a 1-*regular* register. A 1-*regular* register is regular as long as a read operation overlaps at most one write operation. That is, every read operation that overlaps at most one write returns either the register's value before read is invoked

[1]Like previous constructions in this setting [10, 7, 11, 1, 4], we use unbounded timestamps. It is possible to modify our algorithms to employ bounded timestamps by using garbage collection techniques. However, the discussion of such techniques is beyond the scope of this paper.

or the one written by the overlapping write. A read that overlaps more than one write may return an arbitrary value. In Section 4, we give an efficient implementation of a wait-free 1-regular register from components prone to Byzantine faults. Read and write operations are both emulated in a single round, and the space required in each base object is twice that of safe register constructions in this model [10]. Then, in Section 5, we give a simple and efficient implementation of a wait-free single-reader single-writer regular register using 1-regular ones.

2 Related Work

Most wait-free Byzantine-fault-tolerant register constructions for distributed settings provide only safe semantics [10, 6, 1]. Others achieve stronger semantics at the cost of weaker (non-wait-free) termination guarantees [11, 2, 1]. PASIS [4] achieves atomic semantics, but does not allow overwriting of objects, and instead stores, as part of an object's state, all its previous versions.

Wait-free constructions of registers with strong semantics (regular and atomic) from weaker ones have been an actively researched area for several decades [12, 8, 16, 17, 5]. Such constructions are typically fairly elaborate. Our work benefits from several techniques and ideas introduced by Peterson [12] and Tromp [16]. These two papers construct atomic registers from safe bit tracks and additional control bits, and optimize the number of shared memory bits as well as the number of shared memory accesses employed. For example, Tromp constructs an atomic register from 4 safe registers (holding values, but not timestamps), and 8 safe bits; his read operation makes up to 5 memory accesses (in 4 rounds), and his write requires 3. In contrast, our read emulation always takes 3 memory access rounds, and the write emulation always takes 2. Although these two algorithms do not solve the problem we address in this paper (implementing a reliable shared register from Byzantine components), they could be used, in lieu of the algorithm we give in Section 5, on top of known constructions of safe registers in Byzantine settings, e.g., [10], which are similar to the 1-regular register construction we give in Section 4. However, our overall solution is simpler, and requires fewer memory access rounds as noted above, partly due to our use of timestamps, which is a favorable tradeoff in distributed storage settings.

Our 1-regular register differs from the pseudo-regular register of Pierce and Alvisi [13] in that the latter is allowed to return a special value $\perp$ if a read overlaps *any* number of writes, but may never return incorrect values, whereas a 1-regular register must ensure regularity as long as one write overlaps the read, and is otherwise allowed to return arbitrary values.

3 The System Model

We consider an asynchronous shared memory system consisting of a collection of processes interacting with a finite collection of n objects. Up to $t < n/4$ of the objects may suffer NR-Arbitrary failures [6], i.e., may fail to respond to an invocation, or may respond with an arbitrary value. Any number of the processes may fail by stopping. Due to space limitations, we do not detail the formal system model; further details may be found in [8, 6, 1].

A *register* is an object type with a value domain *Vals*, an initial value $v_0 \in Vals$, and two invocations: *read*, whose response is $v \in Vals$, and *write*(v), $v \in Vals$, whose response is *ack*. A read/write register is single-reader single-writer (SRSW) if only one process can write to it and only one can read from it; a register is multi-reader single-writer (MRSW) if multiple processes can read it.

4 Wait-free t-Resilient 1-Regular Register Construction

In order to distinguish between the emulated register's interface and that of the underlying base registers, we henceforth denote the emulated read (resp. write) operation as READ (resp. WRITE).

The protocol uses an underlying layer that invokes operations on different base objects in separate threads. The notation **invoke** $write(X_i, v)$, (resp. **invoke** $x[i] \leftarrow read(X_i)$) indicates that a write(v) operation on register X_i (resp. a read of register X_i whose response will be stored in local variable $x[i]$) is invoked in a separate thread by the underlying layer. Since some of the base objects may be non-responsive, a WRITE or READ operation may return while there are still pending invocations to base objects that did not respond. The underlying layer keeps track of which invocations are pending after the high-level WRITE or READ returns, and uses this information in order to ensure well-formedness, i.e., that a process does not invoke an operation on an object as long as there are pending invocations of the same process on the same object. If a new operation (read or write) is invoked on a base object while a previous operation on the same object is pending, the underlying layer does not invoke the new operation, and instead denotes that it is enabled. Only the most recent enabled operation is stored. An enabled operation is invoked if and when a pending one returns. If a new invocation is enabled on some object when an older pending invocation responds, the underlying layer discards this response and does not deliver it to the high-level WRITE or READ. (See, e.g., [6, 1], for detailed implementations of such layers.)

Figure 1 presents an implementation of a wait-free MRSW 1-regular register from $n > 4t$ wait-free MRSW regular base registers, up to t of which can incur NR-arbitrary faults. Each of the n base registers, X_i, is a cyclic buffer holding two timestamp-value pairs, $X_i[0]$ and $X_i[1]$. Each WRITE(v) operation chooses an increasing timestamp ts and over-writes one of the components, with the pair $\langle ts, v\rangle$, alternating between $X_i[0]$ and $X_i[1]$, while the previous value and timestamp are re-written to the other component. Thus, every READ operation that overlaps at most one WRITE samples either $X_i[0]$ or $X_i[1]$ while its value is not being changed (it may be overwritten, but with a value matching the previous one). Note that local variables are static, i.e., they are not reset in each invocation of WRITE.

The READ emulation reads from at least $n - t$ base registers into the array $x[1 \dots n]$, where element i stores the pair corresponding to the values read from $X_i[0]$ and $X_i[1]$ (lines 2–3). The *candidates* for returning are the read timestamp-value pairs. In line 4, the set of candidates from each component j is set to include only values that appear in the jth component of at least $t + 1$ elements of x, which ensures that they were read from at least one correct base object. Returning the highest timestamped candidate in these sets (line 5) ensures 1-regularity. If there is no such value, then there must be at least two WRITEs overlapping the READ, and it may return an arbitrary value (line 6).

We now formally prove the algorithm's correctness.

Theorem 1. *The algorithm in Figure 1 implements a wait-free* 1*-regular register from* $n > 4t$ *regular base registers, at most* t *of which can incur NR-arbitrary faults.*

Proof. Wait-freedom is immediate, since no process waits for more then $n - t$ responses, and at most t objects can be non-responsive. We now show 1-regularity.

Consider a READ invocation R_1. Let c be the timestamp-value pair written by the last WRITE invocation that completes before R_1 is invoked, or $\langle 0, v_0\rangle$ if none was invoked; c is written to some

Types: *TSVals*= $Integers \times Vals$, with selectors *ts, val*
Shared objects: regular registers $X_i[2] \in$ *TSVals*, initially $X_i[0] = X_i[1] = \langle 0, v_0 \rangle$

WRITE **Emulation**

Local variables (writer):
  $x[2] \in$ *TSVals*, initially $x[0] = x[1] = \langle 0, v_0 \rangle$
  $turn \in \{0, 1\}$, initially 0
  $ts \in$ *Integers*, initially 0

WRITE(v):
  $ts \leftarrow ts + 1$
  $x[turn] \leftarrow \langle ts, v \rangle$
  for $1 \le i \le n$, **invoke** write(X_i, x)
  wait for $n - t$ responses
  $turn \leftarrow 1 - turn$
  return *ack*

READ **Emulation**

Local variables (reader):
  $x[1 \ldots n][2] \in$ *TSVals* $\cup \{\perp\}$
  $C[2]$ *set of TSVals*

READ:
1: **for** $1 \le i \le n,\ j = 0, 1,\ x[i][j] \leftarrow \perp$
2: **for** $1 \le i \le n$, **invoke** $x[i] \leftarrow$ read(X_i)
3: **wait** for $n - t$ responses
4: **for** $j \in \{0, 1\},\ C[j] \leftarrow \{c \in TSVals : |\{i : x[i][j] = c\}| \ge t + 1\}$
5: **if** $(C[0] \cup C[1] \ne \emptyset)$ **then return** $c.val$: $c.ts = max\{c'.ts : c' \in C[0] \cup C[1]\}$
6: **return** an arbitrary value in $Vals$

Figure 1: Wait-free t-resilient 1-regular MRSW register emulation.

component $j \in \{0, 1\}$. (If $c = \langle 0, v_0 \rangle$, let $j = 1$.) There are three cases to consider. (1) If no WRITE invocation overlaps R_1, then throughout R_1, c appears in at least $n - 2t \ge 2t + 1$ correct base objects' jth component, and hence is read at least $t + 1$ times by R_1 and is included in $C[j]$. Moreover, no higher-timestamped values appear in correct objects, hence no higher timestamped value appears in either $C[0]$ or $C[1]$. Therefore, c is returned in line 5. (2) Assume exactly one WRITE invocation, W_1 overlaps R_1. Since the writer alternates between components, this invocation does *not* change the value of the jth component. Therefore, as before, c is included in $C[j]$, and has the highest timestamp in $C[j]$. If c is not returned in line 5, then the return value is a correct higher-timestamped candidate from $C[1-j]$, which must be the one written by W_1. Hence, in both of these cases, regularity is ensured. (3) If more than one WRITE overlaps R_1, the READ is allowed to return an arbitrary value. □

Shared objects:
SRSW safe register $R \in \{0, 1\}$, initially 0, writable by reader, readable by writer
SRSW 1-regular registers $Y[2] \in$ *TSVals*, initially $\langle 0, v_0 \rangle$, writable by writer, readable by reader

WRITE **Emulation**

Local variables (writer):
  $ts \in Integers$, initially 0
  $r \in \{0, 1\}$

WRITE(v):
  $ts \leftarrow ts + 1$
  $r \leftarrow \text{read}(R)$
  $\text{write}(Y[1 - r], \langle ts, v \rangle)$
  return *ack*

READ **Emulation**

Local variables (reader):
  $r \in \{0, 1\}$, initially 0
  $x[2] \in TSVals$

READ:
  $x[r] \leftarrow \text{read}(Y[r])$
  $r \leftarrow 1 - r$
  $\text{write}(R, r)$
  $x[r] \leftarrow \text{read}(Y[r])$
  return $x.val : x.ts = max\{x[i].ts : i \in \{0, 1\}\}$

Figure 2: The SRSW wait-free regular register construction.

5 Wait-free Regular Register Construction

We now present a very simple construction of an SRSW regular register using two 1-regular SRSW registers, $Y[2]$, and one safe bit, R. (A multi-reader regular register can be constructed using m copies of this register. Note that in a distributed storage setting, the m copies can be accessed by the writer in parallel, whereas each reader accesses a single copy.) The algorithm appears in Figure 2. As before, we denote the emulated register's operations WRITE and READ.

The algorithm exploits 1-regular semantics by ensuring that at most one write invocation overlaps each read invocation to either $Y[0]$ or $Y[1]$. This is controlled using the shared safe bit R, which indicates which of the two 1-regular registers is potentially being read by the reader. The writer reads R, and then writes the new value to $Y[1 - R]$ along with a monotonically increasing timestamp. The reader reads both $Y[1]$ and $Y[0]$, and returns the higher-timestamped value.

The following lemma shows that at most one write invocation overlaps each read invocation to either $Y[0]$ or $Y[1]$.

Lemma 1. *Each read of a* 1*-regular base register* $Y[i]$ *overlaps at most one write operation.*

Proof. Consider a basic read operation, r_1, on register $Y[i]$. Consider the first time t at which a write operation on $Y[i]$ completes after r_1 starts. Now consider a WRITE invocation, W_1, invoked after t and before r_1 completes. When W_1 reads R, it is not being written (because the reader is currently reading $Y[i]$), and therefore, by safety, returns i. Therefore, W_1 writes to $Y[1-i]$. $\square$

We are now ready to prove our main theorem:

Theorem 2. *The algorithm in Figure 2 implements a SRSW wait-free regular register from two wait-free* 1*-regular SRSW registers and one safe register.*

Proof. Wait-freedom is immediate, by wait-freedom of the base objects. We now prove regularity. Consider a READ invocation R_1. By Lemma 1, each read operation invoked by R_1 on one of the base registers $Y[\cdot]$ overlaps at most one write operation to the same base register. Hence, by 1-regularity, each of the base registers $Y[\cdot]$ responds with a correct value. Consider the register's value before R_1 is invoked (either the value written by the last WRITE that completes before R_1 is invoked, or $\langle 0, v_0 \rangle$ if none was invoked). This value is written to some $Y[i]$. Therefore, when R_1 reads $Y[i]$, it obtains either this value or a later (higher-timestamped) one. The fact that READ returns the higher-timestamped value it reads ensures regularity. $\square$

6 Conclusions

Constructions of strong (regular and atomic) registers from weaker ones are typically complex. Previous implementations of wait-free registers from Byzantine shared-memory either store the register's entire history of values as part of the register's state [4], or provide only safe semantics [10, 6, 1], which, by themselves, are too weak to be of use to applications.

In this paper, we presented a new abstraction called a 1-regular register. On the one hand, this abstraction is sufficiently strong to support a very simple, easy to follow, and efficient implementation of a regular register. On the other hand, 1-regular registers are easily implementable in distributed Byzantine fault-prone settings, virtually as efficiently as safe registers: our 1-regular register construction emulates each read or write operation in a single round of invocations to base objects, without storing the register's entire history; it stores only the last two values written to the register, whereas safe register implementations [10] store one.

There may well be additional settings where implementing 1-regular registers will be (almost) as easy and efficient as implementing safe ones, and therefore, this abstraction as well as the construction of regular registers from 1-regular ones may have broader applicability.

Using 1-regular registers, we constructed a regular register. It is easy to extend our construction to provide atomic semantics. The reader simply needs to recall the latest returned value and its timestamp. If a later read returns a smaller timestamp, then the reader returns the stored value.

References

[1] I. Abraham, G. Chockler, I. Keidar, and D. Malkhi. Byzantine Disk Paxos: Optimal resilience with byzantine shared memory. In *23rd ACM Symposium on Principles of Distributed Computing (PODC)*, 2004. Full version to appear in Distributed Computing.

[2] H. Attiya and A. Bar-Or. Sharing memory with semi-byzantine clients and faulty storage servers. In *The 22nd Symposium on Reliable Distributed Systems (SRDS)*, 2003.

[3] E. Gafni and L. Lamport. Disk paxos. *Distributed Computing*, 16(1):1–20, 2003.

[4] G. R. Goodson, J. J. Wylie, G. R. Ganger, and M. K. Reiter. Efficient byzantine-tolerant erasure-coded storage. In *International Conference on Dependable Systems and Networks (DSN)*, June 2004.

[5] S. Haldar and P. Vitanyi. Bounded concurrent timestamp systems using vector clocks. *J. ACM*, 49(1):101–126, 2002.

[6] P. Jayanti, T. Chandra, , and S. Toueg. Fault-tolerant wait-free shared objects. *J. ACM*, 45(3):451–500, 1998.

[7] S. Lakshmanan, M. Ahamad, and H. Venkateswaran. Responsive security for stored data. In *Proceedings of the International Conference on Distributed Computing Systems (ICDCS)*, 2003.

[8] L. Lamport. On interprocess communication – Part II: Algorithms. *Distributed Computing*, 1(2):86–101, 1986.

[9] S. Lin, Q. Lian, M. Chen, and Z. Zhang. A practical distributed mutual exclusion protocol in dynamic peer-to-peer systems. In *3rd International Workshop on Peer-to-Peer Systems (IPTPS'04)*, 2004.

[10] D. Malkhi and M. Reiter. Byzantine quorum systems. *Distributed Computing*, 11(4):203–213, 1998.

[11] J.-P. Martin, L. Alvisi, and M. Dahlin. Minimal byzantine storage. In *Proceedings of the 16th International Symposium on Distributed Computing (DISC)*, October 2002.

[12] G. L. Peterson. Concurrent reading while writing. *ACM Transactions on Programming Languages and Systems*, 5(1):46–55, 1983.

[13] E. Pierce and L. Alvisi. A framework for semantic reasoning about byzantine quorum systems. In *Brief announcement in Twentieth ACM Symposium on Principles of Distributed Computing (PODC)*, 2001.

[14] R. Rodrigues and B. Liskov. Rosebud: A Scalable Byzantine-Fault-Tolerant Storage Architecture. Technical Report MIT-LCS-TR-932, MIT Laboratory for Computer Science, 2004.

[15] C. Shao, E. Pierce, and J. Welch. Multi-writer consistency conditions for shared memory objects. In *International Symposium on Distributed Computing (DISC)*. Springer-Verlag, 2003.

[16] J. Tromp. How to construct an atomic variable. In *LNCS 392, Proc. 3rd International Workshop On Distributed Algorithms*, pages 292–302. Springer-Verlag, 1989.

[17] K. Vidyasankar. Concurrent reading while writing revisited. *Distributed Computing*, 4:81–85, 1990.